Lecture 9: Memory

Professor G. Sandoval

Today

- Address Space
- Swapping
- Managing Free Memory
- XV6 Address Spaces

Memory

- RAM is one of the main **resources** managed by an operating system
- RAM is *volatile* storage (does not persist across reboots)
- The portion of the OS that allocates, frees, and tracks the usage of RAM is the *memory manager*

In Ancient Times: No Abstraction

- Early computers had no abstraction for memory
- You ask for data at address 0x1234, you get the data stored at physical memory location 0x1234
- This is often called the *physical* memory model because every address refers directly to a physical location in memory

Physical Memory Model Organization

- Even with such a simple model there are still decisions to be made:
    - Where do we put the OS code?
    - Where do user programs go?
    - If there is code in ROM, where does it live?

No Memory Abstraction

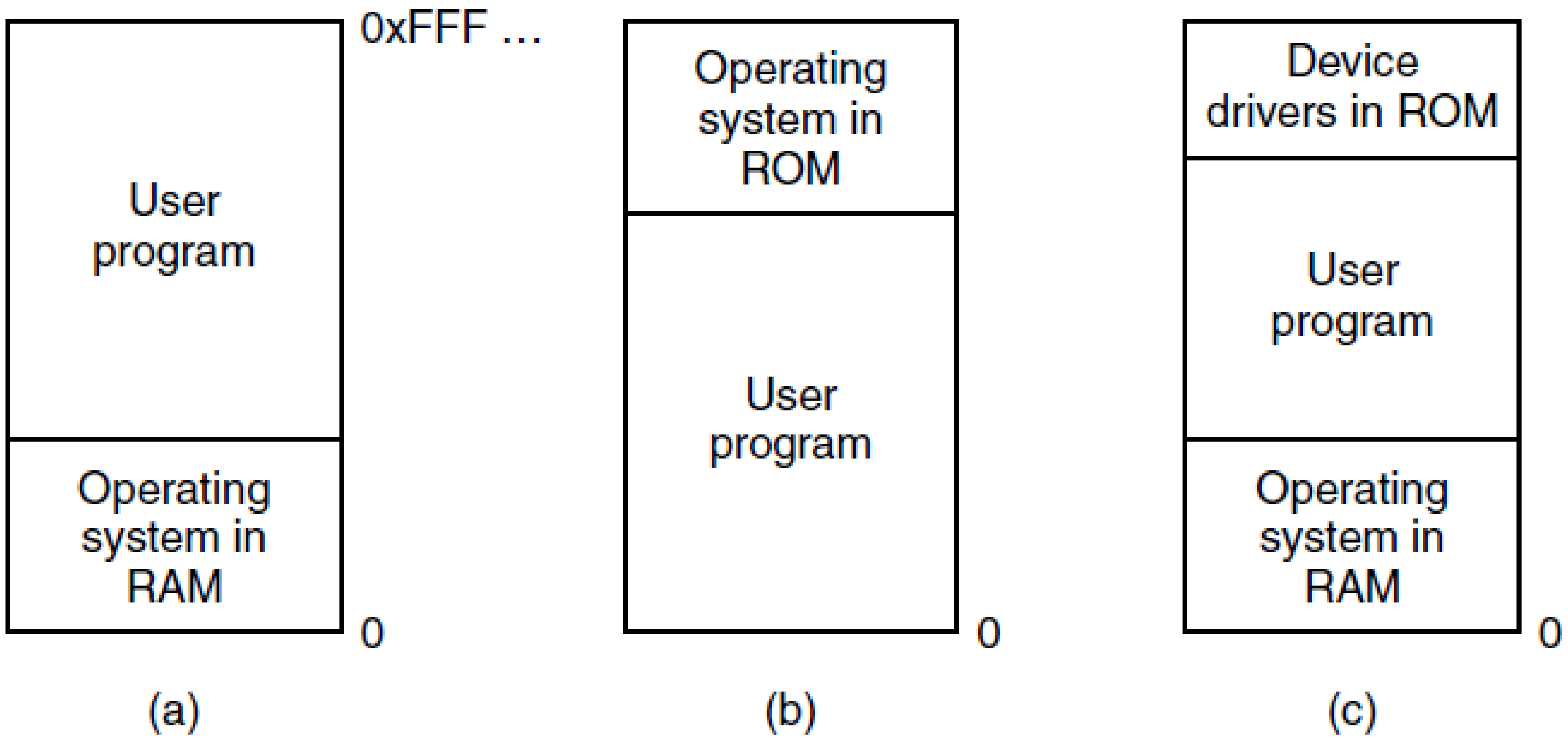


Figure 3-1. Three simple ways of organizing memory with an operating system and one user process. Other possibilities also exist

No Abstraction: Downsides

- Can't really have two independent programs running at the same time
- Each would have to know explicitly about what memory was in use by the other – requiring cooperation
- Note that it *is* possible to have multiple threads with flat memory
  - Threads always share the same address space anyway

Running Multiple Programs Without a Memory Abstraction

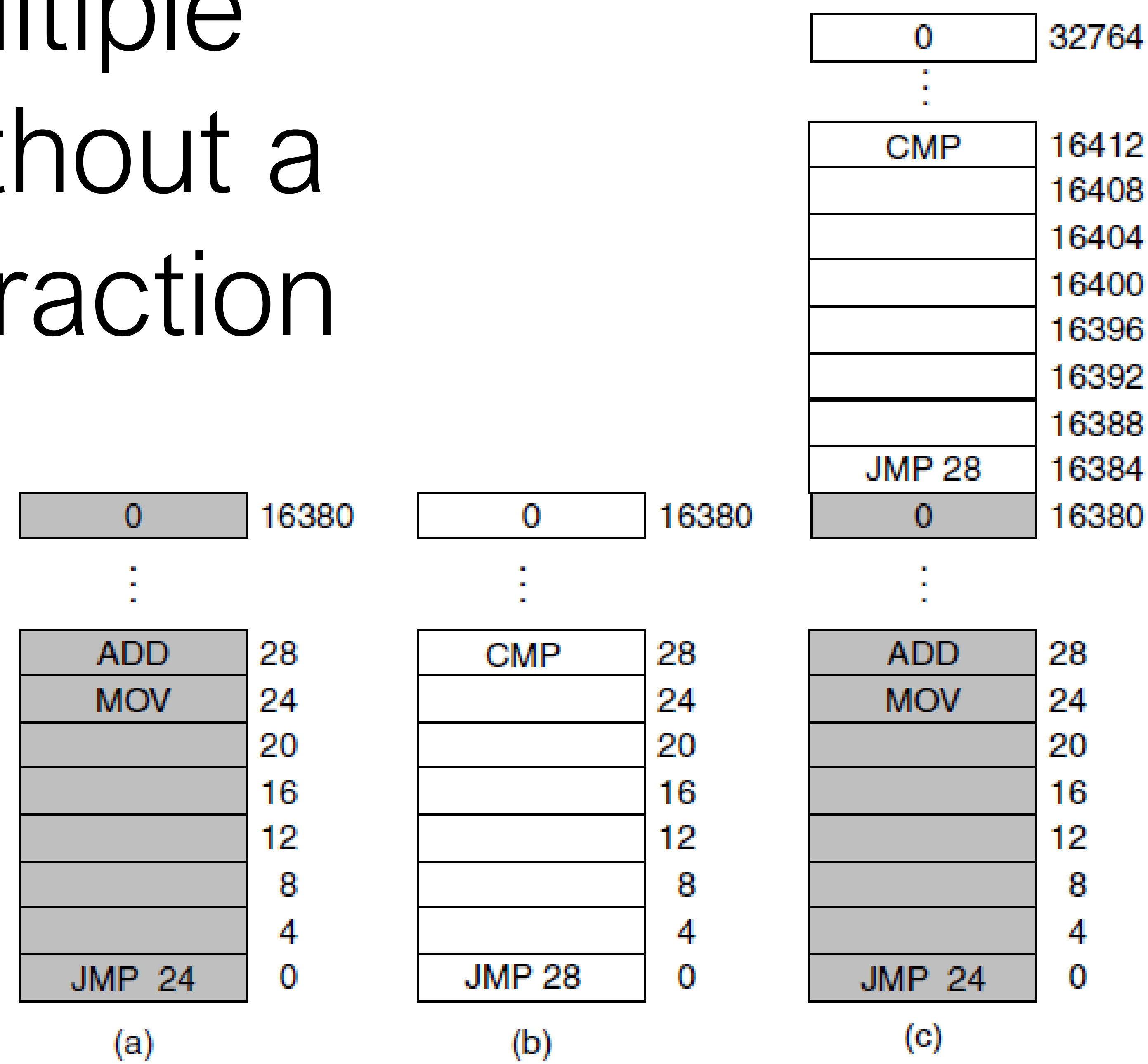


Figure 3-2. Illustration of the relocation problem. (a) A 16-KB program. (b) Another 16-KB program. (c) The two programs loaded consecutively into memory.

Static Relocation

- The workaround for this on the IBM 360 was to *statically relocate* the program when it was loaded

- Maintain a list of all the places in the program where absolute addresses were used (*relocations*)

- Modify them so that they match the program's new load address

- This technique is actually alive and well today: *shared libraries* used by a program may have to be statically relocated before they are loaded

Memory Abstractions

- The set of addresses a program can refer to is called its *address space*
- We have seen that you can get into trouble if you all programs have the same address space
  - No protection from each other – errors in one program can cause damage to others
  - Programs must be written cooperatively, knowing about where others are located in memory
- We would like to create an abstraction, so that each process has a *private address space:* make 0x1234 in Program A different from 0x1234 in Program B

Segmentation

- An early way of providing separate address spaces was *hardware segmentation*
- CPU gets extra *base* and *limit* registers
- Each time a memory address is referenced, the CPU transparently adds the *base* to it and verifies that *base+address* ≤ *limit*
- Downside: memory access becomes slightly slower because of the additional addition

Base and Limit Registers

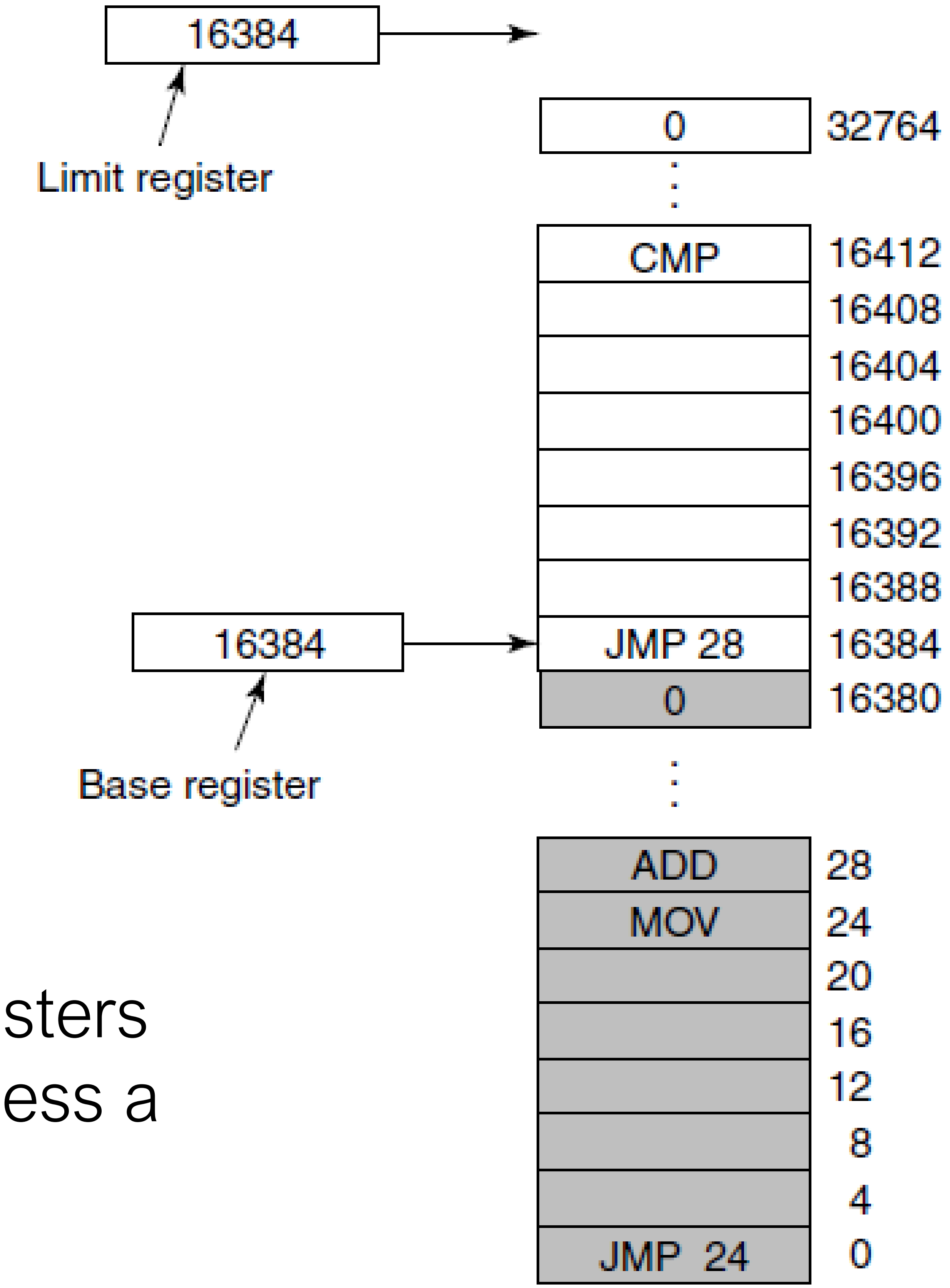


Figure 3-3. Base and limit registers can be used to give each process a separate address space.

Segmentation in x86

- We saw when we went over assembly that the x86 has 6 segment registers
- This allows programs to have different segments for code, data, etc.
- In protected mode, different segments can also have limits, protection

Segmentation in 64-bit x86

- On 64-bit x86, these have all been eliminated except FS and GS, and even there only base can be set (no limits or protection)
- Why? Segments have fallen out of fashion in favor of *virtual memory*
- Why keep FS and GS? Turns out OSes decided they wanted to use them for per-CPU data structures
    - For example, in Windows, FS+0x124 points to the thread running on the current processor
    - XV6 uses them for the current cpu & proc

XV6 GS usage

```c
// Per-CPU variables, holding pointers to the
// current cpu and to the current process.
// The asm suffix tells gcc to use "%gs:0" to refer to cpu
// and "%gs:4" to refer to proc.  seginit sets up the
// %gs segment register so that %gs refers to the memory
// holding those two variables in the local cpu's struct cpu.
// This is similar to how thread-local variables are implemented
// in thread libraries such as Linux pthreads.
extern struct cpu *cpu asm("%gs:0");       // &cpus[cpunum()]
extern struct proc *proc asm("%gs:4");     // cpus[cpunum()].proc
```

Today

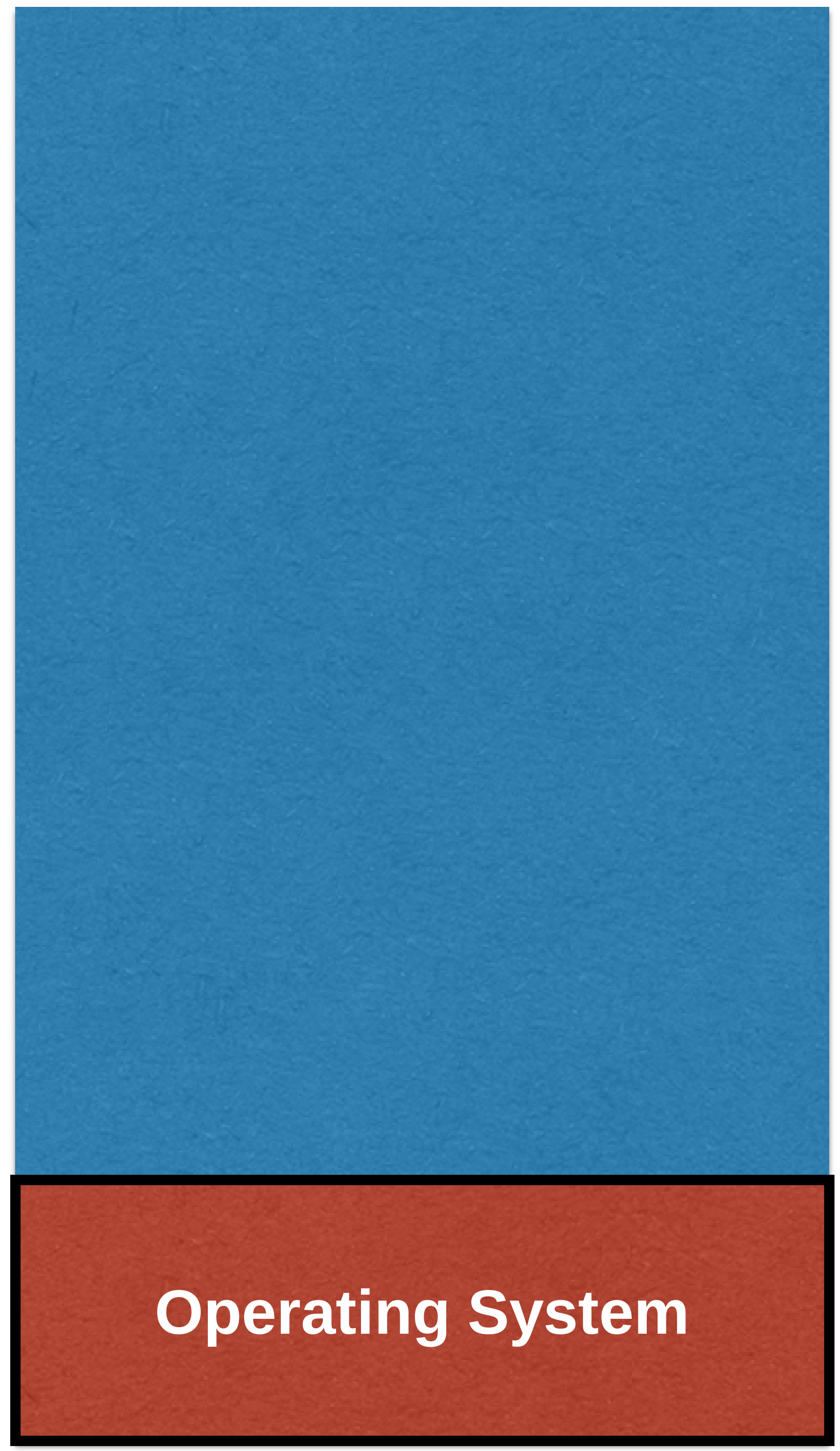
- Address Space
- Swapping
- Managing Free Memory
- XV6 Address Spaces

Swapping

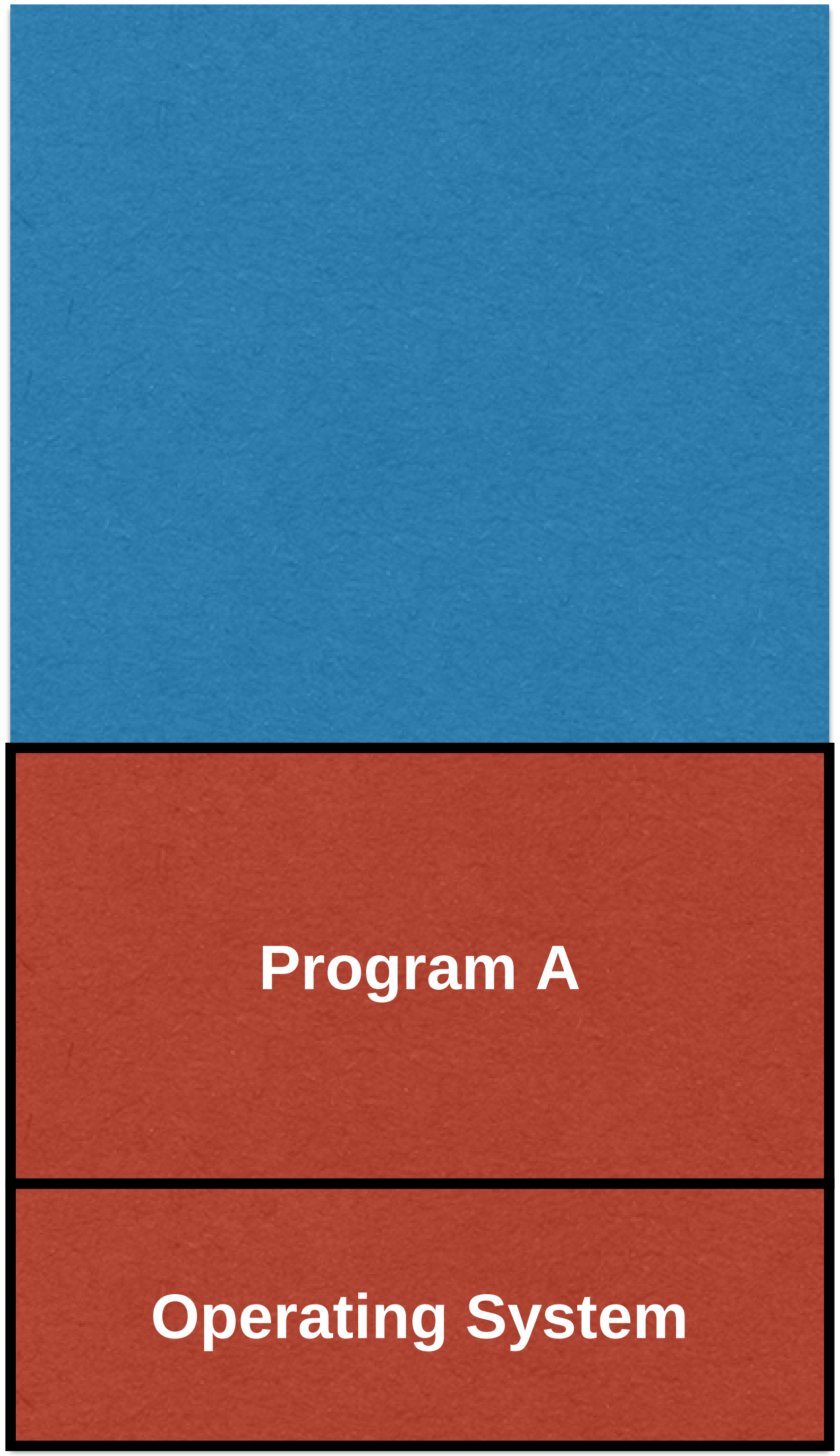
- We may not have enough RAM to keep all the programs we're running in memory at once

- One strategy to get around this is to move programs from memory to disk when they're not being used (*swapping*)

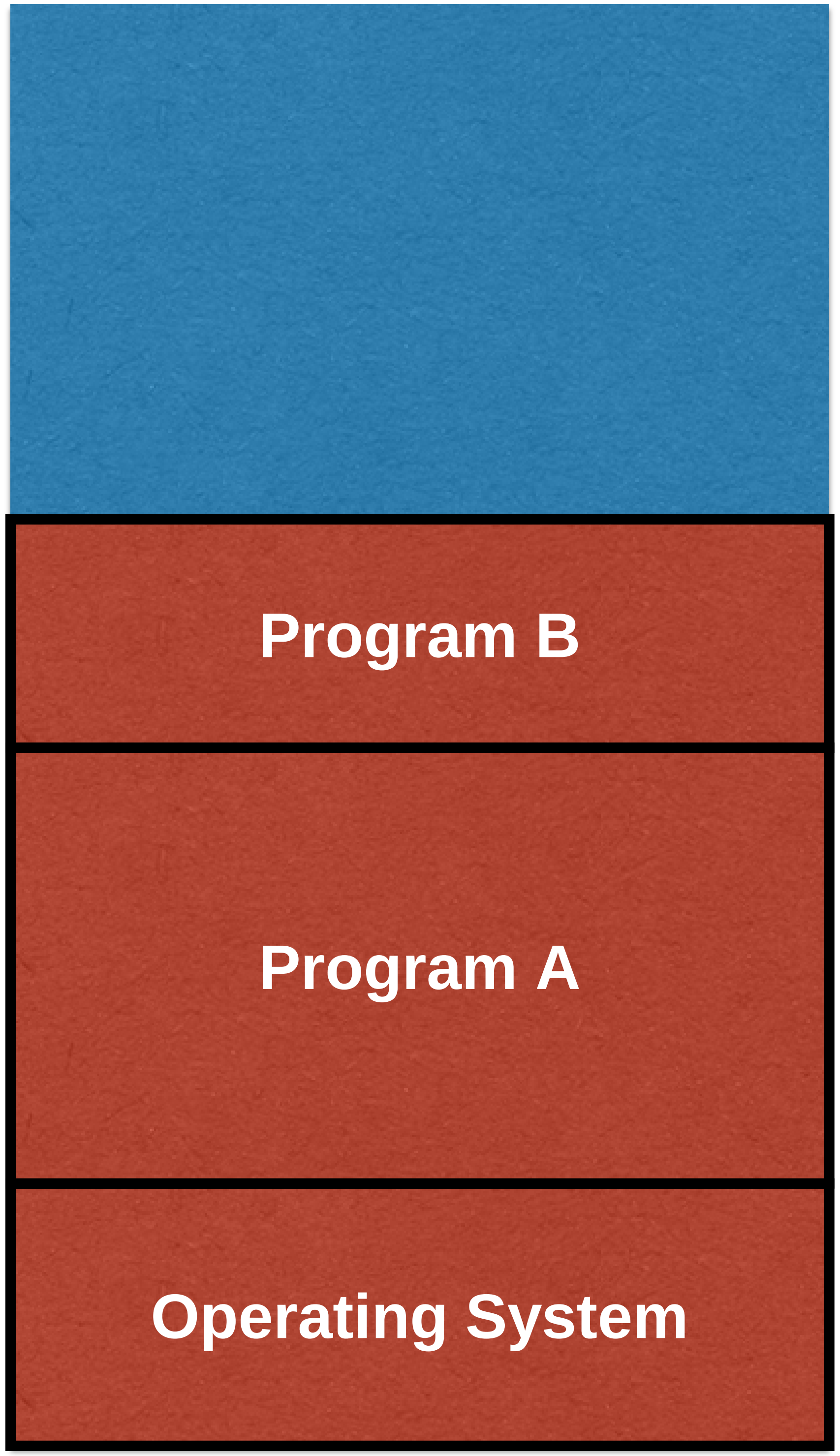
Swapping



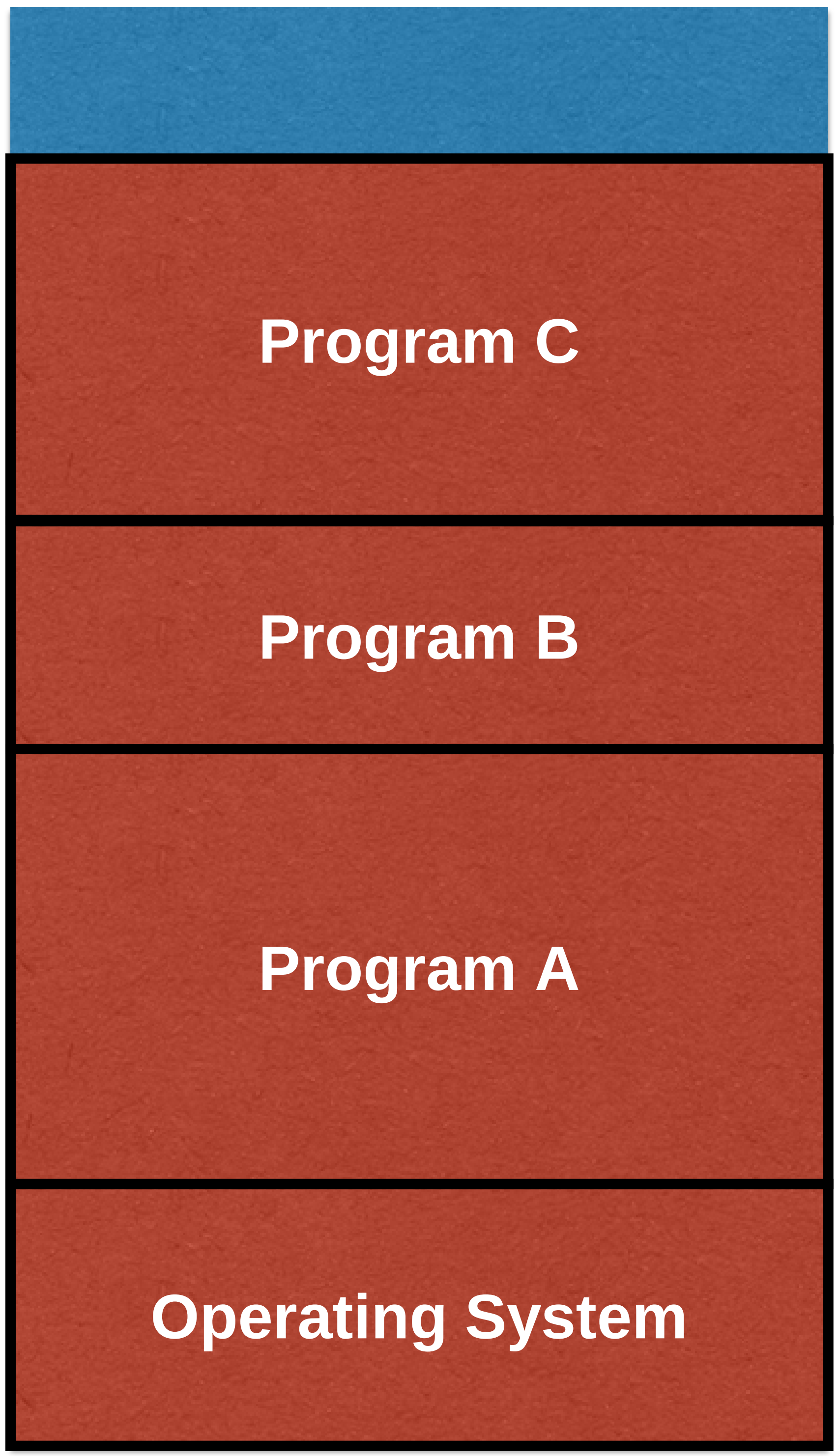
Swapping



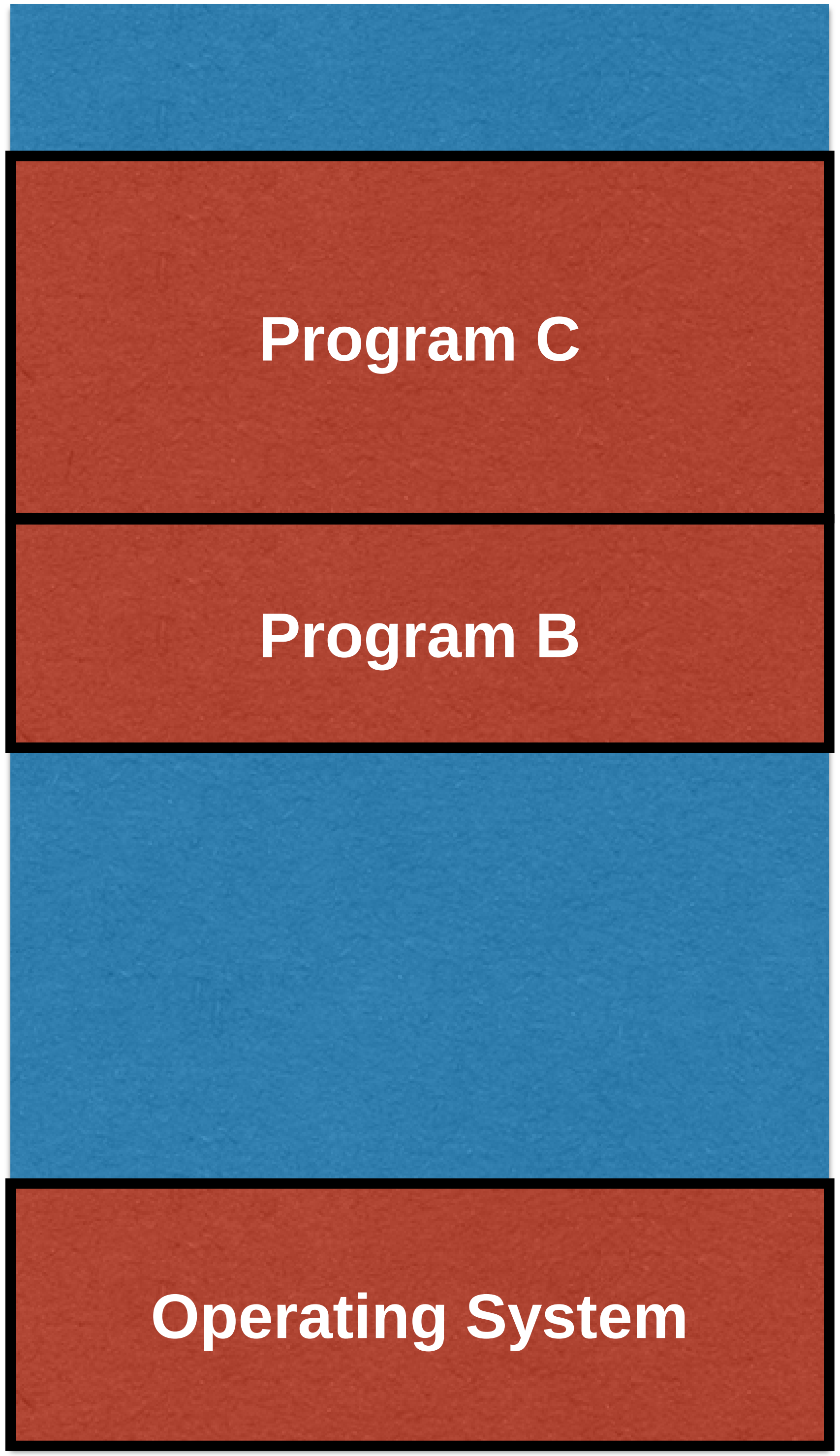
Swapping



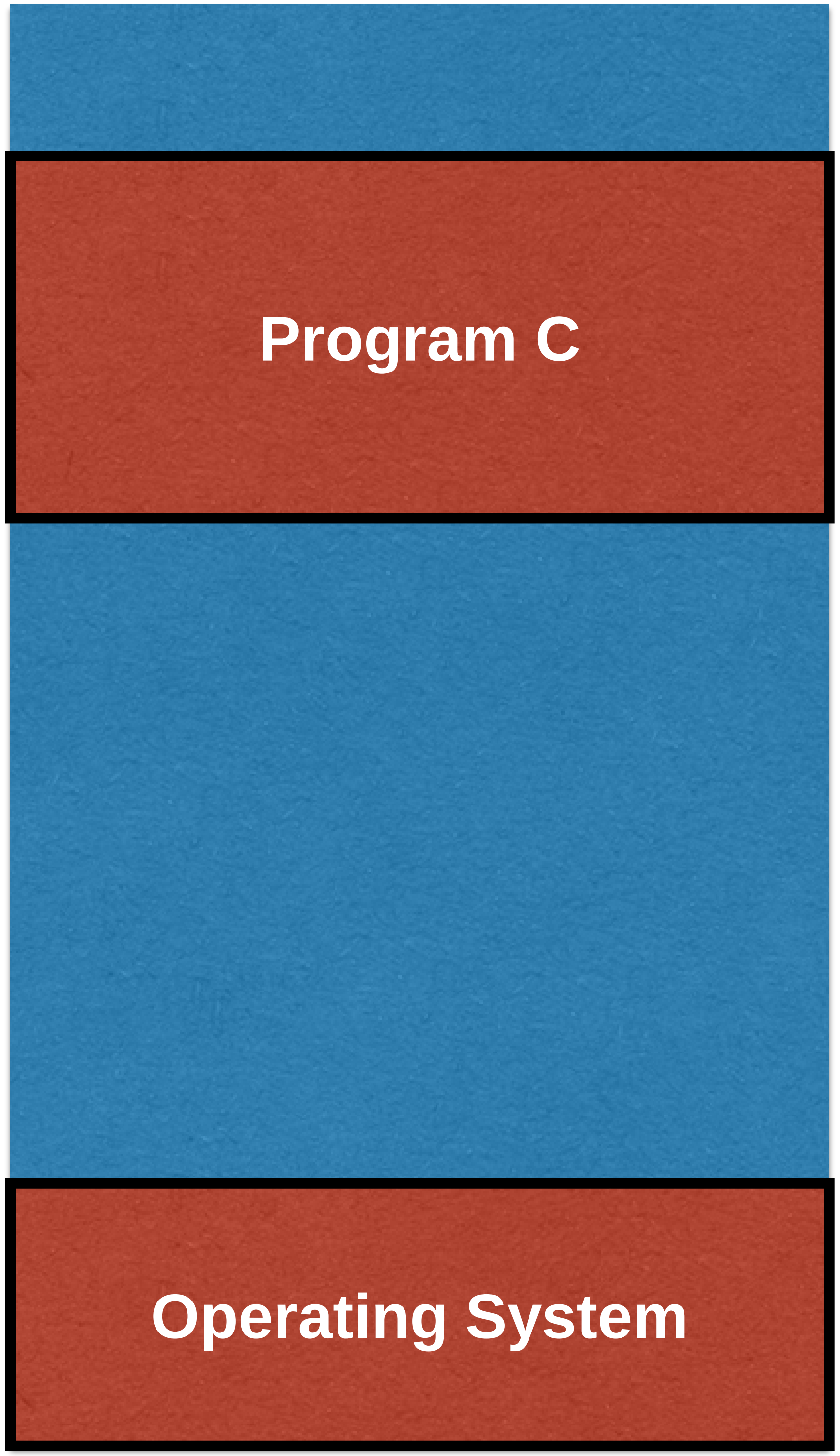
Swapping



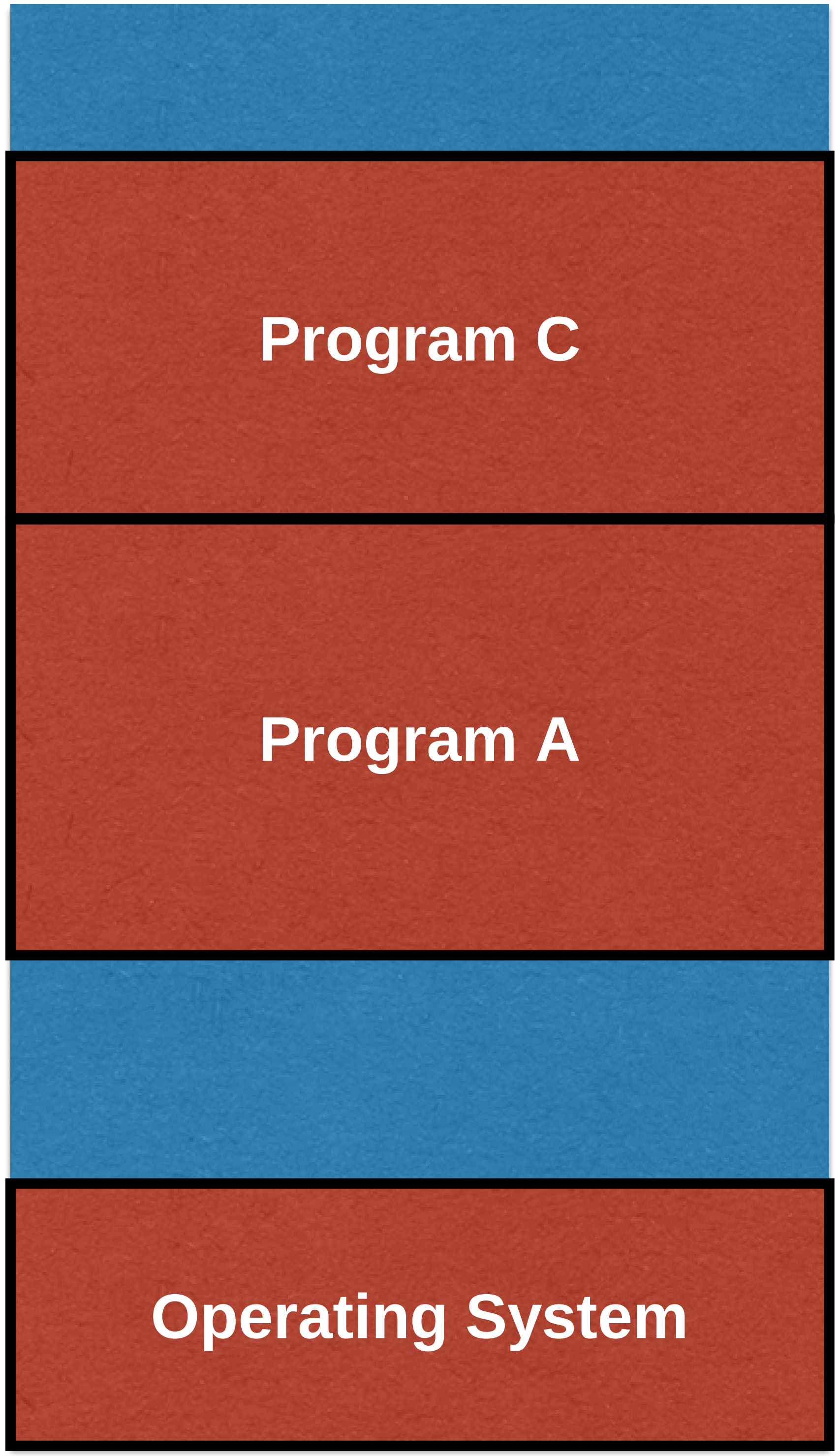
Swapping



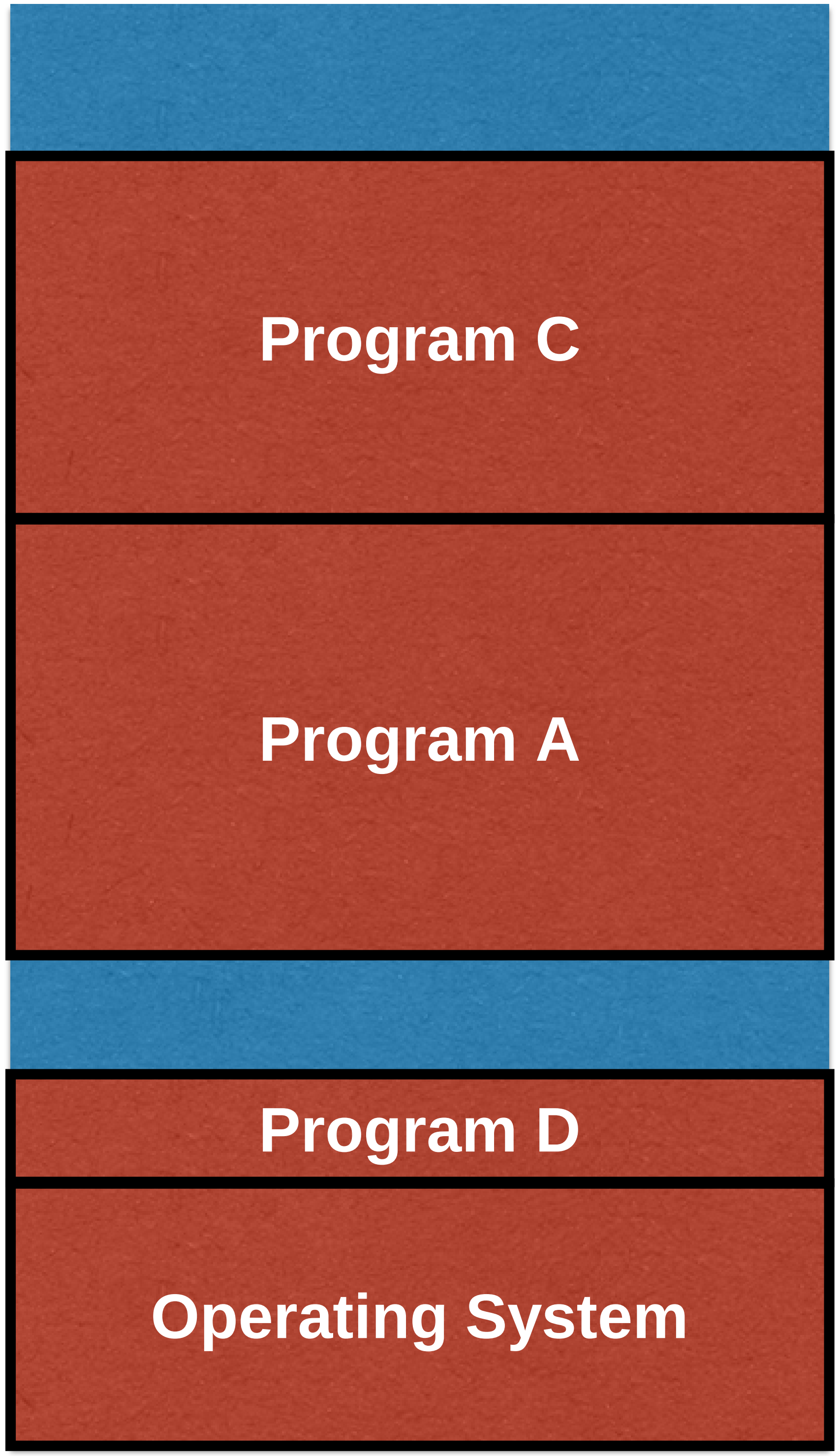
Swapping



Swapping



Swapping



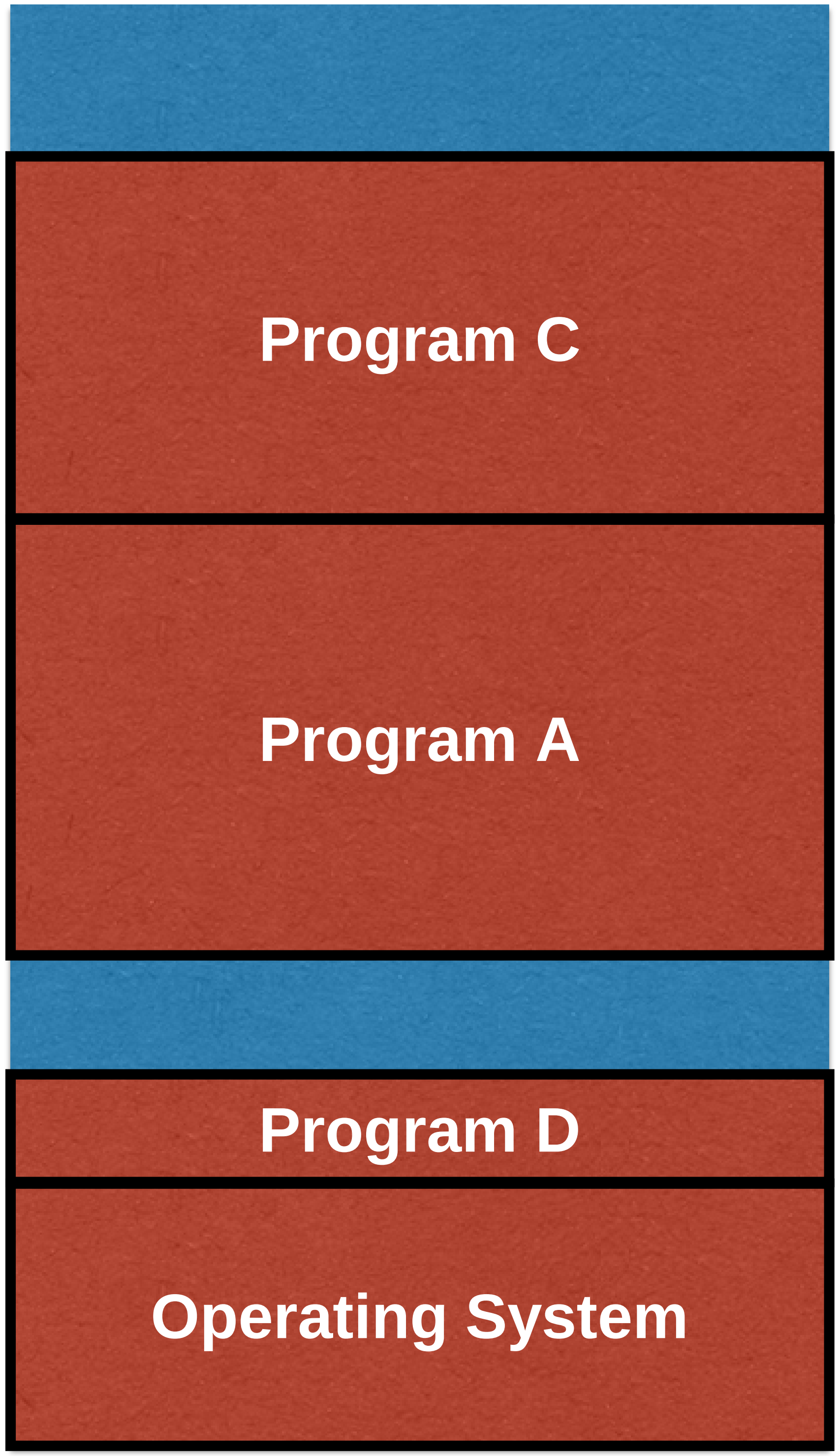
Today

- Address Space
- Swapping
- Managing Free Memory
- XV6 Address Spaces

Memory Compaction

- As a consequence of swapping things in and out of memory, we might *fragment* memory
- This could prevent us from loading a program even though we technically have enough memory for it
- If necessary, we can shuffle things around so that we have one contiguous free space instead of multiple small "holes"
- But: it may be slow! E.g. if it takes us 100 ns to read and then write 8 bytes of memory, ~100 seconds to move 8GB

Compaction



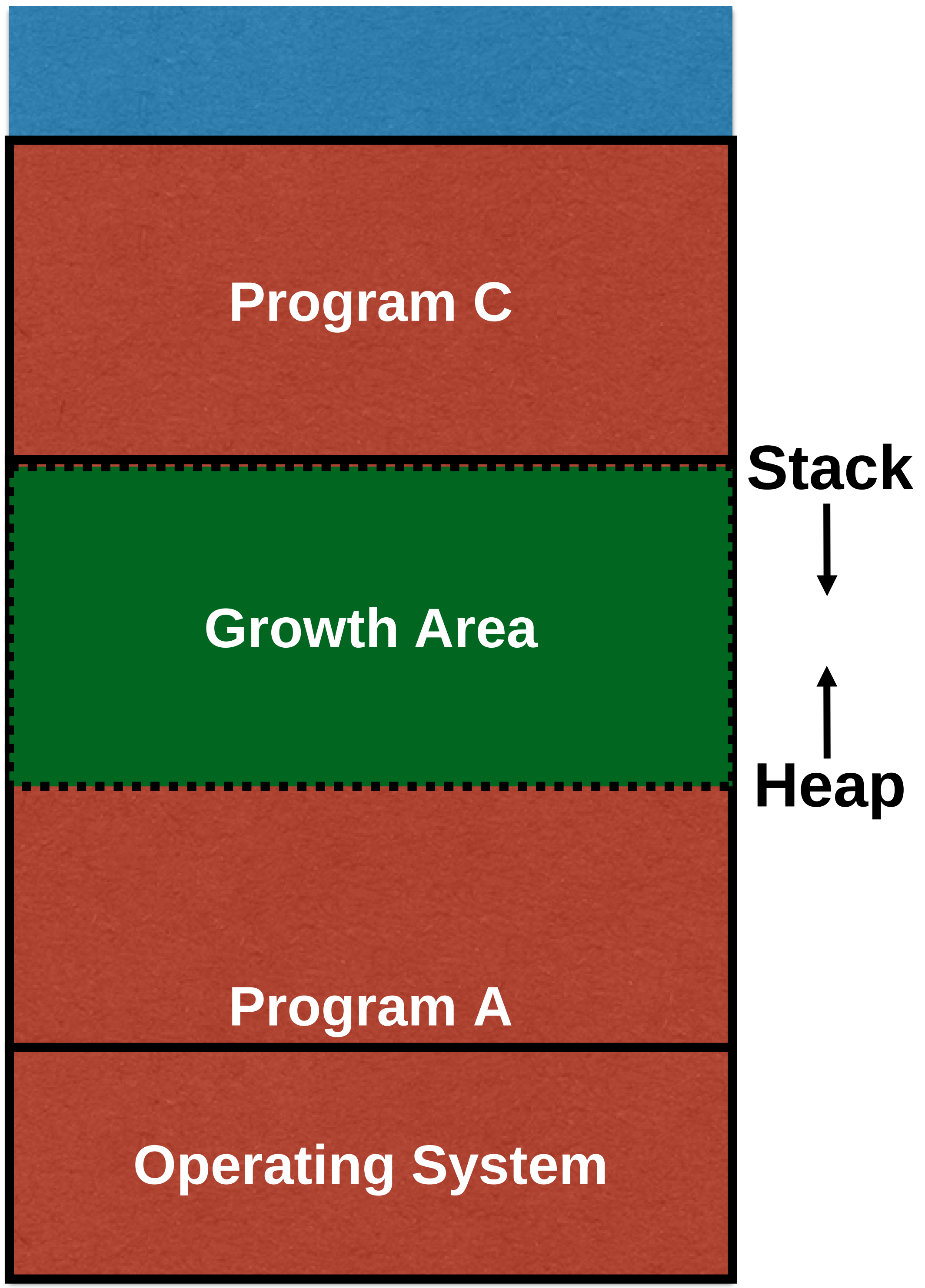
Growing Process Memory

- In general a process will not start off with all the memory it will ever need
    - Function calls will cause it to use more of the stack
    - Dynamically allocated data structures will need space
- So in this case we will need to grow the memory space allocated to the process

Growing Process Memory

- If we allocate processes right next to each other, then we would have to move or swap them the first time the process grows
- Instead, it makes more sense to start each process with room to grow

Growing Process Memory



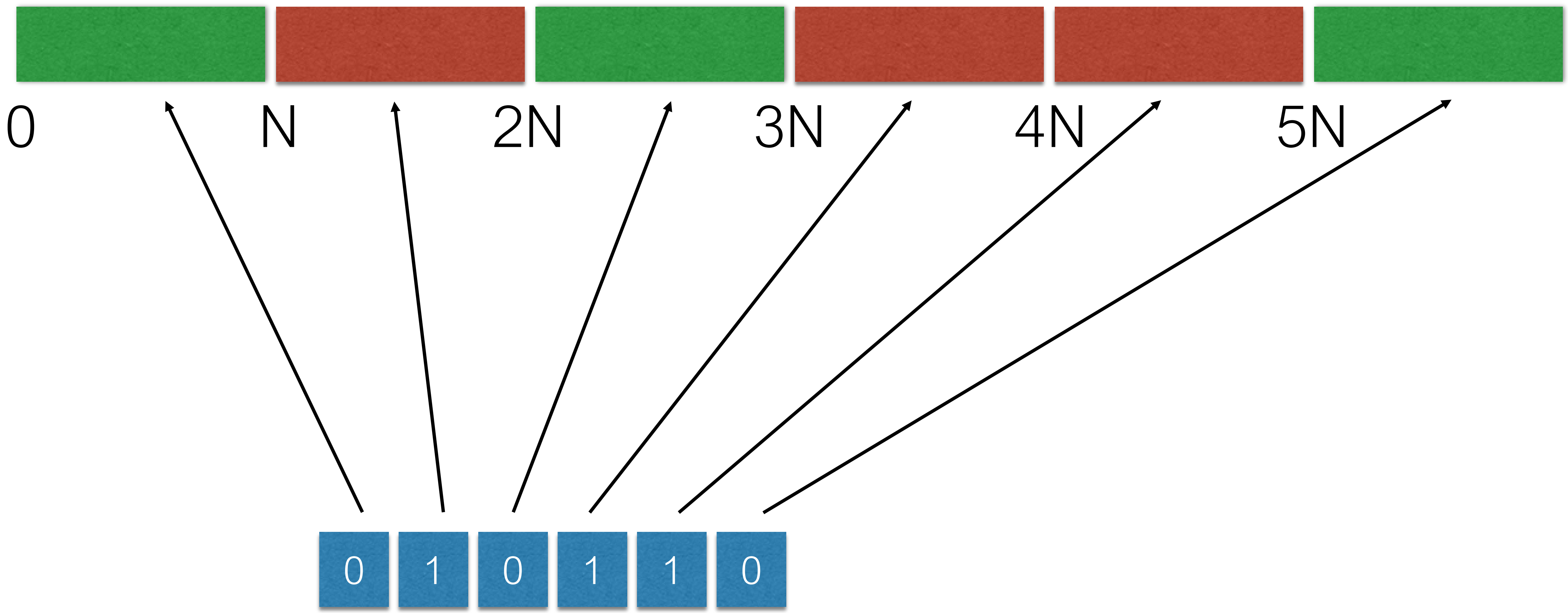
Keeping Track of Memory

- To decide where to put programs, we need to know what memory is used/free
- This is a job for the OS – maintain a data structure that it can use to know what's available
- Two main structures used for this are *bitmaps* and *lists*

Memory Bitmap

- Basic idea – allocate memory in chunks of size N (the *allocation unit*)
- Store a sequence of bits where bit *i* says whether the *ith* chunk is free
- The allocation unit size is yet another balancing act:
  - Large unit sizes mean fewer bits are needed to describe memory, but may waste memory if process is not exact multiple of N

Bitmaps



Suppose N = 8 bytes

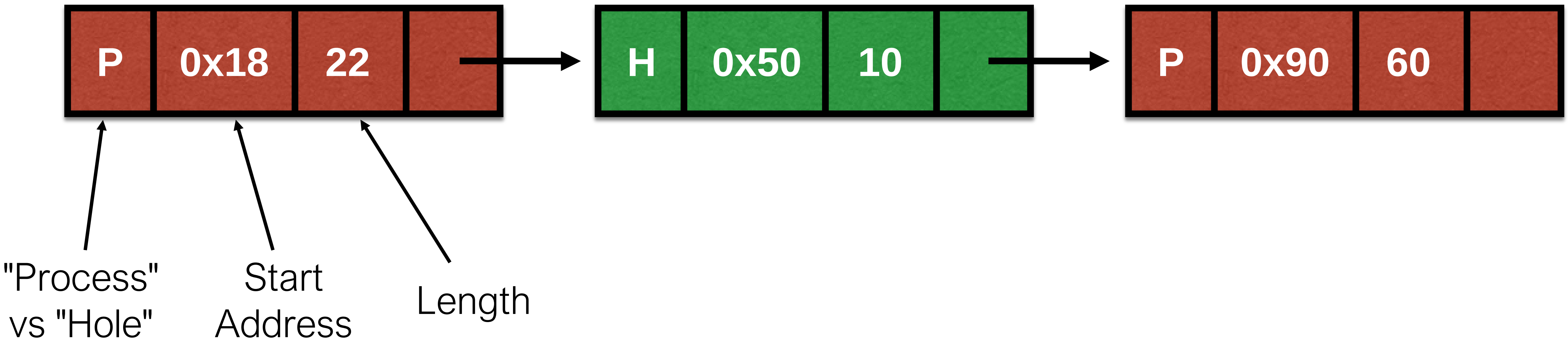
Then tracking 48 bytes of memory takes only 6 bits

Allocating/Freeing Memory

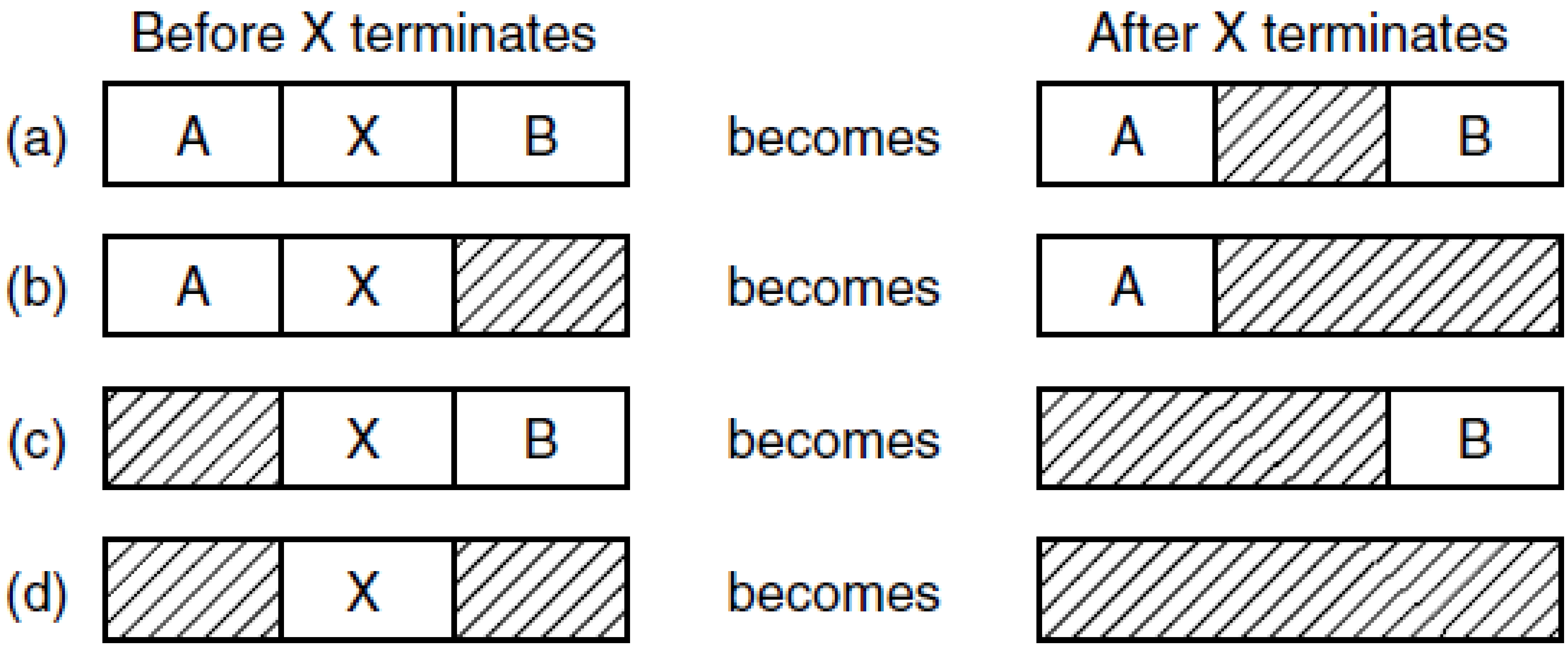
- To mark space as free, just set the right bits to 0
- To find space for a new process *K* units long, we need to search for a consecutive string of *K* zeroes
- This could be very slow, since most CPUs deal in units of multiple bytes, not bits, and the string of 0s could straddle a byte/word boundary

List-Based Memory Tracking

- Keep a linked list describing free and allocated regions



Freeing Memory



Finding Free Memory

- Many **strategies** to find the right place to allocate a process that needs space:
  - First fit – just traverse the list and pick the first free range large enough
  - Best fit – traverse the list and pick the *smallest* big-enough free range
    - Slower and actually wastes more memory than first fit
  - Quick fit – keep separate lists for commonly needed sizes
    - Fast to allocate, but much slower to deallocate – hard to merge adjacent free ranges

Optimizations

- Keep a separate *freelist* of just the unallocated regions
  - One nice trick is that we can actually store the list entries in the unallocated spaces themselves!
- Keep the lists sorted by address, so it's easier to merge free regions later
- Keep the lists sorted by size, so we don't have to search the entire list for the smallest

Today

- Address Space
- Swapping
- Managing Free Memory
- XV6 Address Spaces

Memory Management in xv6

- As in many things, the xv6 memory manager is designed to be *simple* rather than *efficient*
- The basic structure is a simple singly-linked free list where each entry is the same size (4096 bytes, the default *page size*)
- Note: for now we are just talking about how xv6 manages ***physical*** **memory** – the list of what's free and what's not

```
struct run {
  struct run *next;
};
```

Initialization

- When xv6 starts, it goes through and adds all available memory to the free list

```
// #define PHYSTOP 0xE000000 // Top physical memory


int
main(void)
{
...
  kinit2(P2V(4*1024*1024), P2V(PHYSTOP));
...
}
```

Freeing Each Page

```
void
kinit2(void *vstart, void *vend)
{
  freerange(vstart, vend);
}

void
freerange(void *vstart, void *vend)
{
  char *p;
  p = (char*)PGROUNDUP((uint)vstart);
  for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
    kfree(p);
}
```

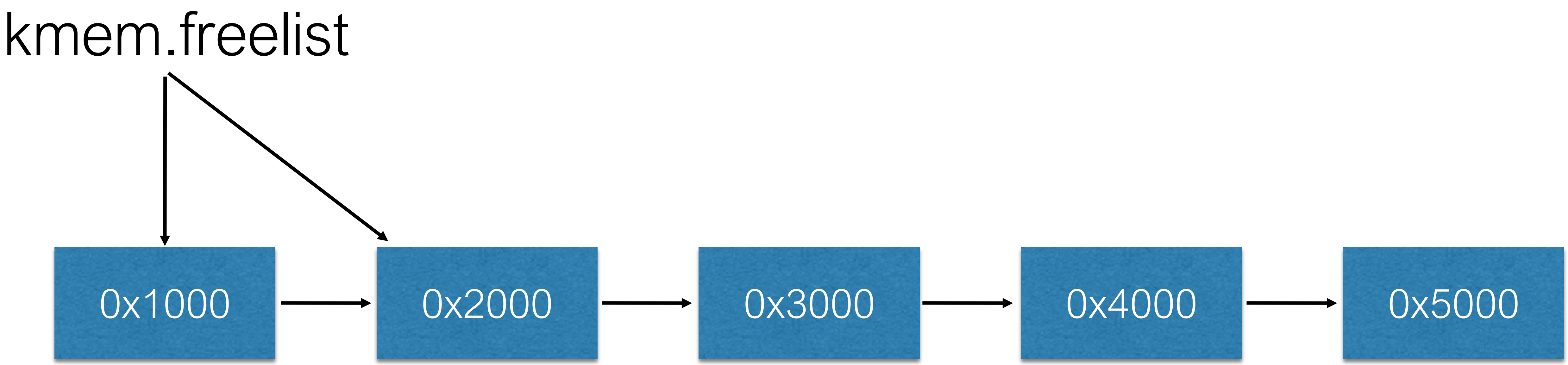
kfree

```
void
kfree(char *v)
{
  struct run *r;

  [….]

  // Fill with junk to catch dangling refs.
  memset(v, 1, PGSIZE);

  r = (struct run*)v;
  r->next = kmem.freelist;
  kmem.freelist = r;

  [….]
}
```

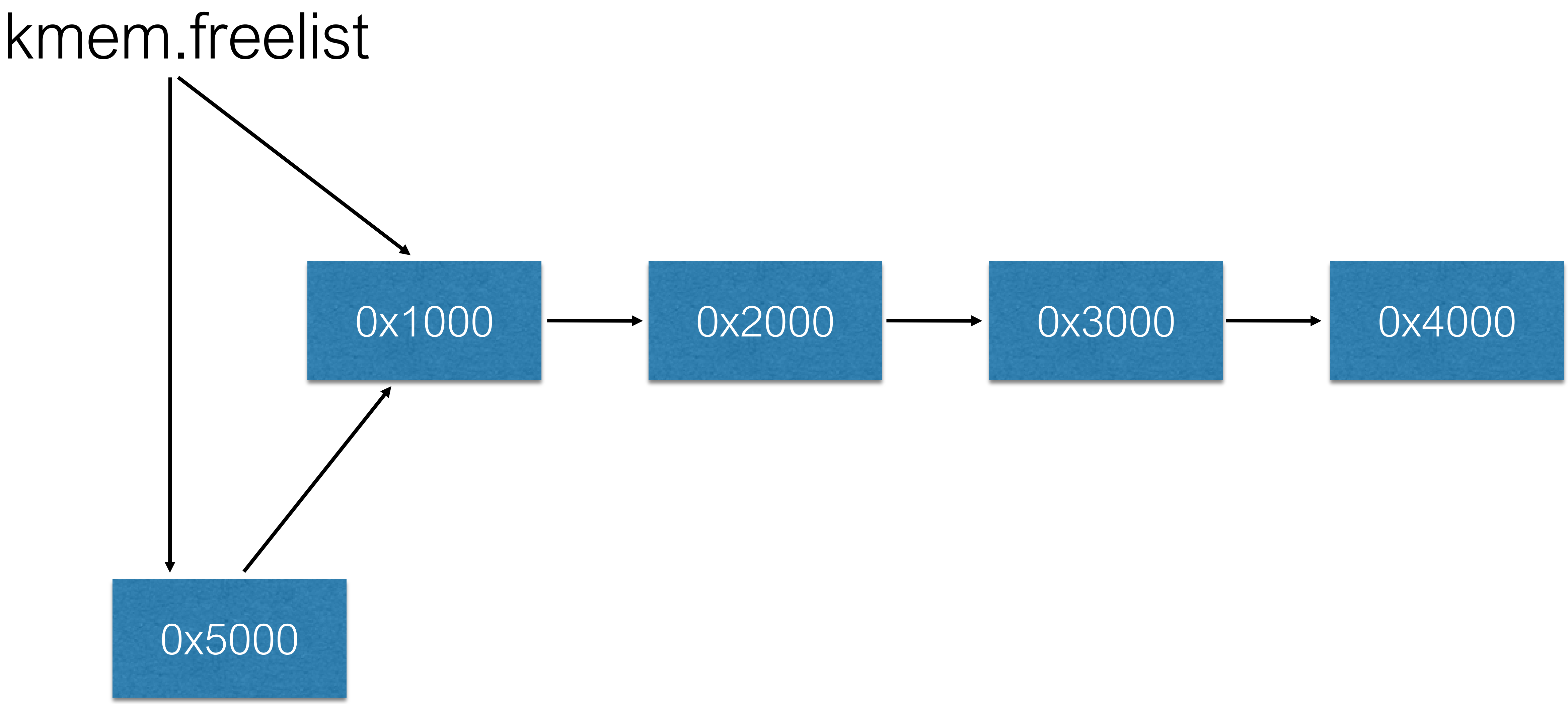
kalloc

```
char*
kalloc(void)
{
  struct run *r;

  r = kmem.freelist;
  if(r)
    kmem.freelist = r->next;
  return (char*)r;
}
```

xv6 Allocation



xv6 Freeing



But Wait...

- In our example of freeing memory, we ended up with a free list that did not contain pages in order
- If kalloc() is called twice, it will hand out 0x5000 and then 0x1000
- But user programs will probably want *contiguous* memory chunks larger than 4096 bytes
- The answer is that user programs never see the addresses used by the memory manager
  - *Virtual memory* is used to map contiguous *virtual addresses* to a discontiguous set of physical pages