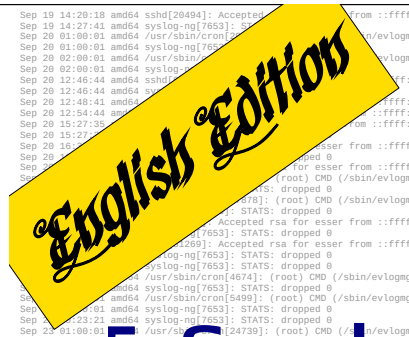

5. Synchronization (6)

5. Synchronization
5.5 Windows

Windows: Processes and Threads

- Processes
  - launching an application creates a new process
  - no **fork()** as on Unix / Linux; process creation always starts a new program
  - process defined by: address space, resources, security profile
- Threads
  - separate threads of control (just like Unix / Linux)
  - common address space (that of the process)
  - but: each process consists of at least one thread
  - scheduler activates threads (not processes!)
  - Synchronization: crit. region, mutex, event, semaphore

5.5 Synchronization on Windows

fork & exec?

- Windows has no equivalent replacement for the Unix **fork()** call
- when porting from Unix to Windows, one has to change the program logic of **fork()**-based applications
- the combination **fork()** & **exec()** for launching a new program can be replaced with **CreateProcess()**
- instead of **fork()** (without **exec**) use threads

fork *without* exec

Linux: fork()

```
main(){

...

while (TRUE) {
  read (&task);
  if (task=="new") {
    pid = fork();
    if (pid==0) { /* child process  */
      do_some_work();
      exit(0); /* terminate process */
    }
    else {        /* parent process */
    }
  }
}

...
wait();
}
```

Windows: CreateThread()

```
main(){

...

while (TRUE) {
  read (&task);
  if (task=="new") {
    CreateThread(do_some_work, ...);
  }
}

...
WaitForMultipleObjects(...)
}
```


Windows vs. Linux

- multi-threaded from the beginning
- at process creation: one thread

Process

Memory
Files
...

(no codehere)

Haupt-Thread

```
main(){
  ...
  ...
}
```

- standard: single-threaded
- multi-threaded on demand

Process

Memory
Files
...

```
main(){
  ...
  ...
}
```


Windows vs. Linux

- Difference between Windows and Linux w.r.t. threads/processes sort of theoretical:
- Linux internally treats threads and processes alike (everything is a „task“), thus:
  - non-threaded process on Linux corresponds to Windows process with one (single) main thread
  - threaded Linux process with *n* threads corresponds to Windows process with *n*+1 threads (1 main thread plus *n* further threads)

Windows vs. Linux

after **CreateThread()**

Process

Memory
Files
...

Main Thread

```
main(){
  ...
  ...
}
```

2nd Thread

```
thr(){
  ...
  ...
}
```

after **pthread_create()**

Process

Memory
Files
...

```
main(){
  ...
  ...
}
```

Thread

Memory
Files
...

```
thr(){
  ...
  ...
}
```

Fibers

- fibers are special user level threads (within a normal thread)
- are not serviced by the scheduler
- usage:
  - before first fiber use: **ConvertThreadToFiber()**
  - create further fibers: **CreateFiber()**
  - make another fiber run: **SwitchToFiber()**
- example code:
  http://msdn.microsoft.com/library/en-us/dllproc/base/using_fibers.asp

Process Creation (1)

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( VOID ) {
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    LPTSTR szCmdline=_tcsdup(TEXT("MyChildProcess"));

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        szCmdline,      // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    ) {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

Source: http://msdn.microsoft.com/library/en-us/dllproc/base/creating_processes.asp

Process Creation (2)

```
typedef struct
_PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD dwProcessId;
  DWORD dwThreadId;
} PROCESS_INFORMATION,
 *LPPROCESS_INFORMATION;
```

hProcess — A handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread — A handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId — A value that can be used to identify a process. The value is valid from the time the process is created until all handles to the process are closed and the process object is freed; at this point, the identifier may be reused.

dwThreadId — A value that can be used to identify a thread. The value is valid from the time the thread is created until all handles to the thread are closed and the thread object is freed; at this point, the identifier may be reused.

Source: http://msdn.microsoft.com/library/en-us/dllproc/base/process_information_str.asp

Thread Creation (1)

- similar to POSIX threads:
  - thread creation call expects a function that will run inside the new thread
  - new thread uses the address space of the process
  - it is possible to wait for termination of a thread

Thread Creation (2)

```
#include <windows.h>
#include <strsafe.h>
#define MAX_THREADS 3
#define BUF_SIZE 255
typedef struct _MyData { int val1; int val2; } MYDATA, *PMYDATA;

DWORD WINAPI ThreadProc( LPVOID lpParam ) {
    HANDLE hStdout;
    PMYDATA pData;
    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.
    pData = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.
    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"), pData->val1, pData->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, cchStringSize, &dwChars, NULL);

    // Free the memory allocated by the caller for the thread data structure.
    HeapFree(GetProcessHeap(), 0, pData);

    return 0;
}
```


Thread Creation (3)

```
void main() {
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;
    // Create MAX_THREADS worker threads.
    for( i=0; i<MAX_THREADS; i++ ) {
        // Allocate memory for thread data.
        pData = (PMYDATA) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, sizeof(MYDATA));
        if( pData == NULL ) ExitProcess(2);
        // Generate unique data for each thread.
        pData->val1 = i;
        pData->val2 = i+100;

        hThread[i] = CreateThread(
            NULL,              // default security attributes
            0,                 // use default stack size
            ThreadProc,        // thread function
            pData,             // argument to thread function
            0,                 // use default creation flags
            &dwThreadId[i]);   // returns the thread identifier

        // Check the return value for success.
        if (hThread[i] == NULL) { ExitProcess(i); }
    }
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);
    // Close all thread handles upon completion.
    for(i=0; i<MAX_THREADS; i++) { CloseHandle(hThread[i]); }
}
```

Source: http://msdn.microsoft.com/library/en-us/dllproc/base/creating_threads.asp

Thread Synchronization

synchronize threads of one single process	synchronize several processes

• **critical sections**
`EnterCriticalSection();`
`LeaveCriticalSection();`
	• **mutexes**
	`hMutex = CreateMutex(...);`
	`WaitForSingleObject(hMutex,...);`
	`ReleaseMutex(hMutex);`
	• **semaphores**
	`hSem = CreateSemaphore(...);`
	`WaitForSingleObject(hSem,...);`
	`ReleaseSemaphore(hSem);`
	• **events**
	`hEvent = CreateEvent(...);`
	`WaitForSingleObject(hEvent);`
	`SetEvent(hEvent);`
	`ResetEvent(hEvent);`

Critical Sections (1)

- critical sections can only be used by the threads of **one** process
- only one thread can enter a critical section at any time (mutual exclusion)
- critical sections are initialized, but have no handle
- recursive: thread may call **EnterCriticalSection** several times, but must leave the section using **LeaveCriticalSection** as often as entering it
- there is no function that queries whether the critical section is used by another thread

Critical Sections (2)

- datatype: LPCRITICAL_SECTION
- create and destroy:

```
InitializeCriticalSection( LPCRITICAL_SECTION sec );
DeleteCriticalSection( LPCRITICAL_SECTION sec );
```

- enter:

```
IEnterCriticalSection( LPCRITICAL_SECTION sec );
bool TryEnterCriticalSection ( LPCRITICAL_SECTION sec );
```

- leave:

```
LeaveCriticalSection( LPCRITICAL_SECTION sec );
```


Critical Sections (3)

```
// Global variable
CRITICAL_SECTION CriticalSection;

void main() {
    ...
    // Initialize the critical section one time only.
    if (!InitializeCriticalSection(&CriticalSection)) return;
    ...
    // Release resources used by the critical section object.
    DeleteCriticalSection(&CriticalSection)
}

DWORD WINAPI ThreadProc( LPVOID lpParameter ) {
    ...
    // Request ownership of the critical section.
    EnterCriticalSection(&CriticalSection);
    // Access the shared resource.
    // Release ownership of the critical section.
    LeaveCriticalSection(&CriticalSection);
    ...
}
```


Synchronization Objects (1)

- Windows kernel knows Synchronization Objects which can be in one of two states:
  - *Signalled* State
  - *Unsignalled* State
- threads can use the Wait Services of the object manager to synchronize with one or many objects:
  - thread blocks until the synchr. object changes to Signalled State
  - when the kernel sets a synchr. object to *Signalled* it makes those threads runnable which wait on it

Synchronization Objects (2)

- the Win32 API supplies two functions for accessing the Wait Services:
  - **WaitForSingleObject()**
  - **WaitForMultipleObjects()**
    - choose whether the thread shall wait for one of the objects or for all of them (atomically!)
  - both functions accept a Timeout argument.

Synchronization Objects (3)

Overview

object type	Signalled State when...	system wakes up...
process	process terminates	all threads
thread	thread terminates	all threads
file	an I/O operation for this file terminates	all threads
mutex	the mutex is held by no thread	one thread
semaphore	the semaphore value is >0	several threads
event	a thread explicitly sets the event	depends on event type
timer	when the specific point in time is reached	depends on timer type

Mutexes (1)

- named / unnamed mutexes for mutual exclusion; type: *HANDLE*
- for threads within one process or for several processes (named mutexes)
- Mutex operations
  - create: **CreateMutex()**
  - aquire: **WaitForSingleObject()**
  - release: **ReleaseMutex()**

Mutexes (2)

- mutex handle can be passed to a child process when calling **CreateProcess()**
- when several processes separately create the same named mutex using **CreateMutex()**,
  - the first call will create the mutex,
  - the next one will enable shared access to this mutex
- all threads of a process can access the mutex

Mutexes (3)

```
#include <windows.h>
#include <stdio.h>

HANDLE hMutex;

// Create a mutex with no initial owner.
hMutex = CreateMutex(
    NULL,     // default security attributes
    FALSE,    // initially not owned
    NULL);    // unnamed mutex

if (hMutex == NULL) { printf("CreateMutex error: %d\n", GetLastError()); }
```

http://msdn.microsoft.com/library/en-us/dllproc/base/using_mutex_objects.asp

Mutexes (4)

```
BOOL FunctionToWriteToDatabase(HANDLE hMutex) {
    DWORD dwWaitResult;
    // Request ownership of mutex
    dwWaitResult = WaitForSingleObject(
        hMutex,   // handle to mutex
        5000L);   // five-second time-out interval

    switch (dwWaitResult) {
        case WAIT_OBJECT_0:  // The thread got mutex ownership
            __try     { /* Write to the database */ }
            __finally { /* Release ownership of the mutex object */
                if (! ReleaseMutex(hMutex)) { /* Deal with error */ } }
            break;
        case WAIT_TIMEOUT:   // Cannot get mutex ownership due to time-out
            return FALSE;
        case WAIT_ABANDONED: // Got ownership of the abandoned mutex object
            return FALSE;
    }
    return TRUE;
}
```


Mutexes (5)

Process 1

```
main () {
  HANDLE hMutex;
  LPCTSTR MutexName;

  MutexName = "MeinTestMutex";
  hMutex = CreateMutex(NULL,
             FALSE, MutexName);
  if (hMutex == NULL) {
    printf("CreateMutex error: %d\n",
    GetLastError());
  }

  while (TRUE) {
    dwWaitResult =
    WaitForSingleObject(hMutex, ...);
    ...
    /* crit. sect.: read access */
    ReleaseMutex(hMutex);
  }
}
```

Process 2

```
main () {
  HANDLE hMutex;
  LPCTSTR MutexName;

  MutexName = "MeinTestMutex";
  hMutex = CreateMutex(NULL,
             FALSE, MutexName);
  if (hMutex == NULL) {
    printf("CreateMutex error: %d\n",
    GetLastError());
  }

  while (TRUE) {
    dwWaitResult =
    WaitForSingleObject(hMutex, ...);
    ...
    /* crit. sect.: write access */
    ReleaseMutex(hMutex);
  }
}
```


Mutexes (6)

when a thread terminates without releasing a mutex it holds,

- the mutex is regarded as abandoned,
- a thread that waits for this mutex will be awoken,
- the woken-up thread will be informed about the abandoned state by returning a corresponding value (in the WaitFor... call).

Mutexes (7)

Comparison of Linux and Windows mutexes

Posix thread mutexes	Windows mutexes
pthread_mutex_init()	CreateMutex()
pthread_mutex_destroy()	CloseHandle()
pthread_mutex_lock()	WaitForSingleObject()
pthread_mutex_unlock()	ReleaseMutex()
pthread_mutex_trylock()	WaitForSingleObject(timeout=0)

Semaphores (1)

similar properties as mutexes have:

- named / unnamed semaphores;
  type: *HANDLE*
- for threads within one process or
  for several processes (named semaphores)
- semaphore operations
  - create: **CreateSemaphore()**
  - **Wait** operation: **WaitForSingleObject()**
  - **Signal** operation: **ReleaseSemaphore()**

Semaphores (2)

```
HANDLE hSemaphore;
LONG cMax = 10;

// Create a semaphore with initial and max. counts of 10.

hSemaphore = CreateSemaphore(
    NULL,   // default security attributes
    cMax,   // initial count
    cMax,   // maximum count
    NULL);  // unnamed semaphore

if (hSemaphore == NULL)
{
    printf("CreateSemaphore error: %d\n", GetLastError());
}
```

http://msdn.microsoft.com/library/en-us/dllproc/base/using_semaphore_objects.asp

Semaphores (3)

```
DWORD dwWaitResult;
// Try to enter the semaphore gate
dwWaitResult = WaitForSingleObject(
        hSemaphore,   // handle to semaphore
        0L);          // zero-second time-out interval
switch (dwWaitResult) {
    case WAIT_OBJECT_0:  // The semaphore object was signaled
        // OK to open another window
        break;
    case WAIT_TIMEOUT:   // Semaphore was nonsignaled; a time-out occurred
        // Cannot open another window
        break;
}
...

// Increment the count of the semaphore
if (!ReleaseSemaphore(
        hSemaphore,  // handle to semaphore
        1,           // increase count by one
        NULL) )      // not interested in previous count
{ printf("ReleaseSemaphore error: %d\n", GetLastError()); }
```


Semaphores (4)

Consumer Producer Problem with named semaphores

```
/* consumer.c */

#define NUM_LOOPS 20 /* no. of iterations    */

int main() {
  int i;             /* Counter for loop     */
  LPCTSTR SemName;   /* Semaphore name       */
  HANDLE hSem;       /* Semaphore            */
  /* create named semaphore; Init 0, Max. 1  */
  SemName = "ProducerConsumerSem";
  hSem = CreateSemaphore(NULL,0,1,SemName);

  for (i=0; i<NUM_LOOPS; i++) {
    /* block unless counter>0    */
    WaitForSingleObject(hSem);
    printf("consumer: '%d'\n", i);
  }
  return 0;
}
```

```
/* producer.c */

#define NUM_LOOPS 20

int main() {
  int i;
  LPCTSTR SemName;
  HANDLE hSem;

  SemName = "ProducerConsumerSem";
  hSem = CreateSemaphore(NULL,0,1,SemName);

  for (i=0; i<NUM_LOOPS; i++) {
    printf("consumer: '%d'\n", i);
    /* increase semaphore counter */
    ReleaseSemaphore(hSem);
  }
  return 0;
}
```

Semaphores (5)

Comparison of Linux and Windows Semaphores

Posix thread semaphores	Windows semaphores
sem_init()	CreateSemaphore()
sem_wait()	WaitForSingleObject()
sem_trywait()	WaitForSingleObject(timeout=0)
sem_post()	ReleaseSemaphore()
sem_destroy()	CloseHandle()

Events (1)

- inform threads about occurrence of an event
- idea is here more like this: when something special happens, run a function (instead of preventing this)
  - waiting for event is regular state of the thread (unlike mutexes, semaphores, etc.)
- create a new event with **CreateEvent()**

```
HANDLE WINAPI CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes,
  BOOL bManualReset,     /* TRUE, FALSE */
  BOOL bInitialState,    /* TRUE: signalled, FALSE */
  LPCTSTR lpName         /* event name */
);
```


Events (2)

- **automatic event:**
  - each call of **WaitForSingleResource**(event) automatically resets the event.
  - if several threads are waiting, only one of them will return from its **Wait**...() call when **SetEvent**() is called
- **manual event:**
  - Event must be reset manually using **ResetEvent**().
  - if several threads are waiting, all of them will return from their **Wait**...() calls when **SetEvent**() is called.

Events (3)

Example:
- Event 1: thread has created new data
- Event 2: main thread has read the data

```
#include <windows.h>
#include <process.h>
#include <stdio.h>

HANDLE hEvent1, hEvent2;
int a[5];

void Thread( void* pParams ) {
  int i, num = 0;

  while (TRUE) {
    WaitForSingleObject(hEvent2,INFINITE);
    for ( i = 0; i < 5; i++ ) a[i] = num;
    SetEvent(hEvent1);
    num++;
  }
}
```

```
int main( void ) {
  hEvent1 = CreateEvent(NULL,FALSE,TRUE,NULL);
  hEvent2 = CreateEvent(NULL,FALSE,FALSE,NULL);

  _beginthread( Thread, 0, NULL );

  while( TRUE ) {
    WaitForSingleObject(hEvent1,INFINITE);
    printf( "%d %d %d %d %d\n",
            a[ 0 ], a[ 1 ], a[ 2 ],
            a[ 3 ], a[ 4 ] );
    SetEvent(hEvent2);
  }
  return 0;
}
```

Source: http://www.codeproject.com/threads/sync.asp

Synchr. in the Windows Kernel

- atomical Test-and-Set operations
- spinlocks: while a lock is unavailable, wait actively
- queued spinlocks: spinlocks with FIFO queue
- since 2003: so-called „Pushlocks“ (Windows name for reader writer spinlocks)