CSCI 340 Operating Systems

Chapter 4: Threads

Stewart Weiss

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Prologue: Processes Revisited

In the preceding chapter, you learned about **processes**. A process was defined as a program in execution, and it was assumed that a process had a single **thread of control**.

What does this mean?

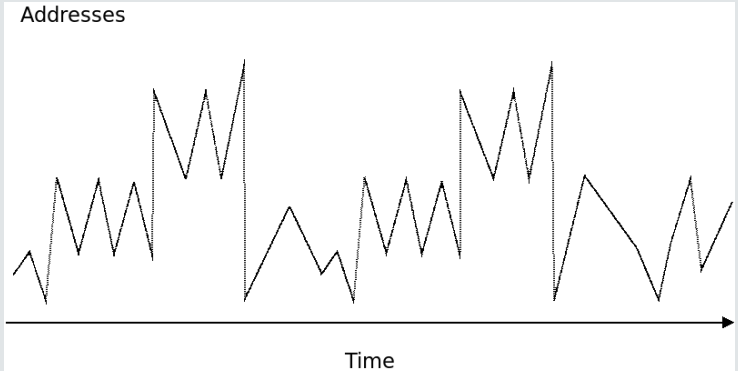
Prologue: Processes Revisited

In the preceding chapter, you learned about **processes**. A process was defined as a program in execution, and it was assumed that a process had a single **thread of control**.

What does this mean?

- A "**thread of control**" is a **sequence** of instructions that is executed **one instruction at a time**, **one after the other**, during the execution of a program[1].

Imagine that when a program executes, each time a machine code instruction is executed, the time of execution and address of the machine code instruction are written to a line of output. A single thread of control through a program would generate a graph such as the one to the right.



Each point is the address of the instruction executed at that time. The sequence is a "thread of control" through the program's address space.

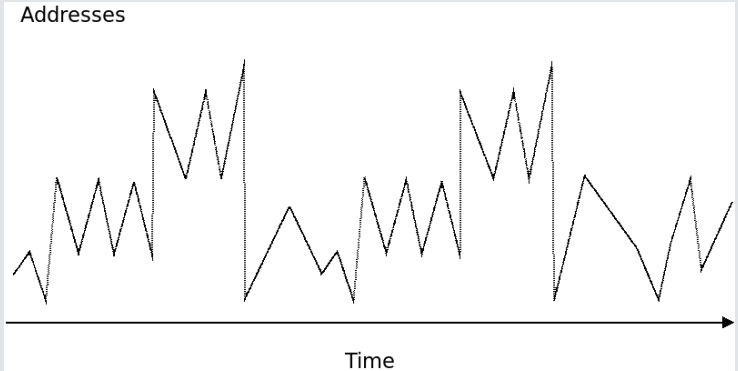
Prologue: Processes Revisited

In the preceding chapter, you learned about **processes**. A process was defined as a program in execution, and it was assumed that a process had a single **thread of control**.

What does this mean?

- A "**thread of control**" is a **sequence** of instructions that is executed **one instruction at a time**, **one after the other**, during the execution of a program[1].

Imagine that when a program executes, each time a machine code instruction is executed, the time of execution and address of the machine code instruction are written to a line of output. A single thread of control through a program would generate a graph such as the one to the right.



Each point is the address of the instruction executed at that time. The sequence is a "thread of control" through the program's address space.

Food for thought: Why does this graph have repeating patterns, and why does it make big jumps?

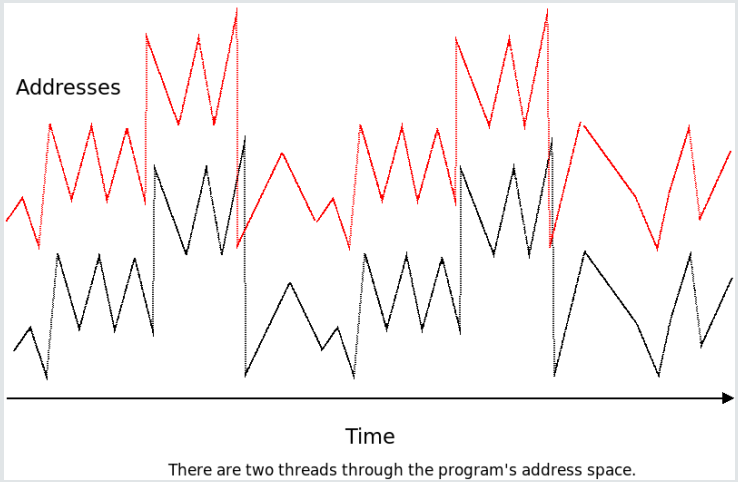
Prologue: Threads

Modern operating systems have made it possible for a process to have **multiple threads of control**. These multiple threads of control are called **threads**.

Prologue: Threads

Modern operating systems have made it possible for a process to have **multiple threads of control**. These multiple threads of control are called **threads**.

Multiple threads of a process can run **in parallel** and can therefore speed up a computation, taking advantage of the multiple processors that exist in almost all modern computers. To the right is a graph of two threads of a multithreaded program.

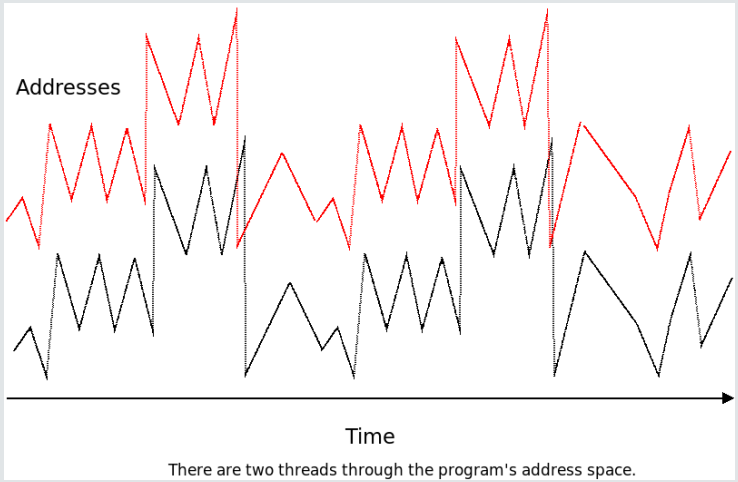


There are two threads through the program's address space.

Prologue: Threads

Modern operating systems have made it possible for a process to have **multiple threads of control**. These multiple threads of control are called **threads**.

Multiple threads of a process can run **in parallel** and can therefore speed up a computation, taking advantage of the multiple processors that exist in almost all modern computers. To the right is a graph of two threads of a multithreaded program.



There are two threads through the program's address space.

Some programs can have hundreds or even thousands of threads. **Modern operating system kernels are always multithreaded.**

About This Chapter

Because threads are so ubiquitous and also so hard to understand, they are an important topic in an operating systems course, and this chapter is entirely devoted to their study. In particular, it will cover the following topics:

- Multicore programming
- Explicit Threading
- Thread libraries
- Implicit threading
- Threading issues

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;
- describe various multithreading models, explain how they differ, and identify the strengths and weaknesses of each model;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;
- describe various multithreading models, explain how they differ, and identify the strengths and weaknesses of each model;
- explain what a thread library is and name three different thread libraries;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;
- describe various multithreading models, explain how they differ, and identify the strengths and weaknesses of each model;
- explain what a thread library is and name three different thread libraries;
- read, understand, and modify simple programs that use the Pthreads library;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;
- describe various multithreading models, explain how they differ, and identify the strengths and weaknesses of each model;
- explain what a thread library is and name three different thread libraries;
- read, understand, and modify simple programs that use the Pthreads library;
- explain what implicit threading is and describe thread pools, the fork-join model, and OpenMP;

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;
- describe various multithreading models, explain how they differ, and identify the strengths and weaknesses of each model;
- explain what a thread library is and name three different thread libraries;
- read, understand, and modify simple programs that use the Pthreads library;
- explain what implicit threading is and describe thread pools, the fork-join model, and OpenMP;
- describe how the Linux operating system represents threads; and

Chapter Objectives

You should be able to

- identify the basic components of a thread, and compare and contrast threads and processes;
- describe the advantages and disadvantages of designing multithreaded programs as compared to single-threaded programs;
- explain and apply Amdahl's Law;
- describe various multithreading models, explain how they differ, and identify the strengths and weaknesses of each model;
- explain what a thread library is and name three different thread libraries;
- read, understand, and modify simple programs that use the Pthreads library;
- explain what implicit threading is and describe thread pools, the fork-join model, and OpenMP;
- describe how the Linux operating system represents threads; and
- read and understand simple multithreaded programs that use the Linux threading API.

Motivation

Why are threads an important topic?

Motivation

Why are threads an important topic?

- Modern kernels are multithreaded. Understanding them requires understanding threads.

Motivation

Why are threads an important topic?

- Modern kernels are multithreaded. Understanding them requires understanding threads.
- Most modern applications are multithreaded. Examples:
  - A web browser process might have one thread to render images and text and another to download data.
  - A web browser might run each plug-in in a separate thread.
  - A chess program might have one thread "listening" for user input and rendering the chess board, and another analyzing future moves, while the user is thinking.

Motivation

Why are threads an important topic?

- Modern kernels are multithreaded. Understanding them requires understanding threads.
- Most modern applications are multithreaded. Examples:
  - A web browser process might have one thread to render images and text and another to download data.
  - A web browser might run each plug-in in a separate thread.
  - A chess program might have one thread "listening" for user input and rendering the chess board, and another analyzing future moves, while the user is thinking.
- Creating processes is much more time-consuming than creating threads.
- Processes use many more resources than threads overall.

Motivation

Why are threads an important topic?

- Modern kernels are multithreaded. Understanding them requires understanding threads.
- Most modern applications are multithreaded. Examples:
  - A web browser process might have one thread to render images and text and another to download data.
  - A web browser might run each plug-in in a separate thread.
  - A chess program might have one thread "listening" for user input and rendering the chess board, and another analyzing future moves, while the user is thinking.
- Creating processes is much more time-consuming than creating threads.
- Processes use many more resources than threads overall.
- Because threads share many of the same resources, cooperation among the threads that share resources **can** be easier.

Motivation

Why are threads an important topic?

- Modern kernels are multithreaded. Understanding them requires understanding threads.
- Most modern applications are multithreaded. Examples:
  - A web browser process might have one thread to render images and text and another to download data.
  - A web browser might run each plug-in in a separate thread.
  - A chess program might have one thread "listening" for user input and rendering the chess board, and another analyzing future moves, while the user is thinking.
- Creating processes is much more time-consuming than creating threads.
- Processes use many more resources than threads overall.
- Because threads share many of the same resources, cooperation among the threads that share resources **can** be easier.
- **But multithreading a program is difficult and dangerous and requires great understanding. Hence it is important to understand how to use threads.**

What is a Thread?

We have been discussing threads as an abstraction until now, so it is time to ask, what exactly is a thread?

What is a Thread?

We have been discussing threads as an abstraction until now, so it is time to ask, what exactly is a thread?

While there is no single definition, the one most commonly used is the one stated by POSIX:

> *A **thread** is the set of all resources necessary to represent a single thread of control through a program's executable code.*

This is the definition implicit in the Silberschatz, Gagne, Galvin textbook.

What is a Thread?

We have been discussing threads as an abstraction until now, so it is time to ask, what exactly is a thread?

While there is no single definition, the one most commonly used is the one stated by POSIX:

> *A **thread** is the set of all resources necessary to represent a single thread of control through a program's executable code.*

This is the definition implicit in the Silberschatz, Gagne, Galvin textbook.

From this definition, it follows that a thread has resources. In particular, because it executes instructions, it needs access to an address space containing executable code, usually the address space of a parent process.

It also follows that a thread is part of a process: processes contain threads and threads do not "live outside of processes."

What is a Thread?

We have been discussing threads as an abstraction until now, so it is time to ask, what exactly is a thread?

While there is no single definition, the one most commonly used is the one stated by POSIX:

> *A **thread** is the set of all resources necessary to represent a single thread of control through a program's executable code.*

This is the definition implicit in the Silberschatz, Gagne, Galvin textbook.

From this definition, it follows that a thread has resources. In particular, because it executes instructions, it needs access to an address space containing executable code, usually the address space of a parent process.

It also follows that a thread is part of a process: processes contain threads and threads do not "live outside of processes."

We will use this definition of threads for now, but will present an alternative definition later.

Thread Resources

When a process has multiple threads, the question is, **which resources are unique to each thread and which are shared by all threads?**

Thread Resources

When a process has multiple threads, the question is, **which resources are unique to each thread and which are shared by all threads?**

Each thread within that process **has its own**:

- unique thread ID
- program counter (PC)
- register set
- user stack

In some implementations, other resources such as signal masks are thread-specific.

Thread Resources

When a process has multiple threads, the question is, **which resources are unique to each thread and which are shared by all threads?**

Each thread within that process **has its own**:

- unique thread ID
- program counter (PC)
- register set
- user stack

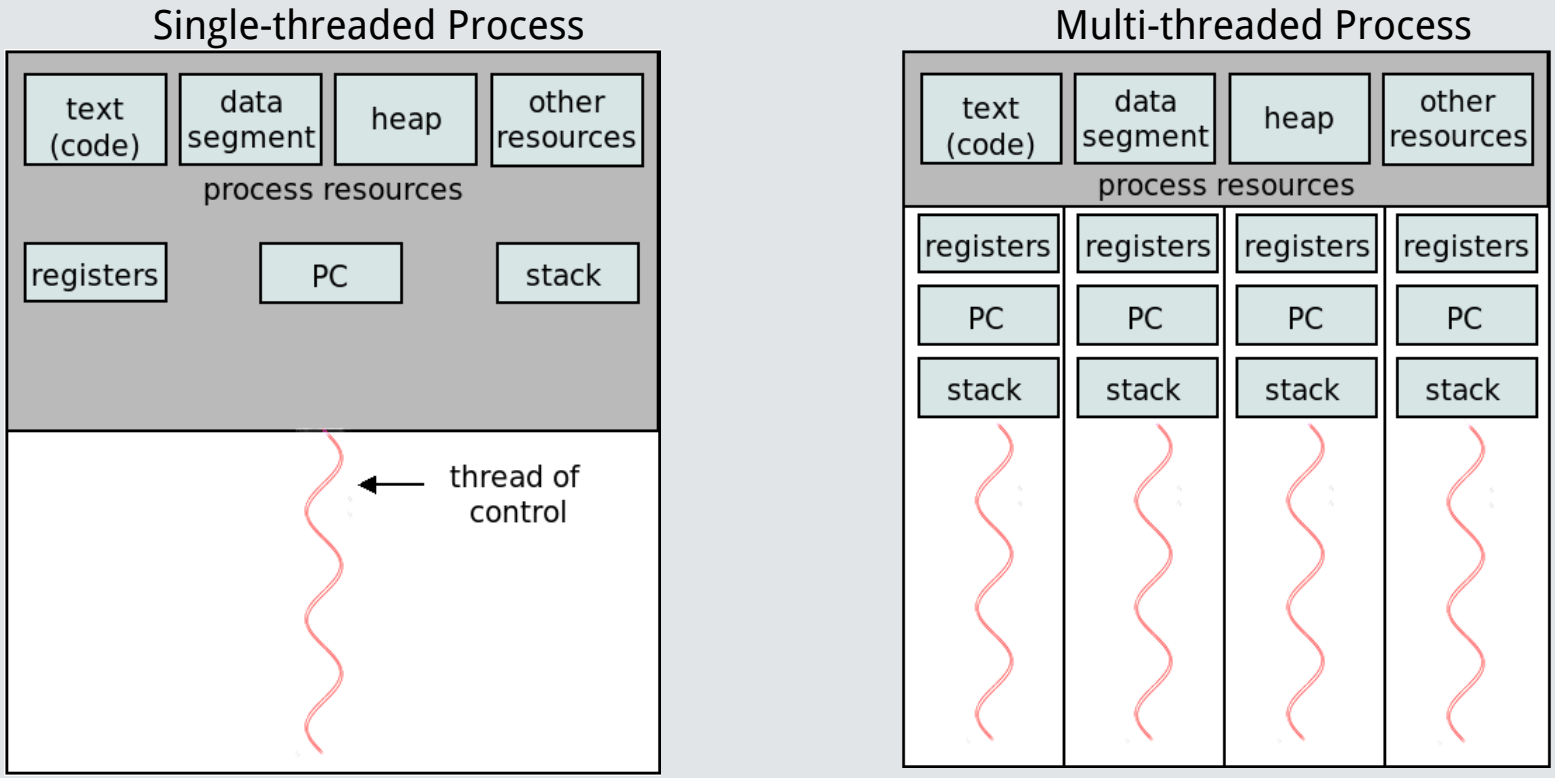
In some implementations, other resources such as signal masks are thread-specific.

All threads **share** many process resources, but the most important are:

- the executable code, i.e., the **text segment**,
- the **heap** memory, where dynamically allocated data is stored,
- the **data segments**, containing initialized and uninitialized data,
- the **command line arguments**, **environment variables**, and **open files**.

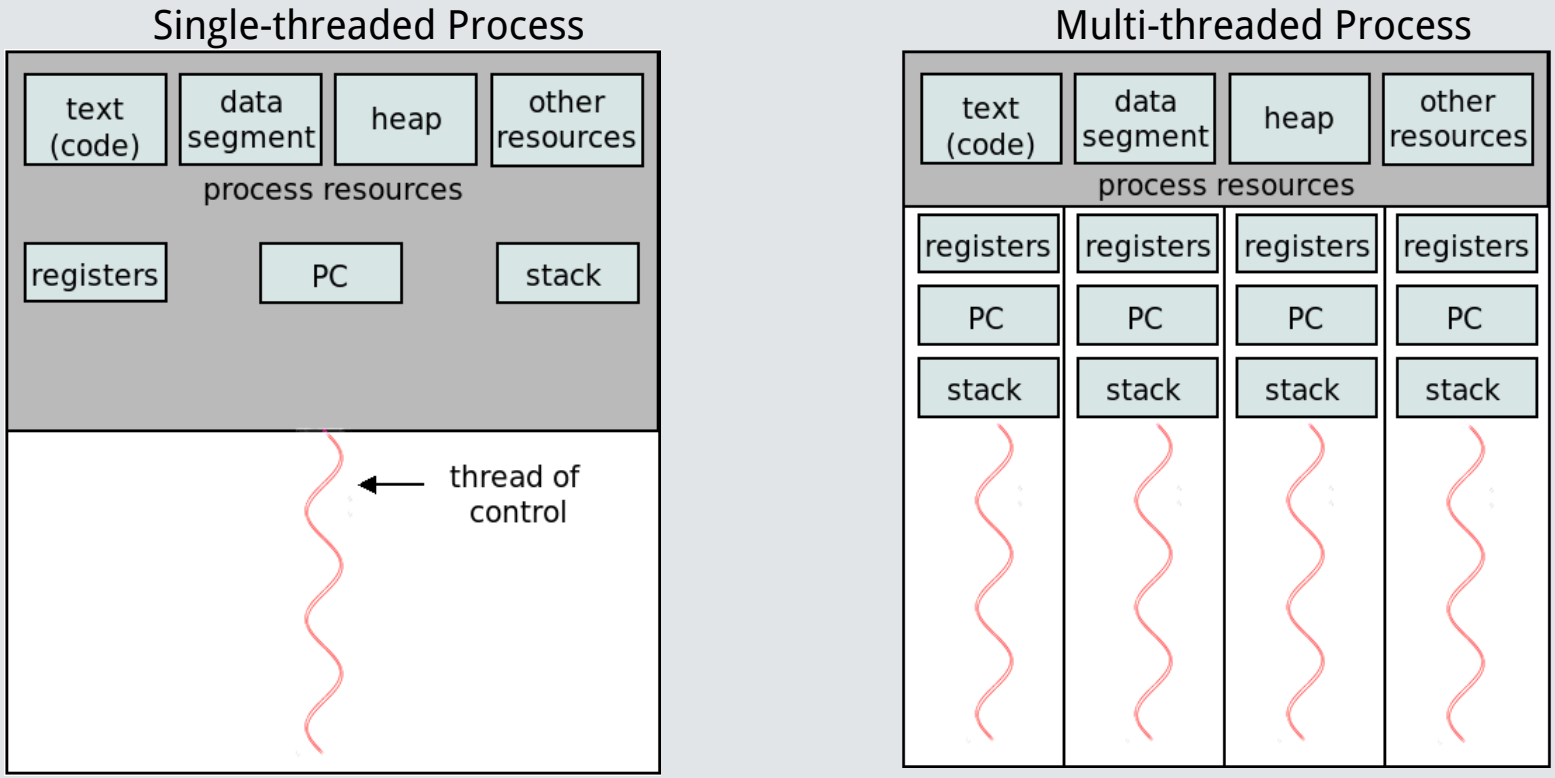
Single- and Multi-Threaded Processes

We can visualize single-threaded and multithreaded processes as shown below:



Single- and Multi-Threaded Processes

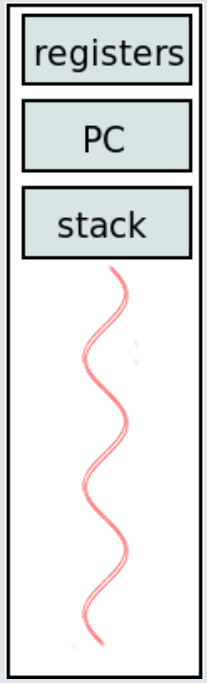
We can visualize single-threaded and multithreaded processes as shown below:



Notice that, in the multithreaded process to the right, each thread's stack, registers, and program counter are no longer process resources, but **thread-private** resources.

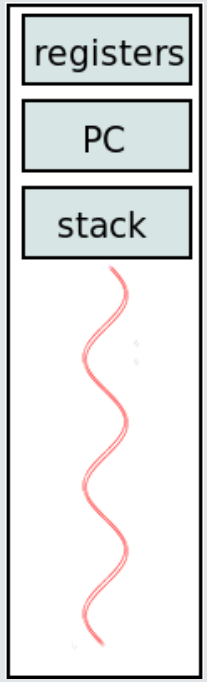
Visualizing Threads

In the preceding illustration of a multithreaded process, there were four threads, each of which was represented by the abstract figure to the right. In this figure, the curved line represents the thread of control through the set of instructions executed by the thread.



Visualizing Threads

In the preceding illustration of a multithreaded process, there were four threads, each of which was represented by the abstract figure to the right. In this figure, the curved line represents the thread of control through the set of instructions executed by the thread.



An individual thread is often depicted in a simpler way, eliminating the hardware resources, and just depicting it as just the squiggly flow of control:

We use this convention in the remainder of the notes.

A Thread Counting Activity

How many threads does a process have? How can you tell?

A Thread Counting Activity

How many threads does a process have? How can you tell?

In a few ways. In this activity you will count threads in one particular way.

Login to `eniac.cs.hunter.cuny.edu`. Then run the following command:

```
ps -efwL | more
```

You will see a list of all running processes on `eniac`, including their threads, one screenful at a time. (To advance to the next screen, press the `space` bar; to exit the command, type "`q`".) The last field on the line is the command that was executed to create the process (or thread).

The 6th field (with heading `NLWP`) is the number of threads that the given process has. A process with the value 1 in this column has one thread and is thus not multithreaded.

Which process has the most threads? What is the command that created it?

How many threads does `gnome-shell` have?

Reasons for Multi-threading

There are several good reasons to write programs that can contain multiple threads.

- Code to handle **asynchronous[1] events** can be executed by a separate thread. Each thread can handle its own event without using some type of asynchronous programming construct.

1 Asynchronous events are events that take place independently of the execution of the program, such as completion of an I/O operation, or the termination of a child process.

Reasons for Multi-threading

There are several good reasons to write programs that can contain multiple threads.

- Code to handle **asynchronous[1] events** can be executed by a separate thread. Each thread can handle its own event without using some type of asynchronous programming construct.
- Whereas multiple processes have to use special mechanisms provided by the kernel to share memory and file descriptors, threads automatically have access to the same memory address space, which is faster and simpler.

1 Asynchronous events are events that take place independently of the execution of the program, such as completion of an I/O operation, or the termination of a child process.

Reasons for Multi-threading

There are several good reasons to write programs that can contain multiple threads.

- Code to handle **asynchronous[1] events** can be executed by a separate thread. Each thread can handle its own event without using some type of asynchronous programming construct.
- Whereas multiple processes have to use special mechanisms provided by the kernel to share memory and file descriptors, threads automatically have access to the same memory address space, which is faster and simpler.
- On a multiprocessor architecture, many threads may run in parallel on separate cores, speeding up the computation. More cores can make the program faster.

1 Asynchronous events are events that take place independently of the execution of the program, such as completion of an I/O operation, or the termination of a child process.

Reasons for Multi-threading

There are several good reasons to write programs that can contain multiple threads.

- Code to handle **asynchronous[1] events** can be executed by a separate thread. Each thread can handle its own event without using some type of asynchronous programming construct.
- Whereas multiple processes have to use special mechanisms provided by the kernel to share memory and file descriptors, threads automatically have access to the same memory address space, which is faster and simpler.
- On a multiprocessor architecture, many threads may run in parallel on separate cores, speeding up the computation. More cores can make the program faster.
- Even on a single processor machine, performance can be improved by putting calls to system functions with expected long waits into separate threads. This way, just the calling thread waits, not the whole process.

1 Asynchronous events are events that take place independently of the execution of the program, such as completion of an I/O operation, or the termination of a child process.

Reasons for Multi-threading

There are several good reasons to write programs that can contain multiple threads.

- Code to handle **asynchronous[1] events** can be executed by a separate thread. Each thread can handle its own event without using some type of asynchronous programming construct.
- Whereas multiple processes have to use special mechanisms provided by the kernel to share memory and file descriptors, threads automatically have access to the same memory address space, which is faster and simpler.
- On a multiprocessor architecture, many threads may run in parallel on separate cores, speeding up the computation. More cores can make the program faster.
- Even on a single processor machine, performance can be improved by putting calls to system functions with expected long waits into separate threads. This way, just the calling thread waits, not the whole process.
- The response time of interactive programs can be improved by splitting off threads to handle user I/O. By handling I/O in separate threads, the application can respond quickly, even when time-consuming operations are taking place in other threads.

1 Asynchronous events are events that take place independently of the execution of the program, such as completion of an I/O operation, or the termination of a child process.

Multicore Programming

We explore how programs can be written that take advantage of the multiple cores and/or processors in a multiprocessor machine.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

In Chapter 3, Concurrency, we stated that two processes are **concurrent** if their computations can **overlap in time**. The same is true of threads.

In contrast, two processes or threads execute **in parallel** if they execute **at the same time on different processors**.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

In Chapter 3, Concurrency, we stated that two processes are **concurrent** if their computations can **overlap in time**. The same is true of threads.

In contrast, two processes or threads execute **in parallel** if they execute **at the same time on different processors**.

A **parallel program** is one which contains instruction sequences that can be executed in parallel.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

In Chapter 3, Concurrency, we stated that two processes are **concurrent** if their computations can **overlap in time**. The same is true of threads.

In contrast, two processes or threads execute **in parallel** if they execute **at the same time on different processors**.

A **parallel program** is one which contains instruction sequences that can be executed in parallel.

If it is executed on a single processor, the threads within it can be interleaved in time on the processor.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

In Chapter 3, Concurrency, we stated that two processes are **concurrent** if their computations can **overlap in time**. The same is true of threads.

In contrast, two processes or threads execute **in parallel** if they execute **at the same time on different processors**.

A **parallel program** is one which contains instruction sequences that can be executed in parallel.

If it is executed on a single processor, the threads within it can be interleaved in time on the processor.

A parallel program therefore contains concurrency and can be called a **concurrent program**.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

In Chapter 3, Concurrency, we stated that two processes are **concurrent** if their computations can **overlap in time**. The same is true of threads.

In contrast, two processes or threads execute **in parallel** if they execute **at the same time on different processors**.

A **parallel program** is one which contains instruction sequences that can be executed in parallel.

If it is executed on a single processor, the threads within it can be interleaved in time on the processor.

A parallel program therefore contains concurrency and can be called a **concurrent program**.

The opposite is not true: a system can have concurrency even though it is not designed as a parallel program and its threads are not running on separate processors.

Concurrency Versus Parallelism

The first step is to clarify the difference between **concurrency** and **parallelism**.

In Chapter 3, Concurrency, we stated that two processes are **concurrent** if their computations can **overlap in time**. The same is true of threads.

In contrast, two processes or threads execute **in parallel** if they execute **at the same time on different processors**.

A **parallel program** is one which contains instruction sequences that can be executed in parallel.

If it is executed on a single processor, the threads within it can be interleaved in time on the processor.

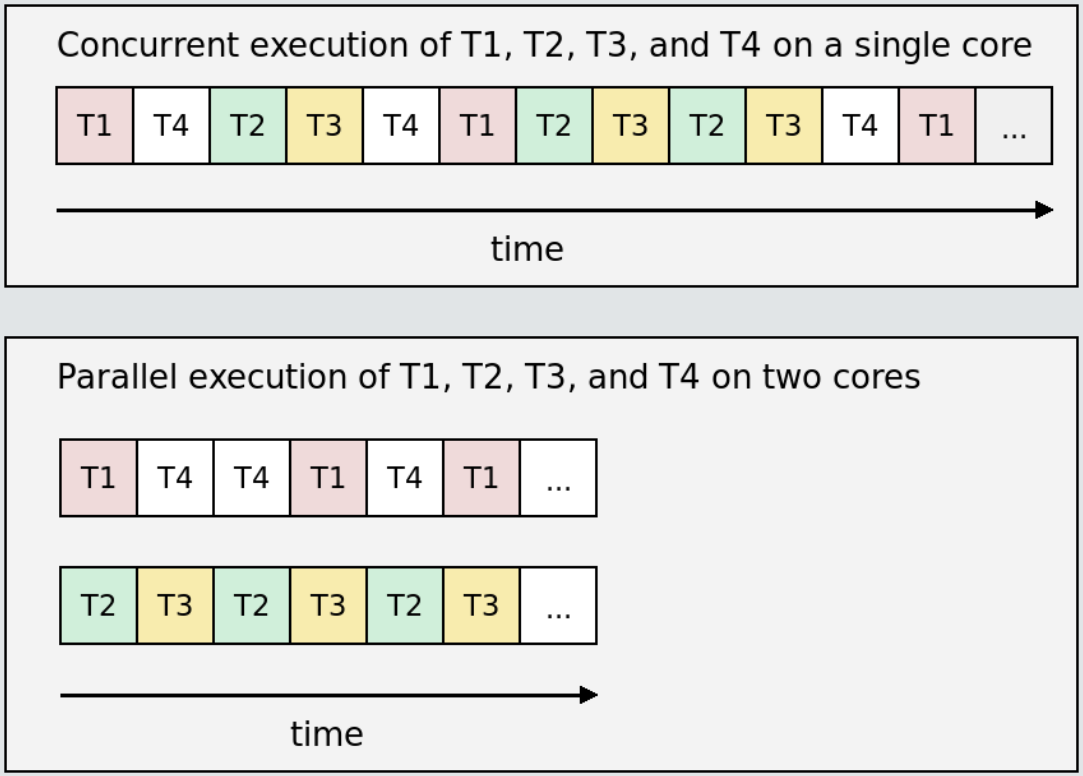
A parallel program therefore contains concurrency and can be called a **concurrent program**.

The opposite is not true: a system can have concurrency even though it is not designed as a parallel program and its threads are not running on separate processors.

The next slide visualizes the distinction.

Visualizing Concurrency and Parallelism

Imagine four threads, T1, T2, T3, and T4, that can be executed in parallel. If they are executed on a single core, as the top figure illustrates, the system is called a concurrent system and the threads are executed **concurrently**. If they are executed on two cores, as shown in the bottom figure, with T1 and T4 on one core and T2 and T3 on the other, it is **parallel execution**.



Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Not so. Although the hardware manufacturers have been making processors cheaper each year, packing more of them into computers, writing software for multiprocessors is a **more difficult task** than writing sequential code. The major problems are:

Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Not so. Although the hardware manufacturers have been making processors cheaper each year, packing more of them into computers, writing software for multiprocessors is a **more difficult task** than writing sequential code. The major problems are:

- **Functional Decomposition**. There is no algorithm to decompose a single task into multiple, independent, parallel subtasks.

Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Not so. Although the hardware manufacturers have been making processors cheaper each year, packing more of them into computers, writing software for multiprocessors is a **more difficult task** than writing sequential code. The major problems are:

- **Functional Decomposition**. There is no algorithm to decompose a single task into multiple, independent, parallel subtasks.
- **Load Balancing**. Decomposing a problem into subtasks in such a way that each has roughly the same amount of computation, to maximize each core's utilization.

Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Not so. Although the hardware manufacturers have been making processors cheaper each year, packing more of them into computers, writing software for multiprocessors is a **more difficult task** than writing sequential code. The major problems are:

- **Functional Decomposition**. There is no algorithm to decompose a single task into multiple, independent, parallel subtasks.
- **Load Balancing**. Decomposing a problem into subtasks in such a way that each has roughly the same amount of computation, to maximize each core's utilization.
- **Data Decomposition**. Dividing the data among the subtasks to minimize IPC and divide the work equally, and to do so in a scalable way.

Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Not so. Although the hardware manufacturers have been making processors cheaper each year, packing more of them into computers, writing software for multiprocessors is a **more difficult task** than writing sequential code. The major problems are:

- **Functional Decomposition**. There is no algorithm to decompose a single task into multiple, independent, parallel subtasks.
- **Load Balancing**. Decomposing a problem into subtasks in such a way that each has roughly the same amount of computation, to maximize each core's utilization.
- **Data Decomposition**. Dividing the data among the subtasks to minimize IPC and divide the work equally, and to do so in a scalable way.
- **Data Dependency Analysis**. When data is distributed to the various subtasks, identifying which data in each subtask depends on the computation performed by a different subtask.

Challenges in Parallel Programming

It sounds simple enough - create programs that use all of the cores in the computer, so that they run faster and utilize the cores fully.

Not so. Although the hardware manufacturers have been making processors cheaper each year, packing more of them into computers, writing software for multiprocessors is a **more difficult task** than writing sequential code. The major problems are:

- **Functional Decomposition**. There is no algorithm to decompose a single task into multiple, independent, parallel subtasks.
- **Load Balancing**. Decomposing a problem into subtasks in such a way that each has roughly the same amount of computation, to maximize each core's utilization.
- **Data Decomposition**. Dividing the data among the subtasks to minimize IPC and divide the work equally, and to do so in a scalable way.
- **Data Dependency Analysis**. When data is distributed to the various subtasks, identifying which data in each subtask depends on the computation performed by a different subtask.
- **Testing and Debugging**. Unlike sequential software, parallel programs can have errors that depend on the relative rates of execution of the subtasks. Designing tests is very difficult and debugging when failure occur is even harder.

Sources of Parallelism in Problems

Two sources of inherent parallelism in computational problems are **data parallelism** and **task parallelism** (also known as **functional parallelism**.)

Sources of Parallelism in Problems

Two sources of inherent parallelism in computational problems are **data parallelism** and **task parallelism** (also known as **functional parallelism**.)

Data parallelism exists when a task can be decomposed into many subtasks that perform identical operations on different sets of data. Examples include

- A digital image on which the same operation must be applied to all pixels or equal-size sub-regions of the image.
- The calculation of payroll taxes for all employees of a large company.
- A digital representation of a 3D shape that must be rotated through same angle in space, since every point in the object undergoes the same linear transformation.

Sources of Parallelism in Problems

Two sources of inherent parallelism in computational problems are **data parallelism** and **task parallelism** (also known as **functional parallelism**.)

Data parallelism exists when a task can be decomposed into many subtasks that perform identical operations on different sets of data. Examples include

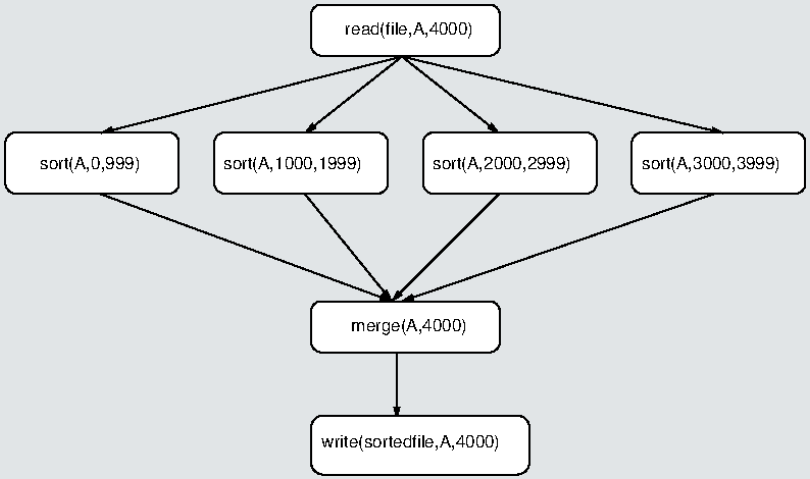
- A digital image on which the same operation must be applied to all pixels or equal-size sub-regions of the image.
- The calculation of payroll taxes for all employees of a large company.
- A digital representation of a 3D shape that must be rotated through same angle in space, since every point in the object undergoes the same linear transformation.

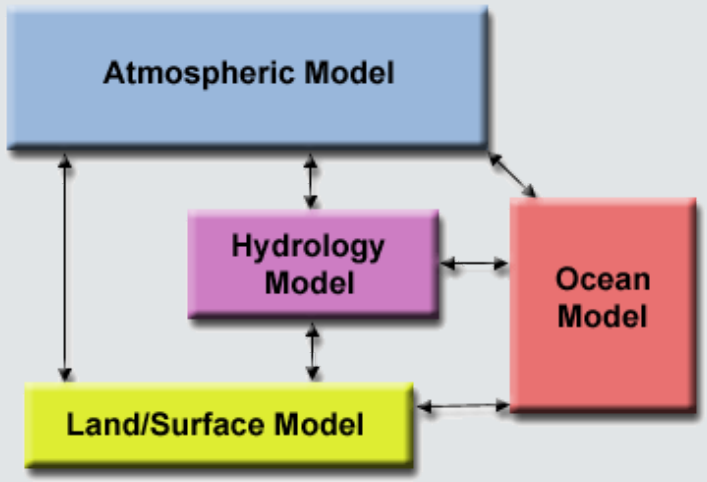
Task parallelism exists when a task can be decomposed into multiple subtasks such that each performs a different function, on the same or different data sets. Examples include

- Analysis of census data by data sub-type, such as demographic analysis, economic analysis, and geographic analysis.
- Audio data processing, in which audio data filters are applied one after another, in a pipeline.

Visual Comparison

A common source of data parallelism exists in linear arrays of data that need to be sorted. Certain sorting algorithms are more amenable than others to decomposition into subtasks that act on segments of the array. The diagram below depicts a variation of a parallel mergesort algorithm.





Meteorological data can be analyzed using a task decomposition. In the diagram above, the same data is input to four different modeling algorithms, to build four different types of environmental models.

Speed-Up

Suppose that you want to add 1 to every element of an array of 1024 elements. A sequential program would increment every element one after the other, performing **1024 increment operations** sequentially.

Speed-Up

Suppose that you want to add 1 to every element of an array of 1024 elements. A sequential program would increment every element one after the other, performing **1024 increment operations** sequentially.

If we had a machine with 1024 cores (not unreasonable in the scientific computing community), and the know-how, we could write a program with 1024 threads that performed this same computation **in the time it takes to execute a single increment**.

Speed-Up

Suppose that you want to add 1 to every element of an array of 1024 elements. A sequential program would increment every element one after the other, performing **1024 increment operations** sequentially.

If we had a machine with 1024 cores (not unreasonable in the scientific computing community), and the know-how, we could write a program with 1024 threads that performed this same computation **in the time it takes to execute a single increment**. **Right?**

Speed-Up

Suppose that you want to add 1 to every element of an array of 1024 elements. A sequential program would increment every element one after the other, performing **1024 increment operations** sequentially.

If we had a machine with 1024 cores (not unreasonable in the scientific computing community), and the know-how, we could write a program with 1024 threads that performed this same computation **in the time it takes to execute a single increment**. **Right?**

Not exactly, because the threads have to access memory to do this, and they might be delayed because of memory stalls. But let's pretend for now that this is not an issue.

Speed-Up

Suppose that you want to add 1 to every element of an array of 1024 elements. A sequential program would increment every element one after the other, performing **1024 increment operations** sequentially.

If we had a machine with 1024 cores (not unreasonable in the scientific computing community), and the know-how, we could write a program with 1024 threads that performed this same computation **in the time it takes to execute a single increment**. **Right?**

Not exactly, because the threads have to access memory to do this, and they might be delayed because of memory stalls. But let's pretend for now that this is not an issue.

The parallel program obviously takes about $1/1024^{th}$ the time that the sequential program takes, or it runs 1024 times faster. This leads to a definition.

Speed-Up

Suppose that you want to add 1 to every element of an array of 1024 elements. A sequential program would increment every element one after the other, performing **1024 increment operations** sequentially.

If we had a machine with 1024 cores (not unreasonable in the scientific computing community), and the know-how, we could write a program with 1024 threads that performed this same computation **in the time it takes to execute a single increment**. **Right?**

Not exactly, because the threads have to access memory to do this, and they might be delayed because of memory stalls. But let's pretend for now that this is not an issue.

The parallel program obviously takes about $1/1024^{th}$ the time that the sequential program takes, or it runs 1024 times faster. This leads to a definition.

The **speed-up** of a parallel program with respect to a sequential program on a computer with N identical processors with the exact same input data is the sequential program's running time on one processor divided by the parallel program's running time on N processors.

Limitations of Parallelization

In 1967, Gene Amdahl argued informally[1] that there was an inherent limitation to the amount of speedup that could be obtained by using more processors to perform a computation.

Although his original article contained no formulas whatsoever, his argument was subsequently formulated mathematically and became known as "Amdahl's Law".

The starting premise is the following observation.

Every program has some fraction of operations in it that must be executed sequentially. For example,

- reading from a file, and
- filling an array `A` such that `A[i+1]`'s value depends on `A[i]`'s value for all `i`

are both inherently sequential. In other words, we cannot simultaneously update `A[i]` and `A[i+1]`.

1 See the references at the end of the slides.

Amdahl's Law

Let f be the fraction of operations in a sequential program's computation on a given input that are inherently sequential. f is often called the **serial fraction**.

Let N be the number of processing cores on the computer on which a parallel version of this same program is run. The speed-up of this parallel version using all N processors has an upper bound given by the formula

$$\text{speedup} \leq \frac{1}{f + \frac{(1-f)}{N}}$$

The value $(1 - f)$ is defined as the **parallel fraction**. This represents the fraction of code that can be run in parallel.

Applications of Amdahl's Law

1. Suppose the serial fraction $f = 0.2$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.2 + \frac{(1-0.2)}{8}} = \frac{1}{0.2 + \frac{(0.8)}{8}} = \frac{1}{0.2 + 0.1} = 3.33$$

Applications of Amdahl's Law

1. Suppose the serial fraction $f = 0.2$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.2 + \frac{(1-0.2)}{8}} = \frac{1}{0.2 + \frac{(0.8)}{8}} = \frac{1}{0.2 + 0.1} = 3.33$$

2. Suppose the serial fraction is $f = 0.04$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{8}} = \frac{1}{0.04 + \frac{(0.96)}{8}} = \frac{1}{0.04 + 0.12} = 6.25$$

Applications of Amdahl's Law

1. Suppose the serial fraction $f = 0.2$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.2 + \frac{(1-0.2)}{8}} = \frac{1}{0.2 + \frac{(0.8)}{8}} = \frac{1}{0.2 + 0.1} = 3.33$$

2. Suppose the serial fraction is $f = 0.04$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{8}} = \frac{1}{0.04 + \frac{(0.96)}{8}} = \frac{1}{0.04 + 0.12} = 6.25$$

   If we increase the number of cores in this example from 8 to 32, how much better can we do?

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{32}} = \frac{1}{0.04 + \frac{(0.96)}{32}} = \frac{1}{0.04 + 0.03} = 14.29$$

Applications of Amdahl's Law

1. Suppose the serial fraction $f = 0.2$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.2 + \frac{(1-0.2)}{8}} = \frac{1}{0.2 + \frac{(0.8)}{8}} = \frac{1}{0.2 + 0.1} = 3.33$$

2. Suppose the serial fraction is $f = 0.04$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{8}} = \frac{1}{0.04 + \frac{(0.96)}{8}} = \frac{1}{0.04 + 0.12} = 6.25$$

   If we increase the number of cores in this example from 8 to 32, how much better can we do?

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{32}} = \frac{1}{0.04 + \frac{(0.96)}{32}} = \frac{1}{0.04 + 0.03} = 14.29$$

   If we keep increasing the number of processors in this example, will speedup keep increasing?

Applications of Amdahl's Law

1. Suppose the serial fraction $f = 0.2$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.2 + \frac{(1-0.2)}{8}} = \frac{1}{0.2 + \frac{(0.8)}{8}} = \frac{1}{0.2 + 0.1} = 3.33$$

2. Suppose the serial fraction is $f = 0.04$. Then the upper bound on speedup for a machine with 8 cores is given by

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{8}} = \frac{1}{0.04 + \frac{(0.96)}{8}} = \frac{1}{0.04 + 0.12} = 6.25$$

If we increase the number of cores in this example from 8 to 32, how much better can we do?

$$\text{speedup} \leq \frac{1}{0.04 + \frac{(1-0.04)}{32}} = \frac{1}{0.04 + \frac{(0.96)}{32}} = \frac{1}{0.04 + 0.03} = 14.29$$

If we keep increasing the number of processors in this example, will speedup keep increasing? No.

Maximum Possible Speedup

The serial fraction itself limits the maximum possible speedup. This is easy to prove.

Let f be the serial fraction of operations in a given program. Then the maximum possible speedup, given any number of processors, is the limit

$$\lim_{N \to \infty} \left(\frac{1}{f + \frac{(1-f)}{N}} \right) = \frac{1}{f}$$

Maximum Possible Speedup

The serial fraction itself limits the maximum possible speedup. This is easy to prove.

Let f be the serial fraction of operations in a given program. Then the maximum possible speedup, given any number of processors, is the limit

$$\lim_{N \to \infty} \left(\frac{1}{f + \frac{(1-f)}{N}} \right) = \frac{1}{f}$$

- If $f = 0.2$, the maximum speedup is $1/0.2 = 5$.
- If $f = 0.5$, the maximum speedup is $1/0.5 = 2$.
- If $f = 0.8$, the maximum speedup is $1/0.8 = 1.25$.

Maximum Possible Speedup

The serial fraction itself limits the maximum possible speedup. This is easy to prove.

Let f be the serial fraction of operations in a given program. Then the maximum possible speedup, given any number of processors, is the limit

$$\lim_{N \to \infty} \left(\frac{1}{f + \frac{(1-f)}{N}} \right) = \frac{1}{f}$$

- If $f = 0.2$, the maximum speedup is $1/0.2 = 5$.
- If $f = 0.5$, the maximum speedup is $1/0.5 = 2$.
- If $f = 0.8$, the maximum speedup is $1/0.8 = 1.25$.

As $f \to 1$, the maximum possible speedup approaches 1, meaning no speedup at all.

Maximum Possible Speedup

The serial fraction itself limits the maximum possible speedup. This is easy to prove.

Let f be the serial fraction of operations in a given program. Then the maximum possible speedup, given any number of processors, is the limit

$$\lim_{N\to\infty}\left(\frac{1}{f+\frac{(1-f)}{N}}\right)=\frac{1}{f}$$

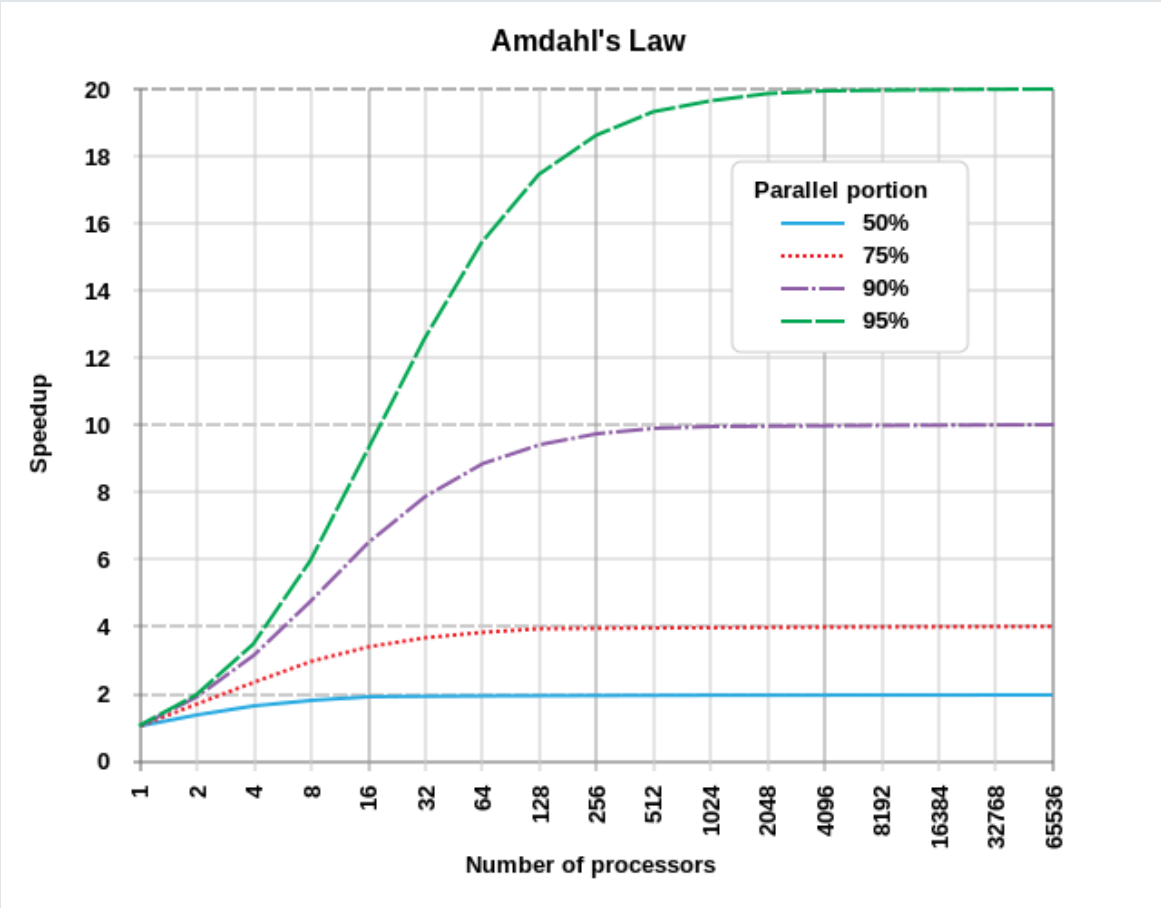
- If $f=0.2$, the maximum speedup is $1/0.2=5$.
- If $f=0.5$, the maximum speedup is $1/0.5=2$.
- If $f=0.8$, the maximum speedup is $1/0.8=1.25$.

As $f\to 1$, the maximum possible speedup approaches 1, meaning no speedup at all.

In short, you cannot speed up a program that has little opportunity to be parallelized!

Plot of Amdahl's Law

Below is a plot of the speedup predicted by Amdahl's Law for various fractions of inherently sequential code. The legend labels the curves using the parallel fraction $1 - f$ rather than f
.



Plot by Daniels220 at English Wikipedia, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6678551

Inverting Amdahl's Law

Suppose that f is the serial fraction of operations in a given program. The maximum possible speedup is $\frac{1}{f}$.

Suppose we want to achieve a specific speedup S less than $\frac{1}{f}$.

How many processors are needed?

1. Use the formula for speedup to write N as a function of f and S.
2. For $f = 0.1$, find the least number of processors needed to obtain a speedup of 5.
3. What if we want a speedup of 8?

Multi-threading Models for Explicit Threading

We examine different methods of implementing support for programmer-defined threads.

Multi-threading Models

Ultimately the goal is to be able to write programs that create threads and manipulate them. How can one do this?

Multi-threading Models

Ultimately the goal is to be able to write programs that create threads and manipulate them. How can one do this?

- Are threads built into programming languages?

Multi-threading Models

Ultimately the goal is to be able to write programs that create threads and manipulate them. How can one do this?

- Are threads built into programming languages?
- Are they supported through the use of special libraries to which a program can link?

Multi-threading Models

Ultimately the goal is to be able to write programs that create threads and manipulate them. How can one do this?

- Are threads built into programming languages?
- Are they supported through the use of special libraries to which a program can link?
- Are they supported directly by the kernel, so that a program must request thread services directly from the kernel?

Multi-threading Models

Ultimately the goal is to be able to write programs that create threads and manipulate them. How can one do this?

- Are threads built into programming languages?
- Are they supported through the use of special libraries to which a program can link?
- Are they supported directly by the kernel, so that a program must request thread services directly from the kernel?

These are all valid questions, and by the end of this section you will understand what the various options are and what limitations each has.

Multi-threading Models

Ultimately the goal is to be able to write programs that create threads and manipulate them. How can one do this?

- Are threads built into programming languages?
- Are they supported through the use of special libraries to which a program can link?
- Are they supported directly by the kernel, so that a program must request thread services directly from the kernel?

These are all valid questions, and by the end of this section you will understand what the various options are and what limitations each has.

We are about to explore two different methods by which computer systems allow programmers to create multithreaded programs. They are called **multithreading models**.

The difference between the methods is that, in one, threads are implemented in user space, and in the other, in kernel space.

Thread Implementations

There are two general ways in which threads can be implemented, which are often called **user level threads** and **kernel level threads**.

Thread Implementations

There are two general ways in which threads can be implemented, which are often called **user level threads** and **kernel level threads**.

User level threads are threads that are managed by a user level thread library. The thread library contains code for creating and destroying threads, for inter-thread communication, scheduling, saving and restoring thread contexts, and all other thread management operations.

The three most commonly used thread libraries are `POSIX Threads`, `Windows Threads`, and `Java Threads`.

Thread Implementations

There are two general ways in which threads can be implemented, which are often called **user level threads** and **kernel level threads**.

User level threads are threads that are managed by a user level thread library. The thread library contains code for creating and destroying threads, for inter-thread communication, scheduling, saving and restoring thread contexts, and all other thread management operations.

The three most commonly used thread libraries are `POSIX Threads`, `Windows Threads`, and `Java Threads`.

Kernel level threads are managed entirely within the kernel. There is no thread management code in user space. Creating, destroying, scheduling, coordinating, and otherwise managing threads is performed completely within kernel code. A kernel level thread is sometimes called a **light weight process** (**LWP**). Almost all modern operating systems support kernel level threads.

Thread Implementations

There are two general ways in which threads can be implemented, which are often called **user level threads** and **kernel level threads**.

User level threads are threads that are managed by a user level thread library. The thread library contains code for creating and destroying threads, for inter-thread communication, scheduling, saving and restoring thread contexts, and all other thread management operations.

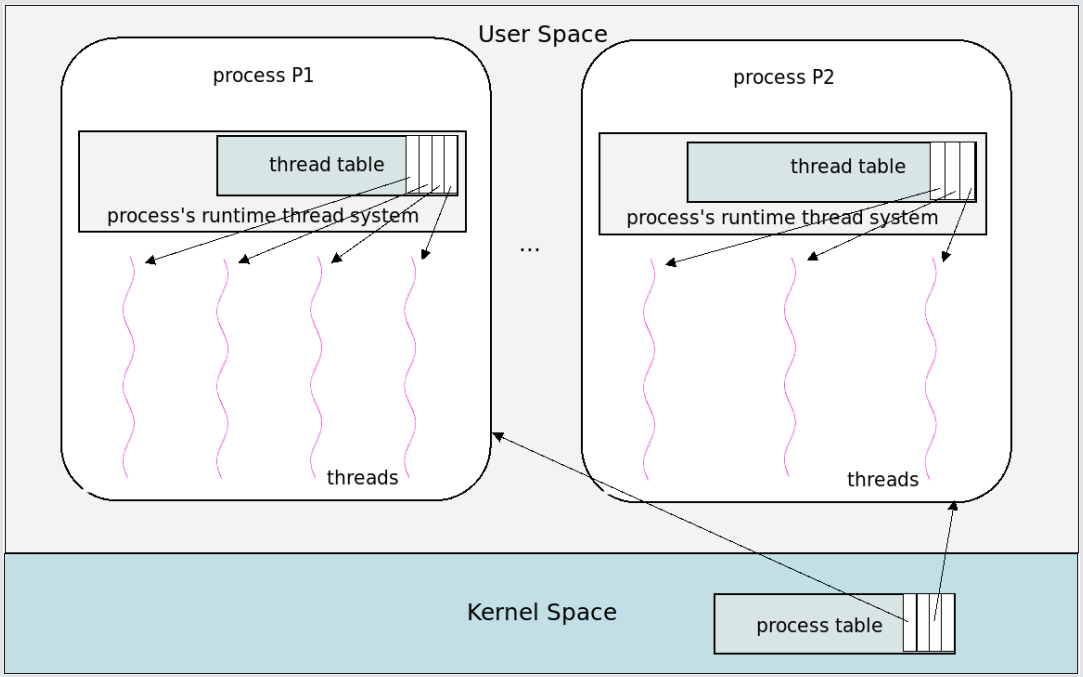
The three most commonly used thread libraries are `POSIX Threads`, `Windows Threads`, and `Java Threads`.

Kernel level threads are managed entirely within the kernel. There is no thread management code in user space. Creating, destroying, scheduling, coordinating, and otherwise managing threads is performed completely within kernel code. A kernel level thread is sometimes called a **light weight process** (**LWP**). Almost all modern operating systems support kernel level threads.

We begin by examining user level threads. After that we examine kernel level threads.

User Level Threads

User level threads are implemented by special thread libraries. A program with multiple threads is linked into the thread library, which handles all aspects of thread management. In the figure below, there are two processes, one with 4 user threads, the other, with 3.



The thread support is handled within the library entirely, in user space. Notice that there are no threads depicted in the kernel in this figure. With user level threads, it is possible to have a multithreaded program in an operating system that may or may not have kernel support for threading. It is independent of kernel support.

A POSIX Threads Example

One of the most common user thread libraries is the one standardized by POSIX, which specifies an API for multithreaded programs commonly known as POSIX threads or Pthreads. This interface is implemented in almost all modern operating systems.

Below is a simple example of a Pthreads program that creates a single child thread.

```
#include <pthread.h> /* Includes of other header files omitted to save space */

void * hello_world( void * unused)
{
    printf("The child says, \"Hello world!\"\n");
    pthread_exit(NULL) ;
}

int main( int argc, char *argv[])
{
    pthread_t  child_thread;
    /* Create and launch thread */
    if ( 0 != pthread_create(&child_thread, NULL, hello_world, NULL ) ){
        exit(1);
    }
    printf("This is the parent thread.\n");
    pthread_join(child_thread, NULL);  /* Wait for the child thread to terminate. */
    return 0;
}
```

Some Explanation

In the preceding program, we highlighted the three calls to functions from the Pthreads API:

```
pthread_exit()
pthread_create()
pthread_join()
```

The `pthread_exit()` call terminates the calling thread. The `pthread_create()` call creates a new thread, called the **child** thread. The caller is the **parent** thread. The `pthread_join()` call makes the main program wait until the child thread makes a call to `pthread_exit()`.

Some Explanation

In the preceding program, we highlighted the three calls to functions from the Pthreads API:

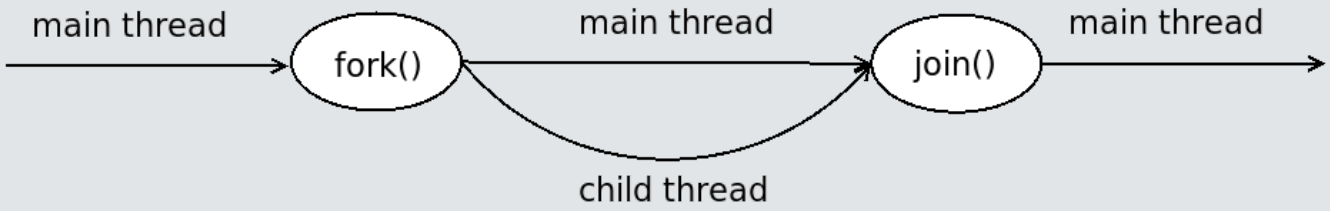
```
pthread_exit()
pthread_create()
pthread_join()
```

The `pthread_exit()` call terminates the calling thread. The `pthread_create()` call creates a new thread, called the **child** thread. The caller is the **parent** thread. The `pthread_join()` call makes the main program wait until the child thread makes a call to `pthread_exit()`.

An important point about these functions is that the library implements them by making calls to the underlying operating system thread support system, if it exists. If not, it simulates the operations within the process. This will be explained more later.

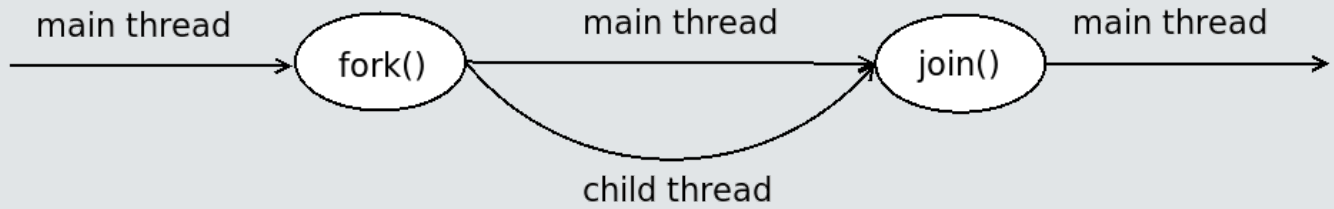
Fork-Join Flow

The parallel flow when `pthread_create()` and `pthread_join()` are used is illustrated by the diagram below.

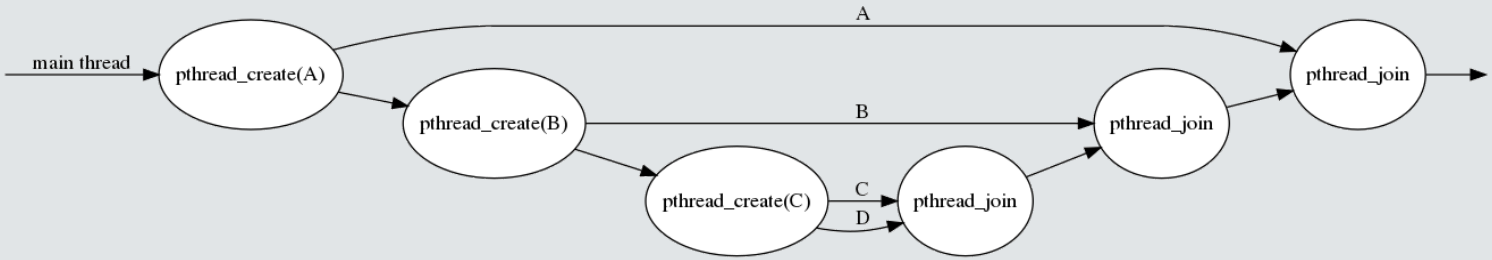


Fork-Join Flow

The parallel flow when `pthread_create()` and `pthread_join()` are used is illustrated by the diagram below.

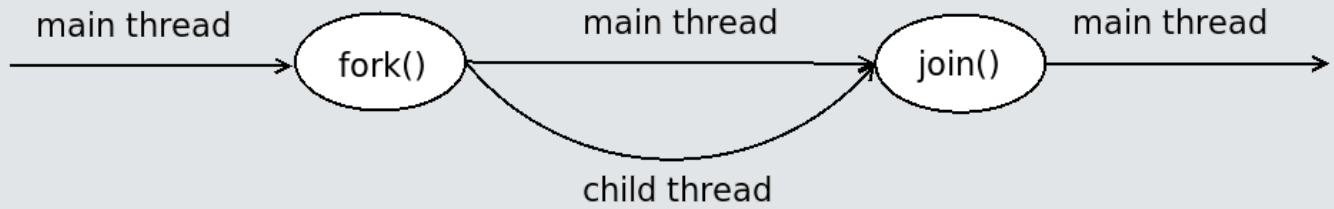


Notice that, after the `join()` call, only the parent thread continues. In this paradigm, a thread creates a child thread and waits for it to finish, after which only the parent thread continues. When multiple threads need to be created, the flow looks something like the diagram below. There is no way to create more than one thread at a time.

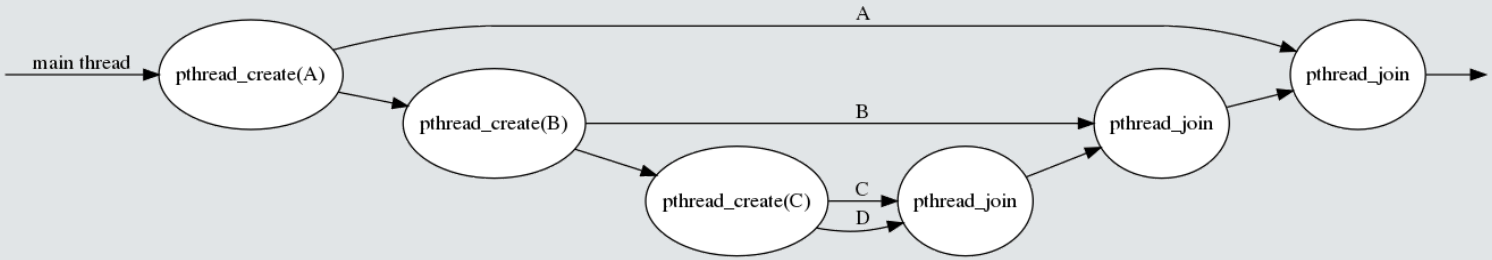


Fork-Join Flow

The parallel flow when `pthread_create()` and `pthread_join()` are used is illustrated by the diagram below.



Notice that, after the `join()` call, only the parent thread continues. In this paradigm, a thread creates a child thread and waits for it to finish, after which only the parent thread continues. When multiple threads need to be created, the flow looks something like the diagram below. There is no way to create more than one thread at a time.



The fork-join paradigm is fundamental to how multithreading is achieved in thread libraries such as Pthreads. It also underlies a type of threading model known as **fork-join parallelism** which we will discuss later in this chapter.

Pthread Activity

The Pthreads program from the Example slide can be found on the network in the `demos` directory. Copy it to your own directory and modify it so that it creates two threads. The second thread will print "Goodbye".

The main program should be designed so that the first thread to print will be the one that prints "Hello World". Look up the `sleep()` function.

The main program should wait for each child thread to terminate.

User Level Threads: Pros

The benefits of user level threads are that:

- Threads can be created very quickly, depending on the underlying implementation. Usually, few, if any, system calls are needed.

User Level Threads: Pros

The benefits of user level threads are that:

- Threads can be created very quickly, depending on the underlying implementation. Usually, few, if any, system calls are needed.
- Switching from one thread to another is also very fast since no context switch is required (because all threads are in user space.)

User Level Threads: Pros

The benefits of user level threads are that:

- Threads can be created very quickly, depending on the underlying implementation. Usually, few, if any, system calls are needed.
- Switching from one thread to another is also very fast since no context switch is required (because all threads are in user space.)
- The kernel does not need to have any support for threads for user programs to be multithreaded. All threads run in the process's context.

User Level Threads: Pros

The benefits of user level threads are that:

- Threads can be created very quickly, depending on the underlying implementation. Usually, few, if any, system calls are needed.
- Switching from one thread to another is also very fast since no context switch is required (because all threads are in user space.)
- The kernel does not need to have any support for threads for user programs to be multithreaded. All threads run in the process's context.
- Because the library is in user space, code written to run against its API can be run on any computer system for which the library has been implemented.

User Level Threads: Cons

The drawbacks of user threads are that:

- A process with dozens of threads may get the same amount of time on the processor as one with a single thread, depending on the implementation, so the fact that it has many threads does not give it more processor time.

User Level Threads: Cons

The drawbacks of user threads are that:

- A process with dozens of threads may get the same amount of time on the processor as one with a single thread, depending on the implementation, so the fact that it has many threads does not give it more processor time.
- A program with multiple threads may not be able to take advantage of multiple processors, since all threads may be mapped to a single processor, depending on the implementation.

User Level Threads: Cons

The drawbacks of user threads are that:

- A process with dozens of threads may get the same amount of time on the processor as one with a single thread, depending on the implementation, so the fact that it has many threads does not give it more processor time.
- A program with multiple threads may not be able to take advantage of multiple processors, since all threads may be mapped to a single processor, depending on the implementation.
- The application programmer generally has no control over how threads are mapped to processors.

Kernel Level Threads

Whereas user level threads are implemented in a user level library, kernel level threads are implemented directly within the kernel. Unlike user level threads, each thread can be individually scheduled.

The kernel also performs thread creation, thread deletion, and all thread management in general. There is no code in the user space for managing the threads, although they exist in user space.

In this sense they are like user processes: processes are created and managed by system calls to the kernel, but they exist in user space and have user privileges.

Kernel Level Threads: The Confusion

The term "kernel level threads" refers to a method by which all applications and programs, whether user level or system level, can be multithreaded by using threads supported directly by the kernel.

Kernel Level Threads: The Confusion

The term "kernel level threads" refers to a method by which all applications and programs, whether user level or system level, can be multithreaded by using threads supported directly by the kernel.

To make this possible, it requires that

- the kernel itself can create and manage threads, and
- although the threads in the multithreaded program are created and managed by the kernel, they are part of the program and have its privileges and share its address space by default.

Kernel Level Threads: The Confusion

The term "kernel level threads" refers to a method by which all applications and programs, whether user level or system level, can be multithreaded by using threads supported directly by the kernel.

To make this possible, it requires that

- the kernel itself can create and manage threads, and
- although the threads in the multithreaded program are created and managed by the kernel, they are part of the program and have its privileges and share its address space by default.

Kernels that can create threads are usually multithreaded themselves, because they create threads to improve their own performance.

This gives rise to a natural confusion in the terminology:

- Threads that run as part of the kernel, in its address space, are called **kernel threads**.
- Many people also use this same term as a shorthand for "kernel level threads"!

Kernel Level Threads: The Confusion

The term "kernel level threads" refers to a method by which all applications and programs, whether user level or system level, can be multithreaded by using threads supported directly by the kernel.

To make this possible, it requires that

- the kernel itself can create and manage threads, and
- although the threads in the multithreaded program are created and managed by the kernel, they are part of the program and have its privileges and share its address space by default.

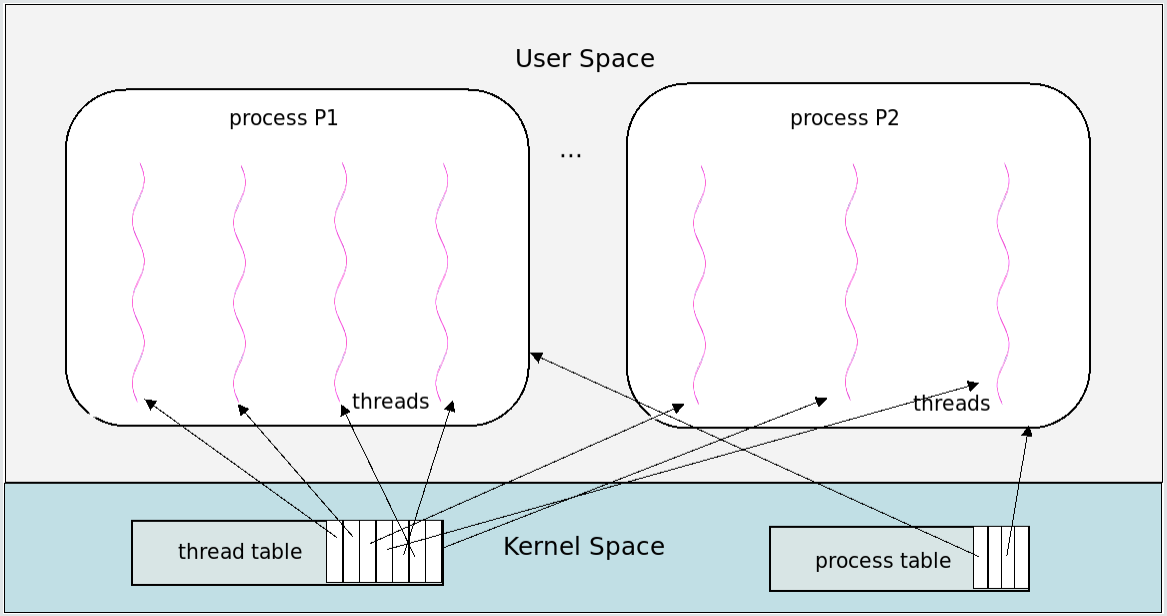
Kernels that can create threads are usually multithreaded themselves, because they create threads to improve their own performance.

This gives rise to a natural confusion in the terminology:

- Threads that run as part of the kernel, in its address space, are called **kernel threads**.
- Many people also use this same term as a shorthand for "kernel level threads"!

Kernel threads are not the same thing as kernel level threads. The former are threads inside kernel space; the latter are user-space threads scheduled and managed by the kernel.

Kernel Level Threads: A Visualization



In the above figure, the two process's threads are kernel level threads. You can tell this because the thread table that keeps track of the threads is in the kernel's space, not user space.

The figure shows the entries in the kernel's thread table as pointers to the threads in the processes. In practice, the thread table consists of data structures that represent these threads. The pointers in the figure are for illustration only.

About Kernel Level Threads in Linux

Linux has a unique implementation of threads, because it treats all threads as standard processes. It does not provide any special scheduling or data structures for threads.

About Kernel Level Threads in Linux

Linux has a unique implementation of threads, because it treats all threads as standard processes. It does not provide any special scheduling or data structures for threads.

To the Linux kernel, processes and threads are both **tasks** and are both represented by a `task_struct`.

About Kernel Level Threads in Linux

Linux has a unique implementation of threads, because it treats all threads as standard processes. It does not provide any special scheduling or data structures for threads.

To the Linux kernel, processes and threads are both **tasks** and are both represented by a `task_struct`.

What distinguishes threads from ordinary processes in Linux is that **threads can share resources, such as their address space, whereas processes do not share any resources.**

About Kernel Level Threads in Linux

Linux has a unique implementation of threads, because it treats all threads as standard processes. It does not provide any special scheduling or data structures for threads.

To the Linux kernel, processes and threads are both **tasks** and are both represented by a `task_struct`.

What distinguishes threads from ordinary processes in Linux is that **threads can share resources, such as their address space, whereas processes do not share any resources.**

The implication for the programmer is that the same system call can be used to create kernel level threads as is used to create processes:

```
#define _GNU_SOURCE
#include <sched.h>

int clone(int (*fn)(void *), void *child_stack,
              int flags, void *arg, ...
             /* pid_t *ptid, void *newtls, pid_t *ctid */ );
```

Because it is so generic, it is complicated to use; the `flags` parameter has many possible values that tell the kernel which resources are shared.

The next slide has a small program that demonstrates using `clone()` to create a thread.

Linux Kernel Level Thread Example

```
/* #includes of all header files omitted to save space */

static int child_function(void* arg) /* Function executed by child thread */
{
    char* buf = (char*) arg;
    printf("Child gets buffer containing the string:\n   \"%s\"\n\n", buf);
    strcpy(buf, "Teach your parents well");
    return 0;
}

int main(int argc, char* argv[])
{
    const int STACK_SIZE = 65536;    /* Allocate stack for child thread */
    char* stack = malloc(STACK_SIZE);
    if ( NULL == stack )   exit(1);

    unsigned long flags = CLONE_VM | SIGCHLD; /* share address space */
    char buf[256];
    strcpy(buf, "You, who are on the road must have a code that you can live by.");
    if (-1 == clone(child_function, stack + STACK_SIZE, flags, buf) )  exit(1);

    int status;
    if (-1 == wait(&status) )  exit(1);
    printf("Thread exited with status %d. It filled buffer with:\n   \"%s\"\n",status,buf);
    return 0;
}
```

Linux Kernel Level Thread Example

When the preceding program is run, the session looks like

```
$ kernel_thread_demo
Child gets buffer containing the string:
   "You, who are on the road must have a code that you can live by."

Child exited with status 0. It filled the buffer with:
   "Teach your parents well"
$
```

Some Notes:

- The program must dynamically create a stack for the cloned thread. Because stacks grow from high to low memory, the starting address of the stack is the value of the address returned by `malloc()` plus the stack size.
- The `CLONE_VM` flag is passed to `clone()`. This tells the kernel that the parent's memory is to be shared with the child rather than copied.
- The `SIGCHLD` flag tells the kernel that the child will call `exit()` and the parent is going to `wait()` for the `SIGCHLD` signal to receive the child's termination status.

Kernel Level Threads: Pros and Cons

The benefits of kernel level threads are that:

- Kernel level threads from a single process can be scheduled simultaneously on multiple processors, taking advantage of the hardware.
- A thread that blocks as a result of a service request does not prevent other threads in the same process from being scheduled to run. This is a big advantage to highly interactive programs that can block frequently for short durations.
- The kernel can allocate more processor time to processes with larger numbers of threads.
- The kernel itself can be multithreaded.

Kernel Level Threads: Pros and Cons

The benefits of kernel level threads are that:

- Kernel level threads from a single process can be scheduled simultaneously on multiple processors, taking advantage of the hardware.
- A thread that blocks as a result of a service request does not prevent other threads in the same process from being scheduled to run. This is a big advantage to highly interactive programs that can block frequently for short durations.
- The kernel can allocate more processor time to processes with larger numbers of threads.
- The kernel itself can be multithreaded.

The drawbacks of kernel level threads are that:

- It is slower to create kernel level threads and more work to manage them because there is more work in the kernel.
- Switching between threads in the same process requires kernel intervention and is therefore slower.
- Representing threads within the kernel requires a complete PCB.

Implementation of User Level Threads

A user level thread library can be implemented in a few different ways, depending on the support from the underlying kernel.

Implementation of User Level Threads

A user level thread library can be implemented in a few different ways, depending on the support from the underlying kernel.

If the underlying kernel has no support for threading at all, then the user level thread library must simulate all thread operations in user space. The library in this case uses what we call the Many-to-One threading model, which we explain first.

Implementation of User Level Threads

A user level thread library can be implemented in a few different ways, depending on the support from the underlying kernel.

If the underlying kernel has no support for threading at all, then the user level thread library must simulate all thread operations in user space. The library in this case uses what we call the Many-to-One threading model, which we explain first.

If, on the other hand, the kernel has support for threading, there are two different choices of thread implementation:

- The One-to-One Model
- The Many-to-Many Model

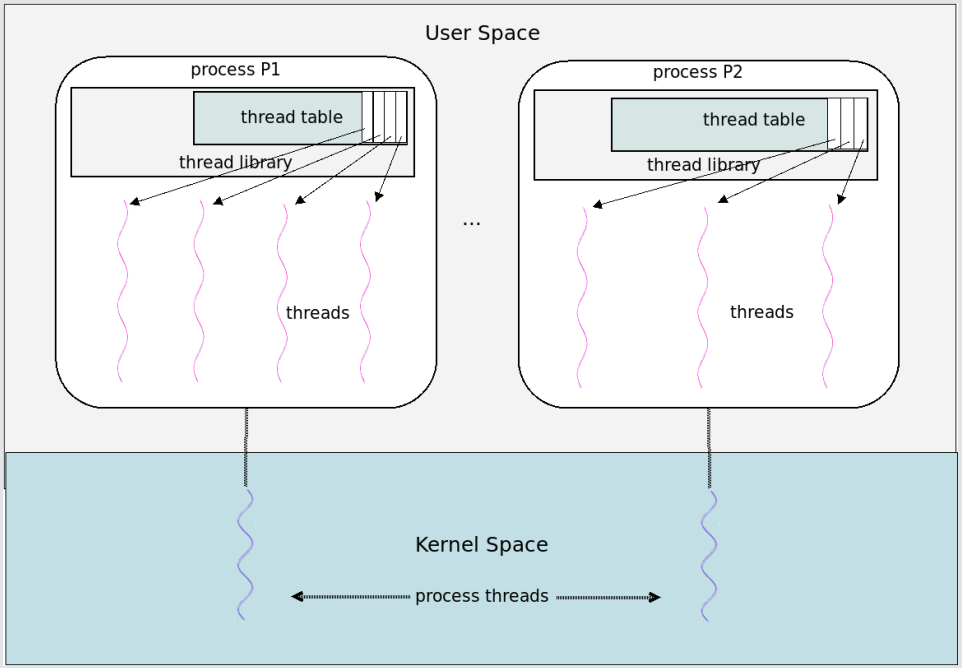
The differences between the different models have to do with how user level threads are related to kernel level threads.

Many-to-One (M:1) Threading Model

This method is rarely used anymore and is mostly of historical interest[1]. It was used originally because it requires no support for multithreading at the kernel level and is the most portable, but it has significant performance problems.

All threads are implemented in user space within the process, which appears to the kernel to be an ordinary, single-threaded process.

In the figure to the right, we draw the process threads on which the user-level threads run, inside the kernel space. This does not mean that the process executes kernel code. It means that the data structures that represent the process are in kernel space.



1 The Green Threads library in Solaris and early Pthreads on Linux used this approach.

Many-to-One (M:1) Threading Model Pros and Cons

The major **benefits** of this model are that it is portable and does not require any support from the underlying kernel.

There are a few **drawbacks**.

Many-to-One (M:1) Threading Model Pros and Cons

The major **benefits** of this model are that it is portable and does not require any support from the underlying kernel.

There are a few **drawbacks**.

- One drawback is that it requires all blocking system calls to be simulated in the library by non-blocking calls to the kernel, which slows down system calls significantly.

Many-to-One (M:1) Threading Model Pros and Cons

The major **benefits** of this model are that it is portable and does not require any support from the underlying kernel.

There are a few **drawbacks**.

- One drawback is that it requires all blocking system calls to be simulated in the library by non-blocking calls to the kernel, which slows down system calls significantly.
- A second is a consequence of the first. Some **blocking system calls cannot be simulated by non-blocking system calls**; as a result, when a thread makes such a call the entire process is blocked.

Many-to-One (M:1) Threading Model Pros and Cons

The major **benefits** of this model are that it is portable and does not require any support from the underlying kernel.

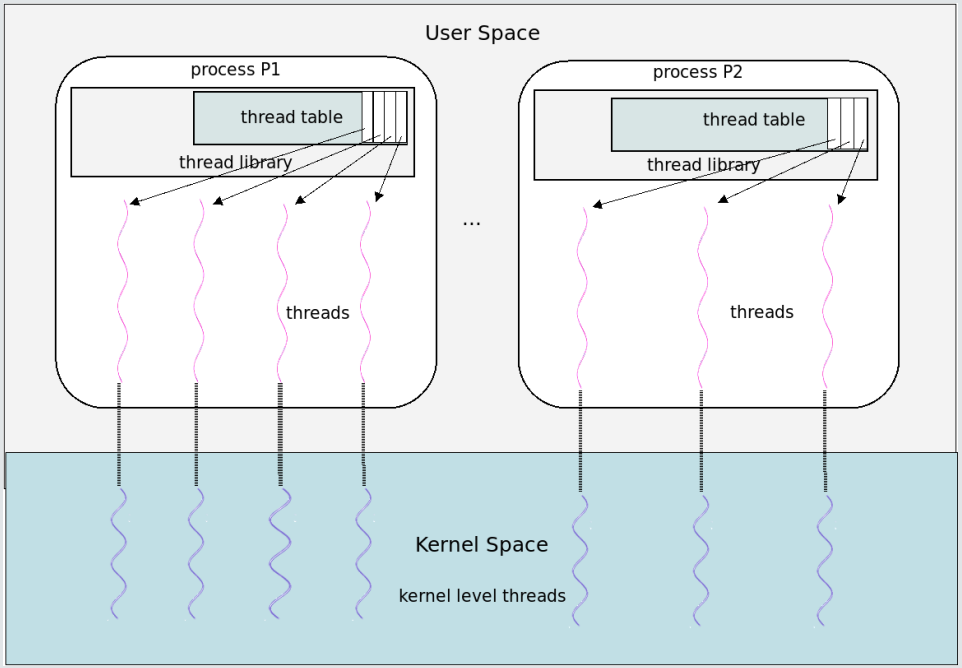
There are a few **drawbacks**.

- One drawback is that it requires all blocking system calls to be simulated in the library by non-blocking calls to the kernel, which slows down system calls significantly.
- A second is a consequence of the first. Some **blocking system calls cannot be simulated by non-blocking system calls**; as a result, when a thread makes such a call the entire process is blocked.
- A third, major drawback is that a **program cannot take advantage of more than a single processor** because the kernel sees it as a single-threaded process (as a single schedulable unit.)

One-to-One (1:1) Threading Model

The **One-to-One model** assigns a kernel level thread to each user level thread. Implementations of `Pthreads` in current Linux versions and **Windows** systems use this approach.

From now on, kernel level threads are drawn in kernel space to indicate that the data structures and code that represent and manage them is in kernel space.



In the figure above, each user level thread is associated with a real kernel level thread. In Linux, this simply means that the library uses the `clone()` system call to create the threads. **Each thread is seen by the kernel as a separately schedulable entity.**

One-to-One (1:1) Threading Model Pros and Cons

This model has several **benefits**:

- It is the simplest to implement within a library,
- It provides the greatest possible concurrency because it can use as many processors as are available, up to the number of threads, and
- One thread's blocking does not block other threads.

One-to-One (1:1) Threading Model Pros and Cons

This model has several **benefits**:

- It is the simplest to implement within a library,
- It provides the greatest possible concurrency because it can use as many processors as are available, up to the number of threads, and
- One thread's blocking does not block other threads.

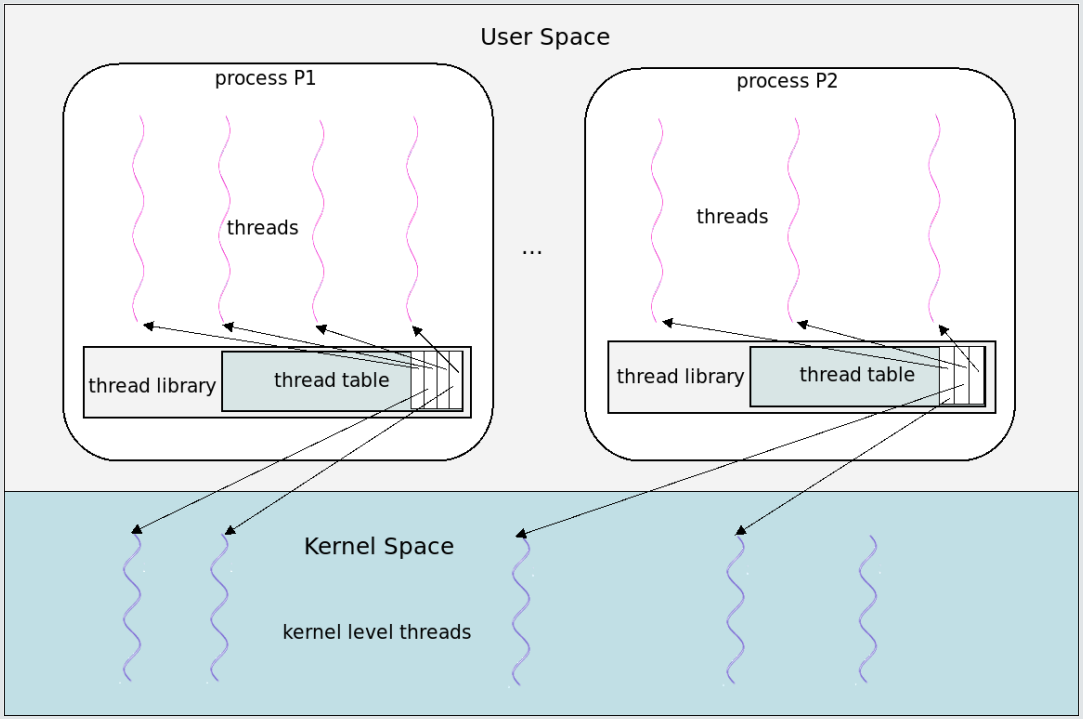
However, the **drawbacks** are the following.

- Creating each thread is more expensive in terms of kernel resources,
- Thread management is slower because most operations on threads require system calls, and
- It is dependent on the multithreading model of the underlying kernel.

Many-to-Many (M:N) Threading Model

The **Many-to-Many model** is the most flexible of these models. It does not create a kernel level thread for each user level thread like the (1:1) model, nor does it force all of a program's threads to be scheduled on a single kernel level thread.

Instead, the library **creates multiple kernel level threads and schedules user level threads on top of them.** Most M:N thread libraries will dynamically allocate as many kernel level threads as necessary to service the user level threads that are ready to run.



Many-to-Many (M:N) Threading Model Pros and Cons

This model has several benefits. The most significant include:

- It does not use kernel resources for user level threads that are not actually runnable.
- The library-level scheduler can switch between threads much faster because it does not make system calls.
- It performs better than the others when user level threads synchronize with each other.
- Applications that create large numbers of threads that only run occasionally perform better on this model than in the others.

Many-to-Many (M:N) Threading Model Pros and Cons

This model has several benefits. The most significant include:

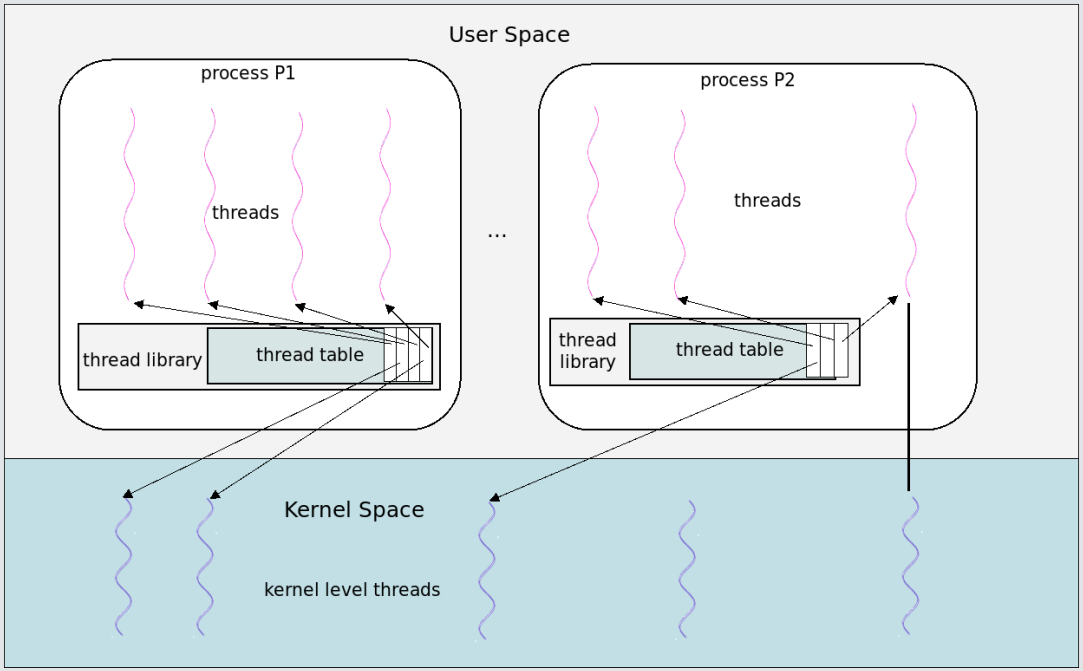
- It does not use kernel resources for user level threads that are not actually runnable.
- The library-level scheduler can switch between threads much faster because it does not make system calls.
- It performs better than the others when user level threads synchronize with each other.
- Applications that create large numbers of threads that only run occasionally perform better on this model than in the others.

The drawbacks of this model include:

- It has more overhead and consumes more system resources because scheduling takes place in both the kernel among the kernel level threads for the process and in the library for the user level threads.
- User level threads that are bound to the same kernel level thread can still be blocked when the thread that is running makes a blocking system call.

Two-Level Threading Model

The two-level model is similar to the M:N model except that it also allows some user-level threads to be bound to a single kernel-level thread. This is useful when certain threads should not be prevented from running because a thread that is sharing its kernel level thread blocks.



Scheduler Activations

In the M:N and two-level models, when an underlying kernel level thread blocks, there is no way for the kernel to notify the user level thread manager that the thread has blocked, so that it can allocate more kernel level threads.

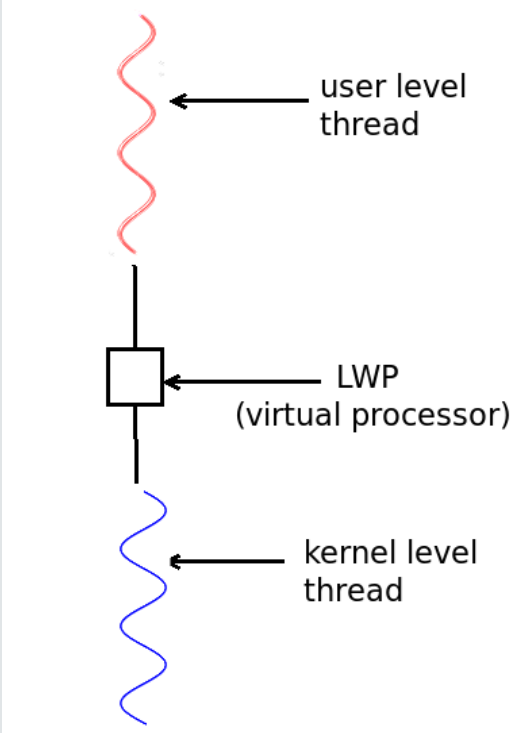
Scheduler Activations

In the M:N and two-level models, when an underlying kernel level thread blocks, there is no way for the kernel to notify the user level thread manager that the thread has blocked, so that it can allocate more kernel level threads.

In 1991, a model called **scheduler activations**[1] was proposed to overcome this problem.

In this model, the kernel provides the process with the abstraction of **virtual processors**. To a user process, a virtual processor acts like an actual processor on which a user level thread can run. It is however just a data structure, sometimes called a **Light Weight Process** (**LWP**).

Each LWP is bound to a kernel level thread. The kernel level thread is scheduled to run on an available processor. If it blocks, the LWP blocks, and the user level thread attached to the LWP blocks.



1 Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. "Scheduler activations: Effective kernel support for the user-level management of parallelism." *Proc. 19th ACM Symposium on Operating System Principles*, pp. 95-109, 1991.

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

1. An executing thread makes a blocking system call.

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

1. An executing thread makes a blocking system call.
2. The kernel blocks the calling user level thread as well as the kernel level thread used to execute the user level thread.

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

1. An executing thread makes a blocking system call.
2. The kernel blocks the calling user level thread as well as the kernel level thread used to execute the user level thread.
3. **Scheduler activation**: The kernel allocates a new virtual processor to the process.

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

1. An executing thread makes a blocking system call.
2. The kernel blocks the calling user level thread as well as the kernel level thread used to execute the user level thread.
3. **Scheduler activation**: The kernel allocates a new virtual processor to the process.
4. **Upcall**: The kernel notifies the user level thread manager that the user level thread blocked and that a new virtual processor is available for other threads.

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

1. An executing thread makes a blocking system call.
2. The kernel blocks the calling user level thread as well as the kernel level thread used to execute the user level thread.
3. **Scheduler activation**: The kernel allocates a new virtual processor to the process.
4. **Upcall**: The kernel notifies the user level thread manager that the user level thread blocked and that a new virtual processor is available for other threads.
5. The user process runs an upcall handler on the new virtual processor. It saves the state of the blocked thread and frees the virtual processor on which that thread was running.

Scheduler Activation Use

Kernel events that could affect the number of runnable threads, such as blocking system calls, are communicated directly to the user process using **upcalls**, which are messages sent to the user level thread manager.

Upcalls are handled within the thread manager by an **upcall handler** (which is like a signal handler). The upcall handlers must run on a virtual processor.

Example Sequence

1. An executing thread makes a blocking system call.
2. The kernel blocks the calling user level thread as well as the kernel level thread used to execute the user level thread.
3. **Scheduler activation**: The kernel allocates a new virtual processor to the process.
4. **Upcall**: The kernel notifies the user level thread manager that the user level thread blocked and that a new virtual processor is available for other threads.
5. The user process runs an upcall handler on the new virtual processor. It saves the state of the blocked thread and frees the virtual processor on which that thread was running.
6. The user level thread manager moves the other threads to the new virtual processor and schedules one of the ready threads to run on it.

Implicit Threading

In this section we examine ways in which multithreaded programs can be generated from single-threaded programs with varying amounts of programmer direction.

Implicit Threading

It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks. Some of the reasons are:

Implicit Threading

It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks. Some of the reasons are:

- Identifying the parallelism in a problem is difficult and there is no algorithm that can do this in general.

Implicit Threading

It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks. Some of the reasons are:

- Identifying the parallelism in a problem is difficult and there is no algorithm that can do this in general.
- Defining the individual tasks, determining how to distribute data to them, and mapping them to processors and load balancing for optimal performance are difficult and there are no algorithms that can do these things in all cases.

Implicit Threading

It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks. Some of the reasons are:

- Identifying the parallelism in a problem is difficult and there is no algorithm that can do this in general.
- Defining the individual tasks, determining how to distribute data to them, and mapping them to processors and load balancing for optimal performance are difficult and there are no algorithms that can do these things in all cases.
- Handling coordination and communication among threads is error-prone.

Implicit Threading

It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks. Some of the reasons are:

- Identifying the parallelism in a problem is difficult and there is no algorithm that can do this in general.
- Defining the individual tasks, determining how to distribute data to them, and mapping them to processors and load balancing for optimal performance are difficult and there are no algorithms that can do these things in all cases.
- Handling coordination and communication among threads is error-prone.

One way to overcome these difficulties is to **offload some of the programming tasks from developers to compilers and run-time libraries.**

Implicit Threading

It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks. Some of the reasons are:

- Identifying the parallelism in a problem is difficult and there is no algorithm that can do this in general.
- Defining the individual tasks, determining how to distribute data to them, and mapping them to processors and load balancing for optimal performance are difficult and there are no algorithms that can do these things in all cases.
- Handling coordination and communication among threads is error-prone.

One way to overcome these difficulties is to **offload some of the programming tasks from developers to compilers and run-time libraries.**

Implicit threading refers to any of several different methods of multithreading a program in which compilers and/or libraries create and manage concurrent threads with little or no explicit guidance from the programmer.

The "implicit" aspect of it is the creation and management of threads, not the specification of parallelism.

Parallelizing Compilers

Much research has been put into the design of compilers that free programmers completely from the tasks of multithreading programs.

For example, it is possible for a compiler to detect that, in a sequential loop such as this:

```
for ( int i = 0; i < 100; i++)
    a[i] = b[i] + c[i];
```

the loop body can be executed in parallel by up to 100 different threads simultaneously[1].

These parallelizing compilers analyze the code and determine automatically where fine-grained parallelism[2] exists. They can then arrange for selected instructions to be executed by separate threads.

1 There are still bottlenecks with respect to accessing to the shared memory.
2 Fine-grained parallelism is parallelism in which the parallel tasks are just a few instructions long. When the tasks are entire functions, it is considered to be coarse-grained.

Parallelizing Compilers

Much research has been put into the design of compilers that free programmers completely from the tasks of multithreading programs.

For example, it is possible for a compiler to detect that, in a sequential loop such as this:

```
for ( int i = 0; i < 100; i++)
    a[i] = b[i] + c[i];
```

the loop body can be executed in parallel by up to 100 different threads simultaneously[1].

These parallelizing compilers analyze the code and determine automatically where fine-grained parallelism[2] exists. They can then arrange for selected instructions to be executed by separate threads.

Studies have shown that 90% of the execution time of most programs is spent in 10% of the code, mostly in loops. Therefore, much of the effort is in detecting the parallelism in loops. Unfortunately there are many difficulties with this, particularly in the presence of pointers and recursion.

Some `C++` and `Fortran` compilers are parallelizing compilers.

1 There are still bottlenecks with respect to accessing to the shared memory.
2 Fine-grained parallelism is parallelism in which the parallel tasks are just a few instructions long. When the tasks are entire functions, it is considered to be coarse-grained.

Types of Implicit Threading

Greater success is achieved when the programmer can assist the compiler in detecting parallelism.

Implicit threading systems typically require some guidance from the programmer. Usually the programmer must identify tasks that can be executed in parallel. Often these tasks are either functions or structured code blocks, like the body of a loop.

Some well-known implicit threading systems include:

Types of Implicit Threading

Greater success is achieved when the programmer can assist the compiler in detecting parallelism.

Implicit threading systems typically require some guidance from the programmer. Usually the programmer must identify tasks that can be executed in parallel. Often these tasks are either functions or structured code blocks, like the body of a loop.

Some well-known implicit threading systems include:

- **OpenMP** (short for Open Multi-Processing) is an API for programs written in C/C++ and FORTRAN that may be used to explicitly specify multithreaded, shared-memory parallelism. It includes compiler directives, runtime library routines, and environment variables.

Types of Implicit Threading

Greater success is achieved when the programmer can assist the compiler in detecting parallelism.

Implicit threading systems typically require some guidance from the programmer. Usually the programmer must identify tasks that can be executed in parallel. Often these tasks are either functions or structured code blocks, like the body of a loop.

Some well-known implicit threading systems include:

- **OpenMP** (short for Open Multi-Processing) is an API for programs written in `C/C++` and `FORTRAN` that may be used to explicitly specify multithreaded, shared-memory parallelism. It includes compiler directives, runtime library routines, and environment variables.

- **Grand Central Dispatch** (GCD) is a technology developed by Apple for its macOS and iOS operating systems. Like OpenMP, it includes a run-time library, an API, and language extensions that allow developers to identify sections of code to run in parallel.

Types of Implicit Threading

Greater success is achieved when the programmer can assist the compiler in detecting parallelism.

Implicit threading systems typically require some guidance from the programmer. Usually the programmer must identify tasks that can be executed in parallel. Often these tasks are either functions or structured code blocks, like the body of a loop.

Some well-known implicit threading systems include:

- **OpenMP** (short for Open Multi-Processing) is an API for programs written in C/C++ and FORTRAN that may be used to explicitly specify multithreaded, shared-memory parallelism. It includes compiler directives, runtime library routines, and environment variables.
- **Grand Central Dispatch** (GCD) is a technology developed by Apple for its macOS and iOS operating systems. Like OpenMP, it includes a run-time library, an API, and language extensions that allow developers to identify sections of code to run in parallel.
- **Intel Threading Building Blocks** (TBB) is a template library that supports the design of multithreaded, shared-memory programs inC++.

Types of Implicit Threading

Greater success is achieved when the programmer can assist the compiler in detecting parallelism.

Implicit threading systems typically require some guidance from the programmer. Usually the programmer must identify tasks that can be executed in parallel. Often these tasks are either functions or structured code blocks, like the body of a loop.

Some well-known implicit threading systems include:

- **OpenMP** (short for Open Multi-Processing) is an API for programs written in `C/C++` and `FORTRAN` that may be used to explicitly specify multithreaded, shared-memory parallelism. It includes compiler directives, runtime library routines, and environment variables.
- **Grand Central Dispatch** (GCD) is a technology developed by Apple for its macOS and iOS operating systems. Like OpenMP, it includes a run-time library, an API, and language extensions that allow developers to identify sections of code to run in parallel.
- **Intel Threading Building Blocks** (TBB) is a template library that supports the design of multithreaded, shared-memory programs in`C++`.
- **Java Concurrency** refers to the multithreading, concurrency and parallelism available from the `Java Virtul Machine`, which is entirely thread-based. `java.util.concurrent` is the class that provides this concurrency.

Underlying Technology

Consider a multithreaded web server. When the server receives a request, it creates a separate thread to service the request.

Underlying Technology

Consider a multithreaded web server. When the server receives a request, it creates a separate thread to service the request.

When the request has been serviced, the thread is deleted.

Underlying Technology

Consider a multithreaded web server. When the server receives a request, it creates a separate thread to service the request.

When the request has been serviced, the thread is deleted.

There are two problems with this:

- Threads are constantly being created and destroyed.
- There is no bound on how many threads can exist at any time.

Underlying Technology

Consider a multithreaded web server. When the server receives a request, it creates a separate thread to service the request.

When the request has been serviced, the thread is deleted.

There are two problems with this:

- Threads are constantly being created and destroyed.
- There is no bound on how many threads can exist at any time.

The first problem leads to poor CPU utilization and wasted memory resources. The second could lead to system degradation.

Underlying Technology

Consider a multithreaded web server. When the server receives a request, it creates a separate thread to service the request.

When the request has been serviced, the thread is deleted.

There are two problems with this:

- Threads are constantly being created and destroyed.
- There is no bound on how many threads can exist at any time.

The first problem leads to poor CPU utilization and wasted memory resources. The second could lead to system degradation.

To solve this, rather than constantly creating and deleting threads, the implementation can maintain a pool of threads, like a collection of workers sitting in a room waiting to be assigned work to do. When work comes in, the **worker thread** is assigned to it. When it finishes, it goes back to the waiting room.

Thread Pools

Many implementations of implicit threading systems use these **thread pools** to improve their performance[1]. The general idea of thread pools is as follows.

1 Thread pools are used by `Java`, `OpenMP`, and `Grand Central Dispatch`.

Thread Pools

Many implementations of implicit threading systems use these **thread pools** to improve their performance[1]. The general idea of thread pools is as follows.

- A thread pool is initialized with a number of threads, where they wait for work.

1 Thread pools are used by `Java`, `OpenMP`, and `Grand Central Dispatch`.

Thread Pools

Many implementations of implicit threading systems use these **thread pools** to improve their performance[1]. The general idea of thread pools is as follows.

- A thread pool is initialized with a number of threads, where they wait for work.
- When a process needs a new thread to perform a task, it requests one from the thread pool.

1 Thread pools are used by `Java`, `OpenMP`, and `Grand Central Dispatch`.

Thread Pools

Many implementations of implicit threading systems use these **thread pools** to improve their performance[1]. The general idea of thread pools is as follows.

- A thread pool is initialized with a number of threads, where they wait for work.
- When a process needs a new thread to perform a task, it requests one from the thread pool.
- If there is an available thread in the pool, it is awakened and assigned to the process to execute the task.
- If the pool contains no available threads, the task is queued until one becomes free.

1 Thread pools are used by `Java`, `OpenMP`, and `Grand Central Dispatch`.

Thread Pools

Many implementations of implicit threading systems use these **thread pools** to improve their performance[1]. The general idea of thread pools is as follows.

- A thread pool is initialized with a number of threads, where they wait for work.
- When a process needs a new thread to perform a task, it requests one from the thread pool.
- If there is an available thread in the pool, it is awakened and assigned to the process to execute the task.
- If the pool contains no available threads, the task is queued until one becomes free.
- Once a thread is available, it returns to the pool and awaits more work.

1 Thread pools are used by `Java`, `OpenMP`, and `Grand Central Dispatch`.

Thread Pools

Many implementations of implicit threading systems use these **thread pools** to improve their performance[1]. The general idea of thread pools is as follows.

- A thread pool is initialized with a number of threads, where they wait for work.
- When a process needs a new thread to perform a task, it requests one from the thread pool.
- If there is an available thread in the pool, it is awakened and assigned to the process to execute the task.
- If the pool contains no available threads, the task is queued until one becomes free.
- Once a thread is available, it returns to the pool and awaits more work.

Benefits

- Servicing a request with an existing thread is faster than creating a thread.
- A thread pool limits the number of threads that exist at any one point.
- Separating the task to be performed from the mechanics of creating the task means different strategies can be used for running and scheduling the task.

1 Thread pools are used by `Java`, `OpenMP`, and `Grand Central Dispatch`.

The Fork-Join Model

Implicit threading systems often use a paradigm for thread creation and management known as the **fork-join model**.

The Fork-Join Model

Implicit threading systems often use a paradigm for thread creation and management known as the **fork-join model**.

The **fork–join model**, also known as **fork-join parallelism**, is like the explicit thread creation that we saw earlier in the Pthreads library in that threads are created by a parent thread, and their executions reach a join point after which only one thread continues.

Unlike Pthreads, the model does not limit the creation of threads to just one thread at a time.

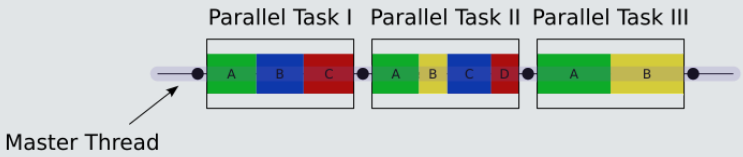
The Fork-Join Model

Implicit threading systems often use a paradigm for thread creation and management known as the **fork-join model**.

The **fork–join model**, also known as **fork-join parallelism**, is like the explicit thread creation that we saw earlier in the Pthreads library in that threads are created by a parent thread, and their executions reach a join point after which only one thread continues.

Unlike Pthreads, the model does not limit the creation of threads to just one thread at a time.

The example we give is based on the following figure from the Wikipedia article about the fork-join model[1].

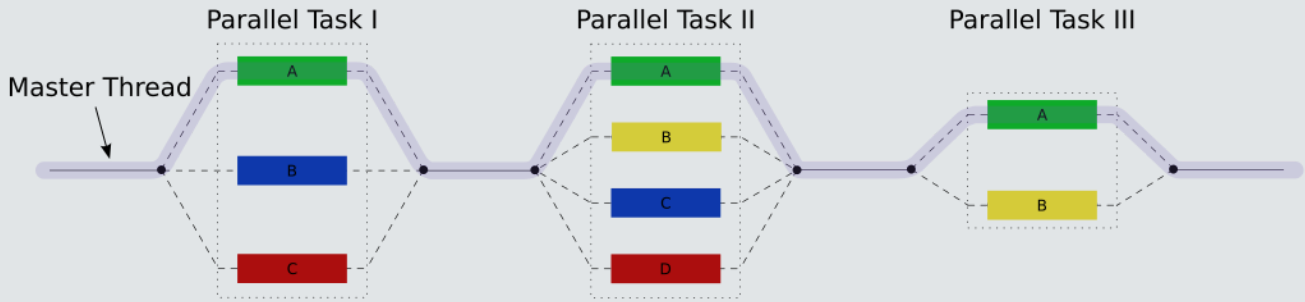


In the figure above, a sequential program has been analyzed, and it is determined that the program has three regions that must be executed sequentially, but that within each region there are varying numbers of parallel tasks, randomly colored and labelled A,B,C, and D.

1 https://en.wikipedia.org/wiki/Fork%E2%80%93join_model#/media/File:Fork_join.svg

Fork-Join Parallelism

A multithreaded program following the fork-join model can be created based on this analysis. The flow through it would be represented by the diagram below.



In the above figure we see that the master thread (always green) forks two new threads and continues to run, so that there are three threads. They join, the master thread executes alone and then forks three threads. The threads join, master runs again and finally just forks one thread and runs with it. After they join, the master thread continues on its own.

This model is used in the implicit threading library employed by Java. It is also the paradigm defined in OpenMP.

1 https://en.wikipedia.org/wiki/Fork%E2%80%93join_model#/media/File:Fork_join.svg

OpenMP Overview

OpenMP was jointly defined by a group of major computer hardware and software vendors. It is an open API for programs written in `C/C++` and `FORTRAN` and is portable, scalable, and provides support on a wide variety of architectures.

OpenMP Overview

OpenMP was jointly defined by a group of major computer hardware and software vendors. It is an open API for programs written in C/C++ and FORTRAN and is portable, scalable, and provides support on a wide variety of architectures.

Because it is an open API, it has implementations on many platforms, many of which are open source, such as all of those provided by GNU under the GNU Public License 3.0.

OpenMP Overview

OpenMP was jointly defined by a group of major computer hardware and software vendors. It is an open API for programs written in C/C++ and FORTRAN and is portable, scalable, and provides support on a wide variety of architectures.

Because it is an open API, it has implementations on many platforms, many of which are open source, such as all of those provided by GNU under the GNU Public License 3.0.

- OpenMP is designed for multiprocessor/core, shared memory machines. The underlying architecture can be either UMA or NUMA.

OpenMP Overview

OpenMP was jointly defined by a group of major computer hardware and software vendors. It is an open API for programs written in `C/C++` and `FORTRAN` and is portable, scalable, and provides support on a wide variety of architectures.

Because it is an open API, it has implementations on many platforms, many of which are open source, such as all of those provided by `GNU` under the GNU Public License 3.0.

- OpenMP is designed for multiprocessor/core, shared memory machines. The underlying architecture can be either UMA or NUMA.
- It has three primary API components:
  - **Compiler directives**, used by programmers to define **parallel regions**
  - **Runtime library routines**, which extend the language with OpenMP functions
  - **Environment variables**, used to control the behavior of OpenMP programs

OpenMP Overview

OpenMP was jointly defined by a group of major computer hardware and software vendors. It is an open API for programs written in C/C++ and FORTRAN and is portable, scalable, and provides support on a wide variety of architectures.

Because it is an open API, it has implementations on many platforms, many of which are open source, such as all of those provided by GNU under the GNU Public License 3.0.

- OpenMP is designed for multiprocessor/core, shared memory machines. The underlying architecture can be either UMA or NUMA.
- It has three primary API components:
  - **Compiler directives**, used by programmers to define **parallel regions**
  - **Runtime library routines**, which extend the language with OpenMP functions
  - **Environment variables**, used to control the behavior of OpenMP programs
- The parallelism in OpenMP programs is exclusively through the use of threads. Most implementations (such as GNU) use thread pools to manage the threads.
- OpenMP uses the **fork-join** model of parallel execution. An OpenMP program begins as a single thread called the **master thread**. The master thread executes sequentially until the first parallel region is encountered.

OpenMP Key Features

- OpenMP defines **parallel regions** as blocks of code that may run in parallel.
- Programmers insert compiler directives into their code at parallel regions; these directives instruct the OpenMP run-time library to execute the region in parallel.

OpenMP Key Features

- OpenMP defines **parallel regions** as blocks of code that may run in parallel.
- Programmers insert compiler directives into their code at parallel regions; these directives instruct the OpenMP run-time library to execute the region in parallel.
- Parallel regions are specified using compiler directives, known as **pragmas**. For example, a simple directive is

```
#pragma omp parallel
```

  which specifies that the following statement is to be executed by some number of threads in parallel.

- Because OpenMP is a shared memory programming model, most data within a parallel region is shared by default. It can also be made explicit:

```
// declare a and i before
#pragma omp parallel shared(a) private(i)
```

  is a `C/C++` pragma that specifies the start of a parallel region with a variable `a` shared by all threads and a thread-private variable named `i` of which each thread has a private copy.

OpenMP Run-time Routines

OpenMP has a rich set of compiler directives for specifying parallel for-loops, critical sections of code, nested parallel regions, and much more.

The run-time library has routines for such things as

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID)
- Querying if execution is within a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying clock time

OpenMP Example Program

Following is a simple OpenMP program, demonstrating two parallel regions with sequential code in between. OpenMP creates a thread for each core by default.

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[])
{
    #pragma omp parallel  // executed by default number of threads
    {
        printf("I am a distinct thread.\n");
    }

    // The following code is executed only by the master thread.
    omp_set_num_threads(8);        // set number of threads to 8
    int N = omp_get_max_threads(); // get max num threads
    printf("There are %d threads\n", N);

    #pragma omp parallel   // Executed by 8 threads
    {
        int ID = omp_get_thread_num();  // get thread id of executing thread
        printf("hello world from thread %d\n ", ID);
    }
    return 0;
}
```

Grand Central Dispatch

Grand Central Dispatch (**GCD**) is a technology developed by Apple Inc. for use on symmetric multiprocessors/cores. In 2009, the underlying library, `libdispatch`, was released as open source under the `Apache 2.0` license.

- It provides support for programs written in various languages, including `C`, `C++`, `Objective-C`, and `Swift`.

Grand Central Dispatch

Grand Central Dispatch (**GCD**) is a technology developed by Apple Inc. for use on symmetric multiprocessors/cores. In 2009, the underlying library, `libdispatch`, was released as open source under the `Apache 2.0` license.

- It provides support for programs written in various languages, including `C`, `C++`, `Objective-C`, and `Swift`.
- GCD is an implementation of **task parallelism** (see definition) based on **thread pools**. Thread pool management is implicit; the library manages the threads without the programmer's involvement. The programmer specifies the tasks that can be run in parallel.

Grand Central Dispatch

Grand Central Dispatch (**GCD**) is a technology developed by Apple Inc. for use on symmetric multiprocessors/cores. In 2009, the underlying library, `libdispatch`, was released as open source under the `Apache 2.0` license.

- It provides support for programs written in various languages, including `C`, `C++`, `Objective-C`, and `Swift`.
- GCD is an implementation of **task parallelism** (see definition) based on **thread pools**. Thread pool management is implicit; the library manages the threads without the programmer's involvement. The programmer specifies the tasks that can be run in parallel.
- A task can be expressed either as a function or as a **block**. A **block** is an extension to the syntax of C, C++, and Objective-C that encapsulates code and data into a single object. A block is specified by a caret `^` inserted in front of a pair of braces `{ }`:

```
^{ printf("This is a block"); }
```

- GCD queues the designated tasks for execution and schedules them onto an available thread to run on any available processor.

Grand Central Dispatch

Grand Central Dispatch (**GCD**) is a technology developed by Apple Inc. for use on symmetric multiprocessors/cores. In 2009, the underlying library, `libdispatch`, was released as open source under the `Apache 2.0` license.

- It provides support for programs written in various languages, including `C`, `C++`, `Objective-C`, and `Swift`.
- GCD is an implementation of **task parallelism** (see definition) based on **thread pools**. Thread pool management is implicit; the library manages the threads without the programmer's involvement. The programmer specifies the tasks that can be run in parallel.
- A task can be expressed either as a function or as a **block**. A **block** is an extension to the syntax of C, C++, and Objective-C that encapsulates code and data into a single object. A block is specified by a caret `^` inserted in front of a pair of braces `{ }`:

```
^{ printf("This is a block"); }
```

- GCD queues the designated tasks for execution and schedules them onto an available thread to run on any available processor.
- The underlying support for `libdispatch` is the POSIX threads library, but most of the support comes from non-POSIX compliant Apple extensions.

Example of a GCD Program

This is an example of a Grand Central Dispatch program. It creates a block that just prints a message with the id of the thread that it is assigned to. The block is dispatched onto a concurrent queue.

The code is designed to show a bit of the internals of GCD. It is based on code written by Jonathan Levin (http://newosxbook.com/articles/GCD.html).

```
#include <stdio.h>
#include <dispatch/dispatch.h>
#include <pthread.h>

int main (int argc, char *argv[])
{
    void (^myblock) (void) =
         ^{ printf("My pthread id is %d\n", (int) pthread_self()); };
    dispatch_queue_t myqueue =
       dispatch_queue_create("example of queue",  DISPATCH_QUEUE_CONCURRENT);

    dispatch_group_t dgroup = dispatch_group_create();
    dispatch_group_async(dgroup, myqueue, myblock);

    int rc= dispatch_group_wait(dgroup, DISPATCH_TIME_FOREVER);
    return rc;
}
```

Threading Issues

We explore some of the issues that arise in the design and implementation of threading systems, whether they are in user level libraries or kernels.

Issues with `fork()`

The `fork()` system call was designed for processes. Threads were not part of UNIX when it was invented. When a process calls `fork()`, a new one is created that is nearly identical to the original.

When a thread that is part of a process issues a `fork()` system call, a new process is created. **Should just the calling thread be duplicated in the child process or should all threads be duplicated?**

Issues with `fork()`

The `fork()` system call was designed for processes. Threads were not part of UNIX when it was invented. When a process calls `fork()`, a new one is created that is nearly identical to the original.

When a thread that is part of a process issues a `fork()` system call, a new process is created. **Should just the calling thread be duplicated in the child process or should all threads be duplicated?**

- Most implementations duplicate just the calling thread. Duplicating all threads is more complex and costly.
- But some systems provide both possibilities with `fork()`. For example, **Oracle Solaris**'s `fork()` duplicates all threads but its `fork1()` duplicates just the calling thread.

Issues with the `exec()` Family

The various system calls in the `exec()` family for processes replace the process's address space entirely, giving it a new program to execute. For example, the call

```
execve("/bin/echo", argv, envp);
```

would cause the calling process to execute the `/bin/echo` program with the arguments passed in `argv` using the environment variables pointed to by `envp`.

With multithreaded programs, the question is, when a thread makes this call, **should the entire process be replaced, including all threads?** What other behavior is possible?

Issues with the `exec()` Family

The various system calls in the `exec()` family for processes replace the process's address space entirely, giving it a new program to execute. For example, the call

```
execve("/bin/echo", argv, envp);
```

would cause the calling process to execute the `/bin/echo` program with the arguments passed in `argv` using the environment variables pointed to by `envp`.

With multithreaded programs, the question is, when a thread makes this call, **should the entire process be replaced, including all threads?** What other behavior is possible?

The `POSIX` requirement is that an `execve()` system call from any thread in a multithreaded process must cause all other threads in that process to terminate and the calling thread to complete the `execve()`.

Modern Linux has this implementation.

Issues with the `exec()` Family

The various system calls in the `exec()` family for processes replace the process's address space entirely, giving it a new program to execute. For example, the call

```
execve("/bin/echo", argv, envp);
```

would cause the calling process to execute the `/bin/echo` program with the arguments passed in `argv` using the environment variables pointed to by `envp`.

With multithreaded programs, the question is, when a thread makes this call, **should the entire process be replaced, including all threads?** What other behavior is possible?

The `POSIX` requirement is that an `execve()` system call from any thread in a multithreaded process must cause all other threads in that process to terminate and the calling thread to complete the `execve()`.

Modern Linux has this implementation.

Earlier versions of Linux detached the calling thread from the original process and ran the new program in it, letting all other threads continue to run in the old address space.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are busy doing something and the phone rings.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are busy doing something and the phone rings.

You stop and answer it. The phone ringing is a **signal** sent to you. Your answering the call means that the signal was **delivered** to you. The act of answering the call is called **handling the signal**.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are busy doing something and the phone rings.

You stop and answer it. The phone ringing is a **signal** sent to you. Your answering the call means that the signal was **delivered** to you. The act of answering the call is called **handling the signal**.

The phone ring is an **asynchronous signal** because it can happen any time in an unpredictable way.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are busy doing something and the phone rings.

You stop and answer it. The phone ringing is a **signal** sent to you. Your answering the call means that the signal was **delivered** to you. The act of answering the call is called **handling the signal**.

The phone ring is an **asynchronous signal** because it can happen any time in an unpredictable way.

The fact that you answer the phone is part of your **disposition** towards the phone ringing signal.

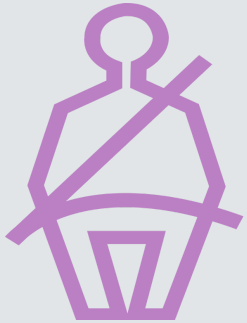
Image from Wikimedia Commons by Videoplasty.com, CC-BY-SA 4.0.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

Now you get in the car and start to drive. An annoying horn sounds to remind you that your seat belt is not fastened.

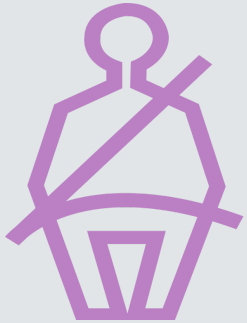


Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

Now you get in the car and start to drive. An annoying horn sounds to remind you that your seat belt is not fastened.



You fasten the seat belt. The horn sounding is a **signal** as well, and your fastening the seat belt means that the horn signal was **delivered** to you, and that you **handled the signal** as well.

Unlike the phone ringing, the horn sounding is a **synchronous** signal. It happened because of your failure to buckle up. As a process, you skipped a step you were supposed to do, the operating system (the car) intervened and reminded you of it.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are now driving the car. You remember that you are not supposed to answer the phone while driving.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are now driving the car. You remember that you are not supposed to answer the phone while driving.

Signals

Signals are a complex topic, even just for single-threaded programs. We will explain signals in an intuitive way to start.

You should think of yourself as a process. Assume that when the phone rings, your default behavior is to answer it. Assume too that you are a responsible driver and that when you are driving a car you always keep your seat belt buckled. Lastly, you will not answer the phone when you are driving.

You are now driving the car. You remember that you are not supposed to answer the phone while driving.

You silence the phone. You have just temporarily **blocked** the phone-ringing signal from being **delivered** to you. While the phone is silenced, people can still call, but you do not answer the phone. When you finish driving, you can restore the phone's volume, unblocking the signal.

Blocking a signal does not prevent someone from calling; it just prevents the signal from being delivered to you.

Signal Handling

A **signal** is an empty message sent to a process because an event occurred. It has type and nothing more.

Signal Handling

A **signal** is an empty message sent to a process because an event occurred. It has type and nothing more.

A signal may be received either **synchronously** or **asynchronously**. Traps and other exceptions are examples of signals received synchronously. Timer interrupts (`SIGALRM`), keyboard interrupts Control-C (`SIGINT`), and terminal disconnections (`SIGHUP`) are examples of asynchronously received signals.

Signal Handling

A **signal** is an empty message sent to a process because an event occurred. It has type and nothing more.

A signal may be received either **synchronously** or **asynchronously**. Traps and other exceptions are examples of signals received synchronously. Timer interrupts (`SIGALRM`), keyboard interrupts Control-C (`SIGINT`), and terminal disconnections (`SIGHUP`) are examples of asynchronously received signals.

The sequence of actions with respect to any type of signal is:

1. A signal is **generated** by some event.
2. The signal is **sent** to a process by the kernel.
3. The signal is **pending** until the next step.
4. The signal is **delivered** to the process when the process takes some action with respect to it, which is either performing the **default action**, **ignoring** it, or catching the signal with a **signal handler**. The **disposition** of the signal is how the process behaves when the signal is delivered.

Signal Handling

A **signal** is an empty message sent to a process because an event occurred. It has type and nothing more.

A signal may be received either **synchronously** or **asynchronously**. Traps and other exceptions are examples of signals received synchronously. Timer interrupts (`SIGALRM`), keyboard interrupts Control-C (`SIGINT`), and terminal disconnections (`SIGHUP`) are examples of asynchronously received signals.

The sequence of actions with respect to any type of signal is:

1. A signal is **generated** by some event.
2. The signal is **sent** to a process by the kernel.
3. The signal is **pending** until the next step.
4. The signal is **delivered** to the process when the process takes some action with respect to it, which is either performing the **default action**, **ignoring** it, or catching the signal with a **signal handler**. The **disposition** of the signal is how the process behaves when the signal is delivered.

A **signal handler** is a function that is run when a signal is delivered. Signal handlers are registered with the kernel. If there is no user-defined signal handler, a default action is taken.

Signal Handling

A **signal** is an empty message sent to a process because an event occurred. It has type and nothing more.

A signal may be received either **synchronously** or **asynchronously**. Traps and other exceptions are examples of signals received synchronously. Timer interrupts (`SIGALRM`), keyboard interrupts Control-C (`SIGINT`), and terminal disconnections (`SIGHUP`) are examples of asynchronously received signals.

The sequence of actions with respect to any type of signal is:

1. A signal is **generated** by some event.
2. The signal is **sent** to a process by the kernel.
3. The signal is **pending** until the next step.
4. The signal is **delivered** to the process when the process takes some action with respect to it, which is either performing the **default action**, **ignoring** it, or catching the signal with a **signal handler**. The **disposition** of the signal is how the process behaves when the signal is delivered.

A **signal handler** is a function that is run when a signal is delivered. Signal handlers are registered with the kernel. If there is no user-defined signal handler, a default action is taken.

Each signal type has a **pending flag** indicating whether or not it is pending, and a **blocked flag** indicating whether or not it is blocked.

Signals and Threads: Issues

The question of how to handle signals in a multithreaded process has been debated since the early days of POSIX threading, and there have been extensive changes in the POSIX standard over the past few decades.

There are many questions. We cannot address them all.

Signals and Threads: Issues

The question of how to handle signals in a multithreaded process has been debated since the early days of POSIX threading, and there have been extensive changes in the POSIX standard over the past few decades.

There are many questions. We cannot address them all.

- When a signal is sent to a process, where is it delivered?
    - to a single specified thread?
    - to any single thread?
    - to all threads?
    - to some threads?
- Does every thread have its own set of flags?

Signals and Threads: Issues

The question of how to handle signals in a multithreaded process has been debated since the early days of POSIX threading, and there have been extensive changes in the POSIX standard over the past few decades.

There are many questions. We cannot address them all.

- When a signal is sent to a process, where is it delivered?
    - to a single specified thread?
    - to any single thread?
    - to all threads?
    - to some threads?
- Does every thread have its own set of flags?
- Can every thread have its own signal handlers?

Signals and Threads: Issues

The question of how to handle signals in a multithreaded process has been debated since the early days of POSIX threading, and there have been extensive changes in the POSIX standard over the past few decades.

There are many questions. We cannot address them all.

- When a signal is sent to a process, where is it delivered?
    - to a single specified thread?
    - to any single thread?
    - to all threads?
    - to some threads?
- Does every thread have its own set of flags?
- Can every thread have its own signal handlers?
- Can threads send signals to specific threads or to all threads in a process?

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

- A signal may be generated for a process as a whole (meaning all threads in it), or for a specific thread.

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

- A signal may be generated for a process as a whole (meaning all threads in it), or for a specific thread.
- Signals that are generated because of execution of a machine-language instruction (traps and exceptions) are sent to the thread that executed that instruction.

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

- A signal may be generated for a process as a whole (meaning all threads in it), or for a specific thread.
- Signals that are generated because of execution of a machine-language instruction (traps and exceptions) are sent to the thread that executed that instruction.
- A process-directed signal may be delivered to any one of the threads that does not currently have that signal blocked. If more than one of the threads has the signal unblocked, it is delivered to any one of them.

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

- A signal may be generated for a process as a whole (meaning all threads in it), or for a specific thread.
- Signals that are generated because of execution of a machine-language instruction (traps and exceptions) are sent to the thread that executed that instruction.
- A process-directed signal may be delivered to any one of the threads that does not currently have that signal blocked. If more than one of the threads has the signal unblocked, it is delivered to any one of them.
- Every thread has its own pending and blocked flags.

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

- A signal may be generated for a process as a whole (meaning all threads in it), or for a specific thread.
- Signals that are generated because of execution of a machine-language instruction (traps and exceptions) are sent to the thread that executed that instruction.
- A process-directed signal may be delivered to any one of the threads that does not currently have that signal blocked. If more than one of the threads has the signal unblocked, it is delivered to any one of them.
- Every thread has its own pending and blocked flags.
- The dispositions of all signals are process-wide, meaning that all threads share the same signal handler, or all ignore the signal, and so on.

Signals and Threads: Solutions

Different systems have solved these problems in different ways.

In Linux and POSIX, for example,

- A signal may be generated for a process as a whole (meaning all threads in it), or for a specific thread.
- Signals that are generated because of execution of a machine-language instruction (traps and exceptions) are sent to the thread that executed that instruction.
- A process-directed signal may be delivered to any one of the threads that does not currently have that signal blocked. If more than one of the threads has the signal unblocked, it is delivered to any one of them.
- Every thread has its own pending and blocked flags.
- The dispositions of all signals are process-wide, meaning that all threads share the same signal handler, or all ignore the signal, and so on.
- In Linux, there are functions (system calls and library routines) that allow a process or thread to send signals to one or more other processes (`kill`) or to specific threads (`tgkill`). A thread can send a signal to itself (`raise`) or to a thread in its same process (`pthread_kill`).

Thread Cancellation

Sometimes a thread might need to be terminated by another thread before it has finished its work.

Thread cancellation is the act of terminating a thread that has not yet terminated itself. With thread cancellation, one thread can **try to terminate** another.

The fact that a thread tries to terminate another thread does not mean it will succeed.

Thread Cancellation

Sometimes a thread might need to be terminated by another thread before it has finished its work.

Thread cancellation is the act of terminating a thread that has not yet terminated itself. With thread cancellation, one thread can **try to terminate** another.

The fact that a thread tries to terminate another thread does not mean it will succeed.

There are many reasons to allow thread cancellation. For example, suppose threads are searching in parallel for a key in a large database. Each is searching through a different portion of the data. Only one thread can find the key. When it does, the others should not continue to run, because they will accomplish nothing.

We need a way for the thread that finds the key to terminate the others.

Thread Cancellation

Sometimes a thread might need to be terminated by another thread before it has finished its work.

Thread cancellation is the act of terminating a thread that has not yet terminated itself. With thread cancellation, one thread can **try to terminate** another.

The fact that a thread tries to terminate another thread does not mean it will succeed.

There are many reasons to allow thread cancellation. For example, suppose threads are searching in parallel for a key in a large database. Each is searching through a different portion of the data. Only one thread can find the key. When it does, the others should not continue to run, because they will accomplish nothing.

We need a way for the thread that finds the key to terminate the others.

Terminology: A thread is **canceled** if it is terminated. The thread to be canceled is called the **target thread**.

Thread Cancelability

Most thread systems provide some means for one thread to request cancellation of another. One issue is whether cancellation should be immediate, or should be delayed. Sometimes a thread is in the middle of a computation that should not be interrupted or else shared data will be corrupted.

Thread Cancelability

Most thread systems provide some means for one thread to request cancellation of another. One issue is whether cancellation should be immediate, or should be delayed. Sometimes a thread is in the middle of a computation that should not be interrupted or else shared data will be corrupted.

POSIX defines two types of cancellation:

- **Asynchronous cancellation**: A thread can cancel the target thread immediately.
- **Deferred cancellation**: A thread attempts to cancel the target thread, but the target thread is not canceled immediately. Instead, it terminates when it reached a point in its execution when it is safe to do so.

Thread Cancelability Control

In POSIX, a thread decides its own fate.

- It can set its **cancelability type** to be either asynchronous or deferred. For example, this code is used by a thread to give it deferred cancelability:

```
retval = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

Thread Cancelability Control

In POSIX, a thread decides its own fate.

- It can set its **cancelability type** to be either asynchronous or deferred. For example, this code is used by a thread to give it deferred cancelability:

```
retval = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

- It can also enable or disable cancelability. For example, this disables cancelability for the thread:

```
retval = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

Thread Cancelability Control

In POSIX, a thread decides its own fate.

- It can set its **cancelability type** to be either asynchronous or deferred. For example, this code is used by a thread to give it deferred cancelability:

```
retval = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

- It can also enable or disable cancelability. For example, this disables cancelability for the thread:

```
retval = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

- Together these give it control over when, if at all, it can be terminated by another thread.

Thread Cancelability Control

In POSIX, a thread decides its own fate.

- It can set its **cancelability type** to be either asynchronous or deferred. For example, this code is used by a thread to give it deferred cancelability:

```
retval = pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
```

- It can also enable or disable cancelability. For example, this disables cancelability for the thread:

```
retval = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
```

- Together these give it control over when, if at all, it can be terminated by another thread.

When threads have deferred cancelability, they terminate when they reach safe points in their code, called **cancellation points**. Cancellation points are calls to selected functions. When the thread calls one of these functions, it terminates.

Thread Cancellation Points

Most of the blocking system calls in the POSIX and standard C library are cancellation points. If you enter the command

```
man pthreads
```

on a Linux system, you can find the complete list of cancellation points on your system.

Thread Cancellation Points

Most of the blocking system calls in the POSIX and standard C library are cancellation points. If you enter the command

```
man pthreads
```

on a Linux system, you can find the complete list of cancellation points on your system.

The Pthreads API has several functions related to thread cancellation. See the program `thread_cancel_demo.c` on the server for a complete, documented example.

References

1. Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities." In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483-485, New York, NY, USA, 1967. ACM.

2. Dave McCracken. "POSIX Threads and the Linux Kernel." In *Proceedings of the Ottawa Linux Symposium*, June 26th–29th, 2002, Ottawa, Ontario, Canada. pp. 330-337.

3. Abraham Silberschatz, Greg Gagne, Peter B. Galvin. *Operating System Concepts*, 10th Edition. Wiley Global Education, 2018.

4. The GNU Operating System. https://www.gnu.org/

5. Stewart Weiss, *UNIX System Programming Lecture Notes*, http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes.php.