UMass Boston CS 444 Spring 2025
Review for Test2

1. Purpose of the Review.

   The purpose of the review is to run through the topics from Chapters 6-11 in Tanenbaum and Bos that best summarize how different operating systems work.

   We look at each operating system (OS) from the point of view of a programmer who types each line of C code carefully, knowing what it asks the OS to do.

   We practice answering each of the three types of test2 questions.

2. Test2 Procedure and Types of Questions.

   You will be given a page with your test2 on it. You will be given printed pages of any figures from the textbook (Modern Operating Systems, Fifth Edition, by Andrew Tanenbaum and Herbert Bos) referred to in your questions.

   (a) Question 1: (4 mins) a list of 6 T/F questions (4 points each, 24 points total) drawn from designated sections in Chapters 6-11 - write answers on whiteboard

   (b) Question 2: (5 mins) a list of 3 short C code questions (12 points each, 36 points total) - write answers as several line(s) of code or a phrase on the whiteboard

   (c) Question 3: (6 mins) multi-part deadlock question (40 points total) - draw and write the parts of the answer on the whiteboard

   For each question, you will give your answer verbally. Time is limited. I will note the start time for each question. Please give your answers within the designated time for each question.

3. Sections within each Chapter for the Review.

   When there is a "0", it refers to the introduction paragraph(s) that doesn't have a subsection number. When there is an "S", it is the summary.

   PRACTICE QUESTION: What are the titles of Chapters 6-11?

   ANSWER: Ch6 Deadlocks, Ch7 Virtualization and the Cloud, Ch8 Multiple Processor Systems, Ch9 Security, Ch10 Case Study 1: UNIX, Linux and Android, Ch11 Case Study 2: Windows 11.

   These sections and subsections are key to understanding the topics you need to know.

Ch6	6.0	6.1	6.2	6.3	6.4.0	6.4.1	6.6	6.9S			
Ch7	7.0	7.1	7.2	7.3	7.4.0	7.6	7.8	7.9.0	7.11.3	7.13S	
Ch8	8.0	8.1.1	8.1.3	8.2.0	8.2.1	8.2.3	8.3.0	8.3.1	8.5S		
Ch9	9.1.6	9.3.2	9.4	9.5.0	9.5.1	9.7.2	9.8.6	9.10S			
Ch10	10.0	10.1	10.2	10.3.3	10.3.5	10.4	10.6.3	10.7		10.9S	
Andr	10.8.0	10.8.1		10.8.3	10.8.4	10.8.5	10.8.8* to p823	10.8.11* to p845			
Ch11	11.3.1	11.4.1	11.4.3	11.4.4	11.5.3	11.10.3	11.11.0	11.11.4* to p 1027	*11.11.4 frm p1031	11.12S	

4. Ch6 DEADLOCK, pp 437-476. Please see sections 6.0 Intro, 6.1 Resources, 6.2 Introduction to Deadlocks, 6.3 Ostrich Algorithm, 6.4.0 Intro, 6.4.1 Deadlock Detection with One Resource of Each Type, 6.6 Deadlock Prevention, 6.9 Summary and slides ch6.pdf.

   Also, in Ch 2 Processes and Threads, please see section 2.1.5 Process States including Figure 2-2. "A process can be in running, blocked or ready state. The four transitions between these states are shown."

   Also section 2.4 Synchronization and Interprocess Communication including race conditions, critical regions, mutual exclusion with Busy Waiting, especially Figure 2-22. "Mutual exclusion using critical regions.", sleep and wakeup, semaphores, mutexes, message passing and priority inversion.

   Also, in 2.5 Scheduling, batch scheduling: First-Come,First-Served and interactive scheduling: Round-Robin Scheduling and priority scheduling.

   In 2.4, the general intent is to cover how processes work together, either by communicating and sharing information or by staying out of each other's way. A race condition (pp119-120) occurs when two processes read or write shared data and the final result depends on who runs precisely when.

   In 2.4.1, see how the race condition in Fig 2-21 happens.

   In 2.4.2, prevent race conditions by establishing critical regions, p121, so there is mutual exclusion. See 4 conditions for good solution on page 121.

   In 2.4.3, see how mutual exclusion can occur with busy waiting (sit in a tight loop, p127,until entering critical region is allowed).

   In 2.4.4, use interprocess communication to block and be suspended until another process wakes it up.

   In 2.4.5, see sleep and wakeup with a semaphore variable to count wakeups saved for future use, p129-130.

   In 2.4.6, the simple mutex which is locked or unlocked to protect a critical region can be used when the counting of a semaphore is not needed.

   In 2.4.8, message passing can also be used; it avoids shared memory and is often used in parallel processing (p147-148).

   In 2.4.10, priority inversion is explained.

In Ch6, read p437 for the overview of resources that can be used by only one process at a time and what deadlock is. In 6.1, understand the preemptable resource example: memory and the nonpreemptable resource example, a printer. Know the three steps on p439 and the note that says we assume that when a process is denied a resource it is put to sleep. Look at the code in Figure 6-1 a) and see that the lingo is semaphore down to request and semaphore up to release.

In Figure 6-2, a) Deadlock-free code, p441, see how A or B will get resource 1 first, whereas in b) code with potential deadlock, the reversal of order makes a difference.

In 6.2, know the 4 conditions for deadlock. In Fig 6-6, p446, know the difference between the arrow for process holds resource (arrow from resource to process p) and the arrow for process requests resource (arrow from process p to resource).

PRACTICE QUESTION FOR DEADLOCK: How come the arrows make a circle in Fig 6-6, part c?

ANSWER: For any process, the arrow in means resource held and the arrow out means process requested, so for two processes and two resources each process holding one and wanting the other makes the cycle.

In 6.3, note how the device driver can prevent deadlock by returning an error code.

In 6.4.1, note the detection figure, Figure 6-8."a) A resource graph. (b) A cycle extracted from (a)."

In 6.6, know how to prevent deadlock by assuring any one of the four conditions for deadlock cannot occur.

More practice for Question 3 on DEADLOCK:

(a) Use Figure 6.2. "a) Deadlock-free and b) Code with potential Deadlock", p441:

   i. PRACTICE QUESTION FOR DEADLOCK: Using Figure 6.2, p441, review a) the code that cannot get deadlocked. Why not?
   ANSWER: The order of resource requests (1 before 2) for processes A and B means B cannot lock on to what A needs.

   ii. PRACTICE QUESTION FOR DEADLOCK: Using Figure 6.2, b) the code that could get deadlocked, draw two timelines for processes A and B that show no deadlock.
   ANSWER: Use round robin scheduling and have A do both downs and then be swapped out.
   A:_down1,down2_ _use,up2_ _up1,exit_
   B: _request1,block_ _block_ _down1,down2_ _use,up2_ _up1,exit_
   Because B cannot get resource 2, it is put to sleep for its timeslice. Then A uses both resources and releases them. Then B uses both. No deadlock.

   iii. PRACTICE QUESTION FOR DEADLOCK: Which condition for deadlock is not satisfied? Note how conditions 1-3 are satisfied.
   ANSWER: Can you see that the circular wait condition did not happen? The reason is that A got both resources before B got swapped in.

(b) Use Figures 6-3, 6-4 Nonsolution and 6-5 Solution, pp 442-443:

i. PRACTICE QUESTION FOR C CODE: using C code in both Figure 6-4 and Figure 6-5, list the lines of code that assign numbers to philosophers and forks. Also, what is the name of operator "%" and how does it work?
ANSWER: The define sets N, the number of philosopers, to 5. The module void philosopher(int i) is passed a value 0-4. The fork picked up first is i. The next fork is (i+1)
The comment says i is the left fork. So the numbering goes counter-clockwise. (But it could be the right fork and the numbering would go the other way.)

(c) Re C code in Figure 6-4:

PRACTICE QUESTION FOR C CODE: Implement a binary semaphore (add lines of C code to create) that allows one philosopher to eat at a time and never deadlocks. (p444 "One improvement..." asks for this.

ANSWER:

```
After the define statement, add:
  typedef int semaphore;
  semaphore mutex =1;
Then in the while loop after think(); is called, add:
   down(&mutex);
and after the second put_fork statement add:
   up(&mutex);
```

(d) Use Figure 6-5. Solution to dining philosophers problem, p443, and consider doing breaking of conditions that assure deadlock.

1.Regarding 6.6.1, for the dining philosophers, breaking mutual exclusion:

PRACTICE QUESTION: How would you break mutual exclusion here?

ANSWER: It would mean not allowing a philosopher to have exclusive access to two forks. That would mean some kind of sharing agreement!

(e) Use Figure 6-5 again.

Regarding 6.6.2, breaking hold and wait:

PRACTICE QUESTION: How would you break hold and wait here?

ANSWER: It would mean making processes request all needed resources at the start is what we see used in Figure 6-5 to prevent deadlock. The take_forks(i) routine means each philosopher either gets two forks or blocks. To make sure this happens fully, the take_forks(i) occurs in a critical region protected by the mutex.

(f) Use Figure 6-5 again.

Regarding 6.6.3, breaking no preemption:

PRACTICE QUESTION: How would you break the no preemption condition here?

ANSWER: It would mean the system could take back a fork that had been given. Figure 6-5 has no limit on how long a philopher can eat. So there is no system preempting the forks then. Forks are only released after eating as long as you like (maybe across timeslices). We see that with timeslices at least two philosophers will be eating so forks are not idle when they could be used.

The giving back of forks in put_forks(i) is tied to some other philosopher being able to eat. So a philosopher who could not get both forks and did a down on the semaphore[i] to request forks will acquire forks (up on semaphore[i]) when a neighbor philosopher determines that neither of his or her neighbors is eating. But a timelimit on giving back a fork is not the issue for making it so that deadlock could occur and then fixing it by letting the system do something. Since this code does not deadlock, breaking no preemption cannot be done in the philosophers solution in Figure 6-5.

But for the code in Figure 6-4, where deadlock could occur, taking back a fork that had been given to a philosopher, would be ending the deadlock by pre-emption.

(g) Use Figure 6-5 again.

Regarding 6.6.4, breaking circular wait does not need to happen in Figure 6-5 because taking forks and putting back forks is done in pairs in code that are protected by mutex.

A deadlock would have to be possible, and then the OS could intervene and take back a resource that had been given or hold back a process.

5. Ch7 Virtualization and the Cloud, pp 477-526. Please see sections 7.0 Intro, 7.1 History, 7.2 Requirements For Virtualization, 7.3 Type 1 and Type 2 Hypervisors (per Popek and Goldberg), 7.4.0 Techniques for Efficient Virtualization Intro, 7.6 Memory Virtualization, 7.8 Virtual machines on multicore CPUs, 7.9.0 Intro, 7.11.3 Challenges in Bringing Virtualization to the x86, 7.13 Summary and slides ch7.pdf.

   Please also see section 1.7.5 Virtual Machines on p69. Subsections cover VM/370 at IBM in 1979, the Popek and Goldberg paper in 1974, the decades later VMWare workstation, the Java Virtual Machine and containers.

   In 7.0, identify why you might want separate machines, intead of combining functions and using virtualization. Virtualization reduces cost. It also reduces the failures due to bugginess. It makes the cloud work well.

   In 7.1, review the importance of the Popek and Goldberg 1974 paper. See also p507 and 510. They proved how to tell whether an architecture would run on a virtualized machine. The instruction set had to include a trap to the kernel whenever a privileged instruction running in user mode was executed.

   In 7.2, requirements for virtualization are stated: safety, fidelity and efficiency, p484. In 7.3, we see the contrast of type 1 and type 2 hypervisors.

   In 7.4.0, see definition of virtual kernel mode.

   In 7.6, the complexity of getting memory management to work in a virtualized system is illustrated.

   In 7.7, the additional complexity of getting I/O to work in a virtualized system is explained.

   In 7.8, Virtual Machines on Multicore CPUs, p501 (less than a page), Tanenbaum makes the prediction that programmers (yes,us!) will be determining "how many CPUs are needed, whether they should be a multicomputer or a multiprocessor and how minimal kernels of one kind or another fit into the picture.

   Here is the practice:

(a) Use section 7-6 Memory Virtualization, p493-495.

PRACTICE QUESTION T/F: Virtualization greatly complicates memory management.

ANSWER: True

PRACTICE QUESTION T/F: In general, for each virtual machine the hypervisor needs to create a shadow page table that maps the virtual pages used by the virtual machine onto the actual pages the hypervisor gave it.

ANSWER: True

PRACTICE QUESTION T/F: The whole process [Virtual Machine exit] may take tens of thousands of cycles, or more.

ANSWER: True

PRACTICE QUESTION T/F: the cost of handling shadow page tables led chip makers to add hardware support for nested page tables.

ANSWER: True

6. Ch8 Multiple Processor Systems, pp 527-604. Please see 8.0 the introduction and selected sections from 8.1 Multiprocessors, 8.2 Multicomputers and 8.3 Distributed Systems: 8.1.1 Multiprocessor Hardware, 8.1.3 Synchronization, 8.2.1 Multicomputer Hardware, 8.3.1 Network Hardware, 8.3.2 Network services and protocols, 8.3.3 Document based middleware, 8.5 Summary and slides for POSIX Threads.

   Incorporate these figures in thinking about Multiple Processor Systems:

   Fig 8-10, in 8.1.3, p546, The TSL instruction (Ch2, p125-126) can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

   Fig 8-18, in 8.2.1, p561, Position of the network interface boards in a multicomputer.

   Fig 8-19, in 8.2.3, p567, Blocking send call vs. Non-Blocking send call.

   Fig 8-29, in 8.3.1, p584, A portion of the Internet.

   Fig 8-32, in 8.3.3, p589, The Web is a big directed graph of documents.

   In ch8, it is important to review blocking. Look back to pages 31-32 which explain that input/output can be done in 3 ways:

   1.busy waiting (no blocking),

   2.driver requests interrupt and returns/interrupt occurs later, and

   3.Direct Memory Access (DMA).

   For the case of busy waiting, the user program issues a system call, the kernel translates it into a procedure call to the appropriate driver, the driver starts the I/O device and sits in a tight loop polling the device to see if it is done. When done, the driver puts the data where needed and returns to the OS which returns to the user process.

   For the case of interrupt requested/received, the user and kernel action is the same but the driver now requests an interrupt from the device when it finishes, so the driver returns and the OS blocks the caller and does something else. When the controller detects the end of the transfer, it generates an interrupt to signal completion. Then the interrupt handling can occur.

We are not covering DMA.

In Fig 1-11, in 1.3.4, p32 a), the steps in starting an I/O device and requesting an interrupt are illustrated. The CPU contacts the Disk controller which contacts the disk. After the data is ready, the disk controller returns to the interrupt controller, which notifies the CPU (which could be busy) and then when the CPU is ready, transfers the data.

In the introduction to Ch8, three models are compared: 1) shared-memory multiprocessor, 2) message-passing multicomputer and 3) wide area distributed system. In 1) accessing a memory word takes 1-10 nsec. In 2) CPU-memory pairs are connected by a high-speed interconnect, which can pass a message in 10-50 microsec. In 3) complete computer systems over a wide area network communicate by message passing that can take 10-100 millisec, so the delay forces the distributed system to be loosely coupled, (p 528-9).

In 8.1, cache-coherence protocol helps caches that are local and distant operate together. There are different designs for the shared memory.

In 8.1.3, we look at how synchronization works in a multiprocessor with many CPUs. Figure 8-10 shows how a mutex can fail if the bus is not locked. The cache block containing the lock could be constantly in motion between lock owner and lock requestor. Spin and switch solves the problem best, p549. This works for a smallish number of CPUs but cannot scale up to many CPUS.

In 8.2, note the remarkable switch and interface boards with RAM in Figure 8-18. Dedicated RAM on the interface board makes the steady flow rate possible.

In 8.3, Distributed Systems, again no shared memory just like 8.2 Multicomputers, but an enormous multiplicity of CPUs. Figure 8-29 emphasizes connections using high and medium fiber optic cables and copper wire. The local routers provide access. Wireless is only used locally. Fig 8-32 shows the Web of documents. Such power for search term. I looked up number of documents recently and found 15 trillion. Growth is astonishing. This means the demand for servers will grow too.

Here is the practice:

PRACTICE FOR C CODE QUESTION: Use Fig 8-10, p546, that shows 2 CPUs and one shared memory and tells how a Test Set and Lock (TSL) instruction can fail. Consider adding a cache to each CPU in a simple sketch that has bus, CPU1, Memory and CPU2. Use a cache block of 32 bytes. Sketch what caches hold after each step.

Think about drawng the cache block for CPU1 after step 1: CPU 1 reads a 0. Assume it's copy on write.

Draw cache block for CPU1 after step 3: CPU 1 writes a 1. But both CPUs got control of the critical regions in shared memory because both read a 0!

So where in the diagram do you disable the bus?

ANSWER: CPU 1 gets to bring in the data showing 0 but CPU2 must not be able to do that. So CPU 1 locks the bus against CPU 2's access. Your mark would be between 2. and 4. in Fig 8-10. Describe whether how a timeline could make this more precise.

PRACTICE QUESTION T/F: Use the introduction to Ch8, p527, and apply Einstein's special theory of relativity to the question of whether we will have teraherz clocks in computers. Einstein's

theory tells us that a 1-THz (1000-GHz) computer will have to be smaller than 100 microns, just to let the signal get from one end to the other and back once within a single clock cycle. Would a computer this small get built? Is the statement about how small it would have to be true or false?

ANSWER: True

7. Ch9 Security, pp 605-702. Please see sections 9.1.6 Can we build secure systems?, 9.3.2 Cryptography, 9.4 Authentication, 9.5.0 Exploiting Software Intro, 9.5.1 Buffer Overflow Attacks, 9.7.2 Back Doors, 9.8.6 Encapsulating Untrusted Code, 9.10 Summary and slides ch9.pdf.

   Incorporate these:

   Fig 9-22, in 9.5.7, p666, Code that might lead to a command injection attack.

   Fig 9-12, in 9.3.2, p633, Relationship between the plaintext and the ciphertext.

   Fig 9-17, in 9.5.1, p649, a) situation when main program is running. b) after the procedure A (see p648) has been called. c) Buffer overflow shown in gray.

   Fig 9-31, in 9.7.2, p680, a) normal code. b) code with a back door inserted.

   In 9.3.2, we consider cryptography. Fig 9-12 shows the relationship between plaintext and ciphertext. We are not doing the mathematics in test2.

   In 9.4, we study authentication. Note the use of salt, p641.

   In 9.5, vulnerabilities in software are explored. Be able to explain the bug in the code on page 648 that permits a buffer overflow attack. It is worth knowing how something as simple as the missing newline key entry can open a vulnerability.

   In 9.7.2, in Insider Attacks, we study the sad phenomenon that sometimes our own team hurts us. Fig 9-31 shows normal code and code with a backdoor created by a programmer who probably just wanted to save time. But it can be a reason for a break-in that does harm. The book suggests making code reviews a standard procedure to combat this. Other team approaches may also help.

   Here is the practice:

   PRACTICE QUESTION FOR C CODE: In Figure 9-31. a) Normal code. b) Code with a back door inserted., p680, in 9.7.2, what does the call to strcmp do?

   ANSWER: In Linux, the strcmp() function compares two strings s1 and s2. It returns an integer less than 0, equal to 0 or greater than 0. See #include string.h. The first byte that differs is compared. s1 greater than s2 returns a positive number, equal to (no bytes differ) returns zero and s1 less than s2 returns a negative number.

   PRACTICE QUESTION FOR C CODE: The login name typed in is compared to ”zzzzz”. What login name gets you in?

   ANSWER: ”zzzzz”

8. Ch10 Case Study 1: UNIX, Linux and Android, pp 703-870. Please see sections 10.3.2 and 10.3.3 in Processes in Linux, 10.6.1 and 10.6.3 in The Linux File System, 10.7.0 and 10.7.1 in Security in Linux, and 10.8.3, 10.8.4 and 10.8.6 in 10.8 Case Study: Android.

   (a) Fig 10-4, in 10.3.1, p725, Process creation in Linux.

(b) Fig 10-7, in 10.3.2, p728, A highly simplified shell.

(c) Fig 10-50, p818, Basic service manager, AIDL.

(d) Fig 10-8, in 10.3.3, p733, The steps in executing the command ls typed to the shell.

(e) Fig 10-34, in 10.6.3, p780, The relation between file-descriptor table, the open-file-description tale and the i-node table.

(f) Fig 10-39, p802, Android Process Hierarchy - much like the Patrick Brady main slide.

In 10.3.1, processes and the system call fork are introduced (p725). A fork produces an exact copy, called the child. After fork, parent and child are running. Fork returns a zero to the child and a nonzero value, the child's process identifier to the parent. Both check the return code and act accordingly. See Figure 10.4, p725. Message passing between processes is also discussed.

In 10.3.2, process management system calls are introduced further. Figure 10-7 presents a highly simplified shell. The child process does an execve on the given command while the parent waits. The copy command cp is discussed. The function declaration main(argc, argv, envp) passes needed information to execve. There are also system calls for signals. Figure 10-8 illustrates the steps for executing command ls.

In 10.6, the Linux file system is discussed. In Figure 10-34, the file-descriptor table is related to the open-file-descriptor table by arrows. Also the open-file-descriptor table is related to the I-Node table by arrows. The disk blocks used are also indicated. The details of indirect blocks are discussed on p781. Additional file systems are also described.

In 10.7.1, file protection modes, read, write and execute for owner, group and everyone are discussed. System calls for security include checking access and making changes.

In 10.8 ANDROID, the operating system based on the linux kernel and designed to run on mobile devices, is explained fully. A large amount of Android OS is written in java and is object-oriented. The kernel and low-level libraries are written in C and C++. Android OS is open source except in its support of third-party applicatons which are closed-source. Android OS includes Wake Locks to for managing how the system goes to sleep, important for saving energy and conserving battery time. When the screen is on, the system holds a wake lock that prevents the device from going to sleep. When the screen is off, the system waits until no more wake locks are held and then goes to sleep. A hardware interrupt will awaken it and acquire an initial wake lock. See pp804-805.

Here is some practice:

(a) Consider a read system call read(fd, buffer, nbytes). Use Fig 1-17, p52. Write it with a return code in case there is an error in the system call.

PRACTICE QUESTION FOR C CODE: Write the call to read on the board. Add a return code, an integer rc, that returns zero if the call is successful and 1 for an error. Looking at the figure, mention a couple of places where an error could occur. Are read and fread the same? Do they have the same definition of return code. Where would you look that up?

ANSWER: The library could fail to have the code or the system call handler could fail to find the file on disk. Type C man fread or C man read into a browser.

(b) Consider a malloc system call.

PRACTICE QUESTION FOR C CODE: Write a malloc request to obtain space for buffer b. Malloc is a library routine which keeps its own space in a linked list and handles aligning your request according to the size of object and number of object. If Malloc needs more space it issues a system call to the kernel, but otherwise it operates independently. Make a sketch of a user process making a malloc call without going to the kernel. Use the approach in Fig 1-17.

ANSWER:

```
--> Malloc: make allocation   |
|                             |
|   next instruction <--------|
--- call malloc
    put size of obj in reg
    put nobj in reg
    put buffer in reg
```

9. Ch11 Case Study 2: Windows 11, pp 871-1040. Please see sections 11.3.1 Operating System Structure, 11.4.1 Fundamental Concepts, 11.4.3 Implementation of Processes and Threads, 11.4.4 WoW64 and Emulation, 11.5.3 Implementation of Memory Management, 11.10.3 Virtualization-Based Security, 11.11.0 Security Intro, 11.11.4 Security Mitigations, 11.12 Summary.

   Please also see section 1.6.5 The Windows API on p60.

   In 11.4.1, we saw that in Windows, processes are generally containers for programs including virtual address space and resources for threads. In today's systems, there are 64-bit address spaces, dozens of processing cores, terabytes of RAM, SSDs have displaced rotating magnetic hard disks and virtualization is everywhere.

   As explained in 11.4.4, the 64-bit version of Windows XP, released in 2001, included Windows-on-Windows (WoW64), an emulation layer for running unmodified 32-bit applications on 64-bit Windows. This continued the tradition of Windows always retaining the ability to run existing software. The design is a paravirtualization layer, so not totally virtualized.

   PRACTICE QUESTION C CODE: What does the rebootless hotpatch do for the code in Figure 11-59?

   ANSWER: Using the patch binary loaded into memory at runtime so it redirects functions foo() and baz() to updates.

10. Lists of Figures

    (a) Review the list of C code figures (if in your test2, will be supplied):

        i. Fig 6-1, in 6.1.2, p440, Using a semaphore to protect resources: a) One resource. b) Two resources.
        ii. Fig 6-2, in 6.1.3, p441, a) Deadlock-free code. b) Code with a potential deadlock.
        iii. Fig 6-3, in 6.1.3, p442, Lunchtime in the Philosophy Department.

iv. Fig 6-4, in 6.1.3, p442, A nonsolution to the dining philosophers problem.
v. Fig 6-5, in 6.1.3, p443, A solution to the dining philosophers problem.
vi. Fig 9-14, in 9.4.1, p638, a) A successful login. b) Login rejected after name is entered. c) Login rejected after name and password are typed.
vii. Fig 9-noname, in 9.5.1, p648, A logging procedure.
viii. Fig 9-20, in 9.5.2, p659, A format string vulnerability.
ix. Fig 9-22, in 9.5.7, p666, Code that might lead to a command injection attack.
x. Fig 9-31, in 9.7.2, p680, a) Normal code b) Code with a back door.
xi. Fig 9-36, in 9.8.6, p693, a) Memory divided into 16-MB sandboxes. b) One way of checking an instruction for validity.
xii. Fig 10-1, in 10.2.2, p714, The layers in a Linux system.
xiii. Fig 10-3, in 10.2.5, p722, Structure of the Linux kernel.
xiv. Fig 10-4, in 10.3.1, p725, Process creation in Linux.
xv. Fig 10-7, in 10.3.2, p728, A highly simplified shell.
xvi. Fig 11-59, in 11.11.4, p1034, Hotspot application for mylib.dll. Functions foo() and baz() are updated in the patch binary, mylib_patch.dll.

(b) Review basic Chapters 1, 2 and 3 diagrams (if in your test2, will be supplied):

i. Fig 1-1, p2, Where the operating system fits in.
ii. Fig 1-5, p12, A multiprogramming system with 3 jobs in memory.
iii. Fig 1-6, p21, Some of the components of a simple personal computer.
iv. Fig 1-9, p25, A typical memory hierarchy.
v. Fig 1-10, p28, Structure of a Disk Drive.
vi. Fig 1-11, p32, a) Steps in starting an I/O device and getting an interrupt. b) Interrupt processing.
vii. Fig 1-12, p33, The structure of a large x86 system.
viii. Fig 1-13, in 1.5.1, p41, A Process Tree.
ix. Fig 1-14, p43, A file system for a university department.
x. Fig 1-16, p45, Two processes connected by a pipe.
xi. Fig 1-17, p52, The steps in making the system call read(fd, buffer, nbytes).
xii. Fig 1-18, p54, Some of the major POSIX system calls.
xiii. Fig 1-20, p57, Processes have 3 segments: text, data and stack.
xiv. Fig 1-23, p62, The corresponding Win32 API calls.
xv. Fig 1-31, p81, Principal metric prefixes.
xvi. Fig 2-22, p122, Mutual exclusion using critical regions.
xvii. Fig 3-8, p194, The position and function of the Memory Management Unit (MMU).
xviii. Fig 3-9, in 3.3.1, p195, The relation between virtual addresses and physical memory addresses is given by the page table. Every page begins on a multiple of 4096 and ends 4095 addresses higher, so 4K-8K really means 4096-8191 and 8K-12K means 8192-12287.

xix. Fig 3-10, in 3.3.2, p197, The internal operation of the MMU with 16 4-KB pages (Note: 4KB = 2$\hat{1}$2 bytes).

xx. Fig 3-13, in 3.3.4, p204, a) A 32-bit address with 2 page table fields. b) Two-level page tables.

(c) Review additional mechanism diagrams (if in your test2, will be supplied):

i. Fig 6-8, in 6.4.1, p450, a) A resource graph. b) A cycle extracted from a).

ii. Fig 7-1, in 7.3.0, p485, Location of Type 1 and Type 2 Hypervisors.

iii. Fig 7-7, in 7.6, p496, Extended/nested page tables are walked every time a guest physical address is accessed – including the accesses for each level of the guest's page tables.

iv. Fig 8-1, in 8.0, p529, a) A shared memory multiprocessor. b) A message-passing multicomputer. c) A wide area distributed system.

v. Fig 8-10, in 8.1.3, p546, The TSL instruction (Ch2, p125-126) can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

vi. Fig 8-18, in 8.2.1, p561, Position of the network interface boards in a multicomputer.

vii. Fig 8-19, in 8.2.3, p567, a) A blocking send call. b) A nonblocking send call.

viii. Fig 8-29, in 8.3.1, p584, A portion of the Internet.

ix. Fig 9-12, in 9.3.2, p633, Relationship between the plaintext and the ciphertext.

x. Fig 9-17, in 9.5.1, p649, a) situation when main program is running. b) after the procedure A (see p648) has been called. c) Buffer overflow shown in gray.

xi. Fig 9-31, in 9.7.2, p680, a) normal code. b) code with a back door inserted.

xii. Fig 10-8, in 10.3.3, p733, The steps in executing the command ls typed to the shell.

xiii. Fig 10-34, in 10.6.3, p780, The relation between file-descriptor table, the open-file-description tale and the i-node table.

xiv. Fig 10-39, in 10.8.4, p802, Android Process Hierarchy.

xv. Fig 11-33, in 11.5.3, p952, Mapped regions with their shadow pages on disk. The lib.dll file is mapped into two address spaces at the same time.