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

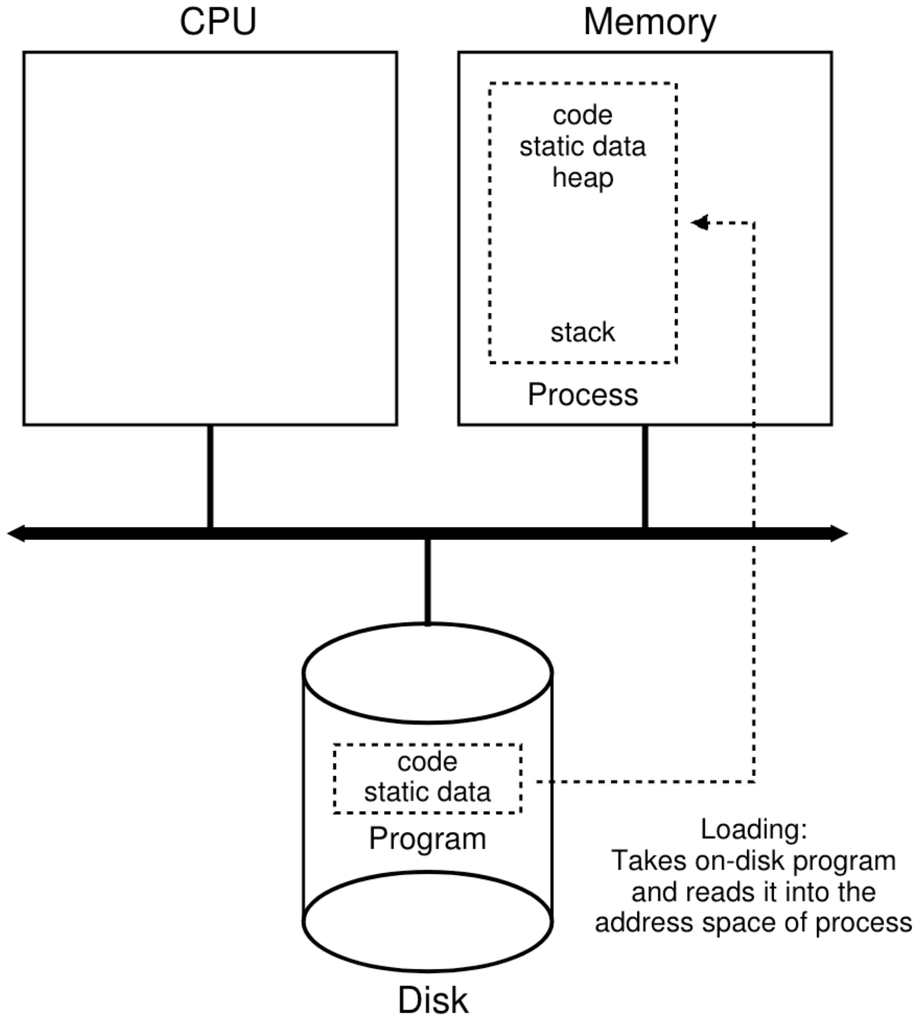


Figure 4.1: **Loading: From Program To Process**

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

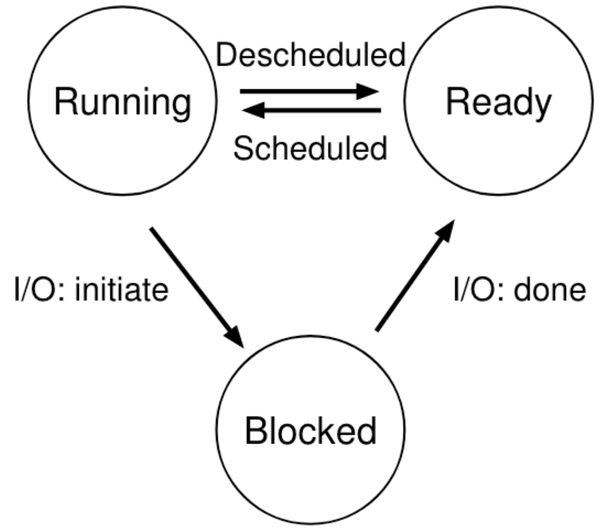


Figure 4.2: **Process: State Transitions**

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked,
5	Blocked	Running	so Process_1 runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 now done
9	Running	–	
10	Running	–	Process_0 now done

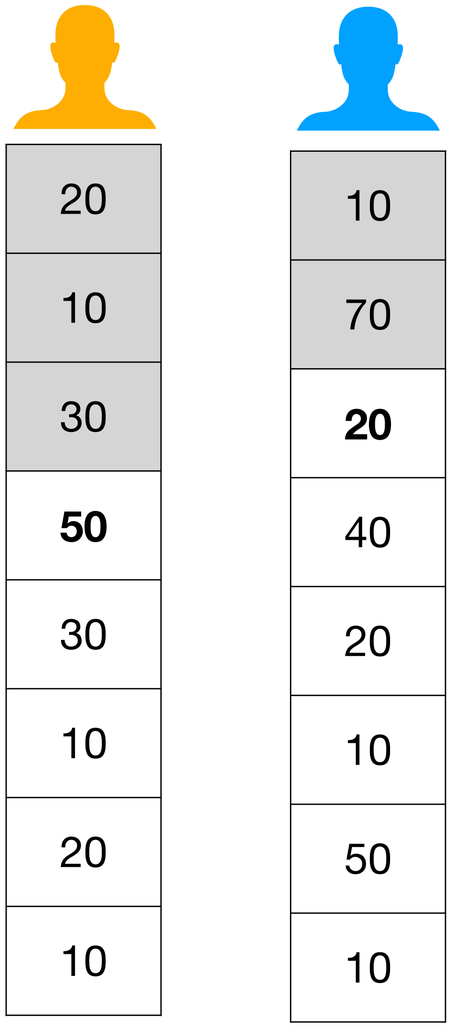
Figure 4.4: **Tracing Process State: CPU and I/O**

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

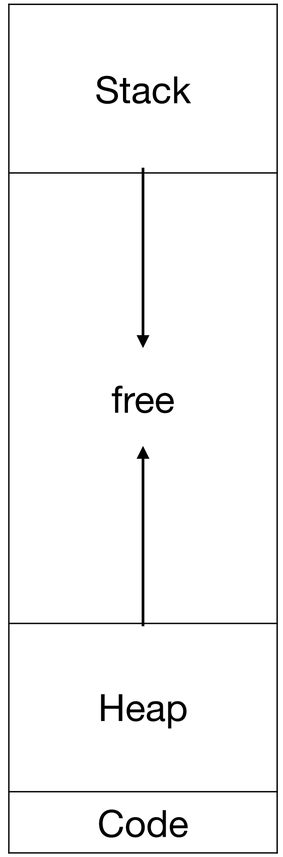
Memory isolation and management

OSTEP Ch. 13-17

Abhilash Jindal

Process Address Space

Code, Heap, Stack



Process address space

Function calling in action

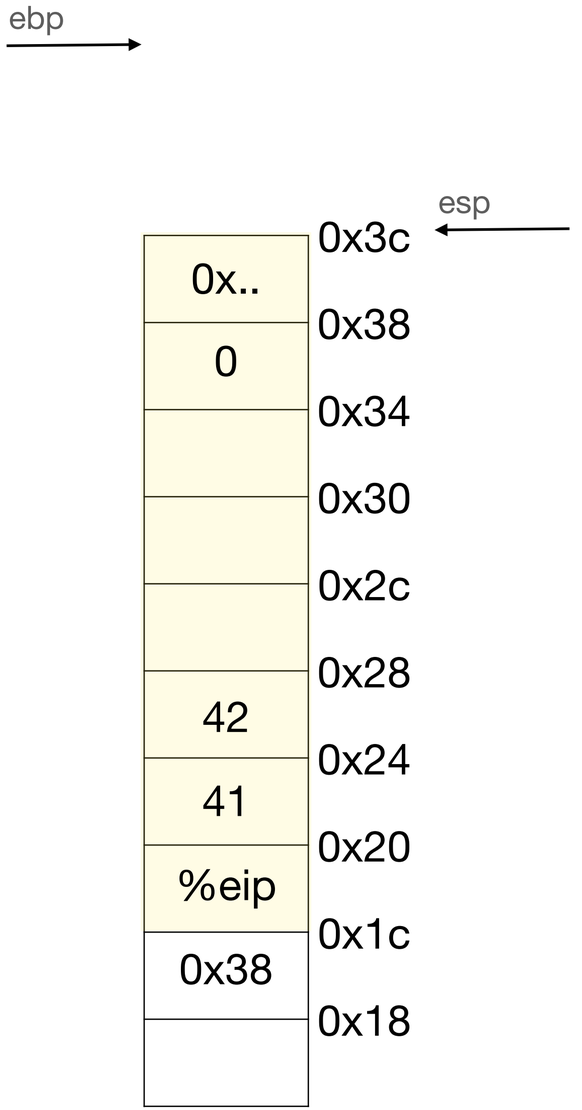
Stack

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 →

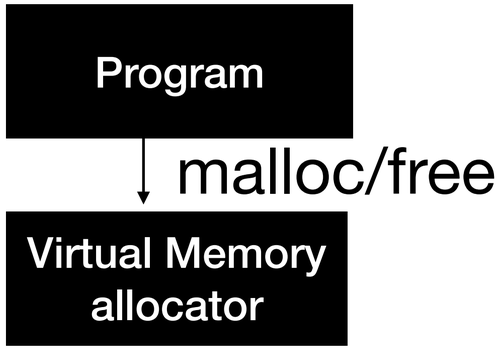


Memory APIs and bugs

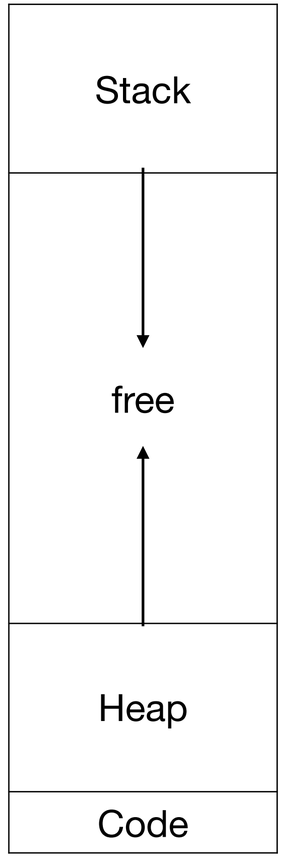
- malloc, free. va.c.
  - malloc is for dynamic allocation. Size is not known at compile time. Slower than stack allocations. Need to find free space.
- Null pointer dereference. null.c
- Memory leak. leak.c
- Buffer overflow. overflow.c
- Use after free. useafterfree.c
- Invalid free. invalidfree.c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

Memory allocator

Works with virtual memory



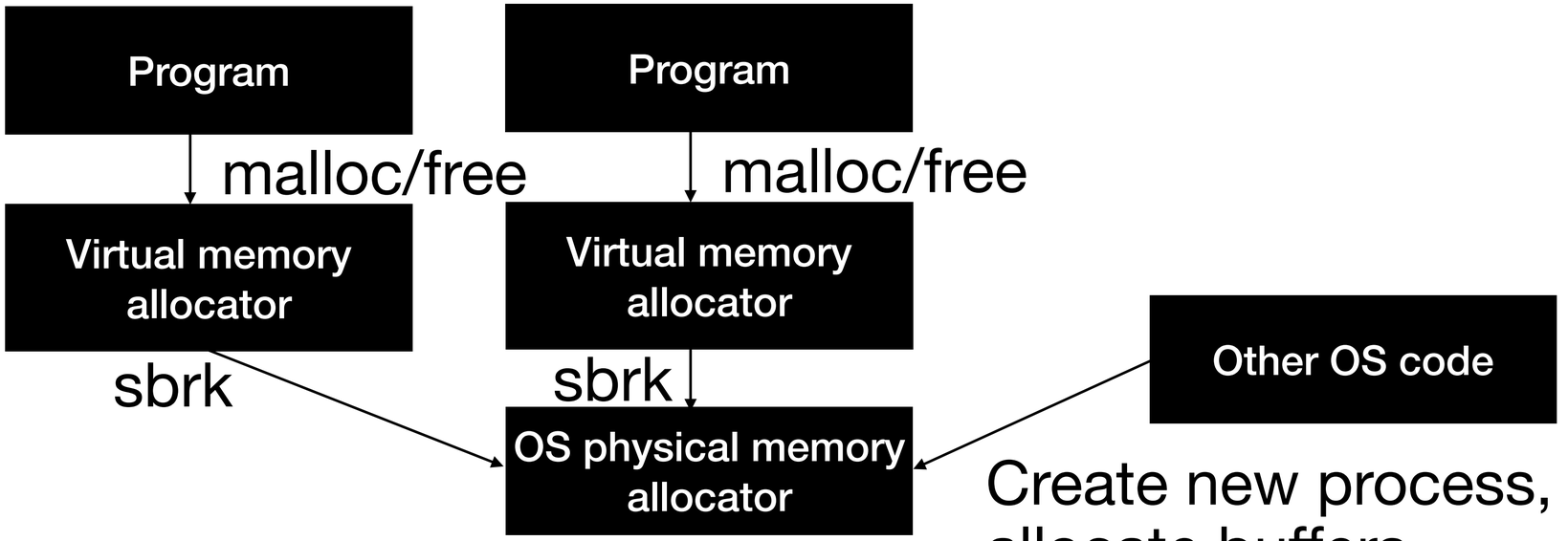
Manages heap memory

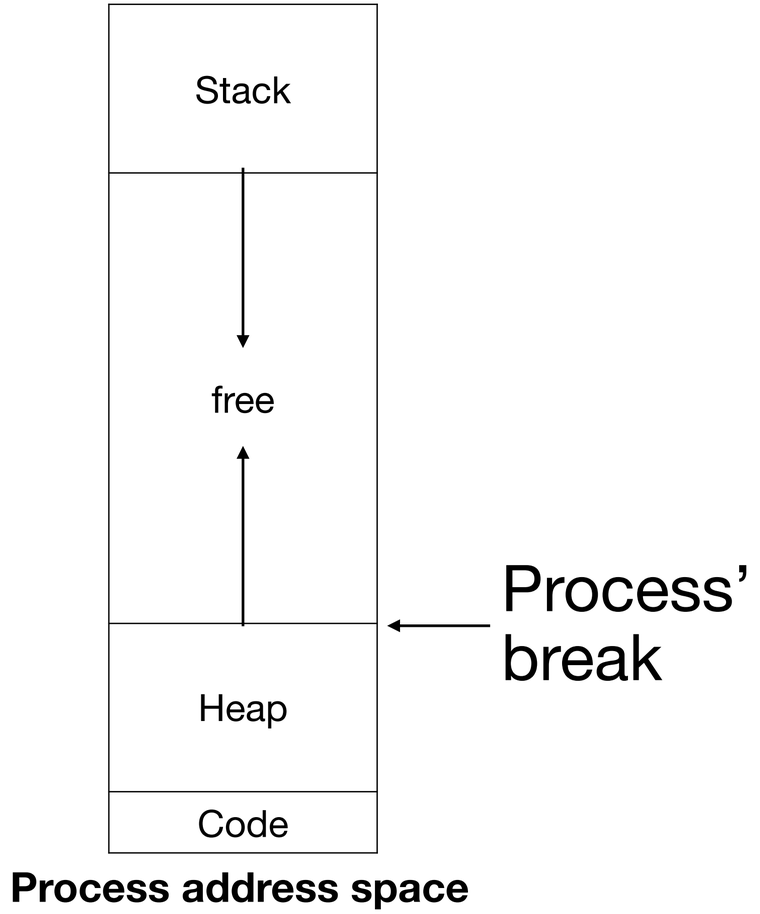


Process address space

OS memory allocator

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





Memory allocation

```
ptr=malloc(size_t size);

free(ptr);
```

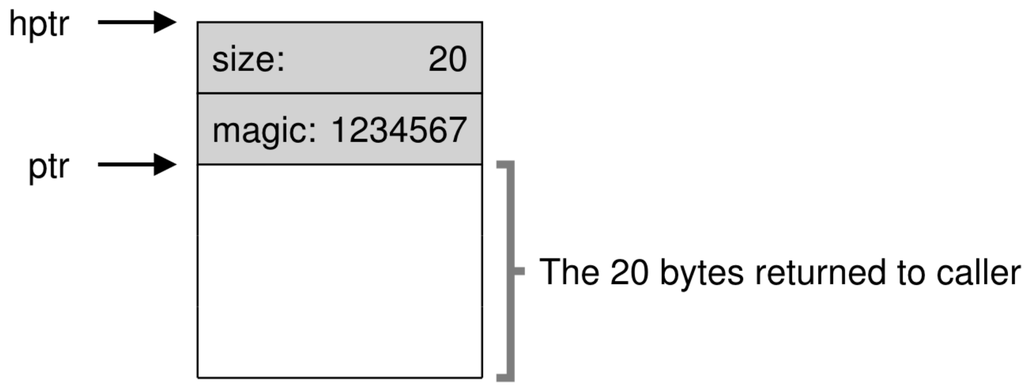
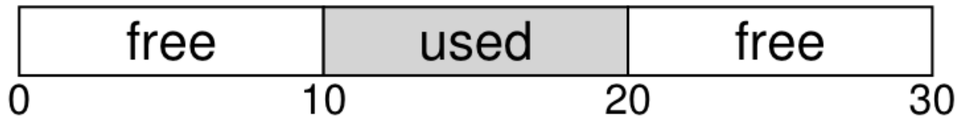


Figure 17.2: **Specific Contents Of The Header**

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)
```

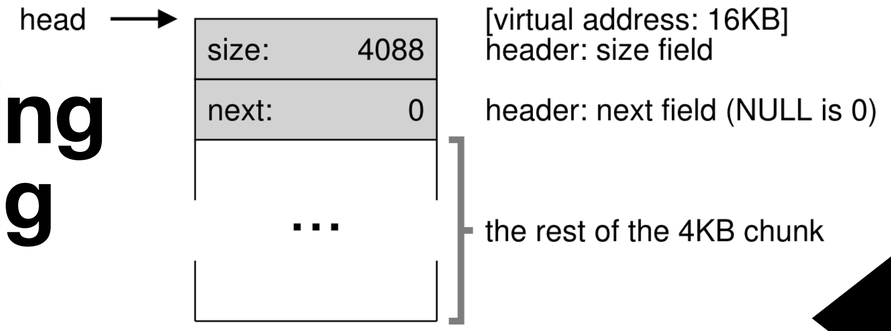


Figure 17.3: **A Heap With One Free Chunk**

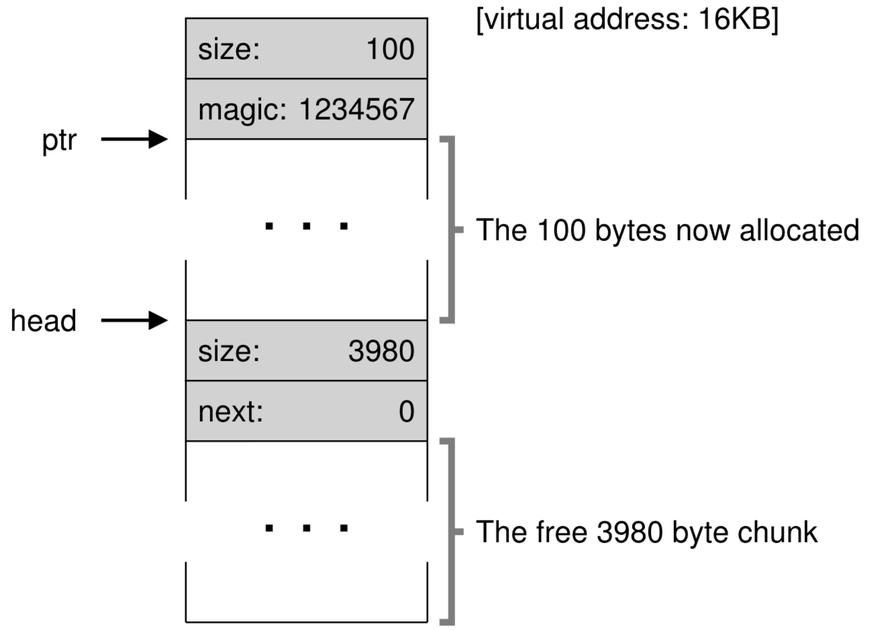


Figure 17.4: **A Heap: After One Allocation**

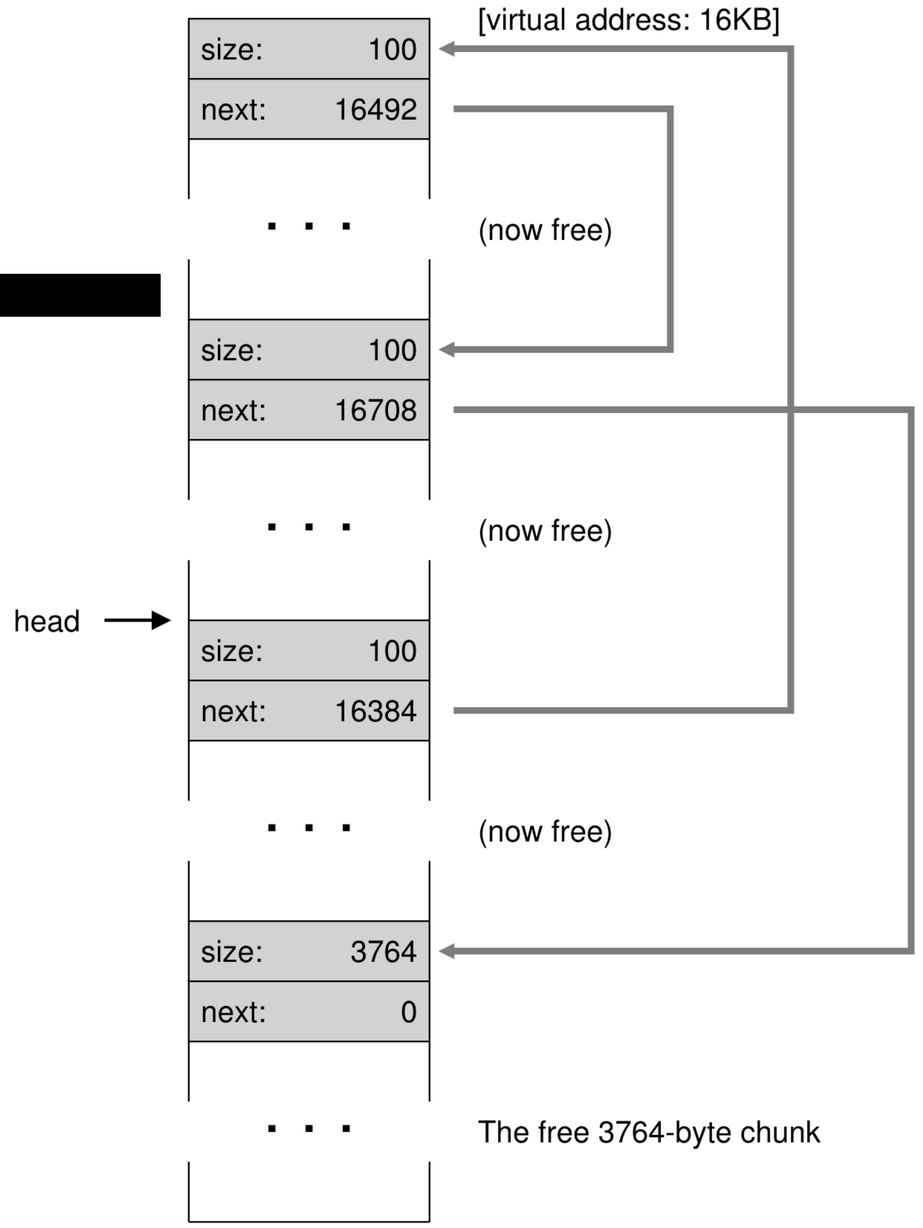


Figure 17.7: **A Non-Coalesced Free List**

Which block to allocate?

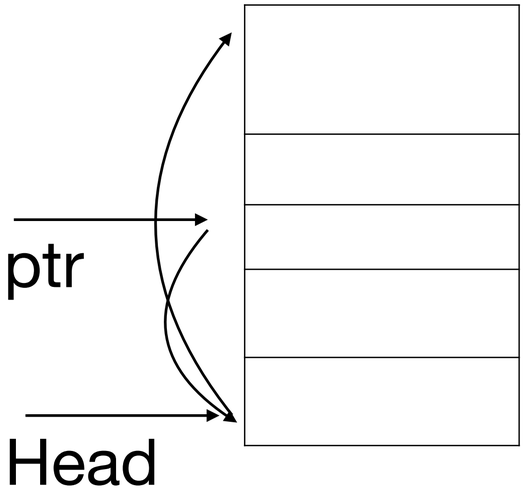
Example: malloc(15)

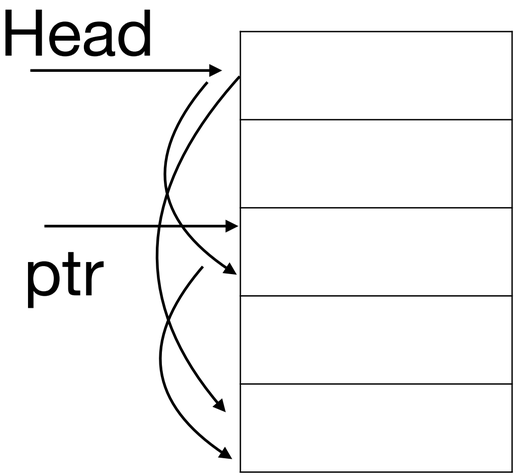
- Best fit
  - Slow. need to search the whole list
- First fit
  - Faster. (xv6: umalloc.c)
- Fragmentation
  - Example: malloc(25)

In which order to maintain lists?

- (De)allocation order

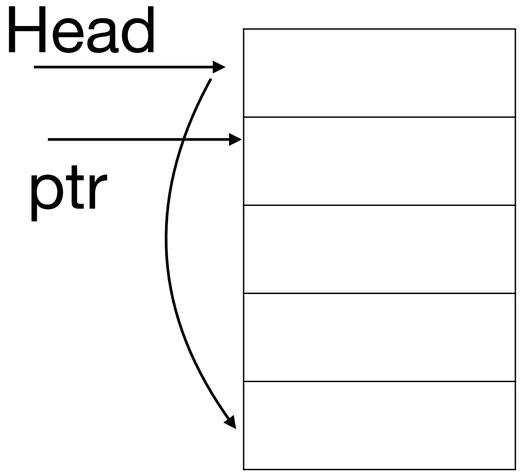
- Address order
  - Slow frees: need to traverse the free list
  - (xv6: umalloc.c)
  - Address order, first fit will allocate back-to-back allocations contiguously.
    - Due to “clustered deaths”, we may get better chances of coalescing



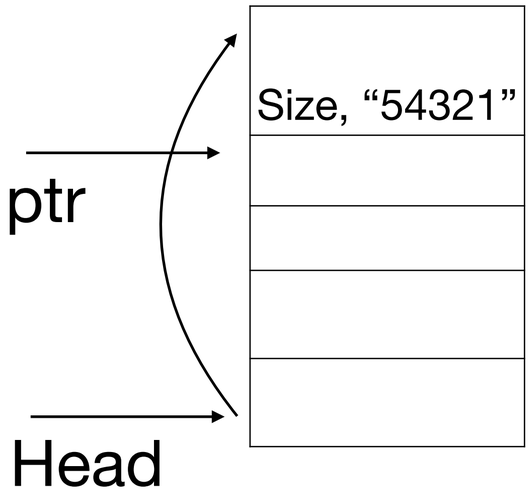


How to do coalescing?

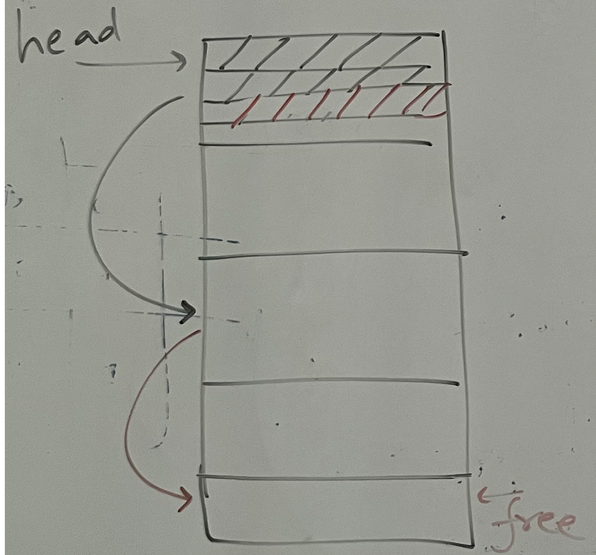
Example: free(ptr)



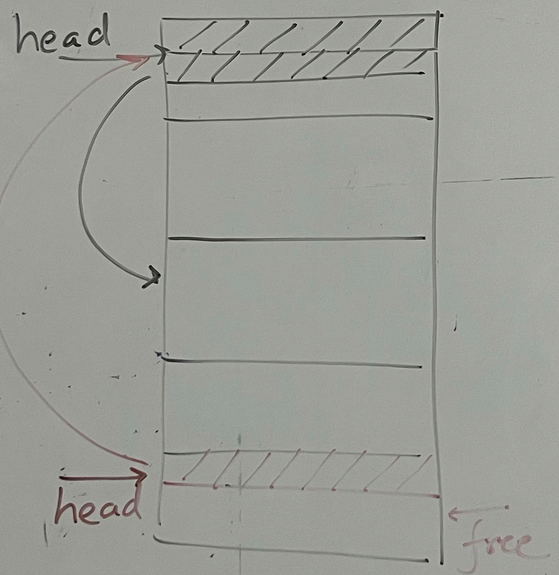
- Straightforward in address order since we are traversing the free list in address order
- In deallocation order: when an area is freed, check if the “boundary tag” is present in the footer above
- First fit and address order
  - Better chances of coalescing for clustered deaths, simpler coalescing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



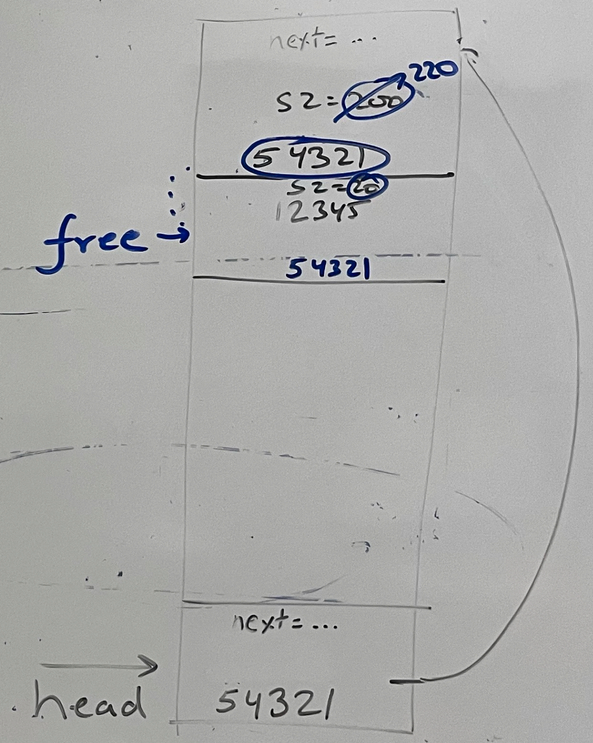
First fit
address order



First fit
(de) allocation order

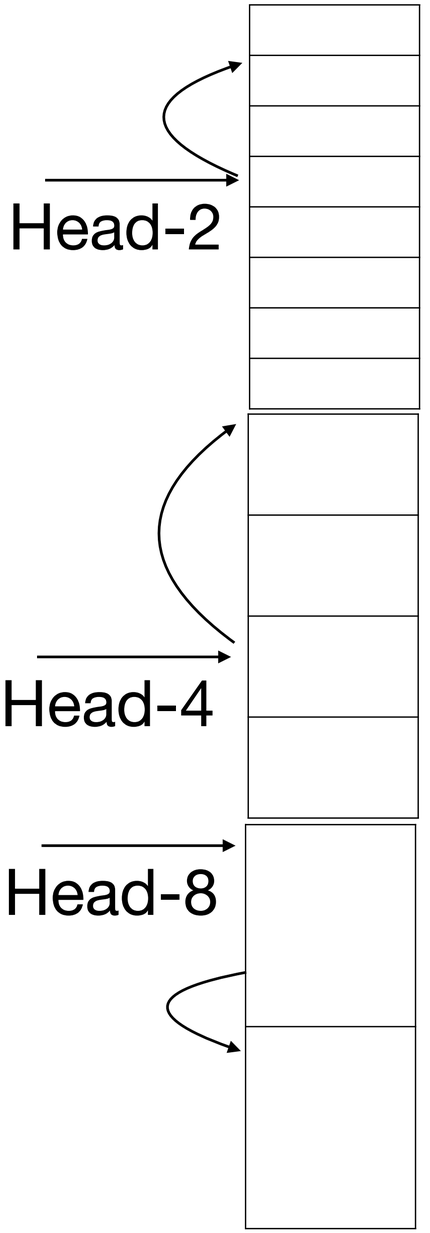


Boundary tag



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



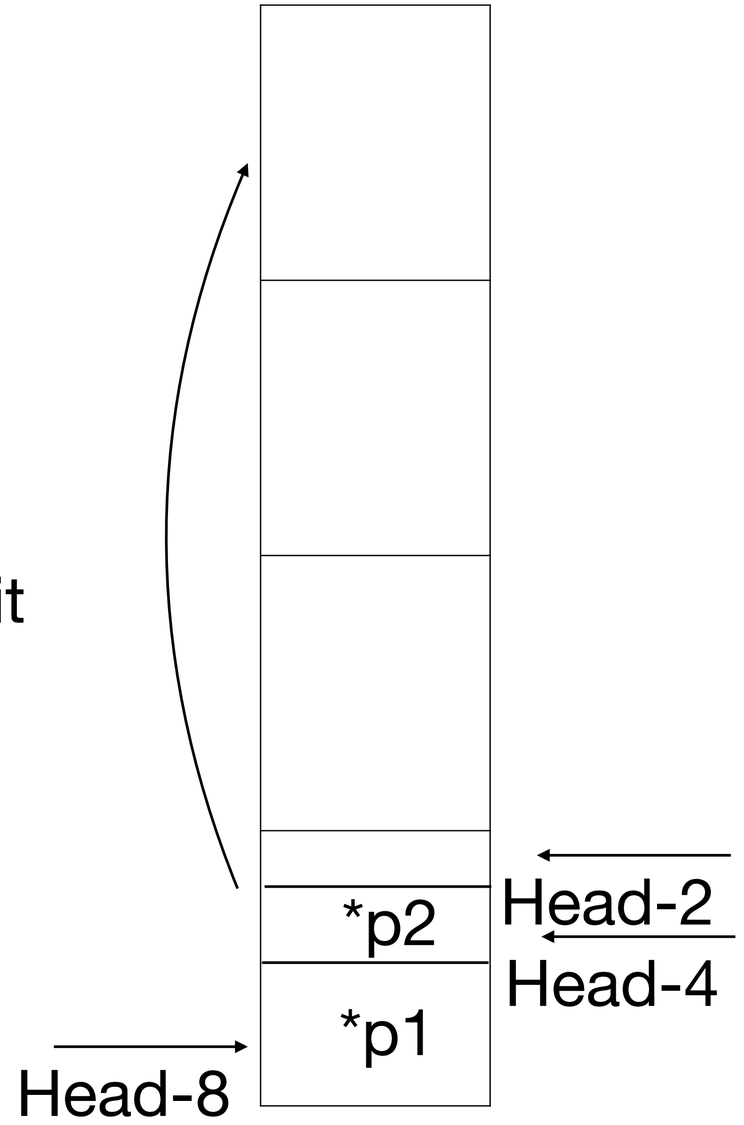
Slab allocator

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

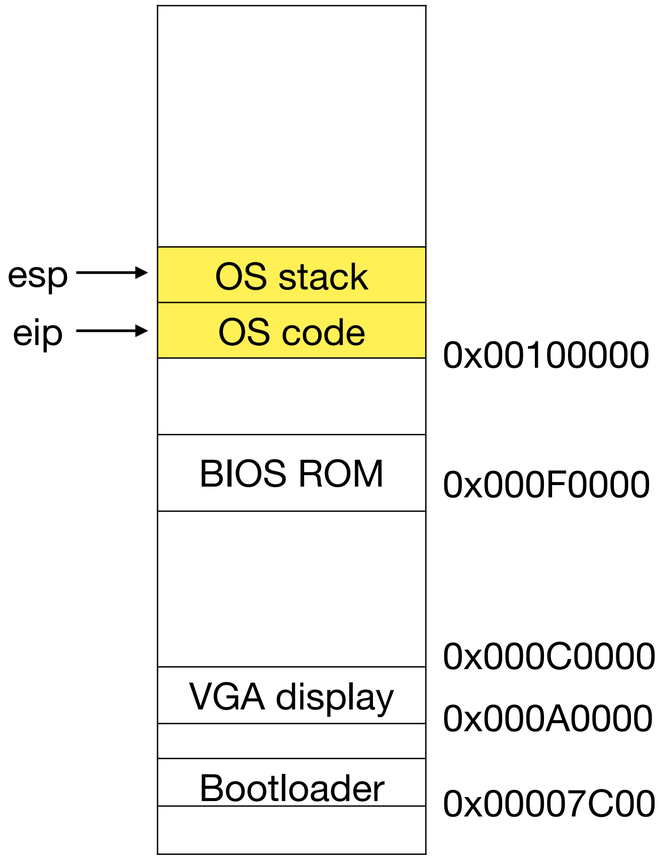
- Example:
  - p1 = malloc(3)
  - p2 = malloc(2)
- First fit with segregated lists approximates best fit
- Straightforward splitting and coalescing
- Deallocation order: fast frees
- Used in Linux kernel

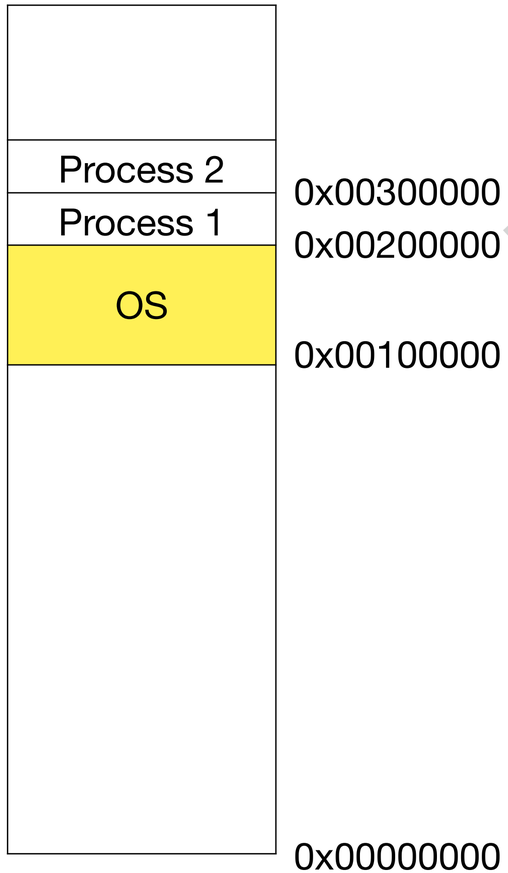


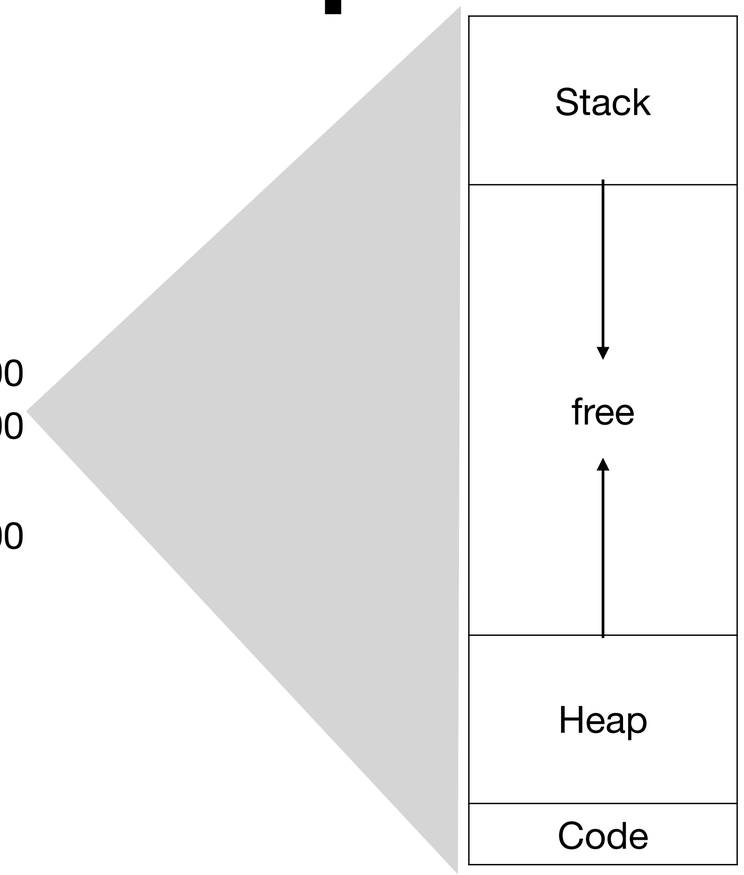
Aside: automatic memory management

- “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



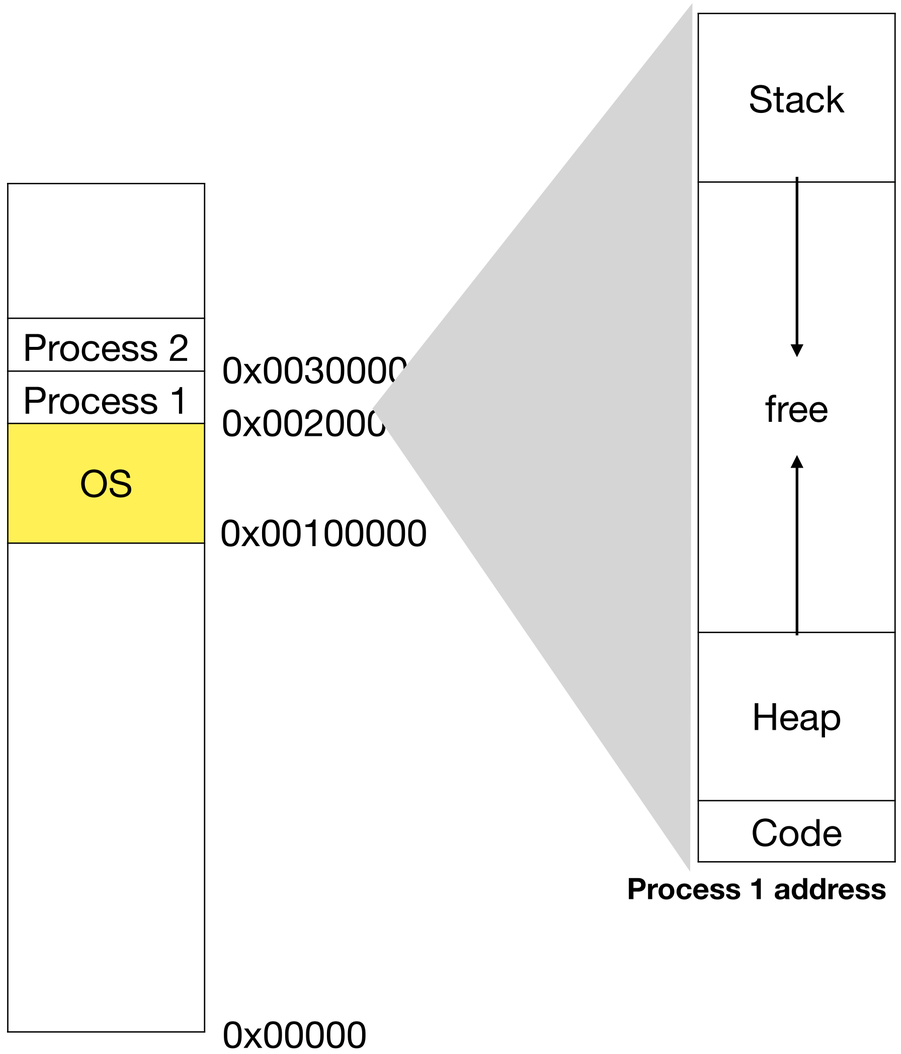




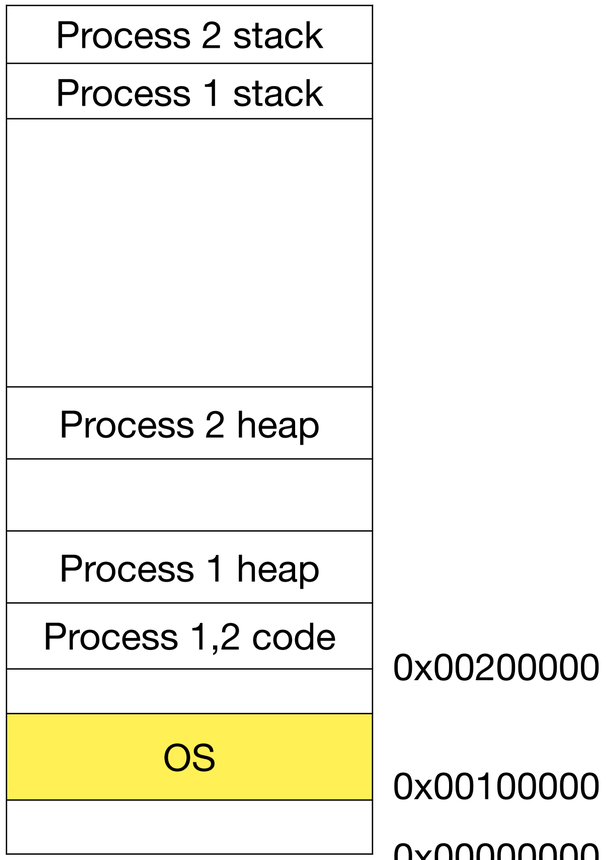
Process 1 address space

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

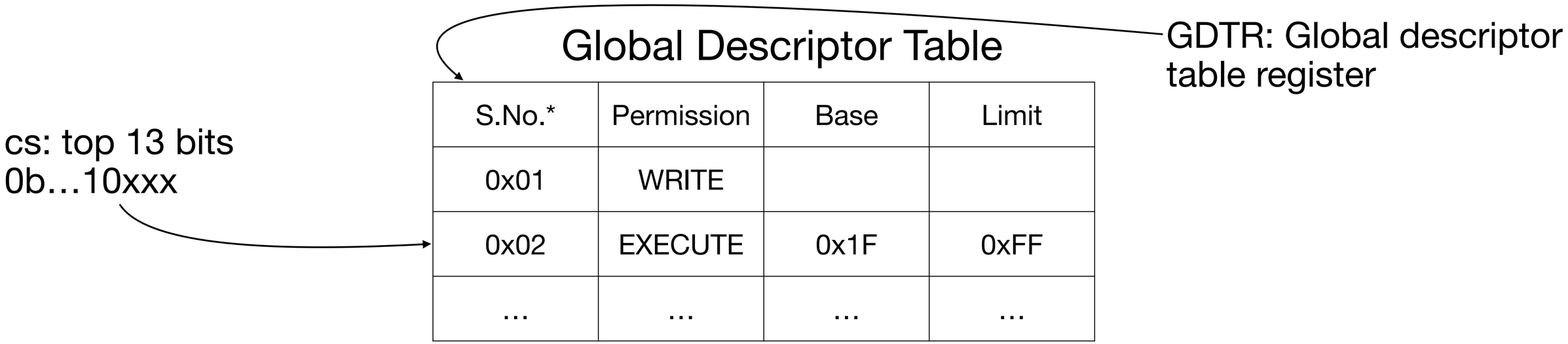
Segmentation



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

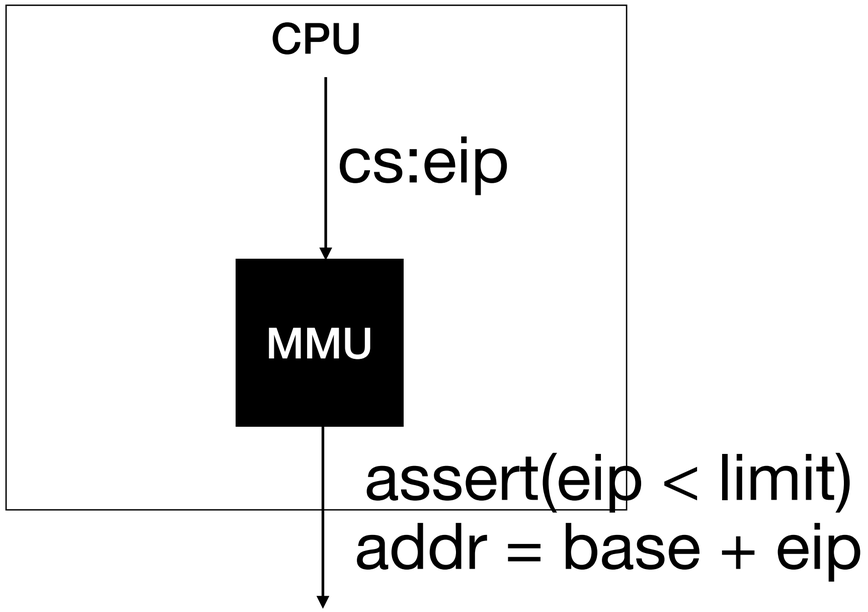


Many segments can be initialised in GDT



*: S.No. added only for illustration

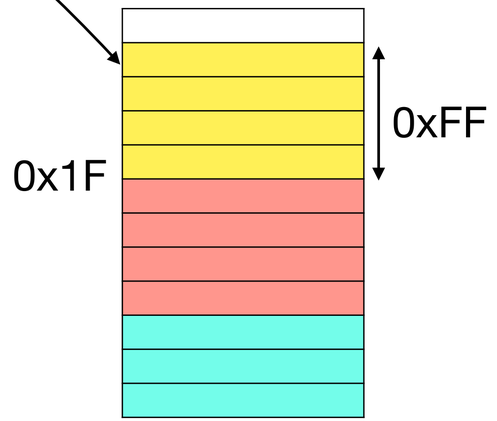
Address translation



Global Descriptor Table

cs: top 13 bits
0b…10xxx

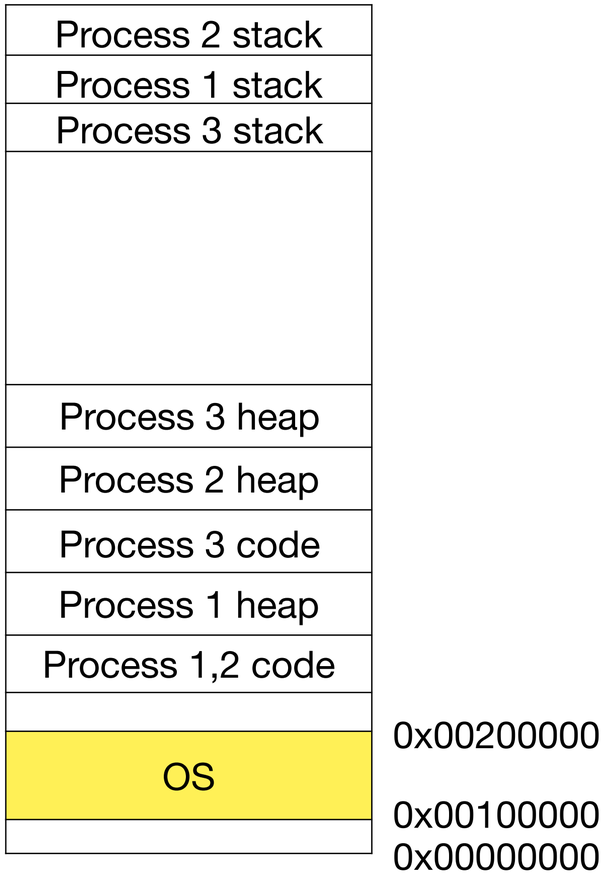
S.No.*	Permission	Base	Limit
0x01	WRITE		
0x02	EXECUTE	0x1F	0xFF
…	…	…	…



- Can “protect” different segments from each other

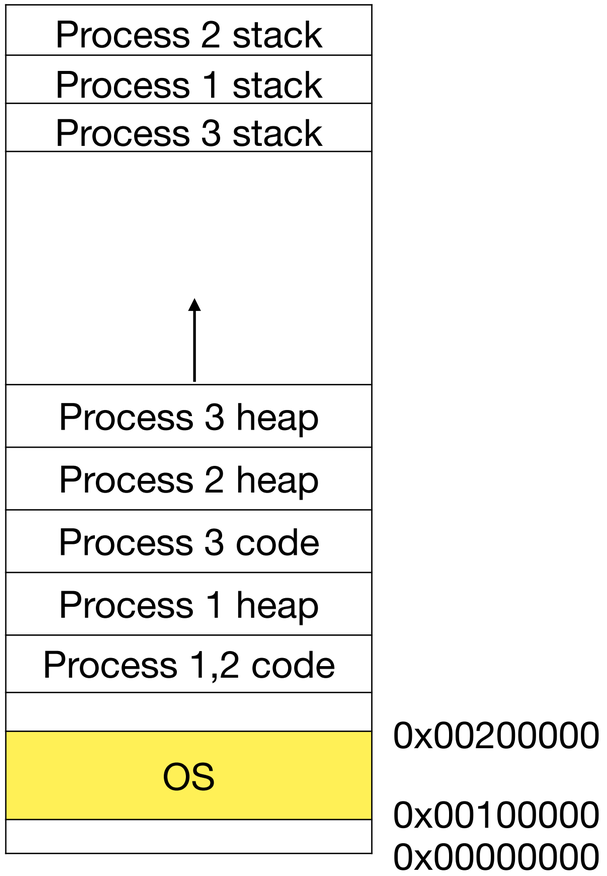
Allocating memory to a new process

- 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

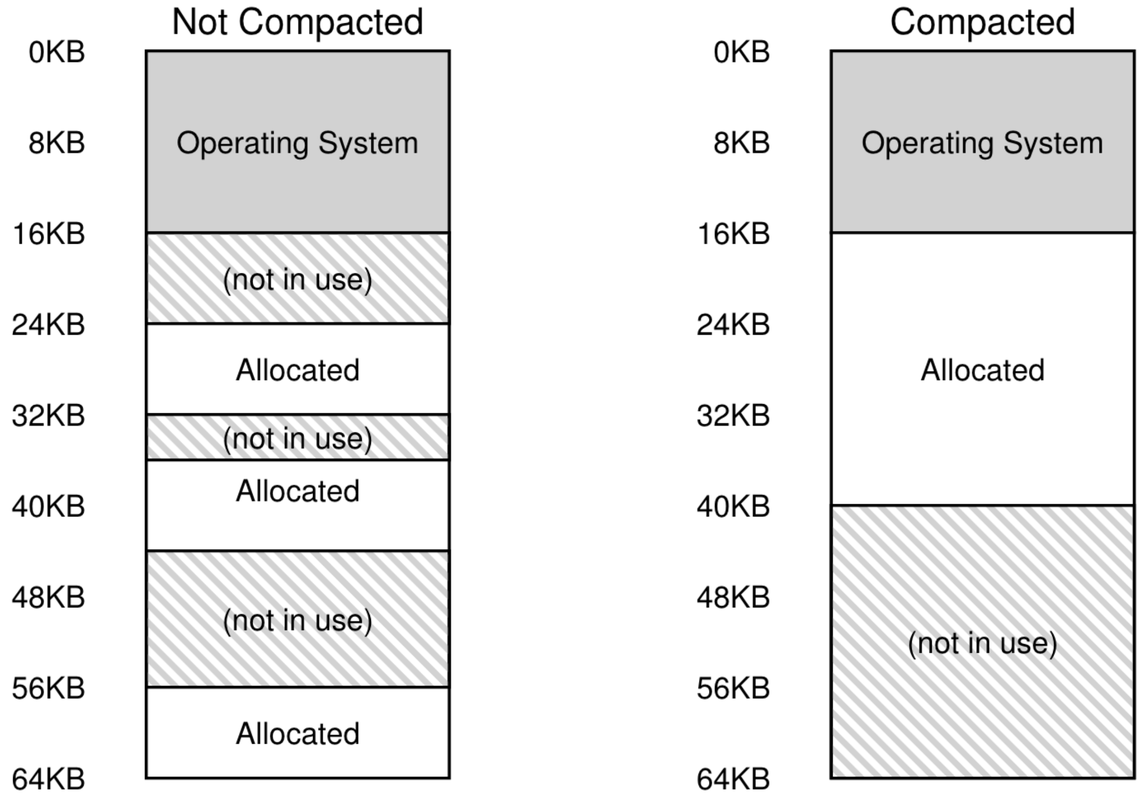
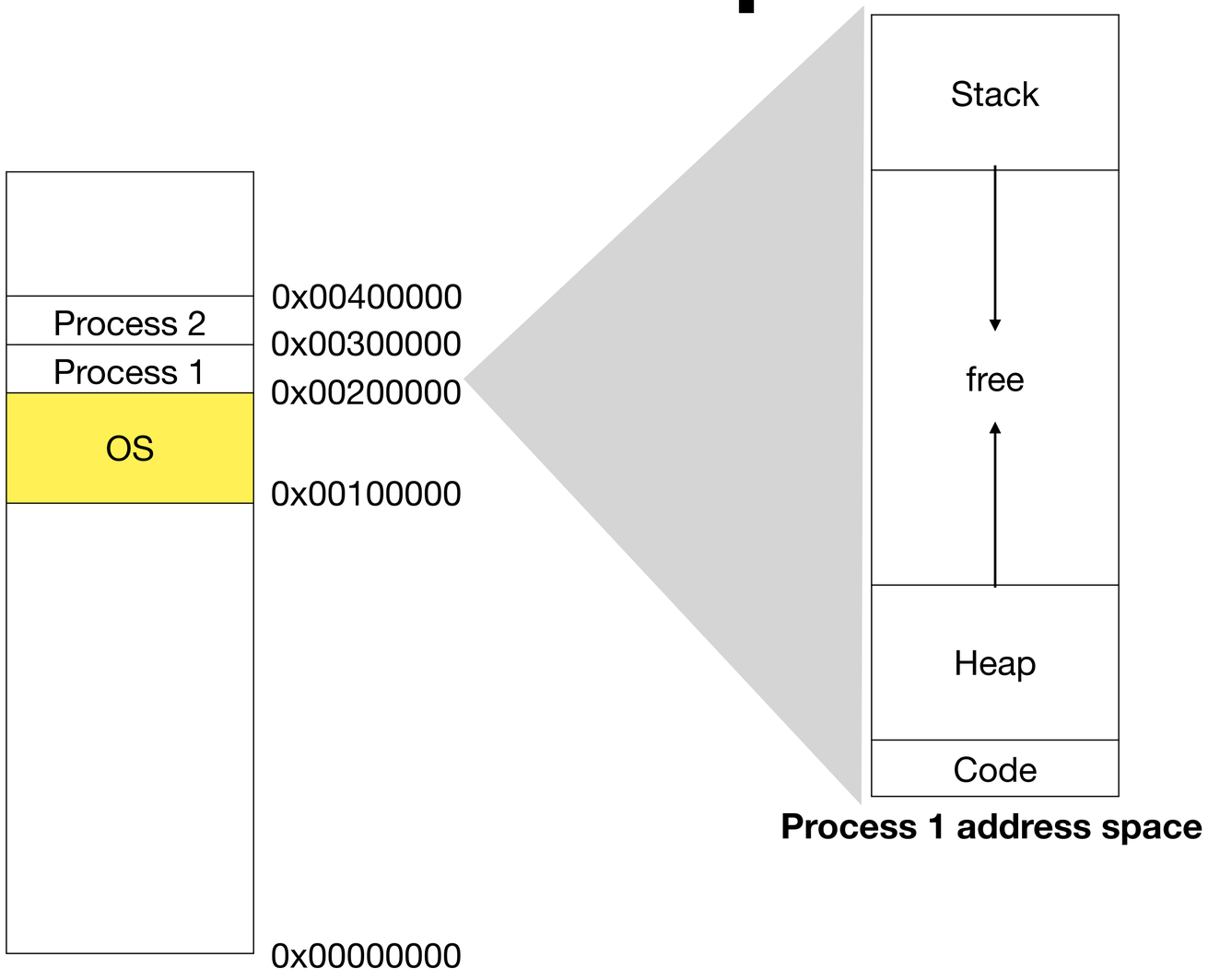


Figure 16.6: **Non-compacted and Compacted Memory**

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 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"

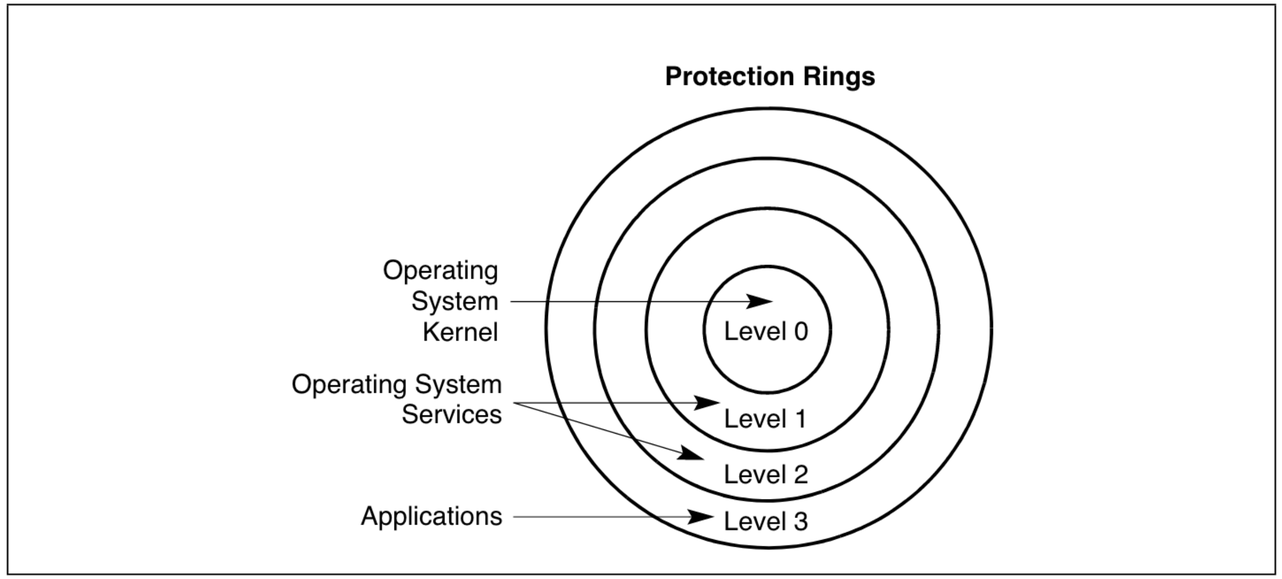
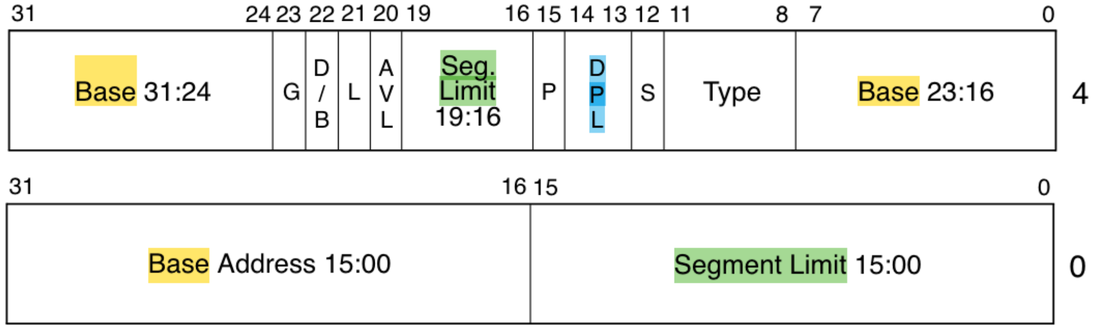


Figure 5-3. Protection Rings

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

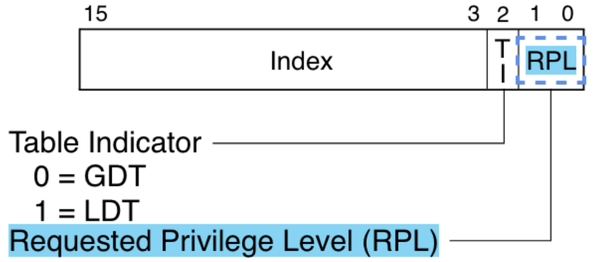


Figure 3-6. Segment Selector

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

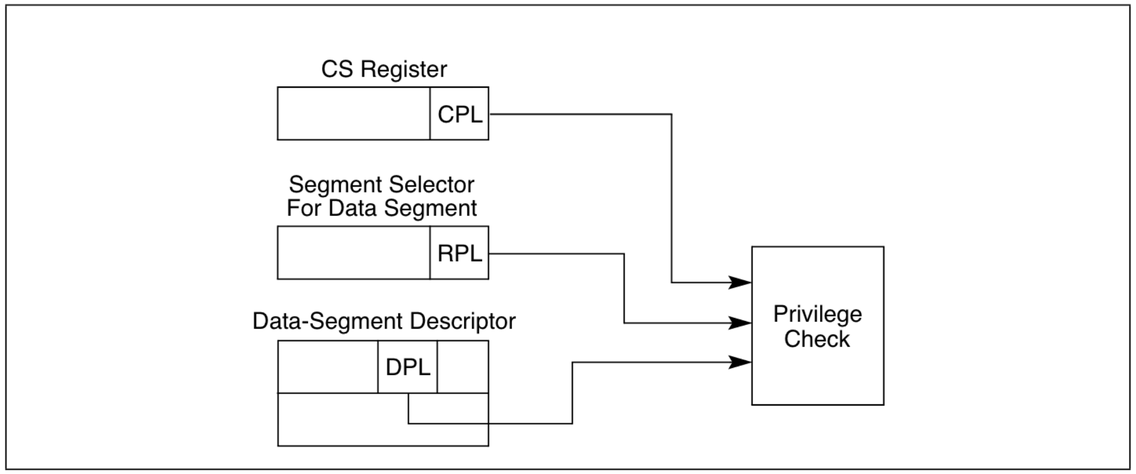


Figure 5-4. Privilege Check for Data Access

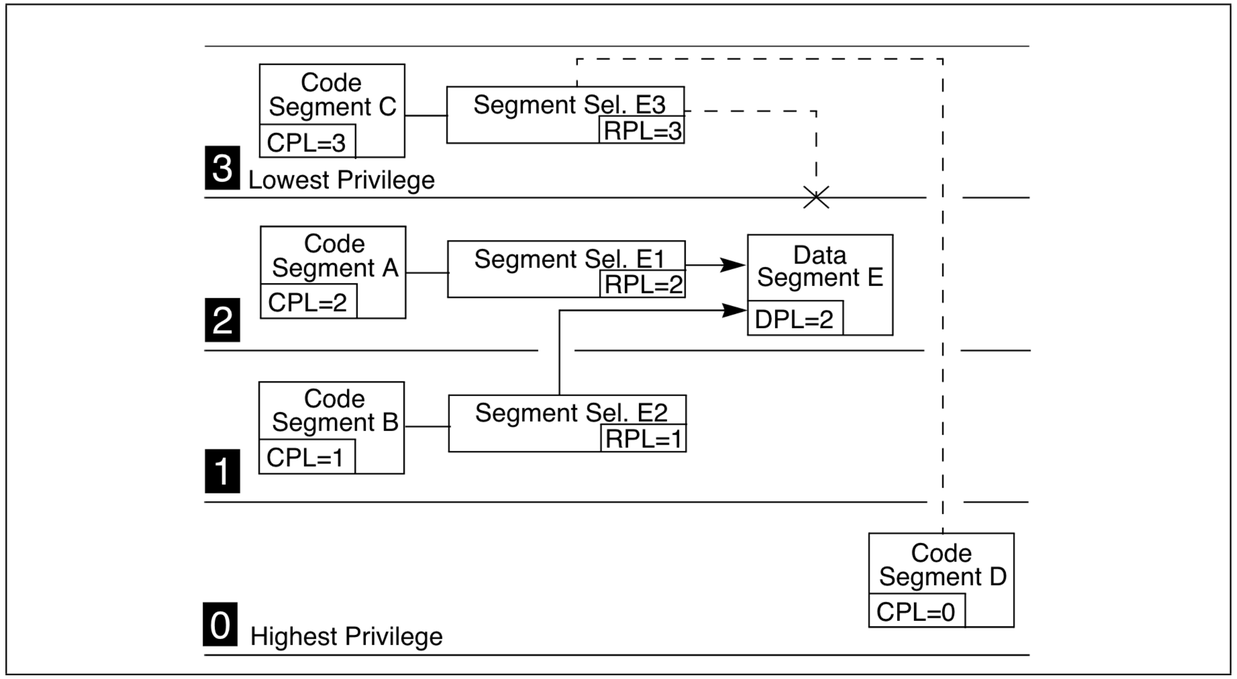
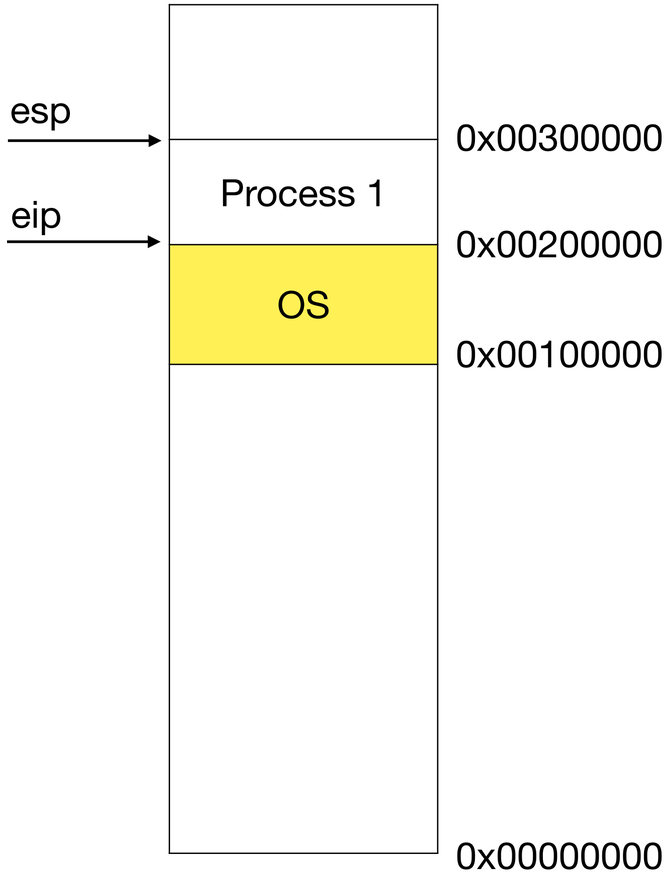


Figure 5-5. Examples of Accessing Data Segments From Various Privilege Levels

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(dst, _binary_initcode_start, ..);
  p->tf->ds, es, ss = …
  p->tf->cs = …
  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
```