Basic Set of Material to Cover in Lecture on Chapter Three of 9th Edition of Silberschatz.

PROCESSES

* In a modern multiprogramming or time-sharing computing system, the process is the unit of work
* Such systems usually have a multiplicity of operating system processes, and also user processes.

OBJECTIVES

* introduce the concept of a process
* describe process features
* describe interprocess communication
* describe client-server communication

SECTION 3.1 - Process Concept

* A very general concept of a (sequential) process is that it's a "CPU activity". The term "job" is synonymous with "process." "Job" is an older term.

3.1.1 - The Process

* A process is "a program in execution."
* A program has a great deal of "context" - context is the information needed to maintain the process - in other words all the information required to restart it if it is suspended.
* Some elements of the context are: the code (text), which is the actual instructions of the process residing in main memory; the value of the program counter; the contents of the registers used by the process; the stack of the process and the contents of the stack - such as parameters, return addresses, and local variables; the data section of the process, containing global variables; and often a the heap storage area of the process.
* The program is not the same as the process. The program is a passive executable file that resides on secondary storage. An executable file has a specific structure, which has to be known to the operating system.
* The process is an active entity.
* More than one process can execute the same program, often concurrently, as when several users are executing the same e-mail program or editor concurrently on a system.

3.1.2 - Process State

* Process State is another possible element of the context of a process.
* The possible states of a process are:
  + New (being created)
  + Running
  + Waiting
  + Ready (to run)
  + Terminated

* Only one process can be running on a CPU at any given instant in time,

because a CPU only executes one instruction at a time.

3.1.3 - Process Control Block

* An OS, like other software, needs to use data structures to represent important entities. There has to be a data structure that represents each process. We refer to that data structure as a Process Control Block (PCB). The PCB has to somehow include all the process context information. Since there is a huge amount and variety of context, the PCB is an abstraction. In a real system, there is always a large number of data structures with links and references to one another that together comprise the abstraction of the PCB.
* Some of the things included in the PCB are:
  + Process state
  + Program counter
  + CPU registers
  + CPU-scheduling information (e.g. process priority, pointers to scheduling queues)
  + Memory-management information (e.g. process memory locations and boundaries)
  + Accounting information
  + I/O status information (e.g. devices used by the process, a list of the files the process has open)

(To see how a PCB is represented in Linux, see the inset on page 110 of chapter 3 entitled "Process Representation in Linux")

3.1.4 - Threads

* A process can contain more than one execution sequence - multiple "threads"
* sets of threads are basically sets of processes that share more context than 'traditional' processes.

SECTION 3.2 - Process Scheduling

* CPU Scheduling is the selection that the OS makes of the next process to execute in the CPU.

3.2.1 - Scheduling Queues

* Typically the OS maintains the PCBs of the processes that are ready to execute in a linked list called the "ready queue".

* However, the ready queue is seldom a FIFO queue.

* The OS also typically maintains such queues for each I/O device, in which it keeps the PCBs of processes that are waiting for a pending I/O operation on that device.

* There are queues where processes can wait for other types of events. For

example a parent process can wait in an "event queue" for a child process to terminate.

* In a running computing system, there is a lot of migration of processes back and forth from one type of queue to another.

3.2.2 - Schedulers

* In a batch oriented system, there can be a queue of processes on secondary storage waiting for a turn to be brought into the primary memory for execution by a part of the OS called the long-term scheduler.

* The short-term scheduler is the CPU scheduler that selects the next process to run from the ready queue. All the jobs in the ready queue are ready to run - so of course they are resident in primary memory.

* It is called a context switch when the process running in the CPU is changed. The short-term scheduler has to execute every time there is a context switch.

* The time each process spends in the CPU before a context switch is called the burst time of the process.

* The time it takes to perform a context switch, including the time it takes for the short-term scheduler to choose the next process to execute, is overhead. That context switch time has to be as short as possible in relation to the average burst time, or else CPU utilization will be low, as well as many other measures of system performance, such as response time, throughput, and turnaround time.

* Performance of a long-term scheduler is seldom critical, since basically all it has to do is put new processes into primary memory at the same rate that processes are terminating and vacating primary memory.

* Long-term schedulers should produce a good "job mix" - a set of processes in memory that keep the CPU and I/O channels as busy as possible.

* Most PCs and time-sharing systems don't have a long-term scheduler. A medium-term scheduler (swapper) is utilized to prevent primary memory from becoming overloaded or to improve the job mix, by migrating jobs back and forth between primary and secondary memory.

* The "degree of multiprogramming" is the number of processes resident in primary memory.

3.2.3 - Context Switch

* If the architecture offers multiple sets of registers, they can be switched to speed up context switch time.

* Naturally context switch time depends on how much data has to be saved and restored, and that varies from OS to OS.

SECTION 3.3 - Operations on Processes

3.3.1 - Process Creation

* Operating systems typically provide every process with the ability to create a "child process" by making a system call.

* Each process in the system has a unique identification number - its process id (pid).

* In unix-like operating systems, there is a process called init with pid 1 that is the ancestor of all user processes. Init is not a child process. It is 'handcrafted' by the OS as it boots.

* Depending on the OS, child processes may or may not be limited in the amount of resources they are able to obtain. Such limits can be useful in preventing arbitrary processes from overloading the system by creating too many child processes, or having too many descendants.

* Parent processes need to be able to pass information and resources to their child processes, so that the child processes will know what work the parent intends for them to perform, and so they will have the resources they need to get the work done. For example, the parent might need to tell the child which program to execute, and give the child some open files to read from and write to.

* A parent process may choose to execute concurrently with one or more of its child processes, or it may choose to suspend itself (wait) until its child process terminates.

* A parent process waits for a child by making a system call.

* A child process may be a clone of its parent, and execute the same program, or it may load and execute a different program.

* In unix, a parent creates an exact duplicate of itself with a fork() system call. If the child wants to load and execute a different program, it makes an exec()

system call after it is 'spawned' by its parent.

* Although they are clones, the parent can tell it is the parent because the fork() system call returns the pid of the child to the parent. It returns 0 to the child.

* The programmer can write the program (shared by both the parent and the child) so that an if-else statement is executed right after the call to fork(). That way the process with the return code of 0 can do something different from the process with the non-zero return code.

* In unix, a child process inherits rights like access privileges and resources like open files from its parents.

* In Windows, a parent creates a child process with the CreateProcess() function. When calling CreateProcess(), the parent is required to specify a program for the child to execute, and many other parameters.

3.3.2 - Process Termination

* Processes typically execute an exit() statement of some kind as their last instruction. The effect is a system call that causes the OS to deallocate all the resources of the process. Often exit() causes the return of an integer 'exit status' to a parent that has called wait().

* A parent can call wait() like this

pid = wait(&status)

where pid and status are integer data types. Such a call to wait() allows the parent to collect the exit status of the child in the status parameter, and to discern which child exited by examining the returned value of the child's process id number.

* There are usually system calls with which a parent or other privileged process can cause the termination of a child process. For example, traditional unix uses the kill() system call to send the SIGKILL signal to a process, and Windows has a TerminateProcess() system call.

* A parent process might terminate a child process because it is operating incorrectly, or for using too many resources, or because it is not needed any more. Perhaps the parent is exiting, and it is appropriate to terminate child processes that have been working on the same

task as the parent. Perhaps the OS has a rule that does not allow user processes to exit while leaving descendant processes active.

* The interaction between a parent and child process is an example of interprocess communication. The nature of the interaction is more complicated than it may seem to be at first. Suppose a unix parent spawns a child and waits for it, and suppose the child executes and then exits. Because the parent and child are separate processes that execute concurrently for some amount of time, the child process MAY exit AFTER the parent waits, or the parent MAY wait AFTER the child exits.

* Therefore the OS has to do something to make sure that it saves the exit status of a child process until it can be delivered to a parent who has not yet waited for it.

* When a child process calls exit(), the OS deallocates most of the resources of the child. However the OS does NOT deallocate the entry of the child process in the system "process table" (which contains the exit status) until the parent (or, in case the parent itself has already exited, init) makes the corresponding call to wait().

* In unix parlance, a *zombie* is a process that has exited before any process has waited for it.

* A related term, *orphan*, is used to denote a child process whose parent never waits for it. Unix assigns the init process the task of waiting for orphaned processes.

SECTION 3.4 - Interprocess Communication

It can be convenient to create multiple processes to cooperate on work to be done.

Cooperating processes can
* share information,
* speed up computations, and
* make programs more modular

* Cooperating processes have to communicate, and, as we've seen, the two fundamental modes of interprocess communication are message passing and shared memory.

* Message passing typically requires the use of system calls, which tends to make it less efficient than communication with shared memory.

* However shared memory communication on some multicore systems appears to

suffer from cache coherence problems to an extent that message passing is actually more efficient on these systems.

3.4.1 - Shared Memory Systems

* Solutions to ***"the bounded buffer problem"*** illustrate one simple way in which two processes can communicate using shared memory in order to cooperate on the solution to a problem.

* Section 3.4.1 describes the interaction of two processes, the ***producer*** and the ***consumer***. For example, the one process in a compiler could *produce* assembly code that is *consumed* by a separate assembly process.

* In the example of the text book, the producer and consumer share a region in the primary memory containing an array of 10 slots. This array is called ***the buffer***. The shared memory region also contains a constant that indicates the size of the buffer, as well as two variables ***in*** and ***out***, that are used as indices.

* The producer and consumer execute code that interoperates in a manner that ensures (assuming 'atomic' reads/writes of integers) that neither process gets ahead of the other - in other words, the producer will not overwrite data in the buffer before the consumer has read it, and the consumer will not re-read any locations in the buffer before the producer has put new data into them.

3.4.2 - Message Passing Systems

* In a message passing system, "send" and "receive" commands have to be made available to communicating processes, and communication "links" of some kind have to be implemented.

* 3.4.2.1 - Naming

  + With ***direct communication***, a send command has the intended recipient as a parameter, and the receive command has the sender as a parameter. For example, send (P, M) would be a command to send the message M to process P.

  + With ***indirect communication***, processes send to and receive from ***mailboxes*** or ***ports***. Processes can use more than one mailbox at various times, so this kind of scheme is considered more flexible. However, care must be taken to insure that messages are received by the intended recipients.

The OS has to support creation and deletion of mailboxes, as well as operations that send to and receive from mailboxes.

* 3.4.2.2 - Synchronization

Send and receive operations can be blocking (synchronous) or non-blocking (asynchronous)

+ With a blocking send, the sending process is blocked (suspended) until the message is received - similar to how a process is suspended while waiting for I/O to complete.

+ With a non-blocking send, the process sends the message and continues execution immediately.

+ With a blocking receive, the process requests a message and blocks until the message is available.

+ With a non-blocking receive, the process returns from the receive request immediately, either with or without having gotten a message. A special NULL message denotes that no message was available.

+ We can create a solution to the bounded buffer problem by programming the producer to use a blocking send, and the consumer to use a blocking receive.

* 3.4.2.3 - Buffering

+ Depending on implementation, sent messages may be queued to wait for receivers. There may or may not be a declared bound on the length of the queue.

SECTION 3.5 - Examples of IPC Systems

3.5.1 - An Example: POSIX Shared Memory

* Shared memory-mapped files
  + One process creates a shared-memory object
  + Other processes that want to use the object must open it.

3.5.2 - An Example: Mach

* Mach uses messages for most communication - even system calls.
* Messages have a specific format, including the names of the sending and receiving mailboxes.
* A process can make a requests to receive a message from any one of a group of mailboxes.
* A source of overhead is double copying of messages from sender's memory to system memory and then from system memory to receiver's memory. In some cases the system can work around the

problem by re-mapping parts of memory.

3.5.3 - An Example: Windows

* The Windows kernel has a message-passing facility called ***advanced local procedure call (ALPC)***
* ALPC uses ports similar to the Mach ports (mailboxes).
* Some shared memory techniques are used to pass larger messages.

SECTION 3.6 - Communication in Client-Server Systems

* Sockets
* Remote Procedure Calls
* Pipes

3.6.1 - Sockets

* A socket is a type of data structure used as an endpoint of communication - for example in computer networks.
* However processes on the same computer can also use sockets to communicate.
* There are socket operations that implement client-server interactions.
* Each socket is assigned an internet protocol (IP) address and a port number. The IP address specifies the connection (of the computer) to the network, and the port number specifies the process using the socket. No two sockets have both the same IP address and the same port number.
* Processes can use sockets in a connection oriented mode, or in a connectionless mode.
* After initializing, socket communication works pretty much the same as file I/O.

3.6.2 - Remote Procedure Call

* RPC is a mechanism for allowing a process on a local host to make a function call that is executed on a remote host.
* The client on the local host sends a message to an RPC server on the remote machine, which identifies the function desired, and furnishes any needed parameters. The server arranges for the function to be called, and passes any needed results back to the client by sending a message.
* RPC is implemented to appear to user programs to be no different than an ordinary function call. Stub code in libraries takes care of connecting to the remote server and 'marshaling' parameters, using external data representation (XDR). Similar stubs on the server side respond to the RPC request and invoke execution of the function on the server host.

3.6.3 - Pipes

* Think of pipes as conduits for passing information. For example one process can put information into a pipe,

and another process can extract that information. We can think of the processes as being stationed at the two 'ends' of the pipe.

3.6.3.1 - Ordinary Pipes

* Ordinary pipes allow information to flow in one direction only. They support the kind of interaction required by a pair of processes with producer-consumer relationship.
* In unix, pipes are considered to be a special kind of file, and the system call interface is very similar to the one for files.
* Ordinary unix pipes can only be shared by related processes, such as a parent and its child.
* Windows has anonymous pipes that are about the same as unix ordinary pipes.

3.6.3.2 - Named Pipes

* Named pipes exist in unix and Windows systems. Processes that are not related are able to utilize them, and they can support bidirectional communication.
* A named pipe is also called a FIFO in unix terminology. Like ordinary pipes, FIFOs are considered and treated as special kinds of files. A FIFO, unlike an ordinary pipe, can exist after the process that created it has exited. A FIFO is visible in the file system. Any process with the required permissions can read from it or write to it.
* Unix FIFOs are sometimes used to support logging. A process can add a line to a system log by writing to a FIFO.
* Unix FIFOs can not be used for communication between processes on different machines. Sockets are required for that.
* Unix FIFOs support bidirectional information flow, but only one direction at a time.
* Windows named pipes do support simultaneous bidirectional information flow (this is called full duplex communication).
* Processes on different machines can communicate using a Windows named pipe.
* Windows named pipes support certain structured message types, whereas unix FIFOs just work on the level of transmitting individual bytes.