Sample Long Form Questions

OS Overview; kinds of operating systems

Question 1

Exokernels partition hardware resources and then give each process direct access to their portion of the hardware. Describe how this might work (or not work) for the following devices:

1. Hard drive
2. Sound card
3. Network card
4. Screen

Question 2

(Tanenbaum Ch 1, Problem 5)

On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

Question 3

(Tanenbaum Ch 1, Problem 12)

Which of the following instructions should be allowed only in kernel mode?

1. Disable all interrupts.
2. Read the time-of-day clock.
3. Set the time-of-day clock.
4. Change the memory map.

Question 4

(Adapted from MIT 6.828 exam question 1, Fall 2011)

Unix's API is carefully designed so that programs compose easily. As an example, `write` and `read` don't take an offset as argument, but instead the kernel maintains an offset for each file descriptor. The following shell command illustrates how this unusual API simplifies composing programs:

```
(echo Your files: ; ls) > out
```

Explain why not having an offset in `read` or `write` helps here.

PC Hardware, Assembly Language, System Calls

Question 1

(Adapted from MIT 6.828 exam question 4, Fall 2007)

Explain why it would be awkward for to give a process different data and stack segments (i.e. have DS and SS refer to descriptors with different BASE fields).

Question 2

(Adapted from MIT 6.828 exam question 5, Fall 2007)

When a user-space process executes `int`, `int` switches to a stack specified in the TSS. xv6 sets up the TSS to point to a stack in kernel memory. What might go wrong if the `int` instruction didn't change stacks?

Question 3

The following shows a portion of the code from the implementation of `exec()` in xv6 that sets up the initial stack. `sp` refers to the stack pointer of the process, and `copyout(pgdir, dest, src, n)` just copies *n* bytes of data from *src* to *dst*. For this question we will assume the stack (`sp`) starts at `0x100`, and that the stack area has been initialized to all zeros.

```
uint sp = 0x100;

// Push argument strings, prepare rest of stack in ustack.
for(argc = 0; argv[argc]; argc++) {
  if(argc >= MAXARG)
    goto bad;
  sp = (sp - (strlen(argv[argc]) + 1)) & ~3;
  if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
    goto bad;
  ustack[3+argc] = sp;
}
ustack[3+argc] = 0;

ustack[0] = 0xffffffff;  // fake return PC
ustack[1] = argc;
ustack[2] = sp - (argc+1)*4;  // argv pointer

sp -= (3+argc+1) * 4;
if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;
```

Suppose we run "cat README" at the xv6 command line. Draw the stack that will be set up, and give the value of `sp` when the above code has finished.

Question 4

(Adapted from MIT 6.828 exam question 8, Fall 2012)

Suppose you wanted to change the system call interface in xv6 so that, instead of returning the system call result in EAX, the kernel pushed the result on to the user space stack. Fill in the code below to implement this. For the purposes of this question, you can assume that the user stack pointer points to valid memory.

```
void
syscall(void)
{
  int num;
  int r;

  num = proc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    r = syscalls[num]();
  }
  else {
    cprintf("%d %s: unknown sys call %d\n",
            proc->pid, proc->name, num);
    r = -1;
  }
  // Your code here:

}
```

Question 5

A microkernel moves many services that monolithic operating systems place in the kernel into userspace. A common criticism of microkernels is that they have extra overhead because they have to context switch between multiple user processes to carry out simple tasks. Describe one feature of the x86 architecture that makes context switching slow.

Question 6

Below is the code for swtch() from xv6:

```
# Context switch
#
# void swtch ( struct context ** old , struct context *new);
#
# Save current register context in old
# and then load register context from new.

.globl swtch
swtch :
movl 4(% esp ), %eax
movl 8(% esp ), %edx

# Save old callee - save registers
pushl %ebp
pushl %ebx
pushl %esi
pushl %edi

# Switch stacks
movl %esp , (% eax )
movl %edx , %esp

# Load new callee - save registers
popl %edi
popl %esi
popl %ebx
popl %ebp
ret
```

(a) How is the program counter (%eip) saved and restored by this code?

Memory Management and Virtual Memory

Question 1

Read the following story:

> ***The Thing King and the Paging Game***
> *By Jeff Berryman, c. 1972*
> ***Rules***
>
> 1. *Each player gets several million things.*
> 2. *Things are kept in crates that hold 4096 things each. Things in the same crate are*

called crate-mates.

3. *Crates are stored either in the workshop or the warehouses. The workshop is almost always too small to hold all the crates.*
4. *There is only one workshop but there may be several warehouses. Everybody shares them.*
5. *Each thing has its own thing number.*
6. *What you do with a thing is to zark it. Everybody takes turns zarking.*
7. *You can only zark your things, not anybody else's.*
8. *Things can only be zarked when they are in the workshop.*
9. *Only the Thing King knows whether a thing is in the workshop or in a warehouse.*
10. *The longer a thing goes without being zarked, the grubbier it is said to become.*
11. *The way you get things is to ask the Thing King. He only gives out things by the crateful. This is to keep the royal overhead down.*
12. *The way you zark a thing is to give its thing number. If you give the number of a thing that happens to be in a workshop it gets zarked right away. If it is in a warehouse, the Thing King packs the crate containing your thing back into the workshop. If there is no room in the workshop, he first finds the grubbiest crate in the workshop, whether it be yours or somebody else's, and packs it off with all its crate-mates to a warehouse. In its place he puts the crate containing your thing. Your thing then gets zarked and you never know that it wasn't in the workshop all along.*
13. *Each player's stock of things have the same numbers as everybody else's. The Thing King always knows who owns what thing and whose turn it is, so you can't ever accidentally zark somebody else's thing even if it has the same thing number as one of yours.*

Notes

1. *Traditionally, the Thing King sits at a large, segmented table and is attended to by pages (the so-called "table pages") whose job it is to help the king remember where all the things are and who they belong to.*
2. *One consequence of Rule 13 is that everybody's thing numbers will be similar from game to game, regardless of the number of players.*
3. *The Thing King has a few things of his own, some of which move back and forth between workshop and warehouse just like anybody else's, but some of which are just too heavy to move out of the workshop.*
4. *With the given set of rules, oft-zarked things tend to get kept mostly in the workshop while little-zarked things stay mostly in a warehouse. This is efficient stock control.*

Long Live the Thing King!

Questions

1. What do the following correspond to in computing terms?
    1. Thing
    2. Crate
    3. Warehouse
    4. Workshop
    5. Zark
    6. Grubby
    7. Thing Number
    8. Thing King
    9. Player
2. Why is it difficult to tell how grubby crate is?
3. Suppose that the workshop now employs someone to draw a chalk mark on each crate whenever one of the things inside it is zarked. Describe one way to use this to keep track of how grubby a crate is.

Question 2

Below is the page table used by xv6 when it first enables paging:

```
#define KERNBASE 0x80000000

// Boot page table used in entry.S and entryother.S.
// Page directories (and page tables), must start on a page boundary,
// hence the "__aligned__" attribute.
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  [0] = (0) | PTE_P | PTE_W | PTE_PS,
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

Draw the memory map that this produces, showing virtual memory, physical memory, and the relationship between them.

Question 3

(Tanenbaum Ch 3, problem 36)

A computer has four page frames. The time of loading, time of last access, and the R and M bits for each page are as shown below (the times are in clock ticks):

```
Page   Loaded   Last ref.   R   M
----   ------   ---------   -   -
 0       26        280      1   0
 1      230        265      0   1
 2      140        270      0   0
 3      110        285      1   1
```

1. Which page will NRU replace?
2. Which page will FIFO replace?
3. Which page will LRU replace?

Question 4

If a two level page table is used, each memory reference requires two additional memory lookups to walk the page tables, slowing memory reads down by a factor of 3.

1. Explain what feature modern CPUs have that alleviates this problem.
2. What new problem arises when we switch between virtual address spaces?
3. How can the feature be extended so that context switches are less expensive?

Processes, Threads, and Scheduling

Question 1

Suppose we changed xv6's default scheduler to use a queue of `RUNNABLE` processes, scheduled using round robin. Any code that changed the state of a process from `RUNNABLE` would take it off the queue, and any code that sets the process state to `RUNNABLE` would place it at the end of the queue.

Now, the scheduler would only have to pull the front item off the queue in order to pick the next process to schedule.

Would you expect this to be faster than xv6's current implementation? Why or why not?

Question 2

If a scheduling algorithm causes a process to never receive any CPU time, we say that the process is `starved`. Can any of the scheduling algorithms we've seen cause a process to starve? How?

Question 3

(Tanenbaum Ch 2, problem 8)

Consider a multiprogrammed system with degree of 6 (i.e., six programs in memory at the same time). Assume that each process spends 40% of its time waiting for I/O. What will be the CPU utilization?

Question 4

(Adapted from Tanenbaum Ch 2, problem 45)

Five batch jobs. A through E, arrive at a computer center at almost the same time. They have estimated running times of 10, 6, 2, 4, and 8 minutes. For each of the following scheduling algorithms, determine the mean process turnaround time. Ignore process switching overhead.

1. First-come, first-served (run in order 10, 6, 2, 4, 8).
2. Shortest job first.

Question 5 Consider the following program:

```
# include <stdio .h>
# include <unistd .h>

int main ( void ) {
  int i = 0;
  for (i = 0; i < 4; i++) {
      fork ();
      printf (" foo \n");
  }

  return 0;
}
```

How many times will be "foo" printed?

How many processes will be created (including the initial process)?

Question 6 Why do you need exec() in addtion to fork() to launch new programs?

Question 7 Name 3 system calls from XV6.

Question 8 Write a program that creates a zombie process.

Question 9 The following is the code that implements the kill system call in xv6:

Assembly code:

```
kill :
  mov $0x6 ,% eax
  int $0x40
  ret
```

C code:

```
1 // Fetch the int at addr from the current process .
2 int
3 fetchint ( uint addr , int *ip)
4 {
5       if( addr >= proc ->sz || addr +4 > proc ->sz)
6           return -1;
7       *ip = *( int *)( addr );
8       return 0;
9 }
10
11 // Fetch the nth 32- bit system call argument .
12 int
13 argint (int n, int *ip)
14 {
15      return fetchint (proc ->tf ->esp + 4 + 4*n, ip );
16 }
17
18 int
19 sys_kill ( void )
20 {
21      int pid ;
22
23      if( argint (0, &pid ) < 0)
24          return -1;
25      return kill (pid );
26 }
```

(a) On line 15 of the C code, in the argint function, why do we add 4 to proc->tf->esp?

(b) On that same line, why do we multiply n by 4?

Question 10 What are the 4 process states in XV6 and what can cause a state change? Can you draw the state transtion diagram.

Question 11 In a batch system, shortest job rst (SJF) provides provably optimal turnaround time, assuming all jobs are available from the beginning. If all jobs are not available at the start, SJF may not be optimal. Construct an counterexample (a set of jobs, their run times, and the times they arrive) where SJF scheduling gives a higher turnaround time than some other ordering of jobs.