

Systemy operacyjne

Lista zadań nr 12

Na zajęcia 11 stycznia 2024

Należy przygotować się do zajęć czytając następujące materiały: [1, 2.3], [2, 6.1 – 6.6], [3, 28, 31 i 32].

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

Zadanie 1. Zapoznaj się z poniższym programem. Rozważamy wartości przechowywane w zmiennych: «myid», «strtab», «vargp», «cnt», «argc» i «argv[0]». Określ czy są one **współdzielone** i które z nich będą źródłem **wyścigów** (ang. *data race*).

```
1 __thread long myid;
2 static char **strtab;
3
4 void *thread(void *vargp) {
5     myid = *(long *)vargp;
6     static int cnt = 0;
7     printf("[%ld]: %s (cnt=%d)\n", myid, strtab[myid], ++cnt);
8     return NULL;
9 }
10
11 int main(int argc, char *argv[]) {
12     ...
13     strtab = argv;
14     while (argc > 0) {
15         myid = --argc;
16         pthread_create(&tid, NULL, thread, (void *)&myid);
17     }
18     ...
19 }
```

Zadanie 2. Podaj definicję **sekcji krytycznej** [2, 6.2]. Następnie wymień i uzasadnij założenia jakie musi spełniać rozwiązanie problemu sekcji krytycznej. Czemu w programach przestrzeni użytkownika do jej implementacji nie możemy używać **wyłączania przerw** (ang. *interrupt disable*)? Odwołując się do *Prawa Amdahla* powiedz czemu programistom powinno zależeć na tym, by sekcje krytyczne były możliwie jak najkrótsze – określa się to również mianem **blokowania drobnociarnistego** (ang. *fine-grained locking*).

Zadanie 3. Podaj w pseudokodzie semantykę **instrukcji atomowej** compare-and-swap i przy jej pomocy zaimplementuj **blokadę wirującą** (ang. *spin lock*) [3, 28.7]. Niech typ «spin_t» będzie równoważny «int». Podaj ciała procedur «void lock(spin_t *)» i «void unlock(spin_t *)». Czemu blokada wirująca nie jest **sprawiedliwa** (ang. *fair*) [3, 28.8]? Uruchamiamy n identycznych wątków. Kolejno każdy z nich wchodzi do sekcji krytycznej, po czym zostaje wywołany przez jądro. Ile czasu zajmie wszystkim wątkom jednokrotne przejście przez sekcję krytyczną – algorytm planisty to **round-robin**, kwant czasu wynosi 1ms.

Zadanie 4. Wiemy, że **aktywne czekanie** (ang. *busy waiting*) nie jest właściwym sposobem oczekiwania na zwolnienie blokady. Powstaje zatem potrzeba stworzenia **blokadę usypiającej**. Czemu oddanie czasu procesora funkcją «yield» [3, 28.13] nie rozwiązuje wszystkich problemów, które mieliśmy z blokadami wirującymi? Zreferuj implementację podaną w [3, 28.14]. Czemu jest ona niepoprawna bez użycia funkcji «setpark»?

Zadanie 5. Podaj cztery warunki konieczne do zaistnienia zakleszczenia. Na podstawie [3, 32.3] wyjaśnij w jaki sposób można **przeciwdziałać zakleszczeniom** (ang. *deadlock prevention*)? Narzędzie lockdep, stosowane w jądrze *Linux* i *Mimiker*, buduje graf skierowany, w którym wierzchołkami są **klasy blokad**. Jak lockdep wykrywa, że może wystąpić zakleszczenie? Z jakimi scenariuszami sobie nie radzi?

Podpowiedź: Narzędzie lockdep jest przystępnie wyjaśnione w rozdziale 3.4 pracy licencjackiej „*Dynamic Verification of Con-*

Zadanie 6. Poniżej znajduje się propozycja² programowego rozwiązania problemu **wzajemnego wykluczania** (ang. *mutual exclusion*) dla dwóch procesów. Znajdź kontrprzykład, w którym to rozwiązanie zawodzi.

```
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)); }
```

Ciekawostka: Okazuje się, że nawet recenzenci renomowanego czasopisma „*Communications of the ACM*” dali się zwieść.

Zadanie 7. Algorytm **Petersona**³ rozwiązuje programowo problem wzajemnego wykluczania. Zreferuj poniższą wersję implementacji tego algorytmu dla dwóch procesów. Uzasadnij jego poprawność.

```
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         turn = 1 - id;
8         while (blocked[1 - id] && turn == (1 - id))
9             continue;
10        /* put code to execute in critical section here */
11        blocked[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**⁴.

Zadanie 8. Poniżej podano błędną implementację **semafora zliczającego** przy pomocy **semaforów binarnych**. Wartość «count» może być ujemna – wartość bezwzględna oznacza wtedy 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 }
12
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
24 void csem::V() {
25     P(mutex);
26     count++;
27     if (count <= 0)
28         V(delay);
29     V(mutex);
30 }
```

¹<https://ii.uni.wroc.pl/media/uploads/2021/12/06/praca-urbanczyk.pdf>

²Harris Hyman, „*Comments on a Problem in Concurrent Programming Control*”, January 1966.

³https://en.wikipedia.org/wiki/Peterson's_algorithm

⁴<https://elixir.bootlin.com/linux/latest/source/arch/arm/mach-tegra/sleep-tegra20.S>

Zadanie 9. Przeanalizuj poniższy pseudokod wadliwego rozwiązania problemu **producent-konsument**. Zakładamy, że kolejka «queue» przechowuje do n elementów. Wszystkie operacje na kolejce są atomowe. 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*).

Literatura

- [1] „*Systemy operacyjne*”
Andrew S. Tanenbaum, Herbert Bos
Helion; wydanie czwarte; 2015
- [2] „*Operating System Concepts*”
Abraham Silberschatz, Peter Baer Galvin; Greg Gagne
Wiley; wydanie dziesiąte; 2018
- [3] „*Operating Systems: Three Easy Pieces*”
Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau
<https://pages.cs.wisc.edu/~remzi/OSTEP/>