Recitation 15

15-213: Introduction to Computer Systems
Recitation 15: December 3, 2012

Daniel Sedra
Section C
Topics

- News
- Proxylab
- Exam review :>)
- Wrap up
Final and Review Session News

- Final is on Monday, December 10 from 8:30-11:30 am in UC McConomy, PH 100, and PH 125C

- Final review session is on Sunday, December 9 in GHC 4401 from 2-5 pm

- Email and bring questions!
Proxylab News

- Proxylab due on Sunday, Dec 2
- Last date to handin is Wednesday, Dec 5
- Each group gets 2 grace days
Things To Know: First Half

- Integer and floating point numbers
- X86 and X86-64 assembly
  - Procedure calls and stack discipline
  - Memory layout
- Memory hierarchy and caching

Reviewed today
Things To Know: Second Half

- Signals and processes
- Concurrent programming and threading
- Virtual memory
- Dynamic Memory
- Networks
Virtual Memory

Figure 3-12. Linear Address Translation (4-KByte Pages)
Virtual Memory

Figure 3-14. Format of Page-Directory and Page-Table Entries for 4-KByte Pages and 32-Bit Physical Addresses
Virtual Memory

- The setup is a 2-level page table for a 32-bit Intel system with 4 KB page tables. Reference diagrams above.

- The relevant contents of memory are shown; anything not shown is presumed zero. The Page Directory Base Address is 0x0c23b000.

- For the following problems, perform the virtual to physical address translation. If an error occurs at any point that would prevent the system from performing lookup, circle FAILURE and not the address that caused it.
Virtual Memory

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00023000</td>
<td>beefbee0</td>
</tr>
<tr>
<td>00023120</td>
<td>12fdec83</td>
</tr>
<tr>
<td>00023200</td>
<td>debcfd23</td>
</tr>
<tr>
<td>00023320</td>
<td>d2e52933</td>
</tr>
<tr>
<td>00023FFF</td>
<td>bcdeff29</td>
</tr>
<tr>
<td>00055004</td>
<td>8974d003</td>
</tr>
<tr>
<td>0005545c</td>
<td>457bc293</td>
</tr>
<tr>
<td>00055460</td>
<td>457bd293</td>
</tr>
<tr>
<td>00055464</td>
<td>457be293</td>
</tr>
<tr>
<td>0c23b020</td>
<td>01288b53</td>
</tr>
<tr>
<td>0c23b040</td>
<td>012aab53</td>
</tr>
<tr>
<td>0c23b080</td>
<td>00055d01</td>
</tr>
<tr>
<td>0c23b09d</td>
<td>0FF2d303</td>
</tr>
<tr>
<td>0c23b274</td>
<td>00023d03</td>
</tr>
<tr>
<td>0c23b9fc</td>
<td>2314d222</td>
</tr>
<tr>
<td>2314d200</td>
<td>0fdc1223</td>
</tr>
<tr>
<td>2314d220</td>
<td>d21345a9</td>
</tr>
<tr>
<td>2314d4a0</td>
<td>d388bcbd</td>
</tr>
<tr>
<td>2314d890</td>
<td>00b32d00</td>
</tr>
<tr>
<td>24AEE520</td>
<td>b58cda1d</td>
</tr>
<tr>
<td>29DE2504</td>
<td>56ffad02</td>
</tr>
<tr>
<td>29DE4400</td>
<td>2ab45cd0</td>
</tr>
<tr>
<td>29DE9402</td>
<td>d4732000</td>
</tr>
<tr>
<td>29DEE500</td>
<td>1a23c0b0</td>
</tr>
</tbody>
</table>
Virtual Memory

1. Read from virtual address 0x080016ba
   
   a) Physical address of PDE:
   
   b) Physical address of PTE:
   
   c) Success, the physical address is:
   
   or
   
   Failure, the entry causing it was:
Virtual Memory

1. Read from virtual address 0x080016ba
   
a) Physical address of PDE: 0xc23b080

b) Physical address of PTE: 0x55004

c) Success, the physical address is: 0x8974d6ba

or

Failure, the entry causing it was:
Virtual Memory

1. Read from virtual address 0x9fd28c10
   
a) Physical address of PDE:
   
b) Physical address of PTE:
   
c) Success, the physical address is:
   
or
   Failure, the entry causing it was:
1. Read from virtual address 0x9fd28c10

   a) Physical address of PDE: 0xc23b9fc

   b) Physical address of PTE:

   c) Success, the physical address is:

or

   Failure, the entry causing it was: 0xc23b9fc
I/O

- Consider the following code assuming all system calls to read() and write() are atomic of each other.

- foo.txt = “ABCDEFG”

Determine

1. The output of the program.

2. The final contents of foo.txt.
I/O

```c
void read_and_print_one(int fd) {
    char c;
    read(fd, &c, 1);
    printf("%c", c); fflush(stdout);
}

int main(int argc, char * argv[])
{
    int fd1 = open("foo.txt", O_RDONLY);
    int fd1 = open("foo.txt", O_RDONLY);
    read_and_print_one(fd1);
    read_and_print_one(fd2);

    if ( !fork() ){
        read_and_print_one(fd2);
        read_and_print_one(fd2);
        close(fd2);
        fd2 = dup(fd1);
        read_and_print_one(fd2);
    }
    else{
        wait(NULL);
        read_and_print_one(fd1);
        read_and_print_one(fd2);
        printf("\n");
    }

    close(fd1);
    close(fd2);
    return 0;
}
```
Consider the following code assuming all system calls to read() and write() are atomic of each other.

foo.txt = “ABCDEFG”

Determine

1. The output of the program.
   “AABCBCD”
1. The final contents of foo.txt.
   “ABCDEFG”
Concurrent Processes

- Assume for the following program that:

1. All processes run to completion and no system calls fail.

2. `printf()` is atomic and calls `fflush(stdout)` after printing.

3. Logical operators such as `&&` evaluate their operands left to right and only evaluate the smallest number of operands to determine the result.
int main()
{
    int counter = 0;
    int pid;

    if( !(pid = fork()) ){
        while( (counter < 2) && (pid = fork()) ){
            counter++;
            printf("%d", counter);
        }
        if( counter > 0 ){
            printf("%d", counter);
        }
    }
    if(pid){
        waitpid(pid, NULL, 0);
        counter = counter << 1;
        printf("%d", counter);
    }
}
Concurrent Processes

- For the above code determine list all possible outputs.
Concurrent Processes

- For the above code determine list all possible outputs.

  112240  121240  122140
Stack Discipline

Consider the following C code (compiled on a 32-bit machine):

```c
void foo(char * str, int a){
    int buf[2];
    a = a; // pacify gcc
    strcpy((char *) buf, str);
}

/* the base pointer for the stack frame caller() is 0xffffd3e8

void caller()
    { foo("123456", 0xdeadbeef);

}*/
```
Stack Discipline

Here is the corresponding machine code on a 32-bit Linux machine:

```
080483c8 <foo>:
080483c8 <foo+0>:
080483c8 <foo+1>:
080483c8 <foo+3>:
080483c8 <foo+6>:
080483c8 <foo+9>:
080483c8 <foo+12>:
080483c8 <foo+16>:
080483c8 <foo+19>:
080483c8 <foo+24>:
080483c8 <foo+25>:
080483c8 <caller>:
080483c8 <caller+0>:
080483c8 <caller+1>:
080483c8 <caller+3>:
080483c8 <caller+6>:
080483c8 <caller+14>:
080483c8 <caller+21>:
080483c8 <caller+26>:
080483c8 <caller+27>:
```
Stack Discipline:

- At what address is the string “0123456” stored (before strcpy)?
- Why doesn’t ret take an address to return to like jmp?
- Fill in the hex values of buf[0]…buf[4].
- Will a function that calls caller() notice any corruption?
Stack Discipline:

- At what address is the string “0123456” stored (before strcpy)? 0x80484d0

- Why doesn’t ret take an address to return to like jmp?
  ret gets return address from top of stack

- Fill in the hex values of buf[0]...buf[4].
  0x33323130, 0x00363534, 0xfffffd3e8, 0x080483fc, 0x080484d0

- Will a function that calls caller() notice any corruption?
  No segfault. "0123456" is only 8 bytes including '\0', and int[2] can store 8 bytes.
Questions?