
Programowanie równoległe i rozproszone

Monitory i zmienne warunku

Problemy współbieżności

→ Problem producentów i konsumentów:

- jedna grupa wątków produkuje (modyfikuje pewien zasób „powiększając” go), druga grupa konsumuje (modyfikuje zasób „pomniejszając” go) – jak zagwarantować sprawny (bez zakleszczeń i zagłódzeń) przebieg tej procedury
 - dostęp do zasobu może mieć tylko jeden wątek, ale dodatkowo istnieją warunki związane z zasobem, które mogą uniemożliwić wykonanie operacji - np. konsument nie może konsumować z pustego zasobu (kiedy jego rozmiar wynosi zero)

→ Problem czytelników i pisarzy

- jedna grupa procesów (pisarze) modyfikuje zasób, druga grupa (czytelnicy) tylko odczytuje stan zasobu
 - nie ma dodatkowych warunków (zawsze można czytać albo pisać - niezależnie od stanu zasobu), ale pisać może jednocześnie tylko jeden wątek, natomiast czytać może wiele wątków na raz

Zmienne warunku Pthreads

→ Zmienne warunku – *condition variables*:

- zmienne warunku są zmiennymi wspólnymi dla wątków, które służą do identyfikacji grupy uśpionych wątków
- tworzenie zmiennej warunku:

```
int pthread_cond_init( pthread_cond_t *cond, pthread_condattr_t *cond_attr)
```

- uśpienie wątku w miejscu identyfikowanym przez zmienną warunku `cond`, wątek śpi (czeka) do momentu, gdy jakiś inny wątek wyśle odpowiedni sygnał budzenia dla zmiennej `cond`

```
int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex)
```

- sygnalizowanie (budzenie pierwszego w kolejności wątku oczekującego “na zmiennej “ `*cond`)

```
int pthread_cond_signal( pthread_cond_t *cond)
```

- rozgłaszanie sygnału (budzenie wszystkich wątków oczekujących “na zmiennej “ `*cond`)

```
int pthread_cond_broadcast( pthread_cond_t *cond)
```

Wykorzystanie zmiennych warunku

- Schemat rozwiązania problemu producenta i konsumenta
- procedura główna

```
pthread_mutex_t muteks= PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nie_pelny, nie_pusty; // należy także zainicjować
int main(){
    pthread_t producent, konsument; zasob *fifo;
    fifo = inicjuj_zasob(); // zasob zawiera bufor do zapisu i odczytu
    // interfejs zasobu: inicjuj_zasob(), zasob_pelny(), zasob_pusty()
    // zasob_wstaw(...), zasob_pobierz()
    pthread_create( &producent, NULL, produkuj, fifo );
    pthread_create( &konsument, NULL, konsumuj, fifo );
    pthread_join( producent, NULL);
    pthread_join( konsument, NULL );
}
```

Wykorzystanie zmiennych warunku

→ procedura producenta

```
void *produkuj( void *q){
    zasob *fifo; int i;
    fifo = (zasob *)q;    // zasob zawiera bufor do zapisu i odczytu
    for(.....){
        pthread_mutex_lock (&muteks);
        while( zasob_pelny(fifo) ) pthread_cond_wait(&nie_pelny, &muteks );
        zasob_wstaw(fifo, ...);
        pthread_mutex_unlock( &muteks );
        pthread_cond_signal( &nie_pusty );
    } }
```

Wykorzystanie zmiennych warunku

→ procedura konsumenta

```
void *konsumuj( void *q){
    zasob *fifo; int i, d;
    fifo = (zasob *)q;    // zasob zawiera bufor do zapisu i odczytu
    for(.....){
        pthread_mutex_lock (&muteks);
        while( zasob_pusty(fifo) ) pthread_cond_wait(&nie_pusty, &muteks);
        zasob_pobierz(fifo, ...);
        pthread_mutex_unlock( &muteks );
        pthread_cond_signal( &nie_pelny );
    } }
```

Monitory

- W ujęciu klasycznym monitor jest modulem posiadającym:
 - stan – atrybuty (stałe i zmienne)
 - zachowanie – metody (procedury i funkcje))
- Metody monitora dzielą się na:
 - udostępniane na zewnątrz - publiczne
 - wewnętrzne - prywatne
- Podstawową zasadą działania monitora jest realizowanie sekcji krytycznej na poziomie obiektu:
 - jeżeli jakiś proces/wątek rozpoczął realizację dowolnej publicznej metody, żaden inny proces/wątek nie może rozpocząć realizacji tej samej lub innej publicznej metody monitora
 - inaczej: wewnątrz monitora może znajdować się tylko jeden proces/wątek

Monitory

- Realizacją wzajemnego wykluczania w dostępie do monitorów zarządza środowisko, w ramach którego funkcjonuje monitor
- Wątek, który wywołuje publiczną metodę monitora jest:
 - wpuszczany do monitora, jeśli w monitorze nie ma innych wątków
 - ustawiany w kolejce oczekujących wątków, jeśli w monitorze znajduje się wątek
- Po opuszczeniu monitora przez wątek system wznowia działanie pierwszego wątku w kolejce
- Istnieje jedna kolejka dla monitora (obsługuje próby wywołań wszystkich funkcji monitora)

Monitory

- Monitor jest skuteczną realizacją wzajemnego wykluczania przy dostępie do swoich wewnętrznych zasobów
 - inaczej: jeśli chcemy dostęp do jakiegoś zasobu uczynić wyłącznym dla pojedynczego wątku, należy umieścić ten zasób jako zasób wewnętrzny monitora
- Monitor umożliwia także synchronizację działania wątków opartą o mechanizm zmiennych warunku
 - zmienne warunku (typ **condition**) są wewnętrznymi zmiennymi monitora udostępniającymi na zewnątrz operacje:
 - **wait(c)** – protokół wejścia do monitora, jeśli wejście zamknięte – wątek jest usypiany „na zmiennej warunku **c**”
 - **empty(c)** - informacja czy na zmiennej **c** oczekują jakieś wątki
 - **signal(c)** – obudzenie jednego z wątków czekających na wejście do monitora lub otwarcie wejścia do monitora

Monitory - przykład

- Rozwiązanie problemu czytelników i pisarzy za pomocą monitorów
- Zakładamy, że system uruchamia dowolną liczbę procesów realizujących programy czytelników i pisarzy
- Pisarze wykonują operację **piszę** – jednak, żeby program był poprawny poprzedzają ją protokołem wejścia – **chcę_pisać** i kończą protokołem wyjścia **koniec_pisania**
- Podobnie czytelnicy wykonują sekwencję operacji: **chcę_czytać** – **czytam** – **koniec_czytania**
- Rozwiązanie polega na umieszczeniu wszystkich procedur zarządzających (**chcę_pisać**, **koniec_pisania**, **chcę_czytać**, **koniec_czytania**) w monitorze o nazwie Czytelnia

Monitory - przykład

```
monitor Czytelnia {  
    int liczba_czyt = 0; int liczba_pisz = 0; // 2. odpowiednia reprezentacja  
    condition czytelnicy, pisarze; // 1 i 3,4 - zmienne warunku, osobno dla  
                                     // uśpionych czytelników (3,4) i pisarzy (1)  
  
    chcę_pisać(){ // 1. prosty protokół wejścia  
        JEŻELI( liczba_czyt+liczba_pisz > 0 ) wait( pisarze );  
        liczba_pisz ++;  
    }  
  
    koniec_pisania(){ // 3. protokół wyjścia - uczciwy względem  
                       // czytelników, nie dopuszcza do zagłodzenia  
        liczba_pisz --;  
        JEŻELI( ~empty( czytelnicy ) ) signal( czytelnicy );  
        WPP signal( pisarze )  
    }  
}
```

Monitory - przykład

```
chcę_czytać(){ // 4. protokół wejścia:
                //    a. bezpieczny
                //    b. uczciwy względem pisarzy
    JEŻELI( liczba_pisz > 0 lub ~empty( pisarze ) ) wait( czytelnicy );
    liczba_czyt ++;
    signal( czytelnicy ); // 6. czytelnicy czekający na wejście budzą się
                          //    kolejno nawzajem – monitor sam musi
                          //    być uczciwy, żeby czytelnicy w kolejce do
                          //    wejścia do monitora nie zablokowali pisarzy
}
koniec_czytania(){ // 5. protokół wyjścia – budzi tylko pisarzy
                  //    (uczciwość), czytelnicy wpuszczeni do czytelnicy
                  //    nie wpuszczają kolejnych przy wychodzeniu

    liczba_czyt --;
    JEŻELI( liczba_czyt = 0 ) signal( pisarze );
}
}
```

Wątki w Javie

- Dla każdego obiektu Javy możemy wywołać funkcje typowe dla zmiennych warunku:
 - **wait()** - tym razem bez argumentów, jedyną identyfikacją miejsca oczekiwania jest fakt wystąpienia w danym obiekcie
 - **notify()** - obudzenie jednego z wątków oczekujących w danym obiekcie (tak jak **signal**)
 - **notifyAll()** - obudzenie wszystkich wątków oczekujących w danym obiekcie (tak jak **broadcast**)
- Powyższe funkcje zdefiniowane są w klasie **Object**, po której dziedziczy każda z klas Javy
- W pakiecie **java.util.concurrent** zdefiniowane są „standardowe” zmienne warunku (typ **condition**)

Java – obiekt jako monitor – przykład

```
public class Pojemnik {  
  
    private boolean pusty = true;  
  
    public synchronized /* dane */ wyjmij() {  
        while (pusty) {  
            try {  
                wait(); // oczekujemy – gdzie? w obiekcie !  
            } catch (InterruptedException e) { /* reakcja na przerwanie */ }  
        }  
        ... // pobierz_dane  
        pusty = true;  
        notifyAll(); // budzimy wszystkie wątki oczekujące w obiekcie  
    }  
}
```

Java – obiekt jako monitor – przykład

```
public synchronized void wloz( /* dane */ ) {  
    while (!pusty) {  
        try {  
            wait();// funkcja wait nie posiada identyfikatora miejsca  
                // oczekiwania, dlatego zalecane jest umieszczenie  
                // jej wewnątrz pętli while sprawdzającej warunek!  
        } catch (InterruptedException e) {}  
    }  
    ... // umieść_dane  
    pusty = false;  
    notifyAll(); // nie ma określonego miejsca oczekiwania - budzimy  
                // wszystkie wątki!  
}
```

Zamki odczytu/zapisu

- Alternatywą dla stosowania konstrukcji programistycznych może być wykorzystanie gotowych API zaprojektowanych do rozwiązywania konkretnych problemów wykonania współbieżnego:
 - przykład: zamki odczytu/zapisu (*read/write locks*) do rozwiązania problemu czytelników i pisarzy
- Zamki odczytu i zapisy są konstrukcją uwzględnioną w nowszym standardzie POSIX (2001), który wprowadza m.in. funkcje:
 - `pthread_rwlock_init` (tworzenie zamków na podstawie odpowiednich obiektów z atrybutami), `pthread_rwlock_destroy`
 - `pthread_rwlock_rdlock` – zamknięcie do odczytu (istnieje też wersja `pthread_rwlock_tryrdlock`)
 - `pthread_rwlock_wrlock` – zamknięcie do zapisu (wersja: `pthread_rwlock_trywrlock`)
 - `pthread_rwlock_unlock` – otwarcie zamka