File System:

+ Why do we need files? - INTRO
Answer: Files are an abstraction mechanism. They provide a way to store information on the disk and read it back later.

+ Naming conventions - INTRO
Answer: The exact rules for file naming vary somewhat from system to system, but all current operating systems allow strings of one to eight letters as legal file names. Many operating systems support two-part file names, with the two parts separated by a period, as in prog.c. The part following the period is called the file extension and usually indicates something about the file.

+ Structure - IMP
Answer: Files can be structured in any of several ways. Three common possibilities are

(a) Byte sequence: In this model, a file is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes.

(b) Record sequence: In this model, a file is a sequence of fixed-length records, each with some internal structure. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record.

(c) Tree: In this model, a file consists of a tree of records, not necessarily all the same length, each containing a key field in a fixed position in the record. The tree is sorted on the key field, to allow rapid searching for a particular key.

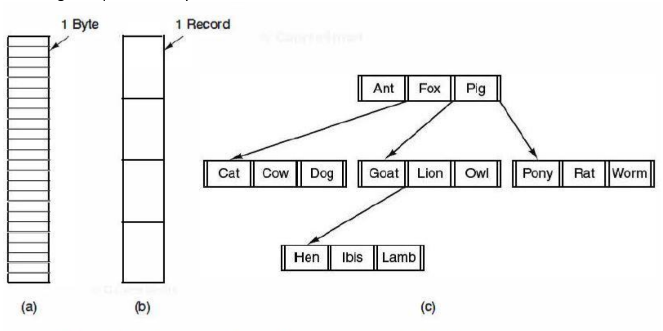


Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

+ Types - IMP
Answer: Types of files are:
a. **Regular files:** These are the ones that contain user information.
b. **Directories:** These are system files for maintaining the structure of the file system.
c. **Character special files:** These are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks.
d. **Block special files:** These are used to model disks.

+ Attributes - IMP
Answer: Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the file's attributes. Some people call them metadata. The list of attributes varies considerably from system to system. Attributes examples are: Protection, Password, Creator, Owner etc.

+ Operations - IMP
Answer: Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.
1. Create. The file is created with no data. The purpose of the call is to announce that the file is corning and to set some of the attributes.
2. Delete. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose.
3. Open. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.
4. Close. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.
5 . Read. Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.
6. Write. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.
7. Append. This call is a restricted form of write. It can only add data to the end of the file. Systems that provide a minimal set of system calls do not generally have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.
8. Seek. For random access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.
9. Get attributes. Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.
10. Set attributes . Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection mode information is an obvious example. Most of the flags also fall in this category.
11. Rename. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a
new file with the new name, and the old file then deleted.

+ Directories - IMP
Answer: To keep track of files, file systems normally have directories or folders, which in many systems are themselves files.
Two types of directory system are:
Single-Level Directory Systems: The simplest form of directory system is having one directory containing all the files. Sometimes it is called the root directory.
Hierarchical Directory Systems: With this approach, there can be as many directories as are needed to group the files in natural ways.
A variant on the idea of linking files is the symbolic link.

+ Implementing Files - layout, types of allocation - INTRO
Answer: Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems.
a. **Contiguous Allocation:** The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks.
b. **Linked List Allocation:** The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 4-1 1 . The first word of each block is used as a pointer to the next one. The rest of the block is for data. Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block).
c. **I-nodes:** Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an i-node (index-node), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 4-13. Given the i-node, it is then possible to find all the blocks of the file.

+ Shared files - INTRO
Answer: When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 4-16 shows the file system of Fig. 4-7 again, only with one of C's files now present in one of B's directories as well. The connection between B's directory and the shared file is called a link. The file system itself is now a Directed Acyclic Graph, or DAG, rather than a tree.

+ Virtual FS - INTRO
Answer: Many different file systems are in use-often on the same computer--even for the same operating system. A Windows system may have a main NTFS file system, but also a legacy FAT-32 or FAT-16 drive or partition that contains old, but still needed, data, and from time to time a CD-ROM or DVD (each with its own unique file system) may be required as well. Windows handles these disparate file systems by identifying each one with a different drive letter, as in C:, D:, etc. When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to. There is no attempt to integrate heterogeneous file systems into a unified whole.

I/O

+ Memory Mapped i/o - IMP
Answer: Each control register is assigned a unique memory address to which no memory is assigned. This system is called memory-mapped I/O. Usually, the assigned addresses are at the top of the address space.

Advantages of Memory Mapped i/o:

- With memory-mapped I/O, a I/O device driver can be written entirely in C. Without memory-mapped I/O, some assembly code is needed.
- With memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O.
- With memory-mapped I/O, every instruction that can reference memory can also reference control registers.

Disadvantages of Memory Mapped i/o:

- Most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous.
- If there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to.

+ DMA - IMP
Answer: Many computers avoid burdening the main CPU with programmed I/O by offloading some of this work to a special purpose processor. This type of processor is called, a Direct Memory Access(DMA) controller. A special control unit is used to transfer block of data directly between an external device and the main memory, without intervention by the processor. This approach is called Direct Memory Access(DMA).

When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Fig. 5-4). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin. The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know or care whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0 , steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete.

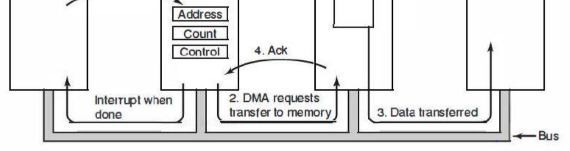


Figure 5-4. Operation of a DMA transfer.

+ Programmed i/o - IMP
Answer: The simplest form of I/O is to have the CPU do all the work. This method is called programmed I/O. The actions followed by the operating system for programed I/O are summarized in Fig. 5-8. First the data are copied to the kernel. Then the operating system enters a tight loop outputting the characters one at a time. The essential aspect of programmed I/O, clearly illustrated in this figure, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called polling or busy waiting.

```
copy_from _user(buffer, p, count);                    I* p is the kernel buffer *I
for (i = 0; i < count; i++) {                                     I* loop on every character *I
            while (*printer status_reg != READY) ;    I* loop until ready *I
            *printer data_ register = p[i];           I* output one character *I
}
return_ to_ user( );
```
Figure 5-8. Writing a string to the printer using programmed I/O.

+ Interrupt driven i/o - IMP
Answer: In Interrupt driven i/o, an input buffer is filled at interrupt time and is emptied by processes that read the device; an output buffer is filled by processes that write to the device and is emptied at interrupt time.
For interrupt driven i/o, let's consider writing a string to the printer using interrupt-driven I/O.

When the system call to print the string is made, the buffer is copied to kernel space, as we showed earlier, and the first character is copied to the printer as soon as it is willing to accept a character. At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed. The work done on the system call is shown in Fig. 5-9(a).

When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt service procedure is run. A crude version of this code is shown in Fig. 5-9(b). If there are no more characters to print, the interrupt handler takes some action to unblock the user. Otherwise, it outputs the next character, acknowledges the interrupt, and returns to the process that was running just before the interrupt, which continues from where it left off.

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer status_ reg != READY) ;
*printer _data_ register = p[O];
scheduler( );
```
Fig. 5-9(a): Code executed at the time the print system call is made.

```
if (count== 0) {
            unblock_ user( );
} else {
            *printer _data_ register = p[i];
            count = count - 1 ;
            i = i + 1 ;
}
acknowledge_interrupt( );
return_from_interrupt( );
```
Fig. 5-9(b): Interrupt service procedure for the printer.

+ i/o using DMA - IMP
Answer: An obvious disadvantage of interrupt-driven I/O is that an interrupt occurs on every character. Interrupts take time, so this scheme wastes a certain amount of CPU time. A solution is to use DMA. Here the idea is to let the DMA controller feed the characters to the printer one at time, without the CPU being bothered. In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU. This strategy requires special hardware (the DMA controller) but frees up the CPU during the I/O to do other work. An outline of the code is given in Fig. 5-10.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```
Fig. 5-10: (a) Code executed when the print

```
acknowledge_interrupt( );
unblock_ user( );
return_from_interrupt( );
```
Fig. 5-10: (b) Interrupt service procedure.

+ i/o layers - INTRO
Answer: I/O software is typically organized in four layers, as shown in Fig. 5-11 . Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The functionality and interfaces differ from system to system, so the discussion that follows, which examines all the layers starting at the bottom, is not specific to one machine.

User-level I/O software
Device-independent operating system software
Device drivers
Interrupt handlers
Hardware

Figure 5-11: Layers of the I/O software system.

+ Interrupt driven handlers - SKIP

+ Device drivers - INTRO
Answer: Each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the device driver, is generally written by the device's manufacturer and delivered along with the device. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.

Deadlocks

+ Resource acquisition - IMP
Answer: For some kinds of resources, such as records in a database system, it is up to the user processes to manage resource usage themselves. One way of allowing user management of resources is to associate a semaphore with each resource. These semaphores are all initialized to 1 . Mutexes can be used equally well.

The three steps listed above are then implemented as a down on the semaphore to acquire the resource, using the resource, and finally an up on the resource to release it. These steps are shown in Fig. 6-l (a).

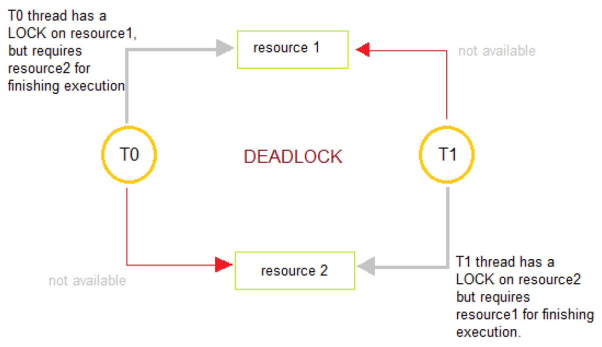
```
typedef int semaphore;
semaphore resource_ 1 ;
void process_A(void) {
            down(&resource_ 1 ) ;
            use_resource_ 1 ( );
            up(&resource_1 );
}
```
Fig: (a) One resource.

```
typedef int semaphore;
semaphore resource_ 1 ;
semaphore resource_2;
void process_A(void) {
            down(&resource_ 1 );
            down (&resource_ 2);
            use_both_resources( );
            up( &resource 2);
            up(&resource_ 1 );
}
```
Fig: (b) Two resources.

Figure 6-1. Using a semaphore to protect resources

For more details, refer the book.

+ Intro - IMP
Answer: Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process. The example of deadlock is given below:


T0 thread has a LOCK on resource1, but requires resource2 for finishing execution
resource 1
not available
T0
DEADLOCK
T1
not available
resource 2
T1 thread has a LOCK on resource2 but requires resource1 for finishing execution.

+ Conditions - IMP
Answer:
1. Mutual exclusion condition: Each resource is either currently assigned to exactly one process or is available.
2. Hold and wait condition: Processes currently holding resources that were granted earlier can request new resources.
3. No preemption condition: Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition: There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.

+ Modelling - resource graphs - IMP
Answer: These four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares.

A directed arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 6-3(a), resource R is currently assigned to process A.
A directed arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 6-3(b), process B is waiting for resource S. In Fig. 6-3(c) we see a deadlock: process C is waiting for resource T, which is currently held by process D. Process D is not about to release resource T because it is waiting for resource U, held by C. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is C-T-D-U-C.

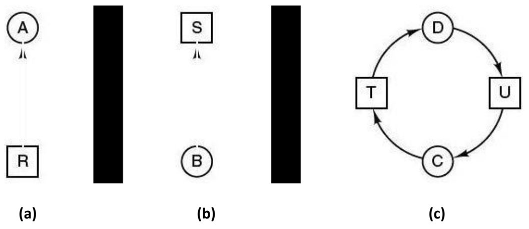
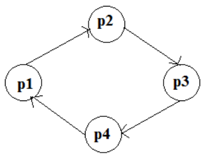


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

+ Detection and Recovery - IMP

Answer: When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to recover after the fact.

If all resource types has only single instance, then we can use a graph called wait-for-graph, which is a variant of resource allocation graph. Here, vertices represent processes and a directed edge from P1 to P2 indicate that P1 is waiting for a resource held by P2. Like in the case of resource allocation graph, a cycle in a wait-for-graph indicate a deadlock. So the system can maintain a wait-for-graph and check for cycles periodically to detect any deadlocks.



The wait-for-graph is not much useful if there are multiple instances for a resource, as a cycle may not imply a deadlock. In such a case, we can use an algorithm similar to Banker's algorithm to detect deadlock.

Once a deadlock is detected, you will have to break the deadlock. It can be done through different ways, including, aborting one or more processes to break the circular wait condition causing the deadlock and preempting resources from one or more processes which are deadlocked.

+ Prevention - IMP

Answer: If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible.

a. **Attacking the Mutual Exclusion Condition:** First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos.

b. **Attacking the Hold and Wait Condition:** If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process would just wait.

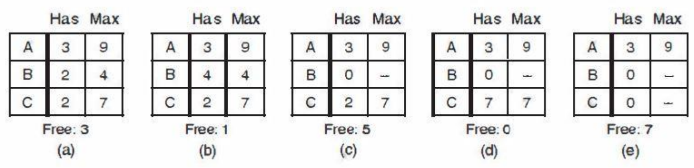
c. **Attacking the No Preemption Condition:** Attacking the third condition (no preemption) is also a possibility If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst. However, some resources can be virtualized to avoid this situation. Spooling printer output to the disk and allowing only the printer daemon access to the real printer eliminates deadlocks involving the printer, although it creates one for disk space. With large disks, however, running out of disk space is unlikely. However, not all resources can be virtualized like this. For example, records in databases or tables inside the operating system must be locked to be used and therein lies the potential for deadlock.

d. **Attacking the Circular Wait Condition:** Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable. Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 6-1 3(a). Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer. With this rule, the resource allocation graph can never have cycles.

+ Safe / unsafe states - IMP

Answer: A state is said to be safe if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately. It is easiest to illustrate this concept by an example using one resource.

In Fig. 6-9(a) we have a state in which A has three instances of the resource but may need as many as nine eventually. B currently has two and may need four altogether, later. Similarly, C also has two but may need an additional five. A total of 1 0 instances of the resource exist, so with seven resources already allocated, there are three still free.



	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	3	9
B	4	4
C	2	7

Free: 1
(b)

	Has	Max
A	3	9
B	0	–
C	2	7

Free: 5
(c)

	Has	Max
A	3	9
B	0	–
C	7	7

Free: 0
(d)

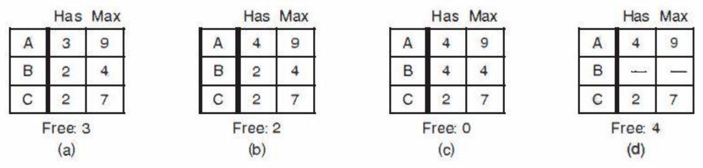
	Has	Max
A	3	9
B	0	–
C	0	–

Free: 7
(e)

Figure 6-9. Demonstration that the state in (a) is safe.

The state of Fig. 6-9(a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler could simply run B exclusively, until it asked for and got two more instances of the resource, leading to the state of Fig. 6-9(b). When B completes, we get the state of Fig. 6-9(c). Then the scheduler can run C, leading eventually to Fig. 6-9(d). When C completes, we get Fig. 6-9(e). Now A can get the six instances of the resource it needs and also complete. Thus the state of Fig. 6-9(a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. 6-lO(a), but this time A requests and gets another resource, giving Fig. 6-lO(b). Can we fmd a sequence that is guaranteed to work? Let us try. The scheduler could run B until it asked for all its resources, as shown in Fig. 6-10(c).



	Has	Max
A	3	9
B	2	4
C	2	7

Free: 3
(a)

	Has	Max
A	4	9
B	2	4
C	2	7

Free: 2
(b)

	Has	Max
A	4	9
B	4	4
C	2	7

Free: 0
(c)

	Has	Max
A	4	9
B	—	—
C	2	7

Free: 4
(d)

Figure 6-10. Demonstration that the state in (b) is not safe.

Eventually, B completes and we get the situation of Fig. 6-lO(d). At this point we are stuck. We only have four instances of the resource free, and each of the active processes needs five. There is no sequence that guarantees completion. Thus the allocation decision that moved the system from Fig. 6- 10(a) to Pig. 6-10(b) went from a safe state to an unsafe state. Running A or C next starting at Fig. 6-1 O(b) does not work either. In retrospect, A's request should not have been granted.

It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig. 6-10(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that A might release a resource before asking for any more, allowing C to complete and avoiding deadlock altogether. Thus the difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be given.

+ Bankers algoritm - INTRO

Answer: A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965); it is known as the banker's algorithm and is an extension of the deadlock detection algorithm. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. What the algorithm does is check to see if granting the request leads to an unsafe state. If it does, the request is denied. If granting the request leads to a safe state, it is carried out.

+ Starvation, livelock, 2-phase locking - INTRO

Answer:

a. **Starvation:** A problem closely related to deadlock and livelock is starvation. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked. Starvation can be avoided by using a first-come, first-served, resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

b. **Livelock:** Imagine a pair of processes using two resources, as shown in Fig. 6-16. Each one needs two resources and they use the polling primitive enter _region to try to acquire the necessary locks. If the attempt fails, the process just tries again. In Fig. 6-16, if process A runs first and acquires resource 1 and then process 2 runs and acquires resource 2, no matter which one runs next, it will make no further progress, but neither process blocks. It just uses up its CPU quantum over and over and over without making progress but also without blocking. Thus we do not have a deadlock (because no process is blocked) but we have something functionally equivalent to deadlock: livelock.

```
void process_A(void) {
        enter region(&resource_ 1 ) ;
        enter region(&resource_2);
        use_both_resources( ) ;
        leave_region(&resource_2);
        leave_region(&resource_ 1 );
}

void process_ B(void) {
        enter _region(&resource_2);
        enter region(&resource_1);
        use_both_resources( ) ;
        leave_region(&resource_ 1 );
        leave_region(&resource_2);
}
```

Figure 6-16. Busy waiting that can lead to livelock.

c. **2-phase locking:** In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase. If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a locked record is encountered during the first phase. In these versions, deadlock can occur.

Multi Processors

+ Intro - IMP

Answer: A shared-memory multiprocessor (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM. A program running on any of the CPUs sees a normal (usually paged) virtual address space. The only unusual property this system has is that the CPU can write some value into a memory word and then read the word back and get a different value (because another CPU has changed it). When organized correctly, this property forms the basis of interprocessor communication: one CPU writes some data into memory and another one reads the data out. For the most part, multiprocessor operating systems are just regular operating systems. They handle system calls, do memory management, provide a file system, and manage 110 devices. Nevertheless, there are some areas in which they have unique features. These include process synchronization, resource management, and scheduling.

+ SKIP details of any individual type of multi-procesor:

+ Remote Procedure Call - IMP

Answer: When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended, and execution of the called procedure takes place on 2. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing or 1/0 at all is visible to the programmer. This technique is known as RPC (Remote Procedure Call) and has become the basis of a large amount of multicomputer software. Traditionally the calling procedure is known as the client and the called procedure is known as the server.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure called the client stub that represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the server stub. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making an RPC are shown in Fig. 8-20. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called marshaling. Step 3 is the kernel sending the message from the client machine to the server machine. Step 4 is the kernel passing the incoming packet to the server stub (which would normally have called receive earlier). Finally, step 5 is the server stub calling the server procedure. The reply traces the same path in the other direction.

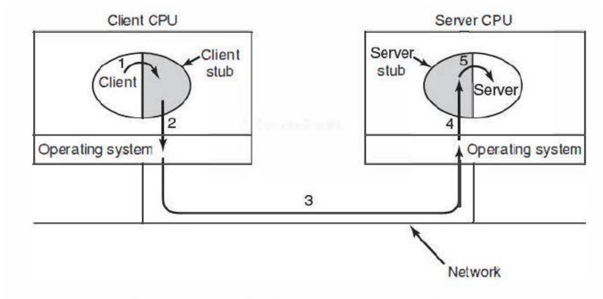


Figure 8-20. Steps in making a remote procedure call. The stubs are shaded gray.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of doing 1/0 using send and receive, remote communication is done by faking a normal procedure call.

+ Virtualization & Paravirtualization - INTRO
Answer:
a. Virtualization: Virtual machine technology, often just called virtualization. This technology allows a single computer to host multiple virtual machines, each potentially running a different operating system. The advantage of this approach is that a failure in one virtual machine does not automatically bring down any others. On a virtualized system, different servers can run on different virtual machines, thus maintaining the partial failure model that a multicomputer has, but at a much lower cost and with easier maintainability.

There are two approaches to virtualization:

1. **type 1 hypervisor:** It is the operating system, since it is the only program running in kernel mode. Its job is to support multiple copies of the actual hardware, called virtual machines, similar to the processes a normal operating system supports.
2. **type 2 hypervisor:** It is just a user program running on, say, Windows or Linux that "interprets" the machine's instruction set, which also creates a virtual machine. We put "interprets" in quotes because usually chunks of code are processed in a certain way and then cached and executed directly to improve performance, but in principle, full interpretation would work, albeit slowly.

The operating system running on top of the hypervisor in both cases is called the **guest operating system**. In the case of a type 2 hypervisor, the operating system running on the hardware is called the **host operating system**.

b. Paravirtualization: In computing, paravirtualization is a virtualization technique that presents a software interface to virtual machines that is similar, but not identical to that of the underlying hardware.

The intent of the modified interface is to reduce the portion of the guest's execution time spent performing operations which are substantially more difficult to run in a virtual environment compared to a non-virtualized environment. The paravirtualization provides specially defined 'hooks' to allow the guest(s) and host to request and acknowledge these tasks, which would otherwise be executed in the virtual domain (where execution performance is worse). A successful paravirtualized platform may allow the virtual machine monitor (VMM) to be simpler (by relocating execution of critical tasks from the virtual domain to the host domain), and/or reduce the overall performance degradation of machine-execution inside the virtual-guest.

Paravirtualization requires the guest operating system to be explicitly ported for the para-API — a conventional OS distribution that is not paravirtualization-aware cannot be run on top of a paravirtualizing VMM. However, even in cases where the operating system cannot be modified, components may be available that enable many of the significant performance advantages of paravirtualization.

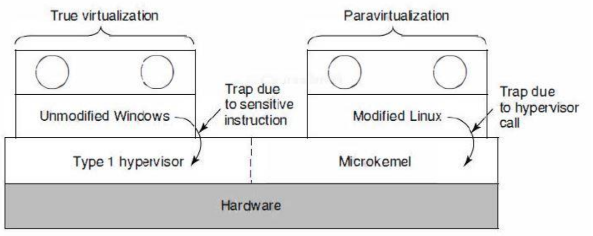


Figure 8-27. A hypervisor supporting both true virtualization and paravirtualization.

Distributed Systems

+ INTRO to various implementation types
Answer: These systems are similar to multicomputers in that each node has its own private memory, with no shared physical memory in the system. However, distributed systems are even more loosely coupled than multicomputers.
To start with, the nodes of a multicomputer generally have a CPU, RAM, a network interface, and perhaps a hard disk for paging. In contrast, each node in a distributed system is a complete computer, with a full complement of peripherals. Next, the nodes of a multicomputer are normally in a single room, so they can communicate by a dedicated high-speed network, whereas the nodes of a distributed system may be spread around the world. Finally, all the nodes of a multicomputer run the same operating system, share a single file system, and are under a common administration, whereas the nodes of a distributed system may each run a different operating system, each of which has its own file system, and be under a different administration.
A typical example of a multicomputer is 5 1 2 nodes in a single room at a company or university working on, say, pharmaceutical modeling, whereas a typical distributed system consists of thousands of machines loosely cooperating over the Internet.

+ Ethernet, Internet, Protocol - SKIP

+ Middleware - INTRO
Answer: Middleware is a software layer situated between applications and operating systems. Middleware is typically used in distributed systems where it simplifies software development by doing the following:

- Hides the intricacies of distributed applications
- Hides the heterogeneity of hardware, operating systems and protocols
- Provides uniform and high-level interfaces used to make interoperable, reusable and portable applications
- Provides a set of common services that minimizes duplication of efforts and enhances collaboration between applications

Types of middleware are:

a. Document-based middleware
b. File-system-based middleware
c. Object-based middleware
d. Coordination-based middleware

+ Publish/Subscribe - IMP
Answer: Our next example of a coordination-based model was inspired by Linda and is called publish/subscribe (Oki et al., 1993). It consists of a number of processes connected by a broadcast network. Each process can be a producer of information, a consumer of information, or both.
When an information producer has a new piece of information (e.g., a new stock price), it broadcasts the information as a tuple on the network. This action is called publishing. Each tuple contains a hierarchical subject line containing multiple fields separated by periods. Processes that are interested in certain information can subscribe to certain subjects, including the use of wildcards in the subject line. Subscription is done by telling a tuple daemon process on the same machine that monitors published tuples what subjects to look for.
Publish/subscribe is implemented as illustrated in Fig. 8-41. When a process has a tuple to publish, it broadcasts it out onto the local LAN. The tuple daemon on each machine copies all broadcasted tuples into its RAM. It then inspects the subject line to see which processes are interested in it, forwarding a copy to each one that is. Tuples can also be broadcast over a wide area network or the Internet by having one machine on each LAN act as an information router, collecting all published tuples and then forwarding them to other LANs for rebroadcasting. This forwarding can also be done intelligently, only forwarding a tuple to a remote LAN if that remote LAN has at least one subscriber who wants the tuple. Doing this requires having the information routers exchange information about subscribers.

# Difference between multicomputer and multiprocessor:

Multicomputer:	**Multiprocessor:**
1.A computer made up of several computers. similar to parallel computing. 2. Distributed computing deals with hardware and software systems containing more than one processing element, multiple programs, running under a loosely or tightly controlled regime. 3. multicomputer have one physical address space per CPU. 4. It can run faster. 5. A multi-computer is multiple computers, each of which can have multiple processors. Used for true parallel processing.	1.A multiprocessor system is simply a computer that has more than one CPU on its motherboard. 2. Multiprocessing is the use of two or more central processing units (CPUs) within a single computer system. 3. Multiprocessors have a single physical address space (memory) shared by all the CPUs. 4. A multiprocessor would run slower, because it would be in ONE computer. 5. A multi-processor is a single system with multiple CPU's.

# Difference between symbolic link and hard link:

symbolic link	**hard link**
1. A symbolic link is a link to another name in the file system. 2. A symbolic link points to another file *by name*.	1. Hard links are only valid within the same File system. 2.A hard link points to the file by inode number.

# Suppose a UNIX file system has some constraints--say, 2 KB blocks and 8B disk addresses. What is the maximum file size if inodes contain 13 direct entries, and one single, double, and triple indirect entry each?
Answer: Total maximum file size: 2KB*(10 + 2**8 + 2**16 + 2**24) = around 32 GB
Alternate method:

each block is 2kb = 2^11
1 disk address is 8b = 2^3
So, in 1 block there are 2^11/2^3 = 2^8 pointers"

for 13 direct entries = (2^8)*13 = 3328
for single = (2^8)^2 = 2^16
for double = (2^8)^3 = 2^24
for triple = (2^8)^4 = 2^32
total pointer is :3328 + 2^16 + 2^24 + 2^32"

size of the disk is : total of pointer*size of the pointer , which is around 34 GB "