Departamento de Arquitectura

Segundo parcial de Sistemas Operativos

2 de diciembre de 2022

Lea detenidamente las siguientes instrucciones. No cumplir los requerimientos puede implicar la pérdida del parcial.

Formato:

- Indique su nombre completo y número de cédula en cada hoja (no se corregirán las hojas sin nombre). Numere todas las hojas e indique la cantidad total de hojas en la primera.
- Escriba las hojas de un solo lado y empiece cada problema en una hoja nueva.
- Si se entregan varias versiones de un problema solo se corregirá la primera de ellas.

Dudas:

- Sólo se contestarán dudas de letra.
- No se aceptarán dudas en los últimos 15 minutos del parcial.

Material:

- El parcial es **SIN** material (no puede utilizarse ningún apunte, libro ni calculadora). Sólo puede tenerse las hojas del parcial, lápiz, goma y lapicera en su banco. Todas sus demás pertenencias debe colocarlas en el piso debajo de su asiento.

Finalización:

- El parcial dura **3 horas**.
- Al momento de finalizar el parcial no se podrá escribir absolutamente nada en las hojas, debiéndose parar e ir a la fila de entrega. Identificar cada una de las hojas con nombre, cédula y numeración forma parte de la duración del parcial.

Importante: Debe justificar todas las respuestas. Siempre.

Problema 1 (20 pts)

Se tiene un sistema operativo con un único procesador y un planificador expropiativo. Inicialmente el sistema no tiene ningún proceso ejecutando y en t=0 se lanza la ejecución del proceso P1 que ejecuta la rutina R1.

R1	R2
pid = fork() create_thread(R2) pid = pid +1 bloquea 2ms	Ejecuta 1ms Bloquea 3ms print (pid % 2)

(a) (14 pts) Realice un diagrama de planificación (tiempo vs hilos). El planificador utiliza el algoritmo Shortest Job First (SJF) y los hilos están implementados con el modelo 1x1.

Tenga en cuenta que:

- En caso de empates entre hilos de **distintos** procesos se asignará la CPU al hilo cuyo identificador de proceso sea mayor. En caso de que sean del **mismo** proceso se asignará al que su identificador de hilo sea mayor
- Se asume que la última función de los bloques de ejecución es la encargada de bloquear al proceso cuando corresponde (i.e. el proceso no debe esperar a tener nuevamente la CPU para bloquearse).
- Salvo que se indique explícitamente el tiempo de ejecución de las operaciones (suma, módulo, asignación, fork, create_thread) es de 1ms.

Solución:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
P1H1	E	L	L	L	L	L	E	L	L	E	L	E	E	B	B	T	
P1H2	-	-	-	-	-	-	-	-	-	-	E	B	B	B	E	E	T
P2H1	-	E	E	L	E	E	B	B	T								
P2H2	-	-	-	E	B	B	B	E	E	T							

(b) (2 pts) ¿Qué valores aparecen en pantalla? Escríbalos en orden de aparición.

Solución: Se asume que el pid del proceso hijo es su subíndice (2), era aceptable asumir otros valores mientras se aclarara.

Para calcular las salidas debe tenerse en cuenta qué hay guardado en la variable pid (que no es el pid del proceso). En el caso del proceso 1 se guardó la salida del fork (pid del hijo = 2) y luego se le sumó 1, con lo cual su valor final es 3. Por otro lado en el proceso 2 se guardó la salida del fork (0) y luego se le sumó 1, por tanto su resultado es 1. Luego de aplicar el módulo ambos tienen resultado 1 con lo que se imprimirá 1 1 siendo el primero correspondiente al print de P2H2 y el segundo al de P1H2.

(c) I. (2 pts) Calcule los tiempos de retorno de cada hilo.
II. (2 pts) Defina tiempo de espera de un proceso.

Solución: Tiempos de retorno:

P1H1: 13 - 0 = 13
P1H2: 16 - 10 = 6
P2H1: 8 - 1 = 7
P2H2: 9 - 3 = 6

Tiempo de espera: Es el tiempo que un proceso pasa en la cola de listos, es decir el tiempo que pasa a la espera de que se le asigne el procesador.

Problema 2 (20 pts)

Se tiene una arquitectura que maneja direcciones de memoria de **32** bits e implementa un mecanismo jerárquico de dos niveles para la paginación de memoria. Cada dirección de memoria virtual cuenta con **8** bits para la tabla de primer nivel, **8** bits para la tabla de segundo nivel y **16** bits para el desplazamiento. Cada entrada de tabla de páginas tiene un tamaño de **32** bits: los primeros **16** bits son utilizados para la dirección de la página, los siguientes **15** bits son utilizados para mecanismos de control (e.g. permiso lectura/escritura, marca accedido/modificado, etc.) y el último bit es el valid-invalid bit. Si el valid-invalid bit es **0** y la página se encuentra en el swap, entonces los primeros **16** bits de la entrada indican la ubicación de la página en el disco. Si el valid-invalid bit es **0** y los primero **16** bits también son **0**, entonces la página no es parte del espacio de memoria del proceso. En este sistema el acceso a una palabra de 32 bits en memoria principal requiere **100** ns (100×10^{-9} s), el tiempo de búsqueda en la TLB requiere **12** ns (12×10^{-9} s) y el tiempo de resolución de un fallo de página es de **15** ms (15×10^{-3} s).

(a) (3 pts) Realice un esquema donde se muestre el mecanismo de traducción de una dirección virtual a una dirección física en la arquitectura planteada, mostrando los principales componentes de hardware que participan en la traducción.

Solución: Ver teórico Administración de Memoria II: slide 14

(b) Se tiene un proceso en ejecución, P1, con el estado de memoria indicado en la figura 1.
Se pide:

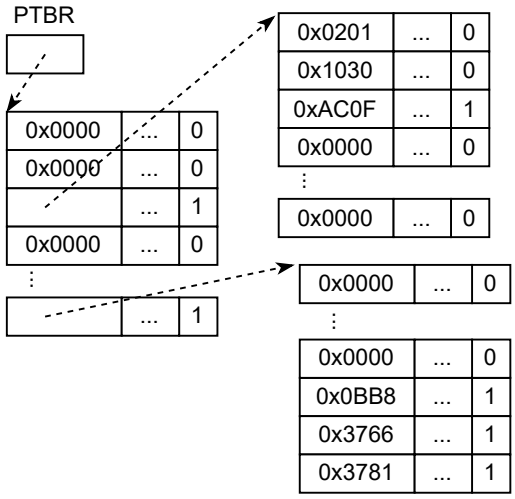

Figura 1: Estado inicial de la memoria del proceso P1.

I. (2 pts) Calcule el espacio de memoria virtual asignado al proceso en bytes.

Solución: Tenemos dos tablas de 2do nivel. En la primera hay 3 páginas con contenido y en la segunda otras 3, en total 6 páginas. Cómo el offset es de 16 bits, entonces cáda página tiene 64KiB. En total tenemos 6x64KiB=386KiB de memoria virtual asignada al proceso.

II. (2 pts) Calcule la memoria residente del proceso en bytes.

Solución: De las 6 páginas del proceso, solo 4 están cargadas en memoria. Así que la memoria residente del proceso es 4x64KiB=256KiB de memoria RAM.

III. (3 pts) Indique qué sucede, en cada caso, si el proceso accede a alguna de las direcciones virtuales: `0x0200F0F1`, `0x02020C01` y `0x02FF0001`.

Solución:

- `0x0200F0F1` → Se accede a una dirección válida pero de una página que no está residente en memoria. Se produce un fallo de página y el SO debe cargar la página a memoria RAM.
- `0x02020C01` → Se accede a una dirección válida de memoria residente.
- `0x02FF0001` → Se accede a una dirección inválida de memoria. Se produce un fallo de página y el SO en general termina el proceso que realizó el acceso.

IV. (3 pts) Dada la siguiente secuencia de accesos a direcciones virtuales: `0x0201A111`, `0x0201F00B` y `0x0202F301`. Calcule el tiempo total requerido para resolver estos pedidos. Suponga que la memoria TLB se encuentra vacía, y que la TLB y la memoria principal no requerirán de la ejecución del algoritmo de reemplazo durante la ejecución del proceso.

Solución:

1. `0x0201A111` → Se busca en la TLB y se produce un cache miss porque está vacía, esto requiere **12** ns. Luego se busca en la tabla de primer nivel y en la de segundo nivel, eso

requiere 2 accesos a memoria, por lo tanto 2×100 ns. Por último, la página accedida no se encuentra en memoria residente por lo que se requiere 15 ms para resolver el fallo de página. Por último se requiere un acceso adicional a memoria para obtener el dato solicitado por el proceso, 100 ns. En total: $312 \times 10^{-9} + 15 \times 10^{-3} \simeq 15 \times 10^{-3}$ s.

2. 0x0201F00B $\rightarrow$ Se vuelve a acceder la misma página que el acceso anterior. Se busca en la TLB y es produce un cache hit porque el acceso anterior fue agregado al caché, esto requiere 12 ns. Luego simplemente se accede a memoria para obtener el dato solicitado por el proceso, 100 ns. En total: 112×10^{-9} s.

3. 0x0202F301 $\rightarrow$ Se busca en la TLB y se produce un cache miss porque está vacía, esto requiere 12 ns. Luego se busca en la tabla de primer nivel y en la de segundo nivel, eso requiere 2 accesos a memoria, por lo tanto 2×100 ns. Por último, la página accedida se encuentra en memoria residente por lo que se requiere 100 ms para accederla. En total: 312×10^{-9} s.

(c) (4 pts) Suponga ahora otro proceso en ejecución, P2, con el siguiente requerimiento de memoria: 2 GiB de memoria para los segmentos de código, datos y heap, y 3 MiB para el segmento de stack. Calcule cuántas páginas de memoria son necesarias para este proceso (incluya en el cálculo las páginas usadas para la tabla de páginas).

Solución: 2 GiB es 2×2^{30} bytes. Cada página contiene 2^{16} bytes, por lo tanto se necesitan $\frac{2^{31}}{2^{16}} = 2^{15}$ páginas. Cómo cada tabla de 2do nivel puede direccionar 2^8 páginas, entonces se requieren $\frac{2^{15}}{2^8} = 2^7 = 128$ tablas de segundo nivel. La tabla de 1er nivel puede direccionar hasta $2^8 = 256$ tablas de 2do nivel, por lo que tendrá la mitád de sus entradas ocupadas por los segmentos de código, datos y heap.

Además, se requieren 3×2^{20} bytes para el segmento de stack (3 MiB). Por lo tanto son necesarias $\frac{3 \times 2^{20}}{2^{16}} = 3 \times 2^4 = 48$ páginas. Alcanza por lo tanto con una tabla de segundo nivel adicional.

En total se requieren $2^{15} + 48$ páginas para la memoria del proceso y $128 + 1$ páginas para las tablas de segundo nivel.

(d) (3 pts) Suponga un estado del sistema en el que se tiene un hit ratio de 0,8 de la TLB. Calcule el tiempo promedio de acceso efectivo a memoria en esta situación para un proceso que cuenta con toda sus páginas residentes en memoria.

Solución: EAT (effective access time) se calcula como: tiempo de busqueda en la TLB + (hit ratio $\times$ tiempo de acceso a memoria) + (1 − hit ratio) $\times$ (3 $\times$ tiempo de acceso a memoria). Cuando no hay caché hit se requieren 3 accesos a memoria porque se tienen 2 niveles de tablas de páginas.

Entonces el EAT de este sistema es: $12 + (0,8 \times 100) + (1 - 0,8) \times (3 \times 100) = 12 + 80 + 0,2 \times 300 = 92 + 60 = 152$ ns.

Justifique todos los resultados.

Problema 3 (20 pts)

Se desea implementar un sistema de archivos siguiendo un modelo de asignación en forma de lista que indexa (en la FAT) bloques agrupados de a dos. Los bloques de un mismo grupo se asignan de a dos, es decir siempre que un archivo/directorio requiera espacio se le asigna un grupo entero aunque necesite usar solo el primero de los bloques. El sistema soporta una estructura jerárquica de directorios

en forma de árbol. Suponiendo que el tamaño de cada bloque de disco es de 2 KiB, que se tiene una cantidad `CANT_BLOQUES` de bloques disponibles y que se cuenta con los siguientes tipos definidos:

```
type sector = array [0..2047] of byte;
type disk = array [0..(CANT_BLOQUES-1)] of sector;
type fat = array [0:(CANT_BLOQUES/2-1)] of integer;
type entrada_dir = Record
    usado: boolean          \\1 bits
    tipo: (file, dir)       \\1 bits
            esAnexo: boolean    \\1 bits, vale true si es el sector
                                 no indexado del grupo
            anexoUsado: boolean \\1 bits, vale true si es el sector
                                indexado y el anexo es usado
    inicioFAT: unsigned integer  \\2 bytes
    nombre: array [0..26] of char \\27 bytes
    tamanio: unsigned integer   \\ 2 bytes
    mascara: array [0..3] of bit    \\ 4 bits
end; // 32 bytes

disk D
fat F
```

Se pide:

(a) (2 pts) Determine cómo (en que estructura y de qué forma) se podría indicar el fin de un archivo y cómo se especificaría que un sector está vacío.

> **Solución:**
> Fin del archivo: entrada en la FAT con -1
> Sector vacío: entrada en la FAT con -2

(b) (3 pts) Determine cuál es el número máximo de archivos y el número máximo de directorios que soporta el sistema de archivos.

> **Solución:** sector/dir $= 2048/32 = 64$ dirs por sector
> En el caso de que sean vacíos solamente es necesario ser capaz de mantener sus referencias (i.e. tener una `dir_entry`) por lo que dado que en cada bloque se pueden guardar 64 entradas el total máximo de archivos/directorios es:
> cantidad máxima de archivos $= \text{CANT\_BLOQUES} * 64$
> cantidad máxima de directorios $= \text{CANT\_BLOQUES} * 64 + 1$
> donde el $+1$ es por el directorio raíz.
>
> Por otro lado si ocupan al menos un bloque será necesario darles por lo menos una entrada en la FAT (i.e. en la práctica tendrán dos bloques), este límite da $\text{CANT\_BLOQUES}/2$ archivos/directorios + el directorio raíz. Sin embargo también debe descontarse a la cantidad de bloques total los que son utilizados por el directorio raíz para guardar `dir_entry`. El resultado es entonces:
> cantidad máxima de archivos $= \text{CANT\_BLOQUES}/2 - \text{CANT\_BLOQUES}/(2 * 64 * 2)$
> cantidad máxima de directorios $= \text{CANT\_BLOQUES}/2 - \text{CANT\_BLOQUES}/(2 * 64 * 2) + 1$
> El segundo número surge de que para almacenar $\text{CANT\_BLOQUES}/2$ archivos/directorios se requieren $\text{CANT\_BLOQUES}/64$ bloques y por cada archivo que se quite se liberan 2.

(c) La función concat recibe la ruta completa de dos archivos y devuelve el sistema con la concatenación de ambos archivos.

`function concat(rutaA1 []: char, rutaA2 []: char): boolean`

I. (1 pt) ¿Qué problema de mal uso del espacio puede ocurrir al implementar la función concat?

II. (14 pts) Implemente la función concat.

Puede usar las siguientes funciones:

- `bool leerBloque(numBloque: int; var buffer[2048]: byte)`
  Lee el bloque apuntado por numBloque en el disco y lo carga en buffer y retorna si fue OK
- `bool escribirBloque(numBloque: int; buffer[2048]: byte)`
  Escribe los datos almacenados en buffer en disco en el bloque apuntado por numBloque y retorna si fue OK
- `int partirRuta(ruta[]: char; var camino[][0..26]: char)`
  Dada una ruta absoluta , devuelve las partes del camino , junto con la cantidad de partes:
  - Para "/home/sistoper/a.txt" se tiene que partes es ["home", "sistoper", "a.txt"] y el valor de retorno de la función es 3.
  - Para "/" se tiene que partes es [] y el valor de retorno de la función es 0.

Solución:

```
def concat(rutaA1 []: char, rutaA2 []: char):
    dir1 = buscarArchivo(rutaA1, entradasD1, iterEntradasD1,
    bloquePadreD1)
    dir2 = buscarArchivo(rutaA2, entradasD2, iterEntradasD2,
    bloquePadreD2)

    if (dir1 or dir2 == False):
        return False
    if (not dir1.usado or not dir2.usado or dir1.tipo != file
    or dir2.tipo != file):
        return False

    entradasD2[iterEntradasD2].usado = False
    entradasD2[iterEntradasD1].tamnio =
        entradasD1[iterEntradasD1].tamanio
        + entradasD2[iterEntradasD2].tamanio
    //concateno la FAT
    punteroFAT = entradasD1[iterEntradasD1].inicioFAT
    while (punteroFAT != -1):
        anterior = punteroFAT
        punteroFAT= fat[punteroFAT]
    punteroFATA2 = entradasD2[iterEntradasD2].inicioFAT
    fat[anterior] = punteroFATA2

    if (not escribirBloque(bloquePadreD1 ,entradasD1)) or
      (not escribirBloque(bloquePadreD2 ,entradasD2)):
        return False
    return True


def buscarArchivo(ruta, var entradas, iterEntradas, bloquePadre):
  cant = partirRuta(ruta,camino)

  iterCamino = 0
  // Comenzamos desde la raiz en el bloque de datos 0 */
  punteroFAT = 0
  usaAnexo = True // como no tengo entry dir para la raiz busco en
                  //los dos bloques
```

```
while (iterCamino < cant):
  encontrado = False
  while(punteroFAT != -1 and not encontrado):
    if (not leerBloque(punteroFAT, entradas)):
      return False
    iterEntradas = 0

    while(iterEntradas < 64 and not encontrado):
      if(entradas[iterEntradas].usado and
        entradas[iterEntradas].nombre == camino[iterCamino]):

        // Si es el ultimo del camino debe ser archivo
        if(iterCamino==cant-1 and
          entradas[iterEntradas].tipo==dir):
            return False

        // Si no es el ultimo del camino debe ser dir
        if(iterCamino<cant-1 and
          entradas[iterEntradas].tipo==file):
            return False

        encontrado = True
        bloquePadre = punteroFAT
      else:
        iterEntradas+=1

    if (not encontrado and usaAnexo):
        // Tengo que buscar en el segundo bloque asignado
        if (not leerBloque(punteroFAT + CANT_BLOQUES/2, entradas)):
          return False
        iterEntradas = 0

         while(iterEntradas < 64 and not encontrado):
            if(entradas[iterEntradas].usado and
              entradas[iterEntradas].nombre == camino[iterCamino]):

              // Si es el ultimo del camino debe ser archivo
              if(iterCamino==cant-1 and
                entradas[iterEntradas].tipo==dir):
                  return False

              // Si no es el ultimo del camino debe ser dir
              if(iterCamino<cant-1 and
                entradas[iterEntradas].tipo==file):
                  return False

              encontrado = True
              bloquePadre = punteroFAT
            else:
              iterEntradas+=1

    if (not encontrado):
      // Sigo buscando en el siguiente bloque del directorio
      punteroFAT = fat[punteroFAT]
    else:
```

```
        punteroFAT = entradas[iterEntradas].inicioFAT
        usaAnexo = entradas[iterEntradas].anexoUsado
        //si ocupa mas de una entrada en la FAT asumo
        //que en todas usa el anexo
    if (encontrado):
      iterCamino+=1
      return True
      //con las var entradas, iterEntradas, bloquePadre cargadas
    else:
      return False
```