B. TECH. TCS-601

SIXTH SEMESTER EXAMINATION, 2008-09

OPERATING SYSTEMS

Time : 3 Hours *Total Marks : 100*

1. Attempt any four questions :

(a) Compare multitasking and multiuser operating system.

Ans. Multiuser Operating Systems : Many different users can share the same machine through time sharing and multiprogramming (e.g., UNIX, MacOS X, Windows NT). The OS divides its system time into time slices (milliseconds).

This gives each user the illusion to have his/her own machine.

The efficiency of a time-sharing system depends on

- the speed of the processor
- the length of the time slices
- how many users perform operations that require a full time slice.

Multiuser OSs are mainly used in distributed, rather than centralized environments, where several machines share resources over a network

In computing, *multitasking* is a method by which multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be *running* at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.

(b) What are the essential and desirable properties of an operating system?

Ans. User friendly, convenient to user, efficient in resource usage, multithreaded, small size of os, easily configurable and installable, fault tolerant etc.

(c) Explain in brief real time operating system. Illustrate some areas where they are used.

Ans. A Real- Time Operating System (RTOS) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers), industrial robots, spacecraft, industrial control (see SCADA), and scientific research equipment.

A RTOS facilitates the creation of a real-time system, but does not guarantee the final result will be real-time; this requires correct development of the software. An RTOS does not necessarily have high throughput; rather, an RTOS provides facilities which, if used properly, guarantee deadlines can be met generally or deterministically (known as soft or hard real-time, respectively). An RTOS will typically use specialized scheduling algorithms in order to provide the real-time developer with the tools necessary to produce deterministic behavior in the final system. An RTOS is valued more for how quickly and/or predictably it can respond

to a particular event than for the amount of work it can perform over a given period of time. Key factors in an RTOS are therefore a minimal interrupt latency and a minimal thread switching latency.

An early example of a large-scale **real-time** operating system was Transaction Processing Facility developed by American Airlines and IBM for the Sabre Airline Reservations System.

Design philosophies : Two basic designs exist:

- Event-driven (priority scheduling) designs switch tasks only when an event of higher priority needs service, called pre-emptive priority.
- Time-sharing designs switch tasks on a clock interrupt, and on events, called round robin.

Time-sharing designs switch tasks more often than is strictly needed, but give smoother, more deterministic multitasking, giving the illusion that a process or user has sole use of a machine. Early CPU designs needed many cycles to switch tasks, during which the CPU could do nothing useful, so early OSes tried to minimize wasting CPU time by maximally avoiding unnecessary task-switches.

Scheduling : In typical designs, a task has three states: 1) running, 2) ready, 3) blocked. Most tasks are blocked, most of the time. Only one task per CPU is running. In simpler systems, the ready list is usually short, two or three tasks at most.

The real key is designing the scheduler. Usually the data structure of the ready list in the scheduler is designed to minimize the worst-case length of time spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled. But, the choice of data structure depends also on the maximum number of tasks that can be on the ready list.

If there are never more than a few tasks on the ready list, then a simple unsorted bidirectional linked list of ready tasks is likely optimal. If the ready list usually contains only a few tasks but occasionally contains more, then the list should be sorted by priority, so that finding the highest priority task to run does not require iterating through the entire list. Inserting a task then requires walking the ready list until reaching either the end of the list, or a task of lower priority than that of the task being inserted. Care must be taken not to inhibit preemption during this entire search; the otherwise-long critical section should probably be divided into small pieces, so that if, during the insertion of a low priority task, an interrupt occurs that makes a high priority task ready, that high priority task can be inserted and run immediately (before the low priority task is inserted).

The critical response time, sometimes called the flyback time, is the time it takes to queue a new ready task and restore the state of the highest priority task. In a well-designed RTOS, readying a new task will take 3-20 instructions per ready queue entry, and restoration of the highest-priority ready task will take 5-30 instructions. On a 20MHz 68000 processor, task switch times run about 20 microseconds with two tasks ready. 100 MHz ARM CPUs switch in a few microseconds.

In more advanced real-time systems, real-time tasks share computing resources with many non-real-time tasks, and the ready list can be arbitrarily long. In such systems, a scheduler ready list implemented as a linked list would be inadequate.

Algorithms : Some commonly used RTOS scheduling algorithms are:

- Cooperative scheduling
  - Round-robin scheduling
- Preemptive scheduling
  - Fixed priority pre-emptive scheduling, an implementation of preemptive time slicing
  - Fixed-Priority Scheduling with Deferred Preemption
  - Fixed-Priority Non-preemptive Sche duling

- Critical section preemptive scheduling
- Static time scheduling

- Earliest Deadline First approach
- Advanced scheduling using the stochastic and MTG

Intertask communication and resource sharing : Multitasking systems must manage sharing data and hardware resources among multiple tasks. It is usually "unsafe" for two tasks to access the same specific data or hardware resource simultaneously. ("Unsafe" means the results are inconsistent or unpredictable, particularly when one task is in the midst of changing a data collection. The view by another task is best done either before any change begins, or after changes are completely finished.) There are three common approaches to resolve this problem:

- Temporarily masking/disabling interrupts
- Binary semaphores
- Message passing

General-purpose operating systems usually do not allow user programs to mask (disable) interrupts, because the user program could control the CPU for as long as it wished. Modern CPUs make the interrupt disable control bit (or instruction) inaccessible in user mode to allow operating systems to prevent user tasks from doing this. Many embedded systems and RTOSs, however, allow the application itself to run in kernel mode for greater system call efficiency and also to permit the application to have greater control of the operating environment without requiring OS intervention.

On single-processor systems, if the application runs in kernel mode and can mask interrupts, often that is the best (lowest overhead) solution to preventing simultaneous access to a shared resource. While interrupts are masked, the current task has *exclusive* use of the CPU; no other task or interrupt can take control, so the critical section is effectively protected. When the task exits its critical section, it must unmask interrupts; pending interrupts, if any, will then execute. Temporarily masking interrupts should only be done when the longest path through the critical section is shorter than the desired maximum interrupt latency, or else this method will increase the system's maximum interrupt latency. Typically this method of protection is used only when the critical section is just a few source code lines long and contains no loops. This method is ideal for protecting hardware bitmapped registers when the bits are controlled by different tasks.

When the critical section is longer than a few source code lines or involves lengthy looping, an embedded/real-time programmer must resort to using mechanisms identical or similar to those available on general-purpose operating systems, such as semaphores and OS-supervised interprocess messaging. Such mechanisms involve system calls, and usually invoke the OS's dispatcher code on exit, so they can take many hundreds of CPU instructions to execute, while masking interrupts may take as few as three instructions on some processors. But for longer critical sections, there may be no choice; interrupts cannot be masked for long periods without increasing the system's interrupt latency.

A binary **semaphore** is either locked or unlocked. When it is locked, a queue of tasks can wait for the semaphore. Typically a task can set a timeout on its wait for a semaphore. Problems with semaphore based designs are well known: priority inversion and deadlocks.

In **priority inversion**, a high priority task waits because a low priority task has a semaphore. A typical solution is to have the task that has a semaphore run at (inherit) the priority of the highest waiting task. But this simplistic approach fails when there are multiple levels of waiting (A waits for a binary semaphore locked by B, which waits for a binary semaphore locked by C). Handling multiple levels of inheritance without introducing instability in cycles is not straightforward.

In a deadlock, two or more tasks lock a number of binary semaphores and then wait forever (no timeout) for other binary semaphores, creating a

cyclic dependency graph. The simplest deadlock scenario occurs when two tasks lock two semaphores in lockstep, but in the opposite order. Deadlock is usually prevented by careful design, or by having floored semaphores (which pass control of a semaphore to the higher priority task on defined conditions).

The other approach to resource sharing is for tasks to send messages. In this paradigm, the resource is managed directly by only one task; when another task wants to interrogate or manipulate the resource, it sends a message to the managing task. This paradigm suffers from similar problems as binary semaphores: Priority inversion occurs when a task is working on a low-priority message, and ignores a higher-priority message (or a message originating indirectly from a high priority task) in its in-box. Protocol deadlocks occur when two or more tasks wait for each other to send response messages.

Although their real-time behavior is less crisp than semaphore systems, simple message-based systems usually do not have protocol deadlock hazards, and are generally better-behaved than semaphore systems.

Interrupt handlers and the scheduler : Since an interrupt handler blocks the highest priority task from running, and since real time operating systems are designed to keep thread latency to a minimum, interrupt handlers are typically kept as short as possible. The interrupt handler defers all interaction with the hardware as long as possible; typically all that is necessary is to acknowledge or disable the interrupt (so that it won't occur again when the interrupt handler returns). The interrupt handler then queues work to be done at a lower priority level, often by unblocking a driver task (through releasing a semaphore or sending a message). The scheduler often provides the ability to unblock a task from interrupt handler context.

Memory allocation : Memory allocation is even more critical in an RTOS than in other operating systems.

First, speed of allocation is important. A standard memory allocation scheme scans a linked list of indeterminate length to find a suitable tree memory block; however, this is unacceptable as memory allocation has to occur in a fixed time in an RTOS.

The simple fixed-size-blocks algorithm works astonishingly well for simple embedded systems.

Examples : These are the best known, most widely deployed real-time operating systems. See list of real-time operating systems for a comprehensive list. Also, see List of operating systems for all types of operating systems.

- QNX
- RLinux
- VxWorks
- Windows CE

(e) Draw the layered structure of an operating system.

Ans. Structure view : Layered system design.

A general philosophy that builds on the above approach.

Decompose functionality into layers such that. Hardware is level 0, and layer t accesses functionality at layer (t-1) or less

Access via appropriately defined system calls.

Advantages

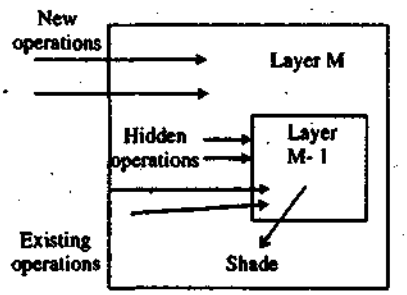
Modular design: well defined interfaces between layers

Prototyping/development.

Association between function and layer eases overall OS desing.

OS development and debugging is layer by layer.

Simplifies debugging and system verification



(d) What are the different services provided by an operating system?

Ans. Functional view :

- Program execution and handling
  - Starting programs, managing their execution and communicating their results.
- I/O operations
  - Mechanisms for initiating and managing I/O
- File-system Management
  - Creating, maintaining and manipulating files
- Communications
  - Between processes of the same user
  - Such as sending result of input request to a user progra
  - Between different users
- Exception detection and handling
  - Protection related issues
  - Safety in the case of power failures via backups
  - Detecting undesirable state such as printers out of paper
- Resource allocation
  - Includes processor and I/O scheduling, memory management
- Accounting
  - To track users usage of resources for billing and statistical reasons
- Protection
  - Maintaining integrity of user's data
  - Integrity checks to keep out unauthorized users

Maintaining logs of incorrect attempts

(f) What do you mean by system protection? How it is achieved?

Ans. Protection must ensure that only those processes that have gained proper authorization from the OS can operate on memory segments, the CPU and other resources.

- Operating system consists of a collection of objects, hardware or software.
- Each object has a unique name and can be accessed through a well-defined set of operations.
- Protection problem - ensure that each object is accessed correctly and only by those processes that are allowed to do so.
- Enforcement of the policies governing resource usage.
- A protection system must have the flexibility to enforce a variety of policies that can be declared to it.

Protection :

- Maintaining integrity of user's data
- Integrity checks to keep out unauthorized users
- Maintaining logs of incorrect attempts

2. Attempt any four :

(a) What is PCB (Process Control Block)?

Ans. A Process Control Block (PCB, also called Task Control Block or Task Struct) is a data structure in the operating system kernel containing the information needed to manage a particular process. The PCB is "the manifestation of a process in an operating system".

Included Information : Implementations differ, but in general a PCB will include directly or indirectly:

- The identifier of the process (a process identifier, or PID)
- Register values for the process including, notably, the Program Counter value for the process
- The address space for the process
- Priority (in which higher priority process gets first preference. e.g., nice value on Unix operating systems)
- Process accounting information, such as when the process was last run, how much CPU time it has accumulated, etc.
- Pointer to the next PCB i.e., pointer to the PCB of the next process to run

- I/O Information (i.e., I/O devices allocated to this process, list of opened files, etc.)

During a context switch, the running process is stopped and another process is given a chance to run. The kernel must stop the execution of the running process, copy out the values in hardware registers to its PCB, and update the hardware registers with the values from the PCB of the new process.

Location of the PCB : Since the PCB contains the critical information for the process, it must be kept in an area of memory protected from normal user access. In some operating systems the PCB is placed in the beginning of the kernel stack of the process since that is a convenient protected location.

(b) Explain principle of concurrency.

Ans. Why Concurrency? On a single-processor machine, the operating system's support for *concurrency* allows multiple applications to share resources in such a way that applications appear to run at the same time. Since a typical application does not consume all resources at a given time, a careful coordination can make each application run as if it owns the entire machine. An analogy is juggling. While there are only two hands, each ball thinks that it is caught and tossed with a pair of dedicated hands. There are a number of benefits for an operating system to provide concurrency:

1. Of course, the most obvious benefit is to **be able to run multiple applications at the same time.**
2. Since resources that are unused by one application can be used for other **applications, concurrency** allows **better resource utilization.**
3. Without concurrency, each application has to be run to completion before the next one can be run.Therefore, concurrency allows a **better average response time** of individual applications.
4. Concurrency does not merely timeshare the computer; concurrency can actually achieve **better performance**. For example, if one application uses only the processor, while another application uses only the disk drive, the time to run both applications concurrently to completion will be shorter than the time to run each application consecutively.

Concurrency also introduces certain drawbacks:

1. Multiple applications need to be protected from one another.
2. Multiple applications may need to coordinate through additional mechanisms.
3. Switching among applications requires additional performance overheads and complexities in operating systems (e.g., deciding which application to run next.)
4. In extreme cases of running too many applications concurrently will lead to severely degraded performance.

Overall, when properly used, concurrency offers more benefits than drawbacks.

Thread : A *thread* is a sequential execution stream, and it is also the smallest scheduling unit of concurrency to run on a processor. The beauty of the thread is that each thread can be programmed as if it owns the entire CPU (e.g., you can use an infinite loop within a thread without halting the entire system). In other words, a thread contains the states of its own program counter, register values, and execution stacks. Therefore, threads provide the illusion of having an infinite number of CPUs, even on a single-processor machine.

Threads simplify programming significantly, and Microsoft Word is an example. As you are typing in Word, there is a thread dedicated for checking grammar, a thread for checking spelling, a thread for reformatting the text, and many other threads for various purposes. Since the thread for grammar checking can be programmed independently from the thread for spelling check, the difficulty for programming a large application like Word is greatly simplified.

Address Space : An *address space* contains all states necessary to run a program–the code, the data, the stack, the program counter, register values, resources required by the program, and the status of the running program.

Process : A *process* is a fundamental unit of computation. A process, under UNIX, consists of everything that is needed to run a sequential stream of execution. In particular, it consists of an address space and at least a thread of execution. The address space offers protection among processes, and threads offer concurrency.

Process =? Program : A *program* is simply a collection of statements in C or any other programming language. A process is a running instance of the program, with additional states and system resources.

In one sense, a process is more than a program, since it is possible for two processes to run the same program. The code of the program is just parts of running states within those two processes.

From a different perspective, a program is more than a process, since it is possible for a program to create multiple processes.

As an analogy, a program is like a recipe, and a process is like everything that is needed (e.g., kitchen) to prepare a dish. Two different chefs can cook the same recipe in different kitchens. One complex recipe can also consist of several simpler recipes that can be made into separate dishes.

Some Definitions : Up to this point, we have encountered many similar terms. Let's take a moment to distinguish among them.

Uniprogramming means running one process at the time.

Multiprogramming means running multiple processes (with separate address spaces) concurrently on a machine.

Multiprocessing means running programs on a machine with multiple processors.

Multithreading means having multiple threads per address space.

Multitasking means that a single user can run multiple processes.

Classifications of Operating Systems : With the vocabulary of process, thread and address space, we can classify operating systems into four major categories.

	Single address space	**Multiple address spaces**
Single thread	MS DOS, Macintosh	Traditional UNIX
Multiple threads	Embedded systems	Windows NT, Solaris, OS/2

Threads and Dispatching Loop : Inside each thread, there is a *thread control block*. The thread control block maintains the execution states of the thread, the status of the thread (e.g., running or sleeping), and scheduling information of the thread (e.g., priority).

Threads are run from a *dispatching loop*.

```
LOOP
    Run thread
    Save states (into the thread control block)
    Choose a new thread to run
    Load states from a different thread (from the
    thread control block)
```

To run a thread, just load its states (registers, program counter, stack pointer) into the CPU, and do a jump. The process of saving the states of one thread and restoring states of another thread is often called a ***context switch***. The decision of which thread to run next involves **scheduling**. Although this dispatching loop looks simple, there are a number of questions we need to address: How does the dispatcher regain control after a thread starts running? What states should a thread save? How does the dispatcher choose the next thread?

How Does the Dispatcher Regain Control?

The dispatcher gets control back from the running thread in two ways:

1. **Internal events** (sleeping beauty-go to sleep and hope Prince Charming will wake you):
   (a) A thread is waiting for I/O.
   (b) A thread is waiting for some other thread.
   (c) Yield–a thread gives up CPU voluntarily.
2. **External events:**
   (a) Interrupts–A completed disk request wakes up the dispatcher, so the dispatcher can choose another thread to run).
   (b) Timer–it's like an alarm clock.

What States should a Thread Save?

A thread should save anything that the next thread may trash before a context switch: program counter, registers, changes in execution stack. Each thread should be treated as an independent stream of execution.

As a side note, a context switch can also occur during an interrupt. During an interrupt, hardware causes the CPU to stop what it's doing, and to run the interrupt handler. The handler saves the states of the interrupted thread, runs the handler code, and restores the states of the interrupted thread.

How Does the Dispatcher Choose the Next Thread?

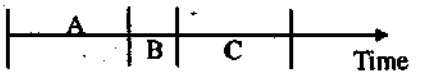
The dispatcher keeps a list of threads that are ready to run.

If no threads are ready to run–the dispatcher just loops.

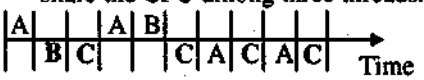
If one thread is ready to run-easy.

If more than one thread are ready to run, we can choose the next thread according to different scheduling policies. Some examples are FIFO (first in, first out), LIFO (last in, first out), and priority-based policies.

The dispatcher also has the control of how to share the CPU among multiple threads. Suppose that we have thread A, B and C. At one extreme, a dispatcher can run one thread to completion before running the other.

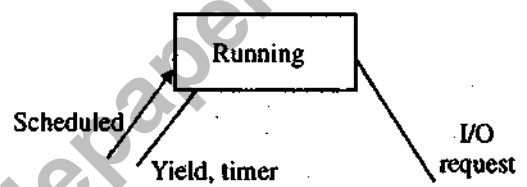


Alternatively, a dispatcher can use the timer to time share the CPU among three threads.



Per-Thread States : Each thread can be in one of three states:

1. Running-has the CPU
2. Blocked-waiting for I/O or another thread
3. Ready to run-on the ready list, waiting for the CPU



(c) Demonstrate process synchronization using procedure consumer problem.

Ans. In computer science the producer-consumer problem (also known as the **bounded-buffer** problem) is a classical example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time the consumer is consuming the data (i.e. removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to go to sleep if the buffer is full. The next time the consumer removes an item from the buffer, it wakes up the producer who starts to fill the buffer again. In the same way, the consumer goes to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer. The solution can be

reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

The problem can also be generalized to have multiple producers and consumers.

Implementations

Inadequate implementation : This solution has a race condition. To solve the problem, a careless programmer might come up with a solution shown below. In the solution two library routines are used, sleep and wakeup. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. itemCount is the number of items in the buffer.

```
int itemCount
procedure producer() {
   while (true) {
      item = produceItem()
      if (itemCount == BUFFER_SIZE) {
         sleep()
      }
      putItem Into Buffer(item)
      itemCount = itemCount + 1
      if (itemCount == 1) {
         wakeup( consumer)
      }
}
   }
procedure consumer() {
   while (true) {
      if (itemCount ==.0) {
         sleep( )
      }
      item = removeItemFromBuffer( )
      itemCount = itemCount - 1
      if (itemCount == BUFFER_SIZE - 1) {
         wakeup(producer)
      }
      consumeItem(item)
      }
}
```

The problem with this solution is that it contains a race condition that can lead into a deadlock. Consider the following scenario:

1. The consumer has just read the variable item Count, noticed it's zero and is just about to move inside the if-block.
2. Just before calling sleep, the consumer is interrupted and the producer is resumed.
3. The producer creates an item, puts it into the buffer, and increases item Count.
4. Because the buffer was empty prior to the last addition, the producer tries to wake up the consumer.
5. Unfortunately the consumer wasn't yet sleeping, and the wakeup call is lost. When the consumer resumes, it goes to sleep and will never be awakened again. This is because the consumer is only awakened by the producer when itemCount is equal to 1.
6. The producer will loop until the buffer is full, after which it will also go to sleep.

Since both processes will sleep forever, we have run into a deadlock. This solution therefore is unsatisfactory.

An alternative analysis is that if the programming language does not define the semantics of concurrent accesses to shared variables (in this case item Count) without use of synchronization, then the solution is unsatisfactory for that reason, without needing to explicitly demonstrate a race condition.

Using semaphores : Semaphores solve the problem of lost wakeup calls. In the solution below we use two semaphores, fillCount and emptyCount, to solve the problem. fillCount is incremented and emptyCount decremented when a new item has been put into the buffer. If the producer tries to decrement empty Count while its value is zero, the producer is put to sleep. The next time an item is consumed, emptyCount is incremented and the producer wakes up. The consumer works analogously.

```
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE
procedure producer ( ) {
   while (true) {
      item = produceItem ()
      down(empty Count)
      putItemIntoBuffer( item)
      up(fillCount)
   }
}
procedure consumer( ) {
   while (true) {
      down(fillCount)
      item = removeItemFromBuffer( )
      up(emptyCount)
      consumeItem(item)
   }
}
```

The solution above works fine when there is only one producer and consumer. Unfortunately, with multiple producers or consumers this solution contains a serious race condition that could result in two or more processes reading or writing into the same slot at the same time. To understand how this is possible, imagine how the procedure putItemIntoBuffer() can be implemented. It could contain two actions, one determining the next available slot and the other writing into it. Ifthe procedure can be executed concurrently by multiple producers, then the following scenario is possible:

1. Two producers decrement empty Count
2. One of the producers determines the next empty slot in the buffer
3. Second producer determines the next empty slot and gets the same result as the first producer
4. Both producers write into the same slot

To overcome this problem, we need a way to make sure that only one producer is executing put ItemIntoBuffer() at a time. In other words we need a way to execute a critical section with mutual exclusion. To accomplish this we use a binary semaphore called mutex. Since the value of a binary semaphore can be only either one or zero, only one process can be executing between down(mutex) and up(mutex). The solution for multiple producers and consumers is shown below.

```
semaphore mutex = 1
semaphore fillCount = 0
semaphore emptyCount = BUFFER_SIZE
procedure producer () {
   while (true) {
      item = produceItem()
      down(empty Count)
      down(mutex)
      putItemIntoBuffer(item)
      up(mutex)
      up(fillCount)
   }
   up (fillCount) lithe consumer may not finish
   before the producer.
}
procedure consumer() {
   while(true) {
      down(fillCount)
      down(mutex)
      item = removeItemFromBuffer()
      up(mutex)
      up(emptyCount)
      consumeItem(item)
   }
}
```

Notice that the order in which different semaphores are incremented or decremented is essential: changing the order might result in a deadlock.

Using monitors : The following pseudo code shows a solution to the producer-consumer problem using monitors. Since mutual exclusion is implicit with monitors, no extra effort is necessary to protect critical section. In other words, the solution shown below works with

any number of producers and consumers without any modifications. It is also noteworthy that using monitors makes race conditions much less likely than when using semaphores.

```
monitor Producer Consumer {
   int itemCount
   condition full
   condition empty
   procedure add(item) {
      while (item Count = = BUFFER_SIZE) {
         wait (full)
      }
      putItemIntoBuffer(item)
      itemCount = itemCount + 1
      if (item Count = = 1) {
         notify(empty)
      }
}
procedure remove() {
   while (itemCount == 0) {
      wait(empty)
   }
   item = removeItemFromBuffer()
   itemCount = itemCount - 1
   if (itemCount == BUFFER_SIZE - 1) {
      notify(full)
   }
   return item;
   }
}
procedure producer() {
   while (true) {
      item = produceItem()
      ProducerConsumer.add( item)
   }
}
procedure consumer() {
   while(true) {
      item = ProducerConsumer.remove()
      consumeItem()
   }
}
```

Using Composable Memory Transactions Composable Memory Transactions is a special form of Software Transactional.

(d) What is critical section? Design algorithm to solve this problem.

Ans. *n* processes all competing to use some shared data. Each process has a code segment, called *critical section,* in which the shared data is accessed.

Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

A solution to critical section problem must satisfy the following conditions :

Mutual Exclusion. If process *Pi* is executing in its critical section, then no other processes can be executing in their critical section.

Progress. At least one process requesting entry into CS will be able to enter it if there is no other process in it.

Bounded Waiting. No process waits indefinitely to enter CS once it has requested entry.

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the *n* processes.

Shared variables:

int turn;

initially **turn** = 0

Turn variable

P0	**P1**
while (turn != 0) ;	while (turn != I);
/ Do nothing */*	*/* Do nothing */*
critical section	*critical section*
turn = 1;	turn = 0;
remainder section	remainder section

Shared variable *turn* indicates who is allowed to enter next, can enter if *turn = me*

On exit, point variable to other process

Deadlock if other process never enters

+Satisfies mutual exclusion: Only one process can enter in CS

-It does not satisfy the progress requirement, as it requires strict alternation of processes to enter CS. The pace of execution is dictated by slower process.

If turn=0, PI is ready to enter into CS, P1 can not do so, eventhough P0 may be in the RS.

If one process fails in CS or RS, other process is blocked permanently.

Problem with Alg 1

It does not retain sufficient information about the state of each process.

AlgI remembers only which process is allowed to enter the CS.

To solve this problem, variable turn is replaced by boolean flag [2]; flag [0] is for P0; and flag[1] is for P1. Each process may examine the other's flag but may not alter it.

When a process wishes to enter CS, it periodically checks other's flag until that flag is false (other process is not in CS)

- The process sets its own flag true and enters CS.

When it leaves CS, it sets its flag to false

initially **flag** [0] = **flag** [1] = **false.**

P0	**P1**
while (flag[1]);	while (flag [0])
/ Do nothing */*	*/*Do nothin*/*
flag [0] = true;	flag [1] = true;
critical section	*critical section*
flag [0] = false;	flag[1] = false;

Mutual exclusion is satisfied.

If one process fails outside CS the other process is not blocked.

Solution is worst than previous solution.

It does not even guarantee ME.

P0 executes the while statement and fins flag [1] set to false.

P1 executes the while statement and fills flag [0] set to false.

P0 sets flag [0] to true and enters its CS.

P1 sets flag [1] to true and enters its CS.

Correct solution : Combining the key ideas of previous algorithms

Dekker's Algorithm

Use *flags* for mutual exclusion, *turn* variable to break deadlock

Handles mutual exclusion, deadlock, and starvation

Peterson's Algorithm

P0	**P1**
flag [0] = true;	flag [1] = true;
turn = 1;	turn = 0;
while (flag[1] && turn==1)	while (flag[0] && turn==0)
/ Do Nothing */*;	*/* Do nothing */*;
critical section	*critical section*
flag [0] = false;	flag [1] = false;
remainder section	**remainder section**

We need to show that

ME is preserved

The progress requirement is satisfied

The bounded-waiting requirement is met.

ME is preserved :

If both processes enter the CS both flag [0]= = flag [1]= = true

Both could not execute while loop successfully as turn is either 0 or 1.

Progress :

While P1 exits CS it sets flag [1] = false, allowing P0 to enter CS.

P1 and P0 will enter the CS (Progress)

Bounded waiting: P1 will enter the CS after at most one entry by P0 and vice versa.

(e) How can the interprocess communication be achieved?

Ans. Inter-process Communication (IPC)

IPC facility provides a mechanism to allow processes to communicate and synchronize their actions. Processes can communicate through shared memory or message passing.

Both schemes may exist in OS.
The shared-memory method requires communication processes to share some variables.
The responsibility for providing communication rests with the programmer.
The OS only provides shared memory.
Example: producer-consumer problem.
Message system - Processes communicate with each other without resorting to shared variables.
If P and Q want to communicate, a communication link exists between them.
OS provides this facility.
IPC facility provides two operations:
send(message) - message size fixed or variable
receive(message)
If *P* and *Q* wish to communication, they need to:
establish a *communication link* between them
exchange messages via send/receive
Implementation of communication link
physical (e.g., shared memory, hardware bus)
logical (e.g., logical properties)
Methods for logical implementation of a link
Direct communication
Indirect communication
Direct Communication : Processes must name each other explicitly:
send *(P, message)* – send a message to process P
receive(Q, *message)* – receive a message from process Q
Properties of communication link
Links are established automatically.
A link is associated with exactly one pair of communicating processes.
Between each pair there exists exactly one link.
The link may be unidirectional, but is usually bi-directional.
This exhibits both symmetry and asymmetry in addressing
Symmetry: Both the sender and the receiver processes must name the other to communicate.
Asymmetry: Only sender names the recipient, the recipient is not required to name the sender.
The send and receive primitives are as follows.

- Send (P, message)– send a message to process P.
- Receive(id, message)– receive a message from any process.

Disadvantages: Changing a name of the process creates problems
Indirect Communication :
The messages are sent and received from mailboxes (also referred to as ports).
A mailbox is an object
Process can place messages
Process can remove messages.
Two processes can communicate only if they have a shared mailbox.
Operations
create a new mailbox
send and receive messages through mailbox
destroy a mailbox
Primitives are defined as:
send(A, message) - send a message to mailbox A
receive(A, *message)* - receive a message from mailbox A
Mailbox sharing
P1, *P2*, and *P3* share mailbox A.
P1, sends; *P2* and *P3* receive.
Who gets a message?
Solutions
Allow a link to be associated with at most two processes.
Allow only one process at a time to execute a receive operation.
Allow the system to select arbitrarily the receiver.
Sender is notified who the receiver was.
Properties of a link:
A link is established if they have a shared mailbox
A link may be associated with more than two boxes.
Between a pair of processes then may be number of links
A link may be either unidirectional or bi-directional.

OS provides a facility

To create a mailbox

Send and receive messages through mailbox

To destroy a mail box.

The process that creates mailbox is a owner of that mailbox. The ownership and send and receive privileges can be passed to other processes through system calls.

Messages are directed and received from mailboxes (also referred to as ports).

Each mailbox has a unique id.

Processes can communicate only if they share a mailbox.

Example:

Producer process:

repeat

.....

Produce an item in nextp

.....

send(consumer,nextp);

until false;

Consumer process

repeat.. .. receive(producer, nextc);

Consume the item in nextc ... until false;

Synchronization : Message passing may be either blocking or non-blocking.

Blocking is considered synchronous

Non-blocking is considered asynchronous

send and **receive** primitives may be either blocking or non-blocking.

Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox.

Non-blocking send: The sending process sends the message and resumes operation.

Blocking receive: The receiver blocks until a message is available.

Non-blocking receive: The receiver receives either a valid message or a null.

(f) Define following

(i) Dispatcher

(ii) Context switching

Ans. (i) Dispatcher: The dispatcher monitors processes and decides when to switch execution from one process to another. When a process completes... the dispatcher reports back to the scheduler, the scheduler notifies the resource allocator, the resource allocator releases any resources held by that process, the scheduler then reports completion of the process to the command processor, the command processor informs the user.

(ii) Context Switch: Context switch is a task of switching the CPU to another process by saving the state of old process and loading the saved state for the new process.

Context-switch time is overhead; the system does no useful work while switching.

Time dependent on hardware support.

1 to 1000 microseconds

3. Attempt any four : **(5×4=20)**

(a) Define following terms

(i) Average waiting time,

(ii) Time Slice or Quantum,

(iii) Response Time,

(iv) Turn around Time,

(v) Cpu Utilization.

Ans. Scheduling Criteria : Scheduling algorithm favors one process to another.

User oriented criteria

Turnaround time : amount of time to execute a particular process.

The interval from the time of submission of a process to the time of completion.

The sum of periods spent in waiting to get into memory, waiting in the ready queue, executing on CPU and doing *I/O*

Response time : amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment)

Deadlines : When the process completion deadlines can be specified, the scheduling discipline should subordinate the goals to that of maximizing the percentage of deadlines met.

System oriented

Throughput - # of processes that complete their execution per time unit

CPU utilization - keep the CPU as busy as possible; percentage of time the processor is busy.

It may range from 0 to 100 percent.

In a real system, it should range from 40 percent (lightly loaded) to 90 percent (heavily loaded)

Waiting time: It is the sum of the periods spent waiting in the ready queue.

System oriented: other

Fairness : In the absence of guidance from user or other system supplied guidance, processes should be treated the same, and no process should suffer starvation.

Enforcing priorities : When processes are assigned priorities, the scheduling policy should favour higher priority processes.

Balancing resources : The scheduling policy should keep the resources of the system busy. Processes that underutilize the stressed resources should be favoured which involves long term scheduling

(b) What should be the selection criteria for scheduling algorithm?

Ans.

Maximize : CPU utilization
Throughput

Minimize : turnaround
time waiting
time response time

(c) Calculate turn around time and average waiting time for following set of processes, if these processer are scheduled using

(i) SJF

(ii) Priority (both preemptive)

Process	burst time	priority	arrival time
P1	7	1	0
P2	3	2	4
P3	9	3	7

Ans. Using Sjf

Avg waiting time =2

Avg turnaround time =25/3

Using priority (taking 1 as highest priority)

Avg waiting time =2

Avg turnaround time =25/3

(d) What is deadlock and its conditions?

Ans. A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused by another process in the set.

Deadlock can arise if four conditions hold simultaneously.

Mutual exclusion: Only one process at a time can use at least one resource.

Hold and wait: A process holding at least one resource is waiting to acquire additional resources held by other processes.

No preemption : Resources cannot be preempted; that is a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Circular wait : There exists a set *{P0, P1, .., P0}* of waiting processes such that *P0* is waiting for a resource that is held by *P1*, *P1* is waiting for a resource that is held by *P2 ..., Pn-1* is waiting for a resource that is held by *Pn*, and *P0* is waiting for a resource that is held by *P0*.

(e) How deadlock can be avoided?

Ans. To ensure that deadlocks never occur, the system can use either deadlock prevention and deadlock-avoidance schemes.

Deadlock prevention: Set of methods for ensuring that at least one of the necessary conditions cannot hold.

Deadlock avoidance : Requires that operating system be given in advance additional information concerning which resources a process will request and use during its life time.

Deadlock detection : Examines the state of the system to determine whether a deadlock has occurred or not. Alternatively, assume that deadlock would not occur. Resort to manual recovery when performance degrades due to deadlock.

(f) Explain the difference between busy waiting and blocking.

Ans. Busy wait often arises when critical sections are used to guard the consistency of control information. It denies the CPU to lower priority process in the system.

A file is said to employ blocking of records if a physical record consists of more than one logical record.

4. Attempt any two :

(a) Explain the difference between internal and external fragmentation. Which one occurs in paging system? Which one occurs in pure segmentation? Discuss various ways to remove fragmentation :

Ans. External Fragmentation-Total memory space exists to satisfy a request, but it is not contiguous.

Given N allocated blocks 0.5 blocks will be lost due to fragmentation.

Internal Fragmentation - Allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used.

Consider the hole of 18,464 bytes and process requires 18, 462 bytes.

If we allocate exactly the required block, we are left with a hole of 2 bytes.

The overhead to keep track of this hole will be substantially larger than the hole itself.

Solution: allocate very small hole as a part of the larger request.

Internal frag occurs in paging and both occurs in segmentation.

Solution to fragmentation :

Compaction

Paging Segmentation

Compaction : Reduce external ftagmentation by compaction

Shuffle memory contents to place all free memory together in one large block. Compaction is possible *only* if relocation is dynamic, and is done at execution time.

I/O problem

Latch job in memory while it is involved in I/O.

Do I/O only into OS buffers.

Compaction depends on cost.

Paging : Solution to external fragmentation Permit the logical address space of the process to be non–contiguous, allowing a process to be allocated physical memory whenever the later is available.

Paging is a solution.

Logical address space of a process can be noncontiguous; process is allocated physical memory whenever the latter is available.

Divide physical memory into fixed-sized blocks called **frames** (size is power of 2, between 512 bytes and 8192 bytes).

Divide logical memory into blocks of same size called **pages**.

Keep track of all free frames.

To run a program of size *n* pages, need to find *n* free frames and load program.

Set up a page table to translate logical to physical addresses.

(b) Explain the concept of virtual memory and how it is obtained by Demand Paging and Segmentation.

Ans. Virtual memory is a technique that allows the execution of processes that may not be completely in memory.

Programs are larger than main memory.

VM abstract main memory into an extremely large, uniform array of storage.

Separation of user logical memory from physical memory.

Only part of the program needs to be in memory for execution.

Logical address space can therefore be much larger than physical address space.

Allows address spaces to be shared by several processes.

Allows for more efficient process creation.

Frees the programmer from memory constraints.

Virtual memory can be implemented via:

Demand paging

Demand segmentation

We only cover demand paging.

For demand segmentation refer research papers.

Demand Paging : Paging system with swapping.

When we execute a process we swap into memory.

For demand paging, we use lazy swapper.

Never swaps a page into memory unless required.

Bring a page into memory only when it is needed.

Less I/O needed

Less memory needed

Faster response

More users

Page is needed => reference to it

invalid reference => abort

not-in-memory => bring to memory

(c) **Write short notes on the following :**

(i) Thrashing,

(ii) Cache memory,

(iii) Allocation of frame,

(iv) Dining-Philosopher problem.

Ans.Thrashing : If a process does not have "enough" pages, the page-fault rate is very high. This leads to:

low CPU utilization.

operating system thinks that it needs to increase the degree of multiprogramming. another process is added to the system.

Thrashing is High paging activity.

Thrashing = a process is spending more time in swapping pages in and out.

If the process does not have # of frames equivalent to # of active pages, it will very quickly page fault. Since all the pages are in active use it will page fault again.

Allocation of Frames : Each process needs **minimum** number of pages.

If there is a single process, entire available memory can be allocated.

Multi-programming puts two or more processes in memory at same time.

We must allocate minimum number of frames to each process.

Two major allocation schemes.

fixed allocation

priority allocation

Dining-Philosophers Problem : Five philosophers spend their lives thinking and eating.

They share a common circular table surrounded by five chairs.

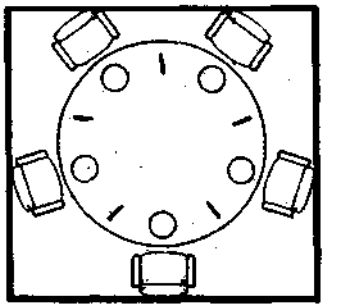
Five single chopsticks are available.

Whenever a philosopher wants to eat, he tries to pick up two chopsticks that are closest to him/her.

A philosopher cannot pick the chopstick in the hand of neighbor.

After finishing, the philosopher puts back the chopsticks and starts thinking.

It is simple representation of the need to allocate several resources among several processes in a deadlock and starvation fee manner



Cache memory : In computer science, a cache (pronounced koe □) is a collection of data duplicating original values stored elsewhere or computed earlier, where the original data is expensive to fetch (owing to longer access time) or to compute, compared to the cost of reading the cache. In other words, a cache is a temporary storage area where frequently accessed data can be stored for rapid access. Once the data is stored in the cache, it can be used in the future by accessing the cached copy rather than re-fetching or recomputing the original data.

A cache has proven to be extremely effective in many areas of computing because access patterns in typical computer applications have locality of reference. There are several kinds of locality, but this article primarily deals with data that are accessed close together in time (temporal locality). The data might or might not be located physically close to each other (spatial locality).

History : Use of the word *cache* in the computer context originated in 1967 during preparation of an article for publication in the IBM Systems Journal. The paper concerned an exciting memory improvement in Model 85, a latecomer in the IBM System/360 product line. The Journal editor, Lyle R. Johnson, pleaded for a more descriptive term than high-speed buffer. When none was forthcoming, he suggested the noun *cache*, from the French noun meaning a safekeeping or storage place [1]. The paper was published in early 1968, the authors were honoured by IBM, their work was widely welcomed and subsequently improved upon, and *cache* soon became standard usage in computer literature.[2]

Operation : A cache is a block of memory for temporary storage of data likely to be used again. The CPU and hard drive frequently use a cache, as do web browsers and web servers.

A cache is made up of a pool of entries. Each entry has a datum (a nugget of data) which is a copy of the datum in some backing store. Each entry also has a tag, which specifies the identity of the datum in the backing store of which the entry is a copy.

When the cache client (a CPU, web browser, operating system) wishes to access a datum presumably in the backing store, it first checks the cache. If an entry can be found with a tag matching that of the desired datum, the datum in the entry is used instead. This situation is known as a **cache hit**. So, for example, a web browser program might check its local cache on disk to see if it has a local copy of the contents of a web page at a particular URL. In this example, the URL is the tag, and the contents of the web page is the datum. The percentage of accesses that result in cache hits is known as the **hit rate** or **hit ratio** of the cache.

The alternative situation, when the cache is consulted and found not to contain a datum with the desired tag, is known as a **cache miss**. The previously uncached datum fetched from the backing store during miss handling is usually copied into the cache, ready for the next access.

During a cache miss, the CPU usually ejects some other entry in order to make room for the previously uncached datum. The heuristic used to select the entry to eject is known as the replacement policy. One popular replacement policy, least recently used (LRU), replaces the least recently used entry (see cache algorithms). More efficient caches compute use frequency against the size of the stored contents, as well as the latencies and throughputs for both the cache and the backing store. While this works well for larger amounts of data, long latencies, and slow throughputs, such as experienced with a hard drive and the Internet, it's not efficient to use this for cached main memory (RAM).[citation needed]

When a datum is written to the cache, it must at some point be written to the backing store as well. The timing of this write is controlled by what is known as the **write policy**.

In a **write-through** cache, every write to the cache causes a synchronous write to the backing store.

Alternatively, in a write-back (or write-behind) cache, writes are not immediately mirrored to the store. Instead, the cache tracks which of its

locations have been written over (these locations are marked **dirty**). The data in these locations is written back to the backing store when those data are evicted from the cache, an effect referred to as a **lazy write**. For this reason, a read miss in a write-back cache (which requires a block to be replaced by another) will often require two memory accesses to service: one to retrieve the needed datum, and one to write replaced data from the cache to the store.

Data write-back may be triggered by other policies as well. The client may make many changes to a datum in the cache, and then explicitly notify the cache to write back the datum.

No-write allocation is a cache policy where only processor reads are cached, thus avoiding the need for write-back or write-through when the old value of the datum was absent from the cache prior to the write.

The data in the backing store may be changed by entities other than the cache, in which case the copy in the cache may become out-of-date or **stale**. Alternatively, when the client updates the data in the cache, copies of that data in other caches will become stale. Communication protocols between the cache managers which keep the data consistent are known as coherency protocols.

5. Attempt any four :

(a) Define following terms

(i) Seek time,

(ii) rotational latency,

(iii) file sharing

Ans. File Sharing : Sharing of files on multi-user systems is desirable.

Sharing may be done through a *protection* scheme.

On distributed systems, files may be shared across a network.

Network File System (NFS) is a common distributed file-sharing method.

Protection

File owner/creator should be able to control:
what can be done
by whom

Types of access
Read
Write
Execute
Append
Delete
List

Access Lists and Groups

Mode of access: read, write, execute
Three classes of users

	RWX	
a) **owner access** 7	⇒	1 1 1
	RWX	
b) **group access** 6	⇒	1 1 0
	RWX	
c) **public access** 1	⇒	0 0 1

Ask manager to create a group (unique name), say G, and add some users to the group. For a particular file (say *game)* or subdirectory, define an appropriate access.

Seek time is one of the three delays associated with reading or writing data on a computer's disk drive, and somewhat similar for CD or DVD drives. The others are rotational delay and transfer time, and their sum is access time. In order to read or write data in a particular place on the disk, the read/write head of the disk needs to be physically moved to the correct place. This process is known as *seeking*, and the time it takes for the head to move to the right place is the *seek time*. Seek time for a given disk varies depending on how far the head's destination is from its origin at the time of each read or write instruction; usually one discusses a disk's *average seek time*.

Solid State Disks (SSDs) have a negligible seek time in comparison, for data can be randomly accessed without waiting for moving parts to move.

Rotational delay is one of the three delays associated with reading or writing data on a computer's disk drive, and somewhat similar for CD or DVD drives. The others are seek time and

transfer time, and their sum is access time. The term applies to rotating storage devices (such as a hard disk or floppy disk drive, and to the older drum memory systems). The rotational delay is the time required for the addressed area of the disk (or drum) to rotate into a position where it is accessible by the read/write head.

Maximum rotational delay is the time it takes to do a full rotation (as the relevant part of the disk may have just passed the head when the request arrived). Most rotating storage devices rotate at a constant angular rate (constant number of revolutions per second). The maximum rotational delay is simply the reciprocal of the rotational speed (appropriately scaled). In 2001, 7200 revolutions per minute is typical for a hard disk drive; its maximum rotational delay will be 60/7200 s or about 8 ms.

Average rotational delay is also a useful concept it is half the maximum rotational delay.

(b) Explain indexed allocation method of disk.

Ans. Indexed Allocation

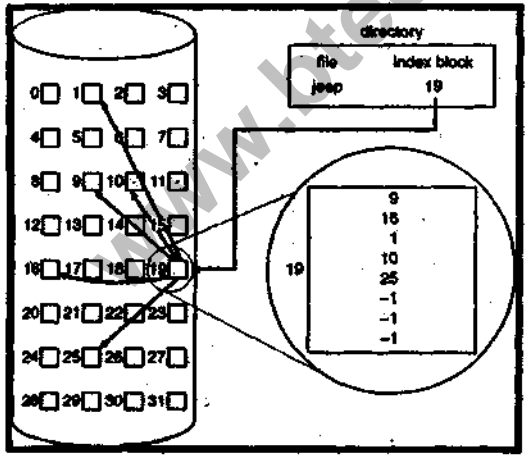
Brings all pointers together into the *index block*.

Logical view

Example of Indexed Allocation

Need index table

Random access



Dynamic access without external fragmentation, but have overhead of index block.

Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.

Mapping from logical to physical in a file of unbounded length (block size of 512 words).

Linked scheme - Link blocks of index table (no limit on size).

(c) What is DMA?

Ans. Direct memory access (DMA) is a feature of modern computers and microprocessors that allows certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the central processing unit. Many hardware systems use DMA including disk drive controllers, graphics cards, network cards and sound cards. DMA is also used for intra-chip data transfer in multi-core processors, especially in multiprocessor system-on-chips, where its processing element is equipped with a local memory (often called scratchpad memory) and DMA is used for transferring data between the local memory and the main memory.

Computers that have DMA channels can transfer data to and from devices with much less CPU overhead than computers without a DMA channel. Similarly a processing element inside a multi-core processor can transfer data to and from its local memory without occupying its processor time and allowing computation and data transfer concurrency.

Without DMA, using programmed input/output (PIO) mode for communication with peripheral devices, or load/store instructions in the case of multicore chips, the CPU is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU would initiate the transfer, do other operations while the transfer is in progress, and receive an interrupt from the DMA controller once the operation has been done. This is especially useful in real-time computing applications where

not stalling behind concurrent operations is critical. Another and related application area is various forms of stream processing where it is essential to have data processing and transfer in parallel, in order to achieve sufficient throughput.

Principle : DMA is an essential feature of all modern computers, as it allows devices to transfer data without subjecting the CPU to a heavy overhead. Otherwise, the CPU would have to copy each piece of data from the source to the destination, making it unavailable for other tasks. This situation is aggravated because access to I/O devices over a peripheral bus is generally slower than normal system RAM. With DMA, the CPU gets freed from this overhead and can do useful tasks during data transfer (though the CPU bus would be partly blocked by DMA). In the same way, a DMA engine in an embedded processor allows its processing element to issue a data transfer and carries on its own task while the data transfer is being performed.

A DMA transfer copies a block of memory from one device to another. While the CPU initiates the transfer by issuing a DMA command, it does not execute it. For so-called "third party" DMA, as is normally used with the ISA bus, the transfer is performed by a DMA controller which is typically part of the motherboard chipset. More advanced bus designs such as PCI typically use bus mastering DMA, where the device takes control of the bus and performs the transfer itself. In an embedded processor or multiprocessor system-on-chip, it is a DMA engine connected to the on-chip bus that actually administers the transfer of the data, in coordination with the flow control mechanisms of the on-chip bus.

A typical usage of DMA is copying a block of memory from system RAM to or from a buffer on the device. Such an operation does not stall the processor, which as a result can be scheduled to perform other tasks. DMA is essential to high performance embedded systems. It is also essential in providing so-called zero-copy implementations of peripheral device drivers as well as functionalities such as network packet routing, audio playback and streaming video. Multicore embedded processors (in the form of multiprocessor system-on-chip) often use one or more DMA engines in combination with scratchpad memories for both increased efficiency and lower power consumption. In computer clusters for high-performance computing, DMA among multiple computing nodes is often used under the name of remote DMA.

Cache coherency problem : DMA can lead to cache coherency problems. Imagine a CPU equipped with a cache and an external memory that can be accessed directly by devices using DMA. When the CPU accesses location X in the memory, the current value will be stored in the cache. Subsequent operations on X will update the cached copy of X, but not the external memory version of X. If the cache is not flushed to the memory before the next time a device tries to access X, the device will receive a stale value of X.

Similarly, if the cached copy of X is not invalidated when a device writes a new value to the memory, then the CPU will operate on a stale value of X.

This issue can be addressed in one of two ways in system design: Cache-coherent systems implement a method in hardware whereby external writes are signaled to the cache controller which then invalidates (for DMA reads) or flushes (for DMA writes) the cache lines in question. Non-coherent systems leave this to software, where the os must then ensure that the cache lines are flushed before an outgoing DMA transfer is started and invalidated before a memory range affected by an incoming DMA transfer is accessed. The os must make sure that the memory range is not accessed by any running threads in the meantime. The latter approach introduces some overhead to the DMA operation, as most hardware requires a loop to invalidate each cache line individually.

Hybrids also exist, where the secondary L2 cache is coherent while the L1 cache (typically on-CPU) is managed by software.

DMA engine : In addition to hardware interaction, DMA can also be used to offload expensive memory operations, such as large copies or scatter-gather operations, from the CPU to a dedicated DMA engine.

Examples

ISA

For example, a PC's ISA DMA controller is based on the Intel 8237 Multimode DMA controller, that is a software-hardware combination which either consists of or emulates this part. In the original IBM PC, there was only one DMA controller capable.

(d) What are the functions of a file system?

Ans. In computing, a **file system** (often also written as **filesystem**) is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data (e.g., procfs). It is distinguished from a directory service and registry.

More formally, a file system is a special-purpose database for the storage, organization, manipulation, and retrieval of data.

Most file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called *sectors*, generally a power of 2 in size (512 bytes or 1,2, or 4 KiB are most common). The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. Most file systems address data in fixed-sized units called "clusters" or "blocks" which contain a certain number of disk sectors (usually 1-64). This is the smallest amount of disk space that can be allocated to hold a file.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (e.g., procfs).

(e) Draw the file structure for UNIX operating system or Disk operating system.

Ans. Unix File System : The arrangement of disk blocks in Unix is as shown in the figure below.

BB : Boot Block IL : inode List
SB : Super Block DB : Data blocks

Fig. 1. Disk arrangement

The boot block contains the code to bootstrap the OS. The super block contains infonnation about the entire disk. The I-node lists a list of inodes, and the data blocks contains the actual data in the form of directories and files. Now let us dissect each block one by one.

The **super block** contains the following information, to keep track of the entire file system.

Size of the file system

Number of free blocks on the system

A list of free blocks

Index to next free block on the list

Size of the inode list

Number of free inodes

A list of free inodes

Index to next free inode on the list

Lock fields for free block and free inode lists

Flag to indicate modification of super block.

Size of the file system represents the actual no. of blocks (used + unused) present in the file system.

The super block contains an array of free disk **block numbers**, one of which points to the next

entry in the list. That entry in turn will be a data block, which contains an array of some other free blocks and a next pointer. When process requests for a block, it searches the free block list returns the available disk block from the array of free blocks in the super block. If the super block contains only one entry which is a pointer to a data block, which contains a list of other free blocks, all the entries from that block will be copied to the super block free list and returns that block to the process. Freeing of a block is reverse process of allocation. If the list of free blocks in superblock has enough space for the entry, then this block address will be marked in the list. If the list is full, all the entries from the super block will be copied to the freed block and mark an entry for this block in the super block. Now the list in super block contains only this entry.

Index indexes to the next free disk block in the free disk block list.

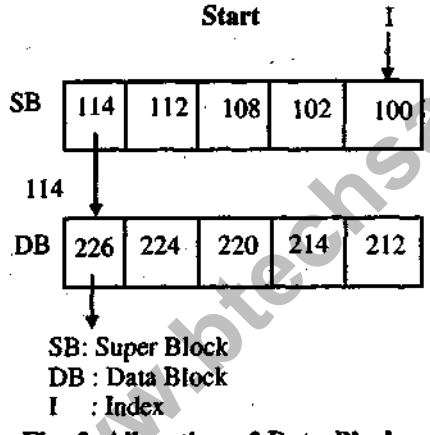


Fig. 2. Allocation of Data Blocks

The super block also contains an array to represent free inodes. The size of this array need not be the actual number of free inodes. During assignment of an inode to a new file, the kernel searches the tree inode list. If one tree inode is found, that one is returned. If the free inode list is empty, then it searches the inode list for free inodes. Each inode will contain a type field, which if 0, means the inode is tree. It then fills the free inode list of super block as much as it can, with number of tree inodes from inode list. It then returns one of these ones. It then remembers the highest inode number. Next time it has to scan the inode list for free ones, it starts from this remembered one. Hence it doesn't have to scan already scanned ones. This improves the efficiency. While freeing an inode, if the free list in super block has space enough, the freed one is put there. If there is not enough space, and the freed inode number is less than the remembered inode, then the remembered inode is updated with the freed inode. If the freed inode number is greater than the remembered one, then it doesn't have to update, because it will be scanned from the remembered node and the freed one will be covered later.

Index indexes to the next free inode in the free disk block list.

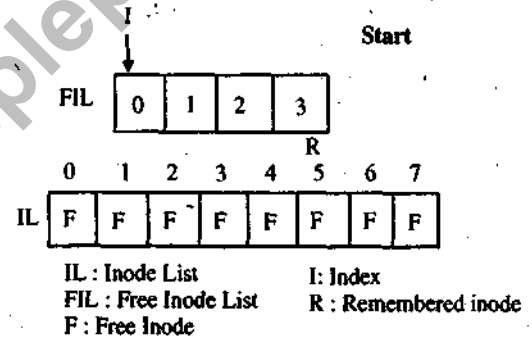


Fig. 3. Assignment of new inodes

During all these allocations, in a multi-tasking environment, there is a chance of the inodes getting corrupted. Like if one process was trying to allocate an inode and was preempted by the scheduler, and a second process does the same for same inode, it will be a critical problem. Hence lock flags are introduced. While accessing inodes, that inode will be locked. One more flag to indicate that the super block has been modified, is present in the super block.

The **I-node list** (which serves the purpose as FAT + directory entries in DOS) is a list of inodes, which contains the following entries.

Owner
Type
Last modified time
Last accessed time
Last inode modified time
Access Permissions
No. of links to the file
Size of the file
Data blocks owned

Owner indicates who owns the file(s) corresponding to this inode.

Type indicates whether inode represents a file, a directory, a FIFO, a character device or a block device. If the type value is 0, then the inode is free.

The times represent when the file has been modified, when it was last accessed, or when the inode has been modified last. Whenever the contents of the file is changed, the "inode modified time" also changes. Moreover it changes when there are changes for the inode like permission change, creating a link etc.

Each file will be having nine access permissions for read, write and execute, for the owner, group and others in rwx rwx rwx format.

In Unix we can create links to some files or directories. So we need to have a count of how many links are pointing to the same inode, so that if we delete one of the links the actual data is not gone.

Size of the file represents the actual size of the file.

In Unix, we have a kind of indexing to access the actual data blocks that contains data. We have an array of which (in each inode) first ten elements indicate direct indexing. The next entry is single indirect, then comes double indirect and then triple indirect. By direct indexing we mean that, the value in the array represents the actual data block. If the file needs more than 10 blocks, it uses single indirect indexing, means this is an index to the a block which contains an array of disk block numbers which in turn represent the actual disk block. If all these are exhausted, then double indirect indexing is used and then triple indirect. For further clarifications, refer figure.

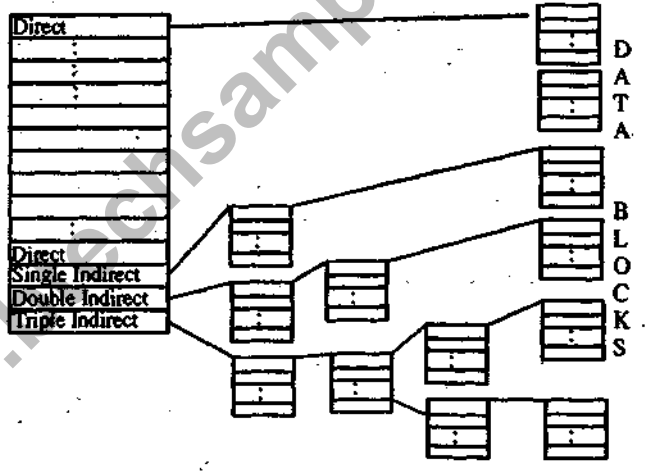


Fig. 4. Inode Data Block representation

The data blocks contain the actual data contained in the files or directories. In Unix, a directory is a special file. A directory file contains names of the subdirectories and files present in that directory and its corresponding inode number.

(f) List five system calls related to file system.

Ans. Open, close( ), read 0 , write ( ), rename () etc.