PC Hardware, Assembly Language, System Calls

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 `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*

3. 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).

(a). Now, finish the following snippet of a malicious user-space program written in assembly so that it will crash the xv6 kernel:

```
//Your code here
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
```

(b). Imagine argstr and fetchstr functions that retrieve a pointer to a C string passed in by the user program and copy the string into a buffer in kernel memory. In addition to the checks done by fetchint, what extra checks (if any) would fetchstr need?

Answer: *Since a string is not a fixed length. We need to check for null terminated '\0' which is the character that ends the string.*

4. **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

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

Answer: *Real Memory uses Physical addresses. These are 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.*

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.

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.*

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.*

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?

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

Answer: (a). 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). 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

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.*

2. **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.*

3. **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.*

4. **Describe the 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

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.*

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 acks the interrupt before being ready to print another, it might run out of storage space and loose data.*

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

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)
 }
```

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;
}
```

(a). 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; The threads will try to map to the same place in the same table while not seeing each other. They are not supposed to be in the same table. Therefore one key maybe seen and one may not be seen*

(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 or why not?

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

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:

Need Matrix = Request Matrix- Allocation Matrix

```
[ 0 2 0 ]
[ 1 0 0 ]
[ 0 1 4 ]
```

Process 1 Completes because A = [0, 4, 2] > [0, 2, 0]. First row of Allocation is freed Now A = [1, 4, 3]
Process 2 Completes because A = [1, 5, 3] > [1, 0, 0]. Second row of Allocation is freed. Now A = [2, 5, 4]
Process 3 Completes because A = [2, 5, 4] > [0, 1, 4]. Third row of allocation is Freed. Now A =[3, 5, 4]

Filesystems

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.

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 file corruption.*

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.*

3. **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:

Step 1: While you write log to memory, it crashes. So it didn't write your changes. So, it doesn't commit.
Step 2: What about if you commit and XV6 doesn't know how much you changed? Its confused. So it doesn't change anything because it doesn't know how much we change.
Step 3: Copy the blocks to its final destination. Now you know what blocks you changed, and how many you changed. While saving, it crashes. You lose everything because you didn't save it to disk.
Step 4: Clear the log by setting the count field in the log header to 0. You already make all the changes. When the XV6 reboots, it sees that there are changes left over so it changes log header to 0. (but the changes are made).

4. **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

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
```

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.*

3. **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.*

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

Answer: *To avoid brute force attacks by hackers.*

Practice Questions

1. 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] ;

Il 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 using insert ( ) at the same time.

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

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

(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 or why not?

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

2. Consider the following assembly program. The program starts executing at the start label. Values starting with $ are constants.

Assembly Code:

```
start:
        mov  $0, %eax
        jmp  two
one:
        mov  $0x1234, %eax
        ret
two:
        cmp  $0x1234, %eax
        je  done
        call one
        mov $0XBADDFOOD, %eax
done:
         jmp  done
```

a. What is the value of the EAX register when execution reaches the done label?

Answer: *$0xBADDFOOD*

AH: FO
AL: OD
AX: FOOD

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

Answer: *AH: FO*

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 of 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 beginning to completion.*

3(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 or else the process may never run. Also, acquiring and releasing locks is a resource intensive process.*

3(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.*

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 when 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? Justify your answer.

Answer: *Yes this is a security improvement because it will slow down the slowhash function, and will make it harder for a hacker to use brute force to decipher the password.*

(b). What else could Wendy 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 side the hash. It will be harder to bruteforce the "password + salt" since longer than just the password.*

5. **Consider a swapping system in which memory consists of the following hole sizes, in memory order: 10MB, 4MB, 20MB, 18MB, 7MB, 9MB, 12MB, and 15MB.**

(a) If we make requests for 12 MB, 10MB, 8MB, and 13MB, in that order, which holes will be chosen for first fit?

Answer:

12MB -> 20 MB
10MB -> 10MB
8MB -> 20MB
13MB ->18MB

(b) For the same requests, what holes will be chosen for worst fit?

Answer:

12MB -> 20MB
10MB -> 18MB
8MB -> 12MB
13MB -> 15MB

6. **Below is the code for a modified version of the xv6 scheduler. The only change is that a break statement has been inserted at line 29.**

```
1  void
2  scheduler(void)
3 {
4  struct proc *p;
5
6    for (;;){
7    // Enable interrupts on this processor.
8    sti ();
9
10   // Loop over process table looking
for process to run.
11   acquire(&ptable.lock);
12   for(p = ptable.proc; p <
&ptable.proc[NPROC]; p++){
13    if(p->state != RUNNABLE)
14    continue;
15
16   // Switch to chosen process. It is the
process's job
17   // to release ptable.lock and then
reacquire it
18   // before jumping back to us.
19   proc = p;
20   switchuvm(p);
21   p->state = RUNNING;
22   swtch(&cpu->scheduler, proc-
>context);
23   switchkvm ();
24
25   // Process is done running for now.
26   // It should have changed its p->state
before coming back.
27   proc = 0;
28
29    break; // CHANGED
30   }
31   release(&ptable.lock);
32     }
33  }
```

(a). Explain what effect this change will have.

Answer*: The scheduler wants to continue to look for another ready process. It will only run the first process it finds. This will turn the scheduler into first come first serve. The break will break it out of the loop before it checks everything in the ptable.*

(b). Is this new scheduling algorithm fair (i.e., does it give each process an equal share of the CPU)? Why or why not?

Answer: *No its not fair the schedule will allocate the CPU to whichever process is near to the front of the process table. It may be entirely table that the process in the later portion in the table never gets ran as ptable will be refreshing each time*

7. **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:**

```
1  // Note: count, p, i are all global
variables
2  // count: total bytes to be printed
3  // p: the data buffer containing data to
print
4  // i: the index of the next byte to be
sent to the printer
5  void handler () {
6  acknowledge_interrupt ();
7  if (count == 0) {
8        unblock_user ();
9  }  else   {
10        *printer_data_register = p[i];
11        i = i + 1;
12        count = count - 1;
13     }
14     return_from_interrupt ();
15  }
```

(a) Because the interrupt is acknowledged on line 6, another interrupt could occur while we are executing lines 7-15, causing handler to be invoked again. This could cause a reentrancy bug. How would this bug manifest? (Hint: It may be helpful to consider a specific case. For example, what happens if i = 0, count = 1, and p is an array of length 1 containing the single character 'x'? At what line would the interrupt have to occur to see incorrect behavior?)

Answer: *If there is an interrupt right before line 8 where the unblock_user() is called it could cause a bug, since the interrupting thread can overwrite what is currently in the print_data_register and print data that was intended to be printed in the order it is printed.*

(b) Suppose we introduced a lock that was acquired just before line 10 and released just after line 12. Would this solve the bug in part (a)? Why or why not?

Answer: *Yes it would solve the bug as it would not allow another thread to call unblock_user on data that isn't it's own.*

5. **Examine the following program:**

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4  int count = 0;
5  pthread_mutex_t  lock =
PTHREAD_MUTEX_INITIALIZER;
6
7  void * thread_routine(void *arg) {
8        for (int i = 0; i  <  100000; i++){
9              count = count + 1;
10     }
11     return NULL;
12  }
13
14  int main(void)  {
15        pthread_t threads[2];
16        // Start two threads counting
17        pthread_create(&threads[0],
NULL, thread_routine, NULL)
18        pthread_create(&threads[1],
NULL, thread_routine, NULL)
19        //Wait for them to finish
20        pthread_join(threads[0], NULL);
21        pthread_join(threads[1], NULL);
22        printf("count  =  %d\n", count);
23        return 0;
24 }
```

(a) When run, this program prints count = 102790. Why? (You don't have to explain why it prints that specific number, just why it doesn't print 200000).

Answer: *Since count = count + 1 at line 9 is not an atomic statement, you need the have locks before and after the actual for-loop so it doesn't get interrupted mid-way through the statement.*

(b) Where should a lock be acquired and released in order to make this program behave correctly and print count = 200000?

Answer: *A lock should be acquired before line 9, where the thread_routine increments the count. The lock should be unlocked after line 9, which is when the count is done incrementing.*

6. **Recall from Homework 4 that the mappages function in xv6 creates the page table entries needed to map memory of size size from virtual address va to physical address pa in the process whose page directory is pgdir, with permissions perm:**
int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);
The available permission flags are: PTE U (page is accessible in user mode) and PTE W (page is writeable).
Recall also that kalloc allocates a single page (4096 bytes) of kernel memory, and v2p translates a kernel-mode virtual address to a physical address.

char* kalloc(void)
uint v2p(void *a)

(a). Use these functions to write a snippet of code that creates a shared memory region that is 4096 bytes large, shared between two processes whose page directories are named pgd1 and pgd2. Both processes should be able to read and write to the region. You may assume that both processes currently have nothing mapped from virtual address 0x10000 to 0x20000.

Answer: *char* mem=kalloc()*
Mappages(pgd1, 0x10000, 4096, V2P(mem), PTE_W|PTE_U);
Mappages(pgd2, 0x15000, 4096, V2P(mem), PTE_W|PTE_U);

(b). What would you need to change so that one process gets read-write access to the shared memory region while the other only gets read access?

Answer: *Get rid of PTE_W from one of them*

7. **Consider the following C program, which will be compiled and run on a 32-bit x86 machine.**

```
1 void win() {
2 printf ("You win!\n");
3}
4
5 int main(int argc, char **argv) {
6       char magic [4];
7
8       //Copy command line input into
magic var
9       strcpy(magic, argv[1]);
10     //Do secret computation to check
for magic value
11     if (((magic[0] * 0x2115) +
(magic[1] * 1222) ^ (magic[2] << 3)) ==
0xbeef)
12           win();
13
14     return 0;
15 }
```

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

0x1000 magic[4]
0x1004 saved EBP
0x1008 return address

(a). Suppose that when the program is run the win function is at address 0x30303030. Give a value for argv[1] that will result in the program printing "You win!". (Hint: You probably don't want to try and find out a value for magic that satisfies the conditional on line 11. Instead, think "buffer overflow".)

Answer: *111130303030*

(b). Would DEP (Data Execution Prevention) prevent an attacker from reaching the win() function without knowing the correct value for magic? What data should be marked non-executable?

Answer: *No, because we are hijacking the return address, and it will take us to that point in the code regardless of the DEP*

(c). Would stack canaries?

Answer: *Stack canaries would only work if there is a stack canary before the EBP register.*

8. **The code below shows the xv6 code for the sys link system call, which creates a hard link (that is, two directory entries that both point to the same i-node).**

```
1 // Create the path new as a link to the
same inode as old.
2 int
3 sys_link(void)
4{

5    char name[DIRSIZ], *new, *old;
6    struct inode *dp, *ip;
7
8    if(argstr(0, &old) < 0 || argstr(1,
&new) < 0)
9        return -1;
10
11   begin_op ();
12   if((ip = namei(old)) == 0){
13       end_op();
14       return -1;
15   }
16
17   ilock(ip);
18   if(ip->type == T_DIR){
19       iunlockput(ip);
20       end_op();
21       return -1;
22   }
23   // ...
24}
```

On line 18, xv6 checks to see if the file being linked is a directory, and returns an error if it is. What is the purpose of this check? What could go wrong if the check wasn't there (i.e., if hard links to directories could be created)?

Answer: *Hardlink to directory creates loops. This defies the logic of the system. You would have infinitely further paths of files.*

9. **Recall that in 32-bit x86, page directories and page tables are each made up of 1024 32-bit entries. Suppose you have a 32-bit x86 system with 4MB of physical memory.**

(a) What is the space overhead of paging for a single process if 4KB pages are used and each virtual address translates to the same physical address? (You may just give a fraction here rather than trying to compute a percentage)

Answers: *With 4KB pages we need one page directory and one page table to map 4MB of physical memory. Each table is 4KB (1024 × 4 bytes), so the overhead is (4KB + 4KB)/4MB = 8KB/4MB = 2/1024 ≈ .002 = 0.2%.*

(b). What is the overhead if 4MB pages are used?

Answer: *This time we only need a page directory, which is 4KB. Taking the same calculation as before but with 4KB used for paging instead of 8KB, we get an overhead of .001 = 0.1%.*

(c). Can you think of a way to reduce the overhead of paging to just 4 bytes on this system? Hint: You can assume that no valid program will try to use memory at virtual address 0, or reference memory above 4MB.

Answer: *We know that no one will try to access memory above 4MB, so we only need one four-byte entry in our page directory if we use 4MB pages. This entry can be set to map virtual addresses 0-4MB to physical addresses 0-4MB. Now, where can we put the page directory? If we place the page directory at physical address 0, and we know that no one will try to use memory at that address, then the only space that programs can't use for data is the first 4 bytes of memory.*

****

True and False

False: *The size of virtual storage is limited by the actual number of main storage locations*

True: *It is the responsibility of the operating system to control the execution of processes.*

False: *A process that is waiting for access to a spinlock does not consume processor time.*

False: *A function call and a system call are basically the same thing and do the same amount of work*

True: *All types of UNIX files are administered by the OS by means of inodes.*

True: *The FAT File System is kept in memory.*

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

False: *Timestamps for a file are a reliable attribute*

False: *XV6 supports multiple processors.*

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

True: *The addresses a program may use to reference memory are distinguished from the addresses the memory system uses to identify physical storage sites.*

True: *Most of the memory management issues confronting the operating system designer are in the area of paging when segmentation is combined with paging.*

False: *Segmentation is not visible to the programmer.*

True: *The placement policy determines where in real memory a process piece is to reside.*

True: *Virtual memory allows for very effective multiprogramming and relieves the user of the unnecessarily tight constraints of main memory.*

False: *The principle of locality states that program and data references within a process do not tend to cluster.*

False: *The smaller the page size, the greater the amount of internal fragmentation.*

True: *The design issue of page size is related to the size of physical main memory and program size.*

True: *Segments may be of unequal, indeed dynamic, size.*

False: *The page currently stored in a frame may still be replaced even when the page is locked.*

True: *One way to counter the potential performance problems of a variable-allocation global scope policy is to use page buffering.*

False: *The Page Fault Frequency (PFF) policy evaluates the working set of a process at sampling instances based on elapsed virtual time.*

True: *A precleaning policy writes modified pages before their page frames are needed so that pages can be written out in batches.*

True: *UNIX is intended to be machine independent; therefore its memory management scheme will vary from one system to the next.*

****

Multiple Choice

1. **The four main structural elements of a computer system are:**

Answer: *Processor, Main Memory, I/O Modules, and System Bus*

2. **A _____ is an integer value, that can count higher than 1, and is used for signaling among processes**

Answer: *Semaphore*

3. **A situation in which a runnable process is overlooked indefinitely by the scheduler, although it is able to proceed, is _____**

Answer: *Starvation*

4. **A real-world example of _____ occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, 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.**

Answer: *Livelock*

5. **The ____ unit is capable of mimicking the processor and of taking over the system bus just like a processor.**

Answer: *Direct Memory Access*

****

Fill In

1. **_____ is a storage allocation scheme in which secondary memory can be addressed as though it were part of main memory**.

Answer: *Virtual Memory*

2. **A _____ is the maximum amount of time a process can execute before being interrupted.**

Answer: *Quanta*

3. **This mutual exclusion mechanism doesn't work in uniprocessor systems:_____**

Answer: *Spinlock*

4. **In XV6 during an interrupt handler we would need to use a lock and also need to _____ in order to avoid reentrancy**.

Answer: *Disable Interrupts*

5. **The address of a storage location in main memory is the _____ .**

Answer: *Real Address*

6. **_____ is the virtual storage assigned to a process.**

Answer: *Virtual Address Space*

7. **_____ is the range of memory addresses available to a process.**

Answer: *Address Space*

8. **The _____ structure indexes page table entries by frame number rather than by virtual page number.**

Answer: *Inverted Page Table*

9. **The _____ states the process that owns the page.**

Answer: *Process Identifier*

10. ***A _____ is issued if a desired page is not in main memory.***

Answer: *Page Fault*

11. **_____ allows the programmer to view memory as consisting of multiple address spaces.**

Answer: *Segmentation*

****

Midterm Concepts

User-space is where ordinary programs run, must trap to Kernel for privilege

Types of Kernel:

Monolithic- 1 large program. Any bug in kernel will bring down entire OS. Ay piece of kernel talks to any other piece through func calls.

Microkernel- Just does scheduling, interprocess communication.

Communication between processes requires context switch

Exokernel- splits up hardware and allows user programs access

Processes:

If a parent exits before its child, child process becomes orphan. If a child process exits and parent didn't call wait() to get its exit status, the child becomes a zombie.

States of Processes:

RUNNING, RUNNABLE, READY, BLOCKED(waiting for IO), SLEEPING, ZOMBIE

Registers and GCC calling Convention:

%Eax, %ecx, %edx- must be trashed by the callee.

%Ebx, %ebp, %esi, %edi- must be saved by callee.

Return value: %eax

Arguments are passed on the stack: f(1, 2, 3) • push 3, push 2, push 1, call <f>.

Pushing something on the stack decrements stack counter.

EIP is program counter that can't be directly accessed.

EBP is typically the same as ESP at function entry

****

General Purpose Reg: EAX, EBX, ECX (counter), EDX, ESI, EDI (source & des index)

Special Purpose Reg: EBP & ESP (Base & Stack pointer)

Segment Reg: CS SS, DS, ES, FS, GS (code, stack, data, extra segment

Program Counter: EIP

****

Scheduling:

Round Robin- Preemptive (uses quantums). Run process until quantum used up and go to next process and repeat.

First Come First Serve- Just single queues of all processes. Finish first before moving down the list. I/O suffers.

Shortest Job First- Assume we know how long each process will take and only choose the shortest process.

Turnaround= $\frac{a+(a+b)+(a+b+c)}{3}$

Guaranteed Scheduling-Of n processes

Lottery Scheduling-

Throughput- $\frac{Jobs\ Completed}{Time\ Interval}$,

Turnaround= $\frac{Time_{(comp.)}-Time_{(subm.)}}{N}$

Priority Scheduling

Process Aging ($T_i = aT_{i-1} + (1-a)T_{i-2}$)

****

Context Switching- Swapping a CPU's registers with the correct data when switching processes.

Switchuvm- switches task state segment (TSS) to mode mode one and changes address space to process's

swtch- does actual work of changing CPU registers

Threads- multiple threads can exist in a process

Unlike proccess, multiple threads within same process share address space, global variables & memory threads are lighter weight than process

Shell- just a normal program that reads 1 line at a time from a user

Fork(): creates exact copy of the current process

```
if (fork() == 0) { printf("child") };
if (fork() == 1) { printf("parent") };
```

****

JUMPS: **JMP** (Always jump), **JE/JZ** (jump if eq/zero), **JNE/JNZ** (jump if !eq/!zero), **JG** (jump if greater), **JGE** (jump if greater/eq), **JL** (jump if less), **JLE** (jump if less/eq)

****

Trap Frame

When we make a system call from the user mode, we trap the frame so that we can restore the state when returning from the kernel mode.

By capturing all of this information in the trap frame structure, we can restore the CPU state exactly when we return from the system call.

When executing the system call, we switch stacks. However the arguments are stored in the user stack. We use the stack pointer value saved in the trap frame to get the arguments.

Arguments are at %esp+4+(4*arg_no)

When fetching arguments, we check to make sure the pointer is not outside proc->sz

In user mode, we can rely on the paging hardware to disallow access to anything outside of process's memory

But in kernel-mode we must do explicit checks, because the kernel has access to all memory syscall() put the return value in proc->tf->eax

So when we get back to userspace, we will have our return value in %eax

****

Change of Process State:

1. Save the context of the processor 2. Update PCB of processor in running state 3. Move PCB of process to appropriate queue 4. Select another process for execution 5. Update PCB of processor selected 6. Update memory management data structure 7. Restore the context of the processor

****

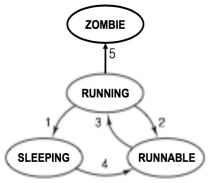
System Call Mechanism

-Have an instruction that initiates the call
-Specify which call we want
-Have the kernel retrieve the system call arguments
-Save the process's current state and restore it when we return
-Do all this securely, without breaking isolation between user and kernel space
-There is priority given to different privilege levels

****

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
5. Process Exits



****

%eip: The Instruction pointer register. It stores the address of the next instruction to be executed. After every instruction execution it's value is incremented depending upon the size of an instrution.
%esp: The Stack pointer register. It stores the address of the top of the stack. This is the address of the last element on the stack. The stack grows downward in memory(from higher address values to lower address values). So the %esp points to the value in stack at the lowest memory address.
%ebp: The Base pointer register. The %ebp register usually set to %esp at the start of the function. This is done to keep tab of function parameters and local variables. Local variables are accessed by subtracting offsets from %ebp and function parameters are accessed by adding offsets to it as you shall see in the next section.

****

DEP (Data Execution Prevention) – A flag to try and prevent data from being overwritten.
Stack Canaries- An extra bit to keep track if that particular stack has been overwritten.
ASLR- Address Space Layout Randomization- Memory protection process that guards against memory overflow.
SEE BETTER EXPLANATION BELOW

****

Translation Lookaside Buffer
1. Start
2. CPU checks the TLB
3. Page Table Entry in TLB?
If Yes-> CPU Generate Physical Address and Updates TLB
If No -> Access Page Table
4. Page in Main Memory?
If Yes->Update TLB
If No-> **Page Fault Routine:**
1. OS Instructs CPU to Read the Page from Disk
2. CPU Activates I/O Hardware
3. Page Transferred from Disk to Main Memory
4. Memory Full?
If Yes-> Perform Page Replacement
If No-> Page Tables Updated and Return to Faulted Instruction.

****

Page Replacement Algorithms
The Optimal Algorithm-
At the time of a fault, consider the set of pages in memory. Given the code executing, they will each be referenced some number of instructions from now. To choose one to evict, just pick the one that's furthest from being referenced.
Not Recently Used (NRU)- We saw that page table entries contain bits that indicate whether the page was recently *referenced* or *modified (R & M)*. We can use these bits to track which pages in memory have not been touched in a while. If a page hasn't been used in a while, it may be a good candidate for eviction. If we just rely on the CPU to mark referenced pages, over time eventually all pages would be referenced. Instead we can have the OS periodically clear the referenced bit (say, every clock tick). Now if the referenced bit is 0, we know the page has not been referenced in at least one clock tick.
First In, First Out (FIFO)- Keep a linked list of pages in the order they were brought into memory. Tail is most recent, head is oldest. To evict, throw out the one at the head of the list
Second Chance- Before throwing out the oldest page, check if it's been referenced. If it has, clear the referenced bit and move it to the back of the list (as though it were a new page). Keep looking through the list for something to evict.
Clock- As an optimization, we can imagine the pages arranged in a circle and keep a pointer (or index) indicating the position of the "clock hand". Clock hand points to the oldest page – so now we can apply Second Chance by just updating the position of the clock hand.
Least Recently Used (LRU)- pages that have been used a lot recently will probably be used again soon. So a good strategy might be: throw out the *least recently used* page.
Not Frequently Used (NFU)- Maintain a list of counters, one per page. At each clock tick, if a page has been referenced, update its counter. Now we have a rough count of how often each page has been referenced over time. A page might be referenced very often early on, and then never referenced again. But its count will remain high for a long time, preventing it from getting evicted. Meanwhile, a page that is referenced periodically (say every 20 ticks) may have a lower count but be in active use.
Aging- To solve this problem, we can have counter values decay over time. Each clock tick, shift all counters right by one bit. Add in the reference bit as the leftmost bit. To evict, just choose the lowest counter value – because more recent references are in the more significant digits, they have greater weight.
Working Set- The set of pages currently being used by a process.
Working Set Algorithm-For performance, we would like to load the exact working set into the process when it's paged back in. In theory, we can just fix some value *k* and then track the pages touched in the last *k* memory references. But once again, doing *anything* on every memory reference is very very slow

****

Page Faults:
Minor page fault – can be serviced by just creating the right mapping
Major page fault – must load in a page from disk to service
Segmentation fault – invalid address accessed; can't service so we usually just kill the program

****

Memory Management Unit- Manages the physical and virtual memory addresses.

****

Concurrency-
Race Condition- Two or more processes run in parallel and output depends on order in which they are executed.
Synchronization- Keeping processes running in sync.
Mutual Exclusion- The way to solve this is through *mutual exclusion* –making sure that only one process has access to the shared resource at a time.
Critical Regions- When accessing shared memory or files
1. No two processes may be simultaneously inside their critical regions
2. No assumptions may be made about speed or the number of CPUs
3. No process running outside its critical region may block any process
4. No process should have to wait forever to enter its critical region
Peterson's Algorithm- Each process indicates its *interest* by setting its entry in the "interested" array. Then, it sets the global *turn* variable to its own process number. Finally, loop until *turn* indicates that it's our turn *and* we see that the other process is no longer interested. There is still a race – but regardless of the winner, only one process will get to enter its critical region

Busy Wait- Whenever a process is waiting to enter a critical section under these schemes, it sits in an infinite loop. This wastes CPU time. It can also interact badly with scheduling.
Priority Inversion- Suppose we're using priority scheduling, and we have a high-priority process **H** and a low priority process **L.** So whenever **H** is runnable, it will always be chosen over **L.** Now, **L** enters a critical region, but is then preempted to run **H. H** wants to enter the critical region, but the lock is already held by **L**, so it enters a busy wait loop. But now **H** is *always* runnable, and will *always* be chosen over **L**, so **L** can never leave its critical region and the system is stuck.
Priority Inheritance- *priority inheritance* for the lock Priority inheritance says that a process holding a lock is elevated to the highest priority of anything waiting for the lock.
Mutex- A mutex is a way to have mutual exclusion without busy waiting. Implementation is very similar to the busy-wait version of critical regions, but instead of looping, we yield the CPU
Semaphore- an integer to count the number of wakeups pending.
Semaphores have two operations: *down*: checks if semaphore is greater than 0, and decrements it if so; otherwise sleep. *up*: increments the value of the semaphore, and if it was previously zero, wakes up one of the sleeping processes at random
Note that these operations are *atomic* and generally implemented as system calls
Read-Copy-Update- If we're okay with some readers seeing an old version of a data structure for a while, we can sometimes avoid locking. The trick is to update the structure but not free deleted items immediately – instead, wait until all readers are done. When can we safely delete old nodes? When there are no more readers. We add an API called a *read-side critical section* that any readers of the structure must call when they read something from it.
Mutexes in Pthreads:

Thread call	Description
Pthread_cond_init	Create a condition var.
Pthread_cond_destroy	Destroy a condition var.
Pthread_cond_wait	Block waiting for a signal
Pthread_cond_signal	Signal another thread and wake up in it
Pthread_cond_broadcast	Signal multiple threads and wake them all up.

Signaling: One thread sends a signal to another thread that something has happened. Signaling makes it possible to guarantee that a section of code in one thread will run before a section of code in another thread.

****

Deadlocks
Acquiring Spinlock

```
Void acquire (struct spinlock *lk) {
   Pushcli(); //disable interrupts to avoid deadlocks
   If (holding(lk))
      Panic("acquire");
  //The xchg is atomic
  //It also serializes, so that reads after acquire are not
  while(xchg(&lk->locked, 1) != 0);

  //Record info about lock acquisition for debugging.
  Lk->cpu = cpu;
  Getcallerpcs(&lk, lk->pcs);
}

//Release the lock
void release(struct spinlock *lk){
   if(!holding(lk))
      panic("release")

lk->pcs[0] = 0;
lk->cpu = 0;

xchg(&lk->locked, 0);

popcli();
}
```

Deadlock: two or more threads are waiting on events that only those threads can generate. Deadlock does not go away by itself
Livelock: thread blocked indefinitely by other thread(s) using a resource. Livelock naturally goes away when system load decreases.

Required Conditions for Deadlock:
Mutual Exclusion: Resources cannot be shared. **Hold and Wait:** A thread is both holding a resource and waiting on another resource to become free.
No preemption: once are thread gets a resource, it cannot be taken away.
Circular Wait: There is a cycle in the graph of who has and who wants what.

Dealing with Deadlock-
Prevention: Design rules so one condition cannot occur
Avoidance: Dynamically deny "unsafe" requests
Detection and Recovery: Let if happen, notice it, and fix it
Ignore- Do nothing, hope it does not happen, reboot often

****

File systems
A *file* is an abstraction – a logical unit of data that is backed by multiple underlying blocks on disk
File Associations as MetaData- HFS+ has a notion of *resource forks* – a hidden set of metadata stored in the filesystem. You can set a specific file to be opened by a particular application, and that information will be saved in the resource fork
Timestamps- Timestamps on files are stored as filesystem metadata. Even regular users can modify the timestamps on files they create! They can't be relied on to prove a file was created at a particular time.
File Directory:
Path- To specify a file, you specify a *path* through the filesystem tree.
Relative Path- We saw that one of the pieces of information stored in a process data structure is the *current working directory*. This allows us to specify files without giving their full path.
Mounting- Connecting multiple file systems together. Attaches the new filesystem hierarchy at an existing (empty) directory.
File Layout- Boot block, superblock, Free space management (bitmap or list), i-nodes, root directory, files and directories.
Possible File Implementations:
-Contiguous allocation
-Linked list (on-disk or in-memory)
-I-nodes
Contiguous Allocation- Conceptually trivial: just put all the blocks for each file one after another on disk. Has many of the same problems of non-virtual memory management: As files are created and deleted, the layout becomes *fragmented*, wasting space.
Advantages:
Simple to implement
Excellent read performance
Disadvantages:
Fragmentation
Linked List Allocation- Inside each file block, keep a pointer to the next block in the file. No issues with fragmentation & wasted space. Storing a linked list on disk has performance issues though: To read disk block *n*, we need to read *n-1* blocks before it. Entire block can no longer be used for data (we need space for the next pointer), so reads of file block must span two physical blocks.
***File Allocation Table (FAT)* and how FAT16/FAT32 filesystem works**-
Improvement on regular linked list implementation: keep a table in memory with just the pointer information. Each entry corresponds to a physical block and contains the pointer to the next block

Disadvantages-
FAT is very large. Suppose we have a 1TB disk, 1KB blocks: (1TB/1KB)entries * (4bytes/entry) = 4GB for FAT.
I-Nodes- A more compact way is to give each file a small data structure that lists its blocks, called an *i-node* (*index node*). Now we only need to store in memory an amount of data proportional to the number of open files. Amount of storage needed to store the mapping grows with the number of files, *not* the total size of the disk.
Implementing Directories-To actually open a file, we supply a path, which gives a list of directories and a filename. The directory entry provides the info to find the disk blocks. A directory must have some representation that stores the list of filenames, attributes, and pointer to the file data. Basically needs to implement the map filename => file data.
Shared Files-
Hard and Symbolic Links-
Hard link: just have each directory store a pointer to the same file information; e.g., same i-node
Symbolic link (symlink): create a new directory entry type, which stores the path to the link target
Hardlink and Symlink Comparison
With hard links, we must keep a count of how many links to the file exist – otherwise we won't know when we can delete a file. Symbolic links have extra overhead – we have to store a full path to the target file.
Keeping Track of Free Space- Linked List or Bitmap.

****

Log Structured File- structure the disk as one big log. Periodically data buffered in memory collected into a single segment and written to the disk.
Buffer writes in memory, then periodically flush them out in one contiguous segment at the end of the log. At the start of the segment, place information about where to find i-nodes and file data within the current log segment. Finally, maintain a map that says which log segment the i-nodes for files/directories can be found. If we overwrite data in a file, or if we delete it, we will waste space (and eventually run out of space). To solve this, a *cleaner thread* constantly runs, removing unused entries from the back of the log and placing old but still-in-use entries at the front.

If the OS crashes or we lose power in a traditional filesystem, we may leave things in an inconsistent state
• For example: wrote file data, but didn't update the directory entry. LFS solves this by keeping a *checkpoint*, which tracks what the most recent consistent filesystem state is .After a crash, we can either just revert to the last checkpoint, or revert and then replay as many log entries as possible (while keeping the filesystem consistent

Journaling Systems- Incorporate recovery in case of a crash into modern filesystems. NTFS, HFS+, and ext3 all support journaling.

Idempotence- Because the operations listed in the journal log entry could be carried out more than once, they must be idempotent (doing it twice is the same as doing it once). For example, marking a block as free in a bitmap is idempotent. Adding a block to a list of free blocks is *not* idempotent. But we can make it idempotent by adding a check to make sure it's not already in the list.

XV6 Filesystem Layers- File descriptor, pathname, directory, Inode, logging, buffer cache, and disk.
Disk layer reads and writes blocks on the IDE drive
Buffer cache layer caches blocks and synchronizes access to them
Logging layer wraps multiple operations in a single atomic *transaction*, providing crash recovery
i-node layer represents files as we saw last time
The directory layer contains a special type of i-node that gives a list of names and i-node pointers
The pathname layer resolves paths to i-nodes
The file descriptor layer provides an abstraction for accessing several kinds of object (pipes, files, devices) as files.
XV6 Disk Layout- Boot sector, super block, inodes, bitmap, data, log.

Buffer Cache Layer- Synchronize access to blocks so that two processes don't try to access the same data simultaneously. Cache commonly used blocks so that we don't have to read from the disk all the time.

LRU Cache Implemented as Doubly Linked List.

Logging Layer- Logging layer allows higher layers to group multiple writes into a single transaction that will be committed atomically.

I-node Layer- i-nodes store pointers to file blocks. Every i-node is the same size and they are stored in a single area on disk, so it's easy to look up an i-node by number: inode_start + inode_num.

i-nodes in xv6 have both an on-disk (struct dinode) and in-memory representation (struct inode)
The type field distinguishes between files, directories and special files (devices)
xv6 maintains a cache of in-memory i-nodes in order to help synchronize access to i-nodes by multiple processes

Directory Layer- Implemented internally much like a file. Its inode has type T_DIR. It's data is a sequence of directory entries. Each entry is struct dirent

****

Security-
Confidentiality – Exposure of data or preventing others from finding out information we don't want them to have
Integrity – Tampering with Data or preventing others from modifying our data without permission
Availability – Denial of Service or preventing others from denying us access to some service ()

Threat Modeling-
-System understanding
-Threat categorization
-Countermeasures and mitigation

Threat Categorization-
STRIDE: **S**poofing, **T**ampering, **R**epudiation, **I**nformation Disclosure, **D**enial of Service, **E**levation of Privilege

Trusted Computing Base
One strategy for building secure operating systems is to organize them into trusted and untrusted components
The goal is that if the *trusted* component performs according to its specification, then some specific set of guarantees about security must hold.

Protection Domain-
A **protection domain** is a set of (object, rights) pairs. A **right**, in this context, means an operation that can be performed on an object

Principle of Least Privilege-
One principle for designing secure systems is *principle of least privilege*, also called the *principle of least authority*.
Idea: Give the minimal set of rights to an object.

Protection Matrix-
Rows represent the domains
Columns represent the objects

Access Control List- attaches to each object a list of domains and the rights for each. we often refer to domains as "**subjects**" or "**principals**"

Discretionary Access Control: users get control over who has access to their files
Mandatory Access Control: some kinds of information cannot be shared

Steganography- Hiding data inside images and in other items.

Hash Functions-
We saw before that one-way functions can let us do useful things like store capabilities
Basic idea: Hash(x) = y
Preimage resistance: It should be very difficult to take y and figure out x
Second preimage resistance: It should be very difficult to find another value z where Hash(z) = y
Collision resistance: It should be very difficult to find x1 and x2 such that Hash(x1) = Hash(x2)
Examples: MD5, SHA-1, SHA-3

How Passwords are Broken-
Precomputation: Spend a lot of time beforehand computing the hash of every possible password, then just look up each hash in your table
Dictionary Attacks: Guess the most common passwords (e.g. using a dictionary), hash them, and compare.

Password Salts-
To avoid precomputation attacks, we can use a *salt*.
Instead of computing and storing Hash(Password), we instead:
-Generate a random string (a **salt**)
- Compute Hash(Salt+Password)
- Store on disk Salt, Hash(Salt+Password)

- Now to do a precomputation, you have to store not just the hash of every possible password (hard but doable) but the hash of every possible password+salt (way too much space)

Kerckhoffs's Principle- Basic principle: it should not matter if the cryptographic *algorithm* is known by everyone
The only secret information should be a *secret key* chosen by the participants

Lamport Hash Chain- The idea is to use a one-way function to create a sequence of hashes
If you repeatedly apply a hash to some starting value, you get a sequence that is:
Easy to compute starting from the seed going forward

Hard to compute starting from a hash value and going backward

Initially, user chooses a password p
Compute fk(p) = f(f(f(...f(p)))) and store this value on the server
To log in, user computes and sends h = fk-1(p)
Server checks that f(h) matches its stored value, and then replaces the current stored value with h

Challenge-Response- The idea is that the server sends a *challenge* (a random value). Client then uses their secret and the challenge to compute a *response*
Server can now verify that the response is the correct one for that challenge

Stack Buffer Overflows- Recall the standard stack frame:
• Local variables Saved frame pointer (optional) Return address
If we try to store too much data in a lock stack variable (e.g. a character array) we will overwrite the frame pointer and return address
When the ret instruction is executed, it will jump to somewhere controlled by user input

Stack Canaries/Cookies-
-Idea: put a special value in between local variables and the return address so that overflowing a local buffer can be detected
-Upon entering the function, set a randomly-generated *cookie* value on the stack and store a backup copy elsewhere
-Before executing a ret, check the stack cookie value against the backup and raise an error if it fails

Address Space Layout Randomization (ASLR)-
Exploiting a buffer overflow typically requires knowing about the precise layout of memory
For example, we may need to know where the stack is located, or where a certain library has been loaded
Thus, to make attackers' lives more difficult, we can place the program, stack, and libraries at random locations each time the program starts

DEP/NX/W⊕X
Another defense is to try and make sure that even if an attacker can overflow a buffer and change the return address, they can't execute code
In the previous example, attacker code was placed into a stack buffer
So, simple solution: don't allow data to be executable!
Generally this requires hardware support
NX bit in x86 page protections

Return Oriented Programming-
Despite DEP, attackers can still run code!
Instead of trying to execute their *own* code, attackers can change the return address to point to existing code in memory
By setting up values on the stack, they can bounce around executing tiny snippets of code ending in ret
Thus, by chaining these together, *arbitrary* computation can be performed – without executing anything marked as data

Dangling Pointers-
An increasingly common and exploited class of vulnerability is the *dangling pointer* or *use after free* vulnerability
Scenario: programmer frees an object, but then continues to use the pointer afterward
Problem: new allocations may use the same space, placing a *different* object at the same location
If an attacker can manage to get his data placed into that freed space, can manipulate the program into executing his code

TOCTOU- One final class of attacks is a *time of check to time of use* attack
These arise any time you have a situation where:
1. A security check on some object is made
2. The program does something with that object
3. An attacker can intervene in between 1 & 2 and cause it to operate on a different (attacker- controlled) object

TOCTOU Example-
Int fd;
If (access ("./my_document", W_OK) != 0){
Exit(1);
Fd = open("./my_document", O_WRONLY)
Write (fd, user_input, sizeof(user_input));

TOCTOU Attack-
To exploit this program, the attacker runs the program with a my_document he controls

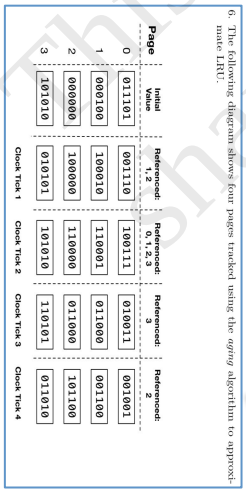
After the access check is done, he deletes my_document and creates a symbolic link from my_document to some sensitive file

Getting the timing right here may be tricky!

The program now opens and writes to my_document, but that now results in writing to the sensitive file

Preventing TOCTOU-
In general, the only way to prevent TOCTOU attacks is through careful API design
In this case, we want to make sure that the file cannot be changed between the access check and the call to open
One way to do this is to open the file and then perform access checks on the file descriptor (which can't be changed by the attacker) rather than the file (which can).

****



6. The following diagram shows four pages tracked using the *aging* algorithm to approximate LRU.

****

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.

(a). Describe one feature of the x86 architecture that make context switching slow.
(b). What hardware feature could be added to reduce the cost of context switching?
Answer: (a). The TLB has to be flushed and reloaded whenever the virtual address space changes. (b). Introduce a tagged TLB.

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

```
int fetchint(uint addr, int *ip) {
  if (addr >= proc->sz || addr + 4 > proc->sz) {
    return -1;
  }
  *ip = *(int*)(addr);
  return 0;
}

int argint(int n, int *ip) {
  return fetchint(proc->tf->esp + 4 + 4*n, ip);
}
```

(a). In the argint function, why do we add 4 to proc->tf->esp?
(b). On the same line, why do we multiply n by 4?
(c). What could go wrong if xv6 didn't have the check at line 5 in fetchint?
Answer: (a). Calling a function pushes the arguments onto the stack, followed by the return address. So the first thing on the stack will be the return address, which is
a 32-bit value (4 bytes), and so arguments begin at %esp+4. (b). Each argument is 32 bits, i.e. 4 bytes. (c). %esp might point outside of the process's memory, potentially allowing the calling process to read kernel memory or memory from another process. Or
%esp might not point to any valid memory at all, in which case we'd get a page
fault in kernel mode, which causes xv6 to panic.

Recall that in xv6 the kernel address space is mapped into every process. For the this
question, you may wish to refer to the xv6 code that sets up a newly created process's
address space, which is on the next page.
(a). What prevents user-level code in a process in xv6 from modifying kernel memory?
(b) What prevents user-level code in a process in xv6 from modifying the memory of another process?
Answer: (a). On line 48 of the code on the next page, xv6 sets the PTE_U bit on the
PTE for every page in the user address space. This tells an x86 CPU that code running in user mode (ring 3) is allowed to access the page. Accessing any page without the PTE_U bit set from user mode will cause a page fault. (b). Each process has its own virtual address space, and the kernel ensures
that each process's virtual addresses map only to pages belonging to the process
itself. It also validates the arguments of every system call to ensure that any data read by the system call is within the process's memory.

(a) Fill out the empty boxes with the age bits each page will have at each clock tick.

(b) If the page replacement algorithm runs after clock tick 4, which page will be chosen for eviction?
Answer: **Page 0. (Compare the age values at the end and pick the lowest age.)**

During a driver's interrupt handling routine, once the interrupt has been acknowledged
it is possible for another interrupt to occur that will invoke the same handler. Drivers
that produce correct results in this scenario are called reentrant. Consider the following
interrupt handler:
(a) Why is this code not reentrant? On what line would an interrupt have to occur to cause the bug to manifest?
(b) How could you fix it so that it is reentrant?
Answer: (a). This code is not reentrant because it uses a global variable; if an interrupt
occured on line 13, 14, or 15, temp would be overwritten. (b). Make temp a local variable rather than a global variable, or wait until
after the swap is done to acknowledge the interrupt.

A computer has four page frames. The time of loading, time of last access, and the R
(read) and M (modified) 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

(a) Which page will NRU replace?
(b) Which page will FIFO replace?
(c) Which page will LRU replace?
(d) Which page will second chance replace?
(e) Why is exact LRU not usually used for memory page replacement in real systems?

Answer: (a). NRU considers pages in the following classes:
• Class 0: not referenced, not modified.
• Class 1: not referenced, modified.
• Class 2: referenced, not modified.
• Class 3: referenced, modified
It picks an arbitrary page from those in the lowest numbered class. In this case, there is exactly one page in class 0, which is page 2.
(b). FIFO simply replaces the page that was brought into memory the longest ago, regardless of whether it has been referenced or modified since then. In this case, that's page 0, which was brought into memory at time 26.
(c). LRU replaces the least recently used page; i.e., the one whose last reference time is the earliest. In this case, that's page 1.
(d) Second chance is a modified version of FIFO. Rather than throwing out the oldest page, it first checks whether the page has been referenced. If so, it clears the reference bit and proceeds to the next-oldest page. So in this case it will examine page 0, clear its reference bit, examine page 3, clear its reference bit, and finally examine page 2, see that its reference bit is 0, and choose it for eviction. So page 2 will be replaced.
(e). The LRU algorithm requires keeping a list of pages, ordered by when they were last referenced, and updating that list every time memory is referenced. Unfortunately, updating the list on every memory reference is expensive, and so doing it on every memory reference would make a system unusably slow.

Recall that in 32-bit x86, page directories and page tables are each made up of 1024 32-
bit entries. Suppose we have 4 processes on a system, each of which has every possible
virtual address mapped.
(a). How much memory is used to store the page directories and page tables
if 4KB pages are used?
(b) If 4MB pages (super pages) are used, then the entries in the page directory point directly to the page frame (i.e., no second-level page tables are used). How much memory would be taken up by page directories in this case?
Answer: (a). For each process, every possible virtual address is mapped, and every page directory entry points to a page table. So we have 1 page directory and 1024 page tables, times 4 processes. Each page table is 4KB, as is the page directory. So the total is: $4 * ((4KB * 1024) + 4KB)) = 4 * 4MB = 16MB$). (The exact answer is 16.015625MB). (b). Since there are no second-level page tables, we just have four page
directories, one for each process. Each one is 4KB, so the total is 4*4KB = 16KB.

****

Main memory divided into a number of equal size frames is the _____ technique.
Answer: simple paging

The chunks of a process are known as _____.
Answer: pages

Available chunks of memory are known as ____.
Answer: frames

The four main structural elements of a computer system are:
Answer: Process, main memory, I/O Modules, and System Bus

The processor itself provides only limited support for multiprogramming, and _____ is needed to manage the sharing of the processor and other resources by multiple applications at the same time.
Answer: software

To overcome the problem of doubling the memory access time, most virtual memory schemes make use of a special high-speed cache for page table entries called a: ***TLB***.

When the system spends most of its time swapping pieces rather than executing instructions it leads to a condition known as:
Answer: Trashing

When instead of using one page table entry per virtual page, the system keeps one entry per physical page frame, it is called:
Answer: Inverted page tables

Executable and Linkable format (ELF) files are the .o files produced by the compiler. The most reliable way to find out they are ELF files is by using the:
Answer: ELF Handler

****