Chapter 3 - Process

1. **Chapter 3: Processes:** In this chapter we will introduce processes, and common operations performed on processes, scheduling and inter-process communication mechanisms.

2. **Process concept:** Job is the old reference for a process.

   Process is an active entity containing a program counter specifying the next instruction to be executed.

3. **Process in Memory:** Text section contains the program code. Data section holds the global variables and statics. (Initialized globals/statics in data section, uninitialized in .bss section). Dynamically allocated data is maintained in the heap. Function activation records (local variables and some other information) are maintained on the stack. Heap and stack grow towards each other.

   All these sections exist in memory in what is called the address space of a process. The OS maintains a PCB containing information about each process.

   We can use the 'size' command to get sizes of various sections in an executable.

   Question: Why does it not show the sizes of stack and heap?

4. **Process State:** OS maintains a process state in a time-shared system. A process may wait for I/O to be completed or some other event to happen. A time-shared OS can then schedule some other ready process to run while this process is waiting.

5. **Diagram of a Process State:** A process may transition between different states during its execution.

6. **Process Control Block:** PCB contains all the information required to swap out a running process, and restart execution of a process that was waiting for some event. It is a repository for any process-specific information.

   The OS needs to hold at least the following types of information in the PCB.
   PC (program counter) – to resume execution.
   Registers – CPU only contains one set of registers, time-sharing systems may run multiple processes.
   Scheduling info priority or other parameters based on scheduling algorithm used.
   Memory management may include values of base and limit registers, page table, segment table information, etc.
   Accounting info includes amount of CPU and real time used, time limit, job or process numbers, etc.
   I/O information includes list of I/O devices allocated to the process, list of open files, etc.

7. **Process Control Block:** Figure shows some fields of a PCB.

8. **CPU Switch from Process to Process:** The PCB holds all necessary process state information to allow this switch. Note that the time spent in saving and restoring process state information in the PCB is OS overhead during which no useful work is done. All OS try to minimize this overhead.

   Some processors may be able to hold the state of multiple processes simultaneously.

9. **Process Scheduling:** Scheduling is only necessary in the presence on multiple processes in a multiprogrammed or time-shared system. Remember that even if there is only user program running, there may be several other OS programs running at the same time.

   Each device may have its own device queue. Process migration can be explained from the next figure.

10. **Ready Queue and Various I/O device queues:** Queues are generally linked lists, which makes managing the lists easy, since processes frequently move between various queues. Only one process in the ready queue may be allocated to the CPU at any instance.

    If a running process quits, waits, or is interrupted (like by OS timer), it may move to one of the device queues or go back to the end of the ready queue.

11. **Representation of Process Scheduling:** This figure is called a *queuing diagram.* Each rectangular box represents a queue. Circles represent the resources that represent a queue. Arrows indicate the flow of processes in the system.

    A running process could:
    1. issue an I/O request and migrate to the I/O queue.
    2. process could voluntarily wait (maybe for child's completion)
    3. scheduler should forcibly remove process from the CPU.

12. **Schedulers:** Many general-purpose OSes including Unix and MS-Windows do not use multiple schedulers. They often do nothing or depend on some system-defined constant to limit the number of active processes in the system at any one point in time.

    I/O-bound process spends more time doing I/O than computations, many short CPU bursts. CPU-bound process spends more time doing computations; few very long CPU bursts.

    Getting the right mix of ready CPU-bound and I/O bound processes is important: if all I/O bound then ready queue will always be empty, if all CPU-bound then I/O queue will be empty and devices will go under-utilized.

    The main difference between the two schedulers is the frequency of invocation, which determines the algorithm used in the two schedulers.

13. **Context Switch:** Context switch happens frequently on most machines today. Therefore, it is important to limit overhead of the switch. A mode switch may not necessarily involve a context switch.

    Context switch overhead depends on the machine (memory speed, number of registers that must be copied, existence of special instructions, etc.) SPARC can hold the states of multiple processes in memory simultaneously.

    Check out the output of the '`time`' command that shows 'user' and 'system' time for each process. These times are spent in different modes, but in the same process context.

14. **Process Creation:** Each OS starts a primordial process during system bootup. This is the *init* process on Unix.

    A process needs resources like CPU time, memory, files, I/O devices, to accomplish its work. Restricting the child to a subset of the parents resources may limit the parent creating excess processes. On Unix, each child gets its own resources from the OS, file and I/O descriptors are copied to the child address space from the parent.

    In Unix parent and child start executing concurrently, but parent can wait for the child to finish execution by explicitly calling the '`wait`' system call.

    On Unix the child gets a duplicate of the parent address space.

15. **Process Creation Example on Unix:** The child process overlays its address space with the Unix command /bin/ls, using a version of the exec() system call.

16. **Process Creation:** Figure shows parent waiting for child to terminate, as in the example before. See fork.c

17. **Process Termination:** If the parent terminates before the child, then the child becomes a child of the init process under Unix. Thus parent exiting does not terminate the child, and init can collect its return status.

18. **Interprocess Communication:** A process is independent if it does not affect and cannot be affected by any other process in the system. If the process affects or is affected by other processes in the system, then the process is called co-operating process.

    Remember this is communication within the same OS between processes active at the same time.

    Use case scenarios for IPC:
    1. Several users may be interested in the same piece of information, like a shared file.
    2. Speedup can be increased with parallel computation, breaking tasks into parallel subtasks. Any such speedup can only be achieved if there are multiple CPUs in the system.
    3. Modularity by dividing the system functions into separate processes. User working on several tasks at the same time, such as editing, printing, compiling, may find if easier to divide work into multiple processes and share information among them.
    4. Convenience – instead of computing the same data in two processes, they can use IPC.

19. **Producer Consumer Problem:** The producer consumer problem can be addressed using various forms of inter-process communication using shared memory. It is a very common type of abstraction. The unbounded buffer cannot be implemented in most systems today.

20. **Models of IPC:** There are two primary IPC models:

    - Shared Memory: Note that this requires over-riding the address space security mechanisms setup by the OS to keep processes separate. Communication is faster since system calls are only required during the initial stage of setting up the shared region. Later, there is direct communication. Accessing shared memory is similar to accessing normal local memory, so is more convenient. With shared memory, both processes writing to the same shared region at the same time, will overwrite each other. So, synchronization may be required.
    - Message Passing: There is more overhead, since messages are copied from the sender to the kernel, and then to the receiver address space. So, sending small amounts of data is preferred. For the same reason, communication is slower, since system calls are needed for each message sent and received. Messages do not overwrite each other, so their is less concern about synchronization.

21. **Models of IPC (2):** Figure shows a schematic.

    - Shared memory: Shared region is generally created in the address space of the process initiating the region. Other interested processes must attach this region to their address space. The form of data and the location in the address space are determined by the communicating processes and not by the OS. The OS removes the restrictions on address space sharing. The processes are also responsible for synchronizing accesses, a very important topic that we will study later as well.

- Message Passing: We can see that the data is copied from the sender to the kernel to the receiver. This copying along with the explicit system calls for each data transfer, makes this approach more expensive than shared memory for processes on the same system. However, this technique is important in distributed environments where the communicating processes may reside on different computers, and thus cannot share address spaces.

22. **Message Passing:** Can be used for communication between non-local processes. The link can be implemented in a variety of ways, including shared memory, hardware bus, network, etc. These implementation issues vary according to the system.

    We will next look at naming techniques to see how links can be established; links can be established first before all communication (like TCP), or for each message sent/received (like UDP)? Link between more than 2 processes, such as one sender and multiple receivers, can create synchronization problems. Any receiver may be able to pick up the message sent by the sender.

    Number of links depends on naming scheme for the links, and system implementation. Link capacity can either be specified by the creator of the link, or can be a system-defined constant.

    Fixed size messages makes it easier for system implementation, but makes it harder for the programmer, unless the link creator makes large links to accommodate all messages. Conversely true for variable size messages.

23. **Message Passing – Naming:** The process only needs to know the identity of the other process to communicate. There is also a scheme with asymmetric naming, such that the sender specifies the receiver queue, but the receiver does not. Then, multiple processes can send to the same queue.

    With hard-coding the process id, we are in trouble if the process id changes. Therefore, we can use the other indirect naming scheme.

24. **Message Passing - Naming (2):** Without synchronization, the multiple receiver problem may be handled by:
    1. Allow a link to be associated with at most two processes
    2. Allow only one process at a time to execute a receive operation
    3. Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

    Mailbox in the process space has a default owner, and mailbox is deleted when process terminates. If mailbox in OS, then may have no owner. Explicit deletion of mailbox may be necessary to release resources.

25. **Message Passing (3):** Synchronization:
    Different combinations of send and receive are possible. When both are blocking, synchronization problem disappears, since send() sends a message and waits until the receive() gets it. Similarly, the receiver does not proceed until it gets a message. The '`select`' system call can be used to implement a non-blocking receive.

    Sent messages reside in a temporary queue. Zero capacity – sender must be blocking.

26. **Interprocess Communication in Linux:** Unix and its many variants provide several mechanisms for interprocess communication.

27. **Pipes**

- Pipes are the earliest form on IPC in Unix systems. They are also the simplest.
- Two-way communication may be achieved via two pipes.
- Ordinary pipes in Unix, also called anonymous pipe, can only be setup between parent and child processes. Another mechanism, called named pipes, that we study later, gets rid of this limitation.
- Since pipes can only be setup between parent-child, inter-computer communication is not allowed.
- If process exits before collecting the data, then it will be lost.
- Data only collected in the order it is entered. We will later study message queues to relax this limitation.

28. **Simple Example Using Pipes:** Pipe is constructed in the same process. The standard Unix system calls, read and write, are used for communication with pipes. The pipe is destroyed when the process ends.

29. **IPC Example Using Pipes:** This works because child copies the entire parent address space. So, file descriptors fd[0] and fd[1] are copied in the child space as well.

30. **Pipes Used for Process Synchronization:** A useful property of pipes is that the read is blocking, until something is written in the write end of the pipe. This property can be used to implement a naive synchronization mechanism.

31. **Pipes Used in Unix Shells:** Unix command-line shells provide several small system programs. A common activity is constructing more complex commands from these simple commands. Pipes are an irreplaceable tool to achieve this.

    ps is used to display status of processes running on the system.
    -e option is used to display all processes.
    -f is used to print additional process status information.
    By itself the screen scrolls. So, *more* is used to view command output one screen at a time. Output of ps is input to more.

32. **Pipes Used in Unix Shells (2):** *dup* is used to duplicate the specified file descriptor. The duplicated descriptor takes on the lowest file descriptor number available. We can also use dup2 to explicitly specify the new file descriptor number.

    Pipe will also be covered during Lab-4.

33. **FIFO (Named Pipes):**

    - Named pipes attempt to over come some limitations of normal pipes.
    - Several process can use the FIFO for communication, it appears in the file system as any other file.
    - If a process opens the FIFO file for reading, it is blocked until another process opens the file for writing. The same goes the other way around.
    - Reading from an empty FIFO does not return a EOF value.
    - FIFOs consume system resources, and must be explicitly removed.
    - FIFOs are half-duplex; file cannot be opened in read/write mode.

34. **Producer Consumer Example with FIFO:** Producer Code: The *open()* blocks on until someone opens the same fifo file for reading.

    The second argument to open specify the file access permissions. A FIFO can also be created on the command line using the mknod() command. Write to the fifo from the command line using standard file commands.

35. **Producer Consumer Example with FIFO (2):** Consumer Code: Consumer code is similar. If producer quits, then read() returns 0. Thus, reader can tell when all writers have closed. If there are multiple writers then read() does not return a zero unless all writers have closed!

    We can have multiple readers and writers for the same FIFO. Any communication then will need to be synchronized. There is no way using FIFOs to specify that a message is for some specific reader. Messages can only be read in FIFO order.

36. **Message Queues:** Message queues provide a way to specify a message type, and messages of only that kind can be removed from the queue in FIFO order, even if there are messages of some other type ahead in the queue.

37. **Message Queue Example:** *key* is specified to be 0. It is just a long integer. It is unique for each queue.

    IPC_CREAT is or'ed with the permissions on the queue. Queue length is initialized by the system, but can be reset using msgctl().

    As long as the first word in the buffer is a long, the rest of the buffer can contain anything of any length. In our example program, we have a character array of 1000 bytes.

    mtype is the type of the message. This can be used during msgrcv() to only retrieve messages of a particular type from the queue in a FIFO order. msgsnd() can block if there is not enough space in the queue.

    We specify 0 as fourth argument in msgrcv(), so that we get the first message from the queue.

    If we don't terminate the queue then it will exist even after process termination, and even if no process uses it, and consume system resources. We can view present queues using ipcs command.

38. **Message Passing in Unix Systems:** This slide answers some of the questions regarding message passing that we had posed earlier, in the context of Linux message queues.

39. **Memory Sharing in Unix:** Processes can also communicate using *Shared memory.*

40. **Shared Memory Example:** This is a naive example showing how a single process uses shared memory on Linux.

    For real IPC, multiple processes may share the shared region which will cause race conditions. So, some kind of synchronization mechanism will be necessary. Therefore, we delay such an example until we learn about process synchronization.

    `shmat()` returns a void pointer, and here we are casting it to a char pointer.

41. **Shared Memory Example (2):** The first argument to shmget is again the key (which can get using `ftok()`). It can also be specified to IPC_PRIVATE for a always new anonymous memory segment.

Access to file region is as for file access for user, group, and others. Linux does not implement execute permissions.

We can specify other commands as well with `shmctl()`. (Read man page for `shmtl()`).

ipcs command can be used to check the status of the message queues and shared memory segments in the system.

42. **Unix Domain Sockets:** Similar to pipes, but allows full-duplex communication. Sockets are most commonly used on the Internet. However, here we will first study Unix domain sockets.

    Like pipes, sockets are a special file in the file system. However, the communication does not use the file interface. Thus, we won't use open(), close(), read(), write() system calls.

    Unix domain sockets are very similar to Internet sockets, with minor differences.

    Main reason for using connectionless sockets is speed. Setting up a connection, and providing for error-free communication involves a lot of overhead. Connectionless communication, which may lose some packets, may be enough for some activities, like streaming video and audio, Even ftp uses UDP, but has its own higher level protocol providing error-free communication, maintaining the same order of data sent.

43. **Unix Domain Sockets – System Calls:** `Socket()`: domain can be used to specify local communication, or a host of protocols like IPv4, IPv6, Novell IPX, etc.
    'type' can be used to specify sequences, reliable, connection-oriented SOCK_STREAM, or SOCK_DGRAM, etc.
    'Protocol' is mostly 0, but can be used if the socket type supports more than one protocol.

    `bind()`: bind socket to a unique address in the Unix domain, a special type of file.

    `listen()`: If there are this many connections waiting to be accepted, additional clients will generate the error ECONNREFUSED.

    `accept()`: The accept() system call is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET). It extracts the first connection request on the queue of pending connections, creates a new connected socket, and returns a new file descriptor referring to that socket.
    The argument '`addr`' is a pointer to a sockaddr structure. This structure is filled in with the address of the peer socket, as known to the communications layer.
    The '`addrlen`' argument is a value-result argument. It should initially contain the size of the structure pointed to by addr; on return it will contain the actual length (in bytes) of the address returned.

44. **Socket Example – Echo Server:** See the examples in the notes.

45. **Communications in Client Server Systems:** All the interprocess communication mechanisms we have seen only allow communication on the same system. We might need to share information between processes in a distributed environment, where multiple computers are linked using a network. Client-server mechanisms help achieve this. They are most commonly based on a message passing paradigm.

    Sockets can use several standard protocols, as we have just seen, for their communication. Further examples using Sockets over Internet will be discussed in the Labs. The book has an example of using Sockets in Java.

46. **Sockets:** Instead of using the file interface, as done by pipes, socket communications use the socket interface.

47. **Remote Procedure Calls:** RPC is built on top of an IPC mechanism that we have seen earlier. Client and server typically do not share an address space, so message-passing is used.

    Messages in common IPC mechanisms may be just bare packets of data. RPC daemon listens to a particular port for requests. Port is simply a number denoting the procedure to be invoked.

    Sever kernel may provide a rendezvous mechanism to locate the correct port on the server. This rendezvous daemon is also called a matchmaker. Client queries this matchmaker daemon.

    Message received by the server will include the service name, and the associated parameters.

    Marshalling is required due to differing architectures on the client and the server. One particularly annoying difference is the architecture byte-order (big-endian or little-endian). RPCs use a standard machine-independent representation, such as external data representation (XDR).

48. **Marshalling Parameters:** Stub is the client-side routine converting all data to a machine-neutral representation. Skeleton converts the marshalled arguments into machine-specific format for the server. The return value may be marshalled again by the skeleton before communicating it to the client.