15-213 Final Exam Review

Monday December 1, 2014

TJ O'Connor

Today: Final Exam Review

- Final Exam Details:
- Monday (8^{th}) – Thursday (11^{th})
- 10 AM – 10 PM
- Similar to midterm in format
- Cumulative

Review Questions

- Virtual Memory – Spring 2011 #12
- Synchronization – Fall 2011 #11
- Signals – Spring 2011 #11
- Processes – Fall 2012 #8

VM – Spring 2011 #12

- **Task:** Perform the virtual to physical address translation.

- **Key Information:**
- 2 - Level Page Table
- Page Directory Base Address is 0x0045d000.
- 32-bit Intel system with 4 KByte page tables.
- 4 sets, 2 lines per set in TLB (by inspection).

VM – Spring 2011 #12 - Reasoning

- 4 byte addresses means Page Directory Entries and Page Table Entries (PDEs and PTEs) are 4 bytes because the entries are pointers to Page Tables.
- Page Table Entries (PTEs) are Physical Page Offsets.
- The VPN will break down into the PDE and PTE.

VM – Spring 2011 #12 - Reasoning

- 32-bit Intel system with 4 KByte page tables.
  - Deduce: (4*1024) / 4 = 2^10 Bytes ==> 10 bits needed to index a page directory or page table.
  - Address = 32 bits. 10 each for Page Directory Index and Page Table Index, so (32 – 20 = 12) bits needed to byte-address each page.
  - Lower order 12 bits of each memory address are the VPO and PPO (Page Offset).

- Virtual Address = (VPN)::(VPO) = ((TLB Tag)::(TLB Index))::(VPO)
- ==> VPN = [(TLBT) :: (TLBI (2 bits))] [PPO/VPO (12 bits)]

  TLBT has the remaining bits (32 - 2 – 12 = 18 bits)

1. Read from virtual address 0x9fd28c10.

Virtual Address = (VPN)::(VPO) = ((TLB Tag)::(TLB Index))::(VPO)

Convert from base 16 to base 2 then regroup bits:

(Base 16)	9	F	D	2	8	C	1	0
(Base 2)	1001	1111	1101	0010	1000	1100	0001	0000

(Seperate:) (10 0111 1111 0100 1010)(00)(1100 0001 0000)

(Base 16) (TLB Tag = 0x27F4A)(TLB Index = 0x00)(VPO = C10)

TLB Lookup (Set = 0, Tag = 0x27F4A)

Valid Bit is 0 ==> Failure!

TLB

Index	Tag	Frame Number	Valid
0	0x03506	0x98f8a	1
	0x27f4a	0x34abe	0
1	0x1f7ee	0x95cbc	0
	0x2a064	0x72954	1
2	0x1f7f0	0x95ede	0
	0x2005d	0xaa402	0
3	0x3fc2e	0x2029e	1
	0x3df82	0xff644	0

Address	Contents
000c3020	345ab236
000c3080	345ab237
000c332f	08e4523f
000c3400	93c2ed98
000c3cbc	34abd237
000c3ff0	93c2ed99
000c4020	8e56e237
000c432f	33345237
000c4400	43457292
000c4cbc	385ed293
000c4ff0	c3726292
0045d000	000c3292
0045d028	000c4297
0045d032	0df2a292
0045d0a0	000c3297
0045d3ff	0df2a236
0045d9fc	0df2a237
0df2a000	deded000
0df2a080	bc3de239
0df2a3fc	000c4296
0df2a4a0	00324236
0df2a4fc	df72c9a6
0df2b080	01f008c3
0df2bff0	000c5112

Access Page Table

TLB failed. What now?

Must go through page table.
First, access the Page Directory at:
Directory Base + Index * sizeof(PDE)

To find indices, regroup into VPNs:
(10 0111 1111)(01 0010 1000)

PDI - Page Directory Index = 0x27F
PTI - Page Table Index = 0x128

Dereference:
(0x45D000) + (0x27F * 4) =
0x45D9FC
Value: 0xDF2A237
Means: Valid, Page Table at 0xDF2A000

Lower Order 12 bits are not relevant data in table base address, except for valid bit.

Dereference Page Table:
0xDF2A000 + (0x128 * 4) =

0xDF2A4A0
Valid Bit is 0 ==> Failure!
Page Fault.

Address	Contents
000c3020	345ab236
000c3080	345ab237
000c332f	08e4523f
000c3400	93c2ed98
000c3cbc	34abd237
000c3ff0	93c2ed99
000c4020	8e56e237
000c432f	33345237
000c4400	43457292
000c4cbc	385ed293
000c4ff0	c3726292
0045d000	000c3292
0045d028	000c4297
0045d032	0df2a292
0045d0a0	000c3297
0045d3ff	0df2a236
0045d9fc	0df2a237
0df2a000	deded000
0df2a080	bc3de239
0df2a3fc	000c4296
0df2a4a0	00324236
0df2a4fc	df72c9a6
0df2b080	01f008c3
0df2bff0	000c5112

2. Read from virtual address 0x0d4182c0.

Virtual Address = (VPN)::(VPO) = ((TLB Tag)::(TLB Index))::(VPO)

Convert from base 16 to base 2 then regroup bits:

(Base 16) D 4 1 8 2 C 0

(Base 2) 1101 0100 0001 1000 0010 1100 0000

(Seperate:) (11 0101 0000 0110)(00)(0010 1100 0000)

(Base 16) (TLB Tag = 0x3506)(TLB Index = 0x00)(VPO = 2C0)

TLB			
Index	Tag	Frame Number	Valid
0	0x03506	0x98f8a	1
	0x27f4a	0x34abe	0
1	0x1f7ee	0x95cbc	0
	0x2a064	0x72954	1
2	0x1f7f0	0x95ede	0
	0x2005d	0xaa402	0
3	0x3fc2e	0x2029e	1
	0x3df82	0xff644	0

TLB Hit!

PPN = 0x98F8A

PPO = 2C0

Physical Address:

0x98F8A2C0

Address	Contents
000c3020	345ab236
000c3080	345ab237
000c332f	08e4523f
000c3400	93c2ed98
000c3cbc	34abd237
000c3ff0	93c2ed99
000c4020	8e56e237
000c432f	33345237
000c4400	43457292
000c4cbc	385ed293
000c4ff0	c3726292
0045d000	000c3292
0045d028	000c4297
0045d032	0df2a292
0045d0a0	000c3297
0045d3ff	0df2a236
0045d9fc	0df2a237
0df2a000	deded000
0df2a080	bc3de239
0df2a3fc	000c4296
0df2a4a0	00324236
0df2a4fc	df72c9a6
0df2b080	01f008c3
0df2bff0	000c5112

3. Read from virtual address 0x0a32fcd0.

Virtual Address = (VPN)::(VPO) = ((TLB Tag)::(TLB Index))::(VPO)

Convert from base 16 to base 2 then regroup bits:

(Base 16) A 3 2 F C D 0

(Base 2) 1010 0011 0010 1111 1100 1101 0000

(Seperate:) (10 1000 1100 1011)(11)(1100 1101 0000)

(Base 16) (TLB Tag = 0x28CB)(TLB Index = 0x11)(VPO = CD0)

TLB Lookup (Set = 0, Tag = 0x28CB

Valid Bit is 0 ==> Failure!

TLB			
Index	Tag	Frame Number	Valid
0	0x03506	0x98f8a	1
	0x27f4a	0x34abe	0
1	0x1f7ee	0x95cbc	0
	0x2a064	0x72954	1
2	0x1f7f0	0x95ede	0
	0x2005d	0xaa402	0
3	0x3fc2e	0x2029e	1
	0x3df82	0xff644	0

Address	Contents
000c3020	345ab236
000c3080	345ab237
000c332f	08e4523f
000c3400	93c2ed98
000c3cbc	34abd237
000c3ff0	93c2ed99
000c4020	8e56e237
000c432f	33345237
000c4400	43457292
000c4cbc	385ed293
000c4ff0	c3726292
0045d000	000c3292
0045d028	000c4297
0045d032	0df2a292
0045d0a0	000c3297
0045d3ff	0df2a236
0045d9fc	0df2a237
0df2a000	deded000
0df2a080	bc3de239
0df2a3fc	000c4296
0df2a4a0	00324236
0df2a4fc	df72c9a6
0df2b080	01f008c3
0df2bff0	000c5112

Access Page Table

Go through page table.
First, access the Page Directory at:
Directory Base + Index * sizeof(PDE)

To find indices, regroup into VPNs:
(10 1000)(11 0010 1111)
PDI - Page Directory Index = 0x28
PTI - Page Table Index = 0x32F

Dereference:
(0x45D000) + (0x28 * 4) =
0x45D0A0
Value: 0xC3297
Means: Valid, Page Table at 0xC3000

Lower Order 12 bits are not relevant data in table base address, except for valid bit.

Dereference Page Table:
0xC3000 + (0x32F * 4) =
0xC3CBC
Value: 0x34abd237
Valid Bit is 1 ==> Success!
Final Address:
0x34ABDCDO
(last 3 bytes are VPO/PPO from earlier)

Address	Contents
000c3020	345ab236
000c3080	345ab237
000c332f	08e4523f
000c3400	93c2ed98
000c3cbc	34abd237
000c3ff0	93c2ed99
000c4020	8e56e237
000c432f	33345237
000c4400	43457292
000c4cbc	385ed293
000c4ff0	c3726292
0045d000	000c3292
0045d028	000c4297
0045d032	0df2a292
0045d0a0	000c3297
0045d3ff	0df2a236
0045d9fc	0df2a237
0df2a000	deded000
0df2a080	bc3de239
0df2a3fc	000c4296
0df2a4a0	00324236
0df2a4fc	df72c9a6
0df2b080	01f008c3
0df2bff0	000c5112

Synchronization – Fall 2011 #11

- Task: use P and V semaphore operations to correctly synchronize access to the queue

- Information:
  - The queue is initially empty and has a capacity of 10 data items.
  - Producer threads call the insert function to insert an item onto the rear of the queue.
  - Consumer threads call the remove function to remove an item from the front of the queue.
  - The system uses three semaphores: mutex, items, and slots

A. What is the initial value of each semaphore?

mutex = ________

items = ________

slots = ________

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */





    add_item(item);
    /* Insert sem ops here */





}
```

```
int remove()
{
    /* Insert sem ops here */





    item = remove_item();
    /* Insert sem ops here */





    return item;
}
```

A. What is the initial value of each semaphore?

```
mutex = ________

items = ________

slots = ________
```

When inserting, one must obtain a slot by calling P(slots)

Then, obtain the mutex lock, with P(mutex). After completing the insertion, V(mutex) and also V(items) so that consumers may gain access to the remove function. The producer does not increment slots because there are now less places to put items, and only consumption will free up slots.

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */




    add_item(item);
    /* Insert sem ops here */




}
```

```
int remove()
{
    /* Insert sem ops here */




    item = remove_item();
    /* Insert sem ops here */




    return item;
}
```

A. What is the initial value of each semaphore?

`mutex` = 1

`items` = 0

`slots` = 10

When removing, one must know there is something to remove.

So, a consumer waits with P(items), gains entry to the critical section with a P(mutex), handles the removal of an item, does V(mutex) and V(slots), freeing a spot for a producer to post an item. It never increments items because the item is now removed

When inserting, one must obtain a slot by calling P(slots)

Then, obtain the mutex lock, with P(mutex). After completing the insertion, V(mutex) and also V(items) so that consumers may gain access to the remove function. The producer does not increment slots because there are now less places to put items, and only consumption will free up slots.

B. Add the appropriate P and V operations to the psuedo-code for the `insert` and `remove` functions:

```
void insert(int item)
{
    /* Insert sem ops here */



    add_item(item);
    /* Insert sem ops here */



}
```

```
int remove()
{
    /* Insert sem ops here */



    item = remove_item();
    /* Insert sem ops here */



    return item;
}
```

Signals – Spring 2011 #11

```
int counter = 0;

void handler1(int sig) {
    printf("%d", counter);
    kill(getpid(), SIGUSR2);
}

void handler2(int sig) {
    counter = 5;
    printf("%d", counter);
}
```

```
int main(int argc, char *argv[])
{
    int pid;

    signal(SIGUSR1, handler1);
    signal(SIGUSR2, handler2);

    if ((pid = fork())) {
        kill(pid, SIGUSR1);
    } else {
        counter++;
        printf("%d", counter);
    }

    return 0;
}
```

Using the following assumptions, list all possible outputs of the code:

- All processes run to completion and no system calls will fail

- printf() is atomic and calls fflush(stdout) after printing argument(s) but before returning

Signals – Spring 2011 #11

- Child does all printing.
- If it handles SIGUSR1 before it increments count, "0" will print first.
  - If child does not receive or handle SIGUSR 2, then the counter will increment print "1" and exit.
  - If the child does handle it, it could do so before incrementing, between increment and printf, or after printf. The results are "56","55", and "15", respectively
- This accounts for "01", "056", "055", and "015"

Signals – Spring 2011 #11

- Alternatively, the child could handle SIGUSR1 After incrementing count, but before printf. This would print “1”.
  - It could then handle SIGUSR2 either immediately thereafter, printing “55”, or after the printf, printing “15”.
  - Or It could fail to handle SIGUSR2 at all, this would just print “1” again.
- This accounts for “155” and “115” and “11”

Signals – Spring 2011 #11

- Combining this list with the option where the child completes before the parent executes, printing just “1” gives us the complete list:

- 1, 01, 11, 015, 055, 056, 115, 155

Processes – Fall 2012 #8

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

A.	01432	Y	N
B.	01342	Y	N
C.	03142	Y	N
D.	01234	Y	N
E.	03412	Y	N

Processes – Fall 2012 #8

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

- A. 01432 - No
- B. 01342 – Yes
  - Parent runs, prints "1"
  - Child runs, prints "34"
  - Parent prints "2"

Processes – Fall 2012 #8

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

- C. 03142 – Yes
  - Child runs, prints “3”
  - Parent runs, prints “1”
  - Child runs, prints “4”
  - Parent prints “2”
- D. 01234 - No

Processes – Fall 2012 #8

```
int main()
{
    int val = 2;

    printf("%d", 0);
    fflush(stdout);

    if (fork() == 0) {
        val++;
        printf("%d", val);
        fflush(stdout);
    }
    else {
        val--;
        printf("%d", val);
        fflush(stdout);
        wait(NULL);
    }
    val++;
    printf("%d", val);
    fflush(stdout);
    exit(0);
}
```

- E. 03412 – Yes
  - Child runs, prints “34”
  - Parent prints “12”