Introduction

- A process is also considered as a unit of work in modern time-sharing system. A process includes **program counter, stack** (containing local variables), **data section** (for global variables) and sometimes **heap** (for dynamic allocation). A program is a **passive entity,** a file containing list of instruction stored on disk. In contrast, a process is an **active entity,** with a program counter specifying the next instruction to execute and a set of associated resources.
- As a process executes, it changes **state.** It can be
    1. **New :** The process is being created
    2. **Running :** Instructions are executed.
    3. **Waiting :** waiting for some event to occur
    4. **Ready :** Waiting to be assigned to a processor.
    5. **Terminated :** The process has finished execution
- Each process is represented in OS by **process control block** - also called **task control block.** It act like a repository of information which changes for process to process like **process state, program counter, CPU registers, CPU-scheduling information** (process priority, pointers to scheduling queues) **, Memory management information** (value of the base and limit registers and page tables …)**, Accounting information** (amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on)**, I/O status information** (I/O devices allocation, list of open files and so on).
- Now with incoming of multi-core OS, we can run multiple thread in parallel, allowing us to perform multiple tasks. On such systems, PCB is expanded to include information for each thread.

Process Scheduling

- To maximize the CPU utilization, we multiprogramming. In time sharing, CPU switches among processes very frequently. To meet such objectives **process scheduler** selects an available process.
- **Job queue** consist of all processes in the system. The processes that are residing in the main memory and are ready and waiting to execute are kept on a list called **ready queue.** We have **device queue,** which stores the list of processes waiting for I/O from device. Each device has its device queue. A new

process kept in ready queue waits there until it is dispatched. Once process is executing, one of several events occur-

1. Process could issue an I/O request and then be placed in I/O queue.
2. The process could create a new child & wait for the child's termination.
3. The process could be removed forcibly from the CPU, as a result of interrupt, and be put back in the ready queue.

- A process migrates among various scheduling queues. The selection process is carried out by the appropriate scheduler. In a batch system, more processes are submitted than can be executed. Hence they are spooled to a mass-storage device. The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The **short-term scheduler,** or **CPU scheduler,** selects from among the processes that are ready to be execute and allocates the CPU to one of them. The difference lies in the frequency of their execution. The LTS executes much less frequently and controls the **degree of multiprogramming.** The LTS should select a good process mix of **I/O-bound processes** (one that spends more of its time doing I/O) and **CPU-bound processes** (one using more of its time doing computation). On some systems, the LTS may be absent or minimal. In those systems we have a **medium-term scheduler,** which removes process from memory (and from active contention for the CPU) and thus reduce degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called **swapping.** It also helps in maintaining process mix.
- We know interrupt cause OS to change a CPU from its current task and to run a kernel routine. 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. The context is represented as PCB of the process. Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as **context switch.** The time required is completely a overhead and very much dependent on hardware and memory management of system.

Operation on Processes

- **Process Identifier** (or pid), which is typically an integer number, used to identify and index processes to access various attributes of a process within the kernel.
- In linux, **init** process serves as the root parent for all user processes with process id 1. When a process creates a child process, that child process receives resources either directly from OS or it may be constrained to a subset of the

resources of the parent process. Parent process either partition its resource or share it with child process. After creating child process, parent process either continues to execute concurrently with its children processes or it waits until some or all of its children have terminated. Similarly, we have two address space possibilities i.e. either child process is a duplicate of the parent process or child process has a new program loaded into it. In UNIX, a new process is created by the **fork()** system call. The new process consist of a copy of the address space of the original problem. The parent process can issue a **wait()** system call to move itself off the ready queue until termination of child. An example code showing the above concept for UNIX in C:

```
#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, ");
        } 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; // an indirect way to call exit
}
```

- A process terminating when it finishes executing its last statement and ask OS to delete it by using the **exit().** The process return the status value to its parent process. Parent can also execute a system call to kill the child process. The reason could be either child is overutilized the allocated resource or task assigned is no longer needed or if parent is exiting and OS does not allow a child to continue if its parent terminates. This third phenomenon is also called **cascading termination.** We can modify the wait call as **pid = wait(&status)** to get the info about the status and pid of child process terminated. When a process terminates, its resources are deallocated by the OS. However the entry in the

process table remains util parent process call the wait(). A process that has terminated, but whose parent has not yet called wait(), is known as **zombie process.** If parent terminates before it calls wait(), there by leaving its child process as **orphans.** Linux and UNIX address this scenario by assigning init process as the new parent to orphan process and init periodically invokes wait() call.

Interprocess Communication

- Processes executing concurrently in OS may be **independent** (if it cannot affect or be affected by the other process) or it can be **cooperating** (if can affect or be affected). Reasons for cooperation may include **information sharing, computation speedup, modularity** and **convenience.**
- The two fundamental models of interprocess communication :
  shared memory : A shared memory lies in the address space of creating process. Other process that wish to communicate using this shared-memory segment must attach it to their address space. The process are responsible for the form of data and location and synchronization between them. This produces the famous **producer consumer problem.** We either use **bounded buffer** or **unbounded buffer** to implement a solution to this problem.
  message passing : It is generally used in distributed environment, where the communicating components may reside on different computer connected by network. For it we need to have **send** and **receive** function from a message-passing facility. We can have several methods for logically implementing a link and the send()/receive() operations:
  1. **Direct** or **Indirect communication: I**n direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. It can be **symmetry** (where both sender and receiver send the others process id) and **asymmetry** (where sender send the id, but receiver receives the id and message). The problem is that if we change the process id, we need to review all the references to modify the identifier. In **indirect communication,** messages are send to and receive from **mailboxes,** or **ports.** Every mailbox is attached with a unique identification. Processes attached with the particular mailbox will receive the message. These mailboxes can be owned by process or OS. If by process help us to differentiate between receiver and sender by using address space of mailbox. If owned by OS, then OS should provide mechanism to manipulate them by process.

2. **Synchronization :** We know communication between process takes place through calls to send() and receive() primitives. Message passing may be **blocking** or **nonblocking.** We can have **blocking send** (sending process is blocked until the message is received by the receiving process or mailbox), **non-blocking send, blocking receive** and **non-blocking receive.**
3. **Buffering :** Whether communication is direct or indirect, messages exchanged by communication processes resides in a temporary queue. It can implemented in three ways:
    a. **Zero Capacity:** The queue has a maximum size of 0. In this case, the sender must be block until the recipient receives the message.
    b. **Bounded Capacity:** The queue has finite length **n;** and sender is blocked only after the queue is full.
    c. **Unbounded Capacity:** The queue length is potentially infinite; thus sender never blocks.