Final Exam
Logistics

- **Exam:** 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.
  - This is a survey of topics and some of the core concepts.
  - There are about 55 slides.. so we will be doing on average a slide a minute!!
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 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

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

- **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.

\[ (Rb, Ri) \quad \text{Mem}[ \ \text{Reg}[Rb] + \text{Reg}[Ri] \ ] \]
\[ D(Rb, Ri) \quad \text{Mem}[ \ D + \text{Reg}[Rb] + \text{Reg}[Ri] \ ] \]
\[ (Rb, Ri, S) \quad \text{Mem}[ \ \text{Reg}[Rb] + \text{Reg}[Ri] \times S \]
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:**

  - **Format** | **Computation**
    |---|---|
    | `addq src, dest` | `dest = dest + src` |
    | `subq src, dest` | `dest = dest - src` |
    | `imulq src, dest` | `dest = dest * src` |
    | `salq src, dest` | `dest = dest << src` (also called `shlq`) |
    | `sarq src, dest` | `dest = dest >> src` (arithmetic) |
    | `shrq src, dest` | `dest = dest >> src` (logical) |
    | `xorq src, dest` | `dest = dest ^ src` |
    | `andq src, dest` | `dest = dest & src` |
    | `orq src, dest` | `dest = dest | src` |

- **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:
  - **lea**: “address” computation
  - **sal**: 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>%rax</th>
<th>%r8</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

Instruction

| CF  | ZF  | SF  | OF  |

Condition

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: \( \text{addq } src, dest \leftrightarrow b = a + b \)
    - **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
      \[ (a > 0 \land b > 0 \land t < 0) \lor (a < 0 \land b < 0 \land t > 0) \]

- Not set by \( \text{leaq } \) 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>
Conditional Move Example

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

### 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>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rdx</td>
<td>Temp variable</td>
</tr>
</tbody>
</table>

### Assembly Code

```
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 \( y \leq x \), 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>

```assembly
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.
**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
Memory
Smaller, faster, and costlier (per byte) storage devices

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

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

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

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

L4: Main memory (DRAM) holds disk blocks retrieved from local disks.

L5: Local secondary storage (local disks) holds files retrieved from disks on remote servers.

L6: Remote secondary storage (e.g., Web servers)
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
- Simplified system structure:
E-way Set Associative Cache (E = 2)

E = 2: Two lines per set
Assume: cache block size 8 bytes

Address of short int:

| t bits | 0...01 | 100 |

---

find set
Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers:
    - 3-10% for L1
    - < 1% for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory
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.
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 Accessing Example

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

- 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[] nyu;
```

```
1 0 0 0 3
16 20 24 28 32 36
```

```c
# %rdi = z
# %rsi = digit
movq (%rdi,%rsi,4), %rax  # z[digit]
```
- 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;
};
```
Alignment Requirement

- For largest alignment requirement K
- Overall structure must be multiple of K

```c
struct S2 {
    double v;
    int i[2];
    char c;
} *p;
```
+ Saving Space (Struct packing)

- Put large data types first

```
struct S4 {
    char c;
    int i;
    char d;
} *p;
```

```
struct S5 {
    int i;
    char c;
    char d;
} *p;
```

- Effect (K=12 -> K=4)

```
| c | 3 bytes | i | d | 3 bytes |
```

```
| i | c | d | 2 bytes |
```
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)

Control flow

\[
\begin{align*}
\text{<startup>} \\
\text{inst}_1 \\
\text{inst}_2 \\
\text{inst}_3 \\
\vdots \\
\text{inst}_n \\
\text{<shutdown>}
\end{align*}
\]
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
- 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
  - Zombie!
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
```c
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);
}
```

**Parent**

- `x==1`
- `main` → `fork` → `printf` → `exit`

**Child**

- `child: x=2`
- `printf` → `exit`

The process graph example illustrates the flow of control and data from `main` to `fork` to `printf` and finally `exit` in both the parent and child processes.
Virtual Memory
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
VM as Caching

- 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*.
VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
Dynamic Allocation
Programmers use dynamic memory allocators to acquire virtual memory at run time.
- For data structures whose size is only known at runtime.

Dynamic memory allocators manage an area of process virtual memory known as the heap.
+ 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 Goals

- **Goal: Increase utilization (by reducing fragmentation)**
  - internal fragmentation
  - external fragmentation

- **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
Implicit Lists

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!

- Standard trick
  - If blocks are word-aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - *(Note: when reading size, must mask out any extra bits)*

`\[
\begin{array}{c|c}
\text{Size} & a \\
\hline
\text{Payload} & a = 1: \text{Allocated block} \\
 & a = 0: \text{Free block} \\
\hline
\text{Optional padding} & \text{Payload: application data (allocated blocks only)}
\end{array}
\]`
Detailed Implicit Free List Example

Start of heap

Unused

8-byte word aligned

Allocated blocks: shaded

Free blocks: unshaded

Headers: labeled with size in bytes/allocated bit
Ideas to remember...

- Allocation
- Freeing
- Fragmentation
  - External
  - Internal
- Coalescing
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:
  - \textit{Races}: outcome depends on arbitrary scheduling decisions elsewhere in the system
Race Example

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

- Possible output...
  - 1
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 executing “critical sections” (usually mutating shared variables!)
  - This is known as mutual exclusion

- Moreover, we must protect critical sections.

- AGAIN, MULTIPLE THREADS WRITING TO THE EXACT SAME LOCATION IN MEMORY CONCURRENTLY IS A BAD THING!
Mutual Exclusion

- A *mutex*...
  - is synchronization variable that is used to protect the access to shared variables.
  - surrounds critical sections so that one threads 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...