PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2333 — Sistemas Operativos y Redes — 2/2018

Midterm (Sistemas Operativos)

Lunes 1-Octubre-2018

Duración: 2 horas
SIN CALCULADORA

1. **[12p]** Responda verdadero o falso. Las falsas solo reciben puntaje si están justificadas.

   Verdaderas: 0 si es incorrecta; 1 si es correcta.

   Falsas: 0 si es incorrecta, no hay justificación, o la justificación es incorecta; 1 si la justificación es correcta.

   1.1) En un sistema *multi-core* se puede implementar *multitasking*, pero en un sistema *single-core* no.

   Falso. *Multitasking* se refiere a que puede mantener el estado de múltiples tareas, no tiene que ver con la cantidad de *cores*.

   1.2) Un sistema operativo basado en microkernel no puede proveer tantas funcionalidades como uno monolítico

   Falso. No tiene que ver con la cantidad de funcionalidades, sino dónde ejecutan los módulos (en *user* o en *kernel space*)

   1.3) El puntero a la tabla de páginas de un proceso se encuentra almacenado en su PCB (*Process Control Block*)

   Verdadero. Hay una por cada proceso, por lo tanto una por cada PCB.

   1.4) Después de ejecutar `fork()`, el proceso padre puede obtener el ID de su hijo, y el hijo puede obtener el ID de su padre.

   Verdadero. Con el `fork()` el padre conoce el ID del hijo, y el hijo puede ejecutar `getppid()` para obtner el de su padre.

   1.5) Un proceso multithreaded CPU-bound en un sistema multi-core funciona más rápido que la versión equivalente single-threaded.

   Verdadero. Un proceso *CPU-bound* hace uso intenso de todos los *core*. Si suponemos que hay suficiente *kernel threads*, pueden correr efectivamente en paralelo, por lo tanto será más rápido. Notar que esto no es necesariamente cierto si no fueran *I/O-bound* porque podrían pasar mucho tiempo bloqueados.

   1.6) Un *scheduler* interactivo intenta minimizar el tiempo que un proceso permanece en el sistema (*turnaround time*).

   Falso. Los *scheduler batch* intentan minimizar el *turnoaround time*. Tambié se puede decir que el interactivo intenta minimizar el tiempo de permanencia en la cola *ready*.

   1.7) En un sistema *single-core*, un protocolo en que, cada vez que un *thread* entra a la sección crítica, se deshabilitan las interrupciones, y al salir de la sección crítica se vuelven a habilitar, cumple exclusión mutua.

   Verdadero. Si sólo hay un *core*, desahabilitando las interrupciones en él, se imposibilita sacarlo de la CPU, por lo tanto naturalmente se provee exclusión mutua.

   1.8) Un semáforo puede ser utilizado para notificar a otro procesos o *thread* que un recurso ha sido liberado.

   Verdadero. Al hacer `V()`, ó `up()`, se está modficiando una condición que podría tener bloqueado a algún proceso que estuviera ejecutando `P()`. Este mecanismo se puede utilizar como una forma de notificación.

   1.9) El uso de segmentación permite eliminar completamente la fragmentación externa.

   Falso. La paginación elimina la fragmentación externa. También se puede decir que la segmentación elimina la fragmentación interna.

1.10) El tamaño del working set de un proceso es constante durante la ejecución y está definido como un parámetro del sistema operativo

Falso. La cantidad de accesos que se utiliza para determinar el tamaño del *working set* (Δ) puede ser constante, pero el tamaño del *working set* va a variar dependiendo de la cantidad de páginas que el proceso referencia en esos Δ accesos.

1.11) El sistema de archivos recibe solicitudes por archivos y escribe directamente en sectores del disco.

Falso. El sistema de archivo solicita escritura a una capa de abstracción de bloques provista por el sistema operativo. No escribe directamente al disco.

1.12) Un disco puede mostrarse al sistema operativo como si fueran múltiples discos.

Verdadero. Mediante particiones.

2. **[20p]** Responda de manera breve y lo más precisa posible las siguiente preguntas.

2.1) **[4p]** Las *syscall*, como servicios del sistema operativo, ejecutan en *kernel space*. Un proceso de usuario, por otro lado, no puede ejecutar arbitrariamente instrucciones en *kernel space*, ni escribir en memoria del *kernel*, ni cambiar arbitrariamente el modo de protección.

a) ¿Cómo consigue un proceso de usuario que el código de una *syscall* se ejecute en modo *kernel*?

R.

2p. Mencionar que se usan (1) bibliotecas de usuarios que preparan el llamado a la *syscall* y (2) interrupciones para pasar el control al *kernel*.

1p. Olvida mencionar (1) ó (2)

b) ¿Cómo consigue un proceso de usuario entregarle argumentos a una *syscall* si no puede escribir en memoria del *kernel*?

R.

2p. Mencionar que se dejan los argumentos en una posición de la memoria del usuario (técnicamente en el *stack* pero no es necesario ser tan específico), y que después de la interrupción, el *kernel* copia los datos de la memoria del usuario a la memoria del *kernel*.

1p. Olvida mencionar que se copian los datos, o bien se deja ver que el código del usuario hace la copia antes de llamar la interrupción.

2.2) **[4p]** El PCB (*Process Control Block*) de un proceso almacena información de estado de un proceso.

a) **[1p]** Mencione 3 elementos que se almacenan en un PCB

R.

1p. Pueden mencionar varios: PC, *stack pointer*, PID, estructuras de contabilidad, *owner*, permisos, punteros a tabla de páginas. Lo que no pueden poner es código binario, valores de variables, sea en el data, heap o stack.

0.5. Hasta uno malo.

0p. Más de uno malo.

b) **[1p]** ¿En qué parte del sistema se almacenan los PCB?

R.

1p. En la memoria del *kernel* (o del sistema operativo)

0p. Otra respuesta

c) **[2p]** ¿Cómo se administran los *program counter* de un proceso *multithreaded*?

R.

2p. Indicar que para los *kernel threads*, el sistema operativo mantiene un *thread control block* por cada *thread*, y que para los *user threads*, la biblioteca de usuario mantiene el *thread control block*. En cualquier caso no los almacena directamente el PCB. Sí los podría referenciar indirectamente.

1p. Falla en mencionarlo para los *user threads* o para los *kernel threads*.

2.3) **[6p]** El algoritmo de *scheduling* MLFQ (*multilevel feedback queue*) es un algoritmo para *scheduling* interactivo que considera múltiples colas, desde las primeras que tienen alta prioridad y *quantum* pequeño, hasta las que tienen menor prioridad y *quantum* mayor. También incluye un mecanismo de envejecimiento (*aging*).

a) ¿Qué significa que un algoritmo de *scheduling* sea interactivo?
R.
2p. Podrían mencionar que minimiza el tiempo de espera en la cola *ready*, o que minimiza el tiempo de respuesta. Alternativamente también se podría decir que funciona bien ante una carga de trabajo mayoritariamente *I/O bound*.
1p. La respuesta se aproxima, pero no es precisa en algún concepto, ya que sea en la cola que espera, o el tiempo que mide.
0p. Incorrecta

b) ¿Por qué el algoritmo MLFQ provee "interactividad"? **R.**
2p. Suponiendo una carga de trabajo mayoritariamente *I/O bound*, los procesos pasarán rápidamente por la primera cola y luego saldrán, muchos de ellos sin consumir su *quantum*. Los que consumen su *quantum* bajan de prioridad pero tienen oportunidad cuando se acaben los de la cola superior. Se sigue dando prioridad a los más "cortos". Por otro lado, los que consumen más tiempo son los que no son *I/O Bound*. Gracias al *aging* ellos igual reciben tiempo, pero no son los importantes al momento de determinar si un sistema es interactivo o no.
1p. Falta de precisión o alguna incorrectitud en la respuesta.
0p. Más de una concepto incorrecto.

c) ¿Por qué se introdujo el mecanismo de *aging*?

2.4) **[2p]** Los sistemas modernos proveen primitivas de sincronización por *hardware* como `test_and_set` o también `compare_and_swap`. ¿Por qué estos sistemas han preferido integrar estas primitivas en *hardware* en lugar de bibliotecas del sistema operativo?

R.

2p. Mencionar que la acción de primitivas como `test_and_set`, que se desea que sean atómicas, requiere ejecutar dos o más instrucciones y la interrupción puede ocurrir entre cualquier de ellas. Por lo tanto el *software* no es suficiente para implementar atomicidad, y se necesita que sean instrucciones donde la atomicidad sea provista por el *hardware*.

1p. Mencionar conceptos incorrectos, como que algunas instrucciones sí se hacen por *software*, o que algunas bibliotecas son suficientes. Es correcto mencionar que el *software* no basta para el caso *multi-core*, pero no es argumento suficiente.

Es incorrecto mencionar que es más rápido por *hardware*. Es cierto, pero no es la razón.

2.5) **[2p]** ¿Por qué el concepto de localidad de referencia es importante al momento de diseñar un algoritmo de reemplazo de páginas?

R.

2p. Mencionar que los algoritmos de reemplazo de páginas se basan en la idea de minimizar los *page fault* suponiendo lo que va a pasar en el futuro. El reemplazo óptimo supone que se conoce todo el futuro, y los algoritmos reales intentan aproximarse a ese futuro. El supuesto de la localidad (pueden hacer especificidad a la de referencia o a la espacial; durante la prueba les dije que pensaran en la espacial, pero ambas sirven) dice que si se usa una página probablemente se usará en el futuro una página cercana (espacial), o se repetirá una página recientemente usada (temporal). Como la mayoría de los programas se comportan de esta manera, los algoritmos de aproximación al óptimo toman este supuesto. También pueden mencionar (al menos 1p si lo hacen) que ante un acceso aleatorio de páginas, todos los algoritmos (salvo el óptimo) se comportan igual de mal.

1p. Les falta alguno de los conceptos que se mencionan en el párrafo anterior.

2.6) **[2p]** Describa el funcionamiento de un sistema de archivos, desde el punto de vista de los servicios que ofrece (al usuario), y los servicios que utiliza.

R.

2p. Sistema de archivos **ofrece** una interfaz/servicio para editar/crear/modificar una abstracción de archivos y directorios (1p); y **utiliza** una interfaz/servicio para manipular bloques del disco.

1p. Uno de ambos aspectos está incorrecto o no es suficientemente preciso.

Es incorrecto decir que el sistema de archivos escribe en disco. El sistema de archivos escribe en bloques. El que escribe en el disco es el *driver* del disco.

3. **[14p]** Se propone el siguiente código para *threads* lectores (*reader*) y escritores (*writer*). Este código utiliza 4 variables compartidas: `int readCount`, `FILE* sharedFile`, y los semáforos `mutex` y `writeBlock`. El objetivo es permitir el acceso a la sección crítica de n lectores simultáneos, o bien de 1 escritor.

```
void reader() {
  while(1) {
    // muchas líneas de código
    mutex.P();
    readCount++;
    if(readCount == 1)
      writeBlock.P();
    mutex.V();
    read(sharedFile);  // seccion critica
    mutex.P();
    readCount--;
    if(readCount == 0)
      writeBlock.V();
    mutex.V();
  }
}
```

```
void writer() {
  while(1) {
    // muchas líneas de código
    writeBlock.P();
    write(sharedFile);
    writeBlock.V();
  }
}
```

Respecto a este código, responda las siguientes preguntas:

3.1) **[4p]** ¿Con qué valores debe iniciar las variables compartidas para que el código funcione correctamente?

R.

Hubo alguna confusión si es que el n debía ser un número preciso (n y no más). La idea de la pregunta, que es similar a un código visto en el material, es que sea cualquier $n > 0$. Se puede permitir que lo hayan interpretado como un n fijo, pero debe estar explícito y ser consecuentes con eso. Las sugerencias a continuación suponen la idea original que es para cualquier cantidad de lectores mayor a 0 (sin límite superior).

No es necesario explicar por qué funciona con esos valores.

1p. `int readCount` en `0`

1p. `FILE* sharedFile` en `NULL` (o ignorarlo, no es importante)

1p. `mutex` en `1`

1p. `writeBlock` en `1`

3.2) **[4p]** Explique si este código cumple la propiedad de progreso.

R.

1p. Decir que cumple progreso.

1p. Explicar que si no hay nadie, los lectores pueden pasar. Si no hay nadie dentro, y llega un lector, pasa inmediatamente por el primer par `mutex.P()`, `mutex.V()` y atómicamente deja `writeBlock` en 0.

1p. Explicar que si hay lectores, más lectores pueden pasar. Si hay lectores dentro y llega un lector, pasa inmediatamente porque no pasa por el `writeBlock.P()` (y está bien que no pase porque `writeBlock` está en 0.

1p. Explicar que si no hay nadie, los escritores pueden pasar. Si no hay nadie dentro, y llega un escritor, pasa por el primer `writeBlock.P()` ya que está en 1, y entra.

3.3) **[4p]** Explique si este código cumple la propiedad de espera acotada.

R.

1p. Decir que no cumple. No es necesario decir qué tendría que hacer para que cumpliera (por ejemplo, usar una cola).

1.5p. Explicar que los lectores no tienen espera acotada. Es posible que haya un escritor dentro, y haya tanto un lector como un escritor esperando en `writeBlock.P()`. Cuando se haga `writeBlock.V()`, cualquiera de los dos puede pasar. Si el lector tiene mala suerte, pasará el escritor.

Alternativamente pueden decir que los lectores sí tienen espera acotada si es que argumentan que el semáforo mantiene una cola ordenada con los que están bloqueados, y que la implementación de `V()` despierta al siguiente de la cola. En ese caso no van a esperar más allá de L turnos, donde L es el largo de la cola cuando el lector ejecutó se bloqueó en `writeBlock.P()`. Si dicen sí tienen esperado acotada no explican algo relativo al orden de llegada, entonces 0,5p.

1.5p. Explicar que los escritores no tienen espera acotada. Si hay $n >= 1$ lectores en la sección crítica y llega un escritor, éste es bloqueará en `writeBlock.P()` porque `writeBlock` estará en 0. Los lectores pueden llegar e irse, pero mientras haya alguno, nadie ejecutará `writeBlock.V()`, y el escritor quedará esperando. El escritor no sabe cuántos turnos tendrá que esperar. Aquí no depende de si el semáforo ordena o no a los que llegan porque los lectores que no son el primero no usan el semáforo `writeBlock`.

3.4) **[2p]** ¿Es posible escribir este código solo reemplazando el semáforo `mutex` por *locks* y las instrucciones `P()` y `V()` por las respectivas `wait()` y `signal()` de las *variables de condición*? Justifique.

R.

El código de esa manera quedaría como (no era necesario escribirlo):

```
void reader() {
  while(1) {
    // muchas líneas de código
    lock.acquire();
    readCount++;
    if(readCount == 1)
      writeBlock.wait(lock);
    lock.release();
    read(sharedFile);  // seccion critica
    lock.acquire();
    readCount--;
    if(readCount == 0)
      writeBlock.signal(lock);
    lock.release();
  }
}
```

```
void writer() {
  while(1) {
    // muchas líneas de código
    writeBlock.wait(lock);
    write(sharedFile);
    writeBlock.signal(lock);
  }
}
```

Se pueden esgrimir razones de programación, por ejemplo que el *lock* no estaría definido para el caso de los escritores (1p si solo dicen eso). Sin embargo, la razon semántica principal es que la instrucción `wait(lock)` de las variables de condición bloquea siempre al *thread* que la llama, y que una vez que éste ha sido liberado, debe volver a verificar que la condición que lo hizo bloquearse es cierta (1p). En este caso habría que reemplazar el `if (readCount == 1)` por un `while(readCount==1)`. La semántica de `wait(lock)` no es exactamente equivalente a la de `P()`.

Por otro lado, la semántica de `signal(lock)` tampoco es equivalente a la de `V()`, ya que si no hay nadie esperando, no hay ningún efecto en el `writeBlock.signal(lock)`. (1p).

4. **[14p]** Considere un sistema con direcciones virtuales de 40 bit, con soporte de hasta 16GB de memoria física, y páginas de 4KB. Cada dirección de memoria referencia 1 Byte.

4.1) **[4p]** ¿Cuánto es la cantidad máxima de memoria que puede direccionar cada proceso?

R.

4p. (3p) Si las direcciones virtuales son de 40 bit, entonces hay 2^{40} direcciones. (1p) Cada dirección referencia 1Byte, por lo tanto la cantidad máxima de memoria es 2^{40}Byte = 1TB.

3p. Solo menciona la cantidad de direcciones, pero no indica la cantidad en Bytes, o bien entrega 2^{40} bit.

2p. Hace el 2^X, pero con un X que no es 40. Involucra memoria física o el tamaño de las páginas.

1p. Hace planteamientos correctos, pero no efectúa ninguna operación de cálculo.

0p. Realiza operaciones incorrecta o ninguna operación, o muestra errores conceptuales como involucrar memoria física o el tamaño de las páginas.

4.2) **[4p]** Si el tamaño de cada entrada en la tabla de páginas (PTE) está alineado a una cantidad de Byte que sea potencia de 2, ¿de qué tamaño, en Byte, es la tabla de páginas? ¿cuántos bit quedan disponible para *metadata*?

R.

4p (0.5p) Si las páginas son de 4KB $= 2^{12}$B, se necesitan 12bit para el *offset*.

(0.5p) Si la dirección física es de 16GB $= 2^{34}$B, entonces la dirección física debe tener 34bit.

(0.5p) De esos 34bit, se usan 12bit para el *offset* dentro del *frame*, por lo tanto quedan $34 - 12 = 22$bit para el número de *frame*.

(1p) Una PTE, entonces necesita al menos 22bit. Como debe ser una cantidad de Byte que sea potencia de 2, entonces debe extenderse hasta 4B $= 32$bit. Estos 10bit que se agregan son los que quedan disponibles para metadata.

(0.5p) La cantidad de bit para indicar un número de página es $40 - 12 = 28$bit, por lo tanto hay 2^{28} PTEs.

(1p) El tamaño de la tabla de páginas es $2^{28} \times 4\text{B} = 2^{30}\text{B} = 1GB$.

Si hay un cálculo erróneo, ese cálculo no lleva puntaje, pero el resto del procedimiento se evalúa de acuerdo a ese cálculo.

4.3) **[6p]** Para reducir la cantidad de memoria necesaria por cada proceso se propone aumentar el tamaño de las páginas. ¿Hasta cuánto debe crecer el tamaño de la página para que la tabla de páginas quepa completamente en una página? Puede ignorar los bits de *metadata*, pero debe considerar el alineamiento a una potencia de 2 en los Byte de la PTE.

R.

Con páginas de 4KB, y sin *metadata*, la tabla es de tamaño $2^{28} \times (22+10)$bit $= 2^{28} \times 4\text{B} = 2^{30}\text{B} = 1$GB.

Con páginas de 8KB, la tabla es de tamaño $2^{27} \times (21 + 11)$bit $= 2^{27} \times 4\text{B} = 2^{29}\text{B} = 512$MB

Con páginas de 16KB, la tabla es de tamaño $2^{26} \times (20 + 12)$bit $= 2^{26} \times 4\text{B} = 2^{28}\text{B} = 256$MB

Con páginas de 32KB, la tabla es de tamaño $2^{25} \times (19 + 13)$bit $= 2^{25} \times 4\text{B} = 2^{27}\text{B} = 128$MB

Con páginas de 64KB, la tabla es de tamaño $2^{24} \times (18 + 14)$bit $= 2^{24} \times 4\text{B} = 2^{26}\text{B} = 64$MB

Con páginas de 128KB, la tabla es de tamaño $2^{23} \times (17 + 15)$bit $= 2^{23} \times 4\text{B} = 2^{25}\text{B} = 32$MB

Con páginas de 256KB, la tabla es de tamaño $2^{22} \times (16)$bit $= 2^{22} \times 2\text{B} = 2^{23}\text{B} = 8$MB

Con páginas de 512KB, la tabla es de tamaño $2^{21} \times (15 + 1)$bit $= 2^{21} \times 2\text{B} = 2^{22}\text{B} = 4$MB

Con páginas de 1MB, la tabla es de tamaño $2^{20} \times (14 + 2)$bit $= 2^{20} \times 2\text{B} = 2^{21}\text{B} = 2$MB

Con páginas de 2MB, la tabla es de tamaño $2^{19} \times (13 + 3)$bit $= 2^{19} \times 2\text{B} = 2^{20}\text{B} = 1$MB

6p. Valor correcto.

5p. Operaciones correctas con un error numérico de arrastre.

4p. Operaciones correctas con algún concepto incorrecto (confudir bit con Byte, o no considerar el alineamiento a potencia de 2B)

3p. Algunas operaciones incorrectas, o bien operaciones correctas pero resultado incompleto.

2p. Errores conceptuales, que permiten entender parte de la idea. Operaciones son mayoritariamente correctas, pero no se alcanzan a terminar.

1p. Errores conceptuales que no permiten encontrar la solución (confundir bits para el número de *frame* con bits para el número de página), pero se efectúan cálculos consistentes.

0p. Errores conceptuales graves, o fórmula mal planteada, o no se efectúan cálculos.