CIIC 4050 / ICOM 5007
Operating Systems

Lecture 5:
Threads

79

Threads (1)

- A **thread** is an execution context for a processor, which handles an execution of the instructions in a process or part of them.
- In a system that does not support multiple threads in a process, each process has only one thread.
- In a system that supports multiple threads in a process, there can be many threads executing concurrently on the same process.

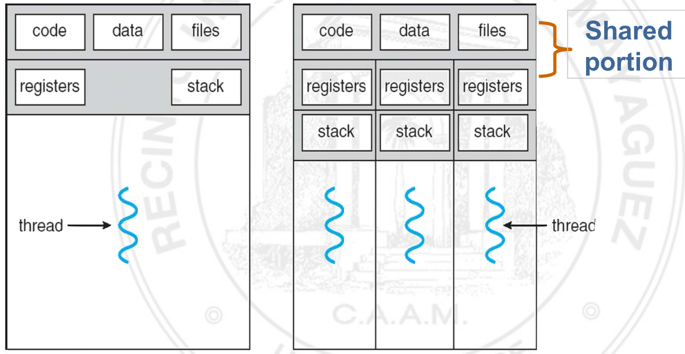
80

Threads (2)

- Each thread in a process
  - can be executing a different portion (method or subroutine) of the process's code.
- In a uniprocessor system,
  - they alternate execution as different processes do. In a multiprocessors system, they can execute in parallel.
- For each thread, the following data is maintained:
  - *ID, program counter, register set,* and *stack*. This is in addition to the other information maintained in PCB of the owner process.

81

Single and Multithreaded Processes



single-threaded process
(typical process as seen so far)

multithreaded process

82

Thread Content

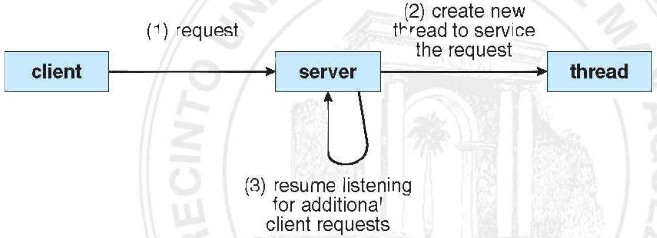
Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

- **Figure 2-7.** The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.

Thread use

- Potential uses of threads:
  - Word Processor – 1 thread display, 1 thread keystrokes, 1 thread spelling & grammar
  - Web browser – 1 thread display images, 1 thread network
  - Web server – ???

Example: Multithreaded Server Architecture



General Algorithm followed by server:

1. Initialization
2. While (true)
   a. get request
   b. initiate thread to serve request concurrently

Benefits (1)

- **Responsiveness** - threads allow a program to continue executing even if part of it is blocked or is performing a lengthy computation. This may result in faster responses during program execution.
- **Resource Sharing** - data and resources can be shared in an efficient manner That kind of sharing in processes is achieved by message passing or by shared memory, and at a major cost in terms of effort from the programmer.

Benefits (2)

- **Economy** - several of the overhead issues involved in creating and managing processes are highly minimized in threads.
- **Scalability** - benefits or multithreading can be greatly increased in multiprocessor architectures.

87

Benefits (3)

- **Thread vs Process Cost***

	User threads	LWP/kernel threads	Processes
Creation time	52	350	1700
Synchronization with semaphores	66	390	200

* on SPARC station 2 (Solaris), from Unix Internals by Uresh Vahalia, PH 1996.

88

Multicore Programming (1)

- Multicore or multiprocessor systems putting pressure on OS designers as well as on application programmers.
- Challengers for OS designers include:
  - Scheduling to allow efficient use of cores and parallel execution
  - Synchronization of threads running in different cores.

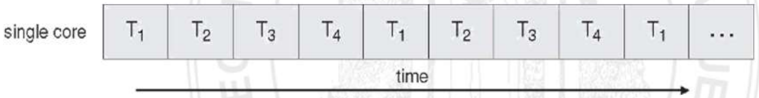
89

Multicore Programming (2)

- Challengers for application programmers include:
  - **Dividing activities** - which can be parallelized
  - **Balance** - equal effort on each parallel activity
  - **Data splitting** - divide data managed by each activity
  - **Data dependency** - between different activities and properly synchronize responsible threads.
  - **Testing and debugging** - involves more effort than in sequential processes since order of execution of parallel tasks may be undetermined.

90

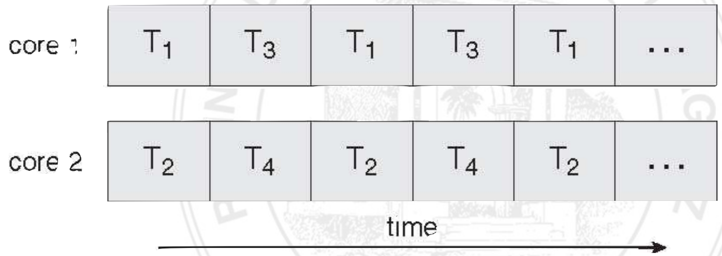
Concurrent Execution on a Single-core System



- Concurrency = Interleaved execution
(time sharing systems)

91

Parallel Execution on a Multicore System



- Concurrency = Parallel execution
(one thread/core + time sharing)

92

Management of Threads (1)

- Thread management can be provided at either:
  - *user level* - are supported above the kernel
  - *kernel level* - are supported and managed directly by the OS.
- Virtually all modern OS's support kernel threads
  - Windows XP/2003, Solaris, Linux, Tru64 UNIX, HP-UX, Mac OS X

93

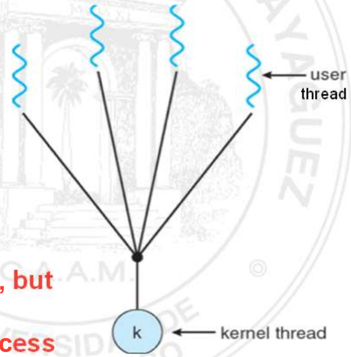
Management of Threads (2)

- Ultimately, there must be a relationship between a running user thread and a kernel thread to allow the execution of that user thread
- Different models are used to match user threads to corresponding kernel threads for their execution

94

Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

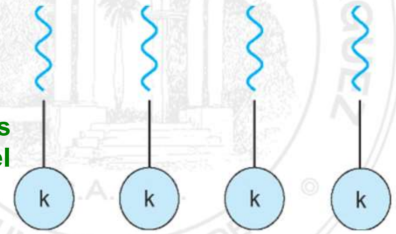


(-) Threads execute concurrently, but not in parallel…
(-) Threads are scheduled by process itself.

95

One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

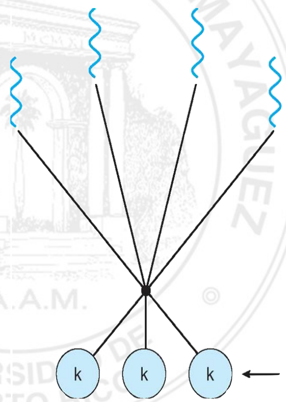


Different threads in a process can run in parallel. The kernel schedules them…
(-) more overhead since kernel needs to create one kernel thread for each user thread created

96

Many-to-Many Model

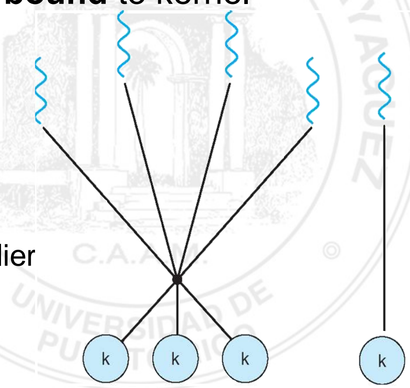
- Allows many user level threads to be mapped to many kernel threads (multiplexed)
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



97

Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



98

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Example of thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

99

PTHREADS

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library (specification), implementation is up to development of the library
- Commonly implemented in UNIX operating systems (Solaris, Linux, Mac OS X)

100

PTHREADS (example)

```
#include <pthread.h>
#include <stdio.h>
int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of attributes for the thread */
   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      /*exit(1);*/
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"Argument %d must be non-
   negative\n",atoi(argv[1]));
      /*exit(1);*/
      return -1;
   }
   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
```

101

PTHREADS (example)

```
 /* now wait for the thread to exit */
pthread_join(tid,NULL);
printf("sum = %d\n",sum);
}
/**
* The thread will begin control in this function
*/
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;
   if (upper > 0) {
      for (i = 1; i <= upper; i++)
        sum += i;
   }
   pthread_exit(0);
}
```

102

Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threading model provided by underlying OS
- Java threads may be created by:
  - Extending `Thread` class
  - Implementing the `Runnable` interface (most common)

103

Java Threads (1)

- Runnable Interface

```
public interface Runnable {
    void run();
}
```

- A class implementing the interface defines objects corresponding to threads
  - once instantiated, the thread object will start executing its `run()` method…

104

Java Threads (2)

- To create and start the execution of a thread object of type **`Runnable`**, the following is required:

  1. Create an object of type Thread while passing an instance of the **`Runnable`** object as parameter of constructor.

  2. To that object of type **`Thread`** apply method **`start()`**.

*** The new thread will automatically start executing the `run()` method of the associated `Runnable` object.

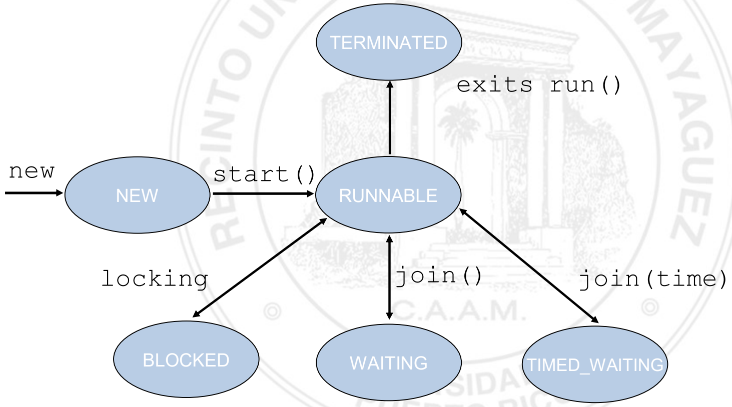
105

Java Threads States

- A Java thread may be in one of **6 possible states**
  - **New** - Thread created but run() method has not started
  - **Runnable** - when the run() method starts
  - **Blocked** - thread is waiting for a lock
  - **Waiting** - thread is waiting for an action of another thread (example: executing join())
  - **Timed waiting** - same as waiting for for a specified maximum time. Useful to eliminate the possibility of starvation…
  - **Terminated** - when the run method of the thread has finished.

106

Java Threads States



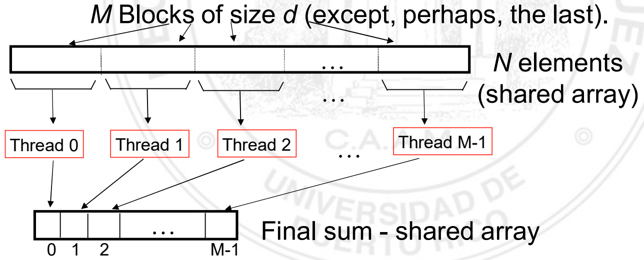
Threads: Sum of Array Elements (1)

- Given an array of *N* integers and a fixed value *d*, create
- *M* threads to add all its elements in portions of size *d*.
- One possible solution:
  - Main thread does the following
    - Create several thread
    - Each thread is assigned a portion of the array
    - Each thread computes the sum of its assigned portion
    - Results are placed in appropriate entries in a shared array

Threads: Sum of Array Elements (2)

- Given an array of *N* integers and a fixed value *d*, create *M* threads to add all its elements in portions of size *d*.

$$M = \begin{cases} N/d \text{ if } N\%d == 0 \\ N/d + 1 \text{ if not} \end{cases}$$



Java: Sum of Array Elements (1)

```
public static void main(String[] args) {
   int[] a = …; // the array to add
   int pSize = …;  // size of blocks
   int nThreads =
    (a.length % pSize == 0 ? a.length / pSize : a.length / pSize + 1);

   Barrier barrier = new Barrier(nThreads);

   int[] sum = new int[nThreads];

   for (int i=0; i<nThreads; i++) {
       Thread t =
         new Thread(new ArrayPortionAdder(a, sum, i, pSize, barrier));
       t.start();
   }

   // wait for all threads to finished as per the barrier object...
   try {
      barrier.waitForRelease();
   }
   catch(InterruptedException e) {}

   // compute the final sum from the array sum …
   …
}
```

Java: Sum of Array Elements (2)

```
public class ArrayPortionAdder implements Runnable {
   private int[] arr, sum;
   private int index, size;
   private Barrier barrier;

   public ArrayPortionAdder(int[] arr, int[] sum, int index, int size,
         Barrier b) {
        … arr is the array to sum, sum is the array were partial results are placed, …
   }
   public void run() {
      int low = index * size;
      int sup = low + size - 1;
      if (sup >= arr.length)
         sup = arr.length - 1;
      int asum = 0;

      for (int i = low; i <= sup; i++)   // thread sums its part
         asum += arr[i];

      sum[index] = asum;       // places result in shared array

      // count one more thread as finished
      …
      barrier.incCount();
      …
   }
}
```

111

Java: Object Type for Synchronization

```
public class Barrier {
  private int threshold,  count = 0;
  public Barrier(int t) {
     threshold = t;
  }
  public void reset() {
     count = 0;
  }
  public synchronized void waitForRelease() throws InterruptedException {
     while (count < threshold)
        wait();
  }
  public synchronized void incCount()
  throws InterruptedException {
     count++;
     if (count==threshold)
       notifyAll();
  }
}
```

*** This an example of a class used for synchronization – we shall study this in more detail later in the course….

112

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

113

Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?
- Some UNIX systems have two versions of fork():
  - one that duplicates all threads active in the parent process
  - one that duplicates only the thread invoking the fork()

114

Thread Cancellation (1)

- Terminating a thread before it has finished
  - Multiple threads performing database search, one returns value, others might be canceled…
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
    - Case where resources have been allocated to canceled thread, thread canceled while updating data needed by other threads
    - java – stop() (deprecated)

115

Thread Cancellation (2)

- **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
  - pthreads – cancellation points
  - java – interrupt(), isInterrupted()

116

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- A signal handler is used to process signals.
- Signal handling follows the pattern:
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled

117

Signal Handling (example)

```
/* Sample program to handle system signals */

#include <signal.h>
#include <stdio.h>

void * myhandler(int myint)
{
   printf("\nSignal Handled!!\n\n");
   exit(0);
}

int main()
{
   signal( SIGINT, (void *)  myhandler );
   signal( SIGTERM, (void *)  myhandler );

   while(1) {
       printf("Doing Nothing...\n");
       sleep(1);
   }
}
```

118

Signal Handling

- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

119

Thread Pools (1)

- To create a new thread, although simpler than to create a process, has its impacts on the system:
  - To create a new thread consumes time
  - Too many active threads may negatively impact system's performance

120

Thread Pools (2)

- An alternative is to create a number of threads in a pool where they await work. This has advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

121

Thread Pools in Java (1)

- Java allows the creation of thread pools as shown next.
- Java classes to support thread pools:

```
public interface Executor {
   // executes the given thread
   // at some time in the future
   void execute(Runnable t);
}
```

122

Thread Pools in Java (2)

- Class **Executors** - provides a set of useful methods to create objects that are relevant to thread management in JVM. One static method creates a pool of a given number of threads. Such pool is created as an object of type **ExecutorService** .

```
public static ExecutorService
       newFixedThreadPool(int maxNoThreads)
```

Interface **ExecutorService**

```
  public interface ExecutorService extends Executor {
    … several methods + execute …
}
```

123

Thread Pools in Java (2)

- Example of a Java thread pool:
- Assume that class Worker is as follows:

```
public class Worker implements Runnable {
   private static int NT = 0;
   public void run() {
     System.out.println("Thread number " + (++NT));
   }
}
```

- The following creates a thread pool:

```
Runnable r1 = new Worker();
Runnable r2 = new Worker();
Runnable r3 = new Worker();
ExecutorService pool = Executors.newFixedThreadPool(3);
pool.execute(r1);
pool.execute(r2);
pool.execute(r3);
```

124

Thread Specific Data

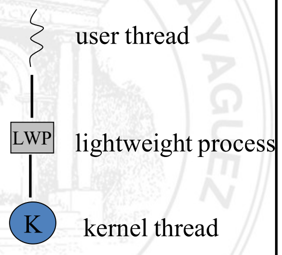
- Allows each thread to have its own copy of data
  - Transaction processing system (each thread handle separate transaction)
- Useful when you do not have control over the thread creation process (i.e., when using thread pools)

Transaction	Thread
new_order()	trans_T1
payment()	trans_T3
order_status()	trans_T5
delivery()	trans_T2
stock_value()	trans_T4

125

Scheduler Activations

- Both M:M and Two-level models require communication to dynamically maintain the appropriate number of kernel threads allocated to the application
- Place a data structure between user and kernel threads (lightweight process)
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads



126

Windows XP Threads (1)

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area

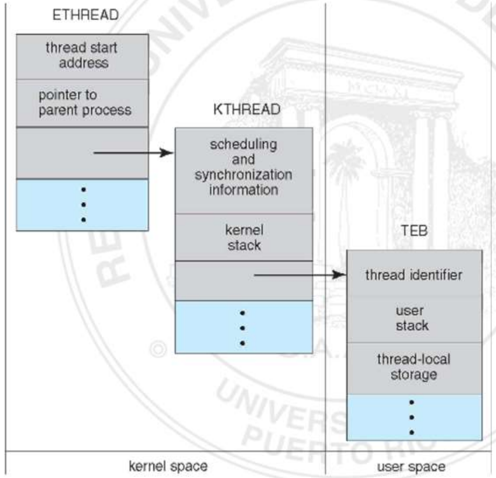
127

Windows XP Threads (2)

- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

128

Windows XP Threads



129

Linux Threads (1)

- Linux refers to them as *tasks* rather than *threads or processes*
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process), several flags control amount of sharing between parent-child, if no flags are set:
- **clone() = fork()**

130

Linux Threads (2)

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

131