Faculty of Engineering

# Principles of Operating Systems

*Semester 1 2023-2024*

Jose Alberto ESPINO PITTI, *UID:3035946813*

2023

Course Code

COMP 3230

## Class Contents

**Check out:** Credits

The contents of these notes are both heavily sourced from the class notes by professor Anthony Tam of COMP3230B at the University of Hong Kong and the book *Operating Systems: Three Easy Pieces* by Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. I do not claim ownership of any of the contents of these notes, and I am only using them for educational purposes. Feel free to contact me at *jespigno (at) connect (dot) hku (dot) hk* if you have any questions or concerns.

# Introduction

Let us introduce some of the main concepts of operating systems that we will cover in this class: Every second a program runs, it fetches instructions from the memory, decodes them, and executes them. This instruction cycle is the basis of the **Von Neumann** model of computer, which is the basis of most modern computers. This process is more complicated than this decievingly simple explanation — a lot of things are going on in the background while a program runs. The **operating system** (OS) is a piece of software that manages the hardware and software resources of a computer system. It is a **resource allocator**, which manages all resources and makes decisions when there are conflicts, and a **control program**, which manages the execution of user programs and I/O devices. The OS ensures that programs run smoothly and efficiently. The OS achieves control over the hardware through *virtualisation*, a virtual representation of a physical resource. The OS also provides a **user interface** (APIs), which allows the user to interact with the computer. An important application of this is the **virtualisation of the CPU**, which allows the computer to run seemingly infinite programs simultaneously on a single CPU. The OS also includes different **policies** to manage the resources, such as **scheduling policies**, which determine which process to run next, and **memory management policies**, which determine how to allocate memory to processes.

The physical concept of memory is quite straightforward: an array of bytes, where information is stored in sections that can be addressed. The OS abstracts this concept of memory into a **logical address space**, which is a virtual address space that is mapped to the physical address space. A memory reference within one running program does not affect the address space of the other running programs. The OS also provides **protection** to ensure that a program cannot access memory that it is not allowed to access. In reality, memory is a shared resource, but virtualisation allows the computer to use memory as if it were a private resource.
**Concurrency** is another key concept. It refers to the ability of the OS to address and work on many issues at once. This leads to several issues to both the OS and muli–threaded programs. A multi–threaded program is a program that can run multiple threads concurrently. A **thread** is a function running with the same memory space as other functions, with one of them active at a time. Sometimes problems arise at the possible inability of the OS to run multiple threads at once, which is known as **starvation**.
Another key term is **persistence**: the OS must be able to store data in a non–volatile way. This is achieved through **file systems**, which are a way to organise data on a storage device. The OS does not create a private virtual representation of the storage device for each application, but rather assumes that the storage device is a shared resource. In order to actually perform I/O operations, the operating system employs **device drivers**, pieces of code that allow the OS to communicate with a specific piece of hardware. OS operations are complicated and the OS must be able to handle errors.
When **designing** an operating system, we need to focus on certain goals, such as:

1. **Performance**: We must minimise the overheads of the OS, focusing on extra time and space.

2. **Protection**: We must ensure that the OS is secure and that it protects the user from malicious programs or from accidental overlap between atomically ran programs. The core idea behind this is **isolation**, which is the idea that each program should not be able to access the memory of other programs.

3. **Reliability**: We must ensure that the OS is reliable and that it does not crash. This is achieved through **fault tolerance**, which is the ability of the OS to continue running in the presence of faults.

4. **Mobility**: We must ensure that the OS is portable and that it can run on different hardware.

### 1.0.1 Operating System Architectures

Operating systems have to offer support to both sophisticated operations that involve manipulation of the underlying hardware as well as simple user–level operations that involve manipulation of files and other resources. The core of the OS is the **kernel**, which allows the OS to manage resources and directly interact with the hardware.
Above the kernel, there is a layer known as the **service layer**. This layer contains a set of utility functions that will be used or called by application programs and the command layer. The functions that this layer contains vary depending on the particular OS. The programs the service layer operates on are **non–kernel** programs; they do not have direct access to the hardware but rather use facilities provided by the kernel.
Outside of the service layer, we have two layers. Firstly, we have the **command layer**, also known as the *shell*, which is the interface between the user and the OS, where communication happens through commands. Secondly, we have the **application layer**, which contains the programs that the user runs.

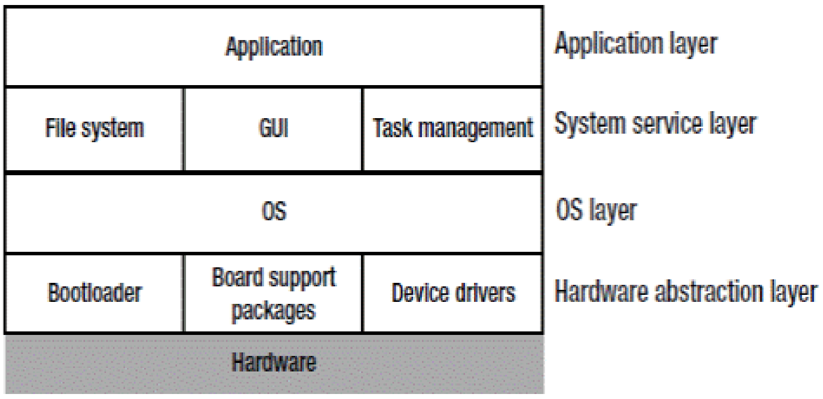


Figure 1.1: Typical software architecture of an embedded system, sourced from here

The OS is a **resource allocator**, which manages all resources, and a **control program**, which manages the execution of user programs and I/O devices. Every operating system has a lot of components, which must support a variety of hardware and software. This can get complex — OS architectures manage this complexity by organising the OS components (functionalities) and **specify the privilege level** with which each component executes. The concept of privilege level is essential: when the processor executes intructions, it does so in (at least) two modes. The first one is **kernel mode**, also known as **supervisor mode**, which is highest privilege level and can execute **any** instruction. The second one is **user mode**, which is the lowest privilege level and can only execute a subset of instructions. The reason why these tiers of privilege exist in the first place is us not wanting any application or user to be able to manipulate hardware, as this could lead to security issues. Instructions related to hardware are executable only in kernel mode.
When an application or user wants to execute an instruction, the operating system executes a `system call`: a special instruction that allows an application program to make a request to the OS for resources or services. The OS then checks the validity of the request and executes the request on behalf of the application and returns control to the application, which continues its execution (a process called **mode switch**). The instructions that trigger the mode switch are also known as **trap instructions**. The set of system calls is the **interface (API)** to the services provided by the OS. System calls are the main mechanism that triggers a switch between user and kernel mode.

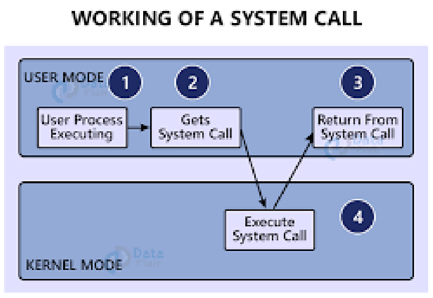


Figure 1.2: System Call in OS, sourced from here

Let us cover some common OS architectures:

- Monolithic Architectures:
  **Monolithic** OS architectures are the simplest, where all OS components execute in the same address space (inside the kernel), and the OS is a single binary. This is the case of MS–DOS,

UNIX, and Linux. The issue with these architectures is that traditionally its code did not structure its components as modules with clearly defined interfaces — all the components are interwoven into a large program that runs in kernel mode. This makes it difficult to add new functionality to the OS and to isolate bugs in the code. In spite of these issues, most modern OSs are still monolithic, but they are structured as a set of modules that can be loaded into and removed from the kernel at runtime. The reason behind sticking to monolithic architectures is that they are more efficient than other architectures, as there is no overhead for communication between kernel–level components; any component within the kernel can directly communicate with any other.

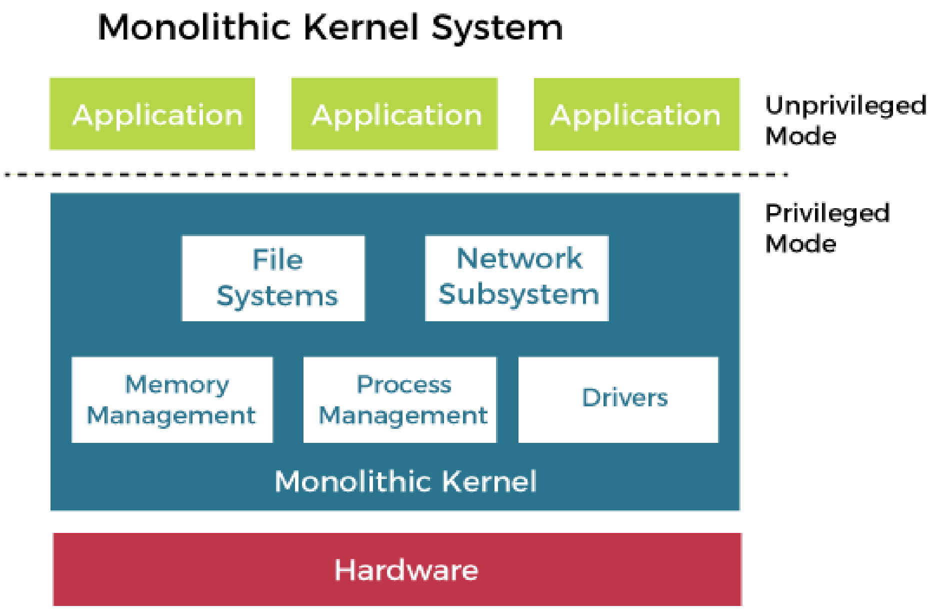


Figure 1.3: Monolithic Architecture, sourced from here

- Layered Architectures:
  It groups components that perform similar functions into a module and layers the modules one on top of the other. The bottom layer is the hardware, and the top layer is the user interface. Each layer only uses functions and services of the layer directly below it.

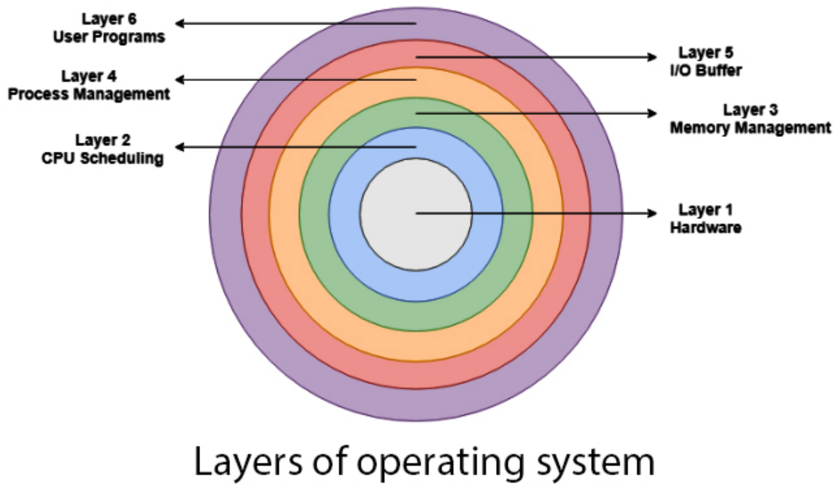


Figure 1.4: Windows NT Operating System (Layered), sourced from here

Only being able to communicate with the layers directly above and below it makes communicatio1n inefficient: processes might have to pass through many layers before being serviced. The advantage of this architecture is that it is simpler to construct and debug, it is more secure because of information hiding, and it is easier to replace layers. An example of this architecture is the *virtual machine*, an isolated environment for your application to run on. It is a software emulation of a physical computer, which allows you to run an OS within an OS. Virtual Machines run on top of the host operating system, in either user mode or incorporated into the host kernel mode. The OS running on the virtual machine is known as the **guest OS**. The guest OS is not aware that it is running on a virtual machine, as it is running in kernel mode. The virtual machine is a **process** in the host OS, and the host OS is aware that it is running on a virtual machine. The host OS is responsible for scheduling the virtual machine, and the virtual machine is responsible for scheduling the guest OS.

- Microkernel Architectures:
  It is a minimalist approach to OS design, where the kernel is broken down into separate processes, known as **servers**, which communicate through message passing. The kernel only provides basic functionality, such as inter–process communication, low–level memory management, and basic scheduling. The rest of the OS is implemented as a set of user–level processes, known as **clients**, which request services from the servers. The advantage of this architecture is that it is more secure, as the kernel is a small piece of code, and it is more reliable, as a bug in a server will not crash the entire system. The disadvantage is that it is slower, as there is overhead for communication between servers and clients — all the communication has to go through the kernel first.

- Modular Approach:
  This is not necessarily an architecture but rather a way to organise the information/code so you have a better environment to work with. Most modern architectures are not fully monolithic, but rather implement the monolithic approach in a modular way. In a nutshell, the modular approach implies that each core component is separated and implemented as a module, making the kernel just a collection of modules. It is similar to the layered approach, but communication is more efficient since all the modules are in the kernel. It also allows for certain features to be

implemented dynamically and loaded as needed, something known as **dynamically loadable modules**. Some examples of this approach are Linux and traditional Unix kernels.

# Process Abstraction

An important question that may arise is how the operating system (a piece of software) can manage an application (another piece of software). The answer is that the OS manages the application by managing the **processes** that make up the application. A process is a program in execution, which consists of the program code, program counter, registers, and data section. Processes can be assigned to and executed on a CPU. Similarly, a **unit of activity** is the execution of a sequence of instructions with an execution state and an associated set of system resources. The OS allows us to run several processes at once by **virtualising the CPU**, which is the process of making the CPU look like it is running many processes at once. This is achieved by **time–sharing**, which consists of dividing the CPU time between multiple processes. The more you time share, the slower the execution of each program. The OS managesthis by **scheduling** the processes, determining which process to run next. Proper CPU virtualisation is achieved through both hardware **mechanisms** and OS **policies**, which are algorithms for making decisions within the OS. Ideally, mechanisms and policies are **modular** and thus separate in their design.
The abstraction of memory is known as **address space**. Every application will have its private space, which is an isolated range of memory addresses. It goes from zero to a maximum value; this value is **fixed** — every process sees the same amount. This maximum value depends on the processor architecture, namely on the size of the address bus. For instance, if you have a 32–bit address bus, the maximum value is $2^{32}$. The address space is divided into two sections: the **stack**, which grows downwards, and the **heap**, which grows upwards. The stack is used for local variables and function calls, and the heap is used for dynamic memory allocation (needed for data structures such as linked lists, hash tables, trees, and others). Before the stack and heap, we have the **text segment**, which contains the program code, and the **data segment**, which contains the global variables.

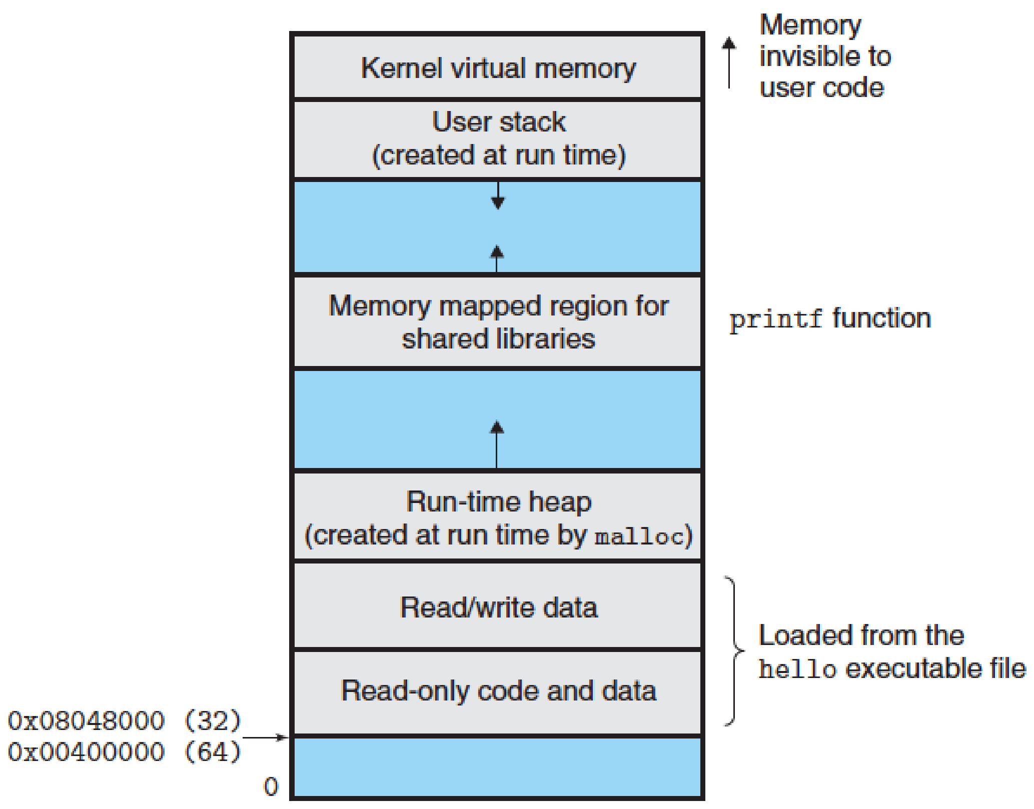


Figure 1.5: Address Space, sourced from here

Address space is not entirely private—the uppermost section will be reserved for OS in kernel privilege level. This is known as **kernel space**, and the rest is known as **user space**. There are some exceptions to this, for instance, MAC OS X reserves an entirely different address space for the kernel space.
In the process execution life cycle, it moves through a series of **discrete** states:

1. **New** (initial): The process is being created.

2. **Ready**: The process is waiting to be assigned to a processor. The OS has to do **scheduling** and see if there is a free processor to handle it. The program might go to either running or blocked state.

3. **Running**: Instructions are being executed on a processor. *Descheduling* might happen at this state; it is when the OS decides to stop the process and give the processor to another process. The program goes back to ready state when this is the case.

4. **Blocked**: The process is waiting for some event to occur before it can proceed (e.g. I/O completion). In this state, the process is not using the CPU.

5. **Terminated** (final/zombie): The process has finished execution but has not yet been cleaned up. It cannot be just discarded because the parent process needs to examine the return code of the process and check if it executed successfully.

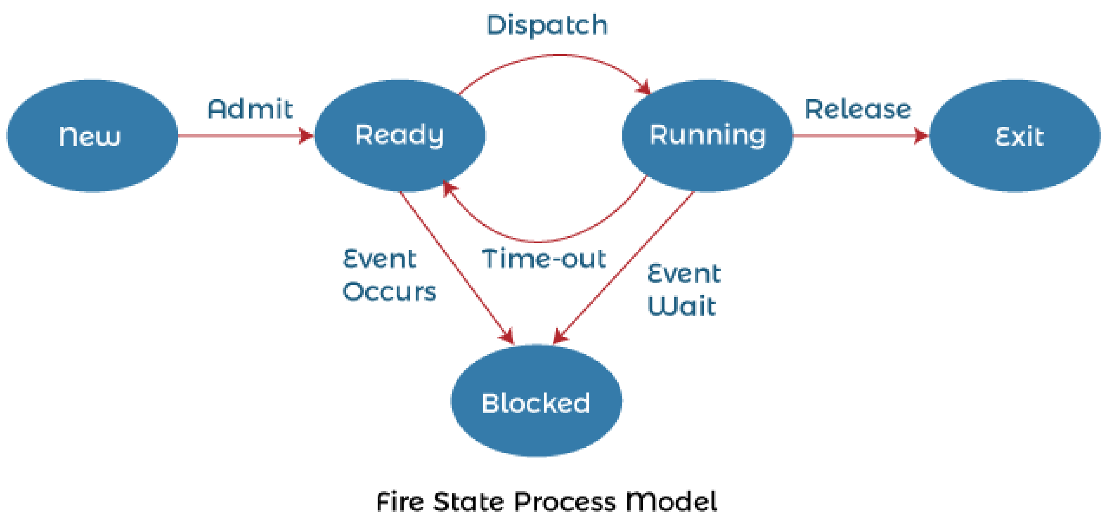


Figure 1.6: Process Execution States, sourced from here

To manage a process, the OS makes use of a **data structure** to keep information about that process. This structure is known as **process control block** (PCB) or process descriptor. Usually, this structure contains the process identification number (PID), the current process state, the program counter (storing the address of the next instruction), the register context (a snapshot of the contents of the registers when the process was last running), and scheduling information (process priority, pointers to scheduling queues, etc). Other items also found in the PCB are credentials (determine the resources the process can access), memory management information, accounting information, a pointer to the process's parent process, pointers to the process's child processes, and pointers to allocated resources, amongst others. In Linux, the process descriptor structure is stored as `struct task_struct` in `/usr/src/linux/include/linux/sched/h`. For instance:

```c
    struct task_struct {
        volatile long state;  /* -1 unrunnable, 0 runnable, >0 stopped */
        void *stack;
        atomic_t usage;
        unsigned int flags;  /* per process flags, defined below */
        unsigned int ptrace;
        int lock_depth;    /* BKL lock depth */
        int load_weight;  /* for niceness load balancing purposes */
        int prio, static_prio, normal_prio;
        struct list_head run_list;
        struct prio_array *array;
        /* and so on */
    };
```

The OS uses a **process table** to manage many processes. This table keeps pointers to each process's PCB. In Linux, this is organised in the form of a hashed table.

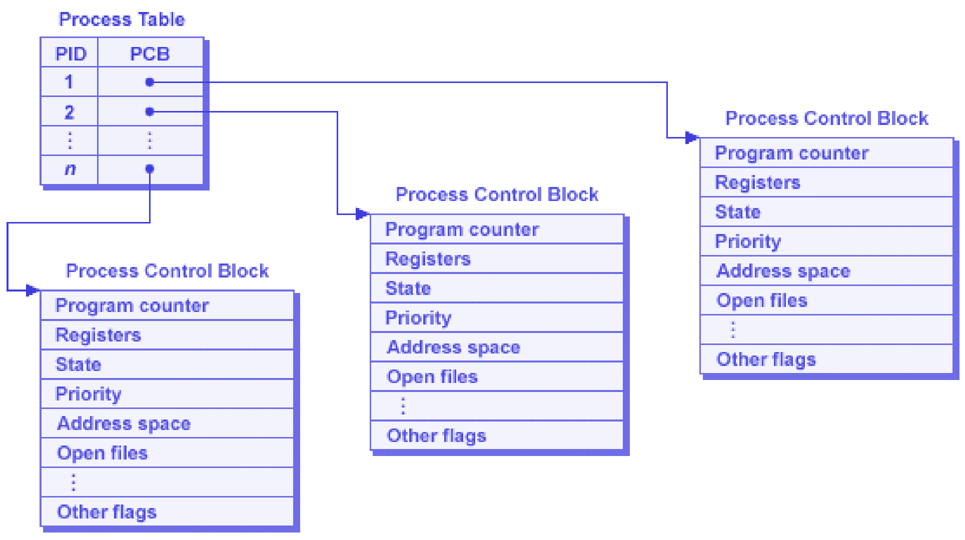


Figure 1.7: Process table in Linux, sourced from here

When a process is terminated for good, the operating system removes it from the table and frees all of the process's resources.

**Check out:** Linux Commands

- `ps [option]`: list the processses running on the system. Some options are:
  - `-e`: list all processes.
  - `-f`: full format.
  - `w`: wide output.
  - `f`: forest output (ASCII-art process hierarchy).
- `pstree [option]`: displays the processes as a hierarchy showing parent and children processes.
- `kill [SIGNAL] PID`: terminates a specified process by sending a signal. Some common signals are `SIGTERM` or `15` (terminate), `SIGKILL` or `9` (kill), `SIGSTOP` (stop), and `SIGCONT` (continue). If you want to see more possible signals, run the command `kill -l`.
  Fun fact: every time you use `ctrl+c`, you are sending a `SIGINT` signal to the process.
- `cd /proc` -> `ls`: displays the processes running on the system. Each process is represented by a directory with its PID as its name. Inside each directory, there are several files that contain information about the process. For instance, `/proc/1/status` contains information about the process with PID 1. The proc file system is an interface to kernel data structures that is used by the OS to provide information about the system. It is also used by the OS to change certain kernel parameters at runtime. The zombie process and its information will be cleared when the parent calls `waitpid()`.

Signals are a way for OS or other software entities to notify another software entity that an event has occurred. They are sometimes referred as *software interrupt*, even though they are **not** interrupt — interrupt has higher priority and is generated by hardware. They can be **synchronous** (generated by the current running process) or **asynchronous** (generated by other processes). Synchronous signals

include division by zero, illegal memory access, and others. Asynchronous signals include `SIGINT` (generated by `ctrl+c`), `SIGTERM` (generated by `kill`), and others. When a process receives a signal from another process, it can either **ignore** it, **catch** it, or **terminate**. If the process ignores the signal, it will continue its execution. If the process catches the signal, it will execute a **signal handler**, which is a function that handles the signal. If the process terminates, it will stop its execution. Another possibility is for the program to **mask** the process: the OS will not deliver signals of that type until the process clears the signal mask. This is useful for critical sections of code, where you do not want the process to be interrupted. Lastly, `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.
The OS knows how a program will respond to a particular signal because the processor's PCB has a pointer to a **vector of signal handlers**, where each entry corresponds to the handler function for that entry (and they are ordered logically by the signal number). A child process inherits the signal handlers of its parent process.
OS also provide fundamental services for us to control processes, mostly as kernel functions with their corresponding system calles. Some of these services are creating and terminating a process, suspending and resuming a process, changing the priority of a process, waiting for a process, check the process's status, interprocess communication, amongst others.
A **parent** process creates a new process, known as the **child** process. The child process can in turn create other processes, forming a **process hierarchy**. The first process created by the OS is known as the **init** process, which is the parent of all processes. When the parent process is destroyed or terminated, OS typically keep the child processes and allows them to continue execution. The steps followed to create a process are:

1. The OS allocates memory for the process — including space for the PCB.
2. The PCB is initialised and the process is added to the process table. The program counter and stack pointers are also set to appropriate values.
3. Other data structures are created, in particular for aspects such as memory, files, and accounting.
4. The process is set to ready and added to the ready queue.

**Check out:** Commands to create a process using C

- `fork()`: system call used to create a new process. The child process birthed by fork will be a copy of its parent, but instead of running from zero, it will be ran from the point at which the fork was called. The value returned by `fork()` is an integer. It will be below zero to represent a failed forking, zero to represent the child process, and the PID of the child for the parent. If–else statements can be used to handle the different cases. Forking is non–deterministic, since the scheduler is complex and we cannot make assumptions as to how it will determine the order of execution of the processes.

- `wait`: forces the parent to wait for the child to finish executing before continuing its execution, thus rendering the process deterministic. The parent can also use `waitpid()` to wait for a specific child process to finish executing. In order to retrieve the information from the *status* value returned by this function, we can use the macros `WIFEXITED()` (return true if child process terminates voluntarily), `WEXITSTATUS()` (return the exit code of the child; should only be called if `WIFEXITED==true`), `WIFSIGNALED()` (returns true if the process was terminated by a signal), `WTERMSIG()` (returns the signal number that caused the child process to terminate), amongst others.

- `exec()`: Runs a program different to the calling program; its inputs are an executable and some arguments. A successful call **never** returns to the calling program, as it replaces the calling program with the new program. There are several variants of this function, such as `execl()`, `execv()`, `execle()`, `execve()`, amongst others. The difference between them is the way in which the arguments are passed to the function.

- `getpid()`: returns the PID of the calling process.
- `getrusage()`: measures the resources used by the calling process or its children. It has two arguments, `int who` and `struct rusage *rusage`. If `who==RUSAGE_SELF`, the function will return information about the calling process itself; else, if `who==RUSAGE_CHILDREN`, it gives information on the resources of the children of the current process that **have been terminated and waited for**.
- wait4(): It has almost the same structure as `wait` (`wait4(pid_t pid, int *wstatus, int options, struct rusage *rusage)`), but it also incorporates usage statistics of the process being waited for.
- `waitid()`: Waits for a child process to change state. It can be used to instruct the system not to clear a zombie process and leave it in a waitable state instead. It has the following structure: `waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options)`. The `idtype` argument specifies the type of ID in `id`, which can be `P_PID` (wait for the child whose process ID matches `id`), `P_PGID` (wait for any child whose process group ID matches `id`), or `P_ALL` (wait for any child; this is the same as `wait()`). The `options` argument is a bit mask that specifies options for the call. The `infop` argument is a pointer to a `siginfo_t` structure that is used to return information about the child process.

The shell normally uses `fork()` and `exec()` to run programs. For instance, if you run `ls -l`, the shell will fork and then exec `ls`. The shell will then wait for the `ls` process to finish executing.

Similarly, a process is usually terminated by:

1. Process executes its last statement and asks the OS to delete it by using the `exit()` system call. This is known as **voluntary termination**.
2. If a termination signal or an error and fault condition is met, the OS can terminate the process. This is known as **involuntary termination**.
3. The termination status is returned from child to parent
4. The terminated process' resources are de–allocated

The reason why the process is first put in a **zombie state** instead of being terminated right away is that it allows the parent to get any necessary information from the child. The parent can use system calls, such as `waitpid()` to inform the operating system that it should completely remove the zombie child process. Processes can also be suspended, which implies they are temporarily deactivated so that they are not considered for processor scheduling. Suspension can be triggered by user request, parent process request, or by OS intervention in order to liberate memory for another ready process. A suspended process must be resumed by another process. The OS can also **block** a process, which implies that the process is waiting for an event to occur. The difference between these two actions is that blocking is triggered by internal activity of the process, whereas suspension comes from external entities.

# Virtualising the CPU / Process Control

The most basic technique to run a program is called **limited direct execution**. Direct execution consists of just running the program directly on the CPU, but with no limits this can expose the system to security and storage risks. This is solved by the introduction of **limits**, which guarantee the operating system to always be in control of the system.
The first reason why limitations are important is the performance of **restricted operations**, such as I/O requests. We cannot allow a process to simply do whatever restricted operation it wants, so the approach taken is to create two processor modes, **user mode** and **kernel mode**. The former is restricted in what operations it can do while the latter has unrestricted access. When a process wants to perform a restricted operation, it performs what we know as a **system call**.

### 1.0.2 System Calls

System Calls are the major mechanism that OS use to switch between user and kernel mode. Applications are mostly accessed via a high–level API (*Application Program Interface*) instead of directly invoking the specific system call. APIs are helpful because they provide a layer of abstraction between the application and the system call, which makes the application more portable — the caller does not need to know the details on how to invoke a system call under the particular OS and hardware (since different OS and hardware implement system calls differently). The system call interface invokes the corresponding call in kernel and returns the status and results of the system call back onto the application via the API. For the programmer, only knowledge of the standard API to invoke the system call is needed.
To execute a system call, a program must execute a **trap instruction**. This instruction jumps into the kernel, switches to kernel mode, and executes the system call. The system call returns control to the user program by using a **return-from-trap** instruction, which switches back to user mode. When executing trap instructions, the system must make sure to store the program counter and other registers so that it can return to the user program.

**Check out:** Trap Tables

The trap instruction knows which code to run inside of the operating system by using a **trap table**. This table is an array of pointers to the code that handles each system call. The index of the array is the system call number, and the pointer is the address of the code that handles the system call. The trap instruction uses the **system call number** to index into the trap table and get the address of the code that handles the system call. The trap instruction then jumps to that address, which switches the processor to kernel mode and starts executing the code that handles the system call.
When a computer boots up, it does so in kernel mode. This allows the kernel to set up a trap table and tell the hardware what code to run when certain exceptions occur. These programs are known as **trap handlers**.

The operating system runs on the CPU — but processes do so to! Through simple direct execution, this is a non–trivial issue: if the CPU is occupied, the operating system is not running, and if the operating system is not running, it cannot do anything at all. An approach to this is known as the **cooperative** approach, which consists of the operating system trusting the processes to be responsible and periodically return control so that the operating system can run another task — or voluntarily surrender control if the application does something illegal. This transferral of control is performed via

**system calls**. This is considered a naive approach, since it leaves the system exposed to malicious or faulty programs that just never return control. This is solved by the **non–cooperative** approach, which consists of the operating system **interrupting** the process and taking control back. This is done by the **timer interrupt**, which is a hardware device that interrupts the CPU at regular intervals (usually every couple milliseconds). When the timer interrupt occurs, the CPU switches to kernel mode and executes the **timer interrupt handler**, which is a piece of code that is part of the operating system. The timer interrupt handler then switches back to user mode and resumes the process that was interrupted. This timer can be turned off, but this is obviously a restricted operation that is not recommended in most scenarios. Interrupts are generally an extremely useful mechanism, as they allow the operating system to keep control.

### 1.0.3 Interrupt

Interrupts are **involuntary** releases of the CPU — a way to regain control of the CPU. Involuntary releases can be triggered by issues such as a process requesting I/O, a process executing an illegal instruction, a process dividing by zero, a process trying to access memory it is not allowed to access, and others, something known as **synchronous** events. Synchronous interrupts are also known as **exceptions**. Interrupts are also used to handle **asynchronous** events, such as a mouse click or a key press.

**Check out:** Exceptions

Exceptions can be put into three categories:

- Faults:
  The CPU detects an issue before executing an instruction. Some examples are page faults, protection faults, and alignment faults. Some faults can be corrected and the interrupted program can resume.

- Traps:
  The CPU detects an issue while executing an instruction. Some examples are illegal instruction, breakpoint, and overflow. After handling the trap, the interrupted is allowed to continue.

- Abort:
  The CPU detects an issue that cannot be corrected. Some examples are parity error and machine check. The interrupted program cannot continue.

When the operating system regains control, it has to decide whether to continue the current process, switch to another one, or do something else. The part of the operating system in charge of making such decisions is known as the **scheduler** (more on this topic is introduced in the next section). When the decision is to switch, the operating system executes a **context switch**.

### 1.0.4 Context Switch

A context switch is the process of saving the state of the current process and restoring the state of another process. The OS must save the state of the current process, which includes the program counter, registers, and other information. The OS must also restore the state of the next process, which includes the program counter, registers, and other information. Usually, the following steps are followed to perform a context switch successfully:

1. Save the context of the current process

2. Change the process's state in the PCB to blocked or ready
3. Move the process to the appropriate queue
4. Select a ready progress (according to the scheduling policy)
5. Load the context of the selected process
6. Update the PCB state of the selected process to running
7. Resume that process by surrendering control to it

Context switch presents an overhead, as it takes time to save and restore the state of the processes. This overhead is usually small, but it can be significant if the processes are short–lived. There are also some *induced* overheads: the newly scheduled process may not have its code and data in the cache, so it may take some time to load them into the cache. The newly scheduled process may also have a different memory mapping, so the memory management unit (MMU) may have to be reloaded.

**Check out:** Mode Switch vs Context Switch

Mode switch refers to when the CPU switches from user mode to kernel mode or vice versa. The process's context remains accessible (see: address space), and after exiting the kernel, the process may resume in user space. Context switch refers to when the CPU switches from one process to another. It involves the suspension of the current process, storing its context, retrieving the context of another process, restoring it to the CPU, and resuming execution of the process. The previous process's context is not accessible during a context switch.

The ensemble of these mechanisms—system calls, context switching, interrupts, trap handlers, and the scheduler—is what we know as **process control**. Together, they allow the operating system to keep control over the system when a different process is using the CPU.

**Check out:** Pipes and signals

You can set up a signal handler using the `signal()` function. This function t The signal handler function is called when the process receives the signal. Additionally, if instead of a function you input *SIG_IGN* or *SIG_DFL*, the signal will be ignored or the default action will be taken, respectively. The `sigaction()` function is similar to `signal()`, but it is more portable and has more options. In order to use it properly, you need to create a `struct sigaction` and pass it as an argument to the function. An example of this can be found in the appendix.
Pipes allow multiple processes to communicate, be it between parent and child or between children. Parent processes create the pipes and their children will inherit both ends. Pipes are bidirectional and must be created just before forking one or more child processes. You should always call `close()` to close the unused ends of the pipe and avoid any input or output errors. There are examples of this in the appendix.

# Process Scheduling

**Process Scheduling** refers to the activity of selecting the next process/thread to be serviced by the processor. The policy used by the operating system influences things like the quality of service offered to the users, the efficiency of the processor utilisation (process switching is expensive!), and the performance of the system (we aim to maximise the number of processes completed per unit time). There are three scheduling levels:

- High–level scheduling: it deals with creating a new process and control the number of processes in the system at one time. Controls whether or not new applications can start up. It also verifies whether there are enough resources for allowed processes to start. It is also known as **job scheduling**.
- Intermediate or medium–term scheduling: determined which processes will be allowed to compete for processors (deals with swapping processes in and out). It also responds to fluctuations in system load. It has the power to suspend or restart already running processes and relocate the resources they were using to improve the overall system performance. It is also known as **swapping**.
- Low–level or short–term scheduling: it determines which process the processor should run next.

**Check out:** Some Useful Terms:

- CPU–bound process: a process that when running tends to use all the processor time that is allocated to it. One example of it is machine learning applications. Such a program would go faster if the CPU were faster.
- I/O bound processes: a process that tends to use the processor only briefly before generating an I/O request and relinquishing the processor. An example of this is network computation. Such a process would be faster if the I/O subsystem were faster.
- Turnaround time: the amount of time to execute a particular process. $T_{turnaround} = T_{completition} - T_{arrival}$.
- Waiting time: the amount of time a process has been waiting in the ready queue. $T_{waiting} = T_{turnaround} - T_{burst}$.
- Response time: the amount of time it takes from when a job arrives in a system until it is first scheduled. $T_{response} = T_{firstrun} - T_{arrival}$.
- Workload: all the processes running in the system.

Initially, we will assume that every process runs for the same amount of time, arrives at the same time, runs until completion, does not perform any I/O, and has a known runtime. As we go on, we will drop these assumptions until we have a realistic scheduling discipline.
An essential element of any scheduling policy is the **scheduling metric**: a way to measure the performance of the policy. As defined above, our main metric will be **turnaround time**. That being said, let us look at some common scheduling algorithms:

- First In, First Out (FIFO):
  As the name implies, the processes will be scheduled in the order in which they arrive in the ready queue and will run until completion. The biggest issue with this algorithm is what is known as the **convoy effect**: when one of the processes is very long, all the other processes will have to wait for it to finish, even if they are short. This will make the average turnaround time increase significantly.
- Shortest Job First (SJF):
  Again, as the name implies, the processes will be scheduled in order of their burst time, from

shortest to longest. Although this solves the issue that FIFO faces when one of the processes is longer than the rest, it is not perfect — in real life, processes do not arrive at the same time. Thus, the process with the longest time might be the first one to arrive and in so doing it might again create a long turnaround time for the shorter processes that arrived afterward (and which were thus scheduled to run afterward).

- Shortest Time–to–Completion First (STCF):
  The previous two algorithms are **non–preemptive**, which means that each process must always run to completion once it has started. This does not have to be the case. STCF will be preemptive by, every time a new job arrives, determining which of all the jobs in the ready queue has the shortest time to completion and then scheduling that one. The problem with this approach arises when our metric is **response time**: the longer process will have to wait until the shorter processes are near completion to be scheduled even a single time. This creates problems when the user expects interactivity from the computer. We need better algorithms!

- Round–Robin (RR):
  This algorithm will run the process at the top of the run queue for a **time slice** (which must be a multiple of the timer–interrupt period) and then switch to the next job in the run queue. This will be done continuously until all the jobs are finished. If a job is not finished by the time its time slice is over, it will be preempted and put back at the end of the run queue. The length of the time–slice is a trade–off between the overhead of context switching and the responsiveness of the system. This is especially true considering context switching does not only involve the OS saving and restoring register values, but also flush and reload the states in the CPU caches, TLBs, etc.
  If we only care about responsiveness, RR is amazing. If the metric is turnaround time, it is terrible. This is because turnaround time is only concerned about **completion time** and RR stretches this out to the maximum.
  We say algorithms like RR are **fair** because CPU time gets divided equally amongst processes. There is always a trade–off between fairness and turnaround time

Having covered the basics of those algorithms, let us consider the case where **I/O operations** are involved. I/O is tricky, because when a program wants to perform such an operation, it is blocked and thus it is not using the CPU. Furthermore, once the I/O operation is done, the CPU has to move the process from blocked to ready and then schedule it. What should the scheduling policy be in those cases? What should the CPU do after the I/O request is sent, and should the program be run right away once it is ready? A way schedulers handle this is by treating each I/O operation as a job in and of itself. Thus, given a process $A$ that has one I/O operation, the scheduler would consider it as $A_0$ which runs from the process's beginnning until the I/O request is sent, and $A_1$, which runs from when the process is on the ready queue and scheduled again until completion. This allows for **overlap** between processes.
As you might have noticed, all of the aforementioned algorithms (with the exception of RR) have a fatal flaw: they assume the scheduler knows how long processes are so it schedules accordingly. In real life, schedulers are not omniscient. How can we deal with this then? Welcome in the **multilevel feedback queue** (MLFQ).
The idea behind MLFQs arose from the need to improve turnaround time (which, in essence, requires knowledge of the duration of processes so that shorter ones go first) and response time (which comes at odds with turnaround time in algorithms like RR). MLFQs tackle this by *learning* about the processes as they run. The idea is to have a set of queues, with a different priority assigned to each. As they enter, jobs that are ready will be put on a single queue, and the order in which they are run depends on the priority of such queue. If there is more than one job in the queue, the scheduler will use RR to

schedule the processes in that queue. These two are the basic rules of MLFQ, which can be summarised as:

1. If Priority(A)>Priority(B), A runs (B doesn't).
2. If Priority(A)=Priority(B), A & B run in RR.

Priorities are set based on *observed behaviour*: if a process keeps on relinquishing the CPU while performing I/O requests, its priority will be kept high since this is how interactive processes behave. If, on the other hand, a given process keeps hoarding the CPU, the scheduler will lower its priority. In so doing, MLFQ learns about the processes and schedules them accordingly.
The core issue at hand is then, how does the MLFQ scheduler know when to change the priority of a job (and which queue to allocate it to accordingly)? **Allotment**, the amount of time a process can spend at a given queue before the scheduler has to reassess its priority level, is the way the scheduler periodically moves jobs around. In short,

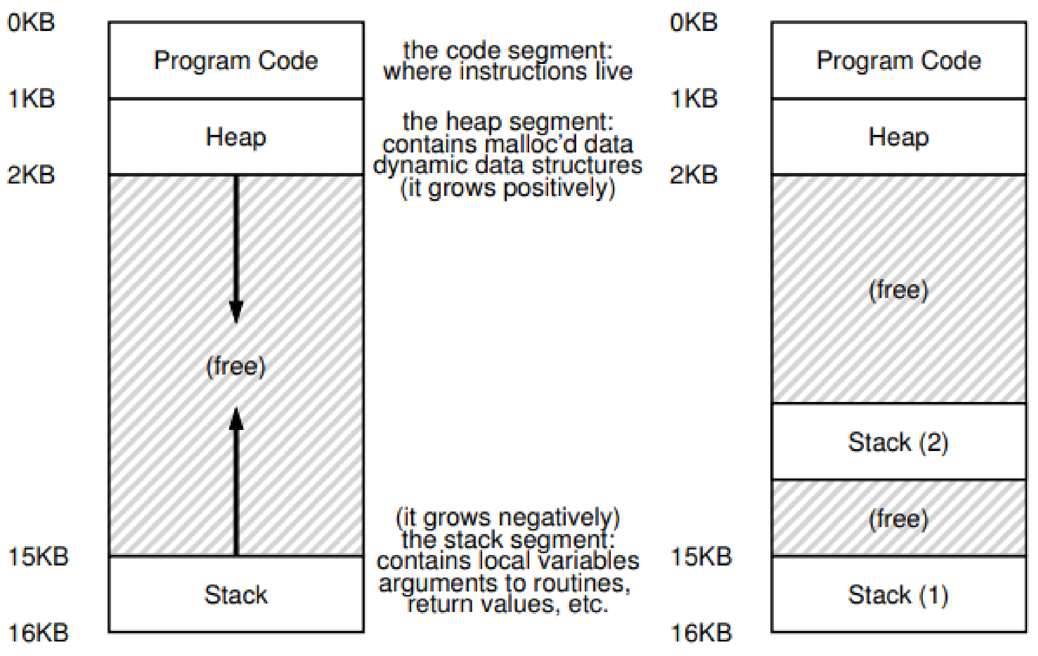
3. When the job first arrives, it is placed at the highest priority queue.
4. If the job eats up the entire allotment time, it is demoted to the next lower priority queue.
5. If the job performs any I/O request or another event that gives up the CPU before the allotment time is done, it stays at the same priority level (thus its allotment is reset). When such a request happens and the job is blocked, the processor will run the next job in the queue (or even go to the next queue if the current one is empty). Once the I/O request is done, the processor will put the job back in the queue it was in before and reset its allotment.

As has been explained so far, this approach has some issues: if there are many interactive processes, the CPU intensive processes in the lower priority queues might **starve** by getting no runtime. Furthermore, a process might get more of its fair share of CPU time by simply relinquishing the CPU before its allotment is over. This is known as **gamification**. And what happens in the case in which a process changes its behaviour as time passes (maybe going to an interactive part after a CPU intensive one)?
A popular way to tackle this issue is by periodically **boosting** all the processes to the topmost queue after some time $S$. This guarantees that no process will starve (although progress will admittedly not be the fastest for CPU intensive processes) and that if a process changes its behaviour, it will get the CPU time it deserves. The choice of an adequate $S$ is non–trivial: if it is too small, the scheduler will spend too much time boosting and not enough time scheduling; if it is too large, the scheduler will not be responsive enough.
What about the issue of **gaming**? We can prevent this by choosing to demote a job once it has used up its allotment time **regardless of how many times it gave up the CPU**. This is known as the **strict** MLFQ.
The parameters for MLFQ implementations vary a lot — the *Solaris* implementation, for instance, has up to 60 queues, where each queue has a different allotment time. Other schedulers have additional functions to the ones described above. For instance, some might accept user input to change the priority of a process, or even allow the user to **pin** a process to a given queue.

Lastly, another commonly–used type of scheduler is known as **proportional–share**. This type of scheduler is based on the idea of **fairness**: each process gets a **share** of the CPU, and the scheduler makes sure that each process gets its fair share. This is done by assigning each process a **weight** and then scheduling the processes in order of their weight. The weight of a process is usually determined by the user, but it can also be determined by the scheduler itself.
This type of scheduling is **non–deterministic**, because CPU time is allocated every so often via a *lottery*. The more tickets a process has, the more likely it is to win the lottery and get CPU time. The number of tickets a process has is proportional to its weight.

# Thread Abstraction & Concurrency

We have previously covered useful **abstractions** employed by the operating system, such as virtual CPUs and virtual memory. Another useful abstraction is that of a **thread**. In short, a thread is a **lightweight process** that can be scheduled independently. Threads are contained within processes, and each process can have multiple threads. Threads share the same address space, but each thread has its own stack and registers. This may lead into issues with stack–intensive threads, such as those that employ recursion, because the stack of one thread might end up clashing with that of another one (see Figure 1.8).



0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows positively)
(it grows negatively) the stack segment: contains local variables arguments to routines, return values, etc.
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)

Figure 1.8: Address space of a process with multiple threads, sourced from here

In a way, you could regard threads as **parallel processes** that share the same address space; however, since they do not share registers, the CPU has to perform a **context switch** every time it switches from running one to running the other.
The main reason behind the use of several threads is twofold: firstly, **parallelisation**. If you have multiple CPUs, you can perform different sections of a single program simultaneously as different threads. Furthermore, there is the issue of **blocking**. If a thread is blocked (e.g. due to I/O request), the CPU can switch to another thread and continue executing. Threads are more convenient than just having different programs because every thread shares the same address space, which makes information sharing a lot more seamless.

**Check out:** Creating a thread in C

The most common command to create a thread in C is `Pthread_create()` (included in the `<othread.h>` library). It has the following structure:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*start_routine) (void *), void *arg);
```

where `thread` is a pointer to a structure of type `pthread.t` (which is initialised the moment is passed into this function), `attr` is a pointer to a structure that specifies thread attributes (such as stack size or scheduling priority), `start_routine` is the function the thread will run, and `arg` is the argument to be passed to the function.
Another important function is `Pthread_join()`, which waits for a thread to terminate. It has the following structure:

```
int pthread_join(pthread_t thread, void **retval);
```

where `thread` is the thread to wait for and `retval` is the expected return value of the thread. Since this routine changes the value of `retval`, it must be passed by reference (i.e. as a pointer).

Which thread the CPU runs first is largely dependent on the scheduler policy, which is why sometimes it is hard to predict which thread will run first.
The fact that every thread shares the same address space can prove problematic. For instance, when changing the values of global variables, a thread will copy the value of the variable to one of its registers, operate on the register, and then copy back the value of that register onto the shared variable. What happens if there is a context switch in the middle of this? The other thread will operate on the unedited global variable — and when the context is switched back to the first thread, it will not see the global variable edited by the second process but rather the one it had copied onto its registers. These kinds of issues are known as **race conditions**, where the output of a program depends on the time at which context switches happen. These kinds of multithreaded programs are thus known as **indeterminate**.
The **critical section** of a program is that in which a shared variable is accessed. The way we can avoid indeterminacy is by making sure that only one thread can access the critical section at a time. This is known as **mutual exclusion**. Essentially, we do not want whichever operation the thread is performed to be interrupted by a context switch. This is known as **atomicity**. We achieve this desirable condition for our overall program by asking the hardware for instructions that allow us to build a set of **synchronisation primitives**, which are variable types that can only be modified through atomic operations.

**Check out:** More on threads in C

The `Pthread_join()` method is not always used. In some instances, such as with web servers, worker threads are created and run indefinitely, while the main thread is in charge of accepting requests and passing them over to the workers. In such cases, the main thread does not need to wait for the worker threads to finish executing.
Furthermore, very often we need to create more than just one thread — calling `Pthread_join()` so many times is not too practical. What we do instead is to use a **procedure call**, which is a function that is called when a thread is created. This function will then create the threads and join them. We will talk more about them later. First, let us have a quick intro to two other important sets of functions related to threads:

- Locks:
  Locks provide **mutual exclusion** to a critical section. The most basic pair of routines that achieve this are:

  ```
  pthread_mutex_lock(pthread_mutex_t *mutex);
  pthread_mutex_unlock(pthread_mutex_t *mutex);
  ```

If no thread is already holding the lock, the current threat trying to access the critical section will have no issue; else, it will be blocked until the lock is released.
Mutexes must always be initialised, usually through the `pthread_mutex_init()` function. This function has two arguments, `pthread_mutex_t *mutex` and `const pthread_mutexattr_t *attr`. The former is a pointer to the mutex to be initialised, and the latter is a pointer to a structure that specifies the mutex attributes (but leaving it to `NULL` usually does the job).

- Condition Variables:
  They are used to signal changes in shared variables amongst threads. They are usually used in conjunction with locks. The most basic pair of routines that achieve this are:

```
pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
pthread_cond_signal(pthread_cond_t *cond);
```

  The first function will block the thread until the condition variable `cond` is signalled. The second function will signal the condition variable `cond`, which will wake up one of the threads waiting for it.

# Synchronisation Tools

This section will go through the three main tools used in multithread programs: the lock (also known as mutex), the condition variable, and the semaphore.

### 1.0.5 Locks

Locks are generally used to wrap around critical sections and make them run essentially as if they were atomical. The most basic pair of routines that achieve this are:

```
lock_t mutex; //create lock variable
lock(&mutex); //acquire lock
//critical section
unlock(&mutex); //release lock
```

Locks are in essence a variable, so we need to declare them before we can use them. As a variable, it will store the state of the lock at a given time. Locks are binary, meaning they are either locked (one thread holds the lock and is in the critical section) or unlocked. If a thread tries to call `lock()` but another thread is already holding the lock, the function will not return until the other thread has called `unlock()`.

**Check out:** Locks in C

C refers to locks as *mutex*es, because they provide **mut**ual **ex**clusion. The functions in C that perform the same operations as the ones described above are:

```c
        pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

        Pthread_muxed_lock(&lock); //wrapper that exits on failure
        //critical section
        Pthread_mutex_unlock(&lock);

```

This being covered, let us talk about how locks are *implemented* — what is happening behind the scenes? The metrics used for evaluating the performance of any given lock mechanism are **fairness** (no thread starves when trying to claim the lock), **performance** (possible overheads when dealing with a single CPU, multiple CPUs, multiple or single contenders for the lock...), and **exclusivity** (whether or not the lock does its job of not allowing any other thread to access the code section it wraps).
The most primitive way of implementing an interrupt is by disabling hardware interrupts when the lock is acquired and re–enabling them when the lock is released. The main issue with this approach is how insecure it can be — any random program can just call `lock()` at the beginning of the program and monopolise the entire CPU! Furthermore, this approach might make the OS miss out on important flags, such as the fact a disk device already finished a read request. Lastly, this approach is not very scalable — if we have multiple CPUs, we would have to disable interrupts on all of them — not to mention there is no way for one CPU to know whether another CPU has already acquired the lock.
Nowadays, locks are implementing using an instruction that was included in hardware for the express purpose of creating locks: `test-and-set` instructions. This is a simple, C–based definition of what such an instruction does:

```c
    int TestAndSet( int *old_ptr, int new){
        int old = *old_ptr; //fetch old value at old pointer
        *old_ptr = new; //store new value in old pointer
        return old; //return old value
    }
```

This instruction works to implement a lock insofar that it is **atomic**; thus, when it is called, it changes the value of the lock and returns its previous one to the caller. This allows the caller to know whether or not the lock was already acquired — and to make sure that only one thread can acquire the lock at a time. This approach in particular is known as **spin lock** because the thread will keep on spinning until it acquires the lock. If we are working on a single CPU without preemption, this does not make sense as a mechanism — the program will never relinquish control!
Another primitive hardware–based approach is what is known as **compare–and–swap**: being given a pointer and an expected value, the instruction will compare the value at the pointer with that of the expected value. If they are the same, it will change the value at the pointer to a new value and return the old value. If they are not the same, it will return the value at the pointer. This is a simple, C–based definition of what such an instruction does:

```c
    int CompareAndSwap(int *ptr, int expected, int new){
        int actual = *ptr; //fetch actual value at pointer
        if(actual == expected) //if actual value is the expected one
            *ptr = new; //store new value in pointer
        return actual; //return actual value
    }
```

We can build a lock by then

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

This type of lock however suffers from the same performance issues as the previous one. The performance issues that spinning brings can be significant; if $N$ threads are contending for a lock, $N-1$ time slices will be wasted because that amount of threads will just be waiting until the lock is released.
The first possible approach to this issue is to make threads **give up the CPU** to another thread every time they are going to spin. Although this is efficient when the number of threads is low, in real programs with hundreds or more threads, the cost from the constant context switching can be high. The most effective approach is to **sleep** instead of yielding. In short, this mechanism consists of putting the thread to sleep until the lock is released. This is done by the `sleep()` function, which takes as an argument the address of the lock and puts the thread to sleep until the lock is released. The core of this approach is a queue that will store the threads that are waiting for the lock to be released. When the lock is released, the thread at the top of the queue will be woken up and given the lock. Here is a sample implementation, where the function `park()` puts the thread to sleep and `unpark()` wakes it up:

```c
    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); //spin
        if (m->flag ==0){
            m->flag =1; //lock has been acquired
            m->guard = 0; //guard represents the lock
        } else{
            queue_add(m_.q, gettid()); //add thread to queue
            m->guard = 0; //release guard
            park(m->q); //put thread to sleep
        }
    }
    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 cause no one wants implement
        else
        unpark(queue_remove(m->q)); //hold lock for next thread
        m->guard = 0; //release guard
    }
```

### 1.0.6 Condition Variables

Very often, a thread needs to check whether or not a condition is true before continuing its execution (e.g. whether a child thread has completed). The most efficient way to achieve this is by putting the parent thread to sleep while the child finalises its execution, and this is achieved through **condition variables**. In a nutshell, condition variables are explicit **queues** that threads put themselves in when they are waiting for a condition to be true. These kinds of variables (declared via `pthread_cond_t c`) have two operations associated with themselves: `wait()` and `signal()`. The first one is executed when the thread wants to put itself to sleep, whereas the second one is executed when something has changed in a thread and it wants to wake up a different thread. These two calls have the following C syntax:

```
␣␣␣␣pthread_cond_wait(pthread_cond_t␣*c,␣pthread_mutex_t␣*m);
␣␣␣␣pthread_cond_signal(pthread_cond_t␣*c);
```

Note that `pthread_cond_wait()` also has a mutex as an argument. This function expects the lock to be held by the thread that is made to wait; it will then release the lock and put the thread to sleep until the condition variable is signalled. Once the thread is woken up, it will reacquire the lock and continue its execution. This is done to prevent race situations.

**Check out:** Tips for a proper implementation

To make sure you do not run into errors, you should always follow these rules when using condition variables:

1. Have a variable that allows the parent to know the completion of the child
2. Wrap the condition variable in a lock (every time you call wait or signal, you must hold the lock)
3. Run join so the parent waits for its child (if necessary)

Having introduced the basic idea behind condition variables, let us look at some of the most common synchronisation problems we might run into

**The Producer/Consumer Problem**

This problem is one of the most common ones in multithreaded programming. It consists of a producer that produces items and a consumer that consumes them. The producer and consumer share a buffer of size $N$ that can hold up to $N$ items. The producer will produce items and put them in the buffer, whereas the consumer will consume items from the buffer. The producer and consumer must be synchronised so that the consumer does not try to consume an item that is not there yet, and so that the producer does not try to produce an item when the buffer is full.

The solution to this issue is to use **two** condition variables to properly signal which type of thread should wake up when the state of the system changes. A sample code showing this solution is as follows:

```c
    /* producer threads wait on the condition empty and signals fills
    consumer threads wait on the condition fills and signals empty */
    int buffer[MAX];
    int fill_ptr = 0;
    int use_ptr = 0;
    int count = 0;

    void put(int value){
        buffer[fill_ptr] = value;
        fill_ptr = (fill_ptr + 1) % MAX;
        count++;
    }

    int get(){
        int tmp = buffer[use_ptr];
        use_ptr = (use_ptr + 1) % MAX;
        count--;
        return tmp;
    }

    //Synchronisation

    cond_t empty, fill;
    mutex_t mutex;

    void *producer(void *arg){
        int i;
        for (i =0; i < loops; i++){
            Pthread_mutex_lock(&mutex);
            while(count == MAX)
                Pthread_cont_wait(&empty,&mutex);
            put(i);
            Pthread_cond_signal(&fill);
            Pthread_mutex_unlock(&mutex);
        }
    }

    void *consumer(void *arg){
        int i;
        for (i = 0; i < loops; i++){
            Pthread_mutex_lock(&mutex);
            while(count == 0)
                Pthread_cond_wait(&fill, &mutex);
            int tmp = get();
            Pthread_cond_signal(&empty);
            Pthread_mutex_unlock(&mutex);
            printf("%d\n", tmp);
        }
    }
```

### 1.0.7 Semaphores

# Sample Programs

**Working with Processes:**
Creating a child process with `fork()`

```
    int main(){
        pid_t who;
        int var = 999;

        printf("Process (%d) starts up\n", int(getpid()));
        who = fork();
        if(who == 0){
            sleep(1); //force child to sleep first
            printf("Variable has value (%d)\n", var);//999
            var = 111;
            printf("Variable has value (%d)\n", var);//111
        } else{
            printf("Variable has value (%d)\n", var);//999
            var = 222;
            sleep(3); //force parent to sleep
            printf("Variable has value (%d)\n", var);//222
        }
        printf("Process (%d) exits\n", int(getpid()));
        return 0;
    }
    /*will print the following:
    Process (1234) starts up
    Variable has value (999)
    Variable has value (999)
    Variable has value (111)
    Process (1235) exits
    Variable has value (222)
    Process (1234) exits
    */
```

Executing a program with `exec()`

```
int main(){
    char *a[3] = {(char *)"ls", (char *)"-l", NULL};
    printf("Execvp: press <enter> to execute ls -l\n");
    getchar();
    if (execvp(a[0], a) == -1){
        perror("exec failed");}
    printf("This line should never be printed\n");
    return 0;
}
```

Waiting for a child using `waitid()`

```
int main(){
    pid_t pid = fork();

    if (pid <0){
        printf("Fork failed\n");
        exit(-1);
    }else if (pid==0){
        printf("I am baby with pid (%d)\n", int(getpid()));
        printf("Use the kill command to kill me\n");
        while(1);
    } else{
        siginfo_t info;
        int status;
        int ret = waitid(P_PID, 0, &info, WNOWAIT | WEXITED);
        if (!ret){
            printf("Child (%d) has terminated\n", info.si_pid);
            printf("Child's exit status is (%d)\n",(int) info.si_status);
            printf("Its signal status is (%d)\n", WTERMSIG(status));
        } else{
            printf("waitid failed\n");
        }
    }
    return 0;
}
```

Read important data from the proc file system

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

int main() {
    pid_t myid;
    char str[50];
    FILE * file;
```

```
        int z;
        unsigned long long i, x;
        unsigned long h, ut, st;

        /* just run for some duration */
        for (i=0; i<1073741824; i += 2) {
            x *= i;
            if (x > 8589934589)
                x = 1;
        }

        myid = getpid();

    printf("My process ID is: %d\n", myid);
    printf("Please use another terminal to run this command to access the path: /proc
/%d/stat -- 'cat /proc/%d/stat'\n", myid, myid);
    printf("Press enter to continue after you obtained the information\n");
    getchar();

        /* get my own procss statistics */
        sprintf(str, "/proc/%d/stat", (int)myid);
        file = fopen(str, "r");
        if (file == NULL) {
            printf("Error in open my proc file\n");
            exit(0);
        }

        fscanf(file, "%d %s %c %d %d %d %d %d %u %lu %lu %lu %lu %lu %lu", &z, str, &
str[0], &z, &z, &z, &z, &z,
            (unsigned *)&z, &h, &h, &h, &h, &ut, &st);
        fclose(file);

        printf("The number of clock ticks per second is: %ld\n", sysconf(_SC_CLK_TCK)
);

        printf("User time: %lf s\n", ut*1.0f/sysconf(_SC_CLK_TCK));
        printf("In system time: %lf s\n", st*1.0f/sysconf(_SC_CLK_TCK));
    printf("The process is in state: %c\n", str[0]);

        return 0;
    }
```

**Interprocess Communication**:
Interact with signals using `signal()`

```
    void sigint_handler(int signum) {
        printf("\nA signal SIGINT is caught\n\n");
    }

    int main() {
        printf("SIGINT can be CAUGHT once in the coming 10 seconds.\n");
        printf("Press Ctrl-c to try\n");
        signal(SIGINT, sigint_handler);

        sleep(10);

        printf("SIGINT is ignored in the coming 10 seconds.\n");
        printf("Press Ctrl-c to try\n");
        signal(SIGINT, SIG_IGN);

        sleep(10);
```

```c

        printf("\n\nSIGINT is set to default action.\n");
        printf("Press Ctrl-c to kill me\n");
        signal(SIGINT, SIG_DFL);

        while (1) {
            sleep(10);
        }
    }
```

Interact with signals using `sigaction()`

```c
    int n=0;
    void sigint_handler2(int signum, siginfo_t *sig, void *v) {
        printf("signal %d (from %d) is caught for %d times\n", signum, sig->si_pid,
    ++n);
        if (sig->si_code == SI_USER) printf("It is sent by a user (kill command)\n");
        else if (sig->si_code == SI_KERNEL) printf("It is sent by the kernel\n");

        if (n == 3) {
            exit(0);
        }
    }

    int main() {
        struct sigaction sa;

        /* use sigaction to install a signal handler named sigint_handler1 */
        sigaction(SIGINT, NULL, &sa);
        sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = sigint_handler2;
        sigaction(SIGINT, &sa, NULL);

        printf("My process ID is %d\n", (int)getpid());
        printf("Press Ctrl-c 3 times to kill me\n");;
        while (1) {
            sleep(1);
        }
    }
```

Piping:

```c
    int main()
    {
        // We use two pipes
        // First pipe to send input string from parent
        // Second pipe to send concatenated string from child

        int fd1[2]; // Used to store two ends of first pipe
        int fd2[2]; // Used to store two ends of second pipe

        char fixed_str[] = "forgeeks.org";
        char input_str[100];
        pid_t p;

        if (pipe(fd1) == -1) {
            fprintf(stderr, "Pipe Failed");
            return 1;
        }
        if (pipe(fd2) == -1) {
            fprintf(stderr, "Pipe Failed");
            return 1;
        }

        scanf("%s", input_str);
        p = fork();

        if (p < 0) {
            fprintf(stderr, "fork Failed");
            return 1;
        }

        // Parent process
        else if (p > 0) {
            char concat_str[100];

            close(fd1[0]); // Close reading end of first pipe

            // Write input string and close writing end of first
            // pipe.
            write(fd1[1], input_str, strlen(input_str) + 1);
            close(fd1[1]);

            // Wait for child to send a string
            wait(NULL);

            close(fd2[1]); // Close writing end of second pipe

            // Read string from child, print it and close
            // reading end.
            read(fd2[0], concat_str, 100);
            printf("Concatenated string %s\n", concat_str);
            close(fd2[0]);
        }

        // child process
        else {
            close(fd1[1]); // Close writing end of first pipe

            // Read a string using first pipe
            char concat_str[100];
            read(fd1[0], concat_str, 100);

            // Concatenate a fixed string with it
            int k = strlen(concat_str);
            int i;
            for (i = 0; i < strlen(fixed_str); i++)
                concat_str[k++] = fixed_str[i];

            concat_str[k] = '\0'; // string ends with '\0'

            // Close both reading ends
            close(fd1[0]);
            close(fd2[0]);

            // Write concatenated string and close writing end
            write(fd2[1], concat_str, strlen(concat_str) + 1);
            close(fd2[1]);

            exit(0);
        }
    }
```

**Threads**:
Creating different threads in one program

```c
typedef struct { int a; int b; } myarg_t;
typedef struct { int x; int y; } myret_t;

void *mythread(void *arg){
    myret_t *rvals = Malloc(sizeof(myret_t));
    rvals->x = 1;
    rvals->y = 2;
    return (void *) rvals;
}

int main(int argc, char*argv[]){
    pthread_t p;
    myret_t *rvals;
    myarg_t args = { 10, 20 };
    Pthread_create(&p, NULL, mythread, &args);
    Pthread_join(p, (void **) &rvals);
    printf("returned %d %d\n", rvals->x, rvals->y);
    free(rvals);
    return 0;
}
```

Self-referencing in a thread:

```c
void *func1 (void *arg){
int x = *((int*)arg);
printf("The integer passed in is %d\n",x);
printf("Thread: Process id is %d\n", (int)getpid());
printf("Thread: The thread id is %d\n", (int) pthread_self());
printf("Thread: The thread id is %d (Linux-specific)\n", (int)syscall(SYS_gettid)
);
pthread_exit(NULL);
}

int main() {
pthread_t thread_id;
int x = 88;

printf("Main process: Process id is %d\n", (int)getpid());

pthread_create(&thread_id, NULL, func1,(void*)&x);
pthread_join(thread_id, NULL);
return 0;
}
```

Cancelling another thread:

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

void *func1 (void *arg){
printf("func1 set the cancellation state to ignore\n");
pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
int remain = sleep(10);
while (remain) { remain = sleep(remain); }
printf("func1 set the cancellation state to enable with immediate action\n");
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
while (1) sleep(1);

}

void *func2 (void *arg){
int i;
pthread_t tid = *(pthread_t *)arg;
for (i=0; i<5; i++) {
    printf("func2 sends a cancellation request to func1\n");
    pthread_cancel(tid);
    sleep(2);
}
pthread_exit(NULL);
}

int main() {
pthread_t thread_id1, thread_id2;

pthread_create(&thread_id1, NULL, func1, NULL);
sleep(2);
pthread_create(&thread_id2, NULL, func2,(void*)&thread_id1);

pthread_join(thread_id1, NULL);
printf("Master: func1 has completed\n");
pthread_join(thread_id2, NULL);
printf("Master: func2 has completed\n");
return 0;
}
```

Using locks:

```c
#include <stdio.h>
#include <pthread.h>

//Store the subrange of computation.
struct Range {
int up;
int down;
};

#define RANGE 10000

pthread_mutex_t m_mutex = PTHREAD_MUTEX_INITIALIZER;
int  sum = 0;

void *functionC(void* var)
{
    int i;
    int SubSum=0;
    //get the assigned range
    struct Range *SubRange = (struct Range*)var;
    //perform summation of this range
    for (i=SubRange->down;i<=SubRange->up;i++)
        SubSum+=i;
    //add to total
    printf("SubSum value(%d, %d): %d\n", SubRange->down, SubRange->up, SubSum);
    pthread_mutex_lock( &m_mutex );
    sum+=SubSum;
    pthread_mutex_unlock( &m_mutex );
    pthread_exit(NULL);
}

```

```c
int main()
{
    int rc;
    pthread_t thread1, thread2, thread3, thread4;
    struct Range SubRange_1, SubRange_2, SubRange_3, SubRange_4;
    int range = RANGE/4;

    /* Create independent threads each of which will execute functionC */
    SubRange_1.up = range;
    SubRange_1.down = 1;
    if( (rc=pthread_create(&thread1, NULL, &functionC, (void *)&SubRange_1)) )
    printf("Thread creation failed: %d\n", rc);

    SubRange_2.up = 2*range;
    SubRange_2.down = 1+range;
    if( (rc=pthread_create(&thread2, NULL, &functionC, (void *)&SubRange_2 )) )
        printf("Thread creation failed: %d\n", rc);

    SubRange_3.up = 3*range;
    SubRange_3.down = 1+2*range;
    if( (rc=pthread_create(&thread3, NULL, &functionC, (void *)&SubRange_3)) )
    printf("Thread creation failed: %d\n", rc);

    SubRange_4.up = RANGE;
    SubRange_4.down = 1+3*range;
    if( (rc=pthread_create(&thread4, NULL, &functionC, (void *)&SubRange_4 )) )
        printf("Thread creation failed: %d\n", rc);

    /* Wait till threads are complete before main continues. */
    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);
    pthread_join( thread3, NULL);
    pthread_join( thread4, NULL);

    printf("Sum of whole range: %d\n", sum);

    //Release mutex;
    pthread_mutex_destroy(&m_mutex);

    return 0;
}
```

**Synchronisation Tools**:
Using condition variables in a program where parent waits for child:

```c
int done = 0;
//if we did not have variable done, parent has no idea of knowing child has
finished if thr_join runs after child has finished
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c = PTHREAD_COND_INITIALIZER;

void thr_exit(){
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
void *child(void *arg){
    printf("I am child\n");
    thr_exit();
    return NULL;
    }
    void thr_join(){
        Pthread_mutex_lock(&m);
        while(done == 0)
            Pthread_cond_wait(&c, &m);
        Pthread_mutex_unlock(&m);
    }
    /*
    Without locks, if parent interrupted while checking value
     of done (before call wait),
    CPU goes to child and child calls thr_exit, parent will never wake up
    */
    int main(int argc, char *argv[]){
        printf("parent: begin\n");
        pthread_t p;
        Pthread_create(&p, NULL, child, NULL);
        /*if parent runs first, run thr_join
        the effect of this function is parent acquiring the lock, putting itself to
   sleep by calling wait (hence releasing the lock), letting the child work, and
   then wake up with lock */
        thr_join(); //block until child calls thr_exit()
        printf("parent: end\n");
        return 0;
    }
```

Conditional Variables

```
        /*
   ****************************************************************************
    * DESCRIPTION:
    *   Example code for using Pthreads condition variables.  The main thread
    *   creates three threads.  Two of those threads increment a "Count" variable,
    *   while the third thread watches the value of "Count".  When "Count"
    *   reaches a predefined limit, the waiting thread is signaled by one of the
    *   incrementing threads.
    * SOURCE: Adapted from example code in "Pthreads Programming", B. Nichols
    *   et al. O'Reilly and Associates.
    ******************************************************************************/
    #include <stdio.h>
    #include <stdlib.h>
    #include <pthread.h>

    #define NUM_THREADS  3
    #define TCOUNT 8
    #define COUNT_LIMIT 10

    pthread_mutex_t count_mutex;
    pthread_cond_t count_threshold_cv;

    int  thread_ids[3] = {0,1,2};
    int  Count=0;

    void *inc_count(void *idp)
    {
        int j,i;
        double result=0.0;
        int *my_id = (int *)idp;

        for (i=0; i < TCOUNT; i++) {
            pthread_mutex_lock(&count_mutex);
            Count++;
```

```c

        /*
        Check the value of count and signal waiting thread when condition is
        reached.  Note that this occurs while mutex is locked.
        */
        if (Count == COUNT_LIMIT) {
            pthread_cond_signal(&count_threshold_cv);
            printf("inc_count(): thread %d, count = %d Threshold reached.\n", *my_id, Count);
        }
        printf("inc_count(): thread %d, count = %d, unlocking mutex\n", *my_id, Count);

        pthread_mutex_unlock(&count_mutex);

        /* Do some work so threads can alternate on mutex lock */
        for (j=0; j < 10000; j++)
            result = result + (double)random();
    }
    pthread_exit(NULL);
}

void *watch_count(void *idp)
{
    int *my_id = (int *)idp;

    printf("Starting watch_count(): thread %d\n", *my_id);

    /*
    Lock mutex and wait for signal.  Note that the pthread_cond_wait routine
    will automatically and atomically unlock mutex while it waits.
    Also, note that if COUNT_LIMIT is reached before this routine is run by
    the waiting thread, the loop will be skipped to prevent pthread_cond_wait
    from never returning.
    */
    pthread_mutex_lock(&count_mutex);
    if (Count < COUNT_LIMIT) {
        pthread_cond_wait(&count_threshold_cv, &count_mutex);
        printf("watch_count(): thread %d Condition signal received.\n", *my_id);
    }
    pthread_mutex_unlock(&count_mutex);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    int i;
    pthread_t threads[3];

    /* Initialize mutex and condition variable objects */
    pthread_mutex_init(&count_mutex, NULL);
    pthread_cond_init (&count_threshold_cv, NULL);

    /* Create the threads */
    pthread_create(&threads[2], NULL, watch_count, (void *)&thread_ids[2]);
    pthread_create(&threads[0], NULL, inc_count, (void *)&thread_ids[0]);
    pthread_create(&threads[1], NULL, inc_count, (void *)&thread_ids[1]);
    //pthread_create(&threads[2], NULL, watch_count, (void *)&thread_ids[2]);

    /* Wait for all threads to complete */
    for (i = 0; i < NUM_THREADS; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Main(): Waited on %d threads. Done.\n", NUM_THREADS);

    /* Clean up and exit */
    pthread_mutex_destroy(&count_mutex);
    pthread_cond_destroy(&count_threshold_cv);
    pthread_exit (NULL);

}
```

Semaphore

```c
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>

#define NITER 1000000

int Count=0;

sem_t binarySem;    /* Use as a mutex */

void * ThreadAdd(void * a)
{
    int i, tmp;

    for(i = 0; i < NITER; i++)
    {
        sem_wait(&binarySem);
        tmp = ++Count;
        sem_post(&binarySem);
        if (!(tmp % 100000))
            printf("\nThread %d\tcount value = %d", *(int *)a, tmp);
    }
    pthread_exit(NULL);
}

int main(int argc, char * argv[])
{
    pthread_t tid1, tid2;
    int id1=1, id2=2;

    if (sem_init(&binarySem, 0, 1) == -1)
    {
        printf("Error initialize the semaphore\n");
        exit(1);
    }

    if(pthread_create(&tid1, NULL, ThreadAdd, (void *)&id1))
    {
        printf("ERROR creating thread 1\n");
        exit(1);
    }

    if(pthread_create(&tid2, NULL, ThreadAdd, (void *)&id2))
    {
        printf("ERROR creating thread 2\n");
        exit(1);
        }

        if(pthread_join(tid1, NULL))  /* wait for the thread 1 to finish */
        {
            printf("ERROR joining thread\n");
            exit(1);
        }

        if(pthread_join(tid2, NULL))        /* wait for the thread 2 to finish */
        {
            printf("ERROR joining thread\n");
            exit(1);
        }

        if (Count < 2 * NITER)
            printf("\nBOOM! count is %d, should be %d.\n", Count, 2*NITER);
        else
            printf("\nOK! count is %d\n", Count);

        sem_destroy(&binarySem);

        pthread_exit(NULL);
    }
```