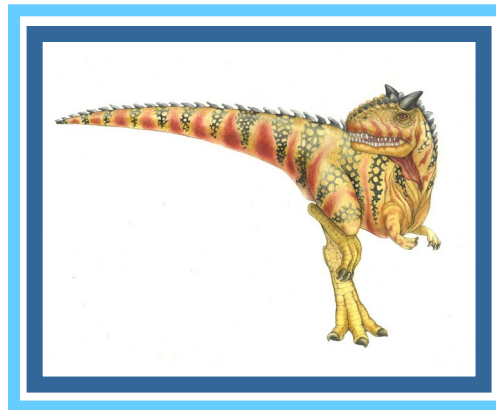# Chapter 4: Threads

Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

Objectives

- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems

- To discuss the APIs for the Pthreads, Windows, and Java thread libraries

- To explore several strategies that provide implicit threading

- To examine issues related to multithreaded programming

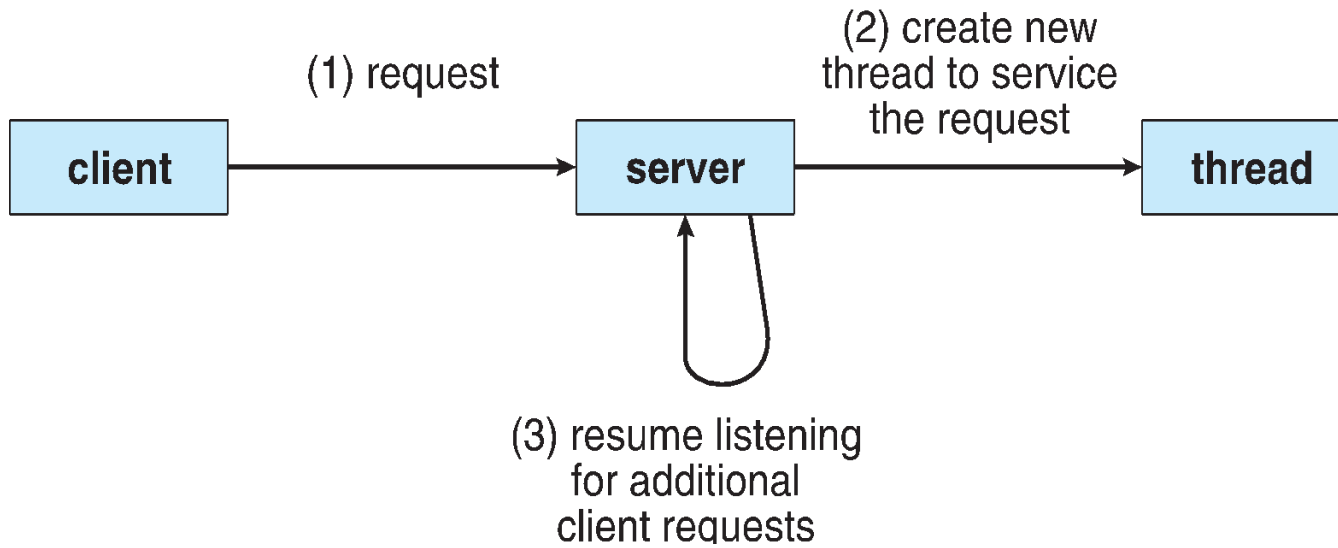- To cover operating system support for threads in Windows and Linux

Motivation

- Most modern applications are multithreaded

- Threads run within application

- Multiple tasks with the application can be implemented by separate threads

    - Update display

    - Fetch data

    - Spell checking

    - Answer a network request

- Process creation is heavy-weight while thread creation is light-weight

- Can simplify code, increase efficiency

- Kernels are generally multithreaded

(1) request

(2) create new thread to service the request

client ⟶ server ⟶ thread

(3) resume listening for additional client requests

Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces

- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing

- **Economy –** cheaper than process creation, thread switching lower overhead than context switching

- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- *Parallelism* implies a system can perform more than one task simultaneously
- *Concurrency* supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism

    - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each

    - **Task parallelism** – distributing threads across cores, each thread performing unique operation

- As # of threads grows, so does architectural support for threading

    - CPUs have cores as well as *hardware threads*

    - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core
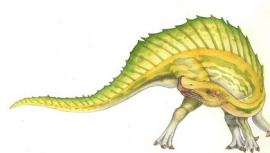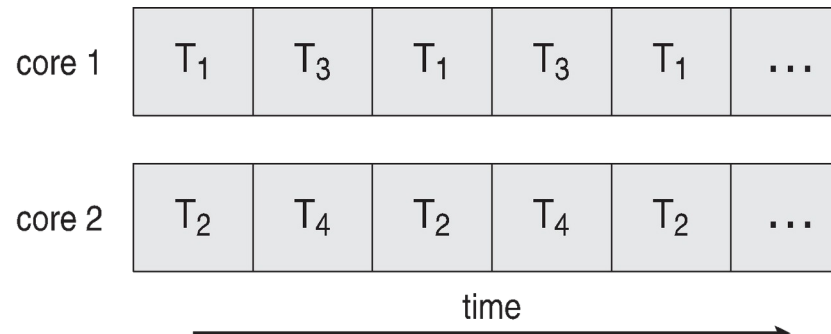
Concurrency vs. Parallelism
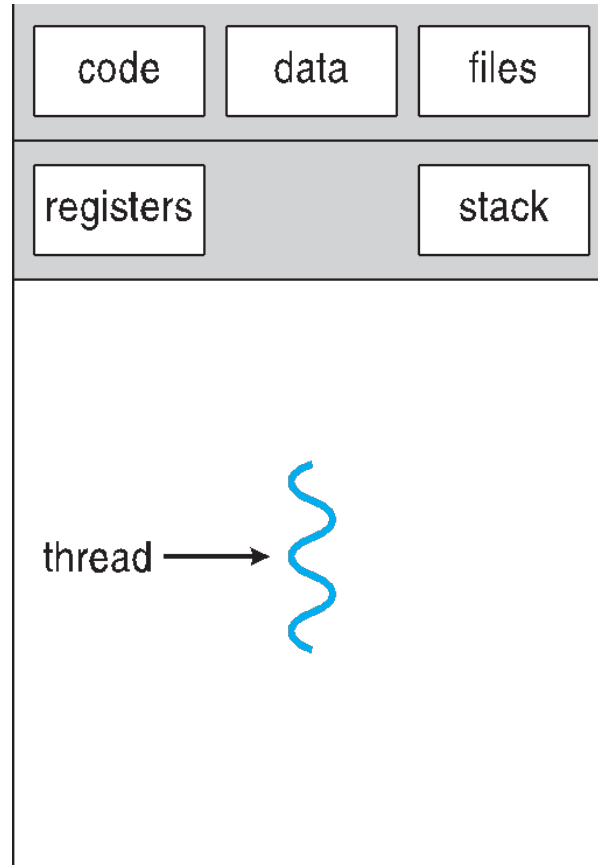
- **Concurrent execution on single-core system:**

| single core | T_1 | T_2 | T_3 | T_4 | T_1 | T_2 | T_3 | T_4 | T_1 | ... |

time →

- **Parallelism on a multi-core system:**

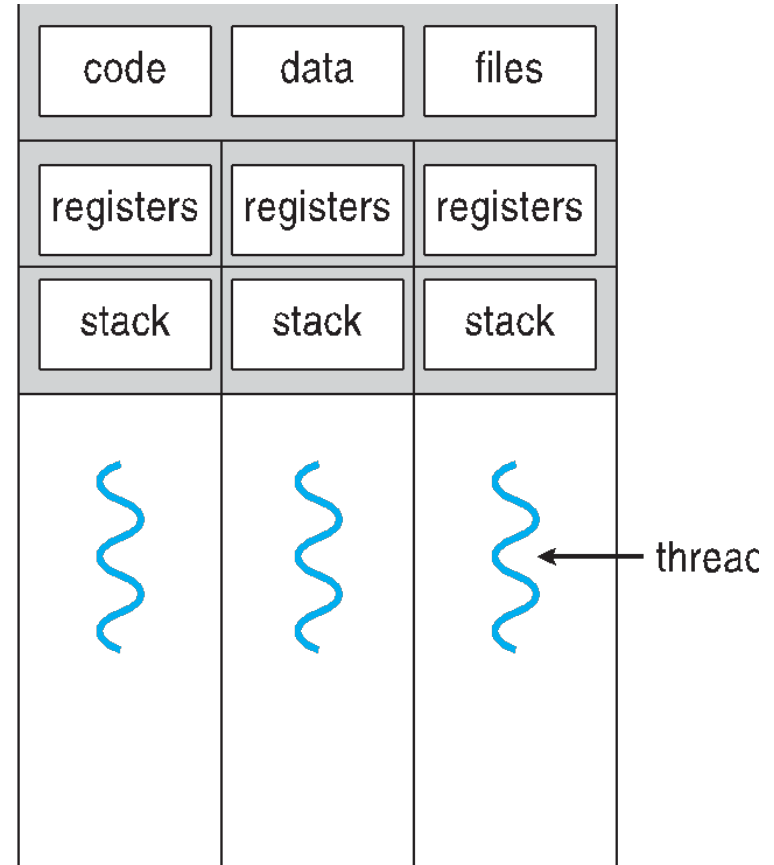| core 1 | T_1 | T_3 | T_1 | T_3 | T_1 | ... |

| core 2 | T_2 | T_4 | T_2 | T_4 | T_2 | ... |

time →

single-threaded process   multithreaded process

User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
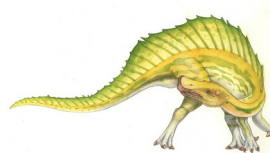
Thread Libraries

- **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

Pthreads

- May be provided either as user-level or kernel-level

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization

- *Specification*, not *implementation*

- API specifies behavior of the thread library, implementation is up to development of the library

- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
```
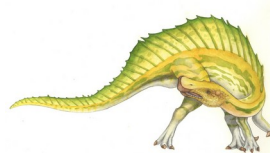
```
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```
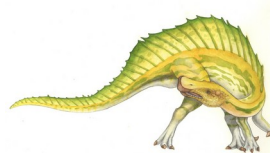
```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?

    - Some UNIXes have two versions of fork

- `exec()` usually works as normal – replace the running process including all threads

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

    1. Signal is generated by particular event
    2. Signal is delivered to a process
    3. Signal is handled by one of two signal handlers:
        1. default
        2. user-defined

- Every signal has **default handler** that kernel runs when handling signal

    - **User-defined signal handler** can override default
    - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?

  - 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

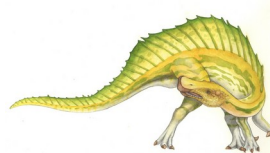 - Assign a specific thread to receive all signals for the process

Thread Cancellation

- Terminating a thread before it has finished

- Thread to be canceled is **target thread**

- Two general approaches:

  - **Asynchronous cancellation** terminates the target thread immediately

  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled

- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

End of Chapter 4