Administración de Memoria

El ciclo de vida de un programa:

1. **Edición:** construcción del código en un determinado lenguaje.
2. **Compilación:** conversión del código a un lenguaje objetivo (lenguaje máquina, en general) . Se revisa la consistencia del código, tipos, sintaxis.
3. **Distribución:** se empaqueta el programa en un ejecutable.
4. **Enlace:** se ligan dependencias y otras bibliotecas. El ligado puede ser estático (al compilar) o dinámico (las bibliotecas son externas).
5. **Carga:** el programa se carga a memoria principal. Se le asignan recursos.
6. **Ejecución**

A diferencia de Java, en C/C++ el programador participa activamente en el ciclo de vida del programa (del 1 al 4 paso).

Sistema Operativo

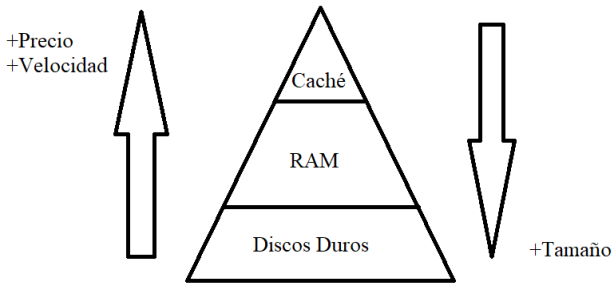
Es software colocado directamente sobre el hardware. De cara al programador el SO provee un API que abstrae los recursos y que permite construir programas de aplicación para el usuario final.

De manera general, las funciones del SO son:

- Administrar procesos (CPU)
- Administrar memoria (RAM)
- Administrar archivos y discos
- Administrar In & Outs

La memoria es un recurso valioso que debe ser administrado cuidadosamente. De lo contrario los programas “lucharán” por acaparar toda la memoria disponible.

Jerarquía de memoria



Esquema de administración de memoria:

- Esquema inexistente
- Espacios de direcciones
- Memoria Virtual

1. **Esquema Inexistente:**
    - Utilizado en los mainframes de antaño (60’s)
    - No había ninguna abstracción de la memoria
    - Los programas “ven” la memoria física
    - Por ejemplo: al ejecutar la instrucción =>mov, Ax1000
      En realidad, se direcciona la posición de memoria
    - El “modelo” es un conjunto de direcciones de 0 al máximo.
    - En un momento dado, sólo establece en memoria el SO y un solo programa de usuario
    - Algunas variaciones
    - Cuando el SO ejecuta un programa, lo copia por entero al RAM, cuando este termina, se ejecuta otro de la misma forma.
    - Eventualmente soporta multiprogramación dividiendo la memoria en bloques. Cada bloque tiene una llave asignada, por el SO. Si un programa trata de leer memoria de un bloque que no le pertenece, se detiene su ejecución, sin embargo, existía un problema. Dado que los programas manipulaban la memoria física, la instrucción 0 del programa B, causa una invasión al programa A.

- Para resolver esto, se usa static relocation, esto significa que en la etapa de carga las direcciones a lo interno del código se cambian sumándoles la dirección en la que empezó a cargar.

Espacios de direcciones

- Cada proceso tiene un conjunto fijo de direcciones de memoria que puede utilizar. Se implementa utilizando dynamic relocation, cada programa cree que empieza en la dirección 0.
- El CPU tiene 2 registro especiales: >Base. >Límite
- Cuando el programa refencia memoria, se le suma la base y se verifica que no sea mayor que el límite.
- En esta generación de SO, se introduce el concepto de swapping
- Swapping permitió correr un conjunto total de programas mayor que el total de RAM.
- Bajo este esquema, un programa corre por cierto tiempo de ser necesario, se baja al disco y se sube otro programa en su lugar.

Memoria Virtual

- “Virtual se ve pero no existe” VS “transparente”
- El tamaño de los programas crece más rápido que la cantidad de RAM disponible
- Existe una memoria virtual de mayor tamaño que el RAM. El programa opera sobre dicha memoria virtual, creyendo que en su totalidad, esté en el RAM
- El programa se divide en páginas. Cada página es un bloque de memoria contigua
- La memoria real se divide en páginas. Las páginas reales y virtuales son del mismo tamaño.
- Cada página virtual se mapea a una página real. Este mapeo se hace mediante el MMU (Memoy Managment Unit)
- La MMU tiene la tabla de páginas. Dicha tabla mantiene el mapeo enbtre páginas reales y virtuales.
- Cuando el programa referencia una página no cargada se dispara un page fault
- En un page fault se trae la página del disco y se actualiza la tabla
- Si la tabla de páginas esta llena, se aplica un algritmo de reemplazo

A nivel de proceso

- La memoria para un proceso tieen una estructura bien definida, esta estructura puede variar según el compilador
- Para el lenguaje de C/C++, la estructura puede ser

Parámetros de línea de comandos y variables de ambiente
Pila
Heap
Datos sin inicializar
Datos inicializados
Código

  - **Código:** código ejecutable
  - **Datos inicializados:** variables globales y estáticas inicializadas por el programador
  - **Datos sin inicializar:** variables globales no inicializadas
  - **Heap y Stack:** secciones especiales
  - **Parámetros de línea de comandos y variables de ambiente:** parámetros pasados del programa y variables definidas en el Shell

Pila

- Sección del memory layout
- Se comporta LIFO (last In, First Out)
- Amigable para el programador
  - Menos pulgas
  - Menos trabajo
- Se compone de Stack Frames
- Un frame contiene:
  - Storage para variables locales
  - Número de línea donde regresar
  - Storage para parámetros
- La pila hace transparente la memoria

- Cada llamada un método es un nuevo stack frame
- Cuando el método termina el frame se elimina por completo y la memoria asociada se libera
    - Las variables locales se conocen como variables antométricas
- Las variables antométricas solo son de tipo primitivo

Heap

- Sección del memory layout
- No tiene estructura
- No es transparente
- No hay API
    - **Matloc (memory allocation):** asigna n bytes en memoria y retorna un puntero
    - **Free:** libera la memoria apuntada pro el puntero
    - **Realloc**
    - **Calloc**
    - **Delete**
    - **New**
- **No imprime limites a las variables usadas**
- **En Java el heap es manejado**
    - **Transparente**
    - **Garbage collector**

Pointer

- Tipo de dato

  int *=> int *aptr=(int *) malloc(sizof(int));

  char *, double *, long *, float *, void *, struct *

  struct A {

  char *b;

  int c;

  }

  Struct A*

  Class Persona

Persona *p=new Persona ();

- Todos los punteros son del mismo tamaño
    - 32 bits ->4 bytes
    - 64 bits -> 8 bytes
- Operaciones:
    - Referenciador &
    - Desreferenciador *
- Un puntero sin inicializador tiene un valor conocido como BAD VALUE
- Siempre inicializar punteros en cero: char *a=null;

STACK	HEAP
+ Temporal	+Lifetime: control total
+ Copias Locales	+Control sobre el tamaño de las variables
-Tiempo de vida corto	-Más trabajo
-Comunicación restringida	-Más pulgas

- Shallow copy
- Deep copy: copia el dato almacenado en el ptr
- Dengling pointer

Paso parámetros por valor

- El caller no ver los cambios hechos por el llamado a los parámetros pasados
- El llamado recibe nuevas copias en su Stack frame

```
Void setvar(int x) {
x=8;
}
Void test () {
Int x=10;
Setvar(x)
}
```

Paso parámetro por referencia

- El llamado recibe una referencia o un pointer como parámetro
- El caller veía cambias hechos por el llamado a los parámetros

  Void setvar (int *p) {

  *p=8;}

  Void test (){}