Linux/Windows concurrency

About threads, mutexes, and events

2024/05/09, version 1.0

by Ingo A. Kubbilun (Germany)

Abstract

*The document in-hand explains how to write concurrent software for MS Windows and Linux systems using **threads**, **mutexes**, and **events**.*

*Especially events, often implemented using the POSIX condition variables on Linux, are a source of problems. In this document, we use **Linux event file descriptors** provided by the Linux kernel instead.*

This ensures that you can write multi-threaded (concurrent) software running seamlessness on both operating systems.

Document revision

Version:	Date:	Author:	Comments / changes:
0.1	05/04/2024	Ingo A. Kubbilun	document creation
1.0	05/09/2024	Ingo A. Kubbilun	1st published version

Table of Contents

DOCUMENT REVISION I

TABLE OF CONTENTS II

LIST OF TABLES III

LIST OF FIGURES IV

1 INTRODUCTION 5

1.1 SIMPLIFICATIONS IN THIS DOCUMENT 5

2 THREADS 6

2.1 A NOTE ON UNIX SIGNALS 6
2.2 THREAD CREATION AND TERMINATION 6
2.2.1 MS Windows 6
2.2.2 Linux 7

3 MUTEXES 8

3.1 SIDEKICK #1: MUTEXES VERSUS SEMAPHORES 8
3.2 SIDEKICK #2: DEADLOCKS 8
3.3 USAGE OF MUTEXES 8
3.3.1 MS Windows 8
3.3.2 Linux 9

4 EVENTS 10

4.1 MS WINDOWS 10
4.2 WINDOWS EVENTS VERSUS POSIX CONDITION VARIABLES 10
4.2.1 The 'lost event' problem (POSIX-specific) 11
4.3 LINUX EVENT FILE DESCRIPTORS (EVENTFD) 11
4.3.1 Note on the 64bit eventfd counter 11
4.3.2 eventfd usage 11
4.3.3 Example (semaphore semantics) 12

5 CONCLUSION 14

A APPENDIX 15

A.1 SYNCHRONIZATION USING VOLATILE VARIABLES 15
A.1.1 The C compiler keyword 'volatile' 15
A.1.2 Simple synchronization variables (INFORMATIONAL) 15

List of Tables

List of Figures

1 Introduction

In former times, a computer came with one single, maybe two CPUs. Today, a single CPU provides multiple CPU cores, sometimes combined with some kind of hyper-threading, too.

The vast majority of the software is still single-threaded. The operation system kernel takes care of the distribution of concurrently running processes among all available CPU execution units (cores, hyper threads, etc.).

Quite often, you have/want to provide concurrent software that runs on MS Windows or Linux based systems, respectively. In fact, this is an easy task if you use the correct synchronization primitives provided by these two operating systems.

This document is about '***intra**-process concurrency/synchronization*' not '***inter**-process concurrency/synchronization*', i.e. we focus on a single process that is split into multiple lightweight processes, called 'threads'.

I do not explain the basics about threads, mutexes, and events. If you are unfamiliar with these concepts, please google them.

Furthermore, I assume that your machine provides multiple CPU cores, maybe hyper-threads, i.e. all threads get distributed among all available 'execution units' running truly in parallel. This includes machines with more than one CPU as well.

MS Windows provides so called '*native* threads' (built in the operating system natively). On Linux, we work with POSIX threads (provided by the shared object `libpthread.so`). Some other UNIX-style operating systems came/come with native threads, too (e.g. the SUN/Oracle Solaris operating system). We focus on Linux here – all UNIX-style operating systems providing POSIX threads should work, too. One exception is the 'event' synchronization mechanism that relies on the so called 'Linux event file descriptors', which are, not-surprisingly, only available on Linux.

Obviously, we use plain C in this document (you may use C++ as well).

1.1 Simplifications in this document

Please do not blame me for some simplifications I am making use of in this document, e.g.:

- (concurrent) memory accesses may access the RAM or a cache line of the CPU cache; in this document, this is just a memory access;
- memory operations are just either reads or writes; of course, this is also more complicated if you think of R-M-V (read-modify-write) instructions; in this document, we just distinguish between reads and writes;
- very limited error checking in the sample C source codes;
- (…)

2 Threads

Threads are the building block of a concurrent process. Without threads, we do not get real concurrency in a process.

Please keep in mind that a started process always comes with one implicit thread, often called the main thread or 'LWP 0' (LWP stands for Light-Weight-Process). Each thread has its own set of CPU registers, CPU flags, instruction pointer, a dedicated stack, and so forth.

2.1 A note on UNIX signals

UNIX signals, e.g. `SIGINT`, `SIGHUP`, `SIGTERM`, `SIGKILL`, `SIGUSR1`, etc. may occur in your process at any time. It is good practice to let just one thread (mostly, the LWP 0) perform all of the required signal handling.

When we start new (POSIX) threads, we will always block all signals at the very beginning of the thread routine to ensure that signals are not handled by '*random*' threads. Nevertheless, you do not need to follow this guideline.

Your code has to be prepared that signals may be raised at any time. If you do not program your signal handlers with the `SA_RESTART` flag, then any system call may be interrupted by a signal. You will get the result code -1 in this case, and the `errno`[1] variable will be set to `EINTR`.

If you have the need to wake-up threads that are currently waiting (blocking) e.g. for an event to occur, then the usage of UNIX signal is a good choice. You may use SIGUSR1 and SIGUSR2 for the cancellation of blocked/waiting threads.

Please note that you can send UNIX signals dedicated to a specific thread using the `pthread_kill` library call (`libpthread.so`).

2.2 Thread creation and termination

2.2.1 MS Windows

The creation of a new thread is very simple. On MS Windows, it works as follows:

```
void *thread_context;
DWORD dwTID = 0;

HANDLE hThread = CreateThread(NULL,0,ThreadRoutine,thread_context,0,&dwTID);
```

This is a very simple call, you may add additional parameters, start the thread in suspended state, etc. The '`void *thread_context`' should be filled with a thread-specific context data structure (in C++, quite often the 'this' pointer is used for this). Moreover, the second parameter (here: zero) can be used to control the stack size of the newly created thread.

The '`ThreadRoutine`' looks like this:

```
DWORD WINAPI ThreadRoutine ( LPVOID lpContext )
{
  …
  return 0;
}
```

`dwTID` is the thread ID. To wait for the thread termination, you have to write something like this:

[1] Please note that errno is MT-safe, i.e. each thread has its own copy of the errno variable.

```
DWORD dwThreadExitCode;

if (WAIT_OBJECT_0 == WaitForSingleObject(hThread, INFINITE)
{
  GetExitCodeThread(hThread, dwThreadExitCode);
  CloseHandle(hThread);
}
```

Please never use:

```
BOOL TerminateThread(
[in, out] HANDLE hThread,
[in]      DWORD  dwExitCode
);
```

to (forcibly) terminate a thread. This terminates the thread abruptly without performing any cleanup in the thread.

2.2.2 Linux

The simplest thread creation works like this:

```
pthread_t tid;
void *thread_context;

if (0 == pthread_create(&tid, NULL, ThreadRoutine, thread_context))
{
  /* OK */
  …
}
```

The ‘`ThreadRoutine`’ looks like this:

```
void *ThreadRoutine ( void *pContext )
{
  sigset_t blockedsig;

  sigfillset(&blockedsig);
  pthread_sigmask(SIG_BLOCK,&blockedsig,NULL);

  …

  return NULL;
}
```

The ‘`ThreadRoutine`’ blocks all signals in the prolog of the thread routine – as stated above. This is optional.

To wait for the thread to terminate (in another thread):

```
void *thread_exitcode;

if (0 == pthread_join(tid, &thread_exitcode))
{
  /* OK */
  …
}
```

Please read the Linux man page `pthread_cancel(3)` to learn more about cancellation, states, etc.

3 Mutexes

When it comes to concurrency in your application, there is a need for synchronization on shared resources. A shared resource may e.g. be a double-linked list that is filled (by one or more producers = threads) and fetched (by one or more consumers = threads) concurrently.

In the computer science, the basic construct is the so called 'semaphore' (see the web). A mutex is nothing else than a 'binary semaphore':

- A mutex can be created and destroyed;
- If a thread wants to access a shared resource, it '*acquires*' the mutex effectively blocking all other threads that also want to acquire it;
- Once the shared resource access is completed, it '*releases*' the mutex so that other threads may now also acquire the mutex.

The key concept here is: If a thread tries to acquire an already acquired mutex, its execution is suspended by the OS kernel until the other thread releases the mutex.

The acquisition and release of mutexes is also often called '*mutex lock*' or '*mutex unlock*', respectively.

3.1 Sidekick #1: mutexes versus semaphores

Mutexes and semaphores are based on an internal (synchronized) counter. Because a mutex is a binary semaphore, its counter is initialized to one (1) – internally. If someone acquires the mutex, the counter is decremented to zero (0). If a second thread tries to acquire a mutex with a zero counter, its execution is suspended until the mutex becomes available again.

A semaphore is also based on a counter, which can be initialized with a value greater than one (1). As an example, a host PC with a quad-port network card (the shared resource in this example) could initialize a NIC port semaphore with the value four (4). The first four threads requesting a dedicated network port all decrement the semaphore by one (1) until it reaches zero (0). The fifth thread that requires a dedicated NIC port is then suspended until one of the other four threads releases (increments) the semaphore.

3.2 Sidekick #2: deadlocks

The nightmare of a programmer is the so called 'deadlock'. It may come to a deadlock if you are using multiple mutexes in parallel, say mutex #1 and mutex #2. If one thread should acquire mutex #1 first and then mutex #2 and another thread acquires mutex #2 first and then mutex #1, then you end-up in a classical deadlock situation.

Both threads are locked (deadlocked) forever because none of them is able to acquire both mutexes.

You should avoid this kind of error situations by simply not acquiring multiple mutexes at a given point in time. Deadlock detection is a very hard debugging job. Quite often, you do not 'see' such a deadlock situation watching your source code because mutex locks may be obfuscated by e.g. library calls that acquire mutexes '*under the hood*'.

3.3 Usage of mutexes

3.3.1 MS Windows

The creation of a mutex is simple:

```
HANDLE hMutex = CreateMutex(NULL, FALSE, NULL);
```

The destruction of a mutex works as follows:

```
CloseHandle(hMutex);
```

The mutex acquisition is:

```
If (WAIT_OBJECT_0 == WaitForSingleObject(hMutex, INFINITE))
{
  /* Go… */
  …
}
```

To release an acquired mutex, call:

```
ReleaseMutex(hMutex);
```

You may replace ‘`INFINITE`’ in the `WaitForSingleObject` call by zero (0) to try to lock the mutex without blocking.

3.3.2 Linux

Again, the `libpthread.so` is used to provide POSIX mutexes; creation:

```
pthread_mutex_t mutex;

if (0 == pthread_mutex_init(&mutex, NULL))
{
  /* OK */
  …
}
```

Destruction:

```
pthread_mutex_destroy(&mutex);
```

You may also declare a mutex statically with an initializer that does not need to be destructed after usage:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

This is very useful if you do not have the chance to actively initialize a mutex in your source code.

Acquisition (lock) is performed by:

```
if (0 == pthread_mutex_lock(&mutex))
{
  /* OK */
  …
}
```

And finally, the release of a mutex:

```
pthread_mutex_unlock(&mutex);
```

There is also a `pthread_mutex_trylock` function if you need a non-blocking version of the mutex acquisition.

4 Events

Events are a central synchronization mechanism in all operating systems: A thread may wait (execution suspended) for some event to occur before continuing its execution.

One party (thread) signals an event while another party (thread) waits for the event to be signaled.

4.1 MS Windows

Windows comes with 'reliable' events. Creation:

```
HANDLE hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);
```

Destruction:

```
CloseHandle(hEvent);
```

To signal an event, one thread calls:

```
SetEvent(hEvent);
```

Another thread waiting for this event calls:

```
if (WAIT_OBJECT_0 == WaitForSingleObject(hEvent, INFINITE))
{
  /* Event signaled */
  …
}
```

That's it – in its simplest form.

4.2 Windows events versus POSIX condition variables

In former times (20+ years ago) I was looking for a synchronization mechanism on Linux comparable to the Windows events. The only possibility at that time was the usage of a POSIX '*condition variable*'.

A condition variable always requires a '*buddy mutex*', i.e. to use a condition variable you always need the combination of the POSIX condition variable and a POSIX mutex.

Creation is done like this:

```
pthread_cond_t condvar;
pthread_mutex_t condmutex;

if (0 == pthread_cond_init(&condvar,NULL))
{
  if (0 == pthread_mutex_init(&condmutex,NULL))
  {
    /* OK */
  }
}
```

The thread that sets (signals) the condition variable executes:

```
pthread_mutex_lock(&condmutex);
/* do something */
something_happened = true;
/* wake-up thread */
pthread_cond_signal(&condvar);
pthread_mutex_unlock(&condmutex);
```

Another thread waiting for the condition to be met executes:

```
pthread_mutex_lock(&condmutex);
while (!something_happened)
{
  pthread_cond_wait(&condvar, &condmutex);
}
something_happened = false;
/* do something */
pthread_mutex_unlock(&condmutex);
```

The crucial point here is that the function `pthread_cond_wait` requires the caller to first lock the condition variable mutex before calling it. `pthread_cond_wait` then unlocks (releases) the mutex waiting for the condition to be met (and suspending the execution of the thread until the condition is met).

4.2.1 The '*lost event*' problem (POSIX-specific)

As you can see from the sample source code above, we need another variable '`something_happened`' beside the POSIX mutex plus condition variable in order to make this thing work.

Compared to the Windows events, POSIX condition variables may suffer from a '*lost event*' or '*lost signal*', respectively. On Windows, it does not matter if the 'producer' first sets the event or if the 'consumer' is already in the waiting state or not, i.e. a Windows event is buffered if no one is currently waiting for it. The `WaitForSingleObject` call on an event immediately returns if such a buffered signal exists.

On UNIX systems (POSIX), the semantics of a POSIX condition variable are different: If no consumer is actually blocked in the `pthread_cond_wait` function and a producer sets (signals) the condition variable by calling `pthread_cond_signal`, then this signal is just lost. There is no buffering. This buffering is implemented via the additional variable '**`something_happened`**'.

4.3 Linux event file descriptors (eventfd)

The Linux event file descriptors, also known as '*eventfd*', have been introduced in the early 2.xer Linux kernels (see `eventfd(2)` man page). All properties explained/used here are available since 2.6.30.

You create an *eventfd* by calling:

```
#include <sys/eventfd.h>
int eventfd ( unsigned int initval, int flags );
```

On error, it returns -1 setting `errno`. On success, you just get a (standard) file descriptor, which you have to free by calling `close()` – unless you set the `EFD_CLOEXEC` flag.

Internally, an *eventfd* is just a 64bit counter (it is an `uint64_t`, see `stdint.h`). As you can see from the function declaration, the initial value is set in the `eventfd` system call. It is just an '`unsigned int`' (32 bit), which is extended to an `uint64_t` internally.

4.3.1 Note on the 64bit eventfd counter

The counter is stored in the order of the actual machine, i.e. either in Little Endian or Big Endian memory order. From the programmer's point of view, this is just an internal detail because you access the counter from your C/C++ program code just as an `uint64_t`. In the `eventfd(2)` man page, this is just called '*the host byte order*'.

4.3.2 *eventfd* usage

The *eventfd* is just a file descriptor, which means that you can use it e.g. in `select`, `poll` or `epoll` calls. This is its biggest advantage over POSIX condition variables: Think of a multi-threaded TCP/IP server waiting for network packets to arrive. In this scenario, you would have set up e.g. an `epoll` with all the client sockets (also file descriptors) waiting for packets to arrive.

You can just add an *eventfd* to this `epoll` reacting on any events that might occur in parallel to the packet receival wait. Such an event could be the '*thread, please close all client sockets and terminate*'.

4.3.2.1 *Initialization*

Upon creation of an *eventfd*, you specify the initial counter value (mostly just zero (0)).

4.3.2.2 *Read operation*

You use the `read` system call on an *eventfd* to read its value. The size of the `read` has to be `sizeof(uint64_t)` (eight bytes) or the calls fails with `errno=EINVAL`.

If the internal counter is zero, then the read system call either blocks or returns with `errno=EAGAIN` if the file descriptor was made non-blocking. This means that no event was signaled so far.

If the internal counter is non-zero, then the `read` system call returns the current internal 64bit counter to the caller. The non-zero counter means that one or more events have occurred already. See the next subsection.

How the internal counter is modified by a `read` system call depends on the flags that you specify upon the *eventfd* creation:

- **EITHER**: return the current counter and reset it to zero (0);
- **OR**: return the current counter and decrement the counter by one (1) – this is the `EFD_SEMAPHORE` semantics of the *eventfd*.

I personally like the semaphore semantics of the *eventfd* because it records and delivers all events (never forgets one). If you need an *eventfd* not relying on the number of event signals, then use the non-semaphore mode of the *eventfd*. The `read` operation just resets the *eventfd* no matter how many event signals had been performed in front of the `read` syscall.

The read operation is comparable to the `WaitForSingleObject` on MS Windows or the `pthread_cond_wait` (POSIX condition variables), respectively.

4.3.2.3 *Write operation*

All write operations on an *eventfd* have to be performed with an `uint64_t` value. The supplied value is just added to the internal *eventfd* 64bit counter, possibly waking up anyone currently blocked (waiting) for this event. The write operation is comparable to the MS Windows `SetEvent` or the `pthread_cond_signal` (POSIX condition variables), respectively.

Notes:

- The maximum value of the internal 64bit counter is `0xFFFFFFFFFFFFFFFE` or `(uint64_t)-2`, i.e. one less than the true 64bit maximum;
- If you try to write `(uint64_t)-1` (the maximum), then the write fails with `EINVAL`;
- If the counter would exceed the maximum value `(uint64_t)-2`, then this results in either a block (blocking file descriptor) or immediately returns with `errno=EAGAIN` if the event file descriptor is non-blocking.

4.3.3 Example (semaphore semantics)

The creation of the *eventfd* works like this:

```
int efd = eventfd(0, EFD_SEMAPHORE | EFD_NONBLOCK);
```

This automatically creates the *eventfd* in semaphore and non-blocking mode.

Destruction is:

```
close(efd);
```

The setting of the event is also straightforward:

```
uint64_t counter = 1;

if (sizeof(uint64_t) != write(efd,&counter,sizeof(counter)))
{
  /* error handling */
  …
}
```

Here comes a more elaborated version of a ‘`WaitForSingleObject`’ (MS Windows) for Linux using `poll`. The parameter `timeout` is specified in milliseconds or `(uint32_t)-1` for an infinite wait:

```
bool eventfd_waitevent ( int eh, uint32_t timeout )
{
  struct pollfd pfd;
  int           rc;
  uint64_t      evt_value;
  ssize_t       readbytes;

  if (-1 == eh)
    return false;

  pfd.fd      = eh;
  pfd.revents = 0;
  pfd.events  = POLLIN;

  do
  {
    rc = poll(&pfd, 1, ((int)timeout) < 0 ? -1 : ((int)timeout));
  }
  while (-1 == rc && EINTR == errno);
  // remark: timeout adjustment missing here

  if (rc <= 0) // 0 if timeout, -1 on error
    return false;

  if (pfd.revents & (POLLERR|POLLHUP))
    return false;

  if (0==(POLLIN & pfd.revents))
    return false;

  do
  {
    evt_value = 0;
    readbytes = read(eh, &evt_value, sizeof(uint64_t));
  }
  while (-1 == readbytes && EINTR == errno);

  // the value of evt_value is not evaluated here

  return (sizeof(uint64_t) == readbytes) ? true : false;
}
```

5 Conclusion

Concurrency and synchronization (intra-process) can be implemented on both operating systems MS Windows and Linux using the built-in mechanisms threads, mutexes, and events on MS Windows or POSIX threads, POSIX mutexes, and Linux *eventfd* on Linux.

The event file descriptors on Linux can be incorporated in existing `select`, `poll` or `epoll` loops. Please keep in mind that a POSIX condition variable is **not** an event mechanism as such.

UNIX signals (sent to a dedicated POSIX thread) may be used to cancel events (you may use `SIGUSR1` or `SIGUSR2` for this purpose). Please note that `SIGUSR2` is already used internally by the POSIX threads for the cancellation of threads.

If you want to implement some kind of the MS Windows `WaitForMultipleObjects` function (maybe also including the `bWaitAll` Boolean flag of this Win32 function), then you can do it using event file descriptors plus some small framework as well.

A Appendix

A.1 Synchronization using volatile variables

A.1.1 The C compiler keyword ‘volatile’

C compilers, no matter if you use the GNU Compiler Collection or the MS Visual C/C++ compiler, optimize your code aggressively – I assume that you compile your code having optimization switches enabled.

If you declare a variable, e.g.:

```
long sync_variable;
```

it might happen that the compiler does not use some space in memory for this variable but uses a CPU register for optimization purposes instead. This is very bad if two threads concurrently access this variable because each thread uses some kind of ‘*register shadow copy*’ in this case.

All modifications performed by one thread are invisible to the other thread (and vice versa).

You can force the C compiler to read a variable from memory every time it is accessed for read or to write a variable to memory every time it is accessed for write, respectively. This is accomplished by prefixing the variable declaration:

```
volatile long sync_variable;
```

It is essential for any intra-process synchronization based on synchronization variables that you always declare synchronization variables as ‘`volatile`’. Each thread has its own set of CPU registers, flags, and instruction pointer. Synchronization can only be performed via the (shared) memory for this reason.

A.1.2 Simple synchronization variables (INFORMATIONAL)

Is it sufficient to declare synchronization variables as ‘volatile’ using ‘standard’ C instructions to read or write from/to them, respectively? Answer: **NO!**

In this section, I introduce very simple synchronization variables for demonstration purposes. You should always use the event mechanism instead (please refer to chapter 4 beginning on page 10).

A CPU providing multiple execution units internally works with ‘bus coherence protocols’ (e.g. the MESI protocol). These protocols are in place just to get cache coherence among the CPU’s execution units.

What we need is the so called ‘*memory barrier mechanism*’, i.e. if one execution unit currently accesses a synchronization variable in memory, then no other execution unit may perform this concurrently. This is some kind of ‘*memory access mutual locking*’.

There are special CPU instructions (or instruction prefixes), which ‘tell’ all other execution units: ‘*hey, do not access this memory location now because I am currently working on it*’. For this purpose, MS Windows has the ‘*`InterLockedXXX`*’ Win32 API functions. On Linux (gcc), there are compiler intrinsics having the prefix ‘*`__sync_XXX_and_YYY`*’.

As an example, if you use the above declared synchronization variable ‘`sync_variable`’ as a Boolean flag, which is set by a producer and cleared by a consumer, then **do not** simply write:

```
(producer) sync_variable |= -1;
…
(consumer) while (0 == sync_variable) { /* wait */ }
sync_variable &= 0;
/* continue */
```

In this example, ‘producer’ and ‘consumer’ are two separate threads. To indicate/implement a memory barrier that yields a clean concurrent access to your synchronization variable, use this code on MS Windows instead:

```
(producer) InterlockedOr(&sync_variable, -1);
…
(consumer)
while (InterlockedCompareExchange(&sync_variable, 0, -1)) { /* wait */ }
```

The Win32 function ‘`InterlockedCompareExchange`’ compares the `sync_variable` with the third parameter, the **comparand**. If the variable currently stores this value, then the second parameter, the **exchange** value, is stored in `sync_variable` (thus clearing the Boolean variable). If the `sync_variable` is not already set to -1, then nothing is performed (a no-op).

This example is some kind of so called ‘spin lock’, i.e. the code waits for a condition to be met in a loop (it performs a spin lock on the variable). Please never implement code like this because it consumes CPU time massively (and is also power consuming). On MS Windows, there is no way to yield the execution to another thread while waiting, which **is** possible on Linux:

```
(producer) __sync_fetch_and_or(&sync_variable, -1);
…
(consumer)
while (0 == __sync_fetch_and_and(&sync_variable, 0)) {
  pthread_yield();
}
```

There are far more ‘interlocked’ function on MS Windows and Linux. Please read the online documentation about these if you need further information.