Problemy czytelników i pisarzy oraz 5 ucztujących filozofów

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/
oraz
na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

Czytelnicy i pisarze (1)

- W systemie działa C>0 procesów, które odczytują pewne dane oraz P>0 procesów, które zapisują te dane. Procesy zapisujące będziemy nazywać pisarzami, a procesy odczytujące --- czytelnikami, zaś moment, w którym procesy mają dostęp do danych, będziemy nazywać pobytem w czytelni.

```
process Czytelnik;
begin
 repeat
   własne_sprawy;
   protokół_wstępny_czytelnika;
   CZYTANIE;
   protokół_końcowy_czytelnika
 until false
end
```

```
process Pisarz;
begin
 repeat
   własne_sprawy;
   protokół_wstępny_pisarza;
   PISANIE;
   protokół_końcowy_pisarza
 until false
end
```

- Zauważmy, że jednocześnie wiele procesów może odczytywać dane. Jednak jeśli ktoś chce te dane zmodyfikować, to rozsądnie jest zablokować dostęp do tych danych dla wszystkich innych procesów na czas zapisu. Zapobiegnie to odczytaniu niespójnych informacji (na przykład danych częściowo tylko zmodyfikowanych).

Czytelnicy i pisarze (2)

- Należy tak napisać protokoły wstępne i końcowe poszczególnych procesów, aby:
    1. Wielu czytelników powinno mieć jednocześnie dostęp do czytelni.
    2. Jeśli w czytelni przebywa pisarz, to nikt inny w tym czasie nie pisze ani nie czyta.
    3. Każdy czytelnik, który chce odczytać dane, w końcu je odczyta.
    4. Każdy pisarz, który chce zmodyfikować dane, w końcu je zapisze.
- Rozpatruje się różne warianty tego problemu:
    1. W czytelni może przebywać dowolnie wielu czytelników.
    2. Czytelnia może mieć ograniczoną pojemność.
    3. Pisarze mogą mieć pierwszeństwo przed czytelnikami (ale wtedy rezygnujemy z żywotności czytelników)
    4. Czytelnicy mogą mieć pierwszeństwo przed pisarzami (ale wtedy rezygnujemy z żywotności pisarzy)

Rozwiązanie I – uprzywilejowanie czytelników

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

void *writer_thread_fun(void *arg) {

   printf("Writer %d started...\n",(int) ((int *) arg));
   sleep(1);
   while(1)
   {  //pthread_mutex_lock(&wsem);
      sem_wait(&wsem);
         shared_data++;
      //pthread_mutex_unlock(&wsem);
      sem_post(&wsem);
      sleep(3);
   }
   pthread_exit(0);
}
```

```
void *reader_thread_fun(void *arg) {
   int data_read;
   printf("Reader %d started...\n",(int) ((int *) arg));
   sleep(1);
   while(1)
   {  pthread_mutex_lock(&x);
         readcount++;
         if(readcount==1)   sem_wait(&wsem);
         printf("Readcount=%d\n",readcount);
      pthread_mutex_unlock(&x);
      data_read=shared_data;
      printf("Reader %d  consumed %d.\n",(int) ((int *)
arg),data_read);
      pthread_mutex_lock(&x);
         readcount--;
         printf("Readcount=%d\n",readcount);
         if(readcount==0) //pthread_mutex_unlock(&wsem);
                  sem_post(&wsem);
      pthread_mutex_unlock(&x);
      sleep(1);
   }
   pthread_exit(0);
}
// Dalej: powołanie czytelników i pisarzy + inicjalizacja
// semaforów
```

Uwagi

- Semafor *wsem* jest wykorzystywany do wymuszenia wzajemnego wykluczania.
- Dopóki jeden pisarz ma dostęp do współdzielonego obszaru danych, żaden pisarz ani żaden czytelnik nie może mieć do niego dostępu.
- Proces czytelnika wykorzystuje semafor *wsem*, by wymusić wzajemne wykluczenie.
- Jednakże aby umożliwić dostęp wielu czytelników, wymaga się, by w sytuacji, gdy żaden czytelnik nie odczytuje danych, pierwszy czytelnik, który próbuje odczytać poczekał na semafor *wsem.*
- Kiedy przynajmniej jeden czytelnik odczytuje dane, kolejni czytelnicy nie muszą czekać, zanim uzyskają dostęp.
- Zmienna globalna *readcount* jest stosowana do śledzenia liczby czytelników, a semafor (mutex) x jest wykorzystywany, by upewnić się, że zmienna *readcount* jest poprawnie aktualizowana.
- Rozwiązanie ma wadę: W sytuacji gdy jeden czytelnik rozpoczął uzyskiwanie dostępu do danych, czytelnicy mogą kontrolować dane pod warunkiem, że przynajmniej jeden czytelnik odczytuje dane. Ro z kolei grozi **zagłodzeniem** pisarzy.

Rozwiązanie II – uprzywilejowanie pisarzy

```
void *writer_thread_fun(void *arg) {
   printf("Writer %d started...\n",(int)((int *) arg));
   sleep(2);
   while(1)
   {   pthread_mutex_lock(&y);
         writecount++;
         if(writecount==1) sem_wait(&rsem);
      pthread_mutex_unlock(&y);
      sem_wait(&wsem);
         shared_data++;
      sem_post(&wsem);
      pthread_mutex_lock(&y);
         writecount--;
         if(writecount==0) sem_post(&rsem);
      pthread_mutex_unlock(&y);
      sleep(3);
   }
   pthread_exit(0);
}
```

```
void *reader_thread_fun(void *arg) {
   int data_read;
   printf("Reader %d started...\n",(int)((int *) arg));  sleep(2);
   while(1)
   {  pthread_mutex_lock(&z);
         sem_wait(&rsem);
            pthread_mutex_lock(&x);
               readcount++;
               if(readcount==1) sem_wait(&wsem);
               printf("Readcount=%d\n",readcount);
            pthread_mutex_unlock(&x);
         sem_post(&rsem);
      pthread_mutex_unlock(&z);
      data_read=shared_data;
      printf("Reader %d  consumed %d.\n",(int)((int *)
arg),data_read);
      pthread_mutex_lock(&x);
         readcount--;
         printf("Readcount=%d\n",readcount);
         if(readcount==0) sem_post(&wsem);
      pthread_mutex_unlock(&x);
      //sleep(1);
   }
   pthread_exit(0);}

// Dalej: powołanie czytelników i pisarzy + inicjalizacja
// semaforów
```

Uwagi

- Rozwiązanie gwarantuje, że żadni nowi czytelnicy nie uzyskają dostępu do obszaru danych, jeśli przynajmniej jeden pisarz zadeklarował, że chce zrealizować operację zapisu.
- W przypadku pisarzy zostały dodane następujące semafory oraz zmienne:
  - Semafor *rsem,* który blokuje wszystkich czytelników, jeśli przynajmniej jeden pisarz spróbuje uzyskać dostęp do obszaru danych
  - Zmienna *writecount* kontrolująca ustawienia semafora *rsem*
  - Semafor y, który steruje aktualizacją zmiennej *writecount*
- W przypadku czytelników potrzebny jest dodatkowy semafor. Nie można dopuścić do powstania dużej kolejki na semaforze *rsem*, bowiem w przeciwnym razie pisarze nie będą w stanie wskoczyć do kolejki. Tak więc, tylko jeden czytelnik może się znaleźć w kolejce semafora *rsem*. Wszyscy dodatkowi czytelnicy muszą być skierowani do kolejki semafora z natychmiast przed oczekiwaniem na semafor *rsem*.

Blokady czytelników i pisarzy w POSIX (1)

- Problem czytelników i pisarzy jest na tyle powszechny, że w wielu systemach do programowania współbieżnego zostały zaproponowane specjalne blokady lub semafory wspierające konstruowanie oprogramowania realizującego ten problem
- Przykładem mogą być blokady czytelników i pisarzy POSIX:

Inicjacja blokady

```
int pthread_rwlock_init(pthread_rwlock_t * rwlock,
pthread_rwlockattr_t * attr)
```

  - **rwlock** Zadeklarowana i zainicjowana zmienna typu **pthread_rwlock_t**
  - **attr** Atrybuty blokady lub NULL gdy maja być domyślne

Zajęcie blokady do odczytu

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock)
```

- Wątek wykonujący funkcję blokuje się gdy blokada jest zajęta do zapisu.
- Zajmuje blokadę do odczytu gdy nie została już wcześniej zajęta do odczytu.

Blokady czytelników i pisarzy w POSIX (2)

Zajęcie blokady do zapisu

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

- Wątek wykonujący funkcję blokuje się gdy blokada jest zajęta do zapisu lub odczytu. Gdy nie jest zajęta to zajmuje blokadę do zapisu.

Zwolnienie blokady

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

Funkcja zdejmuje blokadę nałożoną jako ostatnią przez bieżący wątek. Jeżeli istnieją inne blokady założone na obiekt to pozostają. Jeżeli jest to ostatnia blokada i istnieją wątki czekające na jej zwolnienie to jeden z nich zostanie odblokowany. Wybór wątku do zwolnienia zależy to od implementacji.

Blokady czytelników i pisarzy w POSIX (3)

Nieblokujące zajęcie blokady do zapisu

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock)
```

- Gdy blokada jest wolna następuje jej zajęcie do zapisu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego i zwraca kod błędu.

Nieblokujące zajęcie blokady do odczytu

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t
*rwlock)
```

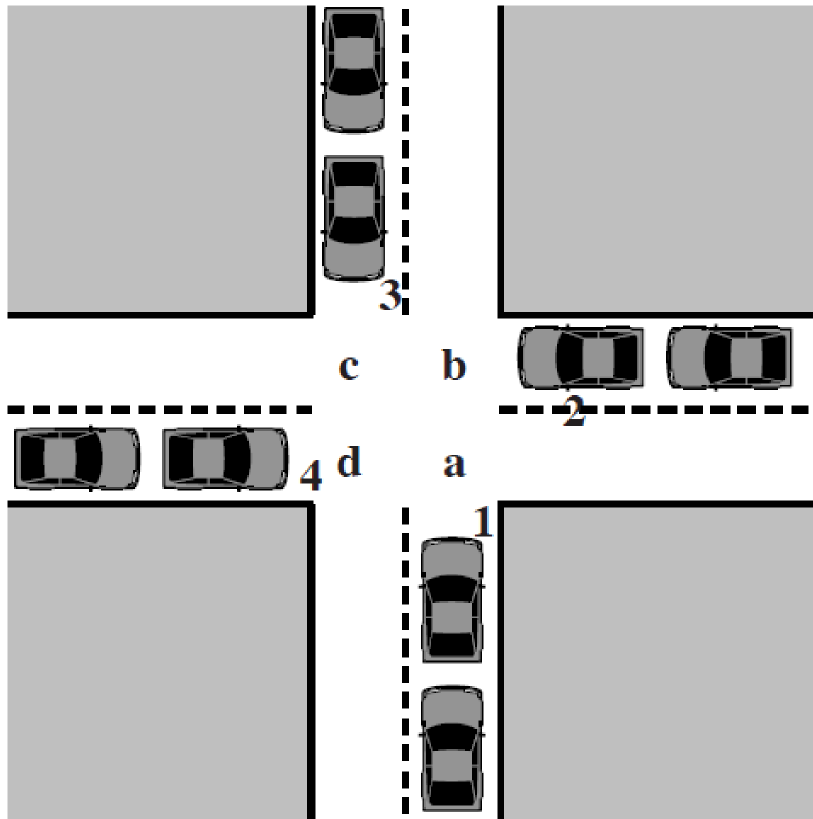
- Gdy blokada jest wolna lub zajęta do odczytu następuje jej zajęcie do

odczytu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego i zwraca kod błędu.

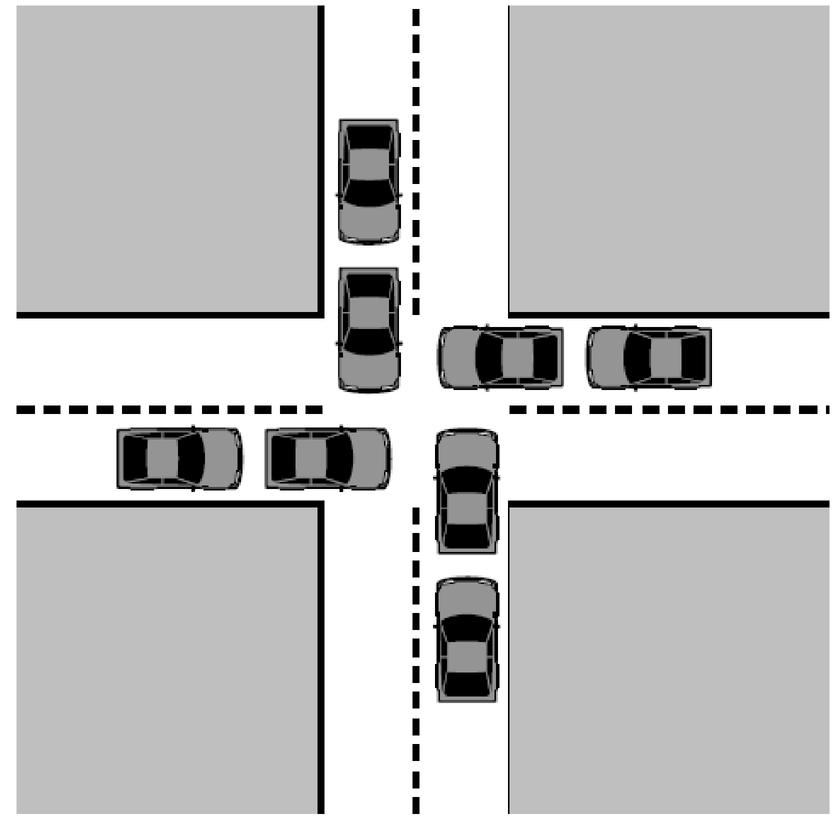
Skasowanie blokady

```
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)
```

Impas/Zakleszczenie



Możliwy impas/zakleszczenie

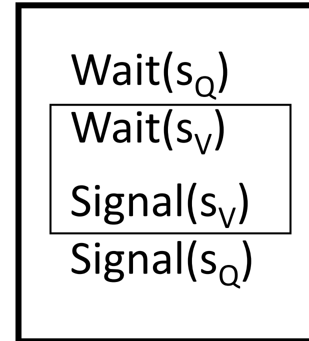


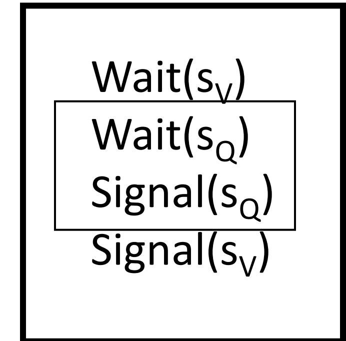
Impas/zakleszczenie

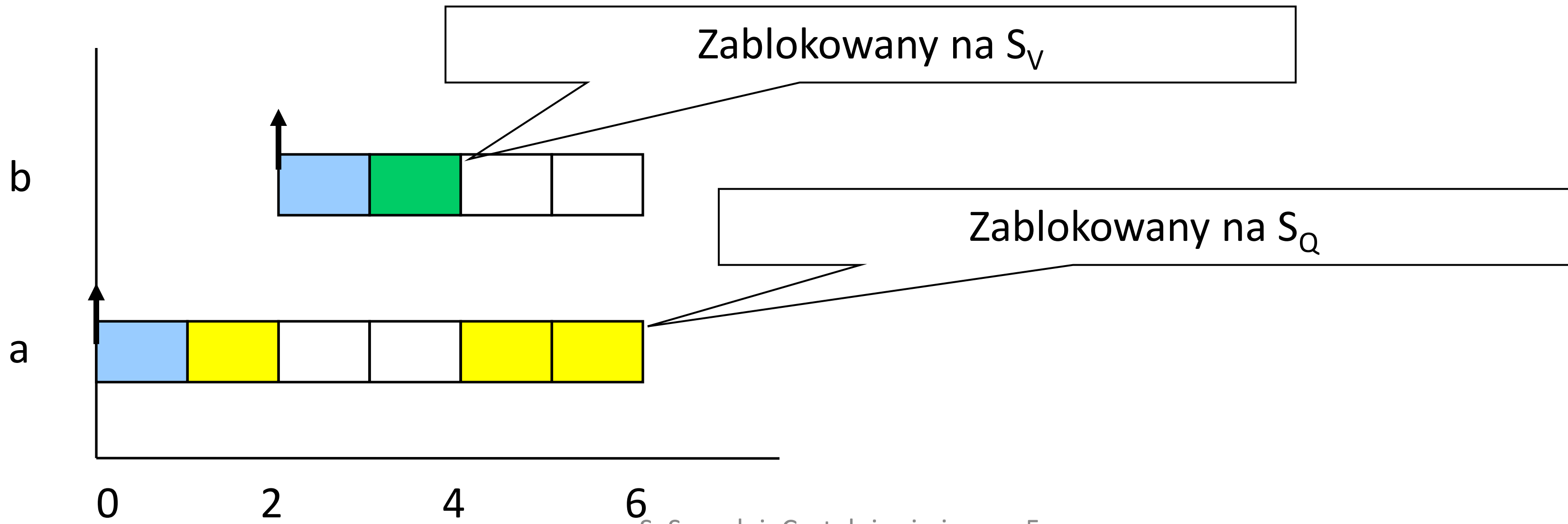
Czym jest impas

- Impas można zdefiniować jako trwałe zablokowanie zestawu procesów, które rywalizują o zasoby lub komunikują się ze sobą nawzajem
- Do impasu dochodzi, każdy każdy proces zestawu jest zablokowany i oczekuje na zdarzenie (zazwyczaj na zwolnienie żądanego zasobu), które może zaistnieć tylko, jeśli zostanie zainicjowane przez inny proces z zestawu procesów.
- Impas jest stanem trwałym, ponieważ żadne ze zdarzeń nigdy nie zachodzi.
- W przeciwieństwie do innych problemów współbieżności, nie istnieje skuteczne rozwiązanie takiej sytuacji.

Przykład impasu/zakleszczenie

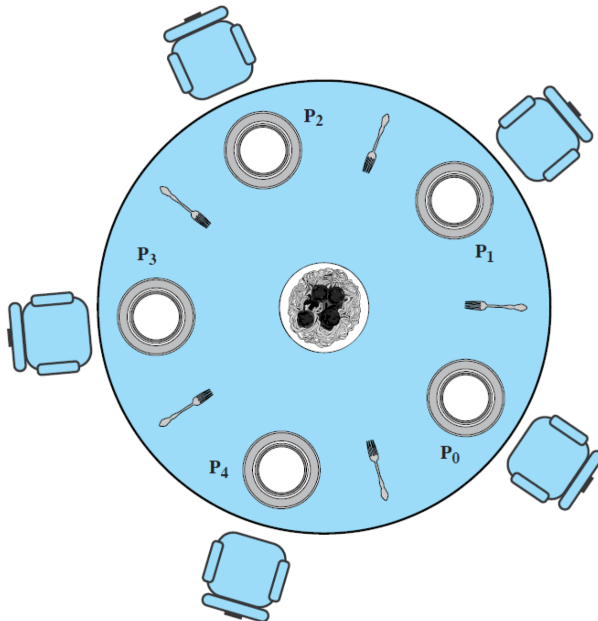






Pięciu filozofów (1)

- Ten problem nie ma praktycznych analogii, jak w przypadku poprzednich klasycznych problemów, ale bardzo dobrze ilustruje problemy występujące przy tworzeniu programów współbieżnych.
- Pięciu filozofów siedzi przy okrągłym stole. Przed każdym stoi talerz. Między talerzami leżą widelce. Pośrodku stołu znajduje się półmisek z rybą. Każdy filozof myśli. Gdy zgłodnieje sięga po widelce znajdujące się po jego prawej i lewej stronie, po czym rozpoczyna posiłek. Gdy się już naje, odkłada widelce i ponownie oddaje się myśleniu.




```
process Filozof (i: 0..4);
begin
  repeat
    myśli;
    protokół_wstępny;
    je;
    protokół_końcowy;
  until false
end;
```

Pięciu filozofów (2)

- Należy tak napisać protokoły wstępne i końcowe, aby:
  1. Jednocześnie tym samym widelcem jadł co najwyżej jeden filozof.
  2. Każdy filozof jadł zawsze dwoma (i zawsze tymi, które leżą przy jego talerzu) widelcami.
  3. Żaden filozof nie umarł z głodu.
- Chcemy ponadto, aby każdy filozof działał w ten sam sposób.

Rozwiązanie I – nieprawidłowe – możliwość zakleszczenia

```
#define NO_OF_PHIL 5
#define NO_OF_FORKS 5
#define MAX_DEL 4

void *philosopher(void *arg);
pthread_t philosopher_threads_table[NO_OF_PHIL];
sem_t _fork[NO_OF_FORKS];

void think(int i)
{
   int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d THINKS...\n",i);
   sleep(think_time);
}

void eat(int i)
{   int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d EATS...\n",i);
   sleep(think_time);
}

void hungry(int i)
{   printf("PHILOSOPHER %d HUNGRY...\n",i);
}
```

```
void *philosopher(void *arg)
{
   int phil_no;
   phil_no=( int)((int *) arg);

   printf("Philosopher %d started\n",phil_no);
   sleep(0);
   while(1)
   {
      think(phil_no);
      hungry(phil_no);
      sem_wait(&_fork[phil_no]);
         printf("Philosopher %d has one\n",phil_no);
      sem_wait(&_fork[(phil_no+1)%NO_OF_FORKS]);
         printf("Philosopher %d has two\n",phil_no);
      eat(phil_no);
      sem_post(&_fork[(phil_no+1)%NO_OF_FORKS]);
      sem_post(&_fork[phil_no]);
   }
}


// Dalej: powołanie filozofów + inicjalizacja
// semaforów + inicjalizacja losowych odcinków czasu
```

Uwagi

- Każdy z filozofów sięga po widelec z lewej strony talerza, a następnie po widelec z prawej strony.
- Kidy dany filozof skończy posiłek, odkłada oba widelce na stół.
- Dodatkowe funkcje think, eat i hungry służą do raportowania stanu danego filozofa oraz do symulowania czasu jedzenia i myślenia
- Takie rozwiązanie prowadzi do **impasu**: Stanie się to wtedy, gdy wszyscy filozofowie poczują jednocześnie głód i usiądą wspólnie przy stole, chwycą widelec z lewej strony, po czym sięgną po widelec z prawej strony.

Rozwiązanie II – bez możliwości zakleszczenia, ale kod filozofów się różni

```
#define NO_OF_PHIL 5
#define NO_OF_FORKS 5
#define MAX_DEL 4

void *philosopher(void *arg);
pthread_t philosopher_threads_table[NO_OF_PHIL];
sem_t _fork[NO_OF_FORKS];

void think(int i)
{   int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d THINKS...\n",i);
   sleep(think_time);}

void eat(int i)
{   int think_time;
   think_time=rand()%MAX_DEL+1;
   printf("PHILOSOPHER %d EATS...\n",i);
   sleep(think_time);}

void hungry(int i)
{   printf("PHILOSOPHER %d HUNGRY...\n",i);;}
```

```
void *philosopher(void *arg){
   int phil_no;
   phil_no=( int)((int *) arg);
   printf("Philosopher %d started\n",phil_no);
   sleep(0);
   while(1)
   {if(phil_no%2) // parzyści      {
         think(phil_no);
         hungry(phil_no);
         sem_wait(&_fork[phil_no]);
            printf("Philosopher %d has one\n",phil_no);
         sem_wait(&_fork[(phil_no+1)%NO_OF_FORKS]);
            printf("Philosopher %d has two\n",phil_no);
         eat(phil_no);
         sem_post(&_fork[(phil_no+1)%NO_OF_FORKS]);
         sem_post(&_fork[phil_no]);      }
      else //nieparzyści      {
         think(phil_no);
         hungry(phil_no);
         sem_wait(&_fork[(phil_no+1)%NO_OF_FORKS]);
            printf("Philosopher %d has one\n",phil_no);
         sem_wait(&_fork[phil_no]);
            printf("Philosopher %d has two\n",phil_no);
         eat(phil_no);
         sem_post(&_fork[phil_no]);
         sem_post(&_fork[(phil_no+1)%NO_OF_FORKS]);}}}...
```

Uwagi

- Filozofów podzielona na „parzystych” i „nieparzystych”
- Parzyści filozofowie zachowują się jak w poprzednim przykładzie, zaś nieparzyści sięgają najpierw po prawy widelec
- W ten sposób uniknięto impasu, ale zachowanie wszystkich filozofów nie jest takie same.
- Innym rozwiązaniem jest wprowadzenie „kelnera”, który nie pozwoli, aby równocześnie przy stole siedziało 5 filozofów lub zastosowanie zmiennych warunkowych decydujących o dostępie do widelców.

Problem wymiany informacji „z potwierdzeniem”

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

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE]; int time_to_exit = 0;

int main() {
   int res;   pthread_t a_thread;   void *thread_result;
   res = pthread_mutex_init(&work_mutex, NULL);
   if (res != 0) {     perror("Mutex initialization failed");
      exit(EXIT_FAILURE);}
   res = pthread_create(&a_thread, NULL,
                                 thread_function, NULL);
   if (res != 0) { perror("Thread creation failed");
                                 exit(EXIT_FAILURE);}
   pthread_mutex_lock(&work_mutex);
   printf("Input some text. Enter 'end' to finish\n");
   while(!time_to_exit) {
      fgets(work_area, WORK_SIZE, stdin);
      pthread_mutex_unlock(&work_mutex);
```

```
while(1) {
         pthread_mutex_lock(&work_mutex);
         if (work_area[0] != '\0') {
            pthread_mutex_unlock(&work_mutex);
            sleep(1);   }  else {   break;     }     }     }
   pthread_mutex_unlock(&work_mutex);
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {     perror("Thread join failed");
                              exit(EXIT_FAILURE); }
   printf("Thread joined\n");
   pthread_mutex_destroy(&work_mutex);
   exit(EXIT_SUCCESS);}

void *thread_function(void *arg) { sleep(1);
   pthread_mutex_lock(&work_mutex);
   while(strncmp("end", work_area, 3) != 0) {
    printf("You input %d characters\n", strlen(work_area) -
1);   work_area[0] = '\0';
     pthread_mutex_unlock(&work_mutex); sleep(1);
     pthread_mutex_lock(&work_mutex);
     while (work_area[0] == '\0' ) {
        pthread_mutex_unlock(&work_mutex); sleep(1);
        pthread_mutex_lock(&work_mutex); } }
   time_to_exit = 1;    work_area[0] = '\0';
   pthread_mutex_unlock(&work_mutex);
   pthread_exit(0);}
```

Wyjście programu

```
Input some text. Enter 'end' to finish
Whit
You input 4 characters
The Crow Road
You input 13 characters
end
Waiting for thread to finish...
Thread joined
```

Jak to działa? (1)

- W obszarze zmiennych globalnych zadeklarowano nowe zmienne work_mutex i time_to_exit
- W głównej funkcji programu zainicjalizowano mutex
- Powołano nowy wątek który:
  - próbuje zamknąć mutex (jeśli jest zamknięty, to oczekuje na jego otwarcie),
  - sprawdza, czy spełniony jest warunek zakończenia wątku, jeśli tak, ustawia zmienną time_to_exit na 1 i pierwszy element tablicy work_area na 0, otwiera mutex
  - jeśli nie, to obliczana jest długość tekstu i ustawiana pierwszy element tablicy work_area na 0, następuje też otwarcie mutex'a (ustawienie pierwszego elementu tablicy na 0 oznacza, że wątek zakończył swoje przetwarzanie współdzielonej zmiennej)
  - po odczekaniu 1 sekundy wątek próbuje zamknąć mutex
  - jeśli zamknięcie się powiedzie, to cyklicznie sprawdza, czy pierwszy element tablicy jest w dalszym ciągu 0, zwalnia mutex, odczekuje 1 sekundę, próbuje zamknąć mutex
  - jeśli zawartość tablicy ulegnie zmianie, to pętla sprawdzająca jest przerywana, mutex pozostaje zamknięty

Jak to działa? (2)

- W wątku macierzystym :
  - następuje próba zamknięcia mutex'a
  - kiedy zamknięcie się powiedzie w pętli następuje odczytywanie tekstów wprowadzanych przez użytkownika (tekst „end" kończy działanie programu) i odblokowanie mutexa
  - w wewnętrznej pętli następuje sprawdzenie, czy tekst nie został przetworzony, sprawdzenie następuje z zachowaniem wzajemnego wykluczania w dostępie do współdzielonej zmiennej (próba zamknięcia a potem otwarcie mutex'a)
  - po wykryciu odebrania danych przez wątek przetwarzający następuje ponownie próba przjęcia dostępu do danych współdzielonych i wpisanie do nich nowego tekstu
  - po wpisaniu do tablicy współdzielonej tekstu „end" następuje zamknięcie wątku macierzystego i potomnego.
- **Uwagi:**
  - **Oba wątki stosują swego rodzaju pooling do wykrywania zmiany stanu zmiennej dzielonej.**

A gdyby zastosować semafor do „potwierdzania”

```
var consyn : semaphore (* init 0 *)
var mutex  : semaphore (* init 1 *)
```

```
process P1;
   sem_wait (mutex);
      produce (X);
      sem_post (consyn)
   sem_post (mutex);
end  P1; `
```

```
process P2;
   sem_wait (mutex);
      sem_wait (consyn);
      consume(X);
   sem_post (mutex);
end P2;
```

Synchronizacja nie może zajść, bo uniemożliwia to „mutex”.
Następuje impas (zakleszczenie).

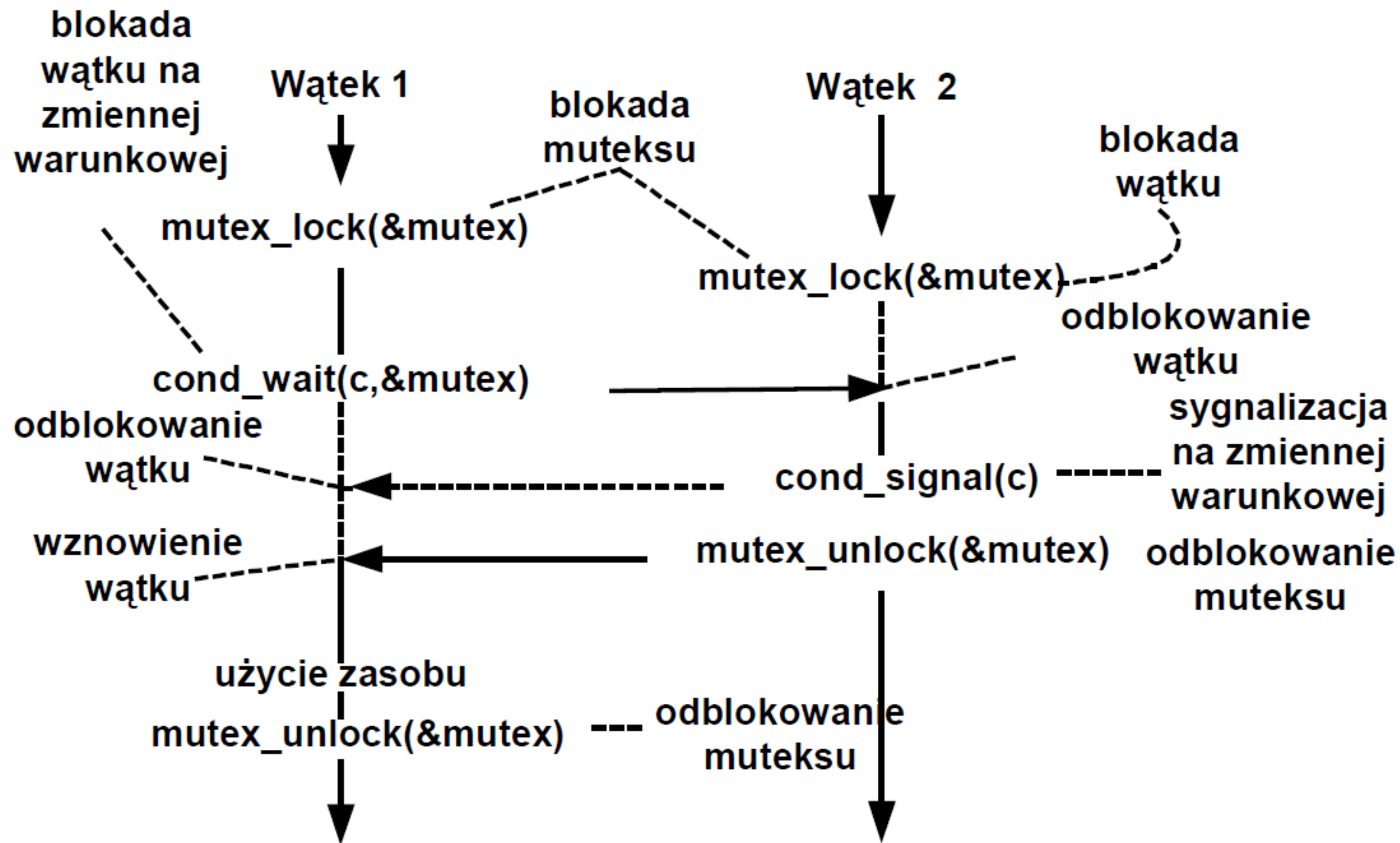
Eliminacja zakleszczenia, ale brak zapewnienia potwierdzenia… ZMIENNA WARUNKOWA

```
var c_v      : condition_var;
var mutex    : semaphore;   (* init 1 *)
var data_send : integer;  (* init 0 *)
```

```
process P1;
   sem_wait (mutex);
      produce (X);
      data_send = 1;
      condition_signal (c_v)
   sem_post (mutex);
end  P1; `
```

```
process P2;
   sem_wait (mutex);
      while(data_send==0)
       condition_wait(c_v);
      consume(X);
      data_send=0;
   sem_post (mutex);
end P2;
```

Graficzna reprezentacja zasady działania zmiennej warunkowej



Zmienna warunkowa

- Zmienne warunkowe dostarczają nowego mechanizmu synchronizacji pomiędzy wątkami. Podczas gdy muteksy implementują synchronizację na poziomie dostępu do współdzielonych danych, zmienne warunkowe pozwalają na synchronizację na podstawie stanu pewnej zmiennej.
- Bez zmiennych warunkowych wątki musiałyby cyklicznie monitorować stan zmiennej (w sekcji krytycznej), aby sprawdzić, czy osiągnęła ona ustaloną wartość. Podejście takie jest z założenia „zasobożerne". Zmienna warunkowa pozwala na osiągnięcie podobnego efektu bez „odpytywania".
- Zmienną warunkową stosuje się zawsze wewnątrz sekcji krytycznej w powiązaniu z zamknięciem muteksu (mutex lock).

Funkcje obsługujące zmienną warunkową

pthread_cond_init	Inicjacja zmiennej warunkowej.
pthread_cond_wait	Czekanie na zmiennej warunkowej
pthread_cond_timedwait	Ograniczone czasowo czekanie na zmiennej warunkowej
pthread_cond_signal	Wznowienie wątku zawieszonego w kolejce danej zmiennej warunkowej
pthread_cond_brodcast	Wznowienie wszystkich wątków zawieszonych w kolejce danej zmiennej warunkowej.
pthread_cond_destroy	Skasowanie zmiennej warunkowej i zwolnienie jej zasobów

Parametry funkcji (1)

Inicjacja zmiennej warunkowej

```
int pthread_cond_init(pthreads_cond_t *zw,
pthreads_condattr_t attr)
```

`zw` Zadeklarowana wcześniej zmienna typu **`pthread_cond_t`**.
`attr` Atrybuty zmiennej warunkowej. Gdy **`attr`** jest równe NULL przyjęte będą wartości domyślne.

Zawieszenie wątku w oczekiwaniu na sygnalizację

```
int pthread_cond_wait(pthreads_cond_t *zw,
pthread_mutex_t *mutex)
```

`zw` Zadeklarowana i zainicjowana zmienna typu **`pthread_cond_t`**.
`mutex` Zadeklarowana i zainicjowana zmienna typu **`pthread_mutex_t`**.

Parametry funkcji (2)

Wznowienie zawieszonego wątku

`int pthread_cond_signal(pthreads_cond_t *zw)`
`zw` Zadeklarowana i zainicjowana zmienna typu
`pthread_cond_t.`
Jeden z wątków zablokowanych na zmiennej warunkowej zw zostanie zwolniony.

Wznowienie wszystkich zawieszonych wątków

`int pthread_cond_brodcast(pthreads_cond_t *zw)`
`zw` Zadeklarowana i zainicjowana zmienna typu
`pthread_cond_t.`
Wszystkie wątki zablokowane na zmiennej warunkowej zw zostaną zwolnione.

Schemat stosowania zmiennej warunkowej

```
// Wątek oczekujący na warunek

pthread_mutex_lock(&m)
...
while( ! warunek )
pthread_cond_wait( &cond, &m)
...
pthread_mutex_unlock(&m)
```

```
//Wątek ustawiający warunek i sygnalizujący jego spełnienie

pthread_mutex_lock(&m)
...
ustawienie_warunku
pthread_cond_signal( &cond)
...
pthread_mutex_unlock(&m)
```

Rozwiązanie synchronizacji bez zakleszczeń (1)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define TCOUNT 5
#define NUM_THREADS 2
pthread_mutex_t cond_mutex;
pthread_cond_t  cond_var1;
int shared_var=0;
int data_send=0;
void *producer(void *t) {
 int i;
 for (i=0; i < TCOUNT; i++) {
  pthread_mutex_lock(&cond_mutex);
    shared_var++;      data_send=1;
    pthread_cond_signal(&cond_var1);
    printf("producer i: %d, shared_var: %d,
            data_send: %d\n",i,shared_var,data_send);
  pthread_mutex_unlock(&cond_mutex);
  //sleep(4);
  }
 pthread_exit(NULL);
}
```

```
void *consumer(void *t) {
 int i;
 for (i=0; i < TCOUNT; i++) {
  pthread_mutex_lock(&cond_mutex);
  while (data_send == 0) {
   printf("consumer_waits\n");
   pthread_cond_wait(&cond_var1, &cond_mutex);
  }
  data_send=0;
  printf("consumer i: %d, shared_var: %d, data_send:
%d\n",i,shared_var,data_send);
  pthread_mutex_unlock(&cond_mutex);
 }
 pthread_exit(NULL);
}
```

Rozwiązanie synchronizacji bez zakleszczeń (2)

```
int main(int argc, char *argv[])
{
  int i;
  pthread_t threads[2];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&cond_mutex, NULL);
  pthread_cond_init (&cond_var1, NULL);
   pthread_cond_init (&cond_var2, NULL);

  /* For portability, explicitly create threads in a joinable
state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate
                 (&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create
                 (&threads[0], &attr, producer, (void *)NULL);
  pthread_create
                 (&threads[1], &attr, consumer, (void *)NULL);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
```

```
  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&cond_mutex);
  pthread_cond_destroy(&cond_var1);
  pthread_exit (NULL);
}
```

Wyjście programu

```
consumer_waits
producer i: 0, shared_var: 1, data_send: 1
producer i: 1, shared_var: 2, data_send: 1
producer i: 2, shared_var: 3, data_send: 1
producer i: 3, shared_var: 4, data_send: 1
producer i: 4, shared_var: 5, data_send: 1
consumer i: 0, shared_var: 5, data_send: 0
consumer_waits
```

Komunikacja z potwierdzeniami
ZMIENNA WARUNKOWA

```
var c_v1, c_v2     : condition_var;
var mutex   : semaphore;   (* init 1 *)
var data_send : integer; (* init 0 *)
```

```
process P1;
  sem_wait (mutex);
    produce (X);
     data_send = 1;
     condition_signal (c_v1)
     while(data_send == 1)
       conditional_wait(c_v2);
  sem_post (mutex);
end  P1;
```

```
process P2;
  sem_wait (mutex);
      while(data_send == 0)
       condition_wait(c_v1);
      consume(X);
     data_send=0;
     conditional_signal(c_v2);
  sem_post (mutex);
end P2;
```

Rozwiązanie komunikacji z potwierdzeniami (1)

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define TCOUNT 5
#define NUM_THREADS 2
pthread_mutex_t cond_mutex;
pthread_cond_t  cond_var1, cond_var2;
int shared_var=0;
int data_send=0;

void *producer(void *t) {
  int i;
  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&cond_mutex);
      shared_var++;
      data_send=1;
      printf("producer i:%d,
              shared_var: %d\n",i,shared_var);
        pthread_cond_signal(&cond_var1);
      while(data_send==1) {
        printf("producer waits\n");
        pthread_cond_wait(&cond_var2, &cond_mutex);
      }
    pthread_mutex_unlock(&cond_mutex);
    }
  pthread_exit(NULL);
}
```

```
void *consumer(void *t)
{
  int i;
  for (i=0; i < TCOUNT; i++)
  {
  pthread_mutex_lock(&cond_mutex);
  while (data_send == 0) {
    printf("consumer_waits\n");
      pthread_cond_wait(&cond_var1, &cond_mutex);
    }
  printf("consumer i: %d, shared_var: %d\n",i,shared_var);
  data_send=0;
  pthread_cond_signal(&cond_var2);
  pthread_mutex_unlock(&cond_mutex);
  }
  pthread_exit(NULL);
}
```

Rozwiązanie komunikacji z potwierdzeniami (2)

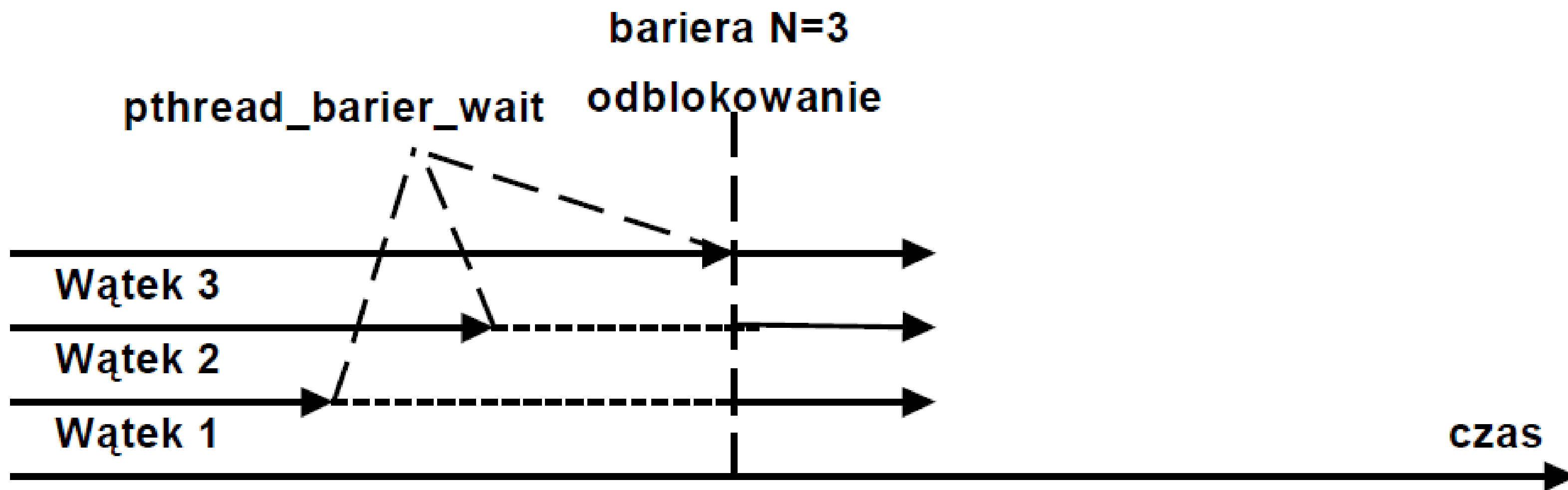
```
int main(int argc, char *argv[])
{
  int i;
  pthread_t threads[2];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&cond_mutex, NULL);
  pthread_cond_init (&cond_var1, NULL);
   pthread_cond_init (&cond_var2, NULL);

  /* For portability, explicitly create threads in a joinable
state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, producer, (void
*)NULL);
  pthread_create(&threads[1], &attr, consumer, (void
*)NULL);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&cond_mutex);
  pthread_cond_destroy(&cond_var1);
  pthread_cond_destroy(&cond_var2);
  pthread_exit (NULL);}
```

Wyjście programu

```
consumer_waits
producer i:0, shared_var: 1
producer waits
consumer i: 0, shared_var: 1
consumer_waits
producer i:1, shared_var: 2
producer waits
consumer i: 1, shared_var: 2
consumer_waits
producer i:2, shared_var: 3
producer waits
consumer i: 2, shared_var: 3
consumer_waits
producer i:3, shared_var: 4
producer waits
consumer i: 3, shared_var: 4
consumer_waits
producer i:4, shared_var: 5
producer waits
consumer i: 4, shared_var: 5
ubuntu@ubuntu:~/Projects/cond_var_p_c1/bin/Debug$
```

Bariery

Bariera jest narzędziem do synchronizacji procesów działających w ramach grup. Wywołanie funkcji pthread barrier wait(...) powoduje zablokowanie zadania bieżącego do chwili gdy zadana liczba wątków zadań nie wywoła tej procedury.



Inicjalizacja bariery

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

barrier Zadeklarowana zmienna typu **pthread_barierr_t.**
attr Atrybuty. Gdy NULL użyte są atrybuty domyślne
count Licznik bariery
Funkcja powoduje zainicjowanie bariery z wartością licznika count.

Czekanie na barierze

```
int pthread_barrier_wait( pthread barrier t * barrier )
```

barrier Zadeklarowana i zainicjowana zmienna typu **pthread_barierr_t.**
Funkcja powoduje zablokowanie wątku bieżącego na barierze. Gdy count wątków zablokuje się na barierze to wszystkie zostaną odblokowane.
Funkcja zwraca **BARRIER_SERIAL_THREAD** dla jednego z wątków (wybranego arbitralnie) i 0 dla wszystkich pozostałych wątków. Wartość licznika będzie taka jak przy ostatniej inicjacji bariery.

Kasowanie bariery

int pthread barrier destroy(pthread barrier t * barrier)

barrier Zadeklarowana i zainicjowana zmienna typu **pthread_barierr_t.**

Funkcja powoduje skasowanie bariery

Przykład zastosowania bariery

```
#include <sys/types.h>
#include <pthread.h>
#include <malloc.h>
pthread_barrier_t * my_barrier;
void * thread1(void * arg){
printf("Watek 1 przed bariera\n");
pthread_barrier_wait(my_barrier);
printf("Watek 1 po barierze \n");}

void * thread2(void * arg){
printf("Watek 2 przed bariera\n");
pthread_barrier_wait(my_barrier);
printf("Watek 2 po barierze \n");}

int main(){
pthread_t w1,w2;
my_barrier = (pthread_barrier_t*)malloc(sizeof(pthread_barrier_t));
pthread_barrier_init(my_barrier, NULL, 2);
pthread_create(&w1, 0, thread1, 0);
pthread_create(&w2, 0, thread2, 0);
pthread_join(w1, 0);
pthread_join(w2, 0);
return 0;}
```

Wirujące blokady

Wirujące blokady są środkiem zabezpieczania sekcji krytycznej.
Wykorzystują jednak czekanie aktywne zamiast przełączenia kontekstu wątku tak jak się to dzieje w muteksach.

Inicjacja wirującej blokady

int pthread_spin_init(pthread spinlock t *blokada, int pshared)

blokada - Identyfikator wirującej blokady **pthread_spinlock_t**

pshared -

- **PTHREAD_PROCESS_SHARED –** na blokadzie mogą operować wątki należące do różnych procesów
- **PTHREAD_PROCESS_PRIVATE –** na blokadzie mogą operować tylko wątki należące do tego samego procesu

Funkcja inicjuje zasoby potrzebne wirującej blokadzie. Każdy proces, który może sięgnąć do zmiennej identyfikującej blokadę może jej używać. Blokada może być w dwóch stanach:

- Wolna
- Zajęta

Zajęcie blokady

`int pthread_spin_lock( pthread spinlock t * blokada)`

`blokada` Identyfikator wirującej blokady – zmienna typu **`pthread_spinlock_t`**

Działanie funkcji zależy od stanu blokady. Gdy blokada jest wolna następuje jej zajęcie. Gdy blokada jest zajęta wątek wykonujący funkcję **`pthread_spin_lock(...)`** ulega zablokowaniu do czasu gdy inny wątek nie zwolni blokady wykonując funkcję **`pthread_spin_unlock(...)`**.

Próba zajęcia blokady

`int pthread_spin_trylock( pthread spinlock t * blokada)`

`blokada` Identyfikator wirującej blokady – zmienna typu **`pthread_spinlock_t`**

Działanie funkcji zależy od stanu blokady. Gdy blokada jest wolna następuje jej zajęcie – funkcja zwraca wartość **`EOK.`** Gdy blokada jest zajęta następuje natychmiastowy powrót i funkcja zwraca stałą **`EBUSY`**.

Zwolnienie blokady

int pthread_spin_unlock(pthread spinlock t * blokada)
blokada Identyfikator wirującej blokady – zmienna typu **pthread_spinlock_t**
Działanie funkcji zależy od stanu blokady. Gdy są wątki czekające na zajęcie blokady to jeden z nich zajmie blokadę. Gdy żaden wątek nie czeka na zajęcie blokady będzie ona zwolniona.

Skasowanie blokady

int pthread_spin_destroy(pthread spinlock t * blokada)
blokada Identyfikator wirującej blokady – zmienna typu **pthread_spinlock_t**
Funkcja zwalnia blokadę i zajmowane przez nią zasoby.

Przykład programu stosującego wirującą blokadę

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void *thread_function(void *arg);
pthread_spinlock_t *blokada;
int main() {
   int res;   pthread_t a_thread;   void *thread_result;
   blokada = (pthread_spinlock_t *)
                 malloc(sizeof(pthread_spinlock_t));
   res =
pthread_spin_init(blokada,PTHREAD_PROCESS_SHARED);
   if (res != 0) {  perror("Spinlock initialization failed");
                    exit(EXIT_FAILURE); }
   res = pthread_create(&a_thread, NULL,
                                  thread_function, NULL);
   if (res != 0) { perror("Thread creation failed");
                   exit(EXIT_FAILURE);    }
   while(1)
   {   pthread_spin_lock(blokada);
      printf("Spin lock taken by thread 1...\n");
      sleep(5);
      pthread_spin_unlock(blokada);
      printf("Spin lock released by thread 1...\n");
      sleep(3);    }
 pthread_spin_destroy(blokada);
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {       perror("Thread join failed");
                         exit(EXIT_FAILURE);}
   printf("Thread joined\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   int res;
   sleep(1);
   while(1)
   {  res = pthread_spin_trylock(blokada);
      if(res != 0)
      {  printf("Spin lock busy...\n");
         sleep(1); }
      else
      {   printf("Spin lock taken by thread 2...\n");
         sleep(1);
         pthread_spin_unlock(blokada);
         printf("Spin lock released by thread 2...\n");
         sleep(1); }
   }
   pthread_exit(0); }
```

Dyskusja

- W przykładowym programie pracują 2 wątki – 1 macierzysty i 1 potomny.
- W wątku macierzystym zastosowano wirującą blokadę do ochrony sekcji krytycznej
- W wątku potomnym następuje cykliczne sprawdzenie, czy blokada jest zamknięta, czy wolna, jeśli jest wolna to następuje przejście sekcji krytycznej. W przeciwnym wypadku użytkownikowi jest przekazywany komunikat, że blokada jest zajęta.