Zajęcia 1: Procesy

2024-10-08

`ps` – wypisuje procesy, `ps ax` wypisuje wszystkie, `ps aux` jeszcze więcej informacji

`fork` tworzy nowy proces i zwraca jego pid (w rodzicu, w dziecku zwraca 0), program po wykonaniu `fork()` wykonuje dalsze instrukcje w obu procesach

`exec` – rodzina funkcji, ładuje inny program w obecny proces, kilka funkcji z różnymi argumentami, np.

- `execvp` – nazwa pliku z programem i argumenty programu, szuka programu w zmiennej `PATH` użytkownika

`wait` – czeka na zmianę stanu procesu dziecka (np. terminated, stopped), w argumencie wskaźnik na strukturę, która ma trzymać, jak zmienił się stan.

Do pisania i czytania służą polecenia `write` i `read`, które czytają/piszą bajty. Pierwszy argument to file id, w szczególności 0 i 1 to zawsze standardowe wejście i wyjście, 2 to stderr. Read zwraca liczbę wczytanych bajtów.

Sposób działania shella: dostaje informację, jaki program na zostać wykonany, forkuje i execuje ten program, rodzic robi wait, żeby potem kontynuować

Zajęcia 2: Pliki

2024-10-09

Syscalle są w bibliotece `unistd.h`

`strace` – śledzi syscalle wykonane przez program

Fork bomb:

```
int main() {
    while (1) fork();
}
```

program namnaża procesy, aż nie zostanie zatrzymany (np. przez brak pamięci albo osiągnięcie maksymalnej liczby procesów).

File Descriptor to unikalna wewnątrz procesu liczba naturalna, identyfikująca otwarty plik. Wartości od 0 do OPEN_MAX. Kernel wstawia odpowiednie wartości dla każdego procesu.

Open File Description to zapis tego jak proces lub procesy korzystają z pliku, każdy file descriptor wskazuje na jakiś open file description. Pamiętany jest offset od początku pliku, status pliku, rodzaj dostępu. Jest to globalna tablica (wspólna dla wszystkich procesów).

`open` – otwiera deskryptor pliku (zwraca go), przyjmuje nazwę, tryb otwarcia – pisanie, czytanie, append, etc., ostatni argument jest opcjonalny, przy tworzeniu pliku określa uprawnienia użytkowników.

`close` – zamyka file descriptor, może się nie udać, gdy plik jest już zamknięty.

`create` – tworzy plik, analogiczne do wywołania open() z odpowiednimi flagami.

Funkcja `read()` może zostać przerwana w trakcie czytania, jeśli nie zdążyła nic odczytać, to zwraca -1, inaczej zwraca liczbę już odczytanych bajtów (czyli tylko części tych, które miał odczytać). W podobny sposób działa `write` – mógł wypisać tylko część przekazanych bajtów.

Zajęcia 3: I/O

2024-10-16

`lseek(int descriptor, off_t offset, int whence)` – przesuwa pozycję w pliku, ostatni argument: sposób przesunięcia, od początku, relatywnie względem aktualnej pozycji, końca pliku (`SEEK_CUT`, `SEEK_CUR`, `SEEK_END`). Można przesunąć pozycję o dużą wartość (większą niż sam plik) i tam coś napisać, wtedy teoretycznie będzie duży (długi, `ls` to pokaże), ale będzie pusty – nie będzie zajmował miejsca, pliki nie muszą być wypełnione w sposób ciągły. Takie niewypełnione miejsca na potrzeby odczytu są wypełnione zerami.

`du` – stwierdza, ile plik faktycznie zajmuje (a nie jak bardzo się rozciąga).

`pipe` – syscall, który oczekuje tablicy dwóch file descriptorów (intów), system (kernel) tworzy dwa file descriptory (i wsadza je do tablicy z argumentu), z których jeden jest podłączony do wejścia pipe'a, a drugi do wyjścia i pipe przekazuje całe wejście na wyjście. Pipe jest tworzony w jednym procesie, ale służy do komunikacji między procesami – umożliwia komunikację między spokrewnionymi procesami.

Wszystkie procesy (przynajmniej w Linuxie) są ze sobą spokrewnione, na samej górze drzewa pokrewieństwa jest proces `init` o id 1.

Podczas forkowania kopiowana jest tabela file descriptorów (w tym tych z pipe'a) wraz z całą pamięcią, więc proces dziecko może się komunikować przez pipe'a z rodzicem (i na odwrót).

Pipe'ów (i innych struktur systemowych) nie trzeba zwalniać, system sam to zrobi, jak nikt nie będzie miał do nich wskaźnika.

Do komunikacji między procesami bez pokrewieństwa (żeby nie trzeba było przekazywać pipe'a przez rodziców, którzy mogą np. należeć do różnych użytkowników) używamy named pipe – jakieś pliki w systemie, do których można się podpinać jak do pipe'a.

`int mknod(path, mode, dev)` – tworzy specjalny plik pod ścieżką `path` (musi wykonać superuser, chyba że tworzymy obiekt fifo).

`mkfifo(path,mode)` – skrót od `mknod(path, (mode & 0777) | S_FIFO, 0)`, to tworzy plik, za pomocą którego można się komunikować.

Wiele równoległych czytań z jednego pipe'a daje unspecified behaviour, read'y nie są atomowe, bo nie da się tego wymusić w terminalach, dlatego nie jest wymuszane nigdzie.

Z pisaniem jest lepiej: paczki danych o wielkości do `PIPE_BUF` nie będą przeplatane, ale większe mogą być dowolnie.

`fcntl(int fd, int cmd, [data])` – file descriptor control functions, w nagłówku `fcntl.h`, zmienia sposób pracy z deskryptorem, różne komendy:

- `F_DUPFR` – duplikacja deskryptora, tworzy nowy do tego samego w pierwszym wolnym deskryptorze od tego podanego jako kolejny argument: `fcntl(fd1, F_DUPFR, fd2)`.
- `F_GETFD` – pobiera flagi deskryptora, zwraca int z flagami, zapalamy flagi ORując z odpowiednim makrem.
- `F_SETFD` – ustawia flagi deskryptora na podaną wartość (pobieramy aktualne flagi, zmieniamy i ustawiamy tą komendą)

Ważne flagi:

- `O_NONBLOCK` – ustawia odczyt na nieblokujący, czyli jeśli na wejściu nie ma jeszcze danych, to na nie nie czeka, tylko ustawia `errno` na `EAGAIN`
- `FD_CLOEXEC` – zamyka file descriptor na wykonaniu execa

Podczas czytania z pipe'a, w którym nic nie ma, program czeka, jeśli jest szansa, że coś tam zostanie wpisane (czyli jeśli jakiś proces ma otwarty file descriptor do pisania do pipe'a). W przeciwnym wypadku zachowuje się jakby dotarł do końca pliku.

Jeśli dwa procesy otwierają named pipe'a i zaczynają czytać (pisać), to może być tak, że zaczniemy czytać (pisać) zanim drugi proces otworzy tego pipe'a. Przez to można dostać informację o końcu pliku (w przypadku czytania) zanim drugi proces zacznie pisać lub analogicznie błąd spowodowany tym, że nikt nie czyta tego, co piszemy. Aby temu zapobiec, defaultowo otwieranie named pipe'ów jest blokujące (proces czeka, aż ktoś będzie po drugiej stronie) – dlatego otwieranie named pipe'a przez dwa procesy po prostu działa i nie trzeba się tym przejmować. Czasem nie chcemy takiego zachowania, do tego służy flaga `O_ONOBLOCK` syscalla `open`.

Zajęcia 4: Sygnały

2024-10-23

`ioctl` – podobnie do `fcntl`, tylko kontroluje urządzenia I/O

Advisory record locking – rozwiązanie współbieżnej pracy z plikami, locki na fragmentach plików. Mamy locki do pisania i czytania, fragmenty zarezerwowane do pisania mogą być zajęte przez tylko jeden proces, fragmenty do czytania mogą się ze sobą pokrywać, ale nie z fragmentami do pisania. Do obsługi tego używa się `fcntl(fd, flaga, struct flock* lkp)`, gdzie struktura `lkp` zawiera przekazywane przez nas informacje, możliwe flagi to:

- `F_SETLK` – rejestruje blokadę, jeśli się nie uda, do zwraca błąd
- `F_SETLKW` – też rejestruje blokadę, ale blokująca, czeka, aż dostanie
- `F_GETLK` – sprawdza, czy ktoś zablokował dany fragment pliku

Wygląd struktury `flock`:

```
struct flock {
    short l_type; // F_RDLCK, F_WRLCK, R_UNLCK
    short l_whence; // typy jak przy lseek'u, od kiedy liczyć offset
    off_t l_start; // offset do początku segmentu
    off_t l_len; // długość segmentu
    off_t l_pid; // id procesu trzymającego locka
}
```

Wypełniamy niektóre, zależnie od flagi. System wypełni pozostałe.

System nie powstrzyma nas przed pisaniem i czytaniem z plików, na których są locki. Blokowanie ma na celu umożliwienie nam współpracy z innymi procesami, ale nie trzeba tego robić.

Sygnał to informacja o asynchronicznym zdarzeniu/błędzie. Może to być np. `SIGINT`, który został wysłany po kliknięciu `Ctrl+C` – program w losowym momencie dostał informacje, żeby się skończyć. Podobnie działa to przy błędach – jeśli program każe procesorowi podzielić przez 0, on generuje wyjątek, a system może wysłać sygnał `SIGFPE`.

Źródłem sygnałów może być np. terminal (`SIGINT` po `Ctrl+c`, `SIGQUIT` po `Ctrl+\`), hardware (dzielenie przez zero, niewłaściwe odwołanie do pamięci `SIGSEGV`), proces (syscall `kill`), system (`SIGALARM` albo `SIGPIPE` przy zepsutym pipe'ie).

Program dostaje informację o sygnałach i może je obsłużyć w dowolny sposób. Sygnały, które nie docierają do adresata to `SIGKILL` i `SIGSTOP` – one mają określone działanie zawsze.

Możliwe sygnały:

- `SIGCHLD` – informacja dla rodzica o tym, że jego dziecko się skończyło
- `SIGINT` – informacja, że proces powinien się skończyć
- `SIGKILL` – kończy proces
- `SIGALARM` – system budzi proces, który na coś czeka

W nagłówku `signal.h` jest syscall `kill(int pid, int sig)`, który wysyła sygnał do procesu. Można wysyłać tylko do procesów tego samego użytkownika (effective user id – wykonywanie programu z uprawnieniami danego użytkownika). Adresat sygnału to proces o zadanym pid lub procesy z tej samej grupy (pid=0), wszystkie procesy (pid=-1), procesy z grupy o id równym |pid| (pid<-1).

Funkcja `signal` przyjmuje numer sygnału i funkcję, która obsługuje ten sygnał (przyjmuje int i zwraca void – typ `void (*func)(int)`) oraz zwraca poprzednią funkcję obsługującą. Wartości `SIG_DFL` oznacza domyślną obsługę sygnału, `SIG_IGN` – sygnał jest ignorowany. Wywołanie funkcji `signal` oznacza tylko obsługę przy najbliższym sygnale, potem zostanie przywrócona domyślna obsługa. Takie zachowanie wynika z tego, że `signal` pochodzi ze standardu ANSI. Raczej się jej nie używa.

W POSIX jest funkcja
`sigaction(sig, const struct sigaction *restrict act, struct sigaction *restrict oact)`,
gdzie struktura zawiera pola:

- `void (*sa_handler)(int)` – funkcja obsługuje wyjątek, dostaje w argumencie numer sygnału, można też tu wstawić domyślne wartości z makr
- `sigset_t sa_mask` – sygnały, które będą zablokowane (nie odbierane) podczas obsługi tego sygnału

- `int sa_flags` – flagi
- `void (*sa_sigaction)(int, siginfo_t *, void *)` – wskaźnik na funkcję łapiącą sygnał, bardziej skomplikowana obsługa sygnałów

Mamy funkcje przyjmujące maskę `sigset_t *` (jest to struktura a nie zwykła maska, bo możliwych sygnałów jest dużo i trzeba radzić sobie inaczej) i zmieniające ją w zadany sposób:

- `int sigempty(sigset_t *set)` – zeruje maskę
- `sigfillset(sigset_t *set)` – wszystkie sygnały
- `sigaddset(sigset_t *set, int signo)` – dodaje sygnał do maski
- `sigdelset(sigset_t *set, int signo)` – usuwa sygnał
- `sigismember(const sigset_t *set, int signo)` – zwraca, czy sygnał jest w masce

Defaultowo obsługiwany właśnie sygnał jest zablokowany na czas obsługi, istnieje flaga, która zmienia to zachowanie.

Syscalle blokujące są przerywane sygnałami. Jeśli `read` i `write` w ogóle nic nie wczytają/wypiszą, to zwracają -1 i ustawiają `errno` na `EINTR`.

Flagi w `sigaction` :

- `SA_SIGINFO` – handlerem nie jest prosta funkcja przyjmująca inta tylko ta bardziej skomplikowana (obie zdefiniowane w strukcie `sigaction`). Funkcja przyjmuje informację o sygnale i kontekst, czyli utworzony przez system stan programu w momencie pojawienia się sygnału (wskaźnik na stos), który zostanie wykorzystany do przywrócenia poprzedniego stanu (w szczególności możemy go zmienić).
- `SA_RESTART` – jeśli syscall blokujący został przerwany, to zostanie wznowiony.
- `SA_NOCLDSTOP` – nie generujemy `SIGCHLD` gdy dzieci się zatrzymują lub są wznawiane.

Ustawienie obsługi `SIGCHLD` na ignorowaną daje undefined behaviour. W Linuxie procesy zombie będą w takiej sytuacji usuwane.

Sygnały nie są kolejkowane, system pamięta, że proces ma obsłużyć dany sygnał, ale nie pamięta, ile razy.

`waitpid` ma opcję `WNOHANG`, która powoduje, że nie czeka na sygnał. Jeśli proces ma jakieś dzieci, ale żadne z nich nie wywołało `SIGCHLD`, to zwracane jest 0.

W handlerach sygnałów najlepiej nie wywoływać żadnych funkcji, ale można te, które są async-signal-safe, czyli dobrze się zachowują z asynchronicznością. Są to na przykład `_exit`, `close`, `kill`, `read`, `write`. Źle działają `malloc`, `exit`, `printf` – wszystkie działają z jakimiś globalnymi danymi, jest szansa, że ktoś inny też to w tym samym czasie robi.

Funkcja `alarm()` – po podanej liczbie sekund system wyśle nam sygnał `SIGALARM`, który defaultowo zabija proces.

W programie sygnały mogą być blokowane. Sygnały zablokowane nie są zapominane, cały czas pamiętamy, że przyszły, ale program nic z tym nie robi. W momencie odblokowania sygnału system od razu nas informuje, że taki sygnał przyszedł.

Funkcja `int sigprocmask()` służy do ustawienia maski blokowanych sygnałów w programie. Przyjmuje wartość `int how` oraz maskę `const sigset_t *restrict set`. `how` przyjmuje opcje:

- `SIG_BLOCK` – zablokuj wszystko z maski
- `SIG_SETMASK` – ustaw maskę
- `SIG_UNBLOCK` – odblokuj wszystko z maski

`sigprocmask()` przyjmuje też trzeci argument, strukturę, do której wsadza poprzednią maskę.

Funkcja `sigsuspend(const sigset_t *sigmask)` blokuje się w oczekiwaniu na masce podanej w argumencie. Dzięki temu możemy zablokować sygnał, na który chcemy czekać później w programie, on

nie pojawi się przed tym czekaniem. Możemy zablokować się na masce z odblokowanym tym sygnałem, dzięki czemu dostaniemy ten sygnał nawet, jeśli przyszedł przed naszym zblokowaniem.

Funkcja `sigpending(sigset_t *set)` ustawia do struktury maskę sygnałów pending, czyli takich, które przyszły do programu (przed blokowaniem).

Syscall `select(int nfds, ..., struct timeval* timeout)` przyjmuje trzy zbiory file descriptorów, które interesują nas w kontekście czytania, pisania lub wypisywania błędów. Blokujemy się, aż któryś z nich będzie gotowy (do pisania, czytania) lub minie ustalony czas. Pierwszy argument to liczba o jeden większa niż największy file descriptor w każdym zbiorze. Select przerwany wydarzeniem lub sygnałem aktualizuje strukturę `timeout` na czas, który pozostał, więc wykonywanie selecta w pętli nie zaczyna się za każdym razem od nowa z tym samym timeoutem. Select zwraca liczbę gotowych plików i ustawia w przekazanych zbiorach te pliki. Syscall `pselect` dodatkowo przyjmuje maskę sygnałów, które będą go przerywać.

Zajęcia 5: Uprawnienia systemowe

2024-11-13

Pliki mają ustawionego właściciela i grupę oraz uprawnienia, które mają właściciel, grupa i wszyscy użytkownicy. Przy próbie jakiegoś działania z plikiem przez użytkownika najpierw sprawdzane jest, czy jest właścicielem i sprawdzane są uprawnienia właściciela. Jeśli nie, to sprawdzane są grupy użytkownika (może mieć ich wiele) – jeśli któraś z jego grup jest grupą pliku, to sprawdzane są jej uprawnienia. W przeciwnym wypadku użytkownik będzie miał uprawnienia takie, jak każdy użytkownik systemu. W niektórych systemach użytkownik ma jedną wybraną aktywną grupę, między którymi się przełącza. Syscall `setgid` ustawia ID grupy pliku.

Syscalle działające z plikami: `access` sprawdza dostęp do pliku, `chmod` ustawia tryb dostępu do pliku, `chown` zmienia właściciela i grupę.

Syscall `unlink` usuwa wpis z katalogu, mając katalog z plikami mamy w nim linki do faktycznych danych, możemy sobie usunąć taki link. Jeśli nikt nie ma linku do pliku, to zostanie on usunięty. Analogicznie `link` tworzy taki link.

Syscall `chroot` zmienia roota – można ustawić korzeń drzewa plików na dany katalog, proces nie będzie wtedy widział niczego co jest wyżej. Może być przydatny do wykonywania programów w ograniczonym środowisku.

Zajęcia 6: Obsługa procesów

2024-11-14

Każdy proces pracuje, jakby miał swoją kopię procesora, której z nikim nie dzieli. Ta kopia jest dużo mniejsza niż cały procesor, ale udostępnia wszystkie jego funkcje. Sam procesor implementuje wiele funkcji, z których część jest dostępna tylko w trybie uprzywilejowanym – dla systemu, nie zwykłych procesów. Umożliwiają one poprawne działanie systemu operacyjnego.

System musi dzielić procesor między procesy, będzie dawał im dostęp do procesora, czasem stwierdzał, że inny proces będzie teraz pracował i przekazuje procesor innemu procesowi. Wtedy tamten proces jest zamrażany. System dzieli procesor na bardzo różne sposoby, zależnie od ilości procesów, ich priorytetów.

Proces jest opisywany przez bardzo dużo różnych informacji. Stan procesu jest opisywany przez stan rejestrów procesora w danym momencie wykonywania programu, program counter (następną instrukcję do wykonania), program status word (maska bitowa, która jest wypełniana przez każdą instrukcję sprawdzającą), który umożliwia wykonywanie instrukcji warunkowych, stack pointer (wskaźnik na stos), process state (czy właśnie się wykonuje, jest gotowy do otrzymania procesora czy jest zablokowany na jakiejś instrukcji).

Do zarządzania procesami pamiętamy exit status i signal status (sygnał, który jest aktualnie obsługiwany), pid, pid rodzica.

Do zarządzania plikami pamiętamy maskę uprawnień, roota, obecny katalog, tabelę file descriptorów, użytkownika i jego grupę.

Jednym ze sposobów zarządzania procesami są przerwania zegarowe – zegar hardware'u co jakiś czas może przerywać proces, wtedy procesor zapisuje obecny stan i wykonuje procedurę obsługi konkretnego przerwania (to, gdzie zapisać i jaki kod wykonać procesor trzyma w rejestrach niedostępnych dla użytkownika). Przerwanie dzieje się natychmiast. Obsługa przerwania to kod schedulera, który decyduje, jaki proces będzie teraz działał. W Minixie zegar przerywa 60 razy na sekundę. Można też nie ustawiać sztywnej wartości, tylko decydować przed każdym przydzieleniem zasobów ile czasu proces powinien dostać.

Scheduler działa też w momencie, gdy procesy same zawieszają się na jakiejś instrukcji. Do tego pojawiają się też przerwania przychodzące od urządzeń (np. klawiatury). Zegar gwarantuje nam, że proces na pewno się przerwie co określony czas. Dzięki temu żaden proces nie zajmie procesora na stałe.

Zmiana kontekstu (procesu działającego na procesorze) jest bardziej kosztowna niż wznowienie procesu – co prawda scheduler korzysta z procesora, więc rejestry i tak trzeba będzie przywrócić, ale pamięć procesu (stos) nie jest zmieniana i nie trzeba jej przywracać przy wznawianiu procesu. Do tego cała pamięć cache przestaje mieć sens – trzeba załadować na nowo.

Wątek to lekki proces – wątki wykonują się wewnątrz jednego procesu, współdzielą pamięć. Wątki mają dzięki temu ułatwioną współpracę, która nie wymaga syscalli. Za to trzeba poprawnie obsługiwać synchronizację. Większość systemów daje możliwości pracy z wątkami, ale nie jest to konieczne – można je samemu zaimplementować wewnątrz jednego procesu. W standardzie C jest praca z wątkami, której implementację możemy wybrać – ta POSIXowa albo osobna implementacja C.

Zajęcia 7: Tryb jądra

2024-11-20

Procesy mogą pracować w trybie użytkownika (user mode) lub jądra (kernel mode). Przejście do trybu jądra poprzez specjalne instrukcje procesora. Wiele procesów może pracować w trybie jądra i tam również występuje współdzielenie. Przerwanie procesu w trybie jądra może być niebezpieczne, dlatego procesy w trybie jądra mogą określać, kiedy systemowi wolno je przerwać. W trybie jądra pracują np. sterowniki urządzeń, scheduler, procesy, które są odpowiedzialne za zapisywanie buforów, zwalnianie stron pamięci i inne podobne rzeczy. Taki kernel (np. linuxowy) nazywamy monolitycznym – mamy jeden codebase, który się wykonuje po stronie jądra, tak samo jak po stronie użytkownika.

Micro kernel (np. w Minixie) to pomysł na zmniejszenie ilości kodu pracującego w trybie uprzywilejowanym. Funkcjonalności systemu są zaimplementowane jako serwery (process manager, file system i inne), z którymi procesy komunikują się za pomocą message passing. Mamy warstwę jądra, która odpowiada za współdzielenie procesów i message passing. Do tego obsługuje też urządzenia – dzięki temu sterowniki mogą pracować w trybie użytkownika i nigdy nie pracują w trybie jądra. W ten sposób nie wpuszczamy losowego kodu do naszego jądra. Mamy procesy użytkownika, pod którymi są serwery. One korzystają ze sterowników. Wszystko to działa w trybie użytkownika i odwołuje się do funkcjonalności implementowanych przez kernel.

Tanenbaum Torvalds debate: dyskusja o Linuxie i tym, że ma monolityczne jądro. Wygrał z kernelem GNU tym, że faktycznie ktoś go napisał. Dlatego powszechnie używa się dziś jąder monolitycznych. Mikrokernele są wykorzystywane tylko w specjalistycznych urządzeniach.

Zajęcia 8: Synchronizacja zasobów

2024-11-27

Race condition to sytuacja, w której wątki na raz pracują z tą samą pamięcią i przez to nie działają poprawnie. Fragment programu, w którym odwołujemy się do pamięci współdzielonej nazywamy sekcją krytyczną. Chcemy, aby w danej chwili co najwyżej jeden wątek był w sekcji krytycznej. Dodatkowo wymagamy, żeby żaden wątek działający poza sekcją krytyczną nie blokował innego, nikt nie czekał w nieskończoność na wejście do sekcji krytycznej i całość działała niezależnie od liczby procesorów i ich szybkości.

Sposobem na zapobieganie race conditions jest wyłączenie przerywania – instrukcja procesora CLI (Clear Interrupt Flag) Wyłączenie przerywania powoduje, że wątek wie, że nie zostanie przerwany (o ile się nie zablokuje) i może to wykorzystać, by zapobiec race condition. Wymaga to jednak uprawnień do

blokowania przerwań. Do tego rozwiązuje to problem tylko, gdy wszystko działa na jednym rdzeniu – w innym wypadku wątek nie musi zostać przerwany, by ktoś inny zaczął pracować na tych samych danych.

Mimo to pomysł jest dobry, gdy chcemy zabezpieczyć tylko lokalne struktury – możemy trzymać wersje naszej struktury odpowiadające różnym rdzeniom (na każdym całkiem inna). Program działający w trybie uprzywilejowanym może sprawdzić, na którym rdzeniu jest. Będzie korzystał z odpowiedniej struktury, a wewnątrz jednego rdzenia ma pewność, że nikt jej nie nadpisze. Co jakiś czas będzie odbywać się synchronizacja struktur.

Inne podejście do synchronizacji to spin locki – wątki współdzielą zmienną lock, która symbolizuje, czy dany obszar krytyczny jest zajęty, czekają w pętli, aż będzie wolne. Oczywiście nie działa to za dobrze. Można też mieć system turowy – zmienna, która mówi, czyja jest tura. Problem jest taki, że muszą teraz pracować na przemian.

Rozwiązanie Petersona – poza turami mamy tablicę trzymającą to, czy proces jest zainteresowany wejściem do danego bloku. Proces ustawia swoje zainteresowanie i turę, gdy wchodzi do obszaru krytycznego. Potem czeka, dopóki drugi wątek jest zainteresowany. Na wyjściu usuwa się zainteresowanie. Tego typu rozwiązanie może nie działać.

Mamy instrukcję procesora `TSL reg, lock` (Test and Set Lock) – wczytuje zawartość pamięci `lock` (początkowo zerową) do rejestru `reg` i zapisuje niezerową wartość pod `lock`. Jeśli po takiej operacji w `reg` będzie 0, to przejęliśmy obszar i możemy w nim pracować.

Czekanie w pętli (busy waiting) powoduje problemy. Na przykład Priority Inversion Problem – mamy wątek ważniejszy i mniej ważny. Jak mniej ważny wątek jest w sekcji krytycznej, to ten ważniejszy czeka, ale w punktu widzenia systemu cały czas pracuje, więc da mu dużo zasobów, które on zmarnuje i będzie czekał jeszcze dłużej, bo mniej ważny wątek będzie miał mniej zasobów.

Klasycznym rozwiązaniem synchronizacji są semafory. Są to zmienne, które można inkrementować i dekrementować, przy czym dekrementacja poniżej 0 jest blokująca – czeka, aż ktoś zwiększy semafor, żeby nie przyjął po drodze wartości 0.

Monitory – mamy synchronizowane metody, które nie mogą być wykonywane na raz. Wtedy możemy zastosować naiwne czekanie na podstawie jakichś zmiennych, bo wewnątrz danego monitora może pracować tylko jeden wątek i nie ma możliwości nadpisanie tych zmiennych przez inne wątki.

Innym pomysłem jest message passing – procesy przesyłają sobie wiadomości, które mogą odbierać. W momencie, gdy zapełni się bufor wiadomości wysyłanie staje się blokujące. Analogicznie dla pustego bufora. W innym wariancie wysyłanie i odbieranie jest blokujące, następuje przekazanie danych, gdy oba procesy będą gotowe do komunikacji. Dzięki temu nie ma buforów.

Zajęcia 9: Działanie schedulera

2024-12-04

Będziemy kategoryzować procesy, żeby jakoś przydzielać im zasoby. Możliwy jest podział na te ograniczone przez procesor i te ograniczone przez wejście/wyjście (CPU bound i I/O bound) – druga kategoria mało pracuje, raczej czeka na odczyty. Pierwsza raczej ma co robić cały czas i nie blokuje się. Procesy mogą zmieniać swoje kategorie (np. kompilator najpierw czyta źródła, potem pracuje przy kompilacji, a potem zapisuje wynik).

Procesy możemy dzielić też na interaktywne (takie, gdzie ważne jest reagowanie na zachowanie użytkownika w czasie rzeczywistym), wsadowe (duże zadanie obliczeniowe, bez potrzeby interakcji) i procesy czasu rzeczywistego (reagowanie na zdarzenia w czasie rzeczywistym, odpowiednio szybko). Na jednym systemie mogą pracować procesy z różnych kategorii.

Scheduler musi zdecydować o przydziale zasobów, gdy proces się blokuje albo kończy. Może zdecydować o przekierowaniu zasobów, jeśli powstanie nowy proces, nastąpi przerwanie I/O (urządzenie I/O wyśle sygnał do systemu) lub nastąpi przerwanie zegarowe.

Jeśli w systemie działają wszystkie rodzaje procesów, to chcemy traktować podobne procesy tak samo, ale uwzględniając ich specyfikę (np. priorytety) i balansując tak, by wszystkie części systemu były równo obciążone.

W systemach wsadowych, które wykonują konkretne zadania obliczeniowe, można korzystać z takich metryk jak throughput (wykonane zadania w jednostce czasu), turnaround time (czas między rozpoczęciem a zakończeniem zadania, najlepiej ze znormalizowaną wielkością zadania) i CPU utilization (obciążenie procesora). Chcemy zachowywać się tak, aby odpowiednio maksymalizować lub minimalizować te metryki. W systemach interaktywnych interesuje nas szybka reakcja na bodźce i opóźnienie reakcji systemu proporcjonalne do czasu trwania zadania. W systemach czasu rzeczywistego ważne jest nadążanie za pojawiającymi się bodźcami.

W procesach wsadowych można stosować kolejkę: wykonujemy zadania w kolejności przyjścia, zablokowane procesy trafiają na koniec kolejki. Takie rozwiązanie ma słabe wskaźniki, zaczynamy bardzo dużo procesów, które po zablokowaniu idą na koniec kolejki i tylko zajmują pamięć operacyjną. Do tego bardzo obciążamy dysk, bo wpuszczamy wiele procesów, które zaczynają czytać i się blokują.

Lepszym pomysłem jest oszacowanie czasu wykonania zadania (możemy to zrobić w systemach wsadowych) i zaczynania od tego o najkrótszym czasie. Jest to optymalne z punktu czasu wykonania, jeśli znamy wszystkie zadania z wyprzedzeniem, ale przestaje być, gdy zaczynają pojawiać się nowe zadania i do tego nie ma mechanizmów równoważenia procesów CPU-bound i I/O-bound.

Możemy wybierać proces o najkrótszym czasie do zakończenia i ewentualnie wywłaszczać przy pojawieniu się nowego procesu. Wtedy krótkie zadania realizują się szybko i da się uwzględniać blokowanie, ale duże zadania mogą być odkładana w przyszłość w nieskończoność.

W miarę dobrym rozwiązaniem jest three-level scheduling. Mamy trzy poziomy:

- Admission scheduler, który wybiera zadania, które będą wykonywanie współbieżnie, równoważy przy tym procesy CPU-bound i I/O-bound i minimalizuje czas wykonania zadań. Procesy dostają jakieś priorytety (np. związane z oczekiwanym czasem wykonania), które scheduler potem dostosowuje.
- Memory scheduler, który działa, gdy procesy nie mieszczą się w pamięci. Wybierane są procesy, które umieszczamy na dysku, zwalniając pamięć operacyjną. Znowu równoważymy procesu CPU-bound i I/O-bound, tym razem już mając informację o tym, w jaki sposób pracowały.
- CPU scheduler, który zarządza tym, kto dostanie procesor. On również równoważy.

W systemach interaktywnych można stosować system round robin: każdy proces dostaje kwant czasu, procesy są ustawione w kolejkę. Proces pracuje, dopóki trwa jego kwant, potem dostaje nowy i idzie na koniec kolejki. Jeśli proces się zablokuje, to po odblokowaniu wraca na początek kolejki z pozostałym czasem. Ważna jest wartość kwantu, dla dużych system ma opóźnioną reakcję, dla małych dużo zasobów jest traconych na zmiany kontekstu. W praktyce stosuje się 20-50ms. Wadą tego systemu jest równe traktowanie wszystkich procesów.

Priority scheduling: uogólnienie round-robin, możemy dawać różne kwanty dla różnych priorytetów, trzymać różne listy dla każdego priorytetu, dawać pierwszeństwo wyższym priorytetom. System może obniżać priorytety, robi to, gdy proces cały czas jest gotowy do pracy – jest sam w swojej kolejce i zużył kilka kwantów pod rząd. Gdybyśmy nie zmniejszyli mu priorytetu, to niższe priorytety nigdy by nie pracowały. Jednocześnie faworyzujemy procesy, które się blokują – jeśli proces zużył małą część swojego kwantu, to dostaje duży priorytet.

Lottery scheduling: procesy dostają żetony (różne ilości), scheduler wybiera losowy żeton i przydziela procesor właścicielowi. Współpracujące procesy mogą sobie przekazywać żetony. Wtedy nasza architektura w pewnym sensie sama się organizuje. Sensownie by było, gdyby procesy użytkownika miały po jednym żetonie a serwery nie miały żadnych. Proces wysyłający wiadomość do serwera daje mu swój żeton – wtedy serwery pracują, jak mają coś do roboty (im więcej, tym więcej mają żetonów), a zablokowane procesy użytkownika nie mają żetonów.

Guaranteed scheduling: każdy proces ma zagwarantowany jakiś procent procesora (różny dla różnych procesów), scheduler tak ustawia procesy na procesorach, żeby to zagwarantować. Tą strategię stosuje Linux. Działa ona tak: pamiętamy, ile każdy proces pracował, trzymamy w kolejce, w której pierwsze są te, które pracowały najkrócej. Pierwszemu procesowi dajemy tyle czasu, aby jego czas pracy wynosił teraz tyle, ile wynosiła średnia. Gdy jesteśmy bardzo blisko średniej, to przydzielane czasy robią się małe. Wtedy przydzielamy jakiś ustalony minimalny czas. Średnią można zmieniać w czasie stałym, ale ciężej ze zmianą pozycji w kolejce. Dlatego zamiast kolejki trzymamy drzewo czerwono-czarne, przesunięcie jest logarytmiczne. Wprowadzenie priorytetów: przydzielony czas nie jest prawdziwym czasem, procesy

z większym priorytetem mogą pracować więcej przy takiej samej liczbie przydzielonych zasobów. Nowo powstałe procesy zaczynają z czasem równym średniej – dzięki temu nie są traktowane lepiej niż długo działające procesy.

Systemy czasu rzeczywistego dzielimy na hard real time i soft real time – te, w których reakcja w odpowiednim czasie jest absolutnie kluczowa i te, w których chcemy ją zapewnić tylko zazwyczaj. Zdarzenia mogą być periodyczne lub aperiodyczne. Periodyczne jest łatwo obsłużyć, bo możemy szeregować statycznie (zawsze tak samo, bo zdarzenia przychodzą zawsze tak samo), w aperiodycznych musimy dynamicznie. Jeśli czas obsługi zdarzenia i to C_i, a zdarza się ono z częstotliwością P_i, to chcemy, aby było $\sum_i \frac{C_i}{P_i} \leq 1$ – oczekiwany czas pracy w jednostce czasu musi być mniejszy niż 1.

W Minixie mamy kolejki dla każdego priorytetu, w obrębie każdej round robin. Przy inicjalizacji jądra ustalamy te kolejki w jakiś ustalony sposób (z największym priorytetem system, potem inne serwery) i przekazujemy kontrolę do schedulera. Procesy, które dwa razy z rzędu dostały procesor i zużyły cały kwant są degradowane, inaczej są promowane. Nie dotyczy to zadań jądra i procesu idle – proces o najniższym priorytecie, sam w swojej kolejce, dostaje procesor, gdy system nie potrzebuje nic robić. Każdy rodzaj procesu ma swój maksymalny priorytet, do tego nie można spaść do priorytetu IDLE.

Minixowy scheduler jest napisany głównie w C, są w nim fragmenty assemblerowe robiące bardzo niskopoziomowe rzeczy, charakterystyczne dla danej architektury.

Zajęcia 10: Message passing w Minixie

2024-12-11

Message passing (IPC) obsługuje 4 tryby: `SEND`, `RECEIVE`, `SENDREC` i `NOTIFY`. Pierwsze trzy są blokujące, ostatni nie jest ani blokujący, ani buforowany.

Rozmawiające ze sobą serwery mogą chcieć przekazać sobie informację na raz (bo dostały wiadomość z zewnątrz, co może się zdarzyć w dowolnym momencie). W takiej sytuacji nastąpiłby deadlock. Rozwiązaniem tego jest zastosowanie `NOTIFY`. Taka wiadomość nie ma żadnej treści, ustawia tylko odpowiednią flagę w odbiorcy. Po dostaniu jej serwer wie, że powinien się skontaktować z tym drugim, i dopiero potem on wysyła mu swoją wiadomość. Przykład: VFS komunikuje się z driverem dysku, każe mu odczytać jakieś dane. Po odczytaniu driver chce to zakomunikować VFSowi, więc wysyła `NOTIFY`, a potem VFS może się zapytać o jego komunikat.

Każdy proces ma maskę operacji IPC, które może wykonać oraz maskę procesów, do których może wysyłać. Procesy użytkownika mogą wysyłać tylko `SENDREC` do PMa, FSa i RSa.

Zajęcia 11: Zegar systemowy

2024-12-18

Wiemy już, że zegar systemowy obsługuje przerwania zegarowe. Do tego musi pamiętać czas (liczba przerwań od startu systemu) i obsługiwać alarmy (funkcjonalność sleep dla serwerów). Minixowy zegar jest częścią kernela, mogą się z nim komunikować tylko serwery.

Kod zegara dzieli się na dwie części. Jedna zajmuje się handlowaniem przerwań, ona tylko odnotowuje, co należy zrobić (nie chcemy tego robić w trybie przerwania) i wykonuje drobne zadania – przede wszystkim aktualizuje swoje wewnętrzne dane. Większe zadania są przekazywane do drugiej części, która działa już w normalnym trybie i na przykład zajmuje się wywłaszczaniem procesów.

Zajęcia 12: Zarządzanie pamięcią

2025-01-08

Programy w pamięci są ułożone w ciągłych blokach. Jak potrzebujemy więcej pamięci, to dostajemy ją, ale w pewnym momencie dalsza pamięć może być zajęta przez coś innego – wtedy musimy przesunąć nasz proces w inne miejsce, które już można rozszerzyć. Taka realokacja wymaga, aby odpowiednio zmienić wskaźniki w programie, żeby wskazywały na nową pamięć.

Gdy brakuje pamięci system może zamrozić jakiś proces i zapisać go na dysku, udostępniając jego miejsce innym procesom.

Procesy w systemie są odseparowane, nie widzą nawzajem swojej pamięci. Chcemy jednak, żeby mogły współdzielić pewne jej elementy, np. biblioteki – procesy (raczej) ich nie zmieniają, więc mogą być wspólne.

Jest kilka prostych strategii alokacji pamięci:

- first fit – ładujemy program w pierwszą lukę w pamięci, do której się zmieści
- next fit – to samo, ale nie zaczynamy od początku
- best fit – szukamy najdokładniej pasującej luki (zostawia najmniej miejsca)
- worst fit – na odwrót, bierzemy największą lukę

Problem fragmentacji – mamy sporo wolnego miejsca, ale w małych blokach, więc nie można nic po prostu wsadzić. Best fit powoduje dużą fragmentację, worst fit jest lepsze.

Proces nie pracuje na żywej pamięci RAM, tylko na pamięci wirtualnej. Mamy jednostkę hardwarową (Memory Management Unit), która jest w procesorze i umożliwia przydzielanie takiej pamięci wirtualnej. Robi to za pomocą stronicowania – pamięć wirtualna i rzeczywista są podzielone na bloki. MMU mapuje te bloki ze sobą. MMU interpretuje pierwsze bity wskaźnika jako adres strony, a następne jako miejsce na stronie. Próba odwołania się do niezmapowanej pamięci generuje wyjątek.

Tablica stron może być wielopoziomowa – w adresie najpierw page directory, które wskazuje na odpowiednią page table i ona dopiero na adres fizyczny. W Minixie stosuje się czteropoziomowe tablice. Powoduje to wolniejszy dostęp, ale dzięki temu nie musimy pamiętać niezmapowanych obszarów pamięci (jeśli na wysokim poziomie coś jest niezmapowane, to pod spodem nie ma potrzeby trzymania kolejnych struktur).

MMU ma cache (TLB – Translation Lookaside Buffers). Jeśli program nie strzela w losowe miejsca w pamięci, to wszystko działa szybko. TLB nie jest bardzo duże (około 64 sloty), a mimo to działa – lokalność referencji, czyli programy raczej korzystają z pamięci w tej samej okolicy (w szczególności programiści piszą programy w ten sposób, bo hardware tego oczekuje).

Pamięć wirtualna powoduje, że pamięć fizyczna nie musi być przydzielana w sposób ciągły, więc potrzeba realokacji spada i staje się ona prosta – nie trzeba zmieniać procesowi wskaźników, wystarczy zmienić je w MMU. Do tego można wyrzucać z pamięci część pamięci procesów (pojedyncze ramki), by oszczędzać pamięć. Współdzielenie pamięci też jest proste.

Mechanizmy wyrzucania stron z pamięci: najlepiej ta, która będzie użyta najpóźniej, ale tak się zazwyczaj nie da. Dlatego wybieramy stronę ostatnio nieużywaną: pamiętamy, czy nastąpiło odwołanie do pamięci i przypisanie do niej czegoś (w ostatniej przeszłości). Najpierw bierzemy całkiem nieużywane strony, potem te tylko modyfikowane, potem tylko accessowane i na koniec jakiekolwiek. W obrębie tych grup można np. losowo.