Sistemas Operativos - 2023

Informe del Proyecto

Reale Guido - Asteasuain Martina

Introducción

Durante el segundo cuatrimestre de 2023, en el marco de la asignatura de Sistemas Operativos, nos enfrentamos a un proyecto que requería la implementación y resolución de diversos desafíos, con la finalidad de aplicar los conceptos que hemos estudiado a lo largo de estos últimos meses. En el presente informe, detallaremos cada uno de estos desafíos siguiendo el mismo orden en el que se presentaron en el enunciado original, y proporcionaremos las herramientas necesarias para la ejecución de cada uno de los archivos de código que hemos entregado junto con este informe.

1. Procesos, threads y comunicación

1.1. BANCO

En la descripción de este problema se nos pidió que desarrollemos una solución a una situación típica de un banco en el día a día, en el cual una persona llega a un banco, si hay lugar para ingresar lo hace, y si no sigue con otra actividad de su rutina. Por otra parte, debimos modelar también la prioridad de algunos clientes sobre otros, donde por ejemplo un cliente político tiene prioridad sobre otros tipos de clientes, como pueden ser clientes empresarios o clientes comunes. así mismo, y como se desprende de la descripción anterior, los tipos de clientes debían ser tres, donde los clientes comunes podían ser atendidos por un empleado, y los empresarios por otros dos, y los tres empleados debían dedicar su tiempo prioritariamente para atender políticos cuando estos estén esperando para concretar dicha actividad.

Todo esto hubo que hacerlo de dos maneras diferentes, la primera con hilos y semáforos, y la segunda a elección entre procesos con colas de mensajes o con pipes.

a) a continuación la descripción de la primer solución:

- aquí, y como bien mencionamos que el enunciado lo pedía, creamos tres hilos, uno correspondiente a cada empleado, donde uno de ellos

ejecutaba el método “empleadoClienteComun()” y los dos restantes ejecutaban el metodo “empleadoEmpresario()”. el comportamiento de estos métodos es análogo, cambiando el tipo de cliente que atienden si no hay políticos esperando, y funciona de la siguiente manera:

Creamos un loop(“while(1)”) donde el empleado “duerme” (modelado con el semáforo “despertarEmpleadoComun”) mientras no haya clientes para atender y cuando es despertado por algún cliente que ingresa a su cola para ser atendido lo “despierta” utilizando el semáforo mencionado antes. en este punto, el empleado hace un sem_trywait sobre el semáforo “politicoEsperando”, y si este devuelve 0 atiende al político con prioridad sobre el resto de empleados. luego de ejecutar este trywait, si pudo ingresar y atender al político, termina su iteración actual y vuelve a iniciar para seguir atendiendo políticos con prioridad (si es que hay alguno más esperando a ser atendido). si por el contrario el trywait dio como resultado algo distinto de 0 pasamos a chequear otro trywait que funciona igual que el anterior pero en este caso para atender a cliente de la categoría que el empleado esté capacitado para atender. modelado de esta manera nos aseguramos que se priorice la atención a los políticos sobre el resto de clientes, así como garantizamos que los empleados no hagan espera ocupada sobrecargando innecesariamente la cpu.

Luego, cada hilo de cliente ejecuta el método correspondiente a su categoría, funcionando los tres de manera muy similar, solo cambiando en una pequeña porción. El comportamiento de estos es el siguiente:

En primer lugar se realiza un trywait sobre el semáforo “mesaEntrada” para modelar el comportamiento de que si hay lugar entra en el banco (y consecuentemente ejecuta todo el resto del código) y si no lo hay se va. Una vez que ingresa en el banco (obviamos las partes en las que se escribe con printf’s) se realiza un wait a la cola correspondiente a su categoría de cliente, para quedarse bloqueado hasta que haya un

lugar en su cola, y cuando logra tomar ese recurso se devuelve el recurso tomado sobre mesaEntrada. Luego postea que hay un nuevo cliente de su tipo esperando y despierta a los empleados que tienen capacidad para darle atención, y como comportamiento final espera al semáforo que avisa que fue atendido y se retira del banco liberando la cola de su categoría de cliente.

para ejecutar el código correspondiente a esta versión del problema, se creó un script bash llamado *banco.sh*, y para ejecutarlo hay que escribir los siguientes comandos en consola:

~ chmod +x banco.sh

~ ./banco.sh

b) ahora la descripción de la segunda solución implementada, esto es, la resolución utilizando procesos y colas de mensajes:
   - al aplicar esta solución, lo primero que hacemos es definir la estructura msg_buffer que se utiliza para representar los mensajes que se enviarán y recibirán a través de las colas de mensajes.

     Luego, declaramos varias variables globales, como msg_entrada, msg_COLA_POLITICOS, msg_POLITICO_ATENDIDO, etc., que se utilizarán para enviar y recibir mensajes en las colas de mensajes.

     Más adelante definimos las funciones que escriben por pantalla simulando la atención de los empleados a los clientes, estas son atenderPolitico, atenderComun y atenderEmpresa.

     Siguiendo, se definen tres funciones para modelar el comportamiento de los clientes que van llegando al banco:

     - clientePolitico

- clienteEmpresa y
- clienteComun.

Cada una de estas funciones simula la llegada de un cliente de una categoría específica al banco. Si hay espacio en la mesa de entrada, el cliente ingresa al banco. Luego, se coloca en una cola específica (por ejemplo, msg_COLA_POLITICOS para políticos) y espera a ser atendido luego de liberar la cola para la mesa de entrada y despertar a él/los empleados capaz/capaces de atenderlo. Una vez que es atendido, se libera la cola particular de su categoría y el cliente se va del banco. Si no hay espacio en la mesa de entrada, el cliente se retira.

Así mismo, se definen dos funciones: empleadoEmpresarios y empleadoComunes. Estas funciones representan a los empleados del banco que atienden a los clientes. Están modelados con un ciclo while(1) donde lo primero que realizan es una espera a ser despertados por algún cliente que esté listo para ser atendido, luego estos empleados atienden primero a los políticos (si es que hay clientes de esta categoría esperando a ser atendidos), y cuando no haya más políticos esperando para ser atendidos continúan por atender a los clientes de su categoría particular, esto es, clientes comunes o clientes empresarios dependiendo del empleado que se trate.

En el main se configuran las colas de mensajes y se crean los clientes y empleados, también se crean varias colas de mensajes y se inicializan con mensajes. Luego, se crean clientes y empleados, cada uno en procesos separados utilizando la función fork(). Los clientes y empleados se crean en función de las constantes CANTIDAD_POLITICOS, CANTIDAD_EMPRESAS, CANTIDAD_COMUNES, MAXIMO_MESA_ENTRADA y MAXIMO_COLAS_PARTICULARES.

para ejecutar el código correspondiente a esta versión del problema, se creó un script bash llamado *bancoColaMensajes.sh*, y para ejecutarlo hay que escribir los siguientes comandos en consola:

~ chmod +x bancoColaMensajes.sh

~ ./bancoColaMensajes.sh

c) Finalmente, comparando las dos soluciones como pedía el enunciado pudimos llegar a la conclusión de que es una opción más viable resolver el problema utilizando hilos y semáforos por algunas razones que detallamos a continuación:
    - en primer lugar tenemos que al utilizar semáforos, las operaciones de wait de los semáforos realizan una espera bloqueante, mientras que los recieve de las colas de mensajes constantemente están verificando si se recibió el mensaje, lo que implica que se aplica una espera ocupada que no es recomendable según lo estudiado en teoría.
    - En segundo lugar, tenemos el beneficio del rendimiento, ya que las implementaciones basadas en hilos suelen ser más eficientes en cuanto al uso de recursos en comparación con procesos, debido a que comparten el mismo espacio de memoria y pueden comunicarse y sincronizarse más fácilmente.
    - En tercer y último lugar, tenemos que los fines principales de cada una de estas herramientas son distintos, y si bien las colas de mensajes pueden usarse para sincronizar como en este caso, su principal función es la comunicación entre procesos, función a la cual no le estamos sacando provecho aquí.

En cuanto a las desventajas de la elección de hilos y semáforos, es

que al compartir memoria, si cometemos errores al programar, podemos caer en condiciones de carrera más difíciles de detectar. esto último podría señalarse como una ventaja de elegir procesos, sumado a una mayor escalabilidad que utilizando hilos, ya que se podría aprovechar de mejor manera la distribución en múltiples núcleos del cpu diferentes máquinas.

1.2. MINI SHELL

El enunciado solicitaba implementar una *mini shell*, es decir, había que realizar una shell pero más reducida y más simple, soportando solamente un par de comandos específicos. Estos comandos eran:

- *help*: Mostrar una ayuda con los comandos posibles.
- *mkdir:* Crear un directorio.
- *rmdir:* Eliminar un directorio.
- *touch:* Crear un archivo.
- *ls:* Listar el contenido de un directorio.
- *cat:* Mostrar el contenido de un archivo.
- *chmod*: Modificar los permisos de un archivo.

Para que la experiencia utilizando la *mini shell* sea más acorde a una shell convencional, decidimos agregar el comando *exit* para poder salir de la ejecución.

El modelo de nuestra *mini shell* se basó en que hay un proceso padre el cual es el encargado de verificar que los comandos ingresados por el usuario sean correctos, que solamente pueda ingresar los comandos anteriormente

listados. Para esto, divide el input en tokens, y verifica que el primer token se corresponda a un comando válido. Si el comando es válido, el padre crea un hijo utilizando el comando fork(), y le carga una nueva imagen ejecutable mediante el comando execv() correspondiente al código que maneja el comando pedido. Por ejemplo, si el comando solicitado era *mkdir*, al proceso hijo se le carga la imagen correspondiente al archivo *Commandmkdir* para que pueda iniciar su ejecución, junto con los argumentos necesarios. Mientras el proceso hijo se ejecuta, el proceso padre lo espera. Después que termina el proceso hijo, se vuelve a repetir el ciclo hasta que el usuario lo desee (que finaliza la ejecución mediante *exit)*. *Aclaración: por motivos de eficiencia, el comportamiento del comando exit está implementado el proceso padre, que es únicamente decir que el ciclo de la ejecución del mini shell tiene que terminar.*

Tomamos como decisión de diseño que el input del usuario en la *mini shell* no iba a superar a los 1024 caracteres, y que la cantidad de tokens no iba a exceder del número 20. Esto lo decidimos de esta forma ya que se pensó una shell bastante simplificada, en la cual sería muy extraño que el usuario solicite algo que supere estos límites.

Luego, como se realiza un chequeo previo para ver si el comando puesto es válido (lo cual requiere recorrer un arreglo), para que en el momento de cargarle una nueva imagen ejecutable al proceso hijo no tenga que volver a preguntar qué comando era para cargarle el nombre del archivo específico al comando execv(), se decidió que cada archivo de código que se encarga del manejo de cada comando empiece por la palabra *Command*, así, solo resta concatenar el primer token a esta palabra para que quede el archivo deseado.

Claramente, como se pedía que únicamente podíamos simular cada comando mediante funciones de librerías, para los comandos se utilizaron las funciones de la librería <sys/stat.h>, <dirent.h> para el comando *ls,* pero para los comandos *touch, cat, help,* utilizamos meramente las librerías clásicas de C para el manejo de archivos, como la <stdio.h>.

Para poder ejecutar el código de la minishell, se creó un script de bash con el nombre *minishell.sh* que, antes de ejecutarlo, hay que modificar los permisos de ejecución para que no haya ningún inconveniente al intentar correr el script, y luego solo resta ejecutarlo normalmente dentro de una consola. Es decir, los pasos a seguir serían:

~ chmod +x minishell.sh

~ ./minishell.sh

Aclaración: El archivo Commandhelp, que maneja la ejecución del comando help, utiliza un archivo helpD.txt en el cual está escrito el texto que devolvería el pedir por help en la minishell. La función fopen asume que helpD.txt se encuentra en el mismo directorio que Commandhelp, por lo que si se cambia de lugar el archivo, el comando help no funciona debidamente.

2. Sincronización

2.1. SECUENCIA

2.1.1. Primera secuencia: ABABCABABCABABC...

Como lo explicita la consigna, para la primera resolución, utilizamos hilos y semáforos para poder realizar la sincronización. Por cada letra, creamos un hilo distinto, entonces un hilo se encarga de imprimir A, otro de imprimir B, y el último hilo se encarga de imprimir C. Luego, cada hilo tiene que esperar su turno de escritura. Para modelar los turnos de escritura, se usaron semáforos por cada turno, es decir, un semáforo sem_A, sem_B y sem_C. Sin embargo, la secuencia no era tan simple como para decir que empieza A, sigue B, sigue C, y así siguiendo. Para poder lograr la correcta, modelamos que el hilo encargado de imprimir A tiene que esperar dos señales antes de poder imprimir, y después le avisa a B que puede seguir. Cuando B recibe su señal, imprime B, pero en vez de avisarle solamente a C,

también manda una señal a A. C en un principio espera por su señal, y cuando llega la mandada por B, aumenta el valor del sem_A, y vuelve a esperar. Entonces, sumado con la señal de B y por esta nueva de C, se vuelve a imprimir A. En el momento que A manda la señal y B realiza lo mismo anteriormente explicado, ya es el turno de que C pueda imprimir, entonces el hilo imprime el carácter C, y termina mandando una señal a A, y todo se vuelve a repetir. De esta forma, se simula A B A B C A B A B C ...

Para poder ejecutar el código brindado, también se ha creado un script en bash llamado *primeraSecuenciaHilos.sh*, que tiene los mismos pasos que los explicados para la *minishell.* Hay que ejecutar los siguientes comandos en la consola:

~ chmod +x primeraSecuenciaHilos.sh

~ ./primeraSecuenciaHilos.sh

En la segunda parte de este enunciado, se solicitaba implementar la solución del mismo problema, salvo que en este caso se debían usar procesos y pipes para la sincronización. La idea es bastante similar, por no decir equivalente. Como tuvimos que utilizar procesos, hay un proceso distinto por carácter a imprimir. En el proceso padre se crean tres pipes distintos, uno que comunica A->B (la flecha indica dirección de la comunicación), B->C, y por último uno que conecta C->A y B->A, llamémoslos pipeAB, pipeBC, y pipeCBA. Las instrucciones correspondientes de los pipes para poder simular *wait* y *post* son *read* y *write*, respectivamente. Entonces, para simular la inicialización de lo que era el sem_A en 2, el proceso padre escribe en el pipeCBA dos letras C (era indistinto escribir C o B), luego crea los tres procesos hijos, A, B y C, en los cuales cada uno tiene el código específico para manejar la impresión del carácter que les tocaba. Desde este punto, la idea para manejar el patrón es igual: el proceso A espera por dos

mensajes, ya sean el carácter B o C, utilizando la función *read*. Cuando esto sucede, imprime 'A' y escribe en el pipeBC mediante la función *write* el carácter 'A', indicando que A acaba de imprimir y ya pasó su turno. Mientras tanto, el proceso B estaba esperando leer un carácter 'A' mandado por el pipeAB, cuando lo recibe, imprime 'B', y escribe en los pipes CBA y BC un carácter 'B', indicando que el turno de B acaba de pasar. Esto simula el *post(A)* y *post(C)* de la solución con hilos y semáforos. El proceso C realiza lo mismo, originalmente está esperando por el mensaje que le envía B, cuando este llega, escribe en el pipeCBA para avisarle al proceso A que tiene que volver a imprimir, y vuelve a esperar por otro mensaje. En el momento que el proceso A imprime nuevamente 'A', le manda el mensaje al proceso B y este imprime 'B' y manda los mensajes a los pipes CBA y BC, ahora el proceso C cuando recibe este nuevo mensaje imprime el carácter 'C', y manda el mensaje 'C' por el pipeCBA para indicar que su turno acaba de pasar, y de esta forma se repite el ciclo.

Claramente, en cada proceso, se cerraron los extremos de los pipes que no se usaban (es decir, si había usar el extremo de lectura, se cerraba el de escritura, y viceversa) y también se cerraron por completo los pipes que no se utilizaban a lo largo de cada código, para respetar el concepto de comunicación entre procesos mediante pipes.

Nuevamente, para ejecutar el código correspondiente a esta versión del problema, se creó un script bash llamado *primeraSecuenciaPipes.sh*, y para ejecutarlo hay que escribir los siguientes comandos en consola:

~ chmod +x primeraSecuenciaPipes.sh

~ ./primeraSecuenciaPipes.sh

2.1.2. Segunda secuencia: AB ABC ABCD AB ABC ABCD …

en este caso, tal y como fue solicitado para la primer secuencia, debimos plantear dos soluciones para resolver el problema, una utilizando hilos y semáforos y otra usando pipes y procesos.

- con hilos y semáforos:

  para esta primer resolución utilizamos cuatro semáforos, semA, semB, semC y semD, donde cada método (cada uno va a ser ejecutado por un hilo distinto) espera por su semáforo para ir avanzando a lo largo del código, y a su vez va enviando señales a algunos de los otros semáforos para que avance la secuencia manteniendo la sincronización necesaria.

  Como se desprende de lo anterior, va a haber 4 hilos ejecutando concurrentemente, donde cada hilo ejecuta un método distinto (cada uno es el encargado de printear una de las 4 letras de la secuencia). el método “printA” espera por el semáforo semA (inicializado en 1 para que la secuencia pueda comenzar), y cuando puede adquirir ese recurso, realiza el print de A y manda una señal al método printB para que este pueda continuar, mientras tanto printA vuelve a iniciar el loop y queda a la espera de una nueva señal en semA.

  por otra parte “printB” espera a una señal el el semáforo semB (que vendrá de printAA), y cuando la recibe, escribe B y envía una señal al semáforo semC, luego espera una nueva señal en el semB (que vendrá desde printC o desde printD, dependiendo de la iteración) y cuando la recibe envía señal a semA y comienza nuevamente el loop.

  con respecto a printC, este método espera señal en semC (que vendrá siempre de printB), envia señal a semB, espera nuevamente señal en semC (esta doble espera existe para saltear la iteración en la cual solo tiene que escribirse AB) y recién cuando puede tomar este recurso hace el primer print

de C y envía una nueva señal a semB, espera una última vez a semC, y cuando logra adquirir este recurso realiza el segundo print de C y envía señal a D.

Mientras tanto, el comportamiento de printD es el siguiente.

En cada iteración espera a una señal en el semáforo semD, y cuando puede adquirir efectivamente ese recurso, escribe D y envía una señal a semB.

Modelado de esta manera (utilizando des prints en el metodo printC), el código es más legible que usando un solo print pero teniendo que agregar más waits, y la sincronización entre los hilos es igual de correcta. las multiples esperas en printC hacen que se envíe la señal a D cuando es pertinente y se evite escribir C cuando solo debe escribirse AB.

para ejecutar el código correspondiente a esta versión del problema, se creó un script bash llamado *segundaSecuenciaHilos.sh*, y para ejecutarlo hay que escribir los siguientes comandos en consola:

~ chmod +x segundaSecuenciaHilos.sh

~ ./segundaSecuenciaHilos.sh

- con Procesos y Pipes:

  En esta resolución, la lógica es análoga al problema anterior, pero intercambiando las operaciones bloqueantes (wait) y las operaciones de envío de señales (post) por las operaciones read y write (operadas sobre pipes) respectivamente, y agregando un pipe extra para que no haya dos procesos que leen sobre el mismo pipe.

Aquí, en el flujo del proceso padre se van creando procesos con la operación fork() guardando su resultado en una variable y luego de verificar con un if que esa variable sea igual a cero (estamos en un proceso hijo) pasamos a ejecutar el código que está dentro de ese if. Este código se va a corresponder con el comportamiento explicado en la anterior resolución, agregando el cerrado de los extremos de los pipes que no se utilicen para administrar los recursos de manera eficiente y haciendo los cambios mencionados antes, esto es, reemplazando los waits por reads y los posts por writes, siempre sobre el pipe correspondiente para el caso.

para ejecutar el código correspondiente a esta versión del problema, se creó un script bash llamado *segundaSecuenciaPipes.sh*, y para ejecutarlo hay que escribir los siguientes comandos en consola:

~ chmod +x segundaSecuenciaPipes.sh

~ ./segundaSecuenciaPipes.sh

2.2. RESERVA DE AULAS

Según este problema, se nos pedía simular el hecho de que un aula tenía una sola computadora que puede ser reservada por los alumnos, únicamente por períodos individuales de una hora, entre las 9:00 y las 21:00 (asumimos que las 21h es un turno válido). Cada alumno (de un total de 25) realiza cuatro operaciones en total, y cada una de ellas podía ser: reservar un turno de la computadora, cancelar un turno, o consultar las reservas. La elección de la operación es al azar, pero estaba ponderada: la reserva tenía un 50% de probabilidades de salir, mientras que cancelar y consultar se

llevaban un 25% cada una. Luego, si al alumno le tocaba reservar, la hora del turno también sería al azar.

I. El enunciado nos dejaba varias libertades a la hora de implementar algunos hechos. Sobre las consultas que puede realizar cada alumno, asumimos que cada alumno solamente puede consultar sus reservas. Sobre las cancelaciones, cada alumno solamente puede cancelar sus propias reservas, y en nuestro modelo asumimos que el alumno cancela la primera reserva que encuentra suya en la tabla de reservas. Si el alumno no podía cancelar ninguna reserva porque no tenía ninguna, y si quería reservar un turno pero no podía porque ya no había lugares disponibles, asumimos que se consumía la operación realizada. Es decir, que se contaba como una operación de las cuatro que tenía que hacer, aunque no tuviera éxito.

   El programa principal larga los 25 hilos y todos los hilos empiezan desde una misma función de inicio, el cual es simplemente un ciclo que va de 0 a 3 para simular las cuatro operaciones que tiene que realizar cada alumno, y en este ciclo llama a *elegirOperacion*. Esta función es la que se encarga de realizar el azar, que simplemente elige un número entre 1 y 100, y si es menor o igual que 50 el alumno ejecuta *reservar*, si es menor o igual que 75 llama a *consultar*, y si no a *cancelar* (para esta última parte con el límite de menor o igual a 75 y si era entre 76 y 100, era indistinto llamar a *cancelar* o *consultar*, de esta forma estamos simulando el que cada operación tenga un 0.25 de probabilidad de salir). En el momento que cada hilo termina este ciclo, finaliza retornando 0, y el proceso de donde se ejecuta el *main* estaba esperando que cada hilo termine. Cuando esto sucede, simplemente termina la ejecución del programa.

   Este problema de reserva de aulas es una versión más compleja del clásico problema de lectores-escritores, por lo que el modelo partió desde esta idea. En este caso, decidimos darle prioridad a los lectores, es decir, los alumnos que consultan la tabla de reservas.

Para modelar la tabla de reservas utilizamos un arreglo de 13 lugares para cada turno válido (como se mencionó, los límites se tomaron como turnos posibles), el cual en un principio está inicializado en 0. Entonces, cada índice del arreglo representa una hora en específico. El índice 0 es la hora 0+9 = 9h, el índice 1 es la hora 1+9 = 10h, y así siguiendo. A su vez, usamos otro arreglo llamado *lugaresLibres*, en el cual se van guardando los turnos disponibles de la tabla de reservas, para que se elija al azar a partir de estos, y no pase que en la elección al azar caiga en un turno ya ocupado y tenga que volver a repetir hasta que encuentre uno disponible. Para la sincronización de los hilos, tenemos cuatro semáforos: *mutex, lectores*, *lugares* y *escribir*, los cuales tienen los siguientes propósitos:

- *mutex*: este semáforo funciona como semáforo binario, es el encargado de garantizar que únicamente *un solo hilo* pregunte a la vez si es el primer alumno que accede a consultar la tabla de reservas, o si es el último. En un inicio, está inicializado en 1.
- *lectores:* este semáforo funciona como contador de la cantidad de alumnos que están consultando la tabla de reservas. A partir de este semáforo se consulta si el valor del mismo es cero, y si esto se cumple, el alumno le quitará la posibilidad a los escritores (los alumnos que tienen que reservar o cancelar) de escribir en la tabla de reservas, o le volverá a dar vía libre cuando vuelva a quedar en cero (ya no hay más lectores), dependiendo del lugar que se realice la consulta. En un inicio, está inicializado en 0.
- *lugares:* este semáforo tiene el objetivo de ir guardando la cantidad de lugares libres que tiene la tabla de reservas, y los alumnos tratan de consumir el semáforo. Si pudieron hacerlo, quiere decir que es posible reservar un turno. Si no, el alumno sabe que ya no hay más lugares disponibles. En un inicio, está inicializado en 13.

- *escribir:* por último, este semáforo que funciona como binario tiene el rol de garantizar exclusión mutua en el momento que se escribe en la tabla de reservas, tanto para reservar un turno o para cancelarlo. El alumno que tenga que realizar una de estas operaciones, tratará de pedir el semáforo, y podrá acceder a editar la tabla de reservas cuando esté disponible (su valor sea 1), y cuando termina su operación, vuelve a incrementar el valor. En un inicio, está inicializado en 1.

II. Cada función que puede ejecutar el alumno funciona de la siguiente manera:
    - *reservar:* si esta fue la opción elegida por el alumno, primero consulta si hay lugar en la tabla de reservas con un *trywait*(*lugares*). Si esto es así, pide el semáforo *escribir* y cuando lo obtiene llama a *elegirReserva*. Esta última función simplemente elige un turno al azar de los disponibles y cuando lo obtiene guarda en la tabla de reservas el ID del hilo. Luego, cuando termina imprime por pantalla quién es (es decir, su ID), y la hora en la que reservó. Libera el semáforo, y termina. Si no había más lugares disponibles, imprime por pantalla su ID y que no pudo reservar.
    - *cancelar*: esta función lo primero que hace es pedir el semáforo *escribir*. Cuando lo obtiene, busca en la tabla de reservas a ver si algún valor en ella coincide con el ID del hilo. Esto quiere decir que el alumno había reservado un turno. En el momento que encuentra el primer turno, modifica la tabla de reservas volviendo a poner su valor en 0, libera un lugar haciendo *post(lugares)* y guarda en una variable la hora del turno. Luego, imprime por pantalla su ID y la hora del turno que canceló. Si no encuentra ningún turno (el alumno no tenía ninguna reserva), imprime por pantalla su ID y que no tenía turnos para cancelar. Luego, libera el semáforo y termina.

- *consultar*: como en esta implementación le dimos prioridad a los lectores, se podría decir que esta función es la más compleja de las tres. Lo primero que hace el alumno que le toca ejecutar esta función es pedir el *mutex* para que sea solo él el que pregunta si es el primer alumno que está consultando la tabla de reservas. Si esta consulta es verdadera, toma el semáforo *escribir*, para que ningún otro alumno pueda editar la tabla de páginas. Si no, incrementa el valor de *lectores* que acaba de consumir. Luego, incrementa el valor de *lectores*, indicando que hay un nuevo alumno consultando la tabla de reservas. Antes de ver los valores de la tabla de reservas, primero el alumno imprime por pantalla que quiere consultar. Después, recorre la tabla de reservas y si algún valor de la misma coincide con su ID, imprime por pantalla su ID y el turno que tiene reservado. Después de terminar este ciclo, decrementa el valor del semáforo *lectores*, para indicar que ya terminó de leer. Sin embargo, luego de esto tiene que preguntar si en realidad era el último lector en la tabla. Para esto, vuelve a pedir el *mutex* y realiza esta consulta. Si es positiva, libera el semáforo *escribir*. Si no, incrementa el valor de *lectores* porque acaba de consumir un valor de más. Libera el *mutex*, y termina.

III. En este inciso del proyecto se nos solicitaba pasar el mismo modelo implementado con hilos en el inciso anterior, pero ahora con procesos y memoria compartida. En definitiva, la mayor complejidad de este ejercicio es únicamente crear el correspondiente segmento de memoria compartida, y asociarla a cada proceso alumno que se va creando. Luego, la lógica de la solución es completamente la misma. El segmento de memoria compartida está constituido por los semáforos que se explicaron previamente en la solución con hilos, la tabla de reservas, y el arreglo de lugares libres. El programa principal

crea el segmento de memoria compartida y luego crea los 25 procesos alumnos mediante un *fork()* y si el fork fue exitoso, cada proceso hijo llama a la función *start()*. Mientras cada proceso hijo ejecuta las cuatro operaciones correspondientes, el proceso padre espera a que terminen. Luego de que termina cada proceso hijo, se destruyen los semáforos, se elimina el segmento de memoria compartida, y la ejecución finaliza. En cada función que ejecutan los procesos hijos, ahora fue necesario que reciba por parámetro el segmento de memoria compartida que cada proceso alumno asoció en la función *start()*, para así poder acceder a la tabla de reservas y modificar los valores de los semáforos.

Para poder ejecutar ambas versiones del problema, existen dos scripts bash llamados *reservaAulas.sh* y *reservaAulasMemoria.sh*, el primero le corresponde a la solución implementada con hilos y semáforos, y la segunda a procesos y memoria compartida. Como siempre, estos son los comandos necesarios de ejecutar en consola para ver el funcionamiento del código:

~ chmod +x reservaAulas.sh

~ ./reservaAulas.sh

—----------

~ chmod +x reservaAulasMemoria.sh

~ ./reservaAulasMemoria.sh

3. Problemas

3.1. LECTURA

En nuestro caso, tuvimos que investigar sobre el sistema QNX. Para la presentación, utilizamos un PowerPoint porque pensamos que sería una buena para mostrar gráficamente lo que habíamos investigado.

Las fuentes que utilizamos fueron las siguientes:

- https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.ide.userguide/topic/about.html
- https://techcrunch.com/2014/12/11/ford-ditches-microsoft-for-qnx-in-latest-in-vehicle-tech-platform/
- https://www.qnx.com/news/pr_1074_1.html
- https://financialpost.com/technology/apple-inc-carplay-ios-monday
- https://www.zdnet.com/article/blackberrys-qnx-why-its-so-valuable-to-apple-google-auto-industry/

3.2. PROBLEMAS CONCEPTUALES

3.2.1. 1)

a) Se nos pedía la cantidad de bits total de la dirección física. Esta incógnita se saca a partir de un dato del enunciado, el cual nos dice que la memoria física es de 2 GB. Luego, si pasamos este valor a bytes, nos da que la memoria física está constituida por 2147483648 B. Entonces, simplemente para averiguar la cantidad de bits tenemos que hacer $log_2(2147483648) = 31$.

Por lo tanto, la cantidad de bits de la memoria física es de 31.

b) En este caso, teníamos que averiguar el número de bits que especifican la sustitución de página y el número de bits para el número de marco de página.

   Para el número de bits que sustitución de página, es un total de varios bits de control. En un principio, fundamentalmente se tiene el bit de válido/inválido, para indicar que la página está en el espacio de memoria del proceso. Tenemos además el bit de referencia, para observar si la página fue usada o no. Otro bit altamente usado es el bit de modificado (dirty bit), que se utiliza para saber si la página fue modificada, es decir, se produjo una escritura en ella. Por lo tanto, tendríamos 3 bits.

   Por último, para averiguar el número de bits para el número de marco de página (frames), necesitamos saber en un principio cuántos números de frames existen. Para esto, nos vamos a basar en el número total del almacenamiento físico que es de 2147483648 B. Luego, si a esta cantidad la dividimos por el tamaño de los frames (en bytes), obtendremos el número de frames. Entonces, tenemos que: $2147483648\,B\,/\,(8\ *\ 1024)\ =\ 262144$ número de marcos de página en total. Ahora, simplemente resta saber cuántos bits necesitamos para toda esta cantidad, que lo obtenemos haciendo: $log_2(262144)\ =\ 18.$

   18 bits son necesarios para el número de frames.

c) Para este inciso, se pedía la cantidad de marcos de páginas. El proceso es bastante similar al anterior, pero ahora nos vamos a la parte de direccionamiento lógico. Sabemos que el espacio total de direcciones lógicas abarca 256 MB. Esta magnitud en bytes nos quedaría de: $256\ *\ 1024\ *\ 1024\ =\ 268435456\,B.$ Para averiguar la cantidad de marcos de páginas, a ese valor lo tenemos que dividir por el tamaño de página (que es igual al

tamaño de frame), entonces: $268435456\ B\ /\ (8\ *\ 1024\ B)\ =\ 32768$.

En total hay 32768 marcos de páginas.

d) Por último, había que indicar el formato de la dirección lógica. La dirección lógica se divide en dos: número de página y desplazamiento (offset). A cada uno le corresponde un número determinado de bits. Al número de página le corresponden tantos bits como para poder representar la cantidad total de las mismas. En el inciso anterior calculamos la cantidad de marcos de página, ahora, solo resta saber con cuántos bits podemos representar ese número, que lo obtenemos así: $log_2(32768)\ =\ 15$. Luego, para el desplazamiento, lo podemos obtener directamente del tamaño de una página, ya que este desplazamiento debe ser capaz de recorrer completamente cada una de ellas. Entonces, tenemos que: $log_2(8\ *\ 1024)\ =\ 13$.

   En conclusión, el formato de la dirección lógica está constituido por: del bit 0 al 14, es el número de página, y desde el bit 15 al 27 es el offset.

   | número de página (15 bits) | offset (13 bits) |

3.2.2. 2)

A. Tenemos la dirección 0, 228. El 0 nos indica cuál es el segmento que tenemos que analizar. En el 0 se tienen los siguientes datos: Dirección base: 830 | Largo: 346. Observamos que 228 < 346, por lo que no da un fallo de segmento. Entonces, sumamos 228 + 830 = 1058, que esta sería nuestra dirección física.
B. La dirección a analizar es 2, 648. Si nos posicionamos en la tabla de segmento en el número 2 obtenemos: Dirección base

1508 | Largo 408. Sin embargo, 648 > 408, nos pasamos del límite, por lo que se produce un fallo de segmento.

C. Ahora, hay que estudiar la dirección lógica 3, 776. El segmento 3 le corresponde: Dirección inicial 770 | Largo 812. En este caso, no se produce ningún fallo de segmento, ya que 776 < 812. Luego, la dirección física es: 776+770 = 1546.
D. Para este inciso, nos dan la dirección 1, 98. La tabla de segmentos, para el segmento 1, nos devuelve: Dirección inicial 648 | Largo 110. El algoritmo para buscar la dirección física sigue su camino porque 98 < 110. Por lo tanto, la dirección física a la cual nos estamos refiriendo es: 98 + 648 = 746.
E. Llegando al final del problema, la última dirección lógica para examinar es 1, 240. Nuevamente, tenemos que ver los valores del segmento 1 de nuestra tabla de segmentos: Dirección inicial 648 | Largo 110. En este caso, se produce un fallo de segmento, ya que 240 > 110, supera ampliamente el límite brindado.

4. Conclusión

Concluyendo, el proyecto ha sido una oportunidad invaluable para aplicar y consolidar los conocimientos adquiridos a lo largo de este período de estudio. A lo largo de este informe, hemos detallado cada uno de los desafíos abordados y proporcionado las herramientas necesarias para la ejecución de los archivos de código entregados. Este proyecto ha sido una experiencia que ha fortalecido nuestra comprensión de los conceptos fundamentales de los sistemas operativos y ha mejorado nuestras habilidades de resolución de problemas. A través de este trabajo, hemos demostrado nuestra capacidad para aplicar de manera efectiva los conceptos teóricos en un contexto práctico.