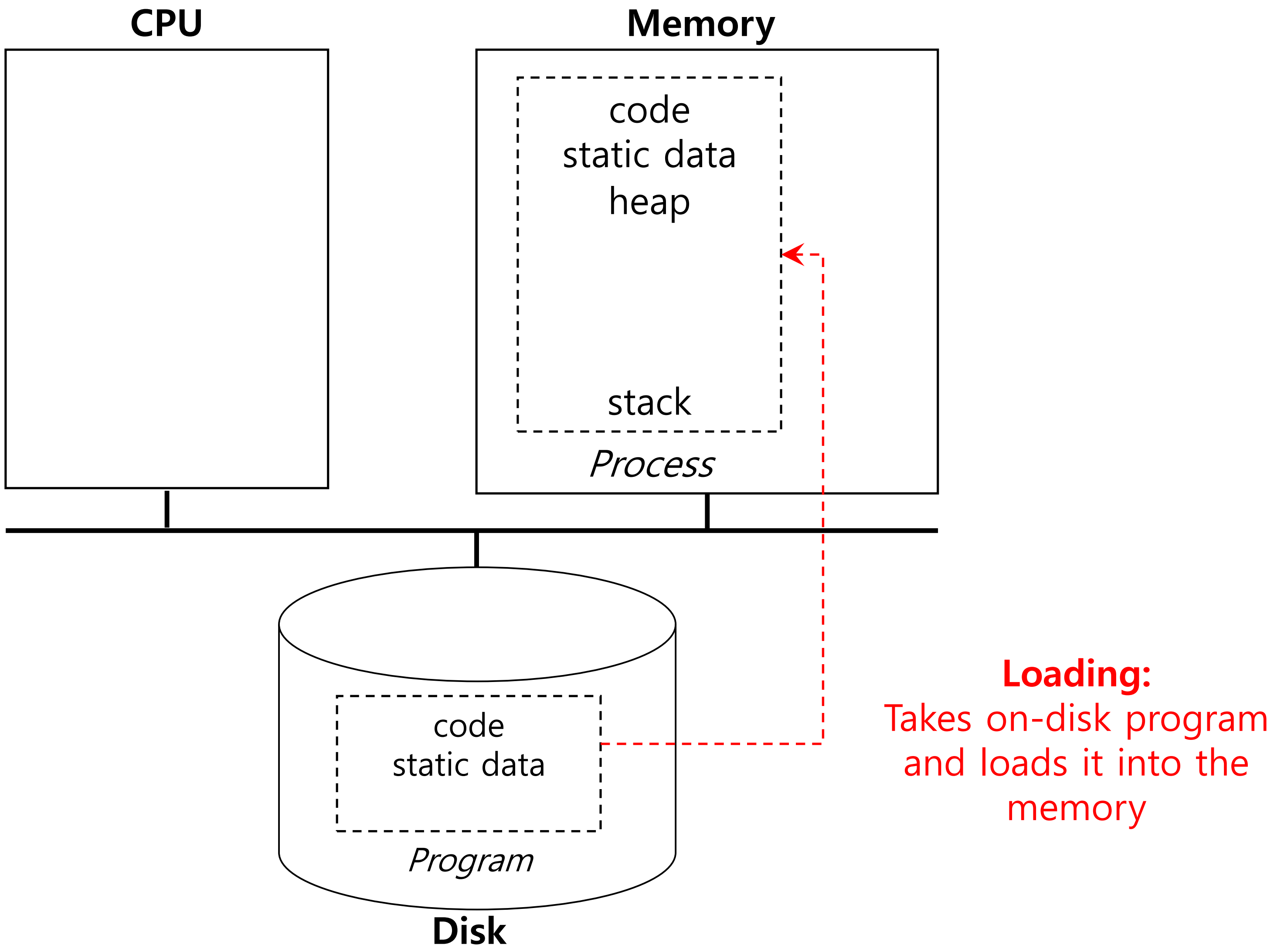
Virtualization: The CPU

Sridhar Alagar

Executing a Program



What is a Process?

- A program in execution
- It is an abstraction
- What constitutes a process?
  - Whatever it can affect
  - Memory: Address space – code, data, heap, stack
  - Registers, IP
  - Open files

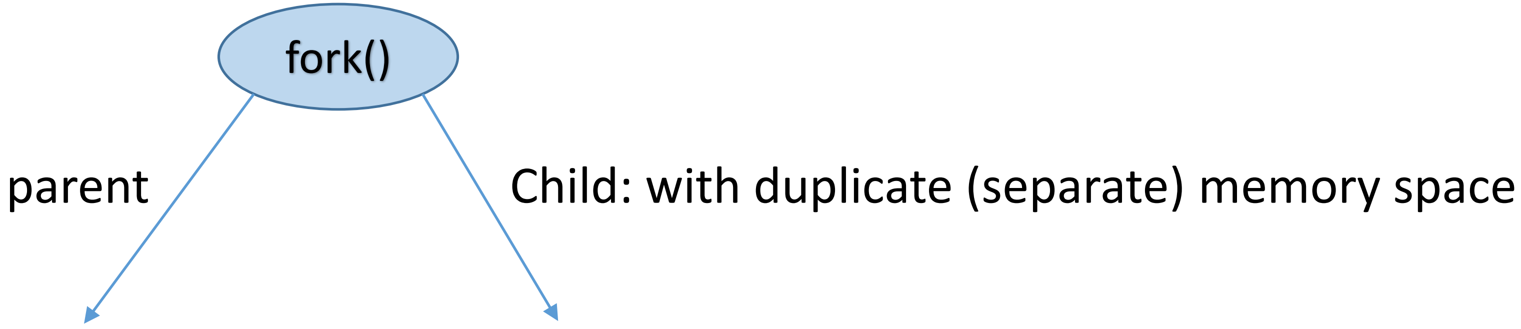
Process API

- Creation
- Destroy
- Wait
  - wait for a process to stop running
- Control
  - suspend and resume
- Status
  - get some status info about the process

Process creation

UNIX examples

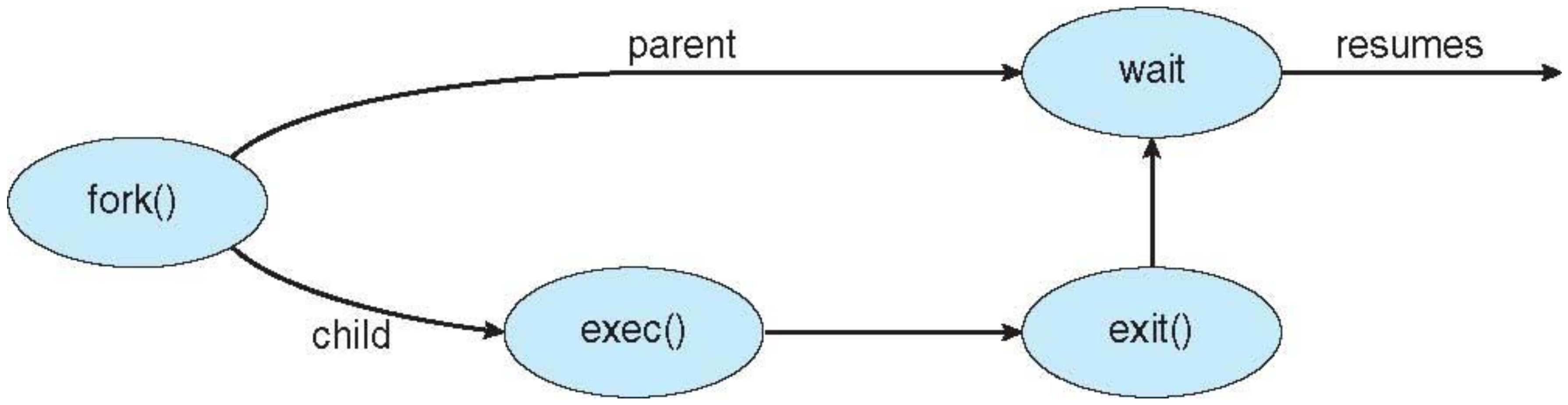
- **`fork()`** system call creates new child process with a duplicate address space
- Parent and child processes have separate memory spaces and execute independently



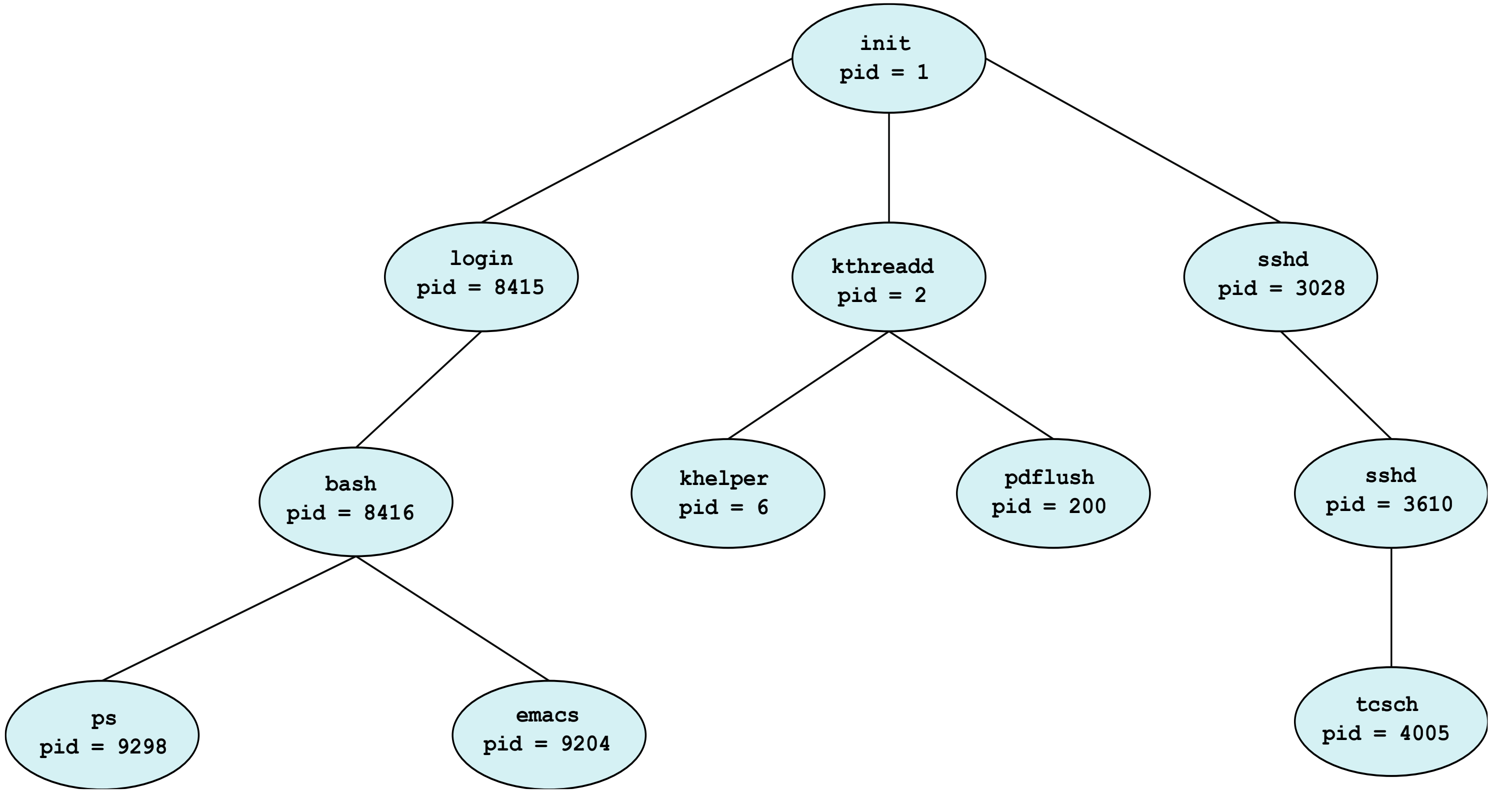
Process creation

UNIX examples

- `exec()` system call used after a `fork()` to replace the process' memory space with a new program



Processes tree



Virtualize the CPU

- CPU needs to be (time) shared by many processes
- Transparent to the process (app)
- Give an illusion that each process own the CPU
  - Each process is allocated a virtual CPU

How to provide good CPU performance?

- Direct Execution
  - Runs directly on CPU
  - Full control of CPU – OS creates process and transfers control to it

- Who should be in control?

Problems with direct execution?

1. Process could do something restricted
   Could read/write other process data (disk or memory)
2. Process could run forever (buggy, or malicious)
   OS needs to be able to switch between processes
3. Process could do something slow (like I/O)
   OS wants to use resources efficiently and switch CPU to other process

Solution:

Limited direct execution – OS and hardware maintain some control

Problem 1: How to restrict process?

- Use privilege levels supported by the hardware
  - Process runs in user level (restricted)
  - OS runs in kernel level (unrestricted)

- How can processes access devices?
  - Need to request OS to do the job through System calls
  - Change privilege level through trap (int)

System Call

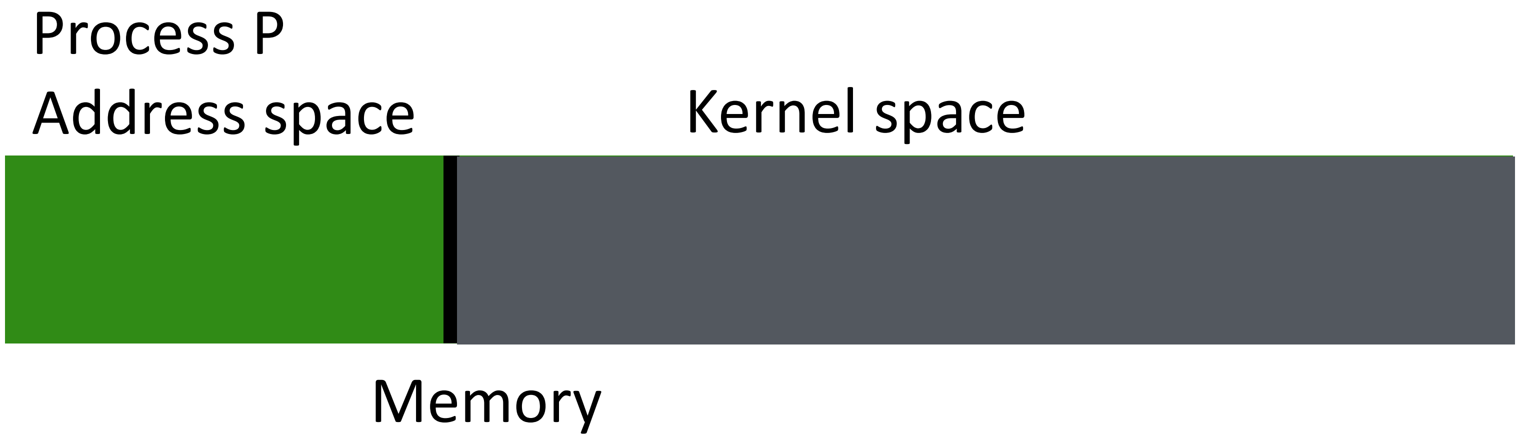
Process P
Address space

Kernel space

Memory

P needs to call sys_read()

System Call

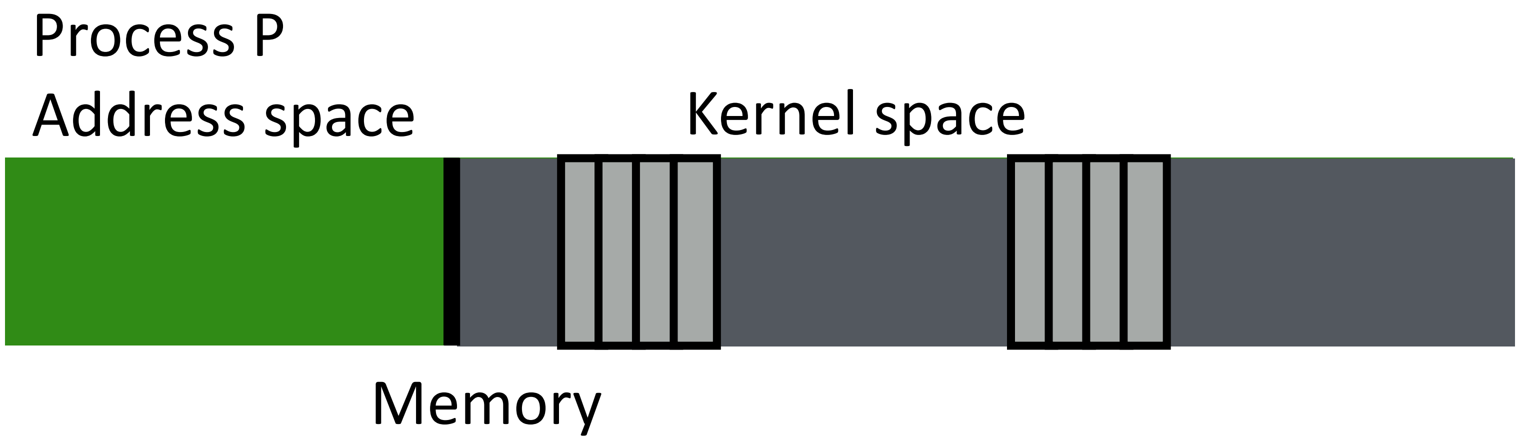


P is in user mode (restricted). It cannot see kernel space or any other space

P wants to call sys_read. But no way to call directly

Need hardware instruction (trap/int) to indirectly call the routine

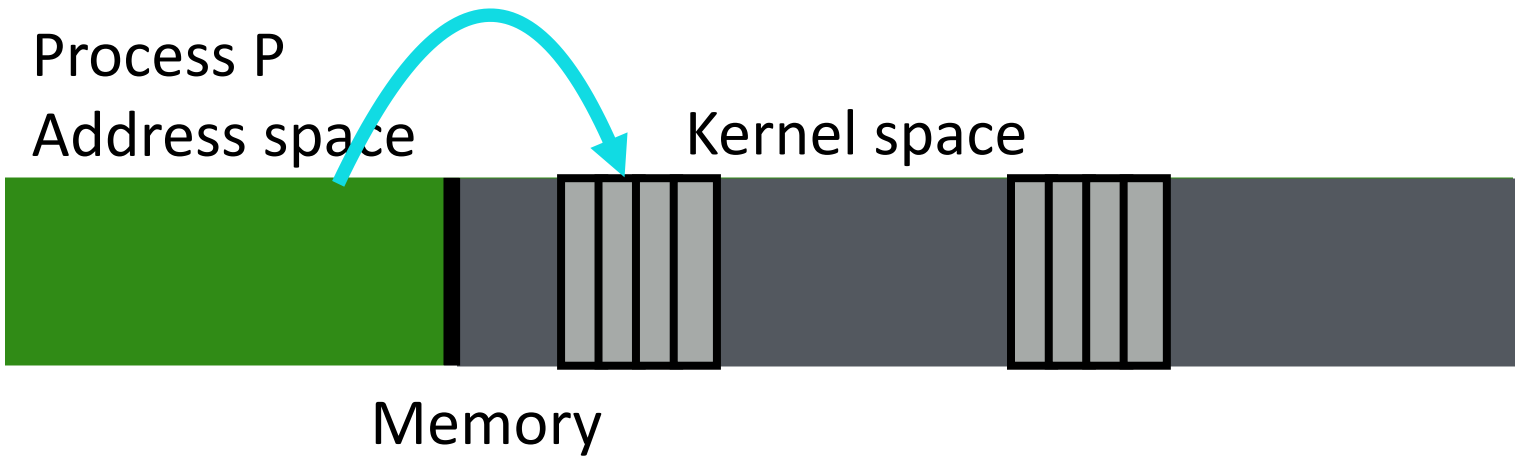
System Call



read():

```
movl $6, %eax;   int $64
```

System Call



read():

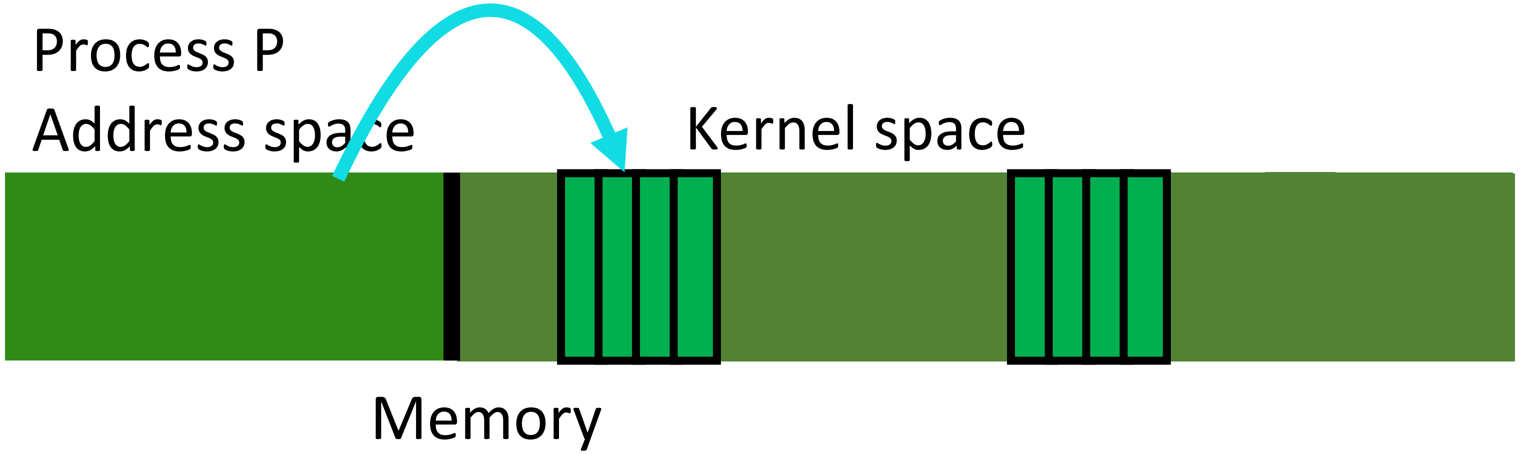
```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

System Call

Kernel mode: all are visible and we can do anything



read():

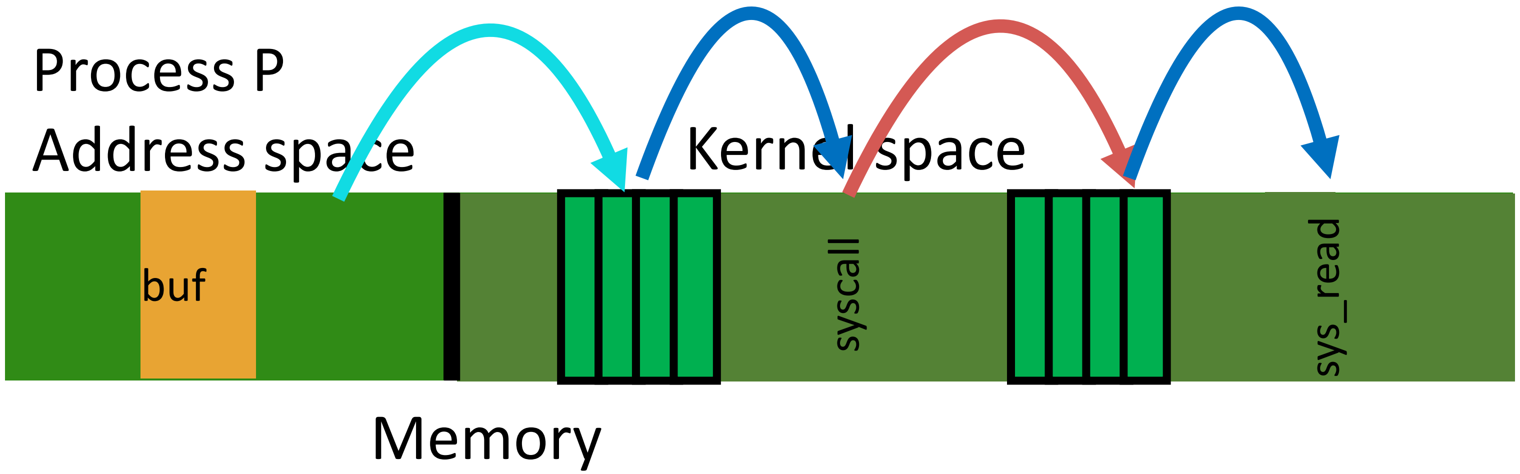
```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

System Call

Kernel can access user memory to fill in user buffer; return-from-trap at end to return to Process P



read():

```
movl $6, %eax;   int $64
```

syscall-table index

trap-table index

Limited direct execution protocol

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system **trap** into OS

Limited direct execution protocol

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		… return from main trap (via `exit()`)
Free memory of process Remove from process list		

What to limit?

- User processes are not allowed to perform
  - General memory access
  - Disk I/O
  - Special x86 instructions like lidt

- What if a process tries to do something restricted?

P2: How to share the CPU?

- Take the CPU away from one process and give it another
  - required for multiprocessing/multitasking
- Design problem
  - Policy: Which process to run next?
  - Mechanism: How to switch between processes?
- Clear separation of policy and design
  - Policy: decided by the decision maker.
    - Scheduler: choose the next process based on the policy
  - Mechanism: low level code to switch between processes
    - Dispatcher

Dispatcher

- OS runs dispatcher
  - Find the next process to execute (use scheduler)
  - Save the **context** of current process
  - Load the context of the next process

- When does dispatcher get control?

- How to switch context?

When does dispatcher gets control?

- Co-operative approach (processes)
  - Whenever process makes a system call, OS gets control. It can run dispatcher.
  - What if a process doesn't need any service from the OS for a long time?
  - It can `yield()` to the kernel
- Problems with yield?
  - Malicious process may run for ever
  - CPU no longer virtualized (sharing is not transparent)

When does dispatcher gets control?

- For true multitasking dispatcher need to be run <u>periodically</u>
  - should not be dependent on processes to yield
- Enter OS by enabling periodic alarm clock
  - Hardware generates timer interrupt (CPU or separate chip)
  - Example: Every 10ms
- Dispatcher counts interrupts between context switches
  - Example: Waiting 20 timer ticks gives 200 ms time slice
  - Common time slices range from 10 ms to 200 ms
- User must not be able to mask timer interrupt

Context Switching

- What to save?
  - Registers, IP, SP, PSR
- Where to save?
  - Process Control Block (PCB)/proc struct

What is stored in PCB (proc)?

- PID
- Process state (I.e., running, ready, or blocked)
- Execution state (all registers, PC, stack ptr)
- Scheduling priority
- Accounting information (parent and child processes)
- Credentials (which resources can be accessed, owner)
- Pointers to other allocated resources (e.g., open files)

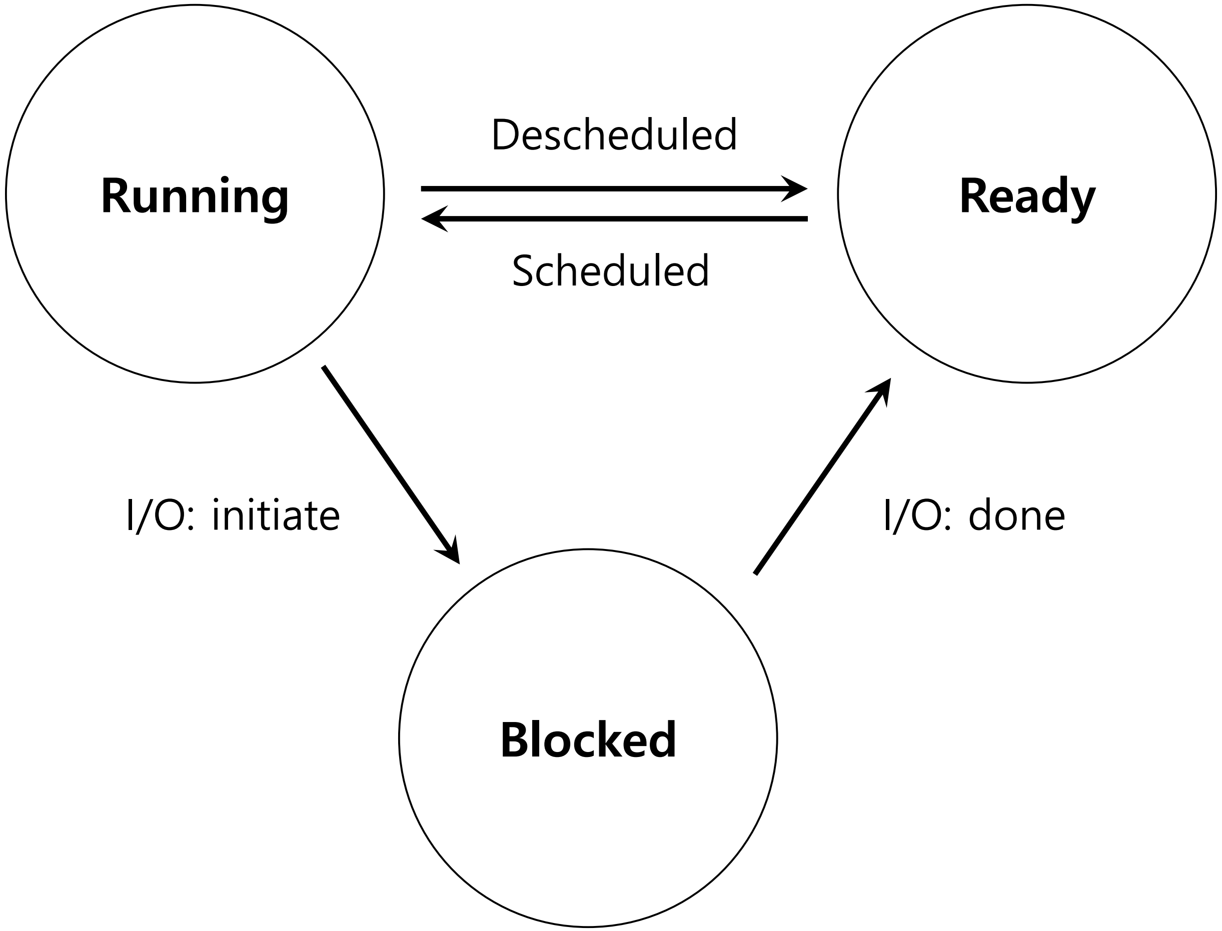
OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A …
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

P3: Process executes slow I/O

- Process issuing i/o operation is blocked till i/o complete
- CPU should be allocated to some other process for better utilization
- Need to track processes state
  - Ready
  - Running
  - Blocked

Process State Transition



Process List

- OS must track every process in system
  - Each process identified by unique Process ID (PID)

- OS maintains queues of all processes
  - Ready queue: Contains all ready processes
  - Event queue: One logical queue per event
    - e.g., disk I/O and locks
    - Contains all processes waiting for that event to complete