Systemy operacyjne

Pracownia 2

Treść zadania zawiera w nawiasach nazwy wywołań bibliotecznych, których należy użyć. Proszę najpierw korzystać z podręcznika systemowego (polecenia `man` i `apropos`), a dopiero potem szukać w Internecie.

Rozwiązania mają być napisane w języku `C` (a nie `C++`). Kompilować się bez błędów (opcje: `-std=gnu99 -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 ćwiczeń na serwerze kno.ii.uni.wroc.pl.

Pożyteczne odnośniki:

1. Podstawy obsługi edytora Vim
2. Szybkie wprowadzenie do GNU Make
3. Krótki opis komend debuggera GDB

Uwaga: Student może nie otrzymać punktów za zadanie, jeśli nie będzie umiał wyjaśnić działania użytej w programie funkcji bibliotecznej wskazanej przez prowadzącego.

Uwagi implementacyjne

1. **Uważaj na interakcje z funkcjami bibliotecznymi!** Pamiętaj, że większość funkcji z `stdio.h` używa blokad potencjalnie zakłócając działanie testu.
2. **Nie można mieszać środków synchronizacyjnych dla procesów i wątków!** Tj. jeśli w zadaniu, które zakłada używanie wątków są potrzebne semafory, to należy je zaimplementować z użyciem zmiennej warunkowej i muteksa.
3. **Pisz kod tak, by dało się go łatwo przeczytać!** Nazwij odpowiednio wszystkie używane zmienne, tak by ich nazwa była samo-objaśniająca. Unikaj wprowadzania zbędnych funkcji. Nie zawsze krótsze nazwy są łatwiejsze do przeczytania (lepiej używać `mutex_lock` niż `lock`). Wszystkie powiązane ze sobą środki synchronizacji zamykaj w strukturze, zamiast “wzbogacać” kod globalnymi zmiennymi.
4. **Pomyśl o sprawnej prezentacji zadania!** Wytłumacz czemu potrzebujesz danych muteksów, semaforów, zmiennych warunkowych itd. Zacznij od prezentacji testu i schodź w dół struktury programu. Wyjaśnij logikę stojąca za rozwiązaniem. Przygotuj się do wytłumaczenia kilku różnych przypadków przeplotu wątków lub procesów. Postaraj się jasno pokazać poprawność swojego rozwiązania nie tylko prowadzącemu zajęcia, ale i innym studentom.
5. **Czy tego zadania nie da zrobić się prościej?** Gdy uda Ci się rozwiązać zadanie to przyjrzyj się mu uważnie. Być może da się je uprościć i będzie łatwiej wytłumaczyć o co w nim chodzi? Może należy zbudować jakiś nowy środek synchronizacji przy pomocy którego łatwiej będzie rozwiązać zadanie? Czy jakiś kod się powtarza i da się go wyabstrahować?
6. **Czy nie zakładasz zbyt dużo?** Upewnij się, że nie korzystasz z jakiś założeń, które nie są w sposób bezpośredni podane w zadaniu. Jeśli masz pytania korzystaj z forum KNO.
7. **Linux implementuje zmienne warunkowe typu Mesa oraz silne semafory i muteksy!**

Zadanie 1

Jednym z klasycznych problemów dot. synchronizacji jest problem czytelników i pisarzy. Rozwiąż ten problem używając semaforów binarnych (`pthread_mutex`) dla wątków (`pthread_create`). Twoje rozwiązanie nie może głodzić ani czytelników ani pisarzy. Pamiętaj, że po wątkach złączalnych (`*_setdetachstate`) trzeba posprzątać (`pthread_join`).

Celem przetestowania swojego rozwiązania startuj w wątku głównym `N` wątków co losową liczbę mikrosekund (`nanosleep`), przy czym jednocześnie ma być uruchomionych nie więcej niż `M` wątków. Prawdopodobieństwo wylosowania czytelnika lub pisarza powinno być konfigurowalne. Wątki mają wprowadzać losowe opóźnienie rzędu 10…100 μs. Wydrukuj najdłuższy czas oczekiwania dla czytelników i pisarzy.

Zadanie 2

Korzystając z semaforów i pamięci dzielonej `POSIX` (`sem_overview`, `shm_overview`) zaimplementuj dwuetapową barierę z trzema operacjami `init`, `wait` i `destroy`, przepuszczającą K procesów. Po przejściu K procesów przez barierę musi się ona nadawać do ponownego użycia – tzn. ma się zachowywać tak jak bezpośrednio po wywołaniu funkcji `init`. Używając Twojej implementacji zaimplementuj wyścig koni składający się z N rund po jednym okrążeniu. Kolejna runda zaczyna się w momencie, gdy wszystkie konie znajdują się w boksach. Zauważ, że Twoje rozwiązanie powinno działać poprawnie również, gdy korzysta z niej więcej niż K wątków.

Zadanie 3

Przekazywanie komunikatów i semafory wraz z pamiecią dzielona są mechanizmem o równoważnej sile wyrazu. Pokażemy to w jedną stronę (prostszą) – tj. przy pomocy kolejek komunikatów `POSIX` (`mq_overview`) zaimplementuj semafory zliczające. Wartość semafora będzie kodowana jako ilość komunikatów w skrzynce. Twój interfejs powinen mieć cztery metody `cs_open`, `cs_wait`, `cs_post`, `cs_close` podobnie jak interfejs semaforów `POSIX`.

Zadanie 4

Korzystając z blokad (`pthread_mutex`) i zmiennych warunkowych (`pthread_cond`) zaimplementuj problem konsumentów i producentów dla ograniczonego bufora długości `N`. Zauważ, że standard `POSIX` dostarcza zmiennych warunkowych typu Mesa. Twoje rozwiązanie ma działać dla wielu wątków producentów i konsumentów (łącznie więcej niż 10).

Napisz test bazujący na fakcie, że istnieje ustalona liczba dóbr (np. 1.000.000). Każdemu konsumentowi wylosuj ilość dóbr, którą zużyje zanim zakończy działanie. Wprowadź losowe opóźnienia przy pomocy `nanosleep`. Nie mieszaj implementacji konsumenta / producenta z implementacją monitora!

Zadanie 5

Problem obiadujących dzikusów. Plemię dzikusów je wspólnie obiad z jednego kociołka, który mieści w sobie M 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. Może się jednak zdarzyć, że kociołek jest pusty – wtedy dzikus budzi kucharza i czeka, aż kociołek napełni się strawą z następnego misjonarza. Po ugotowaniu gulaszu kucharz idzie spać. Gdy dzikus nasyci się porcją gulaszu to zasypia, a po przebudzeniu głodnieje i znów nabiera sobie strawy.

Napisz kod implementujący procesy kucharza i dzikusów. Należy użyć semaforów do synchronizacji procesów. Rozwiązanie nie może dopuszczać zakleszczenia i musi budzić kucharza wyłącznie wtedy, gdy garnek jest pusty.

Zadanie 6

Problem palaczy tytoniu. *Przypuśćmy, że istnieją trzy procesy palaczy i jeden proces agenta. Każdy z palaczy na okrągło robi papierosy i je pali. Zrobienie i zapalenie papierosa wymaga tytoniu, bibułki i zapałek. Każdy palacz posiada nieskończoną ilość wyłącznie jednego typu zasoby tj. pierwszy ma tytoń, drugi bibułki, a trzeci zapałki. Agent kładzie na stole dwa składniki. Palacz, który ma brakujący składnik podnosi ze stołu resztę, skręca papierosa i go zapala. Agent czeka, aż palacz skończy palić po czym powtarza cykl.*

Używając semaforów zaimplementuj wątki agenta i palaczy, tak aby spełniały podane wyżej założenia. Wątek agenta zgłasza dostępność każdego zasobu z osobna. Wątki palaczy mają być wybudzane tylko wtedy, gdy pojawią się dokładnie dwa zasoby, których dany palacz potrzebuje.

Zadanie 7

Problem kolejki górskiej w wesołym miasteczku. *Istnieje n wątków pasażerów i m wątków wózka. Pasażerowie cyklicznie czekają na wózek, aby przejechać się kolejką. Wózek może przewieźć C pasażerów, gdzie C << n. Kolejka może ruszyć tylko wtedy gdy jest pełna. Dopuszczamy na torze wiele wózków – oczywiście nie mogą one zmieniać kolejności.*

Rozwiąż powyższy problem używając monitora Mesa (tj. 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, póki nie zawołano `load`, i nie mogą wyjść z wózka, póki nie zawołano `unload`. Tylko jeden wózek w danej chwili może zapełniać się pasażerami. Pasażerowie mogą opuszczać tylko z jeden wózek w danej chwili.

Zadanie 8

Problem wyszukaj-dołącz-usuń. *Istnieją trzy rodzaje wątków operujących na liście jednokierunkowej: wyszukujące, dołączające i usuwające. Widać, że wiele wątków wyszukujących może działać na liście bez ryzyka uszkodzenia struktury danych. Wątek dołączający dostawia element na koniec listy – w danej chwili co najwyżej jeden taki wątek może działać współbieżnie z wątkami wyszukującymi. Wątki usuwające wyjmują dowolny element z listy – w związku z tym muszą działać na strukturze w pojedynkę.*

Zaimplementuj monitor Mesa nadzorujący operacje `search`, `append`, `remove` według powyższych założeń.

Komunikacja międzyprocesowa bazująca na gniazdach

W poniższych zadaniach należy użyć gniazd domeny uniksowej. Tego samego interfejsu programistycznego używa się do komunikacji z użyciem protokołów sieciowych `TCP` i `UDP`, przy czym jest to bardziej złożone. Będziecie się tego uczyć na kursie “Sieci komputerowe”, więc warto dużo prostszy wariant zrobić już teraz.

Zadanie 9 [bonus]

Mało znaną funkcją gniazd domeny `unix(7)` jest przesyłanie deskryptorów plików z użyciem komunikatów pomocniczych (`cmsg`). Utwórz parę gniazdek do przesyłania datagramów (`socketpair`), a następnie dwóch potomków (`fork`) zamykając niepotrzebne końce. W jednym z potomków otwórz plik (`open`), przeczytaj z niego trochę danych (`read`) i wypisz na `stdout` dodając z początku `pid` (`getpid`). Następnie prześlij deskryptor pliku do drugiego procesu (`sendmsg`) i po odebraniu (`recvmsg`) powtórz wcześniej opisaną operację czytania. Nie wolno korzystać z funkcji `stdio.h`.

Zadanie 10 (2pkt.) [bonus]

Używając gniazd domeny `unix(7)` napisz serwer, który będzie zliczał ilość niezerowych bajtów w ciągu danych wysyłanych przez klientów. Po otrzymaniu bajtu z zerem, należy odesłać tekstowo wartość licznika plus znak końca linii. Za koniec transmisji uznaje się dwa zera pod rząd.

Utwórz gniazdo strumieniowe (`SOCK_STREAM`) przy pomocy funkcji `socket(2)`, a następnie nadaj mu nazwę (widoczną w systemie plików) z użyciem `bind(2)`. Przyjmuj nowe połączenia przy pomocy `accept(2)` by potem czytać (`recv`) i zapisywać (`send`) dane do gniazdka. Po otrzymaniu sygnału `SIGINT` zakończ program zamykając wszystkie gniazda (`close`). Pamiętaj, że sygnały przerywają niektóre blokujące wywołania systemowe z błędem `EINTR` w zmiennej `errno`.

Trudność polega na tym, że **musisz** współbieżnie obsługiwać wiele połączeń bez użycia dodatkowych procesów czy wątków! W tym celu należy nasłuchiwać zdarzeń na gniazdkach przy pomocy funkcji `poll(2)` i stwierdzać czy następna operacja `accept` / `recv` / `send` dla danego gniazda będzie nieblokująca. Do przetestowania serwera może przydać się narzędzie `socat` i urządzenie `/dev/random` do generowania losowego ciągu bajtów.

Zadanie 11 [bonus]

Używając gniazd domeny `unix(7)` napisz prosty serwer realizujący zdalne wywołania procedur. Utwórz gniazdo datagramowe (`SOCK_DGRAM`) przy pomocy funkcji `socket(2)`, a następnie nadaj mu nazwę (widoczną w systemie plików) z użyciem `bind(2)`. Wołaj funkcję `recvfrom(2)` aby odebrać komunikat i `sendto(2)` aby wysłać odpowiedź. Po otrzymaniu sygnału `SIGINT` zakończ program zamykając gniazdo (`close`).

Twój serwer będzie zarządzać pulą identyfikatorów liczbowych z zakresu od `1` do `N`. Zaimplementuj co najmniej dwie procedury o następujących sygnaturach:

- `int acquire()` : pobiera wolny identyfikator z puli, zwraca `0` jeśli pula się wyczerpała,
- `bool release(int id)` : odkłada identyfikator do puli, zwraca `false` jeśli identyfikator nie został wcześniej pobrany.

Wywołaniom zdalnych funkcji będzie odpowiadać wymiana komunikatów reprezentujących: numer funkcji plus jej argumenty oraz wynik – tj. *marshalling* trzeba zrobić ręcznie. Do przetestowania swojego serwera możesz użyć programu `nc` (aka `netcat`) z opcją `-U` lub napisać prosty klient.