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

Rozwiązanie problemów wzajemnego wykluczania oraz producenta i konsumenta nie wyczerpuje wszystkich typowych zjawisk występujących w systemach współbieżnych. Kolejne to problem czytelników i pisarzy oraz pięciu ucztujących filozofów.

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).

Problem czytelników i pisarzy najprościej wytłumaczyć na przykładzie pewnej czytelni działającej w specyficzny sposób. Do czytelni może wchodzić dowolna liczba czytelników (w pewnych wariantach rozwiązania liczba czytelników również może być ograniczona) i wszyscy oni mogą równocześnie czytać pewien zbiór danych. Odczyt odbywa się więc współbieżnie. W pracy czytelni uczestniczą również pisarze, którzy mogą zmodyfikować zawartość zbioru. Pisarz może wejść do czytelni tylko wtedy, gdy jest ona opuszczona przez czytelników. Tylko jeden pisarz może na raz przebywać w czytelni.

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 problemu polega na opracowaniu tak zwanych protokołów wchodzenia i opuszczania czytelni dla czytelnika i pisarza. Moszna sobie wyobrazić, że do wspomnianej czytelni prowadzą drzwi i podczas ich przekraczania trzeba spełnić określone procedury, zarówno przy wejściu jak i wyjściu. Na slajdzie wskazane są warunki, jakie powinny spełnić protokoły czytelników i pisarzy. Warianty rozwiązania problemu mogą pozwolić na wchodzenie do czytelni dowolnej liczby czytelników lub wskazana jest ich maksymalna liczba. Podstawowe rozwiązania problemu, które zostaną dalej omówione preferują albo czytelników, albo pisarzy. Oznacza to, że albo pisarze albo czytelnicy mogą w tych rozwiązaniach być głodzeni, czyli nie dostawać w sposób „sprawiedliwy” dostępu do czytelni. Warto nadmienić, że rozwiązania problemów czytelników i pisarzy stosowane w nowszych systemach programowania współbieżnego (język Java lub język C#) wprowadzają dodatkowe mechanizmy „starające się” równoważyć dostęp czytelników i pisarzy do czytelni. W projektowaniu systemów współbieżnych należy dostrzec, że dany model komunikacji między aktorami systemu odpowiada rozwiązaniu problemu czytelników i pisarzy i zastosować odpowiednie środki programistyczne.

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
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

4

Przeanalizujmy pierwsze z podstawowych rozwiązań problemu – preferujące czytelników. Jest to rozwiązanie „szkolne”, wyjaśniające zasadę działania systemu z zastosowaniem semaforów. Później pokażę, że istnieją gotowe obiekty sytemu operacyjnego, które rozwiązują ten problem na wyższym poziomie abstrakcji. Rozważmy zasadę działania wątku pisarza. Pisarz w celu wejścia do czytelni musi przejąć semafor „wsem”. Jeśli semafor nie jest przejęty przez innego pisarza i innego czytelnika, to następuje wejście do czytelni, a po zmodyfikowaniu danych następnie jego zwolnienie. W tedy do czytelni może wejść inny pisarz, lub czytelnik.

Protokół wstępny czytelnika jest nieco bardziej złożony. Moment próby wejścia do czytelni jest zawarty w sekcji krytycznej chronionej przez semafor/mutex „x”. Wchodzenie do czytelni wykonuje na raz dokładnie jeden z czytelników (tylko jeden z procesów może wejść do sekcji krytycznej chronionej przez „x”). Podczas wchodzenia do czytelni inkrementowany jest licznik „readcount”. Pierwszy z czytelników (dla readcount == 1) próbuje przejąć semafor „wsem”. Jeśli w czytelni jest aktualnie pisarz, to czytelnik czeka na opuszczenie przez niego czytelni, a następnie przejmuje ten semafor. Gdy czytelnia jest pusta następuje po prostu przejęcie semafora „wsem”. Zakończenie protokołu wstępnego polega na opuszczeniu sekcji krytycznej chronionej przez „x”. Tak więc pierwszy czytelnik, któremu udało się wejść do czytelni powoduje jej zablokowanie dla pisarzy (przejęcie semafora „wsem”). Po opuszczeniu protokołu wstępnego czytelnicy nie są ograniczani żadnymi semaforami, czy blokadami i mogą współbieżnie dokonywać odczytu współdzielonego bufora. Po zakończeniu odczytu czytelnik wykonuje protokół wyjścia. Wychodzenie odbywa się znowu w sekcji krytycznej. W czasie opuszczania czytelni przez czytelnika jest dekrementowany licznik „readcount”. Ostatni z czytelników opuszczających czytelnie (gdy readcount == 0) zwalnia semafor (wsem) pozwalający na ew. zapis przez pisarza.

Omówione powyżej rozwiązanie problemu czytelników i pisarzy preferuje czytelników. Można zauważyć, że jeśli czytelnikom uda się przejąć semafor „wsem”, to dopóki wszyscy z nich nie opuszczą dobrowolnie czytelni, żaden z pisarzy nie może do niej wejść. Hipotetycznie stan przejęcia czytelni przez czytelników może trwać w nieskończoność. Dopóki choć jeden z czytelników znajduje się w czytelni, to żaden pisarz nie może do niej wejść.

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. To z kolei grozi **zagłodzeniem** pisarzy.

Slajd zawiera podstawowe uwagi dotyczące przedstawionego rozwiązania problemu czytelników i pisarzy. Pracę protokołów wstępnych omówiono w adnotacji do poprzedniego slajdu.

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
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

6

Drugie z podstawowych rozwiązań problemu czytelników i pisarzy preferuje pisarzy. Gdybyśmy porównali kod wątku pisarza z obecnego rozwiązania i kod czytelnika z poprzedniego, to są bardzo podobne. Pisarz tym razem musi „przejść” przez sekcję krytyczną chronioną przez semafor/mutex „y”. Pierwszy z pisarzy, który wejdzie do sekcji próbuje przejąć semafor „rsem”. Przejęcie tego semafora zablokuje możliwość wejścia do czytelni czytelnikom. Po opuszczeniu tej sekcji pisarz próbuje przejąć semafor „wsem”. Ten semafor zapewnia przebywanie wewnątrz czytelni dokładnie jednemu pisarzowi. Oczywiście, gdyby semafor był przejęty przez innego pisarza lub przez czytelnika, pisarz musi oczekiwać na jego zwolnienie. Po dokonaniu zapisu w czytelni pisarz zwalana semafor „wsem” i rozpoczyna procedurę opuszczania czytelni chronioną przez semafor „y”. Ostatni z pisarzy zwalnia semafor „rsem”.

Po stronie czytelnika protokół wejścia odbywa się w otoczeniu trzech sekcji krytycznych zagnieżdżonych jedna w drugiej. Sens ich stosowania będzie po kolei wytłumaczony. Czytelnik wchodzący do czytelni musi przejąć semafor „z”, potem semafor „rsem”, a następnie semafor „x” żeby w końcu zwiększyć licznik czytelników (readcount). Jeśli jest to pierwszy czytelnik, to następuje próba przejęcia semafora „wsem”. Przejęcie semafora „wsem” oznacza zablokowanie możliwości zapisu przez pisarzy. Czytelnik kończąc protokół wejściowy zwalania kolejno semafory „x”, „rsem”, „z” i wchodzi w strefę współbieżnego odczytywania danych wraz innymi potencjalnymi czytelnikami. Sekcja krytyczna chroniona przez semafor „x” odpowiada za wchodzenie tylko jednego czytelnika na raz do czytelni. Tam następuje policzenie czytelników, którzy weszli do czytelni. Sekcja krytyczna na semaforze „rsem” odgrywa rolę w „sporze” z pisarzem. Każdy czytelnik wchodzący do czytelni musi „na chwilę” przejąć semafor „rsem”, a potem go zwolnić. Gdy jednak ten semafor zostanie przejęty przez pisarza, to pełni on rolę blokady uniemożliwiającej wejście do czytelni kolejnym nowym czytelnikom. W odróżnieniu od wcześniejszego rozwiązania pisarz chcący wejść do czytelni może przejąć semafor „rsem” i tym samym zamknąć drzwi dla kolejnych czytelników. Protokół wyjściowy czytelnika chroniony jest tylko semaforem „x” i przy opuszczaniu czytelni przez ostatniego czytelnika następuje zwolnienie semafora „wsem”. Tak więc pisarz, gdy chce rozpocząć pisanie może zablokować napływ nowych czytelników do czytelni. Czytelnicy obecni w czytelni opuszczają ją z własnej woli. Po opuszczeniu czytelni przez ostatniego z czytelników, pisarz może rozpocząć pisanie. Co więcej konstrukcja pisarza (przejęcie semafora „rsem” przez pierwszego pisarza i zwolnienie przez ostatniego) pozwala na przebywanie w czytelni dowolnie długo bez jakiejkolwiek ingerencji czytelnika.

Skomentowania wymaga jeszcze obecność semafora „z”. Problem byłby poprawnie rozwiązany be jego udziału, jednak wprowadzenie jeszcze jednej zewnętrznej sekcji krytycznej zapewnia, że na kolejce do semafora „rsem” może być ustawiony tylko jeden czytelnik. Jeśli jakiś czytelnik „utknie” w oczekiwaniu na semafor „rsem”, to kolejni czytelnicy wchodzący do czytelni zostaną zatrzymani na semaforze „z”. Takie rozwiązanie daje możliwość szybszego przejęcia przez pisarza semafora „rsem”.

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*.

Slajd zwiera najważniejsze uwagi, które szerzej były omówione w komentarzu do poprzedniego slajdu.

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.

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

8

Problem czytelników i pisarzy w bibliotece pthreads można rozwiązać z zastosowaniem gotowych funkcji. Zdefiniowane zostały tzw. blokady czytelników i pisarzy. Aby zastosować taka blokadę trzeba najpierw utworzyć zmienną typu **`pthread_rwlock_t`** `i z zastosowaniem funkcji` **`pthread_rwlock_init`** `ją zainicjalizować (drugi parametr funkcji można ustawić na NULL, wtedy atrybuty blokady będą domyślne):`

```
pthread_rwlock_t rwlock;
int rc=0;
rc = pthread_rwlock_init(&rwlock, NULL);
```

Samo stworzenie sekcji kodu, w której ma być rozwiązany problem czytelników i pisarzy polega na „otoczeniu” jej na początku wywołaniami funkcji:

pthread_rwlock_rdlock(&rwlock); // gdy chcemy przejąć blokadę w celu odczytu (czytelnik)

lub

rc = pthread_rwlock_wrlock(&rwlock); // gdy chcemy przejąć blokadą w celu zapisu (pisarz)

// tu znajduje się sekcja czytania/zapisu

Kończąc ją wywołaniem funkcji:

pthread_rwlock_unlock(&rwlock);

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.

Slajd zawiera zestawienie dwóch pozostałych funkcji sterujących pracą blokady czytelników i pisarzy.

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)`

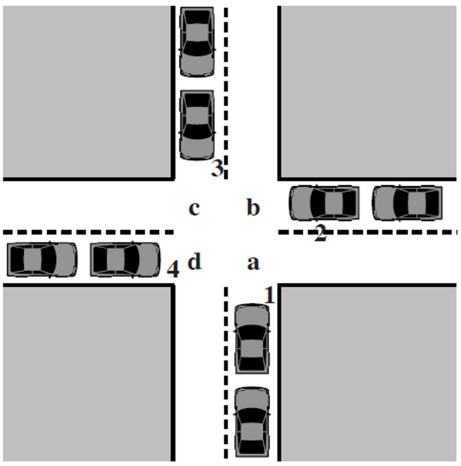
- Gdy blokada jest wolna lub zajęta do odczytu następuje jej zajęcie do odczytu. Gdy jest zajęta funkcja nie blokuje wątku bieżącego i zwraca kod błędu.

Skasowanie blokady

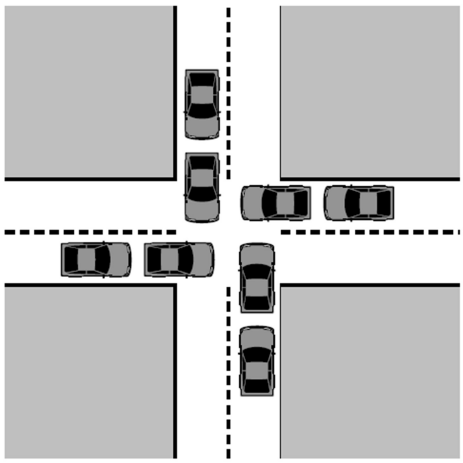
`int pthread_rwlock_destroy(pthread_rwlock_t *rwlock)`

Funkcje z przedrostkiem „try” pozwalają odwołać się do blokady w specjalny sposób. Jeśli blokada jest możliwa do przejęcia, to jest ona przez nie przejmowana. Jeśli jest przejęta przez inny wątek, to funkcja nie zawiesza działania wątku, tylko zwraca błąd. Możliwe jest wtedy zachowanie aktywności wątku, który chce przejąć blokadę. Blokada jest zasobem systemowym. Jeśli w programie nie jest już potrzebna, lub program kończy swoje wykonywanie, to powinna być zwrócona do systemu z zastosowaniem funkcji **`pthread_rwlock_destroy()`**.

Impas/Zakleszczenie



Możliwy impas/zakleszczenie



Impas/zakleszczenie

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

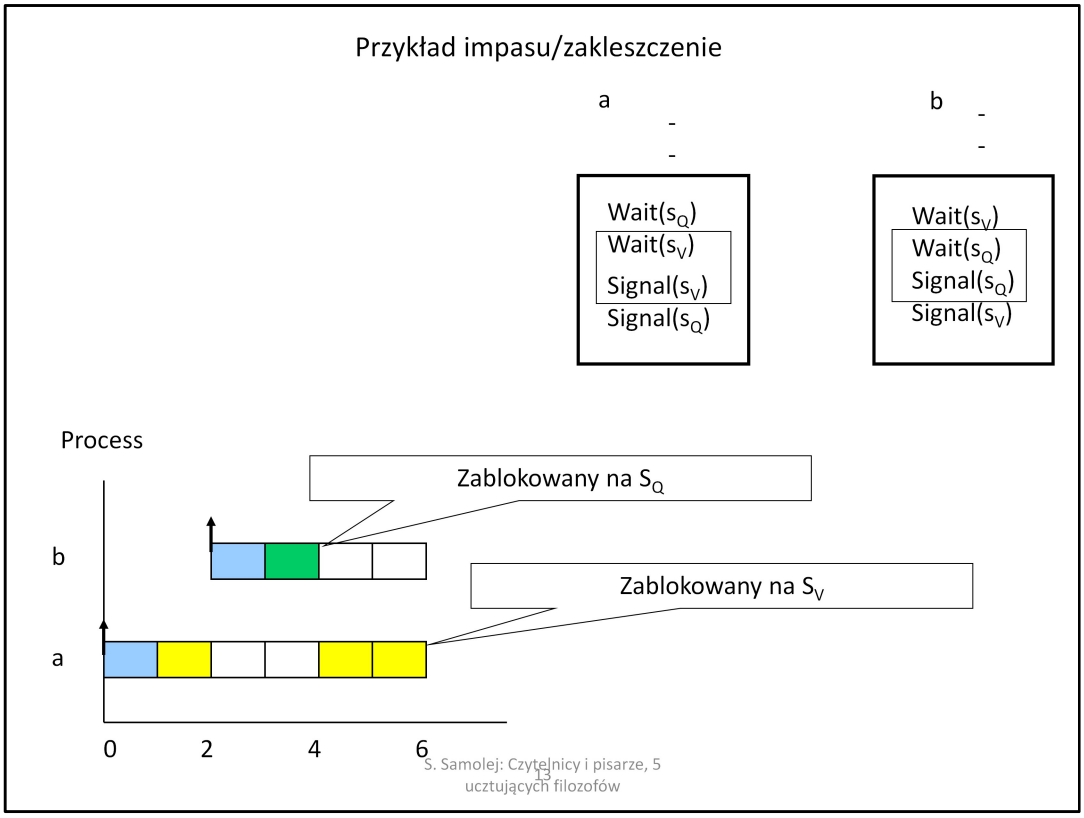
11

Przy rozwiązywaniu problemu czytelników i pisarzy wspomniałem o problemie głodzenia. Czytelnicy, bądź pisarze blokowani przez swoich oponentów nie mogą dokonywać odczytu/zapisu ponieważ ciągle do zasobu odwołują się inne wątki. Taki stan nazywa się głodzeniem. W sprzyjających okolicznościach głodzenie może być przerwane. Problem, o którym teraz będzie mowa ma bardziej drastyczne konsekwencje w systemach współbieżnych. Polega on na trwałym zablokowaniu systemu współbieżnego bez żadnej nadziei na jego odblokowanie (ożywienie). Zjawisko takie można zacząć prezentować na przykładzie skrzyżowania jak na slajdzie. Zakładamy, że samochody są zainteresowane tylko przejechaniem na wprost. Jeśli w jednym momencie samochody ze wszystkich stron ruszą, żeby przejechać skrzyżowanie, to zakleszczą się i w żaden sposób nie można już tej sytuacji odwrócić.

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.

Slajd pokazuje podstawowe cechy impasu/zakleszczenia.

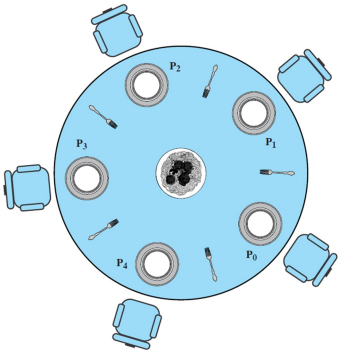


Napisanie oprogramowania, w którym jest impas jest dosyć proste. Wystarczy niefortunnie posłużyć się zagnieżdżonymi sekcjami krytycznymi. Na slajdzie widać 2 zadania a i b. Zadanie a zainteresowane jest najpierw przejęciem zasobu Q a potem posiadając tan zasób chce jeszcze przejąć zasób V. W tym celu rozpoczyna sekcje krytyczną na semaforze, który zabezpiecza wzajemne wykluczanie w dostępie do zasobu Q, a następnie w czasie trwania wcześniejszej sekcji rozpoczyna drugą sekcję krytyczną na semaforze, który zabezpiecza zasób V. Zadanie b jest zainteresowane najpierw przejęciem zasobu V i w czasie jego posiadania – przejęciem zasobu Q. W tym celu rozpoczyna sekcję krytyczną na semaforze, który zabezpiecza wzajemne wykluczanie na zasobie V, a następnie wewnątrz tej sekcji rozpoczyna drugą próbując przejąć semafor zabezpieczający wzajemne wykluczanie na zasobie Q. Jak dotąd sytuacja wydaje się niegroźna.

Problem następuje, jeśli zajdzie nieszczęśliwa, ale możliwa sekwencja zdarzeń w systemie współbieżnym. Załóżmy, że obliczenia prowadzi zadanie a i udało się mu przejąć zasób Q. Z pewnych przyczyn zadanie zostaje wywłaszczone, a w czasie jego wywłaszczenia obliczenia rozpoczyna zadanie b i przejmuje zasób V. Zadanie b próbuje następnie przejąć zasób Q. Nie może go jednak przejąć, bo został przejęty wcześniej przez zadanie a. Zadanie a kontynuuje jeszcze swoje obliczenia, ale w pewnym momencie potrzebuje otrzymać dostęp do zasobu V, który jest w posiadaniu zadania b. Żadne z zadań nie może kontynuować obliczeń. System jest trwale zablokowany i z blokady nie ma wyjścia.

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;
```


Problem 5 ucztujących filozofów nie jest odzwierciedleniem rozwiązania praktycznych zagadnień, jak w przypadku problemów wzajemnego wykluczania, producenta konsumenta, czy czytelników i pisarzy. Jest on traktowany jako swego rodzaju „benchmark” dla systemów programowania współbieżnego. Jeśli tworzony system pozwala prawidłowo opisać i rozwiązać problem 5 ucztujących filozofów, to z dużym prawdopodobieństwem jest systemem prawidłowo skonstruowanym.

Problem opisuje współbieżny system, w którym jest 5 graczy. Każdy z nich co pewien czas głodnieje i chciałby zjeść porcję jedzenia. Do rozpoczęcia jedzenia potrzebuje 2 widelców, które są po obu stronach talerza. Widelce są zasobami współdzielonymi z sąsiadem. Może się okazać, że nie można rozpocząć jedzenia, bo właśnie z widelca korzysta gracz obok. Po jedzeniu gracz odkłada widelce, żeby jeden z sąsiadów mógł rozpocząć jedzenie. Na slajdzie pokazano schemat programu pojedynczego filozofa. Najważniejsze są protokoły wstępne i końcowe, czyli strategie przejmowania i zwracania widelców.

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.

Slajd pokazuje zalecenia co do protokołów wstępnych i końcowych.

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);
}
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

```
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
```

16

Na początku rozważymy rozwiązanie problemu, które jest nieprawidłowe. Filozofowie są modelowani jako wątki/procesy, zaś widelce jako semafory, które dane procesy próbują przejąć. Każdy z filozofów rozpoczyna protokół wstępny, umówmy się od próby przejęcia lewego widelca. Wykonuje więc operację „sem_wait” na tym widelcu. Jeśli przejęcie się uda, to w dalszej kolejności filozof próbuje przejąć widelec po swojej prawej stronie. Przejęcie obu widelców umożliwia jedzenie. Po posiłku widelce zwracane są w odwrotnej kolejności, najpierw prawy a potem lewy (wykonanie operacji „sem_post”).

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.

Najważniejszym problemem, który pojawia się przy tak zaprojektowanym systemie jest możliwość impasu. Jeśli w systemie wszyscy gracze są współbieżni, to można sobie wyobrazić sytuację, kiedy wszyscy filozofowie w jednej chwili podniosą lewy widelec. Wtedy żaden z nich nie może wykonać drugiej części protokołu przystąpienia do jedzenia (przejęcie prawego widelca) i w systemie mamy impas/zakleszczenie.

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]);}}}...
```


S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

W drugim rozwiązaniu filozofowie są podzieleni na parzystych i nieparzystych. Parzyści mają taki sam protokół wstępny, jak w poprzednim przykładzie. Nieparzyści zaś rozpoczynają protokół wstępny od próby przejęcia tym razem najpierw prawego, a potem lewego widelca. Okazuje się, że takie rozwiązanie eliminuje zjawisko impasu.

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.

Na slajdzie zebrano najważniejsze uwagi dotyczące omówionego rozwiązania.

Takie rozwiązanie uznawane jest za mniej eleganckie, ponieważ kody filozofów różnią się od siebie (parzyści i nieparzyści zachowują się inaczej). Na zajęciach laboratoryjnych poproszeni zostaniecie o rozwiązanie problemu w inny sposób. W systemie można wprowadzić jeszcze jednego gracza – kelnera, modelowanego np. z zastosowaniem semafora zliczającego o początkowej wartości 4. Kelner dopuszcza do stołu tylko 4 filozofów na raz. Wtedy kody filozofów mogą wyglądać jak w pierwszym rozwiązaniu, ale z uwagi na niepełne obsadzenie stołu do impasu wtedy nie dochodzi.

Schemat cyklicznego sprawdzania wartości w systemie współbieżnym

```
pthread_mutex_lock(&work_mutex);
      while (work_area[0] == '\0' )
      {
          // ew. instrukcje w sekcji krytycznej
         pthread_mutex_unlock(&work_mutex);
         sleep(1);
         pthread_mutex_lock(&work_mutex);
      }
pthread_mutex_unlock(&work_mutex);
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

20

Pokazany fragment kodu może stanowić szablon fragmentu aplikacji, w której cyklicznie następuje sprawdzenie i/lub zmodyfikowanie pewnej współdzielonej zmiennej w systemie współbieżnym. Fragment kodu rozpoczyna się od próby przejęcia mutexa. Jeśli uda się go przejąć, to można zastosować instrukcję pętli while, a w wyrażeniu, które jest sprawdzane przed rozpoczęciem instrukcji zawrzeć warunek, kiedy oczekiwanie na pewne wydarzenie ma się zakończyć (W przykładzie jest sprawdzane, czy pierwszy element pewnej tablicy wynosi 0). W pętli można zawrzeć dalej kontynuowane w sekcji krytycznej instrukcje (np. ew. modyfikowanie zmiennej). Kolejną instrukcją w pętli jest zwolnienie mutexa. Potem następuje odczekanie pewnego przedziału czasowego i ponowna próba przejęcia mutexa. Jeśli uda się go przejąć, to przechodzi się do sprawdzenia warunku końca pętli. Rozwiązanie pozwala na cykliczne sprawdzenie stanu współdzielonej zmiennej

Podana konstrukcja zapewnia, że sprawdzanie warunku końca pętli i jakiekolwiek instrukcje w pętli poprzedzające zwolnienie mutexa odbędą się z zasadą wzajemnego wykluczania. Ostatnia linijka załączonego kodu kończy sekcję krytyczną, jeśli warunek w instrukcji while przestaje być prawdziwy.

W przykładzie pokazano zalążek jednej z możliwych metod wymiany informacji między komponentami systemu współbieżnego, kiedy wymiana informacji np. przez zmienną współdzieloną odbywa się z potwierdzeniem. Przykładowo, pobranie kolejnej danej jest warunkowane skonsumowaniem przez odbiorcę danej wcześniejszej.

Problem wymiany informacji „z potwierdzeniem”

```
#include <stdio.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>
void *thread_function2(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_function2, NULL);
   if (res != 0) { perror("Thread creation failed");
   exit(EXIT_FAILURE);}
   printf("Input some text. Enter 'end' to finish\n");
   pthread_mutex_lock(&work_mutex);
   while(!time_to_exit)
   {  fgets(work_area, WORK_SIZE, stdin);
      pthread_mutex_unlock(&work_mutex);
      pthread_mutex_lock(&work_mutex);
      while(work_area[0] != '\0')
      {  pthread_mutex_unlock(&work_mutex);
         sleep(1);
         pthread_mutex_lock(&work_mutex);
      }}
   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_function2(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);}
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

21

Tutaj pokazano rozwinięcie koncepcji omówionej na wcześniejszym slajdzie. Program symuluje wymianę informacji między dwoma współbieżnymi procesami z potwierdzeniem odebrania informacji. „Zewnętrzne” pętle kontrolują wymianę informacji pomiędzy nadawcą a odbiorcą, wewnętrzne zaś dokonują „odpytywania” kanału komunikacyjnego sprawdzając od strony nadawcy czy blok danych (tekst) został odebrany (tekst jest wtedy ustawiany na 0), a od strony odbiorcy, czy przesłano nowy blok danych (czy w buforze pojawia się nowy tekst). Przesłanie łańcucha „end” kończy pracę systemu (wątek – dziecko jest zamykany, główny wątek dowiaduje się, że wartość zmiennej time_to_exit wynosi 1 i również kończy działanie, czekając wcześniej na zamknięcie pracy wątku dziecka).

Taki model wymiany z potwierdzeniem ma mankament. Wymaga ciągłego monitorowania stanu zmiennych odzwierciedlających wymianę informacji (time_to_exit i work_area). Konsumuje to czas procesora oraz wymaga zastosowania odpowiedniej konstrukcji programu.

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

Slajd uzupełnia rozważania zawarte w notatkach do slajdu 21.

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 przejęcia 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.**

Slajd uzupełnia rozważania zawarte w notatkach do slajdu 21.

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).

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

25

Powstaje pytanie, czy omawiany wcześniej model komunikacji można zastąpić jakimś mechanizmem współbieżności, który usunąłby aktywne „odpytywanie” . Rozważmy konstrukcję, jak na slajdzie. Próbujemy tu połączyć zastosowane dwóch semaforów do rozwiązania zagadnienia wzajemnego wykluczania (semafor mutex) oraz synchronizacji (semafor cosyn). Proces P1 wytwarza porcję komunikatu, a P2 ma tę porcję odebrać. Modyfikowanie współdzielonej zmiennej X odbywa się w sekcji krytycznej. Wewnątrz tej sekcji zastosowano również drugi semafor (cosyn) do rozwiązania prostego schematu synchronizacji. Proces P2 powinien zawiesić swoje wykonywania (sem_wait(cosyn)) do czasu, gdy proces P1 wyprodukuje nową wartość X i potwierdzi to wykonaniem na semaforze operacji sem_post.

Pokazana konstrukcja ma jednak poważną wadę. Jeśli w którymś momencie pracy systemu proces P2 wejdzie do sekcji krytycznej i „zawiesi się” na semaforze cosyn, bo wartość jego licznika właśnie wnosi 0, to nie ma możliwości opuszczenia sekcji krytycznej chronionej przez semafor mutex. W konsekwencji proces P1 nigdy nie może wejść do sekcji krytycznej chronionej przez semafor mutex i w systemie następuje zakleszczenie.

WNIOSEK: mając do dyspozycji tylko semafory nie można bez wprowadzania mechanizmu poolingu dokonać wymiany informacji z potwierdzeniem.

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; `
```

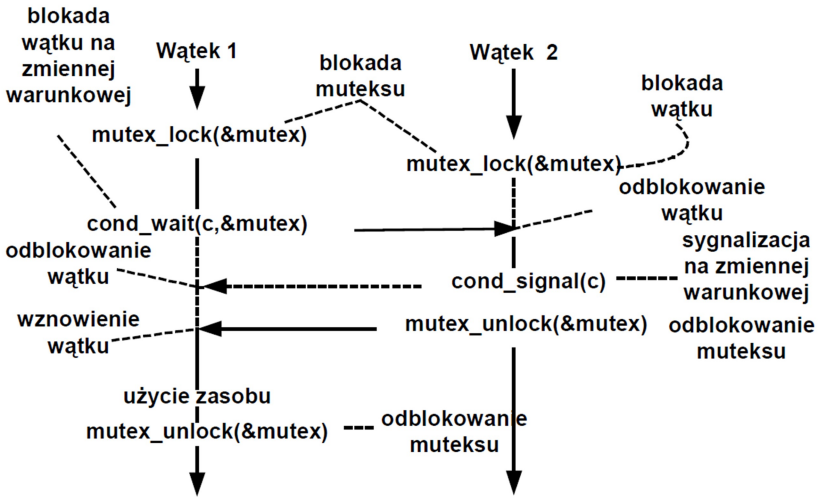
```
process P2;
  sem_wait (mutex);
     while(data_send==0)
      condition_wait(c_v);
     consume(X);
     data_send=0;
  sem_post (mutex);
end P2;
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

26

Na potrzeby możliwości dokonywania synchronizacji i komunikacji (z ew. potwierdzeniem) dokonywanej wewnątrz sekcji krytycznej został prowadzony zupełnie nowy mechanizm. Jest nim zmienna warunkowa. Dodatkowo, odblokowanie procesu zablokowanego na zmiennej warunkowej może odbyć się dopiero po spełnieniu określonego warunku. Wprowadza to nowe możliwości tworzenia modeli synchronizacji i komunikacji w systemach współbieżnych. Na slajdzie pokazano podstawowy schemat programowania synchronizacji z zastosowaniem zmiennej warunkowej. W pracy tego obiektu zawsze współpracują mutex, sama zmienna warunkowa oraz wyrażenie, które decyduje o odblokowaniu zmiennej warunkowej (w naszym wypadku następuje analiza stanu zmiennej data_send). Praca zmiennej warunkowej zawsze odbywa się w sekcji krytycznej. Jeśli chcemy zablokować proces na zmiennej warunkowej wywołujemy funkcję condition_wait(). Typowym rozwiązaniem jest zamknięcie wywołania tej funkcji w instrukcji while(). Warunek podany w instrukcji musi być prawdziwy, jeśli chcemy, żeby nastąpiło zablokowanie procesu na zmiennej warunkowej. Odblokowanie procesu zablokowanego na zmiennej warunkowej standardowo odbywa się z poziomu innego procesu. Musi on uruchomić operację condition_signal(). Wtedy następuje odblokowanie procesu zablokowanego na zmiennej, sprawdzenie, czy warunek zablokowania przestaje być prawdziwy i w zależności od stanu warunku następuje ponowne zablokowanie procesu (warunek jest prawdziwy) lub uruchomienie dalszych instrukcji procesu (warunek jest nieprawdziwy). Warto zaznaczyć, że zastosowanie instrukcji while() nie ma za zadanie prowadzenie w systemie mechanizmu odpytywania, jak to było pokazane na wcześniejszych przykładach. While() po odblokowaniu procesu pozwala ponownie sprawdzić, czy warunek odblokowania został spełniony. Pełni również rolę swego rodzaju zabezpieczenia, gdyby dany wątek samorzutnie uległ odblokowaniu (może się to zdarzyć w niektórych implementacjach systemów operacyjnych). Wtedy również warunek ponownie zostanie sprawdzony i proces zostanie zablokowany.

Graficzna reprezentacja zasady działania zmiennej warunkowej



S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

27

Schemat programowanie synchronizacji ze zmienną warunkową zakłada możliwość zablokowania procesu wewnątrz sekcji krytycznej. Jest to możliwe, ponieważ w implementacji takiej konstrukcji współbieżnej system będzie się zachowywał inaczej niż w przypadku semafora. Rozważmy schemat synchronizacji dwóch wątków z zastosowaniem zmiennej warunkowej pokazany na slajdzie. Wątek1 wchodzi do sekcji krytycznej i zostaje zablokowany na zmiennej warunkowej. Kiedy wątek1 był w sekcji krytycznej, wątek2 również chciał do niej wejść, ale ponieważ mutex został przejęty przez wątek1, to wątek2 został wstrzymany. Właściwością blokad na zmiennej warunkowej jest to, że w czasie gdy jakiś wątek zostaje zablokowany, to może przekazać sterowanie innemu wątkowi, który był stosował ten sam mutex. W omawianym scenariuszu po zablokowaniu wątku1 na zmiennej warunkowej przekazuje on sterowanie do wątku2. W taki sposób eliminuje się zakleszczenie. Wątek2 w pewnym momencie wykonuje operację cond_signal() na zmiennej warunkowej. Powoduje to odblokowanie wątku na niej zablokowanego. Jednak w większości implementacji realne odblokowanie pracy tego wątku (u nas wątku1) następuje po opuszczeniu przez inne wątki sekcji krytycznej (wykonanie mutex_ulock() przez wątek2).

Ważna cecha cond_signal(): operacja ta jest „bezpamięciowa”. To znaczy, jeśli jest jakiś wątek zablokowany, to zostanie do niego wysłana operacja „odblokowująca”. Natomiast jeśli wykonujemy cond_signal(), a w danej chwili nie ma zablokowanego wątku, to system w żaden sposób nie rejestruje próby odblokowania na przyszłość: wykonanie funkcji jest zapominane.

Podsumowując: zmienna warunkowa jest innym niż semafor mechanizmem synchronizacji. Odblokowanie procesu odbywa się na podstawie spełnienia (nie-spełnienia) określonego warunku. Operacje na zmiennych warunkowych wykonywane są tylko w sekcjach krytycznych. Nie następuje w takiej konstrukcji zakleszczenie, ponieważ zablokowane procesy przekazują sterowanie do innych procesów współdzielących mutex (tę samą sekcję krytyczną).

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).

To są uwagi podsumowujące cechy zmiennej warunkowej.

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

W standardzie POSIX zmienna wprowadzono możliwość stosowania zmiennych warunkowych. Podstawowe funkcje zestawiono w tabeli.

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.**

Zastosowanie zmiennej musi być poprzedzone jej inicjalizacją.

Funkcja pthread_cond_wait uruchamia zablokowanie procesu na zmiennej. Proszę zwrócić uwagę na to, że w implementacji POSIX jako parametry funkcji podaje się uchwyt do zmiennej warunkowej i do mutexa, który chroni jej wykonywanie.

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.

Funkcja pthread_cond_signal() zwalnia dokładnie jeden proces zablokowany na zmiennej warunkowej. O tym nie wspominałem, ale na zmiennej może być zablokowane więcej wątków (po prostu każdy z nich wykona cond_wait() na tej samej zmiennej w sekcji krytycznej chronionej przez ten sam mutex).

Funkcja pthread_cond_broadcast() zwalnia wszystkie wątki zablokowane na zmiennej warunkowej. Wszystkie one uruchamiają swoje obliczenia. Te, dla których warunek pozwala na przerwanie zablokowania są odblokowywane, pozostałe zaś ponownie są zawieszane.

Należy pamiętać, że funkcje signal i broadcast są bezpamięciowe. Jeśli w chwili ich wywołania nie będzie zablokowanych procesów, to wywołanie funkcji nie pozostawi żadnego śladu w systemie.

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)
```


Slajd jeszcze raz pokazuje schemat programowania synchronizacji z zastosowaniem zmiennych warunkowych.

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);
}
```


Na slajdzie pokazano pełną implementację synchronizacji 2 procesów z zastosowaniem zmiennej warunkowej. Producent inkrementuje zmienną shared_var i ustawia drugą zmienną data_sent w celu odblokowania konsumenta. To jest rozwiązanie tylko synchronizacji. Konsument zostanie odblokowany tylko jeśli jest zablokowany…

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
```


Druga cześć pełnego kodu synchronizacji z zastosowaniem zmiennej warunkowej.

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;
```

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

35

Stosując dwie zmienne warunkowe w jednej sekcji krytycznej możemy rozwiązać problem komunikacji z potwierdzeniem bez zastosowania poolingu. Nadawca kolejno (w sekcji krytycznej) wytwarza porcję informacji, ustala zmienną data_send na 1, odblokowuje pierwszą zmienną warunkową, zawiesza swoje działanie na drugiej zmiennej warunkowej. Odbiorca zaś zawiesza swoje działanie dopóki nie zostanie wysłana pierwsza porcja danych (modyfikacja zmiennej X). Po pobraniu danych wysyła sygnał odblokowujący nadawcę.

Taki model komunikacji nosi nazwę spotkań (randek). Można z jego zastosowaniem rozwiązać większość problemów programowania współbieżnego. Współbieżny język Ada stosuje go jako bazowy mechanizm do synchronizacji. W podręcznikach do tego języka jest pokazane, jak z mechanizmu spotkań „wyprowadzić" semafory i rozwiązania innych typowych schematów komunikacyjnych w systemach współbieżnych.

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);
}
```


Kolejne 2 slajdy pokazują pełną implementację schematu komunikacji z zastosowaniem spotkania.

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$
```

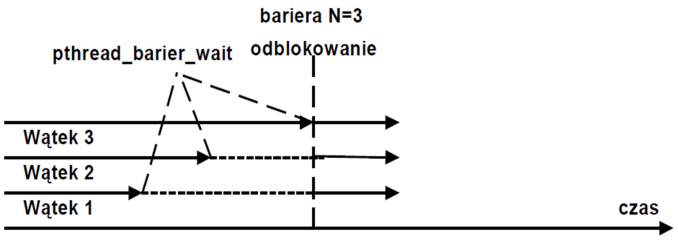
S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

37

Bariera – synchronizacja wielu wątków

- Szkic rozwiązania:

```
1 rendezvous
2
3 mutex.wait()
4 count = count + 1
5 mutex.signal()
6
7 if count == n: barrier.signal()
8
9 barrier.wait()
10 barrier.signal()
11
12 critical point
```



Bariera jest narzędziem do synchronizacji procesów działających w ramach grup. Wywołanie bariery powoduje zablokowanie zadania bieżącego do chwili gdy zadana liczba

wątków zadań nie wywoła tej procedury.

S. Samolej: Czytelnicy i pisarze, 5 ucztujących filozofów

38

Podstawowym mechanizmem synchronizacji wielu wątków jest powołanie puli wątków, a następnie oczekiwanie na zakończenie ich wszystkich poprzez wywołanie zestawu funkcji **pthread_join**(). Tak naprawdę w tym podejściu w chwili synchronizacji wątki są niszczone, a wznowienie obliczeń polega na ponownym ich utworzeniu. Problem bariery zachodzi, gdy chcielibyśmy zsynchronizować kilka wątków z puli, ale nie kończyć ich obliczeń. Przykładowo, 3 z puli 10 wątków musi się zsynchronizować, to znaczy wszystkie 3 muszą zakończyć pewną fazę obliczeń, żeby potem w poprawny sposób prowadzić swoje niezależne obliczenia. Problem można rozwiązać z zastosowaniem dość rozbudowanych konstrukcji opartych na semaforach. W standardzie POSIX jest jednak gotowa konstrukcja pozwalająca na taki sposób synchronizacji.

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.

Obsługa bariery jest nieskomplikowana. Po pierwsze trzeba ją zainicjować pewną początkową wartością oznaczając liczbę wątków, które mają być za jej pomocą zsynchronizowanych. W wątkach, które mają osiągnąć wspólny punkt synchronizacji w tym miejscu programu należy wywołać funkcję ptheread_barrier_wait(). Każde z wywołań funkcji dekrementuje licznik i dopóki jest on większy od 0 to zawiesza dany wątek. Wywołanie funkcji, w którym wartość licznika osiąga wartość 0 nie blokuje danego wątku i równocześnie odblokowuje wszystkie wątki uprzednio zablokowane na barierze. Czyli wywołanie przez ostatni najwolniejszy wątek funkcji ptheread_barrier_wait() odblokowuje wszystkie inne wątki, które w danym miejscu prowadzenia obliczeń miały się zsynchronizować.

Kasowanie bariery

`int pthread barrier destroy( pthread barrier t * barrier )`
`barrier` Zadeklarowana i zainicjowana zmienna typu **`pthread_barierr_t.`**
Funkcja powoduje skasowanie bariery

Funkcja usuwa barierę z systemu.

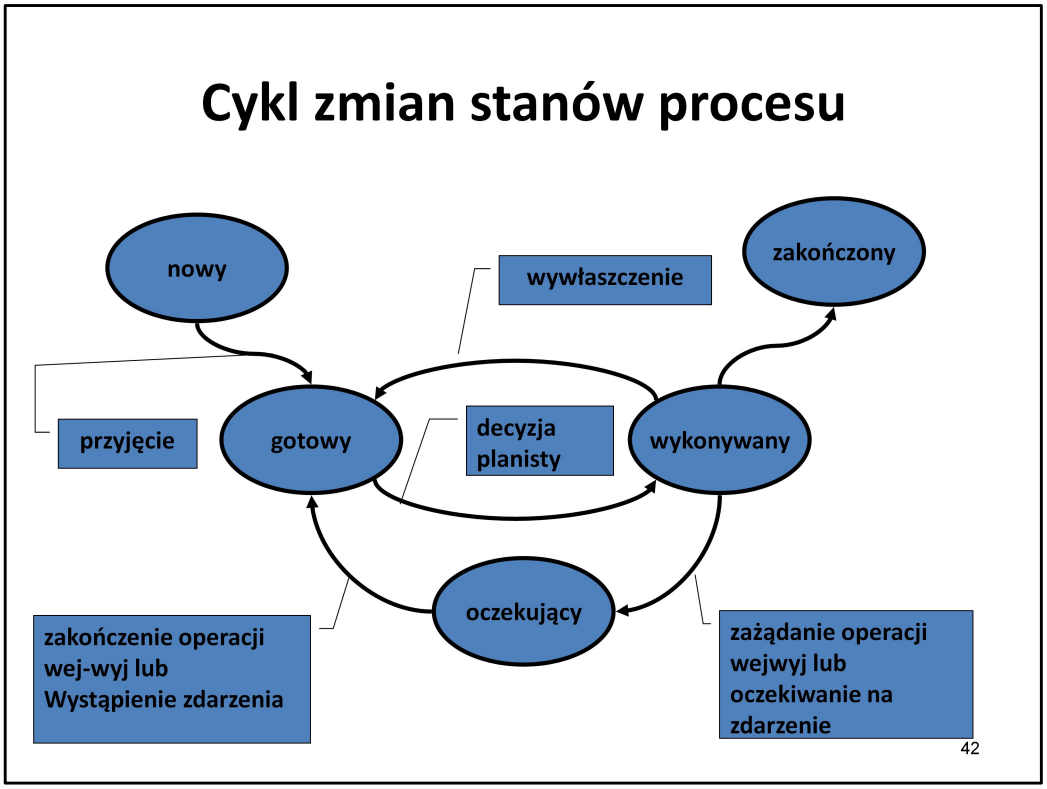
Przykład zastosowania bariery

```
#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;}
```


Aplikacja pokazuje proste zastosowanie barier. W systemie wyróżniono 2 wątki, które na pewnym etapie swoich obliczeń mają się zsynchronizować. W czasie startu systemu barira jest ustawiana na wartość 2. Analizując wykonywanie się programu można zauważyć, że oba wątki wykonują pierwszą część obliczeń, a następnie synchronizują się i wykonują drugą część obliczeń, zgodnie z intencją pracy bariery.



Do wprowadzenia następnego mechanizmu wzajemnego wykluczania potrzebne jest przypomnienie podstawowych stanów procesu w systemie operacyjnym. Proces uruchamiany jest w stanie „nowy". Po otoczeniu go odpowiednimi strukturami danych staje się on „gotowy" do wykonywania. Jeśli system operacyjny uzna, że może być wykonywany, to przechodzi do stanu „wykonywany". Opuszczenie tego stanu może się odbyć na 3 sposoby: zakończenie obliczeń („Zakończony"), przełączenie procesu i przeniesienie go do stanu „gotowy" lub zawieszenie jego wykonywania (np. oczekiwanie na mutex) do momentu pojawienia się jakiegoś zdarzenia („oczekujący"). Jeśli do zapewnienia wzajemnego wykluczania stosujemy mutexy, to wznowienie procesu odbywa się z pewnym naturalnym opóźnieniem. Po odblokowaniu mutexa proces najpierw przechodzi ze stanu „oczekujący" do „gotowy", a dopiero potem do „wykonywany". Zaletą tego rozwiązania jest minimalne zaangażowanie procesora. Reaguje on tylko na zdarzenia.

W pewnych wypadkach może nam zależeć na szybszej reakcji systemu na zwolnienie obiektu zapewniającego wzajemne wykluczanie. Takim wypadku można zastosować wirujące blokady. Wirująca blokada co do zasady zachowuje się jak mutex. Można ją próbować przejąć i zwolnić. Różnica w jej „zachowaniu" wynika z implementacji. W czasie wykonywania operacji lock() następuje przejęcie blokady lub zablokowanie na niej procesu (bo ktoś inny ją już wcześniej przejął). Zablokowany proces nie przechodzi tu jednak w stan „oczekujący" tylko przenoszony jest do stanu „gotowy". Po jakimś czasie proces jest wznawiany, sprawdzany jest stan blokady. Jeśli jest ona zajęta, to znowu proces jest przenoszony do stanu „gotowy"… Następuje „wirowanie" procesu pomiędzy stanami „gotowy" i „wykonywany"…

Taka konstrukcja powoduje samorzutne wznawianie procesu i konsumowanie dodatkowego czasu procesora. Gwarantuje jednak szybszą kontynuację obliczeń przez proces po uzyskaniu dostępu do blokady. Wirujące blokady są też w niektórych systemach operacyjnych jedynymi mechanizmami wzajemnego wykluczania pozwalającymi współdzielić zmienne pomiędzy procedurami obsługi przerwań a procesami.

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

Wirującą blokadę trzeba zainicjalizować…

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`**.

Próba przejęcia może się odbywać z zastosowanie funkcji lock (proces się blokuje, jeśli nie uzyska dostępu do blokady) lub trylock (jeśli blokada jest do przejęcia, to następuje jej przejęcie, jeśli nie funkcja zwraca błąd i sterowanie do procesu). Funkcja trylock może posłużyć do utrzymywania żywotności w systemie pomimo przedłużającego się głodzenia zadań.

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.

Zwolnienie blokady odbywa się z zastosowaniem funkcji unlock. Blokadę po zakończeniu aplikacji należy zwrócić do systemu operacyjnego (destreoy).

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); }
```


Na slajdzie pokazano przykład aplikacji stosującej wirującą blokadę do blokowania dostępu. Rozwiązywany jest tu klasyczny problem wzajemnego wykluczania. Warto zwrócić uwagę na zastosowanie wersji funkcji z przedrostkiem „try” do sprawdzania, czy można przejąć blokadę.

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.

Najważniejsze uwagi do kodu ze slajdu 46.