Laborator 1

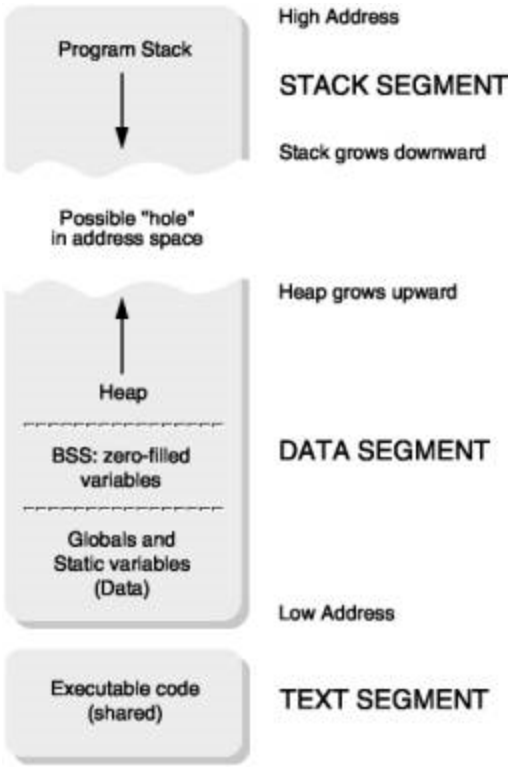
Gestiunea memoriei

Subsistemul de gestiune a memoriei din cadrul unui sistem de operare este folosit de toate celelalte subsisteme: planificator, I/O, sistemul de fișiere, gestiunea proceselor, networking. Memoria este o resursă importantă, de aceea sunt necesari algoritmi eficienți de utilizare și gestiune a acesteia.

Rolul subsistemului de gestiune a memoriei este de:

- **a ține evidența zonelor de memorie fizică (ocupate sau libere)**
- **a oferi proceselor sau celorlalte subsisteme acces la memorie**
- **a mapa paginile de memorie virtuală ale unui proces (pages) peste paginile fizice (frames).**

Spațiul de adresă al unui proces



Spațiul de adresă al unui proces, sau, mai bine spus, spațiul virtual de adresă al unui proces reprezintă zona de memorie virtuală utilizabilă de un proces. Fiecare proces are un spațiu de adresă propriu. Chiar în situațiile în care două procese partajează o zonă de memorie, spațiul virtual este distinct, dar se mapează peste aceeași zonă de memorie fizică.

În figura de mai sus este prezentat un spațiu de adresă tipic pentru un proces. În sistemele de operare moderne, în spațiul virtual al fiecărui proces se mapează memoria kernelului, aceasta poate fi mapată fie la început, fie la sfârșitul spațiului de adresă. În continuare, ne vom referi numai la spațiul de adresă din user-space pentru un proces.

Cele 4 zone importante din spațiul de adresă al unui proces sunt zona de date, zona de cod, stiva și heap-ul. După cum se observă și din figură, stiva și heap-ul sunt zonele care pot crește. De fapt, aceste două zone sunt dinamice și au sens doar în contextul unui proces. De partea cealaltă, informațiile din zona de date și din zona de cod sunt descrise în executabil.

Zona de cod

Segmentul de cod (denumit și `text segment`) reprezintă instrucțiunile în limbaj mașină ale programului. Registrul de tip `instruction pointer` (IP) va referi adrese din zona de cod. Se citește instrucțiunea indicată de către IP, se decodifică și se interpretează, după care se incrementează contorul programului și se trece la următoarea instrucțiune. Zona de cod este, de obicei, o zonă read-only pentru ca procesul să nu poată modifica propriile instrucțiuni prin folosirea greșită a unui pointer. Zona de cod este partajată între toate procesele care rulează același program. Astfel, o singură copie a codului este mapată în spațiul de adresă virtual al tuturor proceselor.

Zone de date

Zonele de date conțin variabilele globale definite într-un program și variabilele de tipul read-only. În funcție de tipul de date există mai multe subtipuri de zone de date.

.data

Zona `.data` conține variabilele globale și variabilele statice inițializate la valori nenule ale unui program. De exemplu:

```
static int a = 3;
char b = 'a';
```

.bss

Zona `.bss` conține variabilele globale și variabilele statice neinițializate ale unui program. Înainte de execuția codului, acest segment este inițializat cu 0. De exemplu:

```
static int a;
char b;
```

În general aceste variabile nu vor fi prealocate în executabil, ci în momentul creării procesului. Alocarea zonei `.bss` se face peste pagini fizice zero (zeroed frames).

.rodata

Zona `.rodata` conține informație care poate fi doar citită, nu și modificată. Aici sunt stocate literalii:

```
"Hello, World!"
```

```
"Hello World!"
```

și **constantele**. **Toate variabilele globale declarate cu keyword-ul** `const` **vor fi puse în** `.rodata`**. Variabilele locale declarate ca fiind** `const` **vor fi puse pe stivă, deci într-o zonă de memorie nemarcată ca *read-only* și vor putea fi modificate prin intermediul unui pointer către acestea. Un caz aparte îl reprezintă variabilele locale constante declarate cu keyword-ul** `static` **care vor fi puse în** `.rodata`**:**

```
const int a;            /* în .rodata */
const char ptr[];       /* în .rodata */

void myfunc(void)
{
   int x;               /* pe stivă */
   const int y;         /* pe stivă */

   static const int z; /* în .rodata */

   static int p = 8;    /* în .data */
   static int q;        /* în .bss */
   ...
```

Stiva

Stiva este o regiune dinamică în cadrul unui proces, fiind gestionată automat de compilator.

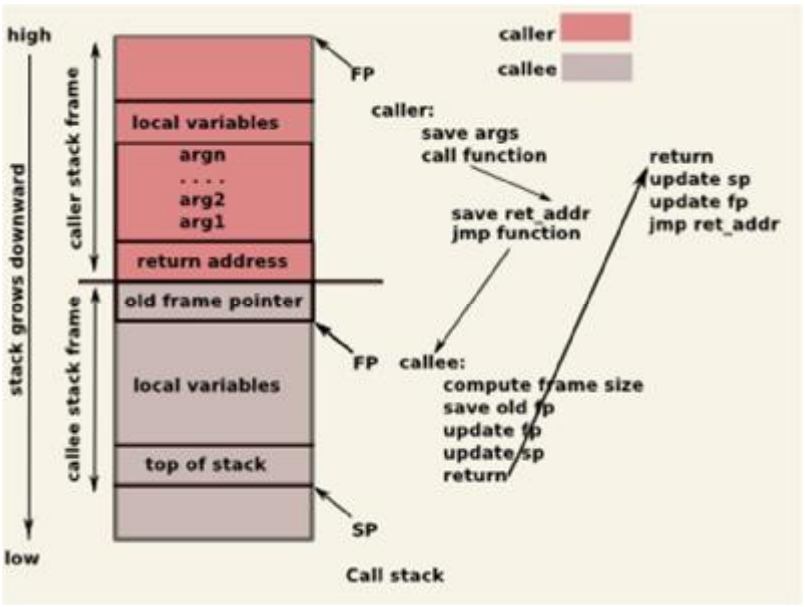
Stiva este folosită pentru a stoca "stack frame-uri". Pentru fiecare apel de funcție se va crea un nou "stack frame".

Un "stack frame" conține:

- **variabile locale**
- **argumentele funcției**
- **adresa de retur**

Pe marea majoritate a arhitecturilor moderne stiva crește în jos (de la adrese mari la adrese mici) și heap-ul crește în sus. Stiva crește la fiecare apel de funcție și scade la fiecare revenire din funcție.

În figura de mai jos este prezentată o vedere conceptuală asupra stivei in momentul apelului unei funcții.



Heap-ul

Heap-ul este zona de memorie dedicată alocării dinamice a memoriei. Heap-ul este folosit pentru alocarea de regiuni de memorie a căror dimensiune este determinată la runtime.

La fel ca și stiva, heap-ul este o regiune dinamică care își modifică dimensiunea. Spre deosebire de stivă, însă, heap-ul nu este gestionat de compilator. Este de datoria programatorului să știe câtă memorie trebuie să aloce și să rețină cât a alocat și când trebuie să dealoce. Problemele frecvente în majoritatea programelor țin de pierderea referințelor la zonele alocate (memory leaks) sau referirea de zone nealocate sau insuficient alocate (accese nevalide).

În limbaje precum Java, Lisp etc. unde nu există "pointer freedom", eliberarea spațiului alocat se face automat prin intermediul unui garbage collector. Pe aceste sisteme se previne problema pierderii referințelor, dar încă rămâne activă problema referirii zonelor nealocate.

Practic.

1. Scrieti un program C, care sa afiseze Hello World! Compilati-l. Rulati objdump -h nume_executabil si identificati zonele descrise mai sus. Gasiti zona de stack si de heap? De ce? Identificati adresele de unde incep si unde se termina zonele gasite.
2. Creati o bucla in executabilul vostru astfel incat acesta sa nu se termine. Rulati comanda cat /proc/nr_pid/maps – unde nr_pid reprezinta numarul pidului vostru. Gasiti unde e mapata stiva si heapul executabilului vostru. De asemenea observati ca mai sunt si alte lucruri mapate in procesul vostru – ce sunt acele lucruri mapate? Rulati comanda ldd nume_executabil – ce observati si ce legatura are cu maparea de mai sus?

Threaduri – fire de executie

Vom reaminti cateva functii importante:

pthread_create creaza un thread

pthread_join asteapta terminarea unui thread

mmap – se poate aloca memorie virtuala

sigaction se poate inregistra un handler al unui semnal.

Cu instructiunea de consola kill se poate trimite un semnal unui proces.

Practic.

1. Scrieti o aplicatie unde sa alocati memorie virtuala doar cu drepturi de read. Inregistrati un handler pentru semnalul SIGSEGV. Gasiti in handlerul semnalului adresa care a generat faultul si schimbati cu mprotect drepturile acelei locatii de memorie.
2. Discutie – daca trimit un semnal unui proces care are mai multe threaduri ce thread il va prelucra? Daca un thread produce un acces invalid de memorie, ce thread va trata acel semnal?

Bibliografie:

https://devarea.com/linux-handling-signals-in-a-multithreaded-application/#.Xksj6GgzZPY

Operating System Concepts 8th Edition – Abraham Silberschatz, Peter B. Gavin, Greg Gagne

https://ocw.cs.pub.ro/courses/so/laboratoare