Systemy operacyjne

Lista programistyczna nr 2

Termin oddawania 21 grudnia 2018

UWAGA! Przeczytaj dokładnie poniższy tekst zanim przystąpisz do rozwiązywania zadań!

Głównym podręcznikiem do zajęć praktycznych jest „The Linux Programming Interface: A Linux and UNIX System Programming Handbook". Należy zapoznać się z treścią §2 w celach poglądowych, a pozostałe rozdziały czytać w razie potrzeby. Bardziej wnikliwe wyjaśnienia zagadnień można odnaleźć w książce „Advanced Programming in the UNIX Environment". Zanim sięgniesz do zasobów Internetu zapoznaj się z odpowiednimi stronami podręcznika systemowego poleceniami «`man`» i «`apropos`».

Rozwiązania mają być napisane w języku `C` (a nie `C++`). 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.

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 pytań i odpowiedzi!
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.

Wybierz i rozwiąż co najwyżej 6 zadań za maksymalnie 12 punktów!

Można uzyskać maksymalnie 6 punktów bonusowych na zajęciach konsultacyjnych.

Można uzyskać maksymalnie 4 punkty bonusowe za rozwiązanie zadań oznaczonych literą B.

Zadanie 1 (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 (1). Problem ucztujących filozofów

Zaprogramuj rozwiązanie przedstawionego na zajęciach problemu ucztujących filozofów. Zrób to z użyciem wątków i 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** wykonanie wszystkich wątków `pthread_cancel(3)`.

Filozofowie posiadają jednolitą (symetryczną) implementację wyrażoną poniższym pseudokodem:

```
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 (1). Podobnie jak w poprzednim zadaniu rozwiąż problem ucztujących filozofów, ale tym razem z użyciem procesów. Należy użyć **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)`.

Zadanie 4 (2). 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`», «`open`», «`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 liczby procesów, które korzystają z bariery, tj. może być ona dużo większa niż n. Do implementacji użyj semaforów `sem_overview(7)` i **pamięci dzielonej** `shm_overview(7)` dla procesów. Weź pod uwagę, że procesy korzystające z bariery nie muszą być skojarzone relacją rodzic-dziecko.

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 5 (2, B). Restauracja ramen.

W restauracji ramen[1] jest pięć siedzeń. Jeśli pojawisz się w restauracji, kiedy jedno z siedzeń jest puste, możesz je zająć od razu. Jeśli wszystkie siedzenia są zajęte, musisz poczekać aż wszystkie pięć osób zje swoje ramen i opuści restaurację. Napisz program implementujący klientów restauracji ramen spełniający powyższe wymagania. Każdy klient musi być procesem. Użyj semaforów i pamięci dzielonej POSIX.1.

Wskazówka: Użyj dodatkowy zmiennych: «`eating`», «`waiting`», «`must_wait`» i dwa semafory.

[1] `https://pl.wikipedia.org/wiki/Ramen`

Zadanie 6 (2). 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 program kucharza i dzikusów używając procesów i semaforów POSIX.1. Rozwiązanie nie może dopuszczać zakleszczenia i musi budzić kucharza wyłącznie wtedy, gdy kociołek jest pusty. Użycie pamięci współdzielonej jest niedopuszczalne.

Zadanie 7 (2). 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 8 (2). 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.

Zadanie 9 (2, B). Problem kolejki górskiej w wesołym miasteczku

Mamy n wątków pasażerów i jeden wątek wózka. Głodni wrażeń pasażerowie czekają na wózek, aby przejechać się kolejką. Wózek może przewieźć C pasażerów, gdzie $C < n$. Kolejka może ruszyć tylko wtedy, gdy wózek jest pełen. Rozwiąż powyższy problem używając muteksów i zmiennych warunkowych. Wątek pasażera powinien wołać funkcje «`board`» i «`unboard`», a wątek wózka «`load`», «`run`» i «`unload`». Pasażerowie nie mogą wsiąść do wózka i wyjść z wózka, póki nie zawołano odpowiednio «`load`» i «`unload`».

Zadanie 10 (2). Problem salonu fryzjerskiego

Salon fryzjerski liczy n krzeseł dla osób oczekujących na strzyżenie oraz jedno krzesło fryzjerskie. Jeśli nie ma klientów, których można by ostrzyc, fryzjer idzie spać. Jeśli klient wejdzie do salonu i nie będzie żadnych wolnych krzeseł, wtedy opuszcza salon z niezadowoleniem. Jeśli fryzjer jest zajęty, ale istnieje jedno krzesło wolne to wtedy klient na nim siada i czeka na swoją kolej. Fryzjer jest wybudzany po pojawieniu się klienta.

Zaimplementuj wątki fryzjera i klientów, tak aby spełniały podane wyżej założenia.

Zadanie 11 (2, B). Problem dwóch wiosek w Peru

W jednym z pasm górskich Andów istnieją dwie wioski przedzielone głębokim kanionem, z których jedna produkuje zboże, a druga hoduje Lamy. Pewien przedsiębiorca postanowił połączyć obydwa końce kanionu stalową liną, aby usprawnić handel między wioskami. Transport odbywa się z użyciem podwieszanych na bloczkach koszy. Po włożeniu dóbr do koszyka rozpędzamy go w kierunku drugiego krańca, a ten zawsze dojeżdża na miejsce. Widać, że nie można wysyłać dóbr z obydwu krańców kanionu na raz. Gdyby ładunki się zderzyły, to spadną w przepaść lub zablokują transport. Dodatkowo mamy ograniczenie na udźwig stalowej liny – w jednej chwili może na niej jechać do pięciu koszyków. Jeśli wioska wyprodukowała jakieś dobra, to należy dać jej szanse na przesłanie do drugiej wioski – nie dopuszczamy głodzenia.

Zaimplementuj rozwiązanie problemu z użyciem wątków lub procesów.