9
Wątki

1. Uruchom poniższy program tworzący pojedynczy wątek:

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

void* worker(void* info)
{
  int i;
  for(i=0; i<10; i++)
  {
    sleep(1);
    printf("thread\n");
  }
  return NULL;
}

int main()
{
  pthread_t th;
  int i;

  pthread_create(&th, NULL, worker, NULL);
  for(i=0; i<10; i++)
  {
    sleep(1);
    printf("main program\n");
  }
  pthread_join(th, NULL);
  return 0;
}
```

   Kompilację należy przeprowadzić zleceniem:

```
# cc -o thtest thtest.c -lpthread
```

2. Wyświetl listę procesów i sprawdź obecność wątków:

```
# ps x
# ps -L x
```

```
# ps -T x
```

Zwróć uwagę na kolumny PID, LWP (*light weight process*) i SPID.

3. Zmodyfikuj powyższy program tak, aby tworzył 2 wątki. Każdy wątek powinien odebrać liczbę jako swój parametr:

```
int d = 1;
pthread_create(&th, NULL, worker, &d);
```

4. Zweryfikuj działanie sekcji krytycznej:

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

pthread_mutex_t mx;

void* worker(void* info)
{
  int i;
  int tid = *(int*)info;
  for(i=0; i<10; i++)
  {
    printf("Thread %d is entering critical section...\n", tid);
    pthread_mutex_lock(&mx);
    printf("Thread %d is in critical section...\n", tid);
    sleep(5);
    pthread_mutex_unlock(&mx);
    printf("Thread %d is leaving critical section...\n", tid);
    sleep(2);
  }
  return NULL;
}

int main()
{
  pthread_t th1, th2, th3;
  int w1=1, w2=2, w3=3;

  pthread_mutex_init(&mx, NULL);

  pthread_create(&th1, NULL, worker, &w1);
  pthread_create(&th2, NULL, worker, &w2);
  pthread_create(&th3, NULL, worker, &w3);

  pthread_join(th1, NULL);
  pthread_join(th2, NULL);
  pthread_join(th3, NULL);

  return 0;
}
```

5. Odbierz dane wyjściowe z wątku podczas wywołania funkcji `pthread_join(3)`.
6. Zaimplementuj program przesyłający dane pomiędzy wątkiem-producentem a wątkiem-konsumentem.

9.1 Zarządzanie wątkami

Wątek może oczekiwać na zakończenie innego wątku funkcją:

```
int pthread_join(pthread_t th, void **retval);
```

Jest to funkcja analogiczna do funkcji `wait` oczekującej na zakończenie procesu potomnego. Wskaźnik `retval` zostanie zainicjowany wartością zwróconą przez wątek.

Wykonanie wątku można zakończyć w dowolnym momencie wywołaniem funkcji:

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

Wartość `retval` może być odczytana przez inny wątek, który zsynchronizuje się z nim wywołaniem `pthread_join`.

Wątek można zatrzymać z poziomu innego wątku funkcją `pthread_cancel`:

```
int pthread_cancel(pthread_t thread);
```

Zatrzymywanie wątku można blokować funkcją:

```
int pthread_setcancelstate(int state, int *oldstate);
```

Wątek może „odłączyć" się od procesu i kontynuować pracę niezależnie od wątku głównego. Służy do tego funkcja:

```
int pthread_detach(pthread_t th);
```

Uniezależnienie wątku oznacza, że jego zasoby będą zwolnione po jego zakończeniu. Z drugiej jednak strony nie będzie możliwe zsynchronizowanie z innym wątkiem poprzez wywołanie `pthread_join`.

9.2 Obsługa sygnałów

Wszystkie wątki współdzielą między sobą jedną tablicę z adresami procedur obsługi. Każdy wątek może zmieniać przypisaną wcześniej do sygnału procedurę standardową funkcją `signal`. Istnieje możliwość wysyłania sygnałów do poszczególnych wątków:

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

Wątki mogą blokować odbiór określonych sygnałów używając funkcji:

```
int pthread_sigmask(int how, const sigset_t *newmask,
                                    sigset_t *oldmask);
```

gdzie `newmask` jest maską sygnałów (4 liczby typu `unsigned long`) tworzoną z użyciem funkcji:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Wywołanie funkcji `pthread_sigmask` z argumentem drugim ustawionym na NULL powoduje odczytanie aktualnej maski sygnałów.

W ramach synchronizacji wątków można wymusić oczekiwanie na określony sygnał:

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

9.3 Synchronizacja wątków

Do synchronizacji wątków wykorzystywane są następujące mechanizmy: zamki (ang. *mutex*), zmienne warunkowe (ang. *conditional variable*) i semafory POSIX.

9.3.1 Zamki

Zamki są binarnymi semaforami, a więc mogą znajdować się w jednym z dwóch stanów: podniesiony lub opuszczony. Do inicjowania zamka wykorzystywana jest funkcja:

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       const pthread_mutexattr_t *mutexattr);
```

Poniższy fragment kodu tworzy zamek z domyślnymi atrybutami:

```
pthread_mutex_t  m;
...
pthread_mutex_init(&m, NULL);
```

Do wykonywania operacji na zamkach służą następujące funkcje:

`pthread_mutex_lock()`
: Zajęcie zamka. Funkcja jest blokująca do czasu aż operacja może zostać wykonana.

`pthread_mutex_trylock()`
: Zajęcie wolnego zamka. Próba zajęcia już zajętego zamka kończy się zasygnalizowaniem błędu (`EBUSY`).

`pthread_mutex_unlock()`
: Zwolnienie zamka. Zwolnienia powinien dokonać wątek, który zajął zamek.

`pthread_mutex_destroy()`
: Usunięcie zamka.

Wszystkie wymienione funkcje pobierają wskaźnik do struktury `pthread_mutex_t` jako jedyny argument i zwracają wartość typu `int`.

9.3.2 Zmienne warunkowe

Zmienne warunkowe pozwalają na kontrolowane zasypianie i budzenie procesów w momencie zajścia określonego zdarzenia.

Poniższy fragment kodu tworzy zmienną warunkową z domyślnymi atrybutami:

```
pthread_cond_t  c;
...
pthread_cond_init(&c, NULL);
```

Budzenie wątków realizowane jest z wykorzystaniem jednej z dwóch poniższych funkcji:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkcja `pthread_cond_signal()` budzi co najwyżej jeden wątek oczekujący na wskazanej zmiennej warunkowej. Funkcja `pthread_cond_broadcast()` budzi wszystkie wątki uśpione na wskazanej zmiennej warunkowej.

Kolejne dwie funkcje służą do usypiania wątku na zmiennej warunkowej:

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

```
int pthread_cond_timedwait(pthread_cond_t *cond,
                           pthread_mutex_t  *mutex,
                           const struct timespec *abstime);
```

Funkcja `pthread_cond_wait()` oczekuje bezwarunkowo do czasu odebrania sygnału budzącego, podczas gdy funkcja `pthread_cond_timedwait()` ogranicza maksymalny czas oczekiwania. Zaśnięcie wątku wymaga użycia jednocześnie zmiennej warunkowej i zamka. Przed zaśnięciem zamek musi być już zajęty. Zaśnięcie oznacza atomowe zwolnienie zamka i rozpoczęcie oczekiwania na sygnał budzący. Obudzenie wątku powoduje ponowne zajęcie zamka. Poniższy przykład pokazuje zastosowanie zmiennej warunkowej do synchronizacji wątków:

- wątek oczekujący

```
pthread_cond_t  c;
pthread_mutex_t  m;
...
pthread_mutex_lock(&m);       /* zajęcie zamka */
pthread_cond_wait(&c, &m);    /* oczekiwanie na zmiennej warunkowej */
pthread_mutex_unlock(&m);     /* zwolnienie zamka */
```

- wątek budzący

```
pthread_cond_signal(&c);      /* sygnalizacja zmiennej warunkowej */
```

Pomimo, że nie jest to wymagane, często do poprawnej synchronizacji wywołuje się funkcję budzącą podczas przetrzymywania zamka:

```
pthread_mutex_lock(&m);
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

9.3.3 Semafory

Do synchronizacji wątków można wykorzystać semafory standardu POSIX. Jest to rozwiązanie całkowicie niezależne od semaforów Systemu V będących częścią zestawu mechanizmów komunikacji międzyprocesowej IPC. Semafor POSIX jest licznikiem przyjmującym wartości od zera wzwyż. Do obsługi semaforów POSIX przewidziano następujące funkcje:

`sem_init()`
: Funkcja tworząca i inicjująca nowy semafor. Semafor może być strukturą wewnętrzną procesu lub może służyć do synchronizacji niezależnych procesów, podobnie jak semafory IPC[1]. Argumentem funkcji `sem_init()` jest początkowa wartość semafora.

`sem_wait()`
: Oczekiwanie na wartość semafora większą od zera i obniżenie jej o 1.

`sem_trywait()`
: Nieblokująca próba zmniejszenia wartości semafora o 1.

`sem_post()`
: Zwiększenie wartości semafora o 1. Operacja zawsze wykonuje się poprawnie i nie jest blokująca.

`sem_getvalue()`
: Pobranie aktualnej wartości semafora.

[1] Aktualna implementacja LinuxThreads nie wspiera semaforów synchronizujących niezależne procesy.

`sem_destroy()`
: Usunięcie semafora.

Przykład 9.1 prezentuje proste zastosowanie semaforów standardu POSIX. Jest to oczywiście zmodyfikowana wersja przykładu **??**.

Zadania

1. Przećwicz przekazywanie argumentów do wątku i odbiór wartości zwrotnych. Uruchom w tym celu 3 nowe wątki, które będą zwracały podwojoną wartość typu `int`.
2. Utwórz 2 wątki i wstrzymaj ich wykonanie zmienną warunkową. Wątek główny powinien wznowić ich pracę sygnalizując to odpowiedniej zmiennej warunkowej. Sprawdź różnicę pomiędzy funkcją `pthread_cond_signal()` a `pthread_cond_broadcast()`.
3. Zaproponuj implementację operacji `bariera()`, której działanie polegałoby na zsynchronizowaniu wskazanej liczby wątków. Operacja kończy się w momencie jej wywołania przez wszystkie wątki.
4. Zaproponuj synchronizację 2 wątków w problemie producenta-konsumenta. Przeanalizuj czy rozwiązanie to działa poprawnie również w przypadku większej liczby producentów i konsumentów.