# INFORME PROYECTO SISTEMAS OPERATIVOS-2023

Aravena Joaquín – De Battista Agustín – Comisión 15

# ÍNDICE

# 1.1 Procesos, threads y Comunicación

## Banco

La implementación del banco simula la llegada de clientes de diferentes tipos, cuya elección es al azar. En primer lugar pasan por una mesa de entrada, la cual tiene una capacidad máxima, y a partir de allí son distribuidos hacia 3 filas, una por cada tipo las cuáles también cuentan con capacidad limitada, en la que esperan a ser atendidos. Al finalizar su atención, se retiran. Además, se encuentran disponibles empleados, todos ellos pueden atender a clientes de tipo político, que además deben ser tratados con prioridad, y luego cada uno se dedica a un tipo específico, ya sea clientes comunes o empresas. Estos empleados verifican si tienen trabajo y si no descansan hasta ser despertados por el arribo de un nuevo cliente.

Cada una de estas capacidades fue modulada con constantes definidas para mantener un código flexible.

a) La implementación de los clientes y empleados es con un hilo por cada uno, la exclusión mutua en la mesa de entrada es modelada con un mutex y cada una de las 3 filas es implementada con un semáforo. Los empleados también son sincronizados con un mutex para no permitir que un cliente sea atendido por más de un empleado. También, se utilizan 3 semáforos que marcan que cada tipo de cliente está siendo atendido, y luego otro general para marcar que el cliente terminó de ser atendido, asumiendo que la implementación de los semáforos es FIFO como fue aclarado por la cátedra.
b) La implementación de los clientes y empleados es con un proceso por cada uno. El modelo de la implementación sigue siendo el mismo. Simulamos el “trywait” de un semáforo utilizando un receive no bloqueante, y comparando el resultado retornado. El “wait” de un semáforo fue modelado con un receive bloqueante, y el “signal” con un send no bloqueante. Los mutex fueron tratados como semáforos también. De esta forma mantuvimos el mismo modelo simplificando los cambios realizados.
   Creamos una cola de mensajes compartida entre todos los procesos, y un tipo de dato a mandar por cada semáforo o mutex que necesitamos.
c) Pudimos identificar que la solución utilizando hilos, es mucho más rápida en terminar de ejecutar, ya que los hilos son menos costosos de crear y terminar que los procesos. Además, los semáforos son fáciles y sencillos de utilizar e implementar.
   Por otro lado, la solución utilizando procesos y colas de mensajes resulta en más protección de memoria y recursos, ya que los procesos no comparten el mismo espacio de direccionamiento entre sí, y un error no afecta directamente a otro proceso, lo cuál si ocurre con los hilos.
   Como principal desventaja de utilizar procesos es la sobrecarga

generada por la creación, terminación y comunicación entre ellos. En cuánto a las colas de mensajes, tienen un desempeño similar al de los semáforos pero en términos de legibilidad de código puede resultar confuso.

Como conclusión, ambas soluciones permiten modelar el problema, cada una con sus ventajas y desventajas, pero al momento de inclinarnos hacia una de ellas preferimos implementarla con hilos y semáforos, principalmente por la rapidez y facilidad de implementación.

Compilación: dirigirse a la carpeta en la Shell, ejecutar el comando “make banco_proc” o “make banco_sem_hilos” para cada punto correspondiente.

## Minishell

Compilación: para compilar la Shell, se requiere ejecutar el comando:

- make

Este comando “make”, compila todos los archivos necesarios, comienza a ejecutar la Shell y luego de finalizar limpia todos los archivos que ya no sean necesarios.

Comandos:

- mkdir: se encarga de crear un directorio en la ubicación donde se encuentra la minishell. Si se pasa más de un argumento, crea únicamente el primer directorio.
- rmdir: remueve un directorio en la ubicación donde se encuentra la minishell. Si se pasa más de un argumento, elimina únicamente el primer directorio.
- touch: crea un archivo en el directorio en donde se encuentra ubicado. Si se especifica más de un nombre, crea tantos archivos como argumentos fueran pasados.
- ls: lista el contenido del directorio en donde se encuentra ubicado.
- showcontent: muestra el contenido del archivo especificado. Si se especifica más de un argumento únicamente muestra el contenido del primero.
- help: muestra los posibles comandos a ejecutar con sus parámetros.
- chmod: modifica los permisos de un archivo en específico. Donde:
    - w: permiso para escribir.
    - r: permiso para leer.
    - x: permiso para ejecutar.
    - Si no, se pueden especificar permisos en octal

        - En caso de pasar un permiso no válido, este comando no tiene efecto
- clear: limpia la consola.

# 1.2 Sincronización

## Secuencias

a) Cada secuencia fue implementada en su respectivo archivo, utilizando un hilo por cada función que imprima una letra. Cada una de estas funciones cicla infinitamente, al final del “main” en cada archivo agregamos la aclaración de que como los hilos nunca terminarán por consigna, no se llegará a ejecutar nunca cada uno de los “join(thread)”. Buscamos simplificar el código lo más posible,

Cabe aclarar que la secuencia “ABABCABCD” podría haber sido modelada con la utilización de un único “print(C)”, pero llevaría a una secuencia de waits y signal entre los 4 semáforos que no creemos que sea justificada por la simplicidad de la solución propuesta.

b) Seguimos el mismo modelo de implementación que el primer inciso, cambiando la utilización de hilos por procesos, y los semáforos por pipes. Mantuvimos la convención de cerrar todos los pipes que no sean utilizados en cada función, para asegurar una mayor robustez frente a situaciones anómalas. Además, limpiamos el buffer de escritura y lectura cada vez que se hace uso de un “read”, para evitar que quede basura sin leer y prevenir todo tipo de errores.

Al igual que en el inciso a), los procesos hijos nunca terminarán de ejecutar ya que ciclan indefinidamente, por lo que la espera por los hijos

Compilación: dirigirse a la carpeta en la Shell, y ejecutar “make primera_sec_hilos”, “make primera_sec_proc”, “make segunda_sec_hilos”, “make segunda_sec_proc”

## Reserva de aulas

Resolvimos el problema utilizando un hilo para cada alumno, el cuál ejecuta la función “decidir” cuatro veces, modelando la elección aleatoria de las 4 operaciones. Para el acceso a la tabla de reservas se garantiza exclusión mutua utilizando un semáforo binario. Las reservas son siempre de 1 hora, es decir que la primera reserva posible es a las 9hs, y la última es a las 20hs.

Operaciones:

- Reservar: utiliza una hora generada aleatoriamente, pide acceso a la tabla y cuando es obtenido verifica si la hora está disponible, en el caso de estarlo la reserva usando su número de alumno.

- Cancelar: busca en la tabla si el alumno tiene reservada alguna hora, en el caso de tenerla pide acceso a la tabla y cuando es obtenido borra la reserva más temprana del día. Si no tenía reservada ninguna hora esto es mostrado por pantalla
- Consultar: genera una hora aleatoria, accede a la tabla sin necesidad de pedir acceso y muestra por pantalla si dicha hora está libre o fue reservada.

Constantes:

- NUM_HORAS: utilizado para modular cuantas horas del día a partir de la primera hora pueden ser reservadas
- PRIMERA_HORA: utilizado para modular a qué hora del día arrancan las reservas
- LIBRE: representa que una hora no está reservada.
- NUM_HILOS: utilizado para modular la cantidad de hilos que son creados, cada hilo representando a un alumno.
- TIEMPO_ENTRE_ACCIONES: utilizado para definir un intervalo de tiempo en el que cada alumno realiza cada una de sus 4 acciones.

Procesos y memoria compartida: definimos una estructura de memoria compartida en la que guardamos la tabla de reservas y los 3 semáforos ya explicados. El modelo de la implementación es el mismo, cambiando únicamente la utilización de procesos en lugar de hilos, y con ellos el manejo de la memoria compartida.

Compilación: dirigirse a la carpeta en la Shell, y ejecutar el comando “make reserva_aulas_hilos” o “make reserva_aulas_proc” respectivamente.

## 2.2. Problemas Conceptuales

1. Considere un sistema de gestión de memoria basado en paginación. El tamaño total de la memoria física es de 2 GB, distribuido en páginas de tamaño 8 KB. El espacio de direcciones lógicas de cada proceso se ha limitado a 256 MB.

Tamaño de página = Tamaño frame: 8KB, offset: log2(8KB) = 13

Cantidad de páginas: 256MB / 8KB = 32.768 = 2^15, 15 bits para direccionar la página

a) Determine el número total de bits en la dirección física.

Log2(2GB) = 31 bits, 18 bits para direccionar al marco y 13 bits para direccionar dentro del marco.

b) Determine el número de bits que especifican la sustitución de página y el número de bits para el número de marco de página.

Bits que especifican la sustitución en cada entrada:

- Valido: 1 representa que la entrada es válida, 0 representa lo contrario
- Referencia: cuándo una página es referenciada se le asocia este bit en 1, sino inicialmente en 0. Se busca reemplazar aquellas entradas con este bit en 0.
- Modificado: dirty bit que representa que la entrada fue modificada y debe ser escrita en disco. Si este bit está en 0, significa que no es necesario volver a copiar el valor al disco ya que no hubo modificaciones

Cantidad de frames: 2GB / 8KB = 262.144 = 2^18, 18 bits para direccionar al marco de página.

c) Determine el número de marcos de página.

Número de marcos de página: 2GB / 8KB = 262.144

d) Determine el formato de la dirección lógica.

Log2(256MB)= 28 bits de dirección lógica.

Cantidad de páginas = 2^15, por lo que se necesitan 15 bits para direccionar a la página.

Tamaño de página= 8KB = 2^13, por lo que se necesitan 13 bits para direccionar dentro de la página

Dirección lógica: |15b (direccionar página)|13b (offset dentro de la página)|

2. Para cada una de las siguientes direcciones lógicas, determina la dirección física o indica si se produce un fallo de segmento:

| Dirección Inicial | Largo (bytes) |
|---|---|
| 830 | 346 |
| 648 | 110 |
| 1508 | 408 |
| 770 | 812 |

a) 0, 228: segmento 0, 228 < 346, dirección física: 1058

b) 2, 648: segmento 2, 648 > 408, hay Segmentation Fault

c) 3, 776: segmento 3, 776 < 812, dirección física: 1546

d) 1, 98: segmento 1, 98 < 110, dirección física: 746

e) 1, 240: segmento 1, 240 > 110, hay Segmentation Fault