

**Ricardo Rocha**

**Department of Computer Science**

**Faculty of Sciences**

**University of Porto**

**Slides based on the book**

***‘Operating System Concepts, 9th Edition,***

***Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley’***

**Chapter 3**

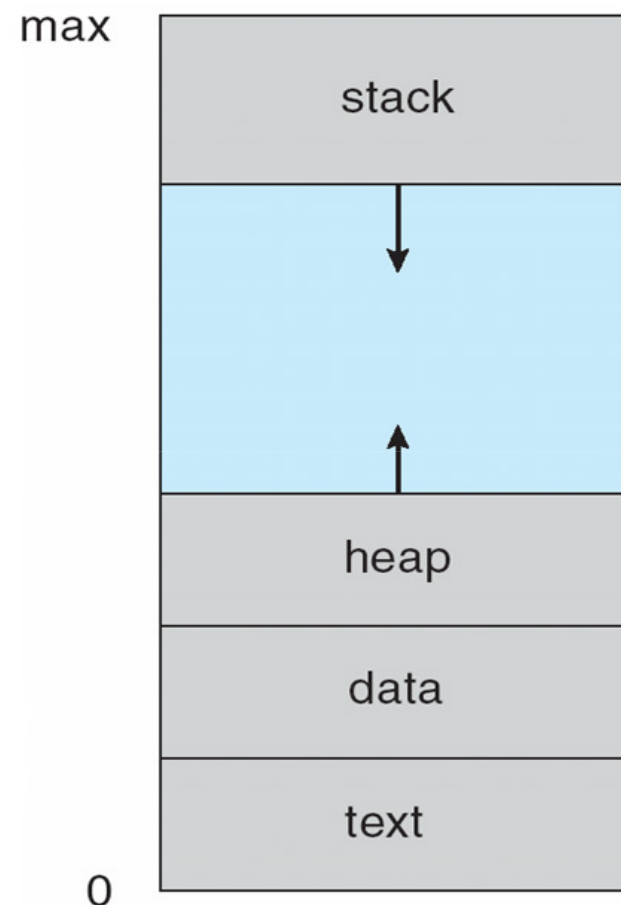
# Process Concept

---

- A **process** is the unit of work in most systems and can be thought of as a **program in execution**
  
- A process **requires resources** to accomplish its task:
  - CPU time
  - Memory
  - Files
  - I/O devices
  - ...
  
- Resources can be allocated to a process either **when it is created or while it is executing**

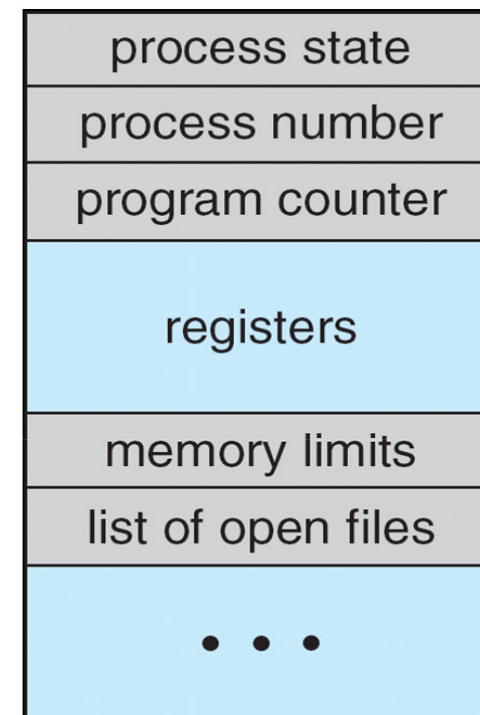
# Process Concept

- A process has multiple parts:
  - **Text section**, which contains the program code
  - **Data section**, which contains global variables
  - **Heap**, which contains memory dynamically allocated during process runtime
  - **Stack**, which contains temporary data (such as function parameters, return addresses and local variables)
  - Process control block, which includes the **program counter** and the **CPU registers**
- The text, data, heap and stack parts form the **process's memory address-space**



# Process Control Block (PCB)

- The operating system represents each process by a **process control block** (also called task control block), which contains many pieces of information associated with the process, including:
  - **Process state** and **process identification**
  - **Program counter**, the next instruction to be executed
  - **CPU registers**, general purpose registers, index registers, stack pointers, etc
  - **Scheduling information**, such as priorities, scheduling queue pointers, etc
  - **Memory information**, the memory allocated to the process
  - **Accounting information**, such as CPU used, clock time elapsed since start, time limits, etc
  - **I/O information**, such as I/O devices allocated, list of open files, etc



# Programs x Processes

---

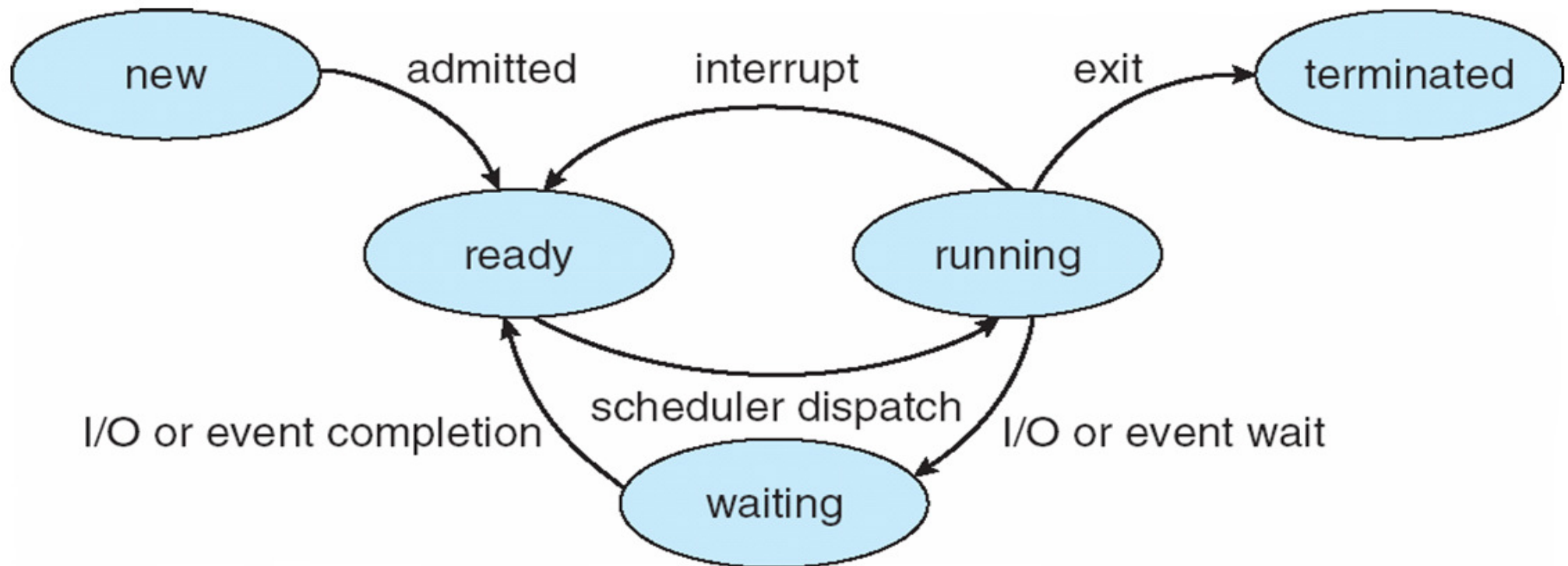
- A **program is a passive entity** (often called an **executable file**) that contains a well-defined list of instructions
- A **process is an active entity** corresponding to an execution sequence of a program
- A **program becomes a process** when it is **loaded into memory**
  - One program can be several processes
  - Two processes associated with the same program are considered two separate execution sequences
- For instance, several users may run different copies of the mail program, or the same user may invoke many copies of the web browser program
  - Each of these is a separate process and although the text sections are equivalent, the data, heap, and stack sections vary

# Process States

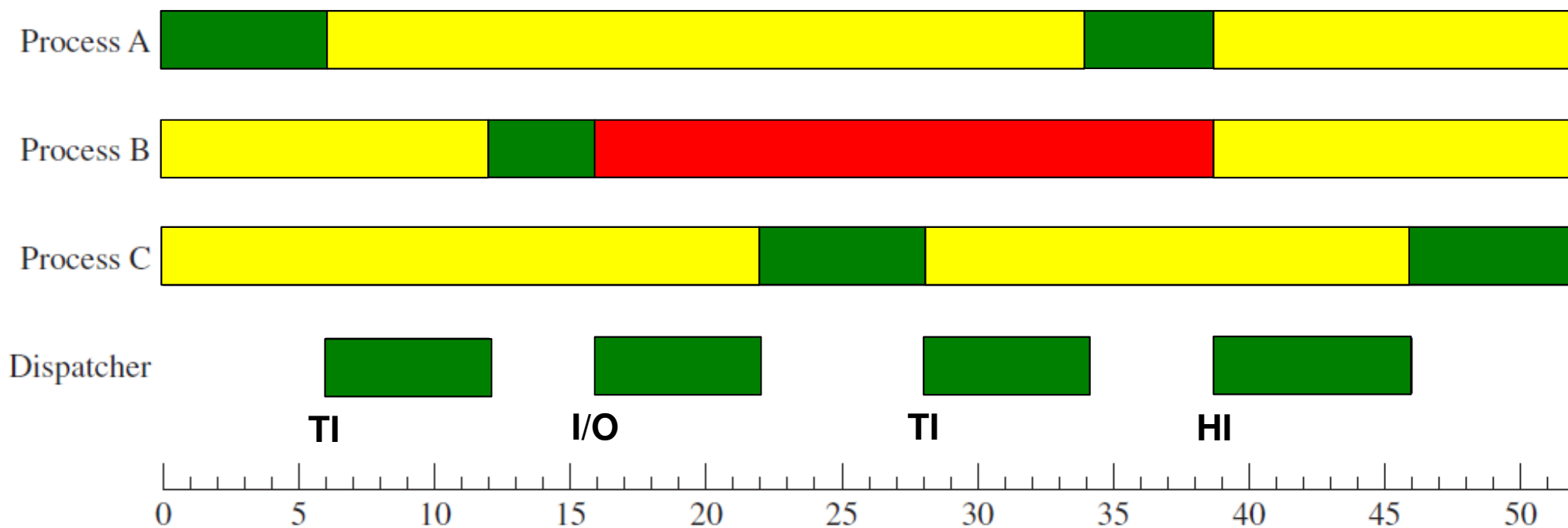
---


- As a process executes, it changes **state** accordingly to its **current activity**
- A process can be in one of the following states:
  - **New**, the process is being created
  - **Running**, instructions are being executed
  - **Waiting/Blocked**, the process is waiting for some event to occur (such as an I/O operation or reception of a signal)
  - **Ready**, the process is waiting to be assigned to a processor
  - **Terminated**, the process has finished execution
- It is important to realize that many processes may be ready or waiting, but **only one process at a time can be running on a processor**

# Process States



# Transition Among States



 = Running

 = Ready

 = Blocked

**TI:** timer interrupt  
**I/O:** I/O system call  
**HI:** hardware interrupt

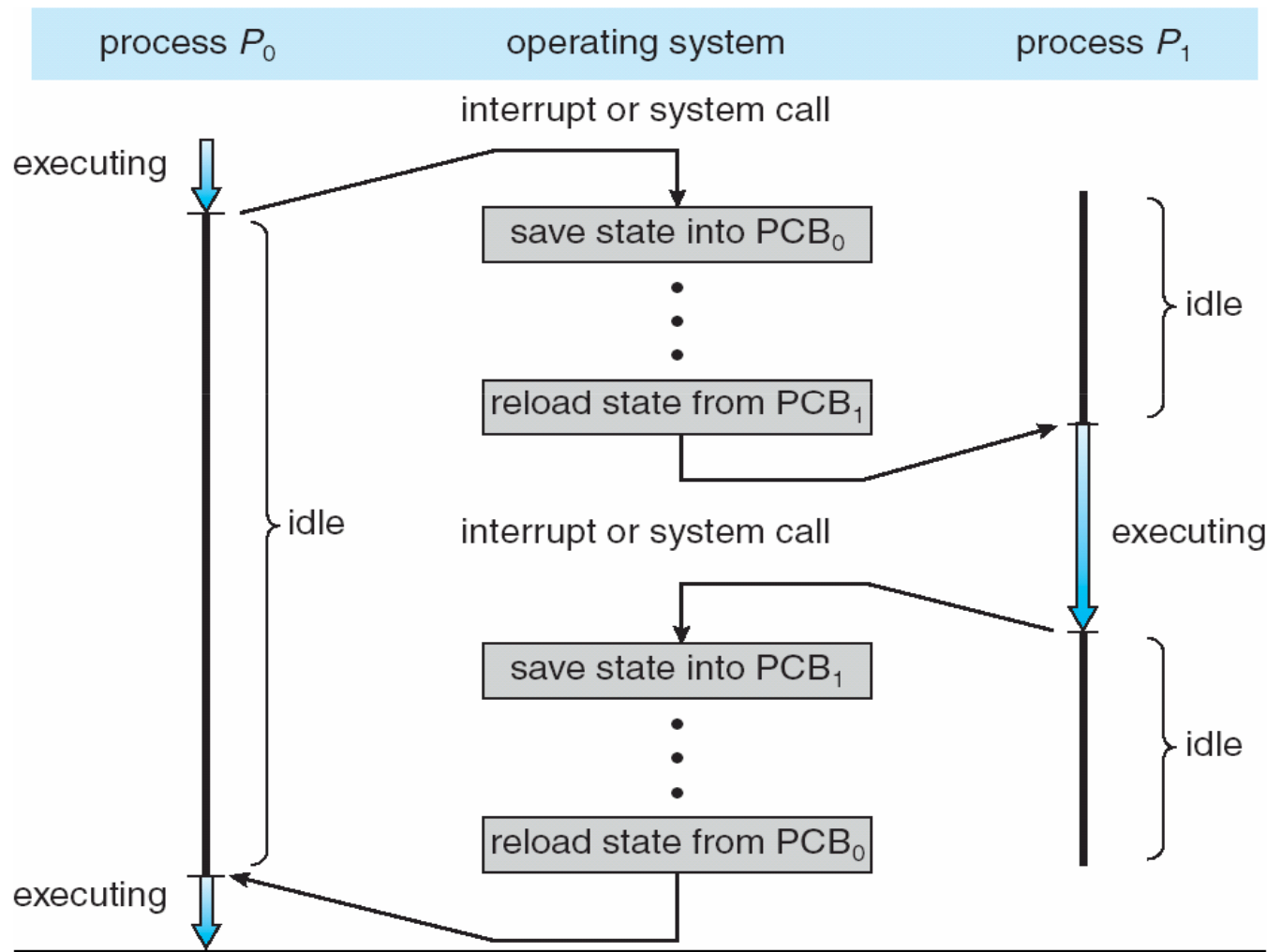


# Context Switch

---

- **Context switch** is the task of switching the CPU to another process
  - Context switch requires **saving the context of the current process to its PCB** and **loading the saved context for the new process from its PCB**
- Context switch time is **pure overhead**, because the system does no useful work while switching
  - A typical context switch speed is a **few milliseconds**
- Switching speed depends on the memory speed, on the number of registers that must be copied, and on the existence of special instructions
  - The more complex the OS and the PCB, the longer the context switch
  - Some machines support a special instruction that loads/stores all registers or provide multiple sets of registers per CPU (here, a context switch simply requires changing the pointer to the new register set)

# Context Switch



# Process Creation

---

- During execution, a process may create several new processes
  - The creating process is called a **parent** process and the new processes are called the **children** of that process
  - Each new process may in turn create other processes, forming a **tree of processes**
  
- Most operating systems identify processes according to a **unique process identifier (or pid)**, which is typically an **integer number**
  - The pid provides a unique value for each process in the system, and it can be used as an index to access various attributes of a process within the kernel

# Parent/Children Creation Alternatives

---

- Resource sharing alternatives:
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
  
- Execution alternatives:
  - Parent and children execute concurrently
  - Parent waits until some or all of its children have terminated
  
- Address space alternatives:
  - Parent and child are duplicates (child starts with parent's program and data)
  - The child process has a new program loaded into it

# What Does it Take to Create a Process?

---

- Assign process identifier and store new PCB
  - Inexpensive
- Copy parent's I/O state (I/O devices allocated, list of open files, etc)
  - Medium expensive
- Set up new memory tables for address-space
  - More expensive
- Copy data from parent process
  - Originally very expensive
  - Much less expensive with copy-on-write

# Process Creation – UNIX/Linux

---

- In UNIX/Linux, a new process is created by the **fork() system call**
  - The new (child) process consists of a **copy of the address space of the original (parent) process**
  - The **return code** for the fork() is **zero for the child** and the (nonzero) **process identifier of the child is returned to the parent**
  - Both processes then continue **execution concurrently** at the instruction after the fork()
  - This mechanism allows the parent process to communicate easily with its children

# Process Creation – UNIX/Linux

---

```
... // parent code before fork
if ((pid = fork()) < 0) {
    ... // fork failed
} else if (pid == 0) {
    ... // child code after fork
} else {
    ... // parent code after fork
}
... // common code after fork
```

# Process Creation – UNIX/Linux

---

- Typically after a `fork()`, one of the two processes (parent or child) invokes the **`exec()` system call**
  - `exec1("/bin/ls", "ls", NULL)`
- The `exec()` system call **replaces the process's memory space – text, data, heap and stack parts – with a brand new program from disk and starts executing the new program at its main function, destroying the previous image of the process**
  - However, since no new process is created, the calling **process keeps its context (PCB)**
  - In this manner, parent and child are able to go in separate ways



# Process Termination

---

- A process terminates execution when it explicitly invokes the **exit() system call** or when it **executes the last instruction** (in this case, exit() is called implicitly by the return statement in main())
  - All process' resources – including physical and virtual memory, open files, and I/O buffers – are then deallocated by the operating system
  - A **exit status value** (typically an integer) is made available to the parent via the **wait() system call**
  
- A process may also terminate execution via its parent:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating system does not allow a child to continue if its parent terminates (cascading termination)

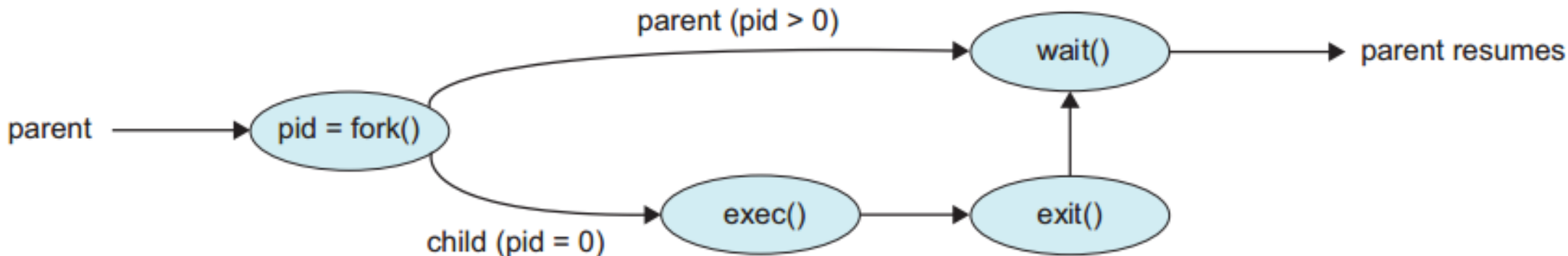
# Process Termination

---

- If a process has terminated and no parent is waiting, then the terminated process is known as a **zombie process**
  - Only when the parent calls `wait()`, the pid of the zombie process and its entry in the process table are released
  
- If a parent process terminates before their children, then such children, if any, are known as **orphan processes**
  - UNIX/Linux address orphan processes by **assigning the init process as the new parent to orphans processes**
  - The init process periodically invokes `wait()`, thereby allowing the orphan's process identifier and process table entry to be released

# Process Termination – UNIX/Linux

- In UNIX/Linux, we can terminate a process by invoking the `exit()` system call, providing an exit status as a parameter
  - `exit(status)`
- To wait for the termination of a child process and obtain its exit status, we can invoke the `wait()` system call, which also returns the pid of the terminated process so that the parent can tell which children has terminated
  - `pid = wait(&status)`



# Forking a 'ls' Command – UNIX/Linux

---

```
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;
}
```

# Interprocess Communication

---

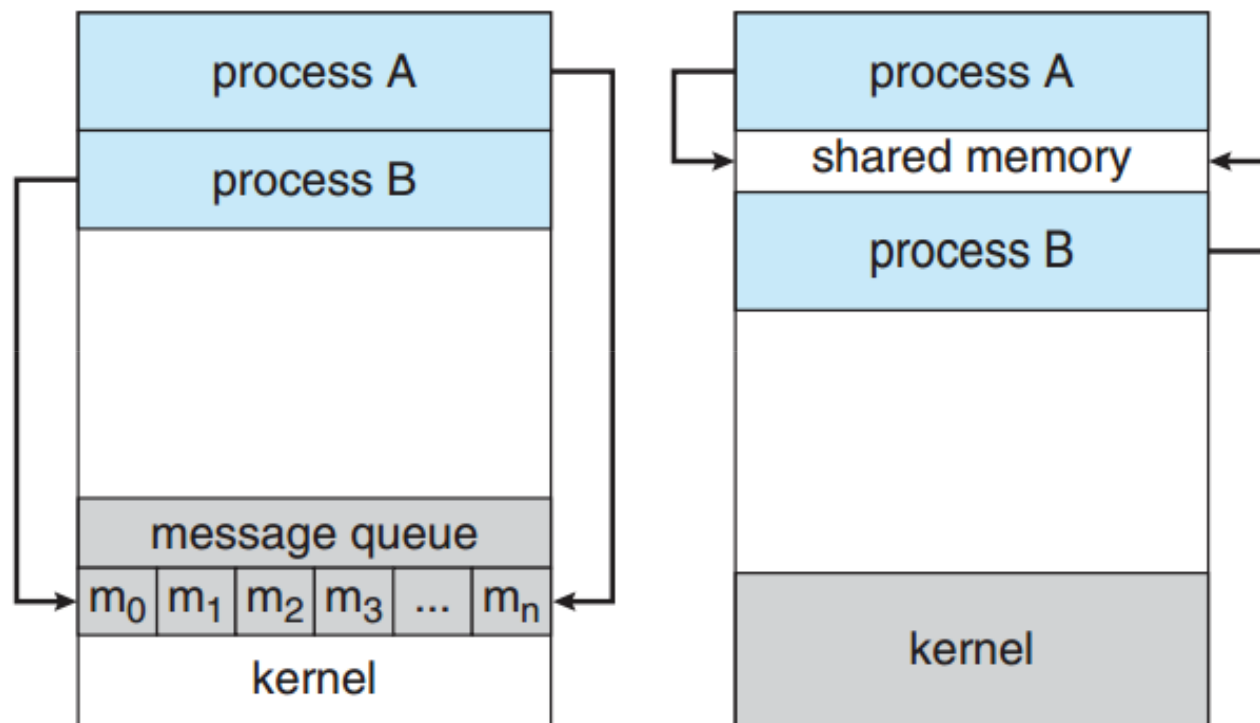
- Processes within a system may be independent or cooperating:
  - **Independent process** cannot affect or be affected by others' execution
  - **Cooperating process** can affect or be affected by others' execution
  
- Main reasons for cooperating processes:
  - **Information sharing**, concurrent access to the same piece of information
  - **Modularity**, break the computation in subtasks that make more sense
  - **Speedup**, execute concurrent subtasks in parallel
  
- To exchange data and information, cooperating processes need support for **interprocess communication (IPC)** mechanisms. There are two fundamental IPC models:
  - **Message passing**
  - **Shared memory**

# IPC Models

---

- Message passing
  - Communication via sending/receiving messages
  - Works across network
  
- Shared memory
  - Communication occurs by simply reading/writing to shared memory
  - Can lead to complex synchronization problems
  
- Typically, shared memory is faster than message passing
  - **Message passing** mechanisms are typically implemented using systems calls and thus **require a time-consuming task of kernel intervention**
  - With **shared memory**, system calls are required only to establish the shared memory regions and, after that, all accesses are treated as routine memory accesses and **no kernel intervention is required**

# IPC Models



# Pipes

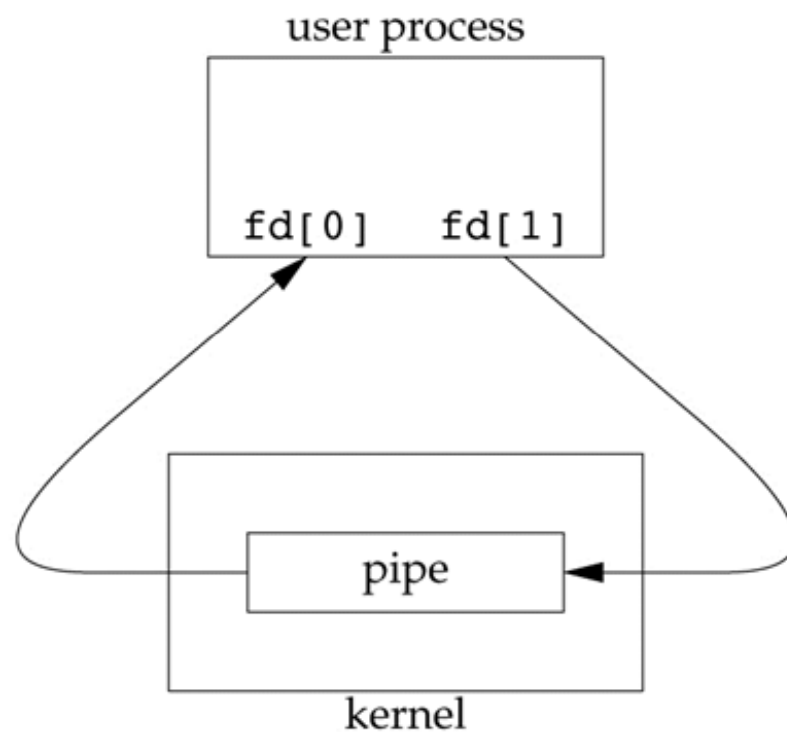
---

- Pipes were one of the first IPC mechanisms in early UNIX systems and provide one of the simpler ways for processes to communicate
- Pipes allow communication in a standard producer–consumer style:
  - The producer writes to one end of the pipe (the **write-end**)
  - The consumer reads from the other end (the **read-end**)
- Pipes are **unidirectional**, allowing only **one-way communication**
  - If two-way communication is required, two pipes must be used, with each pipe sending data in a different direction
- Pipes can be constructed on the command line using the | character:
  - `ls | sort`
  - `cat fich.txt | grep xpto`



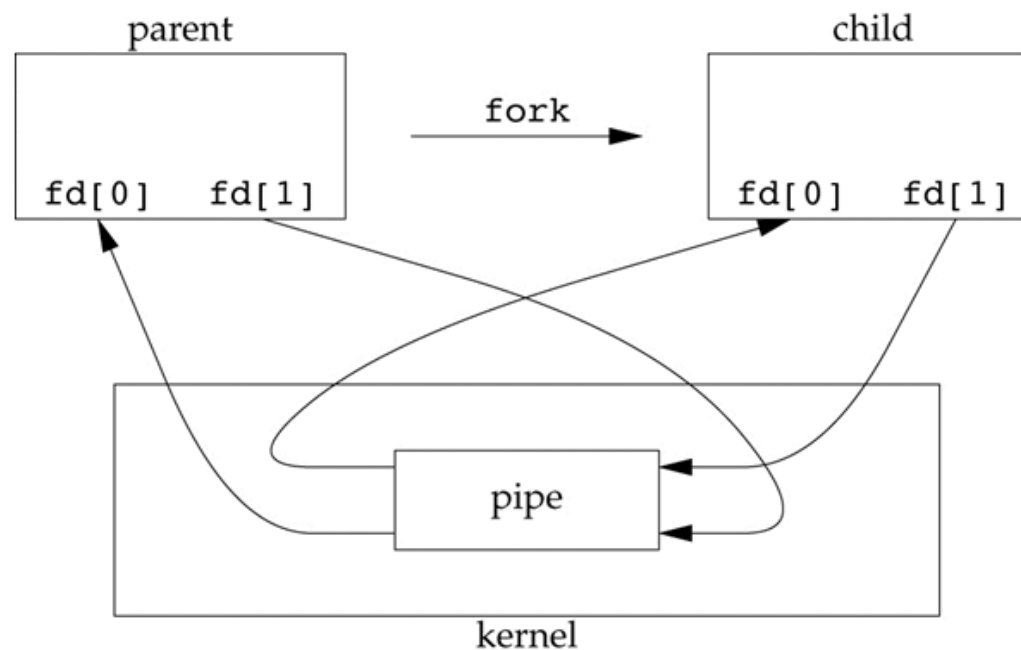
# Pipes – UNIX/Linux

- In UNIX/Linux, pipes are created using the **pipe()** system call
  - `int pipe(int fd[2])`
- The pipe() system call creates a new pipe and initializes fd[2] with the **pipe file descriptors**:
  - fd[0] is the read-end of the pipe
  - fd[1] is the write-end of the pipe
- UNIX/Linux systems treat pipes as a special type of file, which allows **pipes to be accessed using ordinary read() and write() system calls**



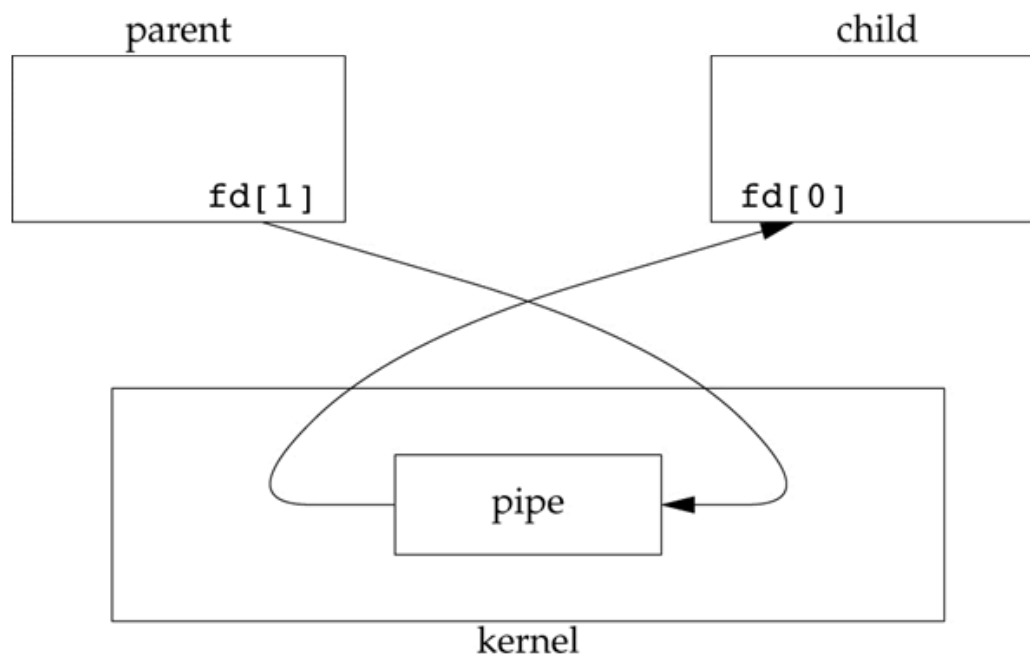
# Forking a Pipe

- A pipe cannot be accessed from outside the process that created it
  - Typically, a parent process creates a pipe and uses it to communicate with a child process that it creates via `fork()`
  - Since a pipe is a special type of file, the **child inherits the pipe from its parent process**



# Pipe from Parent to Child

- After a fork, we can decide the pipe's data flow direction
  - For a pipe from parent to child, the **parent closes the read-end of the pipe `fd[0]`** and the **child closes the write-end `fd[1]`**
  - Reading from an open pipe (i.e., at least one process has `fd[1]` open) blocks while it is empty



# Pipe Communication – UNIX/Linux

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    /* create the pipe */
    if (pipe(fd) == -1) {
        fprintf(stderr, "Pipe failed");
        return 1;
    }

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }

    if (pid > 0) { /* parent process */
        /* close the unused end of the pipe */
        close(fd[READ_END]);

        /* write to the pipe */
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

        /* close the write end of the pipe */
        close(fd[WRITE_END]);
    }
    else { /* child process */
        /* close the unused end of the pipe */
        close(fd[WRITE_END]);

        /* read from the pipe */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s", read_msg);

        /* close the write end of the pipe */
        close(fd[READ_END]);
    }

    return 0;
}
```