# Systemy operacyjne (zaawansowane)

Lista zadań nr 6

Na zajęcia 23 listopada 2017

**UWAGA!** W trakcie prezentacji rozwiązań należy zdefiniować i wyjaśnić pojęcia, które zostały oznaczone **wytłuszczoną** czcionką. Zadania oznaczone jako **(P)** proszę rozwiązać samodzielnie (!!!), a następnie spisać na kartce formatu A4 i oddać prowadzącemu na początku zajęć.

Laptopem z rzutnikiem będą dostępne w trakcie zajęć. W zadaniach oznaczonych **(R)** celem sprawnej prezentacji rozwiązania należy wyświetlić pseudokod odczytany z przenośnego dysku studenta.

**Zadanie 1 (R).** Opisz semantykę operacji `FUTEX_WAIT` i `FUTEX_WAKE` mechanizmu futex(2)[1] wykorzystywanego w systemie *Linux* do implementacji środków synchronizacji w przestrzeni użytkownika. Podaj w pseudokodzie[2] implementację funkcji `lock` i `unlock` **semafora binarnego** korzystając wyłącznie z **futeksów** i **atomowej instrukcji** `compare-and-swap`. Odczyty i zapisy są atomowe.

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

**Zadanie 2 (R).** Podaj w pseudokodzie implementację **semafora** z operacjami `init`, `wait` i `post` używając wyłącznie muteksów i zmiennych warunkowych. Dopuszczamy ujemną wartość semafora.

**Podpowiedź:** `Semaphore = { critsec: Mutex, waiters: CondVar, count: int, wakeups: int }`

**Zadanie 3 (R).** Podaj w pseudokodzie implementację **blokady współdzielonej** z operacjami `init`, `rdlock`, `wrlock` i `unlock` używając wyłącznie muteksów i zmiennych warunkowych. Zachowanie implementacji może być niezdefiniowane 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.

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

**Zadanie 4.** Jądro *Linuksa* oferuje mechanizm synchronizacji zwany seqlock[3] (ang. *sequential lock*). Wyjaśnij zasadę działania **seqlock** i odpowiedz na pytania: Jaki problem rozwiązuje `seqlock`? Jaki jest najczęstszy przewidywany przypadek użycia? W jakich warunkach nie należy go stosować?

**Zadanie 5.** Jedną z technik zachowywania spójności współdzielonych struktur danych bez stosowania blokad jest **RCU** (ang. *read-copy-update*) wyjaśniona w artykule What is RCU, Fundamentally?[4]. Wymień ograniczenia implikowane przez użycie techniki RCU i podaj przykłady w jakich jej stosowanie jest uzasadnione, a następnie odpowiedz na poniższe pytania:

- Czemu użycie procedury `rcu_assign_pointer` i `rcu_dereference` jest niezbędne do modyfikacji i odpowiednio przeglądania struktury danych takich jak lista dwukierunkowa?
- Czemu przeglądanie listy z użyciem `list_for_each_entry_rcu` nie spowoduje błędu, jeśli równolegle wykonuje się operacja `list_add_rcu` lub `list_add_tail_rcu`?
- Kiedy zakończy się wywołanie `synchronize_rcu` tak, by można bylo zwolnić element usunięty procedurą `list_del_rcu` lub `list_replace_rcu`?

[1] `http://man7.org/linux/man-pages/man7/futex.7.html`
[2] „Python is executable pseudocode. Perl is executable line noise.” – Bruce Eckel
[3] `https://0xax.gitbooks.io/linux-insides/content/SyncPrim/sync-6.html`
[4] `https://lwn.net/Articles/262464/`

**Zadanie 6 (P).** 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 7 (P).** 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.

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

**Zadanie 8 (P).** Poniżej podano jedno z możliwych rozwiązań **problemu ucztujących filozofów**[5]. 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, 1, 1, 1, 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 }
```

[5] Tanenbaum, §2.5.1