Assignment - 4

PANKAJ GUPTA
7335
TE COMPS

* TITLE:
Implementation of FCFS, SJF (Preemptive), Priority (Non-preemptive) & Round Robin (Preemptive).

* Objectives:
- To study the process management & various scheduling policies viz. Preemptive & Non-preemptive.
- To study & analyze different schedule algorithms.

* Problem Statements-
- Write a program to simulate CPU Scheduling Algorithms: FCFS, SJF (Preemptive), Priority (Non-Preemptive) & Round Robin (Preemptive).

* Outcomes-
We will be able to understand the Implimentation of different scheduling algorithms.

* Software & Hardware Requirements:-
- Fedora
- Eclipse
- JDK

* Theory-
1) Define process: It can be defined as the fundamental unit of Execution. It represents a program or task that is currently Executing on a computer. Process are crucial for managing and executing tasks efficiently in a multitasking Environment.

→ Process scheduling is a fundamental component of operating systems, and its necessity arises from several key requirements and challenges in modern computing environments.

→ Process scheduling is essential because of the following mentioned problems: multiprogramming & multitasking, fair Resource Allocation, Response time, Throughput, Prioritization, Resource management, Concurrency & management, Prevention of deadlock, Interrupt handling.

→ Effective scheduling can help prevent deadlock, a situation in which two process are waiting for resources that are being held by the other process.

→ Workloads on a computer system can change over time. Scheduling adapts to these changes by reordering the execution of process as needed to meet system goals.

→ In a distributed system, process scheduling can help balance the load across multiple server or nodes, ensuring that no server is overloaded while others are underutilized.

* Scheduling criteria & policies -

→ CPU utilization: This criterian aims to keep the CPU busy as much as possible. A scheduling policy based on CPU utilization attempts to minimize idle time for the processor.

→ Throughput: Throughput measures the numbers of process completed in a given time. Scheduling for throughput seek to maximize

the number of process executed over a specific period of time.

→ Turn around time - Turn around time is the time taken to execute a process from its submission to completion.

→ Waiting time - Waiting time is the total time a process spends waiting in the ready queue before it gets CPU time.

→ Response time - Response time measures the time taken from submitting a request until the first response is produced.

Following are the scheduling policies -

1) First-come, First-Served (FCFS) - Processes are executed in the order they arrive in the ready queue. FCFS scheduling is simple but can result in poor turnaround time, especially if long-running processes arrive first.

2) Shortest Job first (SJF) - The scheduler selects the process with the shortest execution time. SJF scheduling minimizes waiting & turnaround time, but it requires accurate predictions of process execution times.

3) Priority scheduling - Each process is assigned a priority, and the scheduler selects the highest-priority process for execution. Priority scheduling can be either preemptive or non-preemptive. Preemptive priority scheduling allows higher-priority process to interrupt low priority ones.

4) Round Robin (RR) - RR scheduling allocates a fixed time slice (quantum) to each process in a circular manner. This policy ensures fairness and responsiveness but may not be optimal for all the types of workloads.

* Process states -

→ A process can exist in various states as it transitions through its lifecycle. These process states are used to manage and track the progress of a process.

1) New - In this state, a process is being created but has not yet been admitted to the system for execution.

2) Ready - In this state, the process is prepared to execute but is waiting for the CPU to become available.

3) Running - The Running states indicates that the available process is currently being executed by the CPU. At any given time there is typically only one process in the running state.

4) wait - A process in the wait state is temporarily unable to execute because its waiting for an event or resource, such as I/o Exception to complete or a signal from other processes.

5) Exit Terminated - A process has completed it's execution. It enters the terminated state. In this state, the operating system releases the process's resources, deallocates memory & performs the clean up tasks.

* Algorithms -

1) FCFS
i) Input the process along with their burst time (bt).
ii) Find waiting time for all processes.
iii) As the first process comes need not to wait so waiting time of process will be 0 i.e wt[0] = 0.
iv) Find waiting time for all the processes i.e for process i → wt[i] = bt[i-1] + wt[i-1].
v) Find turnabound time = waiting time + burst time.
vi) Find Average waiting time = total waiting time / no of processes.
vii) Find Average turnbaround time = total turnaround time / no of processes.

2) SJF -
i) Traverse until all process gets Executed.
ii) Find process with minimum ~~wait~~ remaining time at Every single ~~lap~~.
iii) Reduce it time by 1.
iv) check if remaining time becomes 0.
v) Increment the counter of process completion.
vi) Completion time of current process = current time + 1;
vii) Calculate waiting time i.e for process i → wt[i] = bt[i-1] + wt[i-1].
viii) Increment time lap by 1.
ix) find turn around time i.e waiting time + burst time.

3) Priority Scheduling -
i) First ~~Input the~~ process with their burst time & priority.
ii) Sort the processes, burst time and priority, according to the priority.
iii) Apply the FCFS algorithm.

4) RR -

i) Create an array rem_bt[] to keep track of remaining burst time of process. This array is initially a copy of bt[].

ii) Create another array wt[] to store waiting time of process. Initialize this array as 0.

iii) Initialize time: t=0.

iv) Keep traversing all the processes while all processes are not done. Do following for ith process if its not done yet.

a) if rem_bt[i] > tq
   i) t = t + tq
   ii) bt_rem[i] -= tq

b) Else
   i) t = t + bt_rem[i]
   ii) wt[i] = t - bt[i]
   iii) rem_bt[i] = 0.

* Conclusion -

CPU policies are implemented successfully.

Assignment -5

PANKAJ GUPTA
7335
TE COMP B

* TITLE:
Implementation of solution of Reader writer synchronization problem.

* Objectives-
1) To understand reader-writer problem.
2) To solve reader-writer synchronization problem using mutex & semaphore.

* Problem statements-
Write a program to solve classical problems of synchronization using mutex & semaphore.

* Outcomes-
We will be able to solve & understand various problems related to Synchronization in real time OS.

* Software & Hardware Requirements-
- Java
- Netbeans/Eclipse IDE.

* Theory-
-> Reader - Writers problem-
- There is a data area shared among a number of processor registers.
- The data area could be a file, a block of main memory, or even a bank of processor registers.
- There are a number of process that only read the data area (readers) and a number that only write to the data area (writes).

- The conditions that must be satisfied are -
  a) Any number of readers may simultaneously read the file.
  b) Only one writer at a time may write to the file.
  c) If a writer is writing to the file, no reader may read it.

→ Semaphore -
- Semaphores are system variables used for synchronization of process.
- Two of Semaphore are there -

1) Counting Semaphore.
2) Binary Semaphore.

1) Counting Semaphore - The Integer value for this Semaphore can range over an unrestricted domain.

2) Binary Semaphore - Integer value can range only between 0 and 1. Binary semaphores are rather simple to implement. They are also known as mutex lock.

→ Semaphore functions -
Package: import java.util.concurrent.Semaphore;

1) To initialize a Semaphore -
   Semaphore Sem1 = new Semaphore(1);

2) To wait on semaphore -
   while S <= 0:
   no-operation
   S--;
   Sem1.acquire();

3) To signal on a Semaphore-
Signal(S)
S++;
mutex.release();

* Algorithm for reader-writer problem-

1) import java.util.concurrent.Semaphore;
2) Create a class RW.
3) Declare a semaphore - mutex & wrt.
4) Declare integer variable readcount=0.
5) Create a nested class reader implements Runnable.
   a) Override run method. (Reader Logic)
      i) wait(mutex)
      ii) readcount+=1.
      iii) if readcount==1 then
      iv) wait(wrt);
      v) signal(mutex);
      vi) reading is performed.
      vii) wait(mutex);
      viii) readcount--1
      ix) if readcount=0 then signal(wrt);
      x) signal(mutex);

2) Create a nested class writer implements Runnable.
   a) Override run method (writer logic)
      i) wait(wrt)
      ii) ----
      iii) writing is performed.
      iv) ----

v) Signal (wrt).

3) Create a class main
   a) Create Threads for reader & writer.
   b) Start these threads.

* Output -
- Execution of Readers & writers.

* Conclusion -
Implemented reader writer Synchronization problem using semaphore in java.

25/9

Assignment-6

PANKAJ GUPTA
7335
TE COMP B.

* TITLE-
Memory placement strategies-

* Objectives-
To acquire knowledge of memory placement strategies and be able to implement it.

* Problem statements-
Write a java/C++ program to simulate memory placement strategies.
1) First fit.
2) best fit.
3) Worst fit.
4) Next fit.

* Software/Hardware requirements-
- OS - Linux/macOS/ windows.
- java jdk installed
- IDE - Eclipse/Netbeans/ VS Code.

* Theory-
Memory placement strategies are essential in computer systems, Especially in the context of memory management, to Ensure Efficient utilisation of available memory resources. By making decisions about where to allocate memory and how to allocate it, these strategies aim to minimise memory fragmentation which

(34)

negatively impact system's performance & efficiency.

Efficient memory placement can lead to better memory utilisation, reducing waste and ensuring that memory is used efficiently. This is particularly important in systems with limited memory resources. In multitasking & multiprogramming environment, proper memory placements can help prevent deadlocks & thrashing.

Fragmentation refers to division of memory its smaller, non-contiguous blocks, making it challenging to allocate large continuous chunks of memory. It is of two types:

- Internal fragmentation - occurs only when memory allocated is fixed blocks, and the allocated block is larger than what is needed.

- External fragmentation - occurs when there are enough free memory spaces is total, but they are scattered throughout memory, making it difficult to allocate a large contiguous block.

Lets illustrate working of memory placement strategies with an example of best fit.

Imagine a computer system with following memory block size.
100KB, 200KB, 150KB, 300KB, 250KB.
Process A, B, C request memory allocations of 120KB, 180KB, 130KB, respectively.

i) when A requests 120 kB, the best fit strategy would allocate it the 150 kB block memory.

ii) Similarly B requiring 180 kB would be allocated 200 kB memory Block

iii) Process C with a request of 130 kB can take 150 kB block leaving 20 kB unused.

* Algorithm -

1) First fit -
- Read all required input.
- for i > 0 to all jobs 'i'
  for j > 0 to all jobs 'b'
  if (block[j] ≥ jobs[i])
  check jth block is already in use or free block.
  Otherwise allocate jth block to ith proc
- Display all jobs with allocated block & fragmentation.

2) Best fit -
a) Initiate a list of available memory blocks.
b) when a new process requests memory allocation.
  i) initialize a variable to keep track of best fit block size.
  ii) initialize a pointer to the best fit memory block.
  iii) ~~for Each~~ available memory block in the list.
  if block size ≥ process request size & & block size < current best size.
  update best fit size → current block size.
  update pointer to ~~worst~~ best fit block.
  iv) if best fit block pointer is still null, there is no suitable block.

v) Otherwise, allocate memory from best-fit block.

3) Next fit -
a) Initialize the list of available memory array.
b) Initialize a pointer to last allocated memory block.
c) When a process requests memory allocation.
i) Start searching for an available memory block from last allocated block.
ii) If a suitable block is found → allocated memory from that block.
→ update last allocated block pointer to current block.
iii) If no suitable block is found in the search, wrap around to beginning of list & continue searching.
iv) If no suitable block is found after complete cycle, memory allocation fails otherwise.

4) Worst fit -
- read all required inputs.
- for i>0 to all jobs 'js'
set wstIdx = -1
for j>0 to all block 'bs';
if (block[j] ≥ job[i]
if the block is fit & wstIdx is == -1 then set wstIdx = j;
if the block[wstIdx] < block[j] then block wstIdx = j;

if wstIdx != -1 the allocate jth block to ith job.

☆ Test Case -
→ Input - no. of job (js) & no. of blocks (bs)
job size of all jobs & block size for all block. eg:
js = 4, bs = 5, block[] = {100, 500, 200, 300, 600}.

7335

jobs[] = { 212, 417, 112, 426 }.

→ Output -

1) first fit -

Process no.	Process size	Allocated block
1	212	1
2	417	4
3	112	3
4	426	2

2) Best fit.

Process no.	Process size	Allocated Block
1	212	3
2	417	1
3	112	2
4	426	4

3) Next fit

Process no.	Process size	Allocated block
1	212	1
2	417	3
3	112	4
4	426	2

4) Worst fit

Process no.	Process Size	Allocated block
1	212	4
2	417	1
3	112	2
4	426	3

* Conclusion -
Successfully Implemented the simulation of memory placement strategies.

Assignment - 7

PANKAJ GUPTA
7335
TE COMP B.

* TITLE-
Page replacement algorithm.

* Objectives-
To study & understand page replacement algorithms/policies & Efficient frame management.

* Problem statement-
Write a java program to implement paging simulation using.
1) FIFO
2) Least Recently used (LRO).
3) Optimal algorithm.

* Software/Hardware requirements-
- OS - linux/macOS/windows.
- java jdk installed.
- IDK - Eclipse/Netbeans/Vscode.

* Theory-
=> Concept of page replacement-
- page fault - absence of page when refland is main memory duri paging leads to page fault.
- page fault - replacement of a already Existing page from main memory by the required new page is called page replacement and the techniques used for it are called page replacement algorithms.

→ Need of page replacements -

page replacement is primarily used for the virtual memory management in virtual memory paging system principal issue is replacement i.e. which page is to be rearranged so as to bring in the new page thus the use of page replacement algorithms. Demand paging is the technique used to increase system throughput. To implement demand paging the primary requirement is page replacement.

→ Page replacement policies -

1) Determine which page is to be removed from main memory.
2) find a free frame.
   a) if the frame is found use it.
   b) else we use page replacement algorithm to replace a victim frame.
3) Write the victim page to disk.
4) Read desired page into mm from frame, change the page & frame tables
5) Restart the user process.

→ Page replacement algorithms -

1) FIFO -
The OS keeps tracks of all pages in memory in a queue, the oldest page is in the front of queue. when a page needs to be replaced the page in the front of the queue is selected for removal.

2) Optimal -
Replace the page that is not used with the longest period of time as compared to other pages in main memory. This algorithm has lowest page fault rate. This algorithm guarantees the lowest

possible page-fault rate for a fixed no of pages but is very difficult to implement, as it requires knowledge of reference strings i.e strings of memory references.

3) Least Recently Used (LRU) -
It uses the time of page's last usage. It uses recent past as an approximation of near future, then we can replace the page that has not been used for the longest period of time i.e. the page having larger idle time is replaced. It is difficult to implement, requires substantial hardware assistance.

* Algorithm -
i) FIFO -
a) start
b) Read no of pages n.
c) read page no into an array a[i].
d) initialize avail[i] = 0 to check page hit.
e) Replace the page with circular queue, while replacing check page availability in the frame. Place avail[i] = 1, if page is replaced in the frame count page faults.
f) print results
g) stop.

2) LRU -
a) start
b) declare size
c) Get no of pages to be inserted.
d) Get the value.

e) declare counter & stack.
f) select least recently used page by counter value.
g) stack them according to selection.
h) display the values.
i) stop.

3) Optimal -
a) start
b) Read no of pages & frames.
c) Read each page value.
d) Search for pages in the frame.
e) if not available allocate free frame.
f) If no frame is free then replace the page that is last used
g) print page no of page fault.
h) stop.

→ Input -
- no of frames
- no of page
- page sequence

→ Output -
- Sequence of allocation of pages in frames
- Cache hit & Cache miss ratio.

* Conclusion -
Successfully implemented all page replacement policies.

11/12/23