THREADS

Professor Ken Birman
CS4414 Lecture 14

IDEA MAP FOR TODAY: THREADS!

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.

THREADS ARE AWESOME…

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.

REMINDER: WHAT IS A THREAD?

C++ lets us 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.

THREADS COME IN SEVERAL COLORS

Heavyweight threads are ones running on different CPUs

- They could be sharing one address space…
- … or could be **distinct processes,** each with a distinct address space.

Lightweight threads are ones that share some single CPU. A scheduler context-switches between them periodically.

THREADS IN THE LINUX KERNEL!

Linux itself is a multithreaded program. Each system call runs on a distinct thread, and the Linux scheduler and file system have additional threads of their own.

Linux evolved over decades to take full advantage of this power. It wasn't obvious or easy!

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.

HEAVYWEIGHT OR LIGHTWEIGHT?

They all share one address space

But WordCount was coded to work correctly in either case:

- Linux decides how many cores to allow WordCount to use
- Then the thread scheduler (a part of std::thread) decides which WordCount threads get to run on their own CPU.

HOW LINUX CREATES THREADS/PROCESSES

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"

"Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood…"
-- Robert Frost

People talk about a "fork in the road" although they rarely get to go both ways

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…

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.

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.

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!

HOW APPLICATIONS CREATE 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 ffect, instead of calling a method, we “fork of its own, and the call occurs there.

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

Join pauses to wait for the designated thread to finish

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();
```

FIRST CHALLENGE?

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

How will each thread know what to do? For example, “wcounter” needs to know its thread number (for tree-merge)

One option is for a main thread to simply tell them. std::thread will pass any extra arguments to the function you provide (much like printf, this is done with variadic template logic)

OTHER OPTIONS FOR SENDING DATA TO A THREAD YOU CREATE?

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".

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);
```

A PUZZLE TO THINK ABOUT

Which value of "**n**" will wcounter see? What if someone is updating **n**: will fopener see the original value of **n** from when std::thread was called to fork the thread, or an updated version?

By-value arguments are "cloned" when std::thread is invoked. By-reference arguments are alias names for the value. With an alias, you will see the updated **n**, not the value from when the thread was called. This can lead to bugs!

SLIGHT DIGRESSION: LAMBDAS

To understand the most widely popular notation C++ uses for launching and coordinating threads we need to pause and discuss lambdas

Then we can resurface and talk more about threads

λ's

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 } );
```

λ's

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

λ's

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 });
```

Lambda syntax: [stuff] (args) { code }

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

λ's

CONCEPT OF CAPTURE

Capture: a way to access variables from the **caller's** scope: in the lambda you can only access variables captured, received as arguments, or declared locally.

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

[nfiles, file_names]… variables "captured" from the caller's runtime context

λ's

CONCEPT OF CAPTURE

You can capture “everything”, but this is considered to be poor stylistic practice. Capture “documents” your intentions

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

[=] means “capture everything” by value.
[&] means capture by reference.

λ’s

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

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

Note that there is a typedef for this kind of “function argument” in the std::functions library. Simpler notation, hence popular.

λ’s

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

```
void CallSomething(int (*f)(std::string), 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;
        return s.size();
}
```

λ’s

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

```
void CallSomething(int (*f)(std::string), 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;
        return s.size();
}
```

CallSomething(func, “Hello”);

λ’s

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

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

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

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

CallSomething(func, “Hello”);

λ’s

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

```
void CallSomething(int (*f)(std::string), std::string str)
{
        cout << “I called f, and it return
}

int func(std::string s)

        cout << “This is f, and my argument was “ << s << endl;
        return s.length();
}
```

Identical logic, but now “func” is passed as a lambda

CallSomething([ ](std::string s){ cout << … << endl; return s.size(); }, “Hello”)

λ’s

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.

λ's

WHY DOES C++ THREAD CREATE 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 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.

WHAT IF YOU ACCESS A VARIABLE THAT ISN'T LISTED IN THE CAPTURE CLAUSE?

In the lambda, the only variables defined "by default" are the ones from the capture clause.

It has to declare its arguments, like any function, and also any other variables used within the lambda code

The danger with [=] and [&] is that it is too easy to forget to declare a local variable like **n**, and end up wastefully making a copy or worse, modifying **n** in the outer scope that defined the lambda!

λ's

… BACK ON TOPIC

Now we know lambda syntax and can return to talking about threads

You’ll see more examples using lambda notation in these slides and in future lectures.

~~λ's~~

… 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 in future lectures. The big risk is that one thread could modify something while another is looking, causing one or both to crash. But we can use locking for thread-safety.

… 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

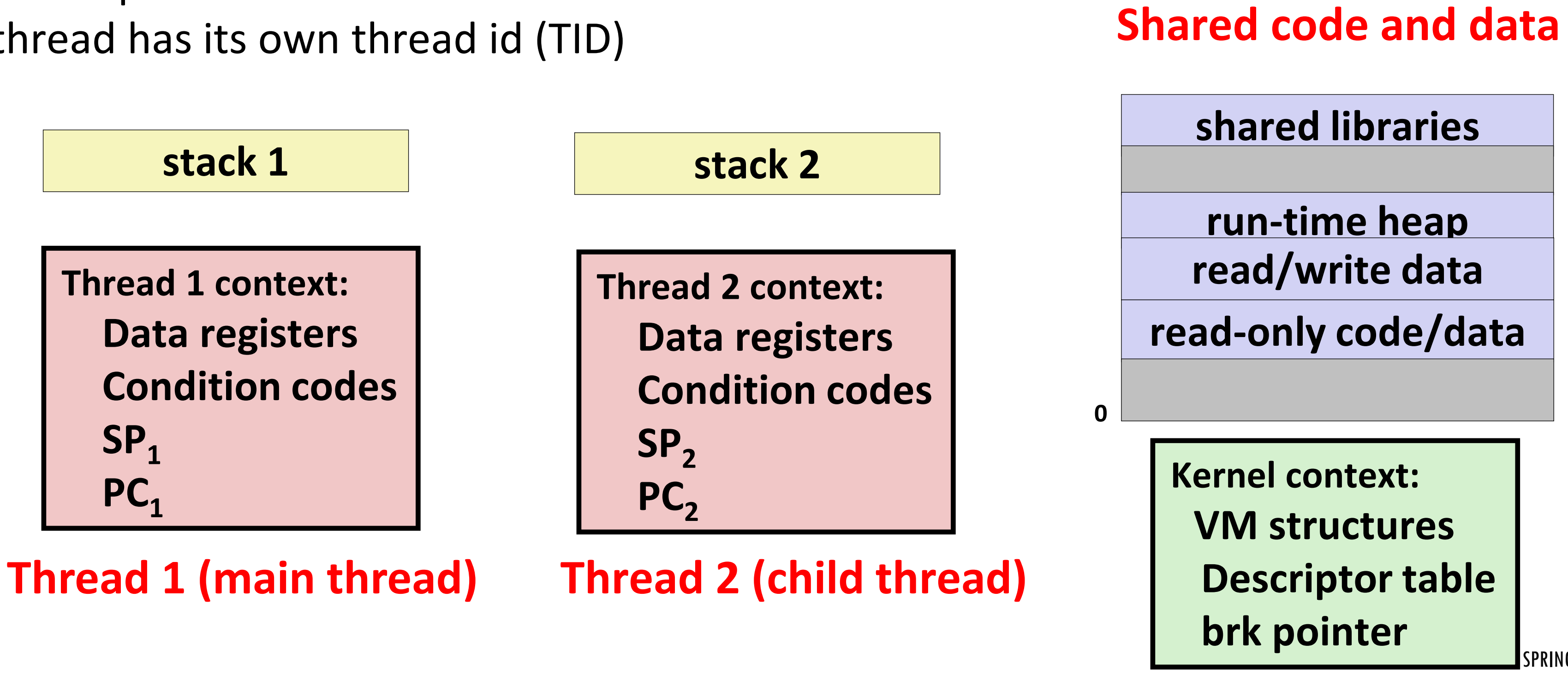
Recall that running a signal handler is like doing a method call, except that the kernel “caused” the method to run.

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)



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

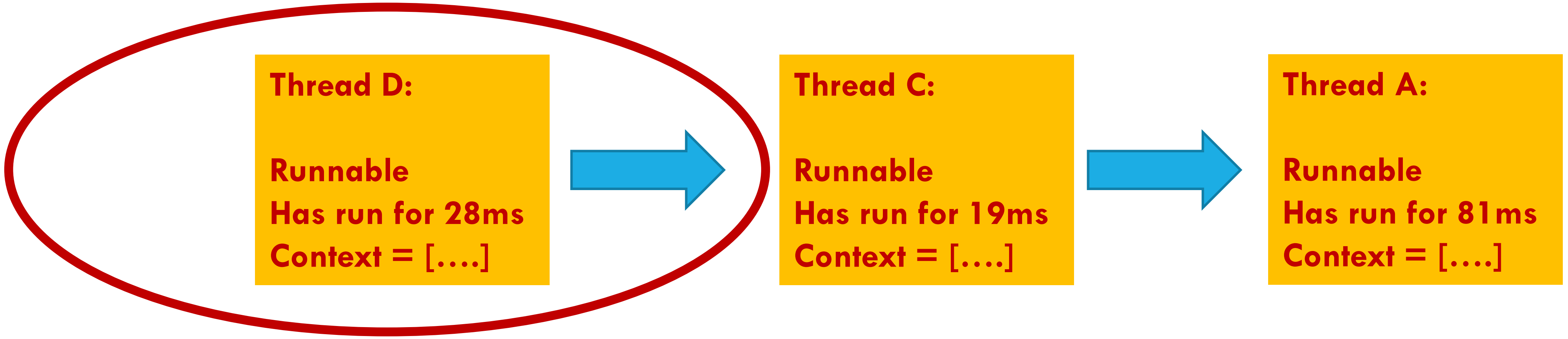
This is a clever idea for letting the behavior of the threads shape the choice of which scheduling quanta to use.

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”



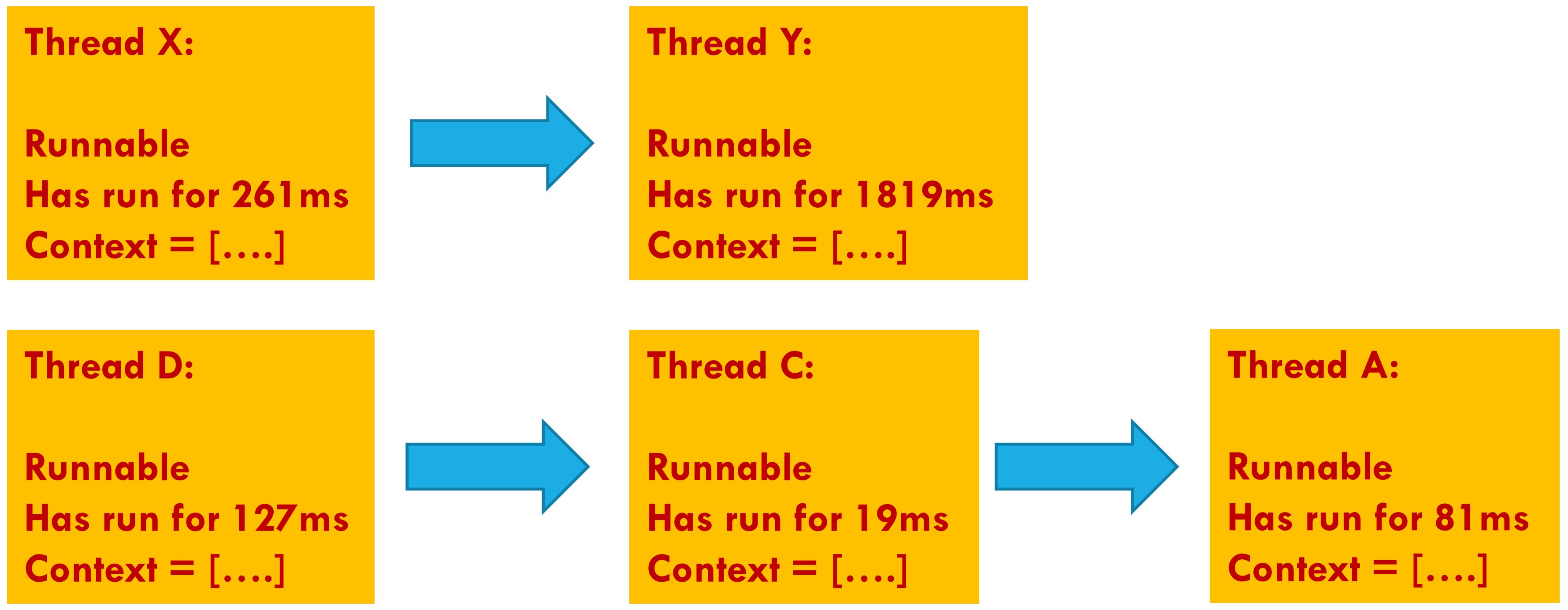
… Here, thread D will run next

MULTILEVEL FEEDBACK QUEUE: AN ARRAY OF QUEUES!

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

Long-running threads
larger scheduling δ

Short-running threads
smaller scheduling δ



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, the default number of cores available to you is hard to predict – it depends on how the OS was configured. To be sure you will run on multiple cores, you use 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 –lpthread

2) Use "taskset *mask*" when launching your program:

Arguments to main in fast-wc

```
taskset 0xFF fast-wc –n7 –s
```

… 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.

WHY IS THIS BAD???

Because if thread A is close to some object X, and thread B is far away from X, their performance can be extremely different.

This is often really confusing if you don't realize that a NUMA effect has snuck into your program.

Some programmers make a local copy of heavily used objects, to ensure that all objects a thread uses intensively are local!

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.

One core can be shared by multiple lightweight threads.

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