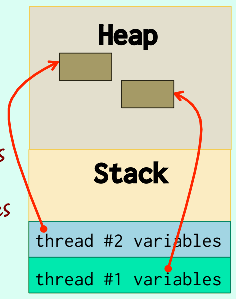
CS205

C / C++

Stéphane Faroult
faroult@sustc.edu.cn

Wang Wei (Vivian) vivian2017@aliyun.com

Threads Basics



No problem with local variables

No problem with heap variables

Problem with static and global variables

We have seen last time that with thread we must be careful with global and static variables that all threads can access.

Threads Basics

```
int pthread_create(pthread_t * thread,
     pthread_attr_t * attr,
     void * (* start_routine) (void *),
     void * arg);
```

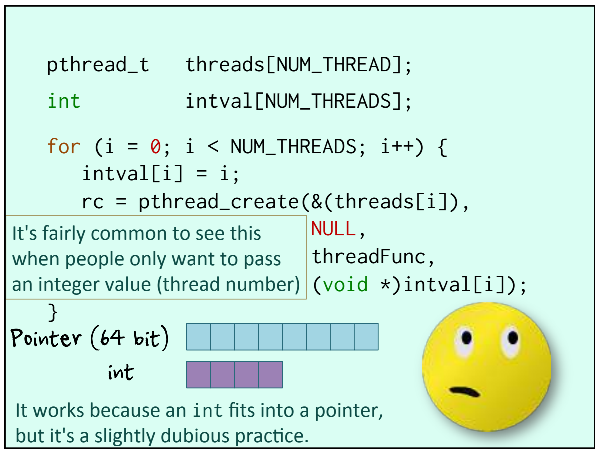
Must be private to the thread

Typically, what is passed to a thread must be for this thread only.

```
pthread_t    threads[NUM_THREAD];
THREAD_DATA_T data[NUM_THREADS];

for (i = 0; i < NUM_THREADS; i++) {
   // Set-up data[i]
   rc = pthread_create(&(threads[i]),
                       NULL,
                       threadFunc,
                       (void *)&(data[i]));
}
```

Can also allocate dynamically

You can pass an address of an array that contains one item per thread, who allocate dynamically for each thread.



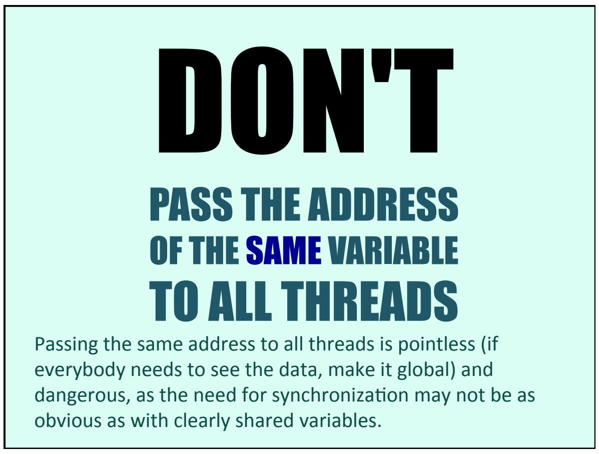
```
pthread_t   threads[NUM_THREAD];
int         intval[NUM_THREADS];

for (i = 0; i < NUM_THREADS; i++) {
   intval[i] = i;
   rc = pthread_create(&(threads[i]),
                       NULL,
                       threadFunc,
                       (void *)intval[i]);
   }
```

It's fairly common to see this when people only want to pass an integer value (thread number)

Pointer (64 bit) ⬜⬜⬜⬜⬜⬜⬜⬜

int ⬜⬜⬜⬜

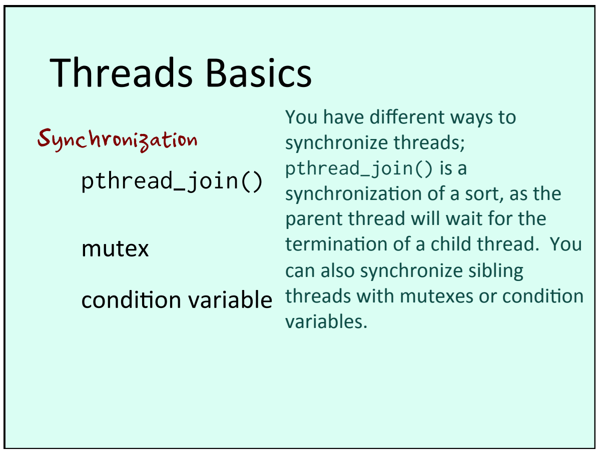
It works because an `int` fits into a pointer, but it's a slightly dubious practice.



DON'T

PASS THE ADDRESS OF THE SAME VARIABLE TO ALL THREADS

Passing the same address to all threads is pointless (if everybody needs to see the data, make it global) and dangerous, as the need for synchronization may not be as obvious as with clearly shared variables.

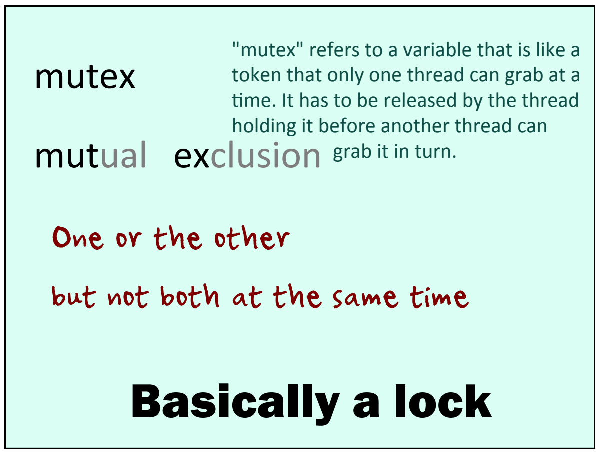


Threads Basics

Synchronization

- pthread_join()
- mutex
- condition variable

You have different ways to synchronize threads; `pthread_join()` is a synchronization of a sort, as the parent thread will wait for the termination of a child thread. You can also synchronize sibling threads with mutexes or condition variables.



mutex

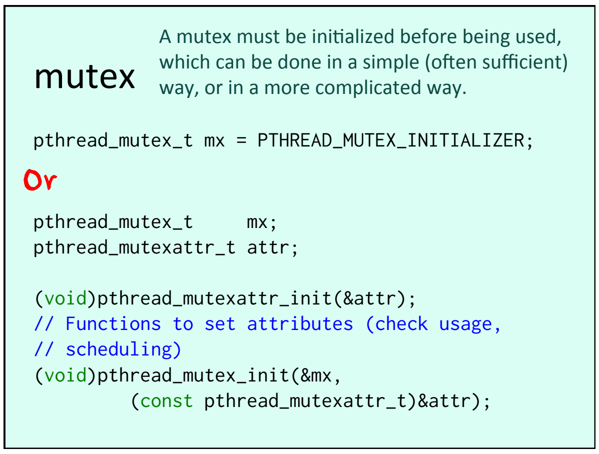
mutual exclusion

"mutex" refers to a variable that is like a token that only one thread can grab at a time. It has to be released by the thread holding it before another thread can grab it in turn.

One or the other

but not both at the same time

Basically a lock



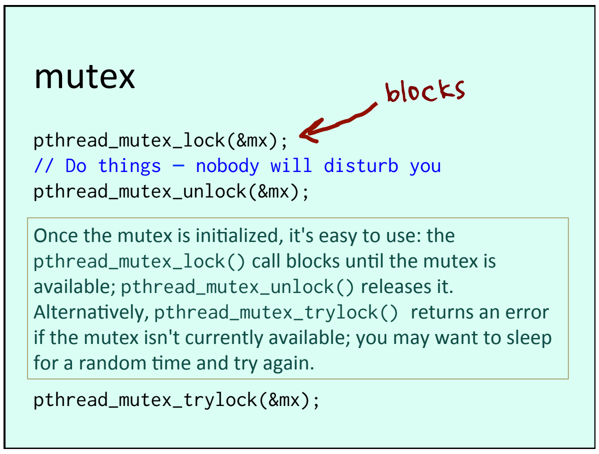
mutex

A mutex must be initialized before being used, which can be done in a simple (often sufficient) way, or in a more complicated way.

```
pthread_mutex_t mx = PTHREAD_MUTEX_INITIALIZER;
```

Or

```
pthread_mutex_t     mx;
pthread_mutexattr_t attr;

(void)pthread_mutexattr_init(&attr);
// Functions to set attributes (check usage,
// scheduling)
(void)pthread_mutex_init(&mx,
         (const pthread_mutexattr_t)&attr);
```



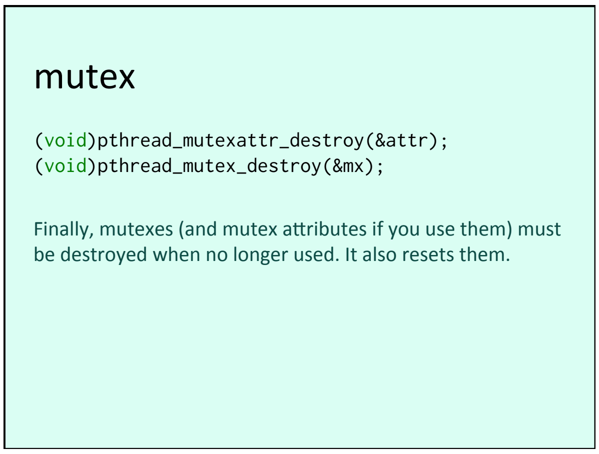
mutex

blocks

```
pthread_mutex_lock(&mx);
// Do things — nobody will disturb you
pthread_mutex_unlock(&mx);
```

Once the mutex is initialized, it's easy to use: the `pthread_mutex_lock()` call blocks until the mutex is available; `pthread_mutex_unlock()` releases it. Alternatively, `pthread_mutex_trylock()` returns an error if the mutex isn't currently available; you may want to sleep for a random time and try again.

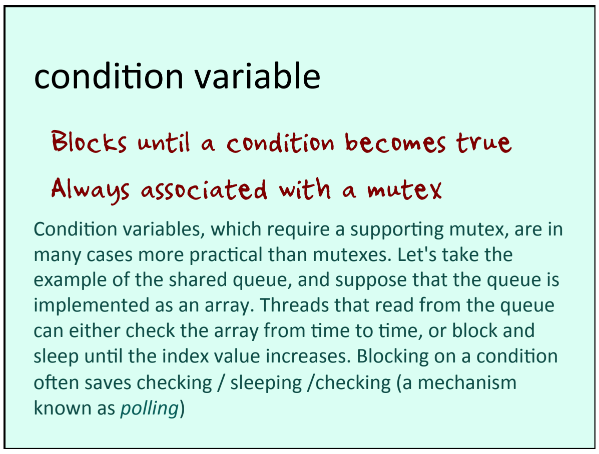
```
pthread_mutex_trylock(&mx);
```



mutex

```
(void)pthread_mutexattr_destroy(&attr);
(void)pthread_mutex_destroy(&mx);
```

Finally, mutexes (and mutex attributes if you use them) must be destroyed when no longer used. It also resets them.



condition variable

Blocks until a condition becomes true

Always associated with a mutex

Condition variables, which require a supporting mutex, are in many cases more practical than mutexes. Let's take the example of the shared queue, and suppose that the queue is implemented as an array. Threads that read from the queue can either check the array from time to time, or block and sleep until the index value increases. Blocking on a condition often saves checking / sleeping /checking (a mechanism known as *polling*)

condition variable

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Or

```
pthread_cond_t      cond;

(void)pthread_cond_init(&cond, NULL);
```

Could be a condition attribute pointer

As with mutexes, you have the simple and the hard way to initialize condition variables.

condition variable

1

```
pthread_mutex_lock(&mx);
// Modify a value
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mx);
```

2

```
pthread_mutex_lock(&mx);
pthread_cond_wait(&cond, &mx);
// Do things
pthread_mutex_unlock(&mx);
```

unlocks the mutex while it waits

Thread one that modifies the variable sends (to all threads) a special signal when the condtition changes. Thread two waits for this signal in a function that first unlocks the mutex when it goes to sleep, then locks it again when it is awaken.

condition variable

```
(void)pthread_cond_destroy(&cond);
```

Like mutexes (and like the associated mutex) condition variables and their attributes have to be destroyed when no longer used.

An overview of the course

Flickr: Dru Bloomfield

Generalities and Control Flow

VERY strong link between C and Unix systems

Not many datatypes – char = byte = integer

Beware of assignment/comparison

Errors, built-in functions, strings and Unicode

Functions very often return an error code

Many standard functions, but not THAT many compared to Java

Primitive strings. Beware with multibyte characters

Pointers, Structures and Files

Directly accessing memory is the strong point of C – and difficult for young (and sometimes for experienced) programmers

Structures are the closest you'll ever get to a class in pure C

Files are easy to manage

Make – Memory Management

make very useful as soon as you have more than one .c file

Keep a close track of memory that you reserve and don't forget to free it!

Recursion

Another point that many people find difficult

Don't abuse it but it makes code easy for complicated operations

Must-have for C data structures

Data Structures

Some generic "patterns" that support many variations

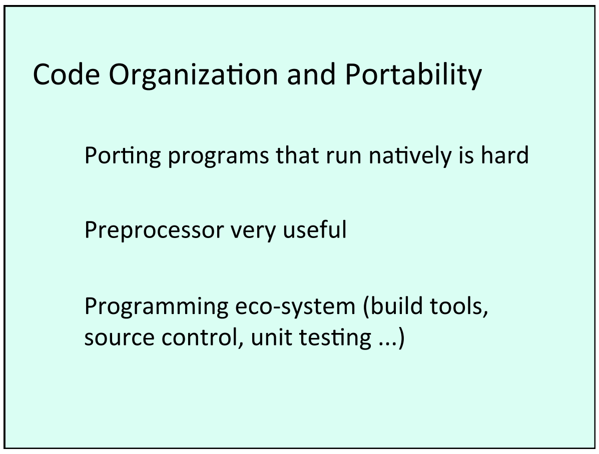
It all depends on how you want to search data

C++

Obvious lack of a safety net in C

C++ thought for "industrial usage"

Class design tough, class usage easy

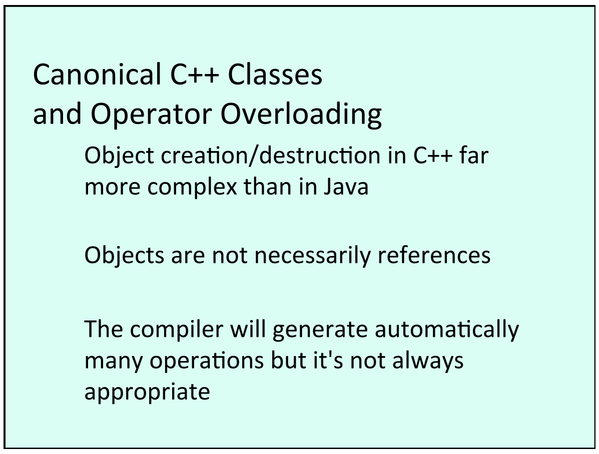


Code Organization and Portability

Porting programs that run natively is hard

Preprocessor very useful

Programming eco-system (build tools, source control, unit testing ...)

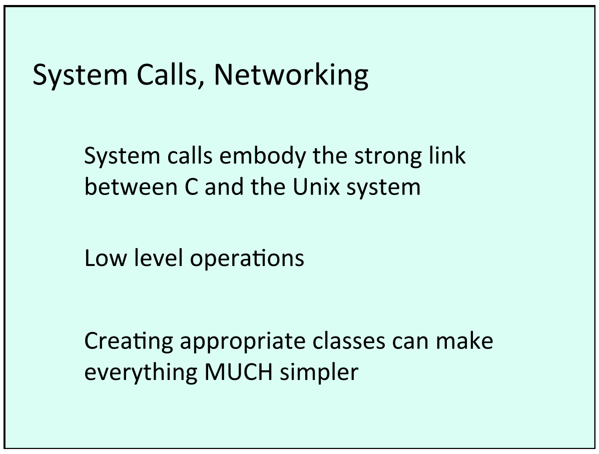


Canonical C++ Classes and Operator Overloading

Object creation/destruction in C++ far more complex than in Java

Objects are not necessarily references

The compiler will generate automatically many operations but it's not always appropriate

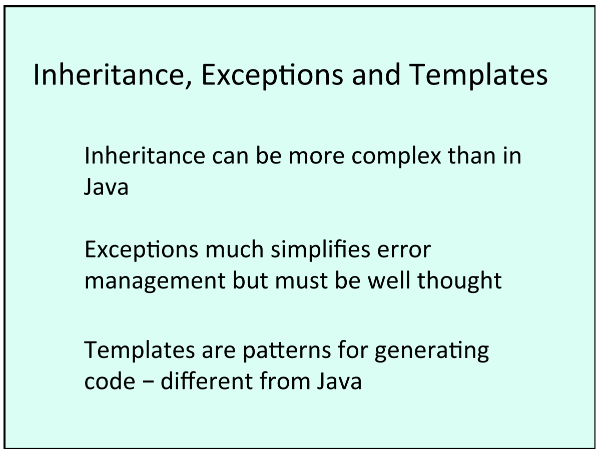


System Calls, Networking

System calls embody the strong link between C and the Unix system

Low level operations

Creating appropriate classes can make everything MUCH simpler



Inheritance, Exceptions and Templates

Inheritance can be more complex than in Java

Exceptions much simplifies error management but must be well thought

Templates are patterns for generating code – different from Java

Processes, Signals, IPC, Threads

- Taking advantage of hardware with many processors
- Almost mandatory for servers
- Communication through signals or more sophisticated means
- Synchronization is tough