Kurs C

Radosław Łukasik

Wykład 5

Wątki a procesy

Procesy:

- osobne programy wykonywane równolegle;
- każdy proces ma osobny stos, zmienne, pamięć.

Wątki:

- funkcje z jednego programu wykonywane równolegle;
- każdy wątek ma osobny stos;
- zmienne i pamięć są współdzielone.

Wielowątkowość w C

POSIX threads - realizowane przy pomocy biblioteki **pthread.h**, dostępne dla Windows, Linux, BSD, MacOS X. Jest to standaryzowana biblioteka (wersja na Windows https://sourceware.org/pthreads-win32/ nie jest!). MinGW zawiera własną wersję pliku **pthread.h** dla Windows, więc (o ile nie ma z nią jakichś problemów) nie trzeba dodawać dodatkowych bibliotek (pod linuxem w Code::Blocks należy dodać -pthread do Project Build Options -> Linker Settings -> Other linker options.

C11 threads - realizowane za pomocą biblioteki **thread.h**, bazuje na **POSIX threads**, działanie zależne od kompilatora i systemu operacyjnego (w Windows nie działa!).

Podstawowe typy i atrybuty związane z wątkami

Prawie wszystkie funkcje związane z POSIX zwracają 0 w przypadku sukcesu, w przeciwnym wypadku zwracają kod błędu, najczęstsze z nich, to **EINVAL** - nieprawidłowa wartość parametru (któregokolwiek), **ENOMEM** - brak pamięci do wykonania operacji, **EPERM** - odmowa dostępu (np. zbyt niski priorytet procesu), **EBUSY** - zablokowany wątek, mutex itp., **EDEADLK** - wykryto zakleszczenie (deadlock), **EAGAIN** - wyczerpanie dostępnych zasobów (nie pamięć), osiągnięcie limitu rekurencji, **ENOTSUP** - nie obsługiwany parametr.
Zanim przejdziemy do funkcji tworzących wątek zacznijmy od przedstawienia podstawowych typów z nimi związanymi:

- **pthread_t** - identyfikator wątku;
- **pthread_attr_t** - atrybuty wątku, odczytywane lub ustawiane pośrednio poprzez odpowiednie funkcje zaczynające się odpowiednio od **pthread_attr_get** i **pthread_attr_set**. Każda zmienna przechowująca atrybuty wątku musi być przed użyciem zainicjowana, a gdy przestaje być potrzebna, to należy zwolnić zasoby z nią powiązane. Robi się to funkcjami:

```
int pthread_attr_init(pthread_attr_t *attr);// inicjacja
int pthread_attr_destroy(pthread_attr_t *attr);// zwalnianie
```

Funkcje odczytujące/zmieniające atrybuty wątku (wszystkie poniższe funkcje zwracają 0 w przypadku sukcesu):

- rodzaj wątku:

```
int pthread_attr_setdetachstate(pthread_attr_t *attr,
                                int detachstate);
int pthread_attr_getdetachstate(const pthread_attr_t *attr,
                                int *detachstate);
```

  zmienna **detachstate** może być jedną z dwóch wartości: **PTHREAD_CREATE_JOINABLE** (domyślna) lub **PTHREAD_CREATE_DETACHED**. Wątki typu **JOINABLE** nie usuwają po sobie danych (byśmy mogli z nich jeszcze skorzystać) i musimy sami to zrobić za pomocą funkcji **pthread_join**. Wątki typu **DETACHED** automatycznie czyszczą po sobie pamięć. Każdy wątek typu **JOINABLE** można przekształcić na typ **DETACHED** (ale nie odwrotnie!) za pomocą funkcji:

```
int pthread_detach(pthread_t id);
```

Wątki mogą być wykonywane z różnymi priorytetami. Aby uszczegółowić te rzeczy możemy użyć poniższych funkcji:

- dziedziczenie ustawień:

```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
int pthread_attr_getinheritsched(const pthread_attr_t *attr,
                                 int *inheritsched);
```

  Zmienna **inheritsched** przyjmują jedną z dwóch wartości: **PTHREAD_INHERIT_SCHED** - domyślna, oznacza dziedziczenie rodzaju szeregowania z wywoływanego wątku lub **PTHREAD_EXPLICIT_SCHED** - oznacza, że parametry są zapisane w atrybutach.

- polityka przełączania się na liście wątków:

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_getschedpolicy(pthread_attr_t *attr,int *policy);
```

  gdzie **policy** może przyjmować wartości **SCHED_OTHER** - domyślny, **SCHED_FIFO**, **SCHED_RR** - cykliczna lista wątków (przełączanie po pewnym czasie na kolejny) Ogólnie rzecz biorąc zdarza się, że działa tylko **SCHED_OTHER**. Wynika to z tego, że pozostałe opcje wymagają uprawnień na poziomie system.

- priorytet wątku:

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
            const struct sched_param *param);
int pthread_attr_getschedparam(pthread_attr_t *attr,
            struct sched_param *param);
int pthread_setschedparam(pthread_t thread, int policy,
            const struct sched_param *param);
int pthread_getschedparam(pthread_t thread, int *policy,
            struct sched_param *param);
```

Zmienna **policy** oznacza politykę, **sched_param** jest strukturą, która zawiera co najmniej jedno pole **sched_priority** typu **int**, oznaczające priorytet, przy czym wartość ta powinna być pomiędzy minimalną i maksymalną możliwą, które można pobrać za pomocą funkcji:

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

Pod linuxem przy polityce **SCHED_OTHER** powyższe dwie funkcje zwracają 0 i jest to jedyny priorytet do wyboru.

- rozmiar stosu wątku:

```
int pthread_attr_setstacksize(pthread_attr_t *attr,size_t size);
int pthread_attr_getstacksize(pthread_attr_t *attr,size_t *size);
```

size=0 oznacza taki sam rozmiar stosu jak w wątku tworzącym. Podawany rozmiar musi być między pewna minimalną i maksymalną wartością na którą zezwala system.

Tworzenie wątku

Aby utworzyć wątek możemy użyć funkcji:

```
int pthread_create(pthread_t *id, const pthread_attr_t *attr,
                   void* (*fun)(void*), void* arg);
```

W **attr** należy podać atrybuty, o których mowa była wcześniej. Jeśli podamy tutaj wartość **NULL**, to będą to domyślne parametry. Należy również podać funkcję, która będzie wykonywana w wątku, ma ona jeden parametr-wskaźnik, przez który możemy przesłać jakieś dane. Zwraca również wskaźnik na dowolny typ, poprzez który możemy coś zwracać o ile jest taka potrzeba. Po zwróceniu wartości wątek kończy swoje działanie. W **id** zostanie zapisany identyfikator utworzonego wątku. Dla wątków możliwe mamy sprawdzenie czy są to te same za pomocą:

```
int pthread_equal(pthread_t id1, pthread_t id2);// 0 gdy różne
```

lub pobranie samego identyfikatora wątku wewnątrz jego funkcji:

```
pthread_t pthread_self();
```

Istnieje również funkcja, która wymusza (wewnątrz wątku) zakończenie działania wątku:

```
void pthread_exit(void *value_ptr);
```

wartość podana jako parametr będzie zwrócona na zakończenie wątku. Należy tutaj pamiętać, że jeżeli główny proces zakończy swe działanie, to wszystkie jego watki zostaną zmuszone do zakończenia swego działania.
W głównym procesie programu może się zdarzyć, że będziemy czekać aż pewne wątki zakończą swe działanie. Do tego służy funkcja:

```
int pthread_join(pthread_t id, void **value_ptr);
```

która czeka na zakończenie wątku **id** i zapisuje wskaźnik na wartość przez niego zwracaną.
Jeżeli wewnątrz wątku chcemy oddać sterowanie do innych wątków, to możemy to zrobić za pomocą:

```
int sched_yield(void);
```

przy czym może się zdarzyć, że funkcja z powrotem uruchomi wątek ją wywołujący.

Watki (slajd 1)

Watki.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <x86intrin.h>
#include <time.h>
#define N 16
#define ROZM 1024*64
typedef struct{
    unsigned long long cykle;
    unsigned int rdzen;
    int prior;
    int* tab;
}wynik;

struct sched_param param;
pthread_attr_t attr;
pthread_t id[N];// miejsce na identyfikatory watków
wynik wyniki[N];
int i,zwrot,pmin,pmax;
```

Watki (slajd 2)

```
void* watek(void* _arg){// funkcja wątków
    unsigned long long licz1, licz2;// przechowywanie liczników cykli
    int polityka, *tab, stala=640000;
licz1=__rdtscp(&(((wynik*)_arg)->rdzen));// odczyt cykli i rdzenia*
    pthread_getschedparam(pthread_self(), &polityka, &param);
    ((wynik*)_arg)->prior =param.sched_priority;
    tab=((wynik*)_arg)->tab;
    if(tab!=0){
        for(int i=0;i<ROZM;i++) tab[i]=rand();
        int wi,wj,value;
        for(wi=1;wi<ROZM;wi++){// sortowanie przez wstawianie
            value=tab[wi];
            for(wj=wi-1;wj>=0 && tab[wj]>value;wj--)
                tab[wj+1]=tab[wj];
            tab[wj+1]=value;
            if(wi*wi==stala){// pozwalamy innym wątkom
                sched_yield();// na włączenie się
                stala+=stala;
            }
        }
    }
licz2=__rdtscp(&(((wynik*)_arg)->rdzen));// odczyt cykli i rdzenia*
    ((wynik*)_arg)->cykle =licz2-licz1;// zapis cykli
    return 0;
}
```

Watki (slajd 3)

```
int main(){
    srand(time(0));
    for(i=0;i<N;i++)// rezerwacja miejsca na tablice
        wyniki[i].tab=malloc(ROZM*sizeof(int));
    pthread_attr_init(&attr);// inicjacja atrybutów
    pmin=sched_get_priority_min(SCHED_OTHER);// pobranie minimalnego
    pmax=sched_get_priority_max(SCHED_OTHER);// i maks. priorytetu
    printf("Zakres priorytetow %d - %d\n",pmin,pmax);
    for (i=0;i<N;i++){/* utworzenie kilku wątków */
        param.sched_priority=((pmax-pmin)*i)/(N-1)+pmin;// priorytety
        pthread_attr_setschedparam(&attr,&param);
        zwrot = pthread_create(&id[i], &attr, watek, &wyniki[i]);
        if(zwrot) printf("Nie powiodło sie pthread_create\n");
    }
    for (i=0;i<N;i++){/* oczekiwanie na zakończenie wątków */
        if(pthread_join(id[i], NULL)) perror("Blad watku");
        else{
            printf("Watek: %2d, prior. %3d, cykle %11llu, rdzen %u\n",
    i,wyniki[i].prior,wyniki[i].cykle,wyniki[i].rdzen);
            free(wyniki[i].tab);// zwalniamy pamięć tablic
        }
    }
    pthread_attr_destroy(&attr);// zwalniamy miejsce atrybutów wątków
    return 0;
}
```

Mutexy

Mutex (MUTual EXclusion) - jest mechanizmem wzajemnego wykluczania, który chroni dane współdzielone przez kilka wątków. W danym momencie, tylko jeden wątek może mieć dostęp do danych współdzielonych.
Z Mutexami w pthreads związane są struktury: **pthread_mutex_t** dla samego mutexu oraz **pthread_mutexattr_t** opisującej jego atrybuty.
Najpierw należy zainicjować mutex za pomocą:

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

przy czym atrybuty mogą być wskaźnikiem **NULL**. Gdy już przestaje nam potrzebny nalezy go zwolnić za pomocą:

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

Mamy również możliwość statycznej inicjacji mutexu podczas kompilacji przypisując do niego jedną ze stałych: **PTHREAD_MUTEX_INITIALIZER** - zwykły, **PTHREAD_RECURSIVE_MUTEX_INITIALIZER** - rekursywny, **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER** - bezpieczny. Mutexy takie nie wymagają obsługi błędów, ale również na koniec należy je zwolnić za pomocą **pthread_mutex_destroy**.
Wątek może pozyskać blokadę na trzy sposoby:

- "do skutku" - czeka w nieskończoność aż mutex zostanie zwolniony poprzez inne wątki:

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

- jednorazowa - jeżeli mutex jest zablokowany, to zwraca od razu błąd **EBUSY**:

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

- "ograniczona czasowo" - czeka określoną ilość czasu aż mutex zostanie zwolniony poprzez inne wątki, gdy się nie doczeka zwraca błąd **ETIMEDOUT**:

```
int pthread_mutex_timedlock(pthread_mutex_t *mut,
                            const struct timespec *timeout);
```

gdzie struktura **timespec** ma dwie składowe **tv_sec** typu **time_t** oraz **tv_nsec** typu **long** oznaczające kolejno czas w sekundach plus czas w nanosekundach, w którym należy zakończyć oczekiwanie (czas bezwzględny, do pobrania aktualnego czasu użyjmy **time(0)**).

Zwalnianie blokady następuje zawsze funkcją:

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

Jeżeli chodzi o atrybuty mutexu, to żeby ich użyć należy najpierw je zainicjować:

```
int pthread_mutexattr_init(pthread_mutexattr_t *mutattr);
```

natomiast, gdy już go nie potrzebujemy, to należy go usunąć za pomocą:

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *mutattr);
```

Do dyspozycji mamy następujące atrybuty:

- współdzielenie mutexu z innymi procesami:

```
int pthread_mutexattr_setpshared(pthread_mutexattr_t *mutattr,
                                 int pshared);
int pthread_mutexattr_getpshared(const pthread_mutexattr_t
                                 *mutattr, int *pshared);
```

gdzie **pshared** może przyjmować wartości **PTHREAD_PROCESS_PRIVATE** (domyślny, bez współdzielenia) lub **PTHREAD_PROCESS_SHARED** (współdzielenie).

- typ mutexu:

```
int pthread_mutexattr_settype(pthread_mutexattr_t *mutattr,
                              int type);
int pthread_mutexattr_gettype(const pthread_mutexattr_t *mutattr,
                              int *type);
```

  gdzie typ jest jednym z:

  - **PTHREAD_MUTEX_NORMAL** - normalny, nie wykrywa zakleszczeń (wątki czekające nawzajem na zakończenie drugiego). Wątek blokujący powtórnie bez wcześniejszego odblokowania powoduje zakleszczenie. Odblokowywanie niezablokowanego lub zablokowanego przez inny wątek mutexu powoduje nieprzewidziane zachowanie.
  - **PTHREAD_MUTEX_ERRORCHECK** - bezpieczny, pozwala na zwracanie błędów w przypadku próby blokowania lub zwalniania już zablokowanego przez inny wątek mutexu, czy też zwolnienia niezablokowanego mutexu.
  - **PTHREAD_MUTEX_RECURSIVE** - rekursywny, pozwala na wielokrotne blokowania (zliczanie w obrębie jednego wątku), które wymagają do zwolnienia taką samą liczbę odblokowań. Nie występuje tutaj zakleszczenie. Błędy są zwracane w przypadku próby odblokowania mutexu zablokowanego przez inny wątek lub niezablokowanego mutexu.

Dla wątków z polityką **SCHED_FIFO** lub **SCHED_RR** znaczenie mają również:

- zmiana protokołu kolejności żądań blokad - pozwala na uwzględnienie priorytetów wątków przy ustalaniu kolejności wątków blokujących mutexy:

```
int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int protocol);
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *attr,
                                  int *protocol);
```

  gdzie protokół może być równy:

  - **PTHREAD_PRIO_NONE** - domyślny, bez uwzględniania;
  - **PTHREAD_PRIO_INHERIT** - wątek blokujący inne wątki o większym priorytecie uzyskuje chwilowo większy priorytet (maksimum z wątków blokujących), by wykonać zablokowaną część szybciej;
  - **PTHREAD_PRIO_PROTECT** - jak wyżej, ale priorytet nie zależy od priorytetów wątków lecz od priorytetu mutexu opisanego poniżej.

- zmiana priorytetu mutexu:

```
int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                     int prioceiling);
int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t
                                     *attr, int *prioceiling);
int pthread_mutex_setprioceiling(pthread_mutex_t *mut,
                                 int prioceiling);
int pthread_mutex_getprioceiling(const pthread_mutex_t *mut,
                                 int *prioceiling);
```

Typy mutexów (slajd 1) TypyMutexow.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
pthread_t id[2];
pthread_attr_t attr;
pthread_mutex_t mut;
pthread_mutexattr_t mutattr;
int x;
int typmut[2]={PTHREAD_MUTEX_RECURSIVE,PTHREAD_MUTEX_ERRORCHECK};
char typmutopis[2][32]={"PTHREAD_MUTEX_RECURSIVE","
    PTHREAD_MUTEX_ERRORCHECK"};

void funkcja(void){
    int zwrot;
    puts("Wywolano funkcje");
    zwrot=pthread_mutex_lock(&mut);
    if(zwrot) perror("Blad blokowania (funkcja)");
    else{
        puts("Pomyslnie zablokowano (funkcja)");
        x++;
        zwrot=pthread_mutex_unlock(&mut);
        if(zwrot) perror("Blad odblokowania (funkcja)");
        else puts("Pomyslnie odblokowano (funkcja)");
    }
}
```

Typy mutexów (slajd 2)

```
void* watek1(void* _arg){
    int zwrot;
    puts("Uruchomiono watek 1");
    zwrot=pthread_mutex_lock(&mut);
    if(zwrot) perror("Blad blokowania (watek)");
    else{
        puts("Pomyslnie zablokowano (watek)");
        funkcja();
        zwrot=pthread_mutex_unlock(&mut);
        if(zwrot) perror("Blad odblokowania (watek)");
        else puts("Pomyslnie odblokowano (watek)");
    }
    return 0;
}

void* watek2(void* _arg){
    if(!pthread_mutex_lock(&mut)){
        x*=3;
        pthread_mutex_unlock(&mut);
    }
    return 0;
}

void* (*wskfwatki[2])(void*)={watek1,watek2};
```

Typy mutexów (slajd 3)

```
int main(){
    int i,j,zwrot;
    if(pthread_mutexattr_init(&mutattr)){
        perror("Blad inicjacji mutexu"); exit(1); }
    for(i=0;i<2;i++){
        x=3;
        printf("Proba %s\n",typmutopis[i]);
        pthread_mutexattr_settype(&mutattr,typmut[i]);
        if(pthread_mutex_init(&mut,&mutattr)){
            perror("Blad inicjacji mutexu"); exit(2); }
        for(j=0;j<2;j++){
            zwrot=pthread_create(&id[j], NULL, wskfwatki[j], NULL);
            if(zwrot) printf("Nie utworzono watku %d\n",j);
        }
        for(j=0;j<2;j++){
            zwrot = pthread_join(id[j], NULL);
            if(zwrot) perror("Blad watku");
            else printf("Watek %d zakonczony prawidlowo\n",j);
        }
        printf("x=%d\n",x);
        pthread_mutex_destroy(&mut);
    }
    pthread_mutexattr_destroy(&mutattr);// zwalniamy atrybuty mutexu
    return 0;
}
```

Blokowanie ograniczone czasowo (slajd 1) timedlock.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
pthread_mutex_t mut;
int x=1;
struct timespec czas;
void* watek1(void* _arg){
    puts("1");
    czas.tv_sec=time(0); czas.tv_nsec=400;
    if(pthread_mutex_timedlock(&mut,&czas)) puts("2");
    else{ puts("3"); x*=2;
        if(pthread_mutex_unlock(&mut)) puts("4");
        else puts("5");}
    return 0;
}
void* watek2(void* _arg){
    puts("A");
    czas.tv_sec=time(0); czas.tv_nsec=400;
    if(pthread_mutex_timedlock(&mut,&czas)) puts("B");
    else{ puts("C"); x*=3;
        if(pthread_mutex_unlock(&mut)) puts("D");
        else puts("E");}
    return 0;
}
```

Blokowanie ograniczone czasowo (slajd 2)

```
pthread_t id[2];
void* (*wskfwatki[2])(void*)={watek1,watek2};
int main(){
    int j;
    if(pthread_mutex_init(&mut,NULL)){
        perror("Blad inicjacji mutexu"); exit(2); }
    for(j=0;j<2;j++){
        if(pthread_create(&id[j], NULL, wskfwatki[j], NULL))
            printf("Nie utworzono watku %d\n",j);
    }
    for(j=0;j<2;j++){
        if(pthread_join(id[j],NULL)) printf("Blad watku %d\n",j);
        else printf("Watek %d zakonczony prawidlowo\n",j);
    }
    printf("x=%d\n",x);
    pthread_mutex_destroy(&mut);
    return 0;
}
```

Blokady zapisu lub odczytu

Mutexy zawsze blokowały dostęp do zmiennej niezależnie od tego czy chcieliśmy ją odczytywać czy zapisywać. Jeśli chcielibyśmy to rozróżniać, to możemy użyć blokad zapisu lub odczytu. Podobnie jak wcześniej mamy typ dla blokady **pthread_rwlock_t** oraz jej atrybutów **pthread_rwlockrattr_t**. Do inicjacji i odpowiednio zwolnienia zasobów blokady służą funkcje:

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        const pthread_rwlockattr_t *attr);
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);
```

przy czym podanie wskaźnika do atrybutów jako pusty oznacza domyślne atrybuty. Można również zainicjować blokadę w sposób statyczny z domyślnymi parametrami za pomocą przypisania do niej stałej **PTHREAD_RWLOCK_INITIALIZER**. Podobnie do inicjacji i zwalniania atrybutów bariery mamy:

```
int pthread_rwlockattr_init(pthread_rwlockattr_t *attr);
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

Jedynym dostępnym atrybutem jest współdzielenie blokady z innymi procesami:

```
int pthread_rwlockattr_setpshared(pthread_rwlockattr_t *attr,
                                  int pshared)
int pthread_rwlockattr_getpshared(pthread_barrierattr_t *attr,
                                  int *pshared);
```

z trybami **PTHREAD_PROCESS_PRIVATE** (domyślny) i **PTHREAD_PROCESS_SHARED**.

Do założenia blokady do odczytu mamy trzy funkcje różniące się sposobem oczekiwania:

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);// w nieskończ.
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);// jedna próba
int pthread_rwlock_timedrdlock(pthread_rwlock_t *rwlock,
                    const struct timespec *abs_timeout);// czasowa
```

Podobnie do założenia blokady do zapisu:

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);// w nieskończ.
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);// jedna próba
int pthread_rwlock_timedwrlock(pthread_rwlock_t *rwlock,
                    const struct timespec *abs_timeout);// czasowa
```

Do zwalniania powyższych blokad służy:

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

W blokadach ograniczonych czasem, czas jest podawany jako bezwzględny. Jeżeli nie uda się uzyskać blokady przed upłynięciem podanego czasu, to zwracany jest błąd **ETIMEDOUT**. Przy blokadzie do odczytu dostęp może wiele wątków na raz, w przypadku blokady do zapisu tylko jeden.

Blokady (slajd 1)

Blokady.c

```c
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#ifdef _WIN32
    #include <windows.h>
#else
    #include <time.h>
#endif
#define N 5 /* liczba wątków odczytujących */
#define K 2 /* liczba wątków zapisujących */
pthread_rwlock_t blokada=PTHREAD_RWLOCK_INITIALIZER;
int wartosc; // obiekt chroniony blokadą
void ms_sleep(unsigned czas){
#ifdef _WIN32
    Sleep(czas);
#else
    struct timespec req;
    req.tv_sec  = (czas / 1000);
    req.tv_nsec = (czas % 1000 * 1000000);
    nanosleep(&req, NULL);
#endif
}
```

Blokady (slajd 2)

```
void* pisarz(void* numer){
    for(int i=0;i<2;i++){
        printf("%c", (int)numer+'0');
        if(!pthread_rwlock_wrlock(&blokada)){
            printf("%c",(int)numer+'3'); ms_sleep(100);
            printf("%c", (int)numer+'6');
            if(pthread_rwlock_unlock(&blokada))
                printf("blad zw. blokady z %d\n",(int) numer);
        } else printf("blad blokady z %d\n",(int) numer);
        ms_sleep(400);
    } return 0;
}
void* czytelnik(void* numer){
    for(int i=0;i<2;i++){
        printf("%c", (int)numer+'a');
        if(!pthread_rwlock_rdlock(&blokada)){
            printf("%c", (int)numer+'A'); ms_sleep(20);
            printf("%c", (int)numer+'A');
            if(pthread_rwlock_unlock(&blokada))
                printf("blad zw. blokady o %d\n",(int) numer);
        } else printf("blad blokady o %d\n",(int) numer);
        ms_sleep(20);
    } return 0;
}
```

Blokady (slajd 3)

```
int main() {
    pthread_t odczyt[N];
    pthread_t zapis[K];
    int i;
    pthread_rwlock_init(&blokada, NULL);
    for(i=0;i<K;i++)
        if(pthread_create(&zapis[i],0,pisarz,(void*)i))
            printf("Blad tworzenia o %d\n",i);
    for(i=0;i<N;i++)
        if(pthread_create(&odczyt[i],0,czytelnik,(void*)i))
            printf("Blad tworzenia o %d\n",i);
    for(i=0;i<N;i++)
        if(pthread_join(odczyt[i],0))
            printf("Blad watku o %d\n",i);
    for(i=0;i<K;i++)
        if(pthread_join(zapis[i],0))
            printf("Blad watku z %d\n",i);
    pthread_rwlock_destroy(&blokada);
    return 0;
}
```

Zmienne warunkowe

Zmienne warunkowe są jednym ze sposobów synchronizacji między wątkami. Mają za zadanie wysłanie sygnałów (z pewnego wątku) do wątków oczekujących na dany sygnał. Jest ona zawsze używana z mutexem (nierekursywnym).
Ze zmiennymi warunkowymi są związane typy **pthread_cond_t** oraz **pthread_condattr_t** do obsługi atrybutów. Zmienna warunkowa wymaga inicjacji, a gdy staje się zbędna, to należy ją zwolnić:

```
int pthread_cond_init(pthread_cond_t *cond,
                      const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Wskaźnik na atrybuty może być zerowy, wtedy są one domyślne. Możliwa jest statyczna inicjacja przy pomocy stałej o domyślnych atrybutach **PTHREAD_COND_INITIALIZER**. Mamy dwa możliwe atrybuty:

- współdzielenie zmiennej warunkowej między procesami:

```
int pthread_condattr_setpshared(pthread_condattr_t *attr,
                                int pshared);
int pthread_condattr_getpshared(pthread_condattr_t *attr,
                                int *pshared);
```

gdzie **pshared** może być **PTHREAD_PROCESS_PRIVATE** - bez współdz., domyślny lub **PTHREAD_PROCESS_SHARED** - współdz.

- ustawienie identyfikatora zegara używanego do odmierzania czasu:

```
int pthread_condattr_setclock(pthread_condattr_t *attr,
                              clockid_t clock_id);
int pthread_condattr_getclock(pthread_condattr_t *attr,
                              clockid_t *clock_id);
```

  domyślnie zegar jest zegarem systemowym, na Windows nie ma innych, na linux są **CLOCK_REALTIME** (domyślny), **CLOCK_MONOTONIC** i inne, przy czym w Linux jeśli uwzględniamy atrybuty, to należy jakikolwiek ustawić, by działo poprawnie oczekiwanie ograniczone czasowo.

Wątek, który zmienia warunek może to sygnalizować (jednemu, nieokreślonemu) lub rozgłaszać (wszystkim) innym wątkom poprzez funkcje:

```
int pthread_cond_signal(pthread_cond_t *cond);//sygnalizacja
int pthread_cond_broadcast(pthread_cond_t *cond);//rozgłaszanie
```

Natomiast oczekujące wątki do czekania na sygnał wykorzystują:

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

gdzie **abstime** oznacza czas bezwzględny.
Należy tutaj uważać, by sygnał był wysyłany gdy inny wątek już na niego czeka (w szczególności wątek oczekujący został już uruchomiony).
Przy oczekiwaniu blokada mutexu zostaje zwolniona aż do uzyskania sygnału.

Zmienne warunkowe (slajd 1) Warunkowe.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#define N 64*1024
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t war=PTHREAD_COND_INITIALIZER;
typedef struct{
    int* wsk;
    int n;
    float med;
}tablica;
void* mediana(void* _arg){
    struct timespec czas; tablica *t=(tablica*)_arg;
    czas.tv_sec=time(0)+4; czas.tv_nsec=400;
    if(pthread_mutex_lock(&mut)) puts("Blad blokowania w mediana");
    else{
        puts("Mediana czeka");
        if(pthread_cond_timedwait(&war,&mut,&czas)) puts("Blad");
        t->med=t->wsk[t->n >>1];
        if((t->n & 1)==0){ t->med+=t->wsk[(t->n >>1)-1]; t->med/=2; }
        pthread_mutex_unlock(&mut);
    }
    return 0;
}
```

Zmienne warunkowe (slajd 2)

```
void* sortuj(void* _arg){
    int pol,ind,wi,wj,value,*tab;
    int stala=(((tablica*)_arg)->n)*(((tablica*)_arg)->n)-640000;
    tab=((tablica*)_arg)->wsk;
    pol=((tablica*)_arg)->n >>1;
    if(pthread_mutex_lock(&mut)) perror("Blad blokowania");
    else{
        for(wi=((tablica*)_arg)->n -1;wi>0;wi--){
            ind=wi;
            for(wj=wi-1;wj>=0;wj--)
                if(tab[wj]>tab[ind]) ind=wj;
            value=tab[ind]; tab[ind]=tab[wi]; tab[wi]=value;
            if(wi==pol){
                puts("Wysylamy sygnal");
                pthread_cond_signal(&war);
            }
            if(stala>=wi*wi){// pozwalamy innym wątkom
                sched_yield();// na włączenie się
                stala=wi*wi -640000;
            }
        }
        pthread_mutex_unlock(&mut);
    }
    return 0;
}
```

Zmienne warunkowe (slajd 3)

```
pthread_t id[2];
int main(){
    int j;
    tablica tab;
    tab.wsk=malloc(N*sizeof(int)); tab.n=N;
    if(!tab.wsk){
        puts("Nie udalo sie utworzyc tablicy"); exit(1); }
    srand(time(0));
    for(j=0;j<N;j++)
        tab.wsk[j]=rand();
    if(pthread_create(&id[1], NULL, mediana, &tab)){
        puts("Nie utworzono watku mediana"); exit(5); }
    if(pthread_create(&id[0], NULL, sortuj, &tab)){
        puts("Nie utworzono watku sortujacego"); exit(4); }
    if(pthread_join(id[0],NULL)) puts("Blad wyjscia watku sortuj");
    else puts("Watek sortuj zakonczony prawidlowo");
    if(pthread_join(id[1],NULL)) puts("Blad wyjscia watku mediana");
    else puts("Watek mediana zakonczony prawidlowo");
    printf("%f\n",tab.med);
    pthread_cond_destroy(&war);
    pthread_mutex_destroy(&mut);
    return 0;
}
```

Bariery

Kolejną metodą synchronizacji wątków jest bariera. Polega ona na tym, że wątki są wstrzymywane aż do momentu dojścia ostatniego z nich do tej bariery. Podobnie jak wcześniej mamy typ dla bariery **pthread_barrier_t**oraz jej atrybutów **pthread_barrierattr_t**. Do inicjacji bariery służy funkcja

```
int pthread_barrier_init(pthread_barrier_t *barrier,
    pthread_barrierattr_t* attr, unsigned int count);
```

count mówi ile wątków jest powiązanych z barierą. **attr** może być wskaźnikiem zerowym - domyślne atrybuty. Do zwolnienia pamięci związanych z barierą służy:

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

Podobnie do inicjacji i zwalniania atrybutów bariery mamy:

```
int pthread_barrierattr_init(pthread_barrierattr_t *attr);
int pthread_barrierattr_destroy(pthread_barrierattr_t *attr);
```

Jedynym dostępnym atrybutem jest współdzielenie bariery z innymi procesami:

```
int pthread_barrierattr_setpshared(pthread_barrierattr_t *attr,
                                   int pshared);
int pthread_barrierattr_getpshared(pthread_barrierattr_t *attr,
                                   int *pshared);
```

z trybami **PTHREAD_PROCESS_PRIVATE** (domyślny) i **PTHREAD_PROCESS_SHARED**.

Aby wstrzymać wątek do momentu dojścia wszystkich wątków do bariery należy użyć funkcji:

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

Funkcja ta tylko dla jednego z wątków zwraca wartość różną od zera i jest to **PTHREAD_BARRIER_SERIAL_THREAD**. Pozostałe zwracane wartości oznaczają błąd. Po zwrocie tej wartości bariera znów nadaje się do użycia (z ostatnio inicjowaną liczbą wątków w niej uczestniczących).

Bariery (slajd 1) Bariery.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <time.h>
#include <string.h>
#include <x86intrin.h>
#define N 8*1024*1024
#define W 4
pthread_barrier_t bar[W/2];
pthread_t id[W];
typedef struct{
    int* wsk;
    int* wsk2;
    int* pom;
}tablica;
tablica t;
void sortowanie_scalanie(int tab[],int pom[],int i_p, int i_k){
    int i_s=(i_p + i_k)>>1,i1=i_p,i2=i_s,;
    if(i_s-i_p>1) sortowanie_scalanie(tab,pom,i_p,i_s);
    if(i_k-i_s>1) sortowanie_scalanie(tab,pom,i_s,i_k);
    for(int i=i_p;i<i_k;i++)
        pom[i]=((i1==i_s)||((i2<i_k)&&(tab[i1]>tab[i2])))?tab[i2++]:
    tab[i1++];
    memcpy(tab+i_p,pom+i_p,(i_k-i_p)*sizeof(int));
}
```

Bariery (slajd 2)

```
void* sortuj(void* _arg){
    int ktory=(int)_arg, bity=1,i_s,i1,i2,nrbar;
    int i_p=(ktory*N)/W, i_k=((ktory+1)*N)/W, *tab=t.wsk,*pom=t.pom;
    sortowanie_scalanie(tab,t.pom,i_p,i_k);
    while(bity<W){
        nrbar=(ktory/(bity*2))*bity;
        switch(pthread_barrier_wait(&bar[nrbar])){
            case 0:
            case PTHREAD_BARRIER_SERIAL_THREAD:
                break;
            default:
                puts("b");
        }
        if((ktory&bity)==0){
            i1=i_p; i_s=i_k; i2=i_s;
            i_k=((2*bity+ktory)*N)/W;
            for(int i=i_p;i<i_k;i++)
                pom[i]=((i1==i_s)||((i2<i_k)&&(tab[i1]>tab[i2])))?tab[
    i2++]:tab[i1++];
            memcpy(tab+i_p,pom+i_p,(i_k-i_p)*sizeof(int));
            bity<<=1;
        }else break;
    }
    return 0;
}
```

Bariery (slajd 3)

```
int main(){
    int j; unsigned long long l1,l2;
    t.wsk=malloc(N*sizeof(int)); t.wsk2=malloc(N*sizeof(int));
    t.pom=malloc(N*sizeof(int));
    for(j=0;j<W/2;j++)
        if(pthread_barrier_init(&(bar[j]),0,2))
            printf("Nie utworzono bariery %d\n",j);
    if(!t.wsk || !t.wsk2 || !t.pom){ puts("Za malo pamieci");
        free(t.wsk); free(t.wsk2); free(t.pom); exit(1); }
    srand(time(0));
    for(j=0;j<N;j++){ t.wsk[j]=rand(); t.wsk2[j]=t.wsk[j]; }
l1=__rdtscp(&j);
    for(j=0;j<W;j++)
        if(pthread_create(&id[j], NULL, sortuj, (void*)j)){
            printf("Nie utworzono watku %d",j); exit(4); }
    for(j=0;j<W;j++)
        if(pthread_join(id[j],NULL)) printf("Blad watku %d",j);
l2=__rdtscp(&j); printf("Wielowatkowe sortowanie %llu\n",l2-l1);
l1=__rdtscp(&j);
    sortowanie_scalanie(t.wsk2,t.pom,0,N);
l2=__rdtscp(&j); printf("Jednowatkowe sortowanie %llu\n",l2-l1);
    free(t.pom); free(t.wsk); free(t.wsk2);
    for(j=0;j<W/2;j++) pthread_barrier_destroy(&bar[j]);
    return 0;
}
```

Semafory

Semafory są podobne do mutexów, przy czym tutaj możemy zadecydować ile wątków jednocześnie może mieć dostęp do chronionego obszaru. Ogólnie rzecz biorąc semafor utożsamiany jest z liczbą nieujemną o pewnej wartości początkowej. Każdy dostęp do niego zmniejsza tą wartość o 1, a zwolnienie dostępu zwiększa o 1, przy czym dostęp jest możliwy gdy wartość ta nie jest zerem. Aby ich użyć należy dołączyć bibliotekę **semaphore.h**. Z semaforami związany jest typ **sem_t**. Zmienna tego typu najpierw musi być zainicjowana przy pomocy:

```
int sem_init(sem_t *semafor,int pshared,int value);
```

przy czym **pshared** może przyjmować wartości **PTHREAD_PROCESS_PRIVATE** (domyślny, bez współdzielenia) lub **PTHREAD_PROCESS_SHARED** (współdzielenie między procesami), a **value** oznacza liczbę wątków mających jednoczesny dostęp do obszaru (musi być większe niż zero! i nie przekraczać **SEM_VALUE_MAX**). Gdy semafor już przestaje nam potrzebny należy go zwolnić za pomocą:

```
int sem_destroy(sem_t *semafor);
```

Powyższe funkcje zwracają 0 w przypadku sukcesu, w przypadku błędu zwracana jest -1.

Blokowanie semafora podobne jak dla mutexów może odbywać się na trzy sposoby:

```
int sem_wait(sem_t *semafor);// czeka do skutku
int sem_trywait(sem_t *semafor);// nie czeka
int sem_timedwait(sem_t *semafor, const struct timespec *abstime);
    // czeka co najwyżej podany czas
```

Do odblokowywania (właściwie zwiększania licznika semafora) służy funkcja:

```
int sem_post(sem_t *semafor);
```

Możemy również sprawdzić aktualny licznik semafora za pomocą

```
int sem_getvalue(sem_t *semafor,int *value);
```

Powyższe funkcje zwracają 0 w przypadku sukcesu (dla funkcji blokujących zmniejszany jest wówczas licznik semafora, a dla odblokowujących zwiększany o 1), w przypadku błędu zwracana jest -1 (licznik semafora nie ulega wtedy zmianie), kod błędu można uzyskać z **errno** (tak jak zwykle **perror** może nam wyświetlić szczegóły). Błędy, które mogą się tu pojawić, to **EINTR** - przerwane wywołanie przez obsługę sygnałów*, **EINVAL** - błędny semafor (lub czas dla blokowania z limitem czasu), **EAGAIN** - dla wersji blokowania bez czekania, oznacza zablokowany semafor, **ETIMEDOUT** - upłynął podany czas oczekiwania na blokowanie, **EOVERFLOW** - odblokowanie osiągnęło maksymalną dostępną wartość (**SEM_VALUE_MAX**).

Semafory.c

Podsumowanie wątków

- Blokowanie innych wątków:
  - mutex (pojedynczy dostęp);
  - semafor (dostęp kilku naraz - sami decydujemy ilu);
  - blokady zapisu/odczytu (dostęp wielu naraz przy odczycie, pojedynczy dostęp przy zapisie).
- Informowanie innych wątków o zakończeniu pewnych działań - zmienne warunkowe (+mutex).
- Oczekiwanie na zakończenie działań przez kilka wątków - bariera.

Typy atomowe (C11)

W przypadku programów wielowątkowych mamy pewne problemy z dostępem do tej samej zmiennej. W prosty sposób możemy sobie zagwarantować, że w zmiennej tej będą zapyswane wszystkie wyniki.

Inkrementacja w dwóch wątkach dla zmiennej

nieatomowej:

wątek 1	wątek 2	zmienna
odczyt		0
modyfikacja	odczyt	0
zapis	modyfikacja	1
	zapis	1
		1
		1

atomowej:

wątek 1	wątek 2	zmienna
odczyt		0
modyfikacja		0
zapis		1
	odczyt	1
	modyfikacja	1
	zapis	2

Żeby uczynić zmienną atomową należy użyć składni: atomic.c

```
_Atomic typ nazwa_zmiennej;
// lub
_Atomic(typ) nazwa_zmiennej;
// można też skorzystać z gotowych typów z biblioteki stdatomic.h, np.
atomic_int nazwa_zmiennej;
```

Sygnały

Program może otrzymywać pewne sygnały o występujących zdarzeniach (np. zamykanie programu). Możemy te sygnały odbierać za pomocą własnych funkcji. Potrzebna do tego nam będzie biblioteka **signal.h**. Mamy w niej dwie funkcje:

```
void (*signal(int sig, void (*func)(int)))(int);
```

która wiąże z sygnałem **sig** funkcję, która go obsługuje. Zamiast własnej funkcji można tutaj podać pewne automatyczne funkcje - domyślną **SIG_DFL** lub ignorującą **SIG_IGN**. Sygnał **sig** może być również jednym z domyślnych sygnałów

sygnał	znaczenie
SIGABRT	Nienormalne zakończenie działania programu (np. funkca **abort**)
SIGFPE	Niedozwolona operacja arytmetyczna
SIGILL	Niedozwolona instrukcja procesora
SIGINT	Przerwanie procesu
SIGSEGV	Naruszenie ochrony pamięci
SIGTERM	Do programu zostało wysłane żadanie jego zakończenia

Funkcja **signal** zwraca wskaźnik na poprzednią funkcję obsługującą dany sygnał, a jeśli się nie powiedzie to zwraca **SIG_ERR**.

Drugą funkcją jest

```
int raise (int sig);
```

która wysyła sygnał **sig** do naszego programu. Zwraca ona 0 w przypadku powodzenia, lub inną wartość w przypadku błędu.
W bibliotece **signal.h** mamy jeszcze typ **sig_atomic_t**, który może być przydatny do współdzielenia zmiennej całkowitej (atomowej).

Przykład

signals.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#ifdef __linux__
#include <time.h>
void ms_sleep(unsigned czas){
    struct timespec req;
    req.tv_sec = (czas / 1000);
    req.tv_nsec = (czas % 1000 * 1000000);
    nanosleep(&req, NULL);
}
#elif _WIN32
#include <windows.h>
void ms_sleep(unsigned czas){
    Sleep(czas);
}
#endif
void sighandler(int signum) {
    printf("Zlapano sygnal %d, konczenie...\n", signum);
    exit(1);
}
int main () {
    signal(SIGINT, sighandler);
    printf("Nacisnij ctrl+c by wyjsc\n");
    while(1) {
        printf("Sleep(1s)...\n");
        ms_sleep(1000);
    }
    return(0);
}
```