Threads em Qt

Prof. Tiago Garcia de Senna Carneiro

2007

Thread - Fundamentos

- **Uma *thread* é um processo leve, portanto possui um fluxo de execução independente e troca de contexto mais rápida que um processo.**

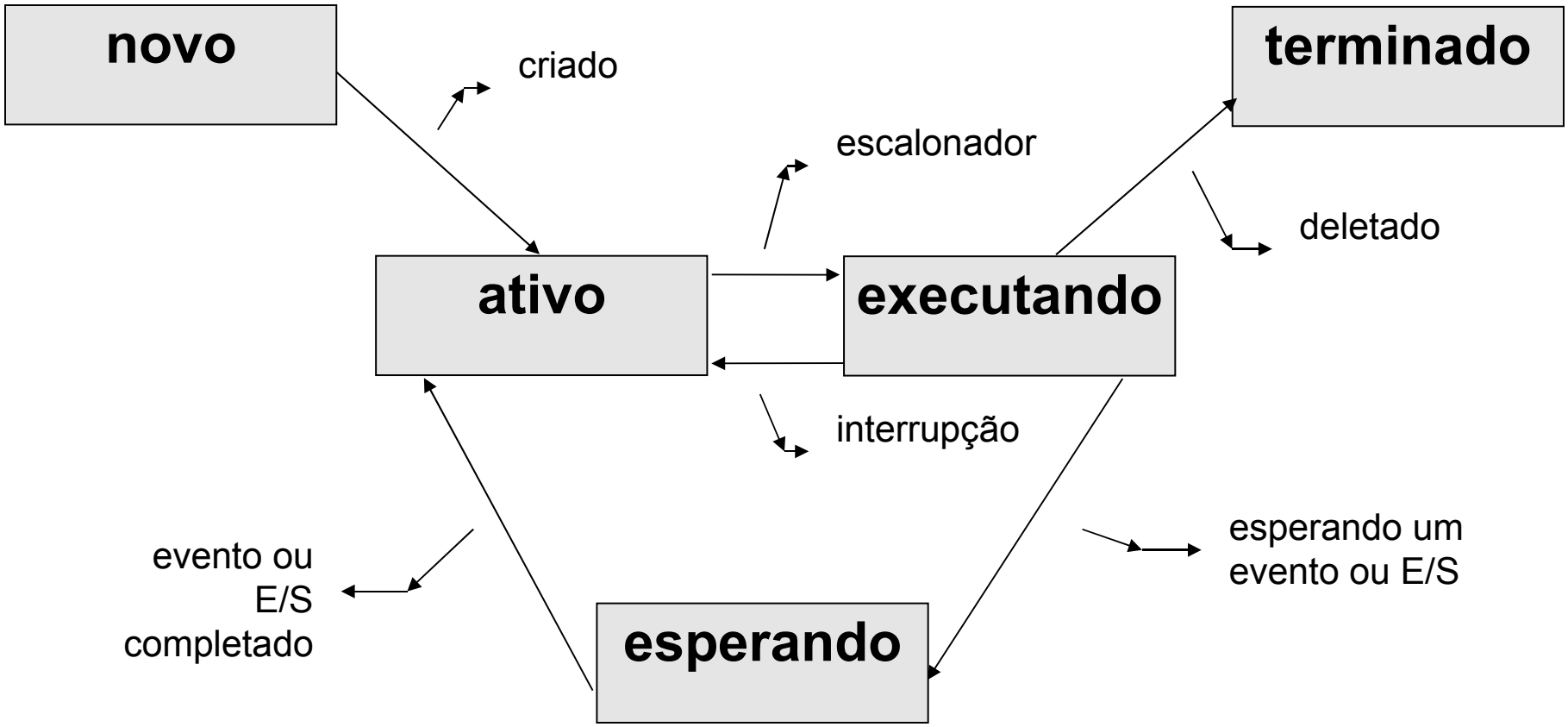
- **Uma *thread* possui um contador de programa separado**

… uma pilha, variáveis locais, etc.

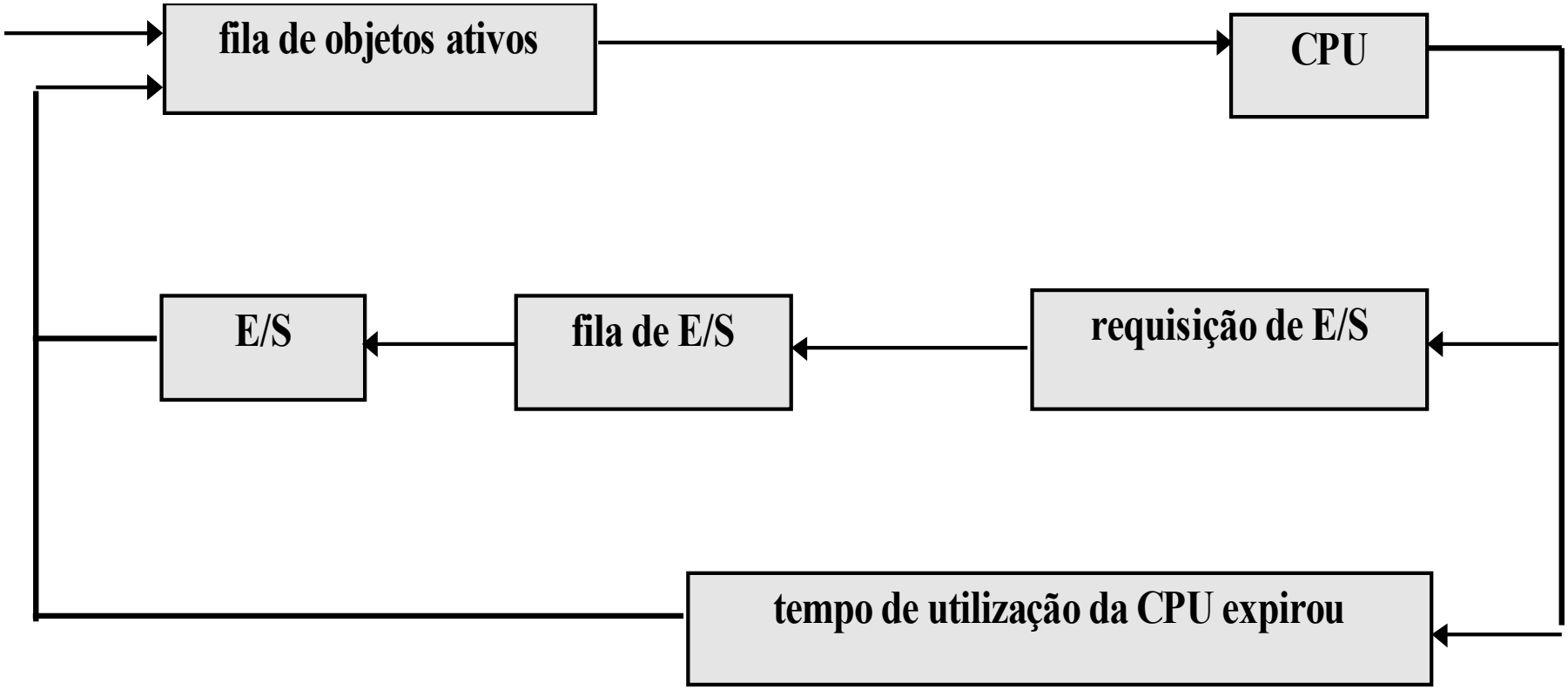
- **Classes, objetos, métodos, etc. não pertencem a uma *thread***

- **Qualquer método pode ser executado por uma *thread*, inclusive simultanemente**

Cliclo de Vida de uma Thread



Escalonamento de Threads



Creating Threads in Qt

```
Class MyThread : public QThread
{
   Q_OBJECT
public:
   MyThread();

   void setMessage(const Qstring &message);
   void stop();

protected:
   void run();

private:
   Qstring messageStr;
   volatile bool stopped;          // will be accessed from
                                   // different threads
}
```

MyThread Implementation

```
MyThread::MyThread{
        stopped = false;
}
```

```
MyThread::stop{
        stopped = true;
}
```

```
MyThread::run{
        while( !stopped )
‘               cerr << qPrintable( messageStr );
        stopped = false;
        cerr << endl;
}
```

void QThread::terminate () [slot]

- Terminates the execution of the thread.
- Discouraged! The thread can be terminated in a not consistent state.

C++ Volatile Variables

- **Volatile** means the storage is likely to change at anytime.
- This **qualifier** tells the compiler to not attempt to optimize the storage referenced by it:
  - the program should always check the physical address
  - not use it in a cashed way

```
// Base address of the data input latch
 volatile unsigned char *baseAddr;

// read parts of output latch
lsb = *handle->baseAddr;
middle = *handle->baseAddr;
msb = *handle->baseAddr;
```

```
lsb = middle = msb = *handle->baseAddr;
```

```
#define TTYPORT 0x17755U

volatile char *port17 = (char)*TTYPORT;
*port17 = 'o';
*port17 = 'N';
```

The compiler would think that the statement *port17 = 'o'
is redundant and would remove it from the object code.

My First *Threaded* Application

- GUI -

```
class QPushButton;

class ThreadDialog : public QDialog
{
    Q_OBJECT

public:
    ThreadDialog(QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void startOrStopThreadA();
    void startOrStopThreadB();

private:
    MyThread threadA; //          what this
    MyThread threadB; //       is doing here?
    QPushButton *threadAButton;
    QPushButton *threadBButton;
    QPushButton *quitButton;
};
```

Prof. Tiago G. S. Carneiro
DECOM - UFOP

My First *Threaded* Application - GUI -

```
ThreadDialog::ThreadDialog(QWidget *parent)
    : QDialog(parent)
{
    threadA.setMessage("A");
    threadB.setMessage("B"); // what this is doing here?

    threadAButton = new QPushButton(tr("Start A"));
    threadBButton = new QPushButton(tr("Start B"));
    quitButton = new QPushButton(tr("Quit"));
    quitButton->setDefault(true);

    connect(threadAButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThreadA()));
    connect(threadBButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThreadB()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addWidget(threadAButton);
    mainLayout->addWidget(threadBButton);
    mainLayout->addWidget(quitButton);
    setLayout(mainLayout);

    setWindowTitle(tr("Threads"));
}
```

My First *Threaded* Application - *slots* -

```
void ThreadDialog::startOrStopThreadA()
{
    if (threadA.isRunning()) {
        threadA.stop();
        threadAButton->setText(tr("Start A"));
    } else {
        threadA.start();
        threadAButton->setText(tr("Stop A"));
    }
}
```

```
void ThreadDialog::startOrStopThreadB()
{
    if (threadB.isRunning()) {
        threadB.stop();
        threadBButton->setText(tr("Start B"));
    } else {
        threadB.start();
        threadBButton->setText(tr("Stop B"));
    }
}
```

```
void ThreadDialog::closeEvent(QCloseEvent *event)
{
    threadA.stop();
    threadB.stop();
    threadA.wait();
    threadB.wait();
    event->accept();
}
```

My First *Threaded* Application - main() -

```
#include <QApplication>

#include "threaddialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ThreadDialog dialog;
    dialog.show();
    return app.exec();
}
```

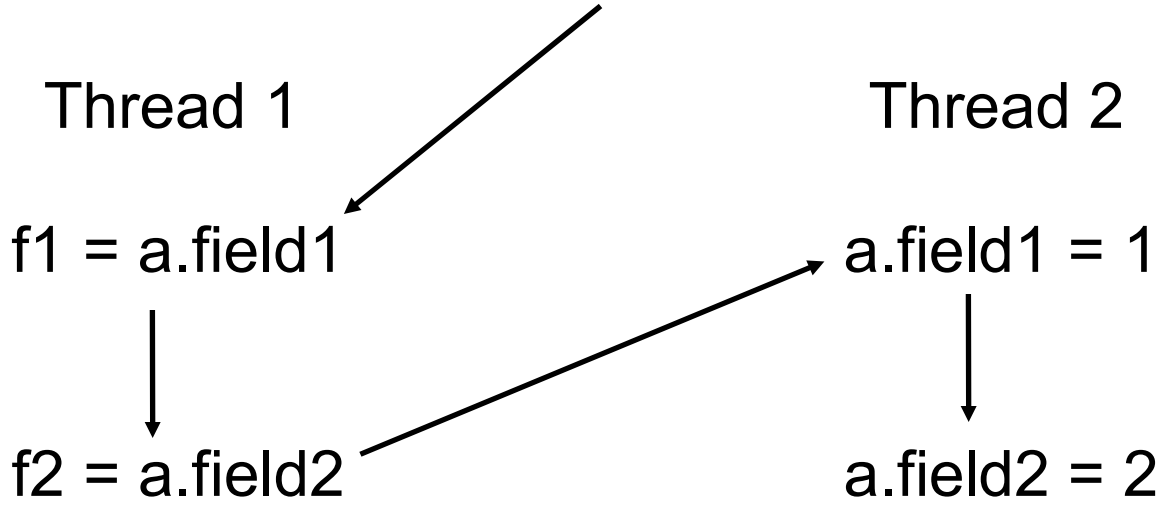
Race Conditions

- In a concurrent world, always assume someone else is accessing your objects
- Other threads are your adversary
- Consider what can happen when simultaneously reading and writing:

Thread 1	Thread 2
f1 = a.field1	a.field1 = 1
f2 = a.field2	a.field2 = 2

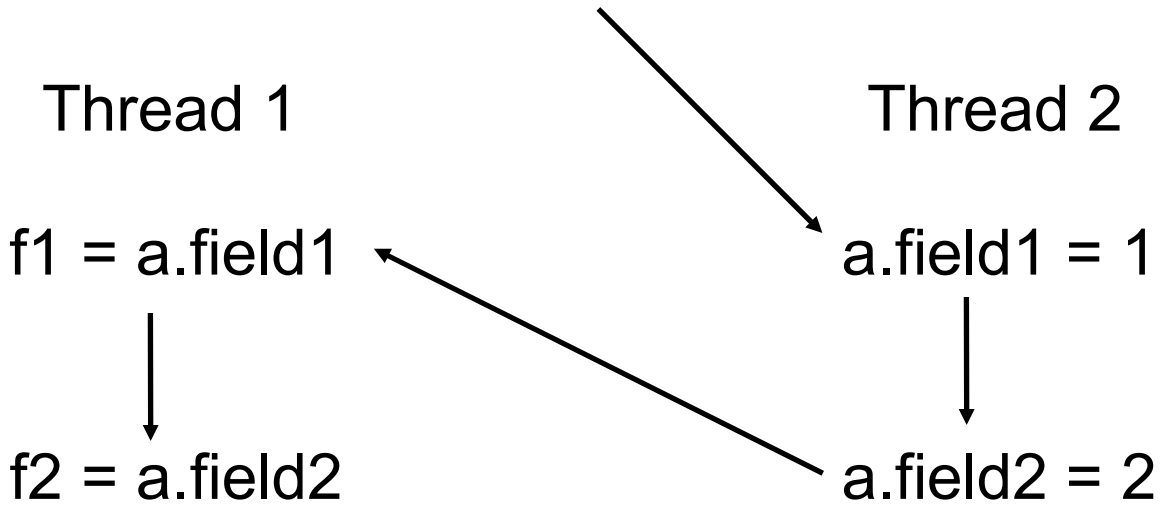
Race Conditions

- Thread 1 goes first
- Thread 1 reads original values



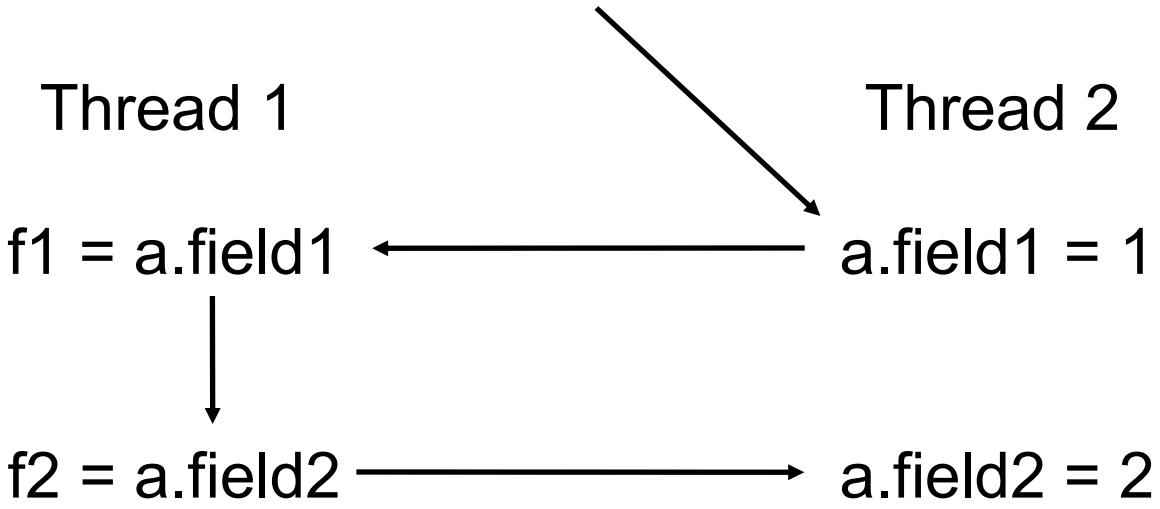
Race Conditions

- Thread 2 goes first
- Thread 1 reads new values



Race Conditions

- Interleaved execution
- Thread 1 sees one new value, one old value



Non-atomic Operations

- 32-bit reads and writes are guaranteed atomic
- 64-bit operations may not be

- Therefore,

int i; double d;

Thread 1	Thread 2	
i = 10;	i = 20;	i will contain 10 or 20
d = 10.0;	d = 20.0;	i might contain garbage

Mutual Exclusion in Qt

- Qt provides 4 classes for *thtreads* synchronization:
  - **QMutex**: low performance, no simultaneous read
  - **QReadWriteLock**: only one resource per lock
  - **QSemaphore**: several resource per semaphore
  - **QWaitCondition**: allows thread to wake up other threads when some condition has been met.

Qmutex & QMutexLocker

- Locking an only resource:

Representation:
- **Boolean**: if ***false*** block.
- **lock**(): while(!boolean) {}
- **unlock**(): boolean = **true**;

```
MyData sharedData;

MyThread{
  ...
private:
  Qmutex mutex;
}
```

```
void MyThread::run(){
    for(;;;){
            mutex.lock();
            changeData( &sharedData );
            mutex.unlock();
    }
}
```

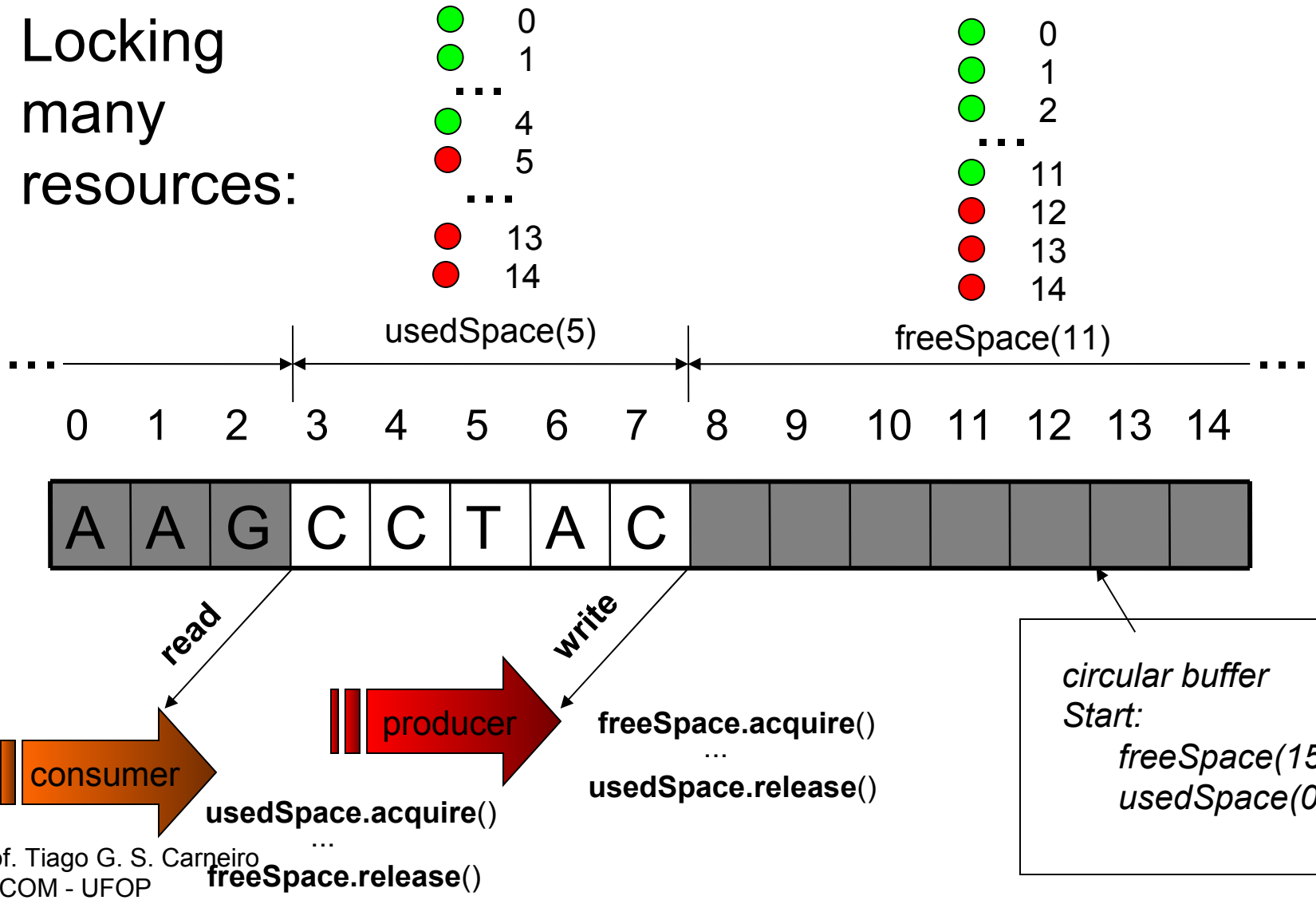
- **QMutex**: What happens if ***changeData()*** raises an exception?

```
void MyThread::run(){
    for(;;;){
            {
            QMutexLocker locker( &mutex);
            changeData( &sharedData );
            locker.unlock(); // not necessary, class destructor releases the mutex
            }
    }
}
```

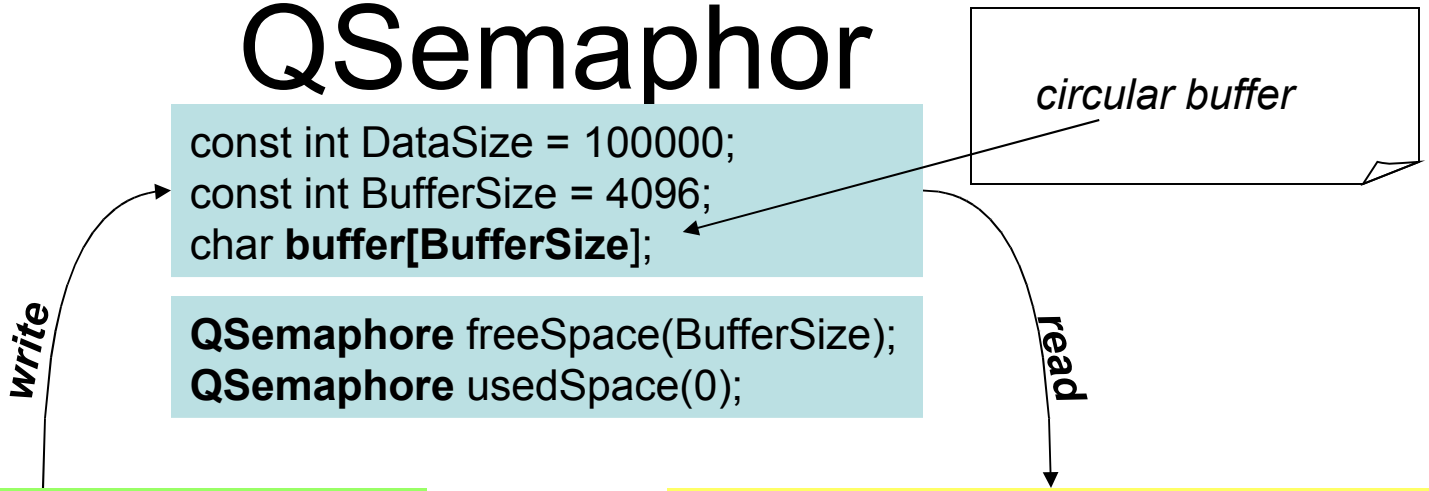
Semaphor

Representação:
- **Inteiro**: se 0 bloqueia.
- **aquire**(): inteiro --;
- **relase**(): inteiro++;

- Locking many resources:



QSemaphor




```
class Producer : public QThread
{
public:
    void run();
};
```

```
void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        freeSpace.acquire();
        buffer[i % BufferSize] =
                    "ACGT"[uint(rand()) % 4];
        usedSpace.release();
    }
}
```

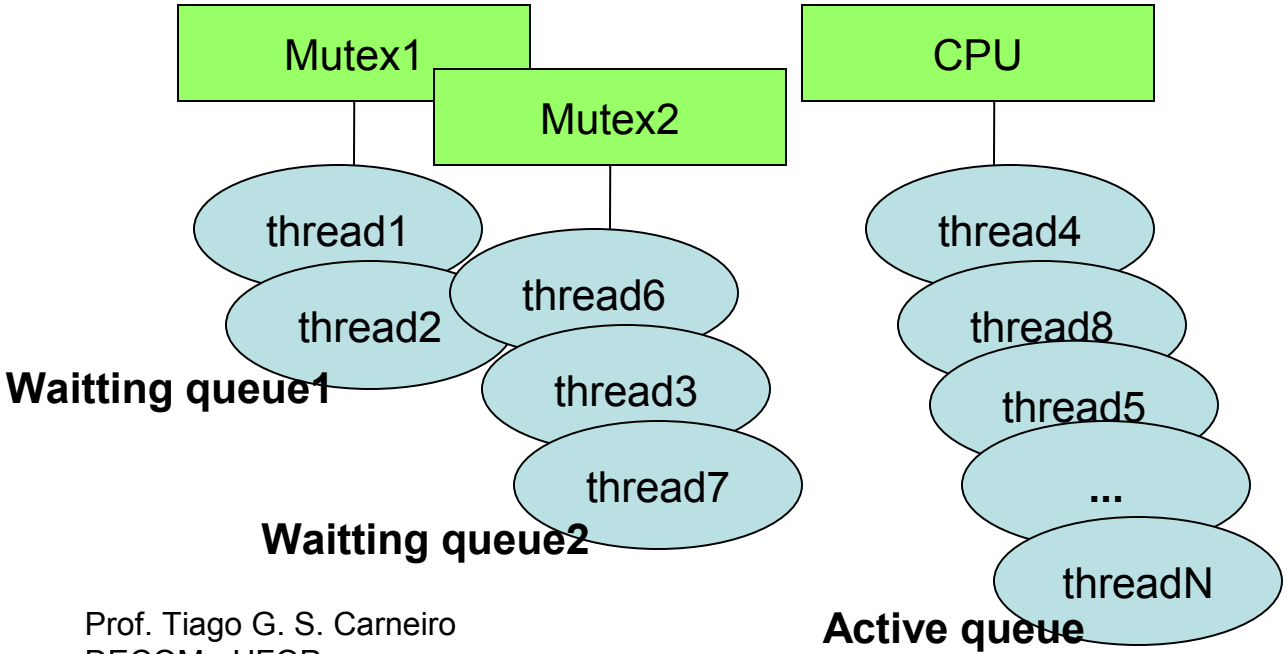
```
class Consumer : public QThread
{
public:
    void run();
};
```

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        usedSpace.acquire();
        cerr << buffer[i % BufferSize];
        freeSpace.release();
    }
    cerr << endl;
}
```

Waiting for a Condition

- Say you want a thread to wait for a condition before proceeding

- An infinite loop may deadlock the system: while (!condition) {}

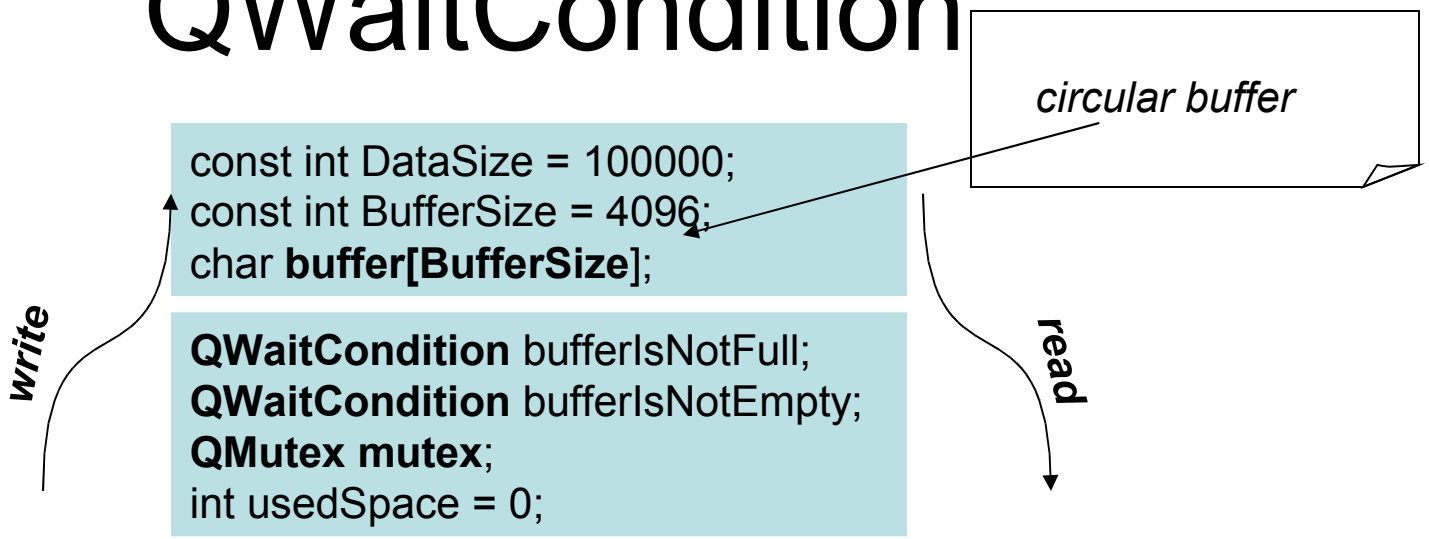


Representação:
- **mutex thread queue**.

- **wait**() = put thread in the mutex waitting queue.

- **notifyAll**() = put thread in the cpu active queue.

QWaitCondition




```
void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        while (usedSpace == BufferSize)
            bufferIsNotFull.wait(&mutex);
        buffer[i % BufferSize] =
                    "ACGT"[uint(rand()) % 4];
        ++usedSpace;
        bufferIsNotEmpty.wakeAll();
        mutex.unlock();
    }
}
```

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        while (usedSpace == 0)
            bufferIsNotEmpty.wait(&mutex);
        cerr << buffer[i % BufferSize];
        --usedSpace;
        bufferIsNotFull.wakeAll();
        mutex.unlock();
    }
    cerr << endl;
}
```

- FIM -

Obrigado!