Systemy operacyjne

Pracownia 1

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 w nawiasach nazwy wywołań bibliotecznych, których należy użyć. Proszę najpierw korzystać z podręcznika systemowego (polecenia `man` i `apropos`), a dopiero potem szukać w Internecie. W systemie opartym na pakietach debianowych (np. Ubuntu) należy zainstalować pakiety `manpages-dev` i `manpages-posix-dev`. Głównym podręcznikiem do zajęć praktycznych jest The Linux Programming Interface. Po szczegóły można sięgać do Advanced Programming in the UNIX Environment.

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.

Zadanie 1

Napisz program, który wyświetli **numer procesu** (`getpid`), **rodzica** (`getppid`), **sesji** (`getsid`) oraz **grupy procesów** do której przynależy (`getpgrp`). W osobnej konsoli wywołaj polecenie `ps` z odpowiednimi argumentami i pokaż, że Twój program drukuje poprawne dane. 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

Napisz program, który będzie prezentował jak zasoby przenoszą się do procesu potomnego:

a. Wydrukuj **środowisko programu** (`getenv`) i **bieżący katalog roboczy** (`getcwd`), a następnie utwórz proces potomny. Jeśli zmienisz w rodzicu środowisko (`setenv`) lub bieżący katalog (`chdir`), to potomek będzie widział te zmiany?
b. W procesie nadrzędnym otwórz plik do odczytu (`open`). Czy zamknięcie pliku (`close`) w procesie rodzica zamyka także plik w procesie potomnym? Czy odczyt z pliku (`read`) zmienia pozycję **kursora** (`lseek`) w drugim procesie?

Zadanie 3

Utwórz program symulujący wywołanie `system` z tą różnicą, że ma nie korzystać z pośredniego procesu powłoki. Przeczytaj argumenty linii poleceń (`argv`) i przekieruj je do wywołania `execve` w procesie potomnym. Przyjmij, że ścieżka do programu jest **ścieżką absolutną**. Poczekaj na zakończenie procesu potomnego (`wait`) i pobierz jego kod wyjścia. Jeśli proces zakończył się w wyniku otrzymania sygnału – wydrukuj jego numer i nazwę.

Zadanie 4

Napisz program, który wygeneruje błąd dostępu do pamięci celem otrzymania sygnału `SIGSEGV`. Obsłuż sygnał (`sigaction`), zinterpretuj dane z nim związane zawarte w drugim (`siginfo_t`) i trzecim argumencie (`ucontext_t`) procedury obsługi sygnału. Wypisz na `stderr` komunikat zawierający informacje o adresie wywołującym błąd (`si_addr`), typie błędu (`si_code`), adresie wierzchołka stosu i adresie instrukcji powodującej błąd (`uc_mcontext`), następnie wydrukuj ślad programu (`backtrace`) po czym zakończ jego działanie. Przetestuj co najmniej dwa warianty usterki – (a) odczyt z niezmapowanej pamięci (b) zapis do pamięci tylko do odczytu. **UWAGA!** W kodzie obsługi sygnału **należy** korzystać wyłącznie z funkcji, które są wielobieżne! (`printf` **nie jest** wielobieżna, dlaczego?)

Zadanie 5

Zaimplementuj odpowiedniki wywołań bibliotecznych `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 następnie 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 6

Utwórz bibliotekę współdzieloną składającą się z kilku modułów – w każdym z nich umieść przynajmniej jedną funkcję. Kod modułów musi być **relokowalny** – tj. przekaż do kompilatora opcję `-fPIC` (ang. *Position Independent Code*). 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.

Zadanie 7

W tym zadaniu zobaczymy jak działa **lokalna pamięć wątków** (ang. *thread local storage*). Utwórz kilka wątków (`pthread_create`) i zsynchronizuj je przy pomocy bariery (`pthread_barrier_wait`). Każdemu wątkowi nadaj unikalny identyfikator i wpisz go do prywatnej zmiennej globalnej (`__thread`). Znów zsynchronizuj wątki po czym odczytaj w każdym swój identyfikator oraz identyfikator systemowy (`pthread_self`). Następnie zaczekaj losową ilość sekund i zakończ działanie wątku. W międzyczasie z użyciem programu `ps` pokaż, że istotnie w Twoim procesie funkcjonuje więcej niż jeden wątek. Upewnij się, że **wątek główny** nie zakończy swojego działania, zanim wątki potomne się nie zakończą (`pthread_join`). Czym charakteryzują są **wątki odczepione** (ang. *detached*)?