Operating Systems (Client)

Lecture 3 – Scheduling

Higher Cert. in Science in Computing (IT) – BN002
Bachelor of Science in Computing (IT) – BN013
Bachelor of Science (Hons) in Computing – BN104

Dr. Kevin Farrell

itb
Institute of Technology
Blanchardstown
Institiúid Teicneolaíochta
Baile Bhlainséir

Contents

Figures

Tables

License

Feedback

Constructive comments and suggestions on this document are welcome. Please direct them to:
kevin.farrell@itb.ie

Acknowledgements

Thanks in particular go to the Dr. Anthony Keane, whose lectures this material was partly based on.

Modifications and updates

Version	Date	Description of Change
1.0	1/6/2005	First published edition.
1.1	10/8/2005	Minor formatting changes
1.2	20/9/2005	Fixed Contents numbering error

1. Overview

1.1. Lecture Summary

In a previous Lecture, we discussed the life-cycle models of processes in an operating system; the so-called Process State Management Models. In this Lecture, we discuss Process Scheduling, an important subsystem of the Process Manager. It is the various schedulers' functions to order the various queues, which occur when many processes compete for limited resources in an operating system. We start by setting out the objective of a good scheduler. We then describe the three different types of scheduler, which may be programmed into an operating system. Following on from this, we focus our discussion on the scheduler, which executes most frequently – the low level or short-term scheduler. We compare and contrast a number of non-pre-emptive and pre-emptive algorithms, using a number of relatively simple metrics.

1.2. Learning Outcomes

After successfully completing this Lecture you should be able to:

- Explain the goals of a good scheduling policy
- Distinguish between high, medium and low level scheduling policies
- Calculate the average wait-time for each process for a particular low level scheduling algorithm.
- Explain the relative merits of various low-level scheduling policies.

1.3. Reading

Material for this lecture was gathered from a number of difference sources. The following are essential reading:

- “Operating System incorporating Windows and UNIX”, Colin Ritchie.

- "Operating Systems", William Stallings, Prentice Hall, (4th Edition, 2000)
- "Modern Operating Systems", Andrew Tannenbaum, Prentice Hall, (2nd Edition, 2001).
- "Operating System Concepts", Silberschatz, Galvin & Gagne, John Wiley & Sons, (6th Edition, 2003).

1.4. Suggested Time Management

This Lecture should take between 2 and 4 hours of your study time:

- Lecture Content: 1 hours
- Further reading: 2 hours

1.5. Typographical Conventions

Throughout this Operating Systems Module, I have tried to use uniform typographical conventions; the aim being to improve readability, and lend understanding:

Key Terms:	**Bold Underline**	for eg: **Multiprogramming**
Emphasis:	*Italics*	
Command names:	**`Courier Bold`**	for eg: **`ls -l`**
Filenames/Paths:	`Courier`	for eg: `/home/kevin/cv.doc`

2. Process Scheduling Policies

In a multiprogramming environment there are usually more processes to be executed than could possibly be run at one time. However, before the operating system can schedule them, it needs to resolve the limitations of the system, for example:

- there are a finite number of resources, e.g. disk drives, printer, etc.
- some resources, once allocated, cannot be shared with another process, e.g. printers.
- some resources require operator intervention – that is, they can't be reassigned automatically from process to process (such as tape drives).

So, what is a "good" process scheduling policy for the OS to adopt?
Some of the objectives relating to what scheduling should achieve in terms of the systems performance and behaviour are:

- maximise throughput
- be fair to all users
- provide tolerable response
- degrade performance gracefully
- be consistent and predictable

If the system favours one type of user then it can be disadvantageous to another user or even be inefficient with its resources. For example, the system designer might decide to maximise CPU utilisation while minimising response time and balancing the use of all system components through a mix of I/O-bound and CPU-bound processes. So the system designer would select the scheduling policy that most closely satisfies their criteria.

In making decisions about the scheduling of processor work, a number of criteria can be taken into account by operating system:

- priority assigned to process
- class of process
- resource requirements
- I/O or CPU bound
- resources used to date
- waiting time to date

Some of the factors are *static* characteristics that can be assigned prior to commencement of the process, while others are *dynamic* changing in value during process execution.

3. Types of Scheduling

The aim of processor scheduling is to assign processes to be executed by the processor over time, in a way that meets system objectives, such as **response time**, **throughput**, and **processor efficiency**. In many systems this scheduling activity is broken down into *three* separate functions: long, medium and short term scheduling. The names suggest the relative frequency with which these functions are performed.

3.1. Long-Term Scheduling

The **long-term** (or **high-level**) scheduler determines which new programs are admitted to the system for processing, i.e. it decides which newly submitted jobs are to be converted into processes and put into the READY queue to compete for access to the processor. This activity is only really applicable to batch systems, since in an on-line environment, processes will be admitted immediately unless the system is fully loaded. In this event, new logins will be inhibited rather than being queued.

Note that UNIX does not incorporate a long-term scheduler. Instead, it relies on users for that function.

3.2. Medium-Term Scheduling

The **medium-term** (or **medium-level**) scheduler is applicable to systems where a process within the systems (but not currently running) can be swapped out of *primary* memory onto disk (*secondary* memory) in order to reduce the system loading. The medium-term scheduler attempts to relieve temporary overloading by removing processes from the system for short periods. Ideally, the processes chosen for swapping out will be currently inactive; they will be re-introduced when the loading situation improves. Meanwhile, the processes are held in the READY SUSPENDED or BLOCKED SUSPENDED state.

3.3. Short-Term Scheduling

Short-term (or **low-level**) scheduler is executed more frequently than long-term or medium-term. Whereas the long and medium term schedulers operate over time scales of seconds or minutes, the short-term scheduler is making critical decisions many times a second. The short-term scheduler is also known as the **dispatcher**. The short-term scheduler will be invoked whenever the current process relinquishes control, which occur when the process calls for an I/O transfer or when some other interrupt arises, like a clock interrupt. A number of different **policies** have been devised for use in short-term schedulers, each of which has its own advantages and disadvantages.

4. Short-Term Scheduling Policies

The Process Scheduler relies on a process scheduling algorithm based on a specific policy, to allocate the CPU and move processes through the system. Early operating systems used **non-preemptive** policies for the short-term scheduler, designed to move batch jobs through the system as efficiently as possible. Most contemporary popular operating systems have their emphasis on interactive users and response time, and therefore use an algorithm that takes care of the immediate requests of interactive users.

There are several process short-term scheduling algorithms that are used extensively:

- **FCFS**: First Come First Served
- **SJF**: Shortest Job First
- **SRT**: Shortest Remaining Time
- **HRRN**: Highest Response Ratio Next
- **Priority**: Priority-Based Scheduling
- **RR**: Round Robin
- **MLQ**: Multi-Level Feedback Queues

4.1. First-Come First-Served (FCFS)

First-Come First-Served (FCFS) is a non-pre-emptive scheduling algorithm that handles processes according to their arrival time, i.e. the earlier they arrive, the sooner they are served. It is a simple algorithm to implement because it uses a First In First Out (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. FCFS favours long processes over short ones (as we will demonstrate shortly) and can lead to poor utilisation of I/O devices.

As a general rule, to be fair to all processes, we would like the length of run-time to be reflected in the length of waiting-time. This rule means that long processes may have to wait a relatively long time, whereas short processes should only have to wait a relatively short length of time. Then the ratio of waiting-time to run-time should be the same for each process. Table 4.1 gives the results of an analysis of scheduling of five fictitious processes using the FCFS algorithm.

The number of each process indicates its order of arrival into the system. The order of the queue is from top to bottom; i.e. The first process to be served is listed first, followed by the next, and so on. The time-units used are arbitrary. It is clear from the results that our rule of thumb is not followed, illustrating a weakness in the FCFS approach. A consequence of this weakness is that a long process can hog the CPU, leading to reduced response time for all users. One way of measuring the response-time is of a particular scheduling algorithm is to examine the average wait-time for all processes. For our sample processes in the table below, we see that the average wait-time using FCFS for all the processes—excluding the first, which has a zero wait-time—is:

FCFS Average Wait-Time = (2+62+63+66)/4 = 48.25

We will compare this figure with other algorithms presented.

FCFS is rarely used on it’s own but is often employed in conjunction with other methods.

	(a)	**(b)**	**(c)**
Process #	**Estimated Run-time**	**Esimated Wait-Time**	**Ratio (b)/(a)**
1	2	0	0
2	60	2	0.03
3	1	62	62
4	3	63	21
5	50	66	1.32

Table 4.1: Analysis of scheduling of five sample processes using a First-Come First-Served (FCFS) Scheduling Algorithm (Ritchie, p.67). The number of each process indicates its order of arrival into the system. The order of the queue is from top to bottom; i.e. The first process to be served is listed in the first row, followed by the next to be served in the second row, and so on.

4.2. Shortest Job First (SJF)

The **Shortest Job First** (or shortest job next (SJN)) is a non-pre-emptive scheduling algorithm that handles processes based on the length of their CPU cycle time. It's easier to implement in batch environments where the estimated CPU time required to run the process is given in advance by each user at the start of each process. It is essentially a priority scheme where the priority is the inverse of the estimated execution time.

If we use this scheme on the processes in the example in FCFS, we obtain a rather different picture; see Table 4.2 below. In this case, we see that the ratios of Wait-Time to Run-Time are very close for each process (within one order of magnitude except for process #5). Notice also that the order of the queue of processes is not the same as the arrival order. Examining the average wait-time for all processes (bar the first), we see that SJF gives an average wait-time of:

SJF Average Wait-Time = (1+3+6+56)/4 = 16.5

The weakness in SJF is that a long process in the queue may be delayed indefinitely by a succession of smaller processes arriving in the queue. This is called **starvation**. However, notwithstanding this, given that SJF results in a lower average wait-time than for FCFS, we can conclude, that on average, it is a better performing algorithm than FCFS.

	(a)	**(b)**	**(c)**
Process #	**Estimated Run-time**	**Estimated Wait-Time**	**Ratio (b)/(a)**
3	1	0	0
1	2	1	0.5
4	3	3	1
5	50	6	0.1
2	60	56	0.9

Table 4.2: Analysis of scheduling of five sample processes using a Shortest Job First (SJF) Scheduling Algorithm (Ritchie, p.67). The number of each process indicates its order of arrival into the system. The order of the queue is from top to bottom; i.e. The first process to be served is listed in the first row, followed by the next to be served in the second row, and so on.

4.3. Shortest Remaining Time (SRT)

Shortest Remaining Time (SRT) is the **pre-emptive** version of the SJF algorithm. The processor is allocated to the process closest to completion but even this process can be pre-empted (i.e. have its execution interrupted) if a newer process in the READY queue has an estimated time to completion that is shorter than the process currently being executed. The danger of starvation of long processes also exists in this scheme. Implementation of SRT requires an estimate of the total execution-time and continual measurement of elapsed run-time. This requires some overhead by OS to carry out these measurements.

4.4. Highest Response Ratio Next (HRRN)

The **Highest Response Ratio Next** (HRRN) scheme is derived from the SJF method, but modified to reduce SJF's bias against long processes to avoid the danger of starvation. HRRN derives a dynamic priority value, *P*, based on the estimated execution-time and the incurred waiting time:

$$P=\frac{(\text{Time Waiting} \;-\; \text{Estimated Execution Time})}{(\text{Estimated Run Time})}$$

The process with the *highest* priority value, *P*, will be selected for running. This technique guarantees that a process can not be starved since the effect of the wait time in the numerator of the priority expression will predominate over shorter processes with a smaller wait time. (*See Colin Ritchie's book for an example p.69/70)*.

4.5. Priority Based Non-Preemptive Scheduling

In **Priority Scheduling**, processes are allocated to the CPU on the basis of an externally assigned priority. This type of scheduling policy has a number of facets:

- Processes are given user (or system) defined priority at creation.
- Can reflect importance of process (or user), type of resources required, etc.
- Processes are sorted in READY queue by higher priority process entering queue.

There are two types of policy:

- Static – initial priority does not change
- Dynamic – priority changes over time e.g. Ageing

HRRN, discussed in the last subsection, is an example of dynamic priority scheduling.

4.6. Round Robin

Round Robin (RR) is a preemptive process scheduling algorithm that is used extensively in interactive systems. It is easy to implement since it is not based on process characteristics. Instead, each process is allocated to the CPU for a predetermined slice of time called a **time quantum**. This ensures that the CPU is equally shared among all active processes, and guarantees that no one process can monopolise the CPU. The size of the time quantum is crucial to the performance of the system. Depending on the OS, it can vary from 100s of microseconds to a few seconds but a more typical value in popular contemporary OSes is of the order of 10 to 20 milliseconds. Figure shows a schematic of a process using up its time quantum in the CPU, and subsequently being “timed-out”.

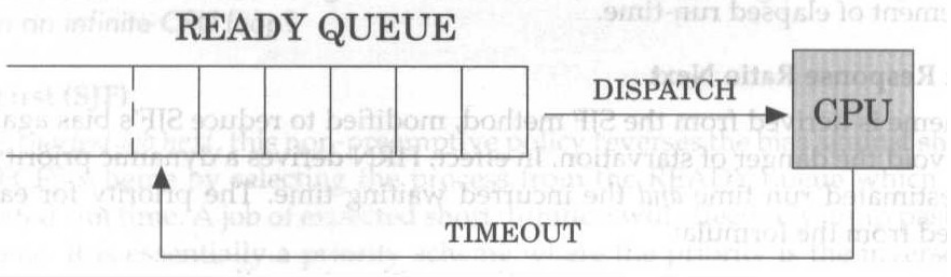

Figure 4.1: In Round Robin scheduling, each process is allowed to be executed by the CPU for a limited time called the time quantum, before it is timed-out and put back into the READY queue. There it will wait until it is its turn again to continue executing.

RR incurs a significant overhead since each time quantum brings a context switch. This raises the question of how long the time quantum should be. The time quantum should be as large as possible to minimise the overheads of context switches while not too long to reduce the users' response times. A compromise between these requirements is needed. Note that if the time quantum is too big, the RR becomes more like a FCFS. If the time quantum is too small, then the amount of context switches slows down the execution of the process and the amount of overhead is dramatically increased.

Figure 4.2 shows illustrates this by examining just one process, and the scheduling behaviour associated with three different values of time quantum. In (a), the time quantum is too long, and the long process will be allowed hog the processor without being timed out. However, in (b) the right choice allows about 80% of CPU cycles of the process to be completed before the process is timed-out, and put back into the READY queue. Since there is only one process in this system, the process is immediately dispatched from the READY queue to the CPU. In (c), the time quantum is too short, and much of the time is spent on performing context switches, each time the process is timed-out, and each time the next process (in this case, the same process) is resumed.

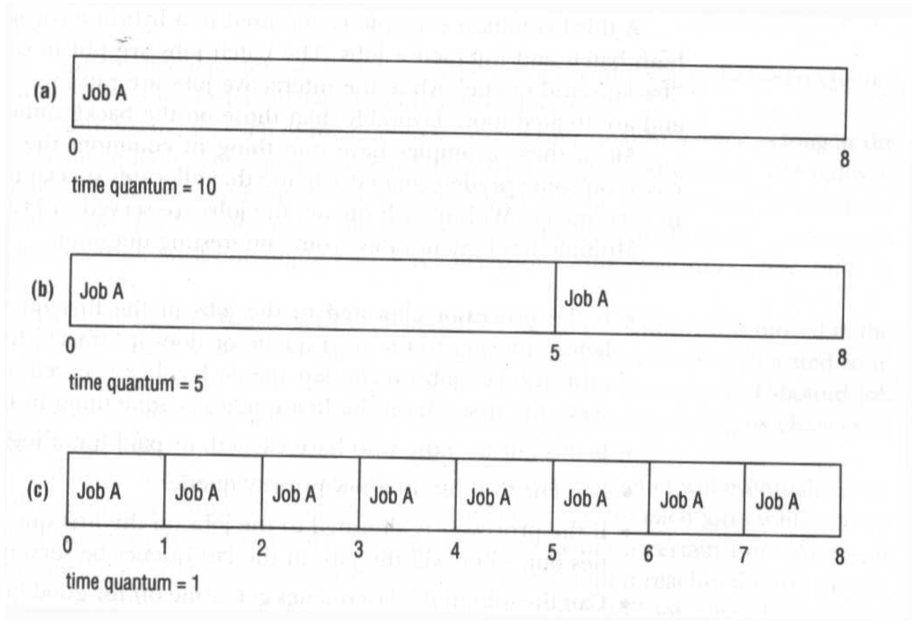

Figure 4.2: The choice of time-quantum is crucial. In this graphic, we consider just one process: (a) If it is too long, long processes will be allowed hog the processor; (b) The right choice will allow about 80% of CPU cycles to be completed before being timed-out; (c) If it is too short, much of the time spent will be on performing context switches.

There are *two* general rules of thumb for selecting the “proper” time quantum:

1. It should be long enough to allow *80%* of the CPU cycles to run to completion
2. It should be at least *100* times longer than the time required to perform one context switch.

4.7. Multiple-Level Feedback Queues

A scheduling policy base on **Multiple-Level Feedback Queues** (MFQ) is not entirely a separate scheduling algorithm but one which works in conjunction with several of the other schemes already discussed in previous sections. It is found in systems with processes that can be grouped according to a common characteristic; for example: priority, interactive or batch – see Figure 4.3.

The scheduling policies already discussed are relatively limited in their ability to cope with a wide variability in the behaviour of the processes in a system. Ideally a scheduling scheme should give priority to short processes, and favour I/O-bound processes, while otherwise being “fair” to all processes. The MFQ scheme is an attempt to provide a more adaptive policy which will treat processes on the basis of their past behaviour. They have a number of interesting features:

- MFQs have two or more READY queues, with each queue containing processes that display similar behaviour patterns.
- Processes in the second queue will only be processed when the first queue is empty. A second queue process will be preempted if a process enters the first queue.
- If a process voluntarily relinquishes control of the CPU then it will stay in the same queue.
- If a process is preempted, then it moves to a lower priority queue.
- Processes entering an I/O bound state of execution will occupy a high priority queue but when it enters a CPU bound state it will be relegated to a lower queue.
- Interactive processes will therefore achieve a good response time while batch processes will only be executed if there are no other non-batch processes in the system.
- MFQ ensures a good mix of CPU- and I/O-bound processes and ensures that CPU and I/O are utilised even when there are no interactive processes in the system; for example: during the night.

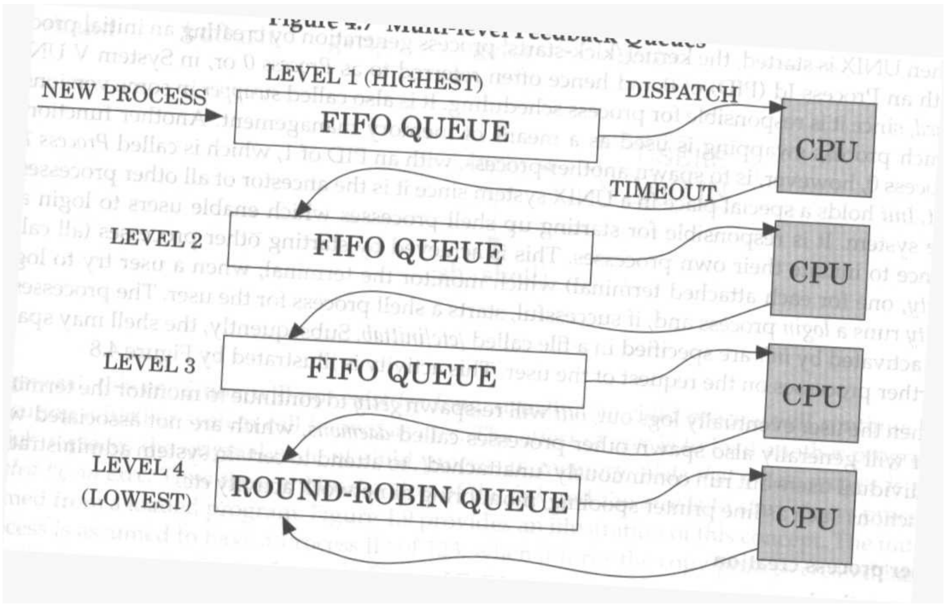

Figure 4.3: Multiple-Level Feedback Queues combine different scheduling algorithms to provide for more adaptive scheduling for the different types of process which run in a contemporary Operating System.

MLQ uses the concept of **ageing**. This term is applied to processes that spend a relatively long time in the system and so are given special treatment. Ageing is used to ensure that processes in the lower level queues will eventually complete their execution.

5. Exercises

1. Create a table with the following headings and fill it in. The first line is done for you:

Algorithm	Policy Type	Best For	Disadvantages	Advantages
FCFS	Non-preemptive	Batch	Unpredictable turn-around times	Easy to implement
SJF				
Priority				
SRT				
RR				
MFQ				

2. Describe **four** different criteria that the OS uses when scheduling processor work.
3. Describe the operation of a *multi-level feedback queue*, with **four** levels with the aid of a diagram. What types of processes would be found in the different queues?
4. List and explain two possible rules of thumb when determining the *time quantum* in a *round robin* scheduling scheme.