Sistemas Operativos

Manuel Mena

5 de agosto de 2021

Índice

1. Práctica 1

1.1.

Se debe guardar el PCB y los registros del proceso desalojado, y luego cargar el PCB y los registros del nuevos proceso asignado.

1.2.

```
Ke_context_switch(PCB* pcb_0, PCB* pcb_1) {
    pcb_0->CPU_TIME += ke_current_user_time();
    pcb_0->STAT = KE_READY;
    pcb_0->PC = PC;
    pcb_0->R0 = R0;
    pcb_0->R1 = R1;
    pcb_0->R2 = R2;
    pcb_0->R3 = R3;
    pcb_0->R4 = R4;
    pcb_0->R5 = R5;
    pcb_0->R6 = R6;
    pcb_0->R7 = R7;
    pcb_0->R8 = R8;
    pcb_0->R9 = R9;
    pcb_0->R10 = R10;
    pcb_0->R11 = R011
    pcb_0->R12 = R12;
    pcb_0->R13 = R13;
    pcb_0->R14 = R14;
    pcb_0->R15 = R15;

    PC = pcb_1->PC
    R0 = pcb_1->R0
    R1 = pcb_1->R1
    R2 = pcb_1->R2
    R3 = pcb_1->R3
    R4 = pcb_1->R4
    R5 = pcb_1->R5
    R6 = pcb_1->R6
    R7 = pcb_1->R7
    R8 = pcb_1->R8
    R9 = pcb_1->R9
    R10 = pcb_1->R10
    R11 = pcb_1->R11
    R12 = pcb_1->R12
    R13 = pcb_1->R13
    R14 = pcb_1->R14
    R15 = pcb_1->R15

    pcb_1->STAT = KE_RUNNING;

    ke_reset_current_user_time();
    ret();
}
```

1.3.

Una system call es un llamado a un servicio del kernel, como puede ser realizar entrada/salida a un dispositivo o lanzar un proceso hijo mientras que una llamada a función de biblioteca realiza operaciones

con los registros y direcciones de memoria en el espacio de direcciones del usuario.

1.5.

```
pid_t homero = fork();
if (homero != 0) {
    wait_for_child(homero);
    printf('Abraham');
    exit();
} else {
    pid_t bart = fork();
    if (bart != 0) {
        pid_t lisa = fork();
        if (lisa != 0) {
            pid_t maggie = fork();
            if(maggie != 0) {
                wait_for_child(bart);
                wait_for_child(lisa);
                wait_for_child(maggie);
                printf('Homero');
                exit();
            } else {
                printf('Maggie');
                exit();
            }
        } else {
            printf('Lisa');
            exit();
        }
    } else {
        printf('Bart');
        exit();
    }
}
```

1.6.

```
void system(const char *arg) {
    pid_t pid = fork();
    if (pid == 0) exec(arg);
    else {
        wait_for_child(pid);
        exit();
    }
}
```

1.7.

1.7.a.

Deben residir tiki y taka en el área de memoria compartida

1.7.b.

temp no debe residir en el espacio de memoria compartida

1.7.c.

```
int main() {
    pid_t child = fork();
    if (child == 0) {
        taka_runner();
    } else {
        share_mem(&tiki);
        share_mem(&taka);
        tiki_taka();
    }
}
```

1.8.

1.8.a.

```
pid_t parent = get_current_pid();
pid_t child = fork();
int count = 0;
if (child == 0) {
    while (true) {
        breceive(parent);
        bsend(parent, 2 * count + 1);
        count++;
    }
} else {
    while (true) {
        bsend(child, 2 * count);
        breceive(child);
        count++;
    }
}
```

1.8.b.

```
pid_t parent = get_current_pid();
int count = 0;

pid_t child_1 = fork();
if (child_1 == 0) {
    while (true) {
        bsend(child_1, 3 * count + 1);
        breceive(child_2);
        count++;
    }
} else {

    pid_t child_2 = fork();
    if (child_2 == 0) {
        while (true) {
            breceive(child_1);
            bsend(parent, 3 * count + 2);
            count++;
        }
    } else {
        while (true) {
            bsend(child_1, 3 * count);
            count++;
            breceive(child_2);
        }
    }
}
```

1.9.

1.9.a.

No es realizable ya que el proceso de la izquierda comienza enviando result, por lo que hasta que el proceso de la derecha no reciba el mensaje, el de la izquierda permanecerá bloqueado, ya que el sistema operativo posee una cola de mensajes de capacidad cero.

Una ejecucion posible es que en el tiempo 1 se ejecute $computo_muy_dificil_2$, y de ahi en adelante si pueden ejecutarse ambos.

1.9.b.

Podría utilizarse una cola de mensajes de mayor capacidad para evitar que el proceso de la derecha se bloquee al enviar el mensaje.

1.10.

1.10.a.

Memoria compartida, porque es más rápida. En este caso las funciones no pueden pisarse.

1.10.b.

Pasaje de mensajes. Si lo hiciera con memoria compartida tendría que haber alguna especie de busy waiing para reconocer en que momento termina cortarBordes. Con mensajes esto se hace sin desperdiciar recursos.

1.10.c.

Pasaje de mensajes. Es imposible usar memoria compartida ya que los procesos se ejecutarán en computadoras distintas. Además, en caso de que sí fuera posible, parece ser que por el nivel de seguridad al que está expuesto eliminarOjosRojos, no es buena idea usar memoria compartida ya que podría ser inseguro.

1.12.

Si sólo hubiera pipes como comunicación interprocesos el sistema se vería enlentecido ya que la falta de señales haría imposible que un proceso fuera interrumpido por lo que debería suplirse haciendo una especie de polling, lo cual desperdiciaría mucho tiempo de ejecución.

1.14.

El error esta en utilizar $shared_child_pid$ siendo el padre para recibir del hijo ya que es posible que el hijo todavia no se haya ejecutado, lo que provocaría que no se le haya asignado un valor todavía a $shared_child_pid$.

La solución es reemplazarlo por child, que ya contiene el pid del hijo.

1.16.

```
int assignPipe[2];
int resultPipe[2];

int file = open(‘‘nombre_de_archivo’’);
int *list;

pipe(assignPipe);
pipe(resultPipe);

for(int i = 0; i < 8; i++) {
    int child = fork();
    if (child == 0) {
        close(assignPipe[1]);
        close(resultPipe[0]);

        do {
            read(assignPipe[0], list, sizeof(list));
            write(resultPipe[1], calcular_promedio(list));
        } while(list != loquesea);

        close(assignPipe[0]);
        close(resultPipe[1]);
        exit();
    }
}

close(assignPipe[0]);
close(resultPipe[1]);

int rowsRead = 0;
while(cargarFila(file, list) || rowsRead < 8) {
    rowsRead++;
    write(assignPipe[1], list, sizeof(list));
}

char *proms;
int prom;
do {
    read(resultPipe[0], prom, sizeof(prom));
    proms ++ prom;

} while ()

close(‘‘nombre_de_archivo’’);
```

2. Práctica 2

2.1.

2.1.a.

CPU: [0,2], [11,13], [21,22] E/S: [3,10], [14,20]

2.1.b.

3 y 2 las de CPU, 7 las de E/S

2.4.

2.4.a.

No

2.4.b.

Si, que se este ejecutando una tarea de cierta prioridad, llegue una de menor prioridad que no es atendida, y que a partir de ahí sigan llegando tareas de mayor prioridad ocasionando que nunca se atienda la tarea de menor prioridad.

2.4.c.

Si, que se este ejecutando una tarea de cierta duración, llegue una de mayor duración que no es atendida, y que a partir de ahí sigan llegando tareas de menor duración ocasionando que nunca se atienda la tarea de mayor duración.

2.4.d.

No

2.5.

2.5.a.

El efecto es que ese proceso se ejecuta más

2.5.b.

Ventaja es que puede discriminarse el quantum de cada proceso, lo cual permite darle importancia en cierto sentido.

La desventaja es que si se usan múltiples procesadores podría ocurrir que mientras un procesador este ejecutando una tarea, otro caiga en el PCB duplicado

2.5.c.

Puede utilizarse una lista con los PCB no duplicados y otra con las duraciones de los quantums de cada proceso.

2.7.

2.7.a.

Tarea	T_0	T_1	T_2	Promedio
Espera	16	21	3	13.3
Turnaround	32	34	24	

3. Práctica 3

3.1.

3.1.a.

No

3.1.b.

Puede imprimir 1 en los siguientes casos:

- Si se ejecuta primero A y luego B
- Si colisionan las ejecuciones de el incremento de X, provocando que se ejecuten ambas líneas pero X se incremente sólo una vez

Puede imprimir 2 en caso de que se ejecuten B y luego A o que se ejecuten simultaneamente pero que el printf sea lo último.

3.2.

Primero siempre imprime un 0 de la primer X, pero luego como Y ya es 1 puede imprimir cualquier número de 'a's, incluyendo 0, seguido de 1, 2, 3 y 4.

3.3.

Primero imprime 'b' ya que A se queda haciendo busy waiting sobre X. Luego, es posible que imprima "adc" o "acd".

3.4.

Si, los procesos acceden y modifican a la variable dentro del mutex, pero lo liberan y luego toman decisiones respecto a ella, podría producirse una race condition.

3.5.

Podría suceder que se produzca una inanición de los procesos que llaman primero a wait() ya que quedan en el fondo de la pila hasta que todos los que llegan después se liberen.

3.6.

Supongamos que wait() y signal() no se ejecutaran atómicamente. Supongamos que tenemos una ejecución de dos procesos y ambos ejecutan wait(). Como wait() no es atómica una ejecución posible es que ambos procesos lean la capacidad, salgan del while, y al momento de decrementarla ya están los dos fuera del ciclo, por lo que ambos ya son capaces de entrar a la zona crítica, violando el principio de exclusión mutua.

3.8.

```
bool TryToLock() {
    return !reg.testAndSet();
}
```

3.9.

```
Semaphore semaforos[N] = {0,...,0,1,0,...,0};

for (int j = 0; j < N; j++) {
    int pid = fork();
    if (pid == 0) {
        while (true) {
            semaforos[j].wait();
            P[j];
            semaforos[(j + 1) % N].signal();
        }
    }
}
```

3.10.

3.10.a.

```
// Semaforos para         A B C
Semaphore semaforos[3] = {1,0,0};

for (int i = 0; i < 3; i++) {
    int pid = fork();
    if (pid == 0) {
        while (true) {
            semaforos[i].wait();
            if(i == 0)
                A();
            else if (i == 1)
                B();
            else if (i == 2)
                C();
            semaforos[(i + 1) % 3].signal();
        }
    }
}
```

3.10.b.

```
// Semaforos para         B B C A
Semaphore semaforos[3] = {1,0,0,0};

for (int i = 0; i < 4; i++) {
    int pid = fork();
    if (pid == 0) {
        while (true) {
            semaforos[i].wait();
            if(i == 0 || i == 1)
                B();
            else if (i == 1)
                C();
            else if (i == 3)
                A();
            semaforos[(i + 1) % 4].signal();
        }
    }
}
```

3.10.c.

```
// Semaforos para          A  ByC
Semaphore semaforos[2] = { 1 , 0 };

for (int i = 0; i < 3; i++) {
    int pid = fork();
    if (pid == 0) {
        while (true) {
            if(i == 0) {
                semaforos[0].wait(2);
                A();
                semaforo[1].signal(2);
            }
            else if (i == 1) {
                semaforos[1].wait();
                B();
                semaforos[0].signal();
            }
            else if (i == 2) {
                semaforos[1].wait();
                C();
                semaforos[0].signal();
            }
        }
    }
}
```

3.10.d.

```
Semaphore sA = Semaphore(1);
Semaphore sB = Semaphore(0);
Semaphore sC = Semaphore(0);

bool sigueB = true;

for (int i = 0; i < 3; i++) {
    int pid = fork();
    if (pid == 0) {
        while (true) {
            if(i == 0) {
                sA.wait(2);
                A();
                if (sigueB) {
                    sigueB = false;
                    sB.signal(2);
                } else {
                    sigueB = true;
                    sc.signal();
                }
            }
            else if (i == 1) {
                sB.wait();
                B();
                sA.signal();
            }
            else if (i == 2) {
                sC.wait();
                C();
                sA.signal(2);
            }
        }
    }
}
```

3.11.

Se utiliza un contador global. Después de cada a_i se incrementa ese contador y luego se hace un wait(), pero antes se pregunta si ese proceso es el último en terminar. Si lo es se hace signal(N) para despertar a todos los procesos que ya hayan terminado a_i. Es necesario utilizar un lock al incrementar el contador para evitar una race condition. El lock debera abarcar también la condición para evitar que dos procesos entren con el contador en el mismo valor.

```
a();
spin.lock();
cont++;
if (cont == N)
    sem.signal(N);
spin.unlock();
sem.wait();
b();
```

3.12.

3.12.a.

No pueden terminar su ejecución.

3.12.b.

No entra en la definición ya que se encuentran esperando a que otros miembros del grupo liberen el lock.

3.12.c.

No puede terminar su ejecución.

3.12.d.

Si, entra en la definición ya que se está a la espera de que se libere el lock.

3.13.

3.13.a.

No se cumplen las condiciones de Coffman.

3.14.

Puede haber deadlock si un recurso toma R_2 y los otros dos toman R_1, y todos necesitan ambos recursos para liberar el lock.

Si R_1 puede ser compartido por los 3 entonces no puede haber deadlock, porque el que haya ganado el lock de R_2 puede liberarlo al obtener R_1, y así con los 3 procesos.

3.15.

No puede haber deadlock ya que como los procesos necesitan a lo sumo dos recursos, y son todos del mismo tipo, no hay forma de que alguno de los 3 procesos no consigo sus dos recursos, entonces si hay 2 procesos con un recurso cada uno y el otro con 2, este los libera al terminar y ya los otros pueden tomar uno cada uno.

3.16.

3.16.a.

Si, que cada proceso ejecute su primera linea, poniendo ambos semáforos en 0, para luego atascarse en el wait del semáforo contrario.

3.17.

Se puede ver analizándolo proceso a proceso.

- P_1 y P_3 ya tienen lo que necesitan por lo que eventualmente liberarán los recursos asignados: $2R_2$, $3R_1$ y $3R_3$
- Con eso P_2 puede cumplir sus necesidades ($2R_1$ y $2R_3$) liberando eventualmente $2R_1$ y $1R_4$ además de lo recientemente obtenido
- Gracias a esto P_4 y P_5 ya pueden solicitar los recursos que necesitaban y así terminar su ejecución, habiendo finalizado todos los procesos

3.17.a.

Si, hay deadlock. Anteriormente ocurría que P_1 y P_3 liberarían los recursos obtenidos ya que ya habían satisfacido sus necesidades, pero en este caso P_3 todavía necesita $2R_2$. El único que puede liberar recursos es P_1, pero como libera únicamente $1R_2$, esto no alcanza para satisfacer a P_3 y por lo tanto P_2, P_3, P_4 y P_5 quedan a la espera de que se les asigne sus respectivos recursos, sin liberarlos nunca.

3.18.

3.18.a.

```
a();
spin.lock();
cont++;
if (cont == N)
    sem.signal(N);
spin.unlock();
sem.wait();
b();
```

3.18.b.

```
a();
atomReg.inc();
while(atomReg.get() != N) {}
b();
```

3.18.c.

```
a();
if (atomReg.getAndInc() == N - 1)
    sem.signal(N);
sem.wait();
b();
```

3.18.d.

La más fácil de entender es en la que se usan sólo el registro atómico debido a su simpleza.

3.18.e.

La más eficiente es la tercera ya que no desperdicia tiempo haciendo busy waiting como las otras dos.

3.19.

```
// Variables globales:
atomic<int> clientesDentro;

queue<Sempaphore cliente> colaLocal;
queue<Sempaphore cliente, Semaphore barbero> colaSofa;

TasLock lockLocal;
TasLock lockSofa;
TasLock lockPagando;

Semaphore quieroPagar, aceptoPago, pagoAceptado;
int esperandoAPagar;

// Cliente:

Semaphore yo = new Semaphore(0);
Semaphore barbero = new Semaphore(0);

// Si el local está lleno me voy al chori
if (clientesDentro.getAndInc() > 20) {
    clientesDentro.getAndDec();
    exit();
}

lockLocal.lock();
colaLocal.push(yo);
lockLocal.unlock();
entrar();

// Si hay espacio en el sofa voy y me siento
lockSofa.lock();
if (colaSofa.size() < 4) {
    lockLocal.lock();
    colaLocal.pop();
    colaSofa.push(<yo, barbero>);
    sentarseEnSofa();
    lockLocal.unlock();
    lockSofa.unlock();
} else {
    lockSofa.unlock();

    // Espero a que me llamen cuando se desocupe un lugar en el sofa
    yo.wait();

    // Me despertaron, me siento en el sofa
    lockSofa.lock();
    lockLocal.lock();
    colaLocal.pop();
    colaSofa.push(<yo, barbero>);
    sentarseEnSofa();
    lockLocal.unlock();
    lockSofa.unlock();
}

// Espero a que el barbero me llame
yo.wait();
```

```
lockSofa.lock();

// Si el sofa estaba lleno tengo que avisarle al proximo que se siente
if (colaSofa.size() == 3) {
    parado.signal();
}

lockSofa.unlock();

sentarmeEnSilla();
barbero.signal();

// Espero a que el barbero termine de cortarme el pelo
yo.wait();

// Se terminó el corte, voy a pagar
lockPagando.lock();
esperandoAPagar++;
lockPagando.unlock();
quieroPagar.wait();

// Pago
pagar();
aceptoPago.signal();
pagoAceptado.wait();

// Pago aceptado, salgo
clientesDentro.getAndDec();
salir();
exit();

// Barbero:

Semaphore yo, cliente;

while(true) {
    lockSofa.lock();
    if (!colaSofa.empty()) {
        <Semaphore,Sempahore> sentado = colaSofa.top();
        colaSofa.pop();

        cliente = sentado.cliente;
        yo = sentado.barbero;

        // Le aviso al cliente que venga a ser atendido
        cliente.signal();

        // Espero a que el cliente se siente en la silla
        lockSofa.unlock();
        yo.wait();

        // El cliente se sienta
        cortarCabello();
```

```
        //Le aviso que termine de cortar el pelo
        cliente.signal();
    } else
        lockSofa.unlock();

    lockPagando.lock();
    if (esperandoAPagar > 1) {
        esperandoAPagar--;
        lockPagando.unlock();

        // Le aviso a alguno que puede pagar
        quieroPagar.signal();

        // Espero a que me paguen
        aceptoPago.wait();

        // Me pagan
        aceptoPago();
        pagoAceptado.signal();
    } else
        lockPagando.unlock();
}
```

3.20.

```
Semaphore semMacho[N] = {0,...,0}
Sempahore semHembra[N] = {0,...,0};

void P(i, sexo) {
    if (sexo == macho) {
        semMacho[i].signal();
        semHembra[i].wait();
    } else {
        semHembra[i].signal();
        semMacho[i].wait();
    }
    entrar(i);
}
```

4. Práctica 4

4.1.

Una dirección de memoria lógica es traducida a una de memoria lineal a través de la segmentación, y ésta es traducida a una dirección de memoria física mediante la paginación.

4.2.

Fragmentación interna es cuando los bloques son muy grandes para los datos almacenados dentro de ellos, lo que provoca que se desperdicie espacio. Fragmentación externa es cuando no hay suficiente memoria contigua para ser asignada, por lo que queda inutilizable.

4.3.

Bloques

1. 8MB 2MB
2. 1MB
3. 4MB 1MB
4. 512KB 12KB
5. 512KB
6. 2MB

Programas

1. 500KB
2. 6MB
3. 3MB
4. 20KB
5. 4MB

4.3.a.

- Programa 1 en bloque 4
- Programa 2 en bloque 1
- Programa 3 en bloque 3
- Programa 4 en bloque 2
- Programa 5 en bloque 1

4.4.

Porque con una dirección lineal de n cantidad de bits pueden direccionarse 2^n páginas, por lo que una la tabla tiene 2^n entradas.

4.7.

65536 bytes divididos en páginas de 4096 bytes hacen 16 páginas en total. Se necesitan 8 páginas para el texto, 5 para los datos y 4 para la pila, por lo que no es posible ejecutarlo.

Si el tamaño de página fuera de 512 bytes, se tendrían direccionadas 128 páginas. El texto del programa requeriría 64, los datos 33 y la pila 31. De esta forma sí podria ejecutarse.

4.8.

4.8.a.

400, porque se debe acceder primero a la entrada correspondiente de la tabla y una segunda vez para la página.

4.9.

Ocurre cuando se intenta acceder a una página que ya no está en memoria. Lo que el sistema hace en ese caso es guardar en disco una de las páginas ya cargadas en memoria para hacer espacio a la página que se está intentando acceder. Una vez hecho esto se lee la página de disco y se la carga en memoria.

4.11.

4.11.a.

Cada pagina contiene 200 posiciones por lo que habrá 50 fallos de página

4.11.b.

Como la matriz está almacenada contiguamente por por fila, este tipo de recorrido salta a través de las columnas, por lo que cada pagina solo albergará 2 posiciones, por lo que habran 5000 fallos de página

4.13.

4.13.a.

Tiene sentido si lo que se hace es dejar un segmento especialmente para atender las llamadas. Si fuese parte de la paginación, sería posible que la página en donde está almacenado el código para la atención de llamadas se encuentre fuera de memoria, por lo que se perdería tiempo en traerla de disco.

4.13.b.

Si, justamente para ahorrarse el tiempo de ir a buscarlas a memoria.

5. Práctica 5

5.1.

Son necesarios N accesos a memoria.

5.2.

5.2.a.

Es necesario leer los primeros 10 bloques directos.

5.2.b.

Es necesario leer los 12 bloques directos, el bloque de indirección simple y los primeros 8 bloques del bloque de indirección. 21 en total.

5.3.

5.3.a.

5.3.b.

Conviene usar inodos ya que FAT no está acotado.

5.3.c.

Conviene usar FAT ya que mediante inodos esta acotado por la cantidad de bloques del inodo.

5.3.d.

Conviene usar inodos porque se carga un inodo y los bloques del archivo a medida que se van leyendo, en FAT se debe cargar la tabla entera a memoria.

5.4.

5.4.a.

Los identificadores de bloque son de 24 bits, por lo que es posible direccionar hasta 2^{24} bloques. El del hash es de 16 bits por lo que el tamaño de la tabla es de 2^{16} entradas.

La FAT contiene solo identificadores de bloque de 24 bits, por lo que su tamaño es de $2^{24} \times 24$ bits. Lo que es lo mismo que $2^{24} \times 3$ bytes $= 48MiB$.

En cuanto a la tabla de hash, cada entrada contiene el identificador de bloque (24 bits) y el tamaño del archivo. El tamaño de un archivo no está actotado, por lo que en principio podría llegar a ser de 16GiB, que es el tamaño del disco. Por lo que para almacenar el tamaño del archivo se necesitan 34 bits, siendo un total de 58 bits por entrada. El tamaño de la tabla de hash es de $2^{16} \times 58$ bits = 3801088 bits = 14848 bytes, poco más de 14KiB.

Como el tamaño de bloque es de dos sectores, y cada sector de 1KiB, la unidad mínima es de 2KiB. La FAT queda almacenada a la perfección en 24Ki sectores, pero el hash no, por lo que sufre de un poco de fragmentación interna; se necesitan 7 bloques para cubrir 14KiB y un bloque extra para el resto.

Ambas estructuras ocupan entonces 24Ki + 8 bloques, lo que equivale a 50348032 bytes. Quedando mas de 15,95 Gib para los archivos.

5.4.b.

El tamaño del bloque debería ser de 2 sectores, ya que es lo que mejor se ajusta al tamaño de los archivos, por mas que se desperdicie la mitad de la memoria. Esto hace que la cantidad de bloques posibles a ser direccionados sea de $\frac{16GiB}{2KiB} = 8Mi$ bloques $= 2^23$ bloques. Por lo que se necesitan 23 bits para direccionarlos, lo cual genera la necesidad de que el tamaño de los identificadores de bloque sea de

24 bits. Por último, como cada archivo requerirá en promedio 1 bloque, puede decirse que se tiene la misma cantidad de archivos que de bloques, entonces el tamaño del hash también es de 24 bits.

5.4.c.

En caso en que los archivos tuviesen un tamaño promedio de 16MiB, uno intentaría usar el tamaño máximo de bloque que es de 8 sectores (8 KiB). Pero la cantidad de bits a utilizar para los identificadores de bloque diferiría en 2 (o en 1 si se usasen bloques de 4 sectores), lo cual lleva a utilizar la configuración de 24 bits de todos modos, y lo mismo para el hash.

5.5.

Si, en caso en que varios procesos intenten leer el mismo bloque, con raid 0 deberán turnarse para leerlo, mientras que con raid 1 cada proceso puede leer una copia distinta.

5.6.

5.6.a.

2 accesos: el bloque a escribir y el de paridad.

5.6.b.

9 accesos: 7 para los bloques, 1 para la paridad de los primeros 4 y otro para la de los últimos 3.

5.8.

5.8.a.

Falso, siempre puede incrementarse el nivel de protección agregando una copia más, en caso de que el resto de las copias fallen.

5.8.b.

La aumenta en el sentido de que esa cinta puede ser robada, pero también puede ser robado el disco en donde está almacenada la información, y de hecho es más vulnerable dentro del disco ya que es posible que sea accedido de manera remota, lo cual no ocurre con la cinta.

5.9.

Un snapshot es una imágen del estado del sistema, una forma de recuperar

6. Práctica 6

6.1.

Polling sería más conveniente en caso de que se realice una comunicación constante, para cualquier caso en que se envíe una ráfaga de datos es mucho más costoso el tiempo perdido en cambios de contexto si se implementara el uso de interrupciones que el tiempo de espera activa del polling.

Si la comunicación tuviese intervalos largos o no determinados, conviene la estrategia de interrupciones, como en un teclado.

El caso híbrido es el que se envían ráfagas de datos separadas por intervalos largos. Se utilizan las interrupciones para establecer la comunicación, se envía la ráfaga de datos atendida mediante polling, y una vez finalizada se espera a la próxima interrupción.

6.2.

El uso de E/S de un procesos en promedio es de 0.77, entonces el uso de la CPU para 6 procesos es

$$U_{CPU}(6) = 1 - U_{E/S}(1)^6 = 1 - 0{,}2084 = 0{,}7916$$

En cuanto al DMA,

$$U_{DMA}(6) = 1 - U_{CPU}(1)^6 = 1 - 0{,}23^6 = 0,9998$$

6.3.

6.3.a.

Si, es posible si el dispositivo tiene su propia memoria en donde encolar los pedidos. También puede lograrse simplemente teniendo una cola a donde almacenar los pedidos y que haya un proceso encargado de desencolar y comunicarse con el dispositivo.

6.3.b.

La latencia mejora ya que los procesos no necesitan quedarse esperando a que el dispositivo esté listo.
La liberación de recursos es mayor ya que no se hace espera activa.
El throughput mejora porque es posible usar los recursos liberados para otras tareas.

6.3.c.

No tiene sentido hacer spooling en una placa de red ya que se necesita una garantía en la comunicación que haciendo spooling se pierde. Si se hiciera spooling se perdería el orden de los pedidos y la sincronicidad en la comunicación. Además no tiene sentido hacer spooling en cuanto a la lectura.

6.4.

6.4.a.

Debe estar al tanto de que haya spooling para saber como reaccionar en cuanto a la velocidad de respuesta.

6.4.b.

El driver no sabe que hay o no spooling, ya que este método se aplica previamente a comunicarse con el mismo.

6.5.

6.5.a.

Si, el driver es código.

6.5.b.

No, el driver es solamente software.

6.5.c.

No. Es una porción de software hecha por el fabricante del dispositivo a controlar.

6.5.d.

No, se ejecuta en espacio de kernel.

6.5.e.

Falso. Además de interrupciones, puede trabajar mediante polling o DMA.

6.5.f.

Falso. El driver debe saber en que SO corre ya que cada uno provee una API diferente y se comporta de manera distinta.

6.6.

```
int driver_init() {}
int driver_remove() {}
int driver_open() {}
int driver_close() {}

int driver_read(int *data) {
    *data = IN(CHRONO_CURRENT_TIME);
    return IO_OK;
}

int driver_write(int *data) {
    OUT(CHRONO_CTRL, CHRONO_RESET);
    return IO_OK;
}
```

6.7.

6.11.

6.11.a.

```
int write(int sector, void *data) {
    if (DOR_STATUS)
        OUT(DOR_IO, 1);
    sleep(50);

    OUT(ARM, sector / cantidad_sectores_por_pista());
    while(ARM_STATUS) {}

    OUT(SEEK_SECTOR, sector % cantidad_sectores_por_pista());

    escribir_datos(data);
    while(DATA_READY) {}
}
```

6.11.b.

```
Semaphore ready;
Semaphore timer;

void arm_or_data_ready() {
    ready.signal();
}

void timer_ready() {
    timer.signal();
}

int driver_init() {
    ready = new Sempaphore(0);
    timer = new Sempaphore(0);
    request_irq(6, arm_or_data_ready);
    request_irq(7, timer_ready);
}

int write(int sector, void *data) {
    if (DOR_STATUS)
        OUT(DOR_IO, 1);
    timer.wait();

    OUT(ARM, sector / cantidad_sectores_por_pista());
    ready.wait();

    OUT(SEEK_SECTOR, sector % cantidad_sectores_por_pista());

    escribir_datos(data);
    ready.wait();

    return(IO_OK);
}
```

7. Práctica 7

7.1.

7.1.a.

Se aplica la función de hash a la contraseña que ingresó el usuario y si el resultado coincide con el valor almacenado entonces el ingreso es exitoso.

7.1.b.

La probabilidad de acertar con una contraseña distinta es de $\frac{1}{2^6 4}$.

7.1.c.

Cantidad de contraseñas a probar: $2^6 3$.
Cantidad de contraseñas por segundo: 10^9.
Cantidad de segundo por año: $3{,}15 \times 10^7$.
Serían necesarios $\frac{2^6 3}{10^1 6 \times 3{,}15} = 293$ años.

7.1.d.

Cantidad de contraseñas a probar: 36^6.
Cantidad de contraseñas por segundo: 10^9.
Serían necesarios $\frac{36^6}{10^9} = 2{,}18$ segundos.

7.2.

7.2.a.

No es seguro ya que es vulnerable a un replay-attack. Consiste en interceptar la información y retransmitirla, haciéndose pasar por otra persona.

7.2.b.

El atacante tendría el seed y el hash, lo que podría hacer es hashear todas las contraseñas posibles con ese seed hasta que coincida con el hash robado. Pero para eso necesitaría tener la función de hash.

7.3.

7.3.a.

El problema se introduce en la funcion *gets*, puesto que esta no se fija que la entrada tenga menor tamaño que la variable a donde se almacenará, y como consecuencia el contenido pisa el resto de la pila de la función pudiendo pisarse la dirección de retorno, desencadenando la ejecución de otro programa almacenado en otra posición de memoria.

7.3.b.

Únicamente la variable nombre.

7.3.c.

La dirección de retorno de *gets* si.

7.3.d.

No. Una vez que se llama a *gets* el usuario ya tomó el control.

7.4.

Primero se apilan los parámetros de *login*, luego la dirección de retorno y finalmente las variables locales, que en este caso son un arreglo de char de 32 posiciones y un credential que contiene otros 2 arreglos de char de 32 posiciones cada uno.

Las variables se apilan de manera tal que al leerlas o escribirlas se hace descendiendo por la pila. El problema es que *fgets* puede escribir hasta la cantidad de caracteres que se pasa por parámetro, y en este caso es errónea ya que debería ser *sizeof(user.pass)* en lugar de *sizeof(user)*. Esto permite que puedan escribirse los siguientes 32 bytes de la pila, que en el caso del segundo *fgets*, es el espacio reservado para la contraseña real. Y asi puede sobreescribirse en la pila la verdadera contraseña y engañar al programa.

7.4.a.

Los valores son *admin* para el usuario, y *cualquiercontraseñad32caracterescualquiercontraseñad32caracteres* para la contraseña.

7.5.

7.5.a.

El mecanismo que hace esto posible es el parámetro setUid a la hora de asignar los permisos del ejecutable de este código. Se establece que este programa que puede ser ejecutado por cualquier usuario, tiene permisos para abrir o ejecutar otros archivos como si fuera root.

7.5.b.

El problema está en que *gets* puede sobreescribir la dirección de retorno si el string que se pasa tiene mas de 128 caracteres. Puede controlar todo lo que este apilado debajo de la contraseña.

7.6.

7.6.a.

Si. Si lo que se le pasa a la funcion es un NaN.

7.6.b.

Podría devolver 0 en caso en que se le pase un NaN.

7.8.

7.8.a.

. ; cat /etc/passwd

7.8.b.

Puede pasarsele: ." ; cat "/etc/passwd

7.8.c.

Puede pasarsele: ." && cat "/etc/passwd

7.8.d.

```
void wrapper_ls(const char * dir) {
    pid_t pid = fork();
    if (pid == 0) {
        execve(‘‘ls’’, dir);
    } else {
        int status;
        while (waitpid(pid, &status, 0) == -1) {}
    }
}
```

8. Preguntas de finales

8.1. Marzo 2020

8.1.a. Procesos - Explicar diferencia entre proceso y thread y relación de este último con funciones reentrantes. Explicar qué es un árbol de procesos y cuál es su importancia

Un proceso es un programa en ejecución. El proceso puede invocar tantos threads como necesite, cada thread corresponde a un único proceso. En algunas implementaciones, los recursos de los threads se distribuyen entre los recursos totales de su proceso padre, lo cual esta descripto por el árbol de procesos, y en otras pueden asignarseles recursos independientemente.

Cada proceso se refleja en un bloque de una tabla llamado Process Control Block (PCB), el cual cuenta con la información necesaria para identificar al proceso univocamente (pid) junto con data del proceso que describe el estado del proceso en el momento en que es pausado (estado, program counter, archivos abiertos, registros, stack), para asi poder reanudarlo cuando se desee y resulte transparente al proceso.

En una implementacion con threads se requiere ampliar esta implementacion de la tabla de PCB adjuntando un identificador del thread (tid), el pid del proceso al que pertenece y tambien data que permita determinar el estado de la ejecución, de la misma manera que con los procesos.

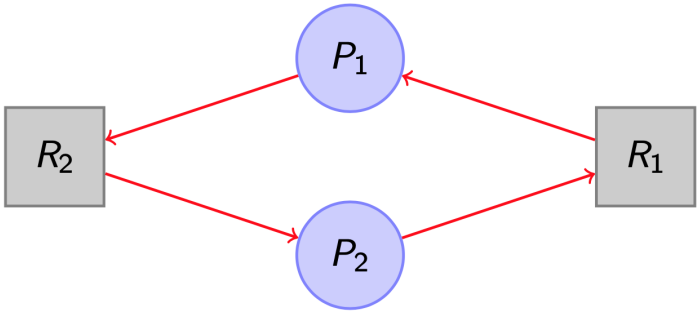
Los threads son una forma de paralelizar la ejecución de un proceso, ya sea para poder tener varias responsabilidades simultaneamente o para dividir un problema en partes y poder atacarlo mas rapidamente.

Una función reentrante es aquella que garantiza que multiples instancias de la misma pueden ejecutarse simultaneamente sin que eso provoque una falla en el sistema. Si una función fuese a ser ejecutada por distintos threads, y no hay garantías de que lo vayan a hacer de manera exclusiva, entonces esta debe ser necesariamente reentrante.

8.1.b. Sincronización - Explicar deadlock con un dibujo y en pocas palabras

Las P son procesos y las R recursos
Arcos:

- De P a R, P requiere R
- De R a P, P adquirió R



8.1.c. Administración de E/S - ¿Por qué los algoritmos de scheduling se miden en cilindros? ¿Está bien? ¿Por qué? Explicar SAN

Se llama *seek time* al tiempo que tarda el brazo de un HDD en transportar los cabezales al cilindro correspondiente. El tiempo de accesso depende mayormente del *seek time* y de la latencia de rotación.

Los algoritmos de scheduling de E/S definen de qué manera se moverá el brazo del disco lo cual determina el orden en que se atenderán los pedidos, en función del número de cilindro de cada uno. Cuanta más distancia recorra el brazo para cumplir los pedidos, mayor será el *seek time* y más tardará en completarlos. Se miden en cilindros porque es la unidad de distancia del movimiento del brazo, cuanto mejor estén ordenados los pedidos, menos cilindros el brazo tendrá que recorrer para servirlos y mejor será el tiempo hasta completarlos.

SAN significa Storage Area Network. Se trata de tener el almacenamiento en la red, pero una red especial, donde los protocolos son específicos para este tipo de datos (son de más bajo nivel).

El problema de tener Network Attached Storage (NAS) es que las operaciones consumen ancho de banda repercutiendo en la comunicación de la red.

8.1.d. Seguridad - Explicar qué es una función de hash segura y ejemplificar dos usos en un sistema operativo. - comparar DAC y MAC

Una función de hash segura es aquella que tiene muy baja probabilidad de generar colisiones y que sea muy difícil de revertir, es decir, sabiendo y encontrar x siendo $f(x) = y$, con f la función de hash.

Discretionary Access Control (DAC): El acceso se controla basandose en las identidades de los usuarios y grupos. Se puede implementar con una matriz donde se almacena para cada sujeto, los distintos permisos sobre cada objeto. Los atributos de seguridad se definen explicitamente (se puede hacer solamente lo que está especificado). Es el usuario dueño del archivo quien determina quiénes tienen acceso al mismo.

Mandatory Access Control (MAC): Cada sujeto tiene un grado (se suele manejar el concepto de label). Los objetos heredan el grado/label del último sujeto que los modificó. Un sujeto sólo puede acceder a objetos de grado igual o menor que el suyo.

8.1.e. Filesystems - Explicar journaling en ext3. ¿Qué problema resuelve?

Cuando el sistema quiere realizar una transacción, en lugar de realizarla inmediatamente, se la escribe en el *journal*. A esto se lo llama realizar un *commit*. Un vez hecho el *commit*, se devuelve el control al proceso usuario mientras el filesystem hace el replay de los commits registrados.

Este otro proceso mantiene un puntero que indica la operación dentro del commit que se está llevando a cabo, el cual va moviendo a medida que va satisfaciendo las operaciones y con ellas, los commits. Este algoritmo permite que ante una falla completa del sistema que requiera un reinicio, se pueda consultar el *journal* para asi reestablecer el sistema mediante el puntero, y continuar con la operación que habia quedado pendiente, asi como con el resto de los commits que aún no habían sido ejecutados.

8.2. Diciembre 2019

8.2.a. Procesos - Una página es compartida por dos procesos. Puede suceder que para un proceso sea de sólo lectura mientras que el otro tenga permitida la escritura?

8.2.b. Sincronización - ¿Cómo podrías sincronizar dos procesos usando IPC? ¿Cuál es la diferencia entre Spin locks y semáforos? ¿Cuándo utilizaría cada uno?

Podrías sincronizarlos usando:

- Memoria compartida
- Algún otro recurso compartido (archivo, base de datos)
- Pasaje de mensaje

Los Spin Locks hacen busy waiting, los semáforos no.

Además, no existe un orden cuando se utiliza un Spin Lock: los procesos continúan compitiendo por entrar a la zona crítica y se puede dar que un proceso nunca pueda acceder ya que el mecanismo se basa en simplemente consultar todos simultáneamente. En cambio, mediante semáforos, los procesos entran en waiting (no hacen espera activa sino que se duermen, esto requiere acceso al kernel) y se añaden a una cola para luego ser desencolados en el momento en que se pueda liberar el lock para dejar entrar a otro (hacer el signal).

8.2.c. Seguridad - Describir setUID y buffer overflow y un ataque que incluya ambos

setUID es un atributo booleano que tienen los archivos que, cuando esta activado, permite que el proceso corra esa rutina con el permiso del owner del archivo. Esto permite realizar distintas acciones que requieren permisos mas elevados, de una manera controlada ya que solo se ejecuta con ese permiso la rutina descripta por el archivo y luego se revocan.

Por ejemplo, si tuviera la necesidad de modificar un archivo con hashes de contraseñas de usuarios, necesitaría permisos de administrador. Gracias a setUID, puede crearse una rutina que modifique ese archivo de la manera controlada en que el diseñador del sistema operativo desee, permitiendo a cualquier proceso invocarla.

Un buffer overflow es un tipo de inyección de código. Es cuando un atacante se aprovecha de un programa que corre una función insegura en términos de tamaños de buffers, como lo es strcpy() que copia dentro del buffer hasta encontrar un byte NULL, haciendo que se escriba tanto contenido en el buffer en memoria que lo sobrepase y sobreescriba memoria por fuera del buffer. De esta forma se pueden sobreescribir las variables que estan almacenadas en el stack y si se sobreescribe un poco mas, se puede modificar la dirección de retorno, logrando que una vez finalizada la función se redirija la ejecución a una porción de memoria controlada por el atacante. De esta forma se termina ejecutando ese código controlado por el atacante, con el ID efectivo del proceso.

Un ataque posible sería realizar un buffer overflow sobre una rutina con setUID activado, y de esta forma conseguir ejecutar código controlado por el atacante, con el UID efectivo del owner del archivo.

8.2.d. E/S - ¿Cómo afectan los dispositivos de memoria de estado sólido a los algoritmos de scheduler de E/S? Definir Storage Area Network (SAN)

Los dispositivos de memoria de estado sólido no tienen problema para leer una posición aleatoria de la memoria como ocurre con los HDD, por lo que suele utilizarse una política FCFS, pero tienen otro problema ocasionado por las características de los semiconductores NAND que los componen: pueden ser leidos y escritos de a páginas (similar a los sectores en HDD) pero no pueden ser sobreescritos. Para lograr eso se deben borrar, lo cual tarda más que escribir y mucho más que leer, y luego escribir. Además, el borrado se hace de a bloques (que tienen varias paginas de tamaño).

Para evitar tener que borrar y así poder ahorrarse el tiempo de hacerlo, lo que se hace es escribir en otra pagina y marcar la vieja como invalida en la FTL (Flash Translation Layer). Cuando ya no hay paginas libres, se busca un bloque inválido (todas las páginas inválidas) para ser borrado y luego escrito. Sin embargo, puede ocurrir que no existan bloques inválidos, pero si bloques con algunas páginas invalidas que no se pueden borrar individualmente porque el borrado se hace de a bloques. Como la lectura y escritura sí es de a páginas, se podrían leer las paginas validas de un bloque y escribirlas en otro para así salvar esas paginas y poder borrar ese bloque (textitgarbage collection). Pero como el problema nace de que el disco este lleno, no hay un lugar a donde salvar esas páginas, por lo que se hace textitover-provisioning: se deja siempre un porcentaje del disco libre para poder salvar esas páginas hasta liberar los bloques.

También ocurre que estos dispositivos se gastan luego de ser borrados muchas veces, por lo que la controladora trata de balancear dónde guarda fisicamente los datos en relación a que tan seguidos son borrados los bloques que los contienen (textitwear leveling).

Estas características de los dispositívos de memoria de estado sólido provocan que se utilize FCFS para requests de lectura pero no para escritura ya que varían de una manera no uniforme que depende de qué tan lleno esté el disco (ahí entran en acción textitgarbage collection y textitover-provisioning).