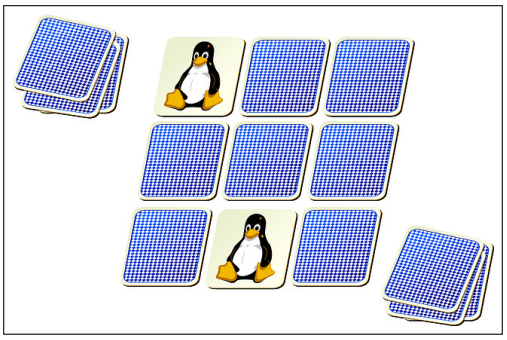
Memoria virtual en Linux



Esteban Ruiz
eruiz0@fceia.unr.edu.ar

Sistemas Operativos II - 2016
Departamento de Ciencias de la Computación
FCEIA-UNR

Índice

1. Organización de la memoria física

1.1. Zonas de memoria

El sistema operativo debe decidir qué porción de la memoria física asigna a memoria interna del núcleo (en donde habrá buffers, datos y el código mismo del núcleo) y qué porción será asignada dinámicamente (memoria dinámica *del sistema*) para procesos y cachés. Además hay porciones de memoria reservadas por el hardware o para uso específico del hardware y que el núcleo no puede utilizar libremente (p.ej.: dispositivos mapeados en memoria, vectores de interrupciones, etc...). La configuración exacta depende de la arquitectura de hardware, pero en los x86 GNU/Linux utiliza la siguiente disposición (fig.: 1):

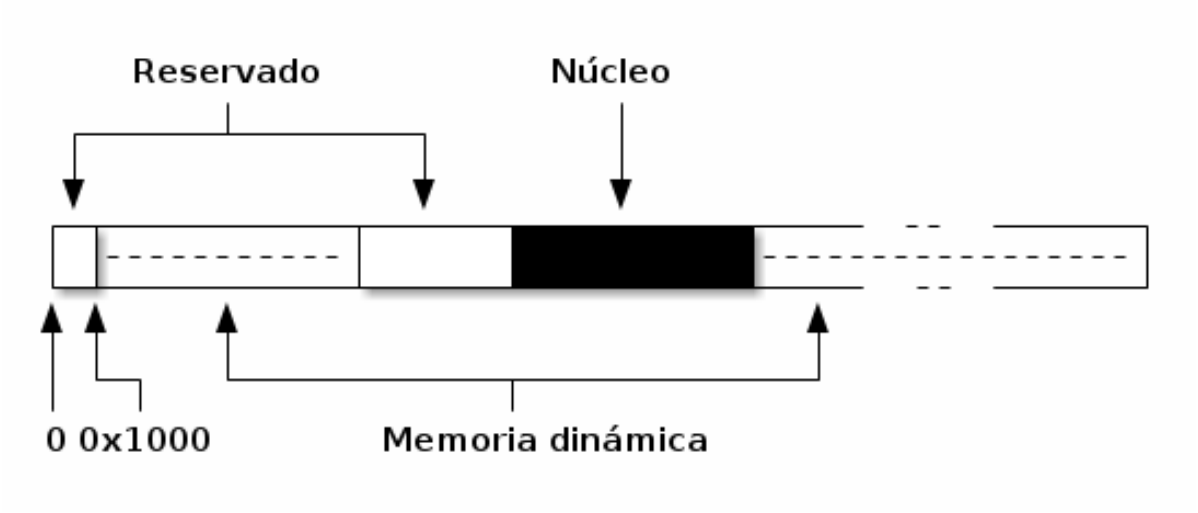


Figura 1: Configuración de memoria en un x86

- Bloque [0 .. 0x0fff] (primeros 4 kb -1 página-): reservado para el hardware (configuración de hardware detectada durante el POST[1]).
- Bloque [0x1000 .. 0x9ffff]: disponible para memoria dinámica.
- Bloque [0xa0000 .. 0xfffff]: reservado para el hardware (dispositivos mapeados en memoria y rutinas de bios), este el concoido "agujero" desde los 640 Kb hasta 1Mb..
- Bloque [0x100000 .. 0x100000+sizeof(kernel)]: núcleo.
- Bloque [0x100000+sizeof(kernel) .. DIRMAX]: disponible para memoria dinámica.

Se decide cargar el núcleo después de la memoria reservada para el hardware para poder cargarlo en un espacio de memoria contiguo (es demasiado grande para cargarlo antes de ls 640 Kb). De esta forma, el código del núcleo se mantiene sencillo (sin huecos) y además es más fácil cargarlo en memoria.

[1]Power On Self Test - Pruebas realizadas por la BIOS durante la inicialización de la máquina

1.2. Algunas optimizaciones y características a tener en cuenta

GNU/Linux implementa memoria compartida entre procesos (e hilos). Así, una página que está en memoria puede estar siendo referenciada desde una, dos o más tablas de paginación. Además se implementa Copy On Write para evitar duplicar todas las páginas de un proceso en `fork()`. Por último, también se soporta carga por demanda: las páginas se van leyendo o creando a partir de la informaciòn del programa binario a medida que es necesario.

Otra característica a tener en cuenta es el mapeo de archivos en memoria (memory mapped files): un proceso puede solicitar que una porción de su memoria virtual se refiera a un archivo. De esta forma, escribir/leer en un archivo (desde el punto de vista del proceso) consiste en escribir/leer directamente en una dirección de memoria. Eventualmente, para liberar esa página no es conveniente escribirla en swap, sino que conviene escribirla directamente en el espacio reservado para el archivo.

Además, para optimizar el rendimiento de los dispositivos de almacenamiento secundario, se implementa la *lectura por adelantado* (readahead): cuando se lee un bloque es muy alta la probabilidad de que sea necesario leer en poco tiempo más el bloque siguiente. Por las velocidades de posicionamiento de los dispositivos tal vez no convenga esperar a que el proceso haga la solicitud: unos pocos milisegundos seguramente harán que el disco pase sobre el siguiente bloque a leer y haya que esperar una nueva rotación completa para volver a leer el bloque (y la cosa empeora mucho más si también se desplazó el brazo lector). Por esto se utilizan cachés de disco que permiten leer secuencialmente varios bloques contiguos cuando en principio sólo se necesita el primero.

1.3. NUMA

Además GNU/Linux soporta *Acceso No Uniforme a Memoria* (NUMA - Not Uniform Memory Access). NUMA se utiliza típicamente en equipos multiprocesadores, en donde determinadas porciones de memoria (nodos) están asociadas a un procesador brindando una velocidad de acceso mayor a ese procesador (con respecto al resto de la memoria del cluster o del equipo). Así, cuando se utiliza NUMA la memoria es particionada en nodos y a su vez cada nodo puede subdividirse en zonas. Muchas de las estructuras de datos relacionadas con la gestión de memoria deben entonces replicarse a cada nodo (y si es necesario en cada zona).

1.4. Páginas compartidas, paginación y reverse mapping

Para la paginación en i386 se decide utilizar páginas de 4Kb, utilizando tablas de paginación de 4 niveles.

El sistema permite que un marco de página sea utilizado por varios procesos (p. ej.: para archivos mapeados en la memoria de dos procesos o para páginas marcadas para Copy On Write).

1.4.1. Reverse mapping

Además de las tablas de paginación de cada proceso, para el funcionamiento del algoritmo de reemplazo de páginas se debe almacenar cierta información sobre cada marco "candidato", por ejemplo: si la página está bloqueada en memoria (temporalmente o por pedido del proceso), qué entradas en tablas de traducciones están apuntando a esta página

(o sea: qué procesos están utilizando esta página), si fue utilizada hace poco, etc... Los punteros a las entradas en las tablas de traducciones son necesarios para poder actualizar las tablas correspondientes cuando se elige la página como víctima.

El proceso de obtener las entradas que apuntan a determinado marco se llama *mapeo inverso* (reverse mapping). Para almacenar los punteros a las entradas de traducción (translation entries) que apuntan a cada marco se podría utilizar una lista. Sin embargo, hacerlo así incrementaría significativamente la sobrecarga del núcleo y en su lugar se utilizan punteros a regiones de memoria que permiten obtener (a través de otras tablas relacionadas) la misma información.

1.5. Otras reservas

Por otro lado, hay otras porciones de memoria que requieren tratamiento particular. Veamos: si se necesita un nuevo marco de memoria como parte del procesamiento de una interrupción no se puede esperar a que el gestor de memoria envíe una página al disco o actualice muchas estructuras de datos (pues los tiempos de procesamiento de muchas interrupciones deben ser cortos para poder sincronizarse con la velocidad de los dispositivos). Otro caso similar es durante el procesamiento de código en una sección crítica del núcleo.

Para estos casos, cuando se requiere memoria, el núcleo no hace un pedido normal de memoria, sino que existe un mecanismo llamado *pedido de ubicación atómica* (atomic allocation request) que debe resolver instantáneamente la demanda (o retornar un error). Para esto se necesita una reserva (pool) de páginas destinadas sólo a esos casos.

2. Ubicación de memoria y gestión de memoria libre

Se ha visto que una solución al problema de la fragmentación externa es utilizar el mecanismo de paginación para que no queden huecos en memoria que luego obliguen a desfragmentar la memoria. Si bien GNU/Linux utiliza paginación en los espacios de direcciones de los procesos, hacerlo también para la memoria gestionada por el núcleo y para rastrear la memoria libre/usada resulta casi imposible:

- Los controladores de DMA requieren de memoria (física) contigua para poder leer/escribir masivamente datos desde/hacia los dispositivos de almacenamiento. Si se utilizara paginación, tal vez se necesite desfragmentar la memoria para construir huecos lo suficientemente grandes (la gestión se complica).

- Cada vez que se hace una modificación en la tabla de paginación activa, la CPU vacía automáticamente la TLB. Hacer eso con la memoria del sistema o del núcleo penalizaría mucho el rendimiento.

- Utilizar un tamaño de página más grande (4Mb) para el núcleo y áreas reservadas aumenta mucho la eficiencia de la TLB, pero si se usa paginación se deberían utilizar páginas de 4Kb.

Por esta razón, se emplea un método muy conocido para hacer el seguimiento de la memoria libre: el *buddy system* (sistema de colegas). Básicamente la información sobre los bloques de memoria libre se guarda en una estructura de datos que puede verse como un arreglo de listas. Los bloques de memoria libre se agrupan en potencias de dos. Esto

es: la lista correspondiente al elemento 0 del arreglo contiene la información (básicamente ubicación) sobre huecos del tamaño de una página (es decir, es una lista enlazada de huecos del tamaño de una página); la lista en el elemento 1 describe todos los huecos de 2 páginas, la del elemento 2 describe los huecos de 4 páginas y así sucesivamente. GNU/Linux utiliza 11 potencias de dos, así los huecos más grandes tienen 2^{11} páginas.

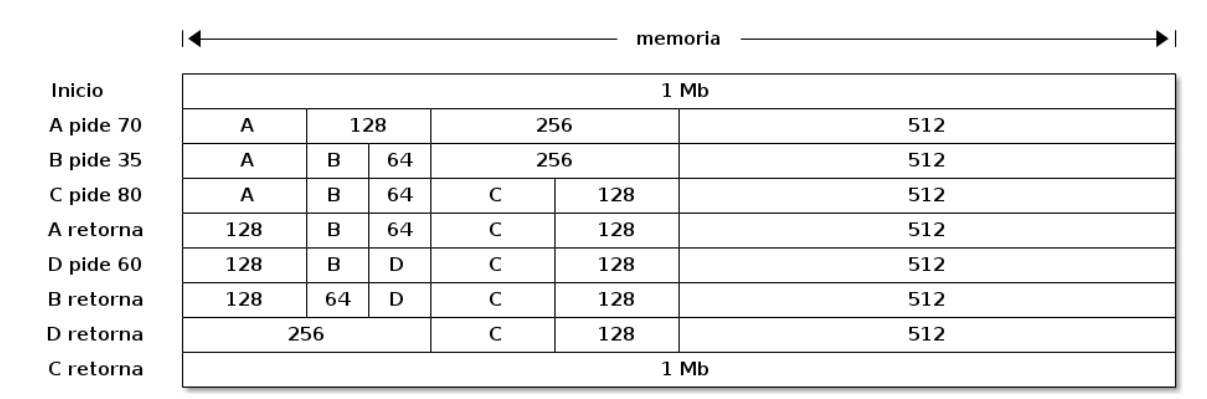


Figura 2: Comportamiento del buddy system ante distintos pedidos

La figura 2 muestra cómo evoluciona la reserva de memoria ante diversos pedidos: cuando se necesita asignar un bloque contiguo de memoria se busca el primer hueco lo suficientemente grande como para satisfacer el pedido (mejor ajuste). Si se necesita partir un hueco grande, se asigna lo necesario (removiendo ese hueco del arreglo) y el espacio restante se añade a las listas correspondientes.

Lo opuesto ocurre cuando se libera una porción de memoria: en este caso se añaden los huecos a las listas correspondientes. Pero esta vez hay que tener en cuenta una cosa: si en una lista quedan dos bloques "colegas" (vecinos en memoria y alineados al siguiente tamaño más grande), esos dos bloques pueden fundirse para formar un nuevo bloque más grande. De esta manera se soluciona el problema de la fragmentación externa.

2.1. Regiones más pequeñas

Dado que el buddy system permite administrar huecos de por lo menos una página, no puede utilizarse para reservar memoria en porciones más chicas. Sin embargo, esto es una tarea también bastante habitual. Por esto existe otra porción del manejo de memoria que se encarga de gestionar pedidos más chicos de memoria: el *slab* allocator. Esta gestión permite reducir la fragmentación interna y es también una porción muy complicada de software.

2.2. Asignación de memoria

La memoria RAM se asigna indistintamente a procesos y a caché. Todos puede crecer indiscriminadamente (si hay pocos procesos posiblemente haya mucha memoria utilizada para caché y si hay muchos procesos, buena parte de la memoria se utilizará para guardar información de los procesos y poca para los cachés).

3. Reclamo de páginas

¿Qué ocurre cuando la memoria se llena? En realidad un poco antes, el *algoritmo de reclamo de marcos del núcleo* (page frame reclaiming algorithm - PFRA -) se encarga de rellenar la lista de bloques libres "robando" marcos de procesos (en modo usuario) y de los cachés del núcleo. Esto se debe hacer antes de que la memoria se agote completamente pues para escribir los datos en disco (si hace falta) se necesitan también algunas páginas de memoria (p. ej.: para cabecera de los buffers a utilizar en E/S). O sea: el algoritmo de reclamo de marcos conserva un pool de marcos libres mínimo.

3.1. Tipos de marcos

El algoritmo maneja los marcos de diferente manera según su contenido:

Tipo	Descripción	Acción
No reclamable	Libres ó reservadas Memoria dinámica del núcleo Stack del núcleo de los procesos Bloqueadas temporalmente Bloqueadas en memoria	Ignorar
Swapeable	Anónimas en espacios de direcciones en modo usuario Páginas del *tmpfs* (p. ej.: páginas de memoria compartida para IPC)	Guardar el contenido en swap
Sincronizable	Mapeadas en espacios de direcciones en modo usuario Caché y contiene datos de archivos en disco Buffer de dispositivos de bloques Algunos cachés de disco (p. ej.: el caché de inodos)	Sincronizar con la imágen en disco (si es necesario)
Descartable	Páginas no usadas del caché de memoria Páginas no usadas del caché de direntries	No hace falta hacer nada

Veamos algunas distinciones:

- Hay páginas que pueden estar *bloqueadas en memoria*. Este bloqueo puede ser temporal (p. ej. si se está usando la página para realizar entradas/salida o en una sección crítica del núcleo) o permanente (p. ej. por pedido explícito del proceso).
- La páginas *anónimas* son aquellas que forman parte del espacio de memoria virtual de un proceso (por ejemplo, el segmento de código, datos, etc...) y no contienen datos provenientes de archivos (como ser archivos mapeados en memoria, librerías compartidas, etc...).
- Algunas páginas pueden contener archivos *mapeados en memoria*: estas páginas aparecen dentro del espacio de memoria virtual de un proceso pero sus datos provienen de un archivo.
- El sistema tiene cachés para distintos tipos de datos: direntries, inodos, datos de archivos, etc... Algunos tipos de datos se usan más frecuentemente que otros. Los

datos que están en caché podrían estar modificados en memoria y en ese caso, para liberar la memoria ocupada se deben escribir los datos en el lugar correspondiente en el disco (y puede involucrar actualizar otros datos).

3.2. Principios del PFRA

El algoritmo de reemplazo de marcos es una pieza de software compleja y no responde (al menos directamente) a ningún marco teórico: es más bien una serie de criterios que se han ido afinando en el tiempo, según parámetros y algoritmos más específicos para casos particulares.

No obstante, hay una serie de reglas que se aplican en general:

- Liberar primero las páginas que provoquen menos daño (p. ej.: la páginas de cachés).
- Todas las páginas de un proceso (salvo las bloqueadas en memoria) son reclamables.
- Antes de liberar una página compartida, se deben ajustar las entradas en las tablas de paginación de los procesos involucrados (veremos luego que esto se hace con *mapeo inverso*).
- Reclamar sólo páginas no utilizadas recientemente (usa una versión simplificada de LRU), utilizando el bit Accessed.
- En lo posible elegir las páginas limpias (no-dirty). Esto ahorra la escritura en el disco.

3.3. Cuando la memoria escasea

El reclamo de páginas se hace en forma periódica a través de un proceso llamado *kswapd*. También hay un mecanismo similar para recuperar memoria de los bufferes utilizados por el slab allocator.

Sin embargo, si falla algún pedido de memoria (low on memory) se disparan mecanismos más agresivos para recuperar la memoria que consisten principalmente en achicar el tamaño de los cachés.

Incluso, si la situación se torna muy crítica, el núcleo puede decidir eliminar un proceso (sin siquiera enviarlo a swap). De tomar la decisión se encarga una pieza especial del rompecabezas llamado el Out of Memory (OOM) Killer. Esta decisión no es nada sencilla: se intenta elegir un proceso bastante grande (para poder liberar bastante memoria), que no haya hecho demasiado progreso (para no desperdiciar trabajo) y que no esté involucrado en secciones críticas, entrada/salida o tareas del núcleo que puedan dejar el sistema inconsistente.

3.4. Algunas consideraciones sobre swap

Para realizar el intercambio de página de swap con el disco se utilizá un caché de swap: de esta forma, si se necesita una página que fue escrita en disco pero que todavía está en el caché no es necesario traerla nuevamente del disco. Sin embargo, también hay que tener en cuenta un par de problemas de concurrencia:

- *Multiple swap-in*: el núcleo debe tomar las medidas para que una página compartida no sea cargada dos veces en memoria (convirtiéndose en una página no compartida). Esto podría darse si el proceso A intenta acceder a una página compartida con el proceso B y que está en swap. En ese caso, lanzará el pedido para obtener tal página. Mientras se efectúa la lectura, puede ocurrir que el proceso B también intente acceder a esa página y el sistema operativo al notar que no está en memoria lance un nuevo pedido. Así, cuando el disco termine las dos lecturas cada proceso tendrá una copia distinta de la misma página que por lo tanto yo no estará compartida.
- *swap-in y swap-out concurrentes*: otra situación nefasta puede darse cuando el sistema decide enviar una página a swap. Si antes de que se efectivice la escritura el proceso intenta utilizar esa página y detecta que no está en memoria podría traer del disco una versión anterior de la página.

A. Apéndice: Demostración con programa que usa mucha memoria

A continuación se muestra una demostración de un programa grande que usa mucha memoria virtual:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int m[1000*1000*1000];

int main(){
    long i;
    printf("Arranqué, durmiendo, pid: %d\n", getpid());
    sleep(20);
    printf("llenando matriz\n");
    for (i=0;i<1000*1000*1000;i++){
      m[i]=i;
    }
    printf("durmiendo\n");
    sleep(20);
    printf("Pidiendo memoria\n");
    char *a = (char *) malloc(1000*1000*1000);
    printf("malloc retorno: %p, durmiendo\n", a);
    sleep(20);
    printf("Usando memoria\n");
    for (i=0;i<1000*1000*1000;i++){
      a[i]=i;
    }
    printf("Fin, durmiendo\n");
    sleep(20);
    return 0;
}
```

La idea de esta demostración es mostrar cuándo realmente se utiliza más memoria y cómo se comportan los cachés y *swap*. Para ver estas características, podemos usar dos comandos: `cat /proc/meminfo` y `top` (presionando la "M" mayúscula aparecen arriba los procesos con más memoria virtual). Con estas dos cosas en ejecución se puede correr el programa anterior (por ejemplo, llamarlo `big.c`) y ver cómo cambian los principales valores a medida que el programa avanza. Luego se pueden correr simultáneamente varias instancias del mismo programa (con `./big &`) y ver cómo se comporta.

Lo que se puede observar es:

- La memoria sólo se consume cuando se usa (i.e. por *demand loading* y algo parecido pasa con el `malloc`, aunque crece el tamaño virtual).
- En labdcc no hay *swap*, así que ahí se puede ver que se achican los cachés y si se ejecutan demasiados procesos juntos (creo que 4 o 5) el procesador decide matar a algunos.
- En las máquinas del laboratorio sí hay *swap*, así que allí si se puede ver cómo se usa el *swap*.

Referencias

[1] Daniel Bovet & Marco Cesati, *Understanding the linux kernel*, 3rd ed., O'Reilly

[2] Andrew Tanenbaum, *Modern operating Systems*, 3rd ed., Prentice Hall