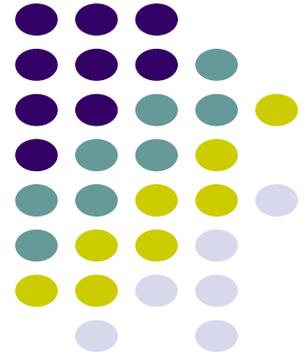


# Midterm Review

---

ECE 469, Mar 10

Aravind Machiry



# Midterm

- Online on Brightspace --- NO LOCKDOWN BROWSER
- Thursday, 12th March --- NO CLASS
- Time: 75 min
- Available Period: 7:30 am - 9:00 pm
- Topics covered till last Thursday (03/05/2026)
- Open notes

# What happens, when we turn on the machine?

1. Power up.
2. BIOS initializes basic devices.
3. After initializing peripheral devices, it will put some initialization code to
  - a. DRAM physical address 0xffff0 ([f000:fff0])
  - b. Copy the code from ROM to RAM
  - c. Run from RAM

```
The target architecture is assumed to be i8086
[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b
0x0000fff0 in ?? ()
```

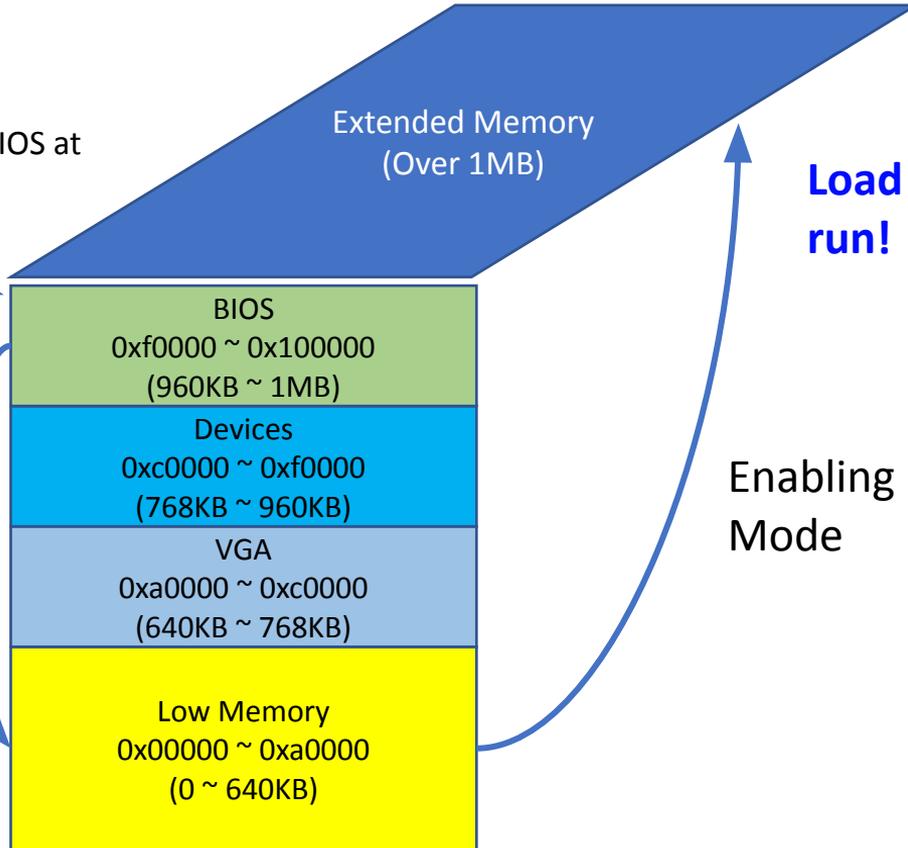
1. What does the code do? Load and run the boot sector from disk
  - b. Read the 1<sup>st</sup> sector from the boot disk (512 bytes)
  - c. Put the sector at 0x7c00
  - d. **Run it!** (set the instruction pointer = 0x7c00)

# Summary!



Read Master Boot Record  
(MBR)  
from the boot disk  
and load it at 0x7c00

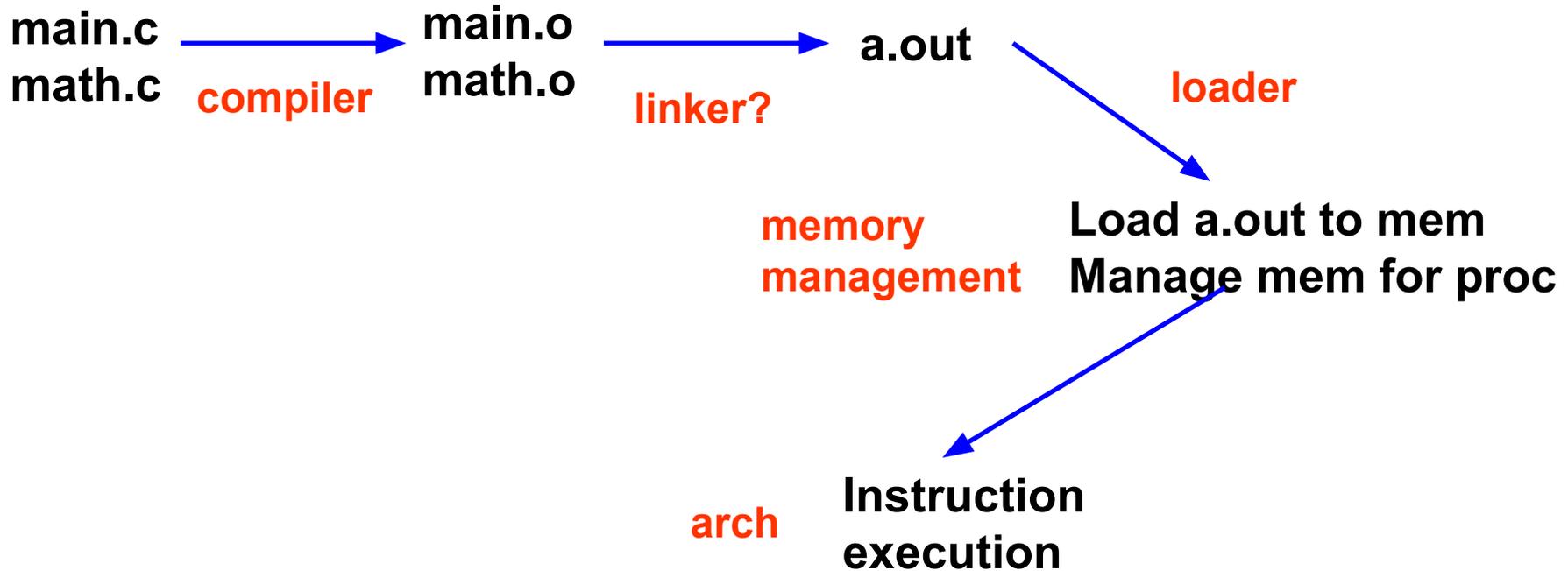
Map code in BIOS at  
f000:ffff



Load kernel and  
run!

Enabling Protected  
Mode

# A gap among Architecture, Compiler and OS courses



# Virtual Memory

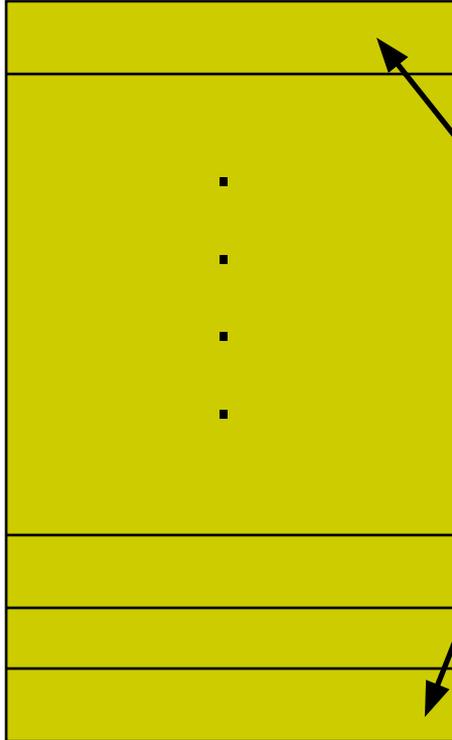
- Three goals
  - Transparency: does not need to know system's internal state
    - Program A is loaded at `0x8048000`. Can Program B be loaded at `0x8048000`?
  - Efficiency: do not waste memory; manage memory fragmentation
    - Can Program B (288KB) be loaded if 288 KB of memory is free, regardless of its allocation?
  - Protection: isolate program's execution environment
    - Can we prevent an overflow from Program A from overwriting Program B's data?

# Paging!

- Idea: Make all chunks of memory the same size, called **pages**
  - Both virtual and physical memory divided into same size chunks.
- For each process, a **page table** defines the base address of each of that process' pages along with existence and read/write bits

# Paging!

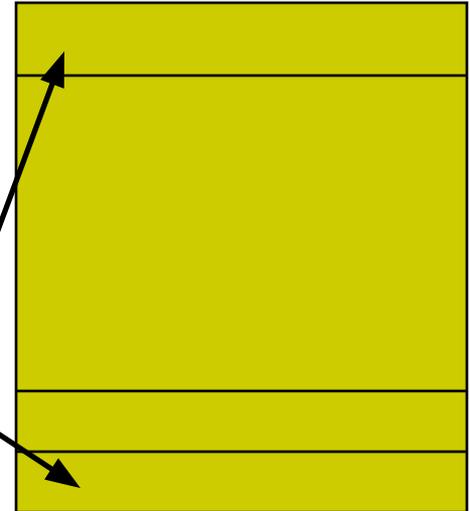
Virtual address



Virtual pages

physical pages

Physical memory



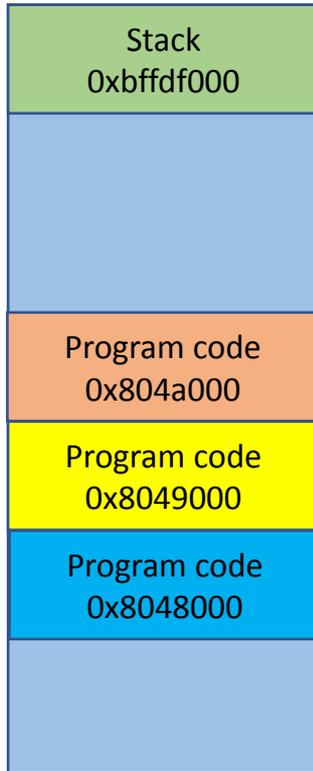
# Page size / fragmentation

- If a page size is too small, it requires a big page table
  - 1B, 4GB
  - 4KB, 4MB
  - 4MB, 4KB
  - 1G, 16B
- If a page size is too big, unused memory in a page will be wasted
  - 1B - 1B (no waste)
  - 4KB – 1B
  - 4MB – 1B
  - 1G – 1B

**Design consideration:  
Memory fragmentation matters!**

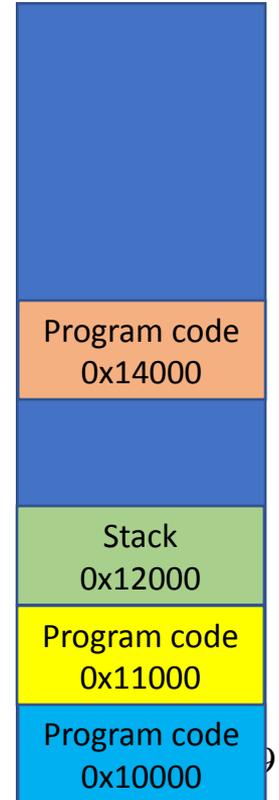
# Virtual Memory - Paging

Having an indirect table that maps virt-addr to phys-addr



Virtual	Physical
0x8048000	0x10000
0x8049000	0x11000
0x804a000	0x14000
0xbffdf000	0x12000
...	...

Physical Memory

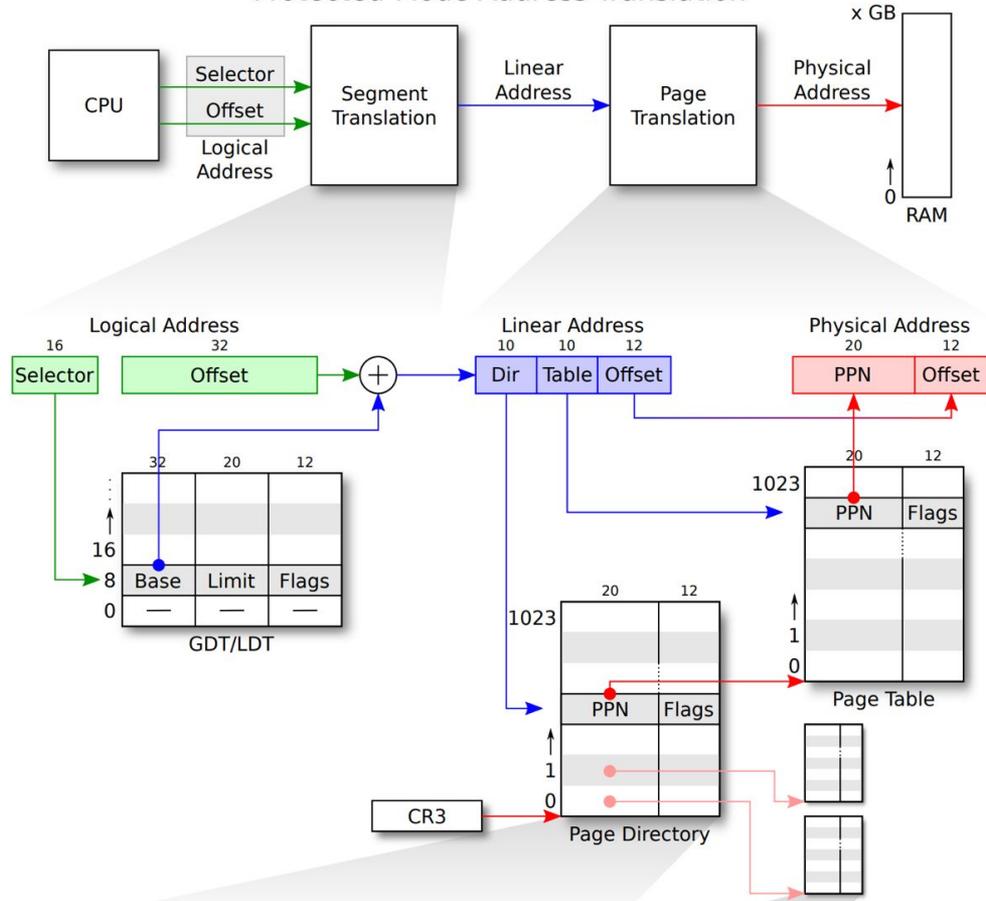


# Page Table

- We access page table by virtual address
- Page size: 4 KB (12bits)
- Page number: 20 bits
- What is the page number and offset of
  - **0x8048000**
  - **0xb7ff3100**



## Protected-Mode Address Translation



# Caching!

- Cache the frequently used page table entries:
  - Exploit locality
    - Translation Lookaside Buffer (TLB)
      - Cache for translation entries.

# Translation Lookaside Buffer (TLB)

- Stores VA-PA mappings and caches them!

VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x0	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

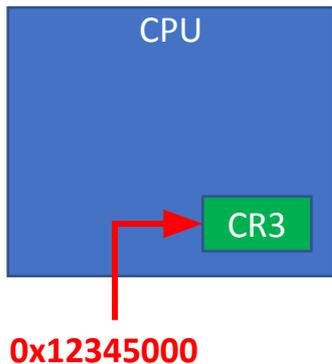
0x12345678 -> 0x678

0x12346678 -> 0x5678

0x12347678 -> 0xff678

0x12348678 -> 0xfff678

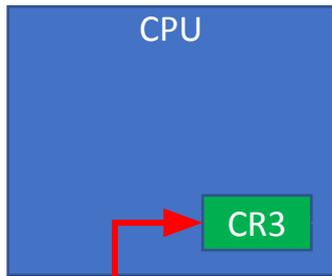
# Without TLB



	Page Directory Entry
0	Addr PT
..	Addr PT
0x48	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x345	0x10000
0x346	0x11000
0x347	0x50000

# Without TLB

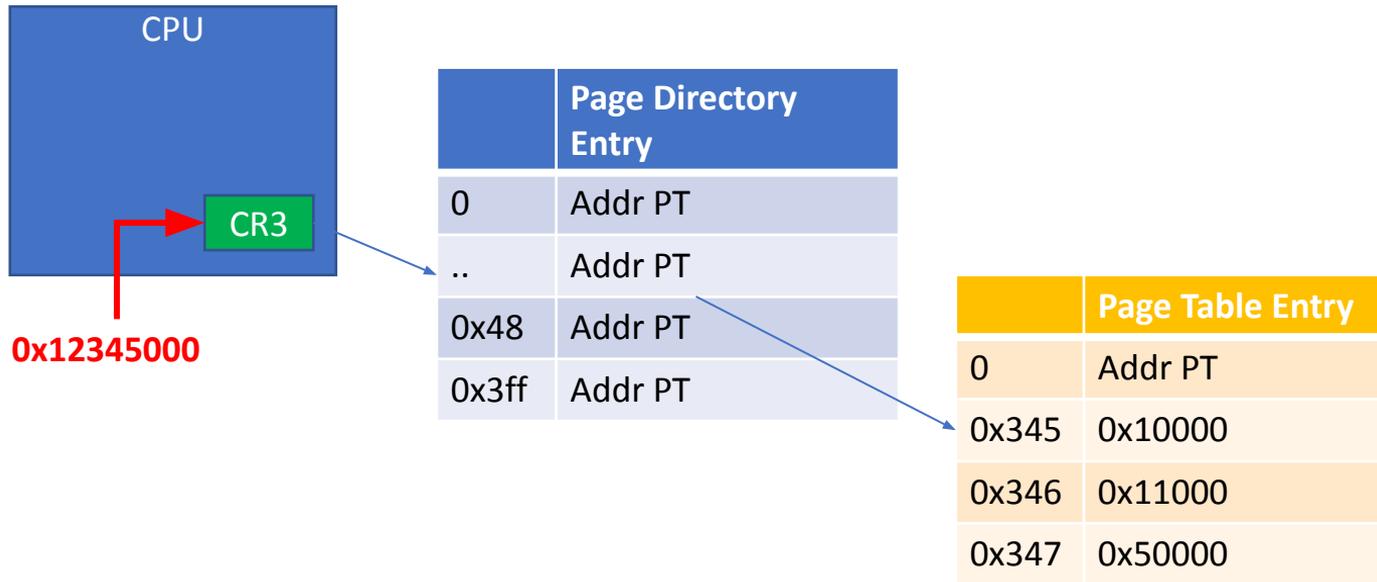


0x12345000

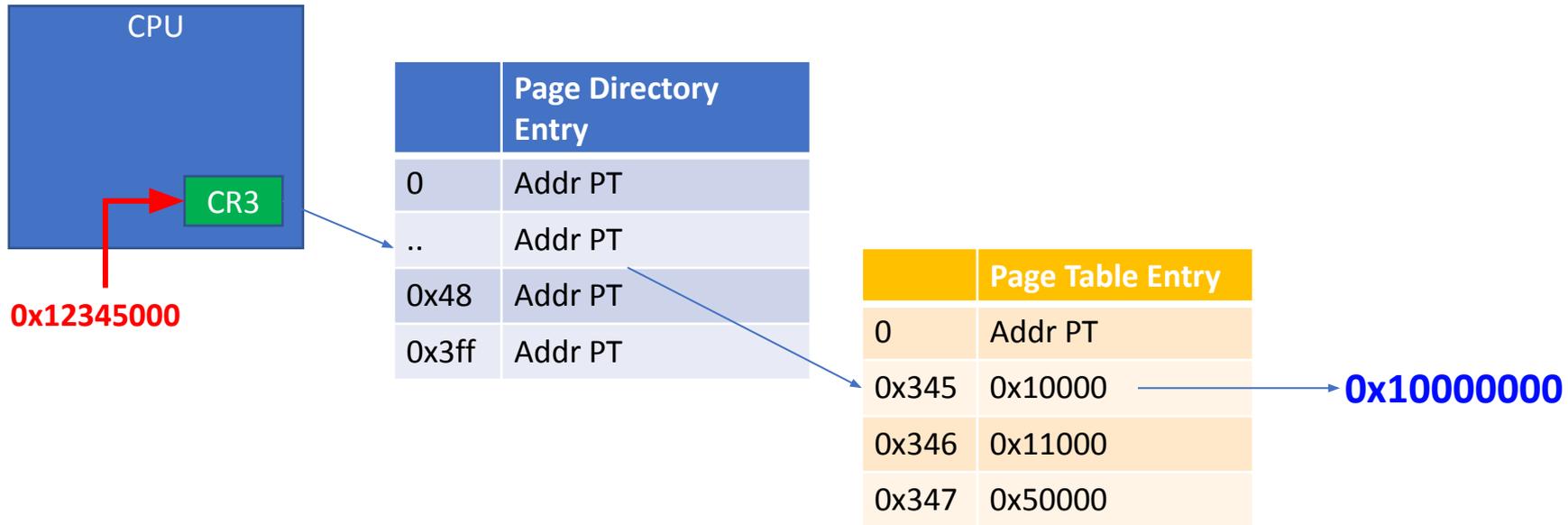
	Page Directory Entry
0	Addr PT
..	Addr PT
0x48	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x345	0x10000
0x346	0x11000
0x347	0x50000

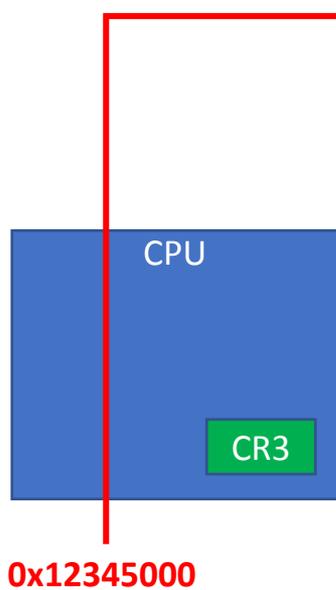
# Without TLB



# Without TLB



# With TLB

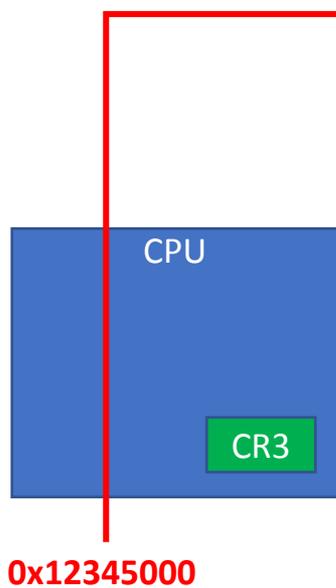


VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x10000	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x48	0x10000
0x49	0x11000
0x4a	0x50000

# With TLB



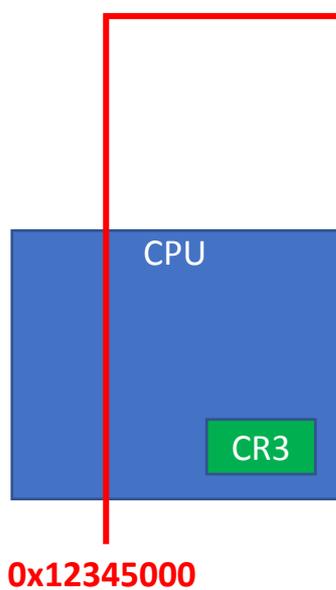
VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x10000	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xfff	1

	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x48	0x10000
0x49	0x11000
0x4a	0x50000

0x10000000

# With TLB



VPN (Virt Page Number)	PPN (Phy Page Number)	Valid
0x12345	0x10000	1
0x12346	0x5	1
0x12347	0xff	1
0x12348	0xffff	1

	Page Directory Entry
0	Addr PT
..	Addr PT
0x20	Addr PT
0x3ff	Addr PT

	Page Table Entry
0	Addr PT
0x48	0x10000
0x49	0x11000
0x4a	0x50000

0x10000000

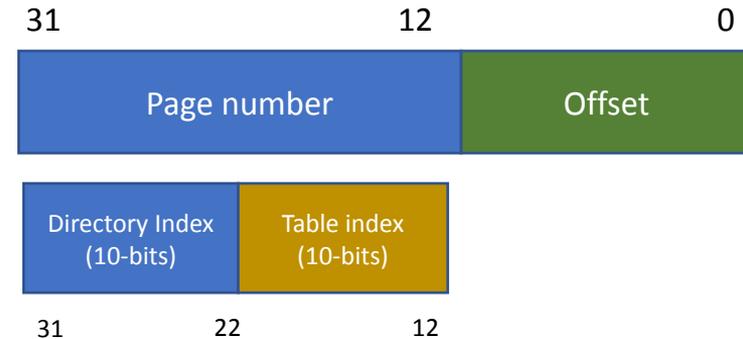
No page table access..

# TLB Performance

- TLB hit requires 4 cycles, 1ns!
- Page table walk requires 2 memory access for translation
  - Uncached: 9 cycles + (42 cycles + 51ns) \* 2
  - [TLB miss] [RAM latency]  
 $2\text{ns} + (10\text{ns} + 51\text{ns}) * 2 = 124\text{ns}$  (**124 times slower...**)
  - Cached:  $9 + 4 * 2 = 17$  cycles if all blocks cached in L1 (4 ns, **4 times slower!**)
    - Data TLB L1: 64 items. 4-way. Miss penalty = 9 cycles. Parallel miss: 1 cycle per access
    - Data TLB L2 (STLB): 1536 items. 12-way. Miss penalty = 17 ? cycles. Parallel miss: 14 cycles per access
    - PDE cache = ? items. Miss penalty = ? cycles.
    - L1 Data Cache Latency = 4 cycles for simple access via pointer
    - L1 Data Cache Latency = 5 cycles for access with complex address calculation (`size_t n, *p; n = p[n]`).
    - L2 Cache Latency = 12 cycles
    - L3 Cache Latency = 42 cycles (core 0) (i7-6700 Skylake 4.0 GHz)
    - L3 Cache Latency = 38 cycles (i7-7700K 4 GHz, Kaby Lake)
    - RAM Latency = 42 cycles + 51 ns (i7-6700 Skylake)

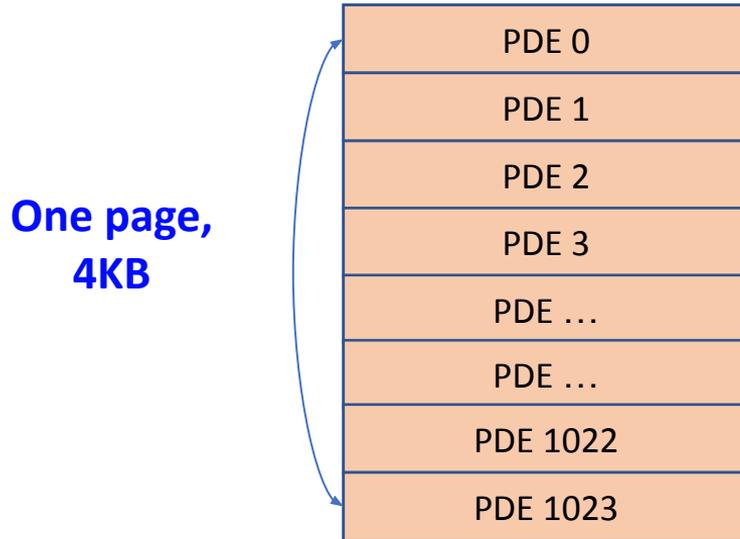
# Page Directory / Table

- In x86 (32-bit), CPU uses 2-level page table
- 10-bit directory index
- 10-bit page table index
- 12-bit offset
- **2-level paging**



# Size of Page Directory!

- Page Size = 4 KB



One page,  
4KB

$$4096 / 4 = 1024 \text{ entries}$$

$$1024 == 2^{10}$$

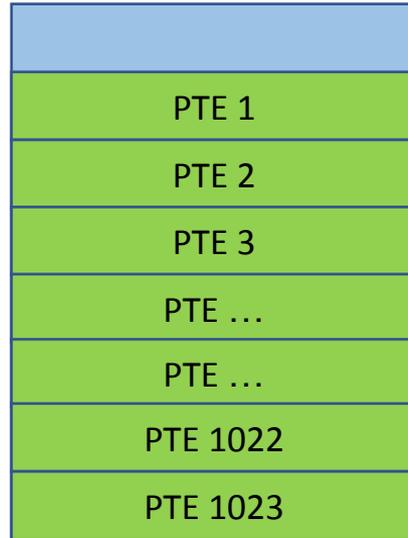
10-bit index for PD

Each entry is 4-byte (32 bits)

# Size of Page Table!

- Page Size = 4 KB

One page,  
4KB



Each entry is 4-byte (32 bits)

$$4096 / 4 = 1024 \text{ entries}$$

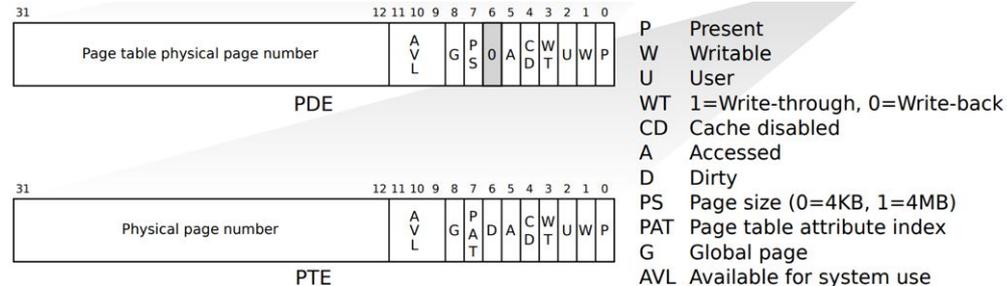
$$1024 == 2^{10}$$

10-bit index for PT

# Permission Flags

- PTE\_P (PRESENT)
  - 0: invalid entry
  - 1: valid entry
- PTE\_W (WRITABLE)
  - 0: read only
  - 1: writable
- PTE\_U (USER)
  - 0: kernel (only ring 0 can access)
  - 1: user (accessible by ring 3)

Page Table Entry		
0	Addr PT	
0x48	0x10000 << 12   PTE_U   PTE_W	Invalid
0x49	0x11000 << 12   PTE_P   PTE_W	Kernel, writable
0x4a	0x50000 << 12   PTE_P   PTE_U	User, read-only



# Cannot have permissions such as ...

- Kernel: RW, User: R
  - PTE\_P | PTE\_W | PTE\_U -> User RW...
  - PTE\_P | PTE\_W -> User --
- Kernel: R, User: RW
  - PTE\_P | PTE\_U | PTE\_W -> Kernel RW...
  - PTE\_P | PTE\_U -> User R...
- Kernel: --, User: RW
  - PTE\_P | PTE\_U | PTE\_W -> Kernel RW...

# Struct PageInfo in JOS

- A **one-to-one** mapping from a **struct PageInfo** to a physical page
  - An 8 byte struct per each physical memory page
  - If we support 128MB memory, then we will create
    - Total number of physical pages:  $128 * 1048576 / 4096 = 32768$
  - Total size = **32768 \* 8 = 262,144 = 256KB**
- A linked-list for managing free physical pages
  - Starting from `page_free_list->pp_link...`
- `pp_ref`
  - Count references
  - Non-zero – in-use
  - Zero – free

```
struct PageInfo {
    // Next page on the free list.
    struct PageInfo *pp_link;

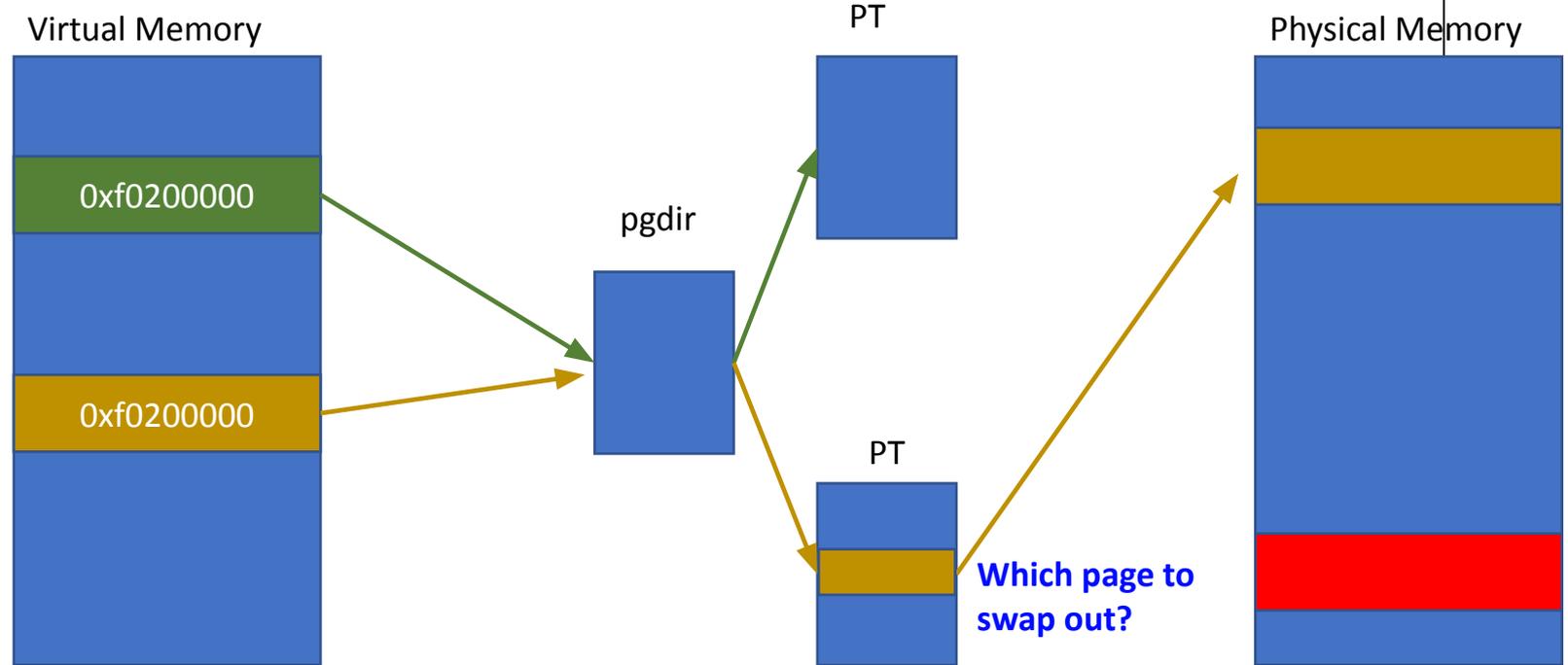
    // pp_ref is the count of pointers (usually in page table entries)
    // to this page, for pages allocated using page_alloc.
    // Pages allocated at boot time using pmap.c's
    // boot_alloc do not have valid reference count fields.

    uint16_t pp_ref;
};
```

# Handling low memory

- Suppose you have 8GB of main memory
- Can you run a program that its program size is 16GB?
  - Yes, you can load them part by part
  - This is because we do not use all of data at the same time
- Can your OS do this execution seamlessly to your application?

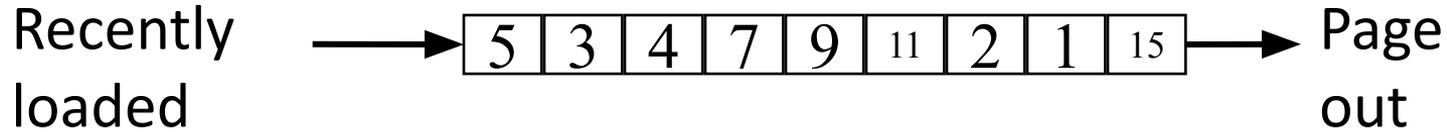
# Selecting Page to Swap out!



# Performance metric for Page Replacement algorithms

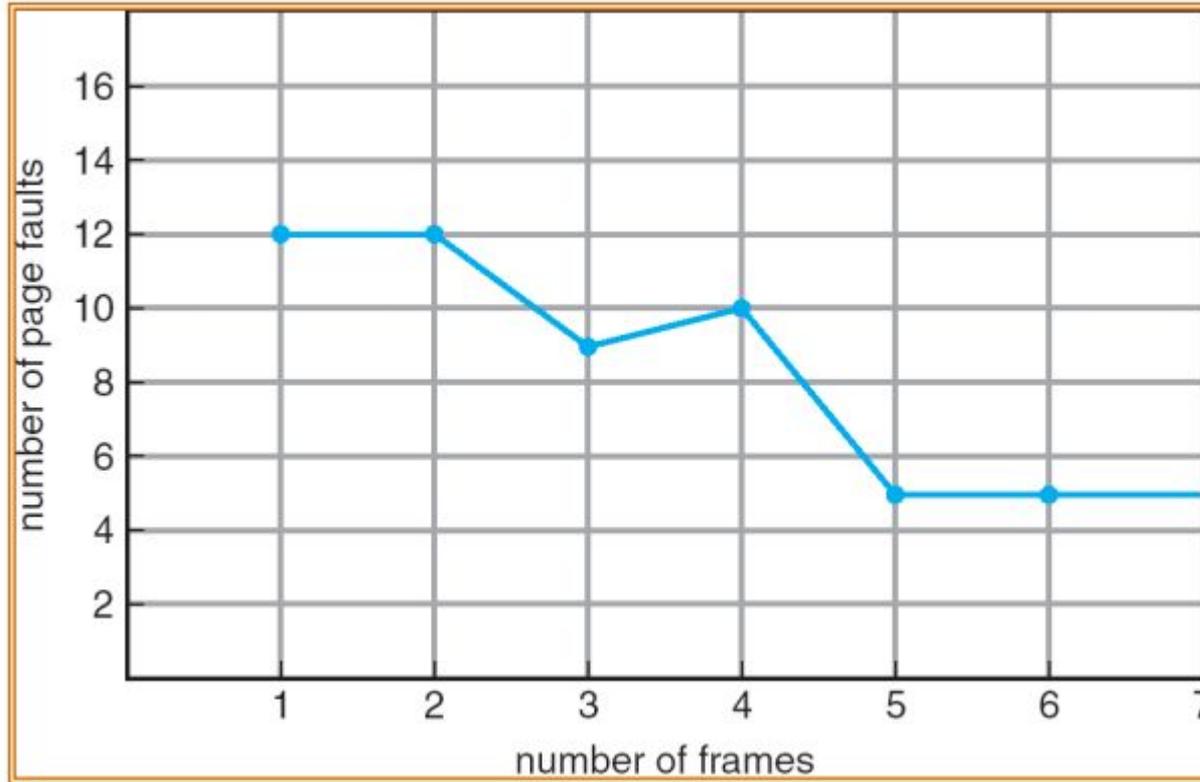
- Give a sequence of memory accesses, minimize the # of page faults
  - Similar to cache miss rate
  - What about hit latency and miss latency?

# First In First Out (FIFO)



- Algorithm
  - Throw out the oldest page
- Pros
  - Low-overhead implementation
- Cons
  - No frequency/no recency  may replace the heavily used pages

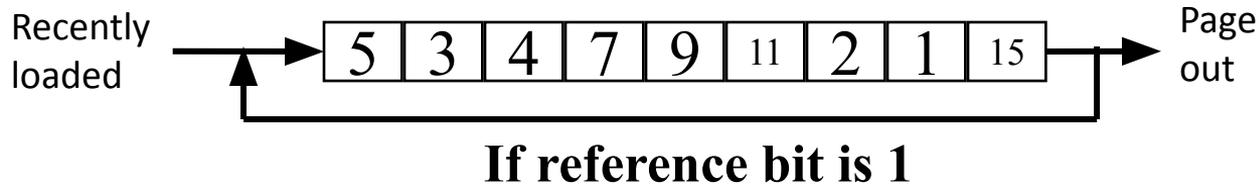
# FIFO illustrating Belady's anomaly



# Optimal or MIN

- Algorithm (also called Belady's Algorithm)
  - Replace the page that won't be used for the **longest** time
- Pros
  - Minimal page faults (can you prove it?)
  - Used as an **off-line** algorithm for perf. analysis
- Cons
  - **No on-line** implementation
- What was the CPU scheduling algorithm of similar nature?

# FIFO with Second Chance



- Algorithm
  - Check the reference-bit of the oldest page (first in)
  - If it is 0, then replace it
  - If it is 1, clear the referent-bit, put it to the end of the list, and continue searching
- Pros
  - Fast
  - Frequency  do not replace a heavily used page
- Cons
  - The worst case may take a long time

# Approximate LRU

Initial	Interval 1	Interval 2	Interval 3	Interval 4
00000000	00000000	00000000	<u>1</u> 0000000	01000000
00000000	00000000	<u>1</u> 0000000	01000000	<u>1</u> 0100000
00000000	<u>1</u> 0000000	<u>1</u> 1000000	<u>1</u> 1100000	01110000
00000000	00000000	00000000	00000000	<u>1</u> 0000000

Page Fault Victim?

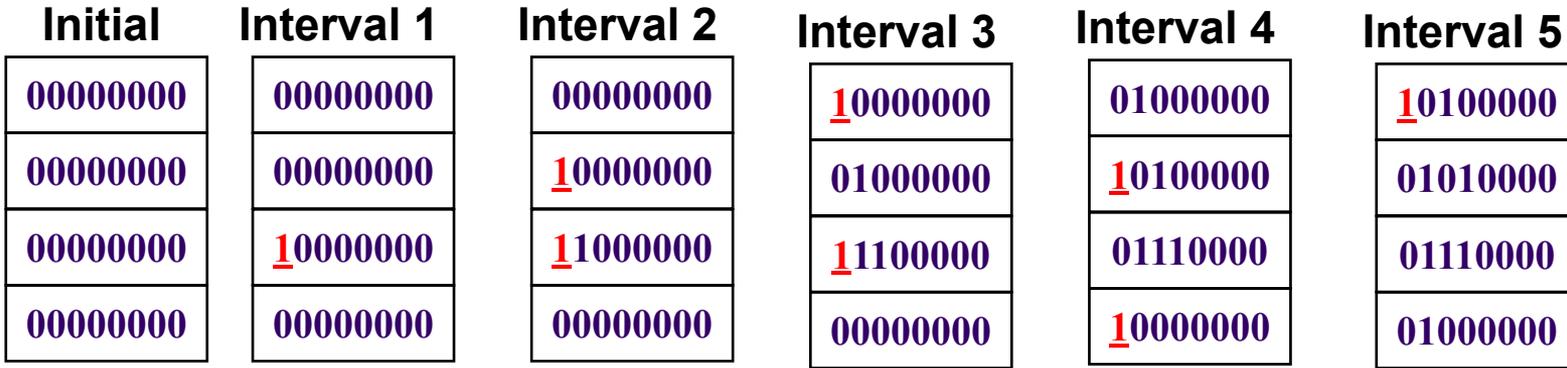
Interval 4

Page Table

Ref	Frame #	Ref	Frame #
1	3	0	3
0	2	0	2
0	0	0	0
1	1	0	1



# Approximate LRU



Page Fault Victim?

**Interval 5**

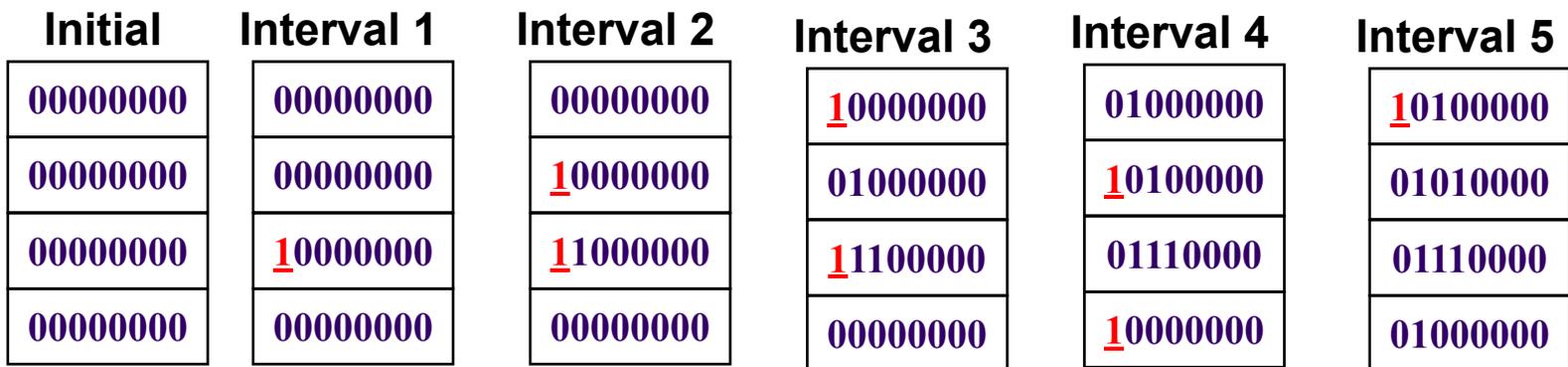
Page Table

Ref	Frame #
0	3
0	2
<b>1</b>	<b>0</b>
0	1



Ref	Frame #
0	3
0	2
0	0
0	1

# Approximate LRU



- Algorithm
  - At regular interval, OS shifts reference bits (in PTE) into counters (and clear reference bits)
  - Replacement: Pick the page with the “smallest counter”
- How many bits are enough?
  - In practice 8 bits are quite good
- Pros: **Require one reference bit, small counter/page**
- Cons: **Require looking at many counters (or sorting)**

# 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**

**Code**

**Data**

**heap**

**stack**

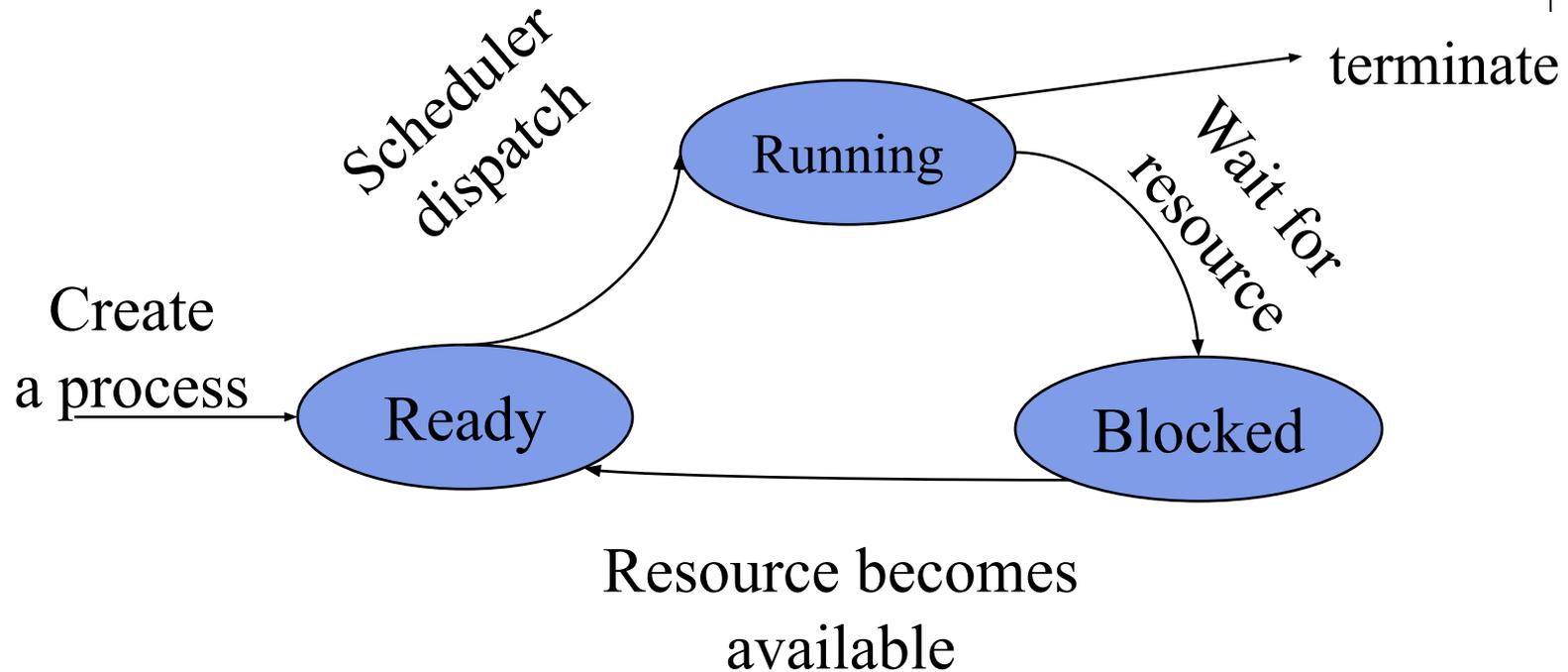
main

foo

**registers**

**PC**

# Process State Transition



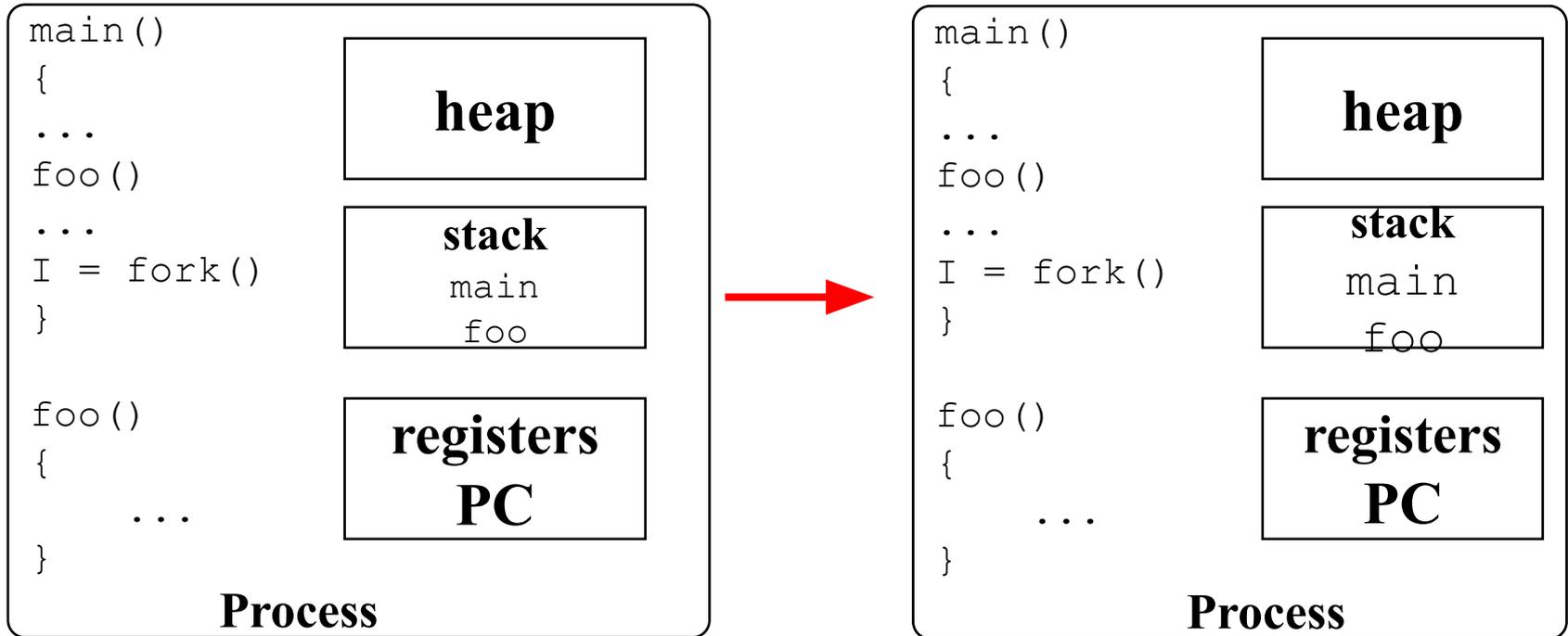
# OS Process API

- 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

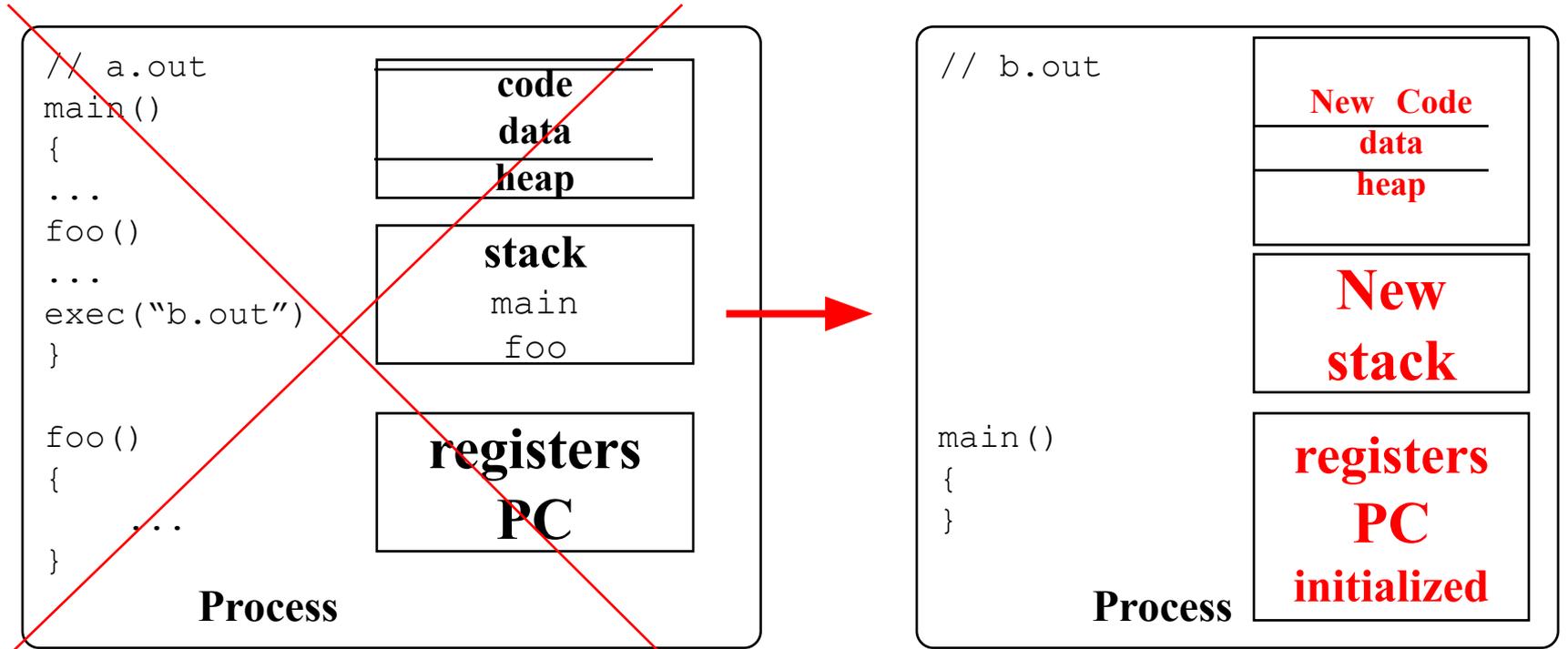
fork causes OS creates a copy of the calling process:

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



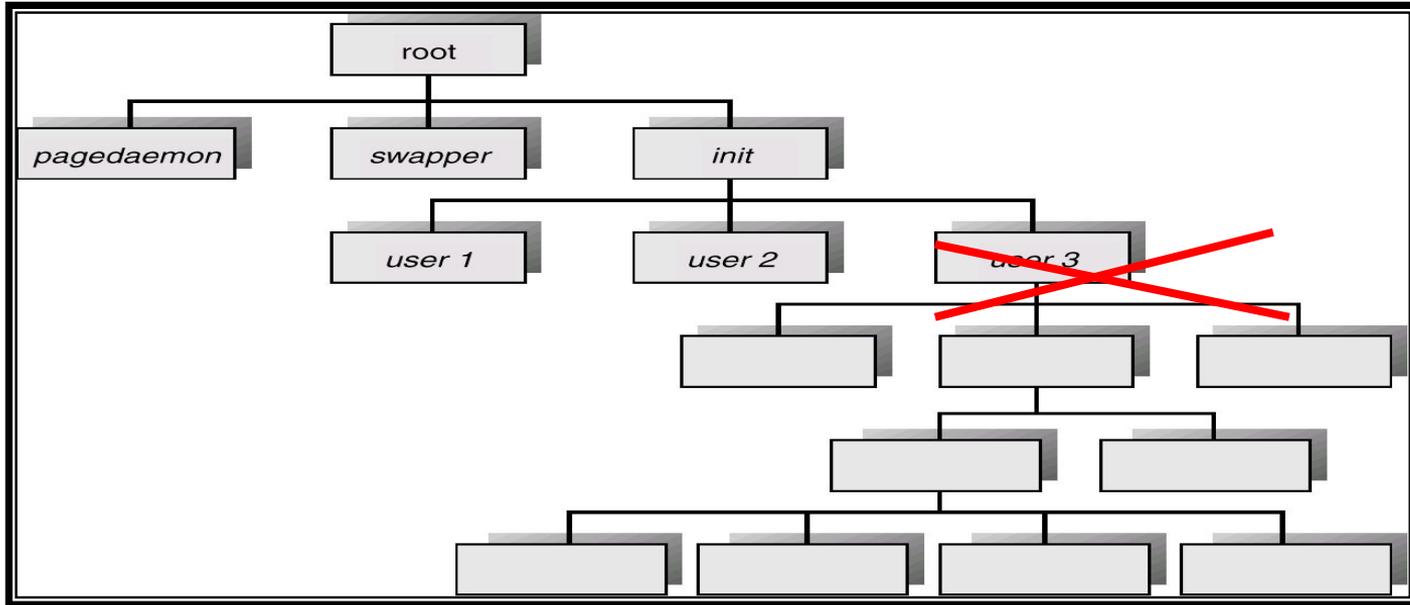
# exec

Replaces current process with the content from new program.



# wait

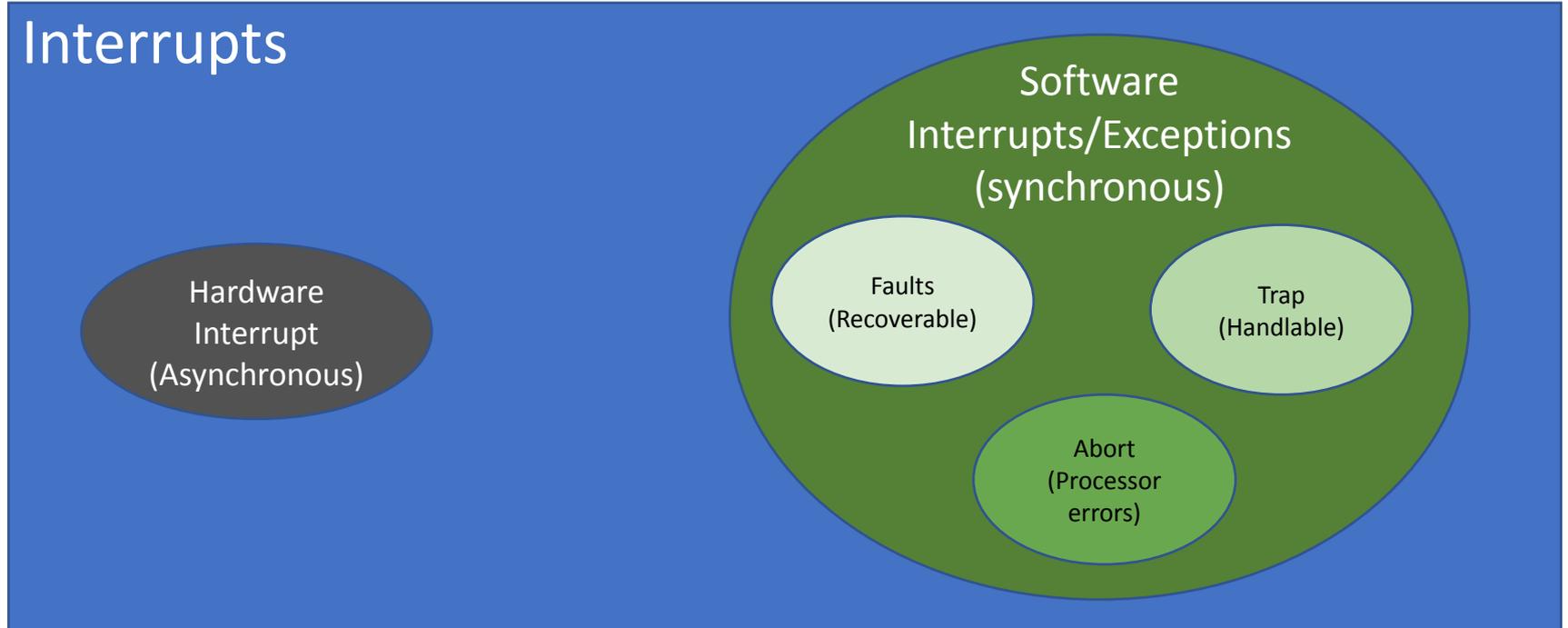
What happens when the parent process dies? what happens to child process?



# Interrupts

- Hardware Interrupts
- Software Interrupts

# Interrupts classification



# Handling Interrupts

- Interrupts are numbered
- We need to define “what to do” (i.e., code to run) when an interrupt with corresponding number occurs

# Handling Interrupts

- Setting an Interrupt Descriptor Table (IDT)

Interrupt Number	Code address
0 (Divide error)	t_divide
1 (Debug)	t_debug
2 (NMI, Non-maskable Interrupt)	t_nmi
3 (Breakpoint)	t_brkpt
4 (Overflow)	t_oflow
...	
8 (Double Fault)	t_dblflt
...	
14 (Page Fault)	t_pgflt
...	...
0x30 (syscall in JOS)	t_syscall

```
TRAPHANDLER_NOEC(t_divide, T_DIVIDE); // 0
TRAPHANDLER_NOEC(t_debug, T_DEBUG); // 1
TRAPHANDLER_NOEC(t_nmi, T_NMI); // 2
TRAPHANDLER_NOEC(t_brkpt, T_BRKPT); // 3
TRAPHANDLER_NOEC(t_oflow, T_OFLOW); // 4
TRAPHANDLER_NOEC(t_bound, T_BOUND); // 5
TRAPHANDLER_NOEC(t_illop, T_ILLOP); // 6
TRAPHANDLER_NOEC(t_device, T_DEVICE); // 7

TRAPHANDLER(t_dblflt, T_DBLFLT); // 8

TRAPHANDLER(t_tss, T_TSS); // 10
TRAPHANDLER(t_segnp, T_SEGNP); // 11
TRAPHANDLER(t_stack, T_STACK); // 12
TRAPHANDLER(t_gpflt, T_GPFLT); // 13
TRAPHANDLER(t_pgflt, T_PGFLT); // 14

TRAPHANDLER_NOEC(t_fperr, T_FPERR); // 16

TRAPHANDLER(t_align, T_ALIGN); // 17

TRAPHANDLER_NOEC(t_mchk, T_MCHK); // 18
TRAPHANDLER_NOEC(t_simderr, T_SIMDERR); // 19

TRAPHANDLER_NOEC(t_syscall, T_SYSCALL); // 48, 0x30
```

# JOS Interrupt Handling

- Setup the IDT at trap\_init() in kern/trap.c

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
}
```

# JOS Interrupt Handling

- Setup the IDT at trap\_init() in kern/trap.c
- Interrupt arrives to CPU!
- Call interrupt handler in IDT
- Call \_alltraps (in kern/trapentry.S)

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
}

#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

# JOS Interrupt Handling

- Setup the IDT at trap\_init() in kern/trap.c
- Interrupt arrives to CPU!
- Call interrupt handler in IDT
- Call \_alltraps (in kern/trapentry.S)
- Call trap() in kern/trap.c

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
}
```

```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds    Build a
    pushl %es    Trapframe!
    pushal
```

# JOS Interrupt Handling

```
struct Trapframe {
    struct PushRegs tf_regs;
    uint16_t tf_es;
    uint16_t tf_padding1;
    uint16_t tf_ds;
    uint16_t tf_padding2;
    uint32_t tf_trapno;
    /* below here defined by x86 hardware */
    uint32_t tf_err;
    uintptr_t tf_eip;
    uint16_t tf_cs;
    uint16_t tf_padding3;
    uint32_t tf_eflags;
    /* below here only when crossing rings, such as from user to kernel */
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```

```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds
    pushl %es
    pushal
```

**Build a Trapframe!**

# JOS Interrupt Handling

- Setup the IDT at trap\_init() in kern/trap.c
- Interrupt arrives to CPU!
- Call interrupt handler in IDT
- Call \_alltraps (in kern/trapentry.S)
- Call trap() in kern/trap.c

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
}
```

```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds    Build a
    pushl %es    Trapframe!
    pushal
```

```
void
trap(struct Trapframe *tf)
{
```

# JOS Interrupt Handling

- Setup the IDT at trap\_init() in kern/trap.c
- Interrupt arrives to CPU!
- Call interrupt handler in IDT
- Call \_alltraps (in kern/trapentry.S)
- Call trap() in kern/trap.c
- Call trap\_dispatch() in kern/trap.c

```
static void
trap_dispatch(struct Trapframe *tf)
{
    // Handle processor exceptions.
    // LAB 3: Your code here.
```

```
void
trap_init(void)
{
    extern struct Segdesc gdt[];

    // LAB 3: Your code here.
    SETGATE(idt[T_DIVIDE], 0, GD_KT, t_divide, 0);
    SETGATE(idt[T_DEBUG], 0, GD_KT, t_debug, 0);
```

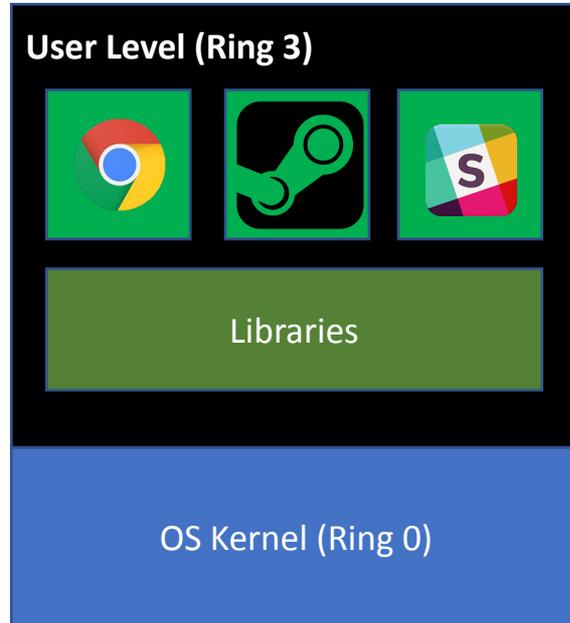
```
#define TRAPHANDLER_NOEC(name, num)
    .globl name;
    .type name, @function;
    .align 2;
    name:
    pushl $0;
    pushl $(num);
    jmp _alltraps
```

```
/*
 * Lab 3: Your code here for _alltraps
 */

_alltraps:
    pushl %ds    Build a
    pushl %es    Trapframe!
    pushal
```

```
void
trap(struct Trapframe *tf)
{
```

# Syscall: User/Kernel communication



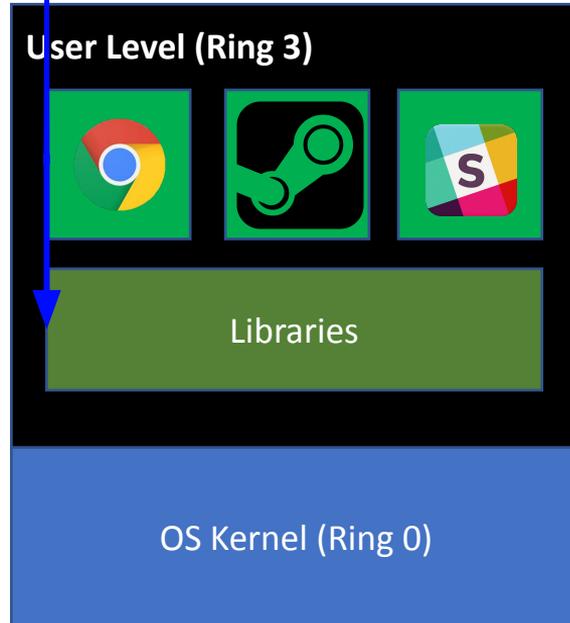
```
int main() {  
    printf("ECE469");  
}
```



# Syscall: User/Kernel communication

`printf("ECE469")`

A library call in ring 3



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



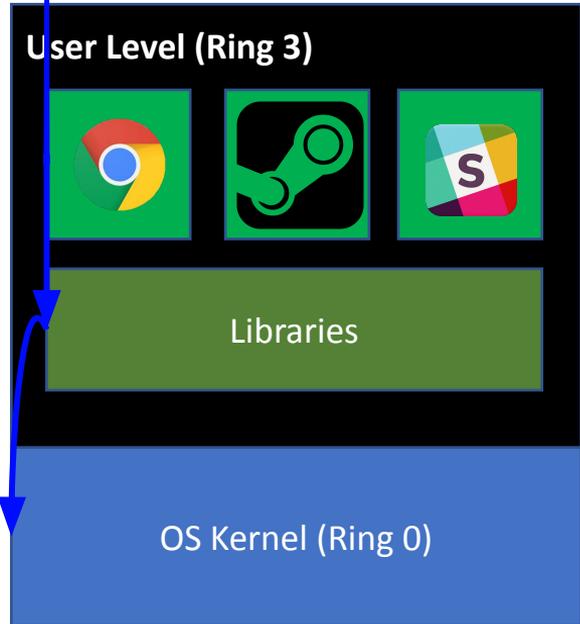
# Syscall: User/Kernel communication

`printf("ECE469")`

A library call in ring 3

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

A system call, **From ring 3**



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



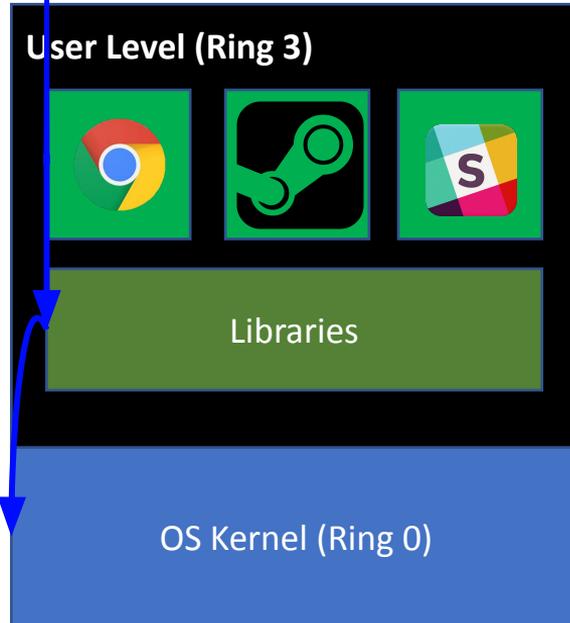
# Syscall: User/Kernel communication

`printf("ECE469")`

A library call in ring 3

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

A system call, **From ring 3**



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

Interrupt!, switch from ring3 to ring0



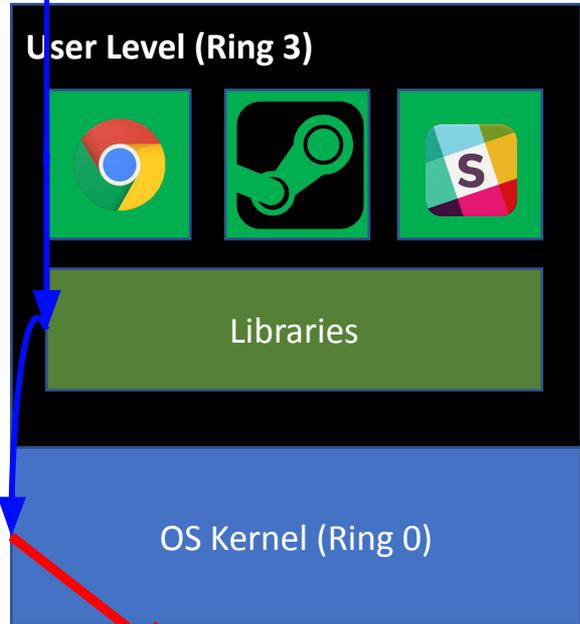
# Syscall: User/Kernel communication

`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

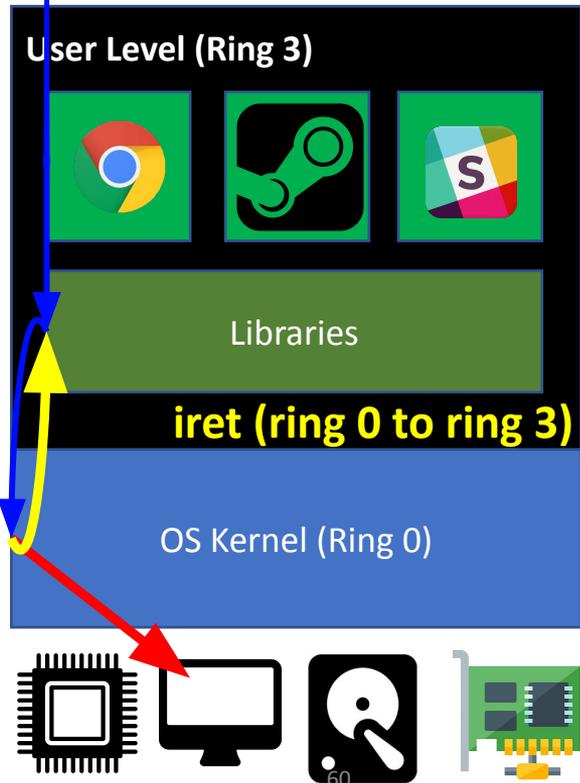
`sys_write(1, "ECE469", 6);`

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");  
}
```

# Syscall: User/Kernel communication

`printf("ECE469")`

A library call in ring 3

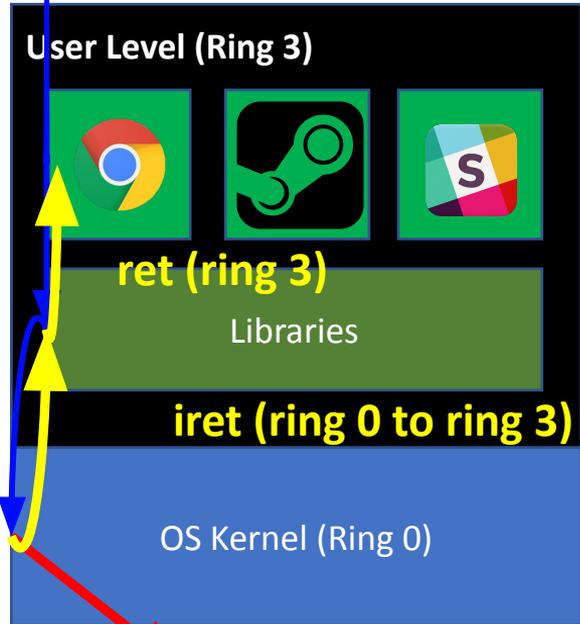
`sys_write(1, "ECE469", 6);`

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");  
}
```

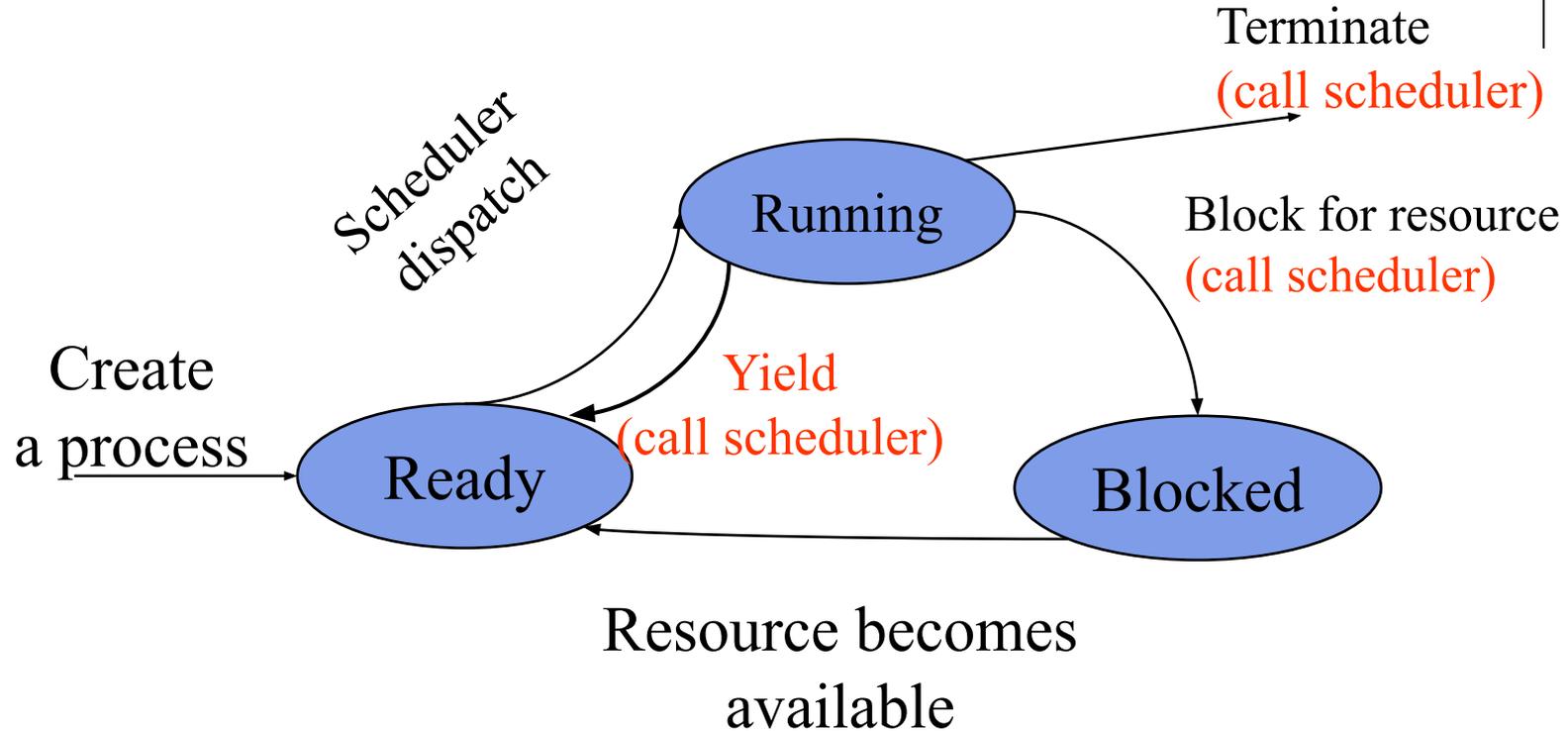
# Invoking Syscalls

- Set all arguments in the registers
  - Order: edx ecx ebx edi esi
- int \$0x30 (in JOS)
  - Software interrupt 48
- int \$0x80 (in 32bit Linux)
  - Software interrupt 128

# Invoking Syscalls in User mode

- User calls a function
  - `cprintf` -> calls `sys_cputs()`
- `sys_cputs()` at user code will call `syscall()` (`lib/syscall.c`)
  - This `syscall()` is at `lib/syscall.c`
  - Set args in the register and then
- `int $0x30`
- Now kernel execution starts...

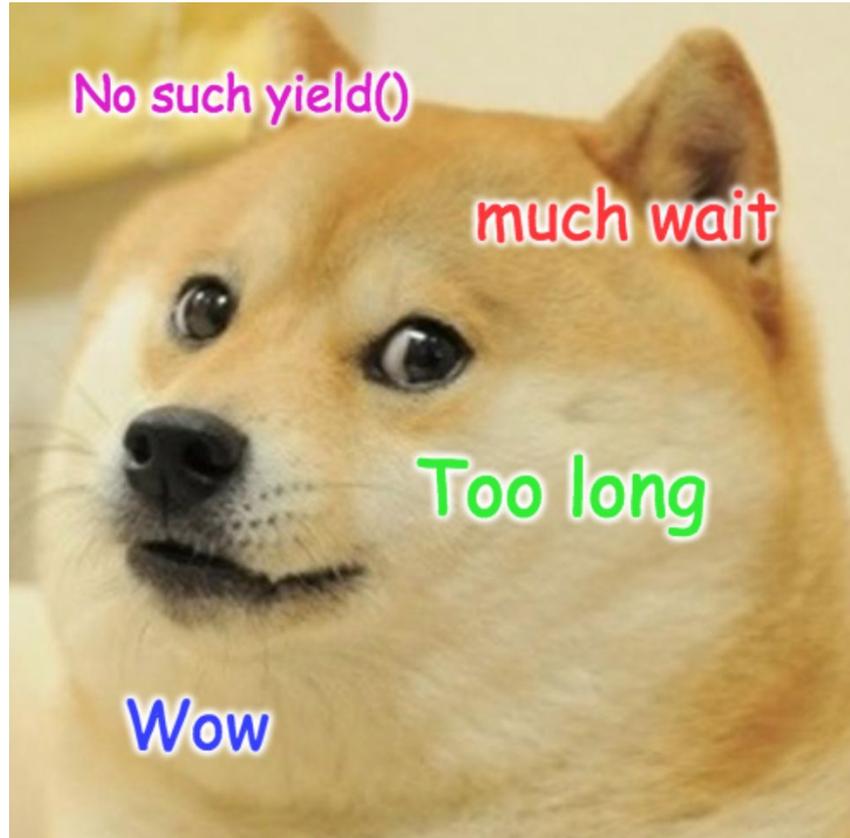
# Non-Preemptive Scheduling



# Non-Preemptive Scheduling

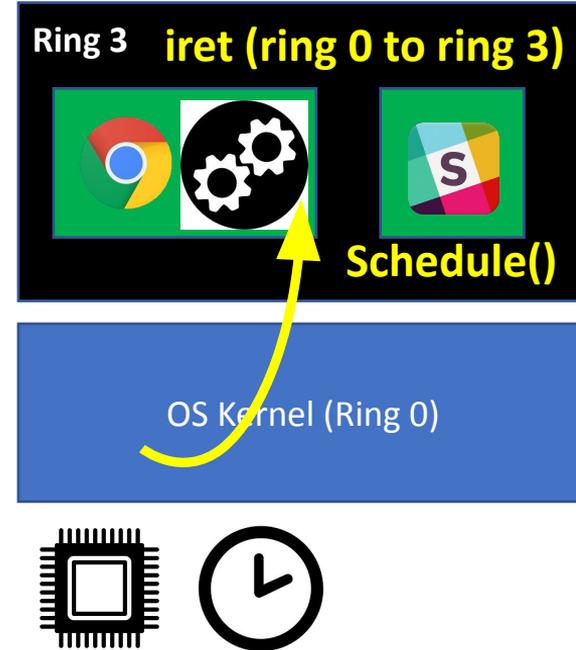
- Any issues?
- What if a process runs:

```
int main() {  
    while(1);  
}
```



# Preemptive Scheduling

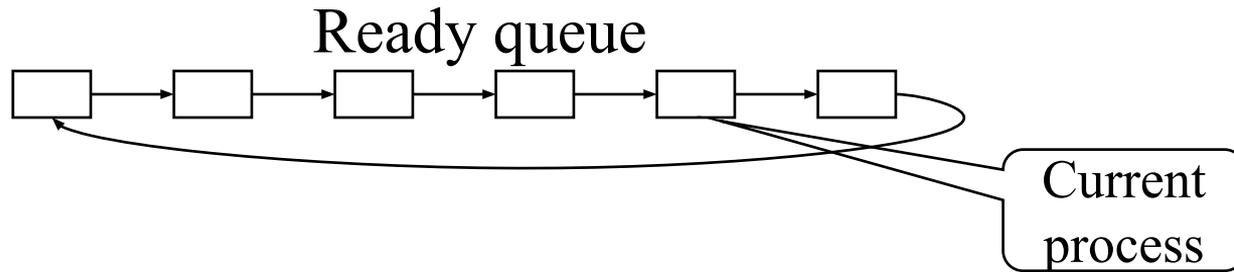
- Preemptive Multitasking (Lab 4)
- CPU generates an interrupt to force execution at kernel after some time quantum
  - E.g., 1000Hz, on each 1ms..
- Guaranteed execution in kernel
  - Let kernel mediate resource contention



# Goals and Assumptions

- Goals (Performance metrics)
  - Minimize turnaround time
    - avg time to complete a job
    - $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
  - Maximize throughput
    - operations (jobs) per second
    - Minimize overhead of context switches: large quanta
    - Efficient utilization (CPU, memory, disk etc)
  - Short response time
    - $T_{\text{response}} = T_{\text{firstrun}} - T_{\text{arrival}}$
    - type on a keyboard
    - Small quanta
  - Fairness
    - fair, no starvation, no deadlock

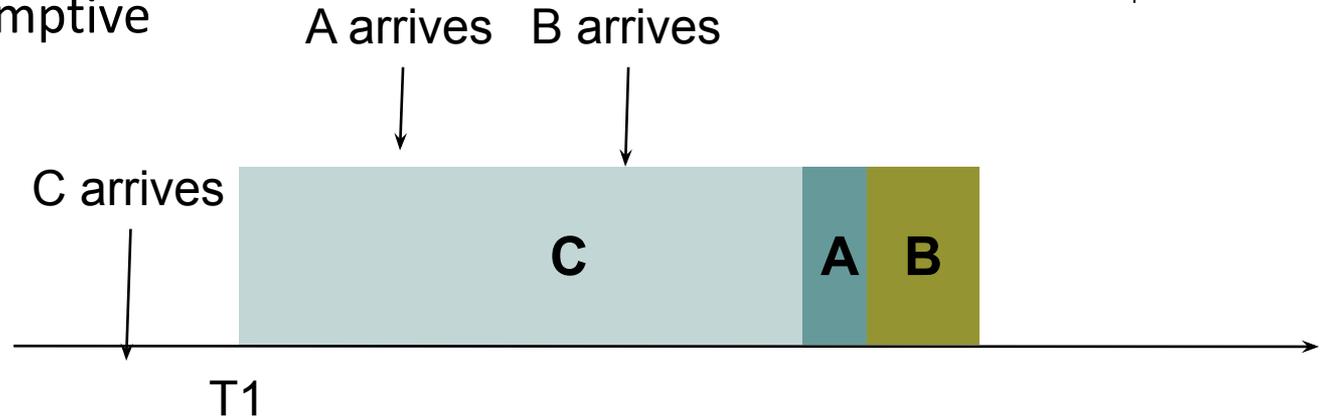
# Round Robin



- Each runs a time slice or quantum: Fair
- How do you choose time slice?
  - Overhead vs. response time
  - Overhead is typically about 1% or less
  - Quantum typically between 10 ~ 100 millisc

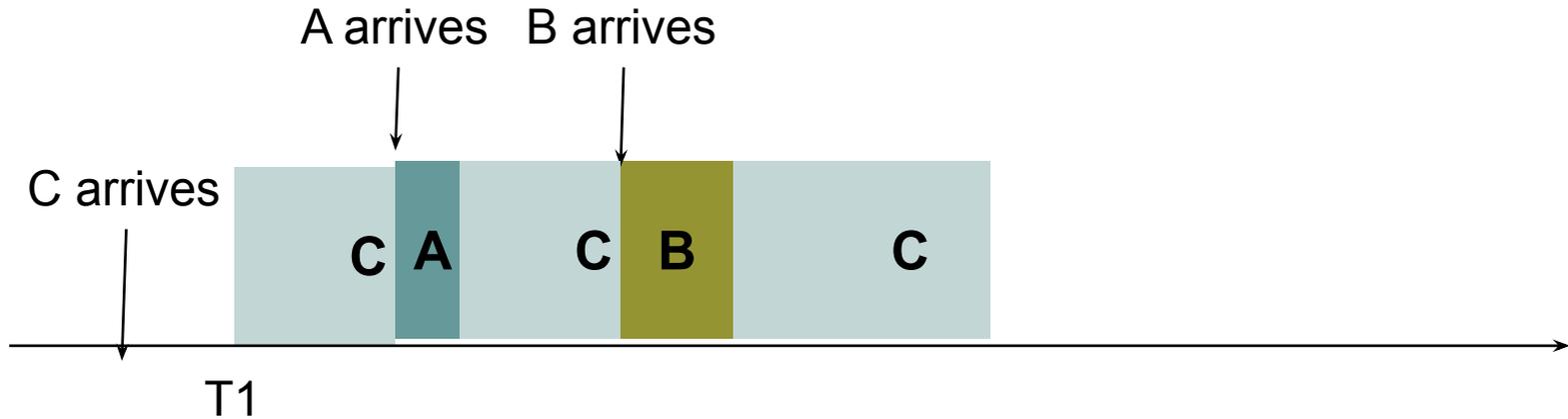
# STCF

- Shortest time to completion first (shortest job first)
  - Non-preemptive



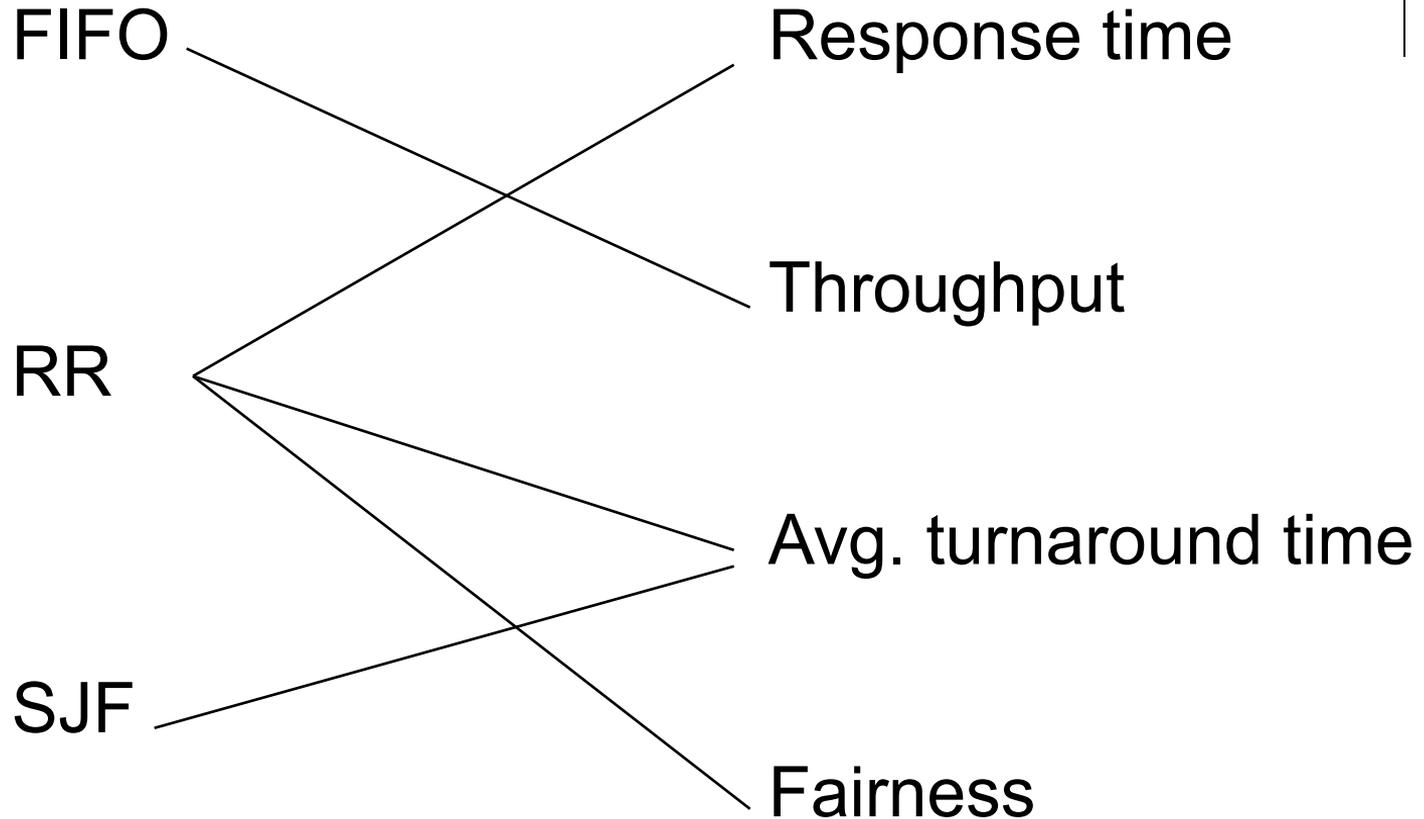
# SRTCFCF

- Shortest remaining time to completion first
  - Preemptive



Any potential problems?  
- Can cause **starvation!**

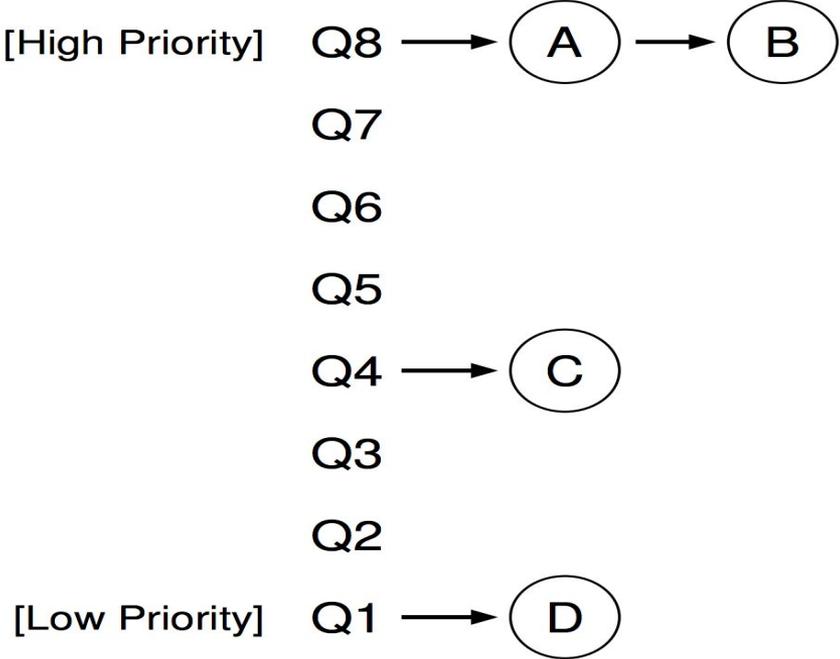
# Scheduling Policies Advantages



# Priority Scheduling

- To accommodate the spirits of SJF/RR/FIFO
- The method
  - Assign each process a *priority*
  - Run the process with highest priority in ready queue first
    - Use FIFO for processes with equal priority
  - Adjust priority dynamically
    - To deal with *all* issues: e.g. aging, I/O wait raises priority

# Multiple Queue Scheduling



# Multilevel Feedback Queue (MLFQ)

- Problem: how to change priority?
- Jobs start at highest priority queue
- Feedback
  - Priority Decreases: If a job uses up an entire time slice while running, its priority is reduced (i.e., it moves down one queue).
  - Priority Unchanged: If a job gives up the CPU before the time slice is up, it stays at the same priority level.
  - Priority Increases: After a long time period, move all the jobs in the system to the topmost queue (aging)