Contents

Representation of a Process by the OS: Process control block (PCB)

A process is basically ***a program in execution***. The execution of a process must progress in a sequential fashion.

A Process Control Block (PCB) is a ***data structure*** maintained by the OS for every process storing process-specific information.

The OS allocates a PCB for the process, space is allocated in memory for the process and the PCB is initialised. Newly started processes are added to the linked list used for the scheduling queue and all other associated files are updated i.e. the accounting files etc.

The OS then puts the PCB on the appropriate queue. As a process computes, the OS moves the PCB from queue to queue and eventually, when the process terminates, the OS deallocates the PCB.

The OS keeps all of the processes execution state in (or linked from) the PCB when the process isn't running

- PC, registers etc.
- When a process is unscheduled, the state is transferred out of the hardware into the PCB
- When a process is running, the state is spread between the PCB and the CPU

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

A PCB keeps all the information needed to keep track of a process. It is a data structure with many, many fields, including:

- **Process ID (PID):** When a process is created, each of the process in the operating system is assigned a unique process identifier (PID).
- **Process State:** The current state of the process i.e., whether it is **ready**, **running**, **waiting** etc.
- **Pointer:** A pointer to parent process.
- **CPU Scheduling Information:** Process priority and other scheduling information which is required to schedule the process.
- **Process Privileges:** This is required to allow/disallow access to system resources.

- **Program Counter:** Program Counter is a pointer to the address of the next instruction to be executed for this process.
- **CPU Registers:** Various CPU registers where process need to be stored for execution for running state i.e. Execution state for each thread
- **Memory Management Information:** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **Accounting Information:** This includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO Status Information:** This includes a list of I/O devices allocated to the process.

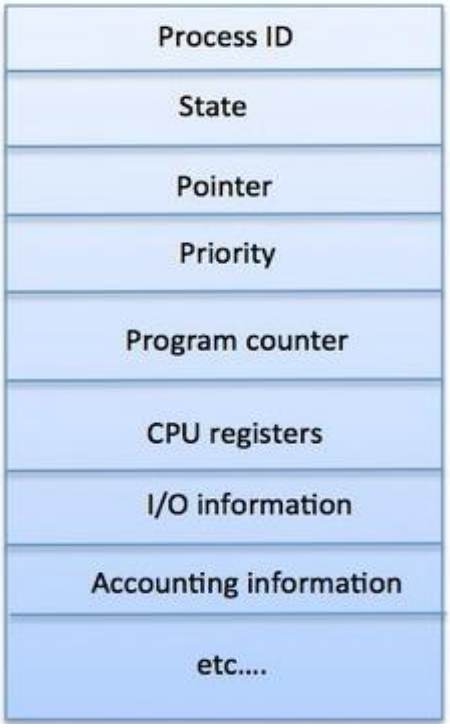


Figure 1 Simplified Version of PCB

The PCB in Linux OS is represented by the C structure task_struct (available in include/linux/sched.h) which contains all the necessary information for representing a process (the state of the process, scheduling and memory management information, list of open files and pointers to the processes parents and to any of its children… it contains approx.. 100 fields!)

Within the Linux kernel, all active processes are represented using a doubly linked list of task_struct and the kernel maintains a pointer to the process currently executing on the system.

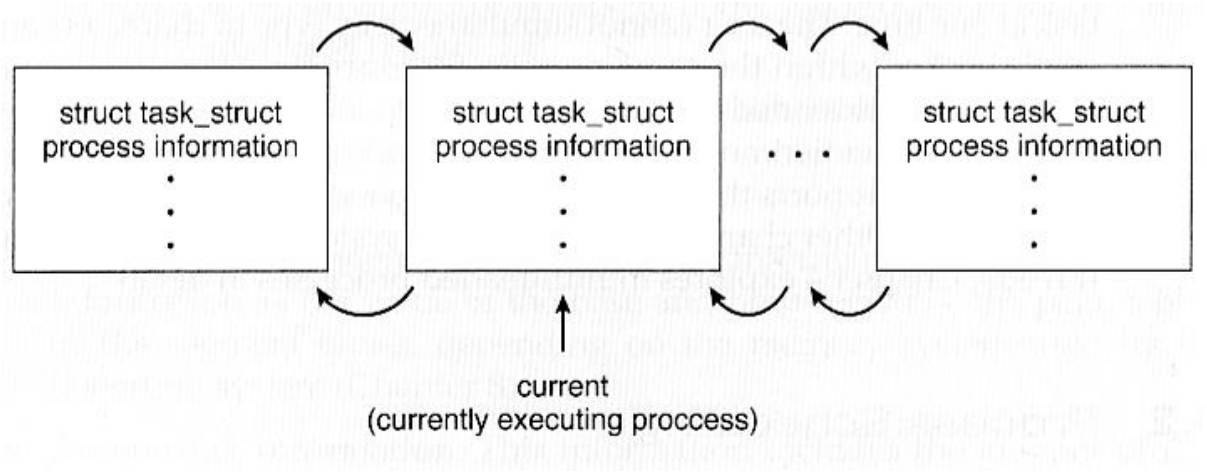


Figure 2 - Active processes in Linux

The process manager stores this job or process control block in the queues instead of the job or process itself:

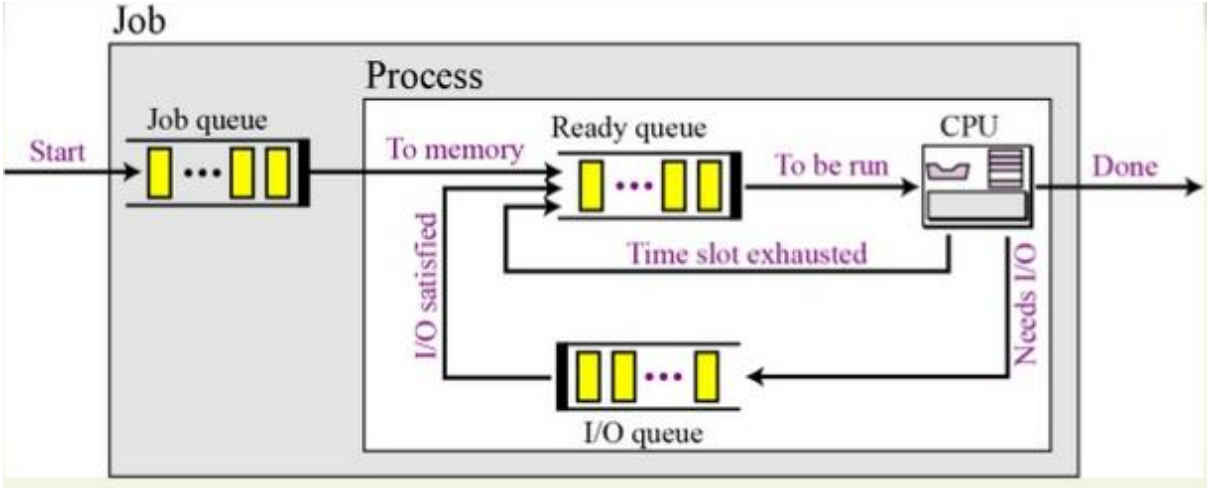
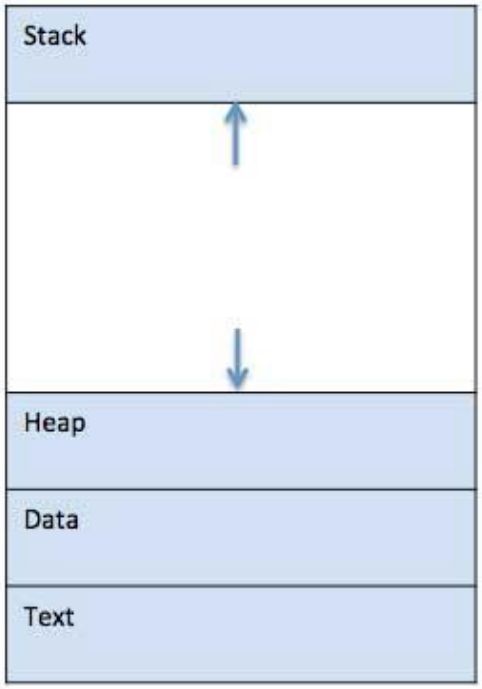


Figure 3: Queue for Process Management

Elements of a process in main memory

Each process is allocated space in memory.

When a program is loaded into the memory and it becomes a process, it can be divided into ***four*** process memory sections ─ stack, heap, text and data



1. **Stack:** The process Stack contains the temporary data (local variables) such as method/function parameters, return address and local variables. Space on the stack is reserved for local variables when they are declared, and the space is freed up when the variables go out of scope.
2. **Heap:** This is used for dynamic memory allocation (during a process run time), and is managed via calls to new, delete, free, etc.

Figure 4 Simplified layout of a process inside main memory

3. **Text:** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers. It's the compiled program code, read in from non-volatile storage when the program is launched
4. **Data:** This section stores the global and static variables, allocated and initialised prior to executing

NOTE from the figure here that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new will fail due to insufficient memory available.

States of a Process

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardised. When processes are swapped out of memory and later restored, additional information must also be stored and restored. Key among them are the program counter and the value of all program registers.

In general, a process can have one of the following five states at a time:

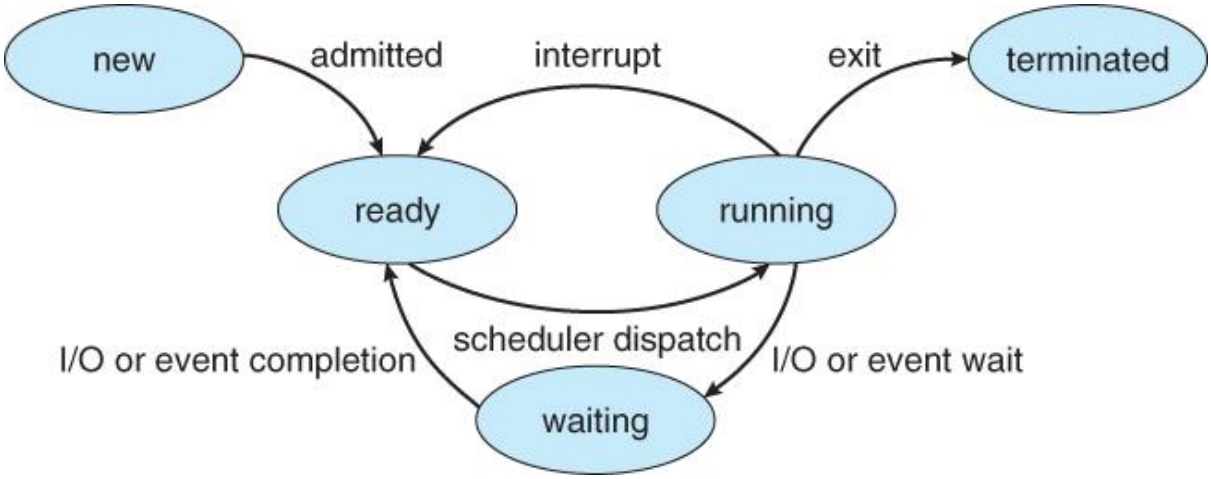


Figure 5 States of a Process

- **New:** This is the initial state when a process is first started/created.
- **Ready:** The process has all the resources available that it needs to run and is waiting to be assigned to a processor (waiting for CPU time) by the operating system so that they can run. Process may come into this state after **New** state or while running it but interrupted by the scheduler to assign CPU to some other process.
- **Running:** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
- **Waiting:** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, waiting for a file to become available, incoming network packet, or waiting for any other event.
- **Terminated or Exit:** Once the process finishes its execution, or it is terminated by the OS, it is moved to the terminated state where it waits to be removed from main memory.

NOTE: Some systems may have other states besides the ones listed here.

This will help to maximise CPU utilisation for multiprogramming.

> Multiprogramming systems explicitly allow multiple processes to exist at any given time, where only one is using the CPU at any given moment, while the remaining processes are performing I/O or are waiting. On a multiprogramming uniprocessor, the execution of multiple processes can be interleaved in time.

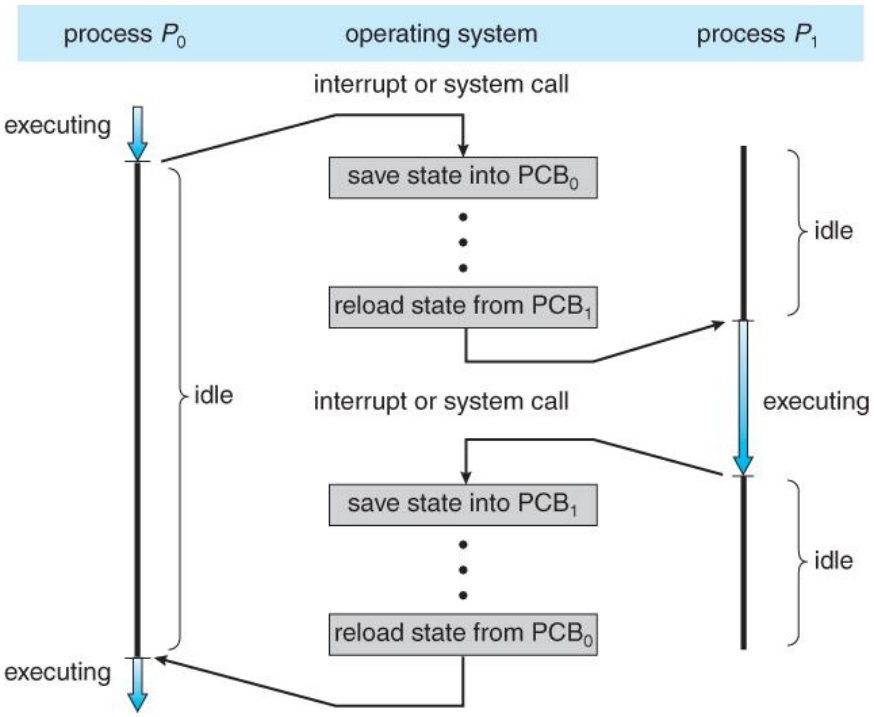


Figure 6: CPU switch from process to process

So, the overall anatomy of a process is:

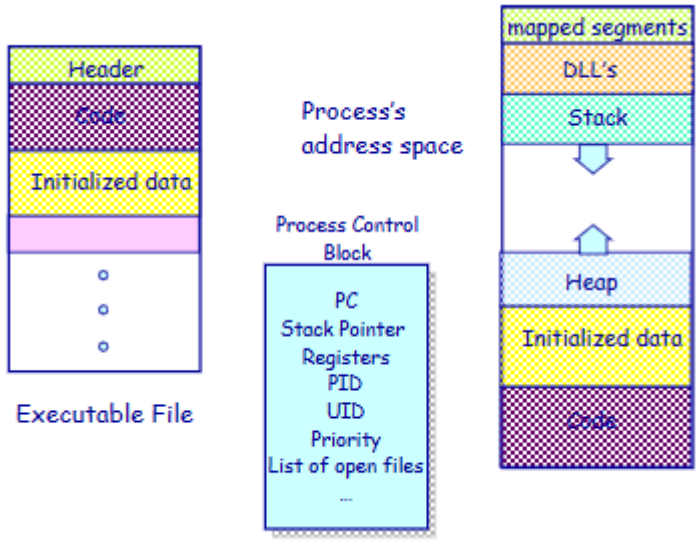


Figure 7 - Anatomy of a process

What causes a process to switch?

- Clock interrupt when the process has executed for the maximum allowable time slice
- I/O interrupt
- Memory fault - memory address is in virtual memory so it must be brought into main memory
- Trap – when an error or exception occurs and this may cause a process to be moved into **Exit** state
- Supervisor call, such as **open file**

When there is a change of Process State, the following should happen:

- Save context of processor including program counter and other registers
- Update the process control block of the process that is currently in the Running state
- Move process control block to appropriate queue – ready; blocked; ready/suspend
- Select another process for execution
- Update the process control block of the process selected
- Update memory-management data structures
- Restore context of the selected process

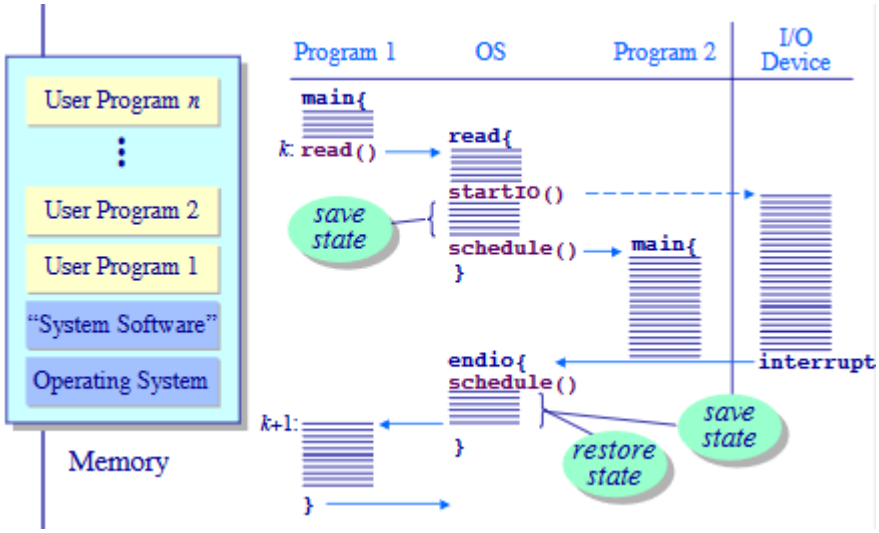


Figure 8 - Multiprogramming Process Context

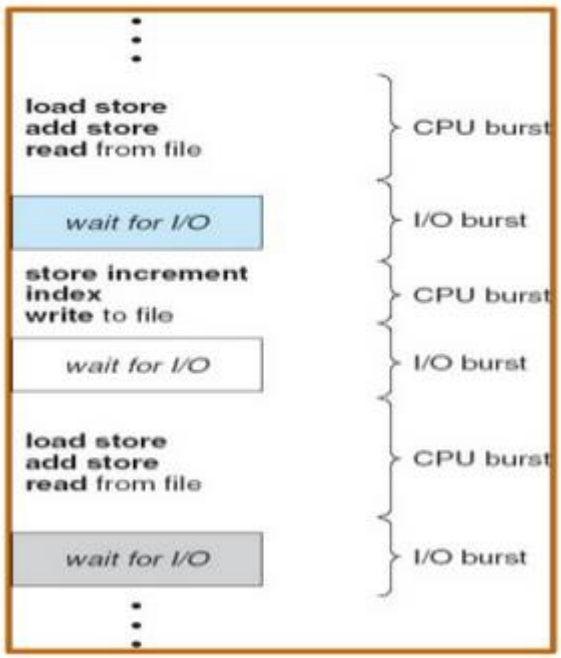
CPU Scheduling is the assignment of physical processors to processes which then allows processors to accomplish work.

Below are different ***time*** with respect to a process.

1. Arrival Time: Time at which the process arrives in the ready queue.
2. Completion Time: Time at which process completes its execution.
3. Burst Time: Time required by a process for CPU execution.
4. Turn Around Time: Time Difference between completion time and arrival time.
   Turn Around Time = Completion Time - Arrival Time
5. Waiting Time (W.T): Time Difference between turn around time and burst time.
   Waiting Time = Turn Around Time - Burst Time

Process execution consists of a cycle of a **CPU time burst** and an **I/O time burst** (i.e. wait)

Processes alternate between these two states and eventually the final CPU burst ends with a systems request to terminate execution.



The problem of determining ***when*** processors should be assigned and ***to which processes*** is called processor scheduling or CPU scheduling.

Objectives of Scheduling

In making decisions about the scheduling of processor work, a number of criteria can be taken into account by the operating system.

In choosing which algorithm to use, the properties of the various algorithms should be considered and these include the following:

- **CPU utilisation** – keep the CPU as busy as possible
- **Throughput** - Maximise the system throughput (the amount of work that a computer can do in a given period of time. The work can be measured in terms of the amount of data processed or transferred from one location to another by a computer, computer network or computer component)
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, *not* the output (for time-sharing environment). This must be "acceptable" for all programs, particularly for interactive ones
- Be fair to all users. This does not mean that all users must be treated equally. However they must be treated consistently, relative to the importance of the work being done.

- Degrade performance gracefully. If the system becomes overloaded, it should not collapse, but avoid further loading and/or temporarily reduce the level of service.
- Be consistent and predictable

For a particular process, the following may be taken into account in deciding **when** to schedule the job:

- Priority assigned to the job
- Class of job – i.e. real-time, batch, or online. Online users require a tolerable response time, while real-time systems often demand instant service.
- I/O or processor bound – i.e. whether the job uses mainly I/O time or processor time. This criterion is often important because of the need to balance the use of the processor and the I/O system. If the processor is absorbed in CPU-intensive work, it is unlikely that I/O devices are being serviced frequently enough to sustain maximum throughput.
- Resource requirements
- Resources used to date – e.g. the amount of processor time already consumed
- Waiting time to date – i.e. the amount of time spent waiting for service so far

Basic process manipulation include: creation, program loading, exiting, and so on

Process Manipulation in Linux

Creation and deletion: **fork**(), **exec**(), **wait**(), **exit**()

Process signalling: **kill**()

Process control: **ptrace**(), **nice**(), **sleep**()

The system creates the first process (**sysproc** in Unix)

The first process creates other processes such that:

- the creator is called the parent process
- the created is called the child process
- the parent/child relationships can be expressed by a process tree

In Unix, the second process is called **init**

- it creates all the gettys (login processes) and daemons
- it should never die
- it controls the system configuration (num of processes, priorities...)

Unix system interface includes a call to create processes, **fork**() system call

Unix's fork()

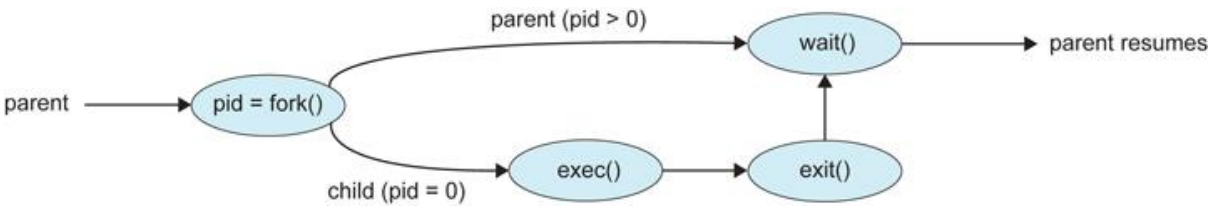


Figure 9 - Process creation using the fork( ) system call

fork() creates and initialises a new PCB and creates a new address space with a copy of the entire contents of the address space of the parent. This new PCB is then placed on the **ready** queue.

fork() creates a child process such that it inherits:

- identical copy of all parent's variables & memory
- identical copy of all parent's CPU registers (except one)

The **fork**() system call "returns twice", once into the parent, and once into the child

- returns the child's PID to the parent

- returns 0 to the child

So **fork**() is similar to saying "clone me" Simple implementation of **fork**():

- allocate memory for the child process
- copy parent's memory and CPU registers to child's

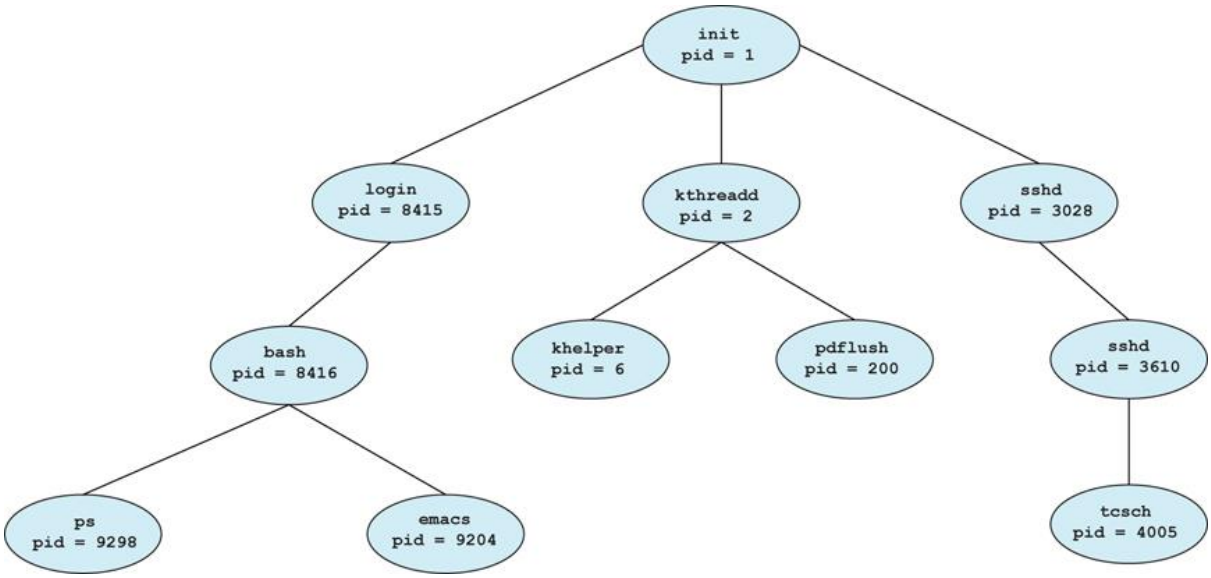


Figure 10 - A tree of processes on a typical Linux system

exec() v's fork()

In 99% of the time, **exec**() is called after calling **fork**()

The **exec**() call allows a process to "load" a different program and start execution **at _start.** It does not start a new process

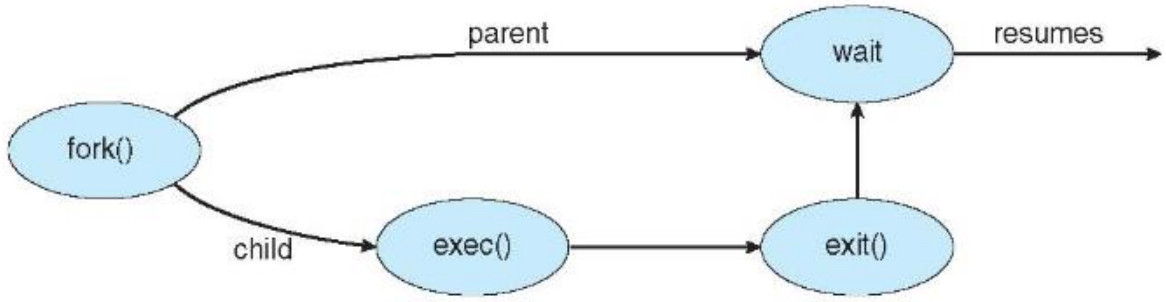


Figure 11 - exec() and fork()

It allows a process to specify the number of arguments (**argc**) and the string argument array (**argv**)

If the call is successful: it is the same process but it runs a different program!

There are two options for the parent process after creating the child:

1. Wait for the child process to terminate before proceeding. The parent makes a **wait**( ) system call, for either a specific child or for any child, which causes the parent process to block until the **wait**( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.
2. Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)

Properties of the fork/exec sequence

Two possibilities for the address space of the child relative to the parent:

1. The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behaviour of the **fork** system call in UNIX.
2. The child process may have a new program loaded into its address space, with all new code and data segments. This is the behaviour of the **spawn** system calls in Windows. UNIX systems implement this as a second step, using the **exec** system call.

vfork() or copy on write

The semantics of **fork**() say the child's address space is a copy of the parent's.

Implementing **fork**() that way is slow:

- Have to allocate physical memory for the new address space
- Have to set up child's page tables to map new address space
- Have to copy parent's address space contents into child's address space

(1) vfork()

- a system call that creates a process "without" creating an identical memory image
- sometimes called lightweight **fork**()
- child process is understood to call **exec**() almost immediately
- After the program finishes execution, it calls **exit**()

This system call:

- takes the "result" of the program as an argument
- closes all open files, connections, etc.
- deallocates memory
- deallocates most of the OS structures supporting the process
- checks if parent is alive

  - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the **zombie/defunct** state
  - If not, it deallocates all data structures, the process is dead

A child program returns a value to the parent, so the parent must arrange to receive that value

- The **wait**() system call serves this purpose:
- it puts the parent to sleep waiting for a child's result
- when a child calls **exit**(), the OS unblocks the parent and returns the value passed by **exit**() as a result of the wait call (along with the PID of the child)
- if there are no children alive, **wait**() returns immediately
- also, if there are zombies waiting for their parents, **wait**() returns one of the values immediately (and deallocates the zombie)

(2) copy-on-write

Retains the original semantics, but copies "only what is necessary" rather than the entire address space

By sharing resources this way it is possible to make significant resource savings in cases there are many separate processes all using the same resource, each with a small likelihood of having to modify it at all. Copy-on-write is the name given to the policy that whenever a task attempts to make a change to the shared information, it should first create a separate (private) copy of that information to prevent its changes from becoming visible to all the other tasks.

On fork():

- Create a new address space
- Initialise page tables with same mappings as the parent's (i.e., they both point to the same physical memory)
- No copying of address space contents have occurred at this point with the sole exception of the top page of the stack
- Set both parent and child page tables to make all pages read-only
- If either parent or child writes to memory, an exception occurs
- When exception occurs, OS copies the page, adjusts page tables, etc.