9. Wątki i programowanie wielowątkowe

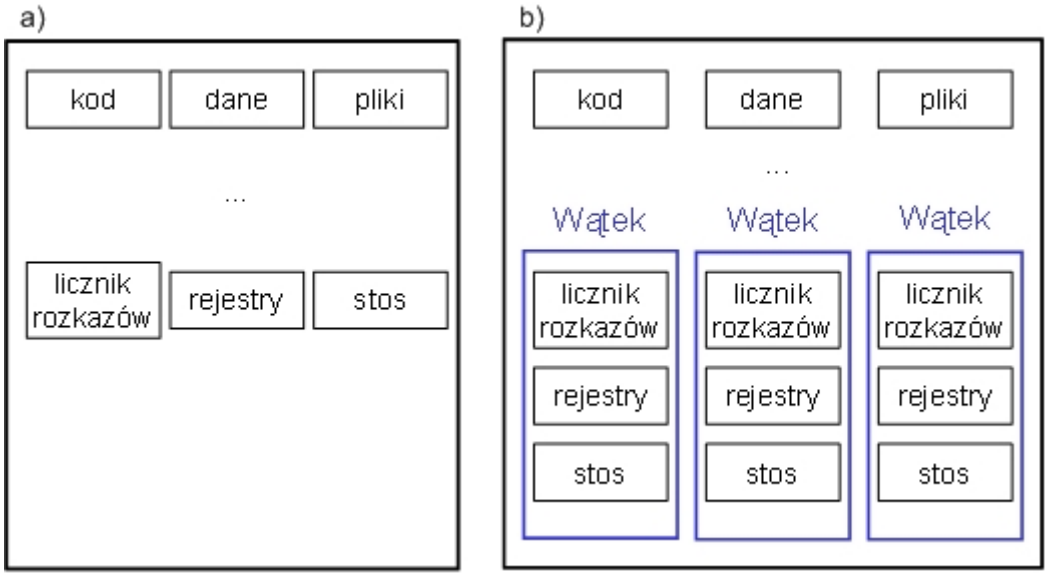
Wstęp

Wykład 9 poświęcamy omówieniu idei wielowątkowości we współczesnych systemach operacyjnych. Wprowadzamy pojęcie wątku i procesu wielowątkowego oraz rozróżnienie pomiędzy wątkami użytkownika a wątkami jądra. Następnie przedstawiamy różne modele implementacji wątków. Omawiamy również podstawy programowania wielowątkowego w systemach uniksowych, prezentując interfejs funkcji systemowych realizujących podstawowe operacje na wątkach.

9.1. Podstawowe zagadnienia

Wątki i wielowątkowość

Wątek to sekwencyjny przepływ sterowania w programie. Jeden program może być wykonywany jednocześnie przez jeden lub więcej wątków, przy czym rzeczywista współbieżność jest ograniczona przez liczbę procesorów. Grupa współpracujących wątków korzysta z tej samej przestrzeni adresowej i zasobów systemu operacyjnego. Wątki dzielą dostęp do wszystkich danych z wyjątkiem licznika rozkazów, rejestrów procesora i stosu, co ilustruje Rys. 9.1. Dzięki temu tworzenie i przełączanie wątków odbywa się znacznie szybciej niż tworzenie i przełączanie kontekstu procesów tradycyjnych.



Rys. 9.1 Ilustracja wielowątkowości: a) proces jednowątkowy, b) proces wielowątkowy

Wątki użytkownika a wątki jądra

Wątki użytkownika realizowane są przez funkcje biblioteczne w ramach procesu i nie są widoczne dla jądra. Jądro systemu obsługuje tylko tradycyjne procesy. Przełączanie kontekstu między wątkami powinno być zrealizowane przez funkcje biblioteczne w procesie na poziomie użytkownika. Z tego względu zarówno tworzenie, jak i przełączanie wątków może odbywać się bardzo szybko. Główną wadą jest jednak brak możliwości równoległego przetwarzania wątków w systemie wieloprocesorowym, ponieważ nie są one widoczne dla jądra systemu. Ponadto, jeden wątek może zablokować cały proces.

Wątki jądra realizowane są przez jądro systemu. Jądro zajmuje się planowaniem wątków i przełączaniem kontekstu podobnie jak dla procesów tradycyjnych. Konieczne jest przy tym przełączanie procesora w tryb jądra, co powoduje wydłużenie czasu tworzenia i przełączania wątków. Wątki jądra mogą być wykonywane równolegle w systemie wieloprocesorowym.

Wątki mieszane łączą cechy obydwu realizacji, tworząc dwupoziomowy lub wielopoziomowy system wątków. Jądro tworzy i zarządza wątkami jądra i procesami lekkimi, które wspierają wykonywanie wątków użytkownika.

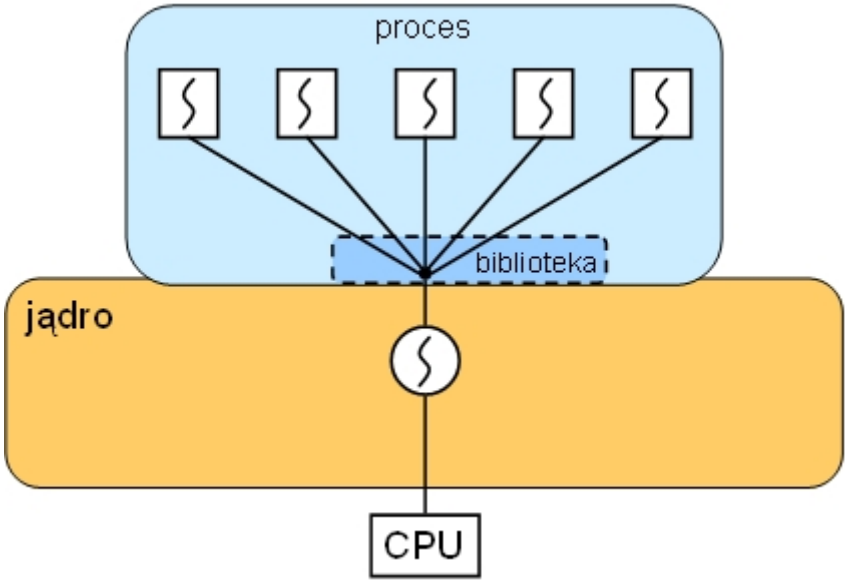
9.2. Modele implementacji wątków

Istnieją trzy modele implementacji wątków, określające zależność między wątkami użytkownika a wątkami jądra:

1. model m-1,

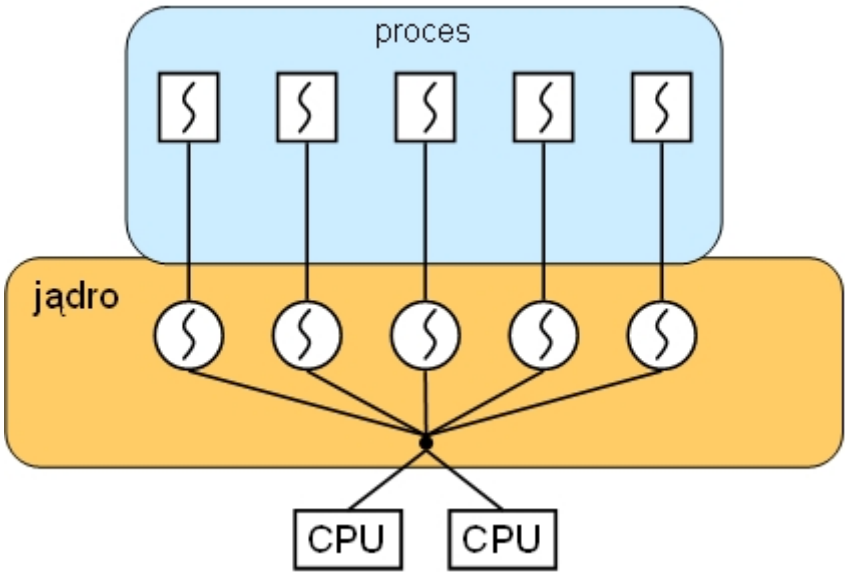
2. model 1-1,
3. model m-n.

W modelu m-1 wiele wątków użytkownika odwzorowanych jest na 1 wątek jądra (Rys. 9.2). Pojedynczy wątek jądra wspiera zatem wykonanie całego wielowątkowego procesu użytkownika. Szeregowanie wątków odbywa się tylko na poziomie biblioteki w procesie. Na poziomie jądra szeregowane są całe procesy.



Rys. 9.2 Model m-1

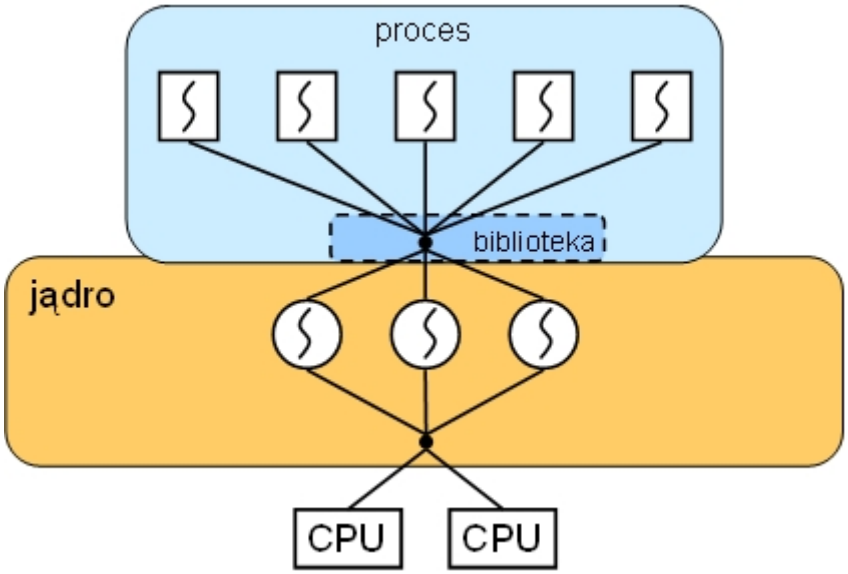
Model 1-1 odwzorowuje każdy wątek użytkownika na 1 wątek jądra (Rys. 9.3). Szeregowanie wątków odbywa się na poziomie jądra systemu operacyjnego. Dzięki temu wątki użytkownika w procesie mogą być wykonywane równolegle na wielu procesorach.



Rys. 9.3 Model 1-1

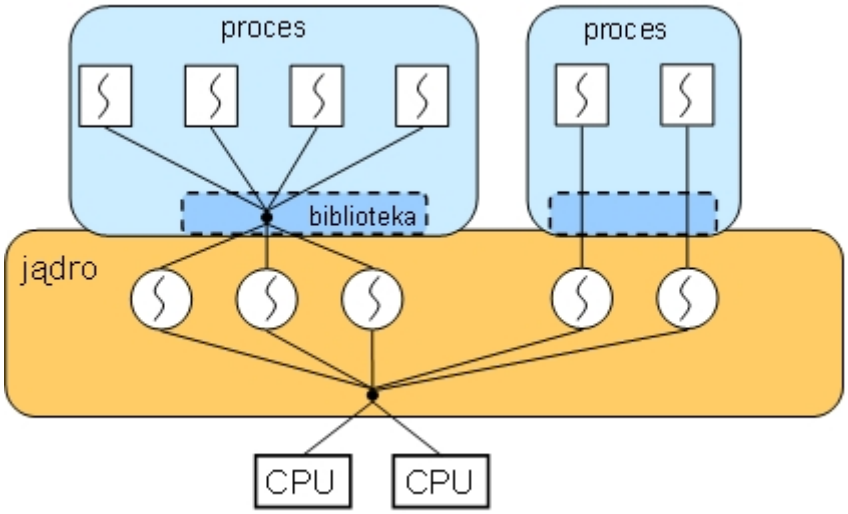
Model m-n przewiduje utworzenie puli wątków jądra do wspierania wykonania grupy wątków użytkownika w procesie (Rys. 9.4). Stwarza to możliwość równoległego przetwarzania wątków procesu, ale stopień wieloprzetwarzania jest zwykle mniejszy niż przy modelu 1-1. Szeregowanie odby-

wa się na dwóch poziomach: w procesie szeregowane są wątki użytkownika, a jądro szereguje wątki jądra.



Rys. 9.4 Model m-n

Możliwy jest również model mieszany (Rys. 9.5), stanowiący połączenie modeli m-n i 1-1. W takim przypadku wybrane wątki użytkownika mogą otrzymać na wyłączność wsparcie pojedynczych wątków jądra. Pozostałe wątki użytkownika korzystają z puli wątków jądra przeznaczonych do obsługi procesu.



Rys. 9.5 Model mieszany

Wątki w systemie Linux

W systemie Linux zaimplementowano model 1-1. Jądro systemu stosuje ten sam mechanizm do tworzenia procesów i wątków jądra. Określając, które zasoby procesu macierzystego mają być współdzielone z procesem potomnym, decyduje o utworzeniu wątku lub tradycyjnego procesu.

Funkcja **clone()** tworzy nowy proces tradycyjny lub wątek, a jej argumenty określają zakres współdzielenia zasobów.

```
int clone(int (*fn)(void *), void *child_stack, int flags, void *arg);
```

gdzie:

fn - funkcja wykonywana przez nowy proces potomny lub wątek,

child_stack - adres stosu procesu potomnego lub wątku,

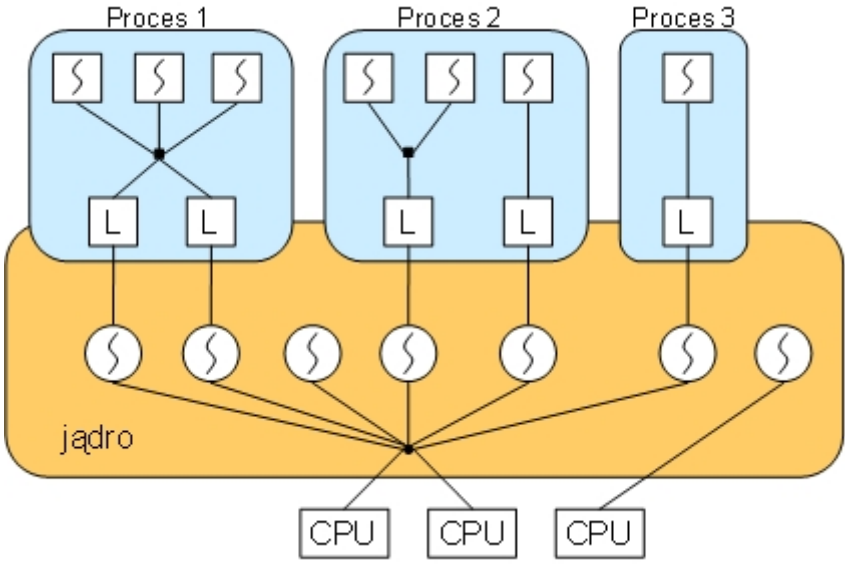
flags - flagi określające zakres współdzielenia zasobów,

arg - argument funkcji **fn**.

Funkcja **clone**() jest specyficzna dla systemu Linux i w związku z tym nie jest zalecana ze względu na przenośność programów. W programach wielowątkowych należy posługiwać się funkcjami z biblioteki wątków.

Wątki w systemie UNIX (Solaris)

Wielopoziomowy system wątków został zrealizowany w systemie operacyjnym Solaris. Jądro obsługuje wyłącznie wątki jądra, które mogą być szeregowane do poszczególnych procesorów. Niektóre wątki jądra realizują bezpośrednio różne usługi systemowe. Pozostałe są związane z procesami lekkimi LWP. Wątki użytkownika realizowane są w ramach procesów. Ich powiązanie z wątkami jądra następuje poprzez procesy lekkie LWP. Każdy proces tradycyjny (ciężki), zawierający dowolną liczbę wątków, jest obsługiwany przez co najmniej jeden proces lekki LWP. Wybrane wątki użytkownika można związać na stałe z konkretnymi procesami LWP. Pozostałe tworzą pulę wątków niezwiązanych, obsługiwanych przez jeden lub więcej LWP. Począwszy jednak od wersji Solaris 8 (inaczej Solaris 2.8) model mieszany był stopniowo zastępowany modelem 1-1 (każdy wątek użytkownika związany z LWP), który okazał się bardziej wydajny. W wersjach Solaris 9 i 10 zaimplementowano już tylko model 1-1.



Rys. 9.5 Model implementacji wątków w systemie Solaris

9.3. Podstawowe operacje na wątkach

Najpopularniejszy obecnie standard POSIX IEEE Std 1003.1:1996 i IEEE Std 1003.1:2001 opisuje interfejs programowania wielowątkowego w systemach uniksowych. Biblioteka funkcji zgodna z tym standardem nosi nazwę **libpthread** a funkcje z biblioteki mają przedrostek **pthred**_.

Tworząc program wielowątkowy, należy dodatkowo umieścić w kodzie dyrektywę włączającą plik nagłówkowy:

```
#include <pthread.h>
```

Wywołanie kompilatora powinno wyglądać następująco:

```
gcc [opcje... ] plik ... –pthread ...
```

Opcja **-pthread** włącza obsługę przetwarzania wielowątkowego poprzez ustawienie odpowiednich flag i makrodefinicji dla prekompilatora i konsolidatora.

Tworzenie wątków

Nowy wątek powstaje w wyniku wywołania funkcji:

```
int pthread_create(pthread_t * thread, pthread_attr_t * attr, void *
(*start_func)(void *), void * arg);
```

Funkcja tworzy nowy wątek z domyślnym zestawem atrybutów i zwraca jego identyfikator przez **thread.** Utworzony wątek działa współbieżnie z wątkiem bieżącym (wołającym funkcję), wykonując podaną funkcję **start_func**() z argumentem **arg**. Najważniejsze atrybuty wątków zestawiono w Tabl. 9.1.

Tablica 9.1 Atrybuty wątków

Atrybut	Wartości	Znaczenie
scope	PTHREAD_SCOPE_PROCESS	wątek szeregowany na poziomie procesu, niezwiązany na stałe z procesem lekkim LWP (ang. unbound)
	PTHREAD_SCOPE_SYSTEM	wątek szeregowany na poziomie jądra, związany na stałe z procesem lekkim LWP (ang. bound)
detachstate	PTHREAD_CREATE_JOINABLE (domyślnie)	wątek dołączalny, inne wątki mogą oczekiwać na jego zakończenie i odebrać status (synchronizować się)
	PTHREAD_CREATE_DETACHED	wątek odłączony, inne wątki nie mogą oczekiwać na jego zakończenie (synchronizować się)
stackaddr	NULL (domyślnie)	adres początkowy stosu (domyślnie: stos alokowany przez system)
stacksize	NULL (domyślnie)	rozmiar stosu (domyślnie: 1 MB)
schedpolicy	SCHED_OTHER (domyślnie)	planowanie priorytetowe
	SCHED_FIFO	planowanie FCFS
	SCHED_RR	planowanie rotacyjne RR
schedparam		parametry szeregowania wątku
inheritsched	PTHREAD_EXPLICIT_SCHED	parametry szeregowania ustawiane na podstawie argumentów wywołania funkcji pthread_create()
	PTHREAD_INHERIT_SCHED	parametry szeregowania dziedziczone po procesie macierzystym

Modyfikację atrybutów umożliwiają funkcje:

```
int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_destroy(pthread_attr_t *attr);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *poli-
cy);

int pthread_attr_setschedparam(pthread_attr_t *attr, const struct
sched_param *param);

int pthread_attr_getschedparam(const pthread_attr_t *attr, struct
sched_param *param);
```

Przykład

Prosty przykład programu 2-wątkowego. Wątek główny uruchamia wątek roboczy, a następnie czeka na jego zakończenie i drukuje wynik. Wątek roboczy wykonuje obliczenia.

```
/* Silberschatz, Galvin and Gagne: Operating System Concepts with Java,
2003 */
#define _REENTRANT    /* basic 3-lines for threads */
#include <pthread.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* funkcja wątku */

main(int argc, char *argv[])
{
        pthread_t tid; /* identyfikator wątku */
        pthread_attr_t attr; /* zbiór atrybutów */

        /* pobranie domyślnych atrybutów */
        pthread_attr_init(&attr);

        /* tworzenie wątku  */
        pthread_create(&tid,&attr,runner,argv[1]);

        /* oczekiwanie na zakończenie wątku */
        pthread_join(tid,NULL);
        printf("sum = %d\n",sum);
}

/* funkcja wątku roboczego */
void *runner(void *param) {
        int upper = atoi(param);
        int i;

        sum = 0;
        if (upper > 0) {
                for (i = 1; i <= upper; i++)
                sum += i;
        }
        pthread_exit(0);
}
```

Wątek może pobrać własny identyfikator za pomocą funkcji:

```
pthread_t pthread_self(void);
```

Kończenie i oczekiwanie na zakończenie wątków

Wątek zostaje zakończony w wyniku:

- powrotu z wykonywanej funkcji **start_func**(),
- wywołania funkcji **pthread_exit**(),
- zakończenia procesu poprzez **exit**().

```
void pthread_exit(void *retval);
```

W przypadku kończenia wątków odłączonych, zasoby pamięci są zwalniane natychmiast. W przeciwnym przypadku zasoby pamięci zostaną zwolnione po odebraniu statusu przez inny wątek.

Oczekiwanie na zakończenie wątka i odebranie jego statusu zakończenia umożliwia funkcja:

```
int pthread_join(pthread_t thread, void **thread_return);
```

która powoduje wstrzymanie wykonywania wołającego wątka do momentu zakończenia wątku wskazanego przez **thread**.

Wątki nie mogą oczekiwać na zakończenie wątków odłączonych. Można to uzyskać wywołując funkcję **pthread_detach**().

```
int pthread_detach(pthread_t thread);
```

Przykład

Wątek główny tworzy kilka wątków roboczych, a następnie oczekuje na ich zakończenie. Każdy wątek roboczy oblicza sumę ciągu liczb naturalnych do wartości podanej jako argument wywołania, a następnie kończy działanie zwracając wynik obliczeń.

```
#include <pthread.h>

#define N 10

void *runner(void *param);     /* funkcja watku roboczego */

main(int argc, char *argv[])
{
     pthread_t tid[N];    /* tablica identyfikatorow watkow */
     pthread_attr_t attr;   /* zbior atrybutow watkow */
     int i, *stat[N];

     if (argc > N+1) {
          printf("Nieprawidlowa liczba argumentow\n");
          exit(1);
     }

     pthread_attr_init(&attr);
```

```
/* tworzenie watkow roboczych - jeden watek na kazdy argument wywolania*/
     for (i=1; i<argc; i++) {
          if (pthread_create(&tid[i-1],&attr,runner,argv[i]))
               perror("watek");
     }

/* oczekiwanie na zakonczenie watkow roboczych */
     for (i=1; i<argc; i++) {
          pthread_join(tid[i-1],(void**)&(stat[i-1]));
          printf("pid=%d tid=%d ",getpid(), pthread_self());
          printf("sum(%d) = %d\n", atoi(argv[i]), *stat[i-1]);
     }
}

/* funkcja wątku roboczego */
void *runner(void *param) {
     int upper = atoi(param);
     int i;
     int sum;

     sum = 0;
     if (upper > 0) {
          for (i = 1; i <= upper; i++)
          sum += i;
     }
     printf("pid=%d tid=%d sum = %d\n",getpid(), pthread_self(), sum);
     pthread_exit(&sum);
}
```

9.4. Obsługa sygnałów w wątkach

Wszystkie wątki w jednym procesie współdzielą sposoby obsługi sygnałów, domyślne i ustawione przez **signal**() lub **sigaction**(). Każdy wątek może natomiast posiadać odrębną maskę blokowanych sygnałów. Sygnały spowodowane przez pułapki, takie jak SIGILL i SIGSEGV, są obsługiwane przez watek, który spowodował pułapkę. Sygnały spowodowane przez przerwania, czyli zdarzenia zewnętrzne (SIGIO, SIGINT itp.) są obsługiwane przez jeden dowolny wątek, którego maska na to pozwala. Następny sygnał, nadesłany w trakcie obsługi poprzedniego, zostanie obsłużony przez następny wątek.

Wątek może wysłać sygnał do innego wątku w ramach tego samego procesu przy pomocy funkcji:

```
int pthread_kill(pthread_t thread, int sig);
```

Ustawienie własnej maski blokowanych sygnałów umożliwia funkcja:

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
```

Znaczenie argumentów jest identyczne jak w funkcji **sigprocmask**() opisanej w wykładzie 8.

Możliwe jest wstrzymanie wątku w oczekiwaniu na jeden z sygnałów z podanego zbioru:

```
int sigwait(sigset_t *set);

int sigwait(const sigset_t *set, int *sig);
```

Funkcja pobiera jeden niezałatwiony sygnał ze zbioru **set** lub wstrzymuje wątek w oczekiwaniu na sygnał z tego zbioru. Pobrany sygnał jest usuwany ze zbioru sygnałów niezałatwionych, a numer

sygnału jest zwracany jako wartość funkcji lub zapisywany w **sig**. Działanie funkcji jest niezależne od maski blokowanych sygnałów, czyli wątek może czekać synchronicznie na sygnał.

Uwzględniając powyższe, obsługę sygnałów w procesach wielowątkowych można zrealizować metodą asynchroniczny lub synchroniczny.

Asynchroniczna obsługa sygnałów przebiega w następujący sposób:

- dowolny wątek (zwykle główny) ustawia funkcją **sigaction**() sposób obsługi sygnałów w całym procesie,
- wszystkie wątki z wyjątkiem jednego ustawiają blokowanie sygnałów,
- jeden wybrany wątek nie ustawia maski blokowania i obsługuje wszystkie sygnały nadsyłane do procesu w funkcji obsługi ustawionej przez **sigaction**().

Synchroniczna obsługa sygnałów realizowana jest przez wybrany wątek bez udziału funkcji obsługi sygnałów:

- wszystkie wątki procesu blokują odbieranie sygnałów,
- jeden wybrany wątek pobiera kolejne niezałatwione sygnały wywołując cyklicznie funkcję **sigwait**() i obsługuje je bezpośrednio w swoim kodzie.

Bibliografia

[1] Silberschatz A., Galvin P.B.: Podstawy systemów operacyjnych, WNT 2000

[2] Rochkind M.J.: Programowanie w systemie UNIX dla zaawansowanych, WNT 2007 (rozdziały: 5, 9)

Słownik

Termin	Objaśnienie
wątek	sekwencyjny przepływ sterowania w programie
wątek jądra	wątek realizowany na poziomie jądra systemu, szeregowany i przełączany przez jądro
wątek użytkownika	wątek realizowany przez funkcje biblioteczne na poziomie procesu, niewidoczny dla jądra
model implementacji wątków	model określający zależność między wątkami użytkownika a wątkami jądra w systemie operacyjnym

Zadania do wykładu 9

Zadanie 1

Napisać wielowątkowy program **calka1**, który oblicza całkę wybranej funkcji na zadanym przedziale (pole powierzchni pod wykresem funkcji). Funkcja jest zapisana w kodzie programu. Program pobiera trzy argumenty określające granice przedziału zmienności funkcji oraz liczbę podprzedziałów. W każdym podprzedziale program przybliża całkę pojedynczym trapezem. Postać wywołania programu:

```
calka1  start  stop  N
```

Wątek główny dzieli przedział zmienności na **równe** podprzedziały i uruchamia nowy wątek roboczy do obliczenia całki w każdym z nich. Wątek roboczy oblicza całkę w podprzedziale, następnie wypisuje swój identyfikator i uzyskany wynik. Wątek główny sumuje wyniki z poszczególnych wątków i wypisuje na stdout.

Zadanie 2

W programie z zadania 1 dodać wątek, który będzie w sposób synchroniczny (przy pomocy funkcji **sigwait**()) obsługiwał sygnały SIGINT przysyłane do procesu. Po każdym sygnale wątek powinien wypisać komunikat na stout.