Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

INFORME TÉCNICO

Implementación de mecanismos de planificación tolerante a fallas en un sistema operativo open-source

Gustavo Fossati LU 591/91 gustavof@coremation.com
Jose Luis Dominguez LU 626/91 Jldoming@dc.uba.ar

Director: Dr. Gabriel Wainer

Pabellón 1 - Planta Baja - Ciudad Universitaria
(1428) Buenos Aires - Argentina

http://www.dc.uba.ar

Mayo 2003

Tabla de Contenidos

1. INTRODUCCIÓN 1
2. INSTALACIÓN DE RT-MINIX 3
3. USO DE LA API 5
   3.1. COMO INSTALAR LOS EJEMPLOS 5
   3.2. COMO CORRER LOS EJEMPLOS 5
   3.3. UN EJEMPLO SENCILLO 7
   3.4. ALGORITMOS DE PLANIFICACIÓN 8
   3.5. MANEJO DE METAS 12
   3.6. COMPUTACIÓN IMPRECISA 16
   3.7. SINCRONIZACIÓN DE PROCESOS 18
   3.8. ESTADÍSTICAS DEL SISTEMA 23
   3.9. TOLERANCIA DE FALLAS TRANSITORIAS 27
   3.10. TOLERANCIA DE FALLAS DE SOFTWARE 30
   3.11. MECANISMOS DE DETECCIÓN DE SOBRECARGA Y MULTICOLAS 34
   3.12. LOG DEL NÚCLEO 37
4. REFERENCIA DE LA API 41
   4.1. FUNCIONES 41
   4.2. ESTRUCTURAS 52
   4.3. SEÑALES 64
   4.4. ERRORES 65
   4.5. COMANDOS 66
5. DETALLES DE IMPLEMENTACIÓN 77
   5.1. PROCESOS DE RT-MINIX 77
   5.2. PLANIFICADOR 86
   5.3. SINCRONIZACIÓN DE PROCESOS 107
   5.4. TOLERANCIA A FALLAS 112
   5.5. PROCESO SUPERVISOR 118
   5.6. INYECCIÓN DE FALLAS 124
   5.7. BIBLIOTECA DE TIEMPO REAL (BTR) 125
GLOSARIO 133

1. Introducción

Este documento incluye ejemplos de uso, un manual de referencia de la nueva API y todos los detalles de implementación de las modificaciones realizadas a MINIX en la construcción de RT-MINIX.

Está orientado a dos tipos de lectores: desarrolladores que quieran hacer uso de la API para construir aplicaciones de tiempo real y desarrolladores que quieran entender y eventualmente extender RT-MINIX.

Existen versiones previas de RT-MINIX con funcionalidades básicas, pero estas iniciativas tienen limitaciones de diseño que son prohibitivas para un sistema de tiempo real y dificultan la incorporación de mecanismos de tolerancia a fallas.

La versión de RT-MINIX que se presenta en este trabajo utilizó las versiones anteriores solo como referencia. La profundidad de los cambios propuestos obligó a rediseñar y replantear el sistema. De las versiones previas solo conservó el manejo de granularidad del reloj del sistema, el controlador de dispositivos para acceder a los conversores A/D provistos por el *puerto de juegos* y los algoritmos de sensado robusto.

El planificador se reformuló por completo incorporando mecanismos de herencia, computación imprecisa y remoción por prioridad. Se implementaron mecanismos para tolerancia de fallas transitorias y de software. Se dotó al planificador de un mecanismo de multicolas para manejar las sobrecargas. Se incluyó soporte para sincronización entre procesos mediante semáforos con techo de prioridades. Se construyó una interfaz de desarrollo (API) muy completa pero a la vez sencilla que permite acceder a todas las características de tiempo real.

En el capítulo 2 se presentan las instrucciones necesarias para la instalación de RT-MINIX. En el capítulo 3 una guía de uso de la API de tiempo real con numerosos ejemplos. En el capítulo 4 una referencia de la API de tiempo real y del *Log del Núcleo*, incluyendo llamadas al sistema, estructuras, errores y comandos.

En el capítulo 5 las herramientas incorporadas para apoyo al desarrollo y análisis de desempeño del sistema: *Log del Núcleo*, *Monitor de Tareas*, *Estadísticas* e inyección de fallas. Por último, en el capítulo 6 se detallan y analizan las modificaciones realizadas al *Núcleo* de MINIX.

2. Instalación de RT-MINIX

Durante la instalación se crearán y reemplazarán archivos y directorios. Luego, el script de instalación compilará todas las bibliotecas, el *Núcleo* y los comandos.

Para realizar la instalación es necesario seguir los pasos que se enumeran a continuación.

Paso 1. copiar los siguientes archivos en un disquete:

```
install.txt (rtminix\install\install.txt)
rtminix.tar (rtminix\install\rtminix.tar)
```

Paso 2. Loguearse como 'root' en la máquina MINIX.

Paso 3. Insertar el disquete y ejecutar los siguientes comandos:

```
cd /usr/tmp
dosread -a /dev/fd0 install.txt > install
sh install
```

Paso 4. Cuando la maquina haya terminado de reiniciarse, desde el Monitor, reiniciar nuevamente con el siguiente comando:

```
exit
```

Paso 5. Presionar 'r' para arrancar la versión de MINIX construída.

Los archivos originales de MINIX se almacenan como backup en el directorio */usr/backup*. Para reestablecer los archivos originales de MINIX y su imagen se debe ejecutar el siguiente comando: *./usr/src/tools/restore*

Para reconstruir el sistema con otras características se debe ejecutar el siguiente comando: *./usr/src/tools/rebuild*

3. Uso de la API

El objetivo de esta sección es brindarle al usuario de la API una ayuda para comprender rápidamente los conceptos y construir las primeras aplicaciones de tiempo real. La ayuda se basa en numerosos ejemplos que reflejan gran parte de las características de RT-MINIX.

3.1. Como instalar los ejemplos

Para instalar los ejemplos es necesario seguir los siguientes pasos:

Paso 1. Copiar los siguientes archivos en un disquete:

```
install.txt
samples.tar
```

Paso 2. Loguearse como 'root' en la máquina RT-MINIX.

Paso 3. Insertar el disquete y ejecutar los siguientes comandos:

```
> cd /usr/tmp
> dosread -a /dev/fd0 install.txt > install
> sh install
```

Durante la instalación se crea el directorio */usr/src/test/*, se copian los archivos fuente y se compilan los programas.

3.2. Como correr los ejemplos

Los ejemplos pueden ejecutarse desde cualquier consola, pero siempre desde el directorio */usr/src/test/samples*.

La mayoría de los ejemplos muestran resultados por pantalla, pero lo interesante es ver las líneas de tiempo y uso del procesador mediante el *Monitor de Tareas*. Como el *Monitor* funciona en la consola 1, es recomendable correr los ejemplos de la siguiente forma:

1. Activar el *Monitor* en la consola 1 (se activa y desactiva con la tecla F10).

2. Cambiar a la consola 2 para lanzar un ejemplo.
3. Cambiar a la consola 1 para observar la corrida.
4. Volver a cambiar a la consola 2 para ver los resultados.

Los resultados se muestran en la consola que se ejecuta el ejemplo. Si se ejecuta en la consola 1 con el *Monitor* activo, los resultados se mezclarán con la gráfica del mismo.

Para facilitar las pruebas, en este documento se incluyen los gráficos de tiempo y uso de procesador de todos los ejemplos. En éstos diagramas la línea de tiempo esta representada en segundos (cada cuadradito es un segundo) y el significado de los símbolos es el siguiente:

Símbolo	Descripción
	Tiempo ocioso
	Tiempo de ejecución de la parte mandatoria de la tarea
	Tiempo de ejecución de la parte opcional de la tarea
	Tiempo de ejecución del Backup
	Falla o meta perdida
	Tiempo de listo y límite de períodos
	Meta
	El semáforo S fue liberado
	El semáforo S fue tomado
	Se intentó tomar el semáforo S

3.3. Un ejemplo sencillo

Este sencillo ejemplo muestra como lanzar un proceso periódico que imprima tres veces y cada cuatro segundos la leyenda "Hello World !!!".

El código fuente es el siguiente:

```
01  #include <stdio.h>
02  #include <minix/rt.h>
03  #include <sys/wait.h>
04
05  _PROTOTYPE( void main, (int argc, char **argv) );
06  _PROTOTYPE( void rt_function, (void) );
07
08  void rt_function(void)
09  {
10       printf(" Hello World !!!\n");
11  }
12
13  void main(argc, argv)
14  int argc;
15  char **argv;
16  {
17       int status;
18       pid_t A_pid;
19       rt_cfg_t A;
20
21       rt_init_cfg(&A);
22
23       A.exec_type          = EXEC_TYPE_PERIODIC;
24       A.period             = 4000;
25       A.max_periods        = 3;
26       A.mandatory_function = rt_function;
27
28       A_pid = rt_fork("Task A", &A);
29
30       if (A_pid>=1)
31            waitpid(A_pid, &status, 0);
32
```

```
33      }
```

Código fuente 1 – sample1.c

Los procesos de tiempo real se lanzan mediante la llamada al sistema *rt_fork* (línea 28), pasando por parámetro el nombre (en éste caso "*Task A*") y una estructura *rt_cfg_t* con las características de tiempo real del proceso. La estructura *rt_cfg_t* debe inicializarse con la función *rt_init_cfg* (línea 21) y luego configurarse. En este caso se trata de un proceso periódico (línea 23), con una duración de período de 4000 milisegundos (línea 24) y una cantidad máxima de tres períodos (línea 25). La función que el *Núcleo* debe llamar en cada período es *rt_function* (línea 26) y esta definida en las líneas 08-11.

Una vez creado el proceso de tiempo real mediante la llamada al sistema *rt_fork*, se espera que finalice (en forma sincrónica) mediante la llamada al sistema *waitpid*.

La ejecución muestra por pantalla lo siguiente:

```
# Hello world !!!
# Hello world !!!
# Hello world !!!
```

El diagrama de tiempos es el siguiente:

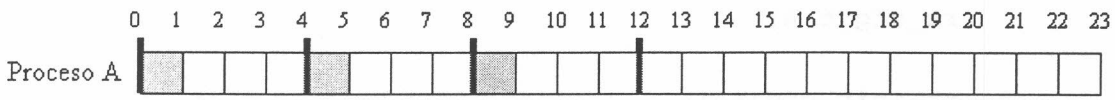

Figura 1 - Diagrama de tiempos de sample1.c

Por simplicidad, los diagramas de tiempo fueron realizados con una granularidad de 1 segundo.

3.4. Algoritmos de planificación

El planificador elige a los procesos de tiempo real en base a los algoritmos de planificación vigentes. En todo momento hay dos algoritmos de planificación activos, uno para procesos mandatorios y otro para opcionales.

Si para el algoritmo de planificación vigente dos procesos tienen la misma prioridad, la elección se basa en el peso, el campo *weight* de la estructura *rt_cfg_t*.

Los algoritmos de planificación se configuran con las llamadas al sistema *rt_set_man_scheduler* y *rt_set_opt_scheduler*. Los algoritmos por defecto son *SCHED_ALG_BWF* y *SCHED_ALG_FIFO* para procesos mandatorios y opcionales respectivamente. En este ejemplo se modifica el algoritmo de planificación para procesos mandatorios.

El código fuente es el siguiente:

```
01   #include <stdio.h>
02   #include <minix/rt.h>
03   #include <sys/wait.h>
04   #include <unistd.h>
05
06   _PROTOTYPE( void main,     (int argc, char **argv) );
07   _PROTOTYPE( void rt_function, (void) );
08
09   void rt_function(void)
10   {
11        proc_info_t proc_info;
12        rt_get_proc_info(getpid(), &proc_info);
13        rt_waste_time(proc_info.rt_cfg.mandatory_time, 0);
14        printf(" Executing: %s\n", proc_info.name);
15   }
16
17   void main(argc, argv)
18   int argc;
19   char **argv;
20   {
21        int status;
22        pid_t A_pid, B_pid, C_pid;
23        rt_cfg_t A, B, C;
24
```

```
25        rt_set_man_scheduler(SCHED_ALG_SJF);
26
27        rt_init_cfg(&A);
28        rt_init_cfg(&B);
29        rt_init_cfg(&C);
30
31        A.exec_type                = EXEC_TYPE_APERIODIC;
32        A.ready_time               = 1000;
33        A.deadline                 = 8000;
34        A.mandatory_function       = rt_function;
35        A.mandatory_time           = 3000;
36
37        B.exec_type                = EXEC_TYPE_APERIODIC;
38        B.ready_time               = 1000;
39        B.deadline                 = 8000;
40        B.mandatory_function       = rt_function;
41        B.mandatory_time           = 1000;
42
43        C.exec_type                = EXEC_TYPE_APERIODIC;
44        C.ready_time               = 1000;
45        C.deadline                 = 8000;
46        C.mandatory_function       = rt_function;
47        C.mandatory_time           = 2000;
48
49        A_pid = rt_fork("Task A", &A);
50        B_pid = rt_fork("Task B", &B);
51        C_pid = rt_fork("Task C", &C);
52
53        if (A_pid>=1)
54             waitpid(A_pid, &status, 0);
55        if (B_pid>=1)
56             waitpid(B_pid, &status, 0);
57        if (C_pid>=1)
58             waitpid(C_pid, &status, 0);
```

```
59    }
```

Código fuente 2 - sample2.c

En el ejemplo se configura el algoritmo *trabajo más corto primero* (*SCHED_ALG_SJF*) para procesos mandatorios (línea 25) y se disparan tres procesos aperiódicos. El algoritmo SJF se basa en los peores casos de ejecución definidos en las líneas 35, 41 y 47. Para cada proceso, también se define un tiempo de listo (líneas 32, 38 y 44) y una meta (líneas 33, 39 y 45), obligatoria para éste tipo de procesos.

Los tres procesos usan la misma función *rt_function* definida en las líneas 09-15. Esta función deja pasar un tiempo equivalente al peor caso de ejecución y luego imprime el nombre del proceso. Mediante una estructura *proc_info_t* y la llamada al sistema *rt_get_proc_info* (línea 12) se puede obtener el tiempo del peor caso de ejecución y el nombre del proceso. Con la llamada al sistema *rt_waste_time* se emula la ejecución del proceso (línea 13). Finalmente, en la línea 14 se imprime el nombre en pantalla el nombre del proceso.

La ejecución muestra por pantalla lo siguiente:

```
# Executing Task B
# Executing Task C
# Executing Task A
```

El diagrama de tiempos es el siguiente:

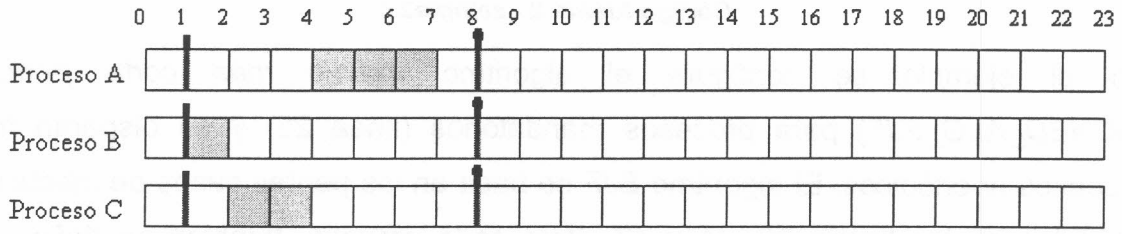

Figura 2 - Diagrama de tiempos de sample2.c

3.5. Manejo de metas

En la creación de cada proceso puede configurarse el tratamiento que el *Núcleo* debe dar al mismo en caso de perder su meta. Existen dos alternativas: eliminar el proceso o dejarlo terminar. En éste último caso para evitar el riesgo de afectar al resto de los procesos, se lo convierte en opcional. Esta política se denomina "Aislamiento temporal de los procesos".

El manejo de pérdida de metas se configura en la creación de los procesos, incluyendo o no el bit *P_KILL_PROC_ON_MAN_DL* (el bit significa eliminar el proceso si pierde la meta) en el campo *flags* de la estructura *rt_cfg_t*. Por defecto el flag está prendido y se eliminan los procesos que no cumplen su meta.

Dependiendo de la configuración, este esquema puede permitir que en los procesos periódicos se acumulen períodos pendientes (partes o períodos enteros). El planificador debe memorizar los períodos pendientes y ejecutarlos cuando pueda.

El código fuente es el siguiente:

```
01    #include <stdio.h>
02    #include <minix/rt.h>
03    #include <sys/wait.h>
04    #include <unistd.h>
05
06    _PROTOTYPE( void main,(int argc, char **argv) );
```

```
07   _PROTOTYPE( void rt_function, (void) );
08
09   void rt_function(void)
10   {
11        proc_info_t proc_info;
12        rt_get_proc_info(getpid(), &proc_info);
13        printf(" Executing %s ... \n", proc_info.name);
14        rt_waste_time(proc_info.rt_cfg.mandatory_time, 0);
15   }
16
17   void main(argc, argv)
18   int argc;
19   char **argv;
20   {
21        int status;
22        pid_t A_pid, B_pid, C_pid;
23        rt_cfg_t A, B, C;
24
25        rt_init_cfg(&A);
26        rt_init_cfg(&B);
27        rt_init_cfg(&C);
28
29        A.exec_type          = EXEC_TYPE_PERIODIC;
30        A.period             = 4000;
31        A.mandatory_function = rt_function;
32        A.mandatory_time     = 2000;
33        A.max_periods        = 3;
34        A.weight             = 3;
35        A.ready_time         = 1000;
36
37        B.exec_type          = EXEC_TYPE_PERIODIC;
38        B.period             = 4000;
39        B.mandatory_function = rt_function;
40        B.mandatory_time     = 3000;
```

```
41          B.max_periods       = 3;
42          B.weight            = 2;
43          B.ready_time        = 1000;
44
45          C.exec_type         = EXEC_TYPE_PERIODIC;
46          C.period            = 4000;
47          C.mandatory_function = rt_function;
48          C.mandatory_time    = 3000;
49          C.max_periods       = 3;
50          C.weight            = 1;
51          C.ready_time        = 1000;
52          C.flags             &= ~P_KILL_PROC_ON_MAN_DL;
53
54          A_pid = rt_fork("Task A", &A);
55          B_pid = rt_fork("Task B", &B);
56          C_pid = rt_fork("Task C", &C);
57
58          if (A_pid>=1)
59               waitpid(A_pid, &status, 0);
60          if (B_pid>=1)
61               waitpid(B_pid, &status, 0);
62          if (C_pid>=1)
63               waitpid(C_pid, &status, 0);
64     }
```

Código fuente 3 - sample3.c

En este ejemplo se disparan tres procesos periódicos *A*, *B* y *C*. El algoritmo de planificación es por defecto *BWF*. Todos los procesos deben empezar a correr a los 1000 milisegundos (líneas 35, 43, 51) y ejecutar solo 3 períodos (líneas 33, 41 y 49). La función asociada a los procesos está definida en las líneas 09-15 y es similar a la del ejemplo anterior.

Los procesos *A* y *B* están configurados por defecto para que el *Núcleo* los elimine si pierden su meta. El proceso *C* es configurado para seguir ejecutando aunque pierda su meta (línea 52).

El proceso *A* es el de mayor peso y ejecuta sin problemas. El proceso *B* es el segundo de mayor peso y es eliminado en el segundo 5 por perder su meta. El proceso C pierde todas sus metas pero el *Núcleo* deja que termine de procesar en todos los casos.

La ejecución muestra por pantalla lo siguiente:

```
# Executing Task A ...
# Executing Task B ...
# Executing Task A ...
# Executing Task C ...
# Executing Task A ...
# Executing Task C ...
# Executing Task C ...
```

Si el Monitor de Tareas no está activo en la consola 1 se muestra lo siguiente:

```
# WARNING: Task C has missed 1 deadlines
# WARNING: Task C has missed 2 deadlines
# WARNING: Task C has missed 3 deadlines
```

El diagrama de tiempos es el siguiente:

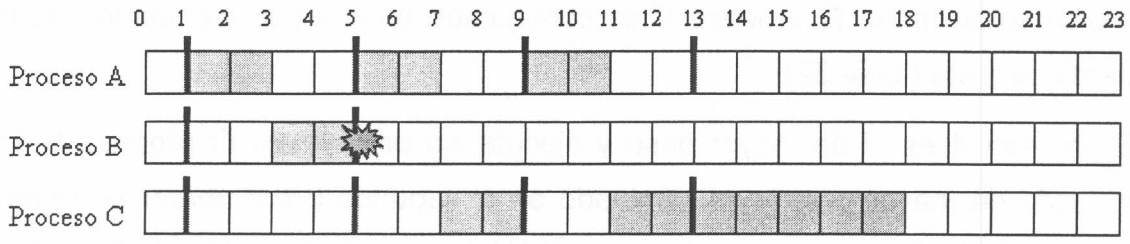

Figura 3 - Diagrama de tiempos de sample3.c

3.6. Computación Imprecisa

En RT-MINIX todos los procesos de tiempo real tienen una parte mandatoria y una parte opcional. Ambas pueden ser eventualmente nulas. La parte mandatoria debe ejecutarse por completo y dentro de las restricciones de tiempo. La parte opcional mejora los resultados pero puede no ejecutarse, ejecutarse parcialmente o postergarse si el sistema se encuentra sobrecargado.

Es aconsejable que el programador haga uso de la llamada al sistema *rt_get_remainder* en la parte opcional. Esta llamada al sistema permite obtener el tiempo restante para la meta y refinar los resultados interativamente sin causar problemas.

El código fuente es el siguiente:

```
01    #include <stdio.h>
02    #include <minix/rt.h>
03    #include <sys/wait.h>
04    #include <unistd.h>
05
06    _PROTOTYPE( void main, (int argc, char **argv) );
07    _PROTOTYPE( void rt_mandatory_function, (void) );
08    _PROTOTYPE( void rt_optional_function, (void) );
09
10    void rt_mandatory_function(void)
11    {
12          proc_info_t proc_info;
13          rt_get_proc_info(getpid(), &proc_info);
```

```
14          printf(" Executing %s (Mandatory Part) ... \n",
15               proc_info.name);
16          rt_waste_time(proc_info.rt_cfg.mandatory_time, 0);
17    }
18
19    void rt_optional_function(void)
20    {
21          proc_info_t proc_info;
22          rt_get_proc_info(getpid(), &proc_info);
23          printf(" Executing %s (Optional Part) ... \n",
24
25          while (1) {
26               if (rt_get_remainder()<=5)
27                    return;
28          }
29    }
30
31    void main(argc, argv)
32    int argc;
33    char **argv;
34    {
35          int status;
36          pid_t A_pid;
37          rt_cfg_t A;
38
39          rt_init_cfg(&A);
40
41          A.exec_type          = EXEC_TYPE_APERIODIC;
42          A.deadline           = 6000;
43          A.mandatory_function = rt_mandatory_function;
44          A.mandatory_time     = 4000;
45          A.optional_function  = rt_optional_function;
46          A.optional_time      = 4000;
47
48          A_pid = rt_fork("Task A", &A);
49
50          if (A_pid>=1)
51               waitpid(A_pid, &status, 0);
```

```
52    }
```

Código fuente 4 - sample4.c

En este ejemplo se lanza un proceso aperiódico y compuesto. La parte mandatoria es implementada por la función *rt_mandatory_function* (definida en las líneas 10-17) y la opcional por la función *rt_optional_*function (definida en las líneas 19-29).

La función mandatoria es similar a la que ya se utilizó en ejemplos anteriores. Obtiene el nombre del proceso, lo imprime y luego consume el tiempo equivalente al peor caso de ejecución, definido en la línea 44.

La función opcional obtiene e imprime el nombre del proceso y luego se queda iterando todo el tiempo posible, hasta que solo resten 5 milisegundos para la meta (línea 26).

La ejecución muestra por pantalla lo siguiente:

```
# Executing Task A (Mandatory Part) ...
# Executing Task A (Optional Part) ...
```

El diagrama de tiempos es el siguiente:

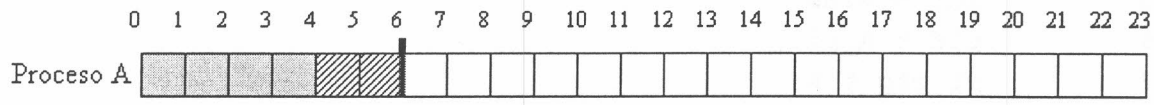

Figura 4 - Diagrama de tiempos de sample4.c

3.7. Sincronización de procesos

El mecanismo de sincronización de procesos implementado en RT-MINIX es el de *semáforos con techo de prioridades*. La secuencia de llamadas al sistema que involucra el uso de un semáforo es la siguiente:

```
...
int s = rt_create_semaphore("SEMAPHORE_NAME") (crea el semáforo)
      ...
      rt_lock_semaphore(s) (toma el semáforo)
            ...
```

```
            Utilización del recurso protegido por el semáforo
            ...
      rt_lock_semaphore(s) (libera el semáforo)
      ...
rt_delete_semaphore(s) (borra el semáforo)
...
```

Este ejemplo muestra como el uso de semáforos con techo de prioridades evita un abrazo mortal y consta de tres procesos aperiódicos *J1*, *J2* y *J3*. El proceso *J1* tiene mayor peso que el proceso *J2*, el proceso *J2* tiene mayor peso que el proceso *J3* y el algoritmo de planificación es *BWF*. En suma, hay dos estructuras de datos compartidas protegidas por los semáforos *S1* y *S2*, respectivamente. Como la prioridad del proceso *J2* es mayor que la del proceso *J3*, los techos de prioridades de los dos semáforos *S1* y *S2* son iguales a la prioridad del proceso *J2*.

El código fuente es el siguiente:

```
01    #include <stdio.h>
02    #include <minix/rt.h>
03    #include <sys/wait.h>
04    #include <unistd.h>
05
06    _PROTOTYPE( void main,      (int argc, char **argv) );
07    _PROTOTYPE( void fn_J1, (void) );
08    _PROTOTYPE( void fn_J2, (void) );
09    _PROTOTYPE( void fn_J3, (void) );
10
11    void fn_J1(void)
12    {
13          int S0 = rt_create_semaphore("S0");
14          rt_waste_time(2000, 0);
15          rt_lock_semaphore(S0);
16          printf(" J1 locks S0 \n");
17          rt_release_semaphore(S0);
```

```
18          rt_waste_time(1000, 0);
19          rt_delete_semaphore(S0);
20    }
21
22    void fn_J2(void)
23    {
24          int S1 = rt_create_semaphore("S1");
25          int S2 = rt_create_semaphore("S2");
26
27          rt_waste_time(2000, 0);
28          rt_lock_semaphore(S1);
29          printf(" J2 locks S1 \n");
30          rt_waste_time(1000, 0);
31          rt_lock_semaphore(S2);
32          printf(" J2 locks S2 \n");
33          rt_waste_time(2000, 0);
34          rt_release_semaphore(S2);
35          rt_waste_time(1000, 0);
36          rt_release_semaphore(S1);
37          rt_waste_time(1000, 0);
38          rt_delete_semaphore(S1);
39          rt_delete_semaphore(S2);
40    }
41
42    void fn_J3(void)
43    {
44          int S1 = rt_create_semaphore("S1");
45          int S2 = rt_create_semaphore("S2");
46          rt_waste_time(1000, 0);
47          rt_lock_semaphore(S2);
48          printf(" J3 locks S2 \n");
49          rt_waste_time(3000, 0);
50          rt_lock_semaphore(S1);
51          printf(" J3 locks S1 \n");
```

```
52          rt_waste_time(1000, 0);
53          rt_release_semaphore(S1);
54          rt_waste_time(1000, 0);
55          rt_release_semaphore(S2);
56          rt_waste_time(2000, 0);
57          rt_delete_semaphore(S1);
58          rt_delete_semaphore(S2);
59    }
60
61    void main(argc, argv)
62    int argc;
63    char **argv;
64    {
65          pid_t J1_pid, J2_pid, J3_pid;
66          rt_cfg_t J1, J2, J3;
67          int status;
68
69          rt_set_man_scheduler(SCHED_ALG_BWF);
70
71          rt_init_cfg(&J1);
72          rt_init_cfg(&J2);
73          rt_init_cfg(&J3);
74
75          J1.exec_type              = EXEC_TYPE_APERIODIC;
76          J1.ready_time             = 6000;
77          J1.deadline               = 22000;
78          J1.mandatory_function     = fn_J1;
79          J1.weight                 = 3;
80
81          J2.exec_type              = EXEC_TYPE_APERIODIC;
82          J2.ready_time             = 3000;
83          J2.deadline               = 22000;
84          J2.mandatory_function     = fn_J2;
85          J2.weight                 = 2;
```

```
86
87          J3.exec_type                = EXEC_TYPE_APERIODIC;
88          J3.ready_time               = 1000;
89          J3.deadline                 = 22000;
90          J3.mandatory_function       = fn_J3;
91          J3.weight                   = 1;
92
93          J1_pid = rt_fork("J1", &J1);
94          J2_pid = rt_fork("J2", &J2);
95          J3_pid = rt_fork("J3", &J3);
96
97          if (J1_pid>=1)
98               waitpid(J1_pid, &status, 0);
99          if (J2_pid>=1)
100              waitpid(J2_pid, &status, 0);
101         if (J3_pid>=1)
102              waitpid(J3_pid, &status, 0);
103    }
```

Código fuente 5 – sample5.c

En el segundo 1, el proceso *J3* es iniciado, comienza su ejecución y toma el semáforo *S2*. En el segundo 3, el proceso *J2* es iniciado y desaloja el proceso *J3*. En el segundo 5, el proceso *J2* intenta entrar en su zona crítica haciendo una llamada al sistema para ejecutar *P(S1)*. Sin embargo, el sistema va a encontrar que la prioridad del proceso *J2* no es mayor que la del techo de prioridad del semáforo tomado *S2*. Por lo tanto, el sistema suspende el proceso *J2* sin tomar *S1*. El proceso *J3* ahora hereda la prioridad del proceso *J2* y continúa su ejecución. Notar que *J2* está bloqueado fuera de su zona crítica. Como *J2* no puede tomar *S1* entonces es suspendido y el abrazo mortal entre *J2* y *J3* es prevenido. En el segundo 6, *J3* está todavía en su sección crítica. En ese momento el proceso *J1* (de mayor prioridad) es instanciado y desaloja *J3*. Mas tarde, *J1* intenta tomar el semáforo S0. Como la prioridad de *J1* es mayor que la prioridad del techo del semáforo tomado *S2*, el proceso *J1* va a poder tomar el

semáforo *S0*. Por lo tanto el proceso *J1* va a continuar y ejecutar su sección crítica, desalojando a *J3* de su sección crítica. En el segundo 10, *J1* ha terminado su sección crítica y completa su ejecución. El proceso *J3* continúa, entonces *J2* es bloqueado por *J3* y no puede ejecutar. *J3* continúa ejecutando y toma *S1*. En el segundo 12, *J3* libera *S1*. En el segundo 13, *J3* libera *S2* y continúa con su prioridad asignada. *J2* desaloja *J3* por tener mayor prioridad y toma *S2*. Luego, *J2* toma *S1*, ejecuta la sección crítica anidada y libera *S1*. Por último libera *S2* y ejecuta su sección de código no crítica. En el segundo 18, *J2* completa su ejecución y libera a *J3*. En el segundo 20, *J3* completa su ejecución.

La ejecución muestra por pantalla lo siguiente:

```
# J3 locks S2
# J1 locks S0
# J3 locks S1
# J2 locks S1
# J2 locks S2
```

El diagrama de tiempos es el siguiente:

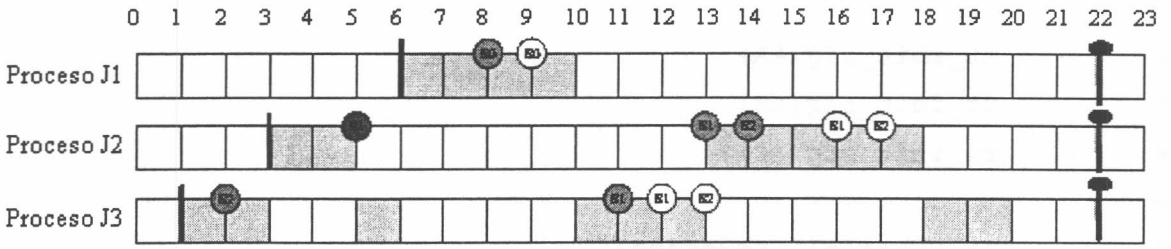

Figura 5 - Diagrama de tiempos de sample5.c

3.8. Estadísticas del sistema

El *Núcleo* de RT-MINIX mantiene internamente un conjunto contadores que permiten analizar el desempeño del planificador. Cada contador es un número entero que se inicializa en cero y se incrementa cuando ocurren determinados eventos.

Este ejemplo muestra algunos de los contadores disponibles y el uso de las llamadas al sistema *rt_reset_counter*, *rt_reset_all_counters* y *rt_get_counter*.

El código fuente es el siguiente:

```
01   #include <stdio.h>
02   #include <minix/rt.h>
03   #include <sys/wait.h>
04   #include <unistd.h>
05
06   _PROTOTYPE( void main, (int argc, char **argv) );
07
08   void main( int argc, char *argv[])
09   {
10        int status;
11        pid_t A_pid, B_pid, C_pid;
12        rt_cfg_t A, B, C;
13        long tpi, spi, tpw, spw;
14        long oldtime;
15
16        rt_init_cfg(&A);
17        rt_init_cfg(&B);
18        rt_init_cfg(&C);
19
20        rt_reset_all_counters();
21        rt_set_man_scheduler(SCHED_ALG_RM);
22
23        A.exec_type          = EXEC_TYPE_PERIODIC;
24        A.period             = 4000;
25        A.ready_time         = 1000;
26        A.mandatory_time     = 1000;
27        A.max_life_time      = 21000;
28        A.mandatory_function = rt_default_man_function;
29        A.weight             = 3;
30        A.flags              &= ~P_KILL_PROC_ON_MAN_DL;
```

```
31
32        B.exec_type           = EXEC_TYPE_PERIODIC;
33        B.period              = 8000;
34        B.ready_time          = 1000;
35        B.mandatory_time      = 2000;
36        B.max_life_time       = 21000;
37        B.mandatory_function  = rt_default_man_function;
38        B.weight              = 2;
39        B.flags               &= ~P_KILL_PROC_ON_MAN_DL;
40
41        C.exec_type           = EXEC_TYPE_PERIODIC;
42        C.period              = 8000;
43        C.ready_time          = 1000;
44        C.mandatory_time      = 5000;
45        C.max_life_time       = 21000;
46        C.mandatory_function  = rt_default_man_function;
47        C.weight              = 1;
48        C.flags               &= ~P_KILL_PROC_ON_MAN_DL;
49
50        oldtime = rt_get_uptime();
51
52        A_pid = rt_fork("Task A", &A);
53        B_pid = rt_fork("Task B", &B);
54        C_pid = rt_fork("Task C", &C);
55
56        if (A_pid>=1)
57             waitpid(A_pid, &status, 0);
58
59        if (B_pid>=1)
60             waitpid(B_pid, &status, 0);
61
62        if (C_pid>=1)
63             waitpid(C_pid, &status, 0);
64
```

```
65          printf(" Processing Time: %lu \n ",
66                rt_get_uptime() - oldtime);
67
68          tpi = rt_get_counter(CT_TOTAL_PERIODIC_INSTANCES);
69          spi = t_get_counter(CT_SUCCESSFUL_PERIODIC_INSTANCES);
70          tpw = rt_get_counter(CT_TOTAL_PERIODIC_WEIGHT);
71          spw = rt_get_counter(CT_SUCCESSFUL_PERIODIC_WEIGHT);
72
73          printf(" Succesful Goals: %u \n", spi );
74          printf(" Total Goals: %u \n", tpi );
75          printf(" Succesful Rate: %g \n",
76                (float) (spi) / (float) 76 (tpi) );
77
78          printf(" Pondered Succesful Goals: %u \n",  (spw) );
79          printf(" Pondered Total Goals: %u \n",  (tpw) );
80          printf(" Pondered Succesful Rate: %g \n",
81                (float) (spw ) / (float) (tpw ) );
82    }
```

Código fuente 6 - sample6.c

En este ejemplo se corren 3 procesos periódicos que ejecutan la cantidad de tiempo especificada en el campo *max_life_time* (líneas 27, 36 y 45) de la estructura *rt_cfg_t*. Todos los procesos tienen asociada la función *rt_default_man_function* que está definida en la biblioteca de tiempo real y permite emular una ejecución equivalente al peor tiempo de ejecución establecido para el proceso. En la línea 20 se inicializan todos los contadores. En la línea 50 se guarda el tiempo actual. En la línea 21 se establece que el algoritmo para procesos mandatorios es SCHED_ALG_RM.

Empieza a ejecutar la tarea A para por tener menor período, luego sigue la tarea B y finalmente C. Si bien las tareas B y C tienen el mismo período, el planificador elige a B porque tiene mayor peso.

El tiempo que duró la ejecución de los procesos se imprime en la línea 65. Para calcularlo se hace la diferencia entre el tiempo actual y el almacenado en la línea 50.

En las líneas 68 a 71 se obtienen los contadores y en las líneas 73 a 80 se imprimen los cálculos realizados.

La ejecución muestra por pantalla lo siguiente:

```
# Processing Time: 20033
# Succesful Goals: 8
# Total Goals: 10
# Succesful Rate: 0,8
# Pondered Succesful Goals: 21
# Pondered Total Goals: 24
# Pondered Succesful Rate: 0,875
```

El diagrama de tiempos es el siguiente:

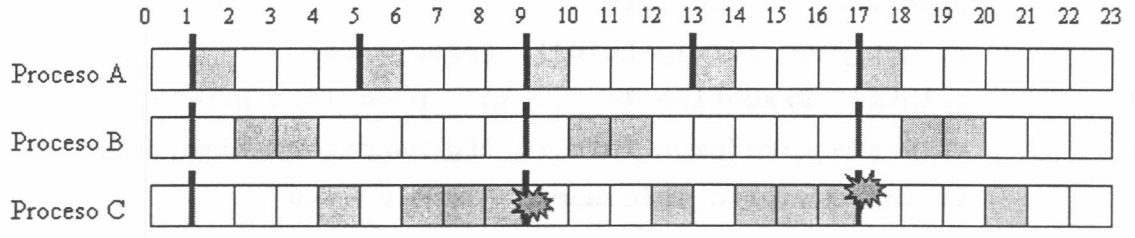

Figura 6 – Diagrama de tiempos de sample6.c

3.9. Tolerancia de fallas transitorias

Las fallas transitorias se recuperan mediante reejecución. La naturaleza transitoria de la falla hace suponer que no volverá a ocurrir, pero si volviese a ocurrir se repite el procedimiento.

La detección de fallas transitorias se realiza mediante la incorporación de CRC en el segmento TEXT y / o en los mensajes del proceso. La ocurrencia de éste tipo de fallas puede emularse con la llamada al sistema *rt_inject_fault*.

El ejemplo consta de dos procesos periódicos. El proceso *A* ejecuta sin problemas y el proceso *B* en el primer período se inyecta a si mismo una falla transitoria mediante la llamada al sistema *rt_inject_fault*.

El código fuente es el siguiente:

```
01   #include <stdio.h>
02   #include <minix/rt.h>
03   #include <sys/wait.h>
04   #include <unistd.h>
05   #include <string.h>
06
07   _PROTOTYPE( void main, (int argc, char **argv) );
08   _PROTOTYPE( void rt_mandatory_function, (void) );
09
10   void rt_mandatory_function(void)
11   {
12        proc_info_t proc_info;
13        rt_get_proc_info(getpid(), &proc_info);
14        printf(" Executing %s ... \n", proc_info.name);
15        rt_waste_time(proc_info.rt_cfg.mandatory_time, 0);
16        if (strcmp(proc_info.name, "Task B")==0)
17        {
18             printf(" Fault injected in Task B...\n");
19             rt_inject_fault(getpid(), TEXT_FAULT);
20        }
21   }
22
23   void main(argc, argv)
24   int argc;
25   char **argv;
26   {
27        int status;
28        pid_t A_pid, B_pid;
29        rt_cfg_t A, B;
```

```
30
31          rt_init_cfg(&A);
32          rt_init_cfg(&B);
33
34          A.exec_type           = EXEC_TYPE_PERIODIC;
35          A.ready_time          = 1000;
36          A.period              = 9000;
37          A.max_periods         = 3;
38          A.mandatory_time      = 2000;
39          A.mandatory_function  = rt_mandatory_function;
40          A.weight              = 1;
41
42          B.exec_type           = EXEC_TYPE_PERIODIC;
43          B.ready_time          = 3000;
44          B.period              = 9000;
45          B.max_periods         = 3;
46          B.mandatory_time      = 2000;
47          B.mandatory_function  = rt_mandatory_function;
48          B.weight              = 2;
49          B.text_crc_period     = 10;
50          B.flags               = P_TEXT_CRC_ENABLED;
51
52          A_pid = rt_fork("Task A", &A);
53          B_pid = rt_fork("Task B", &B);
54
55          if (A_pid>=1)
56               waitpid(A_pid, &status, 0);
57          if (B_pid>=1)
58               waitpid(B_pid, &status, 0);
59    }
```

Código fuente 7 – sample7.c

Ambos procesos tienen asociada la función *rt_mandatory_function*, definida en las líneas 10 a 21. Esta función imprime en pantalla el nombre del proceso y deja

pasar un tiempo equivalente al peor caso de ejecución mediante la llamada al sistema *rt_waste_time*. Luego, si el nombre del proceso es *B*, se inyecta una falla. La reejecución de *B* no falla porque el *Núcleo* le asigna el nombre *B_bkp* (diferente de *B*).

El chequeo de CRC en el segmento TEXT se habilita agregando el bit *P_TEXT_CRC_ENABLED* al campo *flags* (línea 50). La periodicidad del chequeo se configuró en 10 milisegundos mediante el campo *text_crc_period* (línea 49). El valor por defecto del campo *text_crc_period* es 1000 milisegundos.

El proceso *A* ejecuta hasta el segundo 3 donde es interrumpido por el proceso *B* de mayor prioridad. En el primer período, B inyecta la falla y a partir de ese momento sigue ejecutando una copia de B.

La ejecución muestra por pantalla lo siguiente:

```
# Executing Task A ...
# Executing Task B ...
# Executing Task B_bkp ...
# Executing Task A ...
# Executing Task B_bkp ...
# Executing Task A ...
# Executing Task B_bkp ...
```

El diagrama de tiempos es el siguiente:

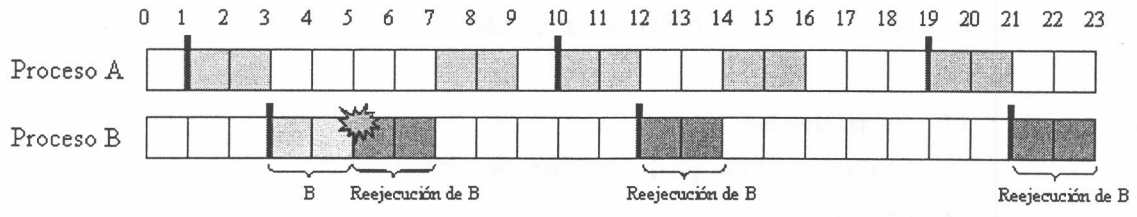

Figura 7 - Diagrama de tiempos de sample7.c

3.10. Tolerancia de fallas de software

Las fallas de software se recuperan mediante la ejecución de versiones alternativas del mismo proceso. Este tipo de fallas puede detectarlas el usuario

mediante aserciones (usando la llamada al sistema *rt_assert*) o el procesador a través de excepciones. Las versiones de backup deben especificarse en la creación del proceso mediante la llamada al sistema *rt_add_bkp*. Se pueden configurar varias versiones que se ejecutarán en forma consecutiva.

El ejemplo consta de dos procesos periódicos *A* y *B*. El proceso *A* tiene solo una versión de backup y el proceso *B* tiene dos.

El código fuente es el siguiente:

```
01 #include <stdio.h>
02 #include <minix/rt.h>
03 #include <sys/wait.h>
04 #include <unistd.h>
05 #include <string.h>
06
07 _PROTOTYPE( void main, (int argc, char **argv) );
08 _PROTOTYPE( void rt_mandatory_function, (void) );
09
10 void rt_mandatory_function(void)
11 {
12   proc_info_t proc_info;
13   rt_get_proc_info(getpid(), &proc_info);
14   printf(" Executing %s ... \n", proc_info.name);
15   rt_waste_time(proc_info.rt_cfg.mandatory_time, 0);
16   rt_assert(strcmp(proc_info.name,"Task B")==0, "Err1");
17   rt_assert(strcmp(proc_info.name,"Task B_bkp1")==0, "Err2");
18 }
19
20 void main(argc, argv)
21 int argc;
22 char **argv;
23 {
24   int status;
25   pid_t A_pid, B_pid;
```

```
26    rt_cfg_t A, B;
27
28    rt_init_cfg(&A);
29    rt_init_cfg(&B);
30
31    A.exec_type          = EXEC_TYPE_PERIODIC;
32    A.ready_time         = 1000;
33    A.period             = 12000;
34    A.max_periods        = 3;
35    A.fail_time          = 16900;
36    A.mandatory_time     = 3000;
37    A.mandatory_function = rt_mandatory_function;
38    A.weigth             = 1;
39
40    rt_add_bkp(&A, "Task A_bkp1", rt_mandatory_function,
41                3000, NULL, 0);
42
43    B.exec_type          = EXEC_TYPE_PERIODIC;
44    B.ready_time         = 3000;
45    B.period             = 10000;
46    B.max_periods        = 3;
47    B.mandatory_time     = 3000;
48    B.mandatory_function = rt_mandatory_function;
49    B.weigth             = 2;
50
51    rt_add_bkp(&B, "Task B_bkp1", rt_mandatory_function,
52              3000, NULL, 0);
53    rt_add_bkp(&B, "Task B_bkp2", rt_mandatory_function,
54                2000, NULL, 0);
55
56    A_pid = rt_fork("Task A", &A);
57    B_pid = rt_fork("Task B", &B);
58
59    if (A_pid>=1)
```

```
60          waitpid(A_pid, &status, 0);
61    if (B_pid>=1)
62          waitpid(B_pid, &status, 0);
63}
```

Código fuente 8 - sample8.c

Las versiones de backup se configuraron mediante la llamada al sistema *rt_add_bkp* en las líneas 40, 51 y 53. La falla del proceso *A* se emula mediante el uso del campo *fail_time* de la estructura *rt_cfg_t*. La falla del proceso B se emula sus dos fallas mediante la llamada al sistema *rt_assert* en las líneas 16 y 17.

La ejecución muestra por pantalla lo siguiente:

```
Executing Task A ...
Executing Task B ...
RT assertion "Err1" failed at sample8.c(16)
Executing Task B_bkp1 ...
RT assertion "Err2" failed at sample8.c(17)
Executing Task B_bkp2 ...
Executing Task A ...
Executing Task B_bkp2 ...
Executing Task A_bkp1 ...
Executing Task B_bkp2 ...
Executing Task A_bkp1 ...
```

El diagrama de tiempos es el siguiente:

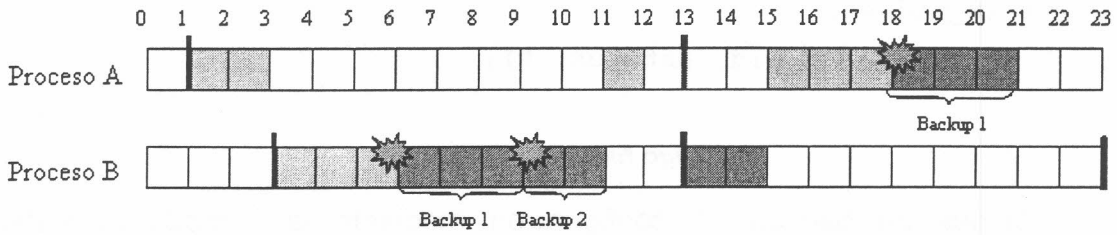

Figura 8 - Diagrama de tiempos de sample8.c

3.11. Mecanismos de detección de sobrecarga y Multicolas

La detección de sobrecarga se habilita y deshabilita mediante las llamadas al sistema *rt_enable_overload_check* y *rt_disable_overload_check*.

Si se habilita el mecanismo y se detecta sobrecarga, el *Núcleo* dispara en forma automática el *mecanismo de multicolas*. La sobrecarga indica que necesariamente se perderán metas y que debe evitarse que estas correspondan a procesos de importancia. La importancia de los procesos se especifica mediante el campo *weight* de la estructura *rt_cfg_t*. Durante el período de sobrecarga se deja de lado el algoritmo de planificación vigente y se planifica por peso y mediante *Multicolas*. Cuando pasa la sobrecarga se activa nuevamente el algoritmo que corresponda.

Este ejemplo debe ejecutarse en la consola 2, habiendo previamente activado el *Monitor de Tareas* en la consola 1. El *Monitor de Tareas* indica en el extremo superior derecho de la consola 1 la presencia de sobrecarga.

El código fuente es el siguiente:

```
01    #include <stdio.h>
02    #include <minix/rt.h>
03    #include <sys/wait.h>
04    #include <unistd.h>
05
06    _PROTOTYPE( void main,      (int argc, char **argv) );
```

```
07
08    void main(argc, argv)
09    int argc;
10    char **argv;
11    {
12          int status;
13          pid_t pid_A, pid_B, pid_C;
14          rt_cfg_t A, B, C;
15
16          rt_enable_overload_check();
17          rt_set_man_scheduler(SCHED_ALG_SJF);
18          rt_select_console(0);
19
20          rt_init_cfg(&A);
21          rt_init_cfg(&B);
22          rt_init_cfg(&C);
23
24          A.exec_type          = EXEC_TYPE_PERIODIC;
25          A.ready_time         = 1000;
26          A.period             = 8000;
27          A.max_periods        = 3;
28          A.mandatory_time     = 2000;
29          A.mandatory_function = rt_default_man_function;
30          A.weight             = 1;
31
32          B.exec_type          = EXEC_TYPE_PERIODIC;
33          B.ready_time         = 1000;
34          B.period             = 8000;
35          B.max_periods        = 3;
36          B.mandatory_time     = 3000;
37          B.mandatory_function = rt_default_man_function;
38          B.weight             = 2;
39
40          C.exec_type          = EXEC_TYPE_PERIODIC;
```

```
41          C.ready_time          = 9000;
42          C.period              = 8000;
43          C.mandatory_time      = 4000;
44          C.mandatory_function  = rt_default_man_function;
45          C.max_periods         = 2;
46          C.weight              = 3;
47
48          pid_A = rt_fork("A", &A);
49          pid_B = rt_fork("B", &B);
50          pid_C = rt_fork("C", &C);
51
52          if (pid_A>=1)
53               waitpid(pid_A, &status, 0);
54          if (pid_B>=1)
55               waitpid(pid_B, &status, 0);
56          if (pid_C>=1)
57               waitpid(pid_C, &status, 0);
58     }
```

Código fuente 9 - sample9.c

En la línea 16 se habilita el mecanismo de multicolas mediante la llamada al sistema *rt_enable_overload_check*. En la línea 17 se configura al algoritmo SJF (Shortest Job First). En la línea 18 se utiliza la llamada al sistema *rt_select_console* para cambiar a la consola 1 en forma automática y ver la corrida en el *Monitor de Tareas*.

Se lanzan tres procesos periódicos: *A* de peso 1, *B* de 2 y *C* de 3. Los procesos *A* y *B* comienzan a ejecutar normalmente. El planificador elige primero el proceso *A* pues tiene menor tiempo de procesamiento. Cuando se suma el proceso *C* el planificador detecta sobrecarga, activa el mecanismo de *Multicolas* y elige al proceso *C* que es el de mayor peso. Luego de *C* ejecuta *B* porque tiene mayor peso que *A*. En el segundo 16 A toma el procesador pero no logra terminar porque llega su meta y muere. Con la muerte de *A* desaparece la sobrecarga, se desactiva el mecanismo de multicolas y vuelve a usarse el algoritmo SJF. Ahora el

planificador elige al proceso *B* pues tiene menor tiempo de procesamiento que el proceso *C*.

El diagrama de tiempos es el siguiente:

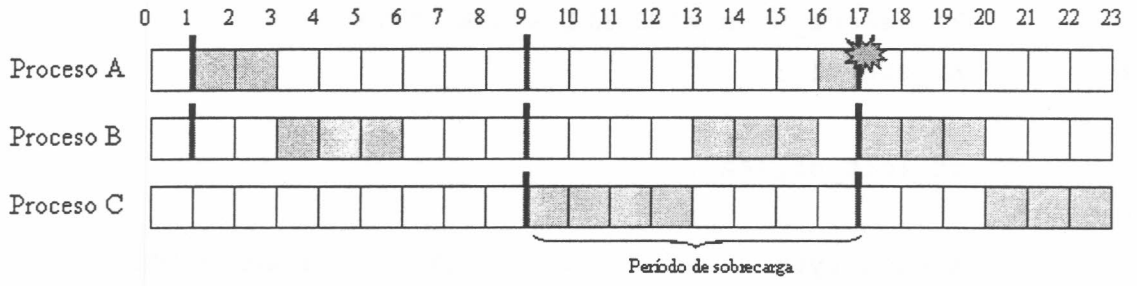

Figura 9 - Diagrama de tiempos de sample9.c

3.12. Log del Núcleo

El *Log del Núcleo* registra los eventos del sistema sin afectar el normal funcionamiento del mismo. Más tarde es posible analizar el comportamiento del sistema mediante el *Visor de Log* o bajando la información a un archivo.

En el ejemplo se dispara un proceso aperiódico registrando todos los eventos del *Núcleo* desde que se inicializa hasta que termina de ejecutar.

El código fuente es el siguiente:

```
01    #include <stdio.h>
02    #include <minix/rt.h>
03    #include <minix/log.h>
04    #include <sys/wait.h>
05
06    _PROTOTYPE( void main, (int argc, char **argv) );
07
08    void main(argc, argv)
09    int argc;
10    char **argv;
11    {
12          int status;
```

```
13          pid_t A_pid;
14          rt_cfg_t A;
15
16          kl_clear();
17          kl_select_events(PE_ALL & ~PE_OTHER);
18          kl_start();
19
20          rt_init_cfg(&A);
21
22          A.exec_type                 = EXEC_TYPE_APERIODIC;
23          A.deadline                  = 4000;
24          A.ready_time                = 1000;
25          A.mandatory_time      = 2000;
26          A.mandatory_function  = rt_default_man_function;
27
28          A_pid = rt_fork("Task A", &A);
29
30          if (A_pid>=1)
31               waitpid(A_pid, &status, 0);
32
33          kl_stop();
34          kl_save("out.txt");
35    }
```

Código fuente 10 - sample10.c

En la línea 16 se limpia el buffer del Log con la llamada al sistema *kl_clear*. En la línea 17 se establece mediante la llamada al sistema *kl_select_events* que se registrarán todos los eventos salvo los creados por el usuario. El Log se activa con la llamada al sistema *kl_start* (línea 18) y se desactiva con la llamada al sistema *kl_stop* (línea 33). Finalmente y mediante la llamada al sistema *kl_save* (línea 34) se graba el Log en el archivo out.txt. Otra opción es usar el *Visor de Log* mediante la llamada al sistema *kl_show*.

El archivo generado es el siguiente:

```
15172           Task A_S-63(63) PE_FORKED
15172           Task A_S-63(63) PE_PICKED
15172           Task A_S-63(63) PE_UNPICKED
15172           sample9-62(62) PE_PICKED
15172           Task A-64(64) PE_FORKED
15172           sample9-62(62) PE_UNPICKED
15172           sample9-62(62) PE_PICKED
15172           sample9-62(62) PE_UNPICKED
15232           Task A-64(64) PE_READY_TIME_REACHED
15232           Task A-64(64) PE_PICKED
15353           Task A-64(64) PE_UNPICKED
15353           Task A-64(64) PE_PICKED
15353           Task A-64(64) PE_TERMINATED_SUCCESSFULLY
15353           Task A_S-63(63) PE_PICKED
15353           Task A_S-63(63) PE_TERMINATED_SUCCESSFULLY
15353           sample9-62(62) PE_PICKED
```

out.txt

En cada línea se muestra: tiempo en ticks, nombre y PID del proceso y evento.

4. Referencia de la API

Las extensiones de tiempo real y tolerancia a fallas incorporadas forman parte de la biblioteca de tiempo real.

Esta biblioteca concentra todas las llamadas al sistema necesarias para la creación y manejo de procesos de tiempo real, semáforos con techo de prioridades, reloj del sistema, planificador, monitor de tareas, estadísticas y mecanismos de tolerancia a fallas.

En otra biblioteca que en esta sección se presenta en forma conjunta se incluyen las llamadas al sistema para manejo del Log a nivel *Núcleo*.

4.1. Funciones

kl_start

```
int kl_start(void)
```

Habilita el Log, el *Núcleo* comienza a registrar eventos.

kl_stop

```
int kl_stop(void)
```

Deshabilita el Log, el *Núcleo* deja de registrar eventos.

kl_clear

```
int kl_clear(void)
```

Limpia al Log, elimina toda la información del buffer. Esta función tiene el mismo efecto que presionar la tecla F8 en la consola 1.

kl_show

```
int kl_show(void)
```

Permite mostrar el visor de Log. Esta función tiene el mismo efecto que presionar la tecla F7 en la consola 1.

kl_save

```
int kl_save(char* file)
```

Permite grabar el contenido del buffer de Log en el archivo *file*.

kl_select_events

```
int kl_select_events(long events)
```

Permite especificar los tipos de eventos que el *Núcleo* debe registrar. La mayoría de los eventos definidos están asociados a cambios de estado de los procesos. El parámetro *events* en un mapa de bits basado en la Tabla 1.

Evento	Descripción
PE_ALL	Todos los eventos (todos los bits en 1)
PE_OTHER	Evento de usuario
PE_EXECUTED	EXEC realizado
PE_FORKED	FORK realizado
PE_UNHANDLED_EXCEPTION	Excepción no manejada
PE_TERMINATED_SUCCESSFULLY	Terminación exitosa
PE_PICKED	Toma el procesador
PE_UNPICKED	Libera el procesador
PE_READY_TIME_REACHED	Tiempo de listo alcanzado
PE_NEXT_PERIOD_TIME_REACHED	Tiempo de próximo período alcanzado
PE_DEADLINE_TIME_REACHED	Tiempo de meta alcanzado
PE_PERIOD_ENDED	Período finalizado
PE_BLOCKED_BY_SEMAPHORE	Intento fallido de toma de semáforo
PE_UNBLOCKED_BY_SEMAPHORE	Semáforo tomado
PE_PAUSE	Pausa
PE_SIGNAL_RECEIVED	Señal recibida

PE_TEXT_CRC_FAILED	Falló el CRC del segmento TEXT
PE_MSG_CRC_FAILED	Falló el CRC de un mensaje enviado
PE_ASSERTION_FAILED	Falló una aserción

Tabla 1 - Eventos

rt_add_bkp

```
int rt_add_bkp(rt_cfg_t *rt_cfg_ptr, char *name, proc_func_t
               mandatory_function, clock_t mandatory_time,
               proc_func_t optional_function, clock_t
               optional_time)
```

Permite agregar una nueva versión de backup del proceso a la estructura *rt_cfg_t* apuntada por *rt_cfg_ptr.* El parámetro *name* establece el nombre de la nueva versión mientras que los parámetros *mandatory_function, mandatory_time, optional_function, optional_time* configuran la función y tiempo de duración de las partes mandatoria y opcional respectivamente.

El *Núcleo* utiliza las versiones de backup cuando se produce una falla de software.

rt_assert

```
void rt_assert(BOOL expression, char *msg)
```

Permite ejecutar una aserción. El parámetro *expresión* es la expresión booleana que debe evaluarse y el parámetro *msg* es el texto de error que debe mostrarse si la evaluación de la expresión es FALSE. En este caso también se dispara el mecanismo de recuperación para fallas de diseño de software.

rt_create_semaphore

```
int rt_create_semaphore(char *name)
```

Si ya existe un semáforo con nombre *name*, retorna su *ID*. Si no existe, lo crea y retorna su *ID*.

rt_default_man_function

```
rt_default_man_function(void)
```

Función que emula la ejecución de la parte mandatoria del proceso que la llama. La emulación se realiza mediante la función *rt_waste_time*.

rt_default_opt_function

```
rt_default_opt_function(void)
```

Función que emula la ejecución de la parte opcional del proceso que la llama. La emulación se realiza mediante la función *rt_waste_time*.

rt_delete_semaphore

```
int rt_delete_semaphore(int handle)
```

Destruye el semáforo identificado por *handle*. Como el mismo semáforo puede estar siendo utilizado por otros procesos es posible no se destruya efectivamente, que solamente se lo desvincule de este proceso. La rutina retorna *EINVSH* si *handle* es inválido sino retorna *OK*.

rt_disable_overload_check

```
int rt_disable_overload_check (void)
```

Deshabilita los mecanismos de detección de sobrecarga y multicolas.

rt_enable_overload_check

```
int rt_enable_overload_check (void)
```

Habilita los mecanismos de detección de sobrecarga y multicolas.

rt_fork

```
pid_t rt_fork(char* name, rt_cfg_t* rt_cfg_ptr)
```

Permite crear un proceso de tiempo real. El parámetro *rt_cfg_ptr* es un puntero a una estructura de tipo *rt_cfg_t* donde se especifican todos los parámetros necesarios para la creación del nuevo proceso. Las estructura *rt_cfg_t* debe inicializarse con la función *rt_init_cfg*, luego configurarse y por último validarse con la función *rt_validate_cfg*.

rt_get_counter

```
long rt_get_counter(int counter)
```

Retorna el valor actual del contador *counter*. Los valores posibles del parámetro *counter* están especificados en la sección *rt_reset_counter*.

rt_get_grain

```
long rt_get_grain()
```

Permite obtener la granularidad del reloj del sistema. La granularidad es la cantidad de ticks por segundo (configurable por software en las IBM-PC). En RT-MINIX vale por defecto 60 pero puede modificarse mediante la rutina *rt_set_grain*.

rt_get_next_pid

```
pid_t rt_get_next_pid(pid_t pid)
```

Permite obtener los PIDs de todos los procesos existentes en el sistema.

Retorna el *PID* del proceso siguiente al identificado por *pid* en la tabla de procesos. Para obtener el primero es necesario mandar la constante *LONG_MIN* como parámetro. Si *pid* es el identificador del último proceso la rutina retorna *LONG_MIN*.

rt_get_proc_info

```
int rt_get_proc_info(pid_t pid, proc_info_t* proc_info_ptr)
```

Permite obtener toda la información asociada a un proceso a partir de su *PID*. Si el parámetro *pid* es inválido la rutina retorna *ESRCH*, en caso contrario retorna *OK* y la información del proceso en la estructura *proc_info_t* apuntada por *proc_info_ptr*.

rt_get_remainder

```
clock_t rt_get_remainder()
```

Retorna el tiempo que falta para que se cumpla la meta del proceso que la invoca, expresado en milisegundos.

rt_get_uptime

```
clock_t rt_get_uptime()
```

Retorna el tiempo transcurrido desde que se inició el sistema, expresado en milisegundos.

rt_hide_task_monitor

```
void rt_hide_task_monitor()
```

Oculta la consola del *Monitor de Tareas*. Tiene exactamente el mismo efecto que presionar F7, permite que se pueda realizar en forma programática, desde un proceso.

rt_init_cfg

```
void rt_init_cfg(rt_cfg_t* rt_cfg_ptr)
```

Permite inicializar los campos de la estructura *rt_cfg_t* apuntada por *rt_cfg_ptr*, que será utilizada más tarde para crear un proceso de tiempo real mediante la rutina *rt_fork*.

rt_inject_fault

```
void rt_inject_fault(pid_t pid, int fault)
```

Esta función se utiliza para prueba y sirve para simular una falla de hardware transitoria en el proceso identificado por *pid*. La falla puede inyectarse en el segmento TEXT (*fault* debe valer *TEXT_FAULT*) o en un mensaje enviado o recibido por el proceso (*fault* debe valer *MESSAGE_FAULT*).

rt_lock_semaphore

```
int rt_lock_semaphore(int handle)
```

Intenta tomar el semáforo identificado por *handle*. Si el semáforo está tomado por otro proceso, éste proceso se bloquea en espera hasta su liberación. La rutina retorna *EINVSH* si *handle* es inválido, *ESALRL* si se produjo un abrazo mortal y *OK* en caso contrario.

rt_release_semaphore

```
int rt_release_semaphore(int handle)
```

Libera el semáforo identificado por *handle* permitiendo que el proceso de mayor prioridad de los bloqueados por este semáforo pase a estar listo para ejecutar. La rutina retorna *EINVSH* si *handle* es inválido o el semáforo no está tomado por éste proceso y *OK* en caso contrario.

rt_reset_counter

```
void rt_reset_counter(int counter)
```

Inicializa el contador *counter*, le asigna el valor 0 para que empiece a contar nuevamente.

La siguiente tabla muestra todos los valores posibles de *counter*.

Contador	**Descripción**
CT_IDLE_TIME	Cantidad de tiempo (en ticks) ocioso del sistema.
CT_CONTEXT_SWITCHS	Cantidad de cambios de contexto realizados por el planificador.

CT_REMOTIONS	Cantidad de remociones de procesos. Las remociones se deben a la aparición de un proceso de mayor prioridad.
CT_TOTAL_PERIODIC_INSTANCES	Cantidad total de períodos planificados para todos los procesos periódicos.
CT_SUCCESSFUL_PERIODIC_INSTANCES	Cantidad total de períodos ejecutados en forma exitosa para todos los procesos periódicos.
CT_TOTAL_APERIODIC_INSTANCES	Cantidad total de procesos aperiódicos planificados.
CT_SUCCESSFUL_APERIODIC_INSTANCES	Cantidad total de procesos aperiódicos ejecutados en forma exitosa.
CT_TOTAL_PERIODIC_WEIGHT	Sumatoria de los pesos asociados a todos los períodos planificados para todos los procesos periódicos.
CT_SUCCESSFUL_PERIODIC_WEIGHT	Sumatoria de los pesos asociados a todos los períodos ejecutados en forma exitosa para todos los procesos periódicos.
CT_TOTAL_APERIODIC_WEIGHT	Sumatoria de los pesos asociados a todos los procesos aperiódicos planificados.
CT_SUCCESSFUL_APERIODIC_WEIGHT	Sumatoria de los pesos asociados a todos los procesos aperiódicos ejecutados en forma exitosa.

Tabla 2 - Contadores de estadísticas

rt_reset_all_counters

```
void rt_reset_all_counters()
```

Inicializa todos los contadores, les asigna el valor 0 para que empiecen a contar nuevamente.

rt_robsen

```
int rt_robsen(int algorithm, struct sensor_reading readings,
              struct sensor_reading result)
```

Aplica un algoritmo de sensado robusto a un conjunto de lecturas de sensores.

El algoritmo a utilizar se indica con el parámetro *algorithm*. Los valores posibles son:

Algoritmo	Descripción
AAA	Approximate Agreement Algorithm
FCA	Fast Convergence Algorithm
BIH	Brooks-Iyengar Hybrid
ORA	Optimal Region Algorithm

Tabla 3 - Algoritmos de sensado robusto

El parámetro *readings* es un arreglo con las lecturas de los distintos sensores.

La lectura correcta luego de aplicar el algoritmo pedido al conjunto de lecturas efectuadas es retornada a través de *result*.

Esta función se conserva de versiones previas de RT-MINIX.

rt_select_console

```
int rt_select_console(int console)
```

Permite cambiar la consola actual. Tiene el mismo efecto que presionar la combinación *ALT+F<console+1>*, permite que se pueda realizar en forma programática, desde un proceso.

rt_set_grain

```
void rt_set_grain(long newgrain)
```

Permite cambiar la granularidad del reloj del sistema. El parámetro *newgrain* representa la nueva granularidad expresada en ticks por segundo.

rt_set_man_scheduler

```
void rt_set_man_scheduler(int newsched)
```

Permite cambiar el algoritmo de planificación que se aplica al conjunto de procesos mandatorios. El parámetro *newsched* puede tomar uno de los siguientes valores:

Algoritmo	Descripción
SCHED_ALG_SJF	Trabajo Más Corto Primero
SCHED_ALG_BWF	Trabajo Más Importante Primero
SCHED_ALG_RM	Tasa Monotónica
SPLSCHED_ALG_EDF	Meta Más Cercana Primero
SCHED_ALG_FIFO	Primero en Entrar Primero en Salir
SCHED_ALG_SRTF	Tiempo Restante Más Corto Primero

Tabla 4 - Algoritmos de planificación

rt_set_opt_scheduler

```
void rt_set_opt_scheduler(int newsched)
```

Permite cambiar el algoritmo de planificación que se aplica al conjunto de procesos opcionales. Los valores posibles del parámetro *newsched* están especificados en la rutina *rt_set_man_scheduler*.

rt_set_proc_info

```
int rt_set_proc_info(pid_t pid, proc_info_t* proc_info_ptr)
```

Permite modificar las características y restricciones de tiempo de un proceso. El parámetro *pid* identifica el proceso y el parámetro *proc_info_ptr* apunta a una estructura de tipo *proc_info_t* con la nueva información del proceso. Se debe invocar primero a la rutina *rt_get_proc_info* para inicializar la estructura con la información actual del proceso, luego cambiar los campos que correspondan y por

último invocar *rt_set_proc_info*. Si el parámetro *pid* es inválido la rutina retorna *ESRCH*, en caso contrario retorna *OK*.

rt_show_task_monitor

```
void rt_show_task_monitor()
```

Muestra la consola del *Monitor de Tareas*. Tiene exactamente el mismo efecto que presionar F7 pero permite que se pueda realizar en forma programática.

rt_times

```
clock_t rt_times(struct tms* buf)
```

Permite obtener varios tiempos asociados al proceso. La rutina recibe como parámetro un puntero a una estructura *tms* en la que se retorna la siguiente información:

- Tiempo de ejecución de usuario de éste proceso.
- Tiempo de ejecución del sistema para éste proceso.
- Tiempo de ejecución de usuario acumulado por todos los procesos hijos.
- Tiempo de ejecución del sistema para todos los procesos hijos.

A diferencia de la rutina *times*, todos los tiempos son en milisegundos y tienen en cuenta la granularidad del reloj del sistema.

rt_validate_cfg

```
int rt_validate_cfg(rt_cfg_t* rt_cfg_ptr)
```

Permite validar los campos de la estructura apuntada por *rt_cfg_ptr*, que será utilizada más tarde para crear un proceso de tiempo real mediante la rutina *rt_fork*.

Puede retornar *OK* o uno de los siguientes errores: *EINVET*, *EINVFNS*, *EINVP*, *EINVMT* o *EINVOT*. Para más detalles sobre estos errores ver la sección Errores.

rt_wait_result

```
void rt_wait_result(pid_t pid)
```

El proceso llamador se bloquea en espera del resultado de la ejecución del proceso identificado por *pid*. Sólo aplicable a procesos con redundancia de N copias (NCR).

rt_waste_time

```
void rt_waste_time(long ms, BOOL opt)
```

Emula una ejecución equivalente en tiempo a *ms*. Si el flag *opt* esta prendido, la ejecución termina también al llegar la meta, cuando la función *rt_get_remainder* sea menor a 5.

4.2. Estructuras

rt_bkp_info_t

Esta estructura es utilizada por la función *rt_add_bkp* para agregar a un proceso una nueva versión en la estructura *rt_cfg_t*.

```
typedef struct rtbkpinfot
{
     char            name[16];
     proc_func_t     mandatory_function;
     proc_func_t     optional_function;
     clock_t         optional_time;
     clock_t         mandatory_time;
} rt_bkp_info_t;
```

En la tabla siguiente se describen los campos de la estructura:

Campo	Descripción
char name[16]	Nombre del proceso.
proc_func_t mandatory_function	Puntero a la función que implementa la parte mandatoria de la versión.

proc_func_t optional_function	Puntero a la funcion que implementa la parte opcional de la versión.
clock_t mandatory_time	Peor tiempo de ejecución de la parte mandatoria.
clock_t optional_time	Peor tiempo de ejecución de la parte opcional.

Tabla 5 - Campos de la estructura rt_bkp_info_t

sensor_reading

La estructura *sensor_reading* representa una lectura de sensor.

```
struct sensor_reading
{
     float value;
     float lbound;
     float ubound;
};
```

Se pensó a una lectura como un valor (value), con un límite inferior (lbound) y un límite superior (ubound), determinados éstos por la exactitud del sensor. Se toma entonces un rango, donde el valor leído pertenece al intervalo [lbound..ubound].

proc_info_t

La estructura *proc_info_t* permite obtener y modificar las propiedades de un proceso mediante las rutinas *rt_get_proc_info* y *rt_set_proc_info*.

```
typedef struct procinfot
{
     char      type;
     char      name[16];
     pid_t     pid;
     int       queue;
     int       nr;
     int       flags;
     pid_t     root_pid;
     clock_t   start_time;
     clock_t   next_period_time;
```

```
    int         missed_deadlines;
    int         periods;
    clock_t     user_time;
    clock_t     sys_time;
    clock_t     child_utime;
    clock_t     child_stime;
    clock_t     alarm;
    char        root_name[50];
    char        exec_type_desc[50];
    char        ic_type_desc[50];
    char        queue_name[50];
    char        type_desc[50];
    char        flags_desc[80];
    char        rt_flags_desc[105];
    rt_cfg_t    rt_cfg;
} proc_info_t;
```

A continuación se describen todos los campos de la estructura:

Campo	Descripción
type	Tipo de proceso: *PT_SYSTEM_DRIVER*: manejador de dispositivo *PT_SYSTEM_SERVER*: servidor (MM / FS) *PT_RT_PROCESS*: proceso de tiempo real *PT_ST_PROCESS*: proceso de tiempo compartido
name	Nombre del proceso.
pid	Identificador interno del proceso.
queue	Indica en que cola de procesos listos se encuentra actualmente el proceso: *SYSTEM_MODE_DRIVER_Q*, *SYSTEM_MODE_SERVER_Q*, *MANDATORY_MODE_Q* u *OPTIONAL_MODE_Q*. Si el proceso no está listo, es decir no está en ninguna cola, *queue* vale -1.
nr	Número de proceso, posición en el arreglo de procesos del *Núcleo*, para acceso rápido.
flags	Máscara de flags: *P_SLOT_FREE*: la estructura no está en uso. *SENDING*: el proceso está bloqueado intentando enviar un mensaje.

	RECEIVING: el proceso está bloqueado intentando recibir un mensaje. *SIG_PENDING*: el proceso tiene una señal pendiente de ser recibida. *P_STOP*: el proceso está por ser debuggeado. *P_SEM_BLOCKED*: el proceso está bloqueado esperando por un semáforo.
root_pid	PID del proceso raíz, proceso origen del pedido que se está procesando. Vale 0 si no se está procesando ningún pedido o si el proceso raíz es el mismo proceso.
start_time	Tiempo en el cual fue creado el proceso.
next_period_time	Tiempo en cual debe ejecutarse el próximo período. Solo aplica a *procesos de tiempo real periódicos*.
missed_deadlines	Cantidad de metas perdidas por el proceso.
periods	Cantidad de períodos ejecutados hasta el momento. Solo aplica a *procesos de tiempo real periódicos*.
user_time	Tiempo de ejecución del proceso.
sys_time	Tiempo de ejecución del sistema a pedido del proceso.
child_utime	Sumatoria de los tiempos de ejecución de todos los procesos hijos.
child_stime	Tiempo de ejecución del sistema a pedido de todos los procesos hijos.
alarm	Tiempo de la alarma. Si el proceso no se encuentra en espera de ninguna alarma el campo vale 0.
root_name	Nombre del proceso referenciado por el campo *root_pid* de esta estructura.
exec_type_desc	Descripción del tipo de ejecución referenciado por el campo *exec_type* de esta estructura.
ic_type_desc	Descripción del tipo de IC referenciada por el campo *ic_type* de esta estructura.
queue_name	Nombre de la cola referenciada por el campo *queue* de esta estructura.
type_desc	Descripción del tipo de proceso referenciado por el campo *type* de esta estructura.
flags_desc	Descripción del campo *flags* de esta estructura.
rt_flags_desc	Descripción del campo *flags* del campo *rt_cfg* de esta

	estructura.
rt_cfg	Almacena la información de tiempo real especificada en la creación del proceso. En la sección siguiente se describen detalladamente todos los campos de ésta estructura.

Tabla 6 – Campos de la estructura proc_info_t

rt_cfg_t

La estructura *rt_cfg_t* almacena todas las propiedades necesarias para la creación de de un proceso de tiempo real.

```
typedef struct rtcfgt
{
     proc_func_t    mandatory_function;
     proc_func_t    optional_function;
     char           exec_type;
     int            weight;
     clock_t        ready_time;
     clock_t        optional_time;
     clock_t        mandatory_time;
     clock_t        deadline;
     clock_t        period;
     long           max_periods;
     int            flags;
     clock_t        max_life_time;
     clock_t        fail_time;
     clock_t        text_crc_period;
     pid_t          supervisor_pid;
     int            bkp_count;
     rt_bkp_info_t  bkps[MAX_BKPS];
     int            cpy_count;

} rt_cfg_t;
```

A continuación se describen todos los campos de la estructura:

Campo	**Descripción**
mandatory_function	Puntero a la rutina que implementa la *parte mandatoria* del proceso. Si el proceso es *opcional*, el campo vale

	NULL.
optional_function	Puntero a la rutina que implementa la parte *opcional* del proceso. Si el proceso es *mandatorio*, el campo vale NULL.
exec_type	Los valores posibles son: *EXEC_TYPE_APERIODIC*: el proceso es de tiempo real y aperiódico. *EXEC_TYPE_PERIODIC*: el proceso es de tiempo real y periódico. *EXEC_TYPE_NULL*: el proceso no es de tiempo real.
weight	Peso, importancia del proceso. Puede valer desde 0 hasta *PW_MAX*, *PW_MAX* vale 4 por defecto.
ready_time	Tiempo a partir del cual debe empezar a correr el proceso.
optional_time	Tiempo máximo de duración de la parte opcional del proceso. Vale 0 si el proceso no tiene parte opcional.
mandatory_time	Tiempo máximo de duración de la parte mandatoria del proceso. Vale 0 si el proceso no tiene parte mandatoria.
deadline	Tiempo máximo de finalización del proceso. Solo aplica a *procesos de tiempo real aperiódicos.*
period	Longitud del período. Solo aplica a *procesos de tiempo real periódicos.*
max_periods	Cantidad máxima de períodos a ejecutar. Solo aplica a *procesos de tiempo real periódicos.*
max_life_time	Cantidad máxima de tiempo a ejecutar. Aplica a todos los *procesos de tiempo real.*
flags	Permite especificar las siguientes propiedades de un proceso: *P_OPTIONAL*: el proceso está ejecutando su parte opcional. *P_BACKUP*: el proceso es backup de otro proceso. *P_KILL_PROC_ON_MAN_DL*: el proceso debe ser eliminado por el planificador si la parte mandatoria no cumple su meta. *P_KILL_PROC_ON_OPT_DL*: el proceso debe ser eliminado por el planificador si la parte opcional no cumple su meta. *P_MSG_CRC_ENABLED*: habilita el chequeo de

	consistencia en los mensajes del proceso. *P_TEXT_CRC_ENABLED*: habilita el chequeo de consistencia en el texto o código del proceso. *P_SUPERVISOR_DISABLED*: inhabilita la creación del supervisor. *P_MSG_FAULT_PENDING*: marca la existencia de una falla transitoria pendiente de ser inyectada en el próximo mensaje enviado. *P_SIGHUP_WITH_SBKP*: indica que el proceso termina por una falla de diseño de software. *P_SIGHUP_WITH_TBKP*: indica que el proceso termina por una falla de transitoria. *P_RESULT_SAVED*: un resultado ya fue grabado.
fail_time	Tiempo absoluto en el que debe emularse una falla de software para este proceso.
text_crc_period	Longitud del período en que el *Núcleo* chequea por la consistencia del texto o código del proceso.
supervisor_pid	PID del proceso supervisor. Si no tiene un supervisor el campo es igual a cero.
bkp_count	Cantidad de backups o versiones del proceso para tolerar fallas de software.
bkps[MAX_BKPS]	Arreglo de estructuras *rt_bkp_info_t* conteniendo la descripción de cada una de las versiones.
cpy_count	Número copias (aparte de la principal) del proceso que debe dispararse en forma concurrente para detectar fallas transitorias mediante el mecanismo *NCR*.

Tabla 7 – Campos de la estructura rt_cfg_t

proc

En MINIX los procesos almacenan sus características (registros, apuntador a la pila, estado, mapa de memoria, límite de la pila, identificador del proceso, información de mensajes, etc.) en la estructura *proc*.

Para poder almacenar las nuevas características de los procesos se incorporaron los siguientes campos a la estructura *proc*:

```
typedef struct proc
```

```
{
    ... campos de MINIX ...

    struct cq_node*     p_callerq_head;
    struct cq_node*     p_callerq_tail;
    struct cq_node      p_my_cq_node;
    struct cq_node      p_ksig_cq_node;
    struct cq_node      p_int_cq_node;
    int                 p_queue;
    char                p_type;
    struct proc *       p_root_proc_ptr;
    struct cq_node*     p_cq_inheritance;
    struct proc *       p_sem_proc_ptr;
    int                 p_sem_blocked;
    clock_t             p_start_time;
    int                 p_missed_deadlines;
    clock_t             p_next_period_time;
    int                 p_periods;
    int                 p_pending_instances;
    rt_cfg_t            p_rt_cfg;
    message             p_ksig_msg;
    message             p_int_msg;
    int                 p_queue_prio;
    struct proc *       p_prevready_prio;
    struct proc *       p_nextready_prio;
    unsigned long       p_text_crc;
    clock_t             p_next_text_crc_time;
    clock_t             p_inst_exec_time;
    int                 p_result;

};
```

En la Tabla 8 detallan los campos incorporados a la estructura *proc.*

Campo	Descripción
struct cq_node p_callerq_head*	Apunta a la cabeza de la lista de procesos en espera.
struct cq_node p_callerq_tail*	Apunta a la cola de la lista de procesos en espera.

struct cq_node p_my_cq_node;	Nodo que utiliza este proceso para encolar los mensajes que envía y quedan en espera de ser recibidos por el proceso destino, formando parte de la cola apuntada por *p_callerq_head* y por *p_callerq_tail* de este proceso.
struct cq_node p_ksig_cq_node	Nodo que utiliza el proceso *HARDWARE* para encolar los mensajes que envía al proceso *MM* y quedan en espera de ser recibidos, formando parte de la cola apuntada por *p_callerq_head* y por *p_callerq_tail* de MM.
struct cq_node p_int_cq_node	Nodo que utiliza el proceso *HARDWARE* para encolar los mensajes que envía a los manejadores de dispositivo para el envío de una señal a éste proceso y que quedan en espera de ser recibidos. Este nodo formará parte de la cola apuntada por *p_callerq_head* y por *p_callerq_tail* del manejador de dispositivo correspondiente.
int p_queue	Indica en que cola de procesos listos se encuentra actualmente este proceso. Si el proceso no está listo el campo vale -1. Los valores posibles son: • *SYSTEM_MODE_DRIVER_Q*: cola para *Drivers* que deben ejecutar en *Modo Sistema*. • *SYSTEM_MODE_SERVER_Q*: cola para *Servidores* que deben ejecutar en *Modo Sistema*. • *MANDATORY_MODE_Q*: cola para procesos de cualquier tipo que deben ejecutar en *Modo Mandatorio*. • *OPTIONAL_MODE_Q*: cola para procesos de cualquier tipo que deben ejecutar en *Modo Opcional*.
char p_type	Indica el tipo de proceso. Los valores posibles son:

	• *PT_SYSTEM_DRIVER*: manejador de dispositivo. • *PT_SYSTEM_SERVER*: proceso servidor. • *PT_RT_SUPERVISOR:* proceso supervisor. • *PT_RT_PROCESS*: proceso de tiempo real. • *PT_ST_PROCESS*: proceso de tiempo compartido.
*struct proc * p_root_proc_ptr*	Es un puntero a otro proceso. El campo generalmente es nulo para los procesos de usuario (de tiempo real o compartido) pero para el caso de los manejadores de dispositivos y servidores apunta al proceso que solicitó el servicio y del cual es posible que herede la prioridad de ejecución. Este campo se actualiza mediante el mecanismo de pasaje de mensajes, implementado en las rutinas *mini_send* (*proc.c*) y *mini_rec* (*proc.c*).
struct cq_node p_cq_inheritance*	Indica si el proceso debe heredar la prioridad de alguno de los procesos en espera y de cual. Si es NULL no se debe heredar, si es diferente de NULL, entonces apunta al nodo correspondiente en la cola *p_callerq_head*.
*struct proc * p_sem_proc_ptr*	Proceso del cual se debe heredar la prioridad mediante el Protocolo del Techo de Prioridades (en inglés Priority Ceiling Protocol o simplemente PCP). Vale NULL si PCP indica que no se debe heredar.
int p_sem_blocked	Número de semáforo en que el proceso se encuentra bloqueado. El campo vale -1 si el proceso no está bloqueado en ningún semáforo.
clock_t p_start_time	Tiempo en el cual fue creado el proceso.
Int p_missed_deadlines	Cantidad de metas perdidas por el proceso
clock_t p_next_period_time	Tiempo en el cual debe ejecutarse el próximo período. Solo aplica a *procesos de*

	tiempo real periódicos.
int p_periods	Cantidad de períodos ejecutados por el proceso. Solo aplica a *procesos de tiempo real periódicos.*
p_pending_instances	Períodos pendientes de ser ejecutados en procesos que admiten pérdida de metas.
rt_cfg_t p_rt_cfg	Estructura que almacena la información de tiempo real especificada en la creación del proceso. Esta información indica si el proceso es periódico o aperiódico, tiempo de comienzo, meta, etc. En la sección "Referencia de la API" se describen detalladamente todos los campos de ésta estructura.
message p_ksig_msg	Mensaje utilizado por el *Núcleo* cuando se envía una señal al proceso.
message p_int_msg	Mensaje utilizado por el *Núcleo* cuando se envía una interrupción a un manejador de dispositivos. Solo aplica a procesos manejadores de dispositivo.
int p_queue_prio	Se usa solo si el proceso es mandatorio. Indica en que cola de prioridad se encuentra el proceso.
struct proc p_prevready_prio*	Se usa solo si el proceso es mandatorio. Es un enlace al proceso previo en la cola de prioridades apuntada por el campo *p_queue_prio.*
struct proc p_nextready_prio*	Se usa solo si el proceso es mandatorio. Es un enlace al proceso siguiente en la cola de prioridades apuntada por el campo *p_queue_prio.*
unsigned long p_text_crc	CRC del segmento TEXT. Si el mecanismo está habilitado, el valor del campo se calcula durante la creación del proceso y se valida en forma periódica de acuerdo al campo *text_crc_period* de la estructura *rt_cfg_t.*
clock_t p_next_text_crc_time	Tiempo el en cual se debe volver a validar el CRC del proceso.

p_inst_exec_time	Cantidad de tiempo ejecutado por la instancia corriente del proceso.
p_result	Resultado grabado por el proceso.

Tabla 8 - Campos incorporados a la estructura proc

También se eliminaron algunos campos ya obsoletos. A continuación se enumeran uno a uno todos los campos eliminados de la estructura *proc*:

Campo	**Descripción**
struct proc p_callerq* y *struct proc* p_sendlink*	Quedaron obsoletos al cambiar la estructura de la cola de procesos / mensajes en espera. En MINIX la cola es de procesos y en RT-MINIX de nodos que apuntan a procesos y mensajes. El campo *p_callerq* fue reemplazado por los campos *p_callerq_head* y *p_callerq_tail*. El campo *p_sendlink* servía para encadenar los procesos y fue reemplazado por los campos *cqn_next* y *cqn_previous* de la estructura *cq_node* que representa los nodos de la nueva lista.
sigset_t p_pending	Almacenaba las señales pendientes de ser enviadas al proceso cuando el proceso MM no estaba ocioso para procesar mensajes. Las señales se acumulaban en el campo *p_pending* en la rutina *cause_sig* (*system.c*) y se blanqueaban en la rutina *inform* (*system.c)* cuando las señales pendientes de ser enviadas al proceso quedaban en manos de MM. En RT-MINIX las señales pendientes de ser enviadas al proceso se almacenan en el campo *p_ksig_msg.SIG_MAP*.
unsigned p_pendcount	Mantenía un contador de señales pendientes de ser enviadas al proceso, era equivalente a la cantidad de bits prendidos en el campo *p_pending*. Adicionalmente RT-MINIX agrega el flag *P_SEM_BLOCKED* al campo *p_flags*, ya existente en la estructura *proc*. La presencia de este flag indica que el proceso está bloqueado en espera de un semáforo por lo cual no puede ejecutar. Por definición, en MINIX, el campo *p_flags* es distinto de 0 si el proceso no puede

	ejecutar y es igual a 0 si puede ejecutar.

Tabla 9 - Campos obsoletos de la structura proc

semaphore

La estructura *semaphore* (*rt_types.h*) almacena el estado de los semáforos.

```
struct semaphore
{
     char           s_name[T_SEMAPHORE_NAME+1];
     struct proc *  s_ceiling;
     struct proc *  s_proc;
     int            s_ref_count;
};
```

A continuación se describen los campos de la estructura:

Campo	Descripción
s_name	Nombre del semáforo.
s_ceiling	Puntero al proceso techo del semáforo,
s_proc	Puntero al proceso que tiene tomado el semáforo.
s_ref_count	Cantidad de procesos que están haciendo uso del semáforo. Cuando *s_ref_count* llega a cero el *Núcleo* destruye el semáforo.

Tabla 10 - Campos de la estructura semaphore

4.3. Señales

Señal	Evento	Manejador
SIGSFAU	Ocurrió una falla de diseño de software en el proceso supervisado.	*on_software_fault* (*supervisor.c*)
SIGTFAU	Ocurrió una falla transitoria en el proceso supervisado.	*on_transient_fault* (*supervisor.c*)
SIGSUCC	El proceso termino de ejecutar exitosamente.	*on_success* (*rtlib.c / supervisor.c*)
SIGEXEM	El *Núcleo* notifica al proceso que debe ejecutar su parte mandatoria.	*on_execute_mandatory (rtlib.c)*

SIGEXEO	El *Núcleo* notifica al proceso que debe ejecutar su parte opcional.	*on_execute_optional (rtlib.c)*

Tabla 11 - Señales de la biblioteca de tiempo real

4.4. Errores

Error	**Descripción**
EOUTGRAIN	La granularidad del reloj del sistema está fuera del rango permitido (50, 1193182).
EINVSH	El ID especificado no concuerda con ninguno de los semáforos activos en el sistema.
ESALRL	Se intentó tomar un semáforo ya tomado (Self-deadlock).
EINVET	El valor especificado para el campo *exec_type* no es correcto. Los valores posibles son: *EXEC_TYPE_APERIODIC* y *EXEC_TYPE_PERIODIC*.
EINVFNS	Los valores especificados para los campos *mandatory_function* y *optional_function* son ambos NULL.
EINVP	El valor especificado para el campo *period* es 0 o no fue especificado y el proceso es periódico.
EINVMT	Se especificó un valor para el campo *mandatory_time* y el proceso no tiene *parte mandatoria* porque el valor especificado para el campo *mandatory_function* es NULL. O el proceso es *periódico* y el valor que se especificó en el campo *mandatory_time* es mayor o igual al valor que se especificó en el campo *period*. O el proceso es *aperiódico* y el valor que se especificó en el campo *mandatory_time* es mayor o igual que la diferencia entre los valores de los campos *deadline* y *ready_time*.
EINVOT	Se especificó un valor para el campo *optional_time* y el proceso no tiene *parte opcional* porque el valor especificado para el campo *optional_function* es NULL.
EINVF	Los únicos flags válidos son: *P_KILL_PROC_ON_OPT_DL*,

	P_KILL_PROC_ON_MAN_DL, *P_MSG_CRC_ENABLED* y *P_TEXT_CRC_ENABLED*.
EINVDL	Se especificó un valor incorrecto para el campo *deadline*: el proceso es periódico y el valor especificado es mayor que el campo *period* o el valor especificado es menor que 0.
EINVW	Se especificó un valor incorrecto para el campo *weigth*. El rango válido es [0, PW_MAX].
EINVMP	Se especificó un valor menor a 0 para el campo *max_periods*.
EINVMLT	Se especificó un valor menor a 0 para el campo *max_life_time*.
EINVTCP	Se prendió el flag *P_TEXT_CRC_ENABLED* pero no se especificó ningún valor para el campo *text_crc_period*.
EBKPSEX	El número de backups o versiones agregadas con la función *rt_add_bkp* superó al valor de *MAX_BKPS*.
ECPYSEX	El número de copias especificadas en el campo *cpy_count* supera la constante *MAX_CPYS*.

Tabla 12 – Errores de la biblioteca de tiempo real

4.5. Comandos

clearlog

Limpia al Log, elimina toda la información del buffer. Esta función tiene el mismo efecto que presionar la tecla F8 en la consola 1.

Línea de comandos:

```
clearlog
```

Parámetros:

No tiene parámetros.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
System log cleared
```

doc

Deshabilita el mecanismo detección de sobrecargas y de multicolas.

Línea de comandos:

```
doc
```

Parámetros:

No tiene parámetros.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
Overload check disabled
```

eoc

Habilita el mecanismo detección de sobrecargas y de multicolas.

Línea de comandos:

```
eoc
```

Parámetros:

No tiene parámetros.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
Overload check enabled
```

getgrain

Muestra la granularidad actual del reloj del sistema. La granularidad por defecto es 60 ticks por segundo.

Línea de comandos:

```
getgrain
```

Parámetros:

No tiene parámetros.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
Current system clock grain: 60
```

getproc

Permite obtener información de un proceso a partir de su PID.

Línea de comandos:

```
getproc <pid>
```

Parámetros:

pid (ID del proceso)

Resultados:

Si PID es inválido muestra un mensaje con el siguiente formato:

```
Invalid PID
```

Si no el sistema muestra la siguiente información:

```
Name:                   Proc1
Type:                   RT Process
Pid:                    324
User time:              0
System time:            250
Children user time:     0
```

```
Children  system time:   0
Queue:
Position:               7
Flags:                  8 (RECEIVING)
Missed deadlines:       0
Start time:             0
Execution type:         Periodic
IC type:                Mandatory
Ready time:             1000
Mandatory time:         10
Optional time:          0
Weight:                 40000
RT flags:               0
Max. life time:         1100
Next period time:       0
Periods:                7
Period:                 1000
Max. periods:           -1
```

La información puede variar según el tipo de proceso.

hidetm

Oculta la consola del *Monitor de Tareas*. El resultado es equivalente a presionar la tecla F10 desde el Shell.

Línea de comandos:

`hidetm`

Parámetros:

No tiene parámetros.

Resultados:

No muestra ningún mensaje por pantalla, solo se oculta la consola.

injfault

Oculta la consola del *Monitor de Tareas*. El resultado es equivalente a presionar la tecla F10 desde el Shell.

Línea de comandos:

`injfault P=xxx F=T|M`

Parámetros:

Parámetro	Descripción
P=xxx	PID del proceso
F=[T|M]	Donde debe inyectarse la falla (T = segmento TEXT y M = mensaje)

Resultados:

No muestra ningún mensaje por pantalla.

rtfork

Permite crear procesos de tiempo real desde el shell. Los procesos son *dummy*, no es posible especificar las instrucciones a ejecutar, se ejecuta por defecto una rutina cuyo único propósito es consumir tiempo de CPU. El tiempo consumido por esta rutina corresponde a los tiempos de peor caso de ejecución de las partes *mandatoria* y *opcional*, indicados en los parámetros M y O.

Línea de comandos:

```
rtfork N=xxx E=[A|P] [R=nnn] [P=nnn] [D=nnn] [X=nnn]
[I=M|O|C] [O=nnn] [M=nnn] [W=n] [L=nnn] [CT=T] [CT=M]
[CP=nnn]
```

Parámetros:

Parámetro	Descripción
N=ccc	Nombre del proceso
E=[P|A]	Tipo de ejecución (P = Periódica y A = Aperiódica)
R=nnn	Tiempo de comienzo de ejecución en milisegundos.
M=nnn	Tiempo *Mandatorio* en milisegundos.
O=nnn	Tiempo *Opcional* en milisegundos.
D=nnn	*Deadline* en milisegundos.
P=nnn	Duración del período en milisegundos.
X=nnn	Cantidad máxima de períodos a ejecutar.
L=nnn	Tiempo máximo de vida.

Resultados:

Si ocurre un error se muestra alguno de los siguientes mensajes:

```
Invalid execution type.

Invalid period.

Invalid functions.

Invalid mandatory time.

Invalid optional time.

Process fork failed.
```

Si el proceso se creó en forma exitosa se muestra un mensaje con el siguiente formato:

```
Process successfully created with PID: 435
```

Ejemplos:

Para disparar un proceso opcional periódico llamado Proc1 con un tiempo de comienzo de ejecución de 1000 ms, un período de 4000 ms, y un tiempo de peor caso de ejecución de 900 ms y un peso de 3, debemos tipear la siguiente línea de comando:

```
rtfork N=Proc1 E=P R=1000 P=4000 I=O O=900 W=3
```

Para disparar un proceso mandatorio aperiódico llamado Proc2 con un tiempo de comienzo de ejecución de 5000 ms, una meta de 4000 ms, y un tiempo de peor caso de ejecución de 700 ms y un peso de 1, debemos tipear la siguiente línea de comando:

```
rtfork N=Proc2 E=A R=5000 D=5000 I=M M=700 W=1
```

rtkill

Permite eliminar todos los procesos de tiempo real del sistema.

Línea de comandos:

```
rtkill
```

Parámetros:

No tiene parámetros.

Resultados:

Si hay algún proceso de tiempo real en ejecución muestra en pantalla un mensaje de la forma:

```
Proc1 killed

Proc2 killed

Proc3 could not be killed

2 RT processes killed
```

Si no hay ningún proceso de tiempo real en ejecución muestra el mensaje:

```
There are no RT processes
```

savelog

Permite grabar el contenido del buffer de Log en un archivo.

Línea de comandos:

```
savelog <file>
```

Parámetros:

Parámetro	Descripción
file	Archivo donde debe grabarse el Log.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
System log saved
```

setgrain

Permite modificar la granularidad del reloj del sistema. La granularidad por defecto es 60 ticks por segundo.

Línea de comandos:

```
setgrain <newgrain>
```

Parámetros:

Parámetro	Descripción
newgrain	nueva granularidad

Resultados:

Muestra en pantalla un mensaje de la forma:

```
System clock grain changed
```

setsched

Permite modificar en forma dinámica el algoritmo de planificación para procesos *mandatorios* y *opcionales*.

Línea de comandos:

```
seteched M=<man_procs_sched> O=<opt_procs_sched>
```

Parámetros:

Parámetro	Descripción
man_procs_sched	Algoritmo de planificación para procesos *mandatorios.*
opt_procs_sched	Algoritmo de planificación para procesos *opcionales.*

Tabla 14 muestra los posibles valores de *man_procs_sched* y *opt_procs_sched*.

Nombre corto	Algoritmo de planificación
"SJF"	Shortest Job First
"BWF"	Biggest Weight First
"RM"	Rate Monotonic
"EDF"	Earliest Deadline First
"FIFO"	First In First Out
"SRTF"	Shortest remaining time first

Tabla 14 – Algoritmos de planificación

Resultados:

Si alguno de los parámetros es inválido muestra en pantalla alguno o los dos mensajes siguientes:

```
Invalid scheduling option for mandatory tasks
```

```
Invalid scheduling option for optional tasks
```

showtm

Muestra la consola del *Monitor de Tareas*. El resultado es equivalente a presionar la tecla F10 desde el shell.

Línea de comandos:

```
showtm
```

Parámetros:

No tiene parámetros.

Resultados:

No muestra ningún mensaje por pantalla, solo se muestra la consola.

startlog

Habilita el Log, el *Núcleo* comienza a registrar eventos.

Línea de comandos:

```
startlog
```

Parámetros:

No tiene parámetros.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
System log started
```

stoptlog

Deshabilita el Log, el *Núcleo* deja de registrar eventos.

Línea de comandos:

```
stoplog
```

Parámetros:

No tiene parámetros.

Resultados:

Muestra en pantalla un mensaje de la forma:

```
System log stopped
```

5. Detalles de implementación

La implementación de RT-MINIX fue un trabajo duro, se invirtieron cientos de horas de estudio, análisis y planificación previa a la codificación propiamente dicha. La codificación fue un trabajo gradual y en etapas, cada funcionalidad fue probada exhaustivamente y estabilizada antes de incorporar la siguiente.

Tanto para la etapa de análisis como para la de codificación fue fundamental la utilización del *Log del Núcleo* y del *Monitor de Tareas.*

Durante la etapa de desarrollo fueron también de gran utilidad los programas de prueba que vienen con MINIX: cuarenta ejecutables para probar las llamadas al sistema y 2 scripts para chequear los comandos del shell. Estas pruebas permitieron verificar en todo momento que la funcionalidad introducida no afectara las características originales de MINIX.

A continuación se describen en detalle las modificaciones y funcionalidades incorporadas en RT-MINIX.

5.1. Procesos de RT-MINIX

RT-MINIX conserva sin grandes cambios los procesos *manejadores de dispositivos* y *servidores* pero desdobla los *procesos de usuario* en dos nuevos tipos:

- *Procesos de tiempo compartido*: tienen las mismas características que los procesos de usuario de MINIX.
- *Procesos de tiempo real*: la principal característica de estos procesos es la presencia de restricciones de tiempo para la ejecución.

Los *procesos de tiempo real* pueden ser *periódicos* o *aperiódicos*. En los primeros es necesario especificar un *período* y opcionalmente una *meta* que puede o no coincidir con el mismo. En los procesos aperiódicos también se puede incluir una

meta. En ambos casos es posible especificar un tiempo de *comienzo* y el *peso* o *importancia*.

- *Mandatorios*: no poseen parte opcional. Al igual que los procesos de un sistema de tiempo real tradicional, deben ejecutar respetando sus metas a fin de que el sistema funcione correctamente.
- *Opcionales*: no poseen parte mandatoria. Tienen menor importancia, pueden omitirse o postergarse sin causar resultados inaceptables. Durante un período de sobrecarga el sistema podría omitir o retrasar la ejecución de algunos procesos opcionales con el fin de que los procesos mandatorios puedan completarse a tiempo.
- *Mixtos*: poseen un parte mandatoria y otra opcional. Mientras ejecutan la parte mandatoria tienen las características de los procesos mandatorios y mientras ejecutan la parte opcional tienen las características de lo procesos opcionales.

Un concepto relacionado al de *tipo de proceso* es el de *modo de ejecución*. En cada proceso, su *tipo* determina los posibles *modos de ejecución*. En RT-MINIX existen tres modos de ejecución:

- *Mandatorio*: pueden ejecutar en este modo los siguientes tipos de procesos: a) procesos mandatorios de tiempo real b) procesos mixtos de tiempo real ejecutando la parte mandatoria c) manejadores de dispositivos y servidores procesando pedidos de procesos con las características mencionadas en a) y b).
- *Opcional*: pueden ejecutar en este modo los siguientes tipos de procesos: a) procesos de tiempo compartido b) procesos de tiempo real opcionales c) procesos de tiempo real mixtos ejecutando la parte opcional d) manejadores de dispositivos y servidores procesando pedidos de procesos con las características mencionadas en a), b) y c).

- *Sistema*: en este modo sólo pueden ejecutar manejadores de dispositivos y servidores mientras procesan pedidos propios del sistema. Por ejemplo el proceso *clock* atendiendo una interrupción del reloj del sistema.
- *Supervisor*: en este modo sólo pueden ejecutar los procesos servidores.

Los procesos que se encuentran ejecutando en *modo sistema* tienen mayor prioridad que los procesos que se encuentran ejecutando en *modo mandatorio* y estos últimos tienen mayor prioridad que los que se encuentran ejecutando en *modo opcional*.

La incorporación de todas estas nuevas características provocó numerosos cambios en las estructuras existentes (especialmente la estructura *proc*) y la incorporación de otras tantas. En la sección "Referencia de la API" explican detalladamente estos cambios.

Para identificar el modo de ejecución de un proceso se utilizan las siguientes macros definidas en *const.h*:

- *RUNNING_AS_MANDATORY(p)*
- *RUNNING_AS_OPTIONAL(p)*
- *RUNNING_AS_SYSTEM(p)*
- *RUNNING_AS_SUPERVISOR(p)*

Todos los procesos pasan por diferentes estados durante su ciclo de vida. Para indicar en que estado se encuentran los procesos en cada instante se agrego el campo *int p_state* en la estructura *proc.c*. Los valores que puede tomar este campo estan definidos en *const.h* y se explican detalladamente en la sección "Referencia de la API".

RT-MINIX implementa la ejecución de procesos de tiempo real mediante envío de señales. Toda la comunicación desde el planificador hacia los procesos de tiempo real se realiza mediante señales:

Señal	**Descripción**

SIGEXEM	El *Núcleo* notifica al proceso que debe ejecutar su parte mandatoria.
SIGEXEO	El *Núcleo* notifica al proceso que debe ejecutar su parte opcional.
SIGSUCC	El *Núcleo* notifica al proceso que debe terminar.

La complejidad asociada al manejo de estas señales es totalmente transparente al programador. La única responsabilidad del programador es codificar una rutina para la parte mandatoria del proceso y si es necesario, otra para la parte opcional. El resto de las tareas se realizan en la biblioteca *rtlib.c*.

Para crear un proceso de tiempo real se debe llamar a la rutina *rt_fork* (*rtlib.c*). Esta rutina valida los parámetros y termina llamando a la rutina *fork_rt_proc* (*rtlib.c*). Si el proceso tiene una parte mandatoria, primero se crea al proceso supervisor con la rutina *fork_supervised_rt_proc* (*rtlib.c*) y el proceso supervisor es el encargado de llamar a la rutina *fork_rt_proc* (*rtlib.c*).

La rutina *fork_rt_proc* asocia las señales *SIGEXEM*, *SIGEXEO* y *SIGSUCC* a los manejadores *on_execute_mandatory*, *on_execute_optional* y *on_success* respectivamente y luego entra en un ciclo infinito en espera de las señales mencionadas.

Los manejadores *on_execute_mandatory* y *on_execute_optional* solo se encargan de llamar a las rutinas de la parte mandatoria y opcional respectivamente, especificadas mediante los campos *mandatory_function* y *optional_function* de la estructura *rt_cfg_t* (definida en *incluye/sys/rt_types.c*) en la llamada a la rutina *rt_fork*. El manejador *on_success* simplemente modifica la variable global *do_exit* para terminar con el ciclo principal del proceso y por lo tanto también con su ejecución.

El envío de todas las señales mencionadas es responsabilidad del *Núcleo*.

Cuando llega el tiempo de comienzo de ejecución (*ready_time*) de un proceso de tiempo real, el planificador le envía la señal *SIGEXEM* si se trata de un proceso mandatorio o mixto y la señal *SIGEXEO* si se trata de un proceso opcional. Esto

ocurre en la rutina *ready_time_reached* (*proc.c*) que es llamada desde *handle_event* (*proc.c*).

Cuando el proceso es *periódico*, el planificador sigue enviando en forma periódica la señal *SIGEXEM* si es el proceso es mandatorio o mixto y la señal *SIGEXEO* si es opcional. Esto ocurre en la rutina *next_period_time_reached* (*proc.c*) que es llamada también desde la rutina *handle_event* (*proc.c*).

La señal SIGSUCC debe enviarse al terminar la ejecución del manejador de la parte mandatoria si el proceso es mandatorio o al terminar la ejecución del manejador de la parte opcional si el proceso es opcional o mixto.

Para los procesos mixtos, la señal *SIGEXEO* debe enviarse al terminar la ejecución del manejador de la parte mandatoria.

Para implementar el envío de señales en los casos mencionados se modificó la rutina *do_sigreturn* (*system.c*). Esta rutina es llamada por el planificador cada vez que concluye la ejecución de un manejador de señal.

Como *do_sigreturn* (*system.c*) es llamada para todas las señales, es necesario poder reconocer desde la rutina cual es la señal y proceso involucrados. En el mensaje que recibe como parámetro la rutina tenemos disponible el proceso pero no la señal asociada al manejador que termina de ejecutarse.

La rutina *do_sigreturn* (*system.c*) es invocada a través de la cadena de llamadas: *sigreturn* (*lib/posix/signal.c*) → *do_sigreturn* (*mm/signal.c*) → *do_sigreturn* (*system.c*).

Cuando el *Núcleo* construye la pila en la rutina *do_sendsig* (*system.c*) para invocar el manejador de una señal, configura como dirección de retorno la rutina *sigreturn* (*lib/posix/signal.c*), garantizando que se invoque al terminar de ejecutarse el manejador. Como ésta rutina recibe como parámetro una estructura *sigcontext* (definida en *include/sys/sigcontext.h*), el *Núcleo* también debe copiarla a la pila en forma contigua.

Para que la señal involucrada pueda llegar como parámetro a la rutina *do_sigreturn* (*system.c*) se modificó la estructura *sigcontext*, agregando el campo *sc_signo*. Un puntero a esta estructura viaja en todos los mensajes involucrados en la cadena de llamadas mencionada previamente.

En MINIX, tanto las señales como las interrupciones son convertidas a mensajes de tipo *KSIG* y *HARD_INT* respectivamente. En ambos casos el origen del mensaje generado es el proceso *HARDWARE*. Los mensajes de señales tienen como destino el proceso *MM* y los de interrupciones al *manejador de dispositivo* correspondiente.

Estos mensajes tienen un manejo bastante particular, no se comportan como los mensajes convencionales, nunca son encolados en el proceso destino, solo son enviados en el momento que pueden ser recibidos. Si el proceso destino (*MM* o *manejador de dispositivo*) no está en condiciones de recibir el mensaje, se pospone el envío hasta que el proceso se encuentre ocioso. Cuando el planificador detecta que el proceso se encuentra en estado ocioso, copia el mensaje directamente al buffer de dicho proceso y lo echa a correr, todo sin hacer uso de la rutina *mini_send*.

El mecanismo descrito tiene dos problemas importantes:

- El tiempo de latencia en la atención de interrupciones y señales puede resultar inaceptable.
- Al no utilizar el mecanismo estándar de envío de mensajes mediante la rutina *mini_send* tampoco respeta el mecanismo de herencia de prioridades.

En primer lugar se permitió que las señales e interrupciones se envíen mediante la rutina *mini_send* y su manejo no difiera del de cualquier otro mensaje, incluyendo la posibilidad de encolarse en el proceso destino si fuera necesario.

Para el caso de las interrupciones se modificó la rutina *interrupt* (*proc.c*) y para el caso de las señales se modificó la rutina *cause_sig* (*system.c*). En ambos casos se envía un mensaje con el proceso *HARDWARE* como origen. En *interrupt* se

llama a la rutina *mini_send* y en *cause_sig* a la rutina *lock_mini_send* debido a que en el segundo caso la llamada no se realiza desde el archivo *proc.c*.

La estructura de procesos definida en MINIX impide:

- Que un proceso pueda almacenar temporalmente más de un mensaje enviado y no recibido. Por ello cada vez que se manda un mensaje y el receptor no está en condiciones de recibirlo este puede almacenarse en el campo *p_messbuf* (*struct proc*) del proceso emisor.
- Que un proceso pueda aparecer más de una vez en las colas de espera de otros procesos. Por ello la lista de procesos en espera es implementada en forma sencilla mediante los campos *p_callerq* y *p_sendlink* (*struct proc*). Esta implementación impide que un proceso pueda encolarse en varios procesos.

Como todos los mensajes de interrupciones y señales tienen como origen al proceso *HARDWARE* y se requiere el envío de múltiples señales e interrupciones en forma simultánea, los campos *p_callerq*, *p_sendlink* y *p_messbuf* se vuelven obsoletos. De todas formas, el campo *p_messbuf* se conserva porque sigue siendo necesario como buffer para la recepción de mensajes.

Para solucionar estos inconvenientes fue necesario crear una nueva estructura *cq_node* para encolar mensajes en espera. La estructura *cq_node* posee los campos inherentes a una estructura de tipo lista (*cqn_next y cqn_previous*) y punteros al mensaje y proceso emisor (*cqn_messbuf* y *cqn_proc* respectivamente). En la estructura *proc*, el campo *p_callerq* fue reemplazado por los campos *p_callerq_head* y *p_callerq_tail*. Ahora la lista de procesos en espera esta doblemente encadenada.

Como mencionamos previamente, queremos que se puedan enviar múltiples señales e interrupciones en forma simultánea. A primera vista pareciera que el proceso *HARDWARE* necesita almacenar un número no acotado de mensajes enviados y no recibidos, al igual que poder encolarse en espera en un número no acotado de procesos destino. Si esto fuese así, la implementación sería

extremadamente complicada, debido a que el *Núcleo* no puede tomar memoria en forma dinámica.

Afortunadamente el diseño de MINIX 2.0 impide enviar simultáneamente dos señales del mismo tipo al mismo proceso. Mientras se está ejecutando el manejador de una señal para un proceso, todas las señales que arriben de ese tipo para ese proceso son ignoradas. Tampoco es posible enviar simultáneamente dos interrupciones al mismo *manejador de dispositivo*, la segunda es ignorada.

Aprovechando estas propiedades es posible conocer el número máximo de mensajes que puede enviar el proceso *HARDWARE* en forma simultánea y, con un poco de ingenio, implementar el mecanismo sin necesidad de memoria dinámica.

La cantidad de mensajes de interrupciones que pueden enviarse en forma simultánea es como máximo la cantidad de *manejadores de dispositivo* del sistema. Por lo tanto:

- Los mensajes de interrupciones pueden ser almacenados en la estructura del *manejador de dispositivo* destino.
- Los nodos (*cq_node*) que necesita el proceso *HARDWARE* para encolarse si el *manejador de dispositivo* se encuentra ocupado y no puede atender la interrupción, pueden ser almacenados también en la estructura de los *manejadores de dispositivo* destino.
- La cantidad de mensajes de señales que pueden enviarse en forma simultánea es como máximo la cantidad de procesos del sistema. Por lo tanto:
- Los mensajes de señales pueden ser almacenados en la estructura del proceso destino de la señal.
- Los nodos (*cq_node*) que necesita el proceso *HARDWARE* para encolarse si el proceso *MM* se encuentra ocupado, pueden ser almacenados también en la estructura de los procesos destino.

Cualquier otro proceso que no sea el proceso *HARDWARE*, no puede enviar más de un mensaje por vez.

Por lo tanto los nodos (*cq_node*) que necesitan los procesos distintos de *HARDWARE* para encolarse en espera de otros procesos pueden almacenarse en la estructura del mismo proceso.

En el ejemplo de Figura 10 el proceso *MM* tiene encolados tres mensajes. El primer mensaje es enviado por el proceso *HARDWARE* por ello se utiliza como nodo el campo *p_hw_cq_node*, es una señal que debe enviarse al proceso A. El segundo mensaje es enviado por el proceso B, en un mensaje convencional, no es una señal, tampoco una interrupción, por ello se utiliza como nodo el campo *p_my_cq_node*. El tercer mensaje es enviado por el proceso *HARDWARE* por ello también se utiliza como nodo el campo *p_hw_cq_node*, es una señal que debe enviarse al proceso C.

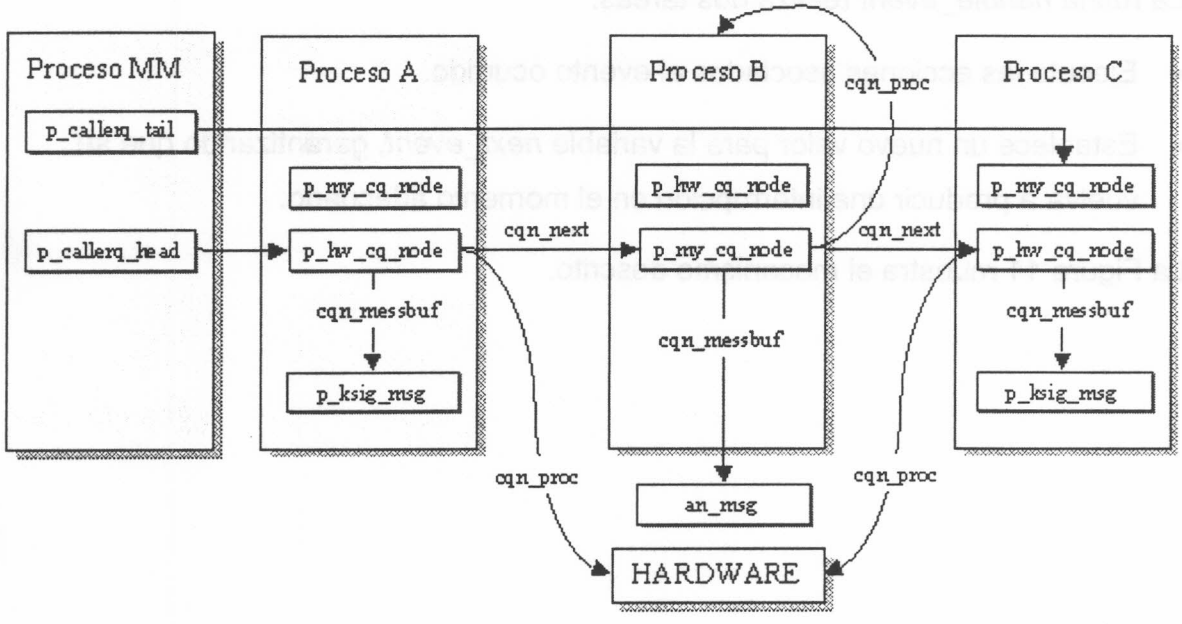

Figura 10 - Ejemplo de colas de mensajes

5.2. Planificador

El planificador de RT-MINIX es *manejado por tiempo* y basado en la siguiente estrategia:

- En cada interrupción de reloj se llama a la rutina *clock_handler* (*clock.c*).
- En la rutina *clock_handler* (*clock.c*) se compara el tiempo actual con la variable global *next_event*. Esta variable almacena el tiempo más cercano en el cual debe ocurrir un evento o realizarse una tarea de planificación.
- Cada vez que se alcanza el valor de *next_event* se envía un mensaje al manejador de dispositivo CLOCK, quién termina ejecutando la rutina *handle_event*.

La rutina *handle_event* realiza dos tareas:

- Ejecuta las acciones asociadas al evento ocurrido.
- Establece un nuevo valor para la variable *next_event*, garantizando que se vuelva a producir una interrupción en el momento adecuado.

La Figura 11 muestra el mecanismo descrito.

Figura 11 - Manejo de eventos disparados por tiempo

En las primeras versiones de RT-MINIX se enviaba un mensaje al manejador de dispositivo *CLOCK* siempre que había un proceso de usuario en la cola de listos. Mediante el uso de la variable global *next_event* (*proc.c*) se optimiza el número de mensajes enviados al *manejador del CLOCK*.

La rutina *handle_event* (*proc.c*) recorre la lista de procesos verificando para cada uno si llegó el tiempo de alguno de sus eventos. Para cada evento cuyo tiempo haya llegado, se llama la rutina asociada para que se realicen las acciones correspondientes.

Luego de procesarse todos los eventos, la rutina *handle_event* (*proc.c*) recorre nuevamente la lista de procesos para saber que nuevo valor establecer para la variable global *next_event* (*glo.h*).

El tiempo almacenado en *next_event* (*glo.h*) está asociado a alguno de los eventos enumerados a continuación. Cada evento tiene asociada una rutina que realiza las acciones correspondientes al mismo.

Evento	**Rutina**

Tiempo de listo alcanzado	*ready_time_reached (proc.c)*
Meta alcanzada	*deadline_time_reached (proc.c)*
Tiempo de proximo periodo alcanzado	*next_period_time_reached (proc.c)*
Tiempo de Falla alcanzado	*fail_time_reached (proc.c)*
Falla del CRC en el segmento TEXT	*text_crc_failed (proc.c)*
Tiempo máximo de vida alcanzado	*max_life_time_reached (proc.c)*
Tiempo de alarma alcanzado	*alarm_time_reached (proc.c)*

En las tablas siguientes se decriben las tareas que realiza cada rutina:

ready_time_reached *(proc.c)*
1. Cambia el estado del proceso a *PS_READY*, modificando el campo *p_state* de la estructura *proc* (*proc.h*)
2. Si el proceso es *periódico* actualiza la cantidad de períodos y el tiempo en que debe ejecutarse el próximo período. Incrementa en uno el campo *p_periods* y suma el tiempo equivalente a un período al campo *p_next_period_time*, ambos campos de la estructura *proc* (*proc.h*).
3. Setea el modo de ejecución del proceso. Si el proceso es *opcional* configura el flag *P_OPTIONAL* al campo *flags* de la estructura *rt_cfg_t* (*rt_types.h*).
4. Actualiza los contadores del sistema.
5. Mediante la rutina *cause_sig* se envía una señal *SIGEXEO* o *SIGEXEM* en función de que ejecute su parte *opcional* o *mandatoria* respectivamente.

deadline_time_reached *(proc.c)*
1. Si el proceso es *opcional*:
1.1. Si el proceso es *aperiódico* se lo elimina. Cambia el estado del proceso a *PS_TERMINATED* modificando el campo *p_state* de la estructura *proc* (*proc.h*) y envía mediante la rutina *cause_sig* una señal *SIGHUP* para terminar el proceso.

1.2. Si el proceso es *periódico* se tiene en cuenta la acción configurada en caso de pérdida de meta: o se lo elimina o se lo pasa a *modo mandatorio* y se lo deja terminar el período. Mediante el flag *P_KILL_PROC_ON_OPT_DL* del campo *flags* de la estructura *rt_cfg_t* (*rt_types.h*) se determina si hay que eliminar o no el proceso. Si hay que eliminarlo se procede de la misma forma que con las *aperiódicas*. Sino pasa el proceso a *modo mandatorio*, sacando el flag *P_OPTIONAL* del campo *flags* de la estructura *rt_cfg_t* (*rt_types.h*) y configurando el campo *deadline* de la misma estructura en *LONG_MAX*.

2. Si el proceso es *mandatorio*:

   2.1. Incrementa en uno la cantidad de metas perdidas. Suma uno al campo *p_missed_deadlines* de la estructura *proc* (*proc.h*).

   2.2. Se tiene en cuenta la acción configurada en caso de pérdida de meta: o se lo elimina o se lo deja seguir corriendo. Mediante el flag *P_KILL_PROC_ON_MAN_DL* del campo *flags* de la estructura *rt_cfg_t* (*rt_types.h*) se determina si hay que eliminar o no el proceso. Si hay que eliminarlo se cambia el estado del proceso a *PS_TERMINATED* modificando el campo *p_state* de la estructura *proc* (*proc.h*) y mediante la rutina *cause_sig* se envía una señal *SIGHUP*. Sino, se configura el campo *deadline* de la estructura *rt_cfg_t* (*rt_types.h*) en *LONG_MAX* para dejar que siga corriendo.

next_period_time_reached *(proc.c)*

1. Actualiza la cantidad de períodos y el tiempo en que debe ejecutarse el próximo período. Incrementa en uno el campo *p_periods* y suma el tiempo equivalente a un período al campo *p_next_period_time*, ambos campos de la estructura *proc* (*proc.h*).

2. Si el proceso está bloqueado y por lo tanto ya terminó de ejecutar el período anterior, se ejecuta un nuevo período. Cambia el estado del proceso a *PS_READY* modificando el campo *p_state* de la estructura *proc* (*proc.h*). Configura el modo de ejecución con que se debe empezar a ejecutar el próximo período, configurando el flag *P_OPTIONAL* del campo *flags* de la estructura *rt_cfg_t* (*rt_types.h*) si el proceso es *opcional*. Mediante la rutina *cause_sig* se envía una señal *SIGEXEO* o *SIGEXEM* en función de que ejecute su parte *opcional* o *mandatorio* respectivamente.
3. Si el proceso no está bloqueado y por lo tanto no terminó de ejecutar el período anterior, se registra el período como pendiente de ejecución. Se incrementa en uno el campo *p_pending_instances* de la estructura *proc* (*proc.h*).
4. En todos los casos se actualizan los contadores del sistema.

alarm_time_reached *(proc.c)*

1. Se envía una señal de alarma al proceso correspondiente, permitiendo que ejecute el manejador definido previamente.

max_life_time_reached *(proc.c)*

1. Cambia el estado del proceso a *PS_TERMINATED* modificando el campo *p_state* de la estructura *proc* (*proc.h*).
2. Mediante la rutina *cause_sig* se envía una señal *SIGHUP* para terminar el proceso.

text_crc_failed *(proc.c)*

1. Cambia el estado del proceso a *PS_TERMINATED* modificando el campo *p_state* de la estructura *proc* (*proc.h*).

2. Setea el flag *P_SIGHUP_WITH_TBKP* de la estructura *p_rt_cfg* en *proc*.
3. Mediante la rutina *cause_sig* se envía una señal *SIGHUP* para terminar el proceso.

fail_time_reached *(proc.c)*

1. Cambia el estado del proceso a *PS_TERMINATED* modificando el campo *p_state* de la estructura *proc* (*proc.h*).
2. Setea el flag de la estructura *p_rt_cfg* en *proc* a *P_SIGHUP_WITH_SBKP*
3. Mediante la rutina *cause_sig* se envía una señal *SIGHUP* para terminar el proceso.

MINIX posee las siguientes colas de procesos listos:

- *SYSTEM_MODE_DRIVER_Q*
- *SYSTEM_MODE_SERVER_Q*
- *SUPERVISOR_MODE_Q*
- *MANDATORY_MODE_Q*
- *OPTIONAL_MODE_Q*

El planificador mantiene punteros al comienzo y fin de las cinco colas mediante las variables globales:

```
struct proc *rdy_head[NQ];
struct proc *rdy_tail[NQ];
```

Como el total de colas es 5, *NQ* se define como 5.

Los procesos se encolan una u otra cola dependiendo del *tipo de proceso* y *modo de ejecución* en que se encuentren. La Tabla 15 muestra la cola asociada a cada combinación de *tipo de proceso* y *modo de ejecución*.

Tipo de Proceso	Modo de Ejecución	Cola
Manejador de Dispositivo	Sistema	SYSTEM_MODE_DRIVER_Q
	Mandatorio	MANDATORY_MODE_Q
	Opcional	OPTIONAL_MODE_Q
Servidor	Sistema	SYSTEM_MODE_SERVER_Q
	Mandatorio	MANDATORY_MODE _Q
	Opcional	OPTIONAL_MODE_Q
Usuario	Mandatorio	MANDATORY_MODE_Q
	Opcional	OPTIONAL_MODE_Q
Supervisor	Supervisor	SUPERVISOR_MODE_Q

Tabla 15 - Relación entre Tipos de Procesos, Modos de Ejecución y Colas de procesos

La rutina *get_target_q* (*proc.c*) implementa está lógica. Esta rutina toma como parámetro un puntero a un proceso y retorna la cola que le corresponde en función de su *tipo* y *modo de ejecución*.

Todas las colas mencionadas son listas de procesos doblemente encadenadas. El encadenamiento de los procesos se realiza mediante los campos *p_prevready* y *p_nextready* de la estructura *proc* (*proc.h*).

La administración de las colas se realiza mediante las rutinas *queue_remove*, *queue_add_at_tail* y *queue_add_at_head* definidas en *proc.c*.

Cada vez que un proceso es encolado se guarda el identificador de la cola en el campo *p_queue* de la estructura *proc* (*proc.h*). Este campo permite obtener en forma eficiente la cola en que se encuentra un proceso.

Las tareas de planificación se realizan en la rutina *pick_proc*. Esta rutina se llamada cada vez que el planificador tiene que elegir entre todos los procesos listos, cual será el que tome el procesador.

El proceso que está haciendo uso del procesador se apunta desde la variable global *proc_ptr* (*glo.h*).

La rutina *pick_proc* recorre las colas buscando procesos para ejecutar. Empieza por la cola *SYSTEM_MODE_DRIVER_Q,* luego con *SYSTEM_MODE_SERVER_Q,* luego con *SUPERVISOR_MODE_Q,* luego con *MANDATORY_MODE_Q* y finalmente con *OPTIONAL_MODE_Q.* Si no hay procesos en la colas, retorna el proceso *IDLE.*

En todo momento, el planificador tiene configurado un algoritmo para cada *modo de ejecución:*

- Para *modo sistema y supervisor* el algoritmo es *FIFO* y es fijo, no puede cambiarse.
- Para *modo mandatorio* se puede configurar cualquiera de los algoritmos de la Tabla 4, mediante la rutina *rt_set_man_scheduler* (*rtlib.c*) de la biblioteca de tiempo real. El planificador almacena la configuración actual en la variable global *rt_mandatory_procs_sched* (*glo.h*).
- Para *modo optional* se puede configurar también cualquiera de los algoritmos de la Tabla 4, mediante la rutina *rt_set_opt_scheduler* (*rtlib.c*) de la biblioteca de tiempo real. El planificador almacena la configuración actual en la variable global *rt_optional_procs_sched* (*glo.h*)

Por defecto se utilizan los algoritmos *BWF* y *FIFO* para procesos *mandatorios* y *opcionales* respectivamente.

Para elegir un proceso en las colas *MANDATORY_MODE_Q* y *OPTIONAL_MODE_Q*, el planificador debe recorrer en ambos casos toda la cola comparando las prioridades de los procesos. Cada vez que el planificador tiene que elegir entre dos procesos *p1* y *p2*, se ejecuta *sel_proc(p1, p2)*.

El algoritmo que implementa la rutina *sel_proc* (*proc.c*) es el siguiente:

Paso 1. Si *p1* es IDLE se retorna *p2*.

Paso 2. Si *p2* es IDLE se retorna *p1*.

Paso 3. Se calculan los procesos *p1_pp* y *p2_pp* de quienes heredan la prioridad *p1* y *p2*. El mecanismo de herencia se explica en detalle mas adelante.

Paso 4. Si *p1_pp* y *p2_pp* son el mismo proceso:

Paso 4.1. Si *p1* tiene tipo de proceso menor o igual que *p2* retorna *p1*.

Paso 4.2. Si *p2* tiene tipo de proceso menor que *p1* retorna *p2*.

Si *p1_pp* y *p2_pp* son el mismo proceso quiere decir que *p1* y *p2* pertenecen a la misma cadena de mensajes. Por ejemplo $p \rightarrow p1 \rightarrow p2$ (con *p1_pp* = *p2_pp* = *p*), es decir *p* pidió un servicio a *p1* y *p1* a *p2*. En este caso debo asignar mayor prioridad al proceso más a la derecha en la cadena, que es el de menor tipo de proceso. Los tipos de proceso se ordenan de la siguiente forma:

PT_SYSTEM_DRIVER < *PT_SYSTEM_SERVER*

PT_SYSTEM_SERVER < *PT_RT_PROCESS*

PT_RT_PROCESS < *PT_ST_PROCESS*

Este ordenamiento coincide con las capas de procesos RT-MINIX.

Paso 5. Si *p1_pp* tiene tipo de proceso menor que *p2_pp* retorna *p1*.

El proceso *p1_pp* pertenece a una capa más baja y más importante que el proceso *p2_pp*. Por lo tanto *p1* (que trabaja para *p1_pp*) tiene mayor prioridad que *p2* (que trabaja para *p2_pp*).

Paso 6. Si *p2_pp* tiene tipo de proceso menor que *p1_pp* retorna *p2*. El proceso *p2_pp* pertenece a una capa más baja y más importante que el proceso *p1_pp*. Por lo tanto *p2* (que trabaja para *p2_pp*) tiene mayor prioridad que *p1* (que trabaja para *p1_pp*).

Quiere decir que *p1_pp* y *p2_pp* tienen el mismo tipo.

Paso 7. Si *p1_pp* y *p2_pp* tienen tipo *PT_SYSTEM_DRIVER* o *PT_SYSTEM_SERVER* o *PT_ST_PROCESS*:

Paso 7.1. Si *p1* tiene menor o igual tipo que *p2*, retorna *p1*.

Paso 7.2. Si *p2* tiene menor tipo que *p1*, retorna *p2*.

Por ejemplo en una secuencia de pedidos como: *p1_pp* → s*1*→ *p1* y *p2_pp* → *p2*. Los procesos *p1_pp* y *p2_pp* son de tiempo compartido he hicieron pedidos a los servidores *s1* y *p2* respectivamente. El servidor *s1* hizo un pedido al manejador de dispositivo *p1*. En este ejemplo retorna *p1*, el de tipo menor.

Paso 8. Si *p1_pp* y *p2_pp* tienen tipo *PT_RT_PROCESS* debo tener en cuenta los *modos de ejecución es* en primer lugar y los algoritmos de planificación vigentes en segundo lugar:

Paso 8.1. Si *p1_pp* está en *modo mandatorio* y en *p2_pp modo opcional* retorno *p1*. Lo *mandatorio* es más prioritario que lo *opcional*.

Paso 8.2. Si *p1_pp* está en *modo opcional* y en *p2_pp modo mandatorio* retorno *p2*. Nuevamente lo *mandatorio* es más prioritario que lo *opcional*.

A esta altura se puede asegurar que *p1_pp* y *p2_pp* están ejecutando en el mismo *modo*.

Si ambos están ejecutando en *Modo Mandatorio* se debe aplicar el algoritmo vigente para *Modo Mandatorio*. Si ambos están ejecutando en *modo opcional* se debe aplicar el algoritmo vigente para *modo opcional*. Si para el algoritmo vigente tienen la misma prioridad se aplica *SCHED_ALG_BWF* y si para *SCHED_ALG_BWF* tienen la misma prioridad se aplica *SCHED_ALG_FIFO*.

Paso 8.3. Si el algoritmo es *SCHED_ALG_FIFO* retorna *p1*, el primero.

Paso 8.4. Si el algoritmo es *SCHED_ALG_RM* retorna el de menor período:

Paso 8.4.1. Si el período de *p1_pp* es menor que el de *p2_pp* retorna *p1*. Evalúa la expresión:

```
GET_PERIOD(p1_pp) < GET_PERIOD(p2_pp).
```

Paso 8.4.2. Si el período de *p1_pp* es mayor que el de *p2_pp* retorna *p2*. Evalúa la expresión:

```
GET_PERIOD(p1_pp) > GET_PERIOD(p2_pp).
```

Paso 8.4.3. Si los períodos de *p1_pp* y *p2_pp* son iguales aplico el algoritmo *SCHED_ALG_BWF.* Si los pesos de *p1_pp* y *p2_pp* también son iguales, *GET_WEIGTH(p1_pp) = GET_WEIGTH(p2_pp),* aplico el algoritmo *SCHED_ALG_FIFO* y retorno *p1.*

La macro *GET_PERIOD(p)* retorna la longitud del período si el proceso es *periódico* y la meta relativa al *tiempo de listo* si el proceso es *aperiódico. GET_PERIOD(p)* se define como:

```
IS_PERIODIC(p) ? p->p_rt_cfg.period:( GET_DEADLINE(p) -
GET_READY_TIME(p))
```

Paso 8.5. Si el algoritmo es *SCHED_ALG_DM* retorna la de menor meta relativa:

Paso 8.5.1. Si al meta relativa de *p1_pp* es menor que el de *p2_pp* retorna *p1.* Se evalúa la expresión:

```
GET_DEADLINE(p1_pp) < GET_DEADLINE(p2_pp).
```

Paso 8.5.2. Si la meta relativa de *p1_pp* es mayor que el de *p2* retorna *p2_pp.* Se evalúa la expresión:

```
GET_DEADLINE(p1_pp) > GET_DEADLINE(p2_pp).
```

Paso 8.5.3. Si las retas relativas de *p1_pp* y *p2_pp* son iguales aplico el algoritmo *SCHED_ALG_BWF.* Si los pesos de *p1_pp* y *p2_pp* también son iguales, *GET_WEIGTH(p1_pp) = GET_WEIGTH(p2_pp),* aplico el algoritmo *SCHED_ALG_FIFO* y retorno *p1.*

La macro *GET_DEADLINE(p)* retorna la meta relativa del proceso *p.* Si el proceso es *periódico*, la meta es relativa al comienzo del período. Si el proceso es *aperiódico*, la meta es relativa al tiempo de listo.
GET_DEADLINE(p) se define como:

```
p→p_rt_cfg.deadline==0 ? (IS_PERIODIC(p) ? p-
>p_rt_cfg.period: LONG_MAX): p→p_rt_cfg.deadline
```

Es decir, si no se especifica meta relativa (*p→p_rt_cfg.deadline==0*), retorna el máximo posible: *LONG_MAX* para procesos *aperiodic* y *GET_PERIOD(p)* para procesos *periódicos*. En otro caso retorna *p→p_rt_cfg.deadline*.

Paso 8.6. Si el algoritmo es *SCHED_ALG_EDF* retorna la de menor meta absoluta:

Paso 8.6.1. Si la meta absoluto de *p1_pp* es menor que el de *p2_pp* retorna *p1*. Evalúa la expresión:

```
GET_ABS_DEADLINE(p1_pp) < GET_ABS_DEADLINE(p2_pp).
```

Paso 8.6.2. Si la meta absoluta de *p1_pp* es mayor que el de *p2_pp* retorna *p2*. Evalúa la expresión:

```
GET_ABS_DEADLINE(p1_pp) > GET_ABS_DEADLINE(p2_pp).
```

Paso 8.6.3. Si las metas absolutas de *p1_pp* y *p2_pp* son iguales aplico el algoritmo *SCHED_ALG_BWF. S*i los pesos de *p1_pp* y *p2_pp* también son iguales, *GET_WEIGTH(p1_pp) = GET_WEIGTH(p2_pp),* aplico el algoritmo *SCHED_ALG_FIFO* y retorno *p1*.

La macro *GET_ABS_DEADLINE(p)* retorna la meta absoluta del proceso *p* y se define como:

```
IS_PERIODIC(p) ? (p->p_state==PS_STARTED ?
(GET_NEXT_PERIOD_TIME(p) +
GET_DEADLINE(p))(GET_NEXT_PERIOD_TIME(p) -
GET_PERIOD(p) + GET_DEADLINE(p)))(GET_START_TIME(p)
+ GET_DEADLINE(p))
```

Paso 8.7. Si el algoritmo es *SCHED_ALG_SJF* retorna el de menor tiempo de procesamiento, para procesos *mandatorios* se basa en el *tiempo mandatorio* y para procesos *opcionales* se basa en el *tiempo opcional*:

Paso 8.7.1. Si el tiempo de procesamiento de *p1_pp* es menor que el de *p2_pp* retorna *p1*. Para procesos *opcionales* se evalúa la expresión:

```
GET_OPTIONAL_TIME(p1_pp) < GET_OPTIONAL_TIME(p2_pp).
```

Para procesos *mandatorios* se evalúa la expresión:

```
GET_MANDATORY_TIME(p1_pp) <
GET_MANDATORY_TIME(p2_pp).
```

Paso 8.7.2. Si el tiempo de procesamiento restante de *p1_pp* es mayor que el de *p2_pp* retorna *p2*. Para los procesos *opcionales* se evalúa la expresión:

```
GET_OPTIONAL_TIME(p1_pp) > GET_OPTIONAL_TIME(p2_pp).
```

Para los procesos *mandatorios* se evalúa la expresión:

```
GET_MANDATORY_TIME(p1_pp) > GET_MANDATORY_TIME(p2_pp).
```

Paso 8.7.3. Si los tiempos de procesamiento de *p1* y *p2* son iguales aplico el algoritmo *SCHED_ALG_BWF. S*i los pesos de *p1* y *p2* también son iguales, *GET_WEIGTH(p1) = GET_WEIGTH(p2),* aplico el algoritmo *SCHED_ALG_FIFO* y retorno *p1*.

La macro *GET_MANDATORY_TIME(p)* se define como:

```
p->p_rt_cfg.mandatory_time
```

La macro *GET_OPTIONAL_TIME(p)* se define como:

```
p->p_rt_cfg.optional_time
```

Paso 8.8. Si el algoritmo es *SCHED_ALG_SRTF* retorna el de menor tiempo de procesamiento restante, para procesos *mandatorios* se basa en el *tiempo mandatorio* y para procesos *opcionales* se basa en el *tiempo opcional*:

Paso 8.8.1. Si el tiempo de procesamiento restante de *p1_pp* es menor que el de *p2_pp* retorna *p1*.

Para procesos *opcionales* se evalúa la expresión:

```
GET_REMAINING_OPT_TIME(p1_pp) <
GET_REMAINING_OPT_TIME(p1_pp)
```

Para procesos *mandatorios* se evalúa la expresión:

```
GET_REMAINING_MAN_TIME(p1_pp) <
GET_REMAINING_MAN_TIME(p1_pp)
```

Paso 8.8.2. Si el tiempo de procesamiento restante de *p1_pp* es mayor que el de *p2_pp* retorna *p2*.

Para procesos *Opcionales* se evalúa la expresión:

```
GET_REMAINING_OPT_TIME(p1_pp) >
GET_REMAINING_OPT_TIME(p1_pp)
```

Para procesos *Mandatorios* se evalúa la expresión:

```
GET_REMAINING_MAN_TIME(p1_pp) >
GET_REMAINING_MAN_TIME(p1_pp)
```

Paso 8.8.3. Si los tiempos de procesamiento restante de *p1* y *p2* son iguales aplico el algoritmo *SCHED_ALG_BWF. S*i los pesos de *p1* y *p2* también son iguales, *GET_WEIGTH(p1) = GET_WEIGTH(p2),* aplico el algoritmo *SCHED_ALG_FIFO* y retorno *p1*.

La macro *GET_REMAINING_MAN_TIME (p)* se define como:

```
GET_MANDATORY_TIME(p) - GET_USER_TIME(p) -
GET_SYS_TIME(p)
```

La macro *GET_REMAINING_OPT_TIME (p)* se define como:

```
GET_OPTIONAL_TIME(p) - GET_USER_TIME(p) -
GET_SYS_TIME(p)
```

Paso 8.9. Si el algoritmo es *SCHED_ALG_BWF* retorna el de menor peso:

Paso 8.9.1. Si el peso de *p1_pp* es mayor que el de *p2_pp* retorna *p1*.
Evalúa la expresión:

```
GET_WEIGTH(p1_pp) < GET_WEIGTH(p2_pp).
```

Paso 8.9.2. Si el peso de *p1_pp* es mayor que el de *p2_pp* retorna *p1*.
Evalúa la expresión:

```
GET_WEIGTH(p1_pp) > GET_WEIGTH(p2_pp).
```

Paso 8.9.3. Si los pesos de *p1_pp* y *p2_pp* son iguales, *GET_WEIGTH(p1_pp) = GET_WEIGTH(p2_pp),* aplico el algoritmo *SCHED_ALG_FIFO* y retorno *p1*.

La macro *GET_WEIGTH(p)* se define como:

p->p_rt_cfg.weigth

El algoritmo de herencia de prioridad permite que un proceso *p1* herede la prioridad de otro proceso *p2* si se cumple alguno de los puntos siguientes:

1. "*p2 se encuentra esperando por un semáforo S tomado por el proceso p1 y p2 tiene mayor prioridad que p1*"

   De esta forma aumentamos la prioridad del proceso *p1*, acelerando la liberación del semáforo *S* y permitiendo que *p2* pueda tomarlo lo antes posible.

   El campo *p_sem_proc_ptr* se agrego a la estrucutura *proc (proc.h)* para manejar estos casos. Este campo en un proceso A apunta a otro proceso B si B está esperando por un semáforo S, y S está tomado por A y B tiene mayor prioridad que A.

   Por lo tanto, el mejor lugar para actualizar el campo es cada vez que un proceso intenta tomar un semáforo en la función *lock_lock_semaphore* (*proc.c*). En esta función, cuando se verifica que el semáforo está tomado por otro proceso, se compara la prioridad de ambos mediante la función *sel_proc* (*proc.c*) y si es necesario, se actualiza el campo *p_sem_proc_ptr*.

También es necesario limpiar el campo *p_sem_proc_ptr* cada vez que se libera un semáforo en la función *release_semaphore* (*proc.c*).

2. "*un mensaje de p2 está esperando ser atendido por p1 y p2 tiene mayor prioridad que p1*"

   De esta forma aumentamos la prioridad del proceso *p1*, permitiendo que el mensaje de *p2* sea atendido lo antes posible.

   Para manejar estas situaciones se agrego el campo *p_cq_inheritance* en la estructura de proceso *proc (proc.h)*.

   El campo *p_cq_inheritance* (*proc.c*) puede apuntar a uno de los nodos de la lista apuntada por *p_callerq_head* y *p_callerq_tail* (*proc.c*) o ser NULL. Cuando apunta a alguno de los nodos de la lista indica que el proceso debe heredar la prioridad del proceso asociado a ese nodo.

   El campo *p_cq_inheritance* debe ser actualizado cuando se agregan y sacan nodos de la lista, cuando cambian los algoritmos de planificación y cuando se modifican las propiedades de tiempo real de los procesos:

   - En la rutina *mini_send* (*proc.c*) si el receptor no puede recibir el mensaje y la prioridad del emisor es mayor que la del receptor.
   - En la rutina *mini_rec* (*proc.c*) cuando el receptor procesa el mensaje asociado al nodo apuntado por *p_cq_inheritance*. En ésta rutina se llama a *remove_node_from_callerq* (*proc.c*) que a su vez llama a *update_cq_inheritance* (*proc.c*).
   - En las rutinas *do_rt_set_opt_scheduler* (*system.c*) y *do_rt_set_man_scheduler* (*system.c*) cuando se cambian los algoritmos de planificación y por lo tanto las prioridades de los procesos.
   - En la rutina *lock_set_proc_info* (*proc.c*) que permite modificar las propiedades de tiempo real de un proceso.

3. "*p1* está procesando un pedido que directa o indirectamente pertenece al proceso *p2*"

   Para determinar de donde vienen los pedidos agrego el campo *p_root_proc_ptr* en la estructura de proceso *proc (proc.h)*.

   Mientras un proceso de tipo *manejador de dispositivo* o *servidor* procesa un mensaje (un pedido), el campo *p_root_proc_ptr* apunta al proceso que originó el pedido.

   p_root_proc_ptr se actualiza mediante el mecanismo de envío y recepción de mensajes.

   MINIX cuenta con tres primitivas para enviar y recibir mensajes: *send*, *receive* y *send_rec*. Los procesos de usuario (tanto de tiempo compartido como de tiempo real) usan en general la primitiva *send_rec* y los procesos Servidores y Drivers las primitivas *send* y *receive*.

   Cada vez que se envía y / o recibe un mensaje es posible que el proceso destino deba actualizar el campo *p_root_proc_ptr*.

   Después de un largo análisis de todos los casos llegamos a la conclusión de que es necesario actualizarlo cuando la primitiva del emisor es *send_rec* y no es necesario (o no se debe) cuando la primitiva del emisor es *send* o *receive*. Intuitivamente, cuando un proceso pide un servicio (y por lo tanto *p_root_proc_ptr* se debe actualizar), éste espera una respuesta (y por lo tanto la primitiva que usa es *send_rec*).

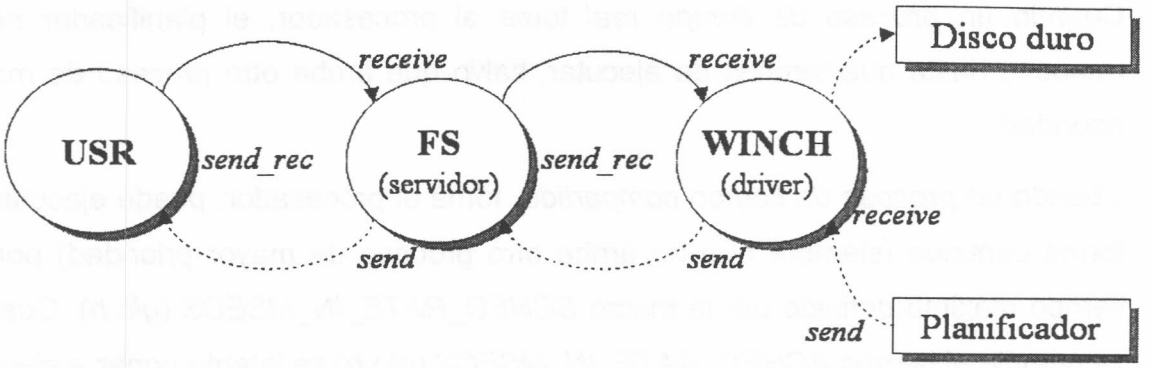

Figura 12 - Cadena de mensajes de una llamada al sistema

En la Figura 12 se muestra la típica secuencia de mensajes que se desencadena cuando un proceso de usuario realiza una llamada al sistema.

Las flechas sólidas representan envíos de mensajes con actualización de *p_root_proc_ptr* y las flechas punteadas representan envíos de mensajes sin actualización.

El proceso de usuario (*USR*) invoca a la llamada al sistema del *FS* usando la primitiva *send_rec*. Siguiendo la regla mencionada, *FS* debe actualizar *p_root_proc_ptr* y apuntarlo a *USR*. Luego *FS* transfiere el pedido al *manejador WINCH* de disco mediante otro *send_rec*. Nuevamente, siguiendo la regla, *WINCH* debe actualizar *p_root_proc_ptr* y apuntarlo a *USR* también. Cuando el disco termina su trabajo genera una interrupción que el planificador convierte en mensaje y envía a *WINCH* usando *send* porque no espera respuesta. En éste caso no debe actualizarse, al igual que cuando *WINCH* responde a *FS* y *FS* responde a *USR*.

Un caso especial sucede en las rutinas *mini_send* (*proc.c*) y *mini_rec* (*proc.c*) cuando el origen del pedido es el proceso *HARDWARE*, es decir cuando se trata de un pedido del sistema. En estos casos hay que limpiar el campo *p_root_proc_ptr* del *manejador de dispositivo* / *servidor* en cuestión, indicando que se trata de un pedido del sistema.

Cuando un *proceso de tiempo real* toma el procesador, el planificador no lo remueve hasta que termina de ejecutar, salvo que arribe otro proceso de mayor prioridad.

Cuando un *proceso de tiempo compartido* toma el procesador, puede ejecutar en forma continua (siempre que no arribe otro proceso de mayor prioridad) por un tiempo máximo definido por la macro *SCHED_RATE_IN_MSECS* (*glo.h*). Cuando se cumple el tiempo *SCHED_RATE_IN_MSECS* (*glo.h*) se intenta poner a ejecutar a otro proceso y solo sigue ejecutando el mismo si no hay ningún otro proceso listo.

Cuando el planificador no dispone de tiempo suficiente para ejecutar todas las tareas mandatorias dentro de sus restricciones de tiempo, se produce una sobrecarga o avería de tiempo.

La detección de sobrecargas se habilita y deshabilita con las llamadas al sistema *rt_enable_overload_check* (*rtlib.c*) y *rt_disable_overload_check* (*rtlib.c*) respectivamente. La implementación de estas llamadas se encuentra en las funciones *do_rt_enable_overload_check* (*system.c*) y *do_rt_inject_fault* (*system.c*) desde donde se configura la variable global *overload_check_enabled* (*glo.h*) en TRUE o FALSE según sea el caso. Por defecto la detección de sobrecarga esta deshabilitada.

Mientras el mecanismo de detección de sobrecarga está habilitado, la detección de sobrecarga se realiza llamando la rutina *overload_check* (*proc.c*). Esta rutina implementa un test de aceptación y actualiza la variable booleana *overload* (*glo.h*). A traves de la variable *overload* se informa al planificador la presencia de sobrecarga y por lo tanto también la necesidad de aplicar el mecanismo de multicolas.

La función *overload_check* (*proc.c*) se llama en dos oportunidades, cuando empieza a ejecutar y cuando termina un proceso mandatorio. En el primer caso se realiza desde la función *ready_time_reached* (*proc.c*) y en el segundo caso desde la función *do_xit* (*system.c*).

La función *overload_check* (*proc.c*), dependiendo de la presencia o no de sobrecarga, puede habilitar o dehabilitar el mecanismo de multicolas. El mecanismo de multicolas se debe activarse cuando empieza la sobrecarga y desactivarse cuando termina la misma.

El funcionamiento del mecanismo de multicolas se basa en el peso de los procesos. El peso de un proceso puede configurarse en su creación mediante el campo *weigth* de la estructura *rt_cfg_*t y es un entero que varia entre 0 y 4. El peso máximo (4) está determinado por la constante *PW_MAX* (*rt_types.h*). Cuanto más grande es el peso, más importante es el proceso.

El uso de multicolas incrementa el número de colas de procesos listos en *PW_MAX* * 3. A estas nuevas colas las llamamos *colas de prioridad*.

Los procesos se encolan en una u otra cola dependiendo del *tipo de proceso* y *modo de ejecución* en que se encuentren. La Tabla 16 muestra la cola asociada a cada combinación de *tipo de proceso* y *modo de ejecución*. Las *colas de prioridad* que se agregan al utilizar multicolas están marcadas en gris.

Tipo de proceso	Modo de Ejecución	Cola
Manejador de Dispositivo	Sistema	SYSTEM_MODE_DRIVER_Q
	Mandatorio	MANDATORY_MODE _Q
		MANDATORY_MODE_DRIVER_Q_0 ... MANDATORY_MODE_DRIVER_Q_31
	Opcional	OPTIONAL_MODE_Q
Servidor	Sistema	SYSTEM_MODE_SERVER_Q
	Mandatorio	MANDATORY_MODE _Q
		MANDATORY_MODE_SERVER_Q_0 ... MANDATORY_MODE_SERVER_Q_31
	Opcional	OPTIONAL_MODE_Q
		MANDATORY_MODE _Q

		MANDATORY_MODE _Q
	Opcional	MANDATORY_MODE_USER_Q_0
Supervisor	Supervisor	SUPERVISOR_MODE_Q
		MANDATORY_MODE_USER_Q_31

Tabla 16 – Esquema de Multicolas

Los procesos con modo de ejecución mandatorio se encolan simultáneamente en dos colas, en la cola *MANDATORY_MODE_Q* y también en la *cola de prioridad* que corresponda de acuerdo a su prioridad. En las funciones *queue_remove* (*proc.c*), *queue_add_at_tail* (*proc.c*) y *queue_add_at_head* (*proc.c*) mediante la macro *RUNNING_AS_MANDATORY* se determina si el proceso está ejecutando en modo mandatorio y si es así, se lo encola o desencola también de la cola de prioridad que corresponda de acuerdo a su peso.

Todas las colas mencionadas son listas de procesos doblemente encadenadas. El encadenamiento de los procesos se realiza mediante los siguientes campos de la estructura *proc* (*proc.h*):

- *p_prevready_prio* y *p_nextready_prio* para las colas de prioridad
- *p_prevready* y *p_nextready* para el resto de las colas.

Para conocer en que cola de prioridad ubicar un proceso en modo mandatorio el *Núcleo* utiliza la rutina *get_target_wq* (*proc.c*) que toma como parámetro un proceso y retorna la cola de prioridad que debe usarse. La lógica de la función *get_target_wq* se basa en la Tabla 16.

Cada vez que un proceso en modo mandatorio es encolado se guarda el identificador de la cola de prioridad en el campo *p_queue_prio* de la estructura *proc* (*proc.h*). Este campo permite obtener en forma eficiente la cola de prioridad en que se encuentra un proceso.

Cuando el planificador busca un proceso listo para correr a través de la rutina *pick_proc* (*proc.c*), elige primero a los procesos mandatorios de mayor peso,

garantizando que en caso de perderse deadlines mandatorios, estos corresponderán a los procesos de menor peso.

5.3. Sincronización de Procesos

El mecanismo de sincronización implementado en RT-MINIX es el de semáforos con soporte para techo de prioridades (en inglés *priority ceiling protocol*).

La variable global *rt_semaphores* (*glo.h*) es un arreglo de estructuras de tipo *semaphore* (*rt_types.h*) que almacena todos los semáforos del sistema. La estructuraa semaphore se describe detalladamente en la sección "Referencia de la API".

La cantidad máxima de semáforos (y tamaño del arreglo *rt_semaphores*) está determinada por la constante *NR_SEMAPHORES* (*rt_types.h*) definida como 30.

Los semáforos se referencian por nombre, permitiendo que dos o más procesos puedan compartir el mismo semáforo. El primero que lo referencia lo crea y el resto usa, aumentando un contador de referencias. El contador de referencias se almacena en el campo *s_ref_count* de la estructura *semaphore* y permite al *Núcleo* saber cuando puede destruir el semáforo porque ningún proceso lo está usando.

Para la implementacion de semáforos fue necesario modificar la estructura de proceso *proc*, incorporando el campo *p_sem_blocked* y el el flag *P_SEM_BLOCKED* al campo *p_flags*. Para más información ver la sección "Referencia de la API".

La secuencia de llamadas al sistema que involucra el uso de un semáforo es la siguiente:

```
...
int s = rt_create_semaphore("SEMAPHORE_NAME") (crea el
semáforo)
     ...
     rt_lock_semaphore(s) (toma el semáforo)
```

```
        ...
        Utilización del recurso protegido por el semáforo
        ...
    rt_release_semaphore(s) (libera el semáforo)
    ...
rt_delete_semaphore(s) (borra el semáforo)
...
```

La lógica de *rt_create_semaphore* es implementada en la rutina *lock_create_semaphore* (*proc.c*). Esta rutina permite crear un semáforo, recibe como parámetro el nombre y retorna un identificador (índice del semáforo en el arreglo *rt_semaphores*).

El algoritmo de la rutina *lock_create_semaphore* (*proc.c*) es el siguiente:

Paso 1. Busca un semáforo con el nombre especificado.

Se recorre el arreglo *rt_semaphores* (*glo.h*).

Paso 1.1. Si encuentra un semáforo ya creado con ese nombre:

Paso 1.1.1. Incrementa en uno las referencias y actualiza el techo del semáforo si es necesario.

Las referencias se aumentan incrementando en uno el campo *s_ref_count*.

El techo del semáforo (el campo *s_ceiling*) debe ser actualizado si la prioridad del proceso es mayor que la prioridad del proceso apuntado por *s_ceiling*.

Paso 1.1.2. Retorna el identificador del semáforo.

El identificador es el índice del semáforo en el arreglo *rt_semaphores* (*glo.h*).

Paso 1.2. Si no encuentra un semáforo ya creado con ese nombre:

Paso 1.2.1. Si no se llegó al máximo posible de semáforos:

Paso 1.2.1.1. Crea un semáforo nuevo con ese nombre.

Configura pone el nombre especificado.

Configura el valor del campo *s_ceiling* para que apunte al proceso.

Configura el valor del campo *s_ref_count* en uno.

Paso 1.2.1.2. Retorna el identificador del semáforo creado.

El identificador es el índice del semáforo en el arreglo rt_semaphores (*glo.h*).

Paso 1.2.2. Si se llegó al máximo posible de semáforos retorna -1.

La lógica de *rt_lock_semaphore* es implementada en la rutina *lock_lock_semaphore* (*proc.c*). Esta rutina permite intentar tomar un semáforo y recibe como parámetro su identificador, el índice en el arreglo *rt_semaphores*. Si no puede tomarlo encola el proceso en espera.

El algoritmo de la rutina *lock_lock_semaphore* (*proc.c*) es el siguiente:

Paso 1. Se busca el semáforo mediante su identificador.

El identificador se usa como índice del arreglo *rt_semaphores* (*glo.h*).

Paso 2. Si el semáforo no existe o se encuentra tomado por este proceso (self-deadlock) se retorna error.

Paso 3. Si el semáforo esta tomado por otro proceso, se bloquea a este en espera por el semáforo:

Para saber si un proceso está tomado y porque proceso, se verifica el campo *s_proc.*

Para bloquear el proceso:

a) Se configura el flag *P_SEM_BLOCKED* en el campo *p_flags* de la estructura del proceso.

b) Se configura el campo *p_sem_blocked* de la estructura del proceso con el identificador del semáforo.

c) Se saca al proceso de la cola de listos llamando a la rutina *unready* (*proc.c*).

Paso 4. Si el semáforo no esta tomado por otro proceso entonces:
Es decir el campo *s_proc* es *NIL_PROC*.

Paso 4.1. Se busca entre todos semáforos no tomados por este proceso, aquel cuyo techo sea el de mayor prioridad.
El techo de los semáforos está determinado por campo *s_ceiling* de la estructura *semaphore* (*rt_types.h*).

Paso 4.1.1. Si no se encuentra ningún semáforo tomado por otro proceso, se toma el semáforo.
Se apunta el campo *s_proc* del semáforo a este proceso.

Paso 4.1.2. Si se encuentra algún semáforo:

Paso 4.1.2.1. Si la prioridad del techo del semáforo encontrado es mayor o igual que la de este proceso, se bloquea a este proceso en espera del semáforo. Además, si la prioridad de este proceso es mayor que del proceso que está tomando el semáforo encontrado, el último hereda la prioridad de este proceso.
Se hace apuntar el campo *p_sem_proc_ptr* del proceso que está tomando el semáforo encontrado a este proceso.

Paso 4.1.2.2. Si la prioridad del techo del semáforo elegido es menor que la de este proceso, el proceso toma el semáforo.
Se hace apuntar el campo *s_proc* del semáforo a este proceso.

La lógica de *rt_release_semaphore* es implementada en la rutina *lock_release_semaphore* (*proc.c*). Esta rutina permite liberar un semáforo tomado y recibe como parámetro su identificador, el índice en el arreglo *rt_semaphores*.

El algoritmo de la rutina *lock_release_semaphore* (*proc.c*) es el siguiente:

Paso 1. Se busca el semáforo mediante su identificador.

El identificador se usa como índice del arreglo *rt_semaphores* (*glo.h*).

Paso 2. Si el identificador es incorrecto se retorna error.

El identificador no es correcto cuando: el semáforo no existe, no está tomado o está tomado por otro proceso.

Paso 3. Si el identificador es correcto:

Paso 3.1. Se libera el semáforo.

Para liberar el semáforo se limpia el campo *s_proc* de la estructura del semáforo.

También se limpia el campo *p_sem_proc_ptr* de este proceso. Este campo apunta al proceso de quien está heredando a acusa de un semáforo.

Paso 3.2. Si hay procesos en espera, se entrega el semáforo al de mayor prioridad.

Los procesos en espera por este semáforo son aquellos cuyo campo *p_sem_blocked* de la estructura *proc* (*proc.h*) tienen el identificador de este semáforo.

Para que el proceso tome el semáforo se realizan las siguientes acciones:
Se configura el valor -1 al campo *p_sem_blocked* de la estructura *proc* (*proc.h*).

a) Se saca el flag *P_SEM_BLOCKED* al campo *p_flags* de la estructura *proc* (*proc.h*).

b) Se llama a la función *ready* (*proc.c*).

La lógica de *rt_delete_semaphore* es implementada en la rutina *lock_delete_semaphore* (*proc.c*). Esta rutina permite destruir un semáforo y recibe como parámetro su identificador, el índice en el arreglo *rt_semaphores*.

El algoritmo de la rutina *lock_delete_semaphore* (*proc.c*) es el siguiente:

Paso 1. Se busca el semáforo mediante su identificador.

El identificador se usa como índice del arreglo *rt_semaphores* (*glo.h*).

Paso 2. Si el semáforo no existe se retorna error.

Paso 3. Si el semáforo está tomado por este proceso se libera.

Se llama a la rutina *release_semaphore* (*proc.c*).

Paso 4. Se decrementa en un la cantidad de referencias al semáforo.

Se decrementa en uno el campo *s_ref_count* de la estructura del semáforo.

Paso 5. Si la cantidad de referencias es cero, se destruye el semáforo.

Se limpian los campos *s_name*, *s_ceiling* y *s_proc* de la estructura del semáforo.

5.4. Tolerancia a Fallas

Uno de los mecanismos para tolerar fallas transitorias es traves del texto de los procesos.

Cuando se crea un proceso mediante la llamada al sistema FORK, el *Núcleo* calcula el CRC del segmento TEXT y lo almacena en un campo de la estructura del proceso. Periódicamente, a intervalos configurables de tiempo, el *Núcleo* recalcula el CRC y lo compara con el almacenado. El intervalo de tiempo se puede configurar por proceso. Si la validación del CRC no es satisfactoria significa que una falla ha sido detectada y por lo tanto el *Núcleo* envía la señal *SIFTFAU* al proceso supervisor.

El proceso supervisor es el responsable de las tareas de recuperación. Para el caso de fallas transitorias, la tarea de recuperación consiste en reejecutar el proceso. Si la reejecución vuelve a fallar debido a otra falla transitoria, el *Núcleo* envía otra señal *SIFTFAU* y se vuelve a reejecutar

Este mecanismo de detección de fallas a traves del texto debe habilitarse desde el programa de usuario prendiendo el bit *P_TEXT_CRC_ENABLED* del campo *flags*

de la estructura *rt_cfg_t* en el momento de configurar un proceso. Por defecto el mecanismo esta deshabilitado para todos los procesos.

También es necesario especificar, en el momento de configurar un proceso, el intervalo tiempo para los chequeos del texto del proceso. El mismo se hace a través del campo *text_crc_period* de la estructura *rt_cfg_t* del proceso.

El mecanismo de detección comienza cuando en algún momento de la creación de un proceso, el *Núcleo* llama a la función *alloc_segments(system.c)* para asignar memoria al proceso. En este lugar se hace lo siguiente:

Paso 1. Se verifica que el mecanismo de detección esté habilitado.

Paso 2. Si el mecanismo de detección esta habilitado se calcula el CRC con la información de texto del proceso con la rutina *compute_text_crc(proc.c)*.

Paso 3. El CRC calculado se guarda en el campo *p_text_crc* de la estructura del proceso.

Para poder determinar en que momento chequear la consistencia del texto, se utilizó el manejador de eventos disparados a través del tiempo. El manejador, representado por la rutina *handle_event* (*proc.c*), considera un nuevo evento que se dispara según el tiempo especificado en el campo *text_crc_period* de cada proceso. De dispararse el evento, la rutina que el manejador de eventos invoca hace lo siguiente:

Paso 1. Vuelve a calcular el CRC con la rutina *compute_text_crc(proc.c)*.

Paso 2. Compara el nuevo CRC calculado con el CRC que la estructura del proceso tiene almacenado en el campo *p_text_crc*.

Paso 3. De observarse alguna diferencia en la comparación de CRCs, se prende el bit *P_SIGHUP_WITH_TBKP* del campo *flags* de la estructura *rt_cfg_t* del proceso y se le manda al proceso la señal *SIGHUP*.

La tarea de recuperación de fallas transitorias esta a cargo del *proceso supervisor* que el *Núcleo* lanza junto a cada proceso de tiempo real.

Cuando al supervisor le llega una señal avisando que debe recuperar una falla transitoria vuelve a lanzar otro proceso igual para reemplazar al proceso fallido.

Si el nuevo proceso lanzado vuelve a fallar, el supervisor recibe otra señal y otro proceso igual es lanzado. La tarea de recuperación de fallas transitorias puede realizarse indefinidamente.

El *Núcleo* es el encargado de enviar la señal al supervisor. La señal es SIGTFAU y la envía en la función do_xit(system.c) cuando detecta que el proceso esta terminando por una falla transitoria.

Otro mecanismo para detección de fallas transitorias es el uso de CRC en mensajes. Permite detectar errores en el área de datos del *Núcleo* cuando los mensajes son almacenados temporalmente en espera de que el receptor pueda recibirlos. El mecanismo debe habilitarse en la creación del proceso y por defecto está deshabilitado.

En cada envío de mensaje, si alguno de los dos procesos involucrados tiene habilitado el mecanismo, el *Núcleo* adjunta su CRC para que pueda verificarse en la recepción. Si la validación del CRC en la recepción no es satisfactoria significa que el mensaje está corrupto y que la falla ha sido detectada.

La recuperación se basa en el reenvío de mensajes. El envío se intenta hasta un máximo de tres veces y luego se retorna error. Si se trata de una falla transitoria, probablemente se consiga el envío en el primer reintento.

El mecanismo debe habilitarse en la creación del proceso prendiendo el bit *P_MSG_CRC_ENABLED* del campo *flags* de la estructura *rt_cfg_t*. Por defecto el mecanismo esta deshabilitado.

El mecanismo de envío de mensajes se encuentra implementado en la función *mini_send* (*proc.c*). Cuando el proceso destino no se encuentra esperando el mensaje enviado y alguno de los procesos involucrados tiene habilitado el mecanismo, el *Núcleo* debe calcular el CRC y adjuntarlo al mensaje para que que pueda chequearse en la recepción. El CRC se calcula invocando la función

compute_message_crc (*proc.c*) y se almacena en el campo *cqn_messbuf_crc* de la estrucura *cq_node* junto al mensaje.

Por otro lado, el mecanismo de recepción de mensajes se encuentra implementado en la función *mini_rec* (*proc.c*). Cuando el mensaje recibido tiene adjunto su CRC (*cqn_messbuf_crc*!=0) el *Núcleo* debe recalcularlo y compararlo con el adjunto. Si validación del CRC no es satisfactoria significa que el mensaje está corrupto y que una falla ha sido detectada.

RT-MINIX cuenta con mecanismos de detección de fallas de diseño de software a *nivel hardware* y *aplicación*.

Los mecanismos de detección a nivel hardware son muy básicos: instrucciones ilegales, violación de dirección, desbordamiento aritmético, etc. En general las excepciones que sea capaz de generar el procesador.

A nivel aplicación, RT-MINIX provee manejo de *aserciones*. En éste mecanismo la tarea de detección es realizada por el programador, el *Núcleo* solo brinda el mecanismo de recuperación

Las versiones de un proceso que se utilizan en la recuperación se deben agregar a la estructura *rt_cfg_t* antes que esta misma sea pasada por parámetro a la función *rt_fork* para crear el proceso.

Para almacenar la información de versiones, se agregaron a la estructura *rt_cfg_t* los siguientes campos:

- *bkp_count:* Que indica la cantidad de versiones existentes.
- *bkps[MAX_BKPS]:* Es un array de elementos de tipo *rt_bkp_info_t* con longitud definida por la constante *MAX_BKPS(rt_types.h)*

Para facilitar la tarea de agregar versiones se creo la llamada al sistema *rt_add_bkp* (*rtlib.c*). Esta llamada al sistema recibe como parámetro una estructura *rt_cfg_t* junto con los campos de la estructura *rt_bkp_info_t*. La función va actualizando el campo *bkp_count* a medida que se agregan versiones.

En el momento de la recuperación, las versiones se lanzan de acuerdo al orden que tienen en el array *bkps.*

A las condiciones de error detectadas por el hardware como la división por cero, el intento de ejecutar una operación no permitida, etc, se les denominan *excepciones.*

El *Núcleo* de MINIX puede generar señales a partir de excepciones.

En la siguiente tabla se muestra el mapeo excepciones – señal que hace MINIX. La columna procesador hace referencia a la versión de procesador a partir de la cual se incorporó la excepción.

Excepción	**Señal generada**	**Procesador**
Divide error	SIGFPE	86
Debug exception	SIGTRAP	86
Nonmaskable interrupt	SIGBUS	86
Breakpoint	SIGEMT	86
Overflow	SIGFPE	86
Bounds check	SIGFPE	186
Invalid opcode	SIGILL	186
Coprocessor not vailable	SIGFPE	186
Double fault	SIGBUS	286
Copressor segment overrun	SIGSEGV	286
Invalid TSS	SIGSEGV	286
Segment not present	SIGSEGV	286
Stack exception	SIGSEGV	286
General protection	SIGSEGV	286
Page fault	SIGSEGV	386
NIL_PTR	SIGILL	-
Coprocessor error	SIGFPE	386

Tabla 17 - Mapeos de excepciones a señales

El administrador de memoria (MM) procesa a todas las señales de la misma forma. Cuando una señal esta dirigida a un proceso que no tiene asociado un manejador, lo termina. Esto incluye también a las señales generadas a partir de excepciones.

La detección usando assertions requiere del programador de la aplicación. Siempre que el programador considere necesario ejecutar otra versión del programa por una situación anómala en tiempo de ejecución debe invocar a la llamada al sistema *rt_assert*.

Esta llamada tiene los siguientes campos

- *exp:* expresión booleana.
- *msg:* mensaje de error asociado a la aserción.

Si la evaluación de *exp* da *FALSE*, el *Núcleo* muestra el texto *msg* en consola y aborta el proceso.

El *Núcleo* implementa la llamada a través de la función *do_rt_assertion_failed (system.c)* que prende el bit *P_SIGHUP_WITH_SBKP* del campo *flags* de la estructura *rt_cfg_t* del proceso y le manda al proceso la señal *SIGHUP*.

Igual que en el caso de fallas transitorias, la recuperación de fallas de diseño de software en un proceso, esta a cargo del supervisor del proceso.

Para recuperar una falla de diseño de software, es necesario que el supervisor ejecute otra versión del proceso. Esto se hace enviando desde el *Núcleo* una señal al supervisor para indicarle que debe realizar una la tarea de recuperación debido a una falla de diseño de software. La señal enviada es SIGSFAU y se hace en la función do_xit (system.c) cuando el *Núcleo* detecta que el proceso esta terminando por una aserción o excepción.

5.5. Proceso Supervisor

Por detecto, cada vez que se crea un proceso de tiempo real que posee parte mandatoria, también se crea un proceso supervisor. En la función de biblioteca *rt_fork* (*rtlib.c*) es donde se verifica si es necesario o no el supervisor. Si éste no es necesario, el proceso es creado invocando la función *fork_rt_proc*. Por otro lado, si el supervisor es requerido, se invoca la función *fork_supervised_rt_proc* definida en el archivo *supervisor.c*. En este archivo se encuentran todas las funciones y datos asociados a los procesos supervisores.

La lógica de la función *fork_supervised_rt_proc* (*supervisor.c*) es la siguiente:

Paso 1. Se crea el proceso supervisor.

El supervisor es creado enviando el mensaje *RT_SUPERVISOR_FORK* al proceso *MM*. La implementación del mensaje *RT_SUPERVISOR_FORK* similar a la del mensaje *FORK* con algunas pequeñas diferencias. Por ejemplo, el supervisor no necesita configurar las restricciones de tiempo propias de los procesos de tiempo real. Además se lo identifica como supervisor asignando la constante *PT_RT_SUPERVISOR* al campo *p_type* de la estructura *proc* (*proc.h*) y como nombre le asigna el nombre del proceso que supervisa más el sufijo "_S".

Paso 2. El PID del proceso supervisor creado se retorna al proceso padre que invocó la función *rt_fork*.

Esto cambia en parte la semántica de la función fork, pero es necesario para evitar que cuando el proceso de tiempo real falle y se dispare un backup, el proceso padre de por terminado todo. Además en el caso de manejarse copias, no existe un PID único, existen tantos PID como copias. Todo esto impone algo de cuidado en el uso de éste PID por parte del proceso padre.

Paso 3. El supervisor registra los manejadores de señales de fallas y fin de proceso.

Los manejadores *on_software_fault*, *on_transient_fault* y *on_success* se

asocian a las señales *SIGSFAU*, *SIGTFAU* y *SIGSUCC* respectivamente. Estas señales son enviadas por el *Núcleo* cuando ocurren fallas transitorias o de software y cuando el proceso termina de ejecutar en forma exitosa.

Paso 4. Se crea el proceso de tiempo real.

Es importante resaltar que el proceso es creado desde el supervisor y no desde el proceso padre. Para la creación del proceso de tiempo real se llama la función *fork_rt_proc*. El proceso creado recibe como parámetro el PID del proceso supervisor para disponer del vínculo entre los dos procesos a nivel *Núcleo*.

Paso 5. El supervisor queda en un bucle de espera.

A partir de éste momento el supervisor solo debe trabajar en respuesta a las señales que reciba del *Núcleo*, recibiendo notificaciones de fallas y fin de proceso. Dentro de alguno de los manejadores de señales, cuando sea oportuno, se actualizará la variable *do_exit* para terminar con el bucle y por lo tanto con la ejecución del supervisor.

La incorporación de un supervisor para cada proceso de tiempo real trae aparejado un incremento de carga en el sistema. Para reducir esta carga, se recomienda deshabilitar la creación del supervisor en los procesos donde no sea imprescindible la tolerancia a fallas. Esto se logra prendiendo el bit *P_SUPERVISOR_DISABLED* del campo *flags* de la estructura *rt_cfg_t* que se pasa como parámetro a la función *rt_fork* (*rtlib.c*).

El *Núcleo* debe notificar al *supervisor* los siguientes eventos: fallas de diseño de software, fallas transitorias y fin exitoso de proceso. Para estas notificaciones se utilizan las señales *SIGSFAU*, *SIGTFAU* y *SIGSUCC* respectivamente.

Los tres eventos se disparan con la finalización del proceso, en los dos primeros casos la finalización es con error y en el tercero la finalización es con éxito.

Por esta razón, el lugar natural para incorporar la notificación al supervisor es en la función *do_xit (system.c)*. Esta función es invocada por el *Núcleo* cada vez que un proceso termina, independientemente del motivo.

Para poder determinar desde la función *do_xit* el motivo por el cual a "muerto" el proceso y disparar la notificación adecuada al supervisor, se modificó la rutina *mm_exit* (*forkexit.c*) para que incluya en la llamada a *do_xit* (*system.c*) la señal que provocó la "muerte" o cero en el caso de finalización exitosa.

De esta forma:

- Si la señal recibida es *SIGHUP* y el campo *flags* de la estructura del proceso tiene el bit *P_SIGHUP_WITH_SBKP* prendido: se trata de una falla de diseño de software provocada por una aserción por lo que se envía al supervisor la señal *SIGSFAU*.
- Si la señal recibida es *SIGHUP* y el campo *flags* de estructura de proceso tiene el bit *P_SIGHUP_WITH_TBKP* prendido: se trata de una falla de transitoria por lo que se envía al supervisor la señal *SIGTFAU*.
- Si la señal recibida no es *SIGHUP* y no es 0: se trata de una falla de diseño de software provocada por una excepción del procesador por lo que se envía al supervisor la señal *SIGSFAU*.
- Si la señal recibida es 0: implica que la tarea terminó exitosamente y debe notificarse al *supervisor* mediante la señal *SIGSUCC*.

Las acciones de recuperación que lleva a cabo el supervisor implican la creación de un proceso de backup con las mismas características del proceso que falló y la continuación de las tareas que estaba realizando. Si el proceso que falló era periódico y ejecutó correctamente dos de cinco períodos, el backup debe ejecutar los tres restantes. Es decir, el proceso supervisor debe copiar la información de tiempo real del proceso "muerto" al backup. Por este motivo, la función *do_xit* (*system.c*) conserva intacta la información de tiempo real que reside en la

estructura *proc*, permitiendo que el manejador de la señal en el *supervisor* pueda accederla aún estando el proceso "muerto".

Antes de explicar el funcionamiento de los manejadores presentaremos el conjunto de variables que mantienen en todo momento el estado del supervisor y que son compartidas por los manejadores y el hilo principal de ejecución.

La variable *primary_name* (*supervisor.c*) permite al supervisor guardar una copia del nombre del proceso inicial en función de poder luego fabricar los nombres de los procesos backups concatenando un sufijo del tipo "_N", donde N es el número de backup.

La variable boolean *do_exit* (*supervisor.c*) es la condición del bucle en queda esperando el supervisor, manteniéndolo vivo hasta que se decida en uno de los manejadores que el trabajo a concluido y que el supervisor puede "morir". Esto se hace asignando TRUE a la variable *do_exit* desde *on_software_fault*, *on_transient_fault* o *on_success*.

La variable *exit_value* (*supervisor.c*) funciona en conjunto con la variable *do_exit* vista previamente. Esta última permite notificar desde un manejador, al hilo de ejecución principal del *supervisor*, que debe terminar de ejecutar. Para terminar de ejecutar debe invocae la llamada al sistema *exit*, pasando por parámetro el valor *exit_value*.

Las variables *ori_cpy_count*, *cpy_count*, *cpy_exit*, *res_count*, *res_votes* y *results* están todas vinculadas al mecanismo de copias para detección de fallas transitorias:

- *ori_cpy_count*: conserva el valor original del campo *cpy_count* de la estructura *rt_cfg_t*. Si no se utiliza el mecanismo de copias *cpy_count* vale 0.
- *cpy_count*: cantidad de copias generadas del proceso, en general debería ser igual a *ori_cpy_count*, pero si falla algún fork, no se genera error y *cpy_count* vale menos.

- *cpy_exit*: cantidad de copias que terminaron de ejecutar, independientemente de que hallan terminado con error o exitosamente. Cuando *cpy_exit* es igual a *cpy_count* quiere decir que todas las copias terminaron de ejecutar.
- *res_count*: cantidad de resultados grabados por las copias. Cada copia puede graba su resultado en cualquier momento de su vida, si no lo hace, porque falló o porque simplemente no lo hizo, su ejecución no cuenta.
- *results:* es un arreglo de enteros que conserva los resultados de las copias. No se almacenan resultados repetidos, si dos o más copias graban el mismo resultado, se incrementa la celda correspondiente en el arreglo *res_votes*.
- *res_votes*: es un arreglo de enteros que almacena los votos de cada resultado almacenado. Cada resultado tiene al menos un voto, el de la copia que lo grabó.

El proceso *supervisor* atiende las siguientes señales:

Evento	Señal	Manejador
Ocurrió una falla de diseño de software en el proceso supervisado.	*SIGSFAU*	*on_software_fault* (*supervisor.c*)
Ocurrió una falla transitoria en el proceso supervisado.	*SIGTFAU*	*on_transient_fault* (*supervisor.c*)
El proceso supervisado termino de ejecutar exitosamente.	*SIGSUCC*	*on_success* (*supervisor.c*)

Tabla 18 - Señales del supervisor

El manejador *on_software_fault* debe intentar lanzar una nueva versión del proceso que falló y se implementa de la siguiente forma:

Paso 1. Si es posible se obtiene el resultado del proceso / copia.

Para ello se invoca la función *on_cpy_exit* (*supervisor.c*). Esta función se invoca desde todos los manejadores del supervisor y realiza tareas propias de la terminación del proceso / copia, independientes del motivo como obtener el resultado del mismo.

Paso 2. Si quedan copias no se hace más nada.

La forma de saber si quedan copias es comparando las variables *cpy_exit* y *cpy_count*.

Paso 3. Si no quedan copias pero hay un resultado grabado, no se hace más nada.

Significa que el proceso / copia grabó el resultado ante de fallar.

Paso 4. Si no quedan copias y aún no se tiene ningún resultado, se debe lanzar un backup con sus respectivas copias:

Paso 4.1. Si no quedan backups se da por terminado el proceso con error.

```
exit_value = EXIT_FAILURE;
do_exit = TRUE;
```

Paso 4.2. Si quedan backups se lanza el siguiente.

Los backup se lanzan en el orden que se crearon. El fork del nuevo backup se configura tomando la información de tiempo real del proceso que falló.

El manejador *on_transient_fault* debe intentar lanzar una nueva copia del proceso que falló y se implementa de la siguiente forma:

Paso 1. Si es posible se obtiene el resultado del proceso / copia.

Para ello se invoca la función *on_cpy_exit* (*supervisor.c*). Esta función se invoca desde todos los manejadores del supervisor y realiza tareas propias de la terminación del proceso / copia, independientes del motivo como obtener el resultado del mismo.

Paso 2. Si quedan copias no se hace más nada.

La forma de saber si quedan copias es comparando las variables *cpy_exit* y *cpy_count*.

Paso 3. Si no quedan copias pero hay un resultado grabado, no se hace más nada.

Significa que el proceso / copia grabó el resultado ante de fallar.

Paso 4. Si no quedan copias y aún no se tiene ningún resultado, se debe lanzar una nueva copia:
El fork de la nueva copia se configura tomando la información de tiempo real del proceso que falló.

El manejador *on_success* solamente pone en *TRUE* a la variable *do_exit* para que el *supervisor* salga del loop y termine.

Paso 1. Si es posible se obtiene el resultado del proceso / copia. Para ello se invoca la función *on_cpy_exit* (*supervisor.c*). Esta función se invoca desde todos los manejadores del supervisor y realiza tareas propias de la terminación del proceso / copia, independientes del motivo como obtener el resultado del mismo.

Paso 2. Si no quedan copias se termina el supervisor en forma exitosa. La forma de saber si quedan copias es comparando las variables *cpy_exit* y *cpy_count* y la forma de terminar el supervisor en forma exitosa:

```
exit_value = EXIT_SUCCESS;
do_exit = TRUE;
```

Paso 3. Si quedan copias se las deja terminar.

5.6. Inyección de fallas

Para inyectar una falla transitoria desde un programa de usuario se debe utilizar la llamada al sistema *rt_inject_fault (rtlib.c)*. Esta función recibe dos parámetros, el PID del proceso perjudicado y lugar donde debe inyectarse la falla. El segundo parámetro soporta dos opciones:

- *TEXT_FAULT:* la falla debe inyectarse en el segmento TEXT del proceso.
- *MESSAGE_FAULT:* la falla debe inyectarse en el próximo mensaje enviado por el proceso.

La implementación de esta llamada al sistema se encuentra en la rutina *lock_inject_fault* (*proc.c*).

Las acciones que realiza la rutina son las siguientes:

Paso 1. Si la falla debe inyectarse en el próximo mensaje que se envíe, se pone una marca en el proceso y para luego en la rutina *mini_send (proc.c)* inyectar la falla mediante la rutina *change_mem (proc.c)*.

```
rp->p_rt_cfg.flags |= P_MSG_FAULT_PENDING
```

Paso 2. Si la falla debe inyectarse en el segmento TEXT, mediante la función *change_mem (proc.c)* inyecta la falla, modificando 10 bytes.

```
change_mem(ori_phys, size_phys, 10);
```

La simulación de fallas de diseño de software es senciblemente diferente. En la creación del proceso que debe fallar, se debe especificar mediante el campo *fail_time* de la estructura *rt_cfg_t* el momento en el tiempo en que debe simularse la falla.

El contenido del campo es tenido en cuenta por las funciónes *lock_handle_event* (*proc.c*) y *update_next_event* (*proc.c*) incorporándolo al manejador de eventos y forzando una interrupción de reloj en el tiempo especificado.

Cuando llega el tiempo de inyectar la falla, dentro de la función *lock_handle_event* (*proc.c*) se produce el efecto deseado con las siguientes instrucciones:

```
log_event0(PE_UNHANDLED_EXCEPTION, rp, NULL);
rp->p_state = PS_TERMINATED;
rp->p_rt_cfg.flags |= P_SIGHUP_WITH_SBKP;
cause_sig(proc_number(rp), SIGHUP);
```

Para poder determinar en que momento en que se debe producir la falla, se utilizó el manejador de eventos disparados a través del tiempo.

5.7. Biblioteca de Tiempo Real (BTR)

En la sección "Referencia de la API" se presenta el prototipo y su significado de todas las funciones de la BTR. La sección mencionada está orientada a los usuarios que no necesitan conocer el "*backstage*". Esta sección la complementa, aportando detalles de implementación y diseño.

Todas las funciones de la BTR están implementadas en el archivo *rtlib.c* y declaradas en el archivo *rt.h*.

La mayoría de las funciones de la BTR tienen el cien por ciento de su lógica implementada dentro del *Núcleo* y por lo tanto no se van a mencionar en esta sección.

La construcción y control de ejecución de los procesos de tiempo real es bastante compleja. Gran parte de la lógica asociada se encuentra dentro del *Núcleo*, pero no es despreciable la participación de la BTR.

Para crear un proceso de tiempo real se debe llamar la función *rt_fork* (*rtlib.c*), enviando el nombre del nuevo proceso y una estructura de tipo *rt_cfg_t* (*rt_types.h*) con toda la información de tiempo real necesaria para su construcción.

En función de facilitar la inicialización y validación de la estructura se incluyen dos funciones adicionales: *rt_init_cfg* y *rt_validate_cfg*. El uso de las mismas es opcional, pero recomendado, fundamentalmente el de *rt_init_cfg*.

La función *rt_init_cfg* permite cargar la estructura con todos los valores por defecto que se muestran en la tabla siguiente:

Campo	Valor por defecto	Observaciones
mandatory_function	NULL	Por defecto el proceso no tiene parte mandatoria y por lo tanto el campo que apunta a la función que implementa la parte mandatoria debe valer NULL.
optional_function	NULL	Por defecto el proceso no tiene parte opcional y por lo tanto el campo que apunta a la función que implementa la parte mandatoria debe valer NULL.
exec_type	0	Es obligatorio que el usuario configure este campo. Los valores posibles son EXEC_TYPE_APERIODIC (1) y EXEC_TYPE_PERIODIC (2). Se

		inicializa con el valor 0 porque no tiene valor por defecto.
ready_time	0	Por defecto el nuevo proceso va a empezar a ejecutar en forma inmediata.
optional_time	0	Por defecto el proceso no tiene parte opcional y por lo tanto la duración de la parte mandatoria es 0. Este valor tiene que estar en sintonía con el campo *optional_function*, esto se valida más tarde en las funciones *rt_validate_cfg* y *rt_fork*.
mandatory_time	0	Por defecto el nuevo proceso no tiene parte mandatoria y por lo tanto la duración de la parte mandatoria es 0. Este valor tiene que estar en sintonía con los campos *mandatory_function*, esto se valida más tarde en las funciones *rt_validate_cfg* y *rt_fork*.
deadline	0	El valor 0 (equivalente a infinito) indica que por defecto, el nuevo proceso no tiene meta.
period	0	El valor 0, indica que por defecto, el nuevo proceso no tiene período. Esto solo es posible si el proceso es a
max_periods	0	El valor 0 (equivalente a infinito) indica que por defecto, el proceso no tiene cantidad máxima de períodos.
max_life_time	0	El valor 0 (equivalente a infinito) indica que por defecto, el proceso no tiene tiempo máximo de vida.
flags	P_KILL_PROC_ON_OPT_DL | P_KILL_PROC_ON_MAN_DL	Por defecto el sistema debe eliminar el proceso si no cumple su meta, sin importar si está ejecutado la parte mandatoria o la parte opcional.

weigth	0	Por defecto el nuevo proceso tiene poca importancia o peso, 0 es el menor valor posible.

La función *rt_validate_cfg* realiza un chequeo de validez y consistencia de todos los campos de la estructura. Todos estos chequeos se realizan también al llamar la función *rt_fork* (*rtlib.c*). La ventaja de la función *rt_validate_cfg* es que retorna códigos de error diferentes para cada chequeo de consistencia, permitiendo identificar el problema. La función *rt_fork* (*rtlib.c*) retorna el PID del proceso creado y 0 en caso de error, no pudiendo identificarse el motivo.

A continuación se enumeran los errores que puede retornar la función *rt_validate_cfg* con la validación asociada:

- *EINVET*: tipo de proceso inválido, el proceso solo puede ser periódico o aperiódico.

```
rt_cfg_ptr→exec_type!=EXEC_TYPE_APERIODIC &&
rt_cfg_ptr→exec_type!=EXEC_TYPE_PERIODIC
```

- *EINVP*: período inválido, si el proceso es periódico, el período no puede ser nulo.

```
rt_cfg_ptr→exec_type==EXEC_TYPE_PERIODIC &&
rt_cfg_ptr→period==0
```

- *EINDL*: meta inválida, si el proceso es periódico, la meta no puede ser mayor que el período.

```
rt_cfg_ptr→exec_type==EXEC_TYPE_PERIODIC &&
rt_cfg_ptr→deadline > rt_cfg_ptr→period
```

- *EINDL*: meta inválida, la meta debe ser positivo o cero.

```
rt_cfg_ptr→deadline<0
```

- *EINVFNS*: funciones inválidas, el proceso debe implementar al menos una función.

```
rt_cfg_ptr→mandatory_function==NULL &&
rt_cfg_ptr→optional_function==NULL
```

- *EINVMT*: tiempo mandatorio inválido, si el proceso tiene parte mandatoria, la duración en tiempo de la parte mandatoria debe ser mayor que cero.

```
rt_cfg_ptr→mandatory_function==NULL &&
rt_cfg_ptr→mandatory_time>0
```

- *EINVOT*: tiempo opcional inválido, si el proceso tiene parte opcional, la duración en tiempo de la parte opcional debe ser mayor que cero.

```
rt_cfg_ptr→optional_function==NULL &&
rt_cfg_ptr→optional_time>0
```

- *EINVMT*: tiempo mandatorio inválido, si el proceso es mandatorio entonces:
  - Si el proceso es periódico: la parte mandatoria no puede ser mayor que el período.
  - Si el proceso es aperiódico: la parte mandatoria no puede ser mayor que la diferencia entre la meta y el comienzo.

```
rt_cfg_ptr→mandatory_function!=NULL &&
((rt_cfg_ptr→exec_type==EXEC_TYPE_PERIODIC &&
rt_cfg_ptr→mandatory_time>rt_cfg_ptr→period) ||
(rt_cfg_ptr→exec_type==EXEC_TYPE_APERIODIC &&
rt_cfg_ptr→mandatory_time>(rt_cfg_ptr→deadline-
rt_cfg_ptr→ready_time))
```

- *EINVW*: peso inválido, el peso o importancia del proceso no puede ser mayor que PW_MAX.

```
rt_cfg_ptr→weigth > PW_MAX
```

- *EINVMP*: cantidad máxima de períodos inválida, la cantidad máxima de períodos no puede ser negativa. 0 es equivalente a infinito.

```
rt_cfg_ptr→max_periods < 0
```

- *EINVMLT*: tiempo máximo de vida inválido, el tiempo máximo de vida no puede ser negativo. 0 es equivalente a infinito.

```
rt_cfg_ptr→max_life_time<0
```

- *EINVTCP*: flags inválidos, los flags que pueden configurarse son *P_KILL_PROC_ON_OPT_DL* y *P_KILL_PROC_ON_MAN_DL*. El resto son internos del sistema.

```
rt_cfg_ptr→flags & ~(P_KILL_PROC_ON_OPT_DL |
P_KILL_PROC_ON_MAN_) != 0
```

La sección "Referencia de la API" explica en detalle todos los campos de la estructura *rt_cfg_t* (*rt_types.h*).

La función *rt_fork* (*rtlib.c*) valida los campos de la estructura *rt_cfg_t* llamando internamente a la función *rt_validate_cfg* y luego llama a la función *fork_rt_proc*.

La función *rt_fork* (*rtlib.c*) valida los campos de la estructura *rt_cfg_t* (llamando internamente a la función *rt_validate_cfg*) y luego llama a la función *fork_rt_proc* (*rtlib.c*).

La función *fork_rt_proc* (*rtlib.c*) crea el nuevo proceso de tiempo real llamando la función *_syscall*, retorna al proceso padre el PID y luego asocia las señales *SIGEXEM, SIGEXEO* y *SIGSUCC* a las funciones *on_execute_mandatory* (*rtlib.c*), *on_execute_optional* (*rtlib.c*) y *on_success* (rtlib.c) respectivamente. Por último entra en un ciclo infinito hasta que la variable global *do_exit* valga *TRUE*, donde solo recibe señales mediante la función *pause*. Es decir, el proceso en si no hace nada, todo lo que hace lo hace en respuesta a las señales que recibe, implementando un diseño netamente orientado a eventos. Luego del ciclo se llama a la función *exit* para terminar el proceso.

Las funciones *on_execute_mandatory* y *on_execute_optional* son triviales, solo se encargan de obtener (mediante la función *rt_get_proc_info*) e invocar las funciones especificadas en la creación del proceso para la parte mandatoria y opcional. Sería posible configurar las señales para que invoquen directamente las funciones de la parte mandatoria y opcional, sin necesidad de las funciones intermedias mencionadas. El único inconveniente es que el prototipo de las funciones asociadas a señales recibe por parámetro el número de señal, obligando al usuario a incluir ese parámetro innecesario en sus propias funciones.

Cuando el *Núcleo* necesita ejecutar la parte mandatoria u opcional de un proceso, lo que hace es enviarle una señal *SIGEXEM* o *SIGEXEO* respectivamente. Estas señales son atrapadas desde la BTR, mediante las funciones *on_execute_mandatory* y *on_execute_optional* que finalmente invocan las funciones propias del usuario.

Como el proceso entra en un ciclo infinito hasta que la variable global *do_exit* valga *TRUE*, para que pueda terminar su ejecución es necesario mandarle una señal que actualice dicha variable. La señal *SIGSUCC* cumple el propósito mencionado y es enviada al proceso cuando ha terminado de ejecutar completamente.

La señal *SIGSUCC* es atrapada por la función *on_success* donde se actualiza la variable global *do_exit*, permitiendo que el proceso pueda salir del ciclo infinito.

Glosario

Durante la investigación y preparación de este trabajo, los términos detallados a continuación se han usado con los siguientes significados:

Término usado	**Término original**
Abrazo Mortal	Deadlock
Arreglo	Array
Artículo	Paper
Avance del texto en la pantalla	Scrolling
Avería	Failure
Bloque de Recuperación	Recovery Block
Ciclo	Loop
Confiablilidad	Dependability
Controlador de Dispositivos	Device Driver
Desbordamiento	Overflow
Elasticidad	Resiliency
Encuestado	Polled
en-línea	on-line
Enmascaramiento	Masking
Falla	Fault
fuera-de-línea	off-line
Identificador	Handle
Inicialización	Reset
Meta	Deadline
Núcleo	Kernel
Palanca de Juegos	Joystick
Período de poca actividad	Slack
Pila	Stack
Planificación	Scheduling
Predecibilidad	Predictability

Programación con N Versiones	N-Version programming
Prueba	Testing
Puerto de Juegos	Joystick port
punta-a-punta	end-to-end
Punto de recuperación	Checkpoint
Puntos de remoción	Preemption points
Recuperación hacia adelante	Forward recovery
Recuperación hacia atrás	Backward recovery
Recuperación hacia atrás	Rollback
Remoción	Preemption
Rendimiento	Performance
Rendimiento de Procesamiento	Throughput
Represión	Containment
Seguridad	Reliability
Señal	Tick
Sobrecarga	Overloading
Suma de comprobación	Checksum
Tiempo de respuesta	Response time
Trabajo adicional	Overhead