Indice

Código de Colores

Preguntas de Yovine

Preguntas de Baader

Preguntas de Schapa

Preguntas de Slezak

Procesos

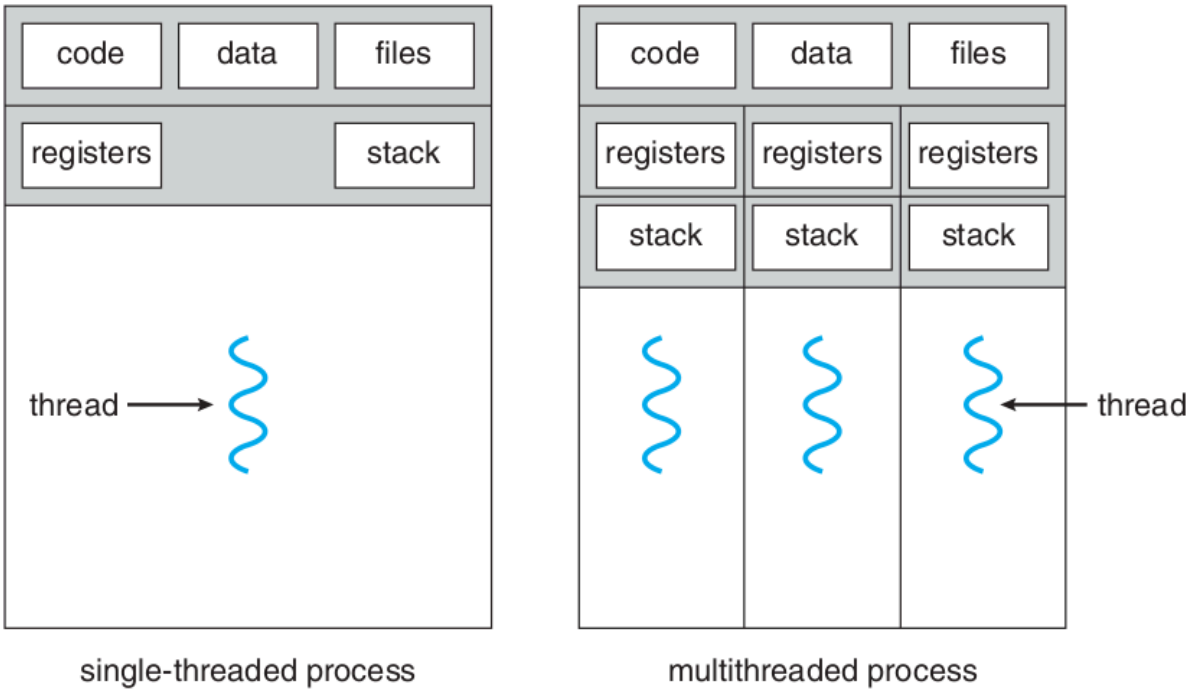
¿Qué es y para qué sirve una System Call? Explicar los pasos involucrados por hardware y software. Ejemplificar. (2017)

Syscall es un tipo de instrucción, son llamadas al sistema. Los pasos son los mismos que para las interrupciones: Sirven para proveer funcionalidad, que solo puede ser accedida por el kernel, a las aplicaciones de usuario.

1. El hardware mete el PC, PSW, registros, etc. a la pila.
2. El hardware carga el nuevo PC del vector de interrupciones.
3. Procedimiento en lenguaje ensamblador guarda los registros.
4. Procedimiento en lenguaje ensamblador establece la nueva pila.
5. El servicio de interrupciones de C se ejecuta.
6. El scheduler decide que proceso se va a ejecutar a continuación.
7. Procedimiento en C regresa al código en ensamblador.
8. Procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Una PCB clásica para el manejo de procesos contiene todos los recursos para que el proceso pueda ejecutar, por ejemplo Registros, Archivos abiertos, etc. ¿Cómo debe modificarse para soportar threads?

En los sistemas operativos que soportan threads, la PCB debe expandirse para incluir información de cada thread individual.
Un thread es la unidad básica de utilización de CPU en un sistema. Se compone de un thread id (tid), un program counter, un set de registros, y un stack.
Otros https://apps4two.com/curso_dba/bibliografia/2-Sistemas%20operativos%20moderno%203ed%20Tanenbaum.pdf como el mapeo de memoria (la sección de código, sección de datos, área de heap), y recursos como archivos abiertos, son del proceso que tiene los threads, y son compartidos entre todos los threads de un mismo proceso.



(SILBERSCHATZ pags 109, 110, 163, 164)

Proponga un escenario en donde un proceso requiera la modificación de algún valor de la PCB. Escriba el pseudocódigo de las rutinas para realizar ese cambio y quién (SO o proceso) es responsable de cada una.

Cuando un proceso escribe algún registro y luego se produce una interrupción de clock y el scheduler decide hacer un cambio de contexto. En este caso se modificará el registro que está guardado en el PCB de ese proceso, por el nuevo valor que se modificó en el proceso.

El proceso únicamente cambia el valor del registro. El SO es el encargado de realizar el cambio en el PCB.

```
changeRegisterPcb(String r,int value,long pid){
        pcbProceso ← obtener el PCB con valor pid de la tabla de procesos
        registroAModificar ← obtener el registro r de pcbProceso
        registroAModifica ← value
        return;
}
```

Cuando abrís un archivo se modifica el File Descriptor Table y por lo tanto, también el PCB.

En la PCB tambien esta el program counter y el stack pointer, asi que con el simple hecho de que el proceso “avance”, se va a modificar la tabla de ese proceso.

En Unix/Linux los procesos no pueden acceder a puertos de entrada/salida directamente, salvo que explícitamente se le dé permiso. ¿Cómo podría implementarse?

When a User Mode process attempts to access an I/O port by means of an in or out instruction, the CPU may need to access an I/O Permission Bitmap stored in the TSS (PCB) to verify whether the process is allowed to address the port.

More precisely, when a process executes an in or out I/O instruction in User Mode, the control unit performs the following operations:

1. It checks the 2-bit IOPL field in the eflags register. If it is set to 3, the control unit executes the I/O instructions. Otherwise, it performs the next check.
2. It accesses the tr register to determine the current TSS, and thus the proper I/O Permission Bitmap.
3. It checks the bit of the I/O Permission Bitmap corresponding to the I/O port specified in the I/O instruction. If it is cleared, the instruction is executed; otherwise, the control unit raises a “General protection” exception.

¿Qué son las funciones reentrantes y cuál es su relación con los threads? (2017)

El término función reentrante hace referencia a que múltiples instancias de la función pueden estar ejecutándose en simultáneo (sin que nada explote). Es decir, se puede "entrar" de nuevo al código antes de haber terminado de ejecutar una instancia anterior. Es un concepto estrechamente relacionado con la sincronización (en particular la exclusión mutua) y el manejo de recursos compartidos. Un ejemplo donde hace falta que el código sea reentrante es cuando hay llamadas recursivas. Además, muchas partes del sistema operativo tienen que ser reentrantes: un ejemplo típico es el código de los drivers.

Relación con threads: si hay chances de que una misma función vaya a ser ejecutada por distintos threads, y no hay garantías de que lo vayan a hacer de manera exclusiva, entonces el código sí o sí tiene que ser reentrante.

Ver: pág 118 de Tanenbaum 4ta edición

Dar dos ejemplos de transición de running a ready. (06/08/2015)

Un ejemplo puede ser en un sistema con un scheduler Round Robin, si se agota el quantum de un proceso, el scheduler lo desaloja, y el proceso que estaba corriendo (running) vuelve a la cola de procesos listos (ready). No pasa a waiting porque el proceso no estaba esperando ningún evento (por ejemplo E/S).

Otro caso podría ser un proceso que decide por cuenta propia dejar de correr para luego ser retomado más adelante. Esto podría tener sentido en sistemas non-preemptive, donde un proceso decide ceder su lugar (quizás para ayudar otros procesos).

Si tenemos un scheduling que maneja prioridades con desalojo (como un SO RT), y una tarea con mayor prioridad aparece, la que se estaba ejecutando pasa de running a ready. Esta tarea de mayor prioridad puede ser una tarea nueva, una evento nuevo de un SO RT (que cree una tarea), como también una tarea en waiting que está lista para volver a correr (porque ya tiene el recurso por el que estaba esperando).

Otro ejemplo más estaría bueno quizás pero no se me ocurre… (interrupciones que no sean la del reloj)

Se puede ir directo de waiting a running? Explicar las razones. (06/08/2015)

Mirando la máquina de estados clásica de los estados de un proceso, la respuesta sería que no se puede. De *waiting* solo se puede pasar a *ready*, y luego de ahí puede pasarse a *running*. Sin embargo, (me suena a que) 8 sistema operativo, y no hay nada que impida que pueda ser diseñado así. Incluso, muchos sistemas operativos modernos implementan más estados que solamente los 5 básicos (new, ready, running, waiting, terminated).
Por ejemplo, podría haber un sistema operativo donde una llamada bloqueante de un proceso pueda ser marcada como “URGENTE” (por ejemplo), de modo que ni bien esté listo el dato, se desaloje la tarea que está corriendo y se reemplace con el proceso que estaba esperando. Dependerá de la implementación del sistema si en el medio el proceso es marcado como *ready* o no. De todos modos, esto podría traer problemas quizás sí muchos procesos usasen este mecanismo.
Podría ser también, que si sólo hay un proceso en el sistema en el momento, y estaba en *waiting* y su dato ya está listo, entonces pase directamente a *running*, dado que no habría ningún otro proceso en *ready* ni *running*.

Cuál es la diferencia entre mode switch y context switch? (06/08/2015)

Context switch se produce al cambiar procesos en ejecución mientras que Mode switch se produce al cambiar de privilegios (por ejemplo, al ejecutar una syscall). La idea es que el Mode Switch está asociado a cuando se cambia de modo user a kernel o viceversa (ver links más abajo, en especial el de Quora).

Context switch implica cargar una nueva entrada de la tabla de procesos y guardar la anterior).

https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/performance/mode_switches.html

What is the difference between a mode switch and a context switch?

Dibujar el mapa de memoria de un proceso. (06/08/2015)

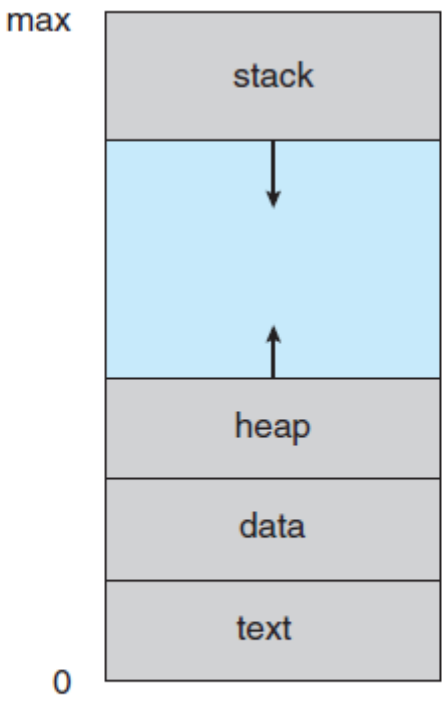


Figure 3.1 Process in memory.

A process is more than the program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's registers.
A process generally also includes the process stack, which contains temporary data (such as function parameters, return addresses, and local variables), and a data section,

which contains global variables. A process may also include a heap,which is memory that is dynamically allocated during process run time.
(Silberschatz pag 106)

Qué pasa con la memoria cuando se crea un proceso hijo? Explicar los casos de fork() y vfork(). (06/08/2015)

En el caso de fork() el proceso hijo apunta a una copia de las páginas de memoria del padre. Aunque en realidad se utiliza la técnica copy-on-write, copiando las páginas sólo cuando sea necesario, para optimizar el uso de memoria.
Por el otro lado vfork() suspende al padre y le da al hijo una copia de las páginas, sin copy-on-write, es decir, cualquier modificación del hijo va a ser vista por el padre al finalizar y retornar el control.

Qué es Copy on Write? Cómo funciona en sistemas con Paginación? (06/08/2015)

Copy on Write es una política de optimización de recursos. Si dos procesos piden recursos que inicialmente son indistinguibles o iguales, se les da un puntero al mismo recurso físico. En el momento en que alguno de los procesos intenta modificar su copia del recurso, el sistema operativo hace la copia auténtica del recurso físico, y asigna esta nueva copia al proceso que intentó modificar (de no hacer esto, los cambios de uno se verían reflejados en los otros también).

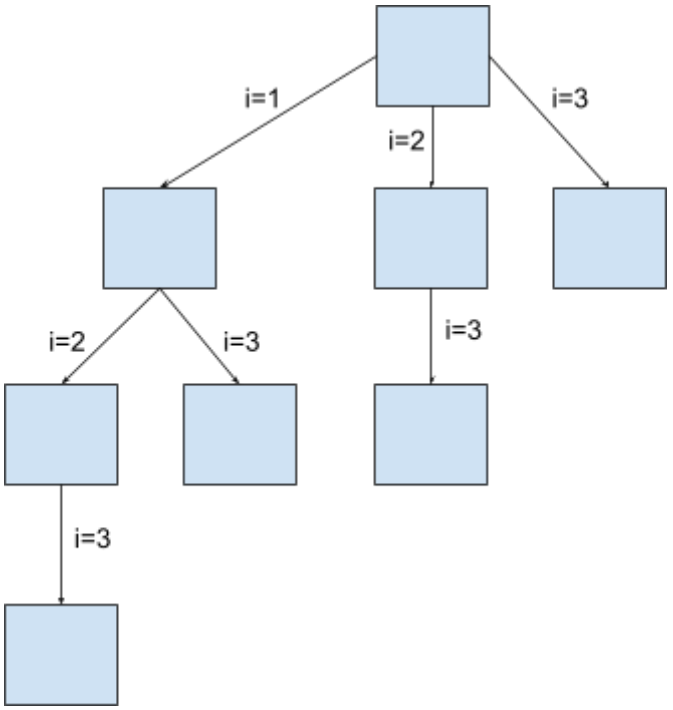
En la paginación puede emplearse esta optimización. Cuando un proceso hace fork() por ejemplo, esto crea una copia completa del proceso (junto con su espacio de memoria). Sin embargo, podría implementarse la política de Copy on Write, de modo que al principio ambos procesos apunten a las mismas páginas de memoria, y sólo en el momento que alguno desea modificar una página es cuando se hace la copia de ella realmente. De esta manera, el fork() se puede hacer más rápido, y la copia de memoria según la necesidad.

Cuántas veces imprime "Hello, world!" el siguiente programa? Explicar. ¿Qué pasa si se usa vfork() en lugar de fork()? (06/08/2015)

```
foo () {
 int i;
 for(i = 0; i<3; i++) fork();
 printf("Hello, world!\n");
}
```

Caso fork():

8 (porque 2^3 = 8). Primero hay 1 proceso, y se hace el primer fork. Ahora hay 2 procesos, a cada uno le quedan dos llamadas más a fork. Cada uno de estos procesos hace fork entonces, dejando un total de 4 procesos con una llamada más a fork. Luego, estos 4 hacen su último fork cada uno, dejando un total de 8 procesos. Luego cada proceso escribe por pantalla.



Caso vfork():
4 porque el primer proceso entra en el for (i=0) hace vfork, se bloquea y el nuevo proceso entra en el fork (como comparten la mem i=1), así dos veces más, bloqueando el proceso que llama al vfork y ejecutando el otro. Al llegar a i=3 el último proceso sale del for (no ejecuta vfork) y al volver el control a los anteriores i ale 3 (porque es la misma memoria, no copias)

Se poseen dos procesadores y se pueden ver los cambios de estado de los procesos. ¿Qué tendría que ver para que se aumente el rendimiento agregando...

...un procesador?

Un caso sería si los procesos pasan mucho de RUNNING a READY. Esto quiere decir que el procesador está desalojando mucho a los procesos en lugar de que se bloqueen solos, lo cual implica que a estos no les está alcanzando el tiempo de CPU que se les asigna. Asumiendo que el quantum no es excesivamente corto, esto quiere decir que hay muchos procesos CPU bound. Luego, tiene sentido agregar un procesador para incrementar el throughput.

Otro caso puede ser si hay muchos procesos READY. Esto implica que el CPU time no está alcanzando para satisfacer la demanda de los procesos, por lo cual se van acumulando en la cola de READY. Agregar un procesador mitigaría esta situación.

En otras palabras: agregar un procesador extra podría hacer que haya menos procesos en ready comparado a la versión sin el procesador extra. En este caso, podemos ver la carga del sistema antes de agregar el procesador, y después ver la carga del sistema luego de agregar el procesador para comparar si disminuyó. (carga del sist ema = cantidad de procesos en ready)

...más memoria?

Los procesos se bloquean mucho. Esto podría deberse a que el d, situación en la que por falta de memoria el SO debe hacer constantemente swapping de páginas entre disco y memoria para poder correr los procesos. Cuando esto pasa cada vez que se quiere ejecutar un proceso sus páginas deben ser traídas a memoria desalojando las de otro proceso, que posteriormente deberá ser traída de vuelta. Esto haría que los procesos se bloqueen todo el tiempo esperando al disco, dejando cada vez menos procesos en READY.

Sincronización

Considere el modelo de memoria compartida. (2/8/2016)

Dé dos primitivas de sincronización.

TAS y CAS (Compare-And-Swap)

Dé implementaciones para cada una de ellas. Explique y justifique todo lo que asuma.

Asumo que se posee un registro atómico SWMR (Single-Writer/Multiple-Reader)

- TAS

```
1  private bool reg;
2
3  atomic bool get() { return reg; }
4
5  atomic void set(bool b) { reg = b; }
6
7  atomic boolean getAndSet(bool b) {
8    bool m = reg;
9    reg = b;
10   return m;
11 }
12
13 atomic boolean testAndSet() { // TAS
14   return getAndSet(true);
15 }
```

- CAS

Compare-and-swap/set

```
1  atomic T compareAndSwap(T u /* expected */,
2                          T v /* update */ ) {
3    T w = register;
4    if (u == w) register = v;
5    return w;
6  }
7
8  atomic bool compareAndSet(T u /* expected */,
9                            T v /* update */ ) {
10   if (u == register) {
11     register = v;
12     return true;
13   }
14   else return false;
15 }
```

Explique ventajas y desventajas de cada una.

No hay mejor o peor, simplemente tienen usos distintos. Ejemplos:

- TAS sirve para hacer TASLock y TTASLock
- CAS tiene número de consenso infinito, a diferencia de TAS que sólo es 2

¿Cómo cambiaría la implementación de los locks distribuidos para sistemas non-preemptive (sistemas sin desalojo)? (2013)

El hint era algo así como pensar en la implementación de los WAIT() y los SIGNAL()
No necesitas que todas las cosas sean atómicas, porque ya no te pueden interrumpir.

Defina condición de carrera (*race condition*).

Se llama condición de carrera a lo que ocurre cuando una ejecución da un resultado erróneo (un resultado que no es equivalente a ninguna ejecución secuencial de los procesos involucrados). Porque el output varía sustancialmente dependiendo de en qué momento se ejecuten las cosas (o del orden en que se ejecuten).

¿Cuáles son las 4 condiciones de Coffman y para qué nos sirven?

- Exclusión mutua: Un recurso no puede estar asignado a más de un proceso.
- Hold and wait: Los procesos que ya tienen algún recurso pueden solicitar otro.
- No preemption: No hay mecanismo compulsivo para quitarle los recursos a un proceso.
- Espera circular: Tiene que haber un ciclo de N≥2 procesos, tal que Pi espera un recurso que tiene Pi+1.

Se deben cumplir todas ellas en un sistema para que exista la posibilidad de deadlock. Por ello, nos sirven para determinar si un sistema está expuesto a deadlock o no.
Es decir, alcanza con que en un sistema no se cumpla alguna de estas condiciones para que esté libre de deadlock. Algunos mecanismos para prevenir deadlock se basan en tratar de eliminar la posibilidad de que se cumpla alguna de las condiciones.

¿Por qué es necesaria la primitiva *Test & Set* para la implementación de semáforos?

It is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute *wait()* and *signal()* operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the *wait()* and *signal()* operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are re-enabled and the scheduler can regain control.
In a multiprocessor environment, interrupts must be disabled on every processor. Otherwise, instructions from different processes (running on different processors) may be in some arbitrary way. Disabling interrupts on every processor can be a difficult task and furthermore can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques— such as compare and swap() or spinlocks*—to ensure that wait() and signal() are performed atomically.
(Silberschatz pag. 275 10ma. edición)

***Spinlocks son los objetos lock que utilizan TAS para su implementación.¿Alcanza con la primitiva T&S en arquitecturas multiprocesador? En caso afirmativo justificar. En caso negativo, proponer una solución.**
Si, ya que alcanza con poder tomar un mutex para modificar la variable del semáforo. Un mutex puede ser implementado usando solo TAS.

En caso de poseer compare-and-swap se puede hacer más “fácilmente” ya que se tiene acceso exclusivo a la variable y se modifica a la vez.

Desarrollar ReentrantLock que soporte locks recursivos. (16/11/2017)

Esquema de implementación

```
1  int calls;
2  atomic<int> owner;
3
4  void create() { owner.set(-1); calls = 0; }
5
6  void lock() {
7    if (owner.get() != self) {
8      while (owner.compareAndSwap(-1, self) != self) {}
9    }
10   // owner == self
11   calls++;
12 }
13
14 void unlock() { }if (--calls == 0) owner.set(-1); }
```

Explicar por qué la implementación de una cola FIFO no es correcta. (16/11/2017)

```
atomic<int> tail, head = 0;
int capacidad = N;
T[] items = new T[N];
void queue(x){
  assert(!full());
  items[tail.getAndInc() % capacidad] = x;
}
T dequeue(){
  assert(!empty());
  T x = items[head.getAndInc() % capacidad];
  return x;
}
bool full(){return tail.get() - head.get() == capacity};
bool empty(){return tail.get() - head.get() == 0};
```

Porque tanto queue como dequeue no son atómicas, lo que podría generar race conditions.

Supongamos que tenemos dos procesos P0 y P1, ambos quieren agregar un item a la cola (B y C respectivamente); ésta tiene un elemento A, ejemplo: tail = 1, head = 0, items=[A] y su capacidad es de dos. Si se ejecuta el orden de las instrucciones de la siguiente manera:

1. P0 se asegura que la cola no está llena
2. P1 se asegura que la cola no está llena
3. P1 agrega su elemento a la cola. Nadie estaba usando tail así que getAndInc hace su trabajo, tail = 2, head = 0, items = [A,C]
4. P0 agrega su elemento a la cola. De nuevo getAndInc es llamado así que tail = 3, head = 0, items = [B,C]

Tenemos más elementos que nuestra capacidad, además de que perdimos uno y en el próximo llamado a dequeue se van a devolver en orden incorrecto.

Una forma de solucionar esto es usando un mutex para agregar/remover elementos de manera atómica.

Código común

```
1  bag<T> buffer; // requiere exclusión mutua interna
2  semaphore filled = 0; // cantidad de items en buffer
3  semaphore empty = N;  // lugares libres en el buffer
```

Productor y consumidor

```
1  void productor() {
2    while (true) {
3      T item = produce();
4      // ¿Hay lugar?
5      empty.wait();
6      // Agregar al buffer
7      buffer.put(item);
8      // Avisar que hay algo
9      filled.signal();
10   }
11 }
```

```
1  void consumidor() {
2    while (true) {
3      // ¿Hay algo?
4      filled.wait();
5      // Sacar del buffer
6      buffer.get(&item);
7      // Avisar que hay lugar
8      empty.signal();
9      consumir(item);
10   }
11 }
```

Defina la operación atómica TestAndSet (TAS) (03/08/2017)

Objeto atómico básico *get/test-and-set*

Operaciones indivisibles no bloqueantes (wait-free) ⚠

```
1  private bool reg;
2
3  atomic bool get() { return reg; }
4
5  atomic void set(bool b) { reg = b; }
6
7  atomic boolean getAndSet(bool b) {
8    bool m = reg;
9    reg = b;
10   return m;
11 }
12
13 atomic boolean testAndSet() { // TAS
14   return getAndSet(true);
15 }
```

Provea dos implementaciones de un objeto lock basadas en TAS y argumente sobre sus ventajas y desventajas. (03/08/2017)

- TASLock:

Spin lock (TASLock)

```
1  atomic<bool> reg;
2
3  void create() {
4    reg.set(false);
5  }
6
7  void lock() {
8    while (reg.testAndSet()) {}
9  }
10
11 void unlock() {
12   reg.set(false);
13 }
```

  - Se implementa en user mode
  - lock() introduce espera no acotada y NO es atómico.
  - NO hay que olvidarse de hacer unlock()
  - Hace busy waiting

- TTASLock:

Spin lock (TTASLock)

```
1  void lock() {
2    while (true) {
3      while (reg.get()) {} // espera activa
4      if ( ! reg.testAndSet() ) return;
5    }
6  }
```

  - Se intenta no iterar sobre testAndSet
  - Hay más caché hits usando TTASLock

Cuál es el número mínimo de registros atómicos RW necesarios para garantizar exclusión mutua para n procesos. (03/08/2017)

Aca iba algo?
Con un registro estás, mirar la implementación de locking de la pregunta anterior.

Es posible hacerlo con menos registros si se asume que el tiempo que tarda ejecutar una operación de lectura o escritura sobre un registro atómico está acotado por una constante δ conocida? Justifique.

Teorema de Burns & Lynch: *"No se puede garantizar EXCL y LOCK-FREEDOM con menos de n registros RW*" (Salvo que se supongan restricciones de tiempo).

Suponiendo restricciones de tiempo, según el algoritmo de Fischer, y asumiendo FAIRNESS es posible garantizar LOCK-FREEDOM si $w > T$.

```
TRY:
waitfor(turn==0);
turn = i;                    // tarda a lo sumo T
pause(w);              // espera un tiempo w > T
if( turn!=i ) {
    goto TRY;
}
CRIT:
…

EXIT:
turn = 0
```

Qué es la instrucción Test and Set? Para qué sirve? Justificar. (06/08/2015)

Es una primitiva de sincronización que de forma atómica obtiene el valor booleano de una variable (es decir, a bajo nivel un registro o una posición de memoria), y al mismo tiempo la setea en true.

Dar dos soluciones (al menos una correcta) para el problema de la cena de los filósofos. Compararlas. (06/08/2015)

Lo importante es que, según Lynch, no hay solución correcta en la que todos los filósofos actúen igual, o en donde todos los tenedores sean iguales (o sea, soluciones simétricas). Las soluciones correctas deben funcionar de forma asimétrica (o sea, que los filósofos hagan cosas distintas, o que los tenedores sean distinguibles unos de los otros)

1. Enumerar los recursos (Tenedores) o procesos (filósofos).
    a. Se enumeran los recursos de manera consecutiva, y cada filósofo intenta levantar primero el tenedor de mayor número. De esa manera no hay deadlock, ya que no puede haber espera circular.
    b. Otra forma parecida es enumerar a los filósofos, y que los filósofos pares levanten primero el tenedor izquierdo y después el derecho. Es similar, pero levemente más optimizado que la anterior solución.
2. Utilizar un árbitro (un mozo). Entonces, cada filósofo le pide permiso al mozo antes de levantar sus tenedores, y el mozo analiza si esto provocará deadlock (le da permiso solo si sabe que va a poder levantar ambos tenedores). Funciona, pero no es eficiente porque el mozo es cuello de botella, y no permite mucho paralelismo.
3. Utilizar envío de mensajes entre los filósofos y se clasifican a los tenedores como “sucio” o “limpio”.

Dos procesos, P0 y P1, ejecutan los siguientes programas: (06/08/2015)

P0	P1

`while(1) {`
`  flag[0] = 1;`
`  waitfor(flag[1] == 0);`
`  /*Sección Crítica*/`
`  flag[0] = 0;`
`}`
	`while(1) {`
	`  flag[1] = 1;`
	`  while(flag[0] == 1) {`
	`  flag[1] = 0;`
	`  waitfor(flag[0] == 0);`
	`  flag[1] = 1;`
	`  }`
	`  /*Sección Crítica*/`
	`  flag[1] = 0;`
	`}`

Hay exclusión mutua? Justificar.

Si, no puede pasar que ambos estén en la sección crítica. Si P0 está en CRIT, flag[0] = 1 y en algún momento anterior flag[1] = 0, por lo que P1 no hubiera pasado el waitfor. Si, en cambio P1 está en CRIT, flag[1] = 1 y en algún momento anterior flag[0] = 1, con la misma lógica, P0 se quedó en el waitfor.

Están libres de deadlock? Justificar.

Si P0 espera a P1, lo hace con flag[0] = 1 y P1 no modifica ese registro. Además, P1 togglea flag[1], y solo espera a que flag[0] = 1 con flag[1] = 0, que le da permiso a P0 de continuar, hasta llegar a flag[0] = 0. Por lo que siempre que P0 espera a P1, va a poder avanzar. Está libre de deadlock.

Hay inanición de algún proceso? Justificar.

Si, puede haber inanición de P1. Si siempre las instrucciones, última y primera de P0 ocurren en el mismo quantum, P1 nunca va a poder avanzar, porque flag[0] no cambiaría de estado.

Si P1 empieza a ejecutar el código de P0, cambiando flag[0] por flag[1], qué ocurriría?

(Entiendo que también cambia flag[1] por flag[0], si no, no se cumpliría EXCL)
Habría deadlock.

Justificar si la primitiva de sincronización propuesta para cada caso es correcta, y sino cuál usaría y por qué:

Acceso exclusivo a disco utilizando TAS Lock.

No es correcta. TAS Lock hace busy waiting, lo cual implica que el proceso se mantiene ciclando hasta que logra obtener el lock. Hacer esto significa un desperdicio de tiempo de CPU dado que mientras cicla el procesador no hace ningún trabajo real. Debido a esto, no es conveniente usar un TAS Lock cuando el lock se mantiene por largos períodos de tiempo, como en el caso de hace E/S a disco. El disco es un dispositivo muy lento en términos del CPU, por lo cual los procesos que se encuentren haciendo busy waiting a la espera de que se libere el TAS Lock desperdiciarán mucho tiempo de procesamiento.

En este caso es mejor usar un mutex implementado mediante un semáforo, ya que el SO bloqueará a los procesos que fallen en obtener el acceso a este y los desbloqueará a medida que se vaya liberando. Esto permite que el SO ponga a ejecutar otras tareas mientras el proceso que tiene el lock realiza el acceso a disco. Los procesos esperando por el lock no hacen busy waiting sino que permanecen bloqueados hasta que se libera el lock, momento en el cual el SO despierta a alguno de ellos.

Acceso a una estructura que permite hasta 3 accesos simultáneos con semáforos.

Es correcta. Los semáforos tienen un valor que puede ser decrementado e incrementado con las operaciones wait y signal. Si el valor del semáforo es menor o igual a 0, un proceso que hace wait se bloquea y queda en espera. Cuando otro proceso hace signal, el SO incrementa el valor del semáforo y despierta a uno de los procesos en espera. Este último continúa su ejecución decrementando el valor del semáforo antes de hacerlo.

Con esto, se puede proteger el acceso a la estructura con el uso de un semáforo cuyo valor inicial es 3. Los procesos hacen wait antes de acceder a ella y hacen signal una vez que terminaron. De esta forma, sólo 3 procesos podrán obtener el acceso mientras el resto queda en espera. A medida que estos van haciendo signal, se van despertando los procesos en espera uno por uno.

Acceso a un contador que se desea incrementar mediante un semáforo binario.

No es correcta. Las operaciones de los semáforos se implementan mediante system calls, lo cual implica que se debe realizar un cambio de contexto cada vez que son realizadas. Esto tiene un overhead que no se justifica para un acceso exclusivo de tan corto tiempo como lo es incrementar un contador.

Para esta situación es mejor usar una variable numérica atómica que puede ser incrementada con una única instrucción atómica del procesador. Esto es muy eficiente, pero requiere soporte del hardware. A falta del mismo, la alternativa es usar un TAS Lock ya que no requiere cambiar el contexto y el hecho de que el lock se mantenga por un tiempo corto implica que no se perderán tantos ciclos haciendo busy waiting.

Memoria

Describa de qué manera el Sistema Operativo protege el espacio de memoria de un proceso. (2017)

Depende de la abstracción de memoria que utilice el SO. Los SO actuales utilizan memoria virtual como método para que cada programa tenga su propio espacio de direcciones (dividido en páginas). Cuando un programa hace referencia a una dirección que no tiene asociada, la MMU (Memory Managment Unit) que se encarga de traducir las direcciones virtuales a direcciones físicas hace que la CPU haga un trap al SO (pagefault) y la rutina del kernel correspondiente termina el proceso (violación de segmento).

Tanenbaum, pags. 188, 189, 190, 191

¿Que es Thrashing y como se puede solucionar sin agregar hardware? (2017)

Thrashing: situación en la que el SO pasa más tiempo swappeando páginas que la CPU ejecutando procesos.

Soluciones:

- Mediante un algoritmo de reemplazo local a cada proceso: cuando un proceso necesita cargar más páginas en memoria solo puede desalojar sus propias páginas.

¿Cómo es la protección de la memoria para los procesos, usando paginación? (2013)

Hay que asignarles una tabla de páginas distinta a cada proceso. Se hace poniéndola en un registro especial, que la MMU levanta cuando quiere transformar la dirección virtual a física. Así un proceso no puede ver el espacio de direcciones de otro.

Defina thrashing y fragmentación, y enuncie para cada caso una solución, con sus ventajas y desventajas. (03/08/2017)

Thrashing: situación en la que el SO pasa más tiempo swappeando páginas que la CPU ejecutando procesos.
Soluciones:

- Mediante un algoritmo de reemplazo local a cada proceso. Puede ser que no sea bien utilizado por los procesos.6
- Agregando memoria. Es costoso.

Fragmentación: sucede al quedar espacios de memoria pequeños, inutilizables. Es un problema porque podría generar situaciones donde tenemos suficiente memoria para atender una solicitud, pero no es continua. Existen 2 tipos de fragmentación:

- Externa: bloques libres pero pequeños y dispersos. Soluciones:
    - Utilizar paginación. Es más difícil de implementar que la memoria llana o segmentos.
    - Reacomodar cada cierto tiempo los datos. Es costoso en operaciones y tiempo
- Interna: espacio desperdiciado dentro de los propios bloques. Por lo general se da cuando el tamaño de página es muy grande en relación al tamaño del proceso (sección de texto, de datos, pila, etc.). Soluciones:
    - Utilizar segmentos de tamaños variables, para ajustarse mejor a los pedidos. Esto puede llevar a fragmentación externa.

Describir los atributos de las páginas y para qué se usan. ¿Es posible que haya páginas compartidas? ¿En qué caso?

Atributos:

- Valid/invalid: indica si la página está en el espacio de direcciones del proceso. Se usa para definir el espacio de direcciones. En paginación se puede utilizar para indicar si la página se encuentra en memoria o si es necesario cargarla del disco.

- Read/write: indica si la página es de solo-lectura o lectura-escritura. Se usa como mecanismo de protección para evitar escrituras no deseadas. También puede ser utilizado para implementar copy-on-write: la página se marca como solo-lectura y cuando es escrita el handler de este error hace una copia de la página. Es necesario guardar información adicional que sirva para distinguir entre errores por escritura de una página que es realmente solo-lectura y otra que usa copy-on-write.

- Execute: indica si la página es ejecutable. Sirve como mecanismo de protección para evitar que se ejecuten páginas de datos que podrían ser escritas por un atacante (por ejemplo, el stack en un ataque de buffer overflow).

- Dirección del marco de página al cual está mapeada

- Dirty: indica si la página fue escrita. Se usa para determinar si una página debe ser bajada a disco si se la quiere desalojar de la memoria. También sirve para implementar algoritmos de desalojo de páginas (como second-chance).

- Referenced: indica si la página fue accedida (leída o escrita). Se utiliza para algoritmos de desalojo de páginas (como second-chance).

- Locked: indica si la página puede ser desalojada. Sirve para que una página que está esperando a ser escrita directamente por un dispositivo de E/S (por ejemplo, usando DMA) no sea desalojada. Se puede usar también para mantener en memoria páginas que son usadas frecuentemente o páginas del kernel si este no soporta page faults de su memoria virtual. Otro uso es en los algoritmos de desalojo. Si un proceso trae una página a disco pero aún no resume su ejecución otro proceso podría desalojar esa página antes de que sea utilizada. Con el bit de lock se puede evitar esto.

Pueden haber páginas compartidas cuando páginas de varios procesos mapean a un mismo frame. Esto sirve por ejemplo para implementar memoria compartida, copy-on-write y bibliotecas compartidas.

Scheduling

¿Qué desafíos enfrentan los algoritmos de scheduling para sistemas con varios procesadores? (2017)

Uno de los problemas que enfrentan los algoritmos de scheduling al haber varios procesadores, es que cada uno de los procesadores tiene su memoria caché, y si uno de los procesos es ejecutado en un cpu para luego ser ejecutado en otro, no hay cache hits y se hace más lento (siempre teniendo en cuenta scheduling con desalojo). Una mitigación para esto es utilizar el concepto de afinidad al procesador, es decir, tratar de que un proceso siempre utilice el mismo procesador, aunque se tarde un poco más de tiempo en obtenerlo. Dentro de afinidad al procesador, esta puede ser afinidad dura, es decir que un proceso siempre se va a ejecutar en el mismo procesador, o afinidad blanda, donde un proceso trata de ejecutarse en el mismo procesador pero podría llegar a cambiar y ejecutarse en otro.
Otro desafío a tener en cuenta es el balance de carga entre los procesadores, para esto se usa push y pull migration. En push migration, una tarea se encarga de revisar las cargas de cada uno de los procesadores y mueve procesos de una cola a otro para balancearlos. Pull migration es cuando un procesador en estado idle le saca un proceso de la cola a otro procesador para ejecutarlo.

¿Qué es el problema de inversión de prioridades y qué schedulers afecta? (2017)

Es un problema que se da en sistemas multitarea con schedulings de prioridades, cuando una tarea de alta prioridad es indirectamente desalojada por una de menor prioridad, efectivamente invirtiendo las prioridades relativas entre ambas.
Por ejemplo, si hay tres procesos L, M y H, y sus prioridades cumplen que L < M < H. Si se da que L está usando el recurso R, y H lo necesita, entonces H va a esperar a que L lo libere. Si luego entra M y desaloja a L, entonces está indirectamente perjudicando a H. Una posible solución es heredar las prioridades, ie mientras H espera a L, L hereda la prioridad de H y ejecuta a las chapas.

Teniendo un algoritmo de scheduling al que los procesos le pueden pedir que se le agregue un quantum en determinado momento, y el scheduler te lo da sólo si eso ayuda al rendimiento total del sistema. ¿En que se tendría que fijar el scheduler para decidir si le corresponde quantum o no? (2013)

Si hay más procesos en ready, si es probable que el proceso termine con ese quantum extra (si hay algún otro esperando que este termine para empezar), si durante ese quantum va a empezar a hacer entrada/salida…

Suponga un sistema operativo que utiliza RR (round robin) como scheduler, donde cada entrada de la cola de procesos es un puntero a la PCB.

¿Es posible implementar un sistema de asignación de prioridades (i.e. darle más tiempo de procesamiento) a procesos manteniendo RR como scheduler? Justificar.

Es posible implementar un sistema de round robin con prioridades. Podríamos imaginarlo como que cada prioridad tiene una cola distinta y se hace round robin entre los procesos de mayor prioridad hasta que no hay ninguno y se pasa a la siguiente prioridad. Notar que este sistema de scheduling puede causar starvation.

Esta implementación, ¿soporta scheduling de threads? Si la respuesta es afirmativa, muestre un ejemplo. Si la respuesta es negativa, proponga una modificación que lo permita.

El problema en este caso es que cada entrada de la cola de procesos tiene un puntero a la entrada de la PCB. Al utilizar scheduling de threads, se expande la PCB para que cada entrada incluya la información de los threads. Con esta implementación, el scheduler no podría saber cual de los threads es el que tiene que ejecutar. Podría modificarse la cola para que cada entrada contenga una tupla con un puntero a la pcb y el id del thread que debe ejecutarse.

¿Puede haber inanición en MFQ (Multilevel Feedback Queue)? En caso afirmativo dar un ejemplo. En caso negativo, indicar el mecanismo que lo evita.

Para que no haya inanición en una MFO debe agregarse un sistema de aging para que un proceso que pasó mucho tiempo en una cola de alta prioridad pase a una de baja prioridad.

Silberschatz 7ma edición, pág 168: "*The **multilevel feedback-queue scheduling algorithm**, in contrast, allows a process to move between queues. The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging **prevents starvation**.*"

Defina la propiedad de carga. (03/08/2017)

Es la cantidad de procesos en estado "ready" en el sistema.

Seguridad

¿Para qué sirve setuid y cuales son sus riesgos? (2017)

When a process does an exec on a file with the SETUID or SETGID bit on, it acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch, since the rights available change.

Files located in user directories, but which are owned by the superuser and have the SETUID bit on, are potential security problems because such files acquire the powers of the superuser when executed by any user.

Ver: Tanenbaum págs 604, 314, también hay varios ejemplos en la parte de seguridad que usan SETUID y una posible solución a los riesgos.

Explicar que es el bit de SETUID. Dar un ejemplo de uso. (16/11/2017)

When a process does an exec on a file with the SETUID or SETGID bit on, it acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch, since the rights available change.

Ejemplo: `sudo` utiliza SETUID para que un usuario pueda ejecutar binarios como root.

Otro ejemplo de uso es el proceso que permite realizar un cambio de contraseña. Como por supuesto ningún usuario común debería poder tocar este tipo de información, es necesario que de alguna manera pueda acceder a cambiar su propia contraseña. Por eso este proceso, cuyo dueño es el super usuario, posee el bit setuid prendido, pudiendo acceder a modificar estos datos aunque sea ejecutado por un usuario común.

Explicar DAC y comparar DAC en unix y windows. (2017)

The main difference is that traditional UNIX access control is much more "coarse", because an administrator can only specify access for three broad sets of people on UNIX (the owner, the group specified at file creation time, and "others").

In Windows, by contrast, you can specify permissions for individual users, individual groups or any combination of the two. The permissions are also less coarse on Windows as well.

(https://www.quora.com/What-are-the--betweens-DACs-of-Windows-and-UNIX-access-control)

La diferencia entre DAC (Discretionary Access Control) y MAC (Mandatory Access Control) es que en el primero el dueño del archivo puede especificar quiénes tienen acceso, mientras que en el segundo esto se configura automáticamente en función del nivel de prioridad que tiene el último usuario que lo modificó.

¿Se puede considerar al deadlock como un problema de seguridad? (2013)

Si, con el ataque de "denegación de servicios" (DoS) te pueden atacar por ese lado.
Es decir, si algún atacante conoce una manera en la cual puede producir un deadlock en un sistema, al producirlo estaría “trabando” varias partes del sistema y dejando fuera de correcta ejecución las mismas, y esto puede verse como que el sistema está negado de servicios.

Explicar una API general para una encriptación asimétrica y las características generales de la implementación. (2013)

Mensaje m, una clave k_1 y otra clave k_2.
$D(E(m, k_1), k_2) = m = D(E(m, k_2), k_1)$

Dé las definiciones de algoritmo de encriptación simétrico y asimétrico. (03/08/2017)

Algoritmos de encriptación

- Simétricos: son aquellos que utilizan la misma clave para encriptar y desencriptar. Ej: Caesar, DES.
- Asimétricos: usan claves distintas. El más famoso: RSA

Detalle RSA. (03/08/2017)

Tiene dos claves numéricas de muchos dígitos. Una es clave pública y la otra privada. Cada persona necesita su clave privada (que protege) y su clave pública (que difunde). Para encriptar un mensaje se utiliza la clave pública del receptor. Para descifrarlo es necesaria la clave privada del mismo.

Detalle:

Tomemos p y q primos (de 200 dígitos aprox).

Multipliquémoslos: $n = pq$.

Calculemos también: $n' = (p-1)(q-1)$.

Elijámos un entero e que esté entre 2 y $n'-1$ y que sea *coprimo* con n'.

¿Qué era ser coprimo? Significa no tener factores comunes.

e y n van a ser nuestra clave de encripción (pública).

Computamos d para que cumpla que el resto de $d.e/n' = 1$ (es fácil de hacer).

d y n van a ser nuestra clave de desencripción (privada).

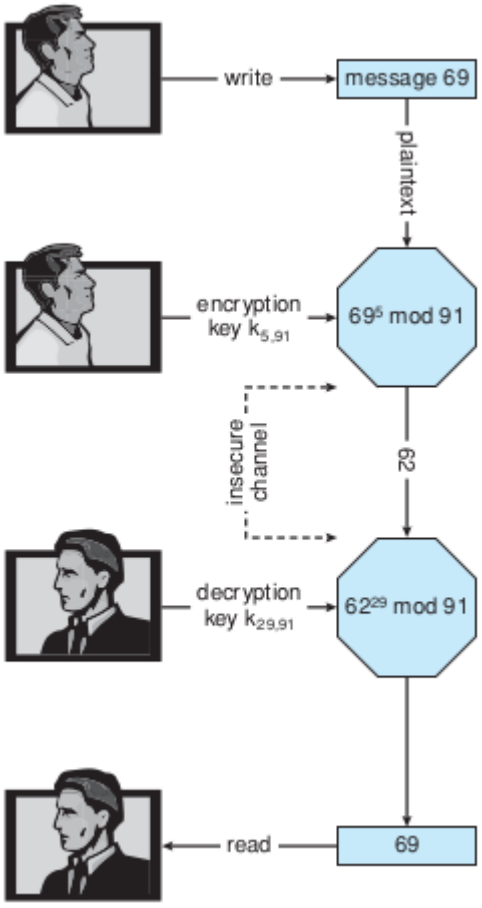
¿Cómo encripto? Para cada letra m calculo el resto de dividir m^e por n.

¿Cómo desencripto? Para cada letra encriptada c calculo el resto de dividir c^d por n.

Lo bueno es que la clave pública la puedo publicar en el diario. El método funciona porque factorizar es muy difícil (NP), aún para las computadoras más potentes.

Si se pudiese factorizar fácilmente, el método no serviría.

An example using small values is shown in Figure 15.8. In this example, we make ***p = 7*** *and* ***q = 13****. We then calculate* ***N = 7∗13 = 91*** *and* ***(p−1)(q −1) = 72****. We next select* $\boldsymbol{k_e}$ ***relatively prime to 72 and < 72****, yielding 5. Finally, we calculate* $\boldsymbol{k_d}$ ***such that*** $\boldsymbol{k_e\ k_d}$ ***mod 72 = 1****, yielding 29. We now have our keys:* ***the public key,*** $\boldsymbol{k_e}$***,N = 5, 91****, and the* ***private key,*** $\boldsymbol{k_d}$***,N = 29, 91****. Encrypting the message 69 with the public key results in the message 62, which is then decoded by the receiver via the private key.*



Explique firma digital con RSA. (03/08/2017)

¿Qué es el Hash de una vía? Dar 2 usos y justificar. (06/08/2015)

Una función de hash de una vía es un hash por el cual no se puede obtener la preimagen a partir del valor del hash. Además le vamos a pedir que sea libre de colisiones, es decir que tampoco sea fácil encontrar otro input de la función que arroje los mismos resultados.
Un uso es para almacenar contraseñas ya que no es deseable poder obtener el valor de la contraseña a partir del hash.
Otro uso posible es mostrar la integridad de un mensaje como por ejemplo en el algoritmo HMAC que las usa. Al pasar el mensaje por una función de una vía, y enviar ambos resultados, si los receptores conocen la función, pueden verificar que el mensaje no ha sido alterado.

¿Qué problemas de seguridad solucionan los stack canaries con respecto al buffer overflow? Indicar desventajas

Los stack canaries se usan para tratar de detectar la sobreescritura de la dirección de retorno en el stack por un buffer overflow. Son un valor generado aleatoriamente con cada
ejecución de un programa y que se pushea al stack luego de la dirección de retorno, entre esta y el buffer que puede ser atacado. Luego, antes de retornar de cada llamado a una función, el programa verifica que el stack canary no haya sido modificado. Si lo fue, hubo un buffer overflow y se debe abortar la ejecución sin retornar.

Las desventajas de los stack canaries son:
- Dado que son generados por el compilador, sólo sirven para programas que son compilados con ellos. No sirven para proteger programas que no fueron compilados con stack canaries, a diferencia de técnicas como stack randomization.
- El valor del stack canary suele ser definido al iniciar la ejecución del programa y mantenerse fijo. Cuando un proceso hace un fork, el proceso hijo hereda el valor del stack canary. Esto puede dar lugar a ataques de fuerza bruta sobre el canary. Por ejemplo, un proceso de un servidor web puede forkear un proceso para atender a cada request entrante. Si el atacante puede hacer un buffer overflow con este proceso hijo, puede sobrescribir byte por byte el stack canary y ver si el proceso aborta o no para adivinar el valor.
- Dado que el programa debe pushear y verificar el stack canary en cada llamado a función, y que los programas suelen tener una cantidad importante de llamados a funciones durante su ejecución, el rendimiento del programa puede ser afectado.

No se si es desventaja de usar stack canaries, pero el hecho de usarlo no implica que dejen de existir los buffer overflow. En el tanenbaum hay un ejemplo donde se saltea la modificación del stack canary y aún así puede modificar la dirección de retorno.

Filesystems

¿Qué es el superbloque en ext2 y que pasa si se pierde su información? (2017)

El superbloque contiene metadatos críticos del sistema de archivos tales como:

- Información acerca del tamaño
- Cantidad de espacio libre
- Dónde se encuentran los datos

Si el superbloque es dañado, y su información se pierde, no podría determinar qué partes del sistema de archivos contienen información.

¿Cuál es la diferencia a nivel de operaciones, de cuando se hace ls y ls -l? (2017)

`ls` solo debe aFcceder al inodo del directorio para listar los nombres de los archivos. `ls -l`, en cambio, accede al inodo de cada archivo para listar sus metadatos.

Comparar FAT vs INODOS para la facilidad de hacer un backup total de un disco. Lo mismo para backup incremental. (2013)

Para el backup total conviene FAT, porque se puede ver rápidamente en la FAT los bloques ocupados de disco. Con inodos tendrías que mirarlos todos.
Para backup incremental conviene inodos, porque te dice para cada archivo que cambios hubo en l a metadata, y así sabés si cambió o no, con la FAT tendrías que buscarlos y entrar a cada uno para fijarte.

Administración de E/S

Teniendo cuatro discos en un RAID-5: (2013)

¿Hasta cuántos discos caídos podés recuperar con el resto?

Sólo 1 (chequeando las paridades del resto).

¿Hay algún disco que te genera más problemas que el resto si se te rompe?

No, son todos iguales.

¿Hace falta hacer copia de seguridad de los datos?

Sí, si borras algo accidentalmente o cambias algo mal en el RAID, se necesita una copia de seguridad. Lo mismo si ocurren errores de software o hardware, el esquema RAID no provee protección contra modificar o eliminar algo accidentalmente, solo provee redundancia.

Explicar ventajas y desventajas de 3 estrategias de E/S a disco. (Se refiere a algoritmos de scheduling de disco) (06/08/2015)

FIFO

Ventajas
- Es el más simple

Desventajas
- La cabeza se mueve de un lado a otro de forma no optimizada ej: cilindros 20,200,10

SSTF (shortest seek time first)

Ventajas
- Atiende el próximo pedido más cercano, de esta forma la cabeza se mueve de forma más optimizada, en el ejemplo anterior si la cabeza se encuentra en 0 hace 10,20,200

Desventajas
- Puede causar starvation

Scan (ascensor)
Ventajas

- Puede mejorar los tiempos de FIFO ya que se mueve primero hasta el final en un sentido y luego en el otro y no tiene starvation como en SSTF

Desventajas

- Los tiempos de espera no son uniformes
- Puede llegar una solicitud para un cilindro inmediatamente anterior y hay que esperar a que se cambie de sentido

Dado un disco de 100 cilindros, numerados de 0 a 99. Determine el orden de visita de los cilindros para una de las estrategias anteriores. Considere que desplazar el cabezal de un cilindro a uno consecutivo toma una unidad de tiempo y que se empieza en el cilindro 0. (06/08/2015)

Tiempo	0	10	20	80	90	100
Cilindro	19	65	17	55	2	90

(Los valores no son los del examen porque no los recuerdo, pero el ejercicio era de este tipo.)

Sistemas distribuidos

Problema sobre un algoritmo de commit distribuido (2017)

un algoritmo de commit distribuido funciona de la siguiente manera: opera sobre una red donde los paquetes pueden perderse o demorarse, y cuando un nodo quiere escribir sobre cierto archivo que está replicado en todos los nodos, envía un pedido de escritura. Si recibe confirmación de todos los nodos, escribe y le notifica a los demás que pueden hacer lo propio. Alguien nota que este algoritmo puede fallar si los nodos se caen entre la escritura y la notificación y propone para solucionarlo el envío de mensajes de confirmación de parte de los nodos. ¿Este algoritmo resuelve el problema? Justifique. s

Lo que se propone para solucionar el problema es pasar de un two-phase commit (2PC) a un three-phase commit (3PC) y si, esto soluciona el problema. El problema se da si se cae el coordinador del commit antes del envío de la notificación. Los nodos se quedarían bloqueados esperando un mensaje. Este problema se soluciona en 3PC al agregar un estado intermedio de “preparado". El coordinador no envía el commit hasta que sepa que los nodos están preparados, esto elimina el problema de que algún nodo complete la transacción antes de que todos los nodos estén al tanto de la decisión.

Algoritmo para balancear la carga entre procesadores (2013)

Planteaba una situación sobre algoritmos distribuidos y tenías que decir que algoritmo de la teórica servía en ese caso (era para balancear la carga entre distintos procesadores que no sabías cómo recibían los procesos, pero que sabías que algunos reciben mucho mas que otros)
Podias mirar con la Instantánea Global usar una cola para todos los procesadores juntos en vez de 1 para cada 1.

¿Qué parte del algoritmo de la panadería de Lamport hacía que fuera para sistemas paralelos y no para distribuidos? (2013)

Usa memoria compartida.
El algoritmo desempata por process id (pid), que es parte de una sola computadora.
Tenes que tener una forma centralizada de desempatar.

Explicar compareAndSwap() y demostrar que su número de consenso es Infinito (∞). (16/11/2017)

Compare-and-swap/set

```
1  atomic T compareAndSwap(T u /* expected */,
2                          T v /* update */ ) {
3    T w = register;
4    if (u == w) register = v;
5    return w;
6  }
7
8  atomic bool compareAndSet(T u /* expected */,
9                            T v /* update */ ) {
10   if (u == register) {
11     register = v;
12     return true;
13   }
14   else return false;
15 }
```

compareAndSwap toma dos argumentos: el esperado y uno para actualizar. Si el valor actual del registro es el esperado entonces se actualiza; en caso contrario no se modifica. El método retorna un booleano indicando si se modificó o no el valor del registro.

```
1  class CASConsensus extends ConsensusProtocol {
2    private final int FIRST = -1;
3    private AtomicInteger r = new AtomicInteger(FIRST);
4    public Object decide(Object value) {
5      propose(value);
6      int i = ThreadID.get();
7      if (r.compareAndSet(FIRST, i)) // I won
8        return proposed[i];
9      else                           // I lost
10       return proposed[r.get()];
11   }
12 }
```

Figure 5.16 Consensus using compareAndSwap().

The threads share an AtomicInteger object, initialized to a constant FIRST, distinct from any thread index. Each thread calls compareAndSet() with FIRST as the expected value, and its own index as the new value. If thread A's call returns true, then that method call was first in the linearization order, so A decides its own value. Otherwise, A reads the current AtomicInteger value, and takes that thread's input from the proposed[] array.

Definir el problema de consenso COMMIT sin fallas. Dar una solución. (16/11/ 2017)

El problema de consenso COMMIT se basa en que todos los threads acuerden sobre si commitear los cambios o abortar. Tiene validez si en los casos en que un thread quiere abortar => todos abortan o todos los threads pueden commitear => todos commitean.

Two-phase commit

Se elige un proceso distintivo, por ejemplo, 1

Fase 1

$\forall i \neq 1$: i envía $in(i)$ a 1. Si $in(i) = 0$, $decide(i) = 0$.
$i = 1$: Si recibe todos 1, $decide(i) = in(i)$, si no, $decide(i) = 0$.

Fase 2

$i = 1$: Envía $decide(i)$ a todos.
$\forall i \neq 1$: Si i no decidió, $decide(i)$ es el valor recibido de 1.

Defina y pruebe el número de consenso para TAS. (03/08/2017)

TAS tiene un número de consenso igual a 2. Demo:

```
1  class RMWConsensus extends ConsensusProtocol {
2    // initialize to v such that f(v) != v
3    private RMWRegister r = new RMWRegister(v);
4    public Object decide(Object value) {
5      propose(value);
6      int i = ThreadID.get();      // my index
7      int j = 1-i;                 // other's index
8      if (r.rmw() == v)            // I'm first, I win
9        return proposed[i];
10     else                         // I'm second, I lose
11       return proposed[j];
12   }
13 }
```

Figure 5.14 2-thread consensus using RMW.

Since there exists f in F that is not the identity, there exists a value v such that f(v) != v. In the decide() method, as usual, the propose(v) method writes the thread's 4input v to the proposed[] array. Then each thread applies the RMW method to a shared register. If a thread's call returns v, it is linearized first, and it decides its own value. Otherwise, it is linearized second, and it decides the other thread's proposed value.

Test and Set es una primitiva de sincronización booleana que setea el valor de un registro y devuelve el valor anterior.

Definición: El número de consenso es n si y sólo si puede haber a lo sumo n threads que se pongan de acuerdo en el valor de la variable y el valor que se define es propuesto por alguno de ellos.

Veamos que el número de consenso de TAS es mayor igual a dos:
Dos threads (t0 y t1) escriben sus valores propuestos en las posiciones 0 y 1 de un arreglo respectivamente.
Ambos llaman a TAS en un registro R. Si TAS devuelve 0 para el ti, entonces significa que ganó y debe devolver su valor. Si devuelve 1, entonces gano el otro y se debe devolver el valor del otro.

Veamos ahora que el número de consenso de TAS es menor igual a dos:

Supongamos que existe un algoritmo de consenso de n = 3 y que tenemos 3 threads t0,t1 y t2 quieren decidir el valor de un registro. Cada uno de ellos quiere proponer v0, v1 y v2 respectivamente.
En alguna de las ramas de las posibles ejecuciones, se puede llegar a un estado bivalente, donde v0 y v1 pueden ser los valores propuestos. En este caso, sin pérdida de la generalidad, t0 y t1 pueden mover y el que mueva primero definirá su valor. Claramente esta operación debe ser un TAS sobre el mismo registro ya que de ser en registros diferentes, no podríamos saber cual sucedió primero. Aquí consideremos dos posibles escenarios:

t0 mueve; luego t1 mueve; luego t2 lee el valor que debe decidir
T1 mueve; luego t2 lee el valor que debe decidir

Estos dos escenarios son indistinguibles para t3, ya que leerá el registro luego de que este sea accedido por t2. Sin embargo, los valores que debe devolver son distintos en los dos escenarios. Esto es una contradicción que salió de suponer que existe un algoritmo de 3-consenso para TAS.

Defina los mecanismos de *afinidad* y *migración* para compartir la carga en sistemas distribuidos. (03/08/2017)

Al hablar de scheduling en sistemas distribuidos tenemos dos niveles, local (dar el procesador a un proceso listo) y global (asignar un proceso a un procesador).
A su vez existen dos formas de compartir la carga entre los procesadores (scheduling global):

- Estática (affinity): en el momento de la creación del proceso. Es decir, nunca se migra el proceso a otro procesador.
- Dinámica (migration): la asignación varía durante la ejecución. También existen distintas técnicas de migración, puede ser sender initiated o receiver initiated.

Describa las estrategias de migración de procesos en sistemas distribuidos (03/08/2017)

Existen dos estrategias de migración:

- Sender initiated: Iniciada por el procesador sobrecargado
- Receiver initiated / work stealing: Iniciada por el procesador libre