Operating Systems

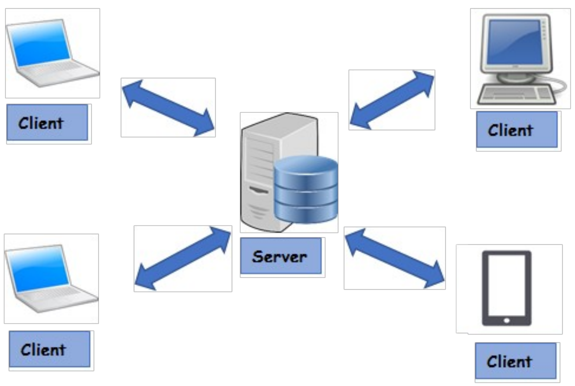
Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

Threads Concepts and API

A Client-Server program

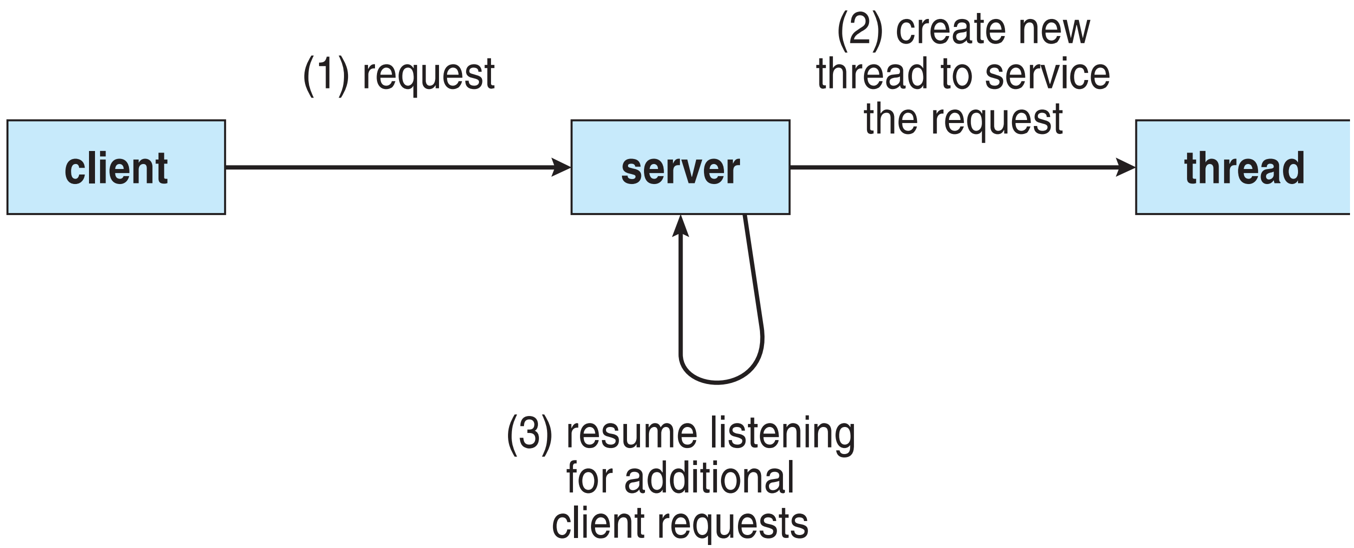
Assume you have a server that is responsible for responding some clients. The clients frequently ask the server to send a requested large file. How does server manage the requests from the clients? What is the problem? What is the solution?



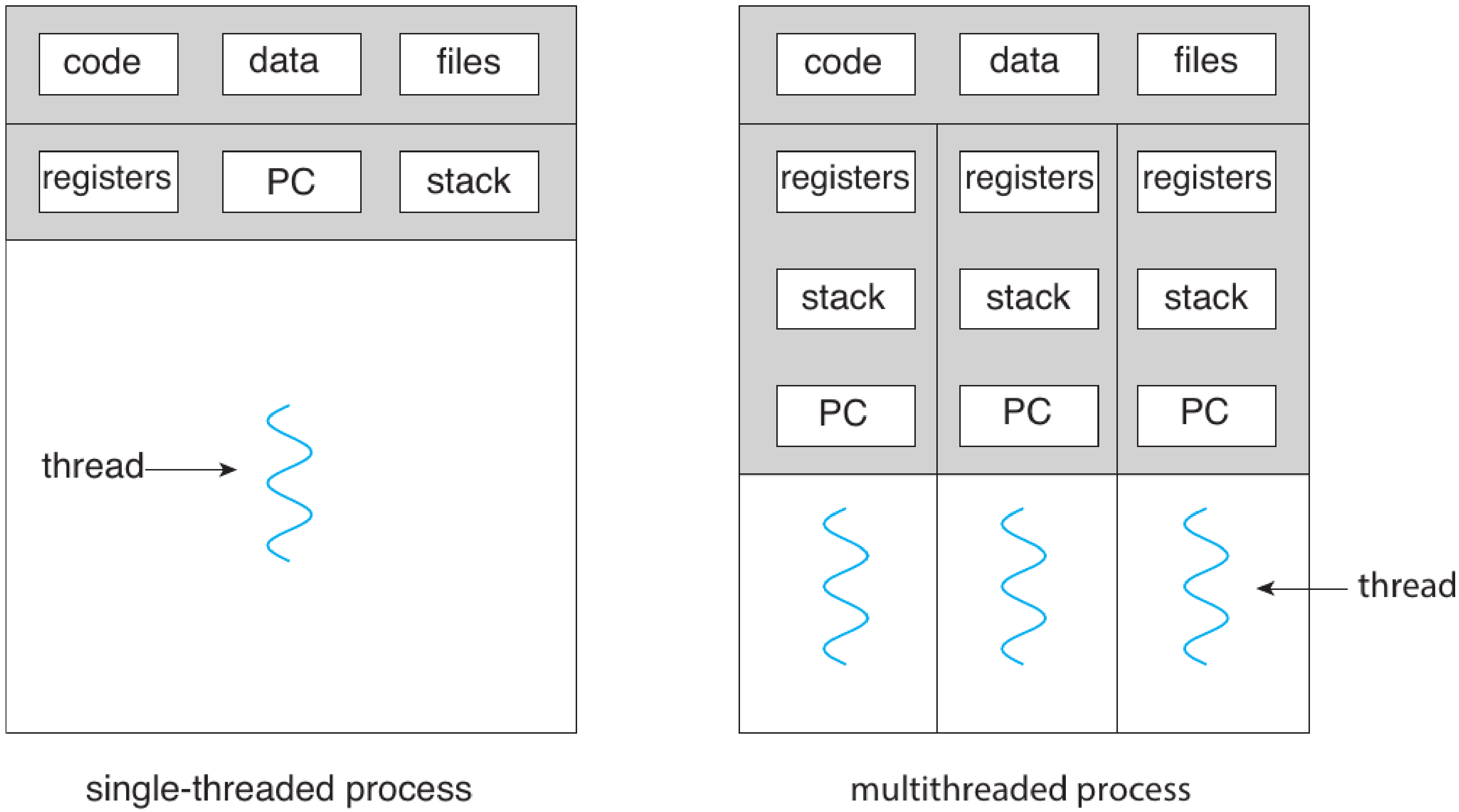
Motivation

- 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
- Examples of multi-thread applications: basic sorting, trees, and graph algorithms, programmers who must solve contemporary CPU-intensive problems in data mining, graphics, and artificial intelligence can leverage the power of modern multicore systems by designing solutions that run in parallel.

Multithreaded Server Architecture



Single and Multithreaded Processes



Benefits

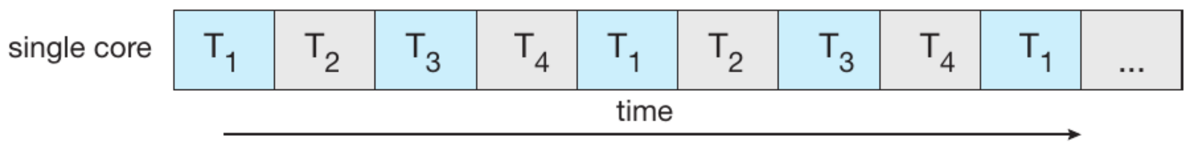
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces (if the time-consuming operation is performed in a separate, asynchronous thread, the application remains responsive to the user)
- **Resource Sharing –** threads share the memory and the resources of the process to which they belong by default, so easier than shared memory or message passing between processes
- **Economy –** thread creation consumes less time and memory than process creation. Additionally, context switching is typically faster between threads than between processes
- **Scalability –** a single process can take advantage of multiprocessor architectures

Multi-thread kernel

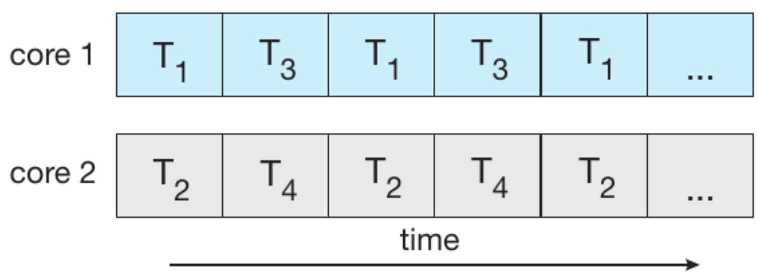
- Most operating system kernels are also typically multithreaded
- The command ps -ef can be used to display the kernel threads on a running Linux system
    - Examining the output of this command will show the kernel thread kthreadd (with pid = 2), which serves as the parent of all other kernel threads.

Concurrency vs. Parallelism

- ***Concurrency*** *supports more than one task making progress*
  - Single processor / core, scheduler providing concurrency
- ***Parallelism*** implies a system can perform more than one task simultaneously
- **Concurrent execution on single-core system:**

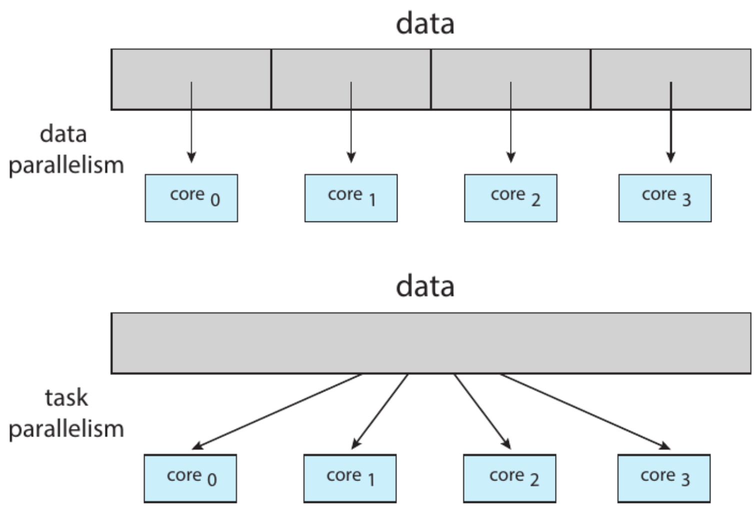


- **Parallelism on a multi-core system:**



Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
    - Ex: calculating an array sum or matrix multiplication
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation
    - Ex: calculating different statistical operation on the array of elements
  - **Hybrid**



Multicore Programming challenges

- **Dividing activities:** examining applications to find areas that can be divided into separate, concurrent tasks
- **Balance:** ensure that the tasks perform equal work of equal value.
- **Data splitting:** the data accessed and manipulated by the tasks must be divided to run on separate cores
- **Data dependency:** When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency
- **Testing and debugging:** When a program is running in parallel on multiple cores, many different execution paths are possible making debugging difficult

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

تحقیق

- با بررسی در هدر فایل sched.h و جستجو، تحقیق کنید آیا برای مدیریت threadها نیز ساختاری مشابه task_struct در سیستم عامل برای هر thread استفاده میشود؟ یا روش دیگری وجود دارد؟ تفاوتها را مشخص کنید
- مدل threadها در زبانهای c، جاوا و پایتون را مقایسه کنید.
- نحوه ساخت kernel thread و user thread با استفاده از system callهایی شبیه fork به چه صورت است؟

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

- 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)

Pthread.h

- set the thread attributes
  - Pthread_attr_init(pthread_attr_t * attr)
  - Examples: setting the stack size or information about the scheduling priority of the thread
- create the thread
  - pthread_create(pthread_t *tid, pthread_attr_t *attr_t, void* thread_runner, void *thread_runner_args)
  - thread_runner: function pointer
  - having a void pointer as an argument to the function thread_runner allows us to pass in any type of argument; having it as a return value allows the thread to return any type of result.

Pthread.h

- wait for the thread to exit
  - pthread_join(pthread_t *tid, void ** thread_runner_ret_val)
  - The first is of type pthread t, and is used to specify which thread to wait for.
  - The second argument is a pointer to the return value you expect to get back.
- terminates the calling thread
  - pthread_exit(void * pthread_runner_ret_val)
  - returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls

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);
}
```

Pthreads Code for Joining 10 Threads

```
#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);
```

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
- some methods explored
  - Thread Pools
  - OpenMP
  - Fork-Join
  - Grand Central Dispatch
  - Intel Threading Building Blocks

Client-Server Example

Remember our own example

What happens if the number of alive threads exceeds the maximum number of concurrent threads that the system can support?

How can we prevent this issue?

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
- Windows API supports thread pools:

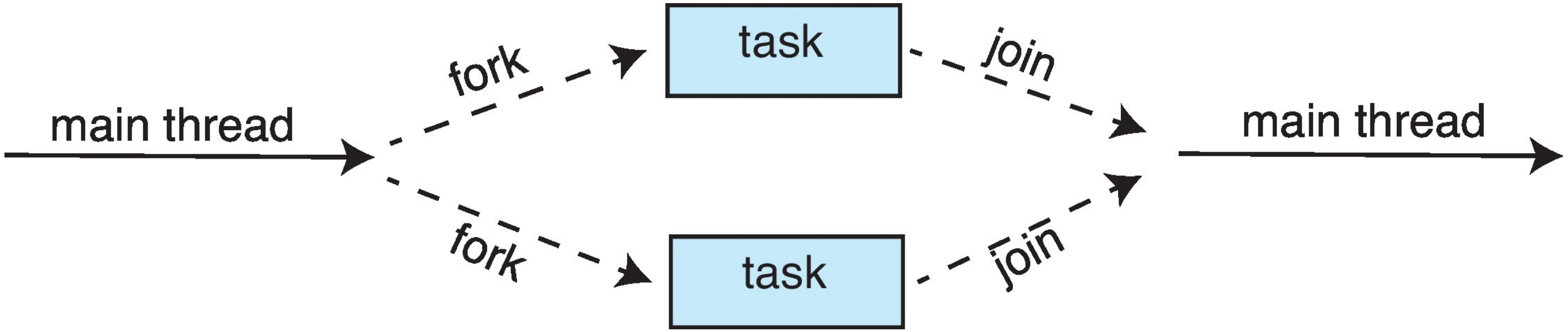
```
DWORD WINAPI PoolFunction(AVOID Param) {
   /*
   * this function runs as a separate thread.
   */
}
```

Sample thread pool API

```
void first_task() {…}
void second_task() {…}
void third_task() {…}
main(){
    // Create a thread pool.
    pool tp(2);
    //Add some tasks to the pool.
    tp.schedule(&first_task);
    tp.schedule(&second_task);
    tp.schedule(&third_task);
}
```

Fork-Join Parallelism

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



Fork-Join Parallelism

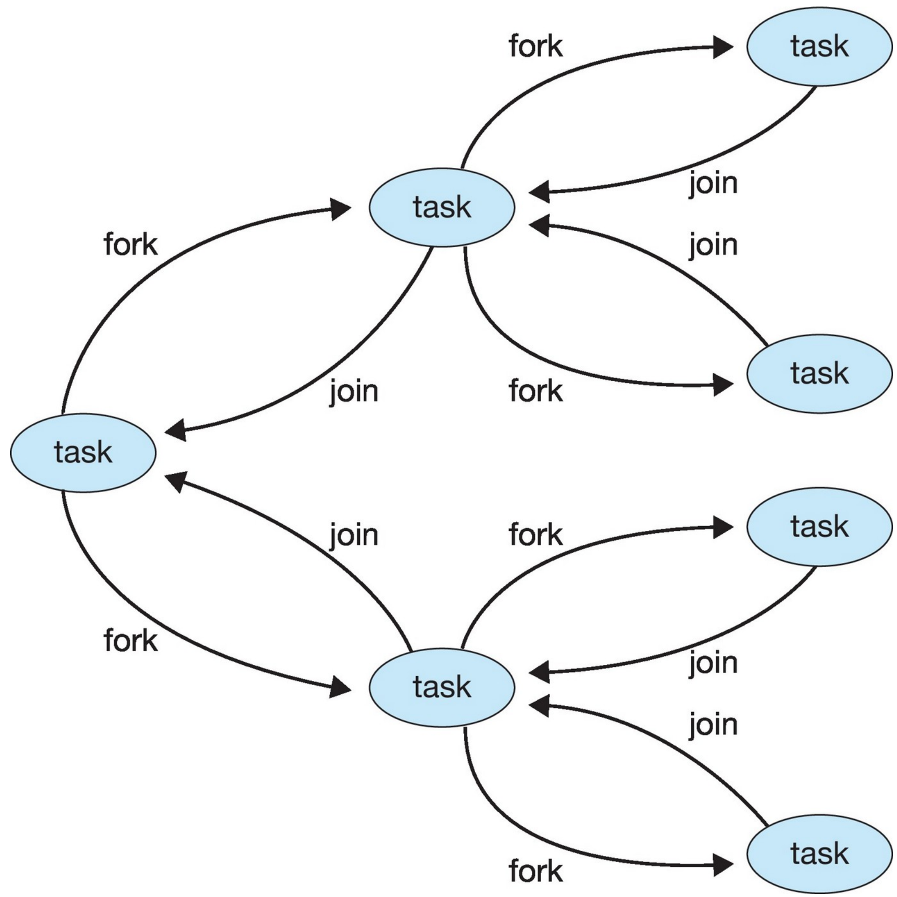
- General algorithm for fork-join strategy:

```
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
```

Fork-Join Parallelism



OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN provides support for parallel programming
- Identifies **parallel regions** – blocks of code that can run in parallel

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred

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
    - one that duplicates only the thread that invoked the fork() system call.
- `exec()` usually works as normal – replace the running process including all threads
  - So when exec() is called immediately after forking, forking only the calling thread is sufficient

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 processes

kill(pid_t, signal)

pthread_kill(thread_t, signal)

Thread Cancellation

- Terminating a thread before it has finished
  - Suppose multiple threads searching for a record in a database and one of them find it
  - a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.
- 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 canceled

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**
    - Ex: **`pthread_testcancel()`**
    - `Ex: read function`
- On Linux systems, thread cancellation is handled through signals

Thread-Local Storage

- **Thread-local storage** (**TLS**) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to `static` data
  - TLS is unique to each thread

Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

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.

- **`struct task_struct`** points to process data structures (shared or unique)

getconf GNU_LIBPTHREAD_VERSION

End of Chapter 4