Systemy operacyjne

Zadania programistyczne 3

“Pamięć wirtualna, I/O i systemy plików”

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)`.

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

Zadanie 1

Utwórz bibliotekę dzieloną składającą się z killku modułów, a w każdym z nich zadeklaruj przynajmniej jedną funkcję – muszą różnić się sygnaturami. Kod modułów musi być relokowalny – tj. należy przekazać do kompilatora odpowiednią opcję (`-fPIC`[2]). Pamiętaj, że 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 2 [3 pkt.]

Zaimplementuj prosty menadżer pamięci przestrzeni użytkownika, składający się z czterech funkcji o nastepujących sygnaturach (ich znaczenie jest dokładnie opisane w podręczniku systemowym):

```
void *malloc(size_t size);
void *calloc(size_t count, size_t size);
void *realloc(void *ptr, size_t size);
void  free(void *ptr);
```

Pamięć dla procesu przydzielaj super-blokami (tj. ciągły zbiór stron) przyłączając anonimowe strony (`mmap(2)`, `MAP_ANONYMOUS`). Nieużywane bloki trzymaj na liście dwukierunkowej posortowanej względem adresów. Strukturę węzła listy zapisuj na początku wolnego bloku. Przy alokacji bloku używaj strategii `first-fit`. Przy zwalnianiu bloków gorliwie wykonuj operację scalania. Udostępnij procedurę do drukowania stanu zarządzanej pamięci (np. lista wszystkich bloków). Super-blok należy zwrócić do systemu, jeśli zawiera wyłącznie jeden blok, który jest wolny.

Następnie skompiluj swój menadżer jako bibliotekę dzieloną. Używając sztuczki ze zmienną środowiskową `LD_PRELOAD` (`ld.so`) przesłoń symbole wyżej wymienionych funkcji ze standardowej biblioteki i pokaż, że proste programy są w stanie korzystać z Twojego menadżera.

Zadanie 3

Napisz program, który podczepi sobie kilka stron pamięci anonimowej. Zapisz do nich trochę danych, a nastepnie zmień uprawnienia stron na tylko do odczytu (`mprotect(2)`). Spróbuj powtórzyć operacje zapisu – dostaniesz sygnał `SIGSEGV`. Przechwyć go (`sigaction`), zinterpretuj dane z nim związane (struktura `siginfo_t`) i wypisz[3] komunikat zawierający co najmniej pola `si_addr` i `si_code` na `stderr` , po czym zakończ działanie programu.

Zadanie 4

Praktycznie wszystkie zasoby w systemach uniksowych mają semantykę pliku. Jest to wygodne, ale uniemożliwia wyrażenie pewnych operacji i dlatego powstała 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)`.

---

[2] Position Independent Code

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

Zadanie 5

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 6

Zbadaj wydajność trzech mechanizmów obsługi operacji na plikach: `write(2)`, `stdio.h(7posix)`, `writev(2)`. Na standardowe wyjście drukuj trójkąty prostokątne równoramienne złożone z gwiazdek (`*`) – niech jedno z ramion 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`".

Zadanie 7

Napisz program, który obserwuje zdarzenia na plikach w podanym katalogu. Wykorzystaj do tego mechanizm `inotify(7)`. Obserwowane zdarzenia drukuj na standardowe wyjście. Następnie napisz program testujący ten mechanizm, a w szczególności działanie flag: `IN_ACCESS`, `IN_ATTRIB`, `IN_CLOSE_WRITE`, `IN_CLOSE_NOWRITE`, `IN_CREATE`, `IN_MODIFY`, `IN_DELETE`, `IN_DELETE_SELF`, `IN_MOVE_SELF`, `IN_MOVED_FROM`, `IN_MOVED_TO` oraz `IN_OPEN`. Wykorzystaj w tym celu następujące funkcje: `chmod(2)`, `utime(2)`, `open(2)`, `read(2)`, `write(2)`, `close(2)`, `mkdir(2)`, `rmdir(2)`, `unlink(2)`, `rename(2)`.

Zadanie 8

Zaprezentuj w jaki sposób system operacyjny podczepia strony do plikowego mapowania pamięci w zależności o żądań i wskazówek udzielanych przez programistę. Wielkość strony pobierz funkcją `getpagesize(2)`. Będzie potrzebny plik wypełniony zerami wielkości kilkuset (`512`?) stron (`creat(2)`, `ftruncate(3)`). Podczep go do przestrzeni adresowej (`mmap(2)`) i przy pomocy bitmapy pokaż, które strony pliku znajdują się w pamięci (`mincore(2)`). Następnie zapisując dane do poszczególnych stron, pokaż efekt działania funkcji `madvise(2)` z parametrami `MADV_RANDOM`, `MADV_WILLNEED`, `MADV_DONTNEED`, `MADV_SEQUENTIAL`. Pokaż, że stron przypiętych do pamięci fizycznej (`mlock(2)`) nie da się usunąć. Wychodząc z programu posprzątaj zasoby i skasuj plik.