Systemy operacyjne

Lista zadań nr 6

Na zajęcia 4–5 grudnia 2018

UWAGA! W trakcie prezentacji należy być gotowym do zdefiniowania pojęć oznaczonych **wytłuszczoną** czcionką. Zadania oznaczone **(S)** proszę rozwiązać samodzielnie! Rozwiązanie zadania oznaczonego **(P)** należy pokazać z użyciem rzutnika.

Zadanie 1. Podaj w pseudokodzie semantykę **instrukcji atomowej** `compare-and-swap` i z jej użyciem zaimplementuj **blokadę wirującą** (ang. *spin-lock*). W jakich systemach komputerowych stosuje się ten typ blokad? Wymień zalety i wady blokad wirujących w porównaniu do **blokad usypiających**. Opisz rozwiązanie pośrednie, czyli **blokady adaptacyjne**.

Zadanie 2. Poniżej widnieje rozwiązanie problemu wzajemnego wykluczania Peterson'a dla dwóch procesów (Tanenbaum, §2.3.3). Uzasadnij jego poprawność i pokaż jak można go rozszerzyć na wiele procesów.

```
1  volatile int turn = 0;
2  volatile int interested[2] = {0, 0};
3
4  void enter_critical(int process) {
5    interested[process] = 1;
6    turn = process;
7    while (turn == process && interested[1 - process] == 1) {}
8  }
9
10 void leave_critical(int process) {
11   interested[process] = 0;
12 }
```

Ciekawostka: Czasami ten algorytm stosuje się w praktyce dla architektur bez instrukcji atomowych np.: tegra_pen_lock[1].

Zadanie 3 (S). Poniżej podano jedno z rozwiązań **problemu ucztujących filozofów** (Tanenbaum, §2.5.1). Zakładamy, że istnieją tylko leworęczni i praworęczni filozofowie, którzy podnoszą odpowiednio lewy i prawy widelec jako pierwszy. Widelce są ponumerowane zgodnie ze wskazówkami zegara. Udowodnij, że jakikolwiek układ pięciu lub więcej 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      wait (fork[(i+1) mod N]);
5      wait (fork[i]);
6      eat ();
7      signal (fork[i]);
8      signal (fork[(i+1) mod N]);
9    }
10 }
```

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

Zadanie 4 (P). Podaj w pseudokodzie implementację **blokady współdzielonej** z operacjami «`init`», «`rdlock`», «`wrlock`» i «`unlock`» używając wyłącznie muteksów i zmiennych warunkowych. Nie definiujemy zachowania dla następujących przypadków: zwalnianie blokady do odczytu więcej razy niż została wzięta; zwalnianie blokady do zapisu, gdy nie jest się jej właścicielem; wielokrotne zakładanie blokady do zapisu z tego samego wątku. Jeśli rozwiązanie dopuszcza głodzenie, to podaj propozycję jak to naprawić.

Podpowiedź: `RWLock = {owner: Thread, readers: int, critsec: Mutex, noreaders: CondVar, nowriter: CondVar, writer: Mutex}`

[1] https://elixir.bootlin.com/linux/latest/source/arch/arm/mach-tegra/sleep-tegra20.S

Zadanie 5 (S). Poniżej podano błędną 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    s.mutex = 1;
9    s.delay = 0;
10   s.count = v;
11 }
```

```
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 6 (S). 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. Następnie zaproponuj poprawki do poniższego kodu i uzasadnij poprawność zmodyfikowanego rozwiązania.

```
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?
```

```
1  def acquire():
2    mutex.wait()
3    if must_wait: # czy while coś zmieni?
4      waiting += 1
5      mutex.signal()
6      block.wait()
7      mutex.wait()
8      waiting -= 1
9    active += 1
10   must_wait = (active == 3)
11   mutex.signal()
```

```
12 def release():
13   mutex.wait()
14   active -= 1
15   if active == 0:
16     n = min(waiting, 3);
17     while n > 0:
18       block.signal()
19       n -= 1
20     must_wait = False
21   mutex.signal()
```

Zadanie 7 (P). Opisz semantykę operacji «FUTEX_WAIT» i «FUTEX_WAKE» mechanizmu futex(2)[2] wykorzystywanego w systemie LINUX do implementacji środków synchronizacji w przestrzeni użytkownika. Podaj w pseudokodzie[3] implementację funkcji «lock» i «unlock» **semafora binarnego** korzystając wyłącznie z **futeksów** i atomowej instrukcji compare-and-swap. Odczyty i zapisy komórek pamięci są atomowe.

Podpowiedź: Wartość futeksa wyraża stany: (0) $unlocked$, (1) $locked \wedge |waiters| = 0$, (2) $locked \wedge |waiters| \geq 0$.

[2] http://man7.org/linux/man-pages/man7/futex.7.html

[3] „Python is executable pseudocode. Perl is executable line noise.” – Bruce Eckel