تعدد المسالك

تخيل لو ان برنامجك توقف بسبب رسالة او بسبب تحميل ملفات التشغيل او لأي سبب آخر و أن هناك مهام لا يجب ان تتوقف بل يجب ان تعمل بمجرد عمل التطبيق مثل الصور المتحركة أو حتى تطبيقات أخرى في برنامجك ويجب ان لا تتوقف عن العمل طوال فترة البرنامج من فتحه الى إقفاله .
وأيضا عندما تكون هناك أعمال ضخمة وتحتاج الى تقسيمها الى أعمال صغير كل على حدى قد يتبادر السؤال الى ذهنك مالحل اذا ؟؟.
من حسن الحظ بإمكانك تجنب مخاطر تكدس المهام وتوقفها عن العمل فالمعالجات الحديثة تمكن لك اجراء اكثر من تطبيق في نفس الوقت فالمعالج ينقسم لينفذ مهام مختلفة ولا يبالي بمهمة الآخر .....
ويمكنك أيضا ان تجعل هذه الأقسام من المعالجات تتساعد فيما بينها لتكون نظام متزامن فيما بينه يعطي القوة لعملك وتكون مشتركة في بيانات موحدة تتناقلها فيما بينها .
وهناك نوعين من المسالك وهو الرئيسي والثانوي والرئيسي هو الذي يتحكم بعمل البرنامج بأسره وعند إقفاله يتم اقفال جميع المعالجات الثانوية اما الثانوي فإنه يعمل عمله الطبيعي متجاهلا ماحوله من معالجات ثانوية .

كيف تبدأ ببناء مسلك جديد ؟

قامت Qt بإنشاء مايربوا عن 10 صفوف لتكون لك عونا في هذا الموضوع و لكي تجعلك أكثر حرية في بناء مسالك جديدة تتعامل فيما بينها بتنظيم وتزامن .
ولكن في بداية الأمر يجب علينا ان نعرف ماهو لب كل واحد من هذه المسالك وأين نضع المهام التي سوف نوكلها لهذا المسلك لمعالجته وكيف نجعله يبدأ بالعمل ؟؟؟
في الحقيقة الوظائف السابقة التي ذكرتها تعتمد على صف وحيد وهو QThread
ويحتوي على دوال تقوم بجميع المهام السابقة وهذا ماسوف نأتي عليه مستقبلا في هذا الفصل
ولكن بقي عليك أن تعرف شيئ واحد فقط هو أنك في حال اردت ان تبني مسلك جديد يقوم بوظائف جديدة عليك بوراثة الصف Qthread وإعادة تعريف الأمر run لتضع جميع ماتريد تنفيذه بداخلها .

QThread

تحدثنا عنه في مقدمة الموضوع ولكن دعنا نتعرف اهم مافي هذا الصف من دوال:
Start ويقوم بتشغيل المعالج وسوف يتم تنفيذ ما بداخل الدالة run.

Wait وينتظر البرنامج حتى ينتهي المعالج من مهامه او يمكن توقيته بحيث اذا انتهى الوقت قبل ان ينتهي المعالج من مهامه يتم اغلق المعالج وبه بعض العمل وإن انتهى المعالج من تنفيذ الأوامر قبل نهاية التوقيت يتم اغلاق المعالج مباشرة .
Sleep يقوم بإعطاء غفوة للمالج يتوقف اثناءها عن المعالجة لفترة انت تقوم بتحديدها بالثواني ومثله msleep ولكن بجزء من الف في الثانية .
Exit يتم الإنتها من تكرار البرنامج ويعيد قيمتين اما صفر او غيرها والصفر يعني عملية خروج صحيحة .
Terminate ويطلق عليه عملية الإقفال العنيف حيث انه لا يدع للمعالج أي فرصة في العمل ويتم اغلاقه بعنف ويتجاهل جميع المهام التي بداخله .
وهناك اشارات يتم قذفها مثل finished وتقذف عن الإنتهاء من المعالج بطريقة سليمة او started عند ابتداء المعالج بالعمل او terminated عند إقفال المعالج ولم يكمل عمله .
عموما دعنا نأخذ مثالاً بسيطاً ونشرح عمله لتستوعب فكرتها .

```
1. class TextThread : public QThread
2. {
3. public:
4. TextThread( const QString &text );
5. void run();
6. private:
7. QString m_text;
8. };
```

بالإضافة الى وراثته من الصف Qthread كما ذكرنا سابقا فإنه يحتوي على دالة البناء ثم الدالة run وهي ماسوف نكتب المهام بداخلها والمتغير النص m_text .

```
bool stopThreads = false;
```

قمنا بإنشاء هذا المتغير العام لينبهنا في طلب توقف المعالج سوف تعرف لماذا وضعناه في القادم .

```
1. TextThread::TextThread( const QString
   &text ) : QThread()
2. {
3. m_text = text;
4. }
```

دالة البناء وتقوم بتهيئة المتغير m_text .

```
1. void TextThread::run()
2. {
3. while( !stopThreads ){
```

```
4. qDebug() << m_text;
5. sleep( 1 );
6. }
7. }
```

وهذه دالة عمل المعالج وسوف تبدأ بالمالجة عند استدعاء الأمر start في الدالة main لقد قمنا بتكرار الحدث لكي لا يتوقف الا اذا طلبنا ايقاف البرنامج وذلك عن طريق تحويل المتغير stopThreads الى true ثم نقوم بطباعة المحارف التي ارسلناها عبر دالة البناء الى الشاشة console وفي النهاية قمنا بإعطاء غفوة للمعالج اذا ما انتهت مدة الغفوة يقوم بتكرير الحدث مرة أخرى .

الآن سوف نتعرف على الدالة main

```
1. int main( int argc, char **argv )
2. {
3. QApplication app( argc, argv );
4. TextThread qt( "qt" ), thread( "thread" );
5. qt.start();
6. thread.start();
7. QMessageBox::information( 0, "Threading",
   "Close me to stop!" );
8. stopThreads = true;
9. qt.wait();
10.thread.wait();
11.return 0;
12.}
```

قمنا بداية الأمر بإنشاء معالجين من التي صنعناها الأول بإسم qt والآخر بإسم thread

ثم طلبنا بتشغيل المعالجات بإستخدام الأمر start ثم امرنا بظهور رسالة من المعلوم لك مسبقا عند ظهور أي رسالة يتم ايقاف عمل المعالج حتى تغلق النافذة ولكن المسالك qt & thread مازالا مستمرين .

بعد إغلاق الرسالة يتم ارسال طلب اقفال البرنامج بتحويل المتغير stopThreads الى true قد تكون بعض العمليات لم تعالج لذ سوف يصبر البرنامج حتى يتم معالجة جميع الأوامر بإستخدام الدالة wait ومن ثم يتم اقفال البرنامج بإرجاع القيمة 0 .

في الرنامج السابق كان الناتج :

```
"qt"
"thread"
"thread"
"qt"
"thread"
```

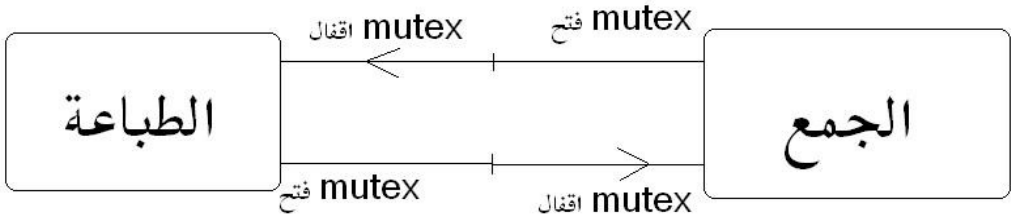
"qt"
"qt"
.....
.....

عند القائك نظرة للمخرجات ستجد أن المسالك تقوم بإرسال مالديها من مخرجات بدون النظر في حالة المسالك الأخرى ولكن ماذا لو أردت مزيد من التنظيم في عملك ؟؟.

التزامن

في مثالنا السابق لقد أطلعت على مدى العشوائية في النواتج فليس هناك تنسق في المخرجات والسبب معروف وهو أن كل مسلك يعمل بإستقلالية عن الآخر في بعض الأعمال انت بحاجة الى التناسق في وظائف كل مسلك فلو أردت ان تصنع مسلك يقوم بعمليات حسابية ومسلك آخر يقوم بطباعة النواتج وبما ان العملية كما ذكرنا سابقا تتم بشكل عشوائي فستجد أن مسلك طباعة النواتج قد يقوم بطباعة الناتج قبل الإنتهاء من العملية ففي هذه الحالة انت بأشد الحاجة الى منظم يعمل كإشارة مرور بحيث لن يتم طباعة النواتج الا بعد الإنتهاء من العملية حسابية .
وتسمى هذه العملية ب mutex والصف الذي يدعمها هو Qmutex ويتم اقفال ال mutex بإستخدام الدالة lock() وفتحها بإستخدام الدالة unlock() انظر للصورة .

مثال:



هذا المثال تطبيق على الصورة في الأعلى فقمنا بصنع مسلكين الأول يقوم بعملية الحساب والآخر بعملية طباعتها وسوف يتم إقفال الmutex عند عملية الحساب وعند الإنتهاء من الجمع يتم فتحها ليستغل ال مسلك الآخر الفرصة ويقفل على نفسه ال mutex وعندما ينتهي من الطباعة يقوم بفتحه وهكذا دواليك .
المتغيرات العامة

```
1. int m=0;
```

2. **QMutex mutex**;

المتغير الأول سوف تجري عليه عملية الجمع ومن ثم طباعته والثاني هو المتغير العام mutex حيث هو من يتحكم ببواابة الدخول في مسالك الطباعة والجمع .

صف عملية الجمع :

```
1. class addition:public QThread{
2. public:
3. void run(){
4. while(1){
5. mutex.lock();
6. m++;
7. sleep(1);
8. mutex.unlock();
9. }
10.}
11.};
```

ليس به الا دالة واحدة وهي run وكما ذكرنا سابقا هي الدالة التي سوف يتم معالجة مابداخلها
ولو نظرت ما بداخلها فستجد ان عملية التكرار لا نهائية ويبدأ بإقفال ال mutex في حال الإقفال فإن صف الطباعة سوف ينتظر حتى يتم فتحه مرة أخرى ليسمح له بالعبور ويقفل على نفسه .
وبعد أن أقفل على نفسه الباب قام بإضافة واحد الى المتغير العام m وإعطاء غفوة للمعالج ومن ثم بعد ما استيقظ قام بفتح الباب ليقوم بتكرار العملية ولكنه تفاجأ بأن الباب تم قفله من قبل صف الطباعة .

صف عملية الطباعة :

```
1. class debug:public QThread{
2. public:
3. void run(){
4. while(1){
5. mutex.lock();
6. qDebug()<<tr("%1").arg(m);
7. sleep(1);
8. mutex.unlock();
9. }
10.}
11.};
```

مثل سابقه فهو ليس به الا دالة وحيدة run فيقوم وأول ما بدأ به هو التكرار الى مالا نهاية فيبدأ قفل ال mutex على نفسه ليمنع اي تغير على المتغير m اثناء طباعته .
ثم قمنا بطباعة الرقم على شاشة console وفتح ال mutex ليقوم صف عملية الجمع بدخول الغرفة وإقفالها

على نفسها قبل ان يقوم هذا الصف بالدخول مرة أخرى وفقا لعملية التكرار.

الدالة main

```
1. int main(int argc, char ** argv)
2. {
3. QApplication app( argc, argv );
4. debug a;
5. addition b;
6. b.start();
7. a.start();
8. return app.exec();
9. }
```

وفيها قمنا بالتصريح عن المتغيرين لنقوم بتنفيذ الالة run بكل منهما عن طريق تنفيذ الأمر start .
وكان ناتج هذا البرنامج كالتالي .

1
2
3
الى مالا نهاية

لو لاحظة ستجد ان هناك تزامن فبعد كل عملية اضفة يتم طباعة الناتج تخيل لو لم يكون ال mutex موجود سوف تجد ان الناتج سوف يصبح هكذا

1
1
3
4
4
5

وبهذه العشوائية أعتقد انك لن تستطيع استنتاج الرقم التالي .
وقد أصبحت الآن مدركا لهذا الموضوع دعنا نتعرف لبعض الصفوف التي سوف تفيدنا في هكذا موضوع .
الصف QMutexLocker وهو يقوم بالإقفال على mutex ويتم اعادة فتحها بمجرد ان يتم استدعاء إجراء التدمير (~) او الكلمة المفتاحية delete او استخدام الدالة unlock المتوفرة داخل الصف QMutexLocker ويمكنك أيضا اعادة إقفالها بإستخدام الدالة relock .
مثال :
هذا البرنامج مشابه في الطريقة والمخرجات للبرنامج السابق والإختلاف فقط في ما داخل الدالة run .

```
1. int m=0;
2. QMutex mutex;
```

المتغير الأول سوف تجري عليه عملية الجمع ومن ثم طباعته والثاني هو المتغير العام mutex حيث هو من يتحكم ببواابة الدخول في مسالك الطباعة والجمع .

```
1. class addition:public QThread{
2. public:
3. void run(){
4. while(1){
5. QMutexLocker locker(&mutex);
6. m++;
7. sleep(1);
8. delete &locker;
9. }
10.}
11.};
```

وهو مشابه للصف addition **في البرنامج السابق ولكن قمنا بتغيير طريقة اقفال وفتح ال mutex فقد قمنا بإستخدام الصف QmutexLocker وأعطيناه مؤشر للمتغير locker للمتغير العام mutex ليقوم بقفله تلقائيا وبعد الإنتهاء من عملية التجميع قمنا بحذف المتغير locker وهذا ما يعني أنه قمنا بفتح ال mutex .**

```
1. class debug:public QThread{
2. public:
3. void run(){
4. while(1){
5. QMutexLocker locker(&mutex);
6. qDebug()<<tr("%1").arg(m);
7. sleep(1);
8. delete &locker;
9. }
10.}
11.};
```

وهو مشابه للصف debug **في البرنامج السابق ولكن قمنا بتغيير طريقة اقفال وفتح ال mutex فقد قمنا بإستخدام الصف QmutexLocker وأعطيناه مؤشر للمتغير locker للمتغير العام mutex ليقوم بقفله تلقائيا وبعد الإنتهاء من عملية طباعة الناتج على الشاشة console قمنا بحذف المتغير locker وهذا ما يعني أنه قمنا بفتح ال mutex لتبدأ عملية الجمع بالولوج وقفل mutex على نفسها بإستخدام QmutexLocker .**

الدالة main هي نفسها الدالة main في البرنامج السابق.

```
1. int main(int argc, char ** argv)
2. {
3. QApplication app( argc, argv );
4. debug a;
5. addition b;
```

```
6. b.start();
7. a.start();
8. return app.exec();
9. }
```

والمخرجات مشابهة تماما لمخرجات البرنامج السابق

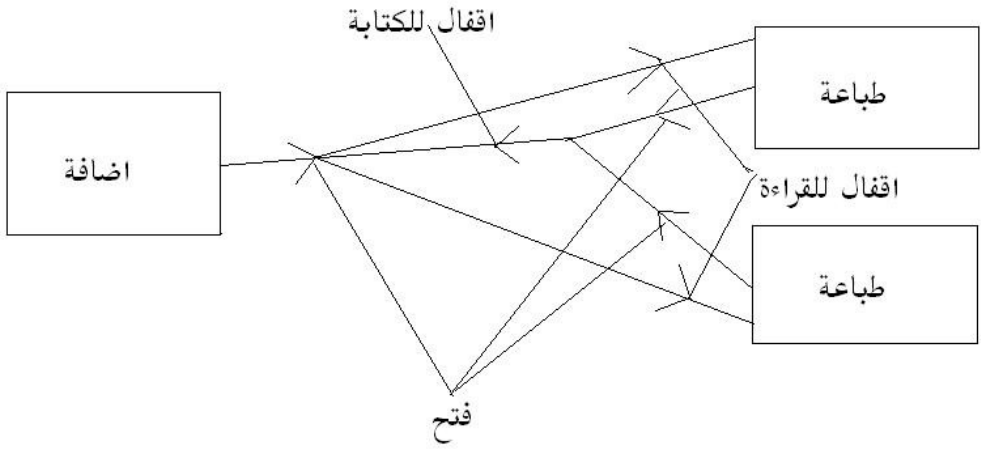
1
2
3
الى مالا نهاية

الإقفال للكتابة أو القراءة

أستخدمنا في السابق ال mutex في عمليات قفل وفتح المسالك وهي قد تكون مجدية وفعالة في حال كان معنا مسلكين يقفل الأول على نفسه ال mutex وينتظر الآخر تخيل لو دخل طرف جديد في الموضوع وأصبح لدينا ثلاثة مسالك سوف يقفل الأول على نفسه وينتظر الثاني والثالث لحين فتح ال mutex فمن برأيك سوف يدخل أثناء فتحه ويقفل على نفسه ؟؟ .
وتخيل لو ان برنامجك به أكثر من مسلكين ولنفترض ثلاثة وتريد ان يتم معالجة مسلكين بشكل آني هل سوف تستخدم mutex بالتأكيد لا فال mutex تسمح بمسلك واحد بالمرور والإقفال على نفسه والإنتظار .
نعم في النقاط السابقة كان هذا عيب ينقص من قدر mutex فكان الحل بإنتاج صفوف أكثر قدرة على تنظيم معالجة المسالك فقامت Qt بإنتاج الصف QreadWriteLock
للإقفال من أجل القراءة او الإقفال من أجل الكتابة وتمثلها الدوال lockForRead و lockForWrite او بإستخدام الصفوف الفتح والإغلاق التلقائي QwriteLocker و QreadLocker وهي تستقبل في وسيطها عنوان من النوع QreadWriteLock لكي تقوم بإقفاله في دالة البناء وإعادة فتحه في دالة الهدم .
ماذا يعني إقفال للكتابة أو إقفال للقراءة ؟؟
كتابة ماذا ؟ وقراءة ماذا ؟ انت لا تعرف لأنك سوف تقفل على الوصول للشفرة وليس هناك قوانين هل الشفرة قراءة ام كتابة ؟
في الحقيقة هذه المسميات اتت لتطابق نظرية التنظيم للمسالك فالإقفال للكتابة يقفل القراءة والكتابة , اما الإقفال للقراءة فيقفل الكتابة فقط من هذا التنظيم تم تسمية هذا الصف وبمعنى آخر في حال كان مقفل للكتابة سينفذ مسلك واحد لتنفيذ الشفرة نظرية mutex اما عندما يكون مقفل للقراءة فستنفذ جميع المسالك التي تقفل

للقراءة وبمعنى آخر يمكنك ان تنفذ أكثر من مسلك في وقت واحد وهذه هي النظرية الجديدة هنا .
سوف نقوم بتطبيق جميع ماتكلمنا عنه مسبقا في المثال الآتي وسوف تلاحظ مدى التنظيم للمسالك الذي قد توفره لك هذه الصفوف في تطبيقاتك .

المثال:




```
1. int m=0;
2. QReadWriteLock lockT;
```

المتغير m لإجراء عمليه الجمع والثاني هو الذي كنا نتحدث عنه في حديثنا السابق وهو الذي يقوم بتنظيم المسالك .

```
1. class addition:public QThread{
2. public:
3. void run(){
4. while(1){
5. lockT.lockForWrite();
6. m++;
7. lockT.unlock();
8. msleep(1);
9. }
10.}
11.};
```

كما ترى فإن الدالة الوحيدة هنا هي run وكما لاحظت فإن الفصل من أوله الى آخره كان يعتمد على اعادة تعريف الدالة run لأنه هو لب المسلك مثل الدالة main للبرنامج ككل . بعد إنشائن للدالة قمنع بعملية تكرار لا نهائية لكي يستمر البرنامج بالعمل ومن ثم أقفلنا منظم المسالك للكتابة

وبمعنى آخر سوف يتم إقفال أي مسلك سواء قراءة او كتابة وسيدخل هو لوحده في المعالجة ليتم اضافة واحد الى المتغير العام m وبعدها نعيد فتح المنظم للمسالك ليدخل فيم بعد المسالك الأخرى وتقفل للقراءة مما يعني ان أي مسلك للقراء يتم تنفيذه وأي مسلك للكتابة سوف ينتظر حتى يتم فتح المنظم للمسالك وفي النهاية قمنا بإعطاء غفوة للمسلك لجزء من الألف لنعطي فرصة لبقية المسالك بالدخول والإقفال على نفسها للقراءة.
ملاحظة : ويمكنك استخدام الصف QwriteLockar(&lockT) دالة البناء سوف تقوم بقفله للكتابة ولكي نفتحه سوف نستخدم دالة الهدم للصف فالأول بدلا عن lockForRead الآخر بدلا عن unlock .

```
1. class adbug:public QThread{
2. public:
3. void run(){
4. while(1){
5. lockT.lockForRead();
6. qDebug()<<m;
7. sleep(1);
8. lockT.unlock();
9. }
10.}
11.};
```

ويختلف عن الصف السابقة انه يقفل للقراءة أي يقفل الكتابة فقط ويبقى كل ماهو للقراءة مفتوح للمعالجة وهي الفكرة الجديدة في الصف الذي جعلته مختلفا هن Qmutex بعد أن يقفل للقراءة يقوم بطباعة الرقم على الشاشة console وفي الآخر يتم إعادة فتحه منظم المسالك مرة أخرى .

```
1. int main(int argc, char ** argv)
2. {
3. QApplication app( argc, argv );
4. addition a;
5. adbug b,c;
6. a.start();
7. b.start();
8. c.start();
9. return app.exec();
10.}
```

قمنا بإنشاء متغير للإضافة ومتغيرين للطباعة وكان ناتج البرنامج كالتالي

0

0
1
1
2
2
3
3

وهكذا الى ان تقوم بغيقاف البرنامج .

الإشتراك في المصادر

في حديثنا وبرامجنا السابقة كانت هناك مصادر مشتركة مثل المتغير m فكنا ننظم العمليات فيه بإستخدام mutex او الإقفال للكتابة والقراءة ولكن هذا غير كاف انت بحاجة لخوارزميات أكثر لكي تكون نظام مسالك يأمن لك تعامل افضل مع المسالك فتم إنتاج نظام اعمدة الإشارة (semaphore)

ماهي أعمدة الإشارة (semaphore)

هو mutex ولك بمفهومه الأوسع (تعميم ل mutex) .
يمكن القول ان أعمدة الإشارة هي عداد mutex , وأن mutex هو عبارة عن أعمدة اشارات ثنائي.
اذا وقد عرفت اعمدة الإشارة وأنها عبارة عن نظام رياضي لإدارة المسالك لكن السؤال كيف نستخدمه ؟
لقد تعلمت على mutex ان مايكافئه في هذه الحالة ه و

```
QSemaphore                  QMutex mutex;
semaphore(1);               mutex.lock();
semaphore.acquire();        mutex.unlock()
semaphore.release();        ;
```

حيث كل عملية وما يقابله في QSemaphore
فالهدف هو الحفاظ على القيمة وهي واحد .
ملاحظة : قيمة أعمدة الإشارات لا تقل عن صفر .
وتزيد القيمة كلما استخدمت الدالة release بمقدار واحد مالم تضع في وسيطها رقم يحدد مقدار الزيادة وتقل القيمة كلما استخدمت الدالة acquire بمقدار واحد مالم تضع في وسيطها رقم يحدد مقدار النقصان اما اذا اردت ان تستفسر عن القيمة تستخدم الدالة available.

مثال :

```
1. const int bufferSize = 20;
2. QChar buffer[ bufferSize ];
```

```
3. QSemaphore freeSpace( bufferSize );
4. QSemaphore availableData( 0 );
5. bool atEnd = false;
6. QString m_text="application development with Qt
   framework";
```

هذه المتغيرات جميعها عامة وأهم مافيها هو المتغيرين

QSemaphore freeSpace(bufferSize);

QSemaphore availableData(0);

وهما خلاصة شرحنا للمثال دعنا نكمل الآن وثم توضح الصورة في ذهنك أكثر

```
1. class TextProducer:public QThread{
2. public:void run();
3. };
4. class TextConsumer:public QThread{
5. public:void run();
6. };
```

صنعنا صفين الأول للإنتاج والثاني للإستهلاك وكلاهم يحويان الدالة run

والتي تعمل بجرد استدعاء الأمر start في الدالة main .

```
1. void TextProducer::run(){
2. for( int i=0; i<m_text.length(); ++i ){
3. freeSpace.acquire();
4. buffer[ i % bufferSize ] = m_text[ i ];
5. if( i == m_text.length()-1 )atEnd = true;
6. availableData.release();}
7. }
```

في هذا الأمر قمنا بعملية التكرار أولا وهي من 0 الى اخر حرف في المتغير m_text ثم

قمنا بإنقاص واحد من freeSpace في كل مرة في عملية التكرار ثم قمنا بتخزين الحرف في buffer ثم اذا وصل عدد مرات التكرار الى عدد الحروف فإنه يرسل القيمة true الى المتغير atEnd ويتم اضافة واحد الى availableData .

في الكود السابق هناك تعبيرات يجب ان تعرفها مثل buffer[i % bufferSize] وهذه الطريقة لتوفير مساحة فيضمن ان الحرف سوف يخزن في عناصر لن يزيد عن المساحة المسموح بها وهي 19 وفي حال تم تخزين أكثر من 20 حرف يبدأ في التخزين من الصفر .

```
1. void TextConsumer::run(){
2. int i = 0;
3. while( !atEnd || availableData.available() ){
4. availableData.acquire();
5. qDebug() << buffer[ i ];
6. i = (i+1) % bufferSize;
7. freeSpace.release();}
8. }
```

وهذه دالة للمستهلك في حال لم يصل المنتج الى النهاية او انتج كمية جديدة من الحروف غير مستهلكة فإن عملية التكرار مستمرة ثم يتم انقاص واحد من availableDataويتم تحرير الحرف المنتج في console وأضفنا للمتغير i واحد وإذا تجاوز الرقم 20 فإنه يبدأ العد من جديد حسب نظرية الباقي لكي يواكب الإنتاج ان كان كمياته كبيرة وفي النهاية تم زيادة freeSpace واحد ل يرجع لقيمته الأصلية وهي 20 .

```
1. int main( int argc, char **argv ){
2. QApplication app( argc, argv );
3. TextProducer producer;
4. TextConsumer consumer;
5. producer.start();
6. consumer.start();
7. producer.wait();
8. consumer.wait();
9. return 0;
10.}
```

قمنا بإدراج الصفين داخل الدالة main ممثلة في متغيرين ليبدآ العمل ولن يتم ايقاف البرنامج الى بعد ان ينتهي من المخرجات بالكامل .
وكانت المخرجات هي حروف الجملة السابقة على التوالي .

A
P
P
L

الى نهاية الجملة .

القالب thread storage

لقد أستخدمت متغيرات عامة في جميع البرامج السابقة ولكن أنت الآن تريد ان تستخدم متغير عام لعدة مسالك ولكن المتغير العام هذا سوف يعطي قيمة مختلفة في كل مسلك مختلف قد تستطيع التحايل وتجلب المعرف للمعالج الحالي `QThread::currentThread()` وعلى أساسه تضع المعلومة ولكن هذا سوف يأخذ منك الكثير من التفكير لمعالجة أفضل .
اذا الحل فقط هو QThreadStorage وهو قالب كما يتضح لك من العنوان وبالرغم من انه قالب فهو يريد فقط مؤشرات لا غير وأهم دواله هي hasLocalData وهي تعيد القيمة true في حال انك قمت بإضافة مؤشر من نفس نوع القالب بإستخدام الدالة setLocalData ويمكنك جلبها بإستخدام الدالة localdata اذا اصبح لديك متغير عام تسطيع اضافة له بيانات محلية

هذا كل مافي الأمر .
مثال

```
1. QThreadStorage<QHash<int, double> *> cache;
2. void insertIntoCache(int id, double value)
3. {
4. if (!cache.hasLocalData())
5. cache.setLocalData(new QHash<int, double>);
6. cache.localData()->insert(id, value);
7. }
8. void removeFromCache(int id)
9. {
10.if (cache.hasLocalData())
11.cache.localData()->remove(id);
12.}
```

في السطر الأول قمنا بالتصريح عن المتغير وجعلنا نوعه الحاوية QHash .
وصنعنا دالتين الأول منها يتأكد من توفر معلومات في المتغير ثم يقوم بإضافة مؤشر من نوع المتغير cache بإستخدام setLocalData وعند إستخدام الدالة hasLocalData فإنه يعيد القيمة true .
وأدخلنا بعدها قيم للحاوية لتختبرها الدالة الأخرى التي تقوم في حال توفر بيانات بحذفها .
وغير أنه يقوم بالمهمة السابقة يمكنه أيضا يمكنك أيضا إستخدامه في global error-state كما في المكتبة errno (احد مكتبات c القياسية) وذلك لتضن عدم تاثيرك على مسالك أخرى أثناء علاجك لمشكلة ما .

التعامل مع المعالج الرئيسي

عند بدء البرنامج في العمل سوف يكون هناك مسلك واحد فقط يقوم بمعالجة البرنامج وهذا المسلك هو المسلك الرئيسي للبرنامج وهو يعالج Qapplication او QcoreApplication ويسمح بإستدعاء الدالة exec() عليها وبعد إستدعائها فهو ينتظر ويعالج وفقا للأحداث ولتستطيع أن تجلبه عليك بإستخدام الشفرة التالية

```
QApplication::instance()->thread()
```

.

كيف تربط بين المسالك الفرعية والرئيسي؟

لكي تربط بين المسالك الفرعية والمسلك الرئيسي عليك بإستخدام دالة الربط بين الكائنات QObject::connect() بإستخدام الإشارات والمستقبلات (signal-slot) وهي عادة تعمل بشكل متزامن ولكن في حال كان التبادل الإشارات والمستقبلات بين مسالك

مختلفة فهي تعمل بشكل غير متوافق وتسطف في طابور الى ان يحن وقتها ليستقبلها المستقبل (slots) وهذا يعني ان الإشارات سوف تبعث قبل او اثناء تنشيط المستقبل لذلك قد تحتاج لإستخدام الوسيط الخامس من الدالة connect وهي تتلقى طريقة تعامل الإتصال بين الإشارات والمستقبلات ويشملها ثابت تعدادي وحيد وهو connectionType لتجعل الإشارات في حالة اصطفاف.

مثال :

```
1. class TextThread : public QThread
2. {
3. Q_OBJECT
4. public:
5. TextThread( const QString& text );
6. void run();
7. void stop();
8. signals:
9. void writeText( const QString& );
10.private:
11.QString m_text;
12.bool m_stop;
13.};
```

الهدف الرئيسي من هذا الصف هو ارسال الإشارات التي تحوي النص الى المستقبلات التي هي في مسلك آخر والإشارة هنا هي writeText وكلا من run و stop عبارة عن دوال عامة الأول يحوي الشفرة والثاني سوف نستخدمه لإيقاف البرنامج .

```
//=========================================
===//
1. class TextDevice : public QThread
2. {
3. Q_OBJECT
4. public:
5. void run();
6. void stop();
7. public slots:
8. void write(QString text );
9. private:
10.int m_count;
11.QMutex m_mutex;
12.};
```

وبالمقابل فإن هذا الصف يقوم بإستقبال الإشارات بواسة دالته write وبقية الدوال شبيه بالصف السابق.

```
//=========================================
===//
```

```
1. TextThread::TextThread( const QString& text ) :
   QThread()
2. {
3. m_text = text;
4. m_stop = false;
5. }
6. void TextThread::stop()
7. {
8. m_stop = true;
9. }
10.void TextThread::run()
11.{
12.while( !m_stop )
13.{
14.emit writeText( m_text );
15.sleep( 1 );
16.}
17.}
```

دالة البناء هي مجرد تهيئة للمتغيرات بإعطائهم قيم والدالة stop تقوم بتحويل المتغير m_stop الى القيمة true لكي تتوقف عملية تكرار البرنامج .
الدالة run تقوم بعملية التكرار مالم تكن قيمة المتغير m_stop تساوي true وعندها فقط سوف يتوقف عن التكرار وفي كل مرة يتكرر فيها يقوم بإرسال إشارة والتي سوف نربطها في الدالة main مع المستقبل .

//==========================//

```
1. void TextDevice::run()
2. {
3. exec();
4. }
5. void TextDevice::stop()
6. {
7. quit();
8. }
9. void TextDevice::write( QString text )
10.{
11.QMutexLocker locker( &m_mutex );
12.qDebug() << QString( "Call %1: %2" ).arg( m_count+
   + ).arg( text );
13.}
```

الدالة run تعمل مالا نهاية مالم تقم أنت بإقفال المسلك وهذا ماسوف يقوم به الدالة stop التي استخدمت الدالة quit في الخروج من المسلك ومن بعدها قمنا بتعريف الدالة write وهي من سوف تستقبل إشارات الصف السابق وتقفل على نفسها ال mutex لكي لا تسمح بإستخدام الدالة مع عدة إشارات في وقت واحد ويقوم بطباعة رقم النص والنص نفسه في الشاشة console .

```
//=========================================
=======//
```

```
1. int main( int argc, char **argv )
2. {
3. QApplication app( argc, argv );
4. TextDevice device;
5. TextThread foo( "Foo" ), bar( "Bar" );
6. QObject::connect( &foo, SIGNAL(writeText(const
   QString&)),&device, SLOT(write(const QString&)) );
7. QObject::connect( &bar, SIGNAL(writeText(const
   QString&)),&device, SLOT(write(const QString&)) );
8. foo.start();
9. bar.start();
10.device.start();
11.QMessageBox::information( 0, "Threading", "Close me
   to stop!" );
12.foo.stop();
13.bar.stop();
14.device.stop();
15.foo.wait();
16.bar.wait();
17.device.wait();
18.return 0;
19.}
```

قمنا بالتصريح عن متغير الطباعة ومغيرين للإشارات كلا منهم يحوي نص مختلف ثم قمنا بربط الإشارات بالمستقبلات لنبدأ بعملية بدء تشغيل المسالك جميعها ونرسل رسالة لإيقاف المسلك الرئيسي بينما يبقى الباقي في عمل دؤوب في الإرسال والطباعة وبمجرد أن تقفل الرسالة يستمر البرنامج في العمل ويقفل جميع المسالك وينتظرهم حتى يفرغوا ثم يقوم بالخروج من البرنامج كليا .
وكان ناتج العملية السابقة :

```
"Call 0: Foo"
"Call 1: Bar"
"Call 2: Bar"
"Call 3: Foo"
"Call 4: Bar"
"Call 5: Foo"
```

الى ان يتوقف البرنامج

كيف تنقل عنصر من مسلك الى مسلك آخر؟

عندما تنشئ عنصر فإنه يعمل مع المسلك الذي انشئ فيه وفي حال اردت نقله من مسلك الى مسلك آخر فأنت بحاجة لإستخدام أحد دوال الصف Qopject ومنها الدالة moveToThread وهي تستقبل مؤشر لعنصر .

ارسال الصفوف بين المسالك

في حال قمت بإرسال فئات Qstring او Qvariant بواسطة الإشارات المصطفة فأن في الغالب لست بحاجة لعمل إضافي ولكن في بعض الأعمال قد تحتاج الى نقل أنواع غير معرفة لذا meta type في هذه الحالة أنت بحاج الى تسجيل هذا النوع من ضمن الأنواع المعرفة لذا meta type .
حاول وقم بإرسال بيانات خاصة الى صف الإشارات وسوف تصدر لك هذه التعليمات

```
QObject::connect: Cannot queue arguments of type
'TextAndNumber'
(Make sure 'TextAndNumber' is registed using
qRegisterMetaType().)
QObject::connect: Cannot queue arguments of type
'TextAndNumber'
(Make sure 'TextAndNumber' is registed using
qRegisterMetaType().)
```

السبب من وراء هذا كله ان الإشارات عندما يتم ارسالها فإنها تصطف في وسائط وهذه الوسائط تقوم بإستقبال محتوى الإشارا لتضيفها الى المستقبلات في وقت لاحق ولكي تعرف Qt كيف تقوم بهكذا أمر عليك بتسجيل جميع الفئات التي سوف ترسلها داخل meta type بإستخدام الدالة qRegisterMetaType وترسل لها النوع والإسم المعرف لها وهم سوف يرسلوا لك رقم لتعريفها ضمن الجدول Typeوبعد عملية التسجيل أصبح من الممكن لك ان تقوم بإضافة هذه الصفوف الى الإشارات وال مستقبلات وتستطيع بالإضافة الى ماسبق إستخدام الماكرو Q_DECLARE_METATYPE لكي تجعل النوع معروف ويجب أن يصرح عنها تحت أي صف وتستخدم في حال أردت ان تستخدم هذه الأنواع الخاصة داخل Qvariant .

ملاحظة : جميع العناصر المعرفة داخل meta type متوفرة في الثابت التعدادي type المتوفر في الصف QMetaType ويجب ان تتوفر لكل نوع إسم ورقم او اسم فقط وهم سوف يعطوا لك رقم لتعريفها .

مثال :

يهدف هذا المثال ارسال اشارات تحوي نوع جديد الى مستقبلات تقوم بطباعتها وهذه النوع الجديد به رقم ونص يمكن طباعته .
للبرنامج السابق أضف الشفرات التالية :

```
1. class TextAndNumber
2. {
```

```
3. public:
4. TextAndNumber();
5. TextAndNumber( int, QString );
6. int number;
7. QString text;
8. };
9. Q_DECLARE_METATYPE( TextAndNumber );
```

غير دالتي البناء فتوفر متغيرين نصي ورقمي يتيح لنا تخزين القيم داخل هذا الصف ثم كما ذكرنا استخدامك للماكرو Q_DECLARE_METATYPE يتيح لك بإستخدام هذا الصف داخل variant .

```
1. int main( int argc, char **argv )
2. {
3. QApplication app( argc, argv );
4. qRegisterMetaType<TextAndNumber>("TextAndNumbe
   r");
5. TextDevice device;
6. TextThread foo( "Foo" ), bar( "Bar" );
7. QObject::connect( &foo,
   SIGNAL(writeText(TextAndNumber)),
8. &device, SLOT(write(TextAndNumber)) );
9. QObject::connect( &bar,
   SIGNAL(writeText(TextAndNumber)),
10.&device, SLOT(write(TextAndNumber)) );
11.……….
12.}
```

الدالة الرئيسية main في برنامجنا السابق فقط قم بتغير المحتوى الأعلى قبل عملية ابدء بهذا المحتوى وكما تلاحظ قمنا بتسجيل النوع ليصبح من ضمن الأنواع التي يمكن ارسالها في طوابير الإشارات qRegisterMetaType وقمنا بعمليات اتصال لتصبح الإشارات ترسل أنواع جديدة بعد أن سجلناها على أنها نوع معروف .

```
1. void TextDevice::write( TextAndNumber tan )
2. {
3. QMutexLocker locker( &m_mutex );
4. qDebug() << QString( "Call %1 (%3): %2" )
5. .arg( m_count++ )
6. .arg( tan.text )
7. .arg( tan.number );
8. }
```

قم بإستبدال الدالة write بهذه الدالة في البرنامج السابق وهذه الدالة هي عبارة عن مستقبل إشارات والإشارة TextAndNumber سوف تحتوي على متغير من النوع الجديد لنقوم بعد ذلك بطباعة محتواه الرقمي والنصي .

```
1. void TextThread::run()
```

```
2. {
3. while( !m_stop )
4. {
5. emit writeText( TextAndNumber( m_count++,
   m_text ) );
6. sleep( 1 );
7. }
8. }
```

وتـستبدلها مكان الدالة run في الصف TextThread والفرق فقط في نوع محتوى الإشارات التي سوف يتم إرسالها .

وبعد إعادة تنفيذ البرنامج كانت المخرجات كالتالي

```
"Call 0 (0): Foo"
"Call 1 (0): Bar"
"Call 2 (1): Bar"
"Call 3 (1): Foo"
"Call 4 (2): Foo"
```

ملاحظات

لقد تعلمنا ان الإشارات في حالة كانت بين عدة مسالك فإنها تستطف في طوابير ويتم أخذ المحتوى منها في وسائط في إنتظار ان يتم تنفيذ المستقبلات لتقوم بإستقبال هذه الإشارات ثم قلنا كيف يمكننا ان نتعامل معها في هذه الحالة وكانت هذه التعليمات جميعها تخبرنا مالذي قد يتغير في حال إعتمادنا على تعدد المسالك ولكن هل هذه هي الشروط الوحيد التي يجب ان نتعامل معها في تعدد المسالك ؟

في الحقيقة الأمر ليس كذلك فهناك شروط أخرى يجب أن تحطاط لأجلها عند إستخدامك للمسالك :

1- لا تستخدم QThread كأب لعنصر إبن .

2- لاتجعل الأحداث في مسالك مختلفة لأن كل مسلك لها عملية تكارها ففي هذه الحالة لايحدث تزامن بين الأحداث مثل ان تجعل المؤقت في مسلك ووضيفته في مسلك آخر .

3- لا تقترب من الأبناء.

ملخص

لقد تعلمنا أنا المعالجات الحديثة أصبحت تقوم بعدة مها في آن واحد فلذلك أستغلينا هذه الفرصة لنقم بصنع عدة مسالك للمعالج بالوراثة من Qthread ثم واجهتنا مشكلة أن هذه المسالك لاتعمل بشكل متزامن فيما بينها ثم تمكنا من حل هذه المشكلة مع QMutex فوجدنا ان هذا الأخير يسمح بمسلك واحد بالدخول والتنفيذ اما الباقي ينتظره فقمنا بإستعمال QreadWriteLock ولكن لم نفكر هل من الممكن أن نجعل هناك نظام رياضي ينظم العمل وشبيه ل mutex ولكن بدون الإقفال على أحد المسالك فوجدنا الحل في الصف Qsemaphore ثم خرجنا من بوابة التزامن والمصادر المشتركة وتعرفنا على بعض الأمور البسيطة مثل المعالج

الرئيسي وأثناء حديثنا عنه واجهتنا مشكلة الإتصال بين هذه المسال كيف نتصل وهي غير متزامنة فقمنا بالتعرف على طابور الإشارات وفي الحقيقة طابور الإشارا هذا يطالب بحقوق أخرى وهي ان يكون محتواه الذي يرسله معرف لكي يستطيع الوسائط ان تلتقطه فقمنا بالتعرف على كل من الماكرو Q_DECLARE_METATYPE والأمر qRegisterMetaType وقلنا أن الأول يجعل الصف معرف لذل variant والآخر يجعلها معرف لذا الإشارات وتعرفنا كيف نستخدمهم ليكون هنا نهاية حديثنا عن هذا الفصل.

أخوكم :محمد العبدلي