Mechanizm bariery dwustopniowej w synchronizacji procesów

Bartosz Bednarczyk

22 stycznia 2016

Motywacja

Podczas zajęć z systemów operacyjnych omówiliśmy wiele metod synchronizacji procesów. W tym sprawozdaniu przedstawię kolejną metodę synchronizacji, zwaną **barierą**, ale najpierw spójrzmy na poniższe problemy i spróbujmy zauważyć w nich pewne podobieństwo.

- Metoda eliminacji Gaussa. Dzielimy macierz na kilka części i równolegle wykonujemy zadane obliczenia.
- Problem wyścigu konnych. Wyścig składa się z N rund po jednym okrążeniu. Kolejna runda zaczyna się w momencie, gdy wszystkie konie znajdują się w boksach.
- Problem ortogonalizacji Grama-Schmidta.
- Szukanie liczb pierwszych za pomocą sita Eratostenesa.

Czym jest bariera?

W każdym z powyższych zadań spotykamy następujący problem: każde zadanie dzielmy na podzadania, których wyniki łączymy w jeden na samym końcu. Nie możemy określić wyniku całego zadania bez wcześniejszego połączenia mniejszych podzadań.

Przejdźmy do abstrakcyjnego pojęcia bariery. Pomyślmy sobie o domu, który ma jedne drzwi wejściowe i jedne wyjściowe. Do takiego domu wchodzi kolejno N gości. Kiedy wejdzie ich już N, to zamykamy drzwi wejściowe i otwieramy, początkowo zamknięte, drzwi wyjściowe. Mechanizm bariery jest czymś w rodzaju takiego domu, w którym gośćmi są procesy. Za pomocą tego mechanizmu możemy skutecznie blokować zakończenie podanej operacji do czasu, aż inne procesy zakończą swoje pozadania.

Przykładowa implementacja bariery z objaśnieniem

Przykładową implementację bariery umieściłem w pliku „bariera.c". W kodzie mamy dwa semafory o nazwach doorIn oraz doorOut. Początkowo semaforowi doorIn jest przypisana wartość N. Oznacza to, że „przepuści on przez drzwi" dokładnie N procesów. Kolejne procesy nie będą wpuszczane do środka. Semafor o nazwie mutex gwarantuje dostęp do licznika procesów. Po jego zdobyciu zwiększmy licznik, a kiedy nie potrzebujemy już danych, to oddajemy je zwalniając mutex. Kiedy N-ty proces zwiększa licznik pomiędzy drzwiami, to otwiera on semafor doorOut, który początkowo ustawiony na 0 (każdy proces się na nim zablokuje). Kiedy pojawia się N-ty proces, to otwiera on drzwi tylko dla jednego procesu. Każdy proces przechodzący przez doorOut od razu zamyka drzwi i otwiera je dla kolejnego procesu. W ten sposób procesy przechodzą sznurem przez drzwi wyjściowe. Na samym końcu, kiedy ostatni proces wyszedł drzwiami wyjściowymi, barierę ustawiamy na stan początkowy.

Testy

Naszą barierę można przetestować próbując zasymulować wyścig konny. Poprawności programu będzie dowodzić to, że nie będzie sytuacji, gdy jeden z koni przebiega metę i dalej biegnie (nie czekając na innych zawodników). Program był uruchamiany wielokrotnie i nie zaobserwowałem by taki przypadek miał miejsce. Poniżej umieszczam przykładową symulację dla pięciu zawodników.

```
vagrant@vagrant-ubuntu-trusty-64:~/Palacze$ ./prog 5
Rozpoczynam wyscig 5 rund i 5 koni
Runda 1: kon 1 dobiegl do mety
Runda 1: kon 4 dobiegl do mety
Runda 1: kon 3 dobiegl do mety
Runda 1: kon 2 dobiegl do mety
Runda 1: kon 5 dobiegl do mety
Runda 2: kon 1 dobiegl do mety
Runda 2: kon 4 dobiegl do mety
Runda 2: kon 3 dobiegl do mety
Runda 2: kon 2 dobiegl do mety
Runda 2: kon 5 dobiegl do mety
Runda 3: kon 1 dobiegl do mety
Runda 3: kon 4 dobiegl do mety
Runda 3: kon 3 dobiegl do mety
Runda 3: kon 2 dobiegl do mety
Runda 3: kon 5 dobiegl do mety
Runda 4: kon 1 dobiegl do mety
Runda 4: kon 4 dobiegl do mety
Runda 4: kon 3 dobiegl do mety
Runda 4: kon 2 dobiegl do mety
Runda 4: kon 5 dobiegl do mety
Runda 5: kon 1 dobiegl do mety
Runda 5: kon 4 dobiegl do mety
Runda 5: kon 3 dobiegl do mety
Runda 5: kon 2 dobiegl do mety
Runda 5: kon 5 dobiegl do mety
Wyscig skonczony
```

Dodatek. Kod źródłowy bariery w języku C

```
// Bartosz Bednarczyk

typedef struct {
  int maximumNumberOfGuests, counter;
  sem_t doorIn, doorOut, mutex;
} Barrier;

void init(Barrier* barrier, int numberOfGuests) {
  barrier->maximumNumberOfGuests = numberOfGuests;
  barrier->counter = 0;
  sem_init(&barrier->mutex, 1, 1);
  sem_init(&barrier->doorIn, 1, numberOfGuests);
  sem_init(&barrier->doorOut, 1, 0);
}

void wait(Barrier* barrier)
{
  sem_wait(&barrier->doorIn);
  sem_wait(&barrier->mutex);
  barrier->counter++;

  if(barrier->counter == barrier->maximumNumberOfGuests)
    sem_post(&barrier->doorOut);

  sem_post(&barrier->mutex);
  sem_wait(&barrier->doorOut);
  sem_wait(&barrier->mutex);
  barrier->counter--;

  if(barrier->counter > 0) sem_post(&barrier->doorOut);
  else {
    for(int i = 0; i < barrier->maximumNumberOfGuests; i++) sem_post(&barrier->doorIn);
  }

  sem_post(&barrier->mutex);
}

void destroy(Barrier* barrier) {
  sem_destroy(&barrier->mutex);
  sem_destroy(&barrier->doorIn);
  sem_destroy(&barrier->doorOut);
  barrier->counter = 0;
  barrier->maximumNumberOfGuests = 0;
}

Barrier* myBarrier;
```