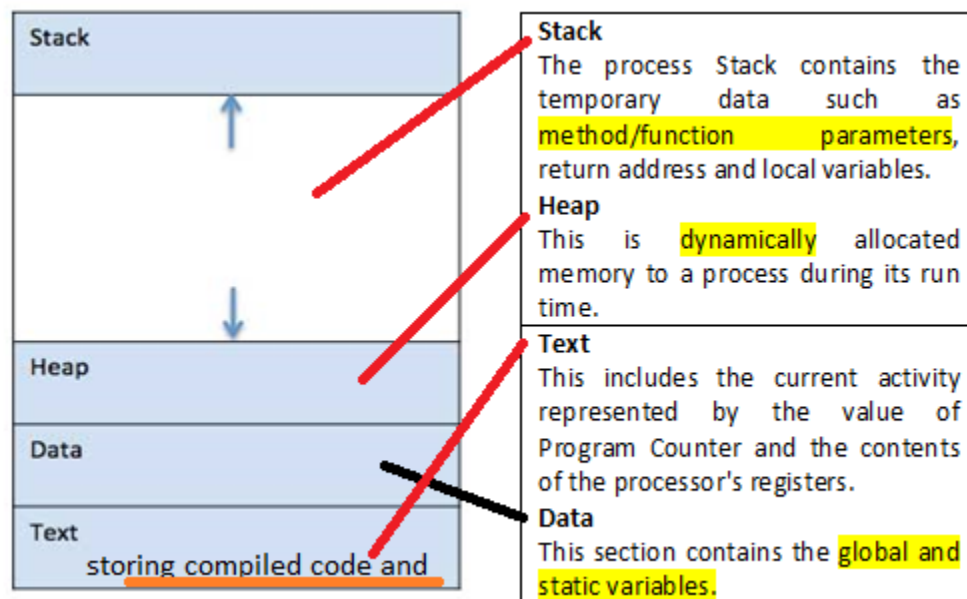


# Defining Process and Thread

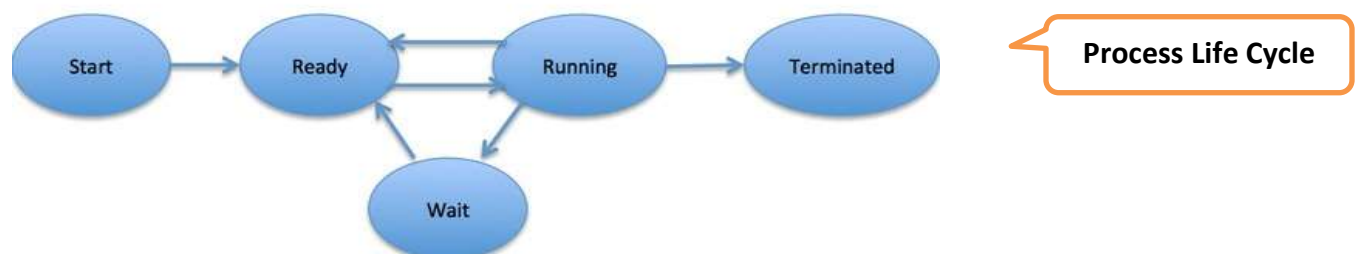
## Process

- An executing instance of a program is called a **process**. Some operating systems use the term '**task**' to refer to a program that is being executed.
- When a program is loaded into the memory by OS it becomes a process. A process has its own **address space** meaning having **various segments** in memory containing **program text (compiled code) and data**, as well as other **resources**. It can be divided into four major sections – **stack, heap, text and data**.

These resources may include open files, child processes, pending alarms, signal handlers, accounting information, and more. For managing the resources easily **OS maintain a PCB** (described latter). The following image shows a simplified layout of address spaces.

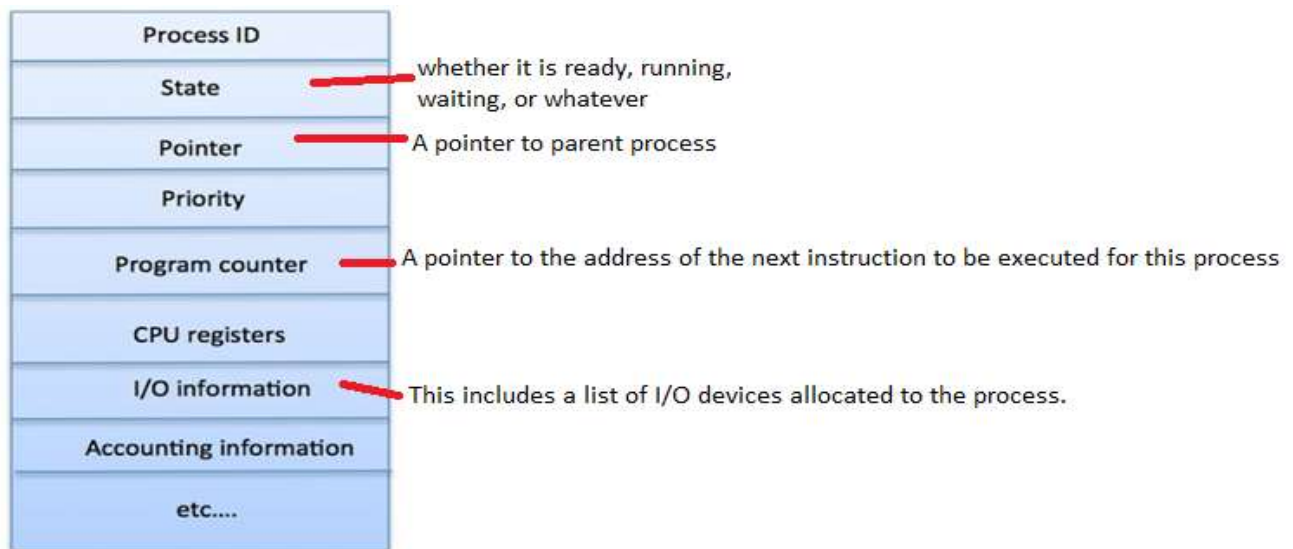


- Each process is started with a single thread, often called the **primary thread**, but can create additional threads from any of its threads.
- Several processes may be associated with a same program.
- **Both processes and threads are independent sequences of execution.**
- As each process uses separate address space it has no **synchronization overhead** like a thread.
- Resources (memory, handles etc.) are release at **process termination, not thread termination**.



- **Process Control Block (PCB)**

- A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as shown below.
- The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB.
- The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.



### Example of process and thread

1. On your computer, open **Microsoft Word and web browser**. We call these **two processes**.
2. In **Microsoft word**, you type something and it gets automatically saved. Now, you would have observed editing and saving happens in parallel. This is called a **thread**.
3. Executing multiple instances of the 'Calculator' program. Each of the instances are termed as a process.
4. Opening multiple tabs in the browser is an example of threads.

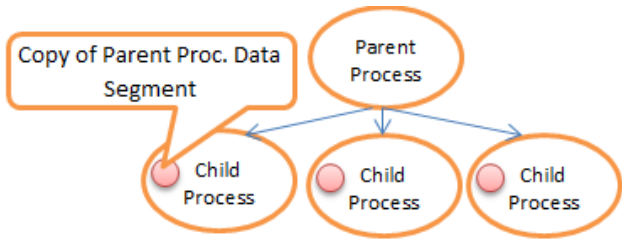
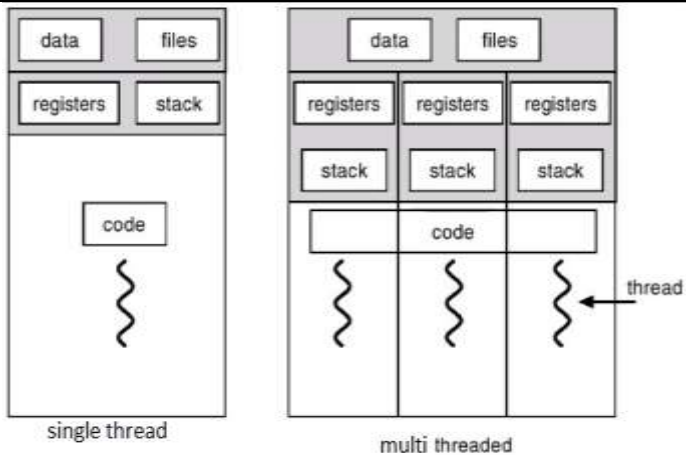
### Thread

- A thread is an entity within a process that can be scheduled for execution. The OS allocates processor time for each thread. A thread can execute any part of the process code, including parts currently being executed by another thread.
- All threads of a process share its **address space, system resources and other process related attributes**. It makes threads lightweight and hence **context switch** between the threads are not much time consuming as Process.
- In addition, each thread **maintains exception handlers, a scheduling priority, thread's own stack for storing local storage, a unique thread identifier etc.**
- Threads do not need inter process communication techniques because they are not altogether

separate address spaces. They share the same address space; therefore, they can **directly communicate with other threads of the process.**

- Threads of a process share the same address space; therefore **synchronizing** the access to the **shared data** within the process's address space becomes very **important**.
- On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

## Comparison Process vs Thread

Process	Thread
A child process initiated by a parent process may run a <b>different executable</b> by calling an exec function.	All threads in a program must run the <b>same executable.</b>
Child Processes have their <b>own copy</b> of the data segment of the <b>parent process.</b>	Threads have <b>direct access</b> to the data segment of its <b>process</b>
	
Processes must use inter-process communication (IPC) to communicate with sibling processes. It is quite difficult and resource-intensive.	Threads <b>share the address space</b> of the <b>process</b> that created it. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. <b>Synchronizing the access to the shared data within the process is very importance.</b>
Processes have considerable overhead.	Threads have almost no overhead. So it is termed as a <b>'lightweight process'</b>
New processes require duplication of the parent process.	New threads are easily created.
Processes can only exercise control over child processes.	Threads can exercise considerable control over threads of the same process.
Changes to the parent process do not affect child processes.	Changes to the main thread (cancellation, priority change, etc.) may affect the behavior of the other threads of the process.

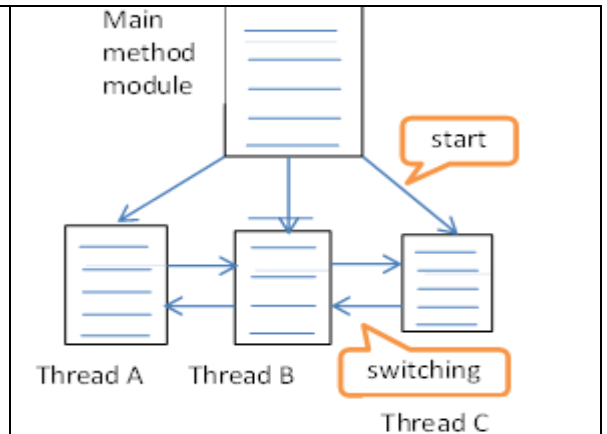
## Each JVM instance has its own OS-level process

- When we issue `java classname1` in command prompt, OS allocate a process to JVM and the OS starts the JVM, the JVM runs a single class's public static void `main()` method. You may link to other classes from this entry-point, and from other classes as well. This would continue to occur within the `same JVM process`, unless you launch another process (to run another program).
- Each Java application uses an independent JVM. Generally speaking, each application will get its `own JVM instance` and its `own OS-level process` and each JVM instance is `independent of each other`. This means there is no sharing of stacks, heaps, etc. (Generally, the only things that might be shared are the `read only segments` that hold the `code of the core JVM` and `native libraries`). Multiple JVM instances might share some data/memory but those have no user-visible effect to the applications. `Garbage collection` operates on `each JVM independently`.
- JVMs are not treated any different to other OS processes. The OS may refuse to start any more JVM processes if it runs out of resources.
- A common scenario however is a single application server (or "web server") such as `Glassfish` or `Tomcat` running multiple web applications. In this case, multiple web applications `can share a JVM` if they're deployed to the same web container.
- An IDE is a good example of an application which is presented to its end users as a `single entity` but which is actually comprised of `multiple underlying applications` (compilers, test runners, static analysis tools, packagers, package managers, project / dependency management tools, etc). In that case there are a variety of tricks which the IDE uses to ensure that the user experiences an integrated experience while also being shielded (to some extent) from the individual vagaries of the underlying tools. One such trick is to do *some* things in a separate JVM, communicating either via text files or via the application-level debugging facilities.

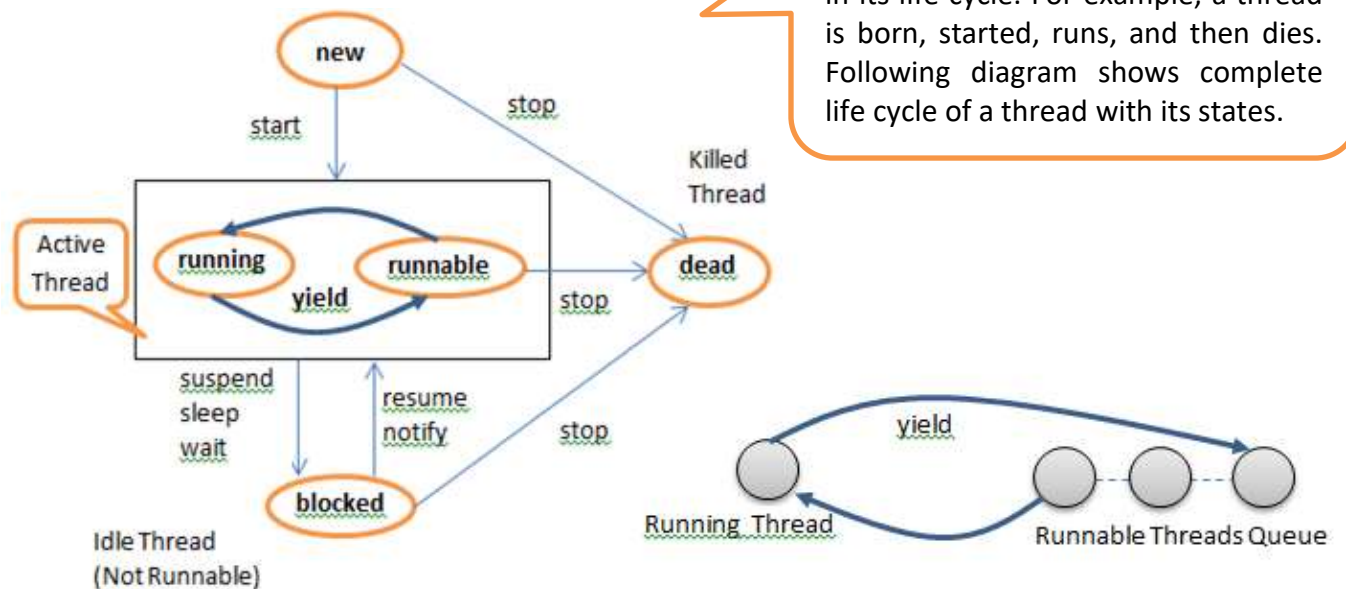
## Multithreading

- A multithreaded program contains two or more parts that can run `concurrently` and each part can handle different task at the same time making `optimal use of the available resources` especially when your computer has `multiple CPUs`.
- Java is a *multithreaded programming language* which means we can develop multithreaded program using Java.
- A thread is a so called `lightweight process`. It shares the `process address space` and in addition
  - It has its `own call stack`,
  - `Can access shared data` of other threads in the same process.
  - Every thread has its own `memory cache`. If a thread reads `shared data` it stores this data in its own memory cache. A thread can re-read the shared data.

- A Java application runs by default in primary thread (single thread) of one process. Within a Java application you work with several threads to achieve parallel processing or asynchronous behavior.
- Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time.



## Life Cycle of a Thread



1. **New:** A new thread begins its life cycle in the new state. It remains in this state until it is scheduled using `start()` method of thread and it will be in runnable state. Kill it using `stop()`.
2. **Runnable:** At this state thread is waiting in a queue for the availability of processor. When processor time slot is allocated it starts executing its task.
3. **Running :** At this state the thread runs until it relinquish control on its own or it is preempted (blocked) by a higher priority thread. The running thread may relinquish control in one of the following situation as discussed below.
4. **Blocked :** At this state the thread is considered “not runnable” but not dead and therefore can run again. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
5. **Dead :** Thread completes its life when finished execution of `run()` method. However thread can be killed by `stop()` method too.

## Various situation of relinquishing control of a running thread ( thread going active to idle state)

<pre> graph LR     Running((Running)) -- suspend --&gt; Runnable((Runnable))     Runnable -- resume --&gt; Running     Runnable -- suspend --&gt; Suspended((Suspended))     Suspended -- resume --&gt; Runnable         </pre>	<p>1. Relinquishing control using <b>suspend()</b> method. A suspended thread can be revived by using <b>resume()</b> method.</p>
<pre> graph LR     Running((Running)) -- sleep(t) --&gt; Runnable((Runnable))     Runnable -- after(t) --&gt; Running     Runnable -- sleep(t) --&gt; Suspended((Suspended))     Suspended -- after(t) --&gt; Runnable         </pre>	<p>2. We can put a thread to sleep for a specified time period using the method <b>sleep(time)</b> method. It is out of runnable thread queue during that time period. It will be back to the runnable state when that time interval expires.</p>
<pre> graph LR     Running((Running)) -- wait --&gt; Runnable((Runnable))     Runnable -- notify --&gt; Running     Runnable -- wait --&gt; Waiting((Waiting))     Waiting -- notify --&gt; Runnable         </pre>	<p>3. It has been told to wait until some event occurs. This is done using the <b>wait()</b> method. The thread can be scheduled to run again using the <b>notify()</b> method.</p>

## Thread Priorities:

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java thread priorities are in the range between **MIN\_PRIORITY** (a constant of 1) and **MAX\_PRIORITY** (a constant of 10). By default, every thread is given priority **NORM\_PRIORITY** (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads.
- However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

## Role of Threads scheduler

- Thread Scheduler is the **Operating System service** that allocates the **CPU time** to the available runnable threads. Once we create and start a thread, its execution depends on the implementation of Thread Scheduler. Allocation of CPU time (Time Slice) to threads can be **based on thread priority of runnable threads** or the thread waiting for longer time will get more priority in getting CPU time.
- Threads scheduling is controlled by **thread scheduler** and not by **java**. So, you cannot guarantee the order of execution of threads under normal circumstances.
- When you call **Thread.start**, the **thread is created and schedule for execution**, it might happen **immediately** (or close enough to it), it might not. It comes down to the **thread scheduler**.
- Typically, each thread will be given **a small amount of time to execute by thread scheduler** before it is put back to "sleep" and another thread is allowed to execute (obviously in multiple processor environments, more than one thread can be running at time, but let's try and keep it simple )

## Context Switching

A context switch is the process of **storing** and **restoring** the **state** (more specifically, the execution context) of a **process** or **thread** so that execution can be **resumed from the same point** at a later time. This enables multiple processes or threads to share a single CPU and is an essential feature of a multitasking operating system.

The concept of context switching is **integral part to threading**. The **context** of a thread only consists of an id, a stack, a register set, and a priority. **The register set contains the program or instruction pointer and the stack pointer**. A hardware timer is used by the processor to determine the end of the time slice for each thread. The timer signals at the end of the time slice and in turn the processor saves all information (context) required for the current thread onto a **stack**. Then the processor moves this information from the **stack into a predefined data structure called a context structure**. When the processor wants to switch back to a previously executing thread, it transfers all the information from the context structure associated with the thread to the **stack**. This entire procedure is known as **context switching**. Java supports thread-based multitasking. Context-switching between threads is normally inexpensive.

### Preemptive and Cooperative Multitasking

In multitasking OS two types of algorithms exists – **preemptive multitasking** and **cooperative multitasking**.

- In **preemptive** multitasking, each process is allotted a certain period of time known as **time slice**. So one process cannot hold the CPU time as long it wants. Thus, CPU time is shared by all the processes and is known as **time sharing**
- In contrary, in **cooperative** multitasking, one process can hold the microprocessor time as long as it would like. But the CPU is empowered to remove the process if the process does not utilize the time after holding it.

### When context switch will take place in multithreading environment?

- All the threads in a single process are called **peers** and are on an equal level regardless of who created whom. Threads can also **suspend, resume, and terminate other threads within its process**.
- When a CPU switches from executing one thread to executing another, the CPU needs to save the local data, program pointer etc. of the current thread, and load the local data, program pointer etc. of the next thread to execute. This switch is called a "context switch". The CPU switches from executing in the context of one thread to executing in the context of another.
- You cannot do it in java. It is the responsibility of JVM to switch the context based on underlying operating system and memory model.
- In any case, concurrency is achieved through context switching.

Generally there are **four** ways to **cause** an active thread (say T1) **to context switch**. **Three of them** require that the programmer has written some code.



1. **Synchronization** : Common means of being **context** switched is for T1 to request a **mutex** lock and not get it. If the lock is already being held by T2, then T1 will be placed on the **sleep** queue, awaiting for the lock, thus allowing a different thread to run.
2. **Preemption** : A common thread (T1) does something that causes a high-priority thread (T2) to become runnable. In this case, the lowest-priority active thread (T1) will be pre-empted and T2 will take its place. The ways of causing this to happen include releasing a lock, and changing the priority level of T2 upward or of T1 downward.
3. **Yielding** : If the programmer puts an explicit call to the **yield** call in the code that T1 is running, the scheduler will look to see if there is another runnable thread (T2). If there is one, that thread will be scheduled. If there isn't one T1 will continue to run.
4. **Time slicing** : If OS allows time slicing T1 might simply have its time slice run out and T2 ( at the same priority level) would then receive a time slice.