Rozdział 1

Opis algorytmu `exit`

autor: Grzegorz Nowakowski
email: gegez@email.com
nr albumu: 181093

1.1 Wprowadzenie

Procesy w systemie UNIX kończą swe wykonywanie wywołując funkcję systemową `exit()`. Może to nastąpić w jednej z trzech sytuacji:

- jawnie poprzez instrukcję `exit (error_code)`,
- automatycznie, gdyż podczas kompilacji programu zostaje dołączona funkcja inicjująca, która po powrocie z funkcji `main` wywołuje funkcję `exit()`,
- jądro wywoła `exit()` dla danego procesu, w przypadku, gdy nadejdzie nieprzechwytywany sygnał.

Algorytm `exit` zwalnia zasoby systemowe, z których korzystał proces (jeżeli jest to możliwe), oczyszcza kontekst, informuje o śmierci procesy sąsiednie (ojciec, potomkowie, itp.) oraz przechodzi do stanu *zombie*. Do komunikacji z procesami sąsiednimi wykorzystuje się sygnały. Ojciec, aby dowiedzieć się o wyniku działania syna wykorzystuje funkcję `wait()`, o której będzie mowa dalej.

1.2 Metoda wywołania oraz przyjęte konwencje

Funkcję `exit()` (zdefiniowaną w pliku *linux/exit.c*) wywołuje się następująco: *exit(code)*. Wartość *code* (typu *long*) jest przekazywana procesowi macierzystemu - może on odczytać ją za pomocą funkcji systemowej `wait()`. Konwencja przyjmowanych wartości jest następująca:

- jeżeli code wynosi 0 oznacza to, że proces wykonał się poprawnie,
- jeżeli code jest różny od 0 oznaczać to może, że nastąpił błąd podczas działania.

Może jednak nastąpić sytuacja, że proces otrzyma sygnał, którego nie przechwytuje, wtedy jądro może wywołać funkcję `exit()` - jako *code* umieszczany jest wtedy numer sygnału.

1.3 Szkic algorytmu `exit` w pseudokodzie

Dane wejściowe: kod powrotu dla procesu-ojca
Wynik: brak

```
jeśli proces jest w trakcie obsługi przerwania to
   wyświetl stosowny komunikat
koniec jeśli
jeśli pid procesu równa się zero to
   zatrzymaj prace jądra wraz z wypisaniem stosownego komunikatu
koniec jeśli
ustaw flagę procesu na PF_EXITING (w trakcie kończenia)
usuń prywatne zegary
jeśli ustawiona flaga CONFIG_BSD_PROCESS_ACCT to
   zapisz dane do pliku rozliczeniowego
koniec jeśli
jeśli używał semaforów ze znacznikiem SEM_UNDO to
   odwróć wykonane przez proces operacje na semaforach
koniec jeśli
zwolnij pamięć przydzieloną na segmenty kodu, danych, itp.
zamknij pliki otwarte przez proces (funkcja wewnętrzna!)
zamknij i-węzły katalogu bieżącego oraz korzenia
usuń tablicę obsługi sygnałów dla danego procesu
przejdź w stan ZOMBIE
ustaw wartość exit_code w strukturze task_struct (potrzebne przy funkcji wait() )
ustaw potomkom pole ojca tak, aby wskazywało na proces INIT
jeśli jakakolwiek grupa zostanie osierocona i w tej grupie są jakieś zatrzymane procesu (ang.
stopped) i ojciec jest w innej grupie niż my to
   wyślij sygnały SIGHUP i SIGCONT do wszystkich jej członków
koniec jeśli
wyślij sygnał aktualnemu ojcu informujący o śmierci dziecka
jeśli proces był liderem grupy to
   wyślij odpowiedni sygnał członkom grupy
koniec jeśli
jeśli ustawiona flaga DEBUG_PROC_TREE to
   sprawdź poprawność drzewa procesów i wyświetl informacje o ewentualnych nieprawidłowo-
   ściach
koniec jeśli
jeśli proces korzystał ze struktur exec_domain lub binfmt to
   zmniejsz odpowiednie liczniki dotyczące tych struktur
koniec jeśli
wywołaj szeregowanie procesów (ta funkcja nie powinna nigdy powrócić)
```

1.4 Opis działania algorytmu

Najpierw sprawdzamy, czy przypadkiem nie jesteśmy w trakcie obsługi przerwania, jeżeli jesteśmy to wyświetlamy odpowiedni komunikat i przechodzimy dalej. Dodatkowo zostaje ustawiona flaga PF_EXITING w strukturze procesu `task_struct`.

Następnie usuwamy prywatny zegar z listy zegarów.

Jeżeli ustawiona jest flaga CONFIG_BSD_PROCESS_ACCT to zapisujemy do pliku rozliczeniowego dane za pomocą funkcji `acct_process()`, która jest zdefiniowana w pliku *acct.h*; są to między innymi: nazwa programu, czas pracy w każdym z trybów, identyfikatory użytkowników, grup i używanych terminali, opis przebiegu działania programu oraz kod zakończenia programu.

Następnie jądro usuwa proces wywołujący `exit()` z listy procesów "śpiących" na semaforze, odwraca wszelkie operacje semaforowe wykonane z flagą `SEM_UNDO` (informacje o modyfikacjach poszczególnych semaforów znajdują się w strukturze `task_struct` w polu *semundo*); ma to na celu zapobieżeniu sytuacji, że proces opuścił semafor po czym zakończył swoje działanie (np. dostając sygnał *SIGKILL*), uniemożliwiając w ten sposób wejście do sekcji krytycznej innym procesom.

W następnym kroku zostaje zwolniona pamięć (pole *mm* struktury `task_struct`) przydzielona procesowi na tablicę stron, segment kodu i danych; jednak może się zdarzyć, że pamięć ta jest współdzielona z innymi procesami, w takim wypadku zostaje zmniejszony tylko odpowiedni licznik.

Po zwolnieniu pamięci, jądro przechodzi do sprzątania ogólnie rozumianej struktury plików (pole *files* w `task_struct`), tzn. zamyka otwarte przez proces pliki (deskryptory) za pomocą wewnętrznie zdefiniowanej funkcji `close_files()`; zwalnia i-węzły katalogu bieżącego oraz aktualnego katalogu głównego (korzenia) - znów w przypadku współdzielenia wyżej wymienionych struktur z innymi procesami zostają zmniejszone odpowiednie liczniki.

Następnie jądro zwalnia pamięć przydzieloną na tablicę obsługi sygnałów, jeżeli pamięć ta była współdzielona zostaje zmniejszony odpowiedni licznik. Dodatkowo zostaje zmieniony stan procesu (pole *state* w `task_struct`) na *zombie* `TASK_ZOMBIE`, ustawiony kod wyjściowy (szerzej przy omówieniu `wait()`).

W tym momencie posprzątaliśmy po sobie w strukturze plikowo-dyskowej, zwolniliśmy większość pamięci (z wyjątkiem miejsca w tablicy procesów!), jądro może przejść do posprzątania w strukturze drzewa procesów, tzn. usunąć proces z tego drzewa poprzez poprawienie wskaźników np. procesów potomnych oraz poinformować sąsiadów o śmierci. Wygląda to następująco:

- ustaw wszystkim potomkow jako ojca proces *INIT* (*child_reaper*)
- jeżeli jakakolwiek grupa zostanie osierocona przez moją śmierć wyślij im odpowiednie sygnały
- wyślij ojcu sygnał o swojej śmierci, przeważnie będzie to `SIGCHLD`, jednak może być inny (pole *exit_signal* w `task_struct`)
- uczyń proces *INIT* procesem macierzystym swoich procesów potomnych
- jeżeli któreś z nich było w stanie *zombie* to wyślij do *INIT* sygnał o śmierci dziecka
- jeżeli byliśmy liderem grupy związanej z terminalem, to wyślij członkom grupy odpowiednie sygnał (zawieszenia).

1.5 Interakcje z procesem macierzystym - `wait()`

Proces w trakcie swojego działania może stworzyć nowy proces za pomocą funkcji systemowej `fork()`. Wtedy powstaje osobny proces z osobnymi strukturami danych (zastosowana jest technika kopiuj przy zapisie (ang. *copy-on-write*), tzn. nowa struktura jest tworzona dopiero podczas próby jakiejkolwiek modyfikacji). Od tego momentu mamy dwa procesy: potomny i macierzysty. W takim przypadku, aby ojciec mógł dowiedzieć się o śmierci potomka (oraz np. o fakcie wystąpienia błędu w trakcie działania dziecka) wywołuje funkcję systemową `wait()`, a dziecko (jawie lub nie) funkcję `exit()`.

Funkcja `wait()` ma jeszcze jedną ważną rolę, otóż zwalnia pamięć przydzieloną na strukturę *task_struct* procesu potomnego oraz jego miejsce w tablicy procesów. Może się jednak zdarzyć, że albo proces macierzysty nie wywoła w swoim kodzie funkcji `wait()`, albo ignoruje sygnał *SIG-CHLD*; w takim przypadku mogło by się zdażyć, że w systemie powstają procesy *zombie* i nikt po nich nie sprząta, a zatem zajmują miejsce w tablicy procesów (które jest oczywiście ograniczone, najczęściej 512 pozycji). Dlatego też w funkcji `do_signal()`, jeżeli zachodzi taka potrzeba, zostaje wywołana automatycznie funkcja `wait()` z opcją *WNOHANG*, która niszczy "niechciane" procesy *zombie*.

1.6 Struktury danych

Główną strukturą danych, z jakiej korzysta algorytm `exit` jest *task_struct* (zdefiniowana w pliku *linux/sched.h*). Po krótce spróbuję objaśnić znaczenie poszczególnych pól, istotnych z punktu widzenia naszego algorytmu:

- *pid* - identyfikator procesu PID
- *pgrp* - numer grupy procesu
- *flags* - flagi procesu (ustawiamy jako *PF_EXITING*)
- *state* - stan w jakim znajduje się proces (ustawiany na *TASK_ZOMBIE*)
- *mm* - informacje o pamięci procesu
- *sig* - wskaźnik do tablicy obsługi sygnałów
- *fs* - informacje o systemie plików (i-węzły)
- *files* - informacje o otwartych plikach
- *exit_signal* - sygnał wysyłany do ojca przy śmierci
- *p_opptr* - wskaźnik do orginalnego ojca
- *p_pptr* - wskaźnik do aktualnego ojca
- *p_cptr* - wskaźnik do najmłodszego potomka
- *p_ysptr* - wskaźnik do młodszego brata (zwanego również poprzednim)

- *p_osptr* - wskaźnik do starszego brata (zwanego również następnym)
- *leader* - czy jest liderem grupy w sesji

W algorytmie często występuje zmienna *tsk*, która jest wskaźnikiem do struktury *current*, która jest po prostu kopią *task_struct* dla procesu, który właśnie się wykonuje, w naszym przypadku jest nim proces wykonujący "`exit()`".

1.7 Historia powstania algorytmu - zmiany

Przez całą swoją ewolucję algorytm *exit* uległ niewielu zmianom. Koncepcja funkcji kończącej pracę procesu pochodząca jeszcze z początków systemu UNIX jest nadal aktualna a zmianom uległy mało znaczące fragmenty kodu. Porównując ideę funkcji `exit()` zawartą w książce Bacha[2] z najnowszą implementacją z jąder 2.4.x nie widzi się wielkich zmian. Tak jak na początku trzeba odwrócić działanie funkcji `fork()`, tzn. usunąć proces z pamięci, posprzątać system plików, pozamykać otwarte pliki, itp.

W jądrze wersji 2.x.x pojawiła się nowa możliwość, jaką jest użycie platform wieloprocesorowych (SMP - symetryczna wieloprocesorowość). Wraz z nią kod jądra musiał ulec paru poprawkom (w tym oczywiście algorytmu *exit*). Trzeba było zagwarantować, że np. dwa procesy nie zapisują w jednym momencie do *task_struct*. Jednak nie zmieniło to idei algorytmu, zostały dodane jedynie dodatkowe instrukcje w rodzaju `write_lock(&tasklist_lock)`.

Można by się pokusić o zastanowienie się, czy kod jest optymalny. Jednak biorąc pod uwagę częstotliwość wykonywania instrukcji `exit()` w całym życiu systemu, można by stwierdzić, że nie ma to wielkiego znaczenia. W jednym momencie w systemie może znajdować się maksymalnie pareset procesów, duża część z nich jest albo na stałe (*idle*, *init*), albo rzadko są zabijane (konsole, demony). Główną czynnością jaką system wykonuje związaną z życiem procesów jest przełączanie ich kontekstów (tak, aby uzyskać pozorną równoległość wykonywania się ich, tzw. współbieżność). Sądzę, że raczej te części kodu można by było brać pod uwagę i patrzeć, czy są optymalne - w przypadku algorytmu *exit* nie ma to większego znaczenia (oczywiście, jeżeli chodzi o czas i szybkość).

Patrząc przez pryzmat poprawności, wydaje się, że algorytm jest dobrze zbudowany, czyści pozostałości po ginących procesach - odwraca działanie `fork()`. Jeżeli chodzi o poprawność działania w systemie wieloprocesorowym (opartym na technologii SMP) nie mogę niczego stwierdzić, gdyż nie mam możliwości sprawdzenia (nawet empirycznego) działania w takich warunkach. Musimy po prostu zaufać tysiącom programistów i użytkowników, że dostając stabilne i oficjalnie wypuszczone jądro, jest ono wolne od podstawowych błędów i niedociągnięć, jakim by było źle działający `exit()` w środowisku SMP.

Zastanawiający wydaje się fakt, że w wersji 2.2.17 brak jest testu, czy proces *INIT* nie wykonuje `exit()`. W wersji 2.4.x taki test się już pojawia; mogło by się wydawać, że to niczego nie zmienia, jednak podjąłem próbę zabicia procesu *INIT* wysyłając mu pierwsze 15 sygnałów (za pomocą polecenia bash'owego *kill -x 1*, gdzie x jest liczbą z przedziału 1-15. Wynik był zastanawiający, na ekranie pojawił mi się komunikat: *Segmentation violation....* i system po 30 sekundach się zrestartował. Być może nie wynika to z samej konstrukcji funkcji `exit()` jednak fakt, że w następnych wersjach taki test ma się pojawić oraz fakt, że system przestał działać prawidłowo może oznaczać, że taki test jest po prostu potrzebny (lub przydatny).

Podobnie wygląda sprawa ze skokiem do etykiety fake_volatile. W komentarzu jest napisane (i wydaje się być słuszne), że `schedule()` nigdy nie powraca a skok jest tylko pro forma. Jednak takie zabezpieczenia mogą poprawić stabilność i bezawaryjność działania systemu. Uwaga ta dotyczy braku testy opisanego w poprzednim akapicie.

1.8 Przykładowe zadania i ćwiczenia

1. Dlaczego funkcja `exit()` nie usuwa struktury *task_struct* oraz miejsca w tablicy procesów?
2. W jaki sposób proces potomny może sprawdzić, czy jego ojciec istnieje (nie wywołał funkcji `exit()`, odwołując się co najwyżej raz do *task_struct*?
3. Skoro `exit()` nie niszczy *task_struct* ani nie zwalnia miejsca w tablicy procesów, ani proces macierzysty nie musi wywołać funkcji `wait()`, to jaką mamy gwarancję, że tablica procesów nie zapełni się bezużytecznymi procesami *zombie*? A może właśnie znaleźliśmy błąd w systemie?

1.9 Rozwiązania

1. Robi to funkcja `wait()`, gdyż gdyby zrobiła to `exit()` zgubilibyśmy dane statystyczne, z których korzysta ta pierwsza; dodatkowo nie poznalibyśmy kodu powrotu (ang. *exit code*) potomka.
2. Sprawdzenie istnieje wskaźnik do ojca w strukturze *task_struct* oczywiście nie ma sensu, ale wystarczy sprawdzić, czy jego aktualnym procesem macierzystym jest *INIT* (*pid*=1).
3. Pomimo tego, że proces macierzysty nie wywoła `wait()`, funkcja `do_signal()` wysyłająca sygnał o śmierci wywoła tą pierwszą.

Literatura

[Rochkind 93] Rochkind M.J.: *Programowanie w systemie Unix dla zaawansowanych.* Warszawa, WNT 1993.

[Bach 95] Bach, Maurice J.: *Budowa systemu operacyjnego UNIX.* Warszawa, WNT 1995.

[Silberschatz 00] Silberschatz A., Galvin P.: *Podstawy systemów operacyjnych.* Warszawa, WNT 2000.

[Źródła jądra] Kod źródłowy jądra systemu linux w wersji 2.2.17 oraz strona http://lxr.linux.no/