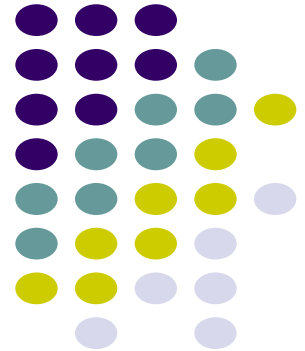


# Processes

ECE 469, Feb 04

Aravind Machiry



# Navigating Lab #2



- **READ COMMENTS IN THE CODE**

# Free Physical Memory (init)

In kern/pmap.c, boot\_alloc

```
static void *
boot_alloc(uint32_t n)
{
    static char *nextfree; // virtual address of next byte of free memory
    char *result;

    // Initialize nextfree if this is the first time.
    // 'end' is a magic symbol automatically generated by the linker,
    // which points to the end of the kernel's bss segment:
    // the first virtual address that the linker did *not* assign
    // to any kernel code or global variables.
    if (!nextfree) {
        extern char end[];
        nextfree = ROUNDUP((char *) end, PGSIZE);
    }
}
```

nextfree will point to the end of the kernel code/data

**nextfree is virtual address.**

**You should allocate n bytes (rounding to PAGE boundary)  
and return the old pointer and update nextfree.**

Physical memory

nextfree end

0x100000

Kernel Code

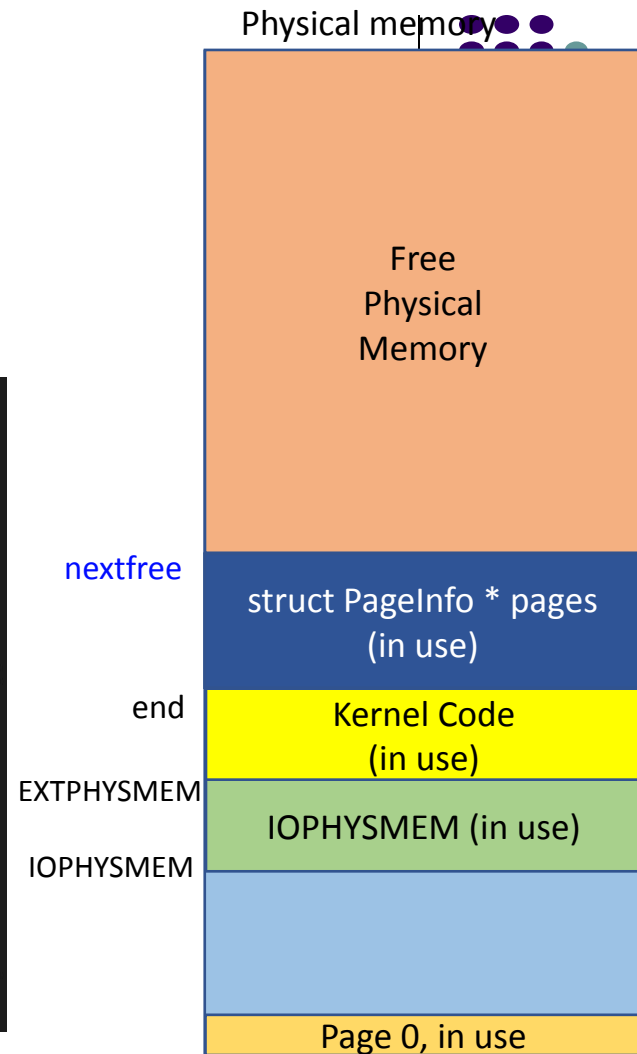


# page\_init: free pages!

- in page\_init()

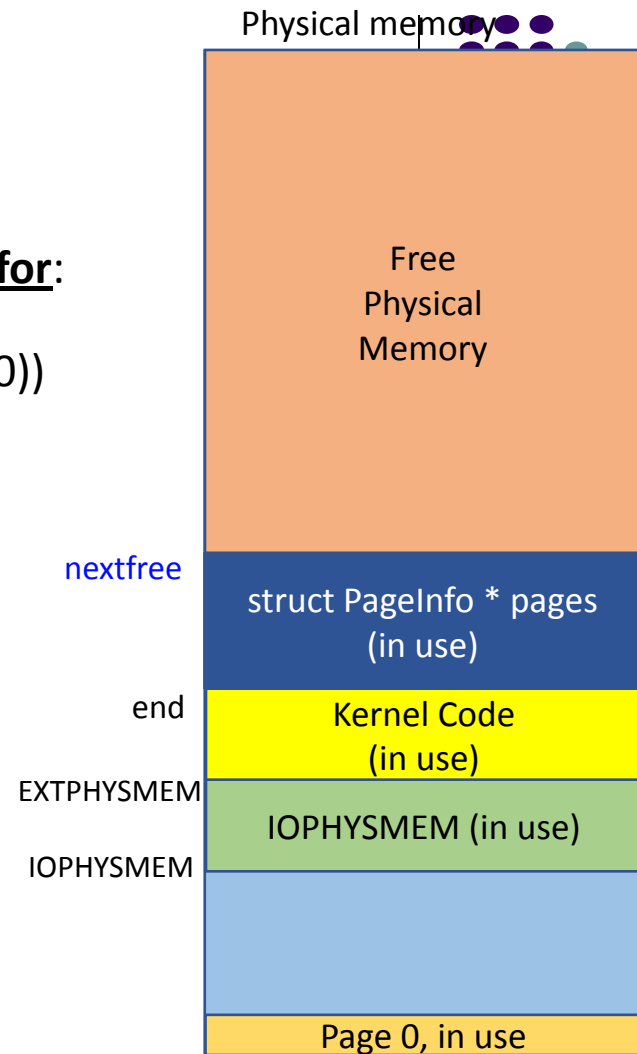
```
// The example code here marks all physical pages as free.
// However this is not truly the case. What memory is free?
// 1) Mark physical page 0 as in use.
// This way we preserve the real-mode IDT and BIOS structures
// in case we ever need them. (Currently we don't, but...)
// 2) The rest of base memory, [PGSIZE, npages_basemem * PGSIZE)
// is free.
// 3) Then comes the IO hole [IOPHYSMEM, EXTPHYSMEM), which must
// never be allocated.
// 4) Then extended memory [EXTPHYSMEM, ...).
// Some of it is in use, some is free. Where is the kernel
// in physical memory? Which pages are already in use for
// page tables and other data structures?
//
// Change the code to reflect this.
// NB: DO NOT actually touch the physical memory corresponding to
// free pages!
```

3



# page\_init: free pages!

- Iterate through the pages and mark all pages **except for**:
  - Page 0
  - Pages from IOPHYSMEM to nextfree (bootalloc(0))
- Add free pages to the list.



# Reuse your code



- **boot\_map\_region**: Inserts mapping of given VA -> PA of given size (page aligned) with the given permission into a page directory.
- **pgdir\_walk**: Gets the page table entry (or creates) corresponding to the given va in the given page directory.
- **page\_lookup**: Look up PageInfo corresponding to the given VA.
- Functions can be written by re-using other functions:
  - **page\_lookup**:
    - i.* Can use `pgdir_walk` to get the `pte`
    - ii.* Can get the physical address of the `pte`
    - iii.* Convert the physical address to `PageInfo` using `pa2page`

# Today's Class

- Users, Programs and Process



# Users, Programs, Processes



- Users have accounts on the system



# Users, Programs, Processes



- Users have accounts on the system
- Users launch programs

# Users, Programs, Processes



- Users have accounts on the system
- Users launch programs
  - Can many users launch the same program?
  - Can one user launch many instances of the same program?

# Users, Programs, Processes



- Users have accounts on the system
- Users launch programs
  - Can many users launch the same program?
  - Can one user launch many instances of the same program?

A process is an “instance” of a program

# Program vs. Process



```
main ()
{
  ...
  foo ()
  ...
}

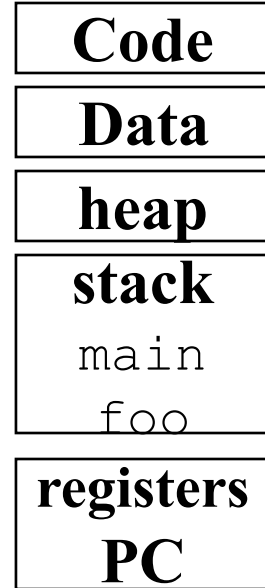
foo ()
{
  ...
}
```

**Program**

```
main ()
{
  ...
  foo ()
  ...
}

foo ()
{
  ...
}
```

**Process**



# So What Is A Process?



- It is a running instance of a program.
- Program becomes **alive** through a process.
- Any relation between multiple instances of a program?

# So What Is A Process?



- It is a running instance of a program.
- Program becomes **alive** through a process.
- Any relation between multiple instances of a program?
  - Ideally, No!
  - How is the separation maintained?

# Process needs to communicate with OS



- Access system resources :
  - Network, Memory, Bluetooth, etc.
  - Maintained by OS.
- How does the communication happen between a process and OS?

# Quick Detour: System Calls



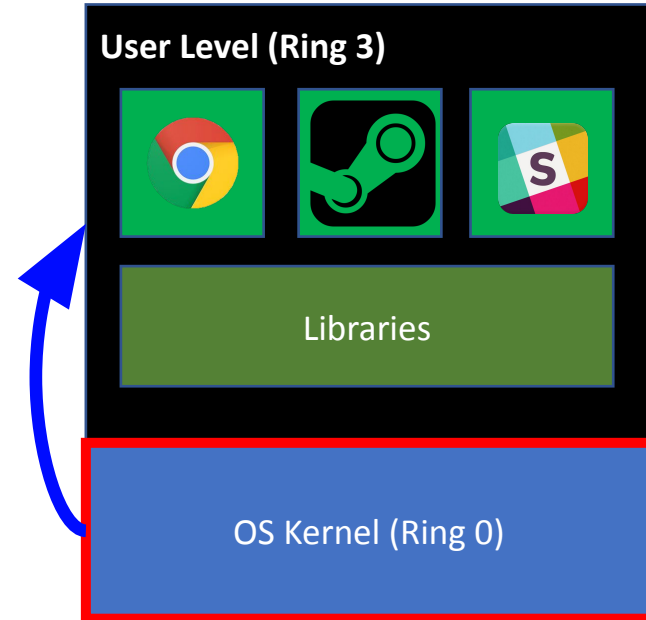
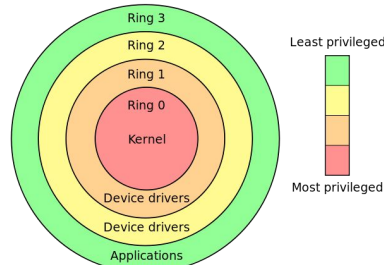
- Interface between a process and the operating system kernel
  - Kernel manages shared resources & exports interface
  - Process requests for access to shared resources
- Generally available as assembly-language instructions:
  - `syscall`



# Why can't a process directly execute kernel code? Because Kernel code ...



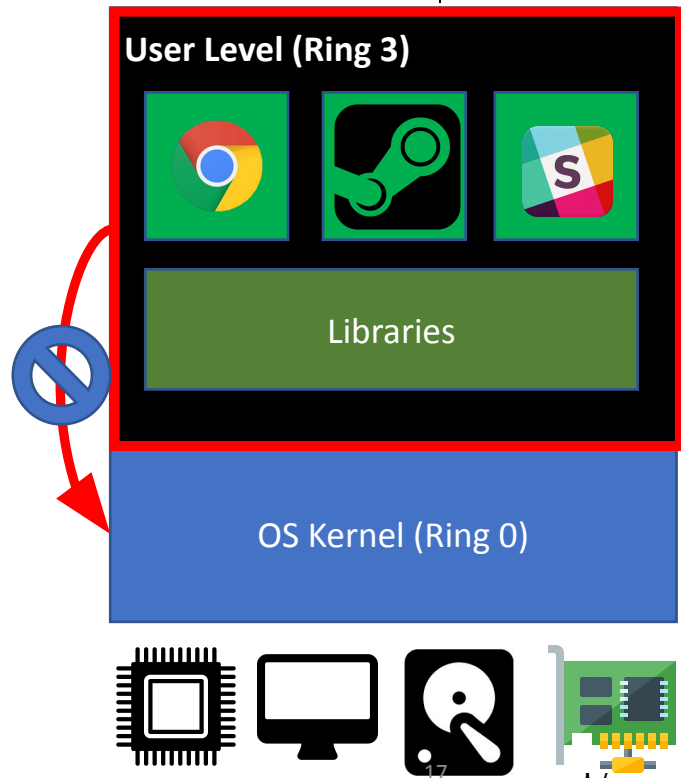
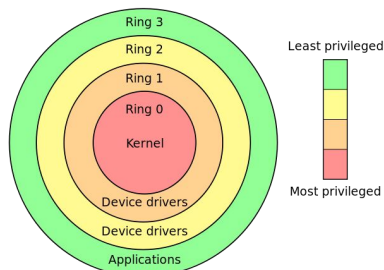
- Runs with the highest privilege level (Ring 0)
- Configures system (devices, memory, etc.)
- Manages hardware resources
  - Disk, memory, network, video, keyboard, etc.
- Manages other jobs
  - Processes and threads
- Serves as trusted computing base (TCB)
  - Set privilege
  - Restrict other jobs from doing something bad..



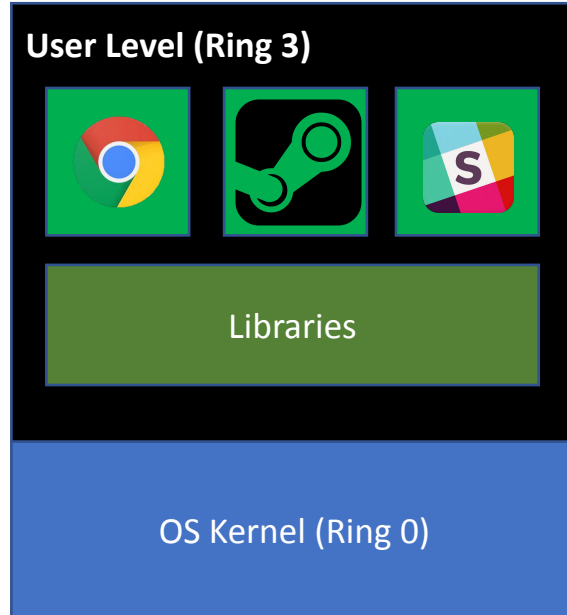


# User mode process runs at Ring 3

- Runs with a restricted privilege (Ring 3)
  - The privilege level for running an application...
- Most of regular applications runs in this level
- Cannot access kernel memory
  - Can only access pages set with PTE\_U
- Cannot talk directly to hardware devices
  - Kernel must mediate the access



# Syscall: User/Kernel communication



```
int main() {  
    printf("ECE469");  
}
```

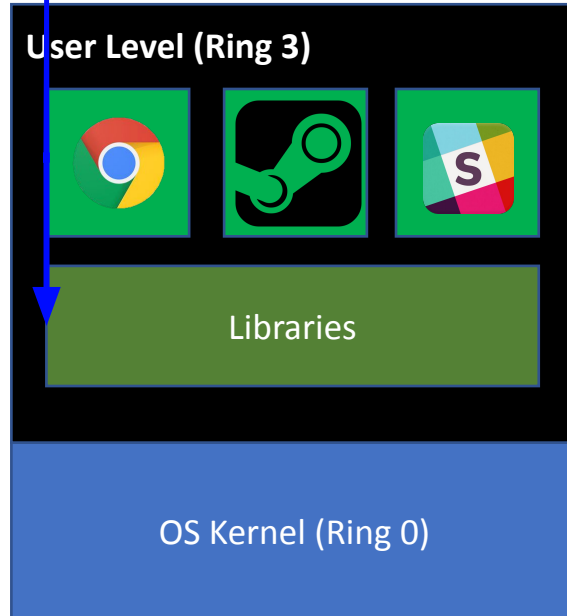


# Syscall: User/Kernel communication



`printf("ECE469")`

A library call in ring 3



```
int main() {  
    printf("ECE469");  
}
```



# Syscall: User/Kernel communication

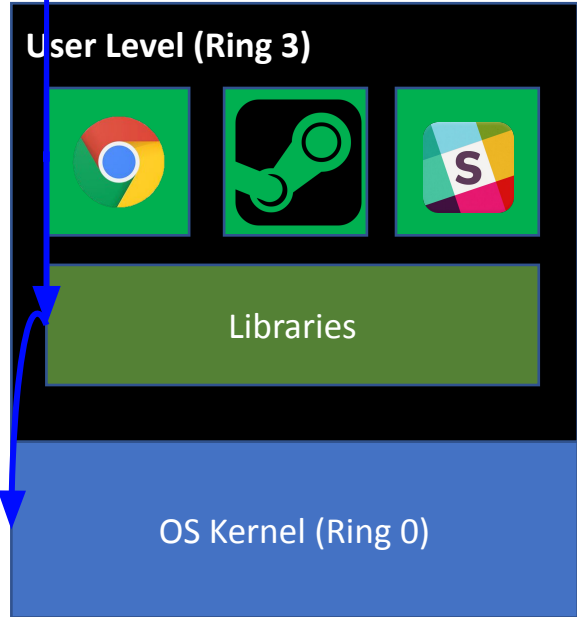


`printf("ECE469")`

A library call in ring 3

`sys_write(1, "ECE469", 6);`

A system call, **From ring 3**



```
int main() {  
    printf("ECE469");  
}
```



# Syscall: User/Kernel communication

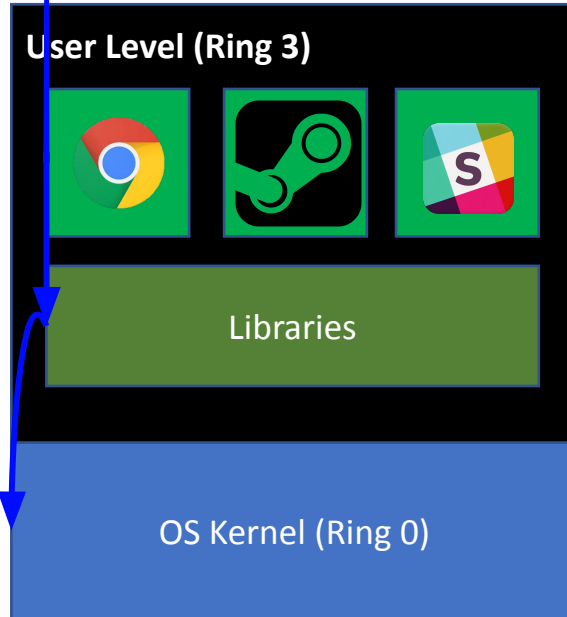


`printf("ECE469")`

A library call in ring 3

`sys_write(1, "ECE469", 6);`

A system call, **From ring 3**



```
int main() {  
    printf("ECE469");  
}
```



# Syscall: User/Kernel communication



`printf("ECE469")`

A library call in ring 3

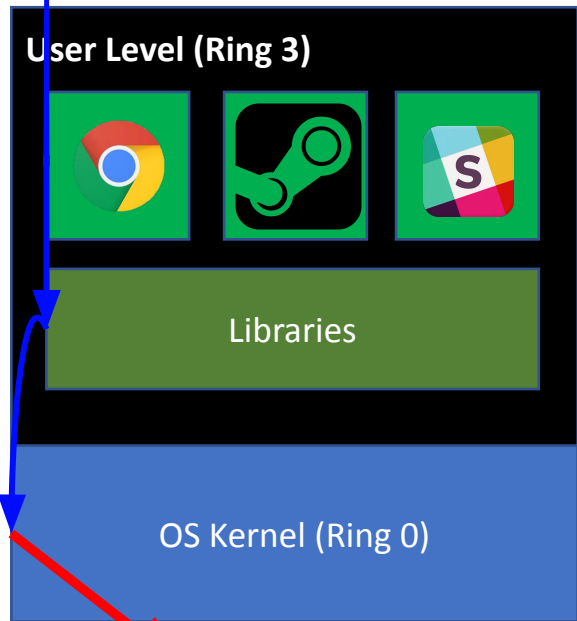
`sys_write(1, "ECE469", 6);`

A system call, **From ring 3**

**Interrupt!, switch from ring3 to ring0**

A kernel function

`do_sys_write(1, "ECE469", 6)`



```
int main() {  
    printf("ECE469");  
}
```

# Syscall: User/Kernel communication

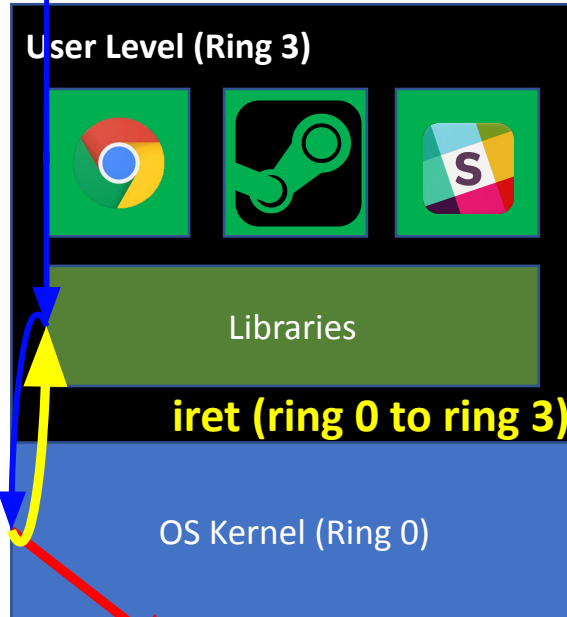


`printf("ECE469")`

A library call in ring 3

`sys_write(1, "ECE469", 6);`

A system call, **From ring 3**



```
int main() {  
    printf("ECE469");  
}
```

**Interrupt!, switch from ring3 to ring0**

A kernel function

`do_sys_write(1, "ECE469", 6)`





# Syscall: User/Kernel communication



`printf("ECE469")`

A library call in ring 3

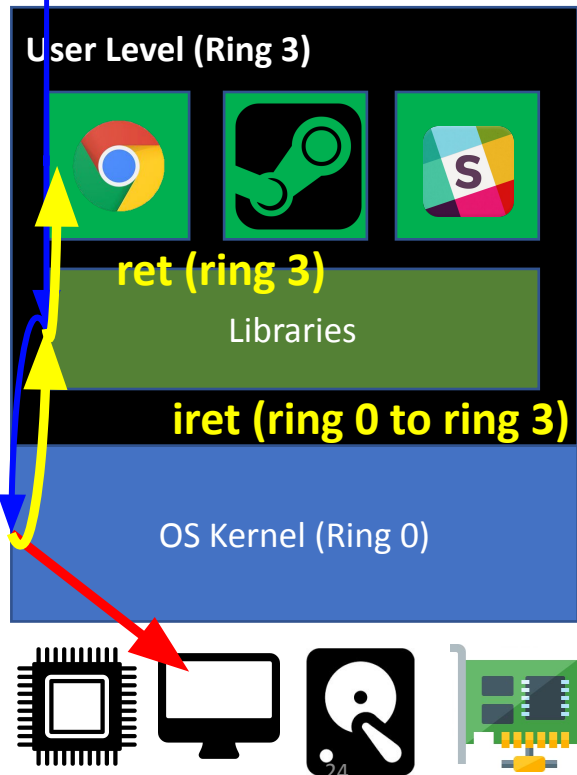
`sys_write(1, "ECE469", 6);`

A system call, From ring 3

Interrupt!, switch from ring3 to ring0

A kernel function

`do_sys_write(1, "ECE469", 6)`



```
int main() {  
    printf("ECE469");  
}
```

# Let's get back: Process

- Who has the ability to create a process?
  - OS
- Who wants to create a process?
- How can we create a process?



# Let's get back: Process

- Who has the ability to create a process?
  - OS
- Who wants to create a process?
  - User/Program
- How can we create a process?



# Let's get back: Process



- Who has the ability to create a process?
  - OS
- Who wants to create a process?
  - User/Program
- How can we create a process?
  - System call
    - But, to do a syscall, we need a process!!!

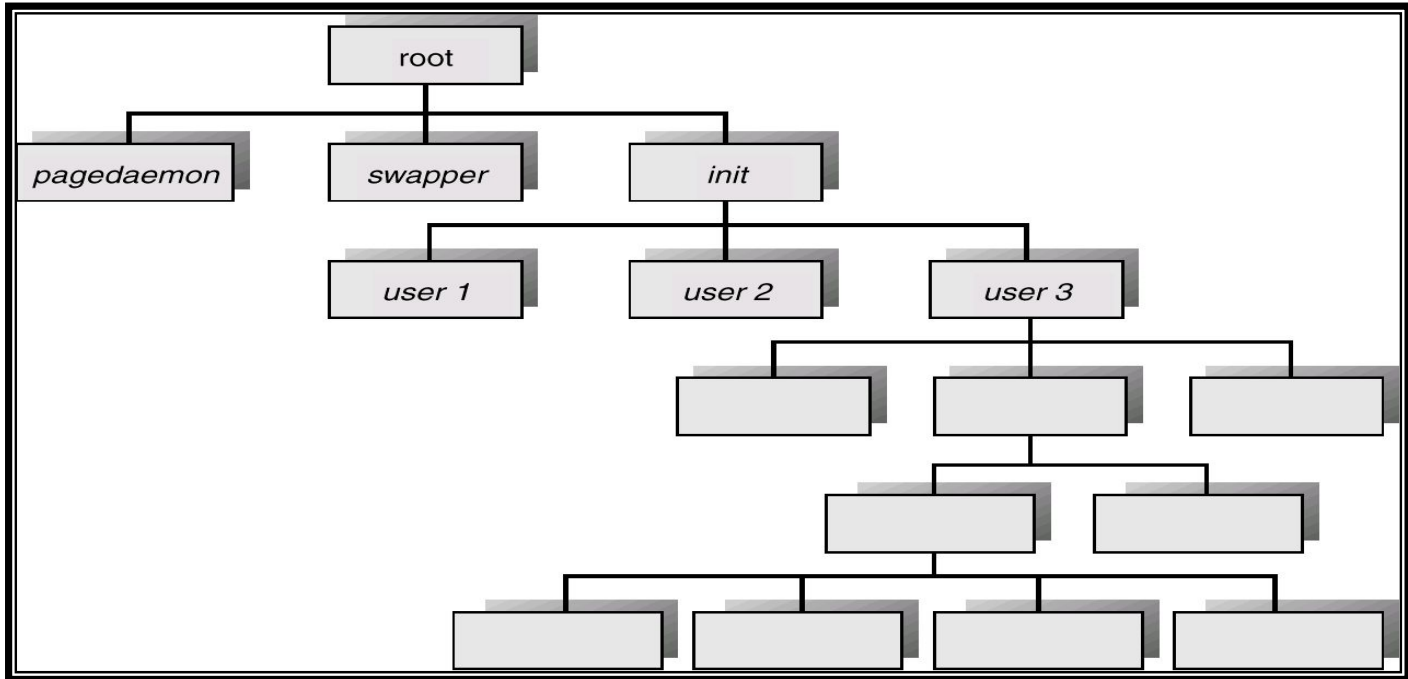
# Process Creation

On boot, kernel starts an init process (usually systemd on ubuntu systems), which takes care of creating all other process.

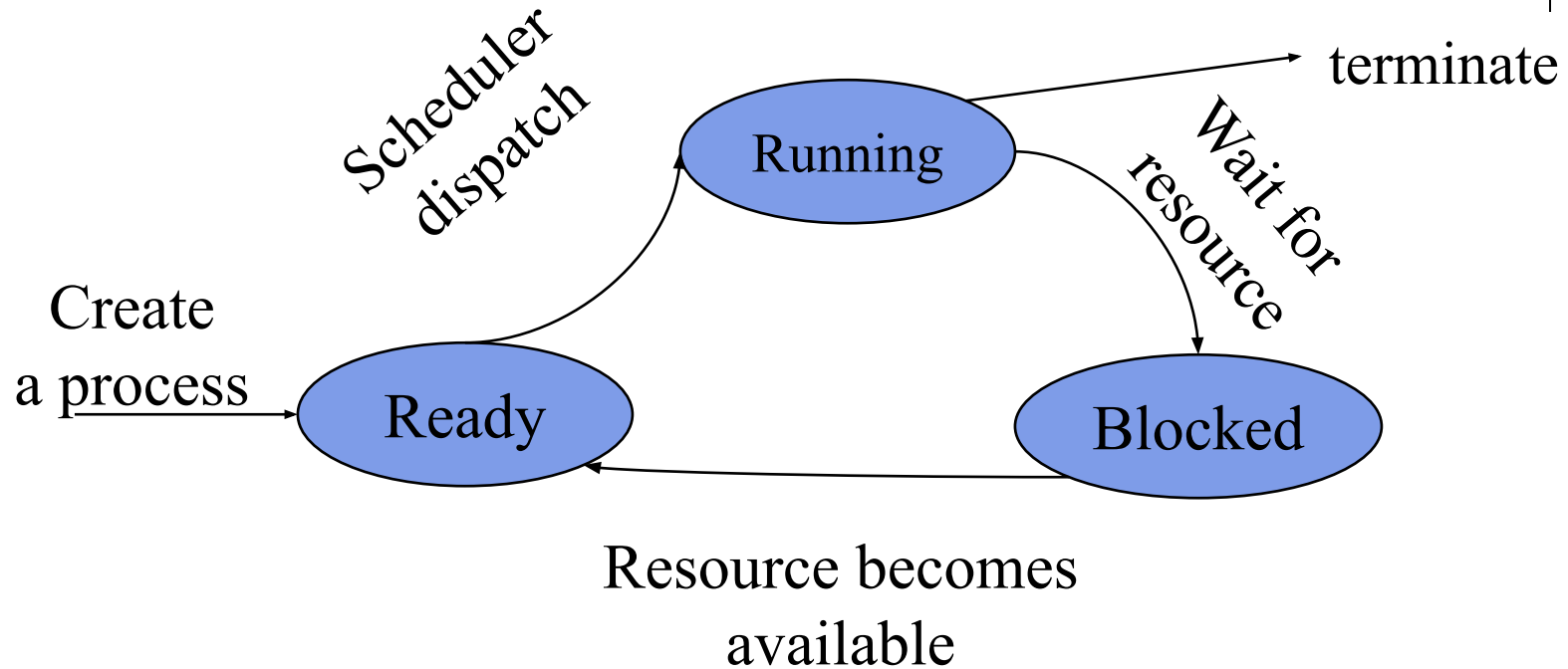
**Init can never die.**



Process Tree



# Process State Transition



# Process Information maintained by OS



- Usually Maintained in a structure called **Process Control Block (PCB)**
- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)
- Memory management info
  - Segments, page table, stats, etc

# Process Identifier



- Every process has an ID – process ID
- Does a program know its process ID?
- When a program is running, how does the process know its ID?



# OS Support for Process



- Support to create process
- Support to wait for a process completion
- Support to terminate a process

# OS Process API



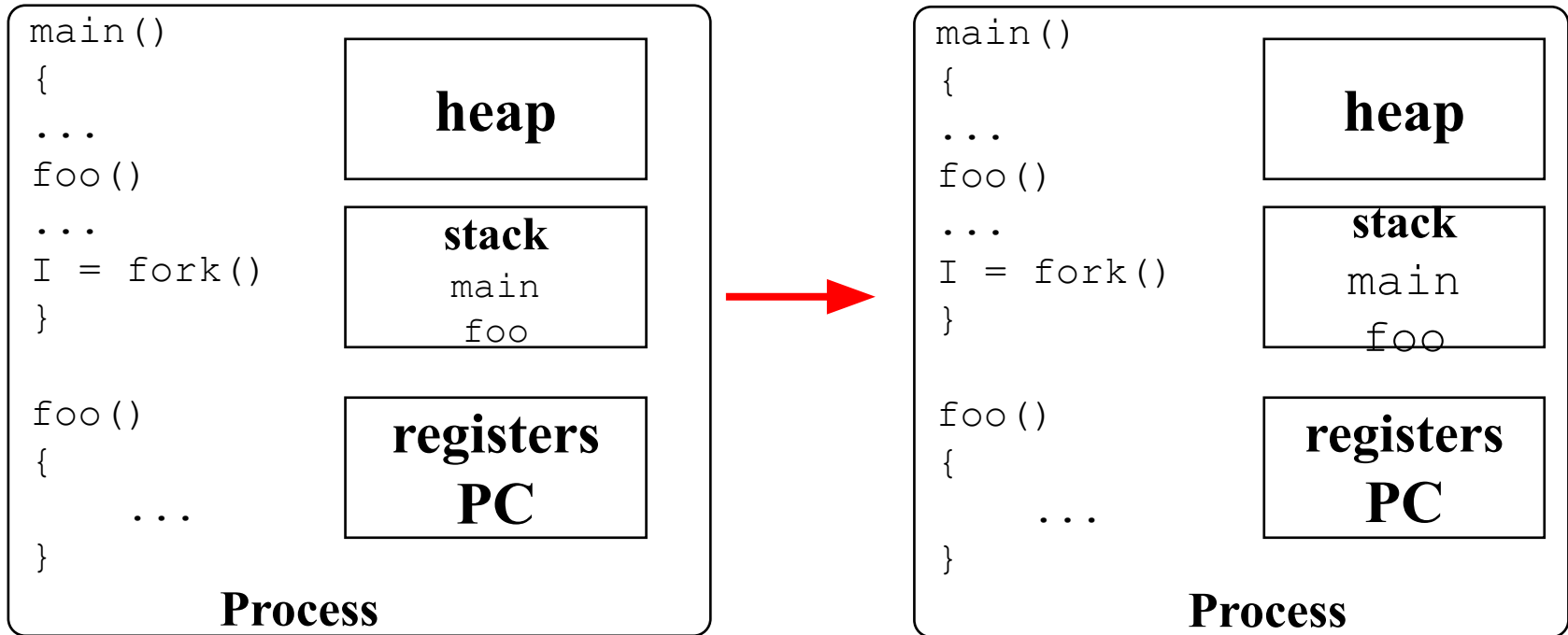
- 4 system calls related to process creation/termination:
  - Process Creation:
    - fork/clone – create a copy of this process
    - exec – replace this process with this program
  - Wait for completion:
    - wait – wait for child process to finish
  - Terminate a process:
    - kill - send a signal (to terminate) a process

# fork



fork causes OS creates a copy of the calling process:

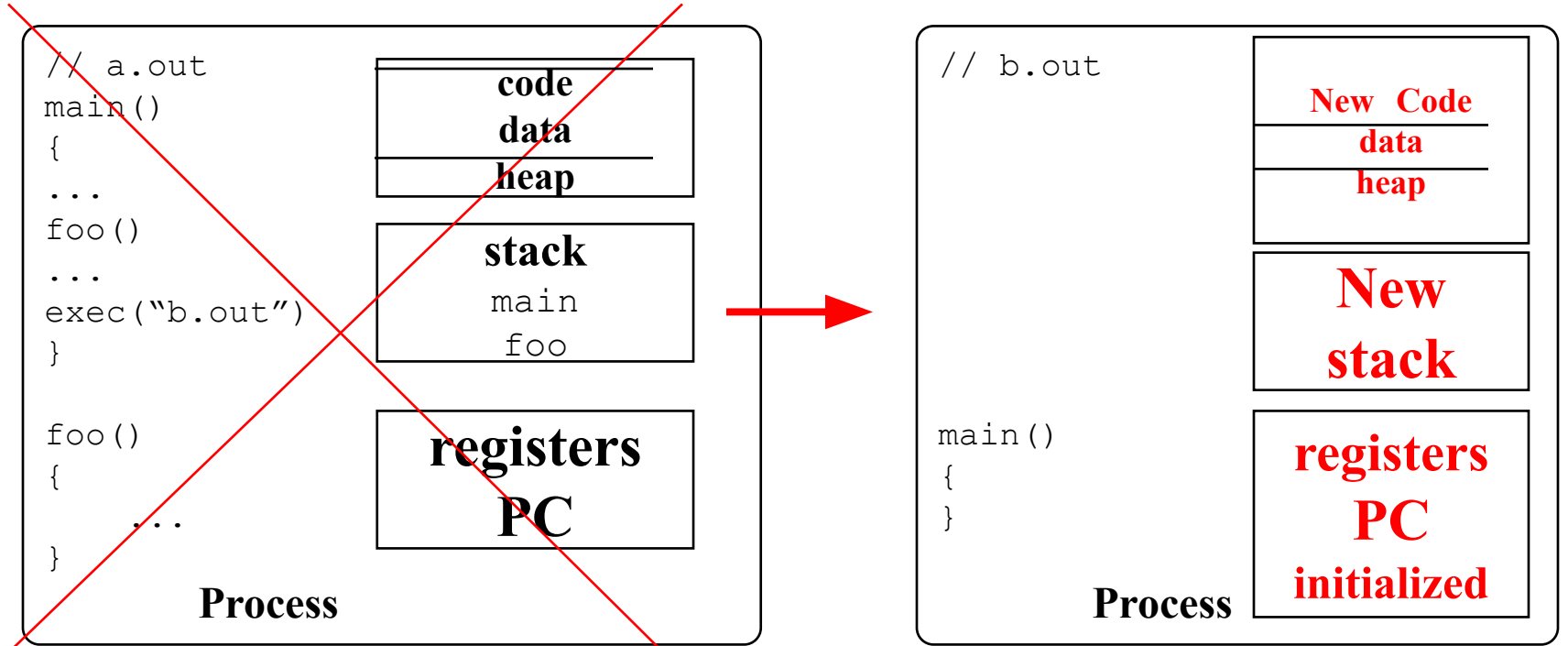
- Why?
- How can we disambiguate between new process and the calling process?



# exec



Replaces current process with the content from new program.



# exec

Example

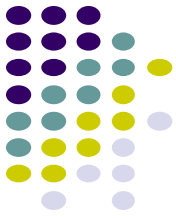


# wait

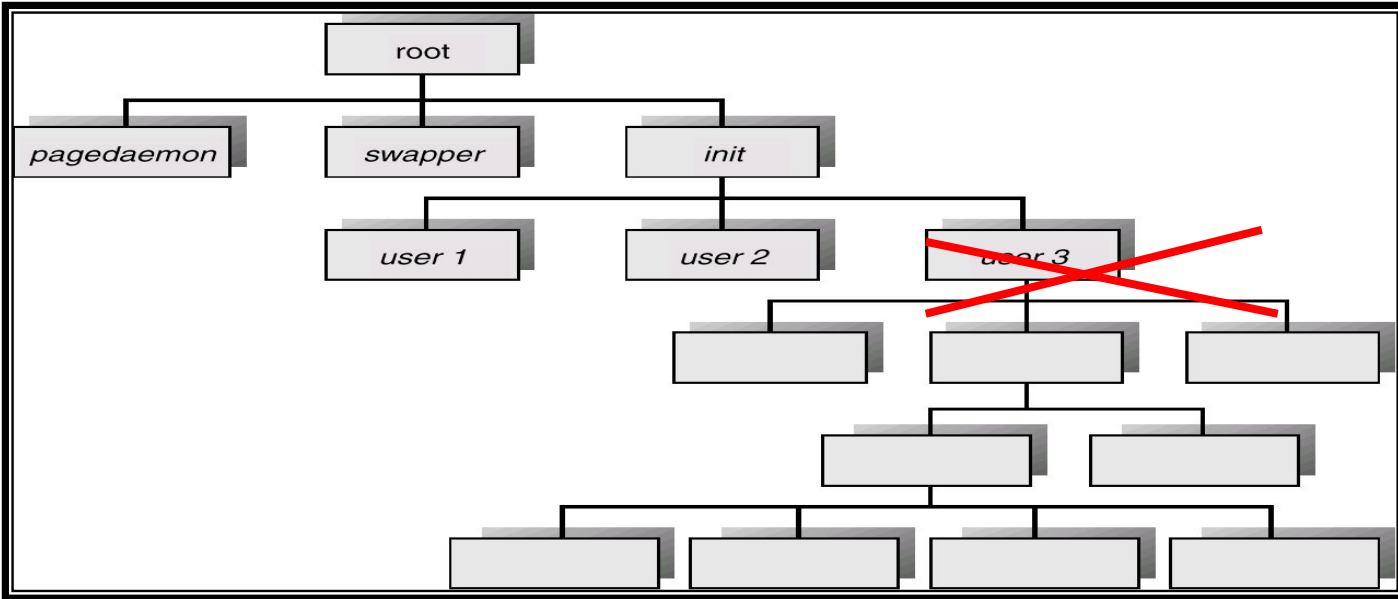
wait for a child process to finish



# wait



What happens when the parent process dies? what happens to child process?



# How our shell works?

- Fork/exec

