1. PROCESS AND THREAD LIFECYCLE IN LINUX AND ANDROID

1.1. Process Life Cycle in Linux:

Process:

- A program in execution is known as process.
- When a program is loaded in memory for execution it is said to be process.
- Program is a set of instruction to be executed in order to perform a specific task.
- A program becomes a process when an executable file is loaded into memory.
- Process is an active entity and program is passive entity.
- A process is the unit of work in a modern time-sharing system.

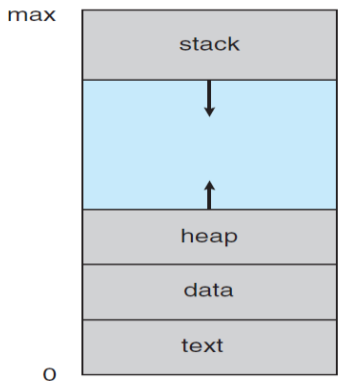
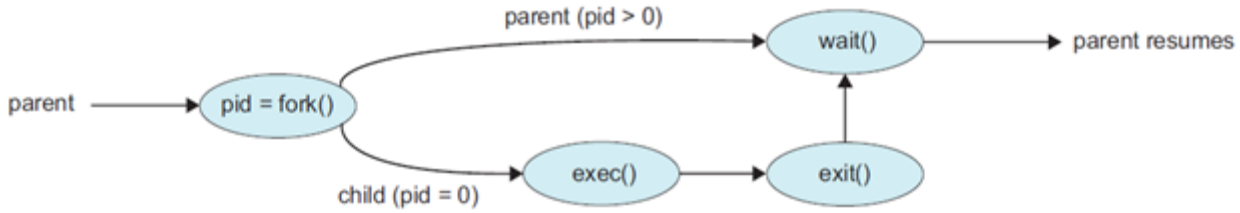


Figure Process in memory.

Process Creation:

- Process creation is the mechanism by which a running process creates a new process called a child process.
- The creating process is known as the parent, and both form a parent–child relationship.
- The child process may share all, some, or none of the parent's resources.
- Parent and child can execute concurrently, or the parent may wait for the child to finish.
- In UNIX systems, process creation is achieved using the fork() system call, often followed by exec().

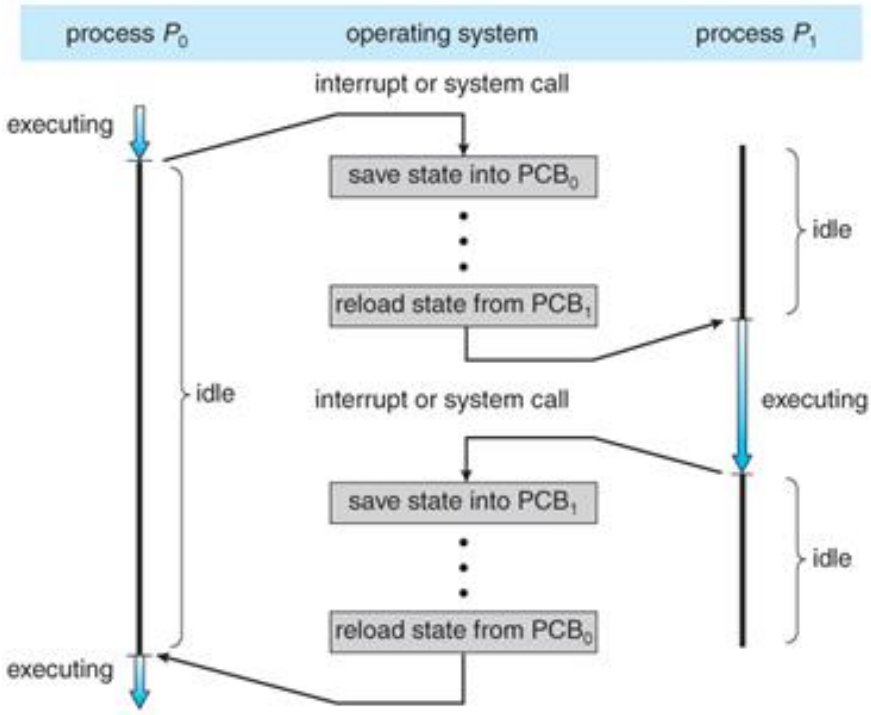


Process creation using the `fork()` system call.

Process Termination:

- **Process Termination** is the final stage of a process life cycle where a process stops execution.
- A process normally terminates after completing its execution using an exit system call.
- A process may also be terminated by its parent due to errors, resource limits, or user requests.
- On termination, the operating system releases all resources allocated to the process.
- If the parent does not collect the child's exit status, the terminated process becomes a zombie.

Context Switching:



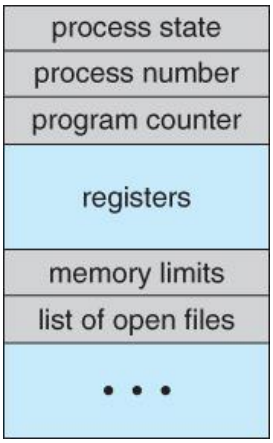
When the CPU switches from executing one process to another, the OS saves the state of the current process and loads the state of the next process

Process Control Block (PCB):

- Each process is represented in the operating system by a **Process Control Block (PCB).**
- **Process control block** also called a **task control block**.

Information associated with process control block are

- ✓ **Process state** - The state may be new, ready, running, waiting, halted, and so on.
- ✓ **Program counter**-The counter indicates the address of the next instruction to be executed for this process.
- ✓ **CPU registers** -The registers vary in number and type, depending on the computer architecture.
- ✓ They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
- ✓ **CPU-scheduling information** - This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- ✓ **Memory-management information** - It contains information such as the value of the base and limit registers and the page tables, or the segment tables,
- ✓ **Accounting information** - This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
- ✓ **I/O status information** - This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

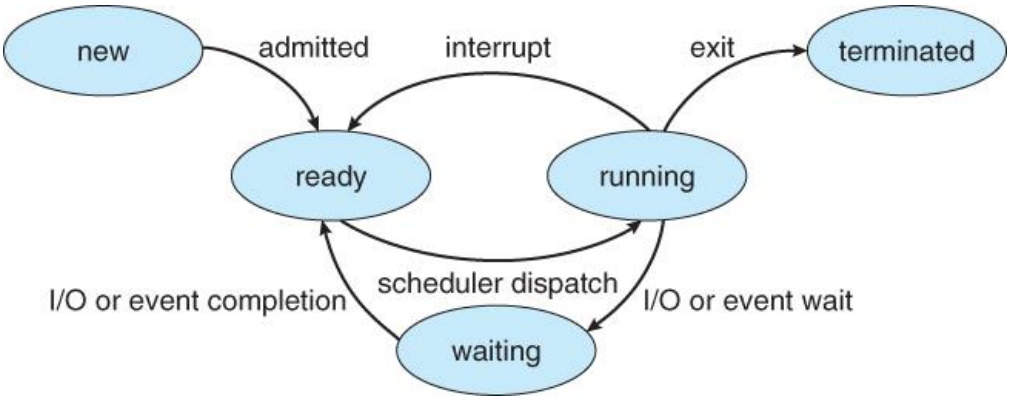


Process Life Cycle:

- A **process** is a program in execution. As a process executes, it changes state according to its current activity.
- This sequence of states through which a process passes is called the **process life cycle**.

Processes transition between states during their execution:

1. **Creation**: Initiated by the operating system using system calls (e.g., fork() in Unix/Linux).
2. **Execution**: The process moves between Ready and Running states based on CPU scheduling.
3. **Waiting**: If the process requires I/O or another event, it transitions to the Waiting state.
4. **Termination**: Once execution is complete or manually stopped, the process is terminated, and resources are released.



Process State:

1. New:

- The process is being **created**.
- OS allocates memory and initializes process control data.
- The program has not yet been admitted to the ready queue.

2. Ready:

- The process is **loaded into main memory**.
- It is ready to execute but **waiting for CPU allocation**.
- Multiple processes can be in the ready state at the same time.

3. Running:

- The process is **currently executing on the CPU**.
- At any instant, only one process can be in the running state on a single-core CPU.
- The process may be preempted by the scheduler.

4. Waiting (or Blocked):

- The process cannot continue execution until **some event occurs**.
- Typical events:
  - I/O completion
  - Signal reception
  - Resource availability

- The process does not compete for CPU in this state.

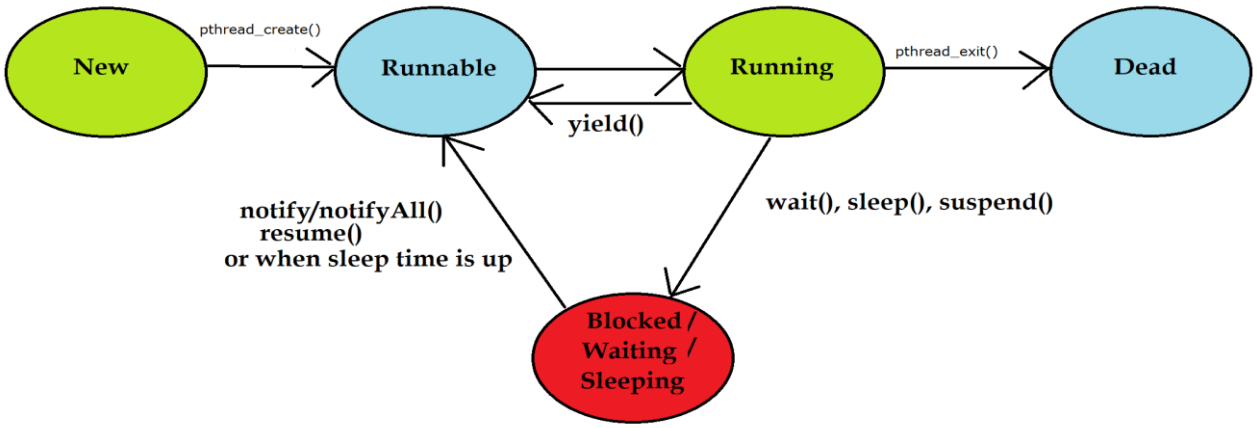
5. Terminated (Exit):

- The process has **finished execution**.
- OS releases all resources allocated to the process.
- The process control block (PCB) is removed.

1.2. Thread Life Cycle in Linux:

The thread life cycle in Linux describes the sequence of states a thread passes through from creation to termination.Threads are created using the clone() system call, usually via pthread_create().

1. **Thread Creation:** A new thread is created using pthread_create().
2. **Thread Execution:** The thread starts executing and runs until it completes or is interrupted.
3. **Thread Sleep:** The thread can sleep or wait for a resource using pthread_cond_wait() or sleep().
4. **Thread Wake-up:** The thread is woken up by a signal or when the resource becomes available.
5. **Thread Termination:** The thread completes execution or is terminated using pthread_exit().



1. New State

- ✓ A thread is in the **New** state when it is **created but not yet scheduled** for execution.
- ✓ Memory and thread control structures are allocated.
- ✓ The thread **has not started executing** its run function.

Transition

- pthread_create()
  → Moves the thread from **New** → **Runnable**

2. Runnable (Ready) State

- ✓ The thread is **ready to execute** and waiting for CPU allocation.
- ✓ It is placed in the **ready queue** by the scheduler.
- ✓ The thread can run **as soon as the CPU is available**.

Transition

- **Runnable → Running** : When the scheduler assigns CPU time.

3. Running State

- ✓ The thread is **currently executing** on the CPU.
- ✓ Instructions of the thread are actively being processed.

Possible Transitions

1. **Running → Runnable**
   - yield() voluntarily gives up the CPU.
   - Preemption by scheduler.
2. **Running → Blocked / Waiting / Sleeping**
   - Thread cannot continue execution because it is waiting for:
     - I/O operation
     - Lock or resource
     - Condition variable
     - Timer expiration
   - Caused by: wait(), sleep(), suspend().
3. **Running → Dead**
   - Thread completes execution or terminates.
   - Caused by:
     - pthread_exit()
     - Returning from thread function
     - Being cancelled

4. Blocked / Waiting / Sleeping State:

- ✓ The thread is **temporarily inactive**.
- ✓ It is **not eligible for CPU allocation**.
- ✓ Waiting for an external event or condition.

Examples:

- Waiting for file I/O.
- Waiting for a mutex or semaphore.
- Sleeping for a fixed time.
- Waiting on condition variables.

Transition:

- **Blocked → Runnable**

  When:

  - notify() or notifyAll() is called.
  - resume() is issued.
  - Sleep time expires.
  - Resource becomes available.

5. Dead (Terminated) State:

- The thread has **finished execution**.
- It no longer consumes CPU.
- Thread resources are released (or reclaimed after pthread_join()).

Causes:

- Normal completion
- Explicit call to pthread_exit()
- Thread cancellation

1.3. Android Process Life Cycle:

- **Android Process Life Cycle**, which shows how Android manages application processes based on **priority and memory availability**.
- Android may move processes between states or terminate them to keep the system responsive.

1. Foreground Process (Highest Priority):

A process is considered foreground if:

- ✓ It contains an **Activity** interacting with the user.
- ✓ It runs a **foreground Service** (e.g., music playback).
- ✓ It is executing a **BroadcastReceiver.**
- ✓ It has a Service bound to a foreground Activity.
- ✓ **Rarely killed by the system.**

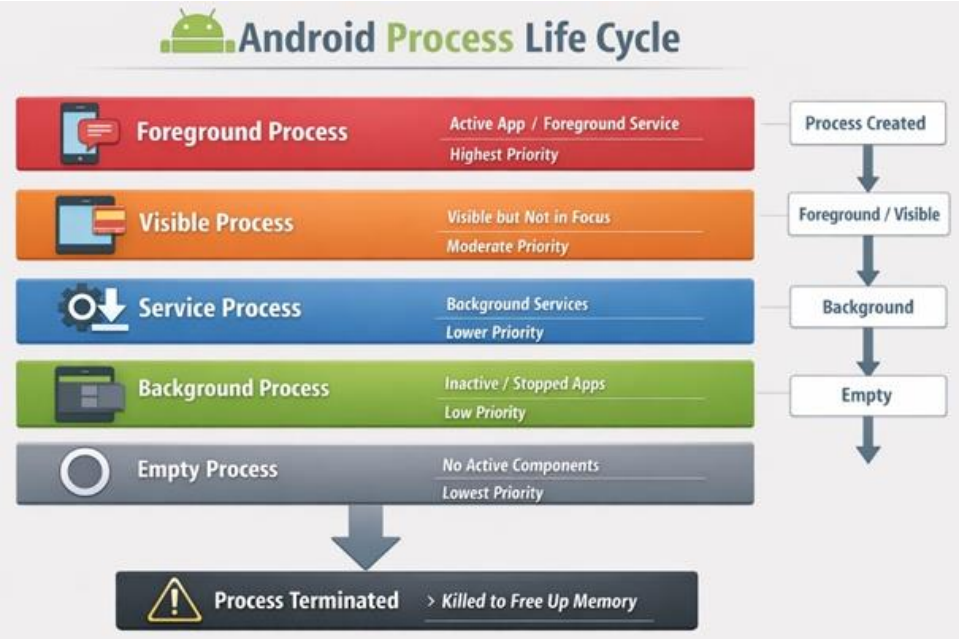
2. Visible Process:

- ✓ Contains an Activity that is **visible but not in focus.**
- ✓ Example: Activity partially covered by a dialog.
- ✓ **Killed only if memory is extremely low.**

3. Service Process:

- ✓ Running a **background Service.**
- ✓ Example: downloading data, syncing information.

- ✓ **More likely to be killed than visible processes.**



4. Background Process:

- ✓ Contains Activities **not visible to the user.**
- ✓ Activities are in the **stopped state.**
- ✓ **Frequently killed to free memory.**

5. Empty Process (Lowest Priority):

- ✓ No active application components.
- ✓ Kept only for **caching** to improve startup speed.
- ✓ **First to be killed when memory is needed.**

Activity Lifecycle in Android:

1. onCreate():

- ✓ It is called when the activity is first created.
- ✓ This is where all the static work is done like creating views, binding data to lists, etc.
- ✓ This method also provides a Bundle containing its previous frozen state, if there was one.

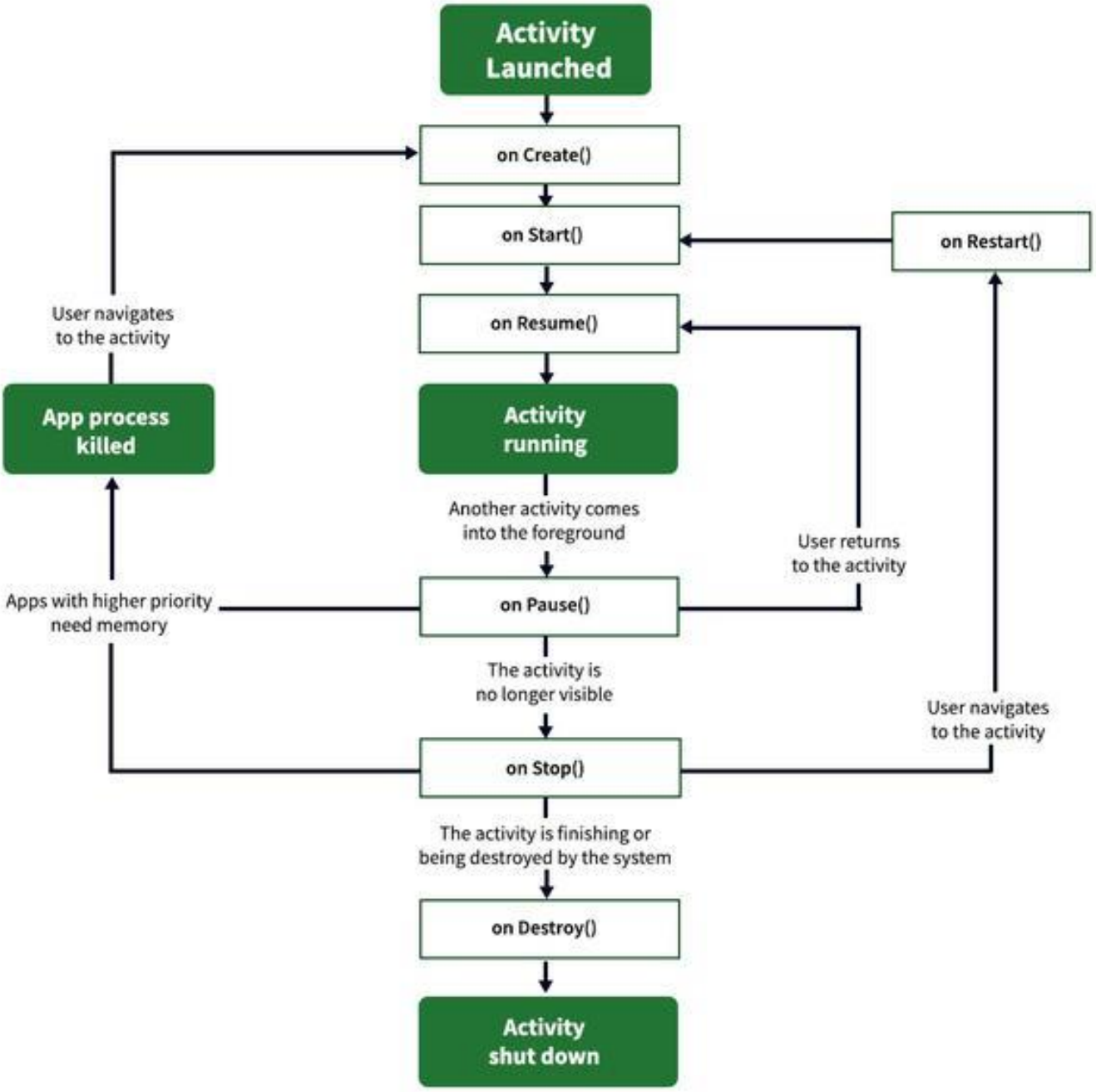
2. onStart():

- ✓ It is invoked when the activity is visible to the user.
- ✓ It is followed by onResume() if the activity is invoked from the background.
- ✓ It is also invoked after onCreate() when the activity is first started.

3. onRestart():

- ✓ It is invoked after the activity has been stopped and prior to its starting stage.

- ✓ Thus is always followed by onStart() when any activity is revived from background to on-screen.



Activity Lifecycle in Android

4. onResume():

- ✓ It is invoked when the activity starts interacting with the user.
- ✓ At this point, the activity is at the top of the activity stack, with a user interacting with it.
- ✓ Always followed by onPause() when the activity goes into the background or is closed by the user.

5. onPause():

- ✓ It is invoked when an activity is going into the background but has not yet been killed.
- ✓ It is a counterpart to onResume(). When an activity is launched in front of another activity, this callback will be invoked on the top activity (currently on screen).
- ✓ The activity, under the active activity, will not be created until the active activity's onPause() returns, so it is recommended that heavy processing should not be done in this part.

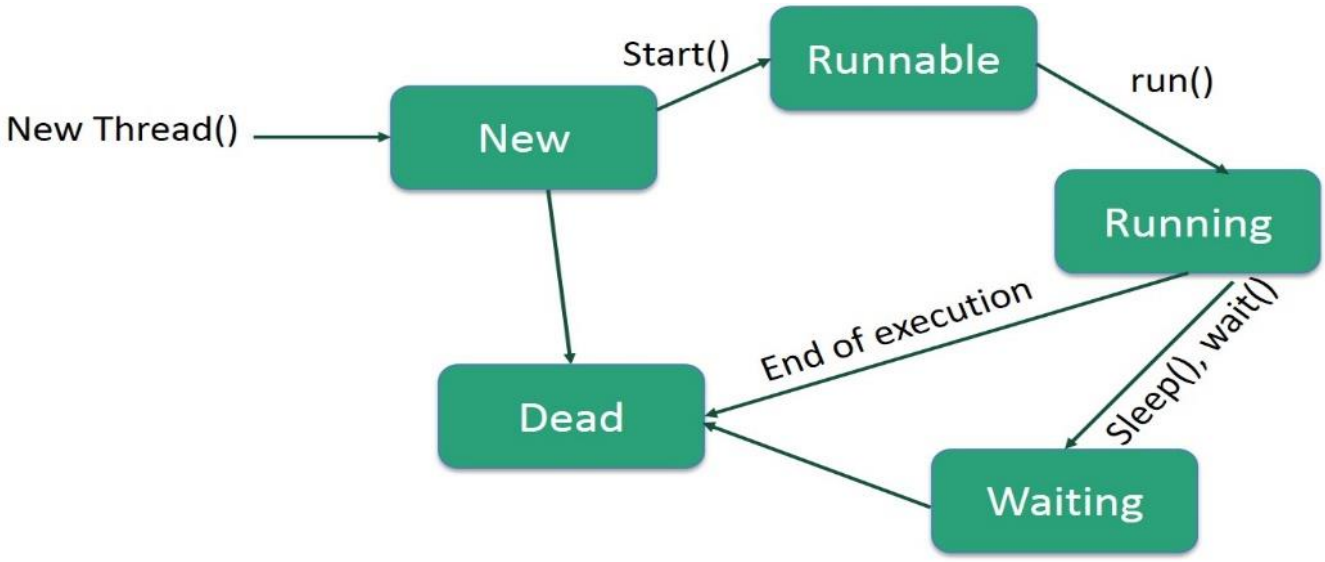
6. onStop():

- ✓ It is invoked when the activity is not visible to the user.
- ✓ It is followed by **onRestart()** when the activity is revoked from the background, followed by onDestroy() when the activity is closed or finished, and nothing when the activity remains on the background only.
- ✓ Note that this method may never be called, in low memory situations where the system does not have enough memory to keep the activity's process running after its onPause() method is called.

7. onDestroy()

- ✓ The final call received before the activity is destroyed.
- ✓ This can happen either because the activity is finishing (when finish() is invoked) or because the system is temporarily destroying this instance of the activity to save space.
- ✓ To distinguish between these scenarios, check it with **isFinishing()** method.

1.4. Thread Lifecycle in Android:



- A **thread** in Android is a separate path of execution that allows concurrent operations within an application.
- Every thread in Android has a **lifecycle**, representing the states it can be in from creation to termination.

1. New (Created):

- ✓ A thread is in the **New** state when an instance of Thread is created but start() has **not yet been called**.
- ✓ **Example:** Thread t = new Thread();
- ✓ **Key point:** The thread is only an object in memory, not yet executing.

2. Runnable:

- ✓ After start() is called, the thread enters the **Runnable** state.
- ✓ It is now eligible to run but may **wait for CPU scheduling**.
- ✓ **Example:** t.start();
- ✓ **Key point:** The thread scheduler decides when it will actually run.

3. Running:

- ✓ When the thread scheduler picks a thread from the Runnable pool, it enters the **Running** state.
- ✓ The run() method executes.
- ✓ **Key point:** Only **one thread runs per CPU core at a time**, but multiple threads can be running on multiple cores.

4. Blocked / Waiting / Timed Waiting:

- ✓ A thread may enter these states if it is **waiting for a resource** or **sleeping**:
  - **Waiting:** wait() – waits indefinitely until notified.
  - **Timed Waiting:** sleep(milliseconds) or join(milliseconds) – waits for a fixed time.
  - **Blocked:** Waiting to acquire a lock on an object for synchronization.

5. Terminated (Dead):

- ✓ After the run() method completes or the thread is **stopped**, it enters the **Terminated** state.
- ✓ **Key point:** A terminated thread **cannot be restarted.**