## Process

- Process is program in execution.
- Process has multiple sections i.e. text, data, rodata, heap, stack. ... into user space and its metadata is stored into kernel space in form of PCB struct.
- PCB contains
  - id, exit status,
  - scheduling info (state, priority, time left, scheduling policy, ...),
  - files info (current directory, root directory, open file descriptor table, ...),
  - memory information (base & limit, segment table, or page table),
  - ipc information (signals, ...),
  - execution context, kernel stack, ...

## OS Data Structures:

- Job queue / Process table: PCBs of all processes in the system are maintained here.
- Ready queue: PCBs of all processes ready for the CPU execution and kept here.
- Waiting queue: Each IO device is associated with its waiting queue and processes waiting for that IO device will be kept in that queue

## Process Life Cycle

### Process States

- New
  - New process PCB is created and added into job queue. PCB is initialized and process get ready for execution.
- Ready
  - The ready process is added into the ready queue. Scheduler pick a process for scheduling from ready queue and dispatch it on CPU.
- Running
  - The process runs on CPU. If process keeps running on CPU, the timer interrupt is used to forcibly put it into ready state and allocate CPU time to other process.
- Waiting
  - If running process request for IO device, the process waits for completion of the IO. The waiting state is also called as sleeping or blocked state.
- Terminated
  - If running process exits, it is terminated.
- Linux: TASK_RUNNING (R), TASK_INTERRUPTIBLE (S), TASK_UNINTERRUPTIBLE (D), TASK_STOPPED(T), TASK_ZOMBIE (Z), TASK_DEAD (X)

## Process Creation

- System Calls
  - Windows: CreateProcess()
  - UNIX: fork()
  - BSD UNIX: fork(), vfork()
  - Linux: clone(), fork(), vfork()

## fork() syscall

- To execute certain task concurrently we can create a new process (using fork() on UNIX).
- fork() creates a new process by duplicating calling process.
- The new process is called as "child process", while calling process is called as "parent process".
- "child" process is exact duplicate of the "parent" process except few points pid, parent pid, etc.
- pid = fork();
  - On success, fork() returns pid of the child to the parent process and 0 to the child process.
  - On failure, fork() returns -1 to the parent.
- Even if child is copy of the parent process, after its creation it is independent of parent and both these processes will be scheduled sepeately by the scheduler.
- Based on CPU time given for each process, both processes will execute concurrently.

## How fork() return two values i.e. in parent and in child?

- fork() creates new process by duplicating calling process.
- The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent.
- Now fork() write 0 in execution context (r0 register) of child process and child's pid into execution context (r0 register) of parent process.
- When each process is scheduled, the execution context will be restored (by dispatcher) and r0 is return value of the function.

#### getpid() vs getppid()

- pid1 = getpid(); // returns pid of the current process
- pid2 = getppid(); // returns pid of the parent of the current process

## When fork() will fail?

- When no new PCB can be allocated, then fork() will fail.
- Linux has max process limit for the system and the user. When try to create more processes, fork() fails.
- terminal> cat /proc/sys/kernel/pid_max

### Orphan process

- If parent of any process is terminated, that child process is known as orphan process.
- The ownership of such orphan process will be taken by "init" process.

### Zombie process

- If process is terminated before its parent process and parent process is not reading its exit status, then even if process's memory/resources is released, its PCB will be maintained. This state is known as "zombie state".
- To avoid zombie state parent process should read exit status of the child process. It can be done using wait() syscall.

#### wait() syscall

- ret = wait(&s);
  - arg1: out param to get exit code of the process.
  - returns: pid of the child process whose exit code is collected.
- wait() performs 3 steps:
  - Pause execution parent until child process is terminated.
  - Read exit code from PCB of child process & return to parent process (via out param).
  - Release PCB of the child process.
- The exit status returned by the wait() contains exit status, reason of termination and other details.
- Few macros are provided to access details from the exit code.
  - WEXITSTATUS()

#### waitpid() syscall

- This extended version of wait() in Linux.
- ret = waitpid(child_pid, &s, flags);
  - arg1: pid of the child for which parent should wait.
    - -1 means any child.
  - arg2: out param to get exit code of the process.
  - arg3: extra flags to define behaviour of waitpid().
- returns: pid of the child process whose exit code is collected.
  - -1: if error occurred.

### exec() syscall

- exec() syscall "loads a new program" in the calling process's memory (address space) and replaces the older (calling) one.
- If exec() succeed, it does not return (rather new program is executed).
- There are multiple functions in the family of exec():
  - execl(), execlp(), execle(),
  - execv(), execvp(), execve(), execvpe()
- exec() family multiple functions have different syntaxes but same functionality.

### Types of Scheduling

#### Non-preemptive

- The current process gives up CPU voluntarily (for IO, terminate or yield).
- Then CPU scheduler picks next process for the execution.
- If each process yields CPU so that other process can get CPU for the execution, it is referred as "Co-operative scheduling".

### Preemptive

- The current process may give up CPU voluntarily or paused forcibly (for high priority process or upon completion of its time quantum)

## Scheduling criteria's

### CPU utilization: Ideal - max

```
* On server systems, CPU utilization should be more than 90%.
* On desktop systems, CPU utilization should around 70%.
```

### Throughput: Ideal - max

```
* The amount of work done in unit time.
```

### Waiting time: Ideal - min

```
* Time spent by the process in the ready queue to get scheduled on the CPU.
* If waiting time is more (not getting CPU time for execution) -- Starvation.
```

### Turn-around time: Ideal - CPU burst + IO burst

```
* Time from arrival of the process till completion of the process.
* CPU burst + IO burst + (CPU) Waiting time + IO Waiting time
```

### Response time: Ideal - min

```
* Time from arrival of process (in ready queue) till allocated CPU for first time.
```