Systemy operacyjne 2016

Lista zadań nr 6

Na zajęcia 17 listopada 2016

UWAGA! Rozwiązania poniższych zadań należy zapisać na kartce formatu A4 i oddać prowadzącemu przed zajęciami. Zakładamy, że wszystkie używane semafory są silne!

Zadanie 1. Rozważmy poniższy program, gdzie rozpoczęcie współbieżnego wykonania procesów odbywa się z użyciem dyrektywy `parbegin`.

```
1  const int n = 50;
2  shared int tally = 0;
3
4  void total() {
5    for (int count = 1; count <= n; count++)
6      tally++;
7  }
8
9  void main() {
10   parbegin (total(), total());
11 }
```

Wyznacz najmniejszą i największą możliwą wartość współdzielonej zmiennej `tally`. Maszyna wykonuje instrukcje arytmetyczne wyłącznie na rejestrach – tj. kompilator musi załadować wartość zmiennej `tally` do rejestru, przed wykonaniem dodawania. Jak zmienią się wartości, gdy wystartujemy n procesów zamiast dwóch? Odpowiedź uzasadnij.

Zadanie 2. Poniżej znajduje się propozycja[1] programowego rozwiązania problemu wzajemnego wykluczania dla dwóch procesów. Znajdź kontrprzykład, dla którego to rozwiązanie nie działa. 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() {
18   parbegin (P(0), P(1));
19 }
```

[1] Harris Hyman, *"Comments on a Problem in Concurrent Programming Control."*, January 1966.

Zadanie 3. Poniżej podano nieprawidłową implementację semafora zliczającego z użyciem semaforów binarnych. Znajdź kontrprzykład i zaprezentuj wszystkie warunki niezbędne do jego odtworzenia.

```
1  struct csem {
2    bsem mutex;
3    bsem delay;
4    int count;
5  };
6
7  void init(csem &s, int v)
8  {
9    s.mutex = 1;
10   s.delay = 0;
11   s.count = v;
12 }
```

```
13 void wait(csem &s) {
14   wait (s.mutex);
15   s.count--;
16   if (s.count < 0) {
17     signal (s.mutex);
18     wait (s.delay);
19   } else {
20     signal (s.mutex);
21   }
22 }
```

```
23 void signal(csem &s) {
24   wait (s.mutex);
25   s.count++;
26   if (s.count <= 0)
27     signal (s.delay);
28   signal (s.mutex);
29 }
```

Zadanie 4. Rozważmy zasób, do którego dostęp jest możliwy wyłącznie w podprogramie otoczonym parą wywołań `acquire` i `release`. Chcemy by wymienione operacje miały następujące właściwości:

- mogą być co najwyżej trzy procesy współbieżnie korzystające z zasobu,
- jeśli w danej chwili zasób ma mniej niż trzech użytkowników, to możemy bez opóźnień przydzielić zasób kolejnemu procesowi,
- jednakże, gdy zasób ma już trzech użytkowników, to muszą oni wszyscy zwolnić zasób, zanim zaczniemy dopuszczać do niego kolejne procesy,
- operacja `acquire` wymusza porządek „pierwszy na wejściu, pierwszy na wyjściu" (ang. *FIFO*).

Podaj co najmniej jeden kontrprzykład wskazujący na to, że poniższe rozwiązanie jest niepoprawne.

```
1  semaphore mutex = 1;              // implementuje sekcję krytyczną
2  semaphore block = 0;              // oczekiwanie na opuszczenie zasobu
3  shared int active = 0;            // ilość użytkowników zasobu
4  shared int waiting = 0;           // ilość użytkowników oczekujących na zasób
5  shared boolean must_wait = false; // czy kolejni użytkownicy muszą czekać?
6
7  void acquire() {
8    wait(mutex);
9    if (must_wait) {                // podpowiedź: czy while zamiast if coś zmieni?
10     waiting++;
11     signal(mutex);
12     wait(block);
13     wait(mutex);
14     waiting--;
15   }
16   active++;
17   must_wait = (active == 3);
18   signal(mutex);
19 }
20
21 void release() {
22   wait(mutex);
23   active--;
24   if (active == 0) {
25     int n = min(waiting, 3);
26     while (n > 0) {
27       signal(block);
28       n--;
29     }
30     must_wait = false;
31   }
32   signal(mutex);
33 }
```

Zadanie 5. Poniżej podano jedno z możliwych rozwiązań **problemu ucztujących filozofów**[2]. Przypuśćmy, że istnieją dwa rodzaje filozofów: leworęczny i praworęczny; którzy podnoszą odpowiednio lewy i prawy widelec jako pierwszy. Widelce są ponumerowane odwrotnie do wskazówek zegara. Pokaż, że jakikolwiek układ pięciu ucztujących filozofów z co najmniej jednym leworęcznym i praworęcznym zapobiega zakleszczeniom i głodzeniu.

```
1  semaphore fork [5] = {1};
2
3  void righthanded (int i) {
4    while (true) {
5      think ();
6      wait (fork[(i+1) mod 5]);
7      wait (fork[i]);
8      eat ();
9      signal (fork[i]);
10     signal (fork[(i+1) mod 5]);
11   }
12 }
13
14 void lefthanded (int i) {
15   while (true) {
16     think ();
17     wait (fork[i]);
18     wait (fork[(i+1) mod 5]);
19     eat ();
20     signal (fork[(i+1) mod 5]);
21     signal (fork[i]);
22   }
23 }
24
25 void main() {
26   parbegin( ?handed(0), ?handed(1), ?handed(2), ?handed(3), ?handed(4));
27 }
```

Zadanie 6. Nawet, gdy uda nam się zapisać poprawny współbieżny program, to może się zdarzyć, że będzie się on wykonywał niepoprawnie na maszynie **SMP** (ang. *Symmetric MultiProcessing*). Pisząc programy współbieżne na takie maszyny trzeba brać pod uwagę właściwy **model pamięci**.

Może się zdarzyć, że procesor stwarzający iluzję sekwencyjnego przetwarzania instrukcji danego wątku będzie wykonywał operacje na pamięci **poza porządkiem programu** (ang. *Out-of-Order execution*). W szczególności inne procesory mogą obserwować kolejność wprowadzania zapisów do pamięci głównej, w innym porządku, niż wynikałoby to z instrukcji programu.

Rozważmy dowolny przeplot równoległego wykonania poniższych dwóch programów. Wszystkie zmienne są przechowywane w pamięci oraz początkowo `a = 1` i `b = 2`. Podaj możliwe wartości zmiennych `c` i `d` po wykonaniu programów z uwzględnieniem instrukcji **barier pamięciowych**[3].

```
a := 3;          c := b;
...              ...
mb();            rmb();
...              ...
b := 4;          d := a;
```

[2] Tanenbaum, §2.5.1
[3] Stallings, §6.8