# Report 1 – Concurrent Programming

**Lee Zong Yang, 100073484**

## Week 1 – Computing Paradigms

There are different types of computing paradigms, namely concurrent computing, parallel computing, distributed computing, cluster computing, grid computing, cloud computing, and fog/edge computing.

### *Concurrent computing & Parallel Computing*

I am going to start with concurrent computing first as it is invented earliest and is the crucial feature of the operating system. A long time ago, there is concept of operating system and the computers executed a single program from start to finish with all resources fully provided to the program. Thus, the development of software was extremely difficult as the programmer needs to be fully aware of the internal of the computer. Moreover, it is a waste of expensive and scarce computer computing power by running only a single program at a time.

The invention of an operating system allows the running of multiple programs together by isolating each and allocate resources to each. Parallel computing also made possible thanks to this feature, but it requires multiple processors or computers, thus it is not widely adopted at that time. Concurrent computing is feasible for a single processor by allows each program to own the CPU for a fair share amount of time and appears to the user that these programs are executed simultaneously, although it is not. Since the programs are sharing the same resources of the computer, thus there must be a way to make sure correct execution of programs, which is coined the term “concurrency control”.

Concurrent computing and parallel computing may appear the same to the end-user, but they differ in the computer’s point of view. Parallel computing requires a computational task to be split into multiple independent tasks and executes at the same time and thus the number of CPU or computers need to be the same as the tasks. Later, the final result will be merged from the results obtained from the independent tasks.

### *Distributed Computing*

Speaking of distributed computing, I do say it is focused on the coordination of several loosely coupled computers when carrying out parallel computing. The system will consist of many computing devices with their operating system and memory but will be perceived as one to the user. Therefore, messages over network are the main way for those devices to communicate with each other as there is no shared memory between those devices. Distributed computing provides better reliability as the risk of single-point failure is reduced by having the system keep functioning even when a small number of nodes malfunction. Better performance is achieved too as it employs parallel computing and problems that deal with huge amounts of data that cannot fit on one device could be solved by splitting the data into several devices.

### *Cluster Computing & Grid Computing*

The organization of multiple devices for distributed computing could be classified into cluster computing and grid computing. The computer cluster consists of multiple computers with the same hardware and software connected with a high-speed local area network, but the computer grid consists of multiple computers without unified hardware and software connected with the internet.

Thus, computers for cluster computing must be located close to each other while computers for grid computing must be located in different regions over the world. The broken-down task for cluster computing is the same but different tasks could be carried out parallelly in grid computing.

### *Cloud Computing & Fog/Edge Computing*

Instead of maintaining infrastructure by oneself, one can just get the computation required from the service provider through the internet, coined as Cloud Computing. Amazon Web Service, Huawei Cloud, Google Cloud Platform are examples of cloud computing provider. There are three types of services, IaaS which provide rental of storage, networking, and virtualization, PaaS which provide a platform with pre-built hardware and software, SaaS which provide ready-made software as a service. Although the end-user does not require knowledge of distributed computing to make use of the cloud computing service, the service provider does use distributed computing to make sure the reliability, availability, and maintainability of the services as many people are using the services concurrently. To reduce the load on the server-side, a new computing paradigm had been proposed, namely fog/edge computing, which carries out the processing close to data-generating devices instead of the server on the cloud. This paradigm is deployed heavily in the Internet of Things application.

# Week 2: Process

In layman's term, a running GUI application is coined as a program. However, a program in Computer Science is just a lifeless thing, a sequence of machine instructions stores on the hard disk waiting to be executed. The correct term for a running program is process. The operating system will load the instructions from the hard disk into memory, read the instructions into the CPU, and carries out the execution. As discussed in week 1, the operating system is the one that provides the concurrent execution of multiple programs by virtualizing the CPU even though there is only one processor. The OS will schedule the processes so that one process will stop after running for some time and allows another process to run, creating an illusion of concurrent execution because the occupation time of CPU for each process is extremely short, a technique known as time-sharing.

To implement virtualization of the CPU, we need to consider low-level mechanism and high-level policy. The policy is an algorithm to determine the execution of the process while the mechanism is to ensure the correctness of policy. For example, context-switch, which allows the OS to switch the running program to another is a form of mechanism; scheduling policy which will be discussed in week 3 is a form of policies.

### *Why we need processes?*

When the computer was invented, it is very simple. The computer will execute the one-time command entered by the user and stopped if the user entering a command, which leads to low efficacy of computing resources. Then, batch processing is invented in which numerous commands are compiled into a program and the computers will run it continuously. However, there is still under-utilization of CPU i.e. a running program A is blocked while waiting for I/O to complete, and thus the CPU is doing nothing. In this case, we wish to run another program on CPU while program A is waiting for I/O to complete. Thus, process is invented to virtualize the CPU, since different programs are using a single physical memory and CPU, but different processes should not affect each other. Memory virtualization is achieved through the OS allocates a portion of memory to the process and each process can only read/write its memory. Furthermore, the state and resources of the process are saved when the CPU switches to another process and is restored when the process is switched back.

### *How a process is working?*

A process consists of an image of a program, CPU state (stack pointer(SP), program counter(PC), registers, etc), memory (static data, program instructions, heap and stack), and operating system state (opened files, accounting statistics, etc).

### *Process Address Space (memory)*

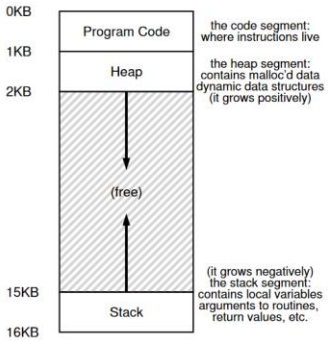


Figure 2.1: An Example Address Space (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p132)

Figure 2.1 shows a simplified address space of a process, consists of program code, heap, and stack. Each creation of process accompanies the creation of distinct address space.

***Context Switch***

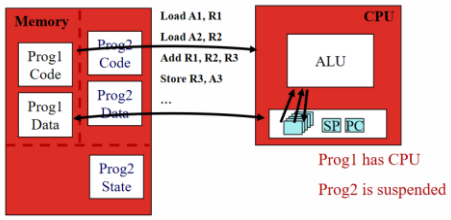


Figure 2.2: Phase 1 (Swinburne University of Technology 2, 2021, p21)

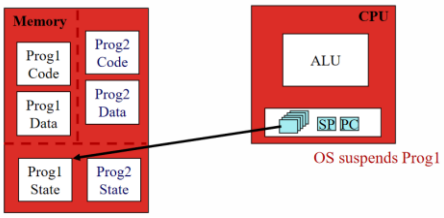


Figure 2.3: Phase 2 (Swinburne University of Technology 2, 2021, p22)

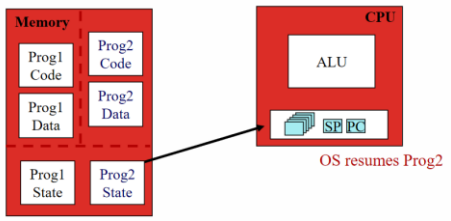


Figure 2.4: Phase 3 (Swinburne University of Technology 2, 2021, p23)

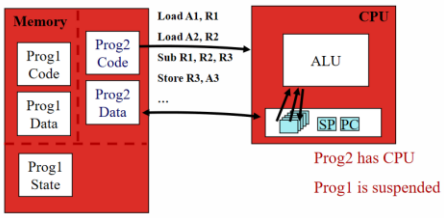


Figure 2.5: Phase 4 (Swinburne University of Technology 2, 2021, p24)

Figure 2.2, 2.3, 2.4, and 2.5 illustrate how context-switch between two process is carried out. In phase 1, Prog1 is the running process on the CPU and Prog2 is suspended. Then, OS decided to switch to Prog2 so it saves registers, program counter, stack pointer of Prog1. At phase 3, the registers, program counter, and stack pointer of Prog2 are restored. Eventually, the CPU starts the running of Prog2 by loading the Prog2's code and data from memory.

***Process States***

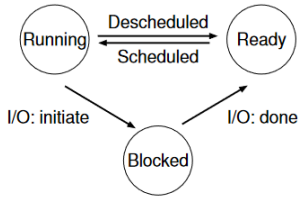


Figure 2.6: Process: State Transitions (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p32)

Three process states are shown in figure 2.6, namely running, ready, and blocked. When a process is running on a CPU, its state is running. If a process's progression requires some condition to be

fulfilled, then it is in a blocked state. If a process is ready to run but the CPU has decided not to run it then it is in ready state. Below are two examples of process state transition, figure 2.7 illustrates the case without I/O while figure 2.8 illustrates the case with I/O. We can see that $Process_0$ in figure 2.7 runs to completion, but in figure 2.8, $Process_0$ changes to blocked state and CPU switches the running process to be $Process_1$ due to $Process_0$ initiates I/O.

| Time | $Process_0$ | $Process_1$ | Notes |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | |
| 4 | Running | Ready | $Process_0$ now done |
| 5 | – | Running | |
| 6 | – | Running | |
| 7 | – | Running | |
| 8 | – | Running | $Process_1$ now done |

Figure 4.3: **Tracing Process State: CPU Only**

Figure 2.7: Tracing Process State: CPU Only (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p32)

| Time | $Process_0$ | $Process_1$ | Notes |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | $Process_0$ initiates I/O |
| 4 | Blocked | Running | $Process_0$ is blocked, |
| 5 | Blocked | Running | so $Process_1$ runs |
| 6 | Blocked | Running | |
| 7 | Ready | Running | I/O done |
| 8 | Ready | Running | $Process_1$ now done |
| 9 | Running | – | |
| 10 | Running | – | $Process_0$ now done |

Figure 2.8: Tracing Process State: CPU and I/O (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p33)

***Other process states:***

There are initial/created and final/terminated state too. Obviously, the initial/created state is the state when a process is being created. A process is in final/terminated state when it has ended but the OS has not freed it yet. Final/terminated state allows the parent process to examine the return value of the child process.

***Process APIs***

In C, there are fork() to create a new process by cloning the parent process, wait() to pause the parent process until the child finishes and retrieves the return value of the child process, and exec() to run a new program by replacing the code and data of current process.

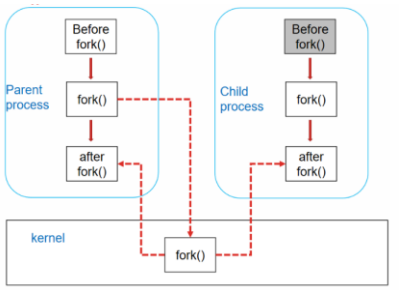


Figure 2.9: Fork() flow (Swinburne University of Technology 2, 2021, p37)

By calling fork(), the OS will allocate a new chunk of memory and kernel data structures for the new process, copy the original process into the new process and add the new process to the list of running process. When all the initializations are done, the parent process will execute the code just after fork() and so too the child process, as shown in figure 2.9.

Since the process identifier of the parent and child process is different, the control structure shown in figure 2.10 can be used to provide different instructions for the parent and child process after fork().

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

Figure 2.10: Calling fork () (p1.c) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p32)

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

Figure 2.11: Output 1 of the program shown in figure 2.10 (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p42)

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

Figure 2.12: Output 2 of the program shown in figure 2.10 (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p43)

There could be two different outputs from the program shown in figure 2.10. We could make the output deterministic by using wait () as shown in figure 2.13, and thus the child process will run to completion first.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {               // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 3.12: Calling fork () And wait () (p2.c) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p43)

Now we will talk about how to use exec () to run a different program in the child process. The method is simple, just call exec () in the child process, then the OS will clear out the program code of the current program in the child process and insert the code of the program specified in the exec (). New memory allocation is issued too to fit the space requirements of the new program.

What shown in below figure 2.14 is a program that run a word count program in child process and the whole control flow is shown in figure 2.15.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {                 // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 2.14: Calling fork (), wait (), And exec () (p3.c) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p43)

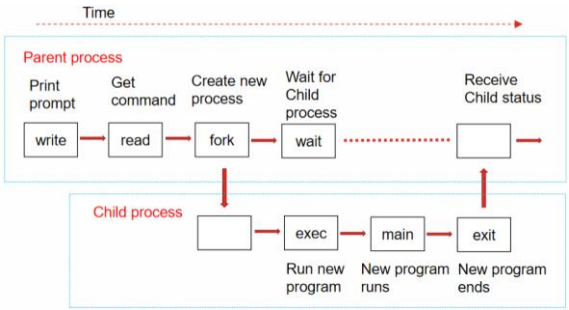


Figure 2.15: control flow of running a new program in child process (Swinburne University of Technology 2, 2021, p43)

## Week 3: Scheduling

This week is about the OS scheduler and the policies it employs from a high-level perspective. Have a good understanding of what workload the scheduler deals with is important before delving into the different variant of scheduling policies. Jobs, an alternative name for process, make up the workload and below are the assumptions about it from Operating System: Three Easy Pieces. Most of the assumptions are unrealistic in practice though. (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p72)

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU (i.e., they perform no I/O)
5. The run-time of each job is known.

### *Scheduling Metrics*

There are certain metrics when comparing different scheduling policies, which are turnaround time, response time, CPU utilization, and missed deadlines. Turnaround time measure the time taken for the job to complete after arrival of the job, $T_{turnaround} = T_{completion} - T_{arrival}$. Response time measure the time taken for the first run of the job after the arrival of the job, $T_{response} = T_{firstrun} - T_{arrival}$.

### *First In, First Out (FIFO)*

First In, First Out (FIFO) scheduling is the most basic and simple algorithm. Provided that assumption 2 is valid but there will be a minor deviation for the arrival time of each job, a case where there are three jobs, A, B, and C arrive is shown in figure 3.1. A arrived first, followed by B with little to no delay, and then C. Each job will take 10 seconds to complete.

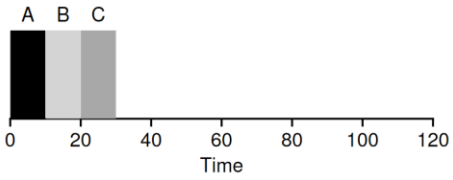


Figure 3.1: FIFO Simple Example (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p73)

With FIFO, the oldest entry will be processed first, and thus A is executed first and completes at 10, then B at 20, and C at 30. The average turnaround time for these three jobs is (10 + 20 + 30)/3 = 20 seconds, a good metric. However, FIFO performs badly when the completion time of each job differs significantly. Consider that A will take 100 seconds to complete, B and C will run for 10 seconds each, the time to completion is shown in figure 3.2.

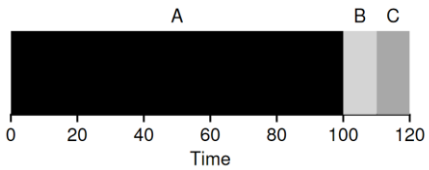


Figure 3.2: Why FIFO is Not That Great (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p73)

The average turnaround time is (100 + 110 + 120)/3 = 110, clearly a painful one. The convoy effect could be used to explain the cause for this problem, the time to be served for customers with little goods purchased will be long if there is a customer with carts of purchased goods queueing in front of them. Therefore, a better algorithm is required for jobs with different running time.

### *Shortest Job First (SJF)*

A simple solution that runs the shortest job first provides optimal average turnaround time given that all jobs arrive at the same time.

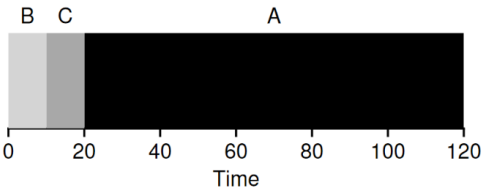


Figure 3.3: SJF Simple Example (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p74)

Assume the same scenario that caused bad average turnaround time in FIFO is now working with Shortest Job First scheduling, the result is shown in figure 3.3. Average turnaround time reduces from 110 seconds to (10 + 20 + 120)/3 = 50 seconds by running B and C before A.

However, assumption 2 (all jobs arrive at the same time) will not hold most of the time in practice, thus there is a possibility for the worst case scenario as shown in figure 3.4.

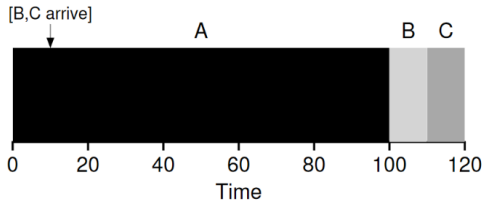


Figure 3.4: SJF with Late Arrivals From B and C (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p75)

In this case, B and C arrive 10 seconds later than A, and thus A will run first and run to the completion without any interference, leading to the average turnaround time of (100 + (110 – 10) + (120 – 10))/3 = 103.33 seconds, a number that is not much better than FIFO's 110 seconds.

### *Shortest Time-to-Completion First (STCF)/Preemptive Shortest Job First (PSJF)*

By relaxing assumption 3 (all jobs must run to completion) and introduce the ability to preempt the current job and change to the job that has the least amount of time to finish, the problem faced by SJF could be addressed.

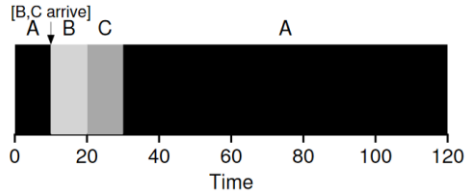


Figure 3.5: STCF Simple Example (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p76)

Assume that B and C arrive 10 seconds later than A, as shown in figure 3.5, the scheduler preempts A and run B and C to completion first, result in an average turnaround time of ((120 – 0) + (20 – 10) + (30 – 10))/3 = 50 seconds, which is the optimal time.

### *Response Time of FIFO, SJF, PSJF*

All these scheduling algorithms suffer from bad response time. For an example, the average response time of scenario shown in figure 1 is (0 + 10 + 20)/3 = 10 seconds. C has a response time of 20 seconds, which means the user must wait 20 seconds for the program to start after he/she

started the program or the response for mouse click/keyboard input will require 20 seconds. Therefore, Round Robin was proposed as a scheduling algorithm with a good response time.

***Round Robin***

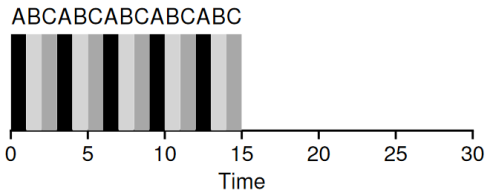


Figure 3.6: Round Robin (Good For Response Time) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p77)

As we can see in figure 3.6, Round Robin runs each job for a time slice, then switches to the next job in the queue, repeatedly till all the jobs are finished. The mechanism underlying is through the timer interrupts, and thus the length of a time slice must be a multiple of the timer-interrupt period. Assume that the time slice is 1 second, then average response time of PR is (0 + 1 + 2)/3 = 1 second, better than SJF's (0 + 5 + 10)/3 = 5 seconds (Figure 3.7).

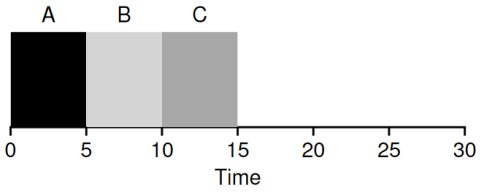


Figure 3.7: SJF Again (Bad for Response Time) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p77)

Although we want the length of the time slice to be as short as possible so the response time is better, this is impossible because the cost of context switching will occupy a larger proportion of the total time and result in a higher actual performance impact. Therefore, amortize analysis should be carried out to find the optimal parameter for the time slice.

Unfortunately, Round Robin's turnaround time is the worst, 13 for A, 14 for B, and 15 for C, as shown in figure 3.6. An inevitable trade-off for good response time.

***Scheduling: The Multi-Level Feedback Queue***

All the previous discussed scheduling algorithm could not achieve good turnaround time and good response time together. Moreover, assumption 5 (the run-time of each job is known) generally does not hold in real practice, and thus algorithms like SJF or STCF could not work.

Luckily, Multi-level Feedback Queue (MLFQ) was proposed as a scheduling algorithm with good performance in both metrics without the need to know the job length in advance. Based on the past working history of the job, MLFQ could predict the behaviour of the job provided that the job has phases of behaviour.

***MLFQ: Basic Rules***

There are few queues, each with a different priority level and a job will be assigned into one of the queues at any given time. MLFQ will execute the job with the highest priority solely if there is only one and employ round-robin scheduling if there are more than one job that share the same highest priority.

The assigning of priority level to the job is done dynamically based on the observed behaviour of the job. A process with higher interactivity will have a higher priority level i.e. a job frequently relinquishes the CPU while waiting for the keyboard's input while keeping its priority level while a job occupied CPU for a long duration will have its priority level reduced by MLFQ.

***Attempt #1: How To Change Priority***

A new job will be placed into the queue with the highest priority. A job's priority level will be reduced if a job uses up an entire time slice for a single run. If a job relinquishes the CPU before using the entire time slice, then its priority level will remain unchanged.

*Example 1: A Single Long-Running Job*

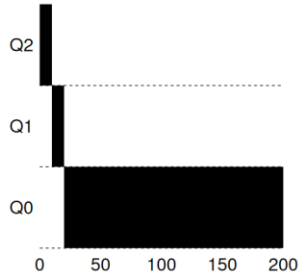


Figure 3.8: Long-running Job Over Time (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p88)

From figure 3.8, we can see that the job's priority is reduced by one each time it uses up the 10 ms time slice, eventually reaches the queue with the lowest priority where it remains.

*Example 2: Along Came A Short Job*

In this section, we will discuss the case with 2 jobs, where A is a length CPU-intensive job while B is a short-running interactive job.

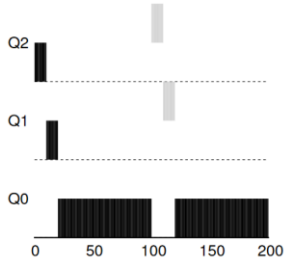


Figure 3.9: Along Came An Interactive Job (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p88)

A is represented by black colour and B by grey colour. A arrives at $T = 0$ then move down to Q0 after two time slices. At $T = 100$, B arrives and is inserted into Q2, and manages to finish before reaching the bottom queue as its run-time takes only 20 ms.

We can see that MLFQ approximates SJF by running the short job over a long job but does not require a priori knowledge of job length. All jobs that arrive will have the highest priority level initially, but the short job will complete quickly while the lengthy job will slowly move down the queues.

*Example 3: What About I/O ?*

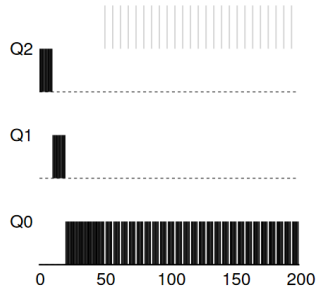


Figure 3.10: A Mixed I/O-intensive and CPU-intensive Workload (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p88)

One of the motives for MLFQ is to provide a good response time, and thus high interactivity job will need to stay in high priority queue. This is accomplished by the rule that maintains the priority level of the job if the job does not use up the entire time slice. By waiting for user input from the I/O device, a job will frequently relinquish the CPU, an indication of high interactivity job.

***Problems With Our Current MLFQ***

Although the current implementation of MLFQ seems fine, it does contain serious flaws. First, those jobs in the lower priority queue will suffer from starvation if there are too many jobs in the higher priority queue. Second, the current rule which allows the job to retain its priority level if the time slice is not used up has a serious security loophole, one can program a program to game the scheduler by relinquishing the CPU just before the time slice is over. Third, a job in the lower priority queue may change its behaviour into a high interactive job but there is no mechanism to promote the job back to the high priority queue.

***Attempt #2: The Priority Boost***

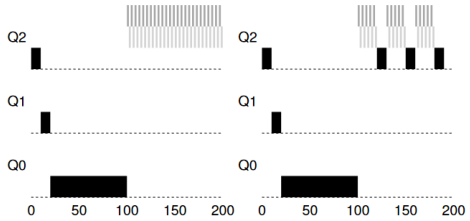


Figure 3.11: Without (Left) and With (Right) Priority Boost (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p90)

A simple solution to solve starvation is to move all the jobs to the highest priority queue periodically. The graph on the left in figure 3.11 shows that the job in the lowest priority queue gets zero chance to execute after two short jobs arrive, but the job in the right graph could make some progress after the arrival of the two short jobs by boosting its priority level periodically. What should the time interval for boosting be? Sadly, a trial-and-error method seems necessary to find the optimal parameter as a long interval will lead to starvation of lengthy job but a short interval will lead to not enough running time for interactive jobs.

***Attempt #3: Better Accounting***

There is still the problem of malicious program gaming of the scheduler. The solution is simple too, just demote the process after it has used up its running time quota in the queue. We can see the differences in figure 3.12.

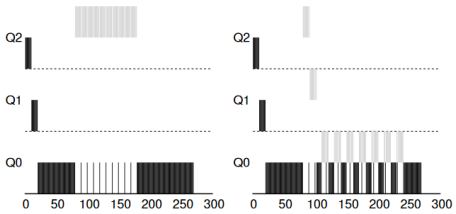


Figure 3.12: Without (Left) and With (Right) Gaming Tolerance (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p91)

***Lottery Scheduling***

Each task is given some tickets and the scheduler randomly selects a process holding that ticket. Thus, the chance for a process to be executed is based on the share of tickets it holds.

## Week 4: Thread

This week I learnt the concept of thread. Without the invention of thread, a process is equipped with the capability to have more than one point of execution. A thread is very similar to a process, except for one difference: threads share the same address space and thus can access the same resource. Thus, a thread has program counter (PC) as a process does and its own private set of registers, and a context switch must take place first before switching from a running thread to another. A process control block (PCB) is used to save the state for the process while thread control blocks (TCBs) are used to save the state for a thread. There are shared resources among threads too, which are user ID, group ID, process ID, heap memory, static data, program code, opened files, sockets, and lock.

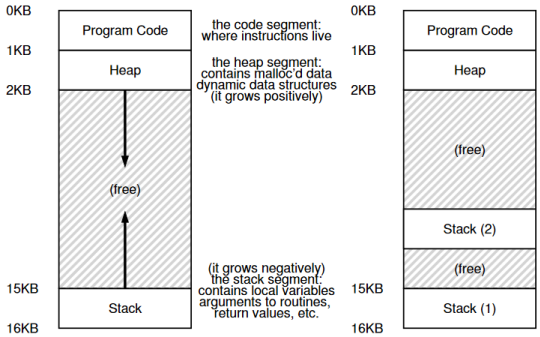


Figure 4.1: Single-Threaded And Multi-Threaded Address Spaces (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p304)

We can see the difference between a single-threaded process and a multi-threaded process from figure 4.1, the multi-threaded process has more than one stack, which corresponds to the number of threads. The purpose of one stack for one thread is to store the variables, parameters, return values and stuff that belong to each thread, and the stack is called thread-local storage. There is a higher risk for multi-threaded to face stack overflow.

### *Why Use Threads?*

The first motivation is to fully utilize the multi-processor computer by parallelism. A single-threaded process could only run on one processor even it is running on a multi-processor computer, but a task could be sped up if it is possible to split the task into multiple subtasks and run each subtask with a thread on each processor. The second motivation is to prevent stagnant program progress due to slow I/O. Using a web server as an example, an I/O has been issued for a new arrival HTTP request, then the webserver could not react to new HTTP request while waiting for the first I/O to complete if there is only one thread. With a multi-threaded process, the webserver could create a new thread for each HTTP request and still reacts to a new HTTP request in its main thread.

Of course, multi-process architecture could accomplish the objectives above too, but with a higher cost for a context switch. Moreover, threads share an address space, and thus multi-threaded programming paradigm could be easier for a program that has the attribute of intensive data-sharing.

### *An Example: Thread Creation*

Figure 4.2 illustrates a program that creates two threads. By calling pthread_create (), a thread is created and pthread_join () allows the main thread to wait for a particular thread to finish. The program in figure 4.2 will create two threads (A and B) and wait for these two to complete, then print “main: end”. The order of the execution of the main thread, thread A and thread B is non-

deterministic, and thus there could be “B” printed before “A” or “A” printed before “B”. The different possibilities for the order of the execution of the main thread, thread A and thread B are shown in figure 4.3, 4.4, and 4.5.

```
#include <stdio.h>
#include <assert.h>
#include <pthread.h>
#include "common.h"
#include "common_threads.h"

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int
main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: end\n");
    return 0;
}
```

Figure 4.2: Simple Thread Creation Code (t0.c) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p306)

| main | Thread 1 | Thread2 |
|---|---|---|
| starts running | | |
| prints “main: begin” | | |
| creates Thread 1 | | |
| creates Thread 2 | | |
| waits for T1 | | |
| | runs | |
| | prints “A” | |
| | returns | |
| waits for T2 | | |
| | | runs |
| | | prints “B” |
| | | returns |
| prints “main: end” | | |

Figure 4.3: Thread Trace (1) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p307)

| main | Thread 1 | Thread2 |
|---|---|---|
| starts running | | |
| prints “main: begin” | | |
| creates Thread 1 | | |
| | runs | |
| | prints “A” | |
| | returns | |
| creates Thread 2 | | |
| | | runs |
| | | prints “B” |
| | | returns |
| waits for T1 | | |
| *returns immediately; T1 is done* | | |
| waits for T2 | | |
| *returns immediately; T2 is done* | | |
| prints “main: end” | | |

Figure 4.4: Thread Trace (2) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p307)

| main | Thread 1 | Thread2 |
|---|---|---|
| starts running | | |
| prints “main: begin” | | |
| creates Thread 1 | | |
| creates Thread 2 | | |
| | | runs |
| | | prints “B” |
| | | returns |
| waits for T1 | | |
| | runs | |
| | prints “A” | |
| | returns | |
| waits for T2 | | |
| *returns immediately; T2 is done* | | |
| prints “main: end” | | |

Figure 4.5: Thread Trace (3) (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p308)

***Common Thread Strategies***

There are two common architectures for multi-threading. First is manager/worker in which the manager thread handles I/O and assigns a task to worker threads, the worker threads could be created dynamically or fetched from a thread pool. The second architecture is the pipeline architecture. There will be multiple threads working together where each thread responsible for different stage of the assembly line. Using a producer-consumer queue, the thread in the earlier stage of the assembly line could insert its work's output into the queue and the thread after it could collect the result in the queue as the input.

### *Pthreads: A Typical Thread API*

Pthreads stands for POSIX standard threads.

1. pthread_create (thread, attr, start_routine, arg)
   returns the created thread's ID in "thread", executes subroutine passes into the "start_routine" with argument specified by "arg".
2. pthread_exit (status)
   terminates the thread invokes this method and returns "status" to joining thread.
3. pthread_join (threaded, status)
   blocks the calling thread and wait till the thread with "threadid" ID terminates, and "status" will store the return status of the thread that invoked pthread_exit.
4. pthread_yield ()
   thread voluntarily gives up the CPU.

### *Threads in Java*

```
class MyThread extends Thread {
        public void run () {
                //......
        }
}
```

Figure 4.6: extends a Thread class.

```
Thread a = new MyThread ();
a.start ();
```

Figure 4.7: initializes a thread object and run it.

The run () method contains the instructions to run for each thread. The actual instructions executed depends on the implementation of the derived class. Get the thread into ready mode through .start() function and the thread will end when run () method finishes.

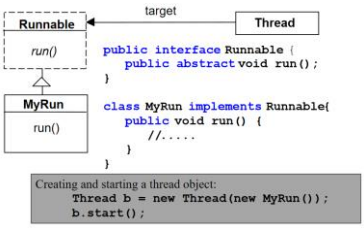


Figure 4.8: use a Runnable (Swinburne University of Technology 4(b), 2021, p5)

To prevent the problem of deadly diamond of dead, Java allows only single inheritance, and thus implement the run () method in a derived class that implements the Runnable interface (with the run () abstract function) provides a more flexible and maintainable solution by allowing the class to extend something else. The initialization of thread is different in that an object with a Runnable interface must be passed into the constructor of the thread.

### *Concurrent independent tasks – Example*

The simplest scenario to employ multi-threading is to run concurrent independent tasks without the needs to share resources between tasks, as shown in figure 4.9, 4.10, and 4.11.

```
public class A extends Thread{
    public void run()
    {
        while(true)
        {
            System.out.println("This is Thread A");
            try{
                Thread.sleep(1000);
            }catch(InterruptedException e){}
        }
    }
}
```

Figure 4.9: class A (Swinburne University of Technology 4(b), 2021, p16)

```
public class B extends Thread{
    public void run()
    {
        while(true)
        {
            System.out.println("This is Thread B");
            try{
                Thread.sleep(3000);
            }catch(InterruptedException e){}
        }
    }
}
```

Figure 4.10: class B (Swinburne University of Technology 4(b), 2021, p16)

```
public static void main() {

    Thread t1 = new A();
    Thread t2 = new B();

    t1.start();
    t2.start();
}
```

Figure 4.11: start thread t1 and thread t2 (Swinburne University of Technology 4(b), 2021, p16)

***Thread safety in Java libraries***

Some of the official libraries of Java are not thread-safe. For example, collections from java.util are not thread-safe, and thus multiple threads modifying the collection could result in an inconsistent state. Therefore, a developer should check the documentation even though quite a few do not state the issue clearly, but one can gain the knowledge based on one general wisdom: If the documentation does not specifically mention that it is thread-unsafe then it should be thread-safe.

***Shared variables in concurrency – Example***

Before discussing shared variables, we need to talk about the concept of atomicity. A common operation in Java that is not atomic is the increment of integer, i++. The operation is split into three stages, first load data from variable i, then increment data by i, finally store data to the variable i. Through figure 4.12, we can see that this operation is dangerous in multi-threading programming. Because the processor switch to thread B before thread A finishes execution, thread B could not perceive the operation done by thread A and thus leading to i equal to 1 instead of 2.

```
initial i = 0

Thread A        Thread B        i(A)    i(B)    i
Load i                          0       0       0
                Load i          0       0       0
Increment i                     1       0       0
                Increment i     1       1       0
                Store i         1       1       1
Store i                         1       1       1
```

Figure 4.12: wrong timing of context-switch leads to inconsistent state.

```
public class IncrementTest implements Runnable {

	static int classData = 0;
	int        instanceData = 0;

	@Override
	public void run () {
		int localData = 0;

		while (localData < 10000000) {
			localData++;
			instanceData++;
			classData++;
		}

		System.out.println("localData: " + localData +
				   "\tinstanceData: " + instanceData +
				   "\tclassData: " + classData);
	}

	public static void main(String[] args) {

		IncrementTest instance = new Incrementtest();

		Thread t1 = new Thread(instance);
		Thread t2 = new Thread(instance);

		t1.start();
		t2.start();

	}
}
```

Figure 4.13: threads that increment classData, instanceData, and localData variable (Swinburne University of Technology 4(b), 2021, p19)

Theoretically, after finished the execution of the program in figure 4.13, the value of localData should be 10 000 000, instanceData be 20 000 000, and classData be 20 000 000, however, only localData always get the correct value but values of instanceData and classData are always equal to or lower than 20 000 000 non-deterministically. This is because instanceData and classData are visible to both threads and lead to the issue mentioned with non-atomic integer increment. If we instead of creating a distinct IncrementTest class for each thread, then the value of instanceData will always be 10 000 000 as the instanceData is not shared by two threads anymore.

### *Thread Pools*

Certain applications have a large amount of short-lived independent tasks. There is overhead for creating and destroying thread, although it is not much for a single thread, the total cost for numerous threads is significant, and there is a limitation on the number of alive threads for different OS. Therefore, Java provides Thread Pool abstraction, which creates several threads and maintains them.

```
each thread is represented by $
each task is represented by +
task queue is represented by []

1.
[               ] <--- {$$$}
                  wait()

2.
[              +] ---> {$$$}
                  notify()

3.
[               ]      {$$ } $ executing +

4.
[               ]      {$$$}

5.
[+++++++++++++++]      {   } all $ are executing +
```

Figure 4.14: How thread pool works.

Figure 4.14 illustrates the working of a thread pool. At the start, threads will wait for tasks if the queue is empty. Then, a thread is notified when a new task entered the queue. The notified thread will then be removed from the thread pool and execute the task. After finishing the task, the thread will join the thread pool again. If all the threads are busy executing tasks, then new arrival tasks will fill up the queue and any additional task will be rejected when the queue is full. This architecture prevents the machine from being overwhelmed when there are too many tasks i.e. web server could spend its limited resources on serving current request instead of wasting resources on accepting

new request while there is no progress on accepted request, eventually leads to a non-functioning web server.

### *Thread Pools in Java – Example*

(Task class implements Runnable)

```
Task task1 = new Task();
Task task2 = new Task();
Task task3 = new Task();

System.out.println("Starting threads");

ExecutorService executor = Executors.newFixedThreadPool(3);

executor.execute(task1);
executor.execute(task2);
executor.execute(task3);
executor.shutdown(); // shutdown worker threads
executor.awaitTermination(1, TimeUnit.NANOSECONDS);
```

Figure 4.15: usage of Executor framework to implement thread pool (Swinburne University of Technology 4(b), 2021, p32)

At the start, three Task objects are created. Then, create a fixed thread pool by invoking Executors.newFixedThreadPool() method. Later, creates a new thread with execute() method and returns instantly from each invocation. The shutdown() method provides a way to shutdown gracefully by allowing no more new task to be submitted while finishing the submitted tasks. Finally, awaitTermination() is similar to join() in C in which waits for submitted to finish.

### *Pool Size*

How large should the thread pool be? This should depend on the characteristics of the application. As each thread consumes resources, starvation is possible when the thread pool is too large where the processor is busy switching thread but rarely spend time on threads with task to execute. On the other hand, a small pool could cause under-utilization of computing resources too, and lead to long waiting time for new arrival tasks even though there are plenty of computing resources available.

## Week 5 – Lock (I)

At this point, we should explore the key concurrency concept, namely critical section, race condition, indeterminate, mutual exclusion, and atomicity. A critical section is a code segment that accesses a shared resource. A race condition happens when multiple threads manipulate a shared resource at roughly the same time and the final state of the shared resource depends on the non-deterministic execution order of threads. An indeterminate program suffers from the problem of race conditions and thus the output of the program differs for each execution. To ensure the correctness of a program, the critical section must be protected that at one time only a single thread can enter. Thus, some form of mutual exclusion primitives could be used and solve the problem of race condition and produce a deterministic program output. Atomicity discusses the execution policy of many discrete instructions, either all execute successfully or failed together.

The mutual exclusion of the critical section could be implemented by using a lock mechanism. Phonebooth can be an analogy to lock, people must wait for the current person occupying the booth to leave first, then only the next person waiting in the line could enter the booth. A thread could only enter a critical section if it could acquire the lock for the critical section. All others thread will block while waiting to acquire the occupied lock.

```
1 lock_t mutex; // some globally-allocated lock 'mutex'
2 ...
3 lock(&mutex);
4 balance = balance + 1;
5 unlock(&mutex);
```

Figure 5.1: using lock in C (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p331)

For C, the lock provided is explicit lock, which means the programmer must perform the acquire and release of the lock manually. Moreover, the acquiring of releasing of the same lock is not confined to the same block of code and could be performed at different places.

```
public class SynchronizedCounter {

    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Figure 5.2: synchronized methods (Swinburne University of Technology 5, 2021, p52)

```
public void addName(String name) {

    // other work goes here

    synchronized(this) {

       lastName = name;
       nameCount++;

    }

    // other work goes here

}
```

Figure 5.3: synchronized statements (Swinburne University of Technology 5, 2021, p55)

Java, however, provides implicit and explicit lock. Implicit locking could be achieved through the "synchronized" keyword and mutual exclusion could be provided to the whole function or multiple statements. If the programmer synchronized a whole function, then the current object of the invocated function will be used implicitly as the lock, otherwise, the programmer can use an arbitrary object as the lock. Then, JVM will perform the releasing of lock automatically when the

function exits. One thing to note is that constructor could not be synchronized as there will be only one thread that has the access to the object at the creation of the object and it thus synchronizing it is meaningless.

```
public interface Lock {

 public void lock();

 public void lockInterruptibly();

 public void unlock();

 public Boolean tryLock();

 public Boolean tryLock(long time, TimeUnit
unit);

}
```

Figure 5.4: Lock interface in Java (Swinburne University of Technology 5, 2021, p62)

```
public class MyClass {
  private int shared_variable;
  private ReentrantLock lock;
  //ReentrantLock class implements Lock interface
  public void myFunction() {
   lock.lock();
   try {

       shared_variable++;

   } finally {
       lock.unlock();
   }
  }
}
```

Figure 5.5: using ReetrantLock in Java (Swinburne University of Technology 5, 2021, p45)

To use an explicit lock in Java, the programmer could use a lock object and function mostly like the one in C by exposing the lock and unlock function as the interface. Three concrete classes extend the Lock interface, which are ReentrantLock, ReentrantReadWriteLock.ReadLock, and ReentrantReadWriteLock.WriteLock. The concrete class ReentrantLock provides more functionalities than the “synchronized” method and thus could be a better alternative. The main problem of implicit lock is that it could not be interrupted, thus the thread will become blocked infinitely if deadlock happens, and the only solution is to restart the program. ReentrantLock provides tryLock and lockInterruptibly function as the solution. tryLock function allows the instant return of the function if acquiring of the lock failed or return after some time if the timeout parameter is provided. With lockInterruptibly function, the thread will be responsive to interruption.

## Week 6 – Lock (II)

Now the question arises on how to build a lock. What is the correct way to build an efficient lock and is hardware support necessary? Or sole OS support is enough? The efficacy of a lock is evaluated on three dimensions, which are the ability to provide mutual exclusion, fairness, and performance. Mutual exclusion is the most crucial criteria as the main purpose of a lock is to prevent multiple threads from entering a critical section. Fairness measures whether each thread contending for a lock get a fair chance to obtain it once it is available. In an extreme case, some threads will never obtain the lock if the fairness level is bad and thus these threads may block infinitely. Performance measures the time overheads for using the lock-in different cases. First is the case of no contention, the overhead incurred when a single thread acquires and releases the lock. Then there are multiple threads on a single CPU contending for the lock. Finally, the case when multiple CPU and multiple threads are contending for a lock.

### *A bad implementation of lock: Controlling Interrupts*

At the start, computer scientist used interrupts disabling to provide mutual exclusion on single-processor systems as shown in Figure 6.1.

```
1  void lock() {
2      DisableInterrupts();
3  }
4  void unlock() {
5      EnableInterrupts();
6  }
```

Figure 6.1: Implement lock with interrupts (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p334)

By disabling interrupts through special hardware instruction before entering a critical section, atomicity of the execution of the code in the critical section is promised as the execution of the code will be completed before switching to another thread, and thus there will be no interference and no race condition.

However, there are many disadvantages of using this approach even though it looks simple and elegant. First, a greedy program could monopolize the CPU by calling the lock () at the start of the program; a malicious program could even freeze the CPU by going into an endless loop after disabling interrupts. Restart the system is the only solution when the latter happened. Second, it fails on multiprocessors. One thread could disable interrupts for the CPU it is running on but other threads running on other CPU could still access the critical section. Third, important interrupts will be missed i.e. the interrupt to notify the waiting process that read request from the hard disk had finished. Finally, switching interrupts on and off incurs higher performance cost than normal instruction execution on modern CPUs. Therefore, disabling interrupts is mostly used by the OS itself as the privileged operations performed by the OS is always trusted.

### *A Failed Attempt: Just Using Loads/Stores*

```
1  typedef struct __lock_t { int flag; } lock_t;
2
3  void init(lock_t *mutex) {
4      // 0 -> lock is available, 1 -> held
5      mutex->flag = 0;
6  }
7
8  void lock(lock_t *mutex) {
9      while (mutex->flag == 1)  // TEST the flag
10         ; // spin-wait (do nothing)
11     mutex->flag = 1;          // now SET it!
12 }
13
14 void unlock(lock_t *mutex) {
15     mutex->flag = 0;
16 }
```

Figure 6.2: First Attempt: A Simple Flag (Arpaci-Dusseau and Arpaci-Dusseau, 2018, p335)

In the solution shown in figure 6.2, a simple variable(flag) indicates whether the lock is available or acquired. By calling lock (), the thread will check whether the flag is equal to 1 and sets the flag to 1 if it is not equal to 1 to indicate that the lock has been acquired. To release the lock, the thread calls unlock () to clear the flag, a signal that the lock is available again. Other threads trying to acquire the occupied lock will just spin-wait in the while loop till the thread holding the lock release the lock.

Unfortunately, the solution suffers from correctness and performance problems. The mutual exclusion is compromised when the execution of thread 1 and 2 interleave as shown in figure 6.3.

| Thread 1 | Thread 2 |
|---|---|
| call `lock ()` | |
| while (flag == 1) | |
| **interrupt: switch to Thread 2** | |
| | call `lock ()` |
| | while (flag == 1) |
| | flag = 1; |
| | **interrupt: switch to Thread 1** |
| flag = 1; // set flag to 1 (too!) | |

Figure 6.3: Trace: No Mutual Exclusion (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p335)

Assume flag is equal to 0 at the start, the CPU switch to thread 2 after thread 1 checked that flag is equal to 0. Thus, both threads will be able to enter the critical section and set the flag to 1. While waiting to acquire an occupied lock, the thread endlessly checks the value of the flag, thus the processor will waste time on the spin-waiting instead of running the thread holding the lock, results in a huge performance impact. The performance impact is lighter if the number of threads is less than or equal to the number of processors though.

### *Building Working Spin Locks with Test-And-Set*

Since simple method using loads and stores failed, hardware support had been invented for atomic loads and stores subroutine. In below figure 6.4, a test-and-set instruction is shown, and a simple spin lock implemented with it is shown in figure 6.5.

```
1  int TestAndSet(int *old_ptr, int new) {
2      int old = *old_ptr; // fetch old value at old_ptr
3      *old_ptr = new;     // store 'new' into old_ptr
4      return old;         // return the old value
5  }
```

Figure 6.4: test-and-set instruction (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p336)

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0: lock is available, 1: lock is held
7      lock->flag = 0;
8  }
9
10 void lock(lock_t *lock) {
11     while (TestAndSet(&lock->flag, 1) == 1)
12         ; // spin-wait (do nothing)
13 }
14
15 void unlock(lock_t *lock) {
16     lock->flag = 0;
17 }
```

Figure 6.5: A Simple Spin Lock Using Test-and-set (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p336)

A test-and-set instruction will return the old value of the lock while setting a new value into it. What this instruction differs from the previous solution is that these two operations will be performed atomically.

In the case when the lock is available, the first thread will receive 0 from TestAndSet (flag, 1) as return value and thus will skip the while loop and acquire the lock by setting the value of the flag to 1. To release the lock after quitting the critical section. If the lock is unavailable, the thread will receive 1 from TestAndSet(flag, 1) as a return value while setting the value of the flag to 1 simultaneously again. The thread will keep on spinning till the lock is available. The mutual exclusion is ensured as there will always be only one thread that manages to acquire the lock thanks to the atomic load and stores subroutine. However, this lock must work together with a preemptive scheduler so that a spinning thread will not monopoly the CPU forever. The fairness is bad too and suffers from the same performance issue as the simple load/stores approach described before.

***Compare-And-Swap***

```
1  int CompareAndSwap(int *ptr, int expected, int new) {
2      int original = *ptr;
3      if (original == expected)
4          *ptr = new;
5      return original;
6  }
```

Figure 6.6: Compare-and-swap (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p340)

There is another type of hardware primitive known as compare-and-swap instruction as shown in figure 6.6. The semantic and efficacy of compare-and-swap instruction is like test-and-set instruction except that it is more powerful and could be used for lock-free synchronization.

```
1  void lock(lock_t *lock) {
2      while (CompareAndSwap(&lock->flag, 0, 1) == 1)
3          ; // spin
4  }
```

Figure 6.7: A Simple Spin Lock using Compare-And-Swap (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p340)

In figure 6.7, if the flag's value is 0 then it will be set to 1 and break off from the while loop, otherwise, the thread will keep on spinning till the lock is available.

***Fetch-And-Add***

```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```

Figure 6.8: fetch-and-add instruction (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p342)

All the previous solutions do not promote fairness, but this solution will. Fetch-and-add instruction as shown in figure 6.8 is another hardware primitive that increases a value by one and returns the old value atomically. A ticket lock could be implemented using fetch-and-add instruction as shown in figure 6.9.

There will be two variables, namely ticket and turn. An analogy to turn variable is the current servicing number of the service desk and an analogy to ticket variable is the servicing number distributed to the customer who wishes to be serviced. Thus, a thread will do an atomic fetch-and-add on the ticket value when it wants to acquire the lock; the thread will then set its "turn" (myturn) variable to be the return value of the fetch-and-add function. When the lock->turn is equal to the thread's turn variable's value, then the thread will be allowed to enter the critical section. After finishing the critical section, the thread will then increment the lock->turn so that the next waiting thread could enter the critical section.

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

Figure 6.9: Ticket Locks (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p344)

We can see that this approach promotes fairness by making sure every thread will be executed at some point in the future. However, it is still a spinlock, thus suffers from the same performance issue as to any other variant of spinlocks.

### *A Simple Approach: Just Yield, Baby*

Instead of spinning endlessly, we could just use an operating system primitive yield () to give up the CPU as shown in figure 6.10.

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Figure 6.10: Lock With Test-and-set And Yield (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p344)

By calling yield (), the thread goes from running state to ready state, and thus free the CPU for other threads to run. The performance impact is huge if many threads are contending for a lock, the added-up cost of context switch is significant if the thread holding the lock is interrupted before releasing the lock and other threads get to run before the lock is released, which leads to stagnant progression on all threads. Thread starvation is possible too, a thread may be unfortunate enough that the lock is always held by other threads when the thread tries to acquire the lock.

### *Using Queues: Sleeping Instead Of Spinning*

The real problem with all the previous solutions is that the scheduler has not enough information to correctly determine the next thread to run. Thus, a thread chosen to run could be either waiting for the lock or immediately yield the CPU. Leading to waste of computing resources and the possibility of thread starvation.

Fortunately, we could use a queue and system call provided by OS to exert some control over the scheduling of thread execution. In figure 6.11, park () and unpark (threadID) are function provided by Solaris, in which the former will put the calling thread to sleep, and the latter will wake a thread with the ID specified through the threadID parameter.

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock
                                    // (for next thread!)
    m->guard = 0;
}
```

Figure 6.11: Lock With Queues, Test-and-set, Yield, And Wakeup (Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018, p346)

A guard lock is used to provide mutual exclusion to the code modify the flag variable and queue. Although spin-waiting is used when obtaining the guard lock, the time wasted on spinning is quite limited because there are fewer instructions for the lock and unlock subroutine than the user-defined critical section i.e. there are 10 threads, the thread holding the lock is interrupted before releasing the lock, then the CPU executes the other 9 thread for 9 seconds and back to the lock holding thread, but the thread is interrupted again without releasing the lock, another 9 seconds for 9 thread, in the end, it takes 18 seconds for the thread to release the lock. However, if the spinlock is used for the user-defined critical section with more instructions, then it could be interrupted more times and results in a longer time taken to release the lock.

For the lock function, the thread will first acquire the guard lock, set the flag to 1 to indicate possession if the flag lock is available, otherwise add the current thread into the queue and put the current thread into sleep. The freeing of the guard lock must be done before putting the thread into sleep otherwise other threads will never acquire the guard lock.

For the unlock function, after acquiring the guard lock, the thread will free the lock if there is no more thread in the queue, or wake a thread waiting in the queue. The flag lock is not free after waking up a thread is intentional since the waking thread will instantly execute the instruction after the park () and thus will not hold the guard lock and could not modify the flag lock.

There is a race condition in the lock function, just before the call to park (). If the thread is interrupted before putting itself to sleep, then the thread holding the lock releases the lock and unpark the interrupted thread, unfortunately, the interrupted thread will execute the park function and put itself into sleep forever. There are several solutions, Solaris solves it by adding a system call: setpark (), allowing the thread to instantly return from park function instead of sleeping if the above race condition happens.

## Reference

Arpaci-Dusseau, R. and Arpaci-Dusseau, A., 2018. *Operating systems*. 1st ed. Arpaci-Dusseau Books.

Lecture 2, *Concurrent Programming*, Swinburne University of Technology, 2021.

Lecture 4(b), *Concurrent Programming*, Swinburne University of Technology, 2021.

Lecture 5, *Concurrent Programming*, Swinburne University of Technology, 2021.