Przetwarzanie wielowątkowe - przetwarzanie współbieżne

Problemy współbieżności

- wyścig (*race condition*) – brak determinizmu wykonania
- zależności danych (*data dependence, data race*)
- synchronizacja
  - realizowana sprzętowo (np. SIMD, komputery macierzowe)
  - realizowana programowo (bariera, sekcja krytyczna, operacje atomowe)
- wzajemne wykluczanie
  - sekcja krytyczna
  - protokoły wejścia i wyjścia z sekcji krytycznej
- cechy rozwiązań zagadnień synchronizacji:
  - pożądane – bezpieczeństwo i żywotność, uczciwość
  - możliwe błędy - zakleszczenie, zagłodzenie

Problemy współbieżności – zależności

- Zależności – wzajemne uzależnienie instrukcji nakładające ograniczenia na kolejność ich realizacji
- Zależności zasobów
  - kiedy wiele instrukcji usiłuje korzystać ze wspólnego zasobu (np. pliku)
- Zależności sterowania
  - kiedy wykonanie danej instrukcji zależy od rezultatów poprzedzających instrukcji warunkowych
- Zależności danych (zależności przepływu)
  - kiedy instrukcje wykonywane w bezpośrednim sąsiedztwie czasowym operują na tych samych danych i choć jedna z tych instrukcji dokonuje zapisu

Wzajemne wykluczanie wątków

- Zamki (*locks*, sygnalizatory dostępu, blokady, flagi)

```
int zamek=0;
procedura_wątek(){
    while (zamek != 0) { } // aktywne czekanie (busy wait)
    zamek = 1;
    sekcja_krytyczna();
    zamek = 0;
    }
```

Wzajemne wykluczanie wątków

- Zamki (*locks*, sygnalizatory dostępu, blokady)

```
int zamek=0;
procedura_wątek(){
    while (zamek != 0) { } // aktywne czekanie (busy wait)
    zamek = 1;
    sekcja_krytyczna();
    zamek = 0;
    }
```

- Problemy:
  - aktywne czekanie zużywa zasoby komputera
  - procedura nie jest bezpieczna ani nie zapewnia żywotności

Wzajemne wykluczanie wątków

- Jak rozwiązać problem protokołów wejścia i wyjścia dla sekcji krytycznej:
  - bardziej rozbudowane algorytmy
    - wiele zmiennych, wiele zapisów, wiele odczytów
  - wsparcie sprzętowe
    - odpowiednie rozkazy procesora
  - wsparcie systemowe
    - procedury systemowe zaimplementowane przy użyciu wsparcia sprzętowego
  - odpowiednie API
    - implementacja wykorzystująca procedury systemowe

Model pamięci

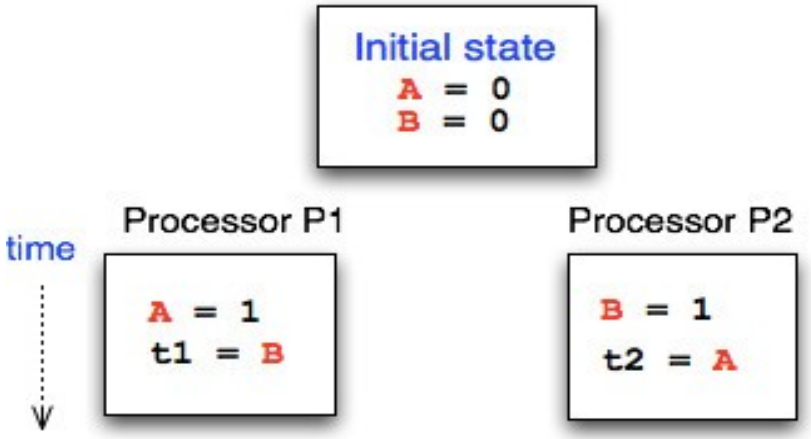
- Problem synchronizacji operacji na zmiennych dotyczy takich problemów jak np.:
  - kiedy kompilator może dokonywać operacji wyłącznie na rejestrach, a kiedy musi dokonać zapisu do pamięci
  - jak utrzymać spójność kopii danych w pamięci podręcznej z wartością w pamięci głównej
  - czy kompilator i procesor mogą zmieniać kolejność wykonywanych operacji (w tym operacji zapisu i odczytu)
- W obliczeniach sekwencyjnych zagadnienie synchronizacji operacji na kopiach wartości zmiennych jest problemem głównie wydajności
  - kompilatory i procesory są zobowiązane działać tak, aby efekt działania systemu komputerowego był zgodny z zapisem w kodzie źródłowym (czyli złudzeniem istnienia tylko jednej kopii danych)
    - nie dotyczy to działania *debugerów,* które śledzą nie ostateczne efekty działania kodu, ale stan systemu po każdej linii kodu

Model pamięci

- W obliczeniach równoległych zagadnienie synchronizacji operacji na kopiach wartości zmiennych wpływa na poprawność programów:
  - brak wymuszenia operacji zapisu może powodować, że różne wątki widzą różne wartości tej samej zmiennej wspólnej
  - rozmaite środowiska wprowadzają dodatkowe konstrukcje (*memory fences*, operacje *flush*) wymuszające zapis do pamięci
  - niejawne wymuszenie zapisu jest także związane z szeregiem operacji synchronizacji (bariery, protokoły sekcji krytycznych)
- W miejsce dotychczasowych umownych (C,C++) lub mało ścisłych specyfikacji (Java) wprowadza się w ostatnich latach w językach programowania ścisły model pamięci regulujący powyższe kwestie
- Środowiska programowania równoległego wprowadzają także własne modele spójności pamięci

Modele spójność pamięci

- Modele spójności pamięci
  - model spójności sekwencyjnej (*sequential consistency*)
  - modele spójności osłabionej (*relaxed consistency*)
- Paradoks pracy współczesnych procesorów
  - procesory ze względu na optymalizacje (kompilatora, samego procesora) nie spełniają prostych modeli spójności, np. spójności sekwencyjnej



Wzajemne wykluczanie wątków

- Specyfikacja POSIX: muteks – ***mut****ual* ***ex****clusion*
  - tworzenie muteksa

    int pthread_mutex_init(pthread_mutex_t *mutex,
    const pthread_mutex attr_t *mutexattr)
  - zamykanie muteksa (zwraca 0 w przypadku sukcesu)

    int pthread_mutex_lock(pthread_mutex_t *mutex)
  - próba zamknięcia muteksa (jeśli muteks jest wolny, wtedy działa jak *pthread_mutex_lock,* jeśli jest zajęty procedura wraca natychmiastowo, zwraca 0 jeśli muteks typu *reentrant* został zamknięty przez ten sam wątek, zwraca EBUSY w pozostałych przypadkach)

    int pthread_mutex_trylock(pthread_mutex_t *mutex)
  - otwieranie muteksa

    int pthread_mutex_unlock(pthread_mutex_t *mutex)
  - odpowiedzialność za poprawne użycie muteksów (gwarantujące bezpieczeństwo i żywotność) spoczywa na programiście

Busy waiting

- Podstawowy wzorzec:

```
while( pthread_mutex_trylock( &muteks ) != 0 ){ /*...busy waiting...*/ }
 ...          // inside critical section
pthread_mutex_unlock( &muteks );
```

- Przykładowe użycie z pracą poza sekcją krytyczną:

```
int work_done = 0; int work_inside = 0; int work_outside = 0;
while( work_done==0 ){
   if( work_inside == 0 && pthread_mutex_trylock( &muteks ) ==0 ){
       ...          // inside critical section
       work_inside=1; // only when all work done
       pthread_mutex_unlock( &muteks );
   } else  if( work_outside == 0 ) {
       ...          // do some work outside critical section
       work_outside=1; // only when all work done
   }
   if(work_inside>0 && work_outside>0) work_done=1;     } // end while
```

Stosowanie zamków

- Podstawowe typy muteksów:
  - jednorazowy (typ w POSIX: PTHREAD_MUTEX_NORMAL, PTHREAD_MUTEX_DEFAULT)
  - wielokrotnego wejścia (*reentrant*, PTHREAD_MUTEX_RECURSIVE)
    - muteks posiada zmienną zliczającą zamknięcia przez wątek
- Wskazówki praktyczne stosowania zamków (muteksów, semaforów, zmiennych warunku itp.):
  - jeśli to możliwe eliminować użycie zamków przez modyfikacje algorytmów, jeśli nie da się wyeliminować, minimalizować liczbę zamków i częstotliwość ich używania
  - zakładać zamki we właściwej kolejności w każdym wątku w celu uniknięcia zakleszczenia
  - dla każdego zamknięcia umieścić we właściwym miejscu otwarcie
    - uwzględnić różnice w przypadku stosowania wątków o wielokrotnym wejściu (*reentrant locks*)

Współbieżne struktury danych

- Alternatywą do stosowania dostępu do struktur danych wewnątrz sekcji krytycznej jest projektowanie współbieżnych struktur danych, czyli struktur, do których dostęp jest bezpieczny z poziomu wielu współbieżnie wykonywanych wątków
- Implementacja współbieżnych struktur danych może korzystać z dowolnych technik gwarantowania bezpieczeństwa i żywotności (np. programowej pamięci transakcyjnej lub innych złożonych algorytmów, ewentualnie technik omawianych na poprzednich slajdach)
- Obiektowe języki programowania i związane z nimi biblioteki zawierają wiele współbieżnych struktur danych, których stosowanie ma ułatwić programistom tworzenie poprawnych i wydajnych programów równoległych

Problemy współbieżności

- Problem producentów i konsumentów
- Problem czytelników i pisarzy
- Problem ucztujących filozofów (pięciu):
  - filozof: albo je, albo myśli
  - filozofowie siedzą przy stole, każdy ma talerz, pomiędzy każdymi dwoma talerzami leży widelec
  - na środku stołu stoi misa z spaghetti
  - problem polega na tym, że do jedzenia spaghetti potrzebne są dwa widelce (po obu stronach talerza)
  - jak zapewnić przeżycie filozofom?

Wzajemne wykluczanie wątków

- Semafory
  - konstrukcja teoretyczna umożliwiająca poprawne rozwiązanie problemu wzajemnego wykluczania
  - semafor jest zmienną globalną, na której można dokonywać dwóch niepodzielnych, wykluczających się operacji, zwyczajowo nazywanych: P (probeer, wait) i V (verhoog, signal) (obie często zaimplementowane w jądrze systemu operacyjnego, gdyż zawierają operacje atomowe)
  - P(int s){ if(s>0) s--; else uśpij_wątek(); }
  - V(int s) { if(ktoś_śpi()) obudź_wątek(); else s++; }
  - dodatkowo inicjacja semafora, np. init(int s, int v) { s=v; }
  - implementacja V decyduje o uczciwości semafora (np. FIFO)
  - wartość s oznacza liczbę dostępów do zasobu – np. semafor binarny

Wzajemne wykluczanie wątków

- Semafory – rozwiązanie problemu ucztujących filozofów
  - rozwiązanie proste dopuszczające blokadę (niepoprawne)

```
widelec[i], i=0..4        // pięć semaforów binarnych dla pięciu widelców
                          // (zainicjowanych wartością 1)
wątek_filozof(int i){     // procedura dla i-tego filozofa
                          // (pięć współbieżnie realizowanych wątków, i=0..4)
    for(;;){
        myśl();
        wait(widelec[i]);
        wait(widelec[(i+1) mod 5]);
        jedz();
        signal(widelec[i]);
        signal(widelec[(i+1) mod 5]);
} }
```

  - kiedy nastąpi blokada ?

Wzajemne wykluczanie wątków

- Semafory – rozwiązanie problemu ucztujących filozofów
  - rozwiązanie poprawne, nieco bardziej skomplikowane

```
widelec[i], i=0..4        // pięć semaforów binarnych dla pięciu widelców
                          // (zainicjowanych wartością 1)
pozwolenie                // semafor poczwórny (zainicjowany wartością 4)
wątek_filozof(int i){     // procedura dla i-tego filozofa
                          // (pięć współbieżnie realizowanych wątków, i=0..4)
    for(;;){
        myśl();
        wait(pozwolenie);
        wait(widelec[i]); wait(widelec[(i+1) mod 5]);
        jedz();
        signal(widelec[i]); signal(widelec[(i+1) mod 5]);
        signal(pozwolenie);
} }
```