Systemy operacyjne

Pracownia 3

Studenci są zachęcani do przeprowadzania dodatkowych eksperymentów związanych z treścią zadań i dzieleniem się obserwacjami z resztą grupy. Dodatkowo student powinien umieć posługiwać się programami `ps`, `kill`, `lsof`, `strace` i `ltrace`.

Treść zadania zawiera nazwy wywołań bibliotecznych, których należy użyć. Proszę wpierw korzystać z podręcznika systemowego (polecenia `man` i `apropos`), a gdy to nie wystarczy z Internetu. W systemie opartym na pakietach debianowych (np. Ubuntu) należy zainstalować pakiety `manpages-dev` (ew. polską wersję), `manpages-posix-dev`.

Rozwiązania mają być napisane w języku `C` (a nie `C++`). Kompilować się bez ostrzeżeń (opcje: `-g -std=gnu99 -Wall -Wextra -Werror`) kompilatorem `gcc` lub `clang` pod systemem `Linux`. Do rozwiązań ma być dostarczony plik `Makefile`, tak by po wywołaniu polecenia:

- `make` – zbudować wszystkie pliki binarne,
- `make progN` – zbudować plik binarny `progN` (przykładowa nazwa rozwiązania),
- `make clean` – pozostawić w katalogu tylko pliki źródłowe.

Plik `Makefile` musi być utworzony ręcznie (programy generujące są niedopuszczalne). Rozwiązania mają być dostarczone w archiwum `tar` (`gz` / `bzip2`), które po dekompresji da jeden katalog w postaci "`${indeks}_${nazwisko}_${imie}_prog${numer_listy}`"[1]. Jeśli wyżej wymienione obostrzenia nie zostaną spełnione, zadania nie będą sprawdzane.

Pożyteczne odnośniki:

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

Dobrym zwyczajem jest przetestować swój program przy pomocy narzędzia `valgrind`, celem znalezienia błędnych odwołań do pamięci (nie zawsze widocznych bezpośrednio). Jeśli potrzebujesz czytać argumenty z linii poleceń – możesz to zrobić przy pomocy funkcji `getopt(3)`.

Informacja: W zadaniach 8 i 9 trzeba 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.

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

[1] Notacja zmiennych powłoki: `${symbol}` jest zamieniany na wartość zmiennej `symbol`.

Zadanie 1

Zasymuluj wywołanie biblioteczne `popen(3)` i `pclose(3)` – z tą różnicą, że podane polecenie wykonuj bez pośrednictwa powłoki, a do komunikacji z potomkiem używaj deskryptora pliku. W tym celu należy utworzyć jeden potok (`pipe`), który będzie przesyłać dane ze standardowego wyjścia potomka (`stdout`) na standardowe wejście (`stdin`) rodzica – lub na odwrót w zależności od parametru polecenia. Standardowemu wejściu i wyjściu odpowiadają deskryptory o numerach `0` i `1`. Po utworzeniu potomka (`fork`) podmień odpowiedni deskryptor (`dup2`), a nastepnie uruchom wybrane polecenie funkcją `execve`. Symulowane wywołanie `pclose(3)` powinno zakończyć potomka przez wysłanie sygnału `SIGHUP`. Utwórz test pokazujący działanie symulowanych funkcji – np. zamianę znaków z dużych na małe i na odwrót, lub zliczanie słów.

Zadanie 2

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 działać 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. Pamiętaj, że większość funkcji z `stdio.h` używa blokady potencjalnie zakłócając działanie testu – drukuj komunikaty wyłącznie po zakończeniu testu.

Zadanie 3

Korzystając z semaforów i pamięci dzielonej `POSIX` (`sem_overview`, `shm_overview`) zaimplementuj dwuetapową barierę dla procesów z trzema operacjami `init`, `wait` i `destroy`. Po przejściu 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 cykliczny wyścig koni.

Zadanie 4

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 5

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 6

Wyobraź sobie restaurację ramen z pięcioma siedzeniami. Jeśli pojawisz się w restauracji, kiedy jedno siedzenie jest puste, możesz od razu je zająć. Jednakże jeśli wszystkich pięć siedzeń jest zajętych, musisz poczekać aż wszystkie pięć osób zje swoje ramen i opuści restaurację, aby zasiąść do stołu. Napisz program implementujący klientów restauracji ramen spełniający powyższe wymagania. Wskazówka: użyj dodatkowy zmiennych: `eating`, `waiting`, `must_wait` i dwa semafory. Każdy klient musi być procesem. Użyj semaforów i pamięci dzielonej `POSIX`.

Zadanie 7

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 8 [2 pkt]

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 9

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.