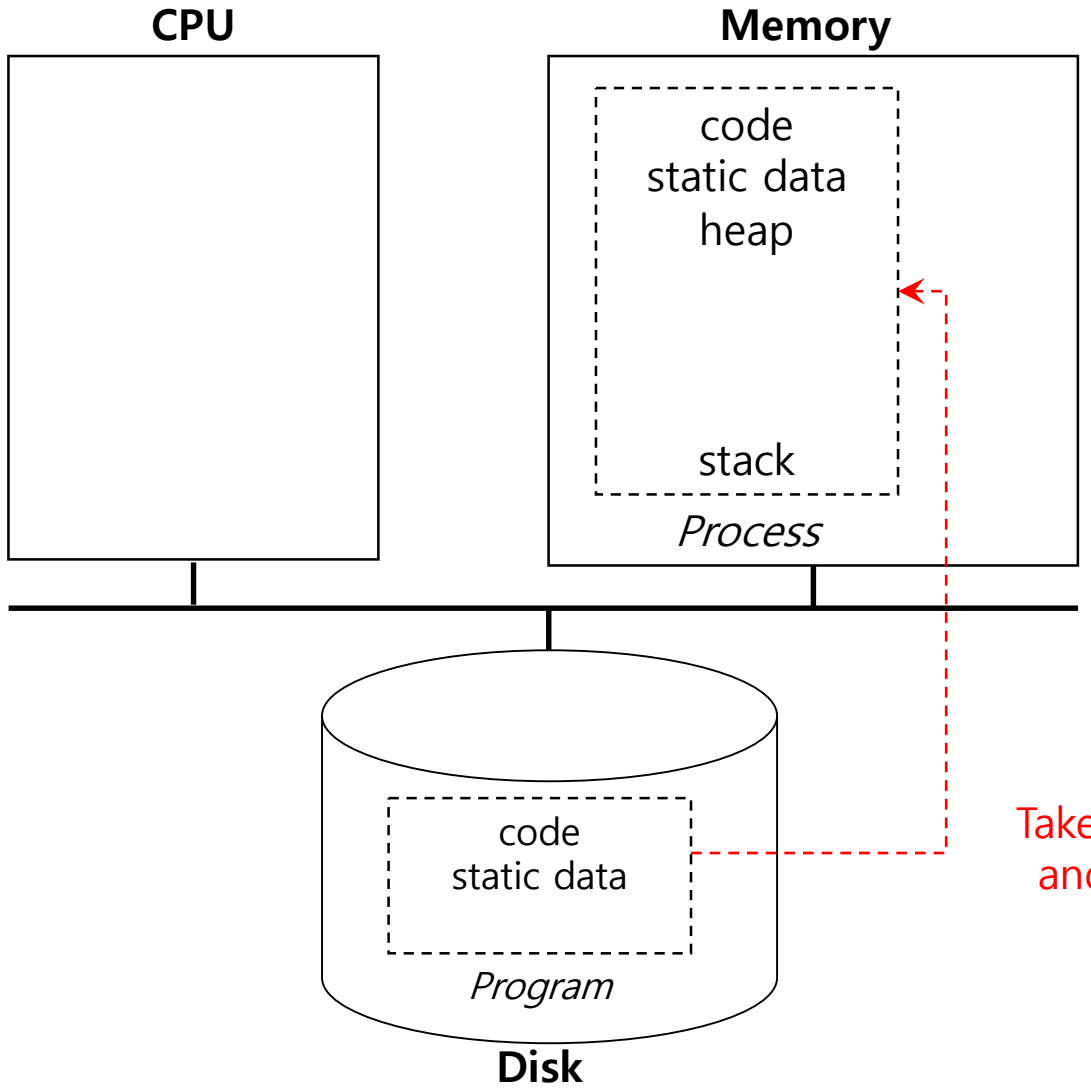
Virtualization: The CPU

Sridhar Alagar

Executing a Program



Loading:
Takes on-disk program and loads it into the memory

What is a Process?

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

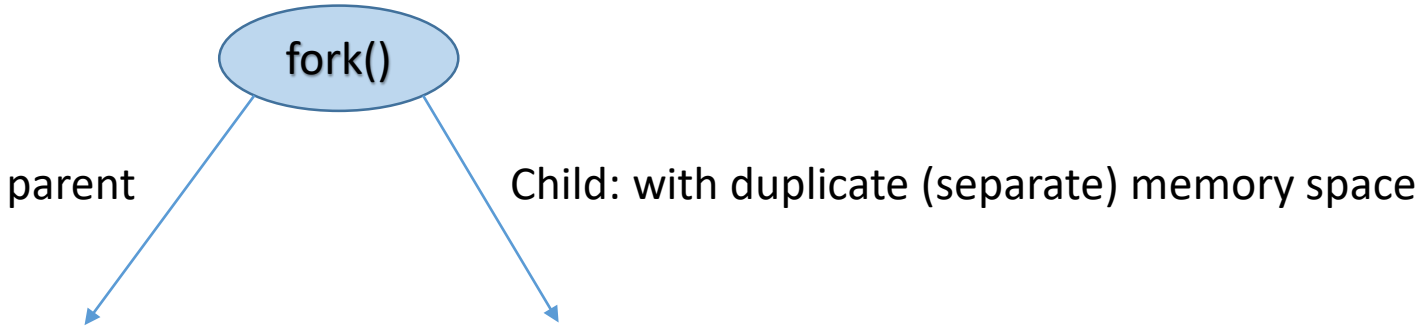
Process API

- 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

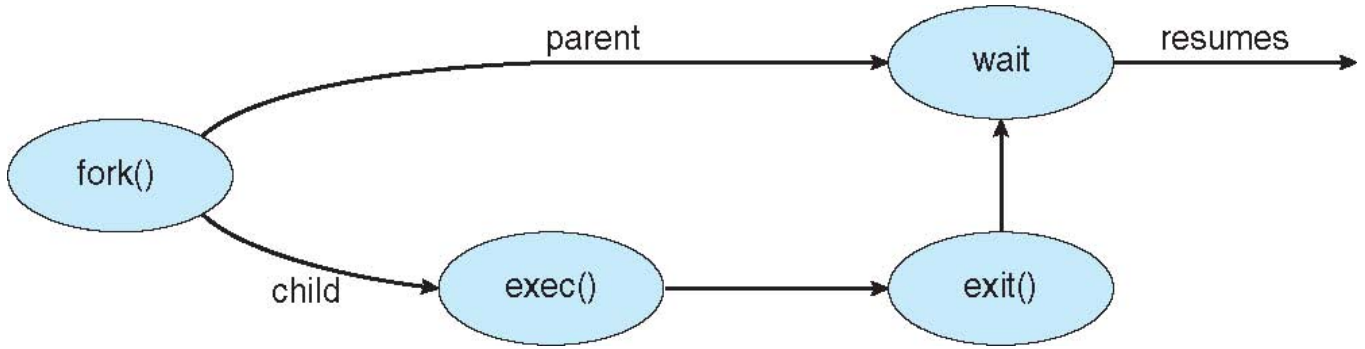
- **`fork()`** system call creates new child process
- Parent and child processes have separate memory spaces and execute independently



Process creation

UNIX examples

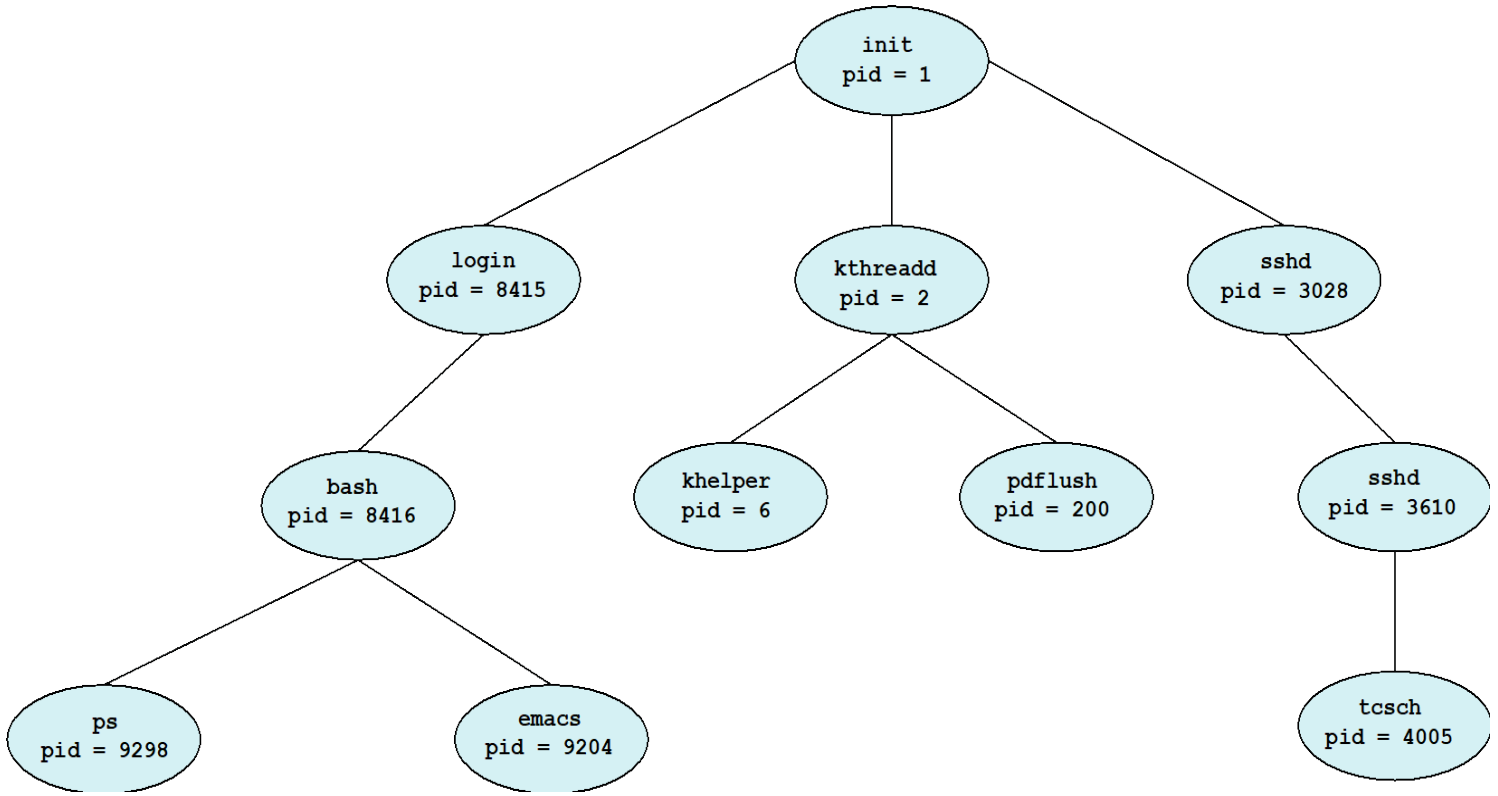
- **`fork()`** system call creates new child process
- **`exec()`** system call used after a **`fork()`** to replace the process' memory space with a new program



Virtualize the CPU

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

Processes tree



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 (slow, 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

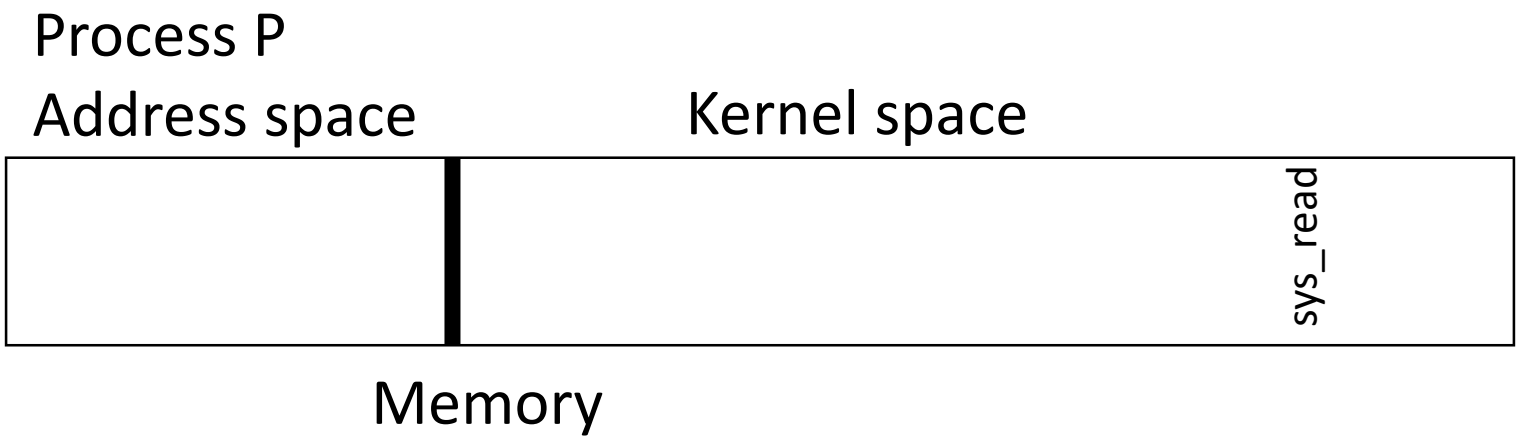
Solution:

Limited direct execution – OS and hardware maintain some control

Problem 1: How to restrict process?

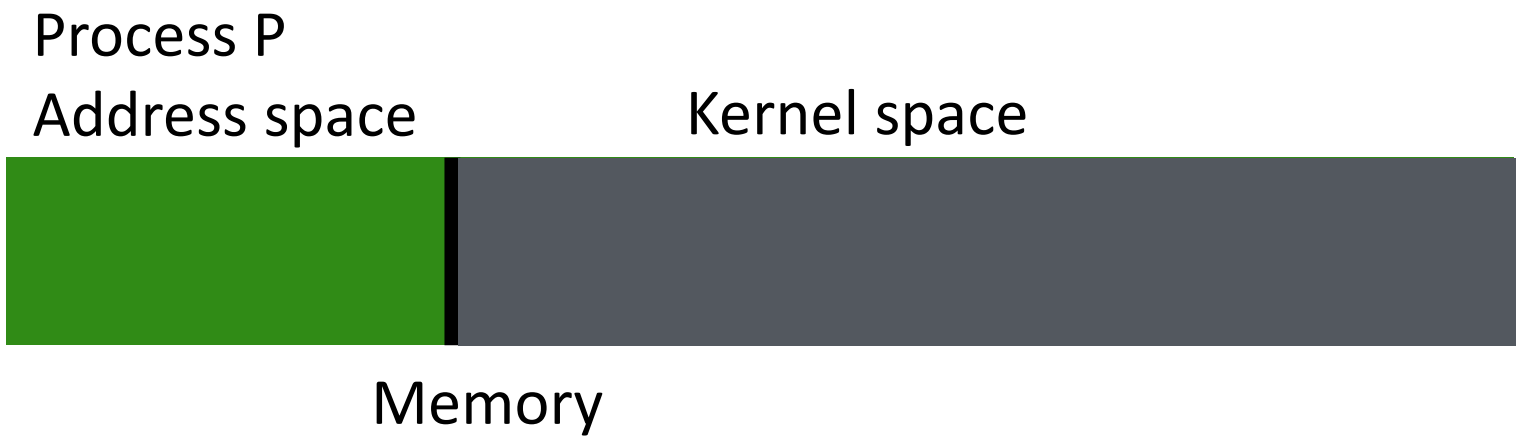
- Use privilege levels supported by the hardware
  - Process runs in user level (restricted)
  - OS runs in kernel level (unrestricted)
- How can processes access devices?
  - Need to request OS to do the job through System calls
  - Change privilege level through trap (int)

System Call



P needs to call sys_read()

System Call

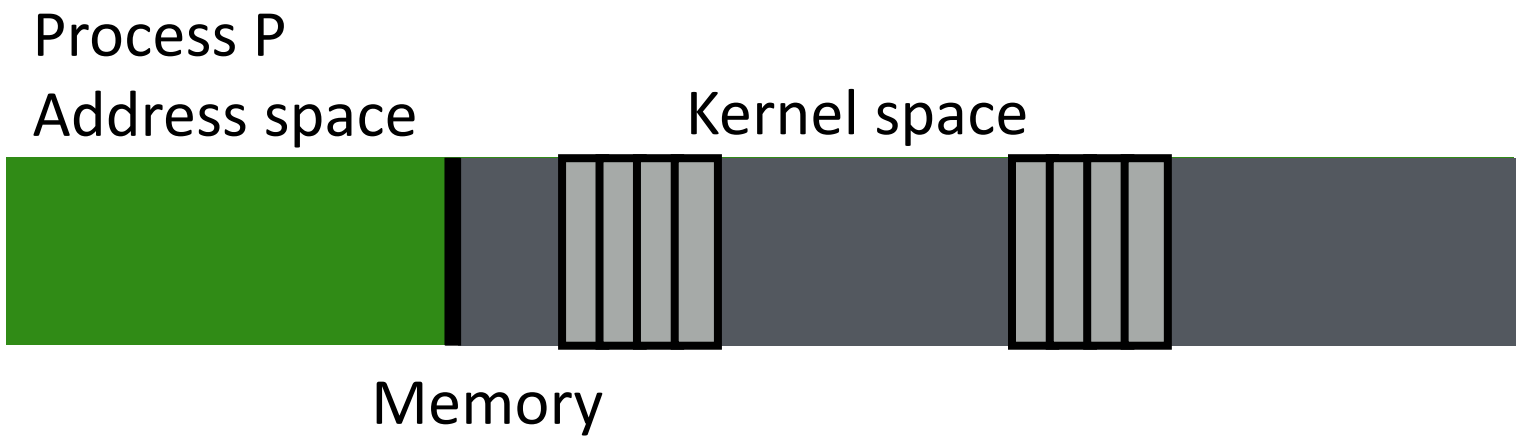


P is in user mode (restricted). It cannot see kernel space or any other space

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

Need hardware instruction (int) to indirectly call the routine

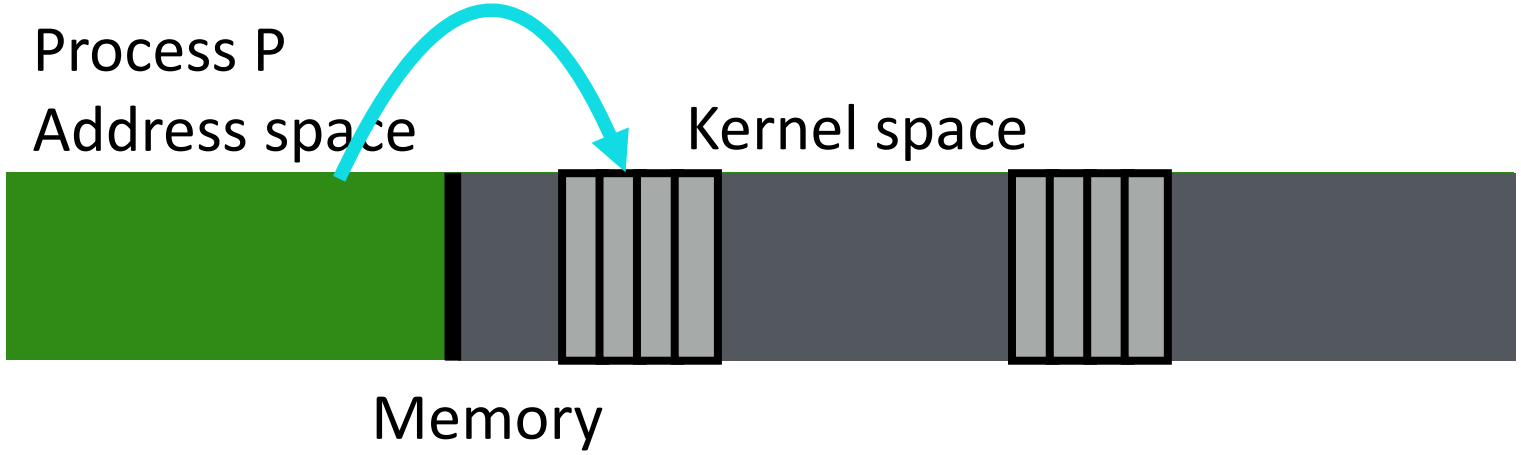
System Call



read():

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

System Call



read():

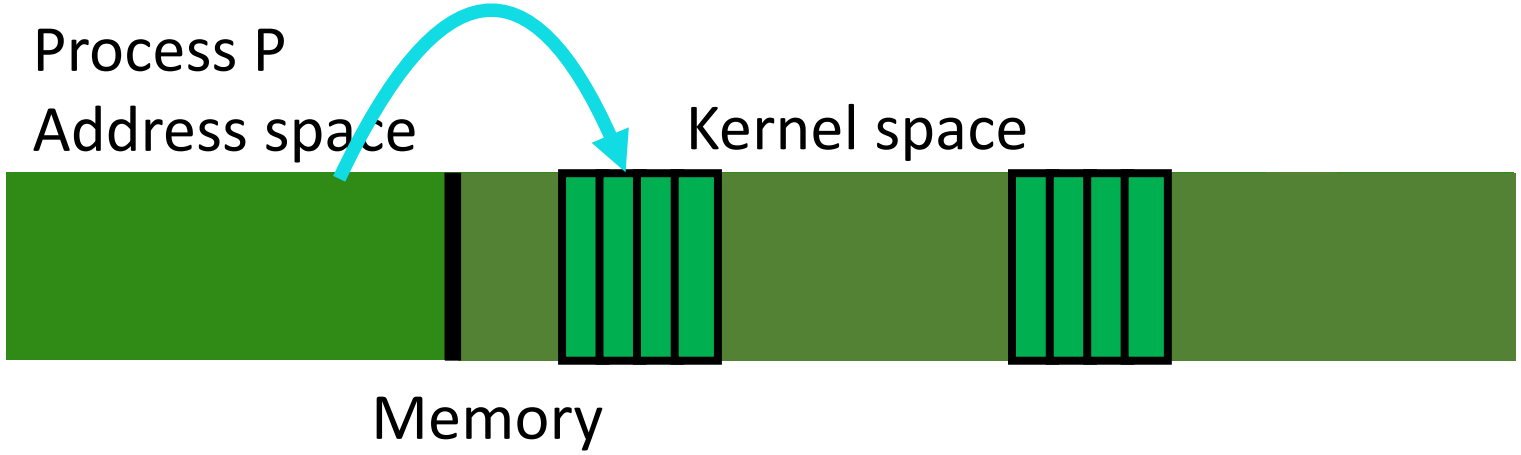
```
movl $6, %eax;    int $64
```

syscall-table index

trap-table index

System Call

Kernel mode: all are visible and we can do anything



read():

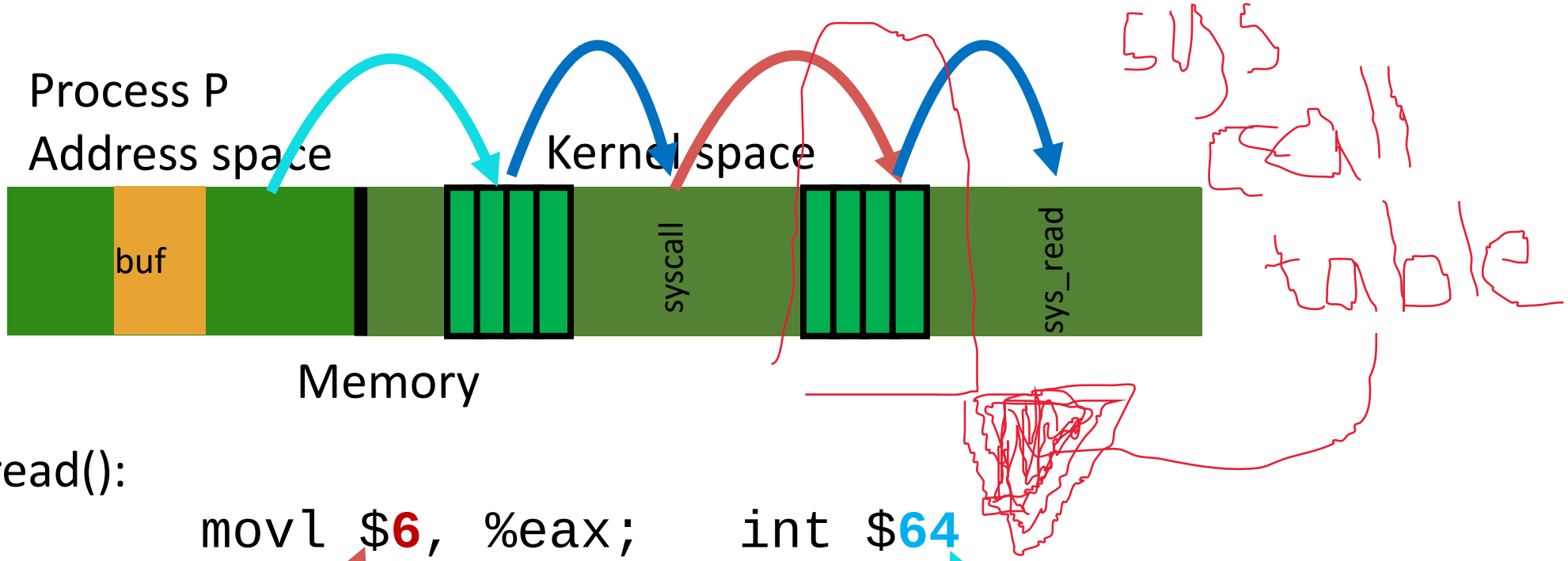
```
movl $6, %eax;    int $64
```

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

- What if process tries to do something restricted?

?

-
  -
-
  -
-
  -