

Multi-programming systems

by Jonathan Ieuan Evans

FILE	4766_JONATHAN_IEUAN_EVANS_MULTI-PROGRAMMING_SYSTEMS_95675_965668631.DOCX (231.94K)		
TIME SUBMITTED	25-JAN-2018 05:22PM (UTC+0000)	WORD COUNT	2612
SUBMISSION ID	82380060	CHARACTER COUNT	17188

Multi-Programming Systems

Jonathan Evans
151713
Multi-User Engineering

25th January

Contents

1. Introduction.....	2
1.1 Introduction to multiprogramming systems	2
1.2 Introducing CPUs.....	2
2. Processes	3
2.1 Process Control block	3
2.2 Multiprogramming functionality.....	4
2.3 Pre-emptive multitasking	4
2.4 Context switches	4
2.5 Interrupts	4
2.6 Inter-process communication.....	5
2.7 Synchronisation.....	6
2.8 Scheduling.....	6
3. Threads.....	7
3.1 System contention scope.....	7
3.2 Processor Affinity	7
3.3 Load balancing.....	7
3.4 Types of thread, communication, and relationships.....	7
3.5 Thread synchronisation	7
3.6 Deadlock	8
4. Summary	9
5. References.....	10
6. Additional Bibliography (Learning resources)	11

1.Introduction

This report will detail the fundamentals of multi-programming systems. It will cover the role processes and threads have within a multi-programming systems, in addition to their use in concurrent and parallel programming.

1.1 Introduction to multiprogramming systems

A multi-programming system is a system that is capable of executing tasks simultaneously or providing the effect of simultaneous execution through rapid task switching. This is achieved through use of context switches, scheduling, interrupts, and inter-process communication, all of which will be described within the following sections.

1.2 Introducing CPUs

The CPU is the component of the computer that is responsible for carrying out arithmetic computations in addition to I/O operations. These instructions are pulled from memory as part of a cycle known as the Fetch and Execute cycle.

The Fetch and Execute cycle describes a CPU's basic functionality. It consists of a continuous loop which ultimately fetches instructions from memory and executes them, fetching more instructions once an instruction has been executed. This is done in three main stages,  fetch, decode, and execute.

The first stage, the fetch, consists of the CPU loading contents of a process into their relevant registers (a place for storing kinds of information). There are three main registers that are used in this process – MAR, the Memory Address Register, which contains addresses of memory locations which data will be fetched from or sent to, MDR, Memory Data Register, which contains data, and IR, Instruction Register, which stores instructions for the CPU to execute.

After a process's contents are loaded into their relevant registers, the CU, Control Unit, determines what is required for completion of the process – such as I/O devices and arithmetic operations.

The final stage of the fetch/execute cycle is to execute the instruction provided to the processor. Following this, is a condition check for an interrupt. This will either result the CPU fetching more instructions of a given process or giving CPU time to another process. [1] [2]

The following sections  will describe processes and threads, outlining what they are, explaining how they interact and execute within a system. Components of multiprogramming will also be detailed. This will be followed up with a summary that will summarise the points made in this paper.

When a process is created, it will have the “New” state, shortly after this, the process will be added to the ready queue, where it will wait for execution, execute, following an interrupt or termination if the process has finished. This kind of functionality allows for multiprogramming to take place.

2.2 Multiprogramming functionality

As described in the introduction, multiprogramming can create the effect of multitasking on a single CPU system. This is achieved through pre-emptive multitasking, which makes use of scheduling, context switches, interrupts, and inter-process communication.

Without multiprogramming, the CPU will be less efficient as it will have to wait for other processes to terminate or it would have to wait for an I/O device to become ready before a process can make use of CPU time.

The following sections will detail components that make multiprogramming possible.

2.3 Pre-emptive multitasking

Past systems achieved multitasking through writing programs that give up CPU time in order for other processes to utilise the CPU. However, in present day computers mostly use pre-emptive multitasking.

Pre-emptive multitasking is the process in which the CPU allows a time slice for a process to execute. This time slice would be very short (usually milliseconds). As a result, the user can carry out other processes at the same time, creating the illusion of parallel execution. [5]

2.4 Context switches

In multitasking systems, processes are commonly switched rapidly to create the effect of multitasking. When a process is swapped out for another, a context switch is saved into memory and another process is loaded.

A context switch stores all current details of a process with the intension of pausing the process and resuming it at a later time. It stores CPU register data, progress from the Program Counter, and relevant data pertaining to the process, then switches to another task, loading the registers and relevant values for that given task. [6]

2.5 Interrupts

Interrupts are used to interrupt a given process, moving its contents on to the stack. There are four kinds of interrupts: timer, error, hardware, and I/O interrupts.

Timer interrupts are triggered by the scheduler when a time slice has expired, such as a time slice provided by the Round Robin scheduler that is discussed in the scheduling section of this report.

Error interrupts occur when an error specifies an interrupt in the code.

Hardware interrupts may occur because of hardware failure.

I/O interrupts may occur because of an I/O operation – such as a keystroke or mouse movement.

2.6 Inter-process communication

Processes can share information using shared memory spaces (through utilising threads), pipes, and message-passing.

Pipes

A program by itself will typically output to a file or stdout (standard output) and will typically receive input from the user or a file. Pipes allow an output of a program to be supplied as an input to another. For example, the UNIX sort function is used to sort input provided to it. The cat function is used to read files sequentially. With a pipe, these can be combined to display a sorted output of a given file.

Message-passing

Message-passing consist of two components – a sender and receiver. For example, a function call returning a value that can be passed into another function. This is one common way of communicating data across processes.

Shared memory

Processes can also share data through use of a shared buffer in memory. This is considered a fast method for accessing shared data as function calls are not required for this transaction to take place.

[7]

2.7 Synchronisation

Synchronisation of processes is crucial in multiprogramming. Two common problems that must be solved when creating multi-process applications are the producer-consumer problem, and the critical section problem.

A Critical-Section within a program would consist of multiple processes which wish to access the same data. There are two techniques that can satisfy these kinds of processes, Mutual Exclusion and Semaphores.

Mutexes (Mutual Exclusion) is a “locking mechanism” that is used to lock all processes accessing shared data and allowing each process access at a time.

A Semaphore is a variable that determines if a process can access shared data. [8]

The producer-consumer problem occurs when a producer process outputs data to a buffer and a consumer consumes data from a buffer. The problem is that consumer must only consume what it can and that it should notify the producer when a buffer is full, or if the buffer is empty, the consumer should sleep. Use of Semaphores can be used for signalling the producer. For example, the consumer can signal the producer when a buffer is full.

2.8 Scheduling

A scheduler determines what process should be allowed CPU time next. There are many kinds of scheduling algorithms for determining this – namely, FCFS, Shortest Job Next, priority-based, shortest remaining time, round robin, and multi-level queues.

FCFS (First-Come-First-Served) picks the process that requests the CPU first. With multitasking operating systems, this is not particularly useful as it may take a long time for a newly added process to execute.

Shortest Job Next (SJN) picks a process based on the lowest CPU burst. This should execute processes quite frequently, however, as new processes are added, the possibility of reaching the biggest jobs lessen, this effect is known as process starvation. [9]

Priority-based scheduling picks the processes with the highest CPU bursts. This will be useful for executing the high CPU tasks. This method may also suffer from process starvation; however, it may not be as bad as the starvation that occurs in SJN scheduling (if there are fewer processes with high CPU bursts).

Shortest Remaining Time prioritises processes that have the least time left to finish execution.

Rounded Robin scheduling allocates time for each process. This can be considered a fair approach as each process is allocated time. As a result, all processes will hopefully have some CPU time, thus lessening the amount of process starvation that can occur.

Multi-Level queues can have a mix of different algorithms. This is used to segment a kind of process with a given algorithm. For example, user-facing elements can be assigned the rounded robin algorithm while a job requiring heavy computation could utilise the priority-based scheduling.

[3] [10]

3. Threads

A thread is a stream of executing code. A process can have multiple threads assigned to it. This achieves faster execution as tasks can be executed in parallel. For example, if two threads are given a balanced set of instructions and data, the time to execute all the instructions should be halved compared to achieving the same result on single threads.

There are two kinds of threads – user level threads and kernel level threads. Kernel level threads handle threads at the kernel level. User level threads are created by the user through a library such as pthreads (POSIX threads). [11] The user level threads are linked to the a LWP (Lightweight process), which may share data amongst over LWPs, which are ultimately mapped to kernel threads.

3.1 System contention scope

Programs on modern operating systems such as Windows or Linux make use of system-contention scope flags for a thread. These determine a priority for a thread that is defined with use of a library such as pthreads. System-contention scope is used to determine which threads should run on the CPU. [12]

3.2 Processor Affinity

A processor affinity defines what thread a process can be run on and if it can migrate to another thread or not. This can be used for assigning tasks, such as assigning kernel level tasks to one thread, and other tasks, such as those created using pthreads, to their own threads. These threads can have a soft affinity, meaning that it prefers to stay on a single CPU core, or a hard affinity, when the cores threads can run on are explicitly determined. [12] [13]

3.3 Load balancing

Load balancing techniques can be used to distribute a load amongst threads. Ideally, an equal distribution of tasks and data will result in faster load times when executed in parallel. [12]

3.4 Types of thread, communication, and relationships

As previously discussed, a system can have two main types of threads – user and kernel level threads. In an operating system, a relationship is established between these threads, such as one-to-one and many-to-many. A thread can communicate synchronisation mechanisms through use of Mutexes and Semaphores, which are discussed in in the following sections. [12]

3.5 Thread synchronisation

Threads can synchronise similarly to processes through use of Mutexes and Semaphores. As previously described, these are mechanisms for synchronising data amongst processes. [8]

3.6 Deadlock

A common problem when implementing a multi-threaded program is known as Deadlock, which is a problem that occurs when a process is dependent on another processes output. This problem can occur when thread synchronisation techniques are used improperly. For example, forgetting to unlock a process. [14]

4. Summary

Multiprogramming systems can achieve true parallel execution or the effect of parallel execution.

Processes

A process is code in execution that is represented as a PCB (Process Control Block) within a system. The PCB contains crucial information such as the PC (Program Counter) values, relevant registers, a pointer to the next process, a context switch for pausing of resuming execution. All this information is used by the scheduler to pause/resume tasks.

Scheduler

The scheduler is responsible for choosing what process should be picked after swapping out the current process. Most operating systems implement the multi-queue scheduler that provides a mix of algorithms such as Round Robin, which uses time slices to allocate CPU time, and SJN (the Shortest Job Next).

Threads

A thread is a unit in which a process can run on. A system can have many threads which can execute concurrently, however, multi-processor system can achieve true parallel execution of code. Threads can migrate between processor cores and have an affinity set to them. Synchronisation techniques such as Mutexes and Semaphores can be used for making sure the correct thread has access to data at the right time, preventing other threads from handling data while one thread is using it.

An example of software that benefits massively from concurrent threads would be a web server, where many requests must be dealt with as soon as possible. On a multicore system, a web server will benefit substantially through executing concurrent tasks across multiple cores in parallel, this significantly reduces the workload for a single core as the workload is ideally distributed amongst many cores, thus improving the performance of the application.

5. References

- [1] BBC, "Low-level operations and computer architecture," [Online]. Available: <https://www.bbc.co.uk/education/guides/zsnbr82/revision/12>. [Accessed 24 January 2018].
- [2] B. Agarwal. [Online]. Available: <https://realboyemulator.files.wordpress.com/2013/01/ife.pdf>. [Accessed 24 January 2018].
- [3] tutorialspoint, "Operating System - Processes," [Online]. Available: https://www.tutorialspoint.com/operating_system/os_processes.htm. [Accessed 24 January 2018].
- [4] P. B. G. G. Abraham Silberschatz, "3.1.3 Process Control Block," in *Operating System Concepts 9th ed*, Wiley, 2012.
- [5] Techopedia, "Preemptive Multitasking," [Online]. Available: <https://www.techopedia.com/definition/8949/preemptive-multitasking>. [Accessed 25 January 2018].
- [6] tutorialspoint, "Operating System - Process Scheduling," [Online]. Available: https://www.tutorialspoint.com/operating_system/os_process_scheduling.htm. [Accessed 25 January 2018].
- [7] P. B. G. G. Abraham Silberschatz, "Interprocess Communication," in *Operating System Concepts 9th ed*, Wiley, 2012.
- [8] EDSGER W. DIJKSTRA, "THE STRUCTURE OF THE "THE" MULTIPROGRAMMING SYSTEM," p. 232, 1968.
- [9] Geeksforgeeks, "Operating System | Starvation and Aging in Operating Systems," [Online]. Available: <https://www.geeksforgeeks.org/starvation-aging-operating-systems/>. [Accessed 25 January 2018].
- [10] P. B. G. G. Abraham Silberschatz, *Operating System Concepts*, 9th ed, Wiley, 2012.
- [11] M. Kerrisk. [Online]. Available: <http://man7.org/linux/man-pages/man7/pthreads.7.html>. [Accessed 25 January 2018].
- [12] P. B. G. G. Abraham Silberschatz, "Thread Scheduling," in *Operating Systems Concepts 9th ed*, Wiley.
- [13] Linux Journal, "CPU Affinity," 1 July 2003. [Online]. Available: <https://www.linuxjournal.com/article/6799>. [Accessed 25 January 2018].
- [14] TechTarget, "Deadlock," [Online]. Available: <http://whatis.techtarget.com/definition/deadlock>. [Accessed 25 January 2018].

6. Additional Bibliography (Learning resources)

- Operating System Concepts, 9th edition
- C++ Concurrency in Action

Multi-programming systems

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

73/100

Instructor

PAGE 1

PAGE 2

PAGE 3



Comment 1

:



Comment 2

I really like how you've structured this report in general - it has a considered narrative flow.

PAGE 4

PAGE 5

PAGE 6

PAGE 7



Comment 3

By convention, this is written the other way around, in the form:

Full term (abbreviation)

So in this case:

Mutual exclusions (mutexes)



Comment 4

Round

**Comment 5**

Actually not very common at all! Just nightmarish to resolve when it happens. Usually hardware-related.

GRADING FORM: SMU GENERIC_GJ

JONATHAN IEUAN EVANS**73****RESEARCH-INFORMED LITERATURE**

A generally good use of sources - though you do draw rather heavily from a number of online sources of questionable veracity. I'd actually prefer to see a slightly smaller number of more academically credible sources. As a rule, sources should come from: Peer-reviewed journals >> Books and other non-peer reviewed formally published material = Authoritative website sources (e.g. Fedora's documentation when talking about Fedora) >> non-authoritative web sources. Your literature review, however, is very well-formed and rounded. The structure is logical, with an excellent narrative - it's only your application of literature which needs a little work. I'm happy to discuss this with you face-to-face to help you to improve in this respect for future work.

35

KNOWLEDGE AND UNDERSTANDING



Very good evidence of knowledge and understanding - an excellent, detailed report which covers specifics in the right areas where needed. Well done, Jonathan.

23

ANALYSIS



Your report would benefit from a little more synthesis of material - rather than using a single source to say a single thing, try to read around the topic from multiple sources, then cite them all. Some very good concluding remarks, both throughout the report, and in the conclusions section.

6

PRACTICAL APPLICATION AND DEP.



N/A - no marks assigned.

SKILLS FOR PROFESSIONAL PRAC.



You write extremely well - I really enjoyed reading this work. Excellent spelling, punctuation and grammar, and a nicely-presented document. You might actually benefit from a few more figures in places, but this is a minor point. An very good piece of work overall, well done!

9