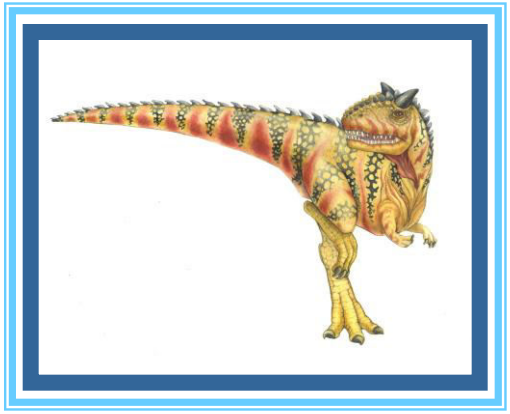
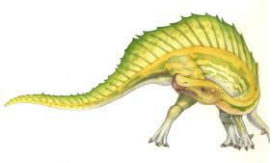
Processes



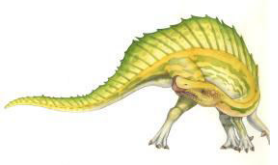
Processes

- Process Concept
- Process Scheduling
- Operations on Processes
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems



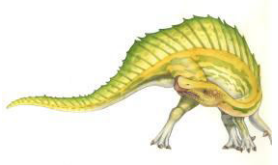
Operations on Processes

- *System must provide mechanisms for:*
  - *process creation*
  - *process termination*



Process Creation

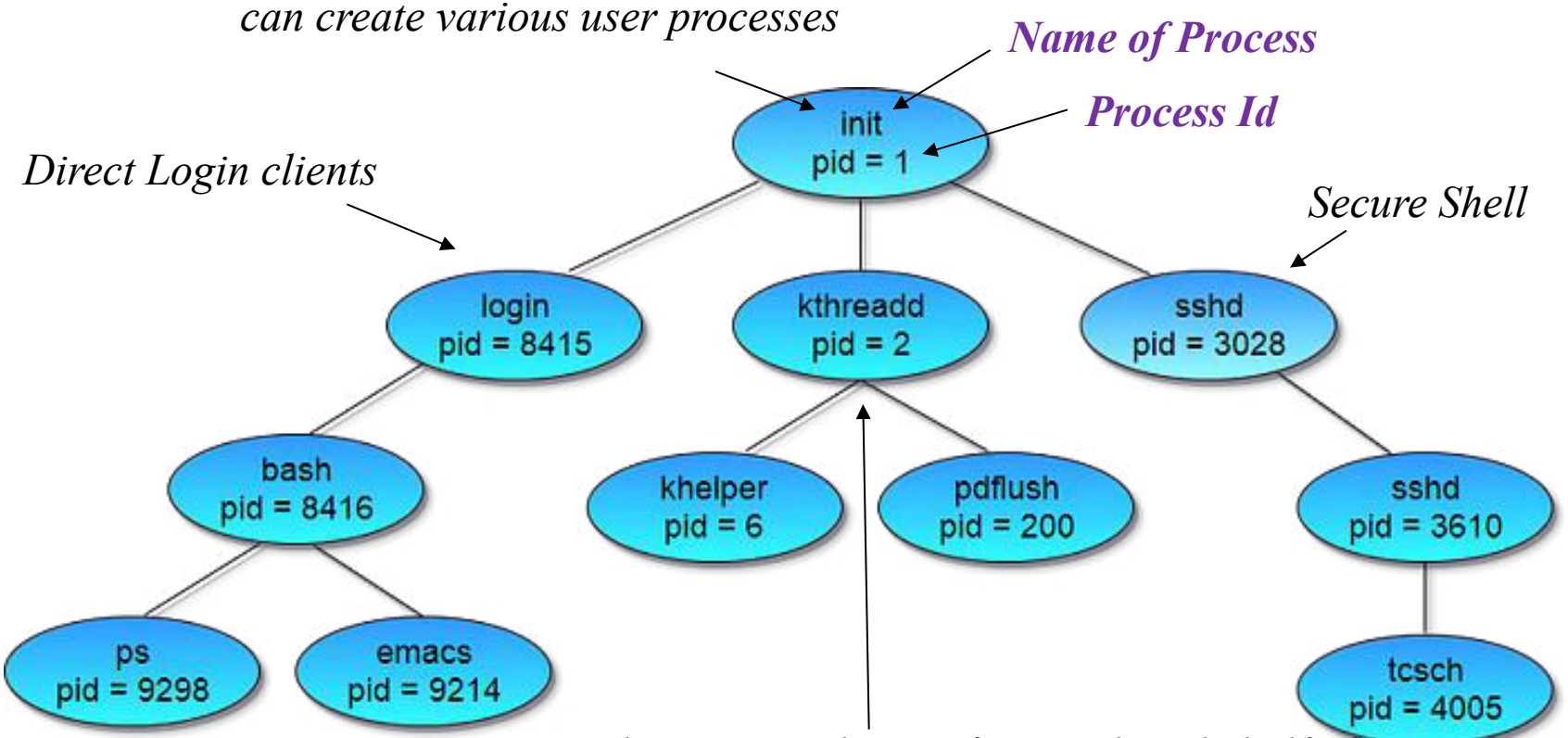
- During the course of execution a **process may create** several **new processes** during the course of execution.
- The creating process is called **parent process** and new process is called the **children** of that process.
- **Children process** in turn create other processes, forming a **tree** of processes
- Each process is given an integer identifier, termed as **process identifier, or PID**.



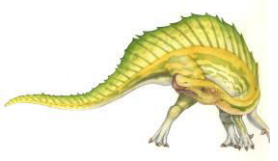
A Tree of Processes in Linux

Root Parent Process of all user processes: on boot

can create various user processes

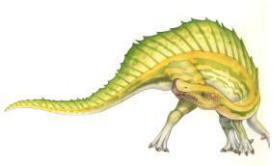


Create subprocesses that perform task on behalf of kernel



Process Creation: Resource sharing

- *Child process need certain resources (CPU time, memory files, I/O devices) to accomplish its task.*
- ***Parent and child share no resources**: Child process will directly get resources from OS*
- ***Children share subset of parent's resources:** Child process is constrained to a subset of the resources of the parent process.*
  - *Parent and children share all resources.*
  - *Parent partition its resources among children.*

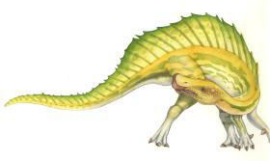


Process Creation

Initialization data passing options:

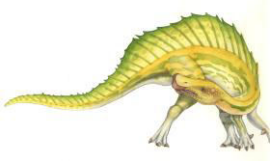
- *The parent process may pass along initialization data (input) to child process.*
- *OS may pass initialization data to child process.*

Example: Consider a process whose function is to display the contents of file (image.jpg) on the screen of terminal (output device).



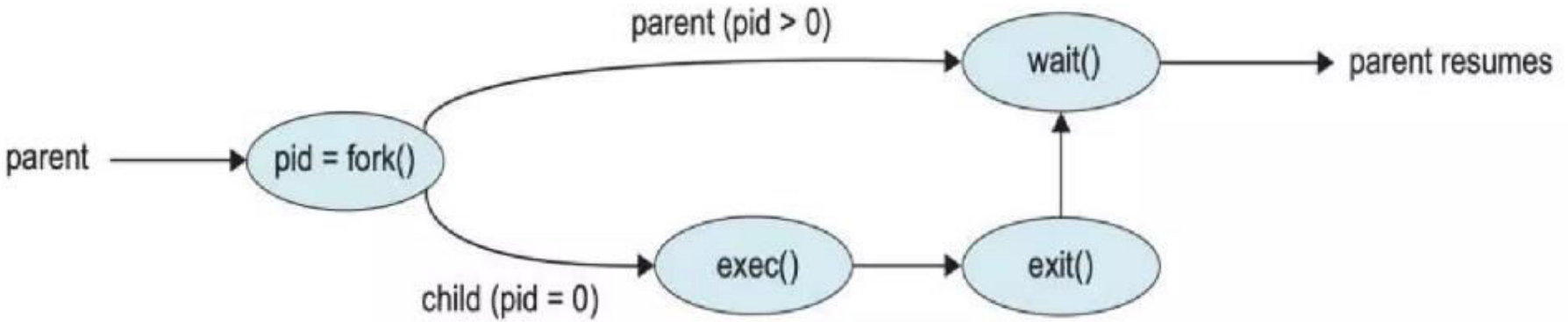
Process Creation

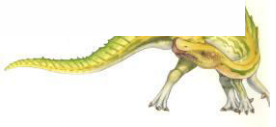
- ***Execution options***
  - *Parent and children execute concurrently.*
  - *Parent waits until some or all children have terminated.*
- ***Address space***
  - *Child process duplicate of parent, it has the same program and data as parent.*
  - *Child process has a new program loaded into it.*



Process Creation (Cont.)

- *UNIX examples*
  - ***fork()*** *system call creates a separate, duplicate process.*
  - ***exec()*** *system call is invoked, the program specified in the parameter to exec() will replace the entire process.*





Process Creation (Cont.)

- ***wait()*** *A call to wait() blocks the calling process until one of its child processes exits or a signal is received. After child process terminates, parent* ***continues*** *its execution after wait system call instruction.*

  pid = wait(&status); // process id of child

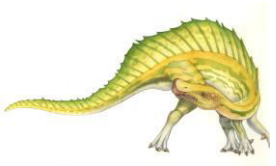
- ***exit()*** *The exit() is such a function or one of the system calls that is used to terminate the process. After the use of exit() system call, all the resources used in the process are retrieved by the operating system and then terminate the process.*

C Program Forking Separate Process

```
#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");
       return 1;
    }
    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;
}
```

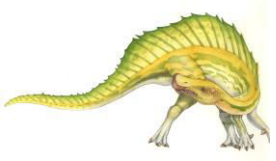
Process Termination

- *A process terminates when it executes its final statement and asks the operating system to delete it using the **exit()** system call.*
  - *At that point, the process may returns status value from child to parent (via **wait()**).*
  - *Process' **resources are deallocated** by operating system*



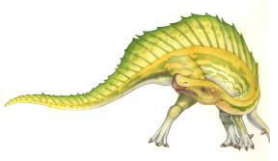
Process Termination

- *Termination in other circumstances:*
  - *Parent may terminate the execution of children processes using the* ***abort()*** *or* ***TerminateProcesses()*** *system call.*
  - *Some reasons for doing so:*
    - *Child has exceeded its usage of some of the resources that it has been allocated.*
    - *Task assigned to child is no longer required*
    - *The parent is exiting and the operating systems does not allow a child to continue if its parent terminates*



Process Termination

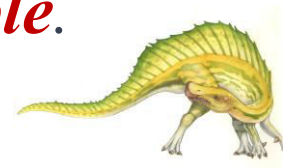
- *Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.*
  - ***Cascading termination:*** *All children, grandchildren, etc. are terminated.*
  - *The termination is initiated by the operating system.*



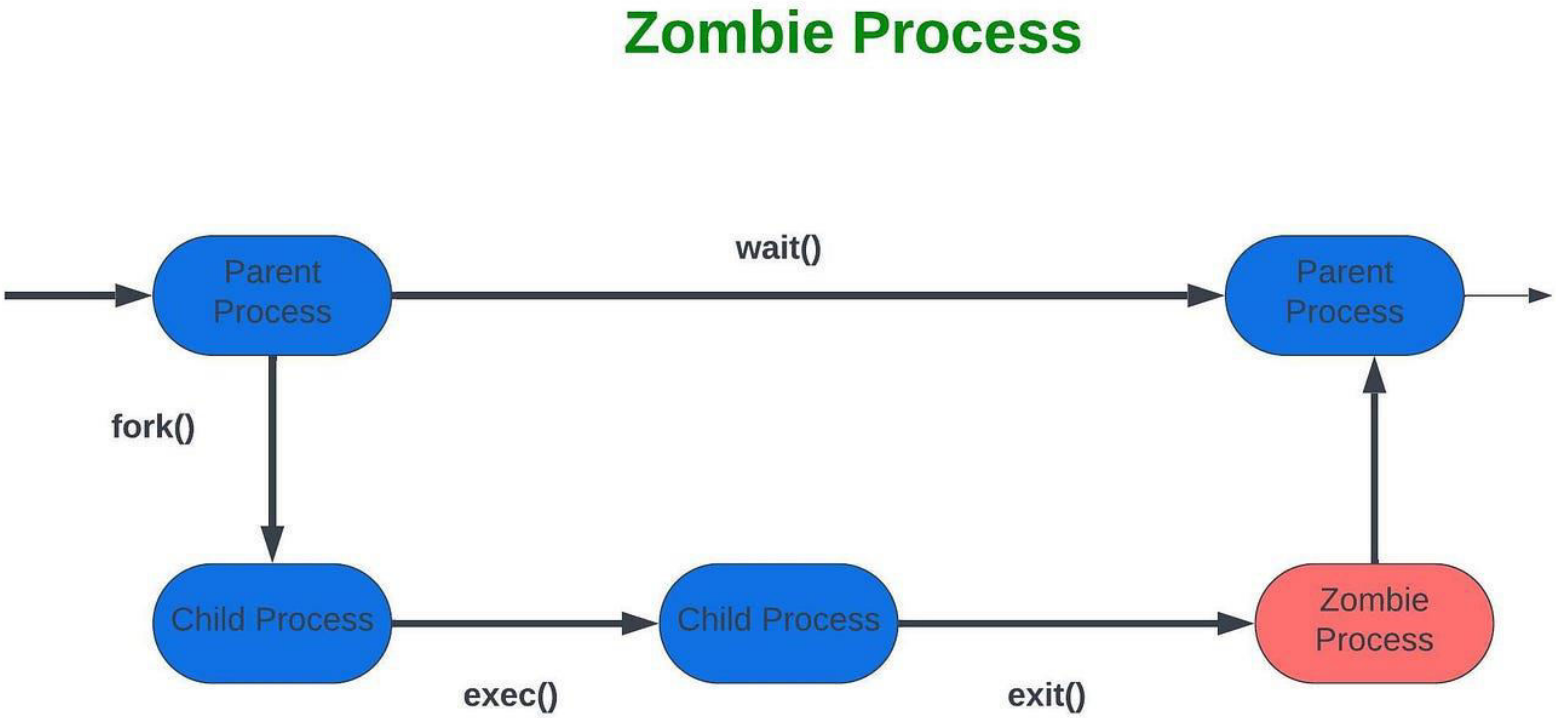
Process Termination: Zombie

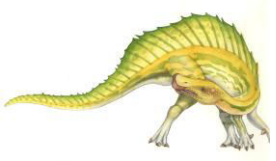
zombie died but there presence is still there

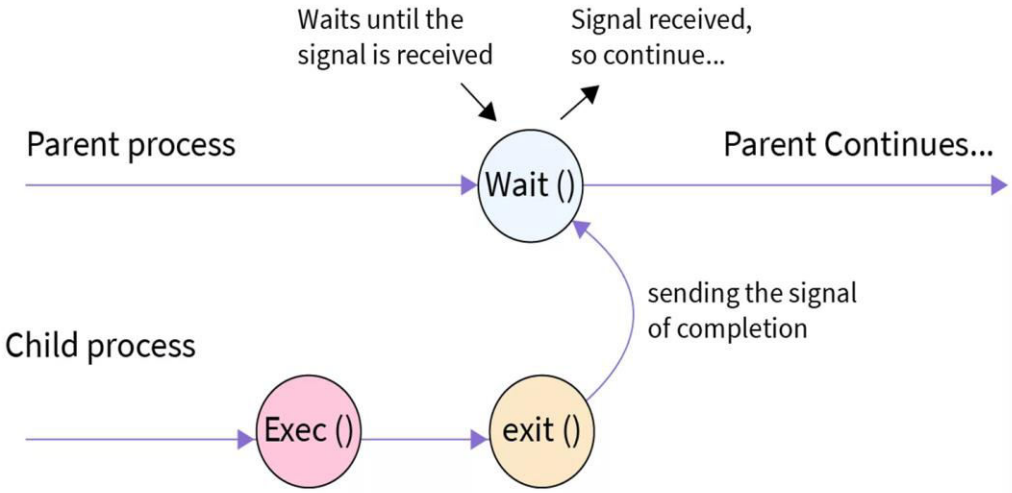
- ***Child Termination:*** *On termination child sends a termination signal (SIGCHLD) to its parent process. At this point, the child process enters a* ***"zombie"*** *state.*
- ***Parent's Responsibility****: It's the responsibility of the parent process to acknowledge the termination of its child process to OS. The parent can do this by calling the* ***wait()*** *system call.*
- ***Cleanup:*** *After getting acknowledgement the OS can remove the entry for the zombie process from the* ***process table****.*



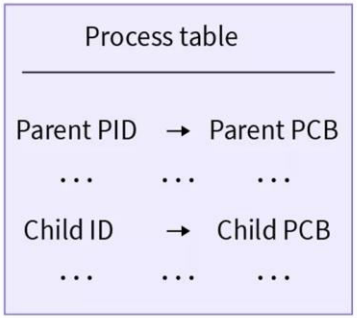
Process Termination: Zombie



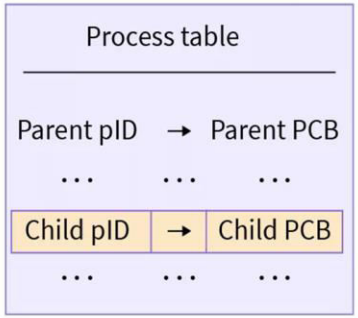




Before exit() system call



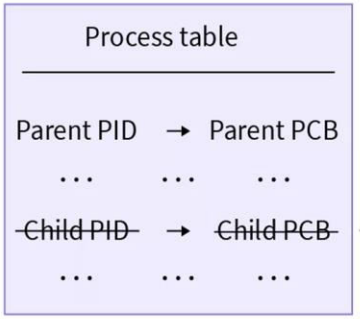
After exit() system call



Zombie process

its indicates that the child process is dome with it's execution & entered into 'Zombie State'

After wait() system call

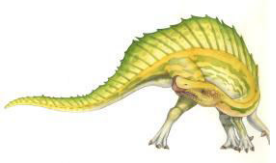


Zombie Gone

Once the wait() system call is called by the parent, it reads the exit status of the child process and reaps it from the process table

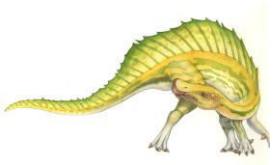
Process Termination: Zombie

- *Zombie Process is one of the process whose execution is done or completed but it entry is still exists in process table.*
- *Zombie processes are the processes which are died but exit status is not picked by the parent process.*
- *If the parent process fails to acknowledge the child's termination (perhaps because it's busy with other tasks and doesn't handle the SIGCHLD signal properly), the zombie process will remain in the system's process table.*



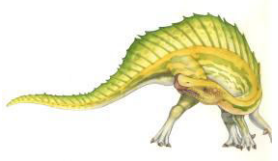
Process Termination: Zombie

- *A lot of zombie processes in os are harmful as The OS has one process table of finite size , so lots of zombie processes will results in a full process table.*
- *A full process table means that OS cannot create a new process when required and Zombie processes in os are of no use as the process has died but it's entry is occupying the space in memory.*
- *PIDs in os are finite , when all the PIDs have been consumed by Zombie Process , no new process can be created.*
- *Solution : Reboot the system*

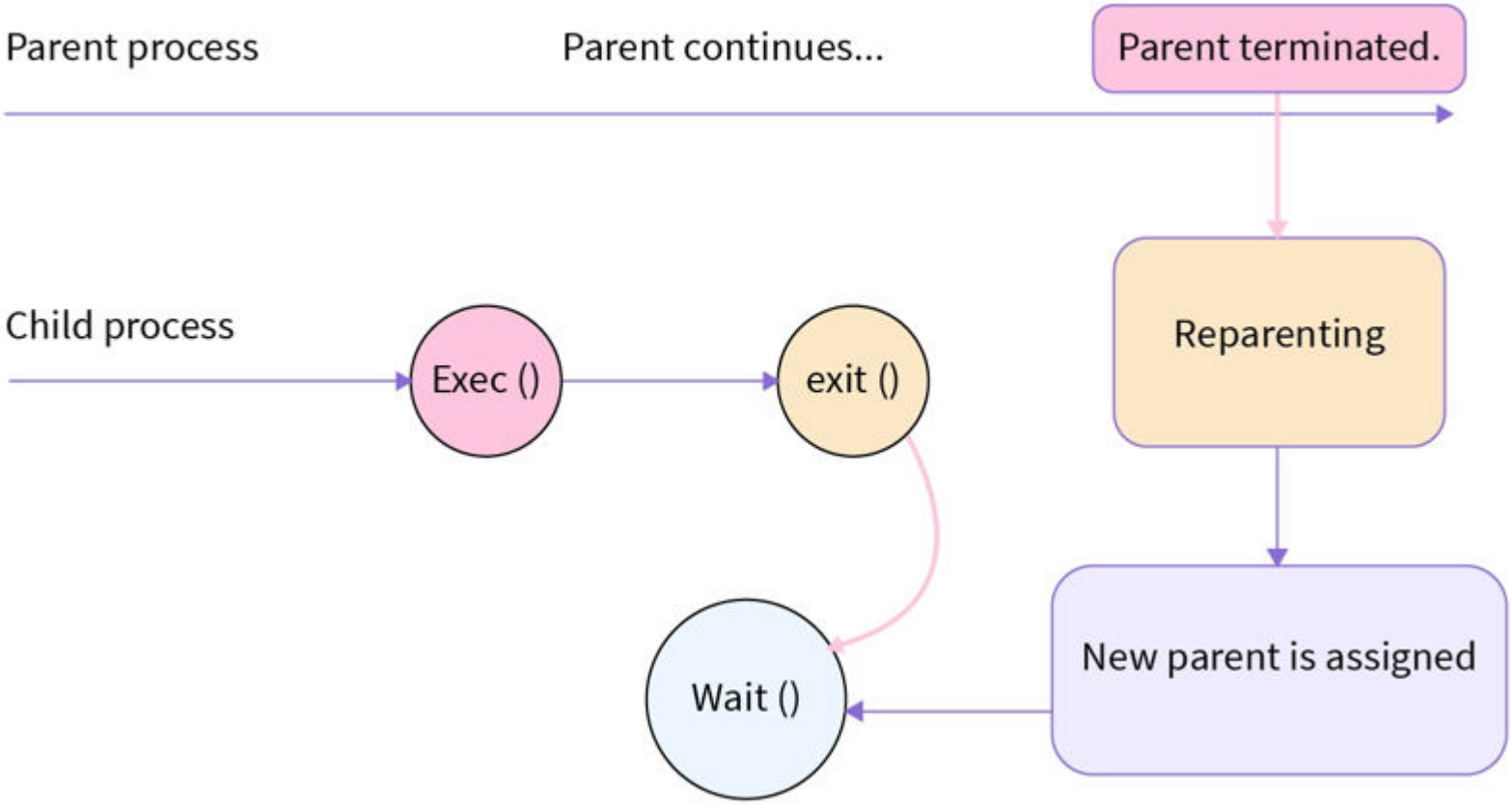


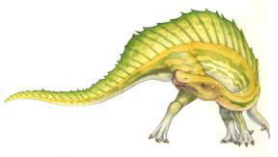
Process Termination: Orphan

- *A process which is executing (is alive) but it's **parent process has terminated (dead)** is called an orphan process.*
- *The orphan process in linux is adopted by a new process , which is mostly init process (pid=1) . This is called **re-parenting.***
- *Reparenting is done by the kernel.*
- *New parent process asks the kernel for cleaning of the PCB of the orphan process and the new parent waits till the child completes its execution.*



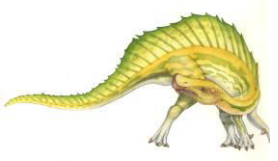
Process Termination: Orphan





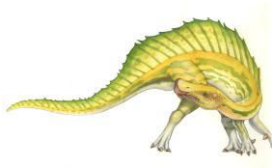
Process Termination: Orphan

```
int main()
{
        int pid;
        pid = fork();
        if(pid == 0)
        {
                printf("My parent's process ID is %d\n",getppid());
                sleep(30);
                printf("My parent's process ID is %d\n",getppid());
                exit(0);
        }
        else
        {
                sleep(20);
                printf("I am the parent, my process ID is %d\n",getpid());
        }
    return 0;
}
```



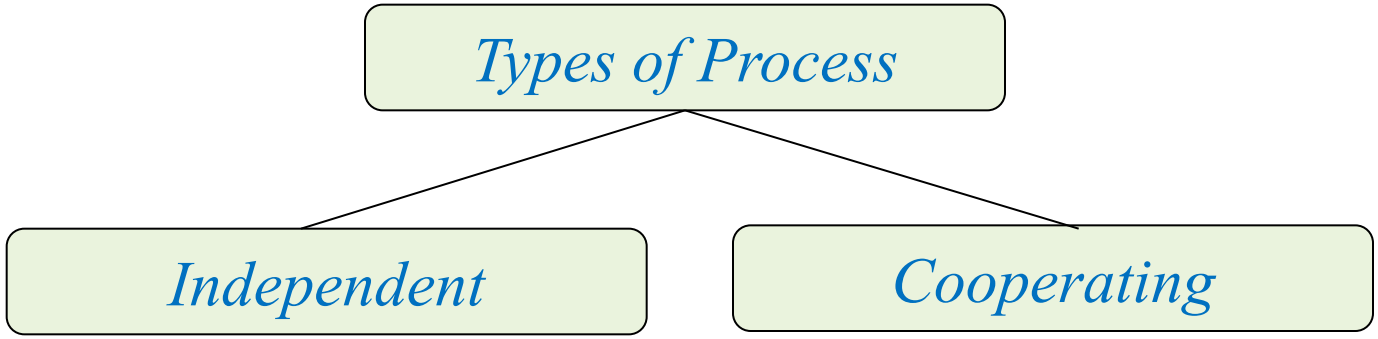
Process Termination: Orphan

- *In the example, the parent process sleeps for 20 seconds while the child process sleeps for 30 seconds.*
- *So after sleeping for 20 seconds the parent completes its execution while the child process is still there at least till 30 seconds.*
- *When the child process becomes an orphan process , the kernel reassigns the parent process to the child process.*
- *As a result the parent process id of the child process before and after sleep() will be different.*
- *Orphan processes in OS hold resources when present the system.*
- *Orphan processes in a large number can overload the init process and hang-up a system.*

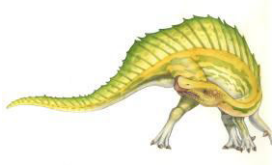


Interprocess Communication

- *Processes executing concurrently in the system may be either **independent process** or **cooperating process.***

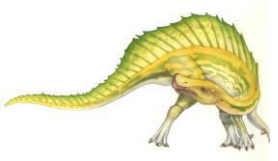


- ***Independent Process:** A process is independent if it **cannot affect** or **be affected by** the **other processes** executing in the system. Any process that doesn't share the data with any other process is independent.*



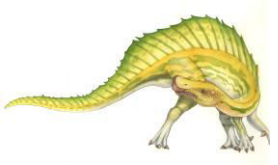
Interprocess Communication

- *Cooperating process can affect or be affected by other processes executing in the system.*
- *Any process that shares the data with any other process is cooperating.*
- *They need to communicate, share information, and coordinate their efforts to achieve a common goal.*



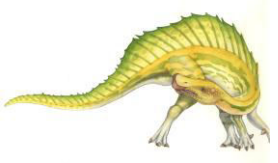
Interprocess Communication

- *There are several reasons for providing an environment that allow **process cooperation**:*
  - ***Information sharing***
  - ***Computation speedup***
  - ***Modularity***
  - ***Convenience***



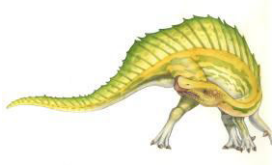
Interprocess Communication

- Cooperating processes require an ***Interprocess communication (IPC)*** mechanism, that will allow "***exchange of data and information***".
- Two ***fundamental models*** of IPC are:
  - ***Shared memory***
  - ***Message passing***

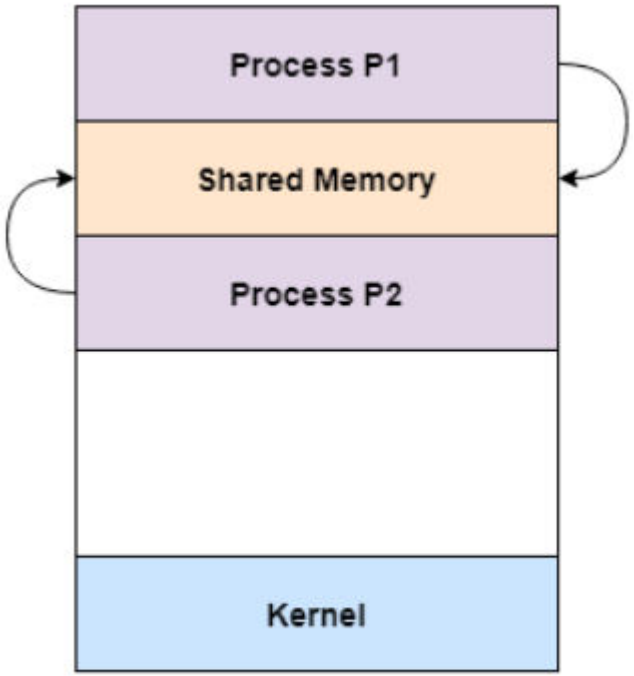


Interprocess Communication –Shared Memory

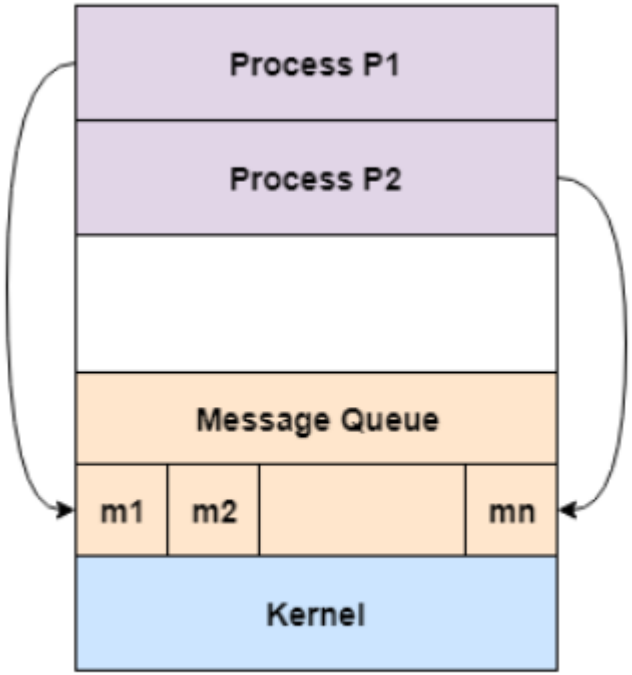
- ***Shared-memory model:*** a ***region of memory*** that is shared by cooperating processes is established. Process can then exchange information by ***reading and writing data to the shared region***.
- ***Message Passing model:*** Communication takes place by means of ***messages exchanged*** between the cooperating processes.



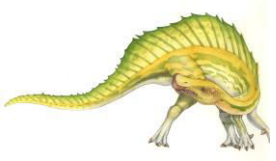
Communications Models



Shared Memory Model



Message Passing Model

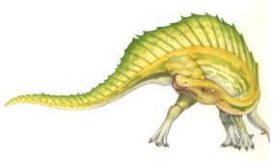


Interprocess Communication

- *Shared memory* ***can be faster than message passing****, since message passing systems are typically implemented using* ***system calls*** *and require more time consuming kernel interventions.*
- *In sheared memory, systems calls are required only to establish shared memory regions.*
- *Message passing is useful for* ***exchanging smaller amount of data.***
- *Message passing models (MPM) are* ***easier to implement*** *in distributed systems than shared memory.*
- *On multi-core* ***MPM*** *provides* ***better performance*** *than* ***SMM.*** *SMM suffers from cache coherency issues.*

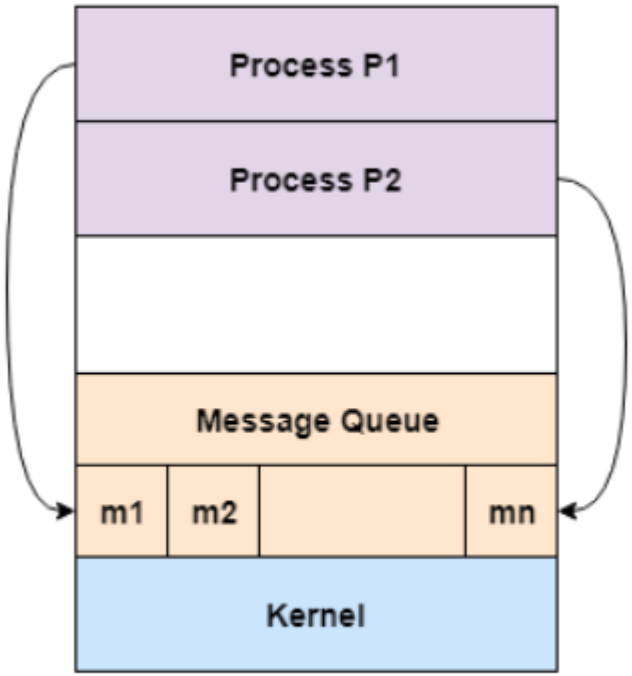
Shared Memory

- *Process can then exchange information by **reading and writing** data to the shared region.*
- *Typically, a **shared memory** region **resides** in the address space of the process creating the shared memory segment. The other process that wish to communicate using this shared memory segment must **attach it** to their address space.*
- *Normally, the operating system tries to prevent **one process from accessing another process's memory**. Shared memory requires that two or more processes agree to remove this restriction.*

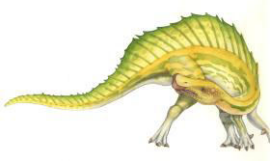


IPC – Message Passing

- *Mechanism for processes to communicate and to synchronize their actions without sharing the same address space.*
- *It is particularly useful in distributed environment. Where communicating processes are on different systems connected by network.*

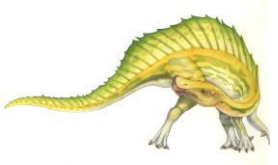


Message Passing Model



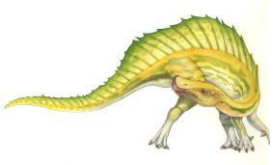
IPC–Message Passing

- *A message passing facility provides two operations:*
  - ***send(message)***
  - ***receive(message)***
- *The message size is either fixed or variable.*



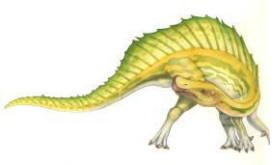
IPC–Message Passing

- ***Fixed size message:***
  - *System level implementation is straightforward.*
  - *While task of programming is more difficult.*
- ***Variable size message:***
  - *System level implementation is complex.*
  - *While task of programming is simpler.*



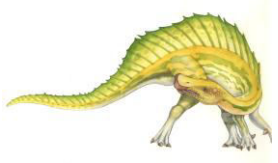
IPC– Message Passing

- *If processes P and Q wish to communicate, they must send messages to and receive messages from each other:*
  - *Establishment of a* ***communication link*** *between them*
  - *Exchange messages via send/receive*



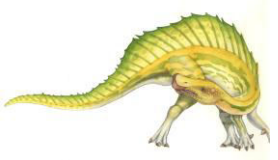
Message Passing (Cont.)

- *There are several ways for the implementation of a logical communication link and send/receive operations*
  - *Direct or indirect communication*
  - *Synchronous or asynchronous communication*
  - *Automatic or explicit buffering*
- ***Issues***
  - *Naming*
  - *Synchronization*
  - *Buffering*



Naming

- *Processes want to communicate must have a way to refer each other, they can use direct or indirect communication.*
- *Under direct communication each process that wants to communicate must explicitly name the recipient or sender of the communication*
  - ***send (P, message)*** *– send a message to process P*
  - ***receive(Q, message)*** *– receive a message from process Q*
  - Provides ***symmetry*** in addressing: (sen and rec both name each other)

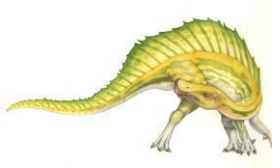


Naming

- ***Properties of communication link:***
  - *Links are established automatically between every pair of processes that want to communicate. Processes needs to know only each others identity.*
  - *A link is associated with exactly one pair of communicating processes, between each pair there exists exactly one link*
  - *The link may be unidirectional, but is usually bi-directional*

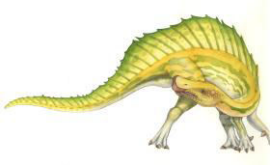
Direct Communication (Naming)

- ***Another Variant Asymmetry in addressing:*** *The sender typically knows the receiver's identity or address, but the receiver may not necessarily know the sender's identity or address. Only sender name the recipient.*
  - ***send (P, message)*** *– send a message to process P*
  - ***receive(id, message)*** *– receive a message from any process. The variable id is set to the name of a process with which communication taken place.*
- ***Methods :*** *Pipe, Socket, message queue, RPC*
- *Use cases: Broadcast*



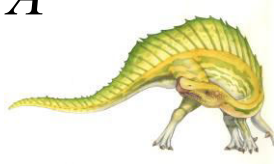
Direct Communication (Naming)

- ***Disadvantage:*** *Limited modularity of the resulting process identifiers : Changing the identifier of a process may necessitate examining all other process identifiers.*



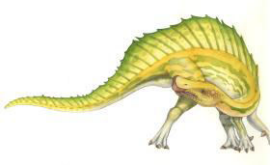
Indirect Communication

- *Messages are sent and received from **mailboxes** (also referred to as ports)*
  - *A mailbox can be viewed abstractly as an object into which message can be placed and removed by the processes.*
  - *Each mailbox has a **unique id**.*
  - *Processes can communicate only if they **share a mailbox**.*
  - ***send (A, message)** – send a message to mailbox A*
  - ***receive(A, message)** – receive a message from mailbox A*



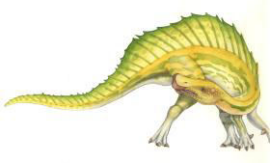
Indirect Communication

- ***Operations***
  - *create a new mailbox (port)*
  - *send and receive messages through mailbox*
  - *destroy a mailbox*



Indirect Communication

- ***Properties of communication link***
  - *Link established only if processes share a common mailbox*
  - *A link may be associated with many processes*
  - *Each pair of processes may share several communication links, each link corresponding to 1 mailbox.*
  - *Link may be unidirectional or bi-directional*



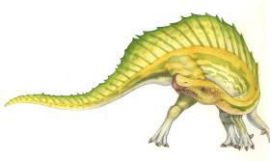
Indirect Communication

- **Mailbox sharing**
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?

- **Solutions**
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

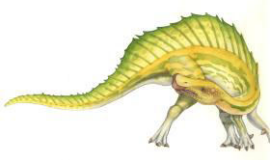
Synchronization

- *Communication between processes will take place through* ***send() and receive()*** *primitives. There are different design options for implementing each primitive.*
- *Message passing may be either* ***blocking and non-blocking*** *also known as* ***synchronous and asynchronous****.*



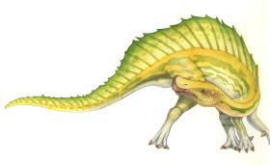
Synchronization

- *Message passing may be either blocking or non-blocking*
- ***Blocking*** *is considered* ***synchronous***
  - ***Blocking send*** *-- the sender is blocked until the message is received by receiving process or mailbox.*
  - ***Blocking receive*** *-- the receiver process is blocked until a message is available.*



Synchronization

- *Non-blocking is considered asynchronous*
  - ***Non-blocking send** -- the sender sends the message and resumes operation.*
  - ***Non-blocking receive** -- the receiver receives either*
    - *A valid message, or*
    - *Null message*



Buffering

- *Whether communication is direct or indirect, message exchanged by communicating process reside in **temporary queue**.*
- *This queue can be implemented in one of three ways:*
  1. ***Zero capacity** – The queue has maximum length zero. Link can not have any message waiting in it. The sender must block until recipient receives the message, (rendezvous). **Message system with no buffering.***
  2. ***Bounded capacity** – queue has finite length of n messages, at most n message can reside in it. Sender must wait if link full*
  3. ***Unbounded capacity** – infinite length, Sender never waits*