PROYECTO SISTEMAS OPERATIVOS

Mandolesi Bruno
Maslein Martin

Índice

1. Experimentación de Procesos y Threads con los Sistemas Operativos

1.1 procesos, threads y comunicación

1.1.1 Banco

a)

hilos: se crean 3 hilos de empleados , 2 empleadosEmpresa y 1 empleadoComun, después se crean 50 hilos de clientes este número fue elegido por nosotros pero podría ser modificado para distintos casos de prueba.

funciones:

- fun_empleado_comun(): esta función es la que se encarga de simular al empleado que atiende a los clientes comunes y políticos, lo que hace es fijarse si hay políticos en la cola para atender y si no los hay se queda esperando a que llegue un cliente de tipo común, esto quiere decir que si no habia politicos y pasó a esperar comunes aunque lleguen políticos no serán atendidos hasta que se atienda un cliente común, optamos por esta opción ya que fue brindada por la cátedra como idea de solución aunque de espera ocupada
- fun_empleado_empresas(): esta función es la que se encarga de simular al empleado que atiende a los clientes empresarios y políticos, lo que hace es fijarse si hay políticos en la cola para atender y si no los hay se queda esperando a que llegue un cliente de tipo empresario , esto quiere decir que si no había políticos y pasó a esperar empresarios aunque lleguen políticos no serán atendidos hasta que se atienda un cliente empresario , optamos por esta opción ya que fue brindada por la cátedra como idea
- fun_cliente(): esta función es la que se encarga de simular al cliente, este llega a la cola de la mesa principal luego se le asigna un tipo y es derivado a la cola específica de ese tipo luego el mismo es atendido por un empleado y se retira del banco, en caso de no tener espacio en la cola de la mesa principal se retira

semáforos:

- cola_mesa_comun: Inicializado en 0 para saber cuantos clientes comunes están listo para ser atendidos en la cola de los clientes comunes
- cont_mesa_comun: este semáforo es inicializado en 15 se usa para ver cuantos lugares disponibles hay en la cola de clientes comunes
- cola_mesa_empresas: Inicializado en 0 para saber cuántos clientes comunes están listo para ser atendidos en la cola de los clientes empresarios
- cont_mesa_empresas: este semáforo es inicializado en 15 se usa para ver cuantos lugares disponibles hay en la cola de clientes de empresas
- cont_mesa_principal: este semáforo es inicializado en 30 se usa para ver cuantos lugares disponibles hay en la cola de la entrada de la mesa principal, en caso de no haberlo deberá retirarse el cliente.
- cola_mesa_politicos: Inicializado en 0 para saber cuántos clientes comunes están listo para ser atendidos en la cola de los clientes políticos
- cont_mesa_politicos: este semáforo es inicializado en 15 se usa para ver cuantos lugares disponibles hay en la cola de clientes políticos
- turno_comun_inicio: este semáforo es inicializado en 0 y se usa para avisarle al cliente comun que está siendo atendido por un empleado
- turno_empresa_inicio: este semáforo es inicializado en 0 y se usa para avisarle al cliente empresario que está siendo atendido por un empleado
- turno_politico_inicio: este semáforo es inicializado en 0 y se usa para avisarle al cliente politico que está siendo atendido por un empleado
- turno_comun_fin: este semáforo es inicializado en 0 y se usa para avisarle al cliente comun que terminó de ser atendido por el empleado
- turno_empresa_fin: este semáforo es inicializado en 0 y se usa para avisarle al cliente de empresa que terminó de ser atendido por el empleado
- turno_politico_fin: este semáforo es inicializado en 0 y se usa para avisarle al cliente político que terminó de ser atendido por el empleado

Ejecución: se deberán ejecutar las siguientes líneas para correr el programa

- gcc -o bancoHilos EjercicioBancoProyecto.c -lpthread
- ./ bancoHilos

Debería mostrar como los clientes van entrando por la mesa principal y se derivan a sus respectivas mesas según su tipo y son atendidos por los empleados. Los clientes y los

empleados tienen un numero que los identifican para que sea mas visible quien es atendido y quien atiende

b)Para este ejercicio se eligio implementar el banco con cola de mensajes, hay un archivo en el cual se definen los clientes, otro en el cual se definen los empleados que atienden a empresarios y politicos y otro para el cual se define el empleado que define al empleado que atiende a clientes comunes y politicos, estos son comunicados por una llave como se mostro en el ejemplo provisto por la catedra. El algoritmo para esta solucion es el mismo al ejercicio anterior.

Tipos de mensajes:

- MAX_COLA_ENTRADA 30
- MAX_COLA_CLIENTE 15
- CONT_MESA_PRINCIPAL 1
- CONT_MESA_COMUN 2
- COLA_MESA_COMUN 3
- TURNO_COMUN_INICIO 4
- TURNO_COMUN_FIN 5
- CONT_MESA_EMPRESARIAL 6
- COLA_MESA_EMPRESARIAL 7
- TURNO_EMPRESARIAL_INICIO 8
- TURNO_EMPRESARIAL_FIN 9
- CONT_MESA_POLITICO 10
- COLA_MESA_POLITICO 11
- TURNO_POLITICO_INICIO 12
- TURNO_POLITICO_FIN 13

Ejecución: se deberán ejecutar las siguientes líneas para correr el programa

- gcc -o empleadoEmpresarial empleadoEmpresarial.c
- gcc -o empleadoComun empleadoComun.c
- gcc -o clientes clientes.c
- gcc -o banco banco.c
- ./ banco

Debería mostrar como los clientes van entrando por la mesa principal y se derivan a sus respectivas mesas según su tipo y son atendidos por los empleados. Los clientes y los

empleados tienen un numero que los identifican para que sea mas visible quien es atendido y quien atiende

c)

Implementación con Hilos y Semaforos:

Ventajas:

- Mayor eficiencia en recursos: Los hilos comparten memoria y recursos dentro de un proceso, lo que puede ser más eficiente en términos de uso de memoria y tiempo de CPU.
- Facilidad de sincronización: Los semáforos y otras estructuras de sincronización son más fáciles de implementar y gestionar, lo que puede simplificar la programación concurrente.
- Mayor flexibilidad: Los hilos permiten una mayor flexibilidad en términos de cómo se pueden diseñar y coordinar las tareas dentro del proceso, lo que facilita la implementación de estrategias de planificación y control personalizadas.

Desventajas:

- Mayor complejidad: La programación con hilos y semáforos puede ser más compleja y propensa a errores que el uso de colas de mensajes, ya que se deben tener en cuenta problemas como la exclusión mutua y la sincronización.
- Posibles condiciones de carrera: La compartición de memoria puede llevar a condiciones de carrera si no se gestionan adecuadamente, lo que puede resultar en errores difíciles de depurar.

Implementación con Colas de Mensajes:

Ventajas:

- Aislamiento de procesos: Las colas de mensajes proporcionan una separación más clara entre procesos, lo que puede mejorar la robustez y la tolerancia a fallos.
- Comunicación entre procesos: Las colas de mensajes son ideales para la comunicación entre procesos independientes y pueden utilizarse en sistemas distribuidos.

- Facilidad de depuración: Los problemas de comunicación a través de colas de mensajes suelen ser más fáciles de depurar que los problemas de concurrencia en hilos.

Desventajas:

- Mayor costo: La comunicación a través de colas de mensajes suele ser más lenta y conlleva una mayor costos en comparación con la comunicación entre hilos.
- Mayor complejidad en la programación: La programación con colas de mensajes puede ser más compleja, ya que los datos deben serializarse antes de enviarse y deserializarse después de la recepción.
- Menor eficiencia en el uso de recursos: Cada proceso que utiliza colas de mensajes tiene su propio espacio de memoria, lo que puede resultar en un mayor uso de recursos en comparación con los hilos.

1.1.2 Minishell

Para ejecutar la minishell, debemos correr el comando “chmod +x archivo.sh” para darnos permiso de administrador. Y luego correr el comando ./ejecutable.sh

El minishell hace uso de un ciclo while para su ejecución, el cual continuará hasta que el usuario ingrese el comando “salir”. Cuando se ingresa un comando, el padre crea un hijo al cual le carga la imagen de la rutina que deberá ejecutar según corresponda.

Los comandos implementados son los siguientes:

● crearArchivo

○ Comando encargado de crear un archivo en el directorio en donde se encuentra ubicado.

○ Formato del comando: crearArchivo [nombre_archivo]

● mostrarArchivo

○ Comando encargado de mostrar el contenido de un archivo.

○ Formato del comando: mostrarArchivo [path_archivo]

● cambiarPermisos

○ Comando encargado de cambiar los permisos de un archivo específico.

○ Formato del comando: cambiarPermisos [path_archivo] [permisos], donde [permisos] es un entero de 3 dígitos referenciando a los permisos de Owner, Group y Others respectivamente, y cada dígito sigue el siguiente formato:

■ 1 Ejecutar

■ 2 Escribir

■ 4 Leer

Permitiéndose la suma de los números, es decir, 7 (1+2+4), permitirá leer, escribir y ejecutar.

● crearDirectorio

○ Comando encargado de crear un directorio en la ubicación donde se encuentra la minishell.

○ Formato del comando: crearDirectorio [nombre_directorio]

● removerDirectorio

○ Comando encargado de remover un directorio en la ubicación donde se encuentra la minishell.

○ Formato del comando: removerDirectorio [nombre_directorio]

● listarDirectorio

○ Comando encargado de listar el contenido de un directorio.

○ Formato del comando: listarDirectorio [path_directorio]

● manual

○ Comando encargado de mostrar el manual del comando correspondiente.

○ Formato del comando: manual [comando_a_consultar]

Cada uno de los comandos fue implementado en una rutina específica, fuera del main, la cual es invocada cada vez que el usuario ingresa el comando con sus respectivos parámetros. Se provee un archivo makeFile.sh que contiene todos los comandos para compilar todos los archivos y generar el ejecutable para poder poner en marcha la minishell.

Observaciones:

Para los comandos que requieren las rutas de los archivos o los directorios, es necesario colocar la ruta completa de los mismos.

1.2 Sincronización

1.2.1 Secuencia

a)

Implementación con hilos y semáforos:

En este punto, estamos simulando tres hilos que imprimen letras en un patrón específico: "ABABC...". Cada hilo representa una letra: "A", "B" y "C".

Utilizamos semáforos para controlar el orden en que se imprimen las letras y asegurarnos de que sigan el patrón "ABABC...".

Declaramos tres semáforos: semA, semB y semC. Cada uno de estos semáforos actúa como una señal que permite que un hilo imprima su letra correspondiente y, al mismo tiempo, bloquea a los otros hilos.

Cada hilo tiene una función (imprimirA, imprimirB y imprimirC) que se ejecuta en un bucle infinito (while (1)). Estas funciones hacen lo siguiente:

La función imprimirA, espera a que semA se active, imprime "A" y luego permite que semB se active, esperando por otra activacion de semA.

La función imprimirB, espera a que semB se active, imprime "B" y luego permite que semC y semA se activen.

La función imprimirC, espera a que semC se active dos veces, imprime "C" y luego permite que semA se active dos veces. De esta manera logramos la alternancia de que se imprima C una vez cada dos.

En la función principal (main), creamos tres hilos (threadA, threadB y threadC) que ejecutan las funciones imprimirA, imprimirB y imprimirC, respectivamente.

Luego, inicializamos los semáforos semA con un contador de 2 (para que el hilo "A" pueda imprimir dos veces antes de bloquearse) y los semáforos semB y semC con un contador de 0.

ejecución:

- gcc -o sec_a sec_a.c -lpthread
- ./ sec_a

Implementación con procesos y pipes:

Este punto se basa en procesos y tuberías para imprimir la secuencia "ABABC..." infinitamente. Vamos a describirlo en detalle:

Se crean tres tuberías utilizando pipeA, pipeB, y pipeC. Cada tubería consta de dos descriptores de archivo: uno para lectura y otro para escritura.

Se escribe el carácter "a" en pipeA para comenzar el proceso. Está "a" inicial es necesaria para que el primer proceso (Proceso A) pueda comenzar la secuencia.

Se inicia un bucle que crea tres procesos hijos (Proceso A, B y C) usando fork(). Cada proceso hijo entra en un caso dependiendo de su valor de i (0 para Proceso A, 1 para Proceso B y 2 para Proceso C).

En el caso de Proceso A, se cierran los extremos de las tuberías que no necesita (B y C) y entra en un bucle infinito. Este proceso lee de pipeA, imprime "A", y luego escribe "a" en pipeB.

El Proceso B cierra las tuberías no utilizadas, lee de pipeB, imprime "B", escribe en pipeC, y luego, lee nuevamente de pipeB antes de escribir en pipeA.

Proceso C cierra las tuberías no utilizadas, lee de pipeC, escribe en pipeB, lee nuevamente de pipeC, imprime "C", y escribe en pipeB antes de comenzar de nuevo el bucle.

La ejecución continúa con todos los procesos en sus bucles infinitos, imprimiendo la secuencia "ABABC...".

El proceso principal (main) espera a que todos los procesos hijos terminen usando wait().

ejecución

- gcc -o sec2_a sec2_a.c
- ./sec2_a

b)

Implementación con hilos y semáforos:

En este punto, estamos buscando imprimir la secuencia "ABABCABCD...".

Se utilizan cuatro semáforos, semA, semB, semC, y semD, para sincronizar la ejecución de los hilos. Cada uno de estos semáforos se inicializa con un valor específico para controlar el orden de ejecución.

Se definen cuatro funciones, imprimirA, imprimirB, imprimirC, y imprimirD, que representan a los hilos que imprimirán las letras "A", "B", "C" y "D", respectivamente.

En imprimirA, el hilo espera (sem_wait) en semA, imprime "A", y luego señala (sem_post) en semB. Esto permite que el hilo de imprimirB imprima "B" después de que "A" haya sido escrita.

En imprimirB, el hilo espera en semB, imprime "B", señala en semC, espera en semB nuevamente, y finalmente señala en semA. Esto asegura que la secuencia "ABAB" sea impresa en ese orden.

En imprimirC, el hilo espera en semC y señala en semD. Luego, espera en semC nuevamente y señala en semB. Luego, espera en semC por última vez, lo que permite imprimir "C" en el momento que buscábamos después de la B.

En imprimirD, el hilo espera en semD, señala en semC, y repite este proceso tres veces. Luego, cuando se espera en semD por cuarta vez, se imprime "D", permitiendo que la secuencia siga con "ABCD...".

En la función principal (main), se inicializan los semáforos y se crean los hilos. Cada hilo representa una de las funciones de impresión. Los hilos se ejecutan en bucles infinitos y, por lo tanto, imprimirán la secuencia "ABABCABCD..." de manera continua.

ejecución

- gcc -o sec_b sec_b.c -lpthread
- ./sec_b

Implementación con procesos y pipes:

Este código tiene una estructura similar al código anterior secuencia con pipes, pero con cuatro procesos en lugar de tres. Cada proceso está diseñado para imprimir una letra ("A," "B," "C," o "D") en secuencia siguiendo el patrón "ABABCABCD...".

Se crean cuatro tuberías (pipeA, pipeB, pipeC, pipeD) para permitir la comunicación entre los procesos.

Se escribe un carácter "a" en pipeA[1] para iniciar el proceso de impresión.

Se utiliza un bucle para crear cuatro procesos hijos, uno para cada letra (A, B, C, D). Cada proceso tiene su propia lógica, como se explica a continuación.

El "Proceso A" espera en pipeA[0], imprime "A," y escribe en pipeB[1]. Este proceso se repite en un bucle infinito.

El "Proceso B" es similar al "Proceso A," pero espera en pipeB[0], imprime "B," y escribe en pipeC[1]`. También escribe y lee en las tuberías correspondientes para mantener el patrón.

El "Proceso C" sigue el mismo patrón, esperando en `pipeC[0], escribiendo y leyendo en las tuberías apropiadas, e imprimiendo "C."

El "Proceso D" es el último en la secuencia y sigue un patrón similar a los demás procesos. Espera en pipeD[0], escribe y lee en las tuberías y finalmente imprime "D."

Todos los procesos se ejecutan en bucles infinitos, lo que significa que imprimirán sus letras respectivas en secuencia de forma continua.

El proceso principal espera a que todos los procesos hijos terminen utilizando la función wait.

Este código implementa una forma de imprimir la secuencia "ABABCABCD..." utilizando cuatro procesos y tuberías para coordinar su ejecución y asegurar que las letras se impriman en el orden correcto.

ejecución

- gcc -o sec2_b sec2_b.c
- ./sec2_b

1.2.2 Reserva de aulas

1) Este código resuelve el problema en el que varios alumnos realizan operaciones de reserva, consulta y cancelación de aulas en un horario concreto.

Para ejecutarlo hay que compilarlo con el comando : -gcc -o aulas1 aulas1.c -lpthread

Para correrlo hay que ejecutar : ./aulas1

Definimos dos constantes: NUM_ALUMNOS y NUM_HORAS. Estas constantes representan el número de alumnos y el número de horas en el horario.

reserva_mutex: Un mutex que protege la zona crítica al acceder a la tabla de reservas.

reservas: Una tabla que representa las reservas de horas. Cada hora está representada por un elemento en el arreglo, donde 0 significa que la hora no está reservada y 1 significa que está reservada.

consulta: Un semáforo que se usa para controlar las consultas de los alumnos.

Inicialización de la Tabla de Reservas (inicializarTablaDeReservas): Inicializa la tabla de reservas, configurando todas las horas como no reservadas (0).

Función reservarHoraAleatoria: Genera y devuelve una hora aleatoria dentro del rango de las horas disponibles.

Función realizarReserva: Simula a un alumno reservando una hora. El alumno elige una hora aleatoria y verifica si está disponible. Si lo está, marca la hora como reservada y muestra un mensaje. Si no está disponible, muestra un mensaje de que no pudo realizar la reserva. Se utiliza un mutex (reserva_mutex) para evitar la concurrencia en la tabla de reservas.

Función realizarConsulta: Simula a un alumno consultando si una hora está reservada o disponible. El alumno elige una hora aleatoria y muestra un mensaje en función de la disponibilidad de la hora.

Función realizarCancelación: Simula a un alumno cancelando una reserva. Similar a la función de reserva, elige una hora aleatoria y, si está reservada, la marca como no reservada. Muestra un mensaje en consecuencia.

Función alumno: Esta función representa el comportamiento de un alumno. Un alumno realiza 3 operaciones aleatorias (reserva, consulta, cancelación). Las operaciones se eligen al azar, y se utilizan semáforos para gestionar las consultas de los alumnos. Si un alumno elige una operación de consulta, intenta adquirir el semáforo consulta. Si tiene éxito, realiza la consulta y luego libera el semáforo para permitir que otros alumnos consulten.

Función main: En la función principal, se realiza la configuración inicial. Se inicializa la tabla de reservas, el mutex y el semáforo. Luego, se crea un número de hilos igual a NUM_ALUMNOS, cada uno representando a un alumno. Cada hilo ejecuta la función alumno. Finalmente, se espera a que todos los hilos terminen antes de finalizar el programa.

Con procesos y memoria compartida:

El programa utiliza una estructura de memoria compartida llamada struct shared_data que contiene los siguientes elementos:

Para compilar el programa hay que ejecutar el comando: gcc -o aulas2 aulas2.c -lpthread

Para correr el compilable hay que ejecutar : ./aulas2

int reservas[NUM_HORAS]: Un arreglo de enteros que representa las horas disponibles. Cada elemento en este arreglo se inicia en 0, lo que indica que la hora está disponible. Cuando un alumno reserva una hora, se marca como 1 para indicar que está reservada.

sem_t consulta: Un semáforo utilizado para controlar el acceso a la función de consulta. Solo un alumno puede consultar una hora a la vez.

sem_t semaforo: Un semáforo utilizado para controlar el acceso a la tabla de reservas. Evita que múltiples alumnos reserven o cancelen la misma hora al mismo tiempo.

inicializarTablaDeReservas(struct shared_data *shared): Esta función inicializa el arreglo de reservas en la estructura compartida, marcando todas las horas como disponibles.

reservarHoraAleatoria(): Genera una hora aleatoria que un alumno intentará reservar.

realizarReserva(int alumno, struct shared_data *shared): Intenta reservar una hora aleatoria. El alumno bloquea el semáforo semaforo para evitar conflictos con otros alumnos. Si la hora está disponible, la marca como reservada y muestra un mensaje. En caso contrario, muestra un mensaje indicando que no pudo reservarla.

realizarConsulta(int alumno, struct shared_data *shared): Intenta consultar una hora aleatoria. El alumno bloquea el semáforo semaforo para evitar conflictos con otros alumnos y verifica si puede bloquear el semáforo consulta. Si puede hacerlo, consulta la hora y muestra si está disponible o reservada. Si no puede bloquear consulta, muestra un mensaje indicando que no pudo realizar la consulta.

realizarCancelacion(int alumno, struct shared_data *shared): Intenta cancelar una hora médica que previamente había reservado. El alumno bloquea el semáforo semaforo para evitar conflictos con otros alumnos. Si la hora está reservada, la marca como disponible y muestra un mensaje. En caso contrario, muestra un mensaje indicando que no pudo cancelarla.

La función alumno(void *arg) se ejecuta en hilos separados para simular el comportamiento de los alumnos. Cada alumno tiene un ID único y accede a la estructura de memoria compartida. Realiza cuatro operaciones diferentes (reserva, reserva, consulta o cancelación) de manera aleatoria. Las operaciones están sincronizadas mediante semáforos para evitar conflictos.

La función principal main() crea la memoria compartida y los semáforos. Luego, inicia un número de hilos igual a NUM_ALUMNOS, donde cada hilo representa un alumno. Los hilos acceden a la estructura de memoria compartida para realizar operaciones en horarios aleatorios. Al final, se liberan los recursos y se destruyen los semáforos.

ejecución

2.Problemas

2.2 Problemas conceptuales

2.2.1 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.

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

Tamaño total de la memoria física: 2 GB.
Tamaño de las páginas: 8 KB.

Número de páginas = Tamaño total de la memoria física / Tamaño de las páginas
Número de páginas = (2 GB) / (8 KB)

2 GB = 2 * 1024 MB = 2 * 1024 * 1024 KB = 2 * 1024 * 1024 * 1024 bytes
8 KB = 8 * 1024 bytes

Ahora, calculemos el número de páginas:

Número de páginas = (2 * 1024 * 1024 * 1024 bytes) / (8 * 1024 bytes)
Número de páginas = (2^31 bytes) / (2^13 bytes)
Número de páginas = 2^(31 - 13) páginas
Número de páginas = 2^18 páginas

Número de bits para direccionar cada página = log2(2^18)
Número de bits para direccionar cada página = 18 bits

Como el tamaño de las páginas es de 8 KB, se requieren 13 bits para direccionar los 8 KB posibles en cada página (ya que 2^13 = 8,192).
Entonces, el número total de bits en la dirección física es la suma de los bits para la página y los bits para el desplazamiento:

Número total de bits en la dirección física = Bits para la página + Bits para el desplazamiento
Número total de bits en la dirección física = 18 bits + 13 bits
Número total de bits en la dirección física = 31 bits

Por lo tanto, el número total de bits en la dirección física es de 31 bits. Estos 31 bits se utilizan para identificar una ubicación específica en la memoria física.

b) **Determine el número de bits que especifican la sustitución de página y el número de bits para el número de marco de página. (bit de valido, podríamos sumar 2 bits más de, 1 de referenciamiento y otro para modificado o dirty bit) (revisar)**

Ya sabemos que el número de páginas en la memoria es de 2^18 (como se calculó en el inciso anterior). Ahora, necesitamos calcular el número de marcos de página, que es igual al número de páginas que pueden caber en la memoria física.

Número de marcos de página = Tamaño total de la memoria física / Tamaño de las páginas
Número de marcos de página = (2 GB) / (8 KB)

Número de marcos de página = (2^31 bytes) / (2^13 bytes)
Número de marcos de página = 2^(31 - 13) marcos
Número de marcos de página = 2^18 marcos

Número de bits para el número de marco de página = log2(2^18)

Número de bits para el número de marco de página = 18 bits

Estos 18 bits se utilizan para especificar el número de marco de página al que se debe cargar una página específica.

En cuanto al número de bits que especifican la sustitución de página, esto depende del algoritmo de sustitución de página que se utilice. Si se utiliza el algoritmo de sustitución de página óptimo, entonces no se necesitan bits adicionales para especificar la sustitución, ya que este algoritmo se basa en la elección óptima de la página a reemplazar en función del futuro. Sin embargo, si utilizamos un algoritmo como FIFO, LRU, o algún otro, se requerirá un número adicional de bits para llevar un registro de la política de sustitución.

En resumen, en este caso, se necesitan 18 bits para especificar el número de marco de página, pero los bits para especificar la sustitución de página depende del algoritmo específico utilizado.

c) **Determine el número de marcos de página.**

Tamaño total de la memoria física: 2 GB.
Tamaño de las páginas: 8 KB.

Tamaño total de la memoria física en bytes:
2 GB = 2 * 1024 MB = 2 * 1024 * 1024 KB = 2 * 1024 * 1024 * 1024 bytes = 2,147,483,648 bytes

Número de marcos de página = Tamaño total de la memoria física / Tamaño de las páginas
Número de marcos de página = 2,147,483,648 bytes / 8,192 bytes
Número de marcos de página = 262,144 marcos de página

Entonces, hay un total de 262,144 marcos de página en la memoria física de este sistema de gestión de memoria basado en paginación. Cada marco de página tiene un tamaño de 8 KB y se utiliza para almacenar una página de un proceso.

d) **Determine el formato de la dirección lógica.**

Espacio de direcciones lógicas de cada proceso en bytes:
256 MB = 256 * 1024 KB = 256 * 1024 * 1024 bytes = 268,435,456 bytes

Tamaño de las páginas:
8 KB = 8 * 1024 bytes = 8,192 bytes

Número de páginas que pueden abarcar 256 MB:
Número de páginas = Espacio de direcciones lógicas de cada proceso / Tamaño de las páginas
Número de páginas = 268,435,456 bytes / 8,192 bytes
Número de páginas = 32,768 páginas

Ahora, necesitamos determinar cuántos bits se necesitan para direccionar 32,768 páginas.
Número de bits para direccionar páginas = log2(32,768) = 15 bits aproximadamente.
Offset = 28 bits - 15 bits de dirección de paginas = 13 bits de offset.

2.2.2 Considere un sistema de segmentación simple que tiene la siguiente tabla de segmentos.

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

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

a) **0, 228.**

La dirección lógica 0, 228 cae en el segmento 0 si está dentro de la dirección inicial 830 y el largo del segmento 346.
El segmento 0 abarca las direcciones desde 830 hasta 1,175 (830 + 346 - 1).
Dado que 0, 228 está dentro de este rango (0 ≤ 228 ≤ 1,175), la dirección lógica está dentro del segmento 0.
Dirección física = Dirección Inicial del Segmento 0 + Desplazamiento = 830 + 228 = 1,058.
Entonces, la dirección física correspondiente a la dirección lógica 0, 228 es 1,058.

b) **2, 648.**

La dirección lógica 2, 648 se refiere al número de segmento 2.
Como estamos buscando el número de segmento 2, este número de segmento es válido en el sistema.
La dirección inicial del segmento 2 es 1,508 y su longitud es de 408 bytes.
El segmento 2 abarca las direcciones desde 1,508 hasta 1,915 (1,508 + 408 - 1).
Como 648 no está dentro de este rango (1,508 ≤ 648 ≤ 1,915), se produce un fallo de segmento.

c) **3, 776.**

La dirección lógica 3, 776 se refiere al número de segmento 3.
Dado que estamos buscando el número de segmento 3, este número de segmento es válido en el sistema.
La dirección inicial del segmento 3 es 770 y su longitud es 812 bytes.
El segmento 3 abarca las direcciones desde 770 hasta 1,581 (770 + 812 - 1).
Dado que 776 está dentro de este rango (770 ≤ 776 ≤ 1,581), la dirección lógica 3, 776 está dentro del límite del segmento 3.

Dirección física = Dirección Inicial del Segmento 3 + Desplazamiento = 770 + 776 = 1,546.

d) 1, 98.

La dirección lógica 1, 98 se refiere al número de segmento 1.
Como estamos buscando el número de segmento 1, este número de segmento es válido en el sistema.
La dirección inicial del segmento 1 es 648 y su longitud es 110 bytes.
El segmento 1 abarca las direcciones desde 648 hasta 757 (648 + 110 - 1).
Dado que 98 está dentro de este rango (648 ≤ 98 ≤ 757), la dirección lógica 1, 98 está dentro del límite del segmento 1.
Dirección física = Dirección Inicial del Segmento 1 + Desplazamiento = 648 + 98 = 746.

e) 1, 240.

La dirección lógica 1, 240 se refiere al número de segmento 1.
Como estamos buscando el número de segmento 1, este número de segmento es válido en el sistema.
La dirección inicial del segmento 1 es 648 y su longitud es 110 bytes.
Dado que 240 está dentro de este rango (110 ≤ 240), la dirección lógica 1, 240 está fuera del límite del segmento 1. Por lo tanto es un fallo.