Week 8: threads and scheduling

CS211: Programming and Operating Systems

Wednesday and Thursday, 04+06 March 2020

This week, in CS211, . . .

Announcements

1. Lab 5 will take place this Friday (6 March), 9–12. Details will be posted to the website tomorrow.
2. Our “field trip” to the Computer and Communications Museum of Ireland is confirmed for 1pm, Thursday 12 March. For more, see `http://www.nuigalway.ie/visitorscomputermuseum/`. This will be **instead** of the usual lecture. We’ll meet at the Museum. Directions are available from the link above.
3. Thanks for the **feed-back**. Most was really positive. Except:
   - many of you would welcome more timely grading of assignments (sorry – will try to do better!!!)
   - a few of you would prefer different lab times, or differently organised (sorry – the University is planning to do better, just not any time soon!)
   - more support in lab (will try to get tutors to spend more time with different people).

Introduction to threads

To date we have always assumed that a process has a single thread of control. However this need not be the case...

Example 1: A word-processor may simultaneous display text, check spellings, read keystrokes, and send a document a printer. All these could be achieved by separate processes, but would need to share certain resources, particularly program data. So, instead of using a traditional process, we might use ***threads***...

Another example: a web server (deamon) may run multiple threads to serve requests. All share the same data, & preform the same tasks, but for deifferent clients.

Introduction to threads

From the text-book, Chapter 26 (Concurrency and Threads) `http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf`

Threads

Instead of our classic view of a single point of execution within a program, a **multi-threaded** program has more than one point of execution.
Perhaps another way to think of this is that each thread is very much like a separate process, except for one difference: they share the same address space and thus can access the same data.

Introduction to threads

A *thread* (or *lightweight process*) is a basic unit of CPU utilisation. Rather than a process begin dedicated to one sequential list of tasks it may be broken up into threads. These consists a set of (distinct)

- **program counter (“PC”)** – the next instruction to be executed
- **register set** – operands for CPU instructions
- **stack** – temporary variables, etc.

Threads belonging to the same process share

- code section, = instruction set.
- data section, = memory (other than stack & registers!
- operating-system resources, (e.g., files)

collectively known as the *task*.

A traditional process is sometimes known as a ***heavyweight process***.

Introduction to threads

Threads differ from the child processes that we looked at last week (created using `fork()`): though a child might inherit a copy of its parent's memory space, a thread shares it.

Example 2: Java uses threads to (among other thing) simulate asynchronous behaviour. If a Java program attempts to connect to a server, it will block until it makes the connection. In order to perform a time-out it sets up two threads – one to make the connection and another to sleep for, say, 60 seconds. When it wakes, it generates an interrupt if a connection has not been established.

Instead of fork() we use pthread().

Introduction to threads

The reasons that an OS might use threads over heavy-weight processes are:

- (i) **Parallelization:** Utilisation of multiprocessor architecture – each thread can execute on a different processor.
- (ii) **Responsiveness**: a part of a process may continue working, even if another part is blocked, for example waiting for an I/O operation to occur.
- (iii) **Resource sharing**: For example, you can have several hundred threads running at the same time. If they were separate procs, each would need their own memory space. However, as threads they can share that resource.
- (iv) **Economy:** Thread creation is somewhat faster than processes creation. Also context switching is faster.
- (v) **Efficiency:** Threads belonging to the same proc share common memory space so they do not need support from the OS to communicate. (so: don't need pipes, for example)

User and Kernel Threads

Threads may be one of two types: ***User*** or ***Kernel***.

- User threads are implemented by a thread library: a collection is routines and functions for manipulating threads. Hence, user threads are dealt at the compiler/library level – above the level of the OS kernel.
  Advantage: User threads are quick to create and destroy because systems calls are not required.
  Disadvantage: If the kernel itself is not threaded, then if one thread makes a blocking system call, then the entire process with be blocked.

The `POSIX Pthreads` are user threads found on most Unix systems.

Read this some time . . .

User and Kernel Threads

- Kernel Threads: thread creation, management and scheduling is controlled by the operating system. Hence they are slower to manipulate then user threads, but do not suffer from blocking problems: if one thread performs a blocking system call, the kernel can schedule another thread from that application for execution.
  Windows has support kernel threads.

Scheduling Processes Single Queue == 1 CPU

For more, see Chapter 7 of the Textbook

CPU scheduling is the basis of multiprogrammed operating systems, and so of multitasking operating systems. The underlying concepts of CPU scheduling are:

Goal: Maximum CPU utilisation – i.e., there should be something running at any given time.

Based around: The *CPU burst–I/O Burst Cycle* – a running process alternates between executing instructions (CPU burst) and waiting for interaction with peripheral devices (I/O burst)

Depends on: the burst distribution – is a given proc predominantly concerned with computation or I/O. Also, a proc usually begins with a long CPU burst and almost always ends with a CPU burst.

Preempting

The CPU Scheduler Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. This happens when a process: for I/O.

(i) Switches from running to waiting state
(ii) Terminates.
(iii) Switches from running to ready state
(iv) Switches from waiting to ready

non-preentive : OS does not remove the process from a running state.

Scheduling that depends only on (i) and (ii) is ***non-preemptive***. All other scheduling is "***preemptive***."

"interupt"

Preempting

If scheduler uses only *non-preemptive* methods, then once a process is allocated the CPU it will retain it until it terminates or changes its state. E.g., Windows 3.1, versions MacOS prior to v8.

For preemptive methods, the scheduler must have an algorithm for deciding when a control of the CPU must be passed to another process.

The Dispatcher

The **Dispatcher** module gives control of the CPU to the process selected by the short-term scheduler; this involves:

1. switching context (see next week!).
2. switching to user mode
3. jumping to the proper location in the user program to restart that program

The dispatcher gives rise to the phenomenon of ***Dispatch latency***, i.e., the time it takes for the dispatcher to stop one process and start another running.

. . . .

We will make the <u>very</u> simplifying assumption that there is no dispatch latency, and that we can switch instantaneously between processes.

Scheduling Criteria

The choice/evaluation of scheduling algorithms is usually based on the following:

- CPU utilisation – keep the CPU as busy as possible
- Throughput – the number of processes that complete their execution per time unit
- Turnaround time – the time that elapses between a process entering the ready state for the first time and leaving the running state for the last time.
- $\boxed{\text{Waiting time}}$ – amount of time a process has been waiting in the ready queue
- $\boxed{\text{Response time}}$ – amount of time it takes from when a request was submitted until the first response is started (but not including the time it takes to conclude the response)

In general one would try to ***maximise*** CPU utilisation and throughput, and ***minimise*** turnaround time, wait time and response time.

Algorithms

There are many **Scheduling Algorithms**, including

1. First-Come-First-Served (FSFS)
2. **Shortest-Job-First** (SJF)
3. Shortest Time-to-Completion First (STCF)
4. Round-Robin (RR)
5. Priority Scheduling ⟵ (not considered)
6. **Multilevel queuing** ⟵ (not considered)

Non-preemptive

Preentive.

more than one CPU.

In describing each of these, we'll consider a few examples. For each example we'll assume that each proc has a single CPU burst, measured in milliseconds.

Algorithms will be compared by calculating various metrics.

~~**Single burst means:**~~

Finished here Wed

Scheduling metrics

We would like to compare algorithms, and determine which is "best". However, there are many choices of what qualifies as best. Here are a few **METRICS**.

1. Turnaround time – the time that elapses between when process arrives in the system, and when it finally completes.
2. Wait time – the amount of time between when a process arrives, and when it completes, that it spends doing nothing.
3. Response time – the time that elapses between when process arrives in the system, and when it executes for the first time.

Order of Importance

1. Wait time
2. Response time
3. Turnaround.

Algorithms — First-Come, First-Served (FCFS)

Also called **First In, First Out** (as in OSTEP), the simplest algorithm is FCFS: the first proc that requests the CPU is given it.

Advantage: This is the simplest algorithm to implement, requiring only a **FIFO** queue. It is non-preemptive.

Disadvantage: It has a long average wait time, and suffers from the ***convoy effect***.

Algorithms — First-Come, First-Served (FCFS)

What are the average turnaround, wait and response times for **FCFS** in the following two scenarios?

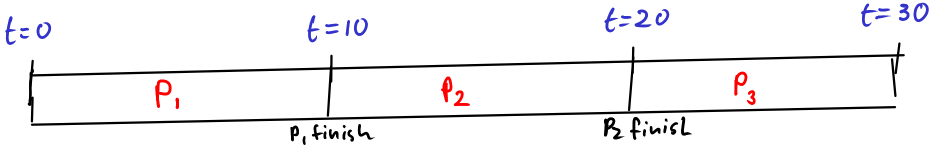
Example 1

Proc	Arrival Time	Burst Time
P_1	0	10
P_2	0	10
P_3	0	10

Example 2

Proc	Arrival Time	Burst Time
P_1	0	19
P_2	0	10
P_3	0	1

Example 1 time line (Gantt Chart).



Wait times: P_1 is 0, P_2 is 10, P_3 is 20.

Average wait time is $(0+10+20)/3 = 10$.

Shortest-Job-First (SJF)

We can see that we could greatly improve the wait and response times for Example 2, if we let P_3 run first, then P_2, and then P_1.

That algorithm is called **Shortest Job First** (SJF): it runs the processes in increasing order of their (expected) burst time.

Advantage: **SJF** is optimal – it gives the minimum average waiting time for a given set of processes. Also: this version in ***non-preemptive***

Disadvantage: It cannot be employed in a realistic setting because we would be required to known the length of next CPU Burst before it happens.

One can only estimate the length. This can be done by using the length of previous CPU bursts, using some form of averaging.

Shortest-Job-First (SJF)

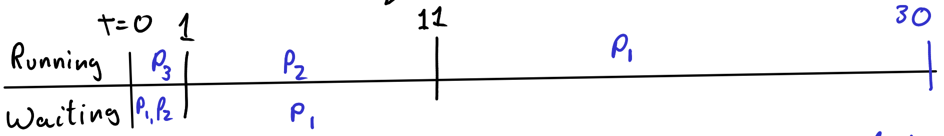
What are the average turnaround, wait and response times for **SJF** for the scenarios we used earlier?

Example 1

Proc	Arrival Time	Burst Time
P_1	0	10
P_2	0	10
P_3	0	10

Example 2

Proc	Arrival Time	Burst Time
P_1	0	19
P_2	0	10
P_3	0	1



Wait times for P_1, P_2, P_3 are 11, 1 & 0, respectivly.
Averag is $(11+1+0)/3 = 4$. Much better!!

Again, Response is the same as Wait time.

Algorithms Shortest Time-To-Completion First (STCF)

In all the previous examples, we assumed that the process all arrived at the same time. That is not realistic. Consider how **SJF** would work with the following variant on the previous examples:

Example 1

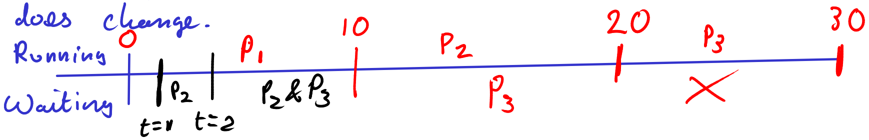
Proc	Arrival Time	Burst Time
P_1	0	10
P_2	1	10
P_3	2	10

Example 2

Proc	Arrival Time	Burst Time
P_1	0	19
P_2	1	10
P_3	2	1

While there is no change for the outcome to Example 1, we now find that SJF is no longer optimal:

That is, no change in the order. Wait time does change.



Algorithms   Shortest Time-To-Completion First (STCF)

The solution is to introduce a **preemptive** version of SJF, giving **Shortest Time-To-Completion First (STCF)**: whenever a new process arrives to run, we check if it has a lower Burst time than the current running one. If so, we **preempt** that, and let the new one take over until it has finished, or a new one has arrive.

(It is also called ***Shortest-Remaining-Time-First*** or ***Preemptive Shortest Job First***).

Advantage – "more optimal" than SJF when processes arrive at different times.

Disadvantage – needs preempting.

Algorithms — Round Robin (RR)

Each process gets a small unit of CPU time called a ***time quantum*** or ***time slice*** –usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units. (***Why?***)

The size of the *quantum* is of central importance to the **RR** algorithm. If it is too large, then its is just the FCFS model. If it is too low, them too much time is spent on context switching.

Algorithms

Suppose the following arrive in the following order:

Proc	Arrive Time	Burst Time
P_1	0	20
P_2	2	15
P_3	4	10
P_4	6	5

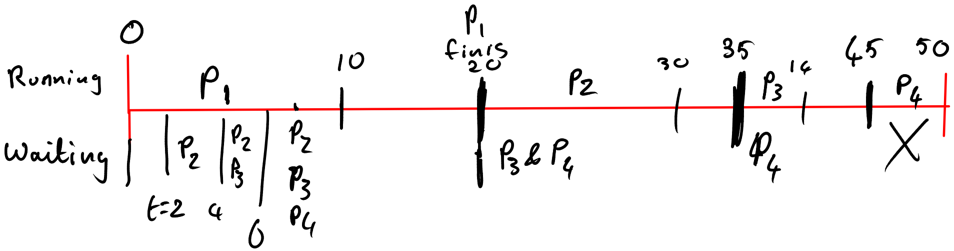
Calculate the

(a) ***Average Turnaround Time***,
(b) ***Average Wait Time***, and
(c) ***Average Response Time*** for

1. FCFS
2. SJF
3. STCF
4. RR with $q = 10$

Finished here

FCFS



TQ

Metric	P1	P2	P3	P4	Ave
Turnaround.	20	33	41	44	? 34.5
Wait	0	18	31	39	22.

Example

(Shortest Job First) SJF

(Shortest Time to Completion First) STCF

(Round Robin with $q = 10$

Exercise

Exercise (8.1)

(This is taken from the CS211 Semester 2 from 2017/2018)
Given the data below for four processes, determine the scheduling result for the policies of

1. *Round Robin (with time quantum 4)*
2. *First Come First Served*

(All times are in seconds).

Process	*Arrival time*	*Process duration*
P1	*3*	*5*
P2	*1*	*3*
P3	*0*	*8*
P4	*4*	*6*

(c) Calculate the average turnaround time and average waiting time for these examples.