AURIGA
Elite Software R&D Services
Since 1990

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Thread synchronization in Linux and Windows systems

Author: Eduard Trunov, Software Engineer, Auriga, Inc.

Introduction

In modern operating systems, each process has its own address space and one thread of control. However, in practice we often face situations requiring several concurrent tasks within a single process and with access to the same process components: structures, open file descriptors, etc.

Organization of a multi-threading model under any circumstances requires simultaneous access to the same resources. This article provides general information about threads in Windows and Linux OSs, and then presents synchronization mechanisms[1] preventing access to shared resources.

This article will be of interest for those who deal with applications porting from one system to another or who create multi-threaded applications in one system and want to know how it is practically realized in the other system. This article will also be useful for those who have never programmed applications with multiple threads but plan to do so in the future.

Thread concept

What are these threads needed for? Why can't we just create processes? The latter paradigm has been working over the course of many years, but process creation has some disadvantages, with the following just a few examples:

- Process creation operation is resource-intensive.
- Processes require complicated mechanisms to access the same resources (named or unnamed pipes, message queues, sockets, etc.), while threads automatically gain access to the same address space.
- Performance of multi-threaded processes is higher than single-threaded.

Multithreading allows several threads to be executed as part of one process. A programming model with threads provides developers with a comfortable abstraction of simultaneous execution. One of the advantages of a programme with threads is that it works faster on computers with a multicore processor. Threads use almost no resources when created, or additional plugins such as resource

[1] For Linux, this article presents the threads interface defined by the POSIX.1-2001 standard (known as "pthreads").

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

access mechanisms; besides, performance and application interactivity of threads are higher. Apart from address space, all threads use:

- Process regulations,
- Signal handlers (settings for working with signals),
- Current directory,
- User and group identifier.

At the same time, each thread has its own:

- Thread identifier,
- Stack,
- Set of registers,
- Signal mask,
- Priority.

Main functions for working with threads

At program startup by exec call, a main thread (initial thread) is created. Secondary threads are created by calling pthread_create for Linux or `_beginthread(ex)` for Windows.

Let's look more closely at threads creation for Linux:

```
#include <pthread.h>
int pthread_create(
       pthread_t *tid,
       const pthread_attr_t *attr,
       void *(*func)(void *),
       void *arg
);
/* Returns 0 in case of a successful completion, positive value in case of an error*/
```

Each thread has its identifier – *pthread_t* – and attributes: priority, initial stack size, daemon feature. When creating a thread, it is necessary to indicate the function address that will be executed (*func*), and also the single pointer argument (*arg*). Threads in Linux shall be exited either explicitly – by calling *pthread_exit* function – or inexplicitly – by returning from this function[2]. If under the conditions of the problem it is required to pass several arguments to a thread, it is necessary to use the address of the structure with arguments.

In Windows, threads are created with the help of *_beginthread(ex)* or *CreateThread* functions. Both are C-runtime calls, and the main difference between them is that *CreateThread* is a "raw"

[2] Threads exit will be covered later in this article.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Win32 API, and *_beginthread(ex)* in its turn calls *CreateThread* inside of itself. In this article, we will discuss *_beginthread(ex)* functions. The syntax of *_beginthreadex* is as follows:

```
uintptr_t _beginthreadex(
       void *security,
       unsigned stack_size,
       unsigned(__stdcall *start_address)(void *),
       void *arglist,
       unsigned initflag,
       unsigned *thrdaddr
);
```

It can be observed that between *pthread_create* and *_beginthreadex* calls there is some vague similarity; however, there are also differences. Thus, in Windows: *security* – pointer to the structure *SECURITY_ATTRIBUTES*, *thrdaddr* – points to 32-bit variable that receives the thread identifier.

Let's consider the following example of threads creation[3]:

```
#include <stdio.h>

#ifdef __PL_WINDOWS__
       #include <windows.h>
#endif //__PL_WINDOWS__
#ifdef __PL_LINUX__
       #include <pthread.h>
#endif //__PL_LINUX__

#define STACK_SIZE_IN_BYTES (2097152) //2MB

#ifdef __PL_WINDOWS__
unsigned int __stdcall process_command_thread(void) {
#endif //__PL_WINDOWS__
#if defined (__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
void *process_command_thread(void *p) {
#endif //(__PL_LINUX__) || (__PL_SOLARIS__) || (__PL_MACOSX__)
       printf("Hello from process command thread\n");
```

[3] In this example, as in other examples in this article, code base will be single for both Linux and Windows. The difference will be in the compilation condition:

```
#ifdef __PL_WINDOWS__
       //Windows code
#endif //__PL_WINDOWS__
#ifdef __PL_LINUX__
       //Code for UNIX OS systems
#endif //__PL_LINUX__
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
	return 0;
}

int main(int argc, char *argv[])
{
#ifdef __PL_WINDOWS__
	DWORD process_command_thread_id;
	HANDLE h_process_command_thread;
	h_process_command_thread = (HANDLE)_beginthreadex(
		NULL,
		STACK_SIZE_IN_BYTES,
		process_command_thread,
		NULL,
		0,
		(unsigned long *)&process_command_thread_id
	);
	if (h_process_command_thread == NULL)
		return -1;
#endif //__PL_WINDOWS__
#ifdef __PL_LINUX__
	pthread_t h_process_command_thread;
	int h_process_command_thread_initialized;
	int ret;
	ret = pthread_create(
		&h_process_command_thread,
		NULL,
		process_command_thread,
		NULL
	);
	if (ret != 0)
		return -1;
	h_process_command_thread_initialized = 1;
#endif // __PL_LINUX__
	printf("Hello from main thread\n");
	return 0;
}
```

The output will be the following:

LINUX	WINDOWS
[root@localhost ~]# ./process Hello from main thread [root@localhost ~]#	C:\>process.exe Hello from main thread C:\>

It is easy to notice that process_command_thread was not run visually. When internal structures used for threads management are initialized by *pthread_create* or *_beginthreadex* function, the main thread finishes executing. We can expect a thread exit after calling pthread_join for Linux.

```
int pthread_join(pthread_t tid, void **retval);
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

A thread may be either joinable (by default) or detached. When a joinable thread is terminated, information (identifier, termination status, thread counter, etc.) is kept until *pthread_join* is called.

In the Windows OS, one of the wait-functions may be considered analogous to *pthread_join*. The wait functions family allows a thread to interrupt its execution and wait for a resource to be released. Let's take a look at an analogue of *pthread_join*, which is *WaitForSingleObject*:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwMilliseconds);
```

When this function is called, the first parameter, *hObject*, identifies the kernel object. This object may be in one of two states: «free» or «busy».

The second parameter, *dwMilliseconds*, indicates how many milliseconds a thread is ready to wait for the object to be released.

The following example illustrates *pthread_join\WaitForSingleObject* call:

```
#ifdef __PL_WINDOWS__
	DWORD status = WaitForSingleObject(
		h_process_command_thread,
		INFINITE
	);
	switch (status) {
		case WAIT_OBJECT_0:
			// The process terminated
			break;
		case WAIT_TIMEOUT:
			// The process did not terminate within timeout
			break;
		case WAIT_FAILED:
			// Bad call to function
			break;
	}
#endif //__PL_WINDOWS__
#ifdef __PL_LINUX__
	int status = pthread_join(
		h_process_command_thread,
		NULL
	);
	switch (status) {
		case 0:
			// The process terminated
			break;
		case default:
			// Bad call to function
			break;
	}
#endif //__PL_LINUX__
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

In Windows code base, there is a constant *INFINITE*, which is passed by the second parameter. This constant indicates that a thread waits for an event infinitely. The constant is declared in *WinBase.h* and is defined as *0xFFFFFFFF (or -1)*.

In addition, Windows code includes *WAIT_TIMEOUT*. This condition is not represented in Linux. In practice, this restriction is bypassed with the help of the following functions:

```
int pthread_tryjoin_np(pthread_t thread, void **retval) [4];

int pthread_timedjoin_np(
        pthread_t thread,
        void **retval,
        const struct timespec *abstime
);
```

If you refer to the *pthread_tryjoin_np* help page, you can see that *EBUSY* may be an error, and *WaitForSingleObject* cannot inform us about it. To know state of the thread and identify its exit code, it is necessary to call the function:

```
BOOL GetExitCodeThread(HANDLE hThread, PDWORD pdwExitCode);
```

The exit code is returned as a variable that *pdwExitCode* points to. If the thread has not terminated when the function is called, then *STILL_ACTIVE* identifier is filled as the variable. If the call is successful, then the function returns *TRUE*.

Let's consider a case of *pthread_tryjoin_np* function usage for Linux and *GetExitCodeThread\WaitForSingleObject* function for Windows.

```
#ifdef __PL_WINDOWS__
	DWORD dwret;
	BOOL bret;
	DWORD h_process_command_thread_exit_code;
	if (h_process_command_thread != NULL) {
		bret = GetExitCodeThread(
			h_process_command_thread,
			&h_process_command_thread_exit_code
		);
		if (h_process_command_thread_exit_code == STILL_ACTIVE) {
			dwret = WaitForSingleObject(
				h_process_command_thread,
				5000   // 5000ms
			);
			switch (dwret) {
				case WAIT_OBJECT_0:
```

[4] Based on reference description, this function is a nonstandard GNU-extension, which requires a "_np" (nonportable) suffix.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
                                        // everything from this point on is good
                        break;
                                case WAIT_TIMEOUT:
                                case WAIT_FAILED:
                                default:
                                        SetLastError(dwret);
                                        break;
                        }
                }
        }
#endif //__PL_WINDOWS__
#ifdef __PL_LINUX__
        int iret;
        struct timespec wait_time = { 0 };
        if (h_process_command_thread_initialized == 1) {
                iret = pthread_tryjoin_np(
                        h_process_command_thread,
                        NULL
                );
                if ((iret != 0) && (iret != EBUSY)) {
                        //TODO: process the error
                }
                if (iret == EBUSY) {
                        clock_gettime(CLOCK_REALTIME, &wait_time);
                        ADD_MS_TO_TIMESPEC(wait_time, 5000);
                        iret = pthread_timedjoin_np(
                                h_process_command_thread,
                                NULL,
                                &wait_time
                        );
                        switch (iret) {
                                case 0:
                                        // everything from this point on is good
                                        break;
                                case ETIMEDOUT:
                                case EINVAL:
                                default:
                                        break;
                        }
                }
        }
#endif //__PL_LINUX__
```

An attentive reader will notice that *ADD_MS_TO_TIMESPEC* is a macro that is not represented in Linux OS. Macros are added to *wait_time* 5000 *ms*, but macros implementation falls outside the scope of this article. It also should be mentioned that in Linux we need to introduce a separate variable, *h_process_command_thread_initialized*, because *pthread_t* is *unsigned long* (in general), and we cannot verify it.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Let's sum up the results. Linux and Windows OSs provide an opportunity to create threads inside of an application. In Windows OS the type is *HANDLE*, and in Linux – *pthread_t*. In case of creating a joinable thread in Linux OS it is necessary to write *pthread_join*, even if we are sure that the thread has terminated. This practice will help us to avoid system resources leakage.

Discussed functions are recorded in Table 1.

Linux functions	Windows functions
pthread_create	beginthreadex
pthread_join	WaitForSingleObject(.., INFINITE)
pthread_timedjoin_np	GetExitCodeThread\WaitForSingleObject
pthread_tryjoin_np	GetExitCodeThread

Table 1. Functions for thread synchronization in Windows and Linux OSs.

Events

Events are instances of kernel objects variation. Events inform about an operation termination and are normally used when a thread performs initialization and then signals to another thread that it can continue working. The initializing thread transforms the «event» object into unsignaled state, after which it proceeds with its operations. When completed, it releases the event to signaled state. In its turn, the other thread that has been waiting for the event to change its state to signaled, resumes, and again becomes scheduled.

Let's take a look at functions for working with «event» objects in Windows and Linux OSs.

In Windows OS an «event» object is created with the *CreateEvent* function:

```
HANDLE CreateEvent(
       PSECURITY_ATTRIBUTES psa,
       BOOL fManualReset,
       BOOL fInitialState,
       PCSTR pszName
);
```

Let's bring to a sharper focus *fManualReset* and *fInitialState* parameters. *FManualReset* parameter of *BOOL* type informs the system about a need to create a manual-reset event (*TRUE*) or an auto-reset event (*FALSE*). The *fInitialState* parameter determines the initial state of the event: signaled (*TRUE*) or unsignaled (*FALSE*).

After the event is created, there is a possibility to manage the state. To transit an event to a signaled state, you need to call:

```
BOOL SetEvent(HANDLE hEvent);
```

To change the event state to unsignaled, you need to call:

```
BOOL ResetEvent(HANDLE hEvent);
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

To wait for an event signal, you need to use the already-familiar-to-us *WaitForSingleObject* function.

In Linux OS, an «event» object represents an integer descriptor. An integer «event» object is created with the *eventfd* function:

```
int eventfd(unsigned int initval, int flags);
```

The *initval* parameter is a kernel serviced counter. The *flags* parameter is required for *eventfd* behavior modification, which may be *EFD_CLOEXEC*, *EFD_NONBLOCK*, or *EFD_SEMAPHORE*. If terminated successfully, *eventfd* returns a new file descriptor, which can be used to link the *eventfd* object.

Analogous to *SetEvent*, we can use a *eventfd_write* call:

```
ssize_t eventfd_write(int fd, const void *buf, size_t count);
```

When write is called from the buffer, an 8-byte integer value is added to the counter. The maximum counter value can be a 64-bit unsigned minus 1. In case of a successful function call, the written number of bytes is returned.

Before we discuss a *ResetEvent* analogue, let's take a look at the *poll* function.

```
#include <poll.h>
int poll(struct pollfd fdarray[], nfds_t nfds, int timeout);
```

The *poll* function allows an application to simultaneously block several descriptors and receive notifications as soon as any of them is ready for reading or writing. Work of *poll* (generally) can be described as follows:

1. Notify when any of the descriptors is ready for an input–output operation.
2. If none of the descriptors is ready, go to sleep mode until one or more descriptors are ready.
3. In case there are available descriptors ready for input–output, handle them without blocking.
4. Go back to step 1.

Linux OS offers three entities for multiplexed input–output: interface for selection (*select*), polling (*poll*), extended polling (*epoll*).

Those who have experience working with *select* might appreciate the advantage of *poll*, which uses a more effective method with three groups of descriptors based on bit masks. *Poll* call works with a single array of *nfds pollfd* structures that file descriptors point to.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Let's take a look at the *pollfd* structure definition:

```
struct pollfd {
       int fd; /* file descriptor */
       short events; /* requested events */
       short revents; /* returned events */
};
```

There is indicated a file descriptor in each *pollfd* structure that will be tracked. Several file descriptors may be passed to the *poll* function (*pollfd* array of structures). The number of elements in an *fdarray* array is determined by the *nfds* argument.

To communicate what events we are interested in to the kernel, it is necessary to write one or more values from Table 2 in the *events* field for each element from the array. After returning from *poll* functions, the kernel specifies what events occurred for each of the descriptors.

Name	Events	Revents	Description
POLLIN	+	+	Data is available for reading (except for high priority)
POLLRDNORM	+	+	Regular data (priority 0) is available for reading
POLLRDBAND	+	+	Data with non-zero priority is available for reading
POLLPRI	+	+	High priority data is available for reading
POLLOUT	+	+	Data is available for writing
POLLWRNORM	+	+	Analogous to POLLOUT
POLLWRBAND	+	+	Data with non-zero priority is available for writing
POLLERR		+	Error occurred
POLLHUP		+	Connection lost
POLLNVAL		+	There is a mismatch between the descriptor and the open file

Table 2. Possible values of *events* and *revents* flags of *poll* function.

Argument *timeout* defines the wait time for occurrence of specified events. There are three possible values for timeout.

- *timeout* = -1: The wait time is infinite (*INFINITE* in *WaitForSingleObject*).
- *timeout* = 0: The wait time is equal to 0, which indicates that it is necessary to inspect all specified descriptors and give the control back to the calling program.
- *timeout* > 0: Wait for no longer than *timeout* milliseconds.

Having reviewed the *poll* function, we can draw a conclusion that there is an analogy to *WaitForSingleObject* for "event" objects in Windows OS.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Let's move to the *ResetEvent* analogue for Linux.

```
#ifdef __PL_LINUX__
	struct pollfd wait_object;
	uint64_t event_value;
	int ret;
	if (eventfd_descriptor > 0) { // Descriptor created by eventfd(0,0)
		wait_object.fd = eventfd_descriptor;
		wait_object.events = POLLIN;
		wait_object.revents = 0;
		ret = poll(&wait_object, 1, 0); // Do not wait
		if (ret < 0) { // Error
		} else {
			if ((wait_object.revents & POLLIN) != 0) {
				iret = eventfd_read(eventfd_descriptor, &event_value);
				if (iret != 0) { // Error }
			}
		}
	}
#endif //__PL_LINUX__
```

Initially we check that *eventfd_descriptor* is greater than zero[5] (really, this was originally created by *eventfd* function without errors). After that, we initialize the *pollfd* function and run *poll*. Execution of *poll* is required to check if there is available data for reading. If such data is available, we will read it.

Through the lens of all of the above-described, let's reflect the outcome in Table 3:

Windows functions	Linux functions
CreateEvent	eventfd
SetEvent	eventfd_write
ResetEvent	poll/eventfd_read
WaitForSingleObject	poll

Table 3. Main functions for working with events in Windows and their analogues in Linux.

Multithreaded applications parallelism

Typically, there are two interrelated but different phenomena in multi-threaded applications: concurrency and parallelism.

Concurrency is the ability of two and more threads to overlap in execution.

Parallelism is the ability to execute two and more threads at the same time.

[5] As it is a file descriptor, it will be the least unusable positive numeric value. It is supposed that an application does not close STDIN_FILENO, elsewise it requires a different verification.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

It is concurrency that causes the majority of complications in streaming – threads can be executed in unpredictable order relative to each other. In case of sharing resources by the threads, this will undoubtedly lead to race conditions.

The term *race condition* commonly refers to a situation when unsynchronized access to a shared resource for two and more threads leads to erroneous program behavior.

Let's take a look at an example of a race.

Nowadays it is hard to imagine our life without plastic cards. ATM cash withdrawal became a daily routine long ago: insert a card, enter PIN-code and desired sum. If completed successfully we receive the planned amount of cash. The bank in its turn needs to verify if the money is available by the following algorithm:

1. *Is there available a minimum of X units of money on the bank account?*
2. *If yes, reduce the balance of the account by X value, dispense X money units to the user.*
3. *Elsewise, generate an error message.*

An example of code with race conditions:

```
int cash_out(struct account *ac, int amount) {
	const int balance = ac->balance;
	if (balance < amount)
		return -1;
	ac->balance = balance - amount;
	discard_money_routine(amount);
	return 0;
}
```

Race can emerge in a situation when a purchase is paid for online and cash is withdrawn from an ATM "*concurrently*".

To avoid a race, it is necessary to introduce the following update to the code:

```
int cash_out(struct account *ac, int amount) {
	lock();
	const int balance = ac->balance;
	if (balance < amount)
		return -1;
	ac->balance = balance - amount;
	unlock();
	discard_money_routine(amount);
	return 0;
}
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

In Windows OS, the code area requiring exclusive access to some shared data is called a «critical section».

The structure type for working with critical sections is *CRITICAL_SECTION*. Let's review its fields:

```
typedef struct _RTL_CRITICAL_SECTION {
    PRTL_CRITICAL_SECTION_DEBUG DebugInfo;

    //
    //  The following three fields control entering and exiting the critical
    //  section for the resource
    //

    LONG LockCount;
    LONG RecursionCount;
    HANDLE OwningThread;        // from the thread's ClientId->UniqueThread
    HANDLE LockSemaphore;
    ULONG_PTR SpinCount;        // force size on 64-bit systems when packed
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Though CRITICAL_SECTION formally does not belong to undocumented structures, Microsoft, nevertheless, believe that there is no need for users to know its organization. In practice, it is a sort of black box. To work with this structure, there is no need to use its field directly, but only through Windows functions, passing to them the address of a corresponding instance of this structure.

The *CRITICAL_SECTION* structure is initialized by the following call:

```
void InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

If we know that *CRITICAL_SECTION* structure will no longer be needed, then we can delete it with the help of the following call:

```
void DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

The code area working with a shared resource shall be preceded by the following call:

```
void EnterCriticalSection(PCRITICAL_SECTION pcs);
```

Instead of *EnterCriticalSection* we can use:

```
bool TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

TryEnterCriticalSection allows a thread to check the resource accessibility and engage in another activity in cases when it is not accessible. In the case of success (the function returned *TRUE*), it is clear that structure elements are updated and the resource is locked.

At the end of the code area using a shared resource, there shall always be the following call:

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
void LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

LeaveCriticalSection inspects *CRITICAL_SECTION* structure elements and decreases resource locking counter (*LockCount*) by 1.

Analogous to *CRITICAL_SECTION* in Linux OS is the variable mutex *pthread_mutex_t*. Before using, this variable needs to be initialized – write the value of the constant *PTHREAD_MUTEX_INITIALIZER* or call to pthread_mutex_init function.

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t *restrict mutex,
      const pthread_mutexattr_t *restrict attr);
```

To initialize a mutex with default attribute values, it is necessary to pass *NULL* to *attr* attribute. Specific mutex attribute values can be located on the help page.

A mutex can be deleted with the help of the following call:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

A mutex is locked by calling to the *pthread_mutex_lock* function:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

If a mutex is already locked, the calling thread will be blocked until the mutex is released. The mutex is unlocked with the help of the *pthread_mutex_unlock* function:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

If we want to check a resource's accessibility, then we can use the *pthread_mutex_trylock* function:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

The above-mentioned function will return *EBUSY* if a mutex is locked.

All functions for working with a mutex return 0 in the case of success and an error code in the case of failure.

Let's sum it up. In Windows OS, to work with a shared resource it is necessary to use a critical section and a special type *CRITICAL_SECTION*. In Linux OS, we can use mutexes of pthread_mutex_t type for the same purpose.

Synchronization functions are recorded in Table 4.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

Windows functions	Linux functions
`InitializeCriticalSection`	`pthread_mutex_init()`
`EnterCriticalSection`	`pthread_mutex_lock()`
`LeaveCriticalSection`	`pthread_mutex_unlock()`
`TryEnterCriticalSection`	`pthread_mutex_trylock()`
`DeleteCriticalSection`	`pthread_mutex_destroy()`

Table 4. Synchronization functions for shared resources.

Thread termination

One of the cases when it is necessary to write a thread termination in practice is mass data processing. A situation is possible when the main thread signals to all threads to exit but one of them is still processing information. If promptitude is a higher priority factor for application performance in comparison to information loss, then the thread needs to be exited and release system resources. This section will cover ways to exit a thread.

A thread can exit in the following ways:

- The thread function returns;
- The thread calls the ExitThread function;
- Any thread of the process calls the TerminateThread function;
- Any thread of the process calls the ExitProcess function.

Let's take a closer look at each of these.

The thread function returns.

A good example of clean code is considered designing a thread function so that the thread is terminated only after the function returns. In Windows OS, this way of thread termination guarantees correct clean-up of resources owned by the thread. In Linux OS, it is necessary to call to one of the join functions in cases where a thread is joinable. In the general case, the following happens:

- The system correctly releases resources taken by the thread.
- The system sets a thread exit code.
- Users counter for this kernel object «thread» is reduced by 1.

In Windows OS, a thread can be forcefully terminated by calling to:

```
void ExitThread(DWORD dwExitCode);[6]
```

[6] ExitThread is a function that kills a thread. In practice, it is best to use the _endthreadex function.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

The thread exit code value will be added to the *dwExitCode* parameter. It is easy to notice that the function does not have return value, because after the function is called the thread ceases to exist.

In Linux OS, there is a complete analogue for *ExitThread*:

```
void pthread_exit(void *rval_ptr);
```

Argument *rval_ptr* represents an untyped pointer, containing a return value. This pointer can be obtained by other process threads calling to the *pthread_join* function.

The function call *pthread_join* takes the thread to a detached state. This state allows the thread resources to be won back. In case if the thread has already been in detached state, the thread calling to *pthread_join* will get the *ESRCH* error code. Sometimes when *pthread_join* is recalled with the second non- *NULL* parameter, Segmentation Fault error output is possible.

Any thread of the process calls the TerminateThread function.

One thread can pass a request to forcefully terminate another thread within the same process. In Windows OS, this is organized with the help of the following function:

```
bool TerminateThread(
             HANDLE hThread,
             DWORD dwExitCode
);
```

The above-mentioned function terminated the *hThread* thread from any other thread. You can add to *dwExitCode* parameter a value that the system will consider as the thread exit code. After the thread is killed, users counter for this kernel object «thread» will be reduced by 1.

In Linux OS, a similar capability is implemented when one thread can pass a request for forceful termination of another thread within the same process by calling to the *pthread_cancel* function:

```
int pthread_cancel(pthread_t tid);
```

This function needs to be used in conjunction with *pthread_setcancelstate* and *pthread_setcanceltype* functions. In case of using *pthread_cancel*, *rval_ptr* will be *PTHREAD_CANCELED*.

Let's take a closer look at *erminateThread* and analogous actions in Linux OS:

```
#ifdef __PL_WINDOWS__
       BOOL bret = FALSE;
       bret = TerminateThread(h, x);
#endif //__PL_WINDOWS__
#ifdef __PL_LINUX__
       int iret = 0, bret;
       iret = syscall(SYS_tkill,tid, 0);
```

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

```
	if (iret == 0) {
		iret = pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,NULL);
		if (iret != 0) {
			bret = FALSE;
		}
		else {
			iret = pthread_cancel(h);
			if (iret == 0 || iret == ESRCH) {
				bret = TRUE;
			} else {
wait_thread:
				clock_gettime(CLOCK_REALTIME, &wait_time);
				ADD_MS_TO_TIMESPEC(wait_time, 1000); //1000 ms
				iret = pthread_timedjoin_np(h, NULL, &wait_time);
				switch (iret) {
					case 0:
						bret = TRUE;
						break;
					case ETIMEDOUT:
						if (retries_count++ < 5) // 5 Attempts

						{
							goto wait_thread;
						}
						bret = FALSE;
						break;
					default:
						bret = FALSE;
						break;
				}
			}
			(void)pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED,NULL);
		}
	}
	else {
		bret = TRUE;
	}
#endif //__PL_LINUX__
```

In Windows OS, *TerminateThread* returns *TRUE* in case of successful termination. Starting out from this logic, let's try to organize analogous functionality in Linux.

To start with, we will check if the thread is alive. To do so, we will send a zero signal. If a thread exists in the process and is being executed, then *tkill* will return 0. After that, we will call to the pthread_setcanceltype function with the *PTHREAD_CANCEL_ASYNCHRONOUS* parameter, and only after this we will try to terminate the thread by calling to the pthread_cancel function. There is a possible situation when the thread will still be executing for some time, and for this we will use the timed wait function *pthread_timedjoin_np* (there are 5 attempts in the example; each takes 1000 ms). After the thread is terminated, we will call to *pthread_setcanceltype* again with the

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

PTHREAD_CANCEL_DEFERRED parameter. In case there are any errors, there will always be returned FALSE.

Any thread of the process calls the ExitProcess function.

If one of the threads calls to *exit* function (for Linux OS), or either *ExitProcess* or *TerminateProcess* (for Windows OS), then the whole process will be terminated. Likewise, if a thread receives a signal which action consists in process termination, this signal will terminate the whole process.

Let's summarize it all. A correctly designed application shall not call to forceful thread termination functions. A recommended way is when the thread function returns. In case of a thread termination in Linux OS, there shall be called a join to release links and system resources owned by the process. Proper allowance must be made for the fact that calling to join functions twice is incorrect (at its best there will be an error). If a thread is forcefully terminated for Windows OS, it is important to note that all nonsystem resources shall be cleaned up in any way. In Linux, there are *pthread_cleanup_push* and *pthread_cleanup_pop* functions for that.

Thread termination functions are recorded in Table 5.

Windows functions	Linux functions
`ExitThread`	`pthread_exit`
`TerminateThread`	`pthread_setcanceltype, pthread_cancel, pthread_setcanceltype`

Table 5. Thread termination functions.

Conclusion

To mitigate influence of interrupts, an OS provides a model that consists of in-parallel running processes. However, this model has its weak points related to interaction, availability of own address space, etc.

On the other hand, it might be advantageous for applications to have several control threads within one process. As these threads belong to the same process, they use the shared address space but their own stack.

Threads may interact with each other using primitives such as semaphores, messages, etc. These primitives are necessary to make it impossible for simultaneous execution of two and more threads in critical sections. A thread can be in one of the following states: runnable or executing blocked. A thread can also interact with another thread using interaction primitives. Keep in mind that while using interaction primitives one should exercise caution to avoid errors and deadlocks.

Headquarters: +1 (866) 645-1119
EU Office: +370 (5) 203-1571
Russian Office: +7 (495) 713-9900
www.auriga.com pr@auriga.com

In this article, we analyzed streaming API for Windows and Linux OS; presented event-based thread creation and synchronization interfaces; discussed functions for working with shared resources; and detailed thread termination ways.

List of references

1. Stevens, W. Richard. Advanced Programming in the UNIX Environment, 3rd Edition. / W. Richard Steven, Stephen A. Rago. - Spb.: Piter, 2018. –944 p.
2. Russinovich, M. Windows Internals. 7th Edition. / M. Russinovich, D. Solomon, A. Ionescu, P. Yosifovich. - Spb.: Piter, 2018. - 944 p.
3. Tanenbaum A. Modern Operating Systems. 4th Edition./ A. Tanenbaum , H. Bos. - Spb.: Piter, 2015. –1120 p.
4. Love, R. Linux System Programming. 2nd Edition./ R. Love. - Spb.: Piter, 2014. –448 p.
5. Richter J. Programming Applications for Microsoft Windows: Master the critical building blocks of 32-bit and 64-bit Windows-based applications. 4th Edition. / J. Richter. - M.: Publishing company «Russian editorship»; 2008. –720 p.
6. Stevens, W. Richard. Unix Network Programming: Interprocess Communications. / W. Richard Stevens. - Spb.: Piter, 2003. –576 p.
7. Linux man pages online [Electronic resource]. – URL: http://man7.org/linux/man-pages/
8. Microsoft Docs [Electronic resource]. – URL: https://docs.microsoft.com/