CHP#04: Multithreading

4.1 Provide three programming examples in which multithreading provides better performance than a single-threaded solution.

1. Web server: A web server that uses a single thread to handle incoming requests can only serve one request at a time. If the server receives multiple requests simultaneously, it must process them one by one, which can result in slow response times and poor performance. By using multiple threads, the server can handle multiple requests concurrently, improving its performance and responsiveness.

2. Parallel processing: Many computational tasks can be divided into smaller subtasks that can be executed concurrently. For example, when processing a large dataset, the data can be split into smaller chunks and processed in parallel by multiple threads. This can significantly reduce the overall processing time compared to a single-threaded solution that processes the data sequentially.

3. In a single-threaded application with a graphical user interface (GUI), the user interface can become unresponsive if the application is performing a long-running task such as downloading a large file or performing a complex calculation. By using multiple threads, the application can perform the long-running task in the background while keeping the user interface responsive to user input.

4.2 Using Amdahl's Law, calculate the speedup gain of an application that has a 60 percent parallel component for (a) two processing cores and (b) four processing cores.

4.14 Using Amdahl's Law, calculate the speedup gain for the following applications:

- **40 percent parallel with (a) eight processing cores and (b) sixteen processing cores**
- **67 percent parallel with (a) two processing cores and (b) four processing cores**
- **90 percent parallel with (a) four processing cores and (b) eight processing cores**

4.4 What are two differences between user-level threads and kernel-level threads? Under what circumstances is one type better than the other?

1. **Management**: User-level threads are managed by a user-level library, while kernel-level threads are managed by the operating system kernel. This means that the creation, scheduling, and synchronization of user-level threads are handled by the user-level library, while these tasks are handled by the kernel for kernel-level threads.
2. **Context switching**: Context switching between user-level threads is generally faster than between kernel-level threads because it does not require a switch to kernel mode. However, if a user-level thread performs a blocking system call, all other threads within the same process may be blocked as well, while kernel-level threads can be scheduled independently.

when considering the level of control over thread scheduling. User-level threads provide more control over scheduling, while kernel-level threads may provide better overall system performance and fairness.

4.5 Describe the actions taken by a kernel to context-switch between kernel level threads.

1. **Save the context of the current thread**: The kernel saves the current state of the thread that is being switched out, including its register values and program counter.
2. **Update the thread control block**: The kernel updates the thread control block (TCB) of the current thread to reflect its new state (e.g., ready or blocked).
3. **Choose the next thread to run**: The kernel uses its scheduling algorithm to choose the next thread to run from the set of ready threads.
4. **Load the context of the new thread**: The kernel loads the saved state of the new thread that is being switched in, including its register values and program counter.
5. **Update the CPU**: The kernel updates the CPU with the new thread's context and sets it to begin executing.

4.6 What resources are used when a thread is created? How do they differ from those used when a process is created?

When a thread is created, several resources are allocated by the operating system:

1. **Thread control block**: A data structure that stores information about the thread, such as its state, priority, and scheduling information.
2. **Stack**: Each thread has its own stack, which is used to store local variables and function call information.
3. **Registers**: Each thread has its own set of registers, including the program counter and stack pointer.

When a process is created, additional resources are allocated by the operating system:

1. **Process control block**: A data structure that stores information about the process, such as its state, priority, and resource usage.
2. **Memory**: The operating system allocates memory for the process's code and data segments.
3. **File descriptors**: The operating system allocates file descriptors for the process's standard input, output, and error streams.

4.8 Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

1. **Computationally-bound tasks**: If a task is computationally-bound and does not involve any I/O operations or blocking system calls, then using multiple threads may not provide any performance benefits. This is because the CPU is already fully utilized by the single-threaded solution, and adding more threads would not increase the amount of work that can be performed concurrently.
2. **High contention for shared resources**: If multiple threads need to access a shared resource frequently and there is high contention for that resource, then using multiple threads may not provide any performance benefits. This is because the threads would spend a significant amount of time waiting to acquire locks on the shared resource, reducing the amount of useful work they can perform.

4.10 Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values

b. Heap memory

c. Global variables

d. Stack memory

In a multithreaded process, **heap memory** and **global variables** are shared across threads. Register values and stack memory are private to each thread.

4.12 In Chapter 3, we discussed Google's Chrome browser and its practice of opening each new tab in a separate process. Would the same benefits have been achieved if, instead, Chrome had been designed to open each new tab in a separate thread? Explain.

No, the same benefits would not have been achieved if Chrome had been designed to open each new tab in a separate thread instead of a separate process.

The main benefit of opening each new tab in a separate process is that it provides better isolation between tabs. If one tab crashes or encounters an issue, it does not affect the other tabs because they are running in separate processes. Processes have their own memory space and resources, so they are better isolated from each other.

On the other hand, threads within a process share memory and other resources. If Chrome had been designed to open each new tab in a separate thread within the same process, then if one thread encounters an issue, it could potentially affect the other threads within the same process. This would defeat the purpose of isolating each tab for better stability and security.

4.13 Is it possible to have concurrency but not parallelism? Explain

Yes, it is possible to have concurrency without parallelism. Concurrency refers to the ability of a system to perform multiple tasks simultaneously, while parallelism refers to the simultaneous execution of multiple tasks using multiple processors or cores.

Concurrency can be achieved even on a single processor system through the use of techniques such as time-slicing, where the processor switches rapidly between tasks, giving the illusion that they are being executed simultaneously. In this case, there is concurrency because multiple tasks are being performed simultaneously, but there is no parallelism because only one task is being executed at a time by the single processor.

4.15 Determine if the following problems exhibit task or data parallelism:

- **Using a separate thread to generate a thumbnail for each photo in a collection**
- **Transposing a matrix in parallel**
- **A networked application where one thread reads from the network and another writes to the network**
- **The fork-join array summation application described in Section 4.5.2**
- **The Grand Central Dispatch system**

1. **Using a separate thread to generate a thumbnail for each photo in a collection**: This problem exhibits **task parallelism** because each thread is performing the same task (generating a thumbnail) on a different piece of data (a photo in the collection).
2. **Transposing a matrix in parallel**: This problem exhibits **data parallelism** because the same operation (transposing) is being performed on different parts of the data (the matrix) in parallel.
3. **A networked application where one thread reads from the network and another writes to the network**: This problem exhibits **task parallelism** because each thread is performing a different task (reading or writing) on the same data (the network).
4. **The fork-join array summation application described in Section 4.5.2**: This problem exhibits **data parallelism** because the same operation (summation) is being performed on different parts of the data (the array) in parallel.
5. **The Grand Central Dispatch system**: Grand Central Dispatch is a technology that allows developers to write concurrent code using blocks and queues. It can be used to implement both task and data parallelism, depending on how it is used by the developer.

A system with two dual-core processors has four processors available for scheduling. A CPU-intensive application is running on this system. All input is performed at program start-up, when a single file must be EX-8 opened. Similarly, all output is performed just before the program terminates, when the program results must be written to a single file. Between start-up and termination, the program is entirely CPU-bound. Your task is to improve the performance of this application by multithreading it. The application runs on a system that uses the one-to-one threading model (each user thread maps to a kernel thread).

- **How many threads will you create to perform the input and output? Explain.**
- **How many threads will you create for the CPU-intensive portion of the application? Explain.**

For the input and output portions of the application, you would create **one thread** each. This is because input and output operations are performed on a single file at the start-up and just before termination respectively. Having more than one thread for these operations would not improve performance.

For the CPU-intensive portion of the application, you could create **four threads**, one for each processor available for scheduling. This would allow the application to fully utilize the processing power of the system and improve its performance.

Consider the following code segment:

```
Pid_t pid;

pid = fork();

if (pid == 0) {

/* child process */

fork();

thread create( . . .);

}

fork();
```

a. How many unique processes are created?

b. How many unique threads are created?

a. In total, **4 unique processes** are created. The first `fork()` creates a child process. Then, both the parent and child processes call `fork()` again, creating two more child processes.
b. **1 unique thread** is created by the `thread_create(...)` function call in the first child process. This is in addition to the main thread of each process, resulting in a total of 5 threads.

Consider a multicore system and a multithreaded program written using the many-to-many threading model. Let the number of user-level threads in the program be greater than the number of processing cores in the system. Discuss the performance implications of the following scenarios.

a. The number of kernel threads allocated to the program is less than the number of processing cores.

b. The number of kernel threads allocated to the program is equal to the number of processing cores.

c. The number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads.

a. If the number of kernel threads allocated to the program is less than the number of processing cores, then not all processing cores will be utilized. This means that the program will not be able to fully take advantage of the available processing power and its performance may be limited.

b. If the number of kernel threads allocated to the program is equal to the number of processing cores, then all processing cores can be utilized. This allows the program to fully take advantage of the available processing power and can result in improved performance.

c. If the number of kernel threads allocated to the program is greater than the number of processing cores but less than the number of user-level threads, then all processing cores can be utilized and some user-level threads will have to share kernel threads. This can result in improved performance compared to scenario a, but may not be as efficient as scenario b due to the overhead of context switching between user-level threads sharing a kernel thread.

4.2. List reasons why a mode switch between threads may be cheaper than a mode switch between processes.

- Threads within the same process share the same address space and resources, so there is no need to flush or reload the memory management unit (MMU) or TLB caches.

- The kernel does not need to switch the virtual memory context or update memory mappings.

- The kernel does not need to save and restore as much information when switching between threads as it does when switching between processes.

4.3. What are the two separate and potentially independent characteristics embodied in the concept of process?

T he two separate and potentially independent characteristics embodied in the concept of a process are resource ownership and scheduling/execution.

4.4. Give four general examples of the use of threads in a single-user multiprocessing system.

- A web browser using separate threads for rendering, networking, and user interface.

- A word processor using separate threads for spell checking, auto-saving, and user interface.

- A video game using separate threads for physics simulation, rendering, and user input.

- A media player using separate threads for decoding, playback, and user interface.

4.5. How is a thread different from a process?

A thread is a unit of execution within a process, while a process is an instance of a program in execution. A process can contain multiple threads that share the same address space and resources, but each thread has its own program counter, stack, and set of registers

4.6. What are the advantages of using multithreading instead of multiple processes?

- Improved performance due to reduced overhead for context switching and inter-process communication.

- Easier sharing of data and resources between threads within the same process.

- More efficient use of system resources such as memory.

4.7. List some advantages and disadvantages of using kernel-level threads.

Some advantages of using kernel-level threads are:

- The kernel can schedule threads independently on different processors/cores.

- Kernel-level threads can take advantage of kernel-level constructs such as semaphores and mutexes.

- Kernel-level threads can be preempted by the kernel scheduler.

Some disadvantages of using kernel-level threads are:

- Creating, destroying, and switching between kernel-level threads may have higher overhead than user-level threads.

- Kernel-level thread operations require system calls which may have higher overhead than user-level thread operations.

4.8. Explain the concept of threads in the case of the Clouds operating system.

Clouds is an object-based distributed operating system that uses threads to support concurrency within objects. In Clouds, each object has its own thread of control that can execute methods on the object concurrently with other threads executing methods on other objects. This allows for efficient parallel processing and improves the responsiveness of the system.