udp Escuela de Informática y Telecomunicaciones
FACULTAD DE INGENIERÍA

Sincronización en Minix

Proyecto 4

Sistemas Operativos

Resumen

Este proyecto consiste en agregar un nuevo servicio a Minix. Para poder resolver el problema necesitan resolver dos tareas (que no pueden paralelizarse), y que deben hacerse secuencialmente. Deben de construir un servicio de semáforos funcional, antes de poder resolver el problema de sincronización. Deben de trabajar con su pareja para poder maximizar el tiempo de producción.

1. Servicio de semáforos

Noten que existen archivos en el kernel actual llamados `sem.c` y `sem.h`. Entonces, deberán de llamar a su servicio `sema` o `semaphore` para evitar algún conflicto.

Después de su proyecto 3 deben de estar conscientes de que la mayoría de la funcionalidad de Minix está implementada como un servicio conectados a las aplicaciones de usuario a través de mensajes. Como Minix no tiene una forma concurrente de sincronizar procesos, la primera parte de este proyecto consiste en agregar el soporte de semáforos a Minix a través de la implementación de dicho servicio. Éste debe de implementarse como un servicio del sistema similar al administrador de procesos o el sistema virtual de archivos. A pesar de que su sistema de semáforos necesita iniciar en tiempo de arranque (ver el manual de arranque), éste no debe de ser parte de la imagen de arranque —la imagen de arranque debe de ser para los servicios que se necesitan durante el arranque (recuerden lo aprendido en el proyecto 2).

El servicio necesita entender cuatro tipo de mensajes: `sem_init`, `sem_release`, `sem_up`, y `sem_down`. La semántica de cada mensaje es la siguiente:

- `int sem_init (int start_value)`: Esta función crea un mensaje `SEM_INIT` que es enviado al servicio de semáforos. El parámetro `start_value` especifica el valor inicial del semáforo. Puede ser un valor entero arbitrario. Típicamente, para un mutex (i.e., un semáforo binario), este valor debería de ser 1. Una vez el semáforo esté inicializado, se considerará activo. El servicio de semáforos que implementen debe de soportar un número ilimitado de semáforos, hasta que no haya más memoria en el sistema.

  Una llamada exitosa a esta función deberá de retornar el siguiente número de semáforo disponible empezando en `0`. Un fallo por falta de memoria deberá de retornar `ENOMEM` (ver kernel API). Todos los otros errores deberán de retornar un código de error apropiado.

- `int sem_down(int sem_num)`: Esta función crea un mensaje `SEM_DOWN` que es enviado al servicio de semáforos. El parámetro `sem_num` corresponde al número de semáforo con el que la función trabajará. Esta llamada puede ser invocada para un semáforo activo solamente. El llamar a la función en un semáforo inactivo resultará en un error `EINVAL`. Esta función implementa la semántica estándar de la función `p()` de un semáforo. Es decir, la llamada decrementa el contador correspondiente al semáforo `sem_num` por uno. Si el contador (después del decremento) es menor que cero, entonces el proceso que llamó a la función debería de dormir (esperar en una cola que corresponde al semáforo). Si la función fue exitosa, deberá de retornar `OK` o un `0`.

- `int sem_up(int sem_num)`: Esta función crea un mensaje `SEM_UP` que es enviado al servicio de semáforos. El parámetro `sem_num` corresponde al número de semáforo con el que la función trabajará. Esta llamada puede ser invocada para un semáforo activo solamente. El llamar a la función en un semáforo inactivo resultará en un error `EINVAL`. Esta función implementa la semántica estándar de la función `v()` de un semáforo. Es decir, la llamada incrementa el contador correspondiente del semáforo `sem_num` por uno. Cuando hay al menos un proceso esperando (dormido) en la cola del semáforo `sem_num`, el primer proceso que fue puesto en la cola deberá de ser despertado (i.e., se utilizará una política FIFO). En éxito, la llamada `sem_up` deberá retornar `OK` (valor `0`).

- `int sem_release(int sem_num)`: Esta función crea un mensaje `SEM_RELEASE` que es enviado al servicio de semáforos. El parámetro `sem_num` corresponde al número de semáforo con el que la función trabajará. Esta llamada puede ser invocada para un semáforo activo solamente. El llamar a la función en un semáforo inactivo resultará en un error `EINVAL`. El propósito de esta función es liberar un semáforo activo, y colocarlo en su estado inactivo. Esta función debe fallar con un error `INUSE`, una constante que deben definir en `errorno.h`, cuando existan procesos esperan-

do en la cola del semáforo `sem_num` que deben despertarse. Si la función fue exitosa, `sem_release` debe retornar `OK` (valor `0`).

Cada función de los semáforos debe de completarse en tiempo constante. Entonces, no es correcto el mantener una lista enlazada de semáforos y ejecutar una búsqueda en $O(n)$ para encontrar el que se busca. Deben de utilizar el conocimiento de estructuras de datos para soportar una búsqueda en tiempo constante. La única excepción es `sem_init`, en donde es correcto que de vez en cuando deban tomar más tiempo para agrandar su estructura de datos. Sin embargo, el número de llamadas a `sem_init` que son completadas en tiempo constante deberá doblarse cada vez que se hace crecer a la estructura de datos.

2. Implementación de llamadas al sistema

Agrega una nueva llamada al sistema (kernel) de Minix no es muy difícil. Involucra agregar funciones que envuelvan (*wrapper*) otro código en `/usr/src/lib/libc/sys-minix` para poder hacer la llamada, archivos cabecera en `/usr/src/include`, y un servidor en `/usr/src/servers` para recibir esas llamadas y poder manejarlas. Ustedes deberán de hacer los cambios necesarios a los archivos de construcción (`Makefile`s) para asegurar que el código que agreguen se compile con el sistema. Finalmente, ustedes deben definir nuevos tipos de mensajes y parámetros en el archivo correcto (`com.h`) para poder enviar los mensajes. Después de implementar el servidor ustedes deberán de establecerlo para que inicie al arrancar la máquina y acepte mensajes.

Como con otros componentes, deben de observar llamadas existentes y servidores en el sistema para entender como funciona el comportamiento de Minix. En particular, deben de prestar atención a las estructuras existentes para replicarlas y mantener un mismo ambiente.

Noten que este proyecto requiere más que solo realizar un `make hdboot`, ya que ustedes deberán cambiar código fuera del kernel también. Cuando realicen cambios al código del kernel deben de realizar un `make hdboot`, o reconstruir todo el ambiente, pero a menudo si están realizando cambios fuera del kernel (la parte del código a nivel de usuario) pueden realizar los cambios haciendo `make` en un directorio padre donde se encuentre el código.

Algunos archivos pueden no ser tomados en cuenta para la reconstrucción del kernel sino se reconstruye todo el ambiente o sino se establecen los parámetros de configuración correctos (en particular `/usr/src/etc/system.conf`). Entonces, si están cambiando uno de esos archivos, ustedes pueden sobrepasar la reconstrucción de todo el ambiente a través de la copia manual de los archivos necesarios a su respectivo lugar (es parte del proyecto el encontrar donde deben de moverse que archivos). Noten que esto funciona solamente para un pequeño número de encabezados y scripts del sistema, cualquier otro código fuente debe de ser compilado de manera normal. Ustedes pueden tratar de ejecutar `make includes etcforce` desde `/usr/src` y observar si eso instala los archivos que necesitan.

2.1. Información adicional

La clave para las llamadas al sistema es que los procesos del usuario deben de ser capaces de usarlas para enviar mensajes a su servicio de semáforos. Ustedes deben de revisar código existente en el sistema para ver como se implementa actualmente. Dado que el servicio de semáforos no se incluye en la imagen de arranque, no puede tener un número de proceso explícito. De tal forma, ustedes deben de encontrar el número del proceso del servicio de semáforos dinámicamente como parte de la llamada al sistema. Existe una función conveniente, `minix_rs_lookup`, que puede servirles de ayuda para resolver este problema. Una vez tengan agregado el código para las llamadas de sistema, deben asegurarse que el manejador de las funciones reciba los datos que están enviándole (e.g., el número del semáforo). Además, deben asegurarse que los archivos que ustedes agregaron están en los `Makefile`s correctos.

2.2. Implementación de un servicio de sistema

El servicio de semáforos requiere de dos cosas, hacer un servicio que inicie en el arranque, y colocar procesos a dormir y despertarlos en el momento apropiado. Para descubrir como construir un servicio, deben de revisar los ejemplos existentes, como por ejemplo, el servicio del sistema de calendarización (localizado en `/usr/src/servers/sched`). Si observan el código, verán rutinas de inicialización y de término del servidor, así como rutinas para obtener y manejar mensajes, y código para despachar mensajes a diferentes manejadores de rutinas. Todo este código se encuentra en `main.c`, pero el código que hace el trabajo real está en `schedule.c`. Ustedes pueden copiar el código en su propio directorio, por ejemplo `/usr/src/servers/semaphore`, y adaptarlo a sus necesidades. Lo más probable es que mantendrán el ciclo de despacho principal, pero deberán administrar los mensajes de los semáforos de manera distinta, y enviarlos a las rutinas de sus semáforos.

Ustedes deben definir sus mensajes en `/usr/src/include/minix/com.h`. Deben revisar como otros mensajes son definidos dentro de Minix. Us-

tedes deberán de establecer los permisos del servicio de manera correcta en `/usr/src/etc/system.conf` para permitir a los procesos de usuario el enviar los mensajes a su servicio de semáforos. Finalmente, ustedes deben de actualizar los distintos `Makefile`s para asegurarse de que su servicio de semáforos se construye e instala correctamente. Deben observar como otros servicios son implementados, y tracen donde aparecen esos archivos en los distintos `Makefile`s para ayudarlos a determinar donde deben hacer los cambios necesarios.

Para hacer que su servicio se ejecute cuando el sistema arranca, ustedes deben agregarlo a uno de los scripts de inicialización `rc`. Intenten encontrar donde otros servicios, como la `printer` o `random`, son iniciados para hacer algo similar con el suyo.

Una vez tengan la mayoría de su código servidor en un lugar, ustedes pueden evaluarlo para comprobar que inicia y apaga correctamente haciendo lo siguiente. Primero, construyan su sistema modificado. Luego de reiniciar su sistema, ustedes deberán de ser capaces de iniciar su servicio de semáforos a través de `/usr/sbin/semaphore` (o el nombre que escogieron), y deberán de poder detenerlo a través de `service down semaphore`.

2.3. Implementación de los semáforos

Ahora deberán de tener todas las piezas que permiten a los procesos de usuario enviar mensajes a su servicio. Prueben si un programa de usuario puede utilizar sus funciones (`wrappers`) en `libc` para enviar mensajes al nuevo servicio. Una vez tengan el componente de comunicación y las llamadas al sistema trabajando correctamente, entonces pueden preocuparse de la implementación de la funcionalidad de los semáforos.

Noten que existe mucha documentación sobre la implementación de servicios de semáforos (e.g., Herder et al. [1]). Según la implementación que sigan, o si deciden hacer una por su cuenta, deben explicitarlo en el informe y explicar lo que hicieron. Finalmente, deben asegurarse de realizar varias pruebas para revisar la funcionalidad de su servicio de semáforos, y que un proceso puede dormir en un semáforo utilizando `sem_down` y que puede despertarse utilizando `sem_up`.

3. Utilización de sus semáforos para resolver un problema de sincronización

Dado el servicio de semáforos que construyeron, ustedes deben de utilizar sus semáforos para resolver el siguiente problema de sincronización.

Los estudiantes de último año (U) y los estudiantes de sistemas operativos (S) han sido invitados a comer pizza. Sin embargo, los estudiantes de ultimo año molestan a los estudiantes de sistemas operativos, y no pueden estar juntos en la misma habitación. Se llegó a un acuerdo en que todos pueden comer pizza, siempre y cuando, no hayan un estudiante U y uno S en la misma habitación. Con el reducido presupuesto, logramos conseguir dos mesas donde se coloca la pizza y en donde solamente un estudiante puede comer a la vez. Si un estudiante está comiendo puede ver la otra mesa, pero cuando termina de comer deja la habitación. Obviamente, no puede dejar que ningún estudiante muera de hambre, así que debe de permitir que todos puedan comer. Entonces, dado que solo un estudiante puede comer en una mesa, solo dos U o dos S pueden compartir la habitación a la vez — ya que no queremos causar un estrés sicológico a los estudiantes S que son molestados por los U. Su trabajo es construir dos programas: uno que simule U y otro que simule S, y que utilice su servicio de semáforos para sincronizarlos. De tal manera que se cumpla que no puede haber un S y un U en la habitación simultáneamente.

Por simplicidad, asuma que existe una cantidad ilimitada de pizza de la cual pueden comer los estudiantes. Además, existen solamente seis U y dos S que asisten al evento. Y que los estudiantes tardan una cantidad aleatoria en comer. Deben explicar su solución en el informe.

4. Instrucciones

4.1. Trabajo a realizar

1. Este proyecto se desarrollará en parejas.

2. Lean detalladamente las instrucciones a realizar (pueden que necesiten más de una pasada para entender y hacer funcionar todo). En especial, noten la guía de desarrollo para la construcción de la solución (revisen los comentarios de la Sección 2.3).

3. Necesitarán más información que los enlaces provistos en este enunciado. Busquen información mientras resuelven los problemas. Y más importante, documéntenla para explicarla en su informe.

4. Deben de generar un parche de los cambios realizados, y evaluar su buen funcionamiento. El parche debe de poder aplicarse exitosamente en un ambiente nuevo y sin ningún error. El parche debe crearse a través de la ejecución de

   ```
   diff -ruNp minix_src_clean/ proj/ > patch
   ```

4.2. Entregables

Deben de entregar una carpeta con la siguiente estructura

- `codigo`: carpeta que contiene todos los parches que definen su proyecto, y el código fuente de la definición de los estudiantes (`u.c`, `s.c`, y un `main.c` que permita simular el problema discutido).
- `informe`: el código fuente para generar el informe que presentan.

Estas carpetas deberán de ser comprimidas en un solo archivo, y subido al sitio del curso junto con un `PDF` del informe. El archivo comprimido deberá ser nombrado utilizando los RUTs de los integrantes siguiendo el patrón `rut-rut`, donde `rut` es el RUT de los integrantes sin puntos ni guión. Por ejemplo, si los integrantes tienen RUT `1234-5` y `6789-0` el archivo comprimido deberá llamarse `12345-67890`.

4.3. Fecha de entrega

La fecha de entrega es el día 03 de julio. En horario de clase deberán de entregar el informe impreso. Y deberán subir los archivos al sitio del curso antes de la entrega impresa. Noten que la entrega digital de los documentos cierra al inicio de la hora de la clase, por lo que deberán subir sus archivos antes de la clase.

Se les recomienda el subir los archivos antes de la hora límite para evitar problemas con el servidor. No se considerarán correos que se envíen horas antes de la fecha límite de la entrega con problemas subiendo los documentos.

Se les recuerda nuevamente que el curso tiene una política de *tolerancia cero* para las entregas tardías.

4.4. Sobre el informe

- Deben preparar un informe de **máximo 4 páginas** de contenido (éstas no incluyen las figuras o referencias) utilizando `IEEEtran.cls` y la guía entregada en clase.
- El informe debe de contener al menos:
  - Resumen
  - Introducción (explicando el contexto del trabajo realizado)
  - Deben desarrollar los siguiente tópicos
    - Explicar su servicio de semáforos (análisis y diseño)
    - Explicar la implementación de las llamadas al sistema (cómo realizaron las llamadas)
    - Explicar la implementación de semáforos (referencias y cómo aseguran que garantizan la atomicidad y que son atómicos)
    - Solución al problema de sincronización
  - Conclusión
- **Si se detecta plagio ustedes obtendrán un uno en el proyecto**, y se dará aviso a las autoridades correspondientes para que se tomen las sanciones del caso.

4.5. Notas de interés y restricciones

- Su informe debe de estar realizado siguiendo la guía de documentos técnicos discutida en clase. Recuerde que se penalizará cada falta de ortografía y de redacción con 1 % de la nota final hasta un 20 %.
- No se aceptarán entregas fuera de la hora de entrega. Esto incluye copia digital fuera del plazo, o copia impresa fuera del horario de clase. Además, no se considerará una entrega completa sino existe la entrega digital y la física.
- Para evitar problemas al subir sus archivos (por ejemplo, problemas con el tamaño de la entrega, o disponibilidad del servidor) se les recomienda el subir los archivos antes de la hora de entrega. Noten que la hora de entrega no es la hora para empezar a subir los archivos, sino un límite para tener un punto de corte. Programen su entrega y háganla antes de la fecha límite.
- En caso se sobrepase el límite de páginas estipulado, se evaluará lo presentado hasta el límite solicitado. Para evitar problemas, sigan las instrucciones y sean concisos.
- El parche que entreguen debe de contener las lineas que sean necesarias para el desarrollo del proyecto. No generen cambios innecesarios que entorpezcan la revisión del mismo.
- Dada la complejidad del código que se puede generar en este proyecto, se recomienda el entregar un archivo `README` comprensivo, que contenga información necesaria para la compilación de su código. Por ejemplo, dependencias que deben existir en la computadora que compila, versione de las distintas herramientas (gcc, ld, nasm, etc.), e instrucciones detalladas para la compilación y ejecución de su aplicación. Si construyeron scripts para la instalación deben de agregarlos en la carpeta `codigo`.
- Se recomienda que utilicen un manejador de versiones para mantener su código (por ejemplo, git).

Tabla 1: Ponderación del proyecto.

	Informe	Práctica	Subtotal
Resumen	2		2
Introducción	5		5
Servicio de semáforos	7	15	22
Llamadas de sistema	7	10	17
Imp. semáforos	6	10	16
Prob. sincronización	10	15	25
Conclusiones	3		3
Parche		10	10
Total	40	60	100

- Usted debe de hacerse responsable de que su código compile, enlace (link), y se ejecute correctamente en cualquier ambiente (se calificará en un ambiente Unix con los emuladores `bochs` o `qemu`).

5. Tips

- Este proyecto puede parecer mucho más sencillo que los proyectos anteriores. Sin embargo, no se engañen por que lo que deben de hacer es la punta del iceberg. Deben de familiarizarse con Minix y con sus archivos y funcionalidades.
- Revisen la documentación oficial de Minix para poder resolver problemas —que por lo general serán comunes, y les han pasado a otras personas antes que ustedes.
- Establezcan metas iniciales y etapas para avanzar en el proyecto. Por ejemplo, definir interfaces, implementar llamadas al sistema, implementación del servicio, y por último la implementación de los semáforos. Además, por su experiencia anterior en los otros proyectos, **no empiecen el proyecto días antes de la entrega**. Planifiquen su entrega y realicen avances con tiempo.

6. Ponderación

El proyecto está ponderado 60 % de práctica y 40 % del informe (esto incluye escritura, estructuración de ideas, redacción, etc.) que será evaluado a través del informe escrito, y del código entregado (parche). La evaluación será general y se evaluarán los aspectos teóricos, prácticos, y la presentación (a través del informe) en conjunto. El detalle de la ponderación está en la Tabla 1.

Referencias

[1] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Modular system programming in minix 3." [Online]. Available: https://www.usenix.org/legacy/publications/login/2006-04/openpdfs/herder.pdf