For this reason, the task switcher is carefully designed and tuned for performance.

Multitasking should not be confused with *multithreading*. In multitasking, an operating system allows a number of programs to run independent of each other. Although these tasks may be able to communicate through pipes and other interprocess communication methodologies, each task has a separate environment, including its own memory space and variables. Generally speaking, tasks can be started and stopped independent of each other.

In multithreading, on the other hand, a single program splits apart into two or more concurrently operating "threads," which run as parts of the same program, sharing memory space and variables. These threads can communicate among themselves in ways that separate programs cannot, including shared variables. Typically, threads cannot be stopped by anything other than the task that created them, and the operating system assigns CPU time and priority to all of the threads in a task as a single unit rather than giving each thread its own time slices.

Even simple single-tasking operating systems can perform some basic multitasking through the use of **interrupts**. In MS-DOS, for example, the hardware or firmware in an I/O controller can be given a task to perform, and it will interrupt the CPU when that task is complete. This allows such programs as print queues to run in the background while the foreground task interacts with the user.

FURTHER READING

Bach, Maurice J. *Design of the Unix Operating System.* Englewood Cliffs, N.J.: Prentice Hall, 1990.

Comer, Douglas, and Timothy V. Fossum. *Operating System Design.* Englewood Cliffs, N.J.: Prentice Hall, 1988.

Nutt, Gary J. *Kernel Projects for Linux.* Boston: Addison-Wesley, 2000.

Stallings, William. *Operating Systems: Internals and Design Principles,* 3rd ed. Upper Saddle River, N.J.: Prentice Hall, 1997.

Tanenbaum, Andrew. *Modern Operating Systems.* Englewood Cliffs, N.J.: Prentice Hall, 1992.

Tanenbaum, Andrew, and Albert S. Woodhull. *Operating Systems: Design and Implementation.* Englewood Cliffs, N.J.: Prentice Hall, 1987; 2nd ed., Upper Saddle River, N.J., 1997.

—*Gary Robson*

Multithreaded Architecture

A computer with a multithreaded architecture usually has a single processor that executes numerous instruction streams (**threads**), switching among them with the help of multiple register sets. One can think of a multithreaded processor as a parallel execution unit based on a single processor. In the late 1990s and early 2000s, several prototypes of multithreaded processors were built.

Computer architects are constantly looking for different ways to extract more performance from the "transistor budget" provided by modern VLSI (**very large scale integration**) chips. Since the 1980s, *pipelining*, which is a form of parallel execution, has been used and has become almost universal in all current processors. In a pipelined processor, the execution of each instruction is divided into several stages and the instruction goes sequentially through each. A common scheme, for example, consists of using five stages, such as instruction fetch (retrieving the instruction from memory), instruction decode, instruction execution proper, memory access, and write back (the result is stored in a register). Once the stages have been defined, they are isolated electrically from each other in such a way that when an instruction has gone through the first stage, a new one can be inserted into the pipeline, and so on. When the pipeline is full, up to five instructions are in the processor and have achieved different degrees of completion. Pipelined execution resembles assembly lines in manufacturing plants—every worker (every stage) receives a new piece of work as soon as the last one has been completed and passes the completed work on to the next worker in the line.

The advantage of pipelining is that instructions can be started faster since each stage is shorter than the full execution path. Under ideal conditions, a pipeline of five stages provides a fivefold performance gain over a similar sequential processor without pipelining. The challenge of pipelining is keeping the pipeline full with normal programs, which is very difficult. Conditional branches in the code can lead to a situation where the execution path has to be changed (retrieving instructions from another place in memory) after other instructions located below the branch have already been loaded into the pipeline. In this case the pipeline has to be *flushed*,

clearing and restarting it with instructions from the new instruction stream. *Data hazards* are another common problem. These are conflicts between instructions loaded back to back into the pipeline. It could be the case that the result of an instruction is needed by the next instruction, which has to stop the pipeline until the first instruction writes back its result into the registers, so that it can be used by the instruction waiting. Such waiting cycles lead to *pipeline bubbles*, in which no useful work is done in some pipeline stages.

An elegant solution to the problem of pipeline bubbles is to execute several threads simultaneously. Threads can be thought of as programs running in parallel in a computer. The name *thread* is used because a single program (e.g., the text processor) can start parallel activities (checking grammar at the same time that text is formatted); since these parallel pieces of code belong to the same program, they are called threads of execution of the mother program.

Consider again the example with a pipeline made of five stages, and assume that five threads are running. The first instruction for the pipeline could be taken from thread 1, the second instruction from thread 2, and so on, until the pipeline is full. Then the loading process is repeated cyclically (taking one instruction from each thread) each time an instruction is complete and abandons the pipeline. The advantage of this scheme is that a new instruction from a thread is fetched only after the previous instruction has finished executing. There are no ambiguities, no problems with conditional branches, and no conflicts between back-to-back instructions in the code.

There is, however, a drawback: Switching between the different threads can be done rapidly only if each thread uses a different set of registers to hold information temporarily in the processor. In our example, we would need five sets of, say, 32 registers each. Each thread can manage its 32 registers as desired. The processor then needs a grand total of 5 times 32 registers (i.e., 160 registers). Some other internal registers also have to be replicated: for example, the program counter, which keeps track of the position in memory of the next instruction to be executed.

Multithreaded processors can therefore keep the pipeline full and achieve the full performance gain of the pipelined processor, under the assumption that there are enough threads waiting for attention by the processor. This requires new compilation techniques, in order to make programs parallel automatically, even if the programmer wrote sequential code.

The type of multithreading described above is called *interleaved multithreading*. When a block of instructions (not a single instruction) is taken from each thread, this is called *block multithreading*. The processor can switch between threads synchronously, according to a clock, or asynchronously (i.e., only when a conflict would generate a pipeline bubble).

One example of a commercial multithreaded system is the one being built by Tera Computers, a supercomputing company. In the Tera architecture, the **central processing unit** switches context every 3 nanoseconds among 128 different threads. The machine can contain up to 256 processors. **Sun Microsystem**'s SPARC architecture, with up to 512 registers and several register windows, has also been used experimentally to implement multithreaded systems.

FURTHER READING

Bokhari, Shahid H., and Dimitri J. Mavriplis. *The Tera Multithreaded Architecture and Unstructured Meshes.* Hampton, Va.: Institute for Computer Applications in Science and Engineering, NASA Langley Research Center; Springfield, Va.: National Technical Information Service, distributor, 1998.

Iannucci, Robert A., et al. *Multithreaded Computer Architecture: A Summary of the State of the Art.* Boston: Kluwer Academic, 1994.

Moore, Simon W. *Multithreaded Processor Design.* Boston: Kluwer Academic, 1996.

—Raúl Rojas

Music, Computer

Music has seen some of the most creative and productive applications of computer technology thus far. Computer-based synthesizers have spawned previously unimaginable electronic orchestras; techniques such as **virtual reality** have blurred the distinction between composer and performer, mind and machine; and computer-inspired models have enabled psychologists to better understand how