Algunas Modificaciones al Planificador de Minix acercándolo al Real Time

Francisco Matías Cuenca
Acuña
Nicolás Wolovick

Taller Sistemas de Tiempo Real
Departamento de Computación – U.B.A

Fa.M.A.F. – U.N.C.

1998

Indice

Planificación de procesos en Minix

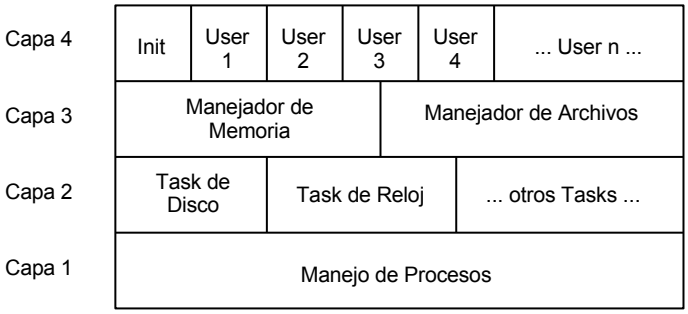
Nuestro trabajo se basa en la hipótesis de que es posible unificar las colas SERVER_Q y USER_Q para poder bajar de categoría los procesos File System (FS) y Memory Manager (MM), y así obtener mejores tiempos de respuesta por parte del sistema operativo, como un paso hacia el tiempo real.

En esta sección se tratará de hacer una breve introducción de todo aquello que esté involucrado con el manejo de estas colas de procesos en el código de Minix 2.0.0.

Estructura general

Minix es un sistema operativo dividido en 4 capas, donde cada una de ellas efectúa una tarea bien definida y puede comunicarse entre ellas o con la inferior y superior por un mecanismo uniforme de pasaje de mensajes.

La capa 1 es la encargada de virtualizar las interrupciones transformándolas en mensajes, manejar los



procesos, además de proveer el mecanismo de paso de mensajes. En la siguiente capa están los tasks o procesos I/O, también llamados device drivers que proporcionan la abstracción necesaria para el manejo de dispositivos. Luego se sitúa la 3ra capa donde están contenidos dos procesos fundamentales que son el memory manager –MM– y el file system –FS– los cuales proveen la máquina extendida que es capaz de manejar system calls de cierta complejidad. Finalmente en el último nivel están todos los procesos del árbol de procesos que surgen de INIT y este es el lugar de las aplicaciones finales como son los shells, editores, compiladores, etc.

Manejo de mensajes

La capa inferior del SO provee primitivas de comunicación para el pasaje de mensajes de longitud fija entre los procesos que conforman los niveles 2 y 3, y es la base de todo el flujo de control e IPC de Minix. Se tienen 2 primitivas básicas:

Primitiva	Semántica
Send	Efectúa un send bloqueante a un proceso nominado.
Receive	Recive de manera bloqueante desde un proceso nominado o no.

El mecanismo básico es el de rendezvous del proceso que quiere enviar y con el que espera recibir. Gracias a esto el SO evita implementar colas de mensajes y distintas semánticas de send/receive que escapan a los fines de un SO didáctico.

La función `mini_send()` que encontramos en `proc.c` es la que en definitiva implementa el envío de mensajes y se le proporcionan parámetros que designan el enviador, el receptor y el mensaje.

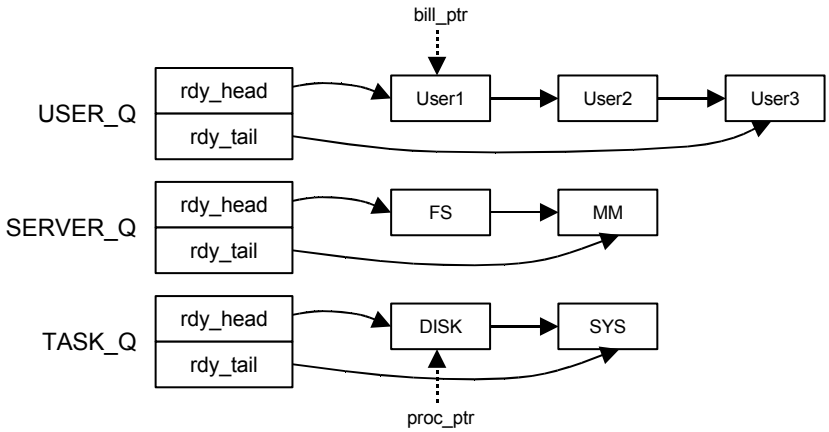
Luego de hacer chequeo de conformidad de parámetros para evitar algún tipo de deadlock e impedir la comunicación entre procesos user y task, se comprueba si el receptor está bloqueado esperando el mensaje, si es el caso y además el receptor no tiene indicadores de operaciones pendientes (`p_flags` debe estar limpio), se realiza un `ready()` que coloca el receptor en la cola de listos para ejecutarse. Si no se puede dar el rendezvous el proceso es quitado de la cola de listos (nuevamente si `p_flags` no indica nada extraño), marcado como “en espera de un receive”, y agregado a la cola de procesos que esperando enviar al receptor.

Para el caso de recepción, está la función `mini_rec()`. También se necesitan parámetros que indican quien es el que espera recibir, desde donde se espera hacerlo (con la posibilidad de recibir desde ANY) y por supuesto el buffer donde se depositará el mensaje.
Esta función no realiza comprobaciones de los parámetros, debido a que cualquier cosa que provenga del "exterior" del conjunto de las capas 1, 2 y 3 que puede ser considerado dañino ya fue filtrado por un send (en la capa 4 solamente se cuenta con mecanismos de sendrec provistos por la función `sys_call()`). Luego, si se produce el encuentro receptor-llamador, se copia el mensaje al buffer y nuevamente el llamador es puesto en juego sólo en caso de que sus banderas indiquen que se encuentra en algún estado limpio. En cambio para una recepción sin un llamador con mensaje disponible, se procede a sacar el receptor de la cola de listos y a marcarlo "en espera de mensaje".
Entre medio se realizan otras operaciones que incumben al manejo de interrupciones y señales, pero no resultan relevantes a nuestro problema.

Finalmente aclaremos que la puerta para los procesos de usuario a estas funciones está dada por `sys_call()` que también está en `proc.c`, y que básicamente se trata de un mecanismo por el cual el llamador envía un mensaje a un proceso para seguidamente esperar una respuesta de este.

Manejo de procesos listos

Todos los procesos que están listos para ejecutarse (de las capas 2, 3 y 4) se hallan en una estructura de datos que a continuación se muestra:



Es un esquema multicolas de 3 niveles, divididos por prioridades. TASK_Q, SERVER_Q y USER_Q es el orden de precedencia que el scheduler toma, es decir, para ejecutar algo de una cola, las de menor prioridad deben estar vacías.

Un proceso tiene el control de la CPU si y solo si `proc_ptr` lo apunta. Para los procesos de usuario existe un apuntador mas que es `bill_ptr` que sirve para manejar la preemtion por quantum de tiempo y realizar el trabajo de accounting. Notamos en el diagrama que no necesariamente `proc_ptr=bill_ptr` y cuando esto sucede, indica que un proceso server o task esta trabajando a favor del proceso de usuario señalado por `bill_ptr`.

Las funciones de manejo de estas colas están en `proc.c` y son básicamente `ready()`, `unready()`, `pick_proc()` y `sched()`.

Para agregar un proceso a la cola de listos se llama a `ready()`. Esta discrimina si es un task, un server o un user, para poder agregarlo a la cola correspondiente. Las 3 partes de la función son similares, excepto para la USER_Q donde se trabaja en modo pila en vez de cola.
No se obliga a elegir nuevo proceso actual con `pick_proc()` luego de haber agregado una tarea.

La función `unready()` procede de manera similar a `ready()` en la parte de discriminación de casos. Una vez conocido de que proceso se trata, se procede a comprobar si el que se quiere borrar está en el

frente de la cola. Si es el caso se lo quita y como tenemos un invariante que marca que `proc_ptr` estará siempre apuntando a uno de los 3 frentes de las colas, entonces habrá que elegir nuevo proceso corriente, por lo que se llama a `pick_proc()`. En caso de no encontrarse en el frente de la cola, se realiza una búsqueda lineal sobre la lista ligada para remover el elemento. No se elige nuevo proceso porque `proc_ptr` solo apunta a las cabezas de las colas.

Hay 2 funciones mas relacionadas con las colas de listos y son `pick_proc()` y `sched()`. La primera es la encargada de elegir el `proc_ptr` de acuerdo a las prioridades de las colas, es decir que si hay algo en TASK_Q se elige el elemento frontal de la cola, sino se trata de elegir el primero de SERVER_Q pero en caso de que esta se encuentre vacía se prueba con USER_Q. Finalmente cuando se comprobó que todas las colas están sin procesos, `proc_ptr` es asignado a una tarea por defecto llamada IDLE. Hay que notar que `bill_ptr` solo es reasignado cuando elegimos algo de USER_Q, esto hace que si un proceso de usuario efectúa la cadena P1-FS-DISK-FS-P1, el puntero `bill_ptr` no cambiará en todo este trayecto.

El SO provee mecanismos de preemption por quantum de tiempo que están relacionados con el task CLOCK y `bill_ptr`. Existe un timer de 100ms que al dispararse provoca que se compruebe si `bill_ptr` no ha cambiado desde la ultima vez. Si es el caso, indica que el proceso usuario ha corrido demasiado tiempo y se procede a la rotación de USER_Q y a la elección de un nuevo proceso (todo esto está en la función `sched()`) con lo cual tenemos un scheduler Round Robin para la cola de procesos usuarios. Las colas SERVER_Q y TASK_Q no son interrumpidas por quantum excedido debido a que se supone trabajan bien y devuelven el control rápidamente.

Finalmente debemos hacer notar que en la función interrupt(), también de `proc.c`, hay cambios de estado en las colas y en `proc_ptr`. Esto se produce cuando hay un proceso task bloqueado esperando un mensaje de hard y llega la interrupción correspondiente. Dada esta situación el proceso es encolado en TASK_Q sin llamar a `ready()`, además de reasignarse `proc_ptr` sin llamar a `pick_proc()` en caso de que la cola estuviese vacía. Se puede ver como un `ready()` con un `pick_proc()` obligatorio. Comentarios en el código indican que estos son reemplazos inline para aumentar la velocidad de procesamiento de interrupciones disminuyendo así su latencia.

Un ejemplo

Presentamos ahora una pequeña corrida de dos procesos usuarios (P1 y P2), para mostrar como interactúan las diferentes partes antes mencionadas. P1 efectúa un system call para obtener un sector del floppy y el otro solo hace cálculos enteros.

En el momento inicial tenemos P1 y P2 en USER_Q, mientras que las otras colas están vacías. El atributo negrita muestra el proceso apuntado por `proc_ptr` y subrayado el apuntado por `bill_ptr`.

Listos	
USER_Q	**P1** P2
SERVER_Q	
TASK_Q	

Bloqueados
FS recv(ANY)
FLOPPY recv(FS)

Cuando P1 efectúa el system call se produce dentro del kernel un `mini_send()` seguido de un `mini_rec()`. Al ejecutarse el mini_send(P1,FS,m), se produce en rendezvous de P1 y FS quedando en el siguiente estado

Listos	
USER_Q	**P1** P2
SERVER_Q	FS
TASK_Q	

Bloqueados
FLOPPY recv(FS)

Notamos que no hay elección de nuevo proceso porque `ready()` no lo implica. Ahora FS tiene el mensaje copiado en su cola y P1 su bandera SENDING limpia. Se sigue dentro del kernel ejecutando la función `sys_call()` y ahora es el turno de completar el sendrec con un llamado a mini_rec(P1,FS,m). Como el FS no esta bloqueado esperando poder enviar un mensaje a P1, el encuentro no se produce, y P1 es quitado de la cola de listos.

Listos	**Bloqueados**

USER_Q	P2
SERVER_Q	**FS**
TASK_Q	

P1 recv(FS) FLOPPY recv(FS)

El `unready()` del proceso al tope de la cola produce un `pick_proc()` que resulta en la elección del proceso de la cola server (aquí están las prioridades de las colas). El FS seguirá ejecutándose hasta que efectúa un send al task que maneja la disquetera, produciéndose el encuentro y estableciendo la situación siguiente

Listos

USER_Q	P2
SERVER_Q	**FS**
TASK_Q	FLOPPY

Bloqueados

P1 recv(FS)

Inmediatamente después FS se bloquea esperando respuesta de FLOPPY por lo que nuevamente se elige proceso y FLOPPY sale favorecido por estar en la cola de mayor prioridad.

Listos

USER_Q	P2
SERVER_Q	
TASK_Q	**FLOPPY**

Bloqueados

P1 recv(FS) FS recv(FLOPPY)

La task envía comandos a la disquetera y se bloquea esperando una respuesta del hard que vendrá en forma de mensaje de HARDWARE encapsulando una interrupción del dispositivo. Como SERVER_Q y TASK_Q están vacías, el llamado a `pick_proc()` que hace `unready()` da como resultado que P2 obtenga el control de la CPU. Notamos que durante todo este tiempo `bill_ptr` apuntó a P1

Listos

USER_Q	**P2**
SERVER_Q	
TASK_Q	

Bloqueados

P1 recv(FS) FS recv(FLOPPY) FLOPPY recv(HARDWARE)

Ahora `proc_ptr` y `bill_ptr` apuntan al proceso CPUBound que continúa ejecutándose sin interactuar con el SO o dispositivos externos manejados por este. En cierto momento se recibe la interrupción de hard, FLOPPY es despertado, toma el control de la CPU (interrupt() hace esto con el código inline), genera el mensaje para el FS y se lo envía, desbloqueándolo. Seguidamente el bucle infinito de FLOPPY que atiende los pedidos se bloquea esperando un nuevo mensaje por parte del FS.

Listos

USER_Q	P2
SERVER_Q	**FS**
TASK_Q	

Bloqueados

P1 recv(FS) FLOPPY recv(FS)

Se produce una cascada similar con FS y P1, con lo que P1 es puesto a ready (notar el encolado por adelante) y FS queda bloqueado esperando un nuevo mensaje por parte de ANY. El unready(FS) genera un `pick_proc()` y P2 es robado de su tiempo para atender al recientemente despertado P1.

Listos

USER_Q	**P1**, P2
SERVER_Q	
TASK_Q	

Bloqueados

FS recv(ANY)

Concluimos aquí el análisis y notamos algunos puntos:

- Un `ready()` no implica elección de nuevo proceso, por lo que no hay preemtion por "nuevo proceso ingresado".
- Esto no es cierto para el `ready()` inline que se hace en interrupt() que si desaloja a cualquier proceso de SERVER_Q y USER_Q.

- El encolado por delante de USER_Q está íntimamente relacionado con esto, tratando de obtener una mejor respuesta interactiva por parte del SO. Si no se encolase por adelante, había que esperar bastante para que P2 fuera desalojado por tiempo excedido, y si la cantidad de procesos CPUBound es grande, los tiempos de respuestas serán muy malos.

Unificación de colas USER y SERVER

Los cambios propuestos se centran alrededor de lo que podríamos llamar microkernel de Minix. Este se encuentra principalmente en el archivo `proc.c` y provee los mecanismos esenciales de manejo de procesos y mensajes. La parte restante del microkernel es código assembler para realizar tareas de bajo nivel que no necesitamos modificar.

Todo el manejo de las 3 colas de procesos se concentra en las funciones `ready()` y `unready()` de `proc.c`. Existe otro punto donde las colas son toqueteadas como se explicó en la sección introductoria pero como no modifica las 2 colas que quieren ser unificadas, no se tiene en cuenta.

Estructura de las funciones ready() y unready()

Ambas funciones presentan una estructura regular y clara que a grandes rasgos es:

```
ready(rp:Process) {
       si (rp.type==TASK) entonces {
                     TASK_Q.encolar(rp)
                     retornar
       }
       si (rp.type==SERVER) entonces {
              SERVER_Q.encolar(rp)
              retornar
       }
       USER_Q.apilar(rp)
}


unready(rp:Process) {
       case (rp.type) {
              TASK: si TASK_Q.esPrimero(rp)  ->  {
                           TASK_Q.borrarCabeza
                           pick_proc()
                     } sino
                           TASK_Q.borrar(rp)
              SERVER: si SERVER_Q.esPrimero(rp)  ->  {
                           SERVER_Q.borrarCabeza
                           pick_proc()
                     } sino
                           SERVER_Q.borrar(rp)
              USER: si USER_Q.esPrimero(rp)  ->  {
                           USER_Q.borrarCabeza
                           pick_proc()
                     } sino
                           USER_Q.borrar(rp)
       }
}
```

Como podemos ver se mantiene una regularidad en las 3 colas, rota en parte por la forma de encolado de USER_Q cuyas razones fueron debidamente justificadas. Es de notar que en el código fuente original de Minix 2.0.0, debajo de la porción de código que realiza el apilado en USER_Q, se halla comentado el "viejo" código que maneja USER_Q a la manera de cola como en Minix 1.5.1. Esto nos sirvió luego para encontrar soluciones parciales a algunos problemas.

Modificaciones realizadas

Los cambios fueron sencillos. Se decidió anular el código que encolaba en SERVER_Q, de manera que por la forma de codificar la función se tenía que ante la entrada de un proceso servidor (MM o FS) que no

caía en el caso task, caía en el caso por defecto que encolaba en USER_Q. Así obtuvimos lo que se buscaba, procesos task en TASK_Q y los otros 2 en USER_Q.

Para `unready()` la situación fue similar, debido a que si el microkernel quería quitar algún proceso de la cola server, este pedido debía ser redirigido a USER_Q, nuevo lugar del MM y FS.

En definitiva se trató de que con pocos y localizados cambios se obtuviera la modificación deseada.

Primer problema, solución parcial

Realizadas las modificaciones y ya con el kernel recompilado, todo intento por arrancarlo terminaba indefectiblemente en un mensaje de error "MM can't reply" y el sistema operativo en estado de espera a un warm reboot. Se hicieron algunos seguimientos y se pudo comprobar que el error surgía de la función `main()` del MM cuando trataba de hacer un reply con el resultado de una system call. El error que ocurría era ELOCKED y se generaba en `mini_send()` de `proc.c` perteneciente al kernel.

La función `mini_send()` realiza diversos testeos de conformidad antes de efectivamente copiar el mensaje en el buffer del receptor y colocarlo en la cola de listos. Uno de estas comprobaciones es la de ver que el receiver esté bloqueado esperando un send y en caso afirmativo seguir la cadena de procesos bloqueados por send hasta que o bien se llega a alguno que no esta esperando enviar o se encuentra el proceso enviador en esa cadena de bloqueados, en cuyo caso se devuelve el control con el error ELOCKED. En definitiva se previenen deadlocks por procesos esperando en cola send.

El problema parecía serio, por lo que se intentó buscar una solución por otro lado.

Se pensó en un problema que solo ocurría en la inicialización, por lo que se probaron versiones donde durante el proceso de inicio se tenía esquema cola para luego pasar a esquema pila. Los resultados fueron negativos, y se pudo comprobar que además del deadlock ocurrían errores bastante poco claros que en los dumps de procesos no encontraban explicación alguna.

Finalmente se cambió el esquema de apilado en USER_Q comentando una porción de código y descomentando otra ya existente, convirtiéndola en una cola. El cambio tuvo su efecto positivo y finalmente se obtuvo un kernel funcionando donde los procesos MM y FS bajaban de categoría. Se midió efectivamente que la SERVER_Q estaba vacía y que el MM y FS eran preemptados cuando agotaban su quantum. Además se realizaron pruebas con procesos CPUBound, IOBound tratando de usar varios dispositivos concurrentemente y cargar el sistema al máximo. Finalmente se ejecutaron satisfactoriamente los tests de Minix incluidos en el paquete original.

Solución al problema

El punto era claro, con modelo cola funcionaba perfectamente pero con modelo pila producía los mas diversos errores. Como ya habíamos comprobado que el esquema de encolado por detrás producía delays importantes en la ejecución de procesos con alguna componente de IO, decidimos investigar porque se daban tanto el deadlock como los otros errores.

Se intentaron soluciones alternativas como encolar los procesos user por delante y los server por detrás, encolar los procesos user por adelante solo si la cabeza de USER_Q no era el FS o el MM, además de otros esquemas poco convencionales que producían los mismos e impredecibles errores que el simple encolado por adelante.

Surgió el siguiente ejemplo para la unificación de colas con apilado para la USER_Q.

Supongamos que tenemos un proceso de usuario P1, donde la situación original es que P1 tiene el control de la CPU y está por realizar un system call al FS que se encuentra bloqueado en el receive del loop principal de atención de mensajes. La situación se muestra a continuación y la negrilla en P1 indica que este tiene el control de la CPU, es decir `proc_ptr` lo apunta.

Bloqueados por send	
Bloqueados por receive	FS
USER_Q	**P1**

En este momento P1 efectúa el system call para el FS. La función `sys_call()` de `proc.c` descompone toda llamada a sistema en un par send-receive, y esto es condición necesaria para todo llamado por parte

de un proceso de usuario (no debemos confundir proceso usuario con proceso cuyo lugar es la USER_Q). Por lo que después de efectuar el send tenemos:

Bloqueados por send	
Bloqueados por receive	
USER_Q	FS **P1**

El FS y P1 tuvieron el rendezvous por lo que FS fue movido por `ready()` a la cola de listos pero no hubo `pick_proc()` y `proc_ptr` sigue apuntando a P1. Después del send, y continuando dentro del kernel, se realiza el receive a favor de P1 buscando respuesta del FS, y como este no la tiene aun, se bloquea esperándola.

Bloqueados por send	
Bloqueados por receive	**P1**
USER_Q	FS

Cuando la función `mini_rec()` detecta que el proceso que debe enviar no está bloqueado esperando el receptor efectúa un `unready()` de este, borrándolo de la cola correspondiente. El problema surge cuando `unready()` debe quitar de la cola un proceso que no está al frente de esta. Minix simplemente efectúa una búsqueda lineal y retira el eslabón de la cadena. No se elige nuevo proceso.
La situación es incorrecta en el esquema de colas unificadas, porque al salir del kernel `proc_ptr` continúa apuntando a P1 y este seguirá ejecutándose como si la respuesta del FS hubiese llegado, cuando este último no procesó nada y no colocó ningún mensaje en el buffer de P1.

Para el código original con 3 colas y USER_Q como pila esto no es un problema. Veamos como se comporta. En la situación original tenemos:

Bloqueados por send	
Bloqueados por receive	FS
USER_Q	**P1**
SERVER_Q	

Se hace el send y el FS se encola en SERVER_Q

Bloqueados por send	
Bloqueados por receive	
USER_Q	**P1**
SERVER_Q	FS

Ahora se sucede el receive

Bloqueados por send	
Bloqueados por receive	P1
USER_Q	
SERVER_Q	**FS**

El caso anómalo no se dá (en Minix 2.0.0 los procesos que son apuntados por `proc_ptr` siempre están al frente de su cola). Al efectuar el `unready(P1)`, este se encontraba al frente de su cola, por lo que se realiza un `pick_proc()`, en donde SERVER_Q y por lo tanto FS salen favorecidos por el esquema de prioridad subyacente. Como bien marca el código, el rastreo lineal de una cola en busca del proceso a quitar solo se puede producir cuando se envía una señal para matarlo, por lo tanto no hay cambio de contexto y una llamada a `pick_proc()` no cambiaría nada, pero consumiría ciclos de reloj, luego es evitada.

La solución consistió en incluir un `pick_proc()` luego del barrido lineal en busca del proceso a quitar. Si el frente de la cola no había cambiado `pick_proc()` elegía el mismo, si se producía el caso anómalo se asignaba `proc_ptr` a FS, por lo que el FS obtenía el control y producía la respuesta que P1 esperaba.

Finalmente se obtuvo la combinación buscada: unificación de colas que baja de categoría el FS y MM y encolado por adelante que provee interactividad cuando en el sistema hay procesos CPUBound.

Tests y Mediciones

Se decidió probar como se comportaban las distintas versiones estables que se pudieron lograr en escenarios de ejecución especialmente preparados.

Había 2 características principales a modificar. Una era la unificación de colas y la otra era el encolado por adelante o por detrás, con lo que se generaron 4 kernels cubriendo el espacio de posibilidades. Cada uno de estos kernels fue debidamente probado con los casos de test dados en el paquete original que se encuentran en el directorio `/usr/src/test`.

Los procesos que se ejecutaron sobre estos kernels fueron sencillos y polarizados al IO o al cálculo.

Veamos una tabla de los kernels y de los casos de test:

Kernel		
Colas Unidas	Modelo Pila	
0	0	Como Minix 1.5.1
0	1	Standard
1	0	1er intento de colas unificadas
1	1	Modelo final

Caso	Fore	Back	
A	Io	Idle	IO vs CPU
B	Io	Compile*	IO vs IO
C	Cpu	Compile*	CPU vs IO
D	Cpu	Idle	CPU vs CPU

El proceso `Io` consistía en repetir 5 veces el test del shell `/usr/src/test/sh1` que produce principalmente una carga la entrada/salida. El proceso `Cpu` era el contrario, hacia millones de iteraciones sin tocar nada del IO. Para procesos background hicimos un pequeño código C que entraba en un bucle infinito (la carga CPUBound) y un script que compilaba el kernel de manera cíclica como carga IOBound.

Los resultados se muestran a continuación

Kernel\Caso	A	B	C	D
00	+20'	3'10"	18"	37"
01	32"	56"	26"	35"
10	+30'	3'05"	18"	57"
11	35"	1'07"	26"	34"

Para el caso A podemos ver la importancia que tiene el encolado por adelante cuando tenemos procesos de alguna manera interactivos, en los kernels 00 y 10 no se terminó de realizar la medición, y se estimó que terminarían en algún momento nada cercano a los kernels con encolado por adelante. Pudimos concluir lo que notábamos en el uso tortuoso de los kernels con modelo de pila.

El caso B es similar a A en cuanto a la estructura de tiempos. Sin embargo notamos que los kernels cola hacen decaer la interactividad, por mas de que los procesos que están en background sean IOBound también.

Para el 3er tipo de comprobación vemos una disposición inversa respecto a los casos anteriores. El modelo cola no prioriza a los IOBound, con lo que los procesos CPUBound no son interrumpidos y terminan antes.

El caso D presenta una pobre performance para el kernel de colas unificadas y esquema cola. No se pudo comprender el porqué de esta diferencia. Un análisis a primera vista muestra que el kernel 00 y el 10 no deberían tener diferencias de tiempos tan grandes cuando todos los procesos en juego no utilizan el FS y MM.

En general podemos decir que con colas unificadas perdemos un poco la interactividad a favor de los procesos background IO o CPU que se están ejecutando, lo cual es un poco el espíritu de lo que

queríamos lograr en pos de un acercamiento al RT por parte del Minix. Para el caso de esquema cola/pila, la respuesta es tajante como las mediciones lo muestran: si queremos priorizar IO será pila y para CPU será cola

Análisis comparativo del scheduler con colas unificadas

Como vimos antes el scheduler de Minix 2.0.0 tiene dos variantes para la cola user, la primera es que esta se comporte como una cola (en cuanto al `ready()` y `unready()`) y la segunda es que la misma se comporte como una pila. En Minix 2.0.0 la versión por defecto es la pila. Es por eso que nosotros analizaremos aquí el impacto de la unificación de colas con respecto a estas dos políticas.

Para llevar a cabo nuestro análisis usaremos los tres procesos que se muestran a continuación:

P1	Call_FS(120ms)
P2	Call_FS(120ms)
P3	Calcular(400ms)

Al lado de cada operación, se detalla entre paréntesis el tiempo esperado de ejecución para cada uno. Supondremos también que el context switch y toda otra operación que pertenezca a las antes mencionadas tendrá un tiempo de ejecución igual a cero. Esta asunción no le quita generalidad al problema y permite simplificar el análisis. El quantum del scheduler será de 100ms.

Los procesos elegidos pertenecen a dos categorías distintas, P1 y P2 realizan un uso intensivo del FS, o de recursos compartidos, por lo cual se los denomina IOBound. En cambio P3 se dedica a realizar cálculos, por lo cual el único recurso que necesita es el CPU, a este se lo denomina CPUBound.

Es importante mencionar que con las hipótesis antes nombradas estamos dejando de lado en este análisis el caso en que el FS (o MM) no son preemtados por el scheduler (por timeout). Esto se debe a que en dicho caso la planificación obtenida no varia mucho de la original, incluso si usamos colas unificadas con semántica de pila las planificaciones serán iguales a las del scheduler tradicional de Minix 2.0.0.

A continuación mostramos las planificaciones obtenidas para los procesos arriba descriptos. Se analizaron tres tipos de scheduler diferentes: el tradicional de Minix 2.0.0 (kernel tipo 01), la versión con colas unificadas y manejo tipo pila (kernel tipo 11) y la versión con colas unificadas y manejo tipo cola (kernel tipo 10).

Scheduler tradicional de Minix 2.0.0 (kernel tipo 01)

Ready	User ready stack	Blocked	Tiempo	Acción
P1	P2,P3	FS	0	Recv(FS)
FS	P2,P3	P1	120	Send(P1)
FS	P1,P2,P3		120	Recv(ANY)
P1	P2,P3	FS	120	Quantum TimeOut
P2	P3,P1	FS	120	Recv(FS)
FS	P3,P1	P2	240	Send(P2)
FS	P2,P3,P1		240	Recv(ANY)
P2	P3,P1	FS	240	Quantum TimeOut
P3	P1,P2	FS	340	Quantum TimeOut
P1	P2,P3	FS	340	Exit
P2	P3	FS	340	Exit
P3		FS	440	Quantum TimeOut
P3		FS	540	Quantum TimeOut
P3		FS	640	Exit

Scheduler de colas unificadas por pila (kernel tipo 11)

Ready	User ready stack	Blocked	Tiempo	Acción
P1	P2,P3	FS	0	Recv(FS)
FS	P2,P3	P1	100	Quantum TimeOut
P2	P3,FS	P1	100	Recv(FS)
P3	FS	P1,P2	200	Quantum TimeOut
FS	P3	P1,P2	220	Send(P1)

FS	P1,P3	P2	220	Recv(ANY)
FS	P1,**FS**,P3	P2	300	Quantum TimeOut
P1	FS,P3	P2	300	Exit
FS	P3	P2	340	Send(P2)
FS	P2,P3		340	Recv(ANY)
P2	P3	FS	340	Exit
P3		FS	440	Quantum TimeOut
P3		FS	540	Quantum TimeOut
P3		FS	640	Exit

Es interesante notar que debido a la semántica del `ready()` y el `unready()`, en la fila siete el proceso que se esta ejecutando no es el tope de la pila. Este caso ya fue comentado en secciones anteriores.

Scheduler de colas unificadas por cola (kernel tipo 11)

Ready	User ready queue	Blocked	Tiempo	Acción
P1	P2,P3	FS	0	Recv(FS)
P2	P3,FS	P1	0	Recv(FS)
P3	FS	P1,P2	100	Quantum TimeOut
FS	P3	P1,P2	200	Quantum TimeOut
P3	FS	P1,P2	300	Quantum TimeOut
FS	P3	P1,P2	320	Send(P1)
FS	P3,P1	P2	320	Recv(ANY)
FS	P3,P1	P2	400	Quantum TimeOut
P3	P1,FS	P2	500	Quantum TimeOut
P1	FS,P3	P2	500	Exit
FS	P3	P2	540	Send(P2)
FS	P3,P2		540	Recv(ANY)
P3	P2	FS	640	Exit
P2	P3	FS	640	Exit

Para analizar que procesos son priorizados por los distintos schedulers, debemos suponer que arrancamos un cronometro con cada proceso y lo detenemos a su finalización. La tabla de los tiempos de ejecución sería

Kernel/ Proc	01	11	10
P1	340	300	500
P2	340	340	640
P3	640	640	640

Como podemos ver los tiempos son muy similares para el scheduler de los kernels 01 y 11, en cambio para el caso del tipo 10 los procesos IOBound han incrementado su demora mientras que el CPUBound se mantuvo constante. Si vemos la última planificación notaremos que si P1 y P2 fuesen mas largos (40ms mas), veríamos como su tiempo de ejecución sería mayor que el de P3. Hay que tener en cuenta que P1 debería ejecutarse en dicho caso en 160ms (si el sistema fuese monotarea) y P3 en 400ms.

Es interesante notar que en la mayoría de las aplicaciones de tiempo real, los procesos que se utilizan no hacen uso de los recursos compartidos. Estos procesos por lo general utilizan hardware especifico por polling, lo cual a los para los fines del sistema operativo puede ser considerado como CPUBound. Es por esto que las modificaciones (kernel tipo 10) realizadas al código de Minix 2.0.0 favorecen a los procesos de tiempo real.

Colas unificadas y deadlocks

Uno de los problemas que pueden surgir al disminuir la disponibilidad de los recursos compartidos, en este caso el FS y MM, es la generación de deadlocks. Para entender este concepto pensemos en el scheduler original de Minix 2.0.0 en el cual, cualquier proceso que quería acceder al FS o MM lo hacia de forma inmediata. Esto se debía a que el FS y MM tenían la suficiente prioridad para comenzar su ejecución en el acto y no ser preemtados.

Para que un deadlock ocurra se debe dar lo que se denomina una espera circular. La misma ocurre cuando un proceso esta bloqueado a la espera de otro y a su ves el segundo espera al primero.

Al realizar este análisis de deadlocks no consideramos los que se pueden producir simplemente por haber cambiado la política de scheduler. Este es el caso de procesos USER que tengan alta dependencia de las planificaciones generadas por el scheduler original de Minix. Nuestro análisis intenta ver la posibilidad de deadlock entre el FS y MM analizando la semántica de los mismos para luego evaluarla a la luz del nuevo scheduler. Luego de analizar el código, se comprobó que la única comunicación entre el FS y MM se realiza durante la inicialización del sistema. Esta es solo en un sentido, por lo cual no puede generar espera circular. Por esto podemos concluir que el FS y MM funcionan de forma independiente y solo se comunican con procesos de nivel TASK (drivers o el kernel). Las tareas de nivel TASK son de alta prioridad por lo cual no son preemptadas y su ejecución se puede considerar instantánea (y atómica) con respecto a un proceso USER.

Finalmente con lo antes expuesto, podemos concluir que al no hay comunicación a nivel usuario, ya que los procesos user no están autorizados para hacerlo y el FS no se comunica con el MM. Luego cualquier espera circular que se produzca debe ser dentro de los procesos task. Esto es absurdo ya que de ser así el scheduler original de Minix 2.0.0 tendría deadlocks, nótese que la unificación de colas no afecto el manejo de las TASK_Q ni los conjuntos de pedidos que pueden recibir los procesos task.

El FS y MM en un entorno de RT

Para poder completar este trabajo, era necesario analizar el desempeño del FS y MM con un scheduler RT (tasa monotónica, MAP, ME, etc.). Para esto supongamos que el mismo posee primitivas de RT y veamos que pasaría si algún proceso quisiese interactuar con el FS o MM.

El primer problema que surge es que no tenemos restricciones de tiempo para las operaciones del FS/MM. Esto hace disminuir la predictibilidad de todo proceso que quiera usar estos servidores. Este problema es fácilmente solucionable para el MM ya que el mismo no involucra dispositivos de que interactúen con el mundo real. Debido a esto, se podría hacer un análisis del código junto con algunas mediciones, para así obtener una tabla con los tiempos que cada system call requiere. El caso del FS es mas complejo ya que al involucrar dispositivos de hardware, se dificulta la tarea de medir las restricciones temporales del servidor. No obstante esto, se puede obtener un conjunto de valores que promedien los peores casos de ejecución. Debemos notar que esta solución no es tan buena como la del MM el cual es totalmente predecible. Otra desventaja del FS con respecto al MM es que resulta mas difícil de preemptar, esto será tratado con mas detalle en la próxima sección.

El segundo problema es un clásico de los sistemas de tiempo real al que se denomina Inversión de Prioridad. Supongamos el caso que un proceso de baja prioridad le pide al FS que lleve a cabo alguna operación. Luego de un tiempo otro proceso de máxima prioridad debe acceder al FS, pero el mismo no lo puede hacer ya que el primer proceso no lo ha liberado. Es así como el proceso de máxima prioridad deberá esperar al de baja prioridad para continuar. La situación se puede poner peor aun si existen procesos de prioridad intermedia que deseen ejecutarse. En vistas de que no hay ningún otro proceso con prioridad mayor ejecutándose, se apropiaran de la CPU, esto hará que la liberación del FS se vea prorrogada. En síntesis la Inversión de prioridad hace que las tareas de máxima prioridad no puedan cumplir sus metas.

Características de un servidor de RT

El párrafo anterior nos abrió varios interrogantes sobre como debería ser un verdadero RT Server (ya sea el FS o MM). Para esto hicimos un poco de estudio sobre publicaciones en el tema y pudimos ver que el mismo es bastante reciente por lo cual no existe un standard bien definido. A pesar de esto existen varias líneas de investigación que han tenido un relativo éxito. La mayoría de estos trabajo se han experimentado bajo RT-Mach. En esta sección intentaremos resumir de forma breve algunas de las ideas principales para la implementación de un RT server.

Como vimos en la sección anterior, surgían dos problemas al interactuar con un recurso compartido (como el FS o MM): la predictibilidad del mismo y la inversión de prioridades.

La predictibilidad es algo difícil de resolver cuando tenemos dependencia con el hardware subyacente, mas aun cuando el mismo es naturalmente monotarea. Es por esto que cada caso se trata por separado. Por ejemplo para el FS la predictibilidad se logra usando schedulers de disco para RT. Estos scheduler introducen la noción de metas y de reserva de bandwidth. Las metas cumplen el mismo objetivo que en un scheduler de RT común, por lo cual no vamos hablar de ellas. La reserva de bandwidth permite que un proceso informe al FS cual es la cantidad de datos que espera leer por segundo. Este tipo de medida es

muy útil para las aplicaciones de multimedia, ya que en general deben acceder de forma repetitiva al disco duro.

Para que un server pueda ser de tiempo real es importante el grado de disponibilidad que el mismo posee. Entendiendo por disponibilidad la probabilidad que en cualquier instante de tiempo un proceso pueda acceder al servidor sin tener que ser demorado. Como nos interesa que la disponibilidad tenga en cuenta las prioridades de los procesos, vamos a requerir que la disponibilidad aumente con la prioridad de los mismos. Este concepto nos obliga a pedir que nuestro server sea preemtable, lo cual no es algo fácil de lograr si pensamos que cuando a un HD se le da un seek, no se lo puede interrumpir. La preemption trae involucrada nociones mas complejas de sistemas operativos como la reentrancia, que es la propiedad de poder detener a un proceso que se esta ejecutando en el server, para poner a otro dentro del mismo. La diferencia entre preemption y reentrancia radica en que la preemption solo hace una pausa en el servidor, mientras que la reentrancia permite pausar un proceso, ejecutar otro y volver al primero. Para poder tener reentrancia el servidor debe mantener el status del mismo con respecto a cada proceso que esta atendiendo y no en forma global. Siguiendo con el ejemplo del FS, lo que se utiliza es descomponer operaciones grandes en varias pequeñas. Por ejemplo podemos convertir la operación de abrir un dado archivo en varias suboperaciones de lectura, escritura o seek. El tener operaciones pequeñas permite al scheduler de disco decidir que proceso atender luego de cada ejecución, de esta forma el RT scheduler de disco puede dejar de atender un conjunto de instrucciones con prioridad k en favor de otro con prioridad j (j>k). Este comportamiento simula reentrancia de procesos en el servidor.

El problema de la inversión de prioridades se puede solucionar permitiendo que el servidor herede la máxima prioridad de los procesos que están esperando por su servicio. De esta forma, si un proceso de baja prioridad se esta ejecutando en el servidor cuando otro de mayor prioridad quiera acceder al mismo, este heredara la prioridad del segundo, así el servidor se podrá desocupar rápido para atender al proceso de mayor prioridad.

Un mejor esquema para el manejo de las prioridades son lo que se denominan worker model. Existen dos tipos de worker model, el de prioridades estáticas y el dinámico. La idea del static prioritized model es que el servidor tenga dentro de él varias threads con distintas prioridades que se encargaran de recibir las ordenes de un proceso cliente y llevarlas a cabo dentro del servidor. La prioridad de la thread receptora deberá ser igual a la del proceso cliente. Si en el momento de acceder al servidor no hubiese ninguna thread con la prioridad apropiada, el proceso deberá espera hasta que una se libere. El caso con prioridad dinámica es igual al anterior, salvo que las threads son creadas para cada proceso que quiere acceder al server. Esto es mas costoso ya que crear una thread lleva tiempo, pero permite que ningún proceso deba esperar para ser ejecutado.

Los worker model permiten tener mas disponibilidad que en un servidor standard. Por ejemplo si pensamos en el FS, este podría tener varias threads que se encargasen de descomponer las macro operaciones en suboperaciones. Las suboperaciones serían luego atendidas por RT scheduler de disco, el cual consideraría cuales deberían ser atendidas con mayor urgencia.

Conclusiones

La pregunta sobre la que se basó el trabajo fue respondida: es posible unificar las colas de usuario y servidor en Minix. Además se comprobó que este cambio aporta nuevas características de Real Time a este sistema operativo que ya fue modificado a tal efecto hace unos años por un grupo en U.B.A.

Comprobamos porqué Minix es un SO orientado a la enseñanza, es abierto, modular, sencillo y auto explicativo. Gracias a estas características ahorramos tiempo en la comprensión del código y su funcionamiento, a pesar de no haber utilizado mas que los fuentes.

Todo el trabajo nos marcó que aunque este era el camino a seguir para Minix, la real dirección está en investigar el problema de fondo que son los servidores RT.

Bibliografía

- Nakajima – Tokuda, "User level Real Time Network System on microkernel based Operating Systems"
- Nakajima - Kitayama – Tokuda, "Experiments with Real-Time Servers in RT-Mach"
- Molano - Juvva – Rajkumar, "Guaranteeing Timing Constraints for Disk Accesses in RT-Mach"

- Wainer G., "Sistemas de Tiempo Real, conceptos y aplicaciones"
- Tanenbaum A., "Sistemas Operativos, diseño e implementación",
- Silberschatz – Galvin, "Operating system concepts", 4th edition
- Bach, "The design of the Unix operating system"