Systemy komputerowe

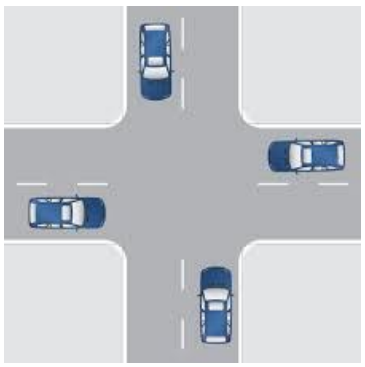
Lista zadań C

Karolina Szlęk 300411

Zadanie 1

Zadanie 1. Zdefiniuj zjawisko **zakleszczenia** (ang. *deadlock*), **uwięzienia** (ang. *livelock*) i **głodzenia** (ang. *starvation*). Rozważmy ruch uliczny – kiedy na skrzyżowaniach może powstać każde z tych zjawisk? Zaproponuj metodę (a) **wykrywania i usuwania** zakleszczeń (b) **zapobiegania** zakleszczeniom. Pokaż, że nieudana próba zapobiegania zakleszczeniom może zakończyć się wystąpieniem zjawiska uwięzienia lub głodzenia.

Zakleszczenie (ang. deadlock) – sytuacja, w której przynajmniej dwie różne akcje czekają na siebie nawzajem. A co za tym idzie, żadna nie może się zakończyć.



Sytuacja, gdy na skrzyżowanie wjechały 4 auta, przy czym, każde nadjechało z innej strony i chce przejechać przez skrzyżowanie na wprost. Wszystkie staną i nie wiedzą co zrobić więc czekają na znak od któregoś z innych kierowców (np. Machnięcie ręką, ze cię przepuszcza).

A.) Wykrywanie i likwidowanie zakleszczeń

Żeby wykrywać możemy stworzyć graf zasobów i sprawdzić, czy jest w nim cykl (każdy typ zasobów może mieć jedną instację)
W przypadku, gdy zasoby mogą mieć kilka instacji tworzymy następujące macierze

Alokowane- macierz rozmiaru n na m. Dla danego typu zasobu oznacza ile każdy proces ma zaalokowane instacji.
Żądania - macierz tego samego rozmiaru co macierz Alokowane, oznaczająca, ile danego typu zasobu żąda każdy proces.
Dostępne - będzie wektorem długości m. Wektor ten będzie zawierał informacje o tym, ile jednostek danego typu zasobu jest dostępnych.

Korzystamy z następującej metody/algorytmu:

Wektory Activ i Ended mają odpowiednio długości m i n.
Activ ma być równy Dostępne (Activ = Dostępne)

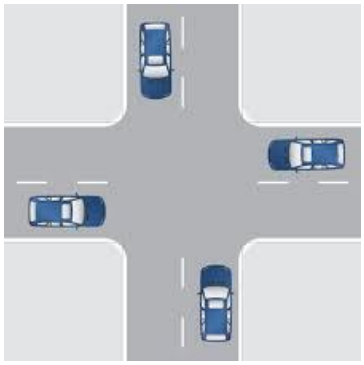
Jeśli i-ty proces nie ma żądań, to Ended[i] = True
Szukamy takiego procesu i, dla którego Żądanie[i] to mniej zasobów niż mamy dostępne. O dostępnych procesach wiemy z Activ.
Ustawiamy Ended[i] na True
Do Active dodajemy Alokowane[i]
Kończymy algorytm, jeżeli nie ma takiego i
Mamy zakleszczenie – po zakończeniu algorytmu, dla danego i, Ended[i] == False

Usunąć zakleszczenie można poprzez zabicie jednego z procesów, który blokuje niezbędne nam zasoby, przez odebranie zakleszczonemu procesowi zasobów na jakiś określany czas bądź też poprzez cofnięcie (do jakiegoś zapisanego stanu) procesu, który blokuje nasze zasoby.

B.) Zapobieganie zakleszczeniom

Zapobieganie zakleszczenia jest niezwykle istotne. Kluczowy i najważniejszy w zapobieganiu zakleszczeniom jest etap projektowania aplikacji oraz systemu. Przykładowe sposoby:
Ograniczenie ilość blokad na zasobach do niezbędnego minimum
Proces powinien blokować wszystkie potrzebne mu zasoby przed startem.
Proces może blokować zasoby tylko, gdy nie blokuje aktualnie innych zasobów. Może blokowa jeden zasób na raz.
Proces może zablokować proces tylko o numerze większym niż największy numer zasobu, które aktualnie blokuje (numerowanie zasobów).

Uwięzienie (ang. Livelock) - stan, który ma miejsce, gdy dwa lub więcej programów stale zmienia swój stan, przy czym żaden program nie robi postępów. Procesy wchodzą w stan blokady aktywnej, gdy zderzają się ze sobą i nie postępują, ponieważ oba zmieniają stan, a zatem mają ten sam stan w danym momencie. Sytuacja, gdy na skrzyżowanie wjechały 4 auta, przy czym, każde nadjechało z innej strony i chce przejechać przez skrzyżowanie na wprost (sytuacja z obrazka). W takiej sytuacji wszystkie auta spotkają się na środku skrzyżowania zajeżdżając sobie wzajemnie drogę i żadne z nich nie będzie mogło się ruszyć. Wszystkie zaczną cofać jednocześnie, a potem znowu spróbują przejechać i tak w kółko.



Głodzenie, Głód zasobów (ang. Starvation) jest problemem napotykanym w obliczeniach równoległych, w których procesowi stale odmawia się niezbędnych zasobów do przetworzenia jego pracy. Głód może być spowodowany błędami w algorytmie szeregowania lub algorytmu wzajemnego wykluczania, ale może być również spowodowany wyciekiem zasobów i może być celowo spowodowany atakiem typu „odmowa usługi”, takim jak bomba widełkowa.
Sytuacja: ruch na skrzyżowaniu kierowany przez policjanta. 4 auta nadjeżdżają z 4 różnych stron, policjant po kolei pozwala przejeżdżać autom, które nadjeżdżają z 3 stron, a jedno auto, które nadjechało z czwartej ciągle stoi i czeka, policjant nie pozwala mu przejechać

Nieudana próba zapobiegania zakleszczeniom może zakończyć się wystąpieniem zjawiska uwięzienia lub głodzenia.

Dla niektórych algorytmów (sposobów) zapobiegania zakleszczeniom, jakiś proces może nie otrzymać zasobów, których potrzebuje. I wystąpi nam zjawisko jego **głodzenia.** Proces zwalnia zasoby, gdy nie może zebrać wszystkich zasobów, które są mu potrzebne, to może prowadzić do **uwięzienia.**

Zadanie 2

Zadanie 2. W poniższym programie występuje **sytuacja wyścigu** (ang. *race condition*) dotycząca dostępów do współdzielonej zmiennej «tally». Wyznacz jej najmniejszą i największą możliwą wartość.

```
1    const int n = 50;
2    shared int tally = 0;
3
4    void total() {
5      for (int count = 1; count <= n; count++)
6        tally = tally + 1;
7    }
8
9    void main() { parbegin (total(), total()); }
```

Dyrektywa «parbegin» rozpoczyna współbieżne wykonanie procesów. Maszyna wykonuje instrukcje arytmetyczne wyłącznie na rejestrach – tj. kompilator musi załadować wartość zmiennej «tally» do rejestru, przed wykonaniem dodawania. Jak zmieni się przedział możliwych wartości zmiennej «tally», gdy wystartujemy k procesów zamiast dwóch? Odpowiedź uzasadnij.

Cała operacja wygląda następująco:

wczytanie tally do rejestru (read)
inkrementacja rejestru (inc)
zapisanie rejestru do tally (write)

Na początku wydaje się, że „suma" mieści się w przedziale 50 ≤ suma ≤ 100.
Wydaje się, że uruchamiając dwa procesy jednocześnie, nie powinniśmy być w stanie uzyskać wyniku niższego niż wynik uzyskany przez wykonanie tylko jednego z tych procesów sekwencyjnie.
Ale co z następującą sytuacją?

Proces 1 ładuje wartość sumy, zwiększa sumę. Udało mu się zwiększyć rejestr o 1, ale nie udało mu się jej zapisać, ponieważ stracił procesor.
Proces 2 ładuje wartość sumy i wykonuje czterdzieści dziewięć pełnych operacji inkrementacji, zapisuje wartości 49 w tally, i traci procesor.
Proces 1 odzyskuje kontrolę i udaje mu się wykonać pierwszą operację zapisu. Wartość sumy, która wynosiła 49 jest zastąpiona teraz przez 1. Nstępnie natychmiast zmuszony do rezygnacji z procesora
Proces 2 wznawia się i ładuje obecną wartość tally (czyli 1) do swojego rejestru. Jak to zrobi, to też jest zmuszony zrezygnować z procesora.
Proces 1 tym razem przebiega do końca, wykonując pozostałe 49 operacji, co powoduje ustawienie wartości liczenia na 50 (1+49).
Proces 2 zostaje ponownie aktywowany mając tylko jedną inkrementacją i operacje zapisu do

wykonania przed zakończeniem. Podnosi swoją wartość rejestru do 2 i przechowuje tę wartość jako wartość końcową zmiennej dzielonej.

Linia „tally = tally +1” zostanie wykonana co najwyżej 100 razy . Tę wartość można osiągnąć, jeśli pierwszy proces uruchomi się do końca, zanim drugi proces uzyska szansę na uruchomienie.

Proces 2 robi wszystko poza jedną instrukcją, Proces 1 zapisuje swój wynik pierwszego incrementa, potem Proces2 wczytuje ten wynik - musi zrobić jeszcze jedną operację. Proces 1 robi wszystko, a na końcu Proces2 wpisuje 2 do wyniku. I dlatego nawet jak zaczynamy od zera, to minimum to 2. Zatem właściwy zakres wartości końcowych wynosi **2 ≤tally ≤ 100**
Widzimy, że zakres zaczyna się na 2 a kończy na 2*n (nasze n było równe 50).

Jak zmieni się przedział możliwych wartości zmiennej «tally», gdy wystartujemy k procesów zamiast dwóch?
W takim wypadku nasze wartości min i max wyniosą odpowiednio 2 i n*k (n-ilość iteracji z zadania tj. 50)

Zadanie 3

Zadanie 3. Przeanalizuj poniższy pseudokod wadliwego rozwiązania problemu producent-konsument. Zakładamy, że kolejka «queue» przechowuje do n elementów. Wszystkie operacje na kolejce są **atomowe** (ang. *atomic*). Startujemy po jednym wątku wykonującym kod procedury «producer» i «consumer». Procedura «sleep» usypia wołający wątek, a «wakeup» budzi wątek wykonujący daną procedurę. Wskaż przeplot instrukcji, który doprowadzi do (a) błędu wykonania w linii 6 i 13 (b) zakleszczenia w liniach 5 i 12.

```
1   def producer():
2     while True:
3       item = produce()
4       if queue.full():
5         sleep()
6       queue.push(item)
7       if not queue.empty():
8         wakeup(consumer)
```

```
9   def consumer():
10    while True:
11      if queue.empty():
12        sleep()
13      item = queue.pop()
14      if not queue.full():
15        wakeup(producer)
16      consume(item)
```

Wskazówka: Jedna z usterek na którą się natkniesz jest znana jako problem zagubionej pobudki (ang. *lost wake-up problem*).

Operacje atomowe to takie, gdzie obserwator nie może zobaczyć wyników pośrednich.
Operacja atomowa, na określonym poziomie abstrakcji jest niepodzielna

Kolejka <<queue>> przechowuje n elementów
<<sleep>> usypia wątek
<<wakeup>> budzi wątek wykonujący procedurę

A.) Doprowadzi do błędu wykonania w linii 6 i 13

Błędem w linii 6 - kolejka jest już pełna, a mamy push:

konsument sprawdza, widzi, że kolejka jest już prawie pełna, ale jeszcze NIE jest pełna
producent dokłada kolejny element do kolejki (kolejka będzie pełna)
producent idzie spać <<sleep>>
konsument budzi producenta <<wakeup>>
producent próbuje dodać nowy element do kolejki (a kolejka jest już pełna)

Błąd w linii 13 - kolejka jest pusta, a mamy pop:

konsument sprawdza, widzi, że coś jest w kolejce, kolejka NIE jest pusta
konsument robi pop na ostatnim elemencie z kolejki (kolejka będzie pusta)
konsument idzie spać <<sleep>>
producent budzi konsumenta <<wakeup>>
konsument wykonuje pop na pustej kolejce

B.) Doprowadzi do zakleszczenie w liniach 5 i 12

konsument sprawdza, widzi, że nie ma nic w kolejce
producent produkuje n elementów (przy każdym budzi konsumenta, konsument wtedy nie śpi)
konsument idzie spać <<sleep>>
producent idzie spać <<sleep>>

Zadanie 4

Zadanie 4. Poniżej znajduje się propozycja[6] programowego rozwiązania problemu **wzajemnego wykluczania** dla dwóch procesów. Znajdź kontrprzykład, w którym to rozwiązanie zawodzi. Okazuje się, że nawet recenzenci renomowanego czasopisma „Communications of the ACM" dali się zwieść.

```
1   shared boolean blocked [2] = { false, false };
2   shared int turn = 0;
3
4   void P (int id) {
5     while (true) {
6       blocked[id] = true;
7       while (turn != id) {
8         while (blocked[1 - id])
9           continue;
10        turn = id;
11      }
12      /* put code to execute in critical section here */
13      blocked[id] = false;
14    }
15  }
16
17  void main() { parbegin (P(0), P(1)); }
```

Algorytmy wzajemnego wykluczania (ang. mutual exclusion) są używane w przetwarzaniu współbieżnym, żeby uniknąć równoczesnego użycia wspólnego zasobu przez różne wątki/procesy w sekcjach krytycznych.

Sekcja krytyczna jest fragmentem kodu, w którym wątki (lub procesy) odwołują się do wspólnego zasobu.

Linijki dotyczą kodu z treści zadania.

Na początek będziemy chcieli, żeby program P0 wykonał po kolei następujące:

Linijka 6
Linijka 7
Linijka 12
Linijka 13

Teraz program P1:

Linijka 6
Linijka 7
Linijka 8

I znowu PO:

Linijka 6
Linijka 7
Linijka 12

I na koniec P1:

Linijka 10
Linijka 7
Linijka 12

Po zakończeniu oba programy znajdują się w sekcji krytycznej.