Threads

Aside

Incidentally, using threads as a way of structuring your program can sometimes be a good approach, even if you are not concerned with parallelism

Threads

Aside

Incidentally, using threads as a way of structuring your program can sometimes be a good approach, even if you are not concerned with parallelism

For example, have a GUI running on one thread and the computation it controls on another thread

Threads

Aside

Incidentally, using threads as a way of structuring your program can sometimes be a good approach, even if you are not concerned with parallelism

For example, have a GUI running on one thread and the computation it controls on another thread

Called *structure by process*

Concurrency Control

POSIX

More realistically we type cast in the create:

```
void hello(int *n)
{
  printf("hello %d\n", *n);
}

int main(void)
{
  int m;
  pthread_t thr;

  m = 1;
  pthread_create(&thr, NULL, (void*(*)(void*))hello, (void*)&m);
  ...
}
```

Concurrency Control

POSIX

How about two new threads?

```
void hello(int *n)
{
  printf("hello %d\n", *n);
}

int main(void)
{
  int m;
  pthread_t thr1, thr2;

  m = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m);
  m = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m);
  ...
}
```

Concurrency Control

POSIX

This creates two threads, both running the same code, namely `hello`, but on separate threads. Each thread has its own stack, thus its own copy of `n`

Concurrency Control

POSIX

This creates two threads, both running the same code, namely `hello`, but on separate threads. Each thread has its own stack, thus its own copy of `n`

Unfortunately, it is buggy code!

Concurrency Control

POSIX

This creates two threads, both running the same code, namely `hello`, but on separate threads. Each thread has its own stack, thus its own copy of `n`

Unfortunately, it is buggy code!

As usual, it may appear to run correctly several times, printing `"hello 1"` and `"hello 2"` (in either order!)

Concurrency Control

POSIX

This creates two threads, both running the same code, namely `hello`, but on separate threads. Each thread has its own stack, thus its own copy of `n`

Unfortunately, it is buggy code!

As usual, it may appear to run correctly several times, printing `"hello 1"` and `"hello 2"` (in either order!)

But sometimes it prints `"hello 2"` and `"hello 2"`

Concurrency Control

POSIX

This is another case of sequential assumptions not following into parallel code: another race condition

Concurrency Control

POSIX

This is another case of sequential assumptions not following into parallel code: another race condition

It *looks* like we update `m` in between the two new threads

Concurrency Control

POSIX

This is another case of sequential assumptions not following into parallel code: another race condition

It *looks* like we update `m` in between the two new threads

But the new threads are in parallel, running *asynchronously* with the main thread

Concurrency Control

POSIX

What we expect is

main	**1**	**2**
creates **1**	1 starts running	
	reads m=1	
updates m	prints 1	
creates **2**		2 starts running
		reads m=2
		prints 2

Concurrency Control

POSIX

What might happen is

main	**1**	**2**
`creates` **1**		
`updates m`	`1 starts running`	
`creates` **2**	`reads m=2`	`2 starts running`
	`prints 2`	`reads m=2`
		`prints 2`

If thread 1 starts running slightly later

Concurrency Control

POSIX

What might happen is

main	**1**	**2**
`creates` **1**		
`updates m`	`1 starts running`	
`creates` **2**	`reads m=2`	`2 starts running`
	`prints 2`	`reads m=2`
		`prints 2`

If thread 1 starts running slightly later

In fact, this is quite likely, as creating a new thread takes a fair amount of time

Concurrency Control

POSIX

There are three threads in the program: the two running `hello` and the one running `main`

Concurrency Control

POSIX

There are three threads in the program: the two running `hello` and the one running `main`

The threads are *sharing* the variable `m` (via the pointers), so the behaviour of the program is dependent on what order the threads happen to access `m`. This is again bad programming, a data race

Concurrency Control

POSIX

There are three threads in the program: the two running `hello` and the one running `main`

The threads are *sharing* the variable `m` (via the pointers), so the behaviour of the program is dependent on what order the threads happen to access `m`. This is again bad programming, a data race

Be very careful about the values you pass into the thread

Concurrency Control

POSIX

We can fix that race by not sharing:

```
void hello(int *n) {
  printf("hello %d\n", *n);
}

int main(void) {
  int m1, m2;
  pthread_t thr1, thr2;

  m1 = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m1);
  m2 = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m2);

  return 0;
}
```

Concurrency Control

POSIX

But now we (still) have another race condition, which fortunately is easier to spot

Concurrency Control

POSIX

But now we (still) have another race condition, which fortunately is easier to spot

We *might* see both hellos, but more likely is we will see nothing at all

Concurrency Control

POSIX

But now we (still) have another race condition, which fortunately is easier to spot

We *might* see both hellos, but more likely is we will see nothing at all

Again, the `main` thread *continues to run* and `main` might return before the new threads have had chance to get started

Concurrency Control

POSIX

But now we (still) have another race condition, which fortunately is easier to spot

We *might* see both hellos, but more likely is we will see nothing at all

Again, the `main` thread *continues to run* and `main` might return before the new threads have had chance to get started

In C, when the `main` function returns the *whole* process exits, and all of the threads are terminated, possibly before they have had chance to print

Concurrency Control

POSIX

To fix this the initial thread should wait for the other threads to finish

Concurrency Control

POSIX

To fix this the initial thread should wait for the other threads to finish

```
int pthread_join(pthread_t thread, void **retval);
```

Concurrency Control

POSIX

To fix this the initial thread should wait for the other threads to finish

```
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits

Concurrency Control

POSIX

To fix this the initial thread should wait for the other threads to finish

```
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits

This is the main use of the thread identifiers: joining threads (waiting for threads to finish)

Concurrency Control

POSIX

To fix this the initial thread should wait for the other threads to finish

```
int pthread_join(pthread_t thread, void **retval);
```

This blocks the calling thread until the named thread exits

This is the main use of the thread identifiers: joining threads (waiting for threads to finish)

A thread can end by returning from its initial function or by calling `pthread_exit(void *retval);`

Concurrency Control

POSIX

The thread can return a value, which is a pointer. This will be copied into where `retval` in `pthread_join` points

Concurrency Control

POSIX

The thread can return a value, which is a pointer. This will be copied into where `retval` in `pthread_join` points

Use `NULL` if you don’t need a return value

Concurrency Control

POSIX

The thread can return a value, which is a pointer. This will be copied into where `retval` in `pthread_join` points

Use `NULL` if you don’t need a return value

Be careful not to return a pointer to something on the stack of the exiting thread!

Concurrency Control

POSIX

The thread can return a value, which is a pointer. This will be copied into where `retval` in `pthread_join` points

Use `NULL` if you don't need a return value

Be careful not to return a pointer to something on the stack of the exiting thread!

Any thread can wait for any other thread to terminate, as long as it knows the thread's id (the `pthread_t`)

Concurrency Control

POSIX

```
int main(void)
{
  int m1, m2;
  pthread_t thr1, thr2;

  m1 = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m1);
  m2 = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m2);
  pthread_join(thr1, NULL);
  pthread_join(thr2, NULL);
  return 0;
}
```

Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies: the `exit` function means “exit process”

Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies: the `exit` function means “exit process”
- if any thread calls `pthread_exit()` anywhere, that thread dies

Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies: the `exit` function means “exit process”
- if any thread calls `pthread_exit()` anywhere, that thread dies
- if any thread returns from its initial function, that thread dies

Concurrency Control

POSIX

- If any thread calls `exit()` anywhere, the entire process dies: the `exit` function means “exit process”
- if any thread calls `pthread_exit()` anywhere, that thread dies
- if any thread returns from its initial function, that thread dies
- there is no hierarchy of threads, all threads are equal and independent once created

Concurrency Control

POSIX

The only thing to watch out for is the thread running `main`, because in C the `main()` function has an implicit `exit()` after its end. So if it finishes, the entire process subsequently dies

Concurrency Control

POSIX

The only thing to watch out for is the thread running `main`, because in C the `main()` function has an implicit `exit()` after its end. So if it finishes, the entire process subsequently dies

Exercise (For later) Think about what coding would be needed if we wanted always to get `hello 1` printed first and `hello 2` second

Exercise Then generalise to *n* threads

Concurrency Control

POSIX

Advanced Exercise The following code might cause a segmentation violation. Why?

```
int main(void)
{
  int m1, m2;
  pthread_t thr1, thr2;

  m1 = 1;
  pthread_create(&thr1, NULL, (void*(*)(void*))hello, (void*)&m1);
  m2 = 2;
  pthread_create(&thr2, NULL, (void*(*)(void*))hello, (void*)&m2);
  return 0;
}
```

Concurrency Control

Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages

Concurrency Control

Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages

And to sequential-trained programmers

Concurrency Control

Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages

And to sequential-trained programmers

There is nothing in the C language itself to stop parallel stupidities as it was designed as a sequential language

Concurrency Control

Threads

It’s not just C that invites these kinds of racy bugs, but they are common to all library-based parallelisms used in sequential languages

And to sequential-trained programmers

There is nothing in the C language itself to stop parallel stupidities as it was designed as a sequential language

As were many other languages in popular use today

Concurrency Primitives

Atomic Update

Back to primitives

Concurrency Primitives

Atomic Update

Back to primitives

The problem with updates is that there is more than one operation involved: first read, then modify, then store

Concurrency Primitives

Atomic Update

Back to primitives

The problem with updates is that there is more than one operation involved: first read, then modify, then store

Another thread may access the shared resource in between the read and store

Concurrency Primitives

Atomic Update

Back to primitives

The problem with updates is that there is more than one operation involved: first read, then modify, then store

Another thread may access the shared resource in between the read and store

This leads us to another approach to the update race condition by having indivisible *atomic update*

Concurrency Primitives

Atomic Update

This where the hardware supplies a special instruction to, say, increment an integer as a single atomic operation (read-add 1-store)

Concurrency Primitives

Atomic Update

This where the hardware supplies a special instruction to, say, increment an integer as a single atomic operation (read-add 1-store)

This must be in the hardware: the increment instruction must prevent other modifications of that value while it is being incremented

Concurrency Primitives

Atomic Update

This where the hardware supplies a special instruction to, say, increment an integer as a single atomic operation (read-add 1-store)

This must be in the hardware: the increment instruction must prevent other modifications of that value while it is being incremented

The hardware sorts out the sequentialisation in the case of simultaneous (or near-simultaneous) update by different threads

Concurrency Primitives

Atomic Update

This where the hardware supplies a special instruction to, say, increment an integer as a single atomic operation (read-add 1-store)

This must be in the hardware: the increment instruction must prevent other modifications of that value while it is being incremented

The hardware sorts out the sequentialisation in the case of simultaneous (or near-simultaneous) update by different threads

The operation is guaranteed not to be interrupted or interleaved with other threads

Concurrency Primitives

Atomic Update

Note that “atomic” does not mean “fast”

Concurrency Primitives

Atomic Update

Note that “atomic” does not mean “fast”

Depending on the cpu architecture, a single atomic instruction might take possibly hundreds of cpu cycles to execute

Concurrency Primitives

Atomic Update

Note that “atomic” does not mean “fast”

Depending on the cpu architecture, a single atomic instruction might take possibly hundreds of cpu cycles to execute

The hardware might need to sort out memory buses, or cache coherence, or pausing other cores trying to do a simultaneous update, or other low-level stuff

Concurrency Primitives

Atomic Update

Atomics are indeed a reasonable approach, used by many, but they have limitations

Concurrency Primitives

Atomic Update

Atomics are indeed a reasonable approach, used by many, but they have limitations

- Atomic instructions are hard to build in the context of the complexity of caching and so on in modern architectures

Concurrency Primitives

Atomic Update

Atomics are indeed a reasonable approach, used by many, but they have limitations

- Atomic instructions are hard to build in the context of the complexity of caching and so on in modern architectures
- you would need an atomic instruction for each kind of update you might want to do

Concurrency Primitives

Atomic Update

Atomics are indeed a reasonable approach, used by many, but they have limitations

- Atomic instructions are hard to build in the context of the complexity of caching and so on in modern architectures
- you would need an atomic instruction for each kind of update you might want to do
- getting a high-level language compiler to generate code using that instruction will not be straightforward

Concurrency Primitives

Atomic Update

Atomics are indeed a reasonable approach, used by many, but they have limitations

- Atomic instructions are hard to build in the context of the complexity of caching and so on in modern architectures
- you would need an atomic instruction for each kind of update you might want to do
- getting a high-level language compiler to generate code using that instruction will not be straightforward
- they can be slow to execute

Concurrency Primitives

Atomic Update

You do see machine instructions in modern CPUs to do some selection of atomic increment and decrement of integers, add, subtract, logical and, logical or, swap a value in a register with a value in memory, swap two values in memory, and a couple of conditional tests but usually nothing much more than those

Concurrency Primitives

Atomic Update

You do see machine instructions in modern CPUs to do some selection of atomic increment and decrement of integers, add, subtract, logical and, logical or, swap a value in a register with a value in memory, swap two values in memory, and a couple of conditional tests but usually nothing much more than those

Instead, the best approach is to use a more flexible machine instruction that you can build on to make more generic higher-level solutions (see “test and set” and friends, later)

Concurrency Primitives

Atomic Update

You do see machine instructions in modern CPUs to do some selection of atomic increment and decrement of integers, add, subtract, logical and, logical or, swap a value in a register with a value in memory, swap two values in memory, and a couple of conditional tests but usually nothing much more than those

Instead, the best approach is to use a more flexible machine instruction that you can build on to make more generic higher-level solutions (see “test and set” and friends, later)

Indeed, we shall soon see how a lock implementation might be built from atomic operations

Concurrency Primitives

Atomic Update

Do not use atomics for the coursework

Concurrency Primitives

Atomic Update

Do not use atomics for the coursework

To use them effectively you need more more detail that we can't go into right now

Concurrency Primitives

Atomic Update

Exercise For hardware geeks: atomic operations often lock an entire cache line, and can stall the CPU for hundreds of clock cycles while the caches synchronise, so they can slow you down more than you think. Read about this

Exercise For hardware geeks: compare the cost of using a lock against the cost of using an atomic update (the answer can depend on the pattern of access)

Exercise Effective use of atomics involves understanding *memory consistency orderings*. Read about this

Exercise Some programming languages offer atomic datatypes, e.g., Java, C++, Rust. These usually eventually just call the machine instruction atomics. Read about this

Concurrency Primitives

Implementation of Locks

A little more to say about locks…

Concurrency Primitives

Implementation of Locks

A little more to say about locks…

How are locks implemented?

Concurrency Primitives

Implementation of Locks

A little more to say about locks…

How are locks implemented?

They are a flag: say an integer, or even just one bit

Concurrency Primitives

Implementation of Locks

A little more to say about locks…

How are locks implemented?

They are a flag: say an integer, or even just one bit

We might use 1 to indicate locked, and 0 to indicate unlocked

Concurrency Primitives

Implementation of Locks

```
int lock = 0;

void get_lock()
{
   while (lock == 1) {
     deschedule();
   }
   lock = 1;
}
```

i.e., test the flag. If it is already 1, wait; else we can grab it by setting the flag to 1

Concurrency Primitives

Implementation of Locks

```
int lock = 0;

void get_lock()
{
   while (lock == 1) {
     deschedule();
   }
   lock = 1;
}
```

i.e., test the flag. If it is already 1, wait; else we can grab it by setting the flag to 1

Spot the bug!

Concurrency Primitives

Implementation of Locks

There is another update race condition

Concurrency Primitives

Implementation of Locks

There is another update race condition

1	2
`test flag: OK`	`test flag: OK`

Concurrency Primitives

Implementation of Locks

There is another update race condition

1	**2**
`test flag: OK`	`test flag: OK`
`set flag`	`set flag`

Concurrency Primitives

Implementation of Locks

There is another update race condition

1	**2**
`test flag: OK` `set flag`	`test flag: OK` `set flag`

And now both calls to `get_lock` succeed and both threads proceed to enter the critical region

Concurrency Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen

Concurrency Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be atomic

Concurrency Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be atomic

That is the test and the set are inseparable: nothing can get between them

Concurrency Primitives

Implementation of Locks

In between the testing of the flag and the setting of the flag all kinds of other things might happen

Code lines that are textually next to each other like this are widely separated in some sense: what we want is the testing and setting to be atomic

That is the test and the set are inseparable: nothing can get between them

This is another kind of critical region, so we could solve it by using locks…

Concurrency Primitives

Implementation of Locks

Fortunately we don’t have to go into an infinite regression as there are two kinds of solution: hardware and software

Concurrency Primitives

Implementation of Locks

Fortunately we don’t have to go into an infinite regression as there are two kinds of solution: hardware and software

Hardware designers understand mutual exclusion, so the instruction sets of all modern processors have an instruction specifically designed for this

Concurrency Primitives

Implementation of Locks

Fortunately we don't have to go into an infinite regression as there are two kinds of solution: hardware and software

Hardware designers understand mutual exclusion, so the instruction sets of all modern processors have an instruction specifically designed for this

For example the *compare and swap* instruction

Concurrency Primitives

Implementation of Locks

Intel has `cmpxchgb` that atomically operates on a register and a byte in memory

Concurrency Primitives

Implementation of Locks

Intel has `cmpxchgb` that atomically operates on a register and a byte in memory

> *CMPXCHG r/m8, r8*
> *Compare AL with r/m8. If equal, ZF is set and r8 is loaded into r/m8. Else, clear ZF and load r/m8 into AL. This instruction can be used with a LOCK prefix to allow the instruction to be executed atomically*

Concurrency Primitives

Implementation of Locks

In C, its action is like

```
int compare_and_swap(int *reg, int *mem, int new)
{
    if (*reg == *mem) {
       *mem = new;
       return 1;  /* got lock */
    }
    *reg = *mem;
    return 0;    /* fail */
}
```

but the entire thing is done *atomically*

Concurrency Primitives

Implementation of Locks

Using this:

```
int flag = 0;
...
int reg = 0;
// try to set flag to 1
while (compare_and_swap(&reg, &flag, 1) == 0) {
  reg = 0; // try again
}
<CR>
flag = 0;
```

This implements a busy wait

Concurrency Primitives

Implementation of Locks

Using this:

```
int flag = 0;
...
int reg = 0;
// try to set flag to 1
while (compare_and_swap(&reg, &flag, 1) == 0) {
  reg = 0; // try again
}
<CR>
flag = 0;
```

This implements a busy wait

You should spend some time going through this!

Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`

Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`

Early architectures did not have such instructions, so software versions were devised

Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`

Early architectures did not have such instructions, so software versions were devised

These include: Dekker, Dijkstra and Lamport

Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`

Early architectures did not have such instructions, so software versions were devised

These include: Dekker, Dijkstra and Lamport

They are very subtle as they must construct an atomic effect from non-atomic code

Concurrency Primitives

Implementation of Locks

Instructions found in other architectures include `test_and_set` and an atomic `swap`

Early architectures did not have such instructions, so software versions were devised

These include: Dekker, Dijkstra and Lamport

They are very subtle as they must construct an atomic effect from non-atomic code

Exercise Go and read about these