Threads

Objectives

- Threads
- Processes Vs Threads
- Why Threads?
- User-Level and Kernel-Level Threads
- Advantages and Disadvantages of Threads over Multiple Processes
- Context Switch

A thread is a single sequence stream within in a process.

They are sometimes called *lightweight processes*

Are popular way to improve application through parallelism

Needs its own stack as threads generally call different procedures and thus hold a different execution history.

Threads are not independent of one other like processes

Shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Threads

Processes vs. Threads

Similarities			
Like processes threads share CPU and only one thread active (running) at a time.	Like processes, threads within a processes, threads within a processes execute sequentially.	Like processes, thread can create children.	And like process, if one thread is blocked, another thread can run.

Differences

Unlike processes, threads are not independent of one another.	Unlike processes, all threads can access every address in the task .	Unlike processes, thread are designed to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Processes vs. Threads

A process with multiple threads make a great server

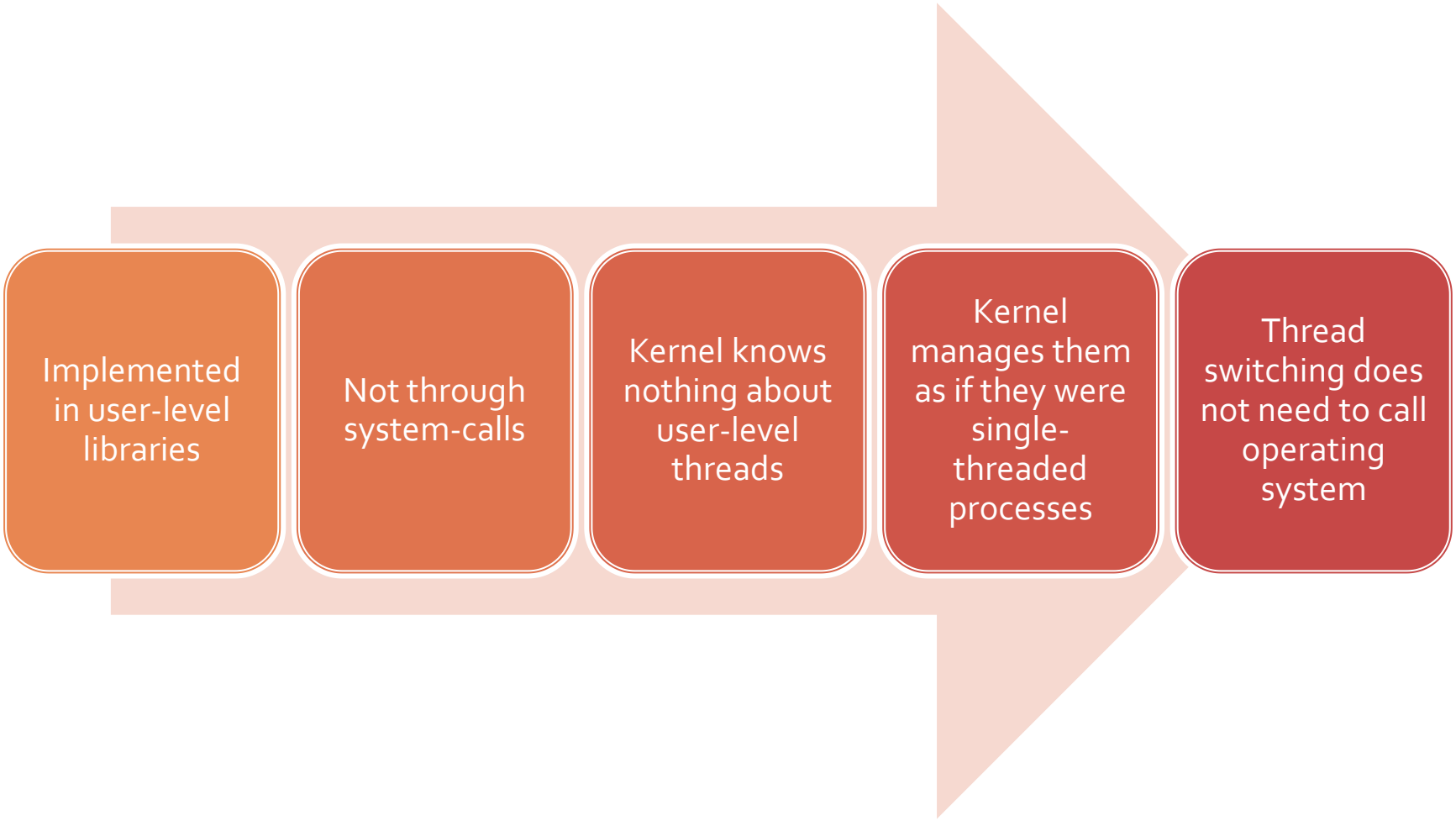
Because threads can share common data, they do not need to use inter-process communication.

Threads can take advantage of multiprocessors.

Are cheap because

- They only need a stack and storage for registers therefore, cheap to create.
- Threads use very little resources of an operating system in which they are working
- Context switching are fast when working with threads. The reason is that we only have to save and/or restore Program Counter, Stack space and registers.

Why Threads?



User-Level Threads

Can be implemented on an Operating System that does not support threads.

Some other advantages are

- User-level threads does not require modification to operating systems.
- Simple Representation - Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
- Simple Management - This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
- Fast and Efficient - Thread switching is not much more expensive than a procedure call.

User-Level Threads-Advantages

Lack of coordination between threads and operating system kernel.

Process as whole gets one time slice irrespective of whether process has one thread or 1000 threads within.

It is up to each thread to relinquish control to other threads.

User-level threads requires non-blocking systems call i.e., a multithreaded kernel.

User-Level Threads - Disadvantages

The kernel knows about and manages the threads.

No runtime system is needed in this case.

Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system.

In addition, the kernel also maintains the traditional process table to keep track of processes.

Operating Systems kernel provides system call to create and manage threads.

Kernel-level Threads

Advantages:

- Because kernel has full knowledge of all threads, Scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
- Kernel-level threads are especially good for applications that frequently block.

Disadvantages:

- The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
- Since kernel must manage and schedule threads as well as processes, it requires a full thread control block (TCB) for each thread to maintain information about threads.
- Results in significant overhead and increase in kernel complexity.

Kernel-level Threads

Context Switching

- Threads are very inexpensive to create and destroy, and they are inexpensive to represent.

Sharing

- Threads allow the sharing of a lot resources that cannot be shared in process, for example, sharing code section, data section, Operating System resources like open file etc.

Advantages of Thread over multiple processes

Blocking

- The major disadvantage is that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.

Security

- Since there is an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread overwrites the stack of another thread (or damaged shared data) although it is very unlikely since threads are meant to cooperate on a single task.

Disadvantages of Threads over Multiple Processes

To give each process on a multi-programmed machine a fair share of the CPU, a hardware clock generates interrupts periodically.

This allows the operating system to schedule all processes in main memory (using scheduling algorithm) to run on the CPU at equal intervals.

Each time a clock interrupt occurs, the interrupt handler checks how much time the current running process has used.

If it has used up its entire time slice, then the CPU scheduling algorithm (in kernel) picks a different process to run.

Each switch of the CPU from one process to another is called a context switch.

Context Switching

Major Steps

- The values of the CPU registers are saved in the process table of the process that was running just before the clock interrupt occurred.
- The registers are loaded from the process picked by the CPU scheduler to run next.

The threads share a lot of resources with other peer threads belonging to the same process.

So, a context switch among threads for the same process is easy.

It involves switch of register set, the program counter and the stack.

It is relatively easy for the kernel to accomplish this task.

Context Switching between threads

- Before a process can be switched its process control block (PCB) must be saved by the operating system. The PCB consists of the following information:
  - The process state.
  - The program counter, PC.
  - The values of the different registers.
  - The CPU scheduling information for the process.
  - Memory management information regarding the process.
  - Possible accounting information for this process.
  - I/O status information of the process.
- When the PCB of the currently executing process is saved the operating system loads the PCB of the next process that has to be run on CPU. This is a heavy task and it takes a lot of time.

Context Switching between processes