CmpE 322

January 29, 2022

Contents

1 Discussion IV

1) Is a register set saved per process or per thread? Why?

Answer: A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter (PC), a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

2) If you want to design a web server, would you prefer a multi threaded one or a single threaded one? Why? Explain briefly.

Answer: I'd prefer designing a multi threaded web server. A web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced. One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests.

3) Can two processes run in parallel but may not be concurrent? Why or why not? Can you give an example?

Answer: Concurrency: Making your system seem like parallel by making each process progres in a certain amount of time. However it's not parallel in essence. When two processes run in parallel, there is both parallelism and concurrency. A system is parallel if it can perform more than one task simultaneously. In contrast, a concurrent system supports more than one task by allowing all the tasks to make progress. Thus, it is possible to have concurrency without parallelism. Consider an application with four threads. On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time, because the processing core is capable of executing only one thread at a time. On a system with multiple cores, however, concurrency means that the threads can run in parallel, because the system can assign a separate thread to each core.

4) Let's assume that the completion time of a task running on a dual core CPU is T ms. If we replace the CPU with a quad core one, how will the completion time change (approximately)?

Assume that s is serial portion of the task. According to the Amdahl's Law, if we run this task on a dual core CPU and quad core CPU, maximum speedups will be $\frac{2}{(1+2s)}$, and $\frac{4}{(1+3s)}$ respectively. If the serial portion of the program is 0, then the maximum speedups are 2 times and 4 times for dual core CPU and quad core CPU respectively. For these maximum speedup values, the replacement in the question results in maximum 2 times speedup approximately so in this situation completion time will be $\frac{T}{2}$. But this is the maximum speed-up situation. Therefore, the completion time will be between $\frac{T}{2}$ and T.

5) Why do we need implicit threading?

Answer: With the continued growth of multicore processing, applications containing hundreds—or even thousands—of threads are looming on the horizon. Designing such applications is not a trivial undertaking: programmers must address not only the challenges like dividing activities, balance, data splitting, data dependency, testing and debugging but additional difficulties as well. One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed implicit threading. As we shall see, these strategies generally require application developers to identify tasks—not threads—that can run in parallel. A task is usually written as a function, which the run-time library then maps to a separate thread, typically using the many-to-many model. The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management.

6) Thread-safe Data Structures

(a) What is a Thread safe Data Structure?

Answer: Thread safety is a computer programming concept applicable to multi-threaded code. Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. There are various strategies for making thread-safe data structures. A program may execute code in several threads simultaneously in a shared address space where each of those threads has access to virtually all of the memory of every other thread.

Thread safety is a property that allows code to run in multithreaded environments by re-establishing some of the correspondences between the actual flow of control and the text of the program, by means of synchronization. Handling some data structures by multiple threads may be a problem. For example, for a linked list, while one of the threads is inserting an element between two elements, say 2 and 3, the other thread removes the element which is one of the next elements of the inserted elements, say 3. In this case, the 4th element should have a backlink to the newly inserted element. But this threat is not aware of that so it will cause a problem. Such data structures are said to be thread unsafe, which means it's not safe to use these structures with the threads. To make it thread safe, you should modify that data structure that will prevent such errors and intel tbb for example provides such versions of hashmap, queue, vector, and some other thread safe data structures.

(b) Please give an example.

Answer: Since they are not modifiable by the definition, immutable data types (such as string, int, etc. in Python), or structures created with **final** keyword are thread-safe data structures. Using lock and synchronized functions from the associated thread library is another way to make a data structure thread-safe, therefore it avoids deadlocks and collisions on data.

7) Ending a Thread

(a) In what context, would a process have "dead threads", i.e. threads that should have been created are not there?

Answer: In POSIX (therefore in Linux), only the thread that calls fork() is duplicated; all other threads are "dead". Normally, in single-thread processes, fork() duplicate the whole process with just one difference, which is the return value of fork and the child process continues from the same statement, does not start from the beginning. for example, let's say the parent process creates 3 threads, so there are 4 many threads with the main thread. When you create a duplicate of the thread that calls fork(), typically it is the main thread, all other threads are not created. The child process will continue from the next statement. So the child is in a strange position. the child is now executing a statement before which three other threads should have been created and if they failed actually you would not have been here but to the child process it appears like those other threats mysteriously have died so if the child process depends on the proper execution of those threads, there will be problems such as deadlock.

(b) When the main thread ends (exits), what happens typically to the threads created by that?

Answer: When main thread ends, typically all other threads will be not detached from the main thread. So, end of the maind thread will mean the end of also other threads, even if they haven't properly complete their execution.

(c) Is there a way for changing that typical behaviour?

Answer: If we detach the other threads from main thread, AND end the main thread not by a "return" statement, but by the "pthread_exit()" statement, then the main thread will end just like any other thread. Since we didn't return in main, end of main won't mean end of the program. It will only mean the end of that thread. All other detached threads will keep working. We can detach a thread from the main thread by calling the related function from pthread library or setting the corresponding attributes while creating the thread.

8) Contention Scope

(a) What are the advantages and disadvantages of Process Contention Scope and System Contention Scope?

Answer: In Process Contention Scope, the thread library schedules user-level threads to run on an available LWP. In System Contention Scope, which kernel-level thread is to be scheduled into the CPU is determined. Process Contention Scope allows the programmer to set the priorities of the threads manually. If all threads are PCS, then context switching, synchronization, scheduling everything takes place within the userspace. This reduces system calls and achieves better performance. The number of LWPs created highly depends on the number of SCS threads created. This increases the kernel complexity of handling scheduling and synchronization.

(b) Do we always need both of them?

Answer: No, we don't. Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS.

(c) What is the difference between a regular process and a Light Weight Process?

Answer: A regular process is essentially a program that is loaded into the memory and being executed. A Light Weight Process is an intermediate step for user-level threads to be mapped to an associated kernel-level thread. The main difference between a light weight process (LWP) and a normal process is that LWPs share same address space and system resources with other processes. Whereas for regular processes, OS provides an independent address space. From the user's point of view, LWP is equivalent to a kernel thread.

9) What are the advantages and disadvantages of Common Ready Queue vs Private Ready Queue for multiprocessor scheduling?

Answer: In Common Ready Queue, all of the processors are put to the same ready queue. In this approach each processor is allocated to the next process in the common ready queue when they become available. One of the issue that one may encounter in this approach is that some processors may try to take the same process from queue, and in order to avoid that proper synchronization between the processors must be designed. Synchronization may cause some processors stay idle for a while and it introduces an overhead, which makes this approach inefficient. Common ready queue harms processor affinity.

In Private Ready Queue, each processors has its own ready queue specific to that processor. This alleviates the burden of synchronization between the processors. This approach is more commonly implemented compared to the Common Ready Queue approach. However, in this approach one processor may stay idle while other processors have a multitude of processes waiting in their private queue. To avoid this issue, a proper load balancing algorithm must be utilized.

10) HW Threads

(a) What are hardware threads?

Answer: Hardware threads are physical CPU's or cores. In today's computers, CPU's have multiple hardware threads per core, which is called hyperthreading. From an operating-system perspective, each hardware thread appears as a logical processor that is available to run a software thread.

(b) Why do we use hardware threads?

Answer: Hardware threads are used to run the processes in **parallel** manner rather than executing them concurrently, thus it increases the performance of the CPU. Usage of hardware threads increases the number of processes executed in a given amount of time, therefore it improves the throughput. Hardware threads can be used as a solution to memory stalls during the execution of a program.

11) Heterogeneous Multi-Processing

(a) What is the difference between Heterogeneous Multi-Processing (HMP) and Asymmetric Multi-Processing?

Answer: In Asymmetric Multi-Processing all scheduling decisions, I/O processing, and other system activities are handled by a single processor - the master server. Other processors execute only the user code.

In Heterogeneous Multi-Processing, processors do not have the same technical properties such as clock speeds and power management schemes. In Asymmetric Multi-Processing, all processors had same properties. In Heterogeneous Multi-Processing there is no definite hierarchy between the processors like in the Asymmetric Multi-Processing.

(b) Why do we need HMP?

Answer: HMP is useful to tackle with a trade-off problem. We can use power-hungry big cores to get better clock speed, however then we'll have a poor power management scheme. On the contrary, we can use little cores that don't demands so much power, but then we can't have high clock speeds. To solve this problem we can use a mixture of power-hungry big cores and power-efficient little cores. Big cores can be used for tasks that require short periods, whereas little cores can be used to run background jobs. Such scheduling of the processes between different cores is called Heterogeneous Multi-Processing.