Semestrální práce KIV/OS

corrOSion

Jiří Láska, Václav Löffelmann

2016

1 Segmentace a stránkování

Operační systém corrOSion pracuje v tzv. *long mode*. V tomoto režimu procesoru je vyžadováno zapnuté stránkování. Segmentace je označena za zastaralou, nicméně je potřeba korektně nastavit GDT (Global Descriptor Table).

Pro jednoduchost jsme použili stránky o velikosti 2 MB.

1.1 Stránkování jádra

Jádro má nastavené identické zobrazení virtuální adresy na fyzickou. Tato vlastnost se hodí například v případě manipulace s uživatelskými programy.

1.2 Stránkování programů

Programy mají identicky namapovanou jadernou oblast (0-64 MB). Stránky v této oblasti mají nastavený příznak nepřístupnosti z neprivilegovaného režimu. Přístup do jaderné oblasti programem tedy vyvolá výpadek stránky (Page Fault). Přístupná oblast pro program začíná na 64 MB. Při zavedení programu jsou mu alokovány rovnou dvě stránky. První, která začíná na jeho 64. MB virtuální adresy je vyhrazena pro programový zásobník. Do druhé stránky, začínající na 66. MB virtuální adresy, je nakopírován kód programu, odkud je pak spouštěn. Následující stránky jsou mapovány jako nepřístupné, avšak program má možnost zavolat systémové volání pro alokaci paměti. Při tomto volání se v jádře posune ukazatel obsazené fyzické paměti o alokovanou stránku (tím se vyhradí pro proces) a dále je tato stránka namapována na následující, ještě nealokovanou, stránku virtuální paměti procesu. Proces tedy dostává nefragmentovaný blok paměti.

1.3 Fyzické rozložení paměti

Na obrázku 1.1 je naznačeno fyzické rozdělení paměti. Od adresy 0 začínají oblasti jako *BIOS Data Area*, *OS Boot Sector*, *VGA buffer* a další. Následuje základní nízkoúrovňový kód kernelu (kontrola instrukčních rozšíření, přepnutí do long modu), vyhrazená paměť pro stránkování kernelu a uživatelských procesů. Pro jednoduchost je alokována rovnou paměť pro pevný počet uživatelských procesů. Pro každý proces (i jádro) je vyhrazeno místo pro tři tabulky stránkování. Naše jádro totiž podporuje jenom jednu tabulku na každé úrovni stránkování. Což, nicméně, při stránkách o velikosti

2 MB dává procesům až 1 GB přístupné paměti. To pro naše účely považujeme za dostatečné. Následuje ještě jaderný zásobník, tabulka deskriptorů a ostatní vysokoúrovňový kód.

Na adresách od 64. MB se nachází kódy, zásobníky a data uživatelských procesů, jak je na obrázku 1.1 znázorněno.

2 Systémová volání

Náš jednoduchý operační systém poskytuje uživatelským procesům základní systémová volání.

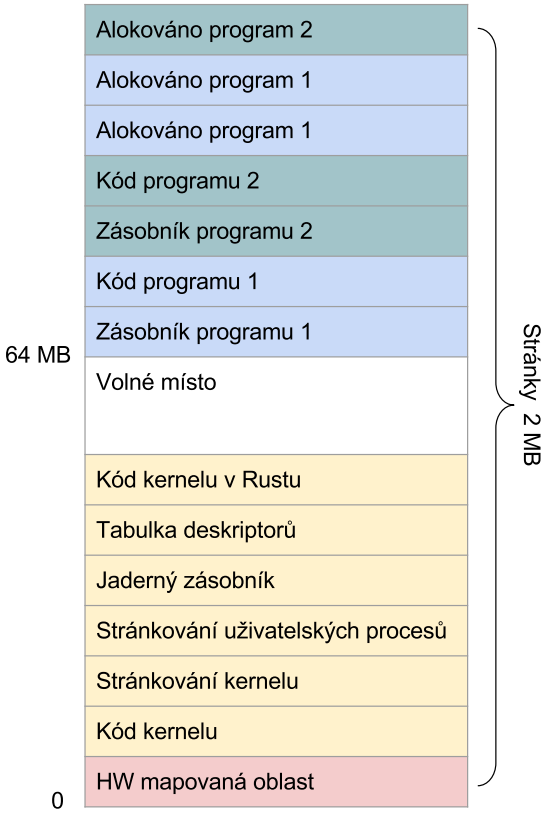
2.1 Volací konvence

Aby uživatelský proces zavolal systémové volání, musí nastavit, v závislosti na volání, minimálně tři parametry. Těmito parametry jsou minimálně:

1. Číslo systémového volání, předávané v registru `r13`.
2. Ukazatel na instrukci v programu před systémovým voláním, když chce uživatelský program zachovat tok programu. V případě, že uživatelský program chce skočit na jinou instrukci, tak ukazatel nastaví na 2 byty před tuto jinou instrukci. Tento parametr se předává v registru `r14`.
3. Ukazatel na vrchol zásobníku v uživatelském programu, předaný v registru `r15`.

Pro vlastní vstup do kódu operačního systému používáme instrukci `sysenter`.

Volající má zaručeno, že obsah některých registrů zůstane zachován mezi zavoláním systémového volání a návratem do programu. Tyto registry jsou následující: `r12, r11, r10, r9, r8, rdi, rsi`.



Obrázek 1.1: Schéma fyzického rozdělení paměti

2.2 Příklad systémového volání

Následující příklad systémového volání ukazuje jednoduché volání systémové služby pauzy. Nejdříve jsou nastaveny potřebné registry a poté je zavolána instrukce `sysenter`. Zajímavé je, že x86 nepodporuje přímé kopírování registru `rip` a proto je potřeba použít instrukci `lea`. Po vykonání systémového volání a naplánování

tohoto procesu bude proces pokračovat právě za instrukcí `sysenter`.

```
mov r13, 0x03  //syscall number (3 - pause)
mov r15, rsp   //pass stack pointer to OS
lea r14, [rip] //pass instruction pointer to OS
sysenter
//next instructions
```

2.3 Seznam volání

Tabulka 2.1 obsahuje kompletní výpis dostupných systémových volání. Hodnoty ve sloupci kód jsou čísla, které označují číslo služby, kterým volající určuje, kterou službu chce vyvolat. Názvy volání mají pouze informativní charakter. Příznak přeplánování určuje, zda-li se na konci systémového volání spustí plánovací procedura a tím pádem se dá výpočetní čas jinému procesu. Systémová volání, která nepřeplánovávají navracejí výpočet do původního procesu, ze kterého bylo volání zavoláno. Speciální registry označují další parametry konkrétního systémového volání (kromě povinných parametrů probraných na začátku kapitoly).

Kód	Název	Popis	Přeplánuje	Speciální registry
1	`alloc`	Alokuje pro proces novou paměťovou stránku o velikosti 2 MB. Virtuální adresa nové paměti je předána v registru `r13`. V případě neúspěchu alokace je v návratovém registru hodnota 0.	Ne	-
2	`terminate`	Ukončí uživatelský proces. Vyčistí záznam v PCB a instance tohoto procesu již nebude nikdy plánována.	Ano	-
3	`pause`	Pozastaví uživatelský proces a spustí přeplánování. Pokud není žádný následující proces v kruhové frontě připravených procesů, bude znovu spuštěn tento proces.	Ano	-
4				
5				

Tabulka 2.1: Systémová volání

3 Bezpečnost

V následující kapitole na náš operační systém nahlédneme po stránce bezpečnosti. Vzhledem k tomu, že se jedná o čistě výukový operační systém, neimplementovali jsme všechna bezpečnostní opatření. Nicméně jsme si několika *situací* vědomi.

3.1 Přetečení uživatelského zásobníku

Díky stránkování a označování paměti jako nepřístupné z neprivilegovaného režimu, přetečení uživatelského zásobníku skončí výpadkem stránky. Uživatelský zásobník začíná na konci první alokované stránky (virtuální adresa 66 MB) a *roste* až k 64. MB virtuální adresy. Při přetečení by přetečení chtěl přistoupit do paměti jádra, což korektně skončí výjimkou.

3.2 Přetečení jaderného zásobníku

V jádře je ovšem situace jiná a tzv. *guard page* nebyla implementována. Jádro tedy není schopno detekovat přetečení vlastního zásobníku. Avšak jaderný zásobník není nijak hojně využívaný a má vyhrazený dostatečný prostor.

3.3 Izolace programů

Programy mají oddělené datové prostory, nemohou si tedy číst ani přepisovat daty.

Současná implementace kopíruje začátek programového kódu, o konstantní velikosti, do adresního prostoru programu. Program tedy může přistoupit na kód, který se ve fyzické paměti nachází za jeho kódem. Teoreticky tedy může číst kopii dat následujícího programu.

3.4 Podvrhnutí ukazatele na zásobník při systémovém volání

Jedná se asi o největší zranitelnost tohoto operačního systému. Program může, při systémovém volání, podvrhnout *svoji* adresu vrcholku zásobníku. Jádro na tuto adresu ukládá registry programu. Pokud program podvrhne tuto adresu, může dostat obsah svých registrů na libovolné místo v paměti jádra. Může tedy například přepsat část kódu tak, aby se zavolala procedura programu v privilegovaném režimu. Proti tomuto útoku by se samozřejmě dalo zabránit kontrolováním předávané adresy zásobníku.