Lets start with below exception details:

Date : 06/27/2012 03:25:28 PM;
System.Threading.ThreadAbortException: Thread was being aborted.
   at System.Threading.Thread.AbortInternal()
   at System.Threading.Thread.Abort(Object stateInfo)
   at System.Web.HttpResponse.End()
   at System.Web.HttpResponse.Redirect(String url, Boolean endResponse, Boolean permanent)
   at System.Web.HttpResponse.Redirect(String url)
   at Redirect(String UrlToRedirect)

So, what went wrong!

Microsoft says:
"Thread.Abort() Raises a ThreadAbortException in the thread on which it is invoked, to begin the process of terminating the thread. Calling this method usually terminates the thread."

In code, i saw one simple line there, some thing like below:

```
if(!IsAllowed())
{
      Response.Redirect("Login.aspx");
}
```

According to stack trace:
Response.Redirect() >>> Resposne.End() >>> Thread.Abort() >>> ThreadAbortException.

Now Question comes:

Can we prevent this exception to occur?
Answer is Yes.

```
Response.Redirect("Login.aspx", false); //Pretty simple!
```

In above line we passed parameter as false. Actually the function takes one optional argument as boolean endResponse. The Response.Redirect() will not call the Response.End() and Thread.Abort() method will not be called and no ThreadAbortException will occur.

Now look at below code:

```
public partial class MemberHome : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (IsAllowed())
        {
            if (!IsPostBack)
            {
                // User seems from valid source. So, do required things here.
            }
        }
        else
        {
            Response.Redirect("Login.aspx", false);
        }
    }
```

```
    private void btnDeleteRecord_Click()
    {
        string FilePathToDelete = Server.MapPath("ImpornatFile.doc");
        if (File.Exists(FilePathToDelete))
        {
            File.Delete(FilePathToDelete);
        }
    }

    private bool IsAllowed()
    {
       // Check what you need to check and return boolean value according to that.
       // For demo purpose, just checking session
       if (Session["User"] == null)
       {
           return false;
       }
       else
       {
           return true;
       }
    }
}
```

Now take this case:

- Suppose user come to MemberHome page and and open different pages in different tabs.
- After some time signed out in one of the tab or leaves it inactive for couple of hours. So, his/ her session will automatically get inactive after session time out(*according to the timeout set into web server*).
- Now, session got timed out.
- But, if I will hit the delete button on the UI. Then!!!
- This will do a postback and first PageLoad event will get called and Is Allowed() function will do their job and will return false and then Response.Redirect will redirect user to login page, but as we mentioned endResponse as false, so, delete button click event btnDeleteRecord_Click will get called, and that file will get deleted, even if you were not allowed to do so, without session!

Now, if we remove that false and call Response.Redirect like below in above code statement:

```
Response.Redirect("Login.aspx");
```

And now do the same thing again and you will see this time btnDeleteRecord_Click() did not get called!
But this will generate the ThreaAbortException, which can be expensive!

So, now the question comes after preventing this exception, cant we prevent the next codes to get executed?

Hmmm.... What if we will add one more line after Response.Redirect(Url, false) like below:

```
Response.Redirect("Login.aspx", false);
return;
```

Now seems fine!!!

Do you think so?
Not at all!

Even if Response.Redirect(url, false) is supposed to hault the execution of the current page, the whole Asp.net page life cycle events gets executed here without any interruption.

So, writing return just after the Response.Redirect(url, false) is not enough.

So, what next?

Now look at the below code:

```
public partial class MemberHome : System.Web.UI.Page
{
    bool IsAllowedForOperations = false;

    protected void Page_Load(object sender, EventArgs e)
    {
        IsAllowedForOperations = IsAllowed();

        if (IsAllowedForOperations)
        {
            if (!IsPostBack)
            {
                // User seems from valid source. So, do required things here.
            }
        }
        else
        {
            Response.Redirect("Login.aspx", false);
        }
    }

    private void btnDeleteRecord_Click()
    {
        if (IsAllowedForOperations)
        {
            string FilePathToDelete = Server.MapPath("ImpornatFile.doc");
            if (File.Exists(FilePathToDelete))
            {
                File.Delete(FilePathToDelete);
            }
        }
    }

    private bool IsAllowed()
    {
        // Check what you need to check and return boolean value according to that.
        // For our demo purpose, lets return false
        if (Session["User"] == null)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

Now looks good....

But we will need to check this IsAllowedForOperations value in each and every postback events in all pages!

A more structured way:

Everybody uses a basepage now a days. We could have used basepage for this.
Whenever a postback happens RaisePostBackEvent event get fired first, before calling particular control event(*like button_Click*).

See this code:

```
public class BasePage : System.Web.UI.Page
{
    protected bool IsRedirected = false;

    public BasePage()
    {
    }

    protected void RedirectUser(string url)
    {
        Response.Redirect(url, false);
        IsRedirected = true;
    }

    protected override void RaisePostBackEvent(IPostBackEventHandler sourceControl,
string eventArgument)
    {
        if (!IsRedirected)
            base.RaisePostBackEvent(sourceControl, eventArgument);
    }

    protected override void Render(HtmlTextWriter writer)
    {
        if (!IsRedirected)
            base.Render(writer);
    }
}

public partial class MemberHome : BasePage
{
    bool IsAllowedForOperations = false;

    protected void Page_Load(object sender, EventArgs e)
    {
        IsAllowedForOperations = IsAllowed();

        if (IsAllowedForOperations)
        {
            if (!IsPostBack)
            {
                // User seems from valid source. So, do required things here.
            }
        }
        else
        {
            RedirectUser("Unauhorized.aspx");
            return;
        }
    }

    private void btnDeleteRecord_Click()
    {
```

```
        string FilePathToDelete = Server.MapPath("ImpornatFile.doc");
        if (File.Exists(FilePathToDelete))
        {
            File.Delete(FilePathToDelete);
        }
    }

    private bool IsAllowed()
    {
        // Check what you need to check and return boolean value according to that.
        // For our demo purpose, lets return false
        if (Session["User"] == null)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

We should also override the Render() event method so that, the Response is not unnecessarily being written to the output stream when we call Response.Redirect(Url, false):

Now some points on the above approach:

- Here the current Thread is not being terminated immediately. It stays alive unnecessarily. It's not a big problem. As, It will execute and complete page life cycle and will return to the thread pool.
- It put one condition to write a return statement after the BasePage.Redirect() method, which may not be the straight forward way to follow everywhere.

Now see below code:

```
    protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsAllowedToViewPage())
            {
                return;
            }
        }

        private bool IsAllowedToViewPage()
        {
            bool IsAllowedToView = false;

            if (!IsAllowedToView)
            {
                RedirectUser("Unauthorized.aspx");
                return false;
            }

            return IsAllowedToView;
        }
```

Here we are returning false from **IsAllowedToViewPage() function** to the the caller. And we will need to matain the chain until we reach the page life cycle method which originally got called.
It does not look conformtable to me.

- There is also a risk of missing to write the return statement or breaking the chain for return statement. Which can be the responsible of getting unwanted things to occur.

Now think in other way:

Suppose we write:

```
// without passing false and without writing in try/ catch
Response.Redirect(url);
```

And we handle it in global.asax file in Application_Error event

OR

Create one RedirectUser function in BasePage like below and call it from everywhere:

```
public void RedirectUser(string Url)
 {
     try
     {
         Response.Redirect(Url);
     }
     catch (System.Threading.ThreadAbortException)
     { }
     catch (Exception Ex)
     {
         ExceptionLog.LogErrorDetails(Ex.ToString());
     }
 }
```

You might ask me "What!!! you are letting an exception to be thrown and catching it".

But I am not suggesting any of the above approach. I just tried to put different ways related to Response.Redirect() function, which I came through while doing google and exploring . You will need to be think to choose right one at right place!

If somewhere my description looks confusing, try to create a demo test page and implement different cases/ approaches there. Then it will make more sense and will be able to explore new things.

I know there must be many other approaches, please suggest if you have...