**PC Hardware, Assembly Language, System Calls**

\*\*Question 1\*\* 1. On x86, if the `EAX` register holds the value 0x712ab211, what value does the `AH` register have?

ANSWER: B2

2. In the following assembly program, what is the value of the `EAX` register when the `done` label is reached?

start:

mov $0, %eax

jmp two

one:

mov $1, %eax

two:

cmp %eax, $1

je done

call one

mov $10, %eax

done:

jmp done

ANSWER: 1

\*\*Question 2\*\* Below is the code for `fetchint` and `argint` in xv6:

// Fetch the int at addr from the current process.

int

fetchint(uint addr, int \*ip)

{

if(addr >= proc->sz || addr+4 > proc->sz)

return -1;

\*ip = \*(int\*)(addr);

return 0;

}

// Fetch the nth 32-bit system call argument.

int

argint(int n, int \*ip)

{

return fetchint(proc->tf->esp + 4 + 4\*n, ip);

}

Suppose we removed the check `(addr >= proc->sz || addr+4 > proc->sz)` (which, as you will recall, is there to guard against malicious user-space programs trying to crash the kernel or read memory they're not supposed to). Now, finish the following snippet of a malicious user-space program written in assembly so that it will crash the xv6 kernel:

mov $0x6, %eax ; kill(int pid) is system call 6

int $0x40 ; execute system call interrupt

ANSWER: Assuming that address FBADBEEF is outside the process address space

MOV $FBADBEEF, %ESP

\*\*Question 3\*\* Using the xv6 system calls below, write C code that creates a file named `hello.txt` and puts the string `hello world` into it. You do not need to write out the include statements or even a proper main function; just include the operations needed to open, write to, and close the file.

#define O\_RDONLY 0x000

#define O\_WRONLY 0x001

#define O\_RDWR 0x002

#define O\_CREATE 0x200

int open(char \*filename, int mode);

int write(int fd, void \*buf, int sz);

int close(int fd);

ANSWER:

fd = open(“hello.txt”, O\_CREATE | O\_WRONLY );

char buffer[] = “hello world”;

write(fd, buffer, strlen(buffer));

close (fd)

**Memory Management and Virtual Memory**

\*\*Question 1\*\*What is the difference between a physical address and a virtual address?

ANSWER: Real Memory uses Physical addresses. These the numbers that the memory chips react on the bus. Virtual addresses are the logical addresses that refer to a process address space. Therefore a machine with a 32 bit address space can generate virtual addresses up to 4GB regardless of whether the machine has more or less memory than 4 GB.

\*\*Question 2\*\*Consider a swapping system in which memory consists of the following hole sizes in memory order: 10

MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB, and 15 MB. Which hole is taken for successive segment requests of:

\* 12 MB

\* 10 MB

\* 9 MB

for first fit? Now repeat the question for best fit.

ANSWER:

First Fit => Takes 20 MB, 10 MB, 18 MB

Best Fit => Takes 12, 10 and 9.

\*\*Question 3 \*\*Why is the principle of locality crucial to the use of virtual memory?

ANSWER: Processes exhibit a locality of reference, meaning that during any phase of execution, the process references only a relatively small fraction of its pages. The set of pages that a process if currently using is it’s working set. If the entire working set is in memory, the process will run without causing many faults until it moves into another execution phase.

\*\*Question 4 \*\*What does TLB stand for and what is it’s purpose?

ANSWER: It stands for Translation Lookaside Buffer, it’s usually inside the MMU and consists of a small cache that contains entries that map virtual pages to physical page frames.

\*\*Question 5 \*\*Consider the following C program:

Int X[N];

Int step = M; /\* M is some constant \*/

For (int i=0; I < N; i+= step) X[i] = X[i] + 1;

a) If this program is run on a machine with with a 4KB page size and 64 entry TLB, what values of M and N will cause a TLB miss for every execution of the inner loop?

>> M has to be at least 4096 to ensure a TLB miss every time we access X. Since N affects only how many times X is accessed, any value of N will do.

b) Would your answer in part (a) be different if the loop were repeated many times? Explain.

>> M should be at least 4096 to ensure a TLB miss for every access to an element of X. But now N should be greater than 256K.

**Processes, Threads, and Scheduling**

\*\*Question 1\*\*Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred twice in the list? Can you think of any reason for allowing this?

ANSWER: If a process occurs multiple times in the list, it will get multiple quanta per cycle. This approach could be used to give more important processes a larger share of the CPU. But when the process blocks, all entries better be removed from the list of runnable processes.

\*\*Question 2\*\* 1. The register set is generally considered to be a per-thread rather than a per-process item. Why? After all, the machine has only one set of registers.

ANSWER: When a thread is stopped, it has values in the registers. They must be saved, just as when the process is stopped, the registers must be saved. Multiprogramming thread is no different than multiprogramming processes, so each thread needs its own register save area.

2. Why would a thread ever voluntarily give up the CPU by calling \*thread\_yield\*? After all, since there is no periodic clock interrupt, it may never get the CPU back.

ANSWER: The key here is that they are threads not processes. Threads in a process cooperate. They are not hostile to one another. If yielding is needed for the good of the application, then a thread will yield. After all, it is usually the same programmer who writes the code for all of them.

\*\*Question 3\*\*Describe conditions that need to occur for a“priority inversion”bug to happen.

ANSWER: The priority inversion bug occurs when a low-priority process is in its critical region and suddenly a high priority process becomes ready and is scheduled. If it uses busy waiting it will run forever. With user-level threads, it cannot happen that a low-priority thread is suddenly preempted to allow a high-priority thread run. There is no preemption. With kernel level threads this problem can arise.

**Drivers and I/O**

\*\*Question 1\*\*What problem does double buffering solve?

ANSWER: Having double the amount of space to store data in the kernel and being able to let the user process read data from one of the buffers while the second one is being filled.

\*\*Question 2\*\*Suppose a printer prints one character at a time, and issues an interrupt when it is ready to print another. An interrupt handler for this device might look like:

// count: total bytes to be printed

// p: the data buffer containing data to print

// i: the index of the next byte to be sent to the printer

if (count == 0) {

unblock\_user();

} else {

\*printer\_data\_register = p[i];

count = count − 1;

i = i + 1;

}

acknowledge\_interrupt();

return\_from\_interrupt();

In this code, the interrupt is not acknowledged until after the next character has been output to the printer. Could it have equally well been acknowledged right at the start of the interrupt service procedure? If so, give one reason for doing it at the end. If not, why not?

ANSWER: If the printer can only print one character at a time, it probably only has space for one character so if it lacks the interrupt before being ready to print another, it might run out of storage space and loose data.

\*\*Question 3\*\* The rate at which a 300 dpi scanner produces data is 1 MB/sec. An 802.11b wireless network has a maximum transmission rate of 900KB/s. Can documents be sent out on the network as fast as they are scanned? Why or why not?

ANSWER: Even assuming double buffering, we can’t sent them as fast because 900KB < 1 MB. We produce 1MB and we can only send 900Kb.

**Concurrency**

\*\*Question 1\*\* Suppose that we have an atomic compare-and-swap instruction that atomically compares a variable with some value and swaps them if they are not equal:

int compare\_and\_swap(int \*var, int val);

Write implementations of `void acquire(int \*lock)` and `void release(int \*lock)` that use this instruction to implement a \*spin lock\* (that is, a lock that loops until it is able to acquire exclusive access to the lock). Note that we will assume here that each lock is represented by a global integer variable.

Answer:

void acquire (int \*lock) {

while (compare\_and\_swap(&lock, 1) != 0)

;

}

void release (int \*lock) {

compare\_and\_swap(&lock, 0)

}

\*\*Question 2\*\* Recall the parallel hashtable implementation from Homework 4:

#define NUM\_BUCKETS 5 // Buckets in hash table

typedef struct \_bucket\_entry {

int key;

int val;

struct \_bucket\_entry \*next;

} bucket\_entry;

bucket\_entry \*table[NUM\_BUCKETS];

// Inserts a key-value pair into the table

void insert(int key, int val) {

int i = key % NUM\_BUCKETS;

bucket\_entry \*e = (bucket\_entry \*) malloc(sizeof(bucket\_entry));

if (!e) panic("No memory to allocate bucket!");

e->next = table[i];

e->key = key;

e->val = val;

table[i] = e;

}

Suppose we have two threads inserting keys with `insert()` at the same time. Describe the exact sequence of events that results in a key getting lost.

Answer: Both of the threads set e->next = table[i] and/or table[i] = e;

\*\*Question 3\*\* Consider the following allocation and request matrices, where E is the vector representing the resources of each type that exist in the system and A is the vector representing the resources currently available.

E = (3, 5, 4) A = (0, 4, 2)

Current Allocation Matrix Request Matrix

[ 1 0 1 ] [ 0 2 0 ]

[ 1 1 1 ] [ 2 0 0 ]

[ 1 0 0 ] [ 1 1 4 ]

Is this system deadlocked? (Show how you arrived at that answer)

Answer:

It is not deadlocked

**Filesystems**

\*\*Question 1\*\* Suppose we have a non-journaled filesystem that uses i-nodes, and a file delete operation that consists of the following actions:

1. Mark the i-node for the file as free in the filesystem bitmap.

2. Mark the data blocks for the file as free in the filesystem bitmap.

3. Remove the directory entry for the file from the directory.

Now suppose that we have a crash after step 2.

1. Describe a scenario where this results in file data being corrupted.

ANSWER: Since event #3 above didn’t happen(remove directory entry) for file1, the entry for the file is still there when we reboot. However the block where the file lived, is marked as free so another file say file2 could come along and use this block. This would lead to filecorruption.

2. How would a filesystem checker like `fsck` that runs at boot detect and fix this condition?

ANSWER:Check that all the directory entry files have corresponding inodes.

\*\*Question 2\*\*In the xv6 logging filesystem, filesystem operations are grouped into transactions, where each transaction consists of the following operations:

1. Write each modified block to the log area, along with its eventual destination.

2. Write a commit record.

3. For each entry in the log, copy the block to its final destination.

4. Clear the log.

For each of these steps, describe what would happen if the system crashed during that step, saying what xv6 would do when it reboots and how this would guarantee that the transaction is carried out atomically (that is, every operation is carried out, or none of them are).

Answer: Again, take a look at the slides and take into account that basically a transaction is complete or not. If it’s complete it’s carried out, if it’s not complete it’s ignored. 😊😊

\*\*Question 3\*\* Suppose we have a filesystem with a block size of 512 bytes and an i-node defined as follows:

#define BLOCKSIZE 512

struct inode {

short type; // File type

short major; // Major device number (T\_DEV only)

short minor; // Minor device number (T\_DEV only)

short nlink; // Number of links to inode in file system

uint size; // Size of file (bytes)

uint blocks[32];

uint indirect;

};

That is, it has 32 direct block pointers and one indirect block pointer. What size (in bytes) is the largest file we can create using this system?

Answer: We have 32 pointers to blocks plus one block that stores just pointers to blocks. So assuming a 32 bit system (4 bytes per pointer) this would be 512/4 = 128 + 32 = 160 block pointers. 160 \* 512 = 81K.

**Security**

\*\*Question 1\*\* Consider the following program:

int main(int argc, char \*\*argv) {

char magic[4];

int winner = 0;

// Copy command line input into magic var

strcpy(magic,argv[1]);

// Do secret computation to check for magic value

if (((magic[0] \* 0x2115) + (magic[1] \* 1222) ^ (magic[2] << 3)) == 0xbeef)

winner = 1;

if (winner) printf ("You win!\n");

else printf("You lose\n");

return 0;

}

When run, the stack layout for the `main()` function looks like:

0x1000 magic[4]

0x1004 winner

0x1008 saved EBP

0x100c return address

1. This program has a buffer overflow. Find it, and use it to give an input (i.e. a value for `argv[1]`) that will cause the program to print "You win!".

ANSWER: 11111

2. Which of the following (if any) would prevent the problem from being exploited?

\* DEP

\* ASLR

\* Stack canaries

ANSWER: Potentially only stack canaries if we put a canary after each stack element.

\*\*Question 2\*\* 1. Explain how adding a \*salt\* to a password makes password cracking more difficult.

ANSWER: to avoid precomputation attacks on passwords Instead of just storing the password, we generate a random string called the salt, then we compute a hash of the password+salt and store on disk salt and hash.

2. Why would we want a password hashing algorithm to be slow?

ANSWER: To avoid brute force attacks by hackers.

**3224 Final**

\*\*Question 1\*\* parallel hashtable implementation

#define NUM\_BUCKETS 5

typedef struct \_bucket\_entry{

int key;

int val;

struct \_bucket\_entry \*next;

}

bucket\_entry \*table[NUM\_BUCKETS];

//inserts a key-value pair into the table

void insert(int key, int val){

int i = key % NUM\_BUCKETS;

bucket\_entry \*e = (bucket\_entry \*) malloc (sizeof(bucket\_entry));

if (!e) panic ("no memory to allocate bucket!");

e->next = table[i];

e->key = key;

e->val = val;

table[i] = e; (line 19)

}

Suppose we have two threads inserting keys using insert() at the same time.

a)Describe the exact sequence of events (ie, the order that the statements in insert() would have to be run by each thread\_ that results in a key getting lost.

b)In an attempt to fix the problem, suppose we add code that locks the table just before line 19 and unlocks it right afterward. Would this fix the problem? Why/why not?

Answer: No, since line 16 still allows for two threads to overwrite one another which will lead to a key getting lost

\*\*Question 2\*\* The program starts executing at the start label. Values starting with $ are constants

Assembly Code:

start:

mov $0, %eax

jmp two

one:

move $0x1234, %eax

ret

two:

cmp $0x1234, %eax

je done

call one

mov $0XBADDF00D, %eax

done:

jmp done

a) what is the value of the EAX register when execution reaches the done label?

EAX: $0xBADDF00D F0 = AH 0D = AL F00D = AX

b) What is the value of AH at the same point?

AH : F0

\*\*Question 3\*\* a)Explain how the system timer, which triggers an interrupt at regular intervals, allows a system with a single CPU to create the illusion that multiple processes are running simultaneously

Answer: The CPU is divided among different processes. Two or more processes can have the illusion or being run simultaneously as each process is given a quanta to run on the CPU. This allows many processes to run at the same time, instead of having one process from the beginning to completion.

b)If there were no system timer, would it still be possible to run more than one process? What would the downsides be?

Answer: You can use a lock. Once a process is done with the CPU and is waiting for IO input, it can release the lock, signaling to other processes that they can acquire the lock and use the CPU. Downsides: the programmer must remember to release the lock of else the process may never run. Also, acquiring and releasing locks is a resource intensive process.

c)Is there any downside to having the timer interrupt more frequently? If so, what is it?

Answer: A process may not finish its computation on the CPU due to an interrupt. It will have to wait for another CPU slot, making it slow down the program.

\*\*Question 4\*\* Wendy Webdev has read that it’s a good idea to use a slow password hashing function. When building the authentication system for her new web site, she creates the following function to be used wen checking user passwords:

char \* slowhash(char \*password){

sleep(5); //wait 5 seconds

return fasthash(password);

}

a)Is this a security improvement over just using fasthash?

Answer:

b)What else could Wendy do to better protect passwords stored in the database?

Answer: Wendy could “salt” the password by adding a randomly generated to the password before hashing. This “salt” is stored on the dish size(?) the hash. It will be harder to brute force the “pass+salt” since longer than just the password.

F The size of virtual storage is limited by the actual number of main storage locations

T It is the responsibility of the operating system to control the execution of processes

? A process that is waiting for access to a spinlock does not consume processor time

F A function call and a system call are basically the same thing and do the same amount of work

T All types of UNIX files are administered by the OS by means of inodes

T The FAT File System is kept in memory

T Double buffering is when a process transfers data to (or from) one buffer while the operating system empties (or fills) the other

F Timestamps for a file are a reliable attribute

F XV6 supports multiple processors

F The principle of locality states that program and data references within a process do not tend to cluster

a)The four main structural elements of a computer system are:

Processor, Main Memory, I/O Modules, and System Bus

b)A semaphore is an integer value, that can count higher than 1, and is used for signaling among processes.

c)A situation in which a runnable process is overlooked indefinitely by the scheduler, although it is able to process, is starvation

d)A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pas, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time

e)The Direct Memory Access unit is capable of mimicking the processor and of taking over the system bus just like a processor

a)Virtual memory is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory

b)A quanta is the maximum amount of time a process can execute before being interrupted

c)This mutual exclusion mechanism doesn’t work in uniprocessor systems: spinlock

d)In XV6 during an interrupt handler we would need to use a lock and also need to disable interrupts in order to avoid reentrancy

Notes

Pipe – connect two processes

System call – goes from low-privilege to high-privilege (user->kernel)

Interrupt descriptor table – handles system calls, interrupts, & exceptions

Trap frame – pushes all general-purpose registers onto the stack + registers from CPU, used to restore CPU state exactly when we return from system call

If a process is prepared to catch a particular signal, then when it arrives, a signal handler is run. If the process is not prepared to handle a signal, then its arrival kills the process

Continuously testing a variable until some value appears is called **busy waiting**. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a **spin lock**

Process = memory(instructions + data + stack)

Process are carried out by executing a system call (except first process, which is created by kernel)