Wykład czternasty

Przestrzeń adresowa procesu w Linuksie

Jądro systemu Linux zarządza zarówno własną przestrzenią adresową, jak i przestrzenią adresową poszczególnych procesów, które są pod jego kontrolą uruchomione. Każdy z tych procesów otrzymuje płaską (liniową) przestrzeń adresową, która jest domyślnie inna dla każdego z nich, choć istnieje możliwość jej współdzielenia. Przestrzeń ta podzielona jest na interwały adresów, do których proces ma prawo dostępu i które nazywane są krótko obszarami pamięci. Jądro systemu daje procesom możliwość dynamicznego włączania do swojej przestrzeni adresowej nowych obszarów. Żaden z procesów nie może odwołać się poza obszary pamięci, które zostały mu wyznaczone. Jeśli to nastąpi, to jest on kończony w trybie awaryjnym z komunikatem "Segmentation Fault". Istnieją następujące typy obszarów pamięci:

- sekcja tekstu programu – zawiera odwzorowany w pamięci kod z pliku wykonywalnego (obszar tekstu),
- sekcja danych programu – zawiera odwzorowany w pamięci obszar zainicjowanych zmiennych globalnych z pliku wykonywalnego,
- sekcja .bss – jest to odwzorowana w pamięci strona zawierająca zera, przeznaczona na niezainicjowane zmienne globalne,
- stos – jest to również wyzerowana strona pamięci, przeznaczona na stos przestrzeni użytkownika,
- obszary z odwzorowanymi w pamięci plikami,
- współużytkowane obszary pamięci (stanowiące realizację koncepcji pamięci współdzielonej),
- anonimowe odwzorowania pamięci, przydzielone np. za pośrednictwem funkcji *malloc()*.

Sekcje tekstu, danych i .bss tworzone są nie tylko dla procesu, ale również dla każdej z bibliotek współdzielonych (*ang. shared objects*), z których korzystają procesy. Obszary pamięci nie nakładają się na siebie. Informacje na temat przestrzeni adresowej pojedynczego procesu przechowuje jego deskryptor pamięci. Jest to struktura typu *struct mm_struct* zdefiniowana w tym samym pliku nagłówkowym, co deskryptor procesu. Zawiera ona kilkanaście pól, które przechowują między innymi takie informacje jak: adresy startowe sekcji tekstu, danych, stosu, adresy końcowe tych sekcji, adresy początkowe i końcowe obszarów argumentów wywołania programu i zmiennych środowiskowych oraz inne. Jeśli pole *mm_count* jest równe „1”, to oznacza to, że przestrzeń adresowa związana z deskryptorem jest wykorzystywana przez kilka procesów, które są wątkami. Dokładna liczba tych procesów jest przechowywana w polu *mm_users*. Pole *task_size* określa rozmiar przestrzeni adresowej procesu. Dodano je w nowszych wersjach jądra, aby umożliwić wykonywanie na sprzęcie 64-bitowym 32-bitowych aplikacji. Dwa pola deskryptora związane są ze strukturami przechowującymi te same informacje, ale w odmienny sposób. Pierwszym z tych pól jest pole *mmap*, które zawiera wskaźnik na listę przechowującą informacje o wszystkich obszarach w pamięci, drugie *mm_rb* zawiera wskaźnik na korzeń drzewa czerwono – czarnego, przechowującego te same informacje. Lista pozwala na proste przeglądanie jej elementów, natomiast drzewo na szybkie wyszukiwanie żądanego elementu[1]. Deskryptory połączone są w listę dwukierunkową, której pierwszym elementem jest deskryptor procesu *init*. Adres deskryptora pamięci jest przechowywany w polu *mm* deskryptora procesu, którego przestrzeń adresową opisuje. Jeśli proces tworzy potomka, to zawartość jego deskryptora pamięci jest kopiowana dla procesu potomnego przy pomocy funkcji *copy_mm()* do obszaru pamięci przydzielonego z plastra dedykowanej pamięci podręcznej. Alokację tę wykonuje makrodefinicja *allocate_mm*. Jeśli wywołanie *clone()* zawierało parametr CLONE_VM, to obydwa procesy będą współdzieliły przestrzeń adresową i będą wątkami. Oznacza to, że w takim przypadku dla nowego procesu nie jest tworzony osobny deskryptor pamięci. Kiedy proces lub wątek kończy swoje działanie, to wtedy wykonywana jest funkcja *exit_mm()*, która aktualizuje statystyki i wykonuje pewne czynności porządkowe, a następnie wywołuje funkcję *mmput()*, która zmniejsza licznik użytkowników *mm_users* deskryptora pamięci. Jeśli to pole osiągnie wartość zero, to wywoływana jest funkcja *mmdrop()*, zmniejszająca wartość licznika *mm_count*. Jeśli i on osiągnie wartość zero, to deskryptor jest zwalniany poprzez wywołanie funkcji *free_mm()*.

Wątki jądra nie mają własnej przestrzeni adresowej i własnego deskryptora pamięci. Pola *mm* deskryptorów procesów takich wątków mają wartość NULL. Jednak te wątki również muszą odwoływać się do pamięci operacyjnej, aby móc wykonywać swoje zadania, dlatego też korzystają z deskryptorów pamięci poprzednio zaszeregowanych procesów użytkownika. Adresy tych deskryptorów pamięci są przechowywane w polu *active_mm*[2] deskryptorów wątków (typu *struct task_struct*). Takie rozwiązanie jest możliwe dlatego, że wszystkie procesy widzą przestrzeń adresową jądra w ten sam sposób[3], a wątki jądra odwołują się tylko do niej.

Podsystem zarządzania obszarami wirtualnej pamięci, w skrócie VMA (*ang. virtual memory areas*) jest oparty, podobnie jak wirtualny system plików, na modelu obiektowym. Typ (klasa) takich obiektów jest określony strukturą *vm_area_struct*. Obiekty te oprócz „zwykłych” danych przechowują wskaźnik na strukturę wskaźników do funkcji realizujących operacje na danym obszarze pamięci. Pola *vm_start* i *vm_end* zawierają adres początkowy i końcowy obszaru pamięci, pole *vm_flags* zawiera znaczniki określające własności i zachowanie stron wchodzących w skład obszaru. Do tych znaczników należą między innymi: VM_READ, VM_WRITE, VM_EXEC, oznaczające odpowiednio obszar do odczytu, zapisu i wykonania, VM_SHARED – znacznik obszaru współdzielonego, VM_IO – obszar pamięci, w którym odwzorowane są rejestry urządzeń wejścia – wyjścia, VM_LOCKED – strony w obszarze nie podlegają wymianie, VM_SEQ_READ, obszar pamięci, gdzie odwzorowany jest plik, na którym wykonywany jest odczyt z wyprzedzeniem, VM_RAND_READ – obszar pamięci, na którym jest odwzorowywany plik, na którym jest wykonywany odczyt swobodny, dla takiego trybu odczytu ogranicza się czytanie z wyprzedzeniem lub wręcz w ogóle się go nie stosuje. Wskaźniki do funkcji realizujących operacje związane z danym obszarem są umieszczane w tablicy operacji reprezentowanej strukturą typu *vm_operations_struct*. Do tych operacji należą między innymi: *open()* - funkcja wywoływana w momencie dodawania obszaru pamięci do przestrzeni adresowej, *close()* - funkcja wywoływana podczas usuwana obszaru z przestrzeni adresowej, *fault()* - wywoływana w wyniku błędu strony, kiedy strona istnieje, ale nie jest obecna w pamięci operacyjnej, *page_mkwrite()* - również wywoływana w następstwie błędu strony, ale wówczas, gdy strona, która była tylko do odczytu staje się stroną także do zapisu, *access*() - funkcja wywoływana, gdy zachodzi konieczność dostępu do przestrzeni adresowej określonego procesu. We wcześniejszych wersjach jądra dostępna była także funkcja *populate()*, która w nowszych jądrach została usunięta. Funkcja *fault()* zastąpiła z kolei funkcję *nopages()*.

Obiekty opisujące obszary pamięci umieszczane są równocześnie na liście i w drzewie czerwono – czarnym. Lista zawierająca te obiekty jest posortowana rosnąco pod względem adresów jakie w niej występują. Drzewo czerwono – czarne jest wyważonym drzewem BST. Z każdym elementem tego drzewa związany jest kolor. Każdy węzeł jest więc albo czerwony albo czarny. Dodatkowo, liście tego drzewa są czarne, a każdy czerwony węzeł musi mieć czarnych potomków. Wszystkie proste ścieżki do dowolnego, ustalonego liścia drzewa czerwono – czarnego zawierają tyle samo węzłów czarnych. Jeśli któraś z tych własności nie jest spełniona, to na elementach drzewa są wykonywane operacje, które ją przywracają. Wszystko to dzieje się po to, aby czas wykonywania operacji na elementach drzewa czerwono – czarnego zawsze wynosił O(lg n).

Informacje o obszarach pamięci danego procesu można uzyskać z pliku */proc/<pid>/maps*, gdzie *<pid>* oznacza numer PID procesu. Z analizy tego pliku wynika, że obszary zawierające sekcje tekstu i danych tylko do odczytu mogą być współdzielone zarówno przez procesy, jak i również biblioteki ładowane dynamicznie.

Drzewo czerwono – czarne jest wykorzystywane przez funkcję *find_vma()*, której zadaniem jest znalezienie obszaru pamięci, w którym leży podany jej przez parametr adres lub obszaru zaczynającego się od adresu większego. Jeśli takiego obszaru nie znajdzie, to zwraca wartość NULL, w przeciwnym przypadku zwraca adres struktury opisującej ten obszar. Podobnie działa funkcja *find_vma_prev()*, która zwraca adres obiektu opisującego obszar leżący przed zadanym adresem. Ostatnią z funkcji umożliwiających wyszukiwanie jest *find_vma_intersection()*, która zwraca adres obiektu opisującego pierwszy obszar pokrywający się choć częściowo z zadanym interwałem adresów.

Obszary pamięci są dodawane do przestrzeni adresowej procesu za pomocą funkcji *do_mmap()*. Taka funkcja może stworzyć nowy obszar pamięci lub rozszerzyć już istniejący. Jej zadaniem jest odwzorowanie fragmentu pliku w pamięci. Jeśli argument tej funkcji, określający strukturę opisującą plik jest równy NULL, to zastosowane zostanie odwzorowanie anonimowe (strona będzie wypełniona zerami), a nie odwzorowanie plikowe. Funkcja ta jest uruchamiana poprzez wywołania *mmap2()* oraz *mmap()*. To pierwsze określa offset miejsca w pliku, które ma zostać odwzorowane w stronach, a nie bajtach. Do usuwania interwałów adresów służy funkcja *do_munmap()*, która jest wywoływana przez wywołanie systemowe *munmap()*, będące po prostu opakowaniem wymienionej funkcji.

1 W jądrze Linuksa serii 2.0 obszary były opisywane za pomocą listy, jeśli było ich stosunkowo mało, po przekroczeniu liczby obszarów określonej pewną stałą informacje o obszarach były równocześnie przechowywane na liście i drzewie AVL.

2 Pole to w przypadku procesów zwykle jest wykorzystywane, kiedy proces zaczyna realizować nowy program.

3 Od wersji 4.15, w której wprowadzono poprawkę KPTI jest to tylko częściowo prawdą – w trybie jądra wykorzystywana jest inna tablica stron niż w trybie użytkownika.