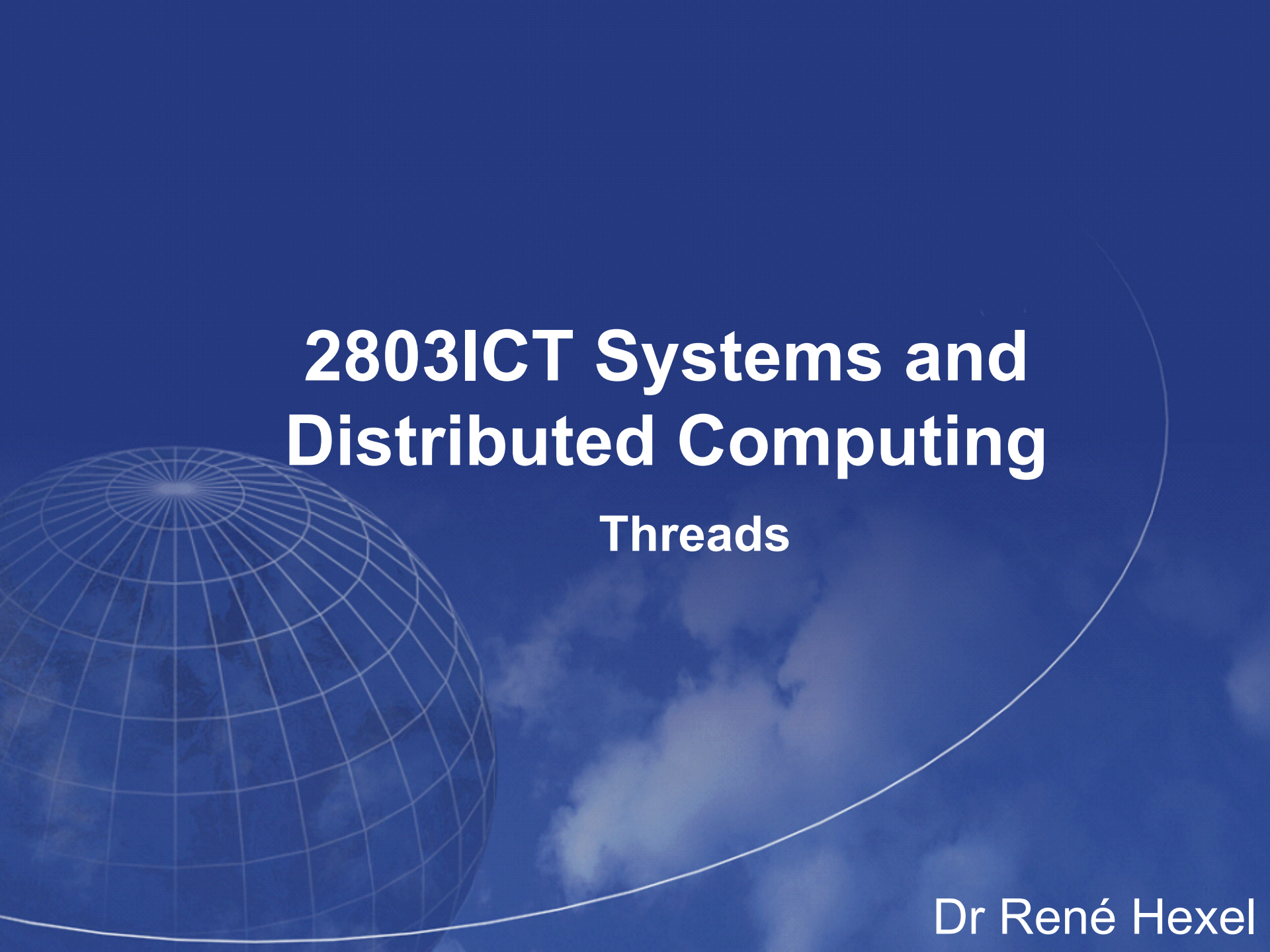


2803ICT Systems and Distributed Computing

Threads

Dr René Hexel

Lecture Outline

1. Concurrency
2. Processes or Threads
3. Unix Threads
4. Windows Threads
5. Multiprocessing
6. Writing Portable Code
7. Server Design

© Ruben Gonzalez 1999-2010, © René Hexel 2018

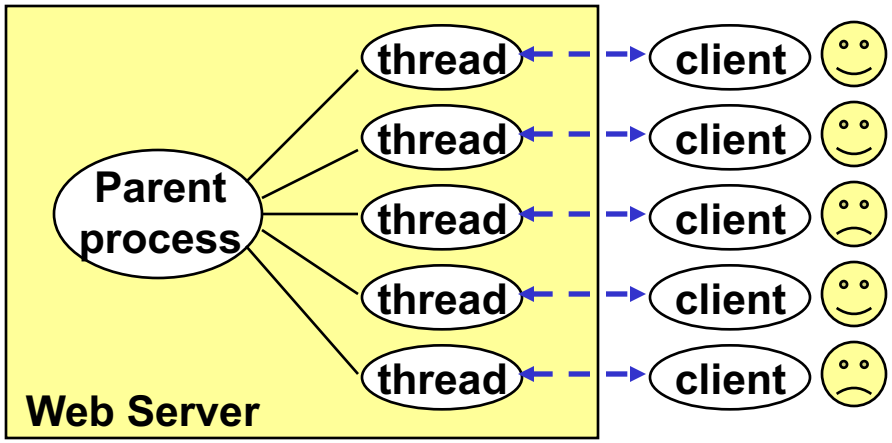
Motivations for Concurrency

- Concurrency is running multiple functions or tasks at the same time
- The main reasons for concurrency are:
- **Performance**
  - Many hard problems need a lot of processing
  - Better utilise widespread multiprocessor systems
- **Software design**
  - More naturally expresses inherently parallel tasks
  - EG webservers, web page with multiple animations
  - EG Pagemaker uses 7 threads
    - Event handling
    - Import Thread
    - Service Thread
    - Screen redraw
    - Autoflow
    - Initialisation
    - Printing

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Motivations for Concurrency

- Example: Web Browser
  - Downloading pages in background
  - Repaginating / layout of pages
  - Playing Multiple Animated gifs
- Web Server Design



© Ruben Gonzalez 1999-2010, © René Hexel 2018

Advantages of Concurrency – 1

```
//- Example: a program to sort a string of unordered integers.
#define NUM    1000;
int    array[NUM];

void sort(int low, int high)          //- ugly bubble sort method
{
   for(int i = low; i < high-1; i++)
        for(int j = i+1; j < high; j++)        //- nested for loop
               if ( array[j] < array[i] )
                {
                      int temp = array[j];
                      array[j] = array[I];
                      array[I] = temp;
               }
}

void main()
{
   for(int n=0; n<NUM; n++) scanf("%d",array[n]);
   sort(0, NUM);
   for( n=0; n < NUM; n++) printf("%d ", array[n]);
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Advantages of Concurrency – 2

- Simple complexity analysis shows that the number of comparisons in the sort is n(n-1)/2 or approx. $n^2/2$
- If n were divided into halves (ie. 2 sets of n/2 numbers) we could arrange for separate computers to sort each half simultaneously. Both halves must then be merged
- In this case sorting n/2 elements requires $(n/2)^2/2 = n^2/8$ comparisons (which is 4 x faster)
- Even if each half were sorted sequentially and then a merge performed the entire sort would only be $n^2/4$ (ie twice as fast as the merge takes n comparisons).

N	$N^2/2$	$(n^2/4) + n$	$(n^2/8) + n$
40	800	440	140
100	5000	2600	1350
1000	500 000	251 000	126 000

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Advantages of Concurrency – 3

```
//- A function to merge two sorted halves :
#define         NUM     40;
int array[2*NUM];

void merge(int low, int mid, int high)
{
   int  cnt1 = low;
   int  cnt2 = mid;
   while (cnt1 < mid)                    //- one loop
        if ( array[cnt1] < array[cnt2] )
         {
                printf("%d ", array[cnt1++] );
                if (cnt1 > mid)
                        for (int idx2 = cnt2; idx2 <  high)
                                printf("%d ", array[idx2++] );
        }
        else {
                printf("%d ", array[cnt2++] );
                if (cnt2 > high)
                        for (int idx1 = cnt1; idx1 <  mid-1)
                                printf("%d ", array[idx1++] );
        }
   }
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Advantages of Concurrency – 4

```
void main()
{
   for (int n = 0; n < 2*NUM; n++) read(array[n]);
   sort(1,n);
   sort(n+1, 2*NUM)
   merge(1, NUM, 2*NUM);
}
```

- To run both sorts in parallel we might say :

```
void main()
{
   for (int n = 0; n < 2*NUM; n++) read(array[n]);
   START_THREAD( sort(1,n) );
   START_THREAD(  sort(n+1, 2*NUM) );
   merge(1, NUM, 2*NUM);
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

The Problem of Concurrency

- The basic problem involves resources:
  - Single CPU, single DRAM, single I/O devices
  - Each process wants to have exclusive access to all
  - How can OS coordinate all activity
- Basic Idea: Virtual CPU abstraction
  - Simple machine abstraction for processes
  - Multiplex/interleave these abstract machines
- Each virtual “CPU” needs a structure to hold:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one CPU to the next?
  - Save PC, SP, and registers in current state block
  - Load PC, SP, and registers from new state block

© Ruben Gonzalez 1999-2010, © René Hexel 2018

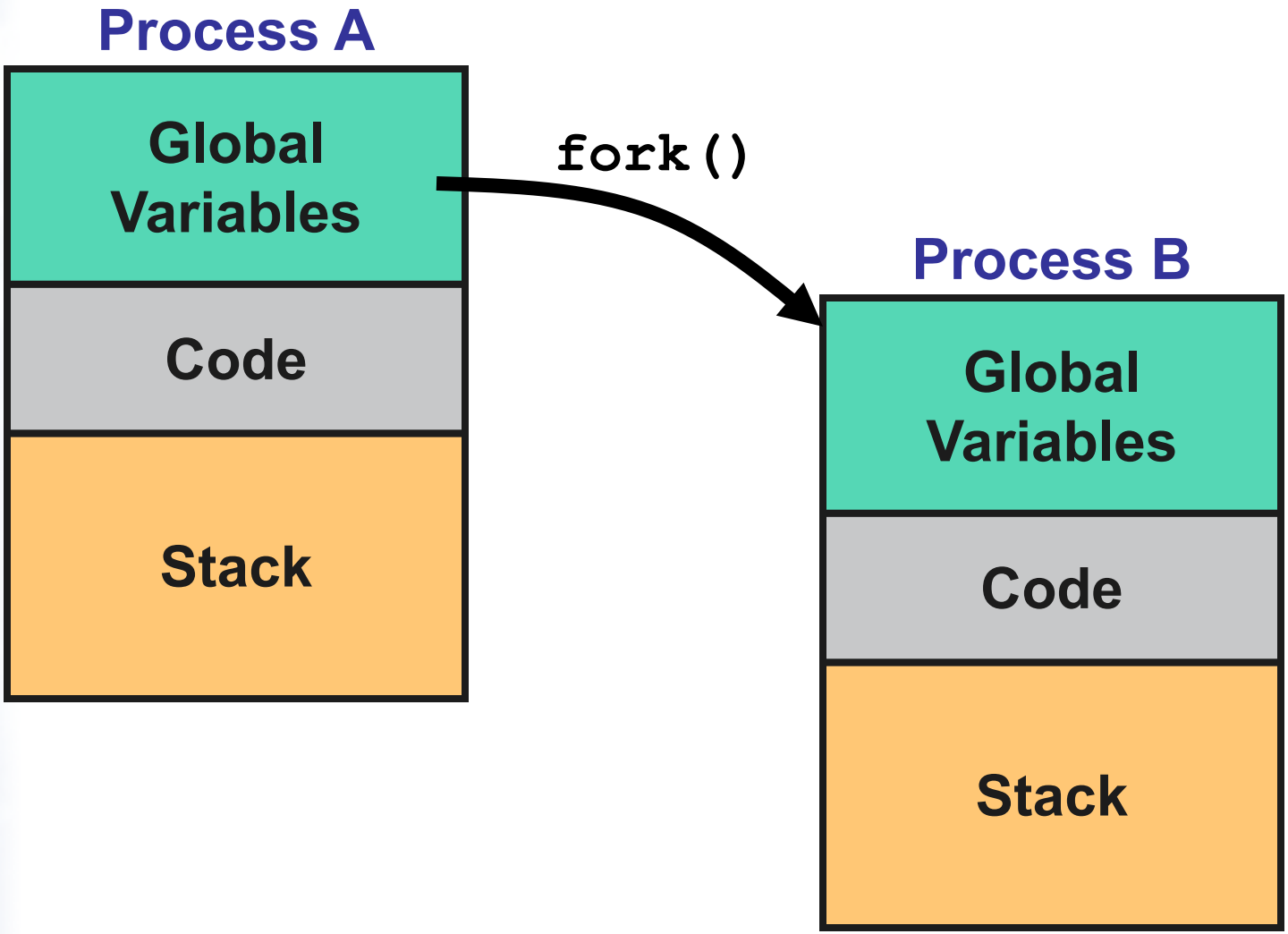
Threads vs. Processes

- Creation of a new process using fork() is *expensive* (time & memory).
  - Must construct new PCB
  - Must set up new page tables for address space
  - Copy data from parent process? (Unix **`fork()`**)
  - Copy I/O state (file handles, etc)

- A thread requires less memory and less startup time
  - Threads are sometimes called *lightweight process*

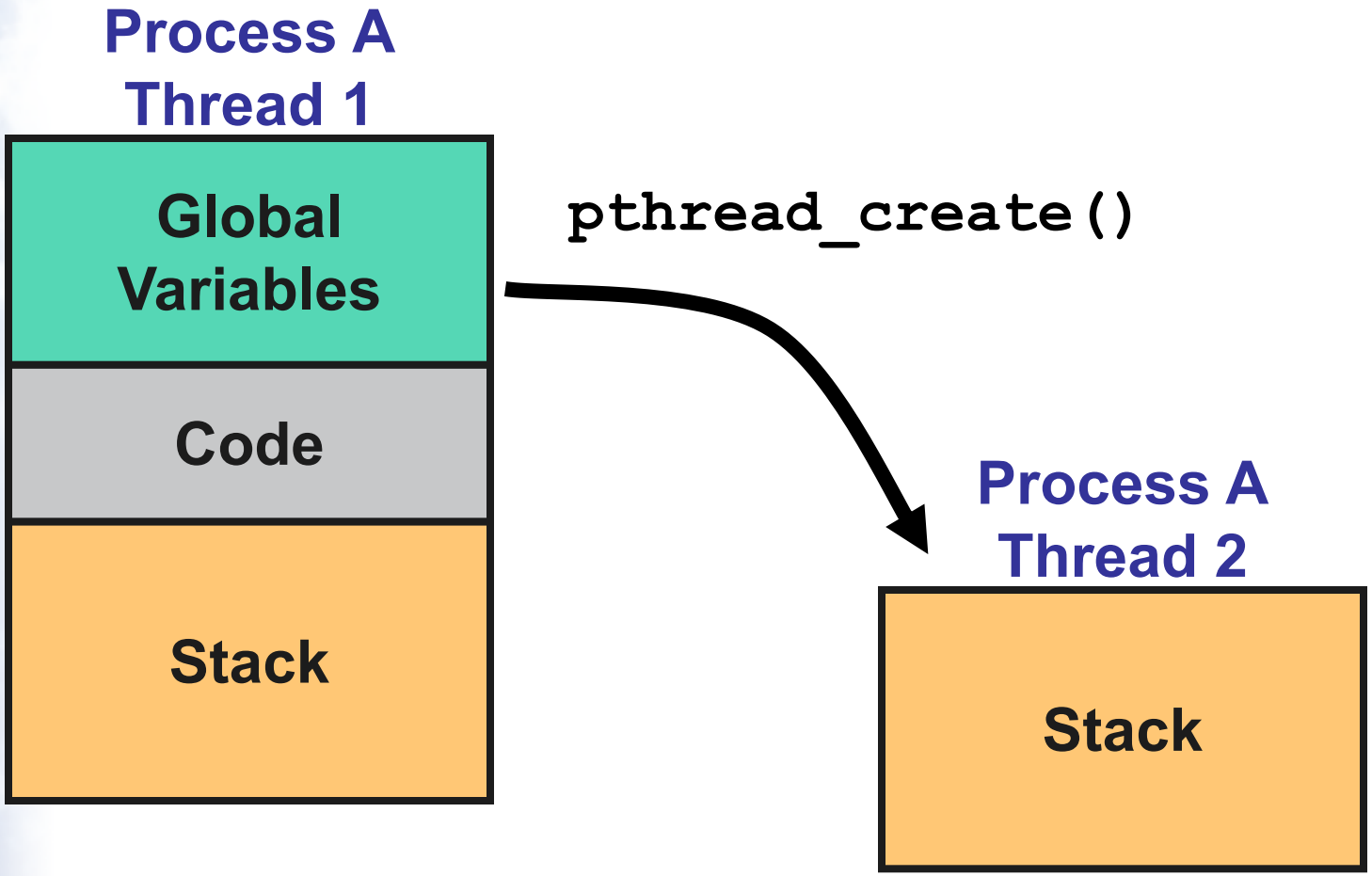
© Ruben Gonzalez 1999-2010, © René Hexel 2018

fork()



© Ruben Gonzalez 1999-2010, © René Hexel 2018

thread_create()



© Ruben Gonzalez 1999-2010, © René Hexel 2018

Benefits of Threads

- Creating a new thread takes less time than creating a process (upto 100x faster).
- Less time to terminate a thread than a process
- A process can have multiple threads
- Switching between two threads takes less time that switching processes
- Threads can directly communicate with each other - without invoking the kernel

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Disadvantages of Threads

While processes have protected access to

- Processors,
- Other processes,
- Files, Memory
- I/O resources

All threads of a process share:

- Process memory (program code and global data)
- Resource handles, open file/socket descriptors
- working environment (current dir, user ID, etc.)
- signal handlers and signal dispositions

A rogue thread can trash all others in a process

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread-Specific Resources

Each thread has it's own:

- Thread ID (integer)
- Stack, Registers, Program Counter
- An execution state (running, ready, etc.)
- Saved thread context when not running
- thread local storage for "global" variables
- **`eg. errno`** (if not - **`errno`** would be useless!)

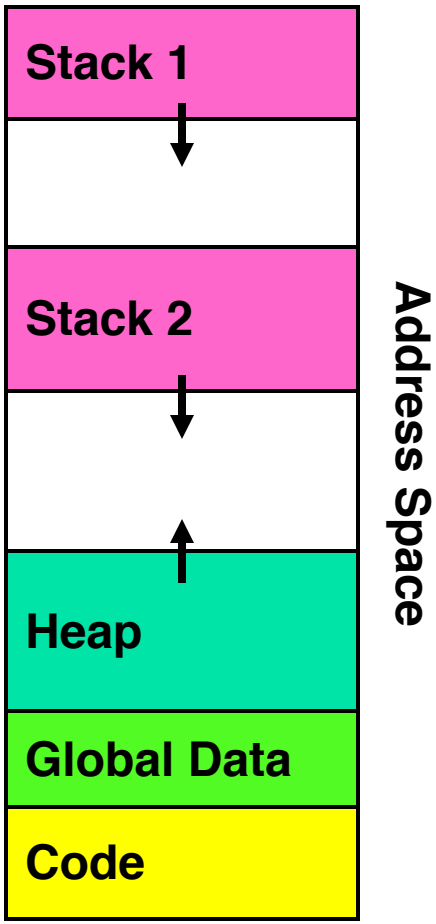
Threads within one process can communicate using global variables in shared memory

Must be done carefully!

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Memory Footprint

- If we stopped a two-threaded program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks



© Ruben Gonzalez 1999-2010, © René Hexel 2018

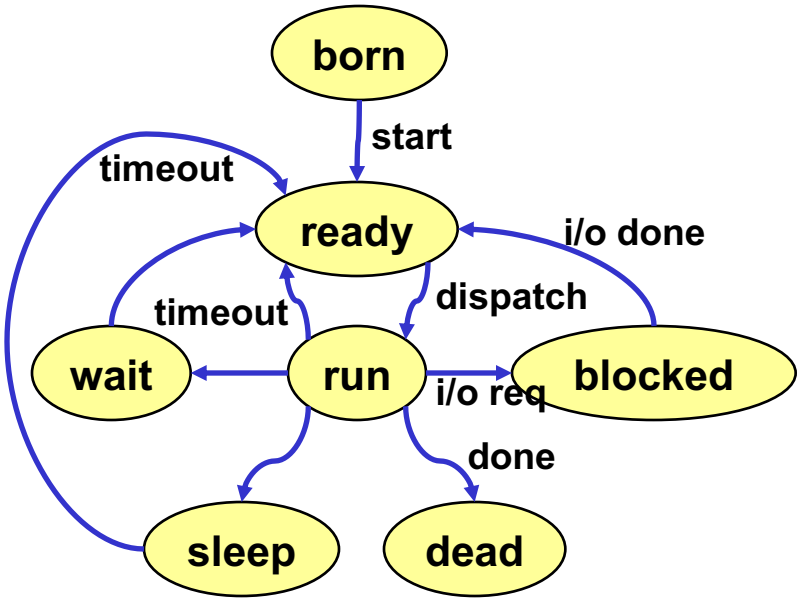
Thread State

- Each thread has a *Thread Control Block* (TCB) that is used to store:
  - Execution State: CPU registers, program counter, pointer to stack
  - Scheduling info: Execution state, priority, CPU time
  - Pointer to enclosing processes resources (PCB)
  - Accounting Info:
  - Various Pointers (for implementing scheduling queues)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Execution

- Threads and processes have many operations in common (e.g. create, exit, resume, suspend)
- Like processes each thread can have different execution states
  - Born/Created
  - Ready state
  - Running state
  - Blocked state
  - Waiting state
  - Sleeping state
  - Dead (Terminated) state



© Ruben Gonzalez 1999-2010, © René Hexel 2018

Sharing Global Variables

WARNING

- Since threads in a process all share the same memory they all have access to any global or static variables
- Sharing global variables is dangerous - two threads may attempt to modify the same variable at the same time.
- ***Just because you don't see a problem when you run your code doesn't mean it won't happen at some later time.***
- You also have to be careful with libraries - If a function uses any static variables (or global memory) it's not safe to use with threads!

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Beware

- Threads are powerful, but dangerous
- You have to pay attention to details or it's easy to end up with code that doesn't always work, or hangs in deadlock.

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread vs Process Summary

- **Process:** unit of resource allocation
  - Owns memory (address space)
  - Owns file descriptors, file system context, …
  - Has one or more threads sharing process resources
- **Thread:** unit of dispatch (execution)
  - Shares process' memory and other resources
- Why use **processes** instead of threads?
  - If you need protection more than performance
  - Less tricky to program
- Why use **threads** instead of processes?
  - Threads are faster - more efficient than processes
  - If you need faster communication between tasks
  - If you need more interaction between tasks

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Models

- Threads may be managed by the operating system or by a user application
- Three most popular threading models
  - User-level threads
  - Kernel-level threads
  - Combination of user-and kernel-level threads

© Ruben Gonzalez 1999-2010, © René Hexel 2018

User-level Threads (ULT)

- Perform threading operations in user space
  - Threads are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly
- User-level thread implementation
  - The kernel is not aware of these threads and does not schedule them individually
  - The kernel only schedules the process as a single entity
  - The user library has to manage/schedule the threads itself
  - Advantages
    - More portable
    - User-level libraries can schedule its threads as it wants
  - Disadvantage
    - If a thread blocks – all the threads in the process will be blocked
    - Cannot be scheduled on multiple processors at once
  - Example : Pthread Library (used as Hybrid in Unix/Linux etc)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Kernel-level Threads (KLT)

- These are individually managed by the kernel
- Scheduling is done on a thread basis
- Advantages:
  - Increased scalability by scheduling multiple threads onto multiple processors
  - More Efficient / Higher throughput - If one thread is blocked, the remaining threads of the same process keep running
- Disadvantages:
  - Context switching overhead
  - Reduced portability due to OS-specific APIs
- Example: Windows

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Hybrid Threads

- The combination of ULT and KLTs
  - Number of user and kernel threads need not be equal
  - Can reduce overhead compared to one-to-one thread mappings by implementing thread pooling
  - Retains the advantages of KLTs
- Worker threads
  - Like 5 children taking turns sharing 3 bicycles
  - Persistent kernel threads that occupy the thread pool
  - Improves performance in environments where threads are frequently created and destroyed
- Example:
  - Linux does not differentiate between threads and processes.
  - Linux maps ULTs into “lightweight” kernel processes that share the same group ID.

© Ruben Gonzalez 1999-2010, © René Hexel 2018


POSIX Threads

Posix Threads

- ***pthreads*** is the most widely supported User-level Thread library.
- Available for most operating systems
- On Linux you need to link with **"-lpthread"**
  - On many systems this also forces the compiler to link in *re-entrant* libraries (instead of plain vanilla C libraries).
- There is a Windows implementation
  - Download from http://sources.redhat.com/pthreads-win32/
  - Win32 pthreads is implemented as a dynamic link library (DLL), and a normal static library (LIB)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Creation

pthread_create(pthread_t *_tid_, const pthread_attr_t *_attr_,
void *(*_func_)(_void *_), _void_ *arg_);

- **Returns** –
  - zero if successful, otherwise nonzero error code
  - does not set errno
- ***tid*** – a pointer to where the thread ID will be stored.
- ***attr*** – use NULL for default system attributes
- ***func*** – a pointer to the function to be called,
  - func() must take a single void * argument and returns a void*.
  - When func() returns the thread is terminated.
- ***arg*** – NULL or a (void*) input parameter for func().

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Passing Arguments to Threads

- You can pass complex parameters to the thread by creating a structure and passing the address of the structure to the thread.
  - See example next slide
- The structure can't be a local variable (of the function calling pthread_create)!! - threads have different stacks!

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Arguments Example

```
struct { int x,y } Point;
void * PrintXY( void *arg)
{
   struct Point *pt = (struct Point *) arg;
   printf("TID=%u; pt=(%d,%d)\n", pthread_self(), pt->x, pt->y);
   return NULL ;
}

int main (int argc, char *argv[])
{
   pthread_t thread;
   struct Point pt = {1,2};
   int rc = pthread_create(&thread, NULL, PrintXY, (void *)&pt);
   if (rc) printf("ERROR = %d\n", rc);
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Setting Thread Attributes

- The ***attr*** paramter in pthread_create is a pointer to a pthread_attr_t structure that has been initialised (before creating the thread) using…

  int pthread_attr_init(pthread_attr_t structure * attr);

- When done the structure should be uninitialised

  int pthread_attr_destroy(pthread_attr_t structure *);

- Available attributes
  - detachstate
  - stackaddr
  - stacksize
  - guardsize

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Setting Thread Attributes

- Thread attributes in the pthread_attr_t structure are set through API functions
- *detachstate* can be one of the following
  - PTHREAD_CREATE_DETACHED – creates an orphan thread
  - PTHREAD_CREATE_JOINABLE – allows termination status to be retrieved
  - To change the value of the attributes use:

```
int pthread_attr_setdetachstate (pthread_attr_t* attr,
                                 int detachstate);
int pthread_attr_getdetachstate (pthread_attr_t* attr,
                                 int* detachstate);
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Setting Thread Attributes

- *stackaddr* and *stacksize* can be changed using
  - In versions < 3.0

    int pthread_attr_getstacksize (pthread_attr_t*, int* *stacksize*);
    int pthread_attr_setstacksize (pthread_attr_t* *attr*, int *stacksize*);
    int pthread_attr_getstackaddr (pthread_attr_t* *attr*,
    int *stackaddr);
    attr int pthread_attr_setstackaddr (pthread_attr_t* *attr*, int *stackaddr*);
  - In more recent versions

    int pthread_attr_getstack (pthread_attr_t* *attr*,
    void** *stackaddr*, int* *stksize*);
    int pthread_attr_setstack (pthread_attr_t* *attr*,
    void* *stackaddr*, int* *stksize*);
- There are other attributes that are not set using the pthread_attr_t structure

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Detached and Joinable States

- A thread can be either *joinable* or *detached*.
- **Detached:**
  - On termination all thread resources are released by the OS.
  - A detached thread cannot be joined – can't wait for it to end
  - A detached thread does not return a value
  - A thread can be detached by calling

```
int pthread_detach(pthread_t tid);
```

- **Joinable:**
  - When the thread ends its ID and exit status are saved by the OS and will be returned to any "joined" thread in the ***status***.
  - Joining a thread means that the current thread will wait (blocks) until the joined thread terminates.
  - A thread is joined by calling:

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

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: Joining Threads

```
#include <pthread.h>
void* ThrdFn(void *arg)
{
  printf("Hi I'm %d\n",pthread_self());
}

main()
{
  pthread_t t1, t2;
  pthread_create(&t1,NULL, ThrdFn, NULL);
  pthread_create(&t2,NULL, ThrdFn, NULL);
  pthread_join(t1,NULL);  //- wait for t1
  pthread_join(t2,NULL);  //- wait for t2
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread IDs

- You can get the thread's unique ID, by calling

```
pthread_t pthread_self(void);
```

- You can print its value by assuming it is an unsigned int

```
printf("Thread %u:\n",pthread_self());
```

- You should not use the C language '=='operator to compare two thread IDs, instead use

```
int pthread_equal(pthread tid1, pthread tid2)
```

- This returns 0 if the two IDs are different 0, otherwise a non-zero value is returned.

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Termination

- Once a thread is created, it starts executing the specified function `func()`.
- If `func()` returns, the thread is terminated.
- If `main()` returns or any thread calls `exit()` all threads are terminated.
- A thread can also be terminated by calling

```
void pthread_exit(void* ptrvalue);
```

  - The thread returns ***ptrvalue*** to the calling function
  - Any files opened inside the thread will remain open after the thread is terminated.
- One thread can request the death of another..

```
void pthread_cancel(pthread_t tid);
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Termination

- Pthreads permits a function to be setup to be called when a thread exits
- Works similar to the `atexit()` function used for process termination
- Uses two functions - both are required

```
void pthread_cleanup_push(void (*fn)(void*),
                          void* arg);
void pthread_cleanup_pop(int execute);
```

- The cleanup function fn(void*) is called with argument arg given to it whenever
  - a thread calls pthread_exit,
  - A thread accepts a cancel
  - *_cleanup_pop() is called with a nonzero argument

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: Count and Print

```
#define NUM 50
int counter = 0;
void *printer(void *arg)
{
    for (int i=0;i<NUM;i++)
    {
          printf("count = %d\n",counter);
          sleep(1);
    }
}
main()
{
    pthread_t tid;
    pthread_create(&tid,NULL,printer,NULL);
    for (int i=0;i<NUM;i++)
    {
          counter++;          //- increment count
          sleep(1);
    }
    pthread_join(tid, NULL);//- wait
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: Add and Subtract

```
//-------------------------------------------------------------------------
//- two threads writing to shared variable 'counter'
//- the value of counter should always be either 0 or 10
//-------------------------------------------------------------------------
int counter = 0;
void *adder(void *arg)
{
     while (1)
     {
            counter += (int)arg;
            printf("count = %d\n",counter);
     }
}
main()
{
     pthread_t t1, t2;
     pthread_create(&t1,NULL,adder,(void*)10);
     pthread_create(&t2,NULL,adder,(void*)-10);
     sleep(10);     //- let threads run for 10 secs
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Specific Global Data

- All threads share access to global data in the heap
- All threads have private variables stored on their stack
- Thread-local storage lets threads have their own private global data variables
- Why do you need it? (**BEST AVOIDED**)
  - Mainly to make it easy to convert single threaded programs to multithreaded ones
  - One example is the error code value ***errno***
- Steps
  - 1. Create a key – pthread_key_create()
  - 2. Associate global data with the key – pthread_setspecific()
  - 3. Access thread specific global data – pthread_getspecific()
  - 4. Disassociate key – pthread_key_delete()

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Specific Global Data

```
int pthread_key_create(pthread_key_t* pkey,
                       void (*destructor)(void*));
```

- ***pkey*** – pointer to where new key will be stored
- ***destructor*** – if not NULL is a pointer to a function that will be called when the thread exits with the address of the data as an argument. (So it can free any memory).
- Threads can allocate multiple keys for different data and provide respective destructors for each data.

```
int pthread_key_destroy(pthread_key_t* pkey);
```

- ***pkey*** – pointer to existing key
- This will not invoke the destructor function associated with the key.

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Specific Global Data

```
int pthread_setspecific(pthread_key_t pkey,
                        void* value);
```

- ***Returns*** – nonzero on error otherwise 0
- ***value*** – pointer to data to associate with key
- ***pkey*** – pointer to existing key

```
void* pthread_getspecific(pthread_key_t pkey);
```

- ***Returns*** – pointer to data or NULL if no data has been associated with key
- ***pkey*** – pointer to existing key

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: Thread-local storage

```
pthread_key_t global_key;

int main( void )
{
        int n;
        pthread_t thread_id[NUMTHREADS];
        ...
        pthread_key_create( &global_key, free );  //- using malloc
        ...
        pthread_create( &thread_id[n], NULL, client_thread, NULL );
        ...
}
void* client_thread( void* arg )
{
       pthread_setspecific( priority_key, malloc( sizeof( int ) ) );
        ...
        *((int*)pthread_getspecific(global_key )) = 1;
        ...
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018


Win32 Threads

Processes, Threads and Fibres

- Processes
  - Own all the hardware resources, data, code
  - Have at least one thread - can create threads
- Threads – (KLT)
  - Do all the work
  - Share resources
  - Have their own stack, copy of CPU registers
  - Can create fibres
- Fibre – (ULT)
  - Like a User-level thread – tasks that are scheduled for execution by the thread that creates the fibre
  - Not covered in this course

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Using Threads in Windows

- Three options to creating and using threads :
- **Windows API:**
  - CreateThread
  - ExitThread
  - GetExitCodeThread
  - GetCurrentThread
  - GetCurrentThreadId
  - GetThreadId
  - OpenThread
  - ResumeThread
  - SuspendThread
  - CreateRemoteThread
- Windows **C runtime library** (LIBCMT.LIB)
  - _beginthreadex()
  - _endthreadex()
- **External libraries – pthreads**, MFC, etc
  - Covered Earlier

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Using Win C Library Threads

- Manual Configuration
  - `#define _MT` in every source file before `<windows.h>`
  - `#include <process.h>`
  - Link with `LIBCMT.LIB` – Override default library *Preferred method* using Visual Studio
- From the menu bar:
  - *Build Settings* — C/C++ Tab
  - *Code Generation category*
  - Select a multithreaded run-time library

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Using Win C Library Threads

- Must include <process.h> and link to LIBCMT.LIB

unsigned long _beginthread(void(*_thrdfunc_)(void *),
unsigned _stack_size_, void *_arglist_);

unsigned long _beginthreadex(void *_security_,
unsigned _stack_size_, unsigned (* _thrdfunc_)(void *),
void *_arglist_, unsigned _initflag_, unsigned *_thrdaddr_);

  - ***Returns:*** a handle to new thread if successful otherwise ≤ 0
  - ***thrdfunc*** – pointer to function taking a void* argument and returning void that will be executed
  - ***arglist*** – if not NULL argument list to be passed to thrdfunc
  - ***Initflag*** – 0 to run the thread immediately or CREATE_SUSPEND
  - ***Thrdaddr*** – pointer to a variable that receives the thread ID
  - ***stack_size*** – the stack size for new thread, or 0 for the default
  - ***Security*** – if not NULL security descriptor for thread

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Using Win C Library Threads

- If thread was created using CREATE_SUSPEND the thread needs to be started using **`ResumeThread();`**
- You can terminate a thread explicitly but calling these functions below, if you don't they will be called automatically when the thread function ends

```
void _endthread( void );
```

//- automatically closes thread handle

```
void _endthreadex(unsigned retval);
```

//- Retval will be returned to the calling function

//- does not close thread handle, you must do it

```
BOOL CloseHandle( HANDLE hObject);
```

- See example next slide

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: _beginthread

```
#include <windows.h>
#include <process.h>
unsigned int threadFunc(void *p)
{
    printf("Inside thread...\n");
    _endthread();            //- close thread handle
}

int main()
{
    HANDLE hThread = (HANDLE)_beginthread(threadFunc, 0, NULL);
    if (hThread == NULL)    printf("Error creating thread...\n");
    else                    printf("Thread running...\n");
    return 0;
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: _beginthreadex

```
#include <windows.h>
#include <process.h>
unsigned int threadFunc(void *p)
{
        printf("Inside thread...\n");
        _endthreadex(0);            //- return '0' – handle not closed
}
int main()
{
        unsigned int threadID;
        HANDLE hThread =
        (HANDLE)_beginthreadex(NULL,0,threadFunc, 0, 0, &threadID);
        if (hThread == NULL)
        {
                printf("Error creating thread...\n");
                return -1;
        }
        printf("Thread running...\n");
        CloseHandle(hThread);       //- close thread handle
        return 0;
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

CreateThread

```
HANDLE CreateThread(
    LPSECURITY_ATTRIBUTES   lpThreadAttributes,
    DWORD                   dwStackSize,
    LPTHREAD_START_ROUTINE  lpStartAddr,
    LPVOID                  lpThreadParm,
    DWORD dwCreationFlags, LPDWORD lpThreadId);
```

- ***Returns*** a handle to the thread or NULL on failure
- ***lpThreadAttributes*** – security attributes structure - NULL
- ***dwStackSize –***
  - the stack size for new thread in bytes, or 0 for the 1M default
- ***lpStartAddr***
  - Pointer to function to be executed that takes a void pointer argument and returning an unsigned long (32 bit)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

CreateThread

```
DWORD MyThreadFunc(PVOID pThParam)
{   . . .
    ExitThread(ExitCode);  //- OR
    return ExitCode;
}
```

- *lpThreadParm*
  - The 32 bit value to be passed to the thread function
- *dwCreationFlags*
  - 0 to run the thread immediately or CREATE_SUSPEND
  - If thread was created using CREATE_SUSPEND the thread needs to be started using **ResumeThread();**
- *lpThreadId*
  - Points to a **DWORD** that receives the new thread's identifier; **NULL** OK

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Termination

- A thread can simply return with its exit code

```
VOID ExitProcess(DWORD dwExitCode);
```

- Exits a process and all its threads

```
VOID ExitThread(DWORD dwExitCode);
```

- This is the preferred technique *in C, but not C++. Also, not in a* **`__try`** *block*
  - The thread's stack is deallocated on termination

- When the last thread in a process terminates, so does the process itself

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Termination

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

- You can terminate a different thread
  - Dangerous: Thread's resources may not be deallocated. Example: handler not called.
  - Better to let the thread terminate itself
- A thread will remain in the system until the last handle to it is closed (using **`CloseHandle`**)
  - Then the thread will be deleted
- Any other thread can retrieve the exit code

```
BOOL GetExitCodeThread(
    HANDLE hThread, LPDWORD lpdwExit);
```

- Can be STILL_ACTIVE == 259

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: CreateThread – 1

```
#include <windows.h>
DWORD threadFunc(LPVOID p)
{
    printf ("Inside thread...\n");
    return 0;                    //- exitthread(0) called automatically
}                                 //- handle not closed
int main()
{
    DWORD threadID= NULL;
    HANDLE hThread =
        CreateThread(NULL,0,threadFunc,0,0,&threadID);
    if (hThread== NULL)
    {
        printMsg("Error creating thread...\n");
        return -1;
    }
    printMsg("Thread running...\n");
    CloseHandle(hThread);          //- close thread handle
    return 0;
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: GetExitCodeThread

```
DWORD myThrd(LPVOID p)
{
    Sleep(1);
    ExitThread(0);              //- return '0' – handle not closed
    return 0;
}
int main()
{
    DWORD tID, dwExit = STILL_ACTIVE;
    HANDLE hThread= CreateThread(NULL, 0, myThrd, 0, 0, &tID);
    if (hThread== NULL) return -1;
    printMsg("Thread running...\n");
    while (dwExit == STILL_ACTIVE)
    {
        GetExitCodeThread(hThread, &dwExit);
        printf("Thread still running\n");
    }
    CloseHandle(hThread);           //- close thread handle
    return 0;
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Attributes

```
HANDLE GetCurrentThread(VOID);
```

- Returns non inheritable pseudo handle to the calling thread

- `DWORD GetCurrentThreadId(VOID);`
- Obtains the current thread's ID, rather than handle

```
HANDLE OpenThread(DWORD dwDesiredAccess,
          BOOL bInheritHandle, DWORD dwThreadId);
```

- Creates a thread handle from a thread ID

```
DWORD GetProcessIdOfThread(HANDLE Thread);
```

- Finds process ID of a thread, given its handle

```
BOOL GetThreadIOPendingFlag(HANDLE hThread,
                            PBOOL lpIOIsPending);
```

- Determines if a thread has any outstanding I/O requests

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Suspending and Resuming

- Every thread has a suspend count
- A thread can execute only if this count is zero
- A thread can be created in the suspended state
- One thread can increment or decrement another's suspend count

```
DWORD SuspendThread( HANDLE hThread );
DWORD ResumeThread( HANDLE hThread );
```

- Both functions take a thread handle, return the previous suspend count
- A Return value of 0xFFFFFFFF indicates failure
- Useful in preventing "race conditions" – don't allow one thread to start until initialisation is complete

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Waiting for Termination

- Wait for a thread to terminate using the same wait functions that ared use for waiting for processes, but using thread handles.
- WaitForSingleObject(HANDLE hnd, DWORD msec)
- WaitForMultipleObjects(int count, HANDLE* hnd,…)
- The wait functions wait for the thread handle to become “signaled”
  - A thread handle is signaled when the thread terminates
  - ExitThread and TerminateThread set the object to the signaled state

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: A Simple Boss Thread

```
HANDLE hWork[K];
LONGLONG i, WorkDone[K];   . . .
for (i = 0; i < K; i++)
{
        WorkDone[i] = 0;
        hWork[i] = _beginthreadex (NULL, 0, WorkTh,
                                   (PVOID)&i, 0, NULL);
}
WaitForMultipleObjects (K, hWork, TRUE, INFINITE);
for (i = 0; i < K; i++)
        printf ("Thread %d did %d workunits\n", i, WorkDone[i]);
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example: A Simple Worker Thread

```
DWORD WINAPI WorkTh(PVOID pThNum)
{
  DWORD ThNum = (DWORD)(*pThNum);
  while (. . .)
  {
     WorkDone[ThNum]++;
  }
  _endthreadex(0);
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Priority and Scheduling

- You can control the priority of individual threads relative to the priority class of their parent process.
- You can get/set the priority of the parent process using

```
DWORD GetPriorityClass(HANDLE hProcess);

BOOL SetPriorityClass(HANDLE hProcess,
                      DWORD dwPriorityClass);
```

- ***dwPriorityClass*** – can be one of the following
  - IDLE_PRIORITY_CLASS(4)
  - NORMAL_PRIORITY_CLASS(9 or 7)
  - HIGH_PRIORITY_CLASS(13)
  - REALTIME_PRIORITY_CLASS(24)
- The Windows kernel runs the highest-priority thread that is ready for execution

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Priority and Scheduling

- You can set the priority of each thread using

```
int GetThreadPriority(HANDLE hThread);
BOOL SetThreadPriority(HANDLE hThread,
        int nPriority);
```

- ***nPriority*** – can be one of the following
  - THREAD_PRIORITY_LOWEST -2
  - THREAD_PRIORITY_BELOW_NORMAL -1
  - THREAD_PRIORITY_NORMAL +0
  - THREAD_PRIORITY_ABOVE_NORMAL +1
  - THREAD_PRIORITY_HIGHEST +2
  - THREAD_PRIORITY_IDLE (1)
  - THREAD_PRIORITY_TIME_CRITICAL (15)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Local Storage

- "Global" data within each thread
- Uses indices instead of keys,
- First need to create an index

```
DWORD TlsAlloc (VOID)
```

  - Returns the allocated *index* or -1 if no index available

```
BOOL TlsFree (DWORD dwIndex)
```

  - Deallocates the given index

```
LPVOID TlsGetValue(DWORD dwTlsIndex)
```

  - Gets the pointer to the data associated with given index

```
BOOL TlsSetValue(DWORD dwTlsIndex,
                 LPVOID lpsTlsValue )
```

  - Associates data pointed to by ***lpsTlsValue*** with given index

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Server Design

Concurrent Server Design Options

- Multiprocess:
  - One child per client
  - Preforking multiple processes
- Multithreaded:
  - One thread per client
  - Prethreaded Server (thread pool)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

One thread per client

- Instead of forking create a new thread on every new connection/message.
- Using threads makes it easier (less overhead) to have sibling processes share information.
- Sharing information must be done carefully (using semaphores/ mutexes)

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Web Server Example

- Webserver Process:

```
serverLoop()
{
    connection = AcceptCon();
    ThreadCreate(ServiceWebPage(), connection);
}
```

- Advantages of threaded version over multi-process version:
  - Can share file caches kept in memory, results of CGI scripts, other things

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Web Server Example

- Problem with unbounded Threads
  - What happens when to many requests come in at once? – throughput sinks
- Instead, allocate a bounded “thread pool”, with a maximum number of threads
  - Extra requests are simply denied

```
Server()
{
    allocThreads(slave,queue);
    while(TRUE)
    {
        con = AcceptCon();
        queue.Add(con);
        wakeUp(queue);
    }
}
```

```
slave(queue)
{
    while(TRUE)
    {
        con = queue.Get();
        if (con==null)
            sleepOn(queue);
        else  DoRequest(con);
    }
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Prethreaded Server (thread pool)

- A multithreaded server with many clients, slows down as new clients connect due to context switching overheads
- It is better to use a thread pool
  - Create a fixed number of worker threads
  - When a connection comes in hand the work over to the next available worker thread.

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Selecting A Server Design

- Depends on many factors:
  - Expected number of simultaneous clients.
  - Transaction size (time to compute or lookup the answer)
  - Fixed or variable transaction size.
  - Available system resources required to run the service.
  - How good a programmer you are.

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Thread Pools and Dispatch Queues

Boss-Worker Model

- The boss thread creates each worker thread as needed (dynamically) for each task and if necessary waits for each to finish

```
void taskX()  { … }
void taskY()  { … }
main() {     //- The Boss thread -
      while (1) {
            get a request
            switch request
            case X : pthread_create( ... taskX)
            case Y : pthread_create( ... taskY)
            …
      }
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Multi-threading Problems

- Thread creation and context switching is an expensive operation:
  - How many threads do I create?
  - How many threads is too many?
  - What's the performance impact of too many threads?
- The answer partly depends on the number of CPU cores and system load which varies greatly.
- Possible Solutions
  - A thread pool avoids the context switching problem
  - A dispatch queue solves the problem of having to manage task allocation to the thread pool
  - Placing a task on the queue requires 10's of CPU instructions verses 100's to create a thread.

© Ruben Gonzalez 1999-2010, © René Hexel 2018

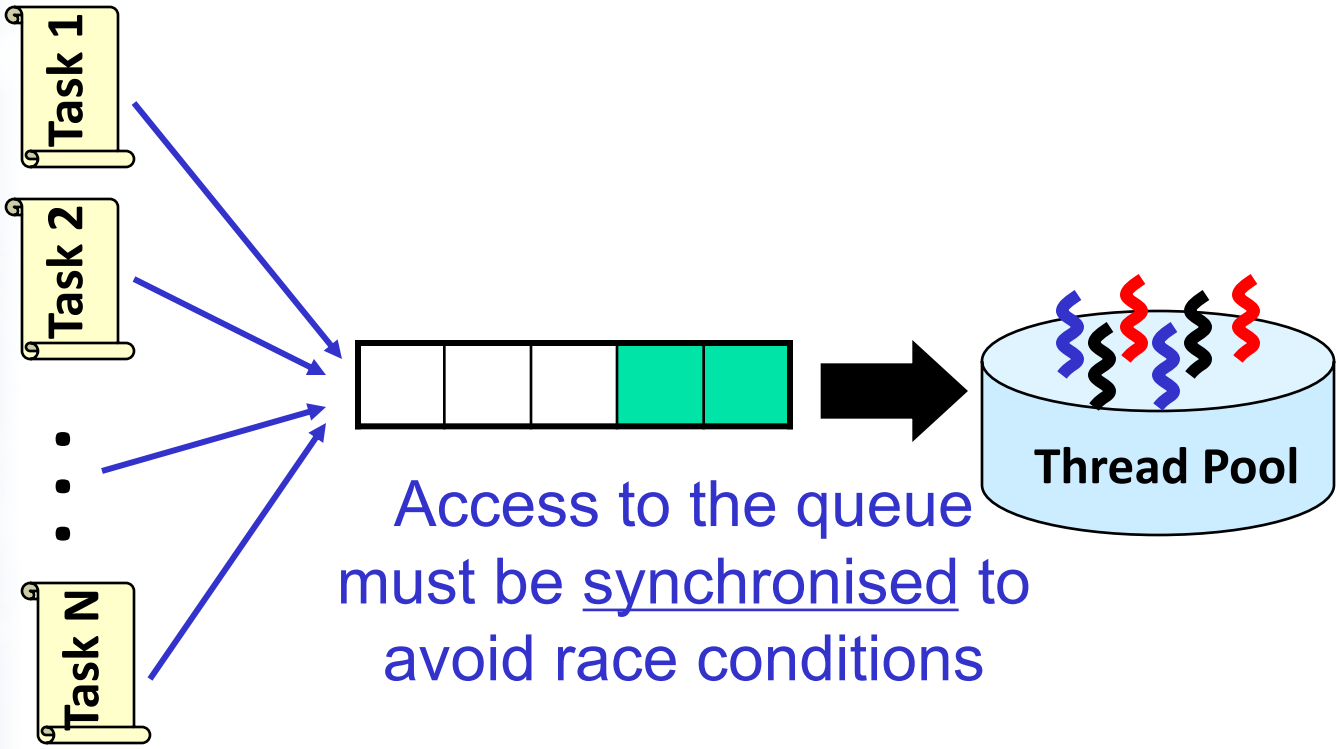
Dispatch Queue with Thread Pool

- The boss thread creates a thread pool up front with all the threads suspended
- The boss places tasks on a queue and wakes up threads to process each task as needed
- Each thread runs in an endless loop, when a thread is finished its task it just goes back to sleep

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Dispatch Queue Driven Thread Pool

- Add as many tasks as you like to the queue
- Only a set number will run concurrently



© Ruben Gonzalez 1999-2010, © René Hexel 2018

Dispatch Queue with Thread Pool

```
main() {          //- The Boss thread -
        for the number of worker threads
                pthread_create( ... pool_thread)
        while (1)  {
                get a request
                place request in work queue
                signal sleeping threads that work is available
         }
void pool_thread() { //- All worker threads -
        while (1) {
                wait until awoken by signal from boss
                take next available task off queue
                switch
                case task_X: taskX()
                case task_Y: taskY()
        }
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Example Implementation

- Apple's Grand Central Dispatch:
  - https://apple.github.io/swift-corelibs-libdispatch/
- Uses 4+ dispatch queues and 1 thread pool
  - Main queue (serial)
  - Global concurrent queues (3 set priorities)
  - Private Serial Queues (user created)
- User manually assigns tasks to queues
- Also tasks can be assigned as event handlers which will automatically be put on a queue if the event happens

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Peer (WorkCrew) Model

- This has no boss thread but one thread must create all the thread pool at the program start.
- Each thread responsible for getting its own input and often run in an infinite loop.

```
main() {
        pthread_create( ... thread1 … task1)
        pthread_create( ... thread2 … task2)
        …
        signal all threads to start
        wait for all threads to finish & clean up
}
void task1()  { wait for start; …do task… }
void task2()  {wait for start; …do task… }
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Pipeline Model

- Used for production line (stream) processing.
- Similar to Peer model but each thread reads the output of the previous thread .

```
main() {
        pthread_create( ... thread1 … task1)
        pthread_create( ... thread2 … task2)
        …
        wait for all threads to finish & clean up
}
void task1()  {
        get next input for program;
        do first task
        pass the output to next thread in the pipeline
}
void task2() {
        get output of previous thread in the pipeline;
        do second task
        pass the output to next thread in the pipeline
}
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Writing Portable Code

Comparable Thread Functions

Pthreads	Native Win
pthread_create	CreateThread, _beginthread
pthread_exit	ExitThread, _endthread
pthread_self	GetCurrentThread
pthread_yield	GetExitCodeThread
pthread_cancel	TerminateThread
pthread_join	WaitForSingleObject
pthread_key_alloc	TlsAlloc
pthread_key_delete	TLSFree
pthread_setspecific	TLSSetValue
pthread_getspecific	TLSGetValue

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Comparable Threading Functions

Pthreads	Native Win
pthread_attr_getshedpolicy, getpriority	GetPriorityClass
pthread_attr_getshedparam	GetThreadPriority
pthread_attr_setshedpolicy, setpriority, nice	SetPriorityClass
pthread_attr_setshedparam	SetThreadPriority

© Ruben Gonzalez 1999-2010, © René Hexel 2018

Emulating pthreads with Win32

```
#define THREAD_FUNCTION                 DWORD WINAPI
#define THREAD_FUNCTION_RETURN          DWORD
#define THREAD_SPECIFIC_INDEX           DWORD
#define pthread_t                       HANDLE
#define pthread_attr_t                  DWORD
#define pthread_create(thhandle,attr,thfunc,tharg)                      \
                    (int)((*thhandle=(HANDLE)_beginthreadex(NULL,0,     \
                    (THREAD_FUNCTION)thfunc,tharg,0,NULL))==NULL)
#define pthread_join(thread, result)                                    \
    ((WaitForSingleObject((thread),INFINITE)!=WAIT_OBJECT_0)            \
                    || !CloseHandle(thread))
#define pthread_detach(thread)          if(thread!=NULL) CloseHandle(thread)
#define pthread_self()                  GetCurrentThreadId()
#define thread_sleep(nms)               Sleep(nms)
#define pthread_cancel(thread)          TerminateThread(thread,0)
#define ts_key_create(ts_key, destructor)   {ts_key = TlsAlloc();};
#define pthread_getspecific(ts_key)     TlsGetValue(ts_key)
#define pthread_setspecific(ts_key, value)  TlsSetValue(ts_key, (void *)value)
```

© Ruben Gonzalez 1999-2010, © René Hexel 2018