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 dostać 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`:

```
1    struct flock {
2        short l_type; // F_RDLCK, F_WRLCK, R_UNLCK
3        short l_whence; // typy jak przy lseek'u, od kiedy liczyć offset
4        off_t l_start; // offset do początku segmentu
5        off_t l_len; // długość segmentu
6        off_t l_pid; // id procesu trzymającego locka
7    }
```

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
- `sigselset(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.