UMass Boston CS 444 Spring 2026
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 also look at the abstractions that help us understand how the OS manages to work with all the running processes in a fair way.

2. Chapter Sections 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.

   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	7.4.0	7.6	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	9.7.2	9.8.6	9.10S			
Ch10	10.0	10.1.7	10.2.3	10.2.5	10.3.1	10.3.2	10.3.3	10.4.3	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	11.1.3	11.4.1	11.4.4	11.5	11.10.3	11.11.1	11.11.2	11.11.3	11.12S	

   Here are the chapter titles:

   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

3. Test2 Procedures.

   Please show your id and turn off your phone. You will be given a page with your test2 on it.

   You will be given printed pages of any figures from the textbook referred to in your questions.

   You will write your answers on the whiteboard.

   (a) Question 1: (2 mins) Hello! 4 T/F questions (5 points each, 20 points total)
   (b) Question 2: (4 mins) 2 short C code questions (20 points each, 40 points total)
   (c) Question 3: (4 mins) deadlock timeline question (20 points total)
   (d) Question 4: (4 mins) mechanism question (20 points total)

   Time is limited. The instructor will note the start time for each question. The instructor will tell you when to move on to the next question.

4. Ch1 review: Computers, OS layer, command line vs GUI, C code features

   On p1-2, Tanenbaum defines the layer of software called the Operating System (OS). Note: The program users interact with is the shell (text-based) or the GUI (graphical user interface.)

```
// ":" after 'f' means -f requires an argument
    while ((opt = getopt(argc, argv, "vf:")) != -1) {
        switch (opt) {
            case 'v':
                verbose = 1;
                break;
            case 'f':
                filename = optarg; //it's argument
                break; }} //more cases are usual...
```

The computer has two modes of operation: kernel mode (also called supervisor mode) and user mode. The OS runs in kernel mode for at least some of its functionality.

On p3, see source code estimates: over 50 million lines for Windows, over 20 million lines Linux.

In section 1.8.1 The C Language, p74-75, see C code features: C has explicit pointers (see how to dereference in in-page figure) and C does not have garbage collections so you have total control over memory, so use malloc and free.

On p77, in 1.8.3, see compile/execute process.

5. C Code memory addresses, malloc and string functions

   In Ch1, see section 1.3.2 Memory and Figure 1-9. Typical memory hierarchy for access times and capacities (p25).

   See 1.5.2 Address Spaces, p41. The number of bits in the address, 32 or 64, gives address space of 2 raised to that power. Use hex to write an address: 0xFFFF FFFF (4 bits/hex digit) is 32 bits, 0xFFFF FFFF FFFF FFFF is 64 bits. For more than that amount of total space, use Virtual Memory.

   On p80-81, review metric units for memory: review powers of 2 that let you size 1 KB of memory (2 raised to the 10), MB of memory (to the 20), GB of memory (power of 30) and TB of memory (power of 40). How many bytes in a TB of memory? 1,099,511,627,776.

   Use malloc: see man malloc or other: Simple call to get space, explain how to draw in memory:

```
int *ptr = (int*)malloc(10 * sizeof(int));
if (ptr == NULL) {
    // Handle allocation failure
}
```

   Use string functions: strcpy, strdup - they assume you need to pass a pointer to a function so you can get a result back vi a pointer. See strcpy to copy a null terminated string pointed to by src toa buffer pointed to by dest:

```
#include <string.h>
char *strcpy(char *dest, const char *src);
```

6. 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.

   In Ch 2 Processes and Threads, section 2.1.5 Process States, see 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.

   Also review: Mars Pathfinder example, and in 2.4.10, priority inversion is explained.

   In Ch6, read p437 about resources that can be used by only one process at a time.

   In 6.1, understand the preemptable resource example: memory and the nonpreemptable resource example, a printer.

   Look at the code in Figure 6-1 a). Say 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.

   In 6.2, know the 4 conditions for deadlock (p445): Mutual exclusion (only assign a resource to 1 process); hold-and-wait (if some held, still can request); no preemption (resources held cannot be taken away); and circular wait (two or more processes have arrows for hold and arrows for request).

   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).

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

(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 counterclockwise. (But it could be the right fork and the numbering would go the other way.)

7. Ch7 Virtualization and the Cloud, pp 477-526.

In Ch1, read 1.7.5 Virtual Machines, pp 69-73. In 1979, the VM/370 was invented by IBM. It contained a virtual machine monitor (VMM) that ran on the bare hardware and could have other operating systems running on it. Be able to explain Figure 1-29, p71, the types of virtual machines.

Know the importance of the Popek and Goldberg paper from 1974. On page 71, ”the CPU must be virtualizable”. ”When an OS running on a virtual machine in user mode executes a privileged instruction the hardware must trap to the VMM so the instruction can be emulated in software.” If the instruction is just ignored, the VMM won't know about it.

In section 7.6, Memory Virtualization, know what shadow page tables are (p494). Recall that virtual memory, explained in ch3, allows a program to have more memory than there is physical memory. Memory pages are mapped onto page frames by the page table. See Figure 3-9, p195.

So then, in Ch7, when the CPU has been virtualized, memory has to be virtualized too. ”Virtualization greatly complicates memory management.” Suppose a virtual machine is running and the guest OS in it makes a mapping (p493) to physical pages. The hypervisor sets it up. Now a second virtual machine maps its same virtual pages to the same physical pages. The hypervisor makes a shadow page table (p494).

In section 7.11, the VMware virtualization solution for the 32-bit x86 computers is described. It was invented in 1999 despite the x86 not being ”virtualizable.” (p510).

8. Ch8 Multiple Processor Systems, pp 527-604.

   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.

   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.

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.

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

9. 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 a different problem. 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.

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"

(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

10. 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 and 10.8 Case Study: Android. The user interface is discussed in Ch5 section 5.6 for both Linux and Windows.

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

(d) 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 10.8 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.

(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, discussed above.

PRACTICE QUESTION FOR C CODE: Write a malloc request to obtain space for an array element. Then write the print statement to identify the address malloc has returned. Make a diagram of the address space for the process that called malloc. Draw the location of the addresses of the linked list of the array including the address returned as

ANSWER: One rectangle for the process address space. A linked list of address blocks controlled by malloc (but not in the process space and not contiguous). The one element of the linked list whose address was returned to the process.

11. Ch11 Case Study 2: Windows 11, pp 871-1040.

Please see sections 11.4.1 Fundamental Concepts, 11.4.1 Fundamentals, 11.4.4 WoW64 and Emulation, 11.5.3 Implementation of Memory Management, 11.10.1 Virtualization, 11.10.3 Virtualization-Based Security, 11.11.1-3 Security, 11.12 Summary and slides on Windows.

Please also see section 1.6.5 The Windows API on p60. Also, section 5.6.1 on user interfaces comparing linux and windows.

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.

See Figure 11-50 for Hyper-V virtualization, p1004.

The Security sections describe the use of the Security ID (SID). Processes run under the user's SID. Each object can only be accessed by threads with authorized SIDs.

12. Lists of Figures

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

i. intext C code, p75 pointer use.
ii. Fig 2-9, in 2.2.1 Threads, p101. Code for two threads.
iii. Fig 6-1, in 6.1.2, p440, Using a semaphore to protect resources: a) 1 resource b) 2 resources.
iv. Fig 2-23, in 2.4.3, p123. Two processes use variable turn.
v. Fig 6-2, in 6.1.3, p441, a) Deadlock-free code. b) Code with a potential deadlock.
vi. Fig 6-3, in 6.1.3, p442, Lunchtime in the Philosophy Department.
vii. Fig 6-4, in 6.1.3, p442, A nonsolution to the dining philosophers problem.
viii. Fig 6-5, in 6.1.3, p443, A solution to the dining philosophers problem.
ix. 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.
x. Fig 9-noname, in 9.5.1, p648, A logging procedure.
xi. Fig 9-20, in 9.5.2, p659, A format string vulnerability.
xii. Fig 9-31, in 9.7.2, p680, a) Normal code b) Code with a back door.
xiii. Fig 9-36, in 9.8.6, p693, a) Memory divided into 16-MB sandboxes. b) One way of checking an instruction for validity.
xiv. Fig 10-1, in 10.2.2, p714, The layers in a Linux system.
xv. Fig 10-3, in 10.2.5, p722, Structure of the Linux kernel.
xvi. Fig 10-4, in 10.3.1, p725, Process creation in Linux.
xvii. Fig 10-7, in 10.3.2, p728, A highly simplified shell.

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

i. Fig 1-5, p12, A multiprogramming system with 3 jobs in memory.
ii. Fig 1-6, p21, Some of the components of a simple personal computer.
iii. Fig 1-9, p25, A typical memory hierarchy.
iv. Fig 1-11, p32, a) Steps in starting an I/O device and getting an interrupt. b) Interrupt processing.
v. Fig 1-13, in 1.5.1, p41, A Process Tree.
vi. Fig 1-17, p52, The steps in making the system call read(fd, buffer, nbytes).
vii. Fig 1-18, p54, Some of the major POSIX system calls.
viii. Fig 1-20, p57, Processes have 3 segments: text, data and stack.
ix. Fig 1-23, p62, The corresponding Win32 API calls.

x. intext C code, p75 pointer use.

xi. Fig 1-29, p71. Three types of virtual machine monitors.

xii. Fig 1-31, p81, Principal metric prefixes.

xiii. Fig 2-2, p93. A process can be in running, blocked or ready states.

xiv. Fig 2-22, p122, Mutual exclusion using critical regions.

xv. 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.

xvi. 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).

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

xviii. Fig 5-34, p407, X Window in Linux.

xix. Fig 5-36, p411, skeleton of Windows main program.

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

i. 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.

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

iii. 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.

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

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

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

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

viii. 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.

ix. Fig 10-8, in 10.3.3, p733, The steps in executing the command ls typed to the shell.
x. Fig 10-39, in 10.8.4, p802, Android Process Hierarchy.
xi. Fig 11-1, Releases, p872.
xii. Fig 11-3, Win32 API, p875.
xiii. Fig 11-4, Programming Layers, p881.
xiv. Fig 11-11, p895, kernel-mode organization.
xv. Fig 11-22, p932, jobs, processes, threads and fibers.
xvi. Fig 11-50, p1004, Hyper-V virtualization.
xvii. Fig 11-59, p1034, Hotpatch application.