Systemy Operacyjne 2

Zarządzanie Procesami

Arkadiusz Chrobot

Katedra Systemów Informatycznych

7 marca 2024

Plan

1. Deskryptor procesu
2. Rodzina Procesów
3. Tworzenie procesu
4. Kończenie procesu

Deskryptor procesu

Systemy operacyjne przechowują wszystkie informacje o każdym pojedynczymi procesie w *deskryptorze procesu*. W przypadku Linuksa jest to struktura typu `struct task_struct` zdefiniowana w pliku nagłówkowym `linux/sched.h`. Pamięć na takie struktury jest przydzielana przez *alokator plastrowy* (ang. *slab allocator*) (Ten mechanizm przydziału pamięci będzie omówiony na 10 wykładzie, jest także opisany w drugiej instrukcji laboratoryjnej.) Rozmiar deskryptora procesu w Linuksie wynosi około 1,7 KiB. Niektóre jego składowe są polami wskaźnikowymi do innych struktur o podobnym rozmiarze lub większych.

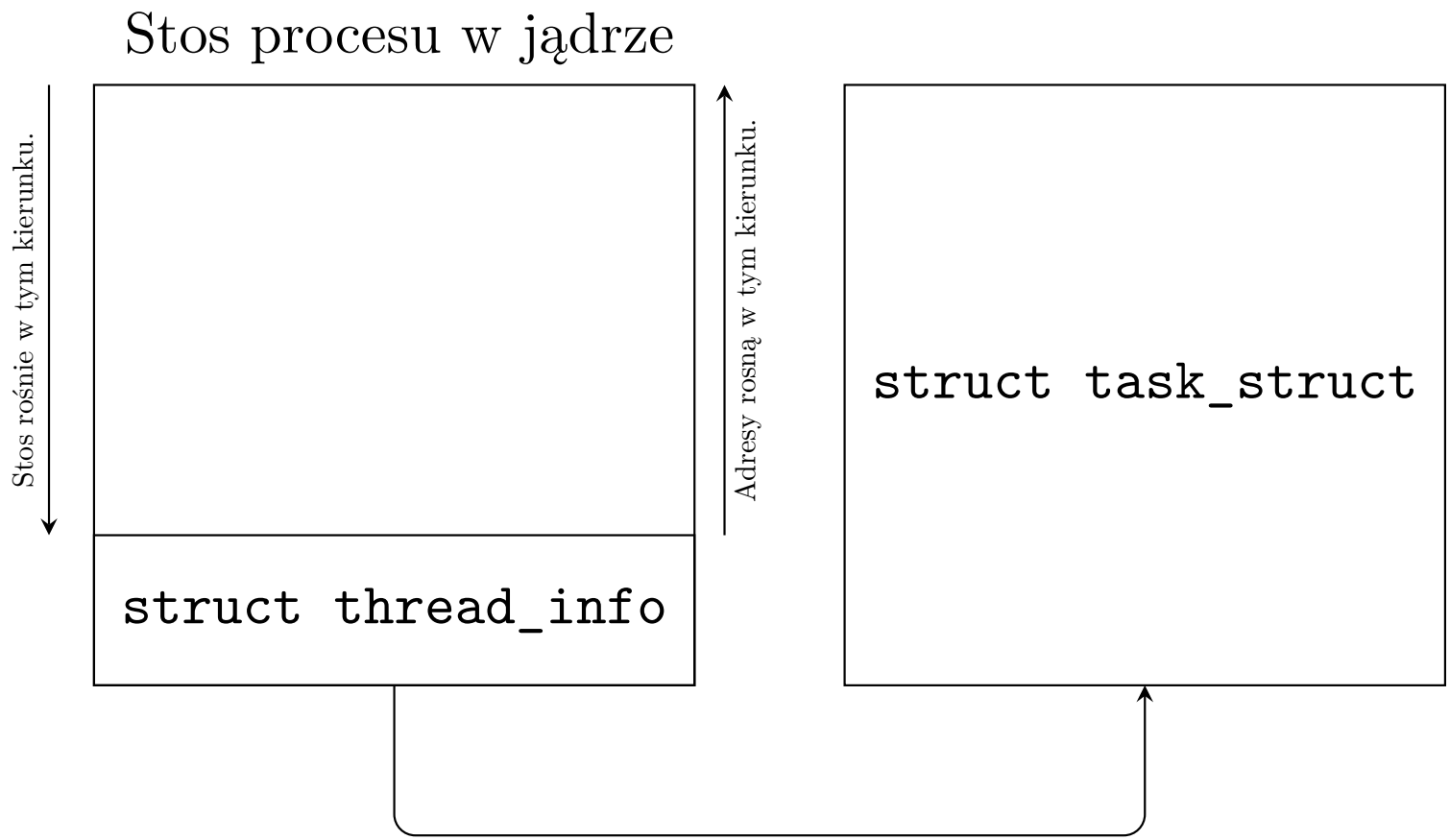
Deskryptor procesu

Umiejscowienie deskryptora

W wersjach jądra Linuksa poprzedzających serię 2.6 deskryptor proces był przechowywany na końcu stosu procesu w przestrzeni jądra (nazywanego w skrócie *stosem jądra*). Począwszy od wspomnianej serii stał się on jednak na tyle dużą strukturą, że zabierał zbyt wiele miejsca na stosie. Dlatego w tym miejscu umieszczono inną strukturę, typu `struct thread_info`, której jedna ze składowych jest wskaźnikiem na deskryptor (Rys. 1). Ta struktura ma mniejszy rozmiar niż deskryptor, ale również jest tworzona dla każdego procesu indywidualnie. Definicja jej typu, w wersji jądra 4.8, dla procesorów o architekturze x86 jest zamieszczona na listingu 1. Pochodzi ona z pliku nagłówkowego `asm/thread_info.h`.

Deskryptor procesu

Umiejscowienie deskryptora



Rysunek 1: Stos procesu w przestrzeni jądra i deskryptor procesu

Deskryptor procesu

Definicja typu `struct task_info`

```
1  struct thread_info {
2          struct task_struct *task; /* main task structure */
3          __u32 flags; /* low level flags */
4          __u32 status; /* thread synchronous flags */
5          __u32 cpu; /* current CPU */
6  };
```

Listing 1: Definicja typu `struct thread_info` dla procesorów o architekturze x86, w wersji jądra 4.8

Deskryptor procesu

Makro `current`

Kod jądra, który wykonywany jest w kontekście procesu, a w szczególności wywołanie systemowe, musi mieć możliwość szybkiego uzyskania dostępu do deskryptora procesu, który go aktywował, ze względu na informacje o tym procesie, zawarte w tej strukturze. W przypadku procesorów o organizacji RISC adres deskryptora można przechować w jednym z rejestrów, ale w procesorach o organizacji CISC musi on być wyliczony na bieżąco za każdym razem, kiedy jest potrzebny. W tym celu programiści jądra stworzyli makro `current`, które zwraca adres deskryptora bieżącego procesu. W tym przypadku słowo „bieżącego” należy rozumieć jako „tego, który uaktywnił kod jądra”. Makro `current` korzysta ze, zdefiniowanej osobno dla każdej architektury procesora, funkcji `current_thread_info()`. Fragment jej implementacji dla 32-bitowych procesorów o architekturze x86 przedstawia listing 2.

Deskryptor procesu

Makro `current`

```
1  movl $-8192, %eax
2  andl %esp, %eax
```

Listing 2: Fragment definicji funkcji `current_thread_info()` dla procesorów o architekturze x86-32

W wierszu nr 1 liczba -8192 jest przesyłana do rejestru `eax`, który jest 32-bitowy. Wartość 8192 to rozmiar stosu jądra (dwie stron, każda o wielkości 4096 bajtów, co daje $2 \times 4096 = 8192$). W zapisie binarnym ma ona postać 00000000000000000010000000000000. W opisywanym kodzie ta liczba jest ujemna, a więc jej reprezentacja w kodzie U2 jest następująca: 11111111111111111110000000000000. Jest ona użyta w wierszu nr 2 do zamaskowania trzynastu najmniej znaczących bitów wskaźnika stosu przechowywanego w rejestrze `esp`. Wynikiem tego działania jest adres „końca” stosu i tym samym także adres struktury typu `struct thread_info`.

Deskryptor procesu

Makro `current`

Po wyznaczeniu adresu struktury typu `struct thread_info` makro `current` zwraca wartość jej pola wskaźnikowego `task`, które przechowuje adres deskryptora bieżącego procesu:

```
current_thread_info()->task
```

Sposób wyznaczania adresu bieżącego procesu wyjaśnia dlaczego stos jądra musi być powiązany z deskryptorem procesu.

Deskryptor procesu

Począwszy od wersji 4.9 jądra, opisane powiązanie występuje tylko dla części z obsługiwanych w Linuksie procesorów. W przypadku procesorów x86, wskaźnik na deskryptor bieżącego procesu jest zdefiniowany jako tzw. zmienna per-cpu lub (w nowszych wersjach jądra) jako pole struktury będącej taką zamienną. Jak sugeruje nazwa, zmienna per-cpu ma osobną instancję dla każdego z procesorów, a więc ta instancja jest dla każdego z nich lokalna. Jest to dogodne rozwiązanie w systemach wieloprocesorowych[1]. Makro `current` zwraca jedynie adres przechowywany w takiej zmiennej. Z kolei w deskryptorze jest pole o nazwie `stack`, które zawiera adres stosu jądra.

[1] W ramach tego wykładu będziemy przyjmować, że wyrażenia *system wieloprocesorowy* i *system wielordzeniowy* oznaczają to somo. Komputer wyposażony w jeden procesor, z tylko jednym rdzeniem, będziemy nazywać *systemem jednoprocesorowym* lub *uniprocesorem*.

Deskryptor procesu

Identyfikator procesu

Wśród danych przechowywanych w deskryptorze procesu jest jego identyfikator (PID). Jest to liczba naturalna, unikatowa dla każdego procesu i przypisywana mu przez jądro. Jej wartość jest ograniczona do 32767. Ten limit wynika z tego, że jądro Linuksa musi być wstecznie kompatybilne z oryginalnym Uniksem. Jednakże wartość tego ograniczenia może zostać zmieniona przez użytkownika z odpowiednim poziomem uprzywilejowania, bez konieczności restartu systemu komputerowego. PID o wartości 1 jest przypisane do procesu użytkownika, który jest przodkiem wszystkich innych uruchomionych procesów. Historycznie tym procesem jest `init`, ale w nowszych dystrybucjach Linuksa został on zastąpiony przez `upstart` lub `systemd`, choć nazwa może być zachowana. Identyfikator procesu jest przechowywany w polu deskryptora o nazwie `pid` i typie `pid_t`.

Deskryptor procesu

Stan procesu

Każdy wykonujący się proces zmienia swój stan. Wewnątrz jądra bieżący stan procesu jest reprezentowany przez specjalną stałą, nazywaną również flagą, której wartość jest przypisana polu `state` deskryptora procesu. Liczba, nazwy, a nawet sposób użycia tych flag zmieniał się w trakcie rozwoju jądra, ale kilka najistotniejszych z nich zostało wymienionych na tym i następnych slajdach:

TASK_RUNNING określa, że proces jest aktywny lub gotów do wykonania; jądro nie rozróżnia tych dwóch przypadków,

TASK_INTERRUPTIBLE proces oczekuje na zdarzenie (śpi lub jest zablokowany, w terminologi Linuksa); może być wybudzony (odblokowany) przez sygnał związany z tym zdarzeniem lub dowolny inny,

TASK_UNINTERRUPTIBLE proces oczekuje na zdarzenie i może być wybudzony (odblokowany) tylko przez sygnał z nim związany; ten stan jest rzadko używany, bo uniemożliwia przerwanie procesu,

Deskryptor procesu

Stan procesu

TASK_KILLABLE proces czeka na zdarzenie i może być wybudzony (odblokowany) przez sygnał z nim związany lub jeden z sygnałów nakazujących jego przerwanie (ang. *fatal signals*),

TASK_STOPPED proces został wstrzymany przez sygnał,

TASK_TRACED proces podlega debugowaniu.

W deskryptorze znajduje się również osobne pole, o nazwie `exit_state`, które przechowuje stan zakończonego procesu. Może ono przyjmować wartości następujących flag:

EXIT_ZOMBIE proces się zakończył, ale proces rodzicielski nie aktywował dla niego wywołania systemowego `wait4()`; w RAM ciągłe jest stos jądra i deskryptor, które należą do tego procesu,

Deskryptor procesu

Stan procesu

EXIT_DEAD proces zakończył się, jego rodzić aktywował wywołanie `wait4()`, ale jądro nie skończyło jeszcze usuwania pozostałości tego procesu; stan ten jest używany do poinformowania kodu jądra wykonywanego na innych procesorach w systemie komputerowym, że proces jest w trakcie usuwania.

Funkcja `set_current_state()` służy do zmiany stanu bieżącego procesu, a w celu zmiany stanu dowolnego z nich można zastosować funkcję `set_task_state()`.

Rodzina Procesów

W Linuksie procesy są ze sobą powiązane. Dzięki tym powiązaniom tworzą *drzewo rodzinne procesów* (ang. *process family tree*). Każdy proces ma swojego rodzica, za wyjątkiem procesu `init` (lub `upstart` lub `systemd`), który jest przodkiem wszystkich pozostałych procesów. Każdy proces może mieć także procesy potomne (ang. *children*). Wszyscy bezpośredni potomkowie procesu są nazywani *rodzeństwem* (ang. *siblings*).
Te związki rodzinne są odwzorowane w deskryptorze procesu. Przykładowo, adres deskryptora rodzica procesu jest przechowywany w polu `parent` deskryptora procesu. Pole `children` jest listą wskaźników na deskryptory procesów potomnych (jeśli istnieją). Następująca instrukcja zapisuje w zmiennej `task` adres deskryptora rodzica bieżącego procesu:

```
struct task_struct *task = current->parent;
```

Rodzina Procesów

Ten kod z kolei umożliwia iterowanie po liście potomków procesu:

```
1 struct task_struct *task;
2 struct list_head *list;
3
4 list_for_each(list, &current->children) {
5       task = list_entry(list, struct task_struct, sibling);
6 }
```

Listing 3: Kod jądra iterujący po liście potomków bieżącego procesu

Rodzina procesów

Kod z listingu 3 używa API ogólnej implementacji listy stworzonej przez programistów jądra, która jest dokładniej opisana w trzeciej instrukcji laboratoryjnej. Deskryptory wszystkich procesów użytkownika są połączone w cykliczną listę dwukierunkową. Jej pierwszym elementem jest deskryptor procesu `init` (lub jednego z jego nowszych zamienników). Iterowanie po niej umożliwia makro `for_each_process(task)` (również opisane w trzeciej instrukcji). Z kolei makro `next_task(task)` zwraca adres następnego deskryptora procesu na tej liście, a `prev_task(task)` adres poprzedniego względem tego wskazywanego przez `task`. Jądro posiada również tablicę `pidhash`, która przechowuje adresy deskryptorów wszystkich procesów. Ta tablica ma 32768 elementów, co oznacza, że zakres wartości jej indeksów pokrywa zakres wszystkich możliwych wartości PID. Pozwala ona jądru szybko uzyskać adres deskryptora danego procesu na podstawie jego PID.

Tworzenie procesu

Linux, tak jak każdy inny system operacyjny kompatybilny z Uniksem, pozwala każdemu procesowi użytkownika stworzyć nowy proces (potomka) za pomocą wywołania funkcji `fork()` lub `vfork()`. Udostępnia on również specyficzną dla niego funkcje o nazwie `clone()`, przeznaczoną do tego samego celu. Linux używa także techniki *copy-on-write* (*COW*), która pozwala rodzicowi i potomkowi współdzielić przestrzeń adresową do momentu, aż któryś z nich nie zmodyfikuje wartości dowolnej zmiennej. W takim przypadku oba procesy otrzymują osobne segmenty danych, ale dalej współdzielą segment tekstu (kodu), który jest tylko do odczytu. Realizacja innego programu przez potomka jest możliwa dzięki rodzinie funkcji `exec()`.
Niezależnie od tego, która z przedstawionych funkcji jest użyta do utworzenia potomka, wszystkie one ostatecznie uruchamiają wywołanie systemowe `clone()`, które wywołuje funkcję jądra `do_fork()`, a ta z kolei korzysta z funkcji `copy_process()`.

Tworzenie procesu

Funkcja `copy_process()` wykonuje następujące czynności:

1. tworzy dla nowego procesu stos jądra i deskryptor,
2. sprawdza, czy utworzenie nowego procesu nie przekroczy limitu liczby procesów bieżącego użytkownika (właściciela procesu rodzicielskiego),
3. ustawia stan nowego procesu na `TASK_UNINTERRUPTIBLE`,
4. ustawia flagi procesu i uzyskuje dla niego PID,
5. w zależności od argumentów przekazanych do wywołania `clone()` kopiuje od rodzica lub tworzy na nowo struktury związane z zarządzaniem zasobami i sygnałami,
6. zwraca adres deskryptora nowego procesu.

Po zakończeniu `copy_process()` sterowanie wraca do funkcji `do_fork()`, która wybudza proces potomny i pozwala mu działać.

Tworzenie procesu

Wątki w przestrzeni użytkownika

Linux pozwala procesom użytkownika tworzyć wątki, ale w przeciwieństwie do innych systemów operacyjnych nie posiada żadnych podsystemów, które przeznaczone byłyby wyłącznie do ich obsługi. W przypadku Linuksa wątek użytkownika jest po prostu procesem użytkownika, który zawsze współdzieli większość swoich zasobów, włącznie z przestrzenią adresową, z innymi procesami-wątkami (rodzicem i ewentualnym rodzeństwem). Aby utworzyć wątek użytkownika wywołanie `clone()` musi zostać wywołane z następującymi argumentami:

```
clone(CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND,0);
```

podczas gdy funkcja `fork()` wywołuje `clone()` tak :

```
clone(SIGCHLD,0);
```

a `vfork()` tak:

```
clone(CLONE_VFORK|CLONE_VM|SIGCHLD,0);
```

Tworzenie procesu

Argumenty wywołania systemowego `clone()`

Istnieje kilka flag, zdefiniowanych w pliku nagłówkowym `linux/sched.h`, które służą jako argumenty wywołania `clone()`:

CLONE_FILES rodzic i potomek współdzielą otwarte pliki,

CLONE_FS potomek i rodzic współdzielą dane systemu plików,

CLONE_IDLETASK ustawia 0 jako PID potomka (będzie to proces bezczynności),

CLONE_NEWNS tworzy nową przestrzeń nazw dla potomka,

CLONE_PARENT „dziadek” nowego procesu staje się jego rodzicem,

CLONE_PTRACE wykonanie potomka jest śledzone (debugowane), tak jak procesu macierzystego,

CLONE_SETTID wartość TID (Thread Identifier) jest zapisywana do przestrzeni użytkownika,

CLONE_SETTLS tworzy nowy obszar TLS (Thread Local Storage — prywatny obszar danych wątku) dla potomka,

CLONE_SIGHAND rodzic i potomek współdzielą struktury obsługi sygnałów,

Tworzenie procesu

Argumenty wywołania systemowego `clone()`

CLONE_SYSVSEM rodzić i potomek korzystają z zachowania operacji `SEM_UNDO` charakterystycznego dla System V,

CLONE_THREAD rodzic i potomek są w tej samej grupie wątków,

CLONE_VFORK rodzic oczekuje na obudzenie przez potomka (potomek został utworzony przez wywołanie funkcji `vfork()`),

CLONE_UNTRACED zapobiega ustawieniu flagi `CLONE_PTRACE` dla potomka ,

CLONE_STOP utwórz potomka w stanie `TASK_STOPPED`,

CLONE_CHILD_CLEARTID wyzeruj TID dla potomka,

CLONE_CHILD_SETTID ustaw TID dla potomka,

CLONE_PARENT_SETTID ustaw TID dla rodzica,

CLONE_VM rodzić i potomek będą współdzielić przestrzeń adresową.

Tworzenie procesu

Wątki jądra

Jądro również może tworzyć swoje własne wątki. Przykładem takich wątków są *ksoftirqd* i *kworker*. Wątki jądra współdzielą przestrzeń adresową z resztą jądra, nie mają własnej. Kod takich wątków jest zaimplementowany w postaci funkcji, która najczęściej wykonuje określone czynności w pętli. Większość tych wątków kończy wykonanie, kiedy system komputerowy jest wyłączany, następuje jego restart lub gdy moduły z nimi związane są usuwane z jądra. Pojedynczy wątek jądra można utworzyć przy pomocy funkcji `kernel_thread()`. W wersji 2.6.1 jądra pojawiła się poprawka autorstwa Rusty'ego Russella, która wprowadziła bardziej wygodne w użyciu API do zarządzania wątkami jądra.

Tworzenie procesu

Wątki jądra

Wspomniane API składa się z następujących funkcji i makr:

`kthread_create()` tworzy nowy wątek jądra i zwraca adres jego deskryptora,

`kthread_run()` tworzy i aktywuje nowy wątek jądra,

`kthread_stop()` wysyła sygnał do wątku jądra, sugerujący, aby ten wątek się zakończył,

`kthread_should_stop()` funkcja wywoływana w wątku jądra i używana jako warunek w głównej pętli wątku, który sprawdza, czy ten wątek powinien się zakończyć,

`kthread_bind()` przypisuje wątek jądra do jednego lub większej liczby procesorów.

Bardziej szczegółowy opis tego API znajduje się w piątej instrukcji laboratoryjnej.

Kończenie procesu

Proces kończy działanie wywołując, bezpośrednio lub pośrednio, funkcję `exit()`, która aktywuje wywołanie `_exit()`, a ono z kolei wywołuje funkcję jądra `do_exit()`. Ta ostatnia jest odpowiedzialna za zwolnienie większości struktur danych jądra związanych z tym procesem i powiadomienie jego rodzica, że jego potomek się zakończył. Tylko deskryptor procesu i stos jądra są pozostawiane w pamięci. Te struktury zwalnia funkcja `release_task()` wywoływana przez wywołania systemowe `wait4()`. Ta funkcja zmniejsza również o jeden licznik procesów należących do użytkownika zakończonego procesu, usuwa deskryptor tego procesu z tablicy `pidhash`, z listy śledzony procesów (jeśli proces był debugowany), z listy wszystkich procesów i w końcu zwalnia pamięć przeznaczoną na deskryptor oraz strukturę typu `thread_info` tego procesu.

Kończenie procesu

Jeśli rodzić zakończonego procesu sam zakończył się przed nim, to ten proces mógłby utknąć w stanie *zombie*. W takim przypadku następuje *adopcja* tego procesu przez proces `init` (lub jego nowszy odpowiednik) lub przez proces, który należy do tej samej grupy, do której należał rodzic zakończonego procesu. Tą adopcją zajmuje się funkcja jądra o nazwie `forget_original_parent()`, która jest wywoływana przez funkcję `do_exit()`. Ta pierwsza sprawdza listę wszystkich procesów i listę śledzonych procesów, aby znaleźć nowego rodzica.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!