Sistemas operativos

Examen Julio 2021

Bloque 1

Pregunta 1)

En un Sistema Operativo UNIX existe un proceso con UID=usr1 y GID=so. Este proceso desea acceder en modo escritura al archivo solucion_examen_Julio.txt donde usr2 y so son el UID / GID del dueño y el archvo tiene los permisos 557 a nivel del sistema de archivos. ¿Qué sucede cuando el proceso intenta acceder el archivo? Justifique su respuesta.

Pregunta 2)

Considere el siguiente código

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int value = 5;

int main()
{
	pid_t pid;

	pid = fork();

	if (pid == 0) { /* proceso hijo */
		value += 15;
		return 0;
	}
	else if (pid > 0) { /* proceso padre */
		wait(NULL); /* EL padre espera a que el hijo termine */
		printf ("PARENT: value = %d\n",value); /* LINEA A */
		return 0;
	}
}
```

¿Qué valor se imprime en la LINEA A? Justifique su respuesta.

Pregunta 3)

Considere un Sistema Operativo multiprogramado simétrico en un ambiente multiprocesador donde el diseñador utiliza un único semáforo para acceder al kernel.
Ej:

Init (Ker_mutex,1) /* al inicio del sistema */

Las system Calls de este Sistema Operativo son de la forma

```
void _SysCall_X ()
P(Ker_mutex)
.... Código de la SysCall X ....
V(Ker_mutex)
```

En cierto momento se detecta una baja de rendimiento en el sistema y se decide agregar más procesadores al sistema.

Esta acción, soluciona los problemas de performance? Justifique su respuesta.

Bloque 2

Un sistema de gestión de memoria, soporta espacios de direcciones lógicas (virtuales) de 36 bits y un sistema de tabla de páginas multinivel, con 2 niveles de tablas de páginas, y páginas de 4KiB

a) (9 puntos) Indique cómo se interpretan las direcciones lógicas para acceder al sistema de memoria virtual, sabiendo que las tablas de páginas de segundo nivel ocupan una página entera y cada entrada de las tablas ocupan 64 bits (explicite los campos de la dirección y el tamaño en bits de cada campo). Justifique.
b) (7 puntos) Considere que al sistema se incorpora un TLB de 128 entradas con política de reemplazo LRU, y considere que un proceso con un código de 8 KiB y que utiliza 16 KiB de **datos**. Suponga que el proceso lee toda su sección de **datos**, un byte a la vez, en forma secuencial.

   Indique la cantidad de **fallos de página** ocurridos y la cantidad de accesos a **memoria principal** a causa de la recorrida de los datos (ignore accesos generados por lectura de instrucciones).

c) (6 puntos) Considere el mismo escenario de la parte anterior, pero suponga que luego de la lectura de 7KiB de la sección de datos ocurre un cambio de contexto, el cual luego de cierto tiempo finaliza y nuestro proceso completa el acceso a los datos. Con la ocurrencia de este cambio de contexto, ¿Cambia la cantidad de accesos calculados de la parte anterior? Justifique.
d) (2 puntos) ¿Cambia la respuesta de la parte anterior si el cambio de contexto se realiza al finalizar la lectura de los 8KiB? Jusitifique

Solución:

a) Como se trata de un sistema de memoria multinivel de 2 niveles, la dirección se divide en 3 campos: - primer nivel - / - segundo nivel - / - desplazamiento -

   Las páginas son de 4KiB, por lo tanto se precisan 12 bits para direccionarla. Por esa razón, el campo desplazamiento es de 12 bits.
   Como las tablas de páginas ocupan 64 bits (8 bytes), y cada página ocupa 4KiB, las tablas de segundo nivel soportan: 4KiB / 8B = 2^{12} / 2^3 = 2^9 = 512 entradas. Por lo

tanto se precisan 9 bits para indicar cuál entrada de la tabla de páginas de segundo nivel se debe acceder.

Por último, como la dirección tiene 36 bits y se usan 12 + 9 para las otras secciones,restan 15 bits para indizar la tabla de páginas de primer nivel.

b) Al leer cada byte de la sección de datos, de no existir la TLB se acceden 3 veces a memoria principal (una vez a la tabla de primer nivel, una vez a la tabla de segundo nivel, y una vez al marco correspondiente). Como existe la TLB, cada HIT en la cache implica que se accede una única vez a memoria para leer el byte correspondiente.

   Los fallos de página se producen al leer el primer byte de cada página, los cuales son 4 ya que las lecturas están alineadas a al página (el código se ubica a partir de la dirección 0 de memoria virtual, y los datos inmediatamente después, a partir de la dirección 8K de memoria, por lo tanto los 16KB de memoria se ubican exactamente en 4 páginas).

   En total se producen 4 fallos de página, los cuales coinciden con *misses* en la TLB, lo que provocan 2 accesos adicionales a memoria principal cada uno (una vez por cada nivel de la tabla de páginas)

   La cantidad de accesos total es 16KiB + 8. Es decir, un acceso a memoria por cada byte y los 8 accesos adicionales por misses en la TLB.

c) Al provocarse un cambio de contexto, hay dos opciones en el sistema:
   1. Si la TLB no usa campo ASID, se debe realizar un *flush* de la TLB, anulando todas sus entradas.
   2. Si se usa el campo ASID, no se debe realizar ningún cambio.

   En el caso 1, el cambio de contexto va a generar 2 accesos extra a memoria para el proceso original, ya que al volver del mismo se obtendrá un miss adicional en la TLB y se deberá acceder a las tablas de páginas de primer y segundo nivel.
   En el caso 2, no se generan accesos extra porque la entrada del a TLB se mantiene, aunque no se puede descartar que el proceso nuevo utilice todas las entradas de la TLB, provocando que la del proceso original se reemplace.

d) En este caso no se generan accesos adicionales en ningún caso, ya que al momento de realizarse el cambio de contexto ya no se volverá a acceder a las entradas previamente cargadas de la TLB.

Bloque 3

Un sistema de archivos utiliza una estrategia indexada multinivel de dos niveles y un mapa de bits para administrar el espacio libre del disco.
A continuación se presentan las estructuras de datos utilizadas por este sistema de archivos.

```
const MAX_BLOQUES = 65536;
const MAX_INODOS = 8192;

type bloque = array [0..1023] of byte; // 1024
bytes type mapa_bits = array [0..MAX_BLOQUES-1] of bit;

type dir_entry = Record
     usado : boolean;                    // 1 bit
     nombre : array [0..59] of char;   // 60 bytes
     es_dir : boolean;                  // 1 bit
     inodo_num : int16;                 // 2 bytes
     permisos : array [0..13] of bit;   // 14 bits
End; // 64 bytes

type inodo = Record
     usado : boolean;                  // 1 bit
     inodo_num : int16;                // 2 bytes
     es_dir : boolean;                 // 1 bit
     tamaño : int32;                   // 4 bytes
     directo : array [0..6] of int16;   // 14 bytes
     directo_tope : int16;             // 2 bytes
     indirecto : array [0..2] of int16; // 6 bytes
     indirecto_tope : int16;           // 2 bytes
     reservado: array [0..13] of bit    // 14 bits
End; // 32 bytes

type inodos_tabla = array [0..MAX_INODOS-1] of inodo;
type disco = array [0..MAX_BLOQUES-1] of bloque;
var
     IT : inodos_tabla;
     MB : mapa_bits;
     D : disco;
```

A su vez, se sabe que el directorio raíz es el inodo número 0, que la tabla de inodos, el bitmap, y el disco son globales.
Por otra parte se dispone de los siguientes procedimientos:

1. `Procedure leerBlq(blq_num: int, Var buff: bloque) : boolean`
   Lee de disco el bloque `blq_num`, pasado como parámetro, y carga el contenido leído en el parámetro de salida `buff`. Retorna **true** en caso de que la operación se ejecute con éxito y **false** en caso contrario.

2. `Procedure escrBlq(blq_num : int, buff : bloque) : boolean`
   Escribe en el bloque `blq_num`, pasado como parámetro, la información que se encuentra en el parámetro buff. Retorna true en caso de que la operación se ejecute con éxito y false en caso contrario.

3. `Procedure parteCamino(camino: array of char, var base: array of char, var resto: array of char);`
   Retorna la primera parte de la ruta en el parámetro base, y las restantes partes en resto.
   Ejemplo:
   ```
   parteCamino('/users/so/a.txt', 'users', '/so/a.txt');
   parteCamino('/a.txt', 'a.txt', '');
   parteCamino('/', '', '');
   ```

4. `Procedure buscarDirEntryEnBloque(buff: bloque, nombre: array of char): dir_entry;`
   Interpreta el parámetro bloque como un arreglo de dir_entry y devuelve una cuyo **nombre** coincida con el parámetro indicado. Devuelve NULL en caso de que no se encuentre

1. Cuantos archivos y directorios pueden ser almacenados como máximo dentro de un inodo de tipo Directorio
2. Implementar una función que retorne el `dir_entry` asociado al archivo o directorio definido por "**cam"**

   ```
   dir_entry buscarDirEntry(cam: array of char);
   ```

   Retorna el `dir_entry` asociado al elemento definido por `cam`, en cualquier otro caso el retorno debe ser `null`.

Solución:

1) Un inodo tiene tiene acceso a 7 bloques directos y 3 bloques indirectos. Como el inodo es de tipo directorio, entonces todos los bloques a los que apunta se interpretan en dir_entries, los cuales pueden ser tanto archivos como directorios.

Como el tamaño de cada bloque es 1024, y el tamaño de la estructura del dir_entry es de 64 bytes, tenemos que en cada bloque entran 1024/64 = 16.

Dado que en cada bloque tenemos 1024/4 (4 bytes es el tamaño de un int16) podemos direccionar 256 bloques por cada indexación indirecta.

Dado esto, tenemos que un inodo puede indexar: 7 + 3*256 = **775** bloques, por lo que la cantidad de archivos y directorios que se pueden almacenar en un directorio es de 775 * 16 = **12400**

Dado que la máxima cantidad de inodos que puede soportar el sistema es 8192, y cada dir_entry que pertenece a un directorio, necesita un Inodo (ya que no son datos por ser directorio), el total máximo es de **8191**, ya que el directorio al cual el directorio pertenece ocupa al menos un Inodo.

2)

```
var CANT_DIR_ENTRIES_PER_BLOCK = TAM_BLOQUE/TAM_DIR_ENTRY;
var CANT_INTEGER_PER_BLOCK = TAM_BLOQUE/sizeof(int16);
var DIRECTORIO_RAIZ = "/";

dir_entry buscarDirEntry(char* cam) {
   char* base;
   dir_entry dirEntry = null;
   int i, j, inodo_iter;
   dir_entry[CANT_DIR_ENTRIES_PER_BLOCK] buff;
   int[CANT_INTEGER_PER_BLOCK] indexDirEntries;

   if (cam == ”“) {
       return null;
   } else if (cam == DIRECTORIO_RAIZ) {
       dirEntry.tipo = dir;
       dirEntry.usado = true;
       dirEntry.inodo_num = 0;
       return dirEntry;
   }

   inodo_iter = 0;
   parteCamino(cam, base, cam);
   while (base != "") {
       if (cam != "" &&
       (!IT[inodo_iter].usado || !IT[inodo_iter].es_dir || !MB[inodo_iter])) {
           return null;
       }
       i=0;
       encontre = false;
       while (!encontre && i < IT[inodo_iter].directo_tope) {
           if (!leerBlq(IT[inodo_iter].directo[i], buff)) {
               return null;
           }
           dirEntry = buscarDirEntryEnBloque(buff, base);
           encontre = dirEntry != null;
           if (encontre) {
               inodo_iter = dirEntry.inodo_num;
           }
           i++;
       }

       if (!encontre) {
           j = 0;
            while (!encontre && j < IT[inodo_iter].indirecto_tope) {
               if (!leerBlq(IT[inodo_iter].indirecto[j], indexDirEntries)) {
                   return null;
               }
               i = 0;
               while (!encontre && i < CANT_INTEGER_PER_BLOCK) {
                   if (!leerBlq(indexDirEntries[i], buff)) {
                       return null;
                   }
                   dirEntry = buscarDirEntryEnBloque(buff, base);
                   encontre = dirEntry != null;
                   if (encontre) {
                       inodo_iter = dirEntry.inodo_num;
                   }
                   i++;
```

```
                }
                j++;
             }
        if (!encontre) {
            return null;
        }
        parteCamino(cam, base, cam);
    }
    return dirEntry;
}
```

Bloque 4

Se desea modelar un servicio de comida rápida donde hay 5 cocineros que preparan hamburguesas. Los ingredientes de cada hamburguesa son carne, pan y 3 adicionales seleccionados por el cliente de 6 posibles. Cada uno de estos 6 ingredientes adicionales se encuentra en un recipiente que solo puede ser accedido por un cocinero por vez y el cocinero precisa acceder a los 3 simultáneamente para preparar la hamburguesa.

Se dispone de los siguientes procedimientos auxiliares:

- que_ingredientes(): array[1..3] of 1..6
  Ejecutado por el cocinero para obtener los ingredientes adicionales que el cliente le quiere poner a la hamburguesa. Siempre retorna 3 ingredientes distintos.
- elaborar_hamburguesa()
  Ejecutado por el cocinero

Se pide:

a. Implementar usando ADA a los cocineros y los recipientes de ingredientes considerando que los cocineros deben obtener los ingredientes en estricto orden de llegada; es decir que un cocinero que pide los ingredientes cuando hay otros esperando antes no puede seguir aunque sus 3 ingredientes estén libres.
   Se pueden usar tareas auxiliares.

b. Implementar usando ADA a los cocineros y los recipientes de ingredientes considerando que los cocineros no deben esperar si tienen todos sus ingredientes libres. Un ingrediente se considera ocupado una vez que lo reservó un cocinero aunque no lo esté usando en ese momento.

   No se pueden usar tareas auxiliares.

Solución:

a)

```
procedure solucion_examen is
      task type cocinero is
      end cocinero;

      task body cocinero is
      begin
            loop
                  ingredientes = que_ingredientes();
                  manager.lock();
                  for (var i = 0; i < 3; i ++) do
                        var ingrediente = ingredientes[i];
                        recipientes[ingrediente].lock();
                  end for;
                  manager.unlock();
                  preparar_hamburguesa();
                  for (var i = 0; i < 3; i ++) do
                        var ingrediente = ingredientes[i];
                        recipientes[ingrediente].unlock();
                  end for;
            end while;
      end cocinero;

      task type mutex is
            entry lock();
            entry unlock();
      end mutex;

      task body mutex is
      begin
            loop
                  accept lock;
                  accept unlock;
            end;
      end mutex;

      cocineros: array(5) of cocinero;
      recipientes: array(6) of mutex;
      manager: mutex;
begin
end solucion_examen;
```

b)

```
procedure solucion_examen is
      task type cocinero is
            function sort_array(array) begin
                  // selection sort
                  for (var i:int = 0; i < array.length; i++){
                        var min:int = i;
                        for (var j = i + 1; j < array.length; j++){
                              if ( array[j] < array[min]){
                                    min = j;
                              }
                        }
                        var aux = array[i];
                        array[i] = array[min];
                        array[min] = aux;
                  }
            end function;
      end cocinero;
      task body cocinero is
      begin
            loop
                  ingredientes = que_ingredientes();
                  // se ordena para prevenir deadlock
                  ingredientes = sort_array(ingredientes);
                  for (var i = 0; i < 3; i ++) do
                        var ingrediente = ingredientes[i];
                        recipientes[ingrediente].obtener_ingrediente();
                  end for;
                  preparar_hamburguesa();
                  for (var i = 0; i < 3; i ++) do
                        var ingrediente = ingredientes[i];
                        recipientes[ingrediente].fin_uso_recipiente();
                  end for;
            end;
      end cocinero;

      task type recipiente is
            entry obtener_ingrediente;
            entry fin_uso_recipiente;
      end recipiente;
      task body recipiente is
            loop
                  accept obtener_ingrediente;
                  accept fin_uso_recipiente();
            end;
      end recipiente;

      cocineros: array(5) of cocinero;
      recipientes: array(6) of recipiente;

begin
end solucion_examen;
```