Departamento de Arquitectura

Segundo parcial de Sistemas Operativos

12 de Julio de 2024

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

- **Indique su nombre completo, número de cédula y número de parcial en cada hoja** (no se corregirán las hojas sin datos). 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**.
- Sólo se contestarán dudas de letra y no se aceptarán dudas en los últimos 30 minutos del parcial.
- El parcial es **SIN** material y dura **3 horas**. Al momento de finalizar el parcial **no se podrá escribir absolutamente nada en las hojas**. El estudiante debe ponerse de pie e ir a la fila de entrega.

Importante: Debe justificar todas las respuestas, siempre.

Problema 1 (14 puntos)

(a) Considere un subsistema de entrada/salida

I. [2 puntos] Explique los conceptos de operaciones no bloqueantes y operaciones asincrónicas.

> **Solución:** En el contexto de un subsistema de entrada/salida, una operación no bloqueante es aquella en la cual la invocación retorna tan pronto como sea posible. Una operación asincrónica es la que se ejecuta en paralelo, el control retorna inmediatamente al proceso invocante y el sistema notifica cuando la operación finaliza.

II. [4 puntos] Un proceso debe realizar dos tareas: i) acceder a 1 Mib de información almacenada en un archivo y procesarla, y ii) procesar datos ya disponibles en memoria, una tarea que demanda entre 10 y 15 ms. La velocidad de transferencia del disco es 40 Mib/s y la suma del tiempo de posicionamiento más la latencia varía entre 4 y 6 ms. Presente un seudocódigo que indique cómo ejecutar de la manera más eficiente posible el proceso. Justifique su respuesta

> **Solución:** La manera mas eficiente de realizar las tareas es mediante una invocación asincrónica a la operación de entrada/salida. La operación de acceso a los datos demandará entre 29 y 31 ms (entre 4 y 6 ms para posicionamiento y latencia y 1/40 s = 25 ms para transferencia), por lo tanto, el procesamiento de los datos en memoria demorará menos que el acceso a los datos del archivo. El proceso deberá esperar la finalización de la operación de entrada/salida. El seudocódigo es:

```
1  begin
2      datos = leer_archivo_async();
3      procesar_datos_en_memoria();
4      leer_archivo_wait();
5      procesar(datos);
6      // otras instrucciones;
7  end
```

(b) [3 puntos] Explique qué es el sector de boot, su cometido y su funcionamiento.

> **Solución:** El sector de boot es el primer sector en la partición de boot. Su cometido es almacenar un código de arranque, con las instrucciones para cargar el kernel del sistema operativo. El proceso de boot inicia ejecutando un código residente en ROM, que lee el Registro Maestro

de Boot, lo carga en memoria, examina su tabla de particiones y determina el primer sector de la partición de boot, que es el *sector de boot*. Luego se cargan otros susbsistemas y servicios del kernel.

(c) [2 puntos] Explique las estrategias RAID 0+1 y RAID 1+0.

Solución: RAID 0+1 aplica primero división (striping) y luego espejado (es un espejo de divisiones). RAID 1+0 aplica primero espejado y luego división (es una división de espejos).

(d) Sobre los hipervisores de tipo 2

I. [2 puntos] Explique su funcionamiento

Solución: Un hipervisor de tipo 2 ejecuta en modo usuario como un proceso más del sistema operativo anfitrión. Aplica el procedimiento de traducción binaria: i) el hipervisor analiza el flujo de ejecución (bloques de código) y traduce las instrucciones sensibles por emulaciones (llamadas a procedimientos del hipervisor, no sensibles) y ii) los bloques traducidos son ejecutados por la CPU directamente. Mediante este procedimiento, permite la virtualización en arquitecturas que no cumplen con las hipótesis de Popek & Goldberg.

II. [1 punto] Indique las características de un bloque de código ejecutado por un hipervisor de tipo 2.

Solución: Cada bloque tiene un único punto de entrada. No tiene instrucciones que cambien el flujo de ejecución (ni saltos, ni invocaciones, ni interrupciones), que se sustituyeron por invocaciones a procedimientos del hipervisor. El bloque termina en un salto, invocación o interrupción.

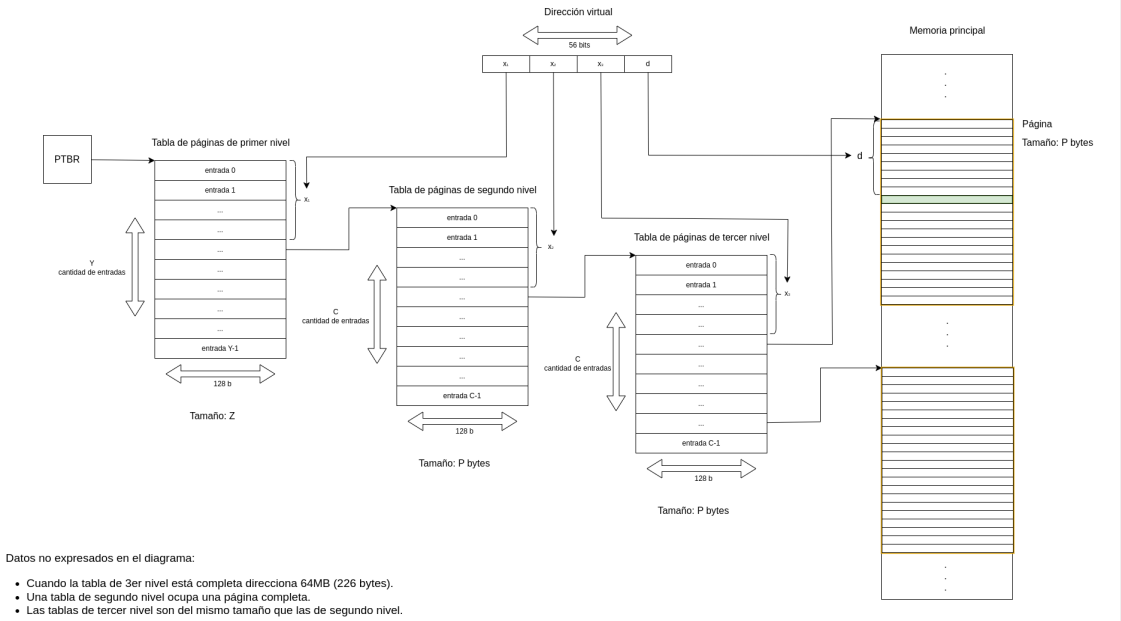
Problema 2 (20 puntos)

Considere un sistema operativo que usa un sistema de memoria virtual paginada multi nivel con un modelo de paginación bajo demanda. El sistema tiene las siguientes características:

- Usa direcciones virtuales de 56 bits y la traducción utiliza tres niveles de tablas de páginas.
- Las entradas en las tablas de páginas son de 128 bits (16 bytes).
- Cuando la tabla de 3er nivel está completa direcciona 64 MiB (2^{26} bytes).
- Una tabla de segundo nivel ocupa una página completa.
- Las tablas de tercer nivel son del mismo tamaño que las de segundo nivel.

(a) [5 puntos] Determinar el tamaño de los marcos del sistema.

Solución: En el siguiente diagrama se ilustran todos los datos conocidos en esta realidad y se parametrizan los que son necesarios descubrir:



Por definición, el tamaño de los marcos es igual al tamaño de las páginas. Veremos entonces el tamaño de una página (P).

Por letra, se conoce que el tamaño de una tabla de tercer nivel coincide con el de una página.

Como el tamaño las entradas son de 16 bytes (2^4), se tiene que: $\mathrm{P}/16$ bytes = Cantidad de entradas de 3er nivel (C).

En este sentido, se puede formular que P bytes = 2^4 bytes $\times$ C, es decir, que el tamaño de la tabla de páginas de tercer nivel es igual al tamaño de cada entrada por su cantidad de entradas.

Por otra parte se sabe que cada una de las entradas de 3er nivel (C) apuntan a un marco de tamaño P. Además, por letra se sabe que una tabla completa de tercer nivel direcciona 2^{26} bytes. Por lo tanto, se tiene que 2^{26} bytes = $\mathrm{C} \times \mathrm{P}$ bytes. La primera parte de la igualdad corresponde a la cantidad de memoria direccionable por cada tabla (dato de letra), que es igual a lo que puede direccionar cada entrada, es decir, cada entrada se mapea a una página de tamaño P.

Finalmente, tenemos las siguientes relaciones:

- P bytes = 2^4 bytes $\times$ C
- 2^{26} bytes = C $\times$ P bytes

Sustitiyendo, se obtiene que P bytes = 2^4 bytes $\times$ (2^{26} bytes / P bytes) y operando se llega a que $\mathrm{P} = 2^{15}$.

Tamaño de un marco: $2^{15}\mathrm{B} = 2^5\mathrm{KiB} = 32\mathrm{KiB}$.

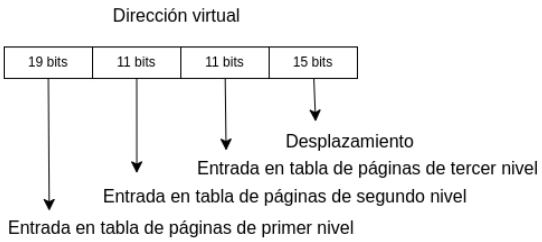
(b) [2 puntos] Especificar para qué se utiliza cada bit de una dirección virtual.

Solución:

Como el tamaño de una página es de 2^{15} bytes, se reservan los 15 bits menos significativos en la dirección virtual para realizar el desplazamiento dentro de la página.

Para calcular la cantidad de entradas de una tabla de tercer nivel se realiza la siguiente ope-

ración: 2^{15} bytes / 2^4 bytes lo que determina que se necesitan 11 bits para referenciarlas.
Por letra, la tabla de páginas de segundo nivel es igual a la de tercer nivel, por lo que de igual forma se requieren 11 bits en la dirección para direccionar sus entradas.
Por último, los bits restantes (más significativos) se ocupan para direccionar las entradas de la tabla de primer nivel. Como la dirección virtual es de 56 bits, corresponden a los primeros 19 bits.



(c) Sea un proceso que requiere 500MiB para almacenar su código, datos globales, datos dinámicos, y 80MiB para almacenar su pila (stack).

I. [4 puntos] Determine cuántas tablas de segundo y tercer nivel son necesarias para poder representar la memoria usada por el proceso.

Solución:
Cada tabla de tercer nivel es capaz de direccionar como máximo 2^{26} bytes, que corresponden a $64\mathrm{MiB}$. Para calcular cuántas tablas de páginas de tercer nivel son requeridas para el código, datos globales y datos dinámicos, es necesario realizar el cociente entre el tamaño de espacio de memoria requerido y el tamaño total de direccionamiento por tabla:
tablas de tercer nivel $= 500\mathrm{MiB}/64\mathrm{MiB}$. Es decir, se precisan 8 tablas de tercer nivel (7 completas y una parcialmente utilizada) para alojar el código, datos globales y dinámicos.
Para calcular las tablas de segundo nivel, recordar que cada entrada en una tabla se segundo nivel apunta a una tabla de tercer nivel. En este sentido, se necesitan 8 entradas de segundo nivel, es decir, una tabla (cada tabla tiene 2^{11} entradas).
Para calcular la cantidad de tablas de tercer nivel para la pila ($80\mathrm{MiB}$), se realiza el mismo razonamiento anterior, resultando en 2 tablas de tercer nivel (una completa y una parcialmente utilizada). Análogamente, se necesita una tabla de segundo nivel con dos entradas ocupadas para alojar el espacio de la pila.
Cantidad de tablas de segundo nivel: 2.
Cantidad de tablas de tercer nivel: 10.

II. [1 punto] Determine cuáles entradas de la tabla de primer nivel son utilizadas.

Solución:
Se utilizan la primera y la última entrada. Por cada entrada de primer nivel se pueden direccionar 128 GiB ($2^{11} \times 2^{11} \times 2^{15}$ bytes), lo que determina con una sola de ellas es suficiente para cada segmento de memoria.
Siguiendo el razonamiento de la parte anterior, se necesita una tabla de primer nivel con dos entradas ocupadas (que apuntan a las dos tablas de segundo nivel necesarias). Además, debido al mapeo de las distintas porciones de memoria a utilizar, queda determinada la entrada 0 y la última (2^{19}–1).

III. [4 puntos] Indique el tamaño de las estructuras necesarias para el correcto funcionamiento del proceso. Compare con el tamaño si el sistema utilizara solo un nivel de tablas de páginas,

manteniendo el mismo tamaño de páginas.

Solución:
Se necesita cargar en memoria la tabla de primer nivel, que posee 2^{19} entradas de 16 bytes. Esto es, $2^{23}\mathrm{B} = 8\mathrm{MiB}$.
A lo anterior se suman 2 tablas de segundo y 10 tablas de tercer nivel (calculado anteriormente en este ejercicio). Como estas tablas ocupan una página y su tamaño es de $2^{15}\mathrm{B} = 32\mathrm{KiB}$ bytes, tenemos para ellas se necesita $12 \times 32\mathrm{KiB}$.
Si el sistema operativo carga en memoria únicamente aquellas tablas que necesita el proceso, en total se requieren: $8\mathrm{MiB} + 32\mathrm{KiB} \times 12 = 8224\mathrm{KiB} \approx 8\mathrm{MiB}$.
Finalmente, si solamente se tuviese un nivel en la tabla de páginas esta precisaría $2^{56}/2^{15}$ entradas, cada una de 2^4 bytes. Es decir $2^{45}\mathrm{B} = 32\mathrm{TiB}$ para almacenar la tabla.

iv. [2 puntos] Calcule cuántos accesos a memoria principal son necesarios para leer de a 1 byte, 32 bytes de la memoria principal en el sistema planteado.

Solución:
Para acceder a un byte, se necesita un acceso correspondiente a cada tabla de páginas (primer, segundo y tercer nivel). Luego se necesita un acceso adicional para obtener el dato aplicando el desplazamiento.
De lo anterior se concluye que para acceder a 32 bytes hay que realizar 32×4 accesos, es decir 128 accesos a memoria.

v. [2 puntos] Proponga una mejora que, sin modificar la jerarquía de tablas de páginas, permita reducir la cantidad de accesos a memoria. Muestre un ejemplo concreto en donde funcione.

Solución:
La adición de una TLB ayuda a reducir la cantidad de accesos a memoria. Cada clave en la TLB está conformada por los primeros 41 bits de la dirección lógica y el valor corresponde al número de marco.
Un ejemplo concreto donde se reduce la cantidad de accesos corresponde a la siguiente secuencia de direcciones:
d_0 = 000...000 | 000...000 | 000...001 | 000...010
d_1 = 000...000 | 000...000 | 000...001 | 110...010
...
d_{31} = 000...000 | 000...000 | 000...001 | 100...010
NOTA: el ejemplo asume que los primeros 41 bits son iguales y lo que cambia son los últimos 15.
Asumiendo que la TLB comienza vacía, para acceder a la primera dirección no se perciben mejoras. En total se realizan 4 accesos a memoria. Sin embargo, al resolver las sucesivas direcciones, únicamente hay un acceso a memoria, ya que la asociación de la dirección y el número de marco se encuentra en la caché asociativa.
De esta manera, se puede concluir que agregar una TLB al esquema, disminuye la cantidad de accesos a memoria. En particular, siguiendo el ejemplo presentado pasa de 128 a 35 accesos para leer 32 bytes.

Problema 3 (16 puntos)

Un sistema de archivos usa una estrategia indexada de dos niveles con 7 bloques directos en el primer nivel y 1 bloque de indirección simple en el segundo nivel, y un mapa de bits para administrar el espacio libre del disco. Las estructuras de datos utilizadas por el sistema de archivos son:

```
const MAX_BLOQ = 65536;                 // 2^16
const TAM_BLOQ = 4096;                  // 2^12
const MAX_INOD = 8192;                  // 2^13

type inodo = Record
    usado: bool;                        // 1 bit
    inodo_num: int16;                   // 2 bytes
    es_dir: bool;                       // 1 bit
    tamano: int32;                      // 4 bytes
    directo: array [0..7] of int16;     // 16 bytes
    directo_tope: int16;                // 2 bytes
    indirecto: int16;                   // 2 bytes
    indirecto_tope: int16;              // 2 bytes
    reservado: array [0..29] of bit;    // 30 bits
end; // 32 bytes
```

```
type entrada_dir = Record
    usado: bool;                        // 1 bit
    nombre: array [0..123] of char;     // 124 bytes
    es_dir: bool;                       // 1 bit
    inodo_num: int16;                   // 2 bytes
    permisos: array [0..13] of bit;     // 14 bits
end; // 128 bytes

type bloque = array [0.. TAM_BLOQ-1] of byte;
type mapa_bits = array [0.. MAX_BLOQ-1] of bit;
type inodos_tab = array [0.. MAX_INOD-1] of inodo;
type disco = array [0.. MAX_BLOQ-1] of bloque;

var IT: inodos_tab;
    MB: mapa_bits;
    D: disco;
```

Se dispone de los siguientes procedimientos:

leerBloque(d: disco; bloque_num: 0..MAX_BLOQ-1; buffer: array [0..1023] of byte): bool, lee desde el disco d el bloque con índice bloque_num en la variable buffer. Retorna verdadero en caso de que la operación haya sido ejecutada con éxito y falso en caso contrario.

obtenerBase(path: array of char): array of char, retorna la primer parte del camino en path. Por ejemplo: obtenerBase('/home/sistoper/a.txt') = 'home'

obtenerResto(path: array of char): array of char, retorna el resto del camino en path sin la primer parte. Por ejemplo: obtenerResto('/home/sistoper/a.txt') = '/sistoper/a.txt'

Sabiendo que: las variables IT, MB, y D son globales, el inodo número 0 es el directorio raíz y las entradas de los directorios son almacenadas utilizando un array de entradas:

(a) I. [2 puntos] Indique el tamaño máximo que puede tener un archivo. Justifique.

Solución:

- Tamaño máximo según el campo `tamaño` del inodo: $2^{32} - 1$ bytes = 4 GB -1 bytes
- Tamaño máximo según la cantidad de bloques del disco: $65536 - 1 \times 4096 = 2^{16} \times 2^{12} = 2^{28}$ bytes
- Tamaño máximo según la cantidad de bloques asignables a un archivo: $(8+(2^{12}/2^{1})) \times 2^{12} = (8 + 2^{11}) \times 2^{12} < 2^{12} \times 2^{12} = 2^{24}$ bytes

  El tamaño máximo está determinado por la menor de estas cotas. Por lo tanto es: $(8 + 2^{11}) \times 2^{12}$ bytes

II. [2 puntos] Indique la cantidad máxima de archivos que puede almacenar el sistema. Justifique.

Solución: Para encontrar la cantidad máxima de archivos se estudia cuántos archivos de tamaño 0 es posible almacenar. Tenemos dos cotas para esto: (a) la cantidad máxima de inodos disponibles y (b) la cantidad máxima de archivos almacenables en la estructura de directorios.
(a) La cantidad máxima de inodos es $2^{13} = 8192$ y descontando el inodo raíz tenemos 8191 disponibles. Como cada archivo necesita un inodo no podemos tener más de 8191 archivos de tamaño 0.
(b) La cantidad máxima de archivos que puede contener el directorio raíz es $(8+2^{11}) \times 2^{12}/2^{7}$ archivos, 2^{5} `entrada_dir` por cada bloque asignado al directorio. Este valor es mayor que

8191.
Entonces, la cantidad máxima de archivos de tamaño 0 es 8191 y se da cuando el directorio raíz contiene todos los archivos.

(b) [2 puntos] Explique como funciona el mapa de bits. Explique el funcionamiento de alguna alternativa al mapa de bits.

Solución: El mapa de bits sirve para llevar registo de cuáles bloques se encuentran ocupados y cuáles libres. Se implementa como un vector de bits con igual tamaño que la cantidad de bloques en el disco: `type mapa_bits = array [0.. MAX_BLOQ-1] of bit`. Se marca con 0 los bloques libres y con 1 los bloques ocupados.
Una alternativa al mapa de bits es utilizar una lista de bloques libres. Esta lista se construye con los propios bloques libres, almacenando en cada bloque libre un puntero al siguiente de la lista.

(c) [2 puntos] Implemente una función que dado el nombre de un elemento (archivo o directorio) y el índice de un bloque asignado a un directorio, busque el elemento en el bloque. La función debe retornar en `inodo_num` el número de inodo correspondiente al elemento encontrado, o −1 si no es encontrado. La función retorna `falso` en caso de error y `verdadero` en caso contrario, y tiene la siguiente firma: buscarEntrada(nombre: array of char; bloqueid: int16; var inodo_num: int16): bool

Solución:

```
Procedure buscarEntrada(nombre: array of char; bloqueid: int16; var inodo_num: int16): bool {
    entrada_dir buffer [32];
    inodo_num = -1;

    if (nombre=="" || bloqueid < 0 || bloqueid > MAX_BLOQ-1 || MB[bloqueid]==0 ) return false;
    if (!leerBloque(D, bloqueid, buffer)) return false;
    for (int j=0; j<32; j++) {
        if (buffer[j].usado and buffer[j].nombre == nombre) {
            inodo_num = buffer[j].inodo_num;
            return true;
        }
    }
    return true;
}
```

(d) [6 puntos] Implemente una función que retorne en `inodo_num` el número de inodo correspondiente al elemento indicado por la ruta absoluta (o path absoluto), o -1 en caso de no encontrar el elemento. La función retorna `falso` en caso de error y `verdadero` en caso contrario. La función a implementar tiene la siguiente firma: darInodo(path: array of char; var inodo_num: int16): bool

Solución:

```
Procedure darInodo(path: array of char; var inodo_num: int16): bool {
    inodo_num = 0;

    if (path == "") return false;
    if (path == "/") return true;

    string primero, resto;
```

```
    primero = obtenerBase(path);
    path = obtenerResto(path);

    int16 indblq[2048], foundinode;
    bool encontrado = false;

    while (primero != "" && !encontrado) {
        for (int i=0; i<IT[inodo_num].directo_tope && !encontrado; i++) {
            if (!buscarEntrada(primero, IT[inodo_num].directo[i], foundinode)) return false;
            encontrado = (foundinode >= 0);
        }
        if (!encontrado &&
            IT[inodo_num].directo_tope == 8 &&
            IT[inodo_num].indirecto_tope>0) {
            if (!leerBloque(D, IT[inodo_num].indirecto, indblq)) return false;
            for (int i=0; i<IT[inodo_num].indirecto_tope && !encontrado; i++) {
                if (!buscarEntrada(primero, indblq[i], foundinode)) return false;
                encontrado = (foundinode >= 0);
            }
        }
        if (encontrado) {
            primero = obtenerBase(path);
            path = obtenerResto(path);
            inode_num = foundinode;
            if (!IT[inodo_num].es_dir && path != "") return false;
        } else {
            inodo_num = -1;
        }
    }
    return true;
}
```

(e) Describa y justifique los cambios a realizar para que al sistema soporte:

I. [1 punto] hardlinks.

Solución: Alcanza con agregar un contador de referencias en el inodo. El contenido de los bloques apuntados por el inodo debe borrarse, y el inodo marcarse como no usado, solamente cuanto el contador llegue a cero.

II. [1 punto] softlinks.

Solución: Para esto se podría crear un nuevo tipo de elemento llamado `softlink`, ademas del archivo y el directorio. Para eso sería necesario cambiar los campos `es_dir` de bool a un enumerado con estos tres tipos. Luego cada softlink podría tener un inodo propio de tipo `softlink` con un único bloque asignado conteniendo el camino absoluto al archivo o directorio al que apunta.