CS 370: OPERATING SYSTEMS [PROCESSES]

Computer Science
Colorado State University

Instructor: Louis-Noel Pouchet
Spring 2026

** Lecture slides created by: SHRIDEEP PALLICKARA

Topics covered in this lecture

- Processes
- Interrupts & Context switches
- Operations on processes
  - Creation

Processes

Process

- The oldest and most important abstraction that an operating system provides
- Support the ability to have (*psuedo*) **concurrent** operation
  - Even if there is only 1 CPU

All modern computers do several things at a time

- Browsing while e-mail client is fetching data
- Printing files while burning a CD-ROM

Multiprogramming

- CPU **switches** from process-to-process quickly
- Runs each process for 10s-100s of milliseconds

Multiprogramming and parallelism

- At any instant of time the CPU is running **only one** process
- In the course of 1 second, it is working on **several** of them
- Gives the illusion of parallelism
  - **Psuedoparallelism**

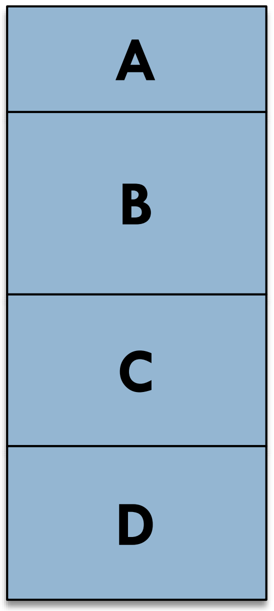
A process is the unit of work in most systems

- Arose out of a need to **compartmentalize** and control *concurrent* program executions
- A process is a program in execution
- Essentially an **activity** of some kind
  - Has a program, input, output and a state.

A process is just an instance of an executing program

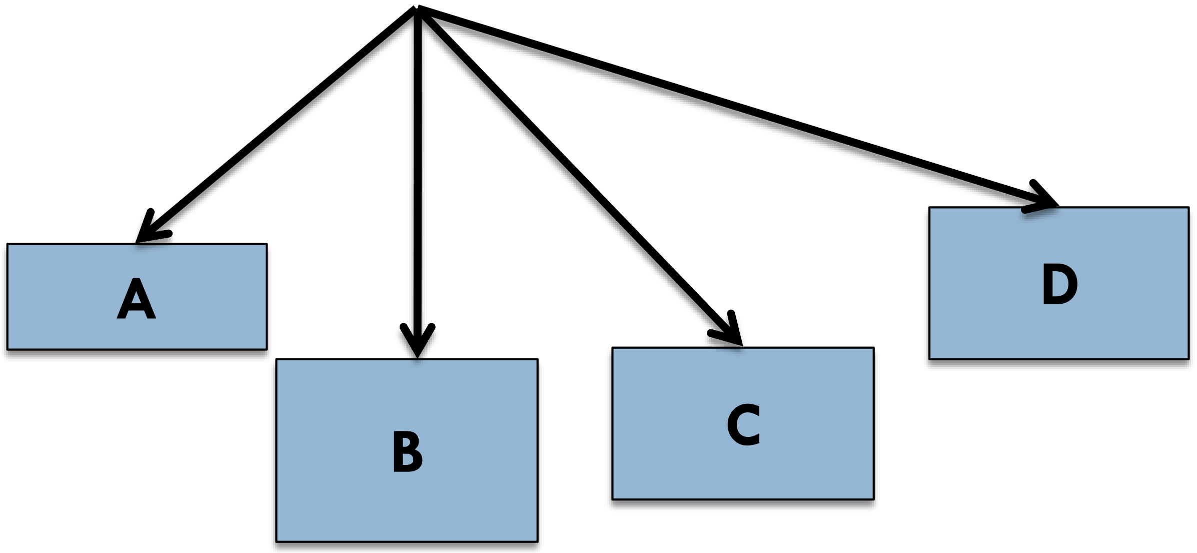
- Conceptually each process has its own **virtual CPU**
- In reality, the CPU switches back-and-forth from process to process
- Processes are <u>not affected</u> by the multiprogramming
  - Or *relative speeds* of different processes

An example scenario: 4 processes

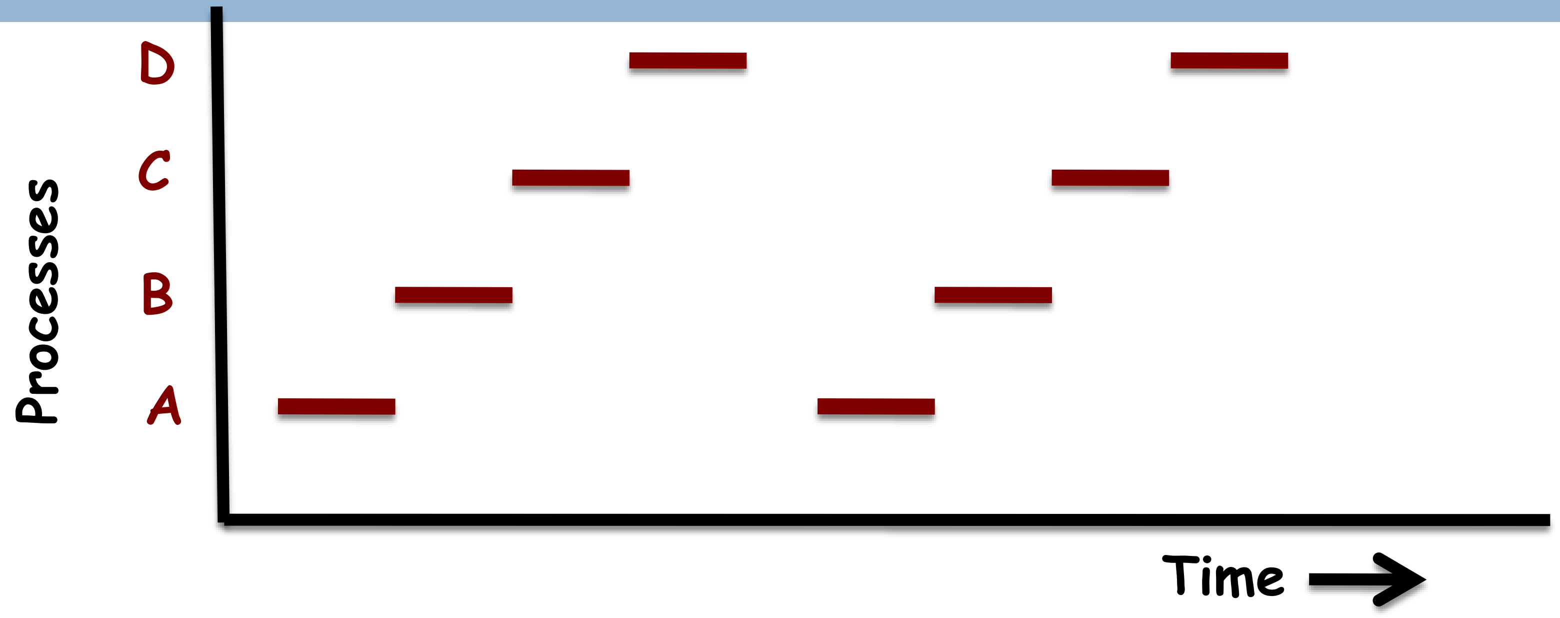


4 processes in memory

Four Program Counters



Example scenario: 4 processes



- At any instant <u>only one</u> process executes
- *Viewed over a long time*, all processes have made **progress**

Programs and Processes

Programs and processes

- Programs are **passive**, processes are **active**
- The difference between a program and a process is subtle, but crucial

Analogy of a culinary-minded computer scientist baking cake for daughter

Analogy	Mapping to real settings
Birthday cake recipe	Program (algorithm expressed in a suitable notation)
Well-stocked kitchen: flour, eggs, sugar, vanilla extract, etc	Input Data
Computer scientist	Processor (CPU)

- **Process is the activity of**
  1. Baker reading the recipe
  2. Fetching the ingredients
  3. Baking the cake

Scientist’s son comes in screaming about a bee sting

- Scientist records *where he was* in the recipe
  - State of current process is saved
- Gets out a first aid book, follows directions in it

In our example, the scientist has switched to a higher priority process …

- FROM Baking
  - Program is cake recipe
- TO administering medical care
  - Program is first-aid book
- When the bee sting is taken care of
  - Scientist **goes back to where he was** in the baking

Key concepts

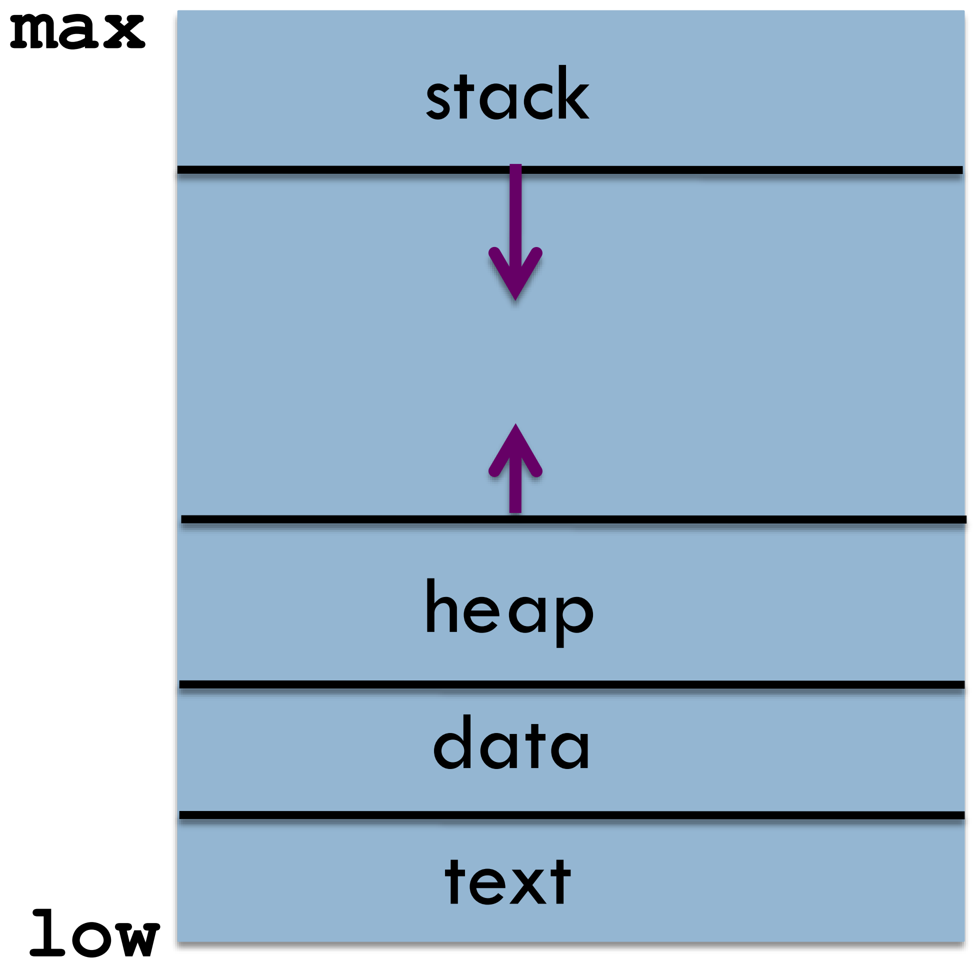
- Process is an **activity** of some kind; it has a
  - Program
  - Input and Output
  - State
- Single processor may be shared among several processes
  - **Scheduling algorithm** decides when to stop work on one, and start work on another

HOW A PROGRAM BECOMES A PROCESS

How a program becomes a process

- When a program is executed, the OS *copies* the program image into main memory
- Allocation of memory is *not enough* to make a program into a process
- Must have a process ID
- OS tracks IDs and process **state**s to orchestrate system resources

A process in memory



{Function parameters,
return addresses,
and local variables}

{Memory allocated dynamically during runtime}

{Global variables}

{Program code}

Program in memory (I)

- Program image *appears* to occupy **contiguous** blocks of memory
- OS **maps** programs into non-contiguous blocks

Program in memory (II)

- Mapping divides the program into equal-sized pieces: **pages**
- OS loads pages into memory
- When processor references memory on page
  - OS looks up page in table, and loads into memory

Advantages of the mapping process

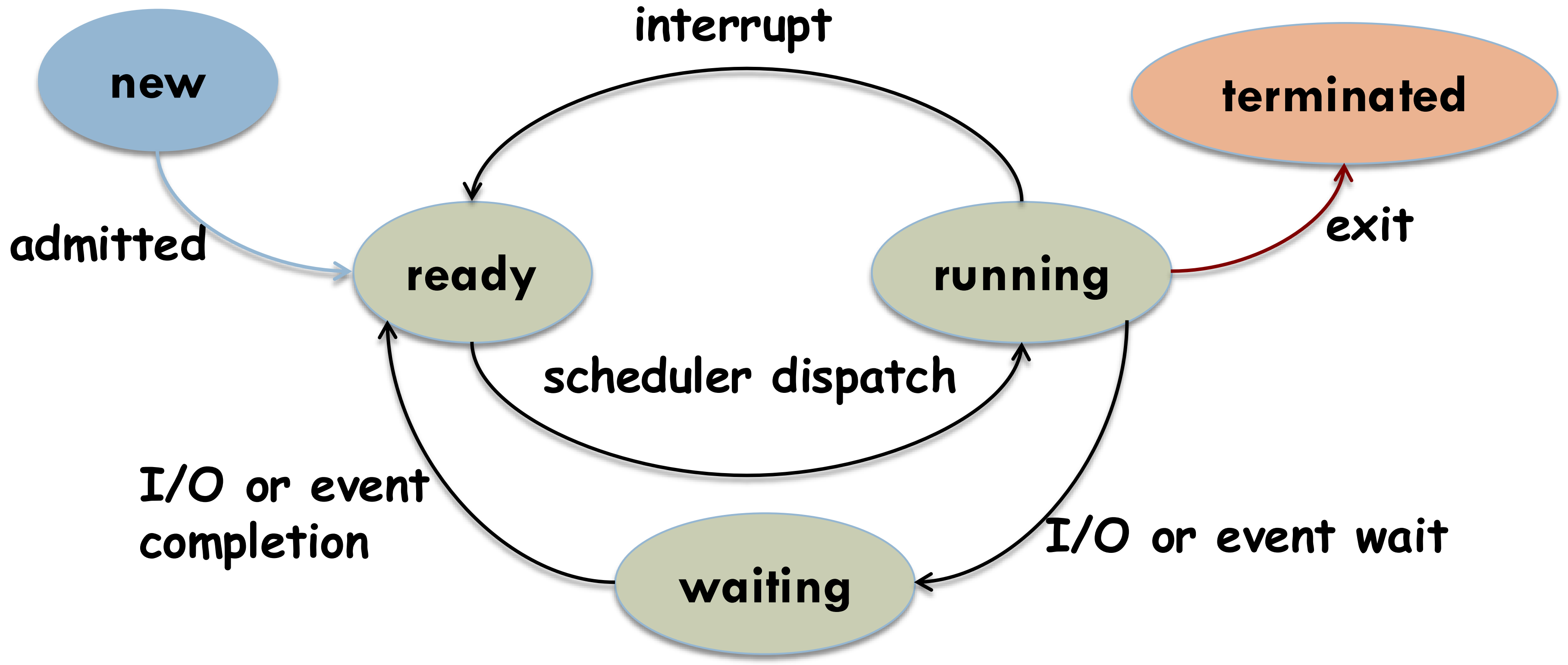
- Allows **large** logical address space for stack and heap
    - **No physical memory used** unless actually needed

- OS hides the mapping process
    - Programmer views program image as **logically contiguous**
    - Some pages may not reside in memory

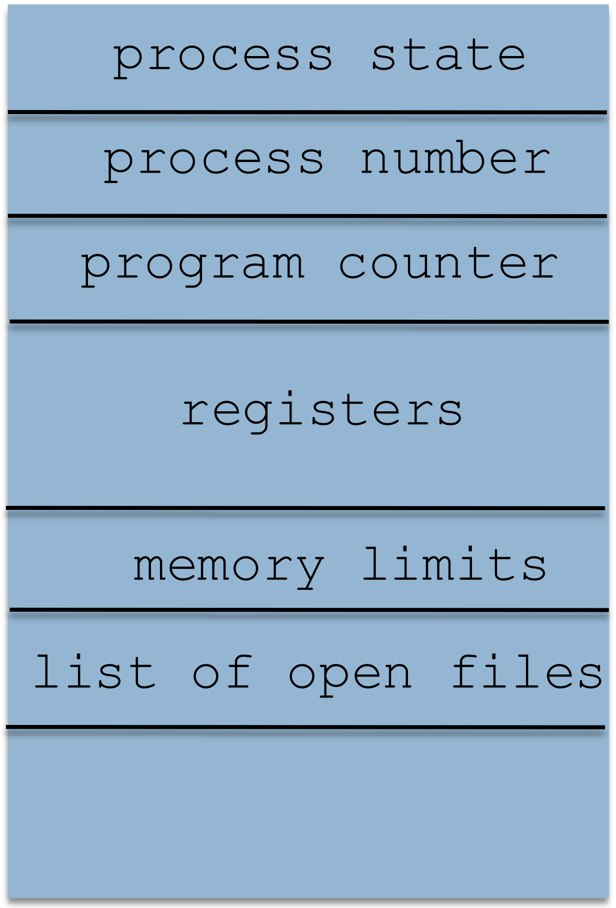
Finite State Machine

- An initial **state**
- A set of possible **input** events
- A finite number of states
- **Transitions** between these states
- Actions

Process state transition diagram: When a process executes it changes state

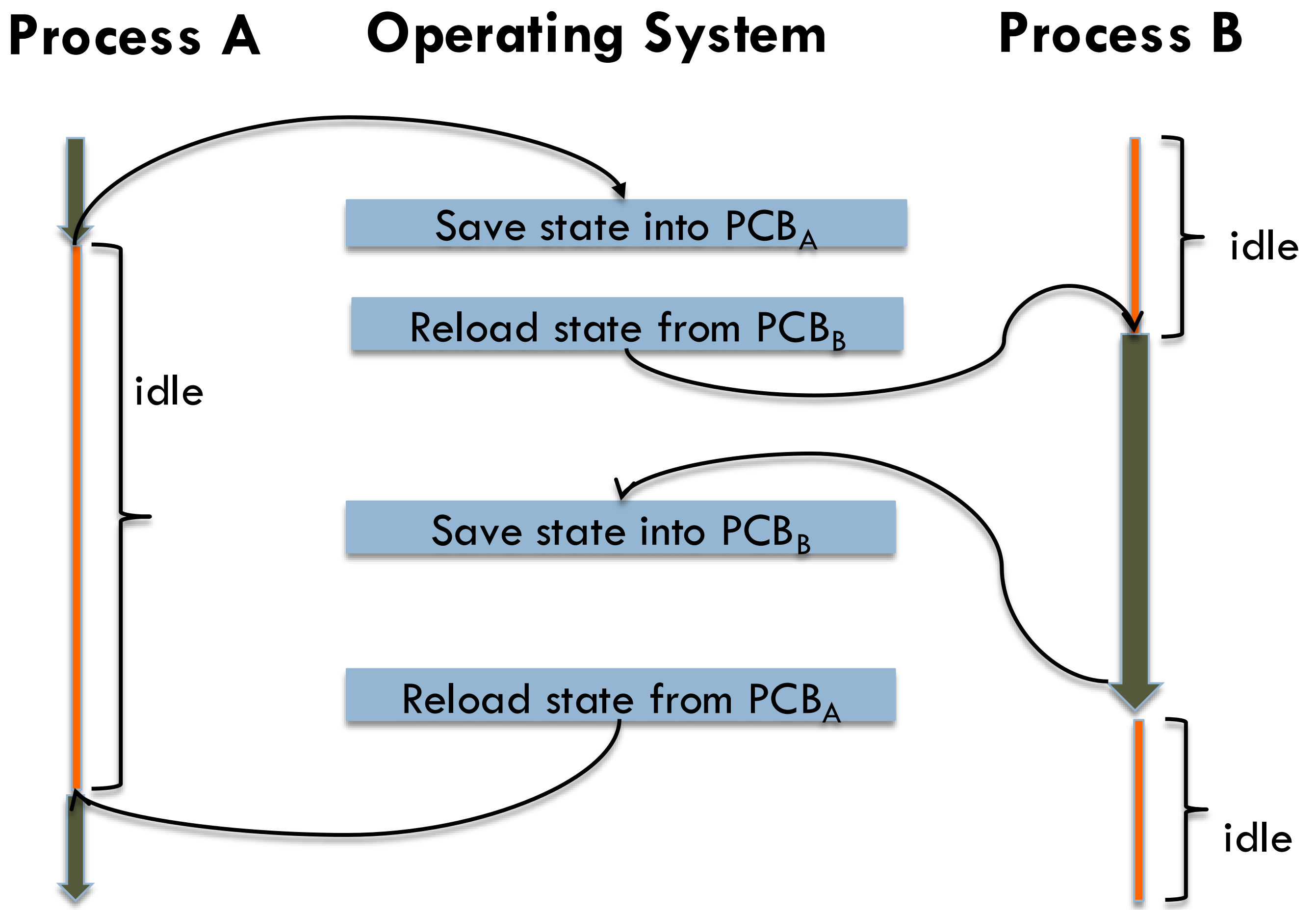


Each process is represented by a process control block (PCB)



PCB is a **repository** for *any* information that *varies* from process to process.

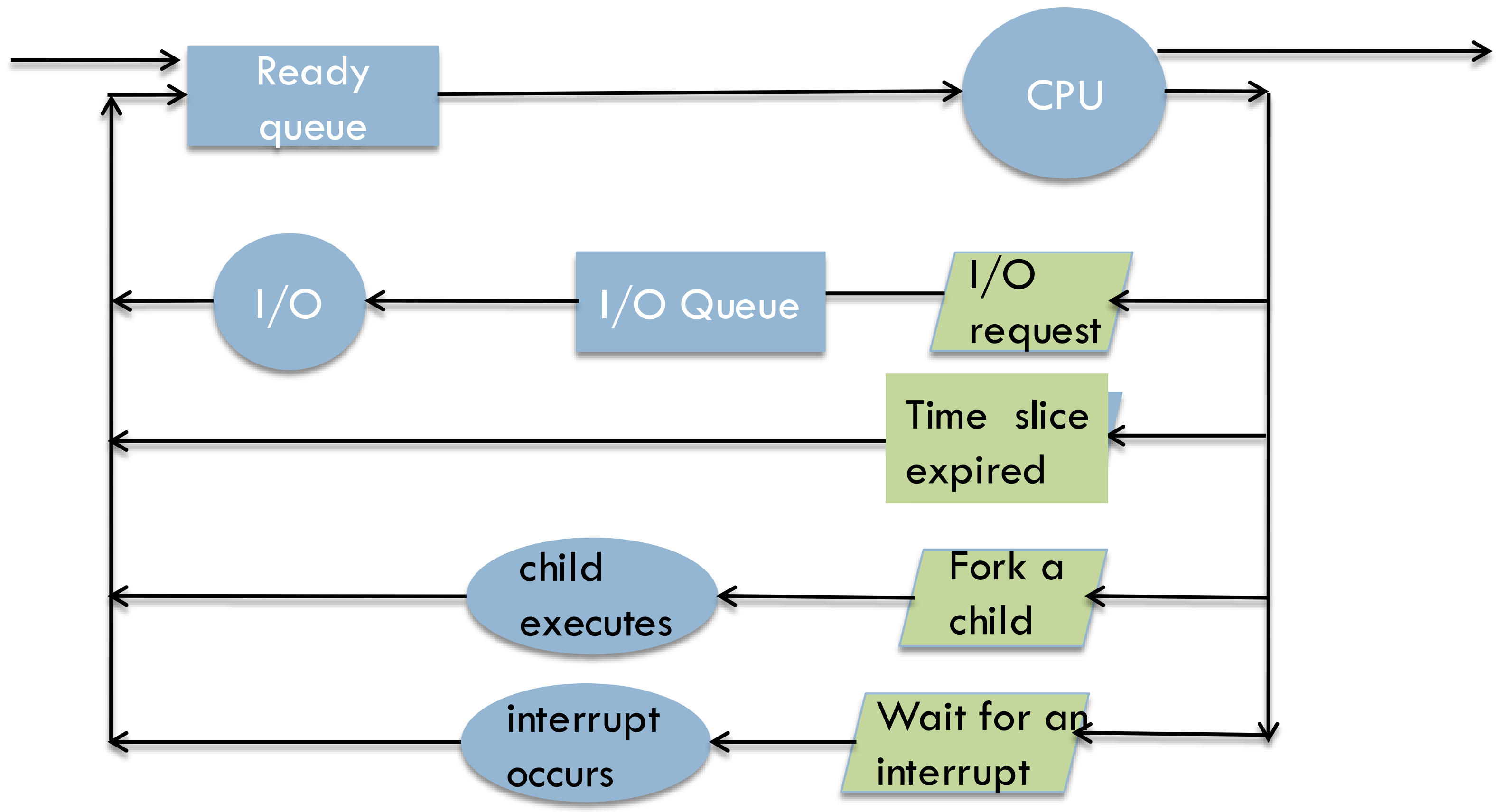
An example of CPU switching between processes



Scheduling Queues

- Job Queue: Contains all processes
  - A newly created process enters here first
- Ready Queue
  - Processes residing in main memory
  - Ready *and* waiting to execute
  - Typically a linked list
- Device Queue
  - Processes waiting for a particular I/O device

Process scheduling



Throughout its lifetime a process migrates among various scheduling queues

- Long-term scheduler: Batch systems
  - Executes much less frequently
  - Can take more time to decide what to select
- Short-term scheduler
  - Select process for CPU frequently
  - Selected process executes for few milliseconds
  - Typically, execute once every 10-100 milliseconds

UNIX and Windows systems often have no long-term scheduler

- Put **every** new process in memory for the short-term scheduler
- **System stability** depends on:
  - Physical limitations: Number of terminals
  - Self-adjusting nature of users

Somewhere in between: The medium term scheduler

- **PREMISE:** It can be advantageous to **reduce degree** of multiprogramming
  - Remove processes from memory
  - Reduce active contention for the CPU
- Reintroduce processes later on: ***Swapping***
- Swapping improves the *process mix*
  - Cope with strains on resources such as memory

Interrupts & Contexts

Interrupts and Contexts

- Interrupt causes the OS to **change** CPU from its current task to run a kernel routine
- Save current context so that ***suspend*** and ***resume*** are possible
- Context is represented in the **PCB**
  - Value of CPU registers
  - Process state
  - Memory management information

Context switch refers to switching from one process to another

① **Save** state of current process

② **Restore** state of a different process

- Context switch time is pure **overhead**
  - No useful work done while switching

Factors that impact the speed of the context switch

- Memory speed
- Number of registers to copy
- Special instructions for loading/storing registers
- Memory management: Preservation of address space

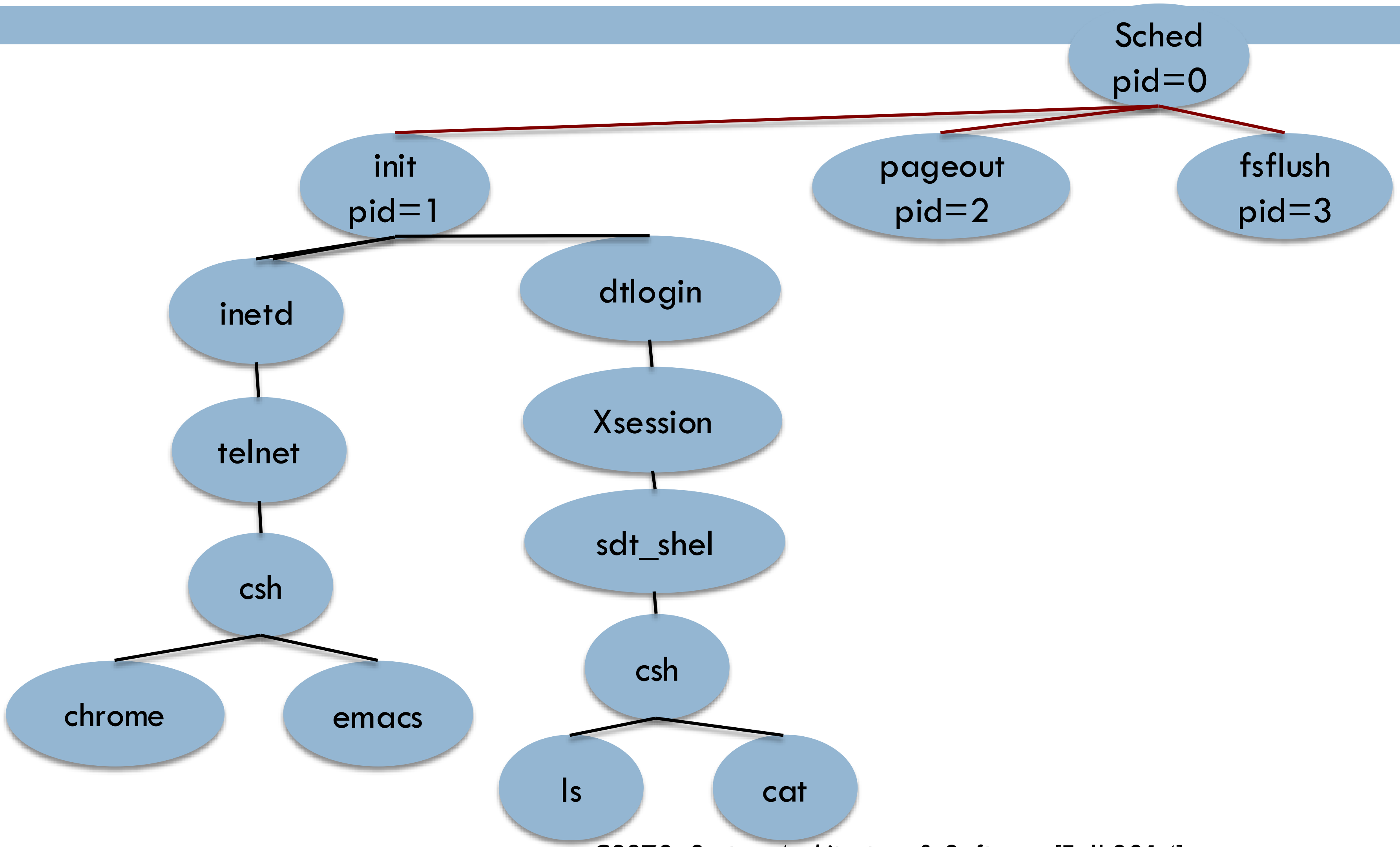
Processes execute concurrently
Can be created and deleted dynamically.

OPERATIONS ON PROCESSES

Process Creation: A process may create new processes during its execution

- **Parent** process: The creating process
- **Child** process: New process that was created
  - May itself create processes: **Process tree**
- All processes have ***unique*** identifiers

Example: Process tree in Solaris



Processes in UNIX

- `init` : Root parent process for all user processes
- Get a listing of processes with **`ps`** command
  - `ps`: List of all processes associated with user
  - `ps -a` : List of all processes associated with terminals
  - `ps -A` : List of all active processes

Resource sharing between a process and its subprocess

- Child process may obtain resources **directly from OS**
- Child may be **constrained** to a subset of parent's resources
  - Prevents any process from overloading system
- Parent process also passes along initialization data to the child
  - Physical and logical resources

Parent/Child processes: Execution possibilities

- Parent executes **concurrently** with children
- Parent **wait**s until some or all of its children terminate

Parent/Child processes: Address space possibilities

- Child is a **duplicate** of the parent
  - Same program and data as parent

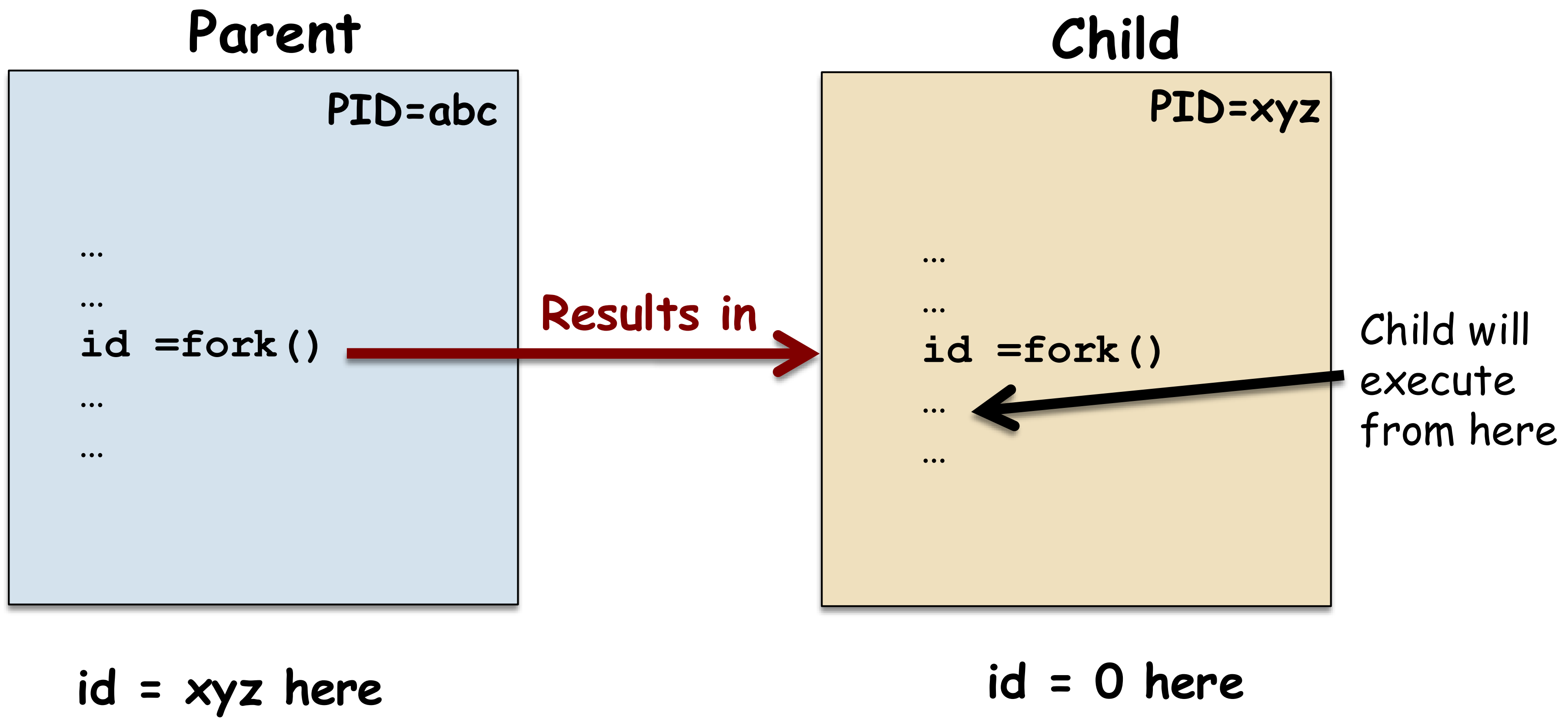
- Child has a **new program** loaded into it

Process Creation

Process creation in UNIX

- Process created using **`fork()`**
  - `fork()` copies parent's memory image
  - Includes copy of parent's address space
- Parent and child continue execution **at instruction after** `fork()`
  - Child: Return code for `fork()` is **0**
  - Parent: Return code for `fork()` is the non-ZERO *process-ID* of new child

`fork()` results in the creation of 2 distinct programs



Simple example:

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

int main(void) {
       int x;
       x=0;
       fork();
       x=1;
         …
}
```

Both parent and child execute this *after* returning from fork()

Another example

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

int main () {
    printf("Hello World\n");
    fork();
    printf("Hello World\n");
 }
```

```
Hello World
Hello World
Hello World
```

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

int main () {
    printf("Hello World\n");
    if (fork()==0) {
       printf("Hello World\n");
    }
}
```

```
Hello World
Hello World
```

What happens when `fork()` fails?

- No child is created
- `fork()` returns **-1** and sets `errno`
  - `errno` is a global variable in `errno.h`

If a system is short on resources OR if limit on number of processes breached

- `fork()` sets `errno` to `EAGAIN`

- Some typical numbers for Solaris
  - `maxusers`: 2 less than number of MB of physical memory up to 1024
    - Set up to 2048 manually in `/etc/system` file
  - `mx_nprocs`: **Default:** `16 x maxusers + 10`
    min = 138, max = 30,000

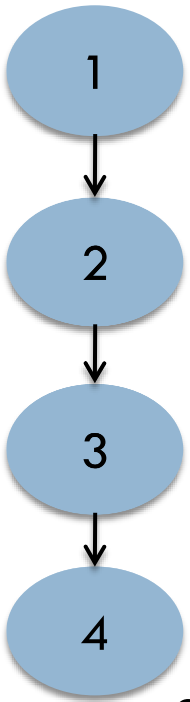
Take different paths depending on what happens with `fork()`

```
childpid = fork();
if (childpid == -1) {
    perror("Failed to fork");
    return 1;
}
if (childpid == 0) {
    ….. child specific processing
} else {
    ….. parent specific processing
}
```

Child (<u>any process</u>) can use **`getpid()`** to retrieve its process ID

Creating a chain of processes

```
for (int i=1; i < 4; i++) {
    if (childid = fork()) {
        break;
    }
}
```



value of **i**
when process leaves loop

<u>For each iteration:</u>

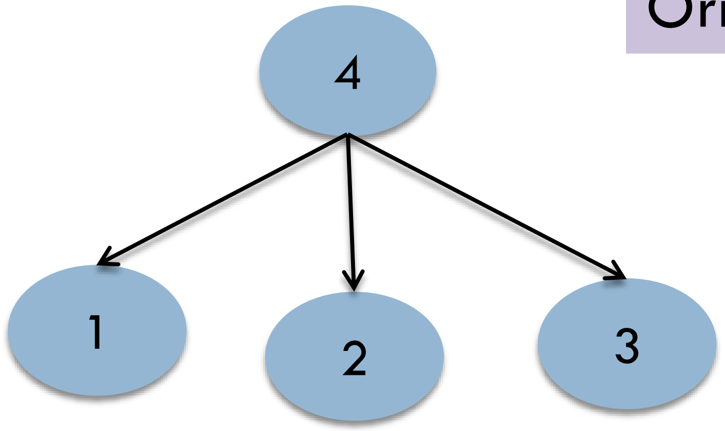
Parent has non-ZERO childid
- So it breaks out

Child process
- Parent in NEXT iteration

Creating a process fan

```
for (int i=1; i < 4; i++) {
    if ((childid = fork()) <= 0) {
        break;
    }
}
```

Newly created process breaks out
Original process continues

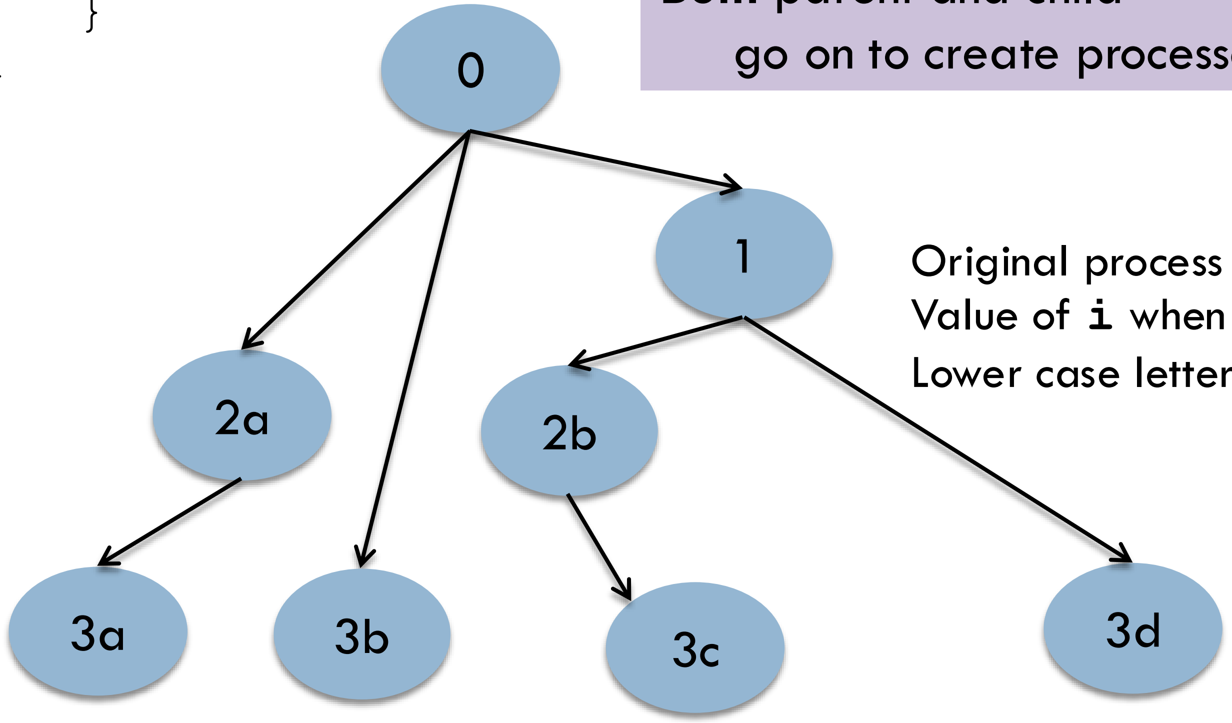


value of **i**
when process leaves loop

Creation of a process tree

```
int i=0;
for (i=1; i < 4; i++) {
   if ((childid = fork()) == -1) {
        break;
   }
}
```

Both parent and child
go on to create processes in the next iteration



Original process has a **0** label
Value of **i** when created
Lower case letters: Process created with same **i**

The contents of this slide-set are based on the following references

- *Avi Silberschatz, Peter Galvin, Greg Gagne. Operating Systems Concepts, 9th edition. John Wiley & Sons, Inc. ISBN-13: 978-1118063330.* [Chapter 3]

- *Andrew S Tanenbaum. Modern Operating Systems. 4th Edition, 2014. Prentice Hall. ISBN: 013359162X/ 978-0133591620.* [Chapter 2].

- *Kay Robbins & Steve Robbins. Unix Systems Programming, 2nd edition, Prentice Hall ISBN-13: 978-0-13-042411-2.*
  [Chapter 2, 3]