Recall: Inter-Process Communication (IPC)

- Mechanism to create communication channel between distinct processes
  - Same or different machines, same or different programming language...
- Requires serialization format understood by both
- Failure in one process isolated from the other
  - Sharing is done in a controlled way through IPC
  - Still have to be careful handling what is received via IPC
- Later in the term: Many uses and interaction patterns
  - Logging process, window management, ...
  - Potentially allows us to move some system functions outside of kernel to userspace

Recall: POSIX/Unix PIPE

write(wfd, wbuf, wlen);

n = read(rfd, rbuf, rmax);

- Memory Buffer is finite:
  - If producer (A) tries to write when buffer full, it blocks (Put sleep until space)
  - If consumer (B) tries to read when buffer empty, it blocks (Put to sleep until data)

int pipe(int files[2]);

- Allocates two new file descriptors in the process
- Writes to files[1] read from files[0]
- Implemented as a fixed-size queue
Recall: Socket Endpoint for Communication

- **Key Idea:** Communication across the world looks like File I/O
  
  ```
  write(wfd, wbbuf, wlen);
  
  n = read(rfd, rbbuf, rmax);
  ```

  - Sockets: Bidirectional Endpoint for Communication
    - Queues to temporarily hold results
    - Queues are NOT Pipes!
  - Connection: Two Sockets Connected Over the network ⇒ IPC over network!
    - How to `open()`?
    - What is the namespace?
    - How are they connected in time?

Recall: Connection Setup over 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 (v1)

```c
// Create socket to listen for client connections
char *port_name;
struct addrinfo *server = setup_address(port_name);
int server_socket = socket(server->ai_family, 
  server->ai_socktype, server->ai_protocol);
// Bind socket to specific port
bind(server_socket, server->ai_addr, server->ai_addrlen);
// Start listening for new client connections
listen(server_socket, MAX_QUEUE);

while (1) {
  // Accept a new client connection, obtaining a new socket
  int conn_socket = accept(server_socket, NULL, NULL);
  serve_client(conn_socket);
  close(conn_socket);
}

close(server_socket);
```

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
Context Switch

- Privilege Level: 0 - sys
- Privilege Level: 3 - user
- Privilege Level: 3 - user

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

Scheduling: All About Queues

- PCBs move from queue to queue
- **Scheduling**: which order to remove from queue
  - Much more on this soon

Ready Queue And Various I/O Device Queues

- Process not running ⇒ PCB is in some scheduler queue
  - Separate queue for each device/signal/condition
  - Each queue can have a different scheduler policy

9/16/20
Kubiatowicz CS162 © UCB Fall 2020
Lec 6.9

9/16/20
Kubiatowicz CS162 © UCB Fall 2020
Lec 6.10

9/16/20
Kubiatowicz CS162 © UCB Fall 2020
Lec 6.11

9/16/20
Kubiatowicz CS162 © UCB Fall 2020
Lec 6.12
Scheduler

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide …
  - Fairness or
  - Realtime guarantees or
  - Latency optimization or ..

```
if ( readyProcesses(PCBs) ) {
  nextPCB = selectProcess(PCBs);
  run( nextPCB );
} else {
  run_idle_process();
}
```

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></td>
<td>Thread Control Block (TCB)</td>
<td>Thread Control Block (TCB)</td>
</tr>
<tr>
<td></td>
<td>Stack Information</td>
<td>Stack Information</td>
</tr>
<tr>
<td></td>
<td>Saved Registers</td>
<td>Saved Registers</td>
</tr>
<tr>
<td></td>
<td>Thread Metadata</td>
<td>Thread Metadata</td>
</tr>
<tr>
<td>Heap</td>
<td>Stack</td>
<td>Stack</td>
</tr>
<tr>
<td>Global Variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td></td>
<td></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?
Administrivia

• Homework 1 due Today
• Project 1 in full swing!
  – 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 your 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!
  – Remember to turn on your camera in Zoom
  – Discussion section attendance is mandatory
• Midterm 1: October 1st, 5-7PM (Three weeks from tomorrow!)
  – We understand that this partially conflicts with CS170, but those of you in CS170 can start that exam after 7PM (according to CS170 staff)
  – Video Proctored, No curve, Use of computer to answer questions
  – More details as we get closer to exam

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
  
  computePI() {
    while(TRUE) {
      ComputeNextDigit();
      yield();
    }
  }

Recall: POSIX API for Threads: 

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg);
  – thread is created executing start_routine with arg as its sole argument.
  – return is implicit call to pthread_exit
void pthread_exit(void *value_ptr);
  – terminates the thread and makes value_ptr available to any successful join
int pthread_join(pthread_t thread, void **value_ptr);
  – suspends execution of the calling thread until the target thread terminates.
  – On return with a non-NULL value_ptr the value passed to pthread_exit by the terminating thread is made available in the location referenced by value_ptr.
void pthread_yield(void);
void sched_yield(void);
  – Current thread yields (gives up) CPU so that another thread can run
Stack for Yielding Thread

- How do we run a new thread?
  - `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:

  ```
  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)

- TCB+Stacks (user/kernel) contains complete restartable state of Thread!
  - Can put it on any queue for later revival!

- 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
Aren't we still switching contexts?

- 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!

**Processes vs. Threads**

- 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

**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

**What we are talking about in Today's lecture**

- Simple One-to-One Threading Model
- Many-to-One
- Many-to-Many

**Processes vs. Threads**

- Process 1
  - Threads
  - Mem.
  - IO state
  - CPU state
- Process N
  - Threads
  - Mem.
  - IO state
  - CPU state
- OS
- CPU sched
- 4 threads at a time
- Core 1
- Core 2
- Core 3
- Core 4

- Simultaneous MultiThread/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

**Simultaneous MultiThread/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!
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

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

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
Use of Timer Interrupt to Return Control

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

Timer Interrupt routine:

```
TimerInterrupt() {
  DoPeriodicHouseKeeping();
  run_new_thread();
}
```

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 ThreadRoot()
  – Two arg registers (a0 and a1) initialized to fcnPtr and fcnArgPtr, respectively

• Initialize stack data?
  – No. Important part of stack frame is in registers (ra)
  – Think of stack frame as just before body of ThreadRoot() really gets started

```
ThreadRoot stub
```

How does Thread get started?

• 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
}
ThreadRoot stub
```

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
  – 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
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()
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 ...
}
```

- 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();
  }
}
```

- What if we missed a blocking I/O step?
- What if we have to split code into hundreds of pieces which could be blocking?
- This technique is used for graphical programming

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:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>load r1, acct-&gt;balance</td>
<td>load r1, acct-&gt;balance</td>
</tr>
<tr>
<td>add r1, amount1</td>
<td>add r1, amount2</td>
</tr>
<tr>
<td>store r1, acct-&gt;balance</td>
<td>store r1, acct-&gt;balance</td>
</tr>
</tbody>
</table>

Problem is at the Lowest Level

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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 2;</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>y = 2;</td>
</tr>
<tr>
<td>x = y+1;</td>
<td>y = y*2;</td>
</tr>
</tbody>
</table>

- What are the possible values of x?
- Or, what are the possible values of x below?

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = 1;</td>
<td>x = 2;</td>
</tr>
</tbody>
</table>

- 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 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

Recall: 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!

Recall: Definitions

• Synchronization: using atomic operations to ensure cooperation between threads
  – For now, only loads and stores are atomic
  – We are going to show that its hard to build anything useful with only reads and writes
• Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  – One thread excludes the other while doing its task
• 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
  – Critical section is the result of mutual exclusion
  – Critical section and mutual exclusion are two ways of describing the same thing
Another Concurrent Program Example

- Two threads, A and B, compete with each other
  - One tries to increment a shared counter
  - The other tries to decrement the counter

```plaintext
Thread A
i = 0;
while (i < 10) { acquire(&mylock) i = i + 1;
release(&mylock) printf("A wins!");
}

Thread B
i = 0;
while (i > -10) { acquire(&mylock) i = i - 1;
release(&mylock) printf("B wins!");
}
```

- Assume that memory loads and stores are atomic, but incrementing and decrementing are not atomic
  - No difference between: "i=i+1" and "i++"
  - Same instruction sequence, the ++ operator is just syntactic sugar

- Who wins? Could be either
- Is it guaranteed that someone wins? Why or why not?
- What if both threads have their own CPU running at same speed? Is it guaranteed that it goes on forever?

Hand Simulation Multiprocessor Example

- Inner loop looks like this:

```plaintext
Thread A
r1=0 load r1, M[i] r1=0 load r1, M[i]
r1=1 add r1, r1, 1 r1=-1 sub r1, r1, 1
M[i]=1 store r1, M[i] M[i]=-1 store r1, M[i]
```

- Hand Simulation:
  - And we're off. A gets off to an early start
  - B says "hmph, better go fast" and tries really hard
  - A goes ahead and writes "1"
  - B goes and writes "-1"
  - A says "HUH??? I could have sworn I put a 1 there"

- Uncontrolled race condition: two threads attempting to access same data simultaneously with one of them performing a write
  - Here "simultaneous" is defined even with one CPU as "could access at same time if only there were two CPUs"

So – does this fix it?

- Put locks around increment/decrement:

```plaintext
Thread A
i = 0;
while (i < 10) { acquire(&mylock) i = i + 1;
release(&mylock) printf("A wins!");
}

Thread B
i = 0;
while (i > -10) { acquire(&mylock) i = i - 1;
release(&mylock) printf("B wins!");
}
```

- What does this do? Is it better???
- Each increment or decrement operation is now atomic. Good!
  - Technically, no race conditions, since lock prevents simultaneous reads/writes
- Program is likely still broken. Not so good...
  - May or may not be what you intended (probably not)
  - Still unclear who wins – it is a nondeterministic result: different on each run
- When might something like this make sense?
  - If each thread needed to get a unique integer for some reason

Recall: Red-Black tree example

- Here, the Lock is associated with the root of the tree
  - Restricts parallelism but makes sure that tree always consistent
  - No races at the operation level
- Threads are exchange information through a consistent data structure
- Could you make it faster with one lock per node? Perhaps, but must be careful!
  - Need to define invariants that are always true despite many simultaneous threads...
Concurrency is Hard!

- Even for practicing engineers trying to write mission-critical, bulletproof code!
  - 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: Radiation Therapy Machine with Unintended Overdoses (reading on course site)
  - Concurrency errors caused the death of a number of patients by misconfiguring the radiation production
  - Improper synchronization between input from operators and positioning software

- Mars Pathfinder Priority Inversion (JPL Account)
- Toyota Uncontrolled Acceleration (CMU Talk)
  - 256.6K Lines of C Code, ~9-11K global variables
  - Inconsistent mutual exclusion on reads/writes

Producer-Consumer with a Bounded Buffer

- Problem Definition
  - Producer(s) put things into a shared buffer
  - Consumer(s) take them out
  - Need synchronization to coordinate producer/consumer

- Don’t want producer and consumer to have to work in lockstep, so put a fixed-size buffer between them
  - Need to synchronize access to this buffer
  - Producer needs to wait if buffer is full
  - Consumer needs to wait if buffer is empty

- Example 1: GCC compiler
  - `cpp | cc1 | cc2 | as | ld`

- Example 2: Coke machine
  - Producer can put limited number of Cokes in machine
  - Consumer can’t take Cokes out if machine is empty

- Others: Web servers, Routers, ...

Circular Buffer Data Structure (sequential case)

```
typedef struct buf {
  int write_index;
  int read_index;
  <type> *entries[BUFSIZE];
} buf_t;
```

- Insert: write & bump write ptr (enqueue)
- Remove: read & bump read ptr (dequeue)
- How to tell if Full (on insert) Empty (on remove)?
- And what do you do if it is?
- What needs to be atomic?

Circular Buffer – first cut

```
mutex buf_lock = <initially unlocked>

Producer(item) {
  acquire(&buf_lock);
  while (buffer full) {}; // Wait for a free slot
  enqueue(item);
  release(&buf_lock);
}

Consumer() {
  acquire(&buf_lock);
  while (buffer empty) {}; // Wait for arrival
  item = dequeue();
  release(&buf_lock);
  return item
}
```

Will we ever come out of the wait loop?
Circular Buffer – 2nd cut

mutex buf_lock = <initially unlocked>

Producer(item)
{
    acquire(&buf_lock);
    while (buffer full)
    {
        release(&buf_lock);
        acquire(&buf_lock);
    }
    enqueue(item);
    release(&buf_lock);
}

Consumer()
{
    acquire(&buf_lock);
    while (buffer empty)
    {
        release(&buf_lock);
        acquire(&buf_lock);
    }
    item = dequeue();
    release(&buf_lock);
    return item
}

What happens when one is waiting for the other?
- Multiple cores?
- Single core?

Higher-level Primitives than Locks

- What is right abstraction for synchronizing threads that share memory?
  - Want as high a level primitive as possible
- Good primitives and practices important!
  - Since execution is not entirely sequential, really hard to find bugs, since they happen rarely
  - UNIX is pretty stable now, but up until about mid-80s (10 years after started), systems running UNIX would crash every week or so – concurrency bugs
- Synchronization is a way of coordinating multiple concurrent activities that are using shared state
  - This lecture and the next presents some ways of structuring sharing

Recall: Semaphores

- Semaphores are a kind of generalized lock
  - First defined by Dijkstra in late 60s
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - Down() or P(): an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - Up() or V(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation
- Note that P() stands for "proberen" (to test) and V() stands for "verhogen" (to increment) in Dutch

Semaphores Like Integers Except…

- Semaphores are like integers, except:
  - No negative values
  - Only operations allowed are P and V – can’t read or write value, except initially
  - Operations must be atomic
    - Two P’s together can’t decrement value below zero
    - Thread going to sleep in P won’t miss wakeup from V – even if both happen at same time
- POSIX adds ability to read value, but technically not part of proper interface!
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:
Two Uses of Semaphores

Mutual Exclusion (initial value = 1)
- Also called “Binary Semaphore” or “mutex”.
- Can be used for mutual exclusion, just like a lock:
  ```c
  semaP(&mysem);
  // Critical section goes here
  semaV(&mysem);
  ```

Scheduling Constraints (initial value = 0)
- Allow thread 1 to wait for a signal from thread 2
  - thread 2 schedules thread 1 when a given event occurs
- Example: suppose you had to implement ThreadJoin which must wait for thread to terminate:
  ```c
  Initial value of semaphore = 0
  ThreadJoin {
    semaP(&mysem);
  }
  ThreadFinish {
    semaV(&mysem);
  }
  ```

Revisit Bounded Buffer: Correctness constraints for solution
- Correctness Constraints:
  - Consumer must wait for producer to fill buffers, if none full (scheduling constraint)
  - Producer must wait for consumer to empty buffers, if all full (scheduling constraint)
  - Only one thread can manipulate buffer queue at a time (mutual exclusion)
- Remember why we need mutual exclusion
  - Because computers are stupid
  - Imagine if in real life: the delivery person is filling the machine and somebody comes up and tries to stick their money into the machine
- General rule of thumb: Use a separate semaphore for each constraint
  - Semaphore fullBuffers; // consumer's constraint
  - Semaphore emptyBuffers; // producer's constraint
  - Semaphore mutex; // mutual exclusion

Full Solution to Bounded Buffer (coke machine)
- Semaphore fullSlots = 0; // Initially, no coke
- Semaphore emptySlots = bufSize; // Initially, num empty slots
- Semaphore mutex = 1; // No one using machine

Producer(item)
```c
  semaP(&emptySlots);
  // Wait until space
  semaP(&mutex);
  Enqueue(item);
  semaV(&mutex);
  semaV(&fullSlots);
} 
```

Consumer()
```c
  semaP(&fullSlots);
  // Check if there's a coke
  semaP(&mutex);
  item = Dequeue();
  semaV(&mutex);
  semaV(&emptySlots);
  // tell producer need more
  return item;
} 
```

Discussion about Solution
- Why asymmetry?
  - Producer does: `semaP(&emptyBuffer), semaV(&fullBuffer)`
  - Consumer does: `semaP(&fullBuffer), semaV(&emptyBuffer)`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency
- What if we have 2 producers or 2 consumers?
  - Do we need to change anything?
    - Decrease # of empty slots
    - Increase # of occupied slots
Where are we going with synchronization?

- We are going to implement various higher-level synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide primitives useful at user-level
- Talk about how to structure programs so that they are correct
  - Under any scheduling and number of processors

Conclusion

- Concurrency accomplished by multiplexing CPU time:
  - Unloading current thread (PC, registers)
  - Loading new thread (PC, registers)
  - Such context switching may be voluntary (yield(), I/O) or involuntary (interrupts)
- TCB + Stacks hold complete state of thread for restarting
- Atomic Operation: an operation that always runs to completion or not at all
- Synchronization: using atomic operations to ensure cooperation between threads
- Mutual Exclusion: ensuring that only one thread does a particular thing at a time
  - One thread excludes the other while doing its task
- 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: synchronization mechanism for enforcing mutual exclusion on critical sections to construct atomic operations
- Semaphores: synchronization mechanism for enforcing resource constraints