Systemy operacyjne 2016

Lista zadań nr 6

Na zajęcia ?

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:

```
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 }
```

Procedura `parbegin` rozpoczyna współbieżne wykonanie procesów, których liczba zależy od ilości argumentów. Każdy z procesów zaczyna wykonywać funkcję określoną przez odpowiedni argument.

Wyznacz dolną i górną granicę na ostateczną wartość zmiennej `tally`. Załóż, że procesy mogą wykonywać się z dowolną prędkością, a maszyna nie posiada instrukcji arytmetycznych operujących na pamięci (tj. musi załadować wartość zmiennej `tally` do rejestru). Jak zmienią się granice, gdy wystartujemy N procesów zamiast dwóch? Uzasadnij swoją odpowiedź.

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     /* critical section */
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. 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 }
```

```
20 void wait(csem &s) {
21   wait (s.mutex);
22   s.count--;
23   if (s.count < 0) {
24     signal (s.mutex);
25     wait (s.delay);
26   } else {
27     signal (s.mutex);
28   }
29 }
```

```
13 void signal(csem &s) {
14   wait (s.mutex);
15   s.count++;
16   if (s.count <= 0)
17     signal (s.delay);
18   signal (s.mutex);
19 }
```

Zadanie 4. Rozważmy zasób z operacjami `acquire` i `release` taki, że:

- mogą być co najwyżej trzy procesy korzystające z tego zasobu;
- jeśli jest mniej niż trzech użytkowników zasobu w bieżącej chwili to natychmiastowo dopuszczamy do zasobu kolejne procesy;
- jednakże, gdy dojdziemy do sytuacji gdzie zasób ma trzech użytkowników to muszą oni wszyscy zwolnić zasób zanim go oddamy następnym procesom.

Pokaż, 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. Przypuśćmy, że istnieją dwa typy ucztujących filozofów — leworęczny i praworęczny, którzy podnoszą odpowiednie lewy i prawy widelec jako pierwszy. Widelce są ponumerowane odwrotnie do wskazówek zegara. Pokaż, że jakikolwiek układ pięciu ucztujących filozofów z co najmniej jednym z każdej grupy — leworęcznych i praworęcznych — 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 już uda nam się zapisać poprawny współbieżny program może się zdarzyć, że procesor zrobi nieoczekiwaną rzecz tj. zmieni kolejność wprowadzania zapisów do pamięci głównej. Ma to związek z przyjętym modelem pamięci, który może mieć wpływ na semantykę programów współbieżnych wykonujących się na różnych procesorach.

Mamy dwie zmienne przechowywane w pamięci a i b zainicjowane odpowiednio na 1 i 2. Jakim możliwym błędom zapobiega użycie barier pamięciowych[2] w poniższym kodzie? Zakładamy, że instrukcje w danym wierszu wykonują się równolegle, ale ich skutki mogą nie być widoczne dla drugiego wątku natychmiastowo.

Wątek 1	Wątek 2
a = 3;	-
mb();	-
b = 4;	c = b;
-	rmb();
-	d = a;

[2] Stallings; rozdział 6.8