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.

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

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

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

   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.

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

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

   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.

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

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

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

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

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

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*?

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

   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

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

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

   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?
b) ¿Por qué el algoritmo MLFQ provee "interactividad"?
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?

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

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.

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?

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

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

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.

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?

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*?

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.