Process (computing)

From Wikipedia, the free encyclopedia

In computing, a **process** is an instance of a computer program, consisting of one or more threads, that is being sequentially executed[1] by a computer system that has the ability to run several computer programs concurrently.

A computer program itself is just a passive collection of instructions, while a process is the actual execution of those instructions. Several processes may be associated with the same program; for example, opening up several instances of the same program often means more than one process is being executed. In the computing world, processes are *formally* defined by the operating system (OS) running them and so may differ in detail from one OS to another.

A single computer processor executes one or more (multiple) instructions at a time (per clock cycle), one after the other (this is a simplification; for the full story, see superscalar CPU architecture). To allow users to run several programs at once (e.g., so that processor time is not wasted waiting for input from a resource), single-processor computer systems can perform time-sharing. Time-sharing allows processes to switch between being executed and waiting (to continue) to be executed. In most cases this is done very rapidly, providing the illusion that several processes are executing 'at once'. (This is known as concurrency or multiprogramming.) Using more than one physical processor on a computer, permits **true** simultaneous execution of more than one stream of instructions from different processes, but time-sharing is still typically used to allow more than one process to run at a time. (Concurrency is the term generally used to refer to several independent processes sharing a single processor; simultaneously is used to refer to several processes, each with their own processor.) Different processes may share the same set of instructions in memory (to save storage), but this is not known to any one process. Each execution of the same set of instructions is known as an *instance*— a completely separate instantiation of the program.

For security and reliability reasons most modern operating systems prevent direct communication between 'independent' processes, providing strictly mediated and controlled inter-process communication functionality.

Contents

Sub-processes and multi-threading

Main article: Thread (computer science)

A process may split itself into multiple 'daughter' sub-processes or threads that execute in parallel, running different instructions on much of the same resources and data (or, as noted, the same instructions on logically different resources and data).

Multithreading is useful when various 'events' are occurring in an unpredictable order, and should be processed in another order than they occur, for example based on response time constraints. Multithreading makes it possible for the processing of one event to be temporarily interrupted by an event of higher priority. Multithreading may result in more efficient CPU time utilization, since the CPU may switch to low-priority tasks while waiting for other events to occur.

For example, a word processor could perform a spell check as the user types, without "freezing" the application - a high-priority thread could handle user input and update the display, while a low-priority background process runs the time-consuming spell checking utility. This results in that the entered text is shown immediately on the screen, while spelling mistakes are indicated or corrected after a longer time.

Multithreading allows a server, such as a web server, to serve requests from several users concurrently, thus avoiding unheard requests when the server is busy with a processing request. One simple solution to that problem is one thread that puts every incoming request in a queue, and a second thread that processes the requests one by one in a first-come first-served manner. However, if the processing time is very long for some requests (such as large file requests or requests from users with slow network access data rate), this approach would result in long response time also for requests that do not require long processing time, since they may have to wait in queue. One thread per request would reduce the response time substantially for many users and may reduce the CPU idle time and increase the utilization of CPU and network capacity. In case the communication protocol between the client and server is a communication session involving a sequence of several messages and responses in each direction (which is the case in the TCP transport protocol used in for web browsing), creating one thread per communication session would reduce the complexity of the program substantially, since each thread is an instance with its own state and variables.

In a similar fashion, multi-threading would make it possible for a client such as a web browser to communicate efficiently with several servers concurrently.

A process that has only one thread is referred to as a *single-threaded* process, while a process with multiple threads is referred to as a *multi-threaded* process. Multi-threaded processes have the advantage over multi-process systems that they can perform several tasks concurrently without the extra overhead needed to create a new process and handle synchronised communication between these processes. However, single-threaded processes have the advantage of even lower overhead.

[2]

Representation

In general, a computer system process consists of (or is said to 'own') the following resources:

- An *image* of the executable machine code associated with a program.
- Memory (typically some region of virtual memory); which includes the executable code, process-specific data (input and output), a call stack (to keep track of active subroutines and/or other events), and a heap to hold intermediate computation data generated during run time.
- Operating system descriptors of resources that are allocated to the process, such as file descriptors (Unix terminology) or handles (Windows), and data sources and sinks.
- Security attributes, such as the process owner and the process' set of permissions (allowable operations).
- Processor state (context), such as the content of registers, physical memory addressing, etc. The *state* is typically stored in computer registers when the process is executing, and in memory otherwise.[2]

The operating system holds most of this information about active processes in data structures called process control blocks (PCB).

Any subset of resources, but typically at least the processor state, may be associated with each of the process' threads in operating systems that support threads or 'daughter' processes.

The operating system keeps its processes separated and allocates the resources they need so that they are less likely to interfere with each other and cause system failures (e.g., deadlock or thrashing). The operating system may also provide mechanisms for inter-process communication to enable processes to interact in safe and predictable ways.

Process management in multi-tasking operating systems

Main article: Process management (computing)

A multitasking* operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one time on a single-core CPU (unless using multi-threading or other similar technology).[3]

It is usual to associate a single process with a main program, and 'daughter' ('child') processes with any spin-off, parallel processes, which behave like asynchronous subroutines. A process is said to *own* resources, of which an *image* of its program (in memory) is one such resource. (Note, however, that in multiprocessing systems, *many* processes may run off of, or share, the same reentrant program at the same location in memory— but each process is said to own its own *image* of the program.)

Processes are often called *tasks* in embedded operating systems. The sense of 'process' (or task) is 'something that takes up time', as opposed to 'memory', which is 'something that takes up space'. (Historically, the terms 'task' and 'process' were used interchangeably, but the term 'task' seems to be dropping from the computer lexicon.)

The above description applies to both processes managed by an operating system, and processes as defined by process calculi.

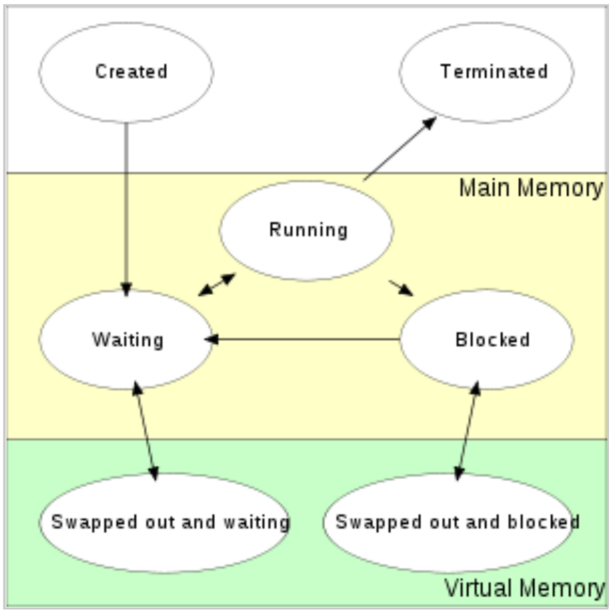
If a process requests something for which it must wait, it will be blocked. When the process is in the Blocked State, it is eligible for swapping to disk, but this is transparent in a virtual memory system, where blocks of memory values may be really on disk and not in main memory at any time. Note that even *unused* portions of active processes/tasks (executing programs) are eligible for swapping to disk. *All parts of an executing program and its data do not have to be in physical memory for the associated process to be active.*

*Tasks and processes refer essentially to the same entity. And, although they have somewhat different terminological histories, they have come to be used as synonyms. Today, the term process is generally preferred over task, *except* when referring to 'multitasking', since the alternative term, 'multiprocessing', is too easy to confuse with multiprocessor (which is a computer with two or more CPUs).

Process states

Main article: Process states

An operating system kernel that allows multi-tasking needs processes to have certain states. Names for these states are not standardised, but they have similar functionality.[2]



The various process states, displayed in a state diagram, with arrows indicating possible transitions between states.

- First, the process is "created" - it is loaded from a secondary storage device (hard disk or CD-ROM...) into main memory. After that the process scheduler assigns it the state "waiting".
- While the process is "waiting" it waits for the scheduler to do a so-called context switch and load the process into the processor. The process state then becomes "running", and the processor executes the process instructions.
- If a process needs to wait for a resource (wait for user input or file to open ...), it is assigned the "blocked" state. The process state is changed back to "waiting" when the process no longer needs to wait.
- Once the process finishes execution, or is terminated by the operating system, it is no longer needed. The process is removed instantly or is moved to the "terminated" state. When removed, it just waits to be removed from main memory.[2][4]

Inter-process communication

Main article: Inter-process communication

When processes communicate with each other it is called "Inter-process communication" (IPC). Processes frequently need to communicate, for instance in a shell pipeline, the output of the first process need to pass to the second one, and so on to the other process.It is preferred in a well-structured way not using interrupts.

It is even possible for the two processes to be running on different machines. The operating system (OS) may differ from one process to the other, therefore some mediator(s) (called protocols) are needed.

History

See also: History of operating systems

By the early 60s computer control software had evolved from Monitor control software, e.g., IBSYS, to Executive control software. Computers got "faster" and computer time was still neither "cheap" nor fully used. It made multiprogramming possible and necessary.

Multiprogramming means that several programs run "at the same time" (concurrently). At first they ran on a single processor (i.e., uniprocessor) and shared scarce resources. Multiprogramming is also basic form of multiprocessing, a much broader term.

Programs consist of sequence of instruction for processor. Single processor can run only one instruction at a time. Therefore it is impossible to run more programs at the same time. Program might need some resource (input ...) which has "big" delay. Program might start some slow operation (output to printer ...). This all leads to processor being "idle" (unused). To use processor at all time the execution of such program was halted. At that point, a second (or n^{th}) program was started or restarted. User perceived that programs run "at the same time" (hence the term, *concurrent*).

Shortly thereafter, the notion of a 'program' was expanded to the notion of an 'executing program and its context'. The concept of a process was born.

This became necessary with the invention of re-entrant code.

Threads came somewhat later. However, with the advent of time-sharing; computer networks; multiple-CPU, shared memory computers; etc., the old "multiprogramming" gave way to true multitasking, multiprocessing and, later, multithreading.

See also

- Child process
- Exit
- Fork
- Orphan process
- Parent process
- Process group
- Process states
- Task
- Thread
- Wait
- Zombie process
- Process management (computing)

Notes

1. ^ Knott 1974, p.8
2. ^ *a b c d* SILBERSCHATZ, Abraham; CAGNE, Greg, GALVIN, Peter Baer (2004). "Chapter 4". *Operating system concepts with Java* (Sixth Edition ed.). John Wiley & Sons, Inc.. ISBN 0-471-48905-0.
3. ^ Some modern CPUs combine two or more independent processors and can execute several processes simultaneously - see Multi-core for more information. Another technique called simultaneous multithreading (used in Intel's Hyper-threading technology) can simulate simultaneous execution of multiple processes or threads.
4. ^ Stallings, William (2005). *Operating Systems: internals and design principles (5th edition)*. Prentice Hall. ISBN 0-13-127837-1.
   Particularly chapter 3, section 3.2, "process states", including figure 3.9 "process state transition with suspend states"

References

- Gary D. Knott (1974) *A proposal for certain process management and intercommunication primitives (http://doi.acm.org/10.1145/775280.775282)* ACM SIGOPS Operating Systems Review. Volume 8 , Issue 4 (October 1974). pp. 7 - 44

External links

Retrieved from "http://en.wikipedia.org/wiki/Process_(computing)"
Categories: Operating system technology | Concurrent computing | Process (computing)