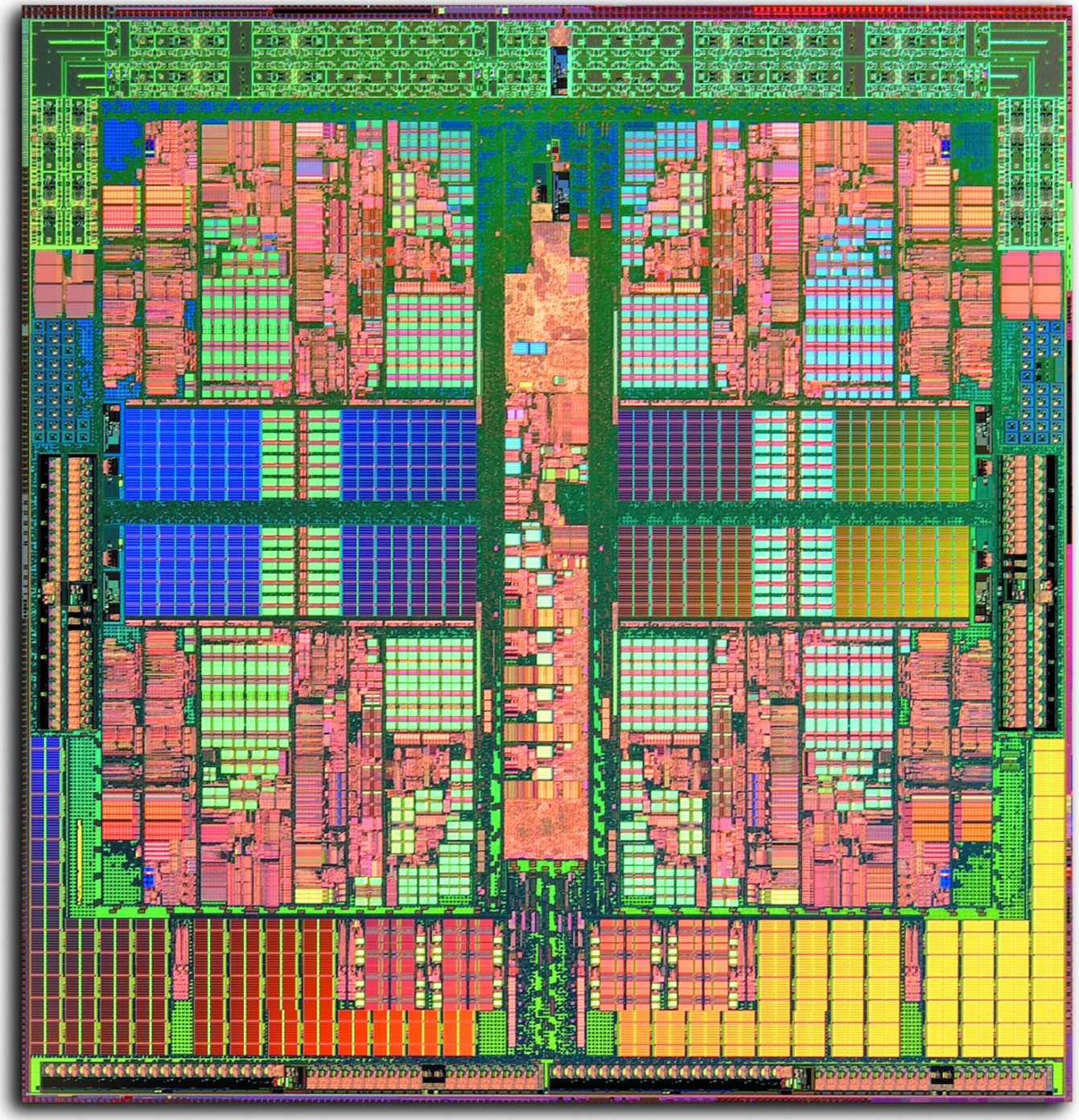
Leow Wee Kheng
CS3249 User Interface Development

Multithreading


Single Player
Battle.net
Local Area Network
Options
Credits
Quit
Processing, please wait...
1.24.1.6374



My processor has 4 cores!

Can it run faster!?

Process

- A process is a running program managed by OS.
- A process consists of
  - executable code in memory
  - static and dynamic data
  - execution context
    - program counter
    - contents of registers
    - stack pointer
    - memory management information,
    - etc.

- Memory protection
  - A process has it own address space or virtual memory space.
  - Cannot access address spaces of other processes.
  - OS maps virtual memory to physical memory.

- Multiprogramming or multitasking
  - A process is given a fixed time to run.
  - When time is up, OS suspends the process, and switches context to another waiting process.

- Processes are heavy-weight
  - Context switching is costly.

Multithreading

- Thread
  - A light-weight sequence of a running program.
  - All threads of a process use the same address space.
  - Context switching is cheaper.

- Multithreading
  - Run multiple threads at the same time.
  - Different threads can run in different processors.
  - Some programming languages support multithreading
    - Ada, Java
  - In Unix/Linux, use pthread to implement multithreading.

Multithreading in Qt

- Qt application can run in multiple threads.
  - All window operations must run in the main thread.
  - Non-window operations can run in secondary threads.

- Qt has a `QThread` class for easy creation of threads.

Thread Creation

```
// Thread.h

class Thread: public QThread
{
    Q_OBJECT

public:
    Thread();
    void stop();

protected:
    void run();

private:
    volatile bool stopped;
};
```

inherits `QThread`

Redefines `QThread::run()`

`volatile` tells compiler not to optimise; variable is accessed from other threads.

```
// Thread.cpp

Thread::Thread()
{
    stopped = false;
}

void Thread::run()
{
    while (!stopped)
        cout << ".";

    stopped = false;
    cout << "\n";
}

void Thread::stop()
{
    stopped = true;
}
```

Initialise

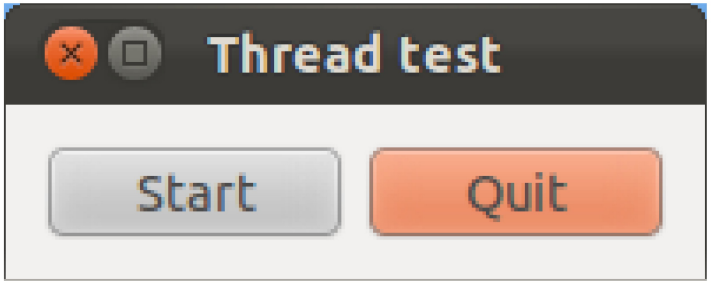
Keep running until `stopped` is set to `true`.

Reset

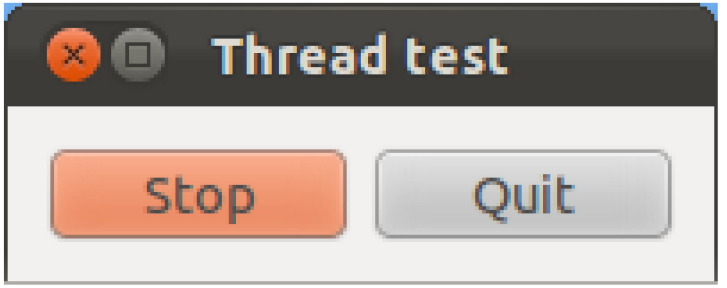
Someone will call it to set `stopped` to `true`.

- Use dialog to illustrate thread creation and stopping.

click start to start thread



click stop to stop thread



```
// ThreadDialog.h

class ThreadDialog : public QDialog
{
    Q_OBJECT

public:
    ThreadDialog(QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void startOrStopThread();

private:
    Thread thread;
    QPushButton *threadButton;
    QPushButton *quitButton;
};
```

```
// ThreadDialog.cpp


ThreadDialog::ThreadDialog(QWidget *parent):
    QDialog(parent)
{
    // Create widgets.

    threadButton = new QPushButton(tr("Start"));
    connect(threadButton, SIGNAL(clicked()),
        this, SLOT(startOrStopThread()));

    quitButton = new QPushButton(tr("Quit"));
    quitButton->setDefault(true);
    connect(quitButton, SIGNAL(clicked()),
        this, SLOT(close()));
```

```
    // Layout widgets.

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(threadButton);
    layout->addWidget(quitButton);
    setLayout(layout);
    setWindowTitle(tr("Thread test"));
}



void ThreadDialog::closeEvent(QCloseEvent *event)
{
    thread.stop();   // Stop the thread.
    thread.wait();   // Wait for thread to finish.
    event->accept();
}
```

```
void ThreadDialog::startOrStopThread()
{
    if (thread.isRunning())
    {
        thread.stop();
        threadButton->setText(tr("Start"));
    }
    else
    {
        thread.start();
        threadButton->setText(tr("Stop"));
    }
}
```

- `isRunning()` and `start()` are inherited from `QThread`.

Staying Responsive

- One way is to dispatch events regularly.

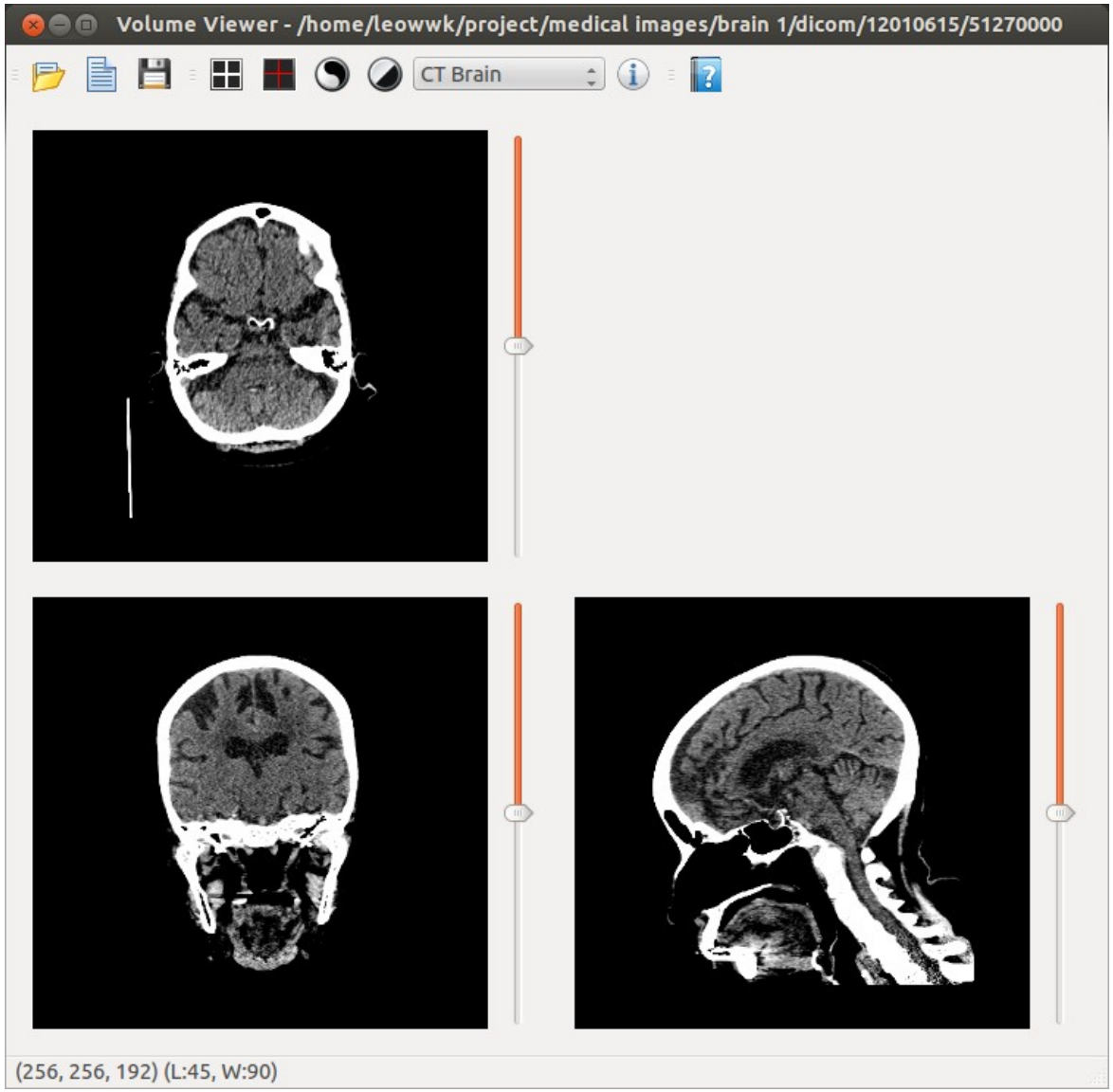
```
long-operation()
{
    for each iteration
    {
        do work of one iteration;
        qApp->processEvents();  // dispatch events
    }
}
```

  - This method is impossible if
    - you don't have source code of `long-operation`, or
    - you don't want to change the source code.

- Alternative: run long operation in another thread.

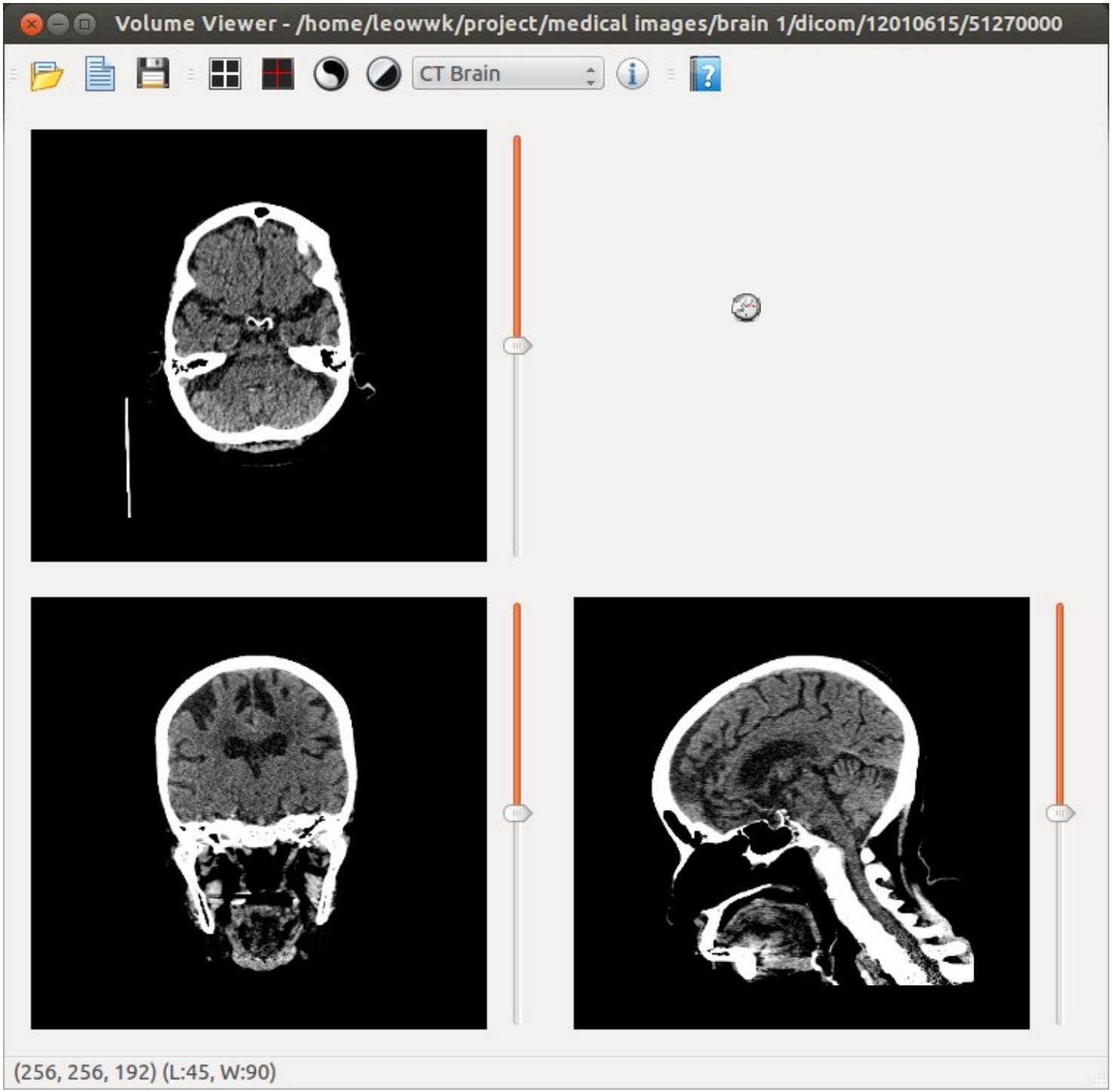
Example: Volume Image Viewer

- 3D image is large, takes a while to load.
- Want to show progress while loading.



```
// program fragment

void VolumeViewer::loadImage(const QString &dirName)
{
    VTKDICOMReader *reader = VTKDICOMReader::New();
    char *dir = dirName.toAscii().data();
    reader->SetDirectoryName(dir);
    reader->Update();

    // display image ...
}
```

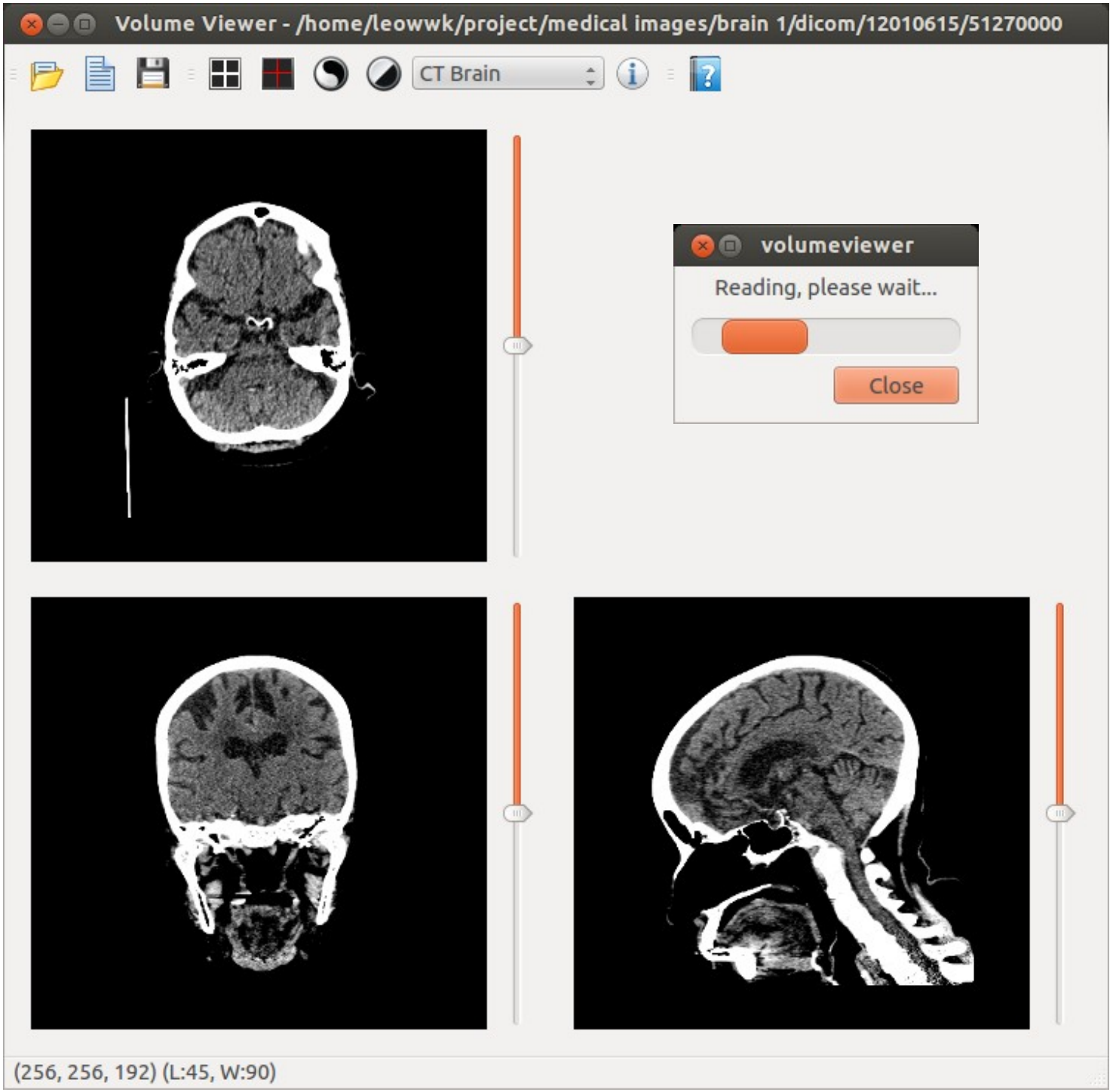
```
// Method 1: use wait cursor

void VolumeViewer::loadImage(const QString &dirName)
{
    VTKDICOMReader *reader = VTKDICOMReader::New();
    char *dir = dirName.toAscii().data();
    reader->SetDirectoryName(dir);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    reader->Update();
    QApplication::restoreOverrideCursor();

    // display image ...
}
```


Volume Viewer - /home/leowwk/project/medical images/brain 1/dicom/12010615/51270000
CT Brain
(256, 256, 192) (L:45, W:90)

```
// Method 2: Multithreading

class ReadThread: public QThread
{
    Q_OBJECT

public:
    ReadThread(vtkDICOMImageReader *rd);
    bool isDone();

protected:
    void run();

private:
    volatile bool done;
    vtkDICOMImageReader *reader;
};
```

```
ReadThread::ReadThread(vtkDICOMImageReader *rd)
{
    reader = rd;  // Keep it for run function.
    done = false; // Init.
}


bool ReadThread::isDone()
{
    return done;
}


void ReadThread::run()
{
    reader->Update();  // Read image.
    done = true;
}
```

```
void VolumeViewer::loadImage(const QString &dirName)
{
    VTKDICOMReader *reader = VTKDICOMReader::New();
    char *dir = dirName.toAscii().data();
    reader->SetDirectoryName(dir);

    QProgressDialog progress(
        QString("Reading, please wait..."),
        QString("Close"), 0, 0);
    progress.show();
    qApp->processEvents();  // Optional

    ReadThread thread(rd);
    thread.start();
    while(!thread.isDone())
        qApp->processEvents();
    thread.quit();  // Stop thread's event loop.
    progress.hide();
    // display image ...
}
```


Volume Viewer - /home/leowwk/project/medical images/brain 1/dicom/12010615/51270000
CT Brain
volumeviewer
Reading, please wait...
Close
(256, 256, 192) (L:45, W:90)

- In general, can allow certain interactions in main thread instead of just displaying progress dialog.

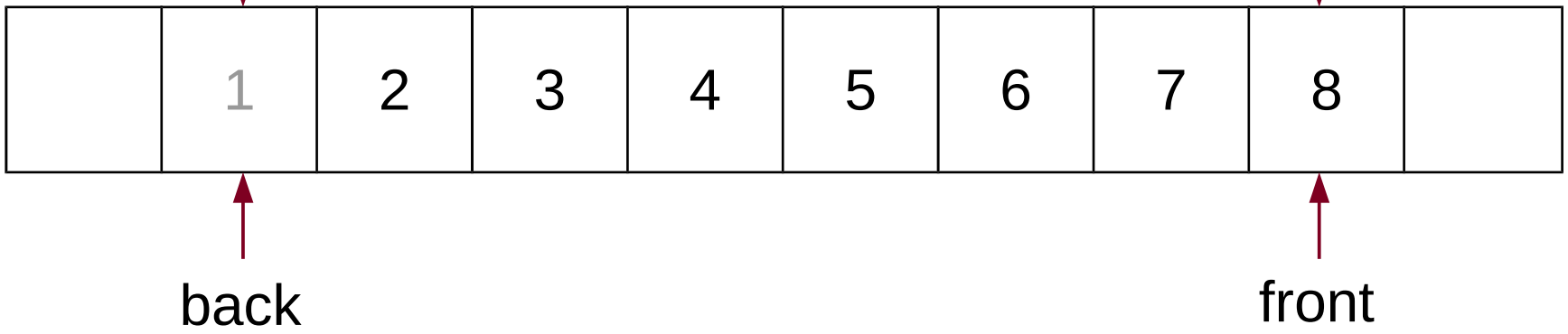
- Caution:
  - Operations in different threads must not conflict.
    - Example: While doing long calculation on data, must disable GUI's read, edit functions.

      ```
      openFileAction->setEnabled(false);
      ```

  - If different threads refer to same data (shared data), must synchronise threads.
    - Otherwise, may get corrupted data or unstable execution.

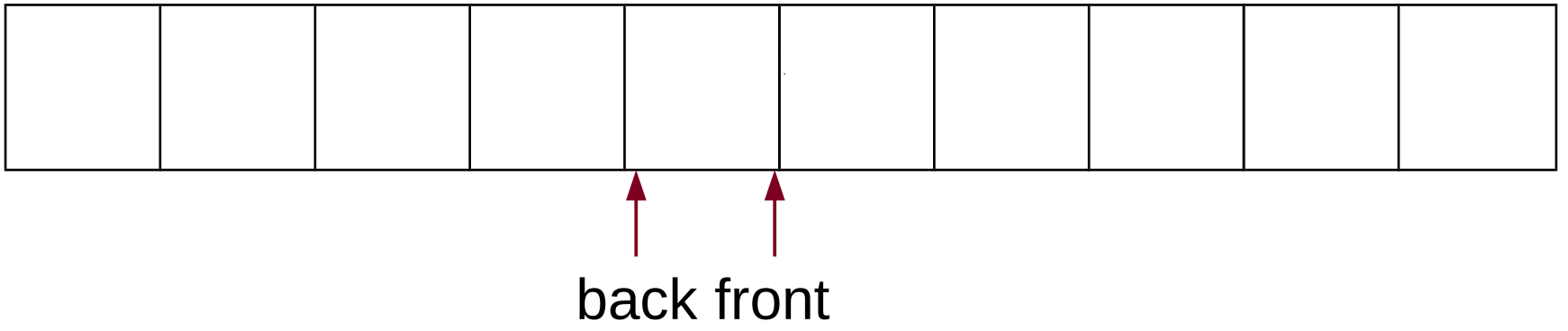
Thread Synchronisation

- Producer-consumer problem:
  - Producer and consumer share common buffer.
  - Producer adds to front of buffer until it is full.
  - Consumer removes from back of buffer until it is empty.

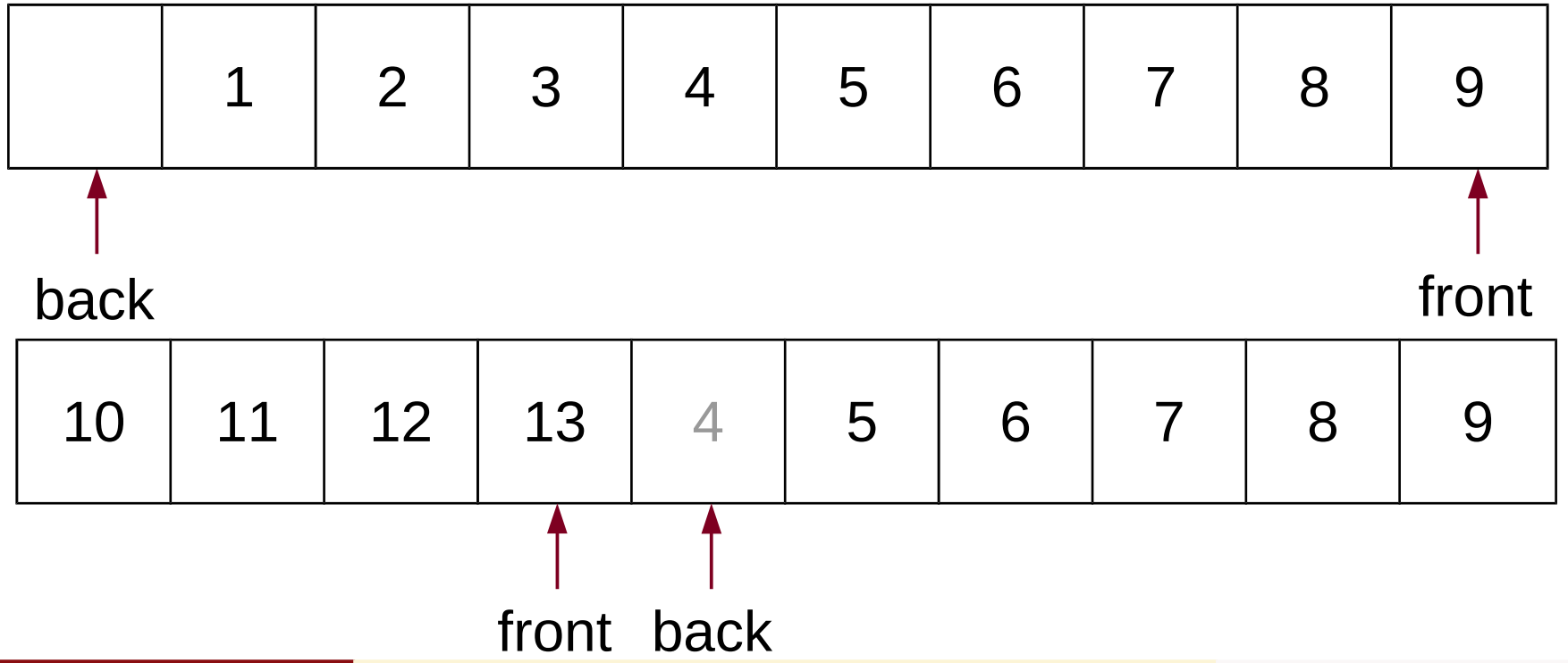


  - Increment pointer before read / write data.

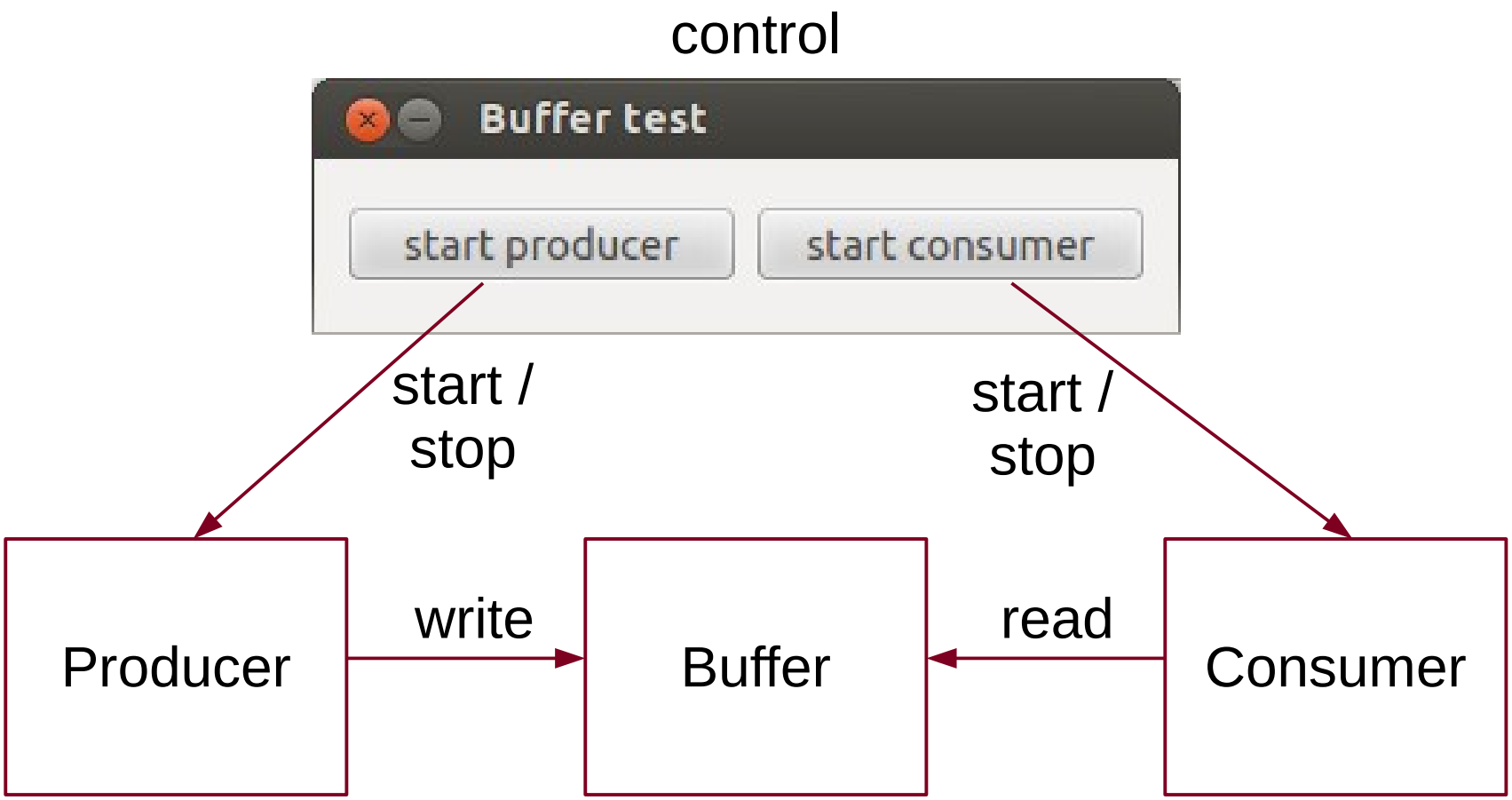
- Empty buffer



- Full buffer



Program Structure



```
// Buffer.h

class Buffer
{
    friend class Producer;
    friend class Consumer;

public:
    Buffer() {front = back = 0;}  // Init to empty buffer.

protected:
    enum {size = 10};
    int data[size];
    int front;  // Points to front of buffer.
    int back;   // Points to back of buffer.
};
```

```
// Producer.h

class Producer: public QThread
{
    Q_OBJECT

public:
    Producer(Buffer *b);
    void stop();

protected:
    void run();

private:
    Buffer *buffer;  // Points to shared buffer.
    volatile bool stopped;
};
```

```
// Producer.cpp

Producer::Producer(Buffer *b)
{
    buffer = b;  // Keeps pointer to shared buffer.
    stopped = false;
}


void Producer::stop()
{
    stopped = true;
}
```

```
void Producer::run()
{
    static int data = 0;

    while (!stopped)
    {
        if (buffer->front != buffer->back - 1 and
            !(buffer->front == buffer->size - 1 and
              buffer->back == 0))  // Not full
        {
            ++(buffer->front);  // Increment front.
            if (buffer->front == buffer->size)
                buffer->front = 0;  // Wrap around.

            msleep(500);  // Simulate put to sleep by OS.
```

```
            ++data;
            // Add data to buffer.
            buffer->data[buffer->front] = data;

            cout << "p." << data << " " << flush;
        }
    }

    stopped = false;
}
```

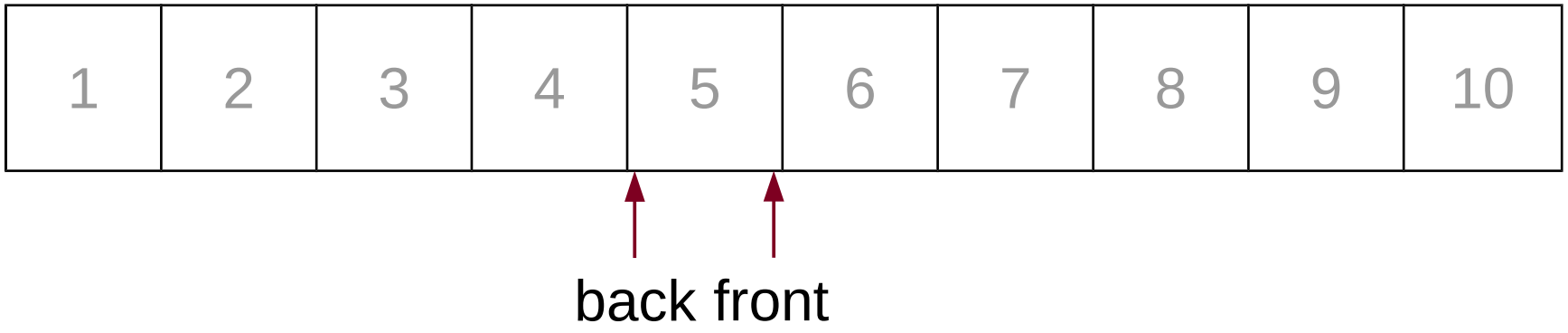
```
// Consumer.h

class Consumer: public QThread
{
    Q_OBJECT

public:
    Consumer(Buffer *b);
    void stop();

protected:
    void run();

private:
    Buffer *buffer;  // Points to shared buffer.
    volatile bool stopped;
};
```

```
// Consumer.cpp

Consumer::Consumer(Buffer *b)
{
    buffer = b;  // Keeps pointer to shared buffer.
    stopped = false;
}


void Consumer::stop()
{
    stopped = true;
}
```

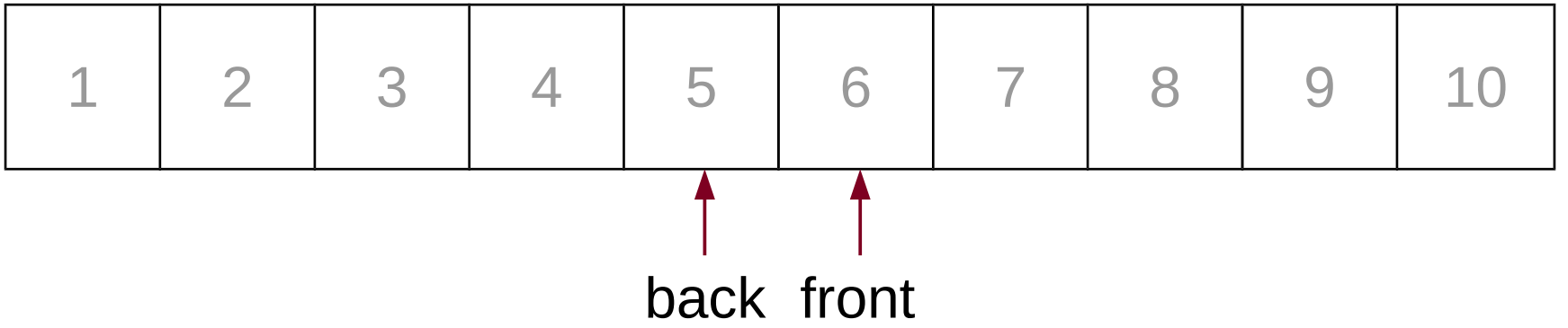
```
void Consumer::run()
{
    while (!stopped)
    {
        if (buffer->back != buffer->front) // Not empty.
        {
            ++(buffer->back);  // Increment back.
            if (buffer->back == buffer->size)
                buffer->back = 0;  // Wrap around.

            msleep(100);  // Simulate put to sleep by OS.

            // Get data.
            int data = buffer->data[buffer->back];
            cout << "c." << data << " " << flush;
        }
    }

    stopped = false;
}
```

```
// main.cpp

int main(int argc, char **args)
{
    QApplication app(argc, args);

    Buffer *buffer = new Buffer;
    Producer *producer = new Producer(buffer);
    Consumer *consumer = new Consumer(buffer);
    Control *control = new Control(producer, consumer);
    control->show();

    return app.exec();
}
```

- Buffer overrun can happen.
  - Example: Empty buffer with previously read data (grey).

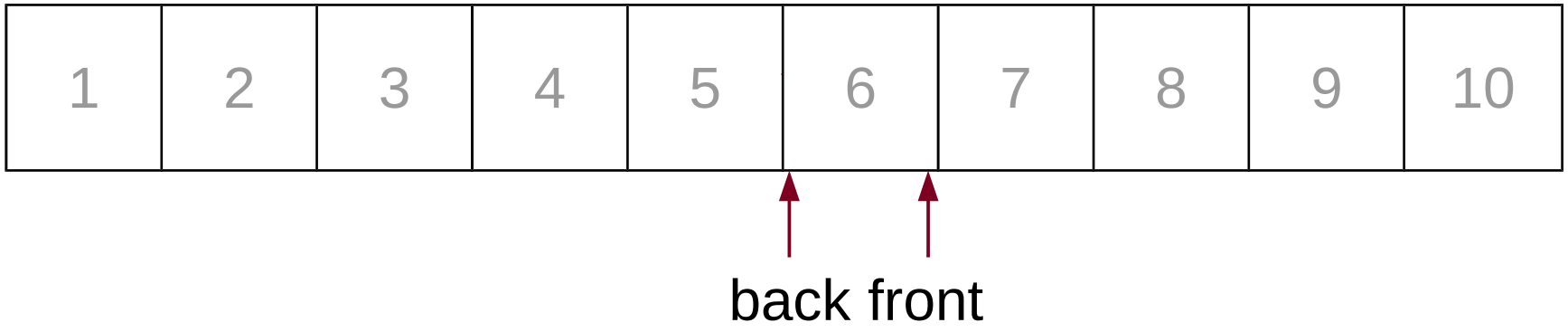


  - Producer increments pointer,
    then it is put to sleep by OS before writing data to buffer.

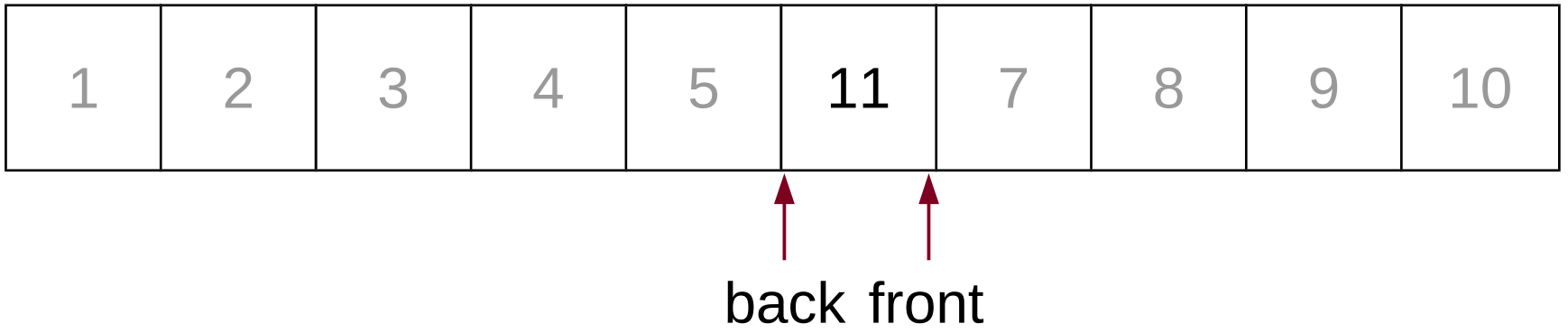


  - Consumer sees non-empty buffer.

- Consumer increments pointer and reads 6 (garbage).

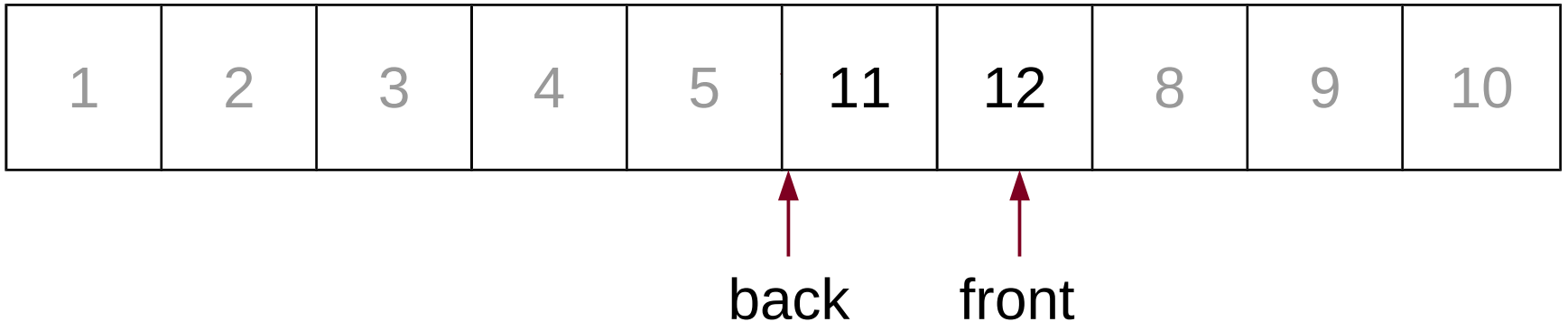


- Producer wakes up and writes new data 11.

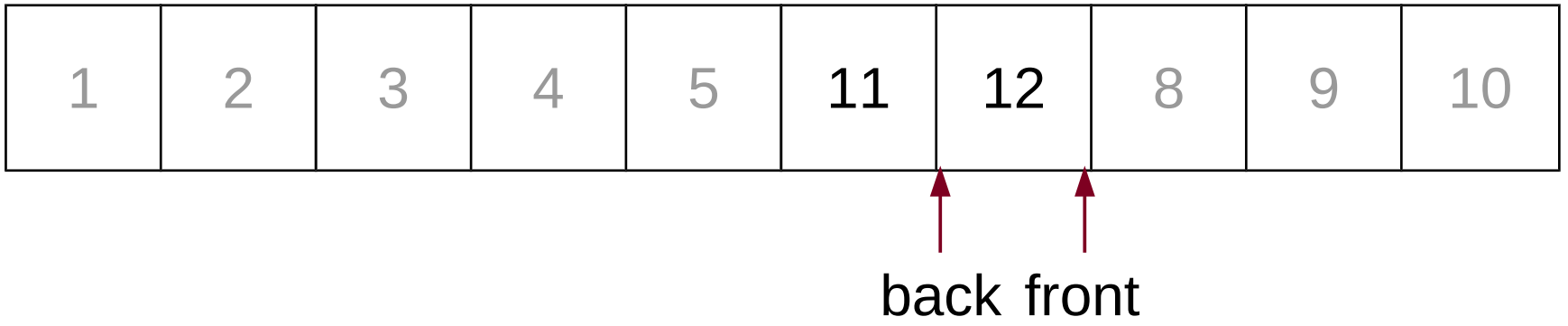


- Consumers sees empty buffer, never reads 11.

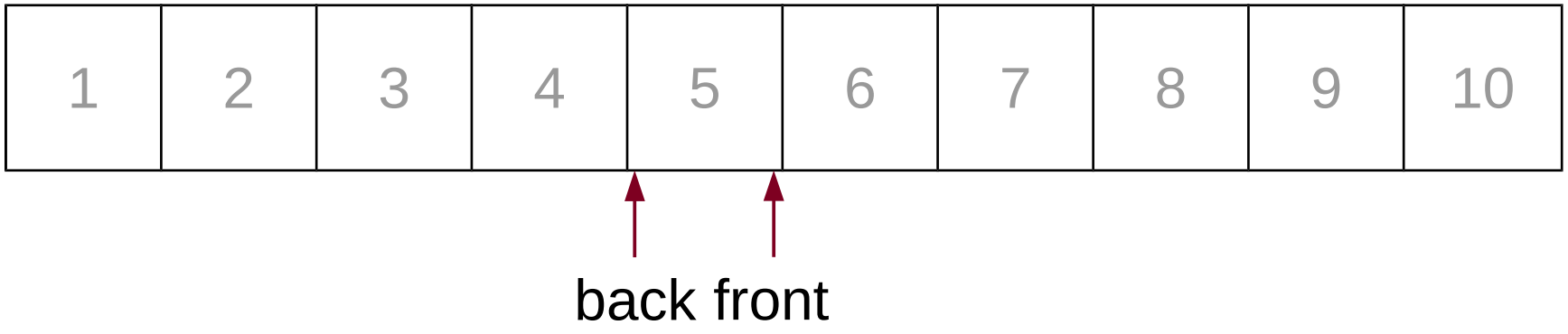
- Producer increments pointer, writes 12.



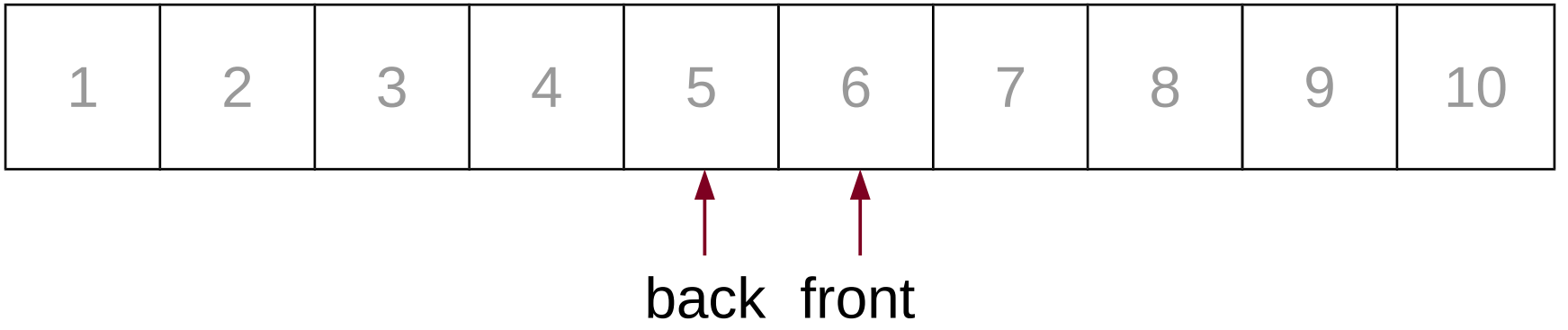
- Consumer increments pointer, reads 12, misses 11.



- If producer always sleeps after incrementing pointer...
  - Example: Empty buffer with previously read data (grey).

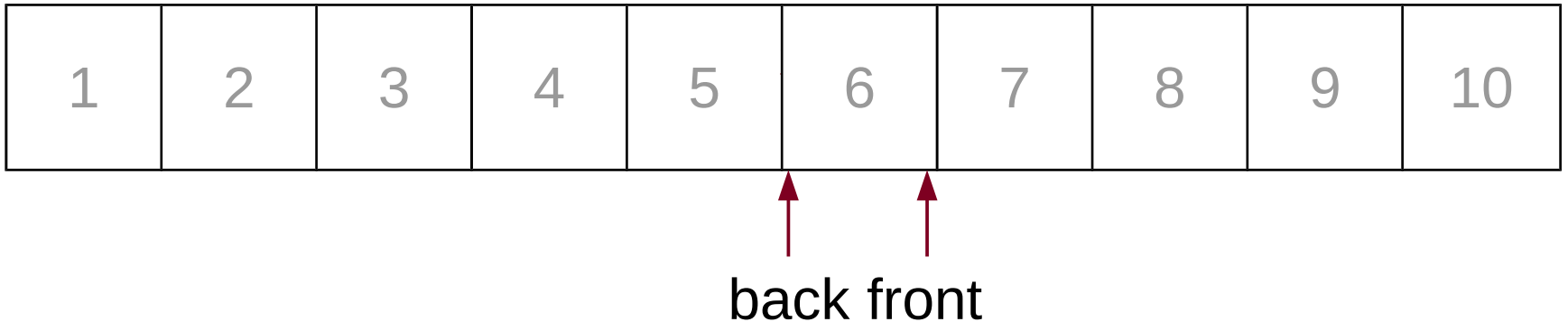


  - Producer increments pointer,
    then it is put to sleep by OS before writing data to buffer.

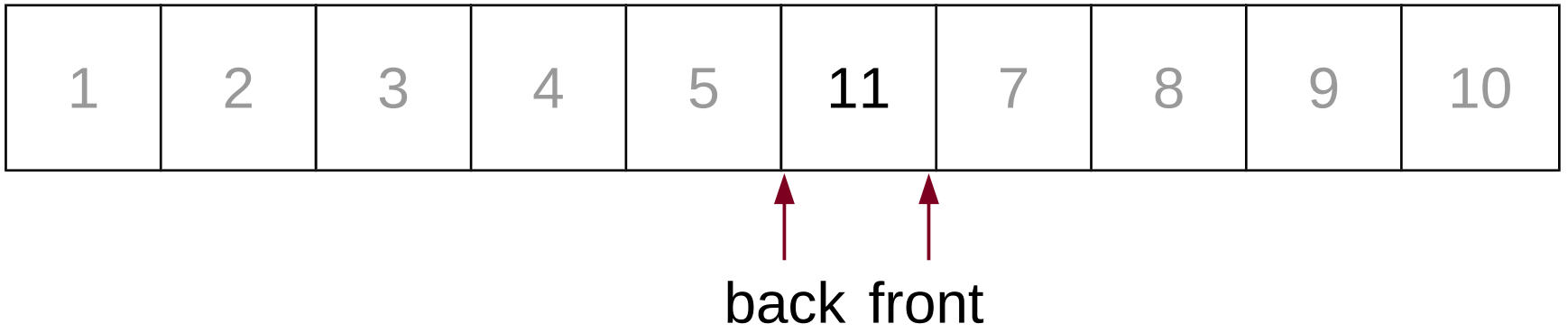


  - Consumer sees non-empty buffer.

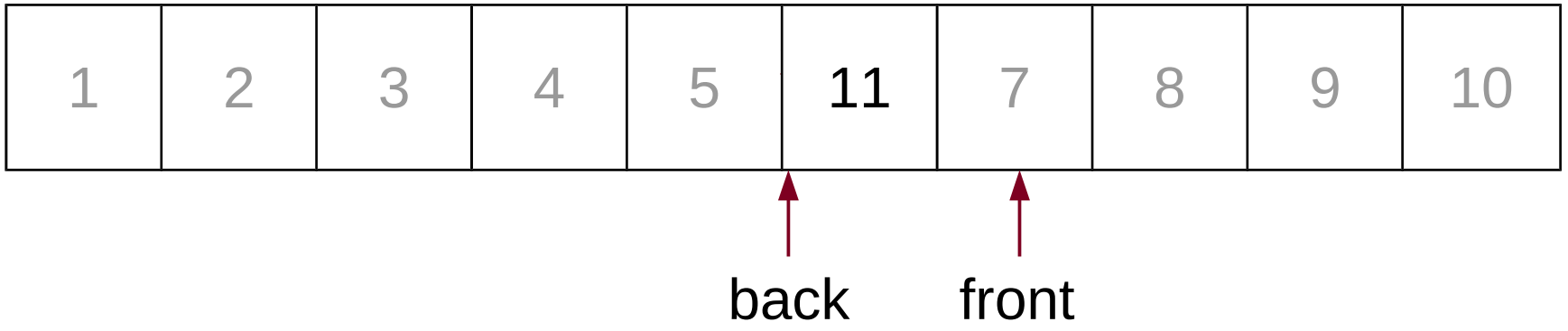
- Consumer increments pointer and reads 6 (garbage).



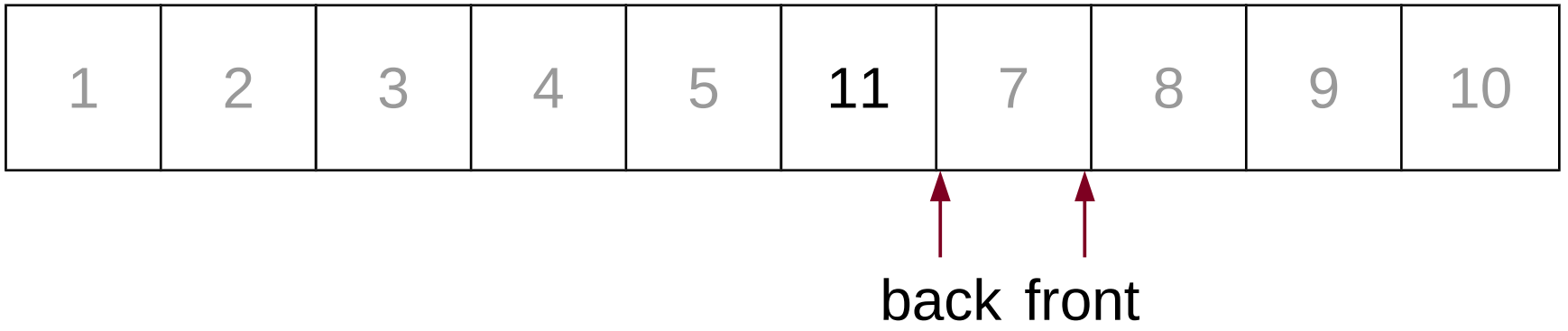
- Producer wakes up and writes new data 11.



- Producer increments pointer, then goes to sleep.



- Consumer increments pointer, reads 7 (garbage).



Unsafe execution

```
leowwk-~/course/cs3249/examples/buffer1$ buffer1-unsafe-p500-c250
p.1 p.2 p.3 p.4 p.5 p.6 c.1 c.2 p.7 c.3 c.4 p.8 c.5 c.6 p.9 c.7 c.8 p.10 c.9 c.1
0 p.11 c.11 c.2 p.12 c.3 p.13 c.4 p.14 c.5 p.15 c.6 p.16 c.7 p.17 c.8 p.18 c.9 p
.19 c.10 p.20 c.11 p.21 c.12 p.22 c.13 p.23 c.14 p.24 c.15 p.25 c.16 p.26 c.17 p
.27 c.18 p.28 c.19 p.29 c.20 p.30
```

buffer overrun

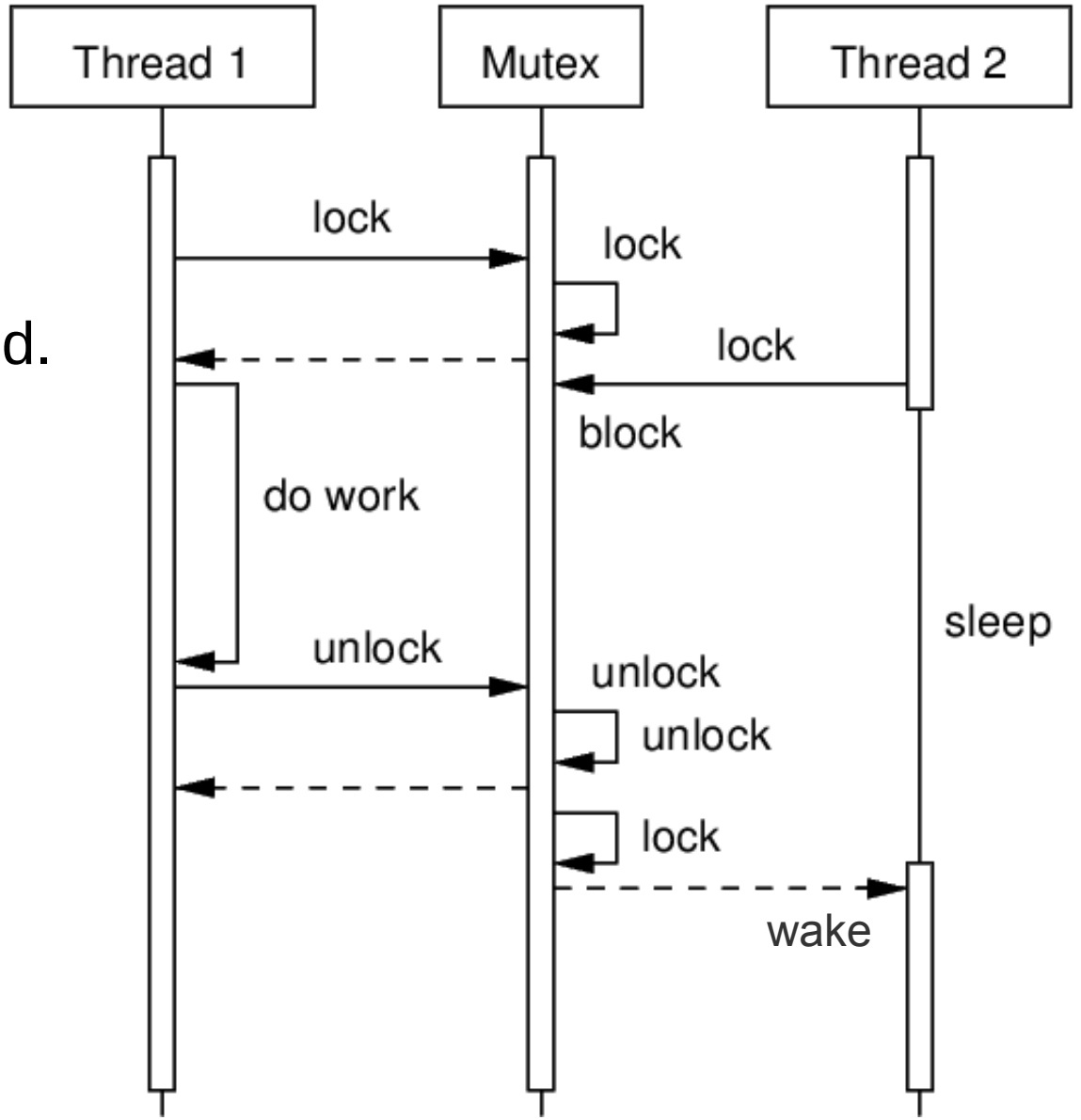
consumer sees empty buffer before producer writes new data

consumer reads garbage before producer writes new data

- Consumer sleeps less...

```
leowwk-~/course/cs3249/examples/buffer1$ buffer1-unsafe-p500-c100
p.1 p.2 p.3 p.4 p.5 c.1 c.2 c.3 c.4 p.6 c.5 c.6 c.1072693248 p.7 c.0 p.8 c.0 p.9
 c.-8923160 p.10 c.1 p.11 c.2 p.12 c.3 p.13 c.4 p.14 c.5 p.15 c.6 p.16 c.7 p.17
c.8 p.18 c.9 p.19 c.10 p.20 c.11 p.21 c.12 p.22 c.13 p.23 c.14 p.24 c.15 p.25 c.
16 p.26 c.17 p.27 c.18 p.28 c.19 p.29
```

- Qt provides `QMutex` to protect critical region.
  - mutex = mutual exclusion
  - If a thread tries to lock a locked mutex, it is put to sleep until mutex is unlocked.



```
// Buffer.h

class Buffer
{
    friend class Producer;
    friend class Consumer;

public:
    Buffer() {front = back = 0;}  // Init to empty buffer.

protected:
    enum {size = 10};
    int data[size];
    int front;  // Points to front of buffer.
    int back;   // Points to back of buffer.
    QMutex mutex;
};
```

```
void Producer::run()
{
    static int count = 0;

    while (!stopped)
    {
        buffer->mutex.lock();
        if (buffer->front != buffer->back - 1 and
            !(buffer->front == buffer->BufferSize - 1 and
              buffer->back == 0))  // Not full
        {
            ...
        }
        buffer->mutex.unlock();
    }

    stopped = false;
}
```

```
void Consumer::run()
{
    while (!stopped)
    {
        buffer->mutex.lock();
        if (buffer->back != buffer->front) // Not empty.
        {
            ...
        }
        buffer->mutex.unlock();
    }

    stopped = false;
}
```

Safe execution

```
leowwk-~/course/cs3249/examples/buffer1$ buffer1-safe-p500-c100
p.1 p.2 p.3 p.4 p.5 p.6 p.7 p.8 p.9 c.1 c.2 c.3 c.4 c.5 c.6 c.7 c.8 c.9 p.10 p.1
1 p.12 p.13 p.14 p.15 p.16 p.17 p.18 c.10 c.11 c.12 c.13 c.14 c.15 c.16 c.17 c.1
8 p.19 p.20 p.21 p.22 p.23 p.24 p.25 p.26 p.27 c.19 c.20 c.21 c.22 c.23 c.24 c.2
5 c.26 c.27 p.28 p.29 p.30 p.31 p.32 p.33 p.34 p.35 p.36 c.28 c.29 c.30 c.31 c.3
2 c.33 c.34 c.35 c.36
```

- Producer and consumer rotate to access whole buffer.
- Reason:
  - Producer keeps locking mutex immediately after unlocking.
  - Consumer can't lock mutex until buffer is full.
- No benefit with multithreading.

- In practice, after unlocking mutex,
  producer does something before locking mutex again.

```
void Producer::run()
{
    while (!stopped)
    {
        spend time to produce data;
        buffer->mutex.lock();
        if (buffer is not full)
        {
            write to buffer;
        }
        buffer->mutex.unlock();
    }

    stopped = false;
}
```

- Safe execution
  - Producer and consumer run concurrently. More efficient.

```
leowwk-~/course/cs3249/examples/buffer1$ buffer1-safe-p250x2-c100
p.1 p.2 p.3 p.4 p.5 p.6 c.1 c.2 c.3 c.4 c.5 c.6 p.7 c.7 p.8 c.8 p.9 c.9 p.10 c.1
0 p.11 c.11 p.12 c.12 p.13 c.13 p.14 c.14 p.15 c.15 p.16 c.16 p.17 c.17 p.18 c.1
8 p.19 c.19 p.20 c.20 p.21 c.21 p.22 c.22 p.23 c.23 p.24 c.24 p.25 c.25 p.26 c.2
6 p.27 c.27 p.28 c.28 p.29 c.29
```

- Question:
  - In the previous examples, producer and consumer increment pointers before writing and read data.
  - What if they increment pointers **after** writing and reading? (exercise)

- Caution
  - Locking and unlocking mutex in complex functions or functions that use C++ exceptions can be error-prone: mutex is locked but not unlocked.
  - `QMutexLocker`'s destructor automatically unlocks mutex.

- Instead of

```
void Consumer::run()
{
    while (!stopped)
    {
        buffer->mutex.lock();
        ... // read data
        buffer->mutex.unlock();
    }
}
```

do

```
void Consumer::run()
{
    while (!stopped)
    {
        QMutexLocker locker(&(buffer->mutex));
        ... // read data
    }
}
```

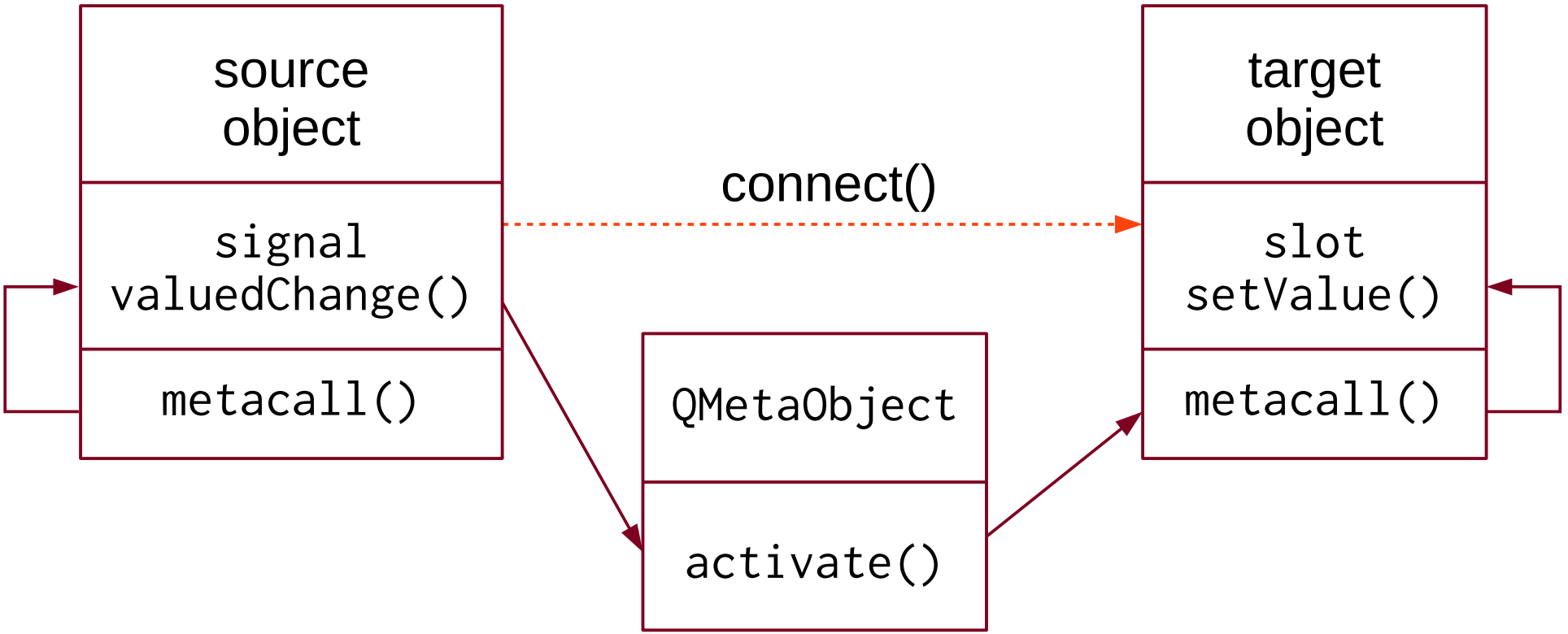
Other Protection Mechanisms

- `QReadWriteLock`
  - Only one object can acquire write lock.
  - Many objects can acquire read lock.
- `QSemaphore`
  - Can acquire locks on multiple resources.

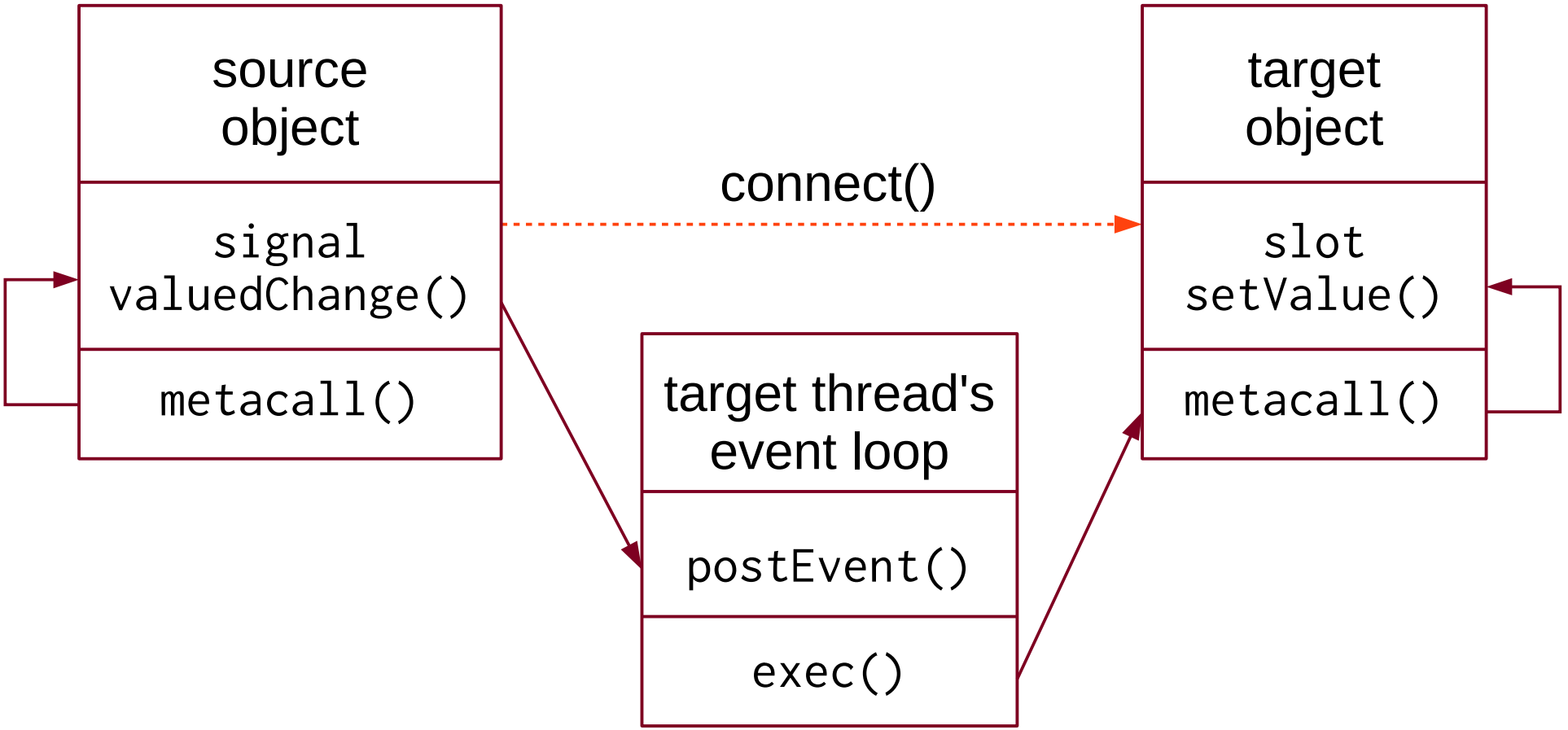
Communication with Main Thread

- Qt application starts with one main thread.
  - Only main thread can create `QApplication` object, call `exec()`.
  - Main thread runs main event loop.
  - Main thread can start secondary threads.
- Secondary threads
  - Run their own event loops.
  - Can talk to each other via shared variables and mutex.
  - Cannot talk to main thread via shared variables and mutex.
    - Would lock main event loop and freeze GUI.
  - Qt's solution: signal-slot.

- Implementation of signal-slot: same thread
  - Synchronous: implemented as function calls via meta object.
    - connection ➔ `metacall(id)`, `id` is signal or slot ID.
    - `emit` signal ➔ `QMetaObject::activate(signalID)`.



- Implementation of signal-slot: different threads
  - Asynchronous: implemented by posting event.
    - Still have `metacall(id)`.
    - `emit` signal ➔ `postEvent(receiver, event)`.



Thread Safety

- Thread-safe function
  - Can be called safely from different threads simultaneously.
- Thread-safe class
  - All class functions are thread-safe.
- Reentrant function
  - Can be called from different threads simultaneously.
  - May not be thread-safe.
  - Functions that don't access shared variables are reentrant.
- Reentrant class
  - Different class instances can be used simultaneously in different threads.

- Most of Qt's non-GUI classes are reentrant.
- C++ classes that don't reference global or shared data are reentrant.
- `QWidget` and its subclasses are not reentrant.
  - They have to run in the same main thread.
  - Cannot directly call GUI functions from secondary thread. Have to emit signals which are connected to slots.
- Qt's network classes are reentrant.

Caution

- Multithreading programs can be difficult to debug.
- Use only if necessary, and with caution.

Summary

- Multithreading runs multiple threads simultaneously.
- UI operations must run in main thread.
- Non-UI operations can run in secondary threads.
- Secondary threads communicate with main thread using signal-slot.
- With multiple threads, need to synchronise threads.
- Protect critical regions that access shared data.
- Avoid locking mutex immediately after unlocking.

Further Reading

[Blan2008] chap. 18:

- Other protection mechanisms:
  - read-write-lock, semaphore, etc.
- Communicating with main thread.

[Summ2011] chap 7:

- QtConcurrent: high-level API for mutlithreading.

[Lewi1995]:

- Introduction to multithreading.

Reference

- J. Blanchette and M. Summerfield, *C++ GUI Programming with Qt 4*, 2nd ed., Prentice Hall, 2008.
- M. Summerfield, *Advanced Qt Programming*, Prentice Hall 2001.
- B. Lewis, SunSoft Press, D. J. Berg, *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, 1995.