OS: Lecture 12

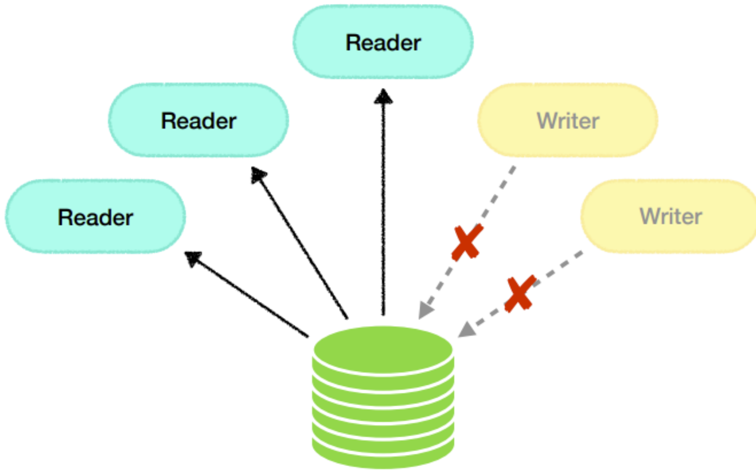
Review

- ### The producer-consumer problem

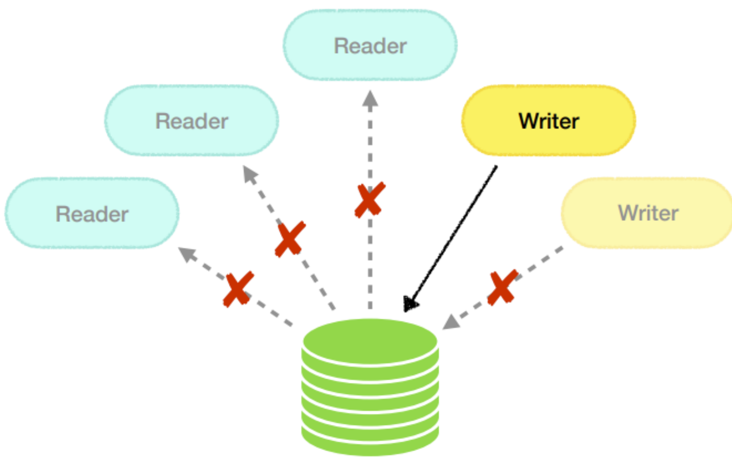
- ### The dining philosopher problem

The readers and writers problem

- Accessing a database



Multiple processes are allowed to read the database at the same time.



Multiple processes are allowed to read the database at the same time.

If a process is writing the database, no other processes may have access to the database.

How to program the readers and writers?

- Requirements

Mutual exclusion

- The database is a shared resource.

Synchronization

- When a reader is reading, other readers are allowed to read the database.
- When a reader is reading, no writers are allowed to write the database.
- When a writer is writing, no readers or writers are allowed to access the database.

Concurrency

- Concurrent access from multiple readers should be allowed.

- Attempt #1: mutex for the database

```
semaphore db = 1;
```

```
void reader() {
  for (;;) {
    down(&db);               // section entry
    data = read_database();  // critical section
    up(&db);                 // section exit
    consume_data(data);
  }
}
```

```
void writer() {
  for (;;) {
    data = produce_data();
    down(&db);               // section entry
    write_database(data);    // critical section
    up(&db);                 // section exit
  }
}
```

No two processes can be in their critical sections at the same time.

However, how to allow concurrent access from multiple readers?

- Attempt #2: allow concurrent readers

Multiple readers do not need to mutually exclude one another.

However, as long as there is a reader in the system, we need to exclude writers.

Any ideas?

The first reader that comes to the system should `down(&db)`.

The last reader that exits from the system should `up(&db)`.

We just need to count the number of readers in the system.

```
semaphore db = 1;       // controls access to the database
semaphore mutex = 1;    // controls access to "reader_count"
int reader_count = 0;   // # of processes reading or wanting to read
```

```
void reader() {
  for (;;) {
    down(&mutex);
    if (++reader_count == 1)
      down(&db);   // the first reader locks db
    up(&mutex);
    data = read_database();  // critical section
    down(&mutex);
    if (--reader_count == 0)
      up(&db);     // the last reader unlocks db
    up(&mutex);
    consume_data(data);
  }
}
```

```
void writer() {
  for (;;) {
    data = produce_data();
    down(&db);               // locks db
    write_database(data);    // critical section
    up(&db);                 // unlocks db
  }
}
```

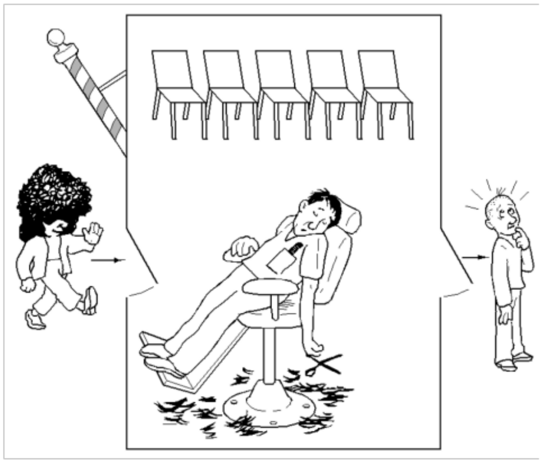
This solution meets all our requirements.

However, **it gives readers a higher priority than writers.**

As long as there is a steady supply of readers, writers will suffer from starvation.

The sleeping barber problem

The barber shop



1 barber, 1 barber chair;
n chairs for waiting customers.

If there are no customers, the barber falls asleep.

When a customer arrives, they wake up the barber.

If more customers arrive while the barber is busy…

- They wait as long as there are empty chairs; or
- They leave the shop if all chairs are full.

How to program the barber and the customers?

- ## The client-server model

This problem is similar to various queueing situations in a client-server model.

What are the processes?

- One long-running server process: the barber;
- Many transient client processes: the customers.

Any shared resources?

- The server (barber) itself is shared by all clients (customers). Only one client can be served at a time.
- Limited wait queue: *n* chairs. When the queue is full, new clients will be dropped.

We need mutual exclusion and proper synchronization.

- ## The solution

```
semaphore barbers = 0;    // # of barbers ready to cut hair (why 0 instead of 1?)
semaphore customers = 0;  // # of customers waiting for service (not being cut)
int waiting = 0;          // same as above (because there’s no way to read a semaphore’s value)
semaphore mutex = 1;      // controls access to “waiting”
```

```
void barber() {
  for (;;) {
    down(&customers);  // sleep if no customer
    down(&mutex);
    --waiting;
    up(&barbers);  // a barber is ready to serve
    up(&mutex);
    cut_hair();    // outside critical section
  }
}
```

```
void customer() {
  down(&mutex);
  if (waiting < NUM_CHAIRS) {
    ++waiting;
    up(&customers);  // wake up barber if needed
    up(&mutex);
    down(&barbers);  // wait if no barber is free
    get_haircut();   // outside critical section
  } else {
    up(&mutex);      // leave if the shop is full
  }
}
```

It’s data-race-free and deadlock-free. Any starvation?

IPC problems Summary

IPC problems involve…

- One or more shared resources;
- Multiple processes that must be synchronized.

A good solution need to…

- Guarantee mutual exclusion;
- Guarantee proper synchronization among processes;
- Be deadlock-free;
- Be starvation-free.

Threads: lightweight processes

What’s a thread?

A **thread** is an **execution entity within a process**.

So far, we have only discussed **single-threaded** processes.

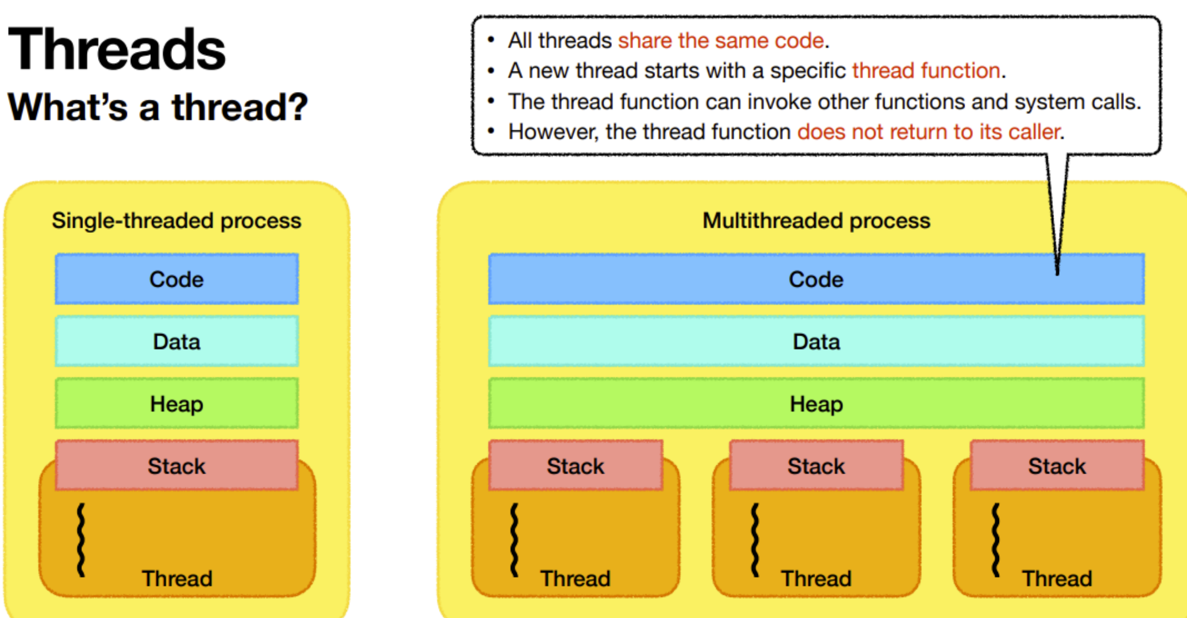
A **multithreaded process** can have more than one execution in it.

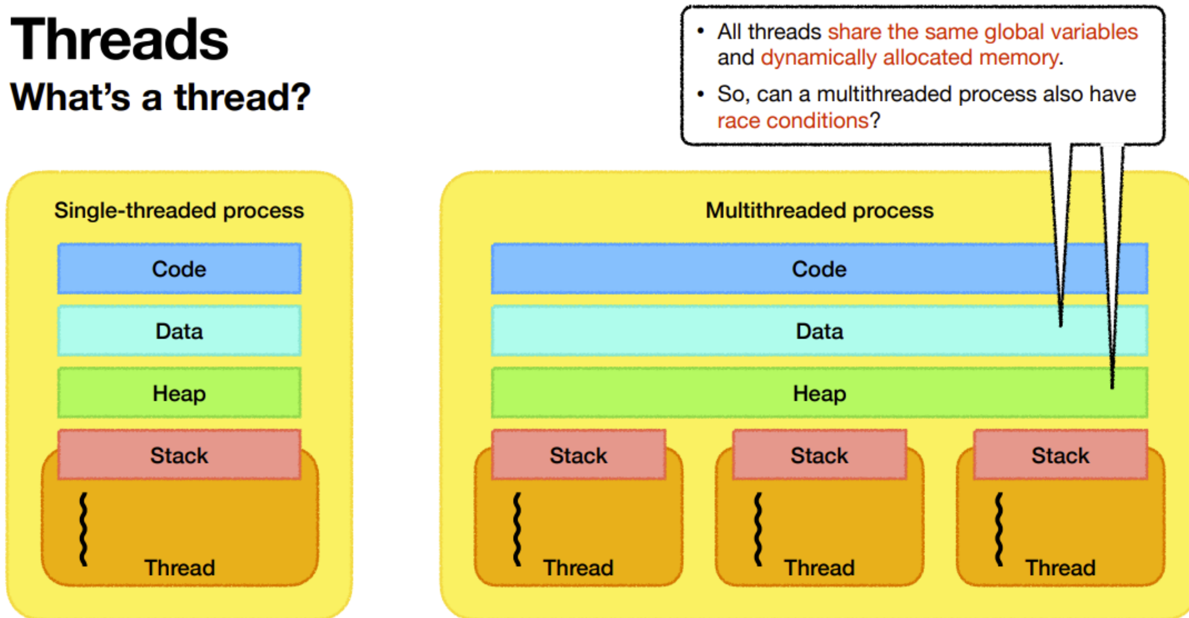
Example: a word processor may have…

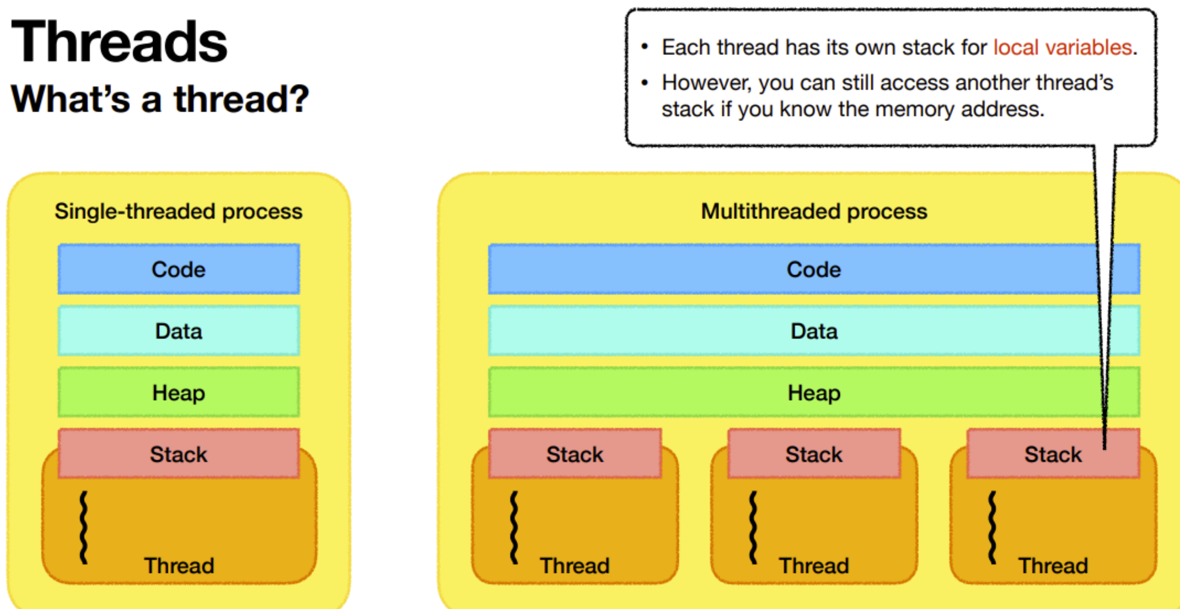
- A thread interacting with the user;
- A thread formatting text in the background;
- A thread handling automatic backups.

Threads

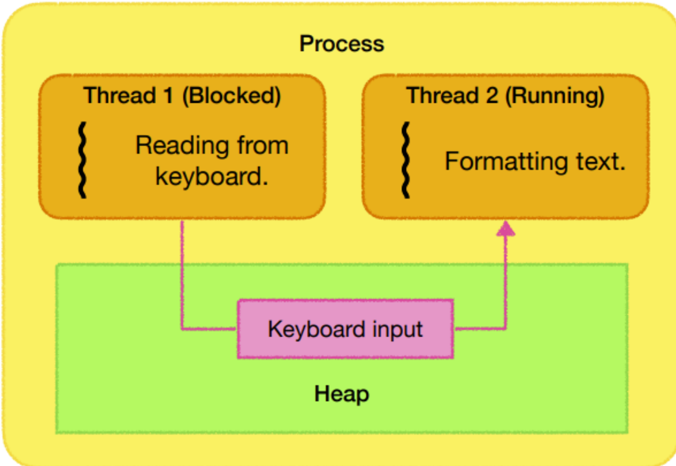
What’s a thread?




Threads
What's a thread?
• All threads share the same global variables and dynamically allocated memory.
• So, can a multithreaded process also have race conditions?
Single-threaded process
Code
Data
Heap
Stack
Thread
Multithreaded process
Code
Data
Heap
Stack
Stack
Stack
Thread
Thread
Thread


Threads
What's a thread?
• Each thread has its own stack for local variables.
• However, you can still access another thread's stack if you know the memory address.
Single-threaded process
Code
Data
Heap
Stack
Thread
Multithreaded process
Code
Data
Heap
Stack
Stack
Stack
Thread
Thread
Thread

- Why threads?



It allows multitasking within a process:

- One thread waits for the user input;
- Another thread performs computation.

⇒ Better performance and responsiveness.

Threads are easier to create and destroy than processes (10-100× faster).

Threads share the same address space; so sharing data is easy.

Thread models

Where to implement threads?

Many-to-one model

- Implement threads in user space.

One-to-one model

- Implement threads in the kernel.

Many-to-many model

- Hybrid implementations.

Many-to-one model

The threads library is entirely in user space.

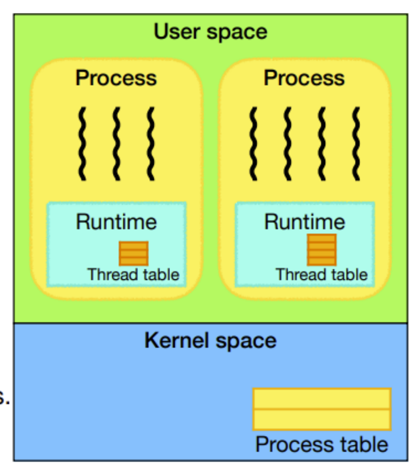
- The kernel knows nothing about threads ("green threads").
- Implemented in earlier operating systems.

Pros:

- Does not need OS support.
- Fast (no trap, no context switch, no need to flush cache…).

Cons:

- When a blocking system call is invoked, all threads will be blocked.
- Page faults (discussed later) in a thread will block the entire process.
- No preemption of threads due to the absence of clock interrupts.



- One-to-one model

The kernel manages all the threads.

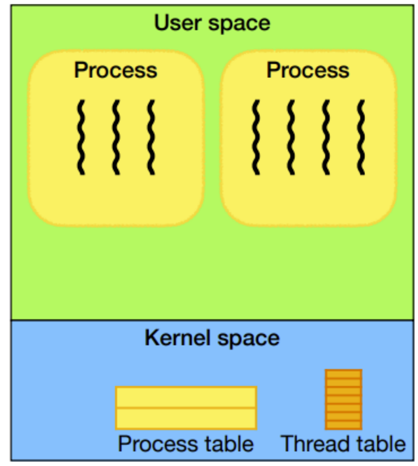
- Each thread is mapped to a kernel thread.
- Implemented in most modern operating systems.

Pros:

- When a thread blocks, the kernel can schedule another thread (from either the same process or a different process).

Cons:

- Creating and destroying threads are more expensive.
- What happens when a multithreaded process forks?
- When a signal comes in, which thread should handle it?



- Many-to-many model

The threads library multiplexes user-level threads onto kernel threads.

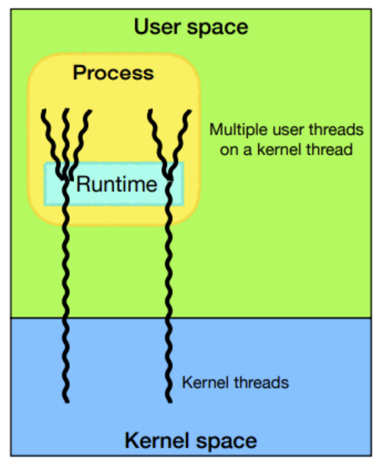
- It's a hybrid, two-level model.
- Implemented in many language runtimes (Erlang, Go, Haskell, JVM…).

Pros:

- The best of both worlds (more flexible).
- No restrictions on the number of threads.

Cons:

- More complex to implement.



Midterm

When: Thursday, March 9, from 12:30 p.m. to 1:45 p.m.

Where: Brightspace Quizzes.

What: Everything up to today's class.

- All lectures.
- Lab 1 (nyuc) and Lab 2 (nyush).
- Homework 1 and 2.

You must connect to the Zoom session and turn on your webcam for the entire duration of the exam.

You can refer to…

- Slides.
- Textbooks.
- Docker & CIMS servers (*e.g.*, read man pages, use `gcc` to try out programs).
- Online websites (you can search existing information, but you are not allowed to ask questions and seek answers).

However, keep in mind that by doing these things, you will likely waste a lot of time with little gain.

You must not communicate with any other people or AI-based assistants.

Format

~~Multiple choice questions?~~

All questions are "choose one out of two" questions!

- True or False or I don't know!
- Possible or Impossible or I don't know!

No fussing, no guessing, just choose "I don't know" to get 50% points!

Don't leave any questions blank!