Temat zajęć: Tworzenie i obsługa wątków.

Czas realizacji zajęć:	*180 min.*
Zakres materiału, jaki zostanie zrealizowany podczas zajęć:	Tworzenie wątków, przekazywanie parametrów do funkcji wątków i pobieranie wyników, synchronizacja wątków, implementacja przykładowych programów obsługi wątków

I. Wątki

Wątki określane są jako wydzielone sekwencje przewidzianych do wykonania instrukcji, które są realizowane w ramach jednego procesu. Każdy wątek ma swój własny stos, zestaw rejestrów, licznik programowy, indywidualne dane, zmienne lokalne, i informację o stanie. Wszystkie wątki danego procesu mają jednak tę samą przestrzeń adresową., ogólną obsługę sygnałów, pamięć wirtualną, dane oraz wejście-wyjście. W ramach procesu wielowątkowego każdy wątek wykonuje się oddzielnie i asynchronicznie.

Wszystkie programy korzystające z funkcji operujących na wątkach normy POSIX zawierają dyrektywę **#include <pthread.h>**. Kompilując przy użyciu **gcc** programy korzystające z tej biblioteki, należy wymusić jej dołączenie, przez użycie opcji **-lpthread**:

gcc -lpthread program.c

Każdy proces zawiera przynajmniej jeden główny wątek początkowy, tworzony przez system operacyjny w momencie stworzenia procesu. By do procesu dodać nowy wątek należy wywołać funkcję **pthread_create**. Nowo utworzony wątek zaczyna się od wykonania funkcji użytkownika (przekazanej mu przez argument **pthread_create**). Wątek działa aż do czasu wystąpienia jednego z następujących zdarzeń: zakończenia funkcji, wywołania funkcji **pthread_exit,** anulowania wątku za pomocą funkcji **pthread_cancel**, zakończenia procesu macierzystego wątku, wywołania funkcji **exec** przez jeden z wątków.

Przetwarzanie realizowane przez wątki musi być odpowiednio synchronizowane, tak, by wykonując operacje na wspólnych strukturach danych nie dopuścić do niespójności danych. Stosowane są dwie metody zapewnienia odpowiedniej koordynacji wątków: korzystanie z zamków (czyli blokad wzajemnie wykluczających, dalej nazywanych muteksami od nazw funkcji) lub korzystanie z konstrukcji nazywanych zmiennymi warunkowymi.

Zmienną muteksową można porównać do semafora binarnego, który wątki mogą posiadać. Mutex albo zezwala na dostęp, albo go zabrania. Zamknięcia muteksu może dokonać dowolny wątek znajdujący się w jego zasięgu, natomiast otworzyć go może tylko wątek który go zamknął. Wątki, które nie mogą uzyskać dostępu do muteksu są blokowane w oczekiwaniu na niego. Operacje wykonywane na muteksach są niepodzielne. Jeśli za pomocą muteksów trzeba synchronizować wątki kilku procesów, należy odwzorować muteks w obszar pamięci współdzielonej dostępny dla wszystkich procesów.

Kiedy niezbędne jest synchronizowanie wątków za pomocą bieżących wartości danych chronionych muteksami, można użyć konstrukcji nazywanych zmiennymi warunkowymi. Zmienna warunkowa jest kojarzona z konkretnym muteksem i predykatem. Podobnie jak muteks może ona być odwzorowana w pamięć współdzieloną, dzięki czemu może być używana przez parę procesów. Głównym zadaniem zmiennych warunkowych jest powiadamianie innych procesów o tym, że dany warunek został spełniony, lub do blokowania procesu w oczekiwaniu na otrzymanie powiadomienia.

W momencie kiedy wątek jest blokowany na zmiennej warunkowej, skojarzony z nim muteks jest zwalniany. W oczekiwaniu na to samo powiadomienie zablokowanych może być kilka wątków. Wątek wystosowuje powiadamianie wysyłając sygnał do skojarzonej zmiennej warunkowej.

II. Funkcje systemowe służące do tworzenia wątków

- **`pthread_create(pthread_t *thread, pthread_attr_t *attr, void* (*start_routine) (void*),void *arg)`** —

  utworzenie wątku. Wątek wykonuje funkcję wskazywaną przez parametr *start_routine.* Parametry funkcji muszą być przekazane przez wskaźnik na obszar pamięci (strukturę), który zawiera odpowiednie wartości. Wskaźnik ten jest przekazywany przez parametr *arg* i jest dalej przekazywany jako parametr aktualny do funkcji wykonywanej przez wątek. Parametr *attr* wskazuje na atrybuty wątku, a przez wskaźnik *thread* zwracany jest identyfikator wątku.

- **`pthread_exit(void *retval)`** —

  zakończenie wątku. Funkcja powoduje zakończenie wątku i przekazanie *retval*, jako wskaźnika na wynik. Wskaźnik ten może zostać przejęty przez inny wątek, który będzie wykonywał funkcję **pthread_join.**

- **`pthread_join(pthread_t th, void **thread_return)`** —

  oczekiwanie na zakończenie wątku. Funkcja umożliwia zablokowanie wątku w oczekiwaniu na zakończenie innego wątku, identyfikowanego przez parametr *th*. Jeśli oczekiwany wątek zakończył się wcześniej, funkcja zakończy się natychmiast. Funkcja przekazuje przez parametr *thread_return* wskaźnik na wynik wątku (wykonywanej przez niego funkcji), przekazany jako parametr funkcji **pthread_exit** wywołanej w zakończonym wątku.

- **`pthread_cancel(pthread_t thread)`** —

  zakończenie wykonywania innego wątku. Funkcja umożliwia wątkowi usunięcie z systemu innego wątku, identyfikowanego przez parametr *thread*.

III. Funkcje systemowe służące do synchronizacji wątków

1) Wzajemne wykluczanie:
Do zapewnienia wzajemnego wykluczania używana jest zmienna (o przykładowej nazwie *mutex*), zadeklarowana następująco:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- **`pthread_mutex_lock(pthread_mutex_t *mutex)`** —

  zajęcie zamka. Funkcja powoduje zajęcie zamka wskazywanego przez parametr *mutex*, (zajęcie sekcji krytycznej) poprzedzone ewentualnym zablokowaniem wątku do czasu zwolnienia zamka, jeśli został on wcześniej zajęty przez inny wątek.

- **`pthread_mutex_unlock(pthread_mutex_t *mutex)`** —

  zwolnienie zamka. Funkcja powoduje zwolnienie zamka wskazywanego przez parametr *mutex* (zwolnienie sekcji krytycznej), umożliwiając jego zajęcie innemu wątkowi.

- **`pthread_mutex_trylock(pthread_mutex_t *mutex)`** —

  próba zajęcia zamka. Funkcja powoduje zajęcie zamka wskazywanego przez parametr

mutex, jeśli nie jest zajęty przez inny wątek. W przeciwnym przypadku zwraca błąd, nie blokując tym samym procesu.

2) Zmienne warunkowe

Synchronizacja za pomocą zmiennych warunkowych polega na usypianiu i budzeniu wątku w sekcji krytycznej. W tym celu używana jest zmienna warunkowa (o przykładowej nazwie *cond*), zadeklarowana następująco:

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

- **`pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`** —

  oczekiwanie na sygnał. Funkcja powoduje uśpienie wątku na zmiennej warunkowej, wskazywanej przez parametr *cond*. Na czas uśpienia wątek zwalnia zamek, wskazywany przez parametr *mutex*, udostępniając tym samym sekcję krytyczną innym wątkom. Po obudzeniu i wyjściu z funkcji (na skutek odebraniu sygnału wysłanego przez **pthread_cond_signal**) zamek zajmowany jest ponownie.

- **`pthread_cond_signal(pthread_cond_t *cond)`** —

  wysłanie sygnału (obudzenie) do jednego z oczekujących na zmiennej warunkowej wskazywanej przez *cond*.

IV. Implementacja przykładowych programów obsługi wątków

Listing 1 prezentuje implementację programu tworzącego wątek.

```
   #include <pthread.h>
   #include <stdlib.h>
3  #include <unistd.h>

   void *Hello(void *arg) {
6     int i;
      for ( i=0; i<20; i++ ) {
            printf("Wątek mówi cześć!\n");
9           sleep(1);
      }
      return NULL;
12  }

   int main(void) {
15    pthread_t mojwątek;

      if ( pthread_create( &mojwątek, NULL, Hello,NULL) )
18          printf("błąd przy tworzeniu wątku\n"); abort();
      if ( pthread_join ( mythread, NULL ) ) {
            printf("błąd w kończeniu wątku\n");
21          exit();
      }
      exit(0);
24   }
```

Listing 1:Tworzenie wątku

Opis programu: W funkcji main() jest zadeklarowana zmienna typu *pthread_t* (zdefiniowany w pthread.h) o nazwie *mojwątek*. Zawiera ona identyfikator wątku i należy jej używać jako odniesienia do danego wątku w celu efektywnego nim zarządzania. Po zadeklarowaniu zmiennej *mojwątek*, wywoływana jest funkcja **pthread_create**, która de facto tworzy właściwy wątek. Samo wywołanie znajduje się w instrukcji warunkowej "if", dzięki czemu programista ma pewność, że funkcja zwróciła zero w przypadku sukcesu i wątek został powołany do życia. W przeciwnym wypadku zostanie wyświetlony komunikat o błędzie. Gdy funkcja **Hello** się wykona, wątek, który ją wywołał również zostanie zakończony. W tym przypadku nie robi ona nic sensownego, po prostu wypisuje "Wątek mówi cześć!" dwudziestokrotnie, na standardowe wyjście. Warto zwrócić uwagę, że przyjmuje ona argument typu void* i takiego samego typu wartość jest zwracana. Oznacza to, że można przekazać jej dodatkową porcję danych w argumencie i można też takową otrzymać.
Przedstawiony program składa się z dwu wątków – program główny także jest traktowany jako

Listing 2 przedstawia program, w którym z powodu braku synchronizacji wątki nadpisują wyniki swojej pracy.

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

int mojazmiennaglobalna=0;

void *Hello(void *arg) {
   int i,j;
   for ( i=0; i<20; i++ ) {
         j=mojazmiennaglobalna;
         j=j+1;
         printf(".");
         fflush(stdout);
         sleep(1);
         jazmiennaglobalna=j;
   }
   return NULL;
}

int main(void) {

   pthread_t mojwątek;
   int i;

   if ( pthread_create( &mojwątek, NULL, Hello, NULL) ) {
         printf("błąd przy tworzeniu wątku.");
         abort();
   }
   for ( i=0; i<20; i++) {
         mojazmiennaglobalna=mojazmiennaglobalna+1;
         printf("o");
         fflush(stdout);
         sleep(1);
   }
   if ( pthread_join ( mojwątek, NULL ) ) {
         printf("błąd przy kończeniu wątku.");
         abort();
   }
   printf("\nMoja zmienna globalna wynosi
               %d\n",mojazmiennaglobalna);
   exit(0);
}
```

Listing 2: Nadpisywanie wyników przez współbieżnie działające wątki

Opis programu: Program, podobnie jak poprzedni, tworzy nowy wątek. Zarówno ten wątek, jak i wątek główny inkrementują zmienną "mojazmiennaglobalna" dwudziestokrotnie. Jednakże w wyniku współbieżnego działania wątki nadpisują zapisywane przez siebie wartości.

Listing 3 przedstawia poprawiony program z listingu 2, który wykorzystuje muteksy w celu synchronizacji wątków.

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

int mojazmiennaglobalna;
pthread_mutex_t mojmuteks=PTHREAD_MUTEX_INITIALIZER;
void *Hello(void *arg) {
   int i,j;
   for ( i=0; i<20; i++ ) {
         pthread_mutex_lock(&mojmuteks);
         j=mojazmiennaglobalna;
         j=j+1;
         printf(".");
         fflush(stdout);
         sleep(1);
         mojazmiennaglobalna=j;
         pthread_mutex_unlock(&mojmuteks);
   }
   return NULL;
}

int main(void) {
   pthread_t mojwątek;
   int i;

   if ( pthread_create( &mojwątek, NULL, Hello, NULL) ) {
       printf("błąd przy tworzeniu wątka.");
       abort();
   }
   for ( i=0; i<20; i++) {
         pthread_mutex_lock(&mojmuteks);
         mojazmiennaglobalna=mojazmiennaglobalna+1;
         pthread_mutex_unlock(&mojmuteks);
         printf("o");
         fflush(stdout);
         sleep(1);
   }
   if ( pthread_join ( mojwątek, NULL ) ) {
         printf("błąd przy kończeniu wątka.");
         abort();
   }
   printf("\nWartość mojej zmiennej globalnej to
         %d\n",mojazmiennaglobalna);
   exit(0);
}
```

Listing 3: Synchronizacja za pomocą muteksów

Opis programu: W przedstawionym programie pojawiły się wywołania funkcji **pthread_mutex_lock** oraz **pthread_mutex_unlock**. Umożliwiają one wykonywanie jednoczesnych operacji. Gdy jeden wątek ma odblokowany muteks, to drugi ma zablokowany. Zatem jeśli wątek A próbuje zablokować muteks, podczas gdy wątek B już go blokuje, to wówczas A zostaje uśpiony. Jak tylko B zwolni muteks (dzięki funkcji pthread_mutex_unlock()), A będzie w stanie go zablokować

na swój użytek (innymi słowy, pthread_mutex_lock() zwróci kod sukcesu). Analogicznie, jeśli wątek C spróbuje zablokować ten muteks, podczas gdy A już blokuje, to C zostanie uśpiony na jakiś czas. Wszystkie wątki, które zostaną uśpione, będą kolejkowane do danego muteksu.

V. Zadania do samodzielnego wykonania.

1) Zaproponować rozwiązanie problemu producenta-konsumenta za pomocą wątków.
2) Napisać program komunikujący się poprzez dwa potoki nazwane z inną kopią tego samego programu. Program powinien działać na zasadzie programu `talk(1)`. Jeden wątek będzie odpowiedzialny za pobieranie danych od użytkownika i wysyłanie ich do potoku wyjściowego, a drugi wątek będzie odbierał dane i wyświetlał je na standardowym wyjściu. Przykładowe wywołanie: ftalk -i /tmp/fifo.1 -o /tmp/fifo.2.

   Przełączniki -i i -o regulują, który potok jakie będzie pełnił funkcje.

VI. Literatura.

[HGS99] Havilland K., Gray D., Salama B., *Unix - programowanie systemowe*, ReadMe, 1999

[Roch97] Rochkind M.J., *Programowanie w systemie UNIX dla zaawansowanych*, WNT, 1997

[NS99] Neil M., Stones R, *Linux. Programowanie*, ReadMe, 1999

[MOS02] Mitchell M., Oldham J., Samuel A., *Linux. Programowanie dla zaawansowanych*, ReadMe,2002

[St02] Stevens R.W., *Programowanie w środowisku systemu UNIX*, WNT, 2002