CSC369 Lecture 5

Larry Zhang, October 19,2015

Describe your A1 experience using the feedback form

CSC369 Fall 2015 -- Weekly Feedback

* Required

Which week are we talking about here? *

Describe your A1 experience.

When did you start? How hard did you find it? How did you get help? What kind of help is useful and what is not? Did the group work well together? Any suggestions?

Announcements

- Assignment 2 out later this week, due November 11
- Midterm next week in lecture: 9:10AM - 10:00AM
  - closed book, bring your T-Card
- Pre-midterm office hour
  - Monday 3-4pm
  - Wednesday 11am-12pm
  - Friday 4-6pm

Midterm

- What does it cover?
  - Everything before (not including) virtual memory
  - Processes, system calls, threads
  - Concurrency, synchronization
  - Scheduling
- What types of questions?
  - Short answers, multiple choices, True/False
  - Analysis, reading code
  - Mechanically apply an algorithm
  - Design mechanism to solve problem.
- Review note and sample past tests posted to web page.
  - past tests may cover different content from us

More announcement

- This week's tutorial needs computer, bring your laptop or use the lab computer.
  - will use gcc, Valgrind, bash, Python

- Final exam: December 7, 9am~12pm, Gym A/B

Recap: Where we are

Virtualization

Virtualizing CPU

- Processes
- Threads
- **Scheduling ...**

Virtualizing Memory

- Address space
- Segmentation
- Paging ...

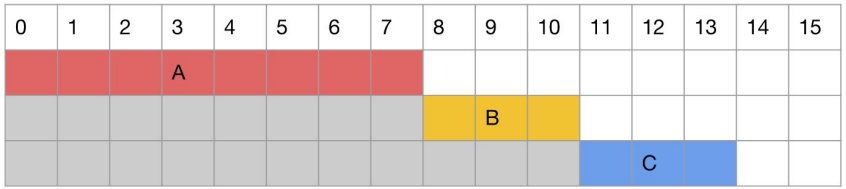
Concurrency

- Threads
- Locks
- Conditional variables
- Semaphores ...

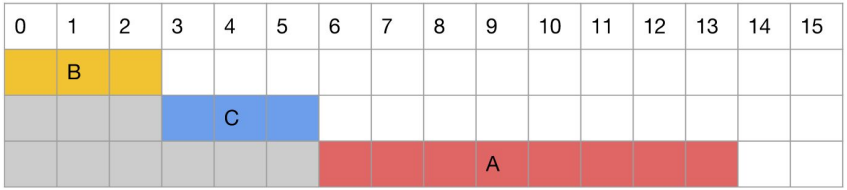
Persistence

- File systems
- Journaling, logging
- I/O devices
- File integrity ...

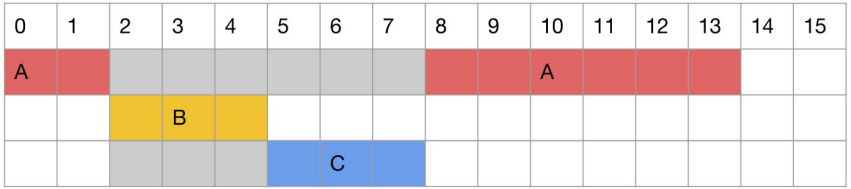
Recap: Scheduling algorithms



FCFS

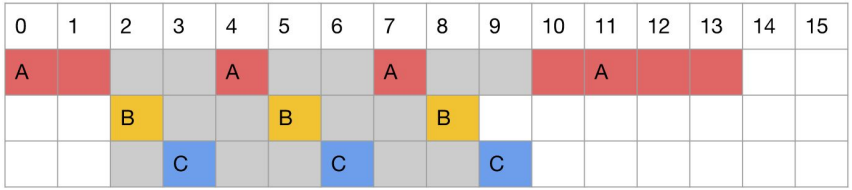


SJF



STCF / PSJF

Optimal for **turnaround time**



RR

Improve **response time** by sacrificing turnaround time

Recap: Assumptions

1. Each job (process/thread) runs the **same amount of time**
2. All jobs **arrive at the same time**
3. Once started, each job **runs to completion** (no interruption in the middle)
4. All jobs only use the **CPU** (no I/O)
5. The runtime of each job is **known**

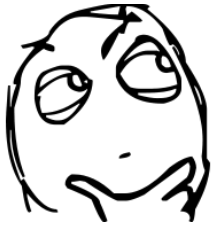
assumption relaxed

assumption relaxed

assumption relaxed

assumption relaxed

The worst assumption of all -- it typically not true in real life, and without it SJF/STCF become nonsense.



How to we schedule jobs **without knowing** their lengths
so that we minimize **turnaround time**
and also minimize **response time** for interactive jobs ?

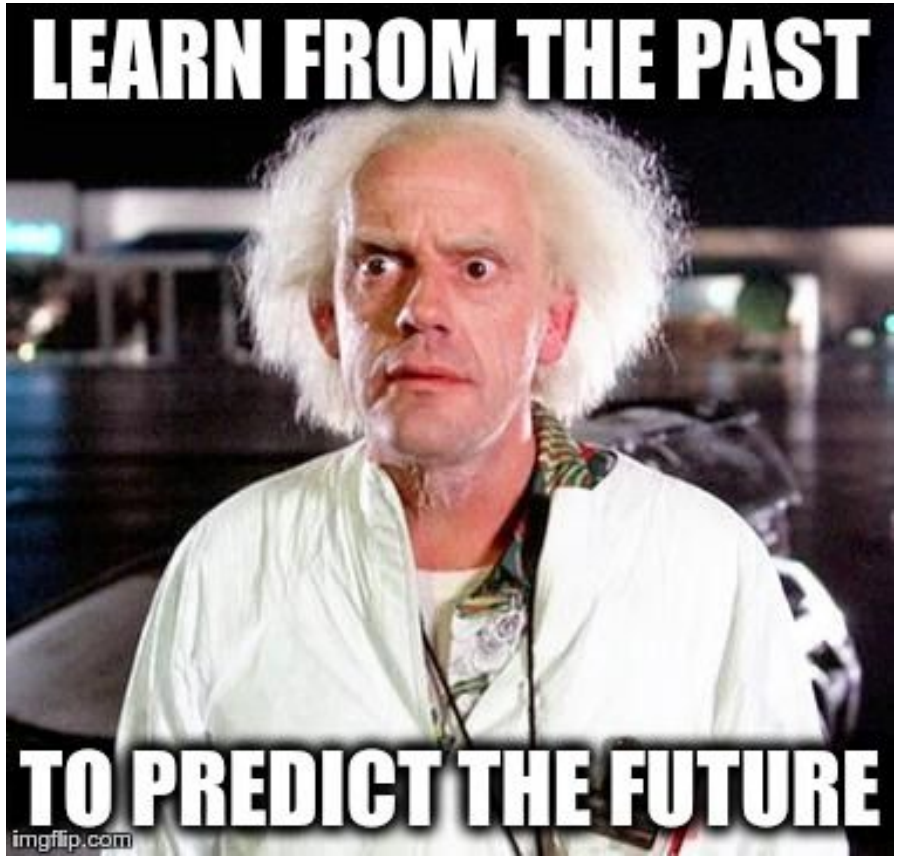
It’s called

Multi-**L**evel **F**eedback **Q**ueue

Invented by Fernando J. Corbató in 1962

won ACM Turing Award

The Basic Idea Behind MLFQ

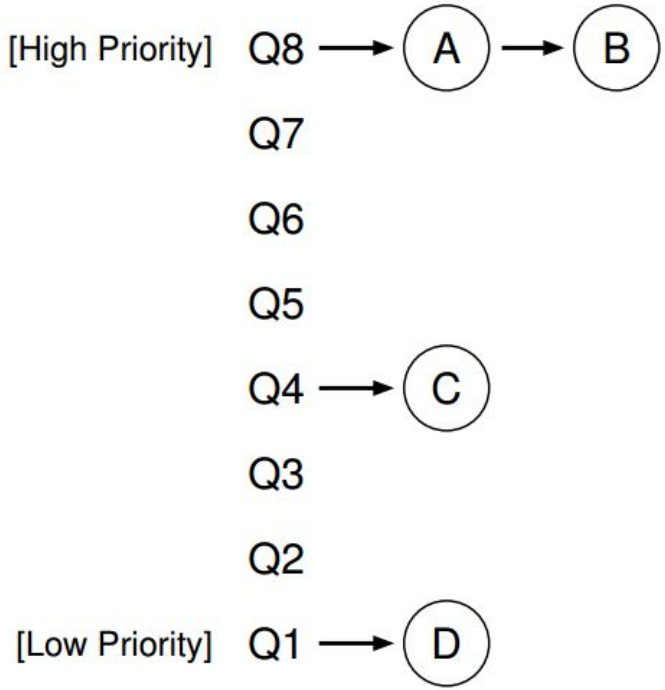


MLFQ: Setup

Assume two categories of jobs:

- ➔ **Long-running** CPU-bound jobs
- ➔ **Interactive** I/O-bound jobs

- ➔ Consists a number of distinct **queues**, each assigned a different **priority level**.
- ➔ Each queue has **multiple** ready-to-run jobs with the **same** priority.
- ➔ At any given time, the scheduler choose to run the jobs in the **queue** with the **highest** priority.
- ➔ If **multiple** jobs are chosen, run them in **Round-Robin**

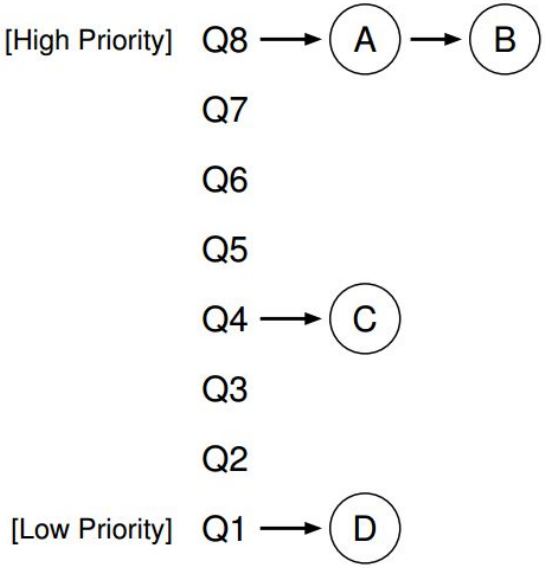


So we see where the MLQ come from, but not the F yet.

MLFQ: Rules (incomplete)

- **Rule 1:** if Priority(A) > Priority(B), A runs (B doesn't)
- **Rule 2:** if Priority(A) = Priority(B), run both in RR
- **Does it work well? Why not?**
- **Rule 3: ????**
- **???**

The trick: **change priorities** of jobs sensibly to achieve good performance (turnaround time and response time)

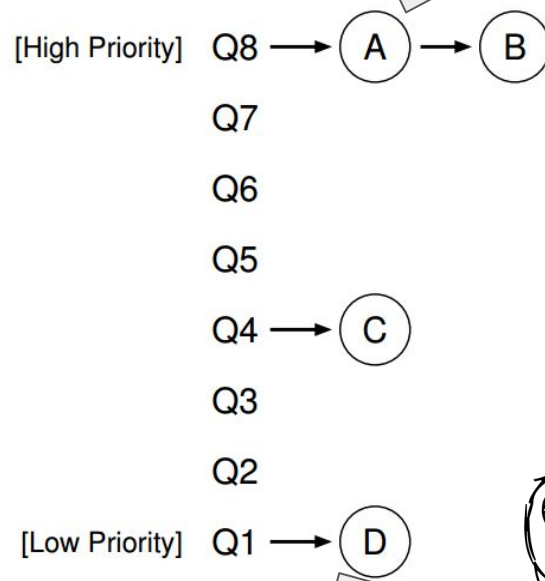


First Attempt: Change Priority

- **Rule 3:** When a job **enters** the system it is place at the **highest** priority.
- **Rule 4:**
  - If a job uses to an **entire time slice** (of some defined length), its priority is **reduced** (move down one queue)
  - If a job **gives up** the CPU **before** the time slice is up, it **stays** at the **same** priority level.

This is the F in MLFQ, i.e., feedback

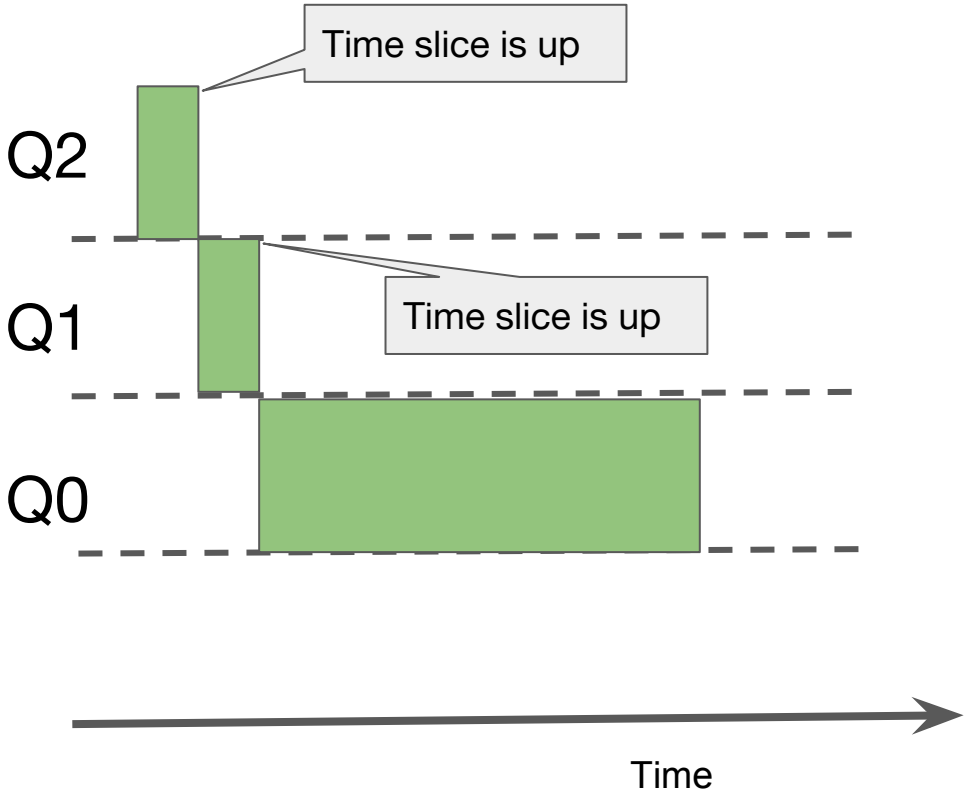
If A and B are long-running while C and D are interactive, response time is going to be crap.



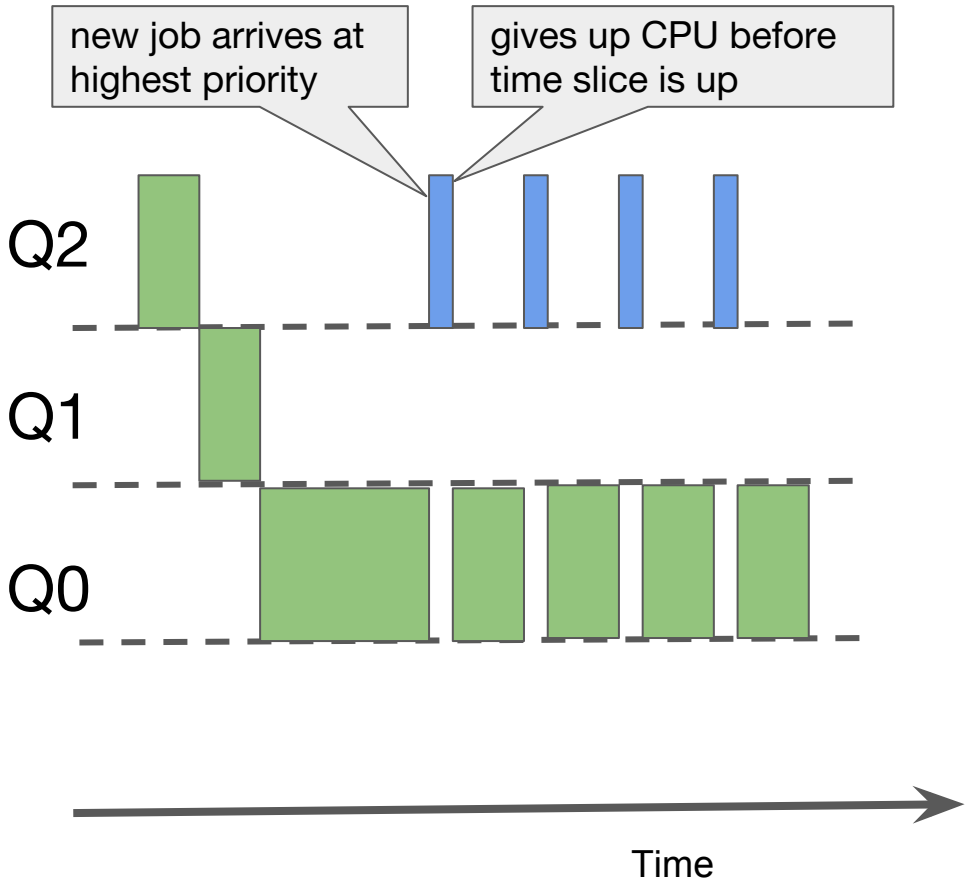
So **long-running** jobs should be **low-priority**.

But remember we **DON'T KNOW** how long a job is!

Example: single long-running job



Example: then an interactive job arrives



Both turnaround time and response time are minimized.

MLFQ first **assumes** every jobs is short.

- ➔ If it actually is short then good, run it quickly;
- ➔ if not, push down its priority and run it slowly; the longer the job is, the downer and slower.
  - ◆ i.e., MLFQ gradually **learns** that it is a long job.
- ➔ It is basically **approximating SJF.**

So it's all good… Right?

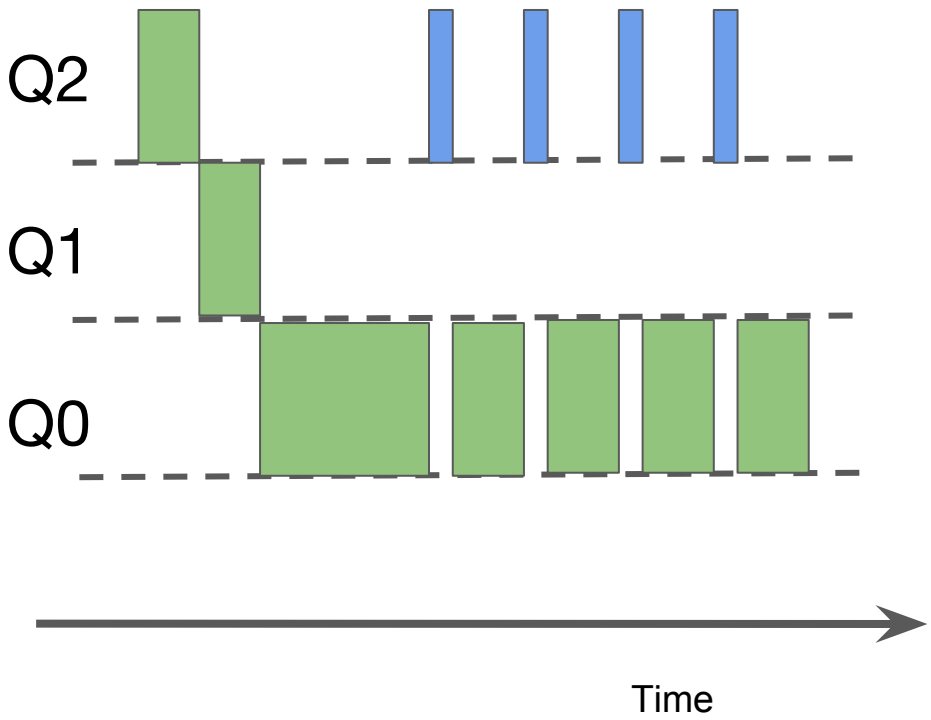
Rule 1: if Priority(A) > Priority(B), A runs (B doesn't)

Rule 2: if Priority(A) = Priority(B), run both in RR

Rule 3: When a job **enters** the system it is place at the **highest** priority.

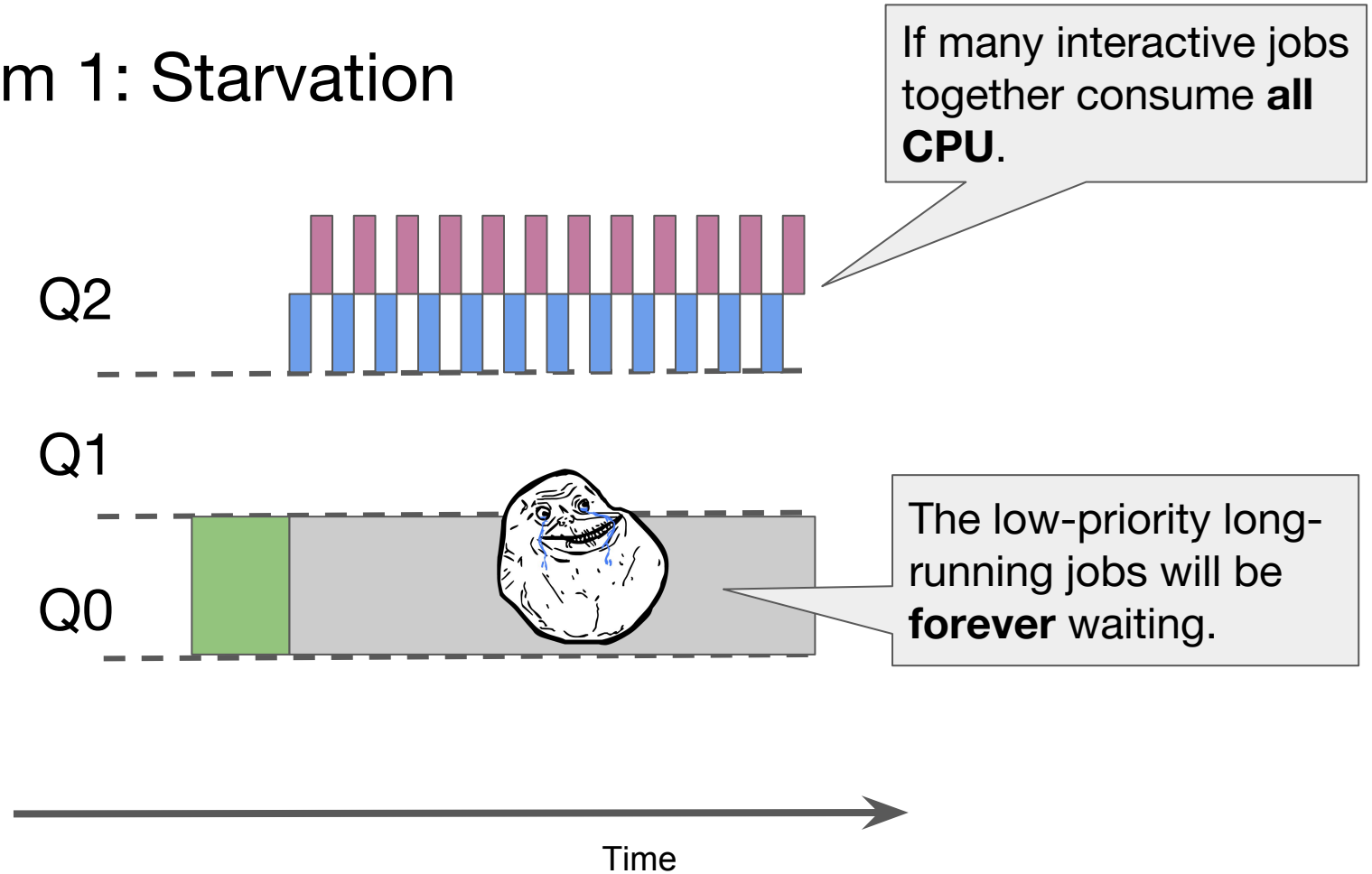
Rule 4:

- ➔ If a job uses to an **entire time slice** (of some defined length), its priority is **reduced** (move down one queue)
- ➔ If a job **gives up** the CPU **before** the time slice is up, it **stays** at the **same** priority level.



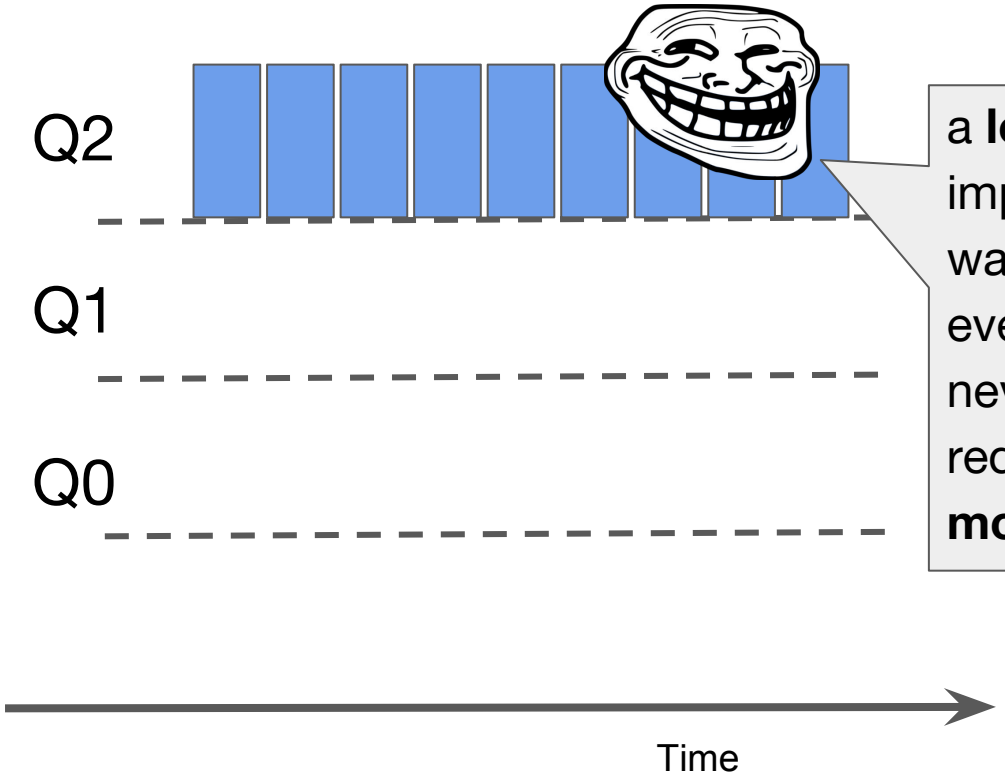
There are two problems!

Problem 1: Starvation



Problem 2: Gaming the scheduler

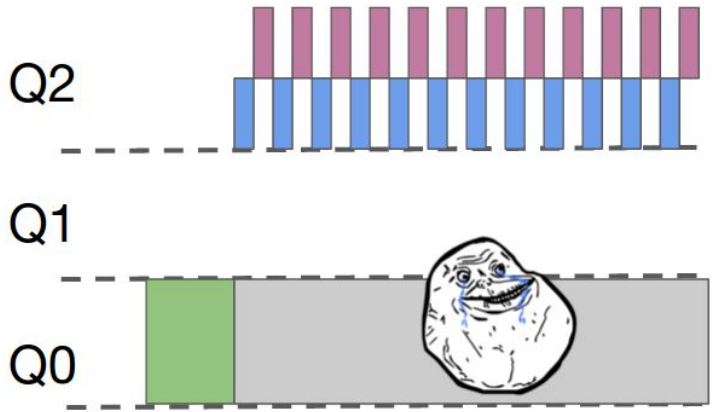
let's say the time slice for reducing priority is 100ms



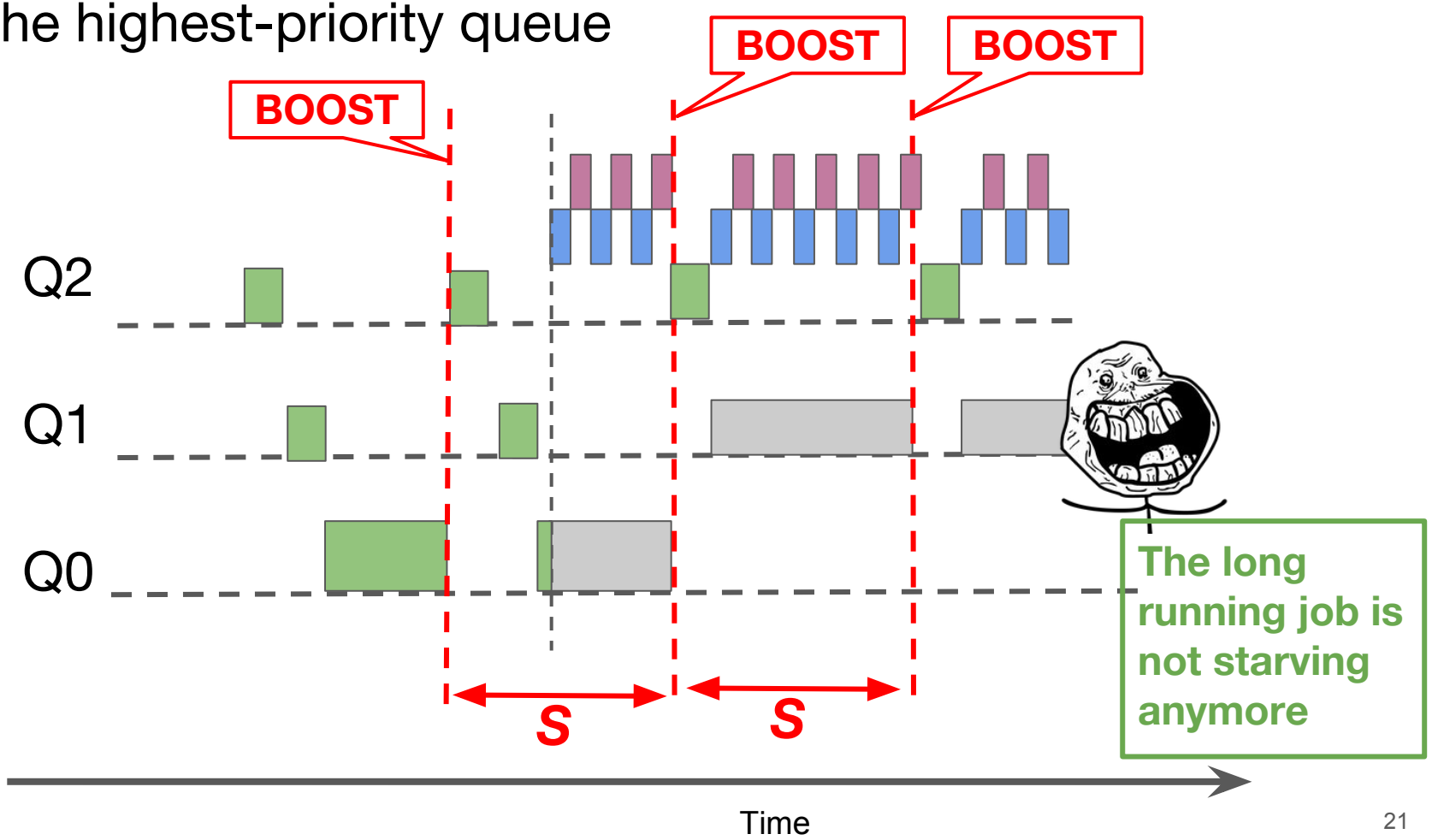
a **long-running** job can be implemented in such a way that it gives up CPU every **99ms**, therefore never gets priority reduced, and therefore **monopolizes** the CPU.

Solve Problem 1: Starvation -- **Priority Boost**

Rule 5: After some time period ***S***, move all jobs to the highest-priority queue



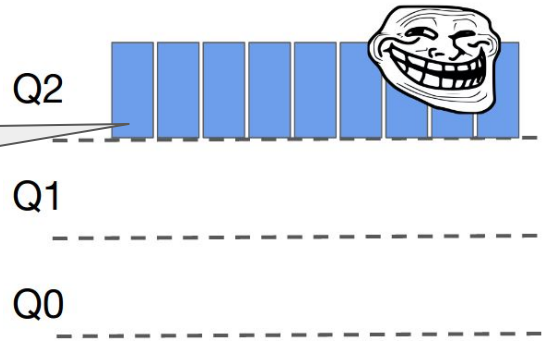
Rule 5: After some time period ***S***, move all jobs to the highest-priority queue



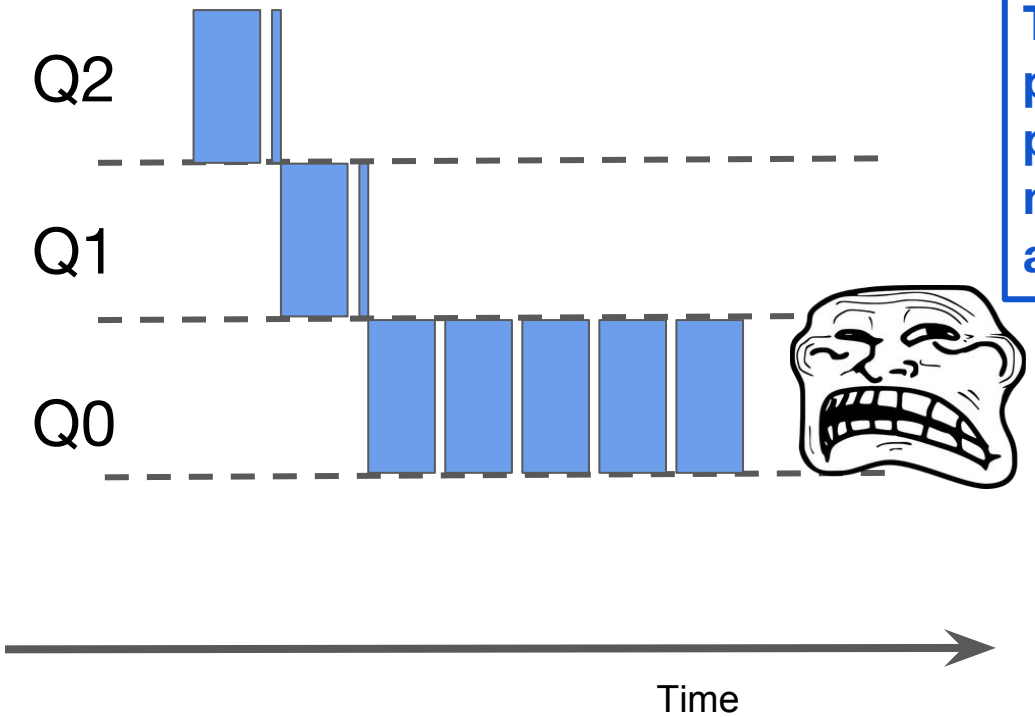
Solve Problem 2: Gaming -- **Better Accounting**

New Rule 4: The scheduler **remembers** of how much of a time slice a process has used at a given level; once a job **uses up** its time allotment at a given level, its priority is **reduced**.

The problem is that we **forget** about the history.



New Rule 4: The scheduler **remembers** of how much of a time slice a process has used at a given level; once a job uses up its time allotment at a given level, its priority is reduced.



The gaming process is pushed down lowest-priority, therefore not monopolizing the CPU anymore

Summary: The complete MLFQ

Rule 1: if Priority(A) > Priority(B), A runs (B doesn't)

Rule 2: if Priority(A) = Priority(B), run both in RR

Rule 3: When a job enters the system it is place at the highest priority.

Rule 4: If a job uses up its time allotment at a given level, its priority is reduced.

Rule 5: After some time period S, boost all jobs to the highest-priority queue.

Instead of demanding to know the lengths of jobs beforehand, MLFQ observes the execution of a job and gradually learns what type of job it is, and prioritize it accordingly.

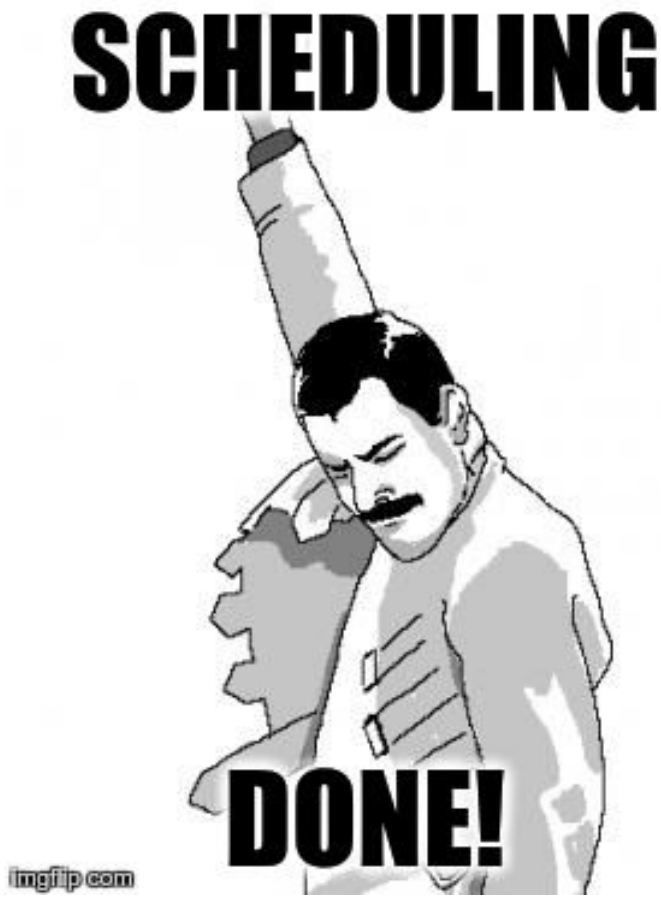
➔ Excellent performance for interactive I/O-bound jobs: good response time

➔ Fair for long-running CPU-bound jobs: good turnaround time without starvation

Used by many systems such as FreeBSD, Mac OS X, Solaris, Linux 2.6, Windows NT

Brief mention: other issues in scheduling

- Fair-share scheduling
  - instead of optimizing for turnaround time and response time, try to guarantee each job gets a certain percentage of CPU time; e.g., lottery scheduling.
- Multiprocessor scheduling
  - need to choose which CPU to use for a job
  - need to worry about concurrency


SCHEDULING
DONE!
imgflip.com

Intro to Virtual Memory

We are now here

Virtualization

Virtualizing CPU

- Processes
- Threads
- Scheduling ...

Virtualizing Memory

- **Address space**
- **Segmentation**
- **Paging ...**

Concurrency

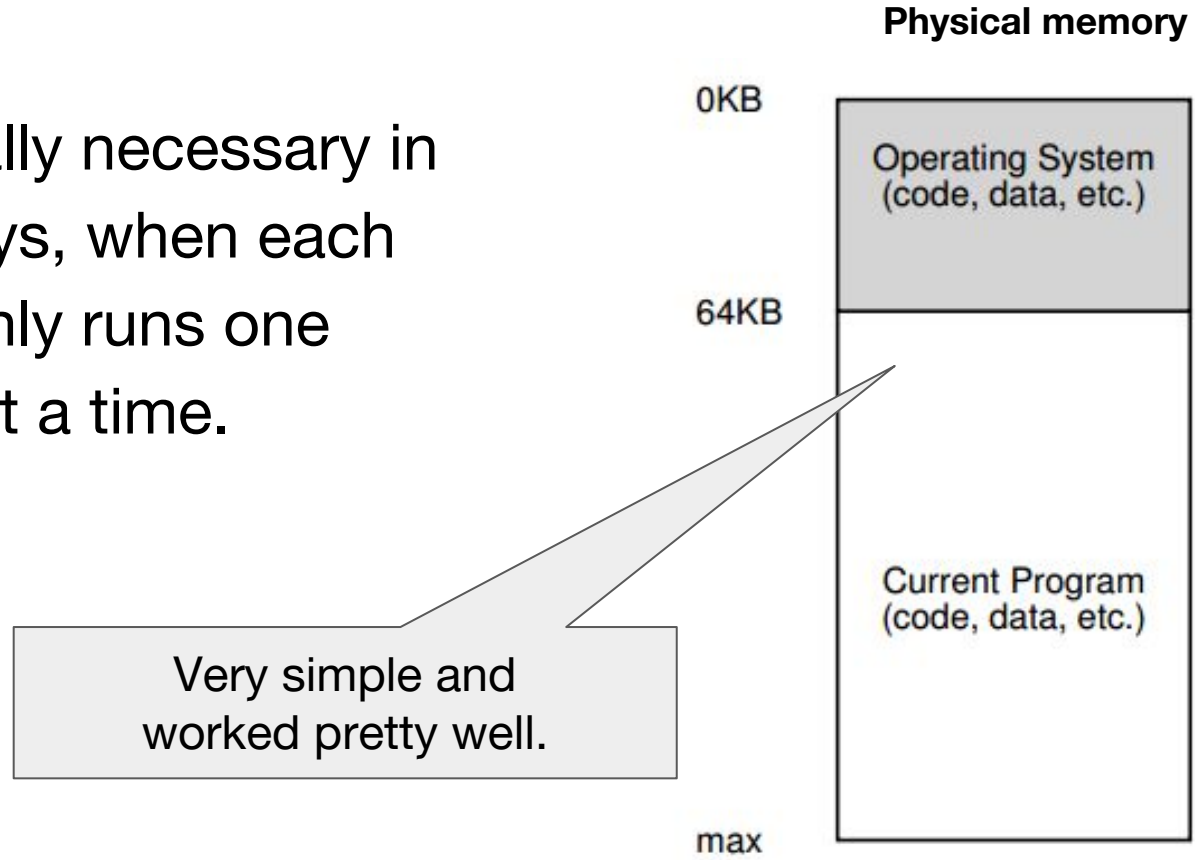
- Threads
- Locks
- Conditional variables
- Semaphores ...

Persistence

- File systems
- Journaling, logging
- I/O devices
- File integrity ...

Why virtualizing memory?

It wasn't really necessary in the early days, when each computer only runs one **single job** at a time.



But then there was multiprogramming...

Multiple programs are running on the computer at the same time, and they **share** the same **physical** memory space -- things became much trickier.

Goals:

- ➔ **Transparency**: processes does NOT see the real physical memory at all, they only see the **illusion** of a continuous chunk of easy-to-use memory.
- ➔ **Efficiency**: the cost of creating and maintaining this illusion should be low, i. e., it is fast in time and small in space.
- ➔ **Protection**: different processes should be **isolated** from each other, so that they don't mess up each other's memory space.

Memory virtualization: hide the ugly reality of physical memory and create the beautiful illusion of an easy-to-use, large, sparse and private space of memory.

Virtual Memory

Physical Memory

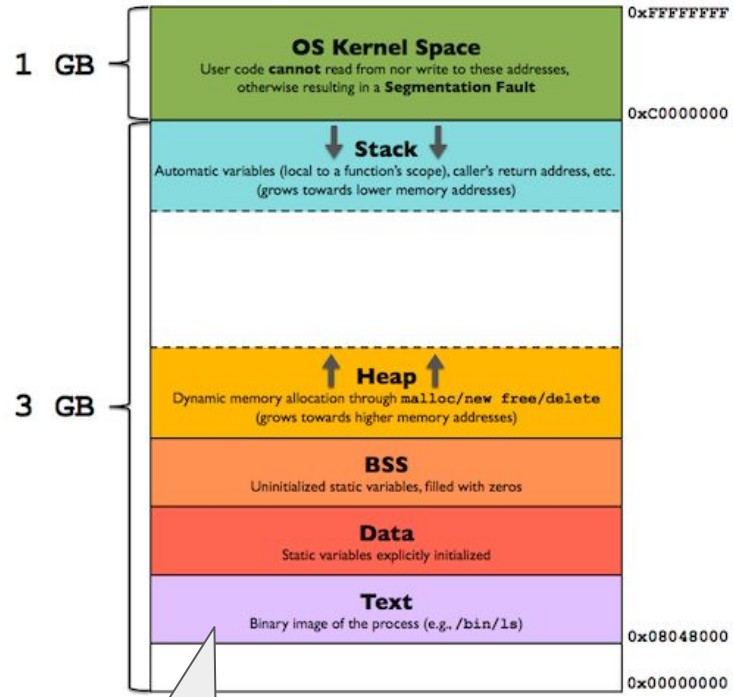
Every address a user sees is **virtual**

BONUS!

```
int main() {
  printf("addr of code: %p\n", (void*) main);
  printf("addr of heap: %p\n", (void*) malloc(1));
  int x = 3;
  printf("addr of stack: %p", (void*) &x);
  return x;
}
```

```
// run on a 32-bit machine
addr of code:  0x804844d
addr of heap:  0x9416008
addr of stack: 0xff83153c
```

virtual!



virtual!

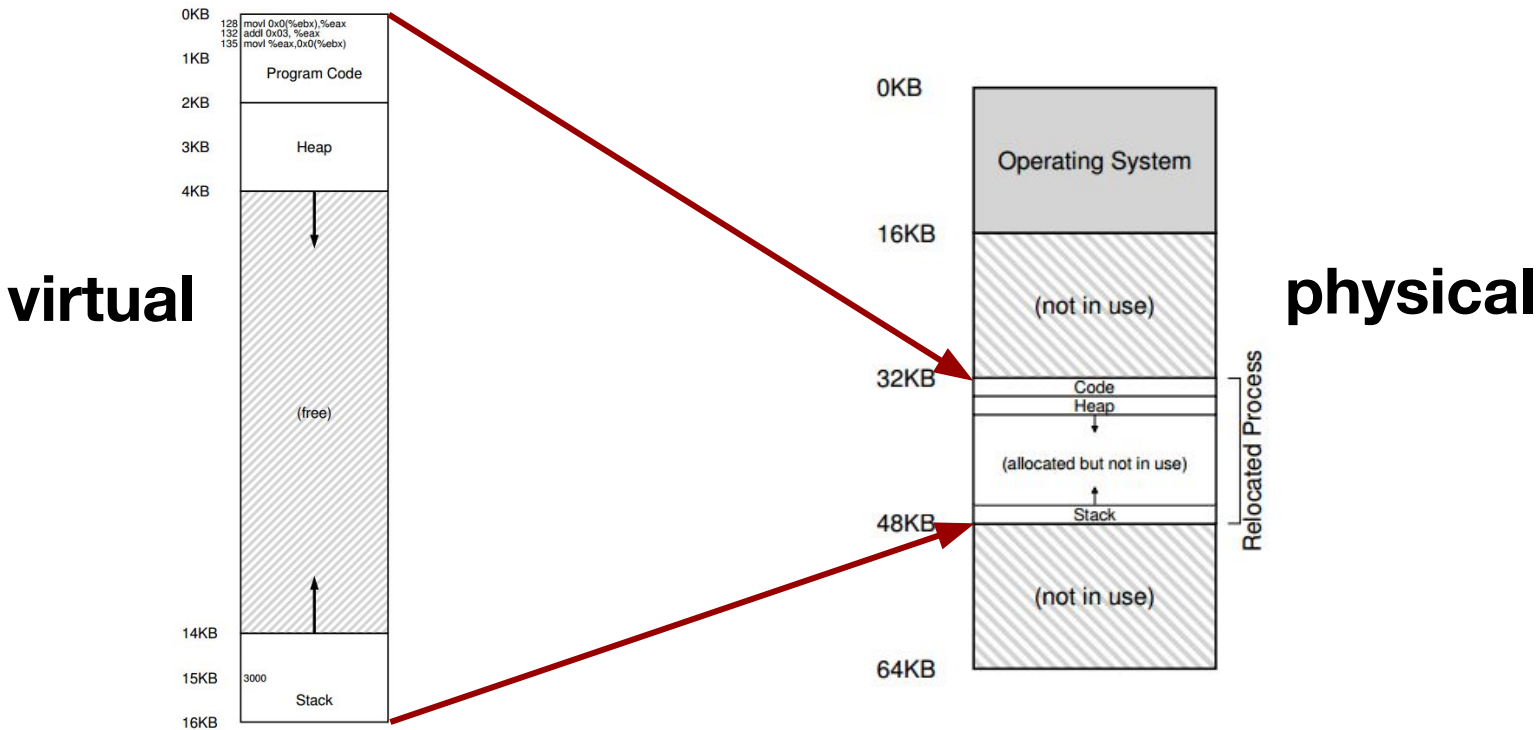
Outline of upcoming discussion

- ➔ Address binding
- ➔ Static relocation
- ➔ Dynamic relocation (base + bound)
- ➔ Segmentation
- ➔ Paging
- ➔ Multi-level paging (smaller page table)
- ➔ TLB (faster paging)
- ➔ Swapping (store memory content on disk)

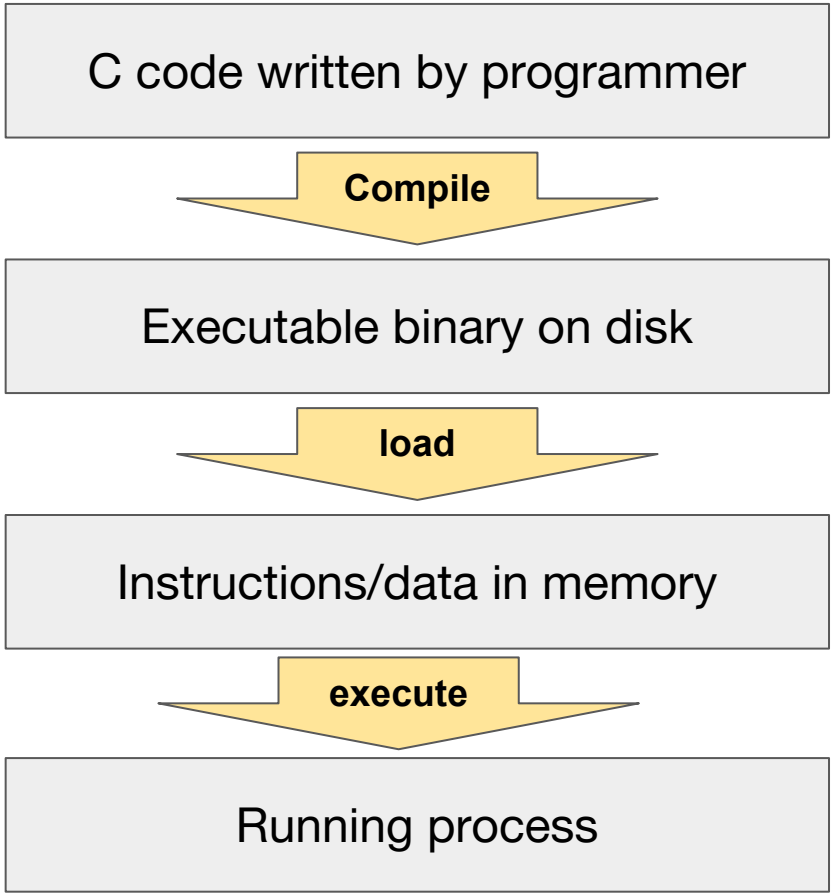
Each step solves a problem from its previous step, a new solution typically require some new kind of **hardware support**.

A question we will ask over and over again

Address translation: How does OS **translate virtual** memory address in a process' address space to the real **physical** memory address?



Quick Recap: A program's life cycle



First issue: Address Binding

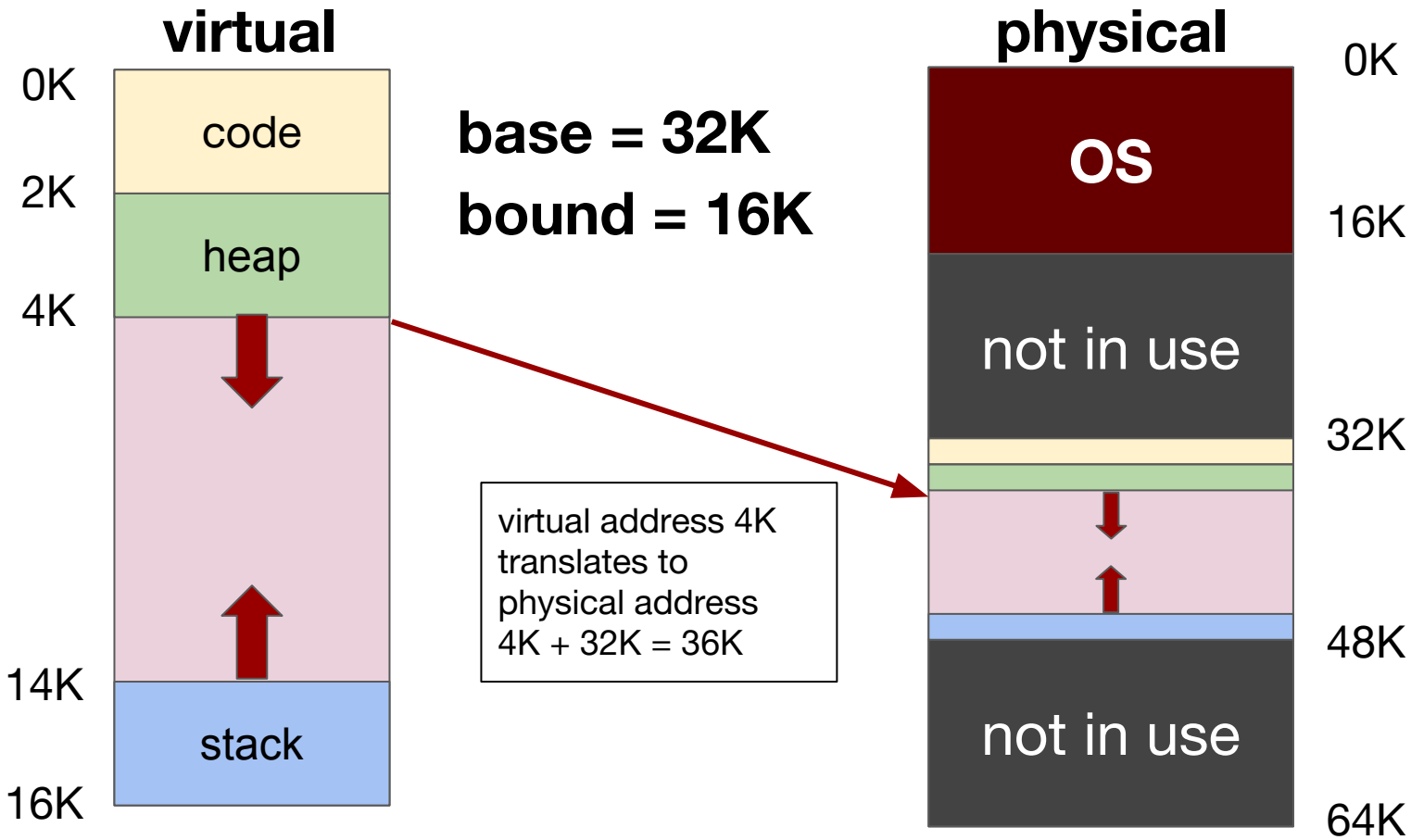
How does program know **where in physical memory** to locate when start running?

- Option 1: **Compile** time binding
  - The compiler translates all virtual address in the code into physical addresses stored in the executable.
  - Also called absolute code. No translation needed at runtime.
  - Used in MS-DOS and some simple embedded systems
- Option 2: **Load** time binding (aka static relocation, software relocation)
  - executable stores virtual address
  - **loader** loads executable into memory before running, and in the meantime translates all **virtual** addresses into **physical** addresses by **modifying** all the memory access instructions of the program.
  - No translation at runtime; loader needs to do a lot of work.

A better option for address binding

- ➔ Bind address at **execution** time (aka **dynamic relocation** or hardware relocation)
- ➔ Hardware support: two registers **base** and **bound** on CPU's Memory Management Unit (MMU).
- ➔ The executable stores **virtual** addresses (starting from 0)
- ➔ When a program starts running, OS decides where in physical to put the program's address space, and set the **base** register, then the physical address can be translated as the following
  - ◆ **physical address = base + virtual address**
  - ◆ Instructions in the executable do NOT need to be modified
- ➔ What is **bound** for? It's value ensures valid access within the process' address space, i.e., **physical addr > base + bound** cannot be accessed.
- ➔ Upon context switch, base and bound are stored in PCB, since one CPU has only one pair of base-bound registers.

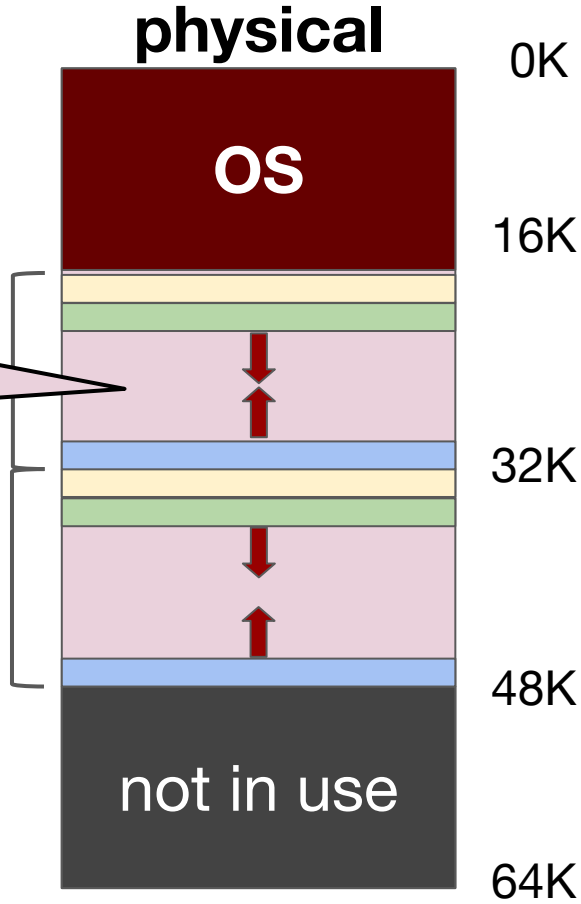
Example: 16K virtual address space relocated into 64K physical memory



So we can just keep relocating process address spaces like this. Any problem?

Problem 1: This is (big) free space that cannot be used by other processes anymore. What a waste! This problem is called **internal fragmentation.**

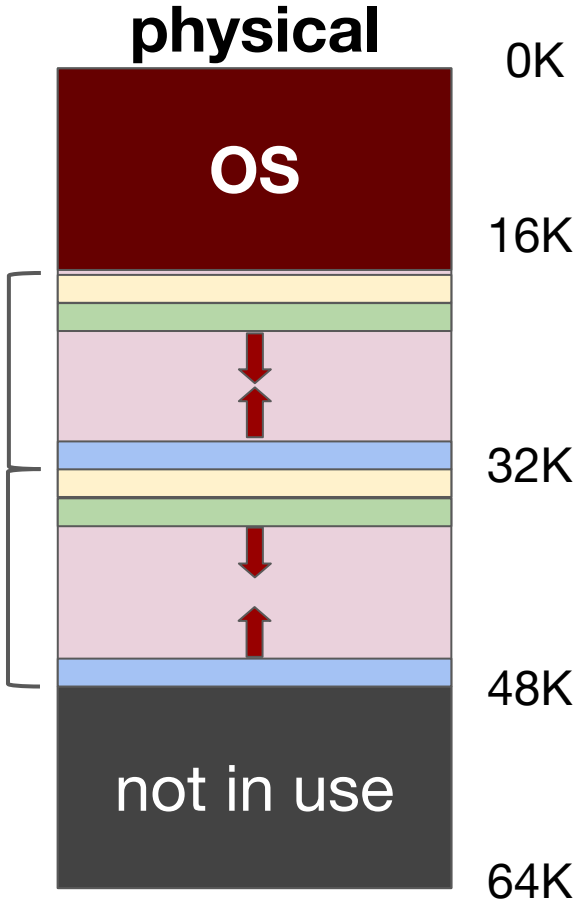
Problem 2: What if the virtual address space is **larger** than the size of the physical memory? E.g., 32-bit address space is 4GB but physical memory can be only 2GB. Not to mention 64-bit address space, which is larger than the physical memory size of any computer for sure!



So we can just keep relocating process address spaces like this. Any problem?

Solution:
Relocate code, heap, stack **independently**!

This is called **segmentation**.

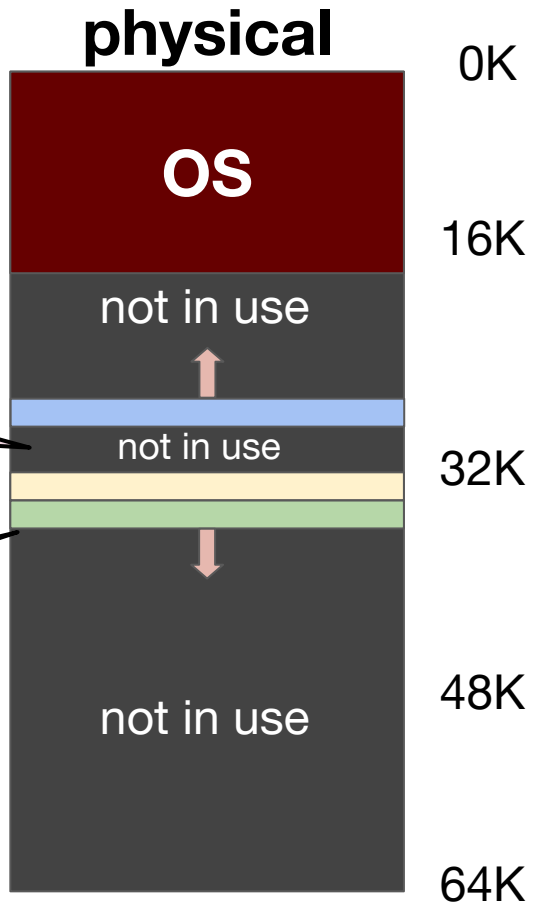


Segmentation

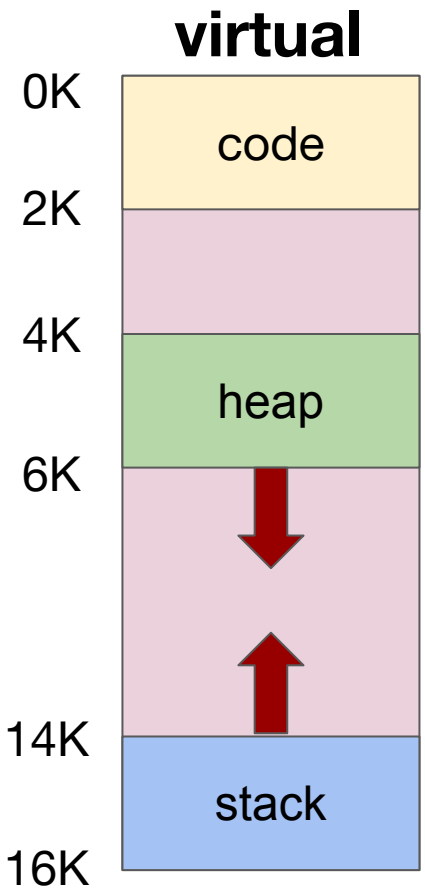
- ➔ Relocate the three segments: code, heap and stack independently
- ➔ Hardware support: base-bound registers for each of the three segments.

All "**not in use**" area can be used to relocate other processes' address space. No more **wasted** free space.

Only need to allocate the **actually used** memory.



Segmentation: Address Translation



seg	base	bound/size
code	32K	2K
heap	34K	2K
stack	28K	2K

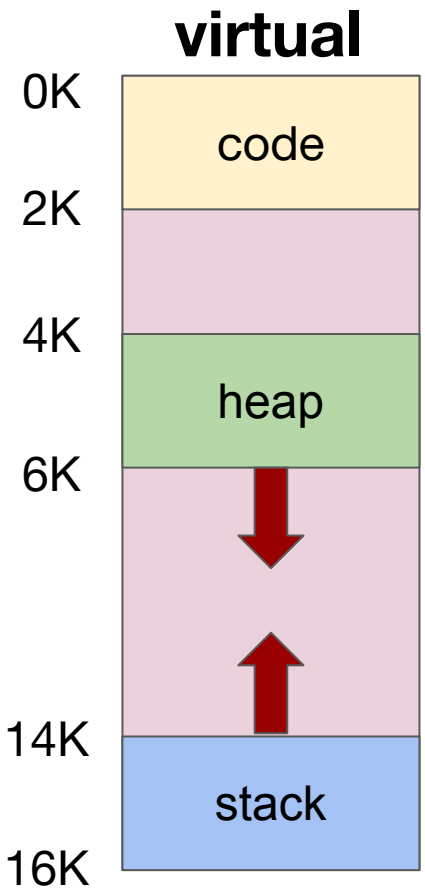
Virtual address: 100

➔ check: 100 in code segment

➔ physical = code base + offset

- 32K + 100 = **32868**

Segmentation: Address Translation



seg	base	bound/size
code	32K	2K
heap	34K	2K
stack	28K	2K

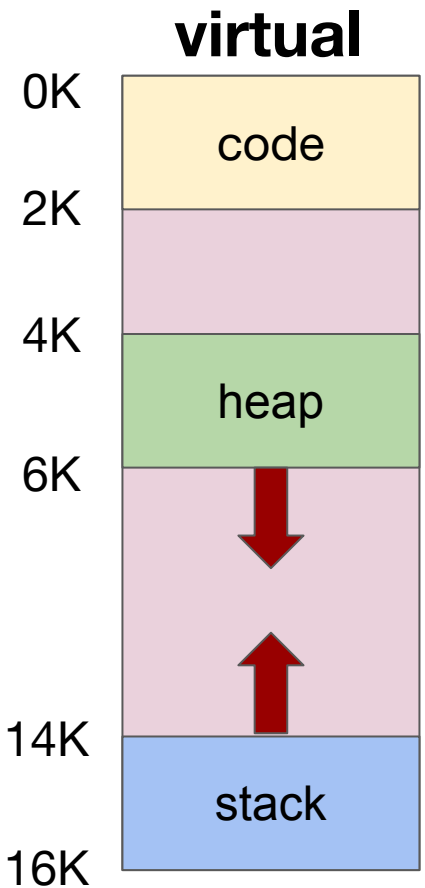
Virtual address: 4200

➔ check: 4200 in heap segment

➔ physical = heap base + offset

- 34K + (4200 - 4K) = **34920**

Segmentation: Address Translation



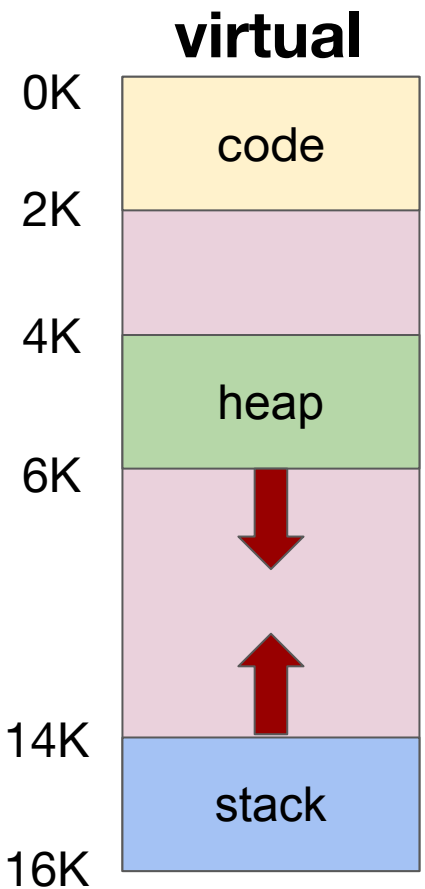
seg	base	bound/size
code	32K	2K
heap	34K	2K
stack	28K	2K

Virtual address: 7000

➔ check: not in any segment

➔ **Segmentation fault**

Segmentation: Address Translation



seg	base	bound/size
code	32K	2K
heap	34K	2K
stack	28K	2K

Virtual address: 15K ?

➔ For home thinking / reading

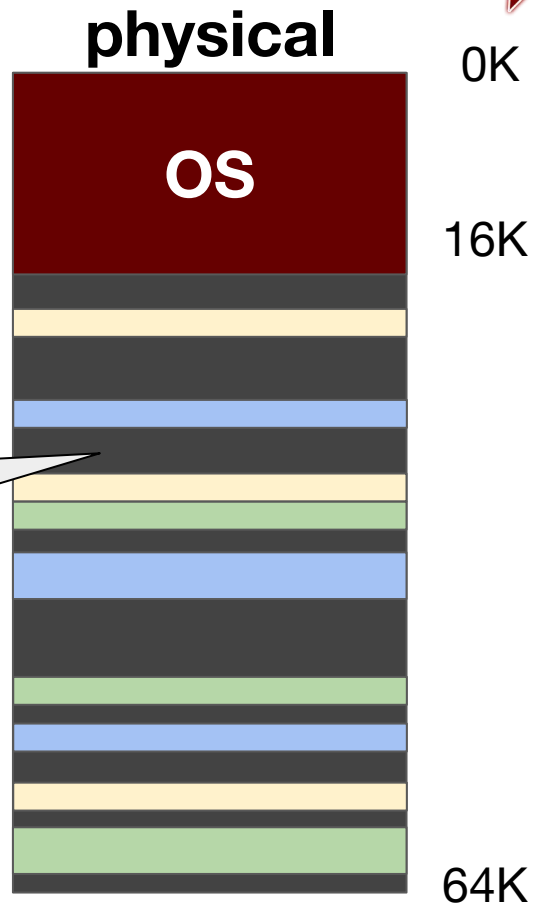
Issues with segmentation

OS need to able to find free space to relocate new processes' address spaces.

After a few relocations the physical memory may look like this.

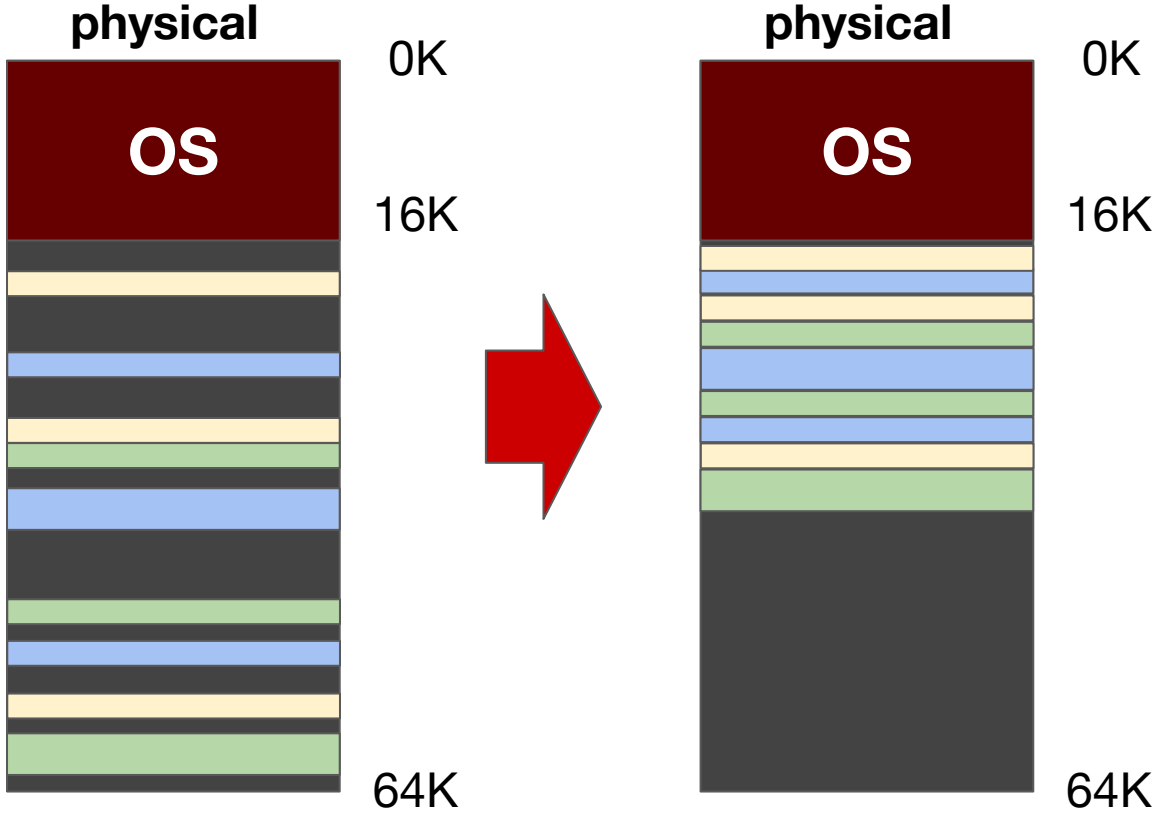
Finding new free spaces becomes increasingly difficult, or even impossible (when?).

This is called **external fragmentation.**



Solution #1 to external fragmentation

Compact the segments once in awhile.



Copy all segments' content to a contiguous region of memory, then update the base registers of all segments.

This is **expensive**!

Solution #2 to external fragmentation

Be very clever when allocating new address space.

Using smart **free-list** management algorithms, e.g.,

- ➔ best-fit
- ➔ worst-fit
- ➔ first-fit
- ➔ next-fit
- ➔ buddy algorithm

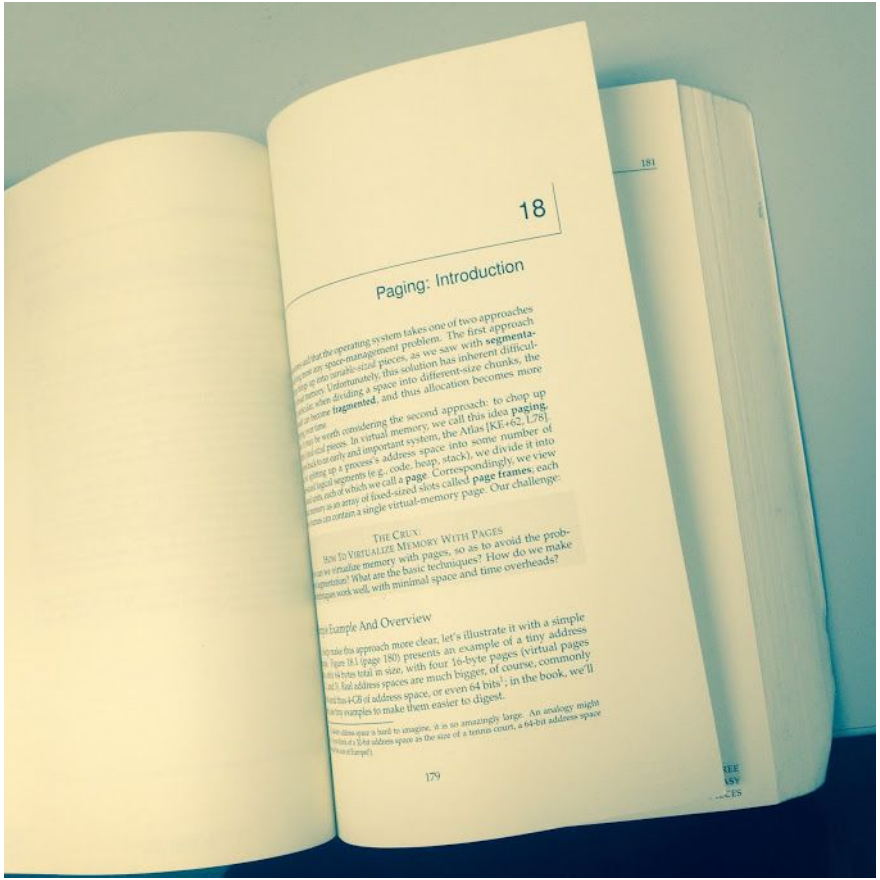
a data structure that keeps track of the free memory regions

However, these algorithms do NOT guarantee eliminating external fragments. They just minimize it as much as they can.

Segmentation is hard! Need a new idea!

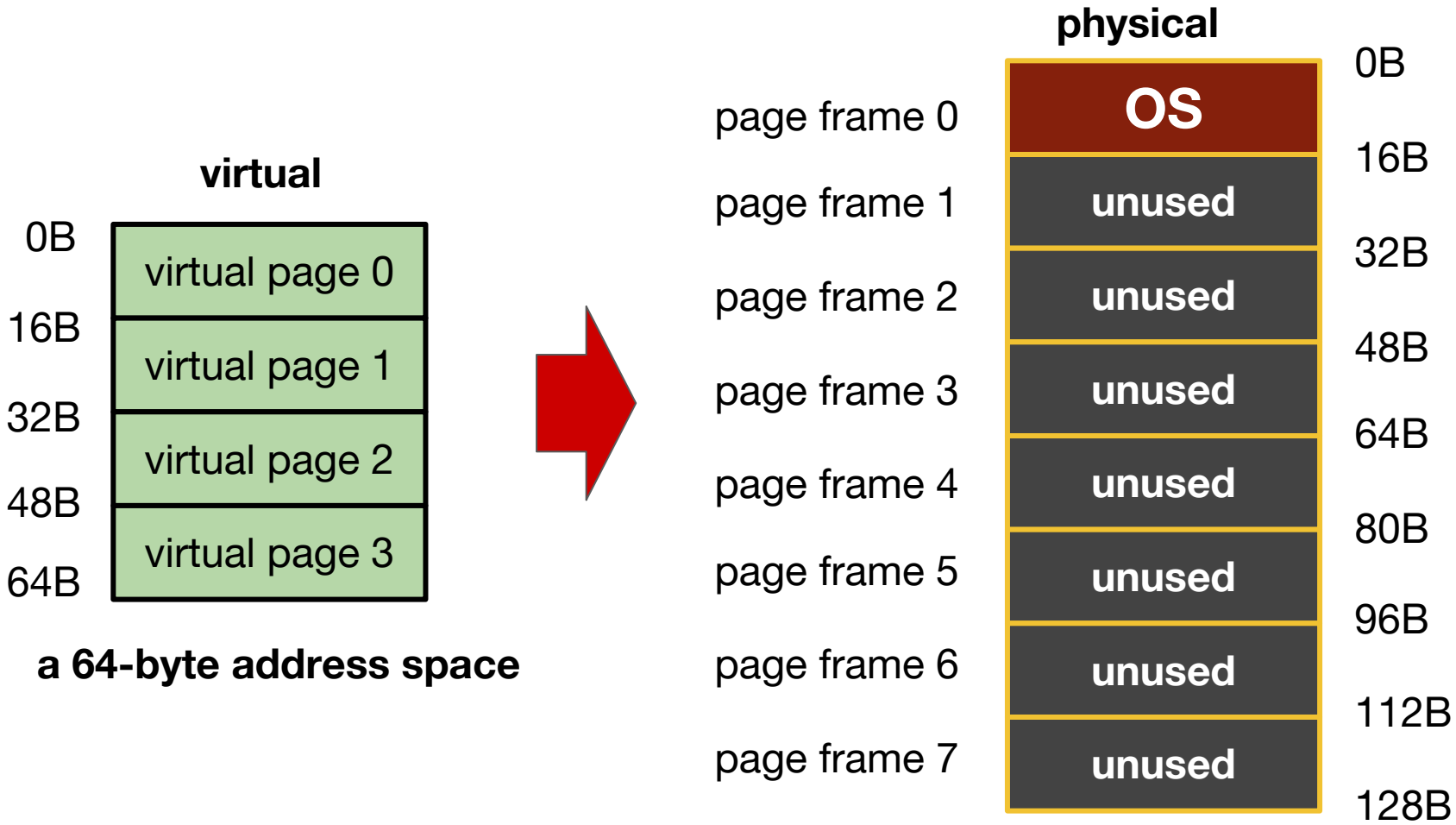
Paging

Not really a new idea...



- ➔ Divide content into **fixed-sized** pages.
  - ➔ Each page has a page number.
  - ➔ Locate content in a page by an offset, e.g., “10th word of Page 8”, …
  - ➔ There is a “table” which tells you which content is in which page.

Tiny example: 128-byte physical memory with 16-byte pages

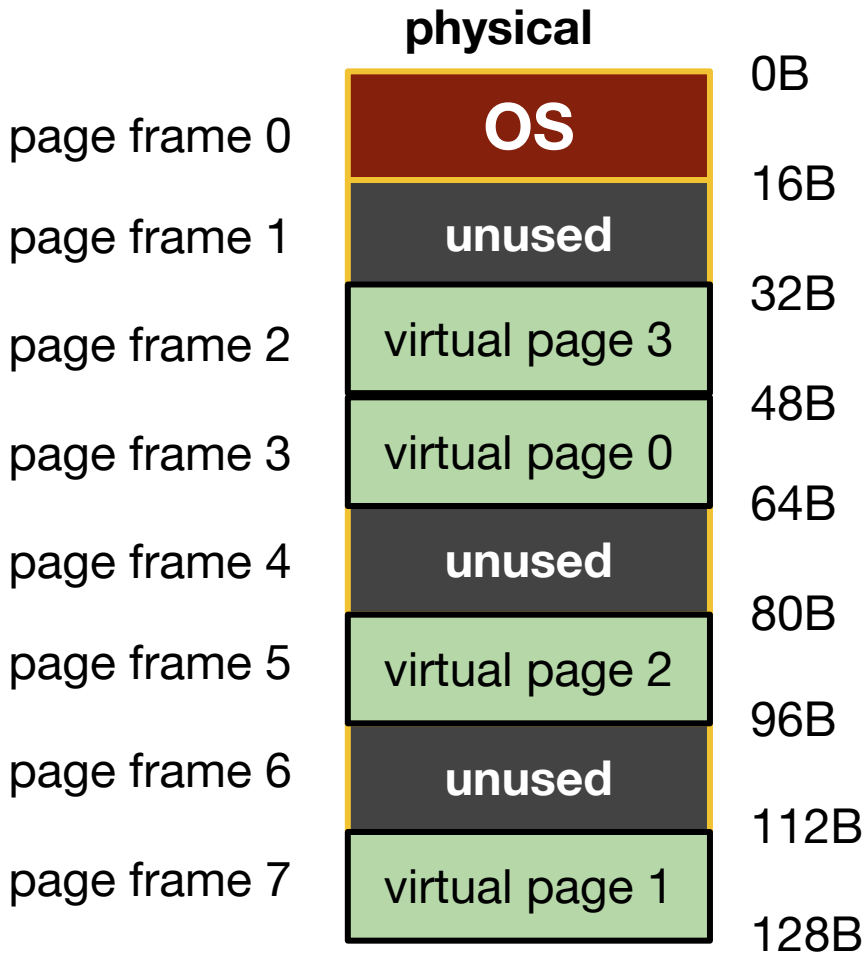


After allocation

For allocation management, OS keeps track a free list of **free (fixed sized) pages**.

➔ This is much simpler than keeping a free list of variable-sized memory regions as in segmentation.

Virtual pages are **numbered**, which preserves the **order** of the address space, so we can feel free to allocate virtual pages **all over the place** in the physical memory.

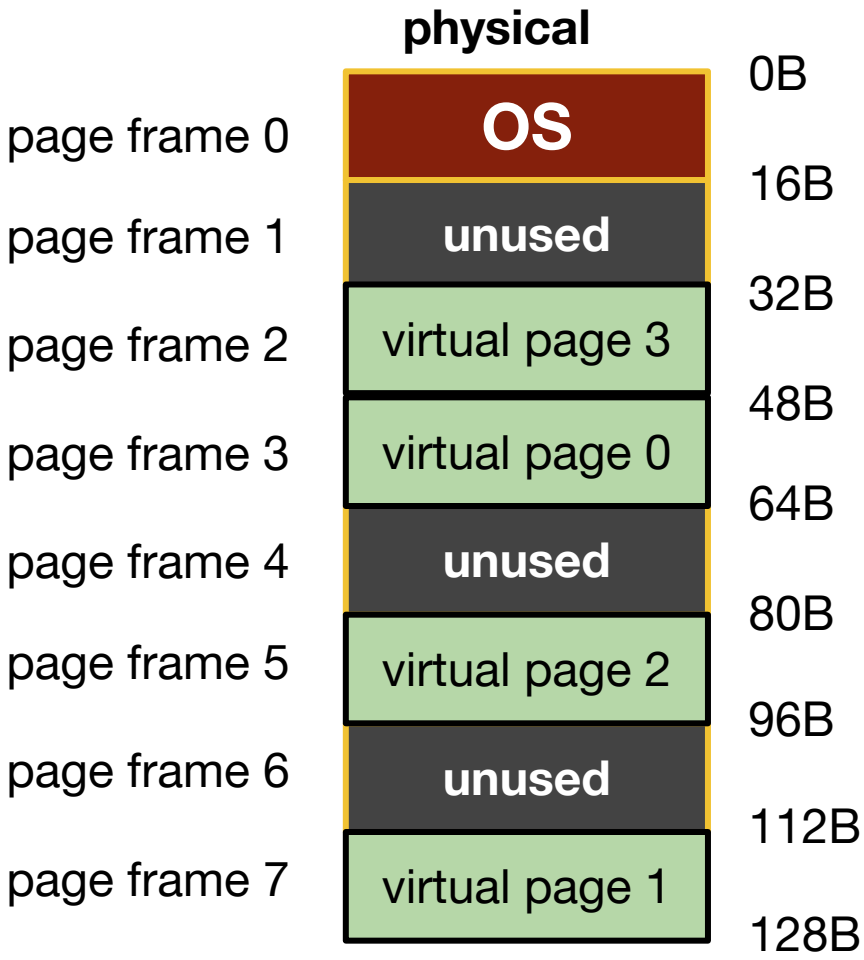


Data structures needed

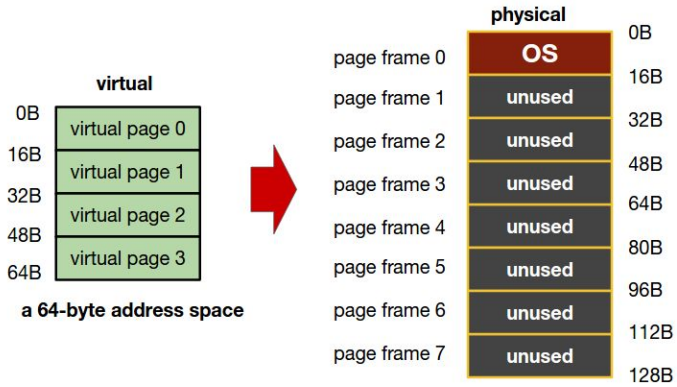
Page table: mapping from virtual page number of physical page frame.

- VP0 -> PF3
- VP1 -> PF7
- VP2 -> PF5
- VP3 -> PF2

Each process has its own page table.



Address Translation with Paging



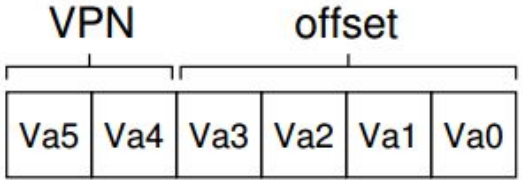
Basically, you need to know the page number and an offset within that page.

For the tiny example, how many address bits are needed?

➔ 6 bits, because 64-byte address space and 2^6 = 64

What do you use these 6 bits for?

➔ 2 bits virtual page number (VPN), 4 bits for offset (which is enough because pages are 16-byte and 2^4 = 16)



So the translation:

physical address =

pagetable[VPN] * page_size + offset

Next week

- ➔ Midterm
- ➔ Paging continued...

This week's tutorial:

- ➔ programming exercise on memory tracing