Sistemas Operativos I 2024
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. Esto de manera round-robin, por lo que una vez que cada proceso a, b y c tuvo su ráfaga, esta política de planificación de alto nivel vuelve a otorgar las ráfagas desde el inicio (dando una nueva ráfaga al proceso a, luego a b, 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.

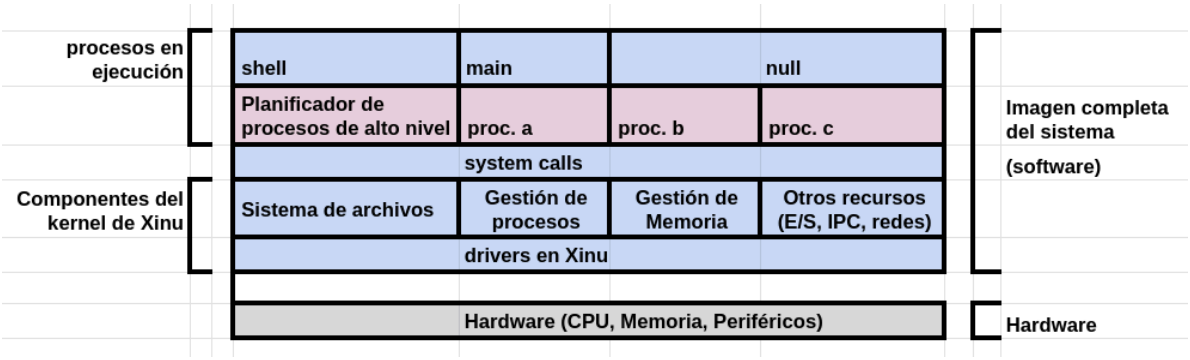

Figura 1. Componentes del sistema para este ejercicio

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). Utilice una pila de 4KB para cada proceso:

```
#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;
        }
}
```

Modificar la cantidad de pila al crear los procesos. Utilice una pila de 8KB para cada proceso creado. 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();
Una posible implementación es utilizar un semáforo binario. Igualmente, HAY UNA DIFERENCIA IMPORTANTE ENTRE UN SEMAFORO BINARIO Y UN MUTEX:

- En un semáforo binario, cualquier proceso puede realizar un signal() sobre el semáforo.
- En un mutex, el único proceso que puede realizar un mutex_unlock() es el proceso que realizó el mutex_lock().

c. Proteger las regiones críticas del programa original con este nuevo mecanismo de exclusión mutua.

Ejercicio 4. *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 5.

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?