Minix 2.0

Inicialización del Minix 2.0

Clossan Andrade Meneces D'Alva
Nayra Quesada Gutiérrez

Contexto

- La imagen del sistema operativo ya está cargada en memoria y ahora hay que **inicializar** las estructuras necesarias para la correcta ejecución del mismo

- Los ficheros **start.c** y **main.c** se encargarán de inicializar todas estas estructuras y dar comienzo a la ejecución del propio sistema operativo

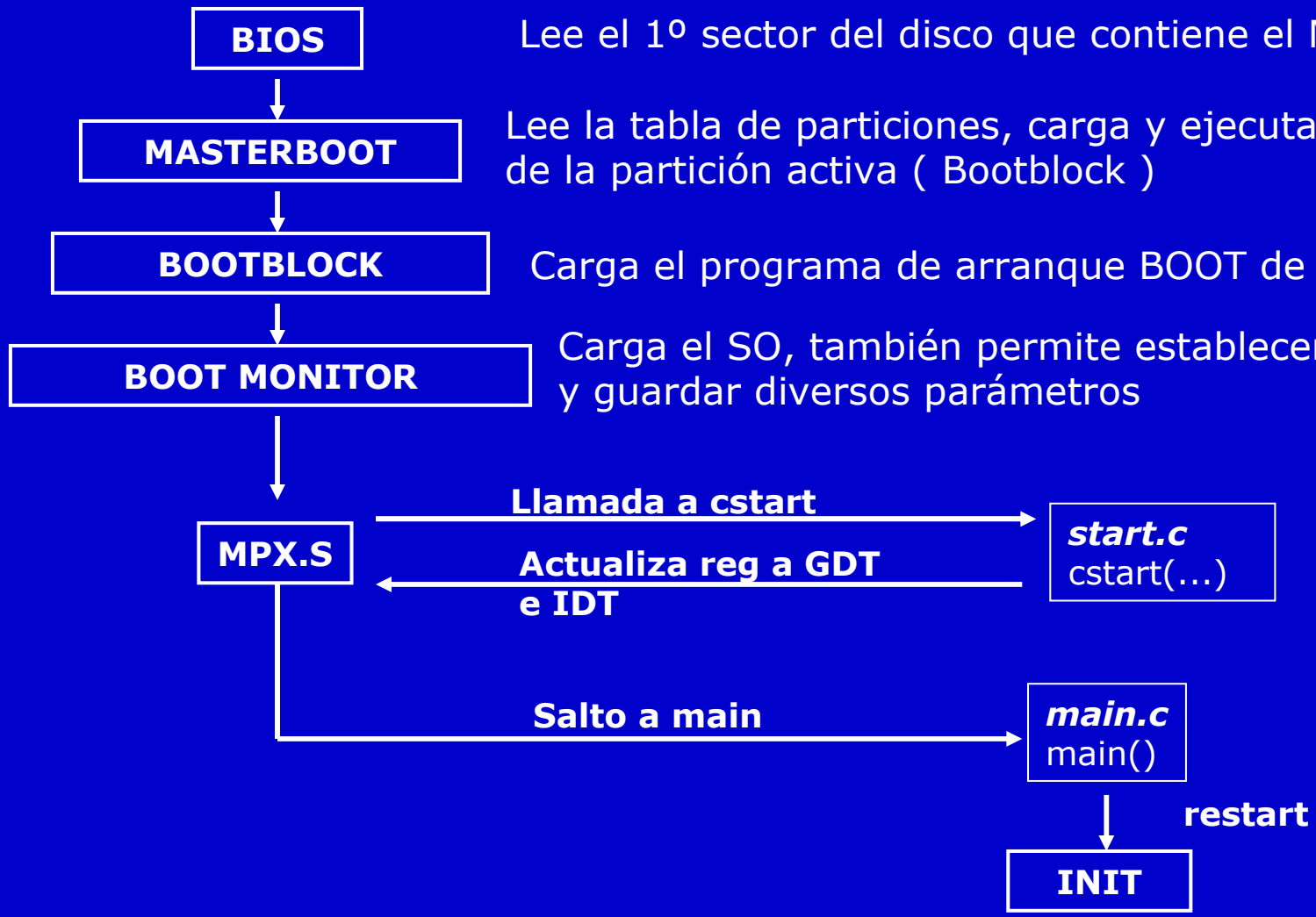
- Nos dirigimos hacia la ejecución de todas las tareas del sistema, la primera de las cuales es la ejecución del proceso **INIT**

Inicialización del MINIX 2.0
Introducción

En el MINIX 2.0 se ha reorganizado el código para que sea independiente de la máquina, de forma que sea mas fácil transportarlo a otras plataformas

Para conseguirlo se ha creado un conjunto de ficheros escritos en ensamblador y otro en C

Inicialización del MINIX 2.0
Esquema de arranque



Lee el 1º sector del disco que contiene el Masterboot

Lee la tabla de particiones, carga y ejecuta el 1º sector de la partición activa (Bootblock)

Carga el programa de arranque BOOT de su partición

Carga el SO, también permite establecer, modificar y guardar diversos parámetros

Inicialización del MINIX 2.0
Ficheros que intervienen

- **mpx386.s**
  Rutina encargada de la inicialización para máquinas de 32 bits. Tras una serie de pasos llama a la función **cstart** y finalmente a main().

- **start.c**
  Archivo que contiene el código de inicialización en C para Minix sobre procesadores Intel. Coopera con mpx386.s para preparar un buen entorno para main().

- **main.c**
  Archivo que contiene la rutina main(), la cual inicializa el sistema (intr, tabla de procesos, ...) y comienza el proceso de ejecución.

- **table.c**
  Archivo donde se definen los atributos (nombre, requerimiento de pila,etc...) de las tareas de sistema.

Minix 2.0

Start.c

Start.c Descripción

- **S**e implementan en este fichero las siguientes funciones:

cstart *(cs, ds, mcs, mds, parmoff, parmsize)*

Es el procedimiento principal de start.c, su función es inicializar:

·La **tabla de descriptores globales:** Es una estructura de datos central empleada por los procesadores Intel de 32 bits para supervisar la protección de memoria.

·La **tabla de descriptores de interrupciones:** Sirve para seleccionar el código que se ejecutará al ocurrir cada tipo de interrupción

·Una serie de **variables de entorno**, tales como el tipo de display, procesador y bus.

k_getenv: Función a la que se le pasa una variable de entorno y devuelve un puntero a la cadena con el valor de la variable.

k_atoi: Función que convierte ristra a entero.

Start.c

Variables de entorno pasadas por el boot

```
#include "kernel.h"
#include <stdlib.h>
#include <minix/boot.h>
#include "protect.h"
/* Cadena de entorno pasada por el cargador*/
PRIVATE char k_environ[256];
FORWARD _PROTOTYPE( int k_atoi, (char *s) );

PUBLIC void cstart(cs, ds, mcs, mds, parmoff, parmsize)
U16_t cs, ds;                 Segmento de código y datos del kernel
U16_t mcs, mds;               Segmento de código y datos del monitor
U16_t parmoff, parmsize;      Desplazamiento y tamaño de los
{                             parámetros de arranque

  register char *envp;
  phys_bytes mcode_base, mdata_base;
  unsigned mon_start;
```

Función **cstart()** y la declaración de sus parámetros

Start.c
Segmentos y prot_ini()

```
{
  register char *envp;
  phys_bytes mcode_base, mdata_base;
  unsigned mon_start;

  code_base = seg2phys(cs);
  data_base = seg2phys(ds);
  mcode_base = seg2phys(mcs);
  mdata_base = seg2phys(mds);

  prot_init();
  if (parmsize > sizeof k_environ - 2) parmsize = sizeof k_environ - 2;
  phys_copy(mdata_base + parmoff, vir2phys(k_environ),
(phys_bytes)
  parmsize);
```

Graba la localización de los segmentos del Kernel y el monitor

*La función **seg2phys()** retorna la dirección base de un segmento*
code_base,data_base:var. Locales
mcode_base,mdata_base: var.boot.h

Inicializa los mecanismos de protección de CPU y las tablas de interrupciones

Start.c
Parámetros de arranque

```
code_base = seg2phys(cs);
data_base = seg2phys(ds);
mcode_base = seg2phys(mcs);
mdata_base = seg2phys(mds);

prot_init();
if (parmsize > sizeof k_environ - 2) parmsize = sizeof k_environ - 2;
phys_copy(mdata_base + parmoff, vir2phys(k_environ),
(phys_bytes) parmsize);

boot_parameters.bp_rootdev = k_atoi(k_getenv("rootdev"));
boot_parameters.bp_ramimagedev = k_atoi(k_getenv("ramimagedev"));
boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
boot_parameters.bp_processor = k_atoi(k_getenv("processor"));
```

Copia las variables de entorno a la memoria del kernel

Convierte variables de entorno a valores numéricos

Start.c

Display, Memoria, Procesador y Bus

Obtiene información del boot monitor

```
boot_parameters.bp_ramsize = k_atoi(k_getenv("ramsize"));
boot_parameters.bp_processor = k_atoi(k_getenv("processor"));
```

```
envp = k_getenv("video");
```
Lee la variable de entorno "video "

```
if (strcmp(envp, "ega") == 0) ega = TRUE;
if (strcmp(envp, "vga") == 0) vga = ega = TRUE;
```
Se determina el tipo de display a usar

```
low_memsize = k_atoi(k_getenv("memsize"));
ext_memsize = k_atoi(k_getenv("emssize"));
```
Determina el tamaño de la memoria

```
processor = boot_parameters.bp_processor;
```
Se indica el tipo de procesador (86,186,286,386, ..)

```
envp = k_getenv("bus");
if (envp == NIL_PTR || strcmp(envp, "at") == 0) {
        pc_at = TRUE;
} else
        if (strcmp(envp, "mca") == 0)  pc_at = ps_mca = TRUE;
```
Se selecciona el tipo de bus

Start.c
Retornar a mpx.s

Actualiza la dirección de retorno al monitor si lo hay y **vuelve al mpx386**.s para actualizar los registros que apuntan a los descriptores y llamar al main().

```
#if _WORD_SIZE == 2
  protected_mode = processor >= 286;
#endif
```

```
if (!protected_mode) mon_return = 0;
  mon_start = mcode_base / 1024;
  if (mon_return && low_memsize > mon_start) low_memsize =
mon_start;
```

Actualiza la dirección de retorno

Start.c
Funciones k_atoi() y k_getenv()

```
PRIVATE int k_atoi(s)
  register char *s;{
  return strtol(s, (char **) NULL, 10);
}
```

Función que convierte ristra a enteros

```
PUBLIC char *k_getenv(name)
  char *name;{
  register char *namep;
  register char *envp;

  for (envp = k_environ; *envp != 0;) {
        for (namep = name; *namep != 0 && *namep == *envp;
        namep++, envp++);
        if (*namep == '\0' && *envp == '=') return(envp + 1);
        while (*envp++ != 0);
  }
  return(NIL_PTR);
}
```

Función a que se la pasa una variable de entorno y devuelve un puntero a la cadena con el valor de la variable

Minix 2.0

Main.c

Main.c
Descripción

- La última instrucción ejecutada por **mpx386.s** es un salto a **main()** que se encarga de completar la inicialización y de comenzar la ejecución normal del sistema.

- Las funciones que se implementan en éste fichero son las siguientes:

  - ***main():*** Se encarga de ...
    - Configurar el hardware de control de interrupciones ***(intr_init)***
    - Establece la tabla de procesos para tareas y servidores
    - Interpreta los tamaños de memoria
    - Planifica la ejecución de las tareas para que se ejecuten por si mismas

  - ***panic(s,n):*** Procedimiento invocado por el sistema cuando se produce una situación que le hace imposible continuar.

Main.c
Tabla de procesos

La **tabla de procesos** es un **vector** en el cual en **cada posición** se almacena información **necesaria** para que los **procesos puedan ser ejecutado**.
Cada entrada contiene una serie de campos de los cuales main.c hace uso de los siguientes:

0	**p_reg.sp:** Puntero de la pila **p_reg.pc:** Contador de programa **p_reg.psw:** Palabra de estado del programa	**p_nr:** Nº de procesos	**p_flags:** Indica el estado de la posición correspondiente en la tabla de procesos. ·**0 =>** el proceso puede ejecutarse. ·**algún bit = 1** => el proceso no puede ejecutarse · **P_SLOT_FREE=1** => posición de la tabla no uso · **NO_MAP=1 =>** proceso hijo no puede ejecutarse hasta que su mapa de mem se ha creado	**p_map**[Nr_segs] El mapa de memoria, en el cual guardamos la posición de inicio y la longitud de los distintos segmentos (código, datos y pila) **Cada campo** contiene a su vez a otros 3 campos: mem_phys,mem_vir y mem_len
1			Idem	
2			Idem	

p_reg: Estructura con todos los registros de la máquina

Main.c
Vector Sizes

- El vector sizes es utilizado por el sistema para obtener los **tamaños del código y datos** de los diferentes módulos que componen el sistema operativo (kernel, mm, fs e init), y si se activa el **servicio de red**, existirán dos posiciones más.

- El array sizes fue introducido en el área de datos del kernel por boot antes de que el kernel iniciara su ejecución

- La utilización de este vector posibilita la modificación de los procesos del sistema (kernel, mm, fs, init), sin que esto suponga alguno en la recompilación del núcleo, o sea, no es necesaria la modificación del main() en tales casos

Main.c
Función main() y declaración de variables

```
...
#include "proc.h"

PUBLIC void main(){
  register struct proc *rp;      /* puntero a la tabla de procesos */
  register int t;                /* identificador de tareas y procesos */
  int sizeindex;                 /* índice para vector sizes */
  phys_clicks text_base;         /* dirección de comienzo de código */
  vir_clicks text_clicks;        /* tamaño de código (sizes) */
  vir_clicks data_clicks;        /* tamaño de datos (sizes)*/
  phys_bytes phys_b;
  reg_t ktsb;                    /* base de la pila de tareas de kernel */
  struct memory *memp;
  struct tasktab *ttp;           /* puntero a vector con requerimientos de pila*/

   intr_init(1);
...
```

Main.c
intr_init(n) y mem_init()

intr_init(n): Función que configura el controlador hardware de interrupciones. Según el valor del parámetro se le indica que MINIX se está iniciando (1) o que reinicie el hardware de interrupciones a sus estados originales (0).

```
…
struct memory *memp;
struct tasktab *ttp;

intr_init(1);
mem_init();

for (rp = BEG_PROC_ADDR, …
```

Asegura que en el proceso de inicialización ninguna interrupción tengan efecto. Para ello:

- Inhibe entradas de cada int. en controlador hw
- Imprime mensajes si se recibe alguna int. en vector de interrupciones.
- A medida que se van ejecutando las tareas de E/S se van rellenando los valores originales de cada entrada del vector.

mem_init(): Inicializa un array que define la localización y el tamaño de cada porción de memoria disponible en el sistema. Esta función también depende del hardware por lo que se separa.

Main.c
Vaciado de la tabla de Procesos

La siguiente tarea que realiza el main consiste en marcar las entradas de la tabla de procesos como libres y les asigna un número identificativo.

Inicializa el vector ***pproc_addr*** que agiliza el acceso a la tabla de procesos

```
...
 mem_init();
  for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t)
  {
        rp->p_flags = P_SLOT_FREE;
        rp->p_nr = t;
        (pproc_addr + NR_TASKS)[t] = rp;
   }
   ktsb = (reg_t) t_stack;
...
```

entrada libre

número de proceso

puntero a proceso

Main.c
Inicializar la pila de tareas

- Todas las tareas están compiladas dentro del mismo fichero que el **kernel**, y la información acerca de sus requerimientos de pila está en un **array** llamado ***tasktab***, definido en **table.c**.
- Dado que las tareas son compiladas dentro del núcleo y pueden utilizar código y acceder a datos localizados en cualquier lugar del Kernel, el tamaño de una tarea individual no es relevante y su tamaño no es significativo ya que utilizará toda la porción del código del Kernel.

```
...
        (pproc_addr + NR_TASKS)[t] = rp;
        }
        ktsb = (reg_t) t_stack;

        for (t = -NR_TASKS; t <= LOW_USER; ++t) {
...
```

Inicializa pila de tareas

Main.c
Bucle Principal

- Bucle principal de **main**, donde se inicializa la tabla de procesos con la información necesaria para ejecutar las tareas de sistema, los servidores y el **INIT**.

proc_addr(n), es una macro necesaria para usar subíndices negativos. Se asigna a **rp** la dirección de una entrada de la tabla de procesos

```
...
 for (t = -NR_TASKS; t <= LOW_USER; ++t)
 {
          rp = proc_addr(t);
          ttp = &tasktab[t + NR_TASKS];
          strcpy(rp->p_name, ttp->name);

          if (t < 0) {
                    if (ttp->stksize > 0) {
...
```

/* entrada en tabla de procesos del proceso t */

Se determina la entrada en la tabla de procesos y sus atributos en el vector ***tasktab***.

Main.c
Bucle principal

Si se trata de una tarea de sistema entonces se asigna puntero de guardia para la pila, puntero de pila y el mismo puntero base que el del kernel.

```
...
if (t < 0) {
    if (ttp->stksize > 0) {                  /* Si requerimiento de pila mayor que 0*/
       rp->p_stguard = (reg_t *) ktsb;       /* asigna puntero guardia a pila */
       *rp->p_stguard = STACK_GUARD;
    }
    ktsb += ttp->stksize;          /* se incluye la pila del proceso en la del kernel */
    rp->p_reg.sp = ktsb;           /* puntero de pila del proceso a base del kernel */
    text_base = code_base >> CLICK_SHIFT;
    sizeindex = 0;                 /* indice a código del kernel en sizes  */
    memp = &mem[0];
} else {
    sizeindex = 2 * t + 2;
}
...
```

Se asegura que se accederá al elemento número 0 de ***sizes*** para todas las tareas

Si no es tarea de sistema (servidores e INIT) se actualiza índice para acceder a sizes

Main.c
Bucle Principal

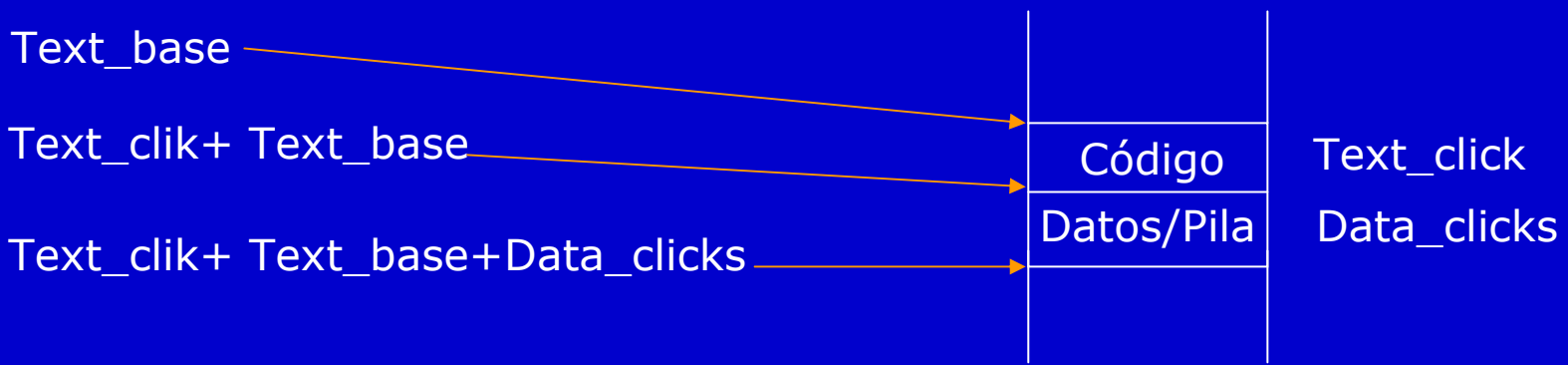
Inicializa el contador de programa (PC) y la palabra de estado del programa (PSW)

```
...
        sizeindex = 2 * t + 2;
    }
        rp->p_reg.pc = (reg_t) ttp->initial_pc;   /* actualiza PC y PSW */
        rp->p_reg.psw = istaskp(rp) ? INIT_TASK_PSW : INIT_PSW;

        text_clicks = sizes[sizeindex];
        data_clicks = sizes[sizeindex + 1];
...
```

Main.c
Bucle Principal

- Es en este momento cuando se le asigna a cada proceso su localización en memoria. Para ello el sistema utiliza el vector sizes.
- Para cada proceso se mapea memoria para código, datos y pila de la siguiente forma:
  - Las tareas poseen todas la misma posición base y tamaño, y que es la del kernel.
  - El segmento de datos es situado a continuación del segmento de código y la posición de la pila está a continuación de ésta ocupando la misma zona de memoria



Main.c
Bucle Principal

Mapear memoria para código, datos y pila

Dejar los índices listos para la siguiente iteración

Marcar la memoria ocupada por el proceso

```
text_clicks = sizes[sizeindex];                    /* tamaño de código de tarea*/
data_clicks = sizes[sizeindex + 1];                /* tamaño de datos de tarea*/
rp->p_map[T].mem_phys = text_base;                 /* mapea memoria para código */
rp->p_map[T].mem_len  = text_clicks;
rp->p_map[D].mem_phys = text_base + text_clicks;    /* para datos */
rp->p_map[D].mem_len  = data_clicks;
rp->p_map[S].mem_phys = text_base + text_clicks + data_clicks; /* para pila*/
rp->p_map[S].mem_vir  = data_clicks;
text_base += text_clicks + data_clicks;   /*actualiza dir  de comienzo de código */
memp->size -= (text_base - memp->base); /* actualiza memoria ocupada */
memp->base = text_base;
```

Main.c
Bucle Principal

Inicializa el puntero de pila del servidor y reserva una palabra para que crtso.s la use como argc.

```
...
memp->size -= (text_base - memp->base);
memp->base = text_base;

if (t >= 0) {
   rp->p_reg.sp = (rp->p_map[S].mem_vir +
                   rp->p_map[S].mem_len) << CLICK_SHIFT;
   rp->p_reg.sp -= sizeof(reg_t);
}
...
```

si es servidor iniciar el puntero a pila de servidores

guardar una palabra para argc

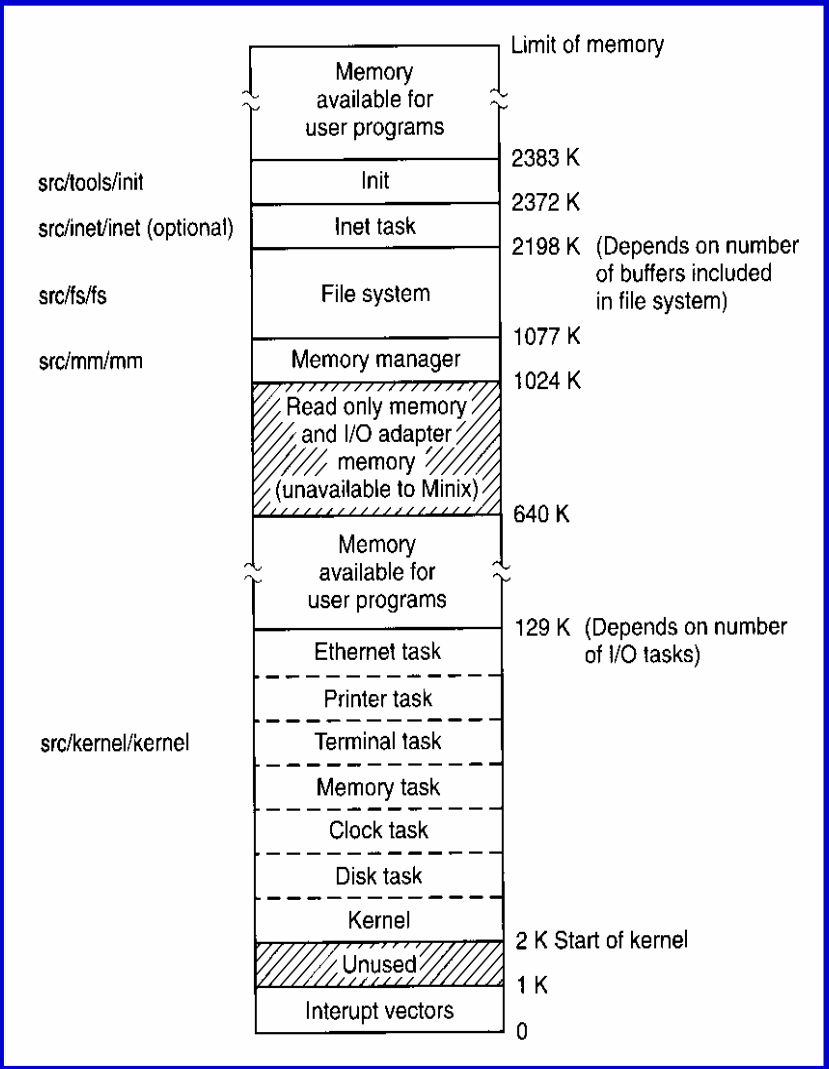
Main.c
Bucle Principal

Debido a problemas de compatibilidad, las máquinas modernas todavía tienen la memoria de sólo lectura (Rom) en las mismas direcciones que las máquinas antiguas.

Con lo cual la memoria Ram es discontinua con un bloque de Rom entre entre 640 k y 1024 k.

El monitor de arranque carga los servidores e INIT en el intervalo de memoria situado por arriba de la Rom.

Esto se hace principalmente pensando en el sistema de fichero, a fin de que pueda usar una caché de bloques muy grande sin tropesarse con la Rom.



Main.c
Bucle Principal

Se presupone que en la siguiente iteración el proceso t no será una tarea del sistema, por lo que deberá estar situado en la zona alta de memoria (> 1024 k)

El código condicional que asegura que este uso del área de memoria alta quede registrado en la tabla de procesos.

```
#if _WORD_SIZE == 4
      if (t < 0) {    /* carga servidor en memoria extendida si modo protegido*/
          memp = &mem[1];
          text_base = 0x100000 >> CLICK_SHIFT;
      }
#endif

if (!isidlehardware(t)) lock_ready(rp);/* prepara proceso ejecutable en cola */
          rp->p_flags = 0;             /* marca el proceso como ejecutable */
          alloc_segments(rp);          /* asigna segmento a proceso */
}
```

Main.c
Bucle Principal

- Los procesos IDLE y HARDWARE requieren un tratamiento especial.
  - El **proceso IDLE** es un bucle que no hace nada y que se ejecuta cuando no hay ningún otro proceso en ejecución.
  - El **proceso HARDWARE** existe para propósitos de contabilización de t, relacionado con el manejo de interrupciones.

isidlehardware(t): Función que pregunta si la tarea t es la tarea ociosa (IDLE) o la HARDWARE y coloca todos los demás procesos en las colas apropiadas.

lock_ready(rp): Llama a la rutina ready(rp), el cual coloca el proceso rp en una de las colas de procesos ejecutables. Estas colas son:

• **TASK_Q**	(Mayor prioridad)	*Tareas ejecutables .*
• **SERVER_Q**	(Prioridad intermedia)	*MM y FS sólamennte*
• **USER_Q**	(Menor prioridad)	*Procesos de usuario*

```
if (!isidlehardware(t)) lock_ready(rp);  // prepara proceso ejecutable en cola
rp->p_flags = 0;                         // marca el proceso como ejecutable
alloc_segments(rp);                       // asigna los segmentos del proceso
…
```

Main.c
Bucle Principal

Marcar el proceso como ejecutable.

alloc_segments(rp): Función dependiente de la máquina que asigna a sus campos correspondientes la localización, tamaño y nivel de permiso para los segmentos de memoria usados por cada proceso.

```
if (!isidlehardware(t)) lock_ready(rp);  // prepara proceso ejecutable en cola
   rp->p_flags = 0;
   alloc_segments(rp);                   // asigna los segmentos del proceso

}          //fin del bucle principal
```

Main.c

Preparar siguiente proceso a ejecutar

Se asigna pid=1 al proceso INIT

- Mediante las variable ***bill_ptr*** *y* ***proc_ptr*** se decide qué proceso se va a ejecutar a continuación.
- *bill_ptr* es el proceso que va a salir de ejecución y *proc_ptr* es el que va a entrar. Como *bill_ptr* tiene que tener un valor inicial,se le asigna la dirección del proceso IDLE, más adelante será modificado por **lock_pick_proc**.

```
proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;

bill_ptr = proc_addr(IDLE);          /* apunta como sig proc IDLE */

  lock_pick_proc();

  restart();
}
```

Main.c

Encolar INIT como siguiente proceso a ejecutar

lock_pick_proc(): Se encargará de modificar el proceso que está activo en el procesador y hará que la variable *proc_ptr* apunte a la entrada en la tabla de procesos del próximo proceso a ejecutar.

En esta parte de la inicialización el puntero *proc_ptr* contendrá la entrada para la tarea de consola, la cual es la primera en ser ejecutada.

```
proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;

bill_ptr = proc_addr(IDLE);          /* apunta como sig proc IDLE */

lock_pick_proc();              /* coloca siguiente proceso a ejecutarse */

restart();
}
```

Main.c
Iniciar ejecución de tarea

restart():

- Rutina que habilita las interrupciones e inicia la ejecución de una tarea o proceso.
- Pasa el control a la rutina que reinicia el funcionamiento normal de Minix.
- El control nunca volverá a main.

```
proc[NR_TASKS+INIT_PROC_NR].p_pid = 1;

bill_ptr = proc_addr(IDLE);          /* apunta como sig proc IDLE */

lock_pick_proc();            /* coloca siguiente proceso a ejecutarse */

restart();
}
```

Main.c
Función panic()

Panic(s,n): Este procedimiento se invoca cuando el sistema no puede continuar.

```
PUBLIC void panic(s,n)
_CONST char *s;
int n;{
 if (*s != 0) {
     printf("\nKernel panic: %s",s);
     if (n != NO_NUM) printf(" %d", n);
     printf("\n");
  }
  wreboot(RBT_PANIC);
}
```

Situaciones que impiden que el sistema continúe:

- Imposibilidad de leer un bloque crítico del disco
- Detección de un estado interno incoherente
- Llamada de una parte del sistema a otra con parámetros incorrectos

wreboot(): Espera hasta que se pulsa una tecla y luego resetea, cargando el sistema operativo.

Minix 2.0

Table.c

Table.c
Pilas de las tareas

```
#define _TABLE
#include "kernel.h"
#include <termios.h>
#include <minix/com.h>
#include "proc.h"
#include "tty.h"

#define SMALL_STACK      (128 * sizeof(char *))
#define TTY_STACK        (3 * SMALL_STACK)
#define SYN_ALRM_STACK   SMALL_STACK
#define DP8390_STACK     (SMALL_STACK * ENABLE_NETWORKING)
```

Tamaño mínimo de la pila

¿ De qué se encarga ?

Reserva espacio para las **pilas** de las tareas

¿ Qué tareas ?

Las tareas definidas en minix/com.h

Tareas definidas

Este símbolo adquiere su valor en minix/config.h

```
#if (CHIP == INTEL)
#define  IDLE_STACK      ((3+3+4) * sizeof(char *))
#else
#define  IDLE_STACK      SMALL_STACK
#endif
```

En función del CHIP se asignan diferentes tamaños de pilas

Table.c
Pilas de las tareas

La definición de los tamaños de pila vuelve a depender de los valores de símbolos

```
#if (CHIP == INTEL)
#define  WINCH_STACK    (2 * SMALL_STACK * ENABLE_WINI)
#else
#define  WINCH_STACK    (3 * SMALL_STACK * ENABLE_WINI)
#endif

#if (MACHINE == ATARI)
#define  SCSI_STACK     (3 * SMALL_STACK)
#endif

#if (MACHINE == IBM_PC)
#define  SCSI_STACK     (2 * SMALL_STACK * ENABLE_SCSI)
#endif

#define CDROM_STACK     (4 * SMALL_STACK * ENABLE_CDROM)
#define AUDIO_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
#define MIXER_STACK     (4 * SMALL_STACK * ENABLE_AUDIO)
```

Tamaños de pila para dispositivos

Table.c
Pilas de las tareas

```
#define  FLOP_STACK              (3 * SMALL_STACK)
#define  MEM_STACK               SMALL_STACK
#define  CLOCK_STACK             SMALL_STACK
#define  SYS_STACK               SMALL_STACK
#define  HARDWARE_STACK          0

#define TOT_STACK_SPACE          (TTY_STACK + DP8390_STACK +\
         SCSI_STACK +  SYN_ALRM_STACK + IDLE_STACK + \
         HARDWARE_STACK + PRINTER_STACK + \
         WINCH_STACK + FLOP_STACK + MEM_STACK +
         CLOCK_STACK + SYS_STACK + \
         CDROM_STACK + AUDIO_STACK + MIXER_STACK)

#define scsi_task          aha_scsi_task
#define cdrom_task         mcd_task
#define audio_task         dsp_task
```

Total de espacio requerido

Table.c
Estructura tasktab[ ]

- Tasktab [] : En esta estructura se almacenará la información necesaria para conocer el tamaño de pila de cada tarea definida.

```
PUBLIC struct tasktab tasktab[] = {
        { tty_task,          TTY_STACK,      "TTY"          },
#if ENABLE_NETWORKING
        { dp8390_task,       DP8390_STACK,   "DP8390"       },
#endif
#if ENABLE_CDROM
        { cdrom_task,        CDROM_STACK,    "CDROM"        },
#endif
#if ENABLE_AUDIO
        { audio_task,        AUDIO_STACK,    "AUDIO"        },
        { mixer_task,        MIXER_STACK,    "MIXER"        },
#endif
```

Bucle principal de la tarea tty definido en tty.c

Tamaño de la pila asignado

Table.c
Estructura tasktab [ ]

```
#if ENABLE_SCSI
        { scsi_task,            SCSI_STACK,         "SCSI"          },
#endif
#if ENABLE_WINI
        { winchester_task,              WINCH_STACK,
"WINCH"         },
#endif
        { syn_alrm_task,        SYN_ALRM_STACK,  "SYN_AL"     },
        { idle_task,            IDLE_STACK,         "IDLE"          },
        { printer_task,         PRINTER_STACK,     "PRINTER"    },
        { floppy_task,          FLOP_STACK,         "FLOPPY"        },
        { mem_task,             MEM_STACK,          "MEMORY"        },
        { clock_task,           CLOCK_STACK,        "CLOCK"         },
        { sys_task,             SYS_STACK,          "SYS"           },
        { 0,                    HARDWARE_STACK,"HARDWAR"    },
        { 0,                    0,                  "MM"            },
        { 0,                    0,                  "FS"            },
#if ENABLE_NETWORKING
        { 0,                    0,                  "INET"
        },
#endif
        { 0,                    0,                  "INIT"
        },
```

Table.c
Estructura tasktab [ ]

```
PUBLIC char *t_stack[TOT_STACK_SPACE / sizeof(char *)];
```

Tamaño máximo de la pila

```
#define NKT (sizeof tasktab / sizeof (struct tasktab) - (INIT_PROC_NR + 1))
extern int dummy_tasktab_check[NR_TASKS == NKT ? 1 : -1];
```

Minix 2.0

Cuestiones

Cuestiones

- **¿Cuáles son las principales funciones que realiza el main()?**

  Son inicializar el **cont. de interrupciones**, interpretar los **tamaños de memoria**, inicializar las entradas en la **tabla de procesos** para tareas, servidores y **pasar el control** de las funciones que controlan el funcionamiento normal del Minix.

- **¿Y el cstart()?**

  Inicializa la **gdt**, la **tabla de descriptores de interrupciones** y una serie de **variables de entorno**.

Cuestiones

- **¿Qué es la tabla de procesos y para que se emplea?**

  Es un **vector** en el cual en **cada posición** se almacena información necesaria para que los procesos puedan ser ejecutados. (Ej: pid, ...)

- **¿Qué utilidad tiene el vector sizes y que ventajas proporciona?**

  • El vector sizes se utiliza para crear el **mapa de memoria** correspondiente a cada proceso para luego ser incluido en la **tabla de procesos**. Con este vector obtenemos los **tamaños del código** y de los datos de los diferentes módulos que componen el sistema operativo.

  • **Ventaja:** Posibilita la modificación del los procesos del sistema (kernel,MM,FS,INIT), sin que esto suponga problema alguno en la recompilación del núcleo, de modo que no es necesaria la modificación del main en tales casos.