CS162
Operating Systems and
Systems Programming
Lecture 6

Concurrency

Professor Natacha Crooks
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 enable users to define each task with *sequential code*. But run each task *concurrently*
What is a thread?

Programmer Abstraction

1
2
3
N

CPU
CPU
CPU
CPU

Physical Reality

1

CPU

2

3

n

...
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!
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);
Thread created and runs start_routine

void pthread_exit(void *value_ptr);
Terminates thread and makes value_ptr available to any successful join

int pthread_yield();
Causes thread to yield the CPU to other threads

int pthread_join(pthread_t thread, void **value_ptr);
Suspends execution of calling thread until target thread terminates.
Pthread Example

```c
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) collection of sub-threads, passing them args to work on...
... and then *joins* with them, collecting results.
Revisit the Server Protocol

// 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);

How would you rewrite the concurrent server example using threads rather than processes?
Multiprocess  Multithreaded server!

// Socket setup code elided...

int

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);
### Reviewing the `pthread_create(...)`

- **Do some work like a normal fn...**
  - place syscall # into %eax
  - put args into registers %ebx, ...
  - special trap instruction

- **Mode switches & switches to kernel stack.**
  - Saves recovery state
  - Jump to interrupt vector table at location 128.
  - Hands control to syscall_handler

- **Use %eax register to index into system call dispatch table.** Invoke `do_fork()` method. Initialise new TCB. Mark thread READY. Push errcode into %eax

- **Restore recovery state and mode switch**

- **get return values from regs**
  - Do some more work like a normal fn...
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, NULL);
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    printf("Counter %d \n", counter);
    return 0;
}
```

What will be the final answer?

crooks@laptop> gcc concurrency.c -o concurrency –pthread
crooks@laptop> ./concurrency
Counter 2000

crooks@laptop> ./concurrency
Counter 1937

crooks@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

\[
\text{\textbullet\, T1\, T1\, T1\, T2\, T2\, T2}
\text{\textbullet\, T1\, T2\, T2\, T1\, T1\, T2}
\text{\textbullet\, T2\, T2\, T2\, T1\, T1\, T1}
\]
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 = Multiple CPUs

Multiprogramming ≡ Multiple Jobs or Processes

Multithreading ≡ Multiple threads per Process
What does it mean to run two threads “concurrently”?

=> **Scheduler** is free to run threads in any order

=> **Dispatcher** can choose to run each thread to completion or time-slice in big chunks 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); /* Involves disk I/O */
}
Suppose we only had one CPU. Still like to overlap I/O with computation. Without threads, we would have to rewrite in event-driven style.

```c
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 proceed 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:

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

```c
ProcessRequest(op, acctId, amount) {
    if (op == deposit) Deposit(acctId, amount);
    else if ... 
}
```

```c
Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* may use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
}
Remember the Race Condition ...

Shared state can get corrupted

Thread 1

load r1, acct->balance
add r1, amount1
store r1, acct->balance

Thread 2

load r1, acct->balance
add r1, amount2
store r1, acct->balance
Many Possible Executions

Thread 1  
Thread 2  
Thread 3  

a) One execution

Thread 1  
Thread 2  
Thread 3  

b) Another execution

Thread 1  
Thread 2  
Thread 3  

c) Another execution
Problem is at the Lowest Level

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

\[
\begin{align*}
\text{Thread A} & \\
& x = 1; \\
& y = 2;
\end{align*}
\]

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

\[
\begin{align*}
\text{Thread A} & \\
& x = 1; \\
& x = y+1;
\end{align*}
\]

\[
\begin{align*}
\text{Thread B} & \\
& y = 2; \\
& y = y*2;
\end{align*}
\]

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 example that produces "3" on previous slide 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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 0;$</td>
<td>$i = 0;$</td>
</tr>
<tr>
<td>while ($i &lt; 10$)</td>
<td>while ($i &gt; -10$)</td>
</tr>
<tr>
<td>$i = i + 1;$</td>
<td>$i = i - 1;$</td>
</tr>
<tr>
<td>printf(&quot;A wins!&quot;);</td>
<td>printf(&quot;B wins!&quot;);</td>
</tr>
</tbody>
</table>

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
Locks need to be allocated and initialized:
- structure 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
}
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!
Machine for radiation therapy

Software control of electron accelerator and electron beam/X-ray production

Software control of dosage

Software errors caused the death of several patients

Figure 1: Typical Therac-25 Facility
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”

<table>
<thead>
<tr>
<th>Time</th>
<th>Person A</th>
<th>Person B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3:00</td>
<td>Look in Fridge. Out of milk</td>
<td></td>
</tr>
<tr>
<td>3:05</td>
<td>Leave for store</td>
<td></td>
</tr>
<tr>
<td>3:10</td>
<td>Arrive at store</td>
<td>Look in Fridge. Out of milk</td>
</tr>
<tr>
<td>3:15</td>
<td>Buy milk</td>
<td>Leave for store</td>
</tr>
<tr>
<td>3:20</td>
<td>Arrive home, put milk away</td>
<td>Arrive at store</td>
</tr>
<tr>
<td>3:25</td>
<td></td>
<td>Buy milk</td>
</tr>
<tr>
<td>3:30</td>
<td></td>
<td>Arrive home, put milk away</td>
</tr>
</tbody>
</table>
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 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 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
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)

```java
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;
    }
}

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 dispatcher does!
Let’s try to fix this by placing note first

```plaintext
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;
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 worse 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!”
Thread A
leave note A;
while (note B) {
    do nothing;
}
if (noMilk) {
    buy milk;
}
remove note A;

Thread B
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
}
remove note B;
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)"

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

leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    } 
    remove note B;
}
remove note A;
```
Case 1

• “leave note A” happens before “if (noNote A)”

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

if (noMilk) {
    buy milk;
}  
remove note A;
```
Case 1

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

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

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

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

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

Wait for note B to be removed
Case 2

- “if (noNote A)” happens before “leave note A”
Case 2

- “if (noNote A)” happens before “leave note A”
Case 2

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

```c
leave note A;
while (note B) {
    do nothing;
}
if (noMilk) {
    buy milk;
}
remove note A;
leave note B;
if (noNote A) {
    if (noMilk) {
        buy milk;
    }
} else {
    remove note B;
}
```
This Generalizes to $n$ Threads...

Leslie Lamport's "Bakery Algorithm" (1974)

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”
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:
```c
acquire(&milklock);
if (nomilk)
  buy milk;
release(&milklock);
```
**Where are we going with synchronization?**

<table>
<thead>
<tr>
<th>Programs</th>
<th>Shared Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Higher-level API</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>Load/Store</td>
</tr>
<tr>
<td></td>
<td>Disable Ints</td>
</tr>
<tr>
<td></td>
<td>Compare&amp;Swap</td>
</tr>
<tr>
<td></td>
<td>Locks</td>
</tr>
<tr>
<td></td>
<td>Semaphores</td>
</tr>
<tr>
<td></td>
<td>Monitors</td>
</tr>
<tr>
<td></td>
<td>Send/Receive</td>
</tr>
<tr>
<td></td>
<td>Test&amp;Set</td>
</tr>
</tbody>
</table>

Implement various higher-level synchronization primitives using atomic operations.