**IT241 Midterm Exam STUDY GUIDE**

1. Definition of Operating System (Section 1.1)

2-A more common definition: the operating system is the one program running at all times on the computer—usually called the **kernel**:

**Along with the kernel**, there are two other types of programs:

**System programs**: associated with the operating system but are not necessarily part of the kernel

**Application programs:** include all programs not associated with the operation of the system.

1. Bluetooth and 802.11 (Section 1.11)

Bluetooth and 802.11 devices use wireless technology to communicateover a distance of several feet, in essence creating a personal-area network (PAN)between a phone and a headset or a smartphone and a desktop computer.

1. Microkernel (Section 2.7)



Researchers at Carnegie Mellon University developed an OS called Mach that modularized the kernel using the **microkemel** approach.

**Microkernels method:**structures the OS by removing all nonessential components from the kernel and implementing them as system and user-level programs. The result is a smaller kernel.

Micro kernels provide minimal process and memory management, and communication facility.

**Main function of the microkemel** is to provide communication throughmessage passing between the client program and the various services running in user space.

**Benefit of the microkernel:**

* Makes extending the operating system easier (new services are added to user space and consequently do not require modification of the kernel.)
* The microkernel also provides more security and reliability, since most services are running as user- rather than kernel- processes.
* Ex:The Mac OS X kernel (also known as **Darwin**) is partly based on the Mach microkernel.

Unfortunately, the performance of microkernels can **suffer** due to increased system-function overhead. EX: Windows NT which corrected later (Windows 95. Windows NT 4.0).

By the time Windows XP was designed, Windows architecture had become more **monolithic** than microkernel.

1. Major categories of system calls (Section 2.4)

System calls can be grouped roughly into six major categories: process control, file manipulation, device manipulation, information maintenance, communications, and protection.

1. Process Control Block (Section 3.1)

Each process is represented in the OS by **a process control block** (**PCB)** - also called a task control block.

A PCB contains many pieces of information associated with a specific process, including these:

**• Process state:** new, ready, running, waiting, halted, and so on.

**• Program counter.** Indicates the address of the next instruction to be executed for this process.

**• CPU registers** vary in number and type, They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an **interrupt** occurs, to allow the process to be continued correctly afterward (Figure 3.4).

**• CPU-scheduling information:** includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

**• Memory-management information:** include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system .

**• Accounting information:** includes the amount of CPU and real time used, time limits, and so on.

**• 110 status information:** includes the list of I/O devices allocated to the process, a list of open files, and so on.



1. List of Process Queues (Section 3.2)
* As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
* The processes in main memory (ready and waiting to execute) are kept on a **linked list** called the **ready queue**.
* **A ready-queue header** contains **pointers** to the first and final PCBs in the list.
* Each PCB includes a pointer field that points to the next PCB in the ready queue.
* The system also includes **other queues**.
1. When a process is allocated the CPU, it executes for a while and eventually quits, is **interrupted**, or waits for the occurrence of a particular event, such as the completion of an I/O request.
2. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process.
* **The list of processes waiting for a particular I/O device is called** a **device queue**. Each device has its own device queue. Figure 3-5..
* A **common representation of process scheduling** is **queuing diagram,** such as that in Figure 3.6

A new process is **initially** put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated the CPU and is executing\_ **one of several events could occur**:

• The process could issue an **I/O request** and then be placed in an I/O queue.

• The process could create a new **child** process and wait for the child's termination.

• The process could be removed forcibly from the CPU, as a result of an **interrupt**, and be put back in the ready queue.

In the first two cases, the process eventually **switches** from the waiting state to the ready state and is then put back in the ready queue. A process **continues** this cycle until it **terminates**, at which time it is removed from all queues and has its PCB and resources **deallocated**.



* Each **rectangular** box represents a **queue**.
* Two types of **queues** are present: the **ready** queue and **a set of device queues**.
* The **circles** represent the **resources** that serve the queues.
* The **arrows** indicate the **flow** of processes in the system.
1. Types of multithreading models (Section 4.3)

Support for threads may be provided either at the user level, for user threads (above the kernel), or by the kernel, for **kernel threads**.

Virtually all contemporary operating systems-including Windows, Linux, Mac OS X, and Solaris support kernel threads.

Three common ways of establishing a relationship between kernel and user thread:

|  |  |  |
| --- | --- | --- |
| 4.3.1 Many-to-One Model | 4.3.2 One-to-One Model | 4.3.3 Many-to-Many Model |
| The many-to-one model maps many user-level threads to one kernel thread. | The one-to-one model maps each user thread to a kernel thread. | The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. |
| Thread management is done by the thread library in user space, so it is efficient.\*Multiple threads are **unable** to run in parallel on multicore systems.Green threads- a thread library available for Solaris used this model.\*very few systems continue to use the model because of its inability to take advantage of multiple processing cores. | Provides **moreconcurrency** than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.\***allows** multiple threads to run in parallel on multiprocessors.\*The only drawback is that creating a user thread requires creating the corresponding kernel thread. | the effect of this design on concurrency:Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for Execution. |
|  |  |  |

One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model**



1. Threads and Thread Library (Section 4.4)

**A thread library** provides the programmer with an API for creating and managing threads.

**Two primary ways of implementing a thread library:**

* The first approach is to provide a library **entirely in user space** with no kernel support: All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call
* The second approach is to implement a kernel-level library supported directly by the operating system: code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.

***Three main thread libraries are in use today:*** POSIX Pthreads (kernel or user), Windows (kernel), and Java.

* **Pthreads**, extension of the **POSIX** standard. Provided as either a user-level or a kernel-level library.
* The **Java thread API**
* Implemented using a thread library available on the host system because in most instances the JVM is running on top of a host operating system.
* Allows threads to be created and managed directly in Java programs.
* ***The Windows thread*** library is **a kernel-level** library available on Windows systems.
* This means that on Windows systems, Java threads are typically implemented using the Windows API; UNIX and Linux systems often use Pthreads.
* **For POSIX and Windows threading**, any data **declared globally**—that is, declared outside of any function—are shared among all threads belonging to the same process.
* Because Java has **no notion of global data**, access to shared data **must be explicitlyarranged** between threads. Data declared local to a function are typically stored on the stack.
* Since each thread has its own stack, each thread has its own copy of local data.

**Two general strategies for creating multiple threads:**

* **Asynchronous**threading: once the parent creates a child thread, the parent resumes its execution, so that the parent and child execute concurrently.
* Each thread runs **independently** of every other thread, and the parent thread need not know when its child terminates.
* **Little** data sharing between threads.
* **Synchronous**threading occurs when the parent thread creates one or more children and then must wait for all of its children to terminate before it resumes-the so-called **fork-join strategy.**
* **Significant** data sharing among threads.
1. Definition of Throughput (Section 5.2)

**Throughput**. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

1. Time Sharing (Section 5.7

Solaris uses priority-based thread scheduling. Each thread belongs to one of six classes:

1. Tune sharing (TS)

2. Interactive (IA)

3. Real time (RT)

4. System (SYS)

5. Fair share (FSS)

6. Fixed priority (FP)

Within each class there are different priorities and different scheduling algorithms.

* The **default scheduling class** for a process is time sharing.
* **The scheduling policy** for the time-sharing class dynamically alters priorities and assigns time slices of different lengths using a multilevel feedback queue.
* By default, there is an inverse relationship between priorities and time slices. The higher the priority, the smaller the time slice; and the lower the priority, the larger the time slice.
1. Priority Inversion (Section 6.6)

A **scheduling challenge** happened when a **higher-priority** process needs to read or modify kernel data that are currently accessed by (one/chain) of **lower-priority** processes.

* Since kernel data are **protected** with a **lock**, the higher-priority process will have to wait for a lower-priority one to finish with the resource.
* **More complicatedSituation** if the lower-priority process is **preempted** in favor of another process with a higher priority.
* **As an example**, assume we have **three** processes-L, M, and H-whose priorities follow the order L < M < H.
* Assume that process H requires resource R, which is currently being accessed by process L. **Ordinarily**, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority- process M- has affected how long process H must wait for L to relinquish resource R.
* **This problem is known aspriority inversion**. It occurs **only** in systems with more than two priorities, so one solution is to have only two priorities (insufficient solution).
* They solve the problem by implementing:

**Priority-inheritance protocol:** all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question.

* When they are finished, their priorities revert to their original values.
* **In the example above,** a priority-inheritance protocol would allow process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. **When** process L had **finished** using resource R, it would **relinquish** its inherited priority from H and assume its original priority. Because resource R would now be available, process H- not M- would run next.
1. Operations for protecting a critical section using mutex locks (Section 6.5)

Instead of hardware-based solutions, operating-systems designers build software tools to solve the critical-section problem **mutex lock (short for mutual exclusion).**

* Used to protect critical regions and thus prevent race conditions.
* A process must acquire (acquire ()) the lock before entering a critical section; it releases (release ()) the lock when it exits the critical section.
* A mutex lock has a **Boolean** variable **available**
* A process that attempts to acquire an unavailable lock **is blocked** until the lock is released.
* 
* The main disadvantage of the implementation given here is that it requires **busy waiting**.
* **Spinlock**: While a process is in its critical section, any other process that tries to enter its critical section must **loop continuously.**
* In fact, this type of mutex lock called a **spinlock** because the process "**spins**" while waiting for the lock to become available.
* **Advantage**: no context switch is required when a process must wait on a lock, and a context switch may take considerable time- employed on multiprocessor systems.
1. Deadlock and necessary conditions for deadlock (Section 7.2)

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

**A deadlock situation** can arise if the following **four conditions** hold simultaneously in a system:

1. **Mutual exclusion**. At least one resource must be held in a **nonsharable** mode; that is, only **oneprocess** at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**. A process must be **holding at least one resource** and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**. Resources **cannot be preempted**; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.
4. **Circular wait.** A set {P0, P1, ... , P,} of **waiting processes must exist** such that Po is waiting for a resource held by P,, P1 is waiting for a resource held by P2, ... , P,\_, is waiting for a resource held by P,, and Pn is waiting for a resource held by Po.
* All four conditions must hold for a deadlock to occur.
1. Logical Address (Section 8.1)

An address generated by the CPU is commonly referred to as a **logical address (virtual address).**

* The set of all logical addresses generated by a program is a **logical address space.**
* (In the range 0 to max).
* The user program generates **only** logical addresses and thinks that the process runs in locations 0 to max. However, these logical addresses must be mapped to physical addresses before they are used.
* The user program **deals with logical addresses**. The memory mapping hardware converts logical addresses into physical addresses.
* The run-time **mapping** from virtual to physical addresses is done by a **hardware** device called the **memory-management unit (MMU):** now called a **relocation register**.
* The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory (see Figure 8.4). For example, if the base is at 14000, then an attempt by the user to address location 0 is dynamically relocated to location 14000; an access to location 346 is mapped to location 14346.
* The user program **never sees the real physical addresses**. The program can create a pointer to location 346, store it in memory, manipulate it, and compare it with other addresses.
* **Only when it is used as a memory address** (in an indirect load or store,
* perhaps) is it relocated relative to the base register.
1. Dynamically linked library (Section 8.1)

**Dynamically linked libraries:** are system libraries that are linked to user programs when the programs are run.

* Some operating systems support only **static linking**: in which system libraries are treated like any other object module and are combined by the loader into the binary program image.
* **Dynamic linking**, in contrast, is similar to dynamic loading. Here, though, linking, rather than loading, is postponed until execution time.
* With dynamic linking, a **stub** is included in the image for each library routine reference.
* The **stub** is a **small piece of code** that indicates how to locate the appropriate memory resident library routine or how to load the library if the routine is not already present.
1. When the stub is executed, it checks to see whether the needed routine is already in memory.
2. If it is not, the program loads the routine into memory. Either way, the stub replaces itself with the address of the routine and executes the routine.
3. Thus, the next time that particular code segment is reached, the library routine is executed directly, incurring no cost for dynamic linking.
4. Under this scheme, all processes that use a language library execute only one copy of the library code.
* This feature can be extended to library updates (such as **bug fixes**). A library may be replaced by a new version, and all programs that reference the library will automatically use the new version.
* **More than one version of a library may be loaded into memory**, and each program uses its version information to decide which copy of the library to use. This system is also known as **shared libraries**.
* Unlike dynamic loading, dynamic linking and shared libraries generally **require** help from the operating system.
1. The vfork() system call in UNIX (Section 9.1)
* Several **versions of UNIX** provide a variation of the vfork() system call:
* Vfork() (for virtual memory fork)-that operates differently from fork () with copy-on-write.
* With **vfork (),the parent process** is **suspended**, and **the child process** uses the address space of the parent.
* Because vfork () **does not use copy-on-write**, if **the child processchanges** any pages of the parent's address space, the altered pages will be visible to the parent once it resumes.
* The vfork () Must be **used with caution** to ensure that the child process does not modify the address space of the parent.
* **Vfork () is intended to be used** when the child process calls exec() immediately after creation.
* Because no copying of pages takes place, vfork() is an extremely **efficient** method of **process creation** and is sometimes used to implement UNIX **command-line shell interfaces.**
1. Examples of executable files (Section 10.1)

The name is split into two parts- a name and an extension, usually separated by a period.

* In this way, the user and the operating system can tell from the name alone what the type of a file is.
* Most operating systems allow users to specify a file name as a sequence of characters followed by a period and terminated by an extension made up of additional characters.
* Examples include resume. docx, server. c, and ReaderThread.cpp.

Only a file with **a.com**, a. **exe**, or a. **shextension can be executed,** for instance.

* The. **com** and **. exe** files are two forms of **binary executable** files, whereas the . sh file is **a shell script** containing, in ASCll format, commands to the operating system.
* **Application programs** also use **extensions** to indicate file types in which they are interested.
* For example, Java compilers expect source files to have a. java extension, and the Microsoft Word word processor expects its files to end with a . doc or . docx extension. These extensions are not always required.
* Consider, too, **the Mac OS X operating system**. In this system, **each** file has a **type**, such as .app (for application). Each file also has a **creator attribute** containing the name of the program that created it.
* This attribute is set by the operating system during the **create ()** call,
* For instance, a file produced by a word processor has the word processor's name as its creator.
* The UNIX system uses a **crude magic number** stored at the beginning of some files to indicate roughly the type of the file-executable program, shell script, PDF file, and so on.
1. File attributes (Section 10.1)

A file's attributes vary from one operating system to another but typically consist of these:

* **Name**. The symbolic file name is the only information kept in human readable form.
* **Identifier**. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
* **Type**. This information is needed for systems that support different types of files.
* Location. This information is a pointer to a device and to the location of the file on that device.
* **Size**. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
* **Protection**. Access-control information determines who can do reading, writing, executing, and so on.
* **Time, date, and user identification.**  This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.
1. File access methods (Section 10.2)

The information in the file can be accessed in several ways. Some systems provide only one access method for files. while others support many access methods, and choosing the right one for a particular application is a major design problem.

**10.2.1 Sequential Access**

The simplest access method and most common.

* **Information in the file is processed in order, one record after the other.**
* **Reads and writes make up the bulk of the operations on a file.**
* **A read operation –read\_next ()-reads the next portion of the file and automatically advances a file pointer, which tracks the I/o location.**
* **The write operation- write\_next () - appends to the end of the file and advances to the end of the newly written material (the new end of file).**
* Such a file can be reset to the beginning, and on some systems, a program may be able to skip forward or backward n records for some integer n perhaps only for n = 1.
* Sequential access, which is depicted in Figure 10.4, is based on a **tape model of a file** and works as well on sequential-access devices as it does on random-access ones.



1. Algorithms for managing a buffer cache File access methods (Section 11.6)
* Some systems maintain a separate section of main memory for a **buffer cache,** where blocks are kept under the assumption that they will be used again shortly.
* Other systems **cache file** data using a **page cache.**
* The page cache uses virtual memory techniques to cache file data as pages rather than as file-system-oriented blocks.
* Caching file data using virtual addresses is far **more efficient than** caching through physical disk blocks, as accesses interface with virtual memory rather than the file system.
* **Several systems**-including Solaris, Linux, and Windows -use page caching to cache both process pages and file data. This is known as **unified virtual memory**.
* Some versions of UNIX and Linux provide a unified buffer cache.
* To illustrate the **benefits of the unified buffer cache**, consider the two alternatives for opening and accessing a file:
1. One approach is to use **memory mapping** (Section 9 .7).
2. the second is to use the standard **system calls** read() and write ().
3. Usefulness of a modify bit File access methods

A modify bit is associated with each page frame. If a frame is modified (i.e. written), the modify bit is then set. The modify bit is useful when a page is selected for replacement. If the bit is not set (the page was not modified), the page does not need to be written to disk. If the modify bit is set, the page needs to be written to disk when selected for replacement.

1. Difference between programmed I/O (PIO) and interrupt driven I/O

To send out a long string of bytes through a memory-mapped serial port, the CPU writes one data byte to the data register to signal that it is ready for the next byte. If the CPU uses polling to watch the control bit, constantly looping to see whether the device is ready, this method of operation is called programmer I/O. If the CPU does not poll the control bit, but instead receives an interrupt when the device is ready for the next byte, the data transfer is said to be interrupt driven.

1. Usefulness of lock bits in I/O requests

A lock bit is associated with every frame. If a frame is locked, it cannot be selected for replacement. To write a block on tape, we lock into memory the pages containing the block. The system then continues as usual with other processes if the I/O request is in a queue for that I/O device. This avoids the replacement of the pages for other processes and the possible unavailability of those pages when the I/O request advances to the head of the device queue. When the I/O is complete, the pages are unlocked.

1. Benefit of using sparse addresses in virtual memory

Virtual address spaces that include holes between the heap and stack are known as sparse address spaces. Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow, or when we wish to dynamically link libraries (or possibly other shared objects) during program execution.

1. Basic file operations when creating/implementing an operating system

The six basic file operations include: creating a file, writing a file, reading a file, repositioning within a file, deleting a file, and truncating a file. These operations comprise the minimal set of required file operations.

1. Internal fragmentation in file systems

Disk space is always allocated in fixed sized blocks. Whenever a file is written to disk, it usually does not fit exactly within an integer number of blocks so that a portion of a block is wasted when storing the file onto the device.

1. Advantages of using file extensions

File extensions allow the user of the computer system to quickly know the type of a file by looking at the file's extension. The operating system can use the extension to determine how to handle a particular file.

1. Terms “raw” and “cooked” used in describing a partition

A raw disk is used where no file system is appropriate. Raw partitions can be used for a UNIX swap space as it does not need a file system. On the other hand, a cooked disk is a disk that contains a file system.

1. Problems associated with linked allocation of disk space routines

The major problem is that a linked allocation can be used effectively only for sequential-access files. Another disadvantage is the space required for the pointers. Yet another problem of linked allocation is the decreased reliability due to lost or damaged pointers.

1. Describe the counting approach to free space management.

The counting approach takes advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously. Thus, rather than keeping a list of n free disk addresses, we can keep the address of the first free block and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count.