Final Exam
Logistics

- **100 minutes (half an hour longer than midterm)**
  - Covers all materials, emphasizes second half
  - You may bring 1 page (8.5 by 11, two-sided) of notes. It must be hand-written.

- **Disclaimer: this review is not complete.**
  - Not all exam materials are mentioned by this review!
  - We are not even mentioning things from the first half of the semester.
General Tips

- Read through all questions, do what you think are the easier ones first.
- Make sure you have completed and understood all the homeworks, recitation exercises, labs, quizzes and exams.
- Use Piazza! Have discussions there with your classmates.
Machine-Level Programming
Machine-Level Programs

- Basic information about x86-64
- Source, assembly, object & machine code
- General purpose registers
- Data movement instructions.
  - `movq %rdi, %rsi`
  - `movq (%rdi), %rsi`
Memory Addressing

- General form

$$D(Rb, Ri, S) \quad \text{Mem}[ \quad D + \text{Reg}[Rb] + \text{Reg}[Ri] \ast S \quad ]$$

- D: Constant “displacement”
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for %rsp
- S: Scale: 1, 2, 4, or 8

- Special cases: you can omit certain arguments if not needed.

$$\quad (Rb, Ri) \quad \text{Mem}[ \quad \text{Reg}[Rb] + \text{Reg}[Ri] \quad ]$$

$$\quad D(Rb, Ri) \quad \text{Mem}[ \quad D + \text{Reg}[Rb] + \text{Reg}[Ri] \quad ]$$

$$\quad (Rb, Ri, S) \quad \text{Mem}[ \quad \text{Reg}[Rb] + \text{Reg}[Ri] \ast S \quad ]$$
Address Computation

- `leaq src, dest`
  - `src` is an address computation expression
  - set `dest` to address denoted by expression

- use case 1
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`

- Example
  ```c
  char* a2(char* x) {
      return &x[2];
  }
  ```
  ```assembly
  leaq 2(%rdi), %rax  # return &x[2]
  ret
  ```
### Arithmetic Operations

- **Two Operand Instructions:**

<table>
<thead>
<tr>
<th>Format</th>
<th>Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addq</code> src, dest</td>
<td>dest = dest + src</td>
</tr>
<tr>
<td><code>subq</code> src, dest</td>
<td>dest = dest − src</td>
</tr>
<tr>
<td><code>imulq</code> src, dest</td>
<td>dest = dest * src</td>
</tr>
<tr>
<td><code>salq</code> src, dest</td>
<td>dest = dest &lt;&lt; src</td>
</tr>
<tr>
<td><code>sarq</code> src, dest</td>
<td>dest = dest &gt;&gt; src</td>
</tr>
<tr>
<td><code>shrq</code> src, dest</td>
<td>dest = dest &gt;&gt; src</td>
</tr>
<tr>
<td><code>xorq</code> src, dest</td>
<td>dest = dest ^ src</td>
</tr>
<tr>
<td><code>andq</code> src, dest</td>
<td>dest = dest &amp; src</td>
</tr>
<tr>
<td><code>orq</code> src, dest</td>
<td>dest = dest</td>
</tr>
</tbody>
</table>

- Watch out for argument order!
### Arithmetic Expression Example

```c
long arith (long x, long y, long z){
    long t1 = x + y;
    long t2 = z + t1;
    long t3 = x + 4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

**Noteworthy instructions:**
- **leaq**: “address” computation
- **salq**: shift
- **imulq**: integer multiplication

**Register Use(s):**

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument z</td>
</tr>
<tr>
<td>%rax</td>
<td>t1, t2, rval</td>
</tr>
<tr>
<td>%rdx</td>
<td>t4</td>
</tr>
<tr>
<td>%rcx</td>
<td>t5</td>
</tr>
</tbody>
</table>


Control

- Information about currently executing program…
  - temporary data ( %rax, … )
  - location of runtime stack ( %rsp )
  - location of current code point ( %rip )
  - status of recent tests ( CF, ZF, SF, OF )

Registers

<table>
<thead>
<tr>
<th>8rax</th>
<th>8r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>8rbx</td>
<td>8r9</td>
</tr>
<tr>
<td>8rcx</td>
<td>8r10</td>
</tr>
<tr>
<td>8rdx</td>
<td>8r11</td>
</tr>
<tr>
<td>8rsi</td>
<td>8r12</td>
</tr>
<tr>
<td>8rdi</td>
<td>8r13</td>
</tr>
<tr>
<td>8rsp</td>
<td>8r14</td>
</tr>
<tr>
<td>8rbp</td>
<td>8r15</td>
</tr>
</tbody>
</table>

Instruction

| CF | ZF | SF | OF |

Current stack ‘top’ Current instruction
Condition Codes (Implicit Setting)

- Single bit registers
  - CF  Carry Flag (for unsigned)
  - ZF  Zero Flag
  - SF  Sign Flag (for signed)
  - OF  Overflow Flag (for signed)

- Implicitly set (think of it as a side effect) by arithmetic operations
  - Example: \texttt{addq } src, dest \leftrightarrow b = a + b
    - CF set if carry out from most significant bit (unsigned overflow)
    - ZF set if t == 0
    - SF set if t < 0 (as signed)
    - OF set if two’s-complement (signed) overflow
      \((a > 0 && b > 0 && t < 0) \lor (a < 0 && b < 0 && t > 0)\)

- Not set by \texttt{leaq} instruction (!!!!)
Conditional Branching by Jumping

long absdiff(long x, long y) {
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}

absdiff:
    cmpq %rsi, %rdi # y, x
    jle .L4
    movq %rdi, %rax
    subq %rsi, %rax
    ret
.L4: # x <= y
    movq %rsi, %rax
    subq %rdi, %rax
    ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
long absdiff(long x, long y){
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rdx</td>
<td>Temp variable</td>
</tr>
</tbody>
</table>

absdiff:
  movq    %rdi, %rax  # x
  subq    %rsi, %rax  # if-val = x-y
  movq    %rsi, %rdx
  subq    %rdi, %rdx  # else-val = y-x
  cmpq    %rsi, %rdi  # %rsi = y, %rdi = x
  cmovle  %rdx, %rax  # if y <= x, result = else-val
  ret
“Do-While” Loop Compilation

```c
long pcount_goto(unsigned long x) {
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if (x) goto loop;
    return result;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

- movl     $0, %rax     #  result = 0
- .L2:     # loop:
- movq     %rdi, %rdx
- andq     $1, %rdx     #  t = x & 0x1
- addq     %rdx, %rax   #  result += t
- shrq     %rdi         #  x >>= 1
- jne      .L2          #  if (x) goto loop
- rep; ret

Note: some processors' branch predictors behave badly when a branch's target or fall-through is a `ret` instruction, and adding the `rep;` prefix avoids this.
Procedure Data Flow

Registers

First 6 arguments

%rdi
%rsi
%rdx
%rcx
%r8
%r9

Return value

%rax

Stack

* Only allocate stack space when needed

Arg n
Arg 8
Arg 7
**x86-64 Caller-saved Registers**

- **%rax**
  - Return value
  - Also caller-saved
  - Can be modified by procedure

- **%rdi, ..., %r9**
  - Arguments
  - Also caller-saved
  - Can be modified by procedure

- **%r10, %r11**
  - Caller-saved
  - Can be modified by procedure
x86-64 Callee-saved Registers

- `%rbx, %r12, %r13, %r14`
  - Callee-saved
  - Callee must save & restore

- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer

- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure
- **Current Stack Frame** (‘Top’ to ‘Bottom’)
  - *Local variables*
    - If can’t keep in registers
  - *Old frame pointer* (optional)

- **Caller Stack Frame**
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call
Memory
Smaller, faster, and costlier (per byte) storage devices

L0: CPU registers hold words retrieved from the L1 cache.

L1 cache (SRAM)
L1 cache holds cache lines retrieved from the L2 cache.

L2 cache (SRAM)
L2 cache holds cache lines retrieved from L3 cache.

L3 cache (SRAM)
L3 cache holds cache lines retrieved from main memory.

Main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.

Local secondary storage (local disks)
Local disks hold files retrieved from disks on remote servers.

Remote secondary storage (e.g., Web servers)
Locality

- **Principle of Locality**: Programs tend to use data and instructions with addresses near or equal to those they have used recently.

- **Temporal locality**: Recently referenced items are likely to be referenced again in the near future.

- **Spatial locality**: Items with nearby addresses tend to be referenced close together in time.
Cache Memories

- Cache memories are small, fast SRAM-based memories managed automatically in hardware
  - Hold frequently accessed blocks of main memory
- CPU looks first for data in cache
- Typical system structure:
Stride-n Access Patterns

- C arrays allocated in row-major order, contiguously

- Stepping through columns in one row:
  - for (i = 0; i < N; i++)
    sum += a[0][i];
  - accesses successive, contiguous elements

- Stepping through rows in one column:
  - for (i = 0; i < n; i++)
    sum += a[i][0];
  - accesses distant elements (stride-rowsize pattern)
  - no spatial locality!
Array Access

- **Basic Principle**
  \[ T \; A[N] ; \]
  - Array of data type \( T \) and length \( N \)
  - Identifier \( A \) can be used as a pointer to array element 0: type \( T^* \)

<table>
<thead>
<tr>
<th>Reference</th>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>val[4]</td>
<td>int</td>
<td>3</td>
</tr>
<tr>
<td>val</td>
<td>int*</td>
<td>( x )</td>
</tr>
<tr>
<td>val+1</td>
<td>int*</td>
<td>( x + 4 )</td>
</tr>
<tr>
<td>&amp;val[2]</td>
<td>int*</td>
<td>( x + 8 )</td>
</tr>
<tr>
<td>val[5]</td>
<td>int</td>
<td>??</td>
</tr>
<tr>
<td>*(val+1)</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>val + ( i )</td>
<td>int*</td>
<td>( x + 4 ; i )</td>
</tr>
</tbody>
</table>
Array Accessing Example

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference (%rdi,%rsi,4)

```c
int get_digit(int[] z, int digit) {
    return z[digit];
}
```
Structure Representation

- Structure represented as block of memory
  - Big enough to hold all of the fields

- Fields ordered according to declaration
  - Even if another ordering could yield a more compact representation
  - Machine-level program has no understanding of the structures in the source code

```c
struct rec {
    int a[4];
    long i;
    struct rec *next;
};
```
Structures & Alignment

- Unaligned Data

```
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- Aligned Data
  - Primitive data type requires $K$ bytes
  - Address must be multiple of $K$
Linking
What Do Linkers Do?

- **Step 1: Symbol resolution**
  - Programs define and reference symbols (global variables and functions):
    - `void swap() {...} /* define symbol swap */`
    - `swap(); /* reference symbol swap */`
    - `int *xp = &x; /* define symbol xp, reference x */`
  - Symbol definitions are stored in object file in symbol table.
  - Symbol table is an array of structs.
  - Each entry includes name, size, and location of symbol.
  - During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.
Linker Symbols

- **Global symbols**
  - Symbols defined by module $m$ that can be referenced by other modules.
  - E.g.: non-static C functions and non-static global variables.

- **External symbols**
  - Global symbols that are referenced by module $m$ but defined by some other module.

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module $m$.
  - E.g.: C functions and global variables with the static attribute.
  - *Local linker symbols are not local program variables*
Resolving Duplicate Symbol Definitions

- All symbol references must be mapped to 1 definition.
  - How to choose when there are duplicates?

- Taxonomy of symbol definitions aids decision.

- Program symbols are either strong or weak
  - **Strong**: procedures and initialized globals
  - **Weak**: uninitialized globals

```plaintext
p1.c
int foo=5;
p1() {
}
```

```plaintext
p2.c
int foo;
p2() {
}
```
Linker’s Symbol Rules

- Rule 1: Multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise: Linker error

- Rule 2: Given a strong symbol and multiple weak symbols, choose the strong symbol
  - References to the weak symbol resolve to the strong symbol

- Rule 3: If there are multiple weak symbols, pick any one
  - Can override this with gcc –fno-common
  - Probably a good idea to use above flag
What Do Linkers Do? (cont)

- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the .o files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.
When Can Linking Happen?

- Linking can happen at different times in a program’s lifetime:
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)
Exceptions & Processes
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

\[\begin{align*}
\text{Control flow} \\
\text{<startup>} \quad \text{<shutdown>}
\end{align*}\]

\[\text{Time} \quad \text{inst}_1 \quad \text{inst}_2 \quad \text{inst}_3 \quad \ldots \quad \text{inst}_n\]
Exceptions

- An exception is a transfer of control to the OS kernel in response to some event (i.e., change in processor state)
  - Examples of events: Divide by 0, page fault, I/O request completes, typing Ctrl-C
Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s *interrupt pin*
  - Handler returns to “next” instruction

- Examples:
  - *Timer interrupt*
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the kernel to take back control from user programs
  - *I/O interrupt from external device*
    - Arrival of a packet from a network
    - Arrival of data from a disk
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - *Traps*
    - Intentional
    - Example: *system calls*
    - Returns control to “next” instruction
  - *Faults*
    - Unintentional but possibly recoverable
    - Example: *page fault*
    - Either re-executes faulting (“current”) instruction or aborts
  - *Aborts*
    - Unintentional and unrecoverable
    - Example: *illegal memory access*
    - Aborts current program
System Call Example: Opening File

- User calls: `open(filename, options)`
- Calls `__open` function, which invokes system call instruction `syscall`

```
00000000000e5d70 <__open>:
...
00000000000e5d79: b8 02 00 00 00     mov $0x2,%rax  # open is syscall #2
00000000000e5d7e: 0f 05               syscall         # Return value in %rax
00000000000e5d80: 48 3d 01 f0 ff ff   cmp $0xffffffffffffff001, %rax
...  
00000000000e5dfa: c3                  retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi`, `%rsi`, `%rdx`, `%r10`, `%r8`, `%r9
- Return value in `%rax`
- Negative value is an error corresponding to negative errno
Processes

- A **process** is an instance of a running program.
  - One of the most successful ideas in computer science
  - Not the same as “program”

- **Process provided with two key abstractions by OS:**
  - *Logical control flow*
    - Each program seems to have exclusive use of the CPU
    - Provided by kernel mechanism called *context switching*
  - *Private address space*
    - Each program seems to have exclusive use of main memory.
    - Provided by kernel mechanism called *virtual memory*
Creating and Terminating Processes

- From a programmer’s perspective, we can think of a process as being in one of three states
  - Running
    - Process is either executing, or waiting to be executed and will eventually be scheduled (i.e., chosen to execute) by the kernel
  - Stopped
    - Process execution is suspended and will not be scheduled until further notice
  - Terminated
    - Process is stopped permanently
Process Management

- Creating processes
  - Call fork
  - One call, *two* returns

- Process completion
  - Call exit
  - One call, no return

- Reaping and waiting for processes
  - Call wait or waitpid

- Loading and running programs
  - Call execve
  - One call, no return
int main() {
    pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0) { /* Child */
        printf("child: x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}

child: x=2
parent: x=0
Virtual Addressing

- Creates illusion to process that it has total address space.
- Used in all modern, non-trivial systems
Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use DRAM as a cache for parts of a virtual address space

- Simplifies memory management
  - Each process gets the same uniform linear address space

- Isolates address spaces
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code
- Conceptually, virtual memory is an array of $N$ contiguous bytes stored on disk.

- The contents of the array on disk are cached in physical memory
  - These cache blocks are called pages
Dynamic Allocation
Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*

- Types of allocators
  - *Explicit* allocator: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - *Implicit* allocator: application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp
Constraints

- **Applications**
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - `free` request must be to a `malloc`’d block

- **Allocators**
  - Can’t control number or size of allocated blocks
  - Can’t reorder or buffer requests for memory
  - Must allocate blocks from `free` memory
  - Can manipulate and modify only `free` memory
  - Can’t move the allocated blocks once they are `malloc`’d
Performance Goal: Throughput

- **Goal: Maximize throughput:**
  - Number of completed requests per unit time
  - Example:
    - 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    - Throughput is 1,000 operations/second
Performance Goal: Peak Memory Utilization

- **Aggregate payload / Current heap size = Utilization**
  - Utilization is the sum of currently allocated payloads divided by the current total heap size.

- **Goal: Increase utilization (by reducing fragmentation)**
  - internal fragmentation
  - external fragmentation
Free Lists: Keeping Track of Free Blocks

- **Method 1:** Implicit list using length—links all blocks

- **Method 2:** Explicit list among the free blocks using pointers

- **Method 3:** Segregated free list
  - Different free lists for different size classes
Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph, each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called root nodes (e.g. registers, locations on the stack, global variables)

A node (block) is **reachable** if there is a path from any root to that node. Non-reachable nodes are **garbage** (cannot be needed by the application).
Network Programming
A Client-Server Transaction

- Many network applications are based on the client-server model:
  - A server *process* and one or more client processes
  - Server manages some *resource*
  - Server provides service by manipulating resource for clients
  - Server activated by *request* from client

Note: clients and servers are processes running on hosts (can be the same or different hosts)
Computer Networks

- A network is a group of connected systems that are able to communicate in order to exchange data.

- There are many kinds of networks, some examples:
  - LAN (Local Area Network) spans a building or campus
    - Ethernet is a prominent example
  - WAN (Wide Area Network) spans country or world
    - Typically high-speed point-to-point telecom lines

- Network devices that originate, route and terminate data are called 'nodes'.
  - Nodes or 'hosts' can be personal computers, phones, servers as well as special networking hardware.
Two devices can be said to be 'networked' when one device is able to exchange information with the other.

An ‘internetwork’ (internet) is a connected set of networks

- The Global IP Internet (uppercase “I”) is the most famous example of an internet (lowercase “i”)

A 'router' is an networking device that forwards data between networks in an internet.

A 'link' is the means of connecting one location to another for the purpose of transmitting and receiving data across networks.

- phone lines, fiberoptic cables, smoke signals, etc…
Sockets

- What is a socket?
  - To the kernel, a socket is an endpoint of communication
  - To an application, a socket is a file descriptor that lets the application read/write from/to the network
    - All Unix I/O devices, including networks, are modeled as files
  - Clients and servers communicate with each other by reading from and writing to socket descriptors

- The main distinction between regular file I/O and socket I/O is how the application “opens” the socket descriptors
Concurrent Programming
Concurrency

- Multiple logical control flows.

- Flows run concurrently if they overlap in time
  - Otherwise, they are sequential

- Examples (running on single core):
  - Concurrent: A & B, A & C
  - Sequential: B & C
Process Concurrency

- Use fork to launch multiple processes to do work.
  - Hard for them to share data. Must use IPC.

- How to communicate across processes? (*inter-process communication or IPC*)
  - via *sockets*
  - via *pipes*
  - via *shared memory objects*
Thread-based Concurrency

- **A thread is...**
  - a unit of execution, associated with a process.
  - the smallest sequence of instructions that can be managed independently by the OS scheduler.

- **Multiple threads can...**
  - exist within one process
  - be executing concurrently
  - *share resources* such as memory
Threads vs. Processes

- **Similarities**
  - Each has its own logical control flow
  - Each can run concurrently (possibly on different cores)
  - Each is context switched

- **Differences**
  - Threads share all code and data (except local stacks)
    - Processes do not
  - Threads are somewhat less expensive than processes
    - Process control (creating/reaping) $2x$ as expensive as thread control
Concurrent Programming is Hard!

- The ease with which threads share data and resources also makes them vulnerable to subtle and baffling errors.

- Classical problem classes of concurrent programs:
  - *Races*: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - *Deadlock*: improper resource allocation prevents forward progress
  - *Livellock / Starvation / Fairness*: external events and/or system scheduling decisions can prevent sub-task progress
Race Example

- What's the expected output on line 11?
  - 2
- Possible output...
  - 1

```c
int numbers[2] = { 1, 1 };
int sum = 0;

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, run, numbers[1]);
    for (int i = 0; i < 1; i++) {
        sum += numbers[i];
    }
    pthread_join(tid, NULL);
    printf("sum is %d\n", sum);
}

void* run(void* arg) {
    int* numbers = (int*) arg;
    for (int i = 0; i < 1; i++) {
        sum += numbers[i];
    }
    return NULL;
}
```
Synchronizing Threads

- Shared variables are sometimes useful but they introduce the possibility of synchronization errors.
  - Like the one we saw last time

- How do we prevent such things?
  - We need to make sure that only one thread is mutating shared variables at a time.
  - This is known as mutual exclusion

- Moreover, we must protect critical sections.
Mutual Exclusion

- A *mutex*…
  - is a synchronization variable that is used to protect the access to shared variables.
  - surrounds critical sections so that one thread is allowed inside at a time.

- However we have to be careful with these because they can lead to code that performs poorly, or in some cases not at all…
Deadlock

- Both processes need resources to continue execution.
- P1 requires additional resource R1 and is in possession of resource R2
- P2 requires additional resource R2 and is in possession of R1; neither process can continue.
Thread Safety

- Functions called from a thread must be *thread-safe*

- Def: A function is thread-safe iff it will always produce correct results when called repeatedly from multiple concurrent threads

- Classes of thread-unsafe functions:
  - Class 1: Functions that do not protect shared variables
  - Class 2: Functions that keep state across multiple invocations
  - Class 3: Functions that return a pointer to a static variable
  - Class 4: Functions that call thread-unsafe functions 😊