Systemy Operacyjne 2

Przestrzeń Adresowa Procesów

Arkadiusz Chrobot

Katedra Systemów Informatycznych

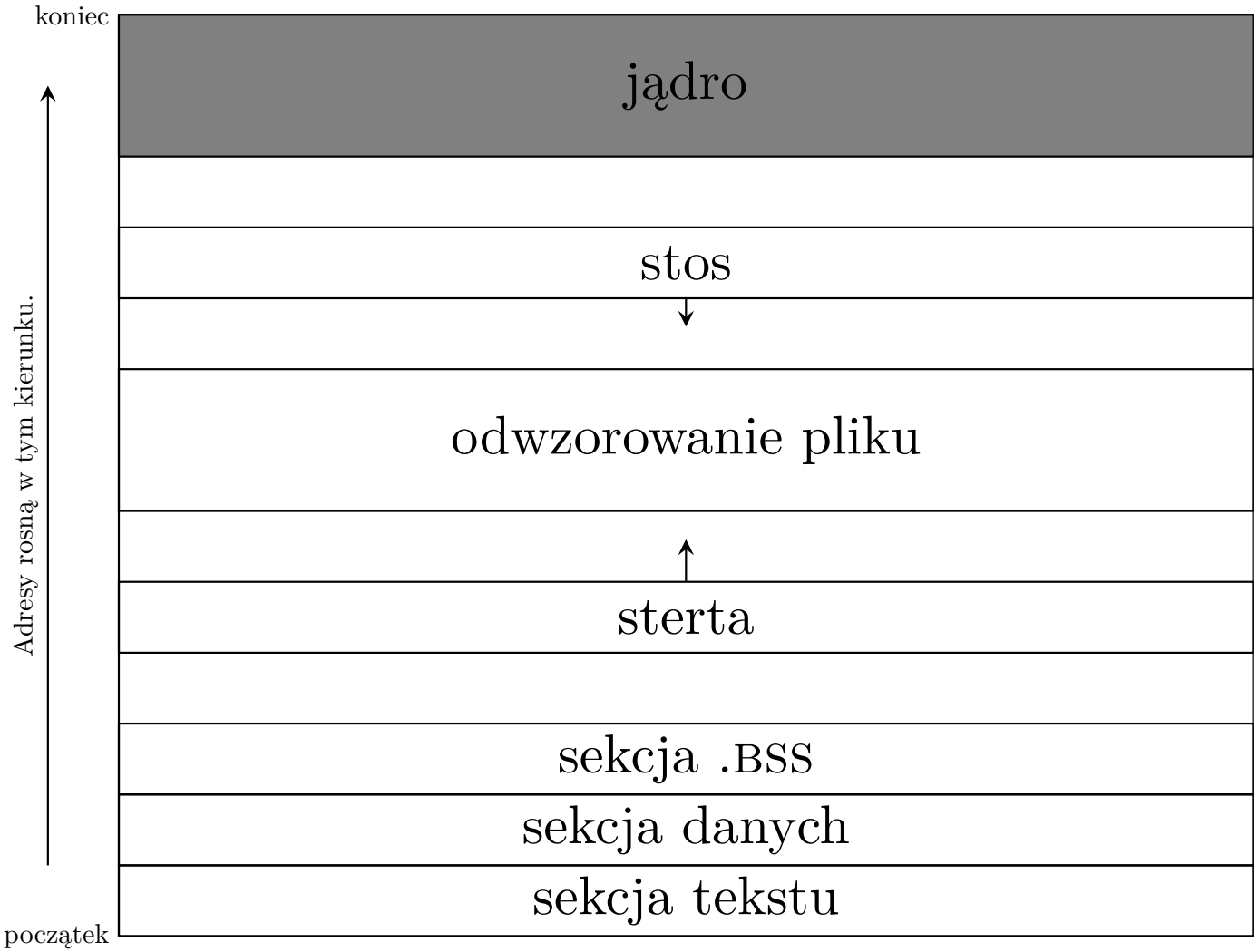
13 czerwca 2024

Plan

Wprowadzenie

Jądro Linuksa zarządza nie tylko swoją *przestrzenią adresową*, ale również przestrzeniami adresowymi procesów użytkownika. Każdy z nich otrzymuje *płaską* (*liniową*) przestrzeń, która domyślnie jest odseparowana od przestrzeni adresowych innych procesów. Oznacza to, że proces nie może odczytywać lub modyfikować danych innych procesów, nawet jeśli posługuje się tymi samymi adresami wirtualnymi co one. Jądro Linuksa umożliwia jednak procesom *współdzielenie* ich przestrzeni adresowych, jeśli one tego zażądają. W ten sposób są zaimplementowane *wątki użytkownika*. Każda przestrzeń adresowa procesu jest podzielona na *interwały* adresów nazwane *obszarami pamięci* (Rysunek 1). Każdy proces może zażądać od jądra dodania nowego obszaru pamięci do jego przestrzeni adresowej, ale nie może odwoływać się do nieistniejących obszarów pamięci lub naruszać praw dostępu (odczyt, zapis, wykonanie), do tych które posiada. W przeciwnym przypadku jądro przerwie działanie takiego procesu, a jego użytkownik zobaczy na ekranie komunikat „Segmentation fault”.

Organizacja przestrzeni adresowej procesu



Rysunek 1: Uproszczony model przestrzeni adresowej w Linuksie

Organizacja przestrzeni adresowej procesu

Obszary pamięci

Istnieje kilka typów obszarów pamięci:

sekcja tekstu jest to odwzorowana w pamięci część pliku wykonywalnego, która zawiera kod wykonywany przez proces,

sekcja danych jest to odwzorowana w pamięci część pliku wykonywalnego, która zawiera zainicjowane zmienne globalne,

sekcja .BSS to obszar pamięci, gdzie jest odwzorowana wyzerowana strona, na potrzeby niezainicjowanych zmiennych globalnych,

stos w tym obszarze jest zorganizowany stos w przestrzeni użytkownika; początkowo jest to odwzorowana wyzerowana strona pamięci,

obszary pamięci współdzielonej obszary pamięci będące realizacją pamięci współdzielonej,

obszary odwzorowanych plików obszary pamięci w których są odwzorowane pliki,

Przestrzeń adresowa procesu

Obszary pamięci

anonimowe odwzorowania pamięci obszary pamięci przydzielone, np. z użyciem funkcji `malloc()`.

Skrót .BSS oznacza „block started by symbol” i jest nazwą używaną ze względów historycznych. Sekcja danych zawiera zmienne globalne, o wartościach różnych od zera, przechowywanych w pliku wykonywalnym. Stąd nazwa tych zmiennych: „zainicjowane”. Sekcja .BSS zawiera zmienne globalne, których wartością początkową jest zero. Ich wartości nie są przechowywane w wykonywalnym pliku i dlatego są one nazywane „niezainicjowanymi”.

Sekcje tekstu, danych i .BSS występują także dla bibliotek ładowanych dynamicznie, nazywanych w systemach kompatybilnych z Uniksem, *bibliotekami współdzielonymi* lub *obiektami współdzielonymi* (ang. *shared objects*).

Każdy obszar pamięci ma zdefiniowane osobne uprawnienia. Obszary te nie nakładają się.

Deskryptor pamięci

Informacje o przestrzeni adresowej pojedynczego procesu są przechowywane w jego *deskryptorze pamięci*. Jest to struktura typu `struct mm_struct`, zdefiniowanym w tym samym pliku co typ deskryptora procesu. Deskryptor pamięci zawiera wiele pól, między innymi pola przechowujące początkowe i końcowe adresy sekcji tekstu, danych, stosu, obszaru pamięci przechowującego argumenty wiersza poleceń i obszaru pamięci przechowującego zmienne środowiskowe. Jeśli wartość pola `mm_count` wynosi 1, to przestrzeń adresowa opisywana przez deskryptor pamięci jest współdzielona przez co najmniej dwa procesy, które są wątkami. Dokładna liczba tych wątków jest przechowywana w polu `mm_users`. Pole `task_size` określa rozmiar przestrzeni adresowej. Zostało ono dodane aby umożliwić wykonywanie 32-bitowych aplikacji na 64-bitowych platformach sprzętowych. Dwa pola deskryptora pamięci są związane z osobnymi strukturami danych, które przechowują tę samą informację, ale w różny sposób. Pierwszym z nich jest pole `mmap` przechowujące adres listy zawierającej dane o wszystkich obszarach pamięci.

Deskryptor pamięci

Drugim jest pole `mm_rb` przechowujące adres korzenia drzewa czerwono-czarnego, które zawiera te same dane co lista, ale oferuje krótszy czas ich wyszukiwania, niż lista. Za to listę prościej jest przeglądać sekwencyjnie. W serii 2.0 jądra Linuksa dane o obszarach pamięci były przechowywane w liście tak długo, jak długo liczba tych obszarów była poniżej 20. Jeśli przekroczyła ona ten próg, to lista była zamieniana w drzewo AVL.
Jądro łączy wszystkie deskryptory pamięci w dwukierunkową listę liniową, rozpoczynającą się od deskryptora pamięci procesu `init` (lub jego odpowiednika). Adres deskryptora pamięci jest również przechowywany w polu `mm` deskryptora procesu, który jest właściciele przestrzeni adresowej opisywanej przez ten deskryptor pamięci. Kiedy tworzony jest nowy proces, to przydzielany jest mu, z użyciem alokatora plastrowego, nowy deskryptor pamięci i wywoływana jest funkcja `copy_mm()`, kopiująca zawartość deskryptora pamięci rodzica do deskryptora pamięci potomka.

Deskryptor pamięci

Jeśli wywołanie systemowe `clone()` otrzyma flagę `CLONE_VM` jako jeden z jego argumentów, to nowy proces będzie dzielił przestrzeń adresową z rodzicem. Innymi słowy, te procesy będą wątkami. W takim przypadku nie jest tworzony deskryptor pamięci dla procesu potomnego, a oba procesy współdzielą ten sam.
Kiedy proces lub wątek kończy pracę, to wywoływana jest funkcja `exit_mm()`, która aktualizuje określone statystyki, wykonuje pewne operacje porządkujące (ang. *cleanup*) i wywołuje funkcję `mmput()`, która zmniejsza o jeden wartość pola `mm_users` w deskryptorze pamięci. Jeśli ta wartość spadnie do zera, to dodatkowo wywoływana jest funkcja `mmdrop()`, która dekrementuje wartość pola `mm_count` tego deskryptora. Jeśli i ta wartość osiągnie zero, to deskryptor pamięci jest usuwany przy pomocy funkcji `free_mm()`.

Deskryptor pamięci

Wątki przestrzeni jądra, lub po prostu wątki jądra, nie mają swojej własnej przestrzeni adresowej, współdzielą ją z jądrem. W związku z tym nie mają również deskryptorów pamięci. Wartość pola `mm` w ich deskryptorach procesów wynosi `NULL`. Jednakże, aby się mogły wykonywać, wątki jądra muszą odwoływać się do pamięci. W tym celu korzystają z deskryptora pamięci procesu użytkownika, który korzystał z CPU tuż przed nimi. Deskryptor pamięci każdego procesu przechowuje informacje o przestrzeni jądra na potrzeby wywołań systemowych. Te dane są takie same dla wszystkich procesów, ale od czasu wydania wersji 4.15, w trybie jądra CPU stosowane są inne tablice stron, niż w trybie użytkownika. Ta zmiana została wprowadzona w ramach poprawki KPTI, aby utrudnić wykorzystanie podatności Meltdown (https://meltdownattack.com/).

Deskryptor pamięci

Adres deskryptora pamięci ostatniego procesu użytkownika, który korzystał z CPU jest przechowywany w polu `active_mm` deskryptora procesu wątku jądra. W przypadku procesów użytkownika, jądro używa tego pola, kiedy proces zaczyna realizować inny program, którego kod jest ładowany z pliku wykonywalnego.

Zarządzanie obszarami pamięci wirtualnej

Kod podsystemu zarządzania *obszarami pamięci wirtualnej* (VMA) został opracowany przy użyciu programowania zorientowanego obiektowo. Każdy obszar pamięci wirtualnej jest reprezentowany przez obiekt będący strukturą typu `vm_area_struct`. Poza „zwykłymi" polami ta struktura posiada również takie, które jest wskaźnikiem na strukturę, której pola są wskaźnikami na funkcje wykonujące określone operacje na obszarach pamięci wirtualnej. Te funkcje są zatem metodami, a ostatnia ze wspomnianych struktur tablicą metod. Pola `vma_start` i `vma_end` obiektu obszaru pamięci wirtualnej przechowują adres początkowy i końcowy tego obszaru. Pole `vm_flags` przechowuje flagi określające właściwości i przeznaczenie stron tworzących ten obszar. Wśród nich są: `VM_READ`, `VM_WRITE`, `VM_EXEC` — określają, że obszar pamięci może być odczytywany, zapisywany lub wykonywany, `VM_SHARED` — oznacza współdzielony obszar pamięci, `VM_IO` — oznacza obszar pamięci, w którym są odwzorowane rejestry wejścia-wyjścia urządzenia, `VM_LOCKED` — określa obszar pamięci, którego strony nie podlegają wymianie, `VM_SEQ_READ` — określa

Zarządzanie obszarami pamięci wirtualnej

obszar pamięci, gdzie odwzorowany jest plik umożliwiający jedynie odczyt sekwencyjny; w takim przypadku jądro może odczytywać z niego dane z wyprzedzeniem, tzn. zanim proces użytkownika ich zażąda, aby podnieść wydajność odczytu pliku, `VM_RAND_READ` — określa obszar pamięci, gdzie jest odwzorowany plik umożliwiający zarówno odczyt sekwencyjny, jak i bezpośredni; w takim wypadku jądro nie stosuje odczytu z wyprzedzeniem, bo nie przynosi on efektu. Tablica metod obiektu obszaru pamięci wirtualnej jest strukturą typu `vm_operations_struct`. Posiada ona kilka pól wskazujących na funkcje wykonujące operacje na reprezentowanym przez obiekt obszarze pamięci, takie jak: `open()` — wywoływana, gdy nowy obszar pamięci wirtualnej jest dodawany do przestrzeni adresowej procesu, `close()` — wywoływana, gdy obszar pamięci wirtualnej jest usuwany z przestrzeni adresowej procesu, `fault()` — wywoływana w ramach obsługi błędu strony, gdy strona istnieje, ale nie jest obecna w RAM, `page_mkwrite()` — wywoływana jest w ramach obsługi błędu strony, przy zamianie strony tylko do odczytu na zapisywaną

Zarządzanie obszarami pamięci wirtualnej

`access()` — metoda jest wywoływana przy pojawieniu się pewnych wyjątków podczas dostępu do przestrzeni adresowej określonego procesu. We wcześniejszych wersjach jądra była jeszcze dostępna metoda `populate()`, ale później ją usunięto. Metoda `fault()` zastąpiła funkcję `nopages()`.

Jak już wspomniano, obiekty obszarów pamięci wirtualnej są połączone w listę i drzewo czerwono-czarne. To drzewo jest wykorzystywane przez funkcję jądra `find_vma()`, która wyszukuje obszar pamięci zawierający adres będący jej argumentem, albo obszar, który rozpoczyna się od większego adresu. Jeśli jej się to nie uda, to zwróci wartość `NULL`, w przeciwnym przypadku zwróci adres obiektu reprezentującego znaleziony obszar. Podobnie funkcja `find_vma_prev()` wyszukuje obszar, który jest umiejscowiony przez adresem przekazanym jako jej argument. Natomiast funkcja `find_vma_intersection()` zwraca adres obiektu reprezentującego obszar pamięci wirtualnej, który choć częściowo pokrywa się z obszarem wyznaczanym przez dwa adresy będące argumentami tej funkcji.

Zarządzanie obszarami pamięci wirtualnej

Dane o wszystkich obszarach pamięci wirtualnej danego procesu są umieszczone w pliku `/proc/<pid>/maps`, gdzie `<pid>` jest PID procesu. Ta sama informacja może zostać wypisana na ekranie, w formie czytelnej dla użytkownika, za pomocą polecenia `pmap`. Z tych danych wynika, że sekcje tekstu i sekcje danych tylko do odczytu mogą być współdzielone zarówno przez procesy, jak i biblioteki współdzielone. Obszar pamięci wirtualnej może zostać utworzony lub poszerzony przy użyciu funkcji jądra `do_mmap()`. Jej głównym zadaniem jest odwzorowanie pliku w pamięci. Jako jeden z argumentów przyjmuje adres obiektu pliku, ale jeśli zamiast niego przekazana zastanie do niej wartość `NULL`, to wykona ona odwzorowanie anonimowe, tzn. odwzoruje w określonym obszarze pamięci stronę z zerami. Ta funkcja jest wywoływana przez wywołania systemowe `mmap2()` i `mmap()`. Pierwsze wymaga podania przesunięcia w odwzorowywanym pliku w stronach, a nie w bajtach. Obszar pamięci wirtualnej może być usunięty przy użyciu funkcji jądra `do_munmap()`, wywoływanej przez wywołanie systemowe `munmap()`.

Pytania

?

Koniec

Dziękuję Państwu za uwagę!