# Sistemas operativos

Super Parcial 2021

## Bloque 1 (15 puntos)

**Pregunta File System)**
SimpleFS 1.0 es un sistema de archivos que utiliza indexación para el manejo de la información. Las principales estructuras de SimpleFS 1.0 son:

```
type entrada_dir = Record
  usado : boolean;                    // 1 bit
  nombre : array [0..123] of char;    // 124 bytes
  es_dir : boolean;                   // 1 bit
  inodo_num : int;                    // 2 bytes
  permisos : array [0..13] of bit;    // 14 bits
End;

type inodo = Record
  usado : boolean;                    // 1 bit
  inodo_num : int;                    // 2 bytes
  es_dir : boolean;                   // 1 bit
  tamaño : long;                  // 4 bytes
  directo : array [0..11] of int;     // 24 bytes
  directo_tope : int;                 // 2 bytes
  reservado : array [0..13] of bit;   // 14 bits
End;

type bloque = array [0..4095] of byte;
type tabla_inodos = array [0..32767] of inodo;
```

Se sabe que el índice `0` de `tabla_inodos` está asignado al directorio raíz.

Parte A)

Modifique la estructura de SimpleFS 1.0 y cree SimpleFS 1.1a agregando soporte para un bloque de indirección simple. ¿Cuál es el tamaño máximo que puede tener un archivo en SimpleFS 1.1a?

Parte B)

Modificando nuevamente la estructura de SimpleFS 1.0 cree SimpleFS 1.1b agregando soporte para tener hasta 65535 hard links por archivo. Suponga que en SimpleFS 1.1b se crea un archivo de texto de 4 KiB (4*2^10 bytes) y luego se crea un hard link a este archivo. ¿Cuántos bytes son requeridos por estas operaciones teniendo en cuenta todas las estructuras presentadas y los datos del archivo?

**Solución parte A)**

- El tamaño máximo contabilizable por el campo tamaño es $2^{32}$-1 =~ 4 GiB.
- Con bloques directos es posible asignar hasta $2^{12}$ * 12 = 4096 * 12 = 49152 bytes.
- Un bloque de indirección simple podría direccionar hasta $2^{12}$ / 2 = $2^{11}$ bloques.

En total entonces un archivo podría tener asignados hasta $2^{12}$ * (12 + $2^{11}$) = 4096*12+$2^{23}$= 49152 + 8388608 (8 MiB) = 8437760 bytes.

**Solución parte B)**

Los elementos ocupados es: 1 inodo + 2 entrada_dir + 4 KiB.

El inodo modificado necesitará un nuevo campo de tipo int para contar los hard links por lo que ocupará 36 bytes en lugar de 34 bytes. Entonces los bytes necesarios son: 36 bytes + 2*128 bytes + 4096 bytes = 4388 bytes.

**Preguntas Múltiple Opción**

**Pregunta 2)**

Un dispositivo **A** genera datos a una velocidad Va y los escribe en un área de memoria intermedia para que los consuma un dispositvo **B** a una velocidad Vb en el mismo orden que fueron generados, con Va > Vb. ¿Qué técnica se está describiendo?

a) Busy Waiting
b) Polling
c) Spooling
d) Caching

**Pregunta 3)**

Se necesita crear una estructura de discos redundantes y se tienen los siguientes datos de la realidad

- Cada disco de 1Tb cuesta $100
- Disponemos en nuestro presupuesto de $ 950
- El aplicativo que va a guardar datos allí en promedio tiene un 70% de escrituras y un 30% de lecturas

Qué tipo de estructura de disco recomienda para organizar los discos, respetando las condiciones, optimizando el uso del aplicativo y proveyendo redunancia para la pérdida de al menos 1 disco.

a) Un raid 0 de 9 discos.
b) Un raid 1+0 de 10 discos.
c) Un raid 0+1 de 8 discos.
d) Un raid 1+0 de 8 discos.

**Pregunta 4)**

En un Sistema Operativo UNIX existe un proceso con **UID=usr1** y **GID=so**. Este proceso desea acceder en modo lectura al archivo solucion_parcial.txt donde **usr2** y **so** son el UID / GID del dueño y tiene los permisos **-wx-wxrwx** a nivel del sistema de archivos.

¿Qué sucede cuando el proceso intenta acceder el archivo?

a) El acceso es denegado al evaluar los permisos del dueño.
b) El acceso es permitido.
c) Se envía una solicitud de acceso al dueño del archivo.
d) El acceso es denegado al evaluar los permisos del grupo.

**Respuestas preguntas múltiple opción (en nota al pie de página)**[1]

# Bloque 2 (20 puntos)

## Parte 1)

Se tiene un sistema operativo simétrico, multiprogramado, con un planificador con dos colas de prioridad (alta/baja) con retroalimentación (es decir, una multi-level feedback queue con dos niveles de prioridad). En la cola de alta prioridad se utiliza un algoritmo round-robin con quantum de 2 unidades de tiempo mientras que en la otra un SJF **expropiativo**. Cualquier caso de empate se define con una prioridad dada por el índice del proceso, donde a **menor** índice, **mayor** prioridad (i.e. P1>P2>…>Pn).

A los procesos al ser creados se les asignan **prioridad alta**, que luego podrá ser modificada según los siguientes criterios:

- Un proceso **baja** a la cola de prioridad baja si en las últimas 2 unidades de tiempo a utilizado completamente el recurso procesador.
- Un proceso **sube** a la cola de prioridad alta si en las últimas 2 unidades de tiempo no ha usado el recurso procesador en absoluto.

**En el instante t=0 se lanza la ejecución de tres procesos (P1, P2 y P3) donde P1 y P3 ejecutan el código de R1 mientras que P2 el de R2**

| R1 | R2 |
|---|---|
| Ejecuta 1 | Ejecuta 1 |
| Bloquea 3 | Bloquea 3 |
| Ejecuta 5 | Ejecuta4 |

1 Solución preguntas MO: Pregunta 2 – C, Pregunta 3 – D, Pregunta 4 - D

Asumiendo un sistema con dos procesadores, complete el diagrama de planificación (tiempo vs. Procesos) arrastrando los estados correspondientes de cada proceso (listo/ejecutando/bloqueado/terminado) en cada intervalo de tiempo **hasta el tiempo 12**. Además, se debe indicar, en cada caso, el nivel de prioridad (alta/baja) en el que se encuentra el proceso.

LA: Listo en la cola Alta
LB: Listo en la cola Baja
EA: Ejecutando prioridad Alta
EB: Ejecutando prioridad Baja
BA: Bloqueado prioridad Alta
BB: Bloqueado prioridad Baja
T: Terminado
-: El proceso aún no se encuentra en el sistema

## Solución:

| Proceso | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P1 | EA | BA | BA | BA | EA | EA | LB | LB | EA | EA | EB | T |
| P2 | EA | BA | BA | BA | EA | EA | EB | EB | T | T | T | T |
| P3 | LA | EA | BA | BA | BA | LA | EA | EA | EB | EB | EB | T |

## Parte 2)

Considerando un sistema operativo monoprocesador y los procesos P1 y P2, iniciados en t=0.

- P1 cuenta con un solo hilo, mientras que P2 tiene dos.
- El sistema utiliza el modelo Mx1 para la planificación.
- Dada la siguiente planificación a nivel de usuario:

| Usuario | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| P1H1 | E | E | B | B | E | E | E | E | T | T | T | T |
| P2H1 | L1 | L1 | E | L1 | L1 | L1 | L1 | L1 | E | B | E | T |
| P2H2 | L2 | L2 | L1 | E | L2 | L2 | L2 | L2 | L1 | E | T | T |

**Indique:**

1) Tiempo de espera del proceso 2
2) % de Utilización de CPU
3) Indique el tiempo cuando la solución no respeta el modelo Mx1. Si la respeta, coloque 0.

## Solución)

1) 6
2) 100%
3) 10

## Bloque 3 (20 puntos)

Un sistema de gestión de memoria, soporta espacios de direcciones lógicas (virtuales) de 32 bits y un modelo de memoria paginada tradicional (tabla de página de un nivel), con tamaños de página de 4KiB.

1. ¿Cuál es el tamaño total en bytes del espacio de direccionamiento virtual?
2. ¿Cuántos bits serían necesarios para la sección desplazamiento dentro de la dirección virtual?
3. Indique la cantidad de entradas de la tabla de páginas.
4. Ahora considere un esquema de tablas de página multinivel, con tablas de dos niveles, suponiendo que ambas tablas tienen la misma cantidad de entradas. ¿Cuántos bits serían necesarios para direccionar en cada tabla?
5. Manteniendo el esquema multinivel de la parte anterior: ¿Cuántas tablas de páginas en total serían necesarias para direccionar 1GiB de código y datos, además de 40MiB de pila?

## Solución:

1. Como las direcciones virtuales tienen 32 bits, el espacio de direccionamiento virtual es 4 GiB ($2^{32}$ bytes)
2. Los bits de desplazamiento deben permitir direccionar una página completa. Como la página es de 4KiB, se precisan 12 bits para direccionarla.
3. Al utilizar una tabla de páginas de un único nivel, la dirección virtual se interpreta como *índice tabla de páginas / desplazamiento* . Como el desplazamiento es de 12 bits, y la dirección virtual es de 32 bits, quedan 20 bits para indizar la tabla de páginas, por lo tanto, la tabla de páginas tiene $2^{20}$ entradas.
4. En este caso la dirección se divide como

   *índice tabla 1er nivel / índice tabla 2ndo nivel / desplazamiento*

   Razonando de igual forma que en el parte anterior, y como ambas tablas tienen la misma cantidad de entradas, se deduce que se usan 10 bits para indizar cada tabla.
5. Se parte de conocer que la sección de código y datos se ubican a partir de la dirección virtual 0, mientras que la pila a partir de la última dirección disponible (0xFFFFFFFF) y crece hacia las direcciones bajas.

   Cada página direcciona 4KB, por lo tanto se precisan 10 * 1024 páginas para guardar el stack. Cada tabla de páginas de último nivel direcciona $2^{10}$ páginas, por lo cual para el stack se precisan 10 tablas de segundo nivel.

   Razonando de igual modo: 1GiB / 4KiB = $2^{30}$ / $2^{12}$ = $2^{18}$ páginas para guardar el código y

datos. Como cada tabla de segundo nivel direcciona $2^{10}$ páginas, se precisan $2^8$ tablas de segundo nivel.

Finalmente, se precisa una tabla de primer nivel (ya que es única).
En total se precisan 1 + 10 + $2^8$ tablas = 267 tablas.

## Bloque 4 (30 puntos)

Se tiene un sistema de peajes con atención manual que cuenta con 6 casillas. Los vehículos, cuando llegan, se ponen en la casilla con la cola más corta para que se le cobre el peaje. Cada cierto tiempo, el supervisor recorre las casillas para levantar el dinero recaudado. Para realizar esta tarea, las casillas deberán terminar de cobrar a los vehículos que están atendiendo y no empezar a cobrar al siguiente de la cola hasta entregar la recaudación.

Se dispone de los siguientes procedimientos auxiliares:

- cobrar_cliente: ejecutada por la casilla para cobrar el peaje
- pasar: ejecutada por el vehículo para cruzar el peaje
- obtener_recaudación: ejecutada por el supervisor para levantar el dinero recaudado
- otras_tareas: ejecutada por el supervisor cuando no está levantando el dinero en las casillas

**Se pide**:

Implemente el monitor peajes para solucionar la realidad anterior (según el código incluido).

```
procedure casilla(id: int){
     while (true){
        peajes.casilla_libre(id);
        cobrar_cliente();
        peajes.fin_cobrar_casilla(id);
      }
}

procedure vehiculo(){
    int casilla = peajes.llega_vehiculo();
    pasar();
    peajes.paso_vehiculo(casilla);
}

procedure supervisor(){
    while (true){
        peajes.llega_supervisor();
        obtener_recaudacion();
        peajes.termina_supervisor();
        otras_tareas()
    }
}
begin
cobegin
```

```
    casilla(0);
    casilla(1);
    casilla(2);
    casilla(3);
    casilla(4);
    casilla(5);
    supervisor();
    vehiculo();
    ...
    vehiculo();

coend
end;
```

## Solución:

```
monitor peajes{
      boolean haySupervisor;
      condition supervisorEnEspera;
      condition esperandoFinSupervisor;
      int cantidadEsperandoSupevisor;

      int casillasTrabajando;
      int autosEnCasilla[6];
      condition filaCasilla[6];
      condition casillaEsperandoAuto[6];
      condition casillaEsperandoPaseAuto[6];

      // auxiliar
      procedure obtenerAutosEnCasillas(){
            int casillaMasCorta = 0;
            int cantMasCorta = autosEnCasilla[0];
            for (int i = 1; i < 6; i++){
                  if (autosEnCasilla[i] < cantMasCorta){
                        cantMasCorta = autosEnCasilla[i];
                        casillaMasCorta = i;
                  }
            }
            return casillaMasCorta, cantMasCorta;
      }

      procedure llegaSupervisor(){
            haySupervisor = true;
            if (casillasTrabajando > 0){ // hay algún auto siendo cobrado
                  supervisorEnEspera.wait();
            }
      }
```

```
procedure terminaSupervisor(){
      haySupervisor = false;

      // despertar todos aquellos esperando por prioridad supervisor
      foreach (cantidadEsperandoSupervisor)
            esperandoFinSupervisor.signal();

      cantidadEsperandoSupervisor = 0;
}

procedure llegaVehiculo(){
      if (haySupervisor){
            cantidadEsperandoSupevisor ++;
            esperandoFinSupervisor.wait();
      }

      int casillaMasCorta, cantMasCorta = obtenerAutosEnCasillas();

      autosEnCasilla[casillaMasCorta]++;
      if (cantMasCorta == 0){
            // hay casilla vacia (esperando auto),
            // despertar casilla correspondiente
            casillaEsperandoAuto[casillaMasCorta].signal();
      }else{
            filaCasilla[casillaMasCorta].wait();
      }

      vehiculoEsperandoCobro[casillaMasCorta].wait();

      return casillaMasCorta;
}

procedure pasoVehiculo(nroCasilla){
      autosEnCasilla[nroCasilla]--;
      // aviso a casilla que terminé de pasar
      casillaEsperandoPaseAuto[nroCasilla].signal();
}

procedure casillaLibre(nroCasilla){
      // codigo repetido de auto
      if (haySupervisor){
            cantidadEsperandoSupevisor ++;
            esperandoFinSupervisor.wait();
      }

      if (autosEnCasilla[nroCasilla] > 0){
            filaCasilla[nroCasilla].signal();
      }else{
            casillaEsperandoAuto[nroCasilla].wait();
      }

      casillasTrabajando++;
}
```

```
	procedure finCobrarCasilla(nroCasilla){
		// despertar auto que espraba cobro
		// y dormir en espera de que el auto pase
		vehiculoEsperandoCobro[nroCasilla].signal();
		casillaEsperandoPaseAuto[nroCasilla].wait();

		casillasTrabajando--;

		// verificar que no haya supervisor esperando
		if (haySupervisor && (casillasTrabajando == 0)){
			supervisorEnEspera.signal();
		}
	}

	begin
		for (int i = 0; i < 6; i++) {
			autosEnCasilla[i] = 0;
		}
		cantidadEsperandoSupevisor = 0;
		casillasTrabajando = 0;
		haySupervisor = false;
	end
}
```