



# Diseño e implementación de Sistemas Operativos

---

*Trabajo práctico 2: Planificación por  
prioridades en Minix*

Año 2006 - 2do cuatrimestre

Grupo:

Gerardo Bassetti (LU 15336) <[gebassetti@gmail.com](mailto:gebassetti@gmail.com)>

Martín Pivideri (LU 15111) <[martinpivideri@gmail.com](mailto:martinpivideri@gmail.com)>

Milton Pivideri (LU 15605) <[miltondp@gmail.com](mailto:miltondp@gmail.com)>

César Sandrigo (LU 15630) <[cmsandrigo@gmail.com](mailto:cmsandrigo@gmail.com)>

Versión final

# Hecho con Software Libre

El trabajo práctico, así como este documento, fueron realizados íntegramente usando Software Libre. Las herramientas utilizadas fueron:

- **L<sup>A</sup>T<sub>E</sub>X** - <http://www.latex-project.org/>  
L<sup>A</sup>T<sub>E</sub>X es un sistema de preparación de documentos. Éste documento está hecho con él.
- **Subversion** - <http://subversion.tigris.org/>  
Subversion es un sistema de control de versiones diseñado específicamente para reemplazar a CVS. Lo utilizamos para versionar tanto el código fuente de Minix como el de éste documento.
- **QEMU** - <http://fabrice.bellard.free.fr/qemu/>  
QEMU es un emulador libre, que se caracteriza por ser rápido.
- **Debian GNU/Linux** - <http://www.debian.org/>  
El Sistema Operativo Universal. Debian es una de las distribuciones de GNU/Linux más utilizadas.
- **GNU Aspell** - <http://aspell.sourceforge.net/>  
GNU Aspell es una utilidad para chequear la ortografía.
- **Vim** - <http://www.vim.org/>  
Vi IMproved es un editor de texto avanzado. Hay disponible una versión gráfica para Windows.



## Índice

<b>1. Sección principal</b>	<b>4</b>
1.1. Constantes de prioridad . . . . .	4
1.2. Estructura del descriptor de proceso . . . . .	5
1.3. Inicialización de la prioridad de un proceso . . . . .	5
1.4. Funcionalidad de la tecla F1 . . . . .	6
1.5. Llamada al sistema . . . . .	8
1.5.1. Comunicación <i>proceso de usuario - Memory Manager</i> . . . . .	8
1.5.2. Memory Manager . . . . .	10
1.5.3. Comunicación <i>Memory manager - SYSTASK</i> . . . . .	11
1.5.4. SYSTASK . . . . .	12
1.6. Comando para modificar la prioridad de un proceso . . . . .	16
1.7. Algoritmo de planificación . . . . .	17
1.7.1. Campos <i>priority</i> y <i>penalty</i> . . . . .	17
1.7.2. Cola de procesos de usuario . . . . .	17
1.8. Test de funcionamiento . . . . .	21
1.9. Organización del disquette . . . . .	22
<b>2. Sección de descargos</b>	<b>23</b>

# 1. Sección principal

## 1.1. Constantes de prioridad

En el archivo `/usr/include/minix/const.h` definimos las 16 constantes que establecen las distintas prioridades que puede tomar un proceso. A continuación se muestran las diferencias con el archivo original (diff -u archivoOriginal archivoModificado) de `const.h`

---

```

--- include/minix/const.h.orig 2006-10-30 23:54:47.000000000 -0300
+++ include/minix/const.h 2006-11-07 00:00:27.000000000 -0300
@@ -106,3 +106,25 @@
#define NO_ZONE          ((zone_t) 0) /* absence of a zone number */
#define NO_DEV           ((dev_t) 0) /* absence of a device numb */

+#ifdef DISO
+/* DISO Priorities */
+#define DISO_PRI00      0x00          /* HIGHEST */
+#define DISO_PRI01      0x01
+#define DISO_PRI02      0x02
+#define DISO_PRI03      0x03
+#define DISO_PRI04      0x04
+#define DISO_PRI05      0x05
+#define DISO_PRI06      0x06
+#define DISO_PRI07      0x07
+#define DISO_PRI08      0x08
+#define DISO_PRI09      0x09
+#define DISO_PRI10      0x0A
+#define DISO_PRI11      0x0B
+#define DISO_PRI12      0x0C
+#define DISO_PRI13      0x0D
+#define DISO_PRI14      0x0E
+#define DISO_PRI15      0x0F          /* LOWEST */
+#define DISO_PRIHIGHEST DISO_PRI00
+#define DISO_PRILOWEST DISO_PRI15
+#define DISO_NR_PRTY     16
+#endif /* DISO */

```

---

Diferencias - `const.h`

## 1.2. Estructura del descriptor de proceso

Además de un campo *priority* en el descriptor de procesos, también incluimos otro, necesario para el funcionamiento de nuestra implementación de las prioridades, llamado *penalty* (más adelante en el informe se describirá su función). Éstas son las diferencias (diff -u archivoOriginal archivoModificado) de *proc.h*

---

```

--- kernel/proc.h.orig 2006-10-27 21:24:23.000000000 -0300
+++ kernel/proc.h 2006-11-04 17:12:00.000000000 -0300
@@ -62,6 +62,11 @@
     sigset_t p_pending;          /* bit map for pending signals */
     unsigned p_pendcount;        /* count of pending and unfinished signals */

+#ifdef DISO
+    int priority;                /* prioridad de procesos de usuario */
+    int penalty;                 /* penalidad */
+#endif /* DISO */
+
+    char p_name[16];             /* name of the process */
+};

```

---

### Diferencias - *proc.h*

## 1.3. Inicialización de la prioridad de un proceso

En la función *main()* del archivo */usr/src/kernel/main.c*, donde se inicializan todos los campos de la estructura de un proceso, inicializamos *priority* y *penalty*. Los cambios hechos en el archivo se muestran a continuación:

---

```

--- kernel/main.c.orig 2006-10-27 21:24:23.000000000 -0300
+++ kernel/main.c 2006-11-02 19:16:00.000000000 -0300
@@ -46,6 +46,12 @@
 * Set up mappings for proc_addr() and proc_number() macros.
 */
for (rp = BEG_PROC_ADDR, t = -NR_TASKS; rp < END_PROC_ADDR; ++rp, ++t) {
+
+    #ifdef DISO
+    rp->priority = DISO_PRILOWEST; /* Prioridad por defecto */
+    rp->penalty = 0;
+    #endif /* DISO */
+
+

```

---

```
rp->p_flags = P_SLOT_FREE;
rp->p_nr = t; /* proc number from ptr */
(pproc_addr + NR_TASKS)[t] = rp; /* proc ptr from number */
```

---

### Diferencias - *main.c*

## 1.4. Funcionalidad de la tecla F1

Para que al oprimir la tecla F1 se visualicen, junto a otros datos de los procesos, los campos *priority* y *penalty* hemos hecho las siguientes modificaciones en */usr/src/kernel/dmp.c*:

---

```
--- kernel/dmp.c.orig 2006-10-27 21:24:23.000000000 -0300
+++ kernel/dmp.c      2006-11-07 13:28:27.000000000 -0300
@@ -8,6 +8,7 @@
```

```
FORWARD _PROTOTYPE(char *proc_name, (int proc_nr));
```

```
+
+/*=====*
+*                               *
+*                               *
+*=====*/
```

```
@@ -21,8 +22,11 @@
```

```
int n = 0;
phys_clicks text, data, size;
int proc_nr;
```

```
-
```

```
+ #ifdef DISO
```

```
+ printf("\n--pid --pc- ---sp- PRTY -user --sys-- -text- -data- -size- -recv- command\n");
```

```
+ #else
```

```
printf("\n--pid --pc- ---sp- flag -user --sys-- -text- -data- -size- -recv- command\n");
```

```
+ #endif
```

```
for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
```

```
proc_nr = proc_number(rp);
```

```
@@ -32,14 +36,23 @@
```

```
data = rp->p_map[D].mem_phys;
```

```
size = rp->p_map[T].mem_len
```

```
+ ((rp->p_map[S].mem_phys + rp->p_map[S].mem_len) - data);
```

```
+ #ifdef DISO
```

```
+ printf(" %5d %5lx %6lx %2d %7U %7U %5uK %5uK %5uK ",
```

```
+ #else
```

```
printf(" %5d %5lx %6lx %2x %7U %7U %5uK %5uK %5uK ",
```

```
+ #endif
```

```

        proc_nr < 0 ? proc_nr : rp->p_pid,
        (unsigned long) rp->p_reg.pc,
        (unsigned long) rp->p_reg.sp,
+       #ifdef DISO
+       rp->priority,
+       #else
        rp->p_flags,
+       #endif
        rp->user_time, rp->sys_time,
        click_to_round_k(text), click_to_round_k(data),
        click_to_round_k(size));
+
        if (rp->p_flags & RECEIVING) {
            printf(" %-7.7s", proc_name(rp->p_getfrom));
        } else
@@ -49,6 +62,7 @@
        if (rp->p_flags == 0) {
            printf(" ");
        }
+
        printf(" %s\n", rp->p_name);
    }
    if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");
@@ -132,7 +146,7 @@
    vir_clicks base, limit;

    printf(
-       "\nproc pid   pc   sp  splow flag user   sys  recv  command\n");
+       "\nproc pid   pc   sp  splow PRTY user sys  recv  command\n");

    for (rp = oldrp; rp < END_PROC_ADDR; rp++) {
        if (rp->p_flags & P_SLOT_FREE) continue;
@@ -147,6 +161,13 @@
        (unsigned long) rp->p_splow,
        rp->p_flags,
        rp->user_time, rp->sys_time);
+
+       #ifdef DISO
+
+       printf(" %-7.7s", rp->priority);
+
+       #else
+
        if (rp->p_flags & RECEIVING) {
            printf(" %-7.7s", proc_name(rp->p_getfrom));
        } else
@@ -156,6 +177,9 @@
        if (rp->p_flags == 0) {
            printf(" ");
        }

```

---

```

    }
+
+   #endif /* DISO */
+
+   printf(" %s\n", rp->p_name);
    }
    if (rp == END_PROC_ADDR) rp = BEG_PROC_ADDR; else printf("--more--\r");

```

---

### Diferencias - *dmp.c*

## 1.5. Llamada al sistema

Dado que en el Trabajo Práctico N° 1 describimos detalladamente cómo implementar una system call, a continuación solo se dará una breve descripción de los pasos que llevamos a cabo para crear la llamada al sistema.

### 1.5.1. Comunicación *proceso de usuario - Memory Manager*

Creamos el archivo */usr/src/lib/syscall/chgprio.s* y modificamos el archivo *Makefile* correspondiente:

---

```

.sect .text
.extern __chgprio
.define _chgprio
.align 2

_chgprio:
    jmp    __chgprio

```

---

### Archivo - *chgprio.s*

Generamos el archivo */usr/src/lib/posix/\_chgprio.c* y también modificamos el archivo *Makefile* correspondiente:

---

```

#include <lib.h>
#define chgprio _chgprio
#include <unistd.h>

PUBLIC int chgprio(pid, priority)
int pid;
int priority;

```



---

```

{
    message m;

    /* Seteamos los parametros del mensaje */
    m.m1_i1 = pid;
    m.m1_i2 = priority;

    if (_syscall(MM, CHGPRI, &m) < 0)
        return -1;
    else
        return (m.m1_i3);
}

```

---

### Archivo - `__chgprio.c`

Modificamos los archivos `/usr/include/unistd.h` y `/usr/include/minix/callnr.h` según se muestra a continuación:

---

```

--- include/unistd.h.orig  2006-09-28 18:31:24.000000000 -0300
+++ include/unistd.h 2006-11-02 18:11:00.000000000 -0300
@@ -90,6 +90,7 @@
 _PROTOTYPE( int access, (const char *_path, int _amode) );
 _PROTOTYPE( unsigned int alarm, (unsigned int _seconds) );
 _PROTOTYPE( int chdir, (const char *_path) );
+_PROTOTYPE( int chgprio, (int _pid, int _priority) );
 _PROTOTYPE( int chown, (const char *_path, Uid_t _owner, Gid_t _group) );
 _PROTOTYPE( int close, (int _fd) );
 _PROTOTYPE( char *ctermid, (char *_s) );

```

---

### Diferencias - `unistd.h`

---

```

--- include/minix/callnr.h.orig 2006-09-28 18:36:21.000000000 -0300
+++ include/minix/callnr.h 2006-11-07 00:00:26.000000000 -0300
@@ -56,6 +56,9 @@
 #define REVIVE 67 /* to FS: revive a sleeping process */
 #define TASK_REPLY 68 /* to FS: reply code from tty task */

+#ifdef DISO
+#define CHGPRI 70
+#endif
#define SIGACTION 71
#define SIGSUSPEND 72
#define SIGPENDING 73

```

---

### Diferencias - *callnr.h*

#### 1.5.2. Memory Manager

Editamos los archivos *proto.h*, *table.c* y agregamos la función *do\_chgprio()* en *trace.c*.

---

```

--- mm/proto.h.orig 2006-10-28 17:36:41.000000000 -0300
+++ mm/proto.h 2006-11-02 19:19:00.000000000 -0300
@@ -68,3 +68,8 @@
 _PROTOTYPE( int no_sys, (void) );
 _PROTOTYPE( void panic, (char *format, int num) );
 _PROTOTYPE( void tell_fs, (int what, int p1, int p2, int p3) );
+
+#ifdef DISO
+/* DISO */
+_PROTOTYPE( int do_chgprio, (void) );
+#endif

```

---

### Diferencias - */usr/src/mm/proto.h*

---

```

--- mm/table.c.orig 2006-10-28 17:37:20.000000000 -0300
+++ mm/table.c 2006-11-02 19:19:00.000000000 -0300
@@ -85,7 +85,11 @@
     no_sys,      /* 67 = REVIVE */
     no_sys,      /* 68 = TASK_REPLY */
     no_sys,      /* 69 = unused */
+
+    #ifdef DISO
+    do_chgprio,   /* 70 = chgprio */
+    #else
+    no_sys,       /* 70 = unused */
+    #endif
+
+    do_sigaction, /* 71 = sigaction */
+    do_sigsuspend, /* 72 = sigsuspend */
+    do_sigpending, /* 73 = sigpending */

```

---

### Diferencias - */usr/src/mm/table.c*

```

--- mm/trace.c.orig 2006-10-28 17:37:42.000000000 -0300
+++ mm/trace.c 2006-11-02 19:19:00.000000000 -0300
@@ -107,3 +107,20 @@
     }
     return;
 }
+
+#ifdef DISO
+/*=====
+ *                                do_chgprio                                *
+ *=====*/
+PUBLIC int do_chgprio()
+{
+    int respuesta;
+    int r;
+
+    r = sys_chgprio(mm_in.m1_i1, mm_in.m1_i2, &respuesta);
+
+    mm_out.m1_i3 = respuesta;
+
+    return r;
+}
+#endif

```

---

Diferencias - */usr/src/mm/trace.c*

### 1.5.3. Comunicación *Memory manager* - *SYSTASK*

Creamos el archivo */usr/src/lib/syslib/sys\_chgprio.c* y modificamos el *Makefile* correspondiente.

---

```

#include "syslib.h"

PUBLIC int sys_chgprio(pid, priority, pidAnterior)
int pid;           /* PID del proceso a cambiar la prioridad */
int priority;      /* La prioridad... */
int* pidAnterior;  /* PID anterior del proceso */
{
    message m;
    int r;

    m.m1_i1 = pid;
    m.m1_i2 = priority;

    r = _taskcall(SYSTASK, SYS_CHGPRI, &m);

```

```

*pidAnterior = m.m1_i3;

return r;
}

```

---

### Archivo - *sys\_chgprio.c*

Modificamos el fichero */usr/include/minix/syslib.h*.

---

```

--- include/minix/syslib.h.orig 2006-09-28 18:37:54.000000000 -0300
+++ include/minix/syslib.h 2006-11-07 00:00:27.000000000 -0300
@@ -18,6 +18,7 @@
 _PROTOTYPE( int send, (int _dest, message *_m_ptr) );

 _PROTOTYPE( int sys_abort, (int _how, ...) );
+_PROTOTYPE( int sys_chgprio, (int _pid, int _priority, int* respuesta) );
 _PROTOTYPE( int sys_adjmap, (int _proc, struct mem_map *_ptr,
                             vir_clicks _data_clicks, vir_clicks _sp) );
 _PROTOTYPE( int sys_copy, (int _src_proc, int _src_seg, phys_bytes _src_vir,

```

---

### Diferencias - *syslib.h*

#### 1.5.4. SYSTASK

Modificamos el archivo */usr/include/minix/com.h* e incluimos en el archivo */usr/src/kernel/system.c* la función *do\_chgprio* encargada de modificar la prioridad del proceso. También en este archivo se pueden ver los cambios hechos al realizarse un *fork* de un proceso. Estos cambios pueden verse a continuación (algunos cambios son intencionales para que se pueda visualizar el nombre de la función modificada):

---

```

--- include/minix/com.h.orig 2006-09-28 18:36:59.000000000 -0300
+++ include/minix/com.h 2006-11-07 00:00:26.000000000 -0300
@@ -138,6 +138,9 @@
 #   define SYS_SIGRETURN 18 /* fcn code for sys_sigreturn(&sigmsg) */
 #   define SYS_ENDSIG 19 /* fcn code for sys_endsig(procno) */
 #   define SYS_GETMAP 20 /* fcn code for sys_getmap(procno, map_ptr) */
+#ifdef DISO
+#   define SYS_CHGPRI 21 /* fcn code for sys_chgprio(pid, priority) */
+#endif

```

## Diferencias - *com.h*

[illegible]

```

    rpc->child_utime = 0;
    rpc->child_stime = 0;

+ #ifdef DISO
+ rpc->priority = DISO_PRILOWEST;
+ rpc->penalty = 0;
+ #endif
+
+ #if (SHADOWING == 1)
+     rpc->p_nflips = 0;
+     mkshadow(rpp, (phys_clicks)m_ptr->m1_p1); /* run child first */
@@ -1156,3 +1167,79 @@
+ }
+ }
+ #endif /* (CHIP == INTEL) */
+
+ #ifdef DISO
+ /*=====
+ *                                     do_chgprio                                     *
+ *=====*/
+ PRIVATE int do_chgprio(m_ptr)
+ register message *m_ptr; /* pointer to request message */
+ {
+     int i;
+     struct proc* aux;
+     struct proc* procesoBuscado; /* Proceso al cual vamos a cambiarle la
+     * prioridad */
+     struct proc* procesoAnterior; /* Proceso cuyo p_nextready apunta al proceso
+     * al cual vamos a cambiarle la prioridad */
+
+     /* m1_i2 es la prioridad. Chequemos que esté en el intervalo correcto */
+     if ( !(m_ptr->m1_i2 >= DISO_PRIHIGHEST &&
+         m_ptr->m1_i2 <= DISO_PRILOWEST)) {
+
+         return(E_CHGPRIO);
+     }
+
+     /* m1_i1 es el pid. Verificamos que sea mayor a cero */
+     if ( !(m_ptr->m1_i1 > 0) ) {
+         return(E_CHGPRIO);
+     }
+
+     /* Verificamos que el pid exista */
+     i = 0;
+     while (i < NR_TASKS + NR_PROCS) {
+
+         if (proc_addr(i)->p_pid == m_ptr->m1_i1) {
+
+             procesoBuscado = proc_addr(i);

```

```
+
+         break;
+     }
+
+     i++;
+ }
+
+ if (i >= NR_TASKS + NR_PROCS) {
+     return(E_CHGPRIO);
+ }
+
+ /* La respuesta de la llamada al sistema es la prioridad anterior */
+ m_ptr->m1_i3 = procesoBuscado->priority;
+
+ /* Si el proceso tiene la misma prioridad que la que se va a setear
+  * no continuamos */
+ if (procesoBuscado->priority == m_ptr->m1_i2)
+     return(OK);
+
+ /* Seteamos la prioridad */
+ procesoBuscado->priority = m_ptr->m1_i2;
+ procesoBuscado->penalty = 0;
+
+ /* Reubicamos el proceso en la lista de procesos listos */
+ aux = rdy_head[USER_Q];
+ if (aux != NIL_PROC) {
+     while (aux->p_nextready != NIL_PROC &&
+           aux->p_nextready != procesoBuscado) {
+
+         aux = aux->p_nextready;
+     }
+
+     if (aux->p_nextready != NIL_PROC) {
+         aux->p_nextready = aux->p_nextready->p_nextready;
+         ubicarProceso(procesoBuscado);
+     }
+ }
+
+ return(OK);
+}
+
+#endif
```

## 1.6. Comando para modificar la prioridad de un proceso

El código del comando que permite modificar la prioridad de despacho de un proceso puede verse a continuación:

---

```
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <time.h>

_PROTOTYPE (int main, (int argc, char *argv []));

int main(argc, argv)
int argc;
char* argv[];
{
    int pid;
    int prioridad;
    int respuesta;

    time_t a;

    if (argc < 3) {
        printf("Pasar por lo menos dos parametros numericos.\n");
        exit(1);
    }

    pid = atoi(argv[1]);
    prioridad = atoi(argv[2]);

    respuesta = chgprio(pid, prioridad);

    if (respuesta < 0) {
        printf("PID o prioridad incorrectos\n");
    }

    return respuesta;
}
```

---

Archivo - *chgprio.c*



## 1.7. Algoritmo de planificación

Según el enunciado del Trabajo Práctico solamente debíamos modificar la función *ready()* para que gestione la cola de usuarios basándose en la prioridad de cada uno de los procesos. Sin embargo, la función *sched()* también modifica la distribución de los elementos en dicha cola, por lo que decidimos modificar ambas funciones para que *USER\_Q* se gestione bajo una única política.

### 1.7.1. Campos *priority* y *penalty*

El campo *priority* cumple la misma función que el sugerido en el enunciado del Trabajo Práctico: almacenar la prioridad del proceso. El campo *penalty* decidimos agregarlo para evitar la inanición de procesos de muy baja prioridad. Cada vez que un proceso termina de ejecutarse debido al vencimiento de su *timeslice*, éste campo se incrementa con el valor almacenado en *priority*. La ubicación de un proceso dentro de *USER\_Q* va a estar dada por la suma de ambos campos (*priority* + *penalty*).

### 1.7.2. Cola de procesos de usuario

Según el esquema presentado en la sección anterior queda en evidencia que la cola de procesos de usuario *USER\_Q* va a estar ahora constituida por subcolas, donde los elementos de cada una de ellas van a ser procesos de una misma prioridad.

A continuación se muestran las modificaciones hechas en las funciones *ready()* y *sched()* para gestionar la cola. Cabe aclarar que hemos definido una nueva función *ubicarProceso()* cuyo objetivo es reubicar, cuando sea necesario, un proceso dentro de la cola de procesos de usuarios. El código de todas estas funciones está suficientemente comentado como para poder comprenderlo y seguirlo.

---

```

--- kernel/proc.c.orig 2006-10-27 21:24:23.000000000 -0300
+++ kernel/proc.c      2006-11-07 11:50:32.000000000 -0300
@@ -25,6 +25,7 @@
         message *m_ptr) );
     FORWARD _PROTOTYPE( int mini_rec, (struct proc *caller_ptr, int src,
         message *m_ptr) );
+_PROTOTYPE( int ubicarProceso, (struct proc*) );
FORWARD _PROTOTYPE( void ready, (struct proc *rp) );
FORWARD _PROTOTYPE( void sched, (void) );
FORWARD _PROTOTYPE( void unready, (struct proc *rp) );
@@ -328,6 +329,65 @@

```

```

}

/*=====
+ *                               ubicarProceso                               *
+ *=====*/
+int ubicarProceso(struct proc* unProceso) {
+/* Ubica el proceso unProceso en la cola de procesos listos. Se encarga de
+ * actualizar todos los punteros, incluyendo rdy_head[USER_Q] y
+ * rdy_tail[USER_Q]. Sin embargo es el llamador el que debe actualizar los
+ * punteros antes de mover este proceso (en la funcion ready no hay problema,
+ * porque el proceso se inserta, pero en sched el proceso se mueve en la lista
+ * de procesos listos. Por lo tanto esta debe quitarlo de la lista antes de
+ * llamar a esta funcion).
+ *
+ * Valores de respuesta:
+ *   -1 : Cola vacia
+ *   0 : El proceso se ubico en la cabeza
+ *   1 : El proceso se ubico en el interior
+ *   2 : El proceso se ubico en la cola
+ */
+
+ struct proc* aux;
+
+ /* Si la cola esta vacia, retornamos NULL */
+ if (rdy_head[USER_Q] == NIL_PROC)
+     return -1;
+
+ /* Si unProceso va a la cabeza... */
+ if (rdy_head[USER_Q]->priority + rdy_head[USER_Q]->penalty >
+     unProceso->priority + unProceso->penalty) {
+
+     unProceso->p_nextready = rdy_head[USER_Q];
+     rdy_head[USER_Q] = unProceso;
+     return 0;
+ }
+
+ /* En caso contrario buscamos al proceso anterior a unProceso */
+ aux = rdy_head[USER_Q];
+ while ((aux->p_nextready != NIL_PROC) &&
+     (unProceso->priority + unProceso->penalty >
+      aux->p_nextready->priority + aux->p_nextready->penalty)) {
+
+     aux = aux->p_nextready;
+ }
+
+ /* Lo ingresamos en el final */
+ if (aux->p_nextready == NIL_PROC) {
+     aux->p_nextready = unProceso;
+     unProceso->p_nextready = NIL_PROC;

```

```

+         rdy_tail[USER_Q] = unProceso;
+         return 2;
+     }
+
+     /* Se ubica en el interior */
+     else {
+         unProceso->p_nextready = aux->p_nextready;
+         aux->p_nextready = unProceso;
+         return 1;
+     }
+ }
+ }
+
+ /*=====
+ *                                     ready                                     *
+ *=====*/
+ PRIVATE void ready(rp)
@@ -372,6 +432,25 @@
+     return;
+ }
+ #endif
+
+ #ifdef DISO
+ /* Respetando la politica original de encolamiento de procesos en Minix,
+  * a los nuevos procesos se los agrega en la cabeza de la subcola de su
+  * misma prioridad. Bajo este enfoque, la cola de procesos de usuario
+  * esta constituida por subcolas cuyo componentes son procesos de igual
+  * prioridad */
+
+ /* Si la cola esta vacia, no importa la prioridad, lo insertamos
+  * al principio */
+ if (rdy_head[USER_Q] == NIL_PROC) {
+     rp->p_nextready = NIL_PROC;
+     rdy_head[USER_Q] = rdy_tail[USER_Q] = rp;
+ }
+ else
+     ubicarProceso(rp);
+
+ #else
+ if (rdy_head[USER_Q] == NIL_PROC)
+     rdy_tail[USER_Q] = rp;
+     rp->p_nextready = rdy_head[USER_Q];
@@ -384,6 +463,8 @@
+     rdy_tail[USER_Q] = rp;
+     rp->p_nextready = NIL_PROC;
+ */
+
+ #endif /* DISO */
+ }

```

```

/*=====*
@@ -457,6 +538,7 @@
    if (*qtail == rp) *qtail = xp;
}

+
/*=====*
*                                     sched                                     *
*=====*/
@@ -467,13 +549,75 @@
* possibly promoting another user to head of the queue.
*/

+ #ifdef DISO
+ /* El proceso que termino de ejecutarse */
+ struct proc* procesoActual = rdy_head[USER_Q];
+ #endif
+
+ if (rdy_head[USER_Q] == NIL_PROC) return;

+ #ifdef DISO
+ /* Si hay un solo proceso en la lista de procesos listos, entonces no hay
+  * nada que hacer mas que llamar a pick_proc. */
+ if (rdy_head[USER_Q] -> p_nextready == NIL_PROC) {
+     pick_proc();
+     return;
+ }
+
+ /* Aumentamos la penalidad del proceso que se estaba ejecutando. El "1" es
+  * para ubicar al proceso que se termino de ejecutar en el final de su
+  * sublista de prioridad. Simplemente es para eso, luego lo restamos. */
+ procesoActual -> penalty += procesoActual -> priority + 1;
+
+ /* Si el proceso que le sigue ahora tiene mayor prioridad, entonces
+  * empezamos un proceso de reorganización de la lista de procesos listos. En
+  * caso contrario no tenemos que hacer nada */
+ if (rdy_head[USER_Q] -> p_nextready -> penalty +
+     rdy_head[USER_Q] -> p_nextready -> priority <
+     procesoActual -> priority + procesoActual -> penalty) {
+
+     /* La nueva cabeza de la cola va a ser el proximo proceso listo */
+     rdy_head[USER_Q] = rdy_head[USER_Q] -> p_nextready;
+
+     /* Reubicamos el proceso el cual termino de ejecutarse con
+      * ubicarProceso. Si su nueva ubicacion es el fin de la cola de procesos
+      * listos (nos damos cuenta de esto si ubicarProceso retorna 2),
+      * entonces verificamos que el proceso que le sigue a procesoActual (que
+      * vendria a ser ahora la cabeza), tenga una prioridad menor que

```

```

+      * procesoActual, o que la penalidad de dicho proceso sea mayor a 16.
+      * Cualquiera de las dos son condiciones para setear todos los atributos
+      * penalty de los procesos listos a cero. */
+      if (ubicarProceso(procesoActual) == 2) {
+          /* Recordar que rdy_head[USER_Q] en este punto, vendría a ser
+          * el proceso listo siguiente antes de que procesoActual
+          * actualice su posicion en la lista. */
+          if (rdy_head[USER_Q]->priority < procesoActual->priority ||
+              procesoActual->penalty > 16) {
+
+              struct proc* aux = rdy_head[USER_Q];
+              while (aux != NIL_PROC) {
+                  aux->penalty = 0;
+                  aux = aux->p_nextready;
+              }
+          }
+          else {
+              procesoActual->penalty -= 1;
+          }
+      }
+  }
+  else
+      procesoActual->penalty -= 1;
+  #else
+
+  /* One or more user processes queued. */
+  rdy_tail[USER_Q]->p_nextready = rdy_head[USER_Q];
+  rdy_tail[USER_Q] = rdy_head[USER_Q];
+  rdy_head[USER_Q] = rdy_head[USER_Q]->p_nextready;
+  rdy_tail[USER_Q]->p_nextready = NIL_PROC;
+
+  #endif /* DISO */
+
+  pick_proc();
+}

```

---

### Diferencias - *proc.c*

## 1.8. Test de funcionamiento

Para verificar el impacto en la asignación de prioridades creamos el programa *mipid* tal como lo sugería el enunciado del Trabajo Práctico. Además, mediante *solocpu.c* y la impresión de los elementos de la cola de procesos

de usuario, verificamos que el algoritmo de planificación funcionara correctamente.

---

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

void main()
{
    while(1) {
        printf("[ %d]\n", getpid());
    }
}
```

---

**Archivo - *mipid.c***

---

```
void main() {
    while (1);
}
```

---

**Archivo - *solocpu.c***

## 1.9. Organización del disquette

En el disquette se encuentran los archivos de código fuente del kernel de Minix en el directorio */usr/src*, los headers en */usr/include* y */usr/include/minix*, y los programas de usuario en el directorio raíz, así como también la imagen del kernel (archivo */image*).

## 2. Sección de descargos

No hay descargos.