Barriers, Readers/Writers, Scheduling Intro

26 December 2024
Lecture 8

Slides adapted from John Kubiatowicz (UC Berkeley)

Concept Review

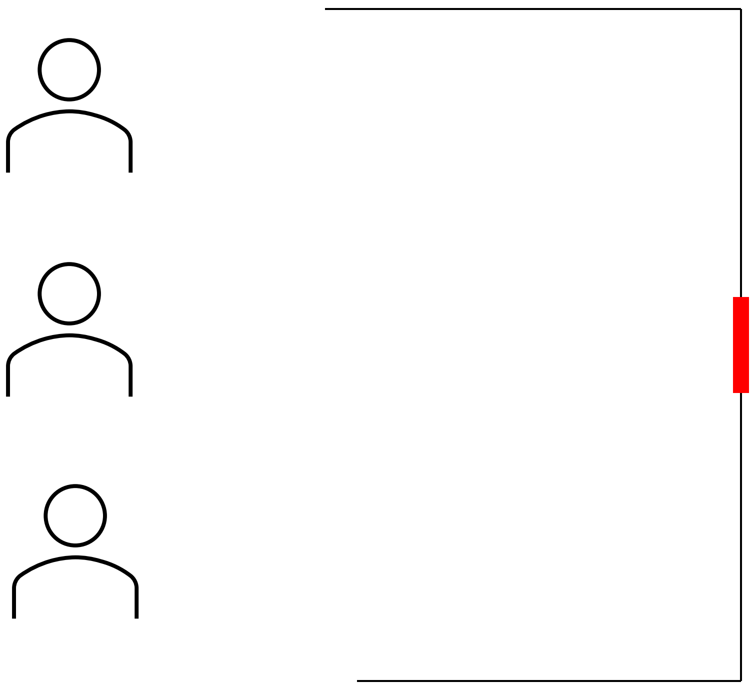
Atomic Reads and Writes	Starvation	Race condition	Mutual exclusion
Critical section	Lock • Spin lock • In-Kernel lock	Busy waiting	Semaphores
	Condition Variables	Monitors	

Topics for Today

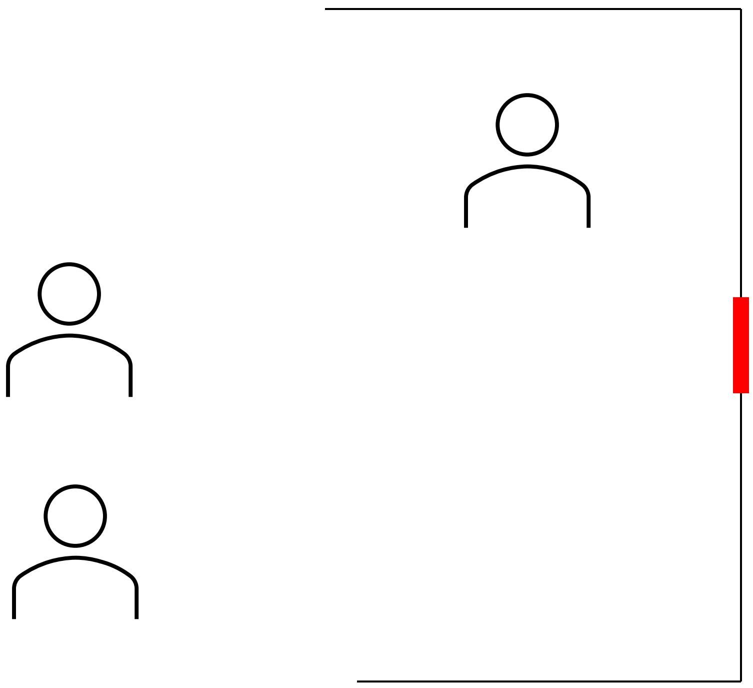
- Higher Level Synchronization Atoms
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language
- Scheduling
  - FIFO
  - Round Robin

Concepts today

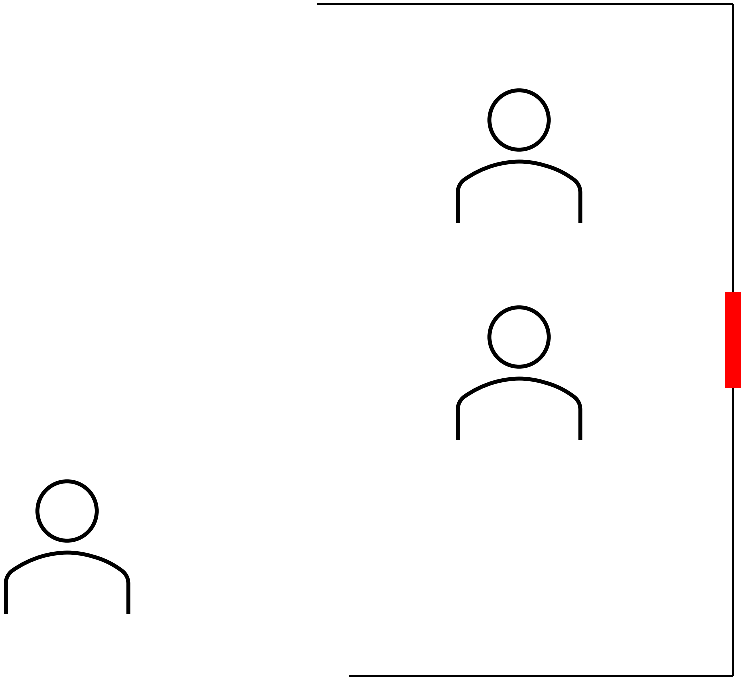
Barrier Synchronization - 3



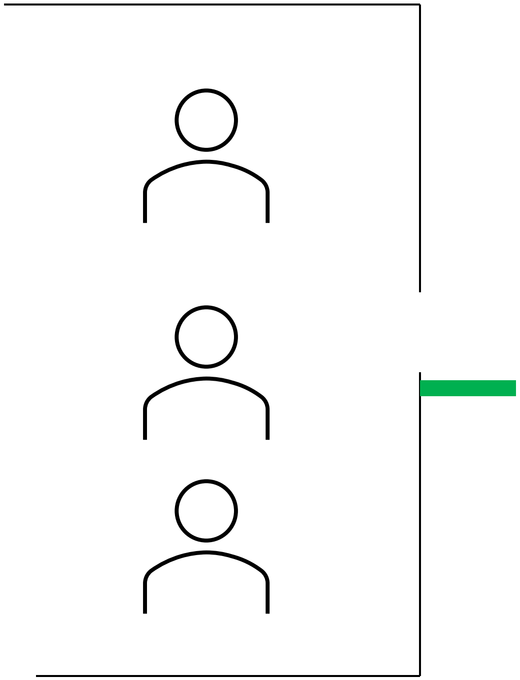
Barrier Synchronization - 3



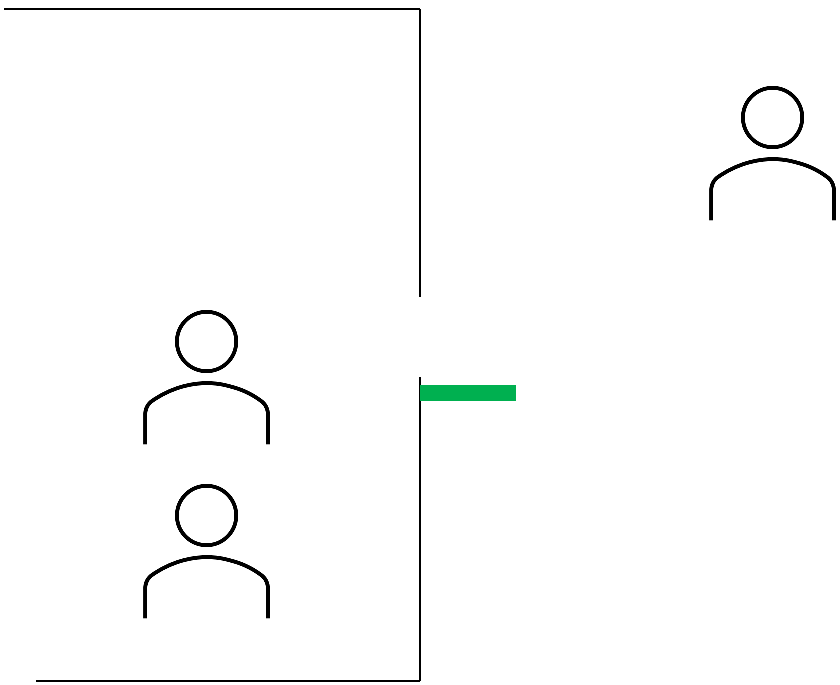
Barrier Synchronization - 3



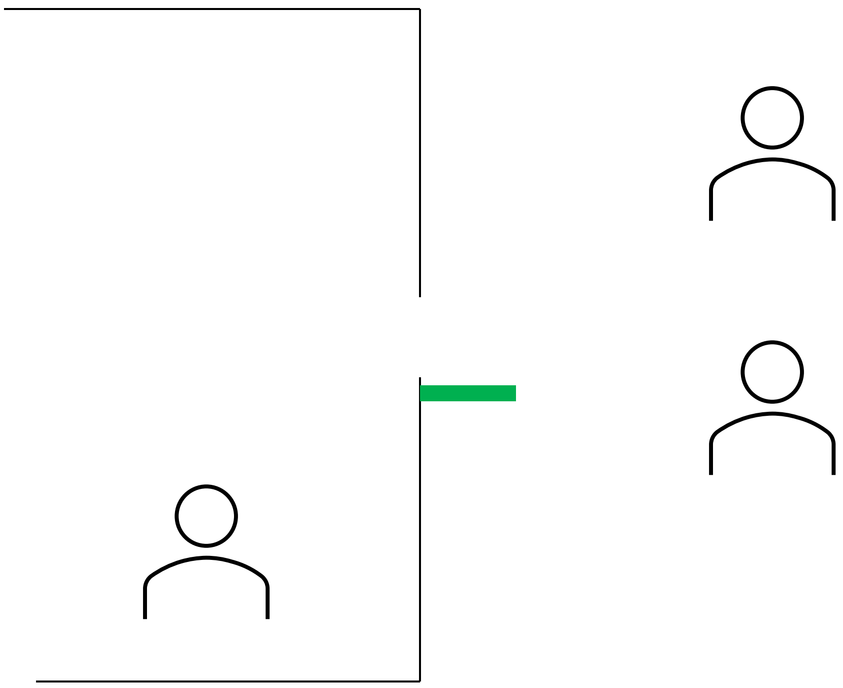
Barrier Synchronization - 3



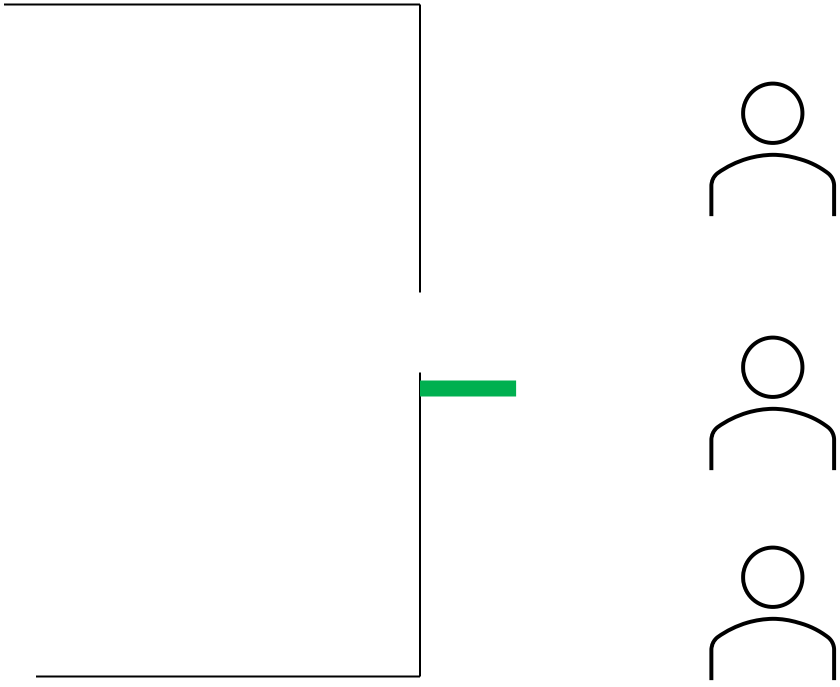
Barrier Synchronization - 3



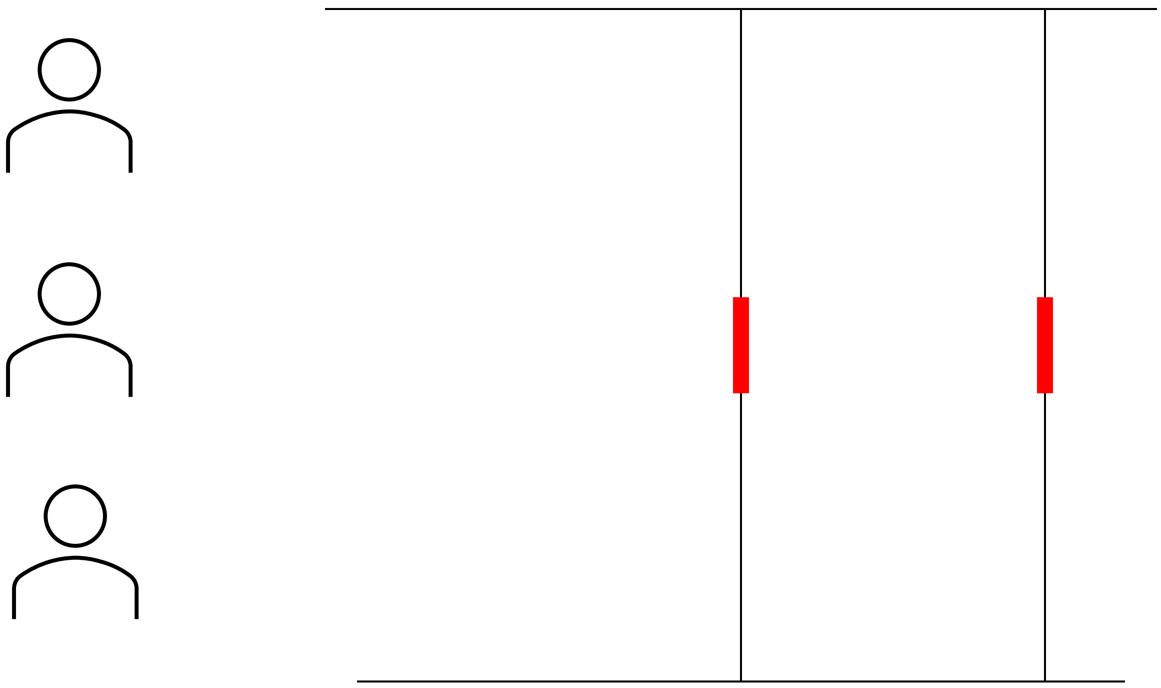
Barrier Synchronization - 3



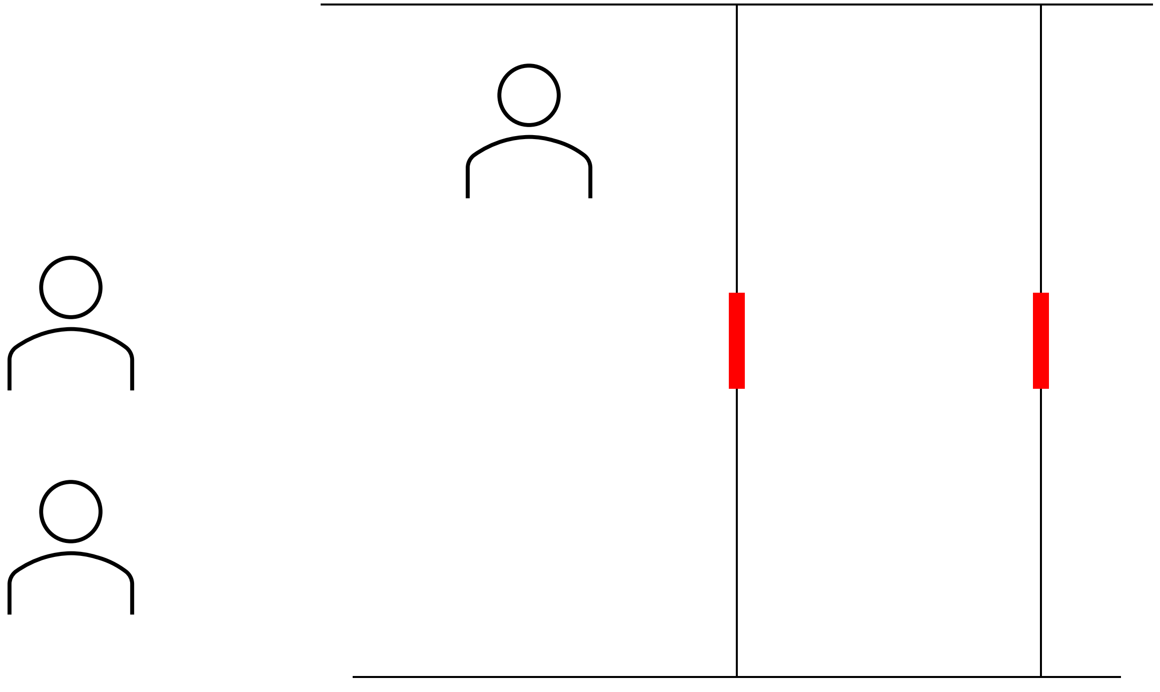
Barrier Synchronization - 3



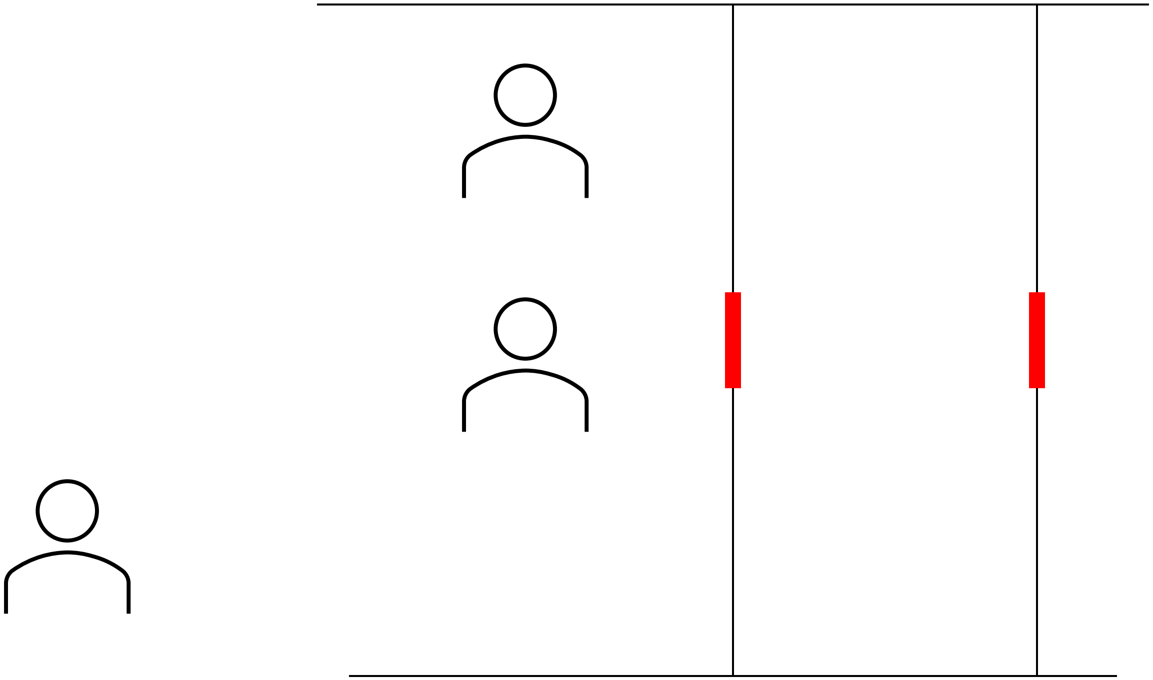
Reusable Barrier Synchronization - 3



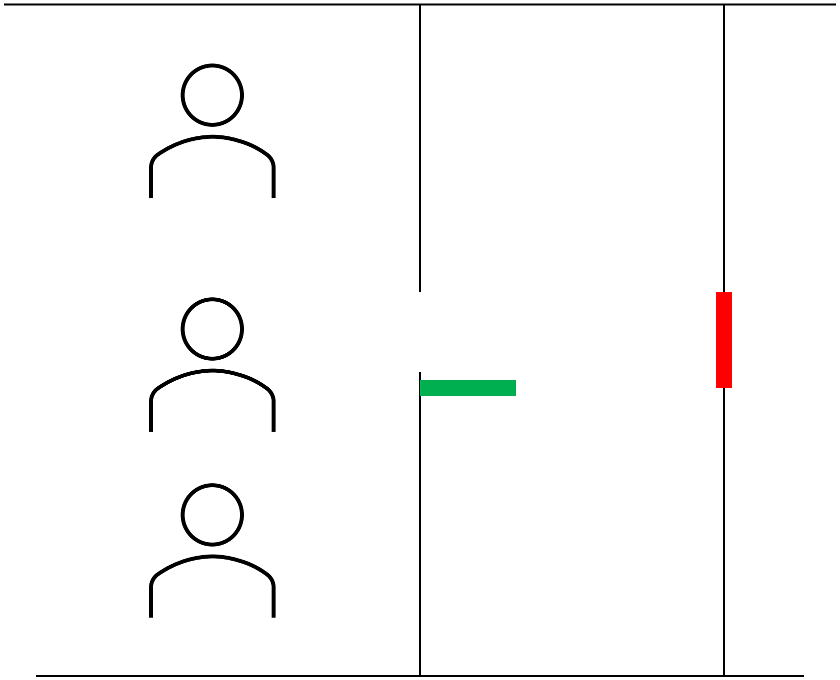
Reusable Barrier Synchronization - 3



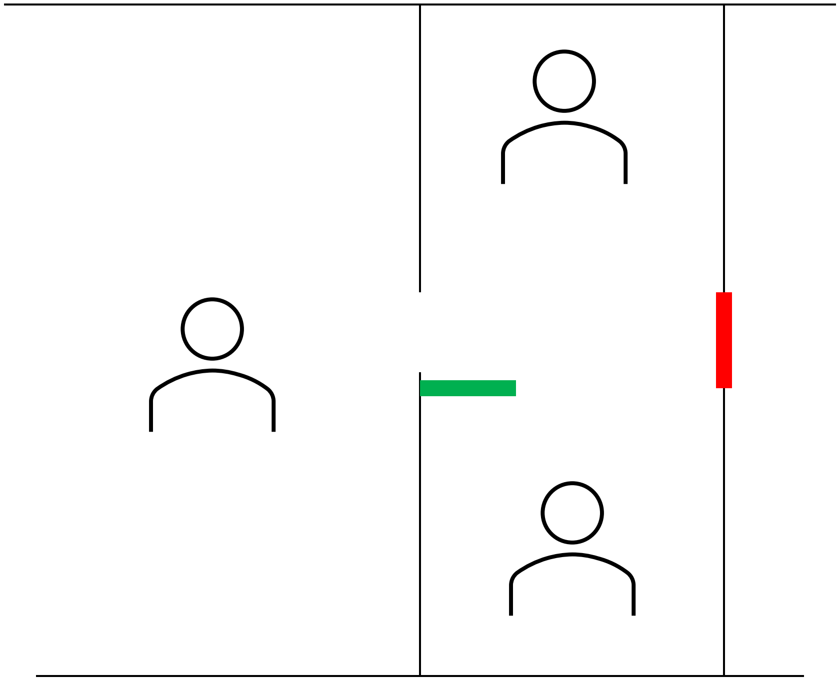
Reusable Barrier Synchronization - 3



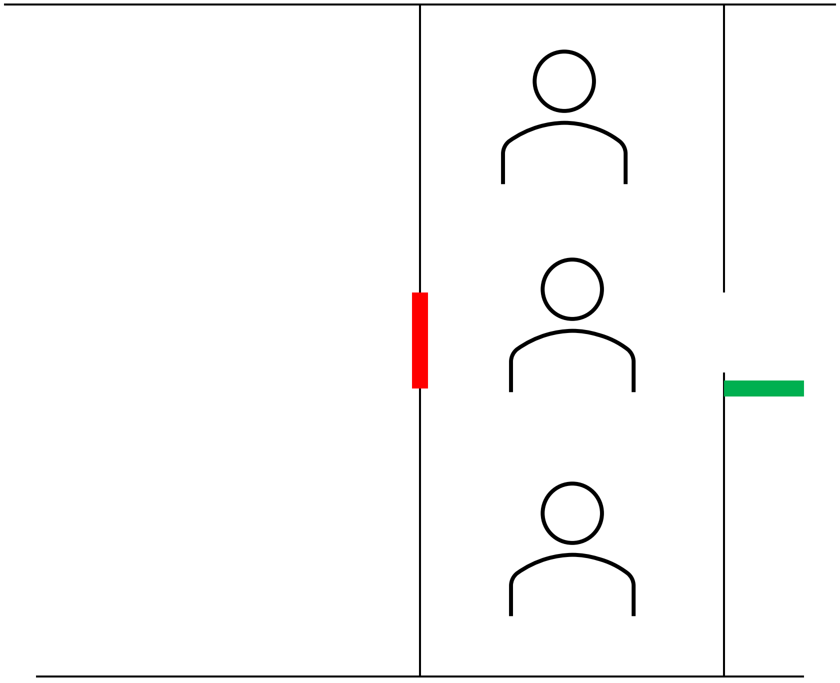
Reusable Barrier Synchronization - 3



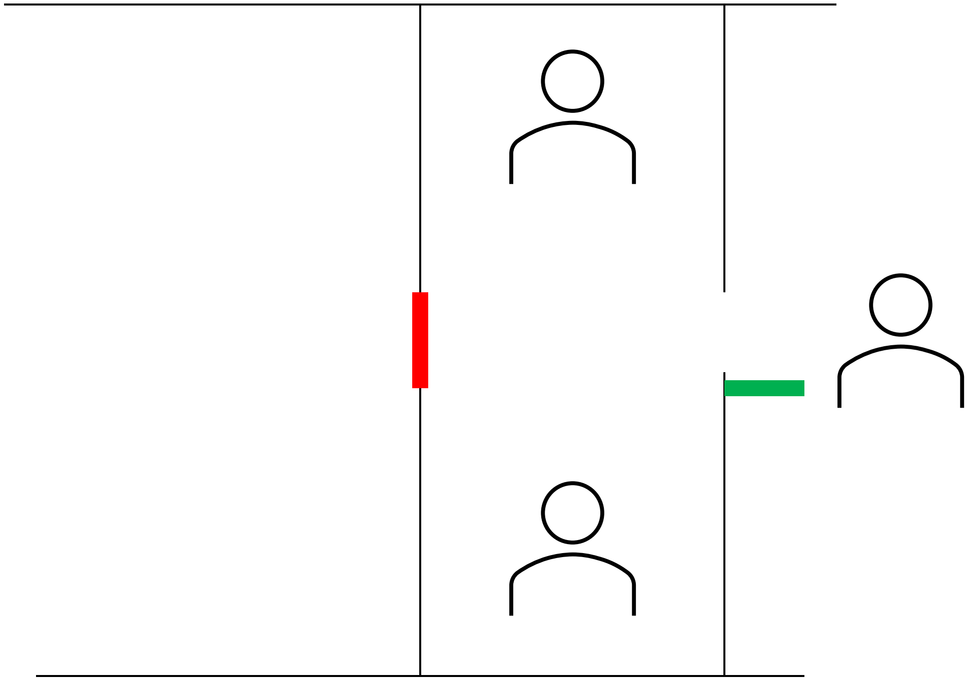
Reusable Barrier Synchronization - 3



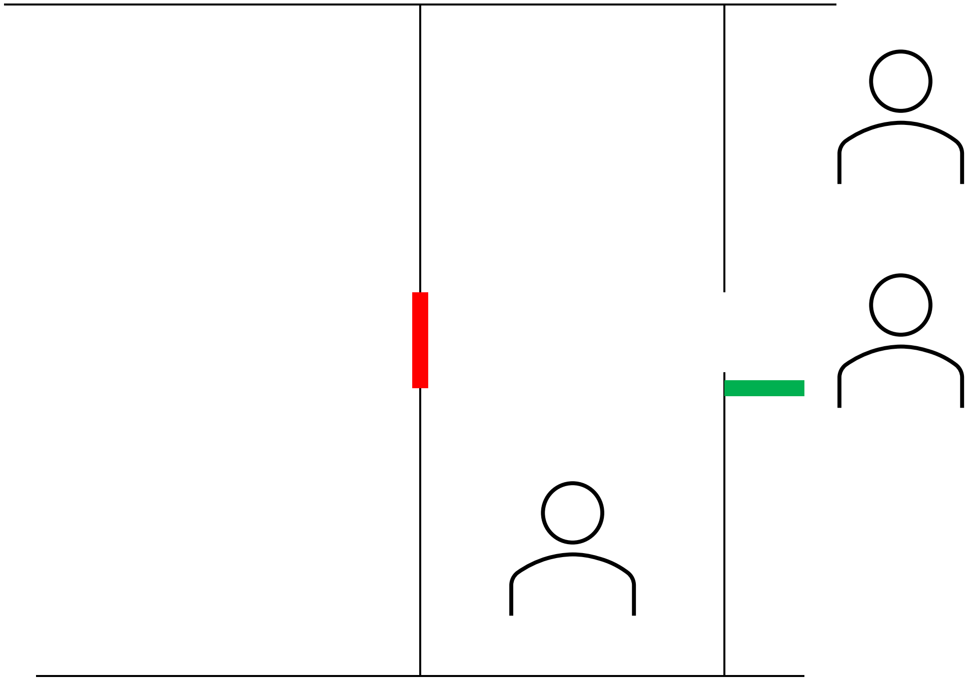
Reusable Barrier Synchronization - 3



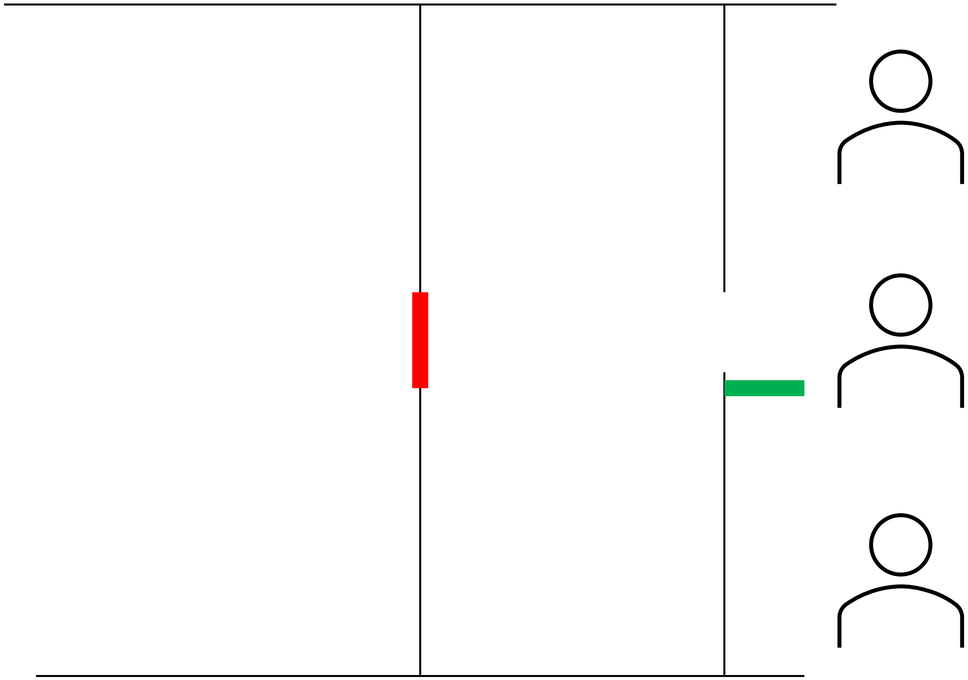
Reusable Barrier Synchronization - 3



Reusable Barrier Synchronization - 3



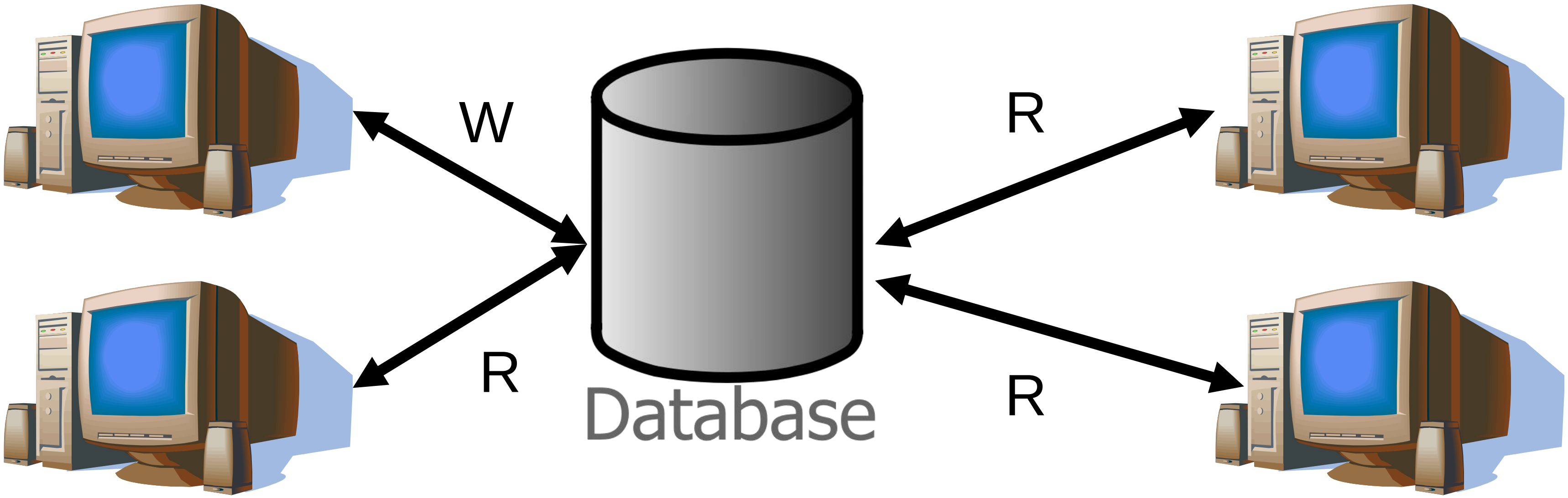
Reusable Barrier Synchronization - 3



So Far

- Higher Level Synchronization Atoms
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language
- Scheduling
  - FIFO
  - Round Robin

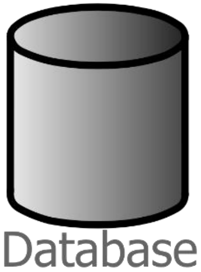
Extended example: Readers/Writers Problem



Motivation: Consider a shared database

- Two classes of users:
  - Readers – never modify database
  - Writers – read and modify database
- Is using a single lock on the whole database sufficient?
  - Allow many readers at the same time
  - Only one writer at a time

Basic Readers/Writers Solution



- Correctness Constraints:
  - Readers can access database when no writers
  - Writers can access database when no readers or writers
  - Only one thread manipulates state variables at a time
- Basic structure of a solution:
  - `Reader()`

```
Wait until no writers
Access data base
Check out – wake up a waiting writer
```

  - `Writer()`

```
Wait until no active readers or writers
Access database
Check out – wake up waiting readers or writer
```

Basic Readers/Writers Solution

- State variables (Protected by a lock called “lock”):
  - `int AR`: Number of active readers; initially = 0
  - `int WR`: Number of waiting readers; initially = 0
  - `int AW`: Number of active writers; initially = 0
  - `int WW`: Number of waiting writers; initially = 0
  - Condition `okToRead = NIL`
  - Condition `okToWrite = NIL`

Code for a Reader

```
Reader() {
  // First check self into system
  lock.Acquire();
  while ((AW + WW) > 0) {     // Is it safe to read?
    WR++;                     // No. Writers exist
    okToRead.wait(&lock);     // Sleep on cond var
    WR--;                     // No longer waiting
  }
  AR++;                       // Now we are active!
  lock.release();
  // Perform actual read-only access
  AccessDatabase(ReadOnly);
  // Now, check out of system
  lock.Acquire();
  AR--;                       // No longer active
  if (AR == 0 && WW > 0)      // No other active readers
    okToWrite.signal();       // Wake up one writer
  lock.Release();
}
```

Why?

Code for a Writer

```
Writer() {
  // First check self into system
  lock.Acquire();
  while ((AW + AR) > 0) {            // Is it safe to write?
      WW++;  // No. Active users exist
      okToWrite.wait(&lock);         // Sleep on cond var
      WW--;  // No longer waiting
  }
  AW++;      // Now we are active!
  lock.release();
  // Perform actual read/write access
  AccessDatabase(ReadWrite);
  // Now, check out of system
  lock.Acquire();
  AW--;      // No longer active
  if (WW > 0){                       // Give priority to writers
      okToWrite.signal();            // Wake up one writer
  } else if (WR > 0) {               // Otherwise, wake reader
      okToRead.broadcast();          // Wake all readers
  }
  lock.Release();
}
```

Why?

Why?

Simulation R/W Step 1

- Consider the following sequence of operators:
  - `R1, R2, W1, R3`
- On entry, each reader checks the following:

```
while ((AW + WW) > 0) {      // Is it safe to read?
    WR++;                    // No. Writers exist
    okToRead.wait(&lock);    // Sleep on cond var
    WR--;                    // No longer waiting
 }
AR++;                        // Now we are active!
```

- First, `R1` comes along:
  `AR = 1, WR = 0, AW = 0, WW = 0`
- Second, `R2` comes along:
  `AR = 2, WR = 0, AW = 0, WW = 0`
- Now, readers make take a while to access database
  - Situation: Locks released
  - Only `AR` is non-zero

Simulation R/W Step 2

- Next, `W1` comes along:

```
while ((AW + AR) > 0) {    // Is it safe to write?
   WW++;                   // No. Active users exist
   okToWrite.wait(&lock);  // Sleep on cond var
   WW--;                   // No longer waiting
}
AW++;
```

- Can't start because of readers, so go to sleep:

```
AR = 2, WR = 0, AW = 0, WW = 1
```

- Finally, `R3` comes along:

```
AR = 2, WR = 1, AW = 0, WW = 1
```

- Now, say that `R2` finishes before `R1`:

```
AR = 1, WR = 1, AW = 0, WW = 1
```

- Finally, last of first two readers (`R1`) finishes and wakes up a writer:

```
if (AR == 0 && WW > 0)     // No other active readers
   okToWrite.signal();     // Wake up one writer
```

Simulation R/W Step 3

- When the writer wakes up, get:

```
AR = 0, WR = 1, AW = 1, WW = 0
```

- Then, when writer finishes:

```
if (WW > 0){                // Give priority to writers
   okToWrite.signal();      // Wake up one writer
} else if (WR > 0) {        // Otherwise, wake reader
   okToRead.broadcast();    // Wake all readers
}
```

- Writer wakes up reader, so get:

```
AR = 1, WR = 0, AW = 0, WW = 0
```

- When reader completes, we are finished

Questions about R/W

- Can readers starve? Consider `Reader()` entry code:

```
while ((AW + WW) > 0) {     // Is it safe to read?
   WR++;                    // No. Writers exist
   okToRead.wait(&lock);    // Sleep on cond var
   WR--;                    // No longer waiting
 }
AR++;                       // Now we are active!
```

- What if we erase the condition check in Reader exit?

```
AR--;                       // No longer active
if (AR == 0 && WW > 0)      // No other active readers
   okToWrite.signal();      // Wake up one writer
```

- Further, what if we turn the `signal()` into `broadcast()`

```
AR--;                       // No longer active
okToWrite.broadcast();      // Wake up one writer
```

- Finally, what if we use only one condition variable (call it "`okToContinue`") instead of two separate ones?
  – Both readers and writers sleep on this variable
  – Must use `broadcast()` instead of `signal()`

Monitors Conclusion

- Monitors represent the logic of the program
  - `Wait` if necessary
  - `Signal` when you change something so any waiting threads can proceed
- Basic structure of monitor-based program:

```
lock
while (need to wait) {
   condvar.wait();
}
unlock
```

Check or update state variables.
Wait if necessary

```
do something so no need to wait

lock

condvar.signal();

unlock
```

Check or update state variables.

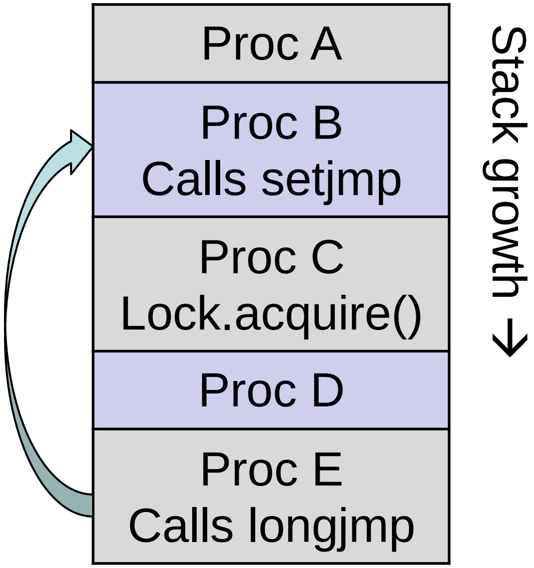
So Far

- Higher Level Synchronization Atoms
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language
- Scheduling
  - FIFO
  - Round Robin

C-Language Support for Synchronization

- C language: Straightforward synchronization
  - Just make sure you know *all* the code paths out of a critical section

```
int Rtn() {
        lock.acquire();
        …
        if (exception) {
            lock.release();
            return errReturnCode;
        }
        …
        lock.release();
        return OK;
}
```



  - Watch out for `setjmp/longjmp`!
    - Can cause a non-local jump out of procedure
    - In example, procedure E calls `longjmp`, popping stack back to procedure B
    - If Procedure C had `lock.acquire`, problem!

C++ Language Support for Synchronization

- Languages with exceptions like C++
  - Languages that support exceptions are problematic (easy to make a non-local exit without releasing lock)
  - Consider:

```
void Rtn() {
   lock.acquire();
   …
   DoFoo();
   …
   lock.release();
}
void DoFoo() {
   …
   if (exception) throw errException;
   …
}
```

  - Notice that an exception in `DoFoo()` will exit without releasing the lock!

C++ Language Support for Synchronization

- Must catch all exceptions in critical sections
  - Catch exceptions, release lock, and re-throw exception:

```
void Rtn() {
   lock.acquire();
   try {
      …
      DoFoo();
      …
   } catch (…) {         // catch exception
      lock.release();    // release lock
      throw;             // re-throw the exception
   }
   lock.release();
}
void DoFoo() {
   …
   if (exception) throw errException;
   …
}
```

  - Even Better: `auto_ptr<T>` facility. See C++ Spec.
    - Can deallocate/free lock regardless of exit method

Java Language Support for Synchronization

- Java has explicit support for threads and thread synchronization
- Bank Account example:

```
class Account {
   private int balance;
   // object constructor
   public Account (int initialBalance) {
      balance = initialBalance;
   }
   public synchronized int getBalance() {
      return balance;
   }
   public synchronized void deposit(int amount) {
      balance += amount;
   }
}
```

   – Every object has an associated lock which gets automatically acquired and released on entry and exit from a *synchronized* method.

Java Language Support for Synchronization

Java also has *synchronized* blocks:

```
int i, j;
void foo() {
   Object locker = new Object();
    synchronized (locker) {
       i += j;
     }
   }
```

- Since every Java object has <u>one</u> associated lock, the statement acquires and releases the object's lock on entry and exit of the block
- Problem is that the code here doesn't protect anything. Why?

Java Language Support for Synchronization

A better form of the code:

```
Object locker = new Object();
int i, j;
void foo() {
    synchronized (locker) {
      i += j;
    }
  }
```

- Now all threads will use the same lock and we'll get some mutual exclusion.

Java Language Support for Synchronization

- Works properly even with exceptions:

```
synchronized (locker) {
   …
   DoFoo();
   …
}
void DoFoo() {
   throw errException;
}
```

- Lock is released when the exception is thrown.

Java Language Support for Synchronization

- Every object also has <u>one</u> condition variable associated with it
  - How to `wait` inside a synchronization method of block:
    - `void wait(long timeout); // Wait for timeout`
    - `void wait(long timeout, int nanoseconds); //variant`
    - `void wait();`
  - How to `signal` in a synchronized method or block:
    - `void notify();      // wakes up oldest waiter`
    - `void notifyAll(); // like broadcast, wakes everyone`

Java Language Support for Synchronization

- Condition variables can wait for a bounded length of time. This is useful for handling exception cases:

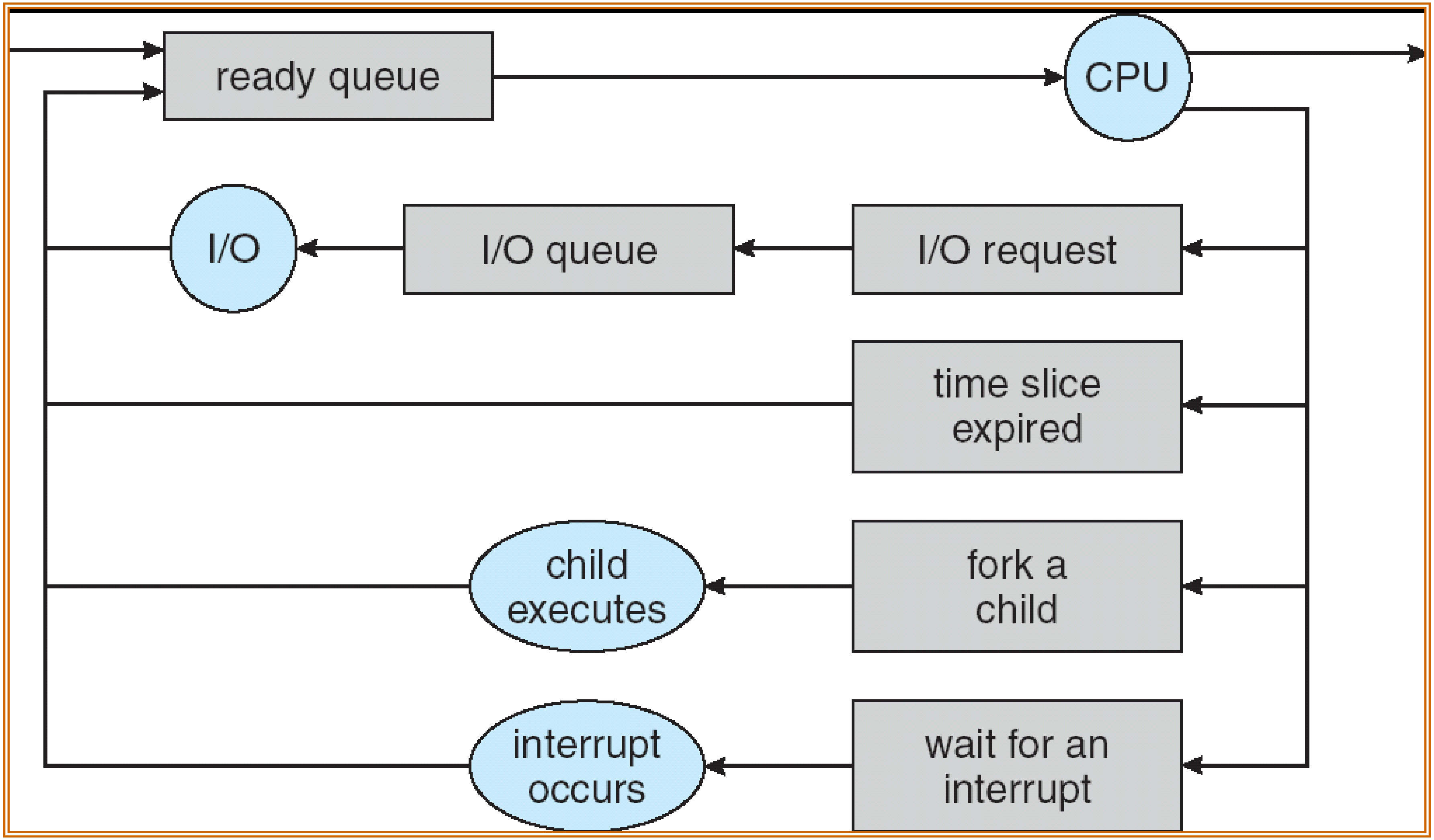
```
t1 = time.now();
while (!ATMRequest()) {
   wait (CHECKPERIOD);
   t2 = time.new();
   if (t2 – t1 > LONG_TIME) checkMachine();
}
```

- Not all Java VMs equivalent!
  - Different scheduling policies, not necessarily preemptive!

So Far

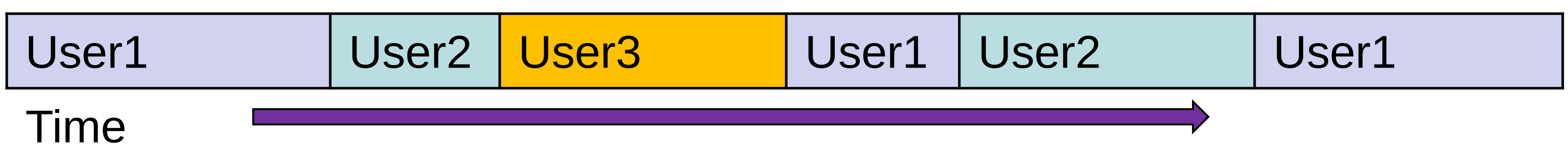
- Higher Level Synchronization Atoms
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language
- Scheduling
  - FIFO
  - Round Robin

Recall: CPU Scheduling

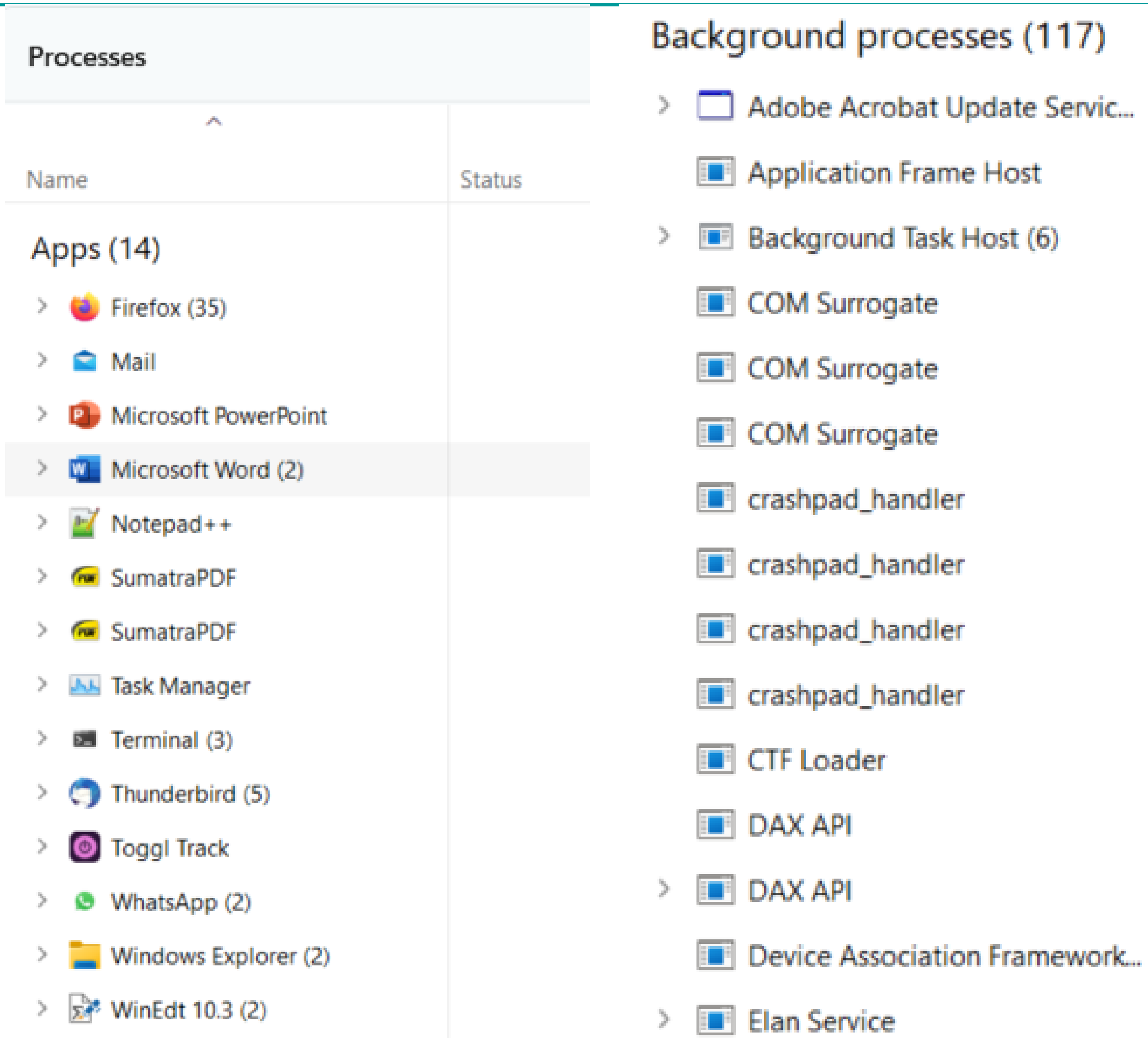


Scheduling Assumptions (review)

- CPU scheduling big area of research in early 70's
- Many implicit assumptions for CPU scheduling:
  - One program per user
  - One thread per program
  - Programs are independent
- Clearly, these are unrealistic but they simplify the problem so it can be solved
  - For instance: is "fair" about fairness among users or programs?
    - If I run one compilation job and you run five, you get five times as much CPU on many operating systems
- The high-level goal: Dole out CPU time to optimize some desired parameters of system



A snapshot

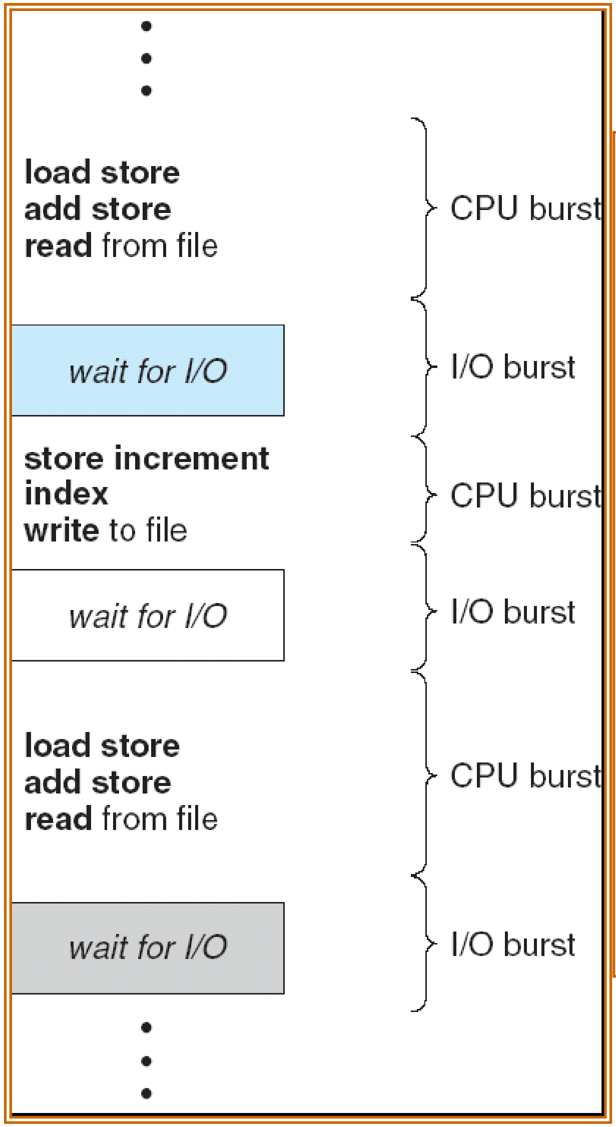


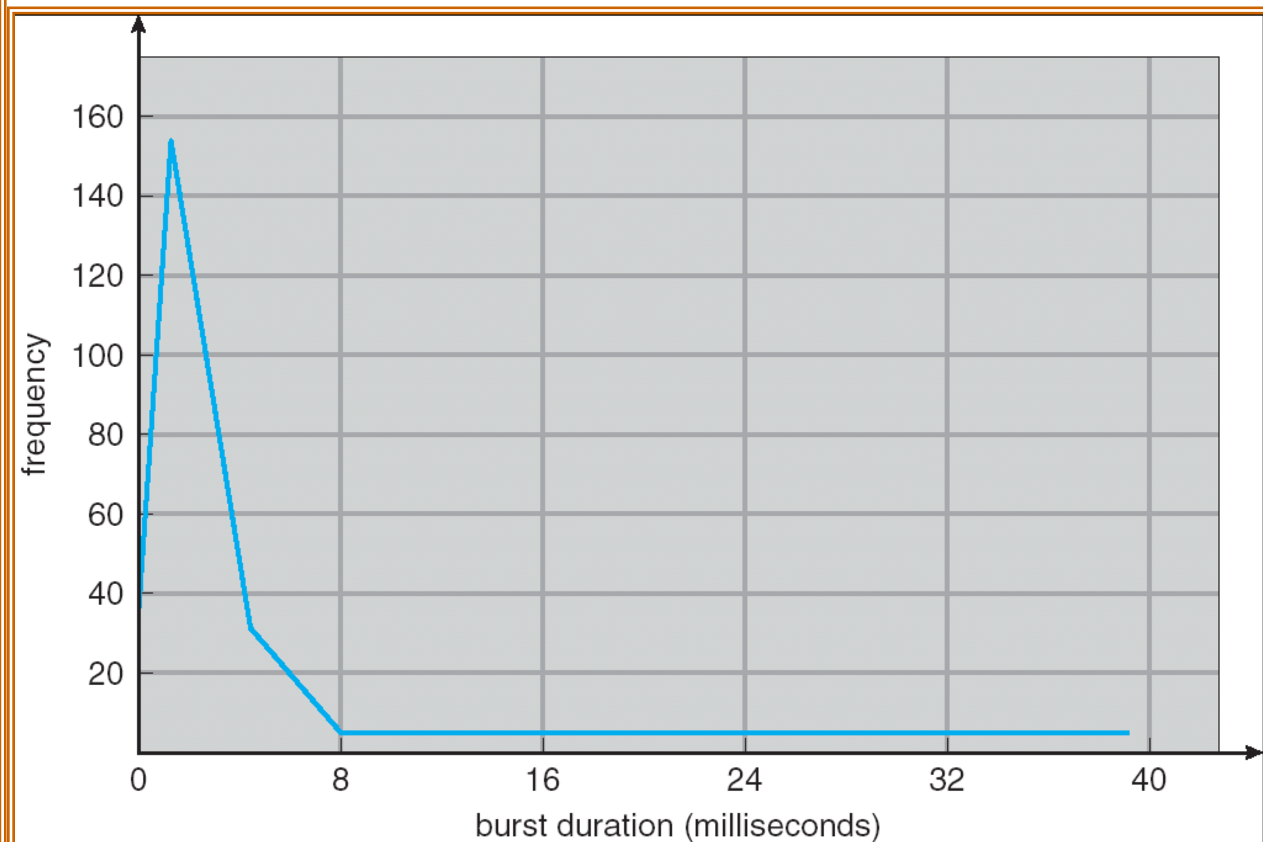
Then I click on

How much CPU resources does each process get?

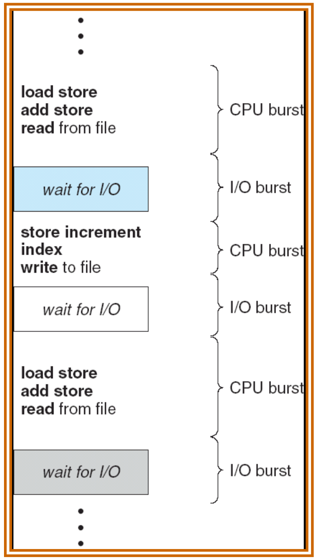
The new one?

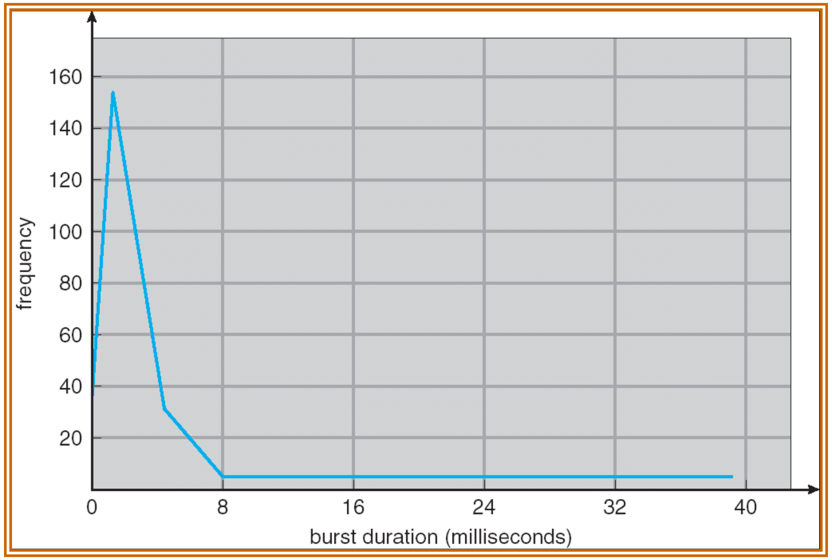
Assumption: CPU Bursts





Assumption: CPU Bursts





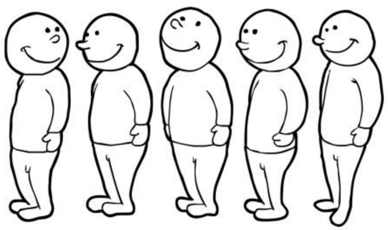
- Execution model: programs alternate between bursts of CPU and I/O
  - Program typically uses the CPU for some period of time, then does I/O, then uses CPU again
  - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
  - With time slicing, thread may be forced to give up CPU before finishing current CPU burst

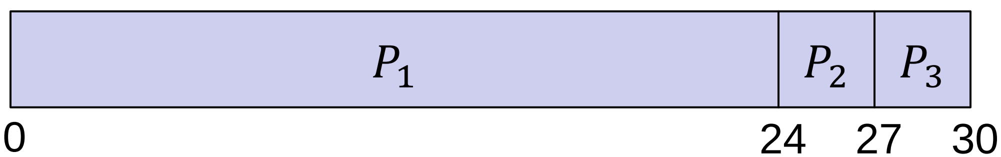
First-Come, First-Served (FCFS) Scheduling

- First-Come, First-Served (FCFS) or FIFO or “Run until done”
  - Used to mean one program scheduled until done (including I/O)
  - Now, means keep CPU until thread blocks
- Example:

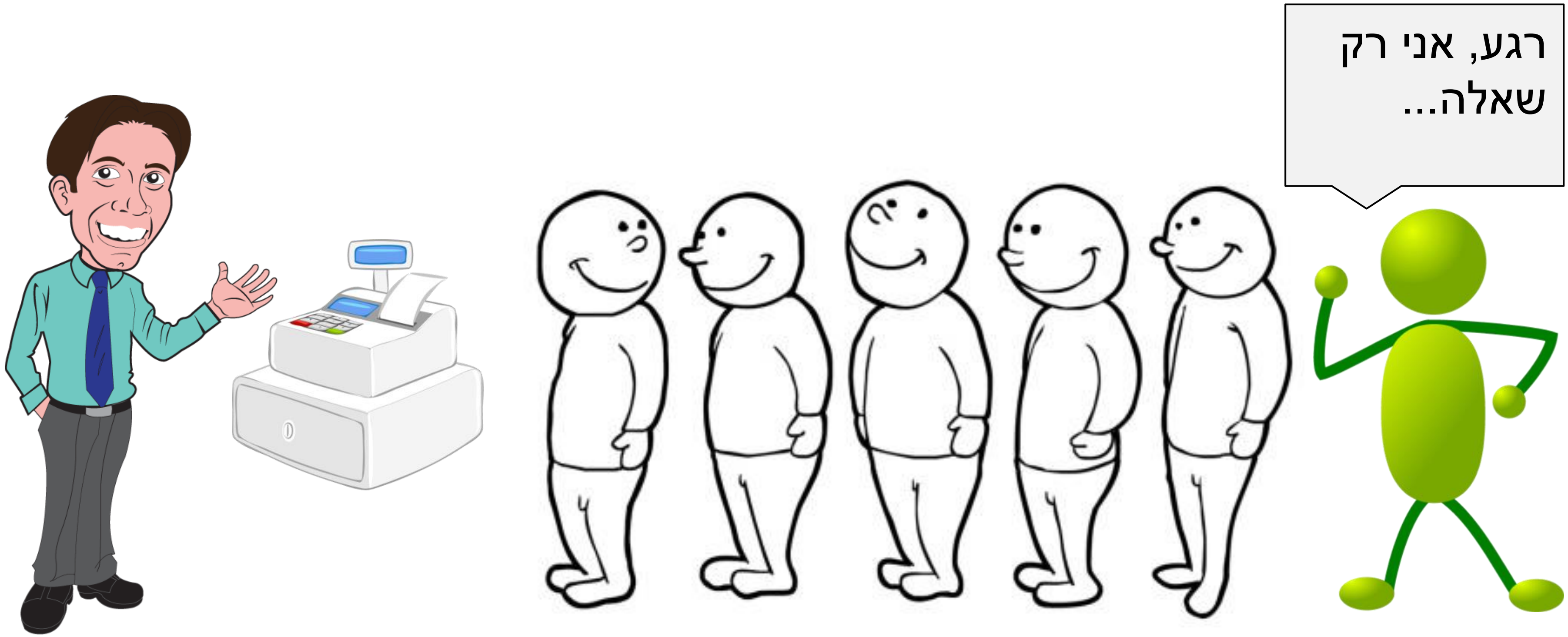
Arrival Order

Process	Burst Time
P_1	24
P_2	3
P_3	3



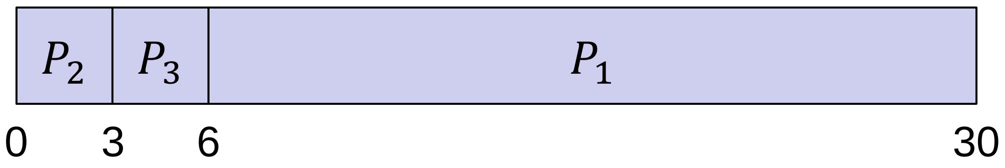


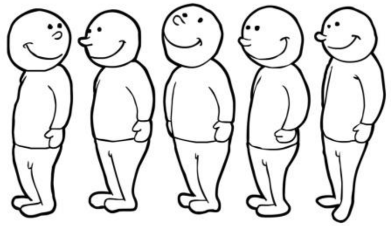
  - Waiting time for $P_1 = 0$; $P_2 = 24$; $P_3 = 27$
  - Average waiting time: $(0 + 24 + 27)/3 = 17$
  - Average Completion time: $(24 + 27 + 30)/3 = 27$
- *Convoy effect:* short process behind long process


רגע, אני רק
שאלה...

First-Come, First-Served (FCFS) Scheduling

- Suppose that processes arrive in order: P_2, P_3, P_1
  Schedule is:





  - Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
  - Average waiting time: $(6 + 0 + 3)/3 = 3$
  - Average Completion time: $(3 + 6 + 30)/3 = 13$
- In second case:
  - Average waiting time is much better (before it was 17)
  - Average completion time is better (before it was 27)
- FIFO Pros and Cons:
  - Simple ()
  - Short jobs get stuck behind long ones ()
    - Rami Levy: Getting humus, always stuck behind cart full of small items. Upside: get to catch up on email and Facebook!

Round Robin (RR)

- FCFS Scheme: Potentially bad for short jobs!
  - Depends on submit order
  - If you are first in line at supermarket with humus, you don’t care who is behind you, on the other hand…
- Round Robin Scheme
  - Each process gets a small unit of CPU time (*time quantum*), usually 10-100 milliseconds
  - After quantum expires, the process is preempted and added to the end of the ready queue.
  - n processes in ready queue and time quantum is $q \Rightarrow$
    - Each process gets $1/n$ of the CPU time in chunks of at most q time units
    - No process waits more than $(n-1)q$ time units

Round Robin (RR)

- Performance
  - q large $\Rightarrow$ FCFS
  - q small $\Rightarrow$ Interleaved (really small $\Rightarrow$ hyperthreading?)
  - q must be large with respect to context switch, otherwise overhead is too high (all overhead)

Round Robin Example

Process	Burst Time
P_1	53
P_2	8
P_3	68
P_4	24

P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 28 48 68 88 108 112 125 145 153

- Waiting times:
  - $P_1 = (68 - 20) + (112 - 88) = 72$
  - $P_2 = (20 - 0) = 20$
  - $P_3 = (28 - 0) + (88 - 48) + (125 - 108) = 85$
  - $P_4 = (48 - 0) + (108 - 68) = 88$
- Average waiting time = $(72 + 20 + 85 + 88)/4 = 66.25$
- Average completion time = $\frac{125+28+153+112}{4} = 104.5$

Round Robin

- Better for short jobs, Fair ()

- Context-switching time adds up for long jobs ()

Round-Robin Discussion

- How do you choose time slice?
  - What if too big? Response time suffers
  - What if infinite (∞)? Same as FCFS
  - What if time slice too small? Throughput suffers!

- Actual choices of time slice:
  - Initially, UNIX time slice was one second:
    - Worked ok when UNIX was used by one or two people.
    - What if three compilations going on? 3 seconds to echo a keystroke!
  - In practice, Need to balance short-job performance and long-job throughput:
    - Typical time slice today is between $10ms - 100ms$
    - Typical context-switching overhead is $0.1ms - 1ms$
    - Roughly 1% overhead due to context-switching

Comparing FCFS and RR

- Assuming zero-cost context-switching time, is RR always better than FCFS?
- Simple example:
  - 10 jobs, each take 100s of CPU time
  - RR scheduler quantum of 1s
  - All jobs arrive at the same time
- Completion Times:

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

Comparing FCFS and RR

Job #	FIFO	RR
1	100	991
2	200	992
…	…	…
9	900	999
10	1000	1000

- Both RR and FCFS finish at the same time
- Average response time is much worse under RR!
  - Bad when all jobs same length
- Also: Cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
  - Total time for RR longer even for zero-cost switch!

Earlier Example with Different Time Quanta

Best FCFS:

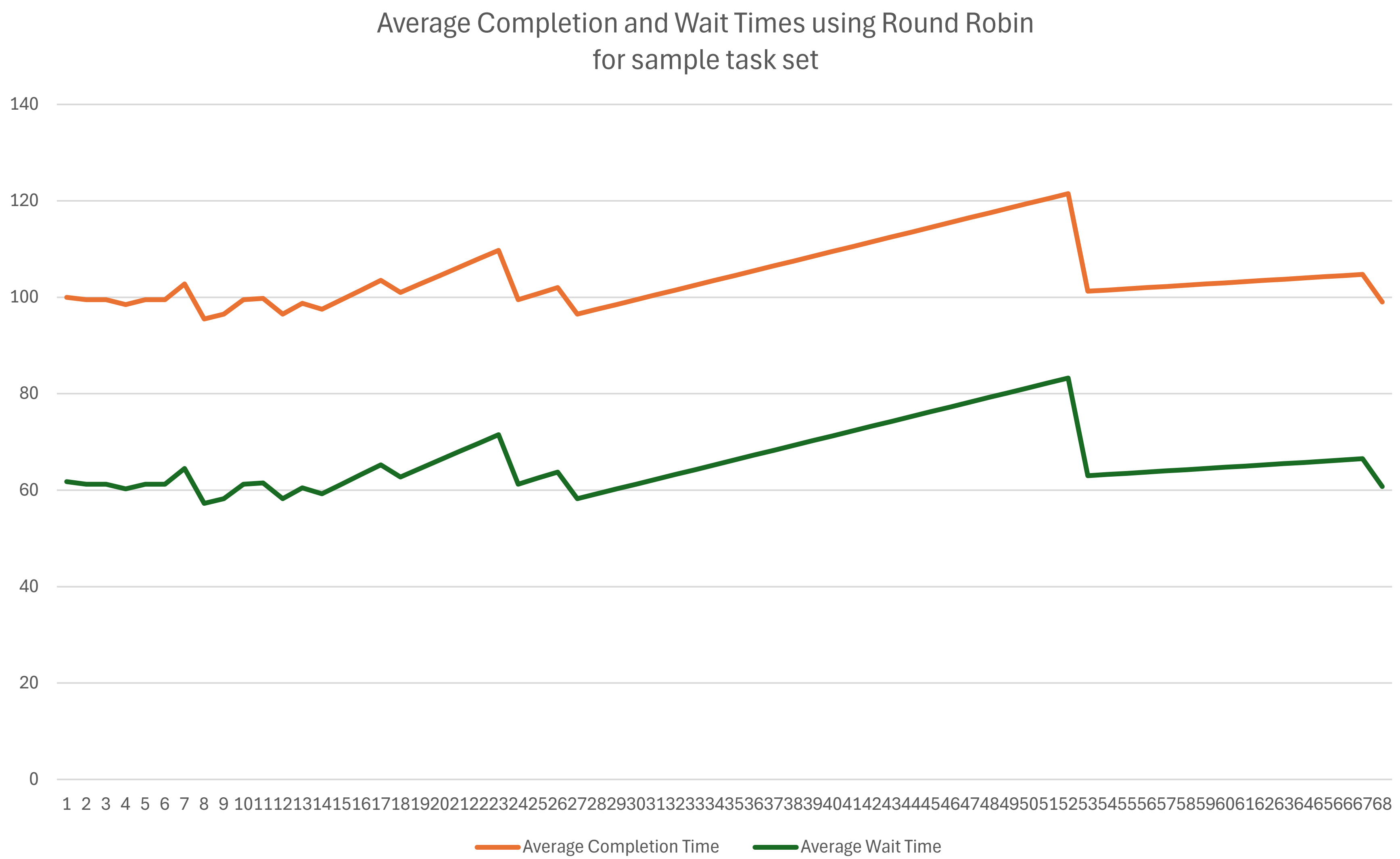
P_2 [8]	P_4 [24]	P_1 [53]	P_3 [68]

0 8 32 85 153

	Quantum	P_1	P_2	P_3	P_4	AVG
Wait Time	Best FCFS	32	0	85	8	31.25
	$Q = 1$	84	22	85	57	62
	$Q = 5$	82	20	85	58	61.25
	$Q = 8$	80	8	85	56	57.25
	$Q = 10$	82	10	85	68	61.25
	$Q = 20$	72	20	85	88	66.25
	Worst FCFS	68	145	0	121	83.5

	Quantum	P_1	P_2	P_3	P_4	AVG
Completion Time	Best FCFS	85	8	153	32	69.5
	$Q = 1$	137	30	153	81	100.25
	$Q = 5$	135	28	153	82	99.5
	$Q = 8$	133	16	153	80	95.5
	$Q = 10$	135	18	153	92	99.5
	$Q = 20$	125	28	153	112	104.5
	Worst FCFS	121	153	68	145	121.75

RR Quanta Graphed



Conclusion

- Higher Level Synchronization Atoms
  - Barrier Synchronization
  - Example: Readers and Writers
- Mutual Exclusion
  - Mutual Exclusion in High Level Language
- Scheduling
  - FIFO
  - Round Robin