Programowanie współbieżne
Zadanie nr 2
Semafory

Cel zadania.

Celem zadania jest poznanie oraz zrozumienie funkcjonowania podstawowego mechanizmu wykorzystywanego w programowaniu współbieżnym jakim są semafory.

Treść zadania.

Należy zorganizować współpracę procesów na podstawie następujących założeń:
- procesy powinny mieć ograniczony dostęp do danych z których wzajemnie korzystają, czyli zapewnienie warunku tzw. wzajemnego wykluczania się procesów.
- procesy powinny być zsynchronizowane za pomocą semaforów.

Podstawa zaliczenia.

1) Zademonstrowanie działającego programu.
2) Przedstawienie sprawozdania zawierającego:
   - krótki opis zastosowanego rozwiązania problemu.
   - udowodnienie, że zastosowane rozwiązanie jest poprawne.

Pytania kontrolne.

1) Co to jest operacja atomowa?
2) Jaka jest różnica pomiędzy aktywnym i pasywnym oczekiwaniem?
3) Co to jest semafor?
4) Na czym polega wzajemne wykluczanie procesów?
5) Co specyfikują nam własności żywotności i bezpieczeństwa?
6) Która z własności uczciwości jest najmocniejsza?
7) Na czym polega zjawisko blokady?

Część teoretyczna.

I. Informacje ogólne.

Zwyczajny program składa się ze zbioru danych oraz działających na nich instrukcji zapisanych w dowolnym języku programowania. Instrukcje te są wykonywane sekwencyjnie przez komputer, który przechowuje dane w swojej pamięci operacyjnej. *Program współbieżny* jest zbiorem zwykłych programów sekwencyjnych wykonywanych *abstrakcyjnie* równolegle. Te sekwencyjne programy będziemy nazywać procesami, a nazwę program zarezerwujemy dla zbioru tych procesów. Wspomniana współbieżność jest abstrakcyjna ze względu na to, że w większości naszych komputerów mamy do dyspozycji tylko jeden procesor, zatem procesy będą wykonywane sekwencyjnie dzieląc moc procesora pomiędzy siebie. Jednakże dla uproszczenia rozważań nad rozwiązaniem problemów programowania współbieżnego, czasem wygodniej jest założyć, że każdy z procesów ma przydzielony własny procesor. Wówczas możliwe wzajemne oddziaływanie pomiędzy procesami będziemy musieli rozważać tylko w dwóch przypadkach:

Współzawodnictwo. Dwa procesy ubiegają się o ten sam zasób: zasób obliczeniowy, dostęp do komórki pamięci
.
Komunikacja. Dwa procesy mogą chcieć się porozumieć, by przesłać dane od jednego do drugiego. Fakt porozumiewania się jest bardzo ważny, gdyż umożliwia synchronizację procesów.

Skoro programy działające współbieżnie wykonywane są sekwencyjnie przez jeden procesor, to możemy spodziewać się dowolnego *przeplotu* instrukcji należących do tych procesów. Elementarną instrukcję,

która nie może być podzielona, nazywać będziemy *instrukcją atomową*. To od tych instrukcji oraz kolejności ich wykonań, zależeć będzie poprawność działania naszego programu. Dopuszczając dowolne przeploty atomowych instrukcji należących do procesów wykonywanych współbieżnie, będziemy mówili o ich poprawnym wykonaniu tylko wtedy, gdy wyniki działań tych procesów z przeplotem będą takie same, jak wyniki uzyskane w momencie działania bez przeplotu.

Są dwa typy własności dotyczących poprawności:

Własność bezpieczeństwa, która musi być *zawsze* prawdziwa;

Własność żywotności, która w *końcu* musi być prawdziwa.

Najbardziej typową własnością bezpieczeństwa jest *wzajemne wykluczanie*. Dwa procesy nie mogą przeplatać pewnych ciągów instrukcji. Innymi słowy, jeden ciąg musi się zakończyć zanim zacznie się drugi. Kolejność wykonywania tych dwóch ciągów, nie ma znaczenia. W ten sposób możemy modelować wiele problemów występujących w systemach operacyjnych, jak na przykład dostęp do zasobów, takich jak drukarki, czy dostęp do usług ogólnodostępnych w systemie. „Zawsze" musi być prawdą, co oznacza, że w danym momencie tylko jeden proces może korzystać z chronionego zasobu. Niedopuszczalna jest sytuacja, w której na przykład dwóm procesom przydziela się drukarkę.

Inną ważną własnością bezpieczeństwa jest brak *blokady*. System, który nie reaguje na żadne komunikaty i żądania, jest zablokowany i znów coś takiego nie może się zdarzyć. Nawet jeśli system wygląda na uśpiony, jeśli nie ma nic do roboty, to wykonuje w tle jakiś jałowy proces i może natychmiast obsłużyć przerwanie.

Jedną z własności żywotności jest tak zwany *brak zagłodzenia*. Oznacza on, że jeżeli jakiś proces zażąda dostępu do pewnego zasobu systemu, np. drukarki, to w końcu zostanie mu ten zasób przydzielony. Stwierdzenie w końcu jest zbyt słabą specyfikacją, gdyż przydzielenie drukarki np. po stu latach z punktu widzenia użytkownika systemu wcale nie oznacza braku zagłodzenia. Jeśli system nie jest złośliwy, to „w końcu" nastąpi dość szybko, pod warunkiem że nie ma współzawodnictwa o drukarkę.

Szczególnym rodzajem własności żywotności jest tak zwana *własność uczciwości*. Jeśli wystąpi współzawodnictwo, to często będziemy chcieli wiedzieć jak zostanie ono rozwiązane. Oto cztery możliwe specyfikacje uczciwości:

Uczciwość słaba. Jeśli proces nieprzerwanie zgłasza żądanie, to kiedyś będzie ono obsłużone.

Uczciwość mocna. Jeśli proces zgłosi żądanie nieskończenie wiele razy, to kiedyś będzie ono obsłużone.

Oczekiwanie liniowe. Jeśli proces zgłasza żądanie, to będzie ono obsłużone zanim dowolny inny proces zostanie obsłużony więcej niż raz.

FIFO (pierwszy wszedł, pierwszy wyszedł)**.** Jeśli proces zgłasza żądanie, to będzie ono obsłużone przed dowolnym żądaniem zgłoszonym później.

FIFO jest najsilniejszą specyfikacją uczciwości, w dodatku łatwą do zaimplementowania w scentralizowanym systemie.

Oczywiście pisząc nasze własne programy z wykorzystaniem już wbudowanych mechanizmów takich jak semafory, nie musimy martwić się o strategię wznawiania procesów wstrzymanych na semaforze, gdyż jest to najczęściej właśnie kolejka FIFO. Problem zagłodzenia może pojawić się w bardziej skomplikowanych sytuacjach, kiedy to my bierzemy na siebie odpowiedzialność za przydzielanie zasobów, na przykład podczas rozwiązywania problemu pięciu filozofów. Znacznie częściej podczas realizacji zadania pierwszego będziemy mieć do czynienia z koniecznością zapewnienia wzajemnego wykluczania oraz synchronizacji współdziałających procesów.

II. Semafory.

Semafory są narzędziem synchronizacyjnym wprowadzonym przez Dijsktrę. *Semaphore* - jest to zmienna całkowita s, przyjmująca wartości nieujemne, na której możliwe jest wykonywanie dwóch typów niepodzielnych (atomowych) operacji - *wait(s)* i *signal(s).* Operacje te w klasycznej definicji semafora wyglądają następująco:

```
wait(s):
        while s<=0 do {nothing}
        s:=s-1;
signal(s):
        s:=s+1;
```

Są to operacje pierwotne, wykluczające się nawzajem. W danej chwili tylko jeden z procesów może zmieniać wartość semafora; dla wait() - niedopuszczalne jest przerwanie w trakcie sprawdzania wartości semafora i podstawiania s-1.
Poza operacjami wait() i signal() semaforowi można nadać wartość początkową. Jeśli semafor przyjmuje wartości wyłącznie {0,1} nazywany jest semaforem binarnym, jeśli natomiast dowolne wartości nieujemne - nosi nazwę semafora ogólnego.

Rozwiązanie problemu wzajemnego wykluczania za pomocą semafora binarnego, zdefiniowanego jak wyżej i zainicjalizowanego wartością 1 okazuje się bardzo proste i de facto sprowadza się do następującego po sobie wywołania obu instrukcji:

```
repeat
        wait(s);
        ..... SEKCJA KRYTYCZNA
        signal(s);
        ..... RESZTA
until false
```

W powyższym "programie" (i w oparciu o klasyczną definicję semafora) występuje nadal aktywne czekanie - proces wykonywać będzie w pętli instrukcje swojej sekcji wejściowej, dopóki inny proces nie zakończy sekcji krytycznej. Tego rodzaju semafor nosi nazwę wirującej blokady i w systemie jednoprocesorowym jest to dość nieekonomiczne podejście do czasu pracy procesora.
Aby uniknąć tego ro3dzaju niedogodności, należy zawiesić wykonanie procesu i ustawić go w kolejce. Należy więc przyjąć mocniejszą definicję semafora i trochę zmodyfikować instrukcję *wait()*

```
wait(s):
        if s=0 then ... {zawiesić proces i ustawić go w kolejce}
        s:=s-1;
```

Procesy które "wiszą na semaforze" nie są aktywne, a więc procesor nie wykonuje "bezużytecznej" sekwencji rozkazów. Proces zablokowany pod semaforem zostaje wznowiony poprzez wykonanie operacji *signal* przez inny proces. Reaktywacja procesu oznacza, że proces przechodzi z kolejki procesów oczekujących do kolejki procesów gotowych. Procesy w systemie Windows oczywiście nie są aktywne podczas oczekiwania na podniesienie semafora.

Z semaforem z kolejką procesów oczekujących wiąże się niebezpieczeństwo blokady *(deadlock)* Może ona powstać w następujący sposób - kilka procesów będzie czekać w nieskończoność na zajście zdarzenia, do którego może dojść wyłącznie na skutek działania jednego z tych procesów. Procesy, należące do tego samego zbioru blokują się wzajemnie.

Złośliwy scenariusz:
Dwa procesy - P1, P2 dzielą dwa semafory s1 i s2 zainicjalizowane wartością 1;

krok1:
P_1: wait(s1) P_2: wait (s2)

krok2:
P_1: wait(s2) P_2: wait(s1)

Teraz P_1 będzie czekał, aż P_2 wykona *signal(s2)*, a P_2 - aż P_1 wykona *signal(s1)* - wartości obu semaforów są równe zero i oba procesy wiszą - nawzajem skutecznie się blokując.

Literatura.

M.Ben-Ari „Podstawy programowania współbieżnego i rozproszonego” WNT 1996

Spis wariantów zadań.

1. Wyznaczyć różnicę symetryczną dwóch zbiorów liczbowych X i Y. Zbiory zadane i wynikowy będą reprezentowane w postaci ciągów liczbowych.

2. Obliczyć współczynniki rozwinięcia dwumianu Newtona $(a+b)^n$ korzystając z trójkąta Pascala.

3. Posortować ciąg liczb wykorzystując algorytm sortowania przez zamianę prostą oraz algorytm scalania dwóch posortowanych ciągów. Zadany ciąg podzielono na dwie części. Każda z nich jest sortowana oddzielnie, następnie posortowane części są scalane. Sortowanie obu połówek oraz procedura scalająca wykonywane mają być współbieżnie.

4. Dane są dwa rozłączne zbiory S i T. Niech s i t będą liczbą elementów odpowiednio w zbiorach S i T. Napisać program, który dokonuje podziału elementów ze zbioru S∪T, tak aby w zbiorze S znalazło się s najmniejszych elementów, a w zbiorze T – t największych elementów. W programie powinny współpracować dwa procesy : jeden zarządzający zbiorem S, a drugi zbiorem T. Procesy powinny tak długo wymieniać między sobą liczby, aż osiągną żądany rezultat.

5. Dane są dwa ciągi $\{x_i\}$, $\{y_i\}$, i=1, 2, ..., n o wartościach całkowitych dodatnich. Wymień pierwszy element parzysty z ciągu x z pierwszym elementem nieparzystym z ciągu y, drugi element parzysty z ciągu x z drugim elementem nieparzystym ciągu, itd., aż do wyczerpania jednego z podzbiorów liczb w ciągach.

6. Napisać program współbieżny rozwiązujący poniższy układ równań liniowych

$$
\begin{aligned}
a_{11}x_1 &= b_1 \\
a_{21}x_1 + a_{22}x_2 &= b_2 \\
&\cdots\cdots \\
a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n &= b_n
\end{aligned}
$$

7. Napisać program wymnażający dwie macierze kwadratowe o wymiarze n na n.

   Przykład dla n = 2:

$$
\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}
$$

8. Napisać program znajdujący liczby pierwsze używając tzw. sita Erastotenesa.

   Przykład:

   Mamy ciąg liczb naturalnych:

   1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, ...

   Eliminujemy wielokrotności liczby 2 przez co otrzymujemy ciąg:

   1, 2, 3, *, 5, *, 7, *, 9, *, 11, *, 13, ...

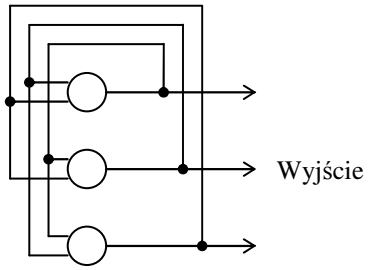
   W następnym kroku usuwamy wielokrotności liczby 3:

   1, 2, 3, *, 5, *, 7, *, *, 11, *, 13, ...

   Później wielokrotności 5, 7 itd..

9. Napisać program znajdujący wartość średnią, tzw. medianę, macierzy kwadratowej o wymiarze n na n, gdzie n jest liczbą nieparzystą.

10. Zamodelować współbieżnie prostą sieć rekurencyjną złożoną z trzech neuronów z dowolnymi funkcjami aktywacji. Przy czym każdy neuron obsługuje jeden proces. Rekurencje należy przerwać po określonej liczbie kroków.



11. Zaimplementować program generujący figurę fraktalną z dokładnością do określonej iteracji według pewnego wzorca, przy użyciu dwóch procesów.
    Niech wzorcem będzie na przykład krok o pewną wartość w przód, a następnie w prawo o tę samą wartość. Generację figury zaczynamy od pewnego punktu początkowego. Dla tego punktu stosując nasz wzorzec otrzymujemy dwa kolejne punkty: pierwszy po wykonaniu kroku w przód, drugi po wykonaniu skrętu w prawo. Punkty te odkładamy na „stos" jednocześnie zdejmując z niego punkt początkowy. Zmniejszamy wartość kroku o określoną wartość i wykonujemy kolejne iteracje, tym razem już dla dwóch punktów początkowych.

12. Napisać program współbieżny wyliczający histogram dla obrazu o wymiarze n na m, przy użyciu k wątków. Dla uproszczenia niech obraz będzie dwuwymiarową tablicą zmiennych typu char. k + 1–wszy wątek powinien zając się wyświetlaniem gotowego już histogramu.

13. Napisać program realizujący obliczenia wyznacznika macierzy o wymiarze 3 na 3, za pomocą trzech procesów , przy czym trzeci proces może wykonywać jedynie operację dodawania.