CSC 224 Principles of OS

4. Threads

Processes

- Processes have two characteristics:
- **1. Resource ownership.** Process includes a virtual address space to hold the process image
- the OS performs a protection function to prevent unwanted interference between processes with respect to resources

- 2. **Scheduling/Execution:** Follows an execution path that may be interleaved with other processes.
- A process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the OS

Threads

- The unit of dispatching is referred to as a ***thread or lightweight process***
- The unit of resource ownership is referred to as a ***process or task***
- ***Multithreading - The ability of an OS to*** support multiple, concurrent paths of execution within a single process

Threads

- A **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler which is typically a part of the operating system.
- A thread is a component of a process.
- Multiple threads can exist within the same process, executing concurrently (one starting before others finish) and share resources such as memory, while different processes do not share these resources.

Threads Vs Processes

- In many respects threads operate in the same way as that of processes.

Similarities

- Like processes threads share CPU and only one thread active (running) at a time.
- Threads within a processes execute sequentially.
- 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.
- All threads can access every address in the task .
- Unlike processes, thread are design to assist one other. Note that processes might or might not assist one another because processes may originate from different users.

Why threads

- 1. A process with multiple threads make a great server for example printer server.
- 2. Because threads can share common data, they do not need to use inter-process communication.
- 3. Because of the very nature, threads can take advantage of multiprocessors.

They are cheap in that

- 1. They only need a stack and storage for registers therefore, threads are cheap to create.
- 2. Threads use very little resources of an operating system in which they are working. That is, threads do not need new address space, data, program code or operating system resources.
- 3. Context switching are fast when working with threads.

User Level Threads (ULT)

- User-level threads implement in user-level libraries, rather than via **systems calls,** *(how a program requests a service from an operating system's kernel)* so thread switching does not need to call operating system and to cause interrupt to the kernel.

Advantages of ULT

- 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.

Disadvantages

- There is a lack of coordination between threads and operating system kernel. Therefore, 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. Otherwise, entire process will blocked in the kernel, even if there are runable threads left in the processes. For example, if one thread causes a page fault, the process blocks.

Kernel Level Threads KLT

- In this method, 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.

Adv of KLT

- 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 of KLT

- 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 require a full thread control block (TCB) for each thread to maintain information about threads.
- As a result there is significant overhead and increased in kernel complexity

Advantages of Threads over Multiple Processes

- Context Switching Threads are very inexpensive to create and destroy, and they are inexpensive to represent. they do not require space to share memory information, Information about open files of I/O devices in use, etc. With so little context, it is much faster to switch between threads. In other words, it is relatively easier for a context switch using threads.

- **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.

Disadvantages of Threads over Multiprocesses

- **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.

Application that Benefits from Threads

- A proxy server satisfying the requests for a number of computers on a LAN would be benefited by a multi-threaded process.
- In general, any program that has to do more than one task at a time could benefit from multitasking. For example, a program that reads input, process it, and outputs could have three threads, one for each task.

Application that cannot Benefit from Threads

- Any sequential process that cannot be divided into parallel task will not benefit from thread, as they would block until the previous one completes.
- For example, a program that displays the time of the day would not benefit from multiple threads.

Context switching in Threads

- 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 accomplished this task.

Context switching in processes

- Context switches among processes are expensive.
- 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.