# POSIX vs. Win32 Threads

In depth threading techniques

Robert.Sayegh@gmail.com (2008)

# Disclaimer

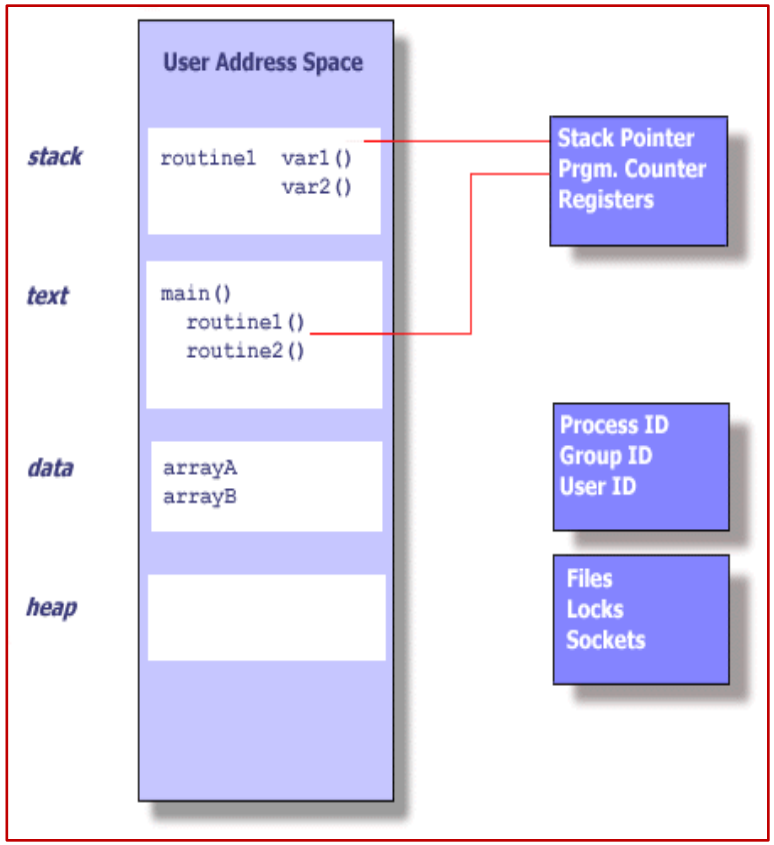
- This presentation provides a summary based on few articles regarding threads in Win32 and POSIX systems. All these articles are mentioned in the ***References*** slide.

# What is Thread

## 1. Processes

- What is a Process
  - A process is created **by OS** to run your **Main** program.

- Contains all information to be handled by the OS.
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - ***Scheduling Properties (priority, etc.)***
  - Working directory.
  - Program instructions
  - ***Registers,*** ***Stack,*** Heap
  - File descriptors
  - ***Signal actions***
  - Shared libraries
  - Inter-process communication tools
    shared memory , message queues,
    semaphores , pipes,

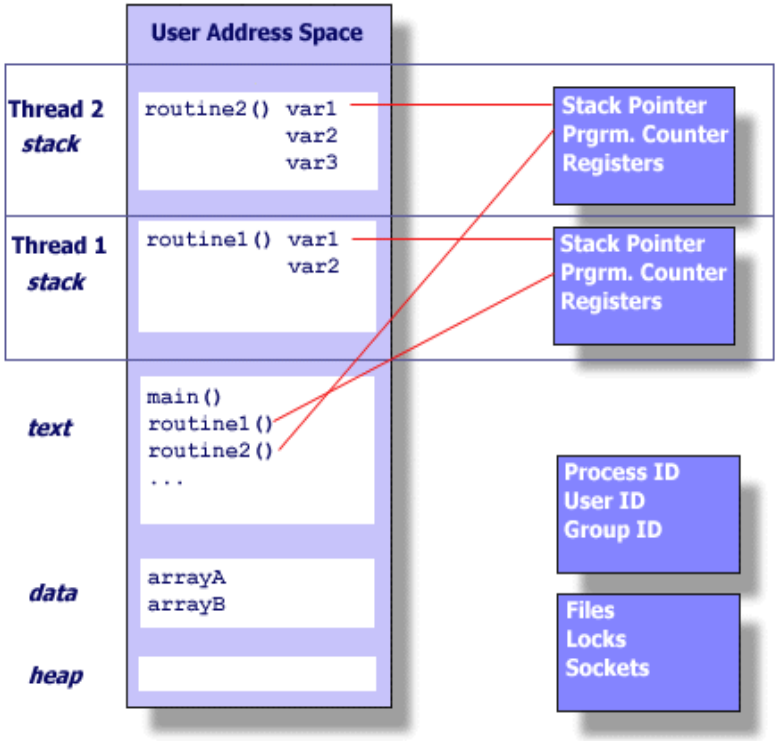


# What is Thread

## 2. Threads

- What is a Thread
  - A thread is created **by OS** to run a stream of instructions.
- Threads contain smaller set of information
  - *Scheduling Properties*
  - *Registers*
  - *Stack*
  - *Signal actions*
- Light-Weight

  *Most overhead accomplished through maintaining its process.*
  - Exists within a process and uses the process resources
  - Has its own independent flow of control (as long as its parent process exists and the OS supports it)
  - Duplicates only essential resources for independent scheduling




Comparing timing results in POSIX for the **fork()** and **pthreads_create()** (reflecting 50,000 creations) shows **30-60 times overhead**!

# Threads Sharing

- **Resource sharing.**
  Changes made by one thread to shared system resources will be seen by all other threads.
  - File open or close at the process level.

- **Memory Sharing (Same address, same data).**
  Two pointers having the same value, exist in the same virtual memory and point to the same data.
  - No copy is needed!

- **Synchronization required.**
  As resources and memory sharing is possible, explicit synchronization is required by the programmer

# Win32 vs. POSIX Interface
## 1. Function calls

| Win32 Threads | PThreads |
|---|---|
| Just one type: HANDLE | Each object has its own data type : **pthread_t**, pthread_mutex_t, ... |
| One function needed to make one functionality (e.g. **WaitForSingleObject)** | Each object has its own functions/attributes |
| Simpler and more generic - is it OOP? (sounds like misusing **void*** in C). | Reading and understanding may be more straightforward and less confusing. |

# Win32 vs. POSIX Interface
## 2. Synchronization overhead

| Win32 Threads | PThreads |
|---|---|
| **Synchronization objects:**<br>– Events<br>– Semaphores<br>– Mutexes<br>– Critical sections | **Synchronization primitives:**<br>– Semaphores<br>– Mutexes<br>– Conditional variables |
| *once an event is in the signaled state, it stays signaled.*<br>However, It is up to the programmer to ensure the proper switching of Windows events from the signaled to unsignaled state.<br>(When an object is signaled, you have to check what other objects in the awaken thread might be waiting for and remove it from those wait queues) | *Signals to condition variables are either "caught" by waiting thread(s) or discarded.*<br>However, use of a well known coding structure at each access of a condition variable will ensure no signals are "lost" by threads that may not be waiting at the exact time of signaling |

# More issues to consider

- Historically, hardware vendors have implemented their own proprietary versions of threads.
- In Windows, the thread is the basic execution unit, and the process is a container that holds this thread.
  In Linux, the basic execution unit is the process
- Some may claim POSIX threads are a low-level API and Windows threads are a high-level API
- In Windows the thread scheduling code is implemented in the kernel.
- In Linux realm, there are several drafts of the POSIX threads standard. Differences between drafts exist! Especially many scheduling mechanisms exist (user and kernel-level threads).
- The current POSIX standard is defined only for the C language.

# Mapping WIN32 to PTHREADS
## 1. Process Mapping

| Win32 | Linux | Classification |
|---|---|---|
| CreateProcess()<br>CreateProcessAsUser() | fork()<br>setuid()<br>exec() | Mappable |
| TerminateProcess() | kill() | Mappable |
| SetThreadpriority()<br>GetThreadPriority() | Setpriority()<br>getPriority() | Mappable |
| GetCurrentProcessID() | getpid() | Mappable |
| Exitprocess() | exit() | Mappable |
| Waitforsingleobject()<br>Waitformultipleobject()<br>GetExitCodeProcess() | waitpid()<br>!! Using semaphores Waitforsingleobject / Waitformultipleobject can be implemented | Context specific |
| GetEnvironmentVariable<br>SetEnvironmentVariable | getenv()<br>setenv() | Mappable |

**Mappable:** Both the Windows and Linux constructs provide similar functionality.
**Context:** The decision to use a specific Linux construct(s) depends on the application context

**!! More on WaitForMultipleObjects may be found under the following reference:** http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html

# Mapping WIN32 to PTHREADS
## 2. Thread Mapping

| Win32 | Linux | Classification |
|---|---|---|
| CreateThread | pthread_create<br>pthread_attr_init<br>pthread_attr_setstacksize<br>pthread_attr_destroy | Mappable |
| ThreadExit | pthread_exit | Mappable |
| WaitForSingleObject | pthread_join<br>pthread_attr_setdetachstate<br>pthread_detach | Mappable |
| SetPriorityClass<br>SetThreadPriority | setpriority<br>sched_setscheduler<br>sched_setparam<br><br>pthread_setschedparam<br>pthread_setschedpolicy<br>pthread_attr_setschedparam<br>pthread_attr_setschedpolicy | Context Specific |

**Mappable:** Both the Windows and Linux constructs provide similar functionality.
**Context:** The decision to use a specific Linux construct(s) depends on the application context

# Mapping WIN32 to PTHREADS
## 3. Synchronization

| Win32 Thread Level | Linux Thread Level | Linux Process Level |
|---|---|---|
| Mutex | Mutex - pthread library | System V semaphores |
| Critical section | Mutex - pthread library | N/A<br>*Critical sections are used only within threads of the same process* |
| Semaphore | Conditional Variable with mutex – pthreads POSIX semaphores | System V Semaphores |
| Event | Conditional Variable with mutex – pthreads POSIX semaphores | System V Semaphores |

**!! Conditional Variables:** While mutexes implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
See more under: https://computing.llnl.gov/tutorials/pthreads

# Mapping WIN32 to PTHREADS
## Synchronization (semaphores)

| Win32 Thread | Linux Thread | Linux Process | Classification |
|---|---|---|---|
| CreateSemaphore | sem_init | semget<br>semctl | Context specific |
| OpenSemaphore | N/A | semget | Context specific |
| WaitForSingleObject | sem_wait<br>sem_trywait | semop | Context specific |
| ReleaseSemaphore | sem_post | semop | Context specific |
| CloseHandle | sem_destroy | semctl | Context specific |

***Notice:*** Win32 semaphores are created within a thread, and propagated all over the system. POSIX semaphores are wither for inter-thread within a process, or inter-process entities.

!! **opensemaphore** function enables multiple processes to open handles of the same semaphore object. The function succeeds only if some process has already created the semaphore by using the **CreateSemaphore** function.

# Mapping WIN32 to PTHREADS
## Synchronization (events)

| Win32 Thread | Linux Thread | Linux Process | Classification |
|---|---|---|---|
| CreateEvent<br>OpenEvent | pthread_cond_init<br>sem_init | semget<br>semctl | context specific |
| SetEvent | pthread_cond_signal<br>sem_post | semop | context specific |
| ResetEvent | N/A | N/A | context specific |
| WaitForSingleObject | pthread_cond_wait<br>pthread_cond_timedwait<br>sem_wait<br>sem_trywait | semop | context specific |
| CloseHandle | pthread_cond_destroy<br>sem_destroy | semctl | context specific |

***Notice (See conclusion in the next slide).***

- Events are Windows specific objects.
- POSIX semaphores with count set to 1 provide similar functionality to the Windows events. However, they don't provide timeout in the wait functions.
- Conditional variables & Mutex in pthreads provide event-based synchronization between threads, but they are **synchronous**.

# Mapping WIN32 to PTHREADS
## Synchronization (events) - conclusion

| Win32 Threads | PThreads |
|---|---|
| Both Manual/ Auto-reset events | Only Auto-reset mechanism. |
| *Named* events to synchronization processes. Un-named events to synchronize threads. | Synchronization is inter-thread based. (System V semaphore or signals can be used) |
| Event objects initial state is set to **signaled**. | Does not provide an initial state,. *(POSIX semaphores provide initial state)* |
| Event objects are **asynchronous** | Conditional variables are **synchronous** *(POSIX/System V events are asynchronous)* |
| Timeout value can be specified | Timeout value can be specified *(Other Linux system s don't support timeout)* |

# Mapping WIN32 to PTHREADS
## Synchronization (mutex)

| Win32 Thread | Linux Thread | Linux Process | Classification |
|---|---|---|---|
| CreateMutex | pthreads_mutex_init | semget<br>semctl | context specific |
| OpenMutex | N/A | semget | context specific |
| WaitForSingleObject | pthread_mutex_lock<br>pthread_mutex_trylock | semop | context specific |
| ReleaseMutex | pthread_mutex_unlock | semop | context specific |
| CloseHandle | pthread_mutex_destroy | semctl | context specific |

***Notice (See conclusion in the next slide).***

*Some major differences reside, including:*

- Named and un-named mutexes.
- Ownershiop at creation.
- Timeout during wait.
- Recursive mutexes.

# Mapping WIN32 to LINUX
## Synchronization (mutex) - conclusion

| Win32 Threads | PThreads |
|---|---|
| *Named* mutexes synchronize process/thread<br>Un-named mutex synchronize threads. | Synchronization is inter-thread based.<br>(System V semaphore or signals can be used) |
| Can be owned during creation. | To achieve the same in Linux, a mutex should be locked explicitly after creation. |
| Recursive by default | *Pthread has recursive mutex (initialized explicitly)*<br>*(other Linux system do not allow recursive mutex)* |
| Timeout value can be specified | Timeout not available.<br>*(can be obtained via application logic)* |

# Mapping WIN32 to LINUX
## Synchronization (Critical Sections)

| Win32 Thread | Linux Thread | Classification |
|---|---|---|
| InitializeCriticalSection<br>InitializeCriticalSectionAndSpinCount | pthread_mutex_init | Mappable |
| EnterCriticalSection<br>TryEnterCriticalSection | pthread_mutex_lock<br>pthread_mutex_trylock | Mappable |
| LeaveCriticalSection | pthread_mutex_trylock | Mappable |
| DeleteCriticalSection | pthread_mutex_destroy | Mappable |

***Notice:*** Since the critical sections are used only between the threads of the same process, Pthreads mutex can be used to achieve the same **functionality** on Linux systems.

**1 -** In general, mutex objects are more CPU intensive and slower to use than critical sections due to a larger amount of bookkeeping, and the deep execution path into the kernel taken to acquire a mutex. The equivalent functions for accessing a critical section remain in thread context, and consist of merely checking and incrementing/decrementing lock-count related data. This makes critical sections light weight, fast, and easy to use for thread synchronization within a process.
**2-** In Win32 The primary benefit of using a mutex object is that a mutex can be named and shared across process boundaries.

# References

- **POSIX Threads Programming**
  https://computing.llnl.gov/tutorials/pthreads/

- **Why pthreads are better than Win32**
  http://softwarecommunity.intel.com/ISN/Community/en-US/forums/post/840096.aspx

  **Why Windows threads are better than pthreads:**
  http://softwareblogs.intel.com/2006/10/19/why-windows-threads-are-better-than-posix-threads/

- **Port Windows IPC apps to Linux (including code examples)**

**http://www.ibm.com/developerworks/linux/library/l-ipc2lin1.html**
**http://www.ibm.com/developerworks/linux/library/l-ipc2lin2.html**
**http://www.ibm.com/developerworks/linux/library/l-ipc2lin3.html**

# Thank you