Systemy komputerowe

Pracownia nr 3

Termin oddawania 25 czerwca 2019

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.

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!

Zadanie 1 (2). 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 (3). Problem ucztujących filozofów

Zaprogramuj rozwiązanie „problemu ucztujących filozofów[1]". 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 (3). 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 (4). 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.

[1] `https://en.wikipedia.org/wiki/Dining_philosophers_problem`