TECNOLÓGICO NACIONAL DE MÉXICO

INSTITUTO TECNOLÓGICO DE TLAXIACO

PRÁCTICA 3.- Unidad Funcionamiento de una CPU - Simulación de administración de procesos/tareas en la CPU

Presentan:
Hernández Martínez Adriana -22620083

Carrera:
Ingeniería en Sistemas Computacionales.

Asignatura:
Arquitectura de computadoras

Semestre:
5BS

Docente:
Ing. Edward Osorio Salinas

Tlaxiaco, Oaxaca, a 29 de octubre de 2024.

Contenido

2. OBJETIVO 1

3. Resumen de los pasos que se llevan a cabo en la administración de procesos/tareas en la CPU. 1

4. DESARROLLO 2

4.1. Estrategias de scheduling. 2

4.2. [Estrategia 1] Simulación de administración de procesos/tareas en la CPU. 3

- **4.2.1. Capturas de pantalla del simulador** 3

4.2.2. Descripción de las etapas de la administración de procesos/tareas. 4

4.3. [Estrategia 2] Simulación de administración de procesos/tareas en la CPU. 5

4.3.1. Capturas de pantalla del simulador. 5

4.3.2. Descripción de las etapas de la administración de procesos/tareas. 6

4.4. [Estrategia 3] Simulación de administración de procesos/tareas en la CPU. 7

4.4.1. Capturas de pantalla del simulador. 7

4.4.2. Descripción de las etapas de la administración de procesos/tareas. 10

5. Conclusiones. 11

6.Referencias. 12

2. OBJETIVO

- El alumno simule la administración de procesos/tareas en la CPU.

3. Resumen de los pasos que se llevan a cabo en la administración de procesos/tareas en la CPU.

- **Creación del Proceso**: Se crea el proceso y se le asignan recursos iniciales, como un identificador (ID) único y espacio en memoria.
- **Planificación del Proceso**: La CPU decide el orden de ejecución de los procesos, organizándolos en colas de espera y asignando prioridad según políticas (como Round Robin o First Come First Served).
- **Cambio de Contexto**: Cuando la CPU cambia de un proceso a otro, guarda el estado actual del proceso en ejecución y restaura el estado del proceso siguiente. Esto permite que varios procesos compartan la CPU.
- **Ejecución del Proceso**: La CPU ejecuta el proceso seleccionado hasta que se completa su tiempo de CPU asignado o se produce una interrupción.
- **Manejo de Interrupciones**: Ante una interrupción, la CPU pausa el proceso actual para atender la solicitud, ya sea de entrada/salida o por prioridades.
- **Finalización del Proceso**: Una vez que el proceso termina, el sistema operativo libera sus recursos, dejándolos disponibles para otros procesos.

4. DESARROLLO

4.1. Estrategias de scheduling.

1. **First Come First Served (FCFS):** Los procesos se ejecutan en el orden en el que llegan a la CPU, lo cual es simple, pero puede llevar a largos tiempos de espera si los primeros procesos son largos.
2. **Shortest Job Next (SJN):** La CPU ejecuta los procesos en orden ascendente de acuerdo a su tiempo de ejecución estimado, reduciendo el tiempo de espera promedio, aunque puede llevar a la inanición de procesos largos.
3. **Round Robin (RR):** Cada proceso recibe un tiempo fijo de CPU (quantum) en orden circular. Cuando el quantum se agota, el proceso regresa a la cola, lo cual permite la ejecución justa y periódica de todos los procesos.
4. **Priority Scheduling:** Los procesos son atendidos en función de su prioridad asignada; los procesos de mayor prioridad tienen preferencia. Los de menor prioridad pueden sufrir inanición si no se gestiona adecuadamente.
5. **Multilevel Queue Scheduling:** Los procesos son asignados a diferentes colas según su prioridad, con cada cola teniendo una política de planificación diferente. Permite manejar distintos tipos de procesos de manera eficiente.
6. **Multilevel Feedback Queue Scheduling:** Similar a Multilevel Queue, pero permite que los procesos cambien de cola según su comportamiento y uso de CPU. Se ajusta dinámicamente, promoviendo la justicia y eficiencia.
7. **Real Time Scheduling:** Prioriza procesos con restricciones de tiempo específicas, ejecutando aquellos que deben cumplir con plazos críticos. Es común en sistemas donde los procesos deben completarse en un tiempo determinado.
8. **Multicore Scheduling:** Distribuye los procesos entre múltiples núcleos de la CPU, aumentando la eficiencia al paralelizar la ejecución de procesos.
9. **Multitasking:** Permite la ejecución concurrente de varios procesos, asignando un quantum de tiempo a cada uno, lo que da la impresión de que todos los procesos se ejecutan al mismo tiempo.

4.2. [Estrategia 1] Simulación de administración de procesos/tareas en la CPU.

- ### 4.2.1. Capturas de pantalla del simulador.

```
print("FIRST COME FIRST SERVE SCHEDULLING")
n= int(input("Enter number of processes : "))
d = dict()

for i in range(n):
    key = "P"+str(i+1)
    a = int(input("Enter arrival time of process"+str(i+1)+": "))
    b = int(input("Enter burst time of process"+str(i+1)+": "))
    l = []
    l.append(a)
    l.append(b)
    d[key] = l

d = sorted(d.items(), key=lambda item: item[1][0])

ET = []
for i in range(len(d)):
    # first process
    if(i==0):
        ET.append(d[i][1][1])
```

```
    # get prevET + newBT
    else:
        ET.append(ET[i-1] + d[i][1][1])

TAT = []
for i in range(len(d)):
    TAT.append(ET[i] - d[i][1][0])

WT = []
for i in range(len(d)):
    WT.append(TAT[i] - d[i][1][1])

avg_WT = 0
for i in WT:
    avg_WT +=i
avg_WT = (avg_WT/n)

print("Process | Arrival | Burst | Exit | Turn Around | Wait |")
for i in range(n):
      print("   ",d[i][0],"   |   ",d[i][1][0]," |    ",d[i][1][1]," |
print("Average Waiting Time: ",avg_WT)
```

```
FIRST COME FIRST SERVE SCHEDULLING

Enter number of processes : 4
Enter arrival time of process1: 1
Enter burst time of process1: 5
Enter arrival time of process2: 0
Enter burst time of process2: 4
Enter arrival time of process3: 3
Enter burst time of process3: 3
Enter arrival time of process4: 2
Enter burst time of process4: 5

Process | Arrival | Burst | Exit | Turn Around | Wait |
    P2     |     0   |      4   |      4    |      4    |     0     |
    P1     |     1   |      5   |      9    |      8    |     3     |
    P4     |     2   |      5   |      14    |      12    |     7     |
    P3     |     3   |      3   |      17    |      14    |     11     |
Average Waiting Time:  5.25
```

4.2.2. Descripción de las etapas de la administración de procesos/tareas.

- **Ingreso en la cola de procesos**: Los procesos se agregan a una cola de espera en el mismo orden en el que llegan al sistema. No se les asigna ninguna prioridad especial, sino que se manejan de manera secuencial.
- **Asignación a la CPU**: La CPU selecciona el proceso en la parte frontal de la cola y lo ejecuta hasta completarlo. No se interrumpe el proceso, lo que significa que la CPU permanece asignada a ese proceso hasta que finaliza.
- **Ejecución del proceso**: El proceso ocupa la CPU durante todo su tiempo de ejecución. No existe un "quantum" o límite de tiempo como en Round Robin; por lo tanto, los procesos de larga duración pueden causar esperas prolongadas para los procesos que llegaron después.
- **Terminación y liberación de la CPU**: Una vez que el proceso termina, la CPU se libera y pasa al siguiente proceso en la cola. La cola avanza y el proceso completado sale del sistema.

- **Actualización de la cola**: La cola de procesos se actualiza eliminando el proceso finalizado y avanzando al siguiente en la fila. Esta estructura se repite para cada proceso nuevo que ingresa al sistema.

4.3. [Estrategia 2] Simulación de administración de procesos/tareas en la CPU.

4.3.1. Capturas de pantalla del simulador.

```python
class Process:
    def __init__(self, pid, burst_time):
        self.pid = pid
        self.burst_time = burst_time

def sjf_scheduling(processes):
    processes.sort(key=lambda x: x.burst_time)  # Sort processes by burst time
    total_processes = len(processes)
    current_time = 0
    waiting_time = 0

    print("Process ID\tBurst Time\tWaiting Time")
    for p in processes:
        print(f"{p.pid}\t\t{p.burst_time}\t\t{waiting_time}")
        waiting_time += current_time
        current_time += p.burst_time

    average_waiting_time = waiting_time / total_processes
    print("\nAverage Waiting Time:", average_waiting_time)

# Example usage
if __name__ == "__main__":
    processes = [Process(1, 6), Process(2, 8), Process(3, 7), Process(4, 3)]
    sjf_scheduling(processes)
```

```
Identificación del proceso Tiempo de ráfaga Tiempo de espera
4 3 0
1 6 0
3 7 3
2 8 12

Tiempo de espera promedio: 7.0
```

4.3.2. Descripción de las etapas de la administración de procesos/tareas.

- **Ingreso en la cola de procesos**: Los procesos llegan al sistema y se colocan en una cola de espera. Cada proceso se evalúa para determinar su tiempo estimado de ejecución, que será el criterio principal para su orden de ejecución.
- **Selección del proceso más corto**: La CPU selecciona el proceso con el tiempo de ejecución más corto entre todos los procesos en la cola de espera. Si un proceso más corto llega mientras otro proceso está en ejecución (en la versión *preemptive* de SJN), el proceso en ejecución puede ser interrumpido para dar prioridad al nuevo proceso más corto. En la versión *non-preemptive*, el proceso seleccionado se ejecuta hasta completarse.
- **Asignación a la CPU y ejecución**: Una vez seleccionado, el proceso más corto recibe la CPU y se ejecuta hasta completarse, si se utiliza la versión *non-preemptive*. En la versión *preemptive*, la CPU puede reasignarse a un proceso más corto que llegue después.
- **Terminación y liberación de la CPU**: Al finalizar el proceso, la CPU se libera para el siguiente proceso más corto en la cola. El proceso completado se retira de la cola y se marca como terminado.
- **Actualización de la cola**: La cola se actualiza reordenando los procesos restantes en función de su tiempo de ejecución. Esta organización asegura que el siguiente proceso más corto siempre tenga la prioridad para ejecutarse.

4.4. [Estrategia 3] Simulación de administración de procesos/tareas en la CPU.

4.4.1. Capturas de pantalla del simulador.

```
class Process:
    def __init__(self):
        self.pid = 0
        self.arrivalTime = 0
        self.burstTime = 0
        self.burstTimeRemaining = 0
        self.completionTime = 0
        self.turnaroundTime = 0
        self.waitingTime = 0
        self.isComplete = False
        self.inQueue = False

# At every time quantum or when a process has been executed before the time quantum,
# check for any new arrivals and push them into the queue
def check_for_new_arrivals(processes, n, current_time, ready_queue):
    for i in range(n):
        p = processes[i]
        # checking if any processes has arrived
        # if so, push them in the ready Queue.
        if p.arrivalTime <= current_time and not p.inQueue and not p.isComplete:
            processes[i].inQueue = True
            ready_queue.append(i)

# Context switching takes place at every time quantum
# At every iteration, the burst time of the processes in the queue are handled using this
def update_queue(processes, n, quantum, ready_queue, current_time, programs_executed):
    i = ready_queue[0]
    ready_queue.pop(0)

    # if the process is going to be finished executing,
    # ie, when it's remaining burst time is less than time quantum
```

```
        # mark it completed and increment the current time
        # and calculate its waiting time and turnaround time
        if processes[i].burstTimeRemaining <= quantum:
            processes[i].isComplete = True
            current_time += processes[i].burstTimeRemaining
            processes[i].completionTime = current_time
            processes[i].waitingTime = processes[i].completionTime - processes[i].arriva
            processes[i].turnaroundTime = processes[i].waitingTime + processes[i].burstT

            if processes[i].waitingTime < 0:
                processes[i].waitingTime = 0

            processes[i].burstTimeRemaining = 0

            # if all the processes are not yet inserted in the queue,
            # then check for new arrivals
            if programs_executed != n:
                check_for_new_arrivals(processes, n, current_time, ready_queue)
        else:
            # the process is not done yet. But it's going to be pre-empted
            # since one quantum is used
            # but first subtract the time the process used so far
            processes[i].burstTimeRemaining -= quantum
            current_time += quantum

            # if all the processes are not yet inserted in the queue,
            # then check for new arrivals
            if programs_executed != n:
                check_for_new_arrivals(processes, n, current_time, ready_queue)
            # insert the incomplete process back into the queue
            ready_queue.append(i)

    # Just a function that outputs the result in terms of their PID.
    def output(processes, n):
        avg_waiting_time = 0
        avg_turntaround_time = 0
```

```
    avg_turntaround_time = 0
    # sort the processes array by processes.PID
    processes.sort(key=lambda p: p.pid)

    for i in range(n):
        print("Process ", processes[i].pid, ": Waiting Time: ", processes[i].waitingTi
              " Turnaround Time: ", processes[i].turnaroundTime, sep="")
        avg_waiting_time += processes[i].waitingTime
        avg_turntaround_time += processes[i].turnaroundTime
    print("Average Waiting Time: ", avg_waiting_time / n)
    print("Average Turnaround Time: ", avg_turntaround_time / n)

# This function assumes that the processes are already sorted according to their arriv
def round_robin(processes, n, quantum):
    ready_queue = []
    ready_queue.append(0) # initially, pushing the first process which arrived first
    processes[0].inQueue = True

    current_time = 0 # holds the current time after each process has been executed
    programs_executed = 0 # holds the number of programs executed so far

    while len(ready_queue) != 0:
        update_queue(processes, n, quantum, ready_queue, current_time, programs_execut

def main():
    n = int(input("Enter the number of processes: "))
    quantum = int(input("Enter time quantum: "))

    processes = []

    for i in range(n):
        print("Enter arrival time and burst time of each process ", i + 1, ": ", sep="
        arrival_time = int(input())
        burst_time = int(input())
        proc = Process()
        proc.arrivalTime = arrival_time




        proc.burstTime = burst_time
        proc.burstTimeRemaining = burst_time
        proc.pid = i + 1
        processes.append(proc)
        print("")

    # stl sort in terms of arrival time
    processes.sort(key=lambda p: p.arrivalTime)

    round_robin(processes, n, quantum)

    output(processes, n)

main()

# This code is contributed by akashish__
```

```
Enter the arrival time and burst time of each process:
0 5
1 4
2 2
3 1Enter the number of processes: 4

Enter time quantum: 2



Process 1: Waiting Time: 7 Turnaround Time: 12
Process 2: Waiting Time: 6 Turnaround Time: 10
Process 3: Waiting Time: 2 Turnaround Time: 4
Process 4: Waiting Time: 5 Turnaround Time: 6

Average Waiting Time: 5
Average Turnaround Time: 8
```

4.4.2. Descripción de las etapas de la administración de procesos/tareas.

- **Ingreso en la cola de procesos**: Los procesos se agregan a una cola de espera en el orden en que llegan. No se les asigna prioridad especial, ya que la estrategia RR busca la equidad en el tiempo de CPU.
- **Asignación del quantum de CPU**: La CPU asigna el *quantum* (una cantidad fija de tiempo) al primer proceso en la cola. Este proceso tiene hasta ese tiempo para ejecutarse; si termina antes, la CPU pasa al siguiente proceso sin agotar el *quantum*.
- **Ejecución del proceso durante el quantum**: El proceso se ejecuta por el tiempo del *quantum*. Si no finaliza en este periodo, se pausa su ejecución temporalmente y se coloca al final de la cola, de modo que pueda continuar en el siguiente ciclo.

- **Liberación de la CPU y actualización de la cola**: Si el proceso termina dentro de su *quantum*, se retira de la cola de espera y la CPU se asigna al siguiente proceso. Si no termina, se mueve al final de la cola para esperar su próximo turno.
- **Rotación continua**: La CPU continúa asignándose a cada proceso en orden circular, garantizando que todos los procesos reciban una parte justa del tiempo de CPU. Este ciclo se repite hasta que todos los procesos en la cola hayan completado su ejecución.

5. CONCLUSIONES.

Esta práctica me permitió comprender de manera práctica cómo se administra el procesamiento de tareas en una CPU, explorando diversas estrategias de scheduling que buscan optimizar el uso de los recursos de procesamiento. La simulación me ayudó a visualizar cómo el sistema operativo decide el orden de ejecución de los procesos según diferentes criterios, desde la simplicidad de ejecutar en orden de llegada (FCFS) hasta enfoques más complejos como el Multilevel Feedback Queue Scheduling, que ajusta las prioridades de manera dinámica. Este ejercicio me mostró la importancia de seleccionar la estrategia adecuada para cada escenario, especialmente en sistemas donde el tiempo de respuesta y la eficiencia son factores críticos.

6.REFERENCIAS.

Bansal, B. I. (31 de julio de 2021). *First Come First Serve Scheduling in Python [FCFS]*. Obtenido de https://www.askpython.com/python/examples/first-come-first-serve: https://www.askpython.com/python/examples/first-come-first-serve

Implementation of Shortest Job First (SJF) Scheduling in Python. (01 de junio de 2024). Obtenido de https://www.geeksforgeeks.org/implementation-of-shortest-job-first-sjf-scheduling-in-python/: https://www.geeksforgeeks.org/implementation-of-shortest-job-first-sjf-scheduling-in-python/

Procesos. (s.f.). Obtenido de https://www3.uji.es/~redondo/so/capitulo2_IS11.pdf: https://www3.uji.es/~redondo/so/capitulo2_IS11.pdf

Round Robin Scheduling with different arrival times. (17 de junio de 2024). Obtenido de https://www.geeksforgeeks.org/round-robin-scheduling-with-different-arrival-times/: https://www.geeksforgeeks.org/round-robin-scheduling-with-different-arrival-times/