Module 03: Threads and Concurrency

1

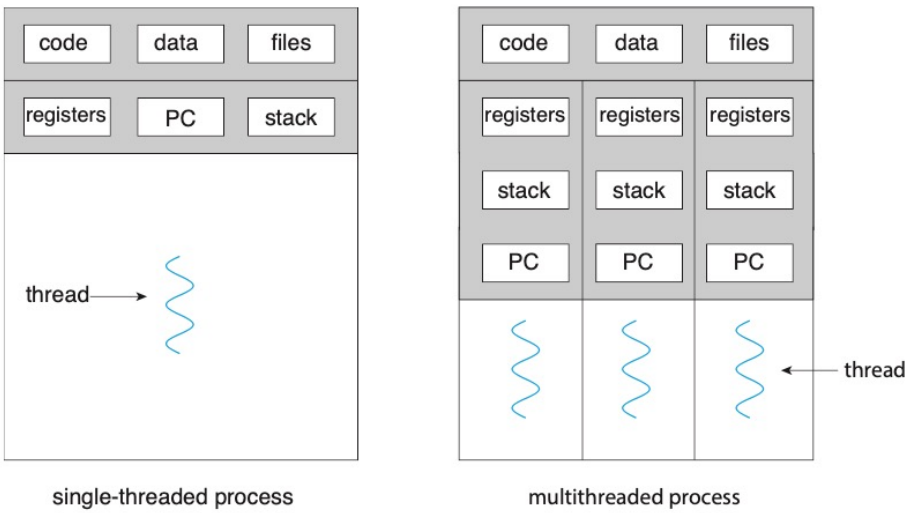


Figure 4.1 Single-threaded and multithreaded processes.

2

Benefits

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

3

Drawbacks – Yes, I know I used the same text

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

4

Drawbacks – Yes, I know I used the same text

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

5

Drawbacks – Yes, I know I used the same text

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had been completed. In contrast, if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user.
2. **Resource sharing**. Processes can share resources only through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes.
4. **Scalability.** The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available. We explore this issue further in the following section.

Because of these challenges, many software developers argue that the advent of multicore systems will require an entirely new approach to designing software systems in the future. (Similarly, many computer science educators believe that software development must be taught with increased emphasis on parallel programming.)

But is #2 the ONLY place I can say that?

6

Concurrency vs. Parallelism

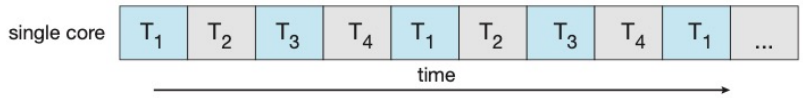


Figure 4.3 Concurrent execution on a single-core system.

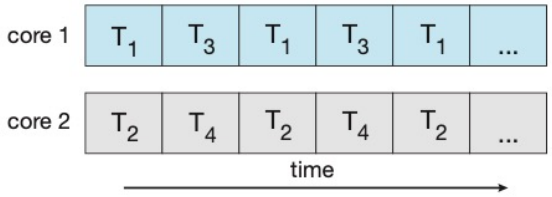
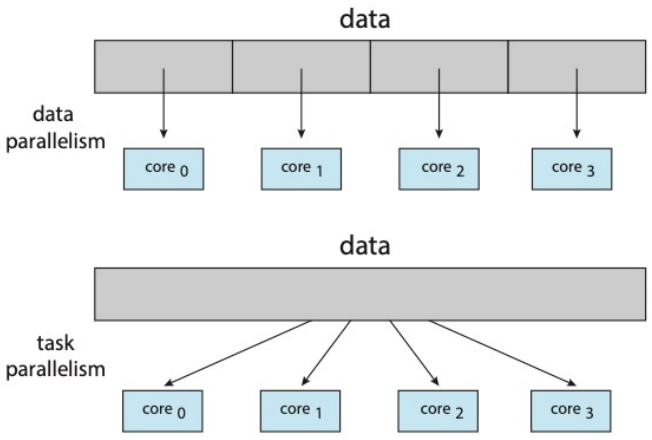


Figure 4.4 Parallel execution on a multicore system.

7

Types of Parallelism



8

Threading Models (User to Kernel Mapping)

- Many-to-One
- One-to-One
- Many-to-One
- Which one is implemented in Linux?

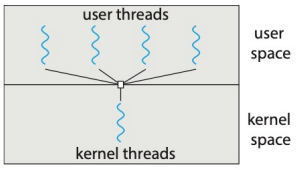


Figure 4.7 Many-to-one model.

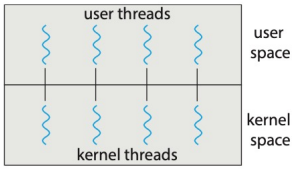


Figure 4.8 One-to-one model.

9

Types of Threads

- Synchronous (joinable)
  - The parent thread that created the child thread(s) waits for all of the joinable threads it made to finish and "rejoin" the main parent thread
  - This type of thread implements a VERY weak synchronization primitive. Trust me, we'll need better.
- Asynchronous (detached)
  - The parent thread that created the child threads will be schedulable along with all of its detached threads.
  - No one waits for anyone. All bets are off about who goes when.
- We will first, and primarily, talk about joinable threads
  - *These provide a "more natural" model for implementing data, task, or other standard forms of parallelism in that we can put all the "serial" processing in a "parent" function and have it launch parallel components all at once, as needed, "pausing" the serial processing until all a needed burst of parallel activity is finished….*

10

Example: Statistics Calculation

```
// THIS IS NOT ACTUAL CODE. IT WILL NOT AND IS NOT MEANT TO COMPILE.
// It is conceptual "pseudo-code" of a 100% serialized computation of
// some statistics realized in "pseudo" C.  NOTHING HERE IS ACTUAL
// PTHREADS SYNTAX

main()
{ read_in_a_data_set(&data_set);
  a = compute_stat_a(data_set);
  b = compute_stat_b(data_set);
  c = compute_stat_c(data_set);
  print_stats(a,b,c);
}
```

11

Example: Statistics Calculation

```
// THIS IS NOT ACTUAL CODE. IT WILL NOT AND IS NOT MEANT TO COMPILE.
// It is conceptual "pseudo-code" of a 100% serialized computation of
// some statistics realized in "pseudo" C.  NOTHING HERE IS ACTUAL
// PTHREADS SYNTAX

main()
{ read_in_a_data_set(&data_set);
  // begin parallel multi-threaded part
  thread_handle1 = THREAD_CREATE(a = compute_stat_a(data_set));
  thread_handle2 = THREAD_CREATE(b = compute_stat_b(data_set));
  thread_handle3 = THREAD_CREATE(c = compute_stat_c(data_set));
  THREAD_JOIN(thread_handle1);
  THREAD_JOIN(thread_handle2);
  THREAD_JOIN(thread_handle3);
  // end parallel multi-threaded part
  print_stats(a,b,c);
}
```

12

Threading Libraries

- Your OS will provide you with one or more threading libraries that provide API calls to create and manage threads
- User (many to one model and local function calls) vs Kernel (one-to-one or many-to-many model and system calls)
- Standards
  - POSIX Pthreads standard (https://en.wikipedia.org/wiki/POSIX_Threads)
  - Windows Threads API (though Windows can do pthreads too…)
- All threading libraries provide:
  - A way to turn "functions" into "threads" and allow them to be scheduled concurrently or in parallel to other threads
  - A way to send simple status information back to the "parent" thread when they finish.
  - A way to manage "joins" to provide simple synchronization as threads finish.

13

Let's Have Some Fun…

- What happens if ANY thread in a process runs exit() or _exit()?
- What happens if main() makes a bunch of threads and then runs a return()?
- Can I make as many threads as I want? SHOULD I make as many threads as I want?
- Just because separate threads CAN write to the same global static variables, should they?

14

Let's Have Some Fun…

- What happens if ANY thread in a process runs exit() or _exit()?
  The whole process ends and ALL the threads in the process are destroyed
- What happens if main() makes a bunch of threads and then runs a return()?
  The whole process ends and ALL the threads in the process are destroyed
- Can I make as many threads as I want? SHOULD I make as many threads as I want?
  The answer is… it depends. If you find yourself making many more threads than cores, you might experience degradation of performance because you pay more in context switching costs that you would have if you used a smaller task chunk size
- Just because separate threads CAN write to the same global static variables, should they?
  The answer is… if you're controlling shared access properly, go for it. If you're not, you will unleash hell upon yourself. We'll cover this in later lectures. For now just don't do it.

15