# ROZDZIAŁ DZIEWIĘTNASTY: **PROCESY, WSPÓŁPROGRAMY I WSPÓŁBIEŻNOŚĆ**

Kiedy większość ludzi mówi wielozadaniowość , zazwyczaj w znaczeniu zdolności do uruchamiania kilku różnych aplikacji w tym samym czasie na jednej maszynie. Dana struktura oryginalnych chipów 80x86 i projektowania oprogramowania MS-DOS jest trudna do osiągnięcia, kiedy jest uruchomiony DOS. Przypatrzymy się jak Microsoft używa Windows do wielozadaniowości.

Po problemach dużych firm takich jak Microsoft z działaniem wielozadaniowości, możemy sądzić,że jest to bardzo trudna rzecz do zarządzania. Jednak nie jest to prawda, Microsoft ma problemy próbując uczynić różne aplikacje, które są nieświadome innych harmonijnych prac. Szczerze mówiąc nie mają one istniejących aplikacji DOS działających dobrze w ramach wielozadaniowości. Zamiast tego, działają na rzecz rozwijania nowych programów, które działają dobrze pod Windowsem.

Wielozadaniowość nie jest sprawą błahą, ale nie jest tak trudna, kiedy piszemy aplikację z wielozadaniowością. Możemy nawet pisać programy, które są wielozadaniowe pod DOS'em, jeśli tylko zabezpieczymy się kilkoma środkami ostrożności. W tym rozdziale będziemy omawiali koncepcję procesów DOS, współprogramy i ogólnie procesy.

## 19.1 PROCESY DOS

Chociaż MS-DOS jest jednozadaniowym systemem operacyjnym, nie znaczy to, że może być tylko jeden program w tym czasie w pamięci. Faktycznie głównym celem poprzedniego rozdziału było opisanie jak ulokować dwa lub więcej działających w pamięci w tum samym czasie. Jednakże, nawet, jeśli zignorujemy początkowo TSR'y, możemy jeszcze załadować kilka programów do pamięci w jednym czasie pod DOS'em. Jedyną pułapką jest to, że DOS dostarcza zdolności im do działania tylko jednego w czasie w bardzo określony sposób. Chyba, że procesy współpracują, ich profil wykonania podąża za bardzo ścisłym wzorcem.

### 19.1.1 PROCESY POTOMNE W DOS

Kiedy aplikacja DOS jest uruchomiona, można załadować i wykonywać jakiś inny program używając funkcji EXEC DOS (zobacz „MS-DOS, PC-BIOS i I/O Plików"). W normalnych warunkach, kiedy aplikacja (macierzysta) uruchamia drugi program (potomny), proces potomny wykonuje się do zakończenia i potem wraca do macierzystej. Jest to bardzo podobne do wywołania procedury, z wyjątkiem tego, że jest trochę trudniejsze przekazanie parametrów między nimi.

MS-DOS dostarcza kilka funkcji, jakie możemy zastosować do ładowania i wykonania kod programu, kończymy proces i uzyskujemy stan wyjścia dla procesu. Poniżej mamy tablicę wielu z tych operacji.

| Funkcja (AH) | Parametry wejściowe | Parametry wyjściowe | Opis |
|---|---|---|---|
| 4Bh | al – 0<br>ds:dx – wskaźnik do nazwy programu<br>es:bx – wskaźnik do struktury LOADEXEC | Ax – kod błędu, jeśli ustawione przeniesienie | Ładuje i wykonuje program |

| | | | |
|---|---|---|---|
| 4Bh | al. –1<br>ds:dx – wskaźnik do nazwy programu<br>es:bx – wskaźnik do struktury LOAD | ax – kod błędu jeśli ustawione | Ładuje program |
| 4Bh | al.- 3<br>ds:dx – wskaźnik do nazwy programu<br>es:bx – wskaźnik do struktury OVERLAY | ax – kod błędu jeśli ustawione przeniesienie | Ładownie nakładki |
| 4Ch | al. – proces zwracania kodu | | Wykonanie zakończenia |
| 4Dh | | al – zwracana wartość<br>ah – metoda zakończenia | |

Tabela 67: Funkcje DOS zorientowane znakowo

---

19.1.1.1 ZAŁADUJ I WYKONAJ

Funkcja „załaduj i wykonaj" wymaga dwóch parametrów. Pierwszy w ds.:dx, jest wskaźnikiem do ciągu zakończonego zerem zawierającego ścieżkę dostępu programu do wykonania. To musi być plik „.COM" lub „.EXE" a ciag musi zawierać rozszerzenie nazwy programu. Drugi parametr, w es:bx, jest wskaźnikiem do struktury danych LOADEXEC. Ta struktura danych przybiera postać:

```
LOADEXEC    struct
EnvPtr      word    ?       ;wskaźnik do obszaru środowiska
CmdLinePtr  dword   ?       ;wskaźnik do lini poleceń
FCB1        dword   ?       ;wskaźnik do domyślnego FCB1
FCB2        dword   ?       ; wskaźnik do domyślnego FCB2
LOADEXEC    ends    ?
```

EnvPtr jest adresem segmentu bloku środowiska DOS stworzonego dla nowej aplikacji. Jeśli to pole zawiera zero, DOS tworzy kopię aktualnego bloku środowiska procesu dla procesu potomnego. Jeśli program, jaki jest uruchomiony nie uzyskuje dostępu do bloku środowiska, możemy zachować kilkaset bajtów do kilku kilobajtów przez wskazanie wskaźnika pola środowiska dla ciągu czterech zer.

Pole CmdLinePtr zawiera adres lini poleceń dostarczonych do programu. DOS będzie kopiował linię poleceń do offsetu 80h w nowym PSP stworzonym dla procesu potomnego. Poprawna linia poleceń składa się z bajtu zawierającego licznik znaku, mniej ważną przestrzeń, znak należący do linii poleceń i kończący znak powrotu karetki (0Dh). Pierwszy bajt powinien zawierać długość znaków ASCII w linii poleceń, nie wliczając powrotu karetki. Jeśli ten bajt zawiera zero, wtedy drugi bajt linii poleceń powinien być powrotem karetki, nie przestrzeni. Przykład:

```
MyCmdLine   byte    12, „file1, file2", cr
```

Pola FCB1 I FCB2 muszą wskazywać dwa domyślne bloki kontrolne pliku dla tego programu. FCB'y stały się przestarzałe wraz z DOS'em 2.0, ale Microsoft zachowała FCB'y dla kompatybilności. Dla większości programów możemy wskazać oba te pole w następującym ciągu bajtów:

```
DfltFCB     byte·3, ' „, 0, 0 ,0 ,0
```

Funkcja załaduj i wykonaj będzie zakończona niepowodzeniem, jeśli jest niewystarczająca ilość pamięci dla załadowania procesu potomnego. Kiedy tworzymy plik"..EXE' używając MASM'a, tworzy on plik wykonywalny, który przechwytuje całą dostępną pamięć, domyślnie. Dlatego też, jeśli nie będzie dostępnej pamięci dla procesu potomnego DOS zawsze zwróci błąd. Dlatego też musimy ustawić alokację pamięci dla procesu macierzystego zanim spróbujemy uruchomić proces potomny. Jak to zrobić opisuje sekcja „Programu półrezydentne"

Są inne możliwe błędy. Na przykład, DOS może nie być zdolny zlokalizować nazwy programu, jaka określiliśmy ciągiem zakończonym zerem. Lub być może, jest zbyt dużo otwartych plików i DOS nie ma wolnego

dostępnego bufora dla I/O pliku. Jeśli wystąpi błąd, DOS zwróci ustawioną flagę przeniesienia i właściwy kod błędu w rejestrze ax. Następujący przykład wykonuje program „COMMAND.COM", pozwalając użytkownikowi wykonywać polecenia DOS z wnętrza naszej aplikacji. Kiedy użytkownik wpisuje, „exit" w lini poleceń DOS, DOS zwróci sterownie do naszego programu.

```
; RUNDOS.ASM- Demonstruje jak wywołać kopię COMMAND.COM, interpretera lini poleceń DOS z naszego programu

                include         stdlib.a
                includelib      stdlib.lib

dseg            segment         para public  'data'

; Struktura EXEC MS-DOS

ExecStruct      word    0                       ;używamy bloku środowiska macierzystego
                dword   CmdLine                         ; dla parametrów lini poleceń
                dword   DfltFCB
                dword   DfltFCB

DfltFCB         byte    3, „ „ , 0, 0 ,0 ,0
CmdLine                 byte    0, 0dh                  ;linia poleceń dla programu
PgmName         dword   filename                ;wskazuje nazwę programu

filename        byte    „C:\command.com", 0

dseg            ends

cseg            segemnt para  public  'code'
                assume  cs:cseg, ds:dseg

Main            proc
                mov     ax, dseg                ;pobranie wskaźnika do segmentu zmiennych
                mov     ds., ax

                meminit                         ;start menadżera pamięci

; Okay, zbudowaliśmy strukturę wykonawczą MS-DOS i potrzebną linię poleceń, teraz zobaczmy uruchamianie programu
; Pierwszym krokiem jest zwolnienie całej pamięci, której ten program nie używa. To będzie wszystko od zzzzzzseg.
;
; Notka: podobnie jak w poprzednich przykładach w innych rozdziałach, jest okay wywoływać  podprogramy Biblioteki
; Standardowej w tym programie po zwolnieniu pamięci. Różnica tu jest to, że podprogramy Biblioteki Standardowej są
; ładowane wcześniej w pamięci i nie możemy zwolnić pamięci , która jest tam usytuowana.

                mov     ah, 62h                 ;pobranie wartości naszego PSP
                int     21h
                mov     es, bx
                mov     ax, zzzzzzseg           ;obliczenie rozmiaru rezydentnego kodu uruchomieniowego
                sub     ax, bx
                mov     bx, ax
                mov     ah, 4ah                 ; zwolnienie nie używanej pamięci
                int     21h
```

```
; powiedzenie użytkownikowi co się dzieje:

                print
                byte    cr, lf
                byte    „RUNDOS – Wykonanie kopii command.com”, cr, lf
                byte    „Wpisanie ‘EXIT’ zwraca sterowanie do RUN.ASM”, cr, lf
                byte    0

; Ostrzeżenie! Żadnej funkcji Biblioteki Standardowej po tym punkcie. Zwolniliśmy pamięć, którą one zajmowały. Więc
; ładując program zlikwidujemy kod Biblioteki Standardowej

                mov     bx, seg  ExecStruct
                mov     es, bx
                mov     bx, offset  ExecStruct                  ;wskaźnik do rekordu programu
                lds     dx, PgmName
                mov     ax, 4b00h                               ;exec  pgm
                int     21h
; w MS-DOS 6.0 poniższy kod nie jest wymagany. Ale w starszych wersjach MS-DOS, stos jest rujnowany od tego punktu.
; będzie bezpieczniej, jeśli zresetujemy wskaźnik stosu  do przyzwoitego miejsca w pamięci.
;
;Zauważmy, że kod ten zachowuje flagę przeniesienia a wartość w rejestrze AX, więc możemy przetestować warunki błędu dla
; DOS kiedy wykonaliśmy poprawiani stosu

                mov     bx, sseg
                mov     ss, ax
                mov     sp, offset  EndStk
                mov     bx, seg dseg
                mov     ds, bx

;Test dla błędu DOS:
                jnc     GoodCommand
                print
                byte    „DOS error #”, 0
                puti
                print
                byte    „   kiedy próbujesz uruchomić COMMAND.COM”, cr, lf
                byte    0
                jmp     Quit


;Wydruk wiadomości końcowej

GoodCommand:    print
                byte    „Witamy ponownie w RUNDOS. Mam nadzieję, że bawiłeś się dobrze”, cr, lf
                byte    „Teraz wracamy do COMMAND.COM w wersji MS-DOS”
                byte    cr, lf, lf,0

; Zwrócenie sterowania do MS-DOS

Quit:           ExitPgm
Main            endp
cseg            ends
```

```
sseg            segment para  stack 'stack'
                dw      128   dup (0)
                ends

zzzzzzseg       segment para  public  'zzzzzzseg'
Heap            db      200h  dup (?)
Zzzzzzseg       ends
                end     Main
```

19.1.1.2 ŁADOWANIE PROGRAMU

Funkcja ładowania i wykonywania daje procesowi macierzystemu bardzo małą kontrolę nad procesem potomnym. Chyba, że potomek komunikuje się z procesem macierzystym poprzez przerwania kontrolowane lub przerwania, DOS zawiesza proces macierzysty dopóki nie zakończy się potomek. W wielu przypadkach pogram macierzysty może chcieć załadować kod aplikacji a potem wykonać jakąś dodatkowe działanie zanim przejmie to proces potomny. Programy półrezydentne, pojawiające się w poprzedni rozdziale, dostarczają dobrych przykładów. Funkcja DOS'a „ładowania programu" dostarcza tej zdolności; będzie ładować program z dysku i zwraca sterowanie z powrotem do procesu macierzystego. Proces macierzysty może robić co konieczne jeśli jest to właściwe przed przekazaniem sterowania do procesu potomnego.

Funkcja ładowania programu wymaga parametrów, które są bardzo podobne do funkcji ładowania i wykonania. Faktycznie , jedyną różnica jest użycie struktury LOAD zamiast LOADEXEC, a nawet te struktury są bardzo podobne do siebie. Struktura danych LOAD dołącza dwa nowe pola nie obecne w strukturze LAODEXEC:

```
LOAD            struct
EnvPtr          word    ?               ;wskaźnik do obszaru środowiska
CmdLinePtr      dword   ?               ;wskaźnik do lini poleceń
FCB1            dword   ?               ;wskaźnik do domyślnego FCB1
FCB2            dword   ?               ;wskaźnik do domyślnego FCB2
SSSP            dword   ?               ; wartość SS:SP dla procesu potomnego
CSIP            dword   ?               ;Inicjalizacja programu z punktu startowego
LOAD            ends
```

Polecenie LOAD jest użyteczna dla wielu celów. Oczywiście, funkcja ta dostarcza podstawowego narzędzia dla tworzenia programów półrezydentnych'; jednakże, jest również całkiem użyteczne przy odzyskiwaniu dodatkowego błędu,przeadresowywania aplikacji I/O i ładowanie kilku wykonywalnych procesów do pamięci dla współbieżnego wykonania.

Po załadowaniu programu poleceniem load DOS'a, możemy uzyskać adres PSP dla tego programu przez wydanie przez DOS funkcji pobrania adresu PSP (zobacz „MS-DOS, PC-BIOS i pliki I/O"). Pozwoliłoby to procesowi macierzystemu na zmodyfikowanie jakiejś wartości pojawiającej się w PSP procesu potomnego przed jego wykonaniem. DOS przechowuje adres zakończenia dla procedury w PSP. Jeśli nie zmieniasz tej lokacji, program będzie wracał do pierwszej instrukcji poza instrukcja int 21h dla załadowanej funkcji. Dlatego też przed rzeczywistym przekazaniem sterowania do aplikacji użytkownika, powinniśmy zmienić ten adres zakończenia.

19.1.1.3 ŁADOWANIE NAKŁADEK

Wiele programów zawiera bloki kodu, które są niezależne jeden od drugiego, to znaczy, jeśli podprogram w jednym bloku kodu się wykonuje, program ni będzie wywoływał podprogramów w innym bloku kodu. Na przykład, nowoczesne gry mogą zawierać jakiś kod inicjalizujący obszar „publicznego udostępnienia" gdzie użytkownik wybiera pewne opcje, „obszar działania" gdzie użytkownik gra w grę i „obszar wypytywania", który sprawdza działania gracza. Kiedy uruchomimy maszynę w 640 k MS-DOS ,cały ten kod może nie zmieścić się w dostępnej pamięci w tym samym czasie. Dla pokonania tego ograniczenia pamięci, wiele dużych programów używa nakładek. Nakładka jest części kodu programu, która dzieli pamięć dla jego kodu z innymi modułami kodu. Funkcja DOS'a ładowania nakładek dostarcza wsparcia dla dużych programów, które muszą używać nakładek.

Podobnie jak funkcje ładowania i ładowania / wykonania, ładowanie nakładek oczekuje wskaźnika do kodu ścieżki dostępu pliku w parze rejestrów ds:dx i adresu struktury danych w parze rejestrów es:bx. Struktura danych nakładki ma następujący format

```
overlay     struct
StartSeg    word    ?
Relocfactor word    0
Overlay     ends
```

Pole StartSeg zawiera adres segmentu gdzie chcemy, aby DOS załadował program. Pole RelocFactor zawiera stałą przemieszczenia. Wartość ta powinna być zerem., chyba , że chcemy aby offset startowy segmentu był inny niż zero.

### 19.1.1.4 ZAKOŃCZENIE PROCESU

Funkcja zakończenia procesu jest niczym nowym dla nas teraz, używamy tej funkcji ciągle i ciągle, jeśli napisaliśmy jakiś program asemblerowy i uruchamiamy go pod DOS'em (makro Biblioteki Standardowej ExitPgm wykonuje to polecenie) W sekcji tej zobaczymy dokładnie jak pracuje funkcja kończenia procesu. przede wszystkim, funkcja zakończenia procesu daje nam umiejętność przekazywania pojedynczego bajtu kodu zakończenia z powrotem do procesu macierzystego. Jakąkolwiek wartość przekazujemy w al do zakończenia staje się kodem zwrotnym lub zakończenia. Proces macierzysty może testować wartość używając funkcji Get Child Process Return Value (zobacz następną sekcję). Możemy również przetestować wartość zwracaną w pliku wsadowym DOS'a używając instrukcji „if errorlevel"

Polecenie zakończenia procesu robi co następuje:

- Opróżnia bufory plików i zamyka pliki
- Przywrócenie adresu zakończenia (int 22h) z offsetu 0Ah w PSP (jest to adres powrotu procesu)
- Przywraca adres programu obsługi Break (int 23h) z offsetu 0Eh w PSP
- Przywraca adres programu obsługi błędu krytycznego (int 24h) z offsetu 12h w PSP
- Dealokuje pamięć przechowywaną przez proces

Chyba ,że rzeczywiście wiemy co robimy, nie powinniśmy zmieniać wartości pod offsetami 0Ah, 0Eh lub 12h w PSP. Przez zrobienie tego możemy stworzyć wewnętrznie sprzeczny system kiedy nasz program się kończy.

### 19.1.1.5 UZYSKANIE KODU POWOROTU PROCESU POTOMNEGO

Proces macierzysty może uzyskać kod powrotny z procesu potomnego poprzez wywołanie funkcji Get Child Process Return Code. Ta funkcja zwraca wartość w rejestrze al w punkcie zakończenia plus informację, która mówi nam jak zakończył się proces potomny.

Ta funkcja (ah =4Dh) zwraca kod zakończenia w rejestrze al. Również zwraca powód zakończenia w rejestrze ah Rejestr ah będzie zawierał jedną z następujących wartości:

| Wartość w AH | Powód zakończenia |
|---|---|
| 0 | Zakończenie normalne (int 21h, ah = 4Ch) |
| 1 | Zakończenie przez ctrl+C |
| 2 | Zakończenie przez błąd krytyczny |
| 3 | Zakończenie TSR (int 21h, ah= 31h) |

Tablice 68: Powody zakończenia

Kod zakończenia pojawiający się w al. jest poprawny tylko dla zakończeń normalnego i TSR

Zauważmy, że możemy tylko raz wywołać ten podprogram po zakończeniu procesu potomnego. MS-DOS zwraca wartość bez znaczenia w AX po pierwszym takim wywołaniu. Podobnie , jeśli użyjemy tej funkcji bez uruchamiania procesu potomnego, wyniki jakie uzyskamy będą bez sensu .DOS nie wraca jeśli to zrobimy.

## 19.1.2 OBSŁUGA WYJĄTKÓW W DOS: OBSŁUGA BREAK

Jeśli kiedykolwiek użytkownik naciśnie klawisze ctrl + C lub ctrl+ Braek MS-DOS może przerwać taką sekwencję klawiszy i wykonać instrukcję int 23h. MS-DOS dostarcza domyślnego podprogramu obsługi break, który kończy program. Jednakże, dobrze napisany program generalnie zamienia domyślny podprogram obsługi break z jednym ze swoich, więc może przechwycić sekwencję klawiszy ctrl + C lub ctrl + Break i wyłączyć program w uporządkowany sposób.

Kiedy DOS kończy program z powodu przerwania break, opróżnia bufor plików, zamyka wszystkie otwarte pliki, zwalnia pamięć należącą do aplikacji i wszystkie normalne rzeczy przy zamykaniu programu. Jednakże, nie przywraca żadnego wektora przerwań (inaczej niż przerwania 23h i 24h) . Jeśli nasz kod zamieniał jakieś wektory przerwań, zwłaszcza wektory przerwań sprzętowych, wtedy wektory te będą jeszcze wskazywały na programy obsługi przerwań naszego programu po zakończeniu przez DOS naszego programu. Wtedy prawdopodobnie wystąpi krach systemu, kiedy DOS załaduje nowy program na szczycie naszego kodu. Dlatego też, powinniśmy napisać program obsługi break, aby nasza aplikacja mogła zamknąć się sama w uporządkowany sposób jeśli użytkownik naciśnie ctrl + C lub ctrl + break.

Najłatwiejszy, i być może najbardziej uniwersalny program obsługi break składa się z pojedynczej instrukcji iret .Jeśli wskażemy wektor przerwania int 23h przy instrukcji iret. MS-DOS po prostu zignoruje każde naciśnięcie klawiszy ctrl+C lub ctrl + Break. Jest to bardzo użyteczne dla wyłączania programu obsługi break podczas sekcji krytycznych kodu, których nie chcemy by użytkownik przerywał.

Z drugiej strony, po prostu wyłączamy program obsługi ctrl + C lub ctrl + break w całym programie jeśli nie jest satysfakcjonujący. Jeśli z tego samego powodu użytkownik chce przerwać program, naciśnięcie ctrl + C lub ctrl + break jest prawdopodobnie tym co spróbuje zrobić. Jeśli nasz program na to nie zezwala, użytkownik może posunąć się do czegoś bardziej drastycznego, jak ctrl + alt + delete, dla zresteowania maszyny. To z pewnością zrujnuje jakiś otwarte pliki i może spowodować inne problemy (oczywiście, oczywiście nie musimy martwić się o przywracanie wektorów przerwań!)

Aktualizowanie naszego własnego programu obsługi break jest łatwe – przechowujemy adres naszego podprogramu obsługi break w wektorze przerwań 23h. Nie musimy nawet zapisywać starej wartości. DOS robi to automatycznie (przechowa wartość wektora pod offsetem 0Eh w PSP). Potem, kiedy użytkownik naciśnie ctrl+ C lub ctrl + break, MS-DOS przekaże sterownie do naszego programu obsługi break.

Być może najlepszą reakcją na przerwanie break jest ustawienie jakiejś flagi mówiącej aplikacji o wystąpieniu break a potem wyjście do aplikacji testującej tą flagę sensownie wskazując czy powinna się zamknąć. Oczywiście wymaga to żebyśmy testowali tą flagę w różnych miejscach naszego programu, zwiększając złożoność naszego kodu. Inną alternatywa jest zachowanie oryginalnego wektora int 23h i przekazanie sterowania do DOS'owego programu obsługi break, po tym jak sami obsłużymy jakąś inna ważną operację .Możemy również napisać wyspecjalizowany program obsługi break zwracający do DOS kod zakończenia, który może odczytać proces macierzysty.

Oczywiście, nie ma powodów abyśmy nie mogli zmienić wektora przerwań 23h w różnych punktach całego naszego programu obsługując wymagane zmiany. W różnych punktach możemy zablokować przerwanie break całkowicie., przywracając wektory przerwań, albo zachęcić użytkownika w innym punkcie.

---

## 19.1.3 OBSŁUGA WYJĄTKÓW W DOS: OBSŁUGA BŁĘDU KRYTYCZNEGO

DOS wywołuje podprogram obsługi błędu krytycznego przez wykonanie instrukcji int 24h gdy tylko wystąpi jakiś rodzaj błędu I/O. Domyślnie program obsługi drukuje dobrze znaną wiadomość:

I/O Devixe Specific Error Message

Abort, Retry , Ignore, Fail?

Jeśli użytkownik naciśnie "A", kod bezpośrednio wróci do programu DOS'a COMMAND.DOM; nie zamknie nawet żadnego otwartego pliku. Jeśli użytkownik naciśnie „R" , dla ponów, MS-DOS będzie ponawiał operację I/O, mimo, że zazwyczaj wynikiem jest wywołanie innego programu obsługi błędu krytycznego. Opcja „I" mówi DOS'owi, żeby zignorował błąd i wrócił do wywołującego programu jak gdyby nic się nie stało. „F" instruuje DOS, aby zwrócił kod błędu do wywołującego programu i pozwolił obsłużyć ten problem.

Z powyższych opcji, naciśnięcie przez użytkownika „A" jest najbardziej niebezpieczne. Powoduje natychmiastowy powrót do DOS, a nasz kod nie dostaje szansy na poprawienie niczego. Na przykład, jeśli zaktualizujemy jakieś wektory przerwań, program nie będzie miał możliwości przywrócenia ich jeśli użytkownik

wybierze opcję przerwij. Może to spowodować krach systemu, kiedy MS-DOS załaduje następny program na szczycie naszych podprogramów obsługi przerwań w pamięci.

Dla przechwycenia krytycznych błędów DOS, będziemy musieli zaktualizować wektor przerwań 24h aby wskazywał nasz podprogram obsługi przerwań. Na wejściu do naszego programu obsługi przerwania 24h stos będzie zawierał następujące dane:

| | |
|---|---|
| FLAGI | Oryginalny adres powrotny INT 24h |
| CS | |
| IP | |
| ES | Rejestry DOS odłożone dla naszego programu obsługi INT 24h |
| DS. | |
| BP | |
| DI | |
| SI | |
| DX | |
| CX | |
| BX | |
| AX | |
| FLAGI | Adres powrotny (do DOS) dla naszego programu obsługi |
| CS | |
| IP | |

Zawartość Stosu Na Wejściu Do Programu Obsługi Błędu Krytycznego

MS-DOS przekazuje ważne informacje w kilku tych rejestrach do naszego programu obsługi błędu krytycznego. Poprzez sprawdzenie tych wartości możemy określić powód błędu krytycznego i urządzenie na którym on wystąpił. Najbardziej znaczący bit w rejestrze ah określa czy wystąpił błąd w strukturze bloku urządzenia (zazwyczaj dysk lub taśma) lub znaku urządzenia. Pozostałe bity w ah mają następujące znaczenie:

| Bit(y) | Opis |
|---|---|
| 0 | 0 = Operacja odczytu<br>1 = Operacja zapisu |
| 1-2 | Wskazuje sztuczne obszary dysku<br>00 – obszar MS-DOS<br>01 – tablica alokacji plików (FAT)<br>10 – Katalog główny<br>11 – Obszar pliku |
| 3 | 0 – Nie uznana błędna odpowiedź<br>1- Odpowiedź błędna jest OK |
| 4 | 0 – Odpowiedź ponowienia nie uznana<br>1- Odpowiedź ponowienia jest OK. |
| 5 | 0 – Odpowiedź zignorowania nie uznana<br>1- Odpowiedź zignorowania jest OK. |
| 6 | Niezdefiniowane |
| 7 | 0 – Błąd urządzenia znakowego<br>1 – Błąd struktury blokowej urządzenia |

Tablica 69: Bity błędów urządzenia w AH

Dodatkowe bity w ah, dla struktury blokowej urządzeń w rejestrze al. zawierają numer urządzenia gdzie wystąpił błąd (0=A, 1=B,2=C, itd.). Wartość w rejestrze al. jest niezdefiniowana dla urządzenia znakowego.

Niższa połówka rejestru di zawiera dodatkowe informacje o błędzie urządzenia blokowego (najwyższy bajt di jest niezdefiniowany, musimy zamaskować te bity zanim spróbujemy przetestować tą daną)

| Kod błędu | Opis |
|---|---|
| 0 | Zapis błędu ochrony |
| 1 | Nieznane urządzenie |
| 2 | Urządzenie nie gotowe |
| 3 | Niewłaściwe polecenie |
| 4 | Błąd danych (błąd CRC) |
| 5 | Długość żądanej struktury jest niewłaściwa |
| 6 | Błąd przeszukiwania na urządzeniu |
| 7 | Dysk niesformatowany dla MS-DOS |
| 8 | Nie znaleziono sektora |
| 9 | Brak papieru w drukarce |
| 0Ah | Błąd zapisu |
| 0Bh | Błąd odczytu |
| 0Ch | Niepowodzenie ogólne |
| 0Fh | Dysk zmieniony w nieodpowiednim czasie |

Tablica 70: Kody błędów struktury blokowej urządzenia (w najmniej znaczącym bajcie DI)

Na wejściu do naszego programu obsługi błędu krytycznego, przerwania są wyłączane. Ponieważ ten błąd wystąpi jako wynik jakieś funkcji MS-DOS, MS-DOS jest już wprowadzony i nie będziemy mogli zrobić żadnego wywołania innych funkcji niż 1-0Ch i 59h (pobranie informacji rozszerzonego błędu)

Nasz program obsługi błędu krytycznego musi zachować wszystkie rejestry z wyjątkiem al. Program musi wrócić do DOS instrukcją iret a al. musi zawierać jeden z poniższych kodów:

| Kod | Znaczenie |
|---|---|
| 0 | Zignoruj błąd urządzenia |
| 1 | Ponowne ponowienie operacji I/O |
| 2 | Zakończenie procesu (przerwanie) |
| 3 | Wywołanie błędu systemu bieżącego |

Poniższy kod dostarcza trywialnego przykładu obsługi błędu krytycznego. Program główny próbuje wysłać znak do drukarki. Jeśli nie ma połączonej drukarki lub wyłączyliśmy drukarkę przed uruchomieniem programu, bezie wygenerowany błąd krytyczny.

```
; Próbka programu obsługi  błędu krytycznego IT 24h
;
; Kod ten demonstruje próbkę programu obsługi błędu krytycznego. Aktualizuje INT 24h i wyświetla właściwą
; wiadomość o błędzie i pyta użytkownika czy chce ponowić, przerwać , zignorować  lub zaniedbać (podobnie jak
; DOS)

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment  para  public  'data'

Value           word     0
ErrCode         word     0

dseg            ends

cseg            segment  para  public  'code'
                assume   cs:cseg, ds:dseg
```

```
; Zastąpienie programu obsługi błędu krytycznego. Zauważmy, że ten podprogram jest nawet gorszy niż DOS'a, ale
; demonstruje jak napisać taki podprogram. Zauważmy, ze nie możemy wywołać  żadnego programu I/O Biblioteki
; Standardowej w programie obsługi błędu krytycznego ponieważ nie używają one funkcji DOS 1-0Ch, które
; są jedynie dostępne w DOS

CritErrMsg      byte    cr, lf
                byte    "DOS Crirtical Error!", cr, lf
                byte    "A)bort R)etry, I)gnore, F)ail? $"

MyInt24         proc    far
                push    dx
                push    ds
                push    ax

                push    cs
                pop     ds

Int24Lp:        lea     dx, CritErrMsg
                mov     ah, 9                   ;wydruk ciągu DOS
                int     21h

                mov     ah, 1                   ;funkcja odczytu DOS
                int     21h
                and     al., 5Fh                ;Konersja l.c →u .c

                cmp     al, 'I'                 ;ignorujemy?
                jne     NotIgnore
                pop     ax
                mov     al, 0
                jmp     Quit

NotIgnore:      cmp     al, 'r'                 ;ponawiamy?
                jne     NotRetry
                pop     ax
                mov     al, 1
                jmp     Quit24

NotRetry:       cmp     al, 'A'                 ;przerywamy?
                jne     NotAbort
                pop     ax
                mov     al., 2
                jmp     Quit24

NotAbort:       cmp     al., 'F'
                jne     BadChar
                pop     ax
                mov     al, 3

Quit24:         pop     ds
                pop     dx
                iret

BadChar:        mov     ah, 2
                mov     dl, 7                   ;znak dzwonka
                jmp     Int24Lp
MyInt24         endp
```

```
Main        proc
            mov     ax, seg
            mov     ds, ax
            mov     es, ax
            meminit

            mov     ax, 0
            mov     es, ax
            mov     word ptr  es:[24h*4], offset  MyInt24
            mov     es:[24h*4+2], cs

            mov     ah, 5
            mov     al, ,a'
            int     21h
            rcl     Value, 1
            and     Value, 1
            mov     ErrCode, ax
            printf
            byte    cr, lf, lf
            byte    "Print char returned with error status %d and "
            byet    "error code %d\n", 0
            dword   Value, ErCode

Quit:       ExitPgm                             ;makro DOS wyjścia z programu
Main        endp

cseg        ends

; Alokacja stosownej ilości pamięci na stosie (8k). Notka: jeśli użyjemy pakietu dopasowani do wzorca powinniśmy
; ustawić jakiś duży stos

sseg        segment  para  stack  'stack'
stk         db       1024  dup ("stack")
sseg        ends

;zzzzzzseg  segemnt para public  'zzzzzz'
LastBytes   db       16 dup (?)
Zzzzzzseg   ends
            end      Main
```

19.1.4 OSBSŁUGA WYJĄTKÓW W DOS: PRZERWANIA KONTROLOWANE

W dodatku do wyjątków break i błędów krytycznych, 80x86 ma wyjątki, które mogą zdarzyć się podczas wykonywania naszego programu. Przykłady to wyjątek błędu dzielenia, wyjątki graniczne i wyjątki nielegalnych opcodów. Dobrze napisana aplikacja powinna zawsze obsługiwać wszystkie możliwe wyjątki.

DOS nie dostarcza bezpośredniego wsparcia dla tych wyjątków, inaczej niż możliwe domyślne pogramy obsługi. W szczególności, DOS nie przywraca takich wektorów kiedy program się kończy; jest jakaś aplikacja, program obsługi break i obsługi błędu krytycznego, które muszą się tym zająć.

19.1.5 PRZEKIEROWANIE I/O DLA PROCESU POTOMNEGO

Kiedy proces potomny zaczyna się wykonywać, dziedziczy wszystkie otwarte pliki z procesu macierzystego ( z wyjątkiem pewnych plików otwieranych funkcjami plików sieciowych). W szczególności, wliczamy w to domyślne pliki otwierane dla DOS urządzeń standardowego wejścia, standardowego wyjścia, standardowego błędu, pomocniczych i drukarki. DOS przypisuje uchwytom plików wartość od zera do cztery, odpowiednio, dla tych

urządzeń. Jeśli proces macierzysty zamyka jeden z tych uchwytów plików a potem zmieniamy uchwyt funkcją Force Duplicat File Handle .

Zauważmy, że funkcja DOSEXEC nie przetwarza operatorów przekierowania I/O („<" i „>" i „|"). Jeśli chcemy przekierować standardowe I/O procesu potomnego, musimy zrobić to przed załadowaniem i wykonaniem tego procesu potomnego. Przekierowując jeden z pięciu standardowych urządzeń I/O, powinniśmy wykonać następujące kroki:

1) Duplikujemy uchwyt pliku jaki chcemy przekierować (np. przrekierowujemy standardowe wyjście, duplikujemy uchwyt pliku jeden)
2) Zamykamy plik (np. uchwyt pliku jeden dla standardowego wyjścia)
3) Otwieramy plik używając standardowej funkcji DOS'a Craete lub Create New
4) Używamy funkcji Force Duplicate File Handle do skopiowania nowego uchwytu pliku na uchwyt pliku jeden
5) Uruchamiamy proces potomny
6) Przy powrocie z potomka, zamykamy plik
7) Kopiujemy uchwyt pliku zduplikowany w kroku jeden z powrotem do standardowego wyjścia uchwytu pliku używając funkcji Force Duplicate Handle

Ta technika wygląda jak gdyby była doskonała dla przekierowania drukarki lub portu szeregowego I/O. Niestety wiele programów omija DOS kiedy wysyła dane do drukarki i używa funkcji BIOS lub, co gorsze, wysyła bezpośrednio do sprzętu. Prawie żadne oprogramowanie nie zawraca sobie głowy wsparciem portu szeregowego DOS – to naprawdę jest złe. Jednakże, większość programów robi wywołanie DOS'a dla znaków wejściowych i wyjściowych w standardowych urządzeniach wejścia, wyjścia i błędu. Poniższy kod demonstruje jak przekierować wyjście procesu potomnego do pliku.

```
; REDIRECT.ASM – Demonstruje jak przekierować I/O dla procesu potomnego. Ten szczególny program wywołuje
; COMMAND.COM do wykonania polecenia DIR, kiedy wysyłamy do określonego pliku wyjściowego

                include         stdlib.a
                includelib      stdlib.lib

dseg            segment para  public 'data'

OrigOutHandle   word    ?                               ;przechowanie kopii uchwytu STDOUT
FileHandle      word    ?                               ;uchwyt I/O
FileName        byte    „dirctry.txt", 0                ;nazwa pliku dla danych wyjściowych

; struktura EXEC MS-DOS

ExecStruct      word    0                       ;używamy bloku środowiska macierzystego
                dword   CmdLine                 ;dla parametrów linii poleceń
                dword   DfltFCB
                dword   DfltFCB

DfltFCB         byte    3, „„", 0, 0, 0, 0,0
CmdLine         byte    7, "/c DIR", 0dh        ;plecenie katalogu
PgmName         dword   PgmNameStr              ;wskaźnik do nazwy pgm
PgmNameStr      byte    „c:\command.com", 0
dseg            ends

cseg            segemnt para  public ' code'
                assume  cs:cseg, ds;dseg

Main            proc
                mov     ax, dseg                        ;pobranie wskaźnika do segmentu zmiennych
```

```
                mov     ds., ax
                Meminit                 ;start menadżera pamięci
; Zwolnienie jakiejś pamięci dla COMMAND.COM:

                mov     ah, 62h         ;pobranie wartości naszego PSP
                int     21h
                mov     es, bx
                mov     ax, zzzzzzseg   ;obliczanie rozmiaru uruchomionego kodu rezydentnego
                sub     ax, bx
                mov     bx, ax
                mov     ah, 4ah         ;zwolnienie nie używanej pamięci
                int     21h

; zachowanie oryginalnej uchwytu pliku wyjściowego
                mov     bx, 1           ;std out jest uchwytem pliku 1
                mov     ah, 45h         ;duplikujemy uchwyt pliku
                int     21h
                mov     OrigOutHandle, ax       ;zachowanie zduplikowanego uchwytu

;Otwieramy plik wyjściowy:

                mov     ah, 3ch         ;tworzymy plik
                mov     cx, 0           ;normalny atrybut
                lea     dx, FileName
                int     21h
                mov     FileHandle, ax  ;zachowanie otwieranego uchwytu pliku

; Wymuszamy standardowe wyjście do wysłania danych wyjściowych do tego pliku
; Robimy to przez wymuszenie uchwytu pliku do uchwytu pliku #1 (stdout)
                mov     ah, 46h         ;wymuszenie uchwytu pliku
                mov     cx, 1           ;istniejący uchwyt do zmiany
                mov     bx, FileName    ;nowy uchwyt pliku do użycia
                int     21h

; Wydruk pierwszej linii do pliku:
                print
                byte    „Redirected  directory listing:”, cr,lf,0

;Okay, wykonujemy polecenie DIR DOS’a (to znaczy, wykonuje COMMAND.COM parametrem
; lini poleceń „/c DIR”)
                mov     bx,  seg  ExecStruct
                mov     es, bx
                mov     bx, offset ExecStruct           ;wskaźnik do rekordu programu
                lds     dx, PgmName
                mov     ax, 4b00h                       ;exec pgm
                int     21h

                mov     bx, sseg                        ;resetujemy stos przy zwrocie
                mov     ss, ax
                mov     sp, offset EndStk
                mov     bx, seg  dseg
                mov     ds, bx

;Okay, zamykamy plik wyjściowy I przekształcamy standardowe wyjście  z powrotem do konsoli

                mov     ah,3eh                          ;zamykamy plik wyjściowy
```

```
                mov     bx, FileHandle
                int     21h

                mov     ah, 46h                 ;wymuszenie duplikacji uchwytu
                mov     cx, 1                   ;StdOut
                mov     bx, OrigOutHandle       ;Przywrócenie poprzedniego uchwytu
                int     21h

;Zwrot sterowania do MS-DOS

Quit:           ExitPgm
Main            endp
cseg            ends

sseg            segment para  stack  'stack'
                dw      128  dup  (0)
endstk          dw      ?
sseg            ends

zzzzzzseg       segment para  public  'zzzzzzseg'
Heap            db      200 dup (?)
Zzzzzzseg       ends
                end     Main
```

## 19.2 PAMIĘĆ DZIELONA

Jedynym problem z uruchamianiem różnych programów DOS jako część pojedynczej aplikacji jest komunikacja międzyprocesowa. To znaczy, jak wszystkie te programy przemawiają jeden do drugiego? Kiedy typowa aplikacja DOS działa, DOS ładuje cały kod i segmenty danych; nie ma żadnego zabezpieczenia, inaczej niż odczytywanie danych z pliku lub kod zakończenia procesu, gdzie jeden proces przekazuje informacje do drugiego. Chociaż I/O plików będzie działało, jest to nieporęczne i wolne. Idealnym rozwiązaniem byłoby aby jeden proces zostawił kopie różnych zmiennych, które mogą dzielić inne procesy. Nasze programy mogą łatwo to zrobić przy użyciu pamięci dzielonej.

Większość nowoczesnych wielozadaniowych systemów operacyjnych dostarcza pamięci dzielonej – pamięci, która pojawia się w przestrzeni adresowej dwóch lub więcej procesów. Co więcej, taka pamięć dzielona jest często trwała, w znaczeniu , że trwa przechowywanie wartości po tym jak proces tworzenia się kończy. Pozwala to innym procesom zaczynać się później i używać wartości pozostawionych przez twórcę zmiennych.

Niestety, MS-DOS nie jest nowoczesnym wielozadaniowym systemem operacyjnym i nie wspiera pamięci dzielonej. Jednakże, możemy łatwo napisać program rezydentny, który dostarcza tej zagubionej przez DOS zdolności. Poniższa sekcja opisuje jak stworzyć dwa typowe regiony pamięci dzielonej – statyczny i dynamiczny.

### 19.2.1 STATYCZNA DZIELONA PAMIĘĆ

TSR implementujący statycznie dzieloną pamięć jest trywialny. Jest to bierny TSR, który dostarcza trzech funkcji – sprawdzania obecności, usuwania i wskaźnika segmentu powrotu. Nierezydentna część po prostu alokuje 64Kb segment danych a potem się kończy. Inne procesy mogą uzyskać adres 64K bloku pamięci dzielonej poprzez wywołanie „wskaźnika segmentu powrotu". Procesy te mogą umieszczać wszystkie swoje dzielone dane w segmencie należącym do TSR'a. Kiedy jeden proces kończy się , dzielony segment pozostaje w pamięci jako część TSR'a. Kiedy drugi proces się uruchamia i łączy z dzielonym segmentem , zmienne z segmentu dzielonego są jeszcze nienaruszone, wiec nowy proces może uzyskać dostęp do tych zmiennych. Kiedy wszystkie procesy przeszły dane dzielone, użytkownik może usunąć dzieloną pamięć TSR'a funkcją usuwania.

Jak przedstawiono powyżej, nie jest prawie niczym zrobienie pamięci dzielonej TSR. Implementuje to następujący kod:

```
; SHARDMEM.ASM
;
```

```
; Ten TSR odkłada 64k region pamięci dzielonej  dla innych procesów
;
; Użycie:
;       SHARDMEM -              Ładowanie rezydentnej części i aktywowowanie zdolności pamięci
;                               dzielonej
;       SHARDMEM  REMOVE -      Usuwanie pamięci dzielonej TSR'a z pamięci
; Ten TSR sprawdza, aby się upewnić, że nie ma już aktywnej kopii w pamięci. Kiedy usuwamy go z pamięci
; upewniamy się, że nie ma innych łańcuchów przerwań w INT 2Fh przed dokonaniem usuwania.
;
; Następujący segment musi pojawić się w tym porządku i przed włączeniem Biblioteki Standardowej

ResidentSeg     segment para  public „Resident'
ResidentSeg     ends

SharedMemory    segemnt para  public  'Shared'
SharedMemory    ends

EndResident     segment para  public 'EndRes'
EndResident     ends

                .xlist
                .286
                .include        stdlib.a
                .includelib     stdlib.lib
                .list

;Segment rezydentny, który przechowuje kod TSR'a:

ResidentSeg     segment para   public 'Resident'
                assume  cs:ResidentSeg, ds:nothing

; numer ID Int 2Fh dla tego TSR'a:

MyTSRID         byte    0
                byte    0

; PSP jest adresem psp tego programu

PSP             word    0
OldInt2F        dword   ?

; MyInt2F -     Dostarcza wsparcia int 2Fh (przerwanie różnych procesów) dla tego TSR'a. Przerwanie
;               różnych procesów rozpoznaje poniższe podfunkcje (przekazane w AL.):
;
;               00h – sprawdzanie obecności:    Zwraca 0FFh w AL i wskaźnik do ciągu ID w es:di
;                                               jeśli ID TSR'a (w AH) jest dopasowany do tego
;                                               szczególnego ciągu.
;
;               01h- usuwanie:                  Usuwa TSR z pamięci. Zwraca 0 w AL jeśli
;                                               powodzenie, 1 w AL jeśli niepowodzenie
;
;               10h- wskaz. Adr.seg. -          Zwraca adres segmentu dzielonego w ES

MyInt2Fh        proc    far
                assume  ds.:nothing
```

```
                    cmp     ah, MyTSRID         ;dopasowano identyfikator naszego TSR'a
                    je      YepItsOurs
                    jmp     OldInt2F

;Okay, wiemy, że to jest  nasze ID, teraz sprawdzamy obecność, usuwanie lub wywołanie zwracanego
; segmentu

YepItsOurs:         cmp     al., 0              ;funkcja weryfikacji
                    jne     TryRmv
                    mov     al., 0ffh           ;zwracane powodzenie
                    lesi    IDString
                    iret                        ;wracamy do kodu wywołującego

IDString            byte    „Static Shared Memory TSR", 0

TryRmv:             cmp     al, 1               ;funkcja usuwania
                    jne     TryRetSeg

;Zobaczmy czy możemy usunąć ten TSR:

                    push    es
                    mov     ax, 0
                    mov     es, ax
                    cmp     word  ptr  es:[2Fh*4], offset MyInt2F
                    jne     TRDone
                    cmp     word  ptr  es:[2Fh*4 +2], seg MyInt2Fh
                    je      CanRemove           ;skok jeśli można
TRDone:             mov     ax,1                ;zwraca teraz niepowodzenie
                    pop     es
                    iret

;Okay,  chcemy usunąć to *i* możemy usuwać go z pamięci
; dopilnujmy wszystkiego tu
                    assume  ds: ResidentSeg

CanRemove:          push    ds.
                    Pusha
                    cli                                 ;wyłączamy przerwania kiedy pracujemy z
                    mov     ax, 0                       ; z wektorami przerwań
                    mov     es, ax
                    mov     ax, cs
                    mov     ds., ax

                    mov     ax, word  ptr OldInt2F
                    mov     es:[2Fh*4], ax
                    mov     ax, word  ptr  OldInt2F+2
                    mov     es:[2Fh*4+2], ax

;Okay, jedna ostatnia rzecz przed wyjściem – oddajemy zaalokowaną pamięć dla tego TSR z powrotem
; do DOS'a
                    mov     ds., PSP
                    mov     es ds:[2Ch]     ;wskaźnik do bloku środowiska
                    mov     ah, 49h         ;funkcja zwalniania pamięci DOS
                    int     21h

                    mov     ax, ds.         ;zwolnienie przestrzeni kodu programu
```

```
                        mov     es, ax
                        mov     ah, 49h
                        int     21h

                        popa
                        pop     ds.
                        po      es
                        mov     ax, 0                   ;zwraca powodzenie

;Zobaczmy czy zwracano adres segmentowy naszego dzielonego segmentu tutaj

TryRetSeg:              cmp     al., 10h                ;opcod segmentu powrotu
                        jne     IllegalOp
                        mov     ax, SharedMemory
                        mov     es, ax
                        mov     ax, 0                   ,zwrot z powodzeniem
                        clc
                        iret
; wywołanie z nielegalną wartością podfunkcji. Próbujemy zrobić jak najmniej szkody jeśli to możliwe

IllegalOp:      mov     ax, 0                           ;kto wie co o tym myśleć?
                iret
MyInt2F         endp
                assume  ds:nothing
ResidentSeg     ends

;Tu , segment, będzie aktualnie przechowywał dzielone dane

SharedMemory    segment para  public  'Shared'
                db      0FFFFh dup (?)
SharedMemory    ends

Cseg            segment para  public ,code'
                assume  cs:cseg, ds:ResidentSeg

;SeeIfPresent-  Sprawdzamy aby zobaczyć, czy nasz  TSR jest już obecny w pamięci. Ustawiamy
;               flagę zera jeśli jest, zeruje ta flagę  jeśli nie jest

SeeIfPresent    proc    near
                push    es
                push    ds.
                push    di
                mov     cx, 0ffh                ;start z ID 0FFh
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                   ; funkcja weryfikacji obecności
                int     2Fh
                pop     cx
                cmp     al., 0                  ;Obecny w pamięci
                je      TryNext
                strcmpl
                byte    „Static Shared Memory TSR" , 0
                je      Success

TryNext:        dec     cl                      ;test ID użytkownika 80f..FFh
                js      IDLoop
```

```
                cmp     cx, 0                   ; zerowanie flagi zera

Success:        pop     di
                pop     ds.
                pop     es
                ret
SeeIfPresent    endp

;FindID -       Określa pierwszy (cóż w rzeczywistości ostatni) ID TSR'a dostępnego w łańcuchu
;               przerwań równoczesnych procesów. Zwraca tą wartość w rejestrze CL
;
;               Zwraca ustawioną flagę zera jeśli lokuje pusty slot.
;               Zwraca wyzerowaną flagę zera jeśli niepowodzenie

FindID          proc    near
                push    es
                push    ds
                push    di

                mov     cx, offh                ;start z ID 0FFh
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                   ;funkcja weryfikacji obecności
                int     2Fh
                pop     cx
                cmp     al., 0                  ;obecny w pamięci?
                je      Success
                dec     cl                      ;test ID użytkownika 80h..FFh
                js      IDLoop
                xor     cx, cx
                cmp     cx, 1                   ;zerowanie flagi zera
Success:        pop     di
                pop     ds.
                pop     es
                ret
FindID          endp

Main            proc
                meminit

                mov     ax, ResidentSeg
                mov     ds, ax

                mov     ah, 62h                 ;pobranie wartości PSP tego programu
                int     21h
                mov     PSP, bx

; zanim cokolwiek zrobimy musimy sprawdzić parametry linii poleceń. Jeśli jest to jeden i jest to słowo
; „REMOVE", wtedy usuwamy kopię rezydentną z pamięci używając przerwania równoczesnych procesów
; (2Fh)

                argc
                cmp     cx,1            ;musi mieć zero lub jeden parametr
                jb      TstPresent
                je      DoRemove
Usage:          print
```

```
                byte    „Usage:”, cr,lf
                byte    “ shardmem”, cr, lf
                byte    “or shardmem REMOVE”, cr, lf,0
                ExitPgm

; sprawdzenie polecenia REMOVE

DoRemove:       mov     ax, 1
                argv
                stricmpl
                byte    “REMOVE”,0
                jne     Usage

                call    SeeIfPresent
                je      RemoveIt
                print
                byte    “TSR  nie jest obecny w pamięci, nie można usunąć”
                byte    cr, lf,0
                ExitPgm

RemoveIt:       mov     MyTSRID, cl
                printf
                byte    “Usuwanie TSR’a (ID #%d) z pamięci…”, 0
                dword   MyTSRID

                mov     ah, cl
                mov     al., 1                  ;usuwanie cmd, ah zawiera ID
                int     2Fh
                cmp     al., 1                  ;Powodzenie?
                je      RmvFailure
                print
                byte    „removed”, cr,lf,0
                ExitPgm
RmvFailure:     print
                byte    cr, lf
                byte    “Nie można usunąć TSR’a z pamięci”, cr, lf
                byte    „Spróbuj usunąć inne TSR’y w odwrotnej kolejności”
                byte    „zainstalowaliśmy je”, cr, lf,0
                ExitPgm
;Okay, zobaczmy czy TSR jest już w pamięci. Jeśli tak, przerywamy proces instalacji

TstPresent:     call    SeeIfPresent
                jne     GetTSRID
                print
                byte    „TSR jest już obecny w pamięci” ,cr, lf
                byte    „przerwanie procesu instalacji”, cr, lf,0
                ExitPgm
;Pobranie ID naszego TSR’a i zachowanie go

GetTSRID:       call    FindID
                je      GetFileName
                print
                byte    „Zbyt wiele rezydentnych TSR’ów, nie można instalować”,cr,lf,0
                ExitPgm
GetFileName:    mov     MyTSRID, cl
                print
```

```
                byte    "Instalowanie przerwań….", 0

;Aktualizacja łańcucha przerwań INT 2Fh

                cli                                             ;wyłączenie przerwań
                mov     ax,0
                mov     es, ax
                mov     ax, es:[2Fh*4]
                mov     word  ptr OldInt2F, ax
                mov     ax, es;[2Fh*4+2]
                mov     word  ptr OldInt2F+2, ax
                mov     es:[2Fh*4], offset MyInt2F
                mov     es:[2Fh*4+2], seg ResidentSeg
                sti                                             ;włączamy ponownie przerwania

; mamy podłączone, jedyna rzecz jaka pozostała to wyzerowanie segmentu pamięci dzielonej  a potem TSR'a

                printf
                byte    „Instalowanie , TSR ID #%d.", cr,lf,0
                dword   MyTSRID

                mov     ax, SharedMemory                ;zerowanie segmentu pamięci dzielonej
                mov     es, ax
                mov     cx, 32768                       ; 32K słów = 64K bajtów
                xor     ax, ax                          ;zachowanie wszystkich zer
                mov     di, ax                          ;zaczynamy spod offsetu zero
        rep     stosw

                mov     dx, EndResident                 ;obliczamy rozmiar programu
                mov     dx, PSP
                mov     ax, 3100h                       ;polecenie TSR'a DOS
                int     21h
Main            endp
cseg            ends

sseg            segemt  para  stack 'stack'
stk             db      256 dup (?)
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16  dup (?)
Zzzzzzseg       ends
                end     Main
```

Pogram ten po prostu  wykrawa kawałek pamięci (64K w segmencie SharedMemory) i zwraca wskaźnik do niej w es, jeśli jakiś program wykonuje właściwe wywołanie int 2Fh (ah = TSR ID a al= 10h) Jedyny problem to jak Zadeklarować zmienne dzielone w aplikacji, która używa pamięci dzielonej? Cóż , jest to dosyć łatwe jeśli zagramy podstępną sztuczkę z MASM'em , LINK'erem , DOS'em i 80x86.

Kiedy DOS ładuje nasz program do pamięci, generalnie ładuje segmenty w takiej kolejności w jakiej pojawiają się w naszym pliku źródłowym. Biblioteka Standardowa UCR, na przykład, wykorzystuje to poprzez naleganie na włączenie segmentu nazywanego zzzzzzseg na końcu wszystkich naszych asemblerowych plików źródłowych. Podprogramy zarządzania pamięcią Biblioteki Standardowej UCR budują stertę zaczynającą się przy zzzzzzseg, która musi być ostatnim segmentem (zawierającym poprawne dane) ponieważ podprogramy zarządzania pamięcią mogą nadpisywać jakikolwiek zzzzzzseg.

Dla naszego segmentu pamięci dzielonej, chcielibyśmy stworzyć segment taki jak poniższy:

```
SharedMemory        segment  para  public 'Shared'
< definiujemy tu wszystkie zmienne dzielone >
SharedMemory        ends
```

Aplikacje, które dzielą dane zdefiniują wszystkie dzielone zmienne w tym dzielonym segmencie. Jest jednakże pięć problemów. Pierwszy , to jak powiadomimy asembler / linker/ DOS/ 80x86, że jest to segment dzielony, zamiast mieć oddzielny segment dla każdego programu? Cóż, ten problem jest łatwy do rozwiązania; nie musimy się martwić powiadamianiem MASM'a , linkera lub DOS o czymkolwiek. Sposobem wykonania tego by różne aplikacje, wszystkie , dzieliły ten sam segment w pamięci, jest wywołanie pamięci dzielonej TSR w powyższym kodzie z kodem funkcji 10h. Zwraca ona adres segmentu SharedMemory TSR'a w rejestrze es. W naszych programach asemblerowych oszukamy MASM, który sądzi, że es wskazuje lokalny segment pamięci dzielonej, kiedy faktycznie es wskazuje segment globalny.

Drugi problem jest drobny ale mimo to irytujący. Kiedy tworzymy segment MASM, linker i DOS rezerwują miejsce w pamięci na segment. Jeśli zadeklarujemy dużą liczbę zmiennych w segmencie dzielonym, może to zmarnować pamięć ponieważ program w rzeczywistości będzie używał przestrzeni pamięci w globalnym dzielonym segmencie. Łatwym sposobem żądania zwrotu pamięci, którą MASM zarezerwował dla tego segmentu jest zdefiniowanie segmentu dzielonego po zzzzzzseg w naszej aplikacji z dzieloną pamięcią. Poprzez zrobienie tego, Biblioteka Standardowa wchłonie zarezerwowaną pamięć dla (fikcyjnego) segmentu dzielonej pamięci na stercie, ponieważ cała pamięć po zzzzzzseg należy do sterty (kiedy używamy standardowej funkcji meminit)

Trzeci problem jest trochę trudniejszy do zajęcia się nim. Ponieważ nie będziemy używali segmentu lokalnego, nie możemy zainicjalizować żadnej zmiennej w segmencie pamięci dzielonej przez umieszczenie wartości w polu operandu dyrektyw bajtu, słowa, podwójnego słowa itd. Robiąc to inicjalizujemy tylko pamięć lokalną na stercie, system nie skopiuje tej danej do segmentu dzielonego globalnie. Generalnie, nie jest to problem ponieważ procesy normalnie nie inicjalizują pamięci dzielonej jeśli są ładowane. Zamiast tego, będą prawdopodobnie pojedyncze aplikacje, najpierw uruchomione, które zainicjalizują obszar pamięci dzielonej dla reszty procesów, które używają globalnego segmentu dzielonego.

Czwartym problemem jest to, że nie możemy zainicjalizować żadnej zmiennej adresem obiektu w pamięci dzielonej. Na przykład, jeśli zmienna shared_K jest w segmencie pamięci dzielonej, nie możemy użyć instrukcji takich jak te:

```
        printf
        byte    „Wartością shared_K jest %d\n", 0
        dword   shared_K
```

Problem z tym kodem jest taki, że MAMS inicjalizuje podwójne słowo po powyższym ciągu adresem zmiennej shared_K w lokalnej kopii dzielonego segmentu danych. Nie drukuje kopii w globalnie dzielonym segmencie danych.

Ostatni problem jest drobny. Wszystkie programy , które używają globalnie dzielonego segmentu pamięci muszą zdefiniować swoje zmienne pod identycznym offsetem wewnątrz dzielonego segmentu MASM przydziela offsety do zmiennych wewnątrz segmentu, jeśli jest jeden bajt w deklaracji jakiejś zmiennej, nasz program będą przydzielone jego zmienne pod różnymi adresami, które inne procesy współużytkują w globalnym dzielonym segmencie. To będzie zaciemniało pamięć i stworzy katastrofę Jedynym sensownym sposobem deklaracji zmiennych dla programów z dzieloną pamięcią jest stworzenie pliku zawierającego deklaracje wszystkich dzielonych zmiennych dla wszystkich odnośnych programów. Potem zawieramy ten pojedynczy plik we wszystkich programach , które współużytkują te zmienne. Teraz możemy dodać, usuwać lub modyfikować zmienne bez martwienia się o deklaracje zmiennych dzielonych w innych plikach.

Następujące dwie próbki programów demonstrują użycie pamięci dzielonej. Pierwsza aplikacja odczytuje ciąg od użytkownika i upycha go w pamięci dzielonej. Druga aplikacja odczytuje ciąg z pamięci dzielonej i wyświetla go na monitorze.

Najpierw, mamy tu plik zawierający deklaracje zmiennej dzielonej używanej przez obie aplikacje:

```
;shmvars.asm
;
; Plik tez zawiera deklarację  zmiennej pamięci dzielonej używanej przez wszystkie aplikacje, które odnoszą się do
; pamięci dzielonej

InputLine       byte    128   dup (?)
```

Tu mamy pierwszą aplikację, która odczytuje ciąg wejściowy od użytkownika i popycha do pamięci dzielonej:

```
; :SHMAPP1.ASM
;
;To jest aplikacja o dzielonej pamięci, która używa statycznie dzielonej pamięci  TSR (SHARDMEM.ASM).
; Program ten wprowadza ciąg od użytkownika i przekazuje ten ciąg do SHMAPP2.ASM w całym obszarze
; pamięci dzielonej
;
                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment para  public  'data'
ShmID           byte    0
dseg            ends

cseg            segment para public  'code'
                assume  cs:cseg, ds:dseg, es:SharedMemory

;SeeIfPresent-  Sprawdzamy czy pamięć dzielona TSR jest już obecna w pamięci. Ustawiamy flagę zera jeśli jest
;               zerujemy flagę zera jeśli nie jest. Podprogram ten zawraca również ID TSR'a w CL

SeeIfPresent    proc    near
                push    es
                push    ds
                push    di
                mov     cx, 0ffh                        ;start z  ID 0FFh
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                           ;funkcja weryfikacji obecności
                int     2Fh
                pop     cx
                cmp     al., 0                          ;obecny w pamięci
                je      TryNext
                strcmpl
                byte    „Static Shared Memory TSR", 0
                je      Success
TryNext:        dec     cl                              ;testujemy ID użytkownika 80h..FFh
                js      IDLoop
                cmp     cx, 0                           ; zerowanie flagi zera
Success:        pop     di
                pop     ds.
                pop     es
                ret
SeeIfPresent    endp

; Program główny dla aplikacji #1  włącza pamięć dzieloną TSR a potem odczytuje ciąg od użytkownika
; (przechowując ciąg w pamięci dzielonej) a potem kończy

Main            proc
                assume  cs:cseg, ds.:dseg, es:Sharedmemory
                mov     ax, dseg
                mov     ds, ax
                meminit
```

```
                print
                byte    "Shared memory aplication #1", cr, lf,0

;zobaczmy czy pamięć dzielona TSR jest w okolicy:

                call    SeeIfPresent
                je      ItsThere
                print
                byte    „Shared Memory TSR (SHARDMEM) is not loaded",cr, lf
                byte    "This program cannot continue execution",cr,lf,0
                ExitPgm

;Jeśli pamięć dzielona TSR jest obecna, pobieramy adres dzielonego segmentu do rejestru ES:

ItsThere:       mov     ah, cl                  ;ID naszego TSR'a
                mov     al, 10h                 ;pobieramy adres dzielonego segmentu
                int     21h

;Pobieramy wejściową linie od użytkownika:
                print
                byte    „Wprowadź ciąg: „ ,0

                lea     di, InputLine           ;ES już wskazuje właściwy segment
                gets

                print
                byte    „Wprowadzono ':, 0
                puts
                print
                byte    „'do pamięci dzielonej.", cr,lf,0

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segemnt  para  stack 'stack'
stk             db      1024  dup ("stack")
sseg            ends

zzzzzzseg       segemnt  para  public  'zzzzzz'
LastBytes       db      16  dup  (?)
Zzzzzzseg       ends

; Segment pamięci dzielonej musi pojawić się po  „zzzzzzseg". Zauważmy ,że nie jest to fizyczna
; pamięć dla danych w dzielonym segmencie. Jest to w rzeczywistości miejsce składowania więc możemy
; zadeklarować zmienne i generować ich właściwe offsety. Biblioteka Standardowa UCR będzie używać
; ponownie pamięci powiązanej z tym segmentem dla sterty. Dla uzyskania dostępu do danych w segmencie
; dzielonym, aplikacja ta wywołuje pamięć dzieloną TSR dla uzyskania prawdziwego adresu segmentu
; pamięci dzielonej . Może potem uzyskać dostęp do zmiennych w segmencie pamięci dzielonej
;
; Zauważmy, ze wszystkie zmienne zadeklarowane wchodzą do pliku wejściowego.  Wszystkie aplikacje ,
; odnoszą się do segmentu pamięci dzielonej  wliczając w to ten plik w segmencie SharedMemory. Zakładamy, że
;  wszystkie dzielone segmenty mają dokładnie takie samo rozmieszczenie

SharedMemory            segment  para  public  'Shared'
```

```
                    Include   shmvars.asm
SharedMemory        ends
                    end       Main
```

Druga aplikacja jest bardzo podobna, oto ona:

```
; SHMAPP2.ASM
;
; Jest to aplikacja z dzieloną pamięcią, która używa statycznie dzielonej pamięci TSR (SHARDMEM.ASM).
; Program ten zakłada, że użytkownik ma już uruchomiony program SHMAPP1 wprowadzający ciąg do
; pamięci dzielonej. Program ten po prostu drukuje ten ciąg z pamięci dzielonej

                    .xlist
                    include         stdlib.a
                    includelib      stdlib.lib
                    .list

dseg                segemnt  para   public ‘data’
ShmID               byte     0
dseg                ends

cseg                segemnt  para  public  ‘code’
                    assume   cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent      Sprawdzamy żeby zobaczyć czy pamięć dzielona TSR jest obecna w pamięci. Ustawia flagę zera
;                   jeśli jest, zeruje flagę zera jeśli nie ma. Podprogram ten również zwraca ID TSR’a w CL

 SeeIfPresent       proc     near
                    push     es
                    push     ds.
                    push     di
                    mov      cx, 0ffh                ;zaczynamy z ID 0ffh
IDLoop:             mov      ah, cl
                    push     cx
                    mov      al, 0                   ;funkcja weryfikacji obecności
                    int      2Fh
                    pop      cx
                    cmp      al., 0                  ;obecny w pamięci?
                    je       TryNext
                    strcmpl
                    byte     „Static Shared Memory TSR”, 0
                    je       Success

TryNext:            dec      cl                              ;test ID użytkownika 80h..FFh
                    js       IDLoop
                    cmp      cx, 0                           ;zerowanie flagi zera
Success:            pop      di
                    pop      ds.
                    pop      es
                    ret
SeeIfPresent        endp

; Program główny dla aplikacji #1 łączy się z pamięcią dzieloną TSR a potem czyta ciąg od użytkownika
; (przechowywany w pamięci dzielonej) a potem kończy
```

```
Main            proc
                assume  cs:cseg, ds.:dseg, es:SharedMemory
                mov     ax, seg
                mov     ds, ax
                meminit

                print
                byte    "Shared memory application #2", cr, lf, 0

;Zobaczmy czy jest pamięć dzielona TSR

                call    SeeIfPresent
                je      ItsThere
                print
                byte    „Shared Memoery TSR (SHARDMEM) is not loaded.",cr, lf
                byte    "This program cannot continue execution.",cr, lf,0
                ExitPgm

; Jeśli pamięć dzielona TSR jest obecna, pobieramy adres dzielonego segmentu do rejestru ES:

ItsThere:       mov     ah, cl                  ;ID naszego TSR'a
                mov     al, 10h                 ;pobieranie  adresu dzielonego segmentu
                int     2Fh
;Wydruk ciągu wejściowego w SHMAPP1:

                print
                byte    „String from SMAPP1 id '",0

                lea     di, InputLine           ;ES juz wskazuje właściwy segment
                puts

                print
                byte    „' from shared memory.", cr, lf,0

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para  stack 'stack'
stk             db      1024  dup  ("stack")
sseg            ends

zzzzzzseg       segment para public 'zzzzzz'
LastBytes       db      16   dup  (?)
zzzzzzseg       ends

; Segment dzielonej pamięci musi pojawić się po "zzzzzzseg". Zauważmy, że nie jest to fizyczna pamięć dla danych
; w segmencie dzielonym. Jest to tylko miejsce przechowywania więc możemy zadeklarować zmienne i generować
; ich właściwe  offsety. Biblioteka standardowa UCR użyje ponownie pamięci powiązanej z tym segmentem dla
; sterty. Aby uzyskać dostęp do danych aplikacja ta wywołuje pamięć dzieloną TSR aby uzyskać prawdziwy adres
; segmentowy segmentu dzielonej pamięci. Może potem uzyskać dostęp do zmiennych w segmencie pamięci
; dzielonej
;
;Zauważmy, że  wszystkie deklaracje zmiennych pochodziły z pliku wejściowego. Wszystkie aplikacje, które
; odnoszą się do segmentu pamięci dzielonej  zawierają ten plik w segmencie SharedMemory. To zakłada, że
```

```
; wszystkie dzielone segmenty mają takie samo rozmieszczenia

SharedMemory   segment para  public  'Shared'

               include       shmvars.asm
SharedMemory   ends
               end           Main
```

19.2.2 DYNAMICZNA PAMIĘĆ DZIELONA

Chociaż statycznie dzielona pamięć opisana w poprzedniej sekcji jest bardzo użyteczna, cierpi na kilka ograniczeń. Przede wszystkim, program, który używa globalnie dzielonego segmentu musi być świadomy lokacji każdego innego programu , który używa segmentu dzielonego. To świadczy, że używanie dzielonego segmentu jest ograniczone do pojedynczego zbioru współpracujących procesów danych w jednym czasie, Nie możemy mieć dwóch niezależnych zbiorów programów używających pamięci w tym samym czasie. Innym ograniczeniem systemu statycznego jest to, że musimy znać rozmiar wszystkich zmiennych, kiedy piszemy nasz program, nie możemy tworzyć dynamicznych struktur danych , których rozmiar różni się w czasie wykonania. Byłoby miłe, na przykład, mieć funkcje jak shmalloc i shmfree, które pozwoliły by nam dynamicznie alokować i zwalniać pamięć w dzielonym regionie. Na szczęście, jest bardzo łatwo pokonać te ograniczenia poprzez stworzenie dynamicznie dzielonego menadżera pamięci.

Sensowny dzielony menadżer pamięci będzie miał cztery funkcje: inicjalizacja, shmalloc, shmattach i shmfree. Funkcja inicjalizacyjna odzyskuje całą używaną pamięć dzieloną. Funkcja shmalloc pozwala procesowi zaalokować nowy blok pamięci dzielonej. Tylko jeden proces w grupie współpracujących procesów robi to wywołanie. Skoro shmalloc alokuje blok pamięci, inne procesy używają funkcji shmattach dla uzyskania adresu bloku pamięci dzielonej. Poniższy kod implementuje dynamicznego menadżera pamięci dzielonej. Kod jest podobny do tego z Biblioteki Standardowej, z wyjątkiem kodu zezwalającego na maksimum 64K pamięci na stercie.

```
; SHMALLOC.ASM
;
; Ten TSR ustawia system dynamicznej pamięci dzielonej
;
; TSR ten sprawdza aby upewnić czy nie ma już aktywnej kopii w pamięci. Kiedy usuwa się z pamięci,
; upewnia się, że nie ma innych łańcuchów przerwań w INT 2Fh zanim dokona usunięcia.
;
; Poniższe segmenty muszą pojawić się w takiej kolejności i przed zawarciem Biblioteki Standardowej

ResidentSeg    segment para  public  'Resident'
ResidentSeg    ends

SharedMemory   segment para  public 'Shared'
SharedMemory   ends

EndResident    segment para public  'EndRes'
EndResident    ends

               .xlist
               .286
               include       stdlib.a
               includelib    stdlib.lib
               .list

; Segment rezydentny  ,który przechowuje kod TSR:

ResidentSeg    segment para  public 'Resident'
               assume  cs: ResidentSeg, ds: nothing
```

```
NULL            equ     0

;Struktura danych dla alokowanego regionu danych
;
; Key-  użytkownik dostarcza ID do powiązanego regionu z określonym zbiorem procesów
;
; Next- wskazuje następny alokowany blok
; Prev- Wskazuje poprzedni alokowany blok
; Size- Rozmiar (w bajtach) alokowanego bloku, nie zwiera struktury nagłówka

Region  struct
key     word    ?
next    word    ?
prev    word    ?
blksize word    ?
Region  ends

Stratmem        equ     Region  ptr  [0]

AllocatedList   word    0               ;Wskazuje łańcuch alokowanego bloku
FreeList        word    0               ; Wskazuje łańcuch wolnych bloków

;Numer ID Int 2Fh dla tego TSR'a

MyTSRID         byte    0
                byte    0               ;możemy go wydrukować

;PSP jest  adresem psp dla tego programu

PSP             word    0

OldInt2F        dword   ?

; MyInt2F-      Dostarczamy int 2Fh (przerwanie równoczesnych procesów) dla tego TSR'a.
;               Przerwanie równoczesnych procesów rozpoznaje następujące podfunkcje
;               (przekazane w AL.):
;
;               00h- weryfikacja obecności:     Zwraca  0FFh w rejestrze AL i wskaźnik
;                                               do ID ciągu w es:di jeśli ID TSR'a (w AH)
;                                               jest dopasowane do tego szczególnego TSR'a
;               01h- usuwanie                   Usuwa TSR z pamięci. Zwraca 0 w AL jeśli
;                                               powodzenie, 1 w AL jeśli niepowodzenie
;               11h- shmalloc                   CX  zawiera rozmiar bloku do alokacji.
;                                               DX zawiera key do tego bloku. Zwraca
;                                               wskaźnik do bloku w ES:DI i rozmiar
;                                               alokowanego bloku w CX. Zwraca kod błędu
;                                               w AX. Zero nie jest błędem, jeden jest „już
;                                               istniejącym key", dwa jest  „niewystarczającym
;                                               zapotrzebowaniem na pamięć"
;               12h-shmfree                     DX zawiera key dla bloku. Funkcja ta zwraca określony
;                                               blok z pamięci.
;               13h-shminit                     Inicjalizuje system pamięci dzielonej zwalniając wszystkie
;                                               bloki aktualnie w użyciu
;               14h- shmattach                  DX zawiera key dla bloku. Przeszukuje ten blok i zwraca jego
;                                               adres w ES:DI. AX zawiera zero jeśli powodzenie, trzy jeśli nie
;                                               można ulokować bloku określonym key.
```

```
MyInt2F     proc    far
            assume  ds.:nothing

            cmp     ah, MyTSRID             ;identyfikator naszego TSR'a dopasowany?
            je      YepItsOurs
            jmp     OldInt2F

;Okay, znamy ten nasz ID, teraz sprawdzamy funkcje weryfikacji, usuwania lub zwracanego segmentu

YepItsOurs  cmp     al., 0                  ;funkcja weryfikacji
            jne     TryRmv
            mov     al., 0ffh               ;zwraca powodzenie
            lesi    IDString
            iret                            ;wraca do kodu wywołującego

IDString    byte    „Dynamic Shared Mmeory TSR",0

TryRmv      cmp     al, 1                   ; funkcja usuwania
            jne     Tryshmalloc

;zobaczmy czy możemy usunąć ten TSR:

            push    es
            mov     ax, 0
            mov     es, ax
            cmp     word  ptr  es:[2Fh*4] , offset MyInt2F
            jne     TRDone
            cmp     word  ptr  es:[2Fh*4+2], seg MyInt2F
            je      CanRemove               ; skok jeśli możemy
TRDone:     mov     ax, 1                   zwraca niepowodzenie
            pop     es
            iret

; Okay chcemy to usunąć i możemy to usunąć z pamięci . Dopilnujemy tego wszystkiego tutaj

            assume  ds: ResidentSeg
CanRemove:  push    ds
            pusha
            cli                             ;wyłączamy przerwania kiedy mieszamy w
            mov     ax, 0                   ; wektorach przerwań
            mov     es, ax
            mov     ax, cs
            mov     ds., ax

            mov     ax, word  ptr  OldInt2F
            mov     es:[2Fh*4], ax
            mov     ax, word  ptr OldInt2F+2
            mov     es:[2Fh*4+2], ax

;Okay , ostatnia rzecz przed wyjściem – Podajemy zaalokowaną pamięć dla tego TSR'a z powrotem do DOS

            mov     ds., PSP
            mov     es,  ds.:[2Ch]                  ;Wskaźnik do bloku środowiska
            mov     ah, 49h
            int     21h
```

```
                mov     ax, ds.                         ;zwolnienie przestrzeni kodu programu
                mov     es, ax
                mov     ah, 49h
                int     21h

                popa
                pop     ds.
                pop     es
                mov     ax, 0                           ;zwraca powodzenie

; Wkładamy BadKey tutaj, aby  zamknąć jego powiązany skok (poniżej)
;
; Jeśli przychodzi tu, odkrywamy zaalokowany blok z określonym key. Zwraca kod błędu (AX =1)
; i rozmiar tego zaalokowanego bloku (w CX)

BadKey:         mov     cx, [bx].Region.BlkSize
                mov     ax, 1                           ;już zaalokowany błąd
                pop     bx
                pop     ds.
                iret
;zobaczmy czy jest to funkcja shmalloc
; jeśli tak, na wejściu –
; DX zawiera key
; CX zawiera liczbę bajtów do zaalokowania
;
; na wyjściu :
;
; ES:DI wskazują na alokowany blok (jeśli pomyślnie)
; CX zawiera aktualny rozmiar alokowanego bloku )>=CX na wyjściu)
; AX zawiera kod błędu, 0 jeśli nie ma błędu

Tryshmalloc:    cmp     al., 11h                ;kod funkcji shmalloc
                jne     Tryshmfree

;najpierw, przeszukujemy całą alokowaną listę aby zobaczyć czy blok z aktualnym numerem key'a
; już istnieje. DX zawiera żądany klucz .
                assume  ds.: SharedMemory
                assume  bx: ptr Region
                assume  di: ptr Region

                push    ds
                push    bx
                mov     bx, SharedMemory
                mov     ds, bx
                mov     bx, ResidentSeg: AllocatedList
                test    bx, bx                          ;coś na tej liście?
                je      SrchFreeList

SearchLoop:     cmp     dx, [bx]. Key                   ;czy klucz już istnieje?
                je      BadKey
                mov     bx, [bx].Next                   ;pobranie kolejnego regionu
                test    bx, bx                          ;NULL? Jeśli nie spróbuj inne
                jne     SearchLoop                      ;wejście na liście

;Jeśli  alokowany blok z określonym key'em nie istnieje, wtedy próbujemy alokować jeden z listy wolnej pamięci
```

```
SrchFreeList:   mov   bx, ResidentSeg: FreeList
                test  bx, bx                    ; Lista pusta?
                Je    OutaMemory

FirstFitLp:     cmp   cx, [bx].BlkSize          ;czy ten blok jest wystarczająco duży?
                jbe   GotBlock
                mov   bx, [bx].Next             ;jeśli nie, następny
                test  bx, bx                    ;czy cos jeszcze na liście?
                jne   FirstFitLp

;Jeśli znaleźliśmy się tutaj ,nie byliśmy w stanie znaleźć bloku, który był wystarczająco duży  aby spełnić żądanie.
; Zwraca właściwy błąd

OutaMemory:     mov   cx, 0                     , nic nie dostępne
                mov   ax, 2                     ; błąd niewystarczającej pamięci
                pop   bx
                pop   ds.
                iret

;Jeśli znajdziemy dość duży blok, możemy wyciąć z niego nowy blok i zwrócić resztę pamięci do listy wolnej
; pamięci. Jeśli wolny blok jest przynajmniej  32 bajty większy niż żądany rozmiar, zrobimy to . Jeśli
; wolny blok jest mniejszy niż 32 bajty, po prostu dajemy ten wolny blok do żądanego procesu. Powód 32 bajtowości
; jest prosty: Potrzebujemy ośmiu bajtów dla nowego nagłówka bloku (wolny blok ma już jeden) i nie ma sensu
; rozkładanie bloków na rozmiar poniżej 24 bajtów. To zwiększałoby czas przetwarzania, kiedy procesy zwalniają
; bloki przez wymaganie większej pracy przy łączenie bloków.

GotBlock:       mov   ax, [bx].BlkSize          ;obliczenie różnicy w rozmiarze
                sub   ax, cx
                cmp   ax, 32                    ;przynamniej 32 bajty?
                jbe   GrabWholeBlk              ;jeśli nie bierzemy ten blok

; Okay, wolny blok jest większy niż wymagany rozmiar 32 bajtów. Wycinamy nowy blok z końca wolnego bloku
; (w ten sposób nie musimy zmieniać wskaźników wolnego bloku, tylko rozmiar)

                mov   di, bx
                add   di, [bx]. BlkSize         ;skok na koniec, minus 8
                sub   di, cx                    ; wskazuje nowy blok

                sub   [bx].BlkSize, cx          ;usuwamy zaalokowany blok I
                sub   [bx].BlkSize, 8           ;miejsce na nagłówek

                mov   [di].BlkSize, cx          ;zachowanie rozmiaru bloku
                mov   [di].Key, dx              ;zachowanie key'a

;Przyłączamy nowy blok do listy zaalokowanych bloków

                mov   bx, ResidentSeg:AllocatedList
                mov   [di].Next, bx
                mov   [di].Prev, NULL           ;NULL poprzedni wskaźnik
                test  bx, bx                    ;zobaczmy czy była pusta lista
                je    NoPrev
                mov   [bx].Prev, di             ;ustawimy poprzedni wskaźnik dla starego
NoPrev:         mov   residentSeg:AllocatedList, di
RmvDone;        add   di, 8                     ;wskazuje aktualny obszar danych
                mov   ax, ds                    ; zwraca wskaxnik w es:di
                mov   es, ax
```

```
                mov     ax, 0                   ;zwraca powodzenie
                pop     bx
                pop     ds.
                iret
; Jeśli bieżący wolny blok jest większy niż żądany, ale nie większy niż 32 bajty, dajemy użytkownikowi cały blok

GrabWholeBlk:   mov     di, bx
                mov     cx, [bx].BlkSize        ;zwraca aktualny rozmiar
                cmp     [bx].Prev, NULL         ;pierwszy człon na liście?
                je      Rmvlst
                cmp     [bx].Next, NULL         ;Ostatni człon na liście?
                je      RmvLast

;Okaz, rekord ten jest wciśnięty między dwa inne w liście. Wycinamy go z pomiędzy nich

                mov     ax, [bx].Next           ;zachowujemy wskaźnik do kolejnej
                mov     bx, [bx].Prev           ; pozycji poprzedniej pozycji w kolejnym
                mov     [bx].Next, ax           ; polu

                mov     ax, bx                  ;zachowujemy wskaźnik do poprzedniej
                mov     bx, [di].Next           ; pozycji w następnej pozycji poprzedniego
                mov     [bx].Prev, bx           ;pola
                jmp     RmvDone

;Blok jaki chcemy usunąć jest na początku listy wolnych bloków. Może być również jedynie pozycją na liście!

RmvLast:        mov     ax, [bx].Next
                mov     FreeList, ax            ;usuwanie z listy wolnych bloków
                jmp     RmvDone

;Jeśli blok jaki chcemy usunąć jest na końcu listy obsługujemy ten tu

RmvLast:        mov     bx, [bx].Prev
                mov     [bx].Next, NULL
                jmp     RmvDone

                assume  ds: nothing, bx:nothing, di:nothing

; ten kod obsługuje funkcję SHMFREE. Na wejściu DX zawiera key dla zwalnianego bloku , musimy przeszukać
; całą listę zaalokowanych bloków i znaleźć blok z tym key'em. Jeśli nie znajdziemy takiego bloku, kod ten wróci
; bez wykonania czegokolwiek. Jeśli znajdziemy blok, musimy dodać jego pamięć do wspólnego wolnego obszaru
; Jednakże, nie możemy po prostu wstawić tego bloku na początku listy wolnych bloków (jaki robiliśmy dla bloków
; alokowanych). Możemy założyć, że ten blok zwolniony jest przyległy do jednego lub dwóch innych wolnych
; bloków. Kod ten musi połączyć takie bloki w pojedynczy wolny blok.

Tryshmfree:     cmp     al., 12h
                jne     Tryshminit

;najpierw, przeszukamy  listę zaalokowanych bloków aby sprawdzić czy możemy znaleźć blok do usunięcia. Jeśli
; nie znajdziemy go na tej liście nigdzie, powrót

                assume  ds: SharedMemory
                assume  bx: ptr Region
                assume  di: ptr egion

                push    ds
```

```
                push    di
                push    bx

                mov     bx, SharedMemory
                mov     ds, bx
                mov     bx, ResidentSeg: AllocatedList

                test    bx, bx                  ;czy pusta lista alokacji?
                je      FreeDone
SrchList:       cmp     dx, [bx].Key            ;przeszukanie dla key'a w DX
                je      FoundIt
                mov     bx, [bx].Next
                test    bx, bx                  ;czy koniec listy?
                jne     SrchList
FreeDone:       pop     bx
                pop     di                      ; nic zaalokowanego, więc powrót do
                pop     ds.                     ; kodu wywołującego
                iret
;Okay znaleźliśmy blok jaki użytkownik chce usunąć. Usuwamy go z listy alokacji. Są trzy przypadki do
; rozpatrzenia: (1) jest na początku  listy alokacji, (2) jest  na końcu listy alokacji i (3) jest w środku listy
; alokacji

FoundIt:        cmp     [bx].Prev, NULL         ;pierwsza pozycja na liście?
                je      Freelst
                cmp     [bx].Next, NULL         ;ostatnia pozycja na liście?
                je      FreeLast

;Okay, usuwamy zaalokowaną pozycję ze środka listy alokacji

                mov     di, [bx].Next           ;[next].prev := [cur].prev
                mov     ax, [bx].Prev
                mov     [di].Prev, ax
                xchg    ax, di
                mov     [di].Next, ax           ;[prev].next := [cur].next
                jmp     AddFree

;Obsłużymy przypadek gdzie usuwamy pierwszą pozycję z listy alokacji. Jest to możliwe, że jest to jedyna pozycja
; na liście (tj. jest to pierwsza i ostatnia pozycja na liście), ale ten kod obsługuje przypadek bez takich problemów

Freelst:        mov     ax, [bx].Next
                mov     ResidentSeg:AllocatedList, ax
                jmp     AddFree

;Jeśli  usuwamy ostatni człon w łańcuchu, po prostu ustawiamy następne pole poprzedniego węzła w liście na NULL

FreeLast:       mov     di, [bx].Prev
                Mov     [di].next, NULL

;Okay, teraz możemy włożyć zwolniony blok do listy wolnych bloków.  Lista wolnych bloków jest posortowana
; według adresów. Musimy wyszukać pierwszy wolny blok, którego adres jest większy niż  blok, który właśnie
; zwolniliśmy i wprowadzić nowy  wolny blok  przed nim. Jeśli dwa bloki są przyległe, wtedy musimy je podzielić
; na pojedyncze wolne bloki. Również, jeśli blok przed jest przyległy, musimy podzielić go. To połączy wszystkie
; wolne bloki w liście wolnych bloków więc jest kilka wolnych bloków możliwych, a bloki te są tak duże jak to
; możliwe

AddFree:        mov     ax, ResidentSeg :FreeeList
```

```
                test    ax,ax                   ,Pusta lista?
                jne     SrchPosn

;Jeśli  lista jest pusta, zlepimy te człony jedynie na wejściu

                mov     ResidentSeg: FreeList, bx
                mov     [bx].Next, NULL
                mov     [bx].Prev, NULL
                jmp     FreeDone
; Jeśli lista wolnych bloków nie jest pusta, wyszukujemy pozycję tego bloku na liście:

SrchPosn:       mov     di, ax
                cmp     bx, di
                jb      FoundPosn
                mov     ax, [di].Next
                test    ax, ax                  ;Koniec listy?
                jne     SrchPosn

;Jeśli jesteśmy tu, znaczy ,że wolny blok należy do końca listy. Zobaczymy czy musimy podzielić
; nowy blok ze starym

                mov     ax, di
                add     ax, [di].BlkSize        ;obliczamy adres pierwszego
                add     ax, 8                   ; bajtu po tym bloku
                cmp     ax, bx
                je      MergeLast

;Okay, właśnie dodajemy wolny  blok do końca listy

                mov     [di].Next, bx
                mov     [bx].Prev, di
                mov     [bx].Next, NULL
                jmp     FreeDone

;Dzielimy zwolniony blok z blokiem wskazywanym przez DI

MergeLast:      mov     ax, [di].Blksize
                add     ax, [bx].BlkSize
                add     ax, 8
                mov     [di].BlkSize, ax
                jmp     FreeDone
; Jeśli znaleźliśmy wolny blok zanim przypuszczalnie wprowadziliśmy aktualny wolny blok, wrzucamy go tu i
; obsługujemy

FoundPos:       mov     ax, bx                  ;obliczamy adres kolejnego bloku w pamięci
                add     ax, [bx].BlkSize
                add     ax,8
                cmp     ax, di                  ; równe temu blokowi?
                jne     DontMerge

; Jeśli  kolejny wolny blok jest przyległy do jednego ze zwolnionych, wiec dzielimy dwa

                mov     ax, [di].BlkSize        ;dzielimy rozmiary razem
                add     ax, 8
                add     [bx].BlkSize, ax
                mov     ax, [di].Next
```

```
                mov     [bx].Next, ax
                mov     ax, [di].Prev
                mov     [bx].Prev, ax
                jmp     tryMergeB4

;Jeśli nie są przyległe, łączymy je tutaj razem

DontMerge:      mov     ax, [di].Prev
                mov     [di].Prev,bx
                mov     [bx].Prev, ax
                mov     [bx].Next, di

;Teraz zobaczymy czy możemy podzielić aktualny wolny blok z poprzednim wolnym blokiem

TryMergeB4:     mov     di, [bx].Prev
                mov     ax, di
                add     ax, [di].BlkSize
                add     ax, 8
                cmp     ax, bx
                je      CanMerge
                pop     bx
                pop     di                      ;Nic zaalokowanego, wracamy
                pop     ds.                     ;do kodu wywołującego
                iret

; Jeśli możemy podzielić poprzedni i aktualny wolny blok, robimy to tutaj:

CanMerge:       mov     ax, [bx].Next
                mov     [di.Next, ax
                mov     ax, [bx].BlkSize
                add     ax, 8
                add     [di].BlkSize, ax
                pop     bx
                pop     di
                pop     ds
                iret

                assume  ds:nothing
                assume  bx:nothing
                assume  di:nothing

; Tutaj obsługujemy funkcję inicjalizacyjną (SHMINIT) dzielonej pamięci. Wszystko co musimy zrobić to stworzyć
; pojedynczy blok w liście wolnych bloków (cała dostępna pamięć), opróżnić listę alokacji i wyzerować całą
; dzieloną pamięć

Tryshminit:     cmp     al., 13h
                jne     TryShmAttach

; Resetujemy obszar alokacji pamięci zawierający pojedynczy, wolny blok pamięci, którego rozmiar to 0FFF8h
; (musimy zarezerwować osiem bajtów dla struktury danych bloku)

                push    es
                push    di
                push    cx

                mov     ax, SharedMemory        ;zerujemy segment pamięci dzielonej
```

```
                        mov     es, ax
                        mov     cx, 32768
                        xor     ax, ax
                        mov     di, ax
                rep     stosw

;Notka: zakomentowane poniższe linie nie są konieczne ponieważ powyższy kod wyzerował już
; cały segment pamięci dzielonej. Notka: nie możemy odłożyć pierwszego rekordu pod offsetem zero ponieważ
; zero jest to specjalna wartość dla wskaźnika NULL. Zamiast tego użyjemy 4
                        mov     di, 4
;                       mov     es:[di].Region.Key, 0           ;Key jest  arbitralny
;                       mov     es:[di].Region.Next ,0          ; Żadnych innych wejść
;                       mov     es:[di}.Region.Prev, 0          ; jak wyżej
;                       mov     es:[di].Region.BlkSize, 0FFF8h  ;Reszta segmentu
                        mov     ResidentSeg:FreeList, di

                        pop     cx
                        pop     di
                        pop     es
                        mov     ax, 0                   ; nie zwrócono błędu
                        iret
;Funkcję SHMATTACH obsługujemy tutaj. Na wejściu, DX zawiera numer key'a. Przeszukujemy zaalokowany
; blok z tym numerem key'a  i zwracamy wskaźnik do tego bloku (jeśli znaleziono) w ES:DI. Zwracamy kod błędu
; jeśli nie można znaleźć bloku

TryShmAttach:           cmp     al., 14h                ;opcod przyłączenia
                        jne     IllegalOp
                        mov     ax, SharedMemory
                        mov     es, ax

                        mov     di, ResidentSeg:AllocatedList
FindOurs:               cmp     dx, es:[di].Region.Key
                        je      FoundOurs
                        mov     di, es:[di].Region.Next
                        test    di, di
                        jne     FoundOurs
                        mov     ax, 3                   ;nie można znaleźć key'a
                        iret

;wywoływanie z niepoprawną wartością funkcji. Spróbujemy zrobić jak najmniej szkód jak to możliwe

IllegalOp:      mov     ax, 0                   ;Kto wie co to ma być?
                iret
MyInt2F         endp
                assume  ds:nothing
ResidentSeg     ends

;tutaj jest segment w którym będziemy przechowywać dzielone dane

SharedMemory    segment   para public 'Shared'
        db      0FFFFh   dup  (?)
SharedMemory    ends

cseg            segment para  public ,code'
                assume  cs:cseg, ds:ResidentSeg
```

```
; SeeIfPresent-     Sprawdza aby zobaczyć czy nasz TSR jest już obecny w pamięci. Ustawia flagę zera jeśli
;                  jest, zeruje flagę zera jeśli nie ma
;

SeeIfPresent       proc    near
                   push    es
                   push    ds.
                   push    di
                   mov     cx, 0ffh                ;start z ID 0FFh
IDLoop:            mov     ah, cl
                   push    cx
                   mov     al, 0                   ;weryfikacja obecności
                   int     2Fh
                   pop     cx
                   cmp     al., 0                  ; obecny w pamięci?
                   je      TryNext
                   strcmpl
                   byte    „Dynamic Shared Memory TSR”, 0
                   je      Success

TryNext:           dec     cl                      ;testuje ID użytkownika 80h..FFh
                   js      IDLoop
                   cmp     cx, 0                   ;zerowanie  flagi zera
Success:           pop     di
                   pop     ds.
                   pop     es
                   ret
SeeIfPresent       endp

;FindID-           Określamy pierwszy (cóż w rzeczywistości ostatni) ID TSR’a dostępny w łańcuchu
;                  równoczesnych procesów. Zwraca tą wartość w rejestrze CL.
;
;                  Zwraca ustawioną flagę zera jeśli  lokuje pusty slot
;                  Zwraca wyzerowaną flagę zera jeśli niepowodzenie

FindID             proc    near
                   push    es
                   push    ds.
                   push    di

                   mov     cx, 0ffh                ;start z ID 0FFh
IDLoop:            mov     ah, cl
                   push    cx
                   mov     al, 0                   ; weryfikacja obecności
                   int     2Fh
                   pop     cx
                   cmp     al., 0                  ;obecny w pamięci?
                   je      Success
                   dec     cl                      ; test ID użytkownika 80h..FFh
                   js      IDLoop
                   xor     cx, cx
                   cmp     cx, 1                   ; zerowanie flagi zera
Success:           pop     di
                   pop     ds.
                   pop     es
                   ret
```

```
FindID          endp

Main            proc
                meminit

                mov     ax, ResidentSeg
                mov     ds., ax

                mov     ah, 62h                 ;pobranie wartości PSP programu
                int     21h
                mov     PSP, bx

; Zanim zrobimy cokolwiek, musimy sprawdzić parametry linii poleceń. Jeśli jest jeden, i jest to
; słowo „REMOVE”, wtedy usuwamy rezydentną kopię z pamięci używając przerwania równoczesnych
; procesów

                argc
                cmp     cx, 1                   ;musi mieć 0 lub 1 parametr
                jb      TstPresent
                je      DoRemove

Usage:          print
                byte    „Usage:”, cr, lf
                byte    “shmalloc”, cr, lf
                byte    “ or shmalloc REMOVE”,cer,lf,0
                ExitPgm

;Sprawdzenie na polecenie REMOVE

DoRemove        mov     ax, 1
                argv
                strimcpl
                byte    “Remove”, 0
                jne     Usage

                call    SeeIfPresent
                je      RemoveIt
                print
                byte    “TSR is not present in memory, cannot remove”
                byte    cr, lf,0
                ExitPgm

RemoveIt:       mov     MyTSRID, cl
                printf
                byte    “rremoving TSR (ID #%d) from memory…..”, 0
                dword   MyTSRID

                mov     ah, cl
                mov     al, 1           ;usuwanie cmd, ah zawiera ID
                int     2Fh
                cmp     al., 1          ;Powodzenie?
                je      RmvFailure
                print
                byte    „removed”,cr,lf,0
                ExitPgm
```

```
RmvFailure:     print
                byte    cr, lf
                byte    "Could not remove TSR from memmory",cr,lf
                byte    "Try removing inne TSR'y in reverse order"
                byte    „you installed them",cr,lf,0
                ExitPgm

;Okay, zobaczmy czy nasz TS jest już w pamięci. Jeśli tak, przerywamy proces instalacji

TstPresent:     call    SeeIfPresent
                jne     GetTSRID
                print
                byte    „TSR jest już obecny w pamięci ",cr,lf
                byte    „Przerwanie procesu instalacji",cr,lf,0
                ExitPgm

; Pobranie ID dla naszego TSR'a i zachowanie go

GetTSRID:       call    FindID
                je      GetFileName
                print
                byte    „Zbyt wiele rezydentnych TSR'ów, nie można instalować",cr,lf,0
                ExitPgm
; Instalujemy przerwania

GetFileName:    mov     MyTSRID, cl
                print
                byte    "Instalowanie przerwań…", 0

;Aktualizacja łańcucha przerwań INT 2Fh

                cli                             ;wyłączamy przerwania
                mov     ax, 0
                mov     es, ax
                mov     ax, es:[2Fh*4]
                mov     word  ptr  OldInt2F, ax
                mov     ax, es:[2Fh*4+2]
                mov     word  ptr OldInt2F+2, ax
                mov     es:[2Fh*4], offset MyInt2F
                mov     es:[2Fh*4+2], seg ResidentSeg
                sti                             ;Ok , włączamy przerwania

; Jedyna rzecz jak nam pozostała to inicjalizacja segmentu pamięci dzielonej a potem TSR

                printf
                byte    „Instalowanie , TSR ID #%d.",cr,lf,0
                dword   MyTSRID

                mov     ah, MyTSRID             ;funkcja inicjalizująca
                mov     al, 13h
                int     2Fh

                mov     dx, EndResident         ;oblicza rozmiar programu
                sub     dx, PSP
                mov     ax, 3100h               ;polecenie TSR DOS'a
                int     21h
```

```
Main            endp
cseg            ends

sseg            segment  para  stack ‘stack’
stk             db       256   dup   (?)
sseg            ends

zzzzzzseg       segment  para  public  ‘zzzzzz’
LastBytes       db       16    dup   (?)
zzzzzzseg       ends
                end      Main
```

Możemy zmodyfikować dwoi aplikacje z poprzedniej sekcji próbując takiego kodu:

```
;SHMAPP3.ASM
;
; Jest to aplikacja z dzieloną pamięcią, która używa dynamicznie dzielonej pamięci TSR (SHMALLOC.ASM)
; Program ten wprowadza ciąg od użytkownika i przekazuje ten ciąg do SHMAPP4.ASM w całym obszarze
; pamięci dzielonej.

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                ;list

dseg            segment para  public  ‘data’
ShmID           byte    0
dseg            ends

cseg            segemnt para public  ‘code’
                assume  cs:cseg, ds: dseg, es:SharedMemory

; SeeIfPresent-         Sprawdza aby zobaczyć czy TSR pamięci dzielonej jest obecny w pamięci
;                       Ustawia flagę zera jeśli jest, zeruje flagę zera jeśli nie. Ten podprogram
;                       również zwraca ID TSR’a w CL

SeeIfPresent    proc    near
                push    es
                push    ds
                push    di
                mov     cx, 0ffh                        ;start z ID 0FFH
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                           ;weryfikacja obecności
                int     2Fh
                pop     cx
                cmp     al., 0                          ;obecny w pamięci?
                je      TryNext
                strcmpl
                byte    „Dynamic Shared Memory TSR”, 0
                je      Success
TryNext:        dec     cl                      ;test ID użytkownika 80f..FFh
                js      IDLoop
                cmp     cx, 0                   ;zerowanie flagi zera
Success:        pop     di
                pop     ds.
```

```
                pop     es
                ret
SeeIfPresent    endp

; Program główny dla aplikacji #1 łączy TSR pamięci dzielonej a potem odczytuje ciąg od użytkownika.
; (przechowując ciąg w pamięci dzielonej) a potem kończy

Main            proc
                assume  cs:cseg, ds.:dseg, es:SharedMemory
                mov     ax, dseg
                mov     ds, ax
                meminit
                print
                byte    "Shared memory application #3",cr,lf,0

;zobaczmy czy jest TSR pamięci dzielonej:

                call    SeeIfPresent
                je      ItsThere
                print
                byte    „Shared Memory TSR (SHMALLOC)  nie jest załadowany ",cr,lf
                byte    „Ten program nie  może kontynuować wykonywania",cr,lf,0
                ExitPgm

; pobranie lini wejściowej od użytkownika

ItsThere:       mov     ShmID, cl
                print
                byte    "Wprowadź ciąg: ",0

                lea     di, InputLine           ;ES już wskazuje właściwy segment
                getsm

; Ciąg jest w naszej przestrzeni sterty. Przesuniemy go ponad segment pamięci dzielonej

                strlen
                inc     cx                      ;dodajemy jeden do zera bajtów
                push    es
                push    di

                mov     dx,1234h                ; wartość "naszego" key'a
                mov     ah, ShmID
                mov     al., 11h                ;funkcja shmalloc
                int     2Fh

                mov     si, di                  ; zachowujemy jako wskaźnik przeznaczenia
                mov     dx, es

                pop     di                      ; odzyskanie adresu źródłowego
                pop     es
                strcpy                          ;kopiujemy z lokalnego do dzielonego

                print
                byte    „Wprowadzono '", 0
                puts
                print
```

```
                byte    „’ do pamięci dzielonej”, cr,lf,0

Quit:           ExitPgm                 ;makro DOS’a do wyjścia z programu
Main            endp

cseg            ends

sseg            segemnt para  stack ‘stack’
stk             db      1024  dup (“stack”)
sseg            ends

zzzzzzseg       segment para  public ‘zzzzzz’
LastBytes       db      16  dup (?)
Zzzzzzseg       ends
                end     Main



; SHMAPP4.ASM
;
; Jest to aplikacja z dzieloną pamięcią, która używa dynamicznie dzielonej pamięci TSR (SHMALLOC.ASM)
; Program ten zakłada, że użytkownik ma już uruchomiony program SHMAPP3 wprowadzający ciąg  do
; pamięci dzielonej. Program ten po prostu drukuje ten ciąg z pamięci dzielonej

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment para public ‘data’
ShmID           byte    0
dseg            ends

cseg            segment para  public  ‘code’
                assume  cs:cseg, ds:dseg, es:SharedMemory

; SeeIfPresent-  Sprawdzamy czy pamięć dzielona TSR jest obecna w pamięci. Ustawiamy flagę zera
;               jeśli jest, zeruje flagę zera jeśli nie. Ten podprogram również zwraca ID TSR’a w CL
;
SeeIfPresent    proc    near
                push    es
                push    ds.
                push    di
                mov     cx, 0ffh                ;start z ID 0FFH
IDLoop:         mov     ah, cl
                push    cx
                mov     al, 0                   ;funkcja weryfikacji obecności
                int     2Fh
                pop     cx
                cmp     al., 0                  ;obecny w pamięci?
                je      TryNext
                strcmpl
                byte    „Dynamic Shared Memory TSR”,0
                je      Success

TryNext:        dec     cl                      ;Test ID użytkownika 80h..FFh
```

```
                js      IDLoop
                cmp     cx, 0                   ;zerujemy flagę zera
Success:        pop     di
                pop     ds.
                pop     es
                ret
SeeIfPresent    endp

;Program główny dla aplikacji #1 łączy pamięć dzieloną TSR a potem odczytuje ciąg od użytkownika
; (przechowywany w pamięci dzielonej) a potem kończy

Main    proc
        assume  cs:cseg, ds.:dseg, es: SharedMemory
        mov     ax, dseg
        mov     ds,ax
        meminit

        print
        byte    "shared mempory application #4", cr,lf,0

;zobaczmy czy jest pamięć dzielona TSR

                call    SeeIfPresent
                je      ItsThere
                print
                byte    „Pamięć dzielona TSR (SHMALLOC) nie jest załadowana" ,cr, lf
                byte    „Program ten nie może kontynuować wykonywania", cr,lf,0
                ExitPgm

;Jeśli pamięć dzielona TSR jest obecna, pobieramy adres dzielonego segmentu do rejestru ES:

ItsThere:       mov     ah, cl                  ;ID naszego TSR'a
                mov     al, 14h                 ;funkcja łączenia
                mov     dx, 1234h               ;wartość naszego key'a
                int     2Fh

; Drukujemy ciąg

                print
                byte    „Ciąg z SHMAPP3 to ' „, 0
                puts
                print
                byte    „ ' z pamięci dzielonej ",cr, lf,0

Quit            ExitPgm
Main            endp

cseg            ends

sseg            segemnt para stack 'stack'
stk             db      1024  dup  ("stack")
sseg            ends

zzzzzzseg       segemnt para public 'zzzzzz'
LastBytes       db      16  dup (?)
Zzzzzzseg       ends
```

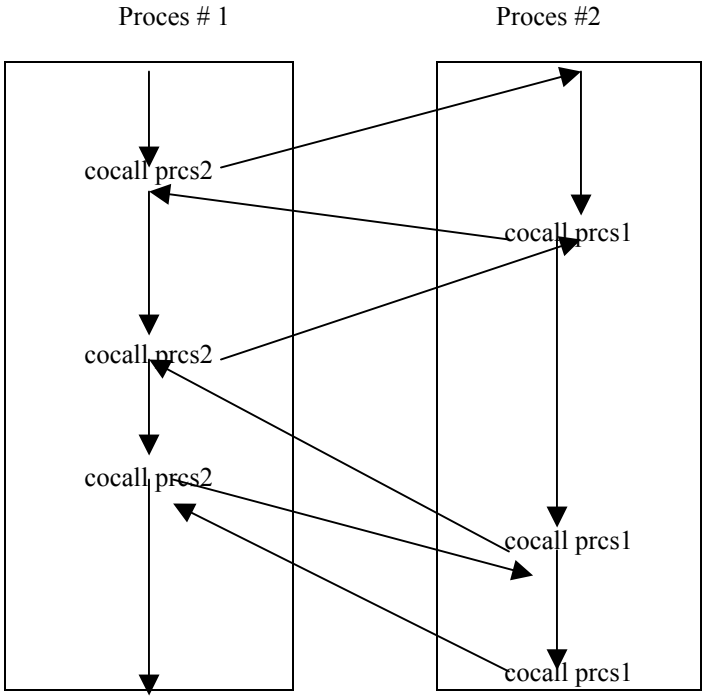
```
end     Main
```

## 19.3 WSPÓŁPROGRAMY

Procesy DOS, nawet kiedy używają pamięci dzielonej, cierpią z powodu jednej poważnej wady – każdy program wykonuje się do końca zanim zwróci sterowanie do procesu macierzystego. Chociaż taki paradygmat jest odpowiedni dla wielu aplikacji, z pewnością nie jest wystarczający dla wszystkich. Popularnym paradygmatem dla dwóch programów jest wymiana sterowania z CPU tam i z powrotem podczas wykonywania. Mechanizm ten, nieznacznie różni się od wywołania podprogramów i mechanizmu powrotu, to współprogram.

Przed omówieniem współprogramów, dobrym pomysłem jest dostarczenie solidnej definicji dla terminu proces. W dużym skrócie, proces jest to program, który jest wykonywany. Program może istnieć na dysku; procesy istnieją w pamięci i mają stos programu (z adresem powrotnym itd.) powiązany z nimi. Jeśli jest wiele procesów w pamięci w tym samym czasie, każdy program musi mieć swój własny stos programu.

Operacja współwywołania przekazuje sterowanie pomiędzy dwoma procesami. Współwywołanie jest skutecznym wywołaniem i zwraca instrukcje wszystkie skierowane na jedna operację. Z punktu widzenia procesu wykonującego współwywołanie, operacja współwywołania jest odpowiednikiem procedury call; z punktu widzenia procesu będącego wywoływanym, operacja współwywołania jest odpowiednikiem operacji powrotu . Kiedy drugi proces współwywołuje pierwszy, sterowanie nie rozpoczyna się od początku pierwszego procesu, ale bezpośrednio po operacji współwywołania . Jeśli dwa procesy wykonują sekwencję wzajemnych współwywołań, sterowanie będzie przekazywane między dwoma procesami w następujący sposób:




Sekwencja współwywołania pomiędzy dwoma procesami

Współwywołania są całkiem użyteczne przy grach, gdzie „gracze” jeden po drugim, wywołuje różne strategie. Pierwszy gracz wykonuje jakiś kod robiąc pierwszy ruch, potem współwywołuje drugiego gracza i zezwala na wykonanie ruchu. Po drugim graczu, który wykonał swój ruch, współwywołuje pierwszy proces i daje pierwszemu graczowi drugi ruch, bezpośrednio po jego współwywołaniu. Takie przekazywanie sterowania występuje dopóki jeden gracz nie wygra.

CPU 80x86 nie dostarczają instrukcji współwywołania. Jednakże, łatwo jest zaimplementować współwywołania z istniejących instrukcji. Mimo to, istnieje potrzeba dostarczenia własnego mechanizmu współwywołania, Biblioteka Standardowa UCR dostarcza pakietu współwywołania dla procesorów 8086 80186 i

80286. Do tego pakietu zaliczają się struktura danych pcb (blok sterowani procesem) i trzy funkcje jakie możemy wywołać: coinit, cocall i cocall1.

Struktura pcb utrzymuje bieżący stan procesu. Utrzymuje wszystkie wartości rejestrów i inne liczące się informacje dla procesu. Kiedy proces dokonuje współwywołania, przechowuje adres powrotu dla współwywołania w pcb. Później, kiedy jakiś inny proces współwywoła ten proces, operacja współwywołania po prostu przeładuje rejestry, wliczając w to cs:ip, z pcb, i zwróci sterowanie do następnej instrukcji po współwywołaniu pierwszego procesu .Struktura pcb przybiera następującą postać:

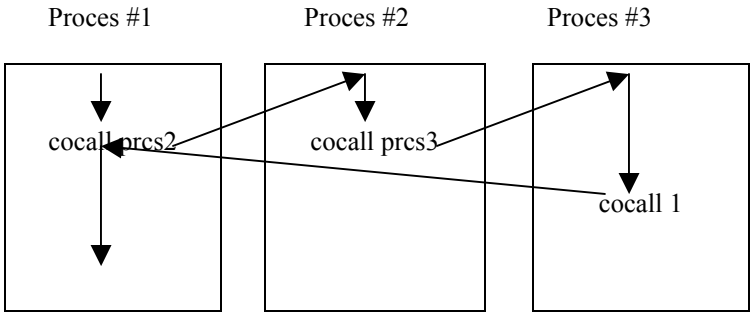
```
pcb     struct

NextProc        dword   ?               ;łącze do następnego PCB (przy wielozadaniowości)
regsp           word    ?
regss           word    ?
regip           word    ?
regcs           word    ?
regax           word    ?
regbx           word    ?
regcx           word    ?
regdx           word    ?
regsi           word    ?
regdi           word    ?
regbp           word    ?
regds           word    ?
reges           word    ?
regflags        word    ?
PrcsID          word    ?
StartingTime    dword   ?
StartingDate    dword   ?
CPUTime         dword   ?
```

Cztery z tych pół istnieje dla wielozadaniowości z wywłaszczeniem i nie ma znaczenia przy współprogramach. Będziemy omawiali wielozadaniowość z wywłaszczeniem w następnej sekcji.

Są dwie ważne rzeczy, które powinny być widoczne z tej struktury. Po pierwsze, głównym powodem istnienia wsparcia przez Bibliotekę Standardową współprogramów jest ograniczenie do 16 bitowych rejestrów ponieważ jest tylko miejsce dla 16 bitowych wersji dla każdego rejestru w pcb. Jeśli chcemy wesprzeć 80386 i późniejsze 32 bitowe zbiory rejestrów, będziemy musieli zmodyfikować strukturę pcb i kod, który zachowuje i przywraca rejestry w pcb.

Druga rzecz jaka powinna być widoczna, jest to ,że kod wpółpółgoramu zachowuje wszystkie rejestry w poprzek współwywołania. To znaczy ,że nie możemy przekazać informacji z jednego procesu do drugiego w rejestrach kiedy używamy współwywołania. Będziemy musieli przekazać dane pomiędzy procesami w lokacji globalnej pamięci. Ponieważ współprogram generalnie istnieje w tym samym programie, nie będziemy musieli uciekać się do technik pamięci dzielonej. Zmienne jakie zadeklarujemy w segmencie danych będą widoczne dla wszystkich współprogramów.

Odnotujmy, że program może zawierać więcej niż dwa współprogramy. Jeśli współprogram jeden współwywołuje współprogram dwa, a współprogram dwa współwywołuje współprogram trzy, a potem współprogram trzy współwywołuje współprogram jeden, współprogram jeden wystąpi bezpośrednio po współwywołaniu go uczynionym przez współprogram trzy.




Współwywołanie pomiędzy trzema procesami

Ponieważ współwywołanie faktycznie wraca do współprogramu docelowego, możemy zastanowić się co się zdarzy przy pierwszym współwywołaniu jakiegoś procesu. W końcu, jeśli ten proces nie wykonywał żadnego kodu, nie ma „adresu powrotnego" gdzie rozpoczyna wykonanie. Jest to prosty problem do rozwiązania, musimy tylko zainicjalizować adres powrotny takiego procesu, adresując pierwszą instrukcję do wykonania w tym procesie.

Podobny problem istnieje dla stosu. Kiedy program zaczyna wykonywania, program główny (współprogram jeden) pobiera sterowanie i używa stosu związanego z całym programem. Ponieważ każdy proces musi mieć swój własny stos, gdzie inne współprogramy mają swoje stosy?

Najłatwiejszym sposobem zainicjalizowania stosu i początkowego adresu dla współprogramu, jest zrobienie tego kiedy deklarujemy pcb dla procesu. Rozważmy następującą deklarację zmiennej pcb:

```
ProcessTwo      pcb     {0,     offset  EndStack2, seg EndStack2,
                                offset  StartLoc2, seg StarLoc2}
```

Definicja ta inicjalizuje pole NextProc NULL'em ( funkcja współprogramu Biblioteki Standardowej nie używa tego pola) i inicjalizuje pola ss:sp i cs:ip ostatnim adresem obszaru stosu (EndStack2) i pierwszą instrukcją procesu (StartLoc2 ). Teraz co musimy zrobić to zarezerwować rozsądną ilość pamięci stosu dla procesu. Możemy stworzyć wiele stosów w sseg SHELL.ASM jak poniżej:

```
sseg            segment para  stack 'stack'

;Stos dla procesu #2:

stk2            byte    1024   dup (?)
EndStack2       word    ?

:Stos dla procesu #3:

stk3            byte    1024  d up(?)
EndStack3       word    ?

;Pierwszy stos dla programu głównego (proces #1) musi pojawić się na końcu sseg

stk             byte    1024   dup (?)
sseg            ends
```

Teraz jest pytanie „jak dużo miejsca powinniśmy zarezerwować dla każdego stosu?". To jest różnie z aplikacjami. Jeśli mamy prostą aplikację, która nie używa rekurencji lub alokuje zmienne lokalne na stosie można założyć najmniej 256 bajtów stosu dla procesu.. Z drugiej strony, jeśli mamy podprogramy rekurencyjne lub alokujemy pamięć na stosie, będziemy potrzebowali znacznie więcej miejsca. Dla prostego programu 1 –8 K pamięci stosu powinno być wystarczające. Zapamiętajmy, ze możemy zaalokować maksimum 64K w sseg SHELL.ASM. jeśli potrzebujemy dodatkowej przestrzeni stosu, będziemy musieli pożyczyć z innych stosów w różnych segmentach (nie muszą być w sseg, jest to konwencjonalne miejsce dla nich) lub będziemy musieli zaalokować inaczej przestrzeń stosu.

Zauważmy, że nie musimy alokować przestrzeni stosu jako tablicy wewnątrz naszego programu. Możemy również zaalokować przestrzeń stosu dynamicznie używając funkcji malloc Biblioteki Standardowej. Poniższy kod demonstruje jak ustawić 8K dynamicznie alokowanego stosu dla pcb zmiennej Proces2:

```
        mov     cx. 8192
        malloc
        jc      InsufficientRoom
        mov     Process2.ss, es
        mov     Process2.ss, di
```

Konfigurowanie wywoływanych współprogramów programu głównego jest całkiem łatwe. Jednakże, istnieje kwestia skonfigurowania pcb dla programu głównego. Nie możemy zainicjalizować pcb dla programu głównego w ten sam sposób jak inicjalizowaliśmy pcb dla innych procesów; jest już uruchomiony i ma poprawne wartości cs:ip i ss:sp. Gdybyśmy zainicjalizowali pcb głównego programu w ten sam sposób jak zrobiliśmy to dla innych procesów, wtedy system zrestartowałby po prostu program główny kiedy wykonalibyśmy współwywołanie z powrotem do niego. Przy inicjalizacji pcb dla programu głównego musimy użyć funkcji coinit Funkcja coinit oczekuje, że przekażemy jej adres pcb programu głównego w parze rejestrów es:di. Zainicjalizuje jakieś zmienne wewnątrz Biblioteki Standardowej aby pierwsza operacja współwywołania zachowała stan maszynowy 80x86 w pcb jaki określiliśmy w es:di. Po wywołaniu coinit, możemy zacząć wykonywać współwywołania do innych procesów w naszym programie.

Do współwywołania współprogramów używamy funkcji cocall z Biblioteki Standardowej. Wywołanie funkcji cocall przybiera dwie formy. Bez żadnych parametrów funkcja ta przekazuje sterowanie do współprogramu , którego adres pcb pojawia się w parze rejestrów es:di. Jeśli adres pcb pojawia się polu operandu tej instrukcji, cocall przekazuje sterowanie do określonego współprogramu (nie zapomnij , nazwa pcb, nie procesu, musi pojawić się w polu operandu)

Najlepszy sposób nauczenia się jak stosować współprogramy jest poprzez przykład. Następujący program jest interesującym kawałkiem kodu, który generuje labirynt na ekranie PC .Algorytm generowania labiryntu ma jedno ważne ograniczenie – musi być nie więcej niż jedno poprawne rozwiązanie. Program główny tworzy zbiór działających w tle procesów zwanych „demonami". Każdy demon wycina część tematu labiryntu głównego ograniczenia. Każdy demon wykopuje jedną komórkę z labiryntu a potem przekazuje sterowanie do innego demona. Okazuje się , że demony „same siebie mogą zagnać do kąta" i umrzeć (demony żyją tylko dla kopania). Kiedy to się zdarzy, demon usuwa się z listy aktywnych demonów. Kiedy wszystkie demony zginą, labirynt (teoretycznie) jest kompletny. Ponieważ demony giną dość regularnie, musi być jakiś mechanizm tworzenia nowych demonów. Dlatego też ten program losowo daje początek nowym demonom, które zaczynają kopanie swoich własnych tuneli pionowych do ich macierzystych. To pozwala założyć , że jest wystarczający zapas demonów do wykopania całego labiryntu; wszystkie demony zginą tylko wtedy kiedy niema, lub kilka, komórek pozostało do wykopania w labiryncie.

```
;AMAZE.ASM
;
; Program do wytworzenia / rozwiązania labiryntu
;
; Pogram generuje labirynt 80x25 i bezpośrednio rysuje labirynt na monitorze. Demonstruje zastosowanie
; współprogramów wewnątrz programu

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

byp             textequ  <byte   ptr>
dseg            segment para  public  'data'

; Stałe:
;
; Definiujemy symbol „ToScreen" dla jakieś wartości) jeśli labirynt  ma 80x25 i chcemy wyświetlić go na monitorze
```

```
ToScreen        equ     0

; Maksymalne współrzędne X i Y dla labiryntu (dopasowanie do wyświetlacza)

MaxXCoord       equ     80
MaxYCoord       equ     25

; Użyteczne stałe X,Y:

WordPerRow      =       MaxXCoord+2
BytePerRow      =       WordPerRow*2

StartX          equ     1                       ;początkowa współrzędna X dla labiryntu
StartY          equ     3                       ;początkowa współrzędna Y dla labiryntu
EndX            equ     MaxXCoord               ;końcowa współrzędna X dla labiryntu
EndY            equ     maxYCoord-1             ;końcowa współrzędna Y dla labiryntu

EndLoc          =       ((EndY-1)*MaxXCoord+End-1)*2
StartLoc        =       ((StartY-1)*MaxXCoord+StartX-1)*2

; Specjalne 16 bitowe kody znaków PC dla ekranu dla symboli malowanych podczas generowania labiryntu.
; Zobacz rozdział o monitorze komputera po szczegóły

                ifdef   mono                    ; monitor monochromatyczny
WallChar        equ     7dbh                    ; stały blok znaków
NoWallChar      equ     720h                    ; spacja
VisitChar       equ     72eh                    ; kropka
PathChar        equ     72ah                    ; gwiazdka

                else                            ;ekran kolorowy

WallChar        equ     1dbh                    ;stały blok znaków
NoWallChar      equ     0edbh                   ;spacja
VisitChar       equ     0bdbh                   ;kropka
PathChar        equ     4e2ah                   ;gwiazdka

                endif

; poniżej są stałe, które mogą pojawić się w tablicy Maze:

Wall            =       0
NoWall          =       1
Visited         =       2

;poniżej są kierunki w jakich mogą iść demony w labiryncie

North           =       0
South           =       1
East            =       2
West            =       3

;jakieś ważne zmienne

; Tablica Maze musi zawierać dodatkowe wiersze i kolumny wokół zewnętrznych brzegów aby
; nasz algorytm działał poprawnie
```

```
Maze            word    (MaxYCOord+2)  dup ((MaxXCoord+2)   dup (Wall))

;Poniższe makro oblicza indeks do powyższej tablicy zakładając, że współrzędne X I Y demona
; są, odpowiednio, w rejestrach dl i dh. Zwraca indeks w rejestrze AX.

MazeAdrs        macro
                mov     al., dh
                mov     ah, WordPerRow          ;indeks do tablicy jest obliczony
                mul     ah                      ; (Y*words / row+X)*2
                add     al, dl
                adc     ah, 0
                shl     ax, 1                   ;konwersja do indeksu bajtowego
                endm

;Poniższe makro oblicza indeks do tablicy ekranu, używając takich samych założeń jak powyżej.
; Zauważmy, że macierz ekranu to 80x25 podczas gdy macierz labiryntu to 82x27; Współrzędne X/Y w DL/DH
; to 1 .. 80 i 1..25 zamiast 0..79 i 0..24 (jak potrzebujemy). To makro poprawia to

SctrnAdrs       macro
                mov     al., dh
                dec     al
                mov     ah, MaxXCoord
                mul     ah
                add     al, dl
                adc     ah, 0
                dec     ax
                shl     ax, 1
                endm
; PCB dla programu głównego. Będzie to wywoływał ostatni żywy demon , kiedy zemrze

MainPCB         pcb     {}

;Lista 32 demonów

MaxDemons       =       32                              ;musi być potęga dwójki
ModDemons       =       MaxDemons –1                    ;maska dla obleczenia MOD

DemonList       pcb     MaxDemons  dup {( )}

DemonIndex      byte    0                               ;indeks do listy demonów
DemonCnt        byte    0                               ;Liczba demonów na liście

;Generator liczb losowych (będziemy używali naszego generatora liczb losowych zamiast z biblioteki
; standardowej ponieważ chcemy móc określać wartość początkowa

Seed            word    0

dseg            ends

;Poniżej mamy adres segmentowy monitora, zmieniamy do od 0B800h do 0B000h jeśli mamy monitor
; monochromatyczny zamiast kolorowego

ScreenSeg       segment at 0b800h
Screen          equ     this word                       ;nie generuj tu danej
ScreenSeg       ends
```

```
cseg            segment para public ‘code’
                assume  cs:cseg, ds:dseg

; całkowicie fałszywy generator liczb losowych, ale nie potrzebujemy więcej jak jednego dla tego programu
; Kod ten używa swojego własnego generatora liczb losowych zamiast tego z Biblioteki Standardowej, więc
; możemy pozwolić użytkownikowi  stosować ustalone zakresy dla tworzenia tego samego labiryntu ( w tym samym
; zakresie) lub różnych labiryntów (przez wybranie różnych zakresów)

RandNum         proc    near
                push    cx
                mov     cl, byte  ptr  Seed
                and     cl, 7
                add     cl, 4
                mov     ax, Seed
                xor     ax, 55aah
                rol     ax, cl
                xor     ax, Seed
                inc     ax
                mov     Seed, ax
                pop     cx
                ret
RandNum         endp

;Init-  Obsługuje wszystkie prace inicjalizacyjne dla programu głównego. W szczególności , inicjalizuje pakiet
;       współprogramu, pobiera zakres liczb losowych od użytkownika i inicjalizuje monitor

Init            proc    near
                print
                byte    „Wprowadź  małą liczbę całkowitą dla zakresu liczby losowej:”, 0
                getsm
                atoi
                free
                mov     Seed, ax

; Wypełniamy wnętrze labiryntu znakami ściany, wypełniamy zewnętrzne dwa wiersze i kolumny wartościami.
; Będą to zapobiegało przed wędrowaniem  demonów na zewnątrz labiryntu

;Wypełniamy pierwszy wiersz wartościami Visited

                cld
                mov     cx, WordsPerRow
                lesi    Maze
                mov     ax,  Visited
        rep     stosw

; Wypełniamy ostatni wiersz wartościami NoWall
                mov     cx, WordsPerRow
                lea     di, Maze+(MaxYCoord+1)*BytesPerRow
        rep     stosw
; Zapisujemy wartość NoWall na pozycji startowej

                mov     Maze+(StartY*WordsPerRow+StartX)*2, NoWall

; Zapisujemy wartości NoWall wzdłuż dwóch pionowych brzegów labiryntu

                lesi    Maze
```

```
                mov     cx, MaxYCoord+1
EdgesLoop:      mov     es:[di],ax                  ;zatykamy lewy brzeg
                mov     es:[di+BytesPerRow-2], ax   ;zatykamy prawy brzeg
                add     di, BytesPerRow
                loop    EdgesLoop

                ifdef   ToScreen

;Okay, wypełnimy ekran wartościami WallChar:

                lesi    Screen
                mov     ax, WallChar
                mov     cx, 2000
        rep     stosw

; Zapiszemy właściwe znaki do lokacji początkowej i końcowej:

                mov     word  ptr  es:Screen+EndLoc, pathChar
                mov     word  ptr  es:Screen+StartLoc, NoWallChar

                endif                               ;ToScreen

; Wyzerowanie DemonList:

                ,mov    cx, (size  pch)*MaxDemons
                lea     di, DemonList
                mov     ax, dseg
                mov     es, ax
                xor     ax, ax
        rep     stosb

                ret
Init            endp

; CanStart-       Funkcja ta sprawdza aktualną pozycję aby zobaczyć czy generator może kopać
; nowy tunel w kierunku pionowym do aktualnego tunelu,. Możemy tylko zacząć nowy tunel jeśli
; są znaki ściany na przynajmniej dwóch pozycjach w żądanym kierunku:
;                ##
;               *##
;                ##
;
; Jeśli „*" jest aktualną pozycją a „#" przedstawia znaki ściany, a bieżącym kierunkiem jest północ
; lub południe, wtedy generator labiryntu zaczyna nową ścieżkę w kierunku wschodnim. Zakładając
; że „ . „ przedstawia tunel, nie możemy zacząć nowego tunelu w kierunku wschodnim jeśli
; wystąpi jakiś z tych wzorów:
;
;       .#      #.              ##      ##              ##      ##
;      *##     *##             *. #    *#.             *##     *##
;       ##      ##              ##      ##              .#      #.
;
;       CanStart zwraca prawdę (ustawiona flaga przeniesienia) jeśli możemy zacząć nowy tunel od ścieżki
; wykopanej przez aktualnego demona.
;
; Na wejściu,     dl jest współrzędną X demona
;                 dh  jest współrzędną Y demona
;                 cl jest kierunkiem demona
```

```
CanStart        proc    near
                push    ax
                push    bx

                MazeAdrs                        ;oblicza indeks do demon(x,y) w labiryncie
                mov     bx, ax

; CL  zawiera aktualny kierunek, 0= północ, 1=południe, 2= wschód, 3=zachód. Zauważmy, że
; możemy przetestować bit #1 dla północ / południe (0) lub wschód / zachód (1)

                test    cl, 10b                 ;zobacz czy północ/południe czy wschód/zachód
                jz      NorthSouth

; Jeśli demon idzie w kierunku wschodnim lub zachodnim, możemy zacząć nowy tunel jeśli jest
; sześć  bloków ściany powyżej lub poniżej aktualnego demona Notka: sprawdzamy czy wszystkie
; wartości w tych sześciu blokach są wartościami Wall. Ten kod zależy od faktu czy znaki Wall są
; zerem a suma tych sześciu bloków będzie zerem jeśli ruch jest możliwy.

                mov     al., byp  Maze[bx+BytesPerRow*2]        ;Mze[x,y+2]
                add     al, byp Maze[bx+BytesPerRow*2+2]        ;Maze[x+1, y+2]
                add     al, byp Maze [bx+BytesPerRow*2-2]       ;Maze[x-1, y+2]
                je      ReturnTrue

                mov     al, byp Maze[bx-BytesPerRow*2]          ;Maze[x, y-2]
                add     al,  byp Maze[bx-BytesPerRow*2+2]       ;Maze[x+1, y-2]
                add     al, byp Maze[bx-BytesPerRows*2-2]       ;Maze[x-1, y-2]
                je      returnTrue
ReturnFalse     clc                                     ;wyzerowana flaga przeniesienia = fałsz
                pop     bx
                pop     ax
                ret

; Jeśli demon idzie w kierunku północnym lub południowym, możemy zacząć nowy tunel jeśli jest sześć
; bloków ścian na lewo lub prawo bieżącego demona

NorthSouth:     mov     al., byp Maze[bx+4]                     ;Maze[x+2, y]
                add     al, byp Maze[bx+BytesPerRow+4]          ;Maze[x+2, y+1]
                add     al, byp Maze[bx-BytesPerRow+4]          ;Maze[x+2, y-1]
                je      returnTrue

                mov     al, byp Maze[bx-4]                      ;Maze[x-2,y]
                add     al, byp Maze[bx+BytesPerRow-4]          ;Maze[x-2, y+1]
                add     al, byp Maze[bx-BytesPerRow-4]          ;maze[x-2, y-1]
                jne     ReturnFalse

ReturnTrue:     stc                                     ;ustawiona flaga przeniesienia = prawda
                pop     bx
                pop     ax
                ret
CanStartendp

;CanMove-       testuje aby zobaczyć czy aktualny demon (kierunek = cl, x=dl, y=dh) może
;               ruszyć się określonym kierunku. Przesunięcie jest możliwe jeśli demon nie będzie
;               pochodził z wewnątrz jednego kwadratu innego tunelu. Funkcja ta zwraca prawdę
;               (flaga przeniesienia ustawiona) jeśli ruch jest możliwy. Na wejściu, CH zawiera
;               kierunek tego kodu, który powinniśmy przetestować.
```

```
CanMove     proc
            push    ax
            push    bx

            MazeAdrs                    ;wkładamy Maze[x,y] do ax
            mov     bx, ax

            cmp     ch, South
            jb      IsNorth
            je      IsSouth
            cmp     ch ,East
            je      IsEast
```

; Jeśli demon porusza się na zachód, sprawdza bloki w prostokącie sformowanym przez Maze
; [x-2, y-1] do Maze[x-1,y+2] aby upewnić się ,że są wszystkie wartości ściany.

```
            mov     al.,byp Maze[bx-BytesPerRow-4]      ;Maze[x-2, y-1]
            add     al, byp Maze[bx-BytesPerRow-2]      ;Maze[x-1, y-1]
            add     al, byp Maze[bx-4]                  ;Maze[x-2,y]
            add     al, byp Maze[bx-2]                  ;Maze[x-1, y]
            add     al, byp Maze[bx+BytesPerRow-4]      ;Maze[x-2, y+1]
            add     al, byp Maze[bx+BytesPerRow-2]      ;Maze [x-1, y+1]
            je      ReturnTrue
ReturnFalse: clc
            pop     bx
            pop     ax
            ret
```

; Jeśli demon idzie na wschód  sprawdza bloki w prostokącie sformowanym przez Maze[x+1, y-1]
; do Maze[x+2, y+1] aby upewnić się , że wszystkie to wartości ściany.

```
IsEast:     mov     al., byp Maze[bx-BytesPerRow+4]     ;Maze[x+2,y-1]
            add     al, byp Maze[bx-BytesPerRow+2]      ;Maze[x+1, y-1]
            add     al, byp Maze[bx+4]                  ;Maze[x+2,y]
            add     al, byp Maze[bx+2]                  ;Maze[x+1,y]
            add     al, byp Maze[bx+BytesPerRow+4]      ;Maze[x+2, y=1]
            add     al, byp Maze[bx+BytesPerRow+2]      ;Maze[x+1,y+1]
            jne     ReturnFalse

ReturnTrue: stc
            pop     bx
            pop     ax
            ret
```

; Jeśli demon idzie na północ, sprawdza bloki w prostokącie sformowanym przez Maze[x-1,y-2] do Maze[x+1,y-1]
; aby upewnić się, że wszystkie są wartościami ściany

```
IsNorth:    mov     al., byp Maze[bx-bytesPerRow-2]     ;Maze[x-1, y-1]
            add     al, byp Maze[bx-BytesPerRow*2-2]    ;Maze[x-1, y-2]
            add     al, byp Maze[bx-BytesPerRow]        ;Maze[x, y-1]
            add     al, byp Maze[bx-BytesPerRow*2]      ;Maze[x+1, y-1]
            add     al, byp Maze[bx-BytesPerRow+2]      ;Maze[x+1, y-1]
            add     al, byp Maze[bx-BytesPerRow*2+2]    ;Maze[x+1, y-2]
            jne     ReturnFalse
            stc
            pop     bx
```

```
                pop     ax
                ret

; Jeśli demon idzie na południe, sprawdza bloki w prostokącie sformowanym przez Maze[x-1, y+2] do
; Maze[x+1, y+1] aby upewnić się, że wszystkie są wartościami ściany

IsSouth:        mov     al., byp Maze[bx+BytesPerRow-2]         ;Maze[x-1, y-1]
                add     al, byp Maze[bx+BytesPerRow*2-2]        ;Maze[x-1, y+2]
                add     al, byp Maze[bx+BytesPerRow]            ;Maze[x, y-1]
                add     al, byp Maze[bx+BytesPerRow*2]          ;Maze[x+1, y+1]
                add     al, byp maze[bx+BytesPerRow+2]          ;Maze[x+1, y+1]
                add     al, byp Maze[bx+BytesPerRow*2+2]        ;Maze[x+1, y+2]
                jne     ReturnFalse
                stc
                pop     bx
                pop     ax
                ret

CanMove         endp

;SetDir- zmienia bieżący kierunek. Algorytm kopania labiryntu decyduje o zmianie kierunku tunelu poczynając
; kopanie od jednego z demonów. Kod ten sprawdza czy możemy zmienić kierunek i wybrać nowy jeśli to możliwe
;
; Jeśli demon idzie na północ lub południe, zmiana kierunku powoduje, że demon idzie na wschód lub zachód.
; Podobnie jeśli demon idzie na wschód lub zachód, zmiana kierunku wymusza kierunek na północ lub południe.
; Jeśli demon nie może zmienić kierunków (ponieważ nie może pójść w nowym kierunku z powodu tego lub innego
; powodu.. SetDir wraca bez robienia czegokolwiek. Jeśli zmiana kierunku jest możliwa , wtedy SetDir wybiera
; nowy kierunek. Jeśli jest możliwy tylko jeden nowy kierunek, demon wysyłany jest w tym kierunku. Jeśli demon
; może wyruszyć w jednym z dwóch różnych kierunków, SetDir  wybiera jeden z tych dwóch nowych kierunków
;
;  Funkcja ta zwraca nowy kierunek w al.

SetDir          proc    near

                test    cl, 10b                 ;zobacz czy północ / południe lub
                je      IsNS                    ;wschód / zachód

; idziemy na wschód lub zachód. Jeśli możemy ruszyć albo na północ albo południe z tego punktu, losowo
; wybieramy jeden z tych kierunków. Jeśli możemy ruszyć tylko jednokierunkowo, wybieramy ten kierunek. Jeśli
; nie możemy iść żadną drogą, wraca bez zmiany kierunku.

                mov     ch, North               ;Zobaczmy czy możemy ruszyć na północ
                call    CanMove
                jnc     NotNorth
                mov     ch, South               ;Zobaczmy czy możemy ruszyć na południe
                call    CanMove
                jnc     DoNorth
                call    RandNum                 ;Pobranie losowego kierunku
                and     ax, 1                   ;północ lub południe

DoNorth:        mov     ax, North
                ret
NotNorth:       mov     ch, South
                call    CanMove
                jnc     TryReverse
DoSouth:        mov     ax, South
```

```
                ret

;Jeśli demon  przesuwa się na północ lub południe, wybieramy nowy kierunek wschód lub zachód, jeśli możliwe

IsNS:           mov     ch, East                ;zobaczmy czy możemy iść na Wschód
                call    CanMove
                jnc     NotEast
                mov     ch, West                ;zobaczmy czy możemy  na Zachód
                call    CanMove
                jnc     DoEast
                call    RandNum                 ;pobranie losowego kierunku
                and     ax, 1b                  ;Wschód lub Zachód
                or      al., 10b
                ret

DoEast:         mov     ax, East
                ret

DoWest:         mov     ax, West
                ret

NotEast:        mov     ch, West
                call    CanMove
                jc      DoWest

;Jeśli nie możemy przełączyć na kierunek pionowy, zobaczymy czy można się odwrócić

TryReverse:     mov     ch, cl
                xor     ch, 1
                call    CanMove
                jc      ReverseDir

; Jeśli nie możemy się odwrócić , wtedy musimy iść w tym samym kierunku

                mov     ah, 0
                mov     al., cl                 ;zostajemy przy tym samym kierunku
                ret

; w przeciwnym razie odwracamy kierunek w dół

ReverseDir:     mov     ah, 0
                mov     al, cl
                xor     al, 1
                ret
SetDir          endp

; Stuck-  Funkcja ta sprawdza aby zobaczyć , czy demon jest zablokowany i nie może ruszyć się w żadnym kierunku.
; Zwraca prawdę, jeśli demon jest zablokowany i musi być zabity

Stuck           proc    near
                mov     ch, North
                call    CanMove
                jc      NotStuck
                mov     ch, South
                call    CanMove
                jc      NotStuck
```

```
                mov     ch, East
                call    CanMove
                jc      NotStuck
                mov     ch, West
                call    CanMove
NotStuck:       ret
Stuck           endp

;NextDemon-     przeszukuje całą listę demonów aby znaleźć następny dostępny demon. Zwraca wskaźnik
;               do niego w es:di.

NextDemon       proc    near
                push    ax

NDLoop:         inc     DemonIndex              ;przejście do następnego demona
                and     DemonIndex, ModDemons   ; MOD MaxDemons
                mov     al, size pcb            ;Obliczenie indeksu do DemonList
                mul     DemonIndex
                mov     di ,ax                  ;zobacz czy demon pod tym offsetem
                add     di, offset DemonList    ;jest aktywny
                cmp     byp [di].pcb.NextProc, 0
                je      NDLoop

                mov     ax, ds
                mov     es, ax
                pop     ax
                ret
NextDemon       endp

; Dig-  To jest proces demona. Przesuwa demona jedną pozycję (jeśli możliwe) w jego aktualnym
;       kierunku. Po przesunięciu o jedną pozycję w przód, jest 25% szansy, że zmieni swój kierunek.
;       jest 25% szans, że ten demon będzie uruchamiał proces potomny dla wykopania w kierunku pionowym

Dig             proc    near

; Zobacz czy bieżący demon jest zablokowany. Jeśli demon jest zablokowany, wtedy musimy usunąć go  z listy
; demonów. Jeśli nie jest zablokowany, wtedy musi kontynuować kopanie. Jeśli jest zablokowany i jest to ostatni
; aktywny wtedy zwraca sterowanie do programu głównego

                call    Stuck
                jc      NotStuck

; Okay, zabijamy aktywny demon.
; Notka: nie zabijemy nigdy ostatniego demona ponieważ mamy uruchomiony proces zegarowy .Proces zegarowy
;  jest tym , który zawsze zatrzymuje program

                dec     DemonCnt

; Ponieważ licznik nie jest zerem, musi być więcej demonów na liście demonów. Zwalniamy przestrzeń stosu
; powiązaną z aktualnym demonem, potem wyszukujemy następny aktywny demon .

MoreDemons:     mov     al., size  pcb
                mul     DemonIndex
                mov     bx, ax

; Zwalniamy przestrzeń stosu powiązanego z procesem. Zauważmy, że ten kod jest krnąbrny. Zakłada, że stos jest
```

```
; zaalokowany podprogramem malloc Biblioteki Standardowej, który zawsze tworzy adres bazowy  8

                mov     es, DemonList[bx].regss
                mov     di, 8
                free

;Oznaczamy wejście demona  do tego jako nieużywane

                mov     byp DemonList[bx]. NextProc, 0          ;oznaczone jako nieużywane

;Okay, lokujemy następny aktywny demon na liście

FndNxtDmn;      call    NextDemon
                Cocall                                          ;nigdy nie wraca

; Jeśli demon nie jest zablokowany, wtedy kontynuujemy kopanie

NotStuck:       mov     ch, cl
                call    CanMove
                jnc     DontMove

;Jeśli możemy ruszyć, wtedy modyfikujemy stosowne współrzędne demona:

                cmp     cl, South
                jb      MoveNorth
                je      MoveSouth
                cmp     cl, East
                jne     MoveWest

; Przesuwanie na Wschód:

                inc     di
                jmp     MoveDone

MoveWest:       dec     dl
                jmp     MoveDone

MoveNorth:      dec     dh
                jmp     MoveDone

MoveSouth:      inc     dh

;Okay, przechowujemy wartość NoWall przy tym wejściu w labiryncie i wyprowadzamy znak NoWall na ekran
; jeśli piszemy dane na monitorze)

MoveDone:       MazeAdrs
                mov     bx, ax
                mov     Maze[bx], NoWall

                ifdef   ToScreen
                ScrnAdrs
                mv      bx, ax
                push    es
                mov     ax, ScreenSeg
                mov     es, ax
                mov     word  ptr es:[bx], NoWallChar
```

```
                pop     es
                endif

; Przed opuszczeniem zobaczmy, czy demon nie powinien zmienić kierunku

DontMove:       call    RandNum
                and     al., 11b                ;25% szansy, że wynik to zero
                jne     NoChangeDir
                call    SetDir
                mov     cl, al.

NoChangeDir:

; Zobaczmy również, czy demon powinien dać początek procesowi potomnemu

                call    RandNum
                and     al., 11b                ;Daje to nam 25% szans
                jne     NoSpawn

;Okay,  zobaczmy, czy jest możliwe uruchomienie nowego procesu w tym punkcie:

                call    CanStart
                jnc     NoSpawn

;Zobaczmy, czy mamy już aktywny MaxDemons

                cmp     DemonCnt, MaxDemons
                jae     NoSpawn
                inc     DemonCnt                ;dodanie innego demona

;Okay, tworzymy nowego demona i dodajemy go do listy

                push    dx                      ;zachowujemy info naszego demona
                push    cx

:Lokujemy wolny slot dla tego demona

                lea     si, DemonList – size  pcb
FindSlot:       add     si, size  pcb
                cmp     byp [si].pcb.NextProc, 0
                jne     FindSlot

;Alokujemy jakąś przestrzeń stosu dla nowego demona

                mov     cx, 256                 ;256 bajtów stosu
                malloc

;Ustawiamy wskaźnik stosu dla niego:

                add     di, 248                 ;wskazuje koniec stosu
                mov     [si].pcb.regss, es
                mov     [si].pcb.regsp, di

;Ustawiamy adres wykonywalny dla niego:

                mov     [si].pcb regcs, cs
```

```
                mov     [si]. Pcb.regip, offset Dig

; Inicjalizujemy współrzędne I kierunek dla niego:

                mov     [si].pcb.regdx, ds.

:Wybieramy kierunek dla niego

                pop     cx                              ; wyszukanie kierunku
                push    cx

                call    SetDir
                mov     ah, 0
                mov     [si].pcb. regcx, ax


                mov     [si].pcb regds, seg  dseg
                sti
                pushf
                pop     [si].pcb.regflags
                mov     byp [si].pcb.NextProc, 1                ;oznaczono aktywację

; Przywracamy parametry aktualnego procesu

                pop     cx                                      ;przywrócenie aktualnego demona
                pop     dx

NoSpawn:

;Okay ,po zrobieniu wszystkiego powyższego, czas przekazać sterowanie do nowego kopania. Poniższe
; współwywołanie przekazuje sterowanie do następnego kopacza w DemonList

GetNextDmn:     call    NextDemon

;Okay, mamy wskaźnik do następnego demona na liście (może to być ten sam demon jeśli jest tylko jeden),
; przekazujemy sterowanie do tego demona

                cocall
                jmp     Dig
Dig             endp

; TimerDemon-   Ten demon wprowadza opóźnienie między każdym cyklem na liście demonów. Zwalnia to
;               generowanie labiryntu  więc możemy zobaczyć budowanie labiryntu (co czyni program
;               bardziej interesującym do oglądania)

TimerDemon      proc    near
                push    es
                push    ax

                mov     ax, 40h                                 ;obszar zmiennej BIOS
                mov     es, ax
                mov     ax, es:[6Ch]                            ;lokacja timera BIOS
Wait4Change     cmp     ax, es:[6Ch]                            ;zmiana BIOS co każde 1/18 sekundy
                je      Wait4Change

                cmp     DemonCnt, 1
```

```
                je      QuitProgram
                pop     es
                pop     ax
                call    NextDemon
                cocall
                jmp     TimerDemon
QuitProgram:    cocall  MainPCB                 ;wyjście z programu
TimerDemon      endp

; funkcja  solvemaze(x,y:integer): boolean

sm_X            textequ <[bp+6]>
sm_Y            textequ <[bp+4]>

SolveMaze       proc    near
                push    bp
                mov     bp, sp

;zobaczmy czy rozwiążemy labirynt:

                cmp     byte  ptr sm_X, EndX
                jne     NotSolved
                cmp     byte  ptr  sm_Y, EndY
                jne     NotSolved
                mov     ax, 1                   ;zwraca prawdę
                pop     bp
                ret     4

;zobaczmy czy przesunięcie do tego miejsca było poprawnym ruchem. To byłaby wartość NoWall
; w tej komórce w labiryncie jeśli ruch jest właściwy

NotSolved:      mov     dl, sm_X
                mov     dh, sm_Y
                MazeAdrs
                mov     bx, ax
                cmp     Maze[bx], NoWall
                je      MoveOK
                mov     ax, 0                   ;zwraca niepowodzenie
                pop     bp
                ret     4

; Cóż jest możliwe przesuniecie do tego punktu, więc umieszczamy właściwą wartość na ekranie
; i poszukujemy rozwiązania

MoveOK:         mov     Maze[bx], Visited

                ifdef   ToScreen
                push    es                      ;zapisuje  znak "VisitChar' na ekranie na
                ScrnAdrs                        ; pozycji X, Y
                mov     bx, ax
                mov     ax, ScreenSeg
                mov     es, ax
                mov     word  ptr  es:[bx], VisitChar
                pop     es
                endif
```

```
; Wywołujemy rekurencyjnie SolveMaze dopóki pobieramy rozwiązanie. Ponieważ wywołujemy SolveMaze dla
; czterech  możliwych kierunków (góra, dół, lewa prawa) w jakie idziemy. Ponieważ opuszczamy wartość „Visited”
; w Maze, nie będziemy  przypadkowo przeszukiwać  ścieżki jaką już przeszliśmy. Co więcej, jeśli nie możemy iść
; w jednym z czterech kierunków, SolveMaze będzie przechwytywał  to bezpośrednio na wejściu (zobaczmy kod na
; początku tego podprogramu.

                mov     ax, sm_X                ;próbujemy ścieżki spod lokacji (X-1,Y)
                dec     ax
                push    ax
                push    sm_Y
                call    SolveMaze
                test    ax ,ax                  ;Rozwiązanie?
                jne     Sovled

                push    sm_X                    ;spróbuj ścieżki spod lokacji (X,Y-1)
                mov     ax, sm_Y
                dec     ax
                push    ax
                call    SolveMaze
                test    ax, ax                  ;Rozwiązanie?
                jne     Solved

                mov     ax, sm_X                ;spróbuj ścieżki spod lokacji (X+1, Y)
                inc     ax
                push    ax
                push    sm_Y
                call    SolveMaze
                test    ax, ax                  ;Rozwiązanie?
                jne     Solved

                push    sm_X                    ;Spróbuj  ścieżki spod lokacji (X, Y+1)
                mov     ax, sm_Y
                inc     ax
                push    ax
                call    SolveMaze
                test    ax, ax                  ;Rozwiązanie?
                jne     Solved
                pop     bp
                ret     4

Solved:
                ifdef   ToScreen                ;rysuje ścieżkę powrotną
                push    es
                mov     dl, sm_X
                mov     dh, sm_Y
                ScrnAdrs
                mov     bx, ax
                mov     ax, ScreenSeg
                mov     es, ax
                mov     word  ptr es:[bx], PathChar
                pop     es
                mov     ax, 1                   ;zwraca prawdę
                endif

                pop     bp
                ret     4
```

```
SolveMaze       endp

;Tu jest program główny, który kieruje całą rzeczą:

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

                call    Init                    ;Inicjalizuje labirynt
                lesi    MainPCB                 ;inicjalizuje pakiet współprogramów
                coinit

;Tworzenie pierwszego demona. Ustawiamy wskaźnik stosu dla niego:

                mov     cx, 256
                malloc
                add     di, 248
                mov     DemonList.regsp, di
                mov     DemonList.regss, es

; Ustawiamy adres wykonania dla niego:

                mov     DemonList.regcs, cs
                mov     DemonList.regip, offset Dig

; Początkowe współrzędne I kierunek dla niego:

                mov     cx, East                ;zaczynamy od wschodu
                mov     dh, StartY
                mov     dl, StartX
                mov     DemonList.regcx, cx
                mov     DemonList.regdx, dx

; Ustawiamy inne różności:

                mov     DemonList.regds,  seg  dseg
                sti
                pushf
                pop     DemonList.regflags
                mov     byp  DemonList. NextProc, 1     ;Demon jest “aktywny”
                inc     DemonCnt
                mov     DemonIndex, 0

; Ustawiamy demona Timer:

                mov     DemonList.regsp+(size  pcb),  offset EndTimerStk
                mov     demonList.regss+(size  pcb), ss

; Ustawiamy adres wykonania dla niego:

                mov     DemonList.regcs +(size  pcb), cs
                mov     DemonList.regip+(size  pcb), offset TimerDemon

; Ustawiamy inne różności:
```

```
                mov     DemonList.regds+(size+ pcb), seg  dseg
                sti
                pushf
                pop     DemonList. Regflags+(size  pcb), seg d seg
                mov     byp  DemonList.NextProc+(size  pcb), 1
                int     DemonCnt
; Puszczenie mechanizmu w ruch

                mov     ax, ds.
                mv      es, ax
                lea     di, DemonList
                cocall

;poczekajmy na naciśnięcie klawisza przez użytkownika:

                getc
                mov     ax, StartX
                push    ax
                mov     ax, StartY
                push    ax
                call    SolveMaze

; Czekamy na inne naciśnięcie przed opuszczeniem:

                getc
                mov     ax, 3                   ;czyścimy ekran i resetujemy tryb video
                int     10h

Quit:           ExitPgm                         ;makro DOS do wyjścia z programu
Main            endp
ceg             ends
sseg            segment para stack ‘stack’

; tworzymy stos dla demona timer (inne stosy alokujemy dynamicznie)

TimerStk        byte    256  dup  (?)
EndTimerStk     word    ?

; Stos programu głównego

stk             byte    512  dup  (?)
sseg            ends

zzzzzzseg       segment para  public  ‘zzzzzz’
LastBytes       db      16  dup  (?)
zzzzzzseg       ends
                end     Main
```

Istniejący pakiet współprogramów Biblioteki Standardowej nie jest odpowiedni dla programów, które używają 80386 i zbioru rejestrów 32 bitowych. Jak wspomniano wcześniej, problem leży w fakcie, że Biblioteka Standardowa zachowuje tylko rejestry 16 bitowe , kiedy przełącza pomiędzy procesami. Jednakże, jest stosunkowo trywialnie rozszerzyć modyfikację Biblioteki Standardowej, żeby zachowywała 32 bitowe rejestry. Zrobimy to zmieniając definicję pcb 9z\robimy miejsce na 32 bitowe rejestry) i podprogram sl_cocall:

```
                .386
                option  segemnt:use16
```

```
dseg            segemnt para  public ‘data’

wp              equ     <word  ptr>

; PCB 32 bitowe. Odnotujmy, że możemy przetrzymać tylko najmniej znaczące 16 bitów SP ponieważ
; działamy w trybie rzeczywistym.

pcb32           struct
regsp           word    ?
regss           word    ?
regip           word    ?
regcs           word    ?

regeax          dword   ?
regebx          dword   ?
regecx          dword   ?
regedx          dword   ?
regesi          dword   ?
regedi          dword   ?
regebp          dword   ?

regds           word    ?
reges           word    ?
regflags        dword   ?
pcb32           ends

DefaultPCB      pcb32   <>
DefaultCortn    pcb32   <>

CurCoroutine    dword   DefaultCortn            ;wskazuje bieżąco wykonywany współprogram

dseg            ends
cseg            segment para  public  ‘slcode’
;=============================================================================
;
; Wsparcie dla 32-bitowych współprogramów
;
; COINIT32- ES:DI zawiera adres bieżącego (domyślnego) PCB procesu

Coinit32        proc    far
                assume  ds:dseg
                push    ax
                push    ds
                mov     ax, dseg
                mov     ds, ax
                mov     wp  dseg:CurCoroutine, di
                mov     wp   dseg:CurCoroutine+2, es
                pop     ds
                pop     ax
                ret
Coinit32        endp

; COCALL32 – przekazuje sterowanie do współprogramu. ES:DI zawiera adres PCB. Podprogram przekazuje
; sterowanie to tego współprogramu a potem zwraca wskaźnik do kodu wywołującego PCB w ES:DI

cocall32        proc    far
```

```
                assume  ds:dseg
                pushfd
                push    ds.
                push    es                      ;zachowanie tego na później
                push    edi
                push    eax
                mov     ax, dseg
                mov     ds., ax
                cli                             ;region krytyczny

; zachowanie stanu bieżącego procesu:

                les     di, dseg:CurCoroutine
                pop     es:[di].pcb32.regeax
                mov     es:[di].pcb32.regebx, ebx
                mov     es:[di].pcb32.regecx, ecx
                mov     es:[di].pcb32.regedx, edx
                mov     es:[di].pcb32.regesi, esi
                pop     es:[di].pcb32.regedi
                mov     es:[di].pcb32.regebp, ebp

                pop     es:[di].pcb32.reges
                pop     es:[di].pcb32.regds
                pop     es:[di].pcb32.regflags
                pop     es:[di].pcb32.regip
                pop     es:[di].pcb32.regcs
                mov     es:[di].pcb32.regsp, sp
                mov     es:[di].pcb32.regss, ss

                mov     bx, es                  ;zachowamy, więc możemy później zwrócić w ES:DI
                mov     ecx, edi
                mov     edx, es:[di].pcb32.regedi
                mov     es, es:[di].pcb32.reges
                mov     di, dx                  ;es:di wskazuje nowe PCB
                mov     wp  dseg:CurCoroutine, di
                mov     wp  dseg:CurCoroutine+2, es
                mov     es:[di].pcb32.regedi,  ecx      ;ES:DI zwraca wartości
                mov     es:[di].pcb32.reges, bx

;Okay przełączamy na nowy proces:

                mov     ss, es:[di].pcb32.regss
                mov     sp, es:[di].pcb32.regsp
                mov     eax, es:[di].pcb32.regeax
                mov     ebx, es:[di].pcb32.regebx
                mov     ecx, es:[di].pcb32.regecx
                mov     edx, es:[di].pcb32.regedx
                mov     esi, es:[di].pcb32.regesi
                mov     ebp, es:[di].pcb32.regebp
                mov     ds, es:[di].pcb32.regds

                push    es:[di].pcb32.regflags
                push    es:[di].pcb32.regcs
                push    es:[di],pcb32regip
                push    es:[di].pcb32.regedi
                mov     es, es:[di].pcb32.reges
```

```
                pop
                iret
cocall32        endp

; Cocall321działa jak  powyższa cocall, z wyjątkiem adresu pcb następującego po wywołaniu w
; strumieniu kodu zamiast być przekazywanym w ES:DI. Notka: kod ten  nie zwraca adresu PCB
; kodu wywołującego w ES:DI

cocall321       proc    far
                assume  ds:dseg
                push    ebp
                mov     bp, sp
                pushfd
                push    ds
                push    es
                push    edi
                push    eax
                mov     ax, dseg
                mov     ds, ax
                cli                                     ;region krytyczny

; Zachowanie stanu bieżącego procesu:

                les     di, dseg:CurCorputine
                pop     es:[di].pcb32.regeax
                mov     es:[di].pcb32.regebx, ebx
                mov     es:[di].pcb32. regecx, ecx
                mov     es:[di].pcb32.regedx. edx
                mov     es:[di].pcb32.regesi, esi
                pop     es:[di].pcb32.regedi
                pop     es:[di].pcb32.reges
                pop     es:[di].pcb32.regds
                pop     es:[di].pcb32.regflags
                pop     es:[di].pcb32.regebp
                pop     es:[di].pcb32.regip
                pop     es:[di].pcb32.regcs
                mov     es:[di].pcb32.regsp, sp
                mov     es:[di].pcb32.regss, ss

                mov     dx, es:[di].pcb32.regip         ;pobranie adresu zwrotnego (wskaźnik do adresu PCB
                mov     cx, es:[di].pcb32.regcs
                add     es:[di].pcb32.regip, 4
                mov     es, cx                          ;pobranie wskaźnika do nowego adresu pcb, potem
                mov     di,dx                           ; pobieramy wartość pcb
                les     di, es:[di]
                mov     wp  dseg:CurCoroutine, di
                mov     wp  dseg:CurCoroutine+2, es

;Okay, przełączamy do nowego procesu:

                mov     ss, es:[di].pcb32.regss
                mov     sp, es:[di].pcb32.regsp
                mov     eax, es:[di].pcb32.regeax
                mov     ebx, es:[di].pcb32.regebx
                mov     ecx, es:[di].pcb32.regecx
                mov     edx, es:[di].pcb32.regedx
```

```
                mov     esi, es:[di].pcb32.regesi
                mov     ebp, es:[di].pcb32.regebp
                mov     ds, es:[di].pcb32.regds

                push    es:[di].pcb32.regflags
                push    es:[di].pcb32.regcs
                push    es:[di].pcb32.regip
                push    es:[di].pcb32.regedi
                mov     es, es:[di].pcb32.reges
                pop     edi
                iret

cocall321       endp
cseg            ends
```

## 19.4 WIELOZADANIOWŚĆ

Współprogramy dostarczają sensownego mechanizmu do przełączania pomiędzy procesami, które muszą się zmieniać. Na przykład, program do generowania labiryntu z poprzedniej sekcji generowałby marny labirynt gdyby procesy demonów nie zmieniały się usuwając jedną komórkę z labiryntu. Jednakże, paradygmat współprogramów nie zawsze jest odpowiedni; nie wszystkie procesy muszą się zmienić. Na przykład przypuśćmy, ze piszemy grę akcji, gdzie użytkownik gra przeciwko komputerowi. Dodatkowo, komputer działa niezależnie od użytkownika w czasie rzeczywistym. To może być, na przykład, kosmiczna gra wojenna lub symulator lotu (gdzie prowadzisz walkę z innymi pilotami). Idealnie, byłoby mieć dwa komputery. Jeden dla interakcji użytkownika i drugi dla komputera. Oba systemy przekazywałyby swoje ruchy jeden drugiemu podczas gry. Jeśli gracz-człowiek siedziałby i obserwował ekran, gracz-komputer wygrałby ponieważ jest aktywny a człowiek nie. Oczywiście, byłoby znacznym ograniczeniem w sprzedaży gry gdyby były wymagane dwa komputery do gry. Jednakże, możemy użyć wielozadaniowości do symulowania dwóch oddzielnych systemów na pojedynczym CPU.

Podstawową ideą wielozadaniowości jest to, że jeden proces działa w okresie czasu (kwantowanie czasu lub odcinek czasu) a potem występuje proces przerwania zegarowego. Czasowy ISR zachowuje stan procesu a potem przełącza sterowanie do innego procesu. Proces ten działa w swoim odcinku czasu a potem przerwanie zegarowe przełącza na inny proces. W ten sposób, każdy proces zabiera jakąś ilość czasu komputera. Zauważmy, że wielozadaniowość jest bardzo łatwa do implementacji jeśli mamy pakiet współprogramów. Wszystko co musimy zrobić to napisać czasowy ISR, który współwywoła różne procesy, jeden na przerwanie zegarowe. Przerwanie zegarowe, które przełącza pomiędzy procesami to dyspozytor.

Decyzję jaką musimy podjąć, kiedy projektujmy dyspozytora jest założenie co do wybrania algorytmu dla procesu. Prostym założeniem jest umieszczenie wszystkich procesów w kolejce a potem rotacja między nimi. Jest to znane jako założenie cykliczne . Ponieważ jest to założenie jakiego używa pakiet procesów Biblioteki Standardowej , zaadoptujemy go również. Jednak, są inne kryteria wyboru dla procesu, generalnie wymagające nadrzędności procesu ,które są również dostępne.

Wybór kwantowania czasu może mieć duży wpływ na wydajność. Ogólnie, będziemy chcieli aby kwantowanie czasu było małe. Podział czasu (przełączanie pomiędzy procesami oparte o zegar) będzie dużo płynniejszy jeśli użyjemy małego kwantu czasu. Na przykład przypuśćmy, że wybraliśmy 5 sekundowy kwant czasu i uruchomiliśmy jednocześnie cztery procesy . Każdy proces będzie miał pięć sekund; będzie uruchamiany bardzo szybko podczas tych pięciu sekund. Jednak na koniec tego kawałka czasu będzie czekała na zmianę trzech innych procesów, 15 sekund, nim ponownie się uruchomi. Użytkownicy takich programów byliby bardzo sfrustrowani tym, użytkownicy chcieliby programów których wydajność jest stosunkowo spójna od jednego momentu do drugiego.

Jeśli zrobimy jedno milisekundowy odcinek czasu, zamiast pięciu sekund, każdy proces działałby przez jedną milisekundę, a potem przełączał do kolejnego procesu. To znaczy, każdy proces korzystałby z jednej milisekundy. Jest to zbyt mały kwant czasu dla użytkownika aby odczuł przerwy między procesami.

Ponieważ mniejsze kwanty czasu są lepsze, możemy zastanawiać się „dlaczego nie uczynić ich tak małymi jak to tylko możliwe?”. Na przykład PC wspiera jedno milisekundowe przerwanie zegarowe. Dlaczego nie użyć go do przełączania pomiędzy procesami? Problem jest tego typu, że jest wymagana równomierna ilość kosztów do przełączenia od jednego do drugiego procesu. Mniejsze uczynią kwantowanie czasu, większe będą kosztowały odcinek czasu. Dlatego też, chcemy wybrać kwant czasu, który balansuje pomiędzy procesem łagodnego

przełączenia a zbyt dużymi kosztami. Okazuje się, że 1/18 sekundowy zegar jest prawdopodobnie najlepszy dla większości wymagań wielozadaniowości

## 19.4.1 PROCESY NIESKOMPLIKWOANE I SKOMPLIKWOANE

Są dwa główne typy procesów w świecie wielozadaniowości: procesy nieskomplikowane , znane również jako wątki i procesy skomplikowane. Te dwa typy procesów różnią się głównie w szczegółach zarządzania pamięcią. Proces skomplikowany zmienia tablice zarządzania pamięcią i przesuwa dużo danych . Wątki zmieniają tylko stos i rejestry CPU. Wątki mają dużo mniejsze koszta niż procesy skomplikowane

Nie będziemy w tym tekście rozpatrywać procesów skomplikowanych. Pojawiają się one w trybie chronionym systemów operacyjnych takich jak UNIX, Linux, OS/2 lub Windows NT. Ponieważ rzadko zarządzanie pamięcią (na poziomie sprzętu) wychodzi dalej niż do DOS, temat zmiany tablic zarządzania pamięcią będą poddane pod dyskusję. Przełączanie z jednej aplikacji skomplikowanej do innej generalnie odpowiada przełączeniu z jednej aplikacji do drugiej.

Używanie procesów nieskomplikowanych (wątków) jest doskonałym zastosowaniem pod DOS. Wątki (skrót od „ wątek wykonania" lub „wątek wykonawczy") odpowiadają dwóm lub więcej jednoczesnym wykonaniom ścieżki wewnątrz tego samego programu. Na przykład, możemy myśleć o każdym demonie w programie generującym labirynt jako będącym oddzielnym wątkiem wykonawczym.

Chociaż wątki mają różne stosy i stany maszynowe, dzielą one kod i dane pamięci. Nie ma potrzeby użycia „pamięci dzielonej TSR" dostarczającej globalnej pamięci dzielonej. Zamiast tego, wykorzystanie lokalnych zmiennych jest trudniejszym zadaniem.. Musimy albo zaalokować zmienne lokalne na stosie procesu (który jest oddzielny dla każdego procesu) albo musimy się upewnić, że nie ma innych procesów używających zmiennych jakie zadeklarowaliśmy w segmencie danych określonym dla jednego wątku.

Możemy łatwo napisać własny pakiet wątków, ale nie musimy; Biblioteka Standardowa dostarcza tej możliwości w pakiecie processes. Zobaczmy jak włączyć wątki do naszych programów, czytając.....

## 19.4.2 PAKIET PROCESSES BIBLIOTEKI STANDARDWOEJ UCR

Biblioteka Standardowa UCR dostarcza sześciu podprogramów pozwalających zarządzać wątkami. Podprogramy te to prcsinit, prcsquit, fork, die, kill i yield. Funkcje te pozwalają nam inicjalizować I zamykać wątki systemu, zaczynać nowy proces, kończyć procesy Ii dobrowolnie przekazać CPU do innego procesu.

Funkcje Prcsinit i prcsquit pozwalają zainicjalizować i zamknąć system. Funkcja prcsinit przygotowuje wątki pakietu. Musimy wywołać ten podprogram przed wykonaniem jakiegoś innego z pięciu podprogramów procesu .Funkcja prcsquit zamyka wątki systemu przygotowując się na zamknięcie programu. Prcsinit aktualizuje przerwanie zegarowe (przerwanie 8). Prcsquit przywraca wektor przerwania 8. Jest to bardzo ważne, że wywołujemy prcsquit zanim program wróci do DOS'a Niepowodzenie w wykonaniu , opuszczenie wektora int 8 wskazującego na pamięć, może spowodować krach systemu, kiedy DOS załaduje kolejny program. Nasz program musi zaktualizować wektory wyjątków break i błędu krytycznego, zakładając, że wywołamy prcsquit w przypadku anormalnego zakończenia programu. Niepowodzenie w wykonaniu tego może spowodować krach systemu jeśli użytkownik zakończy program ctrl-break lub przerwie błąd I/O. Prcsinit i prcsquit nie wymagają żadnych parametrów , nie zwracają żadnych wartości.

Funkcja fork daje początek nowemu procesowi. Na wejściu es:di muszą wskazywać pcb nowego procesu. Pola regss i regsp pcb muszą zawierać adres szczytu obszaru stosu dla tego nowego procesu. Funkcja fork wypełnia inne pola pcb (wliczając w to cs:ip)

Dla każdego wywołania robimy fork., podprogram fork wraca dwukrotnie, raz dla każdego wątku wykonawczego. Proces macierzysty zazwyczaj wraca pierwszy, ale nie jest to takie pewne; proces potomny jest zazwyczaj zwracany jako drugi przez funkcję fork. Rozróżniając te dwa wywołania, fork zwraca dwa identyfikatory procesu (PID'y) w rejestrach ax i bx. Dla procesu macierzystego, fork zwraca ax zawierający zero a bx zawierający PID procesu potomnego. Dla procesu potomnego, fork zwraca ax zawierający PID potomka i bx zawierający zero. Zakuwamy, że oba wątki wracają i kontynuują wykonywanie tego samego kodu po wywołaniu fork. Jeśli chcemy aby potomny i macierzysty proces pobrały oddzielne ścieżki, wykonamy kod podobny do poniższego:

```
lesi    NewPCB                  ;zakładamy, że  regss / regsp są zainicjalizowane
fork
```

```
test    ax, ax                  ; Macierzysty PID to zero w tym miejscu
je      ParentProcess           ; idziemy gdzie indziej jeśli proces macierzysty
```

; Proces potomny kontynuuje wykonywanie tutaj

Proces macierzysty powinien zachować PID potomka. Możemy użyć PID'a do zakończenia procesu w późniejszym czasie.

Powtarzam, że ważne jest to, że musimy zainicjalizować pola regss i regsp w pcb przed wywołaniem fork. Musimy zaalokować pamięć dla stosu (dynamicznie lub statycznie) a ss:sp wskazują ostatnie słowo tego obszaru stosu kiedy wywołamy fork, pakiet procesu użyje jakiejś wartości, która będzie w polach regss i regsp. Jeśli nie zainicjalizujemy tych wartości, będą one prawdopodobnie zawierały zera a kiedy zacznie się proces , zniszczy dane z pod adresu 0:FFFE. Może to spowodować krach systemu a tym lub innym punkcie.

Funkcja die , zabija aktualny proces .Jeśli jest wiele uruchomionych procesów, funkcja ta przekazuje sterowanie do jakiegoś innego procesu czekającego na uruchomienie .Jeśli bieżący proces jest jedynym w systemowej kolejce uruchomień, wtedy funkcja ta spowoduje krach systemu.

Funkcja kill pozwala jednemu procesowi zakończyć inny. Zazwyczaj, proces macierzysty będzie używał tej funkcji do kończenia procesu potomnego. Aby zabić proces, po prostu ładujemy rejestr ax PID'em procesu jaki chcemy zakończyć a potem wywołujemy kill. Jeśli proces dostarcza swojego własnego PID'a do funkcji kill, proces sam się kończy (to znaczy, jest to odpowiednik funkcji die). Jeśli jest tylko jeden proces w kolejce uruchomień a proces sam się zabija, wtedy nastąpi krach systemu..

Ostatnim podprogramem zarządzania wielozadaniowością w pakiecie process jest funkcja yield. Yield dobrowolnie poddaje się CPU. Jest to bezpośrednia funkcja dyspozytora, która będzie przełączać na inne zadanie w kolejce uruchomień. Sterowanie jest zwracane po funkcji yield, kiedy dany jest następny odcinek czasu do procesu. Jeśli aktualny proces jest jedynym w tej kolejce, yield wraca natychmiast. Zwykle używamy funkcji yield dla zwalniania CPU pomiędzy długimi operacjami I/O (jak oczekiwanie na naciśnięcie klawisza). Pozwala to innym zadaniom wykorzystać maksymalnie CPU podczas gdy nasz proces obraca się w pętli oczekując na zakończenie jakiejś operacji I/O .

Podprogramy wielozadaniowe Biblioteki Standardowej pracują tylko ze zbiorem rejestrów 16 bitowych rodziny 80x86. Podobnie jak pakiet współprogramów, będziemy musieli zmodyfikować pcb i kod dyspozytora jeśli chcemy wesprzeć zbiór 32 bitowych rejestrów procesora 80386 i późniejszych. Zadanie to jest stosunkowo proste a kod jest trochę podobny do tego, który pojawił się w sekcji o współprogramach; więc nie trzeba przedstawiać tego rozwiązania tutaj .

---

## 19.4.3 PROBLEMY Z WIELOZADANIOWOŚCIĄ

Kiedy wątki dzielą kod i dane, mogą pojawić się pewne problemy. Przede wszystkim, problem wielobieżności. Nie możemy wywołać nie wielobieżnego podprogramu (jak DOS) z dwóch oddzielnych wątków jeśli jest nawet taka możliwość, że kod nie współbieżny może być przerywany a sterowanie przekazane do drugiego wątku, który współużytkuje ten sam program. Jednak nie jest to jedyny problem. Jest całkiem możliwe zaprojektowanie dwóch podprogramów, które mają dostęp do zmiennych dzielonych a te podprogramy zachowują się źle w zależności od tego gdzie wystąpi przerwanie w sekwencji kodu. Będziemy drążyć ten temat w sekcji o synchronizacji., jednak teraz musimy być świadomi, że ten problem istnieje.

Zauważmy, że proste wyłączenie przerwań (cli) może nie rozwiązać problemu współużytkowania. Rozważmy następujący kod:

```
cli                             ; zapobieżenie współużywalności
mov     ah, 3Eh                 ;funkcja zamykania DOS
mov     bx, Handle
int     21h
sti                             ;powrotne włączenie przerwań
```

Kod ten nie będzie zapobiegał przed współbieżnością, ponieważ DOS (i BIOS) włączają ponownie przerwania!. Jest rozwiązanie tego problemu ale nie przez użycie cli i sti.

---

## 19.4.4 PRZYKŁADOWY PROGRAM Z WĄTKAMI

Poniższy program dostarcza prostej demonstracji pakietu process. Ten krótki program tworzy dwa wątki – program główny i proces zegarowy. Przy każdym takcie zegarowym proces drugoplanowy (zegar)) jest wykopywany i zwiększa się zmienna pamięci. I wtedy zwraca CPU z powrotem do programu głównego następny takt zegarowy przekazuje sterowanie do procesu drugoplanowego i cały cykl się powtarza, Program główny czyta ciąg od użytkownika podczas gdy proces drugoplanowy zlicza takty zegarowe. Kiedy użytkownik kończy linię przez naciśnięcie klawisza Enter, pogram główny zabija proces drugoplanowy a potem drukuje ilość czasu konieczną do wprowadzenia linii tekstu.

Oczywiście nie jest to najbardziej wydajny sposób obliczania jak długo ktoś wprowadza linię tekstu, ale dostarcza przykładu cech wielozadaniowości Biblioteki Standardowej. Ta krótka część programu demonstruje wszystkie podprogramy process z wyjątkiem die. Zauważmy, ze demonstruje również fakt, że musimy dostarczyć programy obsługi int 23h i int 24h, kiedy używamy pakietu process/

```
; MULTI.ASM
; Prosty program demonstrujący zastosowanie multitaskingu

                ,xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment  para  public  'data'
ChildPID        word    0                       ;PID potomka, więc możemy go zabić
BackGndCnt      word    0                       ;zliczanie  taktów zegara w tle

; PCB dla naszego procesu. Tu inicjalizujemy ss:sp

BkgndPCB        pcb             {0, offset EndStk2,  seg EndStk2}

;Bufor danych przechowujący ciąg wprowadzany

InputLine       byte    128  dup  (0)

dseg            ends

cseg            segment para public ,code'
                assume  cs:cseg, ds:dseg

; Zastąpienie programu obsługi błędu krytycznego. Podprogram wywołuje prcsquit jeśli użytkownik zdecydował się
; przerwać program

CritErrMsg      byte    cr, lf
                byte    "DOS Critical Error!", cr, lf
                byte    "A)bort, R)etry, I)gnore, F)ail? $"

MyInt24         proc    far
                push    dx
                push    ds
                push    ax
                push    cs
                pop     ds
Int24Lp:        lea     dx, CritErrMsg
                mov     ah, 9                   ;funkcja DOS drukująca ciąg
                int     21h

                mov     ah, 1                   ;funkcja  DOS odczytu znaku
                int     21h
```

```
                and     al., 5Fh                ;konwertujemy l.c → u.c

                cmp     al, ‚I'                 ;Ignorujemy?
                jne     NotIgnore
                pop     ax
                mov     al, 0
                jmp     Quit24

NotIgnore:      cmp     al, ‚r'                 ;Ponawiamy?
                jne     NotRetry
                pop     ax
                mov     al, 1
                jmp     Quit24

NotRetry:       cmp     al, 'A'                 ; Przerywamy
                jne     NotAbort
                prcsquit                        ;jeśli wychodzimy , poprawiamy INT 8
                pop     ax
                mov     al., 2
                jmp     Quit24

NotAbort:       cmp     al., 'F'
                jne     BadChar
                pop     ax
                mov     al, 3
Quit24:         pop     ds
                pop     dx
                iret

BadChar:        mov     ah, 2
                mov     dl, 7                   ;znak dzwonka
                jmp     Int24Lp
MyInt24         endp

; Będziemy blokowali INT 23h (wyjątek break)

MyInt23         proc    far
                iret
MyInt23         endp

; Okay, to jest słaby proces drugoplanowy, ale demonstruje jak używać  funkcji Biblioteki Standardowej

BackGround      proc
                sti
                mov     ax, dseg
                mov     ds, ax
                inc     BackGndCnt
                yield
                jmp     BackGround
BackGround      endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit
```

```
; Inicjalizujemy wektory programów obsługi wyjątków INT 23h I INT 24h

                mov     ax, 0
                mov     es, ax
                mov     word  ptr  es:[24h*4], offset MyInt24
                mov     es:[24h*4+2], cs
                mov     word  ptr es:[23h*4], offset  MyInt23
                mov     es:[23h*4+2], cs

                prcsinit                                        ;start systemu wielozadaniowego

                lesi    BkgndPCB                                ; odpalamy nowy proces
                fork
                test    ax, ax                                  ;powrót  procesu macierzystego?
                je      ParentPrcs
                jmp     BackGround                              ;idziemy do  drugoplanowego

ParentPrcs:     mov     ChildPID, bx                            ;zachowujemy ID procesu potomnego

                print
                byte    „Podliczam Cię kiedy wpisujesz ciąg. Więc pisz”
                byte    dr, lf
                byte    „szybko: „ , 0

                lesi    InputLine
                gets
                mov     ax, ChildPID                            ;zatrzymanie potomka uruchomionego
                kill
                printf
                byte    „Wpisując ‘%s’  pobrałeś %d taktów zegara”
                byte    cr, lf,0
                dword   InputLine, BackGndCnt

                prcsquit

Quit:           ExitPgm
Main            endp

cseg            ends

sseg            segment para stack ‘stack’

; Tu jest stos  dla procesu drugorzędnego jaki zaczęliśmy

stk2            byte    256  dup (?)
Endstk2         word    ?

;Tu jest stos dla programu głównego / procesu  pierwszoplanowego

stk             byte    1024  dup (?)
sseg            ends

zzzzzzseg       segment para  public  ‘zzzzzz’
LastBytes       db      16 dup (?)
Zzzzzzseg       ends
                end     Main
```

## 19.5 SYNCHRONIZACJA PROCESÓW

Wiele problemów występuje w jednoczesnej współpracy wykonywanych procesów ze względu na synchronizację procesów (lub brak tego). Na przykład jeden proces może tworzyć dane , które inne procesy zużywa. Jednakże, może to być dużo dłuższe dla producenta tworzącego tą dana niż dla konsumującego go. Pewne mechanizmy muszą w pewnym momencie założyć, że konsument nie próbuje używać danej zanim producent go stworzy. Podobnie , musimy założyć, że konsument używa danej tworzonej przez producenta zanim producent stworzy więcej danych

Problem producent – konsument jest jednym z kilku bardzo znanych problemów synchronizacji procesów w teorii systemów operacyjnych. W problemie producent – konsument jest jeden lub więcej procesów, które tworzą dane i zapisują te dane do dzielonego bufora. Podobnie, jest jeden lub więcej konsumentów, którzy czytają dane z tego bufora. Są dwa zagadnienia synchronizacji procesów jakimi musimy się zająć – pierwszy to zapewnienie, że producenci nie tworzą więcej danych niż bufor może przechować ( odwrotnie, musimy zapobiec przed usuwaniem danych przez konsumentów z pustego bufora); po drugie zapewnić integralność struktury danych bufora poprzez zezwolenie na dostęp tylko jednego procesu w czasie.

Rozważmy, co może się wydarzyć w prostym problemie producent – konsument. Przypuśćmy, że procesy producenta i konsumenta dzielą pojedynczą strukturę bufora danych zorganizowanego jak następuje:

```
buffer  struct
Count   word    0
InPtr   word    0
OutPtr  word    0
Data    byte    MaxBufSize  dup  (?)
buffer  ends
```

Pole Count określa liczbę bajtów danych obecnych w buforze. InPtr wskazuje kolejną dostępną lokację mieszczącą dane w buforze. OutPtr jest adresem kolejnego bajtu do usunięcia z bufora. Data jest faktyczną tablicą bufora. Dodawanie i odejmowanie danych jest bardzo łatwe. Następujący segment kodu prawie wykonuje tą pracę:

```
; Producer-     Procedura ta dodaje wartość w al do bufora. Zakładamy, że zmienna buforowa MyBuffer jest w
;               segmencie danych

Producer        proc    near
                pushf
                sti                             ; musimy włączyć przerwania
                push    bx

; Poniższa pętla czeka dopóki jest miejsce w buforze dla wprowadzenia innego bajtu

WaitForRoom:    cmp     MyBuffer.Count, MaxBufSize
                jae     WaitForRoom

; Okay, wprowadzamy bajt do bufora

                mov     bx, MyBuffer.InPtr
                mov     MyBuffer.Data[bx], al
                inc     Mybuffer.Count                  ; dodajemy bajt do bufora
                inc     MyBuffer.InPtr                  ; przesuwamy na następną pozycję w buforze

; Jeśli jesteśmy na fizycznym końcu bufora, zawijamy do początku

                cmp     MyBuffer.InPtr, MaxBufSize
                jb      NoWrap
```

```
                mov     MyBuffer.InPtr, 0

NoWrap:
                pop     ax
                popf
                ret
Producer        endp


; Consumer-     Procedura ta czeka na dane (jeśli konieczne) i zwraca kolejny dostępny bajt z bufora

Consumer        proc    near
                pushf                           ; musimy mieć włączone przerwania
                sti
                push    bx
WaitForData:    cmp     Count, 0                ;Czy bufor jest pusty?
                je      WaitForData             ;Jeśli tak czekamy na przybycie danych

;Okay, pobranie znaku wejściowego

                mov     bx, MyBuffer.OutPtr
                mov     al, MyBuffer.Data[bx]
                dec     MyBuffer.Count
                inc     MyBuffer.OutPtr
                cmp     MyBuffer.OutPtr,  MaxBufSize
                jb      NoWrap
                mov     MyBuffer.OutPtr, 0
NoWrap:
                pop     bx
                popf
                ret
Consumer        endp
```

Jedyny problem z tym kodem jest taki, że nie zawsze działa jeśli jest wiele procesów producenckich i konsumenckich. Faktycznie, łatwo jest zbliżyć się do wersji tego kodu, który nie działa dla pojedynczego zbioru procesów producenckiego i konsumenckiego (chociaż ten powyższy kod będzie działał dobrze, w specjalnych przypadkach) Problem jest taki, że te procedury uzyskują dostęp do zmiennych globalnych i  dlatego nie są współbieżne. W szczególności problem leży w sposobie w jaki te dwie procedury manipulują buforem sterującym zmiennymi. Rozważmy, na chwilę, poniższe instrukcje z procedury Consumer:

```
                dec     MyBuffer.Count
        < przypuśćmy, że tu wysterują przerwania >

                        inc     MyBuffer.OutPtr
                        cmp     MyBuffer.OutPtr, MaxBufSize
                        jb      NoWrap
                        mov     MyBuffer.OutPtr, 0
NoWrap:
```

Jeśli wystąpi przerwania w określonym punkcie powyższego  kodu i przekaże sterowanie do innego procesu konsumenckiego, który współużywa ten kod, drugi konsument będzie działał źle. Problem w tym, że pierwszy konsument pobierał dane z bufora, ale ma jeszcze uaktualniony wskaźnik wyjściowy. Drugi konsument pojawia się i usuwa ten sam bajt co pierwszy konsument. Drugi konsument wtedy właściwie uaktualnia wskaźnik wyjściowy, wskazujący kolejną dostępną lokację w buforze cyklicznym. Kiedy sterowanie ostatecznie zwracane jest do procesu pierwszego konsumenta, kończy on działanie poprzez zwiększenie wskaźnika wyjściowego. Powoduje to ,że system

przeskakuje kolejny bajt, którego żaden proces nie czyta. Końcowym efektem jest to, że dwa procesy konsumenckie pobierają ten sam bajt a potem pomijają bajt w buforze.

Problem ten jest łatwy do rozwiązania poprzez rozpoznanie faktu, ze kod, który manipuluje buforem danych jest regionem krytycznym. Poprzez ograniczenie wykonywania w tym krytycznym regionie do jednego procesu po kolei, możemy rozwiązać ten problem. W prostym powyższym przykładzie, możemy łatwo zapobiec współużytkowaniu poprzez wyłączenie przerwań w regionie krytycznym. Dla procedury konsumenckiej, kod może wyglądać jak ten:

```
; Consumer-     Procedura ta oczekuje na dane (jeśli konieczne) i zwraca kolejny dostępny bajt z bufora

Consumer        proc    near
                pushf                           ;musimy włączyć przerwania
                sti
                push    bx
WaitForData:    cmp     Count, 0                ;czy bufor jest pusty?
                je      WaitForData             ;jeśli tak, czekamy na przybycie danych

; Poniżej mamy region krytyczny więc wyłączamy przerwania

                cli

;Okay pobieramy znak wejściowy

                mov     bx, MyBuffer.OutPtr
                mov     al, MyBuffer.Data[bx]
                dec     MyBuffer.Count
                inc     MyBuffer.OutPtr
                cmp     MyBuffer.OutPtr, MaxBufSize
                jb      NoWrap
                mov     MyBuffer.OutPtr, 0
NoWrap:
                pop     bx
                popf                            ;przywracamy flagę przerwania
                ret
Consumer        endp
```

Zauważmy, że nie możemy wyłączyć przerwań podczas wykonywania całej procedury. Przerwania muszą być dopóki ta procedura oczekuje na dane, w przeciwnym razie proces producencki, nigdy nie będzie mógł odłożyć danych do bufora dla konsumenta.

Po prostu wyłączenie przerwań nie zawsze działa. Pewne regiony krytyczne mogą pobierać znaczne ilości czasu (sekundy, minuty lub nawet godziny) i nie możemy pozostawić wyłączonych przerwań dla tej ilości czasu. Innym problemem jest to ,że region krytyczny może wywołać procedurę, która włącza przerwania ponownie a nie mamy nad tym kontroli. Dobrym przykładem jest procedura, która wywołuje DOS. Ponieważ MS-DOS nie jest współużytkowy, MS-DOS , z definicji, jest sekcją krytyczną; możemy pozwolić tylko na jeden proces w danym czasie wewnątrz MS-DOS. Jednakże, MS-DOS ponownie aktywuje przerwania, więc nie możemy po prostu wyłączyć przerwań przed wywołaniem funkcji MS-DOS, z wyjątkiem tej, która zachowuje współbieżność.

Wyłączenie przerwań nie działa dla konsumenta / producenta danych wcześniej. Zauważmy, że przerwania muszą kiedy konsument czeka na przybycie danych do bufora (odwrotnie, producenci muszą mieć przerwania kiedy czekają na miejsce w buforze). Jest całkiem możliwe dla tego kodu, że wykryje obecność danych przed wykonaniem instrukcji cli, przekazuje sterowanie do drugiego procesu konsumenckiego. Kiedy nie jest możliwe dla obu procesów zaktualizowanie jednocześnie zmiennych bufora, jest możliwe dla drugiego procesu konsumenckiego usunięcie tylko wartości danej z bufora wejściowego a potem przełączenie ponowne do pierwszego konsumenta, który usuwa wartość z bufora (i powoduje ,ze zmienna Count staje się ujemna).

Jedną kiepską stroną tego rozwiązania jest zastosowanie flag dla uzyskania dostępu do regionu krytycznego . Proces, przed wejściem do regiony krytycznego, testuje flagi aby zobaczyć czy jakiś inny proces jest obecnie w regionie krytycznym; jeśli nie, proces ustawia flagi na „ w użyciu” a potem wprowadza region krytyczny. Przy

opuszczaniu regionu krytycznego, proces ustawia flagi na „nie używane”. Jeśli proces chce wprowadzić region krytyczny i wartość flag jest „ w użyciu”, proces musi czekać dopóki proces obecny w sekcji krytycznej zakończy się i zapisze wartość „nie używane” do flag.

Jedyny problem z tym rozwiązaniem jest taki, że nie jest to nic więcej niż specjalny przypadek problemu producent / konsument. Instrukcje, te aktualizują postać flag w użyciu swoją własna sekcją krytyczną, którą musimy chronić. Generalnie, idea flag jako rozwiązanie nie jest najlepsza.

---

## 19.5.1 OPERACJE NIEPODZIELNE, TESTOWANIE I USTAWIANIE , I AKTYWNE CZEKANIE

Problem z ideą flag w użyciu jest taki, że potrzebuje kilku instrukcji do testowania i ustawiania flagi. Przykładowy fragment kodu, który testuje taką flagę, odczytywałby jej wartość i określał czy sekcja krytyczna jest w użyciu. Jeśli nie , wtedy zapisywałby wartość „w użyciu” do flag , co pozwoliłoby innym procesom poznać, że to jest sekcja krytyczna. Problem jest tego typu, że przerwanie może wystąpić po kodzie testującym flagi, ale przed ustawieniem flag do „w użyciu”. Wtedy mogą się pojawić jakieś inne procesy, przetestować flagi i znaleźć tą, która nie jest w użyciu i wprowadzić region krytyczny. System może przerwać drugi proces kiedy jest jeszcze w regionie krytycznym i przekazać sterowanie Z powrotem do pierwszego. Ponieważ pierwszy proces już określił, że region krytyczny nie jest w użyciu, ustawia flagi na „w użyciu” i wprowadza region krytyczny. Teraz mamy dwa procesy w regionie krytycznym i system z naruszeniem wymaganego wzajemnego wykluczenia ( tylko jeden proces w regionie krytycznym w czasie)

Problem ten pojawia się jeśli testowanie i ustawianie flag w użyciu nie jest operacją nieprzerwaną (niepodzielną). Jeśli była, wtedy nie będzie problemu. Oczywiście, łatwo jest wykonać sekwencję instrukcji nie przerywalnych przez wstawienie instrukcji cli między nie.. Dlatego też, możemy testować i ustawiać flagi w operacji niepodzielnej jak następuje (zakładamy w użyciu zero, nie w użyciu jeden):

```
            pushf
TestLoop:   cli                     ;wyłączamy przerwania podczas testowania i
            cmp     Flag, 0         ; ustawiania flag
            je      IsInUse         ; czy już w użyciu?
            mov     Flag, 0         , jeśli nie , zrób to więc
IsInUse:    sti                     ;zezwolenie na przerwania (jeśli już w użyciu)
            je      TestLoop        ; czekaj dopóki nie w użyciu
            popf

: Kiedy dotrzemy tu, flagi były „nie w użyciu” i ustawiam w „w użyciu”. Teraz mamy zastrzeżony dostęp do
; regionu krytycznego
```

Innym rozwiązaniem jest zastosowanie tak zwanych instrukcji „testowanie i ustawianie” – testują one określony warunek i ustawiają flagę na żądaną wartość. W naszym przypadku potrzebujemy instrukcji, które testują flagę aby zobaczyć czy nie jest w użyciu i ustawiają ją na w użyciu w tym samym czasie,. (jeśli flaga była już w użyciu, pozostanie w użyciu później). Chociaż 80x86 nie wspiera określonych instrukcji testowania i ustawiania, dostarcza kilku innych, które mogą osiągnąć taki sam efekt, Do instrukcji tych zalicza się xchg, shl, shr, sar, rcl, ror, rol, btc / btr /bts (dostępne tylko na 80386 i późniejszych procesorach ) i cmpxchg (dostępnej tylko na 80486 i późniejszych procesorach). W ograniczonym znaczeniu możemy również użyć instrukcji dodawania i odejmowania (add, sub, adc, sbb, inc i dec).

Instrukcja wymiany dostarcza najogólniejszej formy operacji testowania i ustawiania. Jeśli mamy flagę (0= w użyciu, 1= nie w użyciu) możemy przetestować i ustawić tą flagę bez bałaganienia w przerwaniach używając takiego kodu:

```
InUseLoop:  mov     al., 0          ;0 = W użyciu
            xchg    al., Flag
            cmp     al, 0
            je      InUseLoop
```

Instrukcja xchg niepodzielnie wymienia wartość w al z wartością w zmiennej flagi. Chociaż instrukcja xchg nie testuje w rzeczywistości wartości, robi miejsce dla oryginalnej wartości flagi w lokacji (al) zabezpieczając ją przed modyfikacją przez inny proces. Jeśli flaga pierwotnie zawierała zero (w użyciu), ta sekwencja wymiany zamieni zero

dla istniejącego zera a potem powtórzy pętle. Jeśli flaga pierwotnie zawierała jeden (nie w użyciu) wtedy ten kod wymienia zero (w użyciu) dla jeden i wypada z używania pętli.

Instrukcje przesunięcia i obrotu również działają jako instrukcje testowania i ustawiania, zakładając, ze używamy właściwych wartości dla flagi w użyciu. Z w użyciu równym zero i nie użyciu równym jeden, poniższy kod demonstruje jak używać instrukcji shr dla operacji testowania i ustawiania:

```
InUseLoop:    shr     Flag, 1                 ;Bit w użyciu do flagi przeniesienia, 0 → Flaga
              Jnc     InUseLoop               ;Powtarzamy jeśli już w użyciu
```

Kod ten przesuwa bit w użyciu (bit numer zero) do flagi przeniesienia i czyści flagę w użyciu. W tym samym czasie zeruje zamienną Flag, zakładając, ze Flag zawsze zawiera zero lub jeden. Kod dla testu niepodzielnej sekwencji testowania i ustawiania używający innych przesunięć i obrotów jest bardzo podobny .

Startując z 80386, Intel dostarcza zbioru instrukcji wyraźnie zaplanowanych do operacji testowania i ustawiania : btc (testuj bit i uzupełnij), bts (testuj bit i ustaw) i btr (testuj bit i resetuj). Instrukcje te kopiują określony bit z operandu docelowego do flagi przeniesienia a potem uzupełniają, ustawiają lub repetują (czyszczą) ten bit. Poniższy kod demonstruje jak używać instrukcji btr do manipulowania naszą flagą w użyciu:

```
InUseLoop:    btr     Flag, 0                 ;flaga w użyciu jest w bicie zero
              jnc     InUseLoop
```

Instrukcja btr jest trochę bardziej elastyczna niż instrukcja shr ponieważ nie musimy gwarantować, że wszystkie pozostałe bity w zmiennej Flag są zerami; testuje i zeruje bit zero bez wpływania na inne bity w zmiennej Flag

Instrukcja cmpxchg 80486 (i późniejszych) dostarcza bardzo ogólnego prymitywu synchronizacji. Instrukcja „porównaj i wymień” wydaje się być jedyną instrukcją niepodzielną jakiej potrzebujemy do implementacji prawie wszystkich prymitywów synchronizacji .Jednakże, jej ogólna struktura oznacza, że jest trochę zbyt złożona dla prostych operacji testowania i ustawiania. Więcej szczegółów o cmpxchg znajduje się w „Instrukcje CMPXCHG i CMPXCHG8B”.

Wracając do problemu producent / konsument, możemy łatwo rozwiązać problem regionu krytycznego, który istnieje w tych podprogramach używając sekwencji instrukcji testowania i ustawiania przedstawionych powyżej. Poniższy kod robi to dla procedury Producer, będziemy mogli zmodyfikować procedurę Consumer w podobny sposób

```
;Producer-     Procedura ta dodaje wartość w al. do bufora. Zakładamy, że zmienna buforowa MyBuffer jest w
;             segmencie danych

Producer      proc    near
              pushf
              sti                             ;musimy włączyć przerwania

;Okay, jesteśmy przy wprowadzeniu regionu krytycznego, więc testujemy flagę w użyciu aby zobaczyć czy ten
; region krytyczny jest już w użyciu

InUseLoop:    shr     Flag, 1
              jnc     InUseLoop

              push    bx

;Poniższa pętla oczekuje dopóki jest miejsce w buforze do wprowadzenia innego bajtu

WaitForRoom:  cmp     MyBufer.Count, MaxBufSize
              jae     WaitForRoom

; Okay, wprowadzamy bajt do bufora

              mov     bx, MyBuffer.InPtr
              mov     MyBuffer.Data[bx], al
```

```
                inc     Mybuffer.Count          ;dodaliśmy bajt do bufora
                inc     MyBuffer.InPtr          ;przesuwamy na kolejną pozycję w buforze

;Jeśli jesteśmy na fizycznym końcu bufora, zawijamy do początku

                cmp     MyBuffer.InPtr, MaxBufSize
                jb      NoWrap
                mov     MyBuffer.InPtr, 0

NoWrap:
                mov     Flag, 1                 ;Ustawienie flagi nie do użycia
                pop     bx
                popf
                ret
Producer        endp
```

Jednym ważnym problem z podejściem do ochrony regionu krytycznego przy testowaniu i ustawianiu jest to ,że używa pętli aktywnego czekania. Kiedy region krytyczny nie jest dostępny, proces kręci się w pętli oczekując swojej kolei w regionie krytycznym. Jeśli proces, który jest obecnie w regionie krytycznym pozostaje tam znaczną ilość czasu (powiedzmy sekundy, minuty lub godziny), proces(y) czekające na wejście do regionu krytycznego kontynuują marnotrawienie czasu CPU oczekując na flagę. To , z kolei, marnuje czas CPU, który mógłby być wykorzystany lepiej, pobranie procesu w regionie krytycznym przez niego, aby mógł wejść inny proces.

Inny problem, który może zaistnieć, to to, że istnieje możliwość dla jednego procesu wchodzącego do regionu krytycznego , zamknięcie innych procesów , opuszczenie krytycznego regionu, wykonanie jakiegoś przetwarzania, a potem współużywać wszystko w tym samym odcinku czasu .Jeśli okaże się, że proces jest zawsze w regionie krytycznym kiedy występuje przerwanie zegarowe, żaden z innych procesów oczekujących na wejście do regionu krytycznego nie będzie tego robił .jest to problem znany jako trwałe zablokowanie – procesy oczekujące na wejście do regionu krytycznego nigdy tego nie zrobią ponieważ jakiś proces zawsze im to wybije „z głowy”

Jedynym rozwiązaniem tych dwóch problemów jest zastosowanie obiektu synchronizacji znanego jako semafor Semafory dostarczają wydajnych i ogólnego przeznaczenia mechanizmów dla ochrony regionów krytycznych.

---

## 19.5.2 SEMAFORY

Semafor jest obiektem z dwoma podstawowymi metodami: oczekiwanie i sygnał (lub udostępnienie). Używając semafora tworzymy zmienną semaforową (instancję) dla poszczególnego regionu krytycznego lub innego zasobu jaki chcemy chronić. Kiedy proces chce używać danego zasobu, oczekuje na semafor. Jeśli żaden inny proces nie jest aktualnie używanym zasobem, wtedy funkcja oczekująca ustawia semafor w użycia, bezpośrednio wraca do procesu. W tym czasie proces ma wyłączny dostęp do zasobu. Jeśli jakiś inny proces używa już tego zasobu (np. jest w regionie krytycznym), wtedy semafor blokuje bieżący proces poprzez przesunięcie go z kolejki uruchomienia do kolejki semaforu. Kiedy proces, który aktualnie przechowuje udostępniony zasób, operacja udostępniania usuwa proces oczekujący z kolejki semafora i umieszcza go ponownie w kolejce uruchomienia. W kolejnym dostępnym odcinku czasu ten nowy proces wraca ze swojej funkcji oczekującej i może wejść do swojego regionu krytycznego.

Semafory rozwiązują dwa ważne problemy z pętlą aktywnego czekania opisanej w poprzedniej sekcji. Po pierwsze, kiedy proces czeka a semafor blokuje proces, proces ten już nie jest w kolejce uruchomienia, więc nie konsumuje więcej czasu CPU, do chwili, kiedy operacja udostępniania umieści go z powrotem w kolejce uruchomienia. W odróżnieniu od aktywnego czekania, mechanizm semaforu nie marnuje (tak bardzo) czasu CPU w procesach, które czekają na jakiś zasób.

Semafory mogą również rozwiązać problem trwałego zablokowania.. Operacja czekania, kiedy blokuje proces, może umieścić go na końcu kolejki semaforu FIFO. Operacja udostępniania może pobrać nowy proces z przodu kolejki FIFO umieszczając z powrotem w kolejce uruchomienia. Ta zasada zakłada, e każdy proces wprowadza kolejkę semaforu która staje się równa priorytetowemu dostępowi do zasobu.

Implementacja semaforów jest łatwym zadaniem,. Semafor ogólnie składa się ze zmiennej całkowitej i kolejki. System inicjalizuje zmienną całkowitą liczbą procesów, które mogą dzielić zasób w jednym czasie (tą wartością jest zazwyczaj jeden dla regionów krytycznych i innych zasobów wymagających zastrzeżonego dostępu) Operacja oczekiwania zmniejsza trą zmienną. Jeśli wynik jest większy niż lub równy zero, funkcja oczekiwania po

prostu wraca do kodu wywołującego; jeśli wynik jest mniejszy niż zero, funkcja oczekiwania zachowuje stan maszynowy. Przesuwa pcb procesu z kolejki uruchomieniowej do kolejki semaforu a potem przełącza CPU na inne procesy (tj. funkcja yield)

Funkcja udostępniania jest prawie odwrotnością. Zwiększa wartość całkowitą. Jeśli wynik nie jest jedynką, funkcja udostępniania przesuwa pcb z przodu kolejki semaforu do kolejki uruchomienia. Jeśli wartość całkowita staje się jedynką, nie ma więcej procesów w kolejce semaforu, więc funkcja udostępniania po prostu wraca do kodu wywołującego. Zauważmy ,że funkcja udostępniania nie aktywuje procesu usuwanego z kolejki procesów semafora. Po prostu umieszcza ten proces w kolejce uruchomienia .Sterowanie zawsze wraca do procesu, który wykonał wywołanie udostępnienia (chyba ,że oczywiście wystąpi przerwanie zegarowe podczas wykonywania funkcji udostępniania).

Oczywiście, obojętnie kiedy manipulujemy systemową kolejką uruchomienia, jesteśmy w regionie krytycznym. Dlatego też, wydaje się nam, że główny problem mamy tu – celem semafora jest ochrona regionu krytycznego, mimo, że semafor sam ma region krytyczny, który musimy chronić. Wydaje się, ze wymaga to wnioskowania cyklicznego. Jednakże ten problem łatwo się rozwiązuje. Pamiętajmy, że głównym powodem dla którego nie wyłączamy przerwań chroniąc region krytyczny jest to, że region krytyczny może pobierać dużo czasu na wykonanie lub może wywołać inny podprogram, który włączy je z powrotem .Sekcja krytyczna w semaforze jest bardzo krótka i nie wywołuje innych podprogramów. Dlatego też, na krótko wyłączamy przerwania podczas gdy w regionie krytycznym semafora jest wszystko w porządku.

Jeśli nie wolno nam wyłączyć przerwań, możemy zawsze użyć instrukcji testowania i ustawiania w pętli do ochrony regionu krytycznego. Chociaż wprowadza to pętlę aktywnego czekania, okazuje się, że nie będziemy nigdy czekali więcej niż dwa odcinki czasu zanim opuścimy pętlę aktywnego czekania, więc nie zmarnujemy dużo czasu CPU czekając na wejście do regionu krytycznego semafora.

Chociaż semafory rozwiązują dwa ważne problemy z pętlą aktywnego czekania, łatwo jest popaść w kłopoty kiedy używamy semaforów. Na przykład, jeśli proces oczekuje na semafor a semafor przyznaje zastrzeżony dostęp do powiązanego zasobu, wtedy ten proces nigdy nie udostępni semafora, żaden proces oczekujący na semafor będą zawieszone w nieskończoność.. Podobnie, proces, który oczekuje na ten sam semafor dwukrotnie bez udostępnienia po środku będzie zawieszał się, a inne procesy, które czekają na semafor , w nieskończoność. Proces, który nie udostępnia zasobu już nie potrzebuje naruszać pojęcia semafor i jest błędem logicznym w programie .Są również inne problemy, które mogą się ujawnić, jeśli proces oczekuje na wiele semaforów przed udostępnieniem jakiegoś. Wrócimy do tego problemu w sekcji o blokadzie systemu (zobacz „Blokada systemu)

Chociaż możemy napisać własny pakiet semafora, pakiet process Biblioteki Standardowej dostarcza własnych funkcji oczekiwania i udostępniania wraz z definicją zmiennej semaforowej. Opisuje to następna sekcja

---

## 19.5.3 WSPARCIE BIBLIOTEKI STANDARDOWEJ UCR DLA SEMAFORA

Pakiet process Biblioteki Standardowej UCR dostarcza dwóch funkcji do manipulowania zmienną semaforową: WaitSemaph i RlsSemaph. Funkcje te , odpowiednio, oczekują i sygnalizują semafor. Podprogramy te zagłębiają się w udogodnienia zarządzania procesem, czyniąc go łatwiejszym do implementacji synchronizacji używając semaforów w naszych programach.

Pakiet process dostarcza poniższych definicji dla typu danych semafora:

```
semaphore       struct
SemaCnt         word    ?
smaphrLst       dword   ?
endsmaphrLst    dword   ?
semaphore       ends
```

Pole SemaCnt określa jak wiele procesów może dzielić zasobów (jeśli dodatnie), lub jak wiele procesów aktualnie oczekuje na zasób (jeśli ujemne). Domyślnie pole to jest inicjalizowane wartością jeden .to pozwala jednemu procesowi w tym czasie używać zasobu chronionego przez semafor. Za każdym razem proces oczekując na semafor, zmniejsza to pole. Jeśli wynik zmniejszony jest dodatni lub zerowy, operacja oczekiwania natychmiast wraca. Jeśli zmniejszony wynik jest ujemny wtedy operacja czekania przesuwa pcb aktualnego procesu z kolejki uruchomienia do kolejki semaforu zdefiniowanej przez pola smaphrLst i endsmaphrLst z powyższej struktury.

Większość czasu będziemy używali domyślnej wartości jeden dla pola SemaCnt. Jest kilka sytuacji, jednak kiedy możemy chcieć zezwolić aby więcej niż jeden proces uzyskał dostęp do zasobu. Na przykład przypuśćmy, że projektujemy grę dla wielu graczy, który komunikuje się pomiędzy różnymi maszynami używając szeregowego

portu komunikacyjnego lub karty sieciowej. Możemy mieć obszar w grze, który ma miejsce dla tylko dwóch graczy w tym samym czasie. Na przykład, gracze mogą ścigać się poszczególnymi „transporterami" w przestrzeni kosmicznej, ale jest miejsce tylko dla dwóch graczy w transporterze w tym samym czasie. Przez inicjalizowanie zmiennej semaforowej liczbą dwa, zamiast jeden, operacja oczekiwania pozwoli dwóm graczom kontynuować w tym samym czasie zamiast tylko jednemu. Kiedy trzeci gracz próbuje wprowadzić transporter, funkcja WaitSemaph będzie blokować gracza przed wprowadzeniem dopóki jedne z pozostałych graczy opuści grę.

Zastosowanie funkcji WaitSemaph lub RlsSemaph jest bardzo łatwe; ładujemy do pary adresów żądaną zmienną semaforową i podajemy właściwe wywołanie funkcji. RlsSemaph zawsze wraca natychmiast (zakładając, że nie wystąpi przerwanie zegarowe podczas RlsSemaph , funkcja WaitSemaph wraca kiedy semafor zezwoli na dostęp do zasobu, który chroni. Przykłady tych dwóch funkcji pojawią się w następnej sekcji.

Podobnie jak współprogramy i pakiet process Biblioteki Standardowej , tak pakiet semafora zachowuje tylko zbiór 16 bitowych rejestrów CPU 80x86. jeśli chcemy użyć zbioru 32 bitowych rejestrów 80386 i późniejszych procesorów będziemy musieli zmodyfikować kod źródłowy funkcji WaitSemaph i RlsSemaph . Kod jaki musimy zmienić jest prawie identyczny z kodem we współprogramach i pakiecie process, więc jest to trywialna zmiana. Zapamiętajmy jednak, ze będziemy musieli zmienić ten kod jeśli używamy udogodnień 32 bitów 80386 i późniejszych procesorów.

---

## 19.5.4 UZYWANIE SEMAFORÓW DO OCHRONY REGIONÓW KRYTYCZNYCH

Możemy użyć semaforów dla uzyskania wspólnego zastrzeżonego dostępu do jakiegoś zasobu. Na przykład, jeśli kilka procesów chce użyć drukarki, możemy stworzyć semafor, który zezwoli na dostęp do drukarki przez tylko jeden proces w czasie (dobry przykład procesu, który będzie w „regionie krytycznym" przez kilka minut) Jednakże większość powszechnych zadań dla semafora to ochrona regionów krytycznych przed współużytkowaniem. Trzema przykładami kodu, które potrzebują ochrony przed współużytkowaniem to funkcje DOS, funkcje BIOS i różne funkcje Biblioteki Standardowej >Semafory są idealne dla sterowania dostępem do tych funkcji.

Chroniąc DOS przed współużytkowaniem przez kilka różnych procesów musimy stworzyć zmienną DOSmsaph i wykonać wywołanie właściwej funkcji WaitSemaph i RlsSemaph przy wywołaniu DOS'a Poniższy kod przykładowy demonstruje jak to zrobić

```
; MULTIDOS.ASM
;
; Program ten demonstruje  jak używać semaforów do ochrony funkcji DOS

                .xlist
                include         stdlib.a
                includelib      stdlib.lib
                .list

dseg            segment  para public 'data'
DOSsmaph        semaphore {}

; Makra oczekiwania i udostępniania  semafora DOS

DOSWait         macro
                push    es
                push    di
                lesi    DOSsmaph
                WaitSemaph
                pop     di
                pop     es
                endm

DOSRls          makro
                push    es
                push    di
```

```
                lesi    DOSsmaph
                RlsSemaph
                pop     di
                pop     es
                endm

; PCB dla naszego procesu drugoplanowego:

BkgndPCB        pcb     {0, offset EndStk2, seg  EndStk2}

; Wydruk danych dla procesów  pierwszoplanowego i drugoplanowego:

StrPtrs1        dword   str1_a, str1_b, str1_c, str1_d, str1_e, str1_f
                dword   str1_g, str1_h, str1_i, str1_j, str1_k, str1_l
                dword   0

str1_a          byte    „Foreground: string ‘a’ „„ cr, lf, 0
str1_b          byte    „Foreground: string ‘b’ „„ cr, lf, 0
str1_c          byte    „Foreground: string ‘c’ „„ cr, lf, 0
str1_d          byte    „Foreground: string ‘d’ „„ cr, lf, 0
str1_e          byte    „Foreground: string ‘e’ „„ cr, lf, 0
str1_f          byte    „Foreground: string ‘f’ „„ cr, lf, 0
str1_g          byte    „Foreground: string ‘g’ „„ cr, lf, 0
str1_h          byte    „Foreground: string ‘h’ „„ cr, lf, 0
str1_i          byte    „Foreground: string ‘i’ „„ cr, lf, 0
str1_j          byte    „Foreground: string ‘j’ „„ cr, lf, 0
str1_k          byte    „Foreground: string ‘k’ „„ cr, lf, 0
str1_l          byte    „Foreground: string ‘l’ „„ cr, lf, 0


StrPtrs2        dword   str2_a, str2_b, str2_c, str2_d, str2_e, str2_f
                dord    str2_g, str2_h, str2_i
                dword   0

str2_a          byte    „Background: string ‘a’ „„ cr, lf, 0
str2_b          byte    „Background: string ‘b’ „„ cr, lf, 0
str2_c          byte    „Background: string ‘c’ „„ cr, lf, 0
str2_d          byte    „Background: string ‘d’ „„ cr, lf, 0
str2_e          byte    „Background: string ‘e’ „„ cr, lf, 0
str2_f          byte    „Background: string ‘f’ „„ cr, lf, 0
str2_g          byte    „Background: string ‘g’ „„ cr, lf, 0
str2_h          byte    „Background: string ‘h’ „„ cr, lf, 0
str2_i          byte    „Background: string ‘i’ „„ cr, lf, 0

dseg            ends

cseg            segment para public  ‘code’
                assume  cs:cseg, ds:dseg

; Zastąpienie programu obsługi błędu krytycznego. Podprogram ten wywołuje prcsquit jeśli
; użytkownik zdecyduje przerwać program

CritErrMsg      byte    cr, lf
                byte    “DOS Critical Error!”, cr, lf
                byte    “A)bort, R)etry, I)gnore, F)ail? $”
```

```
MyInt24     proc    far
            push    dx
            push    ds
            push    ax

            push    cs
            pop     ds
Int24Lp:    lea     dx, CritErrMsg
            mov     ah, 9                   ;funkcja DOS'a drukująca ciąg
            int     21h

            mov     ah, 1                   ;funkcja DOS'a odczytująca znak
            int     21h
            and     al., 5Fh                ; konwersja l.c → u.c

            cmp     al, 'I'                 ;ignorujemy?
            Jne     NotIgnore
            pop     ax
            mov     al, 0
            jmp     Quit

NotIgnore:  cmp     al, 'r'                 ;powtarzamy?
            jne     NotRetry
            pop     ax
            mov     al, 1
            jmp     Quit24

NotRetry:   cmp     al, 'A'                 ;przerywamy
            jne     NotAbort
            prcsquit                        ;jeśli wychodzimy, zmieniamy INT 8
            pop     ax
            mov     al., 2
            jmp     Quit24

NotAbort:   cmp     al., 'F'
            jne     BadChar
            pop     ax
            mov     al, 3
Quit24:     pop     ds
            pop     dx
            iret

BadChar:    mov     ah, 2
            mov     dl, 7                   ; znak dzwonka
            jmp     Int24Lp
MyInt24     endp

; Będziemy blokować INT 23h (wyjątek break)

MyInt23     proc    far
            Iret
MuInt23     endp

; Ten proces drugoplanowy wywołuje DOS do wydrukowania kilku ciągów na ekranie. W między czasie proces
; pierwszoplanowy również drukuje ciągi na ekranie. Aby zapobiec współużywalności, lub przynajmniej  plątaninie
; znaków na ekranie, kod ten używa semaforów do ochrony funkcji DOS. Dlatego też, każdy proces będzie drukował
```

```
; jedną kompletną linie, potem udostępni semafor. Jeśli inny proces oczekuje, będzie drukował swoją linię

BackGround      proc
                mov     ax, dseg
                mov     ds, ax
                lea     bx, strPtrs2                    ;tablica  wskaźników ciągów
PrintLoop:      cmp     word ptr [bx+2], 0              ;czy koniec wskaźników?
                je      BkGndDone
                les     di, [bx]                        ;pobranie ciągu do drukowania
                DOSWait
                puts                                    ;funkcja DOS dla drukowania ciągu
                DOSRls
                add     bx, 4                           ;wskazuje następny wskaźnik ciągu
                jmp     PrintLoop

BkGndDone:      die                                     ;zakończenie tego procesu
BackGround      endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

;Inicjalizacja wektorów programów obsługi wyjątków  INT23h i  INT24h

                mov     ax, 0
                mov     es ,ax
                mov     word  ptr  es:[24h*4],  offset MyInt24
                mov     es:[24h*4+2],cs
                mov     word ptr es:[23h*4],  offset MyInt23
                mov     es:[23h*4+2], cs

                prcsinit                                ;start systemu wielozadaniowego

                lesi    BkgndPCB                        ;odpalamy nowy proces
                fork
                test    ax, ax                          ;powrót macierzystego procesu?
                je      ParentPrcs
                jmp     BackGround                      ; idziemy do  drugiego planu

; Proces rodzicielski będzie drukował wiązkę ciągów w tym samym czasie co proces drugoplanowy. Użyjemy
; semafora DOS dla ochrony wywołania DOS' które wykonuje PUTS

ParentPrcs:     DOSWait                                 ;wymuszenie innych procesów kończących
                mov     cx, 0                           ; oczekiwanie w kolejce semafora przez
DlyLp0:         loop    DlyLp0                          ;opóźnienie przynajmniej dla jednego cyklu z
DlyLp1:         loop    DlyLp1                          ; zegarowego
DlyLp2:         loop    DlyLp2
                DOSRls

                lea     bx, StrPtrs1                    ;Tablica wskaźników do ciągów
PrintLoop:      cmp     word  ptr [bx+2], 0             ;czy koniec wskaźników?
                je      ForeGndDone
                les     di, [bx]                        ;pobranie ciągu do druku
                DOSWait
```

```
                puts                                    ;funkcja DOS dla wydruku ciągu
                DOSRls
                add     bx, 4                           ;wskazuje kolejny wskaźnik do ciągu
                jmp     PrintLoop

ForeGndDone:    prcsquit

Quit            ExitPgm
Main            endp

cseg            ends

sseg            segment para stack 'stack'

; Tu jest stos dla rozpoczętego procesu drugoplanowego

stk2            byte    1024  dup (?)
EndStk2         word    ?

; Tu mamy sto dla  programu głównego / procesu pierwszoplanowego

stk             byte    1024  dup  (?)
sseg            ends

zzzzzzseg       segment para  public 'zzzzzz'
LastBytes       db      16    dup (?)
zzzzzzseg       ends
                end     Main
```

Program ten bezpośrednio nie wywołuje DOS'a, ale wywołuje podprogram puts Biblioteki Standardowej . Ogólnie, możemy użyć pojedynczego semafora do ochrony wszystkich funkcji BIOS'a, DOS'a i Biblioteki Standardowej. Jednak nie jest to szczególnie efektywne. Na przykład podprogramy dopasowania do wzorca Biblioteki Standardowej nie potrzebują żadnych funkcji DOS; dlatego też czekając na semafor DOS'a robimy dopasowanie do wzorca podczas gdy inny proces wykorzystuje funkcje DOS niekoniecznie opóźniając to dopasowanie do wzorca. Nie jest niczym złym mieć jeden proces robiący dopasowanie do wzorca podczas gdy inny korzysta z funkcji DOS'a. Niestety, pewne podprogramy Biblioteki Standardowej robią wywołania DOS'a (puts jest dobrym przykładem), więc musimy użyć semafora DOS przy takich wywołaniach.

Teoretycznie, możemy użyć oddzielnych semaforów do ochrony DOS, różnych funkcji BIOS'a i różnych funkcji Biblioteki Standardowej. Jednak śledząc wszystkie te semafory wewnątrz programu jest dużym zadaniem. Co więcej, zakładając, że funkcja DOS nie wywołuje również niechronionych podprogramów BIOS ,jest trudnym zadaniem. Więc większość programistów używa pojedynczego semafora do ochrony funkcji Biblioteki Standardowej, DOS'a i BIOS'a.

---

## 19.5.5 ZSTOSOWANIE SEMAFORÓW DO SYNCHRONIZACJI BARIERY

Chociaż podstawowym zastosowaniem semaforów jest dostarczenie zastrzeżonego dostępu do jakiegoś zasobu, są inne zastosowania synchronizacji dla semaforów. W tej sekcji przypatrzymy się zastosowaniu obiektów semaforowych Biblioteki Standardowej do tworzenia barier.

Bariera jest punktem w programie, gdzie proces się zatrzymuje i czeka na inne procesy do synchronizacji (docierając do ich właściwych barier) . W tym sensie bariera jest podwójnym semaforem. Semafor uniemożliwia więcej niż n procesom korzystanie z dostępu do jakiegoś zasobu. Bariera nie przyznaje dostępu dopóki przynajmniej n procesów nie zażąda dostępu.

Mając dane różne właściwości tych dwóch metod synchronizacji, możemy pomyśleć., że będzie trudno użyć programów WaitSemaph i RlsSemaph do implementacji barier. Jednak, okazuje się to być całkiem proste . Powiedzmy, że pole semafora SemaCnt było zainicjalizowane zerem zamiast jedynką. Kiedy pierwszy proces oczekuje na ten semafor, system będzie natychmiast blokował ten proces. Podobnie, każdy dodatkowy proces, który

oczekuje na ten semafor będzie blokowany i oczekiwał w kolejce semafora. Normalnie byłaby to katastrofa ponieważ nie ma aktywnego procesu, który sygnalizowałby semafor, więc będą aktywowane procesy zablokowane. Jednak, jeśli zmodyfikujemy funkcję oczekiwania aby sprawdzała pole SemaCnt przed rzeczywistym oczekiwaniem, n-ty proces może przeskoczyć funkcje oczekiwania i reaktywować inne procesy. Rozważmy poniższe makro:

```
barrier     macro   Wait4Cnt
            local   AllHere,  AllDone
            cmp     es:[di].semaphore. SemaCnt,      -(Wait4Cnt-1)
            jle     AllHere
            WaitSemaph
            cmp     es:[di]. Semqphore.SemaCnt ,0
            je      AllDone
AllHere:    RlsSemaph
AllDone:
            endm
```

Makro to oczekuje pojedynczego parametru, który powinien być liczbą procesów (wliczając w to proces bieżący), które muszą być obarierowane zanim jakiś proces może być kontynuowany. Pole SemaCnt jest wartością ujemną której wartość absolutna określa jak wiele procesów aktualnie oczekuje na semafor. Jeśli bariera wymaga czterech procesów, żaden proces nie może kontynuować dopóki czwarty proces uderza w barierę; w tym czasie pole SemaCnt będzie zawierało minus trzy. Powyższe makro oblicza jaka powinna być wartość pola SemaCnt jeśli wszystkie procesy są za barierą. Jeśli SemaCnt dopasowuje tą wartość, sygnalizuje semafor, który zaczyna łańcuch operacji z każdym zablokowanym procesem udostępniającym kolejny. Kiedy SemaCnt trafi zero, ostatni zablokowany proces nie udostępnia semafora ponieważ nie ma innych procesów oczekujących w kolejce.

Ważne jest aby pamiętać o inicjalizacji pola SemaCnt zerem przed użyciem semaforów dla synchronizacji barier w ten sposób. Jeśli nie zainicjalizujemy SemaCnt zerem, funkcja WaitSemaph nie będzie prawdopodobnie blokowała żadnych procesów.

Poniższy przykładowy program dostarcza prostego przykładu zastosowania synchronizacji barier przy użyciu pakietu semaforowego Biblioteki Standardowej:

```
;BARRIER.ASM
;
; Ten przykładowy program demonstruje jak używać obiektów semaforowych Biblioteki Standardowej do
; synchronizacji kilku procesów przy barierze. Program ten jest podobny do programu MULTIDOS.ASM
; na tyle na ile wszystkie procesy drugorzędne drukują zbiór ciągów. Jednakże zamiast używania pętli opóźnienia
; do synchronizacji procesów pierwszorzędnego i drugorzędnego, ten kod używa synchronizacji barier do
; osiągnięcia tego

            .xlist
            include         stdlib.a
            includelib      stdlib.lib
            .list

dseg        segment para  public  'data'

BarrierSemaph   semaphore   {0}                 ;musimy zainicjalizować SemaCnt zerem
DOSsmaph        semaphore   {}

;Makra do oczekiwania i udostępniania semafora DOS:

DOSWait     macro
            push    es
            push    di
            lesi    DOSsmaph
            WaitSemaph
            pop     di
            pop     es
```

```
                endm
DOSRls          macro
                push    es
                push    di
                lesi    DOSsmaph
                RlsSemaph
                pop     di
                pop     es
                endm

;Makro do synchronizacji  w barierze

Barrier         macro
                local   AllHere,  AllDone
                cmp     es:[di]. Semaphore. SemaCnt,  -(Wait4Cnt-1)
                jle     AllHere
                WaitSemaph
                cmp     es:[di]. Semaphore.SemaCnt,  0
                jge     AllDone
AllHere:        RlsSemaph
AllDone:
                endm

; PCB'y dla naszych procesów drugorzędnych:

BkgndPCB2       pcb     {0, offset  EndStk2, seg  EndStk2}
BkgndPCB2       pcb     {0, offset  EndStk3,  seg EndStk3}

;Dane do drukowania procesów pierwszoplanowych i drugoplanowych:

StrPtrs1        dword   str1_a, str1_b, str1_c, str1_d, str1_e, str1_f
                dword   str1_g, str1_h, str1_i, str1_j, str1_k, str1_l
                dword   0

str1_a          byte    „Foreground: string 'a' “, cr, lf, 0
str1_b          byte    „Foreground: string 'b' “, cr, lf, 0
str1_c          byte    „Foreground: string 'c' “, cr, lf, 0
str1_d          byte    „Foreground: string 'd' “, cr, lf, 0
str1_e          byte    „Foreground: string 'e' “, cr, lf, 0
str1_f          byte    „Foreground: string 'f' “, cr, lf, 0
str1_g          byte    „Foreground: string 'g' “, cr, lf, 0
str1_h          byte    „Foreground: string 'h' “, cr, lf, 0
str1_i          byte    „Foreground: string 'i' “, cr, lf, 0
str1_j          byte    „Foreground: string 'j' “, cr, lf, 0
str1_k          byte    „Foreground: string 'k' “, cr, lf, 0
str1_l          byte    „Foreground: string 'l' “, cr, lf, 0

strPtrs2        dword   str2-a, str2_b, str2_c, str2_d, str2_e, str2_f
                dword   str2_g, str2_h, str2_I
                dword   0
str1_a          byte    „Background: string 'a' “, cr, lf, 0
str1_b          byte    „Background: string 'b' “, cr, lf, 0
str1_c          byte    „Background: string 'c' “, cr, lf, 0
str1_d          byte    „Background: string 'd' “, cr, lf, 0
str1_e          byte    „Background: string 'e' “, cr, lf, 0
str1_f          byte    „Background: string 'f' “, cr, lf, 0
```

```
str1_g          byte    „Foreground: string ‘g’ “, cr, lf, 0
str1_h          byte    „Foreground: string ‘h’ “, cr, lf, 0
str1_i          byte    „Foreground: string ‘i’ “, cr, lf, 0

StrPtrs3        dword   str3_a, str3_b, str_c, str3_d, str3_e, str3_f
                dword   str3_g, str3_h, str3_I
                dword   0
str3_a          byte    „Background 2: string ‘j’ “, cr, lf, 0
str3_b          byte    „Background 2: string ‘k’ “, cr, lf, 0
str3_c          byte    „Background 2: string ‘l’ “, cr, lf, 0
str3_d          byte    „Background 2: string ‘m’ “, cr, lf, 0
str3_e          byte    „Background 2: string ‘n’ “, cr, lf, 0
str3_f          byte    „Background 2: string ‘o’ “, cr, lf, 0
str3_g          byte    „Background 2: string ‘p’ “, cr, lf, 0
str3_h          byte    „Background 2: string ‘q’ “, cr, lf, 0
str3_i          byte    „Background 2: string ‘r’ “, cr, lf, 0

dseg            ends

cseg            segment  para public  ‘code’
                assume  cs:cseg, ds:dseg

; Zastąpienie programu obsługi błędu  krytycznego. Ten podprogram wywołuje prcsquit jeśli
; użytkownik zadecyduje się przerwać program

CritErrMsg      byte    cr, lf
                byte    “DOS Critical Error!”, cr, lf
                byte    “A)bort, R)etry, I)gnore, F)ail? $”

MyInt24         proc    far
                push    dx
                push    ds
                push    ax

                push    cs
                pop     ds
Int24Lp:        lea     dx, critErrMsg
                mov     ah, 9                   ; funkcja DOS’a drukowania ciągu
                int     21h

                mov     ah, 1                   ;funkcja DOS’a odczytu znaku
                int     21h
                and     al., 5Fh                ;konwertuje l.c → u.c

                cmp     al., ‘I’                ;ignorujemy?
                jne     NotIgnore
                pop     ax
                mov     al., 0
                jmp     Quit24

NotIgnore:      cmp     al., ‘r’                ;ponowić?
                jne     NotRetry
                pop     ax
                mov     al, 1
                jmp     Quit24
```

```
NotRetry:       cmp     al, 'A'                 ;przerywamy?
                jne     NotAbort
                prcsquit                        ;jeśli wychodzimy, zmieniamy INT 8
                pop     ax
                mov     al., 2
                jmp     Quit24

NotAbort:       cmp     al, 'F'
                jne     BadChar
                pop     ax
                mov     al, 3
Quit24:         pop     ds
                pop     dx
                iret
BadChar:        mov     ah, 2
                mov     dl, 7                           ;znak dzwonka
                jmp     Int24Lp
MyInt24         endp

; Będziemy blokować INT 23 (wyjątek break)

MyInt23         proc    far
                Iret
MyInt23         endp

; Te procesy drugoplanowe wywołują DOS do drukowania kilku ciągów na ekranie. Tymczasem proces
; drugorzędny  również drukuje ciągi na ekranie. Uniemożliwiając współużywalności  , lub przynajmniej
; bałaganowi na ekranie, kod ten używa semaforów do ochrony funkcji DOS. Dlatego też, każdy proces
; będzie drukował jedną skończoną  linię potem udostępnia semafor. Jeśli inny proces oczekuje, będzie drukował
; swoje linie.

BackGround1     proc
                mov     ax, dseg
                mov     ds, ax

; Czekanie czy wszyscy inni są gotowi:

                lesi    BarrierSemaph
                barrier 3

; Okay, zaczynamy drukować ciągi:

                lea     bx, StrPtrs2                    ;tablica wskaźników ciągów
PrintLoop:      cmp     word  ptr [bx+2], 0             ;koniec wskaźników?
                je      BkGndDone
                les     di, [bx]                        ;pobranie ciągu do druku
                DOSWait
                puts                                    ;wywołanie DOS dla drukowania ciągu
                DOSRls
                add     bx, 4                           ;wskazuje kolejny wskaźnik ciągu
                jmp     PrintLoop

BkGndDone:      die
BackGround1     endp

BackGround2     proc
```

```
                mov     ax, dseg
                mov     ds,a x

                lesi    BatrrierSema
                barrier 3

                lea     bx, StrPtrs3            ;tablica wskaźników do ciągów
Print Loop:     cmp     word ptr  [bx+2], 0     ;koniec wskaźników?
                je      BkGndDone
                les     di, [bx]                ;pobranie ciągu do druku
                DOSWait
                puts                            ;funkcja DOS drukowania ciągu
                DOSRls
                add     bx, 4                   ;wskazuje kolejny wskaźnik do ciągu
                jmp     PrintLoop

BkGndDone:      die
BackGFround2    endp

Main            proc
                mov     ax, dseg
                mov     ds, ax
                mov     es, ax
                meminit

; Inicjalizujemy wektory obsługi wyjątków INT 23h i INT 24h

                mov     ax, 0
                mov     es, ax
                mov     word ptr es:[24h*4],  offset  MyInt24
                mov     es:[24h*4 + 2], cs
                mov     word  ptr es:[23h*4],  offset MyInt23
                mov     es:[23h*4+2], cs

                prcsinit                        ;zaczynamy system wielozadaniowy

; Zaczynamy pierwszy z procesów drugorzędnych:

                lesi    BkgndPCB2               ;odpalamy nowy proces
                fork
                test    ax, ax                  ; wracamy do procesu macierzystego?
                je      StartBG2
                jmp     BackGround1             ;wracamy do  podrzędnego

; Zaczynamy drugi proces podrzędny:

StartBG2;       lesi    BkgndPCB3               ;odpalamy nowy proces
                fork
                test    ax, ax                  ;powrót do procesu macierzystego
                je      ParentPrcs
                jmp     BackGround2             ; wracamy do podrzędnego

; Proces macierzysty będzie drukował  grupę ciągów w tym samym czasie co proces drugorzędny.
; Będziemy używali semafora DOS do ochrony funkcji DOS, które robi PUTS.

ParentPrcs:     lesi    BarrierSemaph
```

```
                barrier  3

                lea      bx, StrPtrs1                ;Tablica wskaźników do ciągów
PrintLoop;      cmp      word  ptr [bx+2], 0         ;koniec wskaźników
                je       ForeGndDone
                les      di, [bx]                    ;pobranie ciągu do druku
                DOSWait
                Puts                                 ;funkcja DOS do drukowania ciągu
                DOSRls
                add      bx, 4                       ;wskazuje następny ciąg do wskaźnika
                jmp      PrintLoop

ForeGndDone:    prcsquit
Quit:           ExitPgm
Main            endp

cseg            ends
sseg            aegment  para stack ‘stack’

; Tu są stosy dla procesów drugorzędnych

stk2            byte     1024  dup  (?)
EndStk2         word     ?

stk3            byte     1024 (?)
EndStk3         word     ?

; Tu jest stos dla programu głównego / procesu pierwszoplanowego

stk             byte     1024  dup (?)
sseg            ends

zzzzzzseg       segment para public ‘zzzzzz’
LastBytes       db       16  dup  (?)
Zzzzzzseg       ends
                end      Main
```

Przykładowe dane wyjściowe:

```
Background 1: string ‘a’
Background 1: string ‘b’
Background 1: string ‘c’
Background 1: string ‘d’
Background 1: string ‘e’
Background 1: string ‘f’
Background : string ‘a’
Background 1: string ‘g’
Background 2: string ‘j’
Background : string ‘b’
Background 1: string ‘h’
Background 2: string ‘k’
Background : string ‘c’
Background 1: string ‘i’
Background 2: string ‘l’
Background : string ‘d’
Background 2: string ‘m’
```

Background: string ‘e’
Background 2: string ‘n’
Background : string ‘f’
Background 2: string ‘o’
Background : string ‘g’
Background 2: string ‘p’
Background : string ‘h’
Background 2: string ‘q’
Background : string ‘i’
Background 2: string ‘r’
Background : string ‘j’
Background : string ‘k’
Background : string ‘l’

Zanotujmy jak proces drugorzędny numer jeden działa o jeden cykl zegarowy przed innymi procesami oczekującymi na semafor DOS. Po tej początkowej grupie, wszystkie procesy jeden po drugim wywołują DOS.

## 19.6 ZAKLESZCZENIE

Chociaż semafory mogą rozwiązać każdy problem synchronizacji, nie odnieśmy wrażenia, że semafory nie wprowadzają swoich własnych problemów. Jak już widzieliśmy, niewłaściwe zastosowanie semaforów może wpłynąć na nieskończone zawieszenie się procesu oczekiwania w kolejce semaforu. Jednakże, nawet jeśli oczekujemy i sygnalizujemy pojedyncze semafory, jest całkiem możliwa poprawna operacja na kombinacjach semaforów dających taki sam efekt .Nieskończone zawieszanie procesu z powodu problemów z semaforami jest poważną kwestią. Ta denerwująca sytuacja jest znana zakleszczenie lub martwy punkt.

Zakleszczenie wystąpi kiedy jeden proces przechowuje zasób i oczekuje na inny podczas gdy drugi proces przechowuje ten zasób i oczekuje na pierwszy. Aby zobaczyć jak może wystąpić zakleszczenie rozpatrzmy następujący kod:

```
; Process one:

        lesi    Semaph1
        WaitSemaph
< Zakładamy wystąpienie przerwania tutaj >

        lesi    Semaph2
        WaitSemaph
        -
        -
        -
; Process two:

        lesi    Semaph2
        WaitSemaph
        lesi    Semaph1
        WaitSemaph
        -
        -
        -
```

Proces one chwyta semafor powiązany z Semaph1. Potem pojawia się przerwanie zegarowe które powoduje przestawienie kontekstu do procesu two. Proces two chwyta semafor powiązany z Semaph2 i potem próbuje dostać Semaph1. jednakże, proces one już przechowuje Semaph1, więc proces two blokuje i czeka na proces one i udostępnienie tego semafora To powoduje (ostatecznie) przekazanie sterowania do procesu one. Proces one próbuje wtedy uchwycić Semaph2. Niestety, proces dwa już przechowuje Semaph2, więc proces one blokuje czekając na

Semaph2. teraz oba procesy są zablokowane czekając na inny. Od tego czasu żaden proces nie może się uruchomić, żaden proces nie może udostępnić semaforu dla innego potrzebującego. Oba procesy są zakleszczone.

Jednym z łatwych sposobów uniknięcia zakleszczenia jest nigdy nie pozwolić aby proces przechowywała więcej niż jeden semafor w czasie. Niestety, nie jest to rozwiązanie praktyczne; wiele procesów może potrzebować zastrzeżonego dostępu do kilku zasobów w jednym czasie. Jednakże, możemy obmyślić inne rozwiązanie poprzez zaobserwowanie wzorca, który doprowadził do zakleszczenia w poprzednim przykładzie. Zakleszczenie następuje ponieważ dwa procesy przechwytują różne semafory a potem próbują przechwycić semafor, który przechowuje inny proces. Innymi słowy, przechwytują dwa semafory w różnym porządku ( proces one przechwytywał najpierw Semaph1 potem Semaph2, proces two najpierw Semaph2 a potem Semaph1). Okazuje się, ze dwa procesy nigdy się nie zakleszczą jeśli oczekują one na semafory w takim samym porządku. Możemy zmodyfikować poprzedni przykład eliminując możliwość zakleszczenia:

; Process one:

```
lesi      Semaph1
WaitSemaph
lesi      Semaph2
WaitSemaph
-
-
-
```

Process two:

```
lesi      Semaph1
WaitSemaph
lesi      Semaph2
WaitSemaph
-
-
-
```

Teraz, obojętnie gdzie powyżej wystąpi przerwanie, nie wystąpi zakleszczenie. Jeśli przerwanie wystąpi pomiędzy drugim WaitSemaph wywołanym w procesie one , kiedy proces dwa próbuje oczekiwać Semaph1, będzie zablokowane a proces one będzie kontynuował z dostępnym Semaph2.

Łatwym sposobem na unikanie problemów z zakleszczeniem jest liczba wszystkich zmiennych semaforowych i upewnienie się, ze wszystkie procesy posiadają (obsługują) semafory od najmniejszej liczby semaforów do największej. Zakłada to ,że wszystkie procesy posiadają semafory w takim samym porządku, i potem zakłada ,że zakleszczenie nie wystąpi.

Zauważmy, że to założenie posiadania semaforów stosuje tylko semafory, które proces przechowuje jednocześnie. Jeśli proces potrzebuje semafora sześć na chwilę, a potem potrzebuje semafora dwa po tym jak udostępnił semafor sześć, nie ma żadnego problemu posiadania semafora dwa po udostępnieniu semafora sześć. Jednakże, jeśli w jakimś punkcie proces musi przechować oba semafory, musi posiadać najpierw semafor dwa.

Procesy mogą udostępniać semafory w dowolnym porządku. Porządek w jakim proces udostępnia semafory nie wpływa na to czy zakleszczenie może wystąpić. Oczywiście, procesy powinny udostępniać semafory jak tylko proces upora się z zasobem chronionym przez semafor; może to być inny proces obsługujący ten semafor. Powyższy schemat działa i jest łatwy do implementacji nie jest to absolutnie jedyny sposób obsługi zakleszczenia, i nie jest zawsze najbardziej wydajny. Jednakże jest prosty do implementacji i zawsze działa.

## 19.7 PODSUMOWANIE

Pomimo faktu, że DOS nie jest współużytkowalny i nie wspiera bezpośrednio wielozadaniowości, co nie znaczy, że nasza aplikacja nie może być wielozadaniowa; jest trudno sprawić aby różne aplikacje działały niezależnie jedna od drugiej pod DOS'em.

Chociaż DOS nie przełącza pomiędzy różnymi programami w pamięci, DOS z pewnością zezwala na załadowanie wielu programów do pamięci w jednym czasie. Jedynie jednak tylko jeden taki program jest wykonywany na bieżąco. DOS dostarcza kilku funkcji do załadowania i wykonania plików „.EXE" i „.COM" z

dysku. Procesy te zachowują się jak wywołania podprogramów, ze zwracaniem sterowania do programu wywołującego taki program tylko po zakończeniu programu „potomka”

*”Procesy DOS”
*”Procesy potomne w DOS”
*”Ładowanie i wykonywanie”
*”Ładowanie programu”
*”Ładowanie nakładek”
*”Zakończenie procesu”
*” Uzyskanie kodu powrotu i procesu potomnego”

Podczas wykonywania procesu DOS mogą wystąpić pewne błędy, które przekazują sterowanie do programu obsługi wyjątków. Poza wyjątkami 80x86, DOS’owe programy obsługi break i błędu krytycznego są podstawowymi przykładami. Każdy program który dopasowuje wektory przerwań powinien dostarczyć swojego własnego programu obsługi wyjątków dla tych warunków więc może przywrócić przerwania błędu wyjątku ctrl-C lub I/O. Co więcej dobrze napisany program zawsze dostarcza zastępczego programu obsługi dla tych dwóch warunków, które dostarczają lepszego wsparcia niż domyślne programy DOS’a.

*”Obsługa wyjątków w DOS: Obsługa break”
*”Obsługa wyjątków w DOS: Obsługa błędu krytycznego”
*”Obsługa wyjątków w DOS: Przerwania kontrolowane”

Kiedy proces macierzysty wywołuje proces potomny funkcjami LOAD lub LOADEXEC, proces potomny dziedziczy wszystkie otwarte pliki z procesu macierzystego. W szczególności, proces potomny dziedziczy urządzenia standardowego wejścia, standardowego wyjścia , standardowego błędu, pomocniczego I/O i drukarki. Proces macierzysty może łatwo przekierować I/O do/z tego urządzenia przed przekazaniem sterowania do procesu potomnego. To w praktyce przekierowuje I/O podczas wykonywania procesu potomnego

*”Przekierowanie I/O dla procesu potomnego”

Kiedy programy DOS’owe chcą skomunikować się jeden z drugim, zazwyczaj odczytują lub zapisują dane do pliku. Jednak tworzenie, otwieranie, odczytywanie i zapisywanie plików to dużo pracy, zwłaszcza przy dzieleniu kilku wartości zmiennych . Lepszą alternatywą jest użycie pamięci dzielonej. Niestety DOS nie dostarcza wsparcia dla dwóch programów, aby mogły dzielić wspólny blok pamięci. Jednakże bardzo łatwo jest napisać TSR, który zarządza pamięcią dzieloną dla różnych programów

*”Pamięć dzielona”
*”Statyczna pamięć dzielona”
*”Dynamiczna pamięć dzielona”

Funkcja współprogramu jest podstawowym mechanizmem dla przełączania sterowania między dwoma procesami. Operacja „współwywołania” jest odpowiednikiem podprogramu wywołania i zwraca wszystko zawinięte do jednej operacji. Współwywołanie przekazuje sterowanie do jakiegoś innego procesu. Kiedy jakiś inny proces zwraca sterowanie do współprogramu (poprzez współwywołanie) sterowanie zaczyna się od pierwszej instrukcji po współwywołującym kodzie. Biblioteka Standardowa UCR dostarcza całkowitego wsparcia dla współprogramów więc możemy łatwo wstawić współprogramy do naszego programu asemblerowego.

*”Współprogramy”

Chociaż możemy użyć współprogramów do symulowani wielozadaniowości (wielozadaniowość równoległa), głównym problemem ze współprogramami jest to, że każda aplikacja musi zadecydować kiedy przełączyć do innego procesu poprzez współwywołanie.. Chociaż eliminuje to pewne problemy współużytkowalności i synchronizacji, decydowanie kiedy i gdzie zrobić takie wywołanie zwiększa pracę konieczną do napisania aplikacji wielozadaniowej. Lepszym podejściem jest zastosowanie wielozadaniowości z wywłaszczeniem gdzie przerwanie zegarowe wykonuje przełączanie kontekstowe. Problemy współużytkowania i synchronizacji rozwijają się w takim systemie, ale przy ostrożności problemy te są do przezwyciężenia.

*”Wielozadaniowość”
*”Procesy nieskomplikowane i skomplikowane”
*”Pakiet process Biblioteki Standardowej UCR”
*”Problemy z wielozadaniowością”
*”Przykład programu z wątkami”

Wielozadaniowość z wywłaszczeniem otwiera puszkę Pandory. Chociaż wielozadaniowość czyni pewne programy łatwiejszymi do implementacji, , problemy synchronizacji procesu i współużywalności są groźne w systemie wielozadaniowym .Wiele procesów wymaga jakiegoś rodzaju zsynchronizowanego dostępu do zmiennych globalnych. Dalej, większość procesów będzie musiało wywołać DOS, BIOS lub jakiś inny podprogram (np. Biblioteka Standardowa), która nie jest współużywalna. Jakoś musimy kontrolować dostęp do takiego kodu aby wielokrotne procesy nie wpływały niekorzystnie na inne. Synchronizacja jest osiągalna przez używanie kilku różnych technik. W kilu prostych przypadkach możemy po prostu wyłączyć przerwania, eliminując problem współużywalności. W innych przypadkach możemy użyć testuj i ustaw lub semaforów do ochrony regionów krytycznych.

*”Synchronizacja”
*”Operacje nierozdzielne, testuj i ustaw i aktywne oczekiwanie’
*”Semafory”
*”Wsparcie semaforów Biblioteki Standardowej”
*”Używanie semaforów do ochrony regionów krytycznych’
*”Używanie semaforów do synchronizacji bariery”

Stosując obiektów synchronizacji, takich jak semafory, możemy wprowadzić nowy problem do systemu. Zakleszczenie jest doskonałym przykładem. Zakleszczenie wystąpi kiedy jeden proces przechowuje jakieś zasoby i chce innego a drugi proces przechowuje żądany zasób i chce zasobu przechowywanego przez pierwszy proces. Możemy łatwo uniknąć zakleszczenia poprzez kontrolowanie porządku w jakim różne procesy nabywają grupy semaforów.

*”Zakleszczenie”