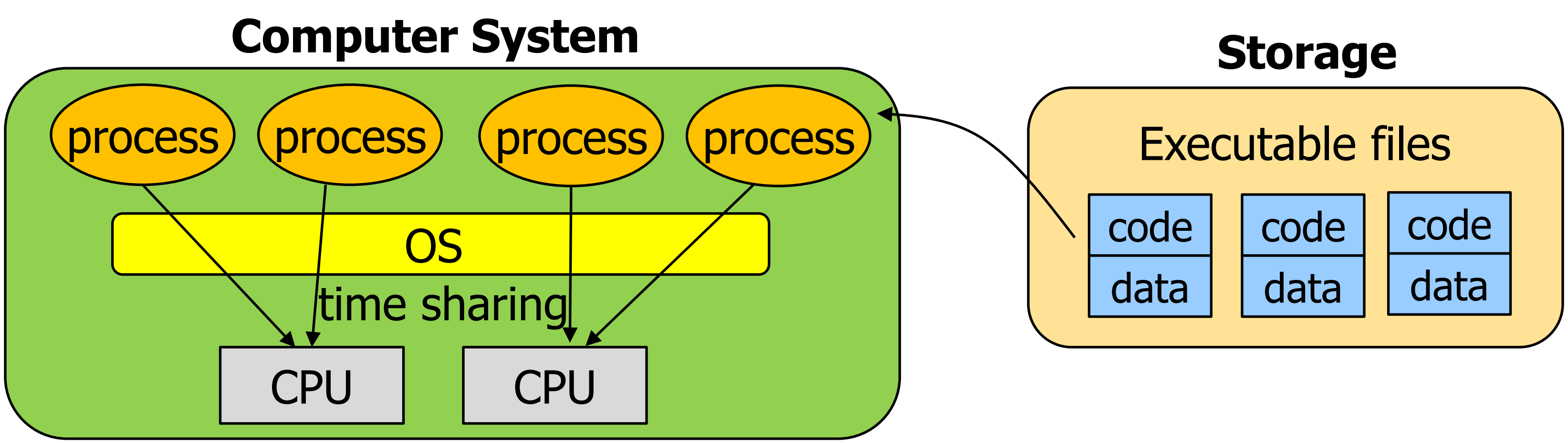
Chap 4, 5: Process

Process Concept

- Process (= running program)
  - An entity that is registered to kernel for execution
  - Kernel manages the processes to improve overall system performance
- A typical system may be seemingly running tens or even hundreds of processes at the same time.
- CPU Virtualization
  - There are only a few physical CPUs available,
  - The OS can promote the illusion that many virtual CPUs exist via **time sharing**
  - By running one process, then stopping it and running another.



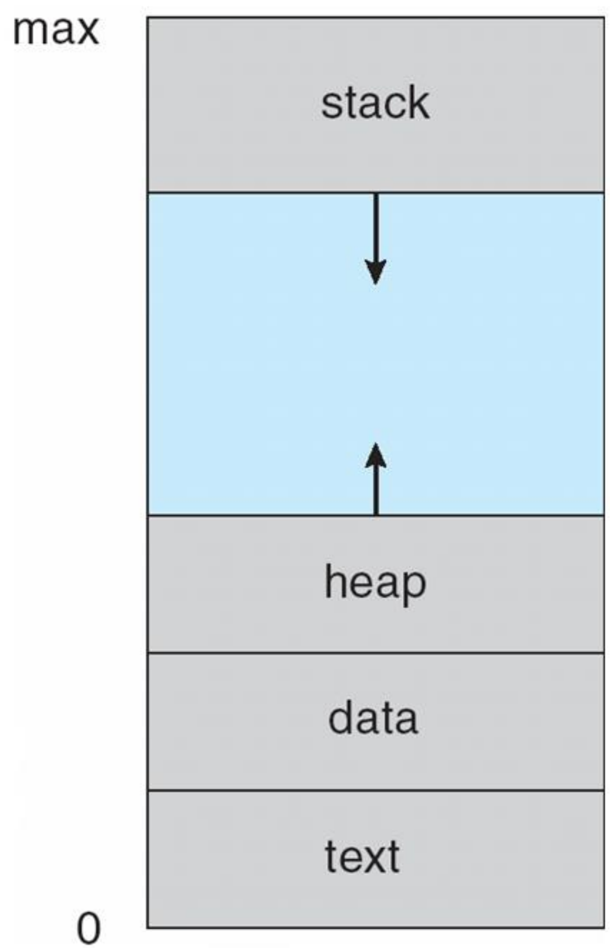
Process Concept

- A process is
  - A program in execution
  - An entity that is registered and being managed by kernel
  - An active entity
    - Request/allocate/release system resources during execution
  - An entity that is allocated the PCB

- **PCB** (Process Control Block)
  - Keeps several information about each process that is registered to the kernel
  - Maintained in kernel space

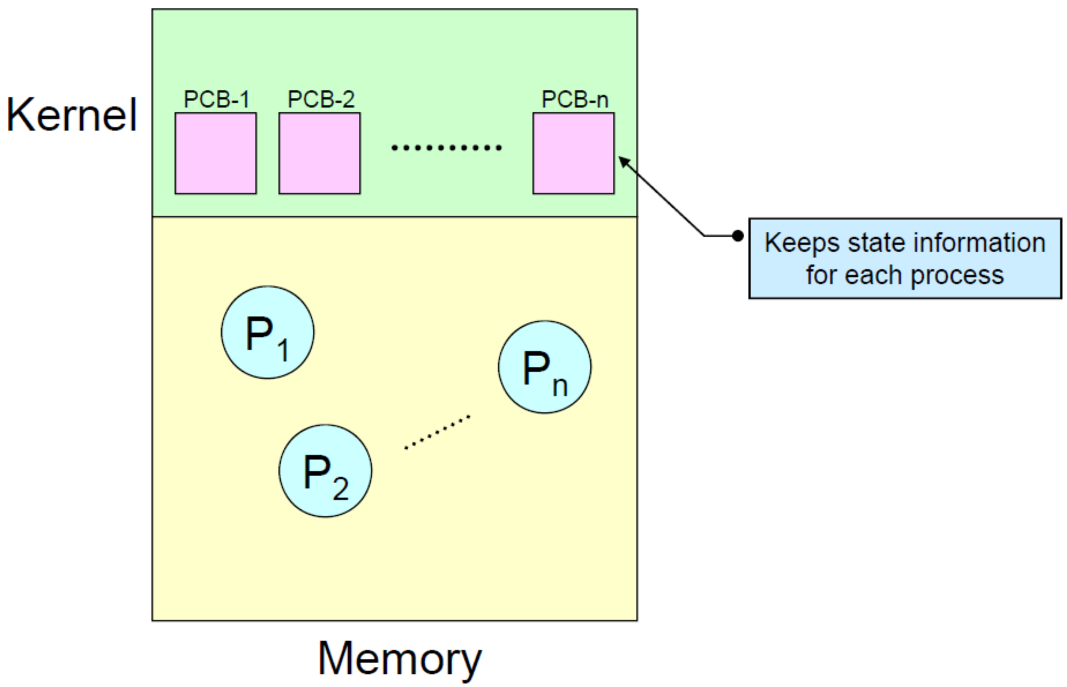
Machine State of Process

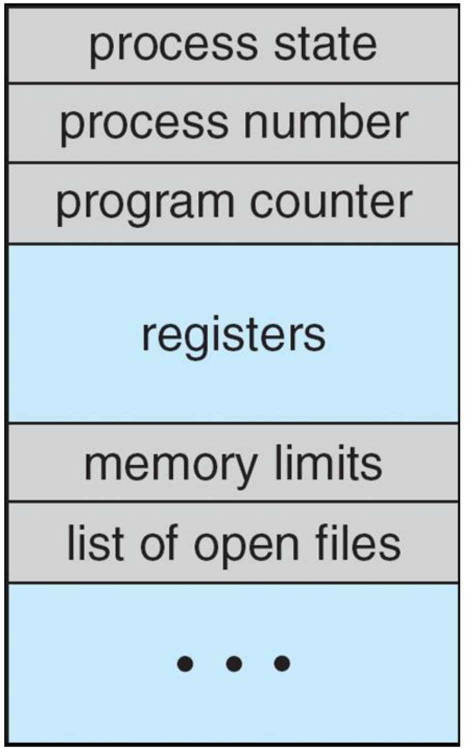
- A process includes
  - Memory
    - Program code → text
    - Global data → data
    - Temporary data → stack
      - Local variables, function parameters, return addresses
    - Heap
      - Memory area that is dynamically allocated during execution
  - Registers
    - Values of the processor registers
    - Include program counter, stack pointer, frame pointer
  - I/O information
    - a list of the files the process currently has open



Process memory address space

PCB


Kernel
PCB-1
PCB-2
PCB-n
Keeps state information for each process
P_1
P_n
P_2
Memory


process state
process number
program counter
registers
memory limits
list of open files
• • •

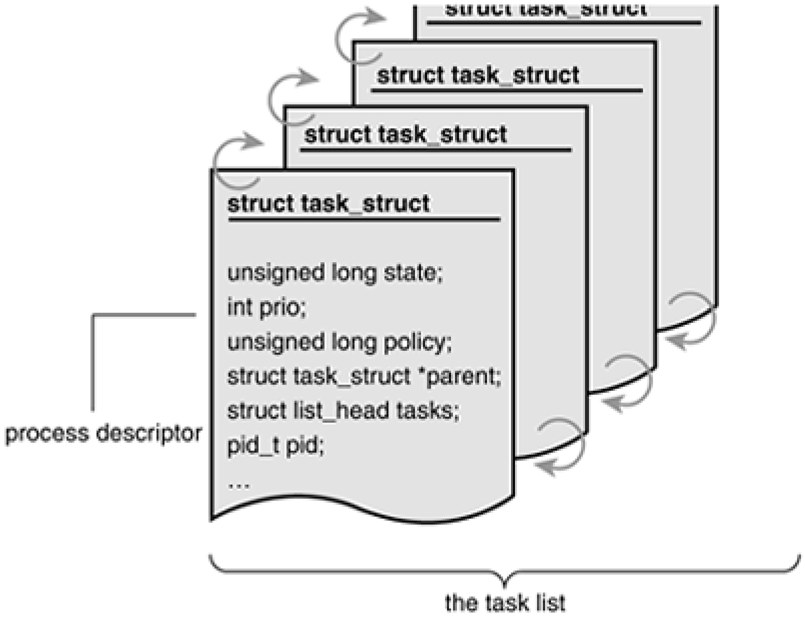
PCB

PCB

- **Information in PCB (task control block)**
  - PID (Process Identification Number)
  - Process state: running, waiting, etc
  - Program counter: location of instruction to next execute
  - CPU registers: contents of all process-centric registers
  - Scheduling information
    - Process priority, scheduling parameters, etc.
  - Memory management information
    - Base/limit registers, page tables, segment tables, etc.
  - I/O status information
    - List of I/O devices allocated, list of open files, etc.
  - Accounting information
    - CPU used, clock time elapsed since start, time limits
  - Context save area
    - Space for saving register context of the process

PCB

- **Notes**
  - PCB information is different for each OS
  - PCB access speed is important for overall system performance
- **In Unix**
  - Process table slot
  - U-area
- **In Linux**
  - Process descriptor (task_struct)



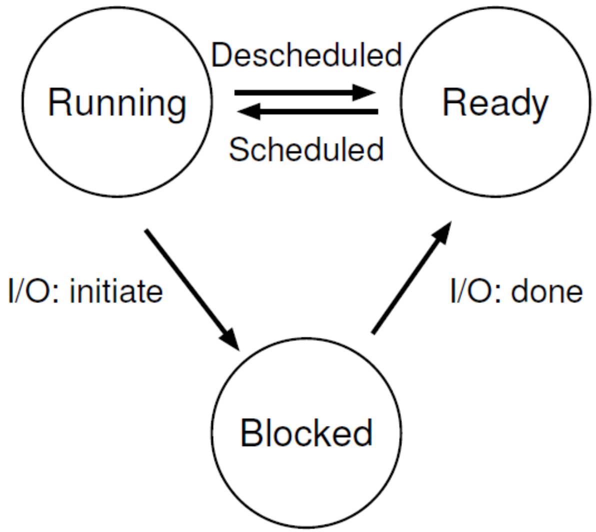
Process API

- **Create**
  - Some method to create new processes.
  - OS is invoked to create a new process to run the program you have indicated.
- **Destroy**
  - an interface to destroy processes forcefully.
- **Wait**
  - Wait for a process to stop running
- **Miscellaneous Control**
  - suspend a process (stop it from running for a while) and then resume it (continue it running).
- **Status**
  - get some status information about a process
  - such as how long it has run for, or what state it is in.

Process States

- **Running**
  - a process is running on a processor, executing instructions.
- **Ready**
  - a process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked**
  - a process has performed some kind of operation that makes it not ready to run until some other event takes place.
  - A common example: when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process State Transition

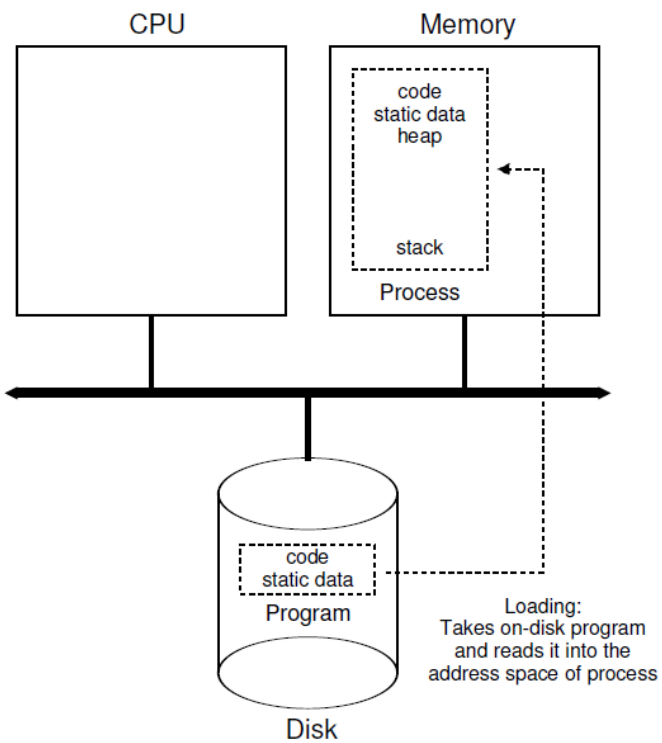


Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	$Process_0$ now done
5	–	Running	
6	–	Running	
7	–	Running	
8	–	Running	$Process_1$ now done

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Process Creation

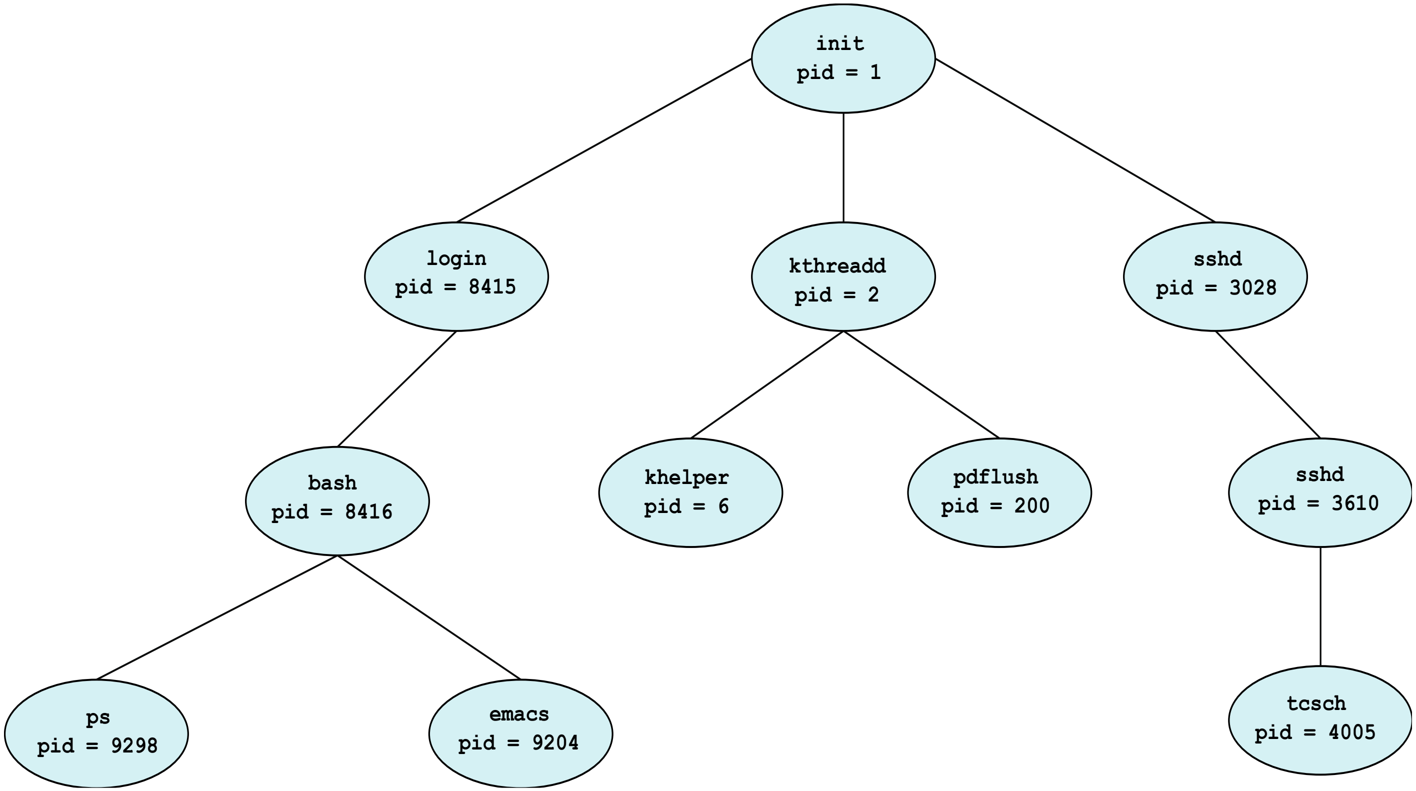
1. Programs initially reside on **disk** in some kind of **executable format**
2. **load** a program's code and any static data into memory, into the address space of the process.
   – Eager or lazy loading (paging and swapping)
3. Allocate run-time stack & initialize it with arguments (argc, argv)
4. Allocate heap
5. Do other initialization tasks related to I/O setup
6. jump to the main() routine, transfers control of the CPU to the newly-created process



Process Creation

- 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 Tree in Linux



‘PS’ command in Linux

1. cs423@localhost:/usr/src/kernels/2.6.40.3-0.fc15.x86_64/include/linux (ssh)

```
[cs423@localhost linux]$ ps -el
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0     1     0  0  80   0 - 13880 epoll_ ?        00:00:29 systemd
1 S     0     2     0  0  80   0 -     0 kthrea ?        00:00:00 kthreadd
1 S     0     3     2  0  80   0 -     0 run_ks ?        00:00:04 ksoftirqd/0
1 S     0     6     2  0 -40   - -     0 cpu_st ?        00:00:00 migration/0
5 S     0     7     2  0 -40   - -     0 watchd ?        00:00:05 watchdog/0
1 S     0     8     2  0  60 -20 -     0 rescue ?        00:00:00 cpuset
1 S     0     9     2  0  60 -20 -     0 rescue ?        00:00:00 khelper
5 S     0    10     2  0  80   0 -     0 devtmp ?        00:00:00 kdevtmpfs
1 S     0    11     2  0  60 -20 -     0 rescue ?        00:00:00 netns
1 S     0    12     2  0  80   0 -     0 bdi_sy ?        00:00:02 sync_supers
1 S     0    13     2  0  80   0 -     0 bdi_fo ?        00:00:00 bdi-default
1 S     0    14     2  0  60 -20 -     0 rescue ?        00:00:00 kintegrityd
1 S     0    15     2  0  60 -20 -     0 rescue ?        00:00:00 kblockd
1 S     0    16     2  0  60 -20 -     0 rescue ?        00:00:00 ata_sff
1 S     0    17     2  0  80   0 -     0 hub_th ?        00:00:00 khubd
1 S     0    18     2  0  60 -20 -     0 rescue ?        00:00:00 md
1 S     0    20     2  0  80   0 -     0 watchd ?        00:00:01 khungtaskd
1 S     0    21     2  0  80   0 -     0 kswapd ?        00:00:42 kswapd0
1 S     0    22     2  0  85   5 -     0 ksm_sc ?        00:00:00 ksmd
1 S     0    23     2  0  99  19 -     0 khugep ?        00:00:12 khugepaged
1 S     0    24     2  0  80   0 -     0 fsnoti ?        00:00:00 fsnotify_mark
1 S     0    25     2  0  60 -20 -     0 rescue ?        00:00:00 crypto
1 S     0    31     2  0  60 -20 -     0 rescue ?        00:00:00 kthrotld
1 S     0    34     2  0  80   0 -     0 scsi_e ?        00:01:50 scsi_eh_0
1 S     0    35     2  0  80   0 -     0 scsi_e ?        00:00:00 scsi_eh_1
1 S     0    38     2  0  60 -20 -     0 rescue ?        00:00:00 kpsmoused
1 S     0   221     2  0  60 -20 -     0 rescue ?        00:00:00 mpt_poll_0
1 S     0   222     2  0  60 -20 -     0 rescue ?        00:00:00 mpt/0
1 S     0   227     2  0  80   0 -     0 scsi_e ?        00:00:00 scsi_eh_2
1 S     0   281     2  0  80   0 -     0 kjourn ?        00:00:13 jbd2/sda1-8
1 S     0   282     2  0  60 -20 -     0 rescue ?        00:00:00 ext4-dio-unwrit
1 S     0   303     2  0  80   0 -     0 bdi_wr ?        00:00:10 flush-8:0
1 S     0   310     2  0  80   0 -     0 kaudit ?        00:00:00 kauditd
4 S     0   319     1  0  80   0 -  5249 epoll_ ?        00:00:00 systemd-logger
4 S     0   328     1  0  76  -4 -  4998 poll_s ?        00:00:00 udevd
4 S    38   538     1  0  80   0 -  7633 poll_s ?        00:00:07 ntpd
0 S     0   539     1  0  80   0 - 31724 poll_s ?        00:00:00 abrtd
4 S    70   541     1  0  80   0 -  6967 poll_s ?        00:00:00 avahi-daemon
4 S     0   545     1  0  80   0 -  4198 hrtime ?        00:00:00 atd
4 S     0   547     1  0  80   0 - 29552 hrtime ?        00:00:14 crond
4 S     0   548     1  0  80   0 -  4715 hrtime ?        00:00:00 smartd
```

systemd = init

child of kthreadd (kernel thread)

child of systemd (user thread)

Process Creation API – fork()

- UNIX examples
- fork() system call creates new process
  - Child duplicate of parent address space, exact copy of the calling process

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (main)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

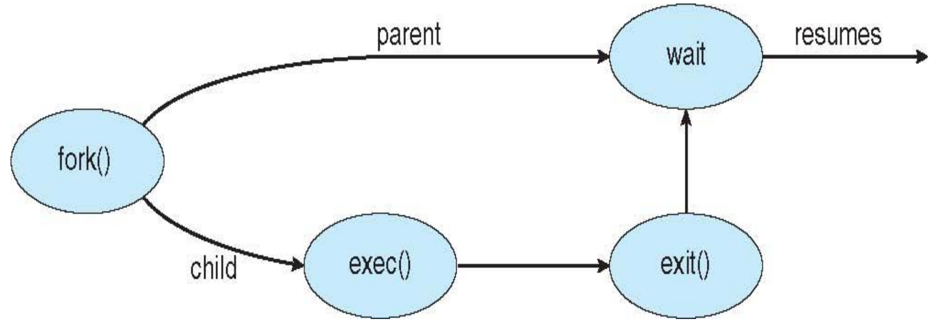
Return value of fork()
- Parent: child PID
- Child: 0

```
prompt> ./p1
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
prompt>
```

or

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

Process Creation API – wait()




```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {          // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else {               // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

The return value of wait() is the child PID.

```
prompt> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
prompt>
```

The child will always print first.

Process Creation API – exec()

- exec() system call used after a fork() to replace the process' memory space with a new program
- On Linux, there are six variants of exec()
  - execl, execlp(), execle(), execv(), execvp(), execvpe()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) {           // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");    // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {                // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
                rc, wc, (int) getpid());
    }
    return 0;
}
```

a successful call to exec() never returns

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

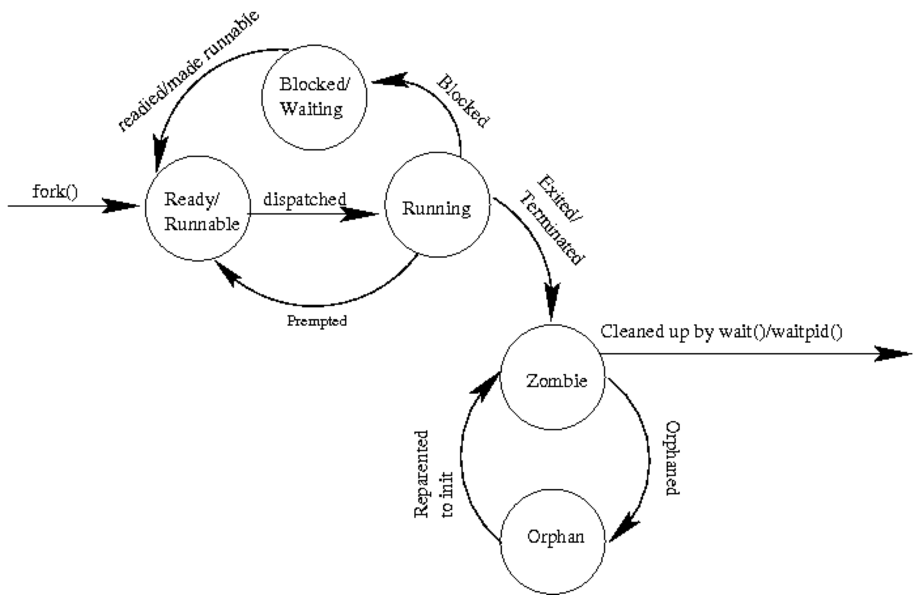
Process Termination

- **Process executes last statement and then asks the operating system to delete it using the `exit()` system call.**
  - Returns status data from child to parent (via **`wait()`**)
  - The resources of the now-extinct process are deallocated by operating system
- **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
  - Some operating systems does not allow a child to continue if its parent terminates.
    - All its children must also be terminated.
    - Cascading termination (All children, grandchildren, etc)
    - The termination is initiated by the operating system.

Zombie & Orphan

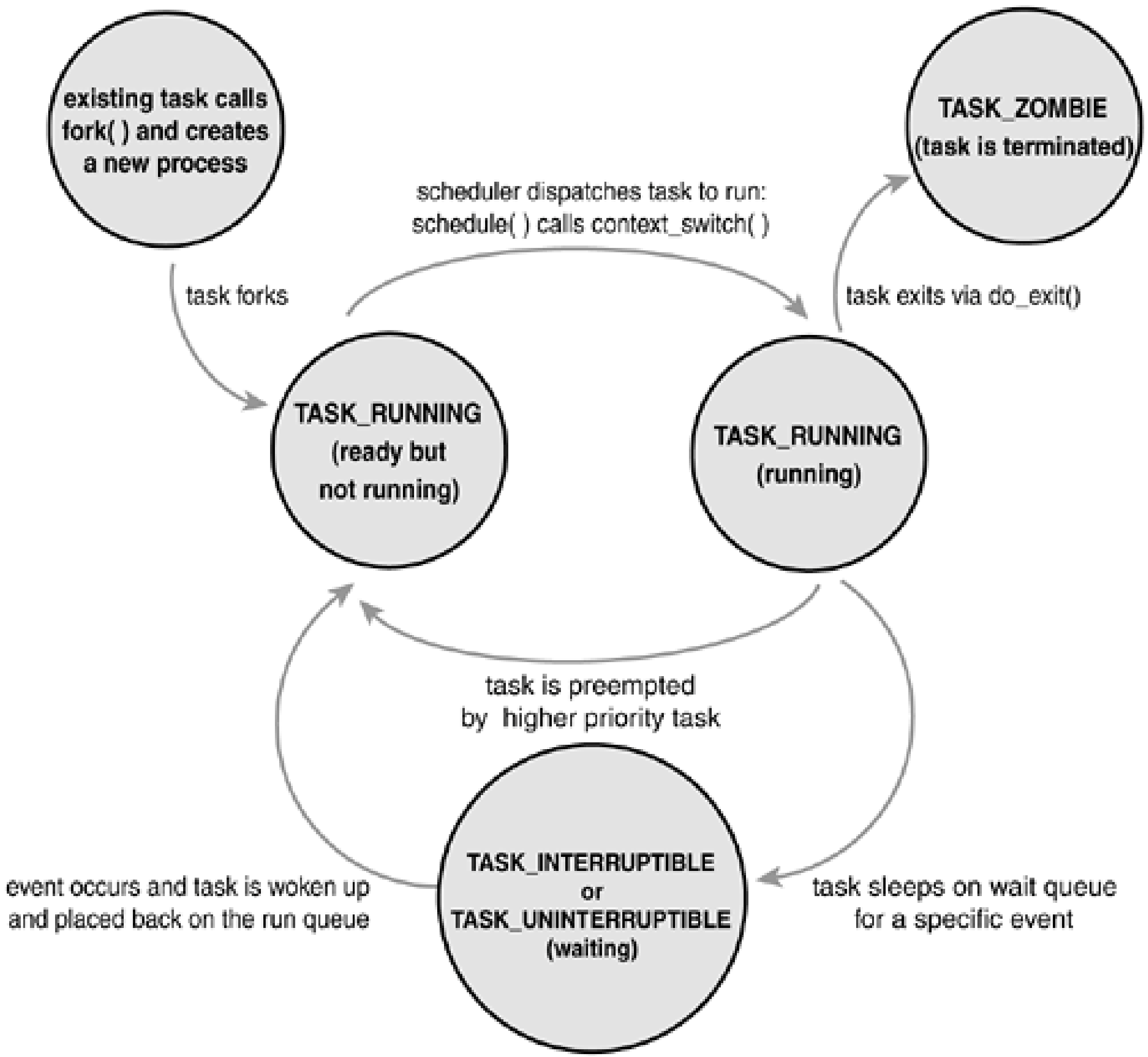
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

  `pid = wait(&status);`
- If no parent waiting (did not invoke `wait()`) process is a zombie
  - It has exited but has not yet been cleaned up
- If parent terminated without invoking `wait`, process is an orphan
  - May be reparented and cleaned up by init.
  - In modern Linux systems, an orphan process may be reparented to a "subreaper" process instead of init. (see prctl(2))



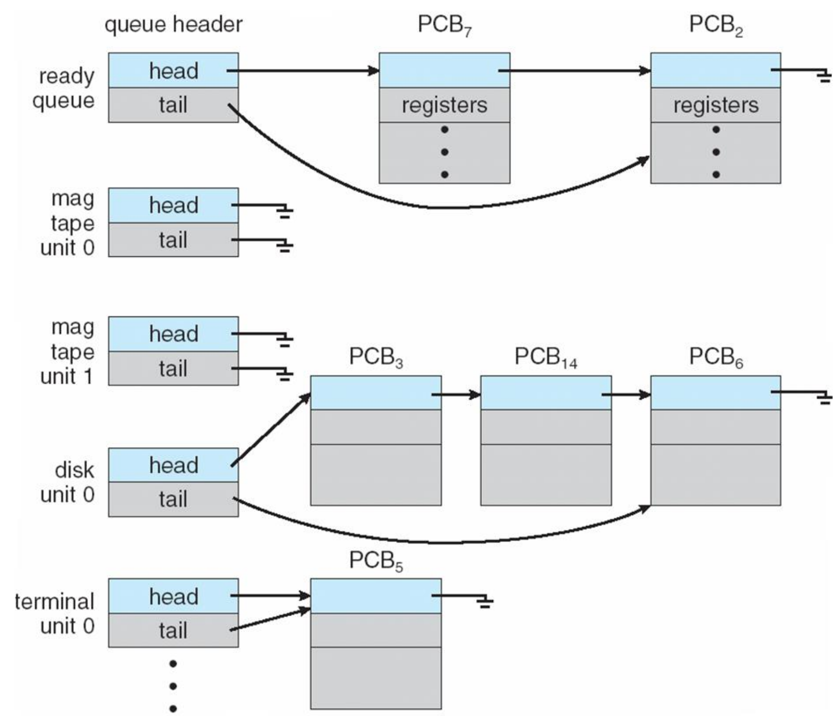
reaper

Process States in Linux



Scheduling Queues

- **Ready queue (ready list)**
  - Processes in ready state
    - Requesting processor, all other resources allocated
  - Process scheduling
    - Selecting a process from the ready list and dispatch it when the processor is available
- **I/O queue (device queue)**
  - Processes in blocked state
    - requesting I/O resources and waiting for their availability
    - Move to ready queue when device responds (via interrupt)
  - Separate list for each resource



Scheduling Queues