G52OSC
OPERATING SYSTEMS AND CONCURRENCY

Processes and Threads

Dr Jason Atkin

Office Hours and Labs

- For lab questions please ask in the lab
  - If we need more time I can get lab helpers to help at other times too
- For coursework issues, we will have extra lab sessions with the lab helpers
- For course questions or other issues please see me either:
  - After the Tuesday G52CPP lecture (5pm outside LT3)
  - Or in office hours (C83), 11-12noon Wednesday
  - Or in the labs on Friday (either G52OSC or G52CPP lab)

This Lecture

- Linux : Creating a file, compiling
  - First three labs cover windows, but we will see Linux versions in lab 4, with walkthrough
- Linux create process and thread
  - fork() and pthreads
- Windows create process and thread
  - CreateProcess() and CreateThread()
  - Handles and WaitFor…
- Shared data vs separate
- Basic windows program

Processes and Threads

- An application consists of one or more processes.
- A *process*, in the simplest terms, is an executing program.
- One or more threads run in the context of the process.
- A *thread* is the basic unit to which the operating system allocates processor time.
- A thread can execute any part of the process code, including parts currently being executed by another thread.
- Source: http://msdn.microsoft.com/en-gb/library/windows/desktop/ms684841%28v=vs.85%29.aspx

Creating a process

- Processes have their own state
- Instance of a running program
  - Will have an associated program/executable
- Create a process:
  - Tell the operating system to run that program
- Special way in Unix/Linux:
  - fork() – copy the current process

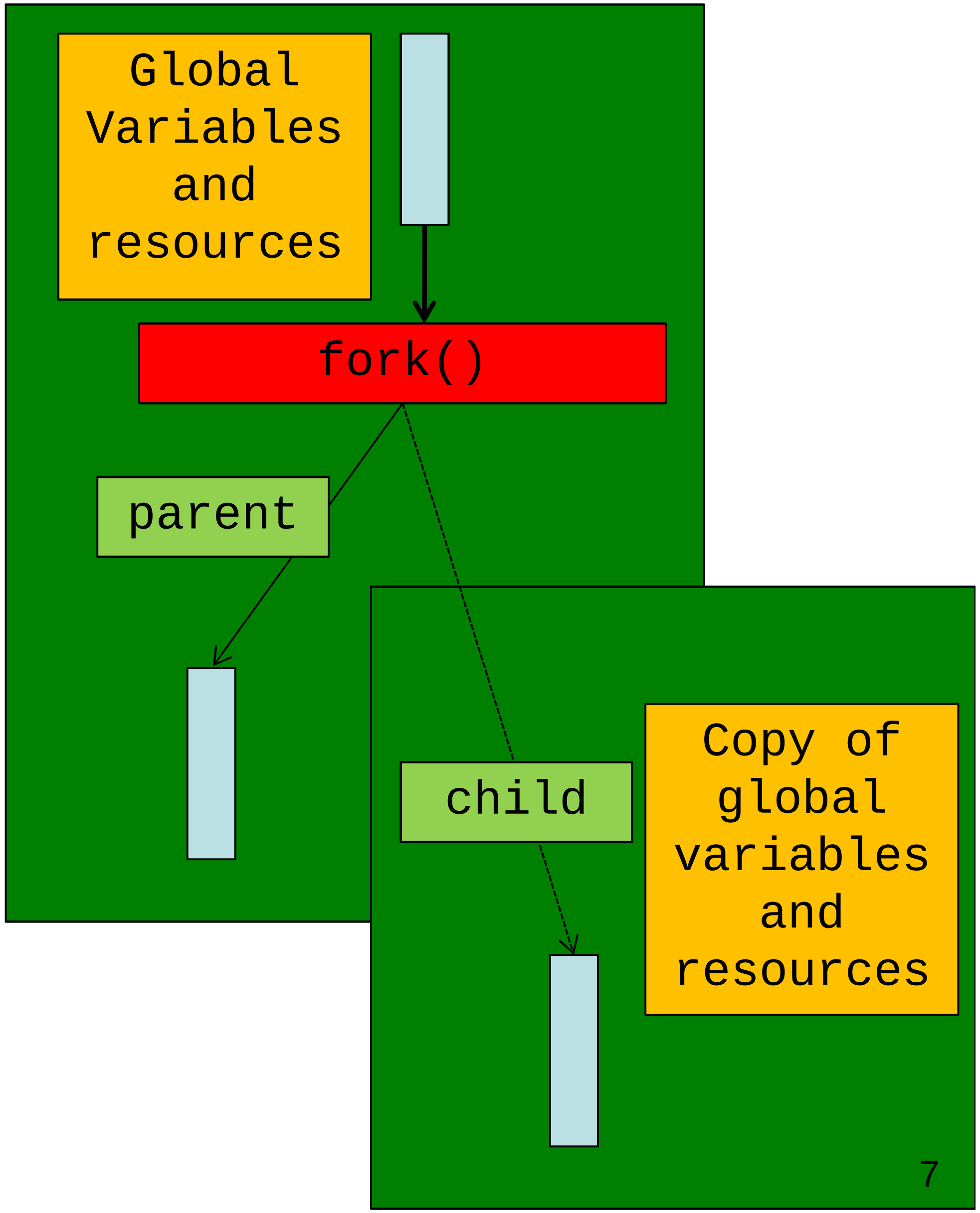
Overview: fork()

- Decide to start a new (copy) process
  - Call fork()
  - Check the return code

- Both processes will continue from the next operation in the program
  - The parent will be told the ID of the new child process as the return value
  - The child will get a 0 as the return value
  - This is the way you know which is which!

fork() – create new process

- Each process has its own resources
- You *can* share memory between both processes
  - but it is a pain
  - have to do it manually
  - often map a file (e.g. file ramdisk) into both processes
  - Next week


Global Variables and resources
fork()
parent
child
Copy of global variables and resources

Example program with fork()

```
#include <stdio.h> // printf
#include <stdlib.h> // system
#include <unistd.h> // sleep
int main()
{
   int iProcID = fork();
   if ( iProcID == 0 )
   {
      printf( "In child. Waiting 3 seconds.\n" );
      sleep( 3 );
      printf( "Child ending now.\n" );
   }
   else
   {
      printf( "In parent. Child proc id is %d.\n",iProcID );
      sleep( 1 );
      printf( "In parent: process list:\n" );
      system( "ps -f" );
      sleep( 3 );
      printf( "Parent ending now.\n" );
   }
   return 0;
}
```

Creating a process

- You get a completely new process environment
- A copy of the old one
  - Often actually a copy-on-write
- From program point of view, a copy of:
  - Global variables
  - Allocated memory
  - System resources
  - etc

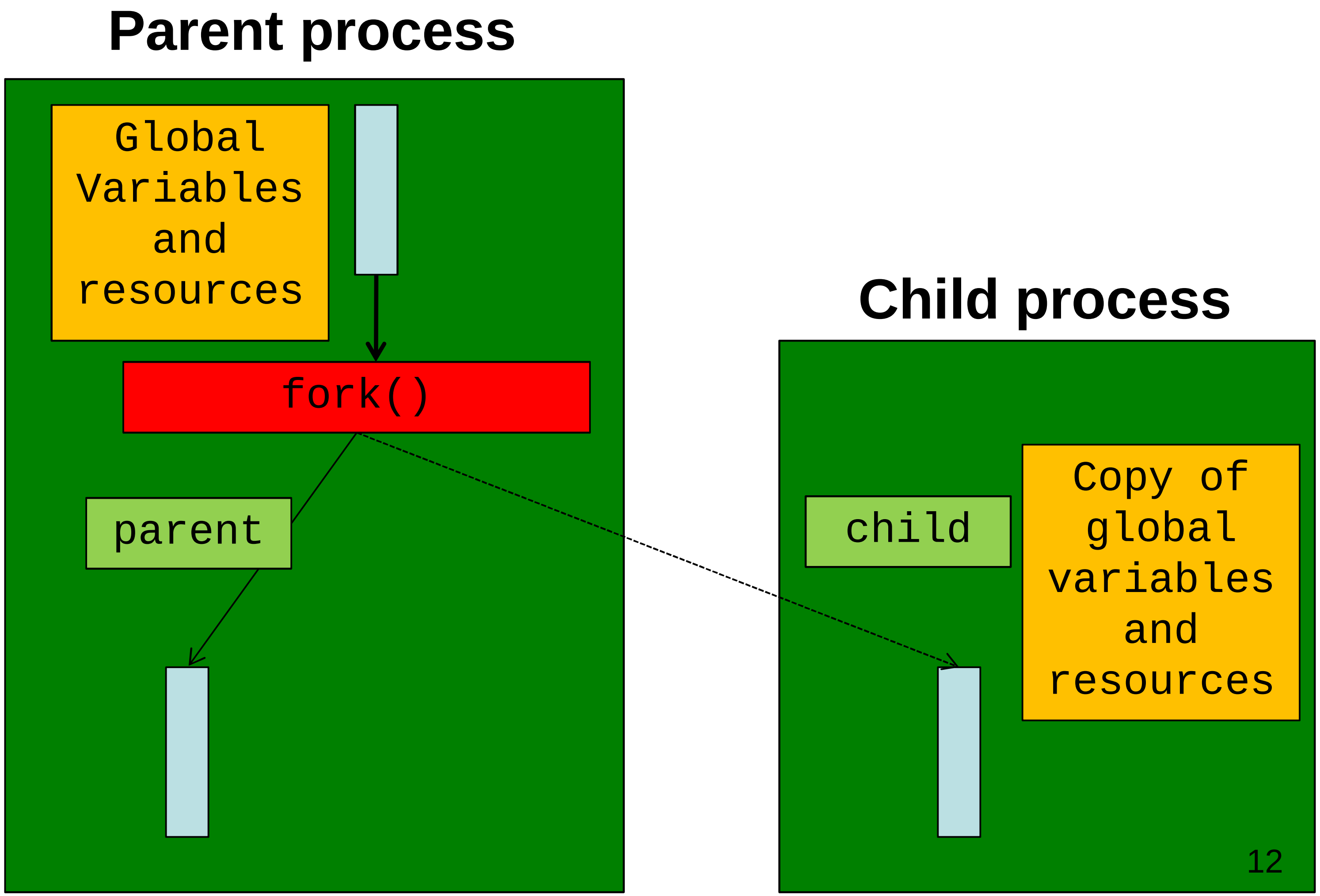
Global variable example (1)

```
#include <stdio.h>
#include <stdlib.h> // system
#include <unistd.h> // sleep

int global_variable = 1;

int main()
{
   int iProcID = fork();

   global_variable = 2;

   if ( iProcID == 0 )
   {
      global_variable = 3;
      printf(
            "In child. Waiting 3 seconds. Global variable is: %d\n",
            global_variable );
      sleep( 3 );
      printf( "Child ending now. Global variable is: %d\n",
            global_variable );
   }
```

Global variable example (2)

```
   else
   {
      global_variable = 4;

      printf( "In parent. Child proc id is %d. Wait 1 second.
Global variable is: %d\n",
           iProcID, global_variable );
      sleep( 1 );
      printf( "In parent: process list:\n" );
      system( "ps -f" );
      sleep( 3 );
      printf( "Parent ending now. Global variable is: %d\n",
           global_variable );
   }
   return 0;
}
```

Two processes

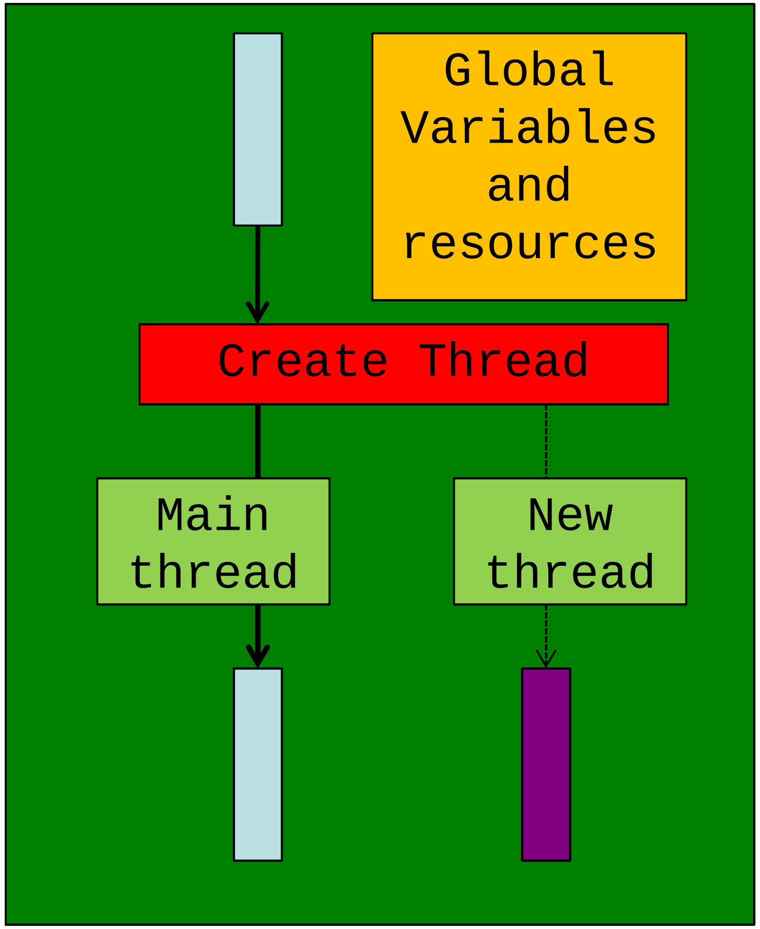


Parent process
Global Variables and resources
fork()
parent
Child process
child
Copy of global variables and resources

Threads

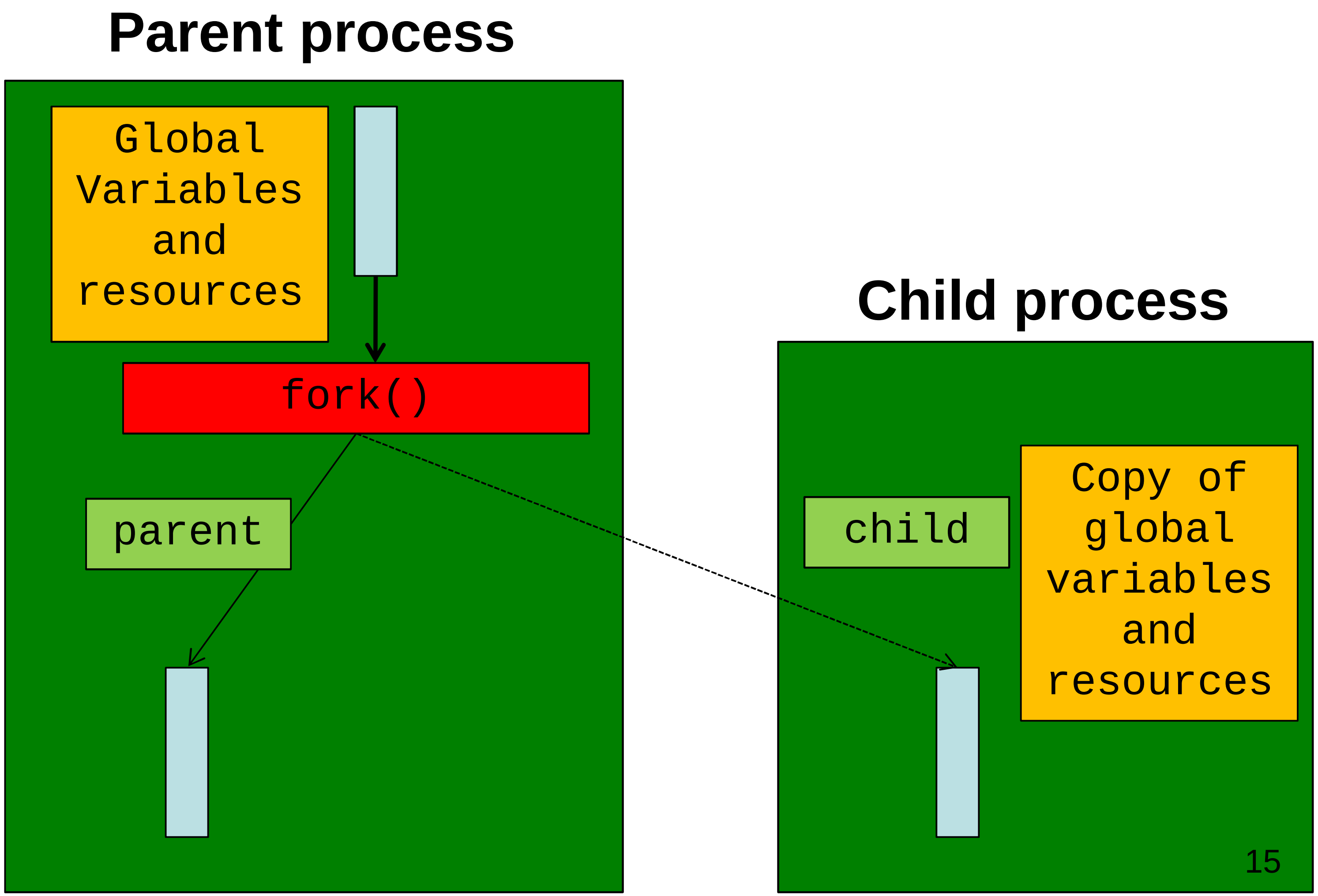
Threads – within a single process

Single process

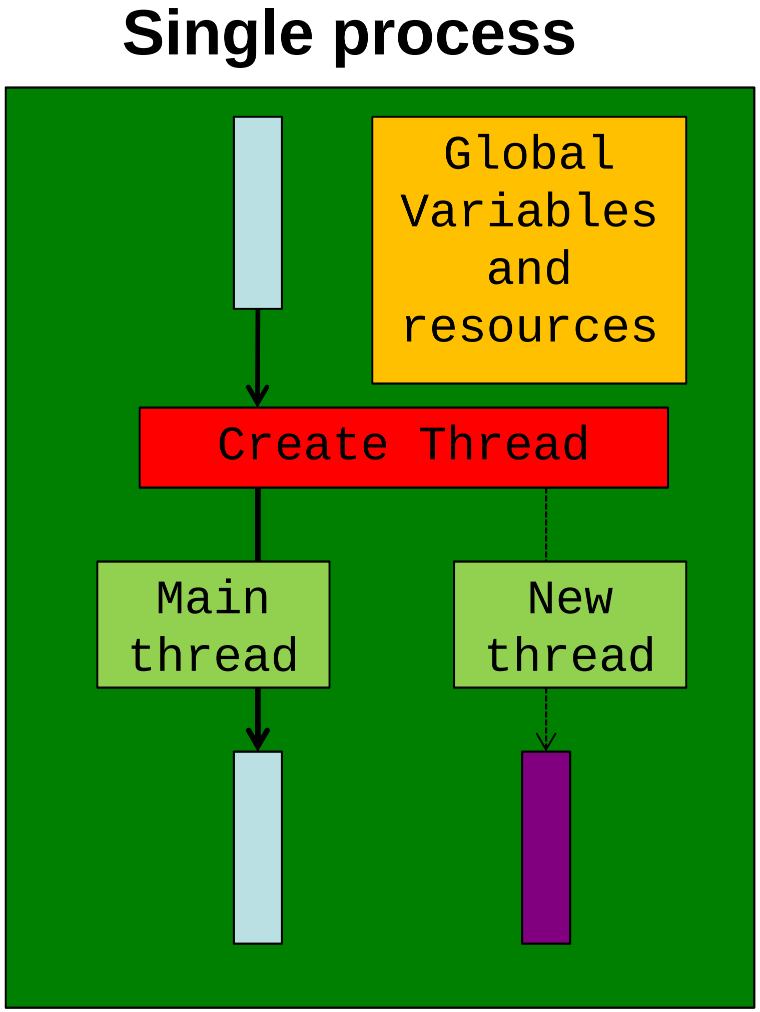


- Another execution point within the same process
- **Same** copy of global variables and resources
- Starts from a specified (new) function

Fork() : TWO processes


Parent process
Global Variables and resources
fork()
parent
Child process
child
Copy of global variables and resources

CreateThread : ONE process



The thread function and pre-amble

```
// Note: compile with 'gcc <filename> -lpthread'

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS     10

void* PrintHello(void *threadid) /*Single parameter*/
{
   long tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   sleep( 5 );
   printf( "Goodbye World! From thread #%ld!\n",tid );
   pthread_exit( NULL );
}
```

The main() function

```
int main(int argc, char *argv[])
{
  pthread_t threads[NUM_THREADS];
  int rc;
  long t;
  for( t=0; t<NUM_THREADS; t++ )
  {
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello,
             (void *)t); /* Pass thread number in */
     if (rc)
     {
        printf("ERROR; return code was %d\n", rc);
        exit(-1);
      }
   }
   /* Exit the main thread - proc ends when the others end */
   pthread_exit(NULL);
   /* Could also call pthread_join to wait for threads.*/
}
```

An example of dividing the workload using multiple concurrent operations in Windows

Call the function a number of times

```
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 2

volatile DWORD dwTotal = 0;
```

```
DWORD WINAPI thread_function(
       LPVOID lpParam )
{
   for ( int i = 0;
      i < 1000000;  i++ )
   {
      dwTotal++;
   }
   return 0;
}
```

```
int main()
{
   int iTN = 0;
   dwTotal = 0;
   for ( iTN = 0;
        iTN < NUM_THREADS;
        ++iTN )
   {
     thread_function( … );
   }
   printf("Total %d\n",dwTotal);

   printf( "Press RETURN" );
   while ( getchar() != '\n' )
     ;
   return 0;
}
```

Windows types

- To try to promote portability, windows has a lot of custom types
  - Prevents sizes changing between platforms
- There are standard naming conventions for these
- LP : long pointer – basically means pointer to
- WORD : 2 byte value, unsigned
- DWORD : double word, 4 byte value, unsigned
- SIZE_T : common type for storing sizes, e.g. number of bytes
- LPVOID : Long pointer to a type that you don’t care about – usually then cast to appropriate type

Reminders: windows things

```
volatile DWORD dwTotal = 0;
```

- Use volatile (C specifier) when the variable may be accessed from multiple threads/processes
  - Much more about this later
- DWORD specifies a size of 4 bytes, and is a name for the 'unsigned long' type here

```
DWORD WINAPI thread_function( LPVOID
lParam )
```

- WINAPI is a flag to tell the compiler the format to lay out the function
- LPVOID means void* (long pointer to void)

The main part which calls function

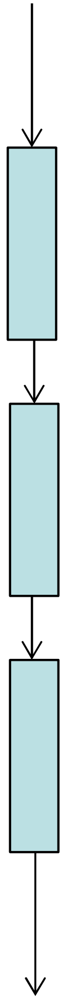
```
int main()
{
   int iTN = 0;
   dwTotal = 0;
   for ( iTN = 0;
        iTN < NUM_THREADS;
        ++iTN )
   {
      thread_function( … );
   }
   printf("Total %d\n",dwTotal);

   printf( "Press RETURN" );
   while ( getchar() != '\n' )
      ;
    return 0;
}
```

- Loops **NUM_THREADS** times, calling the **thread_function** manually each time
- We could tell it to do these simultaneously

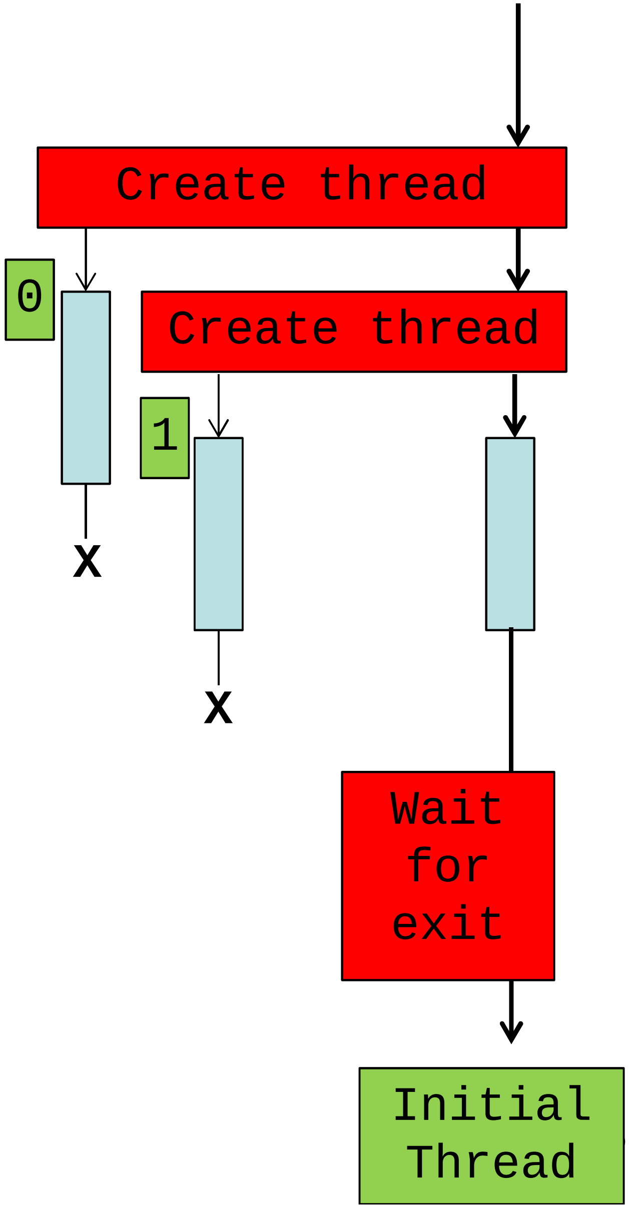
Thread creation

- You create one new thread at a time
- There is an overhead in creating threads
  - Function **`CreateThread()`** takes time
  - Operating system has to allocate the resources for the thread
- You have no idea what order the threads will execute, where any interleaving happens, etc
  - Parts of one may be executed, then parts of another, etc. You don't even know which will finish first
- Your original thread can do some of the work itself (avoiding one creation)
- You may need to wait for things to finish

Parallelising the code (3 times)



- We could make these simultaneous
- By creating multiple threads
  - Create them
  - Run one function call in each
  - Wait for them to exit



Original program, ready for threads

```c
#define WIN32_LEAN_AND_MEAN
#include <Windows.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 2

volatile DWORD dwTotal = 0;

DWORD WINAPI thread_function(
       LPVOID lpParam )
{
   for ( int i = 0;
      i < 1000000;  i++ )
   {
      dwTotal++;
   }
   return 0;
}
```

```c
int main()
{
   int iTN = 0;
   dwTotal = 0;
   for ( iTN = 0;
         iTN < NUM_THREADS;
         ++iTN )
   {
      thread_function( … );
   }
   printf("Total %d\n",dwTotal);

   printf( "Press RETURN" );
   while ( getchar() != '\n' )
      ;
    return 0;
}
```

This part remains unchanged

```
#define NUM_THREADS 2

// Windows name for an unsigned long - 4 bytes
volatile DWORD dwTotal = 0;

// Function called by every thread
DWORD WINAPI thread_function( LPVOID lParam )
{
      for ( int i = 0; i < 1000000; i++ )
      {
            dwTotal++;
      }
      return 0;
}
```

Threaded version of main()

```
HANDLE arrdwThreadHandles[NUM_THREADS];

for ( iTN = 0; iTN < NUM_THREADS - 1; ++iTN )
    arrdwThreadHandles[iTN] = CreateThread(
            NULL, /* No security change */
            0, /* Default stack size */
            thread_function, /* Name of function to call */
            (LPVOID)iTN, /* parameter you can give */
            0,/* Extra flags */
            NULL /* You can get the thread id if you wish */
        );
/* Do the last one in the current thread */
thread_function( (LPVOID)(NUM_THREADS - 1) );

WaitForMultipleObjects( NUM_THREADS - 1,
    arrdwThreadHandles, /* Array of handles */
    TRUE, 10000 ); /* Wait for all, for up to 10 secs */
```

CreateThread

- http://msdn.microsoft.com/en-us/library/windows/desktop/ms682453%28v=vs.85%29.aspx

```
HANDLE WINAPI CreateThread(
    LPSECURITY_ATTRIBUTES   lpThreadAttributes,
    SIZE_T                  dwStackSize,
    LPTHREAD_START_ROUTINE  lpStartAddress,
    LPVOID                  lpParameter,
    DWORD                   dwCreationFlags,
    LPDWORD                 lpThreadId );
```

- When you see LPSECURITY_ATTRIBUTES, just pass NULL to say to give it the permissions of the current process
- Stack size says how much memory it needs for functions and local variables (give it a lot if you use recursion)
- LPTHREAD_START_ROUTINE asks for the function to run in the thread. If must be labelled “**WINAPI**” since you are not calling it

Windows Handles

- A **`Handle`** gives you something to use to ‘grip’ a windows object that the system owns
- Processes, threads, windows, events, mutexes etc all have handles that you can use
- You can’t do anything with these apart from use them to refer to the object
- CreateThread returns a Handle for the thread that was created
- We store all of these thread handles in an array in case we need them later

```
    arrdwThreadHandles[iTN] =
                    CreateThread( … );
```

Waiting for handles

- A thread handle can be used to determine whether a thread has finished or not
- You can wait for handles
  - Waiting for an event to go off
  - For a resource to be free
  - Or for a thread to finish
- Waiting for multiple handles to all be ready can be better than waiting for them one at a time

WaitFor...

- Wait for one or all of multiple handles to be ‘ready’

```
WaitForMultipleObjects(
     NUMBER_THREADS - 1, /*Count*/
     arrdwThreadHandles, /*Array*/
     TRUE,               /*All vs one?*/
     10000 );            /*Timeout ms*/
```

```
DWORD WINAPI WaitForMultipleObjects(
     DWORD nCount,
     const HANDLE *lpHandles,
     BOOL bWaitAll,
     DWORD dwMilliseconds );
```

Next Lecture

- Windows GUI programs
  - Registering window classes
  - Creating windows
  - Message loops

- Note that both X (Unix/Linux) and Java use similar methods