PROYECTO SISTEMAS OPERATIVOS

INFORME

Segundo Cuatrimestre de 2023

Isaí Ezequiel García Caviglione - 131376
Julian Alconcher - 126094

Isaí Ezequiel García Caviglione
Julian Alconcher

ÍNDICE

Isaí Ezequiel García Caviglione
Julian Alconcher

Entorno de Trabajo

Al realizar los experimentos se trabajó con el sistema operativo de Linux, en una máquina virtual utilizando Raspberry PI Desktop.
La máquina virtual utilizada fue VMware Workstation Player. Para compilar en Linux se utilizó el compilador gcc.
La librería utilizada para manejar los threads fue pthread.h y para el manejo de semáforos se utilizó semaphore.h.

Ejecución de las Experiencias

Para la ejecución de las experiencias realizadas en Linux se desarrolló un script llamado “make.sh”. El script brinda una interfaz para poder compilar y ejecutar todas las experiencias del proyecto. Al momento de querer ejecutar alguna, elegimos el número que le corresponde y apretamos enter. Cuando cortamos la ejecución de alguna experiencia, para volver al menú del “make.sh”, solamente volvemos a ejecutarlo.
Este script se puede ejecutar por consola, previo a ejecutar se deben otorgar permisos de ejecución con el comando “chmod +x make.sh”.
Luego se puede ejecutar escribiendo “./make.sh”. Un ejemplo se muestra a continuación en las siguientes imágenes.

make.sh

```
~ $ cd proyectoSO/
~/proyectoSO $ chmod +x make.sh
~/proyectoSO $ ./make.sh
```

Una vez ejecutado, se verá el siguiente menú:

```
Ingrese el numero de acuerdo al inciso que desea ejecutar:

(1) 1.1.1 - Banco con hilos (requiere ser detenido con CTRL C)
(2) 1.1.1 - Banco con procesos (requiere ser detenido con CTRL C)
(3) 1.1.2 - MiniShell
(4) 1.2.1 - Secuencia 1 (ABABC) con hilos (requiere ser detenido con CTRL C)
(5) 1.2.1 - Secuencia 1 (ABABC) con procesos (requiere ser detenido con CTRL C)
(6) 1.2.1 - Secuencia 2 (ABABCABCD) con hilos (requiere ser detenido con CTRL C)
(7) 1.2.1 - Secuencia 2 (ABABCABCD) con procesos (requiere ser detenido con CTRL C)
(8) 1.2.2 - Aulas implementado con hilos (requiere ser detenido con CTRL C)
(9) 1.2.2 - Aulas implementado con procesos (requiere ser detenido con CTRL C)
```

Para ejecutar cualquiera de las experiencias, se debe colocar el número y apretar enter. Recuerde que se debe apretar “ctrl c” para cortar la ejecución de la mayoría de experiencias.

Isaí Ezequiel García Caviglione
Julian Alconcher

Implementación de las experiencias

1.1. Procesos, threads y Comunicación

1.Banco

Se implementó el problema del banco. La siguiente experiencia simula el funcionamiento de un banco. En este banco todos los clientes que llegan esperan en la misma cola hasta poder acceder a la mesa de entrada (hasta 30 lugares). Existen 3 tipos de clientes: Empresas, usuarios comunes y políticos. Estos clientes son atendidos por 3 empleados de los cuales dos atienden empresas y uno clientes comunes, al menos que haya políticos en cola de espera que deben ser atendidos con prioridad por cualquiera de los 3 empleados. Es decir, que los políticos tienen prioridad sobre los demás tipos de clientes.
El programa utiliza múltiples hilos para representar diferentes tipos de clientes (políticos, empresas y usuarios) y empleados que los atienden. También utiliza semáforos para controlar el acceso a las áreas de atención y la capacidad de espera de cada tipo de cliente.
El código incluye varias bibliotecas estándar de C, como stdio.h, stdlib.h, unistd.h, pthread.h, semaphore.h y sys/wait.h, que se utilizan para funciones de E/S, manejo de hilos, semáforos y otros recursos.
Para modelar el problema se optó por la creación de un número fijo de clientes que intentan entrar al banco y ser atendido. Cada hilo representa un cliente. La cantidad de clientes a crear esta dada por un #define llamado CANT_CLIENTES. Una vez que se atendió a un cliente, el mismo hace un sleep(), simulando que tiene que dirigirse de nuevo a la mesa de entrada y volver a intentar que lo atiendan.
En la función principal, se inicializan los semáforos y se crean hilos para los empleados y clientes. Los empleados son de dos tipos (empresa y usuario) y se crean tres. Los clientes se generan de forma aleatoria y pueden ser políticos, empresas o usuarios. Luego, se espera a que todos los hilos de empleados y clientes finalicen y se liberen los recursos de los semáforos.

```
pthread_t clientes[CANT_CLIENTES];
for(int i=0; i<CANT_CLIENTES; i++){
    int tipoCliente = rand() % 3;
    switch(tipoCliente) {
        case 0:
                pthread_create(&clientes[i], NULL, threadPolitico, NULL);
                break;
        case 1:
                pthread_create(&clientes[i], NULL, threadEmpresa, NULL);
                break;
        case 2:
                pthread_create(&clientes[i], NULL, threadUsuario, NULL);
                break;
```

Los tres empleados también fueron modelados con tres hilos. Las funciones empleadoEmpresa y empleadoUsuario representan a los empleados que atienden a las empresas y a los usuarios, pero siempre priorizando la atención de los políticos. Los semáforos son fundamentales en este código para coordinar el acceso de los clientes a las diferentes colas y para garantizar la atención prioritaria a los políticos. Estas funciones utilizan semáforos como semPoliticoLleno, semEmpresaLleno y semUsuarioLleno para verificar si hay clientes esperando ser atendidos. Si hay clientes, los atienden y liberan espacio en la mesa correspondiente, esperando un tiempo simulado antes de volver a estar disponibles.
Para modelar la sincronización y simular las “filas” de espera de los clientes se usó varios semáforos.

La función mesaDeEntrada verifica si hay espacio disponible en la mesa principal de entrada. Si hay espacio, devuelve 1, lo que indica que el cliente puede pasar a la mesa de entrada principal. Si no hay espacio, devuelve -1, lo que significa que el cliente debe retirarse.

El banco tiene tres empleados: dos para empresas y uno para usuarios comunes. Cada empleado es representado por una función: empleadoEmpresa y empleadoUsuario. Estos empleados tienen un ciclo infinito donde intentan atender a los clientes en función de las reglas del banco. Los empleados priorizan la atención de políticos si están disponibles en la cola de políticos, de lo contrario, atienden a empresas y usuarios, respectivamente.

Análisis de la salida del programa:

Primero analicemos que el programa modela la entrada de los clientes a la mesa principal y a sus mesas específicas. Luego empieza a atender políticos exclusivamente. Esto ocurre ya que tienen prioridad. Una vez que en la fila de políticos, no quedan más, los empleados empiezan a atender otro tipo de cliente.

```
Un político entró a la mesa principal.
Un político no pudo entrar a la mesa de políticos.
Un usuario entró a la mesa principal.
Un usuario no pudo entrar a la mesa de usuarios.
Una empresa entró a la mesa principal.
Una empresa no pudo entrar a la mesa de empresas.
Un político entró a la mesa principal.
Un político no pudo entrar a la mesa de políticos.
Una empresa entró a la mesa principal.
Una empresa no pudo entrar a la mesa de empresas.
Un usuario entró a la mesa principal.
Un usuario no pudo entrar a la mesa de usuarios.
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
Una empresa entró a la mesa principal.
Una empresa no pudo entrar a la mesa de empresas.
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
```

Por ejemplo, en esta instancia, una vez que se atendieron los políticos, se empiezan a atender otro tipos de clientes. Igualmente, observamos que podría existir intuición para la atención de los otros tipos de clientes, ya que como los políticos tienen prioridad, siempre se van a tener a ellos primeros y dejar de lado a los demás. Esto lo solucionamos colocando un sleep() mayor en las rutinas ejecutadas por hilos políticos, de esa forma, simulamos como que tardan más en volver a querer ser atendidos.

```
POLÍTICO ATENDIDO
POLÍTICO ATENDIDO
EMPRESA ATENDIDA
EMPRESA ATENDIDA
USUARIO ATENDIDO
```

La experiencia se ejecuta infinitamente hasta que sea cortada mediante un “ctrl+C”.

Isaí Ezequiel García Caviglione
Julian Alconcher

Implementación:: PROCESOS Y COLAS DE MENSAJES

Luego, para la implementación de la experiencia del banco pero usando procesos la idea es similar pero con algunos cambios.

Se pide que en vez de utilizar hilos para su implementación y sincronizarlo con semáforos, en este caso tendremos que utilizar procesos y para su comunicación usar pipes o colas de mensajes. Nuestra elección fue colas de mensajes.

La estrategia que utilizamos es tener, al igual que la primera implementación, a los clientes como procesos separados y además contar con tres procesos empleados. Estos van a ser los encargados de revisar las colas de los clientes (cola de político, cola de empleado común y cola de empresa) y empezar a atender siempre priorizando a los clientes políticos.

Para su comunicación y sincronización usamos colas de mensajes. Luego, tomamos la decisión de hacer una simulación de semáforos con las mismas. Usamos la cola como semáforos contadores. Luego con las dos operaciones principales de las colas “msgsnd” y “msgrcv” pudimos simular post() y wait() respectivamente. Además, si activamos un flag denominado “IPC_NOWAIT” en las operaciones de “msgrcv” podemos simular un “trywait” no bloqueante.

```
if (msgrcv(msg_id1, &message, sizeof(message), 1, IPC_NOWAIT) != -1)
    printf("--- ATENDIENDO POLITICO ---.\n");
```

Como se ve en la imagen, esta sería la simulación de una operación de trywait de un semáforo sobre la cola de la mesa principal.

En esta imagen podemos observar la “inicialización de los semáforos contadores” utilizando a la cola de mensajes como varios semáforos dependiendo el tipo de mensaje enviado y recibido. Por ejemplo, el “semaforo” SEM_MESA_ENTRADA_VACIO se inicializa en 30.

```
//Inicializo mesa en 30, simulando semaforo contador.
for(int i=0; i<BUFFER_SIZE; i++) {
    struct Mensaje mensajeEnviado;
    mensajeEnviado.tipo = SEM_MESA_ENTRADA_VACIO;
    if (msgsnd(msg_id, &mensajeEnviado, sizeof(int), IPC_NOWAIT) == -1) {
        perror("msgsnd:: ERROR AL INICIALIZAR MESA DE ENTRADA CON 30");
    }
}
//Inicializo cada cola en 15, simulando semaforo contador.
for(int i=0; i<(MINI_BUFFER_SIZE); i++) {
    struct Mensaje mensajeUsuario;
    mensajeUsuario.tipo = SEM_COLA_USUARIO_VACIO;
    if (msgsnd(msg_id, &mensajeUsuario, sizeof(int), IPC_NOWAIT) == -1) {
        perror("msgsnd:: ERROR AL INICIALIZAR MESA DE USUARIO CON 15");
    }
```

Luego, a la hora de realizar las operaciones cada empleado o cliente, abre sus colas de mensajes necesarias para empezar a operar. Esto se hace mediante el método de abrirColaMensajes(char ch).

Para concluir, podemos simular este comportamiento del banco de ambas formas. Sin embargo, la alternativa de usar múltiples hilos y sincronizarlos con el uso de semáforos es la opción más obvia e inteligente para realizar. Además es más eficiente con respecto a los recursos. Los hilos comparten el mismo espacio de direcciones y recursos del proceso padre, lo que los hace más eficientes en términos de uso de memoria y tiempo de ejecución en comparación con los procesos que tienen su propio espacio de direcciones. También la comunicación entre los hilos es mucho más rápida y trivial.

Además, implementar esta simulación usando procesos es más complejo y resulta en una experiencia más complicada a la hora de programarlo. Capaz, la experiencia mejoraría usando alguna otra técnica de comunicación entre procesos o utilizar para sincronizar semáforos en memoria compartida.

2. MiniShell

Los comandos elegidos para la creación de la MiniShell son:

1. **HELP**: Este comando proporciona información de ayuda o una lista de comandos disponibles.
2. **MKDIR**: Se utiliza para crear directorios (carpetas) en el sistema de archivos. Por ejemplo, mkdir nombre_directorio crearía una carpeta llamada "nombre_directorio".
3. **RMDIR**: Este comando se utiliza para eliminar directorios vacíos en el sistema de archivos. Si un directorio contiene archivos o subdirectorios, rmdir no funcionará.
4. **TOUCH**: Crea un archivo vacío con el nombre especificado. Por ejemplo, touch archivo.txt crearía un archivo de texto vacío llamado "archivo.txt".
5. **LS**: Listar archivos y directorios en el directorio actual.
6. **CAT**: Muestra el contenido de un archivo en la terminal. Por ejemplo, cat archivo.txt mostrará el contenido del archivo "archivo.txt".
7. **CHMOD**: Se utiliza para cambiar los permisos de un archivo. Los posibles permisos son: 0 para leer, 1 para escribir, 2 para ejecutar, 3 para leer y escribir, 4 para leer y ejecutar, 5 para escribir y ejecutar, 6 para todos los permisos.
8. **CLS**: Se utiliza para limpiar la pantalla de la terminal y eliminar el historial de comandos anteriores para que la pantalla esté limpia.
9. **EXIT**: Cierra la terminal o la sesión actual. Puedes utilizar este comando para salir de la terminal o cerrar una sesión de usuario.

Decisiones de **diseño**

Para facilitar la comparación del comando ingresado por el usuario con los comandos disponibles se utiliza una función hash la cual compara la entrada con los “define” ya establecidos con el valor hash de cada comando.
En un while infinito se compara lo ingresado por el usuario con los posibles comandos y se realiza el comando solicitado o se informa que lo que el usuario ingresó no era un comando válido.
Por ejemplo, cuando se inserta el comando de “help”, nos indica los posibles comandos a realizar y si se lo intenta llamar con algún argumento, se devuelve el error que puede visualizarse en la siguiente captura.

```
CustomShell> help
Comandos disponibles:
a) Mostrar ayuda: help
b) Crear directorio: mkdir <nombre_directorio>
c) Eliminar directorio: rmdir <nombre_directorio>
d) Crear archivo: touch <nombre_archivo>
e) Listar contenido de directorio: ls <nombre_directorio>
f) Mostrar contenido de archivo: cat <nombre_archivo>
g) Modificar permisos de archivo: chmod <nombre_archivo> <permisos>
h) Limpiar el shell: cls
i) Salir del shell: exit
CustomShell> help a
Uso incorrecto. El comando 'help' no requiere argumentos.
CustomShell>
```

Ejemplo de creación exitosa de un nuevo directorio utilizando la MiniShell. Una vez creado, nos arroja el siguiente mensaje de éxito, dando feedback de la situación.

```
CustomShell> mkdir nuevoDirect
Directorio creado con éxito.
```

1.2 Sincronización

1. Secuencia

Esta experiencia consiste en que dadas estas dos secuencias de letras: ABABC y ABABCABCD debemos implementar un programa multihilo para la impresión de manera cíclica de las mismas usando por un lado hilos y semáforos y por el otro lado implementarlo con procesos y pipes.

Secuencia 1 **ABABC con hilos**
El programa crea tres hilos, uno para imprimir "A", otro para imprimir "B" y otro para imprimir "C". Cada hilo utiliza semáforos para controlar el orden en el que se imprimen las letras y garantizar que se imprima "ABABC" en ese orden. Esto se ejecuta infinitamente y para poder cortar su ejecución necesita ser detenida manualmente mediante “ctrl+c”.
Se incluyen las bibliotecas necesarias: stdio.h para la entrada y salida estándar, pthread.h para la gestión de hilos, semaphore.h para semáforos y unistd.h para ciertas funciones relacionadas con el sistema.
Se define la constante CANT_HILOS con un valor de 3, que representa la cantidad de hilos que se crearán en el programa.
Se declaran tres semáforos: sem_a, sem_b y sem_c inicializados en 1, 0 y 0 respectivamente. Estos semáforos se utilizarán para controlar el orden de impresión de las letras.
En conclusión, los tres hilos trabajan juntos para lograr este orden y evitar que se imprima otra letra fuera de secuencia. Se colocó un espacio después de la última letra (“C”) para poder diferenciar mejor las distintas instancias de la secuencia. La salida por consola del siguiente programa se ve algo así:

```
ABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABAB
C ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC
ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC AB
ABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABAB
C ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC
ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC AB
```

Secuencia 2 **ABABCABCD con hilos**
El programa crea cuatro hilos, uno para imprimir "A", otro para imprimir "B", otro para imprimir "C" y otro para imprimir la letra “D”. Cada hilo utiliza semáforos para controlar el orden en el que se imprimen las letras y garantizar que se imprima "ABABCABCD" en ese orden. Esto se ejecuta infinitamente y para poder cortar su ejecución necesita ser detenida manualmente mediante “ctrl c”.
Se incluyen las bibliotecas necesarias: stdio.h para la entrada y salida estándar, pthread.h para la gestión de hilos, semaphore.h para semáforos y unistd.h para ciertas funciones relacionadas con el sistema.
Se define la constante CANT_HILOS con un valor de 4, que representa la cantidad de hilos que se crearán en el programa.
Se declaran cinco semáforos: sem_a, sem_b, sem_c, sem_c2, sem_d, inicializados en 1, 0, 0, 0, 0 respectivamente. Estos semáforos se utilizarán para controlar el orden de impresión de las letras.
En conclusión, los cuatro hilos trabajan juntos para lograr este orden y evitar que se imprima otra letra fuera de secuencia. Se colocó un espacio después de la última letra (“D”) para poder diferenciar mejor las distintas instancias de la secuencia. La salida por consola del siguiente programa se ve algo así:

```
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
```

Implementación usando Procesos y Pipes.

La implementación de las secuencias usando procesos es muy similar. En vez de crear tres hilos para la primera secuencia y cuatro hilos para la segunda, se crean tres procesos y cuatro procesos respectivamente.
La sincronización de los hilos en vez de realizarse mediante semáforos, la misma se realizará con el uso de pipes.

Secuencia 1 **ABABC con procesos**

Se definen las siguientes constantes y estructuras:

- SIZE_MSG: El tamaño de la estructura tMensaje, que es de 12 bytes.
- CANT_PROCESOS: Un valor constante que representa la cantidad de procesos a crear (3).
- CANT_PIPES: Un valor constante que representa la cantidad de tuberías (pipes) a crear (3).
- Struct message es una estructura que contiene un campo cuerpo para almacenar un mensaje de 12 caracteres.

Se declaran tres pipes llamadas pipeA, pipeB y pipeC, y se inicializan los descriptores de archivo para cada una.
Se definen tres funciones print_a, print_b y print_c que se ejecutarán en procesos separados. Cada una de estas funciones realiza lo siguiente: cierra los descriptores de archivo no necesarios y en un bucle infinito, lee un mensaje de una tubería, imprime una letra (A, B o C) y envía el mensaje a la siguiente tubería. Por último, el proceso padre espera a que todos los procesos hijos terminen usando la función wait. La salida por consola es exactamente igual a la implementación usando hilos.

```
ABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABAB
C ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC
ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC AB
ABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABAB
C ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC
ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC ABABC AB
```

Secuencia 2 **ABABCABCD con procesos**

Se definen las siguientes constantes y estructuras:

- SIZE_MSG: El tamaño de la estructura tMensaje, que es de 12 bytes.
- CANT_PROCESOS: Un valor constante que representa la cantidad de procesos a crear (cuatro).
- CANT_PIPES: Un valor constante que representa la cantidad de tuberías (pipes) a crear (cuatro).
- Struct message es una estructura que contiene un campo cuerpo para almacenar un mensaje de 12 caracteres.

Se crea un arreglo de tuberías llamado arregloPipe que contiene las tuberías pipeA, pipeB, pipeC, y pipeD. Luego, se utiliza un bucle para crear estas tuberías con la función pipe. Si hay algún error en la creación de las tuberías, se imprime un mensaje de error y el programa termina.
Se definen cuatro funciones print_a, print_b,print_c y print_d que se ejecutarán en procesos separados. Cada una de estas funciones realiza lo siguiente: cierra los descriptores de archivo no necesarios y en un bucle infinito, lee un mensaje de una tubería, imprime una letra (A, B o C) y envía el mensaje a la siguiente tubería. Por último, el proceso padre espera a que todos los procesos hijos terminen usando la función wait. La salida por consola es exactamente igual a la implementación usando hilos.

```
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD ABABCABCD
```

2. Reserva de Aulas

El siguiente problema consiste en: contamos con un aula que tiene una computadora que se puede reservar en períodos de 1 hora, desde las 9:00hs hasta las 21:00hs. Existen 25 alumnos que pueden reservar períodos individuales de 1 hora, cancelar, reservar y consultar el estado de las reservas. Cada alumno realiza cuatro operaciones de forma aleatoria. La probabilidad para la realización de reservas (50%) es mayor que para la realización de consultas y cancelaciones (25% cada una).
La implementación cuenta con un hilo principal que lanza 25 hilos en paralelo, uno por cada alumno.

Decisiones de implementación:

Contamos con un buffer principal que representa las 12 horas (CANT_HORAS) posibles para reservar. Es decir, desde las 9hs hasta las 21hs. Este buffer es un arreglo de struct llamados “struct hora”. Cada “struct hora” posee un booleano que determina si esa hora está reservada o no (int reservado) y además cuenta con un entero que determina quien reservó esa hora denominado (int numAlumno).

```
struct hora{
    int reservado;
    int numAlumno;
};


struct hora buffer[CANT_HORAS];
```

Al principio de la ejecución de la experiencia, llamamos a la función “inicializarBuffer” que será la encargada de inicializar el buffer con todas las horas libres por defecto.

```
void inicializarBuffer(){
    for(int i=0; i<CANT_HORAS; i++){
        buffer[i].numAlumno = -1;     // Nadie reservo
        buffer[i].reservado = LIBRE;  // LIBRE por defecto
    }
}
```

Para implementar la aleatoriedad con distintas probabilidades de las operaciones se optó por realizar la siguiente función. La misma modela una probabilidad del 50% para reservar, 25% para cancelar y 25% para consultar.

```
void operacionAleatoria(int idA){
    int operacion = rand() % 4;
    switch(operacion) {
        case 0: reservar(idA); break;
        case 1: reservar(idA); break;
        case 2: cancelar(idA); break;
        case 3: consultar(idA);break;
    }
}
```

Observaciones del método de Cancelar y Consultar.
Tomamos como decisión que a la hora de que un alumno quiera cancelar una reserva, la misma tiene que existir, es decir, que ese alumno tenga alguna hora reservada dentro del rango permitido. Entonces, al momento de cancelar una hora, optamos por cancelar la primera reserva (si es que tuviera más de una) colocando esa componente del buffer en LIBRE.
Para la operación de consulta, se hace de la misma forma que reservar, es decir, de manera completamente aleatoria.

Este método es el encargado de devolver la primera hora de reserva (si tuviera más de una) hecha por el alumno con id “idA”. Si el alumno no cuenta con reserva, el método retorna el valor por defecto asignado que es “-1”.

```
int getPrimeraHoraDeMisRevervados(int idA){
    int aRetornar = -1;
    for(int i=0; i<CANT_HORAS; i++){
        if(buffer[i].numAlumno == idA){
            aRetornar = i;
        }
    }
    return aRetornar;
}
```

Por último, para lograr una correcta sincronización de las operaciones, se usaron solo dos semáforos (semResCan, semCantCons, inicializados en 1 y 0) . Los mismos permiten realizar las operaciones de reserva y cancelación de manera exclusiva sin tener problemas en su sección crítica.

La operación de consulta contempla la posibilidad de realizar consultas en simultáneo por mas de un alumno. El semáforo semResCan es un semáforo binario que modela si se puede realizar una cancelación y/o reserva y el semáforo semCantCons se refiere a la cantidad de consultas que se estén realizando. Lo que hace la función consultar es lo siguiente: si es el primer hilo en consultar, entonces “bloquea” el semáforo semResCan para que no se puedan realizar reservas ni cancelaciones, luego se hace un post(semCantCons) para señalizar que hay una consulta más, posteriormente se realiza la consulta y se hace el wait para aclarar que se terminó. Luego, si fue la última consulta se vuelve a habilitar el semáforo para que se puedan hacer reservas y/o cancelaciones.

```
void consultar(int idA){
    if(sem_trywait(&semCantCons)!=0) {
        sem_wait(&semResCan);
    } else {
        sem_post(&semCantCons);
    }
    sem_post(&semCantCons);
    printf("Alumno %d quiere consultar.\n", idA);
    int h = horaRandom();
    if(buffer[h].reservado == RESERVADO){
        printf("La hora %dhs consultada por el alumno %d se encuentra reservada.\n", obtenerHora(h), idA);
    } else {
        printf("La hora %dhs consultada por el alumno %d no se encuentra reservada.\n", obtenerHora(h), idA);
    }
    sem_wait(&semCantCons);
    if(sem_trywait(&semCantCons) != 0) {
        sem_post(&semResCan);
    }else{
        sem_post(&semCantCons);
    }
}
```

Una salida de este programa se ve algo así:

```
Alumno 7 quiere consultar.
La hora 20hs consultada por el alumno 7 no se encuentra reservada.
EL alumno 7 quiere reservar.
Se ha reservado con exito a las 19 hs, por el alumno 7
Alumno 7 quiere consultar.
La hora 18hs consultada por el alumno 7 no se encuentra reservada.
EL alumno 16 quiere reservar.
No es posible reservar a las 10 horas por el alumno 16, no se encuentra disponible.
Alumno 16 quiere consultar.
La hora 12hs consultada por el alumno 16 se encuentra reservada.
EL alumno 16 quiere reservar.
Se ha reservado con exito a las 11 hs, por el alumno 16
Alumno 16 quiere consultar.
```

2. Problemas

2.1 Lectura

INTEGRITY OS

Integrity es un sistema operativo de tiempo real. Está certificado por POSIX y destinado para usarse en sistemas embebidos para arquitecturas de 32 y 64 bits. Este sistema operativo permite a los programadores asegurar que sus aplicaciones cumplan con los mayores requisitos posibles en cuanto a seguridad, confiabilidad y rendimiento.

Integrity usa particiones de seguridad, que aseguran que cada tarea obtenga los recursos que necesitan y protege al sistema operativo y tareas de usuarios de código malicioso. Si bien integrity busca seguridad y confiabilidad, este sistema nunca sacrifica rendimiento de tiempo real por sobre estas dos.

Arquitectura confiable

El kernel de integrity protege al sistema de código malicioso mediante control sobre la escritura por fuera de las regiones asignadas de memoria. Además, el sistema previene el acceso involuntario a datos por fuera de la partición donde los datos se encuentran.

El sistema puede denegar eventos maliciosos o no intencionados a los recursos del sistema y así impedir que los procesos del sistema se ejecuten según lo previsto. Para evitar estos ataques, el sistema puede asignar tiempo fijos de CPU y espacio fijo de memoria a cada proceso.

Rendimiento y Memoria

Integrity es un sistema operativo de tiempo real duro. Este sistema puede responder a eventos en nanosegundos de forma asegurada. Todos los servicios del kernel fueran optimizados de forma cuidadosa para minimizar el overhead de las llamadas del sistema y de esa forma pueden suspenderse para permitirle a otras llamadas poder ejecutarse. Integrity usa un planificador de tiempo real que soporta múltiples niveles de prioridades y permite absoluto control sobre el porcentaje de alocación del CPU. INTEGRITY evita que un espacio de direcciones agote la memoria de cualquier otro.

Para prevenir el riesgo de stack overflow, el kernel posee su propia pila de memoria. Sin esta propiedad, el kernel necesita acceder a la pila de usuario del proceso, pero esto podría ocasionar problemas por que es imposible anticipar el espacio máximo de stack requerido.

Soporte multinúcleo avanzado

La arquitectura moderna de INTEGRITY es muy adecuada para procesadores multinúcleo destinados a sistemas integrados. Proporciona soporte completo de multiprocesamiento asimétrico (AMP) y multiprocesamiento simétrico (SMP) que está optimizado para uso integrado y en tiempo real.

Arquitectura, procesador y soporte de placa

El paquete de soporte arquitectónico de integrity provee inicialización de CPU, manejo de excepciones y veloces cambios de contexto para toda arquitectura líder de CPU.

El Kernel de Integrity

El kernel de Integrity es altamente confiable y seguro, diseñado para operar en entornos críticos donde la estabilidad y la seguridad son de suma importancia. Integrity implementa un planificador de tiempo real que soporta múltiples niveles de prioridades y permite absoluto control sobre el porcentaje de alocación del CPU.

Algunas de las características clave del kernel de Integrity incluyen:

- Tiempo Real: El kernel de Integrity proporciona capacidades de tiempo real para garantizar respuestas rápidas y predecibles a eventos críticos.
- Separación y Particionamiento: Integrity permite la creación de particiones aisladas, lo que significa que las aplicaciones críticas se pueden ejecutar en entornos separados y seguros, evitando interferencias no deseadas.
- Seguridad: El kernel de Integrity incluye medidas de seguridad avanzadas para proteger contra amenazas y garantizar la integridad de los datos y el funcionamiento del sistema.
- Tolerancia a fallos: Integrity también ofrece mecanismos de tolerancia a fallos para garantizar la disponibilidad continua del sistema en caso de problemas.

¿Donde es usado INTEGRITY? Este kernel es ampliamente utilizado en aplicaciones críticas como la automoción, la aeroespacial, la electrónica médica y otras industrias donde la seguridad y la confiabilidad son esenciales. Sistemas de seguridad o sistemas de seguridad críticos incluyendo automóviles, naves espaciales, Industria, IoT, Networking, Seguridad, Storage, entre otras.

Propuesta Multimedia

Como propuesta multimedia para promover/vender/presentar la idea hicimos un flyer que muestra las partes más esenciales del sistema.
Optamos por crear un flyer en lugar de una presentación en PowerPoint o alguna presentación más clásica debido a que el flyer nos permite condensar información clave de manera concisa y directa, ideal para comunicar nuestros mensajes de manera efectiva en un espacio limitado.
En el flyer se indican los puntos claves del sistema operativo INTEGRITY. Se listan de forma que pueda ser un flyer para la venta o difusión del mismo.


INTEGRITY
SISTEMA OPERATIVO
DE TIEMPO REAL DURO
Cumplir con los requisitos más altos de
SEGURIDAD, CONFIABILIDAD Y RENDIMIENTO
INTEGRITY establece el estándar para la seguridad de RTOS
Los sistemas operativos tradicionales pueden fallar, bloquearse o ejecutarse sin control, lo que tiene consecuencias costosas.
INTEGRITY protege tanto las aplicaciones críticas como a sí mismo de las fallas que pueden provocar consecuencias al brindar recursos del sistema que garantizan que el tiempo de CPU y los recursos de memoria siempre estarán disponibles para los procesos individuales.
Rendimiento y Memoria
PLANIFICADOR CON MULTIPLES NIVELES DE PRIORIDADES
Todos los servicios del kernel fueran optimizados de forma cuidadosa para minimizar el overhead de las llamadas del sistema y de esa forma pueden suspenderse para permitirle a otras llamadas poder ejecutarse.
Integrity usa un planificador de tiempo real que soporta múltiples niveles de prioridades y permite absoluto control sobre el porcentaje de alocación del CPU.
Soporte Multinúcleo
AVANZADO
INTEGRITY es muy adecuadao para procesadores multinúcleo destinados a sistemas integrados, tales como Arm, Intel, Power y MIPS.
Garantía de correctitud
Cada tarea tendrá los recursos que necesita para ejecutarse correctamente y protegen completamente el sistema operativo y las tareas del usuario contra códigos maliciosos.
El rendimiento primero.
Integrity nunca sacrificará el rendimiento por sobre la seguridad y protección.
Tiempo de respuesta
Integrity asegura tiempos de respuesta a eventos en nanosegundos.
INTEGRITY™
GLOBAL SECURITY

2.2 Problemas Conceptuales

1. Problema Conceptual 1

Datos:

- Memoria basada en paginación.
- 2 GB de memoria física.
- Tamaño de página de 8 KB.
- 256 MB de espacio de direcciones lógicas.

a) $2\ GB = 2^{31}\ bytes$. *Luego, las direcciones físicas tienen* $log_2(2^{31}) = 31\ bits$.

b)

- $8\ KB = 2^{13}\ bytes$. *Por lo tanto, se tienen* $log_2(2^{13}) = 13\ bits\ de\ desplazamiento$.
- *La cantidad de bits para el número de frame o marco será* $31 - 13 = 18$.

c) $\frac{2\ GB}{8\ KB} = \frac{2^{31}}{2^{13}} = 2^{18} = 262144\ frames$ (*o marcos*).

d)

- $256\ MB = 2^{28}\ bytes$. *Así que las direcciones lógicas son de* $28\ bits$.
- *Dado que el tamaño de página es el mismo, se tienen* $13\ bits\ de\ offset$.
- *Y los restantes* $15\ bits$ *son el número de página*. (*Hay menos páginas virtuales que frames físicos*).

2. Problema Conceptual 2

2. Asumo que las direcciones lógicas son de 16 bits.

a)

Dirección lógica: 00 | 00000011100100

Largo del segmento 0 = 101011010 > Offset de la dirección lógica = 011100100

Dirección física = Dirección inicial del segmento 0 + Offset de la dirección lógica:

$(830)_{10}$ = 0 0 0 0 0 0 1 1 0 0 1 1 1 1 1 0

\+ $(228)_{10}$ = 0 0 0 0 0 0 0 0 1 1 1 0 0 1 0 0

$(1058)_{10}$ = 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0

b)

Dirección lógica: 10 | 00001010001000

Largo del segmento 2 = 0110011000 < Offset de la dirección lógica = 1010001000. Se produce un fallo de segmento.

c)

Dirección lógica: 11 | 00001100001000

Largo del segmento 3 = 1100101100 > Offset de la dirección lógica = 1100001000

Dirección física = Dirección inicial del segmento 3 + Offset de la dirección lógica:

$(770)_{10}$ = 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0

\+ $(776)_{10}$ = 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0

$(1546)_{10}$ = 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0

d)

Dirección lógica: 01 | 00000001100010

Largo del segmento 1 = 1101110 > Offset de la dirección lógica = 1100010

Dirección física = Dirección inicial del segmento 1 + Offset de la dirección lógica:

$(648)_{10}$ = 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0

\+ $(98)_{10}$ = 0 0 0 0 0 0 0 0 0 1 1 0 0 0 1 0

$(746)_{10}$ = 0 0 0 0 0 0 1 0 1 1 1 0 1 0 1 0

e)

Dirección lógica: 01 | 00000011110000

Largo del segmento 1 = 01101110 < Offset de la dirección lógica = 11110000. Se produce un fallo de segmento.

Isaí Ezequiel García Caviglione
Julian Alconcher

Correcciones de entrega final

Los errores marcados en la entrega 1 del proyecto fueron los siguientes:

Banco:

- Falta que un cliente se vaya si no hay lugar en la mesa de entrada.
- Usa trywait para pasar de mesa de entrada a cola usuarios (busy waiting)
- Los empleados también hacen trywait para atender, busy waiting.

Shell:

- Un único proceso. Un único archivo
- Usa una función hash() para comparar strings.

Alumnos:

- Usa sem binario en vez de mutex (en hilos)
- No sincroniza entre lectores (consultas) más de 1 lector puede tratar de bloquear la estructura.

Las soluciones a los mismos fueron las siguientes.

Para el **Banco**, se modelo la salida de los clientes en caso de que no haya lugar en la mesa de entrada. Esto se modeló usando “pthread_exit” y de esa forma eliminar al hilo simulando la salida del cliente del banco.
Se eliminó el busy waiting tanto para pasar de mesa de entrada a cola de usuarios como el busy waiting realizado por los empleados al hacer try-wait. Esto último fue solucionado haciendo un wait() para cliente común o empresa según corresponda, quitando momentáneamente la prioridad de los políticos.

Para la **shell**, se modelaron los comandos en distintos procesos. El main de la shell, según corresponda hacer fork() de un nuevo proceso y carga la imagen ejecutable del comando a ejecutar, determinado por la consola. Además se quitó la comparación de string mediante una función hash().

Para los **alumnos**, se cambió el semáforo binario por un mutex. También se agregó un mutex para agregar la exclusión mutua a la hora de consultar por parte de los alumnos. De esta forma puede haber mas de un lector (consultar) en simultáneo y no bloquea la estructura.