
# Processes

Early computers allowed only one program to be executed at a time. This program had complete control of the system and had access to all the system's resources. In contrast, contemporary computer systems allow multiple programs to be loaded into memory and executed concurrently. This evolution required firmer control and more compartmentalization of the various programs; and these needs resulted in the notion of a **process**, which is a program in execution. A process is the unit of work in a modern time-sharing system.

The more complex the operating system is, the more it is expected to do on behalf of its users. Although its main concern is the execution of user programs, it also needs to take care of various system tasks that are better left outside the kernel itself. A system therefore consists of a collection of processes: operating-system processes executing system code and user processes executing user code. Potentially, all these processes can execute concurrently, with the CPU (or CPUs) multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive. In this chapter, you will read about what processes are and how they work.

## Bibliographical Notes

Process creation, management, and IPC in UNIX and Windows systems, respectively, are discussed in [Robbins and Robbins (2003)] and [Russinovich and Solomon (2009)]. [Love (2010)] covers support for processes in the Linux kernel, and [Hart (2005)] covers Windows systems programming in detail. Coverage of the multiprocess model used in Google's Chrome can be found at http://blog.chromium.org/2008/09/multi-process-architecture.html.

Message passing for multicore systems is discussed in [Holland and Seltzer (2011)]. [Baumann et al. (2009)] describes performance issues in shared-memory and message-passing systems. [Vahalia (1996)] describe interprocess communication in the Mach system.

The implementation of RPCs is discussed by [Birrell and Nelson (1984)]. [Staunstrup (1982)] discuss procedure calls versus message-passing communication. [Harold (2005)] provides coverage of socket programming in Java.

[Hart (2005)] and [Robbins and Robbins (2003)] cover pipes in Windows and UNIX systems, respectively.

## Bibliography

**[Baumann et al. (2009)]** A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, P. Simon, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems" (2009), pages 29–44.

**[Birrell and Nelson (1984)]** A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Volume 2, Number 1 (1984), pages 39–59.

**[Harold (2005)]** E. R. Harold, *Java Network Programming, Third Edition*, O'Reilly & Associates (2005).

**[Hart (2005)]** J. M. Hart, *Windows System Programming, Third Edition*, Addison-Wesley (2005).

**[Holland and Seltzer (2011)]** D. Holland and M. Seltzer, "Multicore OSes: looking forward from 1991, er, 2011" (2011), pages 33–33.

**[Love (2010)]** R. Love, *Linux Kernel Development, Third Edition*, Developer's Library (2010).

**[Robbins and Robbins (2003)]** K. Robbins and S. Robbins, *Unix Systems Programming: Communication, Concurrency and Threads, Second Edition*, Prentice Hall (2003).

**[Russinovich and Solomon (2009)]** M. E. Russinovich and D. A. Solomon, *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition, Fifth Edition*, Microsoft Press (2009).

**[Staunstrup (1982)]** J. Staunstrup, "Message Passing Communication Versus Procedure Call Communication", *Software—Practice and Experience*, Volume 12, Number 3 (1982), pages 223–234.

**[Vahalia (1996)]** U. Vahalia, *Unix Internals: The New Frontiers*, Prentice Hall (1996).