OPERATING SYSTEMS

Module 2 OS Principles

C. Oswald
VIT Chennai

Networking Principles and layered architecture

- Process Concept
- Structures
  - Process Control Block
  - Ready List
  - Threads
- Operations on Processes
  - Process Creation
  - Process Termination

ADDRESSED COURSE OUTCOME

- CO-2 : Apply various types of system calls and to find the stages of various process states.

Process

- An operating system executes a variety of programs:
  - Batch system – **jobs**
  - Time-shared systems – **user programs** or **tasks**
- **Process** – a program in execution; process execution must progress in sequential fashion

Process - Contd.,

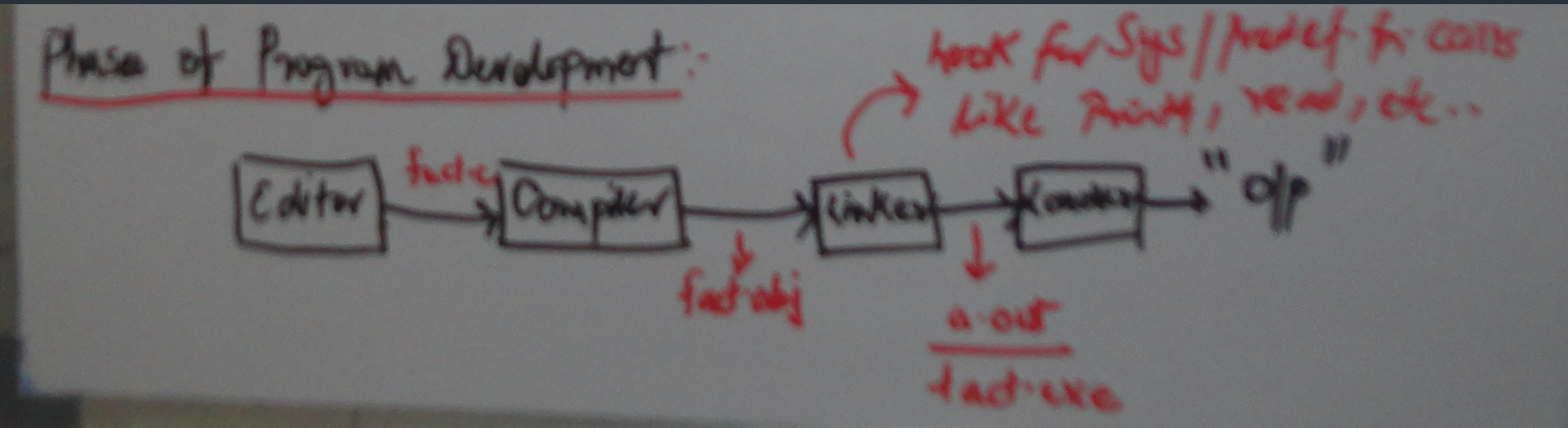
- Parts of Process
  - **Text Section:** Program code
  - Current activity including **program counter**, processor registers
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data section** containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process (Cont.)

- Program is termed as **passive** entity which stores on disk whereas process is **active** entity. Once the executable file gets loaded into memory then program will be termed as process
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program

PROCESS CONCEPT - Intro

- **PROCESS NOTION**
- Program in execution- Programs RUNS!!! – understand the terminology !
- Active Program (Process) v/s Binaries-Exe's on HDD (passive!)
- Program here is not Source Code!
- **Phases of PROGRAM Development (BINARY)**



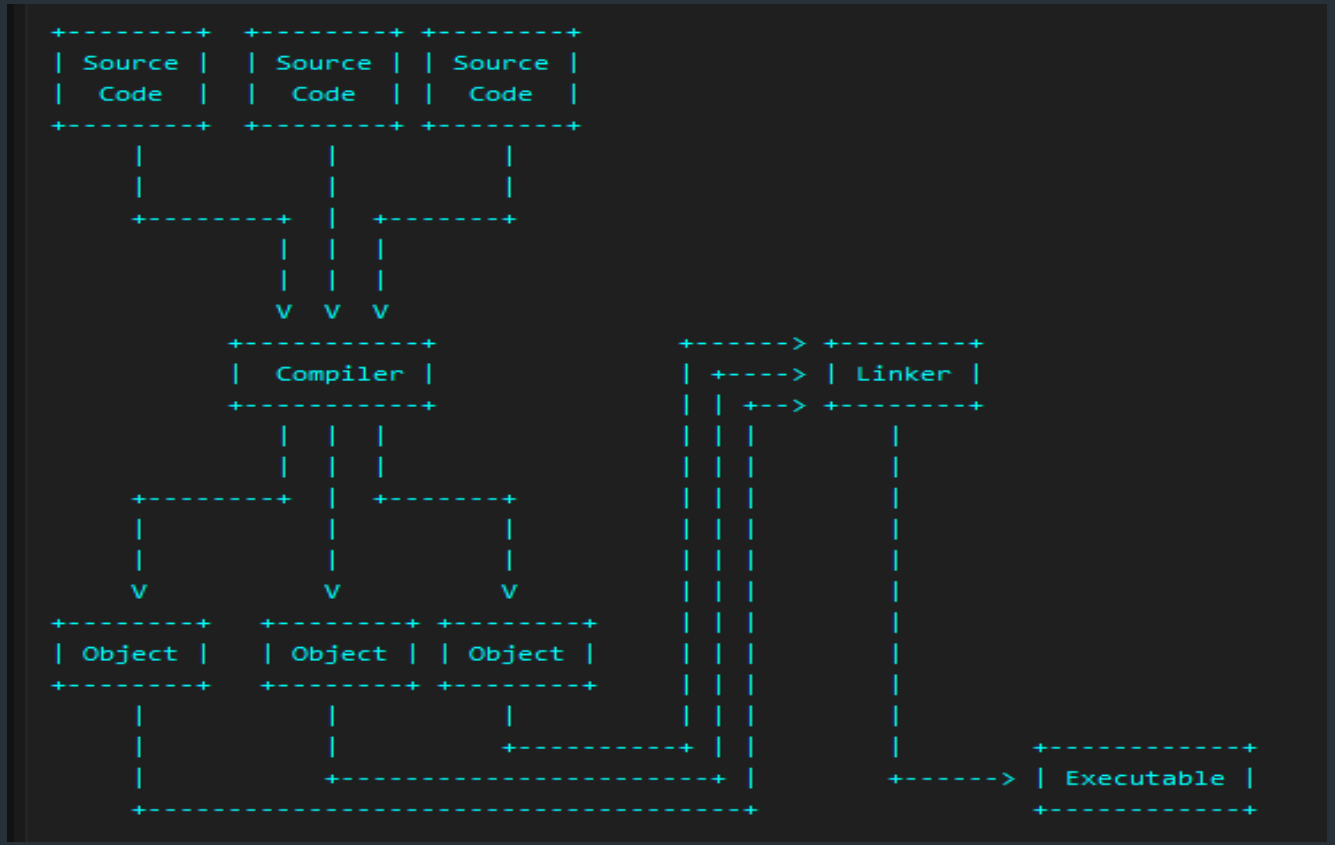
Binary Development Phases

- **Revisit of the Classical Hello World Example!**

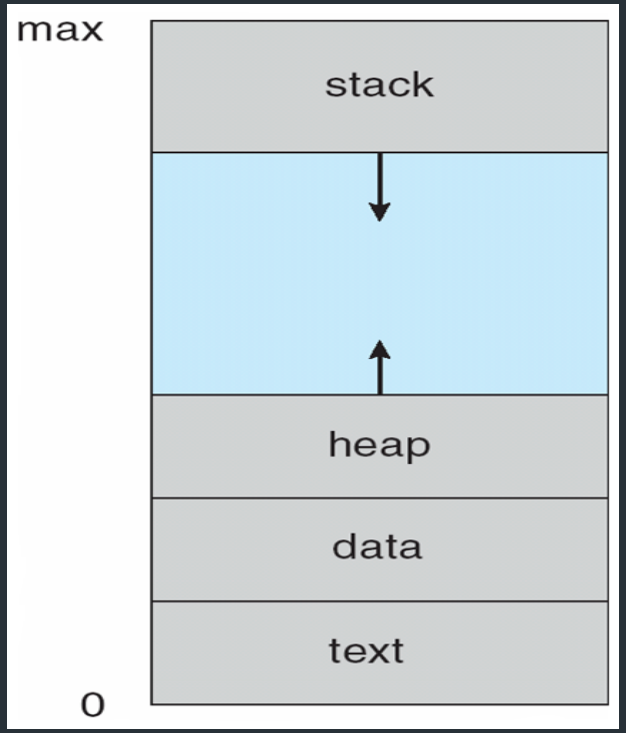
```
#include <stdio.h>
int main() {
printf (“Hello World \n”);
return 0;  }      /* Assume Stored in first.c */
```

- Editors in Linux – gedit, emacs, vim, etc.
- these are again programs later become processes
- Each block in the phases is a program - process
- Header File Inclusion Why ? – System v/s User Defined Headers
- What do They Contain – Function Declaration or Function Definition ; int add (int , int) ; v/s
- int add (int x, int y) { return x+y;}
- Benefit of Having Declaration and Then Definition
- Where is Printf Defined ? - Role of Linker Justification

A Visual Understanding of the phases



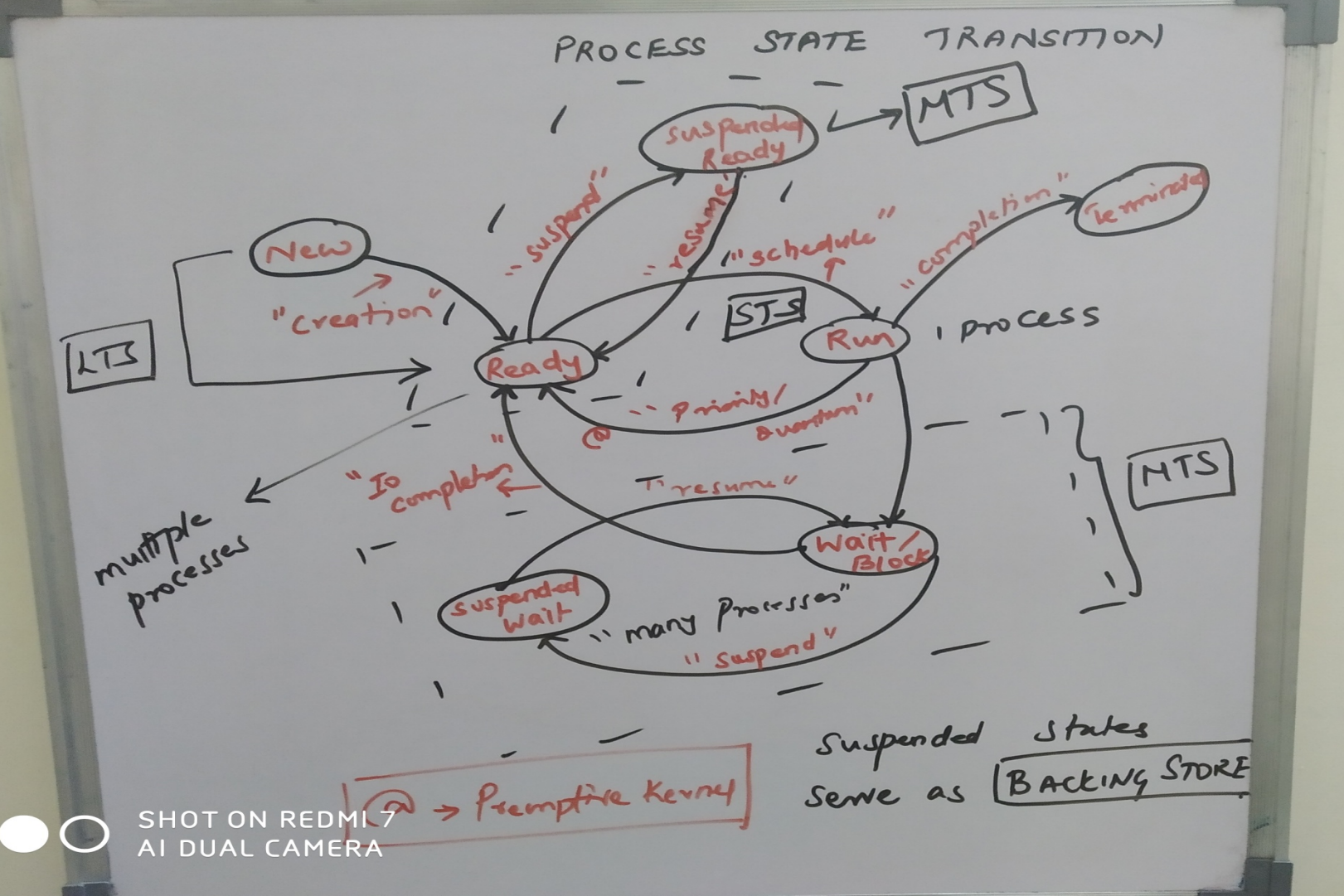
Process in Memory



Source: Operating_System_Concepts,_8th_ Edition

Process State

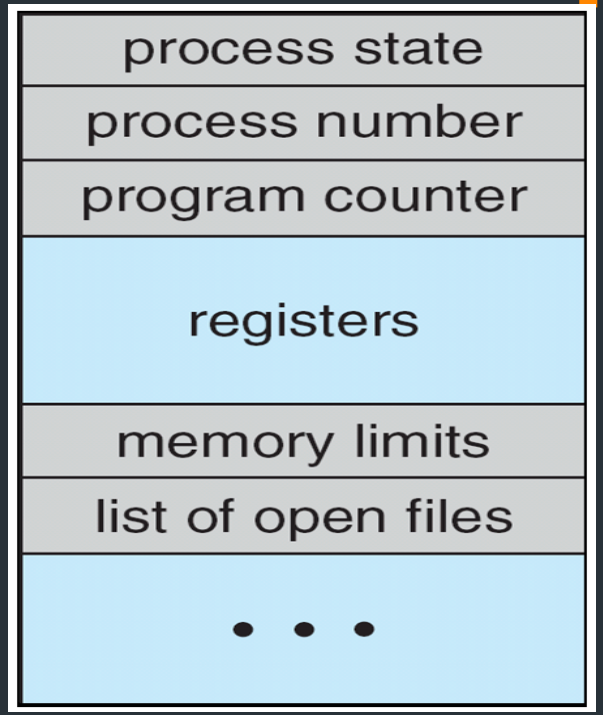
- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution



Process Control Block (PCB)

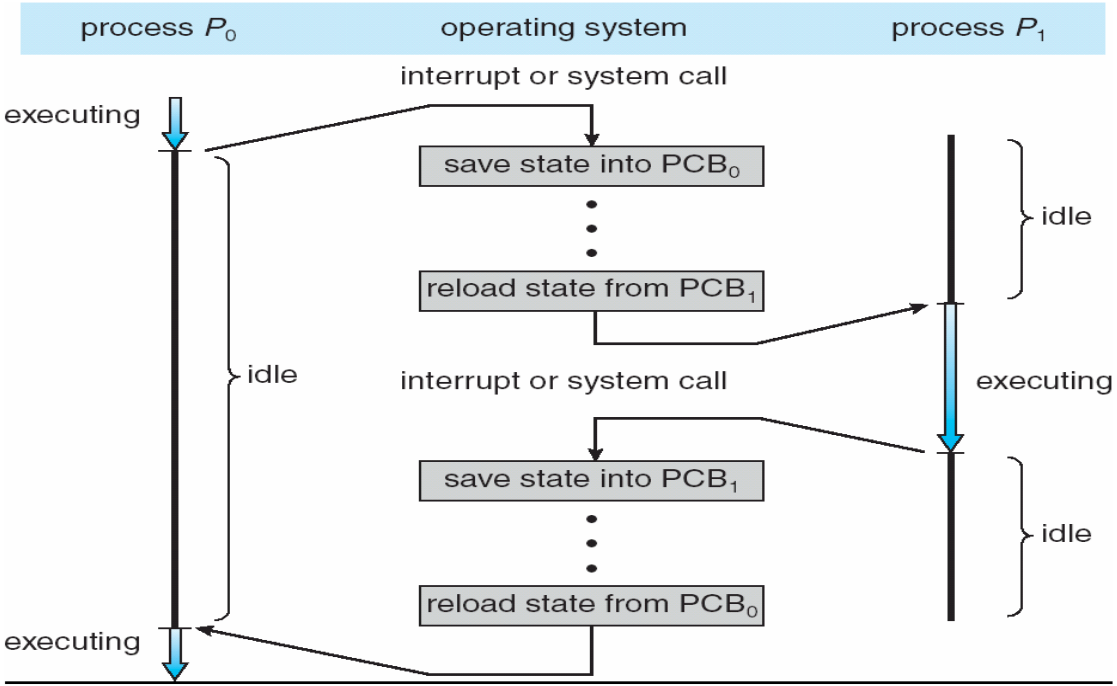
PCB Fields:

- **Process state** – new, ready, running, waiting and so on
- **Program counter** – address of the next instruction to be executed
- **CPU registers** – contents of all process-centric registers
- **CPU scheduling** information- priorities, scheduling queue pointers
- **Memory-management information** – memory allocated to the process
- **Accounting information** – CPU used, clock time elapsed since start, time limits
- **I/O status information** – I/O devices allocated to process, list of open files



Source: Operating_System_Concepts,_8th_Edition

CPU Switch From Process to Process



Source: Operating_System_Concepts,_8th_ Edition

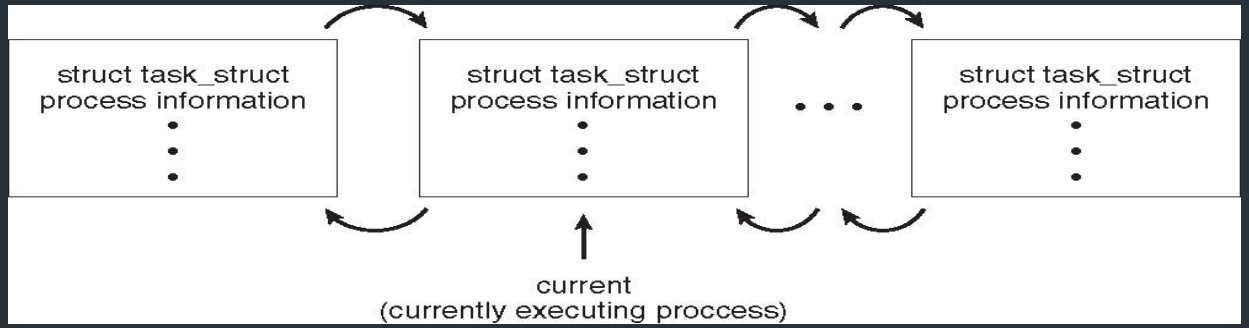
Threads

- A basic unit of CPU utilization, consisting of a **program counter, a stack, and a set of registers**, (and a **thread** ID.) Traditional (heavyweight) processes have a single **thread** of control –
  - There is one program counter, and one sequence of instructions that can be carried out at any given time.
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> **threads**
- Must then have storage for thread details, multiple program counters in PCB

Process Representation in Linux

Represented by the C structure `task_struct`

```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```



Process Creation

- **A tree of process where Parent** process create **children** processes, which, in turn create other processes
- **process identifier** (**pid**) : identifier used for process identification and management
- **Resource sharing options**
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- **Execution options**
  - Parent and children execute concurrently
  - Parent waits until children terminate

LINUX SYSTEM CALLS OVERVIEW

System Call – Interface to OS Services (AP

✓ nothing but your C / C++ routines ; User Mode: Kernel Mode

System Call Types

Process Control ; File Manipulation ; Info Maintenance; Communication ; Protection ; Device Management

PROCESS CONTROL Calls: end() ; abort(); load(); execute();

Create(); terminate(); get / set attributes () ; wait(), fork(), etc.

FILE MANIPULATION Calls: create(), delete(), open(), read(), write(), get / set file attributes ();

INFORMATION MAINTENANCE calls: getpid(), getppid(), etc.

COMMUNICATION Calls: pipe() ; shmopen() ; mmap(), etc.

PROTECTION Calls: chmod() ; chown() ; umask();

DEVICE MANAGEMENT Calls: read(), write(), get / set device attributes

LINUX SYSTEM CALLS OVERVIEW

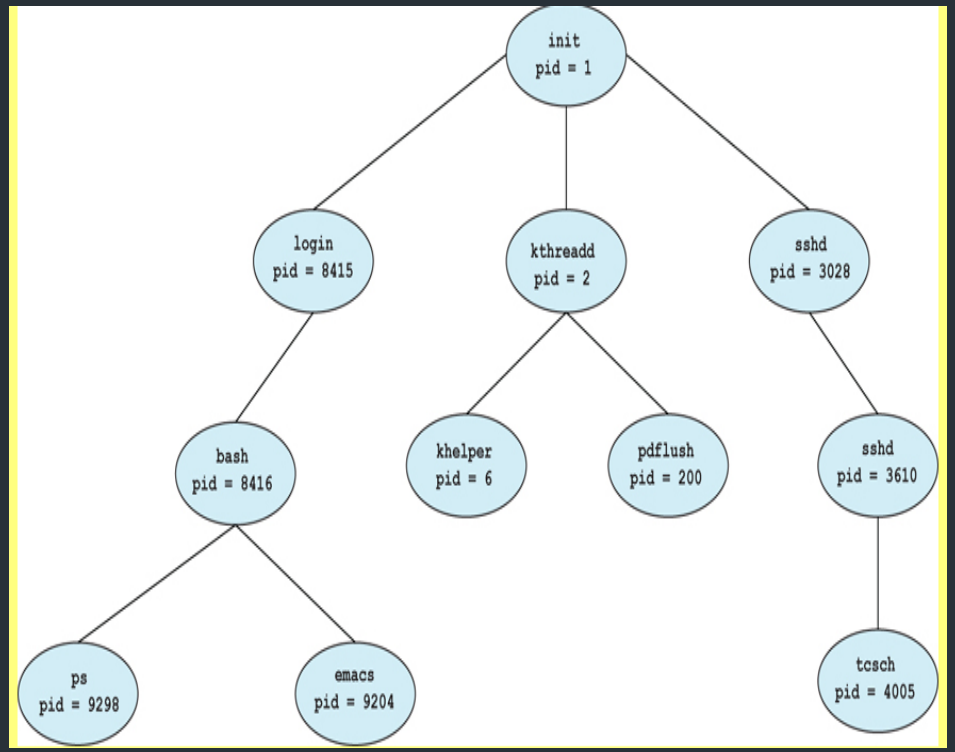
FORK() and related system calls

- ✓ Process creation using fork () system call
- ✓ Process – Program in Execution – Must reside in Main Memory, Occupy CPU
- ✓ Attributes of a Process
- ✓ PID ; STATE ; PC; PRIORITY ; GPR ;
- ✓ List (open files); Open Devices; Protection Info
- ✓ Process Details are stored in Process Control Block (PCB)
- ✓ **Two Types of Processes** : CPU Bound v/s IO Bound
- ✓ Running State v/s Wait State

[More Cpu Time v/s More IO Time] –

Right Balance of processes – Schedulers Challenge

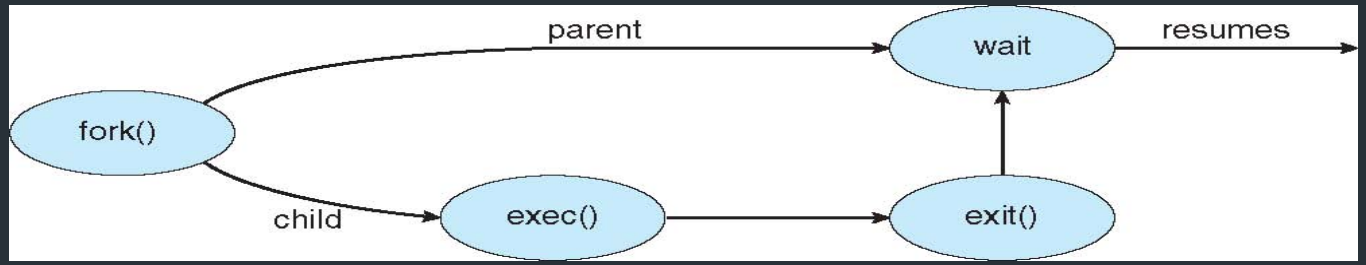
Sample Tree of Processes in Linux



- On typical LINUX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes.

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - `fork()` system call creates new process
  - `exec()` system call used after a `fork()` to replace the process' memory space with a new program



PROCESS MGMT CONTINUED

❑ **3 types of schedulers in Linux – LTS ; STS ; MTS**

✓ LTS – New Process to Ready State (also called as Job Scheduler)

✓ STS – Ready to Run (also called as CPU Scheduler)

✓ MTS – Suspend to Resume (vice versa) – Swapper

✓ Feeling of Seamless or Infinite Process Creation for End User

Despite internal limit on no of processes

❖ Degree of Multiprogramming – No of Processes in Main Memory at a given time t – controlled by the LTS

❖ LTS Challenge – right balance of CPU bound and IO Bound processes – achieve good throughput

❖ Swapper – serves the purpose of Context Switching

- ✓cc first.c
- ✓./a.out and enter – this is parent process in execution
- ✓Leads a call to main – at fork point – a copy of a.out is created and this is the child process
- ✓By default child carries the same image as that of the parent post the forking point
- ✓Order of execution of parent / child operations / statements is in kernel's schedulers hands!
- ✓Next printf example – output statements ordering can change.
- ✓Assume parent gets the control first (which again can vary)
- ✓Subsequent sessions – other calls such as exec, variants of it, wait, etc.
- ✓Overlapped with Scheduling Algorithms – Premptive v/s Non Premptive

FORK System Call

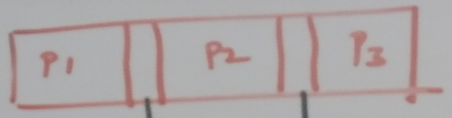
- **Why is it called Fork() – dictionary relevance!**
- ✓ Helps in creating New Processes in Linux
- ✓ New Process is created as child of the parent process
- ✓ Parent is the calling process (normally ./a.out or main call)
- ✓ On successful fork – Two processes reside in Main Memory (Parent and Child)
- **Child Process definition – statements post the forking point**
- **Three Possible Return values of Fork() Sytem call**

  0 indicates child process control

  >0 indicates parent process control

  <0 failed fork call

Content Switching



CST → Content Switch Time

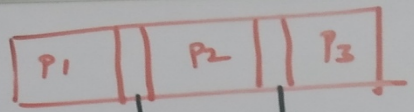
FIRST FORK EXAMPLE

```
int main() {
    int pid;
    Pid = fork();
if (pid < 0)
    Printf("Fork Failed \n");
else if (pid == 0)
    Printf("child Block \n");
else if (pid > 0)
    Printf("Parent Block \n");
return 0;
}.
```

O/P

Parent Block
child Block

Content Switching



CST → Content Switch Time

FIRST FORK EXAMPLE

```
int main() {
    int pid;
    Pid = fork();
    if (pid < 0)
        Printf("Fork Failed \n");
    else if (pid == 0)
        Printf("Child Block \n");
    else if (pid > 0)
        Printf("Parent Block \n");
    Printf("Welcome To as \n");
    return 0;
}
```

How many times will "Welcome to as" Print?

/* ⟶ change */

C Program Forking Separate Process

```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");
       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;
}
```

Creating a Separate Process via Windows API

```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
       fprintf(stderr, "Create Process Failed");
       return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Source: Operating_System_Concepts,_8th_ Edition

Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - 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

- 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.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process

```
pid = wait(&status);
```

- If no parent waiting (did not invoke `wait()`) process is a **zombie**
- If parent terminated without invoking `wait`, process is an **orphan**

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

References

- Abraham Silberschatz, Peter B. Galvin, Greg Gagne-Operating System Concepts, Wiley (2018).

THANK YOU