Systemy operacyjne

Lista programistyczna nr 1

Termin oddawania 23 listopada 2018

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.

Należy użyć szkieletów rozwiązań udostępnionych na stronie przedmiotu. W komentarzu na początku pliku należy wpisać swoje imię, nazwisko oraz numer indeksu, jednocześnie oświadczając, że jest się jedynym autorem kodu źródłowego. Należy odpowiedzieć na wszystkie pytania zadane w komentarzu – będziemy je czytać i uwzględniać przy przydzielaniu punktów. Ocena poprawności rozwiązań nie może wymagać od sprawdzającego czynności innych niż uruchomienie programu! Na wyjście programu należy wydrukować komunikaty niezbędne do przekonania sprawdzającego, że program robi to co powinien.

Zadanie 1 (1). Napisz program, który utworzy proces **zombie**. Należy wskazać nieumarły proces potomny uruchamiając polecenie «`ps`» z użyciem wywołań **fork(2)** i **execve(2)**, a następnie zakończyć cały program. Jeśli użytkownik przekaże parametr «`--bury`» do programu, to należy zapobiec powstawaniu zombie, tj. zignorować sygnał `SIGCHLD` z użyciem **sigaction(2)**.

Zadanie 2 (1+1). Napisz program, który utworzy proces **sierotę**. Niech proces główny przyjmie rolę **żniwiarza** (ang. *reaper*) przy użyciu **prctl(2)**. Używając wywołania **fork(2)** utwórz kolejno syna i wnuka. Następnie osieroć wnuka zabijając syna. Uruchom polecenie «`ps`», aby wskazać kto przygarnął sierotę. Punkt **bonusowy** zostanie przydzielony tylko wtedy, gdy program będzie wolny od potencjalnych wyścigów. Prawidłowe rozwiązanie nie może polegać na arbitralnie dobranych przez programistę opóźnieniach.

Wskazówka: Rozważ użycie grup procesów, przeczytaj podręczniki **setpgrp(2)**, **waitpid(2)**, **killpg(2)**.

Zadanie 3 (2). Korzystając z rodziny procedur **makecontext(3)** utwórz **współprogramy**[1] (ang. *coroutines*) realizujące następujące funkcje:

1. Pobierz z «`stdin`» słowo oddzielone dowolną liczbą białych znaków **isspace(3)**. Zliczaj ilość pobranych słów. Przełącz na #2.
2. Zlicz i usuń ze słowa znaki niebędące znakami alfanumerycznymi **isalnum(3)**. Przełącz na #3.
3. Wydrukuj słowa oddzielone spacją na «`stdout`». Zliczaj znaki w słowach. Przełącz na #1.

Kiedy współprogram #1 natrafi na koniec pliku pozostałe współprogramy mają wydrukować wartość liczników, po czym proces ma zakończyć swe działanie. Stan programu możesz przechowywać w zmiennych globalnych – nie musisz martwić się synchronizacją. Możesz założyć, że na wejściu pojawiają się tylko znaki `ASCII` i żadne słowo nie jest dłuższe niż 255 bajtów. Dane należy wczytywać i zapisywać odpowiednio wywołaniami **read(2)** i **write(2)**. Zawartość liczników należy wydrukować na «`stderr`» procedurą **fprintf(2)**.

[1] Najbardziej prymitywna postać wątków przestrzeni użytkownika, które omówiliśmy na ćwiczeniach.

Zadanie 4 (2). Zaprogramuj poprzednie zadanie z użyciem procesów. Proces główny będzie odpowiedzialny wyłącznie za utworzenie procesów realizujących funkcje z poprzedniego zadania. Procesy potomne będą komunikować się z użyciem potoków **pipe(2)** i wywołań **read(2)** oraz **write(2)**. Proces główny ma przekazać otwarte potoki do dzieci i czekać ich zakończenie z użyciem **wait(2)**. Gdy proces potomny zakończy swe działanie należy zinterpretować jego kod wyjścia – tj. czy zakończył się normalnie, czy w wyniku otrzymania sygnału. Nie dopuszczamy, żeby rodzic zabijał swoje dzieci – podprocesy mają kończyć swoje działanie bez udziału rodzica. Uwzględnij zachowanie potoków **pipe(7)**, gdy jeden z końców zostanie zamknięty.

UWAGA! Pozostawienie w systemie nieaktywnych procesów jest poważną usterką programie.

Zadanie 5 (2). Napisz program, który wygeneruje błąd odwołania do pamięci. Obsłuż sygnał `SIGSEGV` z pomocą **sigaction(2)**. Zinterpretuj dane zawarte w drugim «`siginfo_t`» i trzecim argumencie «`ucontext_t`» procedury obsługi sygnału. Wypisz na «`stderr`» komunikat zawierający informacje o:

- adresie powodującym błąd odwołania «`si_addr`»,
- typie błędu «`si_code`» w postaci czytelnej dla człowieka, tj. `MAPERR` lub `ACCERR`,
- adresie wierzchołka stosu i adresie instrukcji powodującej błąd «`uc_mcontext`».

... i wydrukuj **ślad wywołań** procedurą **backtrace_symbols_fd(3)**, po czym zakończ działanie programu.

Twój program ma przetestować następujące usterki w zależności od podanego parametru:

- «`--maperr`» odczyt z nieodwzorowanej pamięci,
- «`--accerr`» zapis do pamięci tylko do odczytu.

Wygenerowanie powyższych usterek nie może używać adresów arbitralnie wybranych przez programistę. Obszar adresów o pożądanych właściwościach należy utworzyć z użyciem funkcji **mmap(2)** i **unmap(2)**.

UWAGA! Użycie funkcji, która nie jest wielobieżna, w procedurze obsługi sygnału jest poważną usterką w programie.

Zadanie 6 (2). Utwórz **bibliotekę współdzieloną** składającą się z **jednostek translacji** implementujących poniższe procedury. Kod modułów musi być skompilowany z opcją «`-fPIC`» (ang. *Position Independent Code*). Biblioteka musi być skonsolidowana z opcją «`-shared`».

- `int strdrop(char *str, const char *set)`:
  usuwa (w miejscu) z ciągu `str` znaki występujące w `set` i zwracającą nową długość ciągu
- `int strcnt(const char *str, const char *set)`:
  zwraca ilość znaków z ciągu `set` występujących w `str`

Napisz program testujący procedury z biblioteki ładowanej **w trakcie wykonania** (ang. *run-time linking*) programu. Wykorzystaj funkcję **dlopen(3)** do **wyłuskania** procedur z biblioteki «`libmystr.so`». Uruchom program «`pmap`» przed i po wywołaniu procedury «`dlopen`», aby wskazać przedziały adresów, w które konsolidator odwzorował sekcje biblioteki.

Niestety po przeniesieniu pliku wykonywalnego i biblioteki do innego katalogu, konsolidator dynamiczny **ld.so(8)** nie będzie w stanie uruchomić programu. Napraw to używając symbolu «`$ORIGIN`». Zmodyfikuj odpowiednio parametry przekazywane do konsolidatora w pliku «`Makefile`».