Report

Processes

- What are processes and how they work?

29 Nov, 2016

Semester: Autumn 2016
Course: Operating Systems
Authors: Sarpreet Singh Buttar, Phillip Lunyov, Jusuf Omerovic, Robin Reijo & Weibin Yu

Contents

1 Introduction

In the old days, computers could only execute one program at a time and this program got complete control over the system. But now multiple programs can be concurrently executed by the modern computer system. This transformation resulted in a process, which is the unit of work in a modern time-sharing system. The main responsibility of operating system is the execution of the user program. So if the operating system is very complex it is expected to do more on the behalf of its users. A system contains a collection of processes such as system code executed by operating system processes whereas user code is executed by user processes. These processes are executed concurrently with one or more CPUs. We can also make the computer more productive by switching the CPU between the processes. So, the process concept is all about what the processes are and how they work?

2 Process Concept

An operating system executes various activities such as batch system is responsible for jobs or process whereas time shared system executes user programs. Apart from executing single or multiple user programs, the operating system also support its own internal activities such as memory management. All the above activities are know as processes. However, in past these activities were usually called jobs.[1]

2.1 The Process

The process is not only the program code (text section) but also includes the current activity, represented by the program counter's values and the processor's register's contents. It also usually includes the process stack, containing temporary data and a data section containing global variables. It also include a heap, that is dynamically distributed memory during process run time. [1]
A program by itself is not a process, but rather a passive entity, like an executable file containing instructions stored on disk. A process on the other hand is an active entity. A program counter specifies the next instruction to carry out and a set of associated resources. When an executable file is loaded into memory, a program becomes a process. Loading executable files can be done by double-clicking an icon of the executable file as well as entering the name of the executable file on the command line. [1]
Even though two processes can be connected to the same program, they are still considered two separate execution sequences. For example, the users may run different copies of the web browser program. These are all separate processes where the data, heap and stack sections vary. The process can be an execution environment for other code, for example the Java programming environment. An executable Java program is most often executed within the Java virtual machine (JVM). This is a process that interprets the loaded Java cord and takes actions on behalf of that code.[1]

2.2 Process State

A process changes its state during the execution[1]. Following are the possible states for a process to be in:

- When a process is being created its state is NEW.
- During the execution of instructions its state is RUNNING.
- While waiting for the occurrence of other event such as I/O its state is WAITING
- While waiting for its turn for being assigned to a processor its state is READY
- When execution has been finished its state is TERMINATED

2.3 Process Control Board (PCB)

Processes are represented in the operating system by a process control block (PCB), also called task control block. Following information is included in the PCB regarding a specific process:

- State of the process.
- A program counter which points the next instruction to be executed.
- CPU registers which include stack pointers, accumulators and condition code information. It also help the process to continue correctly after the occurrence of interrupt.
- Information related to the process priority and pointers addressing the scheduling queues.
- Information related to the limit of register and page tables. This information is dependent on the memory system used by operating system.
- Information related to the time limits, amount of CPU and real time usage and so on.
- A list of I/O devices allocated to the process.

In other words, it saves all the necessary information related to the process. This information may vary from process to process. [1]

2.4 Threads

So far it's been implied that a process performs a single thread of execution with a single thread of instructions being executed. This thread allows the process to only do one task at a time. Usually the modern operating systems have extended the process, allowing it to have multiple threads which makes it possible to perform more than one task simultaneously. This is a benefit for multicore systems where multiple threads can run in parallel. The PCB is then expanded to include information for each thread.[1]

3 Process Scheduling

This section is about Process scheduling. Process scheduling is use to meet the objective of maximize CPU utilization(CPU have some process running at all times) and the objective to let users interact with each program while it is running(by switch CPU among processes frequently). In order to do that, it selects an available process for program execution on the CPU. In single-processor system only one process can run at a time and other processes must wait until the CPU is free then Process scheduling select a new process for the CPU.

3.1 Scheduling Queues

Scheduling queues refer to these queues:

- Job queue - A job queue consists of all processes in the system and when a new process enter the system, it will be added to this queue.
- Ready queue - Processes that are in the main memory and are ready and waiting for execution are stored in the ready queue. Ready queue is generally stored as a linked list.
- Device queue - This queue is for processes which make I/O request for a device and needs to wait until the device to handle its request is stored in this queue.

All processes in the ready queue is waiting to get select to execute or dispatch. When the process get select from ready queue it will be allocated to a CPU and begin executing. If it make a I/O request it will be placed into device queue and changes to waiting state. If it creates a new child process it may need to wait for the child's termination(Some process may run with the child in parallel) and before that it will in waiting state. If it get interrupt it will be put back to ready queue. And those two with waiting state will also go back to ready state and be put to the ready queue eventually. The figure that follows here is a queueing diagram and it represents the flow of processes in the system. Processes continues this cycle until it is terminated then it is removed from all queues, its PCB and resources will be deallocated.

3.2 Schedulers

The purpose of schedulers are to select process to execute. There are different kinds of schedulers:

- Long-term scheduler/job scheduler- selects processes from mass-storage device(a disk.they need to do this because all process submitted can't be executed immediately so it has to store somewhere) and loads them into memory to wait for execution.
- Short-term scheduler/CPU scheduler- selects process from processes that are ready to execute and allocates the CPU to it.
- Medium-term scheduler-it removes process from memory and active contention for the CPU and then put back to the memory, the process will execute from where it left off. This scheme is called swapping.
- Medium-term scheduler-it removes process from memory and active contention for the CPU and then put back to the memory, the process will execute from where it left off. This scheme is called swapping.

Long and short term scheduler differs in the way of frequency them execute. Short term scheduler must be fast to decide which process to execute otherwise it will waste CPU. Long term scheduler can take more time to decide which process to be add to memory since it controls the degree of multiprogramming(number of process in memory). To make the degree of multiprogramming stable long term scheduler need to be invoked every time a process leaves the system and add a new process to the memory. The selection must be done carefully by long-term scheduler since we want a good mix of I/O bound(spends more time doing I/O) and CPU bound(spend more time doing computations)processes.Also some OS uses medium term scheduler to reduce the degree of multiprogramming. Swapping can help improve the I/O and CPU bound process mix and freed up memory when a change in memory requirements has over-committed available memory.

3.3 Context Switch

Context switch is when you switch to another process and you need to do a state save(state save is when you save the current context of the process running on the CPU. The context is represented in the PCB of the process and it includes value of CPU registers, process state and memory-management information.) for the current process and a state restore(state restore is to resume operations) for the another process. Time needs to perform context switch differs from machine to machine. It depends on memory speed, number of registers that must be copied and the existence of special instruction. But it is typical a few milliseconds.

4 Operations on Processes

During the course of execution, a process may create several new processes. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes.

4.1 Process Creation

Most operating systems (including UNIX, Linux, and Windows) identify processes according to a unique process identifier (or pID), which is typically an integer number. It is an unique value for each process and can be used as index to allocate system resources. In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. A child process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children.
Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many child processes. Also the parent process must provide an initialization data for child process. When main process creates a child process, two ways of executions might be:

- Parent process continues to execute concurrently with its children
- Parent process waits until child processes finish their tasks

Two address-space possibilities for the new process:

- Child process is a duplicate of the parent process
- Child process has a new program loaded into it

4.2 Process Termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call. At that point, the process may return a status value (typically an integer) to its parent process (via the wait() system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system (termination can occur in other circumstances as well). Reason to terminate the execution of the child processes:

- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some systems do not allow a child to exist if its parent has terminated, so the solution is cascading termination - a phenomenon which is used in OS to terminate all child processes if the parent process is terminated and it is usually initiated by OS.
A process that has terminated, but whose parent has not obtained the exit status of the child, is known as a zombie process.Once the parent obtains the exit status of the child, the process identifier of the zombie process and its entry in the process table are released.
There could be situations then parent process has not obtained the exit code of child processes and terminated instead, leaving child processes as orphans. In Linux/Unix-based OS for this type of scenario, the "init"-process assigns child processes to himself, waiting for orphans to provide exit codes and afterwards deallocate orphans from OS.

5 Interprocess Communication

Interprocess communcation(IPC) means that we are using cooperating processes:

- cooperating process means that the process can be affected by other process, because it might share data or similar things. Cooperating processes uses inter process communication among the processes that allows them to exchange data and information among each other. Inter process communication therefore uses two core models - Message passing and shared memory. Shared memory means that processes has established a shared area for the data and information they share. An example of when shared memory is using is given further down. Message passing means that the processes communicate through sending messages to each other.
- independent process is when a process cant be affected by other processes, for example if the process doesn't share any data with another processes then it is independent.

When are cooperating processes beneficial?

- When sharing information, because we might have a lot of users that wants to get to the same information, for example, perhaps a database? or just a file that they all share, then we need a technique how they all can enter the same file/database at the same time, this is shared data.
- Faster computations of a task: We can do this by dividing the task into sub tasks, and this where multi core processors come into play. To be able to divide the task into these sub tasks we will need multi core processors.
- Beneficial because we want to break down the system in modules. For example, a system functionality should be handled by separate processes and not just one.
- It is also very very convenient, we wanna be able to play a game, listen to some music, perhaps have a stream on in the background, all the same time.

5.1 Shared-Memory Systems

Like written earlier, the shared system means that processes share an address, so lets make an example to get a better understanding:
Lets assume we have a producer that creates/produces objects, then we have a consumer that uses these objects. Then we have a buffer that is filled up by the producer and consumed by the consumer, resulting in that the shared memory here is the buffer. This means that buffer can produce an item and at the same consume an item, the shared memory area will be area that the producer and consumer shares. The buffer also has to synchronize the operation of consumer and producer, so that we do not start consuming an item before it has been produced.

5.2 Message-Passing Systems

Message passing like written earlier means that processes communicate through sending messages to each other, to make this more clear, lets make an example:
Lets assume we are connected to a chat room. This means that we have a sender and receiver. The sender being the one that types and send the message and a the receiver that gets the message. This also means that the processes will be on each of these computers. A link is established between these two processes. The link can be implemented in the following ways:

- Indirect or direct communication
- sync or async communication
- Automatic or explicit buffering

5.2.1 Naming

When the processes in message passing wants to communicate with each it has to know something about the processes to know that that specific is the process to communication. So in the direct communication we use names for the processes.For example, S might be the name for the sender and R might be the receiver. The link will be established when:

- When a pair of processes wants to communicate, they only need to know each others names(identities). The link will only be between those specific pairs of processes. Only one link exists. This is therefore a symmetric addressing. Another scheme is the asymmetric, where only the sender knows the receivers identification, but not the way around. Sender for example is named S and then we have an ID for the receiver. The negative thing is that both symmetric and asymmetric has limits when it comes to changing the modularity of the process result definitions. For example, if a process identifier changes then we might have to look at all the other process definitions. Then we also need to find the connections from the old identifier so that they can be connected and modified correctly to the new identifier. This means that we are hard coding, which would not be the best, the identifiers must be clearly stated and more wanted to than this technique.

- Using indirect communication the message will be sent and received using mailboxes and ports. Mailboxes is similarly to reality, in that the box can receive and then store the messages, and of course also removable. Every mailbox has a unique ID. For two processes to communicate they need to share a mailbox, this of course means that processes can communicate over several mailboxes as long as they share them. So to clearly state how it works: We have a sender(a), that sends his message to mailbox(a), then we have a receiver(a) that picks up the message from the mailbox(a). The link in this scheme is also therefore different then direct communication. A link in indirect communication will arrive if the processes has a shared mailbox.

5.2.2 Synchronization

Messages passing can either be synchronous or asynchronous. Synchronous means that the sending process will be blocked until a message is received from the receiving process or if it is from the mailbox. Also it might block the receiver until the message is ready. Asynchronous means that a sending process sends messages and also resumes progress. It also means that the receiver gets valid messages or null ones.

5.2.3 Buffering

Since the processes will be stored in temporary queues, using buffers. There are three ways to these queues can be implemented through, zero capacity, zero length, cant have messages waiting. Bound capacity, has a limited amount of messages in it. Unbound capacity, it has a probability of being infinite, can have any number in it and sender therefore never blocks.

6 Examples of IPC Systems

In this section, we will go over three relatively distinct IPC systems:

- POSIX API using shared memory
- Message sharing in the Mach operating system
- Shared memory as a mechanism of message passing in Windows

6.1 POSIX Shared Memory

POSIX makes use of shared memory objects to share data between processes, using shm_open() with parameters for conditional creation, read/write access and permissions. The method subsequently returns an integer file descriptor for the shared memory object. After the shared memory object is created, the object's size is then configured using ftruncate() and written to a memory mapped file using mmap() that returns a pointer to the memory-mapped file. Different design models can be applied to the shared-memory object and memory-mapped file, such as the producer-consumer model, in which a producer creates or establishes a shared-memory object and the consumer reads from the shared memory. In order for other processes than the producer to access the file and thus make changes to it, the memory-mapped file has to be flagged with MAP_SHARED and after every write, we must increment the pointer of the memory-mapped object with the number of bytes written.
In order to ensure multiple threads or processes do not access the same shared-memory object simultaneously, POSIX makes use of Mutex, or Mutual Exclusion, where a lock is effectively placed on the object while one process is accessing it. POSIX does this via Mutex variables, and pthread_mutex_trylock() and pthread_mutex_unlock().

6.2 Message sharing in Mach

The Mach kernel makes use of a different way to share data between tasks, using Mailboxes (ports). This includes system calls and most inter task communications. New ports are allocated using the system call port_allocate() and both allocates space for the port and its queue of messages, which by default is eight messages. The task that creates the port is the port's owner. Only one task at a time can either own or receive from a port, and these rights can be sent to other tasks. Initially, the port's message queue is empty, and as messages are sent, they are copied to the port with the same priority, ensuring that multiple messages from the same sender are queued using FIFO order, but absolute ordering is not a guarantee and messages from multiple senders can be queued in any order. Every message consists of a fixed-length header followed by a variable-length data portion, which is indicated in the header. Besides the length of the message, the header also carries two port names, one being the name of the receiving port and the other the sending port. If the sending task or thread expects a reply, the sender port is used as a "return address". The variable-length data portion of the message consists of a list of typed data items, carrying the type, size and value. If a port queue is full when a task is trying to send a message, there are four options for the sender task:

- wait indefinitely until the receiving port can receive the message
- wait for n milliseconds
- do not wait, discard the message and return
- Cache the message. In this case, the operating system holds the message and the sending task receives a message when the message is delivered to the port.

The last option is specifically meant for server tasks, because the sending task might not be able to wait, but instead have to serve other service requests.

6.3 Message sharing using shared memory in Windows

Windows utilizes a combination of shared memory and messages for inter-process communications and for communications through the subsystems that a windows system consists of. The message-passing facility in Windows is called the advanced local procedure call (ALPC) facility. Like Mach, windows utilizes ports to establish and maintain connections between processes and utilizes two types of ports: connection and communication ports. Server processes publish the

connection-port objects visible to all processes, and when a client wants service from a subsystem it opens a handle to the connection-port object and sends a connection request to that port, to which the server responds with a connection handle and opens a channel. The channel itself consists of two private communication ports: one for the client and one for the server. When an ALPC channel is created, one of three message-passing techniques is chosen:

- For small messages (>=256 bytes), the port's message queue is used as intermediate storage and messages are copied from one process to the other.
- Larger messages are passed through a section object, a shared memory object associated with the channel.
- If the amount of data is too large to fit into a section object, the server process reads and writes directly into the address space of the client using an API.

The client must decide on creation of the ALPC channel if it will need to send any large messages, and if so, it has to request a section object to be created. Similarly, if the server decides requests will be too large, it creates a section object. In order for the section object to be used, a small message is sent to communicate the pointer to and the size of the section object.

7 Communication in Client-Server Systems

7.1 Sockets

A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to. Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.
On the client-side: The client knows the host-name of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.
If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.
On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets.[2]
Communication using sockets—although common and efficient—is considered a low-level form of communication between distributed processes. One reason is that sockets allow only an unstructured stream of bytes to be exchanged between the communicating threads. It is the responsibility of the client or server application to impose a structure on the data.

7.2 Remote Procedure Calls

By replacing dedicated protocols and communication methods with a robust and standardized interface, RPC is designed to facilitate communication between client and server processes. The functions contained within RPC are accessible by any program that must communicate using a client/server methodology.[3]
The messages exchanged in RPC communication are well structured and are thus no longer just packets of data. Each message is addressed to an RPC daemon listening to a port on the remote

system, and each contains an identifier specifying the function to execute and the parameters to pass to that function. The function is then executed as requested, and any output is sent back to the requester in a separate message. Common issues and their solutions using RPC:

- To solve issue of data representation on client and server machines (big-endian/little endian systems), many RPC systems define a machine-independent representation of data, such as XDR (external data representation).
- The semantics of a call (RPCs can fail, or be duplicated and executed more than once, as a result of common network errors.) - One way to address this problem is for the operating system to ensure that messages are acted on exactly once, rather than at most once.
- The binding problems (solution: fixed port address in compiled code or matchmaker daemon which defines port for RPC)

7.3 Pipes

A pipe acts as a conduit allowing two processes to communicate. Pipes were one of the first IPC mechanisms in early UNIX systems. They typically provide one of the simpler ways for processes to communicate with one another, although they also have some limitations.

7.4 Ordinary Pipes

Ordinary pipes allow two processes to communicate in standard producer–consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. To achieve full-duplex (send/receive transmission) communication, you need to create two pipes. An ordinary pipe cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process.
Ordinary pipes on Windows systems are termed anonymous pipes, and they behave similarly to their UNIX counterparts (Windows requires the programmer to specify which attributes the child process will inherit, it is necessary to prohibit the child from inheriting the write-end of the pipe). Ordinary pipes are used only locally, both in Unix-based and Windows machines.

7.5 Named Pipes

Named pipes provide a much more powerful communication tool. Communication can be bidirectional, and no parent–child relationship is required. Once a named pipe is established, several processes can use it for communication. In addition continue to exist after communicating processes have finished. Both UNIX and Windows systems support named pipes, although the details of implementation differ greatly. Unix named pipes are often referred as FIFO pipes. Properties of FIFO:

- appears like a typical file
- continues to exist if not deleted completely from file system
- one pipe can be used both for sending/receiving information, but only half-duplex is allowed works only locally (on the same machine, If inter-machine communication is required, sockets must be used)
- only byte-oriented data may be transmitted

Properties of Windows-based named pipes:

- full-duplex communication

- can be used locally and externally
- byte- or message-oriented data can be transmitted

8 Summary

A process is a program in execution, process have five different states:new, ready, running, waiting, or terminated. State is defined by process's current activity. Every process is represented in the operating system by its own PCB.
A process is placed in some waiting queue when it is not executing. There are two major classes of queues in an operating system: I/O request queues and the ready queue. Processes that are ready to execute and are waiting for the CPU is placed in ready queue if it is waiting for I/O then it is in I/O queue.
The operating system must select processes from various scheduling queues. Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory management. Short-term (CPU) scheduling is the selection of one process from the ready queue.
Operating systems must provide a mechanism for parent processes to create new child processes. The parent wait for its children to terminate before proceeding or they run concurrently. Reasons for allowing concurrent execution are information sharing, computation speedup, modularity, and convenience.
The processes executing in the operating system are either independent processes or cooperating processes. Cooperating processes require an interprocess communication mechanism to communicate with each other. Principally, there are two ways to achieve communication: shared memory and message passing. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application programmers; the operating system needs to provide only the shared memory. The message-passing method allows the processes to exchange messages. The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.
Communication in client–server systems may use sockets, remote procedure calls (RPCs), or pipes. A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application. Pipes provide a relatively simple ways for processes to communicate with one another. Ordinary pipes allow communication between parent and child processes, while named pipes permit unrelated processes to communicate.

References

[1] Abhraham S., Peter B. G., Greg G. *Operation Systems Concepts*, Vol 9, Chapter - 3 (2013).

[2] Oracle. *What is Socket?* [Online] Available:
https://docs.oracle.com/javase/tutorial/networking/sockets/definition.html

[3] TechNet (2003). *How RPC Works*. [Online] Available:
https://technet.microsoft.com/en-us/library/cc738291(v=ws.10).aspx

NOTE: All the report is referenced to [1].

Contribution		
Section	Written By	Presented By
Introduction	Sarpreet Singh Buttar	Sarpreet Singh Buttar
Process Concept	Sarpreet Singh Buttar	Sarpreet Singh Buttar
Process Scheduling	Weibin Yu	Sarpreet Singh Buttar
Operations on Processes	Phillip Lunyov	Robin Reijo
InterProcesses Communication	Robin Reijo	Robin Reijo
Example of IPC Systems	Jusuf Omerovic	Weibin Yu
Communication in Client - Server Systems	Phillip Lunyov	Weibin Yu
Summary	Weibin Yu	Weibin Yu