Accessing same resource(s): Reading and writing the same data - it's a "race" to see which one "wins". You don't need to be specifically doing threaded work to have to worry about race conditions -- you just need to be in an environment where multiple processes might be competing for the same resource, or accessing the same data (possibly in varying methods).

Edge cases often times cause race conditions (FB example), but edge cases (extreme maximum or minimum, or unexpected outcomes) can happen on their own, too.

Need "atomic" operations to avoid race conditions (whenever possible)... use locks (e.g. mutex or semaphores) as appropriate. Consider this routine:

```
sub increment_db_counter {
    local current_val;
    read_from_db(current_val);
    current_val++;
    write_to_db(current_val);
}
```

This works fine alone, but imagine multiple threads running at the same time, where you end up with interleaved commands ('cv' is local to each thread): (assume the initial value in DB is "5")

```
[T1] read_from_db(cv);     ((T1)cv=5)
[T1] cv++;                 ((T1)cv=6)
[T2] read_from_db(cv);     ((T2)cv=5)
[T1] write_to_db(cv);      (db_value = 6)
[T2] cv++;                 ((T2)cv=6)
[T2] write_to_db(cv);      (db_value = 6) (!!)
```

The value in the DB *should* be incremented twice, but since each thread is individually operating on the database, that doesn't happen. You'd need to ensure this routine is "thread safe", or perhaps create a separate thread that does only database operations and can pipeline them. (Which would be better, and why? :))

Databases, themselves, generally take their own precautions to prevent data from getting mangled by doing locking of either the entire table, or of the row that is having the operation performed on it. This only prevents the simultaneous changing of data within the database, and doesn't account for the example case, above.

Filesystems are another weak point, and unless each process uses file locking (lockf(), flock(), fcntl()), you can potentially clobber a file. A common case this can happen is in standard UNIX mailboxes. The standard format is called "mbox", where all the email messages are strung together in one giant file, using a keyword ("From ") to separate them. If you're reading your mail from your UNIX shell and decide to delete a message just as another new message is coming in (new messages are appended to the file), the process that is deleting the mail can potentially cause the new, incoming mail to "vanish". Something similar can happen if you're trying to read mail from two places at the same time, and the processes are trying to do things like update the flags on messages, etc. One common "solution" to almost completely avoid this is to use the "maildir" format, which utilizes a separate file for each message. (Assuming you can't trust all processes to use proper file locking when manipulating the mailbox file).

One issue I ran into when writing some monitoring software was a race condition when generating graphs of data. I had a perl CGI script that would read data from a database and generate a pretty graph (or network traffic, or server load, etc) on-the-fly, based on things like timeframe and device+resource being viewed. This was all fine and dandy (I even added some smarts so that it wouldn't re-generate the image more often than once every 30 sec or so, since the back-end data didn't actually get collected more often than that), but things fell apart when we had more than a few people trying to view the same graphs at the same time. Since each user's HTTP session spawned an individual instance of the script, there could be two scripts trying to write out an image file at the same time, and Bad Things™ would happen. I solved this fairly simply by using a temporary file (carefully selected naming*) and then moving this temp file to the actual image file name with a call to rename() (essentially the 'mv' command in UNIX).

Along these lines also lies a classic example of how filesystem data may change between the time you check/examine it, and then try to operate on it. Just because you initially check to see if you CAN operate on a file, doesn't mean you shouldn't catch an exception if you then later try to actually operate on it -- it might have changed in the meantime. This can also lead to possible security issues when privileged users try to create files (even if they tried to check to ensure they were not stepping on an existing file).

Distance-induced network latency for large systems can also wreak havoc w/o safety precautions in place. Classic example is IRC (Internet Relay Chat), where the original system could easily have multiple conflicting events happen at the same time, causing channel desync (one set of servers does not agree with another set of servers as to who has operator privileges, or is even on the channel, etc.

Pitfalls you can fall into while trying to avoiding race conditions include deadlock (two or more processes waiting on each other or the same resource) -- but that can be avoided or at least resolved in a variety of ways, like preemption -- but watch out for "livelock"... it's often best to introduce randomness into the resolution to help avoid these situations.

Sometimes, when trying to add some security to your software, you can be unexpectedly affected by race conditions or edge cases (or both). At Blizzard, we were always wary of user input, and if something strange seemed to be input, it would usually result in the user getting booted. However, this isn't always the right solution, since network latency can cause normal actions to appear subversive. Case in point was when Battle.net got reports of users getting kicked off for sending normal chat messages. Turns out these were messages that were sent just as their friend logged off, and so the logic saw this and thought "Oh, this person is trying to be bad and message an user that is offline", and instead of just dropping the message on the floor, it booted the sender.

Example of a race condition in gaming (***World of Warcraft***): Paladin vs. Shaman.

Paladin has helpful buffs (Increase attack power, increase health, decrease incoming damage, etc)
Shaman does, too (but it's irrelevant in this scenario) - but she also has an offensive ability that strips helpful buffs from her target.

The Paladin and his friend happened to be fighting a few other people and they manage to kill one or two of them, but is low on health. As the Shaman and her friends approaches, the Paladin knows he has to heal up, so he activates his immunity shield ("Divine Shield") so that he can heal himself, uninterrupted. The Shaman anticipated this and starts to use her 'Purge' ability, and manages to use it at the same time as the Paladin activated his Shield. The outcome was unexpected, as the Paladin's shield seemed to completely fail, and he was quickly killed by the enemy. This is what happened:

The Paladin's shield is actually comprised of a few steps:

1) Strip all other harmful debuffs from the Paladin.
2) Apply the "shield" buff that prevents incoming damage.
3) Apply a DEBUFF to the Paladin that would prevent a subsequent use of the shield.
4) Apply a hidden immunity to the shield buff that prevents ANY action against the paladin.

The Shaman's offensive ability is simply:

1) Strip one helpful buff from an enemy
2) Strip another helpful buff from an enemy

What happens on the server is that there is a FIFO queue of these events, and each set of events are non-atomic, so they get interleaved. Thus, the server ends up processing them as:

```
Paladin (1)
Paladin (2)
Shaman (1)
Shaman (2)
Paladin (3)
```

(And the application of Paladin(4) never happens, because that actually checks to see if (2) is still there)

These situations are exacerbated by network latency, which can cause upwards of 300+ms of delay between what *I* see on my screen versus what *you* see on your screen. Meaning, if you see me start casting a short cast-time spell, and try to interrupt me, I might actually have completed it before the server gets your command to interrupt me (or, if I have more local latency, I might seemingly have completed my cast, but it doesn't actually "work" because the server determined that I was interrupted).