Chapter 1, Problem 1PE

(271)

Step-by-step solution

Show all steps

100% (57 ratings) for this solution

Step 1/4

Operating system:

It is an environment to interact with computer hardware and software in an easy and efficient way by providing convenient services to users.

Step 2/4

The three main purposes of an operating system are as follows:

Resource allocation:

• The first important purpose of an operating system is Resource allocation.

• If any application is terminated, then the resources are deallocated. Those deallocated resources may allocate to another program.

• Different kinds of available resources are allocated as per the need to perform the provided tasks in an efficient manner.

Step 3/4

Supervision:

• The data may be lost while accessing the resources. Hence, the operating system provides protection to the allocated resources in a different way.

• It supervises the execution of user programs to restrain the occurrence of errors and to avoid the misusage of computer and its resources.

Step 4/4

Managing input and output devices:

• The communication between the system and the user is handled by I/O devices. The user can access all the devices in uniform manner.

• The Operating system manages the functioning of the input and output devices.

Chapter 1, Problem 2PE

(17)
Step-by-step solution

Show all steps

100% (11 ratings) for this solution
Step 1/1

Computer Resources

It may be appropriate for operating systems to forsake the principle to make efficient use of the computer hardware and waste resources in case of **Single – User Systems**. In such systems, the use of computer resources must be maximized for the user.

Such a system is not really wasteful because although a graphical user interface may waste CPU cycles, but by maximizing the use of the system, it optimizes the user's interaction with the system.

Chapter 1, Problem 3PE

(5)
Step-by-step solution

Show all steps

90% (10 ratings) for this solution
Step 1/1

2481-1-17E SA: 8683

SR: 4578

A **real-time system** has well-defined, fixed time constraints. Processing must be done within the defined constraints, or the system will fail. For example, it would not do for a robot arm to be instructed to halt after it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. So, **the main difficulty** is keeping the operating system within the fixed time constraints of a real-time system. Therefore when writing an operating system for a real- time system, the writer must be sure that his scheduling schemes don't allow response time to exceed the time constraint.

Chapter 1, Problem 4PE

(5)

Step-by-step solution

Show all steps

100% (9 ratings) for this solution

Step 1/2

The application programs like mail programs and web browsers should not be added to the operating system because:-

• As per computer architecture, the computer system is categorised into four major components: application programs, operating system, hardware and user. The mail program and web browser belongs to application program, not to operating system.

• Operating system is defined as the program which is running all the time on the computer system. But the mail programs and the web browsers do not run all the time on the system. Hence they cannot be regarded as operating systems.

Step 2/2

The arguments in favour of the mail programs and web browser to be included in operating system are:

• Operating system is defined as the program which acts as the interface between the hardware and the user. As the mail program and the web browser are the part of program between hardware and the computer user, controlling and allocating the resources, thus they are part of operating system.

• Also, the operating systems generally include all those applications which are provided by vendors. As many vendors provide mail programs and web browsers along with operating systems, they are also included in operating systems.

Chapter 1, Problem 5PE

(15)

Step-by-step solution

Show all steps

100% (17 ratings) for this solution

Step 1/2

There is a distinction in execution between kernel mode and user mode of the operating system. This distinction in execution provides a rudimentary form of protection system.

Kernel mode:

• It is mandatory that certain instructions must be executed only when the operating system in kernel mode.

• Access to hardware devices can be done only when the program is executing in kernel mode.

• Interrupts can be controlled only when the CPU is executing in kernel mode.

• The privileged instructions can be executed only when the CPU is in kernel mode.

Step 2/2

User mode:

• When the CPU is executing in user mode, its capacity is limited.

• Executing privileged instructions in user mode will cause a trap to the operating system.

Thus, dual mode of operation provides a rudimentary form of protection system.

Chapter 1, Problem 6PE

(7)

Step-by-step solution

Show all steps

100% (21 ratings) for this solution

Step 1/10

The privileged instructions must execute in the kernel mode only. These instructions cannot execute in the user mode. The instruction which switches to the kernel mode is the privileged instructions.

Step 2/10

a.

The Set value of timer instruction is privileged. This instruction is interfaced with the kernel. If the timer goes off, then the task will stop, and the kernel starts a new task.

Step 3/10

b.

The Read the clock instruction is not privileged. This instruction is not interfaced with the kernel. The kernel does not read the clock value.

Step 4/10

c.

The Clear memory instruction is privileged. The kernel checks whether a process needs to clear the memory and if the process wants to clear the memory, then the kernel uses this instruction.

Step 5/10

d.

The Issue a trap instruction is not privileged. The trap instruction issued in the user mode but not in the kernel mode.

Step 6/10

e.

The Turn off interrupt's instruction is privileged. This instruction is interfaced with the kernel. The kernel provides the instruction to maintain the flow of control of the processes.

Step 7/10

f.

The Modify entries in device-status table instruction are privileged. These instructions are interfaced with the kernel mode. The instructions which must not modify are interfaced with the kernel mode.

Step 8/10

g.

The Switch from user to kernel mode instructions is privileged instruction. The system must transition from user to kernel model to accomplish the user request. At the time of system start the hardware starts in the kernel mode and then run the user applications in the user mode.

Step 9/10

h.

Access I/O device instructions are privileged. These instructions execute in the kernel mode. The kernel mode protects these instructions against the attempts by the user to modify these instructions. The instructions which must not modify are interfaced with the kernel mode.

Step 10/10

Privileged instructions and Un-privileged instructions are tabulated as below:

S.no	Instruction	Type
a.	Set the Value of timer	Privileged
b.	Read the clock	Un-Privileged
c.	Clear memory	Privileged
d.	Issue a trap instruction	Un-Privileged
e.	Turn off interrupts	Privileged
f.	Modify entries in device-status table	Privileged
g.	Switch from user to kernel mode	Privileged
h.	Access I/O device	Privileged

Chapter 1, Problem 7PE

(4)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/1

Operating System

By placing the operating system in a memory partition that could not be modified by either the user job or the operating system itself, the difficulties that can arise are listed below:

1) The critical data such as passwords and access control information that are required by or generated by the operating system would have to be passed through or stored in unprotected memory slots and would be accessible to unauthorized users.

2) The operating system can never be updated or patched, since it is not modifiable or accessible by the user or the operating system itself.

Chapter 1, Problem 8PE

(7)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/3

2481-1-20E SA: 8683

SR: 4578

Most of the CPUs provides two modes of operation.

1. Kernel mode

2. User mode

Step 2/3

Some CPUs have supported **multiple modes** of operation by providing different distinctions in kernel mode and user mode separately, rather than distinguishing between just kernel and user mode.

Step 3/3

Two possible uses of these multiple modes are:

1. One possibility would be to provide different distinctions within kernel code. For example, a speci◊c mode could allow USB device drivers to run. This would mean that USB devices could be serviced without having to switch to kernel mode, there by essentially allowing USB device drivers to run in a quasi-user/kernel mode.

2. Second possibility would be to provide different distinctions within user mode. Multiple user modes could be used to provide a ◊ner-grained security policy. Perhaps users belonging to the same group could execute each other's code. The machine would go into a speci◊ed mode when one of these users was running code. When the machine was in this mode, a member of the group could run code belonging to anyone else in the group.

Chapter 1, Problem 9PE

(5)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution
Step 1/1

Computing Current Time

Timers can take the help of a program that uses the following steps to calculate the current time with the help of timer interrupts:

1. Sets the timer for some time later than the current time and goes to sleep.

2. An interrupt wakes up the program; that is when it updates its local state, which it uses to keep a record of the number of interrupts it has received by far.

3. Then it repeats this process of establishing timer interrupts and updating its local state when these timer interrupts are raised.

Chapter 1, Problem 10PE

(7)
Step-by-step solution

Show all steps

100% (13 ratings) for this solution
Step 1/1

If the active portions of program and data are placed in a fast small memory the average memory access time can be reduced, thus reducing the total execution time of program. Such a fast small memory is referred as cache memory.

Caches are useful when two or more components need to exchange data, and the components perform transfers at different speeds. Caches solve the transfer problem by providing a buffer of intermediate speed between the components. If the fast device finds the data it needs in the cache, it needs not wait for the slower device. The data in the cache must be kept consistent with the data in the components. If a component has a data value change, and the datum is also in the cache, the cache must also be updated. This is especially a problem on multiprocessor systems where more than one process may be accessing a datum. A component may be eliminated by an equal-sized cache, but only if: a) the cache and the component have equivalent state-saving capacity (that is, if the component retains its data when electricity is removed, the cache must retain data as well), and b) the cache is affordable, because faster storage tends to be more expensive.

Chapter 1, Problem 11PE

(3)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/2

The client-server model of the distributed system provides an interface between the client and the server. In the peer-to-peer model of the distributed system, every node in the network acts as either a client or a server.

Step 2/2

Differences between the client-server model and the peer-to-peer model are:

Client-Server model	**Peer-to-Peer model**
The server runs a database and provides services to the client.	Depends on the request of a service, every node in the network behaves either a server or a client.
In this, the server is a bottleneck. The services can be provided by only the server.	In this, the services can be provided by every node in the network.
The data is stored in a centralized system.	Every node in the network stores its data.
The implementation of the client-server system is easy.	If the number of peers in the network increases, this model implementation will become difficult.
A web server is an example of the client-server model.	It is used in searching files, transferring files, and emails.

Chapter 1, Problem 12E

(7)

Step-by-step solution

Show all steps

100% (11 ratings) for this solution

Step 1/3

300-1-1E

Step 2/3

(a)

In a multiprogramming and time-sharing environment, several users share the system simultaneously. This situation results in two problems. Stealing or copying a user's files; writing over another program's (belonging to another user or to the OS) area in memory; using system resources (CPU, disk space) without proper accounting; causing the printer to mix output by sending data while some other user's file is printing.

Step 3/3

(b)

Probably not, since any protection scheme devised by a human can also be broken -- and the more complex the scheme is, the more difficult it is to be confident of its correct implementation.

.

Chapter 1, Problem 13E

(5)

Step-by-step solution

Show all steps

100% (15 ratings) for this solution

Step 1/3

Resource Management

Resource utilization is a major issue that occurs in different forms in different types of operating systems. It is important to manage resources to perform tasks efficiently and in a fair manner.

a) The resources that must be managed in **Mainframe or Minicomputer Systems** are listed below:

1. **Memory Resources**: Main memory (RAM) is an important part of the mainframe systems that must be carefully managed, as it is shared amongst a large number of users.

2. **CPU Resources**: Again, due to being shared amongst a lot of users it is important to manage CPU resources in mainframes and minicomputer systems.

3. **Storage**: Storage is an important resource that requires to be managed due to being shared amongst multiple users.

4. **Network Bandwidth**: Sharing of data is a major activity in systems shared by multiple users. It is important to manage network bandwidth in such systems.

Step 2/3

b) The resources that must be managed in **Workstations Connected to Servers** are mentioned below:

1. **Memory Resources**: When workstations are connected to servers, multiple applications run on multiple workstations on a single server. This is an important factor due to which memory management is required in workstations connected to servers.

2. **CPU Resources**: Multiple workstations make requests to access resources to accomplish the tasks assigned to them. To ensure the fair and efficient completion of tasks, it is important to manage CPU resources in workstations connected to servers.

Step 3/3

c) The resources that need to be managed in **Handheld Computers** are listed below:

1. **Power Consumption**: Handheld computers use compact, portable and small batteries as a source of power. It is important to manage power consumption in such devices to be able to make their use efficient and easy.

2. **Memory Resources:** Due to being small in size, the memory devices used in such computers are also small, thus deteriorating its storage capacity. This makes memory resource management an important requirement in handheld devices.

Chapter 1, Problem 14E

(2)
Step-by-step solution

Show all steps

100% (10 ratings) for this solution
Step 1/3

Time sharing system or multitasking is a logical extension of multiprogramming. In time-sharing, CPU executes multiple jobs by switching one after another process to execute. When switches occur frequently, then the user can interact with each

program while it is running. But, multitasking and multiprogramming is not possible on single user work station.

Step 2/3

The time sharing is best under following circumstances:

• In a work station, if there are few users to complete the large task, and hardware is fast, then at the point of time timesharing system is useful to complete the task with in the stipulated time.

• The other way to use the time sharing is when lots of users need so many resources at the same time.

The Personal Computer is best under following circumstances:

• Personal computer is best, when the job is small enough to be executed with in short amount of time.

Step 3/3

The time-sharing system is better than PC under following circumstances:

• A time-sharing operating system uses CPU scheduling and multiprogramming to provide each user with a small portion of a time-shared computer.

• If other users need to access the same system, then a time-sharing system would work better than a PC or a single-user workstation.

• If there is only a single and exclusive user, then a PC or single-user workstation would be better to use the less amount of resources.

• To work remotely on the system, then time sharing system is useful then PC.

• Time sharing system is better than PC when compared to cost, and ease of use.

Chapter 1, Problem 15E

(8)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/1

• **Symmetric processing** treats all processors as equals; I/O can be processed on any of them. **Asymmetric processing** designates one CPU as the master, which is

the only one capable of performing I/O; the master distributes computational work among the other CPUs.

• **Advantages:** Multiprocessor systems can save money, by sharing power supplies, housings, and peripherals. Can execute programs more quickly and can have increased reliability.

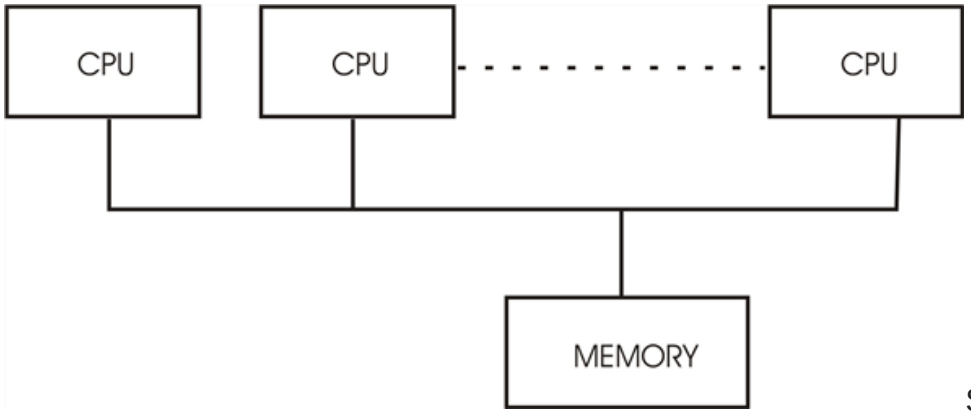
• **Disadvantages:** Multiprocessor systems are more complex in both hardware and software. Additional CPU cycles are required to manage the cooperation, so per-CPU efficiency goes down.

In detail…

The three advantages of multiprocessor systems are:-

i. Increased throughput: - By increasing the member of processor, we hope to get more work done in less time.

ii. Economy of scale: - Multiprocessor system can cost less than equivalent multiple slave-relationships exists between processors. Each processor concurrently runs a copy of the operating system.



SMP Architecture

Example of SMP is Solaris, a commercial version of Unix designed by Sun Micro system.

The difference between symmetric and asymmetric multi processing may result from either hardware or software. Single processor system because they can share peripherals, mass storage, and power supplies.

iii. Increased reliability: if functions can be distributed properly among several processors, then the failure of one processor will not halt the system, only slows it down.

The main disadvantage of multi process or system is that if the operation is continued in the presence of failures, requires a mechanism to allow the failure to be detected, diagnosed, and if possible, corrected. The tandem system uses both hardware and software duplication to insure continued operation despite faults. The

system consists of two identical processor, each with its own local memory. The processors are connected is the primary and the other is the backup. Two copies are kept of each process. One on the primary processor and the other on the primary processor

And the other on the back up. At fixed check points in the execution of the system the state information of each job:-

Including a copy of the memory image is copied from the primary machine to the backup. If a failure is detected, the backup copy is activated and is started from the most recent checkpoint. This solution is expensive, since it involves considerably hardware duplication.

Symmetric Multi procession (SMP) is one in which each processor runs a copy of the operating system and these copies communicate with one another as needed.

Asymmetric multi processing is one in which each processor is assigned a specific task. A master processor controls the system, the other processor either look to the master for instruction or have pre-defined tasks. This scheme defines a master-slave relationship. The master processor schedules to allocates work to the slave processors. SMP (symmetric multi processor) means that all processors are peers; no master

Chapter 1, Problem 16E

(1)

Step-by-step solution

Show all steps

89% (9 ratings) for this solution

Step 1/1

Clustered Systems

It is made up of together of multiple CPUs to accomplish computational work. Clustered systems differ from multiprocessor systems, however, in that they are composed of two or more individual systems compiled together. It shares storage and are closely linked via LAN networking.

It provides high- availability service. i.e. service will continue to be provided even if one or more systems in the cluster fails. High- availability is generally obtained by adding a level of redundancy in the system. A layer of cluster software runs on the cluster nodes. Each node can monitor one or more of the others (over LAN). If the monitored machine fails, the monitoring machine can take ownership of its storage and restart the application that were running on failed machine.

Chapter 1, Problem 17E

(2)
Step-by-step solution

Show all steps

Step 1/1

Cluster Systems gather together multiple CPUs to accomplish computational work.

cluster system differs from parallel system. The cluster computer share storage and are closely linked via LAN networking.

Clustering provides high-availability service will continue to be provided even if one or more system in the cluster fail. Clustering can be structured symmetrically or asymmetrically. In asymmetric clustering, one machine is in hot-standby mode, while the other is running the applications. The host – standby host machine does nothing but monitor the active server. If that server fails, the hot – standby host becomes the active server.

In symmetric mode, two or more hosts are running applications, and they are monitoring each other. This mode is more efficient, as it uses all the available hardware. It requires more than one application be available to run.

Chapter 1, Problem 18E

(1)
Step-by-step solution

Show all steps

100% (5 ratings) for this solution
Step 1/2

Network computers versus personal computers

Personal computers (PC): PCs are general purpose computers; those that are independent and can be used by a single user. Sharing of resources, communication is not possible in PCs. They have all the necessary resources local to the machine and are efficient in processing all the requests locally.

Network Computers: Network computers are the computers that are connected to each other through a network. It is possible to share resources and communicate with other computers in the network. They have very less resources locally and

minimal operating system too. They rely on the server for all their required resources.

Step 2/2

Types of networks:

LANs and WANs are the two basic type of network. LANs enable computers distributed over a small geographical area to communicate and share their resources. Where as WAN allow computers distributed over a large geographical area to communicate.

Real time examples and advantages of using network computers:

• E-mail a type of communication. Passing messages from one system to another system is possible using network computers.

• Web based computing helps to share information and files to all the systems that are connected to the network.

• Using messenger applications, real time communication (voice/text) is possible between the computers that are connected through network.

• Hardware resources can be shared between the systems that are connected through network.

• Troubleshooting the problems of a system can be done using another system on network from a remote location.

Chapter 1, Problem 19E

(9)

Step-by-step solution

Show all steps

84% (18 ratings) for this solution

Step 1/3

Purpose of Interrupts:

Interrupts are generated by the hardware devices. In the case of happening of interrupts, there is no role of software. Interrupts are asynchronous or passive means if there is any instruction running then interrupt has to wait to happen.

Step 2/3

Difference between Trap and Interrupt:

Interrupt	Trap
It is a hardware signal.	It is a software interrupt.
It is caused by external events like graphic card, I/O port etc.	It is caused by software programs like divided by zero.
It will not repeat as it is caused by an external event.	It can be repeat as it is software program. It can be repeated according to the calling of that instruction.
They can be handled by jump statements.	They cannot be handled by jump statements.
Interrupts are asynchronous events as they are related to external events.	Traps are caused by current program instructions; thus they are called as synchronous events.

Step 3/3

Yes, traps can be generated by the user program intentionally.

Purpose of calling Traps:

Traps can be used to call intentionally to call Operating system routines or to catch arithmetic errors like divide by zero.

Chapter 1, Problem 20E

(8)

Step-by-step solution

Show all steps

100% (9 ratings) for this solution

Step 1/3

(a)

• Direct memory access (DMA) command block is written to main memory which contains pointers to the source and destination of the transfer.

• The central processing unit (CPU) writes the address of the command block to the DMA controller. It can initiate a DMA operation by writing values into special registers that can be independently accessed by the device.

• The device initiates the corresponding operation once it receives a command from the CPU. When the device is finished with its operation, it interrupts the CPU to indicate the completion of the operation.

• Here, all devices have special hardware controllers. Normally, an operating system (OS) has device drivers that communicate with the controllers. The device drivers have registers, counters and buffers to store arguments and results.

Step 2/3

(b)

• After the completion of the task the dive controller interrupts the CPU.

• The CPU initiates the transfer by supplying the interface with starting address and number of words needed to be transferred and then proceeds to execute other tasks when transfer is made.

Step 3/3

(c)

• CPU is allowed to execute the other programs when device controller performs transfer of entire block of data from buffer to memory.

• When the device controller sends the interrupt to the CPU, the current task of the CPU is suspended until the interrupt is finished.

• It can access data in its primary and secondary cache memory because there is no interference with the user task.

Chapter 1, Problem 21E

(6)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/3

Designing an operating system without the use of processor privilege levels is possible.

MS-DOS or CP/M (single tasking operating systems) or Microsoft's Windows 3.1 (a multitasking operating system), none of which use this capability are some of the examples of such design.

Step 2/3

• However, security for those designs will become difficult to achieve. All code must be verified by the operating system prior to being run or interpreted with extensive run-time checks when the design requires security. The vaporware operating system JOS is an example of that design.

• The user ID and group ID are sufficient for the users to use the system normally. Sometimes escalate privileges are needed to gain extra permission for some activities.

Step 3/3

• Privileged mode is not provided by few systems, but the data will be secure. This is possible in which the security is provided by operating system by maintaining a list of user names and associated user identifiers (UIDs).

• The ID's for each and every user will be unique. The authentication stage determines the appropriate user Id for the user when the user logged in to the system. User ID's are associated with all the name of the processes and threads. The ID is translated back to username using the user name list if an ID needs to be user readable.

Chapter 1, Problem 22E

(2)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/1

Caching system:

• Cache is the temporary memory which is used for fast access. When a particular piece of information is needed, it is firstly searched into the cache. It is used directly from the cache if it is present here otherwise this information is used form the memory by putting a copy of the information into the cache.

• The design of cache into different levels with local caches near each processing core and the one shared by all cores is due to access speed and cost. The access will be faster if the cache is near the processing core. But the fast caches are costly.

• Hence smaller and faster cache is placed local to each core and the shared cache which is larger but slower is shared among different processing cores.

Chapter 1, Problem 23E

(2)
Step-by-step solution

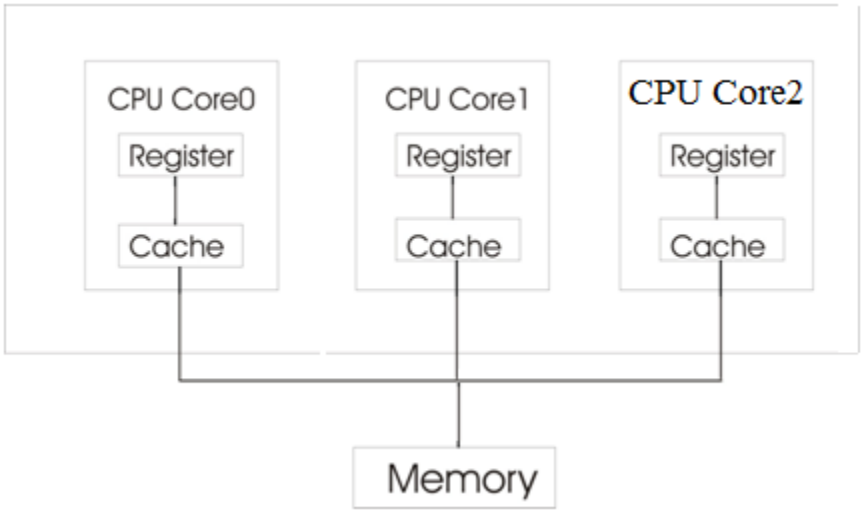
Show all steps

100% (6 ratings) for this solution
Step 1/4

Symmetric Multiprocessing (SMP):

In **Symmetric Multiprocessing (**SMP) system, each processor performs all tasks within the operating system. All the processors in SMP are peers; the relationship of master-slave does not exist between processors. Each processor has its own set of registers, as well as a private or local cache. These processors share physical memory. In SMP, N-processors can run N-processes simultaneously. The I/O should be controlled properly so that the data reaches appropriate processor.

Step 2/4



Symmetric multiprocessing architecture

Step 3/4

Let us consider an example how data residing in memory could in fact have two different values in each of the local caches. Consider a process ‘a’ gets the data from the main (physical) memory. The process is running in the CPU core0. It performs certain operation on the data and it is changed. Before the data is returned to the memory, another process ‘b’ gets the same data from the main memory. Again the process ‘b’ which is running in the CPU core1 performs some operations on the data. Initially the data taken from the physical memory is same. But the data after

reading from the memory is stored in the CPU's own cache and the operations are performed.

Step 4/4

The data in the local cache of CPU core0 is obtained after the operations of process 'a' are performed. Similarly, the data in the local cache of CPU core1 is obtained after the operations of process 'b' are performed. So, the data will be different. From this we can say that the data residing in the memory could in fact have two different values in each of the local caches.

Chapter 1, Problem 24E

(2)

Step-by-step solution

Show all steps

100% (9 ratings) for this solution

Step 1/1

(a)Single-processor system

In an environment where only one process executes at a time, arrangement of cache makes no difficulties. Where as in multi tasking environment it should be dealt properly.

(b)Multiprocessor system

Since the CPU switches back and forth among various process, cache should be taken special care. The situation becomes more complicated in a multiprocessing environment where in addition to maintaining internal registers, each of CPUs also contains a local cache. Hence may exist simultaneously in several caches. Since CPU (various) can execute concurrently, we must make sure that an update to the value 'a' in one cache is immediately reflected in all other cache where 'a' resides. This situation is called Cache Coherency. And is usually hardware problem.

(c) In distributed environment situation becomes even more complex. Several copies of same file can be kept on different systems that are distributed in space.

Since the various replicas may be accessed and updated concurrently, some distributed system ensures that when a replica is updated in one place, all another replicas are brought up to date as soon as possible.

Chapter 1, Problem 25E

(5)
Step-by-step solution

Show all steps

100% (8 ratings) for this solution
Step 1/1

Memory protection:

A program can modify the memory allocated to another program if it requires more memory. This process can cause problems in memory management; this can be avoided by enforcing memory protection mechanism.

• Memory for a program should be allocated as a chunk which is of fixed size.

• If the program requires more memory for execution, allocate extra chunk of free memory.

• The system should keep track of fixed memory chunks allocated to a program, and it should not allow a program to access memory outside of its allocation.

• Simpler method is to keep track of all chunks of memory allocated to programs and during access to the memory chunks its bounds should be checked.

Chapter 1, Problem 26E

(1)
Step-by-step solution

Show all steps

100% (4 ratings) for this solution
Step 1/3

Network configurations for different environments are as follow:

A campus student union

LAN is defined as a computer network which helps in interconnecting computers in a limited geographical area such as school, home, laboratory or office building.

Local area network is used by the campus student union if all the students of the union live in a short distance because it covers a small geographical area.

Step 2/3

Several campus locations across a statewide university system

WAN is defined as the computer network which helps in connecting computers in a large geographical area such as cities or countries.

Wide area network is used for connecting several campus locations across a statewide university stream as they are spread over a large geographical area.

Step 3/3

A neighborhood

Local area network or Wide area network may be used for connecting to the computers in neighborhood depending on the size of the neighborhood. LAN is suitable for smaller neighborhoods while WAN is suitable for larger ones.

Chapter 1, Problem 27E

(4)

Step-by-step solution

Show all steps

100% (4 ratings) for this solution

Step 1/1

In the mobile operating system there is a kernel and also a middleware. The middleware consist of set of software framework which helps in providing additional services to the application developers.

Challenges of designing operating system for mobile devices in comparison to Computer:

• In mobile operating system, besides designing a core kernel, a middleware also has to be designed for supporting set of software frameworks which helps in providing additional services to the application developers. But, in the computer middleware is not required.

• The mobile operating system has to balance its performance with the available battery life. Battery life in the mobile operating system is limited. But, in computer PC they need not to balance its performance with the available battery.

• The mobile operating system should have good support for peripheral such as HDMI or GPS. There are no such facilities of GPS and HDMI in mobile operating system.

• Owing to smaller size of mobile devices, limited resources, like memory and processors, are available. As a result, mobile operating system must manage these

resources carefully. In computer PC there is no issues in the management of the resources.

• The security issue of mobile operating system must also be taken care of as mobile devices are related with privacy of people.

Chapter 1, Problem 28E

(0)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/2

In order to know the advantages of Peer-to-Peer system in comparison to the client-server architecture user need to know the definition of peer-to-peer system and client-server model.

Peer to Peer system: In peer to peer system, client and server are not different to each other. All the nodes belonging to the system are considered as the peers. Client can act as server if it is providing services and the server can act as the client if it is requesting for the service.

Client server system: In client server system, client always requests for the service and the server always responds to the services made by the clients

Step 2/2

Advantages of Peer to Peer systems over client server system:

• Peer to peer system is easy to install and hence the configuration of this type of network is easy.

• Due to constant migration of peers, security is maintained which is not so in case of client server system as he location of server is known on the internet.

• Peer to peer systems facilitate higher bandwidth compared to client server systems because all the bandwidth of peers can be collectively used rather than a server's single bandwidth.

• All contents and the resources are shared by all the nodes unlike client server system in which all the contents and resources are shared by the server only. This minimizes burden on a single server.

• Central dependency which is the dependency on the server is eliminated by the peer to peer system which makes it more reliable than the client server system. If

one node fails then another node can act as server in peer to peer network but in client server network, if server goes down then whole system will be affected.

• Peer to peer systems can work even on basic operating system while specialized operating systems are needed by the client server system.

Chapter 1, Problem 29E

(1)

Step-by-step solution

Show all steps

Step 1/1

In peer to peer system, client and server are not different to each other. All the nodes belonging to the system are considered as the peers. Client can act as server if it is providing services and the server can act as the client if it is requesting for the service.

Following are some distributed application of peer to peer system:

• **File sharing:** Peer to peer substrate is used by the file sharing application for discovering that which peer has made the request.

As soon as the requesting peers are found, a connection between the supplier and the requester is established for providing and storing the files to the requesting nodes. Grokster, Gnutella, kazaa and Overnet are some examples of file sharing application.

• **File and storage system**: Functions which are similar to centralized file server are provided by the distributed file system for file and storage system. Replication, data integration and consistency, caching, directory structure and access control are some functions which are provided by the distributed file system.

Structured peer to peer substrate is used by the distributed file system for providing efficiency and guarantee of locating files. OceanStore and cooperative file system are examples of file and storage system Building the file system by peer to peer substrate will achieve:-

• Cost effectiveness

• High availability

• Huge storage capacity

• **Distributed cycle sharing**: In distributed cycle sharing applications, CPU processing power is the resource of interest. Pieces of large computational problems are processed independently by the peers in which enormous amount of CPU

processing is needed. The large computational problem has to be divided into smaller pieces for facilitating distributed processing.

• **Intellectual property law and illegal sharing:** Intermediate server is not used in peer to peer networking for the data transfer purpose. In peer to peer network, the file sharing is legal until the developer has no authority to prevent sharing of copyright material.

Chapter 1, Problem 30E

(5)

Step-by-step solution

Show all steps

100% (2 ratings) for this solution

Step 1/2

Advantages of open-source operating system:

1. There are many benefits to open-source operating systems, including a community of interested programmers who contribute to the code by helping to debug it, analyze it, provide support, and suggest changes.

2. Open-source code is more secure than closed-source code because many more eyes are viewing the code.

3. Open –source code has bugs, but open-source advocates argue that bugs tend to be found and fixed faster owing to the number of people using viewing the code.

4. Companies that earn revenue from selling their programs tend to be hesitant to open-source their code.

Step 2/2

Disadvantages of open-source operating system:

1. The disadvantage of open-source operating system is software incompatibility means that the open-source operating system may not support all the software present in the computer.

2. Open-source operating system has certain limitations and will work with in this limitation.

3. Open-source operating system is also hardware incompatibility means that the open-source operating system may not support all the hardware devices present in the computer.

4. The open-Source operating system has limited hardware requirements because these systems are free.

Chapter 2, Problem 1PE

(15)

Step-by-step solution

Show all steps

100% (17 ratings) for this solution

Step 1/1

System Calls

A system call is a **program** that describes how a program requests a service from **kernel** of operating system. This may include services like accessing hard disk, executing and creating new processes and defines communication with kernel services like scheduling. These calls provide an interface between operating system and a process.

Purpose of system calls are given as follows:

• **Basic purpose:** Calls provide basic functionality to users to operate the operating system.

• **Process control:** System calls loads, execute and create processes and terminate when the user's task is finished with the process.

• **File management:** It provides file management such as creating a file, deleting it, open, close and save it. It also provides read, write and reposition functionalities.

• **Device management:** All hard disks are managed by system calls such as requesting for a device, releasing the device, reading and writing the device.

• **Information maintenance:** System calls helps in making information maintenance such as get/set time or date, get/set data of system, process, files or attributes of device.

• **Communication between processes:** System calls are useful for communication purpose as they help in creating and deleting communications, sending or receiving messages. They help in attaching or detaching remote devices and in transfer of status information.

Chapter 2, Problem 2PE

(5)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/1

Operating System Activities

An operating system is a collection of various softwares that helps in managing the resources of computer hardware and provides common services for all computer programs.

Five activities of operating system with regard to the process management are given as follows:

1. It helps in protecting the processes from **deadlocks**.

2. It helps in providing mechanisms for **communication** between processes.

3. It provides process **synchronization** for multiple processes.

4. It provides **resumption** and **suspension** of processes.

5. It **creates** and **deletes** processes of both user processes and system processes.

Chapter 2, Problem 3PE

(1)
Step-by-step solution

Show all steps

100% (3 ratings) for this solution
Step 1/1

2481-2-2E SA: 8683

SR: 4578

_____

The three major activities of an operating system in connection with regard to memory management are:

1. Keep track of which parts of memory are currently being used and by whom.

2. Decide which processes are to be loaded into memory when memory space becomes

available.

3. Allocate and de allocate memory space as needed.

Chapter 2, Problem 4PE

(1)
Step-by-step solution

Show all steps

100% (3 ratings) for this solution

Step 1/1

OS with Secondary Storage System

A general system has several layers of storage such as primary, secondary and cache storage. Any program to be executed needs data available in primary memory or in cache storage as main memory can't keep all the data present at a time since it is small in size. In case if the power is lost then computer system must provide secondary storage for backup. Secondary storage is made up of tapes, disks and other things to hold the information that will be accessed in primary memory when needed. The memory is generally divided into fixed number of bytes in which information is stored. Every information or data inside memory has its own address and the set of addresses available to execute a program is called an address space.

Three important activities of operating system with regard to the secondary memory is given as follows:

1. It manages the **free space** available on the secondary storage media.

2. Whenever a **new file** has to be written, it provides the storage space for it.

3. It **schedules** the various requests for memory accesses.

Chapter 2, Problem 5PE

(2)
Step-by-step solution

Show all steps

100% (8 ratings) for this solution
Step 1/1

The main function of command interpreter is to get and execute the next user – specified command. It reads commands from the user or from a file of commands and executes them, usually by turning them into one or more system calls. It is usually not part of the kernel since the command interpreter is more subject to changes. The command interpreter allows a user to create and manage processes and also determine ways by which they communicate (such as through pipes and files). As all of this functionality could be accessed by a user-level program using the system calls, it should be possible for the user to develop a new command-line interpreter.

Chapter 2, Problem 6PE

(1)
Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/1

The fork() system call creates a new process. The new process will have the same address space of the process that executed the fork().

After execution of fork(), exec() has to be called by one of the two processes. The exec() system call loads the new program into memory.

Chapter 2, Problem 7PE

(1)
Step-by-step solution

Show all steps

100% (7 ratings) for this solution
Step 1/1

Purpose of System Programs:

• System programs are also known as system utilities.

• The system programs provide a convenient environment for development and execution of programs.

• A system program is a collection of many system calls and they provide basic functionality to users so that they can operate the system easily.

• These programs allow user level processes to use the services of operating system.

Chapter 2, Problem 8PE

(3)
Step-by-step solution

Show all steps

100% (8 ratings) for this solution
Step 1/3

2481-2-10E SA: 8683

SR: 4478

The **layered approach**, in which the operating system is broken up into a number of layers (or levels), each built on top of lower layers. The bottom layer (layer 0) is the hardware and the highest (layer N) is the user interface.

Fig. An operating-system layered approach.

Step 2/3

The main advantage of the layered approach is **simplicity of construction and debugging**.

As in all cases of modular design, designing an operating system in a modular way has several advantages. The layers are selected such that each uses functions (or operations) and services of only lower-level layers. So, this approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error confined to that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified when the system is broken down into layers.

Step 3/3

Some of the disadvantages or difficulties involved with layered approach are:

1. The major difficulty with the layered approach involves the careful definition of the layers, because a layer can use only those layers below it. For example, the device driver for the disk space used by virtual-memory algorithms must be at a level lower than that of memory management routines, because memory management requires the ability to use the disk space.

2. Another problem with layered implementations is that they tend to be less efficient than other types. For instance, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls memory management layer, which in turn calls the CPU-scheduling layer, which is then passed to the hardware. At each layer, the parameters may be modified; data may need to be passed, and so on. Each layer adds overhead to the system call, results in layered system takes longer time to execute system call than the non layered system.

Chapter 2, Problem 9PE

(6)

Step-by-step solution

Show all steps

100% (12 ratings) for this solution

Step 1/4

Five services provided by the operating system are given as follows:

1. Program execution

2. I/O operations

3. File-system manipulation

4. Communications

5. Error detection

Step 2/4

Program Execution:

• It allows the user to execute programs by providing convenient environment for development and execution of programs.

• The operating system handles memory allocation, multitasking etc.

• A user level program cannot properly allocate CPU time.

I/O Operations:

• Every program may require some input/output such as a file or an I/O device.

• The operating system provides an environment to handle I/O operations.

• A user level program cannot control the I/O devices directly. For some I/O devices, special functions are necessary.

Step 3/4

File-system manipulation:

• All tasks related to files such as creating a file, deleting a file, reading a file, writing to a file etc. are handled by the operating system.

• A user need not have to know the details of secondary storage system. All a user can see is that his task is accomplished.

• User made programs cannot be made to allocate free blocks when available and deallocate the blocks after deletion.

Communications:

• There are times when a process needs to communicate with other process. All this is taken care by operating system.

• Communication takes place in the form of data packets and they needs access to the network device but user level programs cannot provide that.

Step 4/4

Error detection:

• An operating system constantly monitors the system and checks for errors which can cause malfunctioning to the system.

• All the data before writing to the hard disk must be ensured that they are not corrupted and they are not modified when they were written to the media.

• All these errors can frequently occur on the system and there must be a global program to handle all such errors.

• It is difficult for a user level program to handle detection of errors efficiently.

Chapter 2, Problem 10PE

(3)

Step-by-step solution

Show all steps

86% (7 ratings) for this solution

Step 1/2

2481-2-21E SA: 8683

SR: 4478

Firmware: All forms of ROM (Read Only memory) are also known as **firmware**, since their characteristics fall somewhere between those of hardware and those of software.

Step 2/2

For certain devices, such as PDAs, cellular phones and game consoles, a disk with a ◊le system may not be available for the device. In this situation, the operating system must be stored in ◊rmware, because firmware usually contains all of the code necessary to boot the operating system. So, devices with small operating system and simple supporting hardware store their operating system in firmware rather than on disk.

Chapter 2, Problem 11PE

(2)

Step-by-step solution

Show all steps

86% (7 ratings) for this solution

Step 1/2

2481-2-18E SA: 8683

SR: 4478

Consider a system that would like to run both Windows 7 and three different distributions of Linux (e.g., Red Hat, OpenSuse, and Ubuntu). Each operating system will be stored on disk. During system boot-up, a special program (which we will call the boot manager) will determine which operating system to boot into. This means that rather initially booting to an operating system, the boot manager will ◊rst run during system startup. It is this boot manager that is responsible for determining which operating system to boot into. Typically boot managers must be stored at certain locations of the hard disk to be recognized during system startup. Boot managers often provide the user with a selection of operating systems to boot into; boot managers are also typically designed to boot into a default operating system if no choice is selected by the user.

Step 2/2

The bootstrap program can perform a variety of tasks. It will run diagnostics to determine the state of the machine. If the system passes the diagnostic, then the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Chapter 2, Problem 12E

(6)

Step-by-step solution

Show all steps

100% (15 ratings) for this solution

Step 1/4

The operating system provides a convenient environment for development and execution of programs.

The services and functions provided by an operating system are divided into two main categories.

• Services and functions for helping the user.

• Services and functions for efficient operation of the system and provide protection.

Step 2/4

Services and functions for helping the user:

• **User interface:** It provides command-line interface, batch interface and graphical-user interface so that the user can interact with the system.

• **Program Execution:** It allows the user to execute programs by providing convenient environment for development and execution of programs.

• **I/O Operations:** Every program may require some input/output such as a file or an I/O device. The operating system provides an environment to handle I/O operations.

• **File-system manipulation:** All tasks related to files such as creating a file, deleting a file, reading a file, writing to a file etc. are handled by the operating system.

• **Communications:** There are times when a process needs to communicate with other process. All this is taken care by operating system.

• **Error detection:** An operating system constantly monitors the system and checks for errors which can cause malfunctioning to the system.

Step 3/4

Services and functions for efficient operatio n of the system:

• **Resource allocation:** In multiuser environment or multitasking environment, operating system acts as a resource allocator. It allocates resources to multiple users or multiple jobs.

• **Accounting:** Operating system provides services to keep track of usage of resources by the users. It helps in accounting purpose.

• **Protection and security:** Operating system provides services for protection and security of the system.

Step 4/4

The main difference between the two categories is that one category of services is for the convenience of the user and another category of services is for the efficient execution of the system.

Chapter 2, Problem 13E

(6)

Step-by-step solution

Show all steps

100% (17 ratings) for this solution

Step 1/1

Three general methods are used to pass parameters to the OS.

1) Pass parameters in registers

2) Registers pass starting addresses of blocks of parameters

3) Parameters can be placed or pushed onto the stack by the program and popped off the stack by the operating system.

Chapter 2, Problem 14E

(0)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/2

Every program executes different sections of code when the interrupts are occurred and spent some time on their execution. The statistical profile of the amount of time required to execute the different sections of code can be obtained by using Periodic timer interrupts.

Periodic timer interrupts:

This is an interrupt records the value of the program counter for every occurrence of interrupt and record the time spent on the different sections or parts of the program.

• The statistical profile of a program which is active must be consistent with the time spent by the sections of code in the program because the section must return to original program after completing the execution.

Step 2/2

Importance to obtain the statistical profile:

The programmer can determine how much time spent by executing the different sections code, since programmer can optimize the time by optimizing the utilization of resources by obtaining the statistical profile.

Chapter 2, Problem 15E

(3)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/1

The five major activities of an operating system with regard to file management are :

• Creation and deletion of files.

• Creation and deletion of directories.

• Supporting primitives for manipulating files and directories.

• Mapping the files onto secondary storage.

• Backing up files on non volatile storage media

Chapter 2, Problem 16E

(2)

Step-by-step solution

Show all steps

90% (10 ratings) for this solution

Step 1/3

System calls:

• It provides interface to communicate with the kernel.

• It is a software interrupt, which ensure that the user program can obtain system privileged services through some instruction which will be executed by the operating system.

Step 2/3

The advantage of using the same interface are as follows:

• The same set of system call can be used for device and file because once the device has been requested, we can read write and reposition the device, just as we can with files.

Step 3/3

The Disadvantage of using the same interface are as follows:

• The potential for device contention and perhaps leads to dead lock.

Chapter 2, Problem 17E

(2)

Step-by-step solution

Show all steps

Step 1/1

2481-2-6E SA: 8683

SR: 4578

_____

It would be possible for the user to develop a new command interpreter using the system call interface provided by the operating system because that is how they are made. A command interpreter is simply program that forward commands and arguments to the necessary programs, or make the necessary system calls directly

The command interpreter reads commands from the user or from a ◊le of commands and executes them, usually by turning them into one or more system calls.

For example, if we want to delete a file using the UNIX command

rm file.txt

First it would search for a file called ***rm***, load the file into memory, and execute it with the parameter ***file.txt*.** The function associated with the ***rm*** command would be defined completely by the code in the file ***rm***. While running this command, we may go across so many system calls. It may find that there is no file with the given name or that file is protected against access. In this case the program should print a message on the console (sequence of system calls) and then terminate abnormally (another system call). If input file exists then we must delete the file (another system call). It means executing command in command interpreter involves accessing sequence of system calls. So, to develop a new command interpreter by the user, system calls should be available to the user level programs. The system call interface serves as the link to system calls made available by the operating system. In this way user should be able to develop a new command interpreter using the system call interface.

Chapter 2, Problem 18E

(7)

Step-by-step solution

Show all steps

100% (13 ratings) for this solution

Step 1/1

There are two models of inter process communication are:

i. **Message – passing model:** In this, the communicating process exchange messages with one anther to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mail box. Message passing is useful for exchanging smaller amounts of data, because no

conflicts need be avoided. It is also easier to implement than is shared memory for inter computer communication. But the main disadvantage is it can handle only small amounts of data.

ii. **Shared – Memory model:** In this, processes use shared memory creates and shared memory attaches system calls to create and gain access to regions of memory owned by other processes. Two or more processes can exchange information by reading and writing data in the shared areas. Shared memory allows maximum speed and convenience of communication, since it can be done at memory speeds when it takes place within a computer .Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

Chapter 2, Problem 19E

(3)
Step-by-step solution

Show all steps

80% (5 ratings) for this solution
Step 1/1

Mechanism determines how to do something and policy means what will be done. The separation of policy and mechanism is very important for flexibility. Policies are likely to change across places or over time. A general mechanism insensitive to changes in policy would be more desirable. If the mechanism is properly separated from policy ,it can be used to support a policy decision that I/O-intensive programs should have priority over CPU - intensive ones or to support the opposite policy.

Chapter 2, Problem 20E

(1)
Step-by-step solution

Show all steps

Step 1/1

The backing – store driver would normally be above CPU scheduler, because the driver may need to wait for I/O and CPU can be rescheduled during this time. However on large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU schedule.

When a user program executes an I/O operation, it executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory management layer, which in turn calls CPU –scheduling layer, which is then passed to hardware. At each layer parameter may be modified, data may need to be passed and so on. Each layer adds overhead to system call, the net result is a system call that takes longer than does one on a non layered system.

The layers are designed, providing most of the advantages of modularized code while avoiding difficult problem of layer definition and interaction

Chapter 2, Problem 21E

(6)

Step-by-step solution

Show all steps

100% (8 ratings) for this solution

Step 1/4

Microkernel:

It is an approach in operating system to limit the usage of kernel space. It can be done by removing the nonessential components from kernel and implement on the user space. Therefore kernel space is smaller than the user space.

Step 2/4

Advantages:

1. Microkernel provides efficient communication between the client programs and different services running on the user space.

2. Easy to extend the operating system. Because, new services are added easily in user space than the kernel space.

3. Easy to port from one hardware design to another design because modifications required in the kernel are less as it has the small space.

4. Most of the services are running on the user space therefore it provides more security and reliability.

Step 3/4

Interaction in microkernels:

User programs and system services are communicated with message passing. Consider the case if a user program wants to access a file; it needs to interact with file server. Interaction between user programs and services like file servers, disks can’t be done directly. Therefore user programs uses message passing to communicate with the file servers.

Step 4/4

Disadvantages:

The performance of the microkernel is decreased as the overhead due to message passing increases. As the operating system communicates with message passing to interact with the user programs and system services, overhead is increases.

Chapter 2, Problem 22E

(4)

Step-by-step solution

Show all steps

100% (3 ratings) for this solution

Step 1/1

Loadable kernel module is defined as the source file which contains code for extending the running kernel. It is also called as base kernel. Loadable kernel modules are generally used for supporting new hardware and file system for adding system calls. Whenever the functionality provided by the loadable kernel module is not required, it can be unloaded for making resources and memory free.

Following are the advantages of the loadable kernel module:

• Whenever a new functionality is added or a bug is fixed, there is no need of rebuilding the whole kernel. Just compile the new functionalities.

• Memory can be saved because the operating system need not include the functionality which is already compiled in the base kernel.

• Loadable kernel module is more flexible than the layered system because any module can call any other module.

• Loadable kernel module is more efficient than the microkernel approach as there is no need of invoking message passing.

Chapter 2, Problem 23E

(1)

Step-by-step solution

Show all steps

100% (9 ratings) for this solution

Step 1/2

Similarities between iOS and Android:

• The information flow architecture is same in both the operating systems.

• Both are layered stack of software which helps in providing a very rich framework for developing mobile application.

• In both platforms, the applications are organized in hierarchical structure. The user gets access to an application in detail on traversing that path.

• They provide robust framework for developers.

Step 2/2

Differences between iOS and android:

Android	**iOS**
Android is open source	iOS is not open source
Android can be customized to a large extent due to openness in design specifications.	iOs has a very limited customizability due to more defined design specifications.
Programmed using C, C++ and java.	Programmed usingC and C++ besides objective C.
The media transfer depends upon the model	The media transfer is done by desktop applications
Android uses virtual machine for running applications.	iOSruns the program code on native machine only.

Chapter 2, Problem 24E

(1)

Step-by-step solution

Show all steps

Step 1/1

Reason why the Java program which is running on Android system does not use Java API and Virtual machine:

• The Java programs running on android systems do not use the standard java API and virtual machine because they are designed for desktop systems and server systems, not for the mobile devices.

• Dalvik virtual machine and set of library are included by the android runtime environment. The standard Java API is not used by the java environment but it uses android API and virtual machine for mobile devices for java development. This separate API is developed by Google.

• When a java class file is compiled, java byte code is produced which is platform independent and robust. The Dalvik virtual machine is responsible for converting the java byte code into the executable files.

Chapter 2, Problem 25E

(1)

Step-by-step solution

Show all steps

Step 1/1

The advantage of layered approach is modularity. This simplifies debugging and system verification. The design and system implementation is simplified. When the system is broken down into layers. A layer does not need to know how these operations are implemented, it know only what these operations do. Hence, each layer hides the existence of data structure, the operations and hardware from higher level layers.

The disadvantage of layered approaches defining the various layers in appropriate manner. Planning of layers is necessary because always the lower-level layer is used. Secondly, layered approach is less efficient than other method.

Chapter 2, Problem 26PP

(6)

Step-by-step solution

Show all steps

Step 1/1

Program to copy the contents of one file to a destination file:

```
return value
  ↓
          CreateFile(sourceFile)
BOOL      WriteFile    C (HANDLE         file,
                         LPVOID          buffer,
                         DWORD           bytes to write,
                         LPDWORD         bytes write,
                         LPOVERLAPPED    ovl);
          CreateFile(destinationFile)
          ReadFile(source file)
          WriteFile destinationFile((HANDLE       file,
                         LPVOID          buffer,
                         DWORD           bytes to write,
                         LPDWORD         bytes write,
                         LPOVERLAPPED    ovl);
          CloseFile(sourceFile)
          CloseFile(destinationFile)
```

A description of the parameters passed to WriteFile () function.

• HANDLE file – file to be written.

• LPVOID buffer – where file to be read into and write from.

• DWORD bytesToWrite - number of bytes written into buffer.

• LPDWORD bytesWrite - number of bytes written during last writing.

• LPOVERLAPPED ovl – If overlapped I/O is being used.

In windows, the system calls can be traced by using the tool dr strace.

Chapter 3, Problem 1PE

(20)

Step-by-step solution

Show all steps

100% (27 ratings) for this solution

Step 1/3

The fork()is a system call that create a new process. A system call split a process into two processes. These two processes are:

• Parent Process

• Child Process

Step 2/3

The memory pages which are used by original process are then duplicated after fork() system call. Therefore the Parent and Child processes will get the same image because they have same memory as original process.

The difference between both processes is when the call returns.

• If call returns in parent process, return value will be process id of child process.

• If returns in child process then return value will be ‘0’.

• If call fails and no new process created then return value will be -1.

If process created successfully then parent and child process will run the code from the same place where fork() call was running, due to duplicate memory. There may be two possibilities:

• The parent process ends before child process.

• The child process ends before parent process.

If wait() is called in some parent process then that process will be suspended until child ends.

Step 3/3

Hence, the output of this program will be 5 at Line A. The child process updates only its copy of value and after that control will be returned to parent process in which the value of value is 5 hence print statements will print the value as 5. Wait()is used in parent process hence firstly child will end then parents execute. Therefore final value will be from parent process.

Chapter 3, Problem 2PE

(17)

Step-by-step solution

Show all steps

100% (23 ratings) for this solution

Step 1/1

There are 8 processes created, because whenever a fork*()* function called, it creates processes as child nodes of a growing binary tree. If fork() function is called two times, it will create $2^2 = 4$ processes. All these 4 processes will be the child node of binary tree.

If fork() function is called unconditionally l times, we will have 2^l processes.

In this program fork() function is being called 3 times without any condition. So this program will generate total $2^3 = 8$ processes. There will be one parent process and seven child processes.

Chapter 3, Problem 3PE

(4)

Step-by-step solution

Show all steps

100% (11 ratings) for this solution

Step 1/2

Concurrent systems:

• Concurrent systems are those which support the concept of executing more than one applications or processes at the same time. Here the processes running can either be a duplicate of each other or simply two different processes in all.

• The main motive behind going for concurrency lies beneath reducing the overall execution time that may be required in executing a series of processes individually.

Step 2/2

Complications with Concurrency

Concurrency may reduce the overall processing time for some situations, but it has few of its complications as well. Three major complications that concurrency adds to an operating system are as follows:

• As multiple processes are concurrently running on the system, the operating system requires keeping track of all the storage space addresses on main memory to prevent one process from mixing with another or using the information stored for any other running process.

• Context switching between two simultaneous processes requires enough time to locate and maintain register values for programs running. A continuous communication between operating system and program control block may overload the system.

• Process that requires big data blocks for execution may enter deadlocks in wait of getting resources freed up by other processes.

Chapter 3, Problem 4PE

(5)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/1

2481-3-6E SA: 8683

SR: 4478

Context switch time pure overhead, because system does no useful work while switching. Context switch times are highly dependent on hardware support. In **Sun Ultra SPARC**, context switch simply requires the CPU current-register-set pointer is changed to the register set containing the new context, which takes very little time.

If the new context is in memory rather than register set and all the register sets are in use, then one of the contexts in a register set must be chosen and be moved to memory, and the new context must be loaded from memory into the set. This process takes a little more time than on systems with new context is already on one of the register set.

Chapter 3, Problem 5PE

(9)

Step-by-step solution

Show all steps

100% (20 ratings) for this solution

Step 1/2

Processes

When a process uses the *fork()* to create a new process, allthe new processes corresponding to the parent process are created and loaded into a separate memory location for the child process by the operating system. The parent process and the newly created child process only share the **shared memory segments**.

Therefore, the correct answer is **C) Shared memory segments** .

Step 2/2

Stacks and heaps are not shared by these processes. Instead, new copies of the stack and the heap are made for the newly created process, when a process tries to write into these. Therefore, option “**A**” and “**B**” are **incorrect**.

Chapter 3, Problem 6PE

(4)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

Step 1/2

2481-3-3E SA: 8683

SR: 4478

The “exactly once” semantics ensure that a remote procedure will be executed exactly once. For “exactly once”, we need to remove the risk that the server will never receive the request. To accomplish this, the server must implement the “at most once” protocol but must also acknowledge to the client that the RPC call was

received and executed. The client must resend each RPC call periodically until it receives the ACK for that call.

The general algorithm for ensuring this combines an acknowledgment (ACK) scheme combined with timestamps (or some other incremental counter that allows the server to distinguish between duplicate messages) is described as below.

The general strategy is for the client to send the RPC to the server along with a timestamp. The client will also start a timeout clock.

Step 2/2

The client will then wait for one of two occurrences:

(1) It will receive an ACK from the server indicating that the remote procedure was performed, or

(2) It will time out.

If the client times out, it simply assumes the server was unable to perform the remote procedure so the client invokes the RPC a second time, sending a later timestamp.

The client may not receive the ACK for one of two reasons:

(1) The original RPC was never received by the server, or

(2) The RPC was correctly received—and performed—by the server but the ACK was lost.

In situation (1), the use of ACKs allows the server ultimately to receive and perform the RPC. In situation (2), the server will receive a duplicate RPC and it will use the timestamp to identify it as a duplicate so as not to perform the RPC a second time. It is important to note that the server must send a second ACK back to the client to inform the client the RPC has been performed. **So, the server will perform the RPC operation “exactly once” by combining ACK and timestamps.**

Chapter 3, Problem 7PE

(1)

Step-by-step solution

Show all steps

100% (5 ratings) for this solution

Step 1/1

Distributed Systems

The server should keep a history of information and requests regarding the received RPC operations, a track of the successfully performed RPC operations, and the results affiliated to the operations, in stable storage like as a disk log.

In this manner, whenever a server crashes and an RPC message is received, the server can verify whether the RPC had previously been executed or not and therefore guarantee “exactly once” semantics for the execution of RPCs.

Chapter 3, Problem 8E

(9)

Step-by-step solution

Show all steps

100% (7 ratings) for this solution

Step 1/1

• **Short-term** (CPU scheduler)—selects from jobs in memory those jobs that are ready to execute and allocates the CPU to them.

• **Medium-term**—used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.

• **Long-term** (job scheduler)—determines which jobs are brought into memory for processing.

The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system and may wait a while for a job to finish before it admits another one.

Chapter 3, Problem 9E

(16)

Step-by-step solution

Show all steps

100% (10 ratings) for this solution

Step 1/1

The actions taken by a kernel to context-switch between processes are shown below:

• In the context-switching between one process to another, the kernel must save the state of the current task running on the system before switching to another task.

• It saves the registers values, memory address information, the program counter(PC) value, and the state of the current task in the Process Control Block(PCB). This method is referred as **state save**.

• The kernel performs the state save on the currently running task and loads the data information of the new task from the Process Control Block(PCB) to run.

• After the completion of the execution of the new task, the kernel switches to the old task by loading the data information of the old task from the Process Control Block(PCB) to run. This method is referred as **state restore**.

• The kernel would only save and load the CPU state while switching between the thread of the same process. It would save and load the process environment if the thread being interrupted and thread being scheduled to different processes.

Chapter 3, Problem 10E

(6)

Step-by-step solution

Show all steps

100% (6 ratings) for this solution

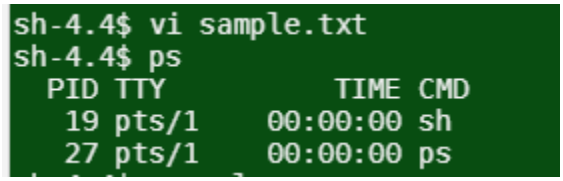
Step 1/5

ps command:

This command displays the process ID, information about a selection of the active processes and the list contains the processes to kill.

The screen shot of the ps command on the terminal window is as given below:

Create a sample text file the enter the command ps



Step 2/5

The screen shot of the ps –ael command on the terminal window is as given below:

Step 3/5

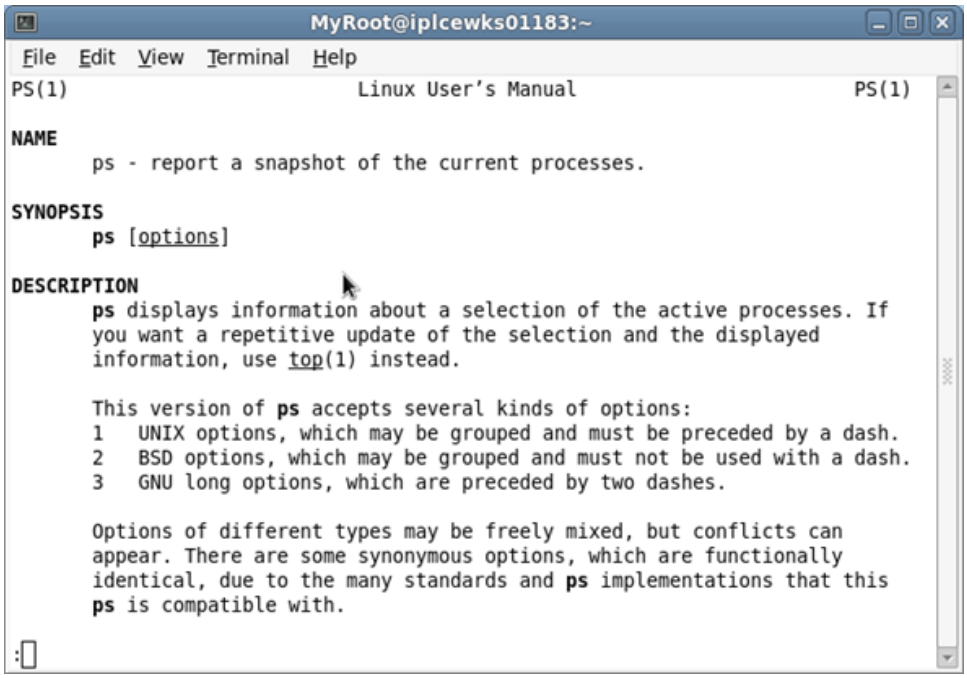
In the Unix terminal is opened by following the given path:

Applications -> System tools -> Terminal

The screen shot of the terminal window is as given below:

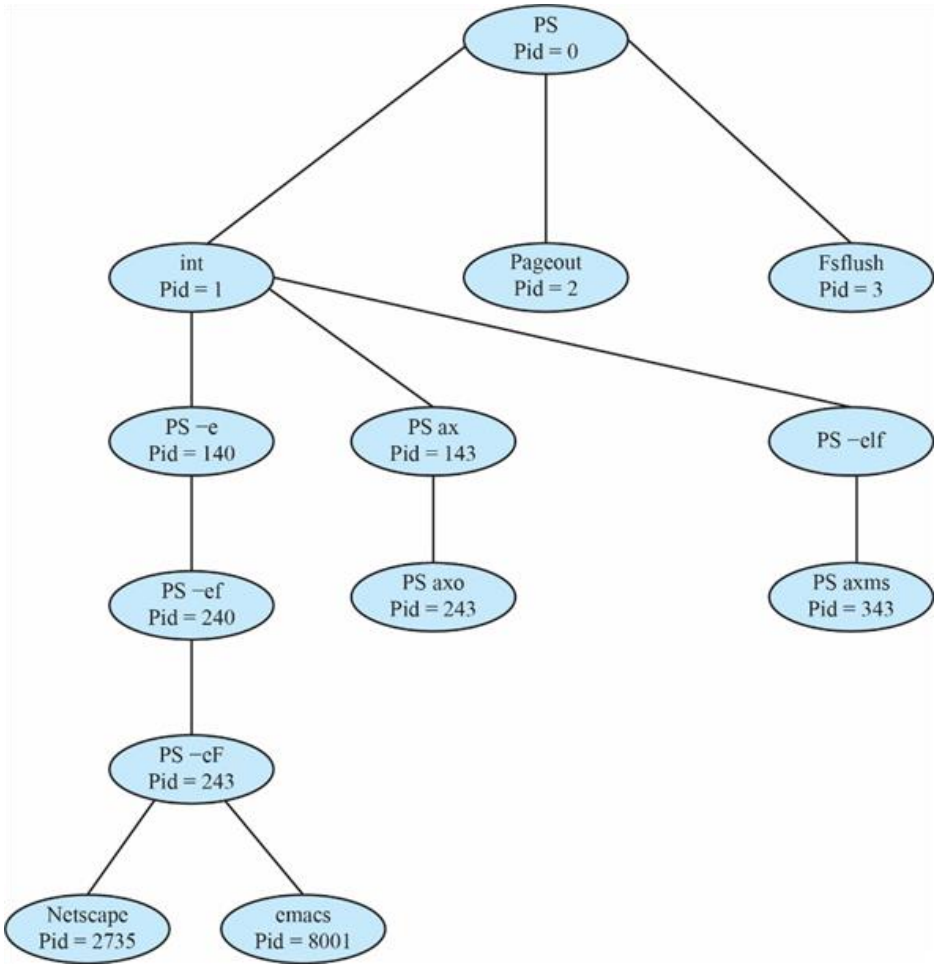
Step 4/5

ps command gives a snapshot of the current processes. The screen shot of the man ps command on the terminal window is as given below:



Step 5/5
Process Tree:

Process tree command shows the current running process ID's as a tree format. Here PS Pid with 0 is the root of the tree and remaining nodes are the child nodes. The process tree as follows:



Chapter 3, Problem 11E

(5)

Step-by-step solution

Show all steps

89% (9 ratings) for this solution

Step 1/1

Role of init process with respect to the process termination:

• The initial thing an init process does on initialization is reading the status of etc/init tab file. This file is the initialization file for this process and commands the system about several to-do instructions on a program.

• Besides this, init process has role in process termination too. If the parent process terminates before calling the wait(), it makes the child as orphans. This case is handled in UNIX and LINUX systems by assigning init process.

• This process periodically calls wait(), helping in collection of exit status of any orphaned child process and releasing its processID and process-table entries.

• It also commands terminated processes to restart with the help of “respawn” action.

Chapter 3, Problem 12E

(16)

Step-by-step solution

Show all steps

100% (32 ratings) for this solution

Step 1/3

The fork()is a system call that create a new process. A system call split a process into two processes. These two processes are:

• Parent Process

• Child Process

Step 2/3

To determine the number of processes created, take the code piece as below:

for(i=0; i<4; i++)

fork();

This piece of code is equivalent to four consequent fork() calls. Hence the code can be written as:-

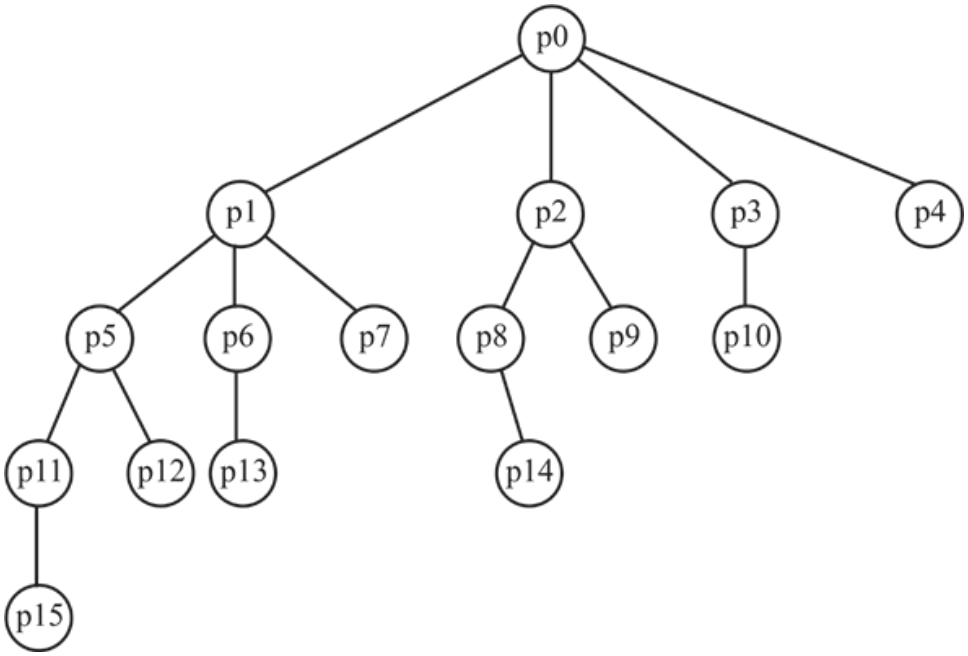
fork();

fork();

fork();

fork();

The diagram in figure 1 illustrates how the child processes are created with calls to fork in the program:-



Step 3/3

Explanation:

• The parent process is taken as “p0”. As there are 4 fork calls in the main program, 4 child processes, named p1, p2, p3 and p4 are created initially with call to each fork (). These child processes execute in similar and in parallel with parent process.

• The newly created child process p1 now encounters the bottom 3 of the four fork calls and create 3 child processes named p5, p6 and p7 with call to second, third and fourth fork () respectively.

• The child process p5 now encounters bottom two fork () calls and creates two new child processes called p11 and p12.

• The process p11 will encounter just one fork () call, that is, the bottom most and create a child p15.

• As p6 is created with call to third fork (), it will also encounter just one fork () call, that is, the bottom most and create a child p13.

• The process p12, created with last fork () call will not meet any new fork () and thus will not create any new child process. Similar will be the case for processes p15, p13 and p7.

• Similarly, the process p2, being created with call to second fork (), encounters bottom two fork () calls and creates two new child processes called p8 and p9.

• As p8 is created with call to third fork (), it will encounter just one fork () call, that is, the bottom most and create a child p14. Similarly, p3 will create p10.

• The processes p14 and p9, each created with last fork () call will not meet any new fork () and thus will not create any new child process. Similar will be the case for p4 and p10.

The whole procedure is depicted with the help of Figure 1.

Thus 15 new child processes will be created. Including the initial parent process, there are 16 processes created by the program.

Hence, in all 16 processes are created.

Chapter 3, Problem 13E

(12)

Step-by-step solution

Show all steps

100% (18 ratings) for this solution

Step 1/2

Consider the code provided in figure 3.33 of chapter 3. The explanation of the code is as follow:

• In main function, the first statement declare a variable pid of pid_t type to store the return value of fork() function.

• After that fork() function is called and store its return value in pid variable. The return value of fork() are as follow:

• If call returns in parent process, return value will be process id of child process.

• If returns in child process then return value will be ‘0’.

• If call fails and no new process created then return value will be -1.

• When value of variable pid becomes 0 then the condition statement else if (pid == 0)is true. Now all the statement inside this conditional statement get executes.

• After this execlp(“/bin/ls”,”ls”,NULL) function get executes. The argument of this function contains the list of other executable files, if file exist then that file execute otherwise returns a value -1 which represent the error.

• If name of the file provided in the execlp() function does not exist then the line Printf(“LINE J”) get executes.

Step 2/2

The above explanation gives a clear view that an error in executing execlp() function would result in achieving the print command for the program. The reasons for this

may be absence of “bin/ls” or it might have been corrupted or no permission for its access.

Chapter 3, Problem 14E

(20)

Step-by-step solution

Show all steps

100% (39 ratings) for this solution

Step 1/2

fork() system call:

The fork() function is used to create the new process.

Step 2/2

The fork() system call returns **the child’s actual PID** to the parent**, “0”** to the child.

Given actual pid of parent = 2600.

actual pid of child = 2603.

So, in the given program

• At line A – It will print “**child : pid = 0**” (value returned by fork to child)

• At line B – It will print “**child : pid1 = 2603**”.(actual pid of the child)

• At line C – It will print “**parent: pid = 2603**”. (value returned by fork to parent)

• At line D – It will print “**parent: pid1 = 2600**”. (actual pid of the parent).

Chapter 3, Problem 15E

(5)

Step-by-step solution

Show all steps

100% (11 ratings) for this solution

Step 1/2

2481-3-8E SA: 8683

SR: 4460

In the following situation ordinary pipes are more suitable than named pipes.

• If we want to establish communication between two specific processes on the same machine, then using ordinary pipes is more suitable than using named pipes because named pipes involve more overhead in this situation.

• In the situation, where we will not allow access to our pipe after the communication is finished between processes using ordinary pipes is more suitable than named pipes.

Step 2/2

In the following situations named pipes are more suitable than ordinary pipes.

• Named pipes are more suitable than ordinary pipes when the medium is bidirectional, and there is no parent child relationship between processes.

• Named pipes are more suitable than ordinary pipes in situations where we want to communicate over a network rather than communicating with the processes resides on the same machine.

• Named pipes can be used to listen to requests from other processes (similar to TCP/ IP ports). If the calling processes are aware of the name, they can send requests to this. Unnamed pipes cannot be used for this purpose.

Chapter 3, Problem 16E

(0)

Step-by-step solution

Show all steps

Step 1/2

When local procedure fails under extreme circumstances, RPC can fail, or be duplicated and executed more than once, as a result of network errors. This is said to be semantics of call.

• Let us consider “at most once” semantic in which this can be made possible by attaching a time stamp to each message.

• The server must keep a history of all the time stamp of messages it has already processed or ensure that repeated message it has already processed or ensure that repeated messages are detected.

Step 2/2

• Incoming message with time stamp will be ignored. The client will send a message more than one times and it will be executed only once.

• “Exactly once” will remove the risk that the server will never receive the request. In order to complete this, the server must implement the “at most once” protocol as well as the acknowledgement should be provided to client that RPC was received and executed.

• These “ACK” message are common throughout networking. The client must resend each RPC call periodically until it receives the “ACK” for that call.

• The concern for communication between a server and a client is of the form of binding takes place during link, load or the execution time, in which the name of the procedure call will be replaced by memory address of the procedure call.

Similar binding of the client and server port is required for the RPC scheme. RPC scheme may be useful.

Chapter 3, Problem 17E

(13)

Step-by-step solution

Show all steps

100% (22 ratings) for this solution

Step 1/2

In the program code there are two process loops; one named as child process and another one as parent process. The program typically runs the child process and after that execution is handed over to the parent process.

The child process provides output at “X” line, with the below written code patch:-

for (i = 0; i < SIZE; i++)

{

nums [i] *= -i;

printf (“CHILD: %d”, nums [i]);

}

As value of SIZE is 5, here each element of the array is multiplied respectively by numbers 0,-1,-2,-3 and -4.

$$0 \times 0 = 0$$
$$1 \times -1 = -1$$
$$2 \times -2 = -4$$
$$3 \times -3 = -9$$
$$4 \times -4 = -16$$

Hence, the output at Line X would be:

CHILD: 0

CHILD: -1

CHILD: -4

CHILD: -9

CHILD: -16

Step 2/2

The parent process provides output at “Y” line, with the below code patch:

wait (NULL);

for (i = 0; i < SIZE; i++)

printf (“PARENT: %d”, nums [i]);

As value of SIZE is 5, here every element of the array is printed. Hence, the output at Line Y would be:

PARENT: 0

PARENT: 1

PARENT: 2

PARENT: 3

PARENT: 4

Chapter 3, Problem 18E

(9)

Step-by-step solution

Show all steps

100% (14 ratings) for this solution

Step 1/5

a. Synchronous and asynchronous communication:

A benefit of symmetric communication is that it allows a rendezvous between the sender and receiver.
At the programmer level, neither process has to block its execution which can result in better performance.

Step 2/5

The disadvantage of asymmetric communication is that, blocking send is a rendezvous and may not be required and the message could be delivered asynchronously and received at a point of no interest to the sender. As a result, message-passing systems often provide both forms of synchronization.
And it is more difficult to program since the programmer must guarantee that the message arrive at the receiver when it is needed. At the system level, asymmetric is more complicated since it requires kernel-level buffering.

Step 3/5

b. **Automatic and explicit buffering:**

Automatic buffering provides a queue with indefinite length. Thus ensuring the sender will never have to block while waiting to copy a message. There are no specifications how automatic buffering will be provided. One schema may reserve sufficiently large memory where much of the memory is wasted.

Explicit buffering specifies how large the buffer is. In this situation, the sender may be blocked while waiting for available space in the queue. However, it is less likely that memory will be wasted in explicit buffering.

Step 4/5

c. **Send by copy and send by reference:**

Send by copy is better for network generalization and synchronization issues. It does not allow the receiver to alter the state of the parameter, but send by reference allows it. Send by reference is more efficient for big data structures but harder to code because it allows the programmer to write a distributed version of a centralized application (shared memory implications).

EX:
Java's RMI (Remote Method Invocation) provides passing a parameter by reference and requires declaring the parameter as a remote object as well.

Step 5/5

d. **Fixed-sized and variable-sized messages:**

The implications of this are mostly related to buffering issues with fixed-size messages (a buffer with a specific size can hold a known number of messages). Fixed-sized messages are easier to implement at the kernel-level but require slightly more effort on the part of the programmer.

Variable sized messages are somewhat more complex for the kernel but somewhat easier for the programmer. The number of variable-sized messages that can be held by such a buffer is unknown.

Example:
Windows 2000 handles this situation with fixed-sized messages (anything < 256 bytes), the messages are copied from the address space of the sender to the address space of the receiving process. Larger messages use shared memory to pass the message.

Chapter 3, Problem 19PP

(10)

Step-by-step solution

Show all steps

Step 1/6

Program Plan:

• Make a startup function main() to forks a child process.

• Call the fork function.

• If process id is grater then 0 than parent process will invoked.

• If process id 0 child process become zombie.

• If fork() did not work it will create error message.

Step 2/6

Program:

//Include the header files for performing the operations.

```
#include
#include
#include
#include
#include
```

Step 3/6

```
//The startup function
int main(int argc, char **argv)
{
// Fork off a process
pid_t pid = fork();
if (pid > 0)
{
// Parent process
printf("Pid of child %d\n", pid);
// Sleep for 10 seconds
```

sleep(10u);

childpid is storing the status of child process and executing child process. In else condition it is terminated by status.

// Wait for child processes

int status;

pid_t childpid = wait(&status);

if (WIFEXITED(status))

printf("Child %d exited with status %d\n",

childpid, WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("Child %d was terminated by signal

%d\n",

childpid, WTERMSIG(status));

return 0;

}

if (pid == 0)

// The child process -> do nothing just exit

// so we can become a zombie

return 0;

Step 4/6

Else condition will run if fork is unable to run. Else statement will give an error message and return -err.

Step 5/6

else

{

// Check if fork didn't work

int err = errno;

```
printf("fork blew %d\n", err);
return -err;
}
}
```

Step 6/6

Sample Output:

```
[abc@localhost 3]$ ./3-19PP &
[2] 5563
Pid of child 5566
[abc@localhost 3]$ ps -l
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
0 S 1000 5563 21589 0 80 0 - 1037 hrtime pts/2 00:00:00 3-19PP
1 Z 1000 5566 5563 0 80 0 - 0 exit pts/2 00:00:00 3-19
0 R 1000 5568 21589 0 80 0 - 30315 - pts/2 00:00:00 ps
0 S 1000 21589 2144 0 80 0 - 29118 wait pts/2 00:00:00 bash
0 S 1000 22210 21589 0 80 0 - 150788 poll_s pts/2 00:00:13 emacs
[abc@localhost 3]$ Child 5566 exited with status 0
!!
ps -l
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
0 R 1000 5572 21589 0 80 0 - 30307 - pts/2 00:00:00 ps
0 S 1000 21589 2144 0 80 0 - 29118 wait pts/2 00:00:00 bash
0 S 1000 22210 21589 0 80 0 - 150788 poll_s pts/2 00:00:13 emacs
[2]+ Done ./3-19PP
```

Chapter 3, Problem 20PP

(3)
Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/7

Program Plan:

• Make a header file for allocation and release of bitmap.

• Make header for allocate and release process id.

• Make a main function to allocate and de-allocate bitmap and process id.

• Allocate the map for the pids. It will return 1 if unable to allocate pids.

• Calculate the size of the bitmap required. Allocate the bitmap.

• Release the bitmap. Allocate a pid in a range. It assumes that bitmap is valid.

Step 2/7

Program:

```
//Include the header files for performing the operations.

#ifndef API_3_20PP_H

#define API_3_20PP_H

// Allocate the bitmap

extern int allocate_map(void);

// Release the bitmap

extern void release_map(void);

// Allocate a pid

extern int allocate_pid(void);

// Release a pid

extern void release_pid(int pid);

#endif
```

Step 3/7

//Include the header files for performing the operations.

#include

#include

#include "3-20PP.h"

Define the value of minimum pid, maximum pid and number of bits at position in the bitmap.

// Minimum value of pid

#define MIN_PID 300

// Maximum value of pid

#define MAX_PID 5000

// The number of bits at a position in the bitmap

#define NUM_BITS 8

Initialize the size. Index, offset and process id of bitmap. All variable are declared as static as there value is not changing throughout the program.

// The bitmap of pids

static unsigned char *s_bitmap = NULL;

// The size of the bitmap

static int s_size = -1;

// The current index into the bitmap

static int s_index = -1;

// The current offset into the index of the bitmap

static int s_offset = -1;

// The last assigned pid

static int s_pid = -1;

Allocate the map for process ids calculating the size of bitmap that is required.

//Allocate the map for the pids.

```
int allocate_map(void)
{
// Does the bitmap need to be allocated?
if (!s_bitmap)
{
// Calculate the size of the bitmap required
s_size = (int) ((double) (MAX_PID
- MIN_PID + 1
+ NUM_BITS - 1)
/ NUM_BITS);
// Allocate the bitmap
s_bitmap = (__typeof__(s_bitmap)) malloc(s_size);
}
// Does the bitmap exist?
if (s_bitmap)
{
// Initialize the variables
s_index = s_offset = 0;
s_pid = MIN_PID - 1;
return 1;
} else
return -1;
}
//Release the bitmap
void release_map(void)
{
```

```
// Does the bitmap exist?

if (s_bitmap)
```

Step 4/7

```
{

// Free it & deinitialize the variables

free(s_bitmap);

s_bitmap = NULL;

s_index = s_offset = s_pid = -1;

}

}
```

Allocate a pid in a range. It assumes that bitmap is valid. While loop is used to execute the condition until pindex is less then size.

```
static int allocate_pid_range(int *pindex,

const int size)

{

// While the index is less than the size

while (*pindex < size)

{

// While the offset within an element

// is less than the number of bits

while (s_offset < NUM_BITS)

{

// Increment the pid

s_pid++;

// Did the pid exceed the maximum?

if (s_pid > MAX_PID)

{
```

```
// Reset the pid and the offset
s_pid = MIN_PID - 1;
s_offset = 0;
return -1;
}
// Is the offset free?
if (!(s_bitmap[*pindex]
& (1u << s_offset)))
{
// Fill the offset
s_bitmap[*pindex] |= 1u << s_offset;
// Increment the offset
s_offset++;
return 0;
}
// Increment the offset
s_offset++;
}
// Reset the offset
s_offset = 0;
// Increment the index
(*pindex)++;
}
return -1;
}
```

In allocate_pid function allocate the pid from current index to size of bitmap. If pid is allocated then ret value will be 0 otherwise less than 0.

```
//Allocate a pid.
int allocate_pid(void)
{
// Does the bitmap exist?
if (s_bitmap)
{
// Try to allocate a pid from the current index
// to the size of the bitmap
int index = s_index;
int ret = allocate_pid_range(&index, s_size);
// Could a pid be allocated?
if (ret < 0)
{
// Reset the index and try again
index = 0;
ret = allocate_pid_range(&index, s_index);
}
// Was a pid successfully allocated?
if (ret == 0)
{
// Update the index
s_index = index;
// Return the pid
return s_pid;
}
}
```

```
return -1;

}

//Release an allocated pid.

void release_pid(int pid)

{

// Does the bitmap exist and is the pid valid?

if (s_bitmap && pid >= MIN_PID && pid <= MAX_PID)

{

// Subtract MIN_PID so it can be
```

Step 5/7

```
// used to access the bitmap

pid -= MIN_PID;

// Clear the entry for the pid in the bitmap

s_bitmap[pid / NUM_BITS]

&= ~(1u << (pid % NUM_BITS));

}

}
```

Step 6/7

```
//Importing header files

#include

#include "3-20PP.h"

int main(void)

{

// Allocate the bitmap

if (allocate_map() != 1) {

fputs("allocate_map blew\n", stdout);

return -1;
```

```
}
// Allocate 10 pids
int pida[10], i;
for (i = 0; i < sizeof(pida) / sizeof(*pida); i++) {
pida[i] = allocate_pid();
printf("allocate_pid %d\n", pida[i]);
}
// Release 5 pids
for (i = 0; i < sizeof(pida) / sizeof(*pida); i += 2) {
release_pid(pida[i]);
}
fputc('\n', stdout);
// Allocate 10 pids
for (i = 0; i < sizeof(pida) / sizeof(*pida); i += 2) {
pida[i] = allocate_pid();
printf("allocate_pid %d\n", pida[i]);
}
release_map();
return 0;
}
```

Step 7/7

Sample Output:

```
allocate_pid 300
allocate_pid 301
allocate_pid 302
allocate_pid 303
```

allocate_pid 304

allocate_pid 305

allocate_pid -1

allocate_pid -1

allocate_pid -1

allocate_pid -1

allocate_pid 300

allocate_pid 302

allocate_pid 304

allocate_pid -1

allocate_pid -1

Chapter 3, Problem 21PP

(9)

Step-by-step solution

Show all steps

100% (1 rating) for this solution

Step 1/4

Program plan:

Input the number from command line.

Apply conditional loop to check that only positive numbers are provided to command line.

Call fork()function that will return processID of the new process created

Apply conditional loop to check if it is child process, parent process or some error occurred while calling fork()function.

If child process is created, change the value of number as per Collatz conjecture.

Step 2/4

Program:

/********** * Program to implement Collatz conjecture*

**********/

//header file section

#include

#include

#include

Define main method with number and processed as variables.

int main**()**

{

int number**,** processID**;**

/*to get number from command line*/

fprintf**(**"Enter the number for series \n"**);**

fscanf**(**"%d"**,&**number**);**

/*ensure that positive integer is passed*/

Error checking to guarantee that positive number is passed.

if (number**<**0**)**

{

fprintf**(**"The number cannot be negative"**);**

}

For positive number,

else

{

/*fork() function will return processID of the process*/

processID**=**fork**();**

/*if any error occurs while process creation*/

Depending on processID it is determined whether child process(processID**=**0) is created or some error occurred while callingfork**().**

if (processID**<**0**)**

```
{
fprintf(stderr,"The child process could not be
created."\n);
return1;
}
/*for child process*/
else if (processID==0)
{
do
{
/*for odd number*/
if (number%2!=0)
{
number=(number*3)+1;
}
/*for even number*/
else
{
number=number/2;
}
fprintf("%2d",number);
} while (number!=1);
}
```

When processID of parent is generated, parent will wait till child process completes its execution.

```
else
/*for parent process*/
```

```
{

fprintf("\n child process completed");

wait(NULL);

}

}

return0;

}
```

Step 3/4
Sample Output:

Enter the number for series

5

16 8 4 2 1

child process completed

Step 4/4

Output Explanation:

On calling fork() function to create child process, the processID is returned. In child process, the numbers are generated as per Collatz conjecture. The parent process will wait using wait(NULL)command till child process is completed. After that parent process gets completed.

Chapter 3, Problem 22PP

(5)
Step-by-step solution

Show all steps

100% (1 rating) for this solution
Step 1/4

Program Plan:

• Make a function shm_allocate to allocate a shared memory segment.

• Create a child process and check that the child executed normally.

• Print out the contents of the shared memory.

• Release the shared memory object using shm_release function.

Step 2/4

Program:

//Include the header files for performing the operations.

#include

#include

#include

#include

#include

#include

#include

#include

#include

#include

Initialize a variable name as const for shared memory segment, then initialized increment size in shared memory segment in variable size.

// The name of the shared memory segment

const char *****NAME **=** "/13258-3-22PP"**;**

// The size by which to increment the size

// of the shared memory segment

const size_t SIZE **=** 4096**;**

// The size of the shared memory segment

static size_t s_size **=** 0**;**

In shm_allocate function, create a shared memory object and shared memory segment.

//Allocate a shared memory segment

static int shm_allocate**(**size_t size**,** int *****pshm_fd**,**

void ******pptr**)**

```
{
if (!pshm_fd || !pptr)
return -1;
// Set the ptr to shared memory to an invalid value
*pptr = MAP_FAILED;
// Create a shared memory object
*pshm_fd = shm_open(NAME, O_CREAT | O_RDWR, 0666);
if (*pshm_fd < 0)
return -2;
// Set the size of the shared memory object
if (ftruncate(*pshm_fd, size) < 0)
return -3;
// Create a shared memory segment
*pptr = mmap(NULL, size, PROT_WRITE | PROT_READ,
MAP_SHARED, *pshm_fd, 0);
if (*pptr == MAP_FAILED)
return -4;
return 0;
}
```

Use shm_release function to unmap the shared memory segment and unlink the shared memory segment.

```
//Release a shared memory segment and object
static void shm_release(size_t size, int shm_fd,
void *ptr)
{
// Unmap the shared memory segment
if (ptr != MAP_FAILED)
```

```
munmap(ptr, size);

// Unlink the shared memory segment

if (shm_fd >= 0)

shm_unlink(NAME);

}

//The startup function

int main(int argc, char **argv)

{

int n;

int status;

// Check that a positive integer was passed

if (argc != 2) {

fprintf(stderr, "%s [+ve integer]\n", *argv);

return 1;

}

// Set n to the argument the program was invoked with

n = atoi(argv[1]);

if (n <= 0) {

fprintf(stderr, "%s [+ve integer]\n", *argv);

return 2;

}

do {

s_size += SIZE;

// The file descriptor of the shared memory object
```

Step 3/4

```
int shm_fd;

// Pointer to the shared memory
```

```
void *ptr;

// Allocate the shared memory

if (shm_allocate(s_size, &shm_fd, &ptr) < 0) {

fprintf(stderr, "Couldn't allocate "

"shared memory of size %lu "

"bytes\n", s_size);

return 3;

}
```

Fork process is called and process id is stored in variable pid. If pid is greater than zero than wait for child to execute.

```
// Fork off a child process

pid_t pid = fork();

// Is is the parent process?

if (pid > 0) {

// Wait for the child to exit

wait(&status);

}

// Is it the child process?

else if (pid == 0) {

int out;

// While n is greater than 1

while (n > 1) {

// Print n to the shared memory

out = snprintf(ptr, s_size, "%d\n", n);

// Did we run out of space?

if (out >= s_size)

return 1;
```

```
// Decrement size
s_size -= out;
// Increment the pointer into
// the shared memory
ptr += out;
// Apply the Collatz conjecture
n = (n % 2) ? 3 * n + 1 : n / 2;
}
// Print out the final value of n
// to the shared memory
out = snprintf(ptr, s_size, "%d\n", n);
// Return 1 if we ran out of space,
// else return zero
return (out >= s_size) ? 1 : 0;
}
// Did the fork system call not work?
else if (pid < 0) {
fprintf(stderr, "fork blew %d\n", errno);
// Release the shared memory and exit
shm_release(s_size, shm_fd, ptr);
return 4;
}
// Check that the child exicuted normally
if (!WIFEXITED(status)) {
fputs("Child did not exit normally", stderr);
// Was it killed by a signal?
```

```
if (WIFSIGNALED(status))
fprintf(stderr, ", it was terminated by "
"signal %d", WTERMSIG(status));
fputc('\n', stderr);
// Release the shared memory
shm_release(s_size, shm_fd, ptr);
return 5;
}
// Did the child exit with a value of zero?
else if (WEXITSTATUS(status) == 0)
// Print out the contents of the shared memory
fputs(ptr, stdout);
// Release the shared memory
shm_release(s_size, shm_fd, ptr);
}
// Loop while the child did not exit
// with a valid value
while (WEXITSTATUS(status) > 0);
return 0;
}
```

Step 4/4

Sample Output:

```
21:32 3$ ./3-22PP 35
35
106
53
```

160

80

40

20

10

5

16

8

4

2

1

Chapter 3, Problem 23PP

(1)

Step-by-step solution

Show all steps

Step 1/4

Program Plan:

• Define a main() function and create a new server socket.

• Write a Try block in which server socket is created and if error occurred then cached by catch block.

• Create an Object to output to the client.

• Check if the file couldn't be opened. Close the connection to the client.

Step 2/4

Program:

//Importing the package

import java.net**.*;**

import java.io**.*;**

//The quote of the day server class

```
public class QotdServer

{

//The startup function

public static void main(String[] args)
```

In the main function create a server socket. Use a try catch block for handling exceptional error. In the while loop connect to the client and read the QOTD file using buffer reader.

```
{

try {

// Create a new server socket

ServerSocket sock = new ServerSocket(6017);

// Loop forever

while (true) {

// Wait for a client to connect

Socket client = sock.accept();

// Object to output to the client

PrintWriter pout = null;

// Object to read the QOTD file

BufferedReader rin = null;

try {

// Create an output stream

pout = new PrintWriter(

client.getOutputStream(), true);

// Open the QOTD file

rin = new BufferedReader(

new FileReader("3-23PP.qotd"));

// Loop while a line can be read
```

```
String line;

while ((line = rin.readLine())

!= null)

// Write it to the client

pout.println(line);

}
```

Check if the file couldn't be opened using catch block and if unable to open then print and error message.

```
catch (FileNotFoundException

exception) {

System.err.println(exception);

}

// Check if an error occurred on

// reading from the file

catch (IOException exception) {

System.err.println(exception);

}

finally

{

// Close the file input object

if (rin != null)

rin.close();

// Close the output object

if (pout != null)

pout.close();

}

// Close the connection to the client
```

```
client.close();
}
}
// Did accepting a client blow up?
catch (IOException exception) {
System.err.println(exception);
}
}
}
```

Step 3/4

Sample Output:

```
[abc@localhost 3]$ javac QotdServer.java
[abc@localhost 3]$ java QotdServer &
[1] 5916
[abc@localhost 3]$ telnet localhost 6017
Trying ::1...
Connected to localhost.
Escape character is '^]'.
The best definition of a gentleman is a man who can play the accordion --
but doesn't.
```

Step 4/4

```
-- Tom Crichton
Connection closed by foreign host.
```

Chapter 3, Problem 24PP

(1)

Step-by-step solution

Show all steps

Step 1/4

Program Plan:

• Make a class HaikuServer that contain main function.

• Use try catch block for error handling related to file.

• Create a server socket in Try block.

• Connect to the client and read the Haiku file using BufferReader.

• Write the lines to the client.

• If file is unable to open print an error message in catch block.

Step 2/4

Program:

```
//Importing the package

import java.net.*;

import java.io.*;

//The Haiku server

public class HaikuServer

{

//The startup function

public static void main(String[] args)

{
```

In the try block create a server socket. Use a while loop and inside this loop connect to the client and read the Haiku File using BufferReader.

```
try {

// create a new server socket

ServerSocket sock = new ServerSocket(5575);

// Loop forever

while (true) {

// Wait for a client to connect
```

```
Socket client = sock.accept();

// Object to output to the client

PrintWriter pout = null;

// Object to read the Haiku file

BufferedReader rin = null;

try {

// Create an output stream

pout = new PrintWriter(

client.getOutputStream(), true);

// Open the Haiku file

rin = new BufferedReader(

new FileReader("3-24PP.haiku"));

// Loop while a line can be read

String line;

while ((line = rin.readLine())

!= null)

// Write it to the client

pout.println(line);

}
```

In the catch block file exception is handled. If file could not be opened then it will show an error message. In finally block, close the file input object.

```
// Check if the file couldn't

// be opened

catch (FileNotFoundException

exception) {

System.err.println(exception);

}
```

```
// Check if an error occurred on
// reading from the file
catch (IOException exception) {
System.err.println(exception);
} finally {
// Close the file input object
if (rin != null)
rin.close();
// Close the output object
if (pout != null)
pout.close();
}
// Close the connection to the client
client.close();
}
}
// Did accepting a client blow up?
catch (IOException exception) {
System.err.println(exception);
}
}
}
```

Step 3/4

Sample Output:

```
[abc@localhost 3]$ javac HaikuServer.java
[abc@localhost 3]$ java HaikuServer &
```

[1] 5512

[abc@localhost 3]$ telnet localhost 5575

Trying ::1...

Connected to localhost.

Escape character is '^]'.

An old silent pond...

Step 4/4
A frog jumps into the pond,

splash! Silence again.

Connection closed by foreign host.

Chapter 3, Problem 26PP

(3)
Step-by-step solution

Show all steps

Step 1/4

Program Plan:

• Create an index for reading and writing files from pipe.

• Initialize the size of buffer.

• Create a function write_string that takes the size of string.

• Create another function read_buffer that read the string.

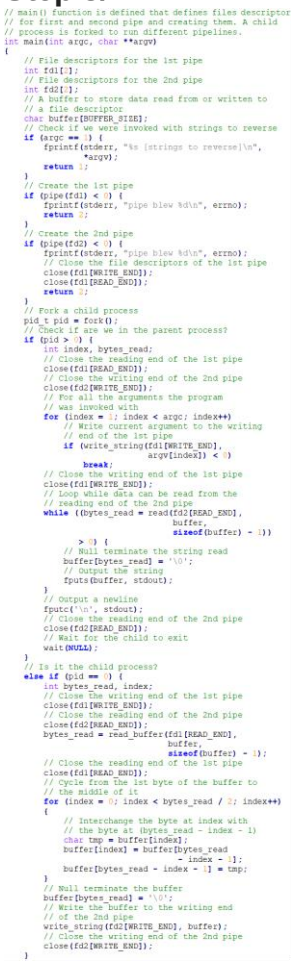
• The main() function that create the pipes. It uses fork function and calls other function to print string in reverse order.

Step 2/4

Program:

```c
//Header file section
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

// The index of the file descriptor for reading from a
pipe
const int READ_END      = 0;
// The index of the file descriptor for writing from a
pipe
const int WRITE_END     = 1;
// The size of the buffer
const int BUFFER_SIZE           = 512;

//Write a null terminated string to a file descriptor
ssize_t write_string(int fd, char *string)
{
    // Used to store the size of the string
    size_t size;
    if (fd < 0 || !string)
        return -1;
    // Get the size of the string
    size = strlen(string);
    // While there are bytes left to send
    while (size > 0) {
        // Try to write size bytes
        ssize_t sent = write(fd, string, size);
        // Check for error
        if (sent < 0) {
            int err = errno;
            fprintf(stderr, "write error %d - %s\n",
                    err, strerror(err));
            return -2;
        }
        // Update size & buffer
        size -= sent;
        string += sent;
    }
    return 0;
}

// Read_buffer function is created to initialized index
and // attempt to read (size - index) bytes into buffer
at
// offset index. Read from a file descriptor into a
// buffer of size bytes.
ssize_t read_buffer(int fd, char *buffer, size_t size)
{
    // The index into the buffer
    ssize_t index;
    if (fd < 0 || !buffer || size < 0)
        return -1;
    // Initialize the index
    index = 0;
    // While the index into the buffer is less
    // than its size
    while (index < size) {
        // Attempt to read (size - index) bytes
        // into buffer at offset index
        ssize_t bytes_read = read(fd, buffer + index,
                                  size - index);
        // If nothing left then break out of the loop
        if (bytes_read == 0)
            break;
        // Has an error occurred?
        if (bytes_read < 0) {
            int err = errno;
            fprintf(stderr, "read blew %d - %s\n",
                    err, strerror(err));
            return -2;
        }
        // Add the bytes read to update the index
        index += bytes_read;
    }
    return index;
}
```

Step 3/4



```
// main() function is defined that defines files descriptor
// for first and second pipe and creating them. A child
// process is forked to run different pipelines.
int main(int argc, char **argv)
{
    // File descriptors for the 1st pipe
    int fd1[2];
    // File descriptors for the 2nd pipe
    int fd2[2];
    // A buffer to store data read from or written to
    // a file descriptor
    char buffer[BUFFER_SIZE];
    // Check if we were invoked with strings to reverse
    if (argc == 1) {
        fprintf(stderr, "%s [strings to reverse]\n",
                *argv);
        return 1;
    }
    // Create the 1st pipe
    if (pipe(fd1) < 0) {
        fprintf(stderr, "pipe blew %d\n", errno);
        return 2;
    }
    // Create the 2nd pipe
    if (pipe(fd2) < 0) {
        fprintf(stderr, "pipe blew %d\n", errno);
        // Close the file descriptors of the 1st pipe
        close(fd1[WRITE_END]);
        close(fd1[READ_END]);
        return 2;
    }
    // Fork a child process
    pid_t pid = fork();
    // Check if are we in the parent process?
    if (pid > 0) {
        int index, bytes_read;
        // Close the reading end of the 1st pipe
        close(fd1[READ_END]);
        // Close the writing end of the 2nd pipe
        close(fd2[WRITE_END]);
        // For all the arguments the program
        // was invoked with
        for (index = 1; index < argc; index++)
            // Write current argument to the writing
            // end of the 1st pipe
            if (write_string(fd1[WRITE_END],
                             argv[index]) < 0)
                break;
        // Close the writing end of the 1st pipe
        close(fd1[WRITE_END]);
        // Loop while data can be read from the
        // reading end of the 2nd pipe
        while ((bytes_read = read(fd2[READ_END],
                                  buffer,
                                  sizeof(buffer) - 1))
               > 0) {
            // Null terminate the string read
            buffer[bytes_read] = '\0';
            // Output the string
            fputs(buffer, stdout);
        }
        // Output a newline
        fputc('\n', stdout);
        // Close the reading end of the 2nd pipe
        close(fd2[READ_END]);
        // Wait for the child to exit
        wait(NULL);
    }
    // Is it the child process?
    else if (pid == 0) {
        int bytes_read, index;
        // Close the writing end of the 1st pipe
        close(fd1[WRITE_END]);
        // Close the reading end of the 2nd pipe
        close(fd2[READ_END]);
        bytes_read = read_buffer(fd1[READ_END],
                                 buffer,
                                 sizeof(buffer) - 1);
        // Close the reading end of the 1st pipe
        close(fd1[READ_END]);
        // Cycle from the 1st byte of the buffer to
        // the middle of it
        for (index = 0; index < bytes_read / 2; index++)
        {
            // Interchange the byte at index with
            // the byte at (bytes_read - index - 1)
            char tmp = buffer[index];
            buffer[index] = buffer[bytes_read
                                   - index - 1];
            buffer[bytes_read - index - 1] = tmp;
        }
        // Null terminate the buffer
        buffer[bytes_read] = '\0';
        // Write the buffer to the writing end
        // of the 2nd pipe
        write_string(fd2[WRITE_END], buffer);
        // Close the writing end of the 2nd pipe
        close(fd2[WRITE_END]);
    }
```

Step 4/4

```
    // Did the fork system call malfunction?
    else
    {
        fprintf(stderr, "fork blew %d\n", errno);
        // Close the file descriptors of both the pipes
        close(fd1[WRITE_END]);
        close(fd1[READ_END]);
        close(fd2[WRITE_END]);
        close(fd2[READ_END]);
        return 3;
    }
    return 0;
}
```

Sample Output:

```
[strings to reverse]
abcdefghijklmnopqrstuvwxyz
zyxwvutsrqponmlkjihgfedcba
```

Step-by-step solution

Show all steps

Step 1/5

Program Plan:

• Create an index for reading and writing files from pipe.

• Write to a file descriptor from a buffer of size bytes using function write_buffer.

• Read from a file descriptor into a buffer of size bytes using function read_buffer.

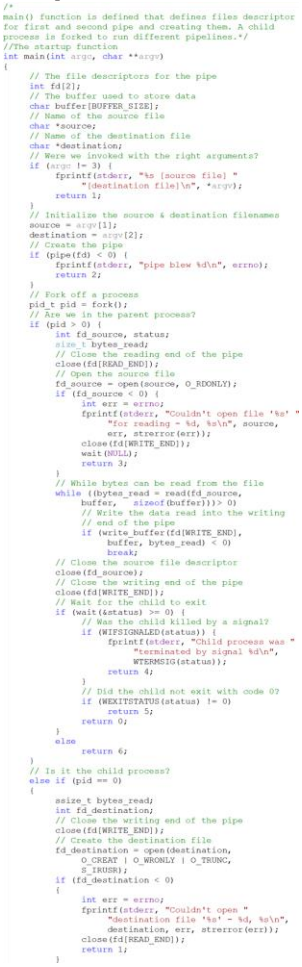
• Write a main() function. It uses fork function and also calls other function to read and write file.

Step 2/5

Program:

```c
//Include the header files for performing the operations.
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>

// The index of the file descriptor for reading from a pipe
const int READ_END = 0;
// The index of the file descriptor for writing from a pipe
const int WRITE_END = 1;
// The size of the buffer
const int BUFFER_SIZE = 512;

//Write to a file descriptor from a buffer of size bytes

ssize_t write_buffer(int fd, char *buffer, size_t size)
{
    if (fd < 0 || !buffer || size < 0)
        return -1;
    // While there are bytes left to send
    while (size > 0) {
        // Try to write size bytes
        ssize_t sent = write(fd, buffer, size);
        // Was there an error?
        if (sent < 0) {
            int err = errno;
            fprintf(stderr, "write error %d - %s\n",
                err, strerror(err));
            return -2;
        }
        // Update size and buffer
        size -= sent;
        buffer += sent;
    }
    return 0;
}


/*Read from a file descriptor into a buffer of size bytes.
read_buffer function is created to initialized index and
attempt to read (size - index) bytes into buffer at
offset index. Read from a file descriptor into a buffer
of size bytes. */
ssize_t read_buffer(int fd, char *buffer, size_t size)
{
    // The index into the buffer
    ssize_t index;
    if (fd < 0 || !buffer || size < 0)
        return -1;
    // Initialize the index
    index = 0;
    // While the index into the buffer is less
    // than its size
    while (index < size) {
        // Attempt to read (size - index) bytes
        // into buffer at offset index
        ssize_t bytes_read = read(fd, buffer + index,
            size - index);
        // If nothing left then break out of the loop
        if (bytes_read == 0)
            break;
        // Has an error occurred?
        if (bytes_read < 0) {
            int err = errno;
            fprintf(stderr, "read blew %d - %s\n",
                err, strerror(err));
            return -2;
        }
        // Add the bytes read to update the index
        index += bytes_read;
    }
    return index;
}
```

Step 3/5



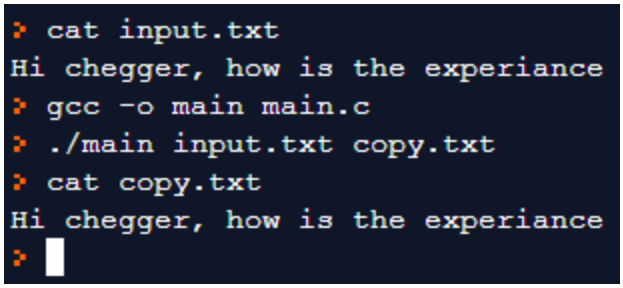
```
/*
main() function is defined that defines files descriptor
for first and second pipe and creating them. A child
process is forked to run different pipelines.*/
//The startup function
int main(int argc, char **argv)
{
    // The file descriptors for the pipe
    int fd[2];
    // The buffer used to store data
    char buffer[BUFFER_SIZE];
    // Name of the source file
    char *source;
    // Name of the destination file
    char *destination;
    // Were we invoked with the right arguments?
    if (argc != 3) {
        fprintf(stderr, "%s [source file] "
            "[destination file]\n", *argv);
        return 1;
    }
    // Initialize the source & destination filenames
    source = argv[1];
    destination = argv[2];
    // Create the pipe
    if (pipe(fd) < 0) {
        fprintf(stderr, "pipe blew %d\n", errno);
        return 2;
    }
    // Fork off a process
    pid_t pid = fork();
    // Are we in the parent process?
    if (pid > 0) {
        int fd_source, status;
        ssize_t bytes_read;
        // Close the reading end of the pipe
        close(fd[READ_END]);
        // Open the source file
        fd_source = open(source, O_RDONLY);
        if (fd_source < 0) {
            int err = errno;
            fprintf(stderr, "Couldn't open file '%s' "
                "for reading - %d, %s\n", source,
                err, strerror(err));
            close(fd[WRITE_END]);
            wait(NULL);
            return 3;
        }
        // While bytes can be read from the file
        while ((bytes_read = read(fd_source,
            buffer,   sizeof(buffer)))> 0)
            // Write the data read into the writing
            // end of the pipe
            if (write_buffer(fd[WRITE_END],
                buffer, bytes_read) < 0)
                break;
        // Close the source file descriptor
        close(fd_source);
        // Close the writing end of the pipe
        close(fd[WRITE_END]);
        // Wait for the child to exit
        if (wait(&status) >= 0) {
            // Was the child killed by a signal?
            if (WIFSIGNALED(status)) {
                fprintf(stderr, "Child process was "
                    "terminated by signal %d\n",
                    WTERMSIG(status));
                return 4;
            }
            // Did the child not exit with code 0?
            if (WEXITSTATUS(status) != 0)
                return 5;
            return 0;
        }
        else
            return 6;
    }
    // Is it the child process?
    else if (pid == 0)
    {
        ssize_t bytes_read;
        int fd_destination;
        // Close the writing end of the pipe
        close(fd[WRITE_END]);
        // Create the destination file
        fd_destination = open(destination,
            O_CREAT | O_WRONLY | O_TRUNC,
            S_IRUSR);
        if (fd_destination < 0)
        {
            int err = errno;
            fprintf(stderr, "Couldn't open "
                "destination file '%s' - %d, %s\n",
                destination, err, strerror(err));
            close(fd[READ_END]);
            return 1;
        }
```

Step 4/5

```
            // While bytes can be read from the reading
            // end of the pipe
            while ((bytes_read = read_buffer(fd[READ_END],
                 buffer,   sizeof(buffer)))> 0)
                 // Write the bytes read to the file
                 write(fd_destination, buffer, bytes_read);
            // Close the destination file descriptor
            close(fd_destination);
            // Close the reading end of the pipe
            close(fd[READ_END]);
            return 0;
        }
        // Did the system call fork malfunction?
        else {
            fprintf(stderr, "fork blew %d\n", errno);
            // Close the pipe
            close(fd[WRITE_END]);
            close(fd[READ_END]);
            return 7;
        }
}
```

Step 5/5

Sample Output:


cat input.txt
Hi chegger, how is the experiance
gcc -o main main.c
./main input.txt copy.txt
cat copy.txt
Hi chegger, how is the experiance