UNIT II – PROCESS AND MEMORY MANAGEMENT IN PRACTICE

Process and thread lifecycle in Linux and Android, POSIX threads, scheduling policies (SCHED_FIFO, CFS), Inter-process communication (pipes, shared memory, semaphores), Virtual memory, paging, segmentation, page faults, Memory allocation: slab allocator, /proc, top, vmstat

PROCESS AND THREAD LIFECYCLE IN LINUX AND ANDROID

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.

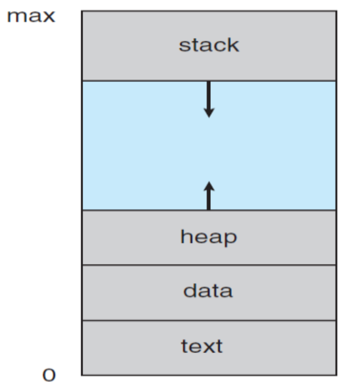


Figure Process in memory.

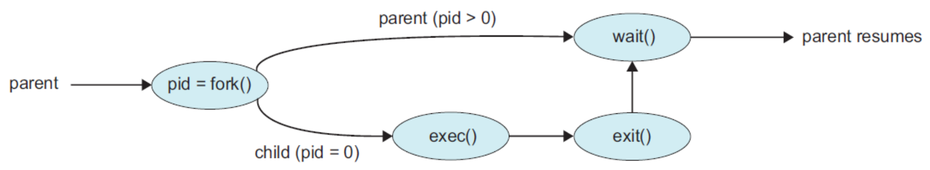
OPERATIONS ON PROCESSES

The processes can be executing concurrently and created and deleted dynamically. Operating systems must provide a mechanism for process creation and termination UNIX and Windows systems.

<u>Process Creation</u>

- During the execution of a process may create several new processes.
- The creating process is called a **parent process.**
- The new processes are called the **children of that process.**
- Each of new processes may create other processes.
- Each process is identified by a **process identifier** (or **pid**)

- pid an integer number.
- Index used to access various attributes of a process with in the kernel.
- when a child process is created, it needs resources to perform its task. (resources - CPU time, memory, files, I/O devices)
- Both parent and child process have their own or shared memory, I/O, address space, file etc.,
- when one process creates a new process, the identity of the newly created process is passed to the parent.
- Execution options:
  - The parent and child executes concurrently.
  - The parent waits until children terminated.
- Two address-space possibilities for the new process:
  - The child duplicates the parent process.
  - A new program loaded into it (child process).



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

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 LIFECYCLE IN LINUX

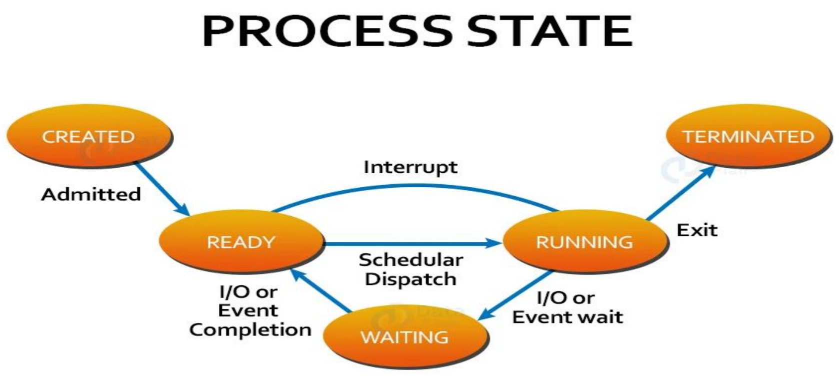
A **process** in Linux goes through the following lifecycle stages:

1. **Creation**
   - The process is being created.
   - Initiated by fork(), vfork(), or clone().
   - fork() creates a new child process (a copy of the parent).
   - exec() can replace the child's memory space with a new program.
2. **Ready**
   - The process is ready to run but waiting for CPU time.
   - Managed by the Linux scheduler.
3. **Running**
   - Process is executing instructions on the CPU.
4. **Waiting (Blocked)**
   - Waiting for I/O (disk, keyboard, etc.) or a resource.
   - This can be further divided into:

   ✓ **Interruptible Sleep:** The process can be easily woken up by signals.
   ✓ **Uninterruptible Sleep:** The process cannot be interrupted, typically waiting for hardware resources.

5. **Stopped**
   - The process has been paused or Temporarily halted (e.g., via SIGSTOP or CTRL+Z).
   - Can be resumed with SIGCONT.

6. **Terminated (Zombie)**
    - The process has finished but the parent has not read the exit status.
    - Becomes a "zombie" until the parent process calls wait() or waitpid().
7. **Cleaned Up**
    - After the parent collects the exit status, system resources are released.



A zombie process, also known as a defunct process, is a process that has completed its execution but remains in the process table because its parent process hasn't yet acknowledged its termination by collecting its exit status. This means the process has finished its work but its entry still exists, consuming a small amount of system resources.

Process Life Cycle:

1. Process Creation: A new process is created using fork() or exec().

2. Process Execution: The process starts executing and runs until it completes or is interrupted.

3. Process Sleep: The process can sleep or wait for a resource using sleep() or other system calls.

4. Process Wake-up: The process is woken up by a signal or when the resource becomes available.

5. Process Termination: The process completes execution or is terminated using exit().

Process Management:

1. Process Scheduling: The Linux kernel schedules processes for execution.

2. Process Prioritization: Processes can be prioritized using nice() or renice().

3. Process Synchronization: Processes can use synchronization primitives like semaphores and mutexes to coordinate access to shared resources.

4. Process Communication: Processes can communicate using IPC mechanisms like pipes, sockets, and shared memory.

System Calls:

1. fork(): Create a new process.

2. exec(): Replace the current process image with a new one.

3. exit(): Terminate a process.

4. wait(): Wait for a child process to complete execution.

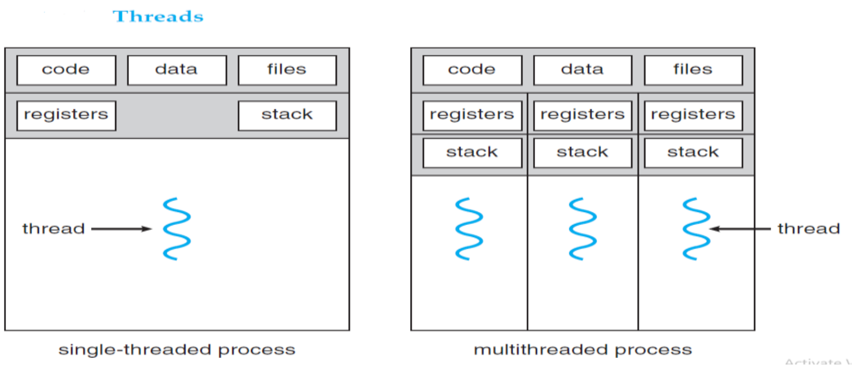
5. kill(): Send a signal to a process.

THREADS

A thread is a lightweight process. A thread is a lightweight unit of execution within a process. Threads allow a program to perform multiple tasks concurrently, improving efficiency and responsiveness.

1. Overview

- A thread is a basic unit of CPU utilization.
- It has a thread ID, a program counter, a register set, and a stack.
- The thread with in the same process shares its code section, data section and other operating-system resources, such as open files and signals.



- It's a single sequence of instructions that can run independently within the same program.
- Heavy weight processor has a single thread of control.
- Each process can have multiple threads
- If a process has multiple thread it can perform multiple task at a time.
- Most software applications that run on modern computers are multithreaded.
- Threads also play a vital role in remote procedure call (RPC) systems.
- RPCs allow interprocess communication using function or procedure calls.
- Most operating-system kernels are now multithreaded.
- Several threads operate in the kernel, and each thread performs a specific task.
- Specific task such as managing devices, managing memory, or interrupt handling.

Advantages of Threads

1. **Improved Performance and Concurrency**: Threads enable concurrent execution of tasks, which can lead to better utilization of CPU resources, especially on multiprocessor systems. This concurrency can improve the overall performance of applications by allowing multiple operations to proceed simultaneously.
2. **Faster Context Switching**: Switching between threads is generally faster than switching between processes because threads share the same memory space and resources, reducing the overhead associated with context switching.
3. **Resource Sharing**: Threads within the same process share code, data, and other resources, facilitating easier communication and data exchange between threads without the need for complex inter-process communication mechanisms.
4. **Responsiveness in Applications**: In interactive applications, such as graphical user interfaces (GUIs), threads can help

maintain responsiveness by performing background tasks without freezing the main interface.

5. **Efficient Utilization of Multiprocessor Systems**: Threads can be distributed across multiple processors or cores, allowing parallel execution and better utilization of hardware capabilities.

Disadvantages of Threads

1. **Complexity in Programming**: Writing multithreaded programs can be complex and error-prone. Developers must carefully manage synchronization to avoid issues such as race conditions and deadlocks.
2. **Debugging Difficulties**: Debugging multithreaded applications is challenging due to the non-deterministic nature of thread execution, making it hard to reproduce and fix bugs consistently.
3. **Shared Memory Issues**: While shared memory facilitates communication between threads, it also introduces risks. One thread can inadvertently modify shared data, leading to inconsistent or unexpected behavior.
4. **Resource Contention**: Threads competing for shared resources can lead to contention, where threads are blocked waiting for resources to become available, potentially degrading performance.
5. **Overhead in Managing Threads**: Although threads are lighter than processes, creating and managing a large number of threads can still introduce overhead, especially if not managed properly, potentially leading to resource exhaustion.

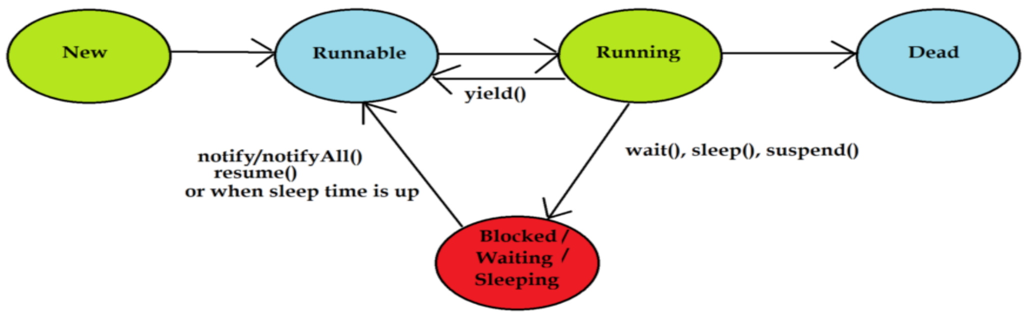
◆ THREAD LIFECYCLE IN LINUX

Threads in Linux are implemented using the **clone()** system call and **POSIX threads (pthreads)**.

1. **Created**
    - first stage in the lifecycle of a thread
    - Via pthread_create().

- Shares process resources (heap, memory space).
- Specify the code or function that the thread will execute

2. **Ready**
   - After a thread is created, it enters the "ready" or "runnable" state.
   - Thread is ready to run
   - waiting for the scheduler to allocate CPU time to them.
3. **Running**
   - when the scheduler selects a thread from the pool of ready threads and allocates CPU time to it, the thread enters the "running" state.
   - Actively executing instructions.
4. **Waiting/Blocked**
   - when they are waiting for some event to occur, such as I/O operations, synchronization primitives (e.g., locks or semaphores), or signals from other threads.
   - When a thread is blocked, it is not eligible to run until the event it is waiting for occurs.
5. **Terminated/Dead**
   - Once a thread has terminated, it enters the "dead" state.
   - Thread's resources (such as memory and handles) are deallocated
   - Dead threads cannot be restarted or resumed.
   - Completes execution via pthread_exit() or finishes the routine.
   - Can be joined by another thread using pthread_join().



Thread Lifecycle using Thread states

Thread Life Cycle:

1. <u>Thread Creation</u>: 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().

Thread Management:

1. Thread Scheduling: The Linux kernel schedules threads for execution.
2. Thread Synchronization: Threads can use synchronization primitives like mutexes and condition variables to coordinate access to shared resources.
3. Thread Communication: Threads can communicate using shared memory, pipes, or other IPC mechanisms.

POSIX Threads (pthreads):

1. pthread_create(): Create a new thread.
2. pthread_join(): Wait for a thread to complete execution.
3. pthread_exit(): Terminate a thread.
4. pthread_mutex_lock(): Lock a mutex to protect shared resources.
5. pthread_cond_wait(): Wait for a condition variable to be signaled.

PROCESS LIFECYCLE IN ANDROID

In Android, a process lifecycle refers to the stages an application's process goes through from its creation to its termination by the system. This lifecycle is managed by the Android OS and understanding it is crucial for developing robust and efficient apps. Key stages include creation, starting, resuming, pausing, stopping, and destruction. The OS may kill background processes to reclaim resources, which is a normal behaviour and not a crash. Android builds on the Linux process model but manages processes through **Activity Manager** and **Zygote**.

Android manages processes based on **importance hierarchy**

1. **Zygote Forking**

    - Android starts with the **Zygote process**, a pre-initialized process.
    - When a new app is launched, Zygote is forked to create the app process.
    - This improves startup speed (due to shared classes and preloading).
2. **Active Process**
    - Foreground app (activity/service) is actively interacting with the user.
3. **Visible Process**
    - Not foreground but still visible (e.g., an activity behind a dialog).
4. **Service Process**
    - Running a background service (e.g., music, download).
5. **Cached Process**
    - Not visible or used recently; kept in memory for faster reopening.
6. **Killed**
    - The system may kill cached or background processes when memory is low.

Process States:

1. **Foreground Process**: A process that is currently visible and interacting with the user.

2. **Visible Process:** A process that is visible but not interacting with the user.

3. **Service Process:** A process that is running a service and is not visible to the user.

4**. Background Process**: A process that is not visible and is not running a service.

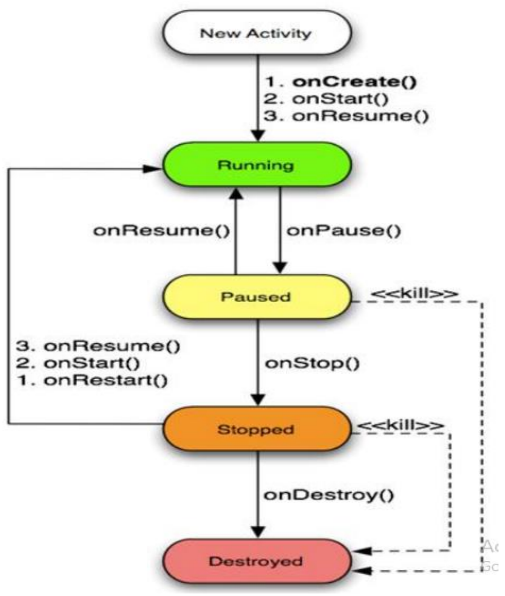
5. **Empty Process**: A process that is not running any components and is only kept in memory for caching purposes.

Process Life Cycle:

1. **Process Creation**: A new process is created when an application component (e.g., Activity, Service) is started.

2. **Process Running**: The process runs and performs its intended function.

3. **Process Termination**: The process is terminated when it is no longer needed or when system resources are low.



Android doesn’t use fork() and exec() in the traditional sense. Key stages and methods:

1. onCreate():

Called when the activity is first created. This is where you'll typically initialize UI components, load data, and set up the activity.

2. onStart():

Called when the activity is becoming visible to the user. It's a good place to start animations or other visual updates.

3. onResume():

Called when the activity will start interacting with the user. This is the active state where the user can interact with the app.

4. onPause():

Called when the activity is going into the background, either partially obscured or stopped. It's a good time to save user data, stop animations, and release resources that the activity doesn't need while in the background.

5. onStop():

Called when the activity is no longer visible to the user. This is a good time to release more resources that are not needed when the activity is not visible.

6. onDestroy():

Called before the activity is destroyed. This is the final callback before the activity is removed from memory.

Process Death:

- Android may kill background processes to free up resources. This is a normal part of the system's process management and is not considered a crash.
- When a process is killed, it may be restarted later, but the system will attempt to restore its state.
- Foreground processes (those actively used by the user) are less likely to be killed than background processes.

Android's Process Management:

1. Low Memory Killer: Android's Low Memory Killer (LMK) mechanism terminates processes when system memory is low.

2. Process Priority: Android assigns a priority to each process based on its state and importance.

3. OOM (Out-of-Memory) Adjuster: Android's OOM Adjuster mechanism adjusts the priority of processes based on their memory usage.

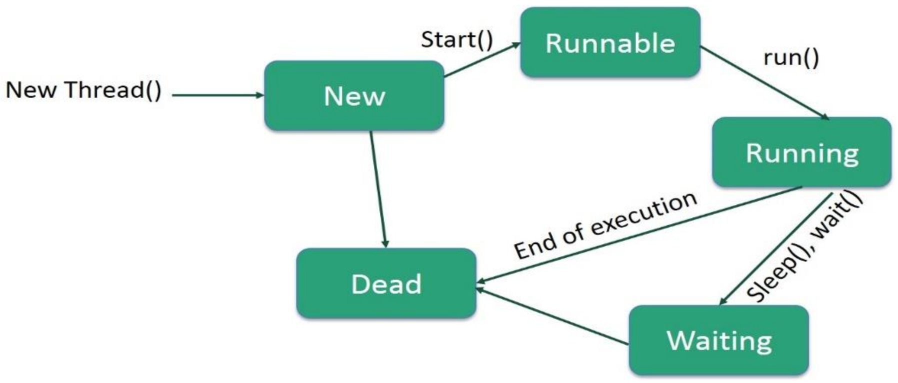
THREAD LIFECYCLE IN ANDROID

In Android, a thread's lifecycle involves several states: New, Runnable, Running, Blocked, and Terminated. The thread starts in the New state, transitions to Runnable when started, moves to Running when the CPU executes its code, and can enter Blocked for various reasons (waiting for resources, etc.). Finally, the thread ends its lifecycle in the Terminated state.

1. **New**
   - This is the initial state when a Thread object is created but before start() is called. The thread is not yet eligible for execution.
2. **Runnable**
   - After start() is called on a Thread object, it enters the Runnable state. Ready to run, waiting for CPU.
3. **Running**

- The thread is executing its run() method.

4. **Waiting/Blocked**
   - Thread is waiting for a resource or blocked due to some reason (eg. lock or I/O operation).
   - It remains blocked until the resource becomes available.
5. **Terminated**
   - Thread has completed its execution or has been terminated (encountered an unhandled exception).
   - It has reached the end of its lifecycle.



<u>**Thread Life Cycle Methods**</u>:

State	**Description**	**Associated Methods**
New	Thread is created but not started.	`Thread()` constructor
Runnable	`start()` is called; thread is ready to run and waiting for CPU scheduling.	`start()`
Running	Thread is actively executing its `run()` method.	`run()`
Blocked	Waiting to acquire a lock to enter a synchronized block or method.	`synchronized`, `wait()`
Waiting	Waiting indefinitely for another thread to perform an action.	`wait()`, `join()`

State	**Description**	**Associated Methods**
Timed Waiting	Waiting for a specified time.	`sleep()`, `join(timeout)`, `wait(timeout)`
Terminated	Thread has finished execution or was interrupted.	Thread ends or `interrupt()` is called

1.2 Benefits:-

There are four benefits in multithreaded programming

1. **Responsiveness: -**

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.

2. **Resource sharing**. Processes can only share resources through techniques such as shared memory and message passing.
3. **Economy**. 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. **Scalability.** The system's must be able to handle increase in workloads or demands without sacrificing performance or functionality and even without any redesign in the multiprocessor architecture.