Is using a Mutex to prevent multiple instances of the same program from running safe?

Asked 15 years, 9 months ago Modified 3 years, 8 months ago Viewed 69k times

47

I'm using this code to prevent a second instance of my program from running at the same time, is it safe?

```
Mutex appSingleton = new System.Threading.Mutex(false,
"MyAppSingleInstnceMutx");
if (appSingleton.WaitOne(0, false)) {
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.Run(new MainForm());
    appSingleton.Close();
} else {
    MessageBox.Show("Sorry, only one instance of MyApp is allowed.");
}
```

I'm worried that if something throws an exception and the app crashes that the Mutex will still be held. Is that true?

c# winforms mutex single-instance

Share
Improve this question
Follow

edited Aug 19, 2013 at 22:22
Blorgbeard
103k ● 50 ● 235 ● 276

asked Mar 14, 2009 at 18:53
Malfist
31.7k ● 64 ● 185 ● 271

3 Please add a related link to stackoverflow.com/questions/229565/… – Sam Saffron May 4, 2009 at 12:55

11 Answers

Sorted by: Highest score (default)

67

It is more usual and convenient to use Windows events for this purpose. E.g.

```
static EventWaitHandle s_event ;

bool created ;
s_event = new EventWaitHandle (false,
    EventResetMode.ManualReset, "my program#startup", out created) ;
if (created) Launch () ;
else         Exit   () ;
```

When your process exits or terminates, Windows will close the event for you, and destroy it if no open handles remain.

Added: to manage sessions, use `Local\` and `Global\` prefixes for the event (or mutex) name. If your application is per-user, just append a suitably mangled logged-on user's name to the event name.

Share

Improve this answer

Follow

edited Jul 13, 2016 at 6:38

answered Mar 14, 2009 at 19:00

Anton Tykhyy

20k ●6 ●56 ●58

Right. This is much cleaner and safer than using a mutex, as it avoids the need to synchronize on the mutex, and trying to handle all kind of race conditions... Note that it isn't even necessary (or useful) to wait on the event. Essentially, you just check that it exists. – oefe Mar 15, 2009 at 11:14

12 You can use the mutex in the same way. That is, just use the createdNew flag to determine if yours is the first instance. No need to synchronize on the mutex. So the EventWaitHandle approach is no better or worse than the Mutex approach. You could use Semaphore, too. – Jim Mischel Mar 16, 2009 at 15:13

3 This so much simpler than 99% of the solutions out there and yet it's the only time I've seen it. Thank you! – Jamona Mican Aug 29, 2012 at 17:19

Could achieve this in a simpler way: using (new EventWaitHandle(false, EventResetMode.ManualReset, "my program#startup", out var created)) { if (created) { Launch(); } } – m3z Sep 4, 2018 at 10:13

2 While more convenient and maybe even safer and cleaner, this approach is not cross-platform compatible. When using .NET Core to develop apps, this code will throw an exception. (Source) – Sandro Jul 18, 2019 at 9:02

43

In general yes this will work. However the devil is in the details.

Firstly you want to close the mutex in a `finally` block. Otherwise your process could abruptly terminate and leave it in a signaled state, like an exception. That would make it so that future process instances would not be able to start up.

Unfortunately though, even with a `finally` block you must deal with the potential that a process will be terminated without freeing up the mutex. This can happen for instance if a user kills the process through TaskManager. There is a race condition in your code that would allow for a second process to get an `AbandonedMutexException` in the `WaitOne` call. You'll need a recovery strategy for this.

I encourage you to read up on the details of the Mutex class. Using it is not always simple.

Expanding upon the race condition possibility:

The following sequence of events can occur which would cause a second instance of the application to throw:

1. Normal process startup.
2. Second process starts up and aquires a handle to the mutex but is switched out before the `WaitOne` call.
3. Process #1 is abruptly terminated. The mutex is not destroyed because process #2 has a handle. It is instead set to an abandoned state.
4. The second process starts running again and gets an `AbanonedMutexException`.

Share

Improve this answer

Follow

edited Nov 18, 2015 at 17:30

the Tin Man

160k ● 44 ● 221 ● 306

answered Mar 14, 2009 at 18:57

JaredPar

753k ● 151 ● 1.3k ● 1.5k

2 If his code is the only one creating the mutex, it will be destroyed when the process terminates, hence the create + waitOne will succeed. – Michael Mar 14, 2009 at 18:59

1 @Michael, if the mutex is not explicitly released it will be put into an abandoned state – JaredPar Mar 14, 2009 at 19:03

@Michael: Try my code in my answer and see. The process exits, the mutex is destroyed. It gets recreated when you re-run the exe. – Michael Mar 14, 2009 at 19:06

@Michael you're ignoring the race condition possibility. It's possible for a second process to start, grad the mutex, first process dies, abandonedmutex exception is thrown. – JaredPar Mar 14, 2009 at 19:07

@JaredPar: True, if the original process throws an exception between his code creating the mutex and the WaitOne, this could happen. My comments are only relevent to the generic process throwing an exception case. – Michael Mar 14, 2009 at 19:11

12

You can use a mutex, but first make sure that this is really what you want.

Because "avoiding multiple instances" is not clearly defined. It can mean

1. Avoiding multiple instances started in the same user session, no matter how many desktops that user session has, but allowing multiple instances to run concurrently for different user sessions.
2. Avoiding multiple instances started in the same desktop, but allowing multiple instances to run as long as each one is in a separate desktop.
3. Avoiding multiple instances started for the same user account, no matter how many desktops or sessions running under this account exist, but allowing multiple

instances to run concurrently for sessions running under a different user account.

4. Avoiding multiple instances started on the same machine. This means that no matter how many desktops are used by an arbitrary number of users, there can be at most one instance of the program running.

By using a mutex, you're basically using the define number 4.

Share Improve this answer Follow

answered Mar 14, 2009 at 19:02

Stefan
43.5k ● 10 ● 76 ● 117

I would like to mention that by using named mutex one typically uses define number 1 since by default mutex will reside at session namespace. – user7860670 Jan 25, 2018 at 22:06

11

I use this method, I believe it to be safe because the Mutex is destroyed if no long held by any application (and applications are terminated if if they can't initially create the Mutext). This may or may not work identically in 'AppDomain-processes' (see link at bottom):

```
// Make sure that appMutex has the lifetime of the code to guard --
// you must keep it from being collected (and the finalizer called, which
// will release the mutex, which is not good here!).
// You can also poke the mutex later.
Mutex appMutex;

// In some startup/initialization code
bool createdNew;
appMutex = new Mutex(true, "mutexname", out createdNew);
if (!createdNew) {
  // The mutex already existed - exit application.
  // Windows will release the resources for the process and the
  // mutex will go away when no process has it open.
  // Processes are much more cleaned-up after than threads :)
} else {
  // win \o/
}
```

The above suffers from from notes in other answers/comments about malicious programs being able to sit on a mutex. Not a concern here. Also, unprefixed mutex created in "Local" space. It's probably the-right-thing here.

See: http://ayende.com/Blog/archive/2008/02/28/The-mysterious-life-of-mutexes.aspx -- comes with Jon Skeet ;-)

Share Improve this answer Follow

answered Apr 13, 2010 at 6:36

user166390

3

On Windows, terminating a process has the following results:

- Any remaining threads in the process are marked for termination.
- Any resources allocated by the process are freed.
- All kernel objects are closed.
- The process code is removed from memory.
- The process exit code is set.
- The process object is signaled.

Mutex objects are kernel objects, so any held by a process are closed when the process terminates (in Windows anyway).

But, note the following bit from the CreateMutex() docs:

> If you are using a named mutex to limit your application to a single instance, a malicious user can create this mutex before you do and prevent your application from starting.

Share Improve this answer Follow

answered Mar 14, 2009 at 19:01
Michael Burr
340k ● 52 ● 548 ● 769

2 Mutex is a *shared* resource and will only be destroyed when all processes holding the resource are stopped. There is a race condition in his code that allows for an AbandonedMutexException to be thrown at the WaitOne call. – JaredPar Mar 14, 2009 at 19:14

1 My apologies - my answer was geared for native code - I should have read more closely. However, the CLR will release a Mutex held by a thread when a thread dies. An when a process is terminated, Windows will *close* any held mutexes (which will release them). Closing a mutex might not destroy it. – Michael Burr Mar 14, 2009 at 20:33

3

Yes, it's safe, I'd suggest the following pattern, because you need to make sure that the `Mutex` gets always released.

```
using( Mutex mutex = new Mutex( false, "mutex name" ) )
{
    if( !mutex.WaitOne( 0, true ) )
    {
        MessageBox.Show("Unable to run multiple instances of this program.",
                        "Error",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
    }
    else
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new MainForm());
    }
}
```

Share
Improve this answer
Follow

edited Sep 8, 2015 at 7:44
slavoo
6,036 ● 64 ● 39 ● 41

answered Mar 14, 2009 at 19:02
arul
14.1k ● 1 ● 59 ● 78

1 At least on Windows (probably on Linux, too), process termination automatically releases all held synchronization objects. – P Daddy Mar 14, 2009 at 19:47

2

Here's the code snippet

```
public enum ApplicationSingleInstanceMode
{
    CurrentUserSession,
    AllSessionsOfCurrentUser,
    Pc
}

public class ApplicationSingleInstancePerUser: IDisposable
{
    private readonly EventWaitHandle _event;

    /// <summary>
    /// Shows if the current instance of ghost is the first
    /// </summary>
    public bool FirstInstance { get; private set; }

    /// <summary>
    /// Initializes
    /// </summary>
    /// <param name="applicationName">The application name</param>
    /// <param name="mode">The single mode</param>
    public ApplicationSingleInstancePerUser(string applicationName,
ApplicationSingleInstanceMode mode =
ApplicationSingleInstanceMode.CurrentUserSession)
    {
        string name;
        if (mode == ApplicationSingleInstanceMode.CurrentUserSession)
            name = $"Local\\{applicationName}";
        else if (mode ==
ApplicationSingleInstanceMode.AllSessionsOfCurrentUser)
            name = $"Global\\{applicationName}{Environment.UserDomainName}";
        else
            name = $"Global\\{applicationName}";

        try
        {
            bool created;
            _event = new EventWaitHandle(false, EventResetMode.ManualReset,
name, out created);
            FirstInstance = created;
        }
        catch
        {
        }
    }

    public void Dispose()
    {
        _event.Dispose();
    }
}
```

Share Improve this answer Follow

answered Jul 13, 2016 at 20:06

Siarhei Kuchuk

5,476 ● 1 ● 30 ● 31

1

Here is how I have approached this

In the Program class:

1. Get a System.Diagnostics.Process of YOUR application using Process.GetCurrentProcess()
2. Step through the collection of open processes with the current name of your application using Process.GetProcessesByName(thisProcess.ProcessName)
3. Check each process.Id against thisProcess.Id and if an instance is already opened then at least 1 will match name but not Id, otherwise continue opening instance

```
using System.Diagnostics;

.....

static void Main()
{
   Process thisProcess = Process.GetCurrentProcess();
   foreach(Process p in Process.GetProcessesByName(thisProcess.ProcessName))
   {
      if(p.Id != thisProcess.Id)
      {
         // Do whatever u want here to alert user to multiple instance
         return;
      }
   }
   // Continue on with opening application
```

A nice touch to finish this off would be to present the already opened instance to the user, chances are they didn't know it was open, so let's show them it was. To do this I use User32.dll to Broadcast a message into the Windows messaging loop, a custom message, and I have my app listen for it in the WndProc method, and if it gets this message, it presents itself to the user, Form.Show() or whatnot.

Share

Improve this answer

Follow

edited Apr 14, 2021 at 9:15
Pang
10.1k ● 146 ● 85 ● 124

answered Dec 14, 2015 at 18:21
Wanabrutbeer
697 ● 6 ● 12

0

If you want to use a mutex-based approach, you should really use a local mutex to [restrict the approach to just the curent user's login session](). And also note the other important caveat in that link about robust resource disposal with the mutex approach.

One caveat is that a mutex-based approach doesn't allow you to activate the first instance of the app when a user tries to start a second instance.

An alternative is to PInvoke to FindWindow followed by SetForegroundWindow on the first instance. Another alternative is to check for your process by name:

```
Process[] processes = Process.GetProcessesByName("MyApp");
if (processes.Length != 1)
{
    return;
}
```

Both of these latter alternatives have a hypothetical race condition where two instances of the app could be started simultaneously and then detect each other. This is unlikely to happen in practice - in fact, during testing I couldn't make it happen.

Another issue with these two latter alternatives is that they won't work when using Terminal Services.

Share

Improve this answer

Follow

edited Mar 15, 2009 at 10:38

answered Mar 14, 2009 at 19:17

HTTP 410

17.6k ● 14 ● 79 ● 129

Checking the process table suffers from a race condition. If two instances are started at the same time, both will report a pre-existing application. Also, all you have to do to defeat it is make a copy of the executable. FindWindow suffers from the same race condition. – Jim Mischel Mar 14, 2009 at 21:28

I changed my answer to respond to your comments. As for "defeating" the check by making a copy of the executable. the OP almost certainly wasn't asking about a deliberate attempt to foil the check. – HTTP 410 Mar 15, 2009 at 10:40

2 I, on the other hand, HAVE encountered the race condition when using FindWindow. I've never used the process table approach, but one thing I've discovered is that if something can happen, it eventually will. Usually at the most inopportune time. – Jim Mischel Mar 16, 2009 at 15:15

0

Use app with timeout and security settings with avoiding AbandonedMutexException. I used my custom class:

```
private class SingleAppMutexControl : IDisposable
    {
        private readonly Mutex _mutex;
        private readonly bool _hasHandle;

        public SingleAppMutexControl(string appGuid, int
waitmillisecondsTimeout = 5000)
        {
            bool createdNew;
            var allowEveryoneRule = new MutexAccessRule(new
SecurityIdentifier(WellKnownSidType.WorldSid, null),
                MutexRights.FullControl, AccessControlType.Allow);
            var securitySettings = new MutexSecurity();
            securitySettings.AddAccessRule(allowEveryoneRule);
            _mutex = new Mutex(false, "Global\\" + appGuid, out createdNew,
securitySettings);
            _hasHandle = false;
            try
            {
                _hasHandle = _mutex.WaitOne(waitmillisecondsTimeout, false);
                if (_hasHandle == false)
                    throw new System.TimeoutException();
            }
            catch (AbandonedMutexException)
            {
                _hasHandle = true;
            }
        }

        public void Dispose()
        {
            if (_mutex != null)
            {
                if (_hasHandle)
                    _mutex.ReleaseMutex();
                _mutex.Dispose();
            }
        }
    }
```

and use it:

```
    private static void Main(string[] args)
    {
        try
        {
            const string appguid = "{xxxxxxxx-xxxxxxxx}";
            using (new SingleAppMutexControl(appguid))
            {
                //run main app
                Console.ReadLine();
            }
        }
        catch (System.TimeoutException)
        {
            Log.Warn("Application already runned");
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Fatal Error on running");
        }
    }
```

Share

Improve this answer

Follow

edited Jul 28, 2015 at 8:56

answered Jul 28, 2015 at 6:45

vivlav

11 ● 2

0

It is important to remember that this mutexes\events can be easily seen via programs like handle.exe from Microsoft sysinternals and then if constant name used someone evil may use this to prevent your app from starting.

Here is some quote from Microsoft recommendations on safe alternatives:

> However, a malicious user can create this mutex before you do and prevent your application from starting. To prevent this situation, create a randomly named mutex and store the name so that it can only be obtained by an authorized user. Alternatively, you can use a file for this purpose. To limit your application to one instance per user, create a locked file in the user's profile directory.

Taken from here : https://learn.microsoft.com/en-us/sysinternals/downloads/handle

Share Improve this answer Follow

answered Apr 10, 2020 at 16:16

David Constantine
587 ● 5 ● 10