

THREADS

Professor Ken Birman
CS4414 Lecture 13

THREADS!



We've heard about them but haven't worked with them.
... well, that's about to change!

This topic starts a whole new unit – our “systems programming tour of Linux and C++” is finished.

IDEA MAP FOR TODAY

Reminder: Thread Concept

Schedulers and multilevel feedback queues with round-robin scheduling

Lightweight vs. Heavyweight threads

Memory access speeds in a NUMA setting with threads

Today starts a whole unit entirely focused on concurrency inside a C++ program and how to safely manage it.

REMINDER: NUMA MACHINES

We saw that modern computers are NUMA machines, and why

... and how the Linux operating system, by allowing multiple user processes to run side by side, “exposes” this parallelism. These “single-threaded” processes benefit from NUMA!

NUMA MACHINES

... virtual machines go even further, and create the illusion of multiple separate computers, all on the single NUMA chip.

Virtualization is a fantastic fit with NUMA hardware.

Owning a NUMA machine is like owning a cluster of computers.

REMINDER: WHAT'S A THREAD?

But we can also write a single program that, at runtime, uses parallelism internally, via what we call a *thread*.

Use of threads requires a deep understanding of performance consequences, overheads, and how to program correctly with concurrency.

Many programs would slow down or crash if you just threw threads in.

EXAMPLE: THE LINUX KERNEL!

Linux itself is a multithreaded program.

When Linux manages a NUMA machine, it could easily be handling requests for many processes concurrently.

Each of those requests would generally have an associated thread to carry out the work on behalf of that process.

EXAMPLE TWO: WORD COUNT

Recall our word count from Lectures 1-3. It had:

- One “main” thread to process the arguments, then launch threads
- One thread just to open files
- N threads to count words, in parallel, on distinct subsets of the files and implement parallel count-tree merge

Main thread resumed control at the end, sorted output, printed it.

FEATURES OF THREADS

Very easy to create: in effect, instead of calling a method, we “fork it off” in a thread of its own, and the call occurs there.

Like this:

```
auto fileopener = std::thread(fopener, nfiles, (char**)file_names);
std::thread my_threads[MAXTHREADS];
for(int n = 0; n < nthreads; n++)
{
    my_threads[n] = std::thread(wcounter, n);
}
for(int n = 0; n < nthreads; n++)
{
    my_threads[n].join();
}
fileopener.join();
```

FEATURES OF THREADS

Very “fork” effect, instead of calling a method, we “fork” a thread of its own, and the call occurs there.

In fast-wc, wcounter was a method that takes an integer id as its single argument

Like this:

```
auto fileopener = std::thread(fopener, nfiles, (char**)file_names);
std::thread my_threads[MAXTHREADS];
for(int n = 0; n < nthreads; n++)
{
    my_threads[n] = std::thread(wcounter, n);
}
for(int n = 0; n < nthreads; n++)
{
    my_threads[n].join();
}
fileopener.join();
```

LINUX CREATES PROCESSES USING A SYSTEM CALL NAMED “FORK”

Any process can “clone itself” by calling `pid = fork()`.

The parent process will receive the pid of its new child.

The child process is identical to the parent (even shares the same open files, like `stdin`, `stdout`, `stderr`), but gets pid 0. Typically, the child immediately “sets up” a runtime environment for itself.

WHY “FORK”

Because of poetry!



*“Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood...”*

-- Robert Frost

Recall that in Linux, every process has a parent process, and `/etc/init` (runs at boot time) is the parent of everything.

The inventors of Unix (first version of Linux) visualized this a bit like that famous road in the woods...

FORK FOLLOWED BY EXEC

In Linux we normally call `exec` after calling `fork`.

`Fork` creates the process and leaves the parent process an opportunity to “set up” the runtime environment of the child.

Then `exec` launches some other program, but it runs in the same process “context” that the forked child set up.

THE TERM “FORK” HAS LINGERED



If someone says “fork off a thread” or “fork off a process” it refers to creating a new concurrent task.



Later, we might wait for that thread or thread to finish. This is called a join (like when a stream joins a river)

WITH THREADS, YOU CAN “FOLLOW BOTH PATHS” IN THE WOODS...

In computing, some ideas (like recursion) are really earth-shaking

Concurrency is one of them! In some ways very hard to do properly, because of mistakes that can easily arise, and hidden costs that can destroy the speedup benefits.

But in other ways, concurrency is revolutionary because we use the hardware so efficiently.

JOIN IS IMPORTANT, TOO!

If the main thread exits while child threads are still running, this kills the child threads in a chaotic way.

They might not get a chance to clean up and release external resources they were using, like special graphics hardware. They could also throw exceptions, causing your program to “crash” *after* the main thread was done!

FIRST CHALLENGE?

We will have to things (or many) running in one address space.

How will each thread know what to do?

One option is for a main thread to simply tell them.

THE THREAD-CREATION OPERATION CAN TAKE ARGUMENTS

A thread calls a method that returns void, but *can* have arguments.

In this example, “fopener” is being passed a list of files to open:

```
auto fileopener = std::thread(fopener, nfiles, (char**)file_names);
```

OTHER OPTIONS?

As we will see later in this lecture, and the next ones, we could also have some form of “queue of work to be done”

Then threads can remove jobs from the work queue.

For example, wcounter (in fast-wc) had a queue of files to be scanned. Each thread looped, scanning the next file. The file opener thread filled this queue, then (after all files) signaled “done”.

A THREAD CAN RUN A METHOD WITH NO NAME. THIS IS POPULAR IN C++

A *lambda* is just a method that doesn't have a given name.

In effect, a lambda is an expression that can be used as a method:

```
auto fileopener = std::thread([nfiles, file_names]() { code for file opener } );
```

ARGUMENTS

A lambda is just a method that doesn't have a given name.

In effect, an expression that can be used as a method:

```
auto fileopener = std::thread([nfiles, file_names]() { code for file opener } );
```

(): this lambda has no arguments.

Arguments (if any) are supplied at the time the lambda is invoked

CONCEPT OF CAPTURE

Capture: a clean way to access variables in the caller's scope:

```
auto fileopener = std::thread([nfiles, file_names](){ code for file opener } );
```

[nfiles, file_names]... variables “captured” from the caller’s runtime context

OUR EXAMPLE USED C++ LAMBDA NOTATION

Up to now we have only worked with global methods like `main()` and objects that support methods.

It will be useful to see an example of a “lambda”, which is a different way to define a function or method.

Many C++ documentation examples use them, including the examples in some parts of the `std::thread` library.

PASSING A FUNCTION OR A VOID METHOD TO SOMETHING THAT CALLS IT

```
void CallSomething( (int)(std::string) f, std::string str)
{
    cout << "I called f, and it returned " << f(str) << endl;
}
```


PASSING A FUNCTION OR A VOID METHOD TO SOMETHING THAT CALLS IT

```
void CallSomething( (int)(std::string) f, std::string str)
{
    cout << "I called f, and it returned " << f(str) << endl;
}

int func(std::string s)
{
    cout << "This is f, and my argument was " << s << endl;
}
```

PASSING A FUNCTION OR A VOID METHOD TO SOMETHING THAT CALLS IT

```
void CallSomething( (int)(std::string) f, std::string str)
{
    cout << "I called f, and it returned " << f(str) << endl;
}
```

```
int func(std::string s)
{
    cout << "This is f, and my argument was " << s << endl;
}
```

CallSomething(func, "Hello");

PASSING A FUNCTION OR A VOID METHOD TO SOMETHING THAT CALLS IT

```
void CallSomething( (int)(std::string) f, string s)
{
    cout << "I called f, and it returned
```

This is a notation for the type corresponding to a function that takes a string argument and returns an int.

```
int func(std::string s)
{
    cout << "This is f, and my argument was " << s << endl;
    return s.length();
}
```

```
CallSomething(func, "Hello");
```

PASSING A FUNCTION OR A VOID METHOD TO SOMETHING THAT CALLS IT

```
void CallSomething( (int)(std::string) f, std::string str)
{
    cout << "I called f, and it returned " << f << endl;
}
```

Identical logic, but now "func" is passed as a lambda

```
int func(std::string s)
{
```

```
    CallSomething([ ](std::string s){ cout << ... << endl; }, "Hello");
```

```
    return s.length();
}
```

CAPTURE SYNTAX OPTIONS

The lambda can obtain a reference to any valuable in the caller's scope [`&x`] or can capture the value [`x`]. Value means “make a copy for this lambda call”

You can also mix the two, by adding “=”, like this: [`&x, =y`]

Once a lambda captures a scope variable by reference, we say that it has an “alias” to that variable.

WHY DOES C++ HAVE BOTH CAPTURE AND ALSO THREAD ARGUMENTS?

It may feel as if the variables in the [...] part are no different from the parameters in the (...) part.

The difference is that when launching a thread, the caller supplies any the arguments. Each could have a different argument.

Capture is useful because a lambda is actually an “expression” – you can define a lambda in one place but use it elsewhere. In the code that calls the lambda, those captured variables might not be in scope.

... AND NOW OUR THREADS CAN RUN!

Once we have created our threads, each will have:

- Its own stack, on which local variables will be allocated
- Its own “PC” register, and other registers
- Its own independent execution.
- **Access to objects that might also be accessed by other threads!**

This last case can cause issues, as we will see

... RUN ON WHICH CORE?

Which core will a thread run on?

In fact, unless you specify that you want to use more than one core, Linux will run all the threads on the *same* core!

So if we do nothing and create 20 threads, the one CPU core must context switch between those 20 threads. (Linux does this automatically).

LIGHTWEIGHT THREADS

We say that a thread is “lightweight” if it doesn’t have a core dedicated to it. A “heavyweight” thread has its own core.

With lightweight threads we can have many per core.

HOW IT WORKS



Internal to Linux is a clock, and this clock is configured by the kernel to interrupt at various rates.

When your lightweight thread is running, the threads library asks Linux to send a signal after, say, 25ms.

The clock interrupts the kernel, and Linux signals `std::thread`

AT THIS POINT THE SIGNAL HANDLER IS RUNNING IN `STD::THREAD`

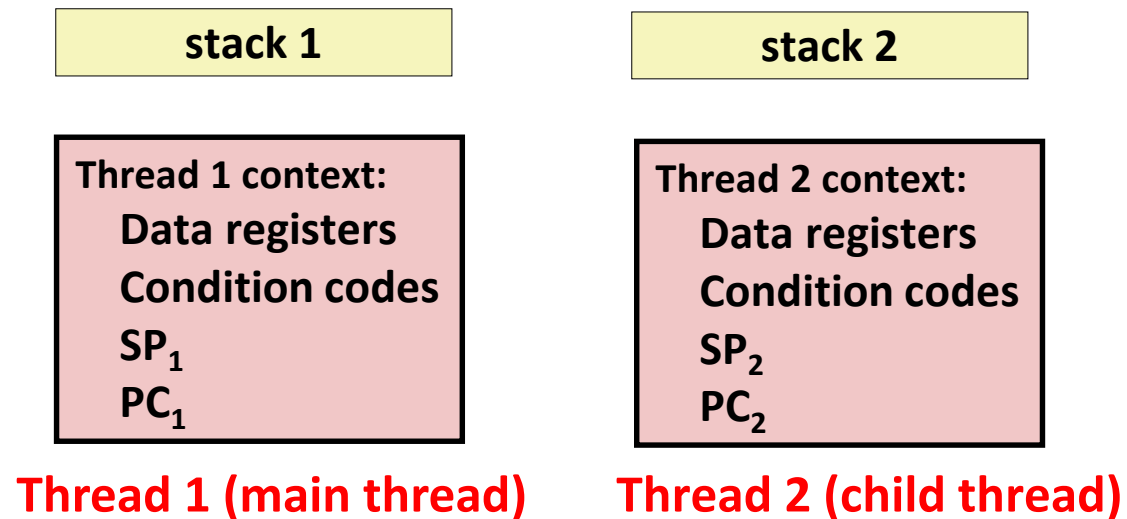
Recall that running a signal handler is like doing a method call, except that the kernel “caused” the method to run.

So at this moment, registers and the PC of the current thread have been pushed to the stack!

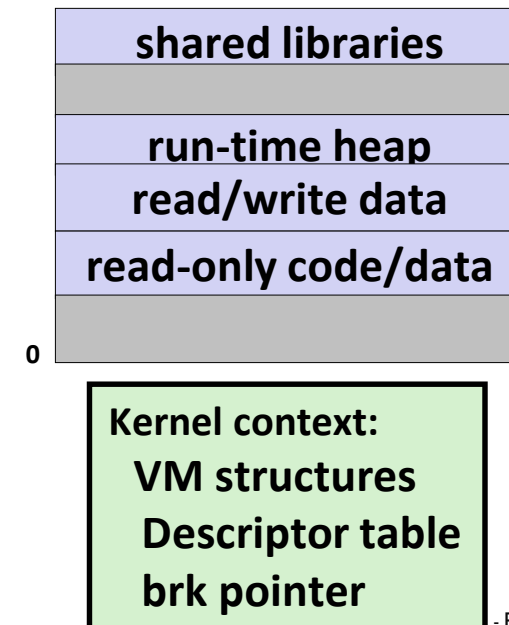
We call this stack and saved register state a “context”.

THREAD CONTEXTS

- **Multiple threads can be associated with a process**
 - Each thread has its own logical control flow
 - Each thread shares the same code, data, and kernel context
 - Each thread has its own stack for local variables
 - but not protected from other threads
 - Each thread has its own thread id (TID)



Shared code and data



THREAD STACKS

Although the main thread has a stack that can grow without limit, this is *not* the situation for spawned child threads.

They have limited stack sizes (default: 2MB, but you can specify a larger size)

Overflow will cause the entire process to crash.

STACK ALLOCATION: SAFE, BUT BE CAUTIOUS

2MB is a large amount of space and won't easily be used up. C++ gives a stack overflow exception if you manage to do so.

But we can't put really big objects on the stack, or do really deep recursion with even medium-sized objects on the stack.

CONTEXT SWITCHING

The `std::thread` scheduler looks at the list of currently active threads to see if any are runnable.

This means: ready to execute, but currently paused.

In some order, it picks one of the runnable threads, and “context switches” to it, meaning that in that other thread, we return from the signal that was used to pause it!

POLICIES FOR SCHEDULING THREADS

We call this context switching step a “scheduling” event

Modern schedulers treat threads differently based on how they are behaving.

A thread that crunches without pausing for long periods will be scheduled for long “quanta” (means “chunks of time”)

... IN CONTRAST

A thread that frequently pauses (like to wait for I/O) will be scheduled more urgently, but with a very small quanta.

The idea is that we want snappy responses to the console, or to other I/O events. In contrast, we shouldn't incur too much overhead for the data-crunching threads, so we let them run for a longer period.

SCHEDULER CONCEPT: MULTILEVEL FEEDBACK QUEUE WITH ROUND ROBIN SCHEDULING

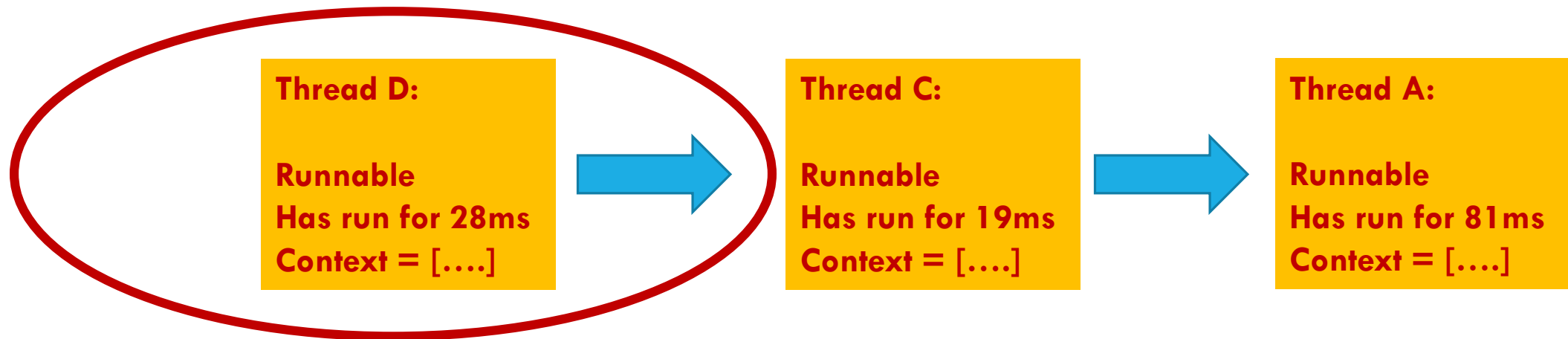
This is a clever idea for letting the behavior of the threads shape the choice of which scheduling quanta to use.

We start with the concept of a round-robin queue.

Our runnable threads are pushed to the end of the queue. The scheduler runs the thread at the front of the queue for a fixed quanta (or until the thread itself pauses to wait for something).

A SCHEDULER QUEUE

Each paused thread has an associated scheduler data structure plus a context. The one on the queue longest is at the front”



... Here, thread D will run next

AN ARRAY OF QUEUES!

```
std::list<std::list<ThreadContext>>
```

*Long-running threads
larger scheduling δ*

Thread X:
Runnable
Has run for 261ms
Context = [...]



Thread Y:
Runnable
Has run for 1819ms
Context = [...]

*Short-running threads
smaller scheduling δ*

Thread D:
Runnable
Has run for 127ms
Context = [...]



Thread C:
Runnable
Has run for 19ms
Context = [...]



Thread A:
Runnable
Has run for 81ms
Context = [...]

RULE FOR MOVING FROM QUEUE TO QUEUE

Track how long each thread has been running (without waiting)

If a thread has run long enough it moves to the next queue up.

If a thread pauses after a very short total time, it moves down.

... SO THERE ARE THREE CONTROL PARAMETERS TO THE ALGORITHM

We can have as many levels as we find helpful, but usually 2 or 3 suffice.

Now, the scheduler can rotate between queues. For total time Δ it runs jobs on the long-running jobs queue. Then it drops to the smaller-jobs queue and runs those.

The long-running jobs get a big per-job δ . Shorter jobs get a smaller δ . So, we run many short jobs compared to long-running jobs.

THAT WAS ALL WITH A SINGLE CORE!

Now we can introduce more than one core to the mix!

In Linux, this requires that you tell bash you wish to use multiple cores, with a command called `taskset`.

Each core has a number, and `taskset` takes a bit-array (in hex) indicating which cores this job will be using, e.g. `taskset 0xFF`.

ONE CORE PER THREAD: -PTHREAD, TASKSET

To activate multicore parallelism you must

1) Compile your program with the gcc flag `-pthread`

2) Use “taskset *mask*” when launching your program:

```
taskset 0xFF fast-wc -n7 -s
```

Arguments to main in fast-wc

... this example says “run fast-wc on cores 0...7”, and also passes in two arguments, `-n7`, `-s`. Fast-wc will run with 7 word-counter threads and one file opener, in “silent” mode.

HOW TASKSET WORKS

Taskset waits for “exclusive ownership” of the requested cores. Only one application can own a given core.

The pthread library is told which cores it owns.

Pthreads will scatter threads over the cores unless you specify a desired core when launching them (via `std::thread`).

DANGER! REMOTE MEMORY!

Recall from early lectures: on a NUMA machine, memory access speeds are very dependent on which core is accessing data in which memory.

NUMA looks like one big memory, but in fact is split into memory banks, and a core is only “close” to one of the memory units.

BUT GOOD NEWS: MALLOC KNOWS ABOUT THREADS

Malloc knows that...

- Objects can be shared without any form of segmentation faults
- But local memory accesses are much faster than remote, unless the remote object is pulled into the L2/L3 cache hierarchy.

Malloc automatically creates new objects in the memory pool closest to that thread's core, if it has room to do so.

EXAMPLE...

In Ken's word counter, each `wcounter` thread had its own `std::map` tree, to hold the (word, counter) pairs.

Those trees were automatically close to the thread that created them, which made them fast to access.

Had `wcounter` used a single tree, shared by all, the program would have been significantly slower!

WE OFTEN MAKE SPARE COPIES TO SHARE AMONG THREADS!

With read-only objects, we often make a replica of the object for each thread to ensure that we will get the fastest possible access to it.

We do this by passing a reference to the “original” copy, and then having the thread make a copy as it starts up, or by passing the object by value as a captured object or argument.

THREADS: THE MAIN RISKS

Synchronization to prevent concurrent memory operations from interfering (if two or more threads access the same data)

Your code might not even speed up, unless you are smart about memory costs and synchronization costs.

Clean termination can be challenging.

SUMMARY

It is quite easy to create both lightweight and heavyweight threads.

They behave similarly, except that lightweight threads are scheduled onto a smaller number of cores.

We also learned new C++ features for writing this kind of code more concisely (lambda expressions).