CS162
Operating Systems and
Systems Programming
Lecture 6

Synchronization 1: Concurrency
and Mutual Exclusion

September 13th, 2022
Rohan Kumar
Slides by Prof. Anthony Joseph and John Kubiatowicz
http://cs162.org
Recall: Connection Setup over TCP/IP

Client Side

Connection request:
1. Client IP addr
2. Client Port
3. Protocol (TCP/IP)

Server Side

Server Listening:
1. Server IP addr
2. well-known port,
3. Protocol (TCP/IP)

• 5-Tuple identifies each connection:
  1. Source IP Address
  2. Destination IP Address
  3. Source Port Number
  4. Destination Port Number
  5. Protocol (always TCP here)

• Often, Client Port “randomly” assigned
  – Done by OS during client socket setup

• Server Port often “well known”
  – 80 (web), 443 (secure web), 25 (sendmail), etc
  – Well-known ports from 0—1023
Recall: Server Protocol (v3)

// Socket setup code elided...
listen(server_socket, MAX_QUEUE);
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) {
        close(server_socket);
        serve_client(conn_socket);
        close(conn_socket);
        exit(0);
    } else {
        close(conn_socket);
        // wait(NULL);
    }
}
close(server_socket);
Recall: Multiplexing Processes: The Process Control Block

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, ...)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, ...
  - Execution time, ...
  - Memory space, translation, ...

- Kernel Scheduler maintains a data structure containing the PCBs
  - Give out CPU to different processes
  - This is a Policy Decision

- Give out non-CPU resources
  - Memory/IO
  - Another policy decision
Recall: CPU Switch From Process A to Process B

![Diagram showing the switch from process to system mode and back to user mode.]

- **Kernel/System Mode**
  - Process $P_0$
    - Executing
    - Interrupt or system call
    - Save state into PCB$_0$
    - Reload state from PCB$_1$
  - Operating system
    - Idle
    - Executing
  - Process $P_1$
    - Idle
    - Executing

- **User Mode**
  - Process $P_0$
    - Executing
  - Process $P_1$
    - Executing

Note: PCB stands for Process Control Block.
Recall: Lifecycle of a Process or Thread

- As a process executes, it changes state:
  - **new**: The process/thread is being created
  - **ready**: The process is waiting to run
  - **running**: Instructions are being executed
  - **waiting**: Process waiting for some event to occur
  - **terminated**: The process has finished execution
Recall: Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?
### Recall: Shared vs. Per-Thread State

<table>
<thead>
<tr>
<th>Shared State</th>
<th>Per–Thread State</th>
<th>Per–Thread State</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heap</strong></td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td><strong>Global Variables</strong></td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td><strong>Code</strong></td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td></td>
<td>Stack</td>
<td>Stack</td>
</tr>
</tbody>
</table>
The Core of Concurrency: the Dispatch Loop

• Conceptually, the scheduling loop of the operating system looks as follows:

   Loop {
     RunThread();
     ChooseNextThread();
     SaveStateOfCPU(curTCB);
     LoadStateOfCPU(newTCB);
   }

• This is an *infinite* loop
  – One could argue that this is all that the OS does

• Should we ever exit this loop???
  – When would that be?
Running a thread

Consider first portion: `RunThread()`

• How do I run a thread?
  – Load its state (registers, PC, stack pointer) into CPU
  – Load environment (virtual memory space, etc)
  – Jump to the PC

• How does the dispatcher get control back?
  – Internal events: thread returns control voluntarily
  – External events: thread gets preempted
Internal Events

- Blocking on I/O
  - The act of requesting I/O implicitly yields the CPU
- Waiting on a “signal” from other thread
  - Thread asks to wait and thus yields the CPU
- Thread executes a `yield()`
  - Thread volunteers to give up CPU

```c
computePI() {
  while(TRUE) {
    ComputeNextDigit();
    yield();
  }
}
```
Stack for Yielding Thread

• How do we run a new thread?

```c
run_new_thread() {
    newThread = PickNewThread();
    switch(curThread, newThread);
    ThreadHouseKeeping(); /* Do any cleanup */
}
```

• How does dispatcher switch to a new thread?
  – Save anything next thread may trash: PC, regs, stack pointer
  – Maintain isolation for each thread
What Do the Stacks Look Like?

• Consider the following code blocks:

```java
proc A() {
    B();
}
proc B() {
    while(TRUE) {
        yield();
    }
}
```

• Suppose we have 2 threads:
  – Threads S and T

Thread S's switch returns to Thread T's (and vice versa)
Saving/Restoring state (often called “Context Switch")

Switch(tCur,tNew) {
    /* Unload old thread */
    TCB[tCur].regs.r7 = CPU.r7;
    ...
    TCB[tCur].regs.r0 = CPU.r0;
    TCB[tCur].regs.sp = CPU.sp;
    TCB[tCur].regs.retpc = CPU.retpc; /*return addr*/

    /* Load and execute new thread */
    CPU.r7 = TCB[tNew].regs.r7;
    ...
    CPU.r0 = TCB[tNew].regs.r0;
    CPU.sp = TCB[tNew].regs.sp;
    CPU.retpc = TCB[tNew].regs.retpc;
    return; /* Return to CPU.retpc */
}
Switch Details (continued)

• What if you make a mistake in implementing switch?
  – Suppose you forget to save/restore register 32
  – Get intermittent failures depending on when context switch occurred and whether new thread uses register 32
  – System will give wrong result without warning

• Can you devise an exhaustive test to test switch code?
  – No! Too many combinations and inter-leavings

• Cautionary tale:
  – For speed, Topaz kernel saved one instruction in switch()
  – Carefully documented! Only works as long as kernel size < 1MB
  – What happened?
    » Time passed, People forgot
    » Later, they added features to kernel (no one removes features!)
    » Very weird behavior started happening
  – Moral of story: Design for simplicity
Administrivia

• Project 1 in full swing! Released on Saturday!
  – We expect that your design document will give intuitions behind your designs, not just a dump of pseudo-code
  – Think of this you are in a company and your TA is you manager
• Paradox: need code for design document?
  – Not full code, just enough prove you have thought through complexities of design
• Should be attending your permanent discussion section!
  – Discussion section attendance is mandatory, but don’t come if sick!!
    » Email your TA if you cannot come to your discussion for a valid reason.
• Midterm 1: September 27th, 7-9PM (Two weeks from today!)
  – Fill out conflict request by Friday!
Are we still switching contexts with previous examples?

- Yes, but much cheaper than switching processes
  - No need to change address space
- Some numbers from Linux:
  - Frequency of context switch: 10-100ms
  - Switching between processes: 3-4 μsec.
  - Switching between threads: 100 ns
- Even cheaper: switch threads (using “yield”) in user-space!

What we are talking about in Today’s lecture
What happens when thread blocks on I/O?

- What happens when a thread requests a block of data from the file system?
  - User code invokes a system call
  - Read operation is initiated
  - Run new thread/switch
- Thread communication similar
  - Wait for Signal/Join
  - Networking
External Events

• What happens if thread never does any I/O, never waits, and never yields control?
  – Could the ComputePI program grab all resources and never release the processor?
    » What if it didn’t print to console?
  – Must find way that dispatcher can regain control!

• Answer: utilize external events
  – Interrupts: signals from hardware or software that stop the running code and jump to kernel
  – Timer: like an alarm clock that goes off every some milliseconds

• If we make sure that external events occur frequently enough, can ensure dispatcher runs
Recall: Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Interrupt identity specified with ID line
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled
Example: Network Interrupt

- An interrupt is a hardware-invoked context switch
  - No separate step to choose what to run next
  - Always run the interrupt handler immediately
Use of Timer Interrupt to Return Control

• Solution to our dispatcher problem
  – Use the timer interrupt to force scheduling decisions

• Timer Interrupt routine:

```c
TimerInterrupt() {
    DoPeriodicHouseKeeping();
    run_new_thread();
}
```
ThreadFork() \( \text{Create a New Thread} \)

- **ThreadFork()** is a user-level procedure that creates a new thread and places it on ready queue.

- Arguments to **ThreadFork()**
  - Pointer to application routine (\( \text{fcnPtr} \))
  - Pointer to array of arguments (\( \text{fcnArgPtr} \))
  - Size of stack to allocate

- Implementation
  - Sanity check arguments
  - Enter Kernel-mode and Sanity Check arguments again
  - Allocate new Stack and TCB
  - Initialize TCB and place on ready list (Runnable)
How do we initialize TCB and Stack?

• Initialize Register fields of TCB
  – Stack pointer made to point at stack
  – PC return address ⇒ OS (asm) routine \texttt{ThreadRoot()}
  – Two arg registers (a0 and a1) initialized to \texttt{fcnPtr} and \texttt{fcnArgPtr}, respectively

• Initialize stack data?
  – Minimal initialization ⇒ setup return to go to beginning of \texttt{ThreadRoot()}
    » Important part of stack frame is in registers for RISC-V (ra)
    » X86: need to push a return address on stack
  – Think of stack frame as just before body of \texttt{ThreadRoot()} really gets started
How does Thread get started?

- Eventually, `run_new_thread()` will select this TCB and return into the beginning of `ThreadRoot()`.
  - This really starts the new thread.
How does a thread get started?

- How do we make a new thread?
  - Setup TCB/kernel thread to point at new user stack and ThreadRoot code
  - Put pointers to start function and args in registers or top of stack
    » This depends heavily on the calling convention (i.e. RISC-V vs x86)
- Eventually, `run_new_thread()` will select this TCB and return into beginning of `ThreadRoot()`
  - This really starts the new thread

```
SetupNewThread(tNew) {
  ...
  TCB[tNew].regs.sp = newStackPtr;
  TCB[tNew].regs.retpc = &ThreadRoot;
  TCB[tNew].regs.r0 = fcnPtr
  TCB[tNew].regs.r1 = fcnArgPtr
}
```

![Diagram showing thread creation process]
What does ThreadRoot() look like?

• **ThreadRoot()** is the root for the thread routine:

  ```c
  ThreadRoot(fcnPTR, fcnArgPtr) {
    DoStartupHousekeeping();
    UserModeSwitch(); /* enter user mode */
    Call fcnPtr(fcnArgPtr);
    ThreadFinish();
  }
  ```

• Startup Housekeeping
  – Includes things like recording start time of thread
  – Other statistics

• Stack will grow and shrink with execution of thread

• Final return from thread returns into **ThreadRoot()** which calls **ThreadFinish()**
  – **ThreadFinish()** wake up sleeping threads
Processes vs. Threads: One Core

- **Switch overhead:**
  - Same process: low
  - Different proc.: high
- **Protection**
  - Same proc: low
  - Different proc: high
- **Sharing overhead**
  - Same proc: low
  - Different proc: high
- **Parallelism:** no

---

**Diagram:**
- **Process I**
  - Threads
  - CPU state
  - IO state
  - Mem.
- **Process N**
  - Threads
  - CPU state
  - IO state
  - Mem.
- **CPU sched.**
- **OS**
- **CPU (1 core)**
- **I thread at a time**
Processes vs. Threads: MultiCore

- Switch overhead:
  - Same process: low
  - Different proc.: high
- Protection
  - Same proc: low
  - Different proc: high
- Sharing overhead
  - Same proc: low
  - Different proc, simultaneous core: medium
  - Different proc, offloaded core: high
- Parallelism: yes
Recall: Simultaneous MultiThreading/Hyperthreading

- Hardware scheduling technique
  - Superscalar processors can execute multiple instructions that are independent.
  - Hyperthreading duplicates register state to make a second “thread,” allowing more instructions to run.
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!

- Original technique called “Simultaneous Multithreading”
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5
Processes vs. Threads: Hyper-Threading

- Switch overhead between hardware-threads: very-low (done in hardware)
- Contention for ALUs/FPUs may hurt performance
## Threads vs Address Spaces: Options

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
</tr>
<tr>
<td>Many</td>
<td>Many (Geoworks, VxWorks, JavaOS, etc)</td>
<td>Embedded systems</td>
<td>Mach, OS/2, Linux</td>
</tr>
<tr>
<td></td>
<td>JavaOS, Pilot(PC)</td>
<td>Win NT to XP, Solaris, HP-UX, OS X</td>
<td></td>
</tr>
</tbody>
</table>

- Most operating systems have either
  - One or many address spaces
  - One or many threads per address space
Multiprocessing vs Multiprogramming

• Some Definitions:
  – 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 and interleaving: FIFO, Random, …
  – Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks
Correctness for systems with concurrent threads

• If dispatcher can schedule threads in any way, programs must work under all circumstances
  – Can you test for this?
  – How can you know if your program works?

• Independent Threads:
  – No state shared with other threads
  – Deterministic $\Rightarrow$ Input state determines results
  – Reproducible $\Rightarrow$ Can recreate Starting Conditions, I/O
  – Scheduling order doesn’t matter (if `switch()` works!!!)

• Cooperating Threads:
  – Shared State between multiple threads
  – Non-deterministic
  – Non-reproducible

• Non-deterministic and non-reproducible means that bugs can be intermittent
  – Sometimes called “Heisenbugs”
Interactions Complicate Debugging

• Is any program truly independent?
  – Every process shares the file system, OS resources, network, etc
  – Extreme example: buggy device driver causes thread A to crash “independent thread” B

• You probably don’t realize how much you depend on reproducibility:
  – Example: Evil C compiler
    » Modifies files behind your back by inserting errors into C program unless you insert debugging code
  – Example: Debugging statements can overrun stack

• Non-deterministic errors are really difficult to find
  – Example: Memory layout of kernel+user programs
    » depends on scheduling, which depends on timer/other things
    » Original UNIX had a bunch of non-deterministic errors
  – Example: Something which does interesting I/O
    » User typing of letters used to help generate secure keys
Why allow cooperating threads?

• People cooperate; computers help/enhance people’s lives, so computers must cooperate
  – By analogy, the non-reproducibility/non-determinism of people is a notable problem for “carefully laid plans”

• Advantage 1: Share resources
  – One computer; many users
  – One bank balance, many ATMs
    » What if ATMs were only updated at night?
  – Embedded systems (robot control: coordinate arm & hand)

• Advantage 2: Speedup
  – Overlap I/O and computation
    » Many different file systems do read-ahead
  – Multiprocessors – chop up program into parallel pieces

• Advantage 3: Modularity
  – More important than you might think
  – Chop large problem up into simpler pieces
    » To compile, for instance, gcc calls cpp | cc1 | cc2 | as | ld
    » Makes system easier to extend
Recall: High-level Example: Web Server

- Server must handle many requests
- Non-cooperating version:
  
  ```
  serverLoop() {
    con = AcceptCon();
    ProcessFork(ServiceWebPage(), con);
  }
  ```

- What are some disadvantages of this technique?
Recall: Threaded Web Server

• Now, use a single process
• Multithreaded (cooperating) version:
  
  serverLoop() {
      connection = AcceptCon();
      ThreadFork(ServiceWebPage(), connection);
  }

• Looks almost the same, but has many advantages:
  – Can share file caches kept in memory, results of CGI scripts, other things
  – Threads are much cheaper to create than processes, so this has a lower per-request overhead

• Question: would a user-level (say one-to-many) thread package make sense here?
  – When one request blocks on disk, all block…

• What about Denial of Service attacks or digg / Slash-dot effects?
Thread Pools: Bounded Concurrency

- Problem with previous version: Unbounded Threads
  - When web-site becomes too popular – throughput sinks
- Instead, allocate a bounded “pool” of worker threads, representing the maximum level of multiprogramming

```java
master() {
    allocThreads(worker, queue);
    while (TRUE) {
        con = AcceptCon();
        Enqueue(queue, con);
        wakeUp(queue);
    }
}

worker(queue) {
    while (TRUE) {
        con = Dequeue(queue);
        if (con == null)
            sleepOn(queue);
        else
            ServiceWebPage(con);
    }
}
```
Correctness with Concurrent Threads?

• Non-determinism:
  – Scheduler can run threads in any order
  – Scheduler can switch threads at any time
  – This can make testing very difficult

• *Independent Threads*
  – No state shared with other threads
  – Deterministic, reproducible conditions

• *Cooperating Threads*
  – Shared state between multiple threads

• Goal: Correctness by Design
ATM Bank Server

- ATM server problem:
  - 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:

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

• How could we speed this up?
  – More than one request being processed at once
  – Event driven (overlap computation and I/O)
  – Multiple threads (multi-proc, or overlap comp and I/O)
Event Driven Version of ATM server

• 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

• Example

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

  – This technique is used for graphical programming

• Complication:
  – What if we missed a blocking I/O step?
  – What if we have to split code into hundreds of pieces which could be blocking?
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:
  ```c
  Deposit(acctId, amount) {
    acct = GetAccount(acctId); /* May use disk I/O */
    acct->balance += amount;
    StoreAccount(acct); /* Involves disk I/O */
  }
  ```
- Unfortunately, shared state can get corrupted:
  ```c
  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
  ```
Recall: 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:

  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 are the possible values of \( x \)?

• Or, what are the possible values of \( x \) below?

  Thread A  Thread B
  \[ x = 1; \]  \[ x = 2; \]

  – \( X \) could be 1 or 2 (non-deterministic!)
  – Could even be 3 for serial processors:
    » Thread A writes 0001, B writes 0010 ➔ scheduling order ABABABBA yields 3!
Atomic Operations

• To understand a concurrent program, we need to know what the underlying indivisible operations are!

• Atomic Operation: an operation that always runs to completion or not at all
  – It is \textit{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

• 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
**Locks**

- **Lock**: 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
    » After this returns, we say the calling thread holds the lock
  - `release(&mylock)` – mark lock as free
    » Should only be called by a thread that currently holds the lock
    » After this returns, the calling thread no longer holds the lock
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
  }

  • Must use SAME lock (mylock) with all of the methods (Withdraw, etc…)
    – Shared with all threads!

  Threads serialized by lock through critical section.
  Only one thread at a time
Conclusion

• Processes have two parts
  – Threads (Concurrency)
  – Address Spaces (Protection)

• Various textbooks talk about *processes*
  – When this concerns concurrency, really talking about thread portion of a process
  – When this concerns protection, talking about address space portion of a process

• Concurrent threads are a very useful abstraction
  – Allow transparent overlapping of computation and I/O
  – Allow use of parallel processing when available

• Concurrent threads introduce problems when accessing shared data
  – Programs must be insensitive to arbitrary interleavings
  – Without careful design, shared variables can become completely inconsistent