Systemy operacyjne

Lista zadań nr 13

Na zajęcia 28 i 29 stycznia 2020

Należy przygotować się do zajęć czytając następujące rozdziały książek:

- 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. Przeczytaj rozdział 2 artykułu „Beautiful concurrency[4]". Na podstawie poniższego przykładu wyjaśnij czemu złożenie ze sobą poprawnych współbieżnych podprocedur używających blokad nie musi dać poprawnej procedury (tj. „locks are not composable"). Jak poprawić procedurę «`transfer`»? Czemu według autorów artykułu blokady nie są dobrym narzędziem do strukturyzowania współbieżnych programów?

```
1  class Account {
2    private int balance;
3    synchronized void withdraw(int n) { balance -= n; }
4    synchronized void deposit(int n) { balance += n; }
5  }
6
7  void transfer(Account from, Account to, int amount) {
8    from.withdraw(amount);
9    to.deposit(amount);
10 }
```

Zadanie 2. 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?
```

```
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()
```

[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
[4] https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/beautiful.pdf

Ściągnij ze strony przedmiotu archiwum «so19_lista_13.tar.gz», następnie rozpakuj i zapoznaj się z dostarczonymi plikami.

UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczone w komentarzu napisem «TODO». Możesz użyć procedury «outc» do prezentowania stanu programu, co może się przydać w trakcie odpluskwiania. Należy używać odpowiednich procedur opakowujących, np. «Sem_wait» i «Sem_post», z biblioteki libcsapp.

Zadanie 3 (P). Program «philosophers» jest błędnym rozwiązaniem problemu „ucztujących filozofów". Dla przypomnienia: każdy z filozofów przez pewien czas śpi, bierze odpowiednio prawą i lewą pałeczkę, je ryż z miski przez pewien czas i odkłada pałeczki. Twoim zadaniem jest poprawienie procedury «philosopher» tak by rozwiązania było wolne od zakleszczeń i głodzenia.

UWAGA! W rozwiązaniu studenta wszyscy filozofowie muszą być praworęczni!

Można wprowadzić dodatkowe semafory, a następnie zainicjować je na początku procedury «main», oraz dodać linie do procedury «philosophers». Inne modyfikacje programu są niedopuszczalne.

Zadanie 4 (P). Problem obiadujących dzikusów

Plemię n dzikusów biesiaduje przy wspólnym kociołku, który mieści w sobie $m \leq n$ porcji gulaszu z niefortunnego misjonarza. Kiedy dowolny dzikus chce zjeść, nabiera sobie porcję z kociołka własną łyżką do swojej miseczki i zaczyna jeść gawędząc ze współplemieńcami. Gdy dzikus nasyci się porcją gulaszu to zasypia. Po przebudzeniu znów głodnieje i wraca do biesiadowania. Może się jednak zdarzyć, że kociołek jest pusty. Jeśli kucharz śpi, to dzikus go budzi i czeka, aż kociołek napełni się strawą z następnego niespełnionego misjonarza. Po ugotowaniu gulaszu kucharz idzie spać.

W udostępnionym pliku źródłowym «savages.c» należy uzupełnić procedury realizujące programy kucharza i dzikusa. Rozwiązanie nie może dopuszczać zakleszczenia i musi budzić kucharza wyłącznie wtedy, gdy kociołek jest pusty. Do synchronizacji procesów można używać wyłącznie semaforów POSIX.1.

Zadanie 5 (2, P). Bariera dwuetapowa

Bariera to narzędzie synchronizacyjne, o którym można myśleć jak o kolejce FIFO uśpionych procesów. Jeśli czeka na niej co najmniej n procesów, to w jednym kroku bierzemy pierwszych n procesów z naszej kolejki i pozwalamy im wejść do sekcji kodu chronionego przez barierę. Po przejściu n procesów przez barierę, za pomocą procedury «barrier_wait», musi się ona nadawać do ponownego użycia. Oznacza to, że ma zachowywać się tak, jak bezpośrednio po wywołaniu funkcji «barrier_init». Z naszej bariery może korzystać dużo więcej niż n współbieżnie działających procesów, choć z reguły jest to dokładnie n.

Należy uzupełnić procedury «barrier_init», «barrier_wait» i «barrier_destroy» w pliku źródłowym «barrier.c». Najpierw należy wybrać reprezentację stanu bariery, który będzie trzymany w strukturze o typie «barrier_t». Możesz tam przechowywać wyłącznie semafory POSIX.1 albo zmienne całkowite.

Testowanie bariery odbywa się poprzez symulację „wyścigu koni". Mamy P aktywnych koni. W każdej rundzie wyścigu startuje N koni. Po wykonaniu pewnej liczby rund koń jest już zmęczony i idzie gryźć koniczynę. Zostaje zastąpiony przez nowego wypoczętego konia. Każda runda zaczyna się w momencie, gdy co najmniej N koni znajduje się w boksach startowych. Może się zdarzyć, że w jednej chwili na hipodromie odbywa się więcej niż jeden wyścig, i nie powinno to mieć dla nas żadnego znaczenia.

Zadanie 6 (2, P). Problem palaczy tytoniu

Mamy trzy wątki palaczy i jeden wątek agenta. Zrobienie i zapalenie papierosa wymaga posiadania tytoniu, bibułki i zapałek. Każdy palacz posiada nieskończoną ilość wyłącznie jednego zasobu – tj. pierwszy ma tytoń, drugi bibułki, a trzeci zapałki. Agent kładzie na stole dwa wylosowane składniki. Palacz, który ma brakujący składnik podnosi ze stołu resztę, skręca papierosa i go zapala. Agent czeka, aż palacz zacznie palić, po czym powtarza wykładanie składników na stół. Palacz wypala papierosa i znów zaczyna odczuwać nikotynowy głód.

Wykorzystując plik «smokers.c» rozwiąż problem palaczy tytoniu. Możesz wprowadzić dodatkowe zmienne globalne (w tym semafory) i nowe wątki, jeśli zajdzie taka potrzeba. Pamiętaj, że palacze mają być wybudzani tylko wtedy, gdy pojawią się dokładnie dwa zasoby, których dany palacz potrzebuje.

UWAGA! Modyfikowanie kodu procedury «agent» jest zabronione!