Chapter 4 - Threads

1. **Chapter 4: Threads – Outline:** Thread programming is steadily growing in prominence for a variety of reason. However, thread programming is both more labor-intensive, and error-prone. We will see what threads are, how to do thread-based programming, its advantages, disadvantages, OS and library provided abstractions, etc.

2. **Process Overview:** We studied processes in the last chapter, what did we learn?

   - Each program in execution is run as a separate process.
   - fork() duplicates the entire process address space, and exec() overwrites it with a new program.
   - OSes provide security among processes, but giving them then separate, isolated address spaces. This protection and isolation also makes it difficult to share information between processes.
   - Context switching is required for time-sharing OSes.
   - We looked at various IPC mechanisms. IPC is expensive due to over-riding OS security mechanisms during sharing memory, or invoking multiple, repeated system calls during message passing.

3. **Is the Process Abstraction Always Suitable ?**

   - Monolithic processes cannot exploit multicore processors or distributed environments. The process abstraction is also expensive and involves a lot of overhead.
   - There are several examples that illustrate the need to run multiple sequences of code concurrently:
     (a) word processing software may have several threads of control displaying graphics, accepting keyboard input, background spelling and grammar checking, etc.
     (b) Web browser displaying images or text, while another thread of control retrieves data from the network.
   - Creating processes and context switching between them involves calls in the OS kernel, so is expensive.

4. **Is the Process Abstraction Always Suitable ? (2):** So, what functionality do we really need for such applications (discussed on the last slide)?

   Single process not enough for single application on multicore machines. Data sharing in processes is very expensive, due to all the protection that is involved. Creating a lot of processes may be a drain on system resources. For several applications, full duplication may not be necessary, code and data regions can be shared to reduce overhead. Thus, using threads may also reduce memory pressure (less swapping).

5. **Threads to the Rescue:** Threads greatly minimize the overhead over processes.

   A process can be single-threaded or multi-threaded. Threads within a single process may not be very concerned about isolation from each other.

   Threads have become so ubiquitous that almost all modern computing systems use thread as the basic unit of computation.

6. **Thread Basics:**

- Thread is an independent flow on control within the same process. It is lightweight because most of the overhead has already been accomplished through the creation of a process.
- Threads can only exist as long as the process is alive, we can have zombie processes but no zombie threads.
- Because threads within the same process share resources: Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads. Two pointers having the same value point to the same data. Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer.

7. **Single and Multithreaded Processes:** We can see that the code, data, and file descriptors are shared. Shared memory is the primary mode of data sharing between threads. Each thread has its own context, so that it may be independently scheduled by the OS.

8. **Thread Benefits:** Some benefits include:

   - A multithreaded web browser could allow user interaction in one thread while an image was being loaded in another thread.
   - Threads exist in the same address space. Resource sharing is easier since no explicit programmer setup of IPC meachinsm is necessary.
   - Sharing resources reduce creation and context-switching overhead. Solaris creation of threads is 30 times cheaper than process creation.
   - A single-threaded process can only exploit one CPU. Multi-core are the direction of the future.
   - Threads are not very easy to program though. There are many things we should be aware of, and we will look at this in a later slide.

9. **Thread Programming in Linux:** Traditionally, hardware vendors implement their own propriety version of threads, making software portability a big issue for multithreaded programs. So, POSIX standard was developed.

   Mutex stands for mutual exclusion. Remember, threads share a lot of information.

   We will next look at an example that using functions from the first set of APIs. The remaining two (Mutexes and Condition Variables) will be studied in lab discussions.

10. **Pthreads Example:** The program calculates the summation of a non-negative integer in a separate thread.

    pthreads library is used. The library may be implemented as a separate library, or as part of libc.

    sum is a global, so it is in the data section.

    Pthreads thread starts execution in a new function. We can only pass it one argument. How to pass multiple arguments?

    i and *upper* defined in the function runner are thread-specific, assigned storage on the stack. The parameter type `void *` can be cast to the appropriate type.

11. **Pthreads Example – API:**

- Attribute objects provide clean isolation of the configurable aspects of threads. Attributes are specified only at thread creation time; they cannot be altered while the thread is being used. An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type. Attributes can be used to configure the scope, detach-state, stackaddr, stacksize, scheduling priority, and scheduling policy.
- This qualifier can be applied to a data pointer to indicate that, during the scope of that pointer declaration, all data accessed through it will be accessed only through that pointer but not through any other pointer. The 'restrict' keyword thus enables the compiler to perform certain optimizations based on the premise that a given object cannot be changed through another pointer.
- If the function called as the starting point of a thread returns, then an implicit call to pthread_exit is assumed. When a process exits, an implicit call to exit is assumed.
- An attribute object is opaque, and cannot be directly modified by assignments. A set of functions is provided to initialize, configure, and destroy each object type.

12. **User Vs. Kernel Level Threads:**

    - **Use level threads:** Often this is called "cooperative multitasking" where the task defines a set of routines that get "switched to" by manipulating the stack pointer. Typically each thread "gives-up" the CPU by calling an explicit switch, sending a signal or doing an operation that involves the switcher. Scheduling can also be made more efficient.

      Kernel is not aware of the user-level threads.
    - **Kernel-Level threads:** kernel schedules each thread within the timeslice of each process. There is a little more overhead with mode switching from user-¿kernel-¿ user and loading of larger contexts, but initial performance measures indicate a negligible increase in time. Linux Pthreads uses the "clone" system call to create threads.
    - User-level threads are not ideal on multicore processors, since if one thread blocks on I/O, then the entire process blocks, since we only have one kernel thread. Having multiple kernel threads gets past this problem.

13. **Multithreading Models:**

14. **Many–to–One Model:** This model particularly shows the disadvantages of user level threads. I/O blocking causes problems, SMPs cannot be exploited.

15. **One–to–One Multithreading Model:** Creating a user thread requires creating a low efficient and bulky kernel thread. Restricts the number of threads that can be created due to this overhead.

16. **Many–to–Many Multithreading Model:** User library can create as many threads as needed.

17. **Two–level Multithreading Model:** Also allows a user-level thread to be bound to a kernel thread.

18. **Threading Issues:** Providing thread support will require answering some different questions than just using processes. We will discuss these issues on the next slides.

19. **Semantics of Fork() and Exec():** The semantics of fork and exec change in a multi-threaded program. One version of fork duplicates all threads, and the other only duplicates the calling thread.

    In Linux, fork only duplicates the calling thread. This may leave shared data, and locks managed by the other threads in an inconsistent state, and cause terrible issues later. Only use fork in a multi-threaded program, if you are going to immediately do an exec after the fork in the child process.
    Reference: *http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them*

    On Linux, `exec()`, the new program replaces the entire process, with all threads.

20. **Thread Cancellation:** Required if activity of the thread in no longer required: like several threads searching for same name in database, and one thread finds the record. Or, in browsers if user wants to stop the page loading, then all threads should stop.

    With deferred cancellation, the thread periodically checks if it should be canceled. Pthreads use deferred cancellation at points called cancellation points. Pthreads function to cancel threads is called pthread_cancel(). Thread can be set to ignore cancellation requests, or to change type to asynchronous or deferred (default) cancellation. On deferred cancellation, the cancel status is checked at certain calls, such as pthread_testcancel() or pthread_join, and cleanup handlers are then called to destroy thread-specific data.

21. **Signal Handling:**

    - How do you send a signal to a process? Use the `kill` or `raise` system calls.
    - Every signal has a default handler provided by the OS. The user process can override the default handler with its handler for most signals. The user-defined signal handler may even be used to completely ignore the signal. Some signals like SIGKILL cannot be overridden by the user process.
    - Synchronous signals: illegal memory access, divide by zero, etc.
      Asynchronous signals: timer interrupt, control-C, etc.
    - (a) Synchronous signals are typically delivered to the same process, asynchronous signals to another process.
      (b) Synchronous signals are generally delivered to a single thread that generates the signal.
      (c) Asynchronous signals vary: some like Control-C are delivered to all threads, for other signals, individual threads may be allowed to ignore them. Also, in many OSes, signals are only delivered once, so delivered to the first thread accepting the signal.

22. **Thread Pools:** Some applications like web-servers may require creation of a number of threads. The kernel does not allow creation of an unlimited number of threads since that could exhaust system resources, and cause more OS overhead in activities such as scheduling.

    Number of threads can also be dynamically adjusted as system state changes. Thread creation only happens at program startup and thread destruction only at program end.

23. **OpenMP** From Wikipedia:
    OpenMP is an implementation of multithreading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

24. **Linux Thread Implementation:** Arguments passed to clone() determine how much sharing needs to take place. Specifying all these parameters is essentially like creating a thread, if none are specified then it would be creating a process.

    Optional: For more details: `man -s7 pthreads` and `man nptl`

25. **Multicore Processors:** Modern processors often feature multiple *cores* on a single processor chip. We will discuss other details during the in-class Zoom sessions.

    What is *hyperthreading*?

26. **Single Core Vs. Multicore Execution:** Concurrent execution on (independent) tasks can result in significant performance gain on multi-core processors.

27. **Challenges for Multicore Programming:** There are many challenges programmers need to overcome when writing effective parallel programs for multicore processors.

    Dividing activities for balanced workload: Tasks need to carefully divided so all processors/core can be kept uniformly busy. A skewed distribution of concurrent tasks (for instance, one long task and several other short tasks) will waste hardware resources as some tasks finish, but the cores are idle while waiting for the longest task to finish.

    Data splitting: The data should be split such that the data needed by each task is located close to its processing.

    Any data dependency between tasks running concurrently will limit the gains from concurrent execution, as tasks routinely stall until the dependency is satisfied.

    Testing and debugging issues: A new category of errors, called race conditions, arise from parallel programming. We will study these further in the next chapter.