EC 440 – Introduction to Operating Systems

Manuel Egele

Department of Electrical &
Computer Engineering

Boston University

Slides from Christopher Kruegel, UCSB

Memory Management

Memory Management

Ideally memory would be

- Large
- Fast
- Non volatile

In practice, a memory hierarchy is used

- Small amount of fast, expensive memory (cache)
- Some medium-speed, medium price (main memory, RAM)
- Gigabytes of slow, cheap disk storage

Memory

- RAM is one of the main resources managed by an operating system
- RAM is *volatile* storage (does not persist across reboots)
  - Though perhaps new technology like persistent memory will change that …
- The portion of the OS that allocates, frees, and tracks the usage of RAM is the *memory manager*

Memory Management

- 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

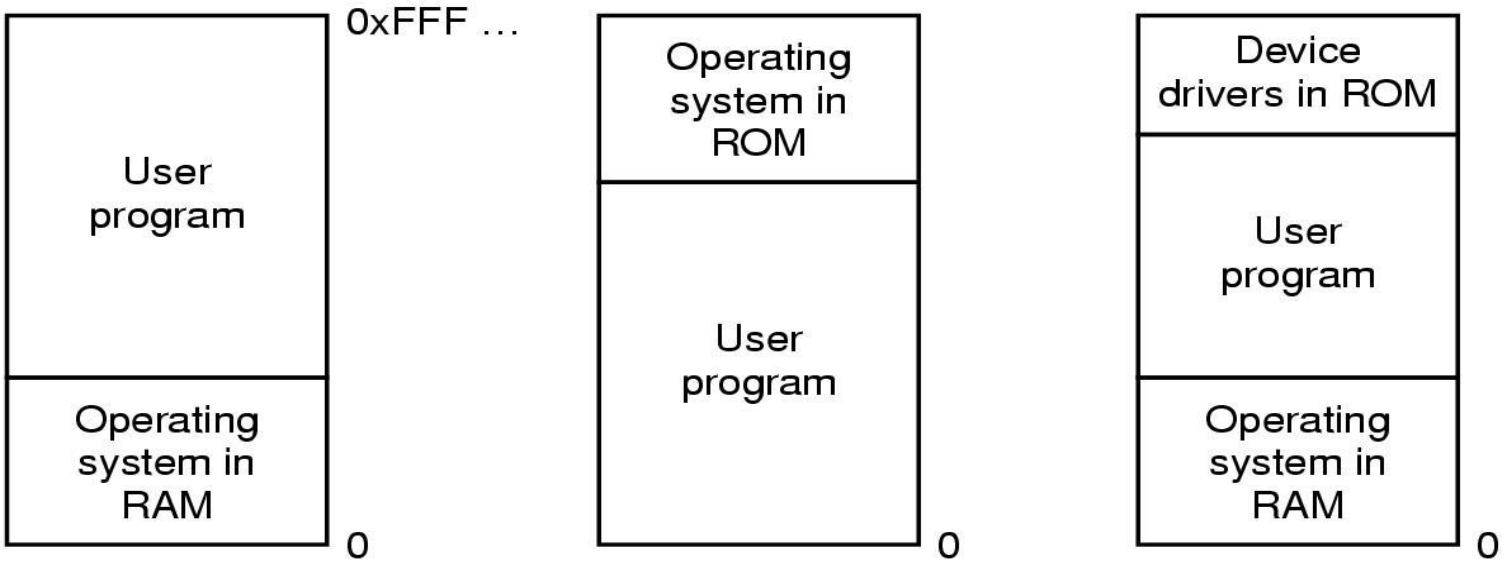
Physical Memory Model Organization

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?

Simplest Memory Management

Sounds easy to do, but what are the downsides?



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. Why?
  - Threads always share the same address space anyways

Memory Management Approaches

Mono-programming vs. multi-programming

Fixed working set vs. swapping

Complete image vs. partial image (virtual memory)

Modern systems

- multi-programmed, with swapping and virtual memory

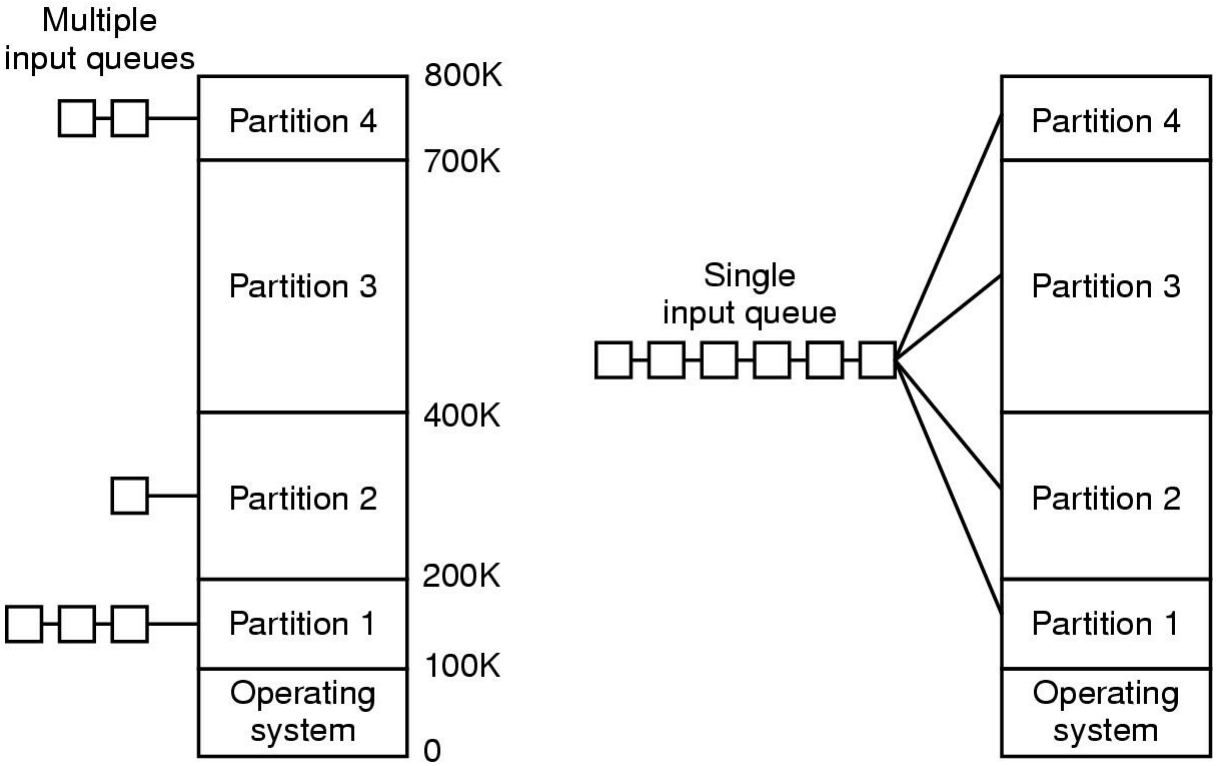
Dealing With Multiprogramming

- Available memory is divided into a number of fixed partitions
- Partitions can be of different sizes
- When process needs to be executed, put in one of available partitions

- Problem: if a small program gets a big partition memory gets wasted

- Solution
  - Use different queues for different partitions
  - Use one queue and choose the largest job that fits an available partition and use some aging mechanisms to ensure that small jobs are not left to starve (e.g., an “ignored” counter)

Multiprogramming with Fixed Partitions



Relocation and Protection

The programmer cannot be sure where the program will be loaded in memory

1. Address locations of variables, code routines cannot be absolute
2. Must keep a program out of other processes' partitions

Relocation can be done at load time

- Maintain a list of all places in the program where absolute addresses were used (relocations)
- At load time, simply add an offset to all absolute memory references
- Introduced by the IBM 360 in 1965. Why would this still be relevant?
  - Shared libraries used by a program may have to be statically relocated before they are loaded

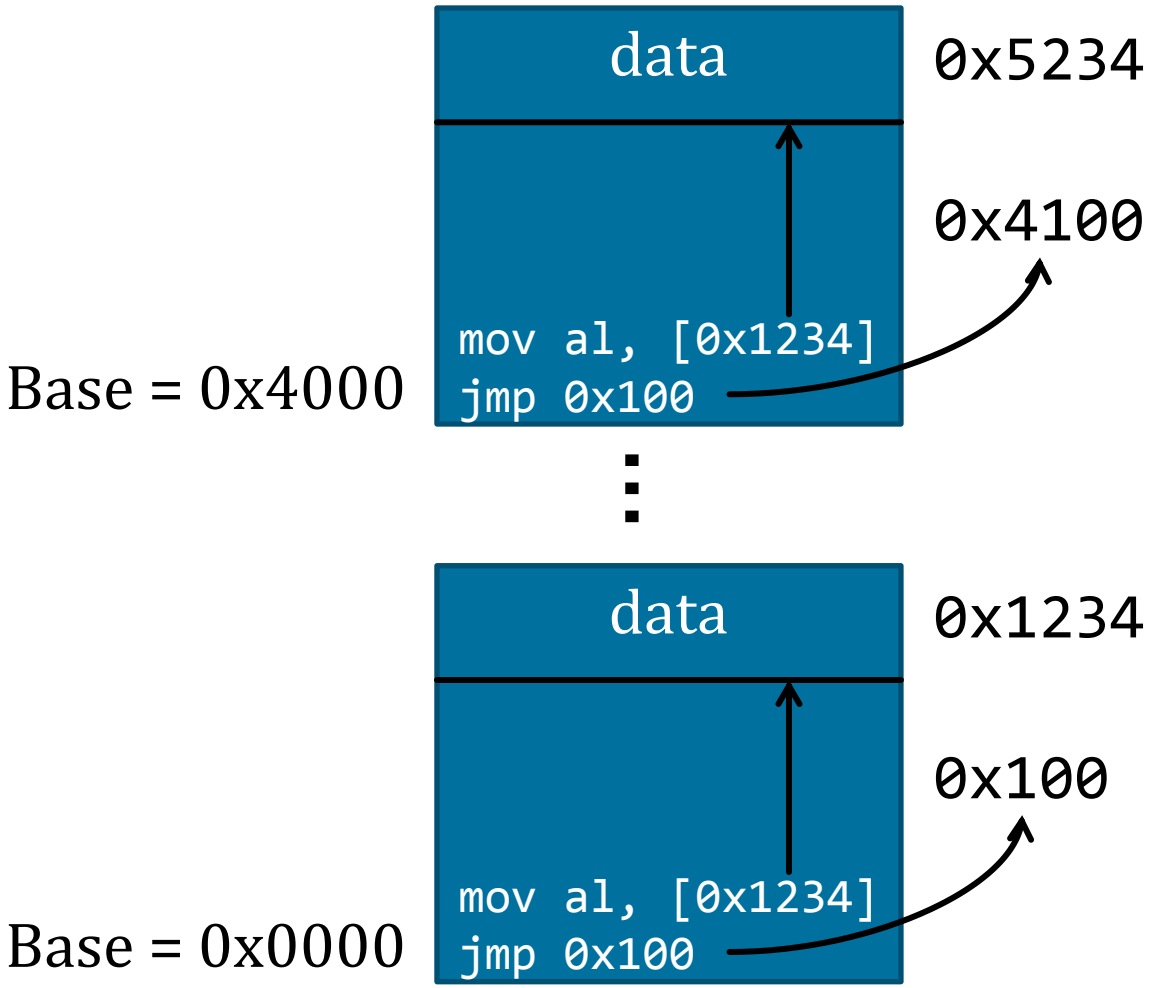
Relocation and Protection – Segmentation

How do we enforce that processes do not touch the memory in other partitions?

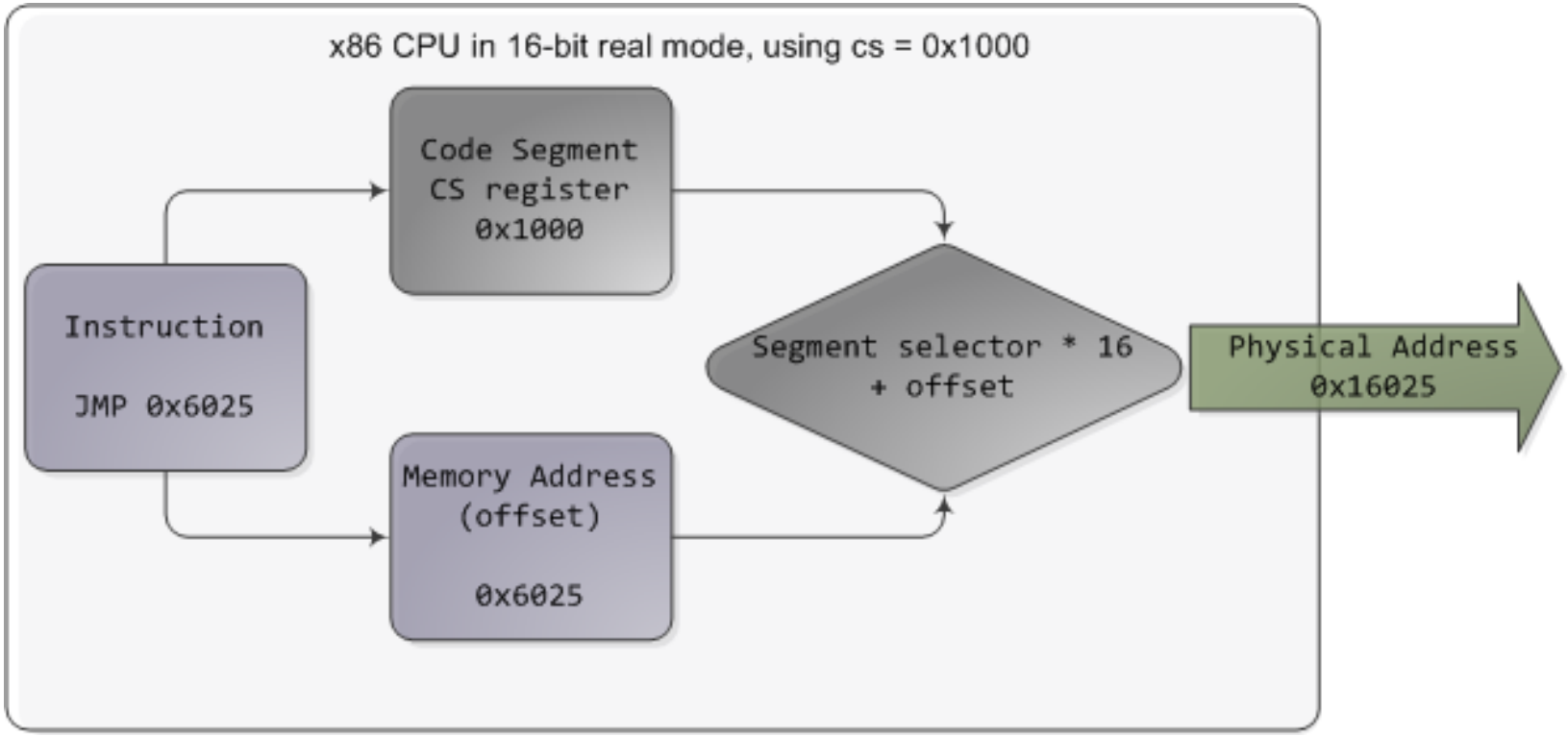
Hardware segmentation

- 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

Multiple Programs with Segmentation



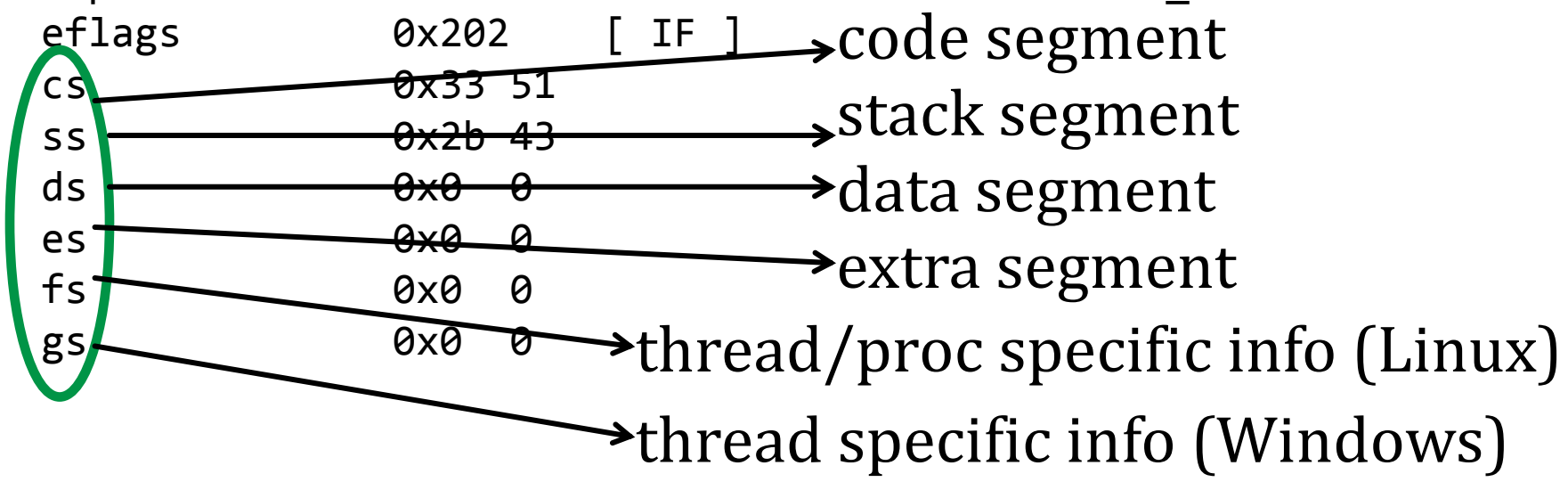
Real Mode Segmentation



source: http://duartes.org/gustavo/blog/post/memory-translation-and-segmentation

x86 Segment Registers

```
(gdb) info registers
rax            0x0  0
…
rdi            0x0  0
rbp            0x0  0x0
rsp            0x7ffe5d982c70   0x7ffe5d982c70
r8             0x0  0
…
r15            0x0  0
rip            0x7fb8eff71c20   0x7fb8eff71c20 <_start>
eflags         0x202    [ IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
```



Segment Register: %gs

Process information
(e.g., the key to mangle PC/SP pointers in a jmp_buf)

```
static unsigned long int ptr_mangle(unsigned long int p) {
        unsigned long int ret;
        asm(" movq %1, %%rax;\n"
            " xorq %%fs:0x30, %%rax;"
            " rolq $0x11, %%rax;"
            " movq %%rax, %0;"
        : "=r"(ret)
        : "r"(p)
        : "%rax"
        );
        return ret;
}
```

Swapping

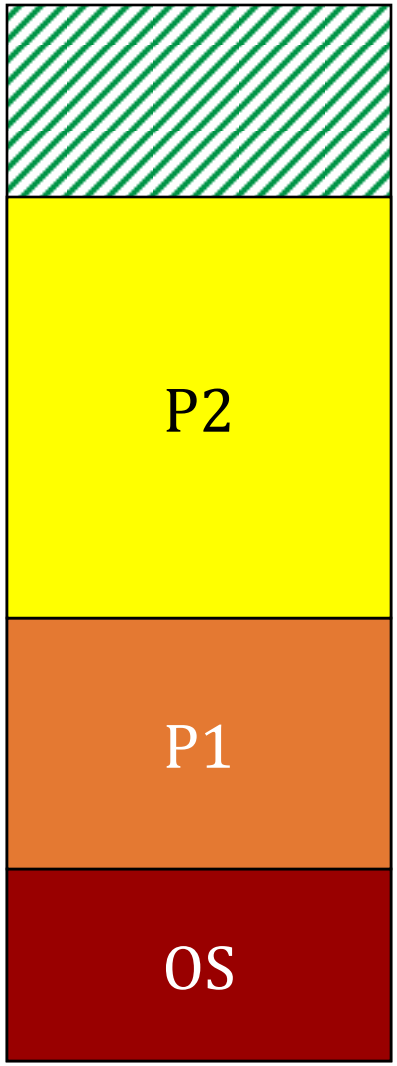
- Most of the time there is not enough memory to hold all the active processes at the same time
- Swapping is the process of saving a task to disk
- In swapping processes are loaded to and discarded from memory following the needs of execution
- Address must be relocated each time either by hardware or by software
- After a while memory can get fragmented and may need compaction, which is computationally expensive
- A process may also try to get bigger and bigger and bigger

An Example

Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P2 can be loaded

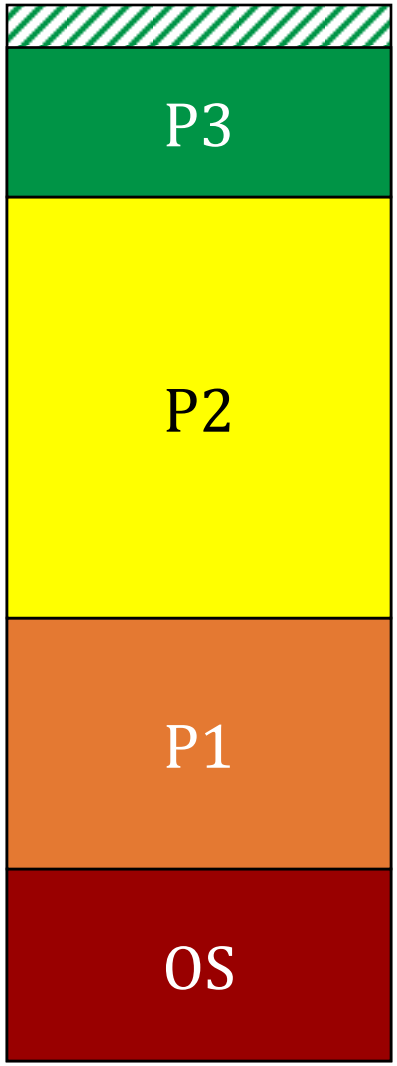
An Example



Process	Memory	Time
P1	600k	10s
P2	1,000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P3 can be loaded

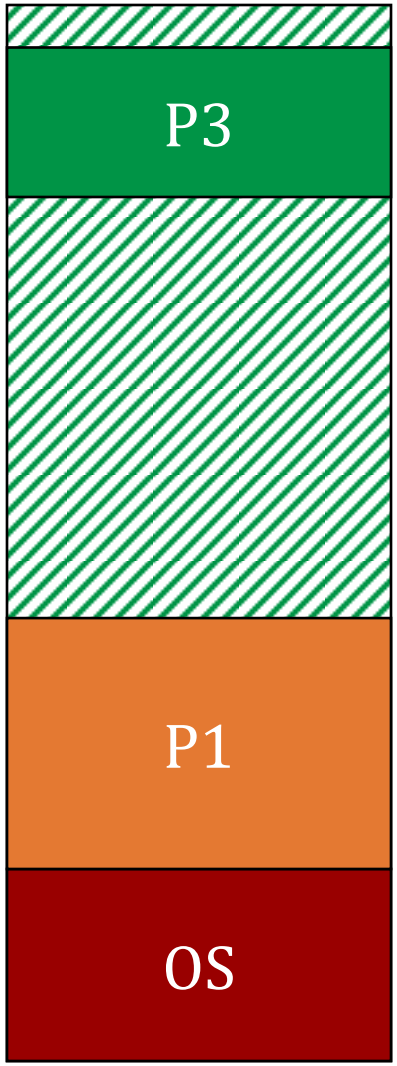
An Example



Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P4 & P5 must wait!

An Example

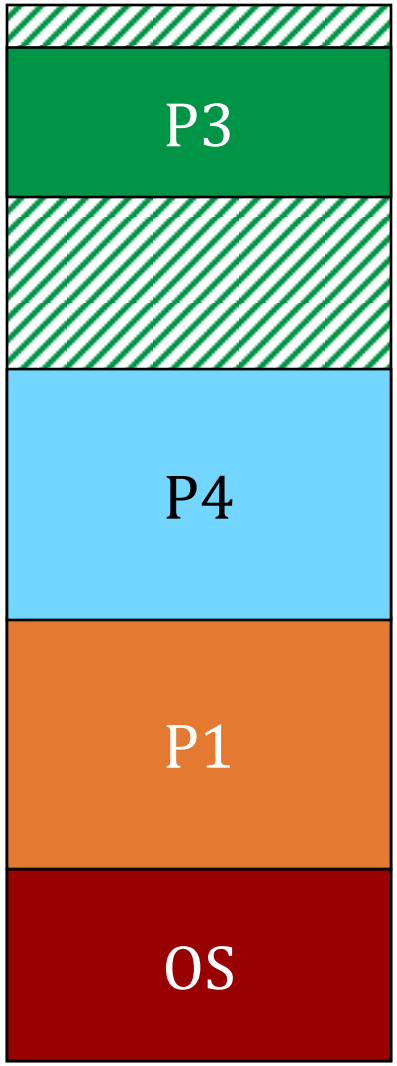


Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

terminates

P4 can be loaded

An Example



Process	Memory	Time
P1	600k	10s
P2	1000k	5s
P3	300k	10s
P4	700k	8s
P5	500k	15s

P5 cannot be loaded!
300k + 260k free,
500k needed

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 100ns to read and then write 8 bytes of memory, ~107 seconds to move 8GB

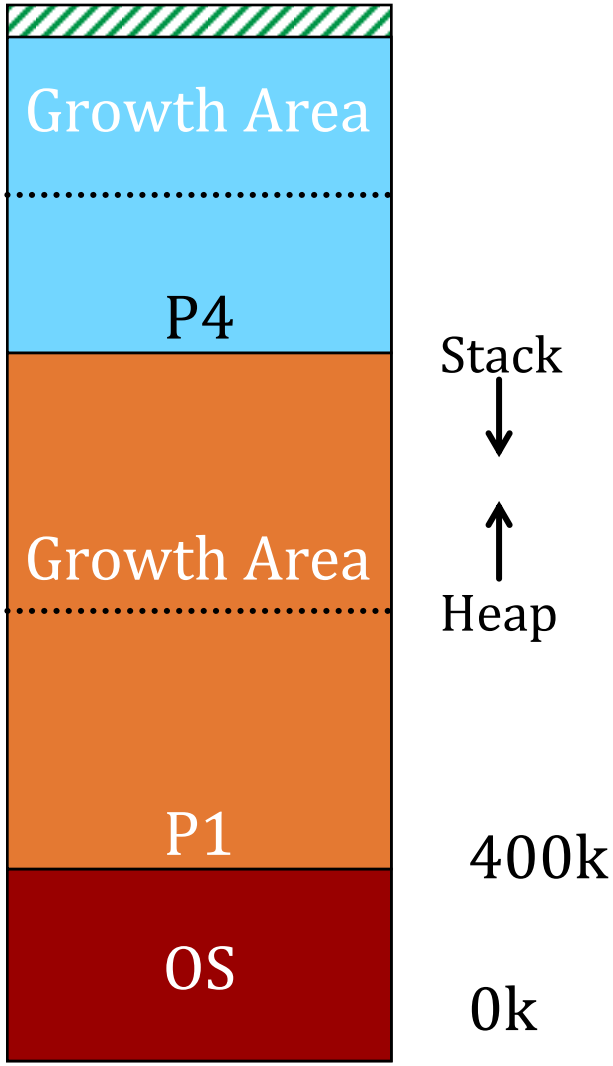
Growing Process Memory

- 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 too
- So in this case we will need to grow the memory space allocated to a 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

Process Growth Area



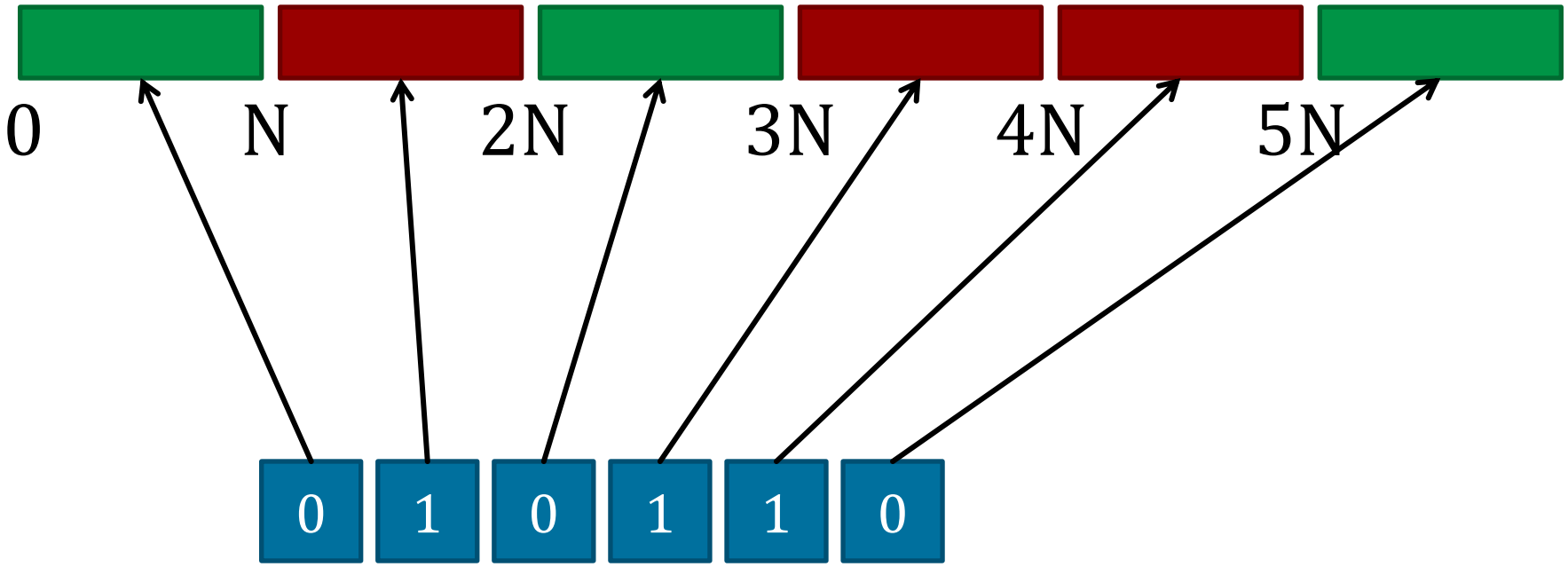
Keeping Track of Memory

- 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 i^{th} 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

Memory Bitmap



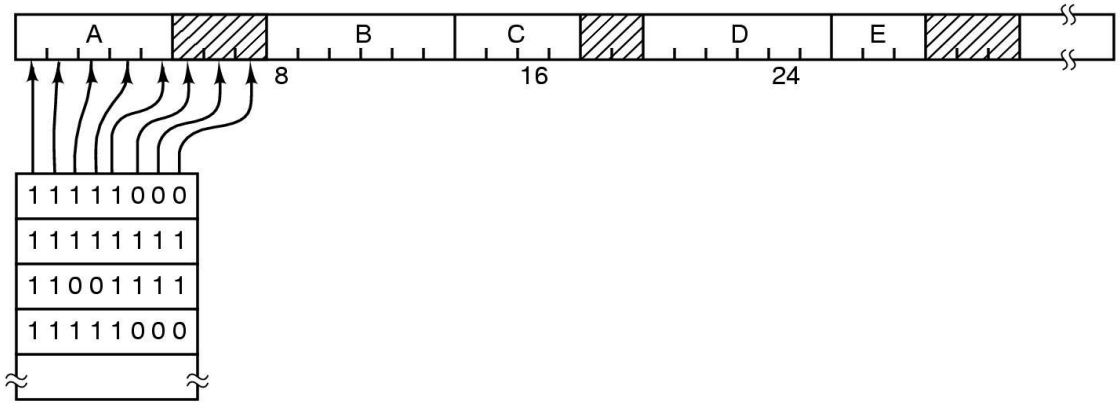
Suppose N = 8 bytes
Then tracking free/used for 48 bytes
of memory takes only 6 bits
How about N = 4 bytes?

Allocating/Freeing Memory

- To mark space as free, just set the corresponding bits to 0
- To find space for a new process K units long, we need to search for a consecutive string of K zeros
- 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

Memory Management with Bitmaps

- Divide memory in allocation units
- Keep track of which units have been used and which ones are free using a bitmap
- Tradeoff:
  - Big allocation units: +small bitmap -may waste memory
  - Small allocation units: -better “fit” +big bitmap

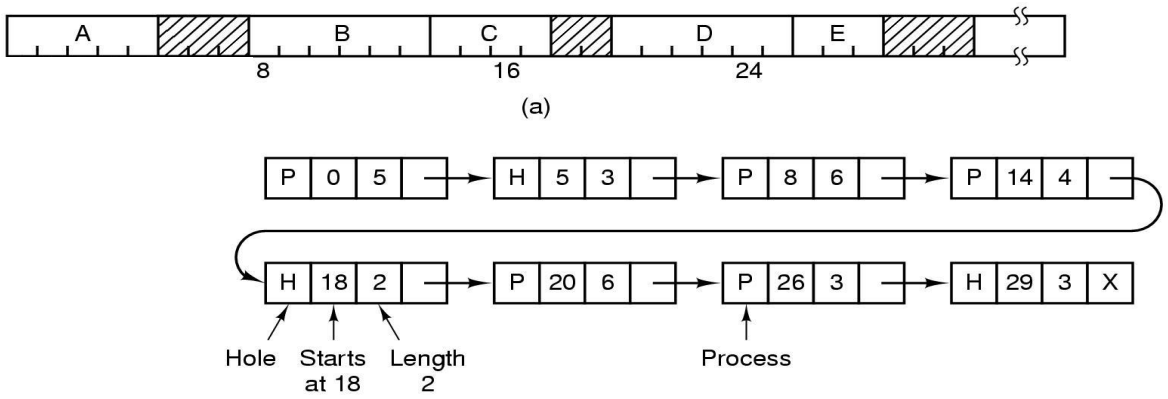


Memory Management with Linked Lists

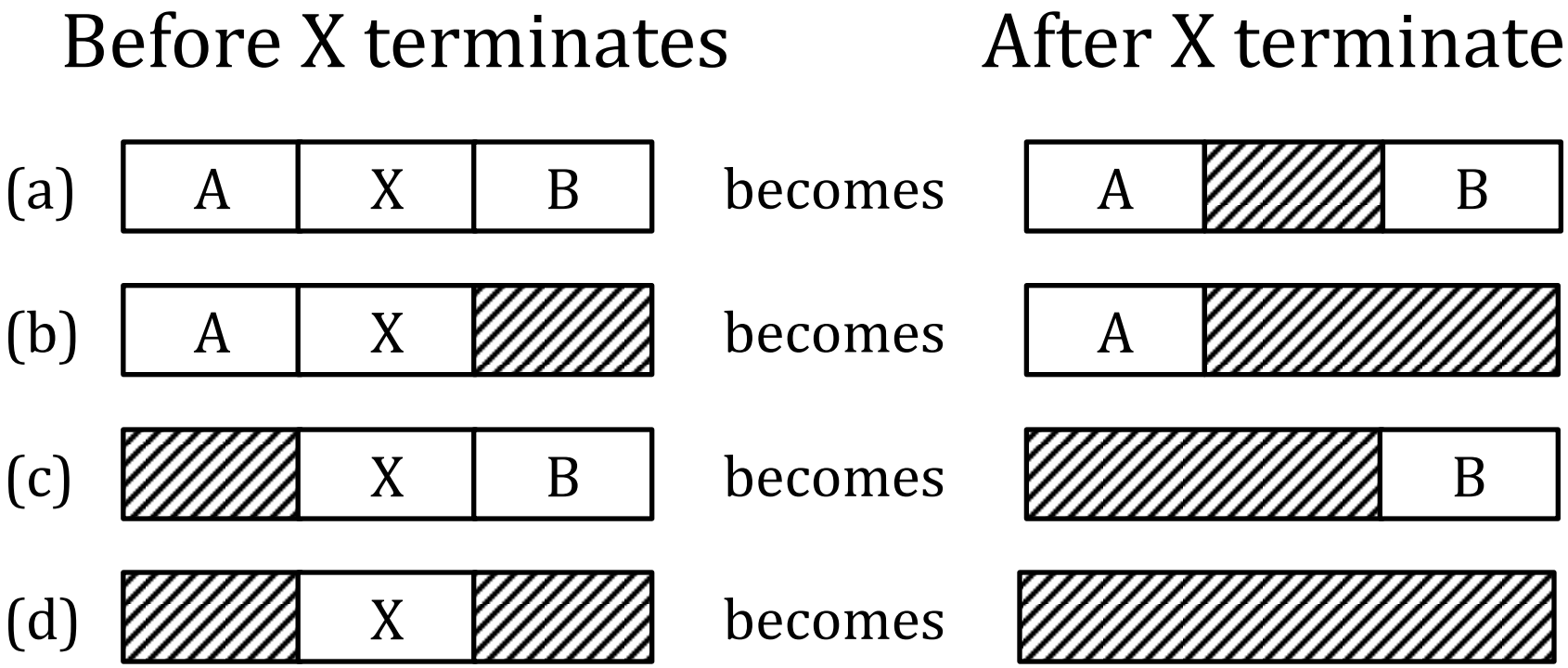
Maintain a linked list where each element

- May represent a process (P) or a piece of free memory ("hole", H)
- Contains start address of initial unit
- Contains length of memory block
- Maintains pointer to each element (single- or double-linked)

List is usually keep ordered



Freeing Memory



Coalescing adjacent free chunks into bigger free chunks.

Finding Free Memory

Many strategies to find the right place to allocate a process that needs space:

First fit

- Search the list until a suitable hole is found
- Split the hole in a P and an H

Best fit

- Search the entire list and use the smallest hole that fits the program
- Slow (requires complete scan through the list)

Quick fit

- Separate lists, hashed by size or size ranges
- Speeds up search
- Makes compaction difficult

List Management – 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

Modern operating systems use paging and virtual memory, but these techniques remain very relevant for heap management (`malloc`)

Virtual Memory

- What if a program is too big to be loaded in memory?
- What if a higher degree of multiprogramming is desirable?
- Physical memory is split into *page frames*
- Virtual memory is split into *pages*
- OS (with help from the hardware) manages the mapping between pages and page frames