Proyecto Sistemas Operativos

Franco Sassola

Julian Gallardo

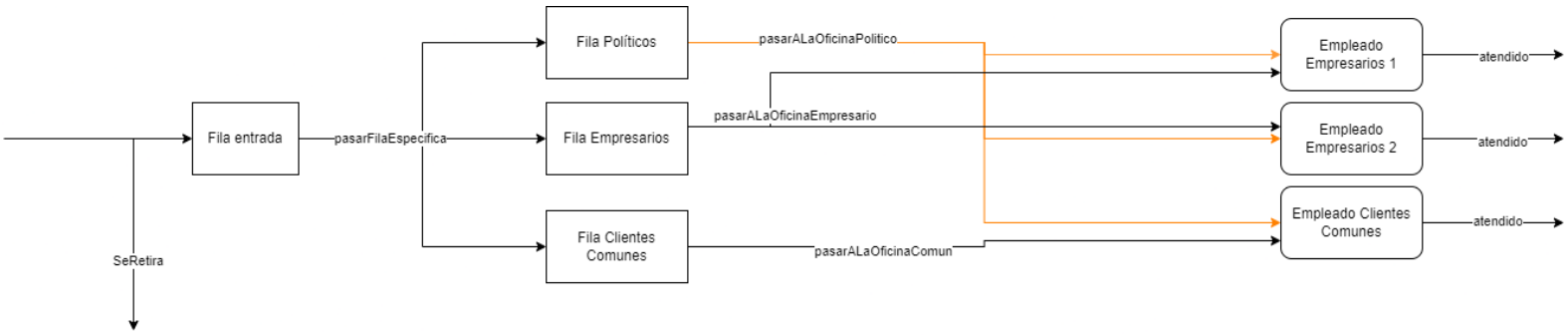
Índice

Experimentación de Procesos y Threads con los Sistemas Operativos

Banco con threads y semáforos

Dado el problema del banco, nos piden implementar un código que simule el funcionamiento de un banco, con una fila de entrada de 30 personas, y 3 filas para cada tipo de cliente, empresario, políticos y usuarios comunes, donde cada una de estas filas tiene un máximo de 15 personas.

La idea del modelo implementado es modelar todas las acciones que se realizan en este circuito como en la imagen de abajo.



La idea consiste en que en el modelo los clientes están representados como hilos, al igual que los empleados, por lo que tenemos n hilos clientes y 3 empleados, donde tenemos 2 empleados especializados para clientes de empresa y 1 para clientes comunes. Cada tipo de cliente es creado al azar, donde el político tiene una posibilidad del 25% de aparecer, luego el empresario y el cliente común tienen cada uno 40% y 35%, usando un sistema de tickets para repartir los porcentajes.

Luego tenemos que modelar los pasos del inicio del diagrama, para esto contamos con 4 semáforos de fila, llamados fila_entrada, fila_Políticos , fila_Empresa y fila_ClienteComun, donde cada uno de ellos están inicializados con los valores especificados por el problema. Luego vemos como cada acción de ingreso a las filas es modelada:

- La acción de ingresar al cliente a la fila de entrada está modelada con un try wait sobre el semáforo fila_entrada donde chequeamos si hay lugar en la fila de entrada, en caso de haber tomamos un lugar dentro de ella, decrementando el semáforo, en caso contrario el cliente se retira y termina el hilo.
- Para que el cliente ingrese a su fila específica debemos hacer un wait sobre el tipo de su fila, por ejemplo, si estamos ante el caso de un cliente de tipo común, debemos de esperar a que se libere un espacio en fila_Cliente_Común. Una vez ingresada a la fila específica del cliente le realizamos un post al semaforo que indica la cantidad de clientes de ese tipo esperando, hay uno para cada caso llamados cantPoliticosEsperando, cantEmpresasEsperando y cantComunesEsperando, veremos más adelante para que se utilizan.

Una vez que tenemos visto cómo se modelan los ingresos a las colas por parte de los clientes, tenemos que ver cómo se modela la llamada de los empleados a los clientes, teniendo en cuenta que los semáforos son FIFO, es decir siempre el primer hilo que toma el wait va a ser el primero en atenderse, tenemos que todos los clientes esperan el llamado para pasar a la oficina del empleado específico que los atiende, modelado con los semáforos pasarALaOficinaEmpresario, pasarALaOficinaPolitico y pasarALaOficinaComun. Pero antes de esperar por el llamado del empleado, el cliente consulta si el empleado que lo tiene que atender no está dormido, tomando los mutex correspondientes para que garantice la exclusión mutua, ya que podemos tener el caso donde tenemos un cambio de contexto y nosotros ya habíamos pasado la consulta y otro hilo lo despierta, para posteriormente nosotros despertarlo otra vez, lo cual sería incorrecto.

Dormimos los empleados ya que sino con ellos estaríamos realizando constantemente una espera ocupada, lo cual no queremos, ya que desperdiciamos ciclos de la cpu en eso. Una vez comprendido esto veamos como están modelados las acciones del empleado:

1. El primer paso que realiza cualquier empleado es consultar si hay políticos para atender, realizando un try wait sobre CantPoliticosEsperando y con un mutex para asegurarnos de que ningún otro empleado pueda realizar esa consulta concurrentemente. ¿Por qué consultamos primero sobre los políticos en vez de nuestros correspondientes clientes? Esto se debe a que los políticos los puede atender cualquiera de los 3 empleados y tienen prioridad. Una vez realizado el chequeo de si hay políticos, en caso de haber lo atendemos, en caso contrario vamos a realizar la misma consulta con un try wait sobre el tipo específico que atiende el empleado, ya sea sobre CantEmpresasEsperando o sobre CantComunesEsperando, en caso de tener clientes lo atendemos y liberamos su lugar en la fila, avisando que pueden pasar a la oficina.
2. Luego modelamos con sleeps el tiempo que conlleva cada paso y por último una vez que terminamos de atender, mandamos la señal de que fue atendido, con el semáforo atendido.
3. Una vez que realizamos las consultas para atender clientes , como vimos ambos modelados con try waits, tenemos que hacer un algoritmo como el del barbero para que no tengamos espera ocupada, ¿En qué consiste esto? en preguntar si hay clientes de nuestro tipo específico, ya sean clientes comunes o de tipo empresa esperando, y también consultar si hay políticos esperando, cada uno con su correspondiente try wait y semáforo de cantidad.
    a. En el caso que tengamos que no hay ningún cliente político, ni de nuestro tipo esperando, el empleado se duerme, y manda la señal de que está dormido con el semáforo de su tipo, ya sea empleadoComunDormido o empleadoEmpresaDormido, para luego esperar la señal para ser despertado.
    b. En el caso de que tengamos algún tipo de cliente esperando, volvemos a empezar desde el principio.

    Una vez que el empleado manda la señal de que el cliente fue atendido, el cliente se retira.

Banco con procesos y colas de mensaje

Luego para el siguiente punto se nos pide realizar el mismo experimento pero usando procesos y colas de mensajes, en este caso para modelar los clientes y los empleados usamos procesos, donde vamos a tener n procesos clientes y 3 procesos de empleados, similar al modelo anterior, donde tenemos 2 empleados que atienden a empresas y 1 empleado que atiende a clientes comunes, ambos pueden atender a los clientes políticos con prioridad. Debido a esto el modelo es igual que el explicado en la sección anterior salvo que en este caso en vez de tener semáforos, utilizamos una cola de mensajes con tipos de mensajes para simular los semáforos.

Para este caso tenemos los receive bloqueantes son equivalentes a un wait de semáforo, los receive no bloqueantes son equivalentes a los try wait de los semáforos y luego los post de los semáforos son los send de mensajes bloqueantes, estos tienen que ser bloqueantes ya que sino nos pueden causar problemas de que los mensajes no se envían.

En si el modelo solo varia en eso, no hubo muchos más cambios, no usamos pipes ya que con las colas de mensajes se permiten simular operaciones de forma más sencilla, como es el caso del try wait que con pipes no queda tan facil de entender cómo con una cola de `mensaje`.

Ventajas y desventajas de las implementaciones

Si comparamos ambas propuestas nos damos cuenta que la de los hilos tienen ventaja, ya que al tener que los clientes y los empleados son hilos dentro de un mismo proceso son más livianos en carga para el sistema operativo, ya que no es lo mismo tener 80 hilos de clientes que 80 procesos de clientes, donde cada uno tiene su propio contexto. Además de que en la primera propuesta usamos los semáforos que son una herramienta hecha para sincronización, en cambio en la propuesta de los procesos tenemos que emular a estos con los tipos y mensajes de la cola de mensajes, lo que puede resultar en un código más difícil de entender, aunque esto se podría solucionar usando semáforos en memoria compartida.

Mini Shell

Luego se nos pide implementar una shell que acepte un conjunto de comandos limitados de Unix, donde se nos obliga a usar las system calls o funciones de librería.

Esta shell tuvo varias ideas, donde por ejemplo implementabamos todos los comandos necesarios como funciones del mismo archivo, pero nos dimos cuenta que hay una manera mas sencilla y más extensible para realizar una shell, esta es hacerla como la implementa el s.o donde tenemos que cada vez que ingresas un comando se crea un proceso que intenta ejecutar ese comando pasado cargando una imagen con el nombre del comando pasado y sus argumentos, para luego ejecutarla.

En nuestra shell modelamos que cada vez que se ingresa un comando, este se separe el string obtenido por la consola en el comando y sus argumentos. Luego ver si no es un comando como el exit, que termina la shell, o un comando invalido, ya sea un comando vacío(donde apretamos enter en la shell sin ingresar nada) o un comando invalido. En caso

de que no sea ninguno de estos ni un cd o pwd que veremos más adelante, tenemos que es un comando válido, por lo tanto forkeamos un nuevo proceso y le ponemos la imagen del comando dado, pasándole sus argumentos. De esta forma emulamos el comportamiento del bash y queda una shell extensible para agregarle todos los comandos que quisiéramos en un futuro. Veamos qué comandos tiene implementada nuestra shell:

1. help : Este comando es una ayuda al usuario que muestra todos los comandos disponibles y sus argumentos. No tiene ninguna llamada al sistema.
2. mkdir: Este comando crea un directorio en el directorio actual de la shell, para realizar esto le debemos de pasar un nombre para ponerle al nuevo directorio o se crea uno con un nombre default. Para este comando se utilizaron las llamadas al sistema opendir, para abrir el directorio actual de la shell y createDir("Nombre directorio") para crear el nuevo directorio.
3. rmdir: Este comando remueve un directorio, con el nombre pasado por parámetro, para esto el directorio tiene que estar vacío. Luego se utilizó la misma llamada al sistema openDir para abrir el directorio actual y la llamada removeDir("Nombre directorio") para eliminar el directorio deseado.
4. touch: Este comando crea un archivo con el nombre indicado por parámetro, utilizamos la llamada fopen.
5. ls: Este comando muestra todos los archivos en el directorio actual, se usó la llamada al sistema opendir para abrir el directorio actual y readdir para leer los archivos.
6. cat: Este cat no es el mismo cat que el integrado con la shell de raspbian, ya que el cat raspbian es un concatenador de archivos, pero en este caso solo le pasamos un parámetro, que es el archivo que queremos mostrar su contenido por consola. Usamos la llamada fopen para abrir el archivo que queremos leer.
7. chmod: Este comando sirve para modificar los permisos de acceso de un archivo, donde tenemos que ingresar el nombre del archivo y los permisos nuevos, estos tienen que estar en octal. Usamos la llamada chmod donde le pasamos el nombre del archivo y los permisos. Estos permisos deben ser de la siguiente forma:
    a. 0 = --- = sin acceso
    b. 1 = --x = ejecución
    c. 2 = -w- = escritura
    d. 3 = -wx = escritura y ejecución
    e. 4 = r-- = lectura
    f. 5 = r-x = lectura y ejecución
    g. 6 = rw- = lectura y escritura
    h. 7 = rwx = lectura, escritura y ejecución

Luego siempre debemos de indicar 3 números ya que el primero modifica el owner(propietario), que es la persona que el sistema reconoce como dueño del archivo, el siguiente número es el grupo, que es el conjunto de usuarios con permisos similares y por último los permisos para cualquier usuario, de forma tal que la manera de ejecutar el

comando si le queremos dar todos los permisos a todos es la siguiente:

chmod <nombre_archivo> 777

Luego todos estos comandos están contenidos en archivos separados para que cuando sean llamados por la minishell, se ejecute su imagen. En cambio los comandos cd y pwd están implementados en la misma minishell, ya que el cd necesita cambiar el directorio del proceso justamente de la minishell, no tendría sentido ejecutarlo en un proceso hijo ya que le estaríamos cambiando el directorio a el hijo. Debido a esto la minishell siempre mantiene el directorio inicial, donde están contenidas todas las imágenes de los comandos nombrados anteriormente. Para cambiar el directorio de la mini shell se usa la llamada al sistema chdir. Por último el pwd es para mostrar por pantalla el directorio actual de la mini shell, se utiliza la llamada getcwd().

Sincronización

Secuencias

La siguiente sección trata sobre sincronizar secuencias de letras, donde dadas las siguientes secuencias:

- ABABCABABCABABC
- ABABCABCDABABCABCD

Debemos resolver la sincronización usando hilos y threads por un lado y por el otro usar procesos y pipes para su sincronización. Primero veamos ambas secuencias resueltas usando hilos y threads.

Hilos y threads

Para solucionar el problema con hilos y threads , tenemos que cada letra de las secuencias son un hilo, por ejemplo en la primera secuencia ABABCABABCABABC, tenemos un hilo para A, otro para B y por último uno para C. Básicamente la idea consiste en tener 3 semáforos donde tenemos semA, semB y semC. Inicializamos los semáforos semA en 1 y los demás en 0. Luego cada hilo espera por su semáforo, comenzando por el hilo A:

1. El hilo A hace wait(semA), luego imprime y manda post(semB).
2. El hilo B hace wait(semB), recibiendo la señal del anterior hilo, imprime, y hace post de semA y semC.
3. El hilo C hace dos wait(semC), ya que necesita que se repitan dos veces los pasos 1 y 2 para ejecutarse.

De esta forma nos queda la secuencia deseada. Para el caso de la secuencia ABABCABCDABABCABCD, la secuencia sigue los mismos pasos que la anterior, agregando un hilo para la letra D, pero con modificaciones en la secuencia de la letra C, ya que ahora seguimos esperando dos wait(semC), imprimimos y volvemos a esperar otro wait(semC), para imprimir y posteriormente hacer un post(semD), luego D imprime y se reinicia la secuencia.

Procesos y Pipes

Luego a comparación de la anterior solución, la lógica sigue siendo la misma, salvo que en vez de hilos usamos procesos, es decir, que para cada letra es un proceso. Luego para la comunicación entre procesos y su sincronización utilizamos los pipes, donde con ellos simulamos el trabajo de los semáforos explicados en la anterior sección. Básicamente hicimos 2 operaciones llamadas sem_Envia y sem_Espera donde sem_Envia es un write en el pipe, y el sem_Espera es un read del pipe, ambas operaciones bloqueantes, de esta forma logramos emular lo que realizamos con hilos y semáforos.

Reserva de Aulas

Llegando al último problema de implementación tenemos que resolver el problema de las reservas de aulas, donde tenemos 25 alumnos que pueden reservar periodos de 1 hora desde las 9 am hasta las 9 pm. Este alumno puede realizar 3 acciones que son reservar, cancelar y consultar, donde cada alumno debe realizar 4 de estas acciones al azar, con una probabilidad del 50% de que la acción sea una reserva y las demás con un 25% cada una.

Para esto tenemos que cada alumno es un hilo, donde cada hilo realiza 4 acciones y cada una de ellas tiene la probabilidad dicha. Para asignar correctamente la probabilidad usamos un sistema de tickets donde se le dan 2 tickets a la reserva, y 1 a la consulta y otro a la cancelación, luego se elige un número al hacer entre 0 y 3 y se ejecuta esa acción.

Para realizar la reserva, se debe hacer el acceso y modificación de la tabla en sección crítica, para esto debemos tomar el mutex, así de esta forma nos aseguramos que solo un hilo esté accediendo a la tabla. Para la cancelación del horario tenemos que es de la misma forma, en ambos casos los horarios para reservar y cancelar son al azar, aunque la cancelación se podría implementar de forma tal que intente cancelar solo sus horarios reservados, pero este no es el caso.

Luego por último para la consulta se utiliza un algoritmo similar al de lector escritor, donde tenemos n consultas concurrentes. Para esto debemos consultar si ya se están realizando consultas, en caso de que no se esté realizando ninguna, nos quedamos esperando a tomar el acceso a la tabla de reservas y una vez tomado, indicamos que se está realizando una consulta sumandonos al semáforo cantConsultas, de esta forma podemos realizar las consultas concurrentemente. Una vez que terminamos de realizar la consulta, vemos si no somos la última consulta que queda, en caso de no ser la última no hacemos nada, pero si somos la última devolvemos el acceso a la tabla de reservas para que se puedan realizar las demás acciones. Como dijimos anteriormente, es muy similar al algoritmo de lector escritor.

A causa de que los randoms son con la seed del tiempo, puede caer el caso que queden algunas consultas con números iguales, pero no influye en el comportamiento. Por último se nos pide resolver el problema con procesos y memoria compartida, el modelo sigue siendo el mismo pero en vez de tener 25 hilos tenemos 25 procesos y debemos acoplarnos a la memoria compartida con shmat, luego ponemos en memoria compartida los semáforos y la tabla de reservas y sigue funcionando de la misma manera que el explicado con hilos.

Compilación de archivos

Todos los archivos tienen un make, para compilarlos correctamente debemos de ejecutar este make indicando make <nombre_archivo>. Por ejemplo queremos compilar el problema de las reserva de aulas con hilos, debemos hacer make ReservaAulasHilos, obviamente en la ubicación del makefile correspondiente al punto que se quiere ejecutar.

Este makefile se encarga de compilar los .c, ejecutar el programa y posteriormente eliminar los archivos compilados.

Problemas Conceptuales

Ejercicio 1

Inciso A

Tenemos un espacio de memoria física de 2GB, distribuido en páginas de 8 KB. Luego el espacio de direcciones lógicas de cada proceso se limitó a 256MB.

Entonces tenemos que para el número total de bits de la dirección física tenemos calcular los bits de dirección al número de frame y el desplazamiento, para esto tenemos que saber primero la cantidad de páginas que tenemos.

$$2GB \ = \ 2 \ x \ 1024 \ x \ 1024 \ = \ 2.097.152$$

Luego tenemos que calcular la cantidad de frames:

$$2.097.152 \ KB \ / \ 8 \ KB \ = \ cantidad \ de \ frames$$
$$262.144 \ = \ cantidad \ de \ frames$$

Como tenemos 262.144 frames tenemos que poder direccionar este número de frames, para esto calculamos el logaritmo base 2 de 262.144

$$log_2 (262.144) \ = \ 18 \ bits$$

Por lo tanto tenemos 18 bits para el número de frames. Ahora veamos cuantos bits tenemos para el desplazamiento:

Para calcular el desplazamiento tenemos que saber cuanto es el tamaño del frame, luego en base a este asumimos un direccionamiento a byte, de manera que nos queda el siguiente desplazamiento :

$1\ KB\ =\ 1024\ bytes$
$8\ KB\ =\ 8\ x\ 1024\ =\ 8192\ bytes$

$log_2(8192)\ =\ offset$
$13\ =\ offset$

Luego tenemos 13 bits de offset, por lo tanto la dirección física tendría un total de
$18\ bits\ de\ número\ de\ frame\ +\ 13\ bits\ de\ offset\ =\ 31\ bits$

Inciso B

Los bits que determinan la sustitución de página son:

- El bit de valido-invalido, que en el caso de ser **válido** indica cuando una página asociada está en el espacio de direcciones lógicas del proceso, por lo tanto es una página legal. En el caso de ser **invalido** indica que la página no está en el espacio de direcciones lógicas del proceso.
- El bit de modificado es aquel dice si un frame fue modificado y debe ser cargado en memoria.
- El bit de referencia, sirve para indicar cuando la página fue referenciada, si la página es referenciada se pone en 1, luego dependiendo del algoritmo de reemplazo es puesta en 0 otra vez. Por ejemplo en el algoritmo LRU tenemos que la página que hace más no es referenciada es puesto en 0.

Luego los números de bits que especifican el marco de la página son 18 bits, calculados en el ejercicio anterior.

Inciso C

Tenemos un espacio de memoria física de 2GB, distribuido en páginas de 8 KB. Luego el espacio de direcciones lógicas de cada proceso se limitó a 256MB.

Entonces tenemos que para el número total de bits de la dirección física tenemos calcular los bits de dirección al número de la página y el desplazamiento, para esto tenemos que saber primero la cantidad de frames que tenemos.

$2GB\ =\ 2\ x\ 1024\ x\ 1024\ =\ 2.097.152$

Luego tenemos que calcular la cantidad de frames:

$2.097.152\ KB\ /\ 8\ KB\ =\ cantidad\ de\ frames$
$262.144\ =\ cantidad\ de\ frames$

Inciso D

El formato de la dirección lógica es el siguiente:

Número de página	Offset

Calculemos la cantidad de bits para cada una, sabemos de los incisos anteriores que
$offset = 13\ bits$

Luego calculemos los bits necesarios para direccionar el numero de pagina:

Tenemos que cada proceso tiene un tamaño de direcciones lógicas de 258 MB
$1\ MB = 1024\ KB$
$258\ MB = 258\ x\ 1024\ KB = 264.192\ KB$

$264.192\ KB\ /\ 8\ KB = cantidad\ de\ páginas$
$33.024 = cantidad\ de\ páginas$
$log_2(33.024) = 15,011$
Luego tenemos que para el número de la página se usan 15 bits y para el offset 13, por lo que para la direccion logica en total usamos :

$dirección\ lógica = 15\ bits\ de\ número\ de\ página + 13\ bits\ de\ offset = 28\ bits$

Ejercicio 2

$dirección\ física = dir\ inicial + offset$

a) Se accede al segmento 0, luego tenemos que la dirección inicial del segmento es 830, por lo tanto:

$dirección\ física = 830 + 228 = 1058$
Luego se puede acceder sin fallos.

b) Se accede al segmento 2, por lo que comenzamos desde la dirección 648
$dirección\ física = 1508 + 648 = 2156$

Pero como el offset es más grande que el largo del segmento, se produce un fallo de segmento.

c)Se accede al segmento 3, luego tenemos que la dirección inicial del segmento es 770, por lo tanto:

$$dirección\ física\ =\ 770\ +\ 776\ =\ 1546$$
Luego se puede acceder sin fallos.

d)Se accede al segmento 1, luego tenemos que la dirección inicial del segmento es 648, por lo tanto:

$$dirección\ física\ =\ 648\ +\ 98\ =\ 746$$
Luego se puede acceder sin fallos.

e)Se accede al segmento 1, por lo que comenzamos desde la dirección 648
$$dirección\ física\ =\ 648\ +\ 240\ =\ 888$$
Pero como el offset es más grande que el largo del segmento, se produce un fallo de segmento.