COMPUTER MULTITASKING

n computing, **multitasking** is a method where multiple tasks, also known as processes, are performed during the same period of time. The tasks share common processing resources, such as

a CPU

and main memory RAM and ROM

In the case of a computer with a single CPU, only one task is said to be *running* at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved.

Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs.

Operating systems may adopt one of many different scheduling strategies, which generally fall into the following categories:

•In *multiprogramming* systems, the running task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a tape) or until the computer's scheduler forcibly swaps the running task out of the CPU. Multiprogramming systems are designed to maximize CPU usage.

•In *time-sharing* systems, the running task is required to relinquish the CPU, either voluntarily or by an external event such as a hardware interrupt. Time sharing systems are designed to allow several programs to execute apparently simultaneously. The expression 'time sharing' was usually used to designate computers shared by interactive users at terminals, such as IBM's TSO, and VM/CMS

•In *real-time* systems, some waiting tasks are guaranteed to be given the CPU when an external event occurs. Real time systems are designed to control mechanical devices such as industrial robots, which require timely processing.

Multithreading

As multitasking greatly improved the throughput of computers, programmers started to implement applications as sets of cooperating processes (e. g., one process gathering input data, one process processing input data, one process writing out results on disk). This, however, required some tools to allow processes to efficiently exchange data.

Threads were born from the idea that the most efficient way for cooperating processes to exchange data would be to share their entire memory space. Thus, threads are basically processes that run in the same memory context. Switching between threads does not involve changing the memory context.

While threads are scheduled preemptively, some operating systems provide a variant to threads, named *fibers*, that are scheduled cooperatively. On operating systems that do not provide fibers, an application may implement its own fibers using repeated calls to worker functions. Fibers are even more lightweight than threads, and somewhat easier to program with, although they tend to lose some or all of the benefits of threads on machines with multiple processors.