Chapter 3: Processes

Chapter 3: Topics

- Process Concept.
- (An introduction to) Process Scheduling.
- (Some) Operations on Processes.
- Interprocess Communication (IPC).

Objectives

- To introduce the notion of a process → a program in execution, which forms the basis of all computation.
- To describe the various features of processes, including scheduling, creation and termination, and communication (by using system calls).

Introduction

- A process is one of the central concepts in operating systems.
  - Chapter 1: one of the goals of an OS is to regulate program execution.
- Process is "a program in execution".
- To execute, it will need access to certain resources ?
- CPU time, memory, files, I/O devices to accomplish its goal.
  - OS allocates & regulates these resources.
- A process is a unit of work in operating systems.
- Two kinds of processes:
  - User processes (e.g. -> web browser).
  - System processes (e.g. -> CPU scheduler).

Introduction

- Just mentioned that a process is "a program in execution".
- Evaluating the "goodness" of an OS is by observing how many processes it can execute per unit time (throughput, **will discuss in chapter # 5**).
- Early operating systems allowed only one program to be executed at a time (think DOS).
  - That program had complete access to all the system resources.
- With better hardware, systems can now support multiple programs to be loaded into memory and executed concurrently.
- Can open the "activity monitor" in a Mac or "task manager" in Windows to see current processes (user + system).
- The CPU (s) are shared among the processes in order to make the system more productive - multiprogramming.

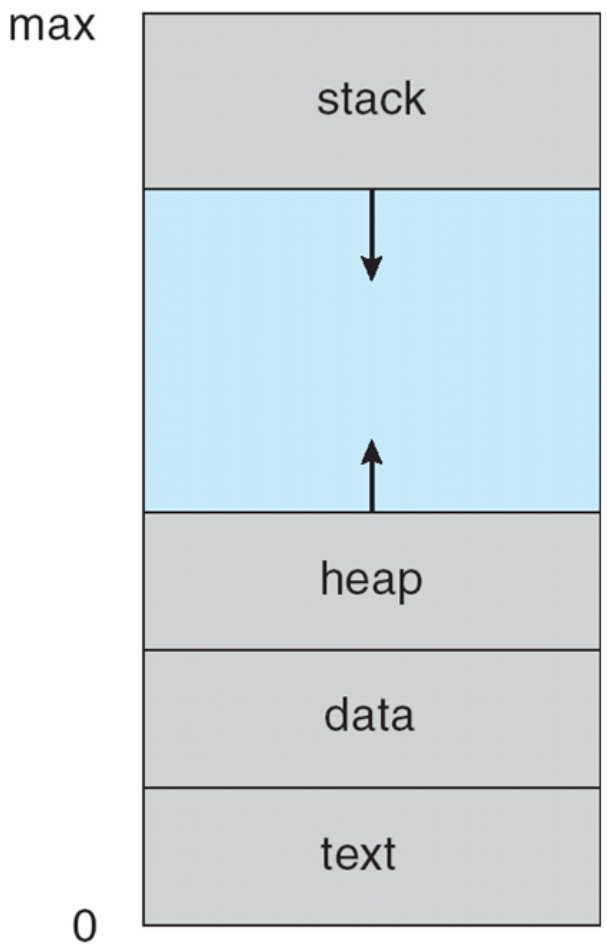
/proc Directory in Linux

- On your Linux terminal, type: cd /proc.
- This takes us to the directory of all processes.
- Each process itself is a directory named after it's process ID.
- Go to a specific process by typing, say: cd 1151.
- For example, one can see the scheduling information about a process by typing: vi sched or vi status.
  - Status information – process ID, user ID, state, memory size, number of threads, number of context switches.
  - Scheduling information – start time, execution time, sleep time etc.

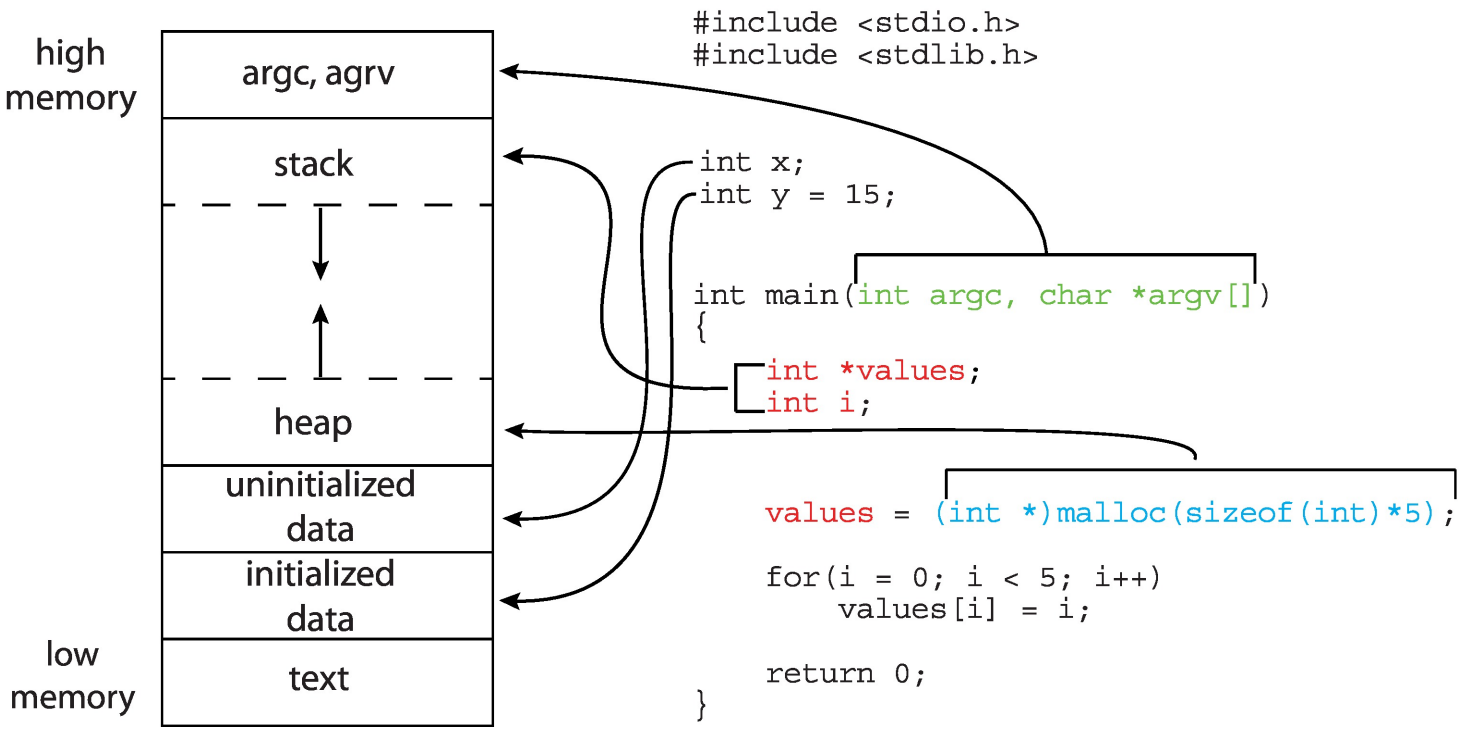
Process Concept

- Textbook uses the terms ***job*** (old name) and ***process*** (new name) interchangeably.
- What does a process contain?
- It contains multiple parts (see picture on next slide).
  - The (program) code, also called **text section**: an executable file.
    - Includes current activity, i.e. **program counter**, contents of processor registers.
  - **Stack** containing temporary data.
    - Function call & parameters, return addresses for functions, local variables.
  - **Data section** containing global variables.
  - **Heap** containing memory dynamically allocated during run time (using *malloc ( )*).

Parts of a Process in Memory



Memory Layout of a C Program



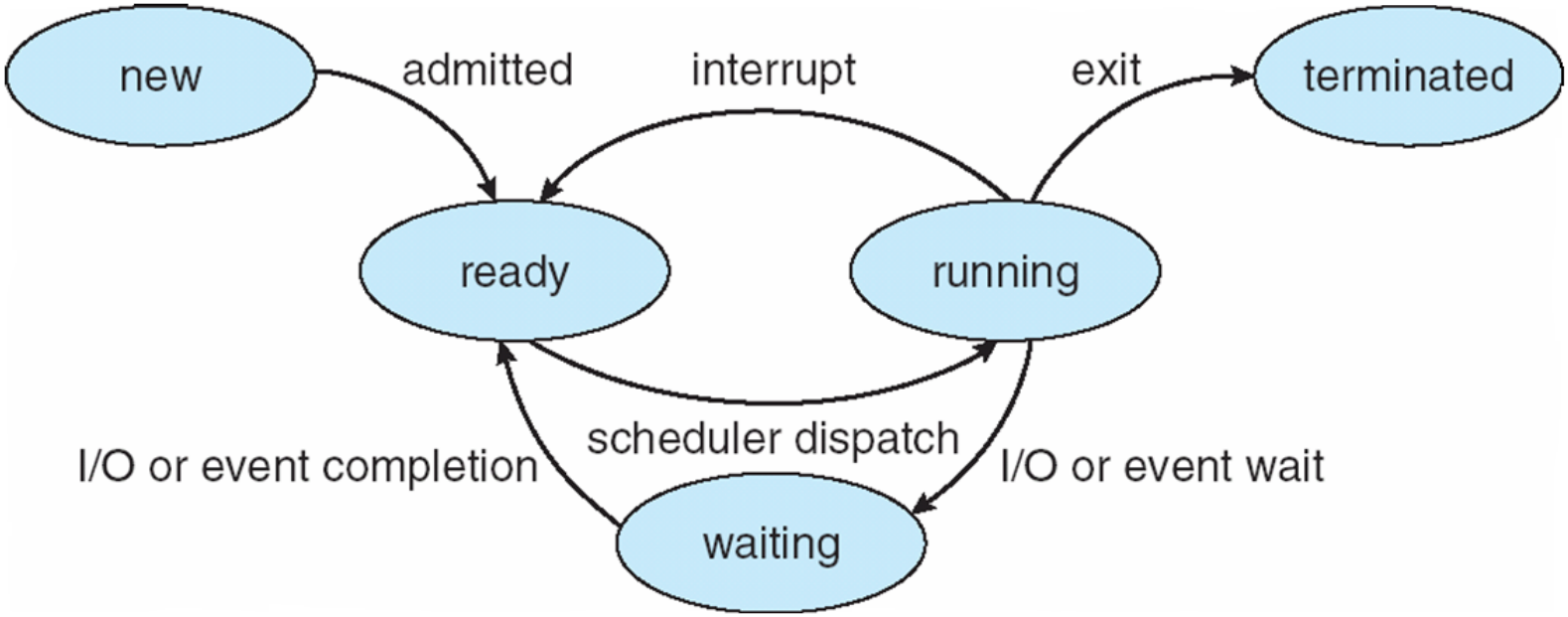
Process Concept (Cont.)

- **Program vs. process?**
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active (means that instructions are being executed).***
  - Program becomes process when executable file loaded into memory.
- **How does a program become a process?**
  - via GUI mouse clicks, command line entry of its name.
- Note: one program can spawn several processes:
  - Consider multiple users executing the same program on a multi-user machine.
  - Each process has its own rate of execution sequence.
  - Text (code) section is same, but the data, stack & heap will vary.
- Separation of process code & data facilitates sharing of code (will see this later in memory management chapter).

Process State

- As a process executes, it changes *state (status),* which are as follows:
  - **new**: The process is being created.
  - **running**: Instructions are being executed.
  - **waiting**: The process is waiting for some event to occur, typically an I/O completion.
  - **ready**: The process is waiting to be assigned to a processor.
  - **terminated**: The process has finished execution.
- Process states are visible in top/htop command output.
- State names can change for different operating systems.
  - Actual states, however, are the same across operating systems.
- Note that on a single CPU, only one process can be running at a time.
  - However, many processes may be in a ready or waiting state.
- Next slide shows the “state diagram” for a process.

Diagram of Process State



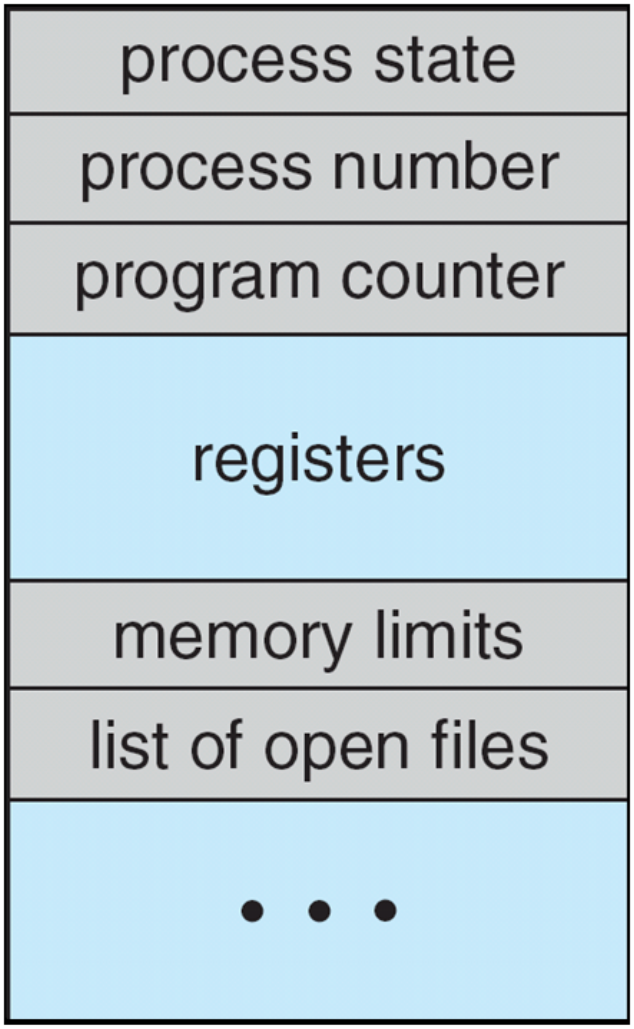
Process Control Block (PCB)

- How is a process represented in the OS?
- Each process is represented in the OS by a data structure called a process control block (PCB).
- It contains information associated with a specific process, such as:
  - **Process state**
    - Ready, running, waiting, etc.
  - **Program counter**
    - Indicating address of next instruction.
  - **CPU registers**
    - Info on registers must be saved when an interrupt occurs, to allow the process to resume its execution later.
  - **CPU scheduling information**
    - Includes process priority, pointers to scheduling queues, scheduling parameters.

Process Control Block (PCB)

- **Memory-management information**
  - Values of base and limit registers, page table information etc.
- **Accounting information**
  - CPU time used, timer limit, process ID.
- **I/O status information**
  - List of I/O devices allocated, open files.
- In short, a PCB is a place for storing the information of processes.
- When an running process (say, P_1) becomes idle (stops executing), its state needs to be saved to its PCB_1.
- State needs to be reloaded from PCB_1 before P_1 starts executing again later.

Process Control Block (PCB)



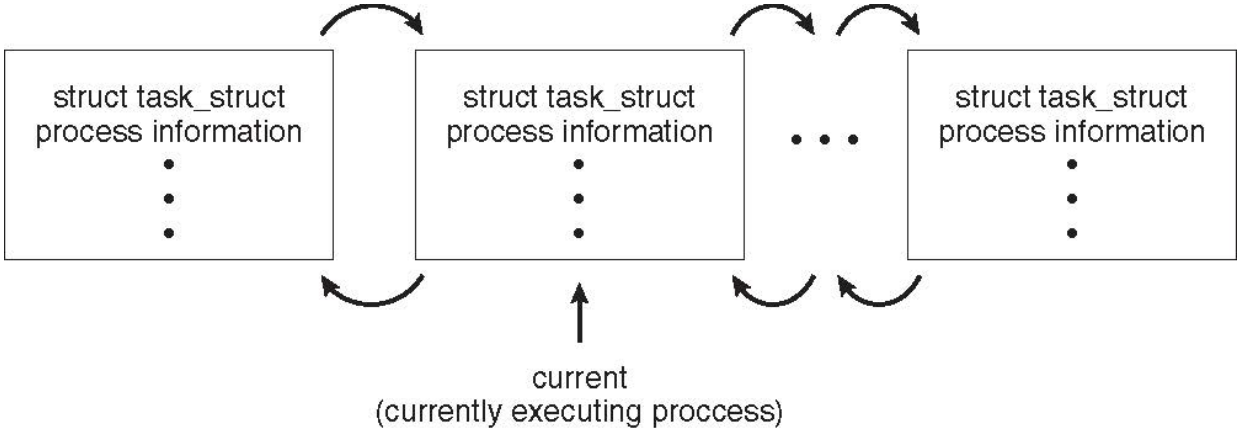
Process Representation in Linux

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 */
```

Complete implementation available at:

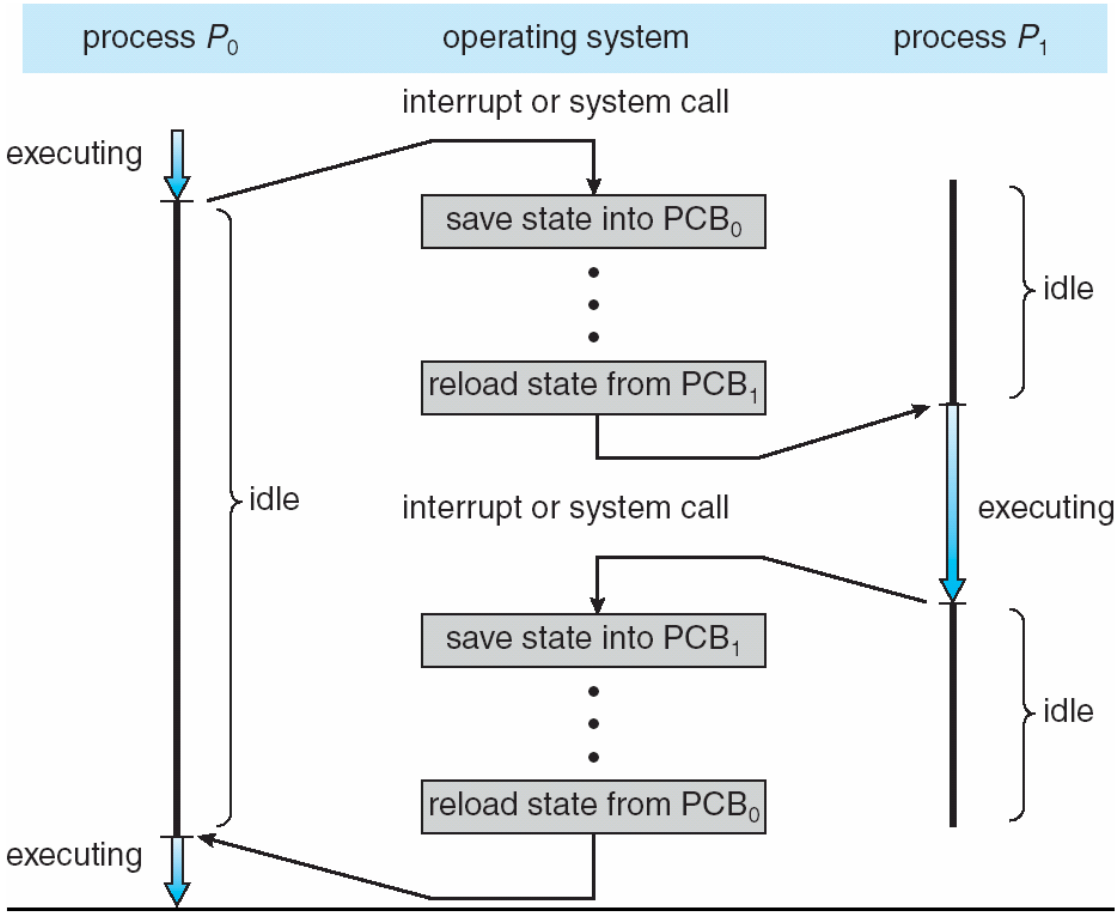
https://github.com/torvalds/linux/blob/master/include/linux/sched.h



Context Switch

- This is needed to support multiprogramming.
- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch.**
- **Context** of a process represented in the PCB.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.
  - Memory speed, # registers to copy.
  - Typical context switch time in micro seconds.
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once.

CPU Switch From Process to Process



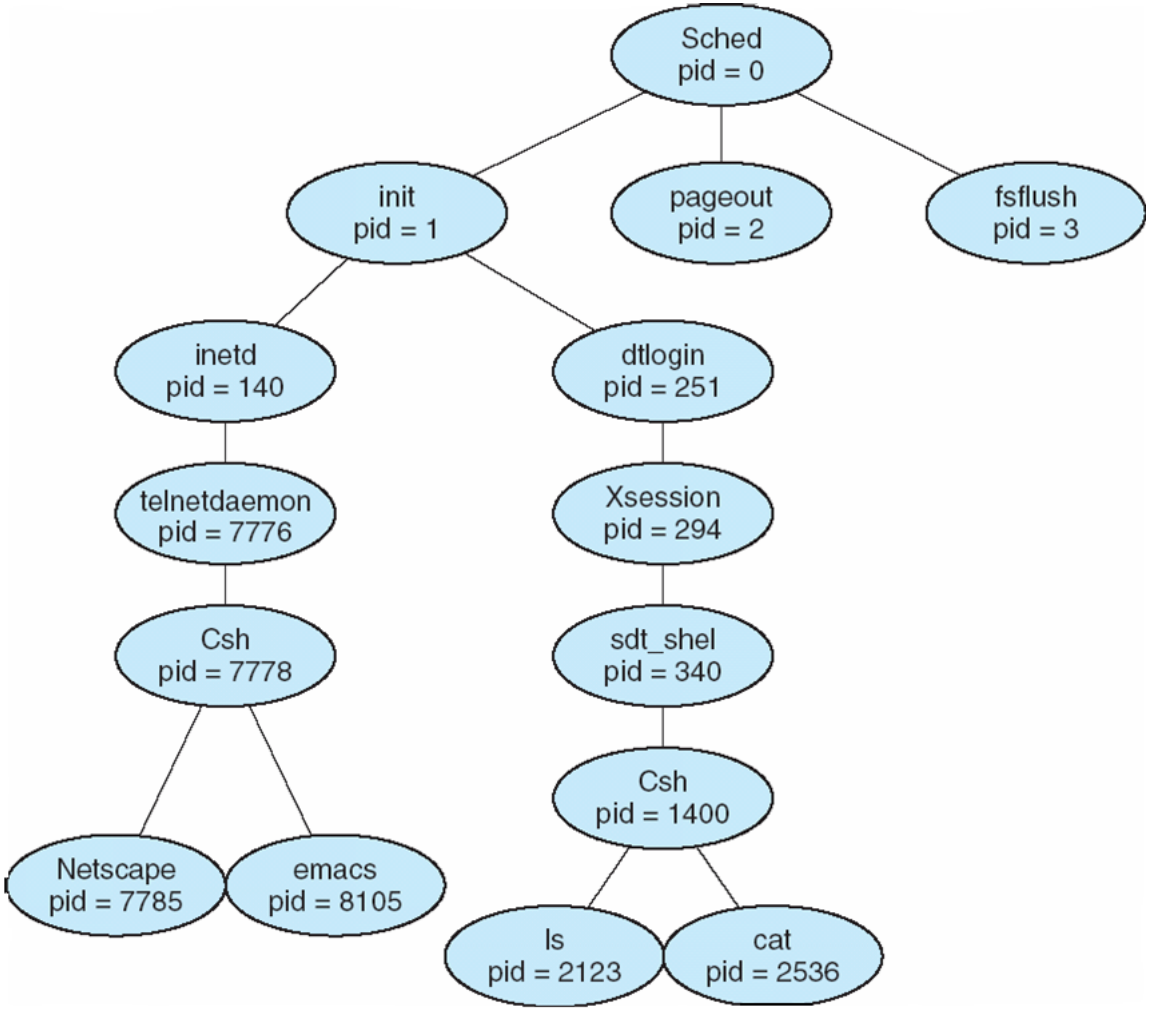
Switching among Processes

- Process A is running on CPU.
- We want to interrupt it's execution, & give the CPU to another process, B.
- Need to save some state information for A. What kind of information?
  - Program counter value (address of next instruction to be executed).
  - Contents of CPU registers (may be needed by future instructions).
- This state information is stored in the PCB for A.
- CPU given to B.
- If B was interrupted earlier, it's state is loaded from it's PCB.
- When A gets it's turn to run later, it's state is loaded from it's PCB.
- In short, the PCB serves as a storage place for process specific information.

Operations on Processes

- Processes in modern operating systems can execute concurrently.
- They can also be created and deleted dynamically (at run time).
  - Compare this to processes that start automatically. Example?
- Now, we will study the mechanism for process creation and termination in Linux.
- How is a process identified uniquely?
  - By its process id (pid), which is an integer.
- A process may, during the course of its execution, create multiple new processes.
  - The creator process is called the parent process.
  - The processes created are called child processes.
- Those child processes, in turn, may further create more processes.
- Hence, a process tree may be formed, shown on next slide.
  - Recall how we used ps command to print the process tree.

Sun Solaris OS Process Tree



Process Tree (continued)

- The root is the sched process, with pid = 0.
- init: root process for all user processes.
- pageout: memory management process.
- fsflush: file system process
- inetd: process responsible for networking services, such as telnet and ftp.
- xsession: user session process.
- csh: Shell process.
- netscape: web browser process.
- emacs: text editor.
- cat: command to look up and combine files.
- On Linux, ps – el will give us a list of all currently active processes.
  - ps is a command to provide process information.
- Please check out the man (manual) pages in Linux for more information on these processes.

Process Creation

- Process need certain resources (CPU time, memory, files) during its lifetime.
- **Many models of resource sharing.**
  - Parent and children share all resources allocated to parent.
  - Children share subset of parent's resources.
  - Parent and child share no resources (child gets resources from OS).
- **Execution models.**
  - Parent and children execute concurrently.
  - Parent waits until children (all, or some) terminate.
- For example, a process that needs to display an image, may get the image path and the output medium from its parent process.
- Let us now study the fork ( ) system call of Unix/Linux.
  - Used to create child process(es).

Fork ( ) System Call

- Used to create new processes in Linux.
- Takes in no arguments and returns a process id.
- New process consists of a copy of the (memory) address space of the original process.
  - This includes the original process code.
- What if we want the new process to run some other program?
  - The exec ( ) system call take care of this.
- Exec ( ) loads a binary executable file of a suitable program and starts its execution.
- In this way, the original and new processes can go their separate ways.
- Let us look at an example.

C Program Forking Separate Process

```
int main()
{
pid_t  pid;
    /* fork another process */
    pid = fork();
    if (pid < 0) { /* error occurred */
          fprintf(stderr, "Fork Failed”);
    }
    else if (pid == 0) { /* child process */
          execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
          /* parent will wait for the child to
    complete */
          wait ();
          printf ("Child Complete");
    }
}
```

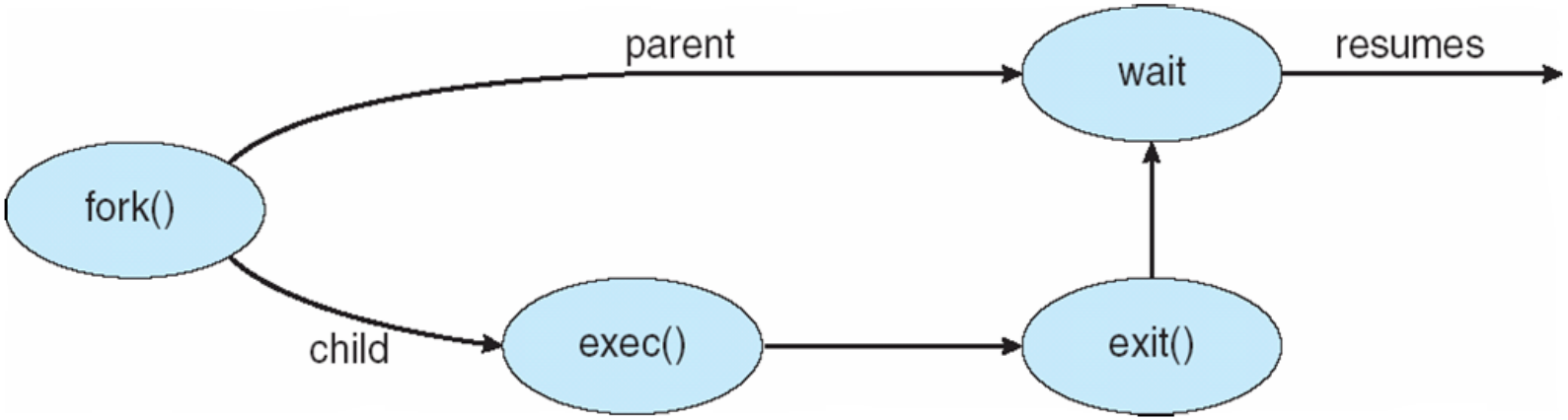
Example Program

- Original process forks a child process.
- From that point onwards, both processes continue execution from the instruction right after the fork.
- **Important question – how do we distinguish between the two processes?**
  - Answer lies in the return value of the fork.
- Three cases:
  - If fork returns a negative value: fork was unsuccessful.
  - Fork returns 0 to the newly created child process.
  - Fork returns a positive value to the original parent process. This value is actually the process identifier (pid) of the child process, so that the parent knows about the child's identity.
- Please note the difference between the pid value returned by fork and the actual pids assigned to the parent and child.
  - Returned pid values are just to distinguish between the parent and child.

Process Creation (Cont)

- Address space
  - Child duplicate of parent.
  - Child can then have a different 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
- Let us now look at some more examples of the **fork** system call.

Process Creation Diagram



Process Termination

- Process terminates using the exit system call.
  - Can write exit (0) for normal termination: good convention.
- In case of errors, one may make an explicit exit ( ) call as well.
- Process terminates when it executes last statement and asks the operating system to delete it (by invoking **exit**).
  - Output data from child to parent (via **wait**). What data?
  - Process' resources (memory, open files, I/O buffers) are deallocated and reused by operating system.
- Termination can occur in other circumstances as well.
  - A process can end another process (parent can choose to terminate child). How will it know child's identity?
    - Note that the pid of child has already been passed to parent when **fork** is called.
  - **What if parent process exits?**

Process Termination

- In Unix/Linux, process termination: exit.
- Parent of process may use the wait call to wait for the termination of its child.
- Process terminates, entry in process table may remain (pid remains). Why?
  - Process table needs info on exit status of all processes.
- How do we know exit status of a process?
- This is supplied to parent via wait.
- Zombie process: process called exit, but parent not called wait. May be for brief while.
- What if parent does not call wait, and terminates?
- The init process can become parent.
- It can periodically invoke wait to free the process table entries of all such zombie process.

Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process (visible on the screen).
  - Visible process (service to foreground).
  - Service process (e.g. streaming music).
  - Background process (e.g. checking mail).
- Android may begin terminating processes that are least important.
- Not so common anymore, with good hardware.
- Used to be common earlier (Samsung Galaxy Note 3).
- https://developer.android.com/guide/components/activities/process-lifecycle

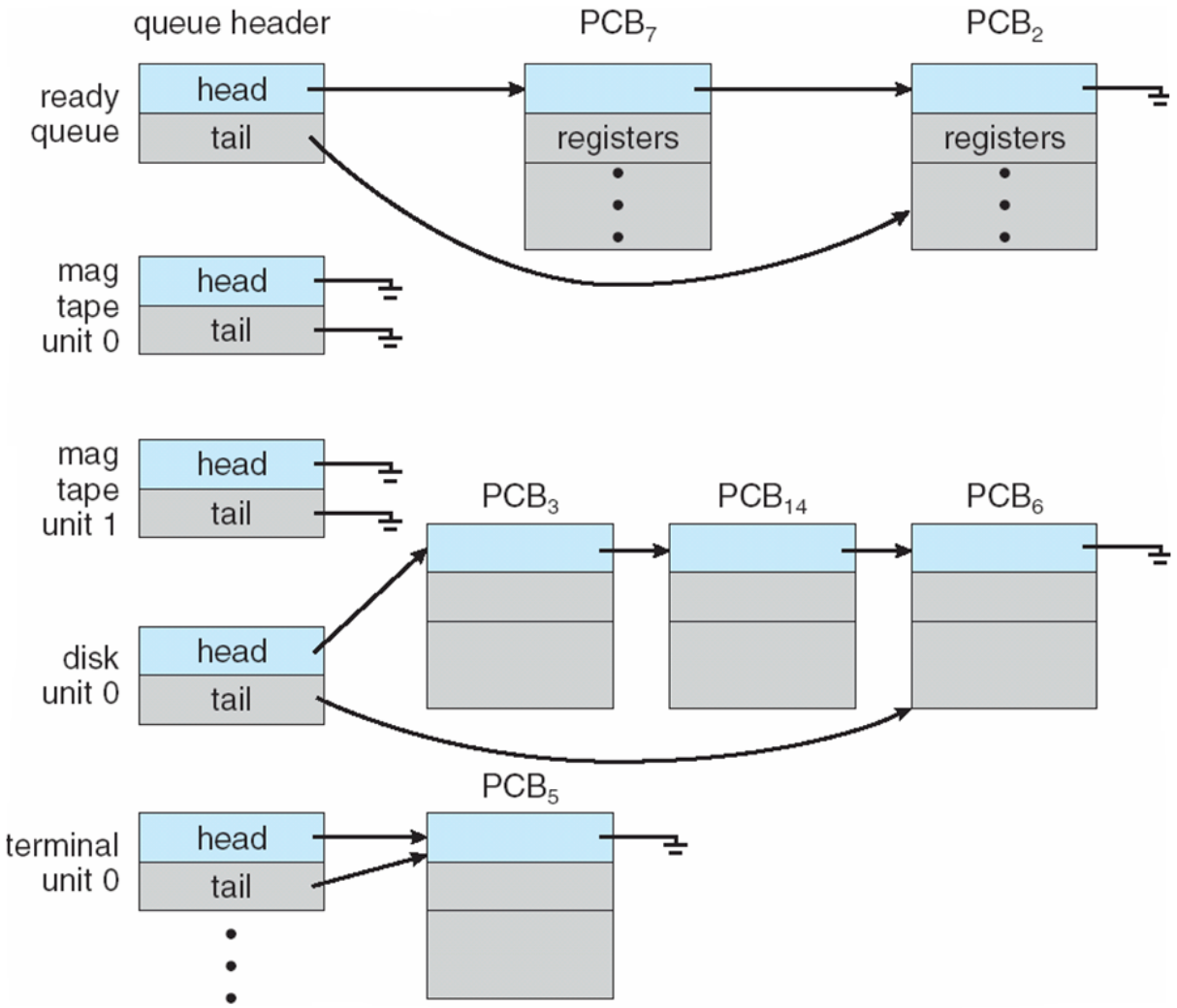
Process Scheduling

- Objective of multiprogramming?
  - Have some process running at all times, to maximize CPU utilization.
- How is this done?
  - By time sharing: switch the CPU among processes so frequently that users can interact with a program while it is running.
- Clearly, there is a need to schedule processes.
  - Selecting an order of execution for processes based on certain objectives.
- Every OS has a process scheduler.
  - Selects an available process (that is ready to execute) for execution on the CPU.

Process Scheduling

- We know that for a single CPU → only one process can run at a time.
- However, it may be the case that many processes "want" to run.
- Hence, we need a queue to store such processes, i.e. a ready queue.
  - Has processes in main memory that are waiting to execute.
  - Implemented as a linked list of PCBs.
- Need some other queues as well.
- Device queue (s)
  - A specific device, such as a disk, might be needed by multiple processes.
  - Each device has its own device queue that stores such processes.
- Wait queue.
  - Name is self explanatory. Which processes in here?
  - Processes waiting for an event to occur, say I/O.

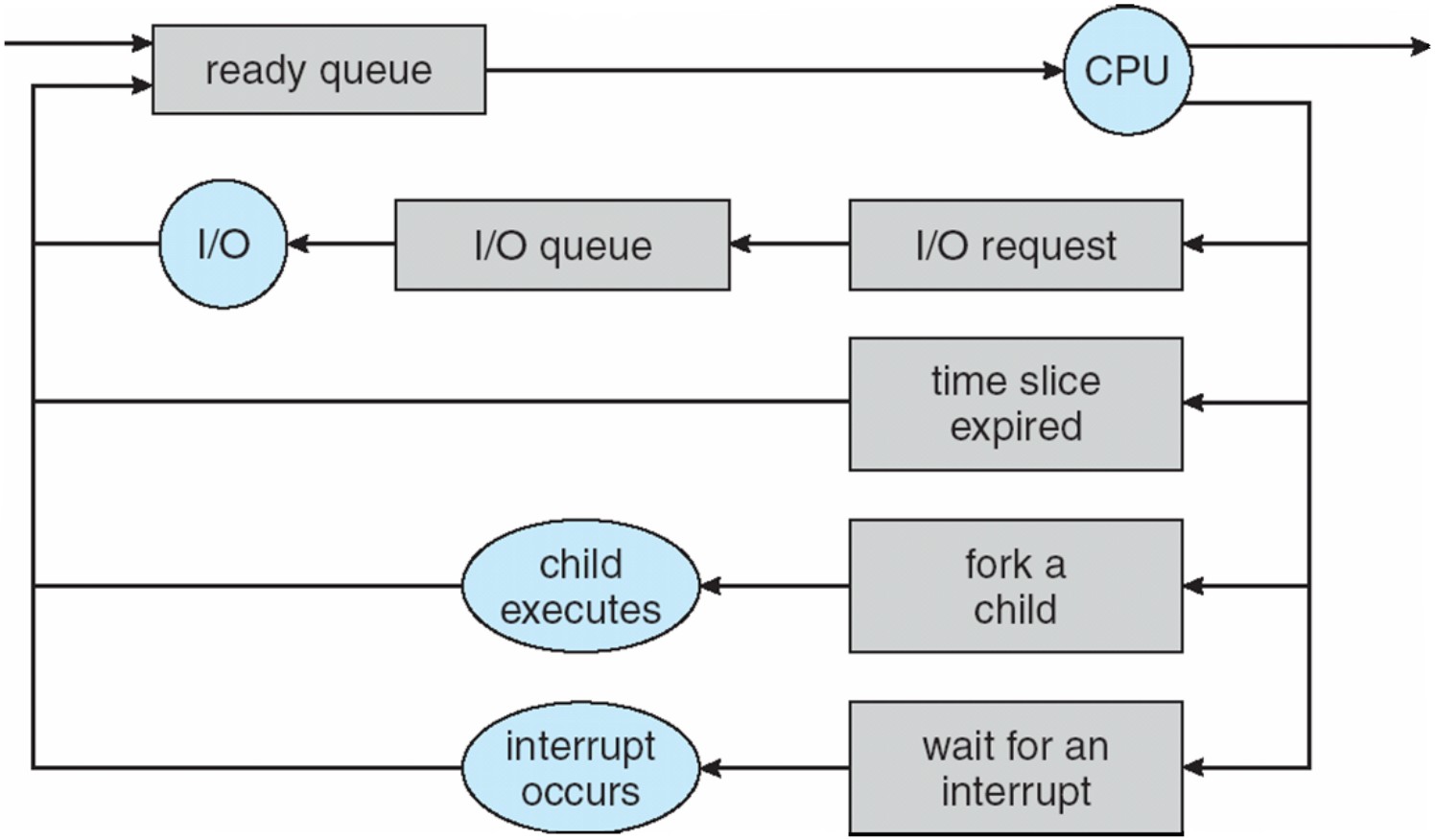
Ready Queue And Various I/O Device Queues



Queuing Diagram

- A process can move from one queue to the other.
  - Whole system is dynamic.
  - Initially, it can be in the job queue.
  - Eventually, it moves to the ready queue.
  - After running for some time, it could move to the device queue.
  - May have many such movements between queues.
- This can be represented by a queuing diagram (next slide).
- Several possibilities exist:
  - After executing for some time, process:
    - Could issue an I/O request and move to the I/O queue.
    - Could create a new sub process and wait for its termination.
    - Could be removed forcibly from the CPU (timer interrupt), and then moved to the ready queue.
- Ultimately, a process will finish, at which time, it is removed from the queue and will have its PCB and resources deallocated.
  - So that other processes may use.
- Think about how all this is modeled in code.

Representation of Process Scheduling



Schedulers: Long Term and Short Term

- The OS scheduler module needs to select processes from the various queues using some algorithm (more in chapter 5).
- Processes may be submitted to the system, one after the other, for execution.
- We may have different kinds of schedulers in the OS.
- **Long-term scheduler**
- Selects which programs are admitted to the system for execution, and when, and which ones should not be admitted (this is called - admission control).
- More prevalent in shared systems, such as HPC facilities.
- Also used in specialty systems, like real-time systems.
- Selects which processes should be brought into the ready queue.
- Picks a set of processes from disk and loads them into memory for execution, eventually.
- The execution may not start immediately.

Short Term Scheduler

- **Also known as: CPU scheduler**
  - Selects which process from the ready queue should be executed next and allocates CPU to it.
- Long term vs. short term schedulers: the frequency of their execution. Which would have greater frequency?
- Short term scheduler may need to pick a new process for the CPU frequently.
  - Process may run on CPU for short time and then wait for user input (which could take a longer time, relatively).
- Since, it may need to pick a new process frequently, it is desirable that it is fast (*employs an efficient scheduling algorithm*).
  - If it takes 10 milliseconds to decide which process to run next, and if a process runs for 100 milliseconds, what % of CPU is used for scheduling work?

Long Term Scheduler

- In comparison, the long term scheduler may need to execute less frequently.
  - E.g., minutes may separate the creation of a new process.
- This scheduler controls the degree of multiprogramming in the system. How is this defined?
  - Defined as the number of processes currently in the memory.
- If the average rate of process creation = average rate of process departure => degree of multiprogramming is stable.
- If creation rate >> departure rate, we have an overloaded system.
- What is the problem with an overloaded system?
  - CPU and memory utilization is very high.
  - Slow response times.

CPU Bound vs. I/O Bound Processes

- I/O bound process spends most of its time doing I/O, rather than computation. Example?
- CPU bound process, in contrast, generates I/O requests infrequently, using more of its time doing computation. Example?
- No consensus on actual %, though.
- Important for long term scheduler to select a good process "mix" of CPU bound and I/O bound processes. Why?
- What would happen if **all** processes were I/O bound?
  - Not much work for short term scheduler.
  - CPU under utilized.
  - I/O device queues full.

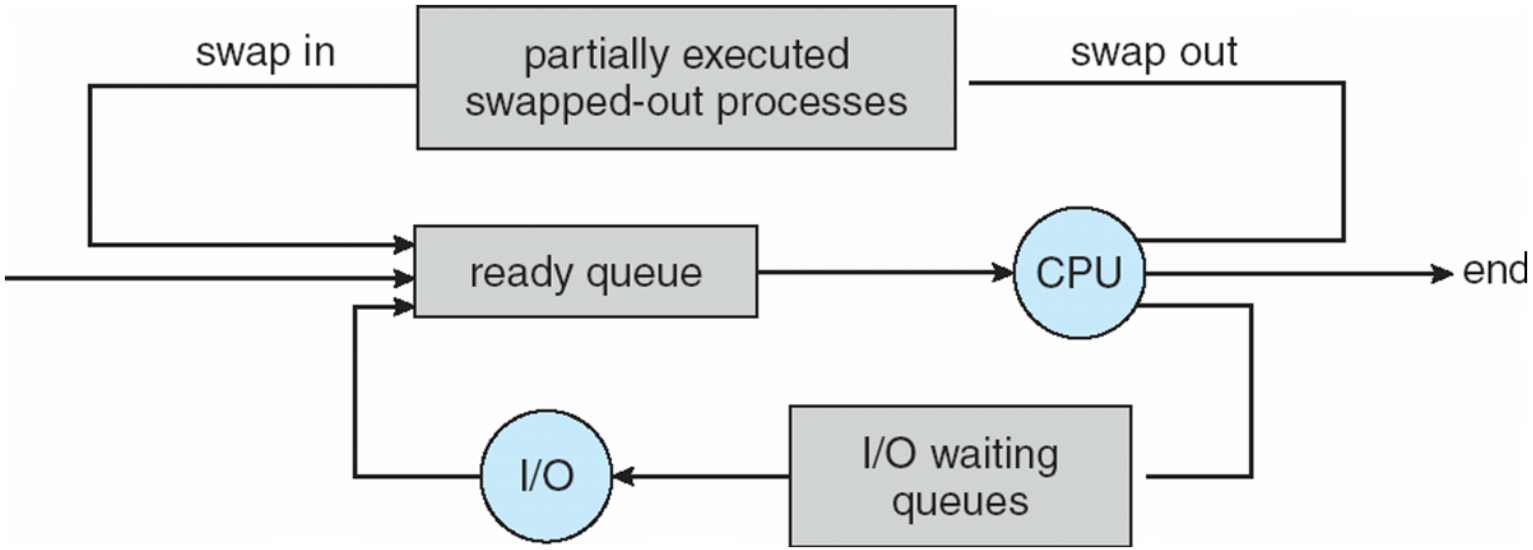
CPU vs. I/O Bound Processes

- Likewise, what would happen if **all** processes were CPU bound?
  - Devices would be little used.
  - CPU super busy.
  - Unbalanced system.
- **Need a good combination of CPU and I/O bound processes for best performance.**

Schedulers

- On some systems, long term scheduler may be minimal or absent.
  - Microsoft Windows, Unix, Android, iOS.
- In this case, every process is forwarded to the short term scheduler.
- Stability of these systems would depend upon:
  - Hardware limitation (amount of RAM).
  - Self adjusting nature of users.
    - If performance is bad, quit one or more processes.
- Some systems (particularly time sharing systems) introduce one more layer of scheduling – medium term in between long and short term schedulers.
- Sometimes, it can be useful to remove processes from memory (and active contention from the CPU), and thus < degree of multiprogramming.
  - Interrupted process may be brought back in later.
- Can be used to improve process mix.
- Change in process memory requirements might require memory to be freed up.

Addition of Medium Term Scheduling



Interprocess Communication (IPC)

- Processes within a system may be need to **cooperate/share information.** **Examples?**
  - **Example: a client process and a server process → web server and web browser client.**
  - **Another example: a shared Word document on Google drive.**
  - **Yet another example: copy and paste.**
- **Why bother with process cooperation/sharing?**
  - **Information sharing**
    - Several users interested in same info, e.g. shared file.
  - **Computation speedup**
    - Break up process into parts for parallel execution on multiple processors – need to put these parts together.
  - **Modularity of the OS itself**
    - Divide system functions into separate processes - modular OS.
    - A module needs to communicate with another module.

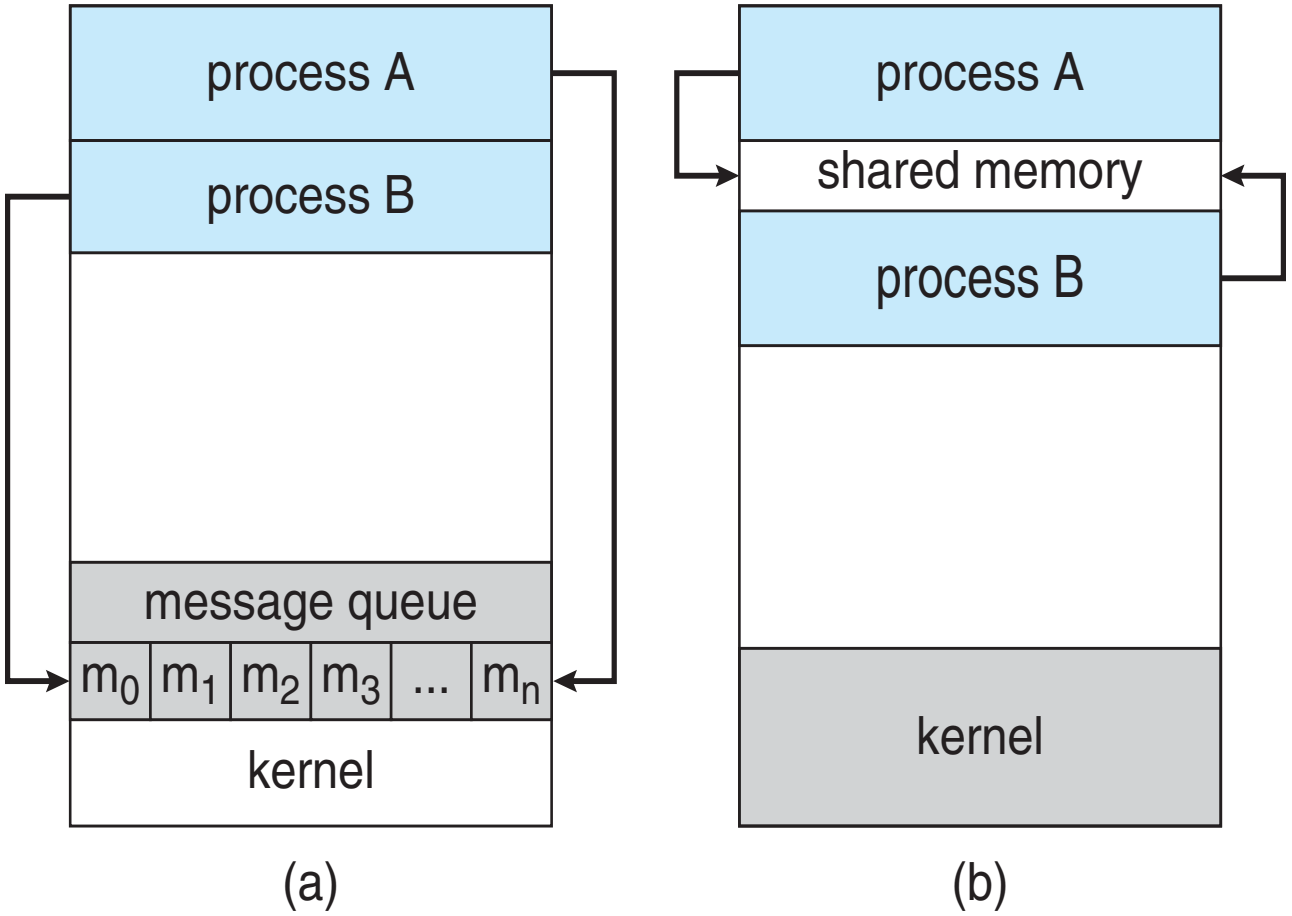
IPC

- Cooperating processes need an **Interprocess communication** (**IPC**) mechanism that will allow them to exchange data.
- Two models for providing IPC:
  - Shared memory
  - Message passing
- Next slide shows both of them in form of a picture.
  - **A: message passing**
    - Communication takes place by message exchanges between processes – send ( ) and recv ( ) system calls.
  - **B: Shared memory**
    - Here, a region of memory that can be shared by cooperating processes is established.
    - Processes can exchange data by reading from or writing to that memory.

Communications Models

(a) Message passing.

(b) shared memory.



Brief Comparison

- **Message passing:**
- Performance of message passing depends on the network bandwidth.
  - Lower bandwidth → messages take longer time.
- Source and destination may be far apart (geographically).
- Each message may require kernel intervention (system call).
- **Shared memory:**
- System calls only to establish shared memory regions.
- Subsequent access plain function calls (in user space).
- Issue of synchronization exists.
  - Read during a write.
  - Write during a write.

Shared Memory

- As stated, this mechanism requires communicating processes to establish a shared region of memory.
- One of the processes can create the shared memory segment and attach it to its address space.
- Other process(es) that wish to communicate using this shared memory must also attach it to their address space.
- Normally, the OS prevents one process from accessing another process's memory (for protection).
- Shared memory relaxes this rule.

LINUX Shared Memory Example

- A bunch of system calls are provided for this.
- Shared memory is implemented using "memory mapped files".
  - Associates a region of shared memory with a file.
- Here is the sequence of events:
  - Create a shared memory object.
  - Set it's size, if needed.
  - Memory map the object.
  - Write to the object or read from it.
  - Remove the shared memory object when finished.
- System call for each stage.

Header Files

- fcntl.h: file control options (e.g. O_CREAT).
- sys/shm.h: shared memory.
- sys/stat.h: file status (e.g. permissions).
- sys/mman.h: memory management.

Shared Memory System Calls

- **System call to create shared memory object.**
- **shm_open(name, O_CREAT | O_RDWR, 0666).**
- Name is the name of the shared memory object (pointer to it).
- O_CREAT: create object, if it does not exist.
- O_RDWR: object is open for reading & writing.
- 0666: directory permissions of the object (read & write allowed).
- If successful, this system call returns an integer file descriptor for the shared memory object.
- **System call to set size of object (new object has 0 length).**
- **ftruncate(shm_fd, SIZE).**
- Used to configure the size of the shared memory object.
- Truncate/shorten a file to a given length.

Shared Memory

- **System call to map the shared memory object to memory of calling process.**
- **mmap(NULL, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);**
- NULL: kernel chooses starting address (in calling process) of mapping.
  - Non 0 value: kernel takes as a hint as to where to place the mapping.
- SIZE specifies the length of the mapping.
- PROT_READ, PROT_WRITE: use the object for reading & writing.
- MAP_SHARED: updates to object are visible to all processes sharing object.
- shm_fd: file descriptor used to access object.
- Last argument for offset is 0: means start mapping at beginning of shared object.
- If successful, mmap ( ) returns a pointer to the mapped area.

Shared Memory

- **System call to remove shared memory segment.**
- shm_unlink(name);
- Removes object, de-allocates memory.
- Once the shared memory object is created, we can do reads & writes using pointers.
- In our example, producer process writes “Studying operating systems is fun!” to the shared memory object.
- Consumer process reads from it.
- Let us now look at the programs.
- Note that we are using sprintf ( ) to write formatted data to a specified string (& not to stdio, that printf ( ) does).
- Data is not being printed, but is stored as a string in the memory pointed to by ptr.

How to execute on Linux machine

- **gcc shm-posix-producer.c –lrt**
- **./a.out**
- **gcc shm-posix-consumer.c –lrt**
- **./a.out**

Another IPC Model: **Pipes**

- A pipe is a data channel that can be used for inter-process communication.
  - This channel is maintained by the kernel.
- Output from one process becomes input of another process.
- Essentially, it's a chunk of main memory that can be accessed from both ends.
- How is a pipe created?
- Using a pipe (int fd [ ]) system call.
- A pipe can be thought of as a file that is shared among multiple processes.
- If the pipe is created successfully, the system call returns two file descriptors:
  - One for the read end of the pipe and other for the write end.
- What is a file descriptor?
- It's a small, positive integer returned by the OS. It can be used instead of the name of the file to interact with it (to read or to write).

Pipes

- As stated, we can use pipe (int fd [ ]) to create a pipe.
- fd [ ] is an array of two elements.
  - Fd [0] is the read-end of the pipe and fd [1] is the write-end.
- Data is accessed on a FIFO basis.
- Since a pipe is a kind of a file, we can use the read ( ) and write ( ) system calls.
- The read system call has the following format:
- int n_read = read (int fd, char *buf, int n);
- Here, fd is the file descriptor, buf is a pointer to a character array where data read is to go , and n is the number of bytes to be read.
- The read system call returns the number of bytes read, and that may be stored in n_read.

Pipes

- Likewise, we have the write system call.
- int n_written = write (int fd, char *buf, int n);
- The names of the variables is self explanatory.
- Another system call used here is – close (fd).
- This essentially “closes” the read or write end of the pipe, depending on the fd value.
- close ( ) closes a file descriptor so it no longer refers to the file, and it may be reused.
- Let’s look at the programming example from the text book.
- Note that both parent & child close their unused ends of the pipe.

Pipes in Practice

- Pipes are used in the command line environment of Linux in which the output of one command serves as the input to the other.
- For example: ls | more.
- As you may recall, the program ls outputs the list of processes. This list may not fit in one command line screen and would then scroll several screens.
- The program more manages output by displaying only one screen of output at a time. User can press the space key to move to the next screen.
- Another example: who | sort.
- The program who lists all current users.
- The program sort sorts them in alphabetical order.
- Can have > 1 pipes:
- who | sort | lpr – what does this do?
- cat list1 list 2 | grep p | sort – what does this do?

End of Chapter 3