Principles of Operating Systems

CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. Concurrency

- ✓ Types of process interaction
- ✓ Race conditions & critical regions
- ✓ Mutual exclusion by busy waiting
- ✓ Mutual exclusion & synchronization
  - mutexes
  - semaphores
  - monitors
  - message passing

d. Deadlocks

2.c Concurrency

Types of process interaction

- Concurrency refers to any form of interaction among processes or threads
  - ✓ concurrency is a fundamental part of O/S design
  - ✓ concurrency includes
    - communication among processes/threads
    - sharing of, and competition for system resources
    - cooperative processing of shared data
    - synchronization of process/thread activities
    - organized CPU scheduling
    - solving deadlock and starvation problems

2.c Concurrency

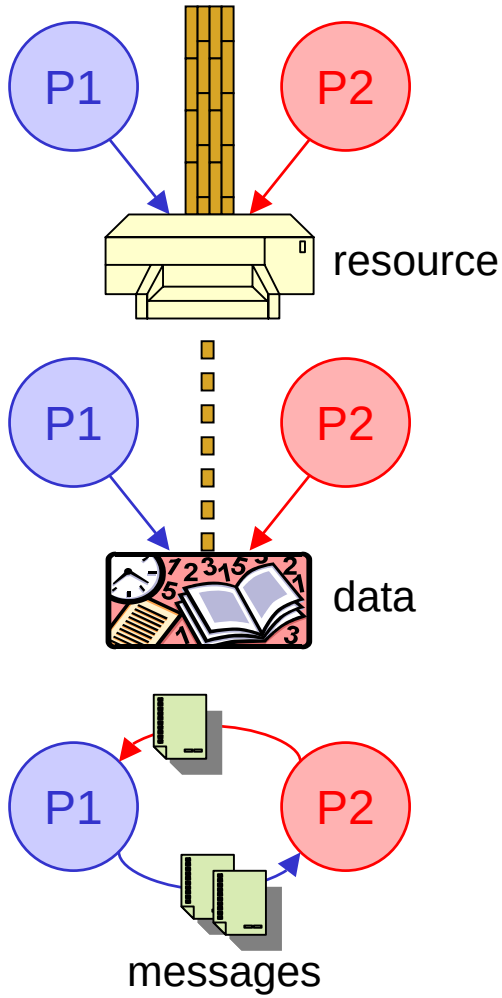
Types of process interaction

- Concurrency arises in the same way at different levels of execution streams
  - ✓ **multiprogramming** — interaction between multiple processes running on one CPU (pseudoparallelism)
  - ✓ **multithreading** — interaction between multiple threads running in one process
  - ✓ **multiprocessors** — interaction between multiple CPUs running multiple processes/threads (real parallelism)
  - ✓ **multicomputers** — interaction between multiple computers running distributed processes/threads
  - → *the principles of concurrency are basically the same in all of these categories (possible differences will be pointed out)*

2.c Concurrency

Types of process interaction

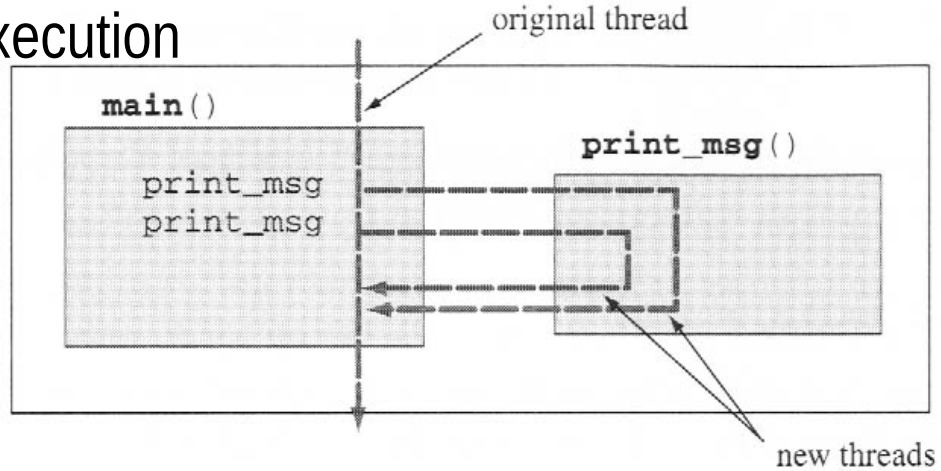
- Whether processes or threads: three basic interactions
  - ✓ processes unaware of each other — they must use shared resources independently, without interfering, and leave them intact for the others
  - ✓ processes indirectly aware of each other — they work on common data and build some result together via the data ("stigmergy" in biology)
  - ✓ processes directly aware of each other — they cooperate by communicating, e.g., exchanging messages



2.c Concurrency

Race conditions & critical regions

- Inconsequential race condition in the shopping scenario
  - ✓ there is a “race condition” if the outcome depends on the order of the execution



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

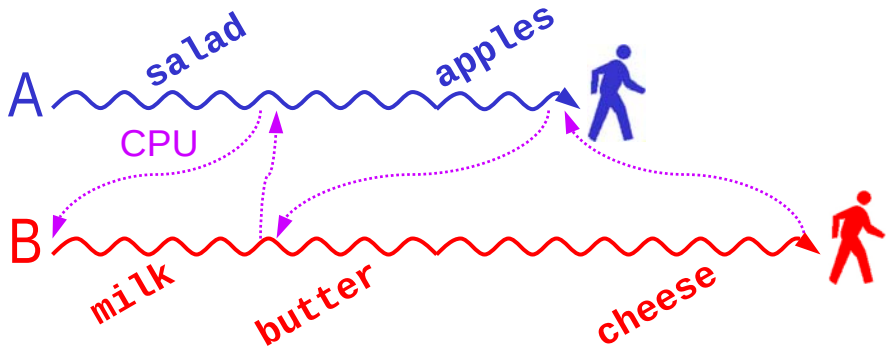
Multithreaded shopping diagram and possible outputs

2.c Concurrency

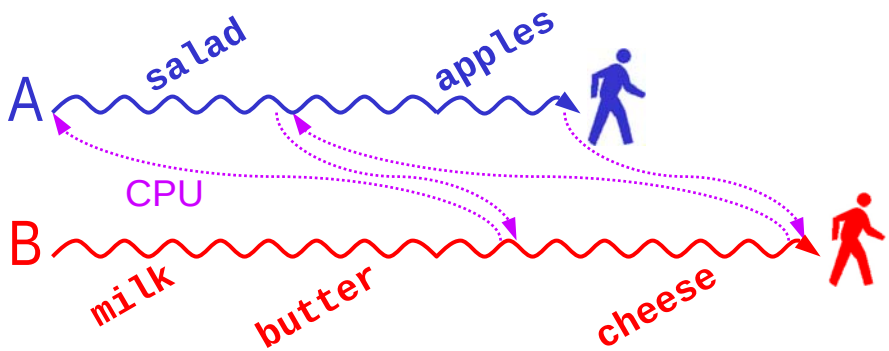
Race conditions & critical regions

- Inconsequential race condition in the shopping scenario
  - ✓ the outcome depends on the CPU scheduling or “interleaving” of the threads (separately, each thread always does the same thing)

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```




```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

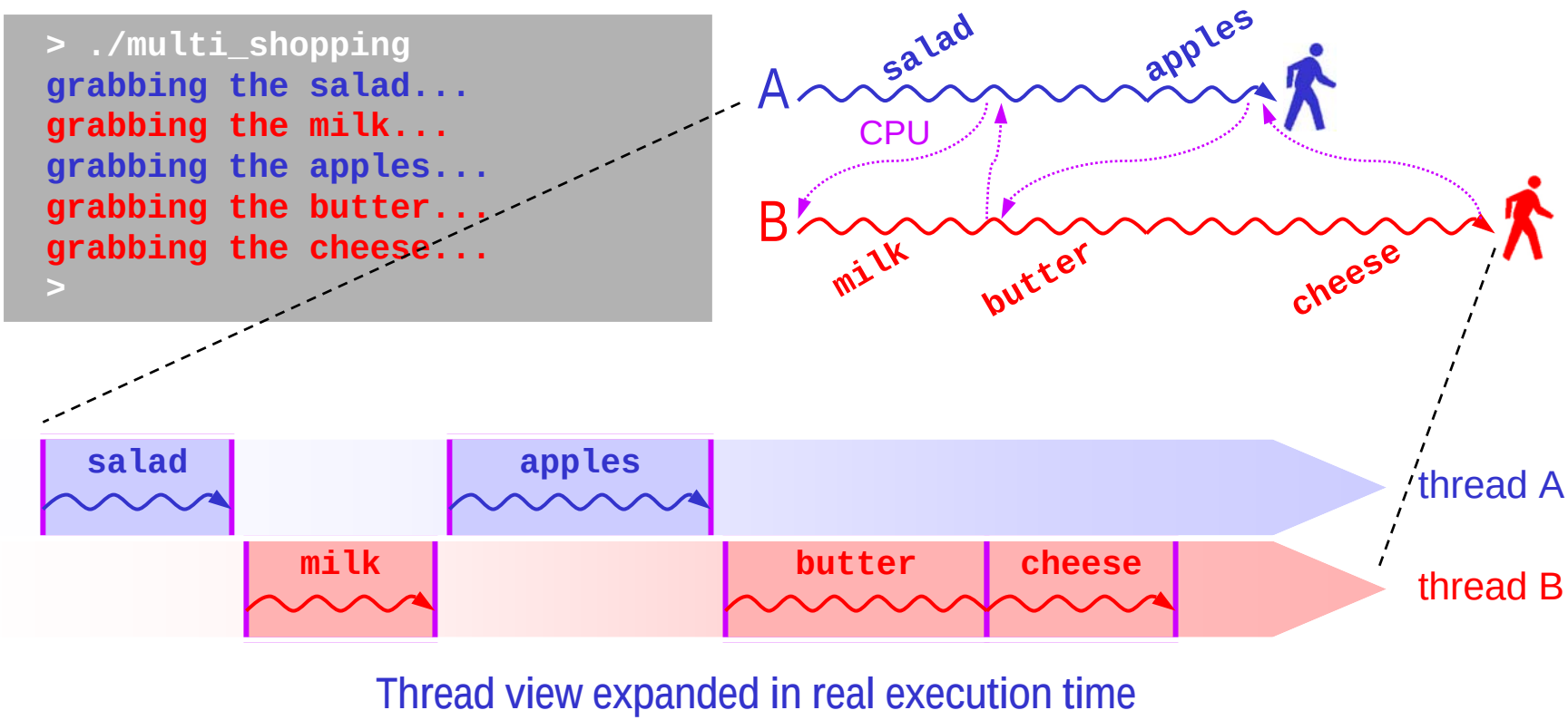


2.c Concurrency

Race conditions & critical regions

- Inconsequential race condition in the shopping scenario
  - ✓ the CPU switches from one process/thread to another, possibly on the basis of a preemptive clock mechanism

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

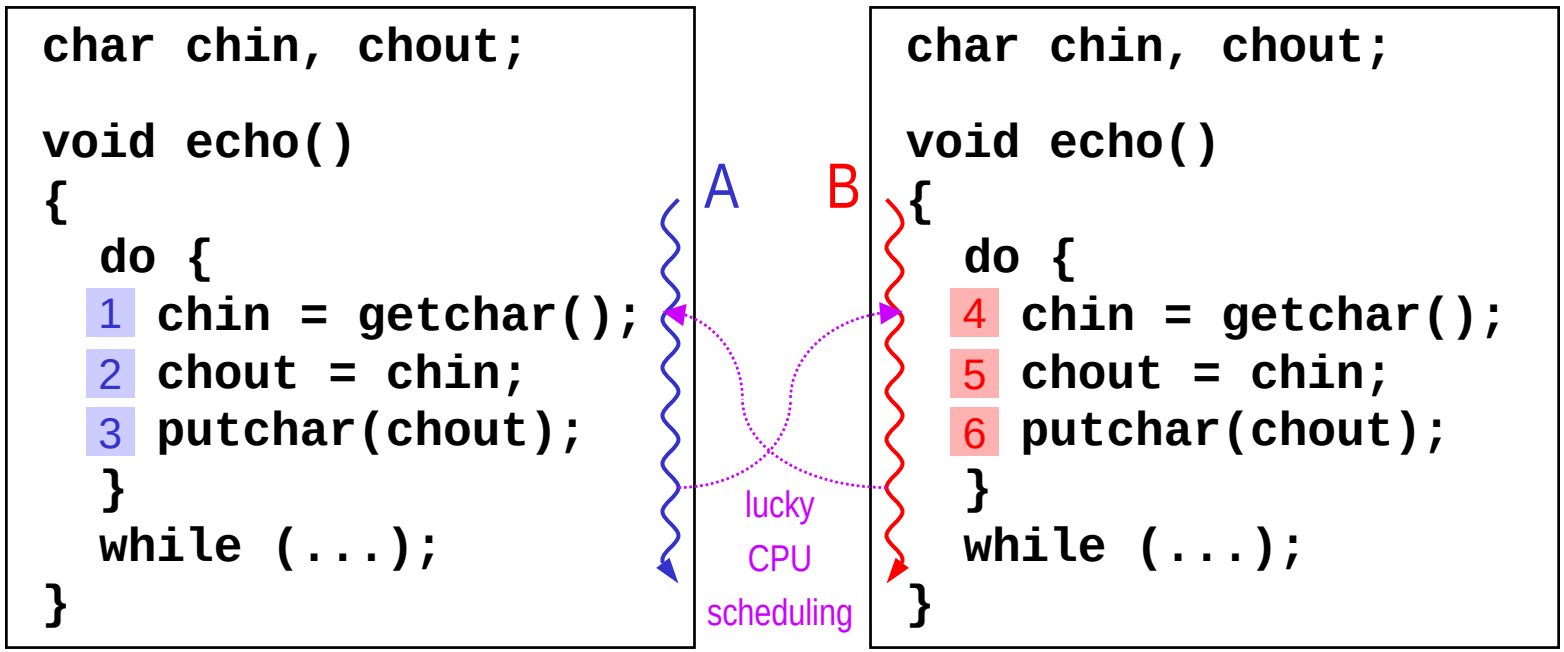


Thread view expanded in real execution time

2.c Concurrency

Race conditions & critical regions

- Consequential race conditions in I/O & variable sharing




```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

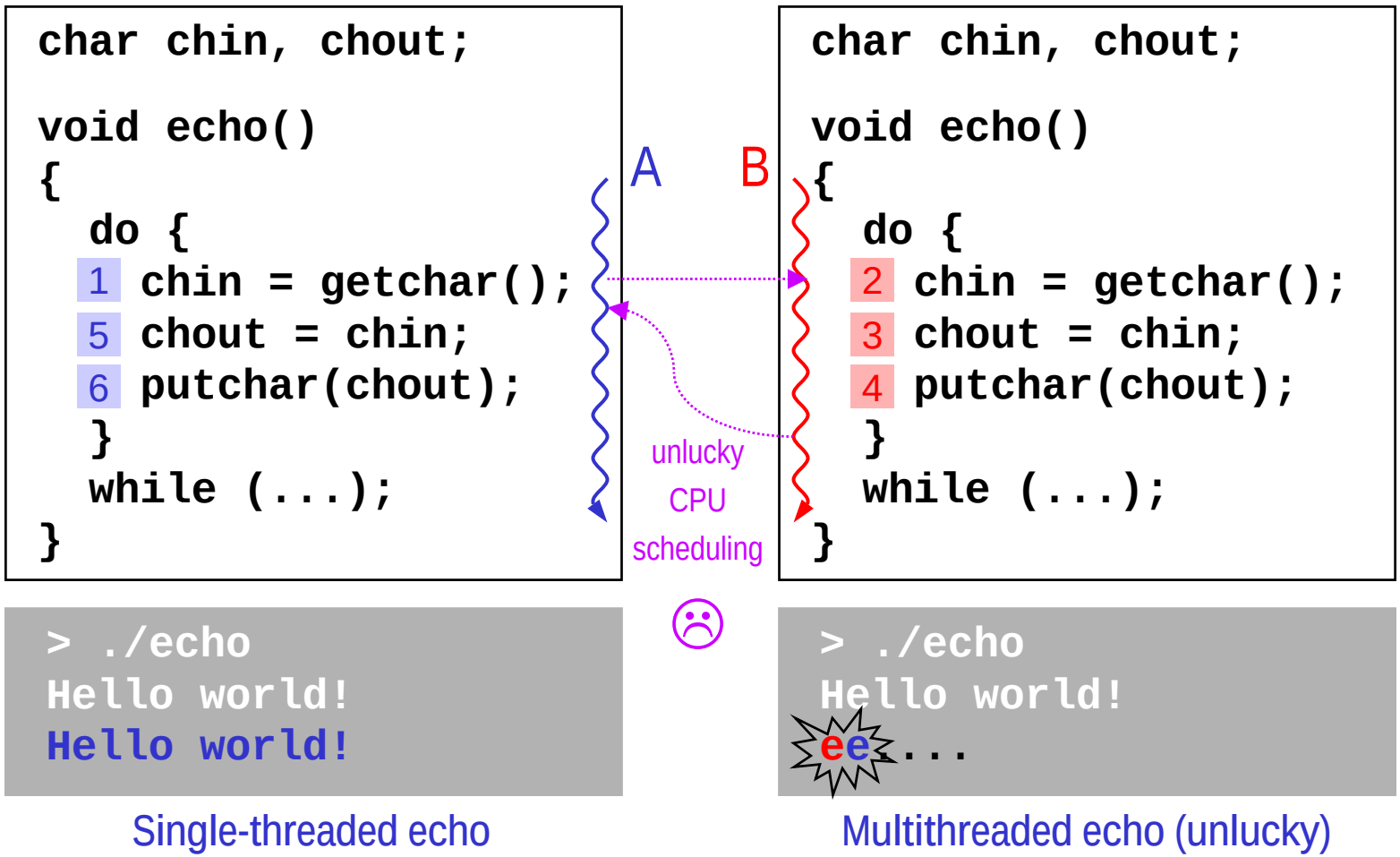
```
> ./echo
Hello world!
Hello world!
```

Multithreaded echo (lucky)

2.c Concurrency

Race conditions & critical regions

- Consequential race conditions in I/O & variable sharing



2.c Concurrency

Race conditions & critical regions

- Consequential race conditions in I/O & variable sharing

changed to local variables →

```
void echo()
{
  char chin, chout;

  do {
1   chin = getchar();
5   chout = chin;
6   putchar(chout);
  }
  while (...);
}
```

A B

unlucky CPU scheduling ☹

```
void echo()
{
  char chin, chout;

  do {
2   chin = getchar();
3   chout = chin;
4   putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
eH...
```

Multithreaded echo (unlucky)

2.c Concurrency

Race conditions & critical regions

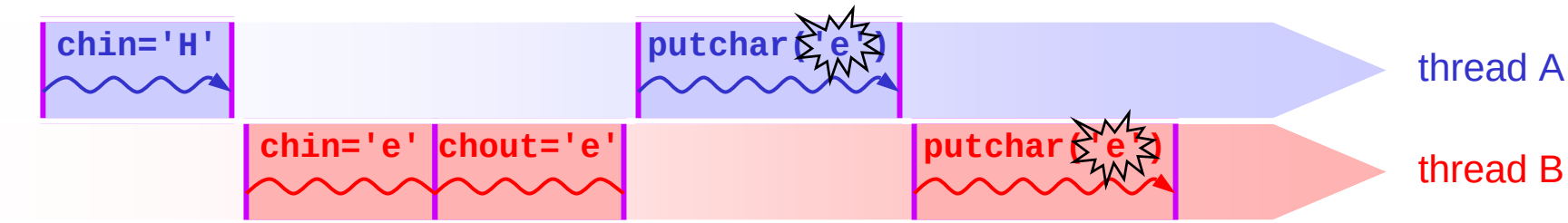
- Consequential race conditions in I/O & variable sharing
  - ✓ note that, in this case, replacing the global variables with local variables did not solve the problem
  - ✓ we actually had two race conditions here:
    - one race condition in the shared variables and the order of value assignment
    - another race condition in the shared output stream: which thread is going to write to output first (this race persisted even after making the variables local to each thread)

→ *generally, problematic race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)*

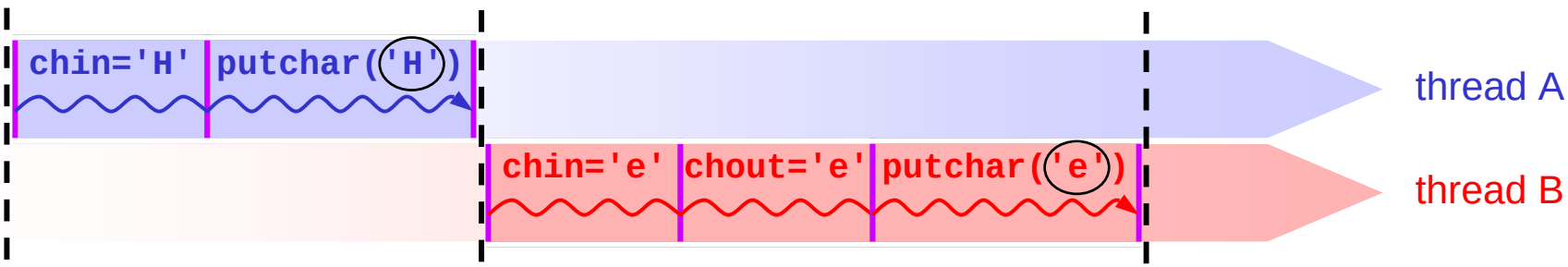
2.c Concurrency

Race conditions & critical regions

- How to avoid race conditions?
  - ✓ find a way to keep the instructions together
  - ✓ this means actually. . . reverting from too much interleaving and going back to “indivisible” blocks of execution!!



(a) too much interleaving may create race conditions

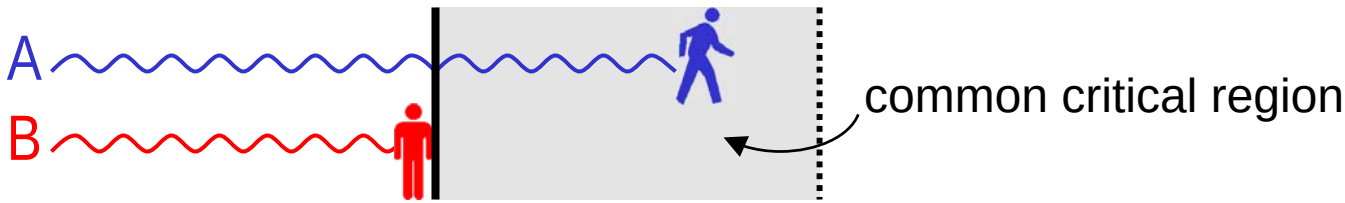


(b) keeping “indivisible” blocks of execution avoids race conditions

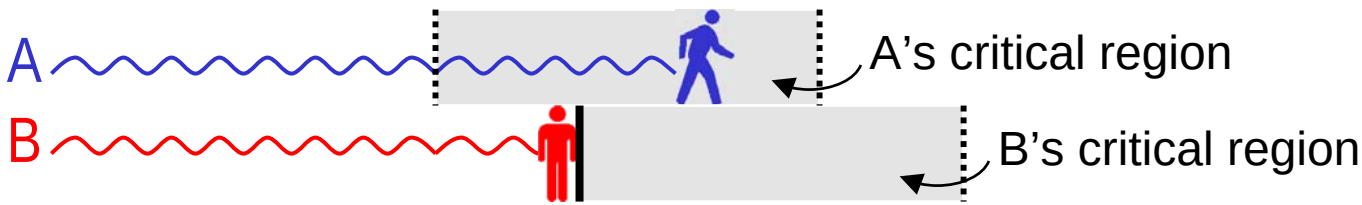
2.c Concurrency

Race conditions & critical regions

- The "indivisible" execution blocks are critical regions
  - ✓ a critical region is a section of code that may be executed by only one process or thread at a time



  - ✓ although it is not necessarily the same region of memory or section of program in both processes

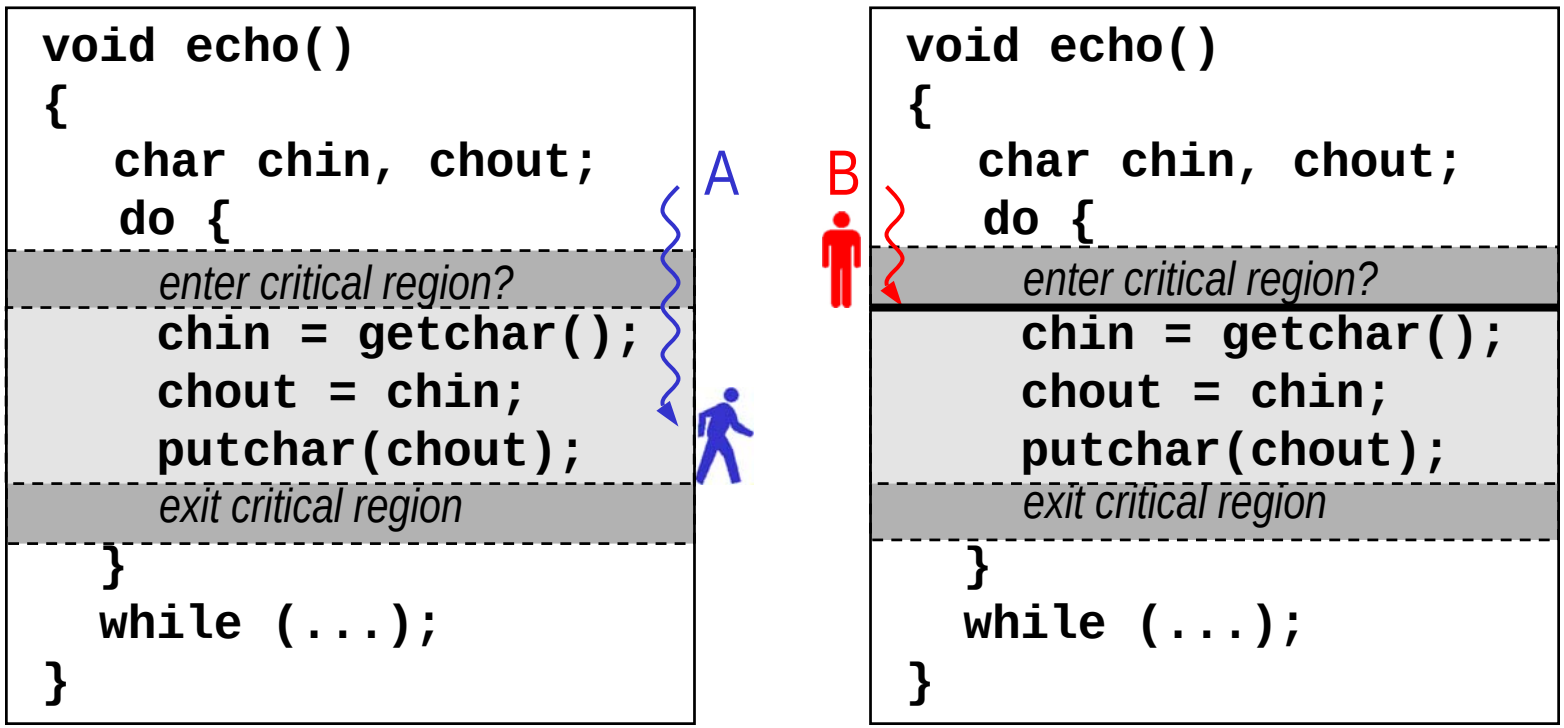


  - → *but physically different or not, what matters is that these regions cannot be interleaved or executed in parallel (pseudo or real)*

2.c Concurrency

Race conditions & critical regions

- We need mutual exclusion from critical regions
  - ✓ critical regions can be protected from concurrent access by padding them with entrance and exit gates (we'll see how later): a thread must try to check in, then it must check out



2.c Concurrency

Race conditions & critical regions

Chart of mutual exclusion

1. **mutual exclusion inside** — only one process at a time may be allowed in a critical region

2. **no exclusion outside** — a process stalled in a *non*critical region may not exclude other processes from their critical regions

3. **no indefinite occupation** — a critical region may be only occupied for a finite amount of time

2.c Concurrency

Race conditions & critical regions

Chart of mutual exclusion (cont'd)

4. **no indefinite delay when barred** — a process may be only excluded for a finite amount of time (no deadlock or starvation)

5. **no delay when about to enter** — a critical region free of access may be entered immediately by a process

6. **nondeterministic scheduling** — no assumption should be made about the relative speeds of processes

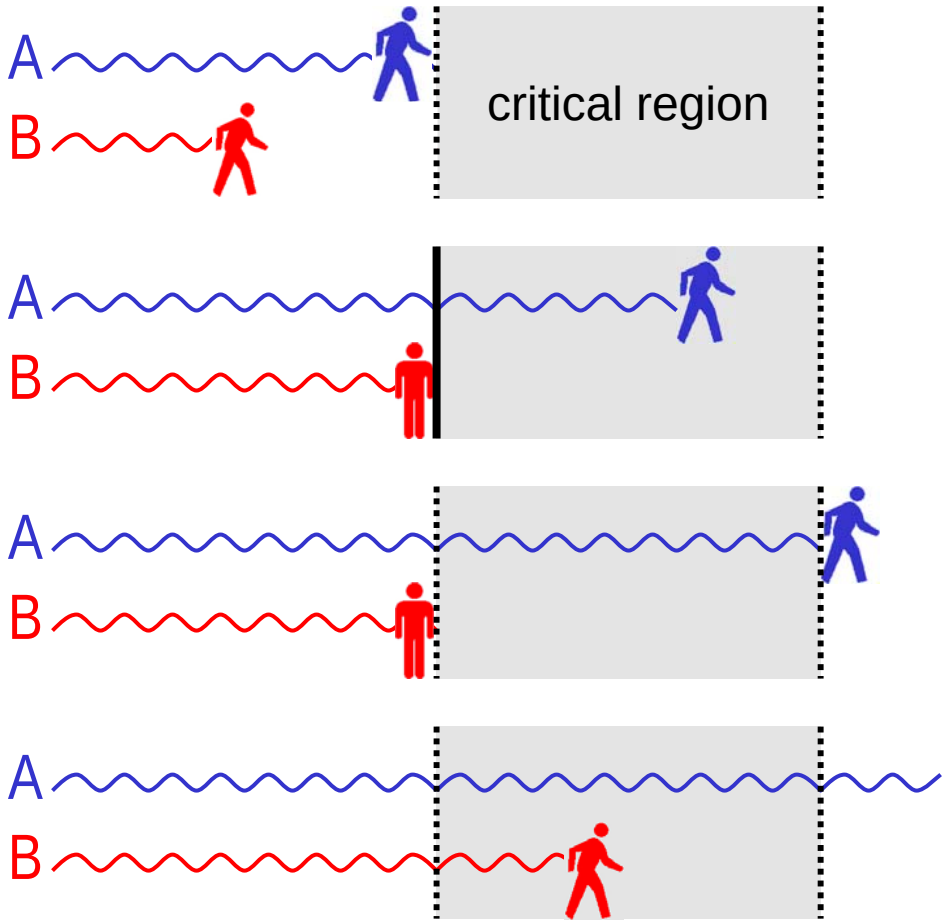
2.c Concurrency

Mutual exclusion by busy waiting

- Desired effect: mutual exclusion from the critical region

HOW is this achieved??

1. thread A reaches the gate to the critical region (CR) before B
2. thread A enters CR first, preventing B from entering (B is waiting or is blocked)
3. thread A exits CR; thread B can now enter
4. thread B enters CR

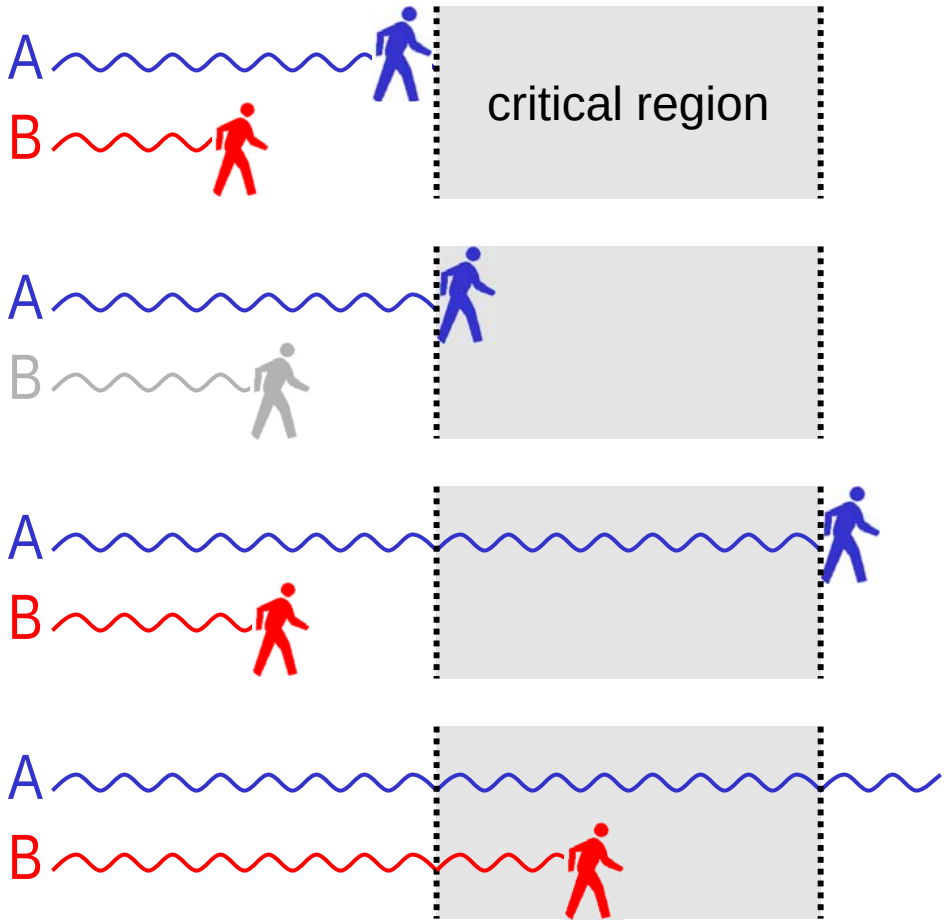


2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 0 — disabling hardware interrupts

1. thread A reaches the gate to the critical region (CR) before B
2. as soon as A enters CR, it disables all interrupts, thus B cannot be scheduled
3. as soon as A exits CR, it reenables interrupts; B can be scheduled again
4. thread B enters CR



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 0 — ~~disabling hardware interrupts~~ 👎
  - ✓ it works, but is foolish
  - ✓ what guarantees that the user process is going to ever exit the critical region?
  - ✓ meawhile, the CPU cannot interleave any other task, even unrelated to this race condition
  - ✓ the critical region becomes one *physically* indivisible block, not logically
  - ✓ also, this is not working in multi-processors

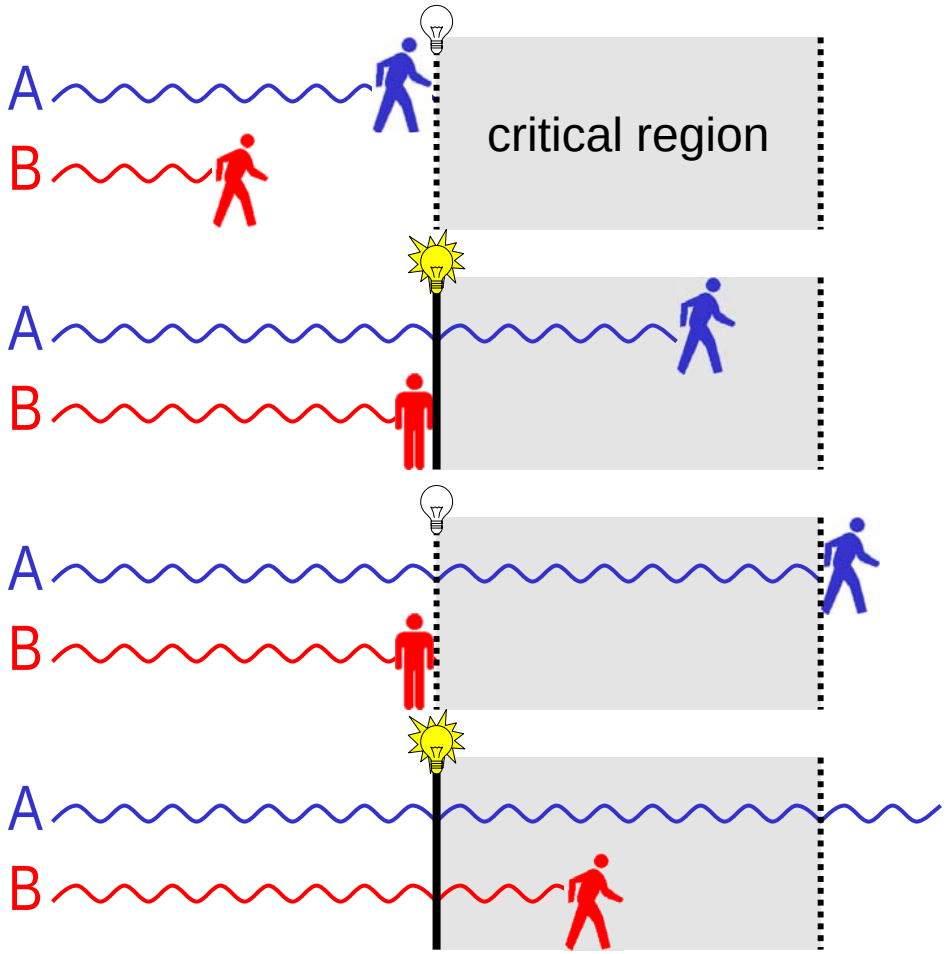
```
void echo()
{
   char chin, chout;
   do {
      disable hardware interrupts
      chin = getchar();
      chout = chin;
      putchar(chout);
      reenable hardware interrupts
   }
   while (...);
}
```

2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 1 — simple lock variable

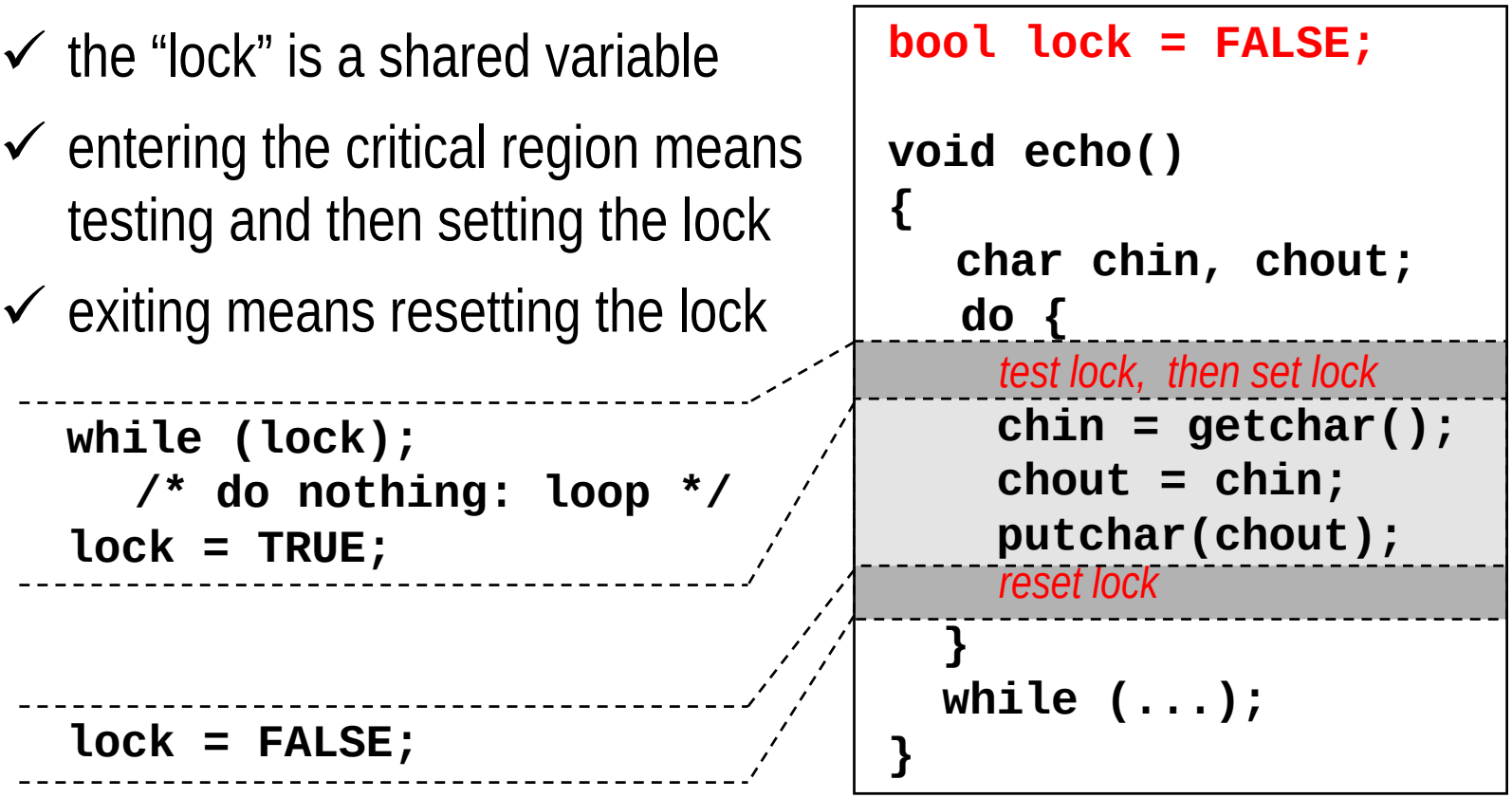
1. thread A reaches CR and finds a lock at 0, which means that A can enter
2. thread A sets the lock to 1 and enters CR, which prevents B from entering
3. thread A exits CR and resets lock to 0; thread B can now enter
4. thread B sets the lock to 1 and enters CR



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 1 — simple lock variable
  - ✓ the "lock" is a shared variable
  - ✓ entering the critical region means testing and then setting the lock
  - ✓ exiting means resetting the lock



2.c Concurrency

Mutual exclusion by busy waiting

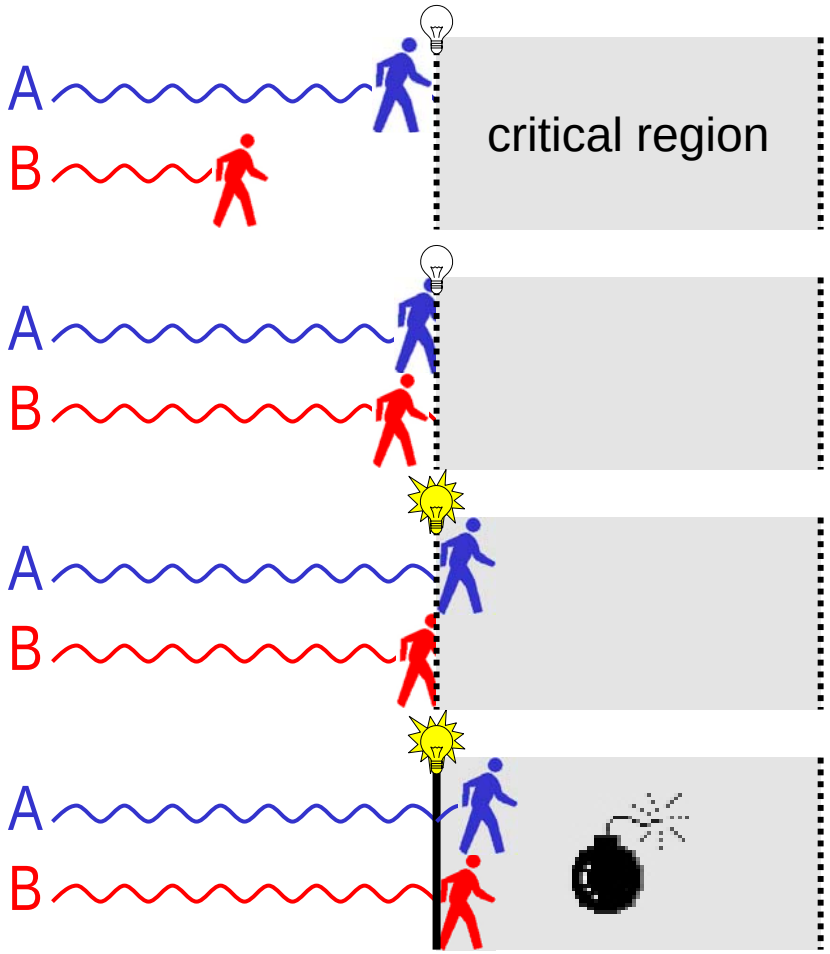
- Implementation 1 — ~~simple lock variable~~ 👎

1. thread A reaches CR and finds a lock at 0, which means that A can enter

1.1 but before A can set the lock to 1, B reaches CR and finds the lock is 0, too

1.2 A sets the lock to 1 and enters CR but cannot prevent the fact that . . .

1.3 . . . B is going to set the lock to 1 and enter CR, too



2.c Concurrency

Mutual exclusion by busy waiting

- Implementation 1 — ~~simple lock variable~~ 👎
  - ✓ suffers from the very flaw we want to avoid: a race condition
  - ✓ the problem comes from the small gap between testing that the lock is off and setting the lock

```
while (lock);   lock = TRUE;
```

  - ✓ it may happen that the other thread gets scheduled exactly inbetween these two actions (falls in the gap)
  - ✓ so they both find the lock off and then they both set it and enter

```
bool lock = FALSE;

void echo()
{
   char chin, chout;
   do {
      test lock, then set lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      reset lock
   }
   while (...);
}
```