AEC2 – Sistemas Operativos

ISMAEL HERNÁNDEZ CLEMENTE

DESCRIPCIÓN DE LA ACTIVIDAD PRÁCTICA

Esta actividad práctica de la unidad 2 consta de 2 partes:

1ª parte es un ejercicio de planificación
2º parte es un ejercicio de fork

1º Problema Planificación:

En un Sistema Operativo que utiliza el planificador Round Robin no expulsivo, con prioridad DINÁMICA: Un proceso que ha consumido 2 quantums de su prioridad original, ve rebajada en 1 la prioridad y es puesto al final de la cola. El cambio de contexto es instantáneo. Quantum = 5 para la prioridad 1, y quantum = 3 para la prioridad 2. Se presenta la siguiente combinación de procesos:

	A	B	C	D
Tarriv	0	1	5	12
Trun	6	4	11	6
Priority	2	2	1	1

A) Realizar la tabla de ejecución completa de todos los procesos (ver ejemplos donde además de la tabla se acompañan diagramas que ayudas. Incluirlos también)
B) Completar la siguiente tabla de tiempos, indicando para el proceso C:

Proceso	Llega	Trun	Inicio	Fin	Trespuesta	Tespera	Penalización ...
C	5	11	A	B	C	D	E

2º Ejercicio fork:

Se pide creatividad a los alumnos para proponer un ejercicio con forks diferente a los vistos en clase. Hay que crear el enunciado, la solución con código C y una explicación de que es lo que hace.

Se valorará la creatividad, dificultad, idoneidad, utilidad y claridad. Lo importante es poner el foco en los forks y no en códigos de C enrevesados.

EJERCICIO 1: PROBLEMA DE PLANIFICACIÓN

Para resolver este problema de planificación usando el algoritmo **Round Robin** con prioridades dinámicas, he estructurado los pasos para abordar cada aspecto que se nos pide. En mi explicación incluyo tanto la lógica del algoritmo como el proceso para armar la **tabla de ejecución completa** y la **tabla de tiempos de respuesta y espera**.

1. El Algoritmo

Como hemos visto en la unidad Round Robin no expulsivo esta caracterizado por un principio en el que cuando un proceso comienza a ejecutarse, se le asigna un intervalo de tiempo llamado quantum. El proceso continúa en ejecución hasta que consume su quantum o finaliza su tarea. Cuando el quantum se agota y el proceso no ha terminado, pasa al final de la cola de espera.

Aquí, la prioridad de un proceso puede cambiar durante la ejecución. Específicamente, si un proceso consume dos quantums completos sin finalizar, su prioridad se reduce en 1. Esto significa que en cada nuevo turno el proceso recibirá menos tiempo de CPU, si es que su prioridad disminuye. Además, cuando se reduce la prioridad, el proceso se mueve al final de la cola.

Quantum según prioridad:

- Prioridad 1: Quantum de 5 unidades de tiempo.
- Prioridad 2: Quantum de 3 unidades de tiempo.

Por lo tanto, los procesos con prioridad 2 tienen menos tiempo en cada turno que aquellos con prioridad 1.

2. Datos Iniciales

Con estos principios claros, procedo a la recopilación de los datos iniciales de procesos para poder aplicar la planificación:

- Procesos: A, B, C, D
- Tiempo de llegada (Tarriv): 0, 1, 5, 12 (este es el momento en que cada proceso entra al sistema y está listo para ejecutarse).
- Tiempo de ejecución total requerido (Trun) para completar su tarea: A=6, B=4, C=11, D=6.
- Prioridad inicial: A=2, B=2, C=1, D=1.

3. Tabla de Ejecución

Para la tabla de ejecución, tomé en cuenta la llegada de cada proceso y apliqué el algoritmo de Round Robin con prioridad dinámica. En cada paso:

- Se ejecuta el proceso disponible en el frente de la cola, en orden de llegada (Tarriv), y se aplica el quantum correspondiente según su prioridad.

- Si el proceso no termina después de dos quantums, su prioridad disminuye en 1 (si es posible) y se coloca al final de la cola.
- Actualizo el tiempo de ejecución restante y el quantum disponible en cada paso.

Tiempo	Proceso en Ejecución	Quantum Restante	Trun Restante	Prioridad Actual	Cola de Espera
0	A	3	6	2	B
3	B	3	4	2	C, A
6	C	5	11	1	D, A, B
11	C	5	6	1	A, B, D
16	D	5	6	1	A, B
21	A	3	3	2	B, C, D
24	B	3	1	2	C, D, A
27	C	5	3	1	D, A, B
32	D	5	1	1	A, B, C
36	A	3	0	2	B, C
39	B	3	0	2	C
42	C	5	0	1	

4. Componentes de la Tabla de Tiempos de Respuesta, Espera y Penalización

Llegada:

- Es el momento en que el proceso se vuelve elegible para ejecutar.
- En este caso, C llega al sistema en el tiempo 5.

Trun:

- El tiempo total que necesita para completar su tarea.
- Para C, Trun es 11 (es decir, necesita 11 unidades de tiempo de CPU para finalizar).

Inicio:

- El primer momento en que C empieza a ejecutarse.
- A pesar de que llega en el tiempo 5, el proceso B está en ejecución en ese momento, por lo que C comienza a ejecutarse en el tiempo 6.

Fin:

- El momento en que C finaliza completamente su ejecución.
- El último turno de C se completa en el tiempo 42.

Trespuesta (Tiempo de Respuesta):

- Es el tiempo que tarda un proceso en comenzar su primera ejecución desde que llega.
- En este caso, Trespuesta = 6 - 5 = 1.

Tespera (Tiempo de Espera):

- Es el tiempo total que un proceso pasa en la cola sin ejecutar.
- Para C, esto significa la suma de todos los períodos de espera entre cada turno de ejecución.
- Sumando estos períodos, el tiempo de espera total es 26 unidades de tiempo.

Penalización:

- La penalización es la medida de cuánto se ha extendido el tiempo total de finalización en relación con el tiempo de ejecución real.

$$\text{Penalización} = \frac{\text{Fin} - \text{Llega}}{\text{Trun}} = \frac{42 - 5}{11} = 3.36$$

5. Tabla de Tiempos de Respuesta Espera y Penalización

Para completar la segunda tabla, calculé cada valor específico para el proceso **C**.

Proceso	Llega	Trun	Inicio	Fin	Trespuesta	Tespera	Penalización
C	5	11	6	42	1	26	3.36

EJERCICIO 2: EJERCICIO DE FORKS

Para este proyecto, he decidido realizar una versión simplificada basada en la implementación completa de un proyecto completado anteriormente, llamado philosophers, basado en el problema de los filósofos comensales de Dijkstra, y que en su versión "bonus" plantea el reto de utilizar procesos y semáforos en vez hilos y mutexes. Debido a la complejidad del proyecto original por su extenso manejo de errores comunes de forks como los son los deadlocks, race conditions y errores de inanición (starvation), he decidido hacer una versión reducida en un solo archivo.c que replique el funcionamiento del proyecto original.

Nombre de programa	philo_bonus
Archivos a entregar	Makefile, *.h, *.c, en el directorio philo_bonus/
Makefile	NAME, all, clean, fclean, re
Argumentos	number_of_philosophers time_to_die time_to_eat time_to_sleep [number_of_times_each_philosopher_must_eat]
Funciones autorizadas	memset, printf, malloc, free, write, fork, kill, exit, pthread_create, pthread_detach, pthread_join, usleep, gettimeofday, waitpid, sem_open, sem_close, sem_post, sem_wait, sem_unlink
Se permite usar libft	No
Descripción	Philosophers con procesos y semaforos

Objetivos principales del proyecto:

1. Simulación de filósofos como procesos: Cada filósofo es un proceso independiente creado mediante fork, en lugar de ser un hilo, y el proceso principal actúa como controlador.
2. Sincronización con semáforos: Los semáforos se utilizan para gestionar la disponibilidad de tenedores en la mesa, permitiendo que cada filósofo los tome o los deje de manera sincronizada.
3. Ciclo de vida de los filósofos: Cada filósofo alterna entre los estados de comer, pensar y dormir, controlados por tiempos específicos (time_to_die, time_to_eat, time_to_sleep). Si un filósofo no come dentro de su time_to_die, muere, y la simulación termina o registra este evento.

Reglas específicas:

- Los tenedores están en el centro de la mesa y su disponibilidad está gestionada por un semáforo compartido.
- Cada filósofo es un proceso separado, lo que requiere coordinar a todos los procesos para sincronizar el acceso a los tenedores sin interferencias.

Enunciado Propuesto

Realice un programa que simule el problema de los filósofos comensales utilizando procesos en lugar de hilos. Cada filósofo debe ser un proceso independiente creado mediante fork y debe alternar entre los estados de pensar, comer y dormir. La sincronización entre los filósofos se logrará utilizando semáforos para gestionar el acceso a los tenedores, los cuales están en el centro de la mesa. El programa debe garantizar que cada filósofo respete el tiempo de vida (time_to_die), el tiempo para comer (time_to_eat) y el tiempo para dormir (time_to_sleep), y que la simulación termine si un filósofo muere por inanición.

Librerías autorizadas: stdio, stdlib, unistd, sys/wait, pthread, sys/time, semaphore, fcntl, signal, string.

Ejemplo de Uso y Capturas

Para la ejecución del programa he decidido ya que no añade excesiva complejidad el uso de un script de tipo Makefile ampliamente usado para el Debug de código ya que el programa será estructurado en un .c con el código y un .h para librerías, definición de colores y prototipados.

Comando: "make"

Comando limpiar: "make clean

Comando para recompilar y limpiar: "make re"

Uso del mini_philosophers: una vez compilado añadir al ejecutables a.out los siguientes parámetros:
número de filósofos que participarán en el simulacro del problema de los filósofos.

- tiempo en milisegundos que cada filósofo dedicará a pensar.
- tiempo en milisegundos que cada filósofo pasará comiendo.
- tiempo en milisegundos que cada filósofo descansará (durmiendo).
- Opcional número de ciclos de comidas que se efectuarán.

Inicio de la ejecución ejemplo A:

Comando Ejecutado: "./a.out 5 500 500 500"

Esto tiene cuatro partes:

- 5: El número de filósofos que participarán en el simulacro del problema de los filósofos.
- 500: El tiempo en milisegundos que cada filósofo dedicará a pensar.
- 500: El tiempo en milisegundos que cada filósofo pasará comiendo.
- 500: El tiempo en milisegundos que cada filósofo descansará (durmiendo).

```
nirmata@nirmata-pc:~/U-Tad/2nd_Course/Q1/Operating_systems/Unit_2/AEC2$ ./a.out 5 500 500 500
0 1 has taken a fork
0 1 is eating
0 2 has taken a fork
0 2 is eating
500 1 is sleeping
500 3 has taken a fork
500 3 is eating
500 2 is sleeping
500 4 has taken a fork
500 4 is eating
```

Final de la ejecución ejemplo A:

Comidas realizadas:

- Filósofo 0 ha comido dos veces (ya que se muestra como "eating" dos veces).
- Filósofo 3 ha comido una vez.
- Filósofo 4 ha comido una vez.

Por lo tanto, hubo 4 comidas en total.

```
500 2 is sleeping
500 4 has taken a fork
500 4 is eating
```

Inicio de la ejecución ejemplo B:

Desglose de los parámetros:

1. 5: Número de filósofos (en este caso, 5 filósofos participarán en la simulación).
2. 600: Tiempo de comer (600 microsegundos).
3. 200: Tiempo de dormir (200 microsegundos).
4. 200: Tiempo de pensar (200 microsegundos).
5. 10: Número de ciclos de comida que cada filósofo debe completar.

Explicación:

- Los 5 filósofos comenzarán su ciclo de actividades (pensar, comer y dormir) de acuerdo con los tiempos definidos en los parámetros.
- Cada filósofo comerá por 600 microsegundos, luego dormirá por 200 microsegundos y pensará durante 200 microsegundos.
- Cada filósofo repetirá este ciclo 10 veces antes de terminar su ejecución.

Este comando simulará la ejecución con tiempos más reducidos en comparación con los valores anteriores, y la salida mostrará el estado de los filósofos en cada etapa (pensando, comiendo, durmiendo) durante el transcurso de los ciclos.

```
nirmata@nirmata-pc:~/U-Tad/2nd_Course/Q1/Operating_systems/Unit_2/AEC2$ ./a.out 5 600 200 200 10
0 1 has taken a fork
0 1 is eating
0 2 has taken a fork
0 2 is eating
200 1 is sleeping
200 3 has taken a fork
200 3 is eating
200 2 is sleeping
200 5 has taken a fork
200 5 is eating
400 1 is thinking
400 3 is sleeping
400 5 is sleeping
400 1 has taken a fork
400 1 is eating
400 4 has taken a fork
401 4 is eating
401 2 is thinking
601 2 died
```

Total, de comidas:

Sumando todas las comidas:

- Filósofo 0: 2 comidas
- Filósofo 3: 1 comida
- Filósofo 4: 1 comida
- Filósofo 5: 1 comida

Total de comidas = 5 comidas.

Resumen:

En esta ejecución, hubo 5 comidas en total. Sin embargo, los filósofos tuvieron tiempos diferentes para comer, y la sincronización de los semáforos permitió que algunos filósofos comieran más veces que otros. El filósofo 2 murió antes de completar su ciclo, por lo que no llegó a comer más veces.