# Problem producenta-konsumenta

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

# Producenci i konsumenci (1)

- W systemie działa *P>0* procesów, które produkują pewne dane oraz *K>0* procesów, które odbierają dane od producentów. Między producentami a konsumentami może znajdować się bufor o pojemności *B*, którego zadaniem jest równoważenie chwilowych różnic w czasie działania procesów. Procesy produkujące dane będziemy nazywać producentami, a procesy odbierające dane --- konsumentami. Zadanie polega na synchronizacji pracy producentów i konsumentów, tak aby:
  1. Konsument oczekiwał na pobranie danych w sytuacji, gdy bufor jest pusty. Gdybyśmy pozwolili konsumentowi odbierać dane z bufora zawsze, to w sytuacji, gdy nikt jeszcze nic w buforze nie zapisał, odebrana wartość byłaby bezsensowna.
  2. Producent umieszczając dane w buforze nie nadpisywał danych już zapisanych, a jeszcze nie odebranych przez żadnego konsumenta. Wymaga to wstrzymania producenta w sytuacji, gdy w buforze nie ma wolnych miejsc.
  3. Jeśli wielu konsumentów oczekuje, aż w buforze pojawią się jakieś dane oraz ciągle są produkowane nowe dane, to każdy oczekujący konsument w końcu coś z bufora pobierze. Nie zdarzy się tak, że pewien konsument czeka w nieskończoność na pobranie danych, jeśli tylko ciągle napływają one do bufora.
  4. Jeśli wielu producentów oczekuje, aż w buforze będzie wolne miejsce, a konsumenci ciągle coś z bufora pobierają, to każdy oczekujący producent będzie mógł coś włożyć do bufora. Nie zdarzy się tak, że pewien producent czeka w nieskończoność, jeśli tylko ciągle z bufora coś jest pobierane.

# Producenci i konsumenci (2)

- Rozpatruje się różne warianty tego problemu:
  1. Bufor może być nieskończony.
  2. Bufor cykliczny może mieć ograniczoną pojemność.
  3. Może w ogóle nie być bufora.
  4. Może być wielu producentów lub jeden.
  5. Może być wielu konsumentów lub jeden.
  6. Dane mogą być produkowane i konsumowane po kilka jednostek na raz.
  7. Dane muszą być odczytywane w kolejności ich zapisu lub nie.
- Problem producentów i konsumentów jest abstrakcją wielu sytuacji występujących w systemach komputerowych, na przykład zapis danych do bufora klawiatury przez sterownik klawiatury i ich odczyt przez system operacyjny.

# Producent-konsument kolejki (producent)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512
struct my_msg_st {    long int my_msg_type;
                      unsigned int pack_no;
                      char some_text[MAX_TEXT]; };
int main(int argc, char **argv)
{
   int running = 1; int msgid;
   struct my_msg_st some_data;
   static unsigned int counter=0;

   if(argc<2)
   { printf("NO MESSAGE DEFINED!\n"); exit(1); }
   msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
   if (msgid == -1) {
      fprintf(stderr, "msgget failed with error: %d\n",
                  errno);
   exit(EXIT_FAILURE); }
```

```
while(running) {
      sprintf(some_data.some_text,argv[1]);
      some_data.my_msg_type = 1;
      some_data.pack_no = counter++;
      if (msgsnd(msgid, (void *)&some_data,
                                MAX_TEXT, 0) == -1) {
         fprintf(stderr, "msgsnd failed\n");
         exit(EXIT_FAILURE);
      }
      sleep(1);
   }
   exit(EXIT_SUCCESS);
}
```

# Producent-konsument kolejki (konsument)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MAX_TEXT 512
struct my_msg_st {    long int my_msg_type;
                      unsigned int pack_no;
                      char some_text[MAX_TEXT];};
int main()
{
   int running = 1;
   int msgid;
   struct my_msg_st some_data;
   long int msg_to_receive = 0;

   msgid = msgget((key_t)1234, 0666 | IPC_CREAT);

   if (msgid == -1) {
      fprintf(stderr, "msgget failed with error: %d\n",
errno);
      exit(EXIT_FAILURE);
   }
```

```
 while(running) {
      if (msgrcv(msgid, (void *)&some_data, BUFSIZ,
            msg_to_receive, 0) == -1) {
         fprintf(stderr, "msgrcv failed with error: %d\n",
errno);
         exit(EXIT_FAILURE);
      }
      printf("You wrote: %s", some_data.some_text);
      printf(" %d\n",some_data.pack_no);
   }


   if (msgctl(msgid, IPC_RMID, 0) == -1) {
      fprintf(stderr, "msgctl(IPC_RMID) failed\n");
      exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);
}
```

# Dyskusja

- Jako medium komunikacyjne zastosowano kolejkę
- Zalety:
  - Prostota
  - Automatyczna synchronizacja i wzajemne wykluczanie w dostępie do współdzielonego kanału komunikacyjnego
  - System operacyjny „sam” zarządza dostępem do współdzielonego zasobu
- Wady:
  - Ograniczoność pojemności kolejki
  - W skrajnym wypadku nie będzie możliwe przesłanie danych, lub będzie możliwe przesłanie tylko pojedynczej struktury danych
- Warto stosować takie rozwiązanie, gdy pomiędzy producentem, a konsumentem przesyłane są niewielkie porcje danych/komunikaty

# Szkic rozwiązania z zastosowaniem semaforów (1)

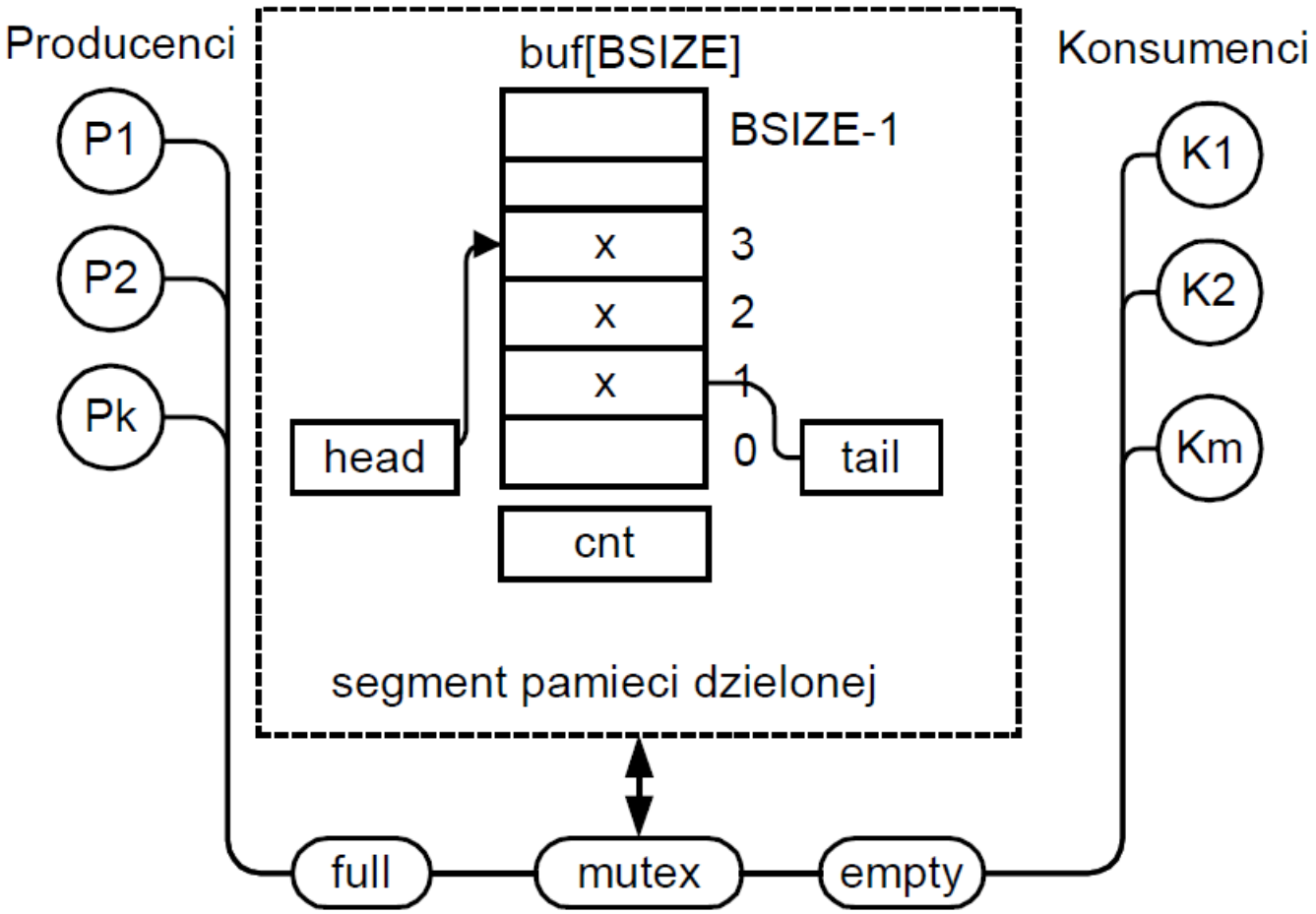
```
#define BufSize 8 // Bufor ma 8 elementów
RecType Buffer[BufSize]; // Buf. na elementy
semaphore Mutex; // Ochrona bufora
semaphore Puste; // Wolne bufory
semaphore Pelne; // Zajete bufory
int count; // Wskaźnik bufora
producent(void) {
RecType x;
do { ...
produkcja rekordu x;
// Czekaj na wolny bufor
sem_wait(Puste);
sem_wait(Mutex);
// Umieść element x w buforze
Buffer[count] = x; count++;
sem_post(Mutex);
// Pojawił się nowy element
sem_post(Pelne);
} while(1);}
```

```
konsument(void) {
RecType x;
do {
...
...
// Czekaj na element
sem_wait(Pelne);
sem_wait(Mutex);
// Pobierz element x z bufora
count--;
x = Buffer[count];
sem_post(Mutex);
// Zwolnij miejsce w buforze
sem_post(Puste);
konsumpcja rekordu x;
...
} while(1);
}
```

# Szkic rozwiązania z zastosowaniem semaforów (1)

```
main(void) {
count = 0;
sem_init(Puste,BufSize);          // Inicjacja semafora Puste
sem_init(Pelne,0);                // Inicjacja semafora Pelne
sem_init(Mutex,1);                // Inicjacja semafora Mutex
pthread_create(…,producent,..);   // Start K wątków producenta
..
pthread_create(…,konsument,..);   // Start L wątków konsumenta
..
}
```

# Rozwiązanie problemu producent-konsument (pamięć dzielona)



# Rozwiązanie – pamięć dzielona (1)

```
##include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <pthread.h>
#include <semaphore.h>
#include <assert.h>
#include <sys/stat.h>
#include <sys/file.h>

#define BSIZE 4 // Rozmiar bufora
#define LSIZE 80 // Dlugosc linii
typedef struct { // Obszar wspólny
char buf[BSIZE][LSIZE];
int head;
int tail;
int cnt;
} bufor_t;
```

```
int main()
{   sem_t *mutex;
   sem_t *empty;
   sem_t *full;
   int shmid;
   void *shared_memory = (void *)0;
   bufor_t *shared_stuff;
   int i=0;
   shmid = shmget((key_t)1234,
                  sizeof(bufor_t), 0666 | IPC_CREAT);
   if (shmid == -1) {
      fprintf(stderr, "shmget failed\n");
      exit(EXIT_FAILURE);
   }
   shared_memory = shmat(shmid, (void *)0, 0);
   if (shared_memory == (void *)-1) {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);
   }
   printf("Memory attached at %X\n",
                                  (int)shared_memory);
   shared_stuff = (bufor_t *)shared_memory;
   shared_stuff->head=0;
   shared_stuff->tail=0;
   shared_stuff->cnt=0;
```

# Rozwiązanie – pamięć dzielona (2)

```
mutex = sem_open("mutex",O_CREAT,S_IRWXU,1);
empty = sem_open("empty",O_CREAT,S_IRWXU,BSIZE);
full =  sem_open("full",O_CREAT,S_IRWXU,0);
//Producenci:
printf("Tworze producentow...\n");
{
  if(fork()==0)
 { printf("Producent %d uruchominy\n",i);
    sleep(1);
    while(1)
    {static int j=0;
     sem_wait(empty);
     sem_wait(mutex);
     printf("Prod: %d - cnt: %d head: %d tail: %d\n",
     i, shared_stuff-> cnt,shared_stuff->head,
     shared_stuff->tail);
     printf(shared_stuff->buf[shared_stuff->head],
     "Komunikat %d producent %d",j++,i);
     shared_stuff-> cnt ++;
     shared_stuff->head = (shared_stuff->head +1) %
                                                  BSIZE;
     sem_post(mutex);
     sem_post(full);
      sleep(1);
     }
     exit(0); } }
```

```
printf("Tworze konsumentów...\n");
   {   if(fork()==0)
       {  printf("Konsument %d uruchominy\n",i);
          sleep(2);
          while(1)
          { sem_wait(full);
            sem_wait(mutex);
            printf("Konsument %d - cnt: %d odebrano %s\n",
             i, shared_stuff->cnt,
             shared_stuff->buf[shared_stuff->tail]);
             shared_stuff-> cnt --;
             shared_stuff->tail = (shared_stuff->tail +1) %
                                                  BSIZE;
             sem_post(mutex);
             sem_post(empty);
             sleep(1);
          } exit(0);
       }
   }
   {   int stat_val;
       pid_t child_pid;
       for(i=0;i<2;i++)
       child_pid = wait(&stat_val);
   }
   return 0;
}
```

# Dyskusja

- Medium komunikacyjnym jest kolejka zrealizowana we współdzielonym obszarze pamięci
- Zalety:
  - Mniejsze ograniczenia na rozmiar przesyłanych danych
- Wady:
  - Konieczność „samodzielnego” zarządzania prawidłowym dostępem do danych
  - Bardziej złożony kod
- Takie rozwiązanie zmniejsza ograniczenie ze względu na pojemność bufora do wymiany danych. Warto stosować takie rozwiązanie, gdy pomiędzy producentem, a konsumentem przesyłane są większe porcje danych/komunikaty

# 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

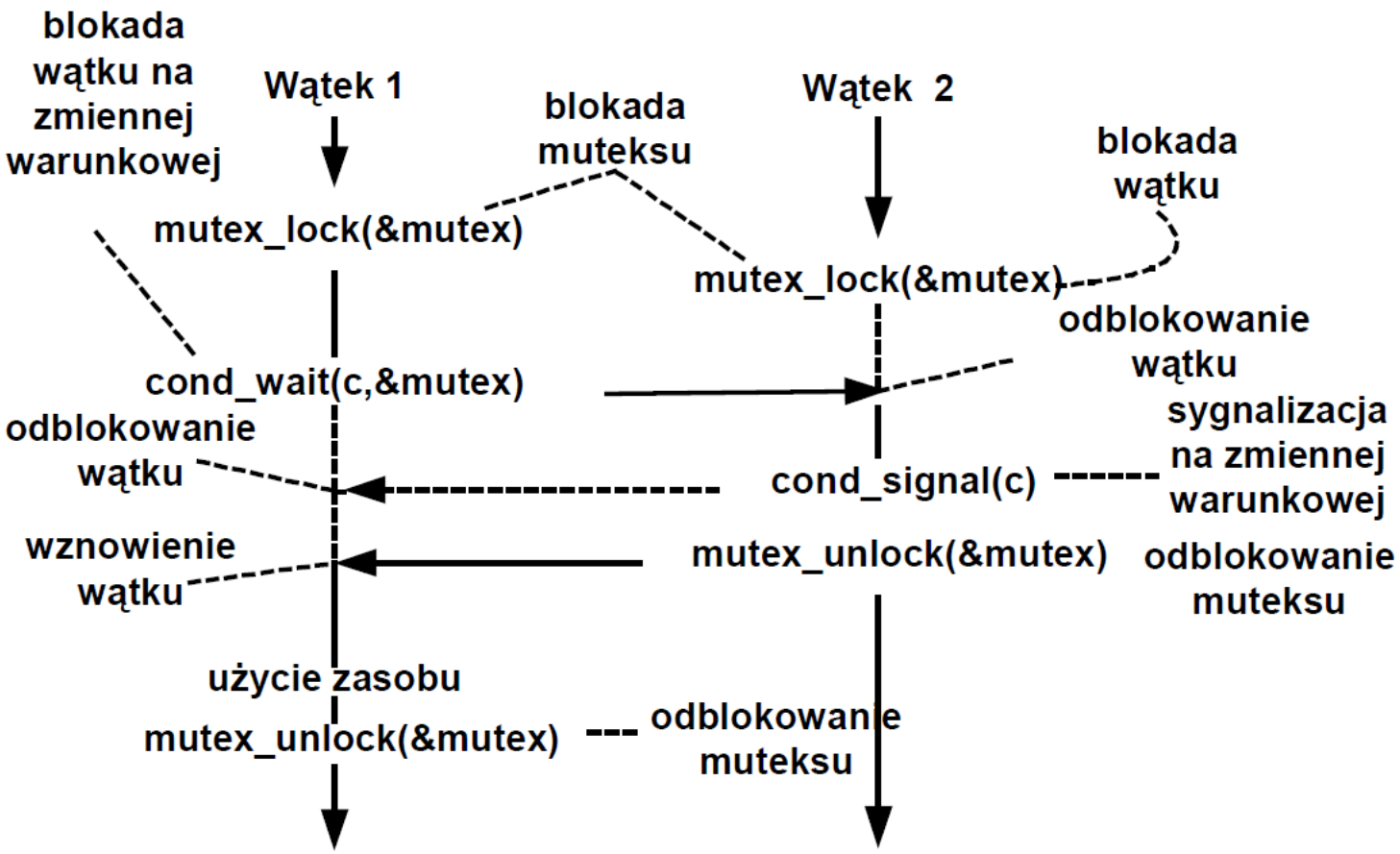
```
// Wątek oczekujący na warunek

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)
```

# Graficzna reprezentacja zasady działania zmiennej warunkowej



# Przykładowa sekwencja posługiwania się zmienną warunkową

**Główny wątek:**

- Deklaruje i inicjalizuje globalne zmienne, które wymagają synchronizacji
- Deklaruje  i inicjalizuje zmienną warunkową (np. „count”)
- Deklaruje i inicjalizuje powiązany z nią muteks
- Tworzy i uruchamia wątki A i B

**Wątek A:**

- Pracuje do punktu gdzie okreslony warunek musi być spełniony (np. count musi osiągnąć pewną wartość)
- Zamyka muteks i sprawdza wartość globalnej zmiennej
- Wywołuje funkcję pthread_cond_wait() aby zablokować swoje wykonanie do momentu otrzymania sygnału od wątku B. Uwaga: wywołanie pthread_cond_wait() automatycznie i atomowo zwalnia muteks powiązany ze zmienną warunkową, aby mogła być używana przez wątek B.
- Kiedy wątek otrzyma sygnał muteks jest automatycznie zamykany
- Następuje otwarcie muteksu
- Obliczenia są kontynuowane

**Thread B:**

- Pracuje
- Zamyka powiązany muteks
- Zmienia wartość zmiennej warunkowej, na która oczekuje wątek A
- Sprawdza wartość globalnej zmiennej warunkowej, na którą czeka wątek A. Jeśli stan zmiennej odpowiada warunkowi, sygnalizuje to wątkowi A.
- Zwalania muteks.
- Kontynuuje obliczenia.

# Zastosowanie zmiennej warunkowej

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUM_THREADS  3
#define TCOUNT 10
#define COUNT_LIMIT 12
int     count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;
void *inc_count(void *t)
{  int i;
  long my_id = (long)t;
  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;
if (count == COUNT_LIMIT) {
     printf("inc_count(): thread %ld, count = %d
                Threshold reached. ", my_id, count);
     pthread_cond_signal(&count_threshold_cv);
     printf("Just sent signal.\n");
     }
    printf("inc_count(): thread %ld, count = %d, unlocking
    mutex\n",   my_id, count);
    pthread_mutex_unlock(&count_mutex);
    sleep(1); } pthread_exit(NULL); }
```

```
void *watch_count(void *t)
{  long my_id = (long)t;
   printf("Starting watch_count(): thread %ld\n", my_id);
   pthread_mutex_lock(&count_mutex);
   while (count < COUNT_LIMIT) {
    printf("watch_count(): thread %ld Count= %d.
    Going into wait...\n", my_id,count);
    pthread_cond_wait(&count_threshold_cv,
                                &count_mutex);
    printf("watch_count(): thread %ld Condition signal
                received. Count= %d\n", my_id,count);
    printf("watch_count(): thread %ld Updating the value of
                count... %d\n", my_id,count);
    count += 125;
    printf("watch_count(): thread %ld count now = %d.\n",
                 my_id, count);
    }
  printf("watch_count(): thread %ld
                                Unlocking mutex.\n", my_id);
  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}
```

# Zastosowanie zmiennej warunkowej

```
int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

pthread_mutex_init(&count_mutex, NULL);
pthread_cond_init (&count_threshold_cv, NULL);

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,
PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0],
                 &attr, watch_count,  (void *)t1);
  pthread_create(&threads[1],
                  &attr, inc_count, (void *)t2);
  pthread_create(&threads[2],
                 &attr, inc_count, (void *)t3);

  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);  }
  printf ("Main(): Waited and joined with %d threads. Final
                 value of count = %d. Done.\n",
                 NUM_THREADS, count);
```

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

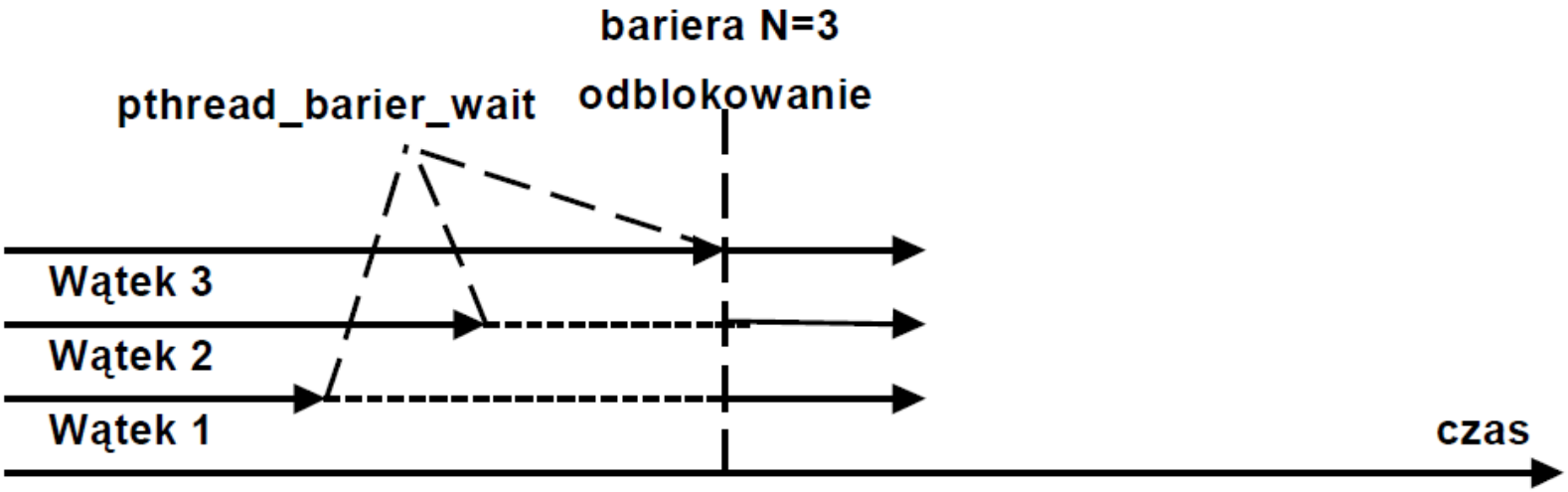
```
Starting watch_count(): thread 1
inc_count(): thread 2, count = 1, unlocking mutex
inc_count(): thread 3, count = 2, unlocking mutex
watch_count(): thread 1 going into wait...
inc_count(): thread 3, count = 3, unlocking mutex
inc_count(): thread 2, count = 4, unlocking mutex
inc_count(): thread 3, count = 5, unlocking mutex
inc_count(): thread 2, count = 6, unlocking mutex
inc_count(): thread 3, count = 7, unlocking mutex
inc_count(): thread 2, count = 8, unlocking mutex
inc_count(): thread 3, count = 9, unlocking mutex
inc_count(): thread 2, count = 10, unlocking mutex
inc_count(): thread 3, count = 11, unlocking mutex
inc_count(): thread 2, count = 12  Threshold reached. Just sent signal.
inc_count(): thread 2, count = 12, unlocking mutex
watch_count(): thread 1 Condition signal received.
watch_count(): thread 1 count now = 137.
inc_count(): thread 3, count = 138, unlocking mutex
inc_count(): thread 2, count = 139, unlocking mutex
inc_count(): thread 3, count = 140, unlocking mutex
inc_count(): thread 2, count = 141, unlocking mutex
inc_count(): thread 3, count = 142, unlocking mutex
inc_count(): thread 2, count = 143, unlocking mutex
inc_count(): thread 3, count = 144, unlocking mutex
inc_count(): thread 2, count = 145, unlocking mutex
Main(): Waited on 3 threads. Final value of count = 145. Done.
```

# Dyskusja

- W przykładowym programie pracują 4 wątki – 1 macierzysty i 3 potomne.
- W dwu z potomnych wątków następuje inkrementacja zmiennej count.
- Od stanu tej zmiennej uzależnione jest odblokowanie trzeciego z wątków potomnych, który stosuje do monitorowania stanu tej zmiennej zmienną warunkową.
- Ten wątek, który „wykryje”, że stan zmiennej spełnia warunek odblokowania wątku zablokowanego, sygnalizuje wątkowi zablokowanemu, że może podjąć dalsze obliczenia.
- W konsekwencji wątek zablokowany modyfikuje stan zmiennej count i kończy obliczenia.
- Wątki 1 i 2 kontynuują modyfikowanie zmiennej współdzielonej do momentu, gdy spełnione są warunki końce wykonywania ich pętli.

# 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.