Course ~~Over~~Review

15-213: Introduction to Computer Systems
28th Lecture, August 5, 2022

Course Theme:
Abstraction Is Good But Don't Forget Reality

- **Most CS and CE courses emphasize abstraction**
  - Abstract data types
  - Asymptotic analysis
- **These abstractions have limits**
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations
- **Useful outcomes from taking 213**
  - Become more effective programmers
    - Able to find and eliminate bugs efficiently
    - Able to understand and tune for program performance
  - Prepare for later "systems" classes in CS & ECE
    - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems, Storage Systems, etc.

Computer Arithmetic

- **Does not generate random values**
  - Arithmetic operations have important mathematical properties
- **Cannot assume all "usual" mathematical properties**
  - Due to finiteness of representations
  - Integer operations satisfy "ring" properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy "ordering" properties
    - Monotonicity, values of signs
- **Observation**
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

You've Got to Know Assembly

- **Key to machine-level execution model**
  - Behavior of programs in presence of bugs
    - High-level language models break down
  - Tuning program performance
    - Understand optimizations done / not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing system software
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting malware
    - x86 assembly is the language of choice!

Memory Isn't Random Access

- **Memory is not unbounded**
  - It must be allocated and managed
  - Many applications are memory dominated
- **Memory performance is not uniform**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements
- **Memory referencing bugs especially pernicious**
  - Effects are distant in both time and space

Memory Referencing Errors

- **C and C++ do not provide any memory protection**
  - Out of bounds array references
  - Invalid pointer values
  - Abuses of malloc/free
- **Can lead to nasty bugs**
  - Whether or not bug has any effect depends on system and compiler
  - Action at a distance
    - Corrupted object logically unrelated to one being accessed
    - Effect of bug may be first observed long after it is generated
- **How can I deal with this?**
  - Program in Java, Ruby, Python, ML, …
  - Understand what possible interactions may occur
  - Use or develop tools to detect referencing errors (e.g. Valgrind)

Constant Factors Matter

- **Even exact op count does not predict performance**
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- **Must understand system to optimize performance**
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

Computers Don't Just Compute

- **They need to get data in and out**
  - I/O system critical to program reliability and performance

- **They communicate with each other over networks**
  - Many system-level issues arise in presence of network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross platform compatibility
    - Complex performance issues

Final Exam

- **August *11th* (NOT the 12th)**
  - 12:20—3:20pm, location TBD (will announce on Piazza)

- **The focus is on the second half of the course**
  - IO
  - Signals
  - Processes
  - Virtual Memory
  - Malloc
  - Threads
  - Thread Synchronization
  - Other

IO

In the following code, a parent opens a file twice, then the child reads a character:

```
char c;
int fd1 = open("foo.txt", O_RDONLY);
int fd2 = open("foo.txt", O_RDONLY);
if (!fork()) { read(fd1, &c, 1); }
```

Clearly, in the child, fd1 now points to the second character of foo.txt. Which of the following is now true in the parent?

- (a) **fd1 and fd2 both point to the first character.**
- (b) **fd1 and fd2 both point to the second character.**
- (c) **fd1 points to the first character while fd2 points to the second character.**
- (d) **fd2 points to the first character while fd1 points to the second character**

Signals

```
void sigint_handler(int sig)
{
    jid_t fg_jid = fg_job();

    /* Masking signals */
    sigset_t mask, prev_mask;
    Sigfillset(&mask);
    Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    if (fg_jid != 0) {
        /* Sending a SIGINT signal for the process group.
         * Deleting the job. */
        pid_t pid = job_get_pid(fg_jid);
        kill(-pid, SIGINT);
        delete_job(pid);
    }

    /* Unblocking the masked signals */
    Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
}
```

Name three bugs in this code

Signals

```
void sigint_handler(int sig)
{
    jid_t fg_jid = fg_job();

    /* Masking signals */
    sigset_t mask, prev_mask;
    Sigfillset(&mask);
    Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

    if (fg_jid != 0) {
        /* Sending a SIGINT signal for the process group.
         * Deleting the job. */
        pid_t pid = job_get_pid(fg_jid);
        kill(-pid, SIGINT);
        delete_job(pid);
    }

    /* Unblocking the masked signals */
    Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
}
```

errno not saved

Job list API used with signals unblocked – should be here instead

delete_job takes a JID,not a PID

delete_job should be called from the SIGCHLD handler, not here

errno not restored

Name three bugs in this code

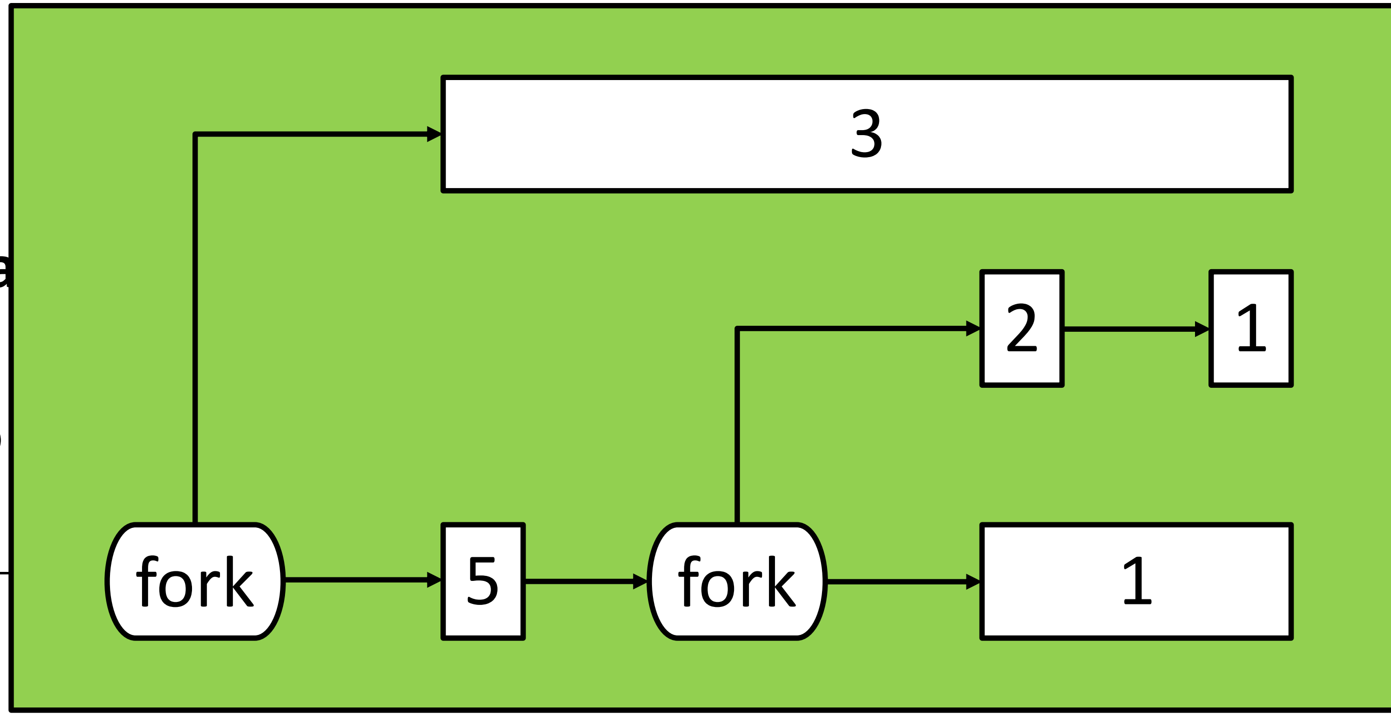
Processes

What outputs are possible? Is “15213”?

```
pid_t Fork(void) {
    pid_t pid = fork();
    if (pid == -1) exit(1);
    return pid;
}
int main(void) {
    setvbuf(stdout, 0, _IONBF, 0); // no buffering
    if (Fork() == 0) { putchar('3'); return 0; }
    putchar('5');
    if (Fork() == 0) { putchar('2'); }
    putchar('1');
    return 0;
}
```

Processes

What outputs a




```
pid_t Fork(void)
    pid_t pid =
    if (pid == -
    return pid;
}
int main(void) {
    setvbuf(stdout, 0, _IONBF, 0); // no buffering
    if (Fork() == 0) { putchar('3'); return 0; }
    putchar('5');
    if (Fork() == 0) { putchar('2'); }
    putchar('1');
    return 0;
}
```

Malloc

- **First-fit allocator, with 16-byte alignment, 8-byte headers / footers, and prologue / epilogue. After:**

  malloc(3)

  malloc(11)

  malloc(40)

  free(40)

  malloc(10)

- **Draw the state of the heap in 8 byte units, label as header / footer (size, alloc or free), payload:**

  F HPpF HPPF HPPF hppf *H*

- **What is the utilization for this allocator, versus 54 bytes?**

  At peak usage, ${}^{54}/_{8 \cdot 18} = {}^{54}/_{144} = 37.5\%$ (ouch!)

- **How much space would be saved by removing footers?**

  16 bytes (*F* HPpp HPPp HPPppp *H*) – alignment padding eats most of the benefit

Threads

- **What is the range of value(s) that main will print?**

  There's no synchronization, so 1 and 2 are both possible.

- **If we remove `i` from `thread` and instead directly access `count`, does the answer change?**

  No, this makes no difference.

```
int count = 0;
void *thread(void *unused) {
    int i = count;
    i = i + 1;
    count = i;
}
int main(void) {
    pthread_t tid[2];
    for (int i = 0; i < 2; i++)
        pthread_create(&tid[i], NULL,
                       thread, NULL);
    for (int i = 0; i < 2; i++)
        pthread_join(tid[i]);
    printf ("%d\n", count);
    return 0;
}
```

Virtual Memory

- Virtual addresses are 20 bits wide
- Physical addresses are 18 bits wide
- Page size is 1024 bytes
- TLB is 2-way set associative with 16 total entries

- Label each bit of a virtual address (Virtual Page offset, Virtual page number, TLB index, TLB tag):

  NNNN NNNN NNoo oooo oooo

  TTTT TTTi ii

- Given virtual address 0x04AA4, what happens?

  VPN is 0x04A >> 2 = 0x12; TLB index is 2, tag is 02

  PPN is 0x68

  Phys addr is 0x68 << 10 | (0x04AA4 & 0x3FF) = 0x1A2A4

TLB			
Index	Tag	PPN	Valid
0	03	C3	1
	01	71	0
1	00	28	1
	01	35	1
2	02	68	1
	3A	F1	0
3	03	12	1
	02	30	1