Amirkabir University of Technology
(Tehran Polytechnic)
Department of Computer Engineering

Threads (ریسمان‌ها، نخ‌ها)

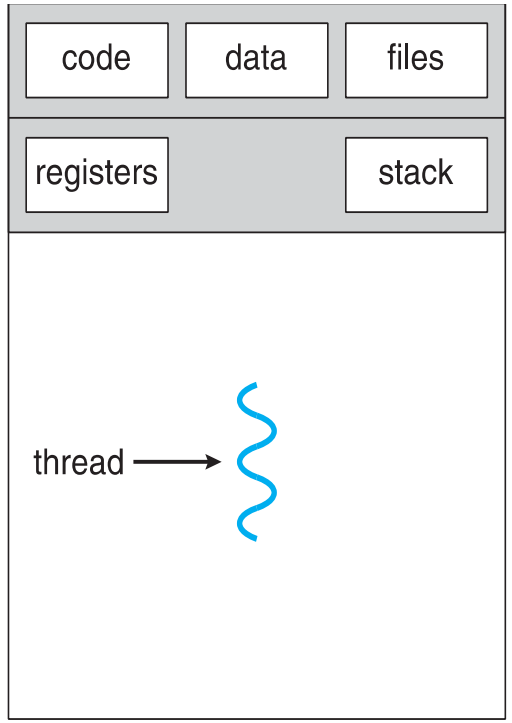
Hamid R. Zarandi
h_zarandi@aut.ac.ir

Definition

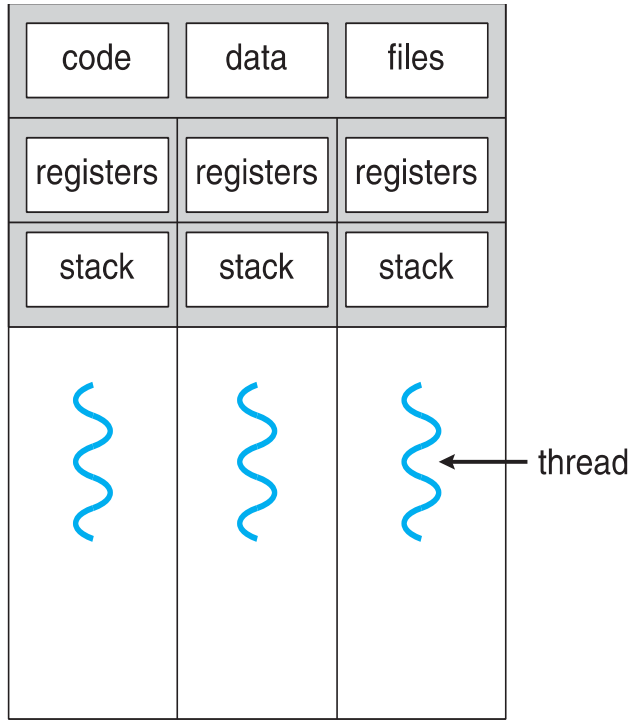
- A basic unit of CPU utilization
  - Private: Thread ID, program counter, register set, stack
  - Shared: code section, data section, OS resources (IO & file)

- Examples:
  - Web browsers
  - Word processors
  - Database engines
  - RPC!

- Versus Process:
  - Time consuming
  - Resource intensive

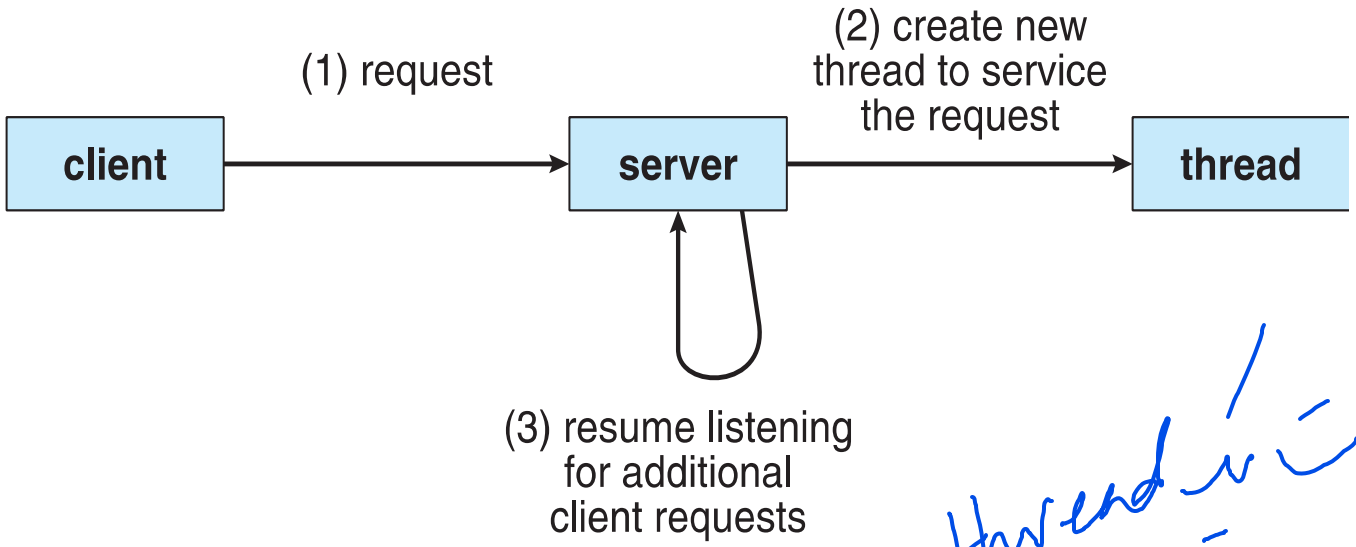


single-threaded process



multithreaded process

Web server application



Advantages of using threads

- **Responsiveness**
  - Allowing a program to continue running even part of it is blocked or lengthy

→ spell checker →

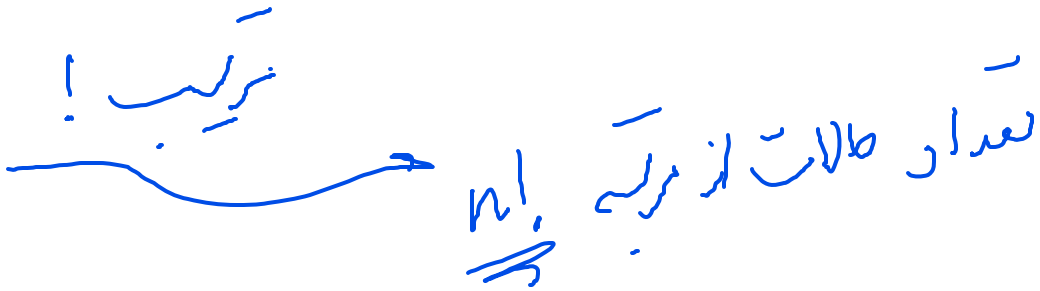
- **Resource sharing**
  - Memory, resources

- **Economy**
  - Fast

- **Scalability** ∝ #cores
  - Threads may be running in parallel on processing cores

Multicore programming

- **Multicore or multiprocessor systems putting pressure on programmers, challenges include:**
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging



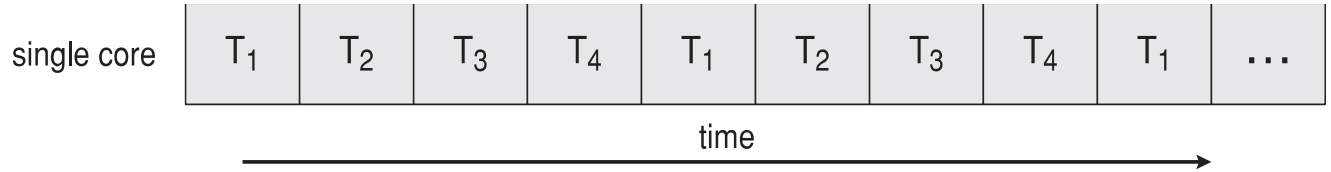
- ***Parallelism*** **implies a system can perform more than one task simultaneously**

↳ # core > 1

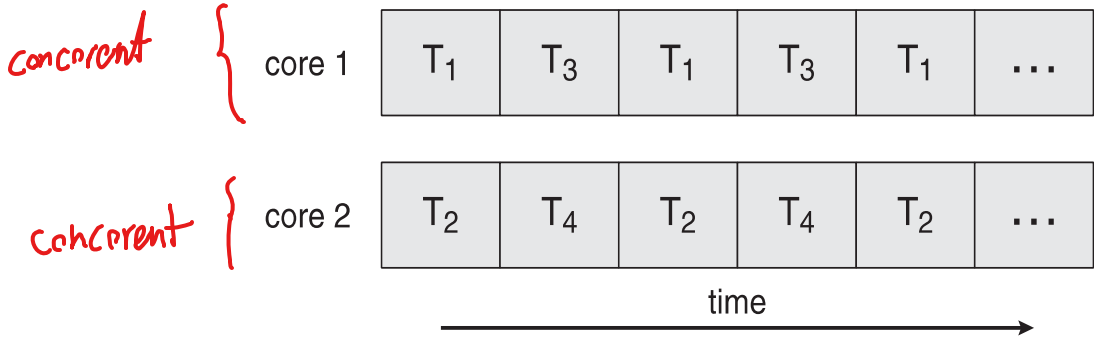
- ***Concurrency*** **supports more than one task making progress**
  - Single processor / core, scheduler providing concurrency

Multicore programming

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



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



AMDAHL'S LAW

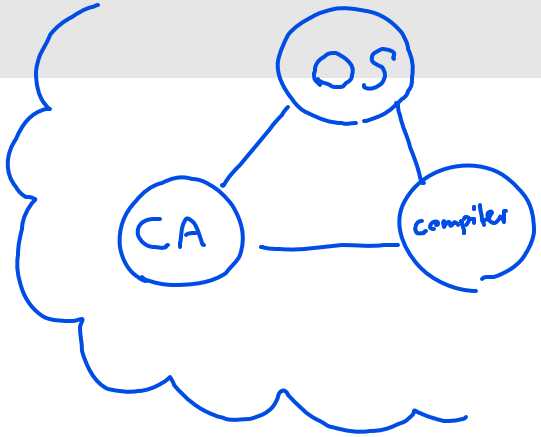
$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

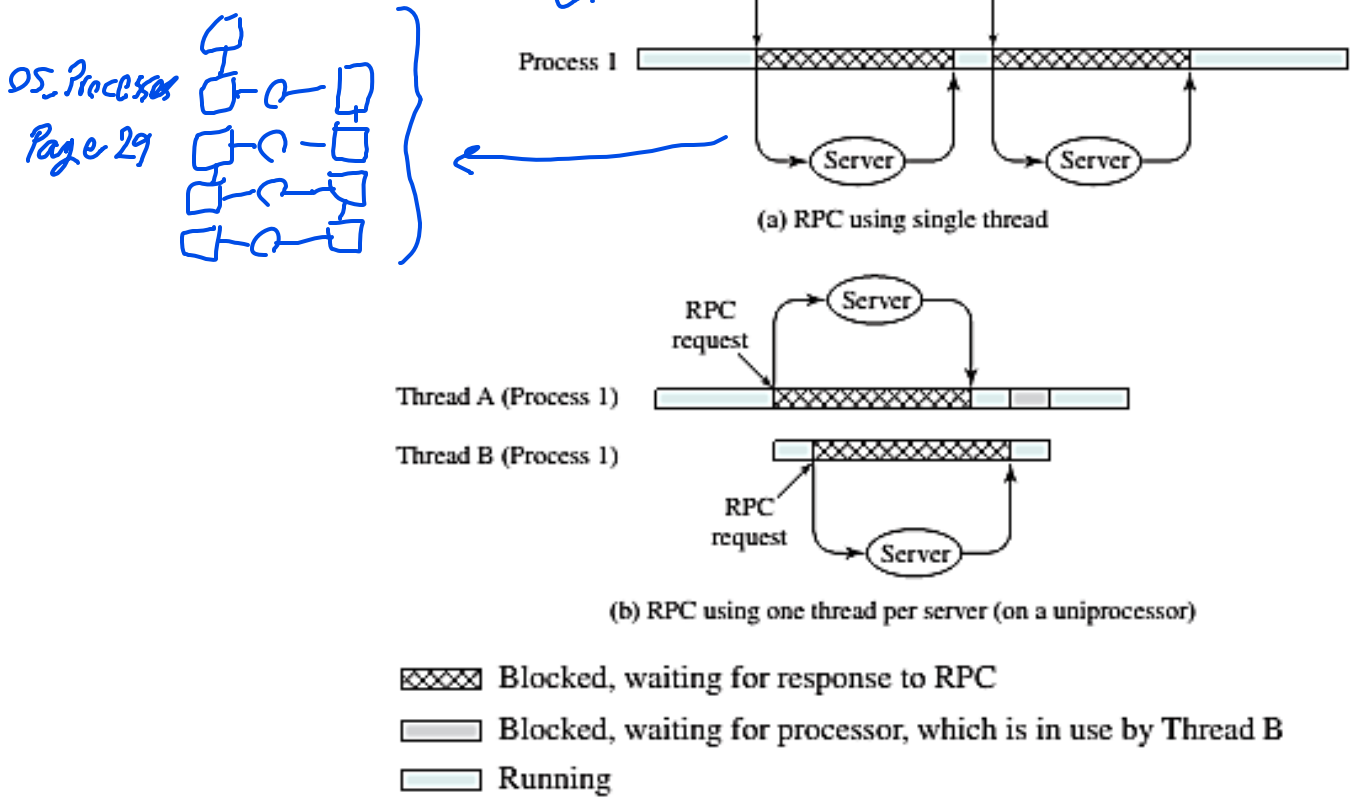
Types of parallelism

- **Types of parallelism**
  - Data parallelism
  - Task parallelism

- **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

RPC using threads





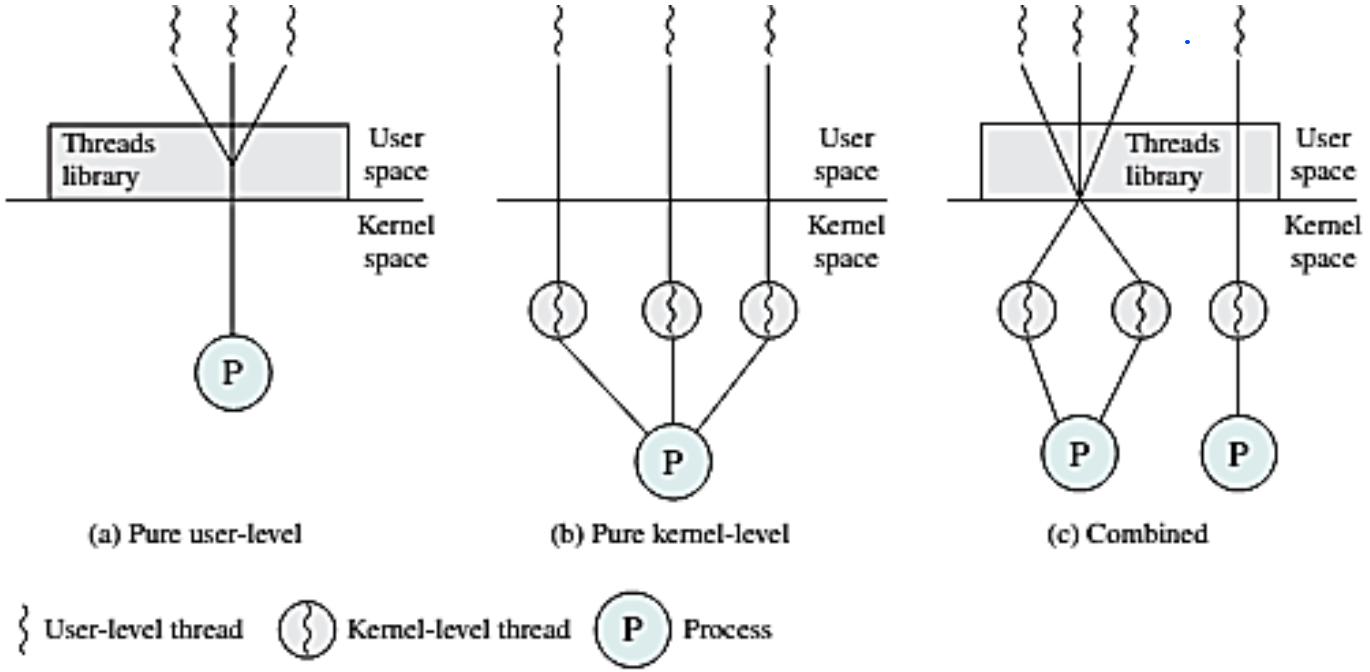
User threads and kernel threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX *Pthreads* (kernel-level lib, user-level lib)
  - Windows threads (kernel-level lib)
  - Java threads (kernel-level lib)

- **Kernel threads** - Supported by the Kernel

- **Asynchronous** vs. **synchronous** threading
  - Parent & child threads

User level vs. kernel level threads



Multithreading models

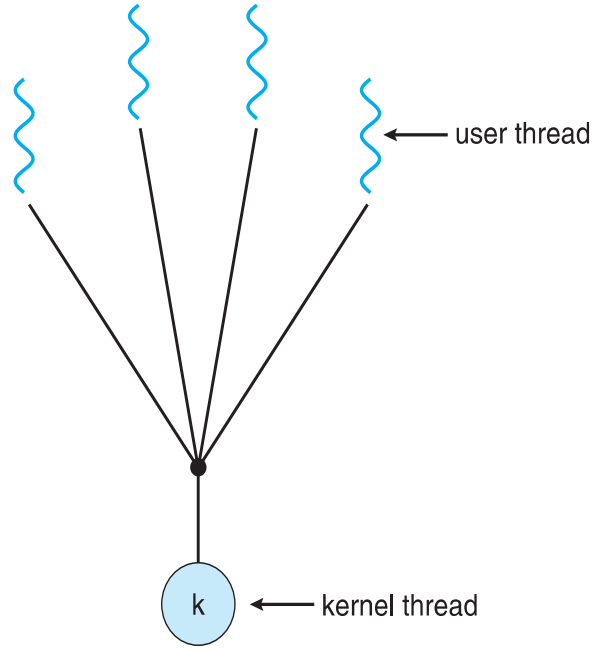
- **Many-to-One**

- **One-to-One**

- **Many-to-Many**

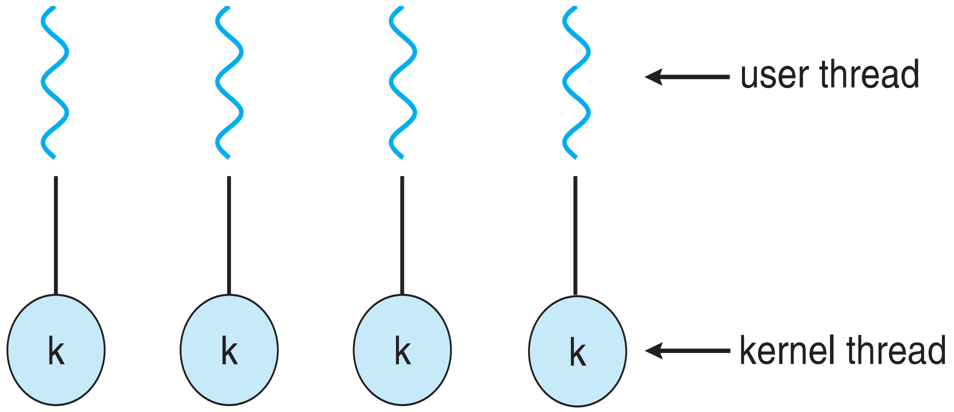
Many-to-one

- 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
- Used in very few systems.



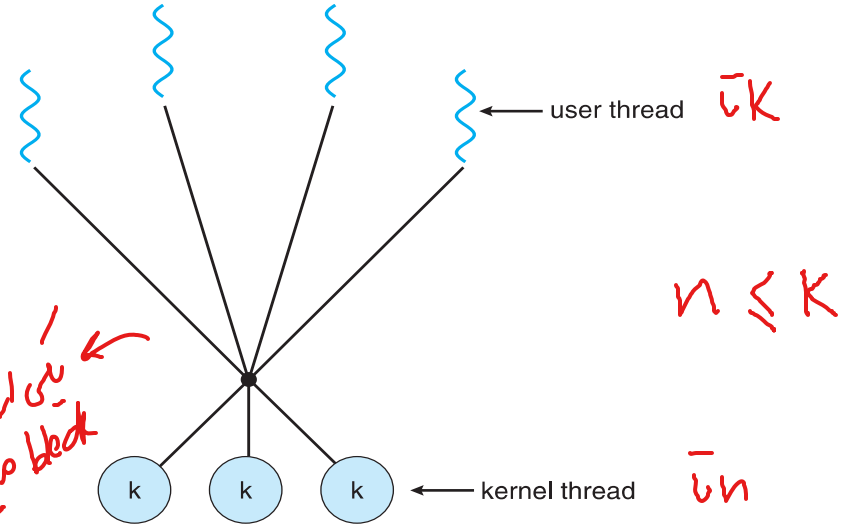
One-to-one

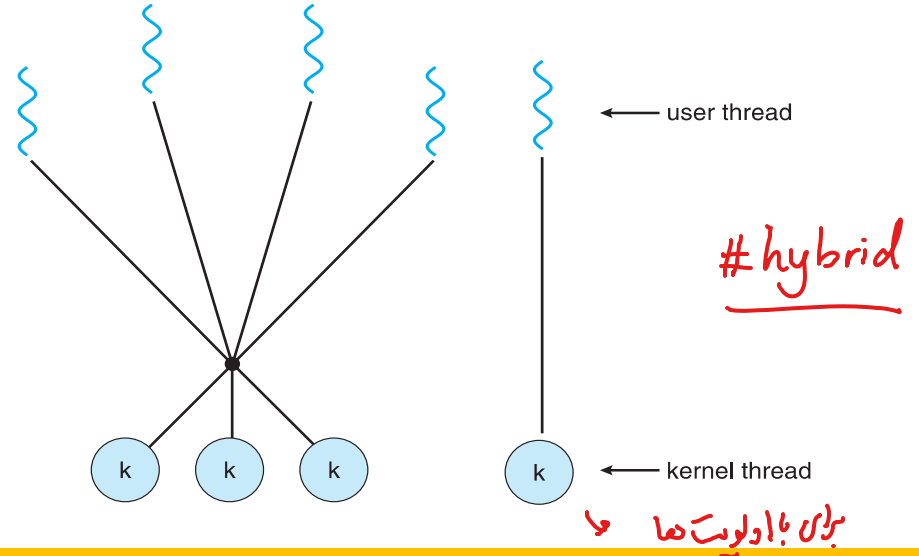
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



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
- Two-level Model:
  - Similar to M:M, except that it allows a user thread to be bound to kernel thread





Thread operations and states

- **Spawn** → تجزیه (new شبه)
  - When a new process is spawned, a thread for that process is also spawned
- **Block** → (waiting شبه)
  - When a thread needs to wait for an event, it will block
- **Unblock** → (running شبه)
  - When the event for which a thread is blocked occurs, the thread is moved to the Ready queue
- **Finish** → (terminate شبه)
  - When a thread completes, its register context and stacks are deallocate

Pthread: POSIX thread

$$sum = \sum_{i=0}^{N} i$$

```
#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);
```

Windows multithread C program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -1;
   }
```

```
   /* create the thread */
   ThreadHandle = CreateThread(
      NULL, /* default security attributes */
      0, /* default stack size */
      Summation, /* thread function */
      &Param, /* parameter to thread function */
      0, /* default creation flags */
      &ThreadId); /* returns the thread identifier */

   if (ThreadHandle != NULL) {
      /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

      /* close the thread handle */
      CloseHandle(ThreadHandle);

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

Java thread programming

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
           thrd.join();
           System.out.println
                   ("The sum of "+upper+" is "+sumObject.getSum());
         } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Implicit threading

- Three methods explored
  - Thread Pools (Win)
  - OpenMP (C lib)
  - Grand Central Dispatch (Mac OS, iOS)

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

Block is in "^{ }" - `^{ printf("I am a block"); }`

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

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

Thread-local storage ≡ Static Variables

- 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

Thread termination

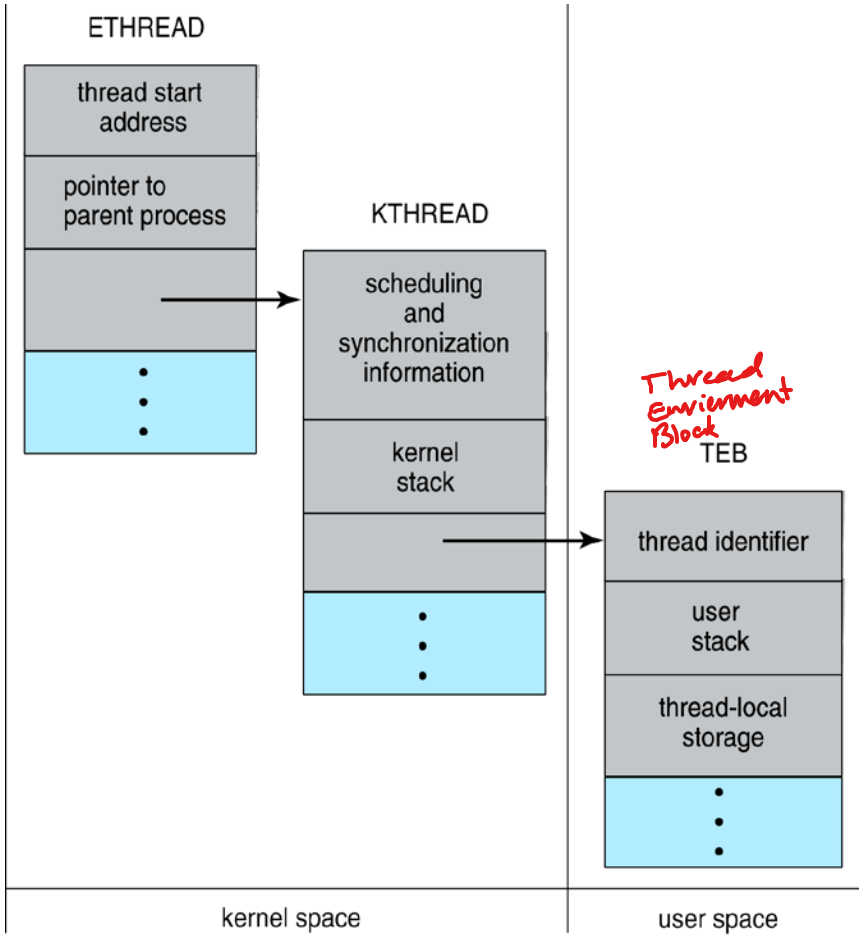
- **Thread cancelation**
  - Asynchronous cancellation
  - Deferred cancellation
- **Who is "target thread"?**

```
pthread_t tid;

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

. . .

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

```
while (1) {
   /* do some work for awhile */
   /* . . . */

   /* check if there is a cancellation request */
   pthread_testcancel();
}
```

Windows threads data structures

- Implements the one-to-one mapping, kernel-level
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the context of the thread
- Data structures:
  - Execution thread block, kernel thread block and thread environment block



جزئیات زیر هم هست.

Linux threads

Linux : { process, thread } → task

- 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.

share { max → thread, min → Process }

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

Questions?