## Variable partitions memory management system

Dynamic partitions.

With dynamic partitions, available memory is still kept in contiguous blocks but jobs are given only as much memory as they request. Here memory is not wasted within the partitions, but it doesn’t eliminate entirely the problem.

The problem is that whereas this scheme fully utilises memory for the first jobs, new jobs that are not of the same size, don’t fully occupy the memory, many free memory blocks, thus wasting memory.

In dynamic partitions, the partitions are not defined at the time of system generation.

**Best-fit versus first allocation**

In this case partitions are allocated on a first-fit or best-fit basis. For both schemes the memory manager organises the memory list of the free and used partitions (free/busy).

The best-fit allocation method keeps the free/busy lists in order by size, smallest to largest. Processing time is long.

The first-fit method keeps the free/busy lists organised by memory locations, low-order memory to high-order memory. Each has some advantage over the other. Best-fit usually makes the best use of memory space. First-fit is faster in making the allocation.

Example of **Best-fit versus First-fit**

Job/process/program **First-fit Best-fit**

O/S

J2

J4

J1

O/S

J1

J4

J2

O/S

100K

25K

25K

50K

J1 – 30k

J2 – 50K

J3 – 35K

J4 – 25K

External fragment is available because the job was not able to be allocated thus there is space in both the first-fit and best-fit since one job is left out. Eg. J3 is left out in First-fit and J3 is also left out in Best-fit.

Then J3 can come in when J2 or J1 has been completed.

**Note**: External fragmentation is when a job is not taken over yet there is enough space that is wasted that would have occupied it.

# Example 2

Job list

0

J1 – 10K

O/S

J5

J2

J3

J6

10K

O/S

J1 – 10K

J2 – 15K

J3 – 20K

J4 – 50K

J2 – 15K

20K

J3 – 20K

35K

J4 – 50K

J5 – 5K

55K

J6 – 30K

J7 – 10K

J8 – 30K

105

Assuming J1 and J4 have ended, then there is deallocation

Assuming J3 has ended then J7 will occupy it.

**Deallocation**

For fixed partition, system is quite straight forward. When the job is completed the memory manager resets the status of the memory block where the job was stored to “free”. Any code, e.g. binary values with 0 indicating free and 1 indicating busy.

A dynamic partition system uses a more complex algorithm because the algorithm tries to combine free areas of memory whenever possible. Therefore, the system must be prepared for three alternative situations.

When the block to be allocated is;

* Adjacent to another free block.
* Between two free blocks.
* Isolated from other free block.

**Relocatable Dynamic partitions**

With this memory allocation scheme, the memory manager relocates programs to gather together all of the empty blocks and compact them to make one block of memory that’s large enough to accommodate some or all of the jobs waiting to get in.

The compaction of memory, sometimes referred to as **“garbage collection”,** is performed by the operating system. The disadvantage is that while compaction is being done everything else must wait.

# NON CONTIGUOUS MEMORY ALLOCATION

**Paged memory allocation**

This is based on the concept of dividing each incoming job into pages of equal size.

The scheme works quite efficiently when the pages, sectors and page frames are all the same size. Page frames are sections of the main memory.

Before executing a program, the memory manager prepares it by:

* Determining the number of pages in the program
* Locating enough empty page frames in main memory.
* Loading all of the program’s pages into them.

During the loading process, pages are stored in any available page frame anywhere in main memory.

In this scheme, compaction scheme is eliminated.

The disadvantage with this scheme is the memory manager system is overworked into arranging a mechanism to keep track of the job’s pages. This enlarges the size and complexity of the operating system software, which increases overhead.

Page frames are arranged in lines, of 100 per page.

The other disadvantage is that internal fragmentation remains still a problem, although only in the last page of each job. The biggest advantage of this scheme is that it allows jobs to be allocated in non-contiguous memory locations so that memory is used more efficiently.

**Demand paging**

Demand paging introduced the concept of loading only a part of the program into memory for processing. With this system, still jobs are divided into equally sized pages that initially reside in secondary storage.

One of the most important innovations of demand paging scheme is that it allows the user to run jobs with less main memory than would be required if the operating system was using the paged memory allocation scheme.

Although demand paging is a solution to inefficient memory utilisation, it’s not free of problems. When there is an excessive amount of page swapping back and forth between main memory and secondary storage, the operation becomes inefficient. This is a phenomenon called **thrashing.**

**Page replacement policies and concepts**

Page replacement policy is crucial to the efficiency of the system and the algorithm to do that must carefully be selected.

Two of the most well-known are **first-in first-out (FIFO)** and **Least Recently Used (LRU)**

**First-In First-Out**

This page replacement policy will remove the pages that have been in memory the longest.

To calculate failure rate, we divide the number of page requests into the number of interrupts. A page interrupt, which will identify with an asterisk (\*), is generated when a new page is brought into memory (whether a page is swapped out or not).

**Least Recently Used (LRU)**

The Least Recently Used (LRU) page replacement policy swaps out the pages that show the least amount of recent activity, figuring that these pages are the least likely to be used again in the immediate future. Conversely, if a page is used, it’s likely to be used again soon; this is the basis of the theory of “locality”.

**Segmented Memory Allocation**

With segmented memory allocation, each job is divided into several segments of different sizes, one for each module which contains pieces that perform related functions.

In this policy, the main memory is no longer divided into page frames because the size of each segment is different. Therefore memory here is allocated in a dynamic manner.

When a programme is compiled or assembled, the segments are set up according to the program’s structural modules. Each segment is numbered and a segment map table is generated for each job.

The memory manager keeps track of the segments in memory and this is done with three tables combining aspects of both dynamic partitions and demand paging memory in arrangement.

* The job table list
* The segment map table list
* The memory map table

The addressing schemes here require the segment number and the displacement within that segment.

**Segmented/Demand Paged Memory Allocation**

This is a combination of segmentation and demand paging. This allocation scheme doesn’t keep each segment as a single contiguous unit but subdivides it into pages of equal size, smaller than most segments, and more easily manipulated than whole segments. Therefore many of the problems of segmentation (compaction, external fragmentation, and secondary storage handling) are removed because the pages are of fixed length.

# Virtual memory management systems

Previous systems both contiguous and non-contiguous were based on the assumption that the entire process image was in the main memory at a time of execution.

If a process was to be removed from memory, the entire process was swapped out and if the process was to be loaded the whole process needed to be swapped in memory. This therefore made the degree of multiprogramming to be limited because the number of processes it could hold was limited.

Virtual memory management system keeps only a part of the process in the memory and the other part on the disks still are able to execute it.

Virtual memory system can also be implemented using paging, segmentation or combined schemes. Eg. Under paging a program consists of a number of logical or virtual pages, which are loaded into specific page frames. When a page not currently in memory is referenced only that page can be brought from the disk. However this is time consuming therefore increasing the throughput.

If there is no page frame free in the physical memory to accommodate this new page, the operating system can over write an existing page in the physical memory. A page to be replaced is governed by “page replacement policy.” However if a page to be replaced had been modified the operating system first copies the “dirty page” back on the disk before overwriting it.

The capability of, moving pages at will between two storage areas (main memory and secondary storage) gave way a new concept appropriately named virtual memory. It gives the users the appearance that their programs are being completely loaded in main memory during their entire processing time.

During the second generation, programmers started dividing their programs into sections that resembled working sets, or really segments, called **“overlays”.**

Virtual memory works well in a multi-programming environment because most programs spend a lot of time waiting – they wait for I/O to be performed; they wait for pages to be swapped in or out; and in a time–sharing environment, they wait when their “time slice is up” i.e. their turn to use the processor is expired.

Virtual memory management has several **advantages**;

1. A jobs size is no longer restricted to the size of main memory (or the free space within main memory).
2. Memory is used more efficiently because the only sections of a job stored in memory are those needed immediately while those not needed remain in secondary storage.
3. It allows an unlimited amount of multi-programming.
4. It eliminated external fragmentation and minimises internal fragmentation.
5. It allows for sharing of code and data.
6. It facilitates dynamic linking of program.

**Disadvantages**

1. Increased hardware costs
2. Increased overload for handling paging interrupts.
3. Increased software complexity to prevent thrashing.

COMMONLY USED JARGONS IN VIRTUAL MEMORY MANAGEMENT SYSTEM.

* Locality of reference
* Page fault
* Working set

#### Page replacement policy

* Dirty page/dirty bit
* Demand paging
* Demand paging Vs demand segmentation

## Processor manager

In single–user systems the processor is busy only when the user is executing a job – at all other times it is idle. Processor management in this environment is simple. However, when there are many users with many jobs on the system (this is known as a **multiprogramming** environment), the processor must be allocated to each job in a fair and efficient manner, and this can be a complex task.

A **processor** which is also known as the Central Processing Unit (CPU) is the part of a machine which does the calculations and executes the program for example a single mathematical calculation is a process.

A **job,** or **program** in an operating systems environment, is a unit of work that’s submitted by the user.

Multiprogramming requires that the processor be allocated to each job or to each process for a period of time and deallocated at an appropriate moment.

A single processor can be shared by several jobs, or several processes, but if, and only if, the operating system has a scheduling policy as well as a scheduling algorithm to determine when to stop working on one job and proceed to another.

**Job Scheduling Versus Process Scheduling**

The processor manager is a composite of two sub managers:-

i). Job scheduler which is in charge of job scheduling.

ii). Process scheduler which is in charge of process scheduling.

A **Job scheduler** is also called the high-level scheduler, which is only concerned with selecting jobs from a queue of incoming jobs and placing them in the process queue. The job scheduler’s goal is to put the jobs in a sequence that will use all of the system’s resources as fully as possible. It keeps most of the components of the computer systems busy most of the time.

**Process scheduler**

Once a job has been placed on the READY queue by the job scheduler, it’s the process scheduler that takes over. It determines which jobs will get the CPU, when, and for how long, it also decides when processing should be interrupted, determines which queue the job should be moved to during its execution, and recognises when a job has concluded and should be terminated.

The process scheduler is the low-level scheduler that assigns the CPU to execute the processes of those jobs **I/O-bound** jobs (such as printing a series of documents) have many brief CPU cycles and long I/O cycles, whereas **CPU-bound** jobs (such s finding the first 300 prime numbers) have long CPU cycles and shorter I/O cycles.

In a single-user environment, there is no distinction made between job and process scheduling because only one job is active in the system at any given time. So the CPU and all other resources are dedicated to that job until it is completed.

**Job and Process Status**

As a job moves through the system its always in one of five status (or at least three). The are hold, ready, running, finished and waiting.

Job scheduler (balancing use of resources memory, devices).

Process scheduler (algorithm)

Job scheduler process scheduler (release resources)

**HOLD**

**READY**

**RUNNING**

**FINISHING**

**WAITING**

Process scheduler (I/O request, page fault)

Process scheduler (signal from device manager or page interrupt handler)

Process scheduler (time interrupt, privity interrupt)

* The transition from HOLD to READY is initiated by the job scheduler according to some predefined policy, like availability of enough main memory and any request devices
* The transition from READY to RUNNING is handled by the process scheduler according o some predefined algorithm ie. FCFS, SJN, priority scheduling, SRT, or round robin.
* The transition from RUNNING to READY is handled by the process scheduler according to some predefined time limit, or other criterion, i.e. a priority interrupt.
* The transition from WAITING to READY is handled by the process scheduler and is initiated by a signal from the I/O device manager that the I/O request has been satisfied and the job can continue.
* Eventually, the transition from RUNNING to FINISHED is initiated by the process scheduler or the job scheduler.

A **process** is the basic unit for CPU scheduling in an operating system. It is an execution stream in the context of a particular process state

A process may be independent of other processes in the system or may interact with some other processes. An execution stream is a sequence of instructions.

A process is an execution stream in the context of a particular process state. An execution stream is a sequence of instructions. Process state determines the effect of the instructions. Process is a key OS abstraction that users see - the environment you interact with when you use a computer is built up out of processes.

Differences between a program and a process.

# Program process

It doesn’t compete for computing resources like the CPU A process does

or memory.

It can exist on paper or reside on a disk, it may be It doesn’t do any of those

Compiled or tested, but it still doesn’t compete for done by the program.

CPU, RAM and other resources.

**Process Transition/State**

NeNEWw

### NEW

Ready

### READY

Waiting

**WAITING**

Running

##### RUNNING

Terminated

##### TERMINATED

Process has terminated executed or execute has been completed or rejected.

Exit

Instructions being executed

I/O Event wait

Process waiting 4 and b4 it can continue.

I/O Execute implementation

Already created

waiting 4 execution

Admitted

Process is being created

Interrupt

Scheduler dispatch

# 

Example of process transition/state

Two concepts: uniprogramming and multiprogramming.

Uniprogramming: only one process at a time. Typical example is DOS. Problem: users often wish to perform more than one activity at a time (load a remote file while editing a program, for example), and uniprogramming does not allow this. So DOS and other uniprogrammed systems put in things like memory-resident programs that invoked asynchronously, but still have separation problems. One key problem with DOS is that there is no memory protection - one program may write the memory of another program, causing weird bugs.

Multiprogramming: multiple processes at a time. Typical of Unix plus all currently envisioned new operating systems. Allows system to separate out activities cleanly. However Multiprogramming introduces the resource sharing problem - which processes get to use the physical resources of the machine and when? One crucial resource: CPU. Standard solution is to use preemptive multitasking - OS runs one process for a while, then takes the CPU away from that process and lets another process run. Must save and restore process state. Key issue is fairness. Must ensure that all processes get their fair share of the CPU.

# Process Control Blocks(PCB)

Is one which gives key information about a process when the operating system switches the attention of processor among processes. It uses the save areas in the PCB to hold information that needs to restart each process when the process next gets the CPU. The PCB is the entity that defines a process of the operating system.

|  |
| --- |
| Process – ID |
| Process state |
| Process priority |
| Register Save Area for PC, IR, SP … |
| Pointers to process’s memory |
| Pointers to other resources |
| List of open files |
| Accounting information |
| Other information if required |
| Pointer to other PCBs |

# Process priority

Some processes are urgently required (set either higher or lower priority). Priority can be set externally by the user or system manager or internally by the OS depending on various parameters or by a combination of both.

# Pointers to the process’s memory

This gives direct or indirect addresses or pointers to the locations where the process image resides in the memory. E.g. in paging system, it could point towards the Page Map Table (PMT) which in turn point to the physical memory (indirect).

# Pointer to other resources

# This gives pointers to other data structures maintained for that process.

### Accounting Information

This gives account of the usage of resources such as CPU time, connect time, disk I/O used especially in a data centre environment or cost centre environment where different users are to be charged for their system usage.

### List of open files

This can be used by the O/S to close all open files not closed by a process on termination.

### Pointer to other PCB

This gives the address of the next PCB (e.g PCB number) within a specific category i.e the O/S maintains a list of ready processes.

### Register Save Area

This is needed to save all the CPU Registers at the context switch

**Process Scheduling Policies**

In a multiprogramming environment there are usually more jobs to be executed than could possibly be run at one time.

The following criteria should be followed when developing an operating system in order to efficiently mange the processor:-

* Maximise throughput i.e as many jobs as possible in a given time.
* Minimise response time
* Minimise turnaround time by moving entire jobs in and out of the system quickly.
* Minimise waiting time by moving jobs out of the READY queue as quickly as possible.
* Maximise CPU efficiency by keeping the CPU busy 100% of the time.

A scheduling policy that interrupts the processing of a job and transfers the CPU to another job is called a **premptive scheduling policy** and it is widely used in time-sharing environments.

A **non-preemptive scheduling policy** functions without external interrupts. Once a job captures the processor and begins execution, it remains in the RUNNING state uninterrupted until it is finished or issues on I/O request (natural wait).

**Process scheduling algorithms**

The process scheduler relies on a process scheduling algorithm, based on a specific policy, to allocate the CPU and move jobs through the system. Here are six process scheduling algorithms that have been used extensively:

**First Come First Served (FCFS)**

This is a nonpreemptive scheduling algorithm that handles jobs according to their arrival time. The earlier they arrive, the sooner they are served. This algorithm uses a FIFO type of queue. This algorithm is fine for most batch systems, but it is unacceptable for interactive systems because interactive users expect quick response times.

With FCFS, a new job is linked to the READY queue and it is removed from the front of the queue when the processor becomes available. In a FCFS system, there are no WAIT queues. Each job is run to completion.

**Examples**

1. Job A has a CPU cycle of 15 milliseconds

Job B has a CPU cycle of 2 milliseconds

Job C has CPU cycle of 1 millisecond

For each job, the CPU cycle contains both the actual CPU usage and the I/O request. That is its arrival sequence of A,B,C, the time line is show are below.

|  |  |  |
| --- | --- | --- |
| Job  A | Job  B | Job  C |

O 15 17 18

If all jobs arrive almost simultaneously, we can calculate that the turnaround time for Job A is 15, for Job B is 17, and Job C is 18. so the average turnaround time is:

15 + 17 + 18 =16.67

3

However, if the jobs arrived in a different order, say C,B,A then the result using the same FCFS algorithm would be as shown below.

|  |  |  |
| --- | --- | --- |
| Job  C | Job  B | Job  A |

O 1 3 18

In this case, the turnaround time for Job A is 18, for Job B is 3, and for Job C is 1 and the average turnaround time is 18 + 3 + 1 = 7.3

3

The above two examples illustrate the primary turnaround times very widely. In fact when there are 3 jobs in the READY queue, the system has only a 1 in 6 chance of running the jobs with most advantageous sequence.

**Shortest Job Next (SJN)**

Shortest job next (SJN) is a non premptive scheduling algorithm (also known as shortest job first (SJF)). This handles jobs based on the length of their CPU cycle time.

Its easiest to implement in batch environments where the estimated CPU time required to run the job is given in advance. However, it doesn’t work in interactive systems because user don’t estimate in advance the CPU time required to run their jobs.

**Examples**

Here are four batch jobs, are in the READY queue, for which the CPU cycle, or run time, is estimated as follows;

Job: A B C D

CPU Cycle: 5 2 6 4

The SJN algorithm reschedules the four programs for processing in the order:

B, D, A, C

|  |  |  |  |
| --- | --- | --- | --- |
| Job  B | Job  D | Job  A | Job  C |

O 2 6 11 17

The average turnaround time = 2 + 6 + 11 + 17 = 9.0

4

The general formular for the average can be proved as below;

T1 (n) + t2(n – 1) + t3 (n – 2) + ……. + tn (1)

n

Where n – is the number of jobs in the queue

Ti (i = 1, 2, 3, …, n) is the length of the CPU cycle for each of the jobs.

However, the SJN algorithm is optimal only when all of the jobs are available at the same time and the CPU estimates are available and accurate.

**Priority scheduling**

This algorithm gives preferential treatment to important jobs. It allows the programs with the highest priority to be processed first, and they aren’t interrupted until their CPU cycles (run times) are completed or a natural wait occurs. If two or more jobs with equal priority are present in the READY queue, the processor is allocated to the one that arrived first (FCFS within priority). With a priority algorithm, jobs are usually linked to one of several READY queues by the job scheduler based on their priority.

Priorities can also be determined by the processor manager based on characteristics intrinsic to the jobs such as;

* Memory requirements: Jobs requiring large amounts of memory could be allocated lower priorities than the opposite.
* Total CPU time: Jobs having long CPU cycle or run time are given lower priority.
* Amount of time already spent in the system: some systems increase the priority of jobs that have been in the system for an unusually long time to expedite their exit. This is known as “**aging**”.

**Shortest Remaining Time (SRT)**

This is a preemptive version of SJN algorithm. The process is allocated to the job closest to completion- but even this job can be pre-empted if a newer job in the READY queue has a “time to completion” that’s shorter.

This algorithm is often used in batch environments.

**Example**

Four jobs arrived in quick succession (1 CPU cycle a part) as shown below. Use SRT algorithm.

Arrival time: 0 1 2 3

Job: A B C D

Cpu cycle: 6 3 1 4

Here job A is pre-empted by job B because job B has less CPU time remaining

Here job B is pre-empted by job C because job C has less CPU time remaining

Now job B can resume because job C has finished.

Job D run next because it needs less CPU time to finish than Job A

Here job A is finally allowed to finish.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Job  A | Job  B | Job  C | Job  B | Job  D | Job  A |

0 1 2 3 5 9 14

In this case the **turnaround time** is the **completion time** of each job minus its **arrival time.**

Job: A B C D

Turnaround: 14 4 1 6

Average turnaround time = 14 + 4 + 1 + 6 = 6.25

4

**Comparing it with non-preemptive SJN policy**

|  |  |  |  |
| --- | --- | --- | --- |
| Job  A | Job  c | Job  b | Job  D |

0 6 7 10 14

In this case the turnaround time is:

Job: A B C D

Turnaround: 6 9 5 11

Average turnaround time = 6 + 9 + 5 + 11 = 7.75

4

In this case, initially A is the only job in the READY queue so it runs first until it is finished since SJN is non preemptive. The next job to run is C because when job A is finished after time 6, all the other jobs (B, C, D) have arrived. Of those three, C is one with shortest CPU cycle, than B and finally D.

A precise comparison of SRT and SJN would have to include the time required to do context switching. When Job A is pre-empted, all of its processing information must be continued, and the contents of job B’s PCB are loaded into appropriate registers so that it can start running again this is a **context switching**.

**Round Robin**

This is a preemptive process scheduling algorithm that is used extensively in interactive systems because it is easy to implement and it is not based on job characteristics but on predetermined slice of time that’s given to each job to ensure that the CPU is equally shared among all active processes and isn’t monopolised by any one job.

This **time slice** is called a **time quantum** and its size is crucial to the performance of the systems. It usually varies from 100 milliseconds to 1 or 2 seconds.

Jobs are placed in the READY queue using a FCFS scheme and the process scheduler selects the first job from the front of the queue, sets the timer to the time quantum and allocates the CPU to this job. If processing isn’t finished when time expires, the job is preempted and put at the end of the READY queue and its information is saved in PCB.

**Example**

The example below illustrates a round robin algorithm with a time slice of 4 milliseconds (I/O requests are ignored).

Arrival time: 0 1 2 3

Job: A B C D

CPU cycle: 8 4 9 5

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Job  A | Job  B | Job  C | Job  D | Job  A | Job  C | Job  D | Job  C |

0 4 8 12 16 20 24 25 26

**Time line** for job sequence A, B, C, D, using the preemptive round robin.

The **turnaround time** is the **computer time** minus the **arrival time**.

Job: A B C D

Turnaround: 20 7 24 22

Average turnaround time: = 20 + 7 + 24 + 22 = 18.25

4

**note:** In the above time line, job A was pre-empted once because it needed 8 milliseconds to complete its CPU cycle, while job B terminated in one time quantum. Job C was pre-empted twice because it needed 9 milliseconds to complete its CPU cycle, and job D was pre-empted once because it needed 5 milliseconds.

In their last execution or swap into memory, both jobs D and C used the CPU for only 1 millisecond and terminated before their last quantum expired, releasing the CPU sooner,

The efficiency of round robin depends on the size of the time quantum in relation to the average CPU cycle. If the quantum is too large, i.e. if it is larger than most CPU cycles, then the algorithm reduces to FCFS scheme. If the quantum is too small, then the amount of context switching slows down the execution of the jobs.