Diseño y simulación de un planificador para un sistema de virtualización basado en Minix*

Paparotti, Lautaro
Prinsich Bernz, Emilio
Quaglia, Constanza
Director: Pessolani, Pablo

Universidad Tecnológica Nacional
Facultad Regional Santa Fe

Resumen El sistema operativo de microkernel Minix[1] brinda un excelente entorno para incorporarle funciones de hipervisor. El presente trabajo busca diseñar y evaluar un planificador jerárquico que permita seleccionar en primera instancia la máquina virtual y luego el próximo proceso a ejecutar de esa máquina virtual.

El proyecto propone un algoritmo de planificación de tipo token bucket (cubeta de fichas)[2] para la selección de las máquinas virtuales y administración de los tiempos de CPU para cada una.

Palabras Clave: Minix, sistema operativo, planificación de procesos, máquinas virtuales.

1. Introducción

Un sistema operativo (OS) realiza dos funciones básicas: proporcionar a los programadores y a los programas un conjunto abstracto de recursos simples, en vez de los complejos conjuntos de hardware; y administrar dichos recursos de hardware[3].

El presente artículo es parte de un proyecto denominado Mhyper, que tiene por objetivo explorar oportunidades para incluir soporte de virtualización en el microkernel de Minix.

El propósito es divulgar el diseño y evaluación de desempeño de un planificador jerárquico utilizando Mhyper como base del proyecto. Se propone un algoritmo de tipo token bucket para planificar las máquinas virtuales y sus respectivos procesos.

Para simular y evaluar los algoritmos propuestos se utilizó el software SimSo[4], que permite realizar un prototipo de planificador, definir procesos, simularlos y obtener estadísticas del tiempo de procesamiento y rendimiento del hardware.

* Este trabajo fue realizado en el marco del proyecto UTN 1766 en la Facultad Regional Santa Fe.

2. Contexto de trabajo

Los sistemas operativos pueden dividirse según su estructura en sistemas monolíticos, sistemas de capas, microkernels, cliente-servidor, máquinas virtuales (VMs) y exokernels. Particularmente, Minix es un OS microkernel, totalmente estratificado, desarrollado por Andrew Tanenbaum en 1987 con fines académicos. En su versión 3, los procesos se dividen en capas, tal como se muestra en la figura 1. Los procesos de las capas 2, 3 y 4 se ejecutan en modo usuario y comprenden las siguientes categorías:

- Procesos de usuarios.
- Procesos servidores, que son los que apoyan y administran los procesos de usuario y el entorno mismo de Minix.
- Drivers o tasks, que son los que administran los distintos drivers del hardware.

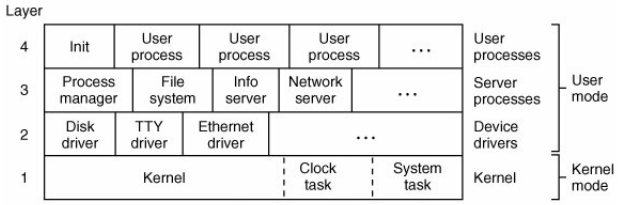

Figura 1. Arquitectura de Minix (de [1]).

La principal característica que posee Minix es la independencia y aislamiento que poseen los procesos entre sí. Cada uno de ellos maneja su propia información y privilegios asignados. Todas las peticiones de información o solicitudes de alguna acción de parte de otro servidor se realizan por medio de la transferencia de mensajes. La información de cada servidor, proceso o driver está aislada del resto, sin la posibilidad de tener acceso a las variables globales del mismo OS que no sea a través de un mensaje.

El kernel se encarga de planificar procesos, de las transiciones entre estados (listo, bloqueado y en ejecución) y de los mensajes entre procesos. El manejo de mensajes implica la verificación de destinatarios válidos, la localización de los bufferes de envío y recepción en la memoria física y la copia del mensaje.

Además posee dos procesos con privilegios especiales: la system task (SYSTASK) y la CLOCK task. La system task es el proceso intermediario entre los procesos de usuario y el microkernel. Provee a los drivers y servidores una serie de llamadas al kernel privilegiadas (kernel calls). Esto incluye lectura y escritura de los puertos de entrada/salida (E/S) y copia de datos entre espacios de direcciones. La CLOCK task es el reloj del sistema.

Por otro lado, la virtualización consiste en un software que emula al hardware y puede ejecutar programas como si fuese una computadora real. Dicho software se denomina hipervisor. Esta tecnología permite que una sola computadora contenga varias VMs, denominadas huéspedes.

Una VM debe contar con las siguientes características:

- *Duplicado*: Debería comportarse de forma idéntica a la máquina real, excepto por la existencia de menos recursos disponibles (incluso diferentes entre ejecuciones) y diferencias de temporización al tratar con dispositivos.
- *Aislamiento*: Las VMs no deben interferirse entre sí. Esta caracteristica refiere a:
  - *Rendimiento*: El rendimiento de una VM no debe afectar el de las otras.
  - *Fallo*: Un fallo en una VM no debe provocar ningún efecto en las otras.
  - *Seguridad*: No debe ser posible acceder a los recursos de una VM desde otra.
- *Eficiencia*: La VM debería ejecutarse a una velocidad cercana a la del HW real, lo cual requiere que la mayoría de las instrucciones se ejecuten directamente por el HW.

Se pueden distinguir varios tipos de virtualización:

- *Hipervisor tipo I (full virtualization)*: El monitor se ejecuta directamente sobre el hardware en modo kernel y los huéspedes sobre el monitor en modo usuario.
- *Hipervisor tipo II (indirecto)*: El monitor se ejecuta sobre un OS en modo usuario y los huéspedes sobre el monitor.
- *Paravirtualización*: Consiste en modificar el código fuente del OS huésped de manera que en vez de ejecutar instrucciones sensibles ejecuten llamadas al hipervisor. De esta manera se logra un rendimiento cercano a tener máquinas reales.

La posibilidad de utilizar el microkernel de Minix como hipervisor es muy buena, dado que los procesos son entidades autónomas y aisladas del resto, característica primordial de la virtualización. Además, la SYSTASK de Minix provee servicios a los drivers y servidores a través de llamadas al kernel, del mismo modo que un hipervisor paravirtualizado provee llamadas al hipervisor, con lo cual se puede decir que la SYSTASK en Minix actúa como hipervisor paravirtualizado de una única VM. El objetivo del proyecto Mhyper es adaptar a Minix para convertir a la SYSTASK en un hipervisor paravirtualizado de múltiples VMs.

La arquitectura de Mhyper se muestra en la figura 2.

Mhyper funciona de la siguiente manera: comienza a correr la VM0, un pseudo-contenedor que tiene asignados todos los recursos del sistema. Como la VM0 tiene los mismos servidores y tareas que Minix, si no se inicia ninguna otra VM, el comportamiento Mhyper es como el de un Minix estándar. La VM0 es una máquina virtual privilegiada con seguridad que permite administrar las distintas VMs.

Además se añadió un nuevo servidor para administrar VMs llamado Virtual Machine Manager (VMM). También se creó una herramienta de administración para cargar, iniciar, detener, reanudar y terminar las VMs utilizando las llamadas al sistema servidas por VMM.

Para mantener el aislamiento de seguridad, recursos y fallos, el espacio de direcciones de memoria asignado a una VM no puede ser accedido por ninguna otra VM, incluyendo la VM0. El Process Manager (PM) de la VM0 no tiene el rango de direcciones de memoria de una VM bajo su gestión hasta que ésta termina. Además, la SYSTASK mantiene la gama del espacio de direcciones para cada VM para poder controlar llamadas al kernel solicitadas, que operan en la memoria y se realizan dentro del área de la memoria de una VM (es decir, la copia de bloques de memoria entre procesos).

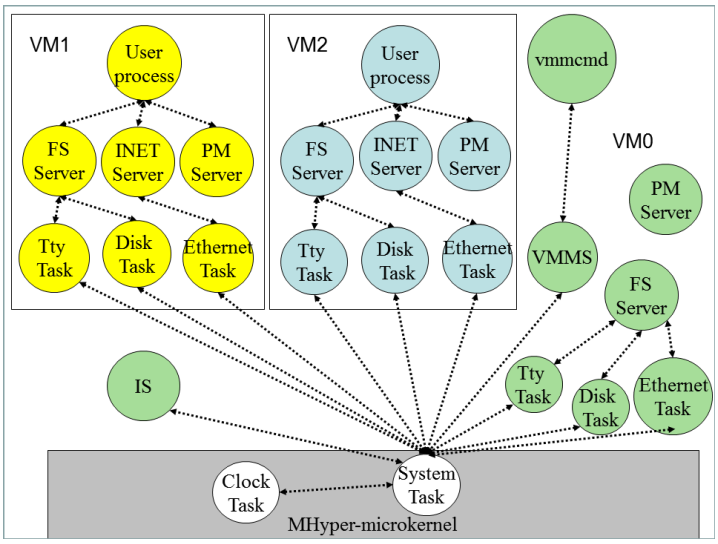

Figura 2. Arquitectura de Mhyper.

3. Planificador actual y planificador propuesto

3.1. Planificación en Minix 3

En un sistema de multiprogramación, frecuentemente existen múltiples procesos que compiten por la CPU. Cuando hay más de un proceso en estado de listo y sólo una CPU disponible, el OS debe decidir cuál de estos procesos correr primero. La parte del OS que toma esta decisión se conoce como planificador.

Minix 3 utiliza un algoritmo de planificación por prioridades. Cada proceso tiene una prioridad inicial, relacionada a la arquitectura que se muestra en la

figura 1, donde los procesos de capas inferiores tienen mayor prioridad que la de las superiores.

El planificador mantiene 16 colas de procesos ejecutables en estado de listo, una por prioridad, como se muestra en la figura 3.

Dentro de cada cola, se aplica un algoritmo de round robin. Cuando a un proceso que se está ejecutando se le termina el quántum es movido al final de la cola y se le asigna un nuevo quántum. En cambio, cuando un proceso que estaba bloqueado pasa a listo, es puesto al principio de la cola y no se le da un nuevo quántum sino que termina de ejecutar lo que le quedaba antes de ser bloqueado. Cuando un proceso se bloquea o recibe una señal *kill*, es removido de la cola del planificador.

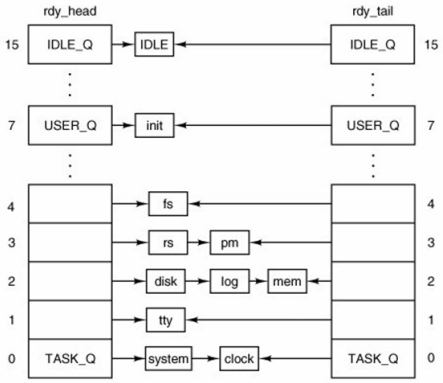

Figura 3. Cola de listos de Minix 3

El algoritmo consiste entonces en encontrar la cola de mayor prioridad no vacía y elegir el primer proceso de esa cola. El proceso IDLE está siempre listo y es el de menor prioridad, por lo tanto se ejecutará cuando todas las colas restantes estén vacías.

Para encolar y desencolar los procesos, Minix se basa en las funciones *enqueue* y *dequeue*. Enqueue coloca un proceso en la cola que corresponda (la crea si es necesario) y llama a *pick_proc*, que determina cuál será el próximo proceso a ejecutar.

No obstante, cuando se trata de VMs, el planificador debe poder planear algo más que procesos: debe ser capaz de planificar las diferentes VMs.

Supóngase que existen dos huéspedes, VM1 y VM2, corriendo procesos con igual prioridad. Si VM1 y VM2 tuvieran 8 y 2 procesos respectivamente en estado listo, VM1 tendría 80 % más de probabilidad de ocupar el CPU que VM2. Peor aún, cuantos más procesos ponga VM1 a ejecutar, menor será la porción de CPU que se le asignará a VM2. De este modo, se afecta el aislamiento de rendimiento

que toda VM debería tener. Actualmente el planificador de Mhyper es de tipo round-robin donde no se distingue a qué VM pertenece un proceso o tarea.

3.2. Soluciones propuestas

Round robin con prioridad. A priori se puede pensar en un algoritmo de round robin que elija entre las diferentes VMs. Sin embargo esta solución simple tiene el defecto de no poder distinguir entre las diferentes prioridades que tienen los procesos dentro de las VMs. Así, es posible que se planifique un proceso de usuario perteneciente a una VM, mientras que una tarea de disco de otra está esperando su turno. El objetivo es que las tareas, que están relacionadas con la gestión de dispositivos de E/S y que son sensibles al tiempo, no se retrasen por ejecutar procesos en otras VMs.

Por este motivo se decidió dividir a los procesos en dos grupos: *tareas* (tasks) y *procesos*. Se considera *tarea* a aquellas que pertenezcan a la capa 2 en la arquitectura de Minix y *procesos* a los pertenecientes a las capas 3 y 4.

De este modo, es posible darle prioridad a las *tareas* por encima de los *procesos*. Un pseudocódigo para dicho planificador podría ser el siguiente:

```
pick_vm():
    search_for_tasks();     // Tareas
    if(proc_ptr != NULL)
        return;
    search_for_proc();      // Procesos
    if(proc_ptr != NULL)
        return;
    proc_ptr = IDLE;        // IDLE
    return;
```

Utilizando dicho algoritmo se simuló un escenario con 4 VMs (VM0 y 3 huéspedes), donde la VM1 ejecuta una *tarea* periódica con un alto requerimiento de CPU. El resultado fue que esta *tarea* se llevó el 60 % del CPU, mientras que los *procesos* no se ejecutaron nunca.

La solución propuesta para este problema es un algoritmo de tipo token bucket similar al utilizado en redes y el algoritmo de créditos utilizado por XEN[5].

Algoritmo de token bucket. Este algoritmo consiste en agregar al round robin ya visto una cubeta o bucket por cada VM. De este modo, se logra la precedencia de las *tareas* por encima de los *procesos*. Las tareas son responsables de gestionar dispositivos de E/S y tienen algunas características de tiempo real. Sin embargo, la ejecución de cada VM se limita para lograr una mejor distribución del CPU.

Cada token representa una fracción de tiempo de CPU de t ms. Cada t ms, se le resta un token a la VM en ejecución.

El tamaño de token t no debe ser tan grande como para que exista demasiada latencia en la atención de un dispositivo, ni tan pequeño como para que se produzca un cambio de VM con demasiada frecuencia, generando mayor overhead.

Por simplicidad de implementación, se eligió $t = \frac{1}{128}$.

Funciona de la siguiente manera: al iniciar la VM0, ésta tiene asignados 128 tokens. Cuando se inicia una VMi el sistema extrae $BSize_i$ tokens de la VM0 y se los asigna a VMi. El tamaño de bucket $BSize_i$ se obtiene como parámetro al iniciar una VM y no puede superar la cantidad de tokens que tenga VM0 en ese momento. Esto implica que la sumatoria total de los $BSize_i$ nunca será mayor a 128. Al detener una VM, sus tokens asignados se le devuelven a la VM0.

La VM0 es un caso especial. La asignación de tokens se realiza con el único propósito de distruibuirlos a las distintas VMs y está excenta del conteo de tokens, ya que por ser el hipervisor sus procesos son prioritarios para el funcionamiento del sistema.

La selección de las VMs es en round robin, con la siguiente prioridad:

- *Tareas* y *procesos* de la VM0.
- *Tareas* de VMs con $tokens_i > 0$.
- *Procesos* de VMs con $tokens_i > 0$.
- *Tareas* de VMs con $tokens_i \leq 0$.
- *Procesos* de VMs con $tokens_i \leq 0$.
- *IDLE*.

Es decir que las VMs con tokens tienen prioridad sobre las VMs sin tokens. Las VMs sin tokens serán seleccionadas solamente si las VMs con tokens no tienen procesos en condiciones de ejecución. Con esto, al asignarle más tokens a una VM se le está dando mayor prioridad en tiempo de ejecución.

Cada cierto tiempo el sistema se refresca, todos los buckets se vuelven a su estado inicial, es decir $tokens_i = BSize_i$. El tiempo de refresco t_r se calcula como el tiempo que tomaría consumir todos los tokens del sistema:

$$t_r = \frac{1}{128} \sum tokens_i \; ms \tag{1}$$

A continuación se presenta un pseudocódigo del algoritmo:

```
on_timer_tick():
    if (running_vm != 0):
        // Le quita un token a la VM en ejecucion
        tokens[running_vm] --;
        // Se reduce el tiempo para el proximo refresh
        time --;
        if(time == 0):
            // Se refresca el sistema
            time = t_r
            for i in vms_activas:
                tokens[i] = BSize[i]
```

```
pick_vm():
    search_VM0();               // Tareas y procesos de la VM0
    search_for_tasks(true);     // Tareas de las VMs con tokens
    if(proc_ptr != NULL):
        return;
    search_for_proc(true);      // Procesos de las VMs con tokens
    if(proc_ptr != NULL):
        return;
    search_for_tasks(false);    // Tareas de las VMs sin tokens
    if(proc_ptr != NULL):
        return;
    search_for_proc(false);     // Procesos de las VMs sin tokens
    if(proc_ptr != NULL):
        return;
    proc_ptr = IDLE;            // IDLE
    return;
```

En resumen, este algoritmo permite:

- Priorizar *tareas* sobre *procesos*, sin descuidar éstos últimos.
- Distribuir equitativamente la CPU (aislamiento de rendimiento), sin comprometer el rendimiento. La asignación de tokens evita que una VM se adueñe del procesador, pero si la carga es baja, les da libertad de uso aunque no tengan tokens.
- Asignar diferentes prioridades a las VMs, a través del tamaño de bucket asignado.
- Aislamiento de fallo. Ante un driver o proceso fallido que requiera demasiado tiempo de CPU, el algoritmo termina bajándole la prioridad, ya que consumirá todos sus tokens.

4. Evaluación de desempeño

Se realizaron diferentes corridas de simulación variando algunos escenarios, de manera de poder evaluar el cumplimiento efectivo de las características mencionadas.

Priorización de *tareas* sobre *procesos*. En la primer simulación se tienen 3 VMs (VM0 y 2 huéspedes). Entre cada corrida se hace variar el tamaño de buckets asignados a cada VM.Como se puede apreciar en la figura 4, inicialmente se les da un bucket de 4 a cada una. Luego, cada corrida se aumenta el bucket de la VM1 en 4 tokens.

A medida que VM1 adquiere mayor proporción de tokens, la *tarea* de la VM1 (T VM1) ocupa mayor porcentaje del CPU, mientras al disminuir la proporción de tokens de la VM2, se disminuye el tiempo asignado a su *proceso* (P VM2) pero no a su *tarea* (T VM2). Por otro lado, la *tarea* de la VM0 se mantiene constante, ya que no compite con las demás.

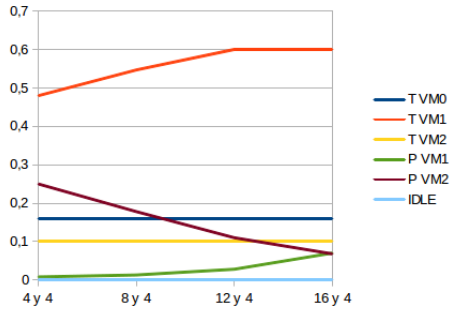

Figura 4. Priorización de *tareas* sobre *procesos*.

Priorización de las VMs. Con el mismo escenario anterior, se mide la proporción de CPU utilizada por cada VM. El porcentaje utilizado por VM0 se mantiene constante, mientras que la proporción de uso de CPU utilizado por VM1 y VM2 es proporcional a sus respectivos tamaños de bucket, como se puede ver en la figura 5. De este modo, la VM con mayor tamaño de bucket tendrá mayor tiempo de uso de CPU.

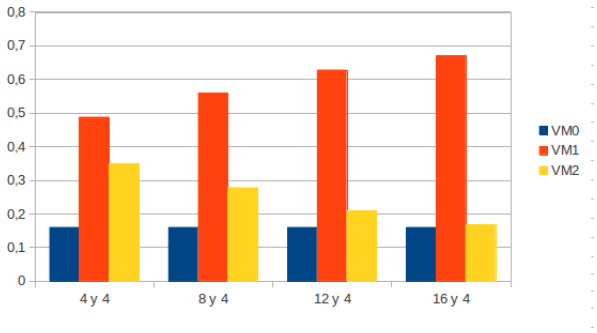

Figura 5. Priorización de las VMs.

Distribución de CPU. Para este caso, se tiene un escenario de 5 VMs (VM0 y 4 huéspedes) con igual tamaño de bucket. En primer lugar, se varía la carga sobre el CPU, variando en forma proporcional la carga de cada VM. Se observa en la figura 6 que con una carga baja (66 %) cada VM utiliza lo necesario. Al duplicarse los requerimientos de uso de CPU (con lo cual la carga pasa a ser 100 %), las VMs deben competir entre sí, distribuyendo el CPU de forma equitativa. Bajo el mismo escenario, se varió el tamaño de bucket, pudiéndose observar que la tendencia se mantiene independientemente del tamaño de bucket

que se le asigne a cada VM. Sin embargo, debe tenerse en cuenta que un tamaño de bucket alto puede generar delays debido a que aumenta el tiempo de refresco del sistema.

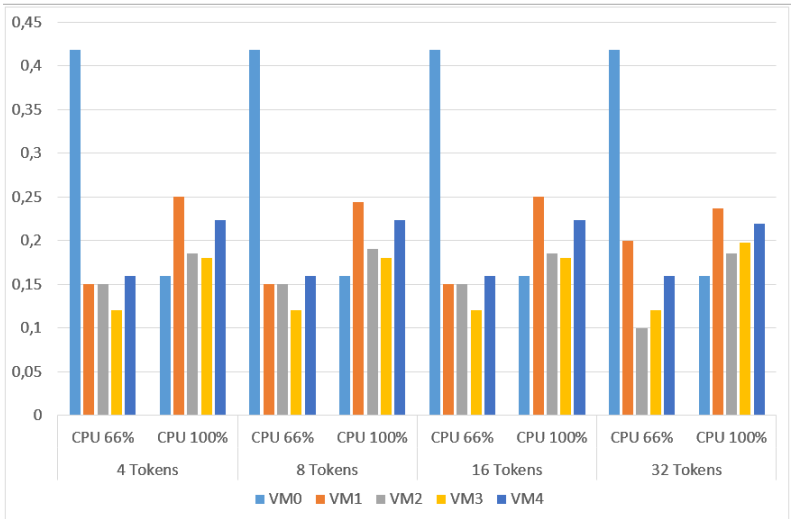

Figura 6. Distribución de CPU.

5. Conclusiones y trabajos futuros

Con este trabajo, se pudo apreciar la complejidad que implica la planificación de VMs y sus procesos, dada la cantidad de factores que se deben tener en cuenta. El trabajar con un sistema como Minix, ayuda a una mejor comprensión de cómo llevar a cabo esta planificación.

Cabe destacar que la utilización de un software de simulación apropiado brinda la posibilidad de realizar un estudio previo a la implementación, de forma simple y versátil. De esta manera, se pueden probar diferentes algoritmos, realizar comparaciones y encontrar defectos de diseño en etapas tempranas del desarrollo.

La utilización de un algoritmo de tipo token bucket es una opción acertada para planificar el OS en cuestión, ya que combina de manera satisfactoria la sencillez de implementación y la efectividad. Es una alternativa que asegura la distribución equitativa de la CPU, y la ejecución de los procesos de usuario y servidores sin descuidar las tareas, que generalmente tienen rendimientos temporales. Además permite dar diferentes prioridades a las VMs a través de la asignación de diferentes tamaños de bucket.

El trabajo a futuro consiste en implementar el algoritmo presentado en este artículo para reemplazar el algoritmo de round-robin que actualmente está utilizando Mhyper.

Referencias

1. Tanenbaum, A. S., Woodhull, A. (2006). *Operating Systems: Design and Implementation - 3º edición*. Prentice Hall.
2. Tanenbaum, A. S. (2003). *Redes de Computadoras - 4º edición*. Prentice Hall.
3. Tanenbaum, A. S. (2009). *Sistemas Operativos Modernos - 3º edición*. Prentice Hall.
4. SimSo - Simulation of Multiprocessor Scheduling with Overheads. *http://homepages.laas.fr/mcheramy/simso/*
5. XEN Credit Scheduler. *http://wiki.xen.org/wiki/Credit_Scheduler*