Concurrency Control

Semaphores, locks, barriers, etc., are likened to assembler: low-level, fast, fine control, but very likely to encourage buggy programs
Concurrency Control

Semaphores, locks, barriers, etc., are likened to assembler: low-level, fast, fine control, but very likely to encourage buggy programs

While many programmers are happy using them, others need higher level solutions
Concurrency Control

Semaphores, locks, barriers, etc., are likened to assembler: low-level, fast, fine control, but very likely to encourage buggy programs.

While many programmers are happy using them, others need higher level solutions.

These come in many forms.
Concurrency control can be supported in a high-level language as
Concurrency Control

Concurrency control can be supported in a high-level language as

- added in to an existing language, in library support. We have seen some of this already: the POSIX examples
Concurrency Control

Concurrency control can be supported in a high-level language as

- added in to an existing language, in library support. We have seen some of this already: the POSIX examples
- fudged into the syntax of an existing language
Concurrency Control

Concurrency control can be supported in a high-level language as

- added in to an existing language, in library support. We have seen some of this already: the POSIX examples
- fudged into the syntax of an existing language
- part of the initial design of a new language
Concurrency Control

Concurrency control can be supported in a high-level language as

- added in to an existing language, in library support. We have seen some of this already: the POSIX examples
- fudged into the syntax of an existing language
- part of the initial design of a new language

We shall be looking at all of these approaches
Concurrency Control

There is a lot of sequential code out there that people would like to run faster on parallel hardware.
Concurrency Control

There is a lot of sequential code out there that people would like to run faster on parallel hardware.

While there is a lot of effort being put into automatic analysis of code to discover and exploit parallelism, the results are sporadic.
Concurrency Control

There is a lot of sequential code out there that people would like to run faster on parallel hardware.

While there is a lot of effort being put into automatic analysis of code to discover and exploit parallelism, the results are sporadic.

Functional languages offer a decent hope here, but not much code is functional style.
Concurrency Control

There is a lot of sequential code out there that people would like to run faster on parallel hardware.

While there is a lot of effort being put into automatic analysis of code to discover and exploit parallelism, the results are sporadic.

Functional languages offer a decent hope here, but not much code is functional style.

So code needs to be rewritten to make best advantage of parallelism.
Concurrency Control

There is a lot of sequential code out there that people would like to run faster on parallel hardware.

While there is a lot of effort being put into automatic analysis of code to discover and exploit parallelism, the results are sporadic.

Functional languages offer a decent hope here, but not much code is functional style.

So code needs to be rewritten to make best advantage of parallelism.

The hope (and economics) is we can take existing code using an existing language and modify it.
It’s not a good way of doing things, but rewriting from scratch is just too expensive
It’s not a good way of doing things, but rewriting from scratch is just too expensive

Plus there are lots of programmers with extensive expertise in languages like C and Fortran
It’s not a good way of doing things, but rewriting from scratch is just too expensive.

Plus there are lots or programmers with extensive expertise in languages like C and Fortran.

So we are led to the approach of taking, say C, and adding parallelism to it.
Concurrency Control
Libraries

It’s not a good way of doing things, but rewriting from scratch is just too expensive.

Plus there are lots of programmers with extensive expertise in languages like C and Fortran.

So we are led to the approach of taking, say C, and adding parallelism to it.

The easiest way is to leave the language itself untouched, just adding a library of functions that do parallelism.
For example, the POSIX pthread approach
For example, the POSIX pthread approach

Note: We have been using C and the POSIX library to illustrate points, but this library technique applies to all sensible languages
We have seen a few POSIX functions `pthread_mutex_init` and so on
We have seen a few POSIX functions `pthread_mutex_init` and so on.

There are a couple more that we need to make parallel programs.
We have seen a few POSIX functions `pthread_mutex_init` and so on.

There are a couple more that we need to make parallel programs.

Chiefly, how do we get multiple threads to run?
Creating threads:

```c
#include <pthread.h>
int pthread_create(pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(*start_routine) (void *),
                  void *arg);
```

Link with `-lpthread`
Creating threads:

#include <pthread.h>
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg);

Link with -lpthread

This looks ugly, but is quite simple in practice: it creates a new thread running the function start_routine on the argument arg
It returns a thread identifier in argument `thread`. This can be used to do things to the thread.
It returns a thread identifier in argument `thread`. This can be used to do things to the thread.

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread).
It returns a thread identifier in argument `thread`. This can be used to do things to the thread.

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread).

The `start_routine` names a function of one argument that the thread will start executing when it begins running.
It returns a thread identifier in argument `thread`. This can be used to do things to the thread.

`attr` is a thread attribute: you probably will never need more than the default (NULL), but occasionally you might (stack size; detached thread).

The `start_routine` names a function of one argument that the thread will start executing when it begins running.

The `arg` is the argument passed to the function (a pointer).
Concurrency Control

POSIX

Roughly:

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

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

    m = 1;
    // should check return value from create ...
    pthread_create(&thr, NULL, hello, (void*)&m);
    ...
}
```
This makes a new thread that runs separately from and possibly simultaneously with the main thread.
This makes a new thread that runs separately from and possibly simultaneously with the main thread.

It runs the function `hello` with argument a pointer to `m`
This makes a new thread that runs separately from and possibly simultaneously with the main thread.

It runs the function `hello` with argument a pointer to `m`.

It does this concurrently with the `main` function, which continues to run.
This makes a new thread that runs separately from and possibly simultaneously with the main thread.

It runs the function `hello` with argument a pointer to `m`.

It does this concurrently with the `main` function, which continues to run.

The `start_function` will generally call lots of other functions to perform whatever the thread needs to do.
This makes a new thread that runs separately from and possibly simultaneously with the main thread.

It runs the function `hello` with argument a pointer to `m`.

It does this concurrently with the `main` function, which continues to run.

The `start_function` will generally call lots of other functions to perform whatever the thread needs to do.

Ugly type casting is common in C.
Aside

This also works on uniprocessor systems: the threads are scheduled in a similar way to processes.
This also works on uniprocessor systems: the threads are scheduled in a similar way to processes.

You can debug a concurrent program on a sequential machine, but it may not exhibit some of the more subtle race conditions or deadlocks as the threads won’t truly be running in parallel.
Threads

Aside

You can make more threads than there are cores: for example, run 10 threads on a 4 core machine
You can make more threads than there are cores: for example, run 10 threads on a 4 core machine.

And the OS will schedule between the threads.
You can make more threads than there are cores: for example, run 10 threads on a 4 core machine.

And the OS will schedule between the threads.

A thread that is blocked (e.g., waiting on a lock or a semaphore) typically would not be scheduled, so it uses no CPU cycles.
Threads

Aside

You can make more threads than there are cores: for example, run 10 threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock or a semaphore) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not, though, as scheduling eats up CPU time
You can make more threads than there are cores: for example, run 10 threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock or a semaphore) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not, though, as scheduling eats up CPU time

A common error is to create hundreds of threads and then wonder why everything is running slowly
Threads

Aside

You can make more threads than there are cores: for example, run 10 threads on a 4 core machine

And the OS will schedule between the threads

A thread that is blocked (e.g., waiting on a lock or a semaphore) typically would not be scheduled, so it uses no CPU cycles

The question remains whether that is worth it or not, though, as scheduling eats up CPU time

A common error is to create hundreds of threads and then wonder why everything is running slowly

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

```c
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);
    ...
}
```