|  |
| --- |
| Programowanie współbiezne |
| Laboratorium 2: Monitory |
| Żądanie dostępu do zasobu |
|  |
| **Tomasz Trębski – 165535**  **Karol Górecki - 165405** |
| **10/28/2013** |

|  |
| --- |
| **Sprawozdanie na laboratoria z przedmiotu Programowanie współbieżne** |

Table of Contents

[Wprowadzenia 2](#_Toc370667191)

[Teoria 2](#_Toc370667192)

[Współbieżność 2](#_Toc370667193)

[Semafory 2](#_Toc370667194)

[Praktyka 3](#_Toc370667195)

[Wnioski 4](#_Toc370667196)

# Wprowadzenia

Celem zadania było zaimplementowanie współbieżnego programu, który będzie posiadał trzy rodzaje procesów żądających dostępu do zasobów 2. typów (A i B):

1. Procesy żądające jednego zasobu typu A (wątki A)
2. Procesy żądające jednego zasobu typu B (wątki B)
3. Procesy żądające jednego zasobu typu A i typu B (wątki AB)

Korzystanie z zasobu wymaga wyłącznego dostępu do niego. Zaplanować strategię przydziału tak**,** aby nie zagłodzić żadnej z grup

# Teoria

## Współbieżność

Współbieżność odnosi się do własności systemów operacyjnych, gdzie możliwe jest równoległe wykonywanie obliczeń i wzajemnych interakcji z miedzy nami. W przypadku procesorów wielordzeniowych możliwe jest uzyskania prawdziwej współbieżności, kiedy to kolejne procesy/wątki w ramach tych procesów, wykonany są na innych rdzeniach procesora. W przypadku procesorów starszej generacji nie było to możliwe, ale takie programy dało się pisać. W tym przypadku współbieżność była emulowana, a czas procesora był dzielony między procesy/wątki.

## Semafory

Monitor jest specjalnym znacznikiem, którego właścicielem w danej chwili dla pojedynczego obiektu może być tylko jeden wątek. Wejście w posiadanie znacznika odbywa się poprzez uruchomienie metody synchronizowanej (lub bloku synchronizowanego). Wątek, który zostanie wywłaszczony przez scheduler nie oddaje znacznika. Z kolei wątek, który potrzebuje monitora do obiektu, po wykonaniu metody synchronizowanej przechodzi do stanu Oczekiwania na monitor. W momencie, gdy monitor stanie się dostępny, scheduler wybierze jeden wątek z puli wątków oczekujących na monitor, przekaże mu monitor i przeniesie do stanu runnable, gdzie zostanie obsłużony tak samo jak reszta wątków.

Cechy monitorów:

* reentrant locking – umożliwia wątkom nakładanie blokady, której są właścicielami bez oczekiwania na jej zwolnienie
* condition locking – umożliwia określenie maksymalnego czasu oczekiwania na zwolnienie blokady (metody: wait() / notify() / notifyAll())
* lock ownership – umożliwia stosowanie blokad współdzielonych i wykluczających – tylko wątek, który pobrał monitor może go zwrócić
* Ważnym ograniczeniem jest także konieczność zwolnienia monitora w tej samej ramce stosu, w której monitor został pobrany, co ogranicza zasięg bloku synchronizowanego tylko do jednej metody

# Praktyka

Część praktyczna została zaimplementowana w języku Java. Za bazę posłużyło abstrakcyjna klasa **CPMonitor**, będąca implementacją interfejsu Runnable[[1]](#footnote-1), w której znajdował się kod pozwalający na uzyskania pozwolenia na dostęp do „puli zasobów”. Zadaniem klasy **ResourcePool** było przechowywanie instancji zasobów oraz nie dopuszczenie do pobrania **tej samej** „sztuki” danego zasobu przez dwa równe wątki.

Algorytm działania wątków A:

1. Sprawdź czy numer rundy równa się maksymalnej ilości rund - jeżeli nie zacznij pętle
2. Pobierz zasób A
3. Wypisz informacje, że pobrałeś zasób A o określonym numerze
4. Oddaj zasób do zbioru
5. Dodaj 1 do numeru rundy
6. Idź spać na określoną liczbę ms
7. Zacznij pętle od nowa

Algorytm działania wątków B:

1. Sprawdź czy numer rundy równa się maksymalnej ilości rund - jeżeli nie zacznij pętle
2. Pobierz zasób B
3. Wypisz informacje, że pobrałeś zasób A o określonym numerze
4. Oddaj zasób do zbioru
5. Dodaj 1 do numeru rundy
6. Idź spać na określoną liczbę ms
7. Zacznij pętle od nowa

Algorytm działania wątków AB:

1. Sprawdź czy numer rundy równa się maksymalnej ilości rund - jeżeli nie zacznij pętle
2. Pobierz zasób A
3. Pobierz zasób B
4. Wypisz informacje, że pobrałeś zasób A o określonym numerze
5. Oddaj zasób A do zbioru
6. Oddaj zasób B do zbioru
7. Dodaj 1 do numeru rundy
8. Idź spać na określoną liczbę ms
9. Zacznij pętle od nowa

Algorytm metody **acquireResource()**:

1. Nałóż blokadę na zbiór
2. Sprawdź czy są dostępne zasoby, jeżeli nie czekaj na sygnał od metody **releaseResource(R resource)**
3. Sprawdź, który zasób się niezablokowany
4. Ustaw flagę **locked** na **true** znalezionego zasobu
5. Zwolnij blokadę ze zbioru
6. Zwróć zasób

Algorytm działania metody **releaseResource(R resource):**

1. Nałóż blokadę na zbiór
2. Ustaw flagę **locked** na **false** znalezionego zasobu
3. Zwolnij blokadę ze zbioru
4. Powiadom **acquireResource()** o zwolnieniu zasobu

# Wnioski

Najtrudniejsze w zadaniu okazało się zaprojektowanie sekcji krytycznych metod do pobierania i oddawania zasobów, aby nie zagłodzić żadnego procesu pobierającego zasoby ze zbiorów. Zadanie to zostało znacznie uproszczone dzięki świetnie napisanej dokumentacji klasy ReentrantLock[[2]](#footnote-2) dla języka Java.

1. <http://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html> [↑](#footnote-ref-1)
2. <http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html> [↑](#footnote-ref-2)