Katedra Architektury Systemów Komputerowych

Laboratorium
Przetwarzanie Rozproszone

Instrukcje wprowadzające do ćwiczeń laboratoryjnych

Gdańsk 2018

Ćwiczenie 1

DYNAMICZNE STRUKTURY DANYCH

1.1 Wstęp

W praktyce programistycznej bardzo często znajdują zastosowanie rekursywne struktury danych. Typowymi ich przykładami są listy jedno- i dwukierunkowe, stos oraz drzewa. Konstrukcja takich obiektów wymaga znajomości złożonych typów danych oraz technik dynamicznego przydziału pamięci. Tematem niniejszego opracowania jest omówienie mechanizmów języka Borland C pozwalających na konstruowanie struktur rekursyjnych.

1.2 Dynamiczna alokacja pamięci

Dynamiczna alokacja pamięci polega na rezerwowaniu obszarów pamięci operacyjnej na żądanie w trakcie działania programu. Mechanizm ten stosowany jest do tworzenia struktur danych o nieznanej z góry wielkości. Język Borland C oferuje grupę kilku funkcji dynamicznego przydziału i zwalniania pamięci. Funkcje przydziału pamięci zwracają tzw. wskaźnik uniwersalny typu `void*`. Jest to wskaźnik na dowolny typ. Dobra praktyka programistyczna wymaga jawnego rzutowania go na typ stosowanego obiektu. Zostanie to później omówione na przykładzie. Prototypy omawianych niżej funkcji znajdują się w zbiorze nagłówkowym `alloc.h`.

Podstawową funkcją dynamicznego przydziału pamięci jest `malloc()`. Jej prototyp jest następujący:

```
void* malloc(size_t size)
```

Parametrem funkcji jest liczba rezerwowanych bajtów. W przypadku błędu zwracana jest wartość `NULL`. Funkcja pozwala rezerwować obszary pamięci nie przekraczające 64KB. Przydzielona pamięć nie jest zerowana.

Obok opisanej wyżej funkcji, która nie inicjalizuje rezerwowanego obszaru, dostępna jest `calloc()` zerująca przydzieloną pamięć:

```
void* calloc(size_t noitems, size_t size)
```

Funkcja alokują bloki `(noitems * size)` bajtów i wypełnia je zerami.

Zwalnianiu dynamicznie rezerwowanych obszarów służy funkcja `free()`. Jej prototyp jest następujący:

```
void free(void *);
```

Przykład użycia funkcji `malloc()` i `free()`:

```
#include <string.h>
#include <stdio.h>
#include <alloc.h>

void main(void)
{
   char *lancuch;
   lancuch = (char *) malloc(10); /* przydziel pamiec */
   strcpy(lancuch, "HELLO"); /* skopiuj napis */
```

```
   printf("\nLancuch zawiera napis: %s", lancuch);
   free(lancuch); /* zwolnij przydzielona pamiec */
}
```

Uwaga! Przy alokacji pamięci zastosowane zostało jawne rzutowanie typu `void*` na typ wskaźnika używanego obiektu, czyli `char*`.

1.3 Struktury

1.3.1 Deklaracja struktury i definicja zmiennych strukturalnych

Obiekt składający się z kilku zmiennych różnych typów nazywany jest w języku C strukturą (struktury w języku C są analogiczne do rekordów w Pascalu). Deklaracja struktury składa się ze słowa kluczowego `struct`, opcjonalnie występującego po nim identyfikatora (nazwy struktury) oraz z ujętej w nawiasy klamrowe listy składowych oddzielonych średnikami. Deklaracja struktury musi być zakończona średnikiem.

```
struct [struct_name]
{
   type_1 field_1;
   type_2 field_2;
   ...
   type_n field_n;
};
```

Składowe struktury mogą być dowolnego typu prostego lub złożonego, a w szczególności tego samego co typ deklarowanej struktury. W takim przypadku mamy do czynienia ze strukturą rekursywną. (O takich konstrukcjach będzie mowa w części 4).

Deklaracja struktury definiuje typ. Zmienne tego typu można definiować w dwojaki sposób. Pierwszy polega na umieszczeniu nazwy zmiennej (listy nazw) przed średnikiem kończącym deklarację struktury.

```
struct
{
   type_1 field_1;
   type_2 field_2;
   ....
   type_n field_n;
} var_1,var_2,...var_n;
```

Jeżeli deklaracja struktury zawiera nazwę, to może być ona użyta do definiowania zmiennych w zwyczajnyśposób.

```
struct struct_name var_1, var_2,...var_n;
```

Sama deklaracja struktury nie rezerwuje pamięci, a tylko opisuje wzorzec typu złożonego. Zmienne strukturalne można inicjalizować umieszczając po ich definicji listę wartości początkowych będących stałymi.

```
stuct struct_name var_1, var_2,...var_n = {const_1, const_2,...const_n};
```

1.3.2 Operacje na strukturach i ich składowych

Do składowej struktury można odwołać się poprzez operator `.` (kropka). Umożliwia to następująca konstrukcja:

```
struct_name.field
```

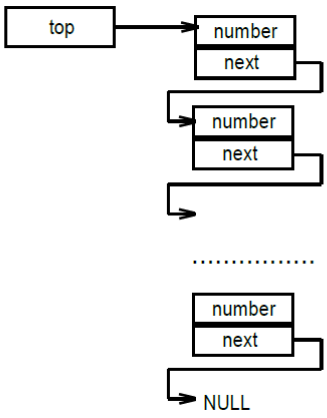
Dodatkowo dozwolone jest przypisywanie struktur, ich kopiowanie, pobranie adresu za pomocą operatora `&`, przekazywanie do funkcji oraz zwracanie jako wartości funkcji. Struktur nie można porównywać.

Zwykle, w przypadku konieczności przesłania struktury do funkcji, przekazuje się wskaźnik zamiast kopiować całą jej zawartość. Wskaźnik do struktury definiuje się tak samo jak do innych obiektów.

```
struct struct_name *pp;
```

W takim przypadku pp jest adresem struktury, a `(*pp).field` jej składową (należy pamiętać o umieszczeniu nawiasów, gdyż operator składowej struktury ma wyższy priorytet od operatora adresowania pośredniego *wyłuskania*).

Bardzo częste stosowanie wskaźników do struktur spowodowało wprowadzenie specjalnego operatora `->`. Jeżeli pp jest wskaźnikiem (adresem) do struktury, to `pp->` field jest jej składową.



Rysunek 1.1: Struktura stosu (dynamicznej listy LIFO)

1.4 Przykłady konstrukcji rekursywnych

Konstruowanie rekursywnych typów danych umożliwia własność struktur pozwalająca na deklarowanie składowych o dowolnym typie złożonym. Przeanalizujmy prosty przykład, którym jest stos (kolejka LIFO). Zadeklarujmy strukturę opisującą stos przechowujący obiekty typu integer.

```
struct stack_struct
{
   int number;
   struct stack_struct *next;
} *top = NULL;
```

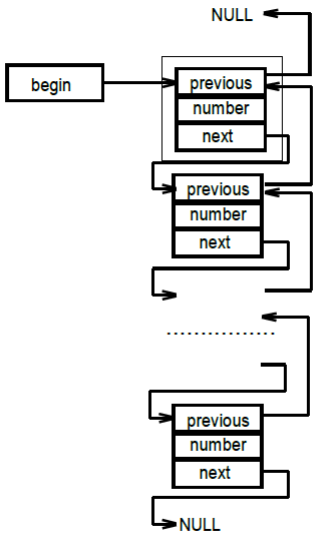
Powyższa struktura zawiera dwie składowe. Pierwszą z nich jest przechowywana liczba, drugą adres (wskaźnik) następnego elementu. Należy zwrócić uwagę, że jej typem jest wskaźnik na deklarowaną strukturę Elementy stosu umieszczone są w dowolnych miejscach przestrzeni adresowej. Wskaźniki pozwalają w każdej sytuacji zlokalizować następny element. Przyjęcie takiego rozwiązania pozwala na połączenie zbioru elementów w jednolitą listę. Wskaźnik `next` ostatniego elementu listy ma wartość NULL. Pozwala to na zidentyfikowanie tego elementu przez funkcję przeszukującą listę. Zdefiniowana została również zmienna `root` wskazująca wierzchołek stosu. Jej inicjalną wartością jest NULL (oznacza to, że stos jest inicjalnie pusty). Konstruowaną strukturę obrazuje Rys. 1.1.

Możemy przystąpić do napisania funkcji operujących na opisanej strukturze. Na stosie można wykonać dwie operacje: umieszczenie elementu na jego wierzchołku i zdjęcie elementu z wierzchołka. Operacje te będą realizowane przez funkcje `push()` oraz `pop()`.

```
void push(int element)
{
   struct stack_struct *p;
   p = top;
   top = (struct stack_struct *)malloc(sizeof(struct stack_struct));
   top->number = element;
   top->next = p;
}

int pop(void)
{
   int element = 0;
   struct stack_struct *p;

   if(top != NULL)
   {
      p = top;
      element = top->number;
      top = top->next;
      free(p);
   }
```



Rysunek 1.2: Struktura listy dwukierunkowej

```
   return element;
}
```

Funkcja `push()` zapamiętuje w zmiennej tymczasowej aktualną wartość wskaźnika wierzchołka stosu. Następnie alokuje blok pamięci dla nowego elementu i jego adres podstawia do wskaźnika wierzchołka stosu (wskazuje on zawsze na ostatnio wpisany element). Następnie wartość umieszczana na stosie jest wpisywana do składowej `number`. Wskaźnik na następny element jest inicjowany poprzednią wartością wierzchołka stosu.

Funkcja `pop()` sprawdza, czy stos nie jest pusty, porównując wskaźnik wierzchołka z wartością `NULL`. Następnie zapamiętuje w zmiennych tymczasowych wartości odczytywanego elementu oraz aktualnego wskaźnika wierzchołka stosu. W kolejnym kroku zostaje przerwane połączenie zdejmowanego elementu ze stosem poprzez podstawienie pod wskaźnik wierzchołka adresu następnego elementu. Na końcu zostaje zwolniony blok pamięci.

Innym, nieco bardziej złożonym, przykładem jest dynamiczna lista liczb całkowitych ułożonych rosnąco. Zadeklarujmy strukturę opisującą element listy dwukierunkowej.

```
struct element
{
   struct element *previous;
   int number;
   struct element *next;
} *begin;
```

Powyższa struktura zawiera trzy składowe. Składowa `number` jest liczbą przechowywaną na danej pozycji listy. Dwie pozostałe są wskaźnikami odpowiednio do poprzedniego i następnego elementu. Podobnie jak w przypadku stosu, elementy listy rozmieszczone są w dowolnych miejscach pamięci operacyjnej. Wskaźniki (zawierające adresy sąsiednich elementów) pozwalają zlokalizować element poprzedzający i następny. Przyjęcie takiego rozwiązania pozwala na połączenie zbioru elementów w listę. W odróżnieniu od stosu wprowadzone są dwa wskaźniki: na element poprzedni i na element następny. Takie rozwiązanie pozwoli później na pisanie funkcji przeglądających listę w obie strony.

Konstruowana struktura schematycznie przedstawiona jest na 1.2. Pierwszy element listy (zakreślony linią przerywaną) jest wyróżniony. Jest on umieszczany na liście inicjalnie, gdyż wymaga się aby zawierała ona co najmniej jeden element. Nie przechowuje on danej. Wskaźniki `previous` pierwszego oraz `next` ostatniego elementu listy mają wartość `NULL`. Umożliwia to zidentyfikowanie tych elementów przez funkcję przeszukującą listę. Dodatkowo wprowadzony został wskaźnik `begin` pozwalający zlokalizować pierwszy element listy.

Tak zadeklarowana struktura pozwoli na napisanie funkcji wpisującej liczby na listę z zachowaniem rosnącego uporządkowania.

```
struct element * insert(int obj, struct element *ptr)
{
   struct element *p;
   p = ptr->next; /* wez adres analizowanego elementu */

   if(p != NULL) /* czy koniec listy */
   {
      if(obj > (p->number))
      {
         p->next = insert(obj, p);
      }
      else
      {
         /* utworz nowy element */
         p->previous = (struct element *) malloc(sizeof(struct element));

         /* zapamietaj adres nowego elementu */
         p = p->previous;

         /* zainicjuj skladowe nowego elementu */
         p->number = obj;
         p->next = ptr->next;
         p->previous = ptr;
      }
   }
   else /* koniec listy */
   { /* utworz nowy element */
      p = (struct element *)malloc(sizeof(struct element));

      /* zainicjuj skladowe nowego elementu */
      p->number = obj;
      p->next = NULL;
      p->previous = ptr;
   }

   return p; /* zwroc adres nowego elementu */
}
```

Zdefiniowana wyżej funkcja wstawiająca liczby do dwukierunkowej listy uporządkowanej ma dwa parametry formalne. Pierwszym z nich jest wstawiana liczba, drugim wskaźnik do elementu listy poprzedzającego aktualnie analizowany. Funkcja zwraca adres nowo utworzonego elementu. Skonstruowana jest w sposób rekurencyjny. Jest to praktyka zwykle stosowana przy manipulacji rekursywnymi strukturami danych.

Inicjalnie funkcja wołana jest z parametrem `begin` (wskaźnikiem na początek listy), a wartość zwracana przez funkcję podstawiana jest pod adres drugiego elementu listy (`begin->next`). W pierwszym kroku pod zmienną pomocniczą p pod stawiany jest adres elementu następnego (ponieważ analizowany będzie element następny za wskazywanym przez parametr funkcji). Następnie funkcja sprawdza, czy kolejny element istnieje (czy wskaźnik jest różny od `NULL`). Jeśli tak, to wstawiana liczba porównywana jest z przechowywaną w tym elemencie listy. Jeśli wstawiana liczba jest większa, to funkcja jest wołana rekurencyjnie z adresem następnego elementu listy. Jeżeli wstawiana liczba jest mniejsza, bądź równa, to funkcja tworzy nowy element po wskazywanym przez parametr aktualny `ptr`. W tym celu rezerwowany jest odpowiedni obszar pamięci, a jego adres wpisywany do wskaźnika na element poprzedni od analizowanego. Do składowych nowo utworzonego elementu są wpisywane odpowiednie wartości. Jeżeli wskaźnik do kolejnego elementu jest pusty, to osiągnięty został koniec listy (lista była pusta albo wpisywana liczba okazała się większa od wszystkich przechowywanych w liście). W takim przypadku należy utworzyć nowy element, umieścić w nim liczbę, wskaźnik na element następny zainicjować wartością `NULL`, a do wskaźnika na element poprzedni wpisać wartość drugiego parametru funkcji (`ptr`), czyli adres poprzedniego elementu listy.

Uwaga! przed wywołaniem funkcji `insert()` musi zostać utworzony pierwszy (fikcyjny) element listy, a oba jego wskaźniki zainicjowane wartością `NULL`. Adres tego elementu powinien być wpisany do zmiennej `begin`.

```
..............
```

```
begin = (struct element *) malloc(sizeof(struct element));
begin->next = NULL;
begin->previous = NULL;
.............
begin->next = insert(i, begin);
```

Jeszcze innym przykładem rekursywnej struktury danych jest drzewo binarne przechowujące w węzłach elementy typu `int`. Opisująca je struktura ma postać:

```
struct tree
{
   int element;
   struct tree *left;
   struct tree *right;
};
```

Zakładając, że zmienna `root` wskazuje na wierzchołek tak zdefiniowanego drzewa można napisać rekursywną funkcję wypisującą wszystkie elementy umieszczone w węzłach.

```
void scan(struct tree root)
{
   if(root != NULL)
   {
      printf("\n element = %d", root->element);
      scan(root->left);
      scan(root->right);
   }
}
```

Ćwiczenie 2

BIBLIOTEKI STATYCZNE I DYNAMICZNE W SYSTEMIE LINUX

2.1 Wstęp

Podobnie jak w innych systemach programowania, także w Linuksie (i Unixie) istnieje możliwość tworzenia i dołączania programów bibliotecznych. Wyróżnia się biblioteki statyczne i wspólne, które w innych systemach nazywane są zwykle bibliotekami dynamicznymi. Kod zawarty w bibliotekach statycznych dołączany do programu w fazie linkowania, natomiast kod w bibliotekach dynamicznych (wspólnych) udostępniany jest w trakcie wykonywania programu.

2.2 Tworzenie biblioteki statycznej (archiwum)

Punktem wyjścia do tworzenia biblioteki statycznej jest utworzenie plików zawierających funkcje biblioteczne w formacie półskompilowanym. Przyjmijmy, że funkcje biblioteczne umieszczone w dwóch plikach `dod.c` i `odej.c`.

```
/* Plik dod.c - testowanie bibliotek statycznych w Linuksie */
int dodaj(int a, int b)
{
   int suma;
   suma = a + b;
   return suma;
}

int dodaj_kwadraty(int a, int b)
{
   int suma;
   suma = a*a + b*b;
   return suma;
}
```

```
/* Plik odej.c - testowanie bibliotek statycznych w Linuksie */
int odejmij(int a, int b)
{
   int roznica;
   roznica = a -b;
   return roznica;
}

int odejmij_kwadraty(int a, int b)
{
   int wartosc;
```

```
    wartosc = a*a -b*b;
    return wartosc;
}
```

W celu przekształcenia funkcji zawartych w podanych plikach źródłowych na format półskompilowany, wykonuje się kompilację z parametrem `-c`, np.

```
$ gcc -c dod.c -o dod.o
$ gcc -c odej.c -o odej.o
```

W wyniku powyższej kompilacji, jeśli kody źródłowe w plikach są bezbłędne, zostaną utworzone pliki półskompilowane `dod.o` i `odej.o`. Przyjmując, że dysponujemy plikami półskompilowanymi `dod.o` i `odej.o` możemy utworzyć bibliotekę statyczną za pomocą polecenia `ar` (archiwum):

```
$ ar r lib_arytm.a dod.o odej.o
```

Parametr `r` poleca utworzenie biblioteki (archiwum) o nazwie `lib_arytm.a` i dodanie do niej wymienionych modułów. Jeśli podane moduły znajdowały się wcześniej w bibliotece, to zostaną zastąpione przez aktualne wersje. Nazwa biblioteki powinna zaczynać się od liter `lib`, a rozszerzenie nazwy powinno mieć postać `.a`, np. `lib_arytm.a`.

Informacje o plikach, które weszły w skład archiwum można uzyskać za pomocą polecenia `ar` z parametrem `tv`, np.:

```
$ ar tv lib_arytm.a
```

Uzyskaną w ten sposób bibliotekę można wykorzystać w innych programach. W celu praktycznego zapoznania się z sposobami łączenia wykorzystamy poniższy program w języku C.

```
/* Plik test_bibl.c - testowanie bibliotek statycznych w Linuksie */
#include <stdio.h>
int main()
{
    int suma, sumakw;
    suma = dodaj(5, 12);
    sumakw = dodaj_kwadraty(3, 5);
    printf("\nSuma = %d Suma kwadratow = %d\n", suma, sumakw);
    return 0;
}
```

Powyższy program trzeba poddać kompilacji z parametrem `-c` wskutek czego uzyskamy plik `test_bibl.o` w formacie półskompilowanym:

```
$ gcc -c test_bibl.c -o test_bibl.o
```

Następnie trzeba wykonać linkowanie ze wskazaniem biblioteki statycznej

```
$ gcc test_bibl.o lib_arytm.a -o wynik.out
```

W rezultacie uzyskamy program gotowy do wykonania zawarty w pliku `wynik.out`. W celu uruchomienia programu wystarczy napisać polecenie

```
$ ./wynik.out
```

2.3 Tworzenie biblioteki dynamicznej (wspólnej)

Opis zasad tworzenia biblioteki dynamicznej przedstawimy na przykładzie funkcji zawartych w plikach źródłowych `dod.c` i `odej.c`, których zawartość została podana w p. 2. Kompilacja plików jest nieco bardziej skompilowana niż w przypadku bibliotek statycznych i wymaga podania poniższych poleceń

```
$ gcc -c -Wall -fPIC -D_GNU_SOURCE dod.c
$ gcc -c -Wall -fPIC -D_GNU_SOURCE odej.c
```

Opcja `-fPIC` powoduje generowanie kodu przesuwnego. Pominięcie tej opcji powoduje, że dla każdego procesu korzystającego z biblioteki dynamicznej (wspólnej) konieczne będzie tworzenie odrębnej kopii biblioteki - zatem podstawowe korzyści ze stosowania bibliotek dynamicznych nie będą wykorzystane. Dodajmy, że biblioteka dynamiczna znajduje się w każdym procesie, który z niej korzysta, pod innym adresem wirtualnym.

Następnie można utworzyć bibliotekę dynamiczną za pomocą polecenia:

```
$ gcc dod.o odej.o -shared -o lib_arytmdyn.so
```

Parametr `-shared` powoduje tworzenie biblioteki wspólnej. Nazwa biblioteki powinna zaczynać się od liter `lib`, a rozszerzenie nazwy powinno mieć postać `.so`, np. `lib_arytmdyn.so`

W celu przetestowania utworzonej biblioteki można wykorzystać poniższy program:

```
/* Plik test_bibldyn.c - testowanie bibliotek dynamicznych w Linuksie */
#include <stdio.h>
int main()
{
   int suma, sumakw;
   suma = dodaj(5, 12);
   sumakw = dodaj_kwadraty(3, 5);
   printf("\nSuma = %d Suma kwadratow = %d\n", suma, sumakw);
   return 0;
}
```

Powyższy program trzeba poddać kompilacji za pomocą polecenia

```
$ gcc -c test_bibldyn.c -o test_bibldyn.o
```

a następnie wykonać linkowanie

```
$ gcc test_bibldyn.o -L. -l_arytmdyn -o test_bibldyn.out
```

Opcja `-L` wskazuje katalog, w którym linker ma szukać bibliotek statycznych i wspólnych (dynamicznych). Jeśli potrzebna biblioteka dostępna jest w postaci biblioteki statyczna i dynamicznej, to linker wybiera bibliotekę dynamiczną (wspólną). Można wymusić używanie biblioteki statycznej za pomocą parametru `-static`. Opcja `-l` podaje bibliotekę, która ma być używana. Nazwa biblioteki nie zawiera rozszerzenia `.so` ani też początkowych znaków `lib`.

Istnieje możliwość sprawdzenia jakie biblioteki są używane w programie wynikowym i czy są dostępne:

```
$ ldd test_bibldyn.out
```

Jeśli pojawi się odpowiedź *not found* to znaczy, że biblioteka wspólna jest niedostępna. Innymi słowy: dynamiczny program ładujący (ang. dynamic loader) nie może znaleźć biblioteki.

Poprawne działanie wszystkich omówionych elementów wymaga jeszcze podania położenia biblioteki. Najprościej można to wykonać na pomocą polecenia

```
$ export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:."
```

Polecenie to powoduje dodanie nazwy katalogu (tu: katalogu bieżącego reprezentowanego przez znak `.`) do istniejących ścieżek przeszukiwania (jeśli takie były zdefiniowane). W końcu, w celu uruchomienia programu trzeba podać polecenie

```
$ ./test_bibldyn.out
```

Ćwiczenie 3

WIELOPROCESOWOŚĆ W SYSTEMIE UNIX

3.1 Wstęp

Wieloprocesowość (ang. multiprocessing) jest obecnie, obok wielowątkowości, jedną z najbardziej istotnych właściwości wielozadaniowych systemów operacyjnych. Pozwala na współbieżne przetwarzanie zadań użytkownika umożliwiając lepsze wykorzystanie zasobów. Ponadto konstrukcja oprogramowania może opierać się o wiele współbieżnych procesów, umożliwiając tworzenie aplikacji równoległych a nawet rozproszonych.

Proces jest to obiekt w systemie operacyjnym, który zawiera segment kodu, danych, stos, zasoby (np. otwarte pliki) itd. Segment kodu może być współdzielony z innymi procesami, podczas gdy segment danych jest prywatny i dostępny jedynie dla jednego procesu. We współczesnych systemach operacyjnych dostępne są także inne mechanizmy współbieżne: wątki. Od procesów różnią się przede wszystkim tym, że pozwalają na współdzielenie segmentu danych.

3.2 Wieloprocesowość dostępna z powłoki (shell)

Powłoka (shell) w systemie UNIX jest częścią systemu operacyjnego, która jest odpowiedzialna za wykonywanie poleceń użytkownika oraz skryptów. Możemy ją kojarzyć z interpreterem poleceń w systemach MsDOS i MsWindows. Wykonywane przez powłokę polecenia użytkownika mogą być zewnętrzne: będą to wówczas programy oraz wewnętrzne: interpretowane i wykonane przez program powłoki. Skrypty są plikami tekstowymi, których kolejne linie są wykonywane tak jak polecenia użytkownika.

W systemie UNIX dostępnych jest wiele programów powłoki. Najbardziej popularnym jest `bash`. Aby przekonać się, jaki program powłoki został przydzielony określonemu użytkownikowi, należy wykonać polecenie `finger`. Przykład działania polecenia `finger` przedstawia poniższy listing:

```
Login: root                             Name: System administrator
Directory: /root                        Shell: /bin/bash
On since Mon Mar 19 10:00 (/etc/localtime) on :0 (messages off)
No mail.
No Plan.
```

Bash jest bardzo rozbudowanym programem powłoki. Aby zapoznać się z jego krótkim opisem proponujemy posłużyć się programem pomocy, uruchamiając: `man bash`. Rozpoczęcie wykonywania każdego polecenia zewnętrznego lub skryptu powoduje utworzenie i uruchomienie odrębnego procesu, w tym czasie proces wywołujący - powłoka - oczekuje na zakończenie procesu potomnego. W poniższym przykładzie konsola użytkownika, obsługiwana przez program powłoki, zostanie zablokowana na 3 sekundy:

```
[~]$ sleep 3
```

O tym, że nowe polecenie jest wykonywane przez odrębny proces, możemy się przekonać wykorzystując polecenie `ps` służące do wyświetlenia informacji o wykonywanych procesach. Użyte z parametrem `-u` lub `u` (zależnie od wersji polecenia) i nazwą użytkownika, wyświetli wszystkie procesy należące do podanego użytkownika. Powyższy przykład możemy prześledzić wykonując podane niżej polecenie:

```
[~]$ ps -u student | grep pts/1
 1427 pts/1 00:00:00 bash
 1464 pts/1 00:00:00 sleep
```

Przyjęliśmy założenie, że polecenie `sleep 3` wykonuje się na konsoli wirtualnej `pts/1`. Natomiast polecenie `grep` służy do filtrowania wyjścia polecenia `ps` tak, aby widoczne były tylko linie zawierające słowo `pts/1`. Jak zauważyliśmy wykonywane były dwa procesy: `bash` i `sleep`. `bash` jako program powłoki oczekuje na zakończenie polecenia `sleep` jako procesu potomnego.

Nie zawsze program powłoki musi oczekiwać na zakończenie procesu potomnego. Umożliwia to mechanizm uruchamiania poleceń "w tle". Aby uruchomić polecenie w tle, należy linię polecenia zakończyć znakiem `&`. Na przykład:

```
[~]$ sleep 3 &
[1] 1545[~]$
[1]+ Done            sleep 3
[~]$
```

Podana w nawiasach kwadratowych liczba oznacza numer zadania, które jest wykonywane w tle. Wyświetlane są dwa takie komunikaty: pierwszy informujący o rozpoczęciu wykonywania zadania w tle:

```
[1] 1545
```

drugi o jego zakończeniu:

```
[1]+ Done            sleep 3
```

Zadania wykonywane w tle pracują współbieżnie względem programu powłoki. Dzięki temu możemy uruchamiać czasochłonne zadania nie tracąc jednocześnie dostępu do powłoki. Ponadto w środowisku graficznym możemy uruchamiać nowe programy, które będą wyświetlane we własnych oknach.

W trakcie wykonywania się polecenia w tle, możemy wyświetlić ich listę, tzw. listę zadań. Służy do tego polecenie `jobs`:

```
[~]$ sleep 100 &
[1] 1546
[~]$ sleep 10 &
[2] 1547
[~]$ jobs
[1]-  Running               sleep 100 &
[2]+  Running               sleep 10 &
```

Jako wynik polecenia `jobs` otrzymujemy listę zadań. Każde z zadań posiada swój numer podany w nawiasach kwadratowych. Dokładnie jedno z nich oznaczone jest znakiem plus. W dowolnym momencie wykonywania zadań w tle, możemy spowodować, że dowolne zadanie zostanie przeniesione jako zadanie bieżące i program powłoki będzie oczekiwał na jego zakończenie. Wykonujemy to za pomocą polecenia `fg <numer_zadania>`. Jeśli numer zadania nie zostanie określony, wybrane zostanie zadanie ostatnie, które jest zaznaczone znakiem plus w wyniku działania polecenia `jobs`.

Analogicznie w dowolnym momencie wykonywania zadania bieżącego, można przenieść je w tło. Wykonujemy to za pomocą wciśnięcia kombinacji klawiszy `^Z` (`CTRL` + `Z`). Zadanie zostanie najpierw zatrzymane:

```
[~]$ sleep 100
[1]+ Stopped             sleep 100
```

a następnie za pomocą polecenia `bg` wznowione jako proces w tle. Na wstępie ćwiczenia użyliśmy konstrukcji `ps -u adam | grep pts/1` aby połączyć działanie dwóch poleceń: `ps` i `grep`. Konstrukcja nazywa się potokiem poleceń i polega na tym, że program powłoki uruchamia dwa procesy: `ps` i `grep`, a następnie standardowe wyjście procesu `ps` łączy ze standardowym wejściem procesu `grep`. Istotne jest to, że oba procesy wykonują się współbieżnie.

Poza łączeniem poleceń w potok, możliwe jest przekierowywanie wejścia i wyjść poleceń z pliku lub do pliku. Na przykład polecenie przeszukiwania systemu plików:

```
[~]$ find / -name *mp3
```

poza wyświetleniem prawidłowych danych do standardowego wyjścia, powoduje wyświetlenie wielu informacji skierowanych do standardowego wyjścia błędu:

```
$ /mnt/multimedia/temp/pw.mp3
find: /mnt/cdrom: Blad wejscia/wyjscia
find: /mnt/cdrom2: Blad wejscia/wyjscia
find: /mnt/floppy: Blad wejscia/wyjscia
```

Stosując przekierowanie dla wyjścia błędu, możemy uniknąć takiej sytuacji:

```
[~]$ find / -name *mp3 2> /dev/null/mnt/multimedia/temp/pw.mp3
```

Poza omówionymi poleceniami, służącymi do zarządzania wykonywaniem wielu procesów uruchamianych z powłoki, warto jeszcze zwrócić uwagę na polecenie `top`. Służy ono do wyświetlenia bieżąco wykonywanych procesów z uwzględnieniem otrzymywanego czasu procesora i zajętością pamięci. Polecenie `top` jest interaktywne: wykonuje się cały czas, aż do otrzymania polecenia zakończenia, odświeżając systematycznie stan systemu. Przykład działania polecenia top podajemy poniżej:

```
CPU states: 2.9% user, 1.6% system, 0.0% nice, 95.4% idle
Mem:   196196K av,  187396K used,     8800K free, 0K shrd, 38276K buff
Swap:  248996K av,       0K used,   248996K free            84920K cached

PID  USER     PRI   NI  SIZE   RSS SHARE STAT %CPU %MEM   TIME COMMAND
1567 adam      11    0   884   884   680 R     4.6  0.4   0:00 top
   1 root       0    0   484   484   420 S     0.0  0.2   0:05 init
   2 root       0    0     0     0     0 SW    0.0  0.0   0:00 kflushd
   3 root       0    0     0     0     0 SW    0.0  0.0   0:00 kupdate
   4 root       0    0     0     0     0 SW    0.0  0.0   0:00 kswapd
   5 root     -20  -20     0     0     0 SW<   0.0  0.0   0:00 mdrecoveryd
 315 root       0    0   592   592   496 S     0.0  0.3   0:00 automount
 337 root       0    0   624   624   520 S     0.0  0.3   0:00 automount
 377 root       0    0   572   572   456 S     0.0  0.2   0:00 syslogd
 387 root       0    0   800   800   388 S     0.0  0.4   0:00 klogd
 400 nobody     0    0   620   620   512 S     0.0  0.3   0:00 identd
 403 nobody     0    0   620   620   512 S     0.0  0.3   0:00 identd
 404 nobody     0    0   620   620   512 S     0.0  0.3   0:00 identd
 405 nobody     0    0   620   620   512 S     0.0  0.3   0:00 identd
 406 nobody     0    0   620   620   512 S     0.0  0.3   0:00 identd
 417 daemon     0    0   492   492   416 S     0.0  0.2   0:00 atd
 430 root       0    0   632   632   520 S     0.0  0.3   0:00 crond
 446 root       0    0   548   548   460 S     0.0  0.2   0:00 lpd
 509 xfs        0    0  3952  3952  1012 S     0.0  2.0   0:07 xfs
```

3.3 Wieloprocesowość w programach użytkowych

Każdy program użytkowy jest reprezentowany przez oddzielny proces. Proces w systemie UNIX może wykonywać się współbieżnie względem innych procesów. Jednak programy użytkowe także mogą składać się z wielu współbieżnych procesów. Procesy wykonywane w ramach jednego programu posiadają tę samą przestrzeń kodu, jednak różne przestrzenie danych. Oznacza to, że do wzajemnej komunikacji potrzebują specjalnych mechanizmów systemowych.

Stworzenie nowego procesu wykonywane jest przez funkcję `pid_t fork(void);` Funkcja ta powoduje, że zostaje utworzona dokładna kopia bieżącego procesu i uruchomiona od następnej instrukcji występującej po funkcji `fork()`. Dlatego, aby rozróżnić, który proces jest procesem pierwotnym a który potomnym, należy zbadać wynik funkcji `fork()`. W procesie macierzystym zwróci ona wartość będącą identyfikatorem stworzonego procesu potomnego, tymczasem proces potomny rozpocznie wykonanie tak, jakby funkcja `fork()` zwróciła wartość zero. Poniższy fragment kodu demonstruje działanie funkcji `fork()`.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
   int pid;
   printf("Proces uruchomiony %d\n", getpid());
   pid = fork();

   if(pid == -1)
```

```
   {
      perror("Blad utworzenia procesu potomnego!");
   return 1;
   }

   if(pid == 0)
   {
      printf("Zglasza sie proces potomny %d\n", getpid());
      sleep(10);
   }
   else
   {
      printf("Zlasza sie proces macierzysty %d. Potomek %d\n",
      getpid(), pid);
      sleep(9);
   }
}
```

Funkcje `fork()` i `getpid()` posiadają swoje prototypy w pliku nagłówkowym `unistd.h`. Funkcja `getpid()` zwraca identyfikator wywoływanego procesu, natomiast podobna funkcja `getppid()` zwraca identyfikator przodka wywoływanego procesu. Zwróćmy uwagę, że w systemie UNIX istnieje bardzo silne powiązanie procesów potomnych. Poniższy przykład demonstruje stworzenie procesu potomnego. Oba procesy wyświetlają identyfikatory własnych i procesów macierzystych.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
   /* utworz proces potomny */
   if(fork() == 0)
   {
      printf("proces potomny pid: %d ppid: %d\n",
      getpid(), getppid());
      sleep(10);
      printf("proces potomny konczy dzialanie\n");
      return 0;
   }
   else
   {
      printf("proces macierzysty pid: %d ppid: %d\n",
      getpid(), getppid());
      sleep(5);
      printf("proces macierzysty konczy dzialanie\n");
      return 0;
   }
}
```

Silne powiązanie procesów potomnych oznacza, że każdy proces (poza wyróżnionymi procesami startowymi) posiada swój proces macierzysty. Możemy się o tym przekonać wykonując polecenie powłoki `ps -u <uzytkownik> --forest`. Przykładowy wynik działania tego polecenia został przedstawiony poniżej. Zakładamy, że został uruchomiony powyższy program pod nazwą `parent_demo`.

```
747 ?        00:00:01 kdeinit
748 pts/1    00:00:00  \_ bash
877 pts/1    00:00:01      \_ kedit
895 pts/1    00:00:00      \_ parent_demo
896 pts/1    00:00:00      |   \_ parent_demo
897 pts/1    00:00:00      \_ ps
```

W systemie UNIX oczywiście możliwe jest stworzenie procesu potomnego, który realizuje kod nowego programu. Realizowane jest to w taki sposób, że po stworzeniu procesu potomnego, wykonuje on jedną z funkcji `exec()`, które powodują załadowanie kodu nowego procesu i rozpoczęcie jego wykonywania. Oto lista funkcji rodziny `exec()`:

```
int execl(const char *path, const char *arg, ...);
```

```
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg , ..., char* const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Funkcje te umożliwiają przekazanie parametrów dla nowego programu w postaci kolejnych argumentów (funkcje z literą `l`) lub w postaci tablicy (funkcje z literą `v`). Przyjmuje się, że za ostatnim argumentem będzie umieszczona wartość `NULL`. Funkcje, które zawierają w nazwie literę `p` szukają podanego programu na ścieżce systemowej. Ponadto funkcja `execle()` umożliwia przekazanie środowiska dla nowego programu. Poniższy przykład demonstruje działanie funkcji `execlp()`. Jako proces potomny zostaje uruchomione polecenie ps. Warto zinterpretować wynik działania programu.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
   /* utworz proces potomny */
   if(fork() == 0)
   {
      printf("proces potomny pid: %d ppid: %d\n", getpid(), getppid());
      execlp("ps", "ps", "-u", "student", "--forest", NULL);
      return 0;
   }
   else
   {
      printf("proces macierzysty pid: %d ppid: %d\n", getpid(), getppid());
      sleep(5);
      return 0;
   }
}
```

Proces macierzysty może przejść w stan oczekiwania na zakończenie działania dowolnego procesu potomnego. Służy do tego funkcja systemowa

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Poniższy program demonstruje działanie funkcji. Proces potomny czeka ok. 5 sekund, po czym wykonuje polecenie systemowe `ps` z opcją wyświetlania stanu procesów. Informacja wyświetlana przez proces macierzysty pojawia się dopiero po zakończeniu działania procesu potomnego.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
   /* utworz proces potomny */
   int pid = fork();
   if(pid == 0)
   {
      printf("proces potomny pid: %d ppid: %d\n", getpid(), getppid());
      sleep(5);
      execlp("ps", "-u student", "-l", "--forest", NULL);
      return 0;
   }
   else
   {
      waitpid(pid, NULL, 0);
      printf("proces potomny %d zakonczony\n", pid);
      return 0;
   }
}
```

Ćwiczenie 4

WIELOPROCESOWOŚĆ W SYSTEMIE MS WINDOWS

4.1 Wstęp

Istotnym elementem współczesnych systemów operacyjnych jest wielozadaniowość (ang. multitasking). Polega ona na możliwości jednoczesnego przetwarzania wielu zadań. Uruchomione programy, są przykładami zadań, które są dostrzegalne bezpośrednio przez użytkownika. W systemie operacyjnym istnieje jednak więcej zadań, które pracują bez dostępu do ekranu.

Praca współbieżna wielu zadań wymaga opracowanego mechanizmu przełączania tych zadań tak, aby każdy z nich otrzymywał wymaganą ilość czasu procesora. Rozróżniamy dwa rodzaje przełączania zadań: z wywłaszczaniem (ang. preemptive multitasking) i bez wywłaszczania (ang. cooperative multitasking).

Jako zadania będziemy rozróżniać procesy i wątki. Są to wykonywalne obiekty systemu operacyjnego. Procesy wykonywane są w izolowanej przestrzeni adresowej, komunikacja między nimi może odbywać się tylko za pomocą funkcji oferowanych przez system operacyjny. W obrębie procesów wykonywane są wątki, po stworzeniu procesu składa się on z jednego wątku - tzw. wątku głównego. Wątki wykonywane są w jednej przestrzeni adresowej procesu. Istnieje więc zagrożenie wzajemnej ingerencji w dane innego wątku. Do zapewnienia prawidłowego dostępu do współdzielonych danych służą odpowiednie metody synchronizacji.

Przetwarzanie wielowątkowe polega na wydzieleniu różnych funkcji programu i przypisaniu im oddzielnych wątków. Dla przykładu jeden wątek może obsługiwać zdarzenia docierające do programu - np. opcje menu, wciśnięcia klawiszy. Inne wątki współbieżnie mogą wykonywać inne zadania. Efektywnym zastosowaniem wątków jest obsługa funkcji blokujących, z których sterowanie może zostać oddane po nieokreślonym czasie. Często takimi funkcjami są funkcje komunikacyjne - na przykład nawiązywanie połączenia sieciowego.

Aplikacje wielowątkowe mogą uruchamiać wątki w miarę potrzeb, dla pojawiających się na bieżąco czynności. Nie wymaga się więc, aby w aplikacji istniała przez cały czas jej działania stała ilość wątków. Zaleca się jednak, ze względów wydajnościowych, aby ilość pracujących wątków w obrębie jednego procesu nie przekraczała 16. Wątki nie muszą przebywać cały czas w

stanie pracującym. Wątki można wstrzymywać lub usypiać. Wątki mogą także być wstrzymane na funkcjach blokujących.

Projektowanie, kodowanie i testowanie programów wielowątkowych jest zadaniem bardzo trudnym. Wzajemne zależności wykonywalnych współbieżnych elementów nie zawsze są deterministyczne. W dowolnym momencie może być wykonywany dowolny z pracujących wątków, powstawanie niepożądanych interakcji jest więc trudne do wykrycia. Do rozwiązania takich problemów stosuje się systemowe metody synchronizacji wątków.

4.2 Zarządzanie procesami

4.2.1 Pojęcie procesu

Na proces składa się przestrzeń zajmowana przez kod i dane programu, otrzymujące czas procesora i korzystający z zasobów systemowych. W systemach Windows 95 i NT proces jest bezpośrednio

związany z programem, który w systemie znajduje się w postaci pliku dyskowego. Utworzenie procesu jest jednoznaczne z uruchomieniem programu lub modułu. Nie został więc zaimplementowany UNIXowy mechanizm rozwidlenia procesu za pomocą instrukcji `fork()`.

4.2.2 Proces i wątek wykonania, identyfikatory i dojścia

W obrębie procesu wykonywane są wątki. Każdy proces posiada co najmniej jeden wątek. Jeden z nich, istniejący od początku działania procesu, nazywany jest wątkiem głównym. Procesy działają w różnych przestrzeniach adresowych, więc wymiana danych między nimi musi odbywać się z wykorzystaniem oferowanych przez system mechanizmów komunikacyjnych, np. przekazywania komunikatów (ang. message passing). W odróżnieniu od procesów wątki wykonują się w jednej przestrzeni adresowej, co stwarza potencjalne zagrożenie uszkodzenia danych jednego wątku przez inny lub niezachowania spójności w korzystaniu ze wspólnych zmiennych. Systemy Windows oferują bogaty zestaw mechanizmów synchronizacji.

Istnieją dwa sposoby identyfikacji procesów i wątków w systemie: identyfikatory i dojścia (nazywane także uchwytami). Identyfikatory są to liczby całkowite (`int`) przydzielane przez system operacyjny. Mechanizm ten jest podobny do mechanizmu spotykanego w systemie Unix (identyfikatorem jest tzw. pid - process identifier). Dojścia są to struktury (typ `HANDLE`), na których wykonywane są wszystkie podstawowe operacje systemowe. Dojścia (uchwyty) są zwracane przez funkcje tworzące procesy i wątki. Mogą one posłużyć do, na przykład, pobrania kodu zakończenia procesu lub oczekiwania na zakończenie działania programu.

4.2.3 Podstawowe funkcje systemowe

Do podstawowych funkcji pozwalających operować na procesach, należą stworzenie procesu i jego zakończenie.

Stworzenie procesu: `CreateProcess`

Funkcja ta tworzy nowy proces i jego wątek główny. Nowy proces uruchamia określony plik uruchamialny. Deklaracja jej jest następująca:

```
BOOL CreateProcess(
   LPCTSTR lpApplicationName,
   LPTSTR lpCommandLine,
   LPSECURITY_ATTRIBUTES lpProcessAttributes,
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
   BOOL bInheritHandles,
   DWORD dwCreationFlags,
   LPVOID lpEnvironment,
   LPCTSTR lpCurrentDirectory,
   LPSTARTUPINFO lpStartupInfo,
   LPPROCESS_INFORMATION lpProcessInformation
);
```

Parametr `lpApplicationName` określa nazwę pliku, do którego wykonania zostanie powołany nowy proces. Argumentem tego programu będzie ciąg podany w parametrze `lpCommandLine`. Program do jego odczytu może wykorzystać funkcję systemową `GetCommandLine`. Oczywiście pozostają dostępne mechanizmy języka C - parametry funkcji `main`. Parametr `lpCommandLine` może zawierać także nazwę modułu, jeśli `lpApplicationName` przyjmie wartość `NULL`. Kolejne dwa argumenty: `lpProcessAttributes` i `lpThreadAttributes` w systemie Windows 95 są ignorowane. Dotyczą one atrybutów nadawanych procesowi i wątkowi głównemu w momencie startu. Parametr `bInheritHandle` służy do określenia sposobu dziedziczenia dojść (uchwytów) systemowych procesu wołającego procedurę `CreateProcess`. Z kolei `dwCreationFlags` specyfikuje parametry utworzenia procesu, np. stworzenie procesu domyślnie uśpionego (suspended), typ znaków w bloku środowiska (Unicode / ANSI). Parametr ten ponadto określa priorytet nowego procesu. Parametr `lpEnvironment` wskazuje na blok, który zawiera zmienne środowiskowe dla nowego procesu i jeśli ma wartość specjalną `NULL`, oznacza to, że otrzyma on środowisko procesu wywołującego. Parametry `lpCurrentDirectory` oznacza katalog, w którym proces zostanie uruchomiony, a `lpStartupInfo` wskazuje na specjalną strukturę określającą parametry wyświetlenia głównego okna procesu (np. pozycję). Ostatni parametr `plProcessInformation` pozwala uzyskać przy pomocy specjalnej struktury `PROCESS_INFORMATION` identyfikatory i dojścia do procesu i wątku głównego.

Tablica 4.1: Wartości priorytetów procesów

Priorytet	Zastosowanie
`REALTIME_PRIORITY_CLASS`	Procesy posiadające najwyższe wymagania czasowe; może spowodować wyprzedzenie takich istotnych zadań systemowych jak opróżnienie bufora dysku lub myszy
`HIGH_PRIORITY_CLASS`	Proces wykonujący zadania o wysokich wymaganiach czasowych; przykładem jest menedżer zadań, który musi zostać reagować szybko na wezwanie użytkownika
`NORMAL_PRIORITY_CLASS`	Wszystkie procesy nie posiadające specjalnych wymagań czasowych
`IDLE_PRIORITY_CLASS`	Proces, którego wątki są wykonywane tylko w momentach bezczynności systemu; przykładem jest wygaszacz ekranu

Zakończenie procesu: `ExitProcess`, `TerminateProcess`

```
VOID ExitProcess (
   UINT uExitCode // kod powrotu wszystkich watkow
);
```

Funkcja `ExitProcess` jest przeznaczona dla procesu, który zamierza zakończyć działanie. Funkcja ta kończy proces i wszystkie jego wątki. Funkcja jednak nie powoduje zamknięcia procesów potomnych. Jest to typowa cecha systemów Windows - brak silnych powiązań procesów dziedzicznych.

```
BOOL TerminateProcess(
   HANDLE hProcess, // dojscie do procesu
   UINT uExitCode // kod powrotu procesu
);
```

Funkcja `TerminateProcess` powoduje bezwarunkowe zakończenie procesu i wszystkich jego wątków, lecz w odróżnieniu od `ExitProcess` może być wołana przez każdy proces. W systemie Windows NT, aby wykonanie się powiodło, proces będący podmiotem działania musi posiadać prawa `PROCESS_TERMINATE`. Podobnie jak i `ExitProcess` nie powoduje zakończenia procesu potomnego.

Wykorzystanie funkcji `TerminateProcess` musi być decyzją uzasadnioną, powoduje ona bowiem nie czyste zakończenie procesu: używane biblioteki DLL nie zostają zwolnione. Otwarte przez proces dojścia zostają jednak zamknięte, a wszystkie wątki zakończone.

4.2.4 Priorytety

Funkcje: `GetPriorityClass` i `SetPriorityClass`

```
DWORD GetPriorityClass(
   HANDLE hProcess // dojscie do procesu
);

BOOL SetPriorityClass(
   HANDLE hProcess, // dojscie do procesu
   DWORD dwPriorityClass // nowa wartosc priorytetu
);
```

Funkcje te służą do pobrania aktualnej wartości priorytetu i ustawienia nowej. Priorytet procesu jest wartością bazową, która służy do obliczania priorytetu rzeczywistych obiektów wykonywalnych - wątków. Priorytety wątków posiadają wartości, które określają relatywną wartość względem właśnie priorytetu procesu. Wartości priorytetów Wartościami priorytetów jakimi należy się posługiwać w operacjami dotyczącymi priorytetów (w tym także `CreateProcess`) są predefiniowane stałe znajdujące się w plikach nagłówkowych o znaczeniu przedstawionym w tabeli 4.1.

4.2.5 Operacje na procesach

Pobranie aktualnego procesu: `GetCurrentProcess` i `GetCurrentProcessId`

```
HANDLE GetCurrentProcess(VOID);
```

Funkcja `GetCurrentProcess` zwraca tzw. pseudodojście do aktualnego procesu. Pseudodojście posiada właściwość polegającą na tym, że nie ma potrzeby dokonania zamknięcia funkcją `CloseHandle`.

```
DWORD GetCurrentProcessId(VOID);
```

Funkcja `GetCurrentProcessId` zwraca identyfikator procesu. Służy on do identyfikacji procesu w całym systemie i do identyfikacji procesu w komunikacji międzyprocesowej.

Pobranie właściwego dojścia do procesu: `OpenProcess`

```
HANDLE OpenProcess(
   DWORD dwDesiredAccess, // flaga dostepu
   BOOL bInheritHandle, // flaga dziedziczenia
   DWORD dwProcessId // identyfikator procesu
);
```

Procedura ta pozwala na zamianę identyfikatora procesu na dojście. Dojście jest wykorzystywane do wielu operacji na procesach. Identyfikator z kolei jest wartością opisującą proces w obrębie całego systemu (nie tylko procesu) i może zostać pobrana, np. z systemowej listy aktywnych procesów.

Informacje o zakończeniu procesu: `GetExitCodeProcess`

```
BOOL GetExitCodeProcess(
   HANDLE hProcess, // dojscie do procesu
   LPDWORD lpExitCode // wskaznik do zmiennej, w ktorej zostanie zapisany kod
   ↪ powrotu
);
```

Funkcja służy do pobrania statusu zakończenia procesu. Może nią być wartość zwrócona przez standardowe funkcje zakończenia procesu (funkcje `ExitProcess` i `TerminateProcess`), zwróconą przez `return` funkcji `main` lub `WinMain` lub wartość powrotu wątku, który spowodował zakończenie procesu. Informację o kodzie powrotu można pobierać tak długo, jak w systemie pozostają nie zamknięte dojścia. Próba odczytania kodu powrotu procesu, który wciąż jeszcze działa powoduje otrzymanie kodu błędu `STILL_ACTIVE`.

4.3 Wielowątkowość

4.3.1 Definicja wątku

O ile procesy są elementami współbieżnymi systemu operacyjnego, wątki są elementami procesu. Każdy proces musi się składać z co najmniej jednego wątku. Określa on fakt pracy procesu, zakończenie ostatniego wątku oznacza zakończenie procesu, a kod zakończenia procesu odpowiada kodowi zakończenia tego wątku.

4.3.2 Stan wątku

Wątek może przebywać w jednym z trzech stanów: wykonywania, czekania i wstrzymania. Tylko w pierwszym przypadku procesor w danej chwili zajmuje się wykonywaniem kodu wątku. W stanie czekania, wątek znajduje się w kolejce wątków czekających na otrzymanie kwantu czasu procesora. Wątek może także znaleźć się w stanie wstrzymania przez rozmyślne działanie programisty, nigdy nie otrzymując czasu procesora.

4.3.3 Podstawowe funkcje systemowe

Stworzenie wątku: `CreateThread`, funkcja wątku

Funkcja `CreateThread` tworzy nowy wątek wykonywany w tej samej przestrzeni adresowej, co proces wołający. Istnieje także funkcja `CreateRemoteThread`, która tworzy wątek należący do innego procesu. Deklaracja funkcji wygląda następująco:

```
HANDLE CreateThread(
   LPSECURITY_ATTRIBUTES lpThreadAttributes,
   DWORD dwStackSize,
   LPTHREAD_START_ROUTINE lpStartAddress,
   LPVOID lpParameter,
   DWORD dwCreationFlags,
   LPDWORD lpThreadId
);
```

Pierwszy z parametrów - `lpThreadAttributes` określa atrybuty bezpieczeństwa wątku, lecz działa jedynie w przypadku systemu Windows NT. Kolejny `dwStackSize` pozwala ustawić wielkość stosu nowego wątku w bajtach. Jeśli zostanie podana wartość specjalna zero, to system przydzieli domyślną wielkość stosu. Parametr `lpStartAddress` pozwala określić punkt wejścia wątku, będący zwykle adresem (wskaźnikiem) do funkcji, od której wykonania rozpocznie pracę nowy wątek. Prototyp tej funkcji powinien być w postaci:

```
DWORD WINAPI ThreadFunc(LPVOID);
```

Procedura wejścia wątku zawiera parametr, który pozwala zidentyfikować konkretny wątek w sytuacji, gdy kilku wątkom zostanie przypisana jedna procedura. Parametr, który otrzyma jest podany w funkcji `CreateThread` jako `lpParameter`. Flagi tworzenia wątku dopuszczają w aktualnych wersjach systemu tylko jedną możliwość - stworzenie wątku w trybie uśpienia: `CREATE_SUSPENDED`.

Uzyskanie identyfikatora wątku, czyli liczby 32-bitowej jednoznacznie identyfikującej wątek w systemie jest wpisywany przez funkcję do zmiennej wskazywanej przez `lpThreadId`. Dojście do nowego wątku jest zaś zwracany przez wartość funkcji i jego poprawność (czyli wartość różna od `INVALID_HANDLE_VALUE`) oznacza pomyślne wykonanie funkcji.

Zakończenie wątku

Zakończenie wątku może zostać wykonywane poprzez opuszczenie funkcji wątku (podanej jako parametr funkcji `CreateThread` lub `CreateRemoteThread`), które w składni języka C ma postać `return n`. Funkcja `ExitThread` pozwala na zakończenie wątku w dowolnym miejscu jego wykonywania. Deklaracja jej wygląda następująco:

```
VOID ExitThread(
   DWORD dwExitCode // kod powrotu
);
```

Jest to preferowany sposób zakończenia wątku, pozwala systemowi zwolnić wszystkie zasoby wykorzystywane przez niego. Jeśli funkcja wywołana jest przez ostatni wątek procesu, powoduje jego zakończenie, a kod powrotu wątku jest zarazem kodem powrotu procesu.

Zakończenie wątku za pomocą funkcji `TerminateThread` może być zainicjowane przez inny wątek. Pozwala ona przekazać kod powrotu kończonego wątku:

```
BOOL TerminateThread(
   HANDLE hThread, // dojscie do watku
   DWORD dwExitCode // kod powrotu
);
```

Wykonanie tej procedury w systemie Windows NT zakończy się sukcesem, jeśli dojście do wątku procesu wołającego posiada prawo `THREAD_TERMINATE`. Jeśli funkcja dotyczy ostatniego wątku procesu, powoduje jego zakończenie, a kod powrotu wątku jest zarazem kodem powrotu procesu.

Wstrzymanie wątku: `SuspendThread`, `Sleep`

Wstrzymanie wątku (suspend) wykonuje się za pomocą funkcji `SuspendThread`:

```
DWORD SuspendThread(
   HANDLE hThread // dojscie do watku
);
```

Wartością zwracaną przez funkcję jest wartość licznika sprzed momentu wywołania funkcji. Jeśli funkcja nie powiodła się, zwraca `0xFFFFFFFF`.

Uśpienie procesu (sleep) na żądany czas można wykonać wywołując funkcję `Sleep`:

```
VOID Sleep(
   DWORD dwMilliseconds // czas przerwy w milisekundach
);
```

Wznowienie wątku: `ResumeThread`

```
DWORD ResumeThread(
   HANDLE hThread
);
```

Tablica 4.2: Wartości priorytetów wątków

Wartość	Znaczenie
THREAD_PRIORITY_ABOVE_NORMAL	jeden punkt powyżej priorytetu procesu
THREAD_PRIORITY_BELOW_NORMAL	jeden punkt poniżej priorytetu procesu
THREAD_PRIORITY_HIGHEST	dwa punkty powyżej priorytetu procesu
THREAD_PRIORITY_IDLE	w zależności od priorytetu procesu: czasu rzeczywistego - poziom o wartości 16; pozostałe - poziom o wartości 1.
THREAD_PRIORITY_LOWEST	dwa punkty poniżej priorytetu procesu
THREAD_PRIORITY_NORMAL	priorytet procesu
THREAD_PRIORITY_TIME_CRITICAL	w zależności od priorytetu procesu: czasu rzeczywistego - poziom o wartości 31; pozostałe - poziom o wartości 15.

Powoduje zmniejszenie licznika, który osiągając wartość niedodatnią umożliwia wznowienie wątku (p. `SuspendThread`). Wartością zwracaną przez funkcję jest wartość licznika sprzed momentu wywołania funkcji. Jeśli funkcja nie powiodła się, zwraca `0xFFFFFFFF`.

4.3.4 Priorytety

Funkcje: `Get` / `SetThreadPriority` Funkcja pozwalająca pobrać priorytet wątku:

```
int GetThreadPriority(
   HANDLE hThread // dojscie do watku
);
```

Funkcja służąca do zmiany priorytetu wątku:

```
BOOL SetThreadPriority(
   HANDLE hThread, // dojscie do watku
   int nPriority // wartosc nowego priorytetu
);
```

Priorytety wątków mają realizację dwupoziomową. Za rzeczywistą wielkość kwantu czasu przeznaczonego dla wątku odpowiada zarazem priorytet procesu jak i wątku. Stałe używane do określenia priorytetów wątków wraz z konfrontacją z priorytetem procesów przedstawione są w tabeli 4.2. Domyślną wartością priorytetu otrzymywaną przez wątek jest normalny.

4.3.5 Dodatkowe operacje na wątkach

Pobranie aktualnego wątku: `GetCurrentThread`, `GetCurrentThreadId`

```
HANDLE GetCurrentThread(VOID);
```

Pozwala uzyskać tzw. pseudodojście do wątku.

```
DWORD GetCurrentThreadId(VOID);
```

Funkcja ta pozwala uzyskać identyfikator wątku - liczbę 32-bitową jednoznacznie identyfikującą wątek w systemie i służącą do przekazywania informacji o wątku pomiędzy procesami.

Kod zakończenia wątku: `GetExitCodeThread`

```
BOOL GetExitCodeThread(
   HANDLE hThread, // dojscie do watku
   LPDWORD lpExitCode // wskaznik do zmiennej do zapisu kodu powrotu
);
```

Funkcja służy do pobrania kodu powrotu wątku. Otrzymaną wartością jest ta, która zostanie podana w funkcji `ExitThread`, `TerminateThread` lub argument return głównej funkcji wątku. Jeśli został zakończony proces, wartością jest kod powrotu procesu.

4.4 Przykład aplikacji wielowątkowej

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
//---------------------------------------------------------------------------
#pragma argsused
struct dane_dla_watku // tablica zawiera dane, ktore otrzymaja watki
{
   char nazwa[50];
   int parametr;
} dane[3] = { { "[1]", 5 }, { "[2]", 8 }, { "[3]", 12 } };

// priorytety watkow
int priorytety[3] = { THREAD_PRIORITY_BELOW_NORMAL,
   THREAD_PRIORITY_NORMAL, THREAD_PRIORITY_ABOVE_NORMAL
};

HANDLE watki[3]; // dojscia (uchwyty) watkow

// deklaracja funkcji watku
DWORD WINAPI funkcja_watku(void *argumenty);
//---------------------------------------------------------------------------
int main(int argc, char **argv)
{
   int i;
   DWORD id; // identyfikator watku
   clrscr();
   printf("Uruchomienie programu\n");

   // tworzenie watkow
   for(i = 0; i < 3; i++)
   {
      watki[i] = CreateThread(
         NULL, // atrybuty bezpieczenstwa
         0, // inicjalna wielkosc stosu
         funkcja_watku, // funkcja watku
         (void *)&dane[i],// dane dla funkcji watku
         0, // flagi utworzenia
         &id);

      if(watki[i] != INVALID_HANDLE_VALUE)
      {
         printf("Utworzylem watek %s o id %x\n", dane[i].nazwa, id);
         // ustawienie priorytetu
         SetThreadPriority(watki[i], priorytety[i]);
      }
   }

   Sleep(20000); // uspienie watku glownego na 20 s
   return 0;
}
//---------------------------------------------------------------------------
//trzy takie funkcje pracuja wspolbieznie w programie
DWORD WINAPI funkcja_watku(void *argumenty)
{
   unsigned int licznik = 0;

   // rzutowanie struktury na wlasny wskaznik
   struct dane_dla_watku *moje_dane = (struct dane_dla_watku *)argumenty;

   // wyswietlenie informacji o uruchomieniu
   gotoxy(1, moje_dane->parametr);
   printf("%s", moje_dane->nazwa);
   Sleep(1000);
```

```
   // praca, watki sa terminowane przez zakonczenie programu
   // -funkcji main
   while(1)
   {
      gotoxy(licznik++ / 5000 + 5, moje_dane->parametr);
      printf(".");
   }
   return 0;
}
```

Ćwiczenie 5

KOMUNIKACJA MIĘDZYPROCESOWA W SYSTEMIE MS WINDOWS

5.1 Wstęp

Przedmiotem ćwiczenia są techniki komunikacji międzyprocesowej stosowane w środowisku systemu Windows: przesyłanie komunikatów, schowek (ang. clipboard) oraz Dynamic Data Exchange.

5.2 Komunikaty

5.2.1 Procedura obsługi okna

Architektura środowiska systemu Windows jest zorientowana na sterowanie komunikatami, tzn. Windows oddziaływuje na programy poprzez komunikaty. Dokładniej: sformułowanie "Windows wysyła komunikat do programuóznacza, że Windows woła funkcję zdefiniowaną w programie, określaną jako *procedurę obsługi okna* (ang. window procedure). Parametry przekazywane do tej funkcji opisują konkretny komunikat.

Każde okno, jakie tworzy program, ma przypisaną funkcję obsługi okna. Ta funkcja może być zdefiniowana albo bezpośrednio w kodzie programu albo bibliotece DLL. Okna tego samego typu (ang. window class) mogą używać tej samej *procedury obsługi okna* [1]. Po wywołaniu tejże funkcji, okno programu wykonuje zdefiniowaną przez użytkownika - dla danego typu komunikatu - akcję.

Kiedy program rozpoczyna się wykonywać, wówczas Windows tworzy dla niego kolejkę na komunikaty (ang. message queue). Kolejka przechowuje komunikaty przeznaczone dla wszystkich okien jakie program może wykreować. Program zawiera (powinien!) krótki fragment kodu - pętlę komunikatów (ang. message loop), której zadaniem jest pobieranie komunikatów z kolejki i przekazywanie ich do odpowiednich procedur obsługi okien. Część komunikatów przesyłana jest bezpośrednio (z pominięciem kolejki) do *procedury obsługi okna.*

5.2.2 Pętla komunikatów

Program pobiera kolejne komunikaty z kolejki utworzonej dla niego przez Windows poprzez wykonywanie pętli, której kod źródłowy może mieć postać następującą:

```
MSG msg;
while(GetMessage(&msg, (HWND) NULL, 0 , 0))
{
   TranslateMessage(&msg);
   DispatchMessage(&msg);
}
return msg.wParam;
```

[1] np. w bibliotece DLL o zawartej w pliku USER.EXE znajduje się funkcja, która przekazuje komunikaty do wszystkich przycisków (ang. buttons) znajdujących się we wszystkich programach pracujących pod kontrolą systemu Windows.

Funkcja `GetMessage` wykonywana na początku pętli pobiera z kolejki jeden komunikat i umieszcza go w zmiennej `msg`. Pozostałe parametry funkcji spełniają rolę "filtrującą"komunikaty. Drugi parametr `GetMessage` określa dla jakiego okna danej aplikacji ma być przeznaczony pobrany komunikat. Jeśli wartością drugiego parametru jest `NULL` to oznacza, że pobrany ma być komunikat przeznaczony dla dowolnego okna aplikacji. Trzeci i czwarty parametr funkcji określają, że ma być pobrany jedynie komunikat o identyfikatorze z podanego zakresu. Jeśli oba parametry wynoszą 0 to oznacza, że identyfikator komunikatu nie jest istotny.

W przypadku, kiedy pobranym komunikatem jest `WM_QUIT`, wówczas funkcja `GetMessage` zwraca wartość 0. To powoduje, że sterowanie w zamieszczonym powyżej fragmencie kodu zostaje przekazane do pierwszej instrukcji za pętlą. Komunikat `WM_QUIT` oznacza żądanie zakończenia działania przez aplikację - zostaje on wygenerowany, gdy aplikacja wywoła funkcję `PostQuitMessage`.

Funkcja `GetMessage` zwraca wartość różną od 0, gdy pobranym komunikatem nie jest komunikat `WM_QUIT`. Gdy w kolejce nie ma komunikatów, wówczas `GetMessage` przekazuje sterowanie do innej aplikacji, aż do chwili, kiedy jakiś komunikat stanie się dostępny.

Funkcja `TranslateMessage` przekazuje komunikat z powrotem do Windows w celu przetworzenia go w zakresie operacji klawiatury. Jeśli komunikat sygnalizuje wciśnięcia klawisza klawiatury (`WM_KEYDOWN`, `WM_SYSKEYDOWN`[2]), wówczas funkcja umieszcza w kolejce jeden z czterech następujących komunikatów: `WM_CHAR`, `WM_SYSCHAR`, `WM_DEADCHAR`, `WM_SYSDEADCHAR`. Kod klawisza jest zawarty w polu `wParam` struktury `MSG` opisanej poniżej. Funkcja `DispatchMessage` wywołuje odpowiednią *procedurę obsługi okna* przekazując jej komunikat. Komunikat `msg` jest typu `MSG` zdefiniowanego w nagłówku `WinUser.h` (include `windows.h`):

```
typedef struct tagMSG {
    HWND        hwnd;
    UINT        message;
    WPARAM      wParam;
    LPARAM      lParam;
    DWORD       time;
    POINT       pt;
} MSG;
```

Typ `POINT` jest zdefiniowany w nagłówku `windef.h` (include `windows.h`):

```
typedef struct tagPOINT
{
    LONG  x;
    LONG  y;
} POINT;
```

Znaczenie poszczególnych pól jest następujące:

- *hwnd* - identyfikator (zwracany przez funkcję `CreateWindow`) okna dla którego komunikat jest przeznaczony;
- *message* - identyfikator typu komunikatu (typy zdefiniowane są jako `WM_...` lub `EM_...`);
- *wParam* - 16-bitowy dodatkowy parametr komunikatu; dokładne znaczenie zależy od typu komunikatu;
- *lParam* - 32-bitowy dodatkowy parametr komunikatu; dokładne znaczenie zależy od typu komunikatu;
- *time* - stempel czasowy; określa moment kiedy komunikat został umieszczony w kolejce;
- *pt* - pozycja kursora myszy w chwili umieszczenia komunikatu w kolejce.

Funkcja `DispatchMessage` wywołując *procedurę obsługi okna* o prototypie:

```
long FAR PASCAL WndProc
(HWND hwnd, WORD message, WORD wParam, LONG lParam)
```

jako parametry przekazuje wartości czterech pierwszych pól struktury komunikatu. W ciele *procedury obsługi okna* programista zwykle umieszcza instrukcję `switch` w celu rozpoznania typu otrzymanego komunikatu, a co za tym idzie - wykonania zaprogramowanej akcji, po której *procedura obsługi okna* powinna zwrócić wartość 0. W przypadku, kiedy odebrany został komunikat, dla którego akcja nie

[2]jest generowany dla klawiszy mających szczególne znaczenie dla Windows - np. kombinacji z `Alt` takich jak `Alt-Tab` przełączające aktywne okno

została zdefiniowana, musi być wywołana funkcja `DefWindowProc`, a wartość przez nią zwrócona musi być zwrócona przez *procedurę obsługi okna*. Czasami w wyniku przetwarzania funkcji `DefWindowProc` zostają wygenerowane inne komunikaty. Przykładowo, jeśli użytkownik wybierze przy pomocy myszy bądź klawiatury opcję `Close` z systemowego menu danej aplikacji, wówczas zostanie wygenerowany komunikat `WM_SYSCOMMAND`. *Procedura obsługi okna* przekaże go do `DefWindowProc`, która to z kolei wyśle do aplikacji komunikat `WM_CLOSE`. Tenże komunikat zostanie także przekazany do `DefWindowProc`, która to funkcja tym razem wywoła funkcję `DestroyWindow`. `DestroyWindow` spowoduje wysłanie przez Windows do aplikacji komunikatu `WM_DESTROY`. Ten komunikat powinien być już obsłużony przez *procedurę obsługi okna*, której przykładowy kod zamieszczony został poniżej:

```
long FAR PASCAL
WndProc(HWND hwnd, WORD message, WORD wParam, LONG lParam)
{
   HDC hdc;
   PAINTSTRUCT ps;
   RECT rect;

   switch(message)
   {
      case WM_PAINT:
         hdc = BeginPaint(hwnd, &ps);
         GetClientRect(hwnd, &rect);
         DrawText(hdc, "Tekst wycentrowany poziomo i pionowo",
            -1, &rect, DT_SINGLELINE | DT_CENTER | DT_VCENTER);
         EndPaint(hwnd, &ps);
         return 0;
      case WM_DESTROY:
         PostQuitMessage(0); /* generuje WM_QUIT */
         return 0;
      default:
         return DefWindowProc(hwnd, message, wParam, lParam);
   }
}
```

5.2.3 Komunikaty kolejkowane i niekolejkowane

Komunikaty mogą być przekazywane do programu na dwa sposoby:

- umieszczenie w kolejce, z której program pobiera je funkcją `GetMessage`;
- wywołanie przez Windows *procedury obsługi okna* programu, czyli bezpośrednie otrzymywanie komunikatu z zewnątrz.

Ostatecznie, niezależnie od rodzaju komunikatu i tak trafiają one w to samo miejsce - do *procedury obsługi okna*. Dzięki temu program jest przejrzysty, ponieważ istnieje jeden centralny punkt, gdzie podejmowana jest decyzja o reakcji programu na dany komunikat.

Komunikaty kolejkowane to:

- oddziaływanie użytkownika poprzez urządzenia wejściowe:
  - klawiaturę (np. `WM_KEYDOWN`, `WM_CHAR`),
  - mysz (np. `WM_MOUSEMOVE`, `WM_RBUTTONDOWN`),
- komunikat timer'a (`WM_TIMER`),
- komunikat o konieczności odrysowania zawartości okna (`WM_PAINT`),
- komunikat o zakończeniu wykonywania programu (`WM_QUIT`).

Pozostałe typy komunikatów są niekolejkowane. Istotną cechą mechanizmu komunikatów w Windows jest to, że obsługa jednego komunikatu nie może być przerwana przez obsługę innego. Jedynie w przypadku kiedy *procedura obsługi okna* woła funkcję, która generuje nowy komunikat, to może być on obsłużony przez *procedurę obsługi okna* przed powrotem z tejże funkcji. Przykładowo, jeśli z ciała *procedury obsługi okna*

wołana jest funkcja `DefWindowProc`, do której przekazuje się otrzymany komunikat, wówczas ta funkcja może wywołać kolejną instancję *procedury obsługi okna* mimo, że poprzednia instancja nie zakończyła się. Należy mieć to na uwadze definiując rozmiar stosu dla programu (standardowo 8kB).

W przypadku kiedy obsługa komunikatu wymaga informacji dostarczonych przez poprzedni, należy tą informację przekazać poprzez zmienne statyczne zdefiniowane w *procedurze obsługi okna*, bądź poprzez zmienne globalne.

Pętla odbioru komunikatów z kolejki nigdy nie wykonuje się współbieżnie z *procedurą obsługi okna*. Powrót z funkcji `DispatchMessage` następuje dopiero wówczas, gdy zakończy się obsługa komunikatu przekazanego przez tą funkcję do *procedury obsługi okna*.

5.2.4 Wysyłanie komunikatów przez program użytkownika

Do wysyłania komunikatów bezpośrednio do *procedury obsługi okna* służy funkcja o prototypie:

```
LRESULT SendMessage(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
```

- `hwnd` identyfikator (zwracany przez funkcje `CreateWindow`) okna dla którego komunikat jest przeznaczony; jeśli jest to `HWND_BROADCAST` wówczas komunikat otrzymają wszystkie okna, także nieaktywne i niewidoczne "bezpańskie"(bez właściciela);
- `message` identyfikator typu komunikatu (typy zdefiniowane jako `WM_...`, lub `EM_...`);
- `wParam` 16-bitowy dodatkowy parametr komunikatu; dokładne znaczenie zależy od typu komunikatu;
- `lParam` 32-bitowy dodatkowy parametr komunikatu; dokładne znaczenie zależy od typu komunikatu.

Funkcja zwraca kod wykonania operacji wysłania komunikatu.

Wywołanie powyższej funkcji powoduje wywołanie przez Windows procedury obsługi okna, związanej z oknem wskazanym przez `hwnd`, i przekazanie jej czterech parametrów tejże funkcji. Dopiero po zakończeniu przez *procedurę obsługi okna* obsługi tego komunikatu, sterowanie zostaje przekazane do następnej instrukcji kodu znajdującej się po wołaniu funkcji `SendMessage`. *Procedura obsługi okna*, do której w ten sposób zostaje wysłany komunikat za pomocą `SendMessage`, może:

- być tą samą *procedurą obsługi okna*, która woła `SendMessage`;
- być inną tego typu procedurą w tym samym programie,
- znajdować się w innej aplikacji.

Przykładowo jeśli okno programu posiada pionowy pasek (ang. scroll bar), to przewijanie o stronami można wykonywać także za pomocą klawiatury w następujący sposób:

```
/* window procedure */
switch(message)
{
   case WM_KEYDOWN: /* Wcisnieto klawisz */
      switch(wParam)
      {
         case VK_PRIOR: /* klawisz PageUp */
            Send Message(hwnd, WM_VSCROLL, SB_PAGEUP, 0L);
            break;
         case VK_NEXT: /* klawisz PageDown */
            Send Message(hwnd, WM_VSCROLL, SB_PAGEDOWN, 0L);
            break;
      }
      return 0;
   case WM_SCROLL: /* Klikniecie na pionowym "scroll bar" lub przesuniecie
   ↪ wskaznika aktualnej pozycji */
      switch(wParam)
      {
         case SB_PAGEUP:
         /* Wykonanie akcji przewiniecia w tyl okna o caly ekran */
         break;
         case SB_PAGEDOWN:
```

```
            /* Wykonanie akcji przewiniecia w przod okna o caly ekran */
            break;
            case . . . .
        }
        return 0;
    case . . . .
}
```

Uwaga! nie wolno wołać bezpośrednio *procedury obsługi okna*! Wynik takiej operacji nie daje się przewidzieć - może ona spowodować nawet upadek systemu.

Funkcją umieszczającą komunikaty w kolejce jest funkcja o prototypie:

```
LRESULT PostMessage(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam);
```

Sterowanie powraca z powyższej funkcji natychmiast po umieszczeniu komunikatu w kolejce. Jeśli funkcja zwróci wartość 0, to oznacza to (najczęściej), że kolejka jest przepełniona i nie można w niej umieścić komunikatu. Dzieje się tak wówczas, gdy przez pewien okres czasu aplikacja wysyłała do kolejki komunikaty, jednocześnie nie umożliwiając ich odbioru. Funkcja PostMessagenie powinna być używana do przesyłania komunikatów sterujących, które powinny być ńatychmiastóbsłużone.

5.3 Schowek

Schowek (ang. clipboard)[3] jest to mechanizm pozwalający na przesyłanie danych z jednego programu do drugiego. Jego podstawową zaletą jest prostota: korzystanie ze schowka umożliwia kilka tylko funkcji znajdujących się w okienkowym module USER.

Wiele programów posiada menu Edit zawierające takie opcje jak Wytnij, Kopiuj i Wklej (ang. Cut, Copy, Paste). Kiedy użytkownik wybierze opcję Wytnij lub Kopiuj, wówczas program przesyła dane z programu do schowka. Dane te są w określonym formacie: jako tekst, jako mapa bitowa lub jako metaplik (ang. metafile). Kiedy użytkownik wybierze opcję Wklej, wówczas program sprawdza, czy schowek zawiera dane w formacie akceptowalnym przez ten program i - w przypadku gdy format jest odpowiedni - przesyła dane ze schowka do programu.

5.3.1 Standardowe formaty danych przechowywanych w schowku

- `CF_TEXT` - łańcuch znaków ASCII zawierający `0x0D`(*carriage return*) i `0x0A`(*line feed*) na końcu każdej linii program, chcąc przekazać dane tego formatu do schowka, umieszcza je w bloku pamięci globalnej, przekazuje dla schowka uchwyt do bloku i od tego momentu blok nie należy już do programu (m.in. nie może zwalniać tego bloku pamięci !);

- `CF_BITMAP` - mapa bitowa zgodna z Windows 2; przekazuje się uchwyt,

- `CF_METAFILE` - grafika w formacie metapliku z dodatkowymi informacjami w strukturze typu `METAFILEPICT`; przekazywany jest uchwyt do bloku pamięci globalnej zzawierającego strukturę `METAFILEPICT`:

  ```
  typedef struct tagMETAFILEPICT {
     int mm; /* tryb odwzorowania obrazu */
     int xExt; /* szerokosc obrazu */
     int yExt; /* wysokosc obrazu */
     HMETAFILE hMF; /* uchwyt do metapliku */
  } METAFILEPICT;
  ```

  `xExt` i `yExt` określają rozmiar prostokąta zawierającego obraz we wszystkich trybach odwzorowania prócz `MM_ISOTROPIC` i `MM_ANISOTROPIC`. Wymiary nie są w bitach, a w jednostkach odpowiednich dla danego trybu. Po przesłaniu do schowka, program nie powinien odwoływać się do pamięci zajmowanej przez strukturę `METAFILEPICT` i przez metaplik;

- `CF_SYLK` - globalny blok danych w formacie Symbolick Link, format używany m.in. przez programy firmy Microsoft takie jak Excel, Multiplan, Chart znaki ASCII, każda linia kończy się znakami `0x0D` i `0x0A`, znak `NULL` na końcu linii nie jest wymagany bo format określa koniec danych;

[3]nie należy utożsamiać z programem CLIPBOARD znajdującym się w grupie Main, który umożliwia jedynie podgląd aktualnej zawartości Clipboard'u (patrz sekcja 5.3.7)

- `CF_DIF` - globalny blok danych w formacie Data Interchange Format, używany przez programy firmy Lotus zawiera znaki ASCII z `0x0D` i `0x0A` na końcu każdej linii, znak `NULL` na końcu linii nie jest wymagany;
- `CF_TIFF` - globalny blok danych w formacie Tag Image File Format; opisuje mapę bitową;
- `CF_OEMTEXT` - podobnie jak `CF_TEXT`, różnica: używa zbioru znaków OEM;
- `CF_DIB` - globalny blok danych rozpoczynający się od struktury `BITMAPINFO` i zawierający mapę bitową:
- `CF_PALETTE` - uchwyt do palety kolorów, używa się w połączeniu z `CF_DIB` w celu określenia palety kolorów użytych przez tą mapę bitową;

Ponadto używane są formaty `CF_PENDATA`, `CF_RIFF`, `CF_WAVE`, które nie będą tu omawiane.

5.3.2 Przesyłanie tekstu do schowka

Niech `pStr` będzie wskaźnikiem na łańcuch znaków tekstu, zaś `wLen` długością łańcucha (nie licząc znaku `NULL`). Kolejne kroki w celu przesłania tekstu do schowka powinny być następujące:

1. umieszczenie tekstu w bloku pamięci globalnej:
   (a) przydzielenie przemieszczalnego[4] bloku pamięci globalnej:
   ```
   hGlMem = GlobalAlloc(GHND, (DWORD) wLen + 1);
   ```
   (b) zablokowanie dostępu do bloku (uniemożliwienie przesunięcia bloku w inne miejsce pamięci oraz jego likwidacj[5]) i pobranie dalekiego wskaźnika do niego:
   ```
   lpGlMem = GlobalLock(hGlMem);
   ```
   (c) kopiowanie łańcucha znaków do bloku pamięci (jeśli pamięć przydzielona została przy użyciu znacznika `GHND`, to została inicjalnie wypełniona zerami, stąd kopiowanie znaku `NULL` nie jest potrzebne):
   ```
   memcpy(lpGlMem, pStr, wLen +1);
   ```
   (d) odblokowanie dostępu do bloku
   ```
   GlobalUnLock(hGlMem);
   ```
2. otwarcie schowka
   ```
   OpenClipboard(hwnd);
   ```
3. wyczyszczenie z dotychczasowej zawartości:
   ```
   EmptyClipboard();
   ```
4. przekazanie uchwytu bloku globalnej (!) pamięci do schowka z równoczesnym wyspecyfikowaniem formatu danych (`CF_...`) przekazywanych w bloku
   ```
   SetClipboardData(CF_TEXT, hGlMem);
   ```
5. zamknięcie schowka:
   ```
   CloseClipboard();
   ```

UWAGI:

- Wszystkie operacje na schowku, począwszy od otwarcia, aż do zamknięcia muszą zostać wykonane podczas obsługi pojedynczego komunikatu. Nie wolno ani kończyć obsługi komunikatu (opuszczać *procedury obsługi okna*) zostawiając otwarty schowek, ani przekazywać sterowania do innego programu np. poprzez wołanie `SendMessage` bądź `PeekMessage`.

[4] `GHND` jest makrem zdefiniowanym w `WinBase.h` (include `Windows.h`) jako `(GMEM_MOVABLE | GMEM_ZEROINIT)`
[5] chyba, że wywołana zostanie funkcja `GlobalReAlloc`

- Nie wolno przekazywać uchwytu do zablokowanego bloku pamięci.
- Po wywołaniu `SetClipboardData` nie wolno odwoływać się do przekazanego bloku pamięci (wyjątkowo można to robić poprzez zwrócony przez tę funkcję uchwyt, przed zamknięciem schowka - jednakże w chwili wywołania `CloseClipboard` blok pamięci musi być odblokowany).

5.3.3 Pobieranie tekstu ze schowka

1. Sprawdzenie, czy schowek zawiera dane w pożądanym formacie (poniższa funkcja zwraca `TRUE` jeśli schowek zawiera dane wyspecyfikowanego formatu):

   ```
   bAvail = IsClipboardFormatAvailable(CF_TEXT);
   ```

2. otwarcie schowka:

   ```
   OpenClipboard(hwnd);
   ```

3. pobranie uchwytu do bloku pamięci globalnej zawierającego tekst; uchwyt ten należy do schowka, stąd można się nim posługiwać jedynie do wywołania funkcji `CloseClipboard`, jednak nie można ani zwolnić bloku, ani zmienić jego zawartości:

   ```
   hCbMem = GetClipboardData(CF_TEXT);
   ```

   Uwaga! jeśli schowek nie zawiera danych wyspecyfikowanego formatu to powyższa funkcja zwróci wartość `NULL` - należy wówczas zamknąć schowek. Powyższej funkcji można używać w charakterze alternatywnego (zamiast p. 1) sposobu sprawdzania, czy schowek zawiera dane odpowiedniego formatu.

4. skopiowanie danych ze schowka; jeden z możliwych sposobów opisano poniżej:

   (a) przydzielenie bloku pamięci globalnej o wielkości identycznej jak przechowywany w schowku:

   ```
   hProgMem = GlobalAlloc(GHND, GlobalSize(hCbMem));
   ```

   (b) (jeśli `hProgMem` nie jest `NULL`) zablokowanie obu obszarów pamięci i pobranie wskaźników do nich:

   ```
   lpCbMem = GlobalLock(hCbMem);
   lpProgMem = GlobalLock(hProgMem);
   ```

   (c) kopiowanie łańcucha znaków:

   ```
   lstrcpy(lpProgMem, lpCbMem);
   ```

   (d) odblokowanie obu obszarów pamięci:

   ```
   GlobalUnlock(hCbMem);
   GlobalUnlock(hProgMem);
   ```

5. zamknięcie schowka:

   ```
   CloseClipboard();
   ```

Od tego momentu można używać uchwytu `hProgMem` do bloku pamięci w którym przechowywana jest kopia (chwilowej) zawartości schowka.

5.3.4 Wszystko w jednym

Jeśli dane muszą być umieszczone w schowku, to po jego otwarciu musi być usunięta jego dotychczasowa zawartość (`EmptyClipboard`). Nie jest możliwe umieszczanie wprost dodatkowych danych tego samego formatu, np. dodatkowej linijki tekstu do tekstu znajdującego się uprzednio w schowku. Jednakże schowek umożliwia przechowywanie danych w różnych formatach (po jednym w każdym), np. można łańcuch tekstu nanieść na mapę bitową oraz zapisać w metapliku, a następnie dokonać przesłania wszystkich trzech bloków pamięci do schowka:

```
   OpenClipboard(hwnd);
   EmptyClipboard();
   SetClipboardData(CF_TEXT, hGlText);
   SetClipboardData(CF_BITMAP, hGlBM);
   SetClipboardData(CF_METAFILEPICT, hGlMFP);
   CloseClipboard();
```

Wywołanie funkcji `EmptyClipboard` usuwa jednocześnie dane wszystkich formatów.

Informację o liczbie formatów danych przechowywanych w schowku w danej chwili można uzyskać wywołując bezparametrową funkcję `CountClipboardFormats`. Natomiast listę wszystkich formatów danych przechowywanych aktualnie w schowku, można uzyskać wywołując cyklicznie funkcję `EnumClipboardFormats`. Funkcja ta jako argument przyjmuje typ formatu (`CF_...`), zwracając także typ - tyle, że kolejny na liście. Jeśli argumentem funkcji jest zero wówczas zwraca ona pierwszy typ formatu z listy. Jeśli funkcja zwróci wartość zero oznacza to, że argument funkcji był ostatnim typem formatu z listy (lub schowek nie został otwarty !). Oto przykładowy fragment kodu umieszczający w tablicy wszystkie typy formatów danych przechowywanych w schowku:

```
int wFrm, i = 0;
WORD tab[TAB_SIZE];
OpenClipboard(hwnd);
wFrm = 0;
do {
   wFrm = EnumClipboardFormats(wFrm);
} while(tab[i++] = wFrm);
CloseClipboard();
```

5.3.5 Opóźnione umieszczanie danych

Często zdarza się, że dane umieszczane w schowku nie są stamtąd pobierane przez żaden program. Jeśli takie dane zajmują duży obszar pamięci, to przydzielanie bloku pamięci w celu wykonania kopii tychże danych jest marnotrawstwem zasobów. Aby uniknąć zaśmiecania pamięci można zastosować technikę zabezpieczającą przed redundancją danych.

Zamiast przekazywać uchwyt do bloku pamięci danych, można funkcję `SetClipboardData` wywołać (dla każdego formatu oddzielnie) z pierwszym argumentem równym `NULL`. Należy jednak zapewnić obsługę trzech następujących komunikatów, które do takiej aplikacji może wysłać Windows:

- `WM_RENDERFORMAT` - komunikat wysyłany w momencie gdy inny program wywoła funkcję `GetClipboardData`, a Windows sprawdził, że dla żądanego formatu (przekazanego jako 16-bitowy parametr) uchwyt ma wartość `NULL`; aplikacja po otrzymaniu komunikatu musi bez otwierania (!) i bez czyszczenia schowka (!) zająć blok pamięci globalnej, umieścić tam dane, a następnie wywołać `SetClipboardData` z wyspecyfikowanym uchwytem do bloku w celu rzeczywistego umieszczenia danych w schowku;
- `WM_DESTROYCLIPBOARD` - komunikat wysyłany w momencie gdy inny program wywoła funkcję EmptyClipboard; aplikacja może już nie przechowywać dodatkowej informacji pomocnej przy przesyłaniu danych do schowka (jeśli taka informacja byłaby w ogóle potrzebna);
- `WM_RENDERALLFORMATS` - komunikat wysyłany wówczas, gdy w schowku nadal znajduje się (dla co najmniej jednego formatu) `NULL` zamiast uchwytu do danych, a aplikacja która tą wartość umieściła kończy swoje działanie; aplikacja powinna wówczas otworzyć schowek, wyczyścić go, zająć na potrzeby każdego potrzebnego formatu po bloku[6] pamięci globalnej, umieścić tam dane, a następnie wywołać `SetClipboardData` dla każdego żądanego formatu oraz zamknąć schowek.

5.3.6 Prywatne formaty danych

Standardowe formaty danych używa się przede wszystkim do wymiany danych pomiędzy różnymi aplikacjami. Ale z tego powodu, że schowek ma umożliwiać wymianę danych także pomiędzy różnymi instancjami tej samej aplikacji, aplikacje mogą umieszczać w Clipboard'zie dane swoich prywatnych formatów (np. oprócz łańcucha tekstu także krój i wielkość czcionki oraz informację formatującą).

Aplikacja może używać standardowego formatu, ale dane mają wówczas znaczenie zrozumiałe tylko dla danej rodziny aplikacji. W wywołaniu funkcji `SetClipboardData` bądź `GetClipboardData` możliwe

[6] na potrzeby niektórych formatów więcej niż jeden

jest użycie stałych CF_DSPTEXT, CF_DSPBITMAP bądź CF_DSPMETAFILEPICT. Program CLIPBOARD może dane tego typu wyświetlać odpowiednio jako tekst, mapę bitową bądź metaplik, ale inna aplikacja nie będzie mogła pobrać danych z Clipboard'u używając stałych CF_TEXT, CF_BITMAP bądź CF_METAFILEPICT. Jednakże aby mieć pewność, że dane w schowku umieściła inna instancja tej samej aplikacji należy wywołać następujące funkcje:

```
hwndClOwn = GetClipboardOwner(); /* kto ostatni umiescil dane w Cb ? */
GetClassName(hwndClOwn, &szClassname, 16); /* char szClassName[16]; */
```

Nazwa klasy aplikacji sprawdzającej jest identyczna z nazwą aplikacji która ostatnio umieściła dane w Clipboard'zie wtedy, gdy są to dwie instancje tej samej aplikacji. Drugi sposób użycia prywatnego formatu opiera się na wykorzystaniu flagi CF_OWNERDISPLAY. Jeśli aplikacja wywoła funkcję SetClipboardData w następujący sposób:

```
SetClipboardData(CF_OWNERDISPLAY, NULL);
```

to wówczas przejmuje kontrolę nad wyświetlaniem zawartości schowka w programach typu ClipboardViewer (patrz sekcja 5.3.7). Ponadto wartość NULL zamiast uchwytu do bloku pamięci oznacza, że program będzie obsługiwać komunikaty opisane w sekcji 5.3.5.

Trzeci sposób użycia prywatnego formatu danych opiera się na rejestrowaniu przez aplikację umieszczającą dane w schowku unikalnej nazwy formatu:

```
wFrmtId = RegisterClipboardFormat(lpszFrmtName);
```

gdzie lpszFRmtName jest wskaźnikiem na łańcuch znaków ASCII zakończony NULL. Wartość zwrócona przez w/w funkcję jest z zakresu od 0xC000 do 0xFFFF, i należy jej używać przy wywołaniu funkcji SetClipboardData oraz GetClipboardData jako identyfikatora formatu danych. Aby program typu *Clipboard Viewer* mógł wyświetlić dane, muszą być one kopiowane także w jednym ze standardowych formatów (dzięki czemu program nie znający prywatnego formatu będzie mógł skopiować dane zawierające prawdopodobnie mniej informacji, np. mapę bitową zamiast grafiki wektorowej). Aby dana aplikacja mogła poznać nazwę zarejestrowanego prywatnego formatu, musi wywołać następującą funkcję:

```
GetClipboardFormatName(wFrmtId, lpsBuffer, nBufSize);
```

która powoduje, że maksymalnie nBufSize początkowych znaków nazwy formatu o identyfikatorze wFrmtId zostanie skopiowanych do bufora wskazywanego przez wskaźnik lpsBuffer.

5.3.7 Program *Clipboard Viewer*

Clipboard viewer (w skrócie: *CbV*) to program który jest powiadamiany o wszelkich zmianach zawartości schowka. Przykładem takiego programu jest program CLIPBOARD znajdujący się w grupie Main. Jednak można stworzyć własny program tego typu - należy jedynie ściśle przestrzegać zasad opisanych w kolejnych punktach. Niestosowanie się do tych zasad może zakłócić działanie także innych programów typu *CbV*.

Łańcuch programów typu Cliboard Viewer

Wszystkie aktualnie wykonujące się programy *CbV* muszą być połączone w jeden łańcuch (jednokierunkową listę). Odpowiedzialność za utrzymywanie tego łańcucha jest scedowana na poszczególne jego elementy - czyli na programy *CbV*. Windows przechowują jedynie identyfikator[7] pierwszego elementu łańcucha nazywanego current clipboard viewer. Wszystkie elementy łańcucha przechowują identyfikator swojego następnika.

Rejestracja nowego Clipboard Viewer

Do żejestracjińowego *CbV* służy funkcja SetClipboardViewer, której jedynym parametrem jest identyfikator rejestrującej się aplikacji. Windows zapamiętują identyfikator nowego *CbV* co jest równoznaczne z umieszczeniem go na początku łańcucha. Do nowego "bieżącego *CbV*"przekazują natomiast identyfikator (wartość zwracana przez funkcję SetClipboardViewer) dotychczasowego "bieżącego *CbV*", który zostaje źepchniętyńa drugą pozycję w łańcuchu. Jeśli rejestrująca się aplikacja *CbV* jest jedyną w łańcuchu, wówczas SetClipboardViewer zwraca wartość NULL:

[7] zwraca go bezparametrowa funkcja GetClipboardViewer

```
static HWND hwndNxtVwr;
/* zapamietany identyfikator nastepnika nie moze byc utracony */
...
switch(message)
{
   case WM_CREATE:
      hwndNxtVwr = SetClipboardViewer(hwnd);
   ...
}
```

Uaktualnianie zawartości okna

Po każdej zmianie zawartości schowka, system Windows wysyła do pierwszego *CbV* w łańcuchu komunikat `WM_DRAWCLIPBOARD`. Po otrzymaniu takiego komunikatu "bieżący *CbV*"powinien przesłać go do aplikacji będącej następną w łańcuchu itd., aż wszystkie aplikacje typu *CbV* zostaną powiadomione. Z kolei poszczególne aplikacje typu *CbV* powinny uaktualnić zawartość swojego okna np. wywołując funkcję `InvalidateRect`, a następnie w obsłudze komunikatu `WM_PAINT` odczytać aktualną zawartość schowka (używając funkcji `OpenClipboard`, `GetClipboard` oraz `CloseClipboard`), po czym umieścić ją w oknie swojej aplikacji:

```
switch(message)
{
...
case WM_CLIPBOARD:
   if(hwndNxtVwr != NULL)
   /* jesli nie jestem ostatni w lancuchu to musze powiadomic nastepny CbV */
   {
      SendMessage(hwndNxtVwr, message, wParam, lParam);
   }

   InvalidateRect(hwnd, NULL, TRUE); /* wygeneruje WM_PAINT */
   return 0;

case WM_PAINT:
   hdc = BeginPaint(hwnd, &ps);
   GetClientRect(hwnd, &Rect);
   OpenClipboard(hwnd);
   hMem = GetClipboardData(...);
   if(hMem != NULL)
   {
      /* uaktualnij okno aplikacji na podstawie zawartosci obszaru
         pamieci wskazywanego przez hMem */
      ...
   }
   CloseClipboard();
   EndPaint(hwnd, &ps);
   return 0;
...
}
```

Usuwanie z łańcucha

Kiedy aplikacja nie chce dalej być elementem łańcucha, musi wywołać funkcję `ChangeClipboardChain`, której pierwszym argumentem jest identyfikator aplikacji wywołującej funkcje, zaś drugim - identyfikator następnego elementu łańcucha. Funkcja ta musi być wywołana jeśli aplikacja kończy działanie będąc jeszcze obecną w łańcuchu:

```
switch(message)
{
case WM_DESTROY:
   ChangeClipboardChain(hwnd, hwndNxtVwr);
   PostQuitMessage(0);
   return 0;
```

```
   ...
}
```

Po wywołaniu funkcji `ChangeClipboardChain` Windows wysyłają do "bieżącego *CbV*"komunikat `WM_CHANGECBCHAIN`. Jako parametr 16-bitowy (`wParam`) przekazują identyfikator aplikacji usuwającej się z łańcucha (kopiują pierwszy argument funkcji `ChangeClipboardChain`). Natomiast mniej znaczące słowo parametru 32-bitowego (`lParam`) zawiera identyfikator aplikacji następnej w łańcuchu (system Windows kopiuje drugi argument funkcji `ChangeClipboardChain`) po aplikacji usuwającej się z niego.

"Bieżący *CbV*oraz następne elementy łańcucha muszą przekazać ten komunikat dalej "wzdłuż"łańcucha, aż do aplikacji będącej poprzednikiem aplikacji usuwającej się. Aplikacja ta zapamiętuje identyfikator swojego nowego następnika, dzięki czemu integralność łańcucha zostaje zachowana:

```
switch(message)
{
   case WM_CHANGECBCHAIN:
   if(wParam == hwndNxtVwr) /* jestem poprzednikiem usuwajacego sie ? */
   {
      hwndNxtVwr = LOWORD(lParam); /* handle nowego nastepnika */
   }
   else
   {
      if(hwndNxtVwr != NULL) /* jesli nie jestem ostatni w lancuchu to */
      { /* musze powiadomic mojego nastepnika */
         SendMessage(hwndNxtVwr, message, wParam, lParam);
      }
   }
   return 0;
...
}
```

Ćwiczenie 6

INTERFEJS GNIAZDEK W SYSTEMIE MS WINDOWS

6.1 Wstęp

Jeśli dwa komputery komunikują się między sobą, to oprogramowanie sieciowe na każdej warstwie jednego komputera komunikuje się z tą samą warstwą innego komputera, np. warstwa transportowa jednego komputera komunikuje się z warstwą transportową innego komputera. Warstwa transportowa nie ma wglądu w to co się dzieje w warstwach niższych. Zatem tworzenie aplikacji użytkownika wymaga tylko znajomości funkcji usługowych udostępnianych przez warstwę niższą i nie wymaga zgłębiania szczegółów związanych z zarządzaniem siecią czy programowaniem karty sieciowej.

Standardowym interfejsem programistycznym, stosowanym w różnych systemach komputerowych do komunikacji w sieci Internet jest specyfikacja gniazdek (ang. socket). Specyfikacja ta jest w pewnym stopniu wzorowana na typowych funkcjach obsługi plików, ale jest od nich bardziej złożona ze względu na specyfikę sieci. Specyfikacja gniazdek tworzy API obejmujące zestaw funkcji, które realizują rozliczne operacje sieciowe, w tym transmisję danych, przy czym umożliwiają współpracę z wieloma protokołami sieciowymi. Warto podkreślić, że same gniazdka stanowią tylko interfejs między programami użytkowymi a oprogramowaniem protokołów, wewnątrz którego implementowane są funkcje zawarte w specyfikacji.

Oprogramowanie sieciowe stosowane w systemie Windows ma strukturę warstwową, która jednak w niewielkim tylko stopniu koresponduje z klasycznym modelem OSI. Struktury oprogramowania sieciowego zostały bowiem tak skonstruowane, by mogły elastycznie dopasowywać się do różnych typów protokołów sieciowych, posługiwać się kartami sieciowymi pochodzącymi od różnych producentów, a jednocześnie udostępniać programom użytkownika pewien jednolity zestaw funkcji sieciowych API, całkowicie niezależny od typu zainstalowanej karty sieciowej, i ewentualnie w niewielkim stopniu uzależniony od rodzaju używanego protokołu. Istotnym czynnikiem jest także zapewnienie możliwości jednoczesnego przesyłania przez sieć (i przez tę samą kartę sieciową) danych pochodzących z różnych aplikacji, posługujących się różnymi protokołami. W bardziej złożonych instalacjach należy także brać pod

uwagę możliwość zainstalowania w komputerze dwóch lub więcej kart sieciowych, a także pracy wieloprocesorowej (w systemie Windows NT).

6.2 Protokoły sieci Internet

W sieci Internet jako standardowe protokoły komunikacyjne wykorzystuje się zbiór (rodzinę) protokołów TCP/IP. Określenie to obejmuje nie tylko IP i TCP, ale także UDP, ARP, FTP, HTTP, SMTP3, POP3 i wiele innych. Protokoły IP i TCP (powstały wcześniej niż model OSI) zaliczane są do protokołów niskiego poziomu - są one odpowiedzialne za transmisję danych.

Zadaniem warstwy sieciowej - protokołu IP jest znalezienie trasy i przesłanie datagramów do komputera przeznaczenia. Zauważmy, że dzięki warstwie łącza danych protokoły warstwy sieciowej nie zależą od technologii wykonania sieci.

Ze względu na konstrukcję IP jest protokołem bezpołączeniowym i nie gwarantuje doręczenia danych. Jednak jeśli datagram dotrze do miejsca przeznaczenia (pakiety IP zawierają informacje o adresach nadawcy i odbiorcy), to zawarta w nim suma kontrolna pozwala sprawdzić czy przesłany datagram nie uległ przekłamaniu.

Dwa protokoły wyższego poziomu: TCP i UDP wykorzystują protokół IP do przesyłania danych. Zadaniem protokołu TCP jest podział wysyłanego komunikatu na datagramy, przesłanie ich, retransmitując te, które zostały zagubione za pomocą protokołu IP, a następnie po stronie odbierającej scalenie ich we właściwej kolejności.

Gdy przychodzi pakiet, który zawiera datagram IP, oprogramowanie obsługi przerwania umieszcza ten pakiet w kolejce i wywołuje `send` aby powiadomić proces IP, że przybył datagram. Z każdym urządzeniem sieciowym związana jest kolejka - jeden proces IP pobiera datagramy ze wszystkich kolejek i przetwarza je.

Za pomocą protokołu TCP tworzy się usługi zorientowane na połączenie. W takim przypadku zostaje utworzony wirtualny obwód dający złudzenie ciągłego połączenia. Aczkolwiek nie jest tworzony obwód rzeczywisty, to dla aplikacji korzystających obwodu wirtualnego, fizyczne nieciągłości kanału komunikacyjnego są niewidoczne.

Z kolei protokół UDP pozwala realizować usługi bezpołączeniowe. Komunikat przesłany przy użyciu takiego protokołu musi zawierać wszystkie informacje adresowe. Warto dodać, że protokoły bezpołączeniowe są wielokrotnie szybsze.

Oprogramowanie UDP składa się ze zwykłych procedur, które wywołuje proces IP. Procedury te obsługują nadchodzące datagramy UDP. Proces IP umieszcza datagram w kolejce, skąd może go pobrać program użytkowy.

Niekiedy TCP i IP wykonują się jako oddzielne procesy. Nie dotyczy to jednak UDP, który jest mniej skomplikowany.

6.3 Koncepcja gniazdek

Specyfikacja gniazdek została opracowana pierwotnie dla systemu Unix (wersji BSD). W 1993r. zdefiniowano specyfikację gniazdek dla MS Windows wzorowaną na gniazdkach BSD, ale zarazem dostosowaną do właściwości systemu Windows. Aktualnie specyfikacja *Windows Socket*, w skrócie *Winsock*, dostępna jest we wszystkich 32-bitowych wersjach Windows. Winsock stanowi sieciowy interfejs programistyczny, obsługuje różne protokoły sieciowe, lecz sam nie jest protokołem.

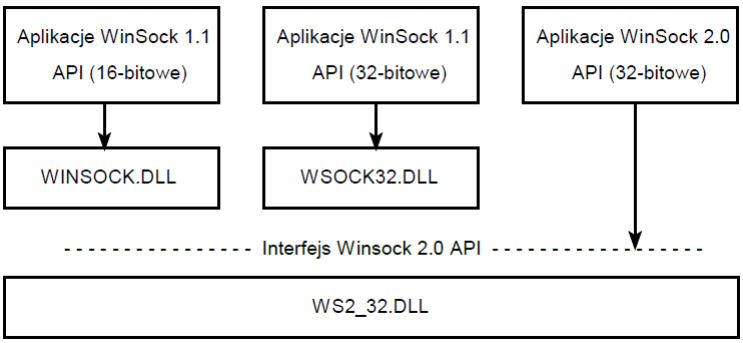
W modelu OSI interfejs gniazdek można umieścić między pomiędzy warstwą sesji a warstwą transportową. Można uważać, że każda z usług transportowych jest metodą przesyłania danych.

Gniazdko stanowi etykietkę końcowego punktu komunikacji - zwykle, każda konwersacja używa dwóch takich punktów: jednego po stronie klienta, drugiego po stronie serwera. Używane są dwa typy gniazdek: strumieniowe i datagramowe. Oba typy są dwukierunkowe. Gniazdko strumieniowe gwarantuje, że informacja będzie dostarczona w porządku, że nie będzie uszkodzona podczas transmisji, a ewentualne podwójne pakiety zostaną odrzucone. Gniazdko datagramowe jest mniej niezawodne. Komputery, które komunikują się za pomocą gniazdek datagramowych nie tworzą połączenia - zamiast tego wysyłają komunikaty bez potwierdzenia.

6.4 Interfejs gniazdek w systemie Windows

Przeniesienie specyfikacji gniazdek do systemu Windows wymagało rozwiązania kilku problemów. Windows jest bowiem systemem, który operuje na modelu komunikatów, a ponadto w jego wcześniejszych wersjach stosowana była wielozadaniowość bez wywłaszczania. Gdy wystąpi pewne zdarzenie, Windows wysyła komunikat do funkcji okienkowej - kod zawarty w tej funkcji musi przetworzyć komunikat tak szybko jak to możliwe i powrócić do stanu nieaktywnego. Zatem realizowane zadania powinny zwracać procesor do systemu po umiarkowanym czasie (praktycznie najpóźniej po około 100 ms) -w przeciwnym razie inne zadania zostałyby unieruchomione. Zauważmy, że problem ten nie występuje w systemie Unix (i nowszych wersjach Windows) ponieważ zadania oczekujące na koniec operacji zostają wywłaszczone.

Podane ograniczenie czasowe mogłoby by być ewentualnie spełnione jedynie w odniesieniu do komunikacji z komputerami lokalnymi. Jednak w przypadku Internetu, gdy przesyłane dane przechodzą przez wiele łączy i komputerów pośredniczących, należy przewidywać czasy znacznie dłuższe. Z tego powodu w specyfikacji *Windows Socket API*, obok typowych operacji na gniazdkach przeniesionych z Unixa, wprowadzono dodatkowe funkcje, stanowiące asynchroniczne wersje funkcji podstawowych. Funkcje asynchroniczne przygotowują i uruchamiają odpowiednie operacje, ale nie czekają na ich pełne zakończenie. Dopiero gdy operacja zostanie wykonana, albo gdy wystąpi możliwość wykonania określonych funkcji sieciowych, aplikacja jest powiadamiana asynchronicznie za pomocą komunikatu. Dodatkową zaletą takiego rozwiązania jest możliwość jednoczesnej komunikacji z wieloma komputerami



Rysunek 6.1: Sposób odwoływania się do interfejsu Winsock

Funkcje realizujące operacje na gniazdkach implementowane są za pomocą bibliotek DLL. Odwołania zgodne ze specyfikacją *Winsock 2* kierowane są do biblioteki `WS2_32.DLL`, natomiast biblioteki `WINSOCK.DLL` (dla aplikacji 16-bitowych) i `WSOCK32.DLL` (dla aplikacji 32-bitowych) dostosowują wołania określone wg specyfikacji 1.1 do wersji 2. Ilustruje to Rys. 6.1

W systemie MS Windows funkcje API są zwykle implementowane za pomocą bibliotek dynamicznych DLL. Z kolei kod zawarty w tych bibliotekach odwołuje się funkcji usługowych udostępnianych przez sterowniki VxD. Ponieważ kod VxD wykonywany jest na poziomie uprzywilejowania systemu, więc nie występują ograniczenia w zakresie dostępu do sprzętu. W tej sytuacji kod zawarty w bibliotekach DLL może być wykonywany na poziomie uprzywilejowania programu użytkownika.

6.5 Przegląd funkcji gniazdek w systemie Windows

Opisy funkcji definiowanych w ramach Windows Socket tworzą interfejs znany jako Winsock API - funkcje te można podzielić na trzy grupy:

1. właściwe operacje na gniazdkach, stanowiące podzbiór zestawu opracowanego dla systemu Unix (BSD);
2. funkcje informacyjne (ang. database functions), przekształcają nazwy komputerów serwera i klienta na format używany przez oprogramowanie;
3. funkcje specyficzne dla systemu Windows, stanowią rozszerzenie istniejącego zestawu funkcji Windows do obsługi zdarzeń.

Pakiet operacji udostępnianych przez *Windows Socket API* musi być zainicjalizowany w początkowej części programu - realizuje to funkcja `WSAStartUp`; w końcowej części programu, analogicznie, musi być wykonana funkcja `WSACleanUp`.

6.6 Aplikacja przykładowa pracująca w trybie konsoli

Rozpatrzymy teraz dwa proste programy [1], tworzące parę klient - serwer, które zilustrują podstawowe zasady komunikacji za pomocą gniazdek. Oba programy pracują w trybie tekstowym i wykorzystują funkcje blokujące, które nie są odpowiednie dla systemu Windows, ale istotnie upraszczają zrozumienie kodu. Po zestawieniu połączenia serwer wyświetla informacje o adresie komputera, z którym nastąpiło połączenie i oczekuje na komunikaty od klienta. Kolejne komunikaty otrzymywane od klienta są wyświetlane aż do zakończenia sesji. Po zakończeniu sesji serwer zamyka połączenie i ponownie wchodzi w stan oczekiwania na klienta.

Oba programy można skompilować za pomocą 32-bitowego kompilatora Borland BCC32 wersji 4.5 lub wyższej.

[1] Zieliński J., Chrostowski D.: TCP/IP i biblioteka Winsock 2.0 - podstawy architektury i sposób wykorzystania.

Tablica 6.1: Wybrane operacje na gniazdkach

Funkcja	Zastosowanie
`accept`	zaakceptowanie nawiązanego połączenia
`bind`	dowiązanie nazwy do gniazdka
`closesocket`	zamknięcie gniazdka
`connect`	zainicjalizowanie połączenia
`getsockname`	podawanie nazwy dowiązanej do gniazdka
`getsockopt`	podawanie parametrów gniazdka
`htonl`	(ang. **h**ost **to** **n**et) zamiana wartości `unsigned long` na format przyjęty w sieci (zmiana kolejności bajtów)
`htons`	(ang. **h**ost **to** **n**et) zamiana wartości `unsigned short` na format przyjęty w sieci (zmiana kolejności bajtów)
`inet_addr`	konwersja adresu liczbowego IP (z kropkami) na liczbę 32-bitową
`inet_ntoa`	konwersja liczby 32-bitowej na adres liczbowy IP (z kropkami)
`ioctlsocket`	ustalanie własności gniazdka
`listen`	oczekiwanie na połączenie z gniazdkiem
`ntohl`	(ang. **n**et **to** **h**ost) zamiana wartości `unsigned long` na format przyjęty w lokalnym komputerze (zmiana kolejności bajtów)
`ntohs`	(ang. **n**et **to** **h**ost) zamiana wartości `unsigned short` na format przyjęty w lokalnym komputerze (zmiana kolejności bajtów)
`recv`	odbiór danych z połączonego gniazdka
`recvfrom`	odbiór danych z gniazdka wg adresu IP i numeru portu
`select`	synchroniczne multipleksowanie wejścia/wyjścia
`send`	wysyłanie danych poprzez połączone gniazdko
`sendto`	wysyłanie danych do podanego gniazdka
`setsockopt`	ustawianie opcji gniazdka
`shutdown`	zamknięcie
`socket`	tworzenie gniazdka

Tablica 6.2: Operacje rozwiązywania nazw i adresów

Funkcja	Zastosowanie
`gethostbyaddr`	wyznaczanie nazwy komputera na podstawie adresu IP
`gethostbyname`	wyznaczanie adresu IP na podstawie nazwy komputera
`gethostname`	podawanie nazwy komputera lokalnego (ang. host)
`getprotobyname`	zwraca numer protokołu na podstawie nazwy
`getprotobynumber`	zwraca nazwę protokołu na podstawie numeru
`getservbyname`	zwraca nazwę usługi i port na podstawie nazwy i protokołu
`getservbyport`	zwraca nazwę usługi i port na podstawie portu i protokołu

Tablica 6.3: Wybrane operacje Winsock

Funkcja	Zastosowanie
WSAAsyncGetHostByAddr	wersja asynchroniczna gethostbyaddr
WSAAsyncGetHostByName	wersja asynchroniczna gethostbyname
WSAAsyncGetProtoByName	wersja asynchroniczna getprotobyname
WSAAsyncGetProtoByNumber	wersja asynchroniczna getprotobynumber
WSAAsyncGetServByName	wersja asynchroniczna getservbyname
WSAAsyncGetServByPort	wersja asynchroniczna getservbyport
WSAAsyncSelect	wersja asynchroniczna select
WSACancelAsyncRequest	kasowanie zaległych wywołań funkcji WSAAsyncGet...
WSAAsyncGet	kasowanie zaległych wywołań blokujących
WSACleanUp	zwalnianie zasobów przydzielonych przez WSAStartUp
WSAGetLastError	informacje o ostatnim błędzie funkcji API
WSAIsBlocking	sprawdzenie czy występuje zaległe wołanie blokujące
WSASetBlockingHook	przechwycenie bazowego mechanizmu blokowania
WSASetLastError	ustawienie kodu błędu zwracanego przez funkcję WSAGetLastError
WSAStartUp	inicjacja bazowej biblioteki DLL
WSAUnhookBlockingHook	przywrócenie pierwotnego mechanizmu blokowania

6.6.1 Program serwera

Serwer jest procesem oczekującym na określoną liczbę połączeń z klientem. Zadanie serwera polega na obsłudze żądań klientów, które w opisanym dalej przykładzie ograniczone jest do wyświetlania na ekranie nadchodzących tekstów.

`socket() -> bind() -> listen() -> accept()`

Serwer musi oczekiwać na połączenia pod powszechnie znaną nazwą. W przypadku stosowania protokołów TCP/IP nazwę tę stanowi adres internetowy IP oraz numer portu lokalnego interfejsu.

W celu zestawienia połączenia konieczne jest wykonanie kilku funkcji, które pokazano na rysunku. Po utworzeniu gniazdka (funkcja `socket`) jest ono wiązane z przypisaną mu nazwą - realizuje to funkcja bind. Funkcja `listen` przełącza gniazdko w tryb nasłuchiwania, a ewentualne próby nawiązania połączenia są akceptowane przez serwer za pomocą funkcji `accept`. Czynności te opisane są dalej szczegółowo na podstawie serwera przykładowego.

W początkowej części programu wykonywana jest funkcja `WSAStartup`, która ładuje wersję 1.1 biblioteki Winsock.

```
WSADATA wsas;
int result;
WORD wersja;
wersja = MAKEWORD(1, 1);
result = WSAStartup(wersja, &wsas);
Nastepnie tworzone jest gniazdko za pomoca funkcji socket
SOCKET s;
s = socket(AF_INET, SOCK_STREAM, 0);
```

Stałe `AF_INET` i `SOCK_STREAM` określają, odpowiednio, tworzenie gniazdka internetowego i strumieniowego. Z kolei przystępujemy do określania poszczególnych pól struktury adresowej `sa`

```
struct sockaddr_in sa;
memset((void *)(&sa), 0, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(ST_PORT);
sa.sin_addr.s_addr = htonl(INADDR_ANY);
```

Najpierw, za pomocą funkcji `memset`, cała struktura `sa` wypełniana jest zerami (pierwszy argument tej funkcji wskazuje adres obszaru, drugi zawiera wpisywaną wartość, trzeci określa rozmiar obszaru). Następnie wpisujemy:

- wartość `AF_INET` do pola `sin_family`, które określa rodzinę protokołów (tu: protokoły internetowe);
- numer portu, na którym będzie nasłuchiwał serwer wpisujemy do pola `sa.sin_port`;

- pole `sin_addr` określa dla jakich adresów będą akceptowane połączenia - adres specjalny `INADDR_ANY` pozwala serwerowi na nasłuchiwanie aktywności klienta we wszystkich interfejsach sieciowych dostępnych w komputerze.

Mamy więc wypełnioną strukturę adresową, którą trzeba skojarzyć z wcześniej utworzonym gniazdkiem - realizuje to funkcja `bind`:

```
result = bind(s, (struct sockaddr FAR*)&sa, sizeof(sa));
```

Funkcja `bind` realizuje dowiązanie. W ten sposób zostaje ustalony port, przez który będą nadchodziły żądania do serwera, a także zostają ustalone rodzaje połączeń, akceptowane przez serwer. Formalnie, prototyp funkcji `bind` ma postać:

```
int bind(SOCKET s, const struct sockaddr FAR * addr, int namelen);
```

Pierwszy parametr stanowi deskryptor identyfikujący gniazdko (nie dowiązane). Drugim parametrem jest adres przypisywany do gniazdka. Struktura `sockaddr` jest definiowana następująco

```
struct sockaddr {
   u_short sa_family;
   char sa_data[14];
};
```

Trzeci parametr określa rozmiar parametru `addr`. Jeśli nie wystąpił błąd, to omawiana funkcja zwraca wartość `0`, w przeciwnym razie wartość `SOCKET_ERROR`.

Dodajmy, że jeśli numer portu w strukturze adresowej był równy zero, to biblioteka Winsock wyznaczy unikatowy numer portu w przedziale 1024 - 5000. Przydzielony numer można później odczytać za pomocą funkcji `getsockname`.

Z kolei następuje wykonanie funkcji `listen`, która nakazuje gniazdku wejść w stan oczekiwania na nadchodzące połączenia tworzy kolejkę, do której będą wpisywane ewentualne połączenia

```
result = listen(s, 5);
```

Funkcja `listen` jest zwykle używana przez serwery, które obsługują w tym samym czasie więcej niż jedno żądanie. Pierwszym parametrem funkcji jest uchwyt gniazdka, zaś drugim jest maksymalny rozmiar kolejki oczekujących połączeń. Wartość 5 jest dostatecznie duża, tak że dopiero żądanie szóstego klienta zostanie odrzucone. Po opisanym dalej zaakceptowaniu połączenia przez serwer, jest ono usuwane z kolejki, dzięki czemu mogą się zgłaszać dalsze procesy. Funkcja `listen` stosowana jest tylko do gniazdek strumieniowych.

Kolejny fragment programu zawiera pętlę obsługi kolejnych połączeń. Istotnym elementem jest tu funkcja `accept`, która zatrzymuje wykonywanie programu (gniazdko blokujące) czekając aż pojawi się próba połączenia z serwerem.

```
SOCKET si;
struct sockaddr_in sc;
int lenc;
for(;;)
{
   lenc = sizeof(sc);
   si = accept(s, (struct sockaddr FAR *) &sc, &lenc);
```

Jeśli taka próba wystąpi, to przeprowadza się negocjacje TCP/IP, i tworzy się dodatkowe gniazdko do komunikacji z klientem. Deskryptor tego gniazdka jest przypisywany zmiennej `si` typu `SOCKET` i dalej wykorzystywany do komunikacji z klientem - to nowe gniazdko powinno być używane we wszystkich dalszych operacjach związanych z tym klientem. Ponadto funkcja może zwrócić poprzez drugi argument adres klienta, z którym nawiązała połączenie.

Po nawiązaniu połączenia oryginalne gniazdko nasłuchowe jest ponownie gotowe do działania i może w dalszym ciągu przyjmować połączenia innych klientów. Połączenia przychodzące są magazynowane przez oprogramowanie w kolejce utworzonej przez funkcję `listen`. Następnie, gdy program wywoła funkcję `accept`, pierwsza pozycja z kolejki jest podawana do obsługi.

Podana niżej pętla służy do odczytu danych przez gniazdko komunikacji z klientem. Głównym elementem jest funkcja `recv`, która odbiera dane z gniazdka. Pierwszy parametr tej funkcji zawiera uchwyt gniazdka, drugi określa adres bufora `(char FAR * buf)`, trzeci - rozmiar bufora, a czwarty zawiera znaczniki specyfikujące wykonywane operacje.

```
SOCKET si;
char buf [80];
while(recv(si, buf, 80, 0) > 0)
{
   if(strcmp(buf, "KONIEC") == 0)
   {
      closesocket(si);
      WSACleanup();
      return;
   }
   printf("\n%s",buf);
};
```

Funkcja `recv` działa następująco:

- jeśli dane nadeszły z sieci, to funkcja kopiuje je do podanego bufora i kończy działanie zwracając liczbę odczytanych bajtów;
- jeśli dane nie nadeszły, to funkcja wchodzi w stan oczekiwania blokując dalsze wykonywanie programu;
- jeśli połączenie zostało zamknięte, to funkcja zwraca zero;
- jeśli wystąpił błąd, to funkcja zwraca wartość `SOCKET_ERROR`.

W podanym przykładzie, jeśli klient prześle słowo KONIEC, to serwer zamyka gniazdko, co kończy pętlę odbioru.

6.6.2 Program klienta

Podobnie jak w programie serwera, następuje zaincjalizowanie biblioteki Winsock, utworzenie gniazdka i przypisanie adresu.

```
SOCKET s;
struct sockaddr_in sa;
WSADATA wsas;
WORD wersja;
wersja = MAKEWORD(2,0);
WSAStartup(wersja, &wsas);
s = socket(AF_INET, SOCK_STREAM, 0);
memset((void *)(&sa), 0, sizeof(sa));
sa.sin_family = AF_INET;
sa.sin_port = htons(10000);
sa.sin_addr.s_addr = inet_addr(argv[1]);
```

Adres internetowy określony jest przez numer komputera, na którym uruchomiono serwer - adres ten podawany jest w linii wywołania programu (`argv[1]`). Może to być w szczególności numer 127.0.0.1 pozwalający na testowanie programu klienta i serwera na tym samym komputerze.

Z kolei wykonywana jest funkcja `connect`, która próbuje połączyć się z komputerem o adresie podanym w strukturze.

```
int result;
result = connect(s, (struct sockaddr FAR *) &sa, sizeof(sa));
if(result == SOCKET_ERROR)
{
   printf("\nBlad polaczenia!");
   return;
}
```

Jeśli połączenie nie zostało zestawione, to funkcja zwraca wartość `SOCKET_ERROR`. Po zestawieniu połączenia użytkownik wpisuje kolejne wiersze, które przesyłane są do odległego komputera za pomocą funkcji `send`. Funkcja ta wysyła dane za pomocą uprzednio połączonego gniazdka. Argumenty tej funkcji są prawie takie same jak argumenty wcześniej omawianej funkcji `recv`. W przypadku pomyślnego wysłania funkcja zwraca liczbę wysłanych znaków, w przeciwnym razie zwracana jest wartość `SOCKET_ERROR`.

```
int dlug;
char buf[80];
for(;;)
{
   fgets(buf, 80, stdin);
   dlug = strlen(buf); buf [dlug-1] = '\0';
   send(s, buf, dlug, 0);
   if(strcmp(buf, "KONIEC") == 0) break;
}
closesocket(s);
WSACleanup();
```

Wpisanie słowa `KONIEC` powoduje zakończenie działania klienta (i serwera). Przed zakończeniem programu następuje zamknięcie gniazdka i biblioteki Winsock.

Ćwiczenie 7

WĄTKI POSIX

7.1 Wstęp

Współczesne systemy operacyjne oferują rozmaite mechanizmy przetwarzania współbieżnego i rozproszonego. Należą do nich m.in.: wieloprocesowość, wielowątkowość, włókna, czyli wątki użytkownika (ang.: fibers), mechanizmy przetwarzania sieciowego: DCOM, CORBA i wiele innych. Ich źródła sięgają daleko wstecz. Wieloprocesowość została wprowadzona w systemie UNIX, a wielowątkowość zdefiniowano w standardzie POSIX. Dzisiaj możemy korzystać z wielu standardów obsługi wątków (np. wątki w systemie MS Windows). Zdarza się, że w ramach jednego systemu operacyjnego dostępnych jest wiele bibliotek oferujących dostęp do wielowątkowości (np. w Linuxie są co najmniej cztery).

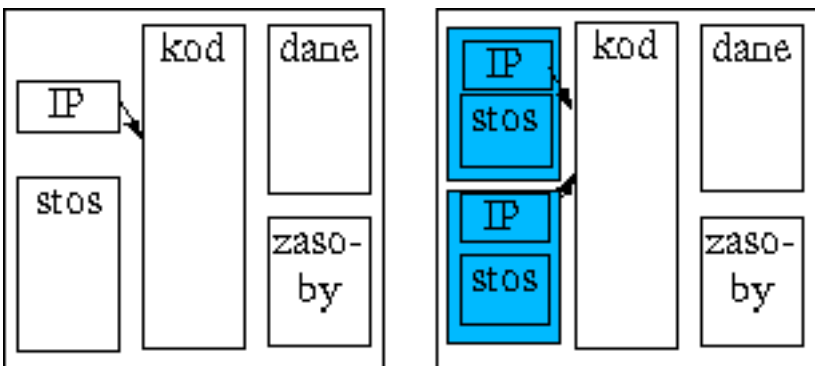
W oprogramowaniu użytkowym wątki zyskują coraz większą popularność. Dzieje się to za sprawą większej wydajności aplikacji wielowątkowych nad wieloprocesowymi. Poza aspektami wydajnościowymi, bardzo często w aplikacjach wielowątkowych towarzyszy wyższy komfort współpracy z interfejsem użytkownika, dzięki temu, że oddzielny wątek obsługujący zdarzenia płynące od użytkownika zawsze jest gotowy do pracy. Ponadto w fazie projektowania aplikacji można podzielić ją na niezależne zadania reprezentowane później przez wątki.

Na rys. 7.1 przedstawiono reprezentację procesów i wątków w systemie. Widzimy, że wiele wątków może być związanych z jednym procesem. W takiej sytuacji współdzielą one część

kontekstu procesu: segment kodu, danych i zasoby systemowe. Jak pamiętamy, procesy, które powstały nawet z jednego procesu macierzystego, posiadały oddzielne segmenty danych i zasobów (mogły jedynie korzystać z tego samego segmentu kodu). Istota wyższej efektywności aplikacji wielowątkowych polega właśnie na większej wspólnej części kontekstu procesu, co zmniejsza czas przełączania takich zadań w systemie oraz ułatwia ich wzajemną komunikację.

7.2 Tworzenie wątków POSIX

Utworzenie nowego procesu w systemie UNIX wymagało stworzenia kopii bieżącego procesu za pomocą funkcji `fork`. Natomiast utworzenie nowego wątku wymaga wywołania funkcji POSIX `pthread_create` i podanie funkcji reprezentującej nowy wątek. Poniższe przykłady prezentują tworzenie procesu i wątku.



Rysunek 7.1: Proces jednowątkowy i wielowątkowy

Tworzenie procesu:

```
int pid = fork();
if(pid == 0)
{
   // proces potomny
}
else
{
   // proces macierzysty
}
```

Tworzenie wątku POSIX:

```
watek glowny: watek potomny:
pthread_create(f_watku)

f_watku(arg)
... { ...

return }
pthread_join
```

Uwaga! jeśli wątek główny skończy się przed zakończeniem wątków potomnych, wątki potomne zostaną zakończone automatycznie.

7.3 Główne funkcje operujące na wątkach

Utworzenie nowego wątku wymaga podania wskaźnika do funkcji, która reprezentuje ten wątek. Funkcja ta będzie wykonywana jako współbieżne zadanie systemu operacyjnego. Przyjmuje się, że funkcja wątku zwraca wartość typu `void*` oraz pobiera jeden argument również typu `void*`. Umożliwia to zwrot lub przekazanie wskaźnika do dowolnej struktury lub rzutowania `void*` na np. typ całkowity `int`. Funkcja tworząca wątek zwraca zmienną identyfikującą wątek. Jest to zmienna typu `pthread_t`, która zwracana jest przez wartość pierwszego argumentu funkcji. Prototyp funkcji tworzącej wątek jest następujący:

```
int pthread_create(pthread_t *w, pthread_attr_t *attr,
void *(* f_watku)(void *), void *arg);
```

Zakończenie funkcji wątku następuje po zakończeniu wykonywania się funkcji wątku. Dlatego najprostszym sposobem jest wykonanie polecenia `return <wartość>`. Istnieje jednak dodatkowa funkcja kończąca wykonanie wątku. Może ona znaleźć zastosowanie tam, gdzie wykonanie `return <wartość>` nie spowoduje zakończenie funkcji wątku, np. w funkcjach wywołanych przez funkcję wątku. Funkcja kończąca wątek ma postać:

```
int pthread_exit(void *wartosc_zwracana) ;
```

Po utworzeniu wątku, wątek główny wykonuje się współbieżnie. Wątek główny może wejść w stan oczekiwania na zakończenie dowolnego wątku potomnego, za pomocą funkcji:

```
int pthread_join(pthread_t w, void ** wynik);
```

Dodatkowo można odczytać wartość zwracaną przez wątek. Funkcja ta zakończy natychmiast działanie, jeśli wątek potomny (wskazywany przez `w`) został wcześniej zakończony. Oznacza to, że informacja o wątku jest przechowywana przez system do czasu wykonania funkcji `pthread_join`. Czasami może być to zbyteczne, np. gdy wątki potomne wykonują operacje nie związane z wątkami macierzystymi i niepotrzebnie obciążać system.

Wątki, o których informacje nie są przechowywane w systemie, nazywają się wątkami odłączonymi (ang. detached). Aby uczynić wątek odłączonym, należy wykonać funkcję:

```
int pthread_detach(pthread_t w);
```

Poza wymienionymi funkcjami istnieje jeszcze wiele innych funkcji operujących na wątkach. Można o nich się dowiedzieć, np. za pomocą standardowych funkcji pomocy w systemie: `man pthread_...`. My wymienimy jeszcze dwie, które mogą być przydatne w zwykłych aplikacjach. Funkcja zwracająca identyfikator bieżącego wątku jest następująca:

```
pthread_t pthread_self();
```

Natomiast do porównywania identyfikatorów wątków służy funkcja:

```
int pthread_equal(pthread_t w1, pthread_t w2);
```

7.4 Kompilacja i uruchamianie programów

Przeanalizujmy program przykładowy (Hello world!) wykorzystujący wymienione funkcje. Program musi włączać plik nagłówkowy `pthread.h` zawierający prototypy funkcji POSIX. Funkcja wątku nie wykorzystuje przekazywanego argumentu ani nie zwraca żadnej konkretnej wartości. Wyświetla jedynie napis informujący o wykonywaniu się funkcji wątku. Wątek główny uruchamia wątek potomny a następnie czeka na jego zakończenie. Potem wyświetla własny napis.

```
#include <stdio.h>
#include <pthread.h>
void *f(void *i)
{
   printf("Thread says: Hallo world\n");
   return NULL;
}

int main()
{
   pthread_t w;
   pthread_create(&w, NULL, f, NULL);
   pthread_join(w, NULL);
   printf("Hello world\n");
}
```

Powyższy program należy skompilować włączając bibliotekę dynamiczną funkcji wątków POSIX. Biblioteka ta nosi nazwę `libpthread.so`, więc opcje kompilatora powinny być następujące:

```
gcc -lpthread nazwa_pliku.c
```

Zadanie

Napisać program, który szuka liczb pierwszych. Wątek główny uruchamia wątki potomne przekazując im kolejne liczby całkowite (do określonej granicy). Wątki potomne, wykonujące tę samą funkcję, sprawdzają, czy przekazana liczba jest pierwsza (za pomocą dowolnego algorytmu) i jeżeli tak jest, to wyświetlają zadaną liczbę. Skutek działania programu powinien być taki, że zostaną wyświetlone wszystkie liczby pierwsze z zadanego przedziału.

7.5 Mechanizmy synchronizacji wątków

Standard POSIX dostarcza także mechanizmów synchronizacji wątków. Należą do nich semafory i zmienne warunkowe. Mechanizmy synchronizacji są bardzo istotne w przypadku aplikacji wielowątkowych, ponieważ wątki współdzielą tę samą przestrzeń danych.

Załóżmy, że poniższa funkcja jest wykonywana w środowisku wielowątkowym. Możemy więc mniemać, że w dowolnym momencie działania tej funkcji może nastąpić przełączenie zadań i może kontynuować wykonanie inny wątek, który także może rozpocząć wykonywanie tej funkcji.

```
int counter()
{
   static int c = 0;
   return c++;
}
```

Jeżeli przełączenie zadania nastąpi pomiędzy operacjami inkrementacji (`c++`) i powrotu z funkcji (`return`), a inny wątek rozpocznie wykonywanie funkcji `counter`, zmienna `c` zostanie zwiększona dwukrotnie, a wątek, który został pierwotnie wstrzymany otrzyma błędną wartość zwróconą przez funkcję `counter`. Aby się przed tym obronić, należy zabezpieczyć się przed rozpoczęciem wykonywania operacji

modyfikacji zmiennej `c` i zwróceniem wyniku. Zabezpieczenie może polegać na zastosowaniu dowolnego mechanizmu synchronizacji i stworzeniu sekcji krytycznej.

Najprostszym mechanizmem są semafory. Są one reprezentowane przez dwie główne funkcje: opuszczenia (`LOCK`) i podniesienia (`UNLOCK`). Funkcja opuszczenia semafora jest blokująca: jeżeli wątek wykonujący funkcję `LOCK` napotka semafor opuszczony, zostaje on wstrzymany w kolejce wątków wstrzymanych. Pierwszy wątek w kolejce zostaje wznowiony, gdy wątek przebywający w sekcji krytycznej podniesie semafor. Jeśli semafor jest podniesiony, funkcja `LOCK` natychmiast po opuszczeniu semafora kończy działanie.

Modyfikacja powyższej funkcji rozwiązująca problem korzystania przez wiele wątków (czyniąca ją tzw. wielowątkową), może być następująca:

```
int counter()
{
   static int c = 0;
   int local_c;
   LOCK;
   local_c = c++;
   UNLOCK;
   return local_c;
}
```

W standardzie POSIX mamy do dyspozycji pięć funkcji operujących na semaforach. Semafor jest identyfikowany przez zmienną typu `pthread_mutex_t`. Funkcja tworząca semafor posiada następujący prototyp:

```
int pthread_mutex_init(pthread_mutex_t *m);
```

Natomiast usunięcie semafora wykonywane jest za pomocą funkcji:

```
int pthread_mutex_destroy(pthread_mutex_t *m);
```

Funkcja opuszczająca semafor posiada następujący prototyp:

```
int pthread_mutex_lock(pthread_mutex_t *m);
```

Z kolei do podniesienia semafora służy funkcja:

```
int pthread_mutex_unlock(pthread_mutex_t *m);
```

W standardzie POSIX dostępna jest jeszcze jedna funkcja synchronizacji za pomocą semaforów. Posiada ona następujący prototyp:

```
int pthread_mutex_trylock(pthread_mutex_t *m);
```

Jest ona nieblokującą wersją opuszczenia semafora. Jeśli wątek napotka semafor opuszczony, nie zostaje wstrzymany w kolejce wątków oczekujących na semaforze, a kończy działanie natychmiast zwracając wartość `EBUSY`.

Zadanie

Zmodyfikować program szukający liczby pierwsze. Każdy wątek znalazłszy liczbę pierwszą nie wyświetla jej na ekranie, a zapisuje do współdzielonej tablicy. Po zakończeniu wszystkich wątków, wątek główny wyświetla zawartość współdzielonej tablicy. Tablica może być reprezetowana przez następujące zmienne:

```
int primes[MAX_PRIME]; // tablica z liczbami pierwszymi
int curr_p = 0; // ile w tablicy jest wpisanych liczb pierwszych
pthread_mutex_t m_p; // semafor chroniacy dostepu do tablicy
```

Ponieważ każdy zapis do tablicy wiąże się ze zwiększeniem zmiennej `curr_p`*, operacje te muszą być wykonywane w sekcji krytycznej.*

Ćwiczenie 8

POTOKI W SYSTEMIE UNIX

8.1 Wstęp

System operacyjny UNIX oferuje kilka różnych mechanizmów komunikacji. Należą do nich: pamięć współdzielona, potoki (anonimowe: pipes i nazwane: fifo), gniazda (sockets), komunikaty i zdalne wywołanie procedury (RPC).

Potoki anonimowe, nazywane także łączami komunikacyjnymi, umożliwiają jednokierunkową i asynchroniczną komunikację między pokrewnymi procesami utworzonymi za pomocą funkcji `fork`. W potokach dane przesyłane są w postaci strumienia bajtów, sekwencyjnie. Interpretacja należy do procesów korzystających z potoku. Odczytanie informacji powoduje usunięcie jej z potoku. Ponadto funkcja odczytu jest blokująca, tzn. jeżeli w potoku nie ma wystarczająco dużo danych, aby spełnić żądanie procesu wywołującego funkcję odczytu, jest on wstrzymywany do czasu zapisu do potoku odpowiedniej ilości danych przez inny proces.

Potoki nazwane (inne określenia: łącza nazwane, kolejki FIFO) są rozszerzeniem potoków anonimowych. Z potoków nazwanych mogą korzystać procesy niespokrewnione. Jest to możliwe dzięki związaniu nazwy potoku z nazwą pliku we wspólnym systemie plików.

8.2 Potoki anonimowe (łącza komunikacyjne)

Potok anonimowy jest tworzony za pomocą funkcji systemowej o następującym prototypie:

```
int pipe(int handles[2]);
```

Jeśli działanie funkcji się powiedzie, zwróci ona wartość zero, a do wskazanej tablicy zostaną wpisane identyfikatory dostępu do potoku. Pod indeksem 0 tej tablicy zostanie wpisany identyfikator dostępu służący do odczytu, pod indeksem 1 identyfikator służący do zapisu. Jeśli po wykonaniu tej funkcji zostanie stworzony proces potomny i będzie on posiadał kopię tablicy identyfikatorów dostępu do potoku, oba procesy będą mogły go wykorzystać do wzajemnej komunikacji. Na potokach obowiązują standardowe operacje odczytu i zapisu jak na plikach. Dla przypomnienia podajemy ich prototypy:

```
int read(int handle, void *buf, int size);
int write(int handle, void *buf, int size);
```

Niewykorzystywany potok należy zamknąć. W tym celu należy wykonać funkcję `close` zarówno dla identyfikatora dostępu odczytu jak i zapisu do potoku. Poniższy program demonstruje użycie potoku anonimowego:

```
#include <unistd.h>
#include <stdio.h>
#define ODCZYT 0
#define ZAPIS 1

int main()
{
   int potok[2];
   int x;
   puts("Program pipes startuje");
   puts("Tworze potok");
```

```
   pipe(potok);

   puts("fork");
   if(fork())
   {
      puts("Proces macierzysty");
      close(potok[ZAPIS]);
      puts("Czekam na proces potomny");
      read(potok[ODCZYT], &x, sizeof(x));
      printf("Proces macierzysty otrzymal %d\n", x);
      close(potok[ODCZYT]);
   }
   else
   {
      puts("Proces potomny");
      close(potok[ODCZYT]);
      x = 10;
      sleep(10);
      write(potok[ZAPIS], &x, sizeof(x));
      puts("Proces potomny wyslal");
      close(potok[ZAPIS]);
   }

   puts("Program pipes konczy");
}
```

Proces potomny przesyła do procesu macierzystego liczbę całkowitą (`int`) o wartości 10. Do praktycznej komunikacji procesów należy zdefiniować odpowiedni protokół wymiany informacji, ponieważ, jak już wspomniano, potok zawiera strumień bajtów, których interpretacja należy jedynie do procesów korzystających z potoku.

Zadanie

Napisać program, który demonstruje przetwarzanie potokowe. Składa się z trzech procesów, z których pierwszy wczytuje ciąg znaków z konsoli i przekazuje go za pomocą potoku do procesu drugiego. Drugi proces zamienia wszystkie małe litery otrzymanego ciągu na duże i przekazuje go dalej do procesu trzeciego, również za pomocą potoku. Ostatni trzeci proces wyświetla odebrany ciąg znaków na ekranie. Po wykonaniu swojego zadania procesy kończą działanie.

8.3 Potoki nazwane (kolejki FIFO)

Potok nazwany jest podobny do potoku anonimowego, z tą różnicą, że może być dostępny jako część systemu plików. Może być otwarty przez wiele niezależnych procesów do odczytu i zapisu. Podczas wymiany danych przez potok nazwany, jądro przesyła dane bez zapisu do systemu plików. W systemie plików potok nazwany jest widoczny jako plik specjalnego typu `pipe`, np.:

```
prw-r--r--  1 0      maj 8 19:56 test|
-rwxr-xr-x  1 13894  maj 8 19:55 a.out*
-rw-r--r--  1 139    maj 8 19:55 fifo.c
-rw-r--r--  1 912    maj 7 19:12 pipes.c
-rw-r--r--  1 20480  maj 8 19:52 potoki.sdw
-rw-r--r--  1 1093   maj 7 20:00 synchro_pipes.c
```

Stworzenie potoku nazwanego wykonywane jest przez funkcję systemową o prototypie:

```
int mkfifo(const char *name, mode_t mode);
```

Istnieje również prostszy sposób utworzenie w systemie plików potoku nazwanego, za pomocą powłoki systemowej. Służy do tego analogiczne polecenie:

```
$ mkfifo <nazwa>
```

Utworzony potok nazwany istnieje w systemie plików aż do jego usunięcia za pomocą standardowych funkcji (`unlink` w C lub `rm` w powłoce). Procesy chcące nawiązać komunikację za pomocą tego potoku muszą go otworzyć jako plik. Standardowo otwarcie potoku nazwanego jest blokujące do czasu, gdy dla dany potok nazwany będzie otwierany dla obu typów operacji: odczytu i zapisu. Poniżej przedstawiamy dwa programy, które komunikują się za pomocą potoku nazwanego `test` w katalogu bieżącym. Po 10 sekundach od uruchomienia obu programów, jeden z nich wysyła daną przez potok nazwany, co powoduje zakończenie działania obu programów.

```
#include <unistd.h>
#include <fcntl.h>
int main()
{
   intx = 0;
   int f = 0;
   puts("Otwieram");
   f = open("test", O_WRONLY);
   sleep(10);
   puts("Wysylam i koncze");
   write(f, &x, sizeof(x));
   close(f);
}
```

Program drugi:

```
#include <unistd.h>
#include <fcntl.h>
int main()
{
   int f;
   int x = 0;
   puts("Otwieram");
   f = open("test", O_RDONLY);
   read(f, &x, sizeof(x));
   puts("Odebralem i koncze");
   close(f);
}
```

Zadanie

Zmodyfikować program z poprzedniego zadania tak, aby korzystał z potoków nazwanych.

8.4 Synchronizacja za pomocą potoków

Niektóre operacje na potokach są blokujące. W szczególności interesujące jest blokowanie operacji odczytu z pustego potoku do momentu, gdy inny proces umieści w nim jakieś dane. Umożliwia to zastosowanie potoków do synchronizacji procesów.

Zastanówmy się teraz nad implementacją semafora za pomocą potoków. Na semaforze można wykonywać dwie operacje: opuszczenie `LOCK` i podniesienie `UNLOCK`. Operacja opuszczenia może być blokująca do czasu, gdy inny proces nie wykona operacji `UNLOCK`. Przyjmując, że semafor jest reprezentowany przez potok, opuszczenie semafora oznacza wykonanie blokującej operacji odczytu z potoku, natomiast podniesienie semafora wiąże się z operacja zapisu do potoku. Jeżeli w potoku jest informacja do odczytu, oznacza to, że semafor jest podniesiony i najbliższa operacja opuszczenia `LOCK` zakończy się natychmiast. Ponadto potok zostanie opróżniony i każda następna operacja odczytu, czyli próby opuszczenia semafora, będą wstrzymywały kolejne procesy. Zapisanie danych do potoku spowoduje wznowienie jednego z ewentualnych procesów wstrzymanych, co jest równoważne z podniesieniem semafora. Implementacją semafora za pomocą potoku wymaga, aby inicjalnie w potoku znajdowała się dana do odczytu.

Poniżej przedstawiamy program, który implementuje semafor za pomocą potoku anonimowego:

```
#include <unistd.h>
#include <stdio.h>
#define ODCZYT 0
```

```
#define ZAPIS 1
#define WORKERS 10

int semafor[2];

void LOCK()
{
   int x;
   read(semafor[ODCZYT], &x, sizeof(x));
}

void UNLOCK()
{
   intx = 0;
   write(semafor[ZAPIS], &x, sizeof(x));
}

void worker(int id, int potok_k)
{
   intx = 0,i,j;
   printf("Worker %d startuje\n", id);
   LOCK();

   for(i = 0; i < 7; i++)
   {
      printf("[%d] ", id);
      fflush(stdout);
      sleep(1);
   }

   UNLOCK();
   printf("Worker %d konczy\n", id);
   write(potok_k, &x, sizeof(x));
}

int main()
{
   int potok_konczacy[2], i, x;
   pipe(potok_konczacy);
   pipe(semafor);
   UNLOCK(); // inicjalizacja semafora
   for(i = 0; i < WORKERS; i++)
   {
      if(!fork())
      {
         worker(i, potok_konczacy[ZAPIS]);
         return 0;
      }
   }

   for(i = 0; i < WORKERS; i++)
      read(potok_konczacy[ODCZYT], &x, sizeof(x));

   puts("koniec");
```

Usunięcie ograniczników sekcji krytycznej: `LOCK` i `UNLOCK` powoduje, że liczby wyświetlane przez procesy nie są zgrupowane.

Zadanie

Zaimplementować za pomocą potoku uproszczony semafor wielowartościowy. Semafor taki pozwala na wejście do sekcji krytycznej nie jednemu procesowi, lecz podanej inicjalnie (przy tworzeniu semafora) ich liczbie. Do demonstracji można wykorzystać powyższy przykład. Udowodnić poprawność działania

stworzonego semafora.

Ćwiczenie 9

WIELOWĄTKOWOŚĆ W JĘZYKU JAVA

9.1 Wstęp

Java, podobnie jak inne języki programowania, umożliwia implementację przetwarzania wielowątkowego. W chwili startu programu w Javie wykonywany jest jeden wątek (główny), w czasie działania programu możliwe jest tworzenie kolejnych wątków, które będą wykonywane równolegle. Wątki mogą posiadać dostęp do wspólnych zmiennych w ramach jednej aplikacji. Trzeba jednak pamiętać, że Java jest językiem obiektowym i implementacja wielowątkowości różni się nieco od modelu znanego z języków strukturalnych, takich jak język C.

Bardziej szczegółowy opis mechanizmów wielowątkowości dostępnych w Javie można znaleźć w bogatej literaturze, np. *"The Java Tutorial": http://java.sun.com/docs/books/tutorial/* lub w książce *S. Oaks, H. Wong: "Java Threads"*.

9.2 Implementacja przetwarzania wielowątkowego

Jednym ze sposobów wykonania wielu wątków jest wykorzystanie standardowej klasy `Thread` dostępnej w pakiecie `java.lang`. Należy zdefiniować własną klasę (np. `HelloThread`), dziedziczącą po klasie standardowej `Thread`. W nowej klasie należy zdefiniować metodę run, która będzie wykonywana w oddzielnym wątku. Metoda run jest częścią interfejsu klasy `Thread`.

```
public class HelloThread extends Thread {
   public void run() {
      System.out.println("Napis z watku utworzonego");
   }
}
```

Dalej należy utworzyć obiekt klasy `HelloThread`. Uruchomienie nowego wątku następuje w wyniku wywołania metody `start` (nie `run`) na obiekcie klasy "wątkowej". Metoda start jest metodą klasy `Thread`, która faktycznie tworzy nowy wątek i przekazuje sterowanie do kodu zdefiniowanego w metodzie run.

```
public class Main {
   public static void main(String args[]) {
      HelloThread helloThread = new HelloThread();
      helloThread.start();
      System.out.println("Napis z watku glownego");
      helloThread.join();
   }
}
```

Metoda `join` klasy `Thread` powoduje, że wątek oczekuje na zakończenie przetwarzania w innym wątku. Metodę `join` wywołuje na obiekcie wątku, na którego zakończenie oczekuje.

Metoda `sleep(long millis)` powoduje, że bieżący wątek zawiesza swoje działanie na określony czas. Jest to skuteczny sposób udostępnienia czasu procesora innym wątkom działającym w aplikacji lub innym aplikacjom działającym na tym samym systemie komputerowym. W metodzie `main` możliwe jest zawieszenie działania przez wywołanie statyczne `Thread.sleep(czasCzekania)`.

Alternatywnym rozwiązaniem utworzenia nowego wątku jest zdefiniowanie klasy, która implementuje interfejs `Runnable`. Następnie należy utworzyć obiektu typu `Thread` podając w konstruktorze obiekt zdefiniowanej klasy:

```
public class HelloRunnable implements Runnable {
   public void run() {
      System.out.println("Napis z watku utworzonego ");
   }
}
...
(new Thread(new HelloRunnable())).start();
...
```

9.3 Dostęp do zmiennych współdzielonych

Wykonywane wątki posiadają dostęp do współdzielonych zmiennych, jednak dostęp ten powinien być zgodny z obiektowym podejściem do programowania. Jednym ze sposobów współdzielenia zmiennych jest przekazanie obiektu (jako zmiennej współdzielonej) w konstruktorze. W tym rozwiązaniu należy zaimplementować odpowiedni konstruktor w klasie wątku, który przyjmie jako parametr zmienną współdzieloną i zachowa ją w zmiennej lokalnej.

```
public class MySharedVariable {
   private int counter = 0;
   public void increment(){
      counter ++;
   }

   public void decrement(){
      counter --;
   }

   public int getCounter(){
      return counter;
   }
}

public class HelloThread extends Thread {
   MySharedVariable mySharedVariable = null;
   public HelloThread(MySharedVariable mySharedVariable) {
      this.mySharedVariable = mySharedVariable;
   }

   public void run() {
      mySharedVariable.increment();
      System.out.println("Wartosc licznik = " +
      mySharedVariable.getCounter());
   }
}
```

W metodzie `main` należy utworzyć obiekt współdzielony oraz przekazać go w konstruktorach obiektów wątków. W tej wersji programu nie zapewniono wzajemnego wykluczania, co może spowodować błędy związane z wyścigami podczas dostępu do zmiennej współdzielonej. Ich rozwiązanie zostanie omówione w następnym punkcie.

```
public static void main(String args[]) {
   MySharedVariable mySharedVariable = new MySharedVariable();
   HelloThread helloThread1 = new HelloThread(mySharedVariable);
   HelloThread helloThread2 = new HelloThread(mySharedVariable);
   helloThread1.start();
   helloThread2.start();
   helloThread1.join();
...
```

Innym sposobem dostępu do zmiennych współdzielonych jest użycie wzorca singleton i wykorzystanie zmiennych statycznych. Przykład klasy implementującej taki wzorzec:

```
public class MySharedVariableSingleton {
   private static MySharedVariableSingleton instance = null;
   public static MySharedVariableSingleton getInstance() {

   if(instance==null)
      instance = new MySharedVariableSingleton();
   return instance;
}

public void increment()

...
```

Metoda `run()` wątku uzyskuje dostęp do zmiennej przez następujące wywołanie:

```
MySharedVariableSingleton sharedVariable = MySharedVariableSingleton.getInstance
    ↪ ();
```

9.4 Mechanizmy synchronizacji

W dotychczasowym przykładzie nie były stosowane mechanizmy wzajemnego wykluczania wątków. W Javie zdefiniowano słowo kluczowe synchronized dla określenia sekcji krytycznych. W kontekście tego ćwiczenia używane jest słowo synchronizacja do opisania sekcji krytycznych, które zapewnią wzajemne wykluczanie. Trzeba jednak pamiętać, że ogólne pojęcie synchronizacja odnosi się również do zapewnienia kolejności wykonania, a nie jedynie wzajemnego wykluczania.

Problem synchronizacji może ilustrować następujący przykład. Załóżmy, że dwa wątki (wątek A i wątek B) uzyskują dostęp do zmiennej współdzielonej MySharedVariable.counter. Jeden z nich zwiększa o 1 wartość zmiennej `licznik`, drugi zmniejsza o jeden. Oczekiwany końcowy wynik to 0, jednak w wyniku niewłaściwego wzajemnego wykluczania wątków wynik może być inny. Faktyczne wykonanie operacji `licznik++` przez jeden wątek przebiega w następujących krokach:

1. pobierz `licznik`
2. dodaj 1 do `licznik`
3. zapisz `licznik`

W przypadku przeplatania się akcji wątku A i B wykonanie może wyglądać następująco:

1. Wątek A: pobierz `licznik`
2. Wątek B: pobierz `licznik`
3. Wątek A: zwiększ `licznik` o 1
4. Wątek B: zmniejsz `licznik` o 1
5. Wątek A: zapisz `licznik` - `licznik` wynosi 1
6. Wątek B: zapisz `licznik` - `licznik` wynosi -1

Wynik działania wątku A jest utracony i nadpisany przez wątek B. Istnieje wiele możliwych kolejności wykonania i wyników, co powoduje, że wykrycie błędów synchronizacji może być trudne.

W języku Java dostępny jest mechanizm synchronizacji na dwóch głównych poziomach: synchronizowane metody lub wyrażenia. Zdefiniowanie metody jako synchronizowanej odbywa się przez dodanie słowa kluczowego `synchronized` do deklaracji, np.:

```
public synchronized void decrement(){
   counter --;
}
```

Taka deklaracja powoduje, że wykonanie metody w ramach jednego obiektu może odbywać się przez jeden wątek. Jeżeli jakiś wątek wykonuje metodę, to inne wątki, które chcą wykonać tę metodę zostają zablokowane do czasu zakończenia wykonania metody. Jeżeli w jednej klasie istnieje kilka metod określonych jako `synchronized`, to wszystkie one tworzą jedną sekcję krytyczną i w danym momencie może wykonywać się tylko jedna z nich. Należy pamiętać, że synchronizacja taka jak pokazano odnosi się do pojedynczego obiektu. Jeżeli utworzono wiele obiektów tej samej klasy, to każdy obiekt będzie posiadał własną sekcję krytyczną.
Uwaga! Powszechnym błędem jest używanie metod typu `synchronized` w klasie wątku. Metody `synchronized` powinny być definiowane w klasie współdzielonego zasobu. W typowych przypadkach wykorzystywanie metod `synchronized` w klasie wątku jest niepoprawne i nie gwarantuje wzajemnego wykluczania między wątkami.

Drugim sposobem synchronizacji jest synchronizacja na poziomie bloków kodu. W tym przypadku synchronizacja (sekcja krytyczna) wymaga podania obiektu, na którym jest uzyskiwana:

```
public void addName(String name) {
   synchronized(this) {
      lastName = name;
      nameCount++;
      nameList.add(name);
   }
...
}
```

Synchronizacja z użyciem (`this`) wykorzystuje domyślny obiekt synchronizacji, który jest związany z każdym obiektem istniejącym w aplikacji. Innym przykładem sekcji krytycznej jest synchronizacja z użyciem innego obiektu niż bieżący.

```
private Object lock1 = new Object();
private Object lock2 = new Object();

public void inc1() {
   synchronized(lock1) {
      c1++;
   }
}

public void inc2() {
   synchronized(lock2) {
   c2++;
}
}
```

9.5 Zakleszczenie - przykład

Zakleszczenie (deadlock) określa sytuację, w której dwa lub więcej wątków jest na zawsze zablokowanych, oczekując na siebie nawzajem. Rozważmy przykładowy scenariusz zakleszczenia. Wątek A oraz wątek B chcą uzyskać dostęp do sekcji krytycznych chronionych przez semafory (obiekty synchronizacji) `X` oraz `Y`. Scenariusz wygląda następująco:

1. Wątek A wchodzi do `synchronized (X)`
2. Wątek B wchodzi do `synchronized (Y)`
3. Wątek A chce wejść do `synchronized (Y)` //zablokowany, czeka na B
4. Wątek B chce wejść do `synchronized (X)` //zablokowany, czeka na A

Poniżej przedstawiono kod innego przykładu, w którym dwa wątki wykonują wzajemnie metody synchronizowane`ping` oraz `pingBack`. Jeżeli oba wątki wykonają ping przed wykonaniem `pingBack`, dojdzie do zakleszczenia.

```
public class PingPong {
   private final String name;
```

```
   public PingPong(String name) {
      this.name = name;
   }

   public String getName() {
      return this.name;
   }

   public synchronized void ping(PingPong ping) {
      System.out.format("%s: %s wyslal pinga!%n",
      this.name, ping.getName());
      ping.pingPong(this);
   }

   public synchronized void pingPong(PingPong pong) {
      System.out.format("%s: %s odpowiedzial na pinga!%n",
      this.name, pong.getName());
   }
}

public class DeadlockPing {
   public static void main(String[] args) {
      final PingPong pingPierwszy = new PingPong("Pierwszy");
      final PingPong pingDrugi = new PingPong("Drugi");
      new Thread(new Runnable() {
         public void run() { pingPierwszy.ping(pingDrugi); }
      }).start();

      new Thread(new Runnable() {
         public void run() { pingDrugi.ping(pingPierwszy); }
      }).start();
   }
}
```

Zadanie

Zaimplementować wielowątkowy program, który implementuje model producent- konsument. W programie dostępny jest magazyn, który posiada ograniczoną ilość miejsca i przechowuje "produkowane towary- typ i ilość. Działa wiele wątków producentów i konsumentów. Producenci zajmują się produkowaniem towarów (jeżeli w magazynie jest miejsce), losują ich typ i ilość. Konsumenci losują typ towaru i jego ilość, następnie odszukują w magazynie, czy mogą taki zakup wykonać. Jeżeli mogą, to dokonują zakupu w całości lub częściowo zmniejszając ilość towaru w magazynie. Wątki powinny wykonywać transakcje cyklicznie, a losowane opóźnienia i ilości powinny zapewnić, że program nie będzie zawieszał sięąlbo chodził w pustych pętlach. Należy zwrócić uwagę na synchronizację metod dostępu do magazynu w celu uniknięcia "wyścigów".

Ćwiczenie 10

MONITORY

10.1 Wstęp

Monitor jest owocem poszukiwań mechanizmów synchronizacji procesów, umożliwiających wbudowanie ich w struktury języka. Twórcami koncepcji monitora są *Brinch Hansen* i *C.A.R. Hoare*. Monitor składa się z wyróżnionych zmiennych, tzw. zmiennych warunkowych (conditional) oraz funkcji działających na tych zmiennych. Dostęp do danych przechowywanych w monitorze możliwy jest tylko za pomocą wyróżnionych funkcji monitora. Podejście takie jest implementowane poprzez klasy w językach obiektowych, np. C++ lub JAVA.

Istotą wyróżnionych funkcji monitora jest to, że wykonanie dowolnej w tych funkcji jest sekcją krytyczną. Oznacza to, że w danej chwili maksymalnie tylko jeden proces może wykonywać funkcję monitora. Ponadto, za pomocą zmiennych warunkowych istnieje możliwość wstrzymywania i wznawiania procesów wewnątrz wyróżnionych funkcji monitora. Zmienne te reprezentują pewne warunki, które powinny być spełnione, aby proces mógł kontynuować działanie. Na zmiennych warunkowych zdefiniowano dwie standardowe funkcje:

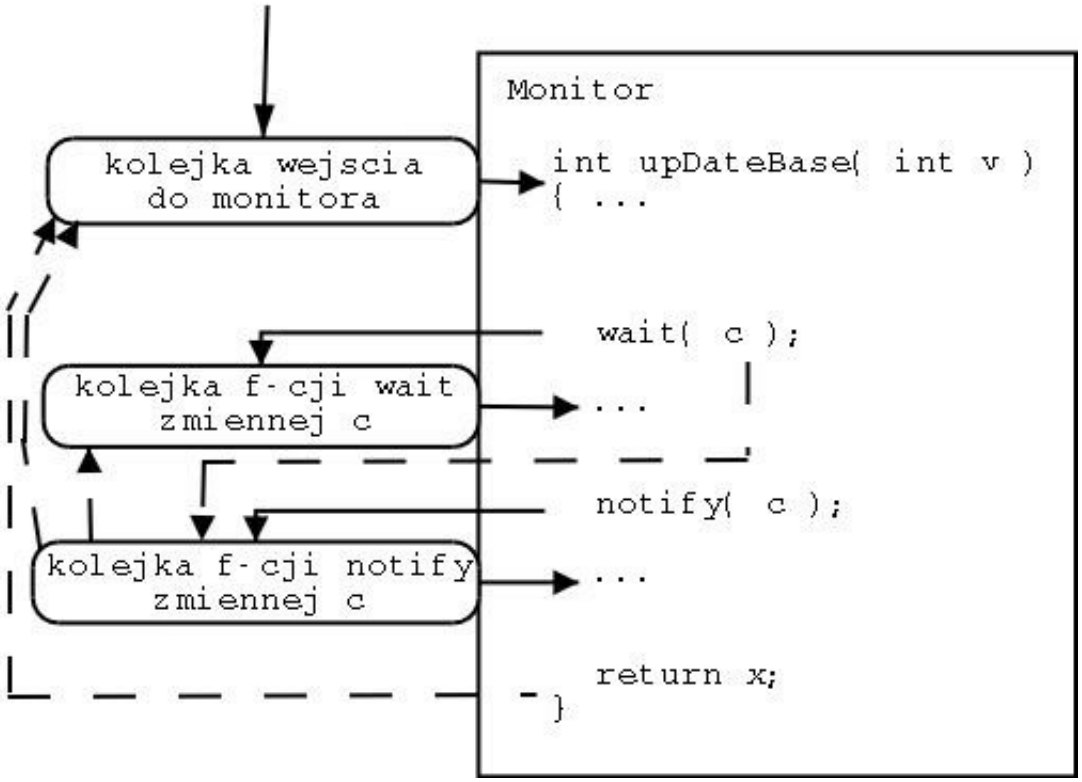
- `wait(c)` - służy do wstrzymania procesu na warunku reprezentowanym przez zmienną `c`; gdy proces jest wstrzymany, wstawiany jest na koniec kolejki procesów związanych z tą zmienną i zostaje odblokowany dostęp do monitora,
- `notify(c)` - służy do wznowienia pierwszego procesu wstrzymanego za pomocą funkcji wait na zmiennej warunkowej `c`; funkcja ta zwykle znajduje się jako ostatnia w funkcji monitora, wówczas proces wykonujący `notify` opuszcza sekcję krytyczną, do której ponownie wchodzi proces wznowiony po `wait(c)`; jeżeli `notify` nie jest ostatnią operacją w funkcji monitora, proces zostaje wstrzymany i umieszczony na stosie procesów wstrzymanych na operacji `notify`.

Tak przyjęty model powoduje, że z monitorem skojarzona jest jedna kolejka procesów oczekujących na wejście do monitora i po dwie kolejki dla każdej zmiennej warunkowej. Przedstawiono to na rysunku 10.1.

Należy pamiętać, że zmienna warunkowa reprezentuje warunek, który musi być spełniony, aby proces mógł kontynuować działanie.

10.2 Implementacja monitorów w języku JAVA

Język JAVA został wyposażony standardowo w monitory. Ponieważ język JAVA jest językiem ściśle obiektowym, istnieje mechanizm dziedziczenia klas. Każda klasa musi dziedziczyć nadrzędną klasę języka: `Object`. Klasa `Object` z kolei, zawiera metody (odpowiedniki funkcji) działające na jednej domyślnej zmiennej warunkowej: `wait` i `notify`. Ponadto każda metoda może posiadać modyfikator `synchronized`, który powoduje, że jej treść jest tożsama z sekcją krytyczną. Modyfikator `synchronized` może być użyty także dla oznaczenia bloku jako sekcji krytycznej. Załóżmy, że chcemy zaimplementować kolejkę fifo, która będzie dostępna przez wiele wątków. Klasa reprezentująca kolejkę fifo posiada dwie funkcję: odczytaj oraz zapisz. Obie funkcje wykonują operacje w sekcji krytycznej. Ponadto chcemy, aby funkcja odczytaj była blokująca: wstrzymywała wykonanie wątku, jeżeli kolejka jest pusta, do czasu, aż inny wątek wykona funkcję dodaj. W tym celu wykorzystamy zmienną warunkową, przy czym warunkiem będzie fakt, że kolejka jest niepusta. Poniższy kod przedstawia kompletny program demonstrujący implementację kolejki fifo.



Rysunek 10.1: Organizacja kolejek w monitorze

```
// element kolejki
class Element
{
   public Element nast = null;
   public String dana;
}

// implementacja kolejki fifo
class Kolejka_fifo
{
   Element pocz = null, koniec = null;
   public synchronized void dodaj(String s)
   { // to juz sekcja krytyczna
      Element e = new Element();
      e.dana = s;

      // dodanie elementu do kolejki
      if(koniec == null) // kolejka pusta
         pocz = e;
      else
         koniec.nast = e;

      e.nast = null;

      // poinformuj czekajacy watek
      notify();
   }

   public synchronized String odczytaj()
   {
      try
      { // warunek zmiennej warunkowej
```

```
         while(pocz == null)
            wait();
      }
      catch(InterruptedException e)
      {
         return null;
      }

      // element do zwrotu
      Element ret = pocz;

      // usuniecie z kolejki
      pocz = pocz.nast;

      // kolejka pusta
      if(pocz == null)

      koniec = null;
      return ret.dana;
   }
}

// Watek korzystajacy z fifo
class Fifo_watek extends Thread
{
   Kolejka_fifo f;

   // konstruktor
   Fifo_watek(Kolejka_fifo _f)
   {
      super();
      f = _f;
   }

   // funkcja watku
   public void run()
   {
      System.out.println("Nowy watek");
      // czytaj z kolejki
      System.out.println("Watek odczytal " + f.odczytaj());
      System.out.println("Watek konczy");
   }
}

// jedyna klasa widoczna na zewnatrz
public class Fifo
{

   // poczatek programu
   public static void main(String arg[]) throws Exception
   {
      System.out.println("FIFO");

      // utworz kolejke
      Kolejka_fifo f = new Kolejka_fifo();

      // utworz i uruchom watek
      Fifo_watek w = new Fifo_watek(f);
      w.start();

      // czekaj 5 sek
      Thread.sleep(5000);
```

```
        // zapisz do kolejki
        System.out.println("Pisze do kolejki");

        f.dodaj("Hello world");

        // czekaj 1 sek
        Thread.sleep(1000);

        System.out.println("Koniec");
    }
}
```

Powyższy kod należy umieścić w pliku o nazwie `Fifo.java` (zwrócić szczególną uwagę na pisownię w systemach rozróżniających wielkie i małe litery w nazwach plików!). Kompilacja pliku wykonywana jest za pomocą polecenia:

```
$ javac Fifo.java
```

Kompilator stworzy kilka plików z rozszerzeniem `class`, po jednym dla każdej klasy występującej w pliku `Fifo.java`. Pliki te są skompilowane do kodu maszyny wirtualnej Javy (JVM) i mogą być uruchamiane bez żadnych przeróbek na wszystkich innych systemach posiadających zainstalowaną maszynę wirtualną Javy. Aby uruchomić program należy użyć polecenia:

```
$ java -cp . Fifo
```

W starszych wersjach JAVy opcję `-cp` należy zastąpić przez `-classpath`.

10.2.1 Komunikaty i kanały

Jednym z mechanizmów komunikacji międzyprocesowej (IPC - Inter Process Communication) w systemie UNIX są kanały. Dane, o strukturze ustalonej przez użytkownika, przekazywane za pomocą kanałów nazywamy komunikatami. Kanał składa się z dowolnej liczby niezależnych podkanałów. Komunikacja za pomocą kanałów jest asynchroniczna (za wyjątkiem kanałów o zerowej pojemności) i dwukierunkowa. Kanały funkcjonują tak jak kolejki FIFO - pierwszy komunikat umieszczony w kanale zostanie jako pierwszy odczytany. Kanał jest identyfikowany za pomocą liczby typu `int`; komunikujące się procesy muszą ustalić unikalny identyfikator kanału, który chcą wykorzystać. Funkcje operujące na kanałach wykorzystują stałe, struktury i deklaracje znajdujące się w plikach: `sys/types.h`, `sys/ipc.h`, `sys/msg.h`. Pierwszą prezentowana przez nas funkcja służy do uzyskania dostępu do kanału. Jest ona w postaci:

```
int msgget(long kanal, int flagi);
```

Flagi określają sposób wykonania funkcji oraz prawa dostępu do kanału. Najczęściej używa się flagi `IPC_CREAT` (tworzenie kanału) z sumą arytmetyczną praw dostępu `0666` (prawa odczytu i zapisu dla wszystkich).

Jeżeli kanał nie istnieje, zostaje utworzony i funkcja zwraca identyfikator dostępu; jeżeli kanał istnieje, możliwe są dwa przypadki:

- proces ma prawo dostępu do kanału: funkcja zwraca identyfikator dostępu,
- proces nie ma prawa dostępu do kanału: wykonanie funkcji kończy się błędem i zwrócona wartość to -1.

Zapis do kanału wykonywany jest za pomocą funkcji:

```
int msgsnd(int kanal, struct msgbuf *komunikat, int wielk, int flagi);
```

Drugi parametr wskazuje na strukturę zawierającą identyfikator podkanału oraz treść komunikatu do wysłania i jest w postaci:

```
struct msgbuf{
   long podkanal;
   // tutaj wstaw tresc komunikatu
};
```

Widzimy, że aby w praktyce przekazywać komunikaty, należy stworzyć własną strukturę podobną do `msgbuf` i przekazywać jej wskaźnik rzutowany na wskaźnik do struktury `msgbuf`:

```
struct m_komunikat
{
   long podkanal;
   char linia1[200];
   int index;
} komunikat;

...
msgsnd(KANAL, (struct msgbuf *)&komunikat,
sizeof(komunikat), FLAGI);
...
```

W praktyce wykorzystujemy jedną flagę: `IPC_NOWAIT`, która oznacza, że funkcja `msgsnd` nie jest blokująca (możliwe jest blokowanie, gdy nie ma miejsca w kanale -por. problem producentów i konsumentów). Prawidłowe zakończenie funkcji powoduje, że zwraca ona wartość 0. W przeciwnym wypadku, gdy nastąpił błąd, wartość -1.

Odczyt z kanału wykonujemy za pomocą funkcji:

```
int msgrcv(int kanal, struct msgbuf *komunikat, int wielk, long podkanal, int
    ↪ flagi);
```

Parametry tej funkcji zasadniczo nie różnią się od parametrów `msgsnd`, jedynie `msgrcv` jest bogatsza o parametr określający podkanał, z którego ma zostać odczytany komunikat. Komunikat jest zapisywany do struktury wskazywanej przez drugi parametr o zadeklarowanej wielkości podanej jako trzeci parametr. Funkcja `msgrcv` może być blokująca (flagi = 0) albo nieblokująca (flagi = `IPC_NOWAIT`). Blokowanie funkcji może mieć miejsce jedynie, gdy podany podkanał jest pusty. Użytkownik może w ograniczonym zakresie modyfikować pojemność kanału. Służy do tego funkcja:

```
int msgctl(int kanal, int polecenie, struct msqid_ds *parametry);
```

Parametr drugi określa jaka operacja kontrolna ma zostać wykonana. W przypadku odczytu pojemności kanału należy podać `IPC_STAT`, zaś ustalenia pojemności `IPC_SET`. Struktura `msqid_ds` zawiera pole `long msg_qbytes`, które określa pojemność kanału (ustalaną lub odczytaną). Pojemność dotyczy całego kanału, to znaczy maksymalnej sumy objętości komunikatów we wszystkich podkanałach. Wielkości te podajemy w bajtach.

Zadanie

Zaimplementować semafor binarny za pomocą kanałów i komunikatów.

10.2.2 Implementacja monitora za pomocą kanałów i komunikatów

Do prawidłowej pracy monitora należy zdefiniować następujące funkcje:

- blokada wejścia do monitora: `wejście` - wołana na początku każdej funkcji monitora
- odblokowanie wejścia do monitora: `wyjście` - wołana na końcu każdej funkcji monitora
- wait - funkcja `wait` monitora dla zmiennej warunkowej
- notify - funkcja `notify` monitora dla zmiennej warunkowej.

Blokada wejścia powinna być wołana na początku każdej funkcji monitora. Na końcu tej funkcji powinna być wywołana funkcja odblokowania lub `notify` (p. uwaga poniżej).

Przyjmujemy ponadto następujące uproszczenia:

- w monitorze istnieje tylko jedna zmienna warunkowa
- funkcja `notify` jest wykonywana jako ostatnia w funkcji monitora; zakładamy więc, że nie należy wywoływać po niej funkcji wyjście
- dwie kolejki reprezentowane przez oddzielne dwa podkanały: pierwszy - kolejka procesów oczekujących na wejście do monitora (związanych z funkcją `wejście`); druga kolejka procesów wstrzymanych na zmiennej warunkowej (funkcja `wait`).

Uwaga! do prawidłowego działania monitora, potrzebna jest znajomość liczby procesów wstrzymanych na zmiennej warunkowej. Może ona stanowić treść komunikatu blokującego na zmiennej warunkowej.

Zadanie

Zaimplementować zdefiniowany powyżej monitor.