Wątki

- Wątek (ang, *thread*) jest to niezależna sekwencja zdarzeń w obrębie procesu. Podczas wykonywania procesu równolegle i niezależnie od siebie może być wykonywanych wiele wątków.
- Każdy wątek jest wykonywany ściśle sekwencyjnie, ma własny licznik rozkazów oraz stos, może tworzyć wątki pochodne, współdzieli się czasem procesora.
- Wszystkie wątki mają jednak tę samą przestrzeń adresową, co oznacza, że mają wspólne zmienne globalne, każdy wątek ma dostęp do stosu innego wątku - może go czytać lub zapisywać. Wszystkie wątki mają ten sam zbiór otwartych plików.
- Aby zapobiec dostępowi do tych samych danych przez wiele wątków, trzeba zastosować odpowiednią ochronę - wprowadzić synchronizację wątków.
- Biblioteka wątków w Linuksie
  - istnieje wiele bibliotek wątków
  - będziemy korzystać z biblioteki opartej o standard Posix.1.c
  - kompilacja: `gcc pliki_kompilowane -lpthread`

Tworzenie wątków

```
#include <pthread.h>
pthread_t thread;
int pthread_create(pthread_t *thread, pthread_attr_t * attr,
                   void *(*func)(void *), void *arg);
pthread_t pthread_self(void);
int pthread_join(pthread_t *thread, void **thread_return);
int pthread_detach(pthread_t *thread);
void pthread_exit(void *thread_return);
```

- Zmienna typu **`pthread_t`** służy do przechowywania identyfikatora wątku.
- Funkcja **`pthread_create`** tworzy nowy wątek.

  Argumenty:
  - *`thread`* - identyfikator wątku; w razie pomyślnego utworzenia wątku funkcja umieszcza w tej zmiennej identyfikator przypisany wątkowi; jeśli nie interesuje nas identyfikator wątku, należy przekazać wskaźnik pusty `NULL`;
  - *`attr`* - atrybuty wątku, ustawianie atrybutów dokonywane jest za pomocą odpowiedniej funkcji – patrz `pthread_attr_init()`. Jeśli chce się pozostawić wartości domyślne atrybutów, należy przekazać wskaźnik pusty `NULL`,
  - *`func`* – adres początkowej funkcji wątku; wątek kończy działanie z chwilą, kiedy następuje powrót z tej funkcji; wynik funkcji jest stanem końcowym wątku
  - *`arg`* - argument przekazywany do funkcji początkowej wątku

  Wartość zwracana:
  - =0 - wątek został poprawnie utworzony
  - >0 - wystąpił błąd; wartość oznacza kod błędu (kody błędów umieszczone są w `sys/errno.h`)
- Funkcja **`pthread_self`** zwraca identyfikatora bieżącego wątku
- Funkcja **`pthread_join`** służy do wcielenie wątku do innego wątku. Funkcja czeka na zakończenie wątku o podanym identyfikatorze ***`thread`*** . Jeśli nas nie interesuje wartość zwracana przez wątek, należy przekazać do funkcji `NULL`. Wartość zwracana przez wątek jest umieszczana w zmiennej ***`thread_return`***.

  Wartość zwracana:
  - =0 - funkcja zakończyła się poprawnie
  - >0 - wystąpił błąd; wartość oznacza kod błędu.
- Funkcja **`pthread_detach`** odłącza wątek o identyfikatorze *`thread`*. Oznacza to, że w momencie, kiedy wątek zostanie zakończony wszystkie związane z nim zasoby automatycznie zostaną zwrócone do systemu.
- Funkcja **`pthread_exit`** kończy wątek. W zmiennej *`thread_return`* umieszczona zostanie wartość zwracana przez wątek do wątku, który wcieli ten wątek za pomocą funkcji **`pthread_join`**. Wartość ta jest zwracana tylko w przypadku wątków nieodłączonych. Wątek zwalnia swoje zasoby (swoje, nie procesu). Jeśli funkcja początkowa wątku się zakończy bez wywoływania **`pthread_exit`** , to system sam automatycznie wywoła tę funkcję.

- Przykład:

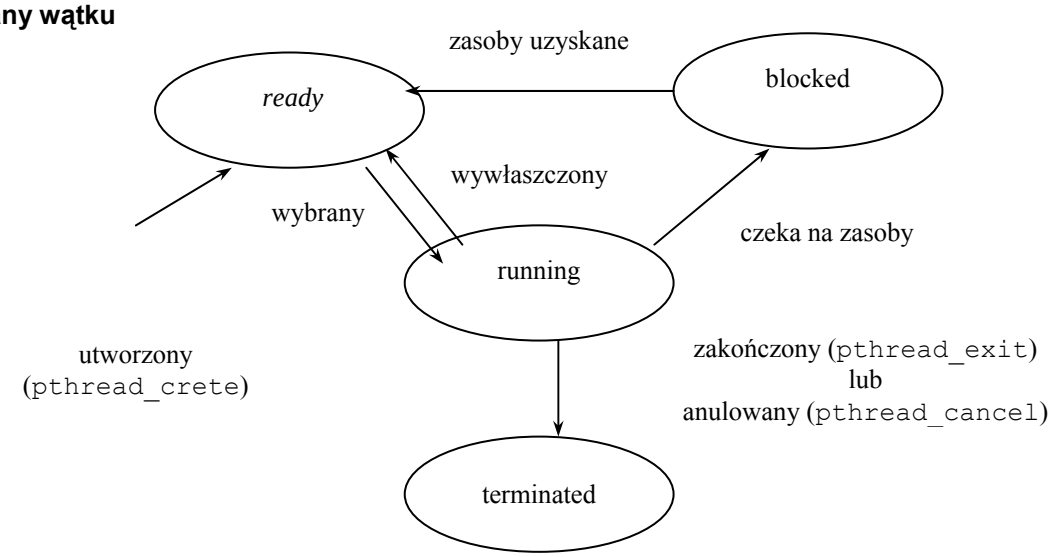
```
#include <stdio.h>
#include <pthread.h>

/* Obsługa błędów */
#define err_abort(code, text) do { \
  fprintf(stderr, "%s w \"%s\":%d: %s\n",\
          text, __FILE__, __LINE__, strerror(code)); \
  abort();\
  } while(0);

/* Funkcja początkowa wątku */
void *thread_routine (void *arg) {
  return arg;
}

/* Funkcja główna wątku */
main (int argc, char *argv[]) {
  pthread_t thread_id;
  void *thread_result;
  int status;
  status = pthread_create (&thread_id, NULL, thread_routine, NULL);
  if (status != 0)
    err_abort (status, "Tworzenie wątku");
  status = pthread_join (thread_id, &thread_result);
  if (status != 0)
    err_abort (status, "Wcielanie wątku");
  if (thread_result == NULL)
    return 0;
  else
    return 1;
}
```

Stany wątku



- Wątek jest tworzony za pomocą funkcji `pthread_create`. Przyjmuje wtedy stan `ready` i czeka na przydzielenie procesora.
- Wątek jest kończony za pomocą funkcji `pthread_exit` (wywołanej w sposób jawny lub niejawny) lub `pthread_cancel`. Jeśli jest to wątek odłączony, jest on natychmiast odzyskiwany (ang. *recycling*). Jeśli nie, to przechodzi do stanu `terminated` i czeka aż jakiś wątek go wcieli. Nie zwalnia zasobów. Wątek, który czeka w funkcji `pthread_join` budzi się i pobiera wartość zwracaną przez kończony wątek, który może być teraz odłączony i odzyskany.
  Podczas odzyskiwania zwalniane są wszystkie zasoby systemowe, które jeszcze nie zostały zwolnione podczas kończenia wątku. Identyfikator wątku staje się nieaktualny i może być przydzielony innemu wątkowi.

- Przykład 1: tworzenie wątku (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 58 - program wypisuje na standardowe wyjście błędów znaki x)

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

/* Watek drukuje x */
void* drukuj_s (void* arg) {
  while (1)
    fputc ('x', stderr);
  return NULL;
}

int main () {
  pthread_t watek_id;
  /* Utworz nowy watek. Będzie on wykonywał funkcje drukuj_s  */
  pthread_create (&watek_id, NULL, &drukuj_s, NULL);
  /* Watek glowny drukuje o  */
  while (1)
    fputc ('o', stderr);
  return 0;
}
```

- Przykład 2: przekazywanie danych do wątku (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 59-61 - dwa wątki, jeden pisze literę x, drugi literę o)

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

struct argumenty {
  char znak;
  int licznik;
};
void* drukuj_znak (void* arg) {
  struct argumenty* p = (struct argumenty*) arg;
  int i;
  for (i = 0; i < p->licznik; ++i)
    fputc (p->znak, stderr);
  return NULL;
}
int main () {
  pthread_t watek1_id;
  pthread_t watek2_id;
  struct argumenty watek1_arg;
  struct argumenty watek2_arg;

  /* Utworz nowy watek, który drukuje x 30 razy  */
  watek1_arg.znak = 'x';
  watek1_arg.licznik = 30;
  pthread_create (&watek1_id, NULL, &drukuj_znak, &watek1_arg);

  /* Utworz nowy watek, który drukuje o 20 razy  */
  watek2_arg.znak = 'o';
  watek2_arg.licznik = 20;
  pthread_create (&watek2_id, NULL, & drukuj_znak, &watek2_arg);

  /* Czekaj na zakończenie pierwszego wątku  */
  pthread_join (watek1_id, NULL);
  /* Czekaj na zakończenie pierwszego wątku  */
  pthread_join (watek2_id, NULL);

  return 0;
}
```

- Przykład 3: wartość zwracana z wątku (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 61-62)

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

/* Wyznacz n kolejnych liczb pierwszych (bardzo nieefektywny algorytm)
   Zwróć n-tą liczbę pierwszą, gdzie n jest wartością wskazywaną przez *arg
*/

void* liczba_pierwsza (void* arg){
  int kandydat = 2;
  int n = *((int*) arg);
  while (1) {
    int liczba;
    int jest_pierwsza = 1;
    /* Sprawdz, czy jest to liczba pierwsza  */
    for (liczba = 2; liczba < kandydat; ++liczba)
      if (kandydat % liczba == 0) {
        jest_pierwsza = 0;
        break;
      }
    /* Czy jest to szukana liczba pierwsza?  */
    if (jest_pierwsza) {
      if (--n == 0)
        /* Zwróc liczbę jako wynik wykonania wątku  */
        return (void*) kandydat;
    }
    ++kandydat;
  }
  return NULL;
}

int main () {
  pthread_t watek;
  int ktora_pierwsza = 5000;
  int pierwsza;

  pthread_create (&watek, NULL, &liczba_pierwsza, &ktora_pierwsza);
  pthread_join (watek, (void*) &pierwsza);
  printf("%d-ta liczba pierwsza to %d.\n", ktora_pierwsza, pierwsza);
  return 0;
}
```

Atrybuty wątku

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

- Atrybuty decydują o tym jak zachowują się wątki. Podstawowe atrybuty:

Atrybut	Domyślna wartość	Opis
detachstate	`PTHREAD_CREATE_JOINABLE`	Wątek może być wątkiem przyłączanym (ang. *joinable*) - może wtedy zostać wcielony przez inny wątek (dopiero wtedy jego zasoby będą zwolnione) lub wątkiem odłączonym (ang. *detached*).
schedpolicy	`SCHED_OTHER`	Przełączanie wątków jest realizowane przez system.
schedparam	`0`	Parametry przełączania wątków. Nie ma znaczenia, gdy wybrana polityka `SCHED_OTHER`
inheritsched	`PTHREAD_EXPLICIT_SCHED`	Sposób wyznaczania polityki przełączania wątków: jawnie za pomocą parametrów *schedparam* i *inheritsched* (**domyślny**) lub poprzez dziedziczenie z wątku macierzystego.
scope	`PTHREAD_SCOPE_SYSTEM`	Sposób konkurowania o zasoby procesora: wątek może konkurować z wątkami z innych procesów (domyślny) lub między wątkami w obrębie tego samego procesu.

- Zmienna typu **`pthread_attr_t`** służy do przechowywania atrybutów wątku.
- Funkcja **`pthread_attr_init`** pozwala zainicjować zmienną opisującą atrybuty. Jeśli zwróci 0, oznacza to, że zakończyła się pomyślnie i wskazana zmienna została zainicjowana wartościami domyślnymi. Wartość większa od zera oznacza kod błędu.
- Każdy z atrybutów ma przypisaną funkcję **`pthread_attr_setxxx`** - do ustawienia wartości atrybutu i funkcję **`pthread_attr_setxxx`** za pomocą której można odczytać atrybut. Funkcje te zwracają 0, jeśli czynność się powiedzie, w przeciwnym wypadku zwracają kod błędu.
- Funkcja **`pthread_attr_setdetachstate`** pozwala ustawić stan wykonywanego wątku. Argument *`detachstate`* może przyjmować dwie wartości:

  `PTHREAD_CREATE_JOINABLE` Wątek musi być wcielony.
  `PTHREAD_CREATE_JOINABLE` Wątek jest odłączony. Nie może być wcielany ani anulowany.
- Funkcja **`pthread_attr_getdetachstate`** pozwala pobrać atrybut związany z aktualnym stanem wykonywanego wątku.
- Funkcja **`pthread_attr_destroy`** usuwa zmienną z atrybutami. Nie ma wpływu na wątku utworzone z atrybutami zawartymi w usuwanej zmiennej.
- Przykład: Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str.64

```
#include <pthread.h>
void* funkcja_watku (void* thread_arg) {
  /* działania wykonywane w wątku...  */
  return NULL;
}

int main () {
  pthread_attr_t atrybut;
  pthread_t watek;
  pthread_attr_init (&atrybut);
  pthread_attr_setdetachstate (&atrybut, PTHREAD_CREATE_DETACHED);
  pthread_create (&watek, & atrybut, &funkcja_watku, NULL);
  pthread_attr_destroy (&atrybut);
  /* Działania wykonywane w wątku głównym...  */
/* Nie trzeba czekać na zakończenie wywołanego wątku.  */
  return 0; }
```

Dane własne wątku

```
pthread_key_t key;
int pthread_key_create (pthread_key_t *key, void (*destructor)(void *));
int pthread_setspecific (pthread_key_t key, const void *value);
void *pthread_getspecific (pthread_key_t key);
int pthread_key_delete (pthread_key_t key);
```

- Wątki mogą mieć swoje własne dane (ang. *thread specific data*). Dane takie mają te same nazwy w każdym z wątków, ale ich wartości są różne. Dostęp do tych danych jest realizowany za pomocą wskaźnika właściwego wątkowi oraz skojarzonego z nim klucza.
- Zmienna typu **`pthread_key_t`** służy do przechowywania klucza.
- Funkcja **`pthread_key_create`** służy do utworzenia klucza danych własnych wątku. Jest on wspólny dla wszystkich wątków. Pierwszym argumentem funkcji jest wskaźnik do zmiennej klucza. Funkcja przypisuje kluczowi wartość `NULL`. Drugim argumentem jest nazwa funkcji, która będzie automatycznie wywołana wtedy, kiedy zakończy się wątek z kluczem różnym od `NULL`. Każdy klucz może być utworzony tylko raz.
- Funkcja **`pthread_setspecific`** przypisuje wskazane dane (`*value`) wątkowi i kojarzy je z kluczem.
- Funkcja **`pthread_getspecific`** zwraca wartość klucza.
- Funkcja **`pthread_key_delete`** usuwa klucz danych własnych wątku.
- Przykład: Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str.68-69, każdy z wątków ma mieć swój własny plik dziennika.

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

static pthread_key_t klucz_do_dziennika;

/* te funkcje będą używane przez wszystkie wątki,
   ale każda z nich ma obsługiwać własny plik wątku */
void zapisz_do_dziennika_watku(const char* komunikat) {
  FILE* dziennik = (FILE*)pthread_getspecific(klucz_do_dziennika);
  fprintf (dziennik, "%s\n", komunikat);
}

void zamknij_dziennik_watku (void* dziennik) {
  fclose ((FILE*) dziennik);
}

void* funkcja_watku (void* arg) {
  char nazwa_pliku_dziennika[20];
  FILE* dziennik;
  sprintf (nazwa_pliku_dziennika, "watek%d.log", (int) pthread_self());
  dziennik = fopen (nazwa_pliku_dziennika, "w");
  pthread_setspecific (klucz_do_dziennika, dziennik);
  zapisz_do_dziennika_watku("Watek rozpoczety.");
  /* ..... */
  return NULL;
}

int main () {
  int i;
  pthread_t watki[5];

  pthread_key_create (&klucz_do_dziennika, zamknij_dziennik_watku);
  for (i = 0; i < 5; ++i)
    pthread_create (&(watki[i]), NULL, funkcja_watku, NULL);
  for (i = 0; i < 5; ++i)
    pthread_join (watki[i], NULL);
  return 0;
}
```

Synchronizacja

- Przykład: program obsługuje zadania umieszczone w kolejce, zadania są przetwarzane przez wiele współbieżnych wątków. Kolejka zadań jest implementowana za pomocą listy dowiązaniowej (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 72-86)

Rozwiązanie 1. (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 73)

```
#include <malloc.h>

/* Elementy listy */
struct zadanie {
  struct zadanie* nastepny;
  /* Pozostałe pola ... */
};

/* Lista dowiązaniowa zadań do wykonania.  */
struct zadanie* kolejka_zadan;

extern void przetwarzaj_zadanie (struct zadanie*);

void* funkcja_watku (void* arg)
{
  while (kolejka_zadan != NULL) {
    struct zadanie* nast_zadanie = kolejka_zadan;
    kolejka_zadan = kolejka_zadan->nast;
    przetwarzaj_zadanie (nast_zadanie);
    free (nast_zadanie);
  }
  return NULL;
}
```

Problem: Załóżmy, że w kolejce pozostało jedno zadanie. Pierwszy wątek sprawdza, czy kolejka jest już pusta. Stwierdza, że jest zadanie do przetworzenia zapamiętuje więc wskaźnik do elementu kolejki w zmiennej `nast_zadanie`. W tym momencie system wywłaszcza ten wątek i przydziela czas następnemu wątkowi. Drugi wątek również sprawdza, czy w kolejce są zadania do wykonania i również zaczyna przetwarzać to samo zadanie.

- Aby wyeliminować tego typu wyścigi musimy pewne operacje uczynić *niepodzielnymi* (ang. *atomic*). Tego typu operacja gdy się rozpocznie nie może być przerwana dopóty, dopóki się nie zakończy.

Zmienne muteksowe

- Muteks jest to specjalny rodzaj semafora dwustanowego pozwalającego na tworzenie blokad wzajemnie wykluczających się (ang. *mutual exlusion*). Muteks zapewnia synchronizację.

```
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int  (pthread_mutex_t *mutex);
```

- Zmienna typu **`pthread_mutex_t`** to zmienna przyjmująca dwa stany: otwarty (nie jest zajęta przez żaden wątek) i zamknięty (jest zajęta przez jakiś wątek). Jeśli jeden watek zamknie zmienną muteksową a następnie drugi wątek próbuje zrobić to samo, to druku wątek zostaje zablokowany do momentu, kiedy pierwszy wątek nie otworzy zmiennej muteksowej. Dopiero wtedy może wznowić działanie.
- Muteks można inicjalizować:
  - za pomocą stałej, na przykład **`PTHREAD_MUTEX_INITIALIZER`** (wątek typu `fast`, domyślny).
  - za pomocą funkcji **`pthread_mutex_init`** ; muteks ma swoje atrybuty, które można przekazać za pomocą drugiego argumentu funkcji; `NULL` oznacza atrybuty domyślne
- Funkcja **`pthread_mutex_lock`** zamyka muteks. Jeśli jest on w tym momencie otwarty, jest zamykany i staje się własnością wątku, który go zamknął. Wątek ten wychodzi od razu z funkcji. Jeśli jednak muteks jest zamknięty, działanie funkcji zależy od typu muteksu. W przypadku muteksu typu `fast`, wątek jest zawieszony do momentu otworzenia muteksu. Prowadzić to może do zakleszczania.
- Funkcja **`pthread_mutex_unlock`** otwiera muteks. Jeśli jest to muteks typu fast, funkcja zawsze go przywraca do stanu otwartego.
- Funkcja **`pthread_mutex_trylock`** działa podobnie do funkcji **`pthread_mutex_lock`**. Jeśli jednak muteks jest zamknięty, nie blokuje wątku, ale wraca z kodem błędu `EBUSY`.
- Funkcja **`pthread_mutex_destroy`** usuwa zmienną muteksową i zwalnia zajmowaną przez nią pamięć.

- **Rozwiązanie 2.** Obsługa zadań z kolejki chroniona przez muteks (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 75)

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

struct zadanie {
  struct zadanie* nastepny;
  /* Pozostałe pola składowe ... */
};
struct zadanie* kolejka_zadan;
extern void przetwarzaj_zadanie (struct zadanie*);
pthread_mutex_t muteks_kolejki=PTHREAD_MUTEX_INITIALIZER;

void* funkcja_watku (void* arg){
  while (1) {
    struct zadanie* nast_zadanie;
    pthread_mutex_lock (&muteks_kolejki);
    if (kolejka_zadan == NULL) nast_zadanie = NULL;
    else {
      nast_zadanie = kolejka_zadan;
      kolejka_zadan = kolejka_zadan->nastepny;
    }
    pthread_mutex_unlock (&muteks_kolejki);
    if (nastepne_zadanie == NULL) break;
    przetwarzaj_zadanie(nast_zadanie);
    free (nast_zadanie);
  }
  return NULL; }
```

Semafory

```
#ifdef POSIXSEMAPHORES
      int sem_init(sem_t *sem, int pshared, unsigned int value);
      int sem_wait(sem_t *sem);
      int sem_post(sem_t *sem);
      int sem_trywait(sem_t *sem);
      int sem_getvalue(sem_t *sem, int *svalue);
      int sem_destroy(sem_t *sem);
#endif
```

- Program z kolejką zadań: co będzie, jeśli zadania nie będą umieszczane w kolejce dostatecznie szybko? Kolejka stanie się pusta i wątki się zakończą. Rozwiązanie: zastosowanie semafora.
- Semafor ma licznik.
- Semafor jest reprezentowany za pomocą zmiennej typu **`sem_t`**. Jest ona inicjowana za pomocą funkcji **`sem_init`**.
- Wątek, który chce czekać na semafor (aby na przykład zamknąć zasoby, lub czekać na zdarzenie) wywołuje funkcję **`sem_wait`**. Jeśli licznik semafora jest większy od zera, funkcja ta zmniejsza licznik i od razu wraca. Jeśli zaś licznik semafora jest równy 0, wątek zostaje zablokowany.
- Wątek, który chce ustawić semafor (odblokować zasoby, obudzić czekającego), wywołuje funkcję **`sem_post`**. Jeśli jeden lub więcej wątków czeka na semafor, funkcja **`sem_post`** budzi jeden wątek. Jeśli żaden z wątków nie czeka, funkcja zwiększa licznik .
- Funkcja **`sem_init`** ustawia wartość początkową licznika semafora (*`value`*). Wartość 1 spowoduje, że jeden wątek będzie mógł wykonać operację **`sem_wait`** bez blokowania i zamknie semafor. Wartość 0 spowoduje, że wszystkie wątki, które wywołają **`sem_wait`** zostaną zablokowane do momentu, w którym któryś z wątków nie wywoła **`sem_post`**.
- Semafor nie ma właściciela. Każdy wątek może zwolnić wątki czekające na semafor.
- Funkcja **`sem_getvalue`** zwraca bieżącą wartość licznika semafora, o ile nie ma wątków czekających na semafor. W przeciwnym wypadku zwraca liczbę ujemną, której wartość bezwzględna wskazuje ile wątków czeka na semafor.
- Funkcja **`sem_trywait`** próbuje zamknąć semafor. Jeśli wartość semafora jest większa od zera, zmniejszana jest ona o 1. Jeśli wartość semafora wynosi 0, następuje natychmiastowy powrót z funkcji z kodem `EAGAIN`.
- Funkcja **`sem_destroy`** zwalnia semafor.
- Funkcje semaforowe korzystają ze zmiennej `errno`, która jest ustawiana jeśli funkcja zwróci -1.

Rozwiązanie wersja 3. Kolejka zadań sterowana semaforem (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 79)

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

struct zadanie {
  struct zadanie * nastepny;
  /* Pozostałe pola składowe... */
};

struct zadanie* kolejka_zadan;

extern void przetwarzaj_zadanie(struct zadanie*);

pthread_mutex_t muteks_kolejki = PTHREAD_MUTEX_INITIALIZER;

sem_t licznik_kolejki;

void inicjalizuj_kolejke_zadan (){
  kolejka_zadan = NULL;
  sem_init(&licznik_kolejki, 0, 0);
}

void* funkcja_watku (void* arg){
  while (1) {
    struct zadanie* nast_zadanie;
    sem_wait (&licznik_kolejki);
    pthread_mutex_lock (&muteks_kolejki);
    nastepne_zadanie = kolejka_zadan;
    kolejka_zadan = kolejka_zadan->nastepny;
    pthread_mutex_unlock (&muteks_kolejki);
    przetwarzaj_zadanie (nast_zadanie);
    free (nast_zadanie);
  }
  return NULL;
}

void wstaw_zadanie_do_kolejki (/* dane zadania */)
{
  struct zadanie *nowe_zadanie;
  nowe_zadanie=(struct zadanie*) malloc(sizeof (struct zadanie));
/* wpisz dane do pozostałych pól struktury zadania */

  pthread_mutex_lock (&muteks_kolejki);

/* umieść zadanie na początku kolejki */
  nowe_zadanie->nastepny = kolejka_zadan;
  kolejka_zadan = nowe_zadanie;

/* prześlij informację do semafora o nowym zadaniu */
  sem_post (&licznik_kolejki);
/* otwórz muteks */
  pthread_mutex_unlock (&muteks_kolejki);
}
```

Zmienne warunkowe

- Zmienne warunkowe dostarczają mechanizmu, który pozwala czekać aż pewien współdzielony zasób osiągnie pożądany stan lub do sygnalizowania, że osiągnął już stan, którym może być zainteresowany inny wątek. Na przykład kolejka nie jest już pusta lub właśnie została opróżniona.
- Każda zmienna warunkowa jest związana z muteksem, który chroni stan zasobu.

```
pthread_cond_t cond=PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *condattr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex,
                           struct timespec *expiration);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

- Zmienna warunkowa jest reprezentowana przez zmienną typu `pthread_cond_t`.
- Zmienną warunkową można inicjalizować:
  - za pomocą stałej, na przykład `PTHREAD_COND_INITIALIZER`
  - za pomocą funkcji `pthread_cond_init` ; zmienna warunkowa ma swoje atrybuty, które można przekazać za pomocą drugiego argumentu funkcji; `NULL` oznacza atrybuty domyślne
- Funkcja `pthread_cond_wait` czeka na zmienną warunkową. Wątek jest budzony za pomocą sygnału lub rozgłaszania. Funkcja odryglowuje muteks przed rozpoczęciem czekania, oraz zaryglowuje go po zakończeniu czekania (nawet jeśli zakończy się ono niepowodzeniem lub zostanie anulowane), przed powrotem do funkcji, z której ją wywołano.
- Funkcja `pthread_cond_timedwait` czeka na zmienną warunkową do otrzymania sygnału pojedynczego lub metodą rozgłaszania. Kończy się jednak również wtedy, kiedy przekroczony zostanie czas podany w parametrze *`expiration`*.
- Funkcja `pthread_cond_signal` ustawia zmienną warunkową *`cond`*, co powoduje obudzenie jednego z wątków. Korzysta się z niej wtedy kiedy tylko jeden wątek ma być obudzony.
- Funkcja `pthread_cond_broadcast` rozgłasza ustawienie zmiennej warunkowej *`cond`*.
- Funkcja `pthread_cond_destroy` usuwa zmienną warunkową.
- Każda zmienna warunkowa jest chroniona za pomocą muteksu.

- Przykład. Prosta implementacja zmiennej warunku (Mitchell, Oldham, Samuel: Linux Programowanie dla zaawansowanych, str. 81)

```
#include <pthread.h>

extern void wykonaj_prace ();

int flaga_watku;
pthread_mutex_t flaga_muteksu;

void initialize_flag (){
  pthread_mutex_init (&flaga_muteksu, NULL);
  flaga_watku = 0;
}

void* watek (void* thread_arg){
  while (1) {
    int flaga_ustawiona;

    /* Chron flage za pomoca muteksu  */
    pthread_mutex_lock (&flaga_muteksu);
    flaga_ustawiona = flaga_watku;
    pthread_mutex_unlock (&flaga_muteksu);

    if (flaga_ustawiona)
      wykonaj_prace ();
    /* W przeciwnym wypadku nic nie rób  */
  }
  return NULL;
}

void ustaw_flage_watku (int wartosc_flagi){
  /* Chron flage za pomoca muteksu  */
  pthread_mutex_lock (&flaga_watku_mutex);
  flaga_watku = wartosc_flagi;
  pthread_mutex_unlock (&flaga_muteksu);
}
```