Systemy operacyjne

Lista zadań nr 7

Na zajęcia 26 listopada 2020

Należy przygotować się do zajęć czytając następujące rozdziały książek:

- Tanenbaum (wydanie czwarte): 3.3, 3.5, 10.7
- Linux Programming Interface: 38, 49
- APUE (wydanie trzecie): 8.11

UWAGA! W trakcie prezentacji należy być gotowym do zdefiniowania pojęć oznaczonych **wytłuszczoną** czcionką.

Zadanie 1. Próba otworzenia `open(2)` pliku wykonywalnego do zapisu, kiedy ten plik jest załadowany i wykonywany w jakimś procesie, zawiedzie z błędem «ETXTBSY». Podobnie, nie możemy załadować do przestrzeni adresowej `execve(2)` pliku, który jest otwarty do zapisu. Co złego mogłoby się stać w systemach implementujących **stronicowanie na żądanie** (ang. *demand paging*), gdyby system operacyjny pozwolił modyfikować plik wykonywalny, który jest uruchomiony?

Zadanie 2. Na podstawie §49.1 wyjaśnij słuchaczom różnicę między **odwzorowaniami plików w pamięć** (ang. *memory-mapped files*) i **odwzorowaniami pamięci anonimowej** (ang. *anonymous mappings*). Jaką zawartością wypełniana jest pamięć wirtualna należąca do tychże odwzorowań? Czym różni się odwzorowanie **prywatne** od **dzielonego**? W jakim celu odwzorowania prywatne wykorzystują technikę **kopiowania przy zapisie**? Czemu odwzorowanie prywatne plików urządzeń blokowych ma niewiele sensu?

Wskazówka: Urządzenie blokowe to na przykład: pamięć karty graficznej albo partycja dysku.

Zadanie 3. Na podstawie opisu do tabeli 49–1 podaj scenariusze użycia prywatnych i dzielonych odwzorowań plików w pamięć albo pamięci anonimowej. Pokaż jak je utworzyć z użyciem wywołania `mmap(2)`. Co się z nimi dzieje po wywołaniu `fork(2)`? Czy wywołanie `execve(2)` tworzy odwzorowania prywatne czy dzielone? Jeśli wystartujemy n instancji danego programu to ile razy jądro załaduje jego plik `ELF` do pamięci? Które z wymienionych odwzorowań mogą wymagać użycia **pamięci wymiany** (ang. *swap space*) i kiedy?

Zadanie 4. Na podstawie slajdów do wykładu wyjaśnij w jaki sposób system Linux obsługuje błąd stronicowania. Kiedy jądro wyśle procesowi sygnał `SIGSEGV` z kodem «SEGV_MAPERR» lub «SEGV_ACCERR»? W jakiej sytuacji wystąpi **pomniejsza usterka strony** (ang. *minor page fault*) lub **poważna usterka strony** (ang. *major page fault*)? Jaką rolę pełni **bufor stron** (ang. *page cache*)? Kiedy wystąpi błąd strony przy zapisie, mimo że pole «vm_prot» pozwala na zapis do **obiektu wspierającego** (ang. *backing object*)? Kiedy jądro wyśle `SIGBUS` do procesu posiadającego odwzorowanie pliku w pamięć (§49.4.3)?

Ściągnij ze strony przedmiotu archiwum «so20_lista_7.tar.gz», następnie rozpakuj i zapoznaj się z dostarczonymi plikami.

UWAGA! Można modyfikować tylko te fragmenty programów, które zostały oznaczone w komentarzu napisem «TODO». Pamiętaj o użyciu odpowiednich funkcji opakowujących (ang. *wrapper*) z biblioteki «libcsapp».

Zadanie 5. Program «forksort» wypełnia tablicę 2^{26} elementów typu «long» losowymi wartościami. Następnie na tej tablicy uruchamia hybrydowy algorytm sortowania, po czym sprawdza jeden z warunków poprawności wyniku sortowania. Zastosowano algorytm sortowania szybkiego (ang. *quick sort*), który przełącza się na sortowanie przez wstawianie dla tablic o rozmiarze mniejszym niż «INSERTSORT_MAX».

Twoim zadaniem jest taka modyfikacja programu «forksort», żeby oddelegować zadanie sortowania fragmentów tablicy do podprocesów. Przy czym należy tworzyć podprocesy tylko, jeśli rozmiar nieposortowanej części tablicy jest nie mniejszy niż «FORKSORT_MIN». Zauważ, że tablica elementów musi być współdzielona między procesy – użyj wywołania `mmap(2)` z odpowiednimi argumentami.

Porównaj **zużycie procesora** (ang. *CPU time*) i **czas przebywania w systemie** (ang. *turnaround time*) przed i po wprowadzeniu delegacji zadań do podprocesów. Na podstawie prawa Amdahla[1] wyjaśnij zaobserwowane różnice. Których elementów naszego algorytmu nie da się wykonywać równolegle?

Zadanie 6. (Pomysłodawcą zadania jest Piotr Polesiuk.)

Nasz serwis internetowy stał się celem ataku hakerów, którzy wykradli dane milionów użytkowników. Zostaliśmy zmuszeni do zresetowania haseł naszych klientów. Nie możemy jednak dopuścić do tego, by użytkownicy wybrali nowe hasła z listy, którą posiadają hakerzy. Listę pierwszych 10 milionów skompromitowanych haseł można pobrać poleceniem «`make download`».

Program «`hashdb`» został napisany w celu utworzenia bazy danych haseł i jej szybkiego przeszukiwania. Pierwszym argumentem przyjmowanym z linii poleceń jest nazwa pliku bazy danych haseł. Program wczytuje ze standardowego wejścia hasła oddzielone znakami końca linii i działa w dwóch trybach: dodawania haseł do bazy (opcja «`-i`») i wyszukiwania (opcja «`-q`»). Żeby utworzyć bazę danych z pliku zawierającego hasła należy wywołać polecenie «`./hashdb -i badpw.db < passwords.txt`». Program można uruchomić w trybie interaktywnego odpytywania bazy danych: «`./hashdb -q badpw.db`».

Implementacja wykorzystuje tablicę mieszającą przechowywaną w pamięci, która odwzorowuje plik bazy danych haseł. Używamy adresowania liniowego i funkcji mieszającej Jenkinsa[2] «`lookup3.c`». Hasło może mieć maksymalnie «`ENT_LENGTH`» znaków. Baza danych ma miejsce na 2^k wpisów. Jeśli w trakcie wkładania hasła do bazy wykryjemy konflikt kluczy, to wywołujemy procedurę «`db_rehash`». Tworzy ona na nową bazę o rozmiarze 2^{k+1} wpisów, kopiuje klucze ze starej bazy do nowej i atomowo zastępuje stary plik bazy danych.

Twoim zadaniem jest uzupełnić kod procedur «`db_open`», «`db_rehash`» i «`doit`» zgodnie z poleceniami zawartymi w komentarzach. Przeczytaj podręcznik systemowy do wywołania systemowego `madvise(2)` i wyjaśnij słuchaczom co ono robi. Należy użyć odpowiednich funkcji z biblioteki «`libcsapp`» opakowujących wywołania: `unlink(2)`, `mmap(2)`, `munmap(2)`, `madvise(2)`, `ftruncate(2)`, `rename(2)` i `fstat(2)`.

Ściągnij repozytorium `so-shell`[3] i zapoznaj się z jego zawartością, w szczególności z plikami «`shell.c`» i «`jobs.c`».

UWAGA! Można modyfikować tylko te fragmenty projektu, które zostały oznaczone w komentarzu napisem «`TODO`».

Zadanie 7. Wykonywanie zewnętrznego polecenia przez powłokę uniksową ma dwa warianty. Gdy w ścieżce do pliku występuje znak ukośnika «/» to zakładamy, że użytkownik podał ścieżkę względną i uruchamiamy `execve(2)` bezpośrednio. W przeciwnym przypadku musimy odczytać zawartość zmiennej środowiskowej «`PATH`» przechowującej katalogi oddzielone znakiem dwukropka «`:`». Każdą ścieżkę katalogu sklejamy z nazwą polecenia (przyda Ci się procedura «`strapp`» z pliku «`lexer.c`») i wołamy «`execve`». Jeśli polecenie nie zostanie znalezione na dysku to «`execve`» wraca z błędem.

Uzupełnij ciało procedury «`external_command`» z pliku «`command.c`» zgodnie z powyższym opisem.

Wskazówka: Rozwiązanie wzorcowe liczy 10 wierszy. Do przetwarzania ciągów znakowych użyto funkcji `strndup(3)` i `strcspn(3)`.

Zadanie 8. Procedura «`do_job`» w pliku «`shell.c`» przyjmuje tablicę tokenów, zawierającą polecenie do uruchomienia i przekierowania, oraz rodzaj tworzonego zadania (pierwszoplanowe lub drugoplanowe). Najpierw sprawdza czy podane polecenie należy do zestawu poleceń wbudowanych (ang. *builtin command*). Jeśli nie, to przechodzi do utworzenia zadania i wykonania polecenia zewnętrznego. Po utworzeniu podprocesu i nowej grupy procesów, zadanie jest rejestrowane z użyciem «`addjob`» i «`addproc`». Jeśli utworzono zadanie pierwszoplanowe, to należy poczekać na jego zakończenie przy pomocy «`monitorjob`».

Uzupełnij ciało procedury «`do_job`» – wykorzystaj funkcje opakowujące wywołania: `sigprocmask(2)`, `sigsuspend(2)`, `setpgid(2)` i `dup2(2)`. Pamiętaj, że uruchomione zadanie musi prawidłowo reagować na sygnały: «`SIGTSTP`», «`SIGTTIN`» i «`SIGTTOU`», a powłoka musi zamykać niepotrzebne deskryptory plików.

Wskazówka: Rozwiązanie ma około 25 wierszy. Funkcja «`monitorjob`» konfiguruje terminal przy pomocy `tcsetpgrp(3)`.

[1] `https://pl.wikipedia.org/wiki/Prawo_Amdahla`
[2] `https://en.wikipedia.org/wiki/Jenkins_hash_function`