Final Review

Jinyang Li
Final logistics

• 100 minutes (half an hour longer than quiz1&2)
  – Cover all materials
  – More emphasis on the second half of class (70%)
• Closed book
  – except for one double-sided cheat sheet
  – No electronic devices
• Read all questions, do the easier ones first.

Disclaimer: this review is not complete.
Not all exam materials are covered by this review!
What we’ve learnt (first half)

1. C language
   – pointers, bitwise operations
   – Compilation, linking

2. Basic program execution
   – Digital representation of numbers and characters
   – CPU state vs. memory, basic x86 instructions
   – Buffer overflow
What we’ve learnt (second half)

3. Dynamic Memory Allocation

4. Advanced program execution
   – virtual memory
   – caching
   – Multi-processing

5. Multi-threading
Topic #1
C programming
Global vs. Local vs. Heap Variable

- Know the whereabouts of variables, when they are allocated/deallocated
- Variables are not automatically initialized upon declaration

```c
void add(int x) {
    x++;
}

void main() {
    int x = 0;
    add(x);
    printf("x is %d\n", x);
}
```

What’s the output?

Answer: 0
Global vs. Local vs. Heap Variable

• Know the whereabouts of variables, when they are allocated/deallocated
• Variables are not automatically initialized upon declaration

```c
int add(int x) {
    x++;
    return x;
}

void main() {
    int x = 0;
    x = add(x);
    printf("x is %d\n", x);
}
```

What’s the output?

Answer: 1
Global vs. Local vs. Heap Variable

• Know the whereabouts of variables, when they are allocated/deallocated
• Variables are not automatically initialized upon declaration

```c
int add(int x) {
    x++;
    return x;
}

void main() {
    int x = 0;
    x = add(x);
    printf("x is %d\n", x);
}
```

What’s the output?

Answer: Could be any number
Pointers

• Pointers are addresses to variables
• You must be aware of whether the variable being pointed to has been allocated or not and where

```c
void add(int *x) {
    (*x) = (*x) + 1;
}

void main() {
    int y = 0;
    int *x = &y;
    add(x);
    printf("x is \%d\n", *x);
}
```

What’s the output?
Answer: 1
Pointers

• Pointers are addresses to variables
• You must be aware of whether the variable being pointed to has been allocated or not, and where

```c
void add(int *x) {
    (*x) = (*x) + 1;
}

void main() {
    int y = 0;
    int *x = &y;
    add(x);
    printf("x is %d\n", *x);
}
```

What’s the output?

Answer: Likely segmentation fault
#1.2 C Programming

• Pointers are addresses to variables
• You must be aware of whether the variable being pointed to has been allocated or not, and where

```c
int *
sum(int x, int y) {
    int z = x + y;
    return &z;
}

void main() {
    int *r1 = sum(1,1);
    int *r2 = sum(*r1, 1);
    printf("%d\n", *r2);
}
```

What’s the output?

Answer: likely garbage
Pointers and arrays

- Arrays store a set of identically typed elements contiguously

```c
void main() {
    int nums[5] = {1, 2, 3, 4, 5};
    int *p;
    p = nums + 2; // equivalent to p = &nums[2];
    p++;
    printf("%d\n", *p);
}
```

What’s the output?

Answer: 4
void main() {
    int x = 1<<31;    // equivalent to x = 0x80000000;
    char *y;
    y = (char *)&x;
    for (int i = 0; i < 4; i++) {
        printf("%d ", y[i]);
    }
}

What’s the output?

Answer: 0 0 0 -128
(little Endian)
C: Other Concepts

• ASCII characters
• C string
  – Null-terminated ASCII character array
• Use malloc appropriately
  – allocate the right size
  – free allocated memory to avoid memory leak
Topic #2
Basic Program Execution
Basic Program Execution

CPU
- ALU (arithmetic logic unit)
- GP Registers (%rax, %rbx, ...)
- PC (%rip)
- Condition Codes
- Floating point registers

Floating point unit

Memory
- Code
- Data
- Stack

Addresses
Data
Instructions

Diagram showing the flow of operations between CPU and Memory, including ALU, GP Registers, PC, Condition Codes, and Floating point registers.
Machine instructions: mov

1. mov instructions

<table>
<thead>
<tr>
<th>Source</th>
<th>Dest</th>
<th>Source, Dest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Imm</td>
<td>Reg</td>
<td>movq $0x4,%rax</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq $0x4,(%rax)</td>
</tr>
<tr>
<td>Reg</td>
<td>Reg</td>
<td>movq %rax,%rdx</td>
</tr>
<tr>
<td></td>
<td>Mem</td>
<td>movq %rax, 16(%rbx, %rdx, 8)</td>
</tr>
<tr>
<td>Mem</td>
<td>Reg</td>
<td>movq (%rax),%rdx</td>
</tr>
</tbody>
</table>

General memory addressing mode:
D(Rb, Ri, S): val(Rb) + S*val(Ri) +D

S: 1, 2, 4, or 8
Machine instructions

• Arithmetic operations
  – add %eax, %ebx
  – sub, mul

• The lea instruction
  – lea 0x1(%rax, %rbx, 2), %rdx

• Bitwise-operations:
  – shl/shr, sal/saq, and, or, xor
Control flow

• Normal control flow is linear
  – load instruction stored at address %rip
  – execute it
  – %rip = %rip + (length of instruction)

• Non-linear Control flow
  – combination of two types of instructions
    • instructions that set conditional codes, CF, ZF, SF, OF
    • jmp instructions that may or may not jump depending on condition codes
  – condition codes can be set
    • implicitly: add, sub ..
    • explicitly: cmp, test, ...
Procedure execution

- call, ret
- push, pop
  - %rsp
- C calling convention
  - first 6 arguments in %rdi, %rsi, %rdx, %rcx, %r8, %r9
  - return value: %rax
  - caller vs. callee save registers

Stack pointer %rsp

Caller Frame
- Arguments
  - 7+
- Return Addr
- Saved Registers +
- Local Variables
- Argument Build (Optional)
Buffer Overflow

Before call to gets

Stack Frame for call_echo

Return Address (8 bytes)

20 bytes unused

void echo()
{
    char buf[4];
    gets(buf);
    puts(buf);
}

buf ← %rsp

echo:
    subq $24, %rsp
    movq %rsp, %rdi
    call gets
    ...
Topic #3
Dynamic Memory Allocation
Dynamic memory allocation

• How to implement malloc/free?
• Goal: high throughput and high utilization
• Design questions:
  – how to keep track of free blocks
  – which free blocks to allocate?
  – free is only given a pointer, how to know its block size?
Dynamic memory allocation

• implicit list
  – one (implicit) list containing all free and non-free blocks

• explicit free list
  – one explicit linked list containing all free blocks

• segregated free list
  – multiple explicitly linked lists for free blocks,
  – each links corresponds to a different size class
Question: given pointer `p` of type `void *` pointing to the payload, how to get a pointer to the current, next, previous block?

```c
curr = (header *)((char *)p - sizeof(header));
next = (header *)((char *)p + sizeof(header) + get_chunksize(curr));
footer = (header *)((char *)p - 2*sizeof(header));
prev = (header *)((char *)p - 3*sizeof(header) - get_chunksize(footer));
```
Topic #4
Advanced topics on program execution

VM, Caching, Multiprocessing
Virtual memory

- User program access virtual address
- 32-bit address $\rightarrow$ address range $[0x00000000, 0xffffffff]$
VM: one-level page table

• Example:
  – 8-bit virtual and physical addresses
  – 16-byte page size

How many pages in the 8-bit address space?
Answer: \(2^8/16 = 2^4 = 16\) pages

Page table is an array of PTEs.

How many PTEs needed to address all pages in address space?
Answer: 16 PTEs
VM: one-level page table

- Example:
  - 8-bit virtual and physical addresses
  - 16-byte page size

VA: 0x11010010
What's the PA?
Answer: page fault

VA: 0x11100010
What's the PA?
Answer: 0xB2
VM: Multi-level page table

- Example:
  - 32-bit virtual and physical addresses
  - 4KB page size

How many pages in the 32-bit address space?
Answer: $2^{32}/4\text{KB} = 2^{20}$ pages

How many PTEs fit in one page?
Answer: $4\text{KB}/4 = 2^{10}$ PTEs
VM: Multi-level page table

• Example:
  – 32-bit virtual and physical addresses
  – 4KB page size

Answer: \( \frac{4KB}{4} = 2^{10} \) PTEs
VM: Multi-level page table

- Example:
  - 32-bit virtual and physical addresses
  - 4KB page size

```
0x12345001 ...
0x85449001
0x87354440
0x12345001 ...
0x85449001
0x87354440
```

```
VA: 0x00401678
What is PA?
Answer: 674A0678
```

```
VA: 0x00001678
What is PA?
Answer: page fault
```
Address space

- Each running program has its own page table and address space

![Diagram showing address space](image-url)
OS and user-level processes

• OS: a layer of software between app and h/w
  – hide h/w details
  – manage resource sharing among apps

• H/w primitive: privileged vs. unprivileged execution
  – exception (e.g. page fault)
  – traps (used for syscall)
  – interrupt (e.g. timer interrupt)
invoking kernel functions: syscalls

• h/w instruction, syscall
  – open, close, read, write
  – futex, fork, clone
  – man 2 fork
OS abstraction: Multi-processing

• Process: an instance of a running program
• Managed by OS, each process has
  – its own virtual address space
  – saved execution context
  – process id
  – ....
fork and exec

```c
void main() {
    printf("hello\n");
    if (fork() == 0) {
        printf("big\n");
        exec("/bin/echo", "world");
        printf("lovely\n");
    }
}
printf("Bye\n");
}
```

• What are potential interleavings?

- hello
- hello
- hello
- big
- big
- big
- bye
- bye
- world
- world
- big
- world
- world
- bye
- world
- world
forked processes have separate address space

```c
int global = 1;
void main() {
    pid_t pid = fork();
    if (pid == 0) {
        global = 2;
        printf("child global=%d\n", global);
    } else {
        waitpid(pid,...)
        printf("parent global=%d\n", global);
    }
}
```

• What are possible outputs?
  child global=2
  parent global=1
Topic #5
Multi-threaded programming
Concurrent programming

• A single process can have multiple threads
  – each thread has its own control flow & stack
  – all threads share the same address space
• Multi-threaded programs need synchronization
• Synchronization problems:
  – races
  – deadlock
Races

• Examples:
  – modifying shared counters
  – modifying shared linked list, hash table etc.

• Caused by arbitrary interleaving of execution among different threads

<table>
<thead>
<tr>
<th>Thread-1 (x++)</th>
<th>Thread-2 (x++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>read x (from memory) into %eax</td>
<td>read x into %eax</td>
</tr>
<tr>
<td>add $1, %eax</td>
<td>add $1, %eax</td>
</tr>
<tr>
<td>write %eax to x (in memory)</td>
<td>write %eax to x</td>
</tr>
</tbody>
</table>
Races

```c
node *head;
list_insert(int x) {
    node *n = malloc ...  
    n->val = x;  
    n->next = head;  
    head = n;
}
```

what can go wrong if two threads insert at the same time?

<table>
<thead>
<tr>
<th>Thread-1: list_insert(1)</th>
<th>Thread-2: list_insert(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>L1</td>
</tr>
<tr>
<td>L2</td>
<td>L2</td>
</tr>
<tr>
<td>L3</td>
<td>L3</td>
</tr>
<tr>
<td>L4</td>
<td>L4</td>
</tr>
</tbody>
</table>
Mutexes

• Protect “critical section”

Big lock implementation

```c
int acc[100];
pthread_mutex_t mu;
void transfer(int x, int y){
    pthread_mutex_lock(&mu);
    acc[x] -=10;
    acc[y] += 10;
    pthread_mutex_unlock(&mu);
}
```

Fine-grained lock

```c
typedef struct {
    int balance;
    pthread_mutex_t mu;
} account_t;

account_t acc[100];

void transfer(int x, int y) {
    pthread_mutex_lock(&acc[x].mu);
    pthread_mutex_lock(&acc[y].mu);
    acc[x].bal -= 10;
    acc[y].bal += 10;
    pthread_mutex_unlock(&acc[x].mu);
    pthread_mutex_unlock(&acc[y].mu);
}
```
Conditional variables

- lets a thread wait for some condition to become true
- Remember the pattern for using conditional variables

Thread-1
mutex_lock(&m)
while (condition != true)
    cond_wait(&c, &m);
//condition is true
modify shared state
mutex_unlock(&m)

Thread-2
mutex_lock(&m);
condition = true;
cond_signal(&c);
//or cond_broadcast(&c)
mutex_unlock(&m);
H/W Atomic instructions

• Basic spinlock implementation

```c
spin_lock(int *m)
{
    while (xchg(m, 1)! = 0);
}

spin_unlock(int *m)
{
    xchg(m, 0);
}
```