

Práctica 4 - ISO

1) Responda en forma sintética sobre los siguientes conceptos:

(a) Programa y Proceso.

- Programa: Es estático (no va cambiando una vez que ya está escrito), reside en disco y no tiene program counter. Existe desde que se edita hasta que se borra
- Proceso: Es dinámico (se puede cargar y descargar de la memoria), tiene program counter (para saber en qué instrucción se quedó la CPU en su último ciclo de ejecución). Se le puede asignar y sacar tiempo de CPU. Su ciclo de vida comprende desde que se lo ejecuta hasta que termina.

(b) Defina Tiempo de retorno (TR) y Tiempo de espera (TE) para un Job

Los procesos tienen distintos en su ciclo de vida

- Tiempo de Retorno: tiempo que transcurre entre que el proceso llega al sistema hasta que completa su ejecución. (todo el tiempo de vida del proceso. Desde que se crea -new ready- hasta -terminated-)
- Tiempo de espera: Tiempo que el proceso se encuentra en el sistema esperando, es decir el tiempo que pasa sin ejecutarse (TR - Tcpu). El tiempo que estuvo esperando en la cola de ready, esperando que sea seleccionado por el short term scheduler para ser ejecutado.

(c) Defina Tiempo Promedio de Retorno (TPR) y Tiempo promedio de espera (TPE) para un lote de JOBS.

Los promedios son importantes de obtener para saber que algoritmo nos conviene usar

- Tiempo Promedio de Retorno (TPR): Es el promedio del tiempo total que cada JOB tarda desde que llega al sistema hasta que termina (incluye espera y ejecución).
Fórmula: -----
- Tiempo Promedio de Espera (TPE): Es el promedio del tiempo que cada JOB pasa esperando en la cola antes de ser ejecutado.
Fórmula: -----

(d) ¿Qué es el Quantum?

El quantum mide lo que determina cuánto tiempo podrá usar el procesador cada proceso.

(e) ¿Qué significa que un algoritmo de scheduling sea apropiativo o no apropiativo (Preemptive o Non-Preemptive)?

- No apropiativos: Una vez que un proceso está en estado de ejecución, continúa hasta que termina o se bloquea por algún evento (ej: I/O). Es decir: una vez que se selecciona un proceso, se empieza a ejecutar (está en estado running), va a quedarse en estado running

y ningún otro evento de que llegue otro proceso con más prioridad lo va a poder sacar. Se va a ejecutar hasta el final sin importar lo que suceda. o hasta que llegue un evento de I/O o sea, que el proceso no termine sin que se atienda lo de I/O, entonces ahí si abandona la CPU, si no se ejecutaría hasta el final.

- Apropiativos: El proceso en ejecución puede ser interrumpido y llevado a la cola de listos:
 - Mayor overhead pero mejor servicio
 - Un proceso no monopoliza el procesador

(f) ¿Qué tareas realizan?:

i. Short Term Scheduler

Admite nuevos procesos a memoria (controla el grado de multiprogramación).

ii. Long Term Scheduler

Realiza el swapping (intercambio) entre el disco y la memoria cuando el SO lo determina (puede disminuir el grado de multiprogramación).

iii. Medium Term Scheduler

Determina qué proceso pasará a ejecutarse.

(g) ¿Qué tareas realiza el Dispatcher?

Es el encargado de realizar el cambio de contexto, control de quantum, gestión de estados (Listo, Ejecutando, Bloqueado), manejo de Interrupciones y optimiza la CPU.

2) Procesos:

(a) Investigue y detalle para que sirve cada uno de los siguientes comandos. (Puede que algún comando no venga por defecto en su distribución por lo que deberá instalarlo):

i. top

Muestra en tiempo real los procesos en ejecución, su consumo de recursos (CPU, memoria, etc.) y otras métricas de rendimiento del sistema.

ii. htop

Similar a top, pero con una interfaz interactiva y más amigable, que permite realizar acciones como matar procesos con solo seleccionarlos.

iii. ps

Muestra una lista de procesos en ejecución en el momento de la ejecución del comando. Es útil para ver información específica de procesos sin actualizaciones en tiempo real.

iv. pstree

Muestra los procesos en ejecución en forma de árbol, mostrando las relaciones jerárquicas entre ellos (procesos padres e hijos).

v. kill

Envía una señal a un proceso específico para terminarlo (generalmente señal SIGTERM o SIGKILL). Se usa junto con el ID del proceso (PID).

vi. pgreppkillkillall

Busca procesos que coincidan con un nombre o patrón especificado y devuelve sus PIDs, útil para identificar procesos específicos en ejecuciones en serie.

vii. killall

Mata todos los procesos que coinciden con un nombre o patrón especificado. Es útil cuando se quiere terminar varios procesos relacionados con un solo comando.

viii. renice

Cambia la prioridad de un proceso en ejecución, ajustando su "niceness" para que tenga más o menos acceso a la CPU en relación con otros procesos.

ix. xkill

Permite finalizar aplicaciones gráficas de manera interactiva. Al ejecutarlo, el cursor se convierte en una "X", y al hacer clic en una ventana, se cierra el programa asociado.

x. atop

Monitorea el rendimiento del sistema, similar a top, pero con una mayor cantidad de detalles sobre recursos como disco, red, CPU, y memoria. También permite ver estadísticas históricas.

(b) Observe detenidamente el siguiente código. Intente entender lo que hace sin necesidad de ejecutarlo

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
    int c;
    pid_t pid;
    printf("Comienzo.:\n");
    for (c = 0; c < 3 ; c++ )
    {
        pid = fork();
    }
    printf("Proceso\n");
    return 0;
}
```

i. ¿Cuántas líneas con la palabra "Proceso" aparecen al final de la ejecución de este programa?.

El programa imprime 8 líneas con la palabra proceso

ii. ¿El número de líneas es el número de procesos que han estado en ejecución?. Ejecute el programa y compruebe si su respuesta es correcta, Modifique el valor del bucle for y compruebe los nuevos resultados.

No, el número de líneas impresas no necesariamente representa el número total de procesos que han estado en ejecución durante la vida del programa. Sin embargo, en este caso particular, el programa está diseñado para que cada proceso

creado imprima "Proceso" antes de terminar, por lo que **el número de líneas impresas es igual al número de procesos activos al final del bucle.**

c) Vamos a tomar una variante del programa anterior. Ahora, además de un mensaje, vamos a añadir una variable y, al final del programa vamos a mostrar su valor. El nuevo código del programa se muestra a continuación.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void) {
    int c;
    int p=0;
    pid_t pid;
    for (c = 0; c < 3 ; c++ )
    {
        pid = fork();
    }
    p++;
    printf("Proceso %d\n", p);
    return 0;
}
```

i. ¿Qué valores se muestran por consola?

Por consola se mostrarán exactamente **8 líneas** con el mismo valor:

Copiar código

```
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
Proceso 1
```

Esto se debe a que cada uno de los 8 procesos generados incrementa su propia copia de **p** y la imprime, pero **ninguno comparte ni modifica el valor de p de los demás procesos.**

(d) Comunicación entre procesos:

i. Investigue la forma de comunicación entre procesos a través de pipes.

Los pipes son un mecanismo de comunicación entre procesos en sistemas UNIX/Linux que permite el intercambio de datos de un proceso a otro utilizando un canal en memoria. Se implementan con la función pipe()

ventajas de los pipes:

- Simplicidad para comunicación rápida entre procesos
- No requiere configuraciones complicadas

Limitaciones:


- Solo funciona para procesos relacionados
- Es unidireccional (a menos que se creen dos pipes para flujo bidireccional)
- No se puede usar para comunicación entre procesos en máquinas diferentes.

ii. ¿Cómo se crea un pipe en C?.

Pasos para crear y usar un pipe:


1. **Declarar el pipe:** Define un array para los descriptores del pipe:

```
c
int pipefd[2];
```

 Copiar código

2. **Crear el pipe:** Llama a `pipe()` e identifica errores:

```
c
if (pipe(pipefd) == -1) {
    perror("Error al crear el pipe");
    return 1;
}
```

 Copiar código

iii. ¿Qué parametro es necesario para la creación de un pipe?.

Explique para que se utiliza

La función `pipe()` requiere un único parámetro: un array de dos enteros, definido como `int pipefd[2]`

Este array sirve para almacenar los descriptores de archivo que representan los extremos del pipe. El sistema operativo los asigna al llamar `pipe()`. Cada extremo tiene una función específica:

- `pipefd[0]`: Es el descriptor para leer datos del pipe. El proceso que lee usa este extremo
- `pipefd[1]`: Es el descriptor para escribir datos en el pipe. El proceso que escribe usa este extremo.

iv. ¿Qué tipo de comunicación es posible con pipes?

- Comunicación unidireccional: significa que los datos fluyen en una sola dirección.

Un proceso escribe en un extremo y otro proceso lee desde el otro extremo. Se utiliza cuando un proceso necesita enviar datos a otro de manera secuencial

- Comunicación bidireccional: Para lograr comunicación en ambas direcciones, se necesitan dos pipes:

Uno para enviar datos desde el proceso A al proceso B

Otro para enviar datos desde el proceso B al proceso A

Se debe tener cuidado al cerrar los extremos adecuados en cada proceso

- Comunicación entre procesos relacionados: Los pipes están diseñados para procesos relacionados, como un proceso padre y sus hijos creados con fork()
Ambos procesos heredan los descriptores del pipe y pueden usarlos para comunicarse
- Comunicación síncrona: Si el lector intenta leer antes de que haya datos disponibles, se bloquea hasta que el escritor escriba algo
Si el escritor intenta escribir en un pipe lleno, se bloquea hasta que el lector consuma datos.

e) ¿Cuál es la información mínima que el SO debe tener sobre un proceso? ¿En que estructura de datos asociada almacena dicha información?

El sistema operativo necesita mantener información crítica sobre cada proceso para administrarlo de manera eficiente. Esta información mínima almacena en una estructura de datos conocida como PCB (Process Control Block) o Bloque de control de proceso

Información almacenada en el PCB:

1. Identificación del proceso (PID)
Un identificador único para diferenciar el proceso de otros
2. Estado del proceso:
Indica si el proceso está listo, en ejecución, bloqueado o terminado
3. Contador de programa (PC, Program Counter):
Dirección de la próxima instrucción a ejecutar para el proceso.
4. Registros del CPU:
Incluyen los registros generales, de estado y específicos que se deben guardar/restaurar durante los cambios de contexto
5. Información de memoria:
Detalles sobre el espacio de memoria asignado al proceso: Dirección base y límite. Tabla de páginas (en sistemas con memoria virtual)
6. Archivos abiertos:
Lista de descriptores de archivo que el proceso está utilizando.
7. Información de planificación:
Prioridad del proceso

Tiempos de ejecución y espera

Punteros a estructuras en la cola de planificación

8. Información de entrada/salida(E/S)

Recursos de E/S asignados y operaciones pendientes

9. Información de comunicación entre procesos(IPC):

Datos necesarios para sincronización y comunicación(pipes, mensajes, señales, etc)

(f) ¿Qué significa que un proceso sea “CPU Bound” y “I/O

Bound”?

a. **CPU Bound:**

Un proceso CPU Bound es aquel que pasa la mayor parte de su tiempo realizando cálculos intensivos en la CPU

Su rendimiento está limitado por la velocidad del procesador

Ejemplos:

→ Procesamiento de datos intensivo (Cálculos matemáticos complejos)

→ Simulaciones científicas o análisis de datos masivos

Características:

Consume mucho tiempo del procesador

Realiza pocas operaciones de entrada/salida

El tiempo de espera para recursos I/O es insignificante

b. **I/O Bound**

Un proceso I/O Bound es aquel que pasa más tiempo esperando operaciones de entrada/salida que utilizando la CPU

Su rendimiento está limitado por la velocidad de los dispositivos de entrada/salida (disco, red, etc)

Ejemplos:

→ Lectura/escritura de archivos en disco

→ Consultas a bases de datos

→ Operaciones de red (descargas o transferencias)

Características:

Consume menos tiempo de CPU

Realiza muchas operaciones de entrada/salida

Pasa más tiempo en estado de bloqueo, esperando que se completen las operaciones I/O

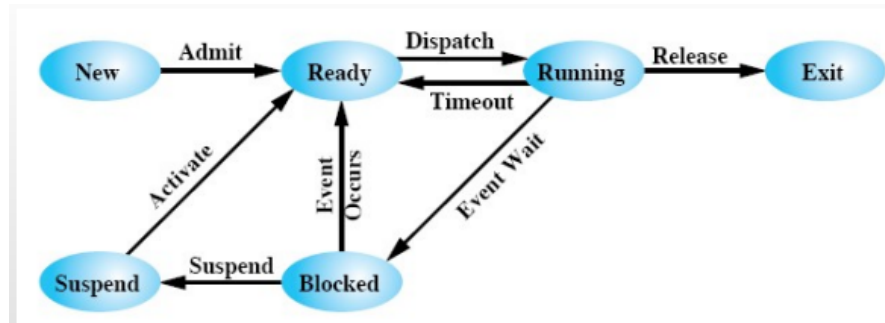
(g) ¿Cuáles son los estados posibles por los que puede

atravesar un proceso?

- New

- Ready
- Running
- Blocked
- Exit

(h) Explique mediante un diagrama las posibles transiciones entre los estados.



(i) ¿Que scheduler de los mencionados en 1 f se encarga de las transiciones?

El scheduler de corto plazo es el encargado de gestionar las transiciones de los estados de los procesos en un sistema operativo

3. Para los siguientes algoritmos de scheduling:

- FCFS (First Come First Served)
- SJF (Shortest Job First)
- Round Robin
- Prioridades

(a) Explique su funcionamiento mediante un ejemplo

(b) ¿Alguno de ellos requiere algún parámetro para su funcionamiento?

(c) Cual es el mas adecuado según los tipos de procesos y/o SO.

(d) Cite ventajas y desventajas de su uso

FCFS (First Come First Served)

Cuando hay que elegir un proceso para ejecutar, se selecciona el mas viejo.

No favorece a ningun tipo de proceso en articular, pero en principio los CPU Bound terminan al comenzar su primer rafaga, mientras que los I/O Bound no.

Ventajas: Es un algoritmo simple de implementar, se pueden predecir lo procesos a ejecutar, eventualmente todos los procesos de ejecutar por lo que no produce inanición y es eficiente para cargas de trabajo simple.

Desventajas: No considera el tiempo de ejecución de un proceso, el promedio de tiempo de espera puede ser elevado,

posee poca flexibilidad y es ineficiente para cargas de trabajo mixtas.

SJF (Shortest Job First)

Política no apropiativa que selecciona el proceso con la ráfaga mas corto, calculando dicha ráfaga basado en la ejecución previa.

Necesita el tiempo de la próxima ráfaga mas corta.

Procesos con ráfagas largas sufren inanición y prioriza procesos cortos.

Ventajas: Menor tiempo de espera promedio, eficiencia en entornos donde hay una mezcla de procesos cortos y largos, no posee inanición si se aplica a SO con procesos cortos.

Desventajas: Dificultad para conocer el tiempo de ejecución de un proceso, produce inanición en SO por procesos largos, posee mucha complejidad de implementación y no se adapta correctamente si se cambian los tiempos de ejecución de los procesos.

Round Robin

Política basada en un reloj, que determina cuanto tiempo podra usar el procesador cada proceso. Cuando un proceso es expulsado de la CPU es colocado al final de la Ready Queue y se selecciona otro (FIFO circular).

Necesita un quantum.

Este algoritmo es adecuado para SO con muchos procesos I/O Bound y que requieran interacción con el usuario.

Ventajas: garantiza que todos los procesos obtienen una porción equitativa de tiempo de CPU, bajo tiempo de respuestas, simplicidad y adaptable a cargas de tareas variables.

Desventajas: Overhead por cambios de contexto y la elección del tamaño del quantum es crucial.

Prioridades

Cada proceso tiene un valor que representa su prioridad, a menor valor, mayor prioridad. Se selecciona el proceso de mayor prioridad de los que se encuentran en la Ready Queue

Necesita el orden de prioridad del proceso.

Ideal para SO con procesos críticos, donde es necesario que los procesos sean atendidos lo mas rápido posible.

Ventajas: Permite que procesos de alta prioridad obtengan acceso inmediato a la CPU

Desventajas: Posibilidad muy latente de inanición, complejidad de administración con prioridades dinámicas, evaluar y reorganizar la cola de procesos en función de sus prioridades genera overhead y ineficiencia para procesos equivalentes