CS 5600 Computer Systems

Spring 2026

Virtualization: Scheduling
Lecture 3.1
January 27, 2026

Prof. Scott Valcourt

s.valcourt@northeastern.edu

603-380-2860 (cell)

Portland, ME

The Roux Institute
Northeastern University

CPU Virtualization: Two Components

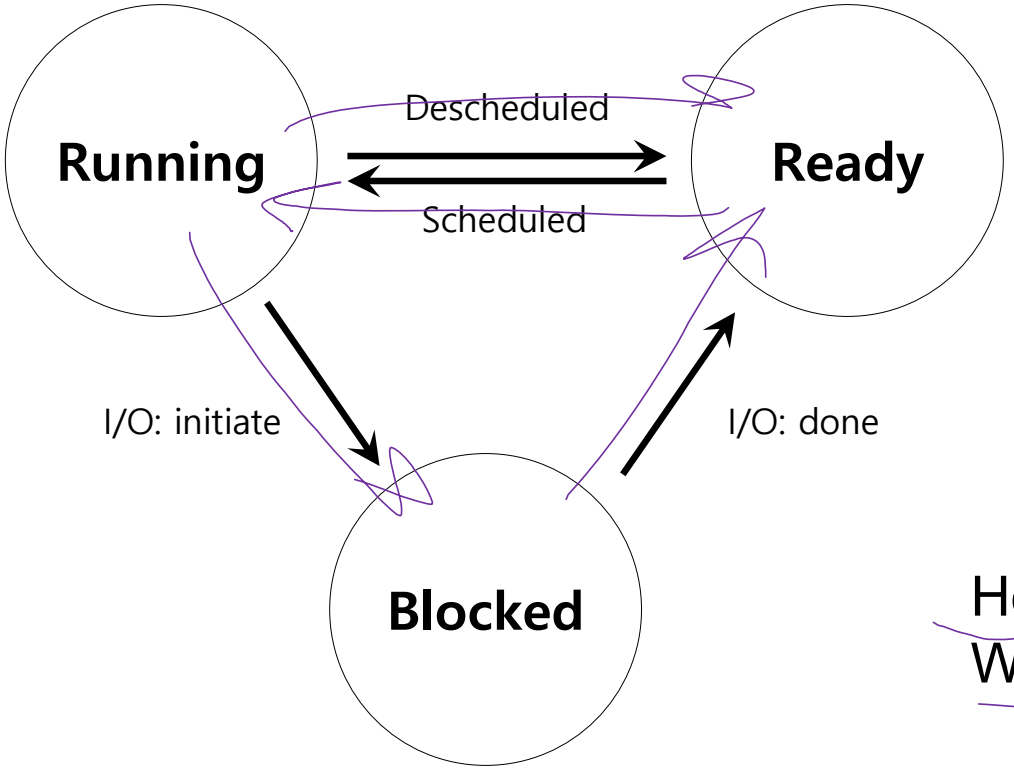
Dispatcher

- Low-level mechanism
- Performs context-switch
  - Switch from user mode to kernel mode
  - Save execution state (registers) of old process in PCB
  - Insert PCB in ready queue
  - Load state of next process from PCB to registers
  - Switch from kernel to user mode
  - Jump to instruction in new user process

Scheduler

- Policy to determine which process gets CPU when

Review: State Transitions



How to transition? ("mechanism")
When to transition? ("policy")

Vocabulary

Workload: set of **job** descriptions (arrival time, run_time)

- Job: Viewed as the current CPU burst of a process
- Process alternates between CPU and I/O
  process moves between ready and blocked queues

Scheduler: logic that decides which ready job to run

Metric: measurement of scheduling quality

Scheduling Performance Metrics

Minimize turnaround time
- Do not want to wait long for job to complete
- Completion_time – arrival_time

Minimize response time
- Schedule interactive jobs promptly so users see output quickly
- Initial_schedule_time – arrival_time

Minimize waiting time
- Do not want to spend much time in Ready queue

Maximize throughput
- Want many jobs to complete per unit of time

Maximize resource utilization
- Keep expensive devices busy

Minimize overhead
- Reduce number of context switches

Maximize fairness
- All jobs get same amount of CPU over some time interval

MGHPCC

Workload Assumptions

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Example: workload, scheduler, metrics

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

FIFO: First In, First Out
- also called FCFS (first come first served)
- run jobs in *arrival_time* order

What is our turnaround?: *completion_time* - *arrival_time*

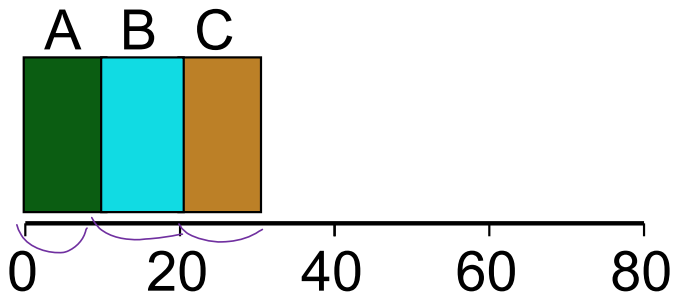
FIFO Event Trace

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10

Time	Event
0	A arrives
0	B arrives
0	C arrives
0	run A
10	complete A
10	run B
20	complete B
20	run C
30	complete C

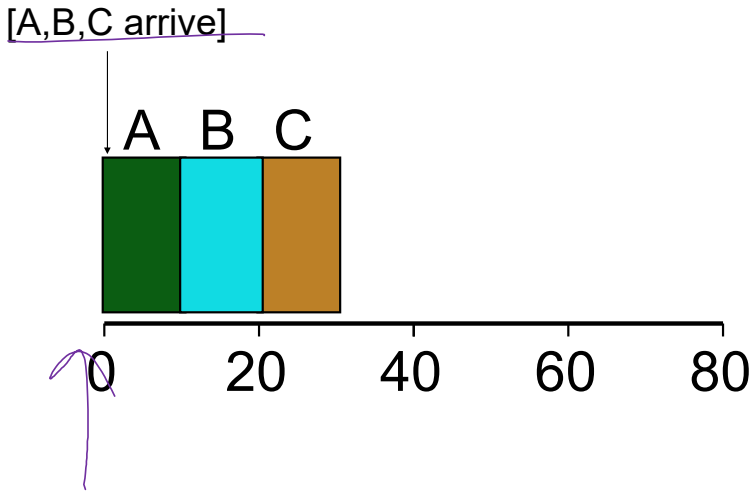
FIFO (Identical Jobs)

JOB	arrival_time (s)	run_time (s)
A	~0	10
B	~0	10
C	~0	10



Gantt chart:
Illustrates how jobs are scheduled over time on a CPU

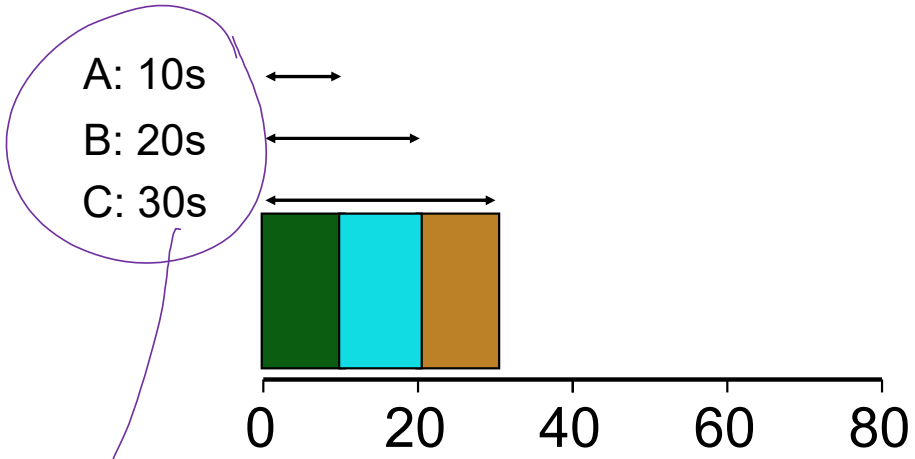
FIFO (Identical Jobs)



What is the average turnaround time?

Def: *turnaround_time* = *completion_time* - *arrival_time*

FIFO (Identical Jobs)



What is the average turnaround time?

Def: *turnaround_time* = *completion_time* - *arrival_time*

(10 + 20 + 30) / 3 = **20s**

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

1. ~~Each job runs for the same amount of time~~
2. All jobs arrive at the same time
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Any Problematic Workloads for FIFO?

- **Workload**: ?
- **Scheduler**: FIFO
- **Metric**: turnaround is high

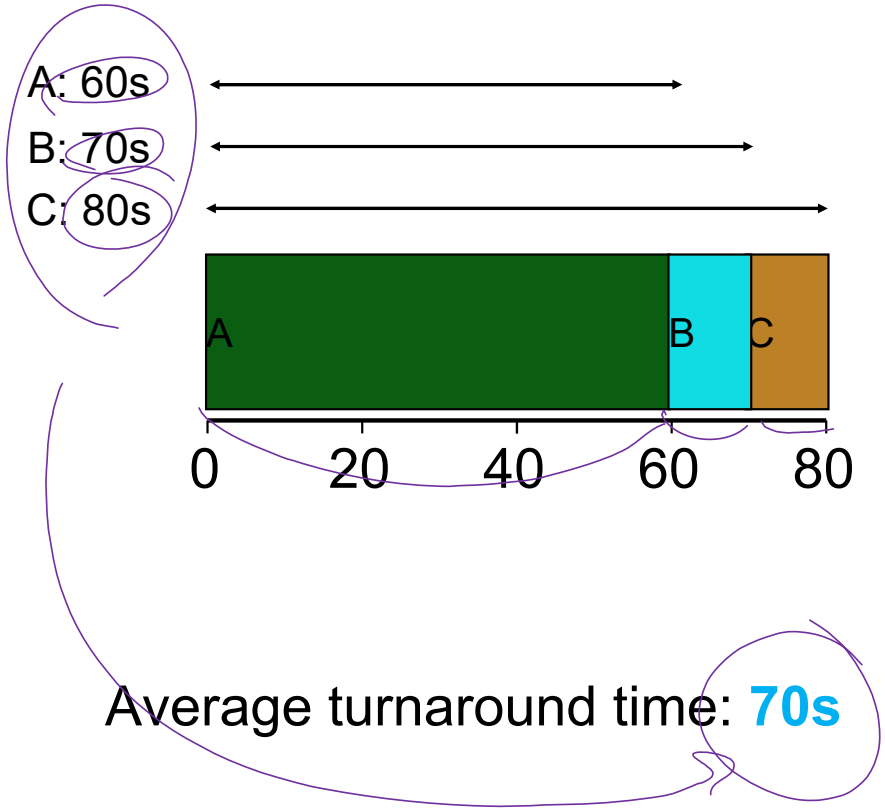
Example: Big First Job

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10

Draw Gantt chart for this workload and policy…

What is the average turnaround time?

Example: Big First Job



Average turnaround time: **70s**

Convoy Effect – Passing the Tractor

Problem with Previous Scheduler:

FIFO: Turnaround time can suffer when short jobs must wait for long jobs

New scheduler:

SJF (Shortest Job First)

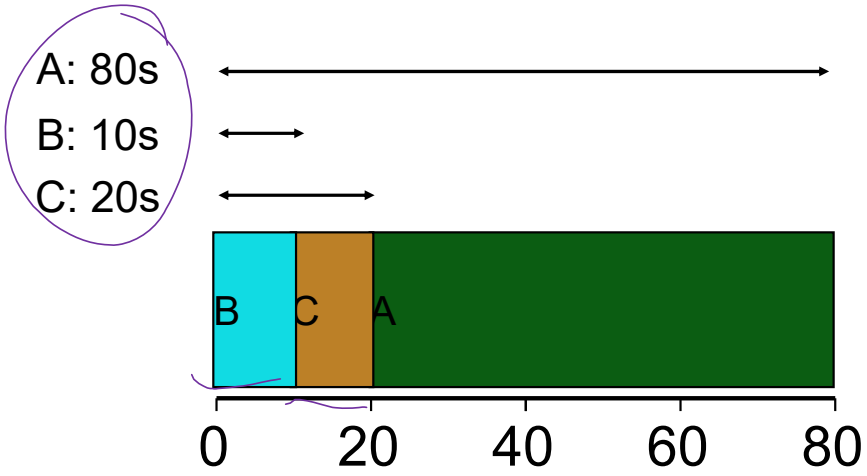
Choose job with smallest *run_time*

Example: Shortest Job First

JOB	arrival_time (s)	run_time (s)
A	~0	**60**
B	~0	10
C	~0	10

What is the average turnaround time with SJF?

SJF Turnaround Time



What is the average turnaround time with SJF?

(80 + 10 + 20) / 3 = ~36.7s

Average turnaround with FIFO: 70s

For minimizing average turnaround time (with no preemption): SJF is provably optimal

Moving shorter job before longer job improves turnaround time of short job more than it harms turnaround time of long job

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

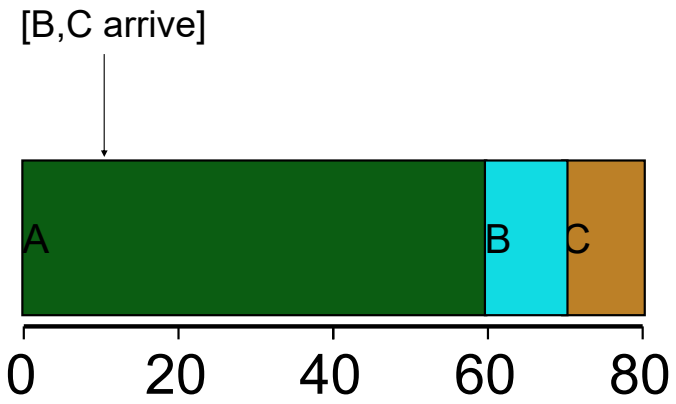
~~1. Each job runs for the same amount of time~~
~~2. All jobs arrive at the same time~~
3. All jobs only use the CPU (no I/O)
4. Run-time of each job is known

Example: Shortest Job First (Arrival Time)

JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time with SJF?

Stuck Behind that Tractor Again



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time?

(60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

Preemptive Scheduling

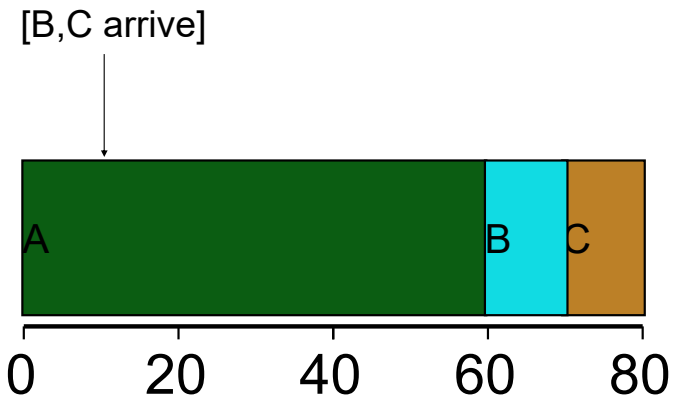
Prev schedulers:

- FIFO and SJF are non-preemptive
- Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)

New scheduler:

- Preemptive: Potentially schedule different job at any point by taking CPU away from running job
- STCF (Shortest Time-to-Completion First)
- Always run a job that will complete the quickest

Non-Preemptive: SJF

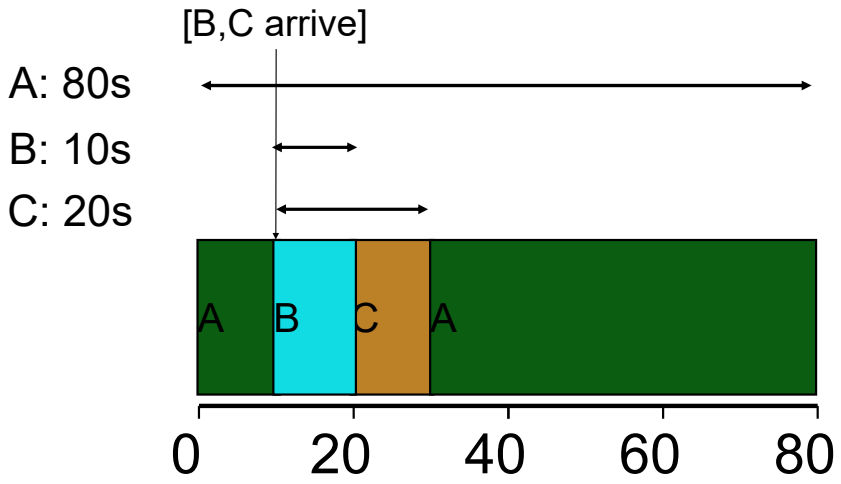


JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

What is the average turnaround time?

(60 + (70 – 10) + (80 – 10)) / 3 = **63.3s**

Preemptive: STCF



JOB	arrival_time (s)	run_time (s)
A	~0	60
B	**~10**	10
C	**~10**	10

Average turnaround time with STCF?

36.6s

Average turnaround time with SJF: **63.3s**

Scheduling Basics

Workloads:
- arrival_time
- run_time

Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

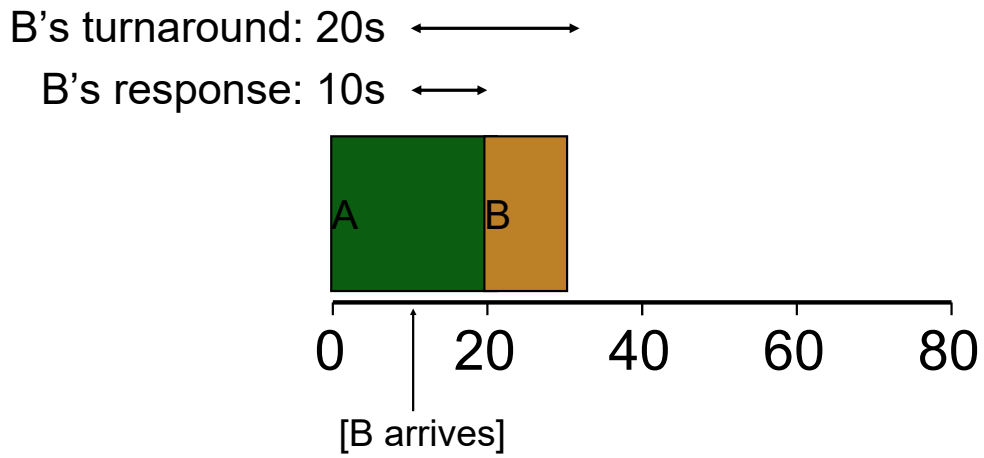
Response Times

Sometimes care about when job starts instead of when it finishes

New metric:

response_time = *first_run_time* - *arrival_time*

Response versus Turnaround

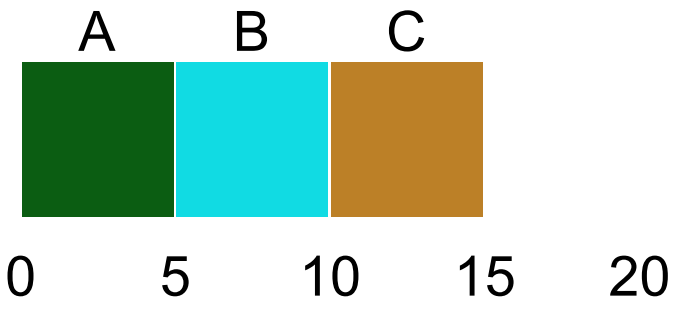


Round Robin Scheduler

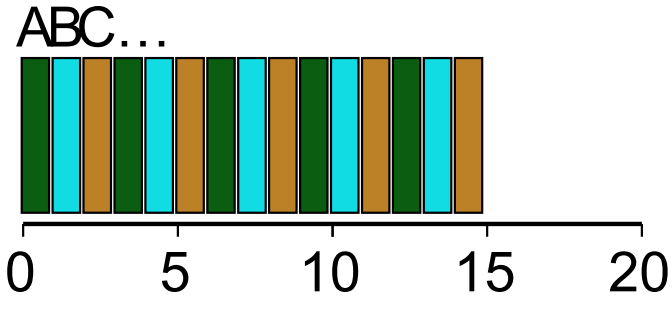
Prev schedulers:
FIFO, SJF, and STCF can have poor response time

New scheduler: RR (Round Robin)
Alternates ready processes every fixed-length time-slice

FIFO versus RR



Avg Response Time?
(0+5+10)/3 = **5**



Avg Response Time?
(0+1+2)/3 = **1**

In what way is RR worse?
Average turn-around time with equal job lengths is horrible

Other reasons why RR could be better?
If don't know run-time of each job, gives short jobs a chance to run and finish fast

Scheduling Basics

Workloads:
- arrival_time
- run_time

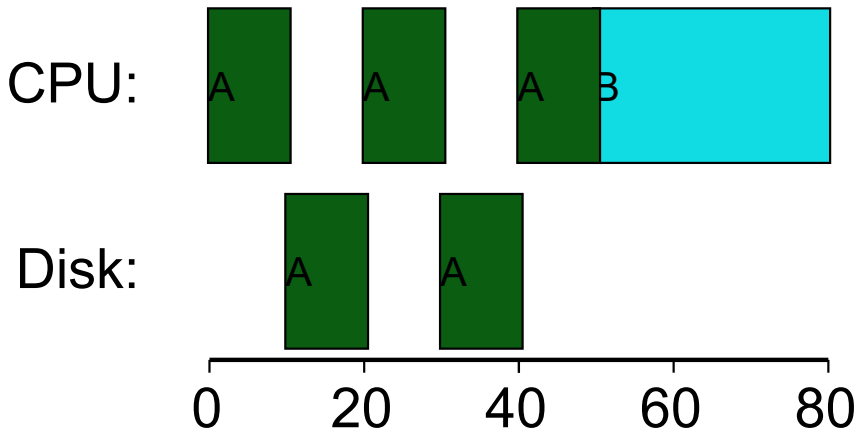
Schedulers:
- FIFO
- SJF
- STCF
- RR

Metrics:
- turnaround_time
- response_time

Workload Assumptions

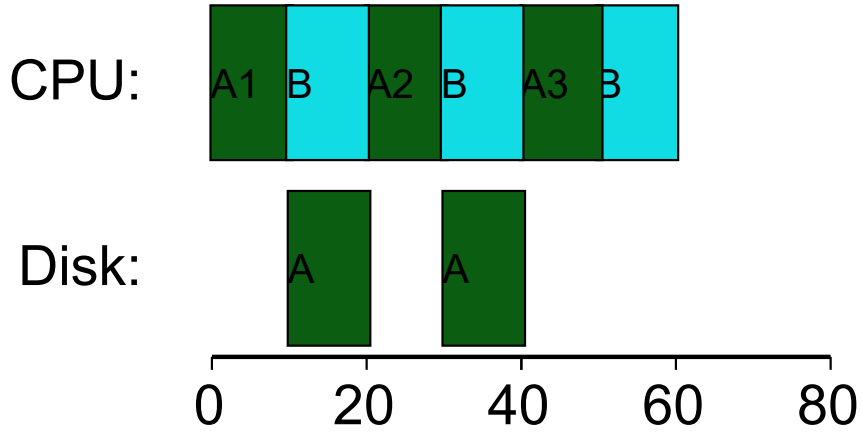
~~1. Each job runs for the same amount of time~~
~~2. All jobs arrive at the same time~~
~~3. All jobs only use the CPU (no I/O)~~
4. Run-time of each job is known

Not I/O Aware



Don’t let Job A hold on to CPU while blocked waiting for disk

I/O Aware (Overlap)



Treat Job A as 3 separate CPU bursts
When Job A completes I/O, another Job A is ready

Each CPU burst is shorter than Job B, so with SCTF, Job A preempts Job B

Workload Assumptions

~~1. Each job runs for the same amount of time~~
~~2. All jobs arrive at the same time~~
~~3. All jobs only use the CPU (no I/O)~~
~~4. Run-time of each job is known~~

(need smarter, fancier scheduler)

MLFQ (Multi-Level Feedback Queue)

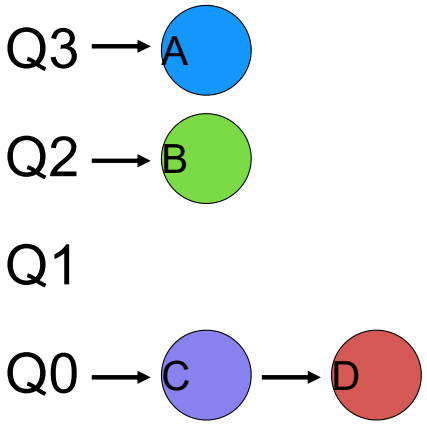
Goal: general-purpose scheduling

Must support two job types with distinct goals
- "interactive" programs care about response time
- "batch" programs care about turnaround time

Approach: multiple levels of round-robin; each level has higher priority than lower levels and preempts them

Priorities

Rule 1: If priority(A) > Priority(B), A runs
Rule 2: If priority(A) == Priority(B), A & B run in RR



"Multi-level"

How to know how to set priority?

Approach 1: nice
Approach 2: history "feedback"

History

- Use past behavior of process to predict future behavior
  - Common technique in systems
- Processes alternate between I/O and CPU work
- Guess how CPU burst (job) will behave based on past CPU bursts (jobs) of this process

More MLFQ Rules

Rule 1: If priority(A) > Priority(B), A runs

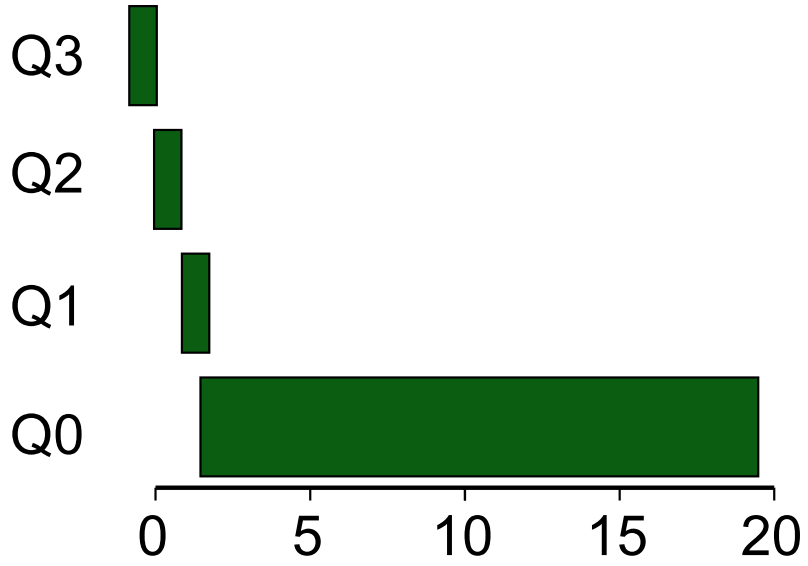
Rule 2: If priority(A) == Priority(B), A & B run in RR

More rules:

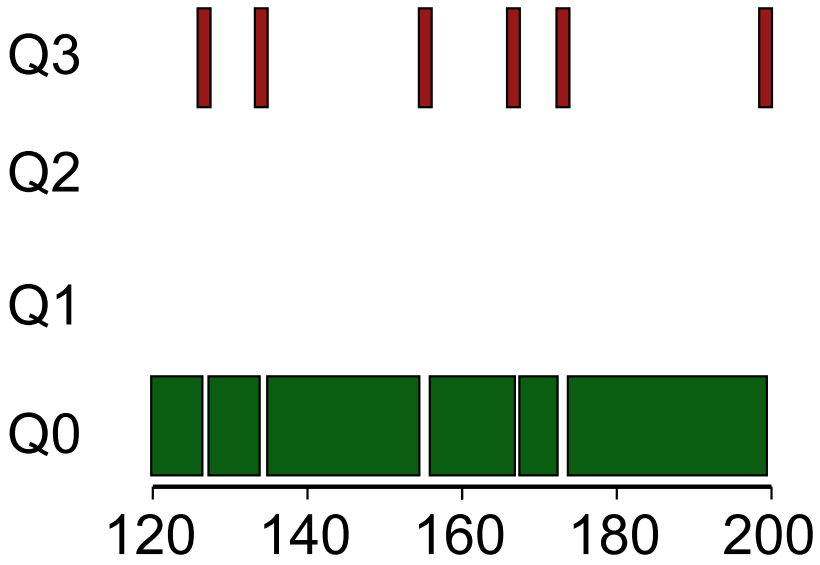
Rule 3: Processes start at top priority

Rule 4: If job uses whole slice, demote process
(longer time slices at lower priorities)

One Long Job Example

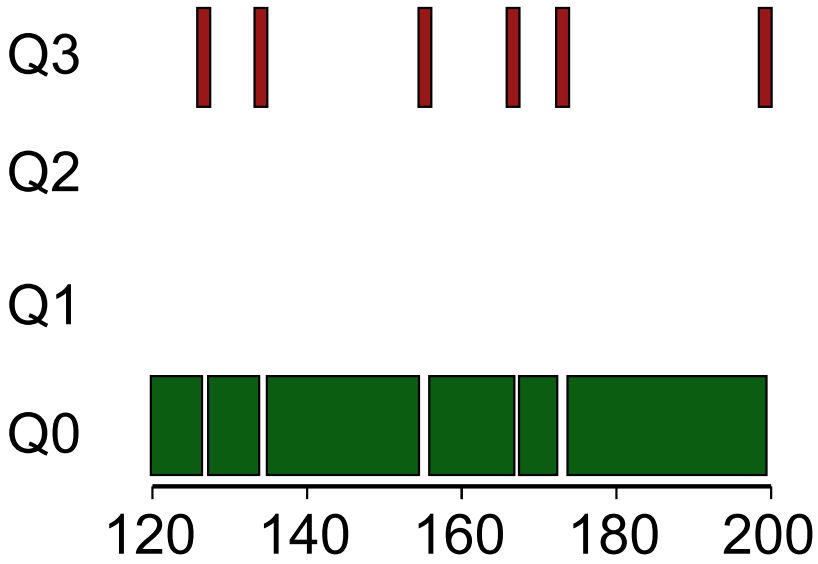


An Interactive Process Joins



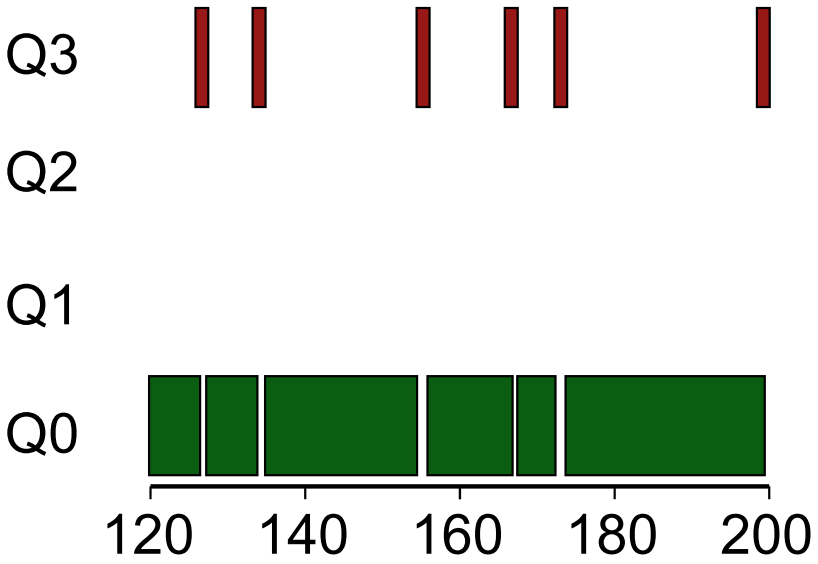
Interactive process never uses entire time slice, so never demoted

Problems with MLFQ?



Problems
- unforgiving + starvation
- gaming the system

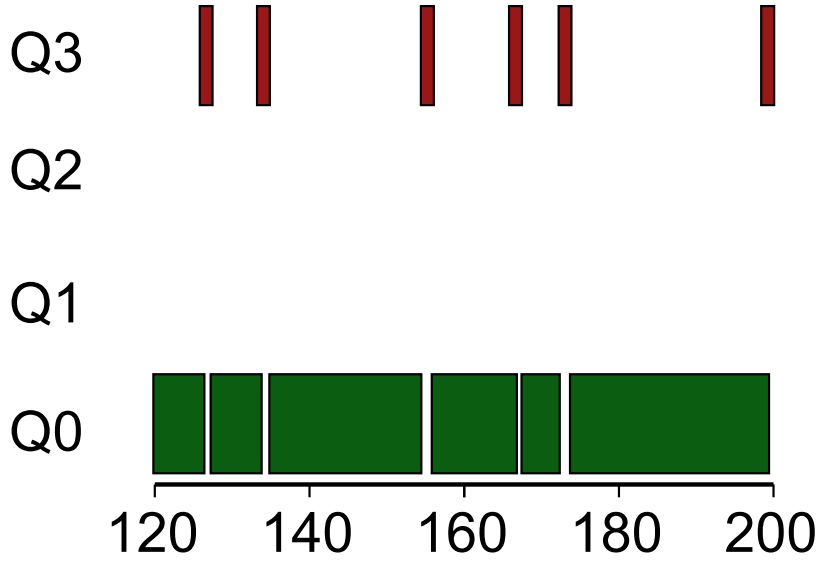
Prevent Starvation



Problem: Low priority job may never get scheduled

Periodically boost priority of all jobs (or all jobs that haven't been scheduled)

Prevent Gaming



Problem: High priority job could trick scheduler and get more CPU by performing I/O right before time-slice ends

Fix: Account for job's total run time at priority level (instead of just this time slice); downgrade when exceeding threshold

Proportional Share Scheduler – Lottery

- Fair-share scheduler
  - Guarantee that each job obtain *a certain percentage* of CPU time.
  - Not optimized for turnaround or response time

Approach:
- give processes lottery tickets
- whoever wins runs
- higher priority => more tickets

Amazingly simple to implement

Basic Concept

- Tickets
  - Represent the share of a resource that a process should receive
  - The percent of tickets represents its share of the system resource in question.

- Example
  - There are two processes, A and B.
    - Process A has 75 tickets → receive 75% of the CPU
    - Process B has 25 tickets → receive 25% of the CPU

Lottery scheduling

- The scheduler picks a winning ticket.
  - Load the state of that *winning process* and runs it.
- Example
  - There are 100 tickets
    - Process A has 75 tickets: 0 ~ 74
    - Process B has 25 tickets: 75 ~ 99

Scheduler's winning tickets:	63	85	70	39	76	17	29	41	36	39	10	99	68	83	63
Resulting scheduler:	A	B	A	A	B	A	A	A	A	A	A	B	A	B	A

The longer these two jobs compete,
The more likely they are to achieve the desired percentages.

Ticket Mechanisms

- Ticket currency
  - A user allocates tickets among their own jobs in whatever currency they would like.
  - The system converts the currency into the correct global value.
  - Example
    - There are 200 tickets (Global currency)
    - Process A has 100 tickets
    - Process B has 100 tickets

User A → *500* (A's currency) to A1 → *50* (global currency)
→ *500* (A's currency) to A2 → *50* (global currency)

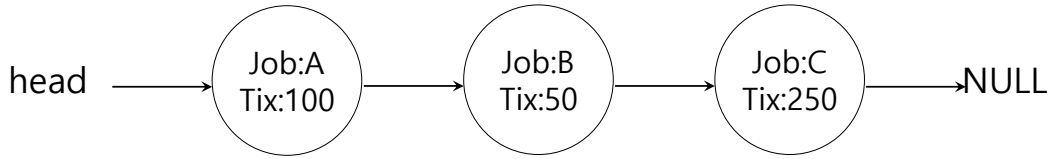
User B → *10* (B's currency) to B1 → *100* (global currency)

Ticket Mechanisms (Cont.)

- Ticket transfer
  - A process can temporarily hand off *its tickets* to another process.
- Ticket inflation
  - A process can temporarily raise or lower the number of tickets is owns.
  - If any one process needs *more CPU time*, it can boost its tickets.

Implementation

- Example: There are there processes, A, B, and C.
  - Keep the processes in a list:



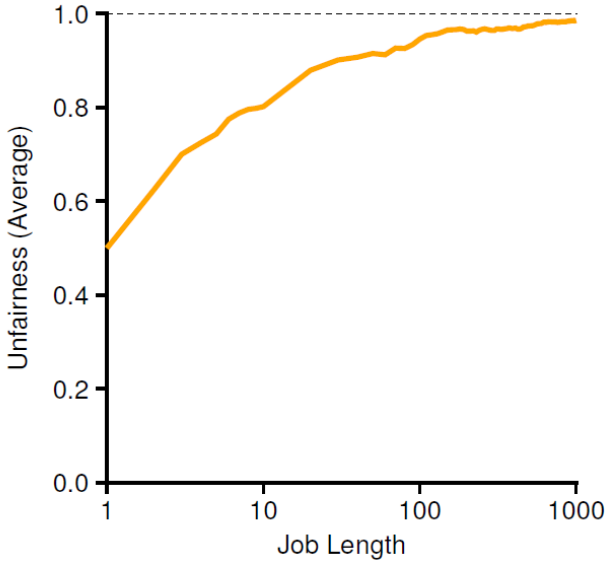

```
1    // counter: used to track if we've found the winner yet
2    int counter = 0;
3
4    // winner: use some call to a random number generator to
5    // get a value, between 0 and the total # of tickets
6    int winner = getrandom(0, totaltickets);
7
8    // current: use this to walk through the list of jobs
9    node_t *current = head;
10
11   // loop until the sum of ticket values is > the winner
12   while (current) {
13        counter = counter + current->tickets;
14        if (counter > winner)
15             break; // found the winner
16        current = current->next;
17   }
18   // 'current' is the winner: schedule it...
```

Implementation (Cont.)

- U: unfairness metric
  - The time the first job completes divided by the time that the second job completes.
- Example:
  - There are two jobs, each jobs has runtime 10.
    - First job finishes at time 10
    - Second job finishes at time 20
  - $U = \frac{10}{20} = 0.5$
  - U will be close to 1 when both jobs finish at nearly the same time.

Lottery Fairness Study

- There are two jobs.
  - Each jobs has the same number of tickets (100).



When the job length is not very long, average unfairness can be quite severe.

Stride Scheduling

- Stride of each process
  - (A large number) / (the number of tickets of the process)
  - Example: A large number = 10,000
    - Process A has 100 tickets → stride of A is 100
    - Process B has 50 tickets → stride of B is 200

- A process runs, increment a counter(=pass value) for it by its stride.
  - Pick the process to run that has the lowest pass value

```
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
```

A pseudo code implementation

Summary

- Understand goals (metrics) and workload, then design scheduler around that
- General purpose schedulers need to support processes with different goals
- Past behavior is good predictor of future behavior
- Random algorithms (lottery scheduling) can be simple to implement and avoid corner cases.

CS 5600 Computer Systems

Spring 2026

Virtualization: Memory
Lecture 3.2
January 27, 2026

Prof. Scott Valcourt

s.valcourt@northeastern.edu

603-380-2860 (cell)

Portland, ME

The Roux Institute
Northeastern University

Lecture Agenda

- Chapter 10 – Multiprocessor Scheduling
- Chapter 13 – Abstraction: Address Space
- Chapter 14 – Memory API

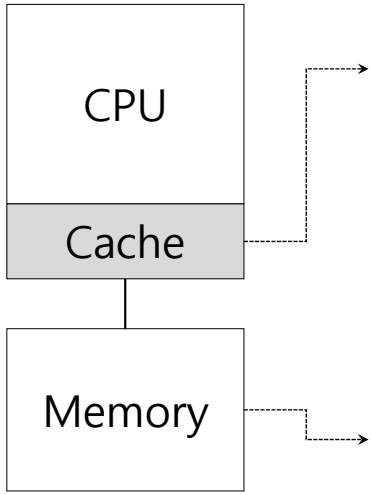
Multiprocessor Scheduling

- The rise of the multicore processor is the source of multiprocessor-scheduling proliferation.
  - **Multicore**: Multiple CPU cores are packed onto a single chip.

- Adding more CPUs does not make that single application run faster. → You'll have to rewrite application to run in parallel, using **threads**.

How to schedule jobs on Multiple CPUs?

Single CPU with cache



Cache

Main Memory

- Small, fast memories
- Hold copies of popular data that is found in the main memory.
- Utilize *temporal* and *spatial* locality

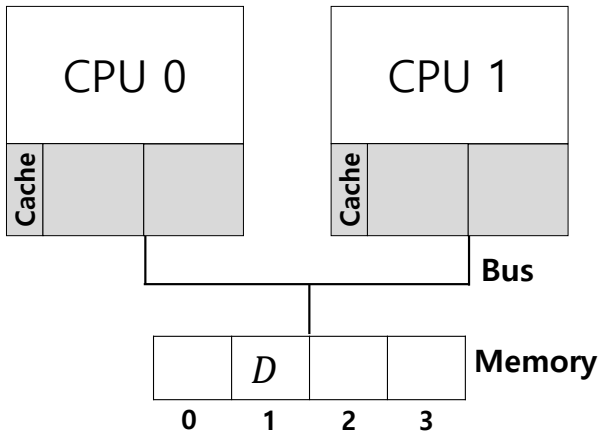
- Holds all the data
- Access to main memory is slower than cache.

By keeping data in cache, the system can make slow memory appear to be a fast one

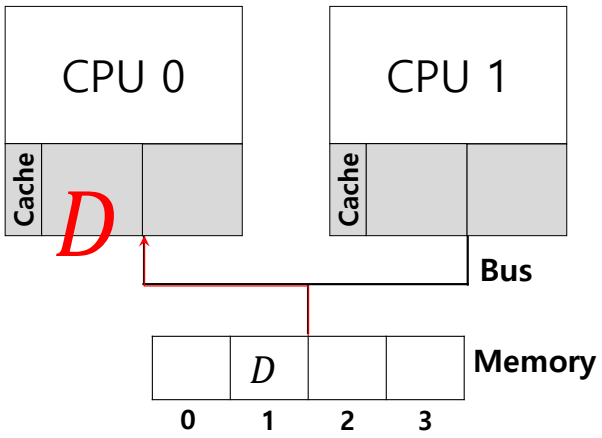
Cache Coherence

- Consistency of shared resource data stored in multiple caches.

0. Two CPUs with caches sharing memory

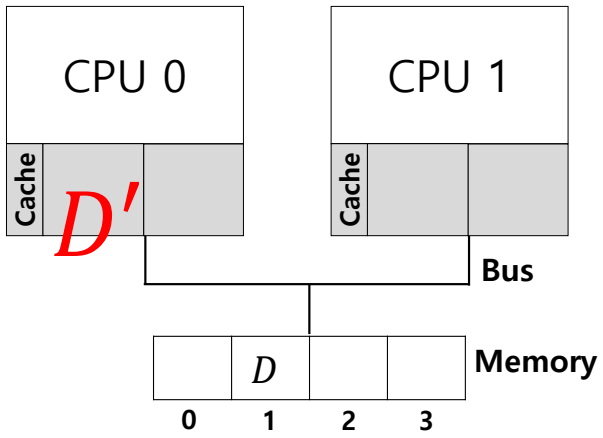


1. CPU0 reads a data at address 1.

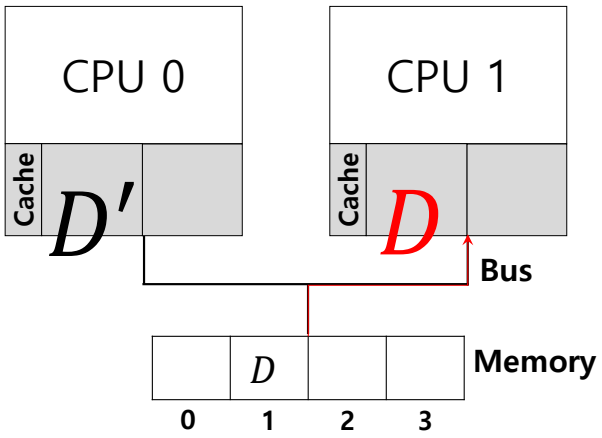


Cache Coherence (Cont.)

2. D is updated and CPU1 is scheduled.



3. CPU1 re-reads the value at address A



CPU1 gets the old value D instead of the correct value D'.

Cache Coherence Solution

- Bus snooping
  - Each cache pays attention to memory updates by **observing the bus**.
  - When a CPU sees an update for a data item it holds in its cache, it will notice the change and either invalidate its copy or update it.

Don’t Forget Synchronization

- When accessing shared data across CPUs, mutual exclusion primitives should likely be used to guarantee correctness.

```
1    typedef struct __Node_t {
2         int value;
3         struct __Node_t *next;
4    } Node_t;
5
6    int List_Pop() {
7         Node_t *tmp = head; // remember old head ...
8         int value = head->value; // ... and its value
9         head = head->next;       // advance head to next pointer
10        free(tmp);           // free old head
11        return value;        // return value at head
12   }
```

Simple List Delete Code

Don’t Forget Synchronization (Cont.)

- Solution

```
1    pthread_mutex_t m;
2    typedef struct __Node_t {
3         int value;
4         struct __Node_t *next;
5    } Node_t;
6
7    int List_Pop() {
8         lock(&m)
9         Node_t *tmp = head; // remember old head ...
10        int value = head->value; // ... and its value
11        head = head->next;       // advance head to next pointer
12        free(tmp);           // free old head
13        unlock(&m)
14        return value;        // return value at head
15   }
```

Simple List Delete Code with lock

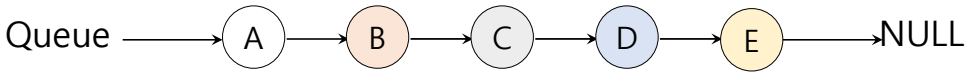
Cache Affinity

- Keep a process on the same CPU if at all possible
  - A process builds up a fair bit of state in the cache of a CPU.
  - The next time the process run, it will run faster if some of its state is *already present* in the cache on that CPU.

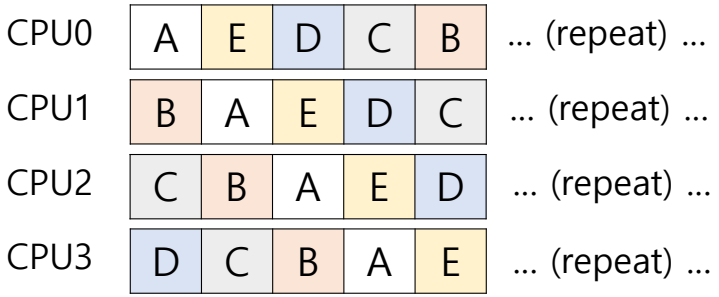
A multiprocessor scheduler should consider cache affinity when making its scheduling decision.

Single Queue Multiprocessor Scheduling (SQMS)

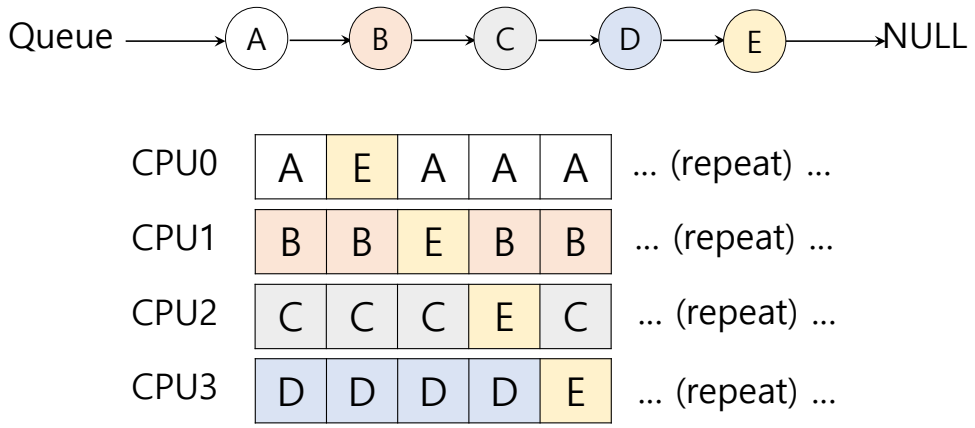
- Put all jobs that need to be scheduled into a single queue.
  - Each CPU simply picks the next job from the globally shared queue.
  - Cons:
    - Some form of **locking** has to be inserted → Lack of scalability
    - Cache affinity
    - Example:



    - Possible job scheduler across CPUs:



Scheduling Example with Cache affinity



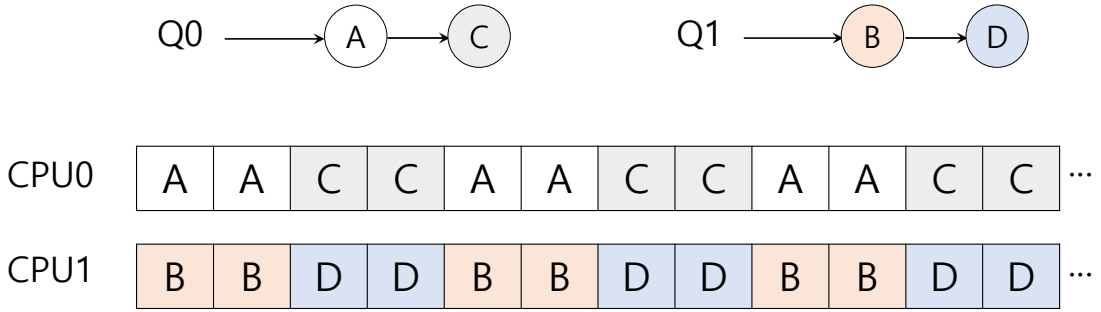
- Preserving affinity for most
  - Jobs A through D are not moved across processors.
  - Only job E Migrating from CPU to CPU.
- Implementing such a scheme can be **complex**.

Multi-queue Multiprocessor Scheduling (MQMS)

- MQMS consists of multiple scheduling queues.
  - Each queue will follow a particular scheduling discipline.
  - When a job enters the system, it is placed on **exactly one** scheduling queue.
  - Avoid the problems of information sharing and synchronization.

MQMS Example

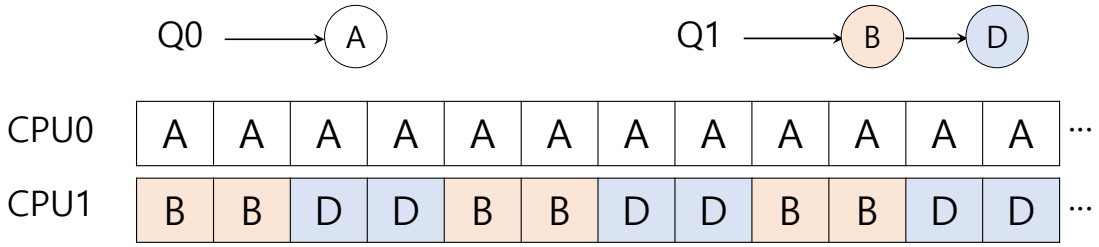
- With **round robin**, the system might produce a schedule that looks like this:



MQMS provides more scalability and cache affinity.

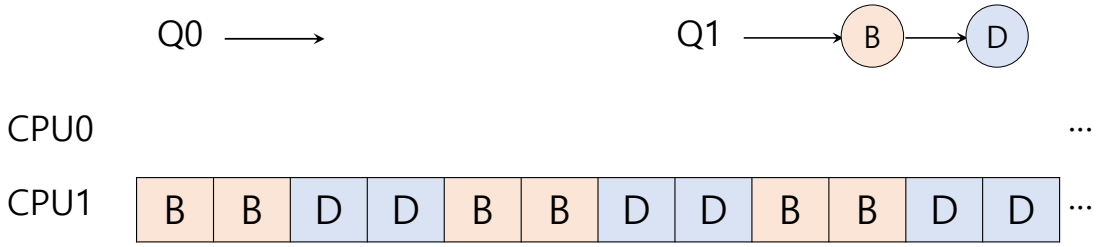
Load Imbalance Issue of MQMS

- After job C in Q0 finishes:



A gets twice as much CPU as B and D.

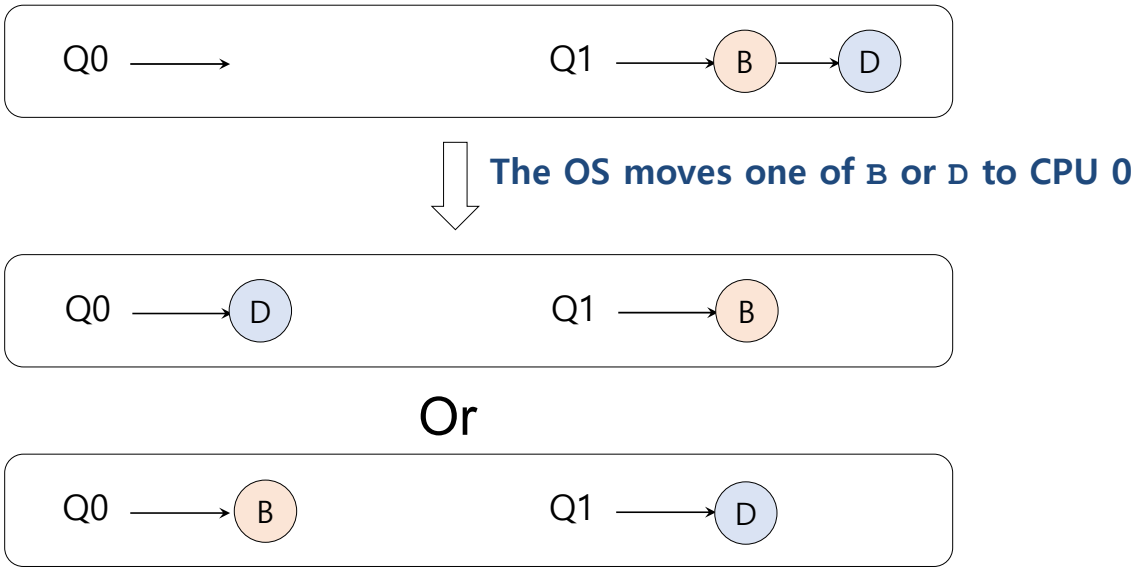
- After job A in Q0 finishes:



CPU0 will be left idle!

How to Deal With Load Imbalance?

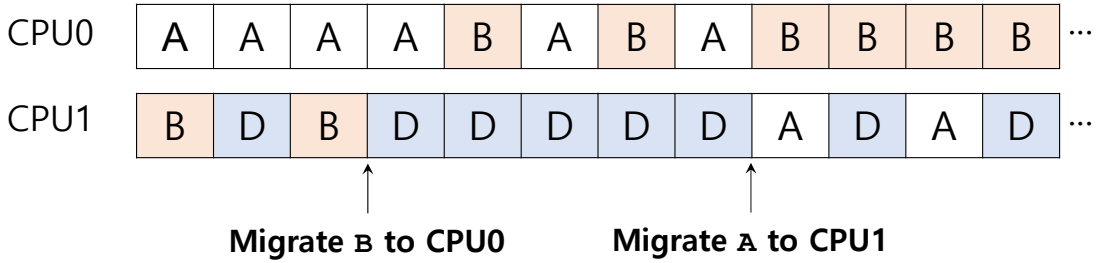
- The answer is to move jobs (**Migration**).
  - Example:



How to Deal With Load Imbalance? (Cont.)

- A tricky case:

- A possible migration pattern:
  - Keep switching jobs



Work Stealing

- Move jobs between queues
  - Implementation:
    - A source queue that is low on jobs is picked.
    - The source queue occasionally peeks at another target queue.
    - If the target queue is "more full" than the source queue, the source will "**steal**" one or more jobs from the target queue.
  - Cons:
    - *High overhead* and trouble *scaling*

Linux Multiprocessor Schedulers

- O(1)
  - A Priority-based scheduler
  - Use Multiple queues
  - Change a process's priority over time
  - Schedule those with highest priority
  - Interactivity is a particular focus

- Completely Fair Scheduler (CFS)
  - Deterministic proportional-share approach
  - Multiple queues

Linux Multiprocessor Schedulers

- Best Fit Scheduler (BFS)
  - A single queue approach
  - Proportional-share
  - Based on Earliest Eligible Virtual Deadline First(EEVDF)

In-Class Activity

- Q1: A FIFO scheduler receives three jobs, each with arrival_time of 0s and run_time of 10s. What is the average turnaround time?
- Q2: A FIFO scheduler receives three jobs, each with arrival_time of 0s, but the first job has a run_time of 50s and the other two jobs have a run_time of 20s. What is the average turnaround time?
- Q3: A SJF scheduler receives three jobs, each with arrival_time of 0s, but the first job has a run_time of 50s and the other two jobs have a run_time of 20s. What is the average turnaround time?
- Q4: A, a 40s job, arrives at time 0s. B and C, each 20s jobs, arrive at time 10s. For STCF, what is turnaround time?
- Q5: A, B, and C all arrive at 0s, and each has a run time of 10s. What is the average response time for FIFO? What is the average response time for RR (1s slice)?

Break

Memory Virtualization

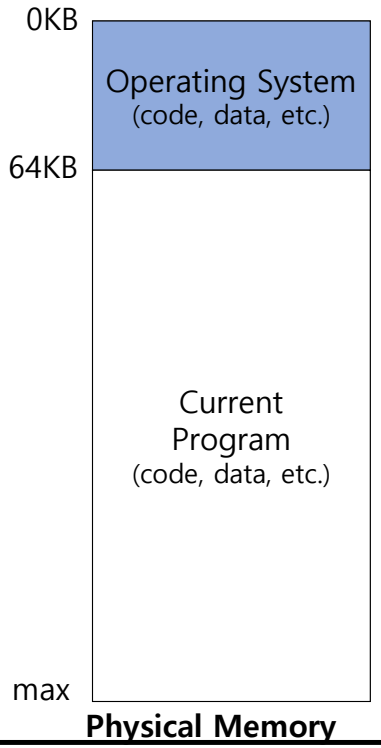
- What is **memory virtualization**?
  - OS virtualizes its physical memory.
  - OS provides an illusion memory space per each process.
  - It seems like each process uses the whole memory.

Benefits of Memory Virtualization

- Ease of use in programming
- Memory efficiency in terms of time and space
- The guarantee of isolation for processes as well as OS
  - Protection from errant accesses of other processes

OS in the Early System Years

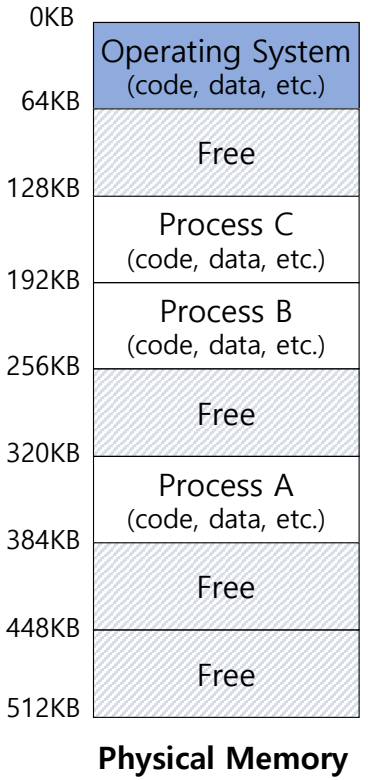
- Load only one process in memory.
  - Poor utilization and efficiency



Multiprogramming and Time Sharing

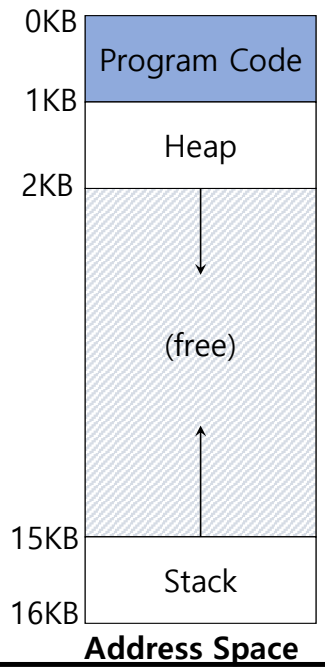
- **Load multiple processes** in memory.
  - Execute one for a short while.
  - Switch processes between them in memory.
  - Increase utilization and efficiency.

- Causes an important **protection issue**.
  - Errant memory accesses from other processes



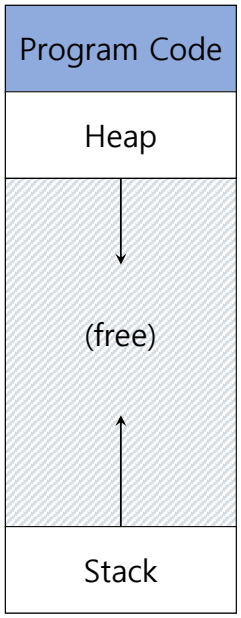
Address Space

- OS creates an **abstraction** of physical memory.
  - Address space contains everything about a running process.
  - That is program code, heap, stack, etc.



Address Space (Cont.)

- Code
  - Where instructions live
- Heap
  - Dynamically allocate memory.
    - `malloc` in C language
    - `new` in object-oriented language
- Stack
  - Store return addresses or values.
  - Contain local variables and arguments to routines.



Address Space

Virtual Address

- **Every address** in a running program is virtual.
  - OS translates the virtual address to physical address

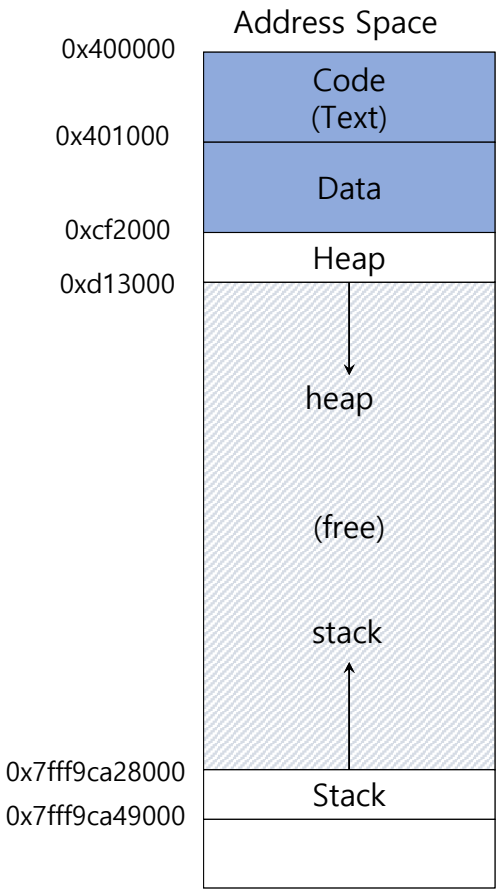
```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]){

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

    return x;
}
```

A simple program that prints out addresses

Virtual Address (Cont.)

- The output in 64-bit Linux machine

```
location of code  : 0x40057d
location of heap  : 0xcf2010
location of stack : 0x7fff9ca45fcc
```



Memory API: `malloc()`

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap.
  - Argument
    - `size_t size` : size of the memory block(in bytes)
    - `size_t` is an unsigned integer type.
  - Return
    - Success: a void type pointer to the memory block allocated by `malloc`
    - Fail: a `NULL` pointer

Memory API: `sizeof()`

- Routines and macros are utilized for `size` in `malloc` instead typing in a number directly.
- Two types of results of `sizeof` with variables
  - The actual size of `'x'` is known at run-time.

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

  - The actual size of `'x'` is known at compile-time.

```
int x[10];
printf("%d\n", sizeof(x));
```

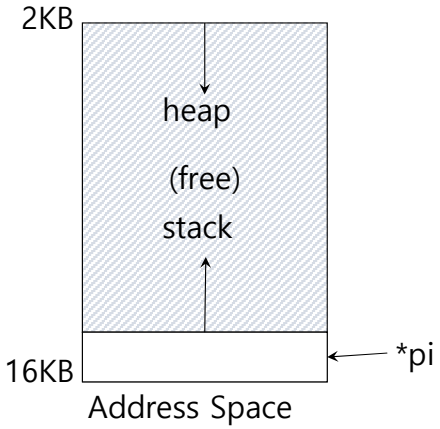
```
40
```

Memory API: `free()`

```
#include <stdlib.h>

void free(void* ptr)
```

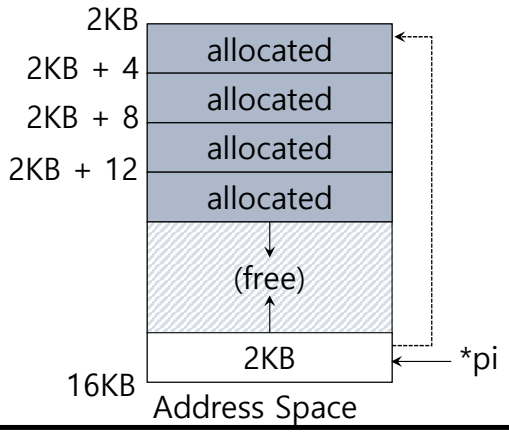
- Free a memory region allocated by a call to `malloc`.
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Return
    - none

Memory Allocating



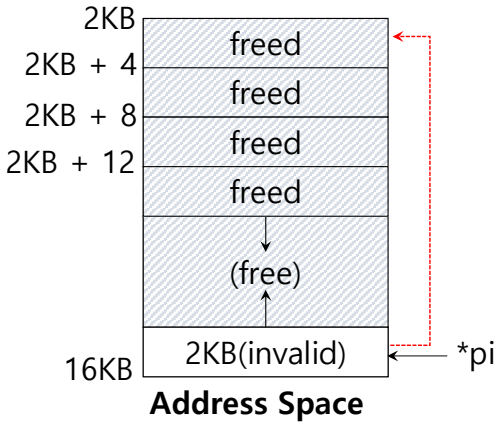
pointer

```
int *pi; // local variable
```

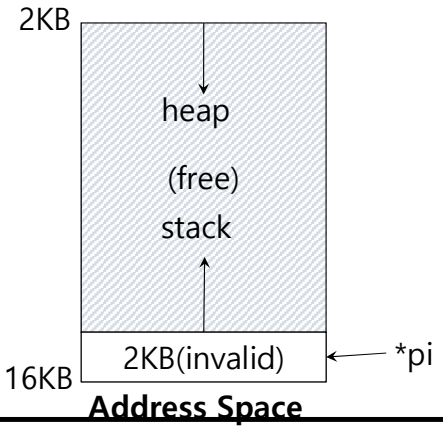



```
pi = (int *)malloc(sizeof(int)* 4);
```

Memory Freeing



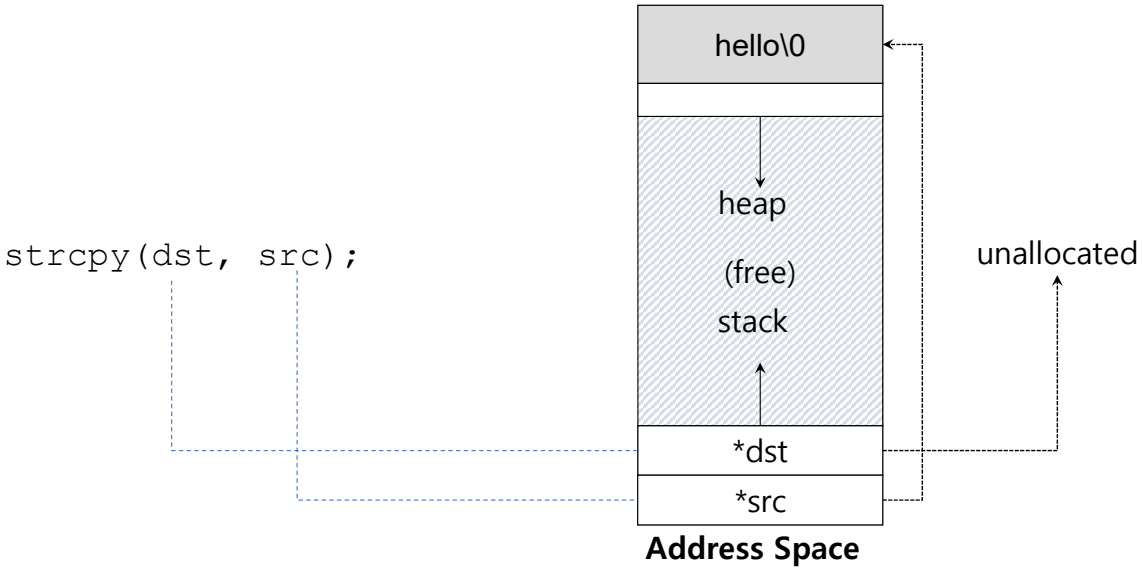

```
free(pi);
```



Forgetting To Allocate Memory

- Incorrect code

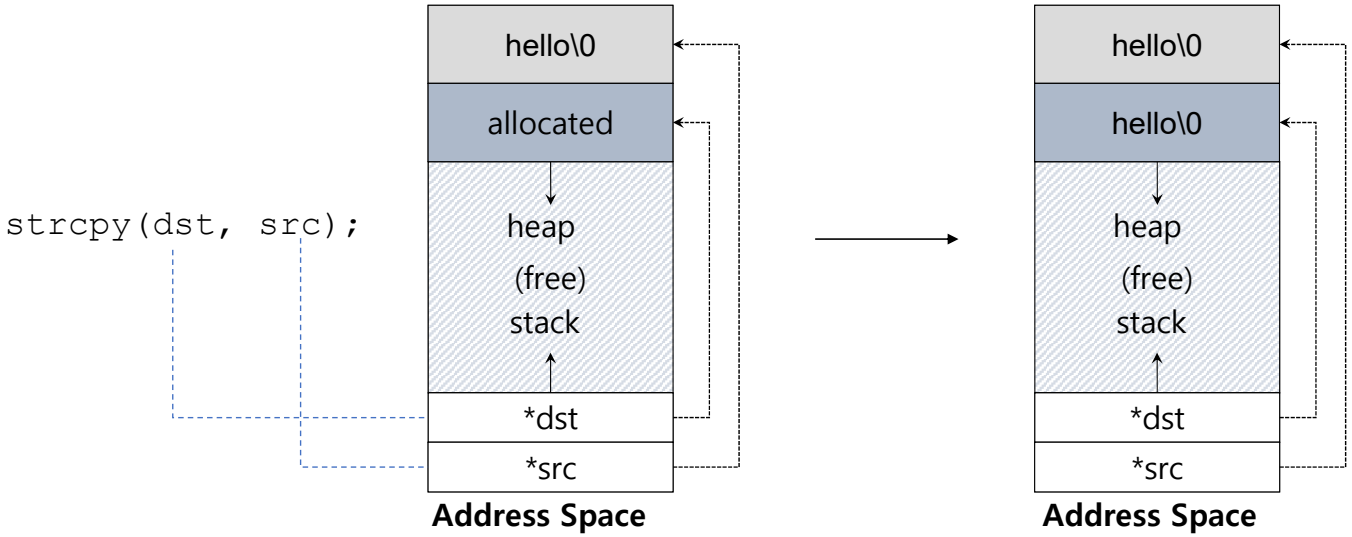
```
char *src = "hello"; //character string constant
char *dst;           //unallocated
strcpy(dst, src);    //segfault and die
```



Forgetting To Allocate Memory(Cont.)

- Correct code

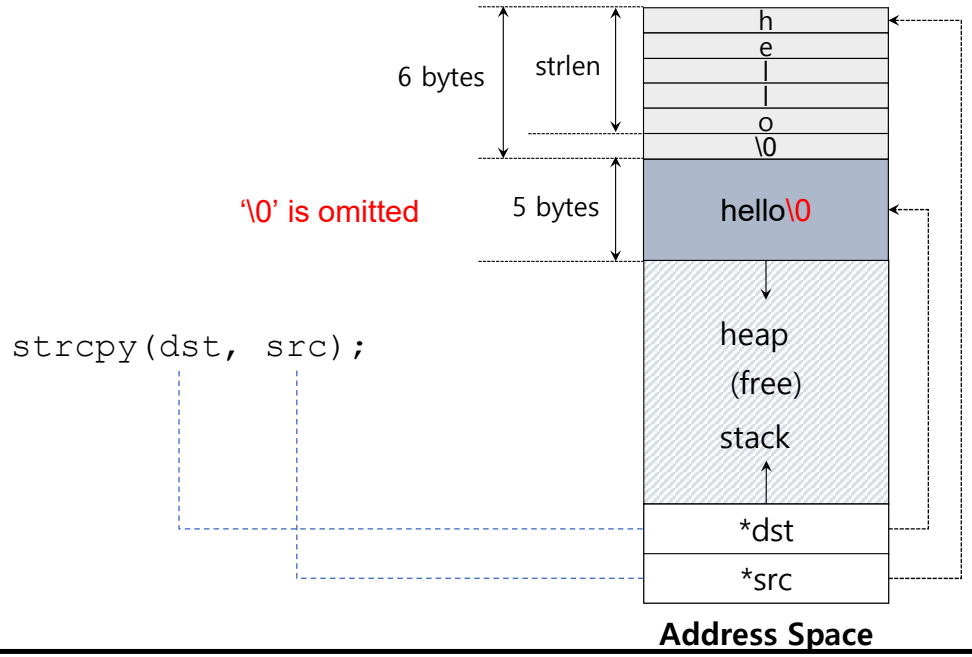
```
char *src = "hello";   //character string constant
char *dst (char *)malloc(strlen(src) + 1 ); // allocated
strcpy(dst, src);      //work properly
```



Not Allocating Enough Memory

- Incorrect code, but works properly

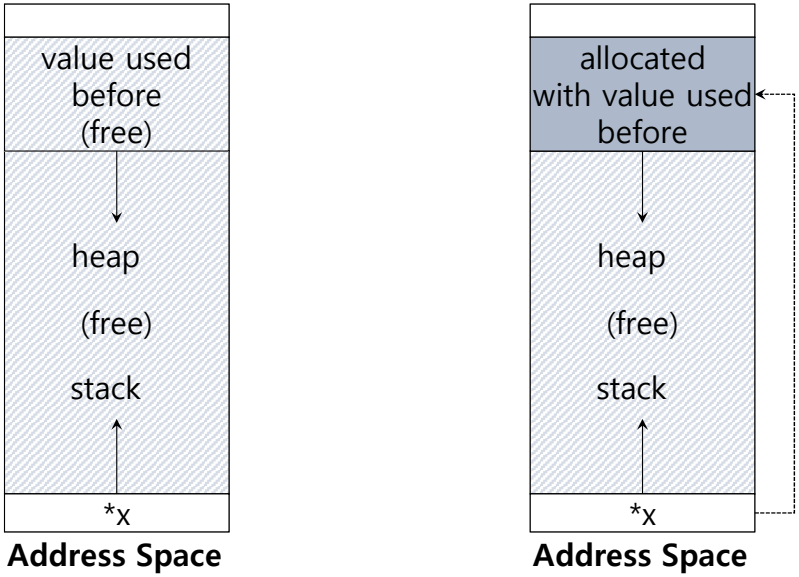
```
char *src = "hello"; //character string constant
char *dst (char *)malloc(strlen(src)); // too small
strcpy(dst, src);    //work properly
```



Forgetting to Initialize

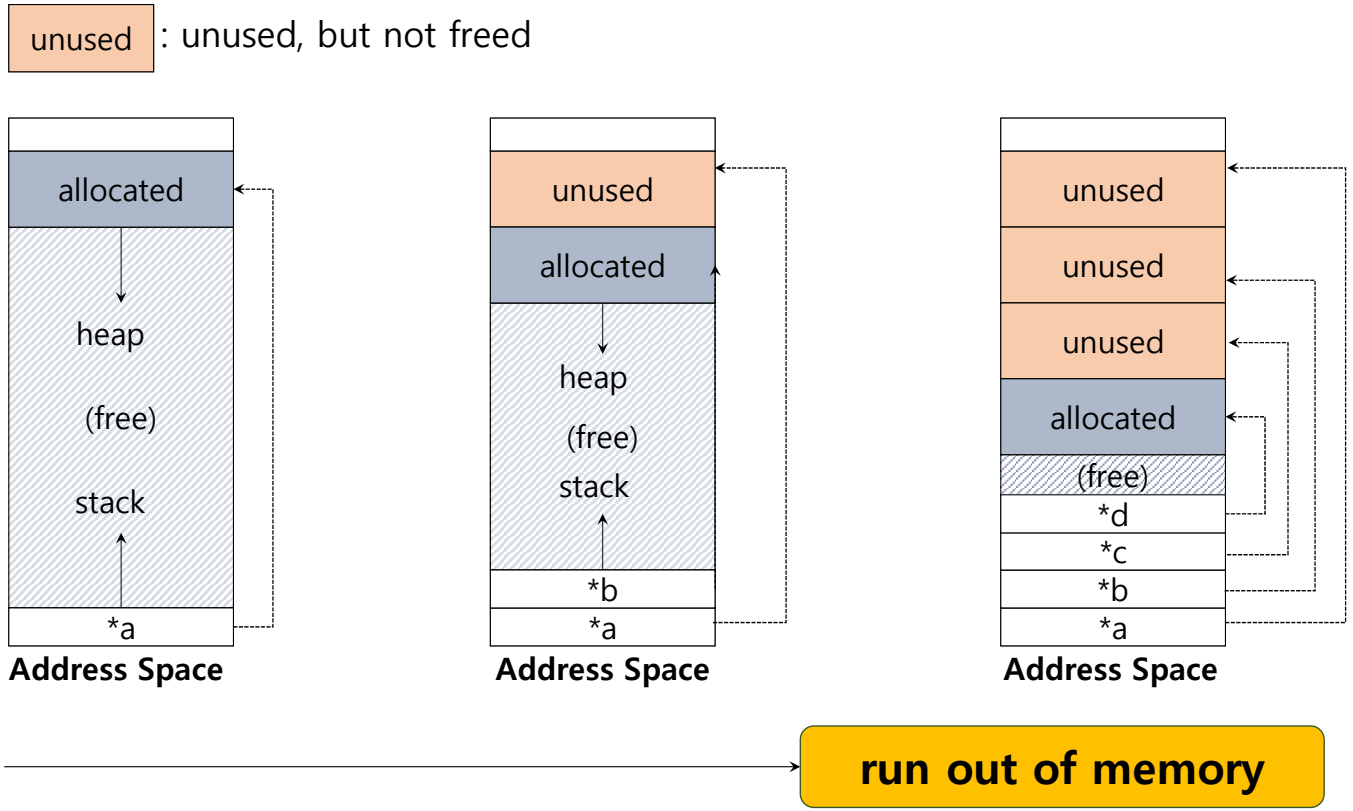
- Encounter an uninitialized read

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x);             // uninitialized memory access
```



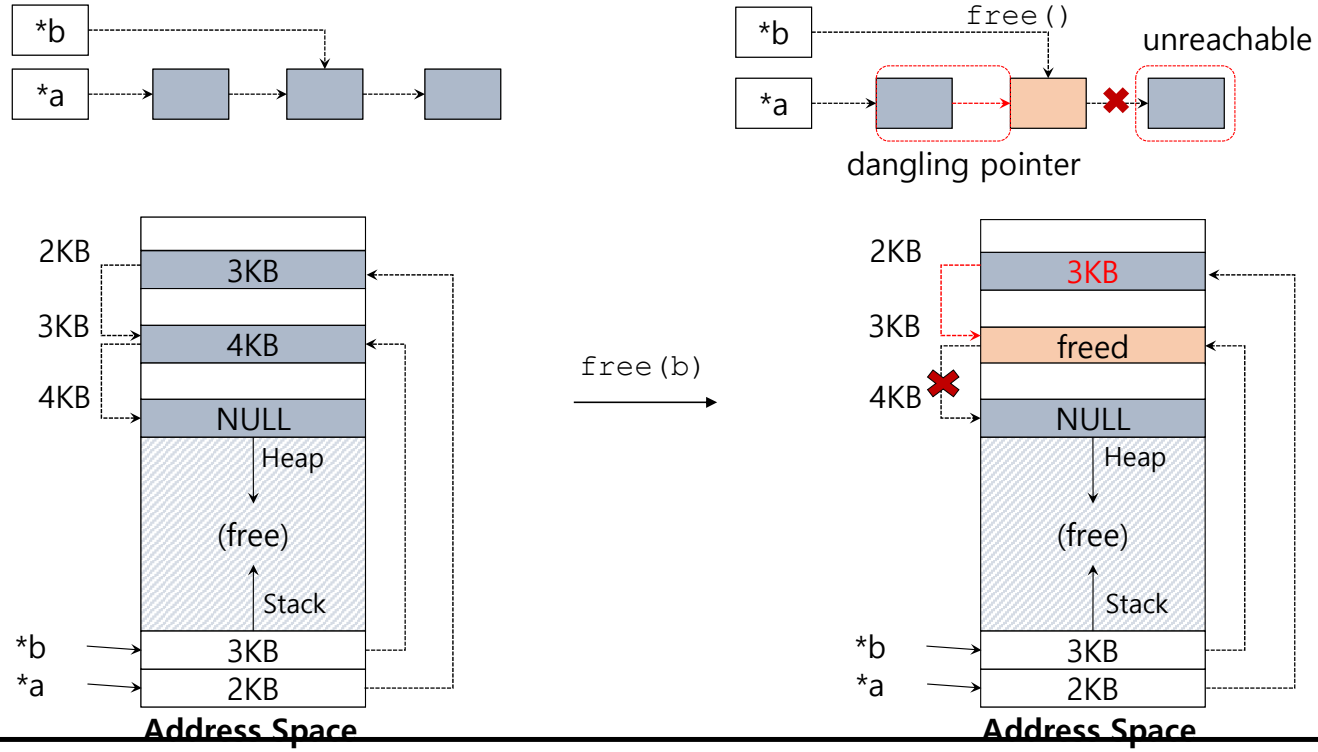
Memory Leak

- A program runs out of memory and eventually dies.



Dangling Pointer

- Freeing memory before it is finished using
  - A program accesses memory with an invalid pointer



Other Memory APIs: `calloc()`

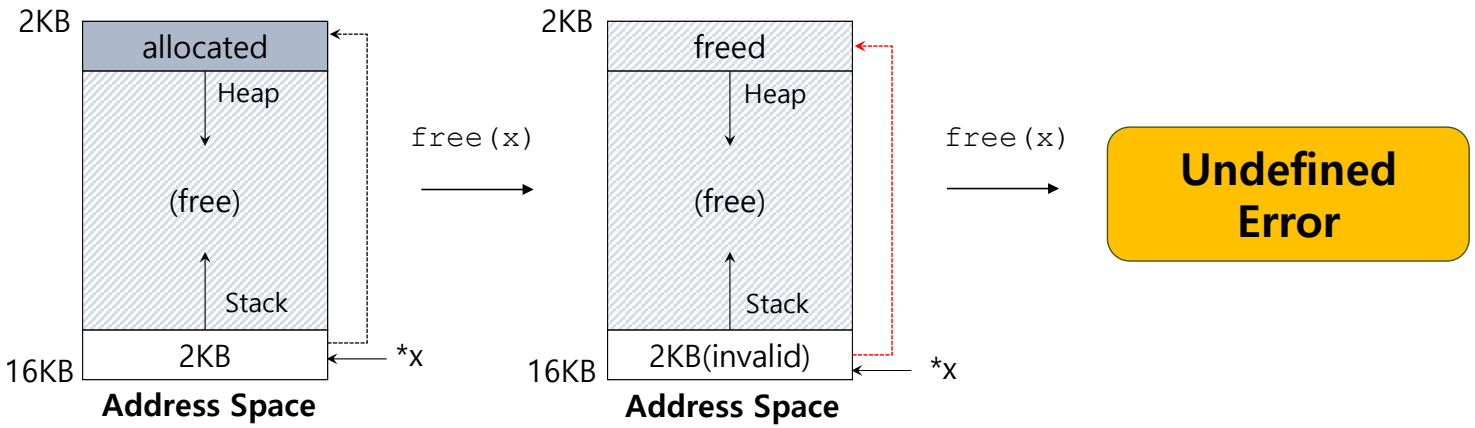
```
#include <stdlib.h>

void *calloc(size_t num, size_t size)
```

- Allocate memory on the heap and zeroes it before returning.
  - Argument
    - `size_t num` : number of blocks to allocate
    - `size_t size` : size of each block(in bytes)
  - Return
    - Success: a void type pointer to the memory block allocated by `calloc`
    - Fail: a `NULL` pointer

Double Free

- Free memory that was freed already.

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x);                             // free memory
free(x);                             // free repeatedly
```



Other Memory APIs: `realloc()`

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Change the size of memory block.
  - A pointer returned by `realloc` may be either the same as `ptr` or a new one.
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc, calloc,` or `realloc`
    - `size_t size`: New size for the memory block (in bytes)
  - Return
    - Success: Void type pointer to the memory block
    - Fail: a `NULL` pointer

System Calls

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

- `malloc` library call use `brk` system call.
  - `brk` is called to expand the program's *break*.
    - *break*: The location of **the end of the heap** in address space
  - `sbrk` is an additional call similar to `brk`.
  - Programmers **should never directly call** either `brk` or `sbrk`.

System Calls (Cont.)

```
#include <sys/mman.h>

void *mmap(void *ptr, size_t length, int port, int flags,
int fd, off_t offset)
```

- `mmap` system call can create **an anonymous** memory region.

Quiz Time! Match that Address Location

```
int x;
int main(int argc, char *argv[]) {
  int y;
  int *z = malloc(sizeof(int)););
}
```

Possible segments: static data, code, stack, heap

Address	Location
x	Static data → Code
main	Code
y	Stack
z	Stack
*z	Heap

What if no static data segment exists?

What's Next

- Module Software Preparation (HW0) (Please let me know you are set)
  - Due by Wednesday, January 17 by 5:59pm ET
  - Install Virtual Machine (VirtualBox or UTM or something else)
  - Install GitHub repository
  - Verify C Compiler works
  - Push file into GitHub private repo

- Programming Assignment 1
  - Due by Monday, February 5 by 11:59pm ET
  - Write a UNIX Shell

- Homework Assignment 1
  - Due by Sunday, January 28 by 11:59pm ET

- Quiz 1
  - Due by Wednesday, January 24 by 11:59pm ET

- Homework Assignment 2
  - Due by Sunday, February 4 by 11:59pm ET

- Quiz 1
  - Due by Wednesday, January 31 by 11:59pm ET

What's Next

- Programming Assignment 1
  - Due by Sunday, February 1 by 11:59pm ET
  - Write a UNIX Shell
- Homework Assignment 1
  - Due by Sunday, February 1 by 11:59pm ET
- Quiz 1
  - Due by Sunday, February 1 by 11:59pm ET
- Homework Assignment 2
  - Due by Sunday, February 8 by 11:59pm ET
- Quiz 2
  - Due by Sunday, February 8 by 11:59pm ET