Grupa lab.
5

Data ćwiczenia
27.11.2023

Laboratorium 8

Imie i nazwisko
Łukasz Sawina

Cel

Celem ćwiczenia było poznanie realizacji synchronizacji przy pomocy zmiennych warunku w C oraz Javie.

Bariera C

Mechanika bariery polega na oczekiwaniu wątków przed barierą do momentu aż wszystkie wątki przed nią trafią i dopier wtedy przepuszczenie ich przez barierę. Tutaj już realizuję wersję rozszerzona z możliwością tworzenia wielu barier (w barierach można ustawić ile wątków musi trafić przed, aby została zwolniona), jednak przykład będzie pokazany na podstawowej barierze.

Do reprezentacji bariery utworzyłem strukturę *typ_bariery*, która zawiera w sobie odpowiednie zmienne jak ilość wątków, która obecnie oczekuje, liczba wymaganych wątków, zmienną warunkową oraz mutex.

```
typedef struct
{
   int liczba_watkow;
   int liczba_watkow_wywolujaca;
   pthread_cond_t cond_v;
   pthread_mutex_t mutex;
} typ_bariery;
```

Dodatkowo do przygotowanego kodu dodałem opcję przekazywania jako parametr bariery, na której są wykonywane operacje. W obsłudze bariery mamy dwie funkcje *bariera_init()* oraz *bariera()*, pierwsza z nich odpowiada za inicjalizację bariery, a druga za samą funkcjonalność bariery.

```
void bariera_init(int l_w, typ_bariery* tb)
{
   tb->liczba_watkow = l_w;
   tb->liczba_watkow_wywolujaca = 0;
   pthread_cond_init(&tb->cond_v, NULL);
   pthread_mutex_init(&tb->mutex, NULL);
}
```

```
void bariera(typ_bariery* tb)
{
   pthread_mutex_lock(&tb->mutex);
   tb->liczba_watkow_wywolujaca++;

   if(tb->liczba_watkow > tb->liczba_watkow_wywolujaca)
   {
      pthread_cond_wait(&tb->cond_v, &tb->mutex);
   }
   else
   {
      pthread_cond_broadcast(&tb->cond_v);
      tb->liczba_watkow_wywolujaca = 0;
   }

   pthread_mutex_unlock(&tb->mutex);
}
```

bariera_init() ustawia odpowiednie zmienne na ilość wątków potrzebnych do dotarcia do bariery, aby zostałą otwarta oraz inicjalizuje zmienne *pthread_cond_t* oraz *mutex*.

Funkcja *bariera()* na początku blokuje mutex, aby bezpiecznie porównać wartości, jeśli ilość wątków przed barierą jest mniejsza niż wymagana, nasz wątek oczekuje na wysłanie sygnału przy pomocy *pthread_cond_wait()*. W przeciwnym wypadku, wysyłany jest sygnał do wszystkich wątków oczekujących o zwolnieniu blokady oraz zerowana jest liczna oczekujących wątków. Na końcu mutex jest uwalniany.

Blokada zostałą utworzona jako zmienna globalna, do której będą się odnosić poszczególne funkcje, na początku musi zostać zainicjalizowana, a następnie w funkcji wątku wywołana z przekazanym wskaźnikiem do blokady.

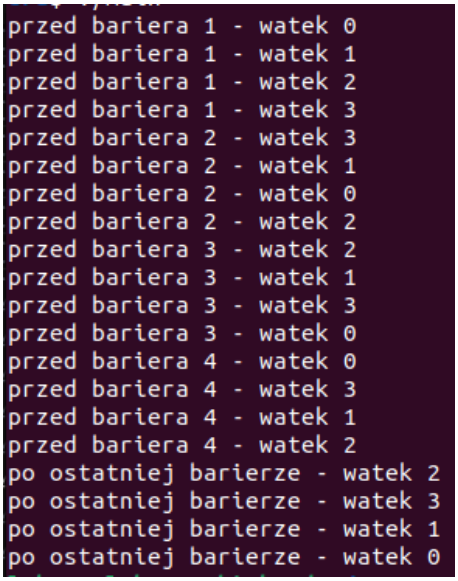
```
bariera_init(LICZBA_W, &b1);
bariera_init(2, &b2);

for(i=0; i<LICZBA_W; i++ ) {
  pthread_create( &watki[i], NULL, cokolwiek, (void *) &indeksy[i] );
}

for(i=0; i<LICZBA_W; i++ ) pthread_join( watki[i], NULL );
```

Jak widać utworzone i zainicjalizowane zostały dwie blokady, pierwsza z nich będzie oczekiwać na wszystkie wątki, a druga na 2.

```
moj_id = *( (int *) arg );

printf("przed bariera 1 - watek %d\n",moj_id);

bariera(&b1);

 printf("przed bariera 2 - watek %d\n",moj_id);

bariera(&b1);
```

Następnie w funkcji wątku wywoływana jest funkcja bariera z przekazanym adresem bariery. W przykładzie każdy wątek wywołuje 4 razy funkcję bariery dla *b1*.

Wynik programu:



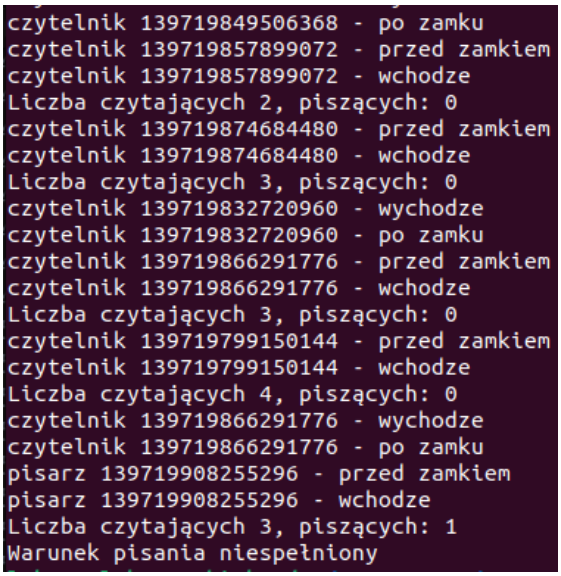
Jak widać przed każdą barierą muszą pojawić się wszystkie wątki i dopiero wtedy mogą one przejść do kolejnej.

Czytelnik Pisarz C – wersja ze zmiennymi warunku

Problem polega na tym, aby zapewnić funkcjonalność czytelni, W czytelni mogą być jednocześnie tylko pisarze lub czytelnicy.

Początkowo pojawia się problem, ponieważ nie mamy żadnej mechaniki wpuszczania osób do czytelni, dlatego wchodzą do niej kto pierwszy ten lepszy.

Wynik błędnego programu:



Jak widać trafiła się sytuacja kiedy w czytelni znaleźli się pisarze oraz czytelnicy.

Głównym zadaniem było wprowadzenie zabezpieczeń oraz funcjonalności oczekiwania na zwolnienie się czytelni, implementując odpowiednio przygotowany wcześniej pseudokod.

W funkcjonalności czytelnika przed samym czytaniem (funkcja *czytam()*) chce on wejść do czytelni (funkcja *my_read_lock_lock()*), naszym zadaniem jest sprawdzenie czy możliwe jest wejście czytelnika do czytelni oraz sprawienie aby ewentualnie oczekiwał na swoją kolej.

```
int my_read_lock_lock(czytelnia_t* czytelnia_p){
   pthread_mutex_lock(&czytelnia_p->mutex);

   if(czytelnia_p->liczba_pis > 0 || (czytelnia_p->o_p >0))
   {
      czytelnia_p->o_c++;
      pthread_cond_wait(&czytelnia_p->cond_c, &czytelnia_p->mutex);
      czytelnia_p->o_c--;
   }

   czytelnia_p->liczba_czyt++;
   pthread_cond_signal(&czytelnia_p->cond_c);
   pthread_mutex_unlock(&czytelnia_p->mutex);
}
```

Na początku mutex jest blokowany, aby bezpiecznie sprawdzić warunki oraz ewentualnie wprowadzić zmiany w zmiennych. Jeśli w czytelni już jest jakiś pisarz lub jakiś oczekuje na wejście wątek jest chwilowo blokowany, aż do zwolnienia. W przeciwnym wypadku liczba czytelników jest zwiększana oraz wysyłany jest sygnał do zmiennej warunkowej dla czytelników.

Po skończeniu operacji czytaj, czytelnik wychodzi przez co musi zmienić ilość czytelników w środku.

```
int my_read_lock_unlock(czytelnia_t* czytelnia_p){
   pthread_mutex_lock(&czytelnia_p->mutex);
   czytelnia_p->liczba_czyt--;

   if(czytelnia_p->liczba_czyt == 0)
      pthread_cond_signal(&czytelnia_p->cond_p);

   pthread_mutex_unlock(&czytelnia_p->mutex);
}
```

Dodatkowo w sytuacji, gdy z czytelni wychodzi ostatni czytelnik wysyła sygnał do ewentualnych oczekujących pisarzy, że mogą wejść do środka, ponieważ wszyscy już wyszli.

W przypadku pisarzy jest podobnie, przed rozpoczęciem pisania muszą spróbować wejść do środka.

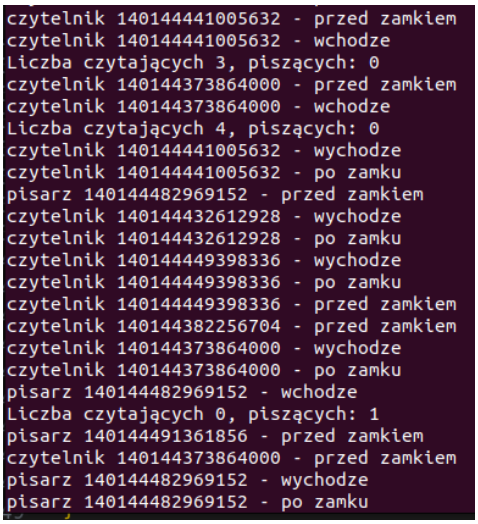
```
int my_write_lock_lock(czytelnia_t* czytelnia_p){
   pthread_mutex_lock(&czytelnia_p->mutex);

   if((czytelnia_p->liczba_czyt+czytelnia_p->liczba_pis) > 0)
   {
         czytelnia_p->o_p++;
      pthread_cond_wait(&czytelnia_p->cond_p, &czytelnia_p->mutex);
      czytelnia_p->o_p--;
   }
   czytelnia_p->liczba_pis++;
   pthread_mutex_unlock(&czytelnia_p->mutex);
}
```

Tutaj podobnie jak u czytelnika, sprwadzamy czy w środku jest jakaś osoba, jeśli tak to oczekujemy na zwolnienie się czytelni. Jeśli jednak jest pusto to zwiększa ilość pisarzy w czytelni.

Po skończonym pisaniu zmniejsza ilość pisarzy w czytelni oraz informuje odpwoiednie osoby o możliwości wejścia. Jeśli przed czytelnią oczekują jacyś czytelnicy mają oni pierwszeństwo, w przeciwnym wypadku wchodzi kolejny pisaż.

```
int my_write_lock_unlock(czytelnia_t* czytelnia_p){
   pthread_mutex_lock(&czytelnia_p->mutex);
   czytelnia_p->liczba_pis--;

   if(czytelnia_p->o_c > 0)
      pthread_cond_signal(&czytelnia_p->cond_c);
   else
      pthread_cond_signal(&czytelnia_p->cond_p);

   pthread_mutex_unlock(&czytelnia_p->mutex);
}
```

Wynikiem programu jest nieprzerwana interakcja między pisarzami i czytelnikami, w której nikt sobie nie przeszkadza.



Tak jak widać powyżej, w czytelni znajduje się 4 czytelników, przed czytelnią pojawia się pisarz jednak nie wchodzi do środka, oczekuje dopiero aż wszyscy z czytelni wyjdą i dopiero wtedy może wejść. Po wyjściu pisarza dopiero nowi czytelnicy mogą na nowo wejść do środka.

W programie potrzebny jest mechanizm sprawdzania czy ktoś oczekuje na wejście, w samym C nie ma takiej funkcjonalności, dlatego potrzebna była jakaś alternatywa. W tym przypadku jest to dodatkowy licznik osobny dla czytelnikóœ i pisarzy, który odpowiednio się zwiększa i zmniejsza przy oczekiwaniu na wejście.

Czytelnik Pisarz C – wersja rwlock

W kolejnej wersji programu zamiast wykorzystywać zmienne warunku, możemy użyć już wbudowany system blokowania wejścia do jakiejś sekcji, wykorzystując pthread_rwlock_t.

W tej wersji programu nie używamy zmiennych pthread_cond_t ani całej funkcjonalności z nimi związanymi.

Tym razem w momencie gdy czytelnik wchodzi oraz wychodzi na początku oraz końcu ustawia blokadę na odczyt oraz ją zwalnia przy pomocy *pthread_rwlock_rdlock* oraz *pthread_rwlock_unlock.*

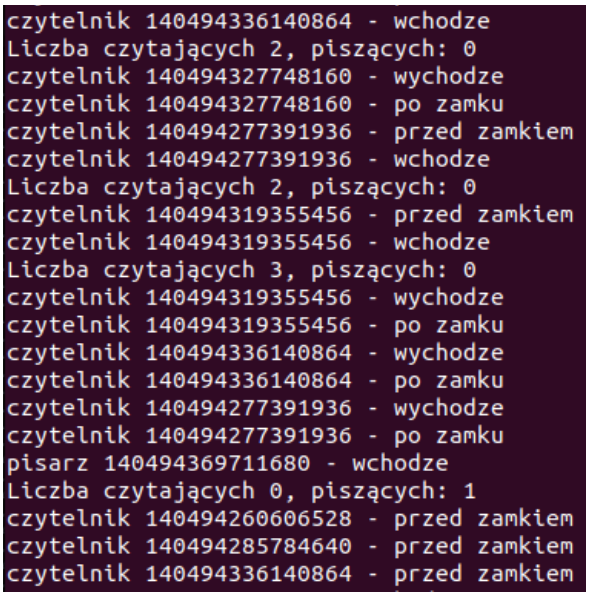
```
int my_read_lock_lock(czytelnia_t* czytelnia_p){
   pthread_rwlock_rdlock(&czytelnia_p->lock);
   pthread_mutex_lock(&czytelnia_p->mutex);
   czytelnia_p->liczba_czyt++;
   pthread_mutex_unlock(&czytelnia_p->mutex);
}

int my_read_lock_unlock(czytelnia_t* czytelnia_p){
   pthread_mutex_lock(&czytelnia_p->mutex);
   czytelnia_p->liczba_czyt--;
   pthread_mutex_unlock(&czytelnia_p->mutex);
   pthread_rwlock_unlock(&czytelnia_p->lock);
}
```

Podobnie pisarz, tylko zamiast blokowania odczytu, blokuje on zapis przy pomocy *pthread_rwlock_wrlock*.

```
int my_write_lock_lock(czytelnia_t* czytelnia_p){
   pthread_rwlock_wrlock(&czytelnia_p->lock);
   pthread_mutex_lock(&czytelnia_p->mutex);
   czytelnia_p->liczba_pis++;
   pthread_mutex_unlock(&czytelnia_p->mutex);
}

int my_write_lock_unlock(czytelnia_t* czytelnia_p){
   pthread_mutex_lock(&czytelnia_p->mutex);
   czytelnia_p->liczba_pis--;
   pthread_mutex_unlock(&czytelnia_p->mutex);
   pthread_rwlock_unlock(&czytelnia_p->lock);

}
```

Dzięki wykorzystaniu funkcjonalności pthread_rwlock pomijamy wszystkie sprawdzenia warunków oraz dodatkowe zmienne sprwadzające czy ktoś oczekuje. Wystarczy tylko ustawić odpowiedni lock w odpowiednije sytuacji. Zmniejsza to ilość napisanego kodu i to drastycznie.

Wynikiem programu jest ponownie nieprzerwana współpraca czytelników oraz pisarzy.



Czytelnik Pisarz Java

Kolejnym zadaniem było zrealizowanie identycznego programu jednak w tym przypadku w języku Java. Java podobnie jak C mechanizm mutexów oraz zmiennych warunku, tylko pod innymi nazwami. Dodatkowo zmienne są od razu przypisywane do „mutexu”, czyl nie wymagają przekazywania ich w zmiennej jak w C.

Cała funkcjonalność czytelni została zawarta w klasie Czytelnia, posiada ona konieczne zmienne do zarządzania czytelnią.

```
private int liczba_czyt = 0;
private int liczba_pis = 0;
private int piszacych = 0;
private Lock lock ;
private Condition condC;
private Condition condP;

public void init()
{
   lock = new ReentrantLock();
   condC = lock.newCondition();
   condP = lock.newCondition();
}
```

Dodatkowo posiada metodę init(), która odpowiada za inicjalizację całej czytelni.

Ponownie czytelnik oraz pisarz posiadają bardzo podobne mechanizmy wejścia oraz wyjścia z czytelni.

```
public void chce_czytac() throws InterruptedException
{
   lock.lock();

   try {
      if (liczba_pis > 0 || o_p > 0) {
         o_c++;
         condC.wait();
         o_c--;
      }

      liczba_czyt++;
      condC.signal();
   } finally {
      lock.unlock();
   }
}

public void czytam()
{
   System.out.println("Liczba czytających " + liczba_czyt + ", piszących: " +
liczba_pis);

   if (liczba_pis > 1 || (liczba_pis == 1 && liczba_czyt > 0) || liczba_pis < 0 ||
liczba_czyt < 0) {
      System.out.println("Warunek czytania niespełniony");
      System.exit(0);
   }

   try {
      Thread.sleep(300000);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}

public void koniec_czytania()
{
   lock.lock();

   try {
      liczba_czyt--;

      if (liczba_czyt == 0) {
         condP.signal();
      }
   } finally {
      lock.unlock();
   }
}
```

Czytelnik na początku chce wejść do czytelni, więc jak poprzednio sprawdza, czy jakiś pisarz się tam znajduje oraz czy któryś pisarz nie oczekuje na wejście. Lock posiada w sobie metodę hasWaiters(Condition) jednak z jakiegoś powodu w IntelliJ IDEA nie istnieje taka opcja, przez co nie chce się uruchomic cały program. Oczywiście problem ten można obejść podobnie jak w C, przy pomocy odpowiednich zmiennych pomocniczych. Po skończeniu czytania zwalnia swoje miejsce i informuje o tym oczekujących pisarzy jeśli był ostanim w czytelni.

Tak samo w przypadku pisarza

```
public void chce_pisac() throws InterruptedException
{
   lock.lock();

   try {
      if (liczba_czyt + liczba_pis > 0) {
         o_p++;
         condP.wait();
         o_p--;
      }

      liczba_pis++;
   } finally {
      lock.unlock();
   }
}

public void pisze()
{
   System.out.println("Liczba czytających " + liczba_czyt + ", piszących: " +
liczba_pis);

   if (liczba_pis>1 || (liczba_pis==1 && liczba_czyt>0) || liczba_pis<0 ||
liczba_czyt<0 ) {
      System.out.println("Warunek pisania niespełniony");
      System.exit(0);
   }

   try {
      Thread.sleep(300000);
   } catch (InterruptedException e) {
      e.printStackTrace();
   }
}

public void koniec_pisania() throws InterruptedException
{
   lock.lock();

   try {
      liczba_pis--;

      if (o_c > 0) {
         condC.signal();
      } else {
         condP.signal();
      }
   } finally {
      lock.unlock();
   }
}
```

Tutaj podobnie jak wcześniej, pisarz sprawdza czy może wejść, po skończeniu pisania sprawdza komu wysłać informację o swoim wyjściu.

Wynikiem programu jak wcześniej nieprzerwana współpraca czytelnika oraz pisarza:

```
Liczba czytających 3, piszących: 0
czytelnik 36 - przed zamkiem
czytelnik 32 - przed zamkiem
czytelnik 41 - wychodze
czytelnik 42 - po zamku
czytelnik 38 - przed zamkiem
pisarz 39 - przed zamkiem
czytelnik 44 - przed zamkiem
czytelnik 33 - wychodze
czytelnik 35 - po zamku
czytelnik 34 - wychodze
czytelnik 43 - po zamku
pisarz 31 - wchodze
Liczba czytających 0, piszących: 1
czytelnik 37 - przed zamkiem
pisarz 46 - przed zamkiem
pisarz 45 - przed zamkiem
pisarz 40 - wychodze
pisarz 31 - po zamku
czytelnik 36 - wchodze
Liczba czytających 1, piszących: 0
czytelnik 32 - wchodze
Liczba czytających 2, piszących: 0
czytelnik 41 - wchodze
```

Główna różnica jest jedynie w składni, bo schematem nic to się nie różni. Oczywiście oba programy się różnią ponieważ Java jest obiektowym językiem, ale merytorycznie robią to samo, tak samo tylko z innymi nazwami.

Wnioski

Możliwość stosowania zmiennych warunku pozwala nam uniknąć mechanizmu busy waiting. Mechanika automatycznego uwalniania oraz wracania do muteksu w sytuacji otrzymania sygnału dużo bardziej usprawnia nasz program. Najważniejszym aspektem jest brak dodatkowych, zbędnych operacji w sytuacji, gdy wątek musi czekać oraz ciągłego sprawdzania czy już może wrócić do działania.

Dodatkową opcją zamiast pthrad_cond_t można wykorzystać gotowe API pthread_rwlock_t, które zostało zaprojektowane do konkretnych problemów wykonywania zadań współbieżnie. Pozwala to nam na określenie co dokładnie blokujemy, czy odczty czy zapis. W tym przypadku w problemie czytelników i pisarzy nie potrzebujemy sprawdzania warunków wejścia, jedynie czytelnik wchodząc do czytelni blokuje odczyt, pisarz zapis a po wyjściu zwalniają swoją blokadę. Krótsze rozwiązanie, jednak w przypadku ptrhead_cond_t mamy większą swobodę w określaniu warunków oraz mechaniki pracy programu.