# File System

1. **Sea un disco que posee 256 bloques de 8kb y un sistema operativo cuyos i-nodos son de 512 bytes. Defina la estructura completa del sistema de archivos unix-like. Justificar cada elección.**

Para definir la estructura completa del sistema de archivos Unix-Like utilizaremos el modelo Very Simple File System, que consta de los siguientes elementos:

* 1 súper bloque para almacenar la información del file system
* 1 bitmap de inodos para almacenar la información que respecta al estado libre (0) u ocupado (1) de un inodo
* 1 bitmap de datos para almacenar la información que respecta al estado libre (0) u ocupado (1) de una región para almacenar datos
* inodos para almacenar la metadata de cada archivo que se guarde en el file system (tamaño del archivo, fecha de modificación, propietario e información de seguridad)
* regiones de datos para almacenar la información persé de un archivo, son arreglos de bytes sin tipo

Veamos que como cada bloque tiene una capacidad de 8kb, cada uno puede almacenar hasta 8192 bytes: 1 kb = 1024 bytes => 8 \* 1024 = 8192 bytes.

Ahora bien, como cada inodo pesa 512 bytes, en cada bloque entrarán como máximo 16 inodos: 8192 / 512 = 16.

De esta forma, y según VSFS, vamos a tener 5 bloques para inodos, pudiendo alojar, como máximo 80 inodos (archivos): 5 \* 16. Además, vamos a tener 248 bloques para alojar datos.

El resultado final es un file system con la siguiente estructura:

| **S** | **i** | **d** | i | i | i | i | i |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |  | d | d | d | d | d | d | d | d |

Veamos que, cuando el Sistema Operativo acceda al File System, lo primero que va a leer será el súper bloque, de modo que pueda cargar toda la información correspondiente antes de administrarlo. Luego tendrá disponibles en los dos bloques siguientes cuáles inodos y regiones de datos están libres o no, y el resto del File System consiste en leer la metadata de cada archivo y acceder con la misma a los datos efectivamente.

# 

# Scheduling y Memoria Virtual

1. **Se tiene el siguiente esquema de procesos:**

| **Proceso** | **Duración** |
| --- | --- |
| **P1** | **73** |
| **P2** | **23** |
| **P3** | **61** |
| **P4** | **43** |
| **P5** | **37** |
| **P6** | **29** |

**Q1 = RR (quantum=5)**

**Q2 = RR (quantum=10)**

**Q3 = FIFO**

**El time arrival de los procesos es 0 para todo, no hay Boost time, se pide calcular utilizando MLFC:**

* **Completion time: tiempo t en el que se completó el proceso**
* **Turnaround time (CT - AT)**
* **Waiting time (TAT - duración)**

**Muestre la evolución del sistema de colas.**

| Q1 | P1(73), P2(23), P3(61), P4(43), P5(37), P6(29) |
| --- | --- |
| Q2 |  |
| Q3 |  |

Como Q1 se ejecuta con Round Robin, vemos cuál es el proceso de mayor prioridad para ser ejecutado, es decir, primero ejecuta P6. Ahora bien, como el mismo tiene una duración mayor al time slice definido para la cola en cuestión, según las reglas de MLFQ baja la prioridad una unidad (y por ende pasa a Q2):

| Q1 | P1(73), P2(23), P3(61), P4(43), P5(37) |
| --- | --- |
| Q2 | P6(24) |
| Q3 |  |

Repetimos hasta que todos los procesos estén en Q2 pues ninguno tiene un tiempo de duración menor al time slice:

| Q1 |  |
| --- | --- |
| Q2 | P1(68), P2(18), P3(56), P4(38), P5(32), P6(24) |
| Q3 |  |

Como Q1 quedó vacía, la próxima cola que será seleccionada para ejecutar procesos será Q2. Nuevamente, ejecutamos RR aunque esta vez con un time slice de 10 y repetimos los mismos pasos:

| Q1 |  |
| --- | --- |
| Q2 |  |
| Q3 | P1(58), P2(8), P3(46), P4(28), P5(22), P6(14) |

Como Q2 quedó vacía, la próxima cola a ejecutar es Q3. Esta tiene un método de ejecución de tipo FIFO de modo que el primer proceso en llegar es el último en ejecutar. De esta forma, los procesos se ejecutarán en el siguiente orden, completando su ejecución: P6, P5, P4, P3, P2, P1.

Finalmente, calculemos las métricas:

* Completion time
  + P6 -> 5 + ( 5 \* 5 ) + 10 + ( 10 \* 5 ) + 14 = 104
  + P5 -> 104 + 22 = 126
  + P4 -> 126 + 28 = 154
  + P3 -> 154 + 46 = 200
  + P2 -> 200 + 8 = 208
  + P1 -> 208 + 58 = 266
* Turn Around Time (como el time arrival es 0, equivale a CT)
  + P6 = 104
  + P5 = 126
  + P4 = 154
  + P3 = 200
  + P2 = 208
  + P1 = 266
* Waiting Time
  + P6 = 104 - 29 = 75
  + P5 = 126 - 37 = 89
  + P4 = 154 - 43 = 111
  + P3 = 200 - 61 = 139
  + P2 = 208 - 23 = 185
  + P1 = 266 - 73 = 193

1. **Explique tres políticas de Scheduling y compárelas entre ellas.**

* FIFO: Se ejecuta un proceso hasta que finaliza. Si hubiera más de uno, el orden de ejecución es que que el primero que se ejecuta es el primero llega.
* SJF: Se ejecuta primero el de menor duración.
* STC: Se ejecuta primero el de menor duración y es preemptive.

# 

# Concurrencia

1. **Dado el siguiente tipo de dato, identificar la sección crítica:**
2. struct QNode {
3. int key;
4. struct QNode\* next;
5. };
6. struct Queue {
7. struct QNode \*front, \*rear;
8. };
9. void deQueue(struct Queue\* q){
10. if (q->front == NULL)
11. return;
12. struct QNode\* temp = q->front;
13. q->front = q->front->next;
14. if (q->front == NULL)
15. q->rear = NULL;
16. free(temp);
17. }

Veamos que al comenzar la función, se accede al atributo *front* del objeto *q*, de modo que el estado del mismo afecta al resto de la implementación y si -durante otro proceso- fuera modificado, el resto de la implementación se vería afectada. Veamos un ejemplo:

1. Un proceso A llama a la función *deQueue(q0)*, donde *q0 != NULL*
2. El proceso A llega hasta la línea 12 y comienza la ejecución de un proceso B que llama a la misma función *deQueue(q0)*
3. El proceso B llega hasta la línea 12 y continúa la ejecución del proceso A
4. El proceso A llega a la línea 13 y modifica el valor de *q->front* por *NULL*, luego continúa hasta la línea 16 y continúa la ejecución del proceso B
5. El proceso B llega a la línea 13 y se lanza una excepción de tipo segmentation fault, pues *q->front == NULL* y no puede asignársele el valor de *q->front->next*.

Esto podría haberse evitado si el objeto *q0* se hubiera lockeado durante la ejecución de ambos procesos. Así, la sección crítica del código en cuestión es toda la implementación de la función *deQueue* y podría corregirse de la siguiente manera:

1. struct QNode {
2. int key;
3. struct QNode\* next;
4. };
5. struct Queue {
6. struct QNode \*front, \*rear;
7. };
8. void deQueue(struct Queue\* q){
9. lock(q);
10. if (q->front == NULL)
11. unlock(q);
12. return;
13. struct QNode\* temp = q->front;
14. q->front = q->front->next;
15. if (q->front == NULL)
16. q->rear = NULL;
17. free(temp);
18. unlock(q);
19. }

# 

# Procesos

1. **Describa las system calls fork(), exec() y wait() (qué hacen, para que se usan, etc.) y su peculiar relación con la creación de procesos en UNIX. Ponga un ejemplo de su funcionamiento en el cual se describe paso a paso lo que está sucediendo.**

* fork: Se usa para crear un nuevo proceso (hijo) que es una copia exacta del proceso que lo llamó (padre). Cada proceso hijo tiene un ID único, un parentId que lo relaciona con su proceso creador y una copia de los mismos archivos abiertos de su padre. Ejemplo:

int main(){

int a = 0;

int f = fork();

if (f == 0){

a = 5; // modifico el valor de a sólo para el proceso hijo

printf(“Child Process”);

printf(“a equals to %d\n”, a); // imprime “a equals to 5”

} else {

printf(“Parent Process”);

printf(“a equals to %d”, a); // imprime “a equals to 0”

}

return 0;

}

El proceso principal crea un proceso hijo que modifica el valor de la variable 5. De esta forma, para cada proceso *a* tiene un valor distinto, de modo que para el proceso hijo equivale a 5 mientras que para el proceso padre equivale a 0. Esto se debe a que cada proceso tiene su propio stack y por ende -si bien en la creación del proceso hijo, *a* equivale a 0- modificar el valor de la variable no afecta al estado del recurso en el proceso padre, que tiene una región de datos distinta a la de su hijo.

* exec: Se usa para reemplazar la ejecución del proceso actual por otro. El Sistema Operativo desaloja el proceso anterior y actualiza los valores de contexto, registros de CPU y archivos por los correspondientes al nuevo proceso que solicita ejecutarse.

int main(){

char\* args[] = {“./CAT”, “hola.txt”, NULL}

execvp(args[0], args);

}

Se ejecuta en la shell el comando *cat hola.txt*. La ejecución del proceso inicial (el que llamó a *main*) finaliza y puede o no lanzar un error según el valor resultante de execvp.

* wait: Se usa para esperar la finalización de un proceso hijo, suspendiendo la ejecución del proceso padre (que es el que lo llama).

int main(){

int a = 0;

int fds[2];

int f = fork();

pipe(fds);

if (f == 0){

close(fds[0]);

a = 5; // modifico el valor de a sólo para el proceso hijo

printf(“Child Process”);

printf(“a equals to %d\n”, a); // imprime “a equals to 5”

write(fds[1], a);

close(fds[1]);

} else {

waitpid(f, 0, NULL);

close(fds[1]);

read(fds[0], a);

printf(“Parent Process”);

printf(“a equals to %d”, a); // imprime “a equals to 5”

close(fds[0]);

}

return 0;

}

El proceso principal crea un proceso hijo y espera a que termine de ejecutarse para verificar el valor de la variable *a*, la cual tomará el peso que le venga asignado por el pipe intercomunicador entre el proceso hijo y padre. Si no se usase el método *wait* no podríamos asegurar que el valor de *a* equivalga a 5 en la ejecución de las instrucciones del proceso padre.

1. **¿Qué es el Kernel? ¿Por qué existe en UNIX? Dé un ejemplo detallado en el cual se vea la necesidad de la existencia del Kernel y describa el proceso que lo involucra desde el user space hasta el user space. ¡Detalladamente!**

El Kernel es la capa de software de más bajo nivel y cumple la función de mediador entre los programas y los recursos del hardware para protegerlos eliminando la ejecución directa. La razón de existir del mismo es mediar entre el programador y el computador, de modo que el Sistema Operativo pueda controlar y administrar los recursos físicos de una forma controlada. Si bien el programador puede acceder a estos, no lo hace de forma directa si no que tiene que interactuar con la API que provee el Kernel efectivamente. En sí, la función del Kernel es:

* planificar la ejecución de los programas
* gestionar el uso de la memoria
* proveer de un sistema de archivos
* gestionar la creación y finalización de procesos
* proveer una API para el usuario
* manejar las señales
* gestionar los recursos físicos de entrada y salida