CBCS Scheme

USN | | | | | | | | | | |

16MCA24

Second Semester MCA Degree Examination, June/July 2017

Operating Systems

Time: 3 hrs. Max. Marks: 80

Note: Answer FIVE full questions, choosing one full question from each module.

Module-1

1 a. What is a processor register? What functions does it serve? **(10 Marks)**

b. What is an interrupt? Explain with an instruction cycle diagram. **(06 Marks)**

OR

2 a. Explain various services provided by the operating system. **(08 Marks)**

b. How does system call work? Explain with neat diagram. Explain the types of system call. **(08 Marks)**

Module-2

3 a. Explain the five state process model with transition diagram, showing how to change process model for a suspend process. **(10 Marks)**

b. Explain with diagram the user-level threads and kernel-level threads. **(06 Marks)**

OR

4 a. Calculate the average waiting time, turnaround time for (i) SJF and (ii) priority scheduling with following set of processes. P_1, P_2, P_3, P_4, P_5 have all arrived at same time.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

(08 Marks)

b. What is reader-writers problem? Explain solution with semaphores. **(08 Marks)**

Module-3

5 a. What is a deadlock? What are the necessary conditions for a deadlock to occur? **(06 Marks)**

b. Design a solution to the dining philosophers problem using monitors/semaphores. **(10 Marks)**

OR

6 a. Mention the different page table structures. Explain in brief. **(10 Marks)**

b. Consider following page reference string:
7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1
How many page faults would occur in the case (i) LRU, (ii) FIFO? **(06 Marks)**

Module-4

7 a. Explain various file operations. **(10 Marks)**

b. Explain the virtual file system with a schematic diagram. **(06 Marks)**

OR

8 a. Explain the following with respect to free space management:
   i) Linked list
   ii) Grouping **(08 Marks)**

   b. List and explain different file allocation methods. **(08 Marks)**

Module-5

9 a. Explain various process scheduling algorithms in Linux operating system. **(10 Marks)**

   b. Write file system types in Linux. **(06 Marks)**

OR

10 Write short notes on:
   a. Components of Linux system
   b. Shared memory
   c. Inter-process communication
   d. Journalling **(16 Marks)**

* * * * *

Module - 1

1. a)

A processor includes a set of registers that provide memory. But, this memory is faster and smaller than the main memory. These registers can be segregated into two types based on their functionalities as discussed in the following sections.

User – visible Registers

These registers enable the assembly language programmer to minimize the main memory references by optimizing register use. Higher level languages have an optimizing compiler which will make a

choice between registers and main memory to store variables. Some languages like C allow the programmers to decide which variable has to be stored in register.

A user visible register is generally available to all programs. Types of registers that are available are: data, address and condition code registers.

Data Registers: They can be assigned to different types of functions by the programmer. Sometimes, these are general purpose and can be used with any machine instruction that performs operations on data. Still, there are some restrictions like – few registers are used for floating-point operations and few are only for integers.

Address Registers: These registers contain main memory addresses of data and instructions. They may be of general purpose or may be used for a particular way of addressing memory. Few examples are as given below:

o **Index Registers:** Indexed addressing is a common mode of addressing which involves adding and index to a base value to get the effective address.

o **Segment Pointer:** In segmented addressing, a memory is divided into segments (a variable-length block of words). In this mode of addressing, a register is used to hold the base address of the segment.

o **Stack Pointer:** If there is a user-visible stack addressing, then there is a register pointing to the top of the stack. This allows push and pop operations on instructions stored in the stack.

Control and Status Registers

The registers used by the processor to control its operation are called as Control and Status registers. This registers are also used for controlling the execution of programs. Most of such registers are not visible to the user. Along with MAR, MBR, I/OAR and I/OBR discussed earlier, following registers are also needed for an instruction to execute:

Program Counter: that contains the address of next instruction to be fetched.

Instruction Register(IR): contains the instruction most recently fetched.

All processor designs also include a register or set of registers, known as ***program status word (PSW)***. It contains condition codes and status information like interrupt enable/disable bit and kernel/user mode bit.

Condition codes (also known as ***flags***) are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may produce a positive, negative, zero or overflow result. Condition code bits are collected into one or more registers. And, they are the part of a control register. These bits only can be read to know the feedback of the instruction execution, but they can't be altered

1. b)

The normal sequence of the processor may be interrupted by other modules like I/O,

memory etc. Table 1.1 gives the list of common interrupts.

Table 1.1 Classes of Interrupts

Table 1.1 Classes of Interrupts

Class	Description
Program	Generated as a result of an instruction execution. For example, arithmetic overflow, division by zero etc.
Timer	Generated by timer within the processor. It allows OS to perform certain functions on regular basis.
I/O	Generated by I/O controller to indicate any error or normal completion of an operation.
Hardware Failure	Generated by failure like power failure or memory parity error.

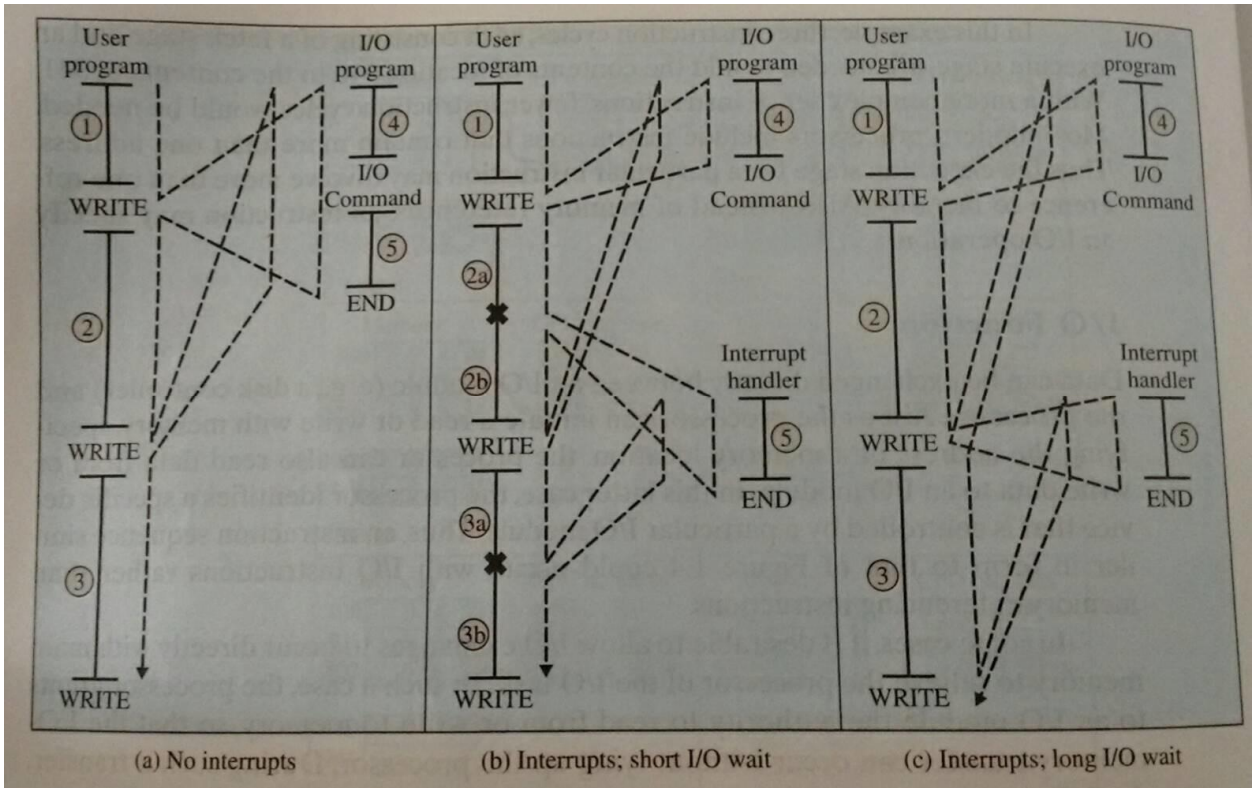


Figure1.5 Program flow of control with and without interrupts

The aim of interrupts is to improve the processor utilization. For example, most I/O sevices are slower than the processor. If the processor gives the instruction for WRITE something on I/O devices, the I/O unit takes two steps for the job –

- I/O program may copy the data to be written into the buffer etc. and prepare for the actual I/O operation.
- The actual I/O command has to be executed.

Without interrupts, the processor would sit idle while the I/O unit is preparing (the first step) itself for the job. But, in case of interrupts, the processor just gives intimation to the I/O unit first. While I/O unit prepares itself, the processor would continue to execute the next

instructions in the program. When I/O unit is ready, in between, it will do the actual I/O command and come back to normal flow of execution. Refer Figure 1.5 for understanding this concept.

Interrupts and the Instruction Cycle

Whenever interrupts are introduced in the system, the processor gives information to the I/O unit and without waiting for I/O operation to complete; it will continue to execute next instruction . When the external device is ready to accept more data from the processor, the I/O module sends an ***interrupt request*** signal to the processor. Now, the processor suspends the current operation of the program and responds to a routine (or a function) of I/O device, which is called as ***interrupt handler***. When the interrupt processing is completed, the processor resumes the execution. To allow interrupts, an ***interrupt stage*** is added along with fetch stage and execute stage as shown in Figure 1.6.

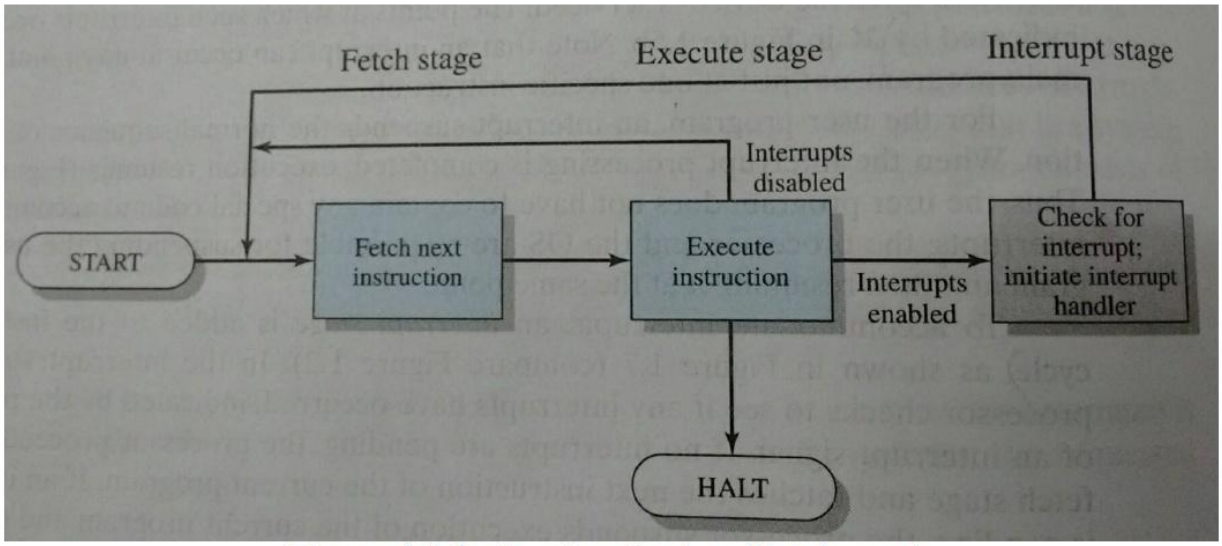


Figure 1.6 Instruction Cycle with interrupts

In the interrupt stage, the processor checks for any possible interrupt signal. If no interrupt is pending, it will go the fetch stage. If an interrupt signal is there, the processor suspends current execution and executes interrupt handler routine. The interrupt handler routine is a part of OS, which identifies nature of interrupt and performs necessary action. After completing interrupt handler routine, the processor resumes the program execution from the point where it was suspended.

It is understood that some overhead is involved in this process. Extra instructions have to be executed in the interrupt handler to determine type of interrupt, to decide the appropriate action etc. But, instead of processor sitting idle for I/O operation and wasting huge amount of time, the concepts of interrupts are found to be efficient.

2.a) OPERATING SYSTEM SERVICES

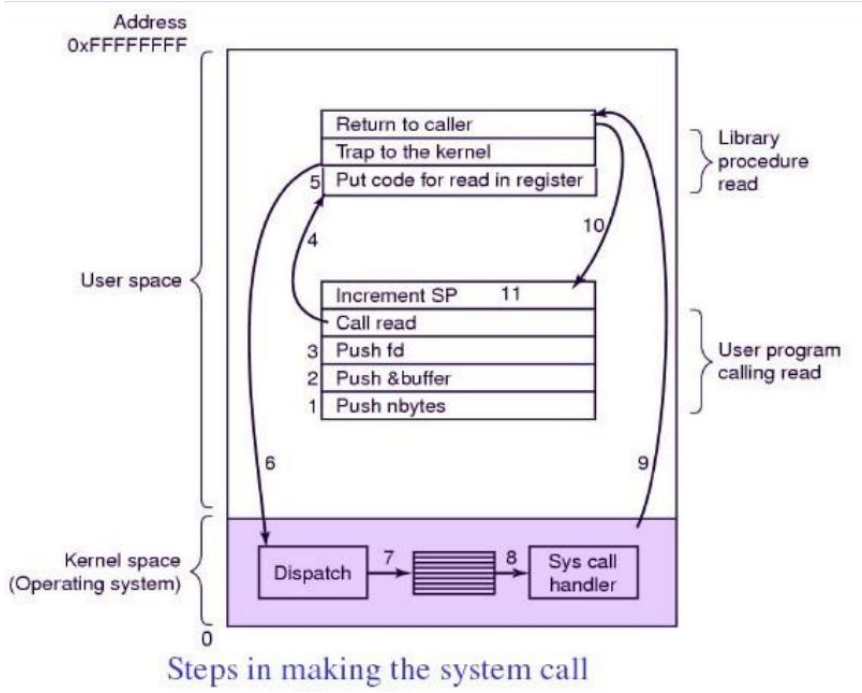
An OS provides an environment for the execution of programs. Also, it provides certain services to the programs and its users. Though these services differ from one OS to the other, following are some general services provided by any OS. Program execution: The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

- I/O operations: A running program may require I/O. This I/O may involve a file or an I/O device. For efficiency and protection, users usually cannot control I/O devices directly. Therefore, the OS must provide a means to do I/O.

 File-system manipulation: The OS must facilitate the programs to read and write the files. And also, programs must be allowed to create and delete files by name.
 Communications: Processes may need to exchange information with each other. These processes may be running on same computer or on different computers.
Communications may be implemented via shared memory, or by the technique of message passing, in which packets of information are moved between processes by the OS.
 Error detection: The OS constantly needs to be aware of possible errors. Errors may occur in any of CPU, memory hardware, I/O devices, user program etc. For each type of error, the OS should take the appropriate action to ensure correct and consistent computing. OS also has another set of functionalities to help the proper functioning of itself.
 Resource allocation: When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. OS has to manage many resources like CPU cycles, main memory, and file storage etc. OS uses CPU scheduling routines for effective usage of CPU. These routines manage speed of the CPU, the jobs that must be executed, the number of registers available, and such other factors.
 Accounting: We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics. Usage statistics may be a valuable tool for researchers who wish to reconfigure the system to improve computing services.
 Protection: Information on a multi-user computer system must be secured. When multiple processes are executing at a time, one process should not interfere with the others. Protection involves ensuring that all access to system resources is controlled. System must be protected from outsiders as well. This may be achieved by authenticating the users by means of a password. It also involves defending external I/O devices, modems, network adapters etc. from invalid access.

2.b)

A special signal or message generated and send by the user program or system itself to invoke a module of operating system so that it can work with it and access the control of the hardware. System call changes the mode of the application from user mode to kernel mode. Now when the mode is changed the application can complete its request to access the control of the hardware.



Steps in making the system call

Let us consider a *read* system call. The system call constitutes of three parameters, file name, pointer to the buffer and number of bytes to read. The function call looks like, count= read(fd, buffer, nbytes). The count consist of total number of bytes to read and if some error occur it is set to -1 which is also indicated by a global variable *errno*. The program should check this error variable timely to check if any error has occurred. The 1st and the 3rd parameters are passed by value and the 2nd parameter is passed by reference i.e. the buffer address is passed. Now when the read procedure is called by the user program the control is transferred to the read procedure. At this stage the parameters are pushed onto the stack (step 1, 2, 3) and the control is now completely transferred to the read procedure. Then the system call number is put into the register(step 5) and a TRAP instruction is executed that switches from user mode to kernel mode(step 6). Now the system call number is examined and then the dispatcher dispatches to the correct system call handler via table of pointers to the system call handler(step 7). After that the system call handler works(step 8). Then the control is transferred to the user program from the read procedure (step 10).
Finally, SP is incremented to clean up the stack. In this way the job of read system call is completed

Types of system call and their examples:

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

Module - 2

3.a) The various States of the Process are as Followings:-

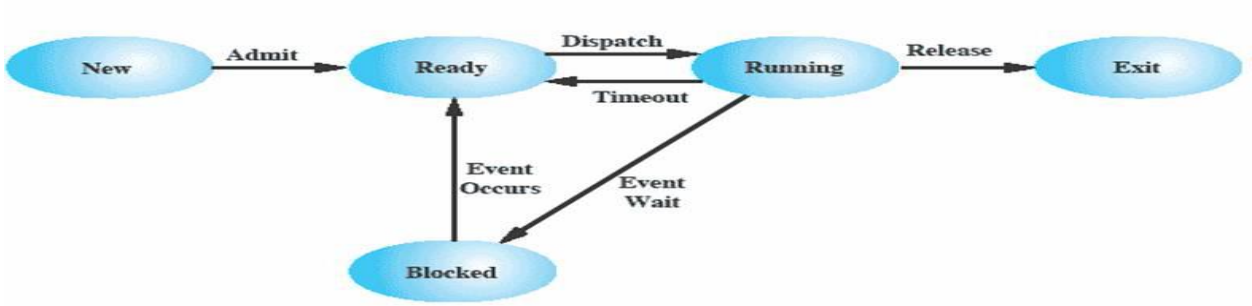
1) **New State:** When a user request for a Service from the System , then the System will first initialize the process or the System will call it an initial Process . So Every new Operation which is Requested to the System is known as the New Born Process.

2) **Running State:** When the Process is Running under the CPU, or When the Program is Executed by the CPU , then this is called as the Running process and when a process is Running then this will also provides us Some Outputs on the Screen.

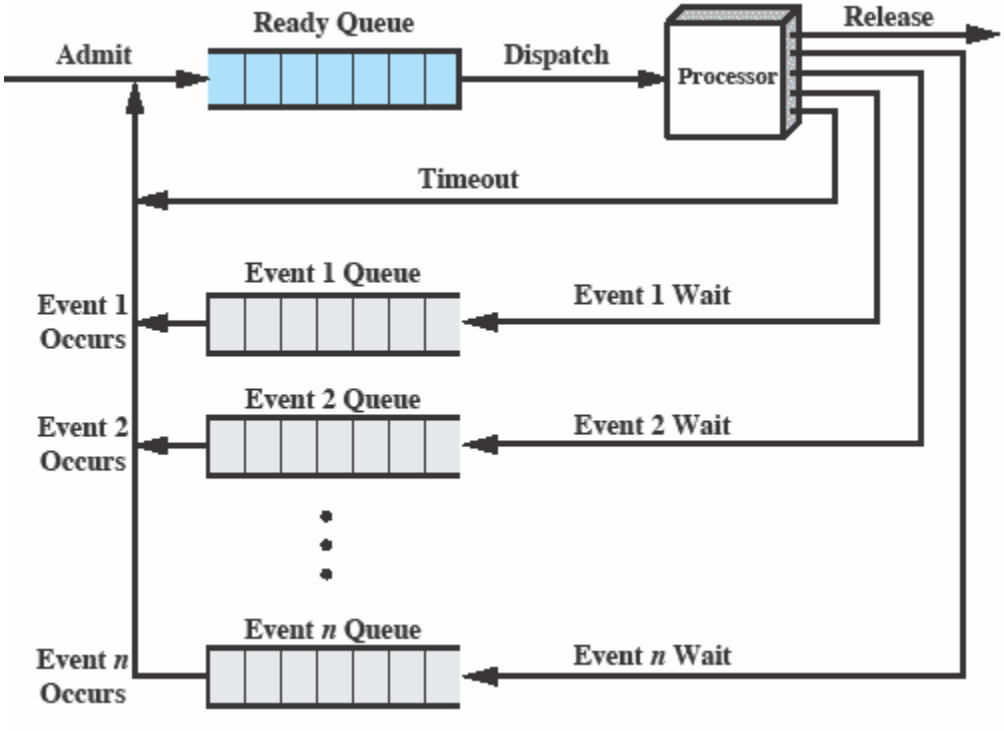
3) **Waiting:** When a Process is Waiting for Some Input and Output Operations then this is called as the Waiting State. And in this process is not under the Execution instead the Process is Stored out of Memory and when the user will provide the input then this will Again be on ready State.

4) **Ready State:** When the Process is Ready to Execute but he is waiting for the CPU to Execute then this is called as the Ready State. After the Completion of the Input and outputs the Process will be on Ready State means the Process will Wait for the Processor to Execute.

Terminated State: After the Completion of the Process , the Process will be Automatically terminated by the CPU . So this is also called as the Terminated State of the Process. After executing the whole Process the Processor will also de-allocate the Memory which is allocated to the Process. So this is called as the Terminated Process.



Five-state Process Model



Multiple Blocked queues

3.b)

User – level and Kernel – level Threads

A thread can be implemented as either a user – level thread (ULT) or kernel – level thread (KLT). The KLT is also known as *kernel – supported threads* or *lightweight processes.*

User – level Threads: In ULT, all work of thread management is done by the application and the kernel is not aware of the existence of threads. It is shown in

Figure 3.12 (a).

Any application can be programmed to be multithreaded by using a threads library, which a package of routines for ULT management. Usually, an application begins with a single thread and begins running in that thread. This application and its thread are allocated to a single

process managed by the kernel. The application may spawn a new thread within the same process during its execution. But, kernel is not aware of this activity.

The advantages of ULT compared to KLT are given below:
 Thread switching doesn't require kernel mode privileges. Hence, the overhead of two switches (user to kernel and kernel back to user) is saved.
 Scheduling can be application specific. So, OS scheduling need not be disturbed.
 ULTs can run on any OS. So, no change in kernel design is required to support ULTs.

There are certain disadvantages of ULTs compared to KLTs:
 Usually, in OS many system calls are blocking. So, when a ULT executes a system call, all the threads within the process are blocked.
 In a pure ULT, a multithreaded application cannot take advantage of multiprocessing.

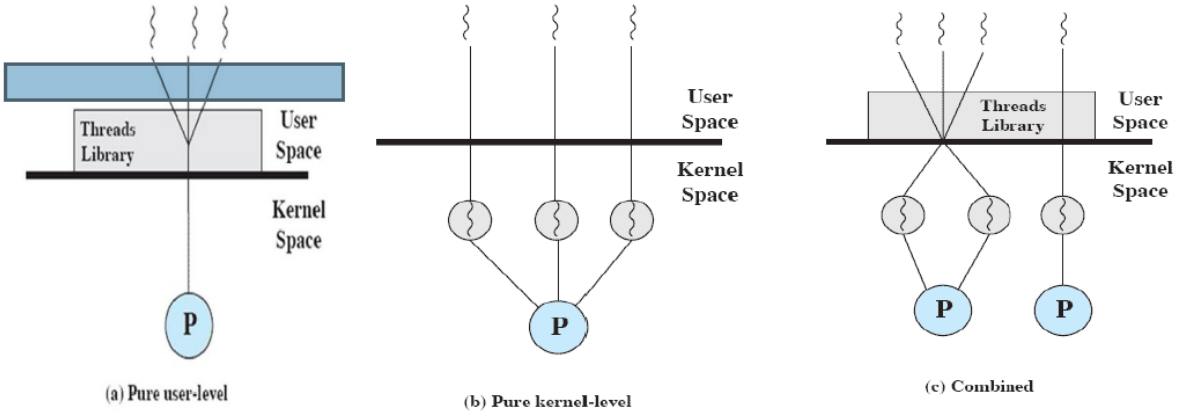


Figure 3.12 ULT and KLT

Kernel – level Threads: In pure KLT model, all work of thread management is done by the kernel. Thread management code will not be in the application level. This model is shown in Figure 3.12(b). The kernel maintains context information for the process as a whole and for individual threads within the process. So, there are certain advantages of KLT :

 The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
 If one thread in a process is blocked, the kernel can schedule another thread of the same process.
 Kernel routines themselves can be multithreaded.
But, there is a disadvantage as well: The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Combined Approach: Some OS provide a combination of ULT and KLT as shown in Figure 3.12 (c). In this model, thread creation is done completely in user space. The multiple ULTs from a single application are mapped onto number of KLTs. The programmer may adjust the number of KLTs for a particular application and processor to achieve the best results.

4.a)

a. Calculate the average waiting time, turnaround time for (i) SJF and (ii) priority scheduling with following set of processes. P_1, P_2, P_3, P_4, P_5 have all arrived at same time.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

SJF (Non preemptive)

P2	P4	P3	P5	P1

0 1 2 4 9 19

AWT = 9+0+2+1+4 = 16/5 = 3.2 ms

Turnaround Time : = 19+1 +4 +2 + 9 = 35/5 = 7 ms

Priority (Non-preemptive)

P2	P5	P1	P3	P4

0 1 6 16 18 19

AWT = 6+0+16+18+1 = 41/5 = 8.2 ms

Turnaround Time = 16 + 1 +18 +19 +6 = 60/5 = 12 ms

4.b) Reader-writers prolem

Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write

Reader's have priority Unless a writer has permission to access the object, any reader requesting access to the object will get it. Note this may result in a writer waiting indefinitely to access the object.

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true) {
      semWait (x);
      readcount++;
      if (readcount == 1) semWait (wsem);
      semSignal (x);
      READUNIT();
      semWait (x);
      readcount--;
      if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
 }
void writer()
{
    while (true) {
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
}
```

Writers Have Priority

When a writer wishes to access the object, only readers which have already obtained permission to access the object are allowed to complete their access; any readers that request access after the writer has done so must wait until the writer is done. Note this may result in readers waiting indefinitely to access the object The following semaphores and variables are added:

– A semaphore ***rsem*** that inhibits all readers while there is at least one writer desiring access to the data area

– A variable ***writecount*** that controls the setting of ***rsem***

– A semaphore ***y*** that controls the updating of ***writecount***

– A semaphore ***z*** that prevents a long queue of readers to build up on ***rsem***

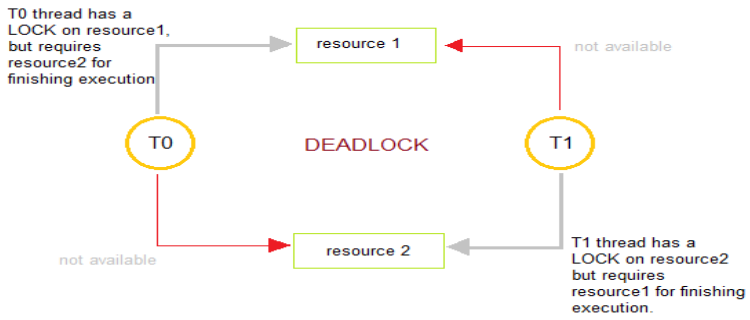
```
/* program readersandwriters */
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true) {
      semWait (z);
           semWait (rsem);
                semWait (x);
                     readcount++;
                     if (readcount == 1) semWait (wsem);
                semSignal (x);
           semSignal (rsem);
      semSignal (z);
      READUNIT();
      semWait (x);
           readcount--;
           if (readcount == 0) semSignal (wsem);
      semSignal (x);
    }
}
```

```
void writer ()
{
    while (true) {
      semWait (y);
           writecount++;
           if (writecount == 1) semWait (rsem);
      semSignal (y);
      semWait (wsem);
      WRITEUNIT();
      semSignal (wsem);
      semWait (y);
           writecount--;
           if (writecount == 0) semSignal (rsem);
      semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

Module – 3

5.a

Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes is deadlocked when each process in the set is blocked awaiting an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered



Deadlock can arise if four conditions hold simultaneously:

- Mutual exclusion
- Hold and wait
- No preemption
- Circular wait

Mutual exclusion

At least one resource must be non-sharable mode i.e. only one process can use a resource at a time. The requesting process must be delayed until the resource has been released. But mutual exclusion is required to ensure consistency and integrity of a database.

Hold and wait

A process must be holding at least one resource and waiting to acquire additional resources held by other processes.

No preemption A resource can be released only voluntarily by the process holding it after that process has completed its task i.e. no resource can be forcibly removed from a process holding it.

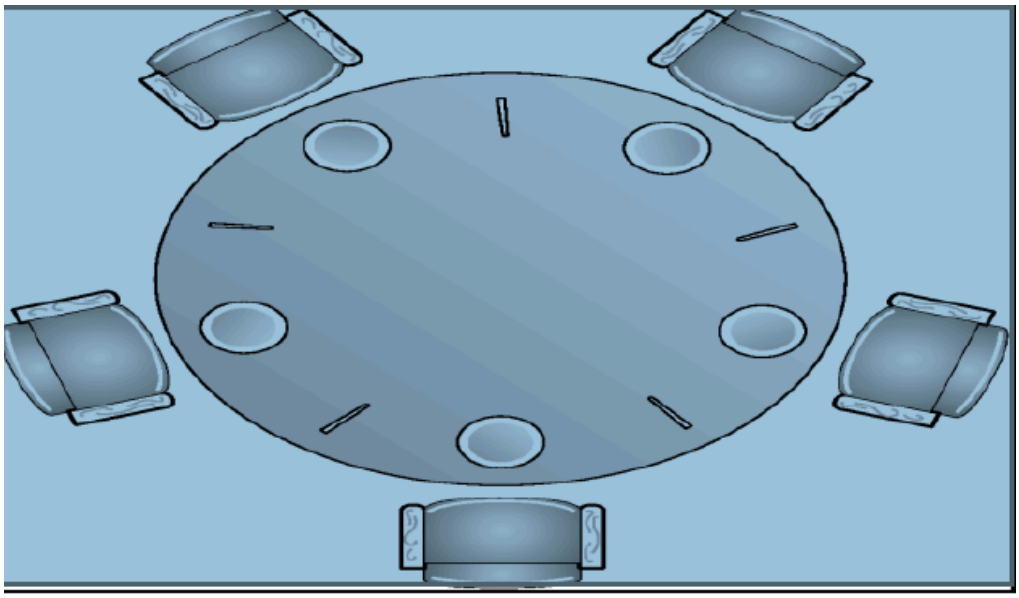
Circular wait

There exists a set {*P*0, *P*1, …, *P*n} of waiting processes such that *P*0 is waiting for a resource that is held by *P*1, *P*1 is waiting for a resource that is held by *P*2,…… …, *Pn*–1 is waiting for a resource that is held by *P*n, and *P*n is waiting for a resource that is held by *P*0.

5.b

- Five philosophers spend their lives thinking and eating.
- Philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher.
- In center of the table is a bowl of rice (or spaghetti), and the table is laid with five single chopsticks.

- From time to time, philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors).



A philosopher may pick up only one chopstick at a time.
She cannot pick up a chopstick that is already in hand of a neighbor.
When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks.
When she finishes eating, she puts down both of her chopsticks and start thinking again. The problem is to ensure that no philosopher will be allowed to starve because he cannot ever pick up both forks.

The dinning philosopher problem is considered a classic problem because it is an example of a large class of concurrency-control problems.

- Shared data

- semaphore chopstick[5];

Initially all values are 1

A philosopher tries to grab the chopstick by executing wait operation and releases the chopstick by executing signal operation on the appropriate semaphores.

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
      while (true) {
            think();
            wait (fork[i]);
            wait (fork [(i+1) mod 5]);
            eat();
            signal(fork [(i+1) mod 5]);
            signal(fork[i]);
      }
}
void main()
{
      parbegin (philosopher (0), philosopher (1), philosopher
(2),
            philosopher (3), philosopher (4));
      }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

This solution guarantees that no two neighbors are eating simultaneously but it has a possibility of creating a deadlock and starvation.

 Allow at most four philosophers to be sitting simultaneously at the table.

 Allow a philosopher to pick up her chopsticks if both chopsticks are available.

 An odd philosopher picks up her left chopstick first and an even philosopher picks up her right chopstick first.

 Finally no philosopher should starve.

6.a)

There are three common techniques for structuring a page table. They are:

Hierarchical Paging - Break up the logical address space into multiple page tables. A simple technique is a two-level page table.

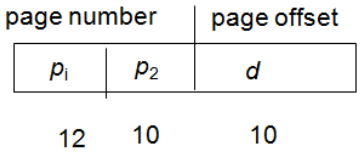
A logical address (on 32-bit machine with 1K page size

- a page number consisting of 22 bits
- a page offset consisting of 10 bits

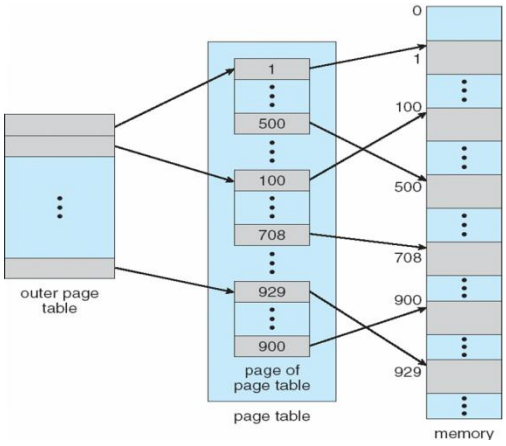
Since the page table is paged, the page number is furt

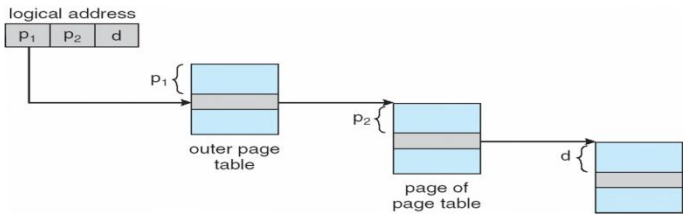
- a 12-bit page number
- a 10-bit page offset

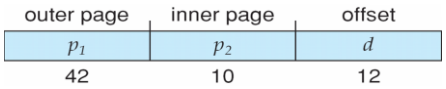
Thus, a logical address is as follows:

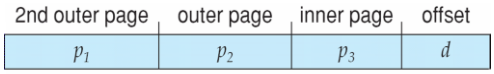


where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table









Hashed Page Table –

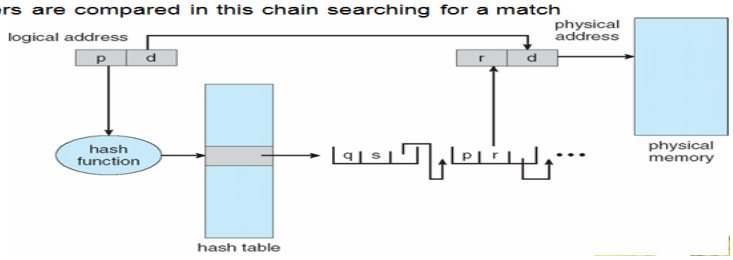
Common in address spaces > 32 bits

The virtual page number is hashed into a page table

- This page table contains a chain of elements hashing to the same location

Virtual page numbers are compared in this chain searching for a match

- If a match is found, the corresponding physical frame is extracted



Inverted Page Table –

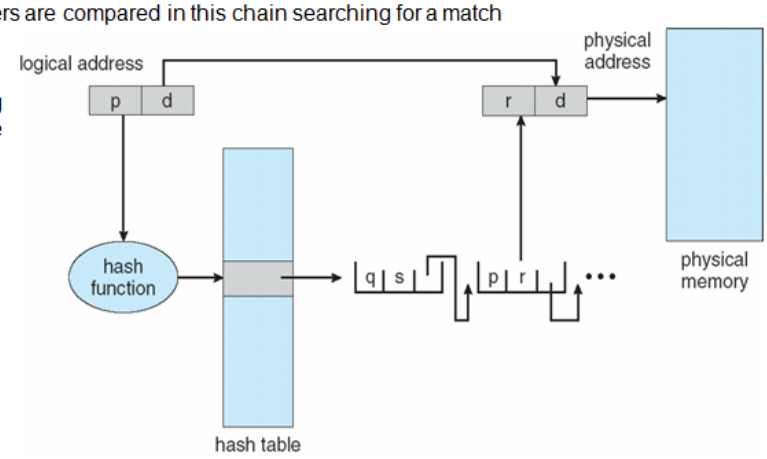
Common in address spaces > 32 bits

The virtual page number is hashed into a page table

- This page table contains a chain of elements hashing to the same location

Virtual page numbers are compared in this chain searching for a match

- If a match is found, the corresponding physical frame is extracted

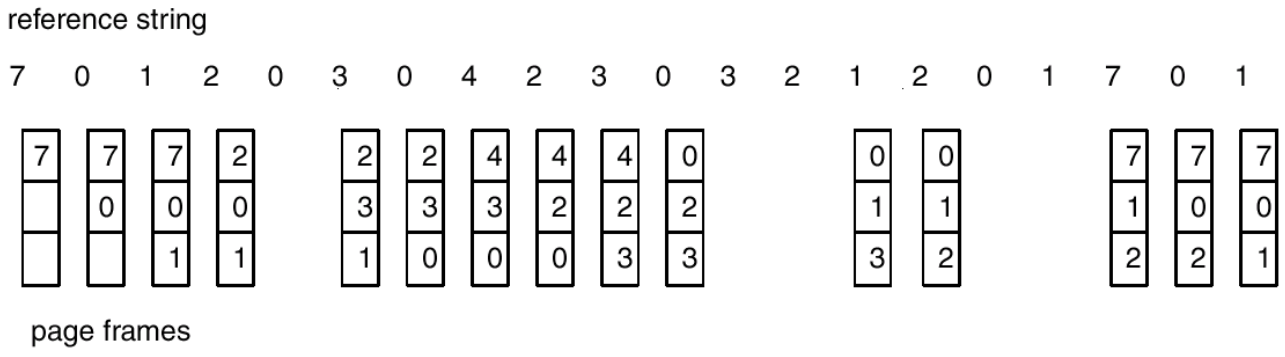


6.b)

Assuming 3 frames, find the number of page faults when the following algorithms are used: i) LRU ii) FIFO iii) Optimal. Note that initially all the frames are empty.

FIFO Page Replacement

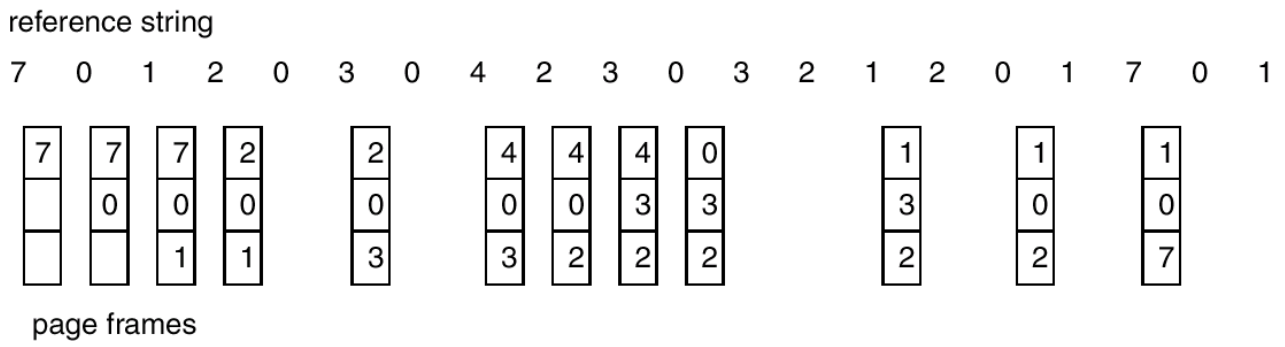
It is the simplest page – replacement algorithm. As the name suggests, the first page which has been brought into memory will be replaced first when there no space for new page to arrive. Initially, we assume that no page is brought into memory. Hence, there will be few (that is equal to number of frames) page faults, initially. Then, whenever there is a request for a page, it is checked inside the frames. If that page is not available, page – replacement should take place. **Example: Consider a reference string:** 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1 Let the number of frames be 3.



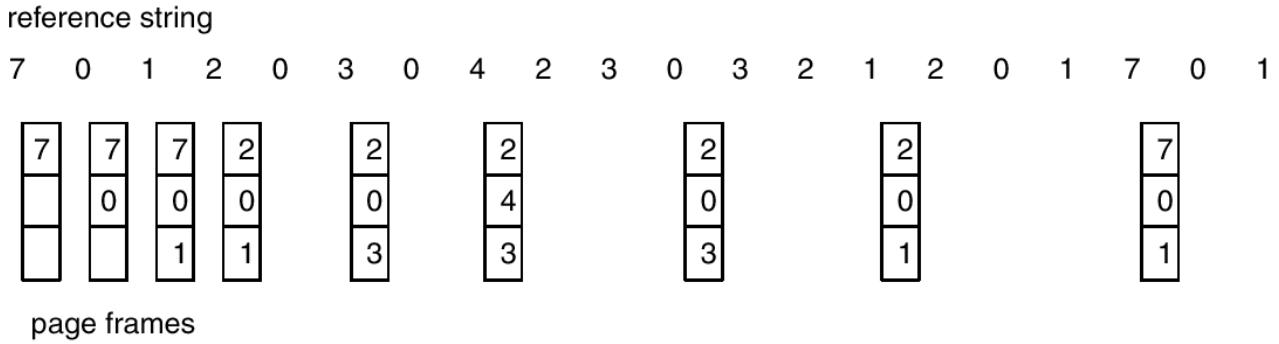
In the above example, there are 15 page faults.

LRU Page Replacement

Least Recently Used page replacement algorithm states that: **Replace the page that has not been used for the longest period of time.** This algorithm is better than FIFO. **Example:**



Optimal Page Replacement Algorithm An Optimal Page Replacement algorithm (also known as ***OPT*** or ***MIN*** algorithm) do not suffer from Belady's anomaly. It is stated as: **Replace the page that will not be used for the longest period of time.**



Here, number of page faults = 9

This algorithm results in lowest page – faults.

7.a) File operations

File is an abstract data type. To define it properly, we need to define certain operations on it:

 Creating a file: This includes two steps: find the space in file system and make an entry in the directory.

 Writing a file: To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Using the name of the file, the system searches the directory to find the location of the file. The system must keep a *write* pointer to the location in the file where the next write is to take place. The *write* pointer must be updated whenever a write occurs.

 Reading a file: To read from a file, we use a system call that specifies the name of the file and where (in memory) the next block of the file should be put.

 Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position is set to a given value. This file operation is also known as *file*

seek.
 Deleting a file: To delete a file, search the directory. Then, release all file space and erase the directory entry.
 Truncating a file: The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, the truncation allows all attributes to remain unchanged-except for file length. The file – length is reset to zero and its file space released.
Other common operations include *appending* new information to the end of an existing file and *renaming* an existing file.
Most of the file operations involve searching the directory for the entry associated with the named file. To avoid this constant searching, the OS keeps small table (known as *open – file table*) containing information about all open files. When a file operation is requested, this table is checked. When the file is closed, the OS removes its entry in the open-file table.
Every file which is open has certain information associated with it:
 File pointer: Used to track the last read-write location. This pointer is unique to each process.
 File open count: When a file is closed, its entry position (the space) in the open-file table must be reused. Hence, we need to track the number of opens and closes using the file open count.
 Disk location of the file: Most file operations require the system to modify data within the file. So, location of the file on disk is essential.
 Access rights: Each process opens a file in an access mode (read, write, append etc).
This information is by the OS to allow or deny subsequent I/O requests.

7.b) The Virtual File System (VFS)

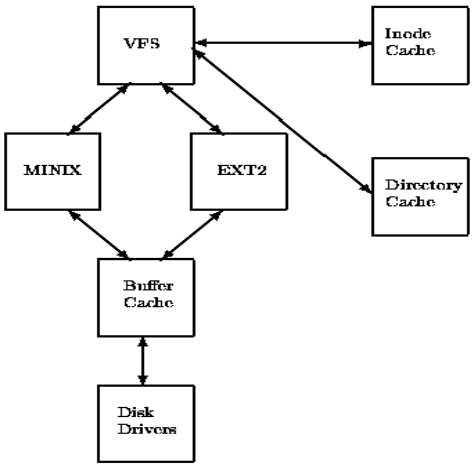


Figure: A Logical Diagram of the Virtual File System

Figure gif shows the relationship between the Linux kernel's Virtual File System and it's real file systems. The virtual file system must manage all of the different file systems that are mounted at any given time. To do this it maintains data structures that describe the whole (virtual) file system and the real, mounted, file systems.

Rather confusingly, the VFS describes the system's files in terms of superblocks and inodes in much the same way as the EXT2 file system uses superblocks and inodes. Like the EXT2 inodes, the VFS

inodes describe files and directories within the system; the contents and topology of the Virtual File System. From now on, to avoid confusion, I will write about VFS inodes and VFS superblocks to distinquish them from EXT2 inodes and superblocks.

As each file system is initialised, it registers itself with the VFS. This happens as the operating system initialises itself at system boot time. The real file systems are either built into the kernel itself or are built as loadable modules. File System modules are loaded as the system needs them, so, for example, if the VFAT file system is implemented as a kernel module then it is only loaded when a VFAT file system is mounted. When a block device based file system is mounted, and this includes the root file system, the VFS must read its superblock. Each file system type's superblock read routine must work out the file system's topology and map that information onto a VFS superblock data structure. The VFS keeps a list of the mounted file systems in the system together with their VFS superblocks. Each VFS superblock contains information and pointers to routines that perform particular functions.

So, for example, the superblock representing a mounted EXT2 file system contains a pointer to the EXT2 specific inode reading routine. This EXT2 inode read routine, like all of the file system specific inode read routines, fills out the fields in a VFS inode. Each VFS superblock contains a pointer to the first VFS inode on the file system. For the root file system, this is the inode that represents the ``/'' directory. This mapping of information is very efficient for the EXT2 file system but moderately less so for other file systems.

8 a) Free space management.

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free-space list. The freespace list records all free disk blocks-those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. Free-space management is done using different techniques as explained hereunder.

Linked List

Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory. This first block contains a pointer to the next free disk block, and so on. However, this scheme is not efficient. To traverse the list, we must read each block, which requires substantial I/O time. Fortunately, traversing the free list is not a frequent action. Usually, the OS simply needs a free block so that it can allocate that block to a file, so the first block in the free list is used.

Grouping

A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block. The first *n-1* of these blocks are actually free. The last block contains the addresses of another *n* free blocks, and so on. The importance of this implementation is that the addresses of a large number of free blocks can be found quickly, unlike in the standard linked-list approach.

8. b)

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files will be stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are: contiguous, linked, and indexed.

Contiguous Allocation

In contiguous allocation, files are assigned to contiguous areas of secondary storage. A

user specifies in advance the size of the area needed to hold a file to be created. If the desired amount of contiguous space is not available, the file cannot be created. A contiguous allocation of disk space is shown in Figure 7.11.

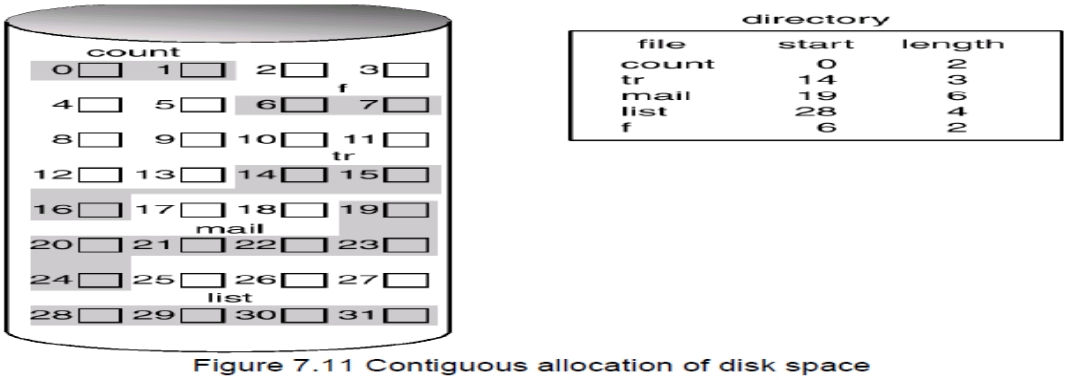


Figure 7.11 Contiguous allocation of disk space

One advantage of contiguous allocation is that all successive records of a file are normally physically adjacent to each other. This increases the accessing speed of records. It means that if records are scattered through the disk it is accessing will be slower. For sequential access the file system remembers the disk address of the last block and when necessary reads the next block. For direct access to block B of a file that starts at location L, we can immediately access block L+B. Thus contiguous allocation supports both sequential and direct accessing. The disadvantage of contiguous allocation algorithm is, it suffers from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. Depending on the total amount of disk storage and the average file size, external

fragmentation may be a minor or a major problem.

Linked Allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file as shown in Figure 7.12.

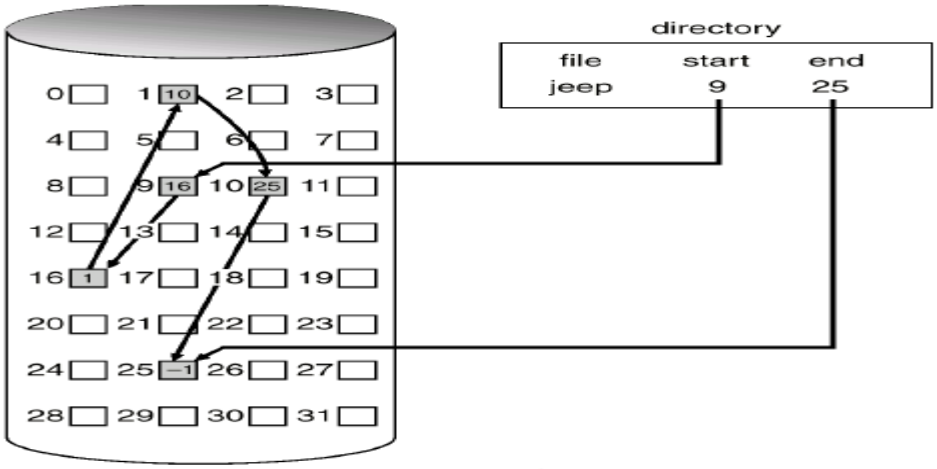


Figure 7.12 Linked Allocation of disk space

Linked allocation solves the problem of external fragmentation, which was present in contiguous allocation. But, still it has a disadvantage: Though it can be effectively used for sequential-access files, to find ith file, we need to start from the first location. That is, random-access is not possible.

Indexed Allocation

This method allows direct access of files and hence solves the problem faced in linked allocation. Each file has its own index block, which is an array of disk-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block as shown in Figure 7.13.

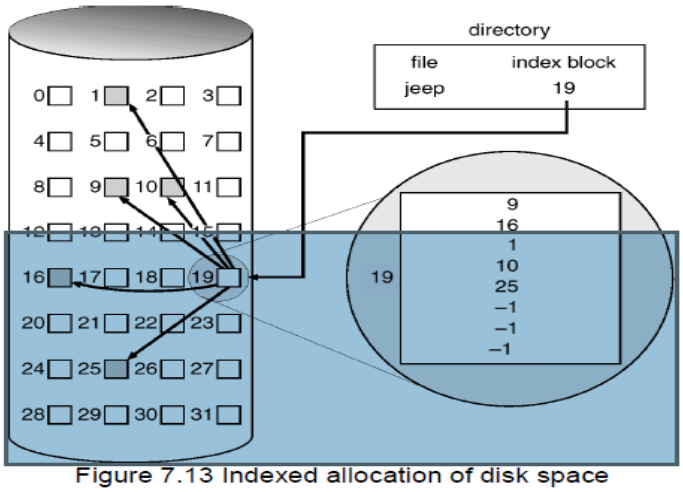


Figure 7.13 Indexed allocation of disk space

9.a)

The job of allocating CPU time to different tasks within an OS. While scheduling is normally thought of as the running and interrupting of processes, in Linux, scheduling also includes the running of the various kernel tasks. Running kernel tasks encompasses both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

Various aspects of scheduling in Linux are discussed here.

Kernel Synchronization: A request for kernel-mode execution can occur in two ways:

o A running program may request an operating system service, either explicitly via a system call, or implicitly, for example, when a page fault occurs.
o A device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt. Kernel synchronization requires a framework that will allow the kernel's critical sections to run without interruption by another critical section.

Linux uses two techniques to protect critical sections:
o Normal kernel code is non-preemptable : when a time interrupt is received while a process is executing a kernel system service routine, the kernel's **need_resched** flag is set so that the scheduler will run once the system all has completed and control is about to be returned to user mode.

o The second technique applies to critical sections that occur in an interrupt service routines. By using the processor's interrupt control hardware to disable interrupts during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.

Process Scheduling: Linux uses two process-scheduling algorithms: o A time-sharing algorithm for fair preemptive scheduling between multiple processes

o A real-time algorithm for tasks where absolute priorities are more important than fairness A process's scheduling class defines which algorithm to apply. For time-sharing processes, Linux uses a prioritized, credit based algorithm.

Symmetric Multiprocessing: Linux 2.0 was the first Linux kernel to support SMP hardware; separate processes or threads can execute in parallel on separate processors. To preserve the kernel's nonpreemptible synchronization requirements, SMP imposes the restriction, via a single kernel spinlock, that only one processor at a time may execute kernel-mode code.

9.b)

To the user, Linux's file system appears as a hierarchical directory tree obeying UNIX semantics. Internally, the kernel hides implementation details and manages the multiple different file systems via an abstraction layer, that is, the virtual file system (VFS). The Linux VFS is designed around object-oriented principles and is composed of two components:

 A set of definitions that define what a file object is allowed to look like

o The inode-object and the file-object structures represent individual files
o the file system object represents an entire file system
 A layer of software to manipulate those objects.

10. a) Component of Linux

Ans: The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

1. Kernel. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.

2. System libraries. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not needthe full privileges of kernel code.

3. System utilities. The system utilities are programs that perform individual, specialized management tasks. Some system utilities may be invoked just once to initialize and configure some aspect of the system; others—

known as *daemons* in UNIX terminology—may run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

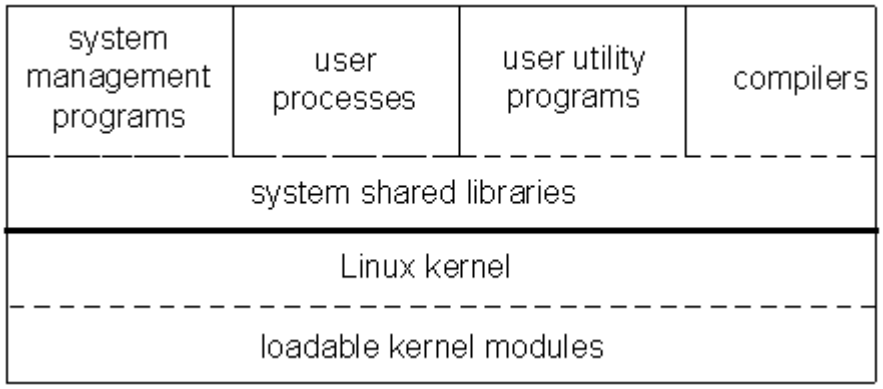


Figure 21.1 illustrates the various components that make up a full Linux system. The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode.**

10.b) Shared memory

In computer programming, shared memory is a method by which program processes can exchange data more quickly than by reading and writing using the regular operating system services. For example, a client process may have data to pass to a server process that the server process is to

modify and return to the client. Ordinarily, this would require the client writing to an output file (using the buffers of the operating system) and the server then reading that file as input from the buffers to its own work space. Using a designated area of shared memory, the data can be made directly accessible to both processes without having to use the system services. To put the data in shared memory, the client gets access to shared memory after checking a semaphore value, writes the data, and then resets the semaphore to signal to the server (which periodically checks shared memory for possible input) that data is waiting. In turn, the server process writes data back to the shared memory area, using the semaphore to indicate that data is ready to be read.

10.c) Inter process communication

Like UNIX, Linux informs processes that an event has occurred via signals. There is a limited number of signals, and they cannot carry information: Only the fact that a signal occurred is available to a process. The Linux kernel does not use signals to communicate with processes with are running in kernel mode, rather, communication within the kernel is accomplished via scheduling states and wait-queue structures.

Passing of Data among Processes: The pipe mechanism allows a child process to inherit a communication channel to its parent; data written to one end of the pipe can be read by the other. Shared memory offers an extremely fast way of communicating; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. To obtain synchronization, however, shared memory must be used in conjunction with another inter process communication mechanism.

10.d) journelling

A *journaling filesystem* is a *filesystem* that maintains a special file called a *journal* that is used to repair any inconsistencies that occur as the result of an improper shutdown of a computer. Such shutdowns are usually due to an interruption of the power supply or to a software problem that cannot be resolved without a rebooting.

A filesystem is a way of storing information on a computer that usually consists of a hierarchy of directories (also referred to as the *directory tree*) that is used to organize files. Each hard disk drive (HDD) or other storagedevice as well as each partition (i.e., logically independent section of a HDD) can have a different type of filesystem if desired.

Journaling filesystems write *metadata* (i.e., data about files and directories) into the journal that is flushed to the HDD before each command returns. In the event of a system crash, a given set of updates may have either been fully *committed* to the filesystem (i.e., written to the HDD), in which case there is no problem, or the updates will have been marked as not yet fully committed, in which case the system will read the journal, which can be *rolled up* to the most recent point of data consistency.