Processes and Threads

50.005 Computer System Engineering

Materials taken from SGG: Ch. 3.1-3.3, 3.4.1, 3.6.1, 4.1-4.2, 4.3.1, 4.4, 4.5.1-4.5.2

Natalie Agus
INFORMATION SYSTEMS TECHNOLOGY AND DESIGN

Basics of Processes in Computer Systems

The Concept of Process vs Program

→ **A process is formally defined as a program in execution.** A *program* is **not** a *process.* Process is an **active**, **dynamic** entity -- i.e: it changes state overtime during execution, while a program is a **passive, static** entity.

→ **A process is much more than just a *program code***, formally known as the **text section.** A single process includes all of the following information:

1. The code / program (**text** section)
2. Value of the Program Counter (**PC**)
3. **Contents** of the processor's **registers**
4. **Dedicated**[1] **address** space (block of location) in memory
5. **Stack** (temporary data such as function parameters, return address, and local variables, grows **downwards**),
6. **Data** (allocated memory during compile time such as global and static variables that have predefined values)
7. **Heap** (dynamically allocated memory -- typically by calling `malloc` in C -- during process runtime, grows **upwards**)[2]

→ **The same program can be run *n* times to create *n* processes simultaneously.** For example, *separate tabs* on some web browser are created as separate processes. The program for all tabs are the same: which is the part of the web browser code itself.

→ **A program becomes a process when an executable (binary) file is loaded into memory** (either by double clicking them or executing them from the command line)

→ **A process couples two abstractions: concurrency and protection.** Each process runs in different address space and sees itself as running in a virtual machine -- unaware of the presence of other processes in the machine. Multiple processes execution in a single machine is concurrent, managed by the kernel scheduler.

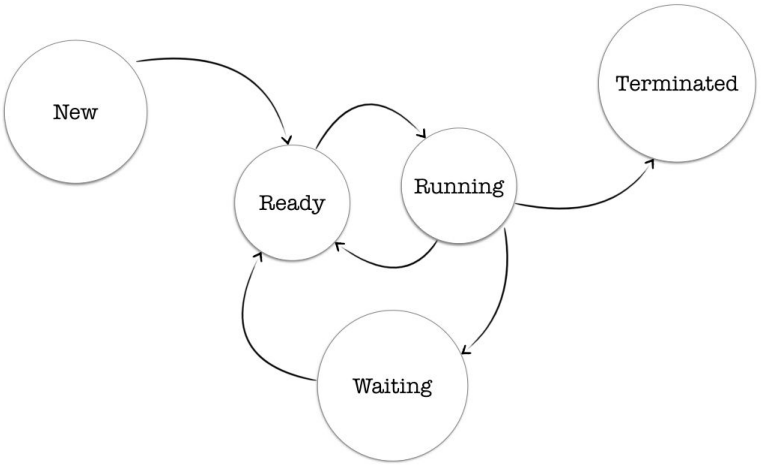
Process Scheduling State

As a process executes, it changes its scheduling *state*: the current activity of the process. In general these states are:

[1] Because **each process is isolated from one another and runs in different address space (forming virtual machines)**
[2] Heap and stack grows in the opposite direction so that it maximises the space that both can have and minimises the chances of overlapping, since we do not know how much they can dynamically grow during runtime. If the heap / stack grows too much during runtime, we are faced with **stack/heap overflow** error. If there's a heap overflow, `malloc` will return a `NULL` pointer.

1. **New**: The process is being created.
2. **Running**: Instructions are being executed.
3. **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
4. **Ready**: The process is waiting to be assigned to a processor
5. **Terminated**: The process has finished execution.

The figure below shows the scheduling state transition diagram of a typical process:



Process Control Block

Each process is represented in the operating system by a particular data structure called the process control block (PCB) — also called a task control block.

Each PCB is an entry in the system-wide **process table**, a bigger data structure that logically contains a PCB for all of the current processes[3] in the system. In short, one can say that **the process table is an array of PCBs.**

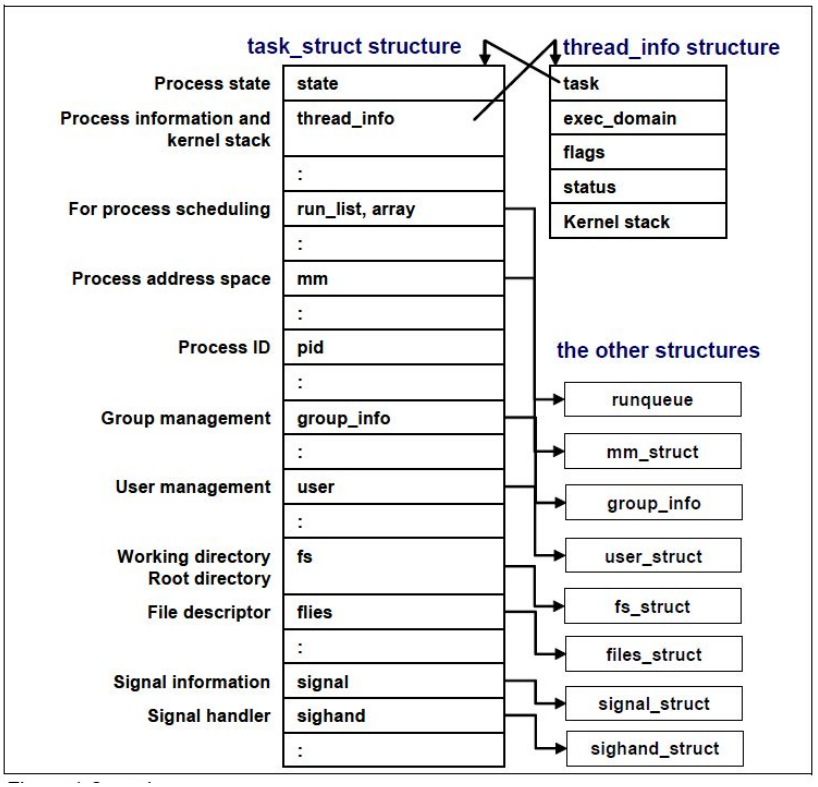
The PCB contains many pieces of information associated with a specific process. These information are updated each time **when a process is interrupted**:

1. **Process state:** any of the state of the process -- new, ready, running, waiting, terminated
2. **Program counter:** the address of the *next instruction* for this process
3. **CPU registers:** the contents of the registers in the CPU when an interrupt occurs, including stack pointer, exception pointer, stack base, linkage pointer, etc. **These contents are saved each time to allow the process to be continued correctly afterward**.
4. **CPU-scheduling information:** access priority, pointers to scheduling queues, and any other scheduling parameters

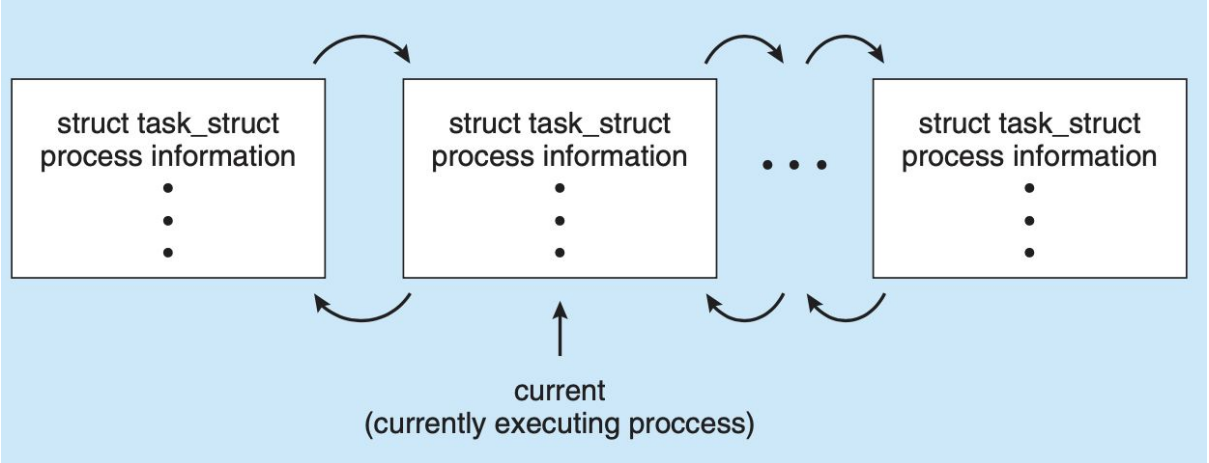
[3] You can list all processes that are currently running in your UNIX-based system by typing `ps aux` in the command line.

5. **Memory-management information:** page tables, MMU-related information, memory limits
6. **Accounting information:** amount of CPU and real time used, time limits, account numbers, ***process id***
7. **I/O status information:** the list of I/O devices allocated to the process, a list of open files

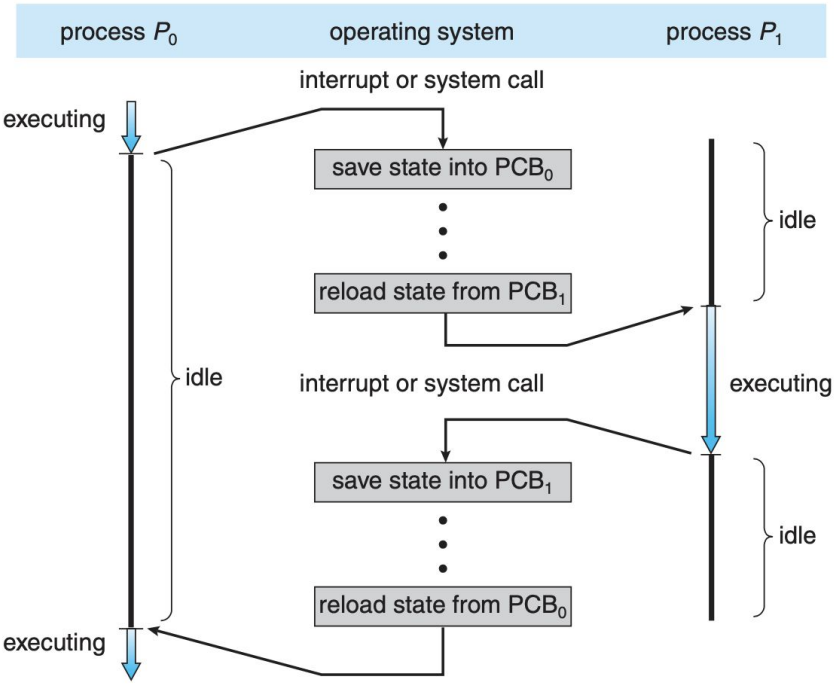
In linux system, the PCB is represented by the C structure called `task_struct`. The diagram below illustrates some of the contents in the structure:



Within the Linux kernel, all active processes are represented using a doubly linked list of `task_struct`. The kernel maintains a pointer —current — to the process currently executing on the system, as shown below:



When a CPU switches execution between one process to another, the operating system has to help all of the process states onto its corresponding PCB, and load the new process' PCB before resuming them.



Rapid Context Switching and Timesharing

Context switch: the mechanism of saving the states of the current process and restoring (loading) the state of a different process when switching the CPU to execute another process.

Recall: timesharing requires interactivity, and this is done by performing rapid context switching between execution of multiple programs in the computer system.

Why context switching happens:
When an **interrupt** occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. **The context is represented in the PCB of the process.**

Why this is beneficial:
1. To give the illusion of concurrency in uniprocessor system
2. Hence improving system responsiveness and intractability, ultimately allowing *timesharing* (users can interact with each program when it is running).
3. To support *multiprogramming*: optimise CPU usage, we cannot just let one single program to run all the time, especially when that program blocks execution when waiting for I/O

Cons: Context-switch time is **pure overhead**, because the system does no useful work while switching. To minimise downtime due to overhead, context-switch times are highly dependent on hardware support -- some hardware supports ***rapid context switching*** .

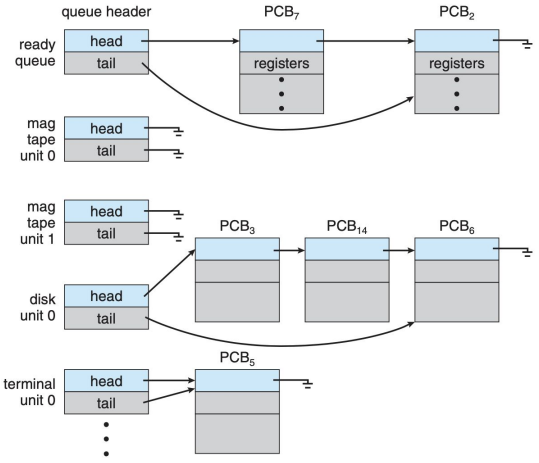
Process Scheduling

Motivation:
- The **objective** of ***multiprogramming*** is to have some process running at all times, to maximize CPU utilization.
- The **objective** of ***time sharing*** is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.
    - For a single-processor system, there will never be more than one running process.
    - If there are more processes, the rest will have to wait in a **queue** until the CPU is free and can be rescheduled.

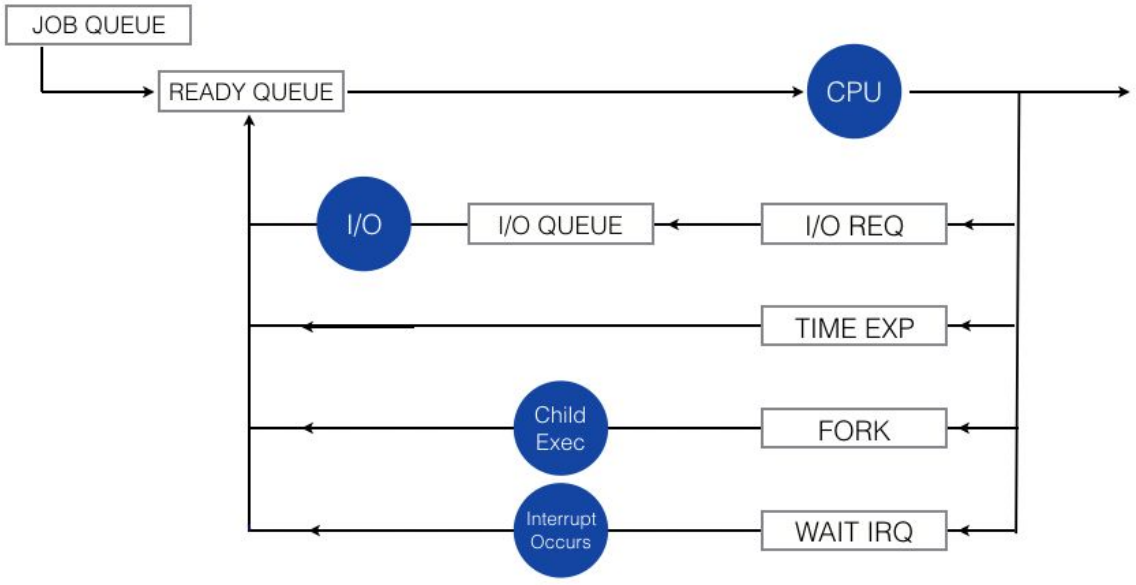
Process scheduling queues:

1. **Job queue** – set of all processes in the system (can be in both main memory and swap space of disk)
2. **Ready queue** – set of all processes residing in main memory, ready and waiting to execute (queueing for CPU)
3. **Device queues** – set of processes waiting for an I/O device **(one queue for each device)**

Each queue contains the pointer to the corresponding PCBs that are waiting for the resource. The diagram below shows a system with a ready queue, and four device queues:



A common representation of process scheduling is a **queueing diagram** as shown below:



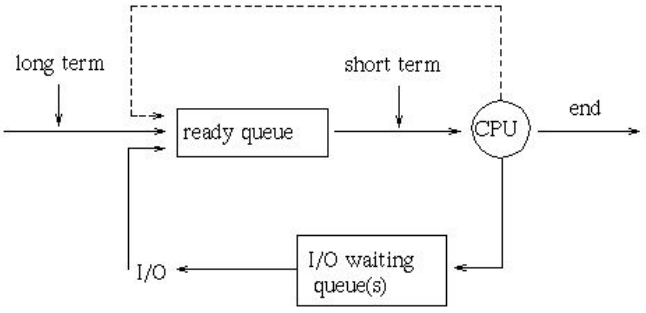
Explanation:

- **All types of queue present :** *job* queue, *ready* queue and a set of *device* queues (I/O queue, I/O Req, time exp, fork queue, and wait irq).
- Rectangular boxes represent queues, circles represent **resources** serving the queue

A new process is initially put in the ready queue. It waits there until it is selected for execution, or dispatched. Once the process is allocated the CPU, a few things might happen afterwards (that causes the process to leave the CPU):

- If the process issue an I/O request, it will be placed onto the I/O queue
- If the process forks (create new process), it will queue for it until fork is serviced by the kernel
- The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

Long term and short term scheduler manages each queue accordingly as shown:



Operations on Processes

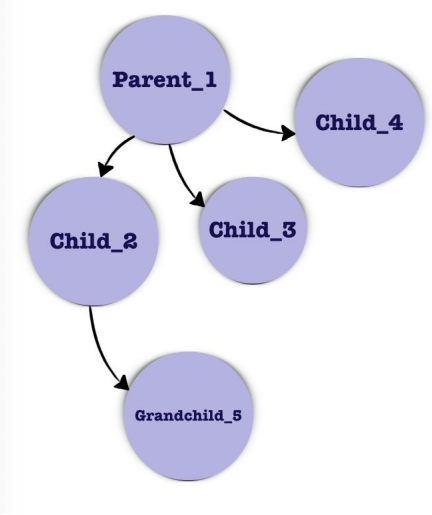
Process Creation

We can create processes using `fork()` system call.

In UNIX-based OS, *init* is the first process started by the kernel during booting of the computer system. Init is a **daemon** process that continues running until the system is shut down. It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes.

During the course of execution, **a process may create several new processes** using the `fork()` system call:

1. The process creator is called a parent process, the new processes are called the children of that process.
2. Each of these new processes may in turn create other processes, forming a tree of processes shown below:



3. Each process is i**dentified by an integer called the process id (pid).** Pid is unique in the system.
4. The new process consists of the **entire** copy of the address space (code, stack, process of execution, etc) of the original process at the point of `fork()`,
5. Hence parent and child processes all operate in **different address space (isolation)**
6. Parent and children processes execute **concurrently**
7. Typically, a parent process **waits** for its children to terminate (using `wait()` system call) to read the child process' exit status and finally its entry in the process table can be removed.
8. Children processes cannot wait for their parents.
9. Since children processes **are a duplicate of their parents**, they can either
    a. **Execute the same code** as their parents concurrently, or
    b. **Load a new program** onto its address space

How `fork()` process creation works

It is best to explain how `fork()` process creation works by example.

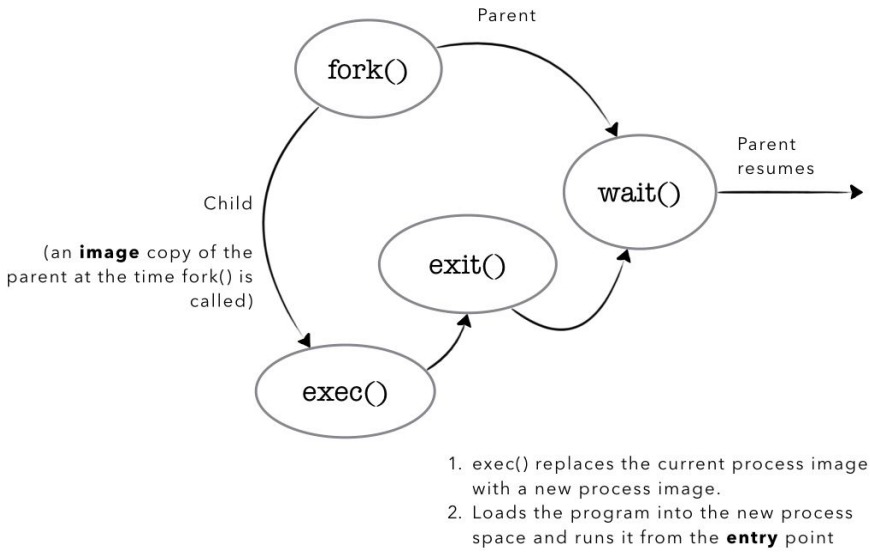
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

The simple C program above is executed and upon the execution of line `pid = fork()`, two processes are present: the parent and the child process.

Both have the **same copy** of the **code** and **resources (any opened files, etc)**, but are at **a different address space**, executed concurrently by the system.

However, `fork()` returns 0 in the child process while in the parent process it returns the pid of the child (>0).

The child executes the line `execlp` and the parent process executes the `else` clause instead. The flow of the execution of both processes are shown in the diagram below:



Explanation:

- `execlp` loads a new program called `"ls"` onto the child process' address space, effectively **replacing** it.
- The parent process at the same time executes `wait(NULL)`, which is a system call that suspends the parents' execution until this child process that is executing `"ls"` has returned.

Process Termination

We can terminate processes using `exit()` system call.

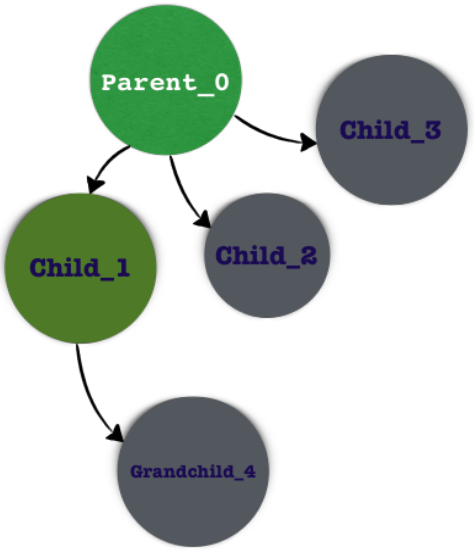
When a process creates a child process, the child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process (be it the children or the parents) wants to terminate itself by calling `exit()` as its last instruction to be executed, these resources are freed by the kernel for other processes.

Parent processes may terminate or abort its children. If a parent process with live children is terminated, the children processes become **orphaned processes**:

- Some operating system is designed to either abort all of its orphaned children (**cascading termination**) or
- Adopt the orphaned children processes (init process usually will adopt orphaned processes)

Zombie Processes

Zombie processes happen when the process has already terminated, and memory as well as other resources are freed, but its exit status is not read by their parents, hence its entry in the process table remains.



The diagram above illustrates three zombie processes in grey, and two *parent processes* (parents are relative, child_1 is a parent to grandchild_4, etc) -- that have yet to call `wait` to read their children's exit status.

Explanation:

- Children processes can terminate itself after they have finished executing their tasks using `exit(int status)` system call.
- The kernel will free the memory and other resources from this process, but not the PCB entry.
- Parent processes are supposed to call `int wait(int* status)` to obtain the exit status of a child. This `wait` system call returns the `pid` of the child.
- Only after `wait()` in the parent process returns, the kernel can remove the child PCB entry from the system wide process table.

If the parents didn't call wait and instead continue execution of other things, then children's entry in the pcb remains -- **a zombie process is created**.

Why this is not a good thing:

- The process status of the child is terminated, and its resources have been freed.
- This generally takes up very little memory space, but pid of the child remains
- Recall **that pid is unique,** hence in a 32-bit system, there's only 32768 available pids. Having too many zombie processes will result in inability to create new processes in the system.

Note: *All processes transition to this zombie state when they terminate, but generally they exist as zombies only briefly. Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.*

What happens if we have zombie processes:

- We can type `ps aux | grep 'Z'` in command line to list all zombie processes
- We cannot kill them, because technically these processes are already terminated
- If they exist, one of the things that are guaranteed **to remove zombie processes** are by **terminating their parents**

Interprocess Communication

Processes executing concurrently in the operating system may be either **independent** processes or **cooperating** processes:

- **By default, processes are independent** and isolated from one another (runs on its own virtual machine)
- A process is cooperating if it can affect or be affected by the other processes executing in the system.

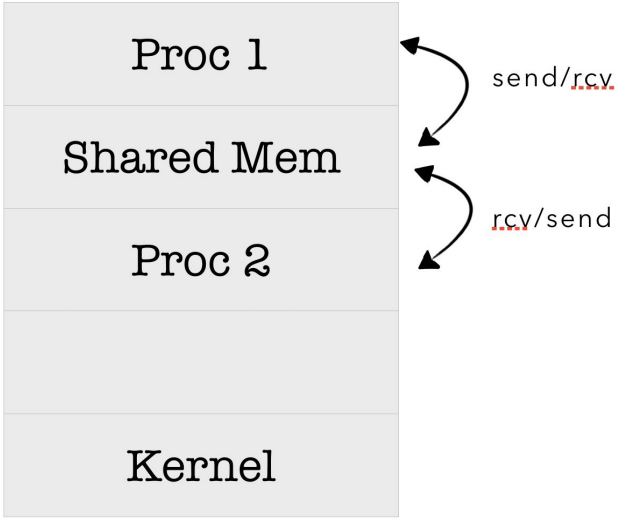
Why do we need some processes to cooperate:

1. Information sharing
2. Speeding up computations
3. Modularity (protect each segments) and convenience

Cooperating processes require Interprocess Communication (IPC) mechanisms -- these mechanisms are provided by or supported by the Kernel. There are two ways to perform IPC:

1. **Shared Memory**
2. **Message Passing (Sockets)**

Shared Memory



How it works:

1. **Create** a shared memory region in the RAM **using system call**: Kernel helps to allocate and establish a region of memory and return to the caller process

2. Once shared memory is established, all accesses are treated as routine memory accesses, and **no assistance from the kernel is required.**

POSIX Shared Memory:

1. Both reader and writer get the shared memory identifier (an integer) using system call `shmget`. `SHM_KEY` is an integer that has to be unique, so any program can access what is inside the shared memory if they know the `SHM_KEY`. The Kernel will return the memory identifier associated with `SHM_KEY` if it is already created, or create it when it has yet to exist. The second argument: `1024`, is the size (in bytes) of the shared memory.

```
int shmid = shmget(SHM_KEY,1024,0666|IPC_CREAT);
```

2. Then, both reader and writer should attach the shared memory onto its address space. You can type cast the return of `shmat` onto any data type you want. In essence, `shmat` returns an address of your address space that translates to the shared memory block[4].

```
char *str = (char*) shmat(shmid,(void*)0,0);
```

3. Afterwards, writer can write to the shared memory:

```
sprintf(str, "Hello world \n");
```

4. Reader can read from the shared memory:

```
printf("Data read from memory: %s\n",str);
```

5. Once both processes no longer need to communicate, they can detach the shared memory from their address space:

```
shmdt(str);
```

6. Finally, one of the processes can destroy it, typically the reader because it is the last process that uses it.

```
shmctl(shmid,IPC_RMID,NULL);
```

Of course one **obvious issue** that might happen here is that **BOTH writer and reader are accessing the shared memory concurrently**, therefore we will run into synchronisation problem whereby writer *overwrites* ***before reader finished reading*** or reader attempts to read an empty memory value ***before writer finished writing*****.** We will address such synchronisation problem in the next lecture notes.

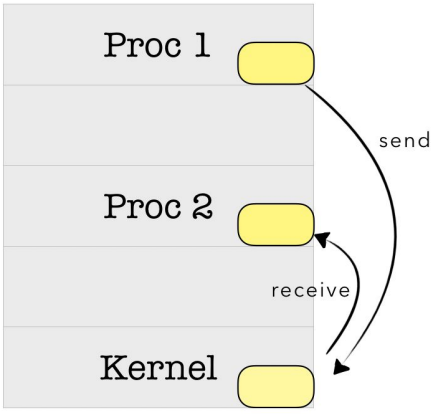
[4] Suppose process 1 and process 2 have successfully attached the shared memory segment. This shared memory segment will be part of their address space, although the actual address could be different (*i.e.*, the starting address of this shared memory segment in the address space of process 1 may be different from the starting address in the address space of process 2).

Sockets (Message Passing)

Every message passed back and forth between writer and reader (server and client) through sockets **must be done using kernel's help**.

A socket is **one endpoint** of a **two-way communication link** between two programs running on the network:

- It is a **concatenation** of an IP address, e.g: 127.0.0.1 for localhost
- **And** TCP(connection-oriented) or UDP (connectionless) port, e.g: 8080. We will learn more about UDP and TCP as network communication protocol in the later part of the semester.
- When concatenated together, they form a socket, e.g: 127.0.0.1:8080
- All socket connection between two communicating processes must be unique



Sockets

For processes in the same machine as shown in the figure above, both processes communicate through a socket with IP *localhost* and a unique port number. Processes can `read()` or `send()` data through the socket through system calls.

Comparison between Socket and Shared Memory

- Socket requires system calls for each message passed through `send()` and `receive()` but much quicker. Shared memory requires one costly system call in the beginning to create the memory segment (this is very costly, depending on the size), but not afterwards when processes are using them.
- Socket is useful for sending smaller amounts of data between two processes, while shared memory is *costly* if only small amounts of data are exchanged.

- Sockets **does not require any synchronisation mechanism** (because the Kernel will synchronise the two), but shared memory requires additional synchronisation to prevent execution issues.

Application Example: Chrome Browser Multi-process Architecture

Websites contain multiple active contents: Javascript, Flash, HTML etc to provide a rich and dynamic web browsing experience. You may open several tabs and load different websites at the same time. However, some web applications contain bugs, and may cause the entire browser to crash if the entire Chrome browser is *just one single huge process*.

Google Chrome's web browser was designed to address this issue by creating separate processes:

1. **The Browser process** (manages user interface of the **browser (not website)**, disk and network I/O) -- Only one browser process is created when Chrome is opened
2. **The Renderer processes** to render web pages: a new renderer process is created for each website opened in a new tab
3. **The Plug-In processes** for each type of Plug-In

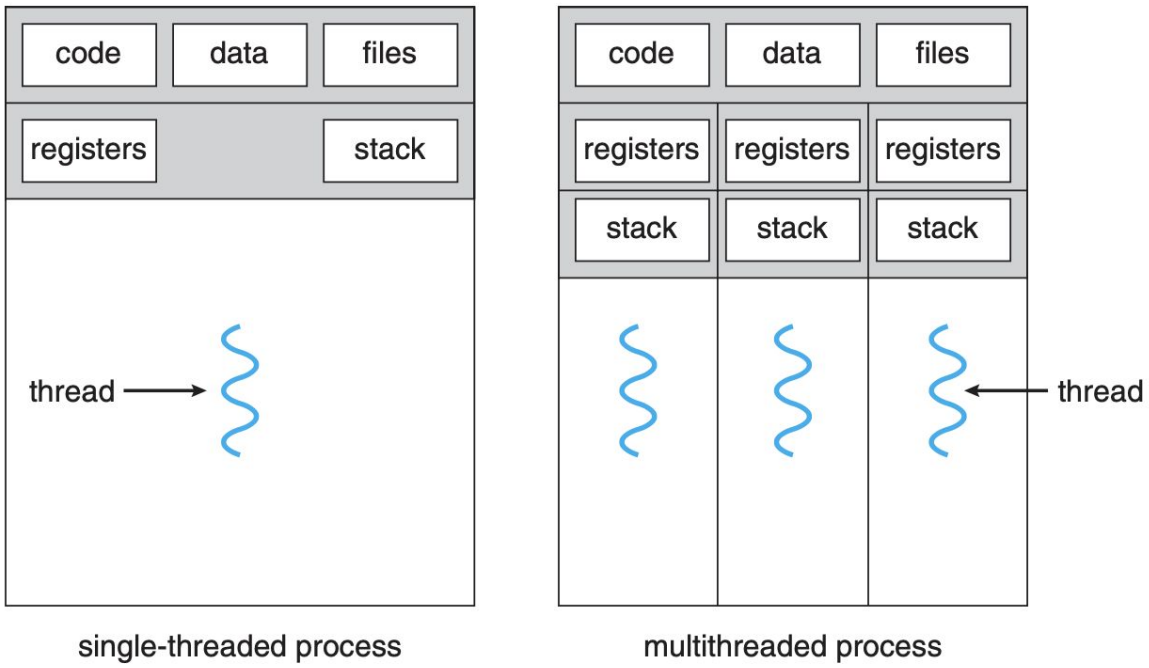
Obviously all processes created by the Chrome have to communicate with one another depending on the application, therefore IPC mechanisms are needed. The advantage of such multiprocess architecture is that:

1. **Each website runs in isolation from one another: if one website crashes, only its renderer crashed and the other processes are unharmed.**
2. Renderer will also be unable to access the disk and network I/O directly (runs in **sandbox: limited access**) thus reducing the possibility of security exploits.

Threads

A process can have multiple **threads**, and we can define thread as a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack.

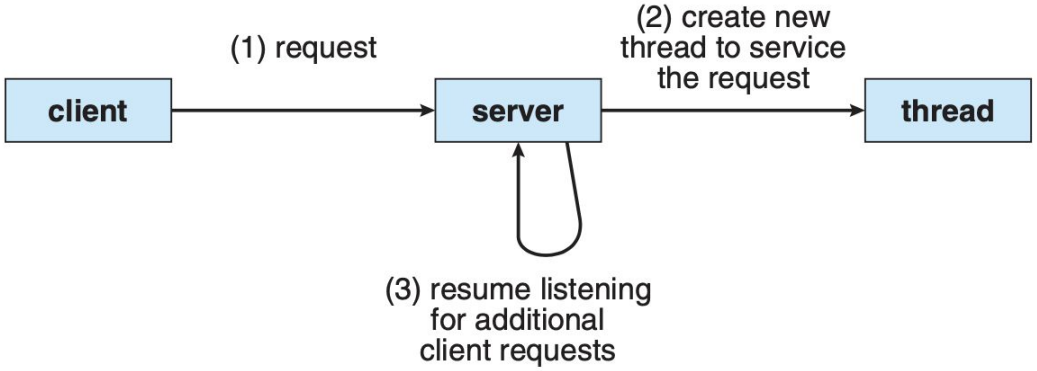
The figure below shows the illustration of a single-threaded and multi threaded processes. As shown, it **shares** with other threads belonging to the same process its **code section, data section**, and **other operating-system resources, such as open files and signals.**



Why threads are useful:

1. **Threads make the program seem more responsive**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

   For example, as illustrated in the multithreaded server architecture below:



2. **Easier means for resource sharing and communication** (since code, data, and files are shared among threads of the same process and they have the same address space), as compared to creating multiple processes. Processes can only share resources through techniques such as shared memory and message passing.

3. **Creating new threads require cheaper resources than creating new process.** Allocating memory and resources for process creation is costly. Because threads

share the resources of the process to which they belong, it is more economical to create and context-switch threads.

4. **Allows us to reap the benefits of true parallel programming in multiprocessor architecture.**. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores.

We can note certain points when comparing between multithreaded and multi process architecture:

Process	Threads	Topic
Processes have both concurrency and protection. One process is isolated from the other, hence providing *fault isolation*.	Threads only have concurrency but *not* protection since code, data, and files are shared between threads of the same process. There's no fault isolation.	**Protection and Concurrency**
IPC is *expensive*, requires system calls and context switch between processes has high overhead	Thread communication has low overhead since they share the same address space. Context switching (basically thread switching) between threads is much cheaper.	**Communication**
Available on multicore systems	Not always available on multicore system, depends on the types of threads (See next section)	**Parallel execution**[5]
We can rely on OS services (system calls) to synchronise execution between processes, i.e: using semaphores	Requires careful programming to synchronise between threads	**Synchronisation**

[5]Notice the distinction between ***parallelism*** and ***concurrency*** mentioned in this table. A system is **parallel** if it can perform more than one task simultaneously (in time). In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism.

Thread Examples -- Java and Pthread

Java Thread

Java threads are managed by the JVM. You can create Java threads in **two ways.**

METHOD 1: The first way is by implementing a runnable interface and call it using a thread:

1. Implement a runnable interface and its `run()` function:
```
public class MyRunnable implements Runnable {
    public void run(){
       System.out.println("MyRunnable running");
    }
  }
```

2. Create a new Thread instance and pass the runnable onto its constructor. Call the `start()` function to begin the execution of the thread:
```
Runnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
thread.start();
```

METHOD 2: The second way is to create a subclass of `Thread` and override the method `run()`:

1. For example,
```
public class MyThread extends Thread {
  public void run(){
     System.out.println("MyThread running");
  }
}
```
2. Create and start MyThread instance to start the thread:
```
MyThread myThread = new MyThread();
myThread.start();
```

C Thread

We can also equivalently create threads in C using the Pthread library. The code below shows how we can create threads in C:

1. Create a function with void* return type (generic return type)
2. Use `pthread_create` to create a thread that executes this function
3. Use `pthread_join` to wait for a thread to finish

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

```
#include <pthread.h>

void *myThreadFunc(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFunc, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

Types of Threads: Kernel and User

There are two types of threads:

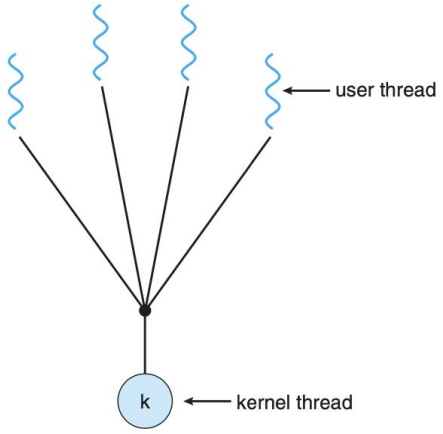
Kernel Threads	User Threads
Known to OS Kernel, runs in Kernel mode	Not known to OS Kernel, runs in User mode
Scheduled directly by Kernel, run in Kernel mode	Scheduled by thread scheduler running in user mode. The thread scheduler exists in the thread library (e.g: phtread of Java thread) linked with the process
Multiple Kernel threads can run on different processors	Multiple user threads on the *same user process* may not be run on multiple cores (since the Kernel isn't aware of their presence). It depends on the mapping model (see next section)
Take up Kernel data structure, is specific to the operating system.	Take up thread data structure (depends on the library), more generic and can run on any system
Requires more resources to allocate / deallocate	Cheaper to allocate / deallocate

Thread Mapping

Since there are two types of threads: kernel and user thread, there should exist some kind of mapping between the two. There are three types of mapping:

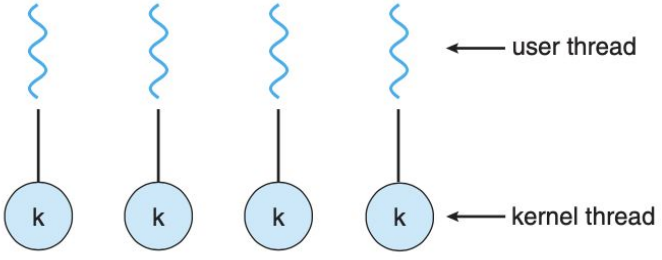
1. **Many to One:** maps many user-level threads to one kernel thread.
   Advantage:
   - Thread management is done by the thread library in user space, so it is more efficient as opposed to kernel thread management.
   - Developers may create as many threads as they want

   Disadvantage:
   - The entire process will block if a thread makes a blocking system call since kernel isn't aware of the presence of these user threads
   - Since only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems

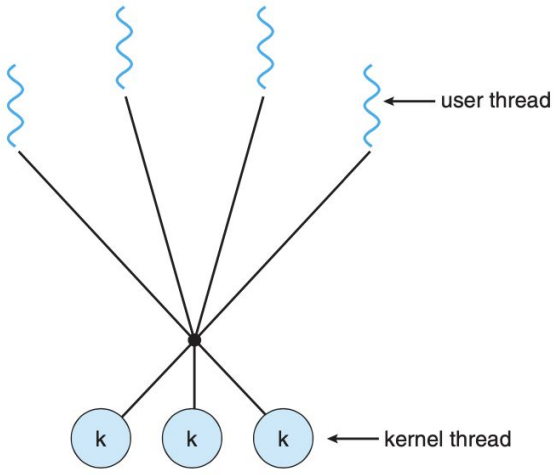


2. **One to One**: maps each user thread to a kernel thread.
   Advantage:
   - Provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
   - Allows multiple threads to run in parallel on multiprocessors.

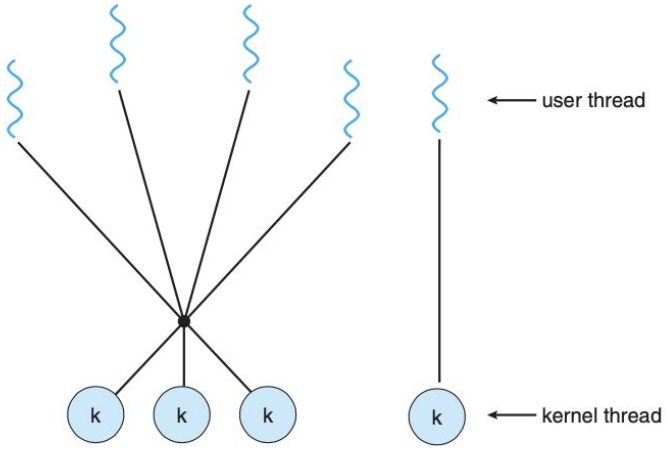
   Disadvantage:
   - Creating a user thread requires creating the corresponding kernel thread (a lot of overhead)
   - Limited amount of threads can be created to not burden the system



3. **Many to Many:** multiplexes many user-level threads to a smaller or equal number of kernel threads. This is the best of both worlds:
    - Developers can create as many user threads as necessary,
    - The corresponding kernel threads can run in parallel on a multiprocessor.



A variation of many-to-many mapping -- The two-level model: both multiplexing user threads and allowing some user threads to be mapped to just one kernel thread:



Intel Hyper-Threading

Hyper-threading was Intel's first effort (proprietary method) to bring *parallel* computation to end user's PCs by fooling the kernel to *think* that there exist *> 1 processors* in a single processor system:

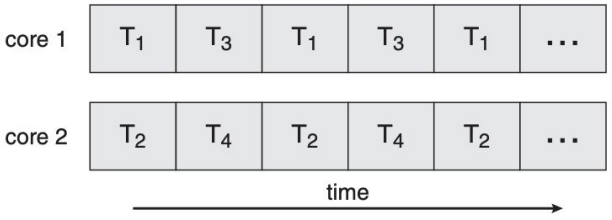
- A single CPU with hyper-threading appears as two or more ***logical*** CPUs (with all its resources and registers) for the operating system kernel

- This is a process where a CPU splits each of its physical cores into virtual cores, which are known as threads. Hence the kernel assumes two CPUs for each single CPU core, and therefore increasing the efficiency of one physical core -- it lets a single CPU to fetch and execute instructions from two memory locations **simultaneously**

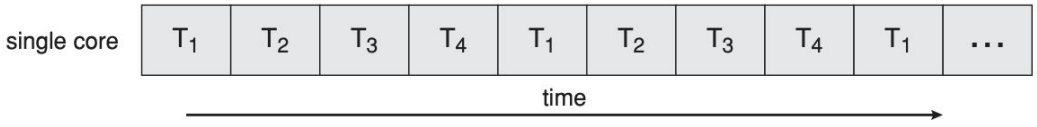
- This is possible since a big chunk of CPU time is wasted as it waits for data from cache or RAM

Multicore Programming

On a system with multiple cores, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core. Parallel execution on a multicore system is as shown (Ti stands for Thread i):



This is inherently different from concurrent execution with a single core as shown:



There are two types of parallelism: **data parallelism and task parallelism**.

Data parallelism focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

Task parallelism involves distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

Ahmdal's Law

Not all programs can be 100% parallelised. Some programs required a portion of its instructions to be serialised. We need to identify potential performance gains from adding

additional computing cores to an application that has both serial (nonparallel) and parallel components.

Given that α is the fraction of the program that must be executed serially, the maximum speedup that we can gain when running this program with N processors is:

$$\frac{1}{\alpha + \frac{(1-\alpha)}{N}}$$

*Note: **it is extremely important** for you to take some time to realise and understand how to derive this formula instead of memorising it blindly.*

Appendix: Daemon Processes

A daemon is a background process that performs a specific function or system task. In keeping with the UNIX and Linux philosophy of modularity, daemons are programs rather than parts of the kernel. Many daemons start at boot time and continue to run as long as the system is up. Other daemons are started when needed and run only as long as they are useful.

The daemon process is designed to run in the background, typically managing some kind of ongoing service. A daemon process might listen for an incoming request for access to a service. For example, the `httpd` daemon listens for requests to view web pages. Or a daemon might be intended to initiate activities itself over time. For example, the `crond` daemon is designed to launch cron[6] jobs at preset times.

When we refer to a daemon process, we are referring to a process with these characteristics:

- **Generally persistent** (though it may spawn temporary helper processes like xinetd does)
- **No controlling terminal** (and the controlling `tty` process group (`tpgid`) is shown as -1 in `ps`)
- Parent process is generally **init** (process 1)
- Generally has its own process group id and session id

[6] **cron** is a Linux utility which schedules a command or script on your server to run automatically at a specified time and date. A **cron job** is the scheduled task itself.