Systemy komputerowe

Lista zadań nr C

Na zajęcia 8,17,18 czerwca 2020

Należy przygotować się do zajęć czytając następujące rozdziały książek:

- Tanenbaum (wydanie czwarte): 2.3, 2.5
- Arpaci-Dusseau: 26 (Concurrency: An Introduction[1]), 27 (Interlude: Thread API[2]), 28 (Locks[3]), 31 (Semaphores[4]), 32 (Common Concurrency Problems[5])

UWAGA! W trakcie prezentacji należy być gotowym do zdefiniowania pojęć oznaczonych **wytłuszczoną** czcionką.

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.

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.

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*).

[1] `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf`
[2] `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf`
[3] `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-locks.pdf`
[4] `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-sema.pdf`
[5] `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-bugs.pdf`

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)); }
```

Zadanie 5. Algorytm Petersona[7] rozwiązuje programowo problem wzajemnego wykluczania. Zreferuj poniższą wersję implementacji tego algorytmu dla dwóch procesów. Wykaż jego poprawność.

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

Ciekawostka: Czasami ten algorytm stosuje się w praktyce dla architektur bez instrukcji atomowych np.: tegra_pen_lock[8].

Zadanie 6. Podaj w pseudokodzie implementację **semafora** z operacjami «`init`», «`down`» i «`up`» używając wyłącznie muteksów i zmiennych warunkowych standardu POSIX.1. Dopuszczamy ujemną wartość semafora.

Podpowiedź: `struct semaphore { pthread_mutex_t critsec; pthread_cond_t waiters; int count; };`

Zadanie 7. Poniżej podano jedno z rozwiązań **problemu ucztujących filozofów** Zakładamy, że istnieją tylko leworęczni i praworęczni filozofowie, którzy podnoszą odpowiednio lewą i prawą pałeczkę jako pierwszą. Pałeczki są ponumerowane zgodnie z ruchem wskazówek zegara. Udowodnij, że jakikolwiek układ $n \geq 5$ ucztujących filozofów z co najmniej jednym leworęcznym i praworęcznym zapobiega zakleszczeniom i głodzeniu.

```
semaphore fork[N] = {1, 1, 1, 1, 1, ...};
```

[6] Harris Hyman, „Comments on a Problem in Concurrent Programming Control", January 1966.
[7] https://en.wikipedia.org/wiki/Peterson's_algorithm
[8] https://elixir.bootlin.com/linux/latest/source/arch/arm/mach-tegra/sleep-tegra20.S

```
void righthanded (int i) {
  while (true) {
    think();
    P(fork[(i+1) mod N]);
    P(fork[i]);
    eat();
    V(fork[i]);
    V(fork[(i+1) mod N]);
  }
}
```

```
void lefthanded (int i) {
  while (true) {
    think();
    P(fork[i]);
    P(fork[(i+1) mod N]);
    eat();
    V(fork[(i+1) mod N]);
    V(fork[i]);
  }
}
```

Zadanie 8 (2pkty). Rozważmy zasób, do którego dostęp jest możliwy wyłącznie w kodzie 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.

```
mutex = semaphore(1)  # implementuje sekcję krytyczną
block = semaphore(0)  # oczekiwanie na opuszczenie zasobu
active = 0            # liczba użytkowników zasobu
waiting = 0           # liczba użytkowników oczekujących na zasób
must_wait = False     # czy kolejni użytkownicy muszą czekać?
```

```
def acquire():
  mutex.wait()
  if must_wait: # czy while coś zmieni?
    waiting += 1
    mutex.signal()
    block.wait()
    mutex.wait()
    waiting -= 1
  active += 1
  must_wait = (active == 3)
  mutex.signal()
```

```
def release():
  mutex.wait()
  active -= 1
  if active == 0:
    n = min(waiting, 3);
    while n > 0:
      block.signal()
      n -= 1
    must_wait = False
  mutex.signal()
```