CS162
Operating Systems and Systems Programming
Lecture 6

Concurrency

Professor Natacha Crooks & Matei Zaharia

https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, , Alison Norman and Lorenzo Alvisi

Goals for Today

- Threads and more threads
- Challenges and Pitfalls of Concurrency
- Synchronization Operations/Critical Sections
- How to build a lock?
- Atomic Instructions

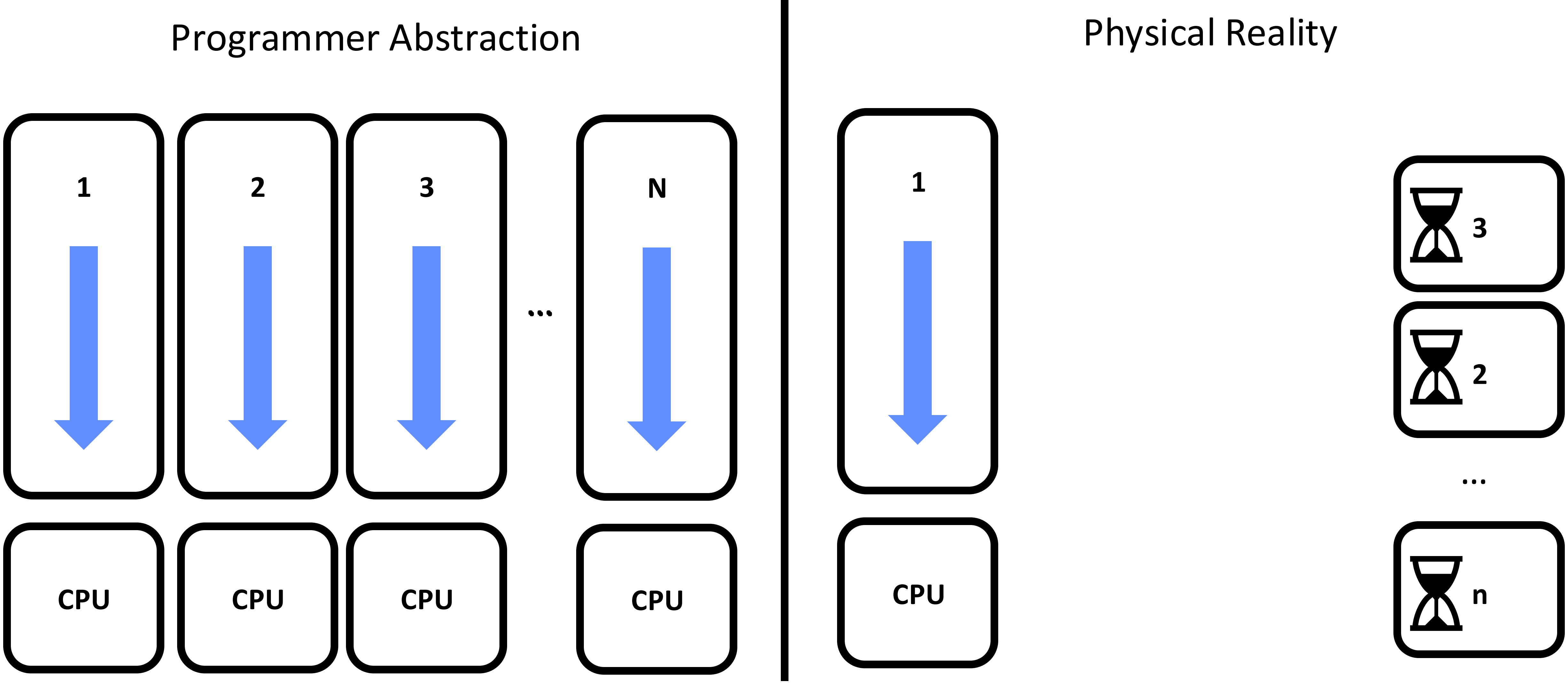
What is a thread?

A single execution sequence that represents
a separately schedulable task.

Virtualizes the processor.
Each thread runs on a dedicated virtual processor (with variable speed).
Infinitely many such processors.

Threads let users define each task with sequential code. But run each task concurrently.

What is a thread?



Recall: Thread ≠ Process

Processes defines the granularity at which the OS offers isolation and protection

Threads capture concurrent sequences of computation

Processes consist of one or more threads!

Process
Protection

Thread
Concurrency

All you need is love (and a stack)

No protection

Threads inside the same process and are not isolated from each other

⇩

Share an address space

& share IO state (FDs)

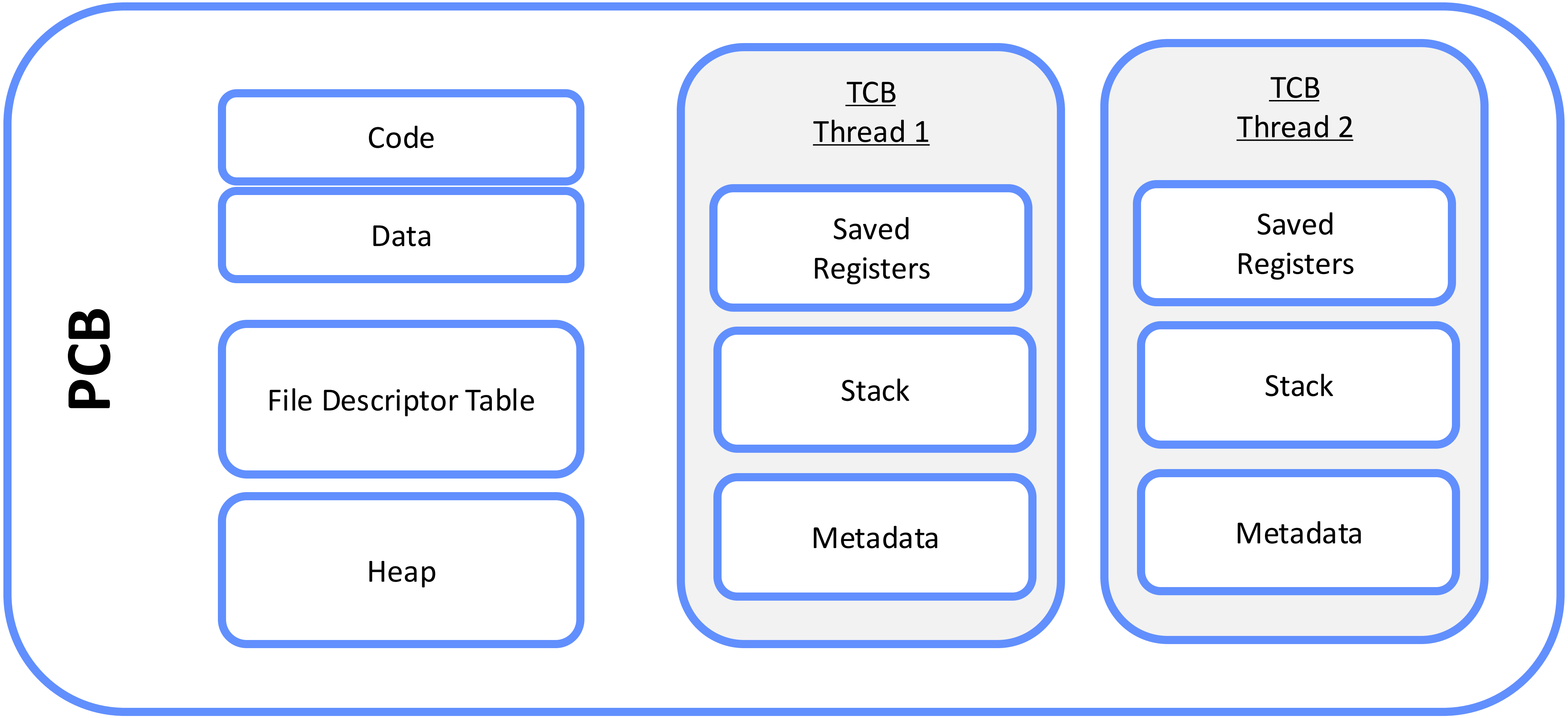
Individual execution

Threads execute disjoint instruction streams. Need own execution context

⇩

Individual stack, register state (including EIP, ESP, EBP)

All you need is love (and a stack)



Recall: Threads in Linux

Everything is a thread (`task_struct`)

Scheduler only schedules `task_struct`

To fork a process:

Invoke `clone(…)`

To create a thread:

Invoke `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0)`

CLONE_VM: Share address space. CLONE_FS: share file system. CLONE_FILES: share open files. CLONE_SIGHAND: share handlers with parents

Processes are better viewed as the containers in which threads execute

OS Library API for Threads (pThreads)

```
int pthread_create(pthread_t *thread, …
            void *(*start_routine)(void*),  void *arg);
```

Creates a thread that runs `start_routine`

```
void pthread_exit(void *value_ptr);
```

Terminates calling thread and returns value_ptr to any successful join call

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Suspends execution of calling thread until target thread terminates

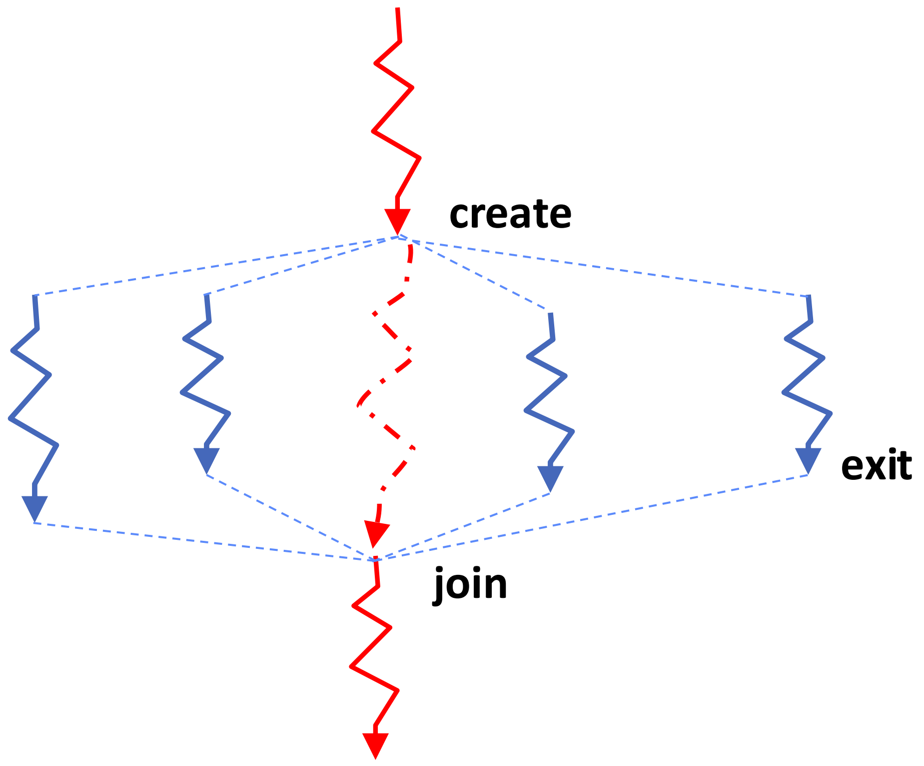
```
int pthread_yield();
```

Makes calling thread yield the CPU to other threads

Pthread Example

```
 void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
 }

 int main(int argc, char *argv[]) {
   pthread_t p1, p2;
   printf("main: begin\n");
   pthread_create(&p1, NULL, mythread, "A");
   pthread_create(&p2, NULL, mythread, "B");
   // join waits for the threads to finish
   pthread_join(p1, NULL);
   pthread_join(p2, NULL);
   printf("main: end\n");
}
```

Fork-Join Pattern



Main thread *creates* (forks) multiple sub-threads, passing them args to work on…

… and then *joins* with them, collecting results.

Example: Multithreaded Web Server

```
// Socket setup code elided…

while (1) {
  // Accept a new client connection, obtaining a new socket
  pthread_t tid;
  int conn_socket = accept(server_socket, NULL, NULL);
  int* arg = (int*) malloc(sizeof(int));
  *arg = conn_socket;
  pthread_create(&tid, NULL &serve_client, &arg);
}
close(server_socket);
```

Comparison: Fork-based Web Server

```
// Socket setup code elided…
while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  pid_t pid = fork();
  if (pid == 0) { // I am the child
    close(server_socket);
    serve_client(conn_socket);
    close(conn_socket);
    exit(0);
  } else { // // I am the parent
    close(conn_socket);
  }
}
close(server_socket);
```

Reviewing the `pthread_create(…)`

Do some work like a normal fn… place syscall # into %eax put args into registers %ebx, … special trap instruction	OS Library
Mode switches & switches to kernel stack. Saves recovery state Jump to interrupt vector table at location 128. Hands control to syscall_handler	CPU
Use %eax register to index into system call dispatch table. Invoke do_fork() method. Initialize new TCB. Mark thread READY. Push errcode into %eax	Kernel
Restore recovery state and mode switch	CPU
get return values from regs Do some more work like a normal fn…	OS Library

With great power comes great concurrency

```c
pthread_t tid[2];
int counter;

void* doSomething(void *arg) {
  unsigned long i = 0;
  for (int i = 0 ; i < 1000 ; i++) {
    counter += 1;
  }
  return NULL;
}

int main(void) {
  int i = 0;
  while(i++ < 2) {
    pthread_create(&(tid[i]), NULL, &doSomething,
  }
  pthread_join(tid[0], NULL);
  pthread_join(tid[1], NULL);
  printf("Counter %d \n", counter);
  return 0;
}
```

What will be the final answer?

```
matei@laptop> gcc concurrency.c -o concurrency -pthread
```

```
matei@laptop> ./concurrency
```

```
Counter 2000
```

```
matei@laptop> ./concurrency
```

```
Counter 1937
```

```
matei@laptop> ./concurrency
```

```
Counter 1899
```

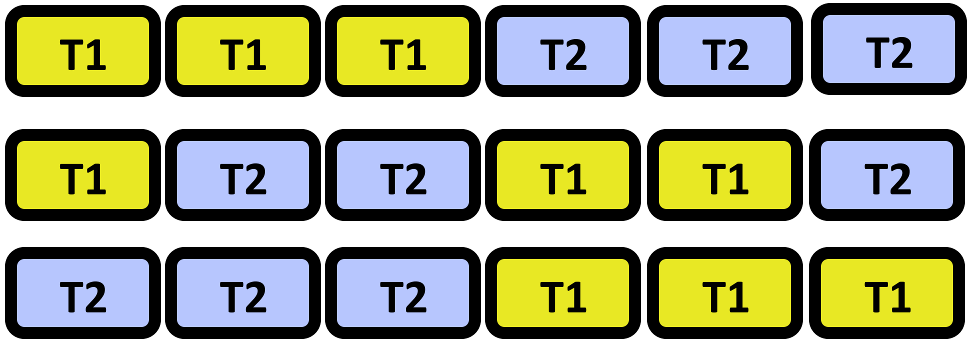
With great power comes great concurrency

Protection is at process level.

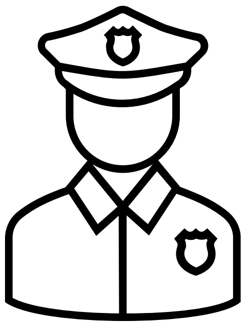
Threads not isolated.

Share an address space.

Non-deterministic interleaving of threads



With great power comes great concurrency



Public Enemy #1:

THE RACE CONDITION

Today and next three lectures: how can we regulate access to shared data across threads?

Multiprocessing vs Multiprogramming

Multiprocessing $\equiv$ multiple CPUs

Multiprogramming $\equiv$ multiple jobs or processes

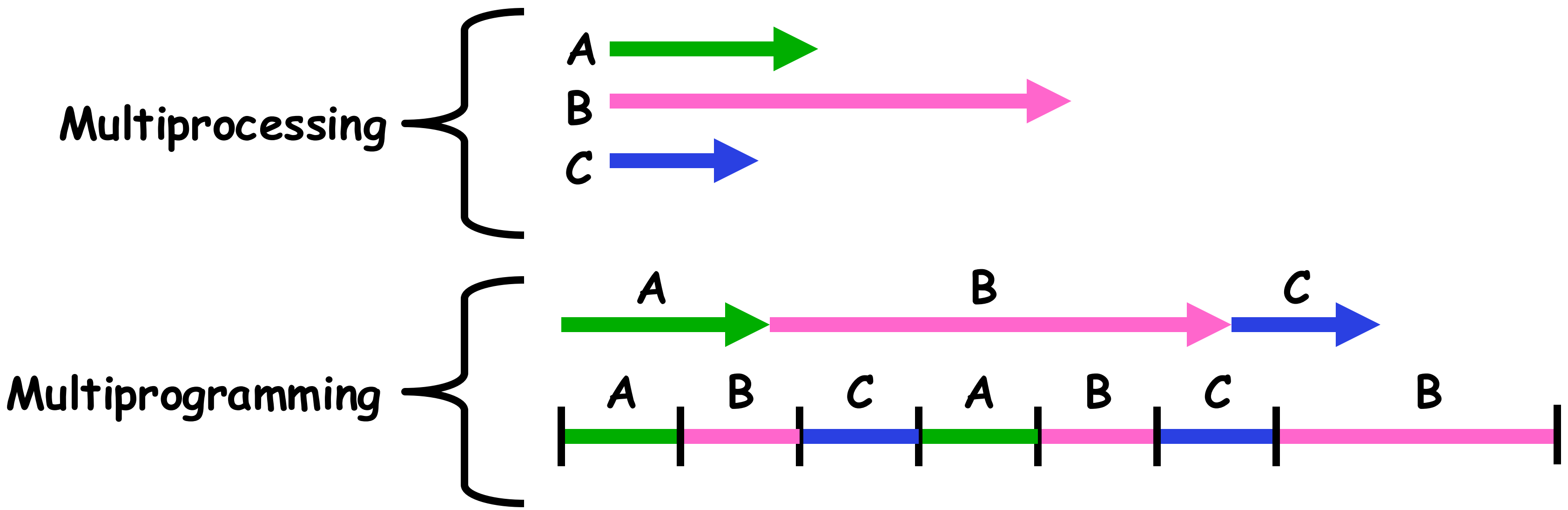
Multithreading $\equiv$ multiple threads per process

Multiprocessing vs Multiprogramming

What does it mean to run two threads "concurrently"?

=> Scheduler is free to run threads in any order

=> Can choose to run each thread to completion or time-slice in big or small chunks

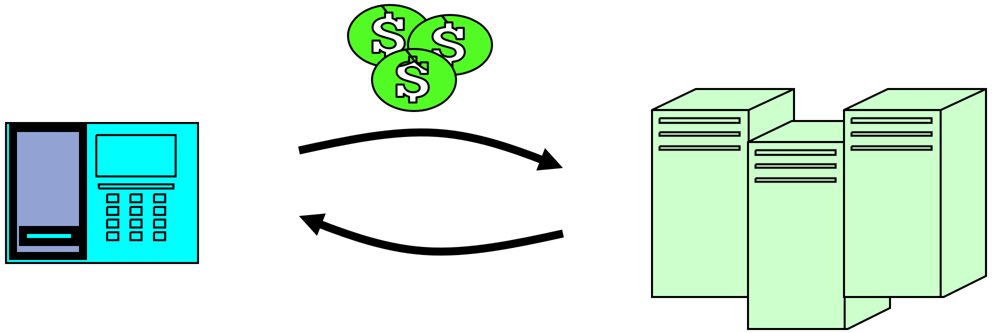


ATM Bank Server

Service a set of requests

Do so without corrupting database

Don’t hand out too much money



ATM Bank server Example

Suppose we wanted to implement a server process to handle requests from an ATM network:

```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      ProcessRequest(op, acctId, amount);
   }
}

ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}

Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* requires disk I/O */
}
```

Event Driven Version of ATM server

Suppose we only had one CPU. Still want to overlap I/O with computation. Without threads, we would have to rewrite in event-driven style:

```
BankServer() {
   while(TRUE) {
      event = WaitForNextEvent();
      if (event == ATMRequest)
         StartOnRequest();
      else if (event == AcctAvail)
         ContinueRequest();
      else if (event == AcctStored)
         FinishRequest();
   }
}
```

Can Threads Make This Easier?

Threads yield overlapped I/O and computation without “deconstructing” code into non-blocking fragments

One thread per request

Requests proceeds to completion, blocking as required

Can Threads Make This Easier?

Suppose we wanted to implement a server process to handle requests from an ATM network:

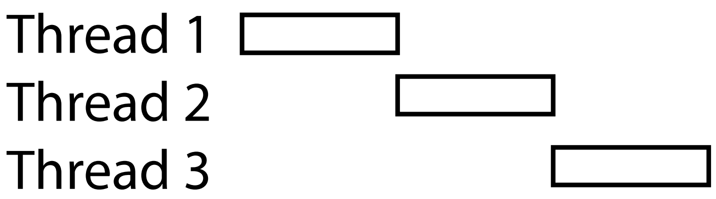
```
BankServer() {
   while (TRUE) {
      ReceiveRequest(&op, &acctId, &amount);
      START_THREAD(ProcessRequest(op, acctId, amount))
   }
}

ProcessRequest(op, acctId, amount) {
   if (op == deposit) Deposit(acctId, amount);
   else if …
}

Deposit(acctId, amount) {
   acct = GetAccount(acctId); /* may use disk I/O */
   acct->balance += amount;
   StoreAccount(acct); /* requires disk I/O */
}
```

Remember the Race Condition …

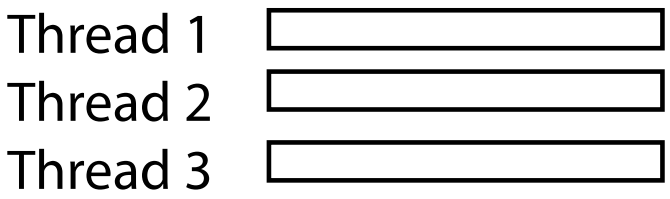
Shared state can get corrupted

Thread 1	Thread 2
`load r1, acct->balance`	
	`load r1, acct->balance`
	`add r1, amount2`
	`store r1, acct->balance`
`add r1, amount1`	
`store r1, acct->balance`	

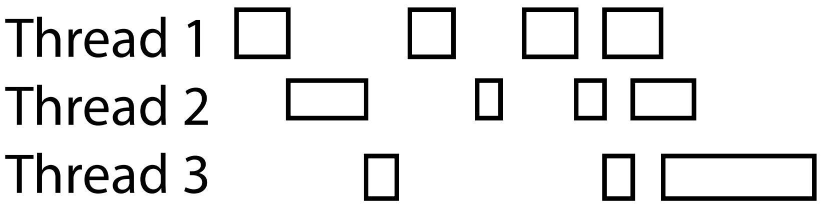
Many Possible Executions



a) One execution



b) Another execution



c) Another execution

Problem is at the Lowest Level

Most of the time, threads are working on separate data, so scheduling doesn't matter

Thread A	Thread B
x = 1;	y = 2;

However, what about (Initially, y = 12):

Thread A	Thread B
x = 1;	y = 2;
x = y+1;	y = y*2;

What if two threads are both writing to x?

Atomic Operations

An operation that always runs to completion, or not at all

It is *indivisible:* it cannot be stopped in the middle, and state cannot be modified by someone else in the middle

Fundamental building block

If no atomic operations, then have no way for threads to work together

Atomic Operations

On most machines, memory references and assignments (i.e. loads and stores) of words are atomic

Consequently – weird examples with partial writes to ints can't happen

Many instructions are not atomic

– Double-precision floating point store often not atomic

– VAX and IBM 360 had an instruction to copy a whole array

Another Concurrent Program Example

Two threads, A and B, compete with each other

Thread A

```
i = 0;
while (i < 10)
   i = i + 1;
printf("A wins!");
```

Thread B

```
i = 0;
while (i > -10)
   i = i – 1;
printf("B wins!");
```

Assume that memory loads and stores are atomic, but incrementing and decrementing are *not* atomic

What happens?

Definitions

Synchronization

Using atomic operations to ensure cooperation between threads

Mutual Exclusion

Ensuring that only one thread does a particular thing at a time

Critical Section

Piece of code that only one thread can execute at once. Only one thread at a time will get into this section of code

Locks

Prevents someone from doing something

`Lock()` before entering critical section and before accessing shared data

`Unlock()` when leaving, after accessing shared data

Wait if locked

Important idea:

All synchronization involves waiting

API for Locks

Locks need to be allocated and initialized:

- `struct Lock mylock`   or   `pthread_mutex_t mylock;`
- `lock_init(&mylock)`   or   `mylock = PTHREAD_MUTEX_INITIALIZER;`

Locks provide two **atomic** operations:

- `acquire(&mylock)` – wait until lock is free; then mark it as busy
- `release(&mylock)` – mark lock as free
  - » Should only be called by a thread that currently holds the lock

How would you fix the ATM problem?

(No, getting rid of money is not an option for this class)

Fix banking problem with Locks!

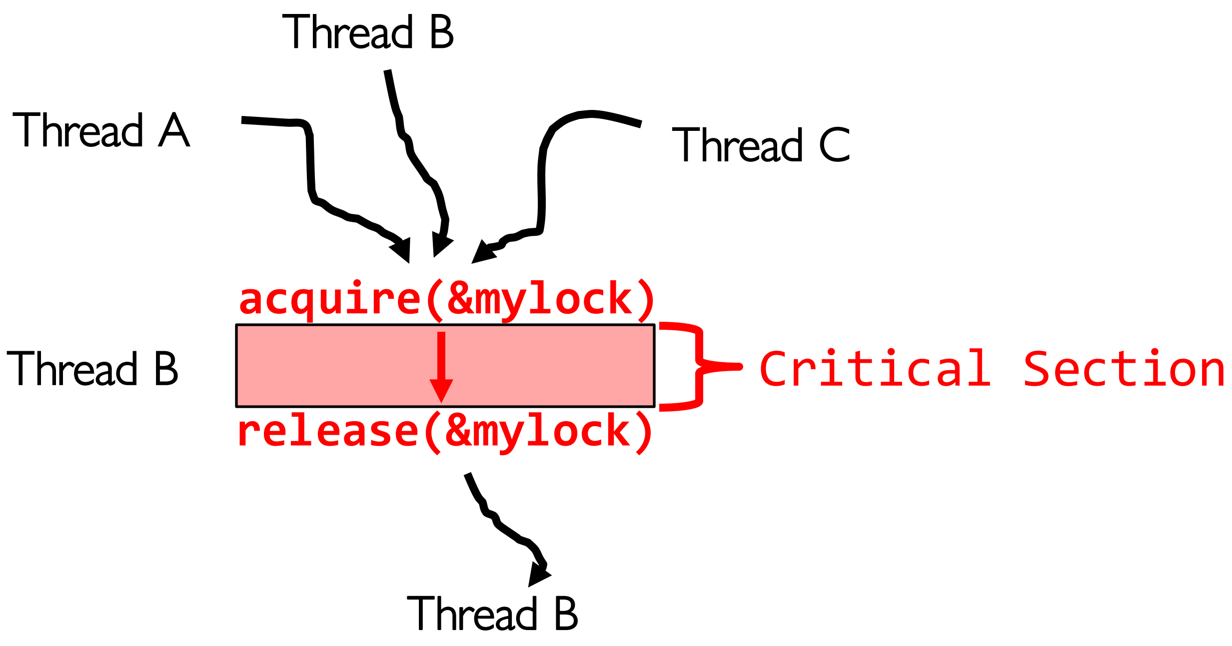
Identify critical sections (atomic instruction sequences)

and add locking

```
Deposit(acctId, amount) {
  acquire(&mylock)          // Wait if someone else in critical section!
  acct = GetAccount(actId);
  acct->balance += amount;
  StoreAccount(acct);
  release(&mylock)          // Release someone into critical section
}
```

Critical Section

Fix banking problem with Locks!



Threads serialized by lock through critical section.

Only one thread at a time

Correctness Requirements

Threaded programs must work for **all interleavings** of thread instruction sequences

Cooperating threads inherently non-deterministic and non-reproducible

Really hard to debug unless carefully designed!

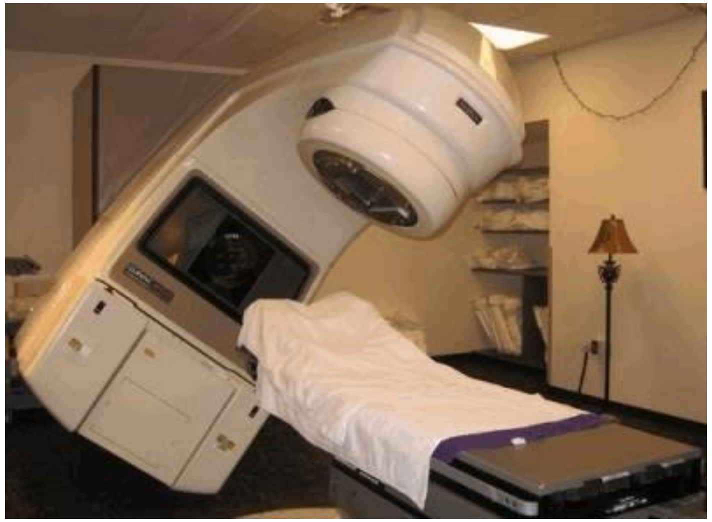
Therac-25

Machine for radiation therapy

Software control of electron accelerator and electron beam/ Xray production

Software control of dosage

Concurrency bugs caused the death of several patients



The Importance of Milk

The Importance of Milk

Great thing about OS's – analogy between problems in OS and problems in real life

Help you understand real life problems better

But, computers are much stupider than people

Motivating Example: “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	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Solve with a lock?

Lock prevents someone from doing something

– Lock before entering critical section

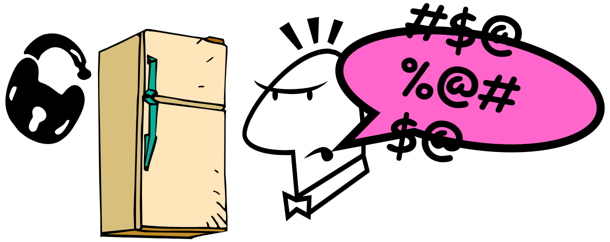
– Unlock when leaving

– Wait if locked

Fix the milk problem by putting a key on the refrigerator

Lock it and take key if you are going to go buy milk

Fixes too much: roommate angry if only wants OJ



Too Much Milk: Correctness Properties

Need to be careful about correctness of concurrent programs, since they are non-deterministic

– Impulse is to start coding first, then when it doesn't work, pull hair out

– Instead, think first, then code!

– Always write down desired behavior first

Too Much Milk: Correctness Properties

What are the correctness properties for the "Too much milk" problem???

– Never more than one person buys

– Someone buys if needed

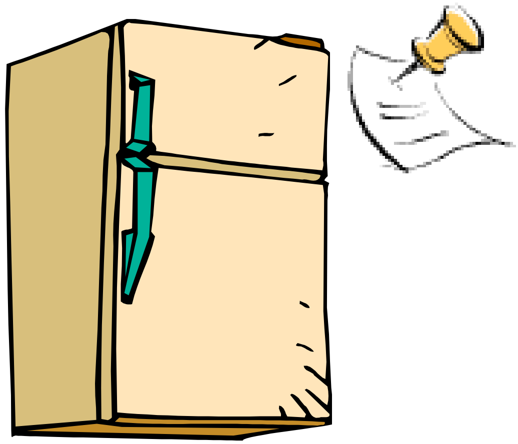
First attempt: Restrict ourselves to use only atomic load and store operations as building blocks

Too Much Milk: Solution #1

Use a note to avoid buying too much milk:
- – Leave a note before buying (kind of “lock”)
- – Remove note after buying (kind of “unlock”)
- – Don’t buy if note (wait)

Suppose a computer tries this
(remember, only memory read/write are atomic)

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```



Too Much Milk: Solution #1

Thread A

```
if (noMilk) {


   if (noNote) {
      leave Note;
      buy Milk;
      remove Note;
   }
}
```

Thread B

```
if (noMilk) {
   if (noNote) {







      leave Note;
      buy Milk;
      remove Note;
   }
}
```

Too Much Milk: Solution #1

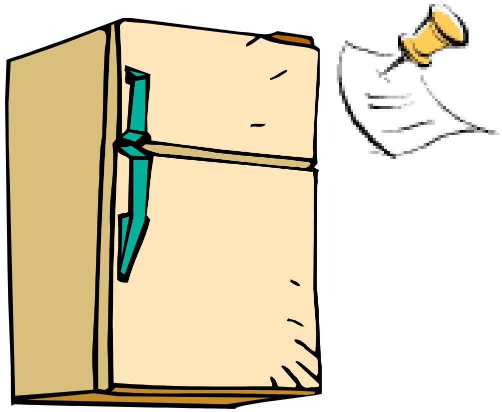
Still too much milk but only occasionally!

Thread can get context switched after checking milk and note but before buying milk!

Solution makes problem worse since fails intermittently

– Makes it really hard to debug…

– Must work despite what the scheduler does!



Too Much Milk: Solution #1½

Let's try to fix this by placing note first

```
leave Note;
if (noMilk) {
   if (noNote) {
      buy milk;
   }
}
remove Note;
```

What happens here?
- Well, with human, probably nothing bad
- With computer: no one ever buys milk

Too Much Milk Solution #2

How about labeled notes?
- Now we can leave note before checking

Algorithm looks like this:

Thread A

```
leave note A;
if (noNote B) {
   if (noMilk) {
      buy Milk;
   }
}
remove note A;
```

Thread B

```
leave note B;
if (noNoteA) {
   if (noMilk) {
      buy Milk;
   }
}
remove note B;
```

Too Much Milk Solution #2

Possible for neither thread to buy milk

– Context switches at exactly the wrong times can lead each to think that the other is going to buy

Really insidious:

– Extremely unlikely this would happen, but will at worst possible time

– Probably something like this in UNIX

Too Much Milk Solution #2: problem!

I'm not getting milk, *You're* getting milk

This kind of lockup is called "starvation!"

Too Much Milk Solution #3

<u>Thread A</u>

```
leave note A;
while (note B) { //X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

<u>Thread B</u>

```
leave note B;
if (noNote A) { //Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

Too Much Milk Solution #3

Both can guarantee that:
- It is safe to buy, or
- Other will buy, ok to quit

At X:
- If no note B, safe for A to buy,
- Otherwise wait to find out what will happen

At Y:
- If no note A, safe for B to buy
- Otherwise, A is either buying or waiting for B to quit

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 1

- "`leave note A`" happens before "`if (noNote A)`"

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Wait for note B to be removed

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

happened before

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note A;
while (note B) {\\X
    do nothing;
};

if (noMilk) {
    buy milk;}
}
remove note A;
```

happened before

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

Case 2

- "`if (noNote A)`" happens before "`leave note A`"

```
leave note B;
if (noNote A) {\\Y
    if (noMilk) {
        buy milk;
    }
}
remove note B;
```

happened before

```
leave note A;
while (note B) {\\X
    do nothing;
};
```

Wait for note B to be removed

```
if (noMilk) {
    buy milk;}
}
remove note A;
```

This Generalizes to n Threads…

Leslie Lamport's "Bakery Algorithm" (1974)

Computer Systems

G. Bell, D. Siewiorek, and S.H. Fuller, Editors

A New Solution of Dijkstra's Concurrent Programming Problem

Leslie Lamport
Massachusetts Computer Associates, Inc.

A simple solution to the mutual exclusion problem is presented which allows the system to continue to operate

Solution #3 discussion

Solution #3 works, but it's really unsatisfactory

- Really complex – even for this simple an example
  - » Hard to convince yourself that this really works
- A's code is different from B's – what if lots of threads?
  - » Code would have to be slightly different for each thread
- While A is waiting, it is consuming CPU time
  - » This is called "busy-waiting"

Too Much Milk: Solution #4?

Recall our target lock interface:

- `acquire(&milklock)` – wait until lock is free, then grab
- `release(&milklock)` – Unlock, waking up anyone waiting
- These must be atomic operations – if two threads are waiting for the lock and both see it's free, only one succeeds to grab the lock

Then, our milk problem is easy:

```
acquire(&milklock);
if (nomilk)
   buy milk;
release(&milklock);
```

Where are we going with synchronization?

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Compare&Swap

Implement various higher-level synchronization primitives using atomic operations