CS241 Principles and Practice of Problem Solving

Lecture 21: GUI Programming with Qt IV

Yuye Ling, Ph.D.

Shanghai Jiao Tong University
John Hopcroft Center for Computer Science

November 25, 2019

Copyright Notice

This lecture covers some preliminary ideas about Operating system. I have reused some lecture notes from Prof. David Culler of UC Berkeley. Details could be found here.
For the Qt part, the original materials could be found here.

Why do we care about multi-tasking in GUI

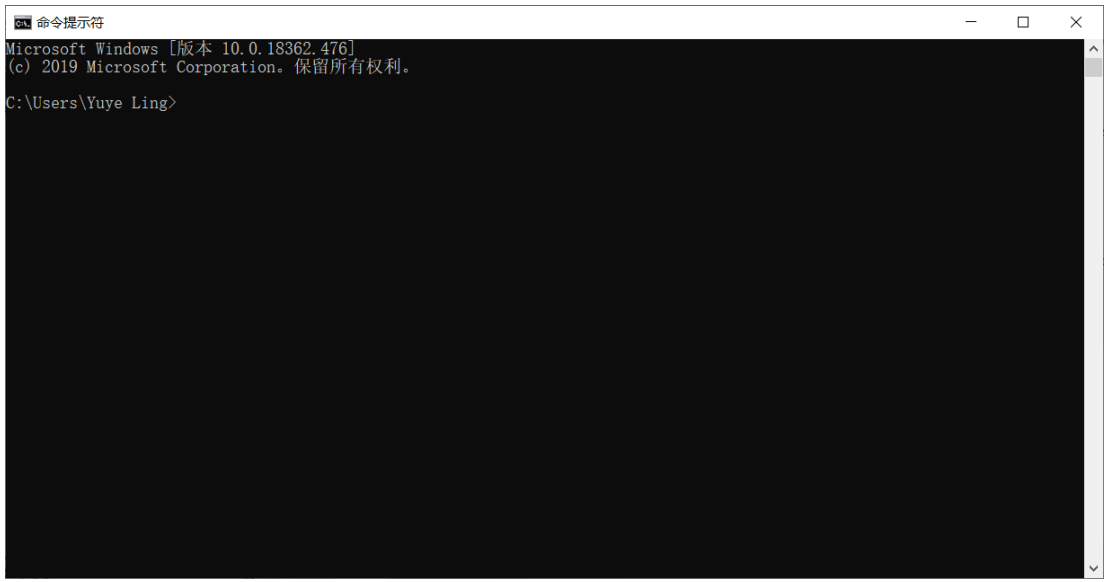
- What is the most sophisticated GUI you could think about?

Why do we care about multi-tasking in GUI

- What is the most sophisticated GUI you could think about? What about operating system?

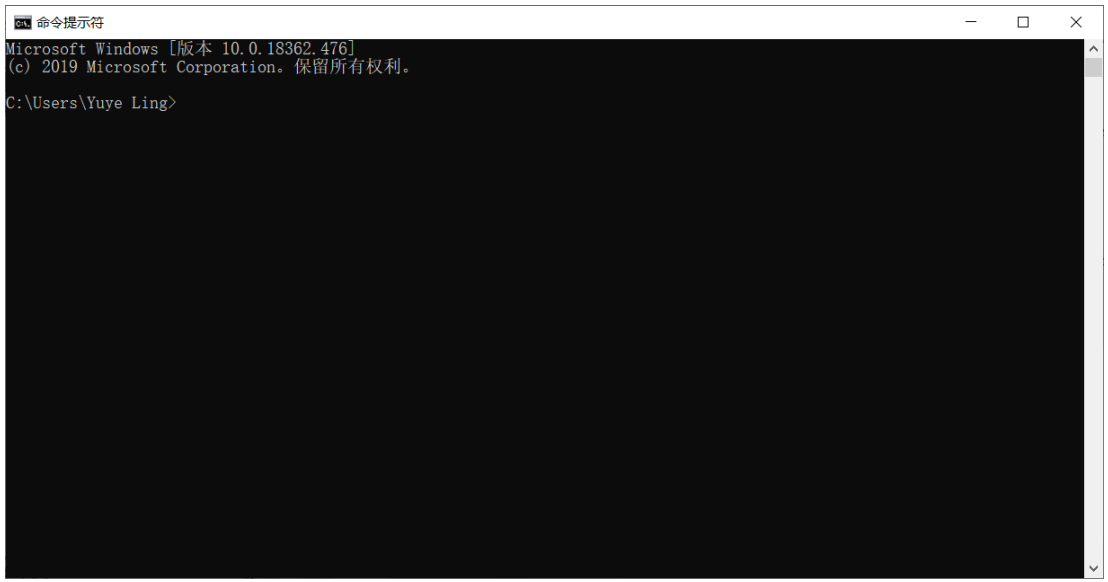
Why do we care about multi-tasking in GUI

- What is the most sophisticated GUI you could think about? What about operating system?
- Think about a CLI OS (e.g. DOS), how many tasks can users perform simultaneously?



Why do we care about multi-tasking in GUI

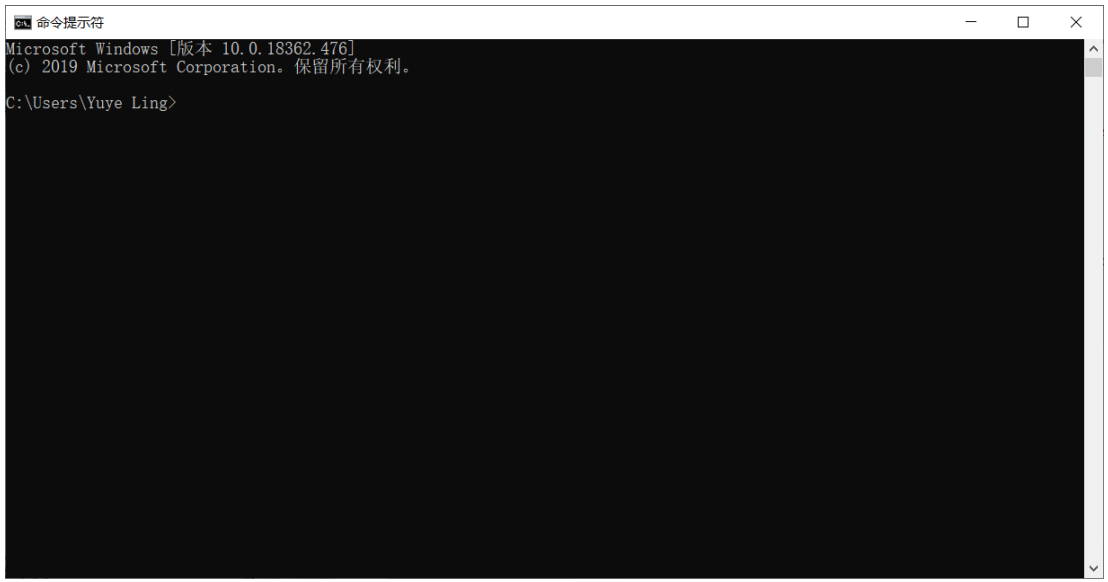
- What is the most sophisticated GUI you could think about? What about operating system?
- Think about a CLI OS (e.g. DOS), how many tasks can users perform simultaneously?



- One.

Why do we care about multi-tasking in GUI

- What is the most sophisticated GUI you could think about? What about operating system?
- Think about a CLI OS (e.g. DOS), how many tasks can users perform simultaneously?



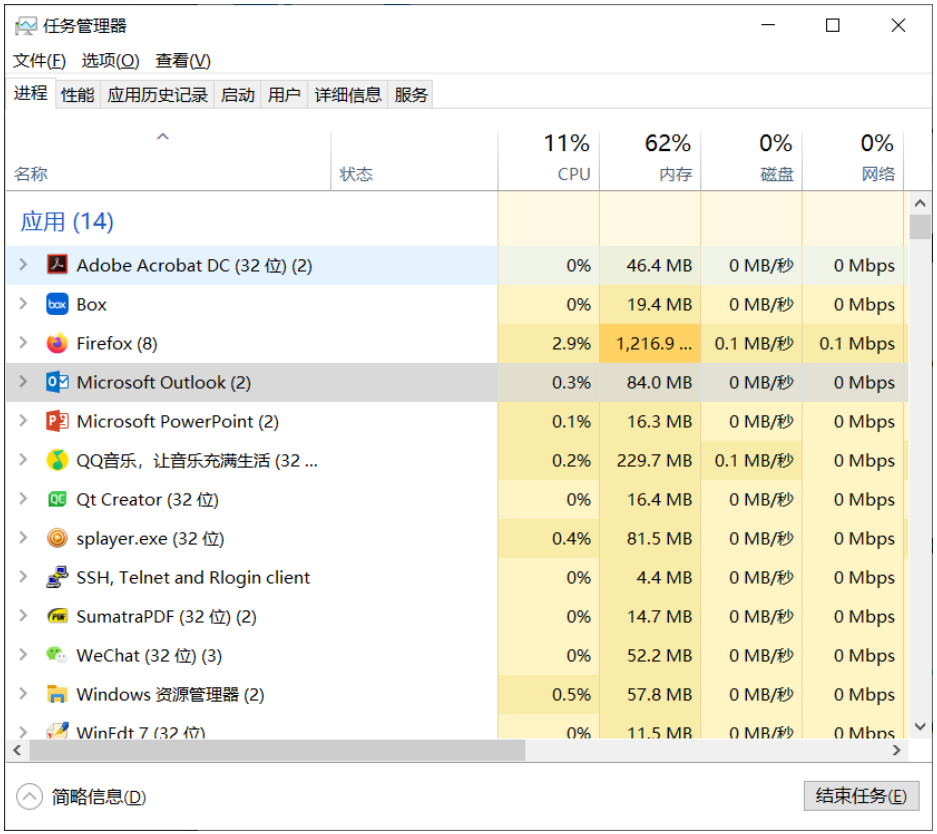
- One. Why is that?

Why do we care about multi-tasking in GUI

- What about Windows (or MacOS)?

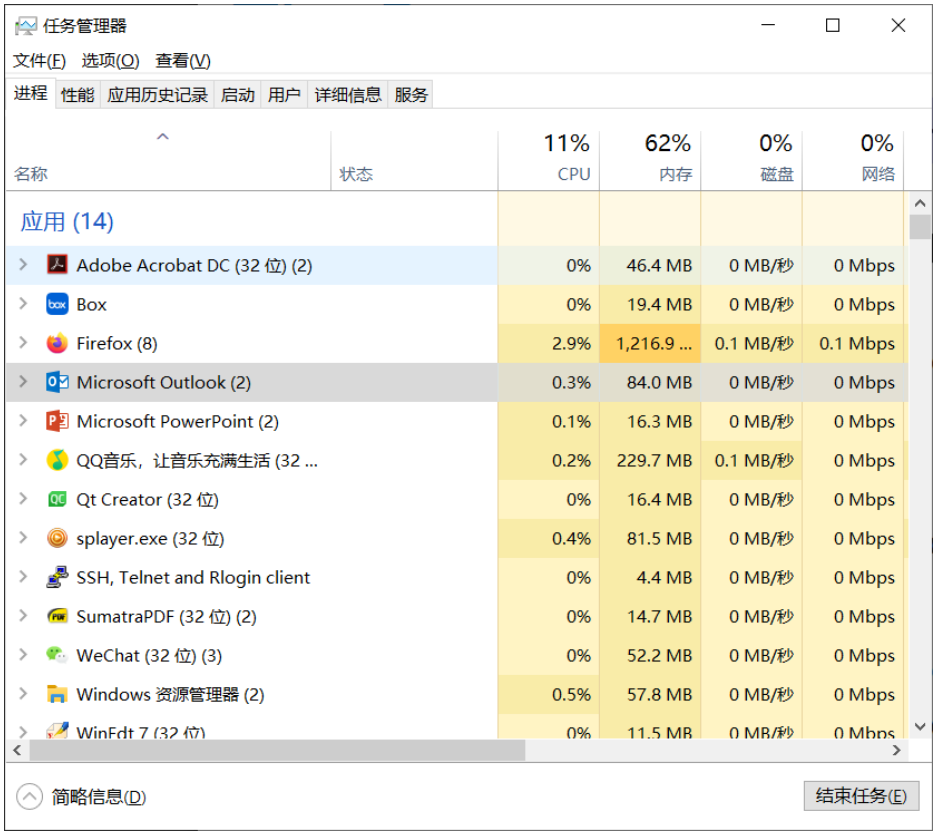
Why do we care about multi-tasking in GUI

- What about Windows (or MacOS)?



Why do we care about multi-tasking in GUI

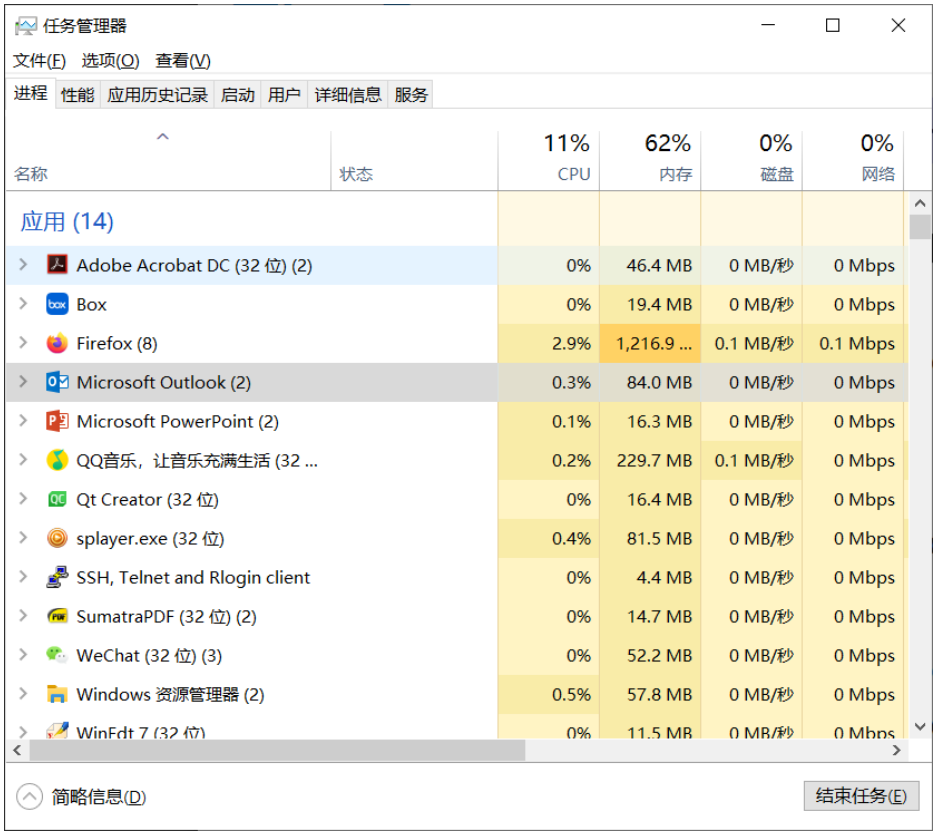
- What about Windows (or MacOS)?



- How is that implemented?

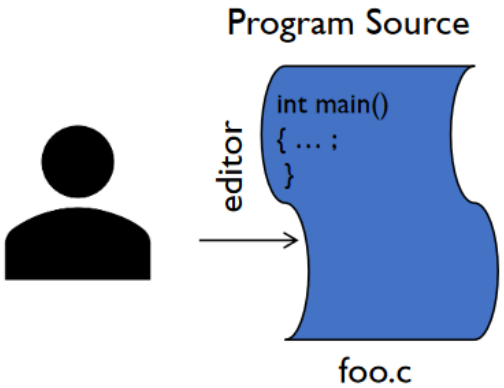
Why do we care about multi-tasking in GUI

- What about Windows (or MacOS)?

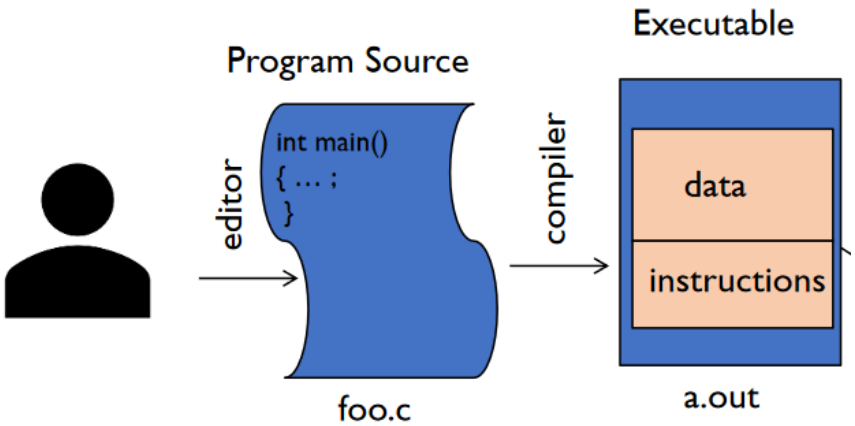


- How is that implemented? Multi-processing/threading

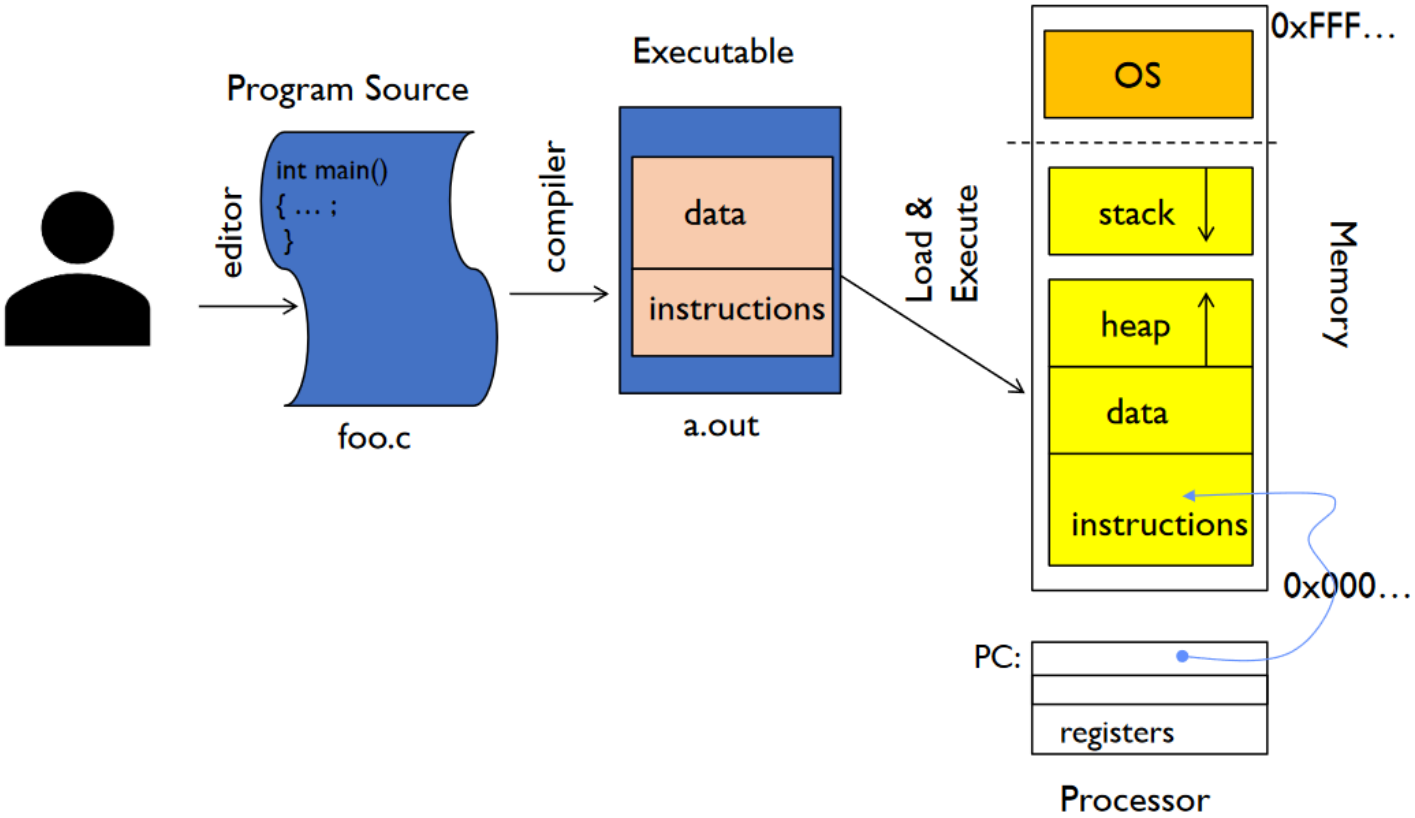
Review: compiling a program and let it run on OS



Review: compiling a program and let it run on OS



Review: compiling a program and let it run on OS



The motivation

The capability of *handling* multiple things at once is important in applications.

The motivation

The capability of *handling* multiple things at once is important in applications.

- For OS: processes, interrupts, background system maintenance

The motivation

The capability of *handling* multiple things at once is important in applications.

- For OS: processes, interrupts, background system maintenance
- For Servers: multiple connections handled simultaneously

The motivation

The capability of *handling* multiple things at once is important in applications.

- For OS: processes, interrupts, background system maintenance
- For Servers: multiple connections handled simultaneously
- Parallel programs: to achieve best performance

The motivation

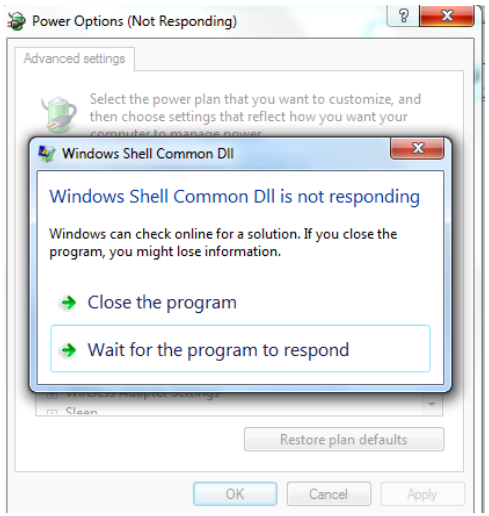
The capability of *handling* multiple things at once is important in applications.

- For OS: processes, interrupts, background system maintenance
- For Servers: multiple connections handled simultaneously
- Parallel programs: to achieve best performance
- GUI: to achieve user responsiveness while doing computation

The motivation

The capability of *handling* multiple things at once is important in applications.

- For OS: processes, interrupts, background system maintenance
- For Servers: multiple connections handled simultaneously
- Parallel programs: to achieve best performance
- GUI: to achieve user responsiveness while doing computation

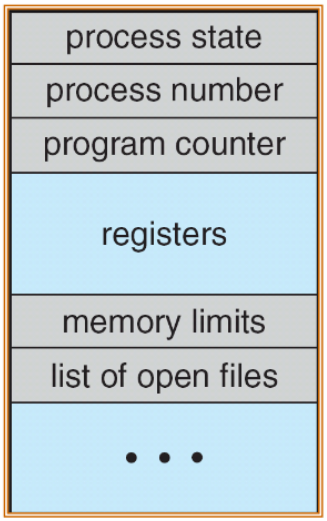


Jeff Dean's "Numbers everyone should know"

L1 cache reference	0.5 ns
Branch mispredict	5 ns
L2 cache reference	7 ns
Mutex lock/unlock	25 ns
Main memory reference	100 ns
Compress 1K bytes with Zippy	3,000 ns
Send 2K bytes over 1 Gbps network	20,000 ns
Read 1 MB sequentially from memory	250,000 ns
Round trip within same datacenter	500,000 ns
Disk seek	10,000,000 ns
Read 1 MB sequentially from disk	20,000,000 ns
Send packet CA->Netherlands->CA	150,000,000 ns

What is a process

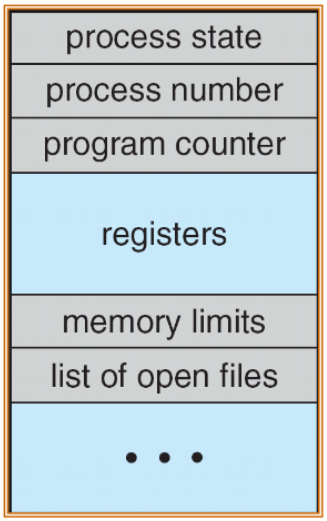
- Definition: **execution environment with restricted rights**



Process Control Block

What is a process

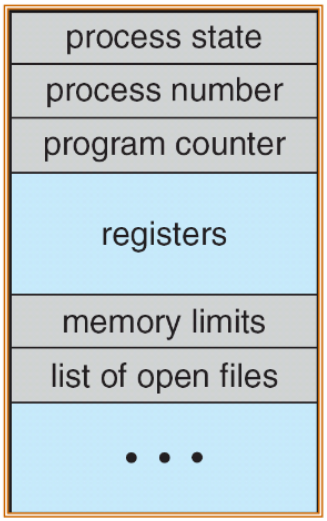
- Definition: **execution environment with restricted rights**



- Human language: an individual application or program

What is a process

- Definition: **execution environment with restricted rights**



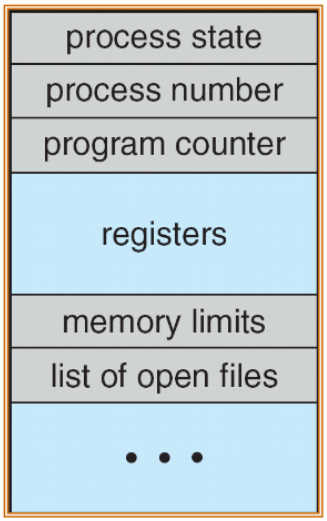
Process Control Block

  - Owns memory

- Human language: an individual application or program

What is a process

- Definition: **execution environment with restricted rights**

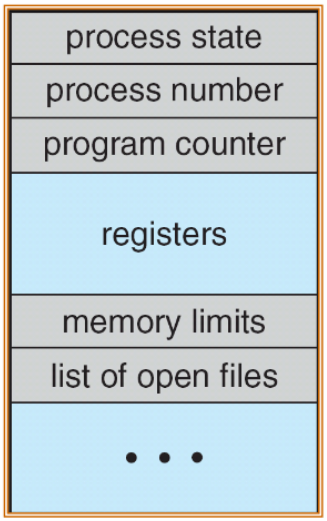


  - Owns memory
  - Owns file descriptors, file system context

- Human language: an individual application or program

What is a process

- Definition: **execution environment with restricted rights**



  - Owns memory
  - Owns file descriptors, file system context
  - Encapsulates *one or more threads* sharing process resources
- Human language: an individual application or program

What about thread?

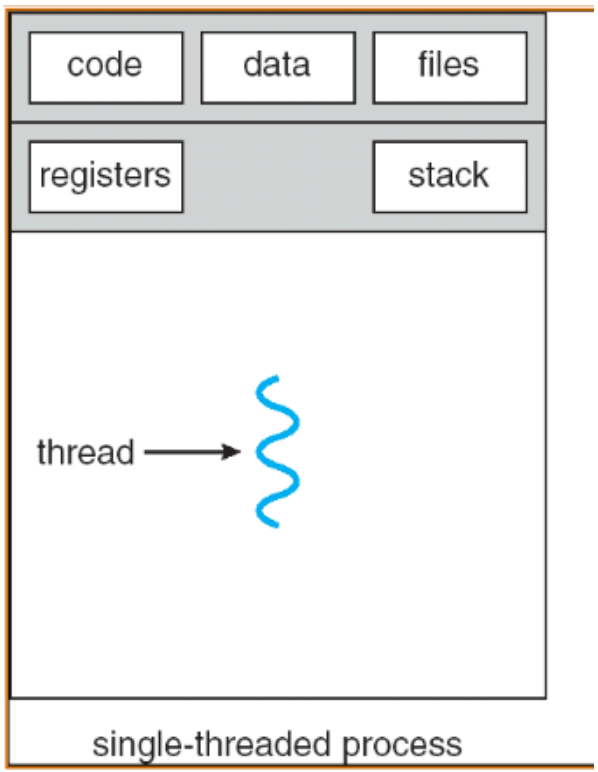
- Definition: **a single, unique execution context**

What about thread?

- Definition: **a single, unique execution context**
- In human language: a *thread* is a single execution sequence that represents a separately schedulable task

What about thread?

- Definition: **a single, unique execution context**
- In human language: a *thread* is a single execution sequence that represents a separately schedulable task



A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

- What will be the behavior here?

A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

- What will be the behavior here?
  - The class list would never be printed out

A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

- What will be the behavior here?
  - The class list would never be printed out
  - Why?

A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

- What will be the behavior here?
  - The class list would never be printed out
  - Why?
  - There is only one thread processing

A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

- What will be the behavior here?
  - The class list would never be printed out
  - Why?
  - There is only one thread processing and `ComputePI` would never finish

A silly example for single-threading

- Imagine the following program:

```
1 main() {
2     ComputePI("pi.txt");
3     PrintClassList("classlist.txt");
4 }
5
```

- What will be the behavior here?
  - The class list would never be printed out
  - Why?
  - There is only one thread processing and `ComputePI` would never finish
- Question: how to solve this issue?

Adding one thread

Adding one thread

- Version of program with Threads (loose syntax):

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- `thread_fork`: Start independent thread running given procedure

Adding one thread

- Version of program with Threads (loose syntax):

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- `thread_fork`: Start independent thread running given procedure
- What will be the behavior here?

Adding one thread

- Version of program with Threads (loose syntax):

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- `thread_fork`: Start independent thread running given procedure
- What will be the behavior here?
  - Now, you would actually see the class list

Adding one thread

- Version of program with Threads (loose syntax):

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- `thread_fork`: Start independent thread running given procedure
- What will be the behavior here?
  - Now, you would actually see the class list
  - This should behave as if there are two separate CPUs

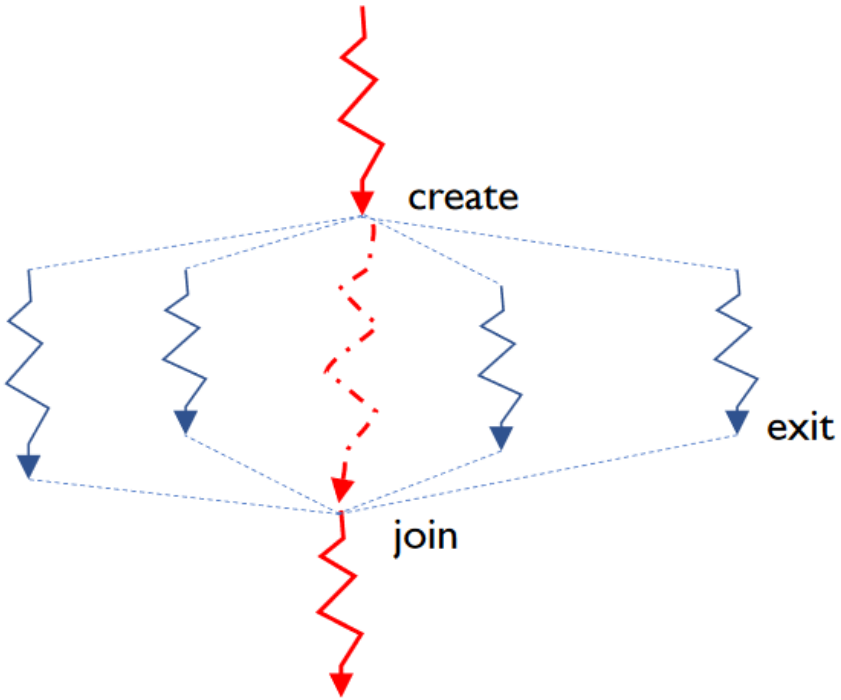
Adding one thread

- Version of program with Threads (loose syntax):

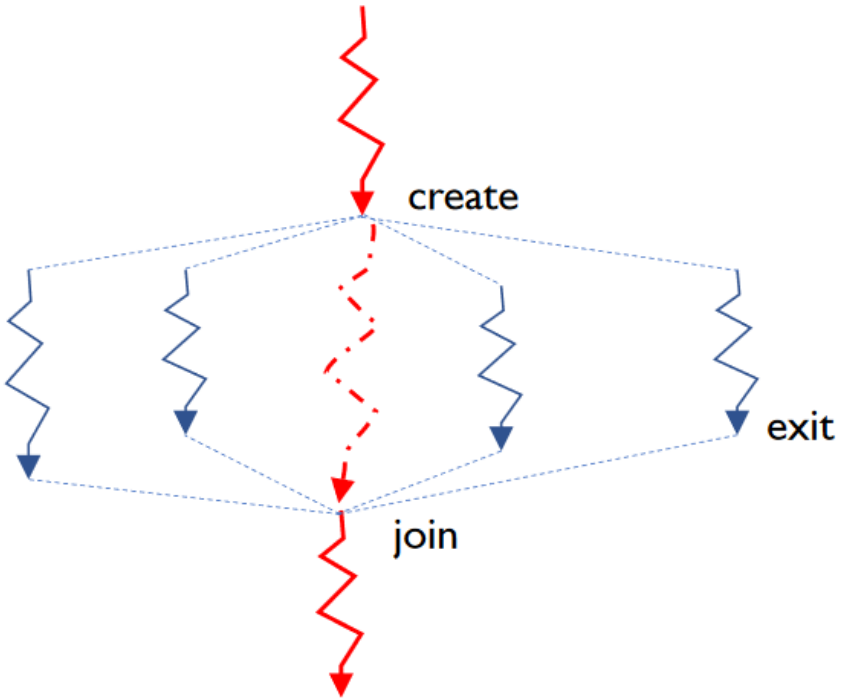
```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- `thread_fork`: Start independent thread running given procedure
- What will be the behavior here?
  - Now, you would actually see the class list
  - This should behave as if there are two separate CPUs
- Multithreading is simple!

Fork-join pattern

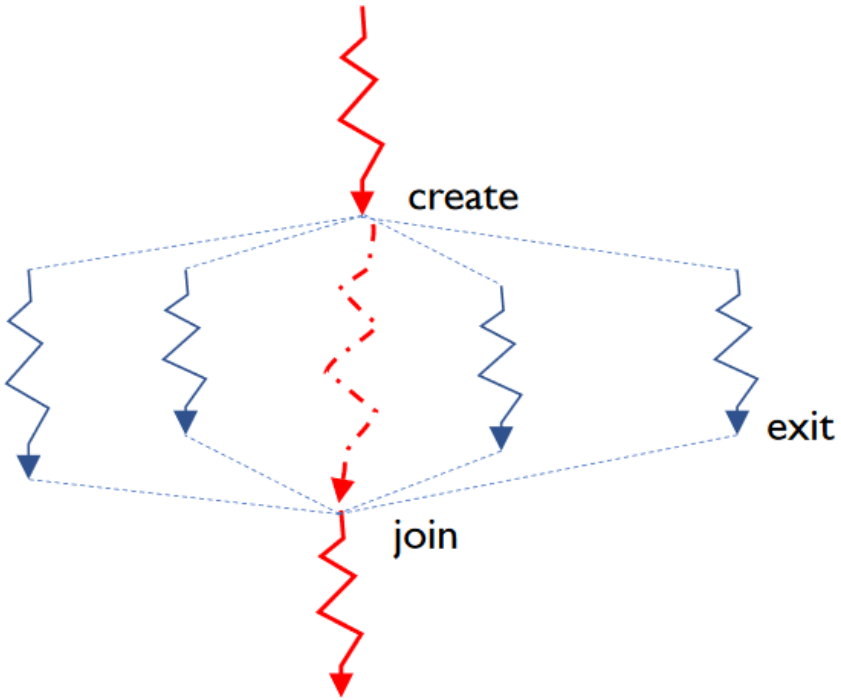


Fork-join pattern



- Main thread creates (forks) collection of sub-threads passing them args to work on, joins with them, collecting results.

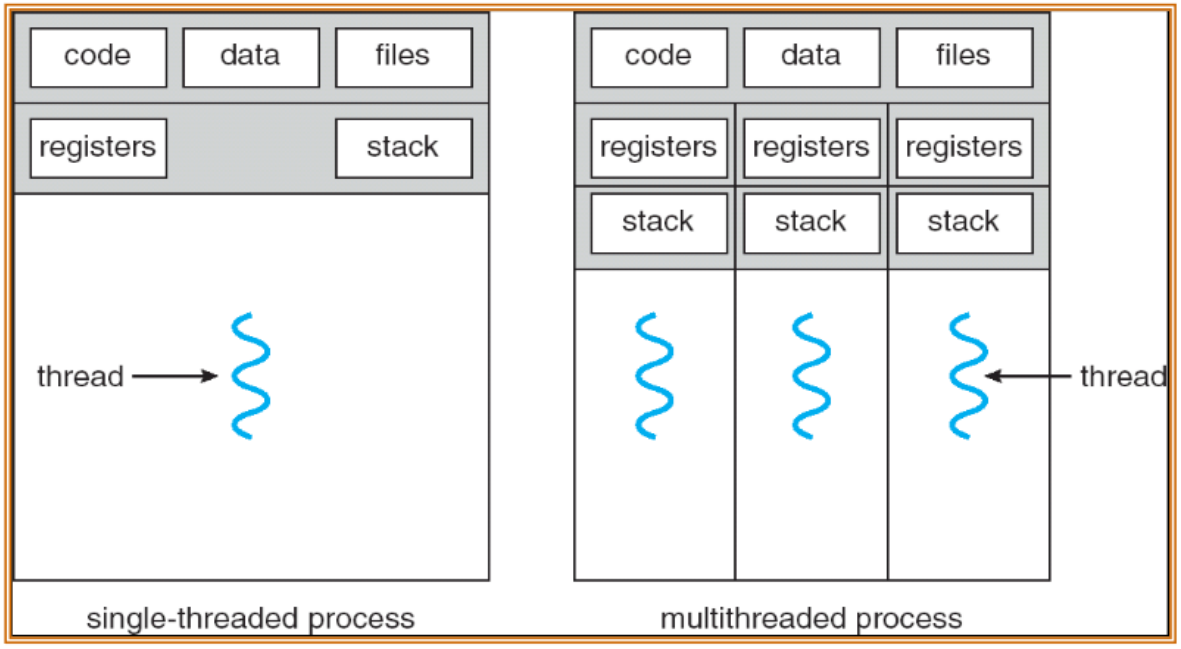
Fork-join pattern



- Main thread creates (forks) collection of sub-threads passing them args to work on, joins with them, collecting results.
- Question: How do you think those threads behaves over time?

Single- and multi-threaded processes

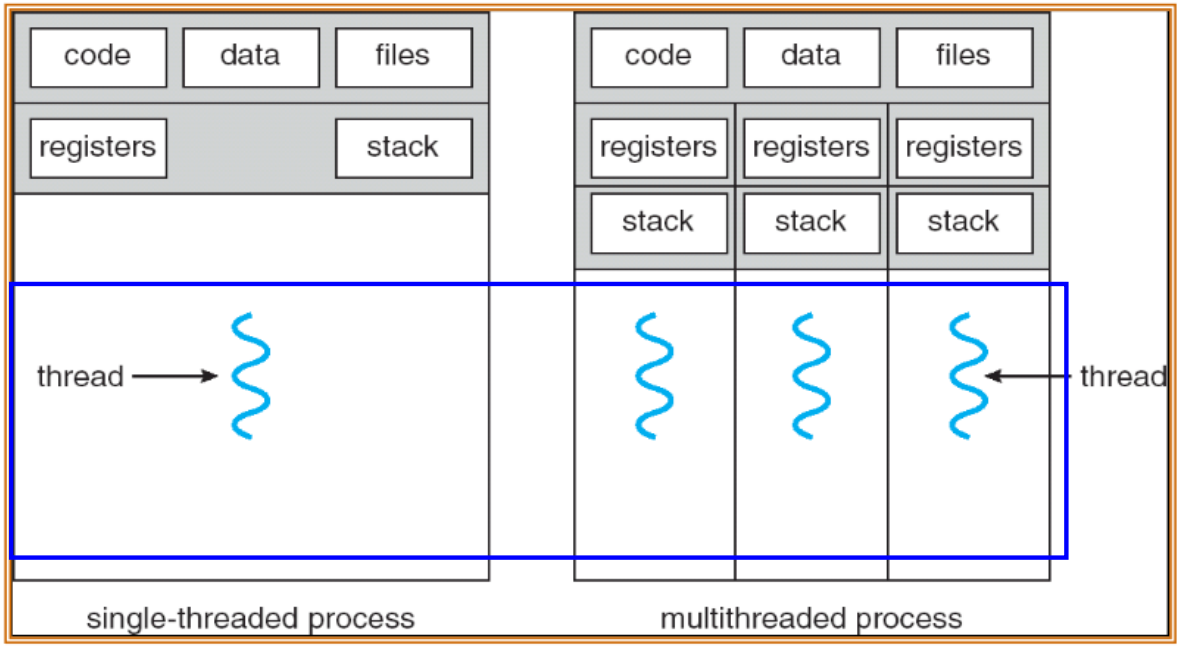
- Let's take a second look on the single- vs multi-threading



- Threads encapsulate concurrency
- Address spaces encapsulate shared resources

Single- and multi-threaded processes

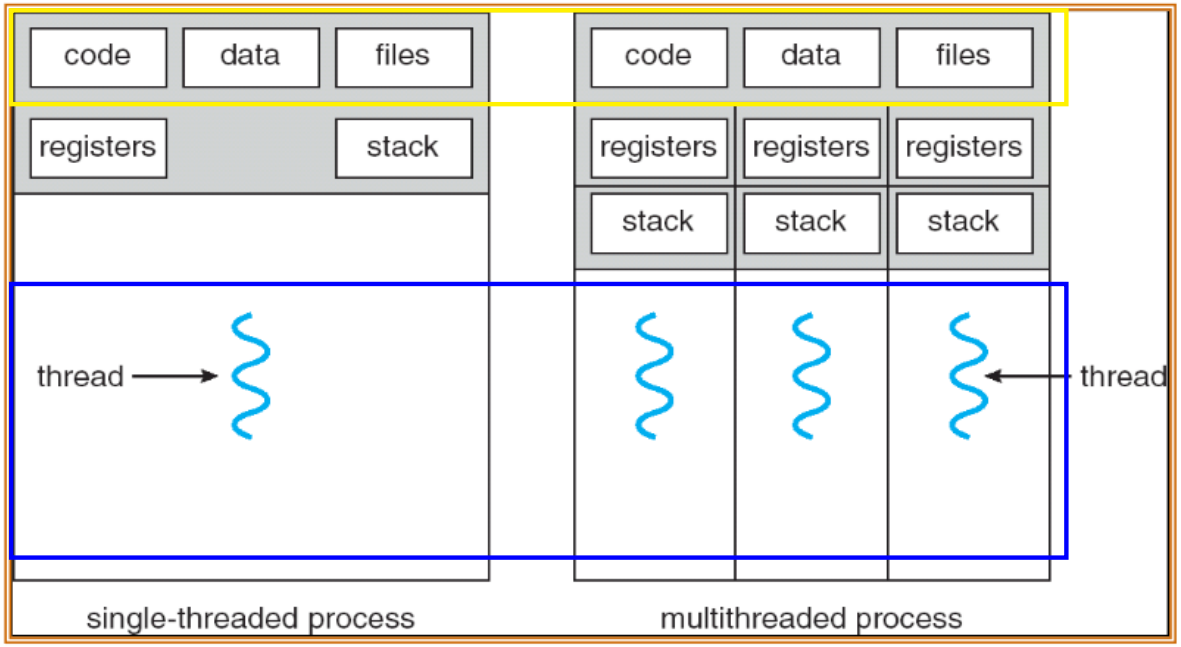
- Let's take a second look on the single- vs multi-threading



- Threads encapsulate concurrency
- Address spaces encapsulate shared resources

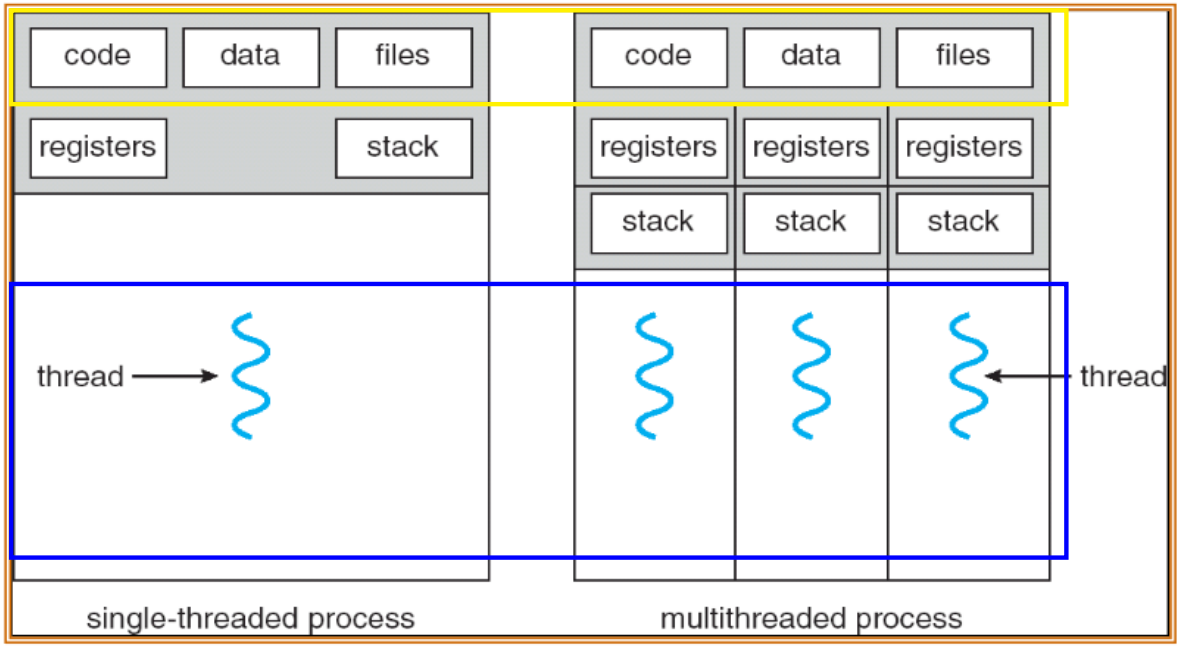
Single- and multi-threaded processes

- Let's take a second look on the single- vs multi-threading



Single- and multi-threaded processes

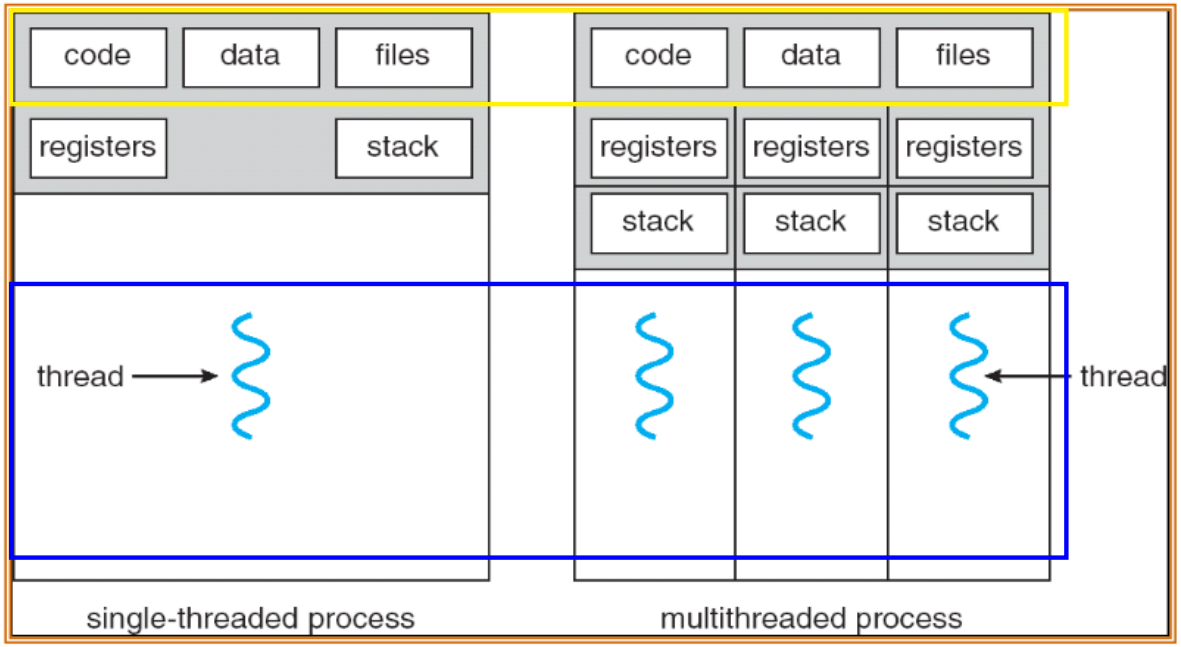
- Let's take a second look on the single- vs multi-threading



- Threads encapsulate concurrency

Single- and multi-threaded processes

- Let's take a second look on the single- vs multi-threading



- Threads encapsulate concurrency
- Address spaces encapsulate shared resources

Concurrency: interleaving

- This is the piece of codes we have just mentioned

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

Concurrency: interleaving

- This is the piece of codes we have just mentioned

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

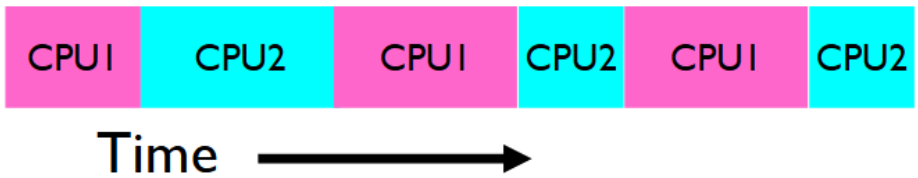
- How do these two threads works out in the time domain?

Concurrency: interleaving

- This is the piece of codes we have just mentioned

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- How do these two threads works out in the time domain?

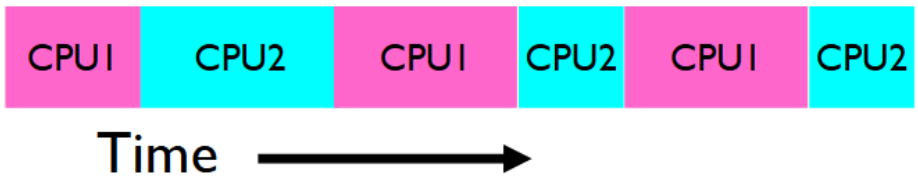


Concurrency: interleaving

- This is the piece of codes we have just mentioned

```
1 main() {
2     thread_fork(ComputePI, "pi.txt" ));
3     thread_fork(PrintClassList, "classlist.txt"));
4 }
5
```

- How do these two threads works out in the time domain?



- Does this bother you?

Programmer vs processor view

- Let's use a little bit more complicated example

Programmer vs processor view

- Let's use a little bit more complicated example
- Assume x, y, and z are all initialized by 0

Programmer's View

```
.
.
.
x = x + 1;
y = y + x;
z = x +5y;
.
.
.
```

Programmer vs processor view

- Let's use a little bit more complicated example
- Assume x, y, and z are all initialized by 0

Programmer's View

```
.
.
.
x = x + 1;
y = y + x;
z = x +5y;
.
.
.
```

- What about creating three threads to execute these three lines?

Programmer vs processor view

- Let's use a little bit more complicated example
- Assume x, y, and z are all initialized by 0

Programmer's View	Possible Execution #1
.	.
.	.
.	.
x = x + 1;	x = x + 1;
y = y + x;	y = y + x;
z = x +5y;	z = x + 5y;
.	.
.	.
.	.

- What about creating three threads to execute these three lines?

Reality: nondeterministic

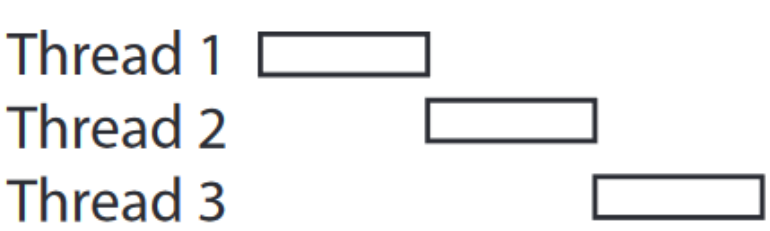
a) One execution

Reality: nondeterministic

a) One execution

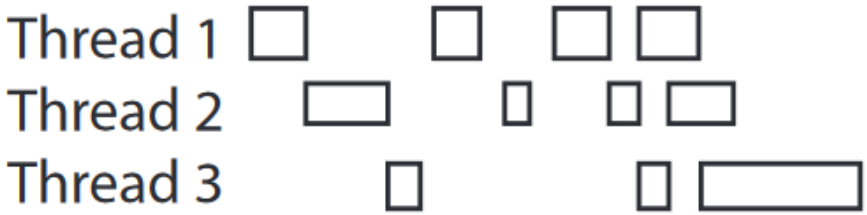
b) Another execution

Reality: nondeterministic



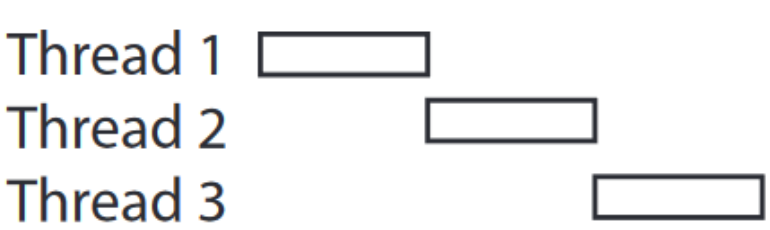
a) One execution

b) Another execution



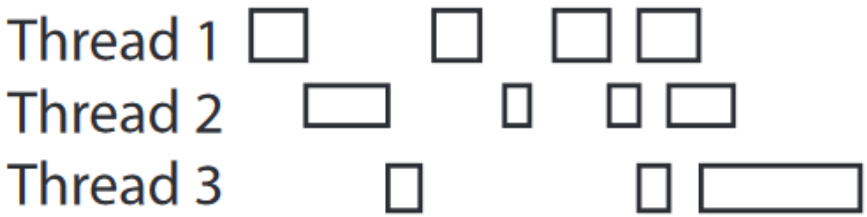
c) Another execution

Reality: nondeterministic



a) One execution

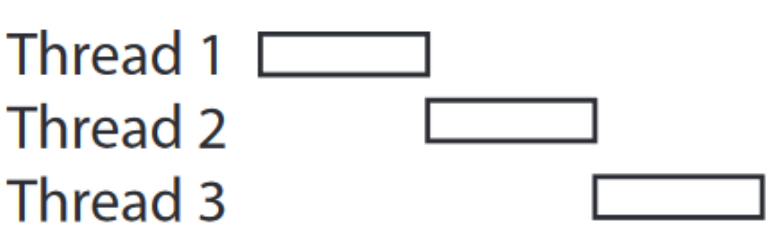
b) Another execution



c) Another execution

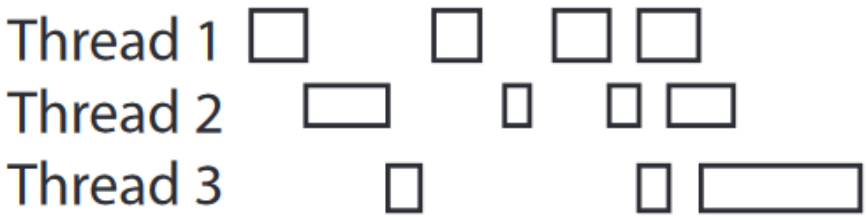
- Now this starts to bother you, right?

Reality: nondeterministic



a) One execution

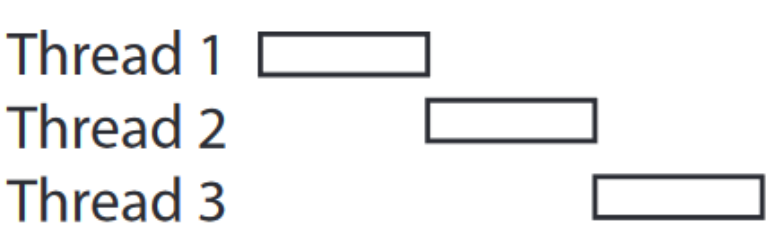
b) Another execution



c) Another execution

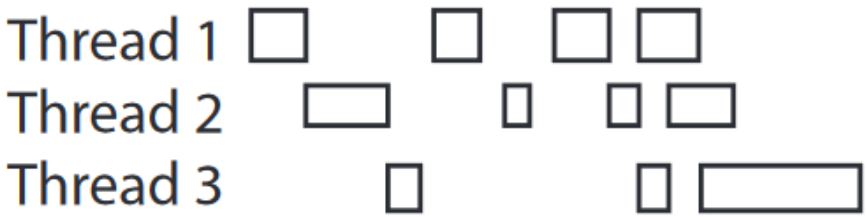
- Now this starts to bother you, right? The nondeterministics is not acceptable in programming

Reality: nondeterministic



a) One execution

b) Another execution



c) Another execution

- Now this starts to bother you, right? The nondeterministics is not acceptable in programming
- BTW, is case 3 even possible?

Example I

What are the possible values of x below?

Example I

What are the possible values of x below?

- Initially x = y = 0;
  - Thread A

```
1 x = y + 1;
2
```

  - Thread B

```
1 y = 2;
2 y = y * 2;
3
```

Example I

What are the possible values of x below?

- Initially x = y = 0;
  - Thread A

```
1 x = y + 1;
2
```

  - Thread B

```
1 y = 2;
2 y = y * 2;
3
```

- 1 or 3 or 5 (non-deterministic!)

Example I

What are the possible values of x below?

- Initially x = y = 0;
  - Thread A

```
1 x = y + 1;
2
```

  - Thread B

```
1 y = 2;
2 y = y * 2;
3
```

- 1 or 3 or 5 (non-deterministic!)
- Race Condition: Thread A races against Thread B

Real-life analogy

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Real-life analogy

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

- How can we avoid this?

Real-life analogy

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

- How can we avoid this?
- Where does the problem come from?

Atomic operation

- Definition: **An operation that runs to completion or not at all**

Atomic operation

- Definition: **An operation that runs to completion or not at all**
- Question: which of the following statements are atomic?

```
1 counter++;
2 ++counter;
3 counter = counter + 1;
4 counter = 1;
5
```

Atomic operation

- Definition: **An operation that runs to completion or not at all**
- Question: which of the following statements are atomic?

```
1 counter++;
2 ++counter;
3 counter = counter + 1;
4 counter = 1;
5
```

- The problem comes from the non-optimized compiling.

Atomic operation

- Definition: **An operation that runs to completion or not at all**
- Question: which of the following statements are atomic?

```
1 counter++;
2 ++counter;
3 counter = counter + 1;
4 counter = 1;
5
```

- The problem comes from the non-optimized compiling.

```
1 count++:
2     mov eax,[count]
3     inc eax
4     mov [count],eax
5
```

Atomic operation

- Definition: **An operation that runs to completion or not at all**
- Question: which of the following statements are atomic?

```
1 counter++;
2 ++counter;
3 counter = counter + 1;
4 counter = 1;
5
```

- The problem comes from the non-optimized compiling.

```
1 count++:
2     mov eax,[count]
3     inc eax
4     mov [count],eax
5
```

- Does the case 3 start to make more sense?

Relevant definitions

Mutual Exclusion (Mutex)

Ensuring only one thread does a particular thing at a time (one thread excludes the others)

Relevant definitions

Mutual Exclusion (Mutex)

Ensuring only one thread does a particular thing at a time (one thread excludes the others)

Critical Section

Code exactly one thread can execute at once.

Relevant definitions

Mutual Exclusion (Mutex)

Ensuring only one thread does a particular thing at a time (one thread excludes the others)

Critical Section

Code exactly one thread can execute at once.

- This is a result of mutual exclusion

Relevant definitions

Mutual Exclusion (Mutex)

Ensuring only one thread does a particular thing at a time (one thread excludes the others)

Critical Section

Code exactly one thread can execute at once.

- This is a result of mutual exclusion

Lock

An object only one thread can hold at a time.

Relevant definitions

Mutual Exclusion (Mutex)

Ensuring only one thread does a particular thing at a time (one thread excludes the others)

Critical Section

Code exactly one thread can execute at once.

- This is a result of mutual exclusion

Lock

An object only one thread can hold at a time.

- Provides mutual exclusion

Relevant definitions

Mutual Exclusion (Mutex)

Ensuring only one thread does a particular thing at a time (one thread excludes the others)

Critical Section

Code exactly one thread can execute at once.

- This is a result of mutual exclusion

Lock

An object only one thread can hold at a time.

- Provides mutual exclusion
- Take advantage of two atomic operations
  - `Lock.Acquire()` – wait until lock is free; then grab
  - `Lock.Release()` - unlock, wake up waiters

Using locks

```
MilkLock.Acquire()
if (noMilk) {
    buy milk
}
MilkLock.Release()
```

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

Using locks

```
MilkLock.Acquire()
if (noMilk) {
    buy milk
}
MilkLock.Release()
```

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

- How does this work out?

Semaphores

- Semaphores are a kind of generalized locks
- First defined by Dijkstra in late 60s

Semaphores

- Semaphores are a kind of generalized locks
- First defined by Dijkstra in late 60s

Definition

A semaphore has a non-negative integer value and supports the following two operations:

Semaphores

- Semaphores are a kind of generalized locks
- First defined by Dijkstra in late 60s

Definition

A semaphore has a non-negative integer value and supports the following two operations:

- P() or down(): atomic operation that waits for semaphore to become positive, then decrements it by 1

Semaphores

- Semaphores are a kind of generalized locks
- First defined by Dijkstra in late 60s

Definition

A semaphore has a non-negative integer value and supports the following two operations:

- P() or down(): atomic operation that waits for semaphore to become positive, then decrements it by 1
- V() or up(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any

Semaphores

- Semaphores are a kind of generalized locks
- First defined by Dijkstra in late 60s

Definition

A semaphore has a non-negative integer value and supports the following two operations:

- P() or down(): atomic operation that waits for semaphore to become positive, then decrements it by 1
- V() or up(): an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
- P()stands for "proberen" (to test) and V()stands for "verhogen" (to increment) in Dutch

Two important semaphore patterns

Lock like

Also called a "binary lock"

```
1 // initial value of semaphore = 1;
2 semaphore.down();
3 // Critical section goes here
4 semaphore.up();
5
```

Two important semaphore patterns

Lock like

Also called a "binary lock"

```
1 // initial value of semaphore = 1;
2 semaphore.down();
3 // Critical section goes here
4 semaphore.up();
5
```

Resource management

Signaling other threads

Two important semaphore patterns

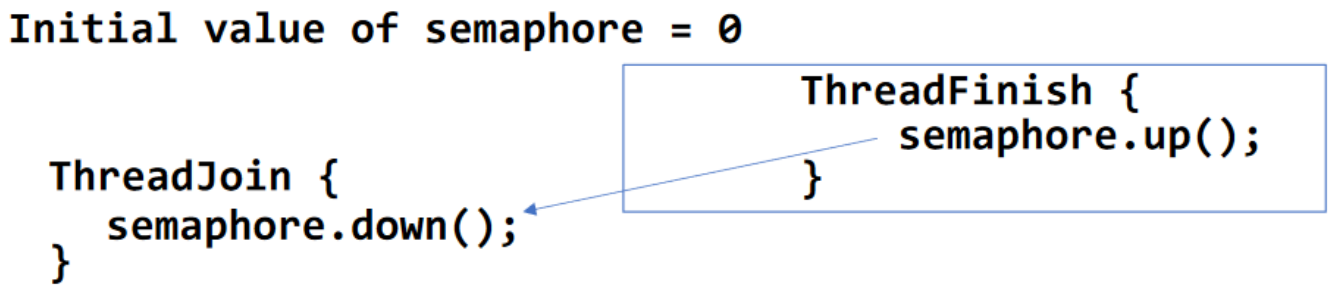
Lock like

Also called a "binary lock"

```
1 // initial value of semaphore = 1;
2 semaphore.down();
3 // Critical section goes here
4 semaphore.up();
5
```

Resource management

Signaling other threads



QThread

- `QThread` is the central class in Qt to run code in a different thread
- It's a `QObject` subclass
  - Not copiable/moveable
  - Has signals to notify when the thread starts/finishes
- It is meant to *manage* a thread

QThread usage

- To create a new thread executing some code, subclass `QThread` and reimplement `run()`
- Then create an instance of the subclass and call `start()`
- Threads have priorities that you can specify as an optional parameter to `start()`, or change with `setPriority()`

QThread usage


```
1 class MyThread : public QThread {
2 private:
3     void run() override {
4         // code to run in the new thread
5     }
6 };
```

```
1 MyThread *thread = new MyThread;
2 thread->start(); // starts a new thread which calls run()
3 // ...
4 thread->wait(); // waits for the thread to finish
```

QThread usage

- The thread will stop running when (some time after) returning from `run()`
- `QThread::isRunning()` and `QThread::isFinished()` provide information about the execution of the thread
- You can also connect to the `QThread::started()` and `QThread::finished()` signals
- A thread can stop its execution temporarily by calling one of the `QThread::sleep()` functions
  - Generally a *bad idea*, being event driven (or polling) is much much better
- You can wait for a `QThread` to finish by calling `wait()` on it
  - Optionally passing a maximum number of milliseconds to wait

QThread caveats

From a non-main thread you cannot:

- Perform any GUI operation
  - Including, but not limited to: using any `QWidget` / Qt Quick / `QPixmap` APIs
  - Using `QImage`, `QPainter`, etc. (i.e. "client side") is OK
  - Using OpenGL may be OK: check at runtime `QOpenGLContext::supportsThreadedOpenGL()`
- Call `Q(Core|Gui)Application::exec()`

Ensuring destruction of QObjects

- Create them on `QThread::run()` stack
- Connect their `QObject::deleteLater()` slot to the `QThread::finished()` signal
  - Yes, this will work
- Move them out of the thread

QThread usage

There are two basic strategies of running code in a separate thread with QThread:

- Without an event loop
- With an event loop

QThread usage without an event loop

- Subclass `QThread` and override `QThread::run()`
- Create an instance and start the new thread via `QThread::start()`

QThread usage without an event loop

- Subclass `QThread` and override `QThread::run()`
- Create an instance and start the new thread via `QThread::start()`

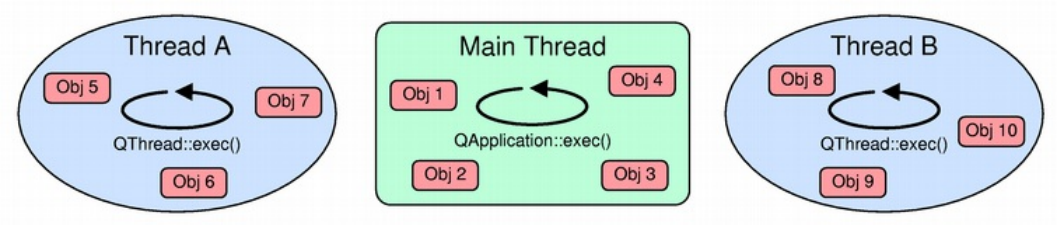
```
1 class MyThread : public QThread {
2 private:
3     void run() override {
4         loadFilesFromDisk();
5         doCalculations();
6         saveResults();
7     }
8 };
```

```
1 auto thread = new MyThread;
2 thread->start();
3 // some time later...
4 thread->wait();
```

QThread usage with an event loop

KDAB

- An event loop is necessary when dealing with timers, networking, *queued connections*, and so on.
- Qt supports per-thread event loops:



- Each thread-local event loop delivers events for the QObjects living in that thread.

QThread usage with an event loop

- We can start a thread-local event loop by calling `QThread::exec()` from within `run()`:

```
1  class MyThread : public QThread {
2  private:
3      void run() override {
4          auto socket = new QTcpSocket;
5          socket->connectToHost(...);
6
7          exec(); // run the event loop
8
9          // cleanup
10     }
11 };
```

- `QThread::quit()` or `QThread::exit()` will quit the event loop

- We can also use `QEventLoop`
  - Or manual calls to `QCoreApplication::processEvents()`

What is the single most important thing about threads?

Synchronization

Qt has a complete set of cross-platform, low-level APIs for dealing with synchronization:

- `QMutex` is a mutex class (recursive and non-recursive)
- `QSemaphore` is a semaphore
- `QWaitCondition` is a condition variable
- `QReadWriteLock` is a *shared mutex*
- `QAtomicInt` is an atomic `int`
- `QAtomicPointer<T>` is an atomic pointer to T

There are also RAII classes for lock management, such as `QMutexLocker`, `QReadLocker` and so on.

Mutex Example


```
1  class Thread : public QThread
2  {
3      bool m_cancel;
4  public:
5      explicit Thread(QObject *parent = nullptr)
6        : QThread(parent), m_cancel(false) {}
7
8      void cancel()   // called by GUI
9      {
10         m_cancel = true;
11     }
12
13 private:
14     bool isCanceled() const   // called by run()
15     {
16         return m_cancel;
17     }
18
19     void run() override { // reimplemented from QThread
20         while (!isCanceled())
21             doSomething();
22     }
23 };
```

Mutex Example (cont'd)

KDAB

```
1  class Thread : public QThread
2  {
3      mutable QMutex m_mutex; // protects m_cancel
4      bool m_cancel;
5  public:
6      explicit Thread(QObject *parent = nullptr)
7        : QThread(parent), m_cancel(false) {}
8
9      void cancel() { // called by GUI
10         const QMutexLocker locker(&m_mutex);
11         m_cancel = true;
12     }
13
14 private:
15     bool isCanceled() const { // called by run()
16         const QMutexLocker locker(&m_mutex);
17         return m_cancel;
18     }
19
20     void run() override { // reimplemented from QThread
21         while (!isCanceled())
22             doSomething();
23     }
24 };
```

Reentrancy definitions

A function is:

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all

Reentrancy definitions

A function is:

- **Thread safe**: if it's safe for it to be invoked at the same time, from multiple threads, on the same data, without synchronization
- **Reentrant**: if it's safe for it to be invoked at the same time, from multiple threads, on different data; otherwise it requires external synchronization
- **Non-reentrant** (thread unsafe): if it cannot be invoked from more than one thread at all

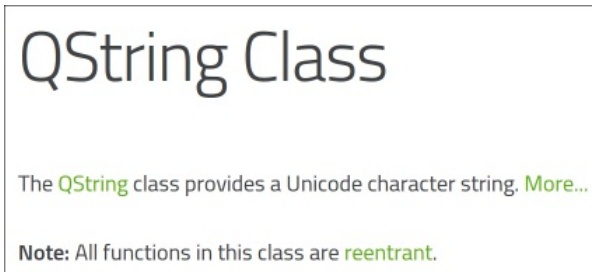
For classes, the above definitions apply to non-static member functions when invoked on the same instance. (In other words, considering the `this` pointer as an argument.)

Examples

- **Thread safe**:
  - `QMutex`
  - `QObject::connect()`
  - `QCoreApplication::postEvent()`
- **Reentrant**:
  - `QString`
  - `QVector`
  - `QImage`
  - value classes in general
- **Non-reentrant**:
  - `QWidget` (including all of its subclasses)
  - `QQuickItem`
  - `QPixmap`
  - in general, GUI classes are usable only from the main thread

Thread safety for Qt classes/functions

The documentation of each class / function in Qt has notes about its thread safety:



Unless otherwise specified, classes and functions are **non-reentrant**.

QObject: thread affinity

What about QObject?

QObject: thread affinity

What about QObject?

- `QObject` itself is thread-aware.
- Every `QObject` instance holds a reference to the thread it was created into (`QObject::thread()`)
  - We say that the object *lives in*, or *has affinity with* that thread
- We can move an instance to another thread by calling `QObject::moveToThread(QThread *)`

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself
- You must delete all `QObjects` living in a certain `QThread` before destroying the `QThread` instance

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself
- You must delete all `QObjects` living in a certain `QThread` before destroying the `QThread` instance
- You can only call `moveToThread()` on a `QObject` from the same thread the object has affinity with (`moveToThread()` is non-reentrant)

QObject: thread safety

`QObject` is **reentrant** according to the documentation, however:

- Event-based classes are non-reentrant (timers, sockets, ...)
- The event dispatching for a given `QObject` happens in the thread it has affinity with
- All the `QObjects` in the same parent/child tree must have the same thread affinity
  - Notably, you can't parent `QObjects` created in a thread to the `QThread` object itself
- You must delete all `QObjects` living in a certain `QThread` before destroying the `QThread` instance
- You can only call `moveToThread()` on a `QObject` from the same thread the object has affinity with (`moveToThread()` is non-reentrant)

In practice: **it's easier to think of `QObject` as non-reentrant**, as it will make you avoid many mistakes.

QObject: queued connections

- If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?

QObject: queued connections

- If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?
- Qt has a solution: **cross-thread signals and slots**

QObject: queued connections

- If `QObject` is non-reentrant, how can I communicate with a `QObject` living in another thread?
- Qt has a solution: **cross-thread signals and slots**
- You can emit a signal from one thread, and have the slot invoked by another thread
  - Not just any thread: the thread the receiver object is living in

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!
- Also, `qRegisterMetaType()` is required for the argument types passed

QObject: queued connections

- If the receiver object of a connection lives in a different thread than the thread the signal was emitted in, the slot invocation will be **queued**.
- Under the hood: a metacall event is posted in the receiver's thread's event queue
  - The event will then get dispatched to the object *by the right thread*
  - Handling such metacall events means invoking the slot
- This requires that the receiver object is living in a thread with a running event loop!
- Also, `qRegisterMetaType()` is required for the argument types passed
- We can force any connection to be queued:

```
connect(sender, &Sender::signal, receiver, &Receiver::slot, Qt::QueuedConnection);
```

QObject: queued connections example

KDAB

```
1  class MyThread : public QThread {
2      Producer *m_producer;
3  public:
4      explicit MyThread(Producer *p, QObject *parent = nullptr)
5          : QThread(parent), m_producer(p) {}
6
7      void run() override {
8          Consumer consumer;
9          connect(m_producer, &Producer::unitProduced,
10                 &consumer, &Consumer::consume);
11         exec();
12     }
13 };
14
15 // in main thread:
16 auto producer = new Producer;
17 auto thread = new MyThread(producer);
18 thread->start();
19 producer->startProduction();
20
21 // Producer::unitProduced gets emitted some time later from the main thread,
22 // Consumer::consume gets run in the secondary thread
```

QObject: queued connections example (2)


```
1  // Same as before, but without the race
2
3  auto producer = new Producer;
4  auto consumer = new Consumer;
5  auto thread = new QThread;
6
7  connect(m_producer, &Producer::unitProduced,
8          consumer, &Consumer::consume);
9  connect(thread, &QThread::finished,
10         consumer, &QObject::deleteLater);
11
12 consumer->moveToThread(thread);
13
14 thread->start();
15
16 // Producer::unitProduced gets emitted some time later from the main thread,
17 // Consumer::consume gets run in the secondary thread
```

QObject: queued connections example (3)

KDAB

```
1  class MyThread : public QThread {
2  public:
3      explicit MyThread(QObject *parent = nullptr)
4          : QThread(parent) {}
5
6  private:
7      void run() override {
8          emit mySignal();
9      }
10
11 signals:
12     void mySignal();
13 };
14
15 // in main thread:
16 auto thread = new MyThread;
17 connect(thread, &MyThread::mySignal, receiver, &Receiver::someSlot);
18 thread->start();
```

QObject: queued connections example (3)


```
1  class MyThread : public QThread {
2  public:
3      explicit MyThread(QObject *parent = nullptr)
4          : QThread(parent) {}
5
6  private:
7      void run() override {
8          emit mySignal();
9      }
10
11 signals:
12     void mySignal();
13 };
14
15 // in main thread:
16 auto thread = new MyThread;
17 connect(thread, &MyThread::mySignal, receiver, &Receiver::someSlot);
18 thread->start();
```

- It is perfectly OK to add signals to `QThread`
- The connection is queued: the thread that emits the signal is not the thread the receiver has affinity with
- `someSlot()` gets invoked by the main thread's event loop

QObject: queued connections example (4)


```
1  class MyThread : public QThread {
2      Socket *m_socket;
3  public:
4      explicit MyThread(QObject *parent = nullptr)
5          : QThread(parent) {}
6
7  private:
8      void run() override {
9          m_socket = new Socket;
10         connect(m_socket, &Socket::connected, this, &MyThread::onConnected);
11         m_socket->connectToHost(...);
12         exec();
13     }
14
15 private slots:
16     void onConnected() { qDebug() << "Data received:" << m_socket->data(); }
17 };
```

QObject: queued connections example (4)


```
1  class MyThread : public QThread {
2      Socket *m_socket;
3  public:
4      explicit MyThread(QObject *parent = nullptr)
5          : QThread(parent) {}
6
7  private:
8      void run() override {
9          m_socket = new Socket;
10         connect(m_socket, &Socket::connected, this, &MyThread::onConnected);
11         m_socket->connectToHost(...);
12         exec();
13     }
14
15 private slots:
16     void onConnected() { qDebug() << "Data received:" << m_socket->data(); }
17 };
```

- `QThread` is a `QObject` and as such has its own thread affinity (it's the thread that created the `MyThread` instance, *not itself!*)
- The connection is queued: the thread that emits the signal is not the thread the receiver has affinity with
  - *This is not what we wanted!*
- Huge recommendation: **avoid adding slots to QThread**