Operating Systems

Instructor: Dr. Shachi Sharma

(Semester: Winter 2019)

Process Management

Process

- A program in execution
  - A process has its own address space consisting of:
    - Text region
      - Stores the code that the processor executes
    - Data region
      - Stores variables and dynamically allocated memory
    - Stack region
      - Stores instructions and local variables for active procedure calls

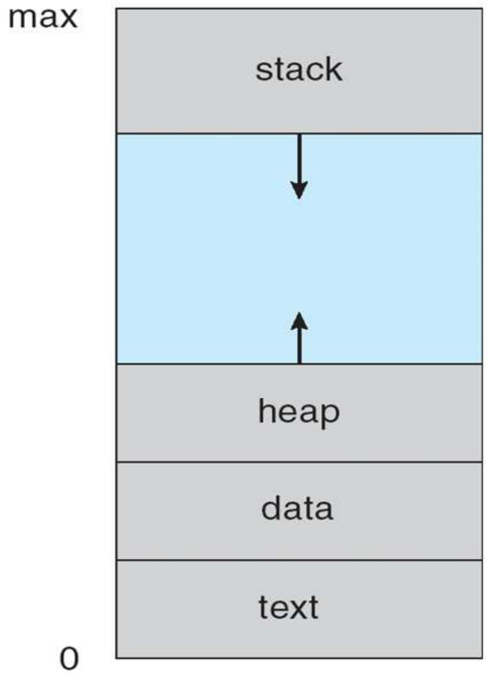
Process

- For every program, the OS allocates an ``isolation'' – unique set of code, stack, heap, data and BSS for each program.
- Each program runs as if there is only itself and none other.
- The programmer need not worry about other programs in the system.
- The OS takes care of starting and stopping programs and allocation blocks of memory to each running program
- **Virtual memory:** Each process essentially uses the same virtual address space – also known as the offset address – which equals the entire address space – X86_32: 2^32 addresses, X86_64: 2^64 addresses.
- The **task scheduler/process scheduler** determines each task to start and stop and everything about the process is saved in individual **Process Control Blocks** (PCBs).

- **Each process has its own code, stack, heap, data and BSS**

Process Memory

- Heap, code, data, rodata, bss – addresses increase from low to high – the registers (and subsequently the instructions) used to address it increment in the increasing order.
- Stack – address increase from higher to lower. Stack registers are decremented everytime the CPU executes a stack operation PUSH/POP/RET/CALL

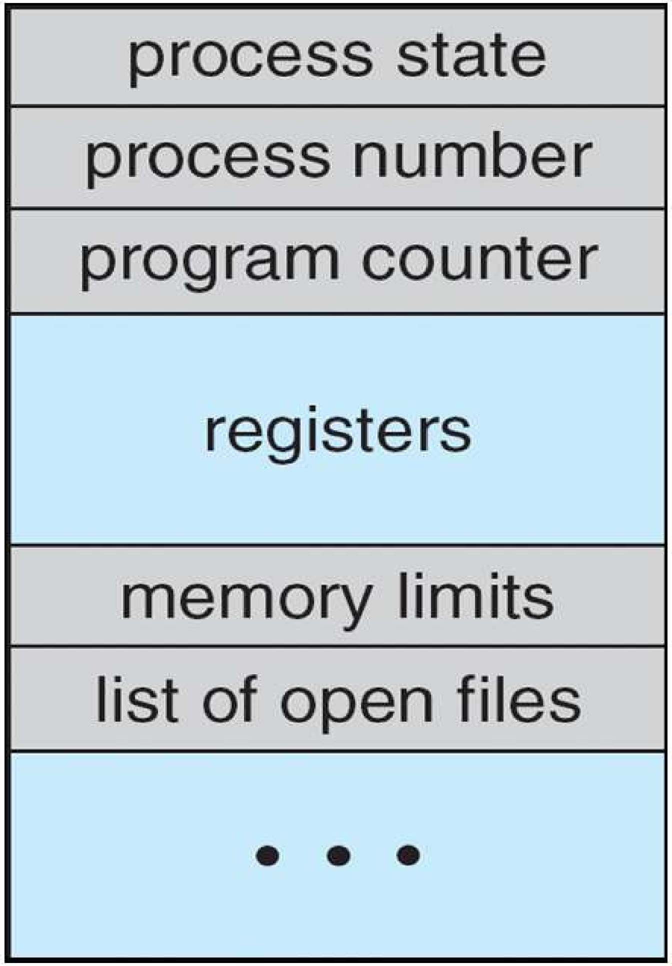


Process Control Block

Information associated with each process

(also called **task control block**)

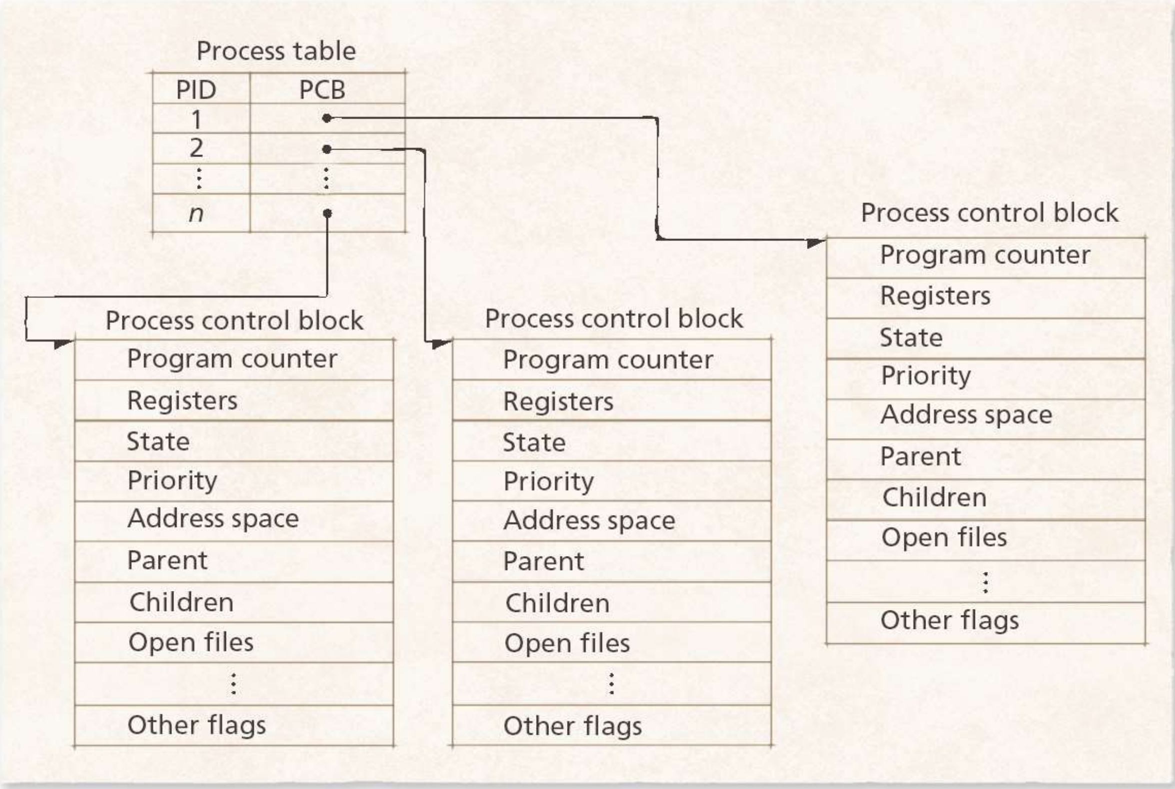
- Process state – running, waiting, etc
- *Priority (Optional)*
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated 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



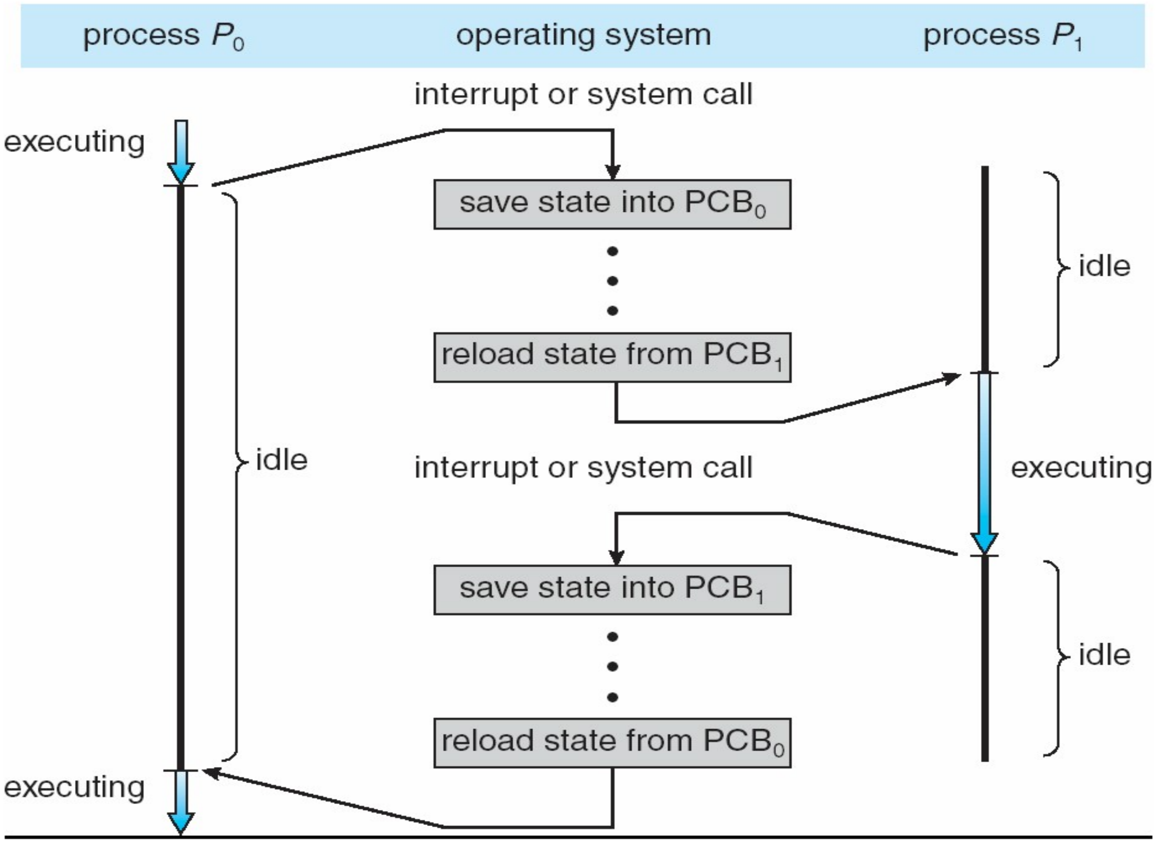
Process Table

- The OS maintains pointers to each process’s PCB in a system-wide or per-user process table
- Allows for quick access to PCBs
- When a process is terminated, the OS removes the process from the process table and frees all of the process’s resources

Process Table and PCBs



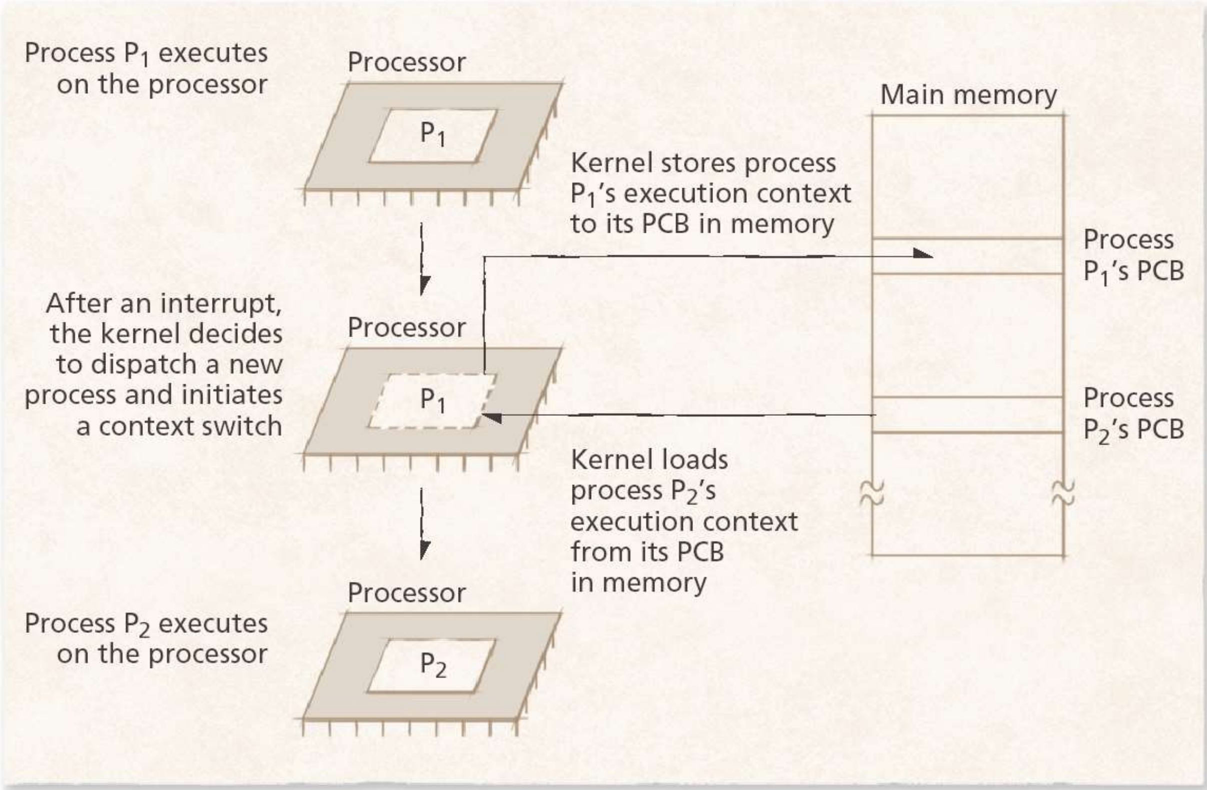
Task/Process Scheduler or Context Switcher



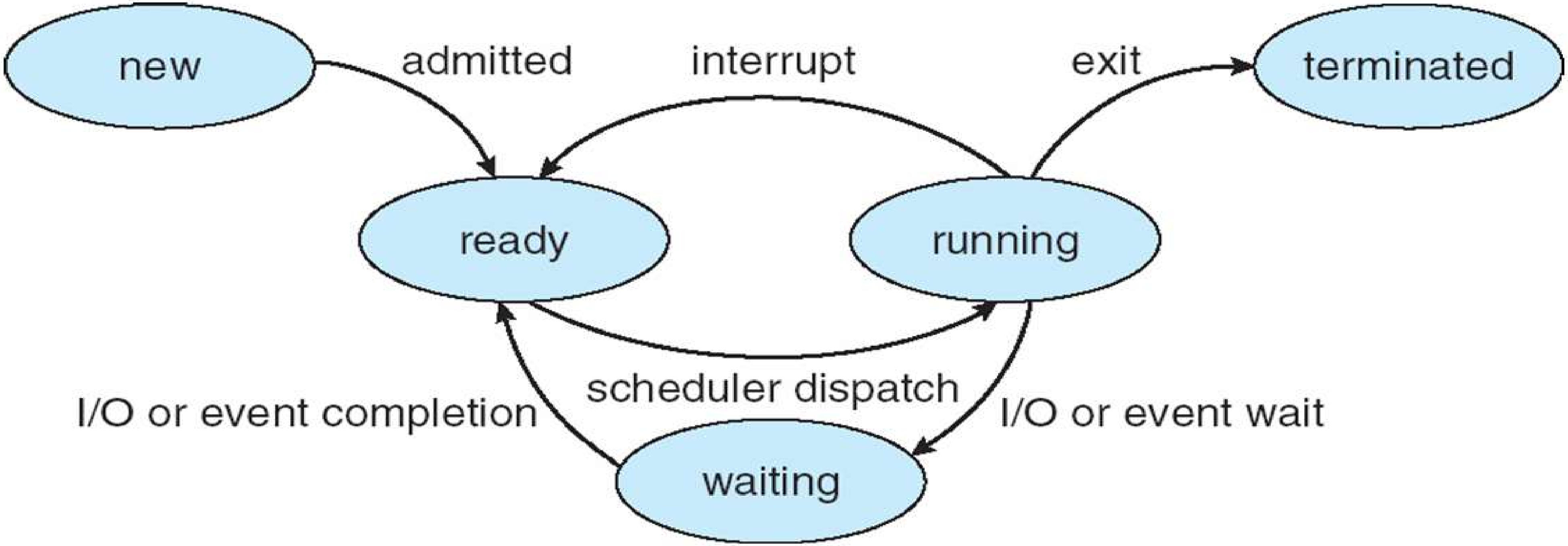
Context Switching

- Context switches
  - Performed by the OS to stop executing a *running* process and begin executing a previously *ready* process
  - Save the execution context of the *running* process to its PCB
  - Load the *ready* process's execution context from its PCB
  - Must be transparent to processes
  - Require the processor to not perform any "useful" computation
    - OS must therefore minimize context-switching time
  - Performed in hardware by some architectures

Context Switching



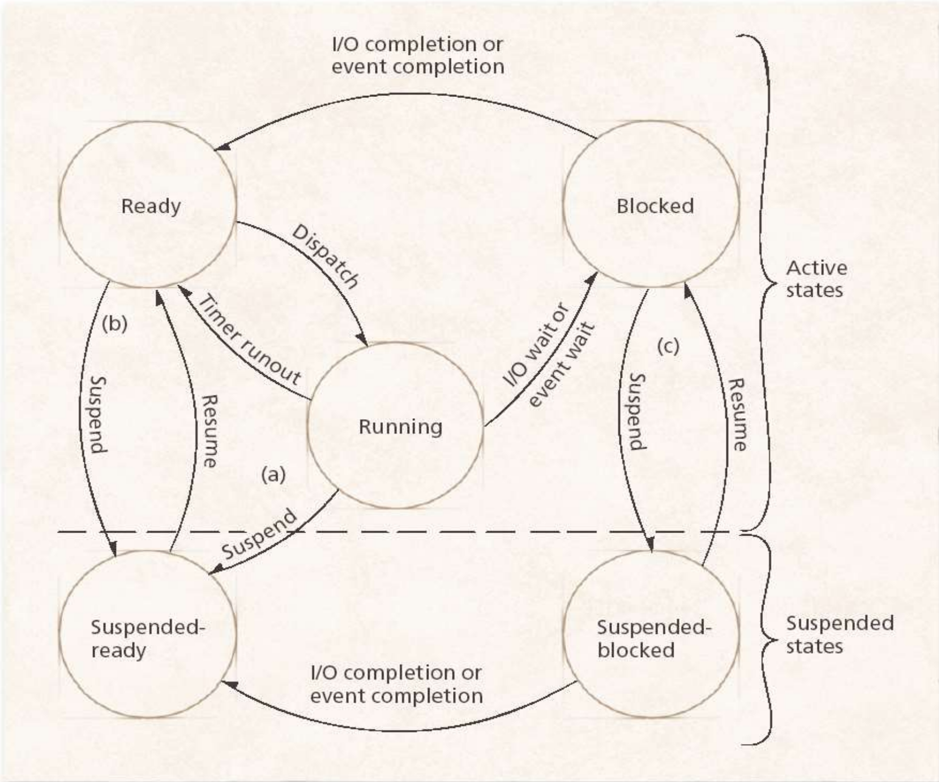
Process State



Suspend and Resume

- Suspending a process
  - Indefinitely removes it from contention for time on a processor without being destroyed
  - Useful for detecting security threats and for software debugging purposes
  - A suspension may be initiated by the process being suspended or by another process
  - A suspended process must be resumed by another process
  - Two suspended states:
    - *Suspendedready (The process is suspended and on disk but ready to run)*
    - *Suspendedblocked (The process is suspended and is waiting for an event)*

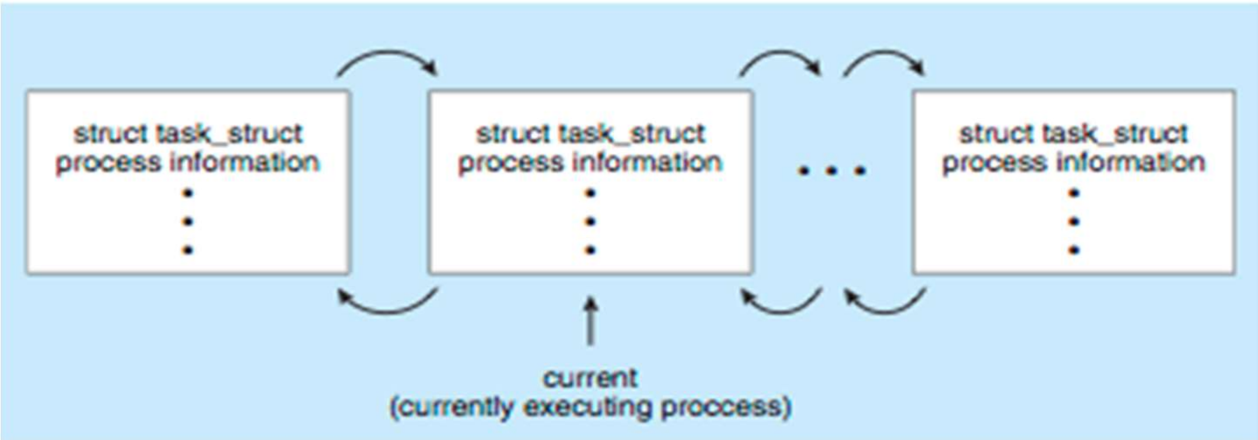
Suspend and Resume



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 */
```

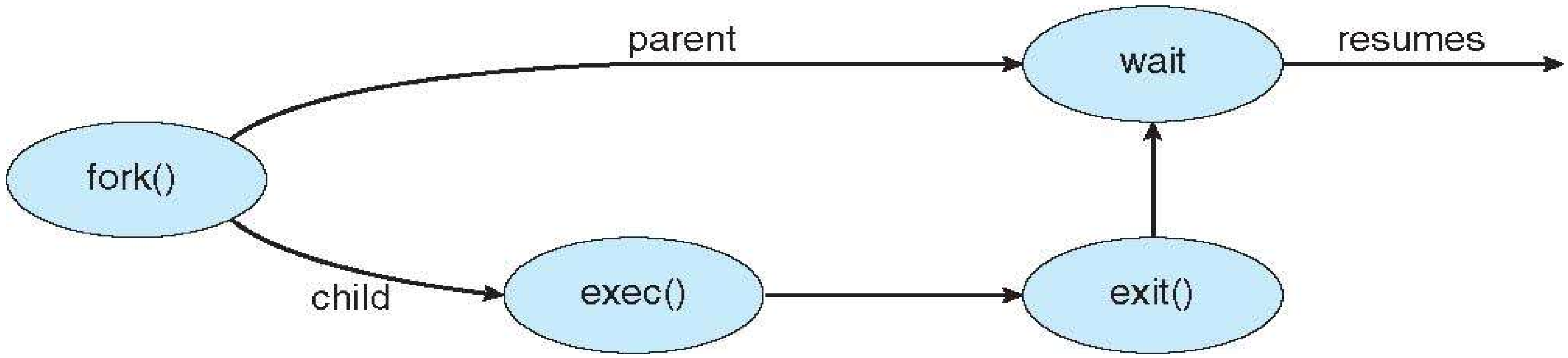


Process Creation

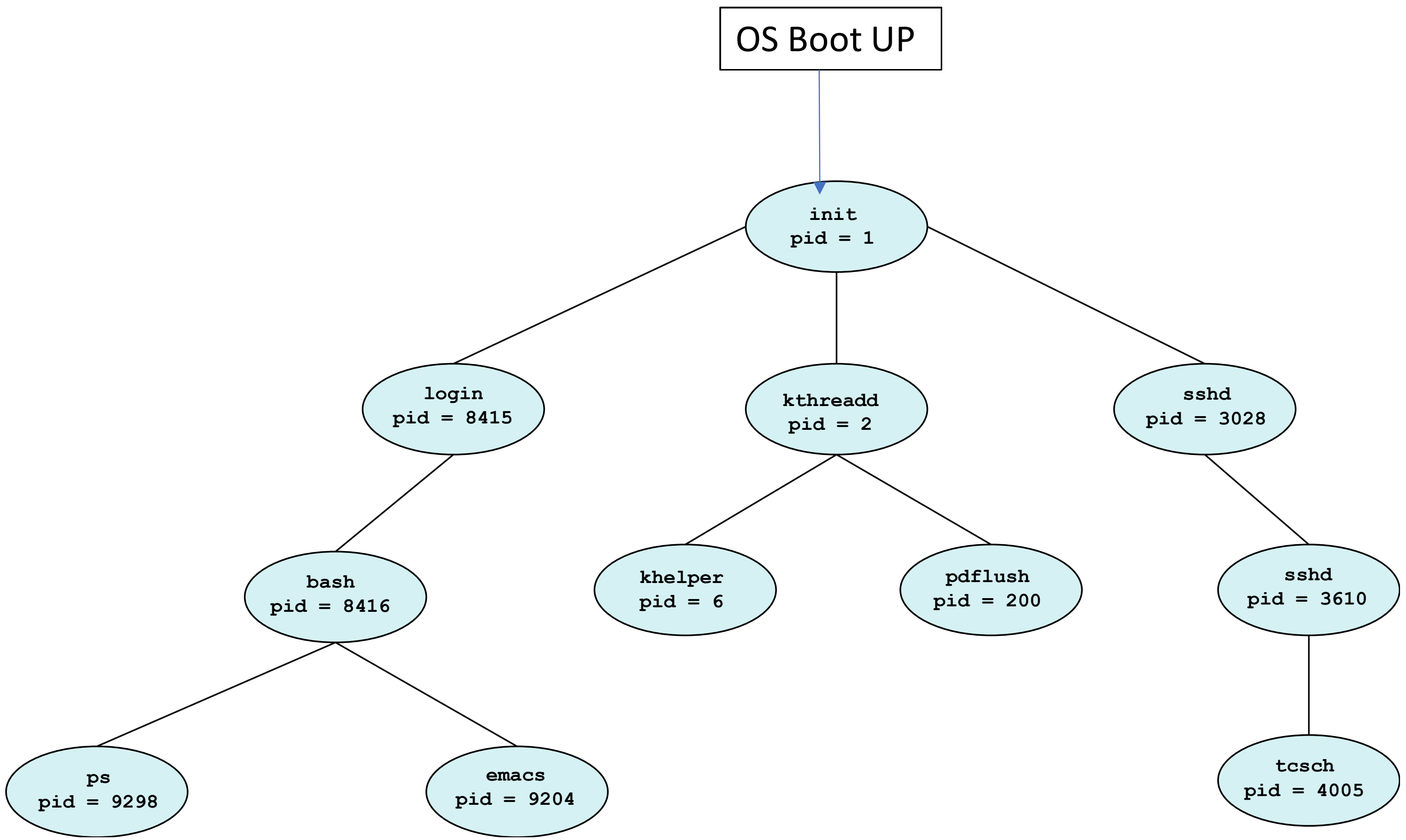
- When process is created by OS on a user request, the process is called **spawning**
- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource 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

Process Creation

- 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



Processes Tree in Linux



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

- Process executes last statement and asks the operating system to delete it (**`exit()`**)
  - Output data from child to parent (via **`wait()`**)
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**`abort()`**)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating systems do not allow child to continue if its parent terminates
      - All children terminated - **cascading termination**
- Wait for termination, returning the pid:

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

- If no parent waiting, then terminated process is a **zombie**
- If parent terminated, processes are **orphans**

Process Queues