Model procesu w systemie Linux

Tomasz Borzyszkowski

Definicja procesu klasyka

Definicja [M.Bach WNT95]

Proces jest wykonaniem programu i składa się ze zbiorowości bajtów, które CPU interpretuje jako instrukcje maszynowe (tzw. tekst), dane i stos.

Jądro steruje wykonaniem procesów (stwarzając wrażenie, że wiele procesów wykonuje się jednocześnie); kilka procesów może być wcieleniem jednego programu.

Proces wykonuje się przechodąc przez ściśle ustalony ciąg instrukcji (stanowiący całość) i nigdy nie wykonuje skoków do instrukcji innego procesu tzn. czyta i zapisuje swoje dane oraz stos, lecz nie może czytać ani zapisywać danych i stosu innych procesów.

Procesy komunikują się z innymi procesami i z resztą świata za pomocą funkcji systemowych.

Co proces *wie* ?

Dla każdego programu/procesu/wykonania jądro systemu utrzymuje informacje o:

- Bieżącym stanie wykonania (np. czekanie na powrót z trybu jądra, ...), zwanym **kontekstem** programu
- Plikach, do których program ma dostęp
- **Uprawnieniach** dotyczących programu (przeważnie dziedziczone po właścicielu procesu lub grupie procesów)
- Bieżącym katalogu programu
- Obszerze pamięci, do którego program ma dostęp, i zawartości tego obszaru

Procesy rozumiane jako wykonania kodu posiadające własności przedstawione powyżej były podstawową jednostką pracy pierwszych systemów Unixowych. Tylko procesy miały możliwość działania na procesorze.

Dodatkowa komplikacja wątki

Wątek umożliwia pojedynczemu programowi działanie w kilku miejscach kodu jednocześnie. Wszystkie wątki utworzone przez program dzielą większość zasobów programu, tj: informacje o otwartych plikach, uprawnienia, bieżący katalog, obraz pamięci, ... Gdy tylko jeden z wątków procesu modyfikuje zmienną globalną, pozostałe natychmiast widzą jej nową wartość.

Wiele implementacji systemów Unixowych, w tym wersje Systemu V (AT&T), zaprojektowano tak, aby podstawową jednostką systemu zarządzaną przez jądro był wątek, a procesy stały się kolekcjami wątków dzielących zasoby. Dzięki współdzieleniu zasobów przełączanie między wątkami jest znacznie szybsze niż między procesami.

Z powyższych powodów w wielu systemach Unixowych istnieje podwójny model procesu, w którym odróżnia się wątki od procesów.

Model procesu Linux

Przełączanie kontekstu procesów w Linuxie było zawsze bardzo szybkie, podobnie jak przełączanie wątków w innych Unixach. Zasugerowało to twórcom Linuxa, że nie należy zmieniać tego co dobrze działa by wprowadzić wątki. Zdecydowano się za to na by pozwolić procesom na korzystanie z zasobów dzielonych w bardziej liberalny sposób.

W Linuxie proces jest zdefiniowany wyłącznie jako jednostka, której działaniem zarządza system, a jedyną rzeczą charakterystyczną procesu jest jego kontekst wykonania. Nie wynika z tego nic dla dzielenia zasobów, ponieważ proces macierzysty, tworzący proces potomny, ma pełną kontrolę nad tym, które zasoby będą między nimi dzielone.

Takie podejście pozwala zachować tradycyjny Unixowy model zarządzania procesami. Interfejs wątków jest budowany poza jądrem.

Atrybuty procesów

Identyfikator procesu (PID, Process ID): numer jednoznacznie identyfikujący proces

Identyfikator rodzica procesu (PPID, Parent Process ID): identyfikator PID procesu będącego rodzicem danego procesu.

Liczba nice: liczba wyznaczająca *ważność* danego procesu w stosunku do innych procesów. Nie jest to to samo co **priorytet** procesu wyliczany dynamicznie na podstawie liczby nice i zużycia zasobów CPU przez dany proces.

TTY: urządzenie reprezentujące terminal kontrolny danego procesu.

RUID i EUID (Real User ID i Effective User ID): RUID to ID użytkownika, który uruchomił dany proces. EUID zwykle równy RUID chyba, że użytkownik uruchomił program z ustawionym prawem SUID.

RGID i EGID (Real Group ID i Effective Group ID): podobnie jak RUID i EUID.

Zobacz: **`guiduid.c`**

Atrybuty procesów implementacja

PID jest dodatnią liczbą całkowitą, która jednoznacznie określa działający proces i jest zapamiętywana w zmiennych typu **`pid_t`**.

Jeżeli proces ginie, jego kod wyjścia jest pamiętany tak długo w tablicy procesów, aż poprosi o niego proces macierzysty. Procesy, które zakończyły działanie, a informacja o nich jest utrzymywana by mieć dostęp do ich kodu wyjścia, są nazywane **zombie**. Zaraz po odczytaniu kodu wyjścia procesu zombie jest on usuwany.

Jeżeli proces rodzica kończy działanie, przed zakończeniem procesów potomnych, procesy potomne stają się potomkami procesu **init**. Proces init jest pierwszym procesem, który powstaje podczas inicjowania systemu. Jego PID ma wartość 1. Jednym z jego głównych zadań jest odbieranie wartości kodów wyjścia procesów, których rodzice *zginęli*, by móc usunąć je z tablicy jądra.

Proces może poznać swój PID i PPID za pomocą funkcji **`getpid()`** i **`getppid()`**.

Zobacz: **`getpid.c`**

Cykl życia procesu

Argumenty programu

Istnieją dwa typy wartości przekazywanych do nowych programów w chwili ich uruchomienia:

- Argumenty wiersza poleceń; prototyp funkcji **`main()`**:

  ```
  int main(int argc, char *argv[]);
  ```

  Wartości zwracane, interpretowane jest tylko 7 najmniej znaczących bitów: 0 = OK; od -1 do -128 = proces zatrzymany przez inny proces lub jądro; od 1 do 127 = zakończenie z powodu błędu

- Zmienne środowiskowe. Zmienna **`environ`** zdefiniowana w pliku **`<unistd.h>`** zawiera wskaźnik do środowiska. Najczęściej ze zmiennych środowiska korzysta się za pośrednictwem funkcji: **`getenv()`**, **`putenv()`** oraz (niezgodna z POSIX ale występuje w BSD i Linuxie) **`setenv()`**.

Zobacz: **`env{1,2}.c`**

Tworzenie procesów potomnych

W Linuxie są dostępne dwie funkcje systemowe do tworzenia nowych procesów: **`fork()`** i **`clone()`**. Funkcja **`clone()`** służy do tworzenia wątków.

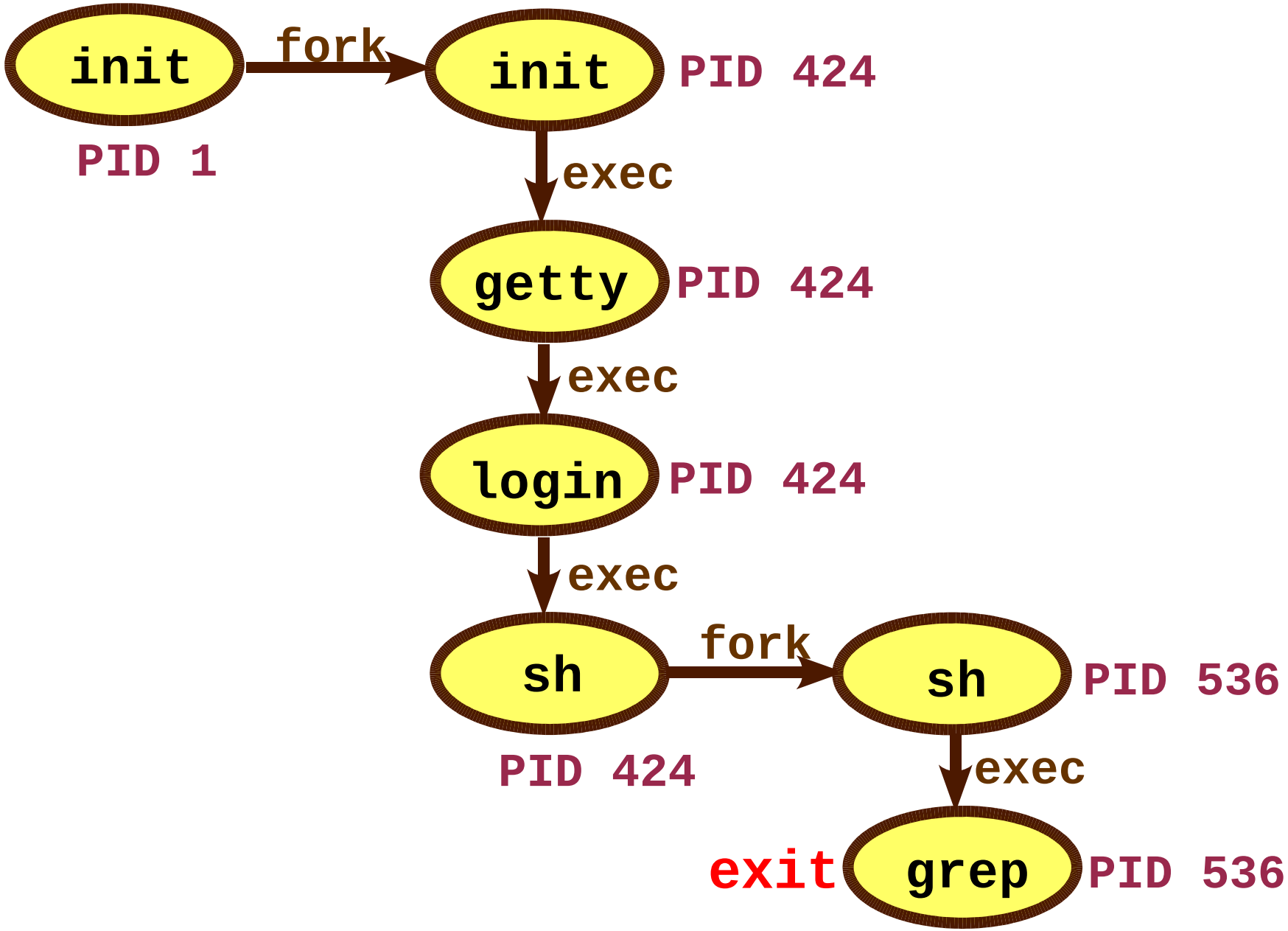
Funkcja **`fork()`** posiada następującą własność: powrót z jej wywołania następuje nie raz, ale dwa razy. Raz w procesie macierzystym i raz w procesie potomnym. Zgodnie z POSIX nie można zakładać, który powrót będzie pierwszy.

Każdy z powrotów z **`fork()`** przekazuje inną wartość. W procesie macierzystym jest to PID nowo utworzonego procesu potomnego, natomiast w procesie potomnym 0.

Powyższa różnica jest jedyną różnicą widoczną w procesach. Proces potomny i macierzysty mają: ten sam obraz pamięci, te same uprawnienia, te same otwarte pliki, te same procedury obsługi przerwań, ...

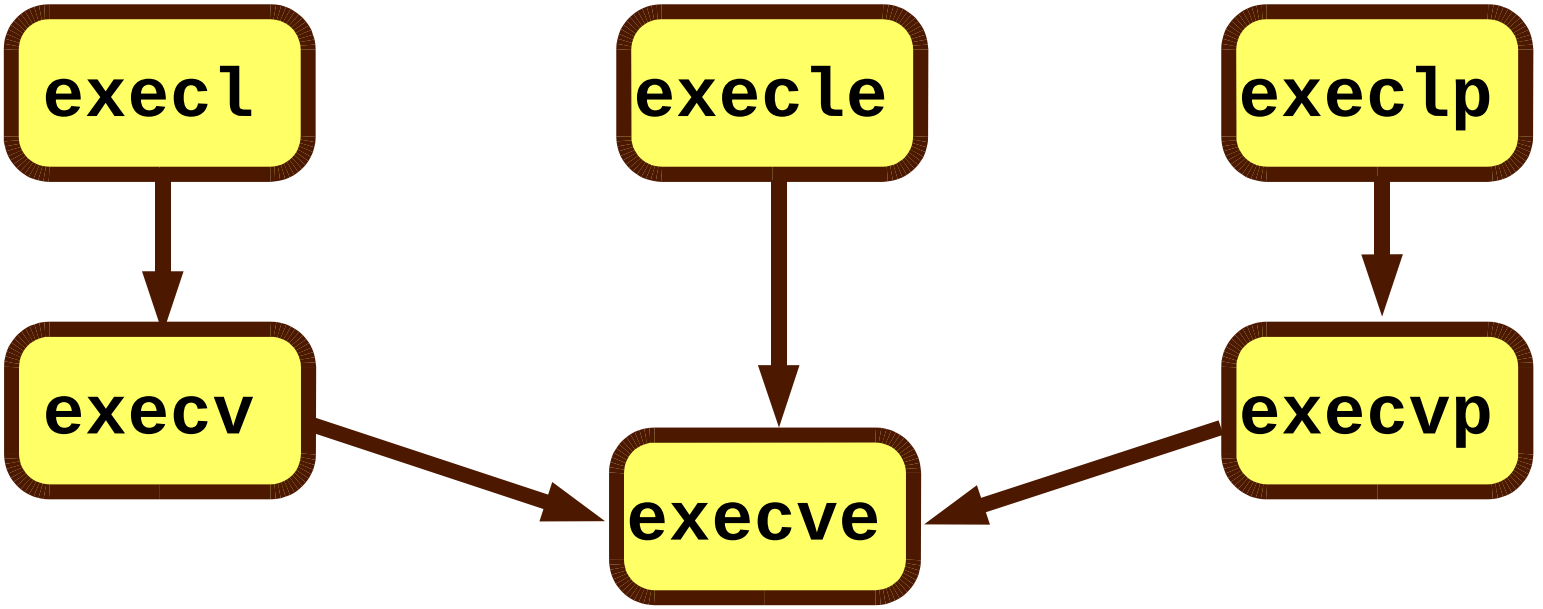
Zobacz: `potomek.c`

Nowe procesy fork-and-exec



Uruchamianie programów exec

Jak pokazano na poprzednim slajdzie do zapoczątkowania nowego procesu używa się funkcji **`fork()`** i funkcji z rodziny funkcji **`exec()`**. Główna różnicą pomiędzy funkcjami rodziny **`exec()`** jest sposób przekazywania parametrów. Wszystkie funkcje tej rodziny ostateczenie wykonują rzeczywistą funkcję systemową **`execve()`**.



Uruchamianie programów cd

Wszystkie warianty **exec()** wykonują to samo zadanie: przekształcają proces wywołujący przez załadowanie nowego programu do jego przestrzeni pamięci. Jeśli wywołanie **exec()** jest pomyślne, wywołujący program zostaje całkowicie przykryty przez nowy program, wykonywany od samego początku. Stary program jest wymazywany przez nowy. Tak więc, nie ma żadnego powrotu z pomyślnego wywołania funkcji **exec()**.

Przykład:

```
int execl(const char *path,
          const char *argv0, ..., const char *argvn,
          (char *)0);
```

Pierwszy parametr jest nazwą (pełną) pliku zawierającego program do wykonania. Drugi jest nazwą programu lub polecenia bez ścieżki, reszta (bez ostatniego), to paramerty wywołania. Ostani wskaźnik **NULL** oznaczający koniec listy parametrów.

Zobacz: **runls.c** **runls2.c** **myecho.c** **runmyecho.c** **runls3.c**

`fork()`, pliki i dane

Tworzony za pomocą **`fork()`** proces potomny jest **kopią** swojego rodzica. W szczególności wszystkie zmienne procesu potomnego zachowują wartość, którą otrzymały w procesie rodzicielskim. Ponieważ **dane** dostępne dla potomka są **kopią** danych rodzica, zmiany w jednym procesie nie będą wpływać na wartość zmiennych w drugim procesie.

Pliki otwarte w procesie rodzicielskim są też otwarte w procesie potomnym. Potomek utrzymuje swoją własną kopię deskryptorów plików. Jednak po wywołaniu **`fork()`** wskaźnik odczytu/zapisu dla każdego pliku jest współdzielony między potomkiem a rodzicem. Jest to możliwe, ponieważ wskaźnik odczytu/zapisu jest utrzymywany przez system i po każdej operacji wej/wyj system *aktualizuje* deskryptory plików.

Zadanie: *Napisz program, pokazujący, że zmienne programu w procesie rodzicielskim i potomnym mają te same wartości początkowe, ale są niezależne.*

Zobacz: **`proc_file.c`**

Kończenie procesów `exit`

Wielokrotnie już używaliśmy funkcji **`exit()`** w programach w C do kończenia procesu. Argument funkcji **`exit()`** nazywamy **stanem zakończenia** procesu. Jego młodsze 8 bitów jest dostępne dla procesu rodzicielskiego, jeżeli wykonywał funkcję **`wait()`**. Proces rodzicielski może w ten sposób badać czy proces potomny zakończył się sukcesem (watość **0**), czy porażką (wartość **nie 0**).
Funkcja **`exit()`** wykonuje także pewną liczbę innych działań, np. zamyka wszystkie otwarte deskryptory plików. Programista może z jej wywołaniem związać własne funkcje obsługi zakończenia i wykonać tzw. **gruntowne działania czyszczące**, np. czyszczenie buforów. Do wiązania takiego używa się funkcję:

```
int atexit(void (*func)(void) );
```

`atexit()` rejestruje funkcję wskazywaną przez **`func`**. Wszystkie zarejestrowane funkcje podczas zakończenia będą wywołane w kolejności odwrotnej do ich rejestracji. Istnieje także funkcja systemowa **`_exit()`**, działająca jak **`exit()`** ale bez czyszczenia.

Zobacz: **`zakoncz.c`**

Synchronizacja procesów

Zobacz: **status.c**

Funkcja **wait()** chwilowo zawiesza wykonanie procesu, podczas gdy proces potomny działa. Po zakończeniu procesu potomnego, czekający proces rodzicielski zostaje wznowiony. Jeśli działa więcej niż jeden proces potomny i gdy tylko jeden z nich się zakończy, **wait()** powraca.

Wartość zwracana przez funkcję **wait()** to zwykle ID procesu zakończonego potomka. Jeśli **wait()** zwróci **(pid_t) -1**, może to oznaczać, że nie istnieje żaden proces potomny, wówczas **errno** będzie zawierać kod błędu **ECHILD**.

Funkcja **wait()** pobiera jeden argument, który jest wskaźnikiem do liczby całkowitej. Jeśli wskaźnik jest równy **NULL**, to argument jest ignorowany. Jeśli wskaźnik był różny od **NULL**, to po powrocie będzie zawierał wskazanie na status zakończenia procesu potomnego, przekazywany przez **exit()**. Wartość ta jest zapamiętywana w 8 starszych bitach wskazywanego wyniku (makro **WEXITSTATUS**).
8 młodszych bitów musi być równe 0 (makro **WIFEXITED**).

Czekanie na konkretny proces

Do oczekiwania na zakończenie procesu potomnego o konkretnym numerze PID służy funkcja:

```
pid_t waitpid( pid_t pid, int *status,
               int options);
```

Pierwszy argument określa ID procesu potomnego, na który proces rodzicielski chce czekać. Jeżeli jest on ustawiony na -1 (oznacza dowolny proces potomny), a trzeci argument na 0, to zachowuje się dokładnie jak **`wait()`**. Drugi argument po powrocie będzie zawierał stan procesu potomnego.

Trzeci argument może przybierać różne wartości zdefiniowane w **`<sys/wait.h>`**. Często używaną wartością jest **`WNOHANG`**. Powoduje powrót z **`waitpid()`** bez blokowania, tj. bez czekania na zakończenie potomka. Zachowanie takie może służyć do monitorowania procesu potomnego. Przy ustawionej fladze **`WNOHANG`**, **`waitpid()`** oddaje 0 jeżeli proces potomny jeszcze się nie zakończył.

Zobacz: **`status2.c`**

Wielkość pliku

Istnieje w systemie ograniczenie wielkości pliku, który może być utworzony za pomocą funkcji systemowej **`write`**. Służy do tego funkcja:

```
long ulimit(int cmd, [long limit]);
```

Aby uzyskać bieżącą wartość ograniczenia wiekości pliku, należy wywołać funkcję z **`cmd`** równym **`UL_GETFSIZE`**. Wartość zwrócona będzie wyrażona w blokach pamięci.

Aby ustawić nowe ograniczenie wielkości pliku, należy wywołać funkcję z **`cmd`** równym **`UL_SETFSIZE`** ustawiając nowe ograniczenie wyrażone w blokach pamięci w drugim parametrze.

Tylko administrator systemu może zwiększać ograniczenie wielkości plików. Natomiast procesy o UID takim jak właściciel pliku mogą zmniejszać to ograniczenie.

Priorytety procesów

System decyduje o porcji CPU przydzielanej procesowi m.in. na podstawie wartości całkowitej **nice**. Wartość ta ma zakres od 0 do zależnego od systemu maksimum. Im wyższa liczba, tym niższy priorytet procesu. *Uspołeczniony* proces powinien, jeżeli to możliwe zmniejszać swój priorytet, przydzielając więcej czasu innym procesom, używając funkcji systemosej **`nice()`**. Pobiera ona jeden argument, liczbę dodawaną do aktualnej wartości nice.

Administrator systemu jest jedynym użytkownikiem, który może dodawać do **nice** procesu wartości ujemne, zwiększając priorytet procesu.

Zadanie: Sprawdzić działanie funkcji **`nice()`** oraz **`ulimit()`**.