

# Threads

Lecture on creating and joining  
threads

# What is a thread?

- A **thread** is a sequence of instructions that can be executed in parallel with other threads
- Threads of a program are not full-blown processes, but are smaller portions of the process running concurrently (or in parallel). Hence, the term **lightweight** is used

# Needles and Threads

- A thread is a fibre used in sewing with a needle
- Think of this
- Think of sewing needles as the CPUs and the threads in a program as the fibre. If you had two needles but only one thread, it would take longer to finish the job than if you split the thread into two and used both needles at the same time.
- Taking this analogy a little further, if one needle had to sew on a button (blocking I/O), the other needle could continue doing other useful work even if the other needle took 4 hours to sew on a single button. If you only used one needle, you would be ~4 hours behind
- We will see the term LWP or lightweight process being used

# Threads and Processes

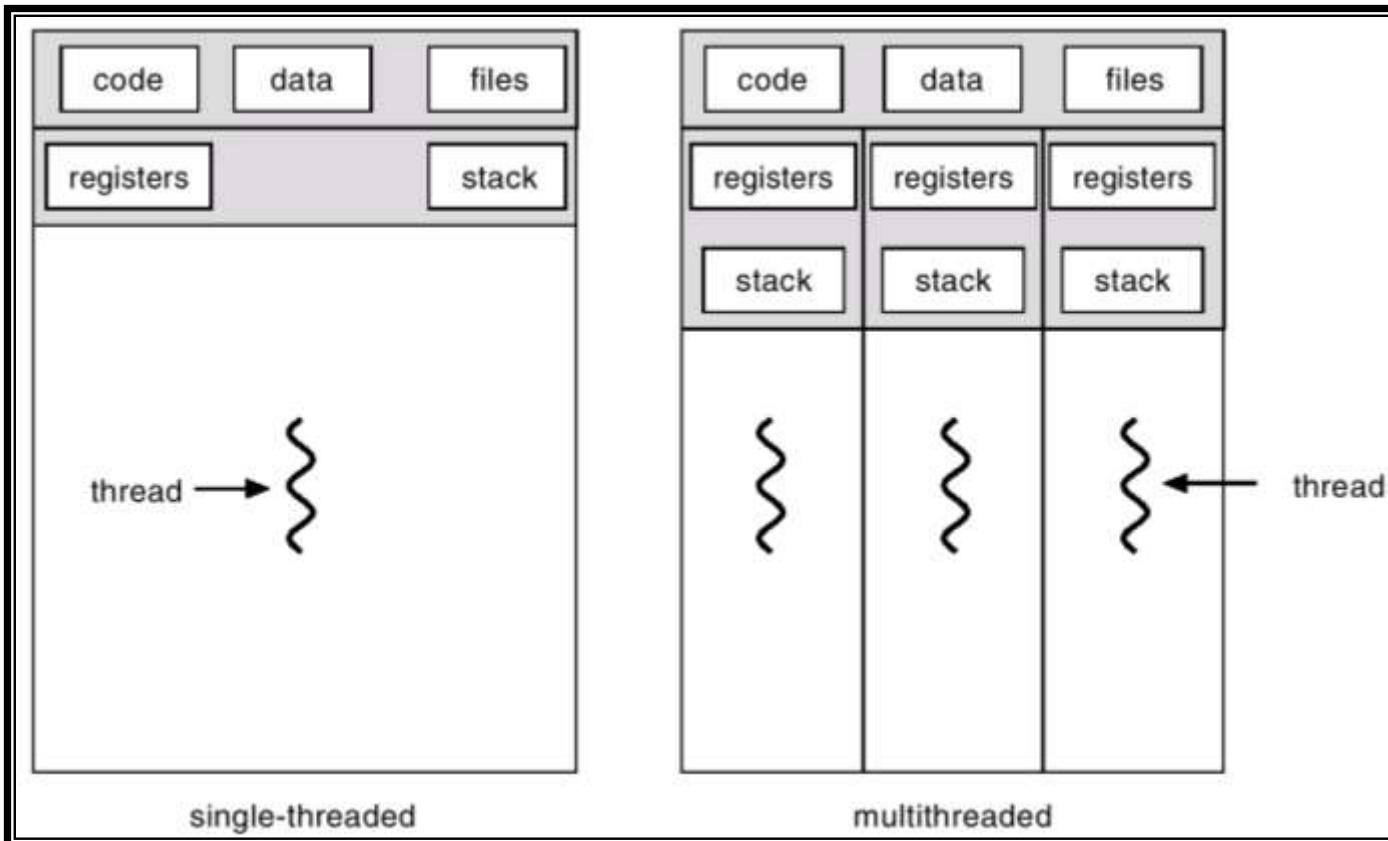
- In traditional single-threaded process systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads.

# Single and Multi-threading

- Single threading
  - This is a process with only one thread of control so is essentially a single process or one processing one command at a time.
- Multithreading
  - Allows applications to manage a separate process with several threads of control; This allows the OS to emulate a system with multiple processors and a common memory.
  - It reduces overheads associated with swapping the whole process in and out of memory.
  - *In other words different parts of the program can be run concurrently in a single processor system or in parallel in a multi-processor system.*

# THREADS

## Single and Multithreaded Processes



4: Threads

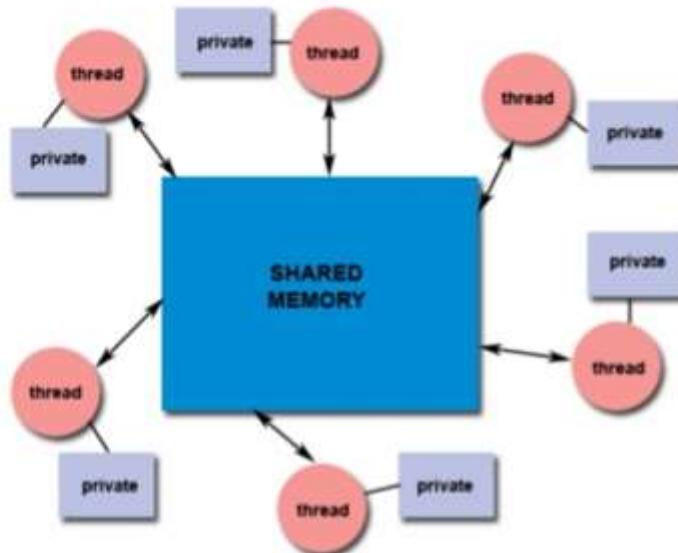
# Examples of multithreading

- Examples of multithreading:
  - Multiple queries to a database (a separate thread for each query);
  - Multiple connections TCP connections: a separate thread to deal with each connection.
  - An application that has a processing element and an i/o element where one thread processing and one thread to deal with the I/O
  - A word processor will have many threads; one for inputting data, one for backup or archiving the word document...

# Property of multi-threading

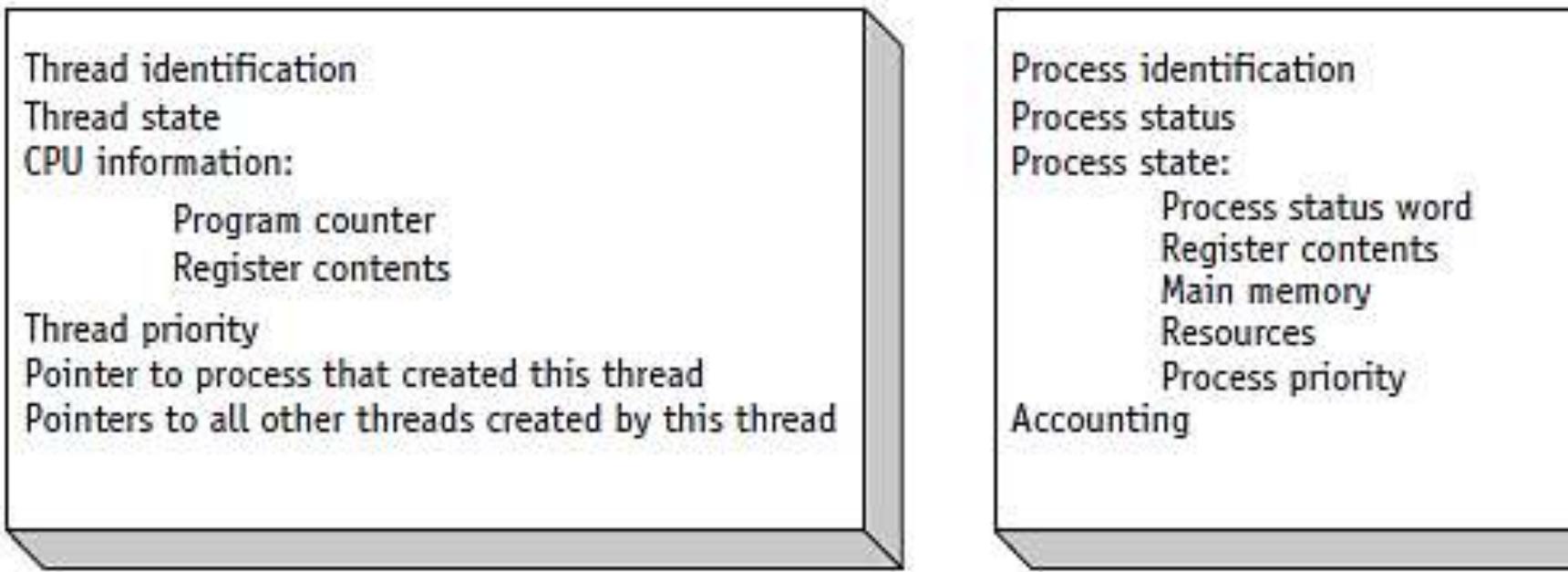
## Shared Memory Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



# Thread properties

- A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:
  - Stack
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Some thread-specific data.
- Threads within a process must not be considered as a group of processes. All threads share the same address space. This means that ***two pointers variables having the same value*** in two threads ***refer to the same data***. Also, if any thread changes one of the shared system resources, all threads within the process are affected; e.g. *Closing files... shared address...*



**(figure 4.4)**

Comparison of a typical Thread Control Block (TCB) vs. a Process Control Block (PCB).

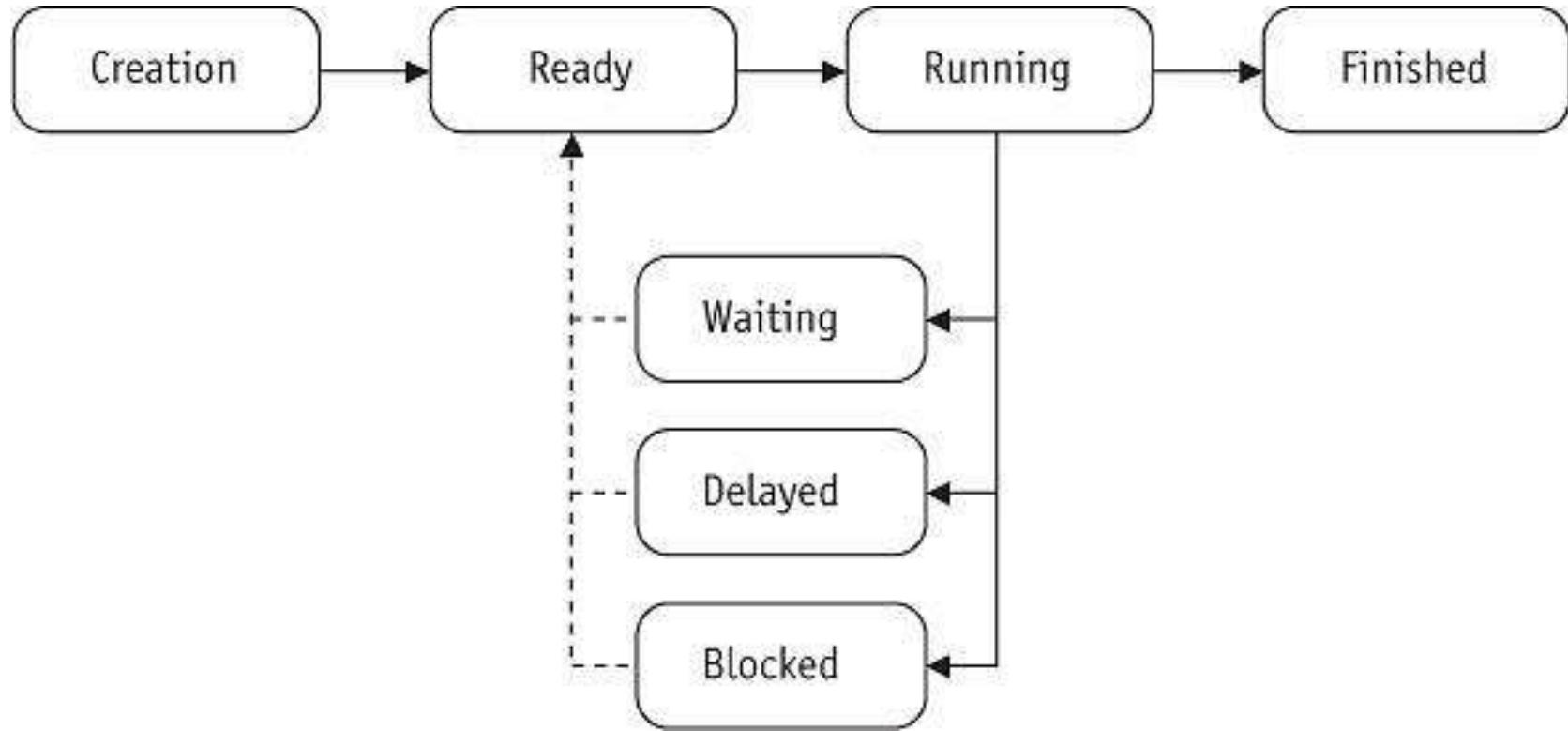
© Cengage Learning 2014

# Thread Creation and Implementation

- **The Initial Thread**
- When a process is created, one thread is automatically created. This thread is called the *initial thread*. It ensures the compatibility between the old processes with a unique implicit thread and the new multi-threaded processes. The initial thread has some special properties, not visible to the programmer, that ensure binary compatibility between the old single-threaded programs and the multi-threaded operating system. It is also the initial thread that executes the **main** routine in multi-threaded programs.
- **Threads Implementation**
- A thread is the schedulable entity, which means that the system scheduler handles threads. These threads, known by the system scheduler, are strongly implementation-dependent. To facilitate the writing of portable programs, libraries provide another kind of thread.

# Thread States

- Operating systems must be able to:
  - **Create** new threads
  - Set up a thread so it is ready to execute
  - **Delay**, or put to sleep, threads for a specified amount of time
  - **Block**, or suspend, threads waiting for I/O to be completed
  - Set threads to a **WAIT** state until a specific event occurs: until another thread finishes execution
  - **Schedule** threads for execution
  - **Terminate** a thread and release its resources



**(figure 4.3)**

A typical thread changes states from READY to FINISHED as it moves through the system.

© Cengage Learning 2014

# Thread States (cont'd.)

- Thread transitions
  - Application creates a thread: placed in READY queue
  - READY to RUNNING: *Process Scheduler* assigns it to a processor
  - RUNNING to WAITING: when dependent on an outside event, e.g., mouse click, or waiting for another thread to finish
  - WAITING to READY: outside event occurs or previous thread finishes

# Thread States (cont'd.)

- Thread transitions (cont'd.)
  - RUNNING to DELAYED: application that delays thread processing by specified amount of time
  - DELAYED to READY: prescribed time elapsed
  - RUNNING to BLOCKED: I/O request issued
  - BLOCKED to RUNNING: I/O completed
  - RUNNING to FINISHED: exit or termination
    - All resources released

# Programming Preliminaries

- Include `pthread.h` in the main file
- Compile program with `-lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise  
but calls will fail
- Good idea to check return values on  
common functions

# Creating Threads:

- Initially, your main() program comprises a single, default thread. All other threads must be explicitly created by the programmer.
- `pthread_create` creates a new thread and makes it executable. This routine can be called any number of times from anywhere within your code.

# Thread creation

- Types: `pthread_t` – data type (thread)
- Some calls:

```
int pthread_create(pthread_t *thread,
                  pthread_attr_t *attr,
                  *start_routine,
                  void *arg);

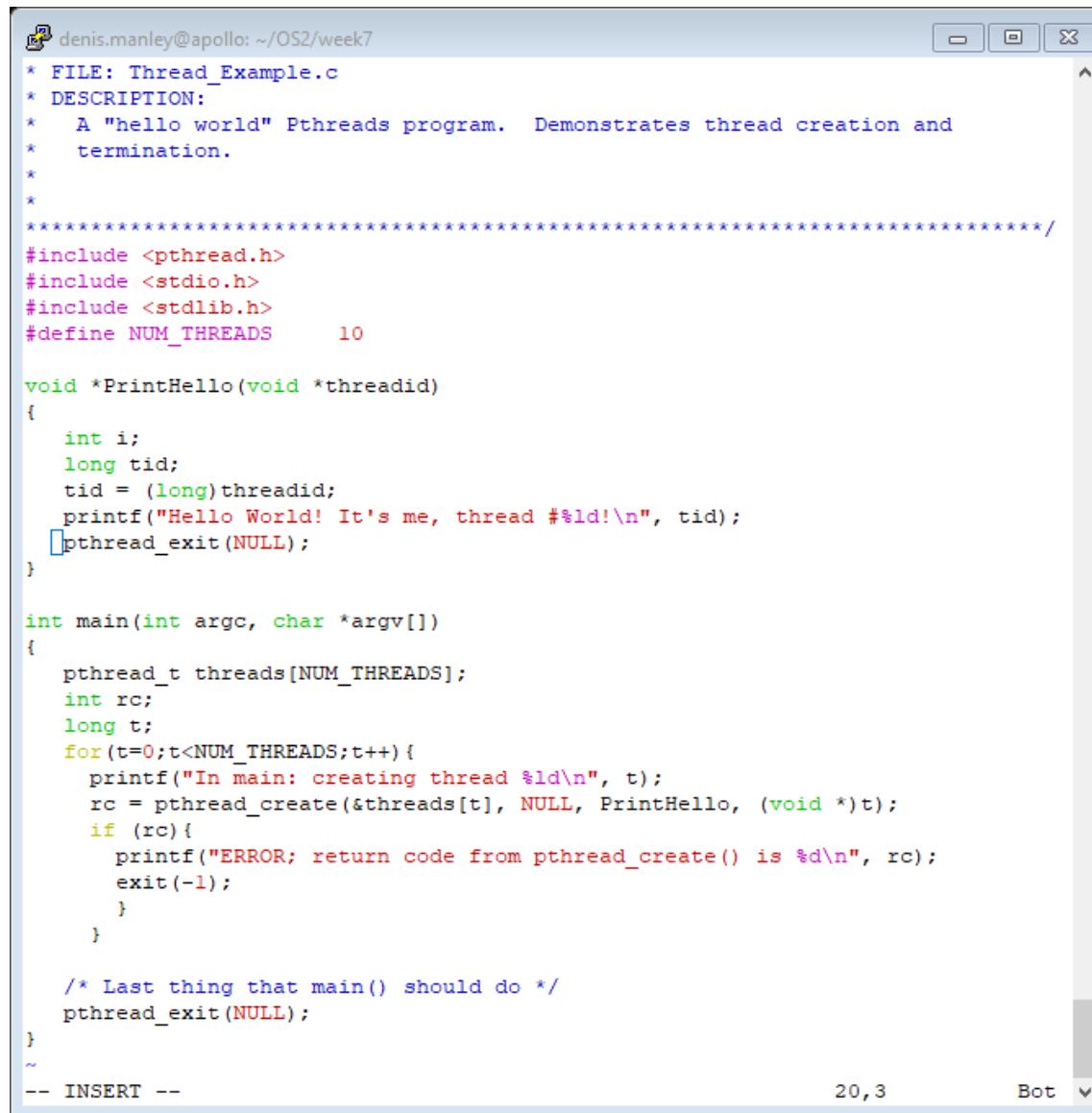
void pthread_exit();
```

- No explicit parent/child model, except main thread holds process info
- Call `pthread_exit` in main, don't just fall through;

# pthread\_create arguments:

- **\*thread**: A unique identifier (pointer) for the new thread returned by the subroutine.
- **attr**: An attribute object that may be used to set thread attributes (joinable, detach, stack size...). You can specify a thread attributes object, or NULL for the default values. (more detail here [thread\\_attributes](#))
- **\*start\_routine**: the C routine that the thread will execute once it is created.
- **arg**: A single argument that may be passed to *start\_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed. Be aware that this single argument can also be an array of arguments (but this is still only a pointer)

# Create Basic Thread example



```
denis.manley@apollo: ~/OS2/week7
* FILE: Thread_Example.c
* DESCRIPTION:
*   A "hello world" Pthreads program. Demonstrates thread creation and
*   termination.
*
*****
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS      10

void *PrintHello(void *threadid)
{
    int i;
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #%-ld!\n", tid);
    pthread_exit(NULL);
}

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);
        if (rc) {
            printf("ERROR: return code from pthread_create() is %d\n", rc);
            exit(-1);
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
~
-- INSERT --
```

# Thread example 1 with pass int

```
denis.manley@apollo: ~/OS2/week7
-rwxr-xr-x 1 denis.manley domain^users 8815 Oct 17 15:15 thread_unsafe
-rw-r--r-- 1 denis.manley domain^users 971 Oct 15 15:49 thread_unsafe.c
denis.manley@apollo:~/OS2/week7$ ./thread_example1
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
In main: creating thread 4
In main: creating thread 5
In main: creating thread 6
In main: creating thread 7
In main: creating thread 8
In main: creating thread 9
Hello World! It's me, thread #6!
Hello World! It's me, thread #7!
Hello World! It's me, thread #5!
Hello World! It's me, thread #4!
Hello World! It's me, thread #8!
Hello World! It's me, thread #9!
Hello World! It's me, thread #3!
Hello World! It's me, thread #2!
Hello World! It's me, thread #1!
Hello World! It's me, thread #0!
denis.manley@apollo:~/OS2/week7$ 
```

# Modify Scheduling output

The image shows two terminal windows side-by-side. The left terminal window displays the source code for a C program named `Thread_Example.c`. The code creates multiple threads and prints their IDs. A red arrow points from the text "Delay creation of threads" at the bottom to the line `rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);`. The right terminal window shows the program's output, which consists of many lines of "Hello World! It's me, thread #X!" where X is a number from 0 to 19. This indicates that all threads were created simultaneously.

```
denis.manley@apollo:~/OS2/week7$ cat Thread_Example.c
/*
 * FILE: Thread_Example.c
 * DESCRIPTION:
 *   A "hello world" Pthreads program. Demonstrates thread creation and
 *   termination.
 *   Modify speed of creation via command line argument.
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 20

void *PrintHello(void *threadid)
{
    int i;
    long tid;
    tid = (long)threadid;
    printf("Hello World! It's me, thread #ld!\n", tid);
    pthread_exit(NULL);
}

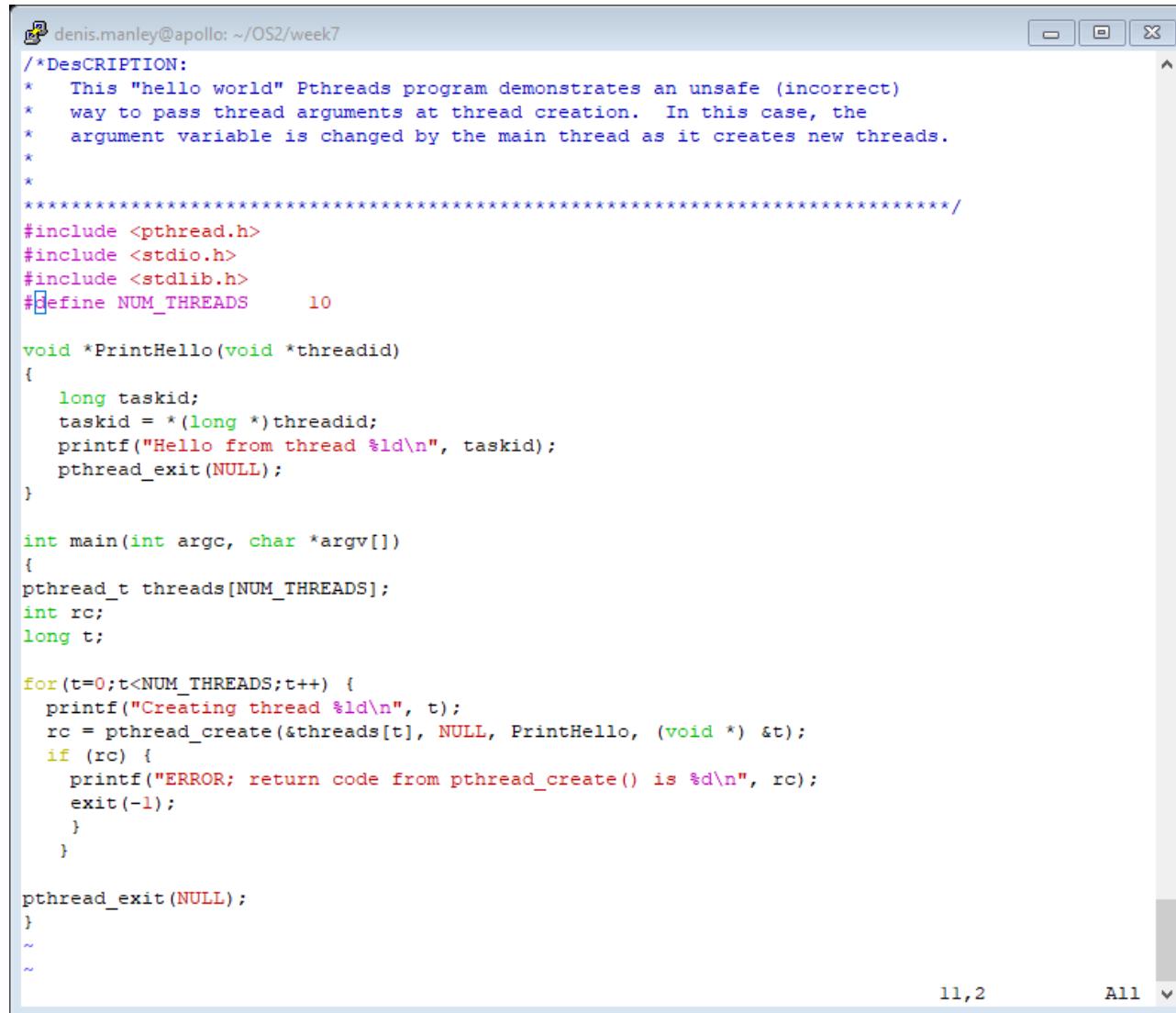
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc, i, Max = atoi(argv[1]);
    long t;

    for(t=0;t<NUM_THREADS;t++){
        printf("In main: creating thread %ld\n", t);
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
        for (i=0;i<Max;i++){
            if (rc){
                printf("ERROR: return code from pthread_create() is %d\n", rc);
                exit(-1);
            }
        }
    }

    /* Last thing that main() should do */
    pthread_exit(NULL);
}
denis.manley@apollo:~/OS2/week7$ ./Thread_Example 1000000
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 4
In main: creating thread 5
Hello World! It's me, thread #3!
In main: creating thread 6
Hello World! It's me, thread #4!
In main: creating thread 7
Hello World! It's me, thread #5!
Hello World! It's me, thread #6!
In main: creating thread 8
Hello World! It's me, thread #7!
In main: creating thread 9
In main: creating thread 10
Hello World! It's me, thread #8!
In main: creating thread 11
Hello World! It's me, thread #9!
In main: creating thread 12
Hello World! It's me, thread #10!
In main: creating thread 13
Hello World! It's me, thread #11!
Hello World! It's me, thread #12!
In main: creating thread 14
In main: creating thread 15
Hello World! It's me, thread #13!
Hello World! It's me, thread #14!
In main: creating thread 16
In main: creating thread 17
Hello World! It's me, thread #15!
In main: creating thread 18
Hello World! It's me, thread #16!
Hello World! It's me, thread #17!
In main: creating thread 19
Hello World! It's me, thread #18!
Hello World! It's me, thread #19!
denis.manley@apollo:~/OS2/week7$
```

Delay creation of threads

# Code to demonstrate shared memory



The screenshot shows a terminal window titled "denis.manley@apollo: ~/OS2/week7". The window contains a C program demonstrating shared memory issues. The code includes a multi-line comment at the top, imports for pthreads, stdio, and stdlib, and defines NUM\_THREADS as 10. It features a PrintHello function that prints its thread ID and exits. The main function creates 10 threads, each printing its ID and exiting. The terminal shows the output of the program, which exhibits undefined behavior due to shared memory issues.

```
/*DesCRIPTION:  
 * This "hello world" Pthreads program demonstrates an unsafe (incorrect)  
 * way to pass thread arguments at thread creation. In this case, the  
 * argument variable is changed by the main thread as it creates new threads.  
 *  
 ****  
#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
#define NUM_THREADS      10  
  
void *PrintHello(void *threadid)  
{  
    long taskid;  
    taskid = *(long *)threadid;  
    printf("Hello from thread %ld\n", taskid);  
    pthread_exit(NULL);  
}  
  
int main(int argc, char *argv[])  
{  
pthread_t threads[NUM_THREADS];  
int rc;  
long t;  
  
for(t=0;t<NUM_THREADS;t++) {  
    printf("Creating thread %ld\n", t);  
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);  
    if (rc) {  
        printf("ERROR; return code from pthread_create() is %d\n", rc);  
        exit(-1);  
    }  
}  
  
pthread_exit(NULL);  
}  
~  
~
```

11,2      All

Thread\_1 unsafe pass pointer &t and not t

# Modify to delay thread creation

- Modify the unsafe code to delay the creation of threads: the delay time is input via command line: hint refer to modified safe code.
- 
- Remove the delay code from within the thread.
- Refer to the following slide for code

# Modified thread\_unsafe.c

```
dennis@manley:~/src/apollor - [OS2] ~ %  
/*DESCRIPTION:  
 * This "hello world" Pthreads program demonstrates an unsafe (incorrect)  
 * way to pass thread arguments at thread creation. In this case, the  
 * argument variable is changed by the main thread as it creates new threads.  
 * AUTHOR: Blaise Barney  
 * LAST REVISED: 07/16/14  
*****  
#include <pthread.h>  
#include <stdio.h>  
#include <stdlib.h>  
#include<unistd.h>  
#define NUM_THREADS      10  
  
void *PrintHello(void *threadid)  
{  
    long taskid;  
    int i;  
    // sleep(1);  
    // for(i=0; i<10000; i++);  
    taskid = *(long *)threadid;      //dereference pointer threadid  
    printf("Hello from thread %ld\n", taskid);  
    pthread_exit(NULL);  
}  
  
int main(int argc, char *argv[]){  
    pthread_t threads[NUM_THREADS];  
    int rc;  
    long t;  
  
    for(t=0;t<NUM_THREADS;t++) {  
        printf("Creating thread %d\n", t);  
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &t);  
  
        // MAX = argv[1];  
        // for(i=0;i<MAX; i++);  
        // for(i=0; i<10000; i++);  
        if (rc) {  
            printf("ERROR: return code from pthread_create() is %d\n", rc);  
            exit(-1);  
        }  
    }  
  
    pthread_exit(NULL);  
}
```

Slow processing

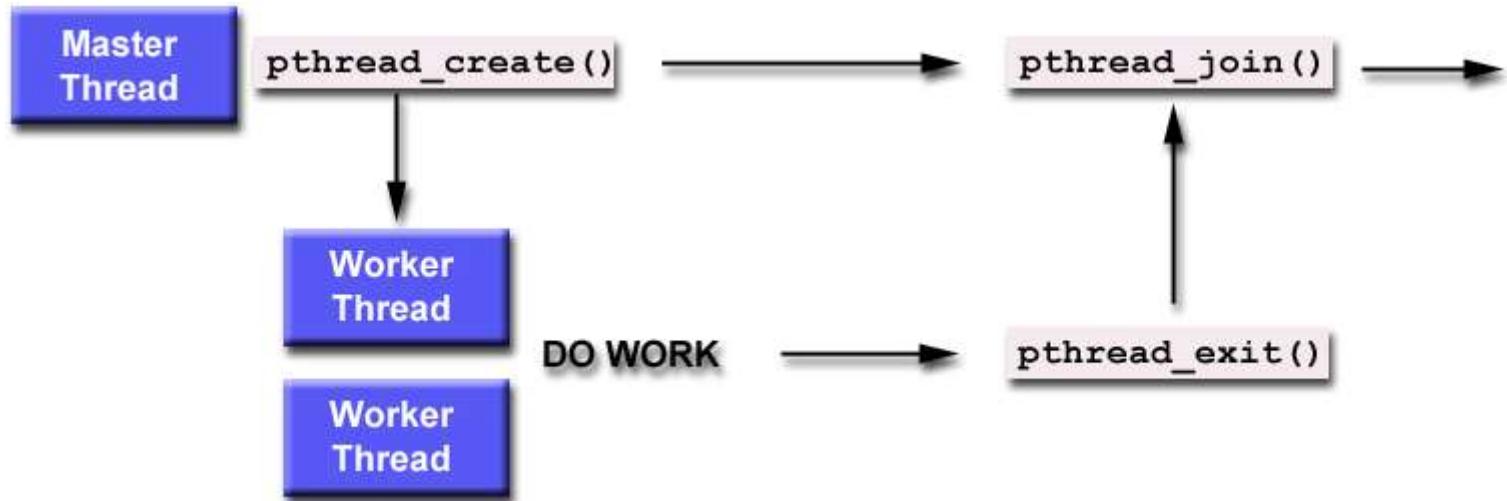
# Modify Thread\_1 unsafe .c to delay thread creation

```
denis.manley@soc-apollo:~/OS2/week7$ ./thread_unsafe
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Hello from thread 3
Creating thread 4
Hello from thread 4
Creating thread 5
Hello from thread 5
Hello from thread 6
Creating thread 6
Hello from thread 6
Hello from thread 6
Creating thread 7
Creating thread 8
Creating thread 9
Hello from thread 9
Hello from thread 10
Hello from thread 10
Hello from thread 10
denis.manley@soc-apollo:~/OS2/week7$
```

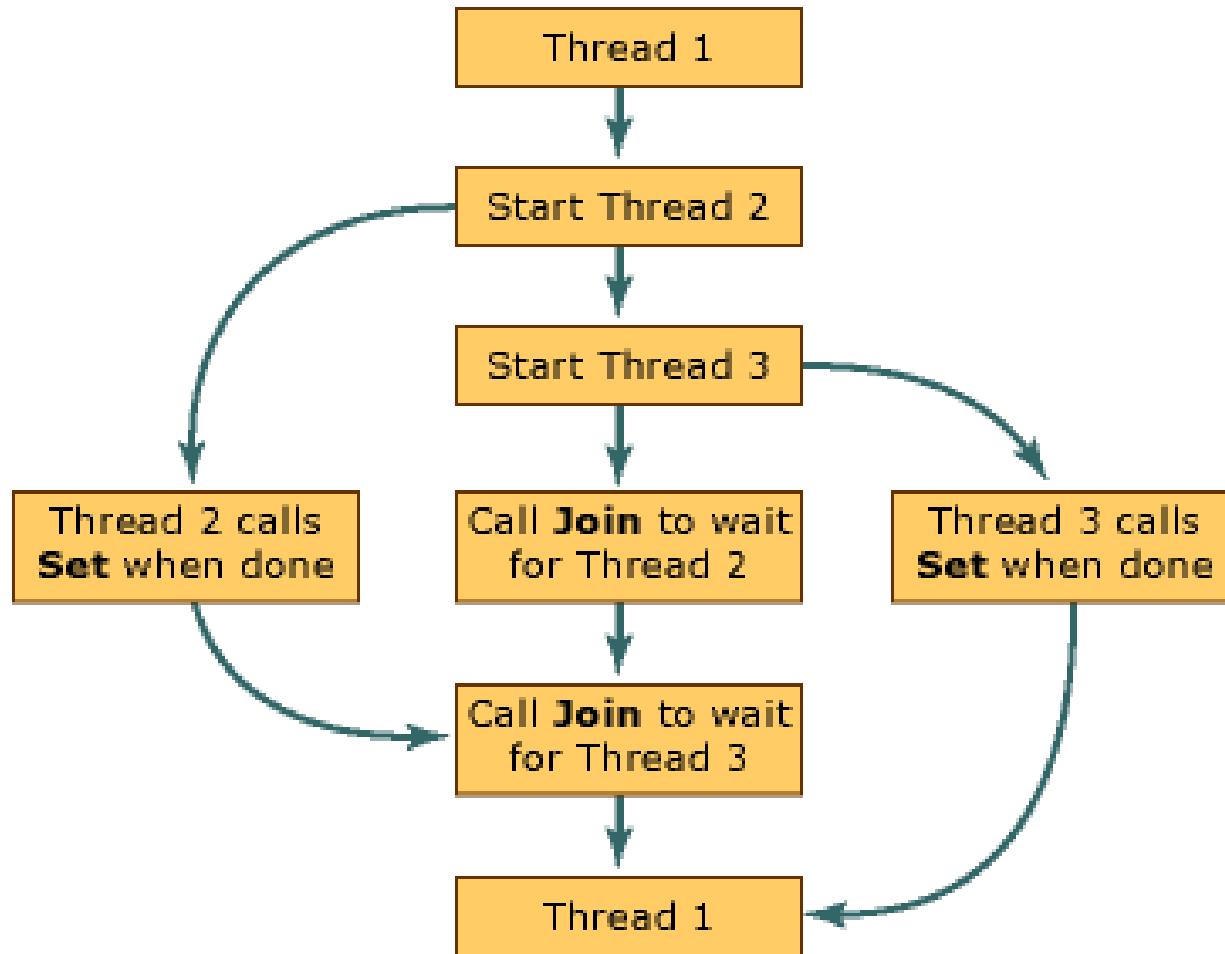
# A Warning of working with threads

- For example, if a thread closes a file, the file is closed for all threads
- If the arg passed in the create function does not take into account the data is shared between threads it may result in unforeseen errors:
  - see **thread\_unsafe.c**
  - In this example the variable t is passed by reference and its data is common to each thread so the “by reference” can change its value so it could lead to problems.
  - **rc = pthread\_create(&threads[t], NULL, PrintHello, (void \*) &t);**
  - *The &t refers to t and t is a shared variable*

# Illustration of thread join



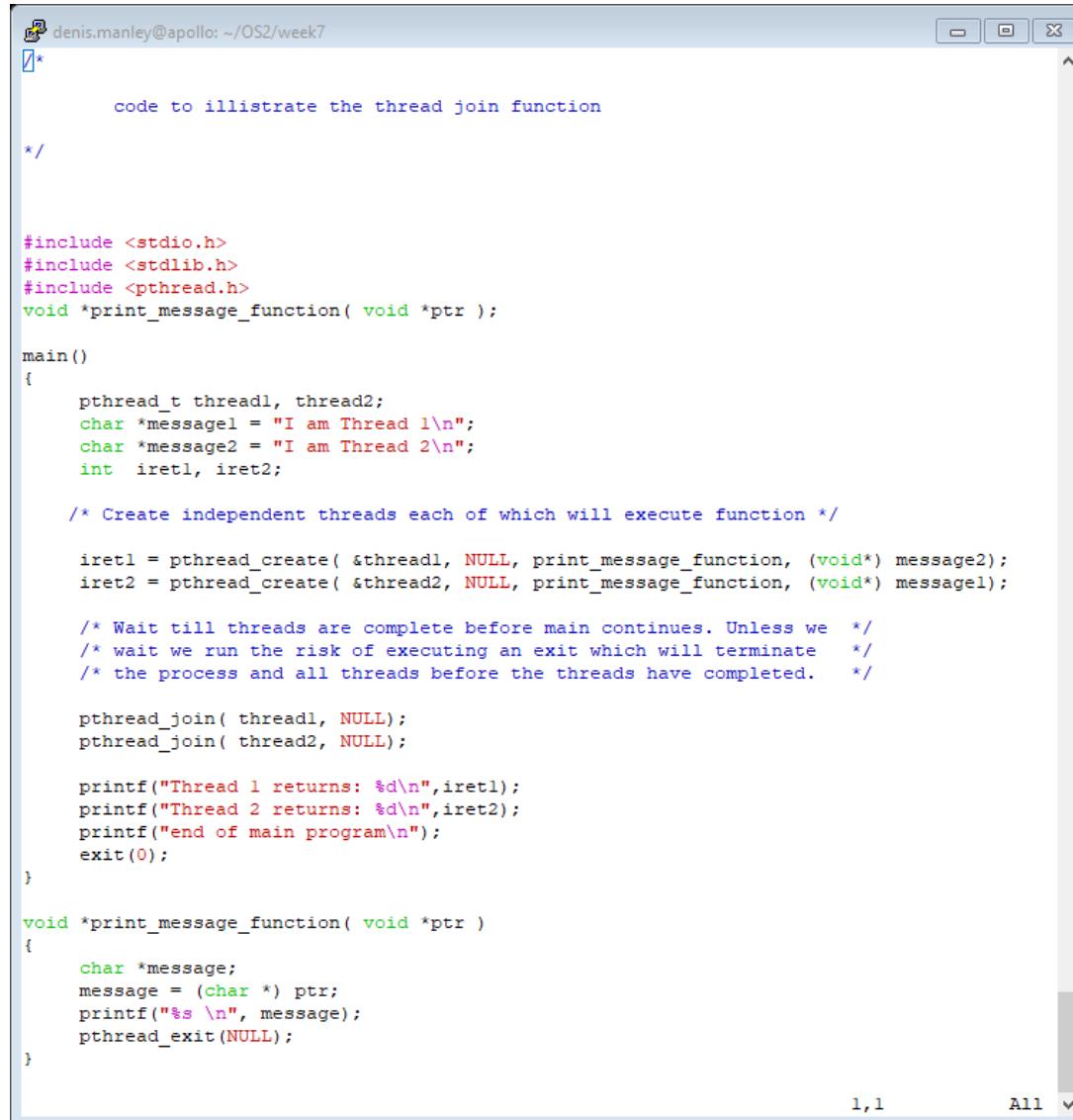
# Thread join (with two threads)



# Thread Joining

- Joining is similar to wait()
- "Joining" is one way to accomplish synchronization between threads.
- The pthread\_join() subroutine blocks the calling thread until the specified threadid thread terminates.
- On success, **pthread\_join()** returns 0; on error, it returns an error number.
- The thread specified by *thread* must be joinable.
- This is a way to ensure that the process itself does not terminate before the threads and so destroy (not execute) the threads

# Example thread join



The screenshot shows a terminal window titled "denis.manley@apollo: ~/OS2/week7". The window contains the following C code:

```
code to illustrate the thread join function

/*
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );

main()
{
    pthread_t thread1, thread2;
    char *message1 = "I am Thread 1\n";
    char *message2 = "I am Thread 2\n";
    int iret1, iret2;

    /* Create independent threads each of which will execute function */

    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message2);
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message1);

    /* Wait till threads are complete before main continues. Unless we */
    /* wait we run the risk of executing an exit which will terminate */
    /* the process and all threads before the threads have completed. */

    pthread_join( thread1, NULL);
    pthread_join( thread2, NULL);

    printf("Thread 1 returns: %d\n",iret1);
    printf("Thread 2 returns: %d\n",iret2);
    printf("end of main program\n");
    exit(0);
}

void *print_message_function( void *ptr )
{
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
    pthread_exit(NULL);
}
```

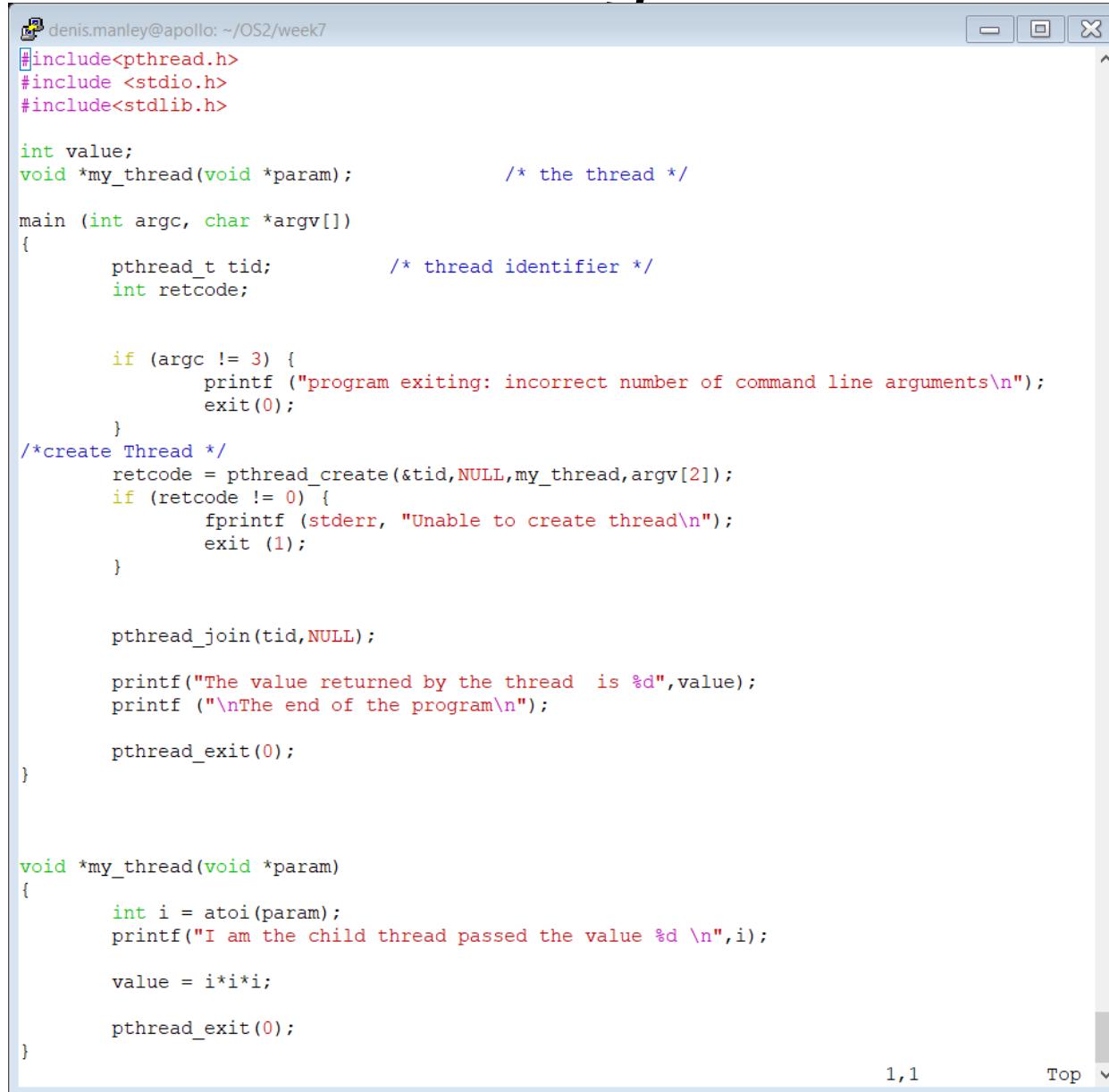
The code demonstrates the creation of two threads, each printing a message and exiting. The main thread then waits for both threads to complete using `pthread_join`, prints their return values, and exits.

Run the following two programs to see the effect of the pthread\_join() function  
**ThreadJoin.c** and **ThreadUnjoin.c**. sample output

```
ubuntu-14.04-server-i386 [Running] - Oracle VM VirtualBox
Machine View Devices Help
I am Thread 1
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./join2
I am Thread 2
I am Thread 1
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./join2
I am Thread 2
I am Thread 1
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./join2
I am Thread 2
I am Thread 1
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./join2
```

```
ubuntu-14.04-server-i386 [Running] - Oracle VM VirtualBox
Machine View Devices Help
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
Thread 1 returns: 0
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
Thread 1 returns: 0
Thread 2 returns: 0
Hi I am Thread 2
Hi I am Thread 1
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
Thread 1 returns: 0
Hi I am Thread 2
Hi I am Thread 1
Thread 2 returns: 0
end of main program
ubuntu@ubuntu-i386:/media/sf_G_DRIVE/c_labs/week8$ ./unjoin
```

# Pass command argument to Thread



The screenshot shows a terminal window titled "denis.manley@apollo: ~/OS2/week7". The window contains the following C code:

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

int value;
void *my_thread(void *param); /* the thread */

main (int argc, char *argv[])
{
    pthread_t tid; /* thread identifier */
    int retcode;

    if (argc != 3) {
        printf ("program exiting: incorrect number of command line arguments\n");
        exit(0);
    }
/*create Thread */
    retcode = pthread_create(&tid,NULL,my_thread,argv[2]);
    if (retcode != 0) {
        fprintf (stderr, "Unable to create thread\n");
        exit (1);
    }

    pthread_join(tid,NULL);

    printf("The value returned by the thread is %d",value);
    printf ("\nThe end of the program\n");

    pthread_exit(0);
}

void *my_thread(void *param)
{
    int i = atoi(param);
    printf("I am the child thread passed the value %d \n",i);

    value = i*i*i;

    pthread_exit(0);
}
```

The code demonstrates how to pass a command-line argument to a thread. The main function checks for exactly three arguments. It creates a thread using `pthread_create`, passing the address of the `my_thread` function and the second command-line argument. The `my_thread` function converts the argument to an integer, prints it, calculates the cube of the integer, and then exits the thread using `pthread_exit`. Finally, the main thread joins with the child thread and prints the calculated value before exiting.

# Joinable or Not?

- When a thread is created, one of its attributes defines whether it is joinable or detached. Only threads that are created as joinable can be joined. If a thread is created as detached, it can never be joined.
- The final draft of the POSIX standard specifies that threads should be created as joinable. It is by default.

# Other synchronisation methods

- Two other synchronization methods, mutexes and condition variables, will be discussed later.
- Semaphores is another type of synchronisation method used to prevent shared memory problems.

# Some Performance Considerations

- The performance gains from using threads is great, but can it be even better? You should consider the following when analyzing your program for potential bottlenecks:
- **Lock granularity** - How "big" (coarse) or "small" (fine) are your mutex locks? Do they lock your whole structure or fields of a structure? The more fine-grained you make your locks, the more concurrency you can gain, but at the cost of more overhead and potential deadlocks.
- **Lock ordering** - Make sure your locks are always locked in an agreed order (if they are not, make sure you take steps to rectify situations where locks are obtained in an out-of-order fashion, e.g. by using trylock/unlock calls).
- **Lock frequency** - Are you locking too often? Locking at unnecessary times? Reduce such occurrences to fully exploit concurrency and reduce synchronization overhead.
- **Critical sections** - This has been mentioned before (twice), but you should take extra steps to minimize critical sections which can be potentially large bottlenecks.

# Sample question

- What is a thread control block **(4 marks)**.
- Distinguish between a single and a multi-threading system. **(4 marks)**
- Give two examples of multi-threading **(4 marks)**
- What are the main states that a thread can undergo **(8 marks)**

# Sample Questions

- To create a thread in posix C programming a number of parameter are required. Explain in your own words what each of the following parameters of the thread function mean:
- *int pthread\_create(pthread\_t \*tidp, const pthread\_attr\_t \*attr, \*start\_rtn, void \*restrict arg)*  
**(8 marks)**

# Sample Question

- Explain the code for the **threadjoin.c** program (you will be given the code without comments) **(8 marks)**
- Explain, using sample output, what will happen if the two “`thread_join`” statements were removed **(6 marks)**
- *You will be given the above code in the exam. (you are not expected to remember and write these programs)*

# Reference material

- **Kernal threads:** The programmer has no direct control over these threads, *unless writing kernel extensions or device drivers*. See *AIX Version 4.3 Kernel Extensions and Device Support Programming Concepts* for more information about kernel programming.
- *Posix threads:*