Sistemas Operativos
1° Parcial 2C2017 - TM - Resolución

Aclaración: La mayoría de las preguntas o ejercicios no tienen una única solución. Por lo tanto, si una solución particular no es similar a la expuesta aquí, no significa necesariamente que la misma sea incorrecta. Ante cualquier duda, consultar con el profesor del curso.

Teoría

1. Los algoritmos sin desalojo sólo re-planifican cuando se libera la CPU, es decir, cuando los procesos la abandonan voluntariamente (finalizan o se bloquean). Los algoritmos con desalojo consideran además otros eventos, esto es, por ejemplo, cuando hay un proceso nuevo en la cola de listos (desde nuevo o bloqueado) o si es que se implementa un quantum.
Sin desalojo se podría usar en un sistema que no sea crítico (no tenga procesos de tiempo real que tengan que ejecutar sí o sí), en los que se quiera minimizar el overhead por process switches extras.

2. V o F:

a .Falso. Luego de cada ciclo de instrucción se valida si es que hay interrupciones por lo que se atenderá al finalizar de ejecutar la instrucción en curso. Luego de atender la interrupción seguirá con la syscall.

b. Verdadero. La comunicación entre distintos módulos del SO ahora tiene que realizarse a través de paso de mensajes pasando por el kernel, lo cual genera un overhead extra.

3. Debería asignar un inodo, modificando el bitmap de inodos. Asignar los bloques necesarios, modificando el bitmap de bloques. También se modificará el inodo con la información del inodo. Se creará la entrada de directorio apuntando a dicho inodo. Se actualizará también el superbloque. Al crear el hardlink se modifica el inodo y se agrega una entrada en el directorio donde es creado.
Si se abre un archivo se lo buscará en la tabla de archivos abiertos, si el mismo no se encuentra abierto se cargará el FCB en memoria sino, se incrementará en un el contador de aperturas de dicho archivo. Luego se creará una entrada en la tabla de archivos abiertos del proceso apuntando a la entrada global e indicando el modo de acceso.

4 . Cumple con mutua exclusión ya que sólo un proceso podrá adquirir el semáforo a la vez, el resto se quedarán bloqueados. Cumple progreso ya que los únicos que pueden afectar que un proceso pueda adquirir el semáforo son los que también están realizando el wait. Cumple con espera limitada ya que los procesos se bloquearán en una cola de dicho semáforo y luego se irán despertando a medida que se haga signal respetando el orden de bloqueo. No depende de la velocidad de los procesos.
Los wait y signal operan sobre los semáforos que son compartidos, por lo que también

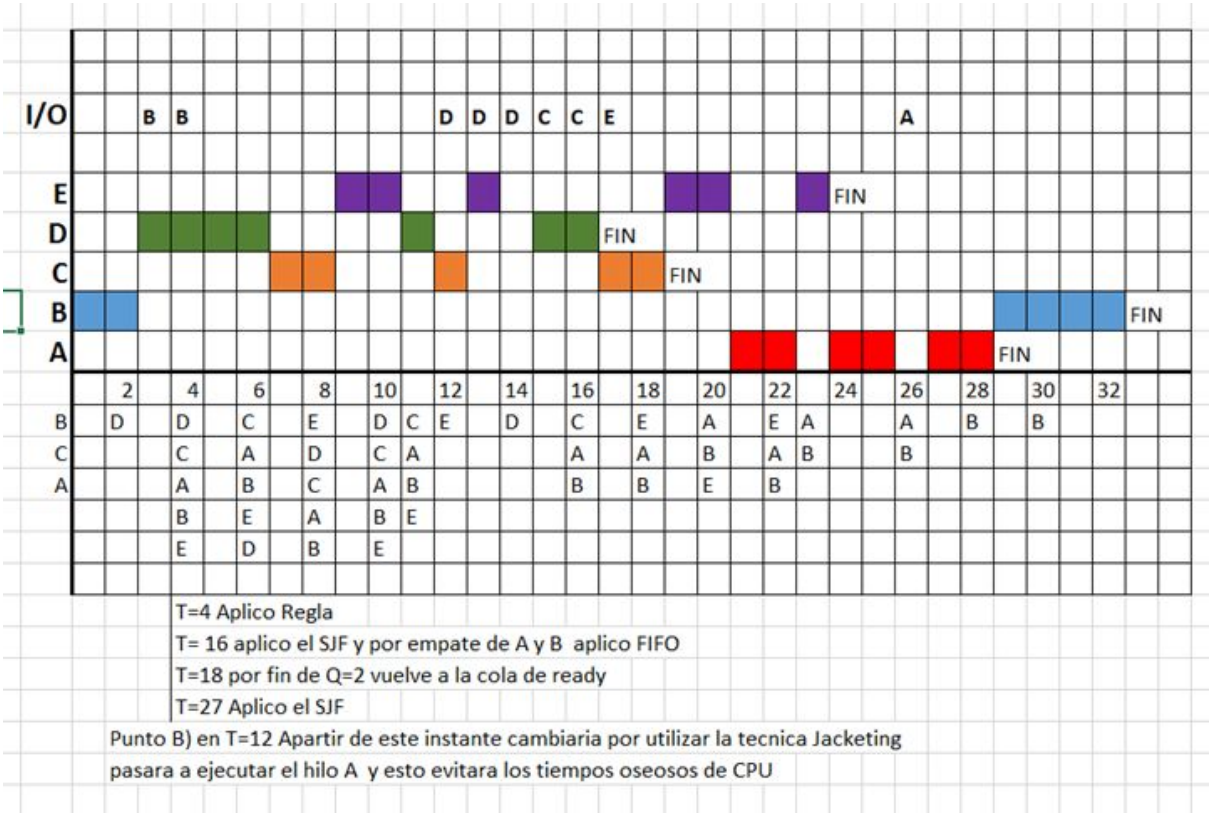
tienen una región crítica. Para que sea atómica el SO podría usar una técnica de HW como por ejemplo testAndSet.

5.

- Se guarda una parte inicial del contexto (PC, FLAGS) en el stack del sistema.
- El SO ingresa su ejecución y guarda el resto de los registros en el stack del sistema.
- El SO determina que debe realizar un thread switch.
- Mueve el contexto de ejecución completo del stack del sistema al TCB del hilo.
- Elige el próximo hilo a ejecutar.
- Copia todo el contexto del TCB del nuevo hilo al procesador y realiza un cambio de modo (a usuario), todo al mismo tiempo.
- Aclaración: No se actualizan registros relacionados con punteros a segmentos de memoria estáticos, heap o código, dado que pertenecen al proceso y son compartidos por todos los hilos.

Práctica

Ejercicio 1



Ejercicio 2

1) 8GiB=2^28xT;
T=32Bytes;
15MiB/32Bytes=491520 bloques

2) 15MiB / 8Kib = 1920 Bloques
1920- 10 Directos – 256 Ind Simples = 1654 Bloques que faltan del Ind doble.
1654 / 256 = 6,46 = 7 Indirecciones.

Cantidad de accesos = 1920 Bloques de Datos + 1 Indirección Simple + 8 indirecciones Dobles
Cantidad de accesos = 1929

3) 15MiB (15728640 bytes) + 8.574.205.940 bytes= 8.589.934.580 bytes. Como el FS tiene 8GiB totales = 8x2^30-8.589.934.580 = 12 bytes libres. Con esto solo estamos contemplando el espacio utilizado por los datos, sin tener en cuenta el lugar ocupado por los

bloques de punteros. Claramente vemos que no hay espacio suficiente para alocar estos bloques, con lo cual, no es posible cargar el archivo con esta configuración de UFS.
Una solución posible, sería modificar la estructura del i-nodo para que tenga solo punteros directos, evitando así, usar bloques de punteros.

8.574.205.940 bytes / 2^13 bytes/bloque => 1.046.656 bloques => 1.046.656 punteros directos.

Con esta configuración, podríamos tener ambos archivos en el mismo UFS FS, con 12 bytes de sobra (no teniendo en cuenta lo que ocupan las estructuras administrativas en el FS).

Ejercicio 3

tarjeta_insertada = tarjeta_leida = pin = pin_validado = cheque = respuesta = billetes = pedido_billetes = 0;
lectura_cheque = ok_cheque = 0; mutex = 1;

<table>
<tr><th>Usuario (N)</th><th>Cajero (1)</th><th>Lector de cheques (1)</th></tr>
<tr>
<td rowspan="2">wait(mutex);
insertar_tarjeta( );
signal(tarjeta_insertada);
wait(tarjeta_leida);
colocar_pin("1234");
signal(pin);
wait(pin_validado);
insertar_cheque( );
signal(cheque);

wait(respuesta);
if(rta == "OK") {
  guardar_billetes( );
} else {
  guardar_cheque( );
}
signal(mutex);</td>
<td rowspan="2">wait(tarjeta_insertada);
leer_tarjeta( );
signal(tarjeta_leida);
wait(pin);
validar_pin( );
signal(pin_validado);
solicitar_cheque( );
wait(cheque);
signal(lectura_cheque);

wait(ok_cheque);
if(lectura == "OK") {
  pedir_billetes_a_contadora( );
  signal(pedido_billetes);
  wait(billetes);
  entregar_billetes( );
  rta = "OK";
} else {
  devolver_cheque( );
}
signal(respuesta);</td>
<td>wait(lectura_cheque);
leer_cheque( );
lectura = validar( );
signal(ok_cheque);</td>
</tr>
<tr>
<td>Contadora de billetes (1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wait(pedido_billetes);
contar( );
enviar_billetes_al_cajero( );
signal(billetes);</td>
</tr>
</table>