Sistemas Operativos I 2022
Trabajo Práctico Obligatorio 2

Objetivos

- Analizar diferentes algoritmos de **planificación de procesos**.
- Examinar los **estados de un proceso** en un sistema operativo.
- Estudiar semáforos y la implementación de otras herramientas de **sincronización de procesos**.

Referencias

[1] Tanenbaum, Bos – Modern Operating Systems - Prentice Hall; 4 edition (March 10, 2014) - ISBN-10: 013359162X
[2] Douglas Comer - Operating System Design - The Xinu Approach. CRC Press, 2015. ISBN : 9781498712439
[3] Silberschatz, Galvin, Gagne - Operating Systems Concepts - John Wiley & Sons; 10 edition (2018) – ISBN 978-1-119-32091-3

Software y Hardware

La versión de Xinu que utilizamos es para arquitectura PC (x86). Ejecutamos el sistema operativo Xinu en una máquina virtual llamada QEMU, que emula una PC básica.
El trabajo puede realizarse sobre las máquinas de los laboratorios (RECOMENDADO).
Quienes tengan Linux en sus casas, podrían intentar instalar todo lo necesario y llevarlo a cabo ahí también. Una tercera posibilidad es el acceso remoto RDP comentado en la web de la materia.

Ejercicio 1. *Implementar una política de planificación de procesos de alto nivel con ráfagas de CPU estáticas.*

a. Describir el planificador de procesos de Xinu.

b. Implementar en Xinu un planificador de CPU de alto nivel, que define una política de planificación. La política es la siguiente: dado tres **procesos a, b y c**, el sistema operativo le asigne CPU como sigue:
   - a: 60% del tiempo
   - b: 30% del tiempo
   - c: 10% del tiempo
   -

   Las 3 ráfagas de CPU tiene una duración total de 200ms. Es decir, que el nuevo planificador, le otorgará al **proceso a** una ráfaga de CPU con una duración que equivale al 60% de 200ms, al proceso b el 30% de 200ms, etc.

> Este nuevo planificador de alto nivel define la **política de planificación,** mientras que el actual planificador en el kernel de Xinu es el **mecanismo**. El mecanismo es siempre el mismo, mientras que la política puede ser definida por el programador en base a sus necesidades. La política definida por el programador debe poder ser implementada utilizando el único mecanismo en el kernel. Cuanto más básico o genérico es el mecanismo en el kernel, más sencillo será implementar políticas desde los procesos.

Una posible implementación es que el planificador de alto nivel sea un proceso, con la más alta prioridad, como se muestra en la Figura 1. Este proceso, cambia la prioridad de uno de los 3 procesos (a, b o c) con una prioridad alta (pero menor que la de este planificador de alto nivel). Luego, el planificador de alto nivel se pone a dormir durante el tiempo de ráfaga para el proceso planificado (el tiempo para a, b o c). De esta manera, cuando el planificador de alto nivel "duerme", Xinu le asigna CPU al proceso con la más alta prioridad recién configurado. Luego de la ráfaga, Xinu reanudará al planificador de alto nivel porque: "despertará" al proceso, lo pondrá en estado de LISTO, e inmediatamente le asignará la CPU (por tener, el planificador de alto nivel, la prioridad mas alta). En ese momento el planificador de alto nivel puede planificar la siguiente ráfaga de CPU para el siguiente proceso (a, b o c), como anteriormente, y volver a dormir. Y así sucesivamente.

En la Figura 1. puede observarse un diagrama del sistema. Los 4 procesos en color rosado son los procesos a implementar en Xinu para este ejercicio.

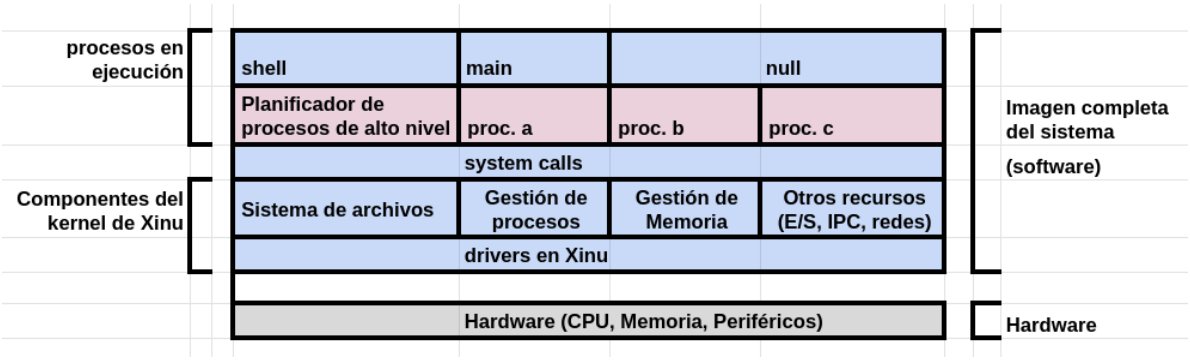

Figura 1. Componentes del sistema para este ejercicio

Como las ráfagas de CPU serán estáticas (cada proceso a, b y c recibirá un tiempo de CPU fijo), el proceso planificador de CPU de alto nivel se puede implementar con los siguientes system calls de Xinu:

```
suspend(pid);
resume(pid);
getprio(pid);
chprio(pid, newprio);
getpid();
sleepms(ms);
```

Por ejemplo, si al **proceso a** le toca una ráfaga de CPU de 20ms, entonces el planificador podría ser implementado como sigue:

```
void high_level_scheduler (                         /* argumentos:                  */
                            int pid_a, int ms_a,    /* PID de a y rafaga de a en ms */
                            int pid_b, int ms_b,    /* PID de b y rafaga de b en ms */
                            int pid_c, int ms_c     /* PID de c y rafaga de c en ms */
                          )

                    /* obtener el PID del planificador (proceso actual) */
                     /*obtener la prioridad del planificador (proceso con la mas alta prioridad)l */

            while (1) {

                    /* obtener la prioridad del proceso a */
                    /* cambiar la prioridad del proceso a, a un valor igual a la
                    /* prioridad del planificador menos 1 */
                    /* liberar la CPU por ms_a ms (ponerse a dormir), por lo
                            que Xinu asignará la CPU  al proceso a */
                    /* devolverle al proceso a su prioridad original */

                    /* planificar aquí el proceso b */

                    /* planificar aquí el proceso c */

            }
}
```

Para poder observar el comportamiento del planificador de alto nivel, desarrolle los 3 **procesos a, b y c**, de manera que sólo realicen cómputo, sin llamadas al sistema bloqueantes. Una posibilidad (ejemplo) podría ser que cada proceso incrementa una variable local, y vaya mostrando en una línea particular de la terminal su valor. Con lo cual, cada proceso solo muestra el valor de su variable incrementada en una línea particular de la pantalla conectada a la terminal. Luego de pasado un tiempo de ejecución, cada variable debería ser de un valor proporcional a las otras, de acuerdo a las ráfagas de CPU asignadas.

c. Explique por qué este planificador no le asignaría el tiempo de CPU a cada proceso correctamente si uno o varios procesos solicitan un servicio al Sistema Operativo que sea bloqueante.

Ejercicio 2. *Sincronización entre procesos.*

a. Agregar este programa al shell de Xinu (recuerde cambiar el nombre de main por el nombre que seleccione para el programa):

```
#include <xinu.h>

void    produce(void), consume(void);

int32   n = 0;          /* Global variables are shared by all processes   */

/*------------------------------------------------------------------------
 *  main  -  Example of unsynchronized producer and consumer processes
 *------------------------------------------------------------------------
 */
void    main(void)
{
        resume( create(consume, 1024, 20, "cons", 0) );
        resume( create(produce, 1024, 20, "prod", 0) );
}

/*------------------------------------------------------------------------
 * produce  -  Increment n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    produce(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                n++;
}

/*------------------------------------------------------------------------
 * consume  -  Print n 2000 times and exit
 *------------------------------------------------------------------------
 */
void    consume(void)
{
        int32   i;

        for( i=1 ; i<=2000 ; i++ )
                printf("The value of n is %d \n", n);
}
```

b. Este programa es un típico programa compuesto de 3 procesos, main, productor, consumidor. Los procesos productor y consumidor se “comunican” a través de una variable compartida. El productor genera valores, y el consumidor los muestra en pantalla. Ejecutar el programa. Describir qué sucede. ¿El programa funciona como el programador pretendió?

c. Modifique el programa utilizando algún mecanismo provisto por el sistema operativo para la sincronización de procesos, de manera que el programa funcione como el programador pretendía.

Ejercicio 3. *Implementación de recursos lógicos.*

a. Agregar este programa al shell de Xinu.

```
/* mut.c - mut, operar, incrementar */

#include <xinu.h>

void    operar(void), incrementar(void);

unsigned char x = 0;

/*------------------------------------------------------------------------
 * mut  --  programa con regiones criticas
 *------------------------------------------------------------------------
 */
void    mut(void)
{
        int i;

        resume( create(operar, 1024, 20, "process 1", 0) );
        resume( create(incrementar, 1024, 20, "process 2", 0) );

        sleep(10);

}


/*------------------------------------------------------------------------
 * operar x e y
 *------------------------------------------------------------------------
 */
void    operar(void)
{
        int y = 0;

        printf("Si no existen mensajes de ERROR entonces todo va OK! \n");

        while (1) {

                /* si x es multiplo de 10 */
                if ((x % 10) == 0) {

                        y = x * 2;       /* como y es el doble de x entonces
                                          * y es multiplo de 10 tambien
                                          */

                        /* si y no es multiplo de 10 entonces hubo un error */
                        if ((y % 10) != 0)
                                printf("\r ERROR!! y=%d, x=%d \r", y, x);
                }
        }
}

/*------------------------------------------------------------------------
 * incrementar x
 *------------------------------------------------------------------------
 */
void    incrementar(void)
{
        while (1) {
                x = x + 1;
        }
}
```

Ejecutar el programa. ¿Por qué este programa (sus procesos) emiten mensajes de error? ¿Qué es posible hacer para solucionar el problema?.

b. El sistema operativo Xinu no provee mecanismos para resguardar regiones críticas. Pero, su autor, define a Xinu como un microkernel, lo que significa que provee mecanismos mínimos que permiten luego construir otros más complejos. Usando el mecanismo de semáforos provisto por Xinu, implementar una protección de exclusión mutua (mutex). Este mutex debe estar compuesto por dos funciones :
mutex_init(), mutex_lock(); y mutex_unlock();
c. Proteger las regiones críticas del programa original con este nuevo mecanismo de exclusión mutua.

Ejercicio 4. *Algoritmos de planificación de procesos de Sistemas Operativos*

Investigue cómo se llaman y cómo funcionan los algoritmos de planificación de procesos del sistema operativo Windows y Linux. Redacte una respuesta-aprendizaje. Esto implica: **no buscar en google y sólo copiar y pegar.** Estar seguro de que la respuesta es una respuesta correcta (ejemplo: buscar en google puede otorgar 1 millón de respuestas de sitios webs distintos. ¿Cómo puede saber si algunas de las respuestas son válidas? ¿Cómo saber si están completas?. Tanto Linux como Windows han cambiado sus algoritmos de planificación con el correr de los años. La respuesta debe presentar el algoritmo (o los algoritmos actuales).
Explique los algoritmos coloquialmente. Esto implica que debe entender su funcionamiento.
Ayuda: Abraham Silberschatz

Ejercicio 5. *Ejemplos de Planificación de CPU*

Sea un sistema con la siguiente carga de procesos. El planificador de CPU del sistema es por prioridades, round-robin, apropiativo.

Proceso	Prioridad	Ráfaga	Arribo a la cola de listos
P1	8	15	0
P2	3	20	0
P3	4	20	20
P4	4	20	25
P5	5	5	45
P6	5	15	55

A más alto número de prioridad mayor prioridad. El quantum del sistema es de 10 unidades. A misma prioridad el planificador es round-robin.

a. Mostrar el orden de ejecución y las ráfagas de CPU de los procesos.
b. ¿Cuánto es el tiempo de turnaround para cada proceso? ¿Y el valor medio contando todos los procesos?
c. ¿Cuál es el tiempo de espera para cada proceso? ¿Y el valor medio contando todos los procesos?

Ejercicio 6.

Sea un sistema con la siguiente carga de procesos.

Proceso	Ráfaga	Prioridad
P1	5	4
P2	3	1
P3	1	2
P4	7	2
P5	4	3

Los procesos arriban al sistema en el momento 0, en este orden: P1 , P2 , P3 , P4 , P5.

a. Mostrar el orden de ejecución y las ráfagas de CPU de los procesos si el planificador es FCFS, SJF, con prioridades no-apropiativo, round-robin (quatum = 2).
b. ¿Cuánto es el tiempo de turnaround para cada proceso? ¿Y el valor medio contando todos los procesos? (para cada algoritmo).
c. ¿Cuál es el tiempo de espera para cada proceso? ¿Y el valor medio contando todos los procesos? (para cada algoritmo).
d. ¿Cuál de los algoritmos produce el mínimo promedio de tiempo de espera?

Preguntas extras teóricas. No se entregan, son de repaso.
Son para quienes quieran analizar los conceptos vistos. Este análisis podría servir para poder elaborar respuestas, a preguntas generales teóricas como las siguientes. Para la promoción y final.:

Procesos

1. Responda las siguientes cuestiones:
    a. a) ¿Qué es un proceso?
    b. b) ¿Cuáles son sus componentes?
    c. c) ¿Dónde se encuentra el contador de programa (PC) del proceso?
2. Modelo de estados de procesos.
    a. Describa el modelo de 5 estados de procesos.
    b. ¿Qué eventos provocan la transición de estados de los procesos?
    c. ¿Por qué se requiere agregar los estados Bloqueado/Suspendido y Listo/Suspendido al modelo de 5 estados de procesos en un sistema que soporta swapping? Para estos nuevos estados y para los estados con los que se relacionan, ¿qué eventos provocan el cambio de estados?
3. El tiempo de vida de un proceso está comprendido entre su creación y su finalización.
    a. ¿Por qué razones puede ser creado un proceso?
    b. ¿Por qué razones puede finalizar un proceso?
    c. Asocie los motivos de creación/finalización de un proceso con su correspondiente estado y transición enel modelo de 7 estados de procesos.
4. Describa las posibilidades que existen cuando un proceso crea un nuevo proceso:
    a. en términos de ejecución de procesos.
    b. en términos de espacio de direcciones de memoria.
    c. en términos de compartimiento de recursos.
5. En UNIX:
    a. Enumere los pasos realizados por el sistema operativo cuando un proceso realiza un system call fork().
    b. Asumiendo que se crea el proceso hijo, ¿cómo es posible que la variable pid tenga un valor en el proceso padre y otro en el proceso hijo?
    c. ¿Qué transiciones de estados produce el fork() tanto en el proceso padre como en el proceso hijo?
    d. ¿Podría fallar un fork()? ¿Por qué? ¿Se puede solucionar?

6. En UNIX: Inmediatamente después que el sistema operativo completó un system call fork() solicitado por un proceso:
    a. a) Mencione similitudes y diferencias entre el proceso padre e hijo.
    b. b) Cuando el proceso hijo alcanza por primera vez el estado Ejecución, ¿cuál es la primera instrucción que ejecuta? Justifique su respuesta.
7. En UNIX: ¿Puede un proceso padre, sin ejecutar un system call wait(), recibir el estado de finalización de sus procesos hijos?

8. ¿Qué significa que un proceso sea cargado en E/S o cargado en CPU?
9. ¿Qué razones motivan a proveer un entorno que permita procesos cooperativos?
10. Explique con sus palabras lo que es IPC (Interprocess Communication). ¿Cuáles son los dos modelos fundamentales de IPC? ¿Es posible utilizar más de un esquema de comunicación simultáneamente dentro de un mismo Sistema Operativo? ¿y dentro de un mismo proceso? Justifique.

Planificación de procesos

1. Analizar la siguiente afirmación: Un sistema multiprogramado con una única CPU, permite la ejecución concurrente de los procesos.
2. Analizar la siguiente afirmación: En un sistema monoprocesador que utiliza un planificador de corto plazo sin desalojo, los únicos cambios de contexto que ocurren son los generados por el abandono voluntario de la CPU por parte del proceso en ejecución, por ejemplo, por finalización o por una llamada al sistema.
3. Analizar la siguiente afirmación: En un Sistema Operativo que maneja threads a nivel de usuario mediante librerías, cuando un thread realiza una llamada al sistema, otro thread del mismo proceso puede continuar la ejecución.
4. La técnica de “aging” se utiliza para disminuir las probabilidades de ocurrencia de starvation.¿Cómo se aplica esta técnica en el ambiente de planificación de CPU?

Threads

1. ¿Que es multithreading?
2. Explique las diferencias entre threads y procesos desde el punto de vista de la gestión de procesos.
3. ¿Por qué los threads proveen capacidad de paralelismo? Justifique desde el punto de vista del software y desde el punto de vista del hardware.
4. Sea un proceso P1 que crea cinco procesos hijos (P11, P12, P13, P14, P15), donde cada uno de ellos realiza una operación de suma de dos números. Por otra parte, se dispone de un proceso P2 con cinco threads, donde cada uno de ellos realiza la misma operación de suma que los procesos P11 a P15. ¿Qué proceso , P1 o P2, cree usted que finaliza más rápido y por qué?
5. Para los diferentes tipos de threads:
    a. Describa los threads a nivel de usuario (user-level threads, ULTs).
    b. Describa los threads a nivel de kernel (user-level threads, KLTs)
    c. En un sistema operativo sin soporte para KLTs ¿Cómo son soportados los ULTs?
    d. Para la relación entre ULTs y KLTs:

Sincronización de procesos:

1. ¿Qué es la condición de carrera (race condition)?
2. ¿Qué significa el término busy waiting (espera activa)? ¿Es posible evitarla? Justifique.
3. ¿Por qué las interrupciones no son apropiadas para implementar las primitivas de sincronización en sistemas multiprocesadores? ¿Y en sistemas monoprocesadores? Justifique.
4. Analice las siguientes oraciones y justifique su respuesta:
    a. Si las operaciones de semáforo wait( ) y signal( ) no se ejecutan atómicamente, entonces podría no cumplirse la exclusión mutua.
    b. La condición de carrera sólo puede ocurrir en sistemas con multiprocesadores.
5. ¿Qué diferencia existe entre deadlock y starvation?
6. ¿Cómo puede un sistema detectar si algunos de sus procesos están en starvation? ¿Cómo podría un sistema tratar ese problema?
7. ¿Es posible la existencia de un deadlock que involucre a solo un proceso? Justifique.