Systemy operacyjne

Lista zadań nr 12

Na zajęcia 21 i 22 stycznia 2020

Należy przygotować się do zajęć czytając następujące rozdziały książek:

- Tanenbaum (wydanie czwarte): 2.3, 2.5, 6.2, 6.6
- Arpaci-Dusseau: 28 (Locks[1]), 31 (Semaphores[2]), 32 (Common Concurrency Problems[3])

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

Zadanie 1. Podaj cztery warunki konieczne do zaistnienia zakleszczenia. Przeczytaj §6.6 oraz §32.3, a następnie wyjaśnij w jaki sposób można **przeciwdziałać zakleszczeniom** (ang. *deadlock prevention*)? Które z proponowanych rozwiązań stosuje się w praktyce i dlaczego? Czemu pozostałe są zbyt kłopotliwe?

Zadanie 2 (P). Podaj w pseudokodzie semantykę **instrukcji atomowej** `compare-and-swap` i z jej użyciem zaimplementuj **blokadę wirującą** (ang. *spin lock*) (§28.7). Zakładając, że typ «`spin_t`» jest równoważny «`int`», podaj ciało procedur «`void lock(spin_t *)`» i «`void unlock(spin_t *)`». Dlaczego użycie blokad wirujących ma sens tylko w oprogramowaniu uruchamianym na maszynach wieloprocesorowych? Podaj główne różnice między blokadami wirującymi, a **blokadami usypiającymi** (§28.14).

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*).

Zadanie 4. Poniżej podano błędną implementację semafora zliczającego z użyciem semaforów binarnych. Dopuszczamy, żeby «`count`» był liczbą ujemną, w takim przypadku jej wartość bezwzględna oznacza liczbę uśpionych procesów. 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 csem::csem(int v) {
8    mutex = 1;
9    delay = 0;
10   count = v;
11 }
```

```
13 void csem::P() {
14   P(mutex);
15   count--;
16   if (count < 0) {
17     V(mutex);
18     P(delay);
19   } else {
20     V(mutex);
21   }
22 }
```

```
23 void csem::V() {
24   P(mutex);
25   count++;
26   if (count <= 0)
27     V(delay);
28   V(mutex);
29 }
```

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

Zadanie 5. 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, ...};
```

```
1  void righthanded (int i) {
2    while (true) {
3      think();
4      P(fork[(i+1) mod N]);
5      P(fork[i]);
6      eat();
7      V(fork[i]);
8      V(fork[(i+1) mod N]);
9    }
10 }
```

```
13 void lefthanded (int i) {
14   while (true) {
15     think();
16     P(fork[i]);
17     P(fork[(i+1) mod N]);
18     eat();
19     V(fork[(i+1) mod N]);
20     V(fork[i]);
21   }
22 }
```