MOVIE SCIENCE MONTAGE
ACTUAL SCIENCE MONTAGE
SQUEAK!
...WHIRRRRRR...
...WHIRRRR... BING!
PAINT FLECKS FROM THE KILLER'S CLOTHING MATCH AN ANTIMATTER FACTORY IN BELGRADE!
LET'S GO!
OKAY, WE'VE DETERMINED THERE'S NEITHER BARIUM NOR RADIUM IN THIS SAMPLE.
PROBABLY.

Start of Lecture: January 25, 2014

Reminders

- Assignment 1 is due this Friday at 6:00 p.m.
- Couple comments about Exercise 1:
  - Thought questions: be honest and sincere about questions you have while reading; I won't judge you. If you legitimately have the question, its a good one. Some questions seemed almost like textbook questions
  - Clarity is important: I can only mark what is in front of me.
- Examples of some interesting thought questions
  - What are the more basic/essential services that an OS should offer to be considered an OS?
  - Would it be a good approach for scheduling for the operating system to give CPU priority to the currently active/selected process?

Processes and Threads Review

- http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/dynamicprocess.htm
- POSIX threads (Pthreads): https://computing.llnl.gov/tutorials/pthreads/
- Look at code examples (particularly mem.c, syscalls.c, sig1.c, sig2.c, fork1.c, fork2.c, pthread_join.c)

UNIVERSITY OF ALBERTA

Lecture 5: Synchronization

CMPUT 379, Section A1, Winter 2014
January 29 and 31

Objectives

- Introduce the critical-section problem, whose solutions can be used to ensure consistency of shared data
- Produce both software and hardware solutions to the critical-section problem
- Examine some classical process synchronization problems and solutions

Concurrency and Parallelism

Motivation: overlap computation with I/O; simplify programming.

- **hardware parallelism**: CPU computing, one or more I/O devices are running at the same time.
- **pseudo parallelism**: rapid switching back and forth of the CPU among processes, pretending that those processes run concurrently.
- **real parallelism**: can only be achieved by multiple CPUs. Single CPU systems cannot achieve real parallelism.
- Unfortunately, keeping track of multiple activities is **difficult**.

Concurrent Processes

- In a multiprogramming environment, processes executing concurrently are either *competing* for the CPU/other global resources, or *cooperating* with each other to share resources.
- An OS deals with **competing processes** by carefully allocating resources and properly isolating processes from each other.
- For **cooperating processes**, the OS provides mechanisms to share some resources in certain ways as well as allowing processes to properly interact with each other.
- Cooperation is either by **implicit sharing (shared memory)** or by **explicit communication (message passing)**.

Processes Competing

- Processes that do not exchange information cannot affect each other's execution, but can compete for resources; can be unaware of one another
- **Example**: independent processes — text editor and 'ps'
- **Properties:**
  - Deterministic
  - Reproducible
  - Can stop and restart without side effects
  - Can proceed at arbitrary rate

Processes cooperating

- Processes that are aware of each other, and directly (by message passing) or indirectly (by shared memory) work together; **may** affect the execution of each other.
- **Example**: Transaction processes in airline reservations.
- **Properties:**
  - Share memory or exchange messages
  - Non-deterministic (a problem!)
  - May be irreproducible (a problem!)
  - Subject to race conditions (a problem!)

Why do we want process cooperation?

We allow processes to cooperate with each other to:

- share some resources
  - One checking account file, many tellers.
- do things faster
  - Read next block while processing current one.
  - Divide jobs into smaller pieces and execute them concurrently.
- construct systems in a modular fashion
  - UNIX example: cat names | sort | uniq -c

A potential problem

- Instructions of cooperating processes might be ***interleaved arbitrarily***. Order of (some) instructions are irrelevant, but other instruction combinations must be eliminated e.g

Process A	**Process B**	**concurrent access**
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

- A ***race condition*** is when two or more processes access shared data concurrently **and** correctness depends on specific interleaving of operations (i.e. good luck).

Why do instructions get interleaved?

- Kernel constantly context-switching between processes, not necessarily paying attention to when a process is **pre-empted** (process is paused before completion)
- First 10 instructions of a process may complete, then it is paused, added to ready queue, and next process is set to run. Then, after some number of instructions complete for that process (say 15), it will be pre-empted and another process will become the active process.
- A process rarely finishes completion without being pre-empted; it is difficult to guess the number of instructions executed and the order the processes will execute

Example: Producer and Consumer

Producer —>

```
while (true) {
     /* produce an item in next produced */

     while (counter == BUFFER_SIZE) ;
          /* do nothing */
     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

Consumer —>

```
while (true) {
     while (counter == 0)
          ; /* do nothing */
     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;
     /* consume the item in next consumed */
}
```

Race condition: modify counter concurrently

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

```
S0: producer execute register1 = counter           {register1 = 5}
S1: producer execute register1 = register1 + 1     {register1 = 6}
S2: consumer execute register2 = counter           {register2 = 5}
S3: consumer execute register2 = register2 – 1     {register2 = 4}
S4: producer execute counter = register1           {counter = 6 }
S5: consumer execute counter = register2           {counter = 4}
```

So how do we avoid race conditions?

- We have to ensure that the correctness of the code does not rely on how the operations are interleaved
- So how do we do that with N producers and M consumers (say for servers and clients)?
- We'll get there, but let's start with something simpler

A simpler example: buying milk

- What does correct mean here? Someone gets milk but NOT everyone (because thats too much milk!)

time	**Person A**	**Person B**
3:00	Look in fridge. *Out of milk*.	
3:05	Leave for store.	
3:10	Arrive at store.	Look in fridge. *Out of milk*.
3:15	Buy milk.	Leave for store.
3:20	Leave the store.	Arrive at store.
3:25	Arrive home, put milk away.	Buy milk.
3:30		Leave the store.
3:35		Arrive home. ***OH! OH!***

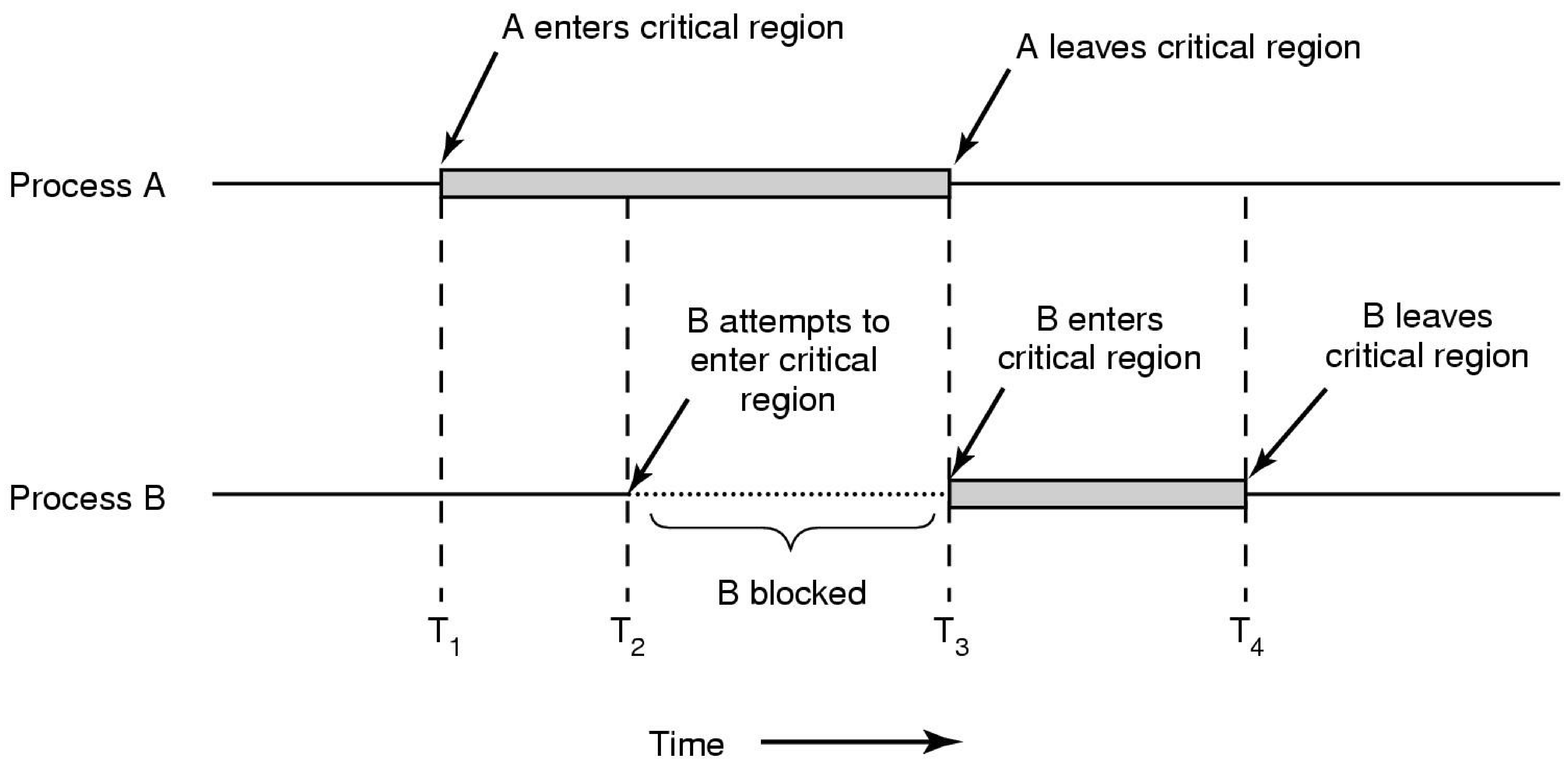
Mutual exclusion

- The "too much milk" example shows that when cooperating processes are not synchronized, they may face unexpected "timing" errors.
- **Mutual exclusion** is a requirement that only one process (or person) is doing certain things at one time, thus avoiding data inconsistency. All others should be prevented from modifying shared data (i.e., the fridge or milk) until the current process finishes.
- e.g., only one person *buys milk* at a time.

Critical section

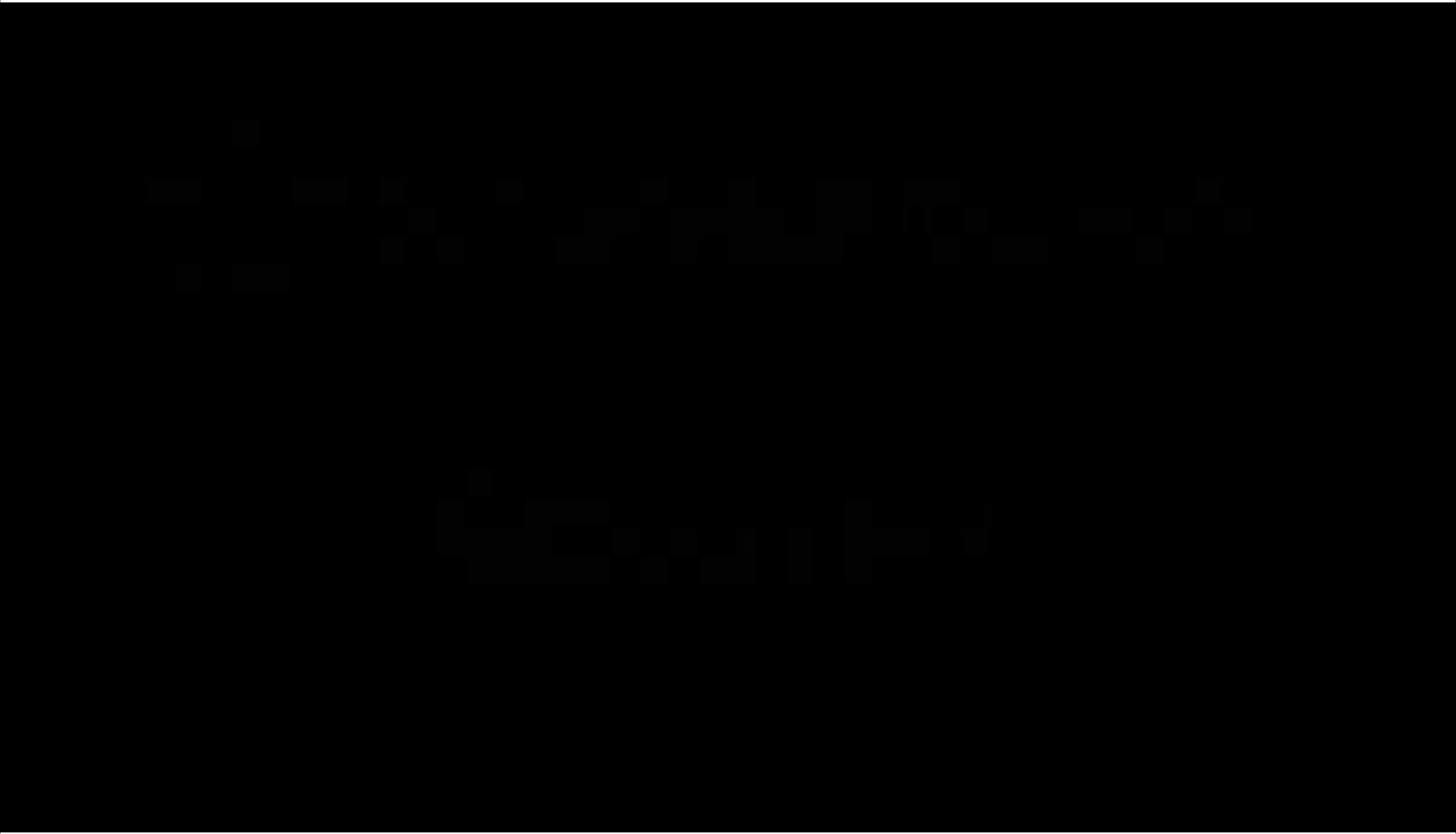
- A **section of code**, or a **collection of operations**, in which only one process may be executing at a given time and which we want to make "sort of" atomic.
- **Atomic** means either an operation happens in its entirety or *not at all*; i.e., it cannot be interrupted in the middle.
- Atomic operations are used to ensure that cooperating processes execute correctly. e.g. variable read and modified at same time (so not using inconsistent value)
- **Mutual exclusion mechanisms** are used to solve the **critical section** problem.

Critical sections (aka Critical regions)



Mutual exclusion using critical regions

Video Break: Animals are cool



First attempt at computerized milk buying

- This solution works for people because we assume the first three lines are performed **atomically**
- But, if the first three lines not executed atomically, this solution does not work!
- What happens if A executes first two lines, then is paused and then B executes first two lines and either continues from there?

Processes A & B

```
if ( NoMilk ) {
    if ( NoNote ) {
        Leave Note;
        Buy Milk;
        Remove Note;
    }
}
```

Second attempt at computerized milk buying

- Second attempt: use two notes
- What can you say about this solution? Any issues?

Process A

```
Leave NoteA;
if ( NoNoteB ) {
    if ( NoMilk ) {
        Buy Milk;
    }
}
Remove NoteA;
```

Process B

```
Leave NoteB;
if ( NoNoteA ) {
    if ( NoMilk ) {
        Buy Milk;
    }
}
Remove NoteB;
```

Exercise: Synchronize two processes

- Try to fix the below two-process milk-buying implementation
- Don’t worry about blocking vs busy waiting; you can go ahead and use busy waiting if you want
- Do you think you could extend your solution to more than two processes?

Process A

```
Leave NoteA;
if ( NoNoteB ) {
    if ( NoMilk ) {
        Buy Milk;
    }
}
Remove NoteA;
```

Process B

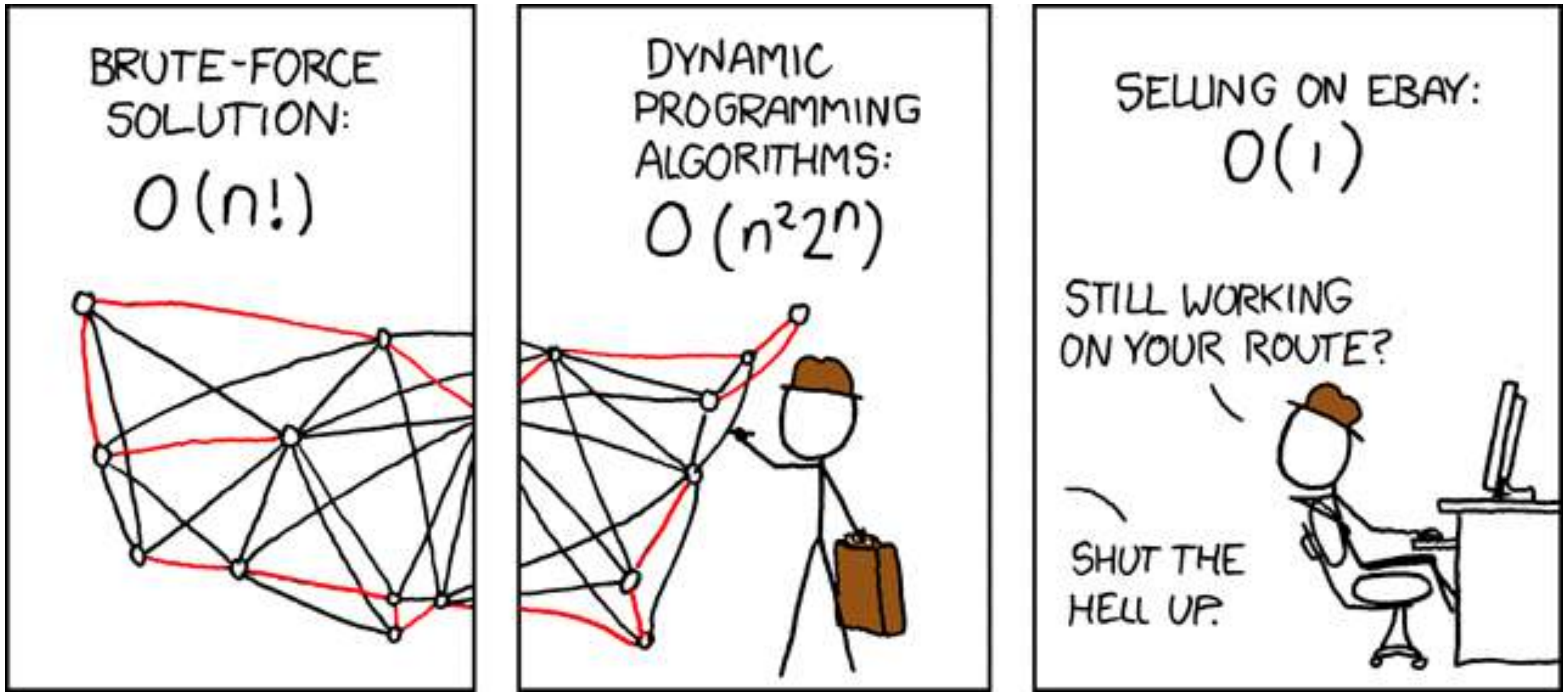
```
Leave NoteB;
if ( NoNoteA ) {
    if ( NoMilk ) {
        Buy Milk;
    }
}
Remove NoteB;
```

Possible solution for two processes

- I’m not including this online yet, because I want you to try to solve it

Asymmetric mutual exclusion using notes is a solution that does not scale

- Asymmetric solution to buying-milk problem is too complicated: difficult to trace and ensure it is correct
- Process B is busy waiting: consuming CPU cycles while waiting. B can be pre-empted, but when running, its pure waste.
- The solution complicated to extend to many processes



There are many many solutions/approaches

- The number of (classical) solutions to the critical section problem can feel overwhelming
  - Dekker's algorithm, Peterson's solution, syntonization hardware (giving atomic operations), mutex locks, semaphores, monitors, and others
- I don't expect you to know all the algorithms, but rather to understand the issues when implementing synchronization solutions, and recognize/find correct solutions
- Whatever your solution (and whether it has a name, or is specific to your scenario), it must meet certain requirements

Fundamental requirements

- **Mutual exclusion** — if a process is executing in its critical section, then no other processes can be executing in their critical sections
- **Progress** — process can make progress and complete execution (avoid deadlock and livelock)
- **Bounded waiting** — limited number of times that process has to wait to enter its CS, i.e. bound on number of times other processes are allowed to enter CS instead of allowing this process to enter CS

Critical section within OS code

- How does the operating system deal with critical sections within its own code? Ensure correctness?
- Two approaches depending on kernel type
  - Preemptive kernel — allows preemption of process when running in kernel mode
  - Non-preemptive kernel — runs until exits kernel mode, blocks, or voluntarily yields the CPU.
- Non-preemptive kernels are essentially free of race conditions *in kernel mode* (e.g. system calls); in user space, the user programmer has to ensure that they take care of their critical sections