Systemy operacyjne (zaawansowane)

Lista programistyczna nr 2

Termin oddawania 7 grudnia 2017

Studenci są zachęcani do przeprowadzania dodatkowych eksperymentów związanych z treścią zadań i dzieleniem się obserwacjami z resztą grupy. Proszę najpierw korzystać z podręcznika systemowego (polecenia `man` i `apropos`) i w ostateczności sięgać do zasobów Internetu. Głównym podręcznikiem do zajęć praktycznych jest „The Linux Programming Interface: A Linux and UNIX System Programming Handbook". Bardziej wnikliwe wyjaśnienia zagadnień można odnaleźć w książce „Advanced Programming in the UNIX Environment".

Rozwiązania mają być napisane w języku `C` (a nie `C++`) i kompilować się bez błędów i ostrzeżeń (opcje: «`-std=gnu11 -Wall -Wextra`») kompilatorem `gcc` lub `clang` pod systemem `Linux`. Do rozwiązań musi być dostarczony plik `Makefile`, tak by po wywołaniu polecenia `make` otrzymać pliki binarne, a polecenie `make clean` powinno zostawić w katalogu tylko pliki źródłowe. Rozwiązania mają być dostarczone poprzez system oddawania zadań na stronie zajęć.

UWAGA! W trakcie prezentacji programów należy wyjaśnić pojęcia, które zostały oznaczone **wytłuszczoną** czcionką. Brak zrozumienia używanych funkcji systemowych może spowodować nieprzydzielenie punktów za zadanie.

Uwagi do realizacji zadań

1. Uważaj na interakcje z procedurami bibliotecznymi! Pamiętaj, że większość procedur z pliku nagłówkowego «`stdio.h`» używa blokad potencjalnie zakłócając działanie testów. Używanie funkcji drukujących komunikaty w sekcji krytycznej nienaturalnie zwiększa współzawodnictwo, a poza sekcją krytyczną wprowadza sekwencjonowanie procesów i wątków.
2. Dbaj o czytelność kodu! Przejrzystość ułatwia analizę i odpluskwianie programu. Nazywaj zmienne i procedury tak, by ich nazwy były samo-objaśniające się. Powiązane ze sobą dane zamykaj w struktury. Minimalizuj rozmiar globalnego stanu.
3. Rozwiązanie prostsze ma większą szansę być poprawne! Masz już rozwiązanie – to jeszcze nie koniec! Przyjrzyj się programowi uważnie. Może należy wprowadzić jakiś nowy środek synchronizacji, który znacząco uprości logikę programu. Czy widzisz powtarzające się sekwencje kodu – może czas zamknąć je w procedurze? Czy program da się skrócić w inny sposób? Prostsze rozwiązanie łatwiej jest przeanalizować i trudniej w nim o błędy.
4. Czy nie zakładasz zbyt dużo? Upewnij się, że nie korzystasz z jakiś założeń, które nie są w sposób bezpośredni podane w treści zadania. Jeśli masz pytania korzystaj z forum Yebood v2[1].
5. Przygotuj się do sprawnej prezentacji rozwiązania! Uzasadnij potrzebę użycia danego środka synchronizacji. Zacznij od prezentacji testu i schodź w głąb struktury programu. Wyjaśnij rozumowanie stojące za rozwiązaniem. Przygotuj się do wytłumaczenia kilku różnych przypadków przeplotu wątków lub procesów. Postaraj się jasno przekazać poprawność swojego rozwiązania nie tylko prowadzącemu zajęcia, ale i innym studentom.

[1] `https://forum.iiuwr.me/tags/so-zaaw`

Zadanie 1. Zaprogramuj semafory, o niżej zadanym interfejsie, dla wątków `pthreads(7)` używając **muteksów** `pthread_mutex_init(3)` oraz **zmiennych warunkowych** `pthread_cond_init(3)`. Pamiętaj, że jedynym wątkiem uprawnionym do zwolnienia blokady jest jej właściciel – tj. wątek, który założył tę blokadę. By wymusić sprawdzanie poprawności operacji na blokadzie nadaj jej wartość początkową `PTHREAD_MUTEX_ERRORCHECK`, a wynik operacji sprawdzaj z użyciem `assert(3)`.

```
typedef struct { ... } sem_t;

void sem_init(sem_t *sem, unsigned value);
void sem_wait(sem_t *sem);
void sem_post(sem_t *sem);
void sem_getvalue(sem_t *sem, int *sval);
```

Zadanie 2. Rozwiąż klasyczny problem *„ucztujących filozofów"* w dwóch wariantach:

(a) Dla wątków z użyciem semaforów z poprzedniego zadania. Wątki tworzy się z wykorzystaniem `pthread_create(3)`. Wątek główny ma **czekać** `pthread_join(3)` na zakończenie wątków pobocznych. Obsługa sygnału `SIGINT` ma **anulować** `pthread_cancel(3)` wykonanie wątków.

(b) Dla procesów z użyciem **semaforów nazwanych** `POSIX.1` opisanych w `sem_overview(7)`. W procesie nadrzędnym należy utworzyć semafor z użyciem `sem_open(3)`. Obsługa sygnału `SIGINT` ma zakończyć procesy potomne i usunąć semafor procedurą `sem_unlink(3)`.

Filozofowie posiadają jednolitą (symetryczną) implementację wyrażoną w poniższym pseudokodzie:

```
def philosopher(int i):
  while True:
    think()
    take_forks(i)
    eat()
    put_forks(i)
```

Procedury `think` i `eat` mają wprowadzać losowe opóźnienie z użyciem funkcji `usleep(3)`.

Zadanie 3. Bariera to narzędzie synchronizacyjne, o którym można myśleć jak o kolejce *FIFO* uśpionych procesów. Jeśli oczekuje na niej co najmniej n procesów, to w jednym kroku odcinamy prefiks kolejki składający się z n procesów i pozwalamy im wejść do sekcji kodu chronionego przez barierę.

Zaprogramuj dwuetapową barierę dla n procesów z operacjami `init`, `wait` i `destroy`. Po przejściu n procesów przez barierę musi się ona nadawać do ponownego użycia – tj. ma zachowywać się tak, jak bezpośrednio po wywołaniu funkcji `init`. Nie wolno robić żadnych założeń co do maksymalnej ilości procesów, które korzystają z bariery. Do implementacji użyj semaforów `sem_overview(7)` i **pamięci dzielonej** `shm_overview(7)` dla procesów.

Przetestuj swój kod bariery implementując wyścig koni składający się z k rund po jednym okrążeniu. Kolejna runda zaczyna się w momencie, gdy co najmniej n koni znajduje się w boksach startowych. Niech proces o nazwie `gates` odpowiada za utworzenie i usunięcie bariery pełniącej rolę n boksów startowych. Każdy z procesów `horse` podłącza się do bariery i bierze udział w k wyścigach.

Zadanie 4. 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.

Zaprogramuj procesy kucharza i dzikusów używając semaforów `POSIX.1`. Rozwiązanie nie może dopuszczać zakleszczenia i musi budzić kucharza wyłącznie wtedy, gdy kociołek jest pusty.

Zadanie 5. Problem wyszukaj-dołącz-usuń

Istnieją trzy rodzaje wątków operujących na liście liczb całkowitych: *wyszukujące*, *dopisujące* i *usuwające*. Wiele wątków *wyszukujących* może działać na liście bez ryzyka naruszenia spójności struktury danych. Wątek *dopisujący* dostawia element na koniec listy – w danej chwili co najwyżej jeden taki wątek może działać współbieżnie z wątkami wyszukującymi. Wątek *usuwający* wyjmuje dowolny element z listy – w związku z tym musi operować na strukturze samodzielnie.

Korzystając z muteksów i zmiennych warunkowych zaimplementuj monitor nadzorujący operacje `search`, `append`, `remove` według powyższych założeń. Przetestuj swoje rozwiązanie z użyciem dużej liczby wątków, których pseudokod podano niżej:

```
n = 100000

def reader():
  while True:
    usleep(random(500))
    search(random(n))

def writer():
  s = set()
  while True:
    usleep(random(1000))
    if random(3) == 0:
      x = random(n)
      s.add(x)
      append(x)
    else:
      remove(s.pop())
```

Zadanie 6. Problem palaczy tytoniu

Przypuśćmy, że istnieją trzy wątki *palaczy* i 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 akcję. Poniżej podano pseudokod *agenta*:

```
def agent():
  while True:
    smoke.wait()
    x = random(3)
    if x == 0:
      tobacco.signal()
      paper.signal()
    if x == 1:
      tobacco.signal()
      matches.signal()
    if x == 2:
      paper.signal()
      matches.signal()
```

Używając semaforów z zadania pierwszego zaimplementuj wątki agenta i palaczy, tak aby spełniały podane wyżej założenia. *Palacze* mają być wybudzani tylko wtedy, gdy pojawią się dokładnie dwa zasoby, których dany palacz potrzebuje.