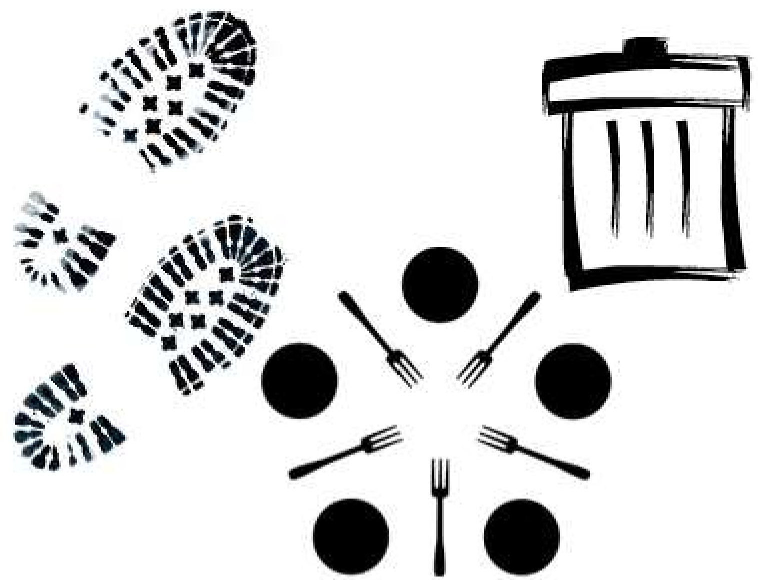
# Operating Systems

## Concurrency

## ENCE 360

# Outline

- Introduction
- Solutions
- Classic Problems



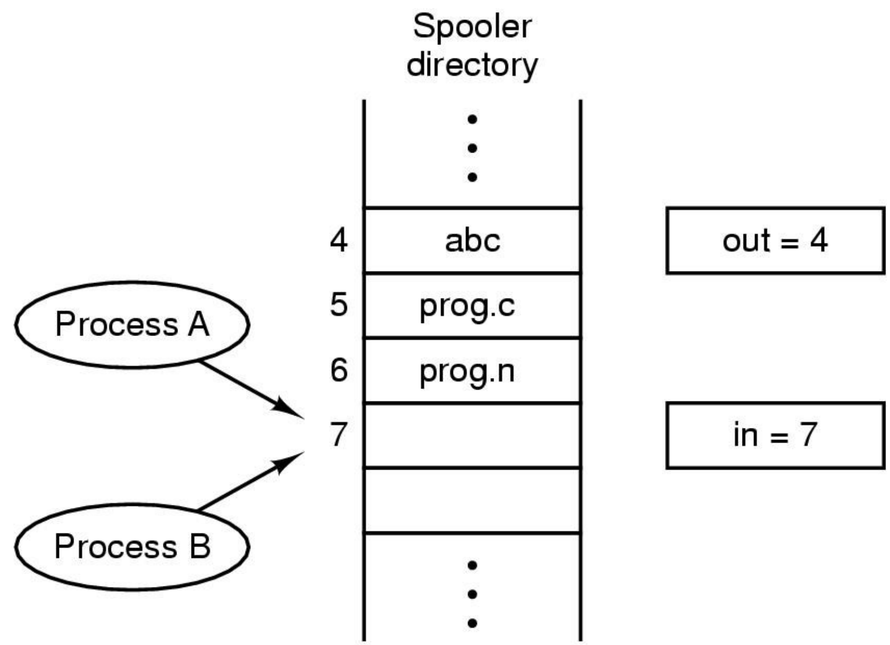
Chapter 2.3
MODERN OPERATING SYSTEMS (MOS)
*By Andrew Tanenbaum*

Chapter 26, 28, 31
OPERATING SYSTEMS: THREE EASY PIECES
*By Arpaci-Dusseau and Arpaci-Dusseau*

# A long time ago, …

- Remember day 1?
- Yes, single number, but what if bank account?
- What if print spooler?
- What if database?

```
68 prompt% threads-v0 100000
 Initial value: 0
 Final value: 200000
69 prompt% threads-v0 100000
 Initial value: 0
 Final value: 146796
```




| Thread 0 | Thread 1 | Thread 2 | Thread 3 |
|---|---|---|---|
| Paycheck | Buy fancy new TV | Roommate pays rent | Buying a video game |
| retrieve balance<br>add 450 to balance<br>store balance | retrieve balance<br>subtract 450 from balance<br>store balance | retrieve balance<br>add 300 to balance<br>store balance | retrieve balance<br>subtract 50 from balance<br>store balance |

# The Heart of the Problem

Display information from object file - machine instructions:

```
objdump --source thread-v0
```

Source code from “-g” flag

```
[line 415]
g_counter++;
400c38: 8b 05 6e 14 20 00  mov 0x201465,%eax   # 6020ac <g_counter>
400c3e: 83 c0 01           add $0x1,%eax
400c41: 89 05 65 14 20 00  mov %eax,0x201465   # 6020ac <g_counter>
```

Address | Object code | Assembly code | Reference location

Let’s zoom in …

# The Heart of the Problem (Zoom)

```
mov 0x20146e(%rip),%eax
add $0x1,%eax
mov %eax,0x201465(%rip)
```

```
mov g_counter %eax
add 1 %eax
mov %eax g_counter
```

"critical section"

Counter is 50. Thread T1 & T2, one processor. WCGW?

| | | | (after instruction) | | |
|---|---|---|---|---|---|
| **OS** | **Thread 1** | **Thread 2** | **PC** | **%eax** | **counter** |
| | *before critical section* | | 100 | 0 | 50 |
| | mov 0x8049a1c, %eax | | 105 | **50** | 50 |
| | add $0x1, %eax | | 108 | **51** | 50 |
| **interrupt** | | | | | |
| *save T1's state* | | | | | |
| *restore T2's state* | | | 100 | 0 | 50 |
| | | mov 0x8049a1c, %eax | 105 | **50** | 50 |
| | | add $0x1, %eax | 108 | **51** | 50 |
| | | mov %eax, 0x8049a1c | 113 | 51 | **51** |
| **interrupt** | | | | | |
| *save T2's state* | | | | | |
| *restore T1's state* | | | 108 | 51 | 51 |
| | mov %eax, 0x8049a1c | | 113 | 51 | **51** |

"race condition"

Not 52!

# The Heart of the Problem – 3 not 1

```
mov g_counter %eax
add 1 %eax
mov %eax g_counter
```

- 3 operations instead of 1. What if had:

```
memory-add 0x201465 1
```

- Atomic action – can't be interrupted

  → Seems simple. Problem solved!

- But … what if wanted to "subtract 1", or "add 10", or "atomic update of B-tree"
  - Won't be atomic instructions for everything!

# The Heart of the Solution

- Instead, provide synchronization primitives

→ Programmer can use for atomicity (and more)

THE CRUX OF THE PROBLEM:
HOW TO PROVIDE SUPPORT FOR SYNCHRONIZATION?

What synchronization primitives should be provided?
What support needed from hardware to build?
How to make correct and efficient?
How do programmers use them?

# Useful Terms*

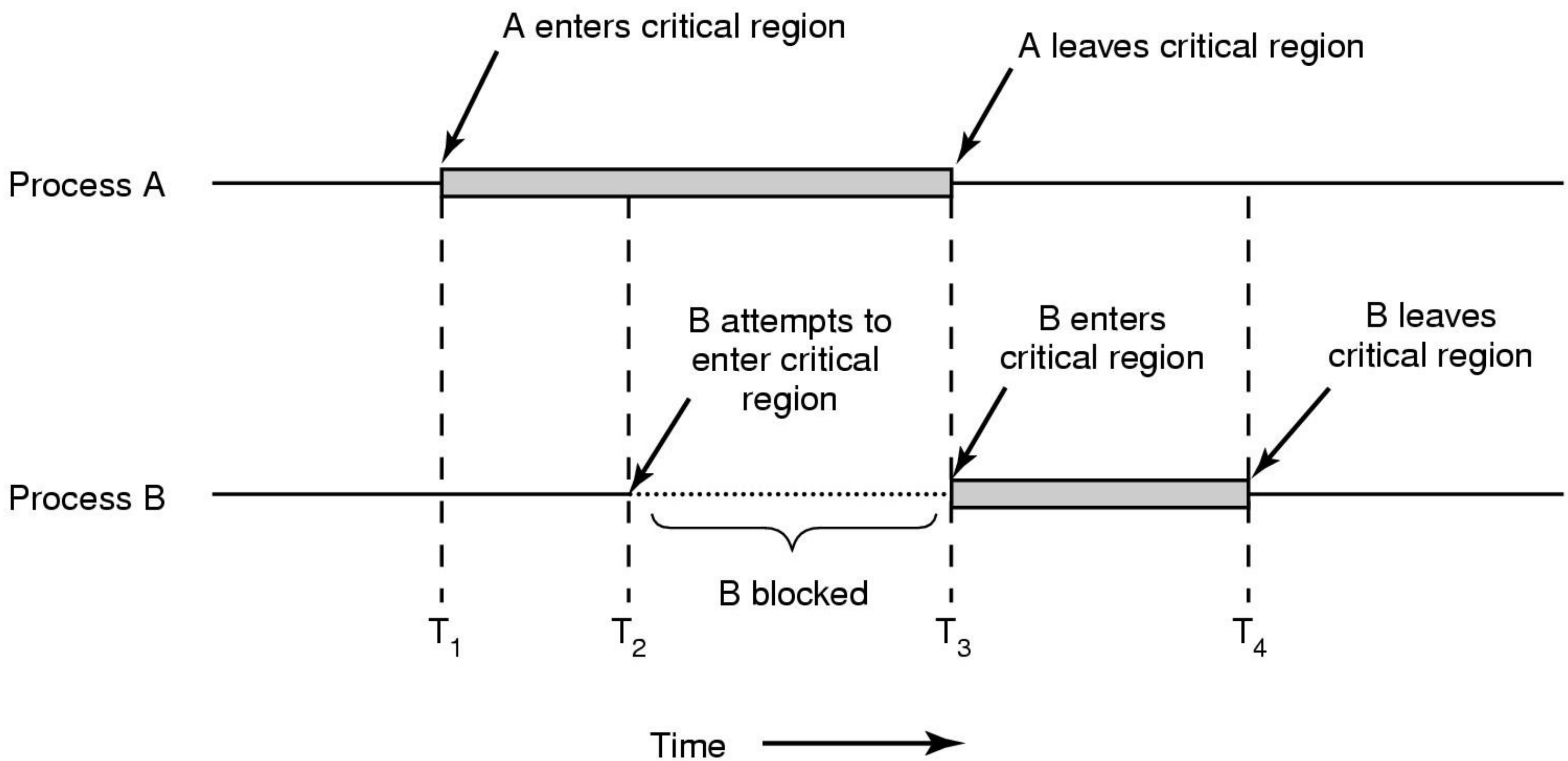
- Critical section – code that access shared resource (e.g., variable or data structure)
- Race condition – arises when multiple threads/processes simultaneously enter critical section leading to non-deterministic outcome
- Indeterminant program – program with 1+ race conditions, so output varies run to run
- Mutual exclusion – method to guarantee only 1 thread/process active in critical section at a time

* That all good systems-programmers (you!) should know

# Outline

- Introduction (done)
- Solutions (next)
- Classic Problems

# Illustration of Critical Region




What *basic* mechanism can stop B from entering critical region when A in?
Hint: just need to block access

# How to Use a Lock

```
lock_t mutex;          // globally-allocated 'mutex'
…
lock(&mutex);
x = x + 1;      // critical region
unlock(&mutex);
```

See: "thread-v1.c"

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or general CR
pthread_mutex_unlock(&lock);
```

THE CRUX: HOW TO BUILD A LOCK?

How to build efficient lock?

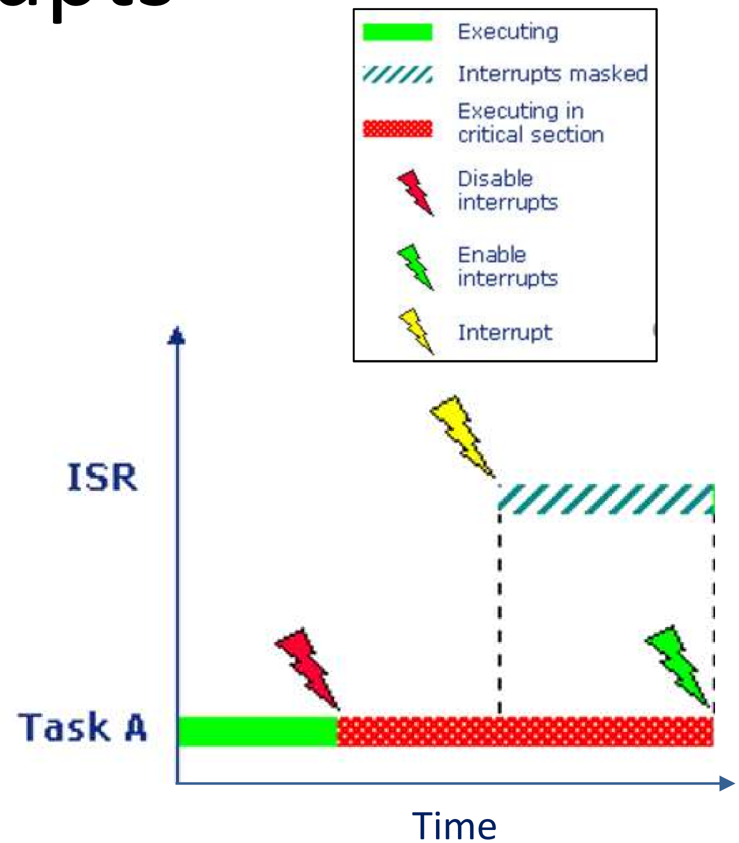
What hardware support is needed?

What OS support?

# Simple Lock Implementation - Disable Interrupts

- If no interrupts, no race condition

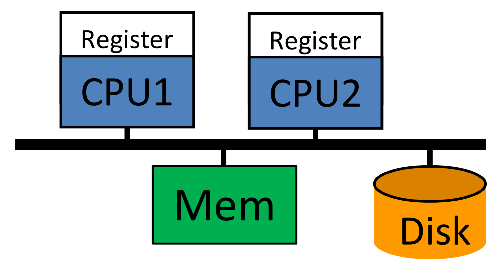
```
void lock() {
   DisableInterrupts();
}
void unlock() {
   EnableInterrupts();
}
```




What is the potential problem?
Hint: consider all sorts of user programs

# Many Problems with Disabling Interrupts in General

- Privileged operations, so must trust user code
  - But may never unlock! (unintentional or malicious)
- Does not work for multiprocessors
  - Second processor may still access shared resource




- When interrupts off, subsequent ones may become lost
  - E.g., disk operations

# Lock Solution, Take 2

```
int mutex; // 0 -> lock available, 1 -> held

void lock(int *mutex) {
  while (*mutex == 1) // TEST flag
     ; // spin-wait (do nothing)
  *mutex = 1; // now SET it!
 }

void unlock(int *mutex) {
  *mutex = 0;
}
```

This almost works … but not quite. Why not?

Hint, has race condition - Can you spot it?

# Lock Solution, Take 2

```
int mutex; // 0 -> lock available, 1 -> held

void lock(int *mutex) {
  while (*mutex == 1) // TEST flag
     ; // spin-wait (do nothing)
  *mutex = 1; // now SET it!
 }

void unlock(int *mutex) {
  *mutex = 0;
}
```

This almost works ... not quite...

If can TEST mutex and SET it in *atomic* operation, would be ok

But ... aren't back to square 1?

No! Only need hardware support for 1 operation → build lock primitive

# Synchronization Hardware – Test and Set

Test-and-Set: returns and modifies *atomically*

```
int TestAndSet(int *mutex) {
  int temp;
  temp = *mutex;
  *mutex = true;
  return temp;
}
```

Done with hardware support.
All modern computers since 1960's
e.g., x86 has compare-and-exchange
Others: compare-and-swap, fetch-and-add, … all *atomic*

# Lock Solution, Take 3

```
int mutex; // 0 -> lock available, 1 -> held

void lock(int *mutex) {
  while (TestAndSet(mutex)) // 1 if held
      ; // spin-wait (do nothing)
  // once here, have lock!
 }

void unlock(int *mutex) {
  *mutex = 0;
}
```

Note, no need to protect `unlock()`
(Exercise: why not?)

Now, what is major remaining shortcoming?
Hint: code works, but could be more efficient

# Lock Solution, Take 4

```
int mutex; // 0 -> lock available, 1 -> held

void lock(int *mutex) {
  while (TestAndSet(mutex)) {
    queueAdd(*mutex);
    park(); // put process to sleep
  }
 }

void unlock(int *mutex) {
  *mutex = 0;
  if (!queueEmpty(*mutex))
    unpark(); // wake up process
}
```

Note: almost right, but need to protect queue, too (see OSTEP, 28.14 for final touch)

# Synchronization Primitive - Semaphore

- “Special” integer, provided by OS
- Only accessible through two routines:
  `sem_post()`
  `sem_wait()`
- Both routines are *atomic*

```
int sem_wait(sem_t &s) {
  s = s - 1
  if (s < 0)
    add process to queue and sleep
}

int sem_post(sem_t &s) {
  s = s + 1
  if (s <= 0)
    remove process from queue and wake
}
```

<u>Operational Model</u>

value of counter = number of procs that may pass before closed
counter <= 0 → gate closed!
blocked process "waits" in Q
counter < 0 → number of processes waiting in Q

# How to Use a Semaphore

```
semaphore mutex; // globally-allocated
…
wait(&mutex);
x = x + 1;   // critical region
signal(&mutex);
```

Easy, peasy!
And available on most operating systems
Can use for general synchronization problems (next)

# SOS: Semaphore

See: "`semaphore.c`"

- How does the OS protect access to the semaphore integer count?
  - Previously said this was a bad idea … why is it ok in this context?
  - How else might the OS protect this critical region?
- Challenge: Implement "attach" and "detach" functions

Design Technique

Reducing Problem to Special Case

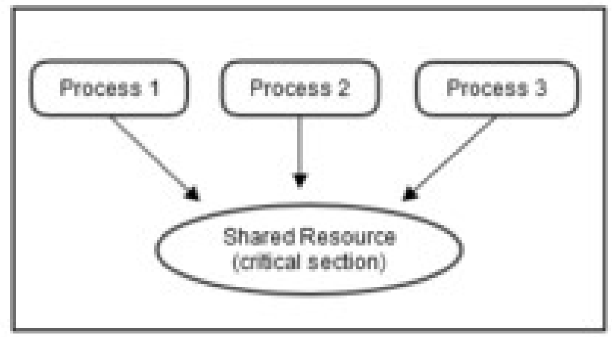
Other examples:
name servers, on-line help

```
/* Attach to OS semaphore */
int AttachSemaphore(int key);

/* Deattach from sem id */
int DetachSemaphore(int sid);
```

# Other Synchronization Primitives

- Monitors
- Condition Variables
- Events
- …
- Execise: learn on own



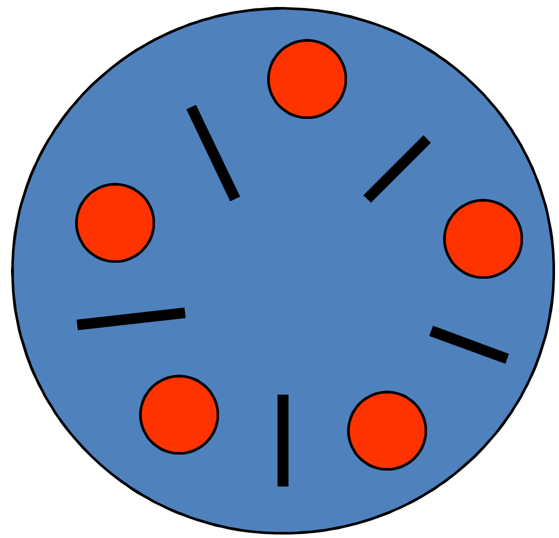

- Fortunately, if have one (e.g,. Lock) can build others

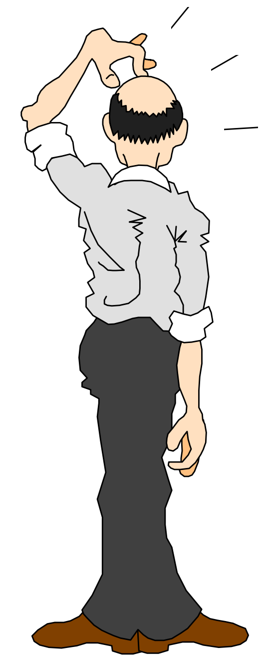
# Outline

- Introduction (done)
- Solutions (done)
- Classic Problems (next)
  - Dining Philosophers
  - Readers-Writiers

# Dining Philosophers

- Philosophers
  - Think
  - Sit
  - Eat
  - Think
- Need 2 chopsticks to eat





# Dining Philosophers

Philosopher i:

For 5 Philosophers

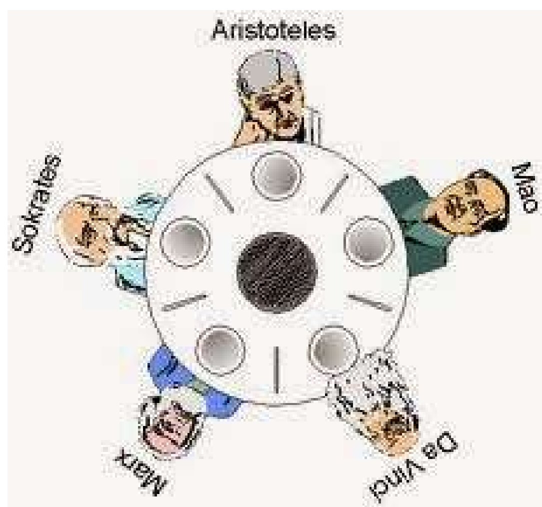
```
while (1) {
  /* think… */
  wait(chopstick[i]);
  wait(chopstick[i+1 % 5]);
  /* eat */
  signal(chopstick[i]);
  signal(chopstick[i+1 % 5]);
}
```

This almost works, but not quite. Why not?

Solutions?

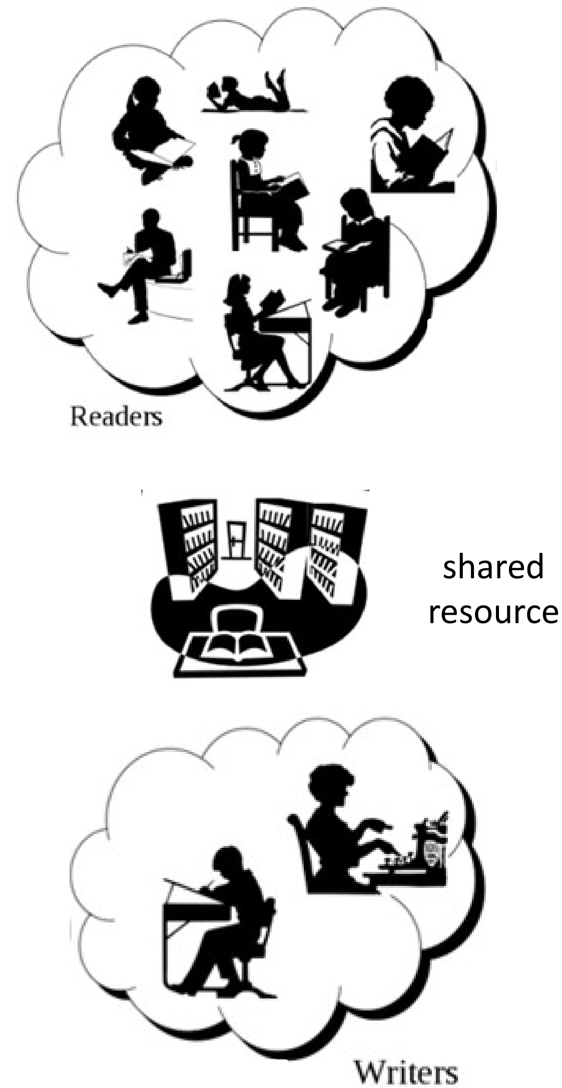
# Dining Philosopher Solutions

- Allow at most N-1 to sit at a time
- Allow to pick up chopsticks only if both are available
- Asymmetric solution (odd L-R, even R-L)



# Readers-Writers

- *Readers* only read the content of object
- *Writers* read and write the object
- Critical region, one of:
    1. No processes
    2. One or more readers (no writers)
    3. One writer (nothing else)



# Readers-Writers

Shared:

```
semaphore mutex;
semaphore wrt;
int readcount;
```

Writer:

```
wait(wrt)
/* write stuff */
signal(wrt);
```

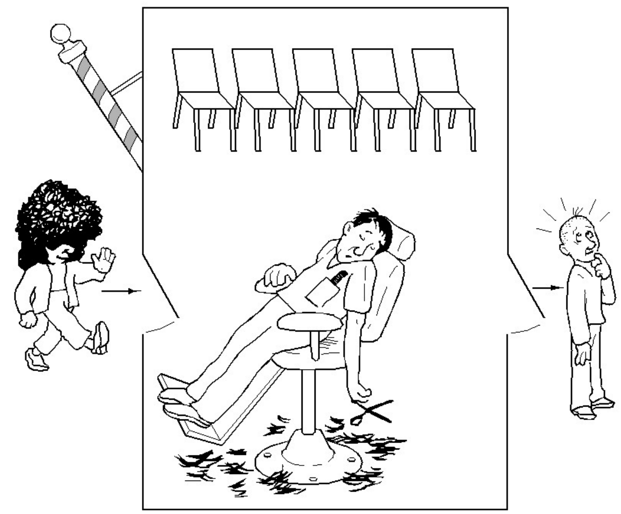
Reader:

```
wait(mutex);
readcount = readcount + 1;
if (readcount==1)
  wait(wrt);
signal(mutex);
/* read stuff */
wait(mutex);
readcount = readcount - 1;
if (readcount==0)
   signal(wrt);
signal(mutex);
```

Solution “favors” readers.
Can you see why?

# Other Classic Problems

- Bounded Buffer
- Sleeping Barber
- Bakery Algorithm
- Cigarette smokers
- …



- If can model your problem as one of the above → Solution
- Akin to *Software Design Patterns*

# Outline

- Introduction (done)
- Solutions (done)
- Classic Problems (done)