Systemy operacyjne 2016

Lista programistyczna nr 1

10 listopada 2016

Studenci są zachęcani do przeprowadzania dodatkowych eksperymentów związanych z treścią zadań i dzieleniem się obserwacjami z resztą grupy. Proszę najpierw korzystać z podręcznika systemowego (polecenia `man` i `apropos`) i w ostateczności sięgać do zasobów Internetu. 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 resztę książki czytać w razie potrzeby. Bardziej wnikliwe wyjaśnienia zagadnień można odnaleźć w książce „Advanced Programming in the UNIX Environment".

Rozwiązania mają być napisane w języku `C` (a nie `C++`). Kompilować się bez błędów i ostrzeżeń (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 zajęć.

UWAGA! W trakcie prezentacji programów należy wyjaśnić pojęcia, które zostały oznaczone **wytłuszczoną** czcionką. Brak zrozumienia używanych funkcji systemowych może spowodować nieprzydzielenie punktów za zadanie.

Zadanie 1. Zaprezentuj jak powstają **procesy zombie** i **sieroty**.

(a) Z procesu nadrzędnego wykonaj polecenie `ps` (używając **fork(2)** i **execve(2)**) celem wskazania nieumarłego procesu potomnego. Następnie pokaż, jak zapobiegać powstawaniu zombie – zignoruj sygnał `SIGCHLD` z użyciem **sigaction(2)**. Wariant ma być wybieralny z linii poleceń.

(b) Wydrukuj numer procesu **żniwiarza** (ang. *process reaper*) przy użyciu **prctl(2)**. Utwórz proces potomny po czym zakończ działanie głównego procesu. Wskaż poleceniem, kto stał się nowym rodzicem procesu sieroty.

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

1. Pobierz z `stdin` słowo oddzielone spacjami. Zliczaj ilość pobranych słów. Przełącz na `#2`.
2. Usuń ze słowa znaki niebędące znakami alfanumerycznymi **isalnum(3)**. Przełącz na `#3`.
3. Wydrukuj słowo. Zliczaj znaki w słowach. Przełącz na `#1`.

Gdy współprogram `#1` wczyta puste słowo współprogramy mają wydrukować wartość liczników, po czym proces ma zakończyć swe działanie. Stan programu możesz przechowywać w zmiennych globalnych – nie musisz się martwić synchronizacją, gdyż w pełni kontrolujesz moment przełączania kontekstu (ang. *cooperative multitasking*).

[1] przypominające mocno wątki przestrzeni użytkownika omówione na ćwiczeniach

Zadanie 3. Zaprogramuj poprzednie zadanie z użyciem procesów. Proces główny będzie odpowiedzialny za utworzenie trzech procesów realizujących funkcje z poprzedniego zadania. Komunikację zrealizuj przy pomocy potoków utworzonych z pomocą **pipe(2)** i wywołań **read(2)** oraz **write(2)**. Proces główny ma czekać na dzieci z użyciem **wait(2)**. Podprocesy mają kończyć swoje działanie bez udziału rodzica – przeczytaj o zachowaniu potoków **pipe(7)**, gdy jeden z końców zostanie zamknięty.

Zadanie 4. 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`),
- adresie wierzchołka stosu i adresie instrukcji powodującej błąd (`uc_mcontext`).

... i wydrukuj **ślad wywołań** procedurą **backtrace(3)**, po czym zakończ działanie programu. Przetestuj dwie usterki: odczyt z niezmapowanej pamięci i zapis do pamięci tylko do odczytu.

UWAGA! W procedurze obsługi sygnału należy korzystać wyłącznie z funkcji wielobieżnych[2].

Zadanie 5. Utwórz **bibliotekę współdzieloną** składającą się z dwóch **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. W pierwszym wariancie konsolidacja ma przebiegać w trakcie **ładowania programu** (ang. *load-time linking*), a w drugim **w trakcie wykonania** (ang. *run-time linking*). W drugim przypadku skorzystaj z **dlopen(3)** do **wyłuskania** procedury.

Pamiętaj, że program ładujący musi znać ścieżki poszukiwania bibliotek (`rpath`) – sekcję `.dynamic` pliku `ELF` wyświetl poleceniem `readelf -a`. Przed i po załadowaniu biblioteki wskaż programem `pmap` miejsce w przestrzeni adresowej procesu, gdzie konsolidator umieścił bibliotekę.

Zadanie 6. Utwórz program usługowy (ang. *daemon*) posługując się przykładem z „Advanced Programming in the UNIX Environment" §13. Nową sesję należy utworzyć funkcją **setsid(2)**, co zagwarantuje, że proces stanie się **przywódcą sesji** (ang. *session leader*), **przywódcą grupy** (ang. *process group leader*) i odłączy się od **terminala kontrolnego** (ang. *controlling terminal*).

Demon ma zliczać ilość otrzymanych sygnałów `SIGUSR1` do momentu odebrania `SIGTERM`. Licznik ma być resetowany po otrzymaniu sygnału `SIGHUP`. Każde odebranie sygnału wraz z wartością licznika należy zapisać do dziennika systemowego procedurą **syslog(3)**. Działanie demona należy zakończyć opuszczając procedurę `main`.

[2] `printf` nie jest wielobieżna, a `snprintf` jest, dlaczego?