CSC369 Lecture 4

Larry Zhang

Announcements

- ➔ No lecture next Monday (October 12)
- ➔ Yes tutorial next Thursday
- ➔ Office hour this Wednesday temporarily changed to
  - ◆ 2-4pm, or open-door

Questions about Linux kernel

Is Linux kernel a process?

➔ No, “process” is a user-space thing
➔ Kernel is just a static pool of code, sitting there, waiting to be invoked (by syscalls from user space), i.e., kernel is completely **event-driven**.

Is Linux kernel multithreaded?

➔ Yes, all kernel threads share the same address space.
➔ Different threads share stuff (what stuff?) therefore synchronization is needed in Assignment 1.

Tips on A1

Want use the `syscall(syscall_no, arg1, arg2, ...)` to make the original system call?

No can do, this is a user-space function therefore cannot be used in the kernel.

You need to remember the original syscall's function pointer in the global data structure then use it later to call it.

Learn about function pointer: http://www.cprogramming.com/tutorial/function-pointers.html

More A1 tips

- ➔ Know clearly what helper functions are provided in the starter code, and use them. Don’t reinvent the wheel.
- ➔ Expected number of new lines of code to be added:
  - ◆ ~300 lines
- ➔ Do NOT modify header file or Makefile

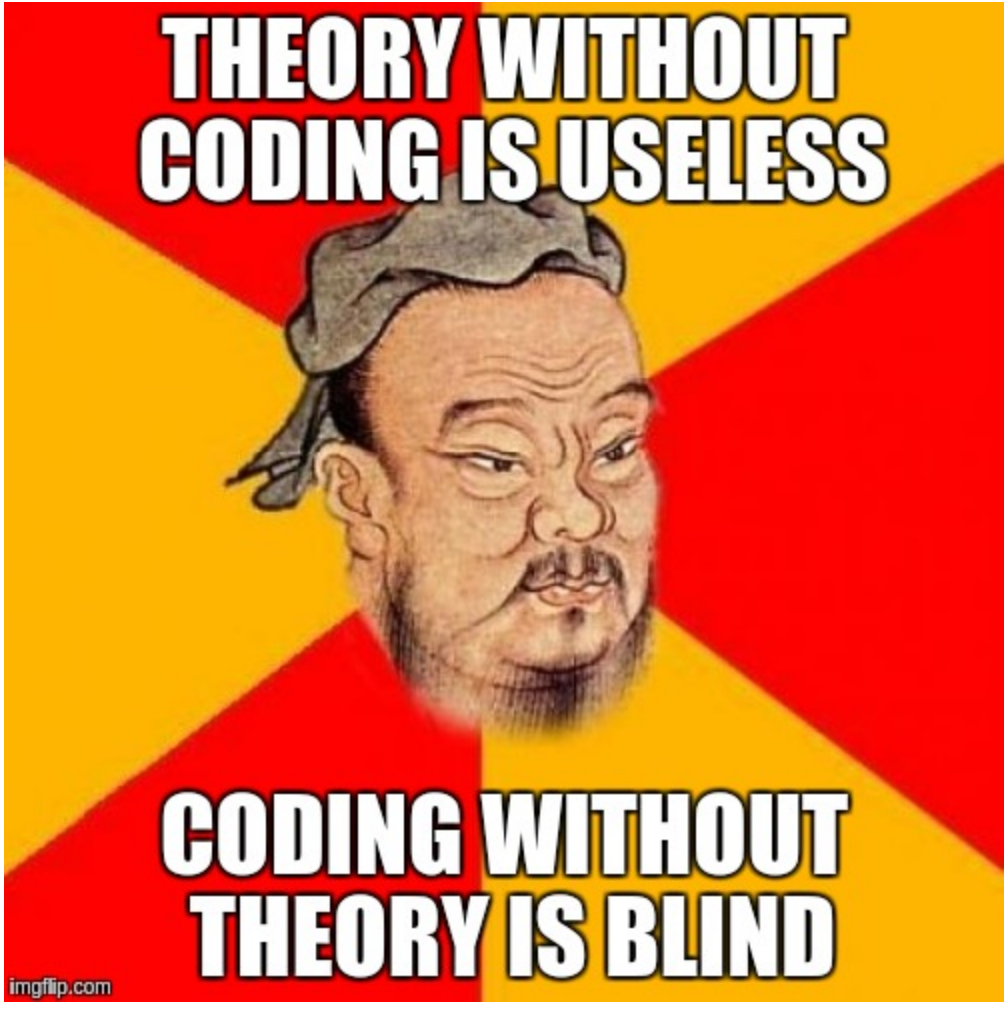
From feedback form http://goo.gl/forms/rSHh0Y2uHL

"The high level concepts about kernels seem straightforward, but the specific implementation in C is ~~really hard.~~"

more detail-involved.

True, and this is designed to be, we will learn from assignments hands-on things that we don't learn from lectures and tutorials.

Theoretical knowledge + practical skills are the two things both of which we want to learn in this course.


THEORY WITHOUT CODING IS USELESS
CODING WITHOUT THEORY IS BLIND
imgflip.com

Recap: Last week

- **Threads**
  - multiple control flows in one process
  - faster to create, faster to switch between
  - leads to **concurrency** issues, need **synchronization** mechanism
- **Synchronization**
  - need to be careful whenever multiple threads share resource
  - **Critical Section Problem**: a systematic model for synchronization problems
    - Use **4 criteria** to check whether a synchronization mechanism is good: mutual exclusion, progress, no starvation, performance
- **High level abstractions**
  - Semaphores, Locks, Condition variables

Clarification about Semaphore

```
sem_wait:
value = value - 1
if value < 0
    wait (add to wait queue)

sem_post:
value = value + 1
if one or threads waiting:
    wake up one thread
```

Use this one!

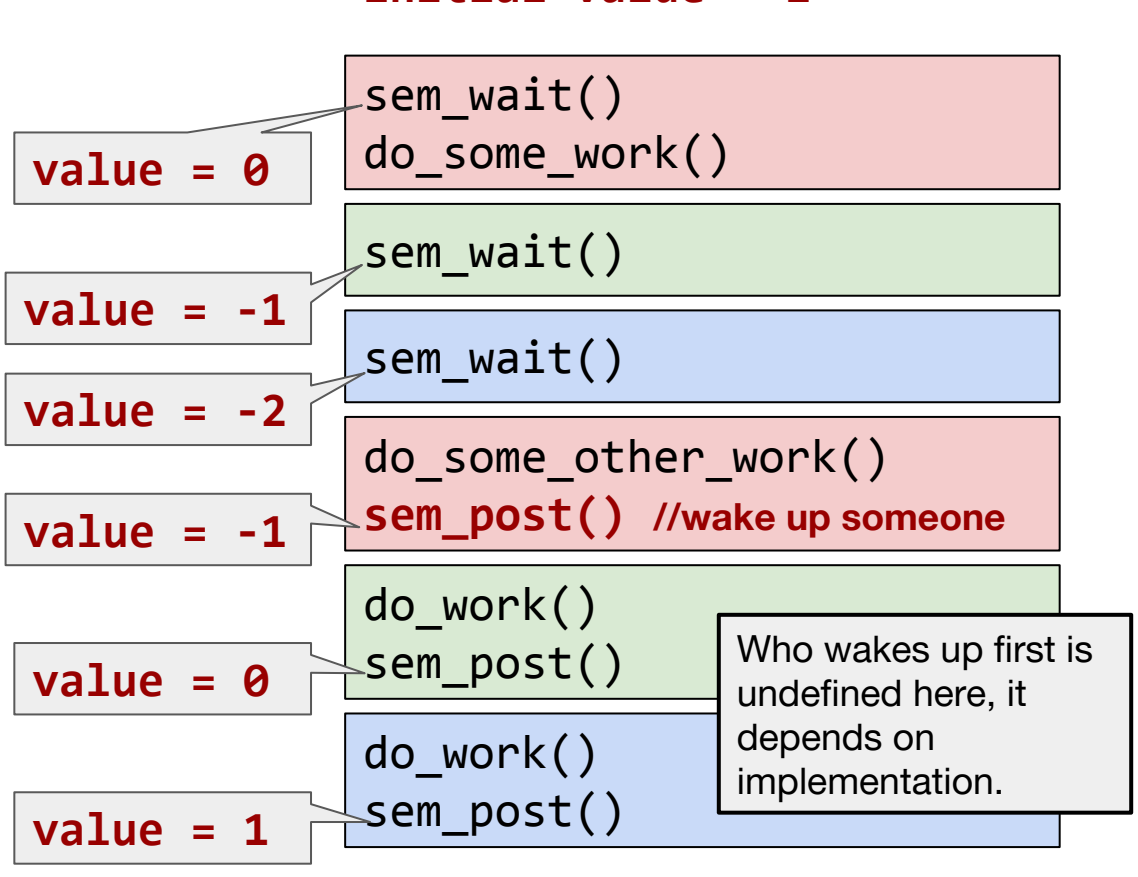
VS

```
sem_wait:
if value <= 0:
    wait
else:
    value = value - 1

sem_post:
value = value + 1
if one or threads waiting:
    wake up one thread
```

More about Semaphore

```
sem_wait:
value = value - 1
if value < 0
    wait (add to wait queue)

sem_post:
value = value + 1
if one or threads waiting:
    wake up one thread
```

wake up a thread even when value is negative



Lock (pthread_mutex_t)

Function Definitions

```
Withdraw(acct, amt) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}
```

```
Deposit(account, amount) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}
```

Possible schedule

```
acquire(lock);
balance = get_balance(acct);
balance = balance - amt;
```

```
acquire(lock);
```

```
put_balance(acct, balance);
release(lock);
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct, balance);
release(lock);
```

Lock *vs* Semaphore

- ➔ A binary semaphore (with initial value 1) can be just used as a lock
- ➔ But they are semantically different
  - ◆ Logically, lock has an owner, a lock can only be release by who acquired it.
  - ◆ Semaphore doesn’t have the concept of owner.
  - ◆ Lock is easier for reasoning about synchronization than semaphore is.

Condition Variable

another high level abstraction for synchronization

Condition Variable (`pthread_cond_t`)

- ➔ Behaviour
  - ◆ sleep and wait for a condition to become true
- ➔ It is a queue of waiting threads
  - ◆ wait in the queue when the condition is not satisfied
  - ◆ Dequeue when woken up
- ➔ Always use CV together with a lock
  - ◆ use lock to protect the CV

Condition Variable Routines

```
mutex_t m;  // the lock
cond_t c;   // the CV

lock(&m);     // lock before using CV

cv_wait(&c, &m);
// release lock m, sleep (add to queue) until woken up
// will re-acquire the lock when woken up

cv_signal(&c);     // wake a thread in the queue
cv_broadcast(&c); // wake all threads in the queue
```

Example: parent wait for child to finish

```
parent:
    print “parent begins”
    thread_t t;
    thread_create(&t, child);
    // wait for child to finish
    print “parent ends”
    return
```

```
child:
    print “child”
    return
```

```
// Expected output:
parent begins
child
parent ends
```

Solution #1: Use a ***done*** variable and some spinning

```
global done = 0
```

```
parent:
    print "parent begins"
    thread_t t;
    thread_create(&t, child);
    while done == 0:
        pass //spin
    print "parent ends"
    return
```

```
child:
    print "child"
    done = 1
    return
```

But busy-waiting (spinning) is not desirable.

Solution #2: Use condition variable and lock

```
global done = 0, mutex_t m, cond_t c
```

```
parent:
    print "parent begins"
    thread_t t;
    thread_create(&t, child);
    lock(&m)
    while done == 0:
        cv_wait(&c, &m);
    unlock(&m)
    print "parent ends"
    return
```

```
child:
    print "child"
    lock(&m)
    done = 1
    cv_signal(&c)
    unlock(&m)
    return
```

unlock, sleep, lock when woken up

parent acquire;
check done;
release lock;
sleep;
child acquire;
set done = 1;
wake up parent
release lock;
parent wake up
and acquire lock;
check done and
exit loop;
release lock;

the while condition is checked only **twice**, i.e., NO busy waiting

Lock & Condition Variables: Design Patterns

1. Always **acquire the lock** before accessing the CV
2. Always **release the lock** before return
3. Call `cv_wait()` only in **while** loops
4. Whenever one of the condition being waited on ***might*** have changed from FALSE to TRUE, call `cv_signal()` on the corresponding CV, so that while loop around the `cv_wait()` can double-check if the condition is really satisfied.

Lock, Condition Variable and Semaphore

- ➔ Lock: used to provide **mutual exclusion**
- ➔ Condition variable: use to **avoid busy waiting**, build on top of mutual exclusion (use together with a lock)
- ➔ Semaphore: has flavours of both lock and CV; can be built using locks and condition variables; can also use semaphore to implement lock or condition variable (but trickier).

The Producer / Consumer Problem

a classic synchronization problem

Producer / Consumer, bounded buffer

```
int buffer[369]
```

```
Producer:
    while true:
        write(buffer)
```

```
Consumer:
    while true:
        read(buffer)
```

- ➔ First posed by Dijkstra
- ➔ Occurs in many real systems
  - ◆ multithreaded web server
  - ◆ Linux pipe: `cat foo.txt | grep hello`
  - ◆ can have many producers / consumers running concurrently

Producer / Consumer, bounded buffer

```
int buffer[369]
```

```
Producer:
    while true:
        write(buffer)
```

```
Consumer:
    while true:
        read(buffer)
```

➔ Issues to worry about

- only write if buffer is **not full**
- only read if buffer is **not empty**
- write and read buffer correctly without losing data

➔ Use one or more **semaphores** to synchronize correctly

- `sem_init(value), sem_wait(), sem_post()`

Cheatsheet for exercises

`sem_init()`: initialized semaphore to a certain value

`sem_wait()`: decrement value, wait if value is negative

`sem_post()`: increment value, wake up one waiting thread

How to make sure one can only write when buffer is not full?

How to make sure one can only read when buffer is not empty?

What are the four criteria that need to be satisfied?

Solution #1: use two semaphores


```
int buffer[369]; sem_t not_full; sem_t not_empty;
    not_full.init(369);    not_empty.init(0);
```

Producer:

```
P1   while true:
P2       sem_wait(&not_full)
P3       write(buffer)
P4       sem_post(&not_empty)
```

Consumer:

```
C1   while true:
C2       sem_wait(&not_empty)
C3       read(buffer)
C4       sem_post(&not_full)
```

Does it work?

➔ No, if have multiple producers, will enter **critical section** (buffer) at the same time. **Violating Mutual Exclusion!**

Solution #2: add **mutual exclusion** (binary semaphore)


```
int buffer[369]; sem_t not_full; sem_t not_empty; sem_t mutex;
   not_full.init(369);   not_empty.init(0);  mutex.init(1);
```

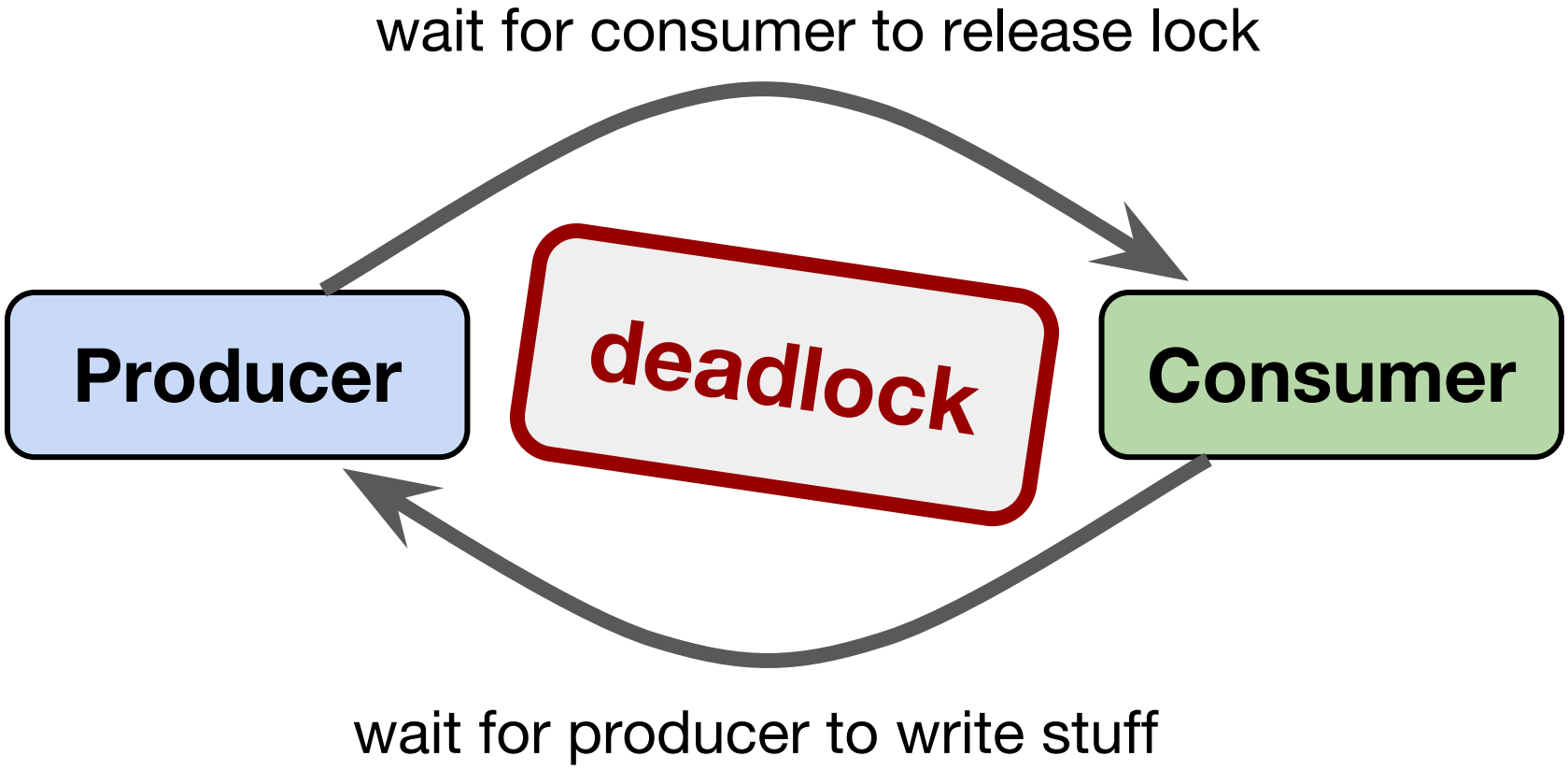
Producer:

```
P1   while true:
P2        sem_wait(&mutex)
P3        sem_wait(&not_full)
P4        write(buffer)
P5        sem_post(&not_empty)
P6        sem_post(&mutex)
```

Consumer:

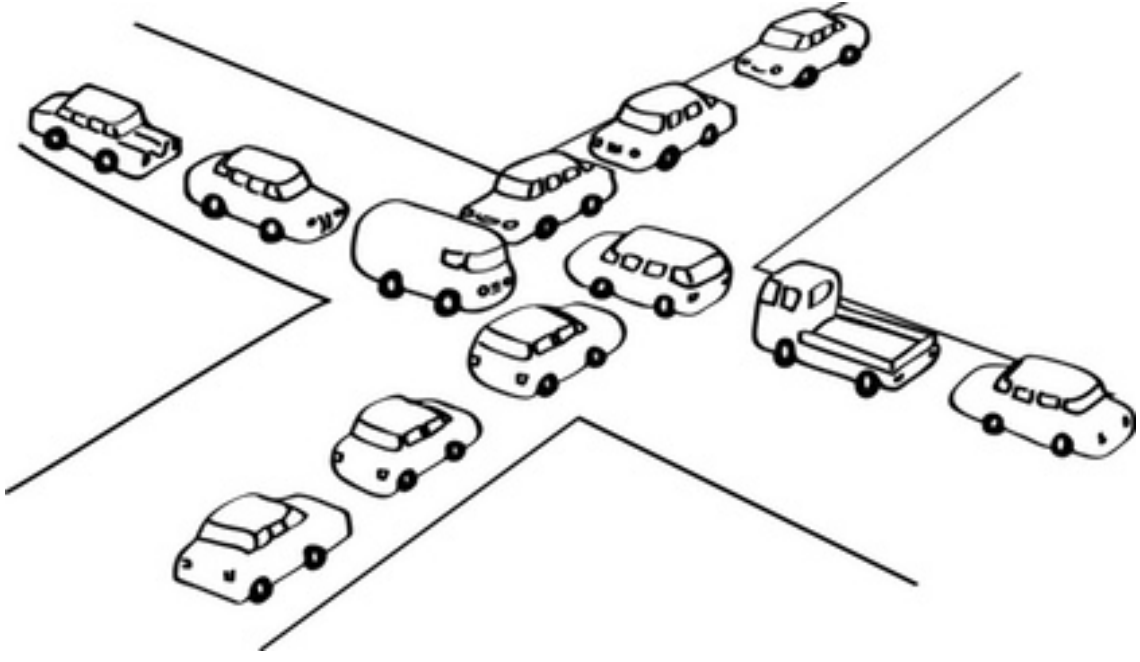
```
C1   while true:
C2        sem_wait(&mutex)
C3        sem_wait(&not_empty)
C4        read(buffer)
C5        sem_post(&not_full)
C6        sem_post(&mutex)
```

What can go wrong here? A consumer first runs, acquires lock, then block because buffer is empty. Then a producer runs, cannot write since it's locked.



Deadlock means starvation.

Cyclic dependencies cause deadlock



Solution #3: how to fix the deadlock


```
int buffer[369]; sem_t not_full; sem_t not_empty; sem_t mutex;
   not_full.init(369);   not_empty.init(0);  mutex.init(1);
```

Producer:

```
P1   while true:
P2        sem_wait(&mutex)
P3        sem_wait(&not_full)
P4        write(buffer)
P5        sem_post(&not_empty)
P6        sem_post(&mutex)
```

Consumer:

```
C1   while true:
C2        sem_wait(&mutex)
C3        sem_wait(&not_empty)
C4        read(buffer)
C5        sem_post(&not_full)
C6        sem_post(&mutex)
```

Solution #3: **deadlock fixed!**


```
int buffer[369]; sem_t not_full; sem_t not_empty; sem_t mutex;
   not_full.init(369);   not_empty.init(0);  mutex.init(1);
```

Producer:

```
P1   while true:
P2        sem_wait(&not_full)
P3        sem_wait(&mutex)
P4        write(buffer)
P5        sem_post(&mutex)
P6        sem_post(&not_empty)
```

Consumer:

```
C1   while true:
C2        sem_wait(&not_empty)
C3        sem_wait(&mutex)
C4        read(buffer)
C5        sem_post(&mutex)
C6        sem_post(&not_full)
```

Consumer waits on empty buffer without locking the mutex, so producer is free to write stuff, no more cyclic dependency.

Takeaway

Always remember the **four criteria** when designing a good synchronization mechanism

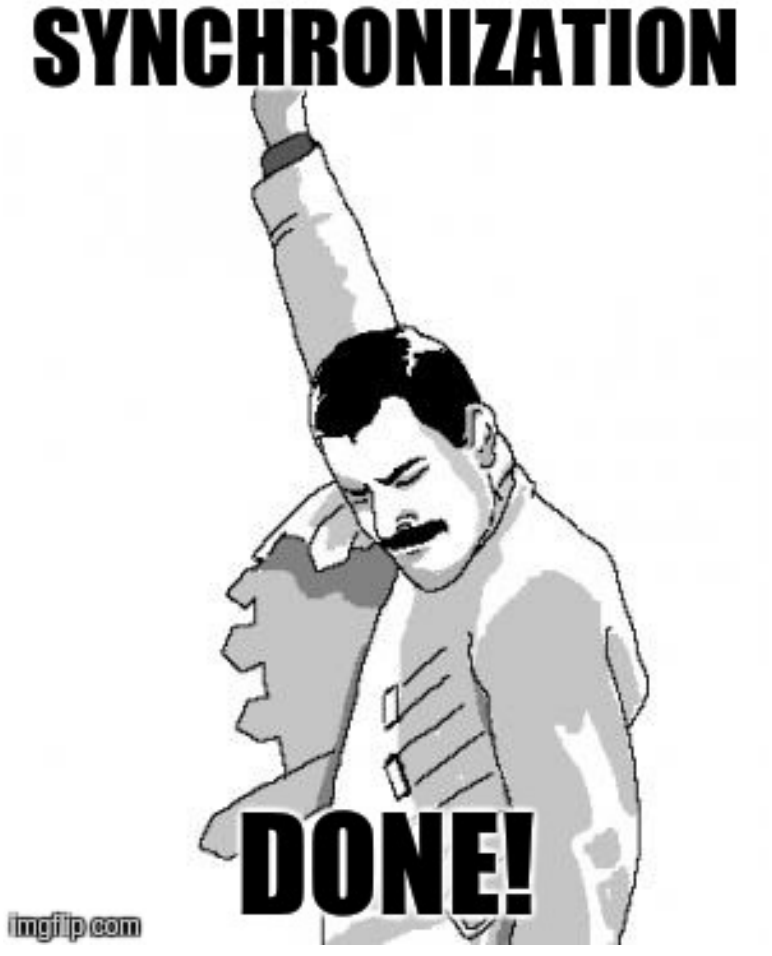
- ➔ **mutual exclusion**: only one thread in critical section
- ➔ **progress**: entry to CS only depends on who are trying to enter.
- ➔ **no starvation, no deadlock**
- ➔ **performance**: synchronization takes much shorter time than the actual work.

Home practice challenge

Solve the producer / consumer problem using only **lock** and **condition variable**, i.e., not using any semaphore.

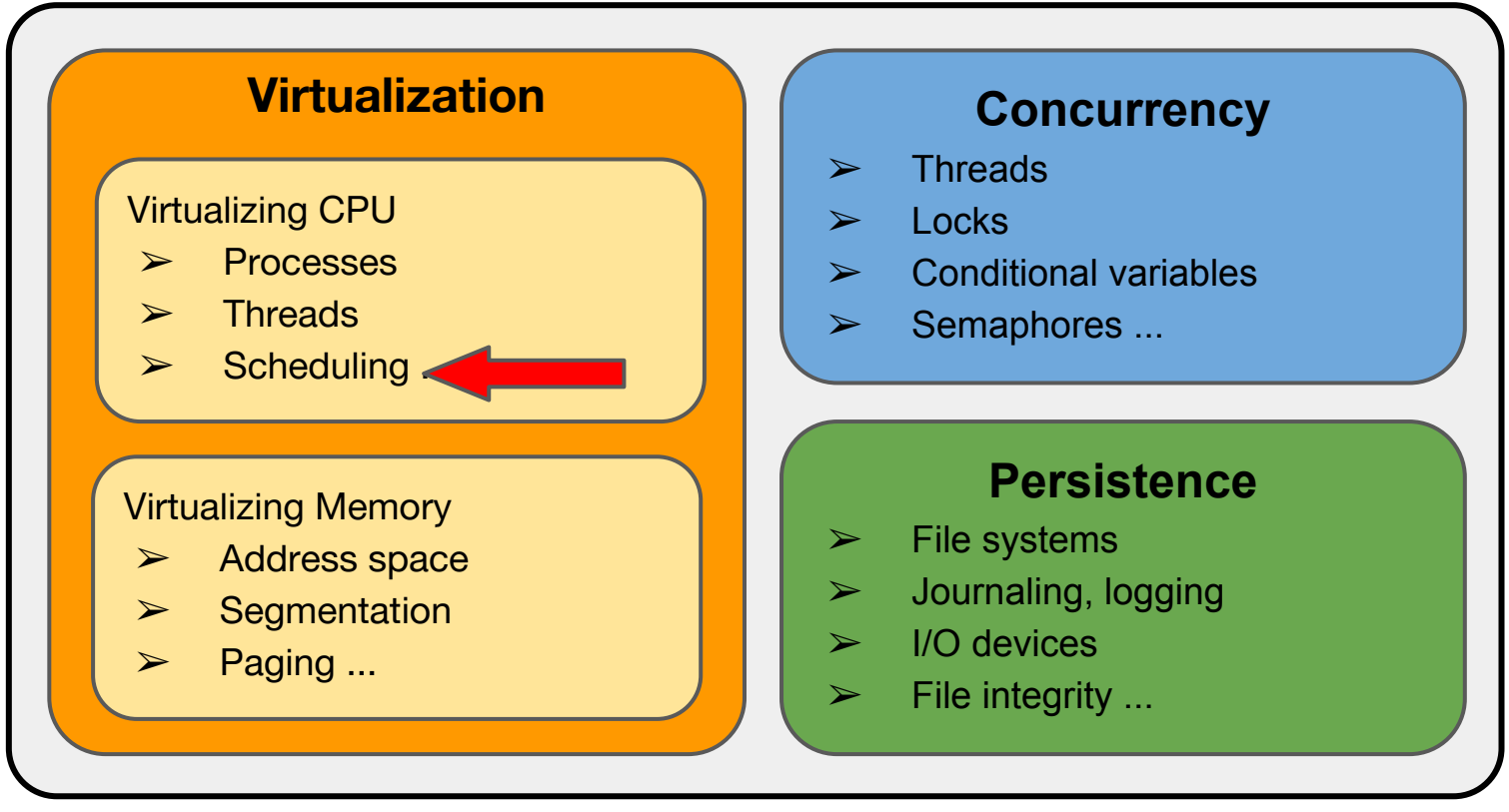
Brief mention: **Monitors**

- ➔ An abstract data type with the restriction that only one thread at a time can be active in the monitor
- ➔ Local data only accessed by procedures in the monitor
- ➔ Different threads trying to enter the monitors cooperate with each other, i.e., they block and wake up each other
- ➔ Consists of a lock and condition variables
- ➔ a.k.a. thread-safe class
- ➔ Provided in Java
- ➔ Read more: http://www.artima.com/insidejvm/ed2/threadsynch.html


SYNCHRONIZATION
DONE!
imgflip.com

Scheduling

We are 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 ...

Intuition

How do we schedule the service for these customers so that their waiting time is short (on average) ?

Overview

- ➔ Goals of scheduling
- ➔ Types of schedulers
- ➔ Scheduling algorithms

What is processor scheduling?

- ➔ The allocation of processors to processes over time
- ➔ Key to **multiprogramming**
- ➔ Want to increase CPU utilization and job throughput by overlapping I/O and computation
- ➔ We have different process queues representing different process states (ready, running, blocked, ...)
- ➔ **Policies**
  - ◆ Given more than one runnable processes, how do we choose which one to run next?

Start simple: 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**

These assumption are quite unrealistic, we will relax them later, therefore get closer to how it works in real systems

Performance metric

Average turnaround time of all jobs

turnaround time = job completion time - job arrival time

There are other metrics such as response time, fairness, but we don’t care about them for now and just focus on turnaround time.

Algorithm 1: First Come, First Serve (FCFS / FIFO)

Job	Arrival Time	Service Time
A	0	3
B	0	3
C	0	3

Is this scheduling optimal?
➔ Yes!

Convention: when choice is arbitrary, choose A before B, B before C

0	1	2	3	4	5	6	7	8	9	10	11
	A										
	waiting			B							
		waiting					C				

Average turnaround time = (3+6+9) / 3 = 6

Assumptions

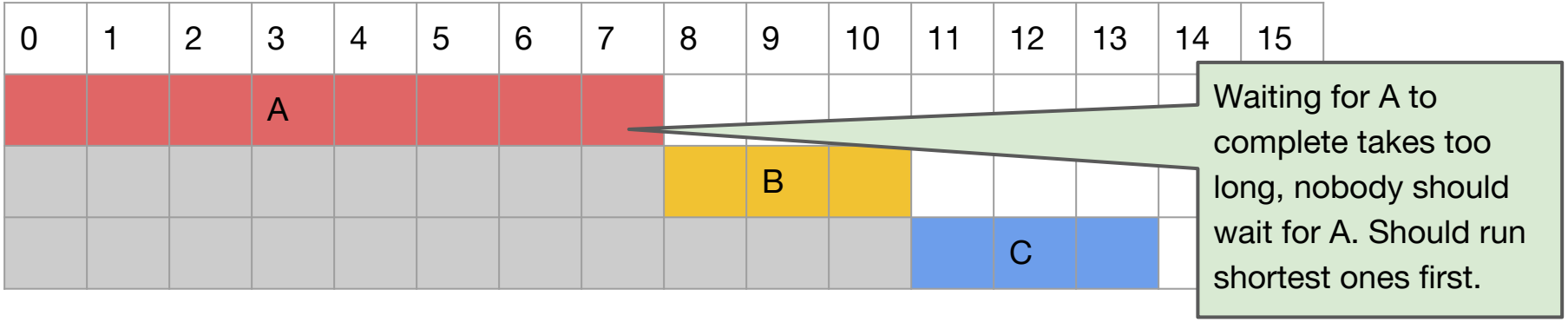
Relax this assumption

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**

Another Case for FCFS

Job	Arrival Time	Service Time
A	0	**8**
B	0	3
C	0	3

Is this scheduling optimal?
➔ NO!
➔ How to improve?



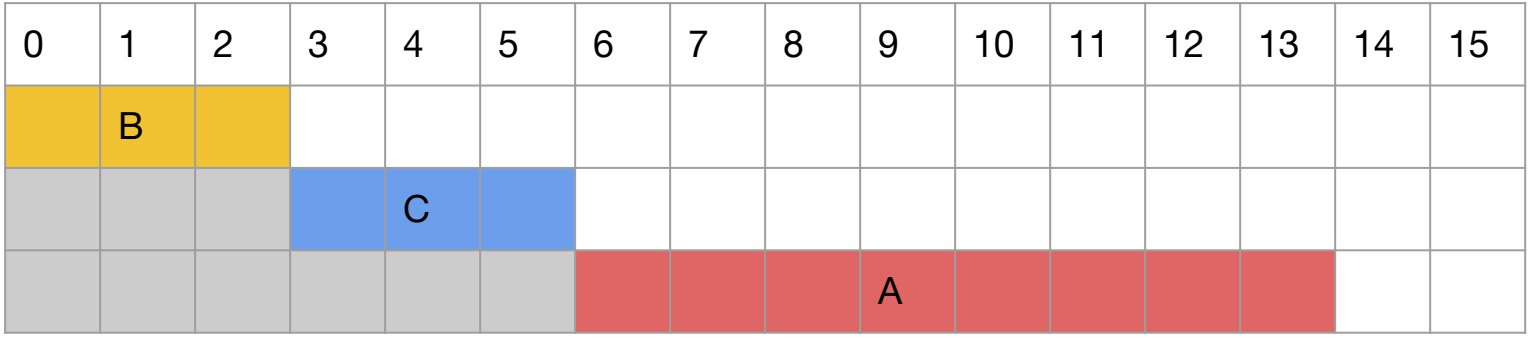
Average turnaround time = (8+11+14) / 3 = 11

Algorithm 2: Shortest Job First (SJF)

BONUS!

Job	Arrival Time	Service Time
A	0	**8**
B	0	3
C	0	3

Is this scheduling optimal?
➔ Yes!



Average turnaround time = (3+6+14) / 3 = 7.67

Assumptions

assumption relaxed

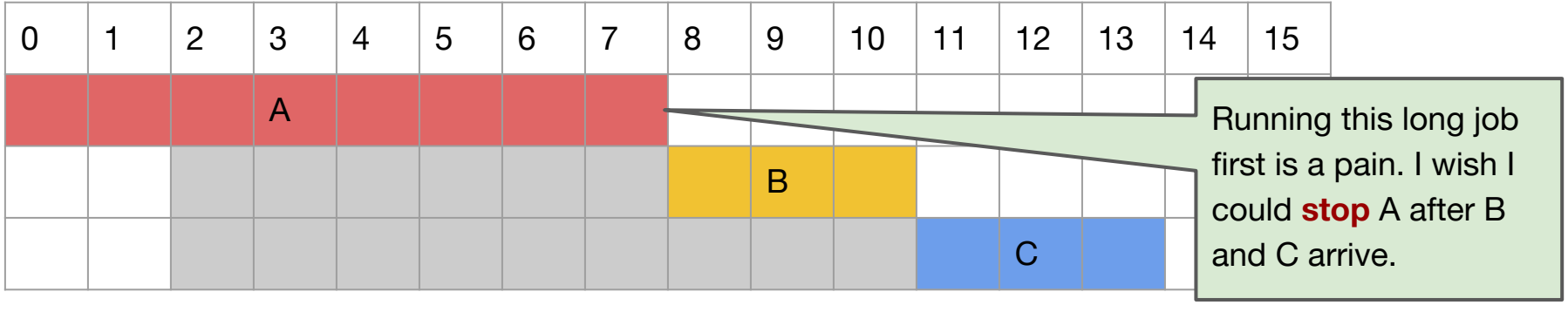
1. Each job (process/thread) runs the **same amount of time**
2. All jobs **arrive at the same time**
3. Once started, each job **runs to completion** (no interruption in the middle)
4. All jobs only use the **CPU** (no I/O)
5. The runtime of each job is **known**

Relax this assumption

Another Case for SJF

Job	Arrival Time	Service Time
A	0	8
B	**2**	3
C	**2**	3

Is this scheduling optimal?
➔ NO!
➔ How to improve?



Average turnaround time = (8+9+12) / 3 = 9.67

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

Relax this assumption

Preemptive *VS* Non-preemptive

- ➔ Non-preemptive scheduling
  - once the CPU has been allocated to a process, it keeps the CPU until it terminates or blocks
  - suitable for batch scheduling, where we only care about the total time to finish the whole batch
- ➔ Preemptive scheduling
  - CPU can be taken from a running process and allocated to another (remember how we do this in hardware?)

    timer interrupt and context switch
  - Needed in interactive or real-time systems, where **response time** of each process matters

Now we make our scheduler preemptive!

Algorithm 3: Shortest Time-to-Completion First (STCF) a.k.a. Preemptive Shortest Job First (PSJF)

Job	Arrival Time	Service Time
A	0	8
B	**2**	3
C	**2**	3

Is this scheduling optimal?
➔ Yes!

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A										A					
			B												
						C									

Average turnaround time = (14+3+6) / 3 = 7.67

So far, by “**optimal**” we mean we mean optimal **average turnaround time**.

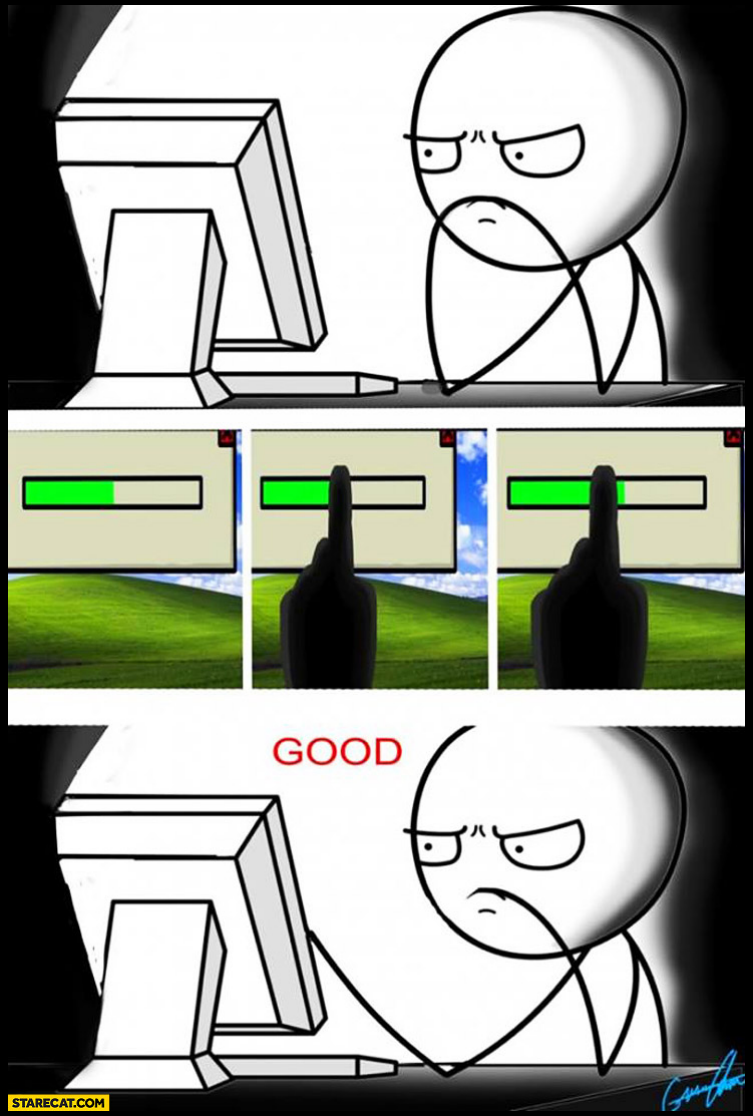
Now we want a new performance metric: **response time**

➔ the time from when the job arrives to when it is first scheduled

response time = first-scheduled time - arrival time

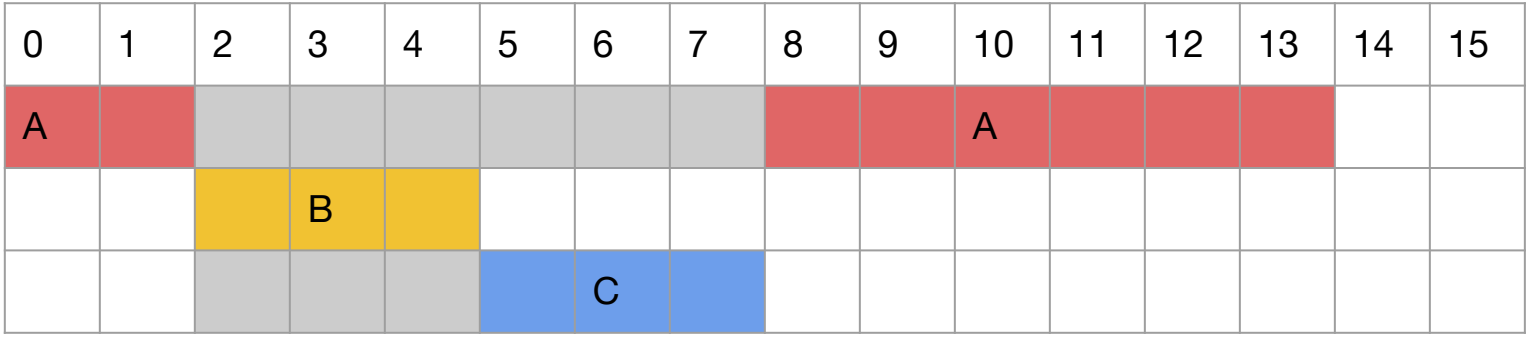
makes more sense for **interactive** systems, which most of our OSs are.

Response time matters more than turnaround time



Algorithm 3: Shortest Time-to-Completion First (STCF) a.k.a. Preemptive Shortest Job First (PSJF)

Job	Arrival Time	Service Time
A	0	8
B	**2**	3
C	**2**	3



Average **response time** = (0+0+3) / 3 = 1

Algorithm 4: Round Robin (RR)

➔ All jobs put on a **circular run queue**

➔ Each job is allowed to run for **time quantum q** before being preempted and put back on queue

Algorithm 4: Round Robin (RR)

Job	Arrival Time	Service Time
A	0	8
B	**2**	3
C	**2**	3

choose q = 1

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A				A			A				A				
		B			B			B							
			C			C			C						

Average **response time** = (0+0+1) / 3 = 0.33

Algorithm 4: Round Robin (RR)

The choice of **time quantum q** is important

➔ if q = ∞, RR becomes …

- ◆ non-preemptive FCFS

➔ if q = 0, RR becomes …

- ◆ simultaneous sharing of processor, an idealisation which is not really possible.

➔ q should be a multiple of the timer interrupt interval

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

Relax this assumption

All useful processes perform I/O

➔ when a job is performing I/O, it is not using the CPU, i.e., it is **blocked** waiting for I/O completion

➔ it makes sense to use the waiting time to run some other jobs.

Jobs with I/O

Job	Arrival Time	CPU Time	I/O
A	0	5	one per 1 sec
B	0	5	none

Assume scheduler does not know how long I/O takes

Normal STCF treating A as **one single job**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	I/O	A	I/O	A	I/O	A	I/O	A							
										B					

STCF treating A as **5 sub-jobs**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	I/O	A	I/O	A	I/O	A	I/O	A							
	B		B		B		B		B						

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

This one is tough. Will talk about it next time.

Tutorial this week

➔ exercises on synchronization

Next lecture

➔ more on scheduling

➔ intro to virtual memory

Gentle reminder:

Assignment 1 due next Wednesday!