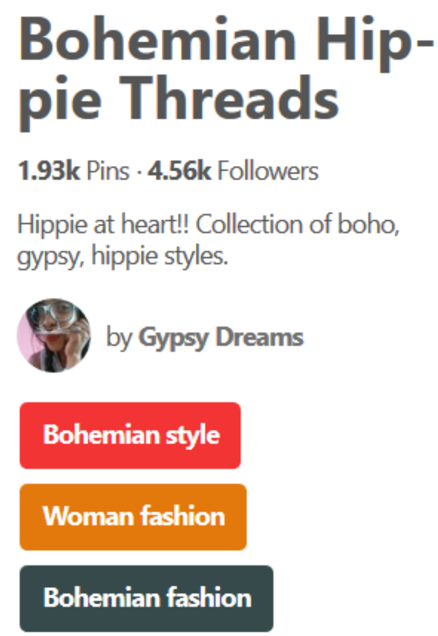
Threads & Concurrency

CS201 Lecture 4

Jason Hibbeler
University of Vermont
Spring 2019



Thread

Basic unit of CPU utilization

- an ID
- a program counter
- register values
- a stack

It's like the essential essence of a process

- remember that the process has data, text (code), system resources (such as files)
- all threads in a process share these resources

Thread

General statement:

If a process has more than one thread, and these threads can execute simultaneously, then the process should be quicker*

*we'll discuss what this actually means

Threads in Action

Think about a word-processing program

- at the same time you're typing text, the program is doing spellchecking for you
- it might also do an automatic save while you're typing text

Or an even better example: an IDE

- with all of the suggestions, checking, and cross-referencing that an IDE does under the covers while you are typing in program text

Web Server

Another example: a web server

Structure of web server

- listen for a request
- when a request appears, run code to handle that request

If the server has a single thread of execution, then it won't be able to handle new requests while it's tending to an existing request

- why not have the server just fork off a new process to handle each new request? We will answer this next.

Benefits of Multithreading

1. Responsiveness
2. Resource sharing
3. Economy
4. Scalability

Responsiveness

Without threading, then if an application blocks or is performing a lengthy computation, then the whole program blocks

- UI must always be responsive
- put the computation in a separate thread
- put the calls that have unknown response time in a separate thread (think about any application that needs to get input from the web)

Resource Sharing

Resource sharing among processes is somewhat complex for application developers (shared memory, message passing)

- threads share memory and system resources
- this makes programming multithreaded applications somewhat less awkward that multi-process applications (at least from the standpoint of sharing resources)

Economy

Since threads inherit the system resources of the parent process, it's faster and cheaper to create threads than it is to create new processes

- and the context switch between threads is faster as well

Scalability

In a multicore machine, different threads of a process can run on different cores

- whereas a single-threaded process can run only on a single core

Concurrency vs. Parallelism

Think about how a single-core system processes tasks (threads) for a user. Here is an idealized diagram showing the processing of six threads in the system:

T_1	T_2	T_3	T_4	T_5	T_6	T_1	T_2	T_3	T_4	T_5	T_6

time $\longrightarrow$

What's the throughput per unit time?

- it's 1 / numthreads

The threads are being executed *concurrently*

- all of them are making progress

Concurrency vs. Parallelism

Now suppose we have two CPUs (cores):

core #1	T_1	T_3	T_5	T_1	T_3	T_5	T_1	T_3	T_5	T_1	T_3	T_5
core #2	T_2	T_4	T_6	T_2	T_4	T_6	T_2	T_4	T_6	T_2	T_4	T_6

time →

What's the throughput per unit time? It's 2 / numthreads

- twice as fast!

Two threads are being executed *in parallel*

- if we can swing this, it's clearly better

In what circumstances can we actually do this?

Multithreading: Challenges

Creating multithreaded applications is challenging.

A general observation:

It is much more difficult to rewrite a single-threaded application to be multithreaded than it is to design the application to be multithreaded from the beginning.

Multithreading: Challenges

Identifying tasks:

- in the processing of the application, we want to have separate tasks that can execute independently of each other.

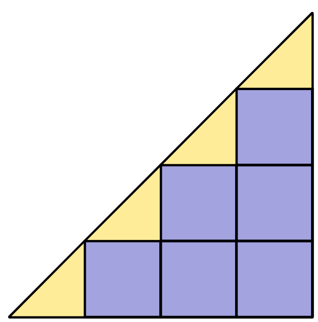
Is this easy to accomplish?

- it depends on the application

Multithreading: Challenges

Balance: even if we can decompose the application into separate, independent tasks, we want to insure that the tasks all perform approximately the same amount of work.

More specifically: the work they each perform should take approximately the same amount of time.

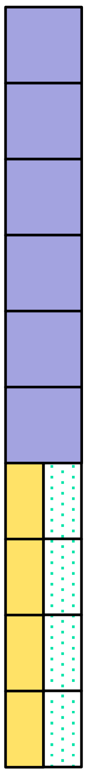


- example: suppose we are doing some processing on a triangular matrix
- here, the yellow tasks have only half as much work as blue tasks

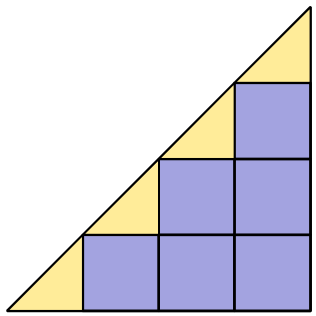
Multithreading: Challenges

Ten threads start together, each on its own core

- but four of the threads have only half as much work



- if each CPU is mapped to a core, then four of the cores are idle for half of the time
- this gives us a total utilization of 80%



Multithreading: Challenges

Data splitting:

- in order to enable separate tasks to run independently, we need to be able to partition the data so that the tasks can access* the data that they need in an independent fashion

*key point here: if the tasks merely read the data, then we don’t have to do anything special; but if tasks modify the data then we have to arrange things so that the data can be independent

Problem Decomposition

Example: adding two vectors

- worker thread #1 can add A[0..999] + B[0...999]
- worker thread #2 can add A[1000..1999] + B[1000...1999]
- etc.

Each thread can do its work independently of the other threads

Problem Decomposition

Another example: finding the maximum of the values in an array

- worker thread #1 can find the max in A[0..999]
- worker thread #2 can find the max in A[1000..1999]
- etc.

But what has to happen next?

Problem Decomposition

Another example: finding the maximum of the values in an array

- worker thread #1 can find the max in A[0..999]
- worker thread #2 can find the max in A[1000..1999]
- etc.

But what has to happen next?

- another thread has to look at the max value that each thread has found and take the max of that
- so this "master thread" can do its work only after the "worker threads" have finished

Problem Decomposition

Another example: sorting an array

- worker thread #1 can sort the values in A[0..999]
- worker thread #2 can sort the values in A[1000..1999]
- etc.

What has to happen next?

Problem Decomposition

Another example: sorting an array

- worker thread #1 can sort the values in A[0..999]
- worker thread #2 can sort the values in A[1000..1999]
- etc.

What has to happen next?

- the master thread has to merge all of the sorted subarrays together

Multithreading: Challenges

Thread-local storage

- Threads inherit the data space of the process that created them
- But it's also important for threads to have their own private storage (we'll see examples)
- And the casual modification of a non-thread-local variable by a thread can cause unpredictable behavior in other threads that are using that variable

Multithreading: Challenges

Data dependency:

- if task T_k depends on data that is produced by task T_j, then T_k can execute only after T_j is complete

Furthermore:

- we must impose a synchronization regime so that T_k knows when T_j has produced the data that T_k needs.

Multithreading: Challenges

Testing and debugging:

- instead of just a single thread of execution to trace, we will have several threads
- and the order in which they execute could be indeterminate (or, more importantly, unpredictable)

Summary:

- does it seem like it would be difficult to design and develop multithreaded application programs?
- yes!

Two Basic Models of Parallelism

Data parallelism

- split up the data across tasks; create a thread for each task
- each task is identical but operates on its own subset of the data.

Simple example is the addition of two vectors

- task #1 sums A[0:999] + B[0:999]
- task #2 sums A[1000:1999] + B[1000:1999]
- etc.
- so each thread is working safely in its own region of A and B

Two Basic Models of Parallelism

Task parallelism

- the two task are different and perform different operations.
- simple example: one task computes the mean of the values in an array
- and a second task computes the max of the values

Models of Multithreading

user thread: the user application manages the control of the thread

- here, your code actually creates the threads

kernel thread: the kernel manages the control of the thread

- the OS itself creates the threads

Models of Multithreading

The OS schedules kernel threads

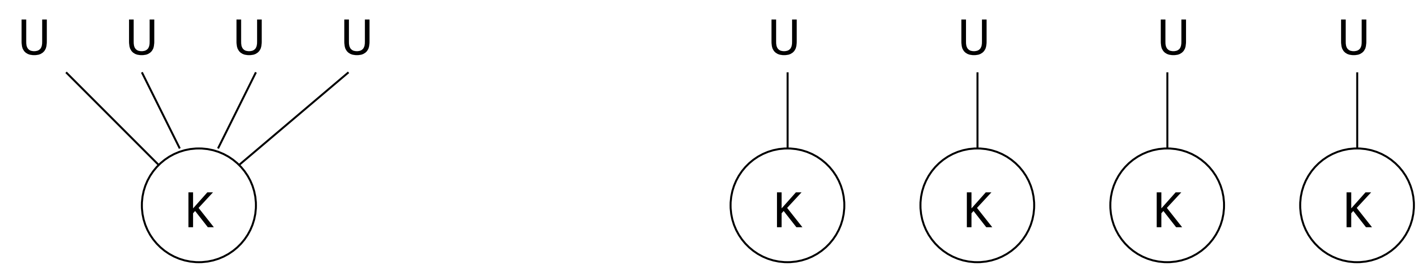
- in other words, it’s the kernel threads that get put onto a CPU
- the threading library (the part of the OS that actually provides the threading interface for application developers) decides how to map user threads to kernel threads

Models of Multithreading

many-to-one vs. one-to-one

- the OS is configured to do one of these

K is kernel thread
U is user thread



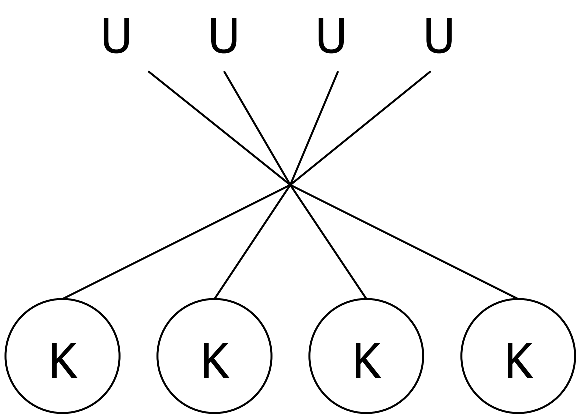
Consequence of this mapping:

- many-to-one cannot actually achieve parallelism

Models of Multithreading

Most effective model is many-to-many

- so if one user thread blocks, the OS can schedule a different kernel thread for execution



Thread Libraries

A thread library provides the APIs for the application developer to create and manage threads

- we'll look at a couple of python examples (on gitlab)
- we'll also look at a Unix pthread example: given an integer *N*, sum all the integers from 1 to *N*

Unix Example: pthreads

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

#define NUMCHILDREN 2

void *runner(void *param);

typedef struct {
  int lowVal;
  int highVal;
  int sum;
} SumStruct;

int main(int argc, char *argv[]) {
  SumStruct data[NUMCHILDREN];   // holds data we want to
                                 // give to child thread
  pthread_t tid[NUMCHILDREN];    // thread identifier
  pthread_attr_t attr; // thread attributes
  int bigSum;
  int i;

  data[0].lowVal = 1;
  data[0].highVal = 50;
  data[1].lowVal = 51;
  data[1].highVal = 100;

  // get default thread attributes
  pthread_attr_init(&attr);

  for (i=0; i<NUMCHILDREN; ++i) {
    // create child thread
    pthread_create(&tid[i], &attr, runner,
                   &data[i]);
  }

  for (i=0; i<NUMCHILDREN; ++i) {
    // wait for the child threads to terminate
    pthread_join(tid[i], NULL);
  }

  bigSum = 0;
  for (i=0; i<NUMCHILDREN; ++i) {
    bigSum = bigSum + data[i].sum;
  }

  printf("sum is %d\n", bigsum);
  return(0);
}
```

this code is in gitlab, in pthreads-example.c, under Examples/

Unix Example: pthreads

```
void *runner(void *param) {
  SumStruct *data;
  int i, sum;

  sum = 0;
  data = (SumStruct *) param;

  printf("(R) I am runner; will sum integers from %d to %d\n",
         data->lowVal, data->highVal);

  for (i=data->lowVal; i<=data->highVal; ++i)
    sum = sum + i;
  data->sum = sum;

  printf("(R) sum is %d\n", data->sum);

  pthread_exit(0);
}
```

this code is in gitlab, in pthreads-example.c, under Examples/

Threads in Java

How is multithreading actually supported in Java?

Java code executes in a self-contained environment called the Java Virtual Machine.

So, this means that the JVM is managing the threads for us

See Java example in textbook (section 4.4.3).

Threads in Python

Python has a structure called the "global interpreter lock" (GIL)

- this prevents more than one thread from executing at a single time
- so naïve threading in Python won't provide any speed-up
- if you read a discussion of threading in Python, you'll see that knowledgeable people say "don't do threading in Python"
- but it's a good way to prototype an algorithm or a program
- and there is a different way to achieve parallelism in Python

Skip

We will skip Section 4.5 and 4.6 and 4.7.