

COS40003: Lab Report 1

Vi Luan Dang - 103802759

Faculty of Science, Engineering and Technology

Swinburne University of Technology

Ho Chi Minh, A35 Bach Dang, Tan Binh District

Week 1:

Overview: The first week provides information regarding common computing paradigms such as concurrent computing, parallel computing, distributed computing, cluster computing, grid computing, cloud computing, for/edge computing. It is crucial to understand and differentiate between these paradigms to effectively understand how programs work.

Concurrent Computing and Parallel Computing

It is crucial to draw a clear line between concurrent computing and parallel computing, as it is commonly thought to be the same thing. To be able to do that, we must first understand Concurrency and Parallelism.

In the context of computer science, **Concurrency** refers to running multiple threads or processes at the same time, even on single CPU core, by interleaving their executions, although it may appear to the user that the process is running at the same time, from the perspective of the CPU, the instructions from these processes are run one at a time. **Parallelism**, on the other hand, refers to the running of multiple threads/processes in parallel over different CPU cores.

With that in mind, **Concurrent Computing** is the development, and subsequently execution, of tasks (or instructions) from one or many processes one at a time. **Parallel Computing**, on the other hand, divides tasks into smaller sub tasks and having those processed independently and simultaneously in provided memory allocation using provided CPU cores.

Distributed Computing

Distributed computing refers to the practice of utilizing multiple computers or nodes to work together on a common task or problem. In distributed computing, the computers are connected through a network and collaborate to achieve a common goal. Each of the nodes comprising of the distributed computing system has its own processing power, memory, and storage. They then communicate and coordinate with each other by sharing data and messages over the network.

Cluster Computing and Grid Computing

Cluster Computing is somewhat similar to distributed computing in a sense that there are multiple physical machines involved, however, cluster computing connects these nodes, often built from many affordable and low powered device with unified software and hardware, locally to allow for better communication. **Grid Computing** is similar to that of cluster computing, but it is not built using unified software and hardware and often sparse a large geographic location.

Cloud Computing

Cloud computing refers to the delivery of computing resources, such as servers, storage, databases, networking, software, and analytics, over the internet. In cloud computing, service providers maintain and manage the underlying infrastructure, including hardware, software, and networking components. Users can then access these resources to perform their wanted tasks.

There are certain requirements for one service to be considered Cloud Computing:

- + **On-Demand Self Service:** Provision and termination using UI or CLI without actual human interaction.
- + **Broad Network Access:** Access services over any network, on any devices, using standard protocols and methods.
- + **Resource Pooling:** Economies of scale, cheaper service.
- + **Rapid Elasticity:** Scale UP and DOWN automatically in response to system load.
- + **Measured Service:** Usage is measured. Users pay for what they consume.

There are three main types of service that Cloud Computing provider provides:

- + **Infrastructure As A Service (IAAS):** Abstracting the complexity of managing the underlying infrastructure.
- + **Platform As A Service (PAAS):** Abstracting the complexity of managing the platform upon which program and services are hosted.
- + **Software As A Service (SAAS):** Abstracting the complexity of developing and managing the software, providing the user ready-to-use application.

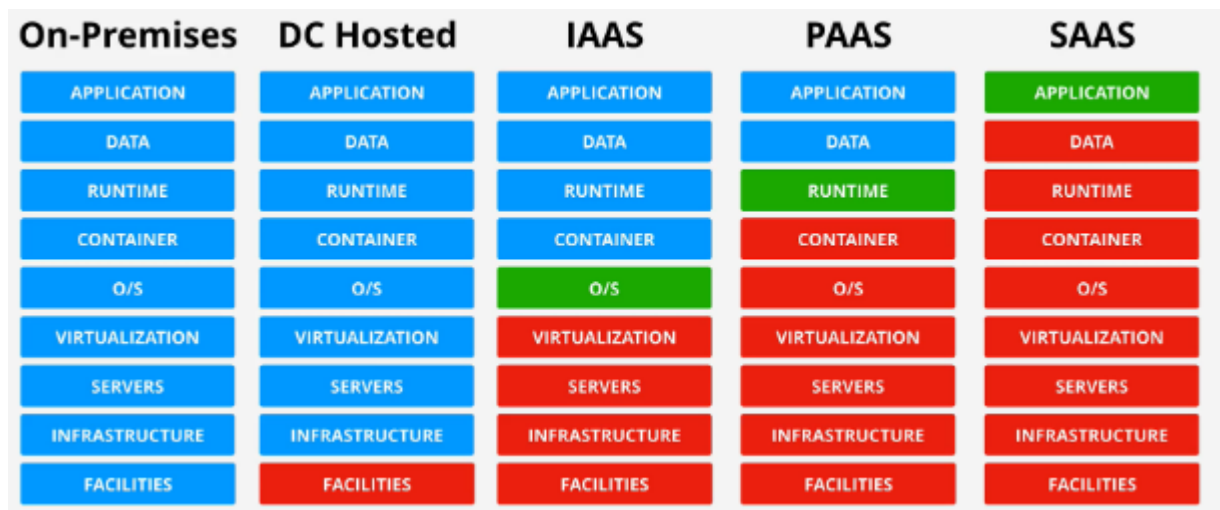


Figure 1: Infrastructure stacks

Fog/Edge Computing

Edge computing, also known as fog computing, shares similarities with cloud computing, but it offers advantages for systems that involve sensors or Internet of Things (IoT) devices that need to communicate with servers quickly and with minimal delay. In this architecture, fog nodes are positioned between the sensor devices and a central server. These fog nodes are located in close proximity to the devices, enabling efficient and low-latency communication.

Week 2

Overview: This week's knowledge revolves around the **process**, which is the running program. This knowledge is essential to understand how the Operating System works and how programs are run in the operating system. Before we can dive deeper into the process, we must understand a few things about what happens when we run a program.

What happens when we run a program?

Imagine we have a simple C program, what will happen when we run that program?

1. A compiler translates high level programs into an executable.
2. The OS manages program memory, loading program executable (code, data) from disk to memory.
3. The OS manages CPU, initializes program counter (PC) and other registers to begin execution.
4. The CPU fetches instruction pointed at by PC from memory, loads data required by the instruction into register, decodes and executes the instruction one by one until the program completes.

As we can see, the OS provides the CPU with the necessary resources for it to execute instructions, and subsequently, operates the program. Therefore, there must be a way for the OS to manage the CPU, in order to do all of that, the solution to that is the **process abstraction**.

What is the process abstraction?

When we run an exe file, the OS creates a process (which is an abstraction of a running program), to be considered a process it must contain the following attributes:

- + A Process Unique Identifier (PID): to identify itself from other processes.
- + A Memory Image: including static data (code and provided data), as well as dynamic data (stack and heap)
- + A CPU context: refer to the state of the executing process from the perspective of the CPU, this will include various registers such as the Program Counter, the Stack Pointer, and the Current Operands
- + A File Descriptors: such as Standard Output, Standard Input, and Standard Error.

These attributes collectively define the process abstraction and allow the operating system to manage and control the execution of multiple processes concurrently. However, as there might be many processes running concurrently in a system, there are various states that a process may have in order to effectively timesharing the CPU.

States of the process

There are three noteworthy states of a process: **Running**, **Ready**, and **Block**

- + **Running**: In the running state, a process is running on a processor.
- + **Ready**: In the ready state, a process is ready to run but the OS is currently descheduled its execution for some reason and process in this state is waiting for the OS to schedule its execution.
- + **Block**: In the blocked state, a process is currently not running, as it may have initialized some request the required other processes. One simple example for this is the process may have initiated an I/O request to a disk and is waiting for that request to complete.

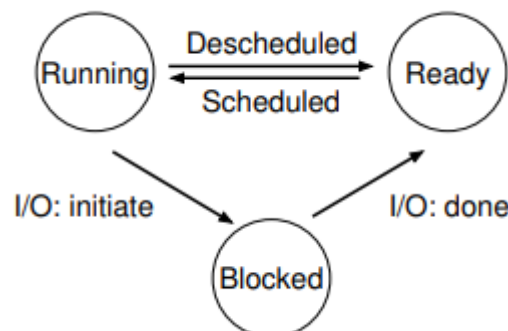


Figure 2: Process: State Transitions

Figure 3 below visualizes how different states of a process switch to facilitate Concurrent Computing. There is only one process **running** at a time, other processes are either **blocked** or **ready**, waiting for their turn to use the CPU. When *Process₀* initiates I/O, its state will change from running to blocked. *Process₁*'s state, initially ready, will change to running as the CPU is released from *Process₀*'s usage. The flow of execution will continue until both processes are done.

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 3: Tracing Process State

We will dive a bit further into the creation, execution, and termination of processes from a technical perspective by understanding Process Application Programming Interface (API).

Process Application Programming Interface

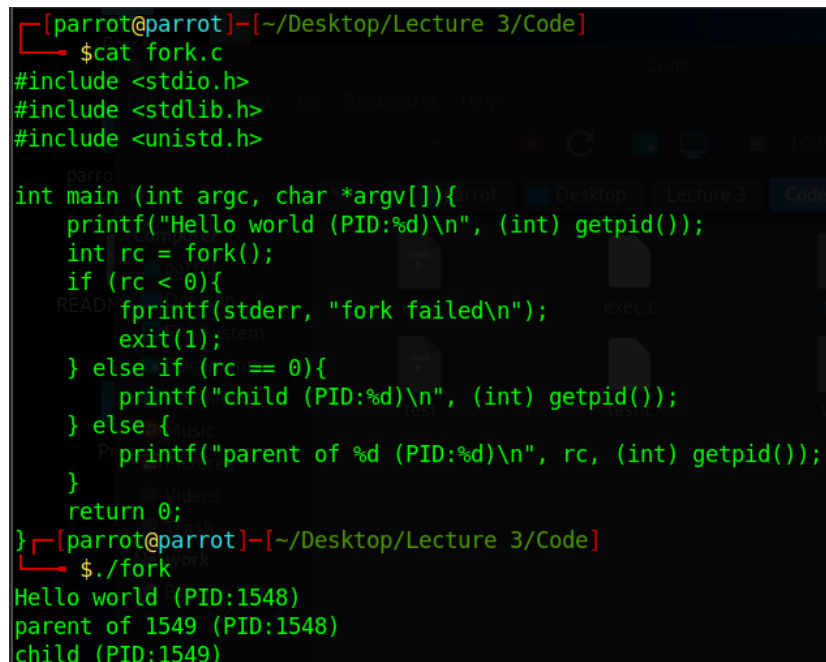
API stands for Application Programming Interface, which is just a fancy way of addressing **a set of functions available to write user's programs**, so let's ask ourselves a question. What functions or API are called when we double-click on a program? Or type a command into the shell?

They may be Create(), Execute(), Wait(), and Destroy(), those are the set of functions, the API that could be involved in our execution of a program. These APIs, in some form, are available on any modern operating system. **The API provided by the OS is a set of system calls** which invoke OS code that runs at higher privilege level of the CPU. Some of the most common system calls related to process execution are:

- + **Fork()**: creates a new child process, all processes are created by forking from a parent and the *init* process is the ancestor of all processes.
- + **Exec()**: makes a process execute a given executable.
- + **Exit()**: terminates a process.
- + **Wait()**: causes a parent to block until child terminates.

What happened during a fork()?

The `fork()` system call is used to create a new process, a new process is created by making a copy of the parent's memory image, the new process is added to the OS process list and schedule for execution. The parent and child start execution just after fork with different values. The below code snippet illustrates how a `fork()` operates.

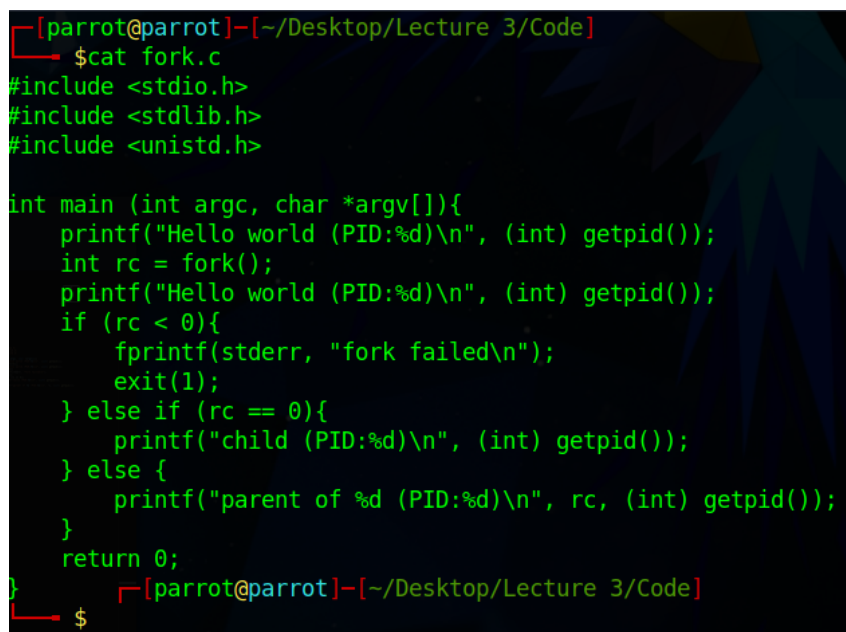


```
[parrot@parrot]~/Desktop/Lecture 3/Code
$cat fork.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char *argv[]){
    printf("Hello world (PID:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0){
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0){
        printf("child (PID:%d)\n", (int) getpid());
    } else {
        printf("parent of %d (PID:%d)\n", rc, (int) getpid());
    }
    return 0;
}
[parrot@parrot]~/Desktop/Lecture 3/Code
$./fork
Hello world (PID:1548)
parent of 1549 (PID:1548)
child (PID:1549)
```

Figure 4: `fork()` system call.

It is noteworthy that the newly created process (called the **child** henceforward) does not start running at `main()` like the current process (called the **parent** henceforward), as if it was, there would have been two “Hello world” printed out; rather, it starts when we call `fork()`. Therefore, we can note that the parent and the child process execute and modify the memory data independently after the process. To experiment this point, we shall add another `printf()` line after we call `fork()` as follows:



```
[parrot@parrot]~/Desktop/Lecture 3/Code
$cat fork.c
#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();
    printf("Hello world (PID:%d)\n", (int) getpid());
    if (rc < 0){
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0){
        printf("child (PID:%d)\n", (int) getpid());
    } else {
        printf("parent of %d (PID:%d)\n", rc, (int) getpid());
    }
    return 0;
}
[parrot@parrot]~/Desktop/Lecture 3/Code
$
```

Figure 5: Modified `fork()` system call.

So, according to our belief in figure 4, the child process will execute only after the line of code that it is called, meaning that it will not contain the `printf()` function. Now, with the modified code in figure 5, we will examine our understanding by adding another `printf()` after `fork()`, we expect to have three “Hello world” printed out with its according PID, two of which will be from the parent process, and one of which will be from the child process.

```

child (PID:1954)
[parrot@parrot]--[~/Desktop/Lecture 3/Code]
$ ./fork2
Hello world (PID:1953)
Hello world (PID:1953)
parent of 1954 (PID:1953)
Hello world (PID:1954)
child (PID:1954)

```

Figure 6: Modified result

The result of the modified `fork()` system call usage confirms our understanding, the parent and the child process do execute and modify the memory data independently after the process, and the child execute the instructions after it had called `fork()`. But at the time that `fork()` is called, there would be two different processes waiting to be executed, and either of them could be running. What if we want to delay the parent so that the child will always execute first?

What is the relationship between the child and the parent process?

When the child process is created, there are now two active processes in the system: the parent and the child. Assuming that we run the program on a single CPU system, then either the child or the parent might run at that point. As shown in the below output:

```

prompt> ./p1
hello (pid:29146)
child (pid:29147)
parent of 29147 (pid:29146)
prompt>

```

Figure 5: Different output of `fork()`

The parent process may need to wait for the child process to complete before continuing the execution, and this can be done via the `wait()` system call as `wait()` will block the parent until its child terminates, ensures synchronization for our program.

```

[x]--[parrot@parrot]--[~/Desktop/Lecture 3/Code]
$ cat wait.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <wait.h>

int main (int argc, char *argv[]){
    printf("Hello world (PID:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0){
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0){
        printf("child (PID:%d)\n", (int) getpid());
    } else {
        int rc_wait = wait(NULL);
        printf("parent of %d (rc_wait:%d) (PID:%d)\n", rc, rc_wait, (int) getpid());
    }
    return 0;
}
[parrot@parrot]--[~/Desktop/Lecture 3/Code]
$ ./wait
Hello world (PID:1956)
child (PID:1957)
parent of 1957 (rc_wait:1957) (PID:1956)

```

Figure 6: `wait()` system call.

With this code, we know that the child will always terminate first, and then the parent will terminate, and finally the program will end. If the OS decides to run the parent process first, the `wait()` function will block its execution, and the block will not proceed until the child process has been executed and exited. This will ensure the synchronization that we need for this program.

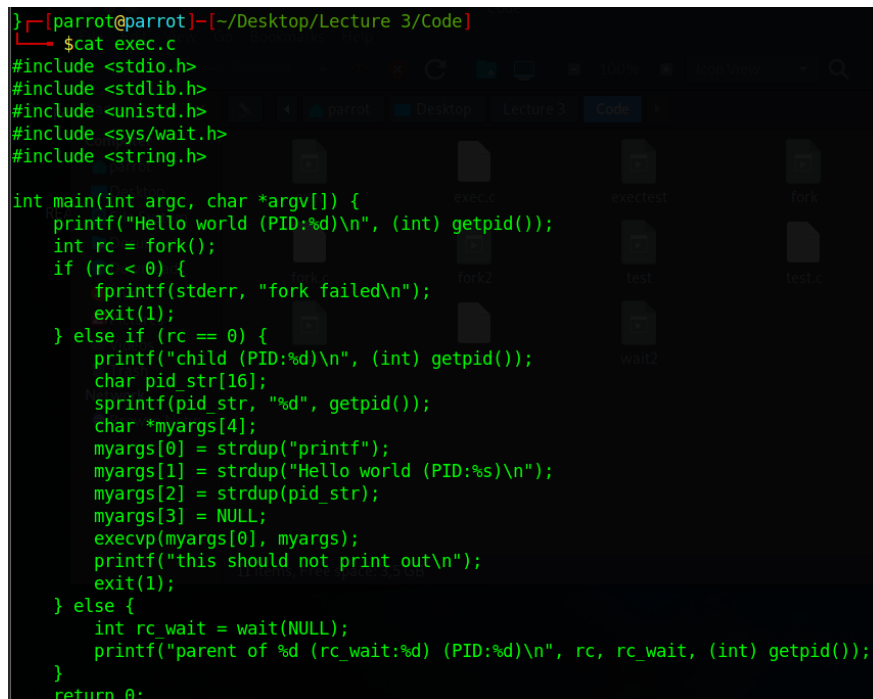
What happened during an exec()?

From all of the above examples, our execution of the child process is simply the copy of its parent, which is not very useful as we might want to run a different program, this can be done using the *exec()* system call.

Exec(), expects two parameters:

- + The first parameter is a pointer to a null-terminated string that specifies the path or filename of the program we want to execute.
- + The second parameter is an array of pointers to null-terminated strings. This array represents the program's argument list. The last element of the array must also be a null pointer, which indicates the end of the argument list.

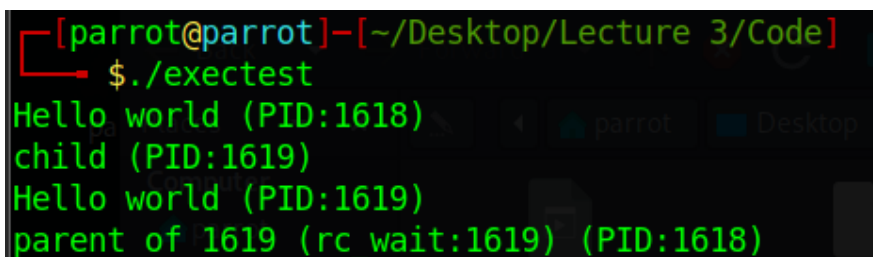
We shall examine the use of *exec()* in the following example:



```
[parrot@parrot]~/Desktop/Lecture 3/Code
$ cat exec.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>

int main(int argc, char *argv[]) {
    printf("Hello world (PID:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        printf("child (PID:%d)\n", (int) getpid());
        char pid_str[16];
        sprintf(pid_str, "%d", getpid());
        char *myargs[4];
        myargs[0] = strdup("printf");
        myargs[1] = strdup("Hello world (PID:%s)\n");
        myargs[2] = strdup(pid_str);
        myargs[3] = NULL;
        execvp(myargs[0], myargs);
        printf("this should not print out\n");
        exit(1);
    } else {
        int rc_wait = wait(NULL);
        printf("parent of %d (rc_wait:%d) (PID:%d)\n", rc, rc_wait, (int) getpid());
    }
    return 0;
}
```

Figure 7: Exec() system call



```
[parrot@parrot]~/Desktop/Lecture 3/Code
$ ./exectest
Hello world (PID:1618)
child (PID:1619)
Hello world (PID:1619)
parent of 1619 (rc_wait:1619) (PID:1618)
```

Figure 8: Exec() system call output

In the above example, we use another version of *exec()* called *execvp()*, stands for execute vector path, so that we don't need to provide the full path to the program file like *exec()*. As we can see, a process running *exec()* to load another executable to its memory image, effectively allowing the child process to run a different program from its parent.

How do we know that the child process using *execvp()* runs a different program from its parent? There is another *printf()* after *execvp()* that proves this, as if the child uses the same code as its parent, the *printf()* should have been written out. Therefore, we can confirm that when *execvp()* is called, it replaces the current process image with the new program provided in its argument. Subsequently, the child process starts executing the new program from its main entry point, any code that follows the *execvp()* call in the original program is not executed in the child process as the entire process image has been replaced. We will end our week by looking at the concept of Process Execution Mechanism and Context Switch.

Process Execution Mechanism and Context Switch

Although this course provided information regarding how program works, I want to take an extra step in understanding how program works from the perspective of the Operating System, and how the Operating System works as well, therefore, at this point of the course, I wonder how process is executed and the problems of doing that.

As discuss above, when the OS wishes to start a program, it creates a process entry for it in a process list, allocates some memory for, loads the program code into the memory from the disk, locates the process entry point, jumps to it, and start executing the code. This simple approach, however, presents two problems:

- + **Problem 1:** If we just run a program like that, how can the OS make sure the program doesn't do anything that we don't want it to do (i.e., accessing the hardware without permission), while still running it efficiently?
- + **Problem 2:** When we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing required to virtualize the CPU.

Problem 1: Restricted Operations

What if a process wishes to perform some kind of restricted operation? We cannot give the process complete privilege, nor can we disregard its request. The solution for this problem is two modes: **user mode** and **kernel mode**. **User mode** has code that is restricted in what it can do, **kernel mode** runs in higher privilege that has access to mostly all the function of the system.

Whenever the **user mode** wishes to have privileged access, it will make a system call to the **kernel mode**; the system will then switch to **kernel mode** to perform the requested action. Upon completion, the system will switch back to **user mode**.

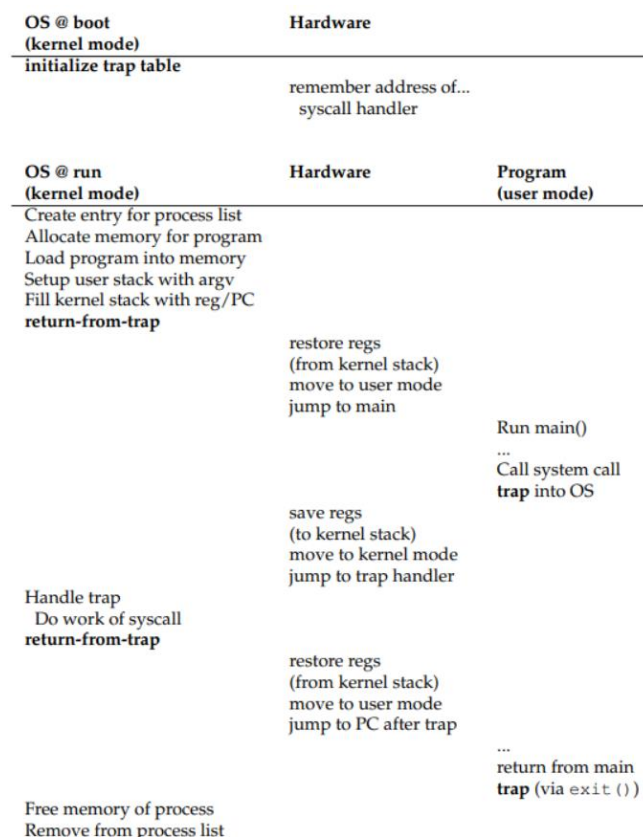


Figure 9: Limited Direct Execution Protocol

Therefore, when in **user mode** and wishes to perform privilege actions, the system will make **system call** to **kernel mode**, the system will execute a special **trap instruction** that will **jumps into kernel** (jump into the memory allocated for kernel) and **raises the privilege**. Then the system can do what it wishes if it is allowed with the appropriate permissions. Upon completion, the **kernel** will execute **return-from-trap instruction** to return back to the **user mode**. However, the state of each mode needs to be reserved so that it can switch back and forth, Context Switching facilitates this.

Context Switching

Context Switching involves the process of saving the state of the currently running process, loading the state of another process, and transferring control to that process. During this, the current process's state, including register values, program counter, stack pointer, processor flag, and other information, is saved; and the state of the next process to be executed is loaded. This allows the operating system to switch between processes and provide the illusion of concurrent execution.

In the case of switching between user mode and kernel mode, context switching is performed to reserve the state of the current mode before switching to the other mode. This also occurs when the OS decides to switch from the current process to a different one. In that case, the OS will save the state of the currently executing process and restore the state for the soon-to-be-executing process. We understand that the OS switches between user mode and kernel mode when a privileged call is performed or when an interrupt happens, but how does the OS switch between processes?

Problem 2: Switching between processes

If a process is running on the CPU, this means that the CPU (which is the component that executes commands) is occupied by the said process, then how can the OS (which also needs the CPU) retain control of the said CPU to then switch between processes?

A Cooperative Approach

It turns out that programs relied heavily on **privilege actions** - which needs to be **trap** to the OS to perform. So, one way or another, the program will **traps** back to the OS, effectively giving the CPU to the OS and the OS can determine if it needs to switch between processes.

Thus, in a cooperative scheduling system, the OS regains control of the CPU by waiting for a system call or an illegal operation of some kind to take place which will invoke an interruption. This, however, is a passive approach and is less than ideal if a process ends up in an infinite loop and never makes a system call.

Non-Cooperative Approach

A timer device can be programmed to raise an interrupt every so many milliseconds; when the interrupt is raised, the current running process is halted, and a pre-configured interrupt handler in the OS runs. At this point, the OS has regained control of the CPU, and thus can do what it pleases: stop the current process and start a different one.

However, whether the current process is allowed to continue running or should it be stopped for another process to take place relies on another aspect that we will discover soon.

Week 3

Overview: This week's knowledge revolves around the **scheduling of processes**, including but not limited to First come, first serve (FIFO), shortest job first (SJF), preemptive shortest job first/shortest remaining time/shortest time – to – completion first (PSJF/STCF), Round Robin.

Why do we need scheduling?

We now understand that time sharing the CPU is crucial for concurrent programming, and switching between states, processes, modes facilitate a smooth operation of our processes. However, when the OS decides to switch between one process to another, how and what criteria enable it to make such a decision? There should be a certain set of rules or policies for the OS to assess before switching between processes.

While reading and listening to the lecture for this week about scheduling policies, I was a bit baffled of what were the criteria that enabled the policies to even exist at all. The provided materials, luckily, give me the necessary information to dive deeper into this topic. Therefore, before we get to the policies themselves, let's lay the groundwork for two concepts, **workload** – so as not to wonder what kind of tasks we are referring to in the policies, and **metrics** – so as to have a unified measurement for the policies.

Workload and Metrics

Workload, in the context of understanding scheduling policies, refers to the simplified tasks of the processes running in the system. This simplified concept of processes' task is made by assuming the following:

1. Each process runs for the same amount of time. (henceforward, **assumption 1**)
2. All processes arrive at the same time. (henceforward, **assumption 2**)
3. Once started, each process runs to completion. (henceforward, **assumption 3**)
4. All processes only use the CPU, meaning they do not perform any I/O. (henceforward, **assumption 4**)
5. The run-time of each process is known. (henceforward, **assumption 5**)

With that in mind, we shall not wonder why some tasks, although in reality takes much more time than others, are the same as their peers. We will also need a unified measurement for the policies, **a scheduling metric**. We will use one single metric: **turnaround time**, which is calculated by using the time the job completes minus the time at which it arrives and as we assume all job arrives at the same time, the turnaround time will be equal to the completion time, as illustrated in the formula below.

$$T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$$

With all the necessary information in hand, we shall discuss the first, and considered the simplest, policies, First In, First Out (FIFO)

First In, First Out (FIFO)

The FIFO policies executes tasks in the order that they arrive, meaning that the task that comes first, will be executed first and continually until there is no task left. Using the assumption made above, we can see that although this policy is simple, it is fairly effective. Let's observe its effectiveness using our metric.

Assuming three tasks, each of which arrives roughly at the same time ($T_{\text{arrival}} = 0$), and they all take the same amount of time to complete, let's say 10 seconds. We will have the average turnaround time for these jobs as:

$$\frac{10 + 20 + 30}{3} = 20 \text{ (seconds)}$$

The figure below illustrates our example:

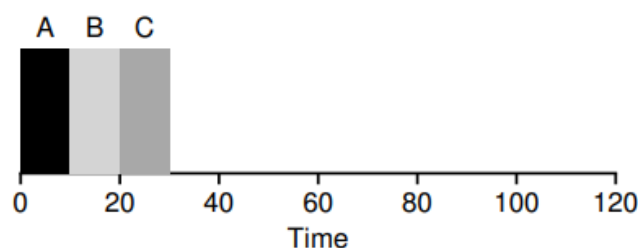


Figure 10: FIFO Example

However, if we loosen up **assumption 1** for this example, meaning that not all tasks have the same execution time, this policy will present a problem. Let's change the execution time for task A to 100 seconds, we will have our average turnaround time as:

$$\frac{100 + 110 + 120}{3} = 110 \text{ (seconds)}$$

As illustrated in the figure below:

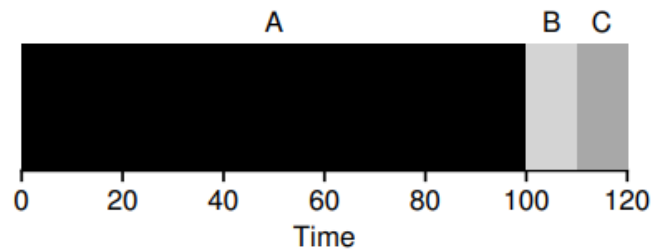


Figure 11: FIFO's problem

This problem is generally referred to as the **convoy effect**, when low-resource consumed tasks are queued behind a heavy-resource consumed task. So, is there any other policy that may fix this problem?

Shortest Job First (SJF)

The policy that can solve the above problem is called Shortest Job First policy, where we will run the shortest job first, hence the name, then turn to the next shortest and so on. Let's take the above example, applied SJF, and see the effect that it can bring to our average turnaround time.

$$\frac{10 + 20 + 120}{3} = 50 \text{ (seconds)}$$

As illustrated in the figure below:

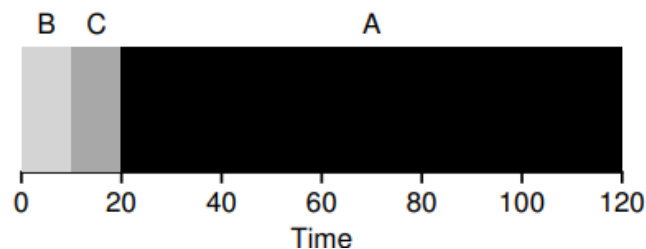


Figure 12: SJF example

Given the assumption that all tasks arrive at roughly the same time, we can see that SJF is fairly *'realistic'* as well as optimal compared to FIFO. But if we loosen up **assumption 2**, meaning that not all tasks arrive at the same time, and it can arrive at the execution time of another tasks, the scheduling of shortest tasks is now no longer feasible.

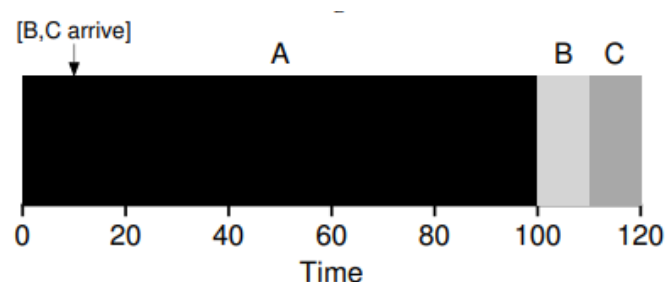


Figure 13: SJF example

As tasks B and C now have to wait for the completion of task A before they can be scheduled for execution, our average turnaround time is also negatively affected:

$$\frac{100 + (110 - 10) + (120 - 10)}{3} = 103.33 \text{ (seconds)}$$

Shortest Time-to-Completion First (STCF)

In order to address this concern, we will loosen up assumption 3, meaning that task currently running is not forced to run to completion. This effectively allows us to stop one task when another task, which has a shorter time of completion, arrives.

As discussed in Week 2 about the Non-Cooperative Approach and Context Switching, timer interrupts allow the OS to take back the CPU's usage from currently-process to assess whether there is another process that takes less time to complete, and if there is, it can switch to that process before completing the currently-running process. In other words, it can **preempt** job A in our previous example to run the lesser-time consuming tasks B and C.

SJF suffers from convoy effect as it lacks the preemptive aspect of our currently discussed policy. With preemption now added to SJF, our policy is called **Shortest Time-to-Completion First (STCF)** or **Preemptive Shortest Job First (PSJF)**. Let's observe the average turnaround time to see the improvement made by this policy:

$$\frac{(120 - 0) + (20 - 10) + (30 - 10)}{3} = 50 \text{ (seconds)}$$

As illustrates in the figure below:

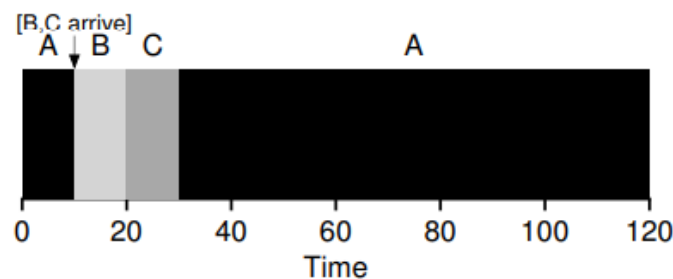


Figure 14: STCF Example

Another metric: Response Time

STCF seems to be perfect solution for our scheduling question, but it is only great in the provided environment, namely we know the task lengths, the task only used the CPU (without any I/O), and our metric was turnaround time. How would the users feel from the perspective of task A, when they have to wait for tasks B and C to complete, without any interactive response from the system? A problem arises as time-shared systems require another metric: the **Response Time**

We can define response time as the time from when the job arrives in a system to the first time it is scheduled, illustrated by the following formula:

$$T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$$

Therefore, if we use the example in figure 14, our average response time 3.33 as:

$$\frac{(0 - 0) + (10 - 10) + (10 - 20)}{3} = 3.33 \text{ (seconds)}$$

This proves that STCF, and related policies are not particularly great for response time, which leads to adverse effects on interactivity and timed-shared aspect of our system. With that in mind, a policy that is response-time-sensitive is required.

Round Robin

A new scheduling policy has been introduced to solve this problem, called **Round-Robin (RR)**. Its idea is fairly simple, instead of running tasks to completion, RR runs a task for a period of time and then switches to the next job in the queue repeatedly until all the tasks are finished. It is noteworthy that the period for each task must be a multiple of the timer-interrupt period so that the OS can regain control of the CPU to switch to another task.

Let's take a look at the following example to understand RR, if we have three tasks, each wish to run for 5 seconds. An SJF policy will run each of them to completion before running another, which is indeed bad for time response. We will have our average response time for RR as: $\frac{(0-0)+(1-0)+(2-0)}{3} = 1 \text{ (second)}$, and

$$\text{SJF as: } \frac{(0-0)+(5-0)+(10-0)}{3} = 5 \text{ (second)}$$

As illustrated in the following figures:

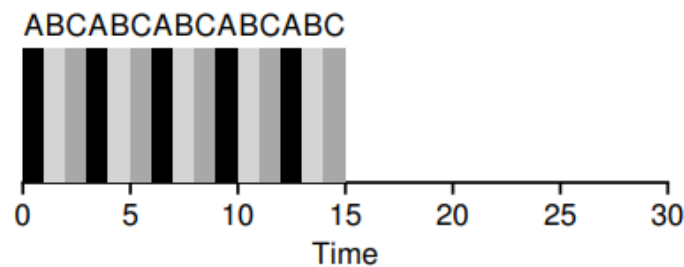


Figure 15: Round Robin Policy

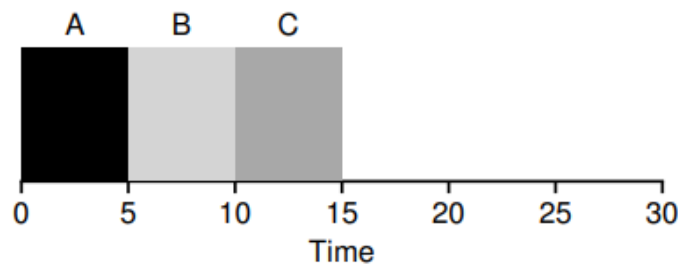


Figure 16: SJF Policy

Incorporating I/O

Our **assumption 4** restricts the use of I/O in all of our tasks, which is rather unrealistic. So, we will loosen up this assumption to see how I/O is incorporated into our policy.

As shown in *figure 3*, when a process initiates I/O, its state changes from running to blocked waiting for I/O to complete. During that time, the CPU is free and will probably be used for another task. When the I/O completes, however, an interrupt is raised and the currently-running job will return the CPU to the OS so that it can move the process that initiates I/O back to ready state. This answers our question of how I/O is incorporated into our policy, but how can the OS maximize the CPU usage during I/O execution?

Let's assume that we have two processes, A and B, both of which need 50 ms of CPU time. A, however, requires an I/O request after 10 ms of CPU usage, whereas B simply uses the CPU for 50 ms and performs no I/O. If we use STCF, the 50 ms CPU usage of A will be divided into 5 10-ms sub-process, and our scheduler will treat each sub process as an independent process. During the I/O execution of A, the scheduler will swap B's execution in, and after 10 ms, it will swap A's execution back in. Doing this allows for overlapping usage of CPU, which maximizes the CPU usage. This can be better illustrated with the two figures below:

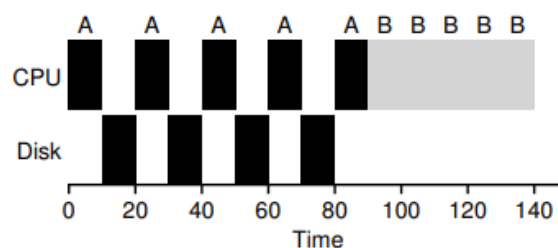


Figure 17: CPU usage with no overlapping

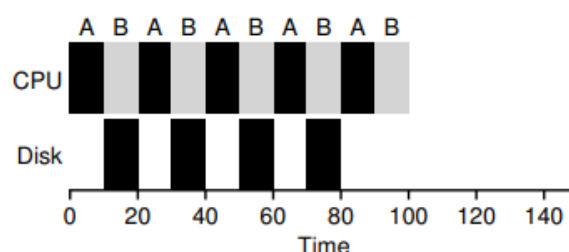


Figure 18: CPU usage with overlapping

We have come a long way in understanding how scheduling policy works, with 4 assumptions being loosened up, there is still one remaining, **assumption 5** which states that the run-time of each process is known.

The Multi-Level Feedback Queue

MLFQ exists to solve two problems. First, it tries to **optimize turnaround time**, which is done by running the shortest job first; however, as we stated above, we loosened up **assumption 5** meaning that the OS does not know how long a job will run for, subsequently unable to run the shortest job first. Second, MLFQ tries to make a system that feels responsive to interactive users, and thus **minimizes response time**; however, policies like Round Robin is great at reducing response time but are terrible for turnaround time.

MLFQ Basic Rules

In order to build such a policy, MLFQ utilizes a number of **distinct queues**, each with a different **priority level**. The priority levels are used to determine which job will be run at a given moment, a job with higher priority (meaning that it is on a higher queue) will be chosen to run over a job with lower priority (meaning that it is on a lower queue). Subsequently, as there may be more than one job on a same queue, therefore, having the same priority, the first two rules describe which job is run when the priority level is the same, and when the priority level differs:

Rule 1: If $Priority(A) > Priority(B)$, A runs (B does not).

Rule 2: If $Priority(A) = Priority(B)$, A & B run in Round Robin Policy.

Our CPU may have two kinds of processes, a short-running one and a long-running one. The short-running job is kept at a higher queue (with a higher priority) effectively allowing it precedence of CPU usage over the long-running job, which is kept at a lower queue (with lower priority). The illustration below visualizes our current understanding of MLFQ:

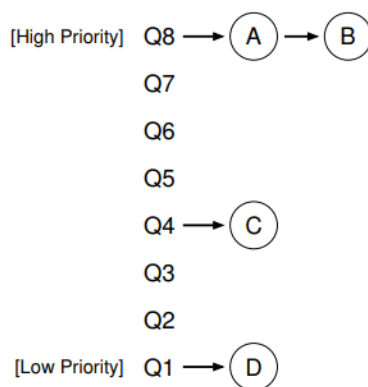


Figure 19: MLFQ example

Currently, however, as job A and B are in the higher queue, the CPU usage will just rotate between A and B in RR fashion, leaving no space for C and D. This is not appropriate, and therefore, requires a way to **change priority**.

As stated above, our workload includes a mix of **interactive jobs that are short-running** (and may periodically give up CPU usage for I/O request) and some **CPU-bound long-running** jobs that need a lot of CPU usage, but the response time is not important. Therefore, to ensure that our workload is complete we need a way to change the priority of jobs. This is done via a new concept, which is called **job's allotment**. Allotment refers to the amount of time a job can spend at a given priority level before the scheduler reduces its priority. With that we have some more rules:

Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).

Rule 4a: If a job uses up its allotment while running, its priority is reduced (i.e., it moves down one queue).

Rule 4b: If a job gives up the CPU (for example, by performing an I/O operation) before the allotment is up, it stays at the same priority level (i.e., its allotment is reset) as it could possibly be a short-running job.

Let's take a look at some examples to examine our rules in action:

Example 1: A Long-running Job with a short-running job

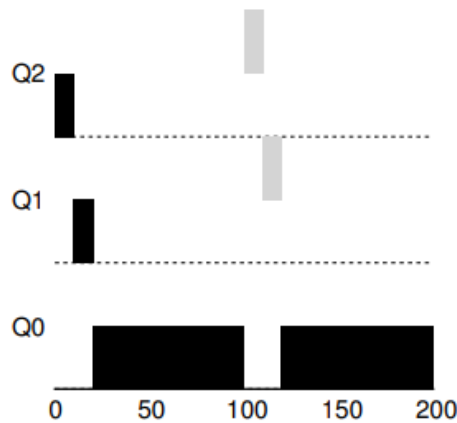


Figure 20: Long job and short job allocation using current rules.

With the stated rules, *figure 20* shows a long-running job (shown in black) enter our topmost queue in its initial execution process, having its queue (and priority) gradually reduces over an allotment of 10 ms, until it reaches the bottom most queue and stays there. At 100 ms, a short running-job (shown in grey) enter our topmost queue and also gradually moves to lower queue as it uses up its allotment, but the short-running job finishes its execution within 20 ms, after that the CPU usage is return to our long-running job. This satisfies **Rules 3 and 4a**, ensuring the CPU usage for both the short-running and long-running jobs.

Example 2: A Long-running Job with an I/O job

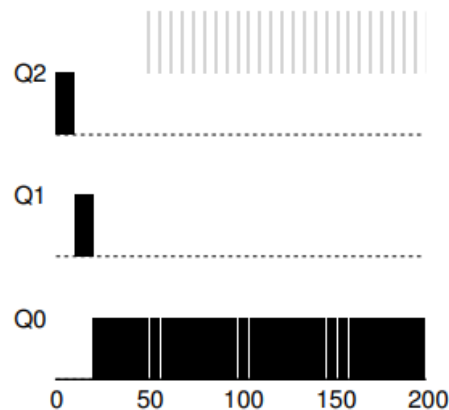


Figure 21: I/O and long job allocation using the current rules.

I/O job can be described as short-running job that does not used up its allotment (although not all jobs that does not use up its allotment is I/O job), in such case, **Rule 4b** is applied allowing for I/O job to retain its priority (and staying on the same queue). This will ensure interactivity of I/O job, and in conjunction with **Rule 3, Rule 4a**, ensure fairness in our workload.

The Problem with our current MLFQ

Perfect as it seems, our current MLFQ contains two major flaws. First, what if there are too many interactive jobs which will consume the CPU usage at a higher priority level over long-running job? This means that the long-running job will never have CPU usage, making them **starve** as shown in the figure below. Second, **Rules 4a and 4b** allow for a process to keep its priority level if its allotment is not used up before switching to another process. This means that if a process uses 99% of its allotment and then issue an I/O operation (write something to a file for example), before switching back to its execution, it can do this repeatedly to trick the OS into giving it full use of the CPU all the time. This is referred to as **Gaming the Scheduler**.

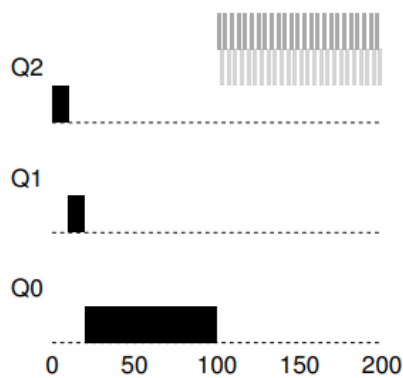


Figure 22: Starvation problem

Problem 1: Starvation

To address the problem of starvation, we can periodically boost the priority of all the jobs in the system to the highest priority meaning throwing them to the topmost queue:

Rule 5: After some time period S , move all the jobs in the system to the topmost queue.

As shown in figure 22, without **Rule 5**, long-running job is starved of CPU usage once two short-running jobs arrive. This, however, changes when we move all the jobs periodically to the topmost queue after a certain period, ensuring that the long-running job will run in Round Robin (according to **Rule 1**) with other short-running jobs. This is illustrated in the following figure:

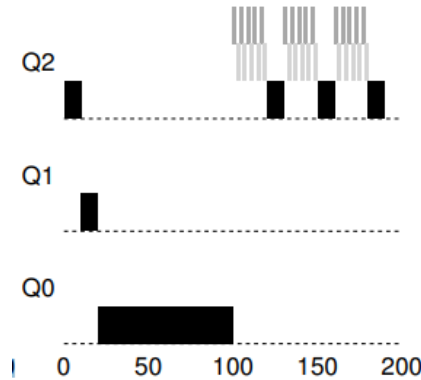


Figure 23: Rule 5 prevents starvation.

Problem 2: Gaming the Scheduler

The main reason why jobs can be rewritten to game the scheduler is because we allow them to retain their priority if they do not use up their allotment before switching to other job (**Rules 4a and 4b**), without accounting for how much allotment they have use at a certain priority level.

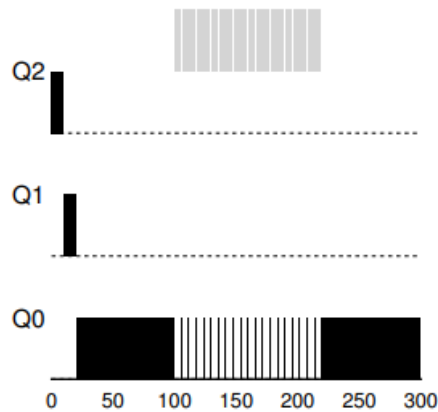


Figure 24: Gaming the Scheduler

Therefore, we need to account for the CPU usage of each job at each level, once a process has used up its allotment, it should be demoted to the lower priority queue. We thus rewrite **Rules 4a and 4b** to the following single rule to facilitates **Gaming Tolerance**:

Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).

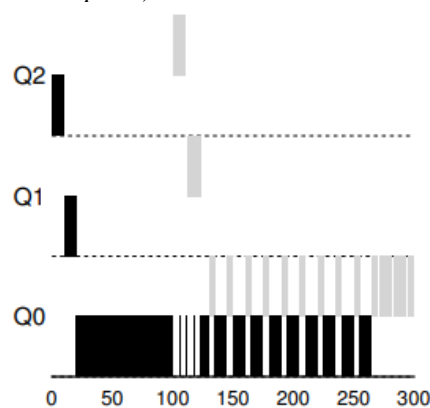


Figure 25: Gaming Tolerance

There is still one major problem regarding MLFQ, it is how can we parameterize MLFQ's attributes, namely, how many queues should there be? How big should the time slice be per queue? How often should we boost all jobs to the topmost queue? How long should the allotment be? And unfortunately, there is no single easy answer for all of those questions; they all depend on the workload and the tuning of the scheduler time after time to find an appropriate answer.

Week 4

Overview: This week's knowledge revolves around the **concept of threading**; however, I am a bit disappointed as we did not mention **address space** in this course, without it, I could not understand the fundamental differences between process and thread.

Address Translation

For simplified OS, it places the entire memory image in one chunk, and when a process is initiated, it allocates parts of the physical memory for the process. Making the process believes that it has the entire physical memory to its usage (memory virtualization aspect of the OS)

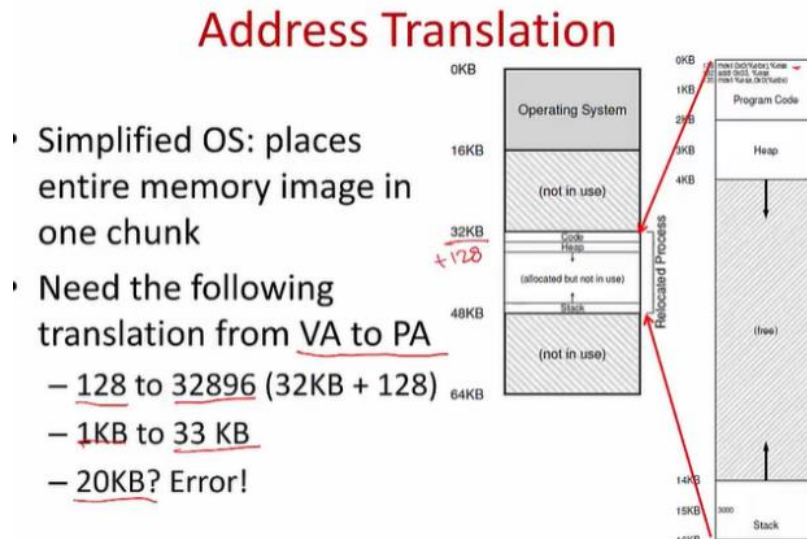


Figure 26: Address Translation

Single-threaded process and multi-threaded process

So far, we have studied the concept of single threaded process and process execution, but a program can also be a multi-threaded process.

So, what is a thread? A thread is like another copy of a process that executes independently, threads share the same address space (code, heap), each thread has separate program counter meaning each thread may run over different part of the program. Each thread has a separate stack for independent function calls.

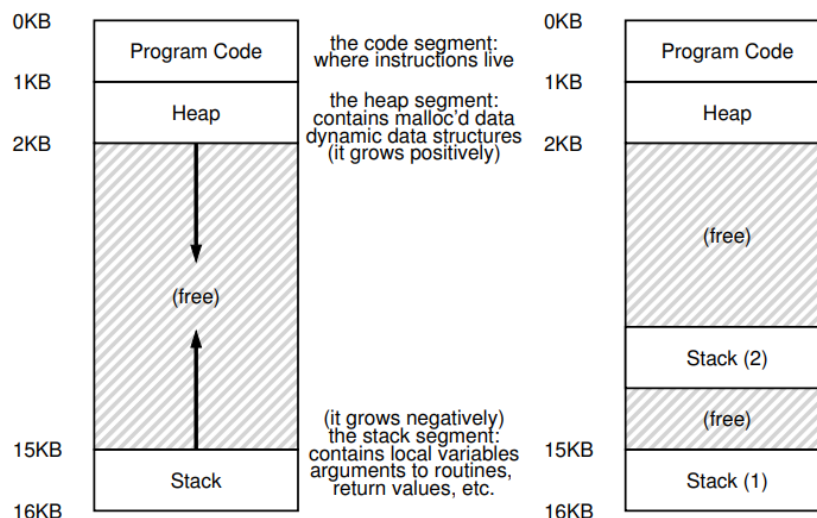


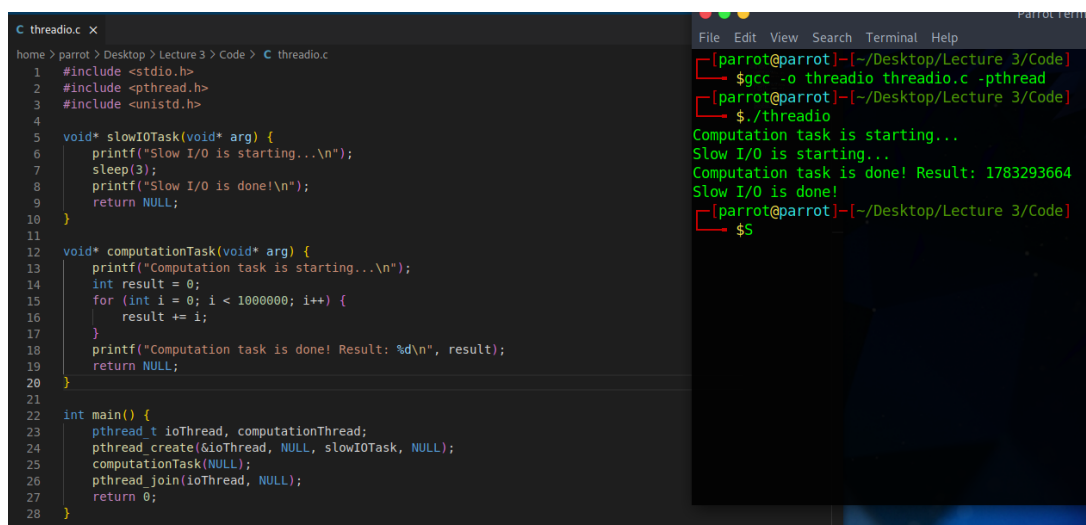
Figure 27: Single-threaded and multi-threaded address space

Why do we use thread?

Reason 1: Thread facilitates parallelism, if all of our programs are single-threaded running on a system with one processor, all of our operation will only be executed once at a time using one processor. But if we have a system with two or more processors, multi-threaded programs have the potential of speeding up our execution process considerably by using the processors to perform each portion of the work.

Reason 2: Thread enables overlap of I/O with other activities within a single program, so that our program progress is not blocked due to slow I/O.

The code below illustrates this point as we have two tasks: *slowIOTask* and *computationTask*. The *slowIOTask* simulates a slow I/O operation by sleeping for 3 seconds. The *computationTask* performs a computationally intensive task by calculating the sum from 0 to 999,999. By creating a separate thread for the slow I/O task using `pthread_create`, we allow the computation task to run concurrently in the main thread. This enables overlap of I/O with computation. The main thread starts the computation task immediately after creating the I/O thread, without waiting for the I/O operation to complete.



```
C threadio.c x
home > parrot > Desktop > Lecture 3 > Code > C threadio.c
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4
5  void* slowIOTask(void* arg) {
6      printf("Slow I/O is starting...\n");
7      sleep(3);
8      printf("Slow I/O is done!\n");
9      return NULL;
10 }
11
12 void* computationTask(void* arg) {
13     printf("Computation task is starting...\n");
14     int result = 0;
15     for (int i = 0; i < 1000000; i++) {
16         result += i;
17     }
18     printf("Computation task is done! Result: %d\n", result);
19     return NULL;
20 }
21
22 int main() {
23     pthread_t ioThread, computationThread;
24     pthread_create(&ioThread, NULL, slowIOTask, NULL);
25     computationTask(NULL);
26     pthread_join(ioThread, NULL);
27     return 0;
28 }
```

```
[parrot@parrot]~/Desktop/Lecture 3/Code
$ gcc -o threadio threadio.c -pthread
[parrot@parrot]~/Desktop/Lecture 3/Code
$ ./threadio
Computation task is starting...
Slow I/O is starting...
Computation task is done! Result: 1783293664
Slow I/O is done!
[parrot@parrot]~/Desktop/Lecture 3/Code
$
```

Figure 28: Simple multi-threaded program

Difference between Thread and Process

The state of a single thread is similar to that of a process. It has a program counter (PC) that keeps track of the instruction being fetched. Each thread has its own set of registers for computation. When switching between threads on a single processor, a context switch occurs, where the register state of the current thread is saved, and the register state of the next thread is restored. This context switch is similar to the one that occurs between processes, but instead of saving state to a **process control block (PCB)**, we need one or more **thread control blocks (TCBs)** to store the state of each thread within a process. However, unlike process context switches, thread context switches do not require switching the address space or page tables because the address space remains the same.

The concept of thread and process in combination facilitates multitasking at multiple levels. The OS multitasks by having multiple processes, process multitasks by having multiple threads. This in turn enables applications to be made more real-time continued as different levels of continuity existed throughout their operations.

Key Approaches of Threading

Manager/Worker approach: This approach consists of two entities, a primary managing thread and many worker threads. The primary manager thread is responsible for handling I/O or other resources and it will assign smaller work for a collection of worker threads to parallelize the work. Typically, the collection of worker threads is created from a **thread pooling process**, where a collection of threads is initialized at once and kept around as long as they are needed.

Pipeline approach: This approach consists of many threads working together as an “assembly line” each product of a line is passed to the next line until the work is completed.

Example 1: Thread Creation in C on Unix System

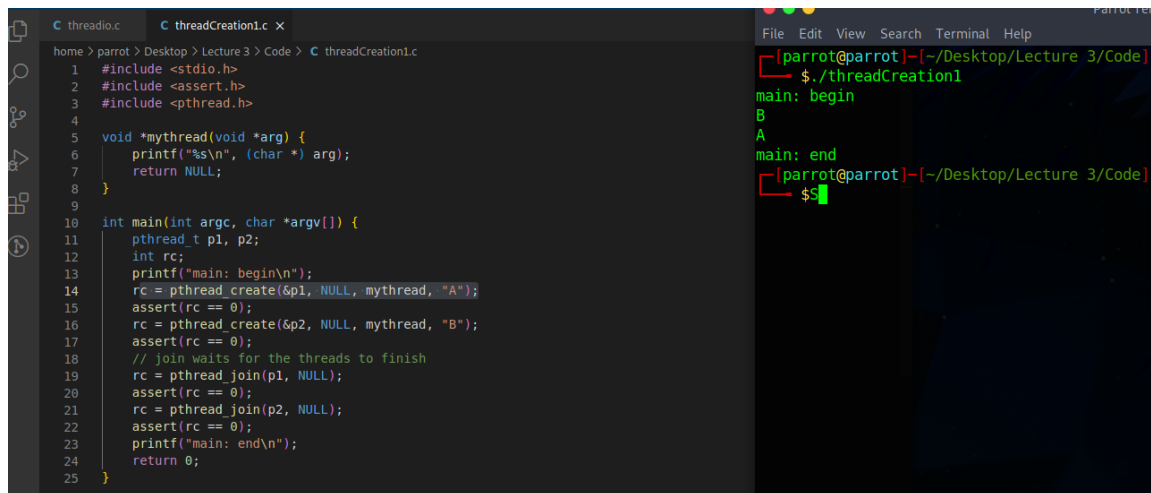
The image shows a code editor with a C program named 'threadCreation1.c' and a terminal window showing its execution. The C code includes `<stdio.h>`, `<assert.h>`, and `<pthread.h>`. It defines a function `mythread` that prints a string and returns `NULL`. The `main` function creates two threads, 'A' and 'B', using `pthread_create`. It then joins both threads using `pthread_join` and prints 'main: end' before returning 0. The terminal output shows the execution of the program, with 'main: begin' printed first, followed by 'A' and 'B' printed in parallel, and finally 'main: end' printed after both threads have finished.

Figure 29: Thread Creation in C on Unix System

In the above code, the code creates two threads using the *pthread* library functions. Each thread executes the *mythread* function, which simply prints a string. The program waits for both threads to finish before printing the final message and terminating.

This can also be done in *Java* using *Java APIs*, threads can be used in *Java* by either extending the *Thread* class or by implementing the *Runnable* interface.

```
class MyThread extends Thread
{
    public void run()
    {
        // ...
    }
}
// ...
Thread a = new MyThread();
a.start();
```

Figure 30: Extending thread class.

```
class MyRun implements Runnable
{
    public void run()
    {
        // ...
    }
}
// ...
Thread b = new Thread(new MyRun());
b.start();
```

Figure 31: Runnable interface.

Uncontrolled Scheduling in multi-threaded program

The simple thread example above demonstrates how threads are created, but if there is some calculation that involves shared data involves, problems will arise. Take the following code and outputs as an example:

```

1 #include <stdio.h>
2 #include <pthread.h>
3
4 static volatile int counter = 0;
5
6 void *mythread(void *arg) {
7     char *thread_id = (char *)arg;
8     printf("%s: begin\n", thread_id);
9     for (int i = 0; i < 10000000; i++) {
10         counter = counter + 1;
11     }
12     printf("%s: done\n", thread_id);
13     return NULL;
14 }
15
16 int main(int argc, char *argv[]) {
17     pthread_t p1, p2;
18     printf("main: begin (counter = %d)\n", counter);
19     pthread_create(&p1, NULL, mythread, "A");
20     pthread_create(&p2, NULL, mythread, "B");
21
22     // join waits for the threads to finish
23     pthread_join(p1, NULL);
24     pthread_join(p2, NULL);
25
26     printf("main: done with both (counter = %d)\n", counter);
27     return 0;
28 }

```

```

in'
/usr/bin/ld: threadSharedData.c:(.text+0x108): undefined reference to 'main'
collect2: error: ld returned 1 exit status
[parrot@parrot]~/Desktop/Lecture 3/Code
$gcc -o threadSharedData threadSharedData.c -lpthread
[parrot@parrot]~/Desktop/Lecture 3/Code
$./threadSharedData
main: begin (counter = 0)
B: begin
A: begin
B: done
A: done
main: done with both (counter = 13935787)
[parrot@parrot]~/Desktop/Lecture 3/Code
$./threadSharedData
main: begin (counter = 0)
B: begin
A: begin
B: done
A: done
main: done with both (counter = 12431193)
[parrot@parrot]~/Desktop/Lecture 3/Code
$

```

Figure 30: Data sharing multi-threaded program.

In our code, specifically, the *mythread* function, we add 1 to the variable counter, and do so 1e7 meaning 10 million time for each thread, therefore, our desired value should be 20,000,00. However, the output of our code shows the calculated value to be considerably less than expected. In order to understand why this happens, we identify the segment of code that is responsible for our calculations and inspect its assembly code. *Mythread* function would be our target, and the code that is responsible for the calculation is as follows:

```

118a: 8b 05 b4 2e 00 00    mov     0x2eb4(%rip),%eax    # 4044 <counter>
1190: 83 c0 01             add     $0x1,%eax
1193: 89 05 ab 2e 00 00    mov     %eax,0x2eab(%rip)    # 4044 <counter>

```

Figure 31: Critical section

This is pretty much the expected behavior of multi-threaded program without controlled scheduling, and is commonly referred to as a **race condition**, whereas concurrent execution can lead to different results. The portion of code that led to the race condition is called critical section.

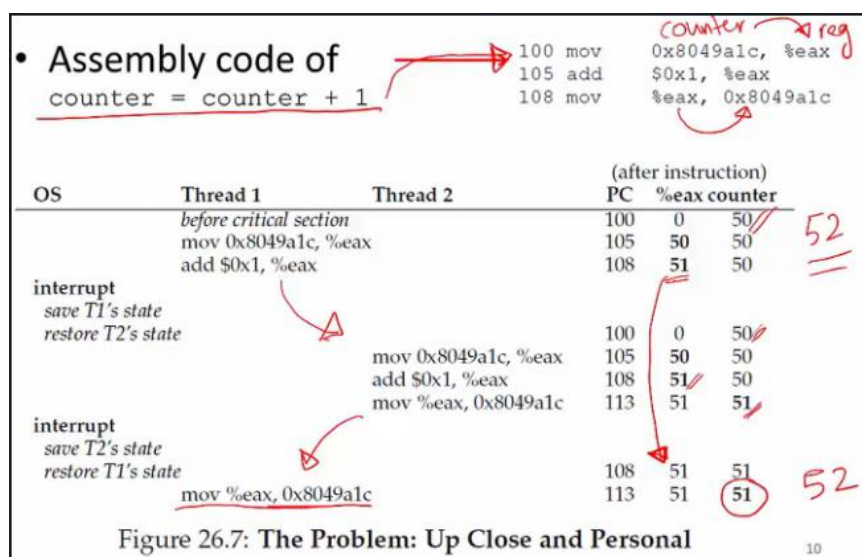


Figure 26.7: The Problem: Up Close and Personal

Figure 32: Simplified example

We can now see why our code did not produce the expected output, as when *Thread1* is about to store its calculated value to the register, an interrupt occurs that switch context to *Thread2*. *Thread2*, with a different stack meaning different variables,

performs its calculation, stored the value to the register and context switch back to *Thread1* when an interrupt occurs. *Thread1*, at this time, executes the instructions that it missed in the previous turn, save its calculated value to the register, overwriting the calculated value of *Thread2*. Making the output not consistent and accurate.

In the essence of **critical section** we need a **mutual exclusion**, meaning only one thread should be executing the critical section at any time, enforcing the **atomicity** of the critical section. This is achieved using **Locks**.

Week 5

Overview: This week's knowledge revolves around the **concept of locking**, and why do we need locks, as I had discussed the concept of data sharing multi-threaded program in week 4, as well as the problem coming with it. We can just review some of the key concepts and then move to lock right away.

Key concepts regarding Concurrency

A **critical section** is a piece of code that accesses a shared resource, usually a variable or data structure.

A **race condition** arises if multiple threads of execution enter the critical section at roughly the same time; eg., both attempt to update the shared data structure, leading to an undesirable outcome.

An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when.

To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

Implement mutual exclusion of critical section using lock mechanism.

The purpose of lock mechanism is to enforce atomicity, meaning that only a task is allowed to execute and complete, or the task does not run at all, this will, in turn, implement mutual exclusion required for avoiding race condition. Let's take a look at how we can implement this in Java.

Lock object.

In Java, lock is an instance of a lock object, which is used for thread synchronization, it is responsible for coordinating access to a shared resource or a critical section of code.

We will rewrite the C code from Week 4 into Java and use the *ReentrantLock* Class, it is noteworthy that our counter variable is a **static variable**, meaning that it is associated with a *class* rather than an *instances* (objects), therefore, any changes made to a **static variable** will be reflected across all objects of the class as shown in the example below:

```
1 import java.util.concurrent.locks.Lock;
2
3
4 public class CounterExampleUsingReentrantLock {
5     private static volatile int counter = 0;
6     private static Lock lock = new ReentrantLock();
7
8     public static void main(String[] args) throws InterruptedException {
9         Thread thread1 = new Thread(new MyThread("A"));
10        Thread thread2 = new Thread(new MyThread("B"));
11
12        System.out.println("main: begin (counter = " + counter + ")");
13        thread1.start();
14        thread2.start();
15
16        thread1.join();
17        thread2.join();
18
19        System.out.println("main: done with both (counter = " + counter + ")");
20    }
21
22    static class MyThread implements Runnable {
23        private String threadId;
24
25        public MyThread(String threadId) {
26            this.threadId = threadId;
27        }
28
29        public void run() {
30            System.out.println(threadId + ": begin");
31            for (int i = 0; i < 10000000; i++) {
32                lock.lock();
33                try {
34                    counter = counter + 1;
35                } finally {
36                    lock.unlock();
37                }
38            }
39            System.out.println(threadId + ": done");
40        }
41    }
42 }
```

Figure 33: ReentrantLock Class

In the above code, specifically inside the `run()` method, a loop is executed 10 million times to increment the counter variable, before incrementing, the thread acquires the lock using `lock.lock()`. This ensures that only one thread can enter the critical section at a time. After the calculation is done, the final block is used to ensure that the lock is always released using `lock.unlock()`. Our **static variable** is provided mutually exclusive access using the lock as we had declared the lock as static also. Thus, solving the problem in week 4 as we did not implement atomicity. The result output is as follows:

```
<terminated> CounterExampleUsingReentrantLock [Java Application] C:\Users\LUAN\
main: begin (counter = 0)
A: begin
B: begin
B: done
A: done
main: done with both (counter = 20000000)
```

Figure 34: ReentrantLock result

Synchronized keyword for thread synchronization.

Different from the Lock object above, the synchronize keyword uses intrinsic locks, also known as monitor locks. This is feasible because each object in Java has an associated monitor, and the synchronized keyword acquires and releases the monitor lock of the object. Making it operated at the block or method level, granting synchronization for the entire block or method. Lock object, on the other hand, provides more fine-grained control and can be used to acquire and release locks at specific points within a method as shown with the `counter = counter + 1` example above. Let's take a look at how this is implemented in Java.

```
1 public class CounterExampleUsingSynchronized {
2     private static int counter = 0;
3
4     public static void main(String[] args) throws InterruptedException {
5         Thread thread1 = new Thread(new MyThread("A"));
6         Thread thread2 = new Thread(new MyThread("B"));
7
8         System.out.println("main: begin (counter = " + counter + ")");
9         thread1.start();
10        thread2.start();
11
12        thread1.join();
13        thread2.join();
14
15        System.out.println("main: done with both (counter = " + counter + ")");
16    }
17
18    static class MyThread implements Runnable {
19        private String threadId;
20
21        public MyThread(String threadId) {
22            this.threadId = threadId;
23        }
24
25        public void run() {
26            System.out.println(threadId + ": begin");
27            for (int i = 0; i < 10000000; i++) {
28                synchronized (CounterExampleUsingSynchronized.class) {
29                    counter = counter + 1;
30                }
31            }
32            System.out.println(threadId + ": done");
33        }
34    }
35 }
```

Figure 35: Synchronized keyword

```
<terminated> CounterExampleUsingSynchronized [Java Application] C:\Users\LUAN\p2
main: begin (counter = 0)
A: begin
B: begin
A: done
B: done
main: done with both (counter = 20000000)
```

Figure 36: Synchronized keyword output

Inside the run method, a loop is also executed 10 million times similar to the above example. However, this time, the synchronized block is used to ensure that only one thread can enter the critical section at a time and the *CounterExampleUsingSynchronized.class* is casted to the synchronized keyword to ensure mutually exclusive access to our **static variable**.

From the two examples above, we can understand how lock is implemented to solve the problem in week 4, it can also be seen that the use of synchronized keyword is simpler than the use of Lock object. However, it is crucial to note that Lock object provides additional fine-grained control with more functionality. For example:

+ *Lock.tryLock()*: This will return a Boolean to confirm whether a lock can be acquired or not, we can also specify a timeout from nanoseconds to days (*lock(tryLock(50, TimeUnit.SECONDS))*).

Week 6

Overview: We will dive deeper into the concept of locking in this week, some of the bullet points in this week is already covered last week (namely static variable), so we will go briefly through them to get to our main discussing topics.

Static Variable

As discussed in week 5, counter variable is a **static variable**, meaning that it is associated with a *class* rather than an *instances* (objects), therefore, any changes made to a **static variable** will be reflected across all objects of the class. We solve this problem by declaring the lock object as *static* and casting the class of the object to the *synchronized* keyword. We will, therefore, summarize our understanding about locks this week.

What are the goals of lock implementation?

Mutual Exclusion: This is obvious as lock is implemented to prevent multiple threads from accessing the critical section and corrupting the underlying data.

Fairness: This states that all threads should eventually get the lock, and no thread should starve (of CPU usage of course).

Low overhead (performance): This states that acquiring, releasing, and waiting for lock should not consume too many resources.

Let's take a look at some examples of lock implementation to assess their effectiveness in satisfying the above criteria.

Disabling Interrupts

Recall from the C code in Week 4, we can see that our problem about atomicity arises when a thread is executing its instructions while suddenly an interrupt happens and the thread is forced to switch context to another thread, so can we just implement our lock that disable the interrupt?

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

Figure 37: Disabling Interrupts

It seems that controlling interrupts alone is not sufficient as a lock implementation for the following reasons:

- + **Security implications:** As disabling interrupts is a privileged instruction, program can misuse it to make all the processors to run forever.

- + **Not feasible on multiprocessor systems:** Since another thread on another core can also enter critical section making this lock implementation meaningless.

- + **Potential serious system problems:** As turning off interrupts for extended periods of time can lead to the OS not knowing when to wake other processes.

Load and Stores

This seems to be a reasonable implementation of a lock, as the *lock()* method will spin on a flag variable until it is unset, then set it to acquire lock while the *unlock()* method will unset the flag variable.

```
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 38: Load and store

This implementation is also not feasible, as it does not ensure mutual exclusion similar to the problem of the code we saw in Week 4, *Thread1* has yet to set the flag to 1 before the interrupts happens and the context is switched to *Thread2*. But at *Thread2* it will set the flag to 1 and when context switches again *Thread1* once again executes the command that it has yet

to do and set the flag to 1. Meaning that we have $\text{flag} = 1$ in both threads. This method is also extremely inefficient as resources must be allocated to check if the flag is set by the while loop or not.

Test and Set

As shown in the previous implementation, it is very difficult to implement lock at software level, therefore, modern architectures provide hardware atomic instruction, namely *test-and-set*:

```
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 39: Test and set instruction.

The reason it is called “test and set” is that it enables us to “test” the old value (which is what is returned) while simultaneously “setting” the memory location to a new value.

```
1  typedef struct __lock_t {
2      int flag;
3  } lock_t;
4
5  void init(lock_t *lock) {
6      // 0 indicates that lock is available, 1 that it 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 40: Simple spin lock using test-and-set

The above code implements test-and-set using the *TestAndSet(flag, 1)* return 1, it means the lock is held by someone else. This is also called a spin lock, meaning spinning until a lock is acquired. This implementation ensures mutual exclusion, however, it does not satisfy the fairness criteria as each thread will occupy an entire time slice on the CPU spinning and waiting.

There is another atomic hardware instruction called **compare-and-swap**

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

Figure 41: Compare and Swap

We can therefore build a lock in a manner quite similar to that with *test-and-set*

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

This method does not set the flag every time it is called, as it compares the actual value with the expected value and set the flag only if they match, making it more efficient than *test-and-set*.

Fetch and Add

Fetch and add simply adds to the value pointed to by ptr. The result is a “ticket” and “turn” variable that implements the locking functionality.

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

Figure 42: Fetch-and-add.

Although the methods discussed above, including test and set, compare and swap, and fetch and add, offer locking capabilities and perform well on multiprocessors, they still face the challenge of context switching. In these methods, a thread holding the lock can monopolize the critical section, preventing other threads from accessing it and causing them to spin indefinitely. We need an alternative to a spinlock, and *yield()* is the next implementation that we will discuss.

Yield()

This implementation is a type of sleeping mutex that, instead of spinning for a lock, a contending thread could simply give up the CPU and check back later. This approach is simply a system call that moves the caller from the running state to the ready state, and thus promotes another thread to running. This although alleviates some performance concerns, our problem still exists as if there are a large number of threads, they would still be contending for a lock repeatedly.

```
1  void init() {  
2      flag = 0;  
3  }  
4  
5  void lock() {  
6      while (TestAndSet(&flag, 1) == 1)  
7          yield(); // give up the CPU  
8  }  
9  
10 void unlock() {  
11     flag = 0;  
12 }
```

Figure 43: Yield()

Using Queues

By utilizing a queue, threads that are waiting for a lock can be organized in a sequential order, where the first thread to request the lock is the first one to be served. Moreover, the operating system provides additional methods that enable threads to sleep instead of continuously spinning. The *park()* method allows a thread to be put to sleep, while the *unpark(threadID)* method can be used to wake up a specific thread identified by its *threadID*. By employing these two calls, a thread can be put to sleep whenever it attempts to acquire a lock that is already taken, and it can be awakened once the lock becomes available again.

The table below summarizes the characteristics of each method that we discussed in this week.

	Correctness	Fairness	Performance
Controlling Interrupts	No	No	No
Load and Store	No	No	No
Test and Set	Yes	No	Bad for single processor. Ok for multiprocessor.
Compare and Swap	Yes	No	Better than test and set but still bad for single processor. Ok for multiprocessor.
Fetch and add	Yes	Yes (First come first serve)	Better than test and set but still bad for single processor. Ok for multiprocessor.
Yield() (in concert with other methods i.e. Test and Set)	Yes	Yes	Yes (although a large number of threads can cause delays if a thread is pre-empted before releasing the lock)
Using queues (in concert with other methods i.e. Test and Set)	Yes	Yes	Yes (with additional logic and methods (i.e. <code>park()</code> , <code>unpark()</code>))

Figure 44: Lock implementation summary