ISIM: Systemy Komputerowe

Pracownia 3: "Programowanie w systemie Linux"

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 na początku 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`. Dokumentacja biblioteki standardowej GNU C Library wyjaśnia niektóre zagadnienia w bardziej wyczerpujący sposób.

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. Szybka edycja linii poleceń powłoki Bash
2. Podstawy obsługi edytora Vim
3. Wprowadzenie do GNU Make
4. 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)`.

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

Napisz program, który tworzy proces (`fork`) zombie. W procesie nadrzędnym (a nie w konsoli!) wykonaj polecenie `ps`, aby pokazać, że istotnie proces potomny stał się zombiakiem. By zapobiec powstawaniu nieumarłych zignoruj sygnał `SIGCHLD` (`sigaction`). Wariant zadania ma być wybieralny poprzez parametr linii poleceń.

Zadanie 2

Wydrukuj środowisko programu przy pomocy funkcji `getenv` i `getcwd`. Utwórz proces potomny. Czy jeśli zmienisz w rodzicu środowisko (`setenv`) lub bieżący katalog (`chdir`), to potomek będzie widział te zmiany? Napisz program, który to zaprezentuje.

Zadanie 3

Korzystając z semaforów `POSIX` (`sem_overview`) zaimplementuj dwuetapową barierę dla wątków (`pthread_create`) z trzema operacjami `init`, `wait` i `destroy`. Po przejściu wątków 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 – wątki mają losować czas biegu; po dobiegnięciu do mety wszystkich wątków wyścig zaczyna się od nowa.

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

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.

Zadanie 6

Utwórz bibliotekę współdzieloną składającą się z killku modułów – w każdym z nich umieść przynajmniej jedną funkcję. Kod modułów musi być relokowalny – tj. przekaż do kompilatora odpowiednią opcję (`-fPIC`[2]). Biblioteka musi być skonsolidowana inaczej niż plik wykonywalny (`-shared`). Utwórz program korzystający z funkcji Twojej biblioteki wprost (load-time linking) oraz drugi (run-time linking), który będzie używał jawnie dynamicznego konsolidatora (`dlopen`, `dlsym` i `dlclose`) do wyłuskania funkcji po symbolu. Przed i po załadowaniu biblioteki wskaż (programem `pmap`) miejsce w przestrzeni adresowej procesu, gdzie konsolidator umieścił bibliotekę.

[2] Position Independent Code

Zadanie 7

Napisz program, który podczepi sobie kilka stron pamięci anonimowej. Zapisz tam trochę danych, zmień uprawnienia stron na tylko do odczytu (`mprotect`) i powtórz operacje zapisu – dostaniesz sygnał `SIGSEGV`. Przechwyć go (`sigaction`), zinterpretuj dane z nim związane zawarte w drugim i trzecim argumencie procedury obsługi sygnału o typach odpowiednio `siginfo_t` i `ucontext_t`. Wypisz[3] na `stderr` komunikat zawierający informacje o adresie wywołującym błąd (`si_addr`), typie błędu (`si_code`), adresie wierzchołka stosu (`uc_stack`) i adresie instrukcji powodującej błąd (`uc_mcontext`), po czym zakończ działanie programu.

Zadanie 8

Większość zasobów w systemach uniksowych ma semantykę pliku. Do odczytu / zmiany właściwości urządzeń reprezentowanych jako pliki służy funkcja `ioctl(2)`. Napisz program, który będzie odczytywał bieżący rozmiar terminala (`TIOCGWINSZ`) z deskryptora standardowego wejścia (`STDIN_FILENO`) i drukował go na wyjście. Rób to cyklicznie – za każdym razem gdy Twój proces otrzyma sygnał zmiany rozmiaru terminala (`SIGWINCH`). Program ma się zakończyć po otrzymaniu `SIGINT` (skrót `CTRL+C`). Upewnij się, że deskryptor pliku jest terminalem `isatty(3)` w przeciwnym wypadku zakończ program z kodem `EXIT_FAILURE`. Opis operacji kontrolnych znajdziesz w `tty_ioctl(4)`.

Zadanie 9

Przy wprowadzaniu zmian do systemu budowania oprogramowania przydaje się skrypt, który porównuje rekursywnie czy dane dwa katalogi posiadają tą samą zawartość. Napisz program, który tworzy listing podobny do wygenerowanego poleceniem "`find mój_katalog -not -type d -ls`". Niech każda linia wyjścia zawiera ścieżkę pliku (bez prefiksu `mój_katalog`), wielkość pliku i uprawnienia w formie ósemkowej. Katalogi skanuj funkcją `readdir(3)`, a właściwości plików funkcją `stat(3)`.

Zadanie 10

Zbadaj wydajność trzech mechanizmów obsługi operacji na plikach: `write(2)`, `stdio.h(7posix)`, `writev(2)`. Na standardowe wyjście drukuj wiele trójkątów prostokątnych równoramiennych złożonych z gwiazdek (`*`) – niech jedno z ramion zawsze znajduje się w pierwszej kolumnie. Z argumentów programu `argv` odczytaj wartość `N` oznaczającą ilość linii do wydrukowania na standardowe wyjście, oraz `L` – długość przyprostokątnej. Do mierzenia wydajności użyj polecenia `time(1)`. Aby wyeliminować relatywną powolność terminala przekieruj standardowe wyjście do `/dev/null`. Zauważ, że `writev(2)` zapisuje do `IOV_MAX` bloków na raz. Czy odpowiednia zmiana buforowania `stdout` przy pomocy `setvbuf(3)` jest w stanie znacząco poprawić wynik? Do diagnostyki wydajności użyj polecenia "`strace -c`".

[3] W kodzie obsługi sygnału wolno korzystać wyłącznie z funkcji wielobieżnych! Tj. nie z `fprintf` i podobnych.