LINUX PROGRAMMING

Thread and Concurrency

www.cce.hcmut.edu.vn

06/10/2022

Le Thanh Van

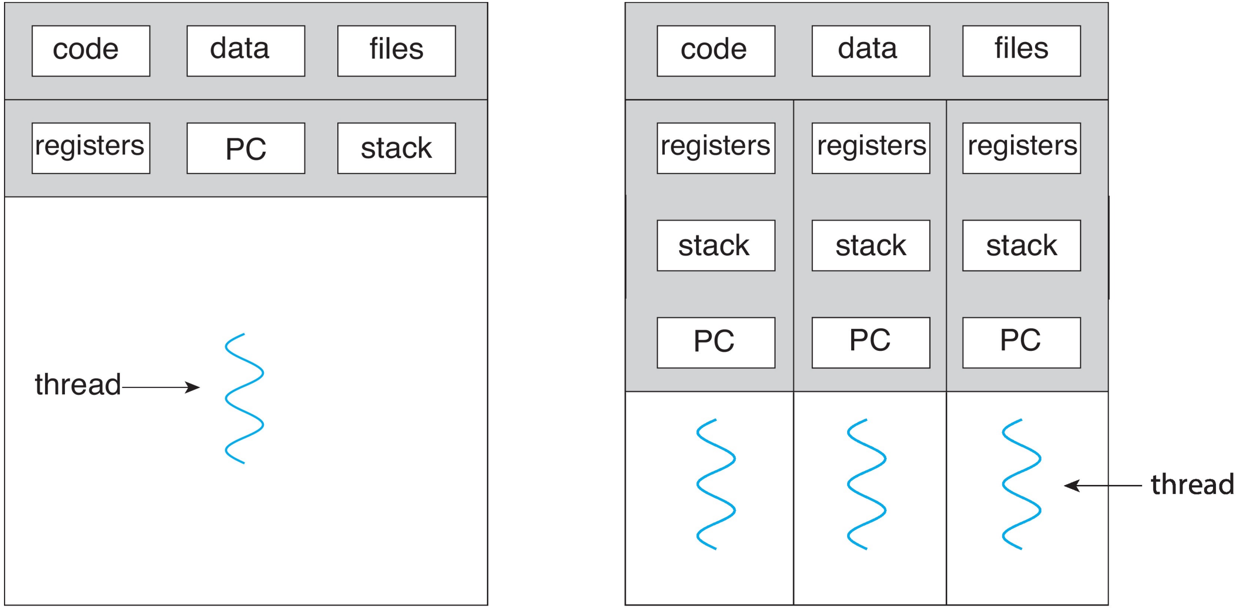
Outline

- Basic concepts
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues

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

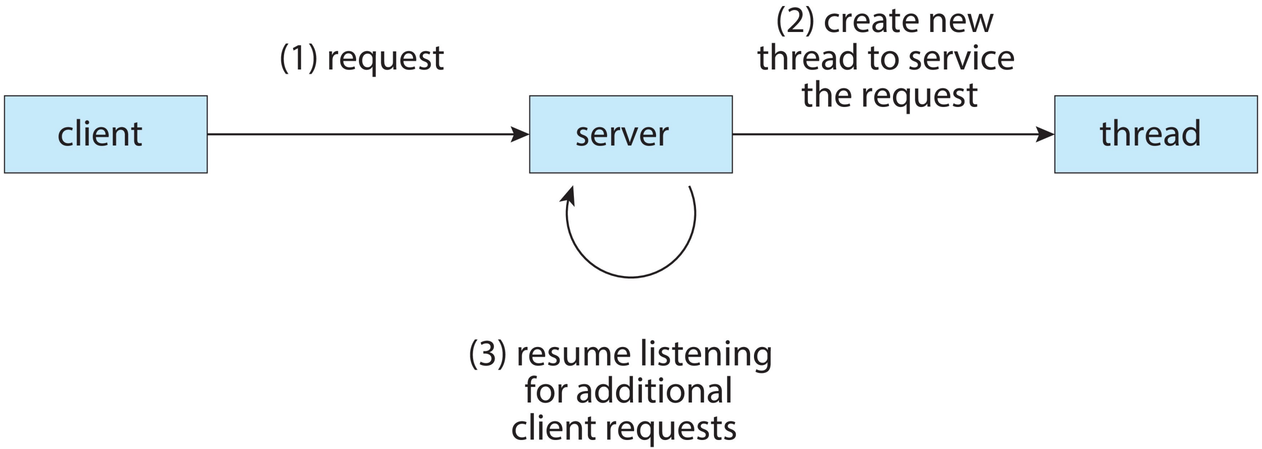
Single and Multithreaded Processes



single-threaded process

multithreaded process

Multithreaded Server Architecture



Benefits

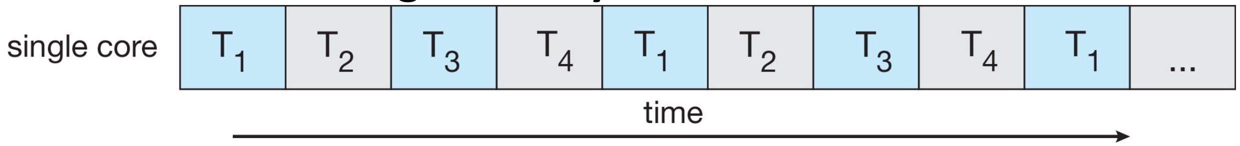
- *Responsiveness* – may allow continued execution if part of process is blocked, especially important for user interfaces
- *Resource Sharing* – threads share resources of process, easier than shared memory or message passing (IPC)
- *Economy* – cheaper than process creation, thread switching lower overhead than context switching
- *Scalability* – process can take advantage of multicore architectures

Multicore Programming

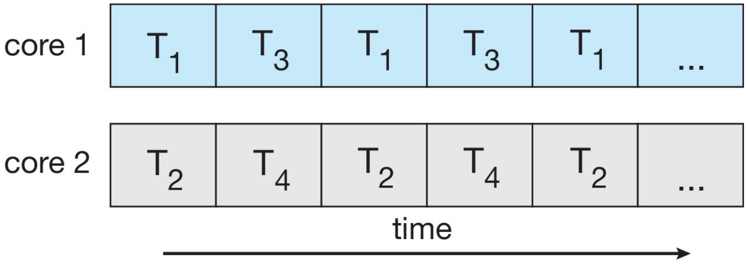
- *Multicore* or *multiprocessor systems* putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- *Parallelism implies a system can perform more than one task simultaneously*
- *Concurrency supports more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

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

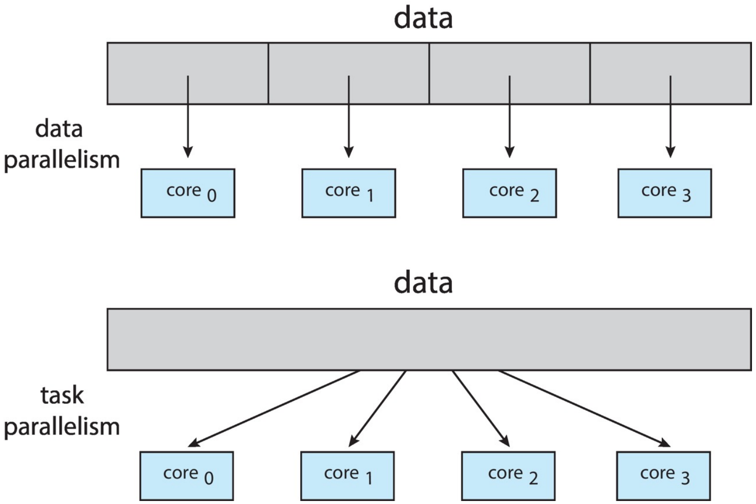


- **Parallelism on a multi-core system:**



Multicore Programming

- 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



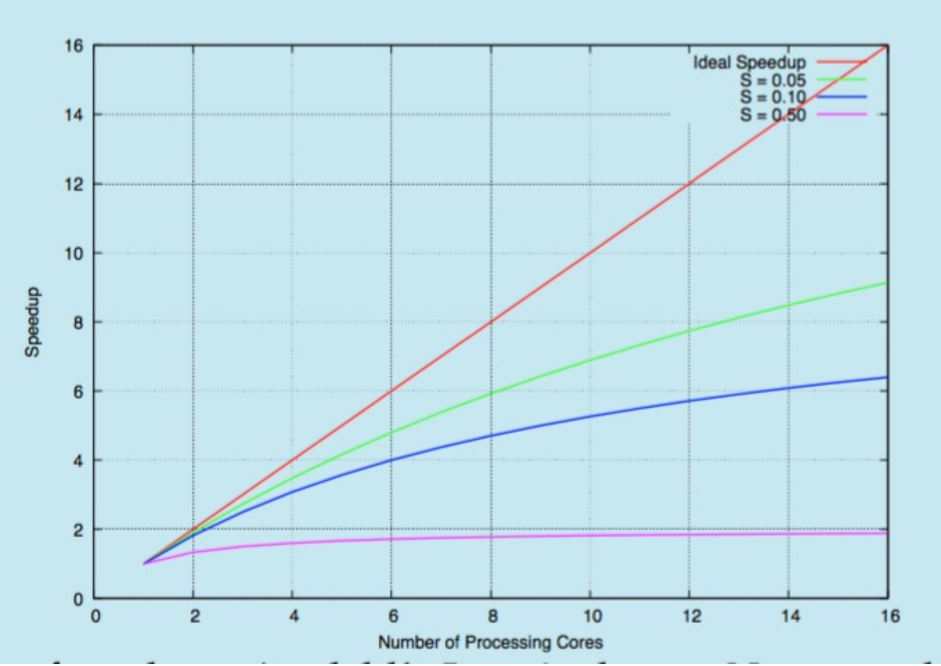
Amdahl's Law

- Identifies *performance gains* from adding additional cores to an application that has both serial and parallel components
  - **S** is serial portion
  - **N** processing cores
  - That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
  - As **N** approaches infinity, speedup approaches **1/S**

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Amdahl's Law

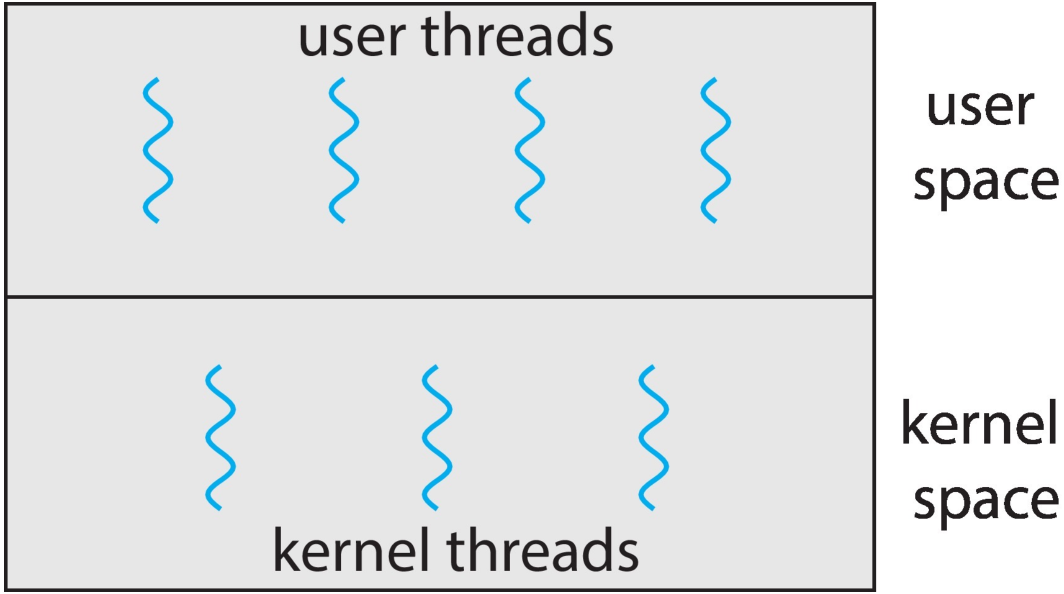


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

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, Linux, Mac OS X
  - iOS, Android

User and Kernel Threads

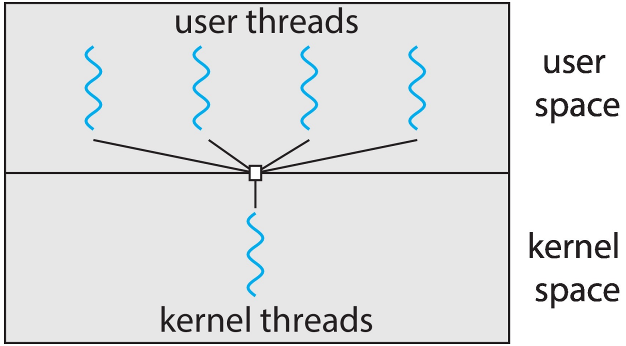


Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
- Two-level

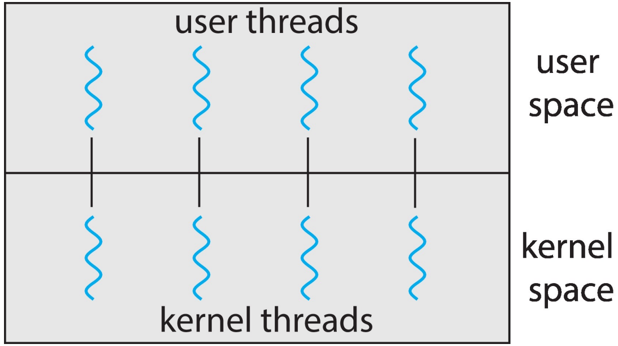
Many-to-One

- *Many user-level threads* mapped to *single kernel thread*
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- *Few systems currently use this model*
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



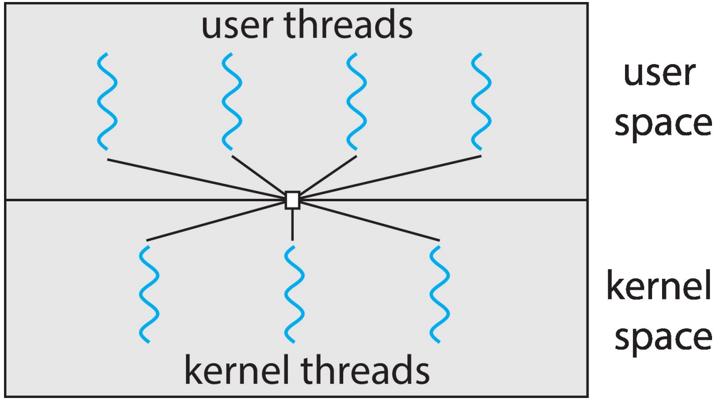
One-to-One

- *Each user-level thread* maps to *one kernel thread*
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



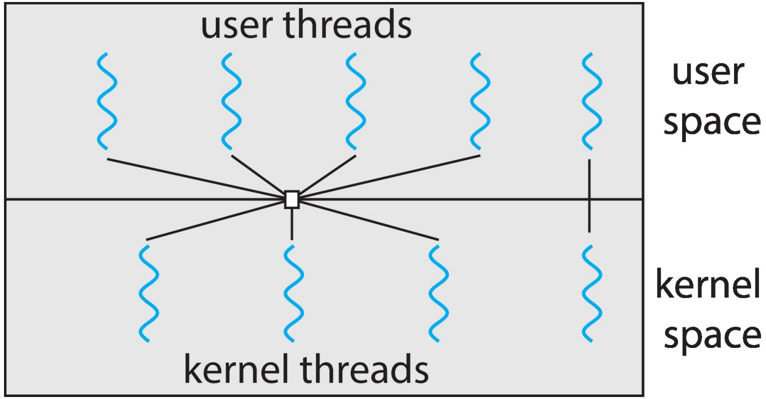
Many-to-Many Model

- Allows *many user level threads* to be mapped to *many kernel threads*
- Allows the operating system to create a sufficient number of kernel threads
- No suffer the shortcomings of many-to-one and one-to-one models
- Difficult to implement
- Contemporary concurrency libraries support many-to-many model



Two-level Model

- Similar to Many-to-Many, except that it *allows a user thread to be bound to kernel thread*



Thread Libraries

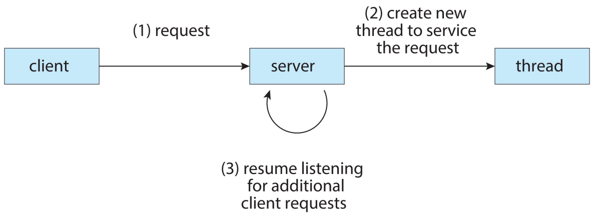
- ***Thread library*** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely *in user space*
  - Kernel-level library *supported by the OS*

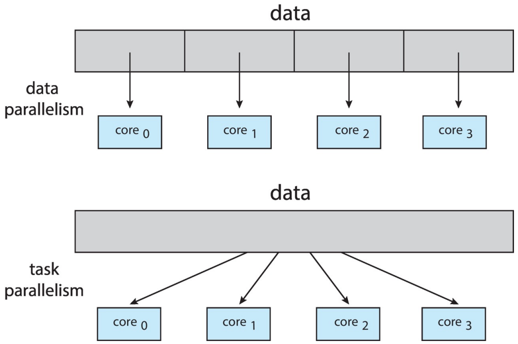
Pthreads

- 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 (Linux & Mac OS X)

Thread creation

- Asynchronous threading
  - Parent and child execute concurrently and independently of one another
  - Commonly used for designing responsive user interfaces.
- Synchronous threading
  - Parent must wait for all of its children to terminate before it resumes
  - Involves significant data sharing among threads, parent thread may combine the results calculated by its various children





Synchronous or asynchronous threading?

Thread creation (cont)

- #include<pthread.h>
- pthread_t tid: declares the identifier tid for the thread
- pthread_attr_t attr: represents the attributes for the thread, including stack size and scheduling information
- pthread attr init(&attr): set the default attributes
- pthread_create(): create a separate thread
- pthread_join(): parent thread will wait for child thread to terminate
- pthread_exit(): child thread call to terminate

pthread_create() function

- A pointer to a pthread_t variable, tid
- A pointer to a thread attribute object. If NULL, a thread created with the default thread attributes
- A pointer to the thread function with type void*.
- A thread argument value of type void*

Thread function take a parameter of type void* and have a void* return type.

Linux schedules both threads asynchronously.

Pthreads Example

Constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread:

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

#include <stdlib.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 */

  /* set the default attributes of the thread */
  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);
}
```

Pthreads Example (cont)

Constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread:

```
/* The thread will execute 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);
}
```

Pthreads Example (cont)

Constructing a multithreaded program that calculates the summation of a non-negative integer in a separate thread:

```
#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);
```

Passing Data to threads

- Can’t pass a lot of data directly via the argument
- Define structure for each thread function, representing parameters needed.

```
struct char_print_parms
{
    /* The character to print.*/
    char character;
    /* The number of times to print it. */
    int count;
};
```

```
void* char_print (void* parameters)
{
```

```
pthread_t thread1_id;
struct char_print_parms thread1_args;
. . .
pthread_create (&thread1_id, NULL, &char_print, &thread1_args);
```

Implicit Threading

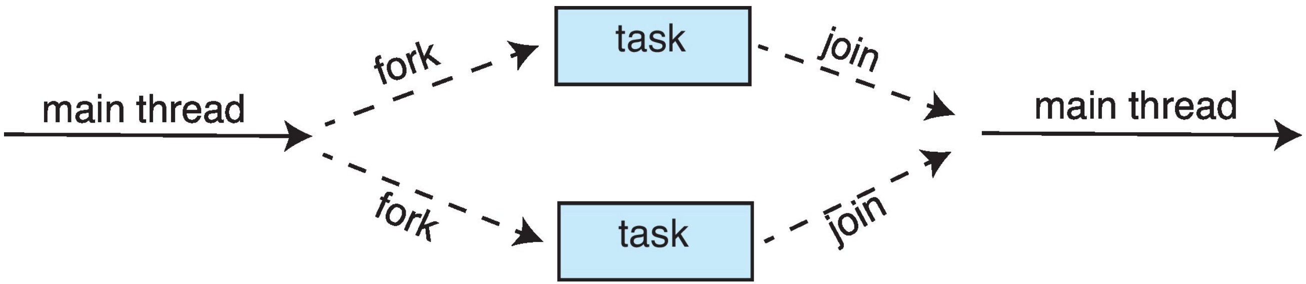
- Growing in popularity as numbers of threads increase, program correctness more difficult with *explicit threads*
- *Creation and management of threads done by compilers and run-time libraries* rather than programmers
- Thread Pools
- Fork-Join

Thread Pools

- Create a *number of threads in a pool* where they await work
- 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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e., Tasks could be scheduled to run periodically

Fork-Join Parallelism

- *Multiple threads (tasks) are forked*, and then *joined*.



Fork-Join Parallelism

- General algorithm for *fork-join strategy*:

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Threading Issues

- Semantics of `fork()` and `exec()` system calls
  - Does `fork()` duplicate only the calling thread or all threads?
  - `exec()` usually works as normal – replace the running process including all threads
- Thread cancellation of target thread
  - Asynchronous or deferred (synchronous)

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 (synchronous) 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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - i.e., **`pthread_testcancel()`**
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

Thread cancelation

- pthread_cancel()
- Initially created, a thread is synchronously cancelable
- pthread_setcanceltype to make a thread asynchronously cancelable.
- int pthread_setcanceltype(int type, int *oldtype);
  - type: PTHREAD_CANCEL_ASYNCHRONOUS, PTHREAD_CANCEL_DEFERRED, PTHREAD_CANCEL_DISABLE
  - oldtype: if not NULL, receive the previous cancellation type for the thread

PTHREAD_CANCEL_DISABLE can be used to protect the critical section

```
pthread_setcancelstate (PTHREAD_CANCEL_DISABLE,
&old_cancel_state);
//Critical section
pthread_setcancelstate (old_cancel_state, NULL);
```

Cancellation point creation

- pthread_testcancel(): invoke a cancellation point.
- If a cancellation request is found to be pending, the call to pthread testcancel() will not return, and the thread will terminate; otherwise, the call to the function will return, and the thread will continue to run

PROCESS	THREAD
A process is an instance of a program that is being executed or processed.	Thread is a segment of a process or a lightweight process that is managed by the scheduler independently.
Processes are independent of each other and hence don't share a memory or other resources.	Threads are interdependent and share memory.
Each process is treated as a new process by the operating system.	The operating system takes all the user-level threads as a single process.
If one process gets blocked by the operating system, then the other process can continue the execution.	If any user-level thread gets blocked, all of its peer threads also get blocked because OS takes all of them as a single process.
Context switching between two processes takes much time as they are heavy compared to thread.	Context switching between the threads is fast because they are very lightweight.
The data segment and code segment of each process are independent of the other.	Threads share data segment and code segment with their peer threads; hence are the same for other threads also.
The operating system takes more time to terminate a process.	Threads can be terminated in very little time.
New process creation is more time taking as each new process takes all the resources.	A thread needs less time for creation.

Exercise 1

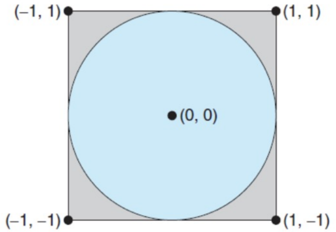
Given the content of a C source file named “code.c”

```
#include <stdio.h>
#include <pthread.h>
void * hello(void * tid) {
        printf("Hello from thread %d\n", (int)tid);
}
int main() {
        pthread_t tid[10];
        int i;
        for (i = 0; i < 10; i++) {
                pthread_create(&tid[i], NULL, hello, (void*)i);
        }
        pthread_exit(NULL);
}
```

Modify this program to have those threads to print their thread ID in ascending order every time we run the program.

Exercise 2

- The Monte-Carlo method presents an interesting way of calculating pi number (pi=3.14). Given a circle inscribed within a square, the pi number will be calculated as follows:



  – Generate a series of random points with simple (x, y) coordinates. These points must fall within the Cartesian coordinates that bound the square.
  – Determine if the point is inside the circle by using this formular: x^2+y^2 < 1.
  – Count the total points belong to the circle.
  – Next, estimate pi by performing the following calculation:

    pi = 4 x (number of points in circle) / (total number of points)

- By using Monte-Carlo method, write pseudo-code for calculating pi with **multiple threads**.

THANK YOU !

Center Of Computer Engineering

www.cce.hcmut.edu.vn