Systemy operacyjne

Lista zadań nr 13

Na zajęcia 25, 26, 27 i 30 stycznia 2023

Należy przygotować się do zajęć czytając następujące materiały: [3, 28, 31 i 32], [1, 2.3.5 i 2.3.6].

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

WAŻNE! We wszystkich zadaniach zakładamy, że implementacja semaforów i planisty zadań jest sprawiedliwa (np. działa zgodnie z polityką `FIFO`). Rozwiązania używające aktywnego czekania uznajemy za błędne!

Zadanie 1. Przypomnij z wykładu na czym polega problem **odwrócenia priorytetów** oraz metodę jego rozwiązywania o nazwie **dziedziczenie priorytetów**? W jakim celu **mutex** pamięta właściciela, tj. wątek który trzyma blokadę? W jaki sposób należy rozszerzyć implementację operacji «`mutex_lock`» i «`mutex_unlock`», żeby nie dopuścić do odwrócenia priorytetów? Czy semafory są odporne na problem odwrócenia priorytetów?

Zadanie 2. Podaj implementację (w języku `C`) **semafora**[1] z operacjami «`init`», «`wait`» oraz «`post`» używając wyłącznie muteksów i zmiennych warunkowych standardu *POSIX.1*. Pamiętaj, że wartość semafora musi być zawsze nieujemna.

Podpowiedź: `typedef struct Sem { pthread_mutex_t mutex; pthread_cond_t waiters; int value; } Sem_t;`

Zadanie 3. Opisz semantykę operacji «`FUTEX_WAIT`» i «`FUTEX_WAKE`» mechanizmu `futex(2)` [1, 2.3.6] wykorzystywanego w systemie `Linux` do implementacji środków synchronizacji w przestrzeni użytkownika. Czym różnią się **blokady adaptacyjne** (ang. *adaptive lock*) od zwykłych blokad usypiających? Zreferuj implementację prostej blokady z operacjami `__lock` i `__unlock`. Przyjmujemy, że zmienna «`libc.need_locks`» ma wartość 1. Funkcje «`__futexwait`» i «`__wake`» są zdefiniowane w pliku `pthread_impl.h`. Instrukcje atomowe zwracają starą wartość modyfikowanej komórki pamięci. Co wyraża wartość blokady? Jak zachowuje się blokada w warunkach wysokiego **współzawodnictwa**? W jakich warunkach usypiamy i wybudzamy wątki?

Zadanie 4. 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*).

```
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.post()
6      block.wait()
7      mutex.wait()
8      waiting -= 1
9    active += 1
10   must_wait = (active == 3)
11   mutex.post()
```

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

Podaj dwa istotnie różne kontrprzykłady wskazujące na to, że powyższe rozwiązanie jest niepoprawne.

[1] Semafor, podobnie jak ten kolejowy, jest płci męskiej.

Ściągnij ze strony przedmiotu archiwum «so21_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 lub odpluskwiania. Należy używać procedur «Sem_wait», «Sem_post» i «Sem_getvalue» z biblioteki libcsapp, gdyż wprowadzają do programu losowe opóźnienia i przełączenia kontekstu.

Zadanie 5. 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.

W Twoim rozwiązaniu 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 6. 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 7 (2). 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* i 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.

Wskazówka: Przyjrzyj się konstrukcji śluzy wodnej, do której może wpływać n statków. Zauważ, że działa ona analogicznie do bariery dwuetapowej przy założeniu, że statki płyną w górę rzeki.

Zadanie 8 (2). 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 wyłącznie wtedy, gdy pojawią się dokładnie dwa zasoby, których dany palacz potrzebuje.

UWAGA! Modyfikowanie kodu procedury «agent» jest zabronione!

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/