**Wykład drugi**

***Zarządzanie procesami w Linuksie***

Podobnie jak w innych systemach operacyjnych w Linuksie wszystkie informacje związane z procesem są przechowywane w bloku kontrolnym procesu (deskryptorze procesu). Jego rozmiar wynosi 1,7 KiB i jest to struktura typu „struct task_struct", w której część pól stanowią wskaźniki na inne struktury. Pamięć na deskryptor jest przydzielana przez alokator plastrowy[1]. W starszych wersjach Linuksa deskryptor był przechowywany na końcu stosu procesu przeznaczonego dla wywołań systemowych i znajdującego się w przestrzeni jądra. Od wersji 2.6 na stosie przechowywana jest struktura typu „struct thread_info". Typ tej struktury jest zdefiniowany w pliku nagłówkowym <asm/thread_info.h> następująco[2]:

```
struct thread_info {
      struct task_struct *task;
      struct exec_domain *exec_domain;
      __u32  flags;
      __u32 status;
      __u32 cpu;
      int  preempt_count;
      mm_segment_t addr_limit;
      struct restart_block restart_block;
      void __user *sysenter_return;
 #ifdef CONFIG_X86_32
      unsigned long previous_esp;
      __u8 supervisor_stack[0];
 #endif
     int uaccess_err;
};
```

W jądrze zdefiniowana jest markodefinicja o nazwie „current", która pozwala na szybki dostęp do deskryptora bieżącego procesu. W przypadku 32-bitowych procesorów o architekturze x86 działanie tej makrodefinicji składa się z dwóch etapów. Najpierw za pomocą funkcji „current_thread_info()" ustalany jest adres struktury „thread_info" bieżącego procesu, poprzez zamaskowanie 13 najmniej znaczących bitów wskaźnika stosu:

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

Następnie z tej struktury jest otrzymywany wskaźnik na deskryptor bieżącego zadania:

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

Procesy użytkownika wykonują się w przestrzeni użytkownika. Aby móc skorzystać z usług jądra mogą one wywołać którąś z funkcji zdefiniowanych w bibliotece (g)libc, które stanowią interfejs dla wywołań systemowych. Jeśli dojdzie do realizacji takiego wywołania, to mówimy, że jądro wykonuje kod w kontekście procesu. W takim kontekście możliwe jest korzystanie z makrodefinicji „current". Po zakończeniu wywołania działanie procesu jest wznawiane, chyba, że w systemie pojawił się proces o wyższym priorytecie.

Linux, podobnie jak inne systemy uniksowe, przypisuje każdemu procesowi unikatowy identyfikator określany skrótem PID. Wartość tego identyfikatora jest przechowywana w polu „pid" deskryptora. Typ tego pola nazywa się „pid_t", a jego implementacja jest zależna od architektury procesora, na którym jądro pracuje. Ze względu na kompatybilność z innymi Uniksami pole to może przyjmować jako wartości liczby całkowite domyślnie od 0 do 32767, choć zawsze możliwe jest zwiększenie tego zakresu.

W deskryptorze procesu, w polu „state" jest przechowywany stan zadania. To pole może przyjmować wartość odpowiadającą jednej spośród wymienionych[3] stałych:

- TASK_RUNNING – proces jest gotów do uruchomienia, ten stan nie oznacza jednak, że w danej chwili proces jest wykonywany,
- TASK_INTERRUPTIBLE – proces został zawieszony i oczekuje na jakieś zdarzenie, może on jednak zostać ustawiony w stan gotowości przez inne zdarzenie niż to, na które oczekiwał,
- TASK_UNINTERRUPTIBLE – podobnie jak wyżej, ale proces nie będzie reagował na żadne inne zdarzenia niż to, na które oczekuje, stan ten stosowany jest rzadko, gdyż uniemożliwia np.: usunięcie procesu z systemu,
- TASK_KILLABLE – proces zareaguje na wystąpienie oczekiwanego przez niego zdarzenia lub na sygnał, który spowoduje jego zakończenie (ang. *fatal signals*), ten stan został zdefiniowany przez twórców jądra po to, aby zastąpić nim w większości przypadków TASK_UNINTERRUPTIBLE,
- TASK_STOPPED – działanie procesu (zadania) zostało wstrzymane po otrzymaniu odpowiedniego sygnału,
- TASK_TRACED – zadanie zostało wstrzymane celowo, przez inny proces (najczęściej debugger), który śledzi jego wykonanie.

Stan procesów, które zostały zakończone przechowywany jest w polu „exit_state" deskryptora. Może ono przyjmować następujące wartości:

- EXIT_ZOMBIE[4] – proces zakończył się, w systemie został jego deskryptor, który jest usuwany przez wywołanie o nazwie ,,wait4()" uruchamiane przez proces macierzysty,
- EXIT_DEAD – proces zakończył się, uruchomiono dla niego wywołanie „wait4()", ale jądro nie zakończyło jeszcze jego usuwania z systemu; ten stan ma znaczenie w sytuacji, gdy proces macierzysty jest wielowątkowy i wszystkie jego wątki wywołują „wait4()" dla tego samego procesu potomnego.

1 Działanie alokatora plastrowego zostanie omówione na wykładzie poświęconym zarządzaniu pamięcią.
2 Ta postać typu tej struktury dotyczy wyłącznie procesorów z rodziny x86 (zarówno 32-bitowych, jak i 64-bitowych), w przypadku innych procesorów może być ona inna.
3 W niektórych wersjach jądra pojawiały się (lub znikały) inne wartości stanów. Obecnie te stałe pełnią rolę argumentów dla funkcji zmieniających stan procesu.
4 We wcześniejszych wersjach jądra ten stan był przechowywany w polu „state" i określony stałą TASK_ZOMBIE.

Stan procesu może zostać zmieniony za pomocą funkcji „set_task_state()". Jeśli zmiana ma dotyczyć stanu bieżącego procesu, to można użyć funkcji „set_current_state()".

Procesy w systemach uniksowych tworzą hierarchię, na której szczycie znajduje się proces o nazwie „init", o PID równym 1. Pozostałe procesy są jego potomkami. Deskryptor każdego procesu zawiera wskaźnik o nazwie „parent" na proces macierzysty, oraz listę wskaźników o nazwie „children" na deskryptory procesów potomnych. Aby uzyskać wskaźnik na deskryptor rodzica bieżącego zadania trzeba wykonać następujący kod:

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

Aby przejrzeć listę procesów potomnych należy wykonać:

```
sturct task_struct *task;
struct list_head *list;

list_for_each(list, &current->children) {
    task = list_entry(list, struct task_struct, sibling);
}
```

Deskryptory procesów są powiązane ze sobą w dwukierunkową listę procesów. Aby otrzymać wskaźnik na deskryptor poprzedniego lub następnego zadania trzeba użyć odpowiednio makra „prev_task(task)" lub „next_task(task)". Istnieje również makrodefinicja „for_each_process(task)", która pozwala „przejrzeć" wszystkie procesy na liście. Jądro utrzymuje również tablicę o nazwie „pidhash", której indeksy mają wartości z takiego samego zakresu jako pole „pid" deskryptora procesu, natomiast wartościami elementów są wskaźniki na deskryptory procesów. Dzięki tej tablicy można bardzo łatwo otrzymać deskryptor procesu znając jego PID.

Linux, tak jak pozostałe systemy uniksowe pozwala na tworzenie nowych procesów za pośrednictwem funkcji fork() i vfork(). Dodatkowo w Linuksie można utworzyć wątek (lub proces) przy pomocy funkcji clone(). Wszystkie te funkcje korzystają z wywołania systemowego „clone()". Funkcja fork() tworzy nowy proces, który współdzieli z procesem macierzystym obszar kodu, zwany w Uniksach obszarem tekstu, oraz obszar danych. Stosowana jest tu technika „copy-on-write", która pozwala na tak długie współdzielenie przez procesy obszaru danych, dopóki jeden z nich nie zechce do niego zapisywać, wówczas ten obszar jest kopiowany i od tej chwili każdy z procesów dysponuje własnym obszarem danych. Aby nowy proces mógł wykonać nowy kod (program) należy wywołać jedną z funkcji „exec()". Jak już wcześniej zostało wspomniane „fork()" wywołuje „clone()", ono z kolei wywołuje funkcję „do_fork()", która wywołuje funkcję „copy_process()". Do zadań tej ostatniej funkcji należy: utworzenie stosu jądra i deskryptora dla nowego procesu, sprawdzenie, czy powstanie nowego procesu nie wyczerpie limitu zasobów określonego dla użytkownika procesu, ustawienie procesu w stan TASK_UNINTERRUPTIBLE, ustawienie flag procesu, pozyskanie dla niego numeru PID, skopiowanie z uwzględnieniem znaczników przekazanych do „clone()" struktur związanych z zasobami i obsługą sygnałów oraz zwrócenie wskaźnika na deskryptor nowego procesu. Po wykonaniu tej funkcji następuje powrót do „do_fork()", która uruchamia proces potomka jako pierwszy. Funkcja vfork() zakłada, że proces potomny wywoła natychmiast po powstaniu funkcję „exec()" i wstrzymuje wykonanie procesu macierzystego. Funkcja ta jest obecna w systemie ze względów związanych z kompatybilnością, ale korzystanie z niej nie jest zalecane.

W przeciwieństwie do wielu innych systemów operacyjnych Linux obsługuje wątki w ten sam sposób jak zwykłe procesy. Zarówno jedne jak i drugie są tworzone przez wywołanie systemowe „clone()". Różnica polega tylko na tym jakie parametry są przekazywane dla tego wywołania. Poniżej przedstawiono większość możliwych wartości tych parametrów:

CLONE_CLEARTID – zeruje wartość TID (Thread Identifier),

CLONE_DETACHED – proces macierzysty nie wysyła przy kończeniu działania sygnału SIGCHLD,

CLONE_FILES – procesy współużytkują otwarte pliki,

CLONE_FS – procesy współużytkują informacje o systemie plików,

CLONE_IDLETASK – zeruje wartość PID dla zadań jałowych,

CLONE_NEWNS – tworzona jest nowa przestrzeń nazw, dla procesu potomnego,

CLONE_PARENT – proces potomny ma PID rodzica takie samo jak jego rodzic (oba procesy mają to samo PID rodzica),

CLONE_PTRACE – proces potomny będzie śledzony, tak jak proces macierzysty,

CLONE_SETTID – zapisuje wartość TID do przestrzeni użytkownika,

CLONE_SETTLS – tworzony jest nowy obszar TLS (Thread-Local Storage – prywatny obszar danych wątku), dla procesu potomnego,

CLONE_SIGHAND – kopiowane są do procesu potomnego procedury obsługi sygnałów,

CLONE_SYSVSEM – oba procesy korzystają semantyki operacji SEM_UNDO charakterystycznej dla wersji System V,

CLONE_THREAD – oba procesy będą w tej samej grupie wątków,

CLONE_VFORK – powoduje, że wywołanie „clone()" zachowuje się jak „vfork()",

CLONE_VM – powoduje, że procesy będą współdzieliły przestrzeń adresową.

Również jądro może tworzyć wątki, jeśli musi wykonać jakąś pracę w tle. Przykładami takich wątków są „ksoftirqd" i „kworker". Wątki jądra nie mają własnej przestrzeni adresowej, działają w obrębie przestrzeni jądra. W wersji 2.6.0 jądra są one tworzone za pomocą wywołania funkcji o nazwie „kernel_thread()". Zazwyczaj wątki jądra w pętli wykonują okresowo jakąś czynność, a więc nie kończą się aż do chwili przeładowania (ang. *reboot*) lub wyłączenia systemu (ang. *shutdown*) albo wtedy, gdy z jądra systemu są usuwane moduły, z którymi są związane. Rusty Russell zaproponował poprawkę do jądra, która wprowadzała szereg udogodnień w obsłudze wątków jądra. Ta poprawka została przyjęta do wersji 2.6.1. Wprowadza ona kilka nowych funkcji i makr. Makro „kthread_create()" tworzy nowy wątek i zwraca wskaźnik do jego deskryptora. Wątek nie jest po utworzeniu aktywny. Makro „kthread_run()" jest używane i działa podobnie jak „kthread_create()", ale

dodatkowo uruchamia nowo powstały wątek. Funkcja „kthread_stop()" przerywa wykonanie wątku jądra. Funkcja „kthread_should_stop()" wywoływana jest wewnątrz pętli wątku i sprawdza, czy ten wątek otrzymał sygnał, który nakazuje mu zakończenie. Funkcja „kthread_bind()" jest przydatna w systemach wieloprocesorowych. Jej zadaniem jest powiązanie wątku jądra z określonym procesorem (wątek będzie się wykonywał wyłącznie na tym procesorze). Opisana poprawka nie blokuje możliwości tworzenia wątku z użyciem „kernel_thread()", jednak zalecane jest korzystanie z nowych funkcji.

Proces kończy się wywołując (jawnie, bądź niejawnie) funkcję „exit()", która z kolei korzysta z wywołania „_exit()", a to uruchamia funkcję „do_exit()". Ta funkcja jest odpowiedzialna za zwolnienie większości struktur związanych z procesem i powiadomienie procesu macierzystego o zakończeniu procesu potomnego. W systemie pozostaje jedynie deskryptor procesu i stos jądra wraz ze strukturą „struct thread_info". Te dwie struktury są zwalniane przez funkcję „release_task()", z której korzysta wywołanie „wait4()". Funkcja ta zmniejsza wartość licznika procesów należących do użytkownika, usuwa deskryptor procesu z tablicy haszującej „pidhash" i z listy zadań, usuwa go także z listy procesów śledzonych (o ile proces był śledzony) oraz zwalnia pamięć przydzieloną na struktury „struct task_struct" i „struct thread_info". Jeśli proces macierzysty zakończy się przed procesem potomnym, to nie wywoła dla niego „wait4()" i ten ostatni pozostanie w stanie zombie. Aby uniknąć takiej sytuacji przydzielany jest mu nowy rodzic, który należy do tej samej grupy procesów co poprzedni rodzic lub jest procesem „init". Przydział ten wykonuje funkcja „forget_original_parent()" wywoływana w ramach „do_exit()". Od wersji 2.6 jądra przegląda ona dwie listy procesów potomnych: listę procesów zwykłych i listę procesów śledzonych.