# Processes

Abhilash Jindal

# Agenda

- Vocabulary (OSTEP Ch. 4)
  - What is a process? System calls? Scheduler? Address space?
- Memory management (OSTEP Ch. 13-17)
  - How to manage and isolate memory? What are memory APIs? How are they implemented?
- Processes in action (xv6 Ch. 3: system calls, x86 protection, trap handlers)
  - Process control block, user stack<>kernel stack, sys call handling
- Scheduling (xv6 Ch5: context switching, OSTEP Ch. 6-9)
  - Response time, throughput, fairness

# Process is a running program

- Load program from disk to memory
  - Exactly how we loaded OS
- Give control to the process. Jump cs, eip

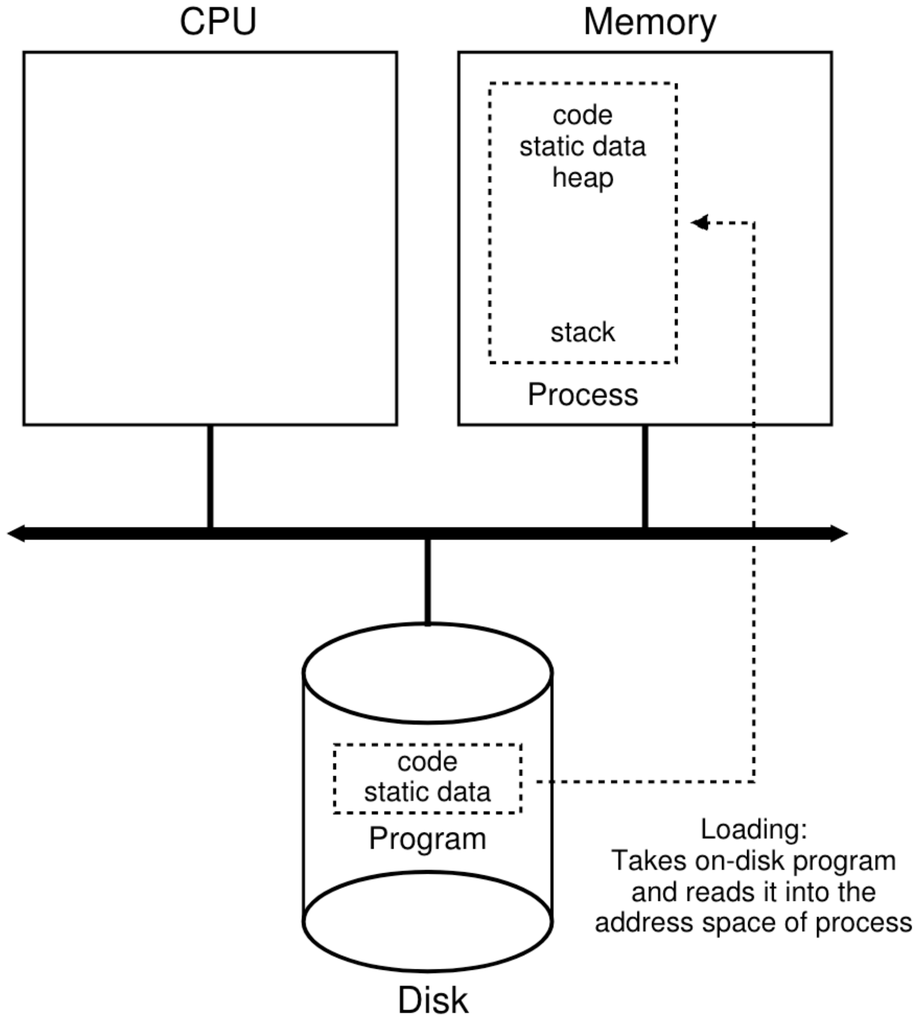



Figure 4.1: **Loading: From Program To Process**

# Processes can ask OS to do work for them

## System calls

```
$ strace cat /tmp/foo
…
openat(AT_FDCWD, "/tmp/foo", O_RDONLY)  = 3
read(3, "hi\n", 131072)                 = 3
write(1, "hi\n", 3)                     = 3
…
```

# OS maintains process states

## Scheduler switches between processes

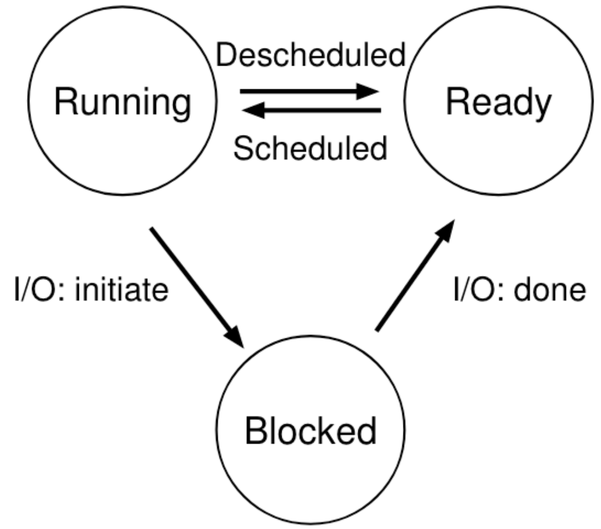



Figure 4.2: **Process: State Transitions**

| Time | $\text{Process}_0$ | $\text{Process}_1$ | Notes |
|---|---|---|---|
| 1 | Running | Ready | |
| 2 | Running | Ready | |
| 3 | Running | Ready | $\text{Process}_0$ initiates I/O |
| 4 | Blocked | Running | $\text{Process}_0$ is blocked, |
| 5 | Blocked | Running | so $\text{Process}_1$ runs |
| 6 | Blocked | Running | |
| 7 | Ready | Running | I/O done |
| 8 | Ready | Running | $\text{Process}_1$ now done |
| 9 | Running | – | |
| 10 | Running | – | $\text{Process}_0$ now done |

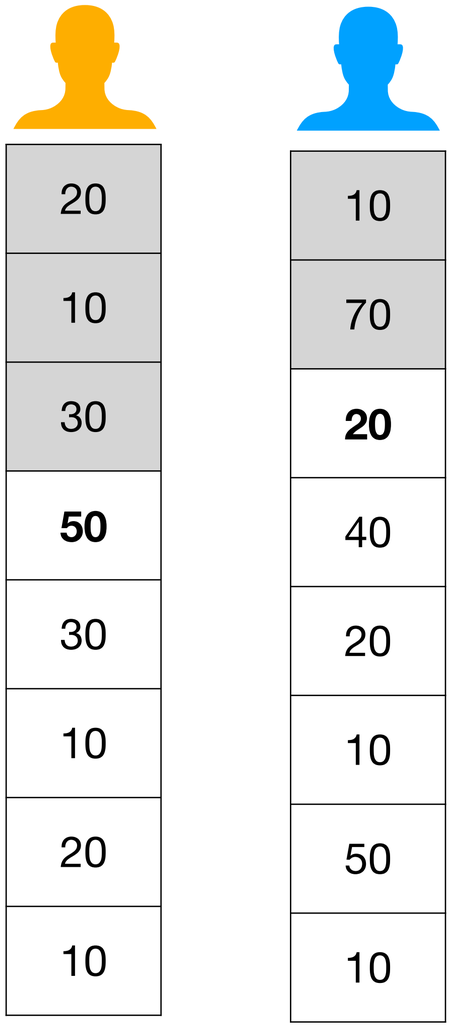
Figure 4.4: **Tracing Process State: CPU and I/O**

# Calculator analogy: Computing long sum

| |
|---|
| 20 |
| 10 |
| 30 |
| **50** |
| 30 |
| 10 |
| 20 |
| 10 |

- 2 0 = (move pointer to 10)
- + 1 0 = (move pointer to 30)
- + 3 0 = (move pointer to 50)
- + 5 0 = (move pointer to 30)
- + 3 0 = (move pointer to 10)
- + 1 0 = (move pointer to 20)
- + 2 0 = (move pointer to 10)

# Sharing the calculator



| | |
|---|---|
| 20 | 10 |
| 10 | 70 |
| 30 | **20** |
| **50** | 40 |
| 30 | 20 |
| 10 | 10 |
| 20 | 50 |
| 10 | 10 |

- Steps to share the calculator:
  - 20 + 10 = 30 + 30 = 60
  - Write 60 in notebook, remember that we were done till 30, give calculator
  - 10 + 70 = 80
  - Write 80 in notebook, remember that we were done till 70, give the calculator back

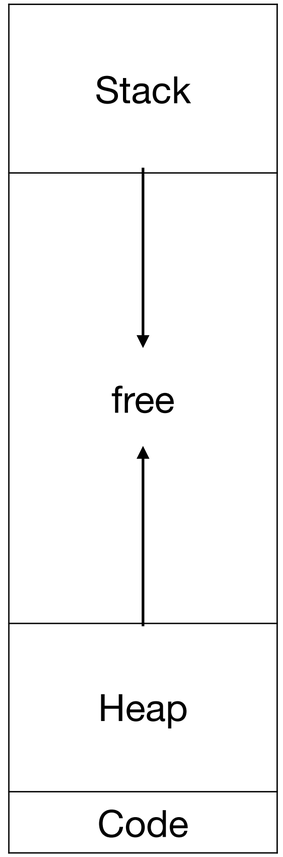
# Memory isolation and management

## OSTEP Ch. 13-17

Abhilash Jindal

# Process Address Space

## Code, Heap, Stack




**Process address space**

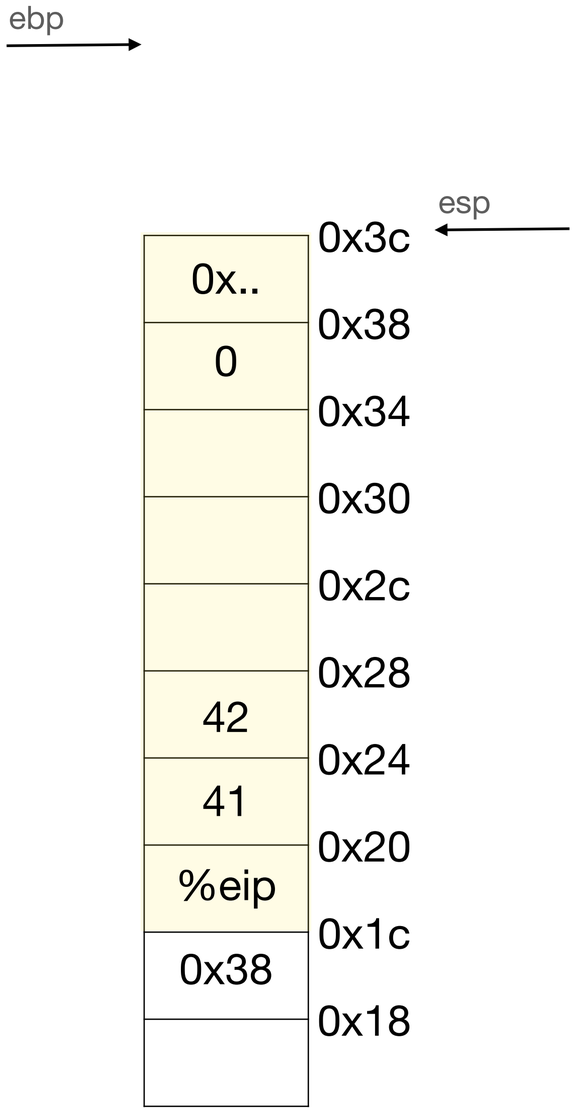
# Function calling in action

## Stack

**02.s**

```
_foo:
  pushl %ebp                Save caller's base pointer
  movl  %esp, %ebp          ebp = esp
  movl  8(%ebp), %eax       eax = *(ebp + 8)
  addl  12(%ebp), %eax      eax = eax + *(ebp + 12)
  popl  %ebp                Restore caller's base pointer
  retl                      change eip to return address

  .globl  _main                         ## -- Begin function main
  .p2align  4, 0x90
_main:
  pushl %ebp                Save caller's base pointer
  movl  %esp, %ebp          ebp = esp
  subl  $24, %esp           esp = esp - 0x18
  movl  $0, -4(%ebp)        *(ebp-4)=0
  movl  $41, (%esp)         *(esp) = 41
  movl  $42, 4(%esp)        *(esp+4) = 42
  calll _foo                Push current eip on to stack, jump to foo
  addl  $24, %esp           esp = esp + 24 (Restore caller's esp)
  popl  %ebp                Restore caller's ebp
  retl
```

eip →

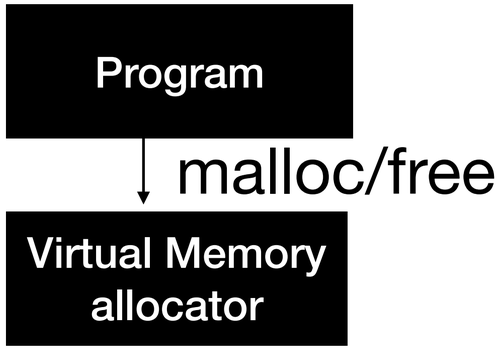


# Memory APIs and bugs

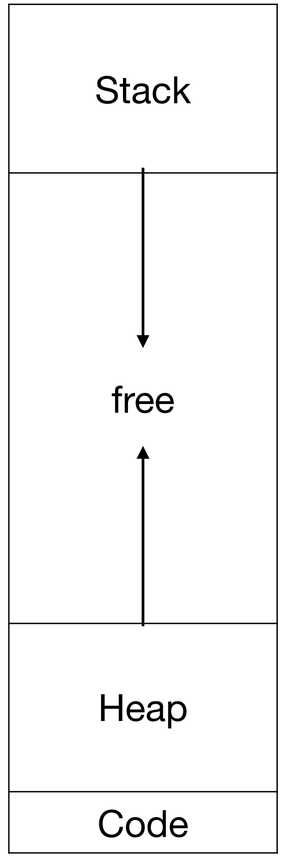
- malloc, free. va.c.
  - malloc is for dynamic allocation. Size is not known at compile time. Slower than stack allocations. Need to find free space.
- Null pointer dereference. null.c
- Memory leak. leak.c
- Buffer overflow. overflow.c
- Use after free. useafterfree.c
- Invalid free. invalidfree.c
- Double free. doublefree.c
- Uninitialised read. uninitread.c

# Memory allocator

## Works with virtual memory



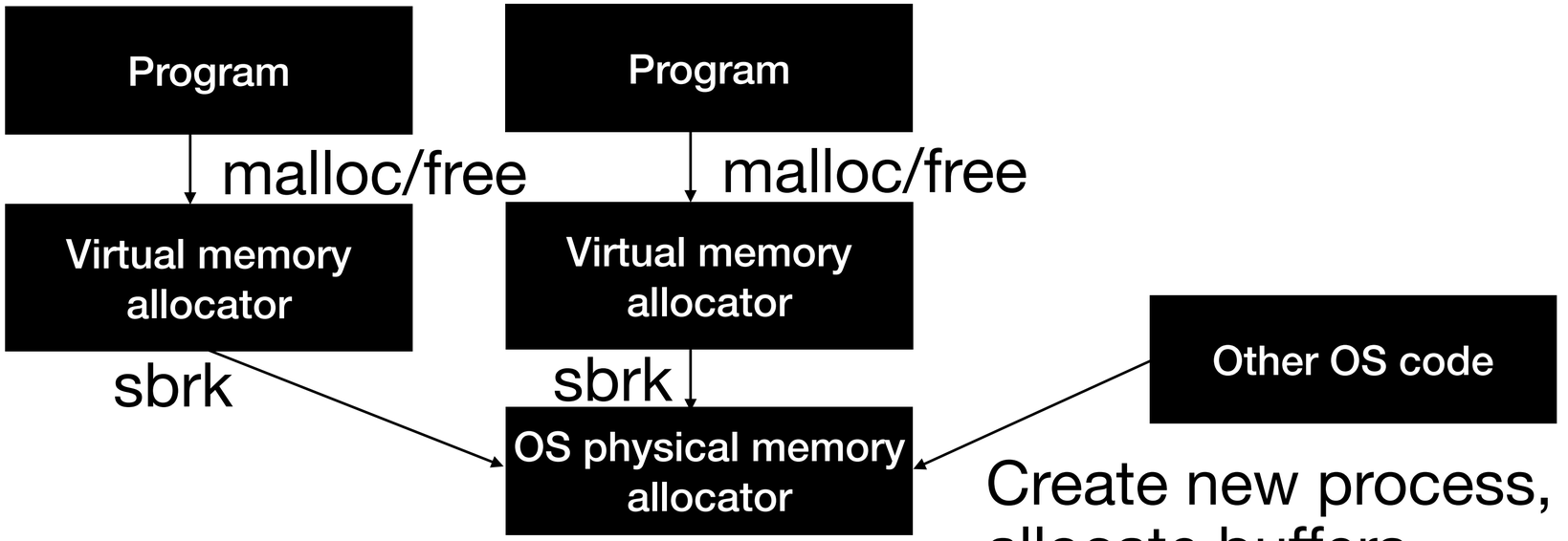

Manages heap memory

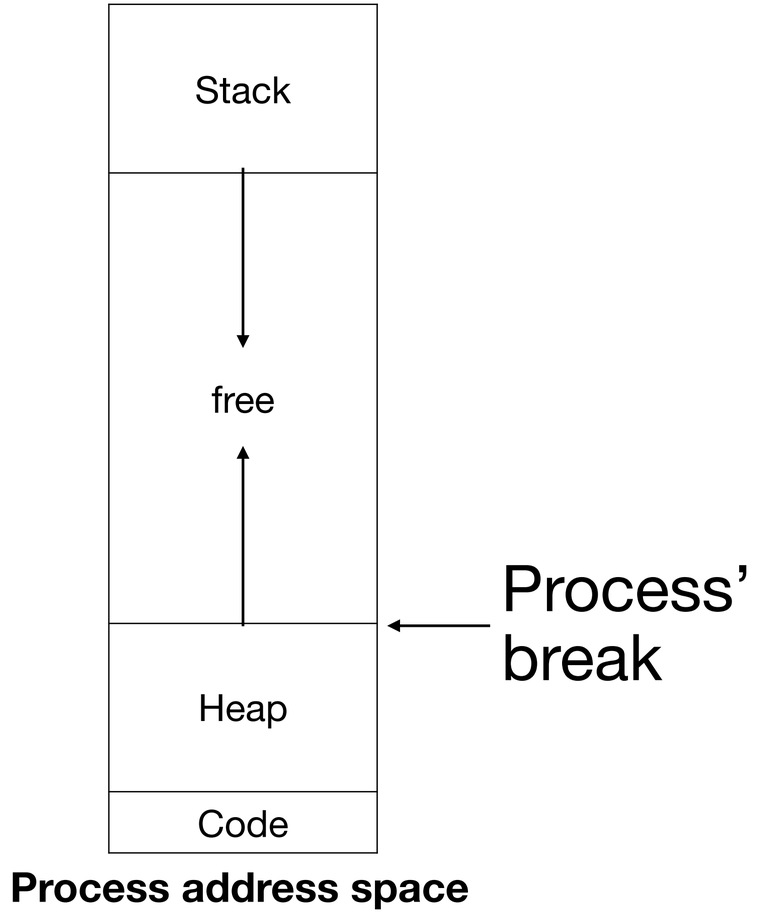



**Process address space**

# OS memory allocator

- sbrk(int *increment*) increments process' break. *increment* can be negative.



# Memory allocation

```
ptr=malloc(size_t size);

free(ptr);
```

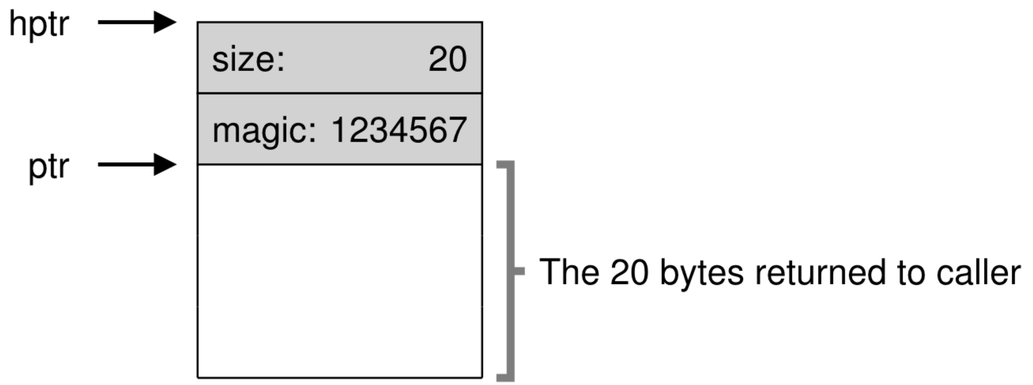
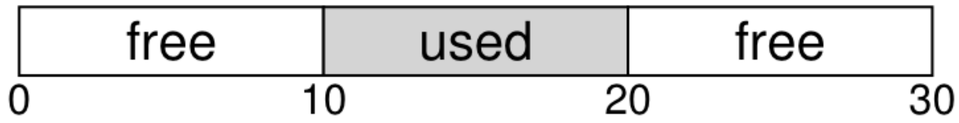



Figure 17.2: **Specific Contents Of The Header**

Stack allocations are faster than heap allocations. No need to find space.

# Memory allocator




Fragmented heap over time

- Assumptions
  - Do not apriori know allocation size and order
  - Cannot move memory once it is allocated. Program might have the pointer to it.
- Goals
  - Quickly satisfy variable-sized memory allocation requests. How to track free memory?
  - Minimize fragmentation

# Memory (de)allocation patterns

- Small mallocs can be frequent. Large mallocs are usually infrequent.
  - After malloc, program will initialise the memory area.
- “Clustered deaths”: Objects allocated together die together.

# Free list splitting and coalescing

```
ptr = malloc(100)

sptr = malloc(100)

optr = malloc(100)

free(sptr)

free(ptr)

free(optr)
```

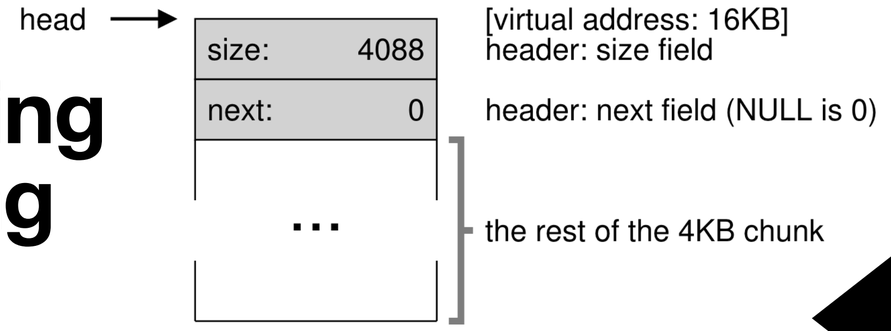



Figure 17.3: **A Heap With One Free Chunk**

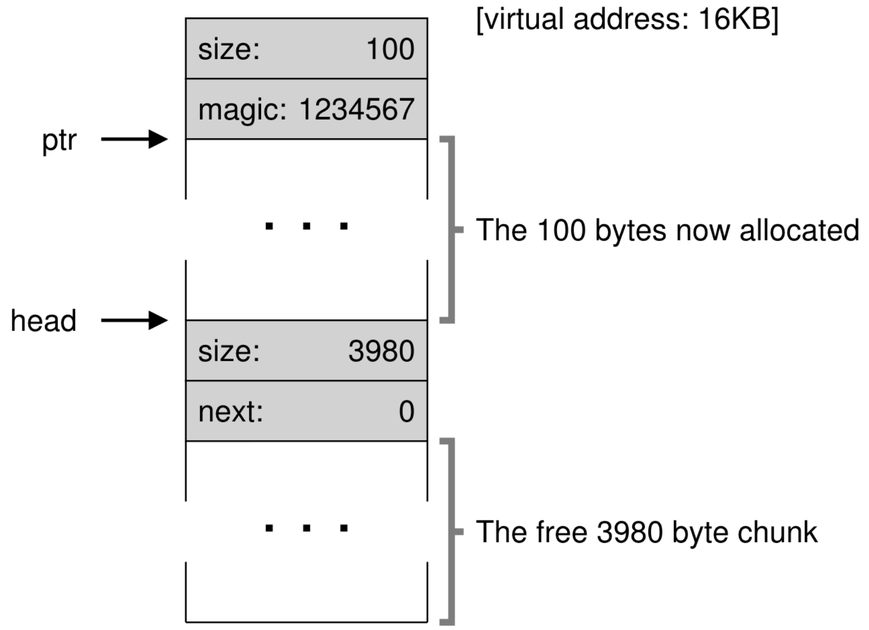



Figure 17.4: **A Heap: After One Allocation**

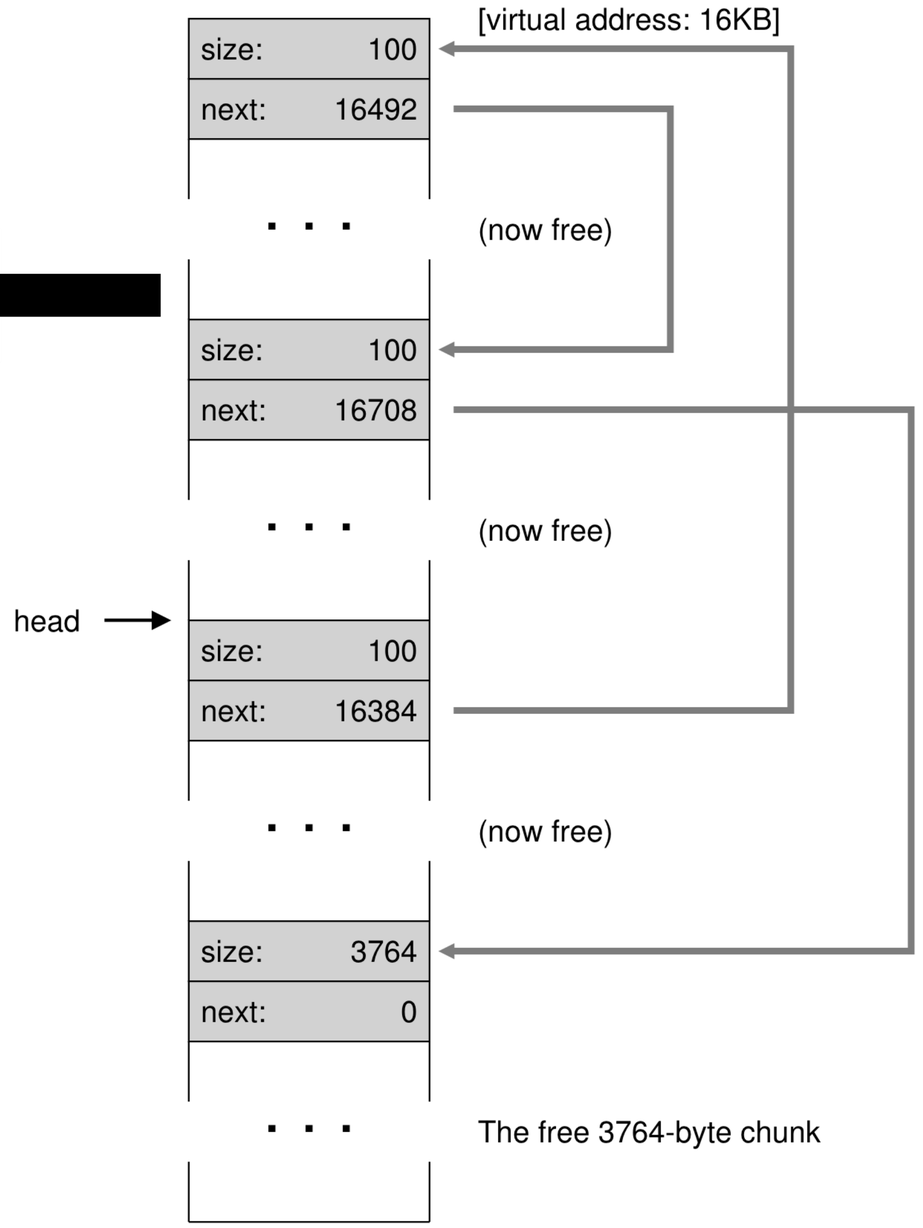



Figure 17.7: **A Non-Coalesced Free List**

# Which block to allocate?

## Example: malloc(15)

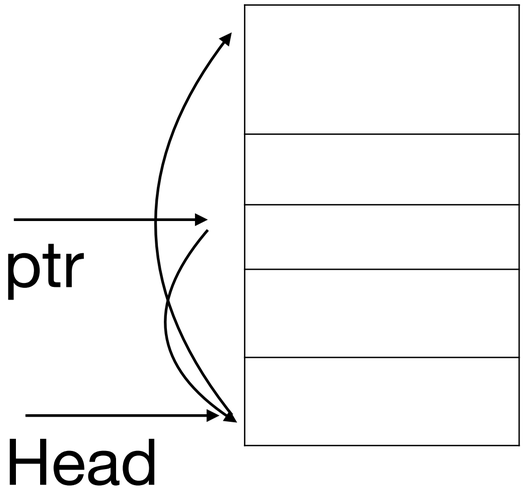


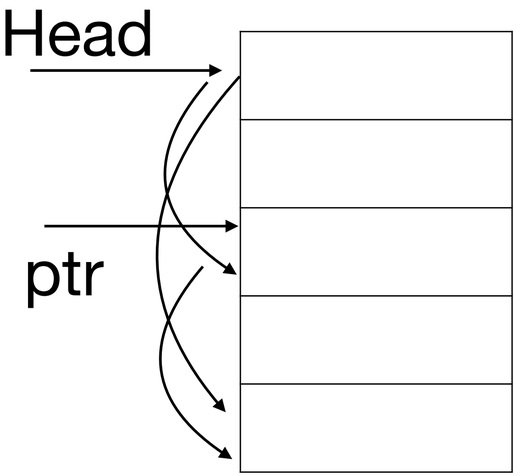

- Best fit
  - Slow. need to search the whole list
- First fit
  - Faster. (xv6: umalloc.c)
- Fragmentation
  - Example: malloc(25)

# In which order to maintain lists?

- (De)allocation order

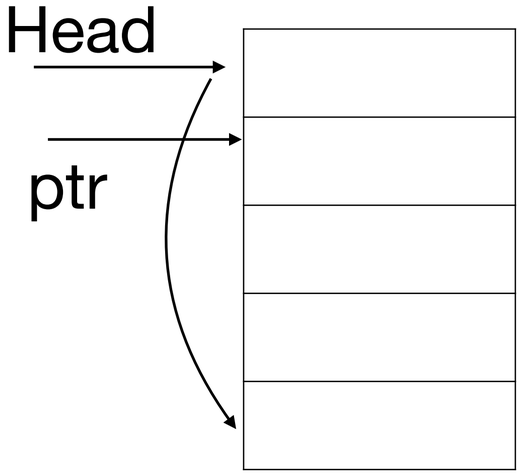
- Address order
  - Slow frees: need to traverse the free list
  - (xv6: umalloc.c)
  - Address order, first fit will allocate back-to-back allocations contiguously.
    - Due to “clustered deaths”, we may get better chances of coalescing



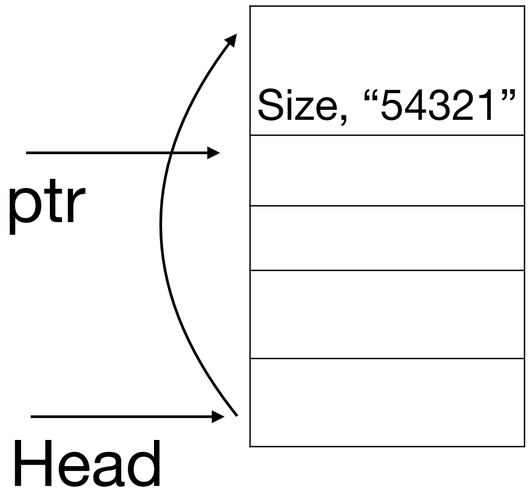



# How to do coalescing?

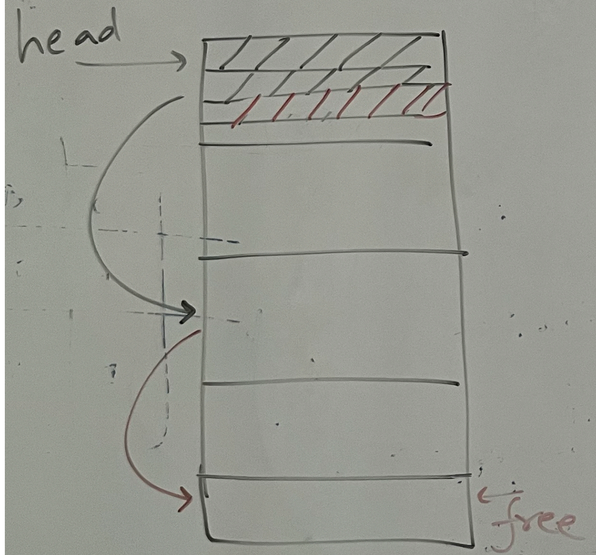
## Example: free(ptr)



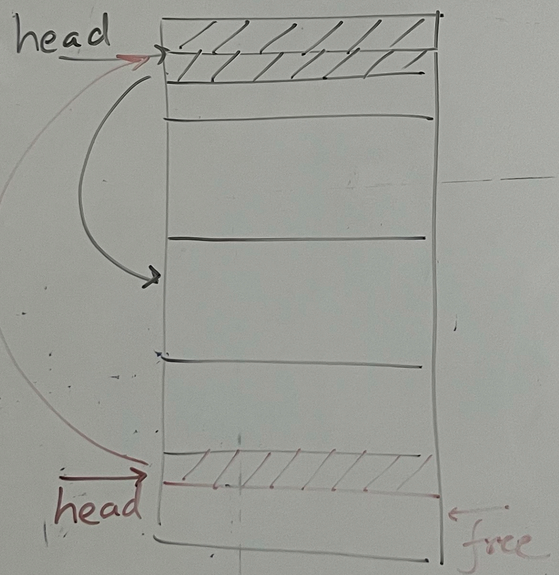

- Straightforward in address order since we are traversing the free list in address order
- In deallocation order: when an area is freed, check if the “boundary tag” is present in the footer above
- First fit and address order
  - Better chances of coalescing for clustered deaths, simpler coalescing (no boundary tag)
  - First fit causes fragmentation
  - Address order slows frees due to traversing free list



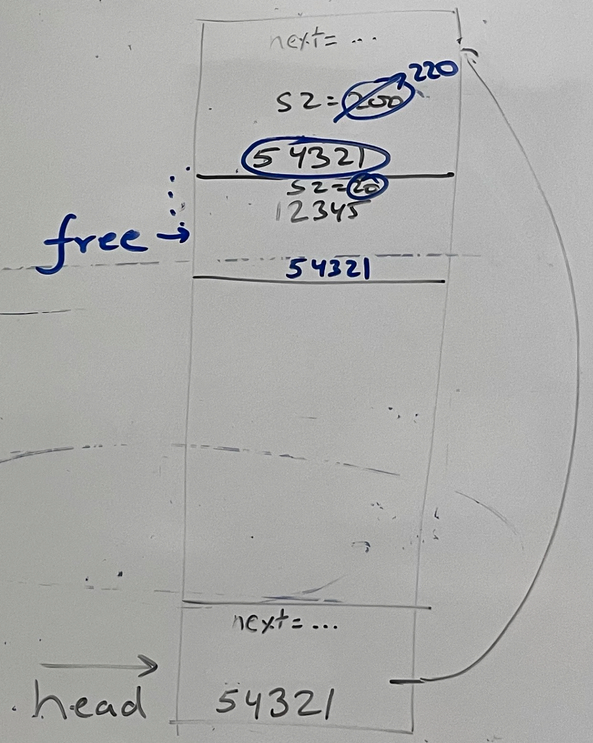
First fit
address order




First fit
(de) allocation order

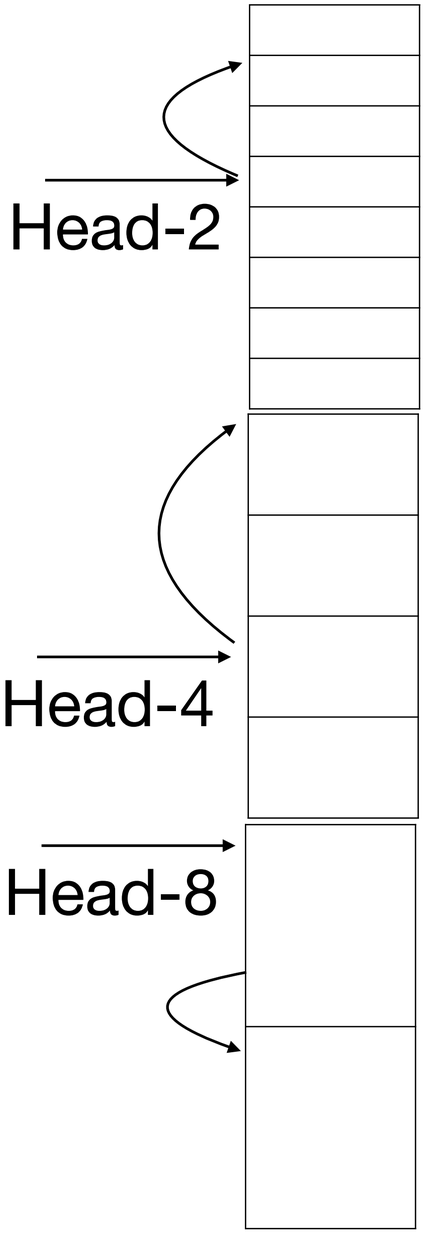



Boundary tag



# Segregated lists

- Separate lists for each size.
- “Segregated fit”: First fit in the smallest object list that can fit the object. Approximates best fit.
  - No splitting, coalescing
- Internal fragmentation: allocates more than asked
- Wastes memory: If no allocation from object size, have unnecessary reserved space for it



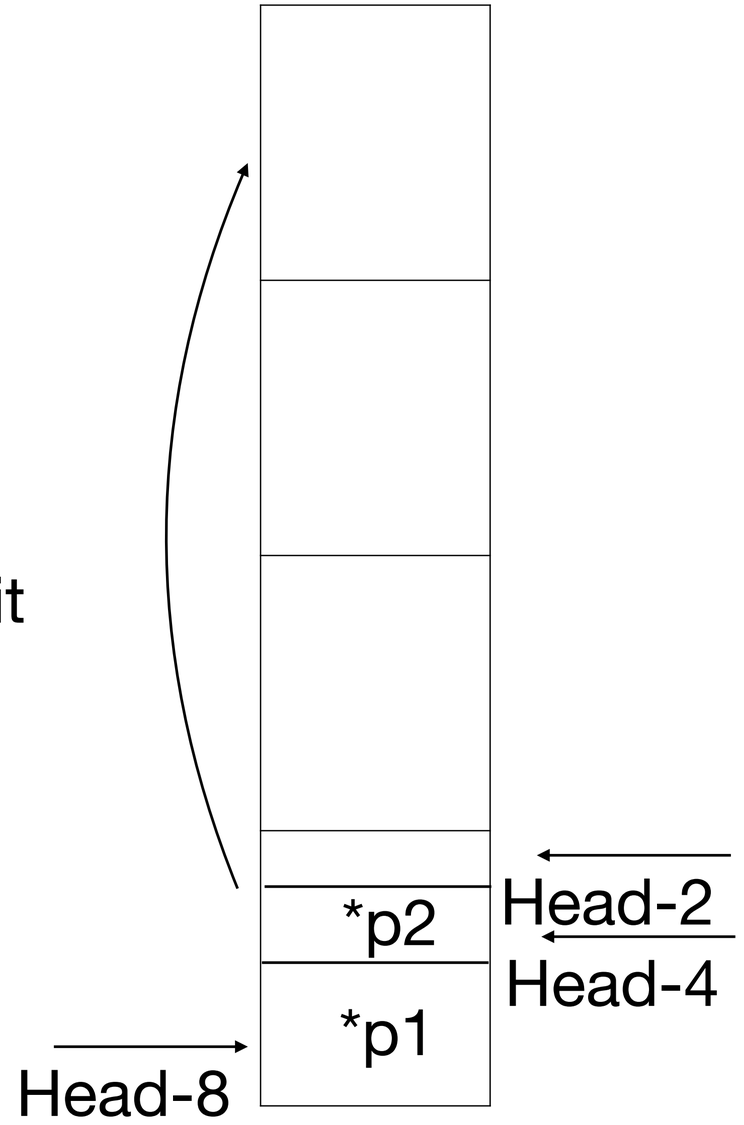
# Slab allocator

## Used in Linux Kernel

- Knows the size of allocations made by the Linux kernel. Keep segregated lists for each such struct.
  - No internal fragmentation: Exactly the size of the struct
  - Hierarchical allocator: Return unused lists back to global allocator

# Buddy allocator

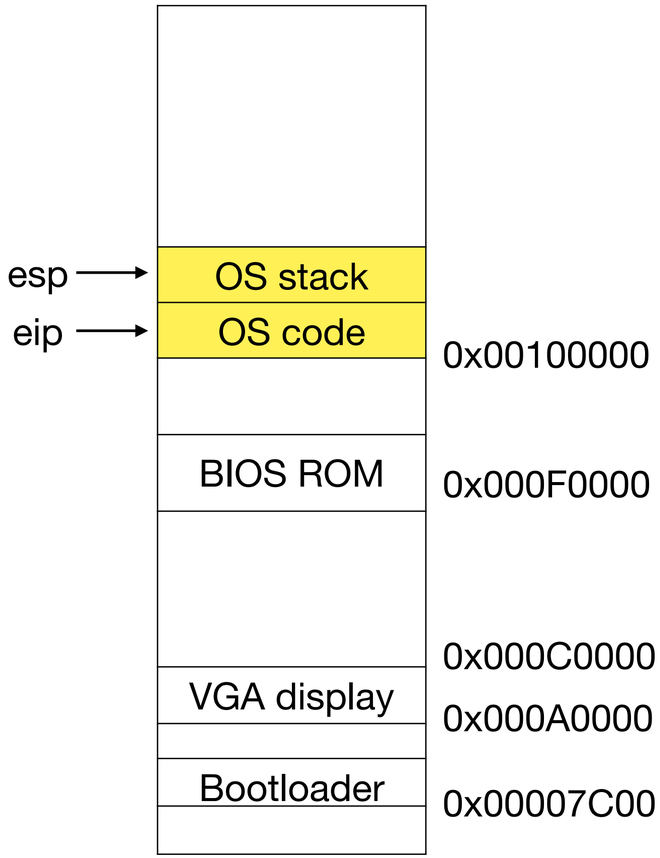
- Example:
  - p1 = malloc(3)
  - p2 = malloc(2)
- First fit with segregated lists approximates best fit
- Straightforward splitting and coalescing
- Deallocation order: fast frees
- Used in Linux kernel

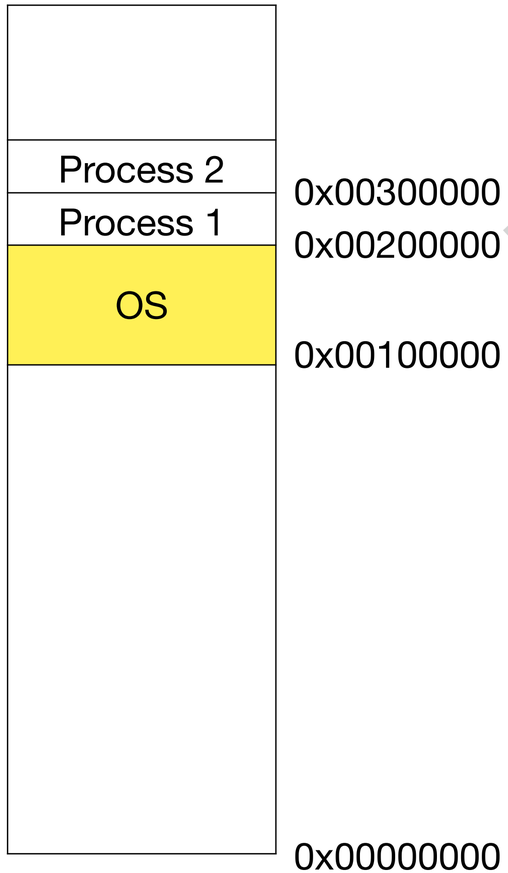


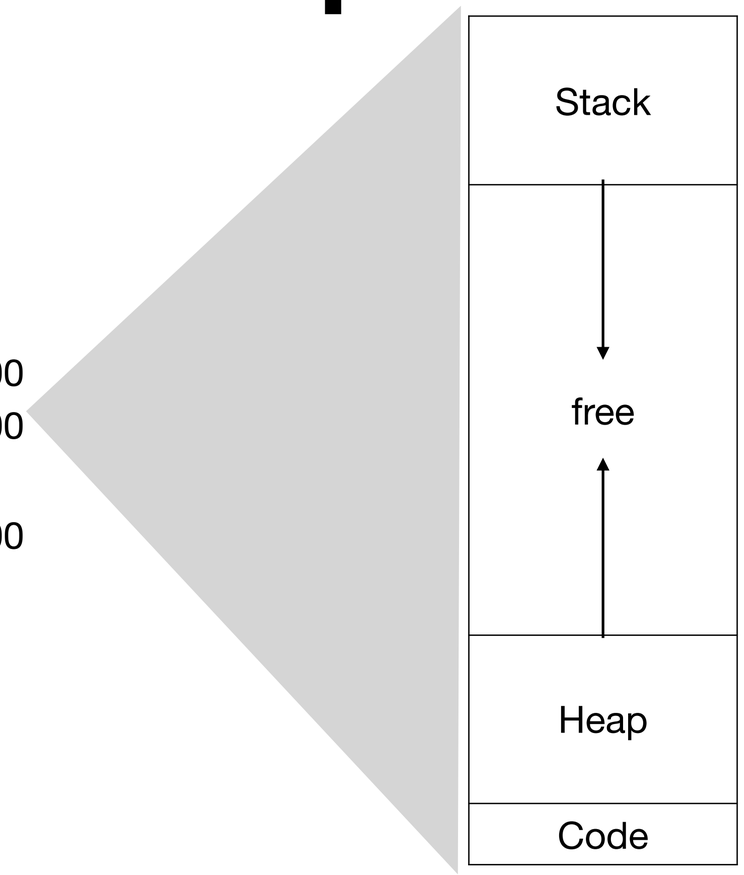
# Aside: automatic memory management

- “Higher level” languages do not expose raw pointers to programmers. Example: Java, Python, Go
  - The language runtime manages memory. Programmer need not call free.
  - Largely prevents memory leaks, dangling references, double free, null pointer dereference
  - Mark and sweep garbage collector, reference counting based garbage collector
  - Copying GC can do compaction to defragment heap. Will rewrite pointers.
  - Can incur heavy performance penalty

# Memory isolation and address space



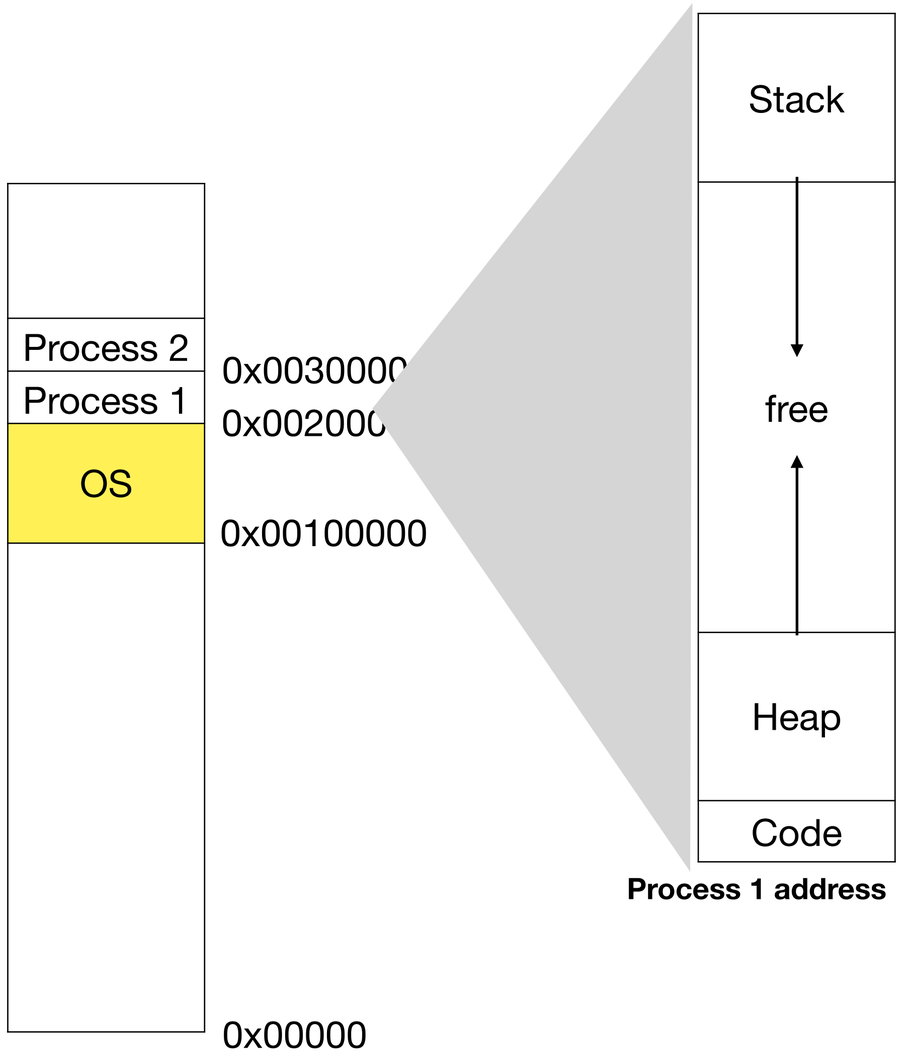



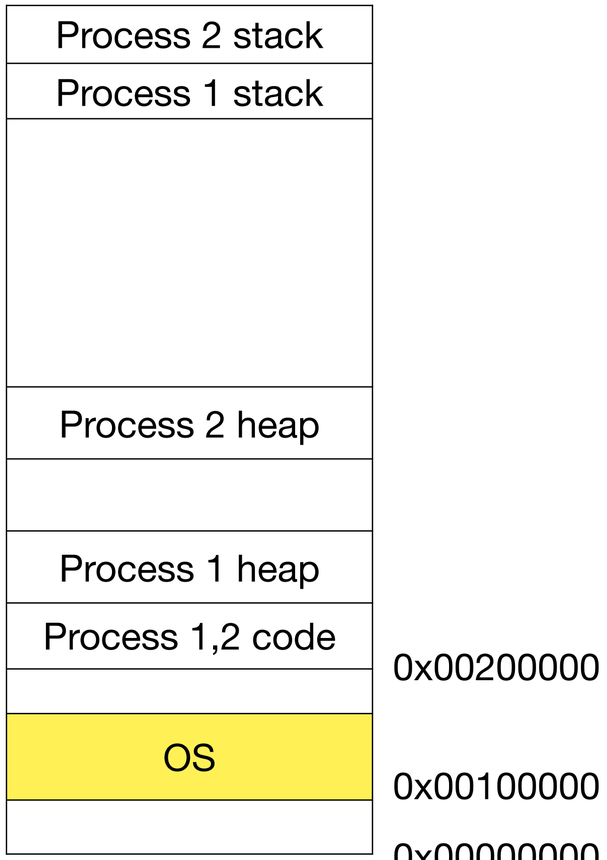

**Process 1 address space**

Due to address translation, compiler need not worry where the program will be loaded!

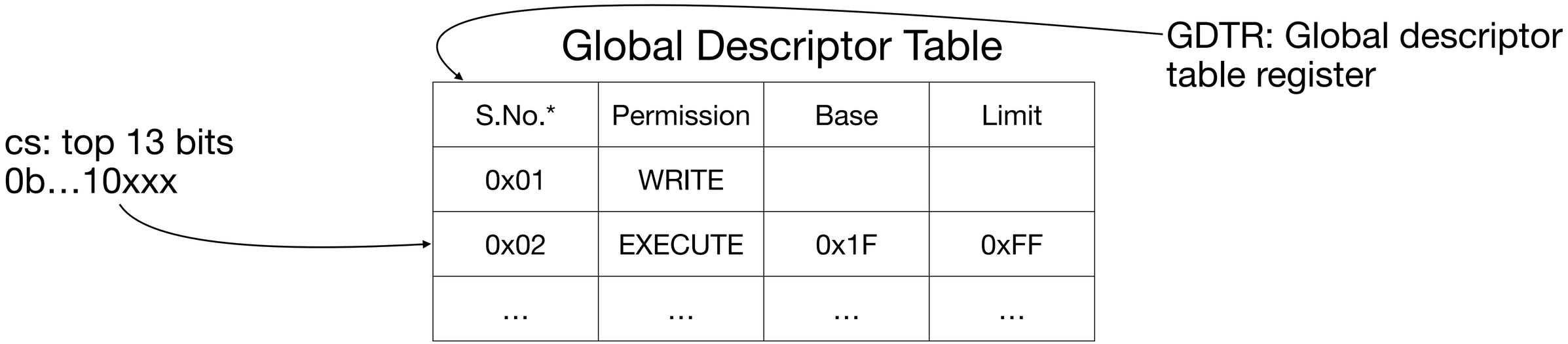
# Segmentation




- Place each segment independently to not map free space
- Share code segments to save space
  - Mark non-writeable

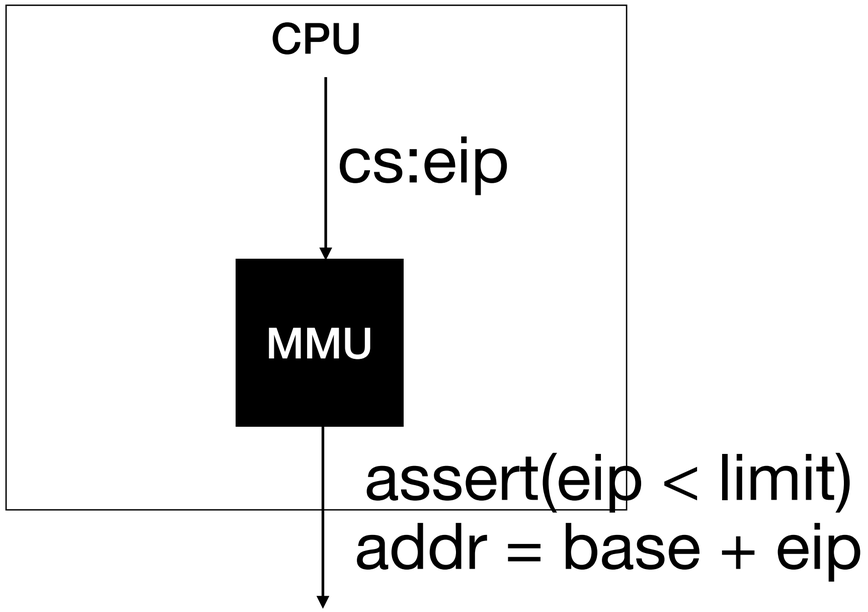


# Many segments can be initialised in GDT




*: S.No. added only for illustration

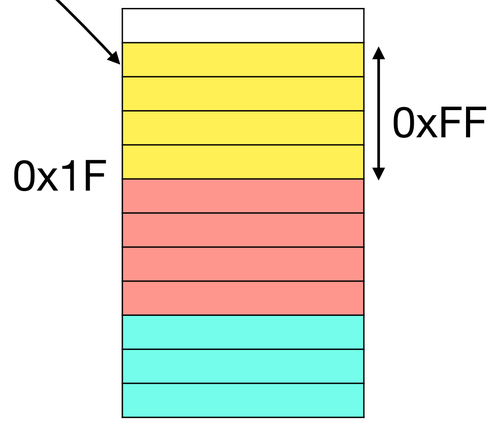
# Address translation




Global Descriptor Table

cs: top 13 bits
0b…10xxx

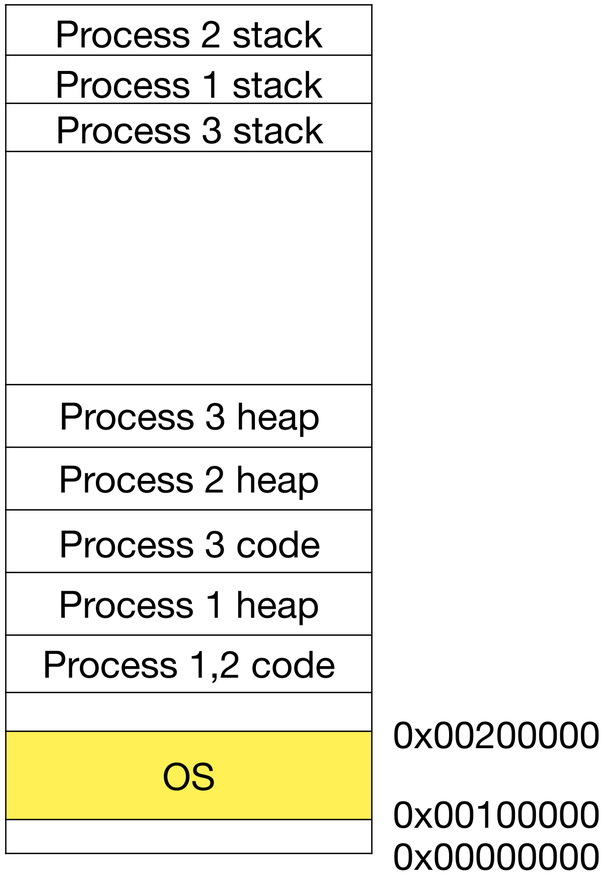
| S.No.* | Permission | Base | Limit |
|---|---|---|---|
| 0x01 | WRITE | | |
| 0x02 | EXECUTE | 0x1F | 0xFF |
| … | … | … | … |




- Can “protect” different segments from each other

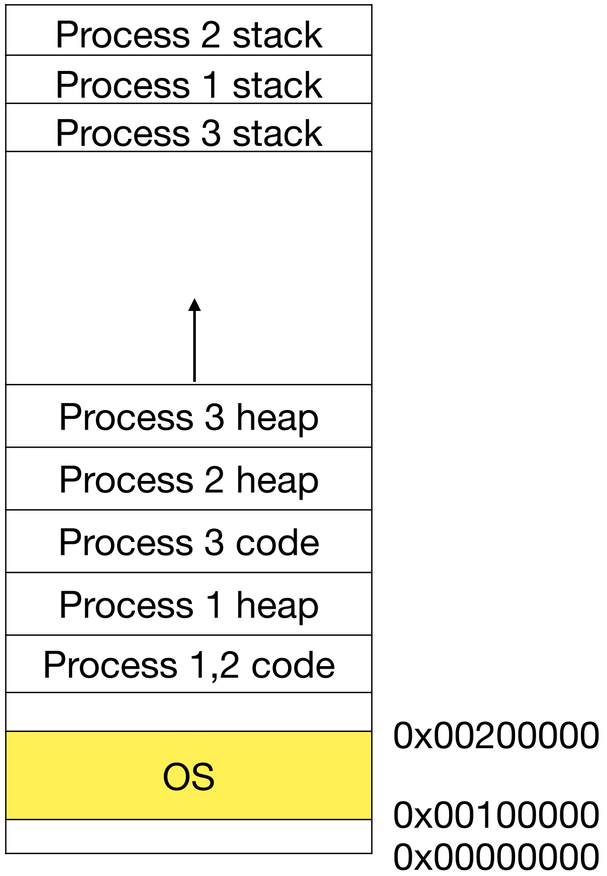
# Allocating memory to a new process

- Find free spaces in physical memory.
- Create new entries in GDT for the new process.



# Growing heap

- sbrk can grow heap segment



# External fragmentation

- After many processes start and exit, memory might become "fragmented" (similar to disk)
  - Example: cannot allocate 20 KB segment
- Compaction: copy all allocated regions contiguously, update segment base and bound registers
  - Copying is expensive
  - Growing heap becomes not possible

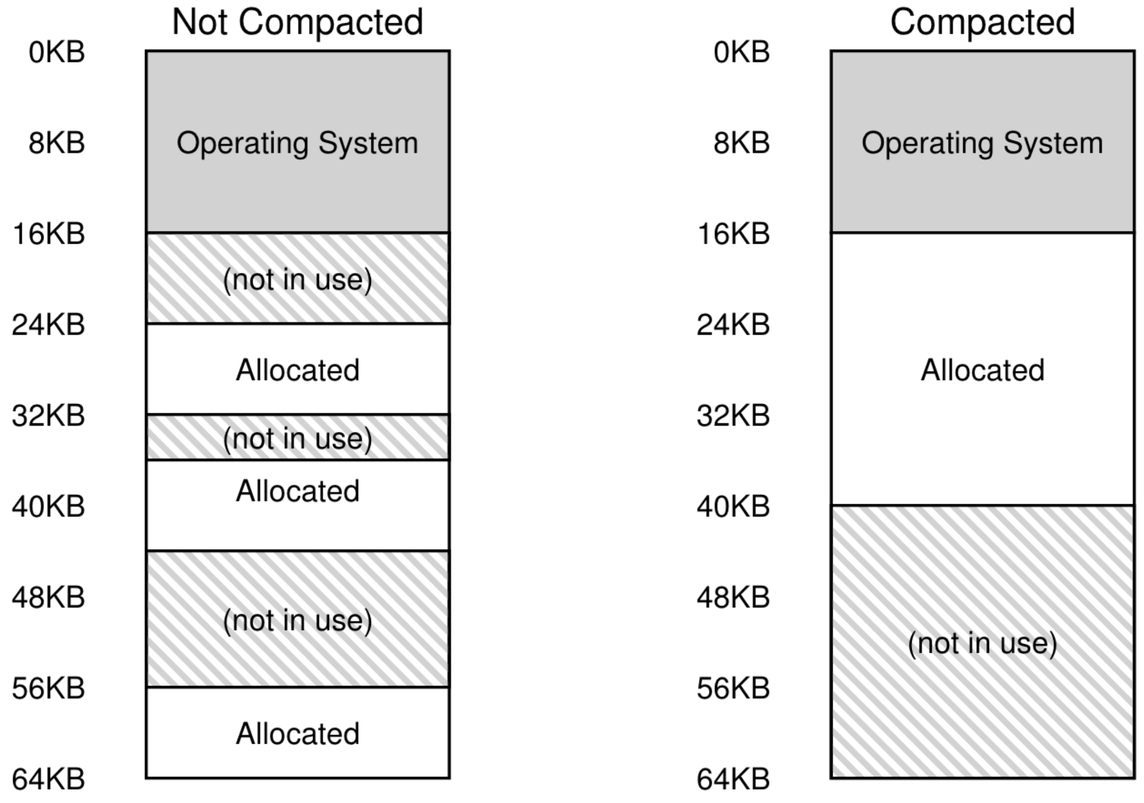
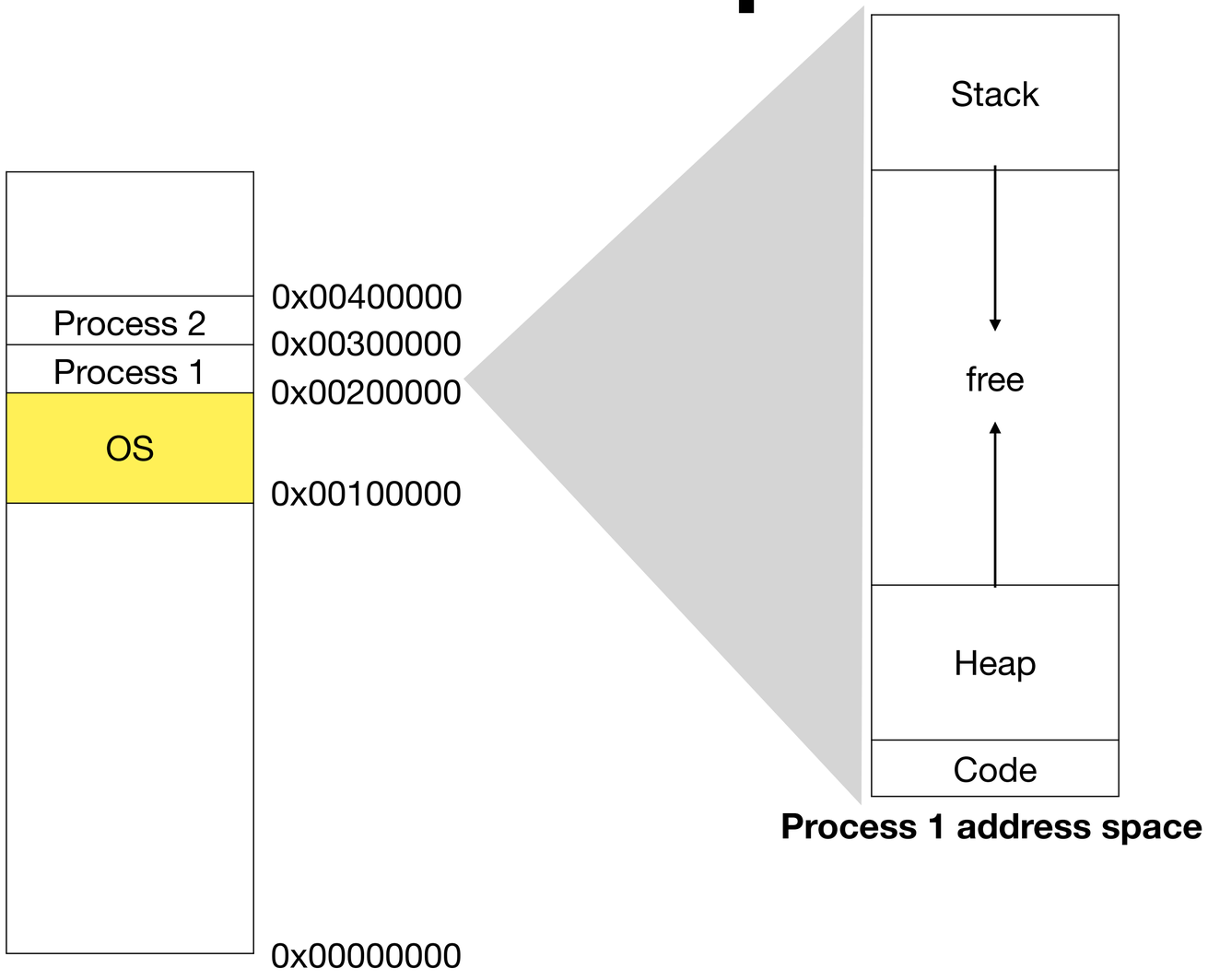



Figure 16.6: **Non-compacted and Compacted Memory**

# Processes in action

## xv6 Ch. 3: system calls, x86 protection, trap handlers

# Memory isolation and address space



# Protection

- Process cannot modify OS code since it is not in process' *address space*
- Process cannot modify GDT and IDT entries since GDT, IDT are not in its address space
- What stops process from calling lgdt and lidt instructions?
  - Ring 0: kernel mode. Ring 3: user mode
  - lgdt and lidt are *privileged instructions* only callable from "ring 0"

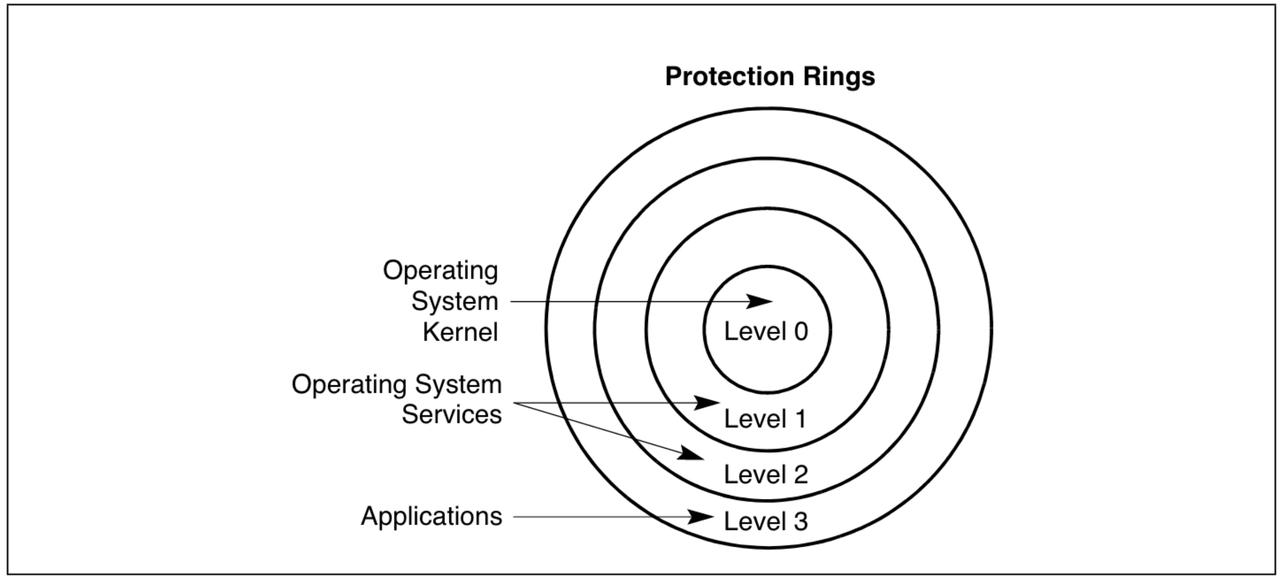
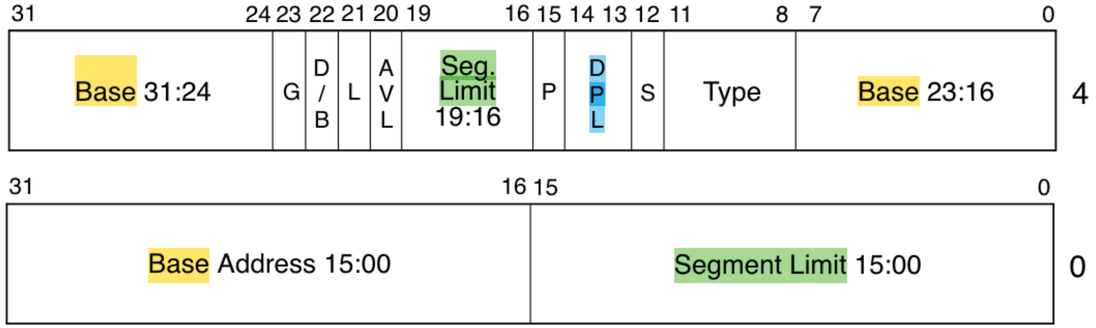



Figure 5-3. Protection Rings

# How does hardware know the current privilege level?

- Two LSBs of %cs determine “current privilege level” (CPL)
- Two LSBs of other segment selectors specify “required privilege level” (RPL)
- Segment descriptor specifies “descriptor privilege level” (DPL)




L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

Figure 3-8. Segment Descriptor

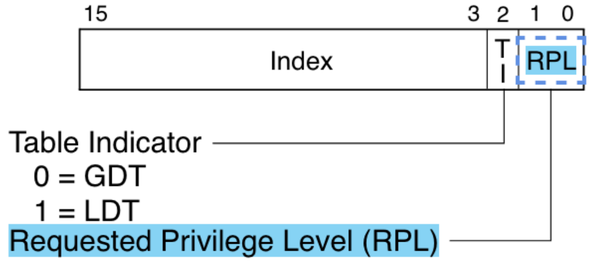



Figure 3-6. Segment Selector

# How does hardware enforce the current privilege level?

- Privileged instructions can only be called when CPL=0
- Programs can change their segment selectors
  - Programs cannot lower their CPL directly
  - INT instruction causes a software interrupt to set ring = 0
  - CPL <= DPL and RPL <= DPL

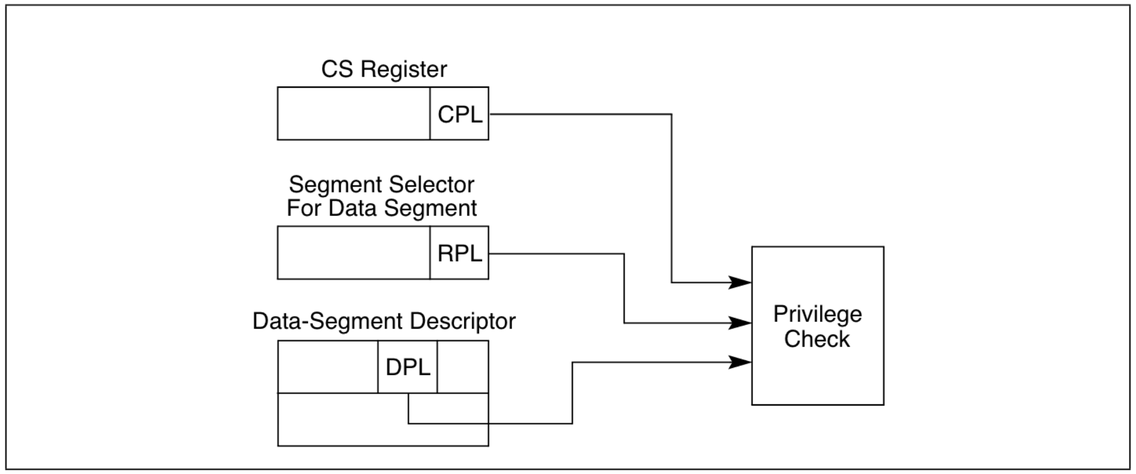



Figure 5-4. Privilege Check for Data Access

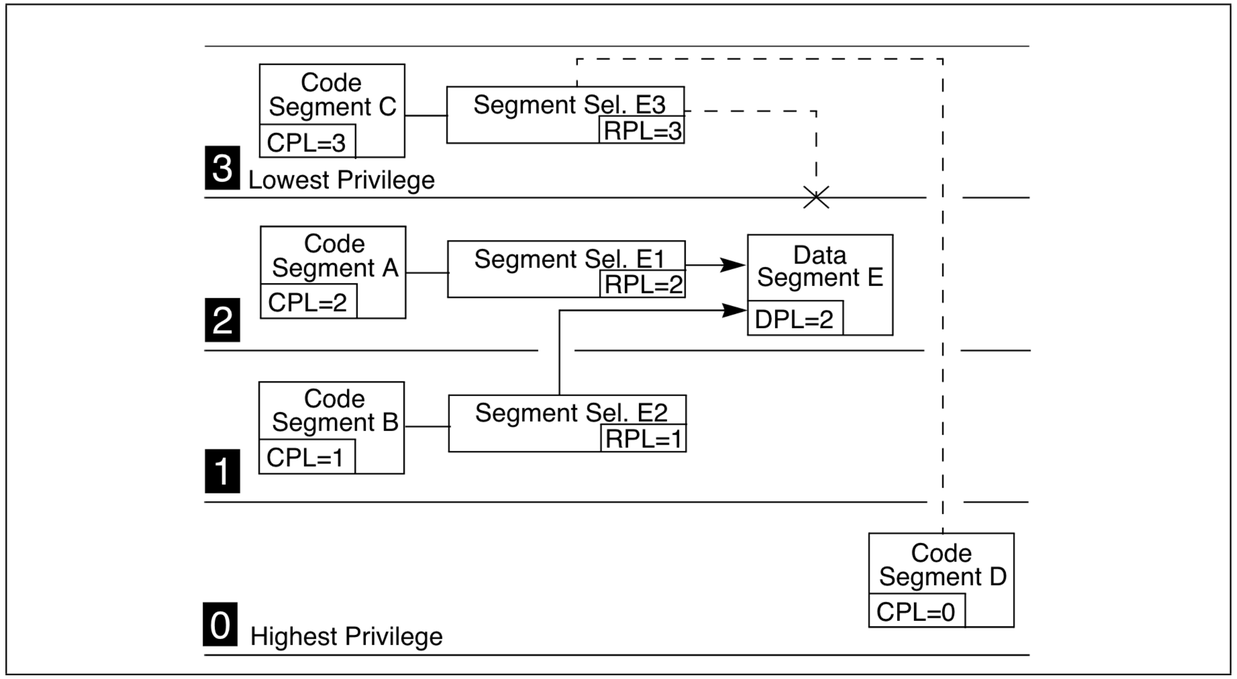
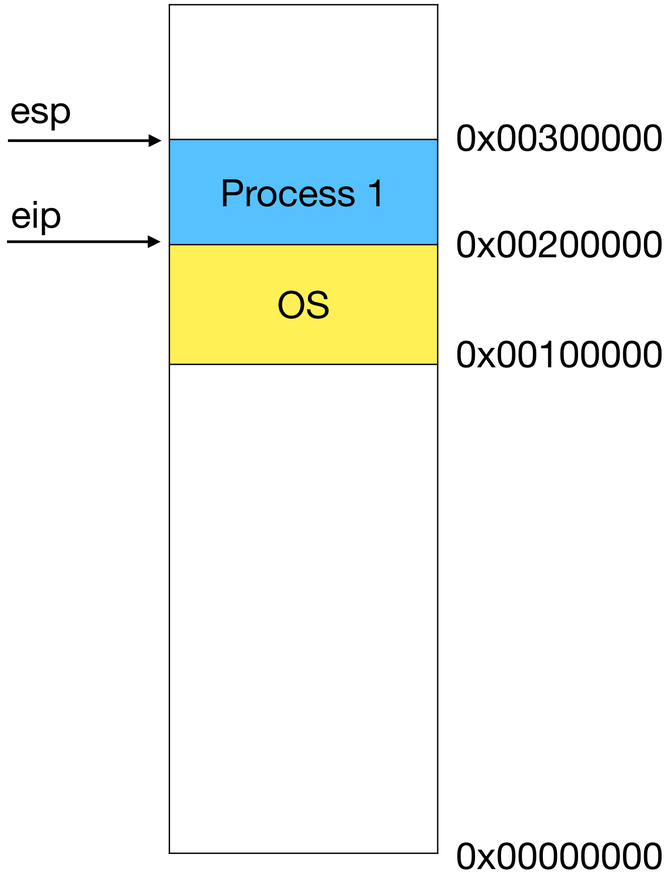



Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

# Setting up our first process in xv6!

**p15-initproc**

- main.c calls seginit, then pinit, then scheduler.
- seginit in vm.c creates code (UCODE) and data segments (UDATA) from STARTPROC (2MB) to 3MB. Flat memory model!
- pinit in proc.c copies program binary to STARTPROC and sets cs,ds,ss,es to the program's segments. Last two bits are set to DPL_USER. Sets eip=0, esp=1MB. Enables interrupt flag in eflags. proc.c maintains list of processes in ptable. We are just starting one process for now.
- scheduler in proc.c selects RUNNABLE process and switches to it.
- Process cannot change GDT entries, IDT entries since they are not in address space of the process! Process cannot call lgdt, lidt since it will run in ring 3.

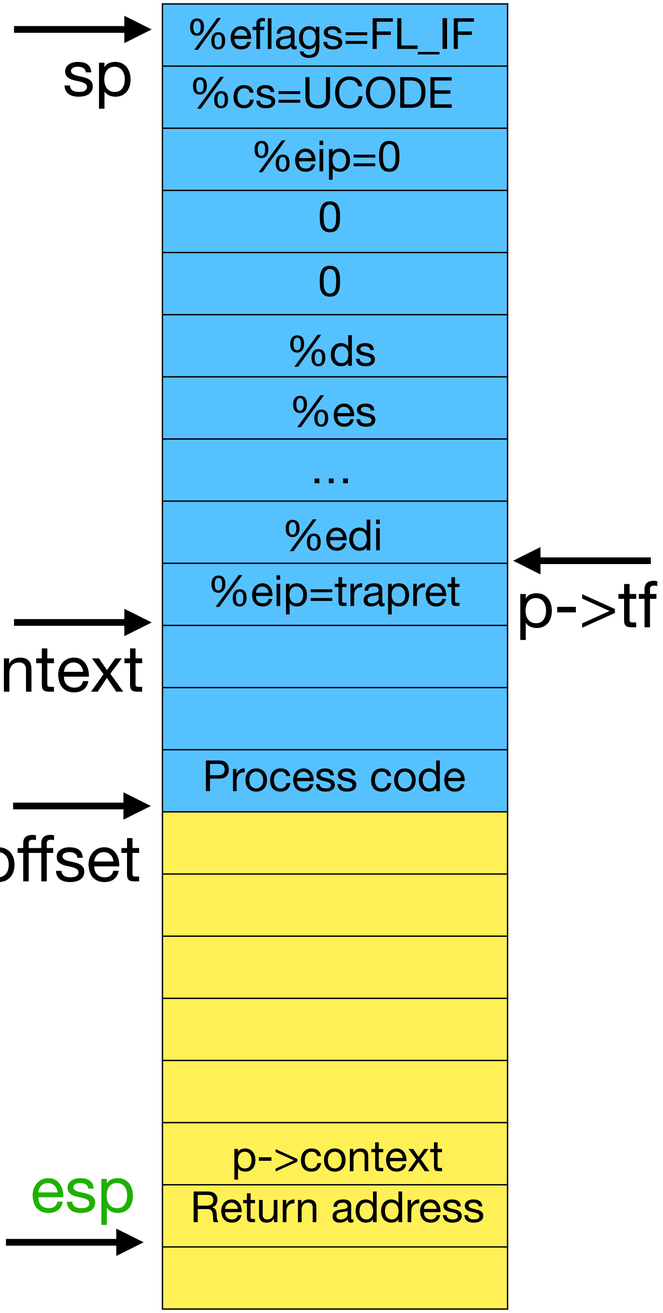


# Understanding swtch

```
pinit(){
  p = allocproc();
  memmove(p->offset, _binary_initcode_start,);
  p->tf->ds,es,ss = (SEG_UDATA<<3) | DPL_USR;
  p->tf->cs = (SEG_UCODE<<3) | DPL_USR;
  p->tf->eflags = FL_IF;
  p->tf->eip = 0;
}

allocproc() {
  sp = (char*)(STARTPROC + (PROCSIZE<<12));
  sp -= sizeof *p->tf;
  p->tf = (struct trapframe*)sp;
  sp -= sizeof *p->context;
  p->context = (struct context*)sp;
  p->context->eip = (uint)trapret;
  return p;
}
```

eip

```
scheduler() {
  …
  swtch(p->context);
}
swtch:
  movl 4(%esp), %eax
  movl %eax, %esp
  movl $0, %eax
  ret

.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp
  iret
```



# Interrupt handling revisited

eip

```
for(;;)
;
```

**vectors.S**

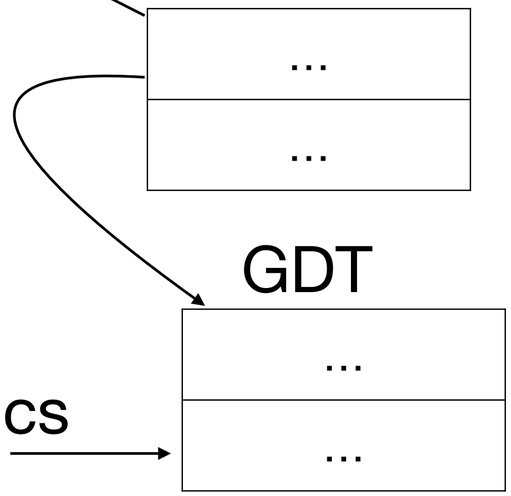
```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```

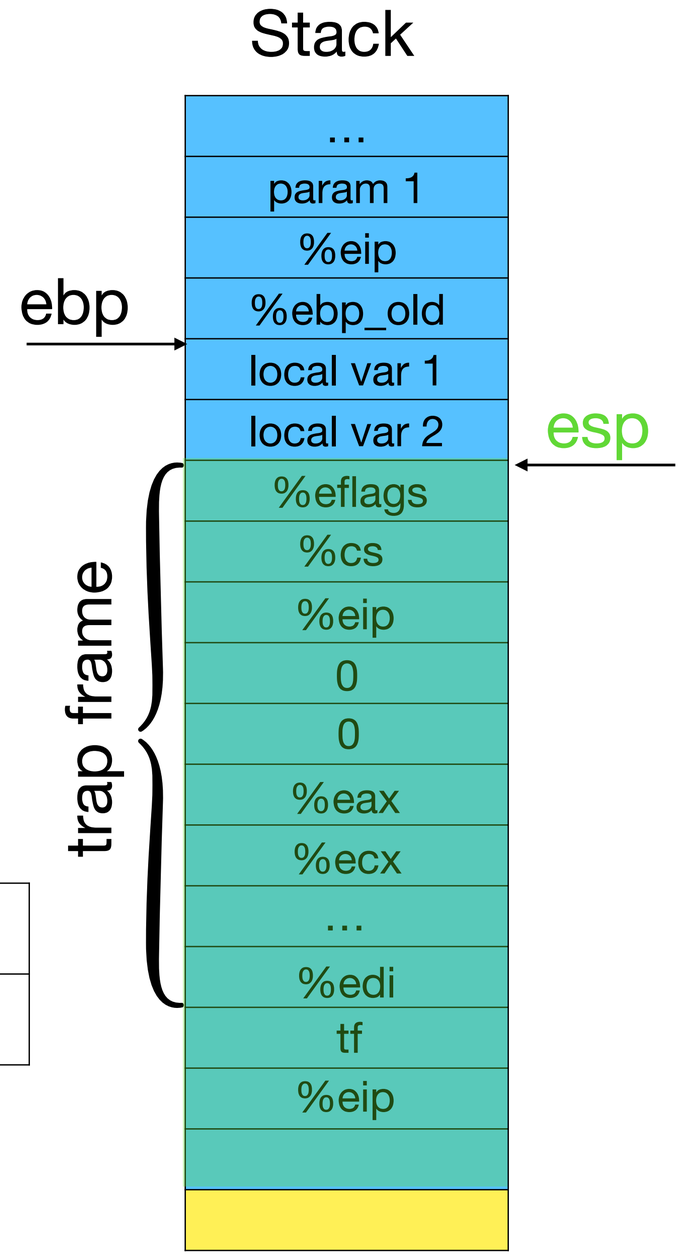
**trap.c**

```
void
trap(struct trapframe *tf)
{
  switch(tf->trapno){
  case T_IRQ0 + IRQ_TIMER:
    ticks++;
    cprintf("Tick! %d\n\0", ticks);
    lapiceoi();
..
return
```

**trapasm.S**

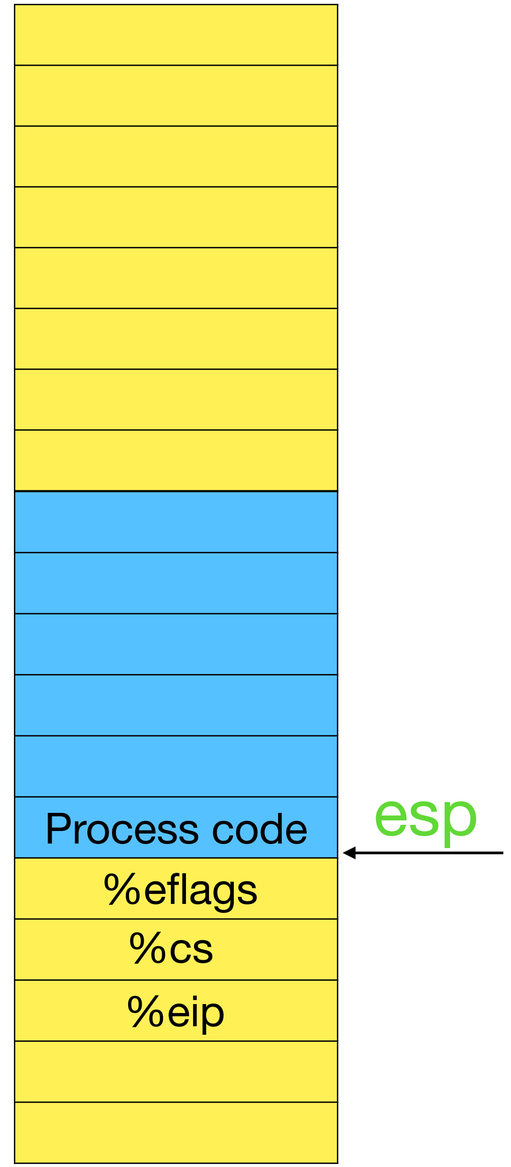
```
alltraps:
  pushal
  pushl %esp
  call trap
  addl $4, %esp
  popal
  addl $0x8, %esp
  iret
```



# Interrupt handling problem

- We cannot trust %esp of the process. Hardware might write (%eflags, %cs, %eip) into another process' address space or into OS memory.
  - Use a separate stack set up by the OS!



# OS tells hardware where the kernel stack is

- In task-state segment (TSS), OS sets up
  - Stack segment for transition to ring 0 in SS0 and
  - Stack pointer for transition to ring 0 in ESP0

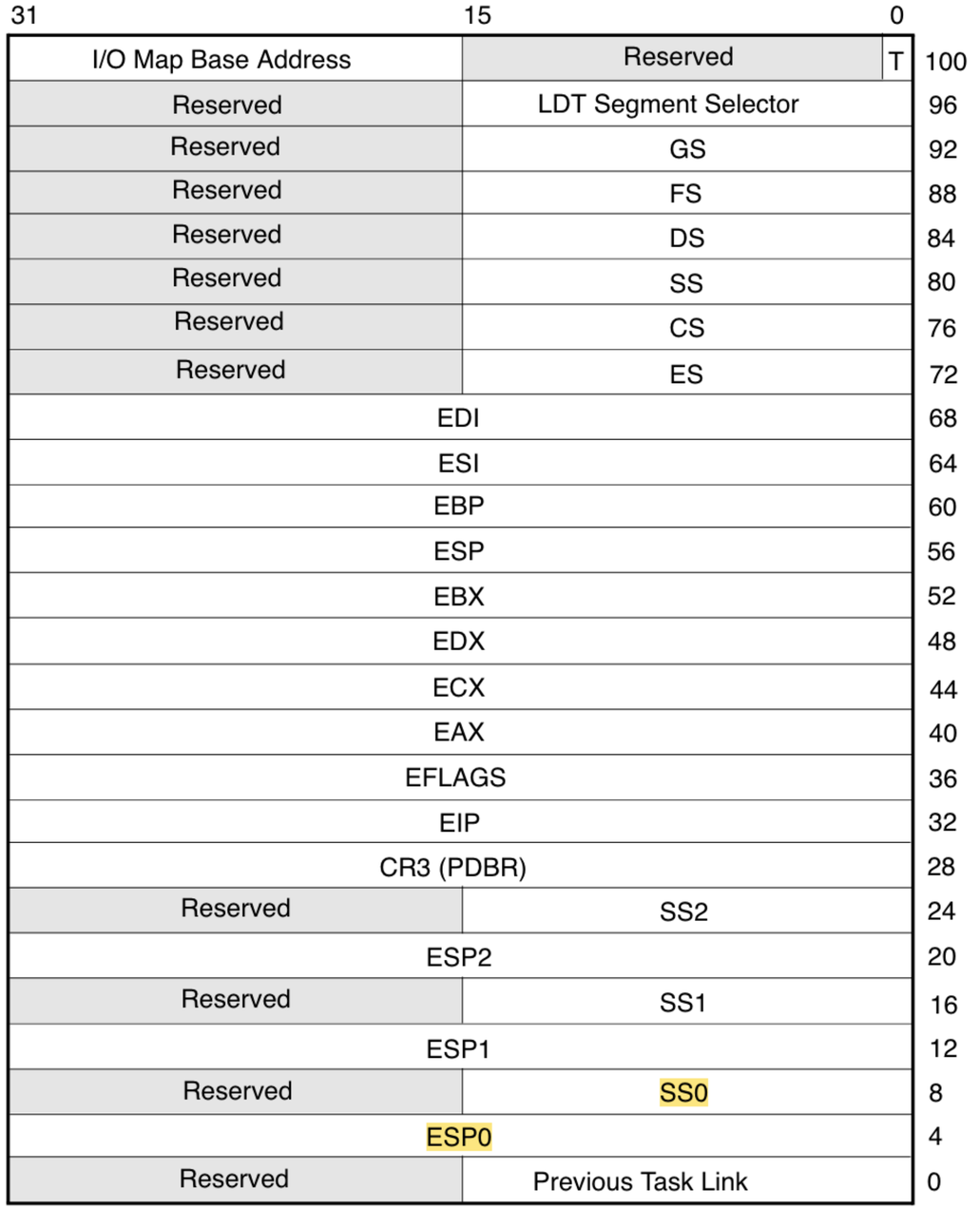



Figure 7-2. 32-Bit Task-State Segment (TSS)

# OS tells hardware where the kernel stack is (2)

- OS sets up an entry in GDT that contains location of TSS
- LTR instruction changes task register

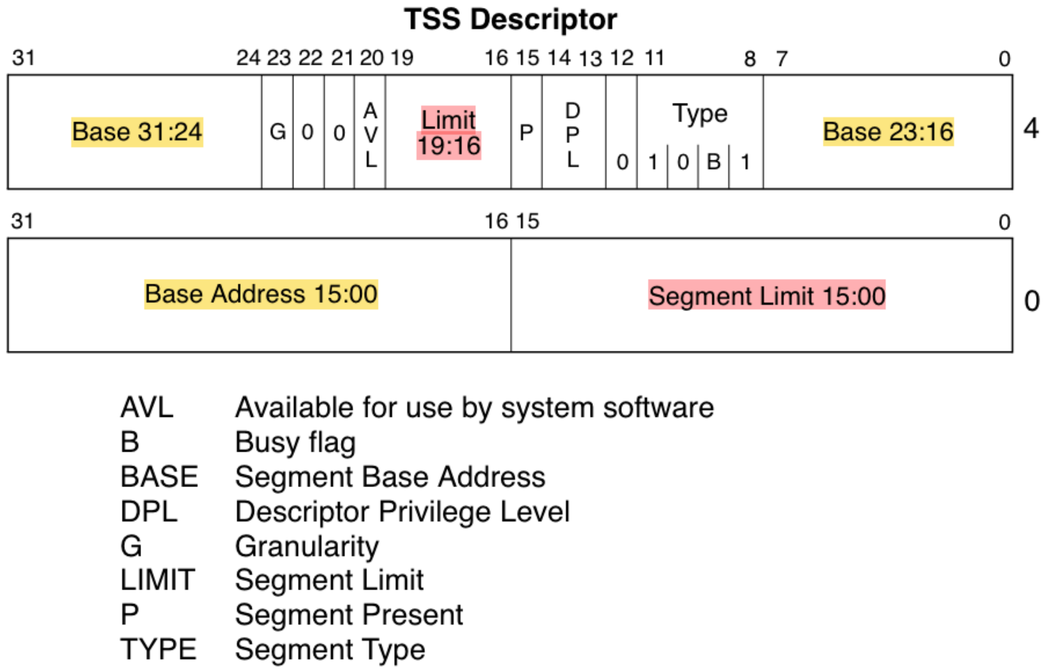



Figure 7-3. TSS Descriptor

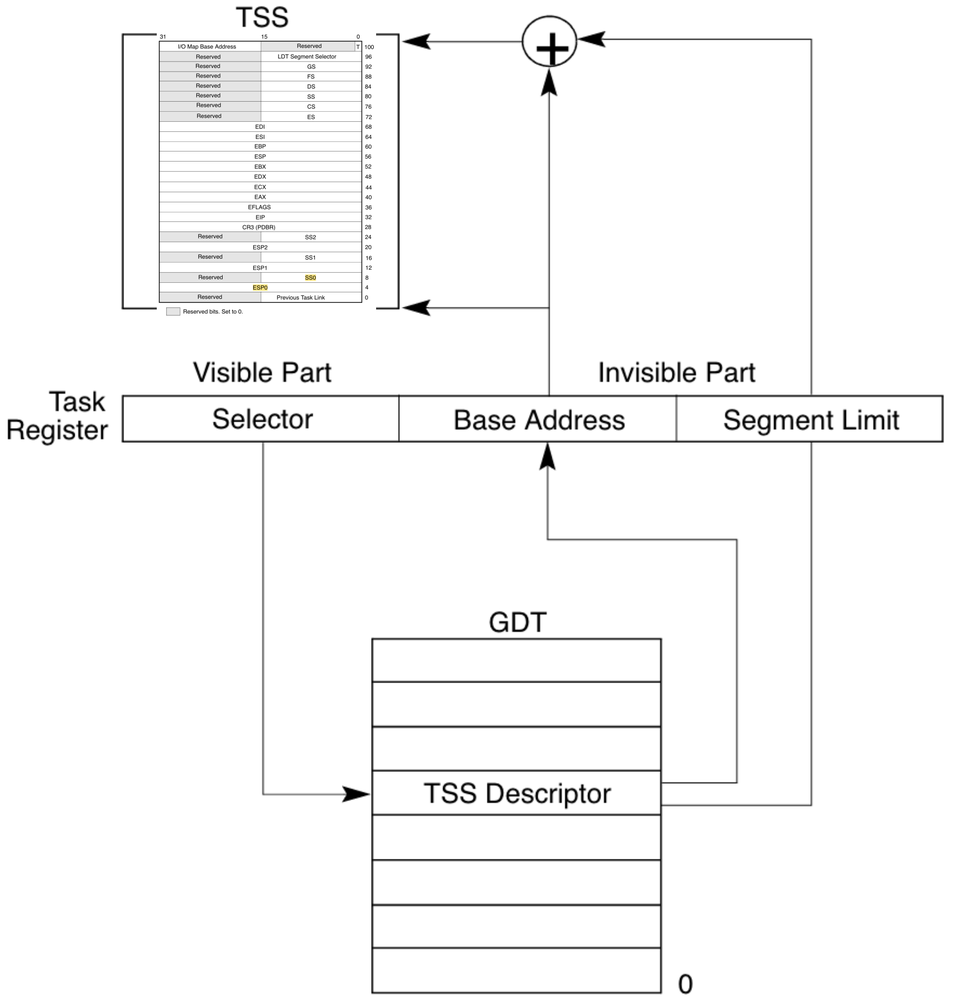


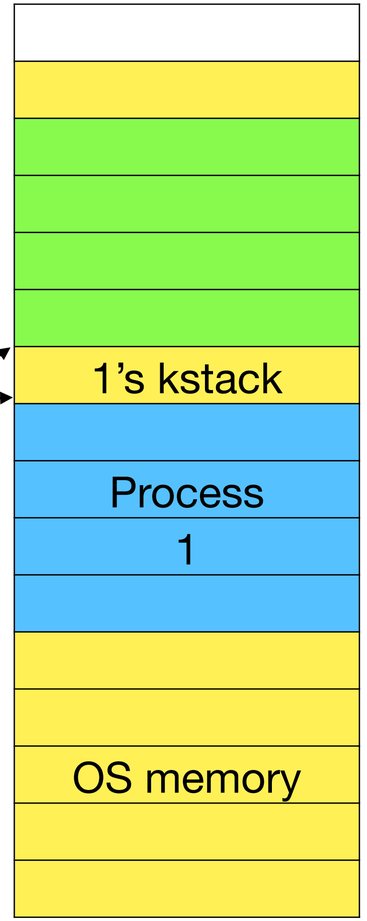

Figure 7-5. Task Register

# Code walkthrough: setting up kernel stack

p16-tss

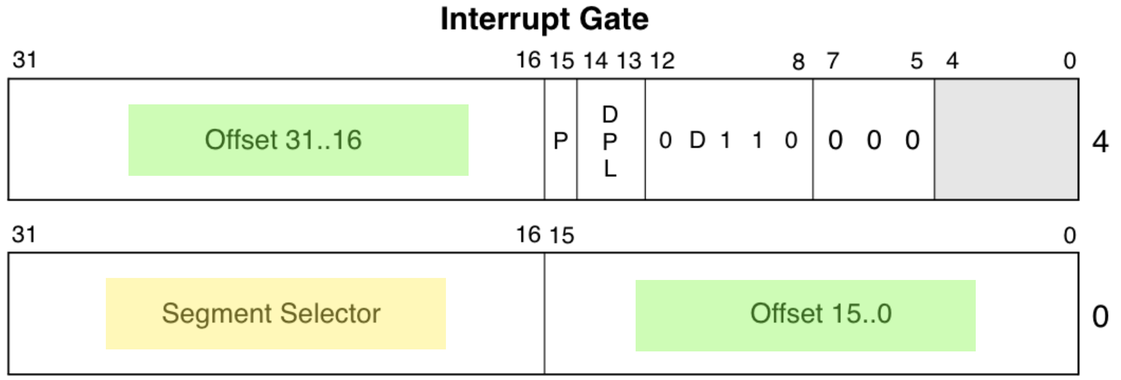
```
void seginit(void) {
  c->gdt[SEG_UCODE] = SEG(.., STARTPROC, (PROCSIZE-1) << 12, DPL_USER);
  c->gdt[SEG_UDATA] = SEG(.., STARTPROC, (PROCSIZE-1) << 12, DPL_USER);
}

static struct proc* allocproc(void) {
  sp = (char*)(STARTPROC + (PROCSIZE>>12));
  p->kstack = sp - KSTACKSIZE;
}

void switchuvm(struct proc *p) {
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  ltr(SEG_TSS << 3);
}
```

```
void scheduler(void) {
 // pick RUNNABLE process p
 switchuvm(p);
 swtch(p->context);
}
```



# Interrupt handling

- Each IDT entry is 64-bits. Contains code segment and eip
- When interrupt appears,
  - Hardware changes SS and ESP according to the TSS
  - Hardware changes CS and EIP to the one pointed by IDT entry.



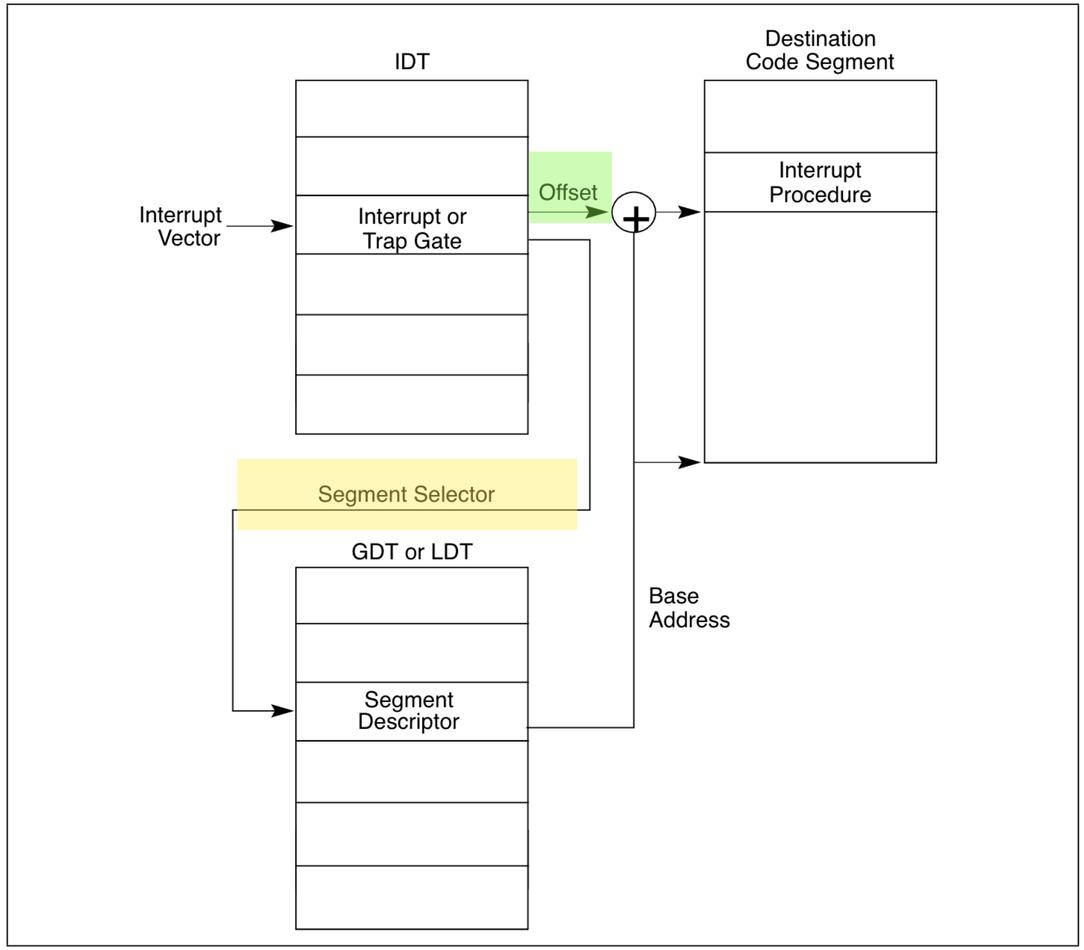




Figure 6-3. Interrupt Procedure Call

# Interrupt handling with user process running

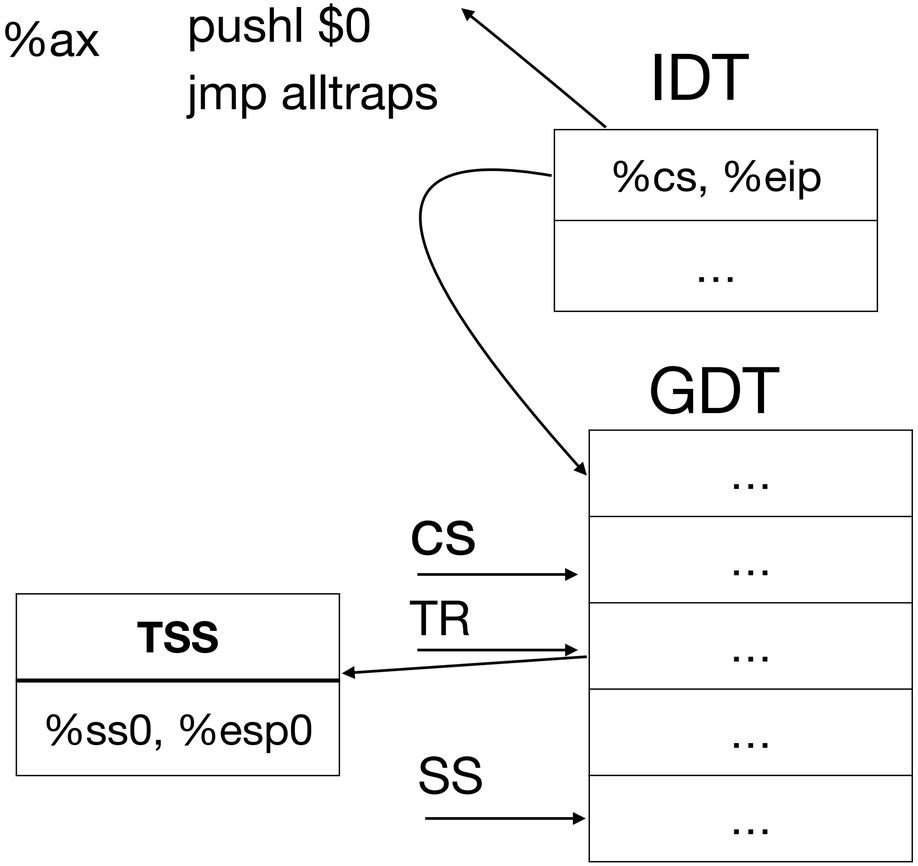
eip → for(;;)
;

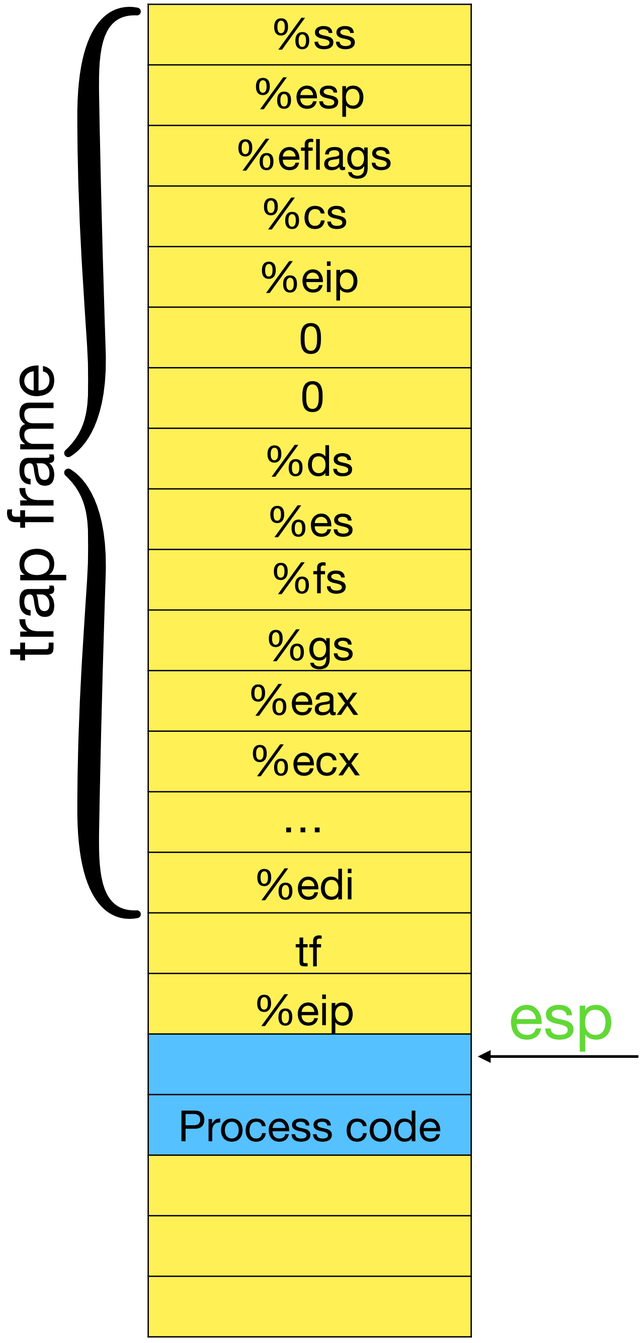
**trapasm.S**

```
alltraps:
  pushl %ds..
  pushal
  movw $(SEG_KDATA<<3), %ax
  movw %ax, %ds..
  pushl %esp
  call trap
  addl $4, %esp
  popal
  popl %ds..
  addl $0x8, %esp
  iret
```

**vectors.S**

```
.globl vector0
vector0:
  pushl $0
  pushl $0
  jmp alltraps
```



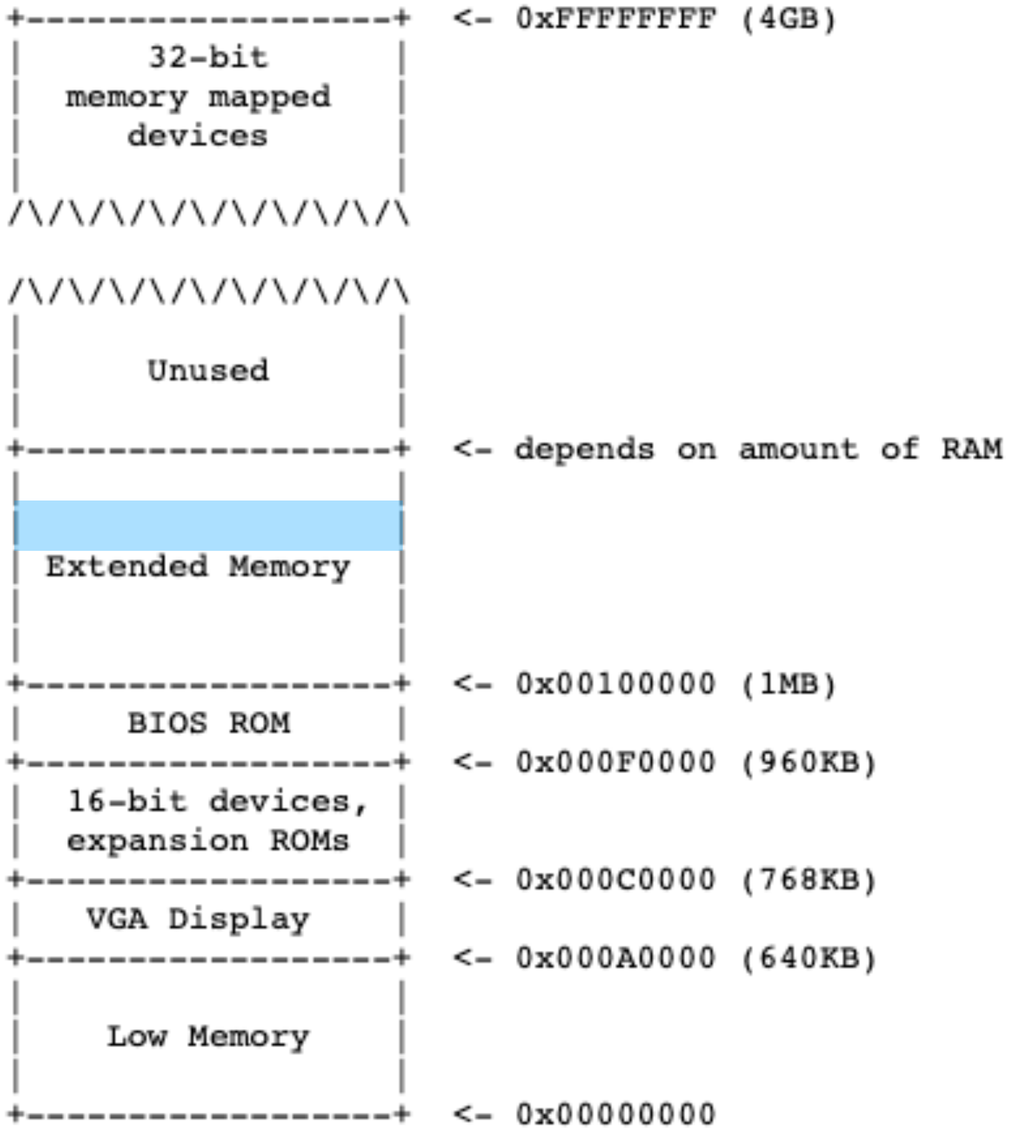



# Protection so far

- Can processes read/write each other's memory?
  - No, since the other process' memory is not *addressable* by (not in the *address space* of) the process
- Can process take over IDT, GDT, TSS?
  - IDT, GDT, TSS are not in address space. Cannot run LIDT, LGDT, LTR instructions since they are *privileged instructions* (only runnable from ring 0)
- Can process change its privilege to ring 0?
  - Not directly. It can change only via trap handling mechanism. OS code runs upon a trap. Trap handling writes to kernel stack.
- Can process run away with the CPU?
  - CPU will be snatched at the time of timer interrupt

# IO protection

- Memory mapped IO
  - Can directly read from / write to IO devices if OS keeps it in *process address space*



# Port-mapped IO protection

```
inb(0x1F7), outb(0x1F2, 1), ..
```

- Option 1: Make in and out instruction privileged. Processes must do IO via kernel.
  - Expensive user mode-kernel mode transitions for IO intensive processes.
- Option 2: Set EFLAGS.IOPL = 3
  - Processes (CPL=3) cannot modify EFLAGS.IOPL
  - Gives permission to access all IO ports

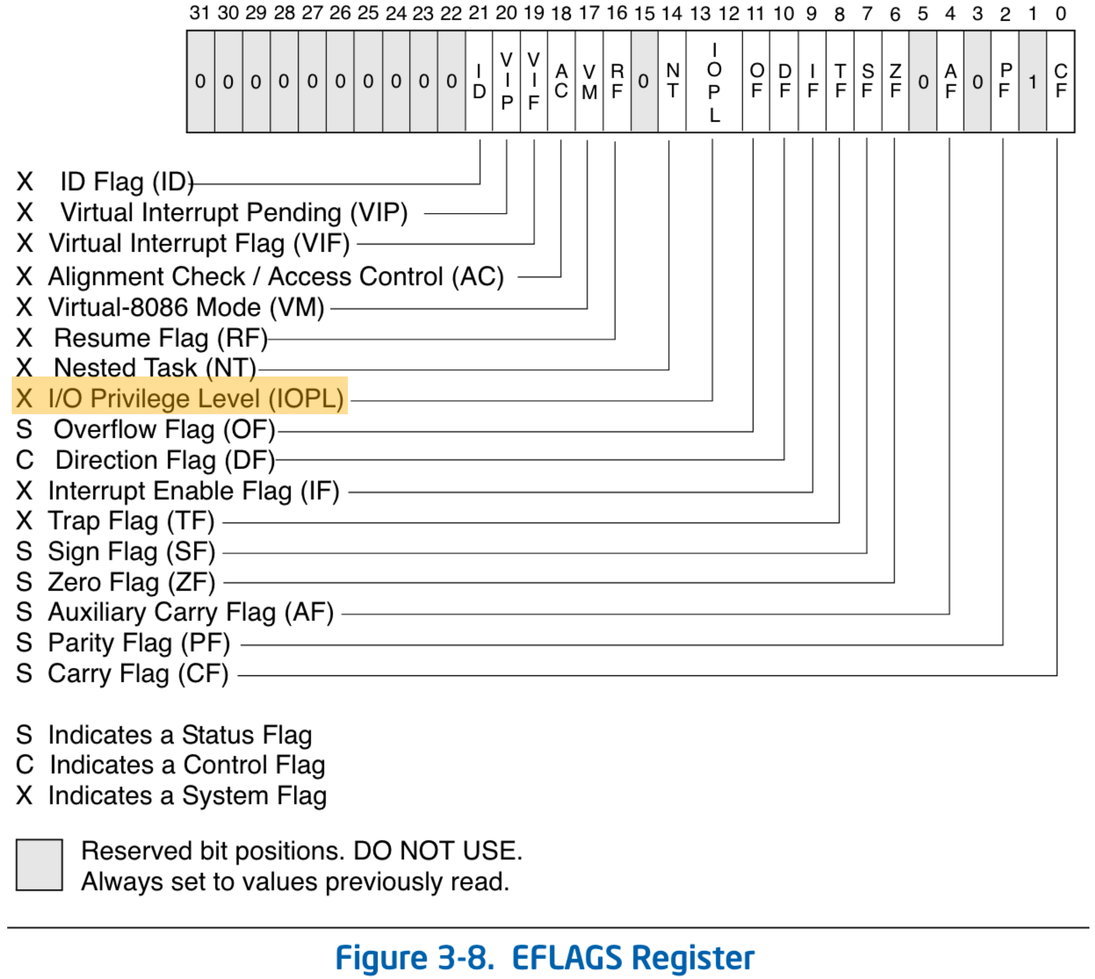



Figure 3-8. EFLAGS Register

# Port-mapped IO protection (2)

- Option 3: Use a bitmap to decide which IO ports are directly accessible
  - If CPL > EFLAGS.IOPL, use bitmap to decide if in, out instruction should generate a trap
  - Example: outb(41, 1) is allowed if 41st bit is set to 0

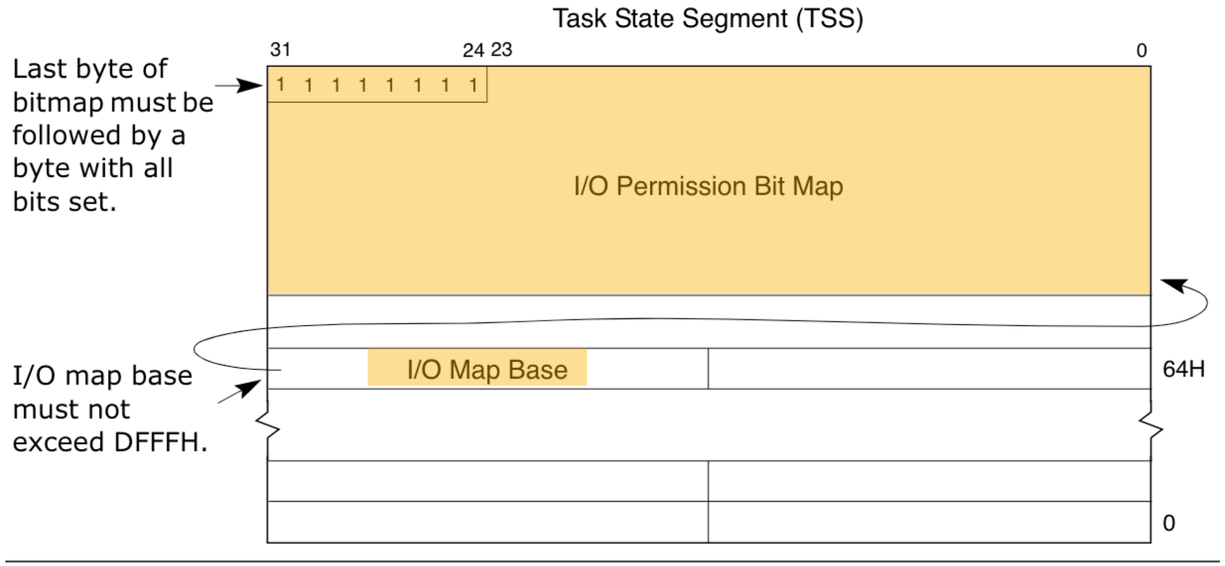



Figure 18-2. I/O Permission Bit Map

| 31 | 15 | 0 | |
|---|---|---|---|
| I/O Map Base Address | Reserved | T | 100 |
| Reserved | LDT Segment Selector | | 96 |
| Reserved | GS | | 92 |
| Reserved | FS | | 88 |
| Reserved | DS | | 84 |
| Reserved | SS | | 80 |
| Reserved | CS | | 76 |
| Reserved | ES | | 72 |
| EDI | | | 68 |
| ESI | | | 64 |
| EBP | | | 60 |
| ESP | | | 56 |
| EBX | | | 52 |
| EDX | | | 48 |
| ECX | | | 44 |
| EAX | | | 40 |
| EFLAGS | | | 36 |
| EIP | | | 32 |
| CR3 (PDBR) | | | 28 |
| Reserved | SS2 | | 24 |
| ESP2 | | | 20 |
| Reserved | SS1 | | 16 |
| ESP1 | | | 12 |
| Reserved | SS0 | | 8 |
| ESP0 | | | 4 |
| Reserved | Previous Task Link | | 0 |

Reserved bits. Set to 0.

Figure 7-2. 32-Bit Task-State Segment (TSS)

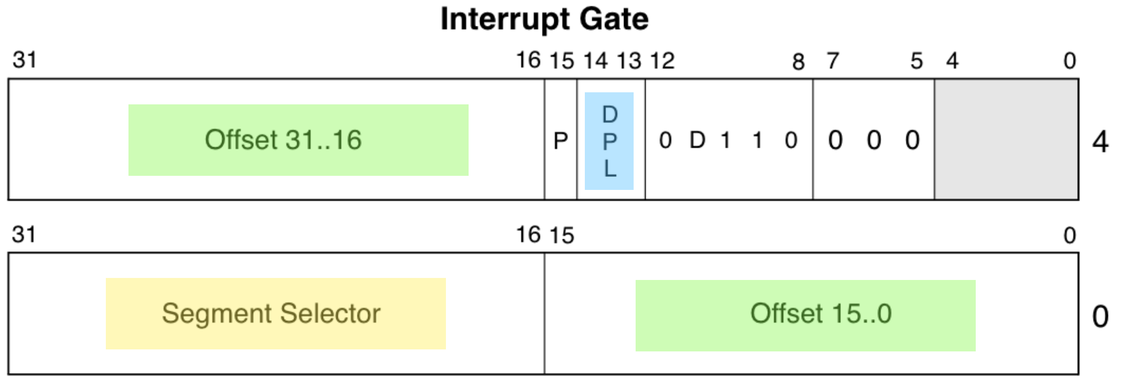
# IO protection in xv6

```
void pinit(void) {
  ..
  p->tf->eflags = FL_IF;
}
```

- Processes cannot directly do IO
  - MMIO: Do not map IO addresses in process address space
  - PIO: Running in and out instructions will generate a trap (jump to OS)

```
void switchuvm(struct proc *p) {
  mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
                                sizeof(mycpu()->ts)-1, 0);
  mycpu()->ts.ss0 = SEG_KDATA << 3;
  mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
  // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
  // forbids I/O instructions (e.g., inb and outb) from user space
  mycpu()->ts.iomb = (ushort) 0xFFFF;
  ltr(SEG_TSS << 3);
}
```

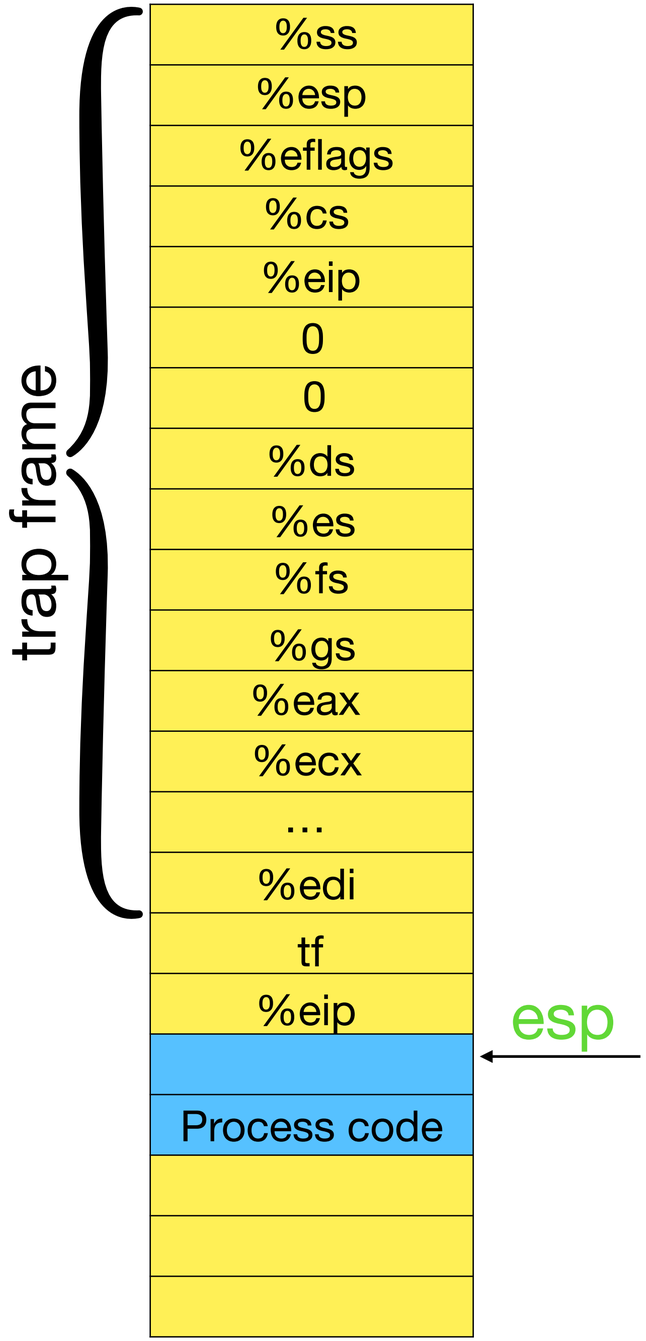
# System calls




- Process wants OS to work on its behalf
  - Run INT xx instruction to trigger interrupt number xx
  - INT xx is allowed if CPL <= DPL for the xx interrupt vector

# Code walkthrough p19-syscall

- trap.c
  - tvinit sets DPL of IDT entry for T_SYSCALL to 3
  - trap calls syscall if interrupt vector is T_SYSCALL
- syscall.c
  - syscall reads eax from trap frame to find which syscall is made and calls that particular call
  - Return value of sys call is set in trap frame's eax
- initproc.S
  - Calls three system calls: open "console" file, write "hello world", close



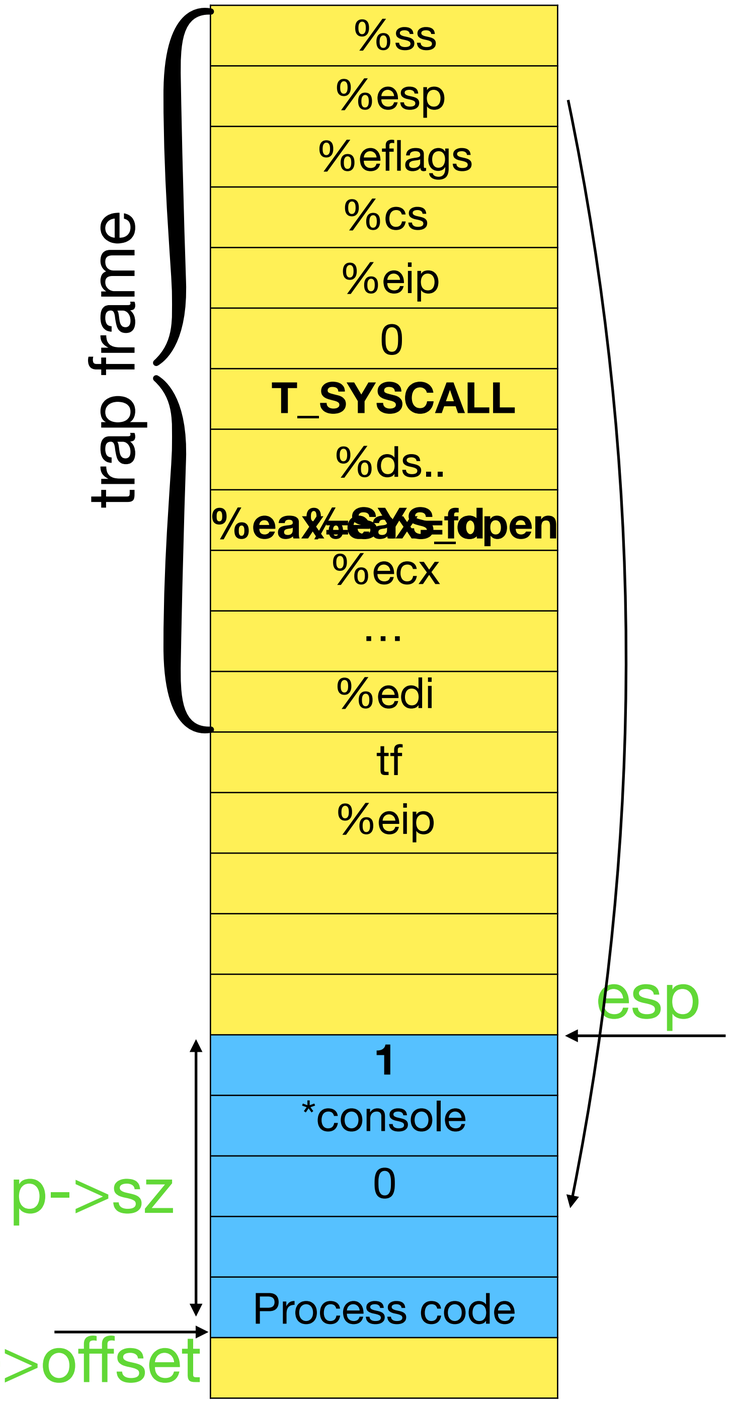
# Visualising syscall handling p19-syscall

```
# sys_open("console", O_WRONLY)
    pushl $1
    pushl $console
    pushl $0
    movl $SYS_open, %eax
    int $T_SYSCALL
    pushl %eax
```

eip

```
int sys_open(void) {
  int fd, omode;
  if(argint(1, &omode) < 0) {
    return -1;
  }
  ..
  return fd;
}
```

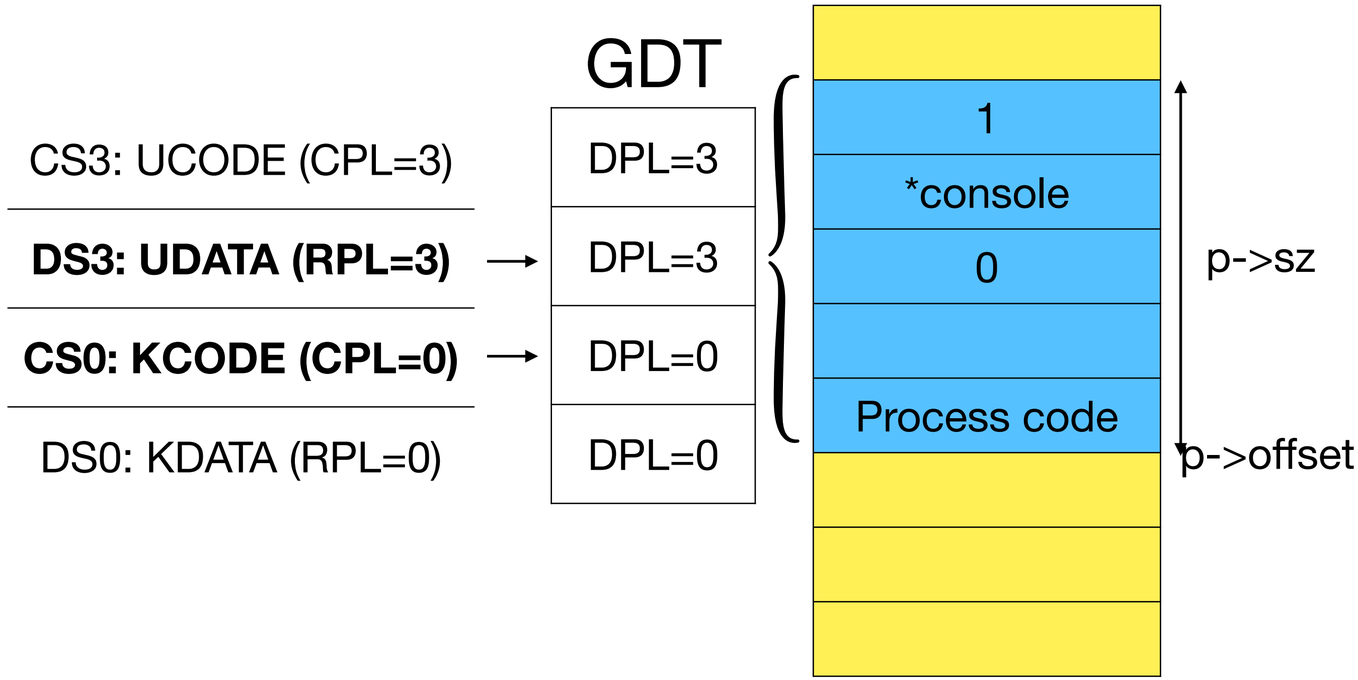
```
int fetchint(uint addr, int *ip) {
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(addr + p->offset);
}

int argint(int n, int *ip) {
  return fetchint((myproc()->tf->esp)
                  + 4 + 4*n, ip);
}
void syscall(void) {
  int num = curproc->tf->eax;
  curproc->tf->eax = syscalls[num]();
}
```



# Syscall parameter checking and translation

```
int fetchint(uint addr, int *ip) {
  if(addr >= p->sz || addr+4 > p->sz)
    return -1;
  *ip = *(int*)(addr + p->offset);
}
```

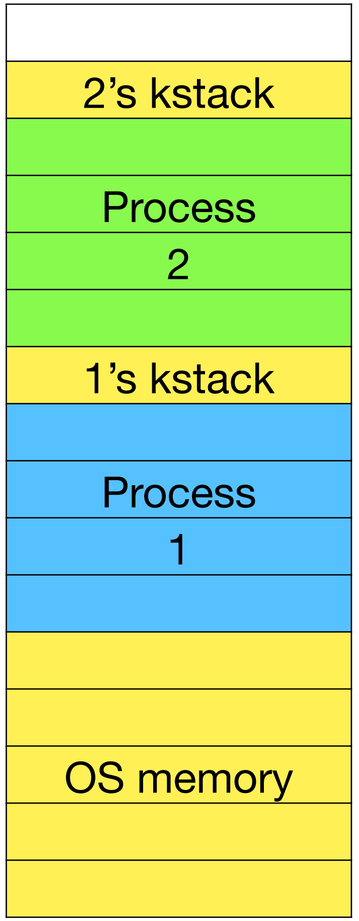
- Syscall parameters must be in process' address space:
  - Check virtual address (VA) is within limit
  - Add base to VA to get physical address (PA)
- Alternative design: use process' DS to read syscall parameters
  - CPL <= DPL and RPL <= DPL



# Code walkthrough: Setting up multiple processes

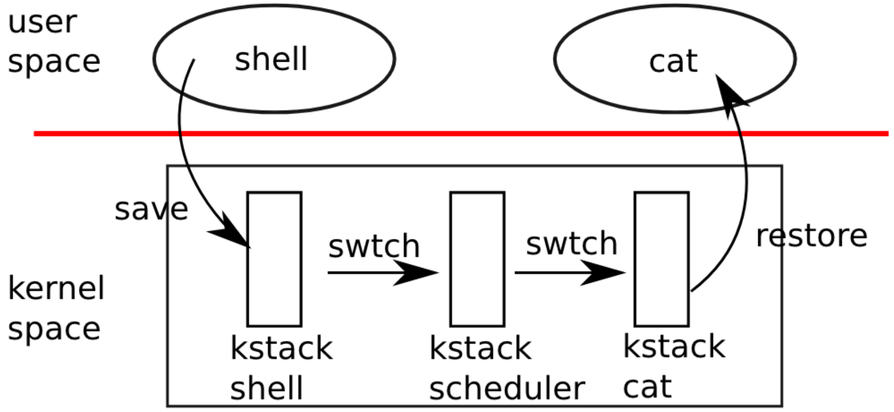
## p18-sched

- main.c
  - Calls pinit twice to start two processes and then calls scheduler
- proc.c
  - pinit calls allocproc which sets up processes like earlier with the added difference that it calls kalloc to find empty 1MB of space
- kalloc.c
  - Maintains free list in chunks of 1MB. kalloc  returns the first element from free list. There is no coalescing and splitting since our OS always asks for 1MB.



# Code walkthrough: context switching

## p18-sched




- main.c
  - Calls scheduler to run the first RUNNABLE process. When giving control, we also remember “scheduler context” to come back to scheduler. Earlier, we were only running one process and were never coming back to the scheduler.
- trap.c
  - Calls yield when timer interrupt happens
- proc.c
  - yield calls sched which switches control from the process to the scheduler
  - scheduler looks for the next RUNNABLE process and switches to it

# Context switching in action: giving control

## p18-sched

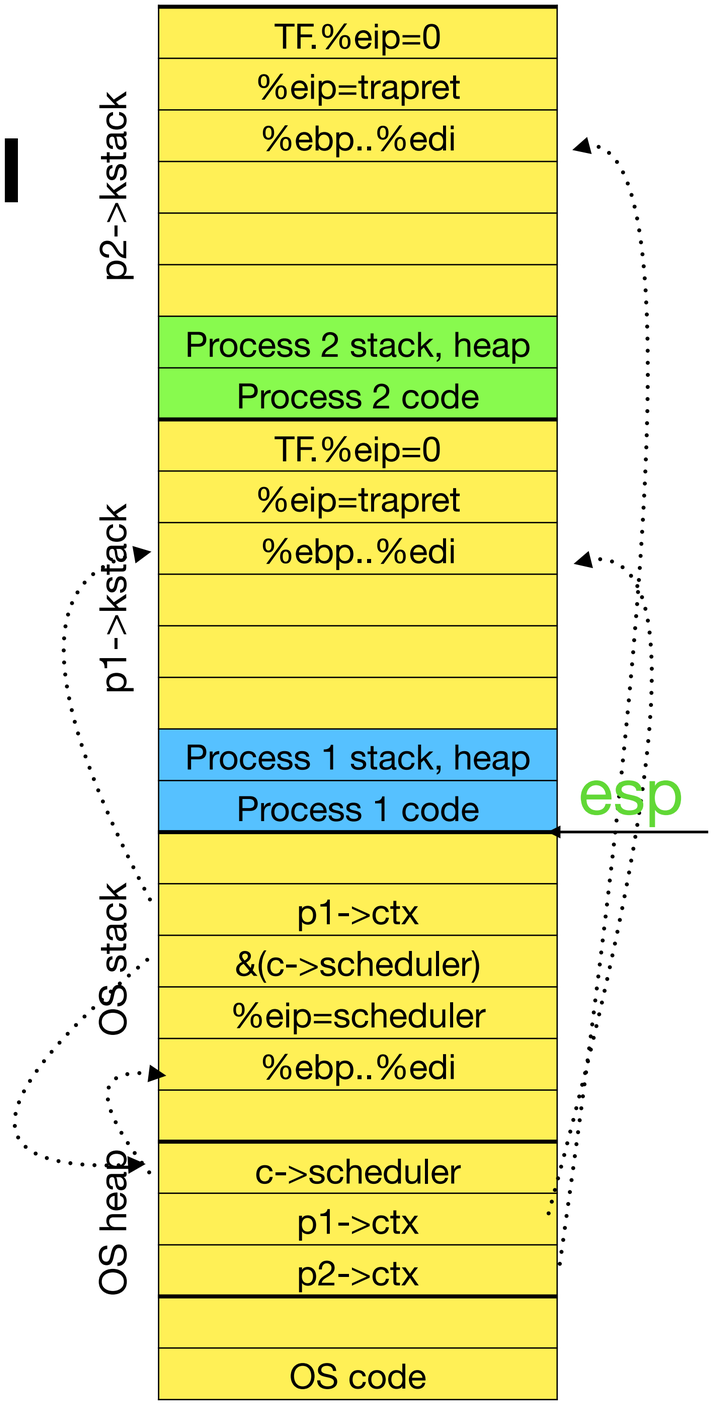
```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi
  movl %esp, (%eax)
  movl %edx, %esp
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

eip →

```
void scheduler(void) {
 struct proc *p;  struct cpu *c = mycpu();
 for(;;){
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
     if(p->state != RUNNABLE)
       continue;
     p->state = RUNNING;
     switchuvm(p);
     swtch(&(c->scheduler), p->context);
   }
  }
}
```

```
void yield(void) {
  struct proc *p = myproc();
  p->state = RUNNABLE;
  swtch(&p->context, c->scheduler);
}
```

```
void trap(struct trapframe *tf) {
  ..
  if(tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
}
```



# Context switching in action: taking control

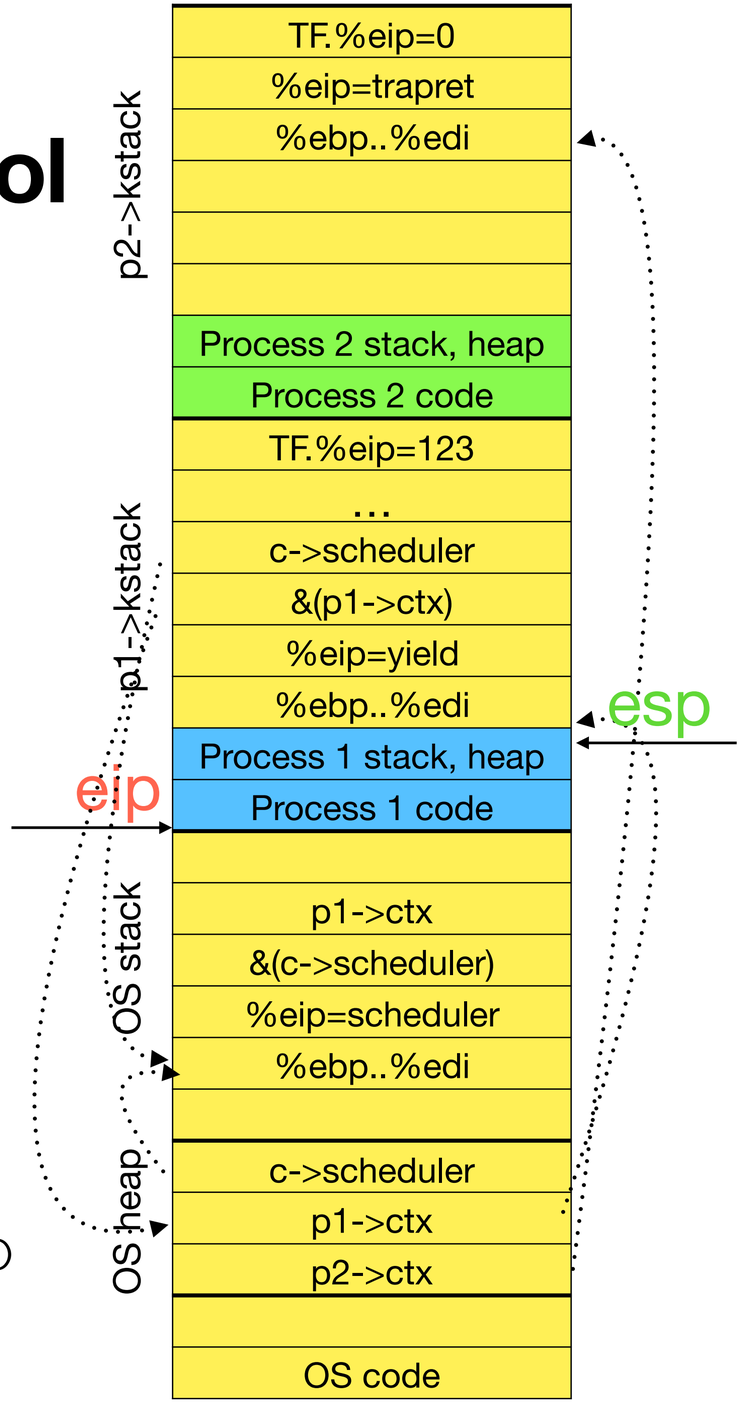
## p18-sched

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi
  movl %esp, (%eax)
  movl %edx, %esp
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

```
void scheduler(void) {
  struct proc *p;  struct cpu *c = mycpu();
  for(;;){
   for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;
      p->state = RUNNING;
      switchuvm(p);
      swtch(&(c->scheduler), p->context);
    }
  }
}
```

```
void yield(void) {
  struct proc *p = myproc();
  p->state = RUNNABLE;
  swtch(&p->context, c->scheduler);
}
```

```
void trap(struct trapframe *tf) {
  ..
  if(tf->trapno == T_IRQ0+IRQ_TIMER)
    yield();
}
```

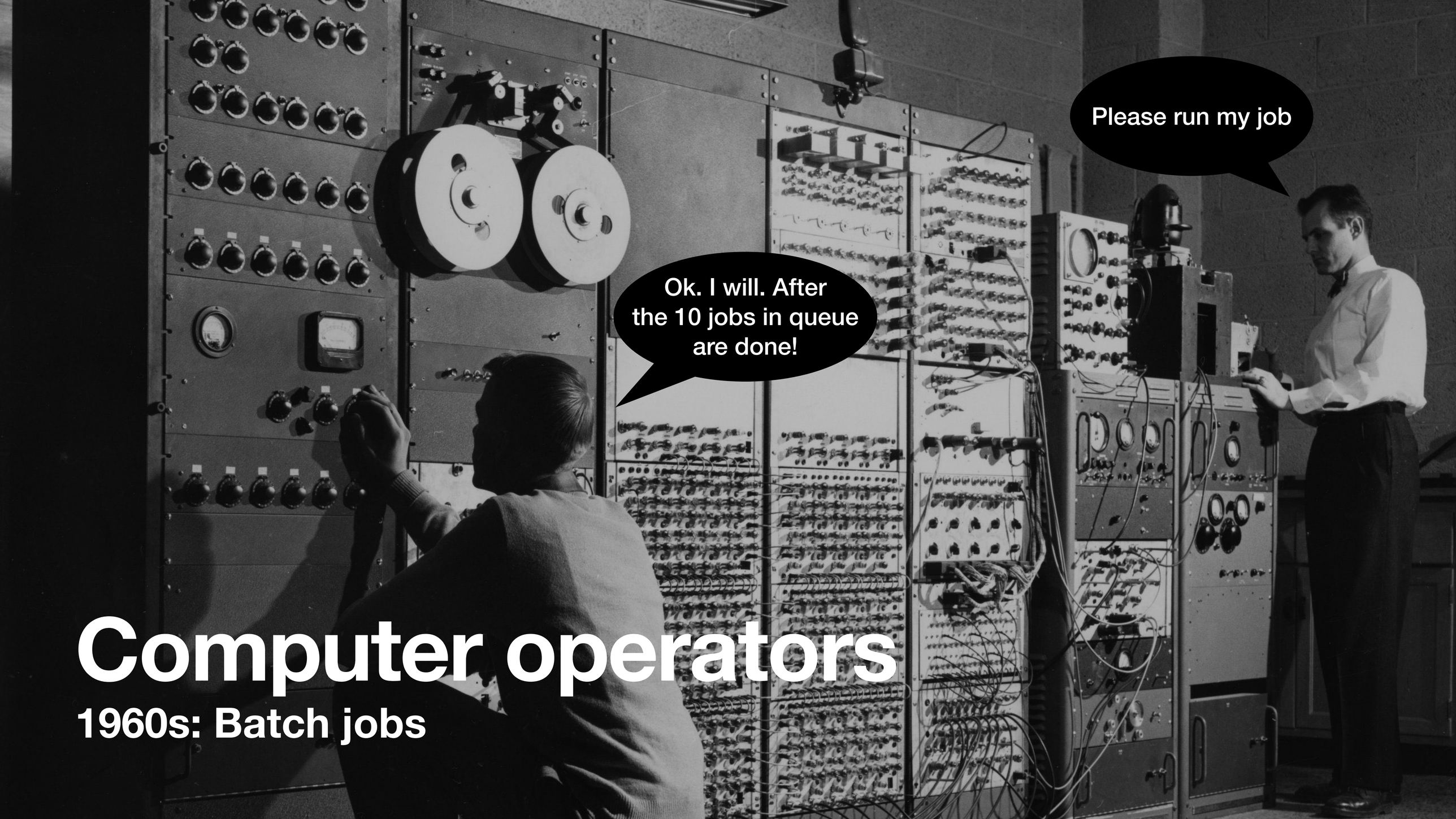


# Scheduling policies

## OSTEP Ch. 7-9

# Scheduling problem

- A number of processes can be runnable.
  - Which process to run next?
  - How long to run each process for?
- What should scheduler try to optimise?
  - What do we know about the behaviour of each process?




# Computer operators

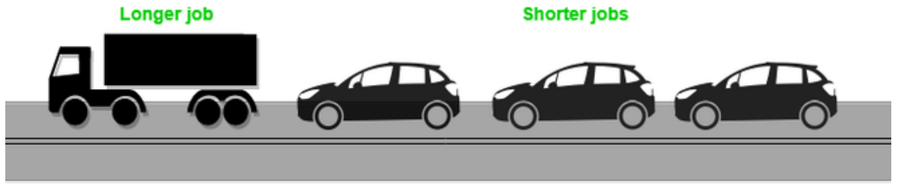
## 1960s: Batch jobs

# Batch job characteristics

- Once started, each job runs to completion
- Run-time of each job is known
- Main metric: turnaround time = (completion time - arrival time)

# FIFO

- Avg turnaround time = (10+20+30)/3 = 20
- Avg turnaround time = (100+110+120)/3 = 110
- Convoy effect: fast jobs are stuck behind slow jobs



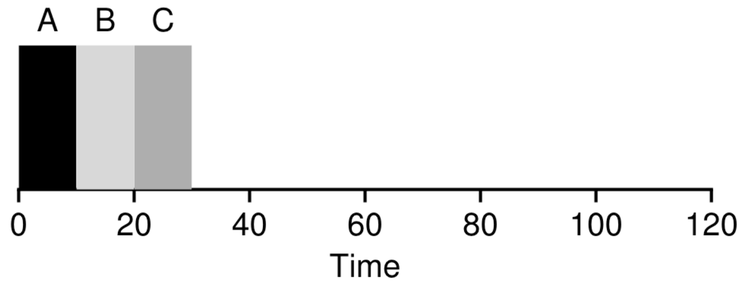




Figure 7.1: **FIFO Simple Example**

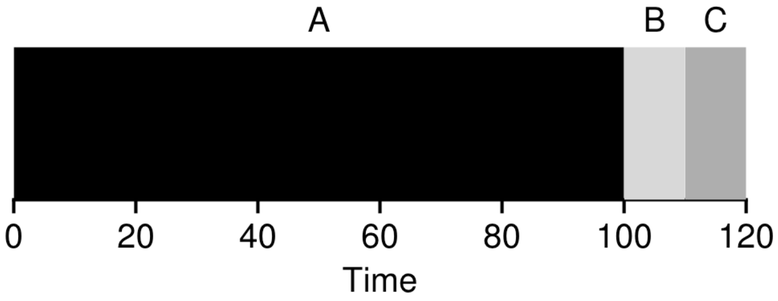
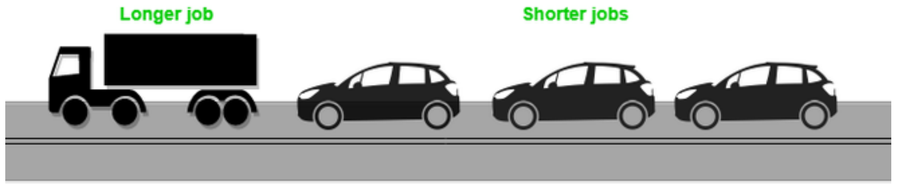



Figure 7.2: **Why FIFO Is Not That Great**

# Shortest job first

- Avg turnaround time = (10+20+120)/3 = 50
- Optimal scheduling algorithm
- Jobs may not arrive at the same time.
- Avg turnaround time = (100+100+110)/3 = 103.33
- Again leads to convoy effect: fast jobs are stuck behind slow jobs



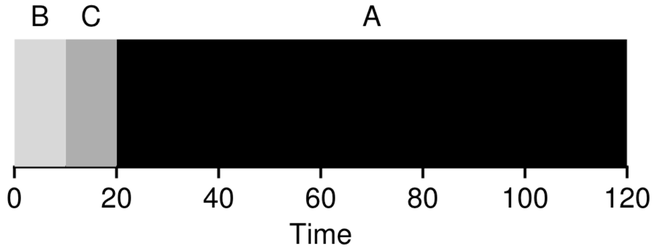




Figure 7.3: **SJF Simple Example**

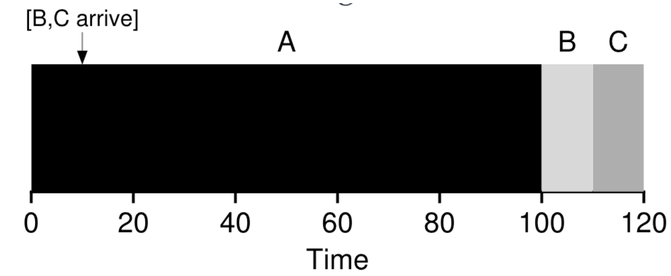



Figure 7.4: **SJF With Late Arrivals From B and C**

# Batch job characteristics

- ~~Once started, each job runs to completion~~
- Preemptive schedulers switch to other processes before a job finishes
- Run-time of each job is known
- Main metric: turnaround time = (completion time - arrival time)

# Shortest time to completion first

- Avg turnaround time = (10+20+120)/3 = 50
- Optimal scheduling algorithm

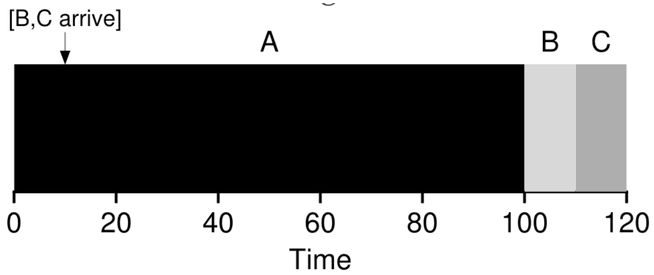



Figure 7.4: **SJF With Late Arrivals From B and C**

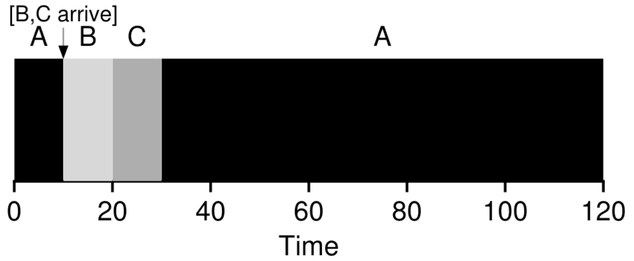



Figure 7.5: **STCF Simple Example**

# Personal computers

## 1980s: Interactive jobs!

# Interactive job characteristics

- Preemptive schedulers switch to other processes before a job finishes
- ~~Run-time of each job is known~~
- ~~Main metric: turnaround time ≡ (completion time - arrival time)~~
- Main metric: response time = (first run - arrival time)
- Avg response time in SJF = (0+5+10)/3 = 15

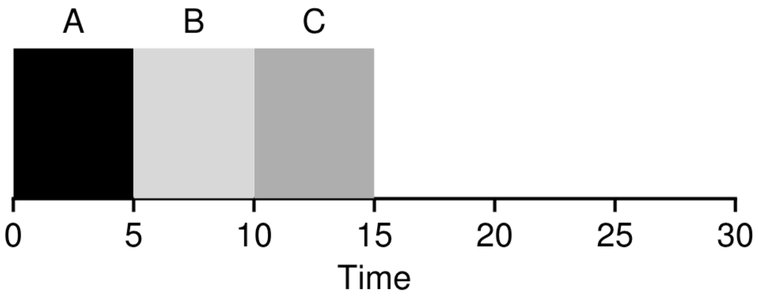



Figure 7.6: **SJF Again (Bad for Response Time)**

# Round robin

- Run for a time slice T (scheduling quantum)
- Avg response time = (0+1+2)/3 = 1
- How long should the time slice T be?
  - T >> context switch time (~1 us)
  - T ~ human response time (few ms)
  - T ~ Used to be ~10ms in 1970s. Still ~10ms. CPU speeds: 60MHz -> 3GHz.
- Worst policy for turnaround time: elongates completion times of all jobs
- Gives “fair share” to all jobs within small amount of time

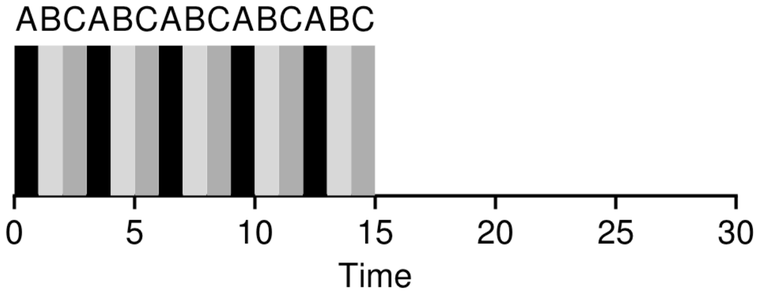



Figure 7.7: **Round Robin (Good For Response Time)**

# Proportional share

- All jobs are not equally important
  - Dropbox syncing should get lesser CPU time than music playback and gameplay
  - Give “proportional share” to jobs based on their importance

# Stride scheduling

```
current = remove_min(queue);       // pick client with minimum pass
schedule(current);                 // use resource for quantum
current->pass += current->stride; // compute next pass using stride
insert(queue, current);            // put back into the queue
```

- Higher priority processes have low strides
- Example: A:B:C = 2:1:5
- Deterministic scheduler

| Pass(A) (stride=100) | Pass(B) (stride=200) | Pass(C) (stride=40) | Who Runs? |
|---|---|---|---|
| 0 | 0 | 0 | A |
| 100 | 0 | 0 | B |
| 100 | 200 | 0 | C |
| 100 | 200 | 40 | C |
| 100 | 200 | 80 | C |
| 100 | 200 | 120 | A |
| 200 | 200 | 120 | C |
| 200 | 200 | 160 | C |
| 200 | 200 | 200 | ... |

Figure 9.3: **Stride Scheduling: A Trace**

# Lottery scheduling

- Probabilistic scheduler
  - A:B ~ 3:1
  - Generate a random number 0-99 in each time quantum
    - 0-74: run A, 75-99: run B

```
63 85 70 39 76 17 29 41 36 39 10 99 68 83 63 62 43  0 49 49

 A     A  A     A  A  A  A  A  A     A     A  A  A  A  A  A
    B        B                    B     B
```

# Lottery scheduling implementation

- Each process holds its own number of tickets
- Generate a random number (e.g. 300) between 0 and total number of tickets (e.g. 400) in each time quantum
- Traverse list while reducing by tickets to find the job (e.g. C)


| | | |
|---|---|---|
| 300 | 200 | 150 |
| 125 | 25 | |

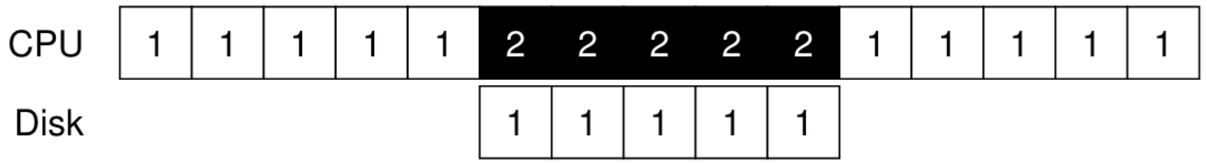
# Setting priority

- When processes are trusted, they can raise or lower their priority
  - Example: music player lowers its priority when user pauses the song
    - e.g, B changes its tickets to 250
- Otherwise, users can manually lower priority:

```
time taskset -c 1 ./nicedemo
time nice -n 19 taskset -c 1 ./nicedemo
```

# Process states

- Processes can be blocked upon doing IO



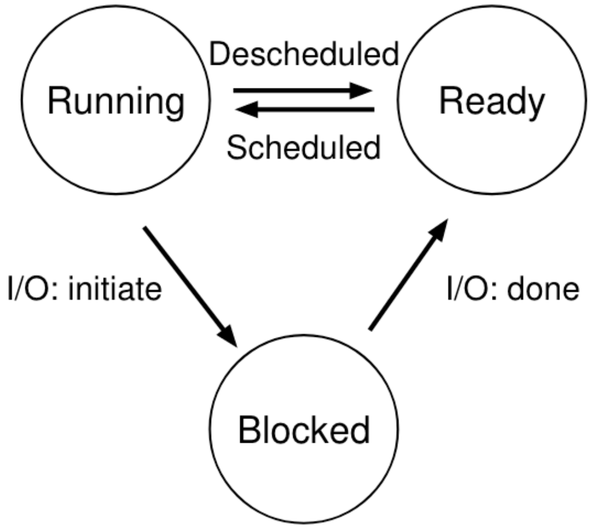
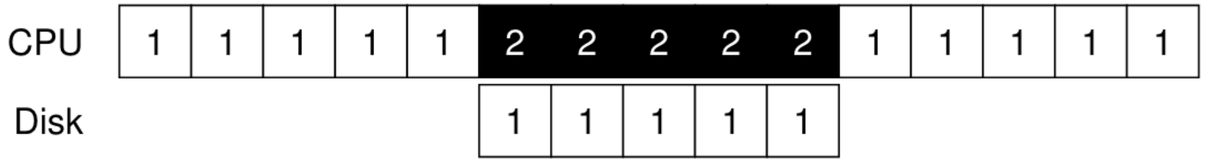




Figure 4.2: **Process: State Transitions**

# Blocking in action

## Case study: xv6 disk driver




```
void sleep(void *chan) {
  struct proc *p = myproc();
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  // Tidy up.
  p->chan = 0;
}
void wakeup(void *chan) {
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if(p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}
```

```
void ideintr(void) {
  struct buf *b = idequeue;
  if(!(b->flags & B_DIRTY) && idewait(1) >= 0)
    insl(0x1f0, b->data, BSIZE/4);
  b->flags |= B_VALID;
  b->flags &= ~B_DIRTY;
  wakeup(b);
}

void iderw(struct buf *b){
  if(idequeue == b)
    idestart(b);
  while((b->flags & (B_VALID|B_DIRTY)) != B_VALID){
    noop();
    sleep(b);
  }
}
```

# IO aware scheduling

- Mixture of CPU intensive and IO intensive processes
  - Turnaround time is important for CPU intensive processes, e.g, training an ML model
  - Response time is important for interactive processes, e.g, editor
- Challenge: OS does not know which process is CPU intensive / IO bound
- Round robin (fair share) and proportional share algorithms assume all processes are CPU intensive
- Example: xv6 scheduler gives poor response time for interactive processes

| CPU | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 1 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Disk | | | | | | | | | | | | | | | | | | | |

| CPU | 1 | 1 | 2 | 3 | 3 | 3 | 2 | 2 | 2 | 3 | 3 | 3 | 1 | 1 | 2 | 3 | 3 | 3 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Disk | | | 1 | 1 | 1 | 1 | 1 | | | | | | | | 1 | 1 | 1 | 1 | 1 |

# Multi-level feedback queue (MLFQ)

- Strict priority using multiple queues
  - (A > C) => A runs
  - (A = B) => Round robin
- Goal: IO bound (interactive) processes shall take higher priority over CPU intensive processes
- Challenge: OS does not know nature of the process

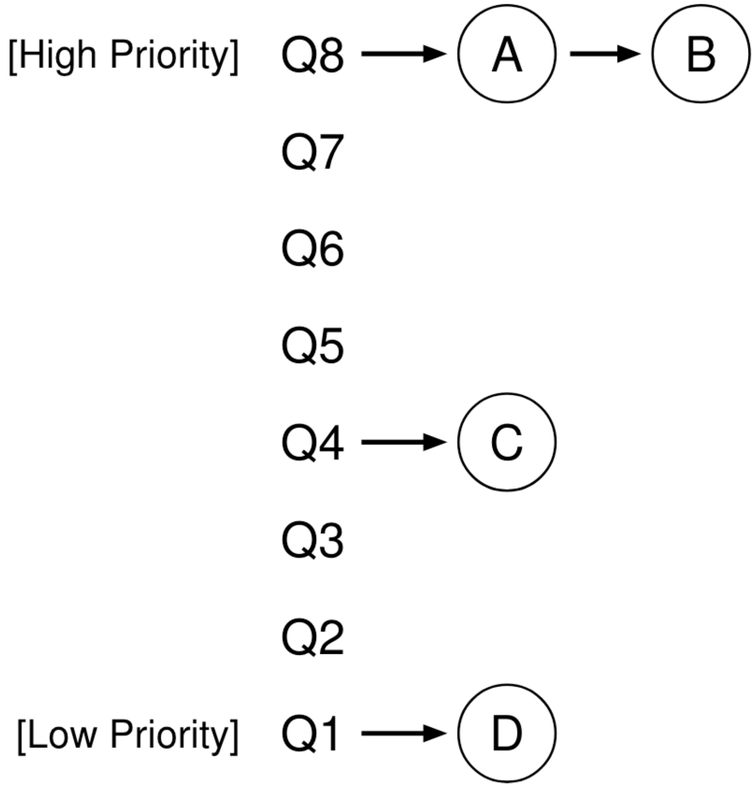



Figure 8.1: **MLFQ Example**

# MLFQ level management

- Observation: A CPU intensive process completely uses up its time slice. e.g, process 2

| CPU | 1 | 1 | 2 | 3 | 3 | 3 | 2 | 2 | 2 | 3 | 3 | 3 | 1 | 1 | 2 | 3 | 3 | 3 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Disk | | | 1 | 1 | 1 | 1 | 1 | | | | | | | | 1 | 1 | 1 | 1 | 1 |

- Goal: IO bound (interactive) processes shall take higher priority over CPU intensive processes
  - If a process completely uses up its time slice, lower its priority

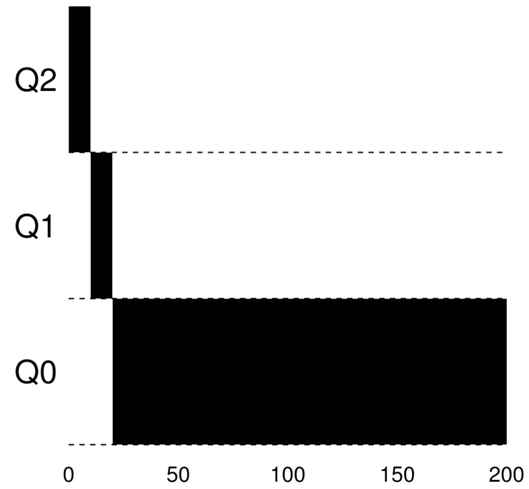



Figure 8.2: **Long-running Job Over Time**

# MLFQ approximates STCF

- Short jobs get priority over long-running jobs
- IO intensive jobs are prioritised over CPU intensive processes => better response time for interactive jobs

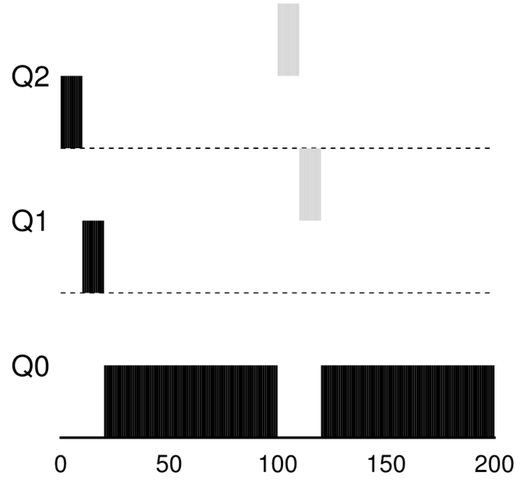



Figure 8.3: **Along Came An Interactive Job**

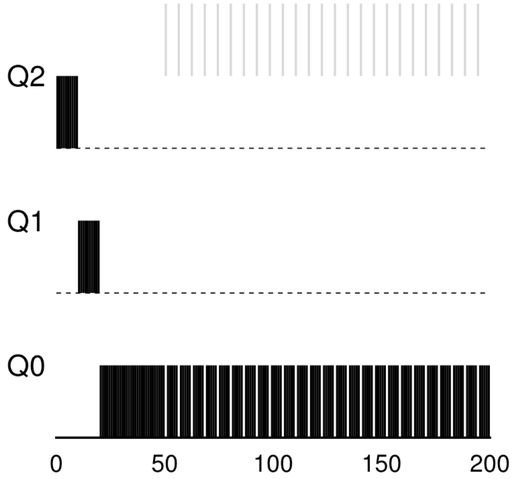
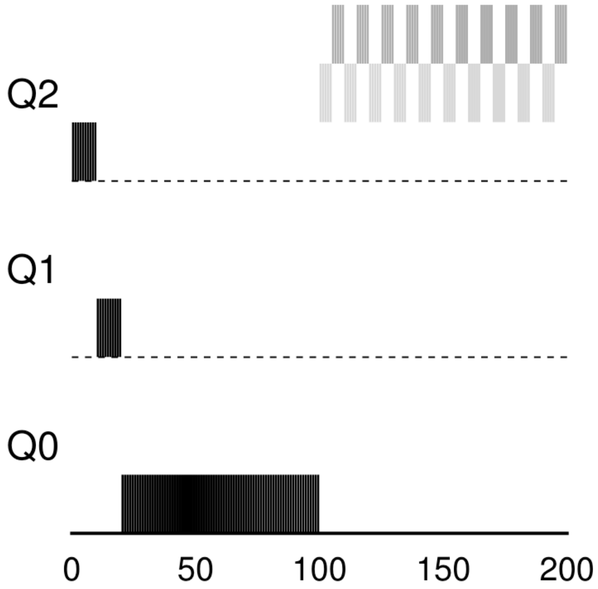
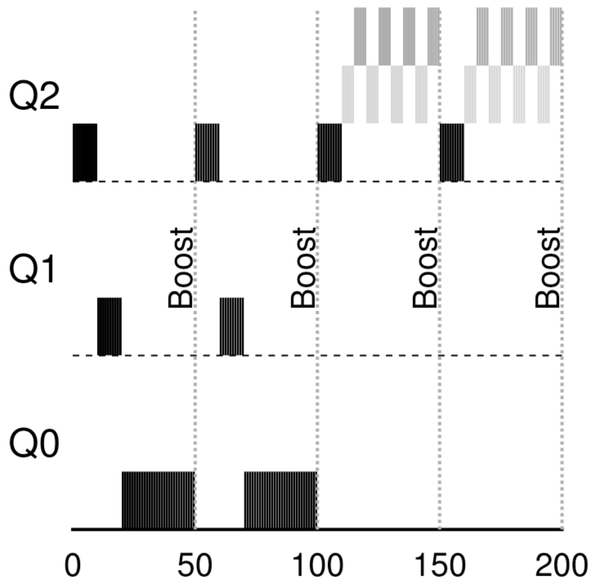



Figure 8.4: **A Mixed I/O-intensive and CPU-intensive Workload**

# Starvation freedom

- Many IO bound jobs can starve CPU intensive jobs
- Boost all jobs to highest priority queue after some time quantum
- Also helps if CPU intensive job has become IO intensive job



# Reduce scheduling overhead for CPU intensive jobs

- If there are no IO bound jobs, CPU intensive jobs run without preemption for a longer time slice

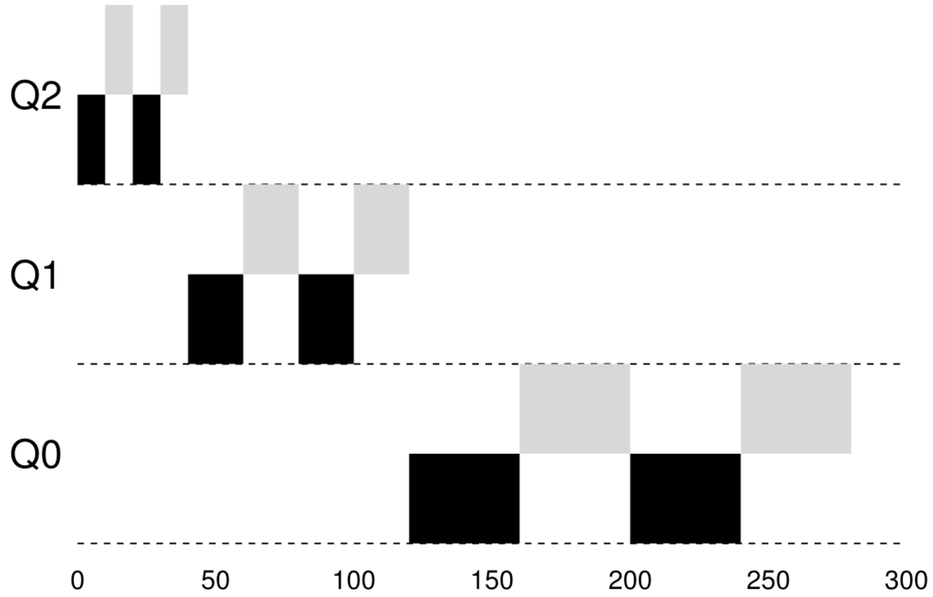
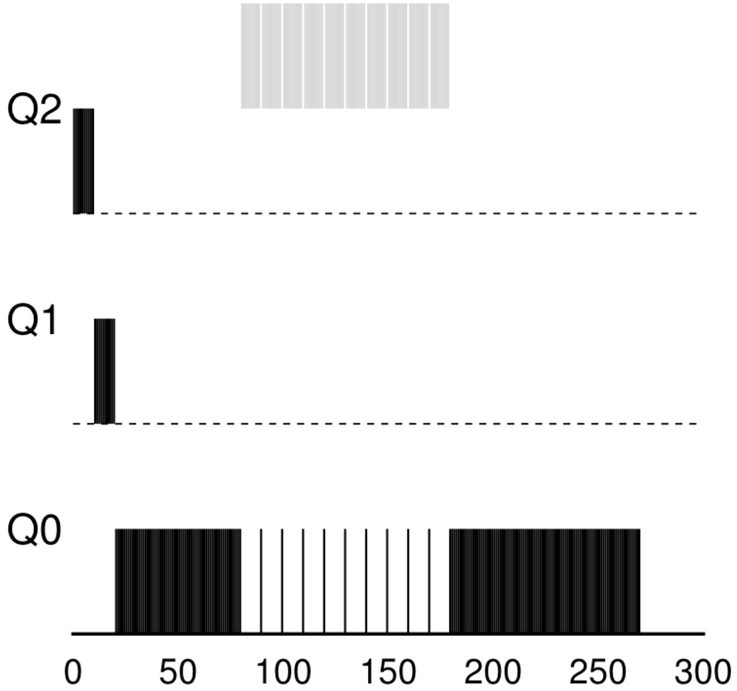



Figure 8.7: **Lower Priority, Longer Quanta**

# Scheduler attacks

- Example: yield CPU right before time quanta expires to stay in highest priority
- Big concern in Linux 2.4 ~2006. Freeze up desktops



# Linux Completely Fair Scheduler (CFS)

- Priority scheduler:
  - Low nice value => More CPU time
  - High priority processes get longer time slices

$$\texttt{time\_slice}_k = \frac{\texttt{weight}_k}{\sum_{n=0}^{n-1} \texttt{weight}_i} \cdot \texttt{sched\_latency}$$

```
static const int prio_to_weight[40] = {
 /* -20 */     88761,     71755,     56483,     46273,     36291,
 /* -15 */     29154,     23254,     18705,     14949,     11916,
 /* -10 */      9548,      7620,      6100,      4904,      3906,
 /*  -5 */      3121,      2501,      1991,      1586,      1277,
 /*   0 */      1024,       820,       655,       526,       423,
 /*   5 */       335,       272,       215,       172,       137,
 /*  10 */       110,        87,        70,        56,        45,
 /*  15 */        36,        29,        23,        18,        15,
};
```

# Completely Fair Scheduler (CFS)

- Each process has vruntime. Update vruntime after each time process is run.

$$\texttt{vruntime}_i = \texttt{vruntime}_i + \frac{\texttt{weight}_0}{\texttt{weight}_i} \cdot \texttt{runtime}_i$$

- Maintain runnable processes in a red-black tree (sorted by vruntime). Run the process with lowest vruntime.
- When a new process is run:
  - Assign it lowest vruntime (instead of assigning it zero). Otherwise the new process monopolises the CPU

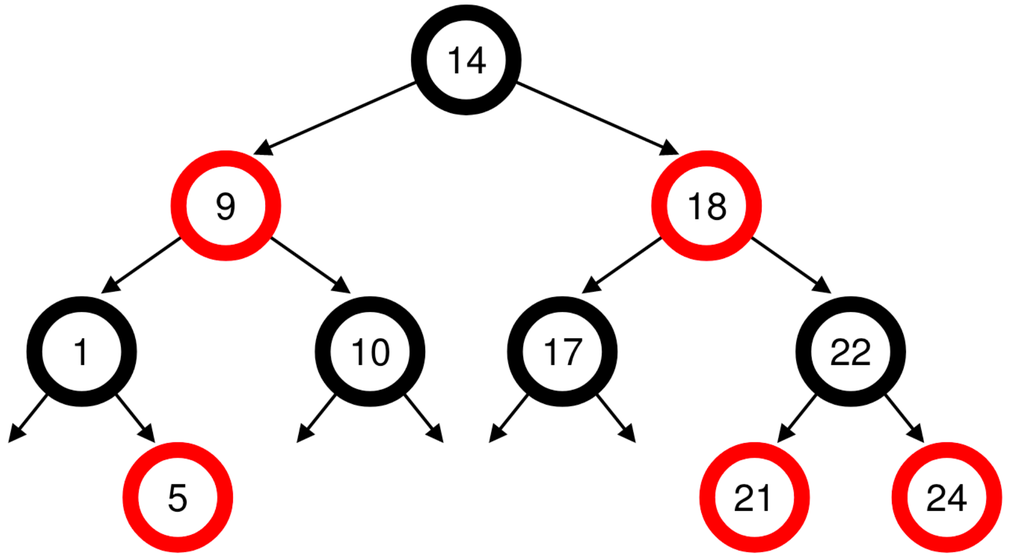



Figure 9.5: **CFS Red-Black Tree**

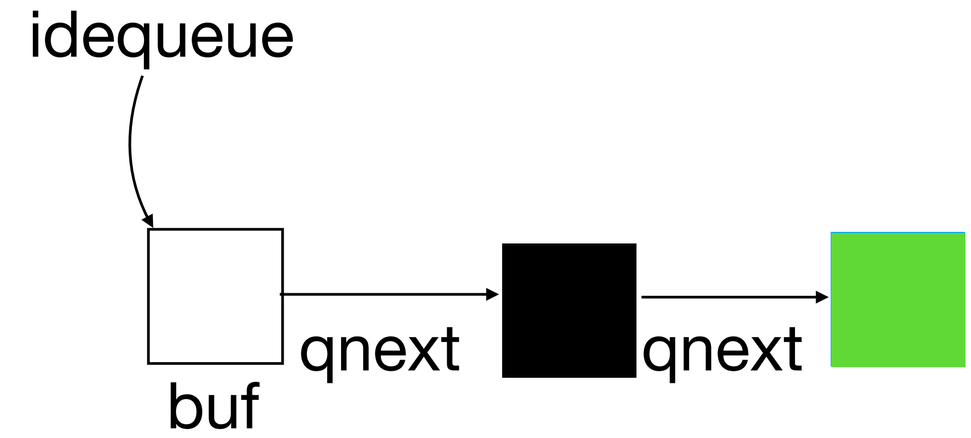
# Concurrency (single processor)

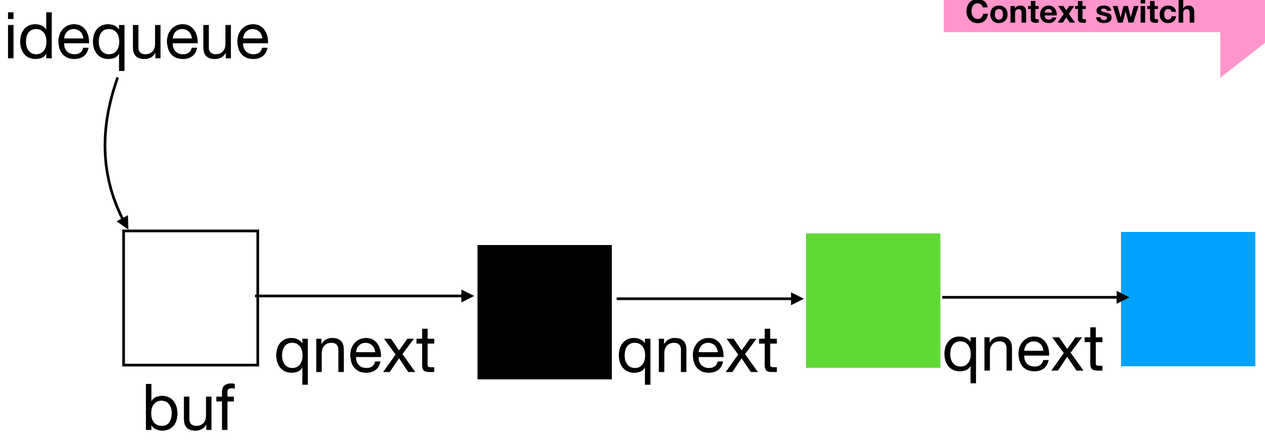
# Uniprocessor concurrency

- Software interrupts like system calls do not disable interrupts
  - While kernel is servicing system calls, timer interrupt can happen which can switch to other processes
  - If the kernel was updating its data structures, it needs to be careful to not run into races

# Race conditions

**Example: Processes reading a block in ide.c**




Context switch

```
struct buf {
  ..
  struct buf *qnext; // disk queue
  uchar data[BSIZE];
};

static struct buf *idequeue;

void iderw(struct buf *b) {
  struct buf **pp;
  b->qnext = 0;
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
  *pp = b;
  ..
}
```

# Lock implementation

```
void iderw(struct buf *b) {
  struct buf **pp;
  acquire();
  for(pp=&idequeue; *pp; pp=&(*pp)->qnext);
  *pp = b;
  ..
  release();
}
```

- Timer interrupt and hence context switch cannot happen between acquire and release

```
void acquire() {
  pushcli();
}
void pushcli(void) {
  int eflags = readeflags();
  cli();
  if(cpu->ncli == 0)
    cpu->intena = eflags & FL_IF;
  cpu->ncli += 1;
}
void release() {
  popcli();
}
void popcli(void) {
  cpu->ncli--;
  if(cpu->ncli == 0 && cpu->intena)
    sti();
}
```

# Summary

- Memory management: How to manage and isolate memory? What are memory APIs? How are they implemented?
- Processes in action: Process control block, user stack<>kernel stack, sys call handling, system calls, protection
- Scheduling: response time, turnaround time, fairness
- Uniprocessor concurrency