main
del
manejador de memoria de MINIX 2.0

Manejo de Memoria en MINIX

- Panorama de la memoria en MINIX
- Asignación de la memoria
- Manejo de mensajes
- Estructuras de datos y algoritmos que administran la memoria
- Programa principal

Panorama de la memoria en MINIX

La forma en que se ha implementado el manejo de memoria en MINIX está condicionada principalmente por tres de sus objetivos de diseño:

- **Pensado para PC's**
- **No permite Tiempo Compartido**
- **Deseo de hacer un sistema fácil de implementar en otros pequeños ordenadores**

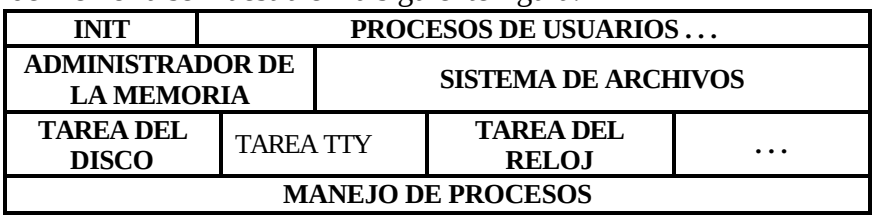
- En primer lugar, MINIX es un Sistema Operativo para computadores personales, en los que el número medio de procesos en ejecución es pequeño, por lo que normalmente no hay problemas de memoria (hay suficiente memoria).

- En segundo lugar, el desarrollo de MINIX se hizo concretamente en un PC IBM. Este micro está basado en la familia de procesadores de Intel 8088 que no soporta memoria virtual ni detección por hardware del desbordamiento de la pila.

- Por último, se pretendía que MINIX fuera fácilmente transportable a otros pequeños PC por lo que debería ser lo más simple posible. En consecuencia, no se utiliza ni paginación ni intercambio.

El administrador de la memoria conserva una lista de huecos ordenada en secuencia de direcciones de memoria. Cuando se necesita memoria debido a una llamada al sistema FORK o EXEC, se busca en la lista de huecos utilizando el Algoritmo FIRST FIT (se elige el primer hueco libre lo suficientemente grande). Una vez que un proceso se ha colocado en la memoria, éste permanece exactamente en el mismo sitio hasta que termina. Nunca se intercambia ni se mueve a otro sitio de la memoria. Tampoco crece ni se comprime el área asignada.

Otro aspecto inusual de MINIX es la forma en la que se implementa la administración de la memoria. En lugar de ser parte del Kernel. es manejado por el proceso administrador de la memoria, el cual se ejecuta en el espacio de usuario y se comunica con el Kernel por medio del mecanismo estándar de mensajes. La posición del manejador de memoria se muestra en la siguiente figura:



El retiro del administrador de la memoria del Kernel es un ejemplo de la separación de política y mecanismo. Las decisiones acerca de qué proceso se colocará en qué sitio de la memoria (política) son tomadas por el administrador de la memoria. La colocación real de mapas de memoria para los procesos (mecanismo) es realizada por la tarea del sistema contenida en el Kernel. Esta división facilita en forma relativa el cambio de la política de la administración de la memoria (algoritmos, etc...) sin tener que modificar los estratos inferiores del sistema operativo.

De esta forma, cuando el MINIX arranca, forma el mapa de memoria de las ocho tareas de E / S, así como la de otros tres procesos que se ejecutan durante toda la vida del sistema operativo. Estos procesos son:

- **MM Manejador de Memoria.**
- **FS. Sistema de Ficheros.**
- **Init. Proceso que se encarga de crear un proceso para cada terminal del sistema.**

El mapa de memoria, una vez cargado el sistema operativo queda como sigue:

Funcionamiento del manejador de memoria.

Al igual que todos los otros componentes del MINIX, el administrador de memoria es impulsado por mensajes. Después de que el sistema se ha inicializado, el administrador de memoria entra en su ciclo central, el cual consta de:

- **espera por un mensaje**
- **ejecución de la solicitud contenida en el mensaje**
- **emisión de una contestación**

Tipos de mensajes

- Relacionados con la asignación y desasignación de la memoria
  - FORK, EXIT, WAIT, WAITPID, BRK, EXEC.
- Pueden afectar al contenido de la memoria (una señal que elimina un proceso también provoca que su memoria sea desasignada)
  - KILL, ALARM, PAUSE, SIGACTION, SIGSUSPEND, SIGPENDING, SIGMASK, SIGRETURN
- Con efectos en todo el Sistema Operativo, (pero su primera tarea es enviar señales para terminar todos los procesos de forma controlada)
  - REBOOT
- No afectan al manejo de la memoria, (pero tienen que ir en el sistema de archivos o en el administrador de memoria, ya que cada llamada al sistema es manejada por uno u otro; y el sistema de archivos ya era bastante grande)
  - GETUID, GETGID, GETPID, SETUID, SETGID, SETSID, GETPGRP, PTRACE.
- Utilizado por el Kernel para informar al administrador de memoria de una señal que se origina en el Kernel (no es una llamada al sistema)
  - KSIG

Estructuras de Datos y Algoritmos del Administrador de Memoria

La principal estructura algorítmica tanto de los procesos MM como de FS es la siguiente:

```
Inicialización
while (TRUE)
{
        Recibir un mensaje
        Ejecutar el procedimiento adecuado
        Devolver una respuesta si es necesario
}
```

La inicialización se encarga de poner toda la memoria como desocupada y de marcar a MM, FS e INIT como usados.

Una vez que se ha recibido la correspondiente llamada, se decide el proceso que debe ejecutarse por medio de un vector de procedimientos (callvec[NCALLS]) previamente inicializado. Este vector se declara en el fichero “src/mm/table.c”. El ciclo central del administrador de la memoria extrae el tipo de mensaje y lo almacena en mm_call que se utiliza como índice en el vector (callvec) para determinar el apuntador al procedimiento que maneja el mensaje recién llegado. Por ejemplo, si tenemos que callvec(mm_call1) = do_xxx quiere decir que cuando se reciba la llamada al sistema mm_call1 se ejecutará el procedimiento do_xxx.

El administrador de memoria tiene dos estructuras de datos principales:

- **la Tabla de Procesos (mproc)**
- **la Tabla de Huecos (HOLE)**

Como es sabido, algunos campos de la tabla de procesos se necesitan para el manejo de procesos, otros para la administración de la memoria y otros más para el sistema de archivos. En MINIX cada una de estas tres partes del sistema operativo tiene su tabla de procesos, la cual contiene sólo aquellos campos que necesita. Las entradas corresponden exactamente, para mantener las cosas simples. Así la entrada k-ésima de MPROC (tabla de procesos del manejador de memoria) corresponde al mismo proceso de la entrada k de PROC (tabla de procesos del Kernel) y de FPROC (tabla de procesos del FS). Cuando se crea o destruye un proceso se actualizan las tres tablas.

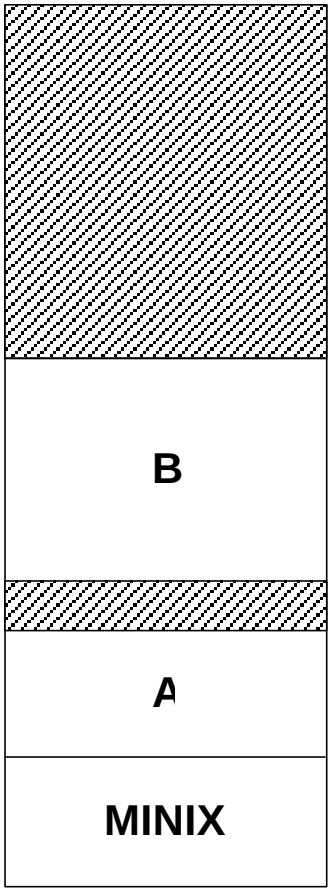
La Tabla de Procesos (mproc)

Su definición se encuentra en /usr/src/mm/mproc.h. Contiene todos los campos relacionados con la asignación de memoria a un proceso, así como algunos elementos adicionales. El campo más importante es el array *mp_seg* [NR_SEGS]; Está declarado como *mem_map* (estructura declarada en "h/type.h").

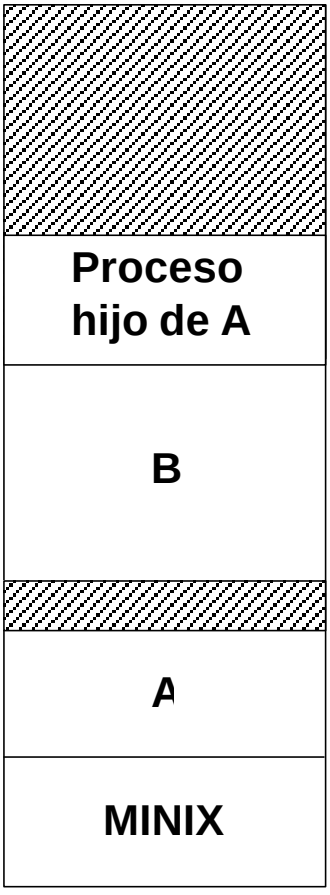
```
struct mem_map
{
        vir_clicks mem_vir;   /* Dirección virtual */
        phys_clicks mem_phys;        /* Dirección física */
        vir_clicks mem_len;   /* Longitud */
}
```

NR_SEGS es el número de segmentos (3).

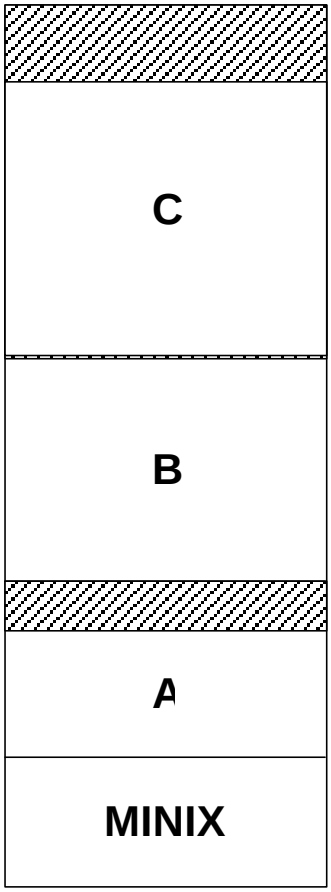
El vector *mp_seg* tiene tres entradas, cada una correspondiente al segmento de datos, al de texto y al de pila. A su vez cada entrada es una estructura *mem_map* que contiene la dirección virtual, la dirección física y la longitud del segmento, todas medidas en "clicks" de 256 bytes. Esta estructura se utiliza para delinear direcciones virtuales en direcciones de memoria física.



Dos procesos, A y B, en memoria.



Situación después de que A ejecute un FORK.



Situación después de que el proceso hijo de A ejecute el fichero C.

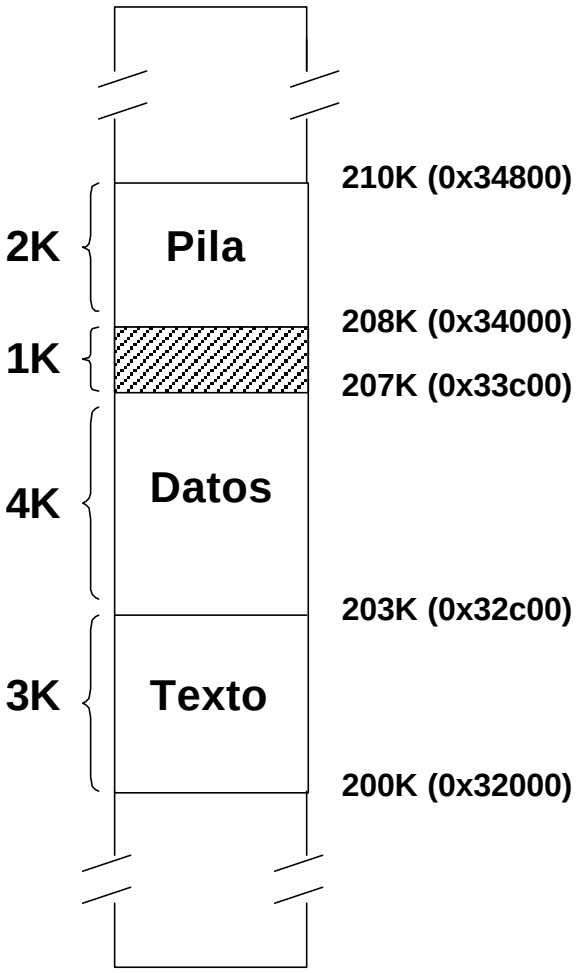
Procesos con espacio para Instrucciones y Datos combinado

Los procesos MINIX sencillos usan espacio de instrucciones y datos (I-D) combinado, en el cual todas las partes del proceso (texto, datos y pila) comparten un bloque de memoria que se asigna y libera como un bloque. La asignación de memoria a un proceso se puede realizar en dos momentos diferentes: cuando un proceso se ramifica o cuando ejecuta un programa.

En el primer caso, Un proceso ejecuta la primitiva FORK para crear un proceso hijo, a este nuevo proceso se le asigna la misma cantidad de memoria que tiene el padre.

En el segundo caso, un proceso ya existente (tiene asignada memoria) ejecuta un programa mediante la primitiva EXEC; esto hace que se libere la memoria que el proceso tenía asignada y a continuación se le asigne la cantidad de memoria indicada en la cabecera del fichero que va a ejecutar. Es posible que el nuevo fichero sea ubicado en un lugar de memoria diferente

La memoria se libera siempre que el proceso termina, ya sea porque salga del sistema o porque sea eliminado por una señal (EXIT y KILL).



Un proceso en memoria

	Virtual	Física	Longitud
Pila	0x20	0x340	0x8
Datos	0	0x320	0x1c
Texto	0	0x320	0

Con espacio de I-D no separado.
En este caso el campo longitud de texto es siempre cero, ya que ésta se representa junto con los datos.

Procesos con espacio para Instrucciones y Datos separado

Los procesos pueden compilarse también para usar espacio de instrucciones y datos separado. Estos procesos pueden utilizar la memoria más eficientemente, pero la ventaja de esta característica complica las cosas.

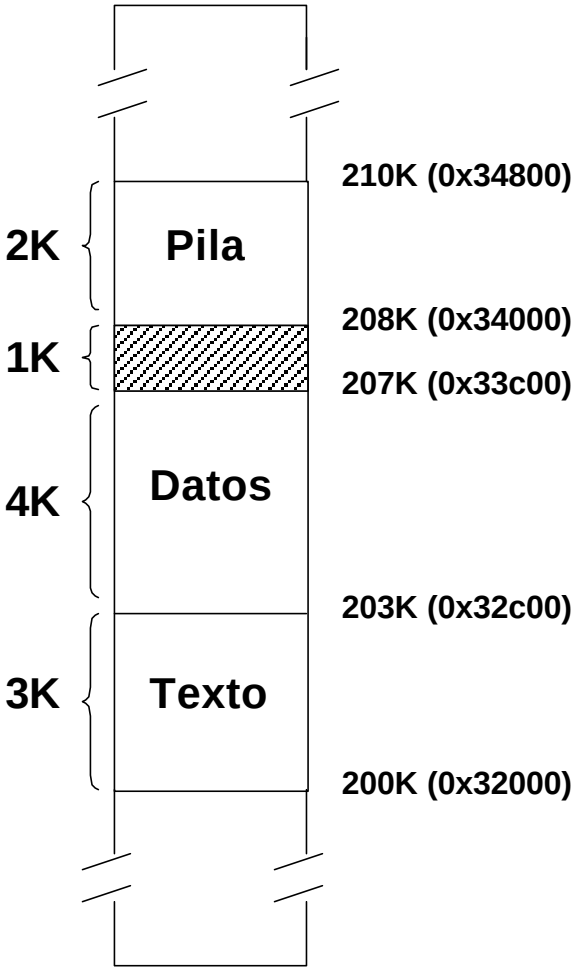
Los programas con espacio de I-D separados tienen la ventaja de un modo de manejo de memoria mejorado llamado Texto compartido. Cuando un proceso ejecuta un FORK, sólo se asigna la cantidad de memoria necesaria para la copia de los datos y pila del nuevo proceso. Ambos, padre e hijo comparten el código ejecutable que ya estaba en uso por el padre. Cuando un proceso hace un EXEC, se hace una búsqueda en la Tabla de Procesos para ver si otro proceso ya está usando el código ejecutable necesario

Si se encuentra alguno, se asigna nueva memoria, sólo para los datos y la pila,, y el texto ya en memoria se comparte. El Texto compartido complica la finalización de un proceso. Cuando un proceso termina siempre libera la memoria ocupada por sus datos y pila. Pero sólo libera la memoria ocupada por su segmento de texto después de que una búsqueda en la tabla de procesos revele que ningún otro proceso actual está compartiendo esa memoria. Por consiguiente se puede asignar más memoria a un proceso cuando empieza de la que éste libera cuando termina, si cargó su propio texto cuando comenzó pero dicho texto está siendo compartido por uno o más procesos cuando el primer proceso termina.

La cabecera en el fichero de disco contiene información sobre los tamaños de las diferentes partes de la imagen, así como el tamaño total. En la cabecera de un programa con espacio de I-D común, un campo especifica el tamaño total de las partes de texto y datos; estas partes se copian directamente a la imagen en memoria. La parte de datos en la imagen se amplía en la cantidad especificada en el campo *bss* de la cabecera. Esta área se deja libre para contener todos los nulos y se usa para los datos estáticos no inicializados. La cantidad total de memoria a ser asignada se especifica en el campo *total* de la cabecera. Un archivo de programa en el disco puede también contener una tabla de símbolos, que se usa en *debugging* y no se copia en memoria.

Para un programa que utiliza espacio I-D separados el campo *total* en la cabecera se aplica sólo al espacio combinado para datos y pila. El límite del segmento de datos puede ser movido sólo por la llamada al sistema BRK. Todo lo que BRK hace es verificar si el nuevo segmento de datos se solapa con el actual puntero de pila, y si no, observar el cambio en algunas tablas internas. Si el nuevo segmento de datos se solapa con la pila la llamada falla.

Esta estrategia se ha elegido para hacer posible la ejecución de MINIX en IBM PC, el cual no verifica por hardware si hay desbordamiento de la pila por lo que es peligroso poner la pila adyacente a cualquier elemento, a excepción de una gran cantidad de memoria no utilizada, ya que la pila puede crecer rápidamente y sin advertencia.



Un proceso en memoria

	Virtual	Física	Longitud
Pila	0x14	0x340	0x8
Datos	0	0x32c	0x10
Texto	0	0x320	0xc

Con espacio de I-D separado.
Donde la longitud del segmento de texto debe ser distinta de cero.

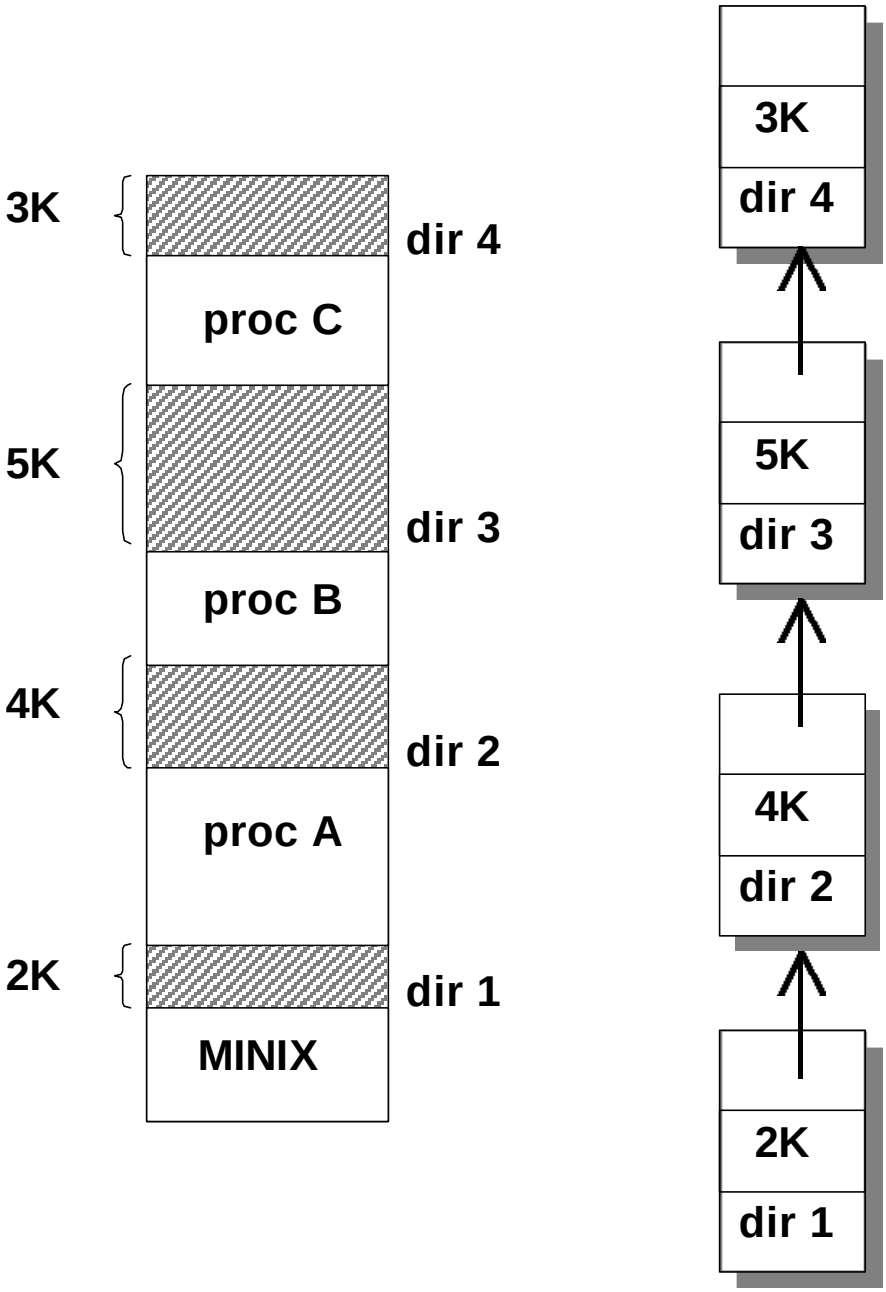
La tabla de Huecos (HOLE)

Está definida en alloc.c. Pone en una lista cada hueco en memoria en orden creciente de direcciones de memoria. Los espacios entre los segmentos de datos y pila no se consideran huecos (ya se han asignado a los procesos). Cada entrada (hueco) en la lista de huecos tiene tres campos:

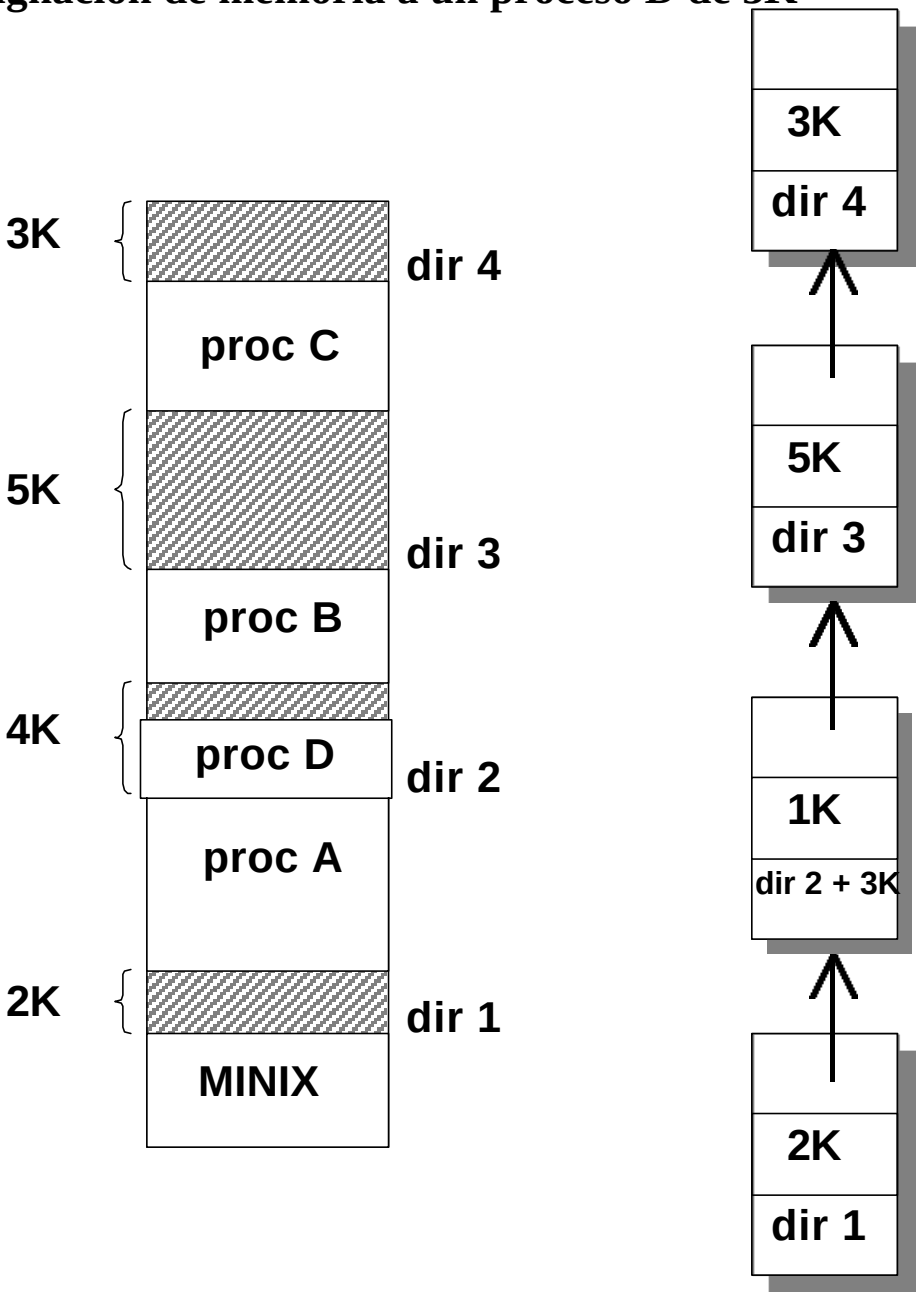
- la dirección de comienzo del hueco, en clicks
- su longitud, en clicks
- un puntero al siguiente hueco de la lista.

En el arranque del sistema sólo hay un hueco (su longitud es toda la memoria)

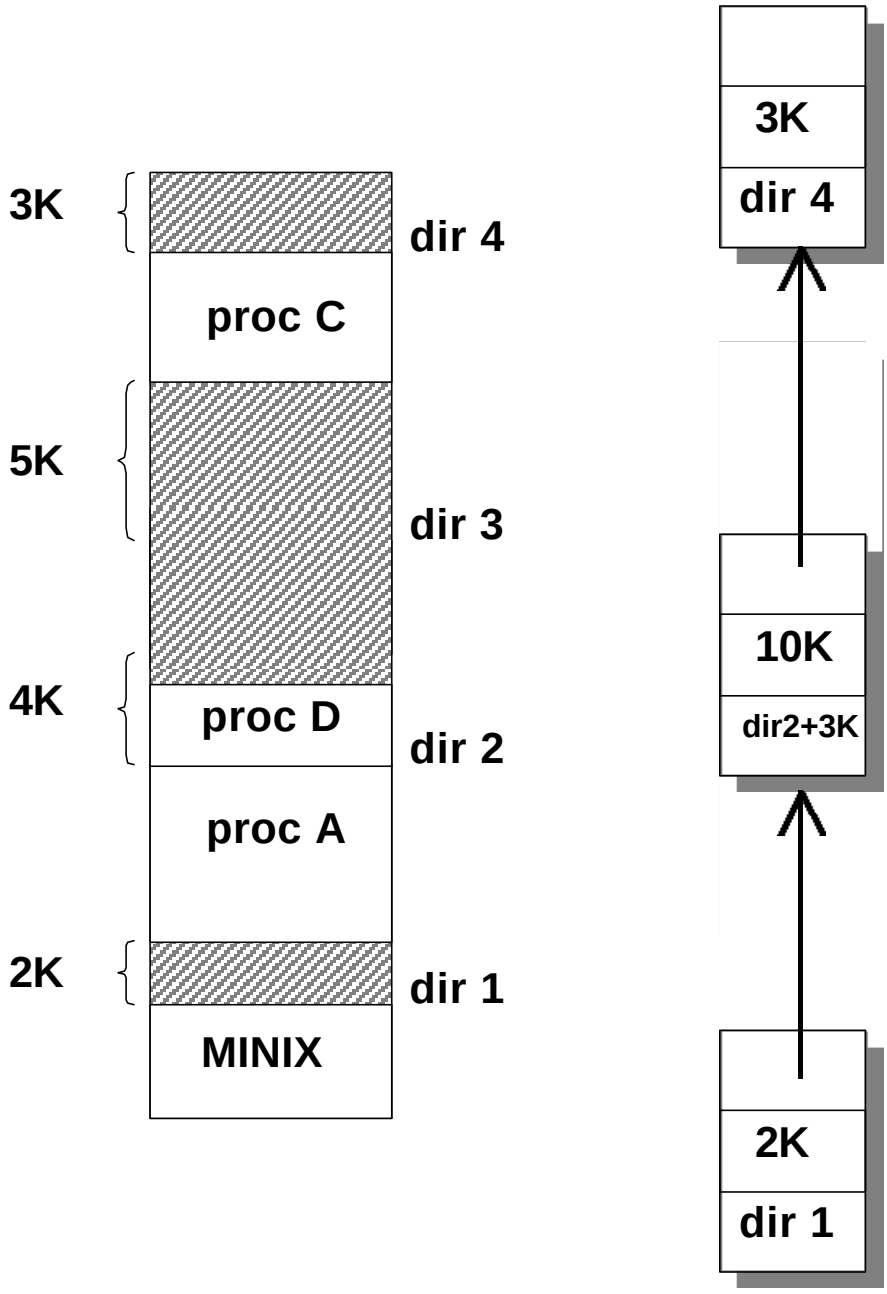
Mapa de Memoria y Lista de Huecos Libres



Asignación de memoria a un proceso D de 3K



Liberación de la memoria ocupada por el proceso B



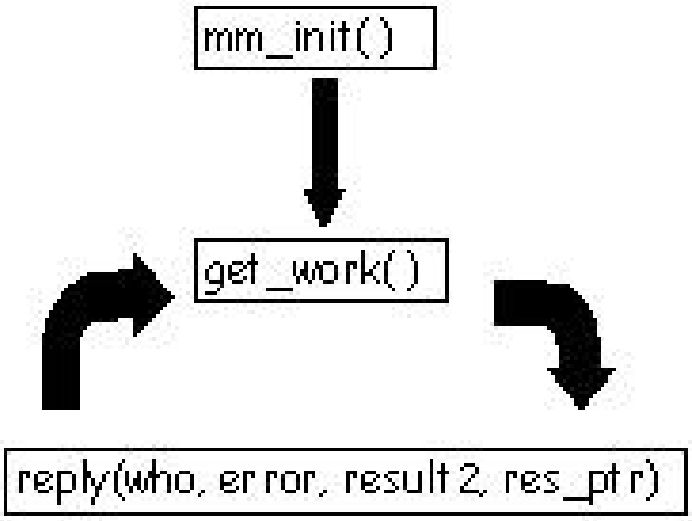
Asignación y desasignación de memoria

Cuando se va asignar memoria a un proceso, se busca en la lista el primer hueco lo bastante grande (FIRST FIT). Entonces se asigna el segmento, reduciendo el tamaño del hueco en la cantidad requerida por el segmento, o en el raro caso de una coincidencia exacta, extrayendo el hueco de la lista.
Cuando un proceso termina y es desalojado, su memoria de datos y pila se devuelven a la lista de huecos libres.

Si usa **instrucciones y datos no separados**, esto libera toda su memoria, ya que tales programas nunca tienen una asignación separada de memoria para el texto.
Si usa **instrucciones y datos separados** y ningún otro proceso está compartiendo el texto, la asignación de texto también será devuelta.

Para cada region devuelta, si cualquiera de los dos o ambos vecinos de las regiones son huecos, se fusionan, para que no existan nunca huecos adyacentes. De esta manera, el número, ubicación, y tamaños de los huecos varían contínuamente durante la operación del sistema.

src/mm/main.c



CODIGO DE LA TABLA DE PROCESOS:

/* Esta tabla tiena una ranura por proceso. Contiene toda la información de administración de memoria para cada proceso. Entre otras cosas define los segementos de texto, dato y pila, uids y gids, y varias banderas. El kernel y el FS tienen tablas indizadas también por proceso, y el contenido de las ranuras correspondiente se refiere al mismo proceso en las tres.
*/

```
EXTERN struct mproc {

        struct mem_map mp_seg[NR_SEGS];         /* apunta a texto, dato y pila */
        char mp_exitstatus;             /* almac. P/situac. si proceso sale */
        char mp_sigstatus;              /* almac. p/num. Señal p/proc matado */
        pid_t mp_pid;                   /* id del proceso */
        pid_t mp_procgrp;               /* pid del grupo de proc. (p/señales)*/
        pid_t mp_wpid;                  /* pid que este proceso espera */
        int mp_parent;                  /* índice de proceso padre*/

        /* uids y gids reales y efectivos. */
        uid_t mp_realuid;       /* uid real del proceso*/
        uid_t mp_effuid;        /* uid efectivo del proceso*/
        gid_t mp_realgid;       /* gid real del proceso*/
        gid_t mp_effgid;        /* gid efectivo del proceso*/

        /*Identificación de archivo para compartir. */
        ino_t mp_ino;           /* númerode nodo-i del archivo */
        dev_t mp_dev;           /* número de dispositivo del sistema de archivos */
        time_t mp_ctime;        /* nodo-i que cambió el tiempo */

        /* Información para manejo de señales. */
        sigset_t mp_ignore;     /* l = ignorar señal, 0 = no */
        sigset_t mp_catch;      /* l = atrapar señal, 0 = no */
        sigset_t mp_sigmask;            /* señales que se bloquearán */
        sigset_t mp_sigmask2;           /* copia guardada de mp_sigmask */
        sigset_t mp_sigpending;         /* señales que se bloquean */
        struct sigaction mp_sigact[_NSIG + 1];          /* como en sigaction(2) */
        vir_bytes mp_sigreturn;         /* dir. de func __sigreturn de bibl. C */

        /* Compatibilidad hacia atrás de las señales. */
        sighandler_t mp_func;           /* todas señales remit. A 1 fcn usuario */
        unsigned mp_flags;              /* bits de bandera*/
        vir_bytes mp_procargs;          /* apunt a args de pila inic de proc */
} mproc[NR_PROCS];
```

```
/* Valores de banderas */

#define IN_USE          001   /* 1 si ranura 'mproc' en uso */
#define WAITING         002   /* encend. por llamada sis. WAIT  */
#define HANGING         004   /* encend. por llamada sis. EXIT  */
#define PAUSED          010   /* encend. por llamada sis. PAUSE  */
#define ALARM_ON        020   /* enc. al niciarse tempor. SIGALRM  */
#define SEPARATE        040   /* 1 si arch. Es espacio I & D separado */
#define TRACED          0100  /* 1 si el proceso debe rastrearse */
#define STOPPED         0200  /* 1 si proc. Detenido por rastreo */
#define SIGSUSPENDED    0400  /* encend. por llamada sis. SIGSUSPEND  */
#define NIL_MPROC ((struct mproc *) 0)
```

CODIGO DE MAIN.C:

```
/*************************************************************
                src/mm/main.c
*************************************************************/
```

/* Este archivo contiene el prog. Pricipal del adm. De memoria y algunos procedinientos relacionados. Cuando MINIX arranca, el kernel se ejecuta un rato, inicializándose a sí mismo y a sus tareas, y luego se ejecuta MM y FS.
Ambos se inicializan hasta donde pueden . Luego FS llama a MM porque MM debe esperar a que FS adquiera un disco en RAM. MM pide al kernel toda la memoria libre y comienza a atender solicitudes.

Los puntos de entradaa este archivo son:
main: pone en marcha el MM
reply: responder a un proceso que efectúa llamada al sistema de MM
*/

```
PUBLIC void main( )
{
        /* Rutina principal del administrador de memoria .*/
        int error;
        mm_init( );                     /* inicializar tablas del administrador de memoria */

        /* Ciclo principal del MM- obtener trabajo y efectuarlo eternamente. */
        while (TRUE)
        {
                /* Esperar un mensaje. */
                get_work( );            /* esperar llamada al sistema MM */
                mp = &mproc[who];

                /* Izar algunas banderas. */
```

```
                error = OK;
                dont_reply = FALSE;
                err_code = -999;

                /* Si núm. de llamada válido, ejecutar la llamada . */
                if (mm_call < 0 || mm_call >= NCALLS)
                        error = EBADCALL;
                else
                        error = (*call_vec[mm_call])();

                /* Devolver resultados al usuario p/indicar que terminamos. */
                if (dont_reply) continue;       /* no reply for EXIT and WAIT */
                if (mm_call == EXEC && error == OK) continue;
                reply(who, error, result2, res_ptr);
        }
}
```

CODIGO DE TABLE.C:

```
/*************************************************************
                src/mm/table.c
*************************************************************/

/* este archivo contien la tabla que sirve para transformar los números de llamada al
sistema en las rutinas que las ejecutan.
 */

#define _TABLE

#include "mm.h"
#include <minix/callnr.h>
#include <signal.h>
#include "mproc.h"
#include "param.h"

/* Miscellaneous */
char core_name[] = "core";    /* archivo donde se prod. imág. núcleo */
_PROTOTYPE (int (*call_vec[NCALLS]), (void) ) = {
        no_sys,                 /*  0 = no se usa        */
        do_mm_exit,             /*  1 = exit*/
        do_fork,                /*  2 = fork*/
        no_sys,                 /*  3 = read*/
        no_sys,                 /*  4 = write*/
        no_sys,                 /*  5 = open*/
        no_sys,                 /*  6 = close*/
        do_waitpid,             /*  7 = wait*/
        no_sys,                 /*  8 = creat*/
        no_sys,                 /*  9 = link*/
        no_sys,                 /* 10 = unlink*/
```

```
do_waitpid,     /* 11 = waitpid*/
no_sys,         /* 12 = chdir*/
no_sys,         /* 13 = time*/
no_sys,         /* 14 = mknod */
no_sys,         /* 15 = chmod */
no_sys,         /* 16 = chown */
do_brk,         /* 17 = break*/
no_sys,         /* 18 = stat*/
no_sys,         /* 19 = lseek*/
do_getset,      /* 20 = getpid*/
no_sys,         /* 21 = mount*/
no_sys,         /* 22 = umount*/
do_getset,      /* 23 = setuid*/
do_getset,      /* 24 = getuid*/
no_sys,         /* 25 = stime*/
do_trace,       /* 26 = ptrace*/
do_alarm,       /* 27 = alarm*/
no_sys,         /* 28 = fstat*/
do_pause,       /* 29 = pause*/
no_sys,         /* 30 = utime*/
no_sys,         /* 31 = (stty)*/
no_sys,         /* 32 = (gtty)*/
no_sys,         /* 33 = access*/
no_sys,         /* 34 = (nice)*/
no_sys,         /* 35 = (ftime) */
no_sys,         /* 36 = sync*/
do_kill,        /* 37 = kill*/
no_sys,         /* 38 = rename*/
no_sys,         /* 39 = mkdir*/
no_sys,         /* 40 = rmdir*/
no_sys,         /* 41 = dup*/
no_sys,         /* 42 = pipe*/
no_sys,         /* 43 = times*/
no_sys,         /* 44 = (prof)*/
no_sys,         /* 45 = no se usa */
do_getset,      /* 46 = setgid*/
do_getset,      /* 47 = getgid*/
no_sys,         /* 48 = (signal)*/
no_sys,         /* 49 = no se usa */
no_sys,         /* 50 = no se usa */
no_sys,         /* 51 = (acct)*/
no_sys,         /* 52 = (phys)  */
no_sys,         /* 53 = (lock)*/
no_sys,         /* 54 = ioctl*/
no_sys,         /* 55 = fcntl*/
no_sys,         /* 56 = (mpx)*/
no_sys,         /* 57 = no se usa */
no_sys,         /* 58 = no se usa */
do_exec,        /* 59 = execve */
no_sys,         /* 60 = umask  */
```

```
        no_sys,             /* 61 = chroot */
        do_getset,          /* 62 = setsid*/
        do_getset,          /* 63 = getpgrp*/
        do_ksig,            /* 64 = KSIG: señales originadas en el kernel*/
        no_sys,             /* 65 = UNPAUSE*/
        no_sys,             /* 66 = no se usa */
        no_sys,             /* 67 = REVIVE*/
        no_sys,             /* 68 = TASK_REPLY  */
        no_sys,             /* 69 = no se usa */
        no_sys,             /* 70 = no se usa */
        do_sigaction,       /* 71 = sigaction   */
        do_sigsuspend,      /* 72 = sigsuspend  */
        do_sigpending,      /* 73 = sigpending  */
        do_sigprocmask,     /* 74 = sigprocmask */
        do_sigreturn,       /* 75 = sigreturn   */
        do_reboot,          /* 76 = reboot*/
};
```

CODIGO DE MM_INIT:

```
PRIVATE void mm_init( )
{
        /* Inicializar el administrador de memoria. */

        static char core_sigs[] = {
                SIGQUIT, SIGILL, SIGTRAP, SIGABRT,
                SIGEMT, SIGFPE, SIGUSR1, SIGSEGV,
                SIGUSR2, 0 };
        register int proc_nr;
        register struct mproc *rmp;
        register char *sig_ptr;
        phys_clicks ram_clicks, total_clicks, minix_clicks, free_clicks, dummy;
        message mess;
        struct mem_map kernel_map[NR_SEGS];
        int mem;

        /* Crear el conjunto de señales que causan vaciados de núcleo. Hacerlo estilo
        Posix, así que no necesitamos conocer posiciones de bits.
        */
        sigemptyset(&core_sset);
        for (sig_ptr = core_sigs; *sig_ptr != 0; sig_ptr++)
                sigaddset(&core_sset, *sig_ptr);

        /* Obtener mapa de memoria del kernel para ver cuánta memoria usa, includo
        el espacio entre la dir. 0 y el principio del kernel.
        */

        sys_getmap(SYSTASK, kernel_map);
        minix_clicks = kernel_map[S].mem_phys + kernel_map[S].mem_len;

        /* Inicializar tablas del MM. */
```

```
        for (proc_nr = 0; proc_nr <= INIT_PROC_NR; proc_nr++)
        {
                rmp = &mproc[proc_nr];
                rmp->mp_flags |= IN_USE;
                sys_getmap(proc_nr, rmp->mp_seg);
                if (rmp->mp_seg[T].mem_len != 0) rmp->mp_flags |= SEPARATE;
                minix_clicks += (rmp->mp_seg[S].mem_phys +
                        rmp-> mp_seg[S].mem_len) -
                        rmp->mp_seg[T].mem_phys;
        }
        mproc[INIT_PROC_NR].mp_pid = INIT_PID;
        sigemptyset(&mproc[INIT_PROC_NR].mp_ignore);
        sigemptyset(&mproc[INIT_PROC_NR].mp_catch);
        procs_in_use = LOW_USER + 1;



        /* Esperar que FS envíe un mensaje con tam. De disco RAM y entre "en línea. */
        if (receive(FS_PROC_NR, &mess) != OK)
                panic("MM can't obtain RAM disk size from FS", NO_NUM);

        ram_clicks = mess.m1_i1;

        /* Inicializar tablas a toda la memoría física. */
        mem_init(&total_clicks, &free_clicks);

        /* Imprimir información de memoria. */
        printf("\nMemory size =%5dK   ", click_to_round_k(total_clicks));
        printf("MINIX =%4dK   ", click_to_round_k(minix_clicks));
        printf("RAM disk =%5dK   ", click_to_round_k(ram_clicks));
        printf("Available =%5dK\n\n", click_to_round_k(free_clicks));

        /* Decir a FS que continue. */
        if (send(FS_PROC_NR, &mess) != OK)
                panic("MM can't sync up with FS", NO_NUM);

        /* Decir a la tarea de mem. dónde está mi tabla de proc. para bien de ps(1). */
        if ((mem = open("/dev/mem", O_RDWR)) != -1)
        {
                ioctl(mem, MIOCSPSINFO, (void *) mproc);
                close(mem);
        }
}
```

CODIGO DE GET_WORK:

```
PRIVATE void get_work()
{
        /* Esperar siguiente mensaje y extraer información útil de él */
```

```
        if (receive(ANY, &mm_in) != OK) panic("MM receive error", NO_NUM);
        who = mm_in.m_source;                   /* uqién envió el mensaje */
        mm_call = mm_in.m_type;                 /* núm. de llamada al sistema */
}
```

CODIGO DE REPLY:

```
PUBLIC void reply(proc_nr, result, res2, respt)
int proc_nr;                    /* proceso al cual responder */
int result;                     /* resultado de la llamada (usually OK or error #)*/
int res2;                       /* resultado secundario */
char *respt;                    /* resultado si apuntador */
{
        /* Enviar una respuesta a un proceso de usuario. */

        register struct mproc *proc_ptr;

        proc_ptr = &mproc[proc_nr];

        /*
        * Para hacer a MM robusto, ver si el destino sigue vivo.
        * Debe omitirse esta verificación si el invocador es una tarea.
        */
        if ((who >=0) && ((proc_ptr->mp_flags&IN_USE) == 0 ||
                (proc_ptr->mp_flags&HANGING))) return;

        reply_type = result;
        reply_i1 = res2;
        reply_p1 = respt;
        if (send(proc_nr, &mm_out) != OK) panic("MM can't reply", NO_NUM);
}
```

CUESTIONES

1.- ¿En qué momento se realiza la asignación de memoria a un proceso?

La asignación de memoria a un proceso se puede realizar en dos momentos diferentes: cuando un proceso se ramifica o cuando ejecuta un programa. En el primer caso, un proceso ejecuta la primitiva FORK para crear un proceso hijo, a este nuevo proceso se le asigna la misma cantidad de memoria que tiene el padre. En el segundo caso, un proceso ya existente (tiene asignada memoria) ejecuta un programa mediante la primitiva EXEC; esto hace que se libere la memoria que el proceso tenía asignada y a continuación se le asigne la cantidad de memoria indicada en la cabecera del fichero que va a ejecutar, evitando que se creen huecos. La memoria se libera siempre que el proceso termina, ya sea porque salga del sistema o porque sea eliminado por una señal (EXIT y KILL).

2.-¿En qué característica queda descrita que se trata por separado el mecanismo de la política?

En el momento en el que el administrador de memoria queda separado del kernel.

3.-¿Por qué el sistema de administración de Minix es tan sencillo?

Porque no emplea ni paginación ni intercambio, sólo una tabla de huecos.

4.- Explicar cómo se lleva a cabo el procesamiento de mensajes.

Una vez que se ha recibido la correspondiente llamada, se decide el proceso que debe ejecutarse por medio de un vector de procedimientos previamente inicializado. Este vector se declara en el fichero "/mm/table.c" como int *call_vec[NCALLS]. El ciclo central del administrador de la memoria extrae el tipo de mensaje y lo almacena en mm_call que se utiliza como índice en el vector (call_vec) para determinar el apuntador al procedimiento que maneja el mensaje recién llegado. Por ejemplo, si tenemos que call_vec[mm_call1]=do_xxx quiere decir que cuando se reciba la llamada al sistema mm_call1 se ejecutará el procedimiento do_xxx.

5.- ¿Cuáles son las estructuras de datos más importantes para el administrador de memoria?. Comentar.

El administrador de memoria tiene dos estructuras de datos principales: la Tabla de Procesos (MPROC) y la Tabla de Huecos (HOLE). Como es sabido, algunos campos de la tabla de procesos se necesitan para el manejo de procesos, otros para la administración de la memoria y otros más para el sistema de archivos. En MINIX cada una de estas tres partes del sistema operativo tiene su tabla de procesos, la cual contiene sólo aquellos campos que necesita. Las entradas corresponden exactamente, para mantener las cosas simples. Así la entrada k-ésima de MPROC (tabla de procesos del manejador de memoria) corresponde al mismo proceso de la entrada k de PROC (tabla de procesos del Kernel) y de FPROC (tabla de procesos del FS). Cuando se crea o destruye un proceso se actualizan las tres tablas.

	Virtual	Física	Longitud
Pila	0x20	0x340	0x8
Datos	0	0x320	0x1c
Texto	0	0x320	0

6.- Diferencias entre el registro de asignación de memoria entre procesos con espacio I-D separado y no separado.

Podemos encontrarnos con una representación de instucciones y datos (I-D) no separados, en cuyo caso quedan así los campos de dirección virtual, física y longitud del proceso:

	Virtual	Física	Longitud
Pila	0x14	0x340	0x8
Datos	0	0x32c	0x10
Texto	0	0x320	0xc

En este caso el campo longitud de texto es siempre cero, ya que ésta se representa junto con los datos. Si se representa instrucciones y datos por separado obtenemos:

Donde la longitud del segmento de texto debe ser distinta de cero.

7.- ¿Cuáles son y para qué se utilizan los procedimientos a los que llama el FS en la inicialización del sistema?

do_brk2 ( ): Este procedimiento lo llama el File System durante la inicialización del sistema, y sirve para marcar como ocupada en la tabla de huecos libres la memoria utilizada por MINIX y el disco RAM. El tamaño de MINIX viene dado por la dirección donde cornienza INIT y su longitud, ya que es la frontera del Sistema Operativo.

Get_mem ( ): Este procedimiento es llamado por do_brk2 para comprobar si existe memoria extendida. Si es memoria extendida se refiere a la que está por encima de 1 MB, la cual no sirve para ubicar los programas pero sí el disco RAM.

8.- Explique detalladamente cuál es la función del procedimiento mm_init() y cómo la realiza.

La función del procedimiento mm_init() es la inicialización de las tablas del manejador de memoria. Para ello realiza los siguientes pasos:

1. Inicializar el vector "core_sigs[]" con las señales que provocan "core dumps" y a continuación construir el conjunto de señales (core_sset).
2. Obtener el mapa de memoria del kernel para ver cuánta memoria usa.
3. Para cada proceso se inicializa su entrada en la tabla de procesos del manejador de memoria y se obtiene su mapa de memoria para ver cuánta utiliza. Además se determina si el proceso utiliza espacio de instrucciones y datos separados o no.
4. Con los datos obtenidos en el apartado anterior, se calcula la cantidad de memoria utilizada por el MINIX (en clicks).
5. Construye el conjunto de señales a ignorar y el conjunto de señales a capturar.
6. Obtiene del FS el tamaño del disco RAM.
7. Muestra la información de memoria.
8. Indica al FS que continúe.
9. Indicar a la tarea de memoria dónde está la tabla de procesos.