Lecture 3

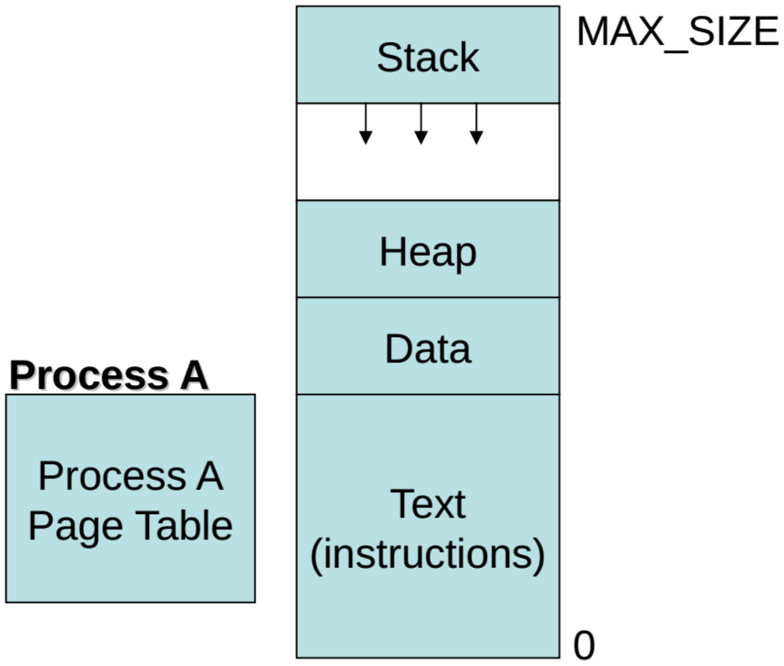
Jan 8, 2024

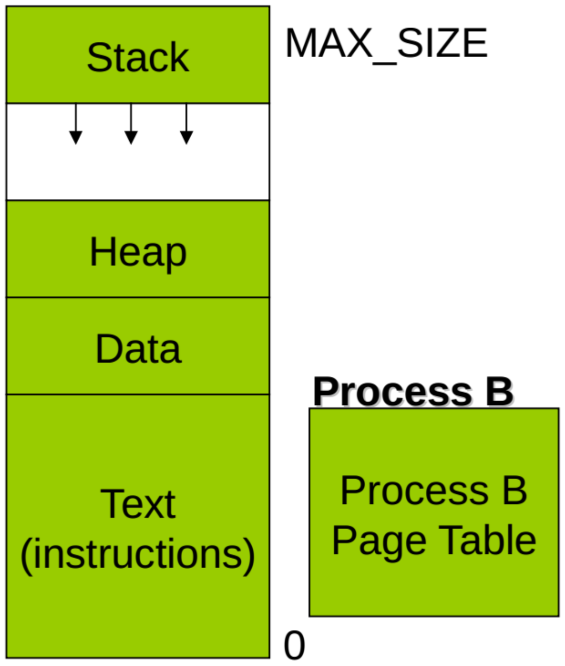
What constitutes a process ?

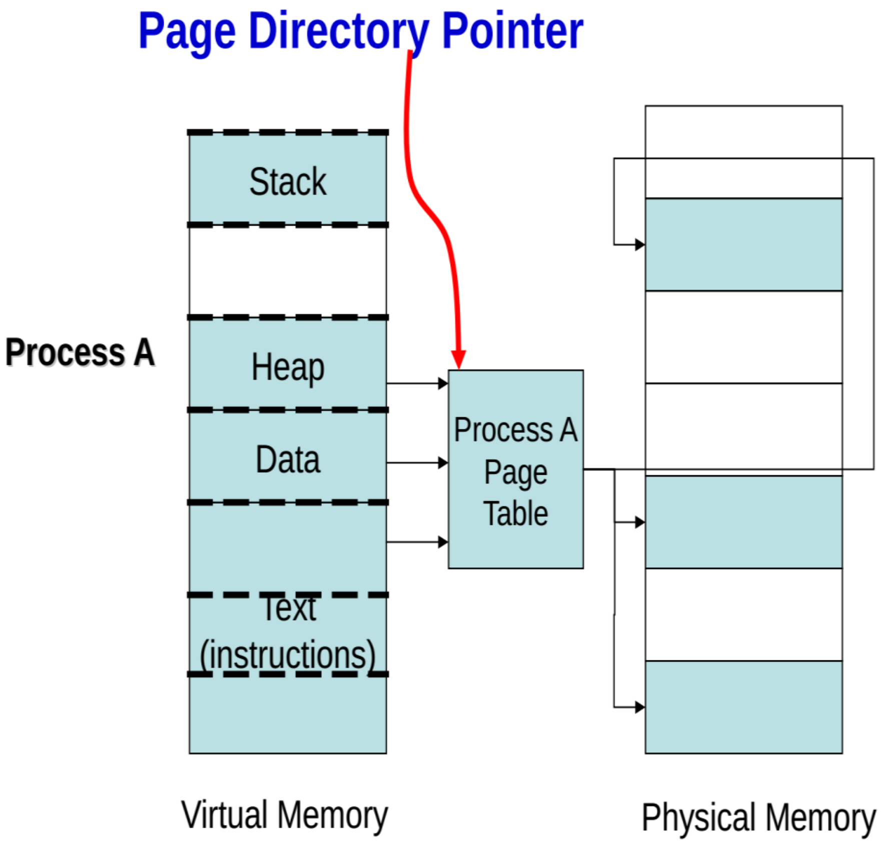
- A unique identifier (PID)
- Memory image
  - Code & data (static)
  - Stack and heap (dynamic)
- CPU context: registers
  - Program counter
  - Current operands
  - Stack pointer
- File descriptors – Pointers to open files and devices

Process {Address Space / Memory Image}

- Each process has a different address space
- This is achieved by the use of virtual memory
- Ie. 0 to MAX_SIZE are virtual memory addresses



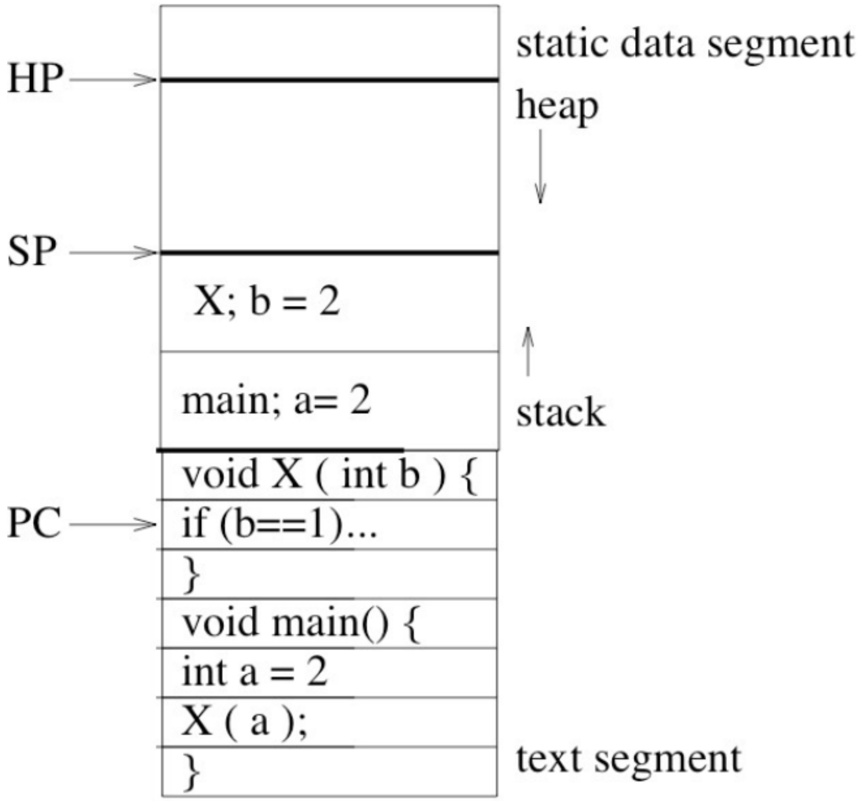




What you wrote:

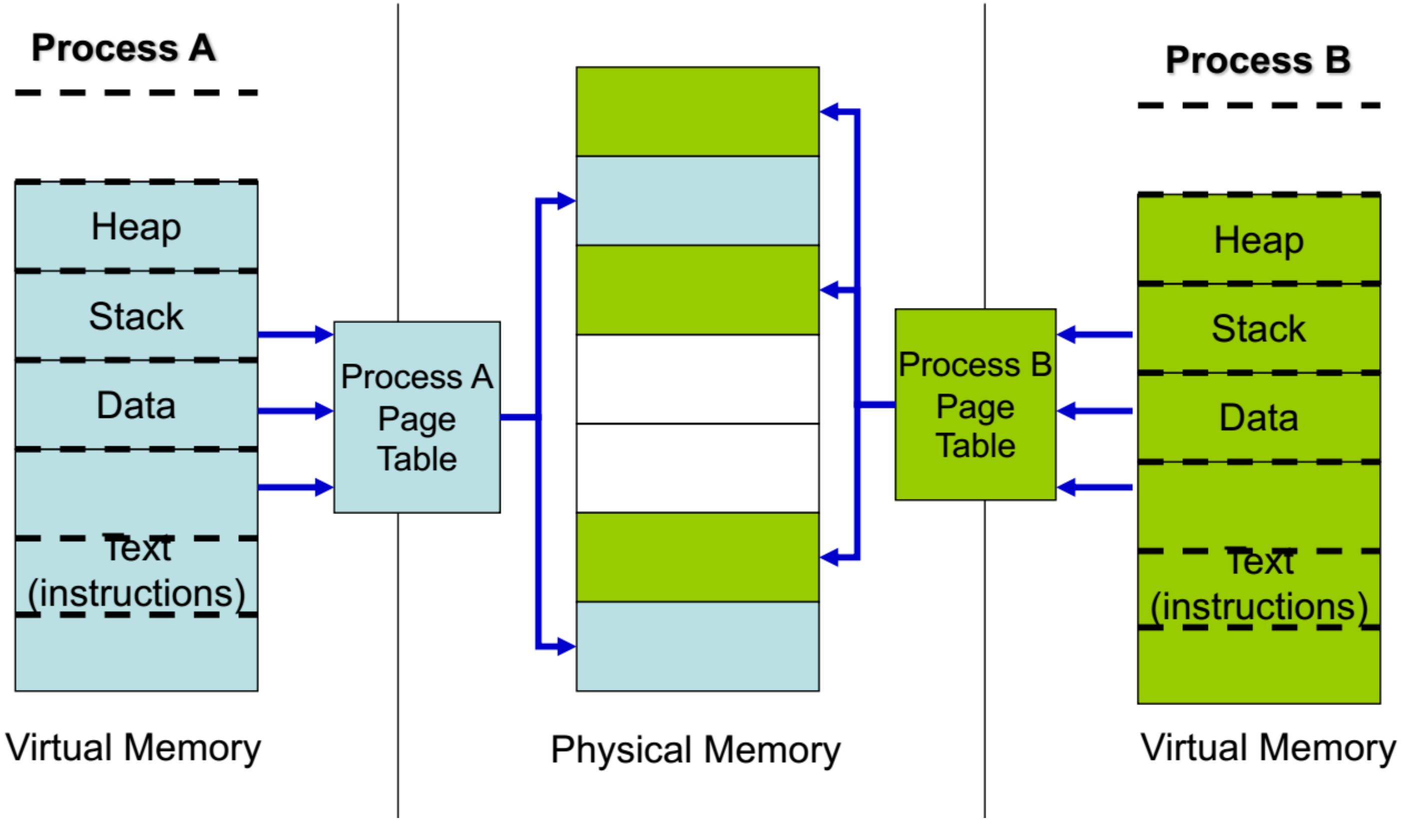
```
void X (int b){
PC ->   if ( b == 1 ) …
   }

main(){
   int a = 2;
   X ( a );
}
```

What's in memory



```
X; b = 2
main; a= 2
void X ( int b ) {
if (b==1)...
}
void main() {
int a = 2
X ( a );
}
```

Virtual Address Mapping



How does OS create a process?

Allocates memory and creates memory image

– Loads code, static data from disk [exe] in to memory [Addr. Space]

– Creates runtime stack [local var, func. paramaters, ret.Addr], heap[malloc]

Opens basic files [Initializes basic I/O]

– STD IN, OUT, ERR [Three open file descriptors] (read from terminal, print output to screen]

Initializes CPU registers [main() - entry point]

– PC points to first instruction

Process States

- When a process is first started/created, it is in new state.
- It needs to wait for the process scheduler (of the OS) to set its status to ”new” and load it into main memory from secondary storage device(such as a hard disk or a CD-ROM).
- Once it is loaded into memory it enters the ready state.
- Once the process has been assigned to a processor by the OS scheduler, a context switch is performed (loading the process into the processor) and the process state is set to running- where the processor executes its instructions.

Process States

If a process needs to wait for a resource (such as waiting for user input, I/O, or waiting for a file to become available), it is moved into the waiting state until it no longer needs to wait - then it is moved back into the ready state.

The ready state means that the process is ready to run, but some other process is already running.

A process may also transition from the running state to the ready state due to a context switch(the OS has scheduled another process even though the current process hasn’t finished).

Once the process finishes execution, or is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

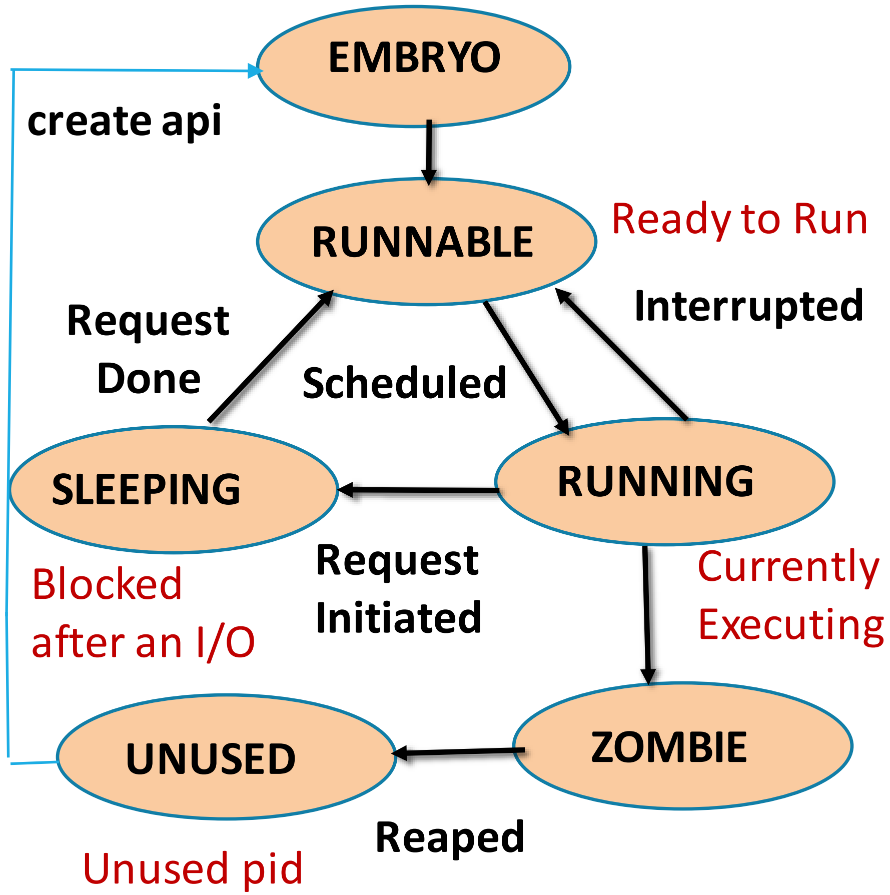
Process States

Running: A process is running on a processor. This means it is 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.

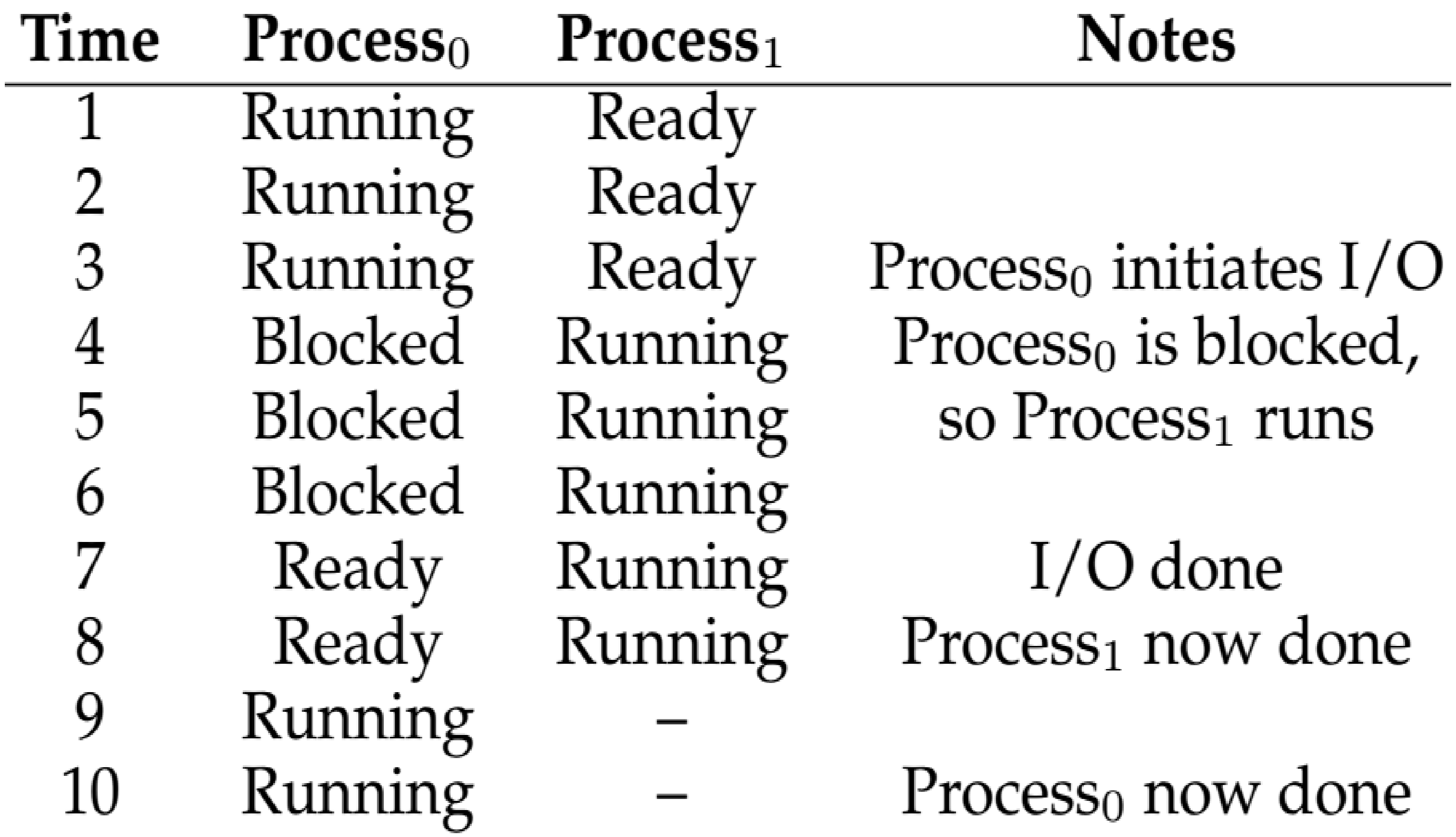
Eg. when a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor. When is it unblocked? Disk issues an interrupt when data is ready



Process States

New: being created, yet to run

Dead: terminated



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

OS data structures

A Process Control Block is a data structure in the OS kernel containing the information needed to manage a particular process.

A PCB is created in the kernel whenever a new process is started.

Each process has a PCB (process control block)

- Holds important process specific information
- Why?
  - Allows process to resume execution after a while
  - Keep track of resources used
  - Track the process state

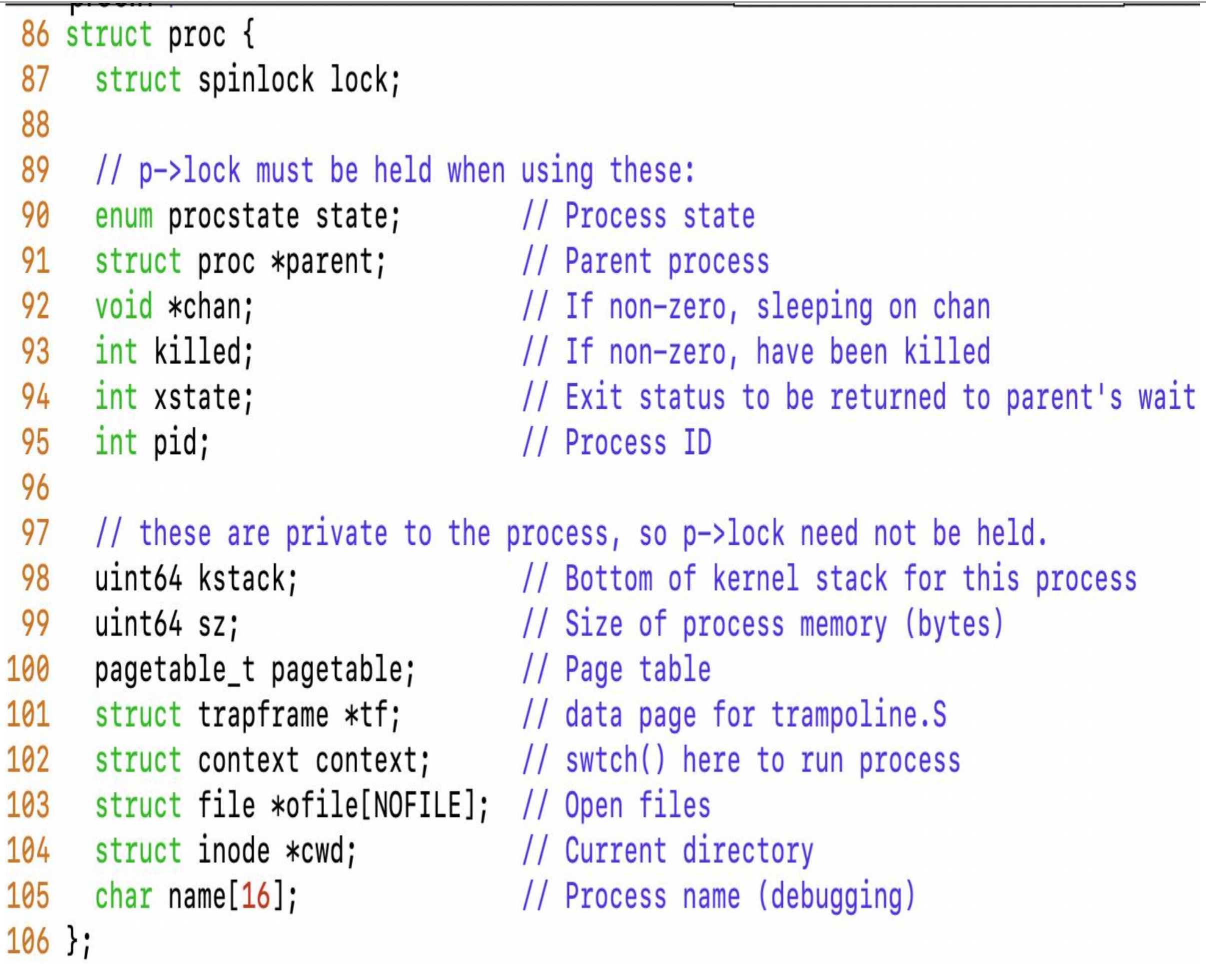
The OS maintains a queue of PCBs, one for each process running in the system.

Process Control Block

- OS maintains a data structure (e.g., list) of all active processes
- Information about each process is stored in a process control block (PCB)
  - the identifier of the process (a process identifier, or PID)
  - Process state
    - register values for the process including the program counter
    - the address space for the process
    - CPU context of the process (saved when the process is suspended)
  - scheduling information such as priority, process accounting information such as when the process was last run, how much CPU time it has accumulated etc
  - Pointers to other related processes (parent)
  - pointer to the next PCB i.e. pointer to the PCB of the next process to run
  - I/O Information (i.e. I/O devices allocated to this process, list of opened files, etc).

– Pointers to memory locations – Pointers to open files

Process Control Block



```c
struct proc {
  struct spinlock lock;

  // p->lock must be held when using these:
  enum procstate state;       // Process state
  struct proc *parent;        // Parent process
  void *chan;                 // If non-zero, sleeping on chan
  int killed;                 // If non-zero, have been killed
  int xstate;                 // Exit status to be returned to parent's wait
  int pid;                    // Process ID

  // these are private to the process, so p->lock need not be held.
  uint64 kstack;              // Bottom of kernel stack for this process
  uint64 sz;                  // Size of process memory (bytes)
  pagetable_t pagetable;      // Page table
  struct trapframe *tf;       // data page for trampoline.S
  struct context context;     // swtch() here to run process
  struct file *ofile[NOFILE]; // Open files
  struct inode *cwd;          // Current directory
  char name[16];              // Process name (debugging)
};
```

Process Control Block

Since the PCB contains the critical information for the process, it must be kept in an area of memory protected from normal user access.

In some operating systems the PCB is placed in the beginning of the kernel stack of the process since that is a convenient protected location.

The PCB is maintained for a process throughout its lifetime, and is deleted once the process terminates.

Process State Queues

The OS maintains a collection of queues that represent the state of all processes in the system

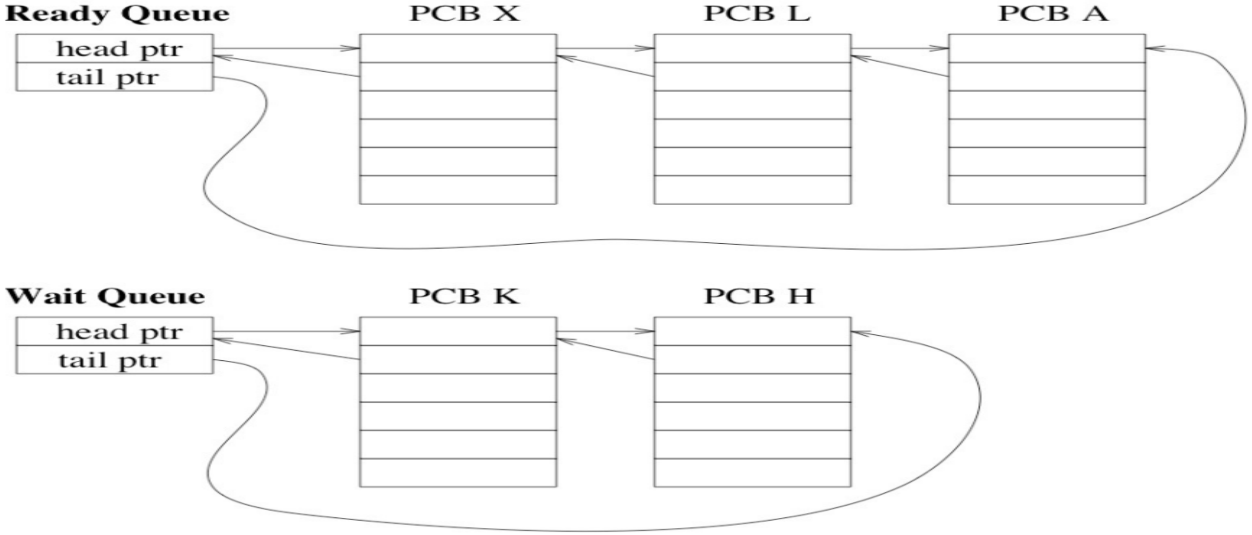
Typically, the OS has one queue for each state Ready, waiting, etc.

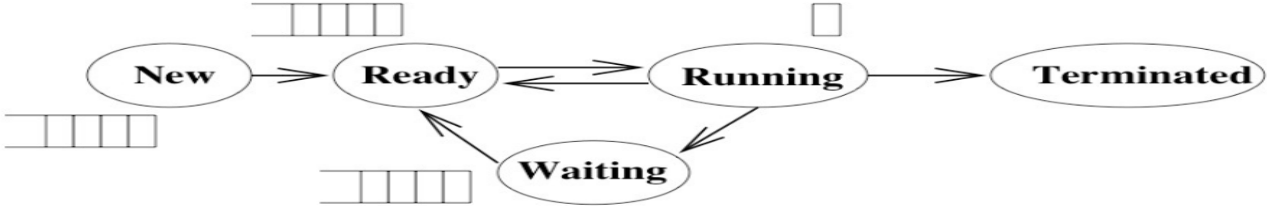
Each PCB is queued on a state queue according to its current state

As a process changes state, its PCB is unlinked from one queue and linked into another

The queue for each state is unbounded in length except for the run queue.

The run queue’s length is always less than or equal to the number of CPUs in the system.





Context Switch

- Starting and stopping processes is called a **context switch**, and is a relatively expensive operation.
- The OS starts executing a ready process by loading hardware registers (PC, SP, etc) from its PCB
- While a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- When the OS stops a process, it saves the current values of the registers, (PC, SP, etc.) into its PCB
- This process of switching the CPU from one process to another (stopping one and starting the next) is the context switch.
  - Time sharing systems may do 100 to 1000 context switches a second.
  - The cost of a context switch and the time between switches are closely related

Process Stacks

Each process has 2 stacks

– **User space stack** Used when executing user code

– **Kernel space stack** Used when executing kernel code (for eg. during system calls)

– Advantage : Kernel can execute even if user stack is corrupted (Attacks that target the stack, such as buffer overflow attack, will not affect the kernel)

What API does the OS provide to user programs?

API = Application Programming Interface= functions available to write user programs

- API provided by OS is a set of “system calls”
  - System call is a function call into OS code that runs at a higher privilege level of the CPU
  - Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
  - Some “blocking” system calls cause the process to be blocked and descheduled(e.g.,read from disk)

Mechanism – Process API

Create

Kill

Suspend

Wait

Status

Process related system calls (in Unix)

- fork() creates a new child process (creates a copy of the current process)
  - All processes are created by forking from a parent
  - The init process is ancestor of all processes
- exec() makes a process execute a given executable.
  - replaces the current process’ code and address space with the code for a different program

- exit()terminates a process
- wait()causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

The first Process

Unix : **/sbin/init**

- Unlike the others, this is created by the kernel during boot
- **Super parent.**
  - Responsible for forking all other processes
  - Typically starts several scripts present in /etc/init.d in Linux

Two special processes

The swapper process: Pid = 0; the swapper is its own parent.

The swapper is a kernel daemon. Swapper moves whole processes between main memory and secondary storage (swapping out and swapping in) as part of the operating system's virtual memory system.

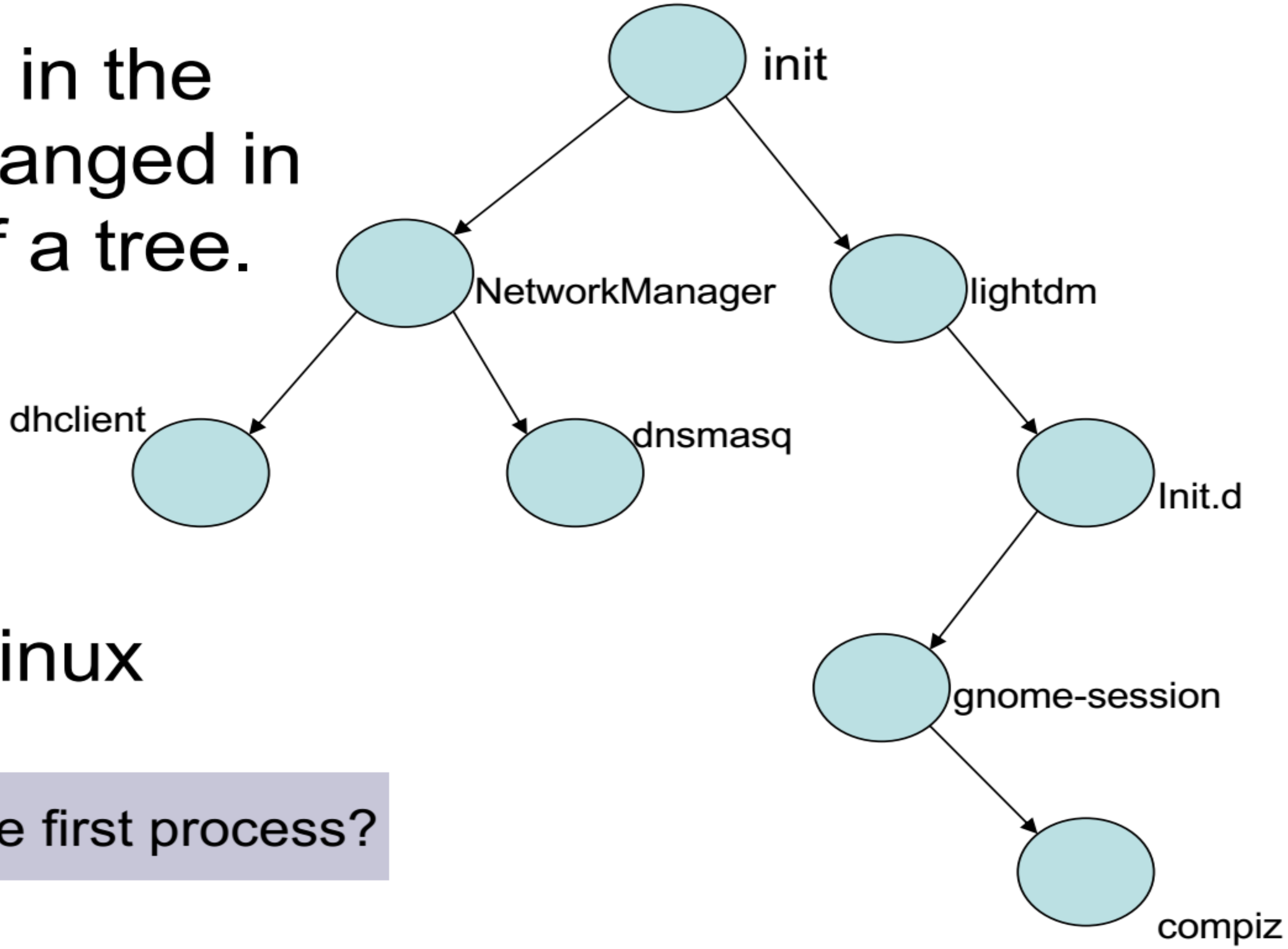
The init process: Pid = 1; its parent is the swapper process.

The first user process (user == root) started by the kernel. **All other processes are children of *init*.** Depending on the boot parameters *init* either:

- spawns a single-user shell at the console
- begins the multi-user start-up scripts (*which are, unfortunately, not standardized across UNIX systems)*.

Process Tree

Processes in the system arranged in the form of a tree.



pstree in Linux

Who creates the first process?