

CS 33

Multithreaded Programming I

Source code used in this lecture is available on the course web page.

Why Threads?

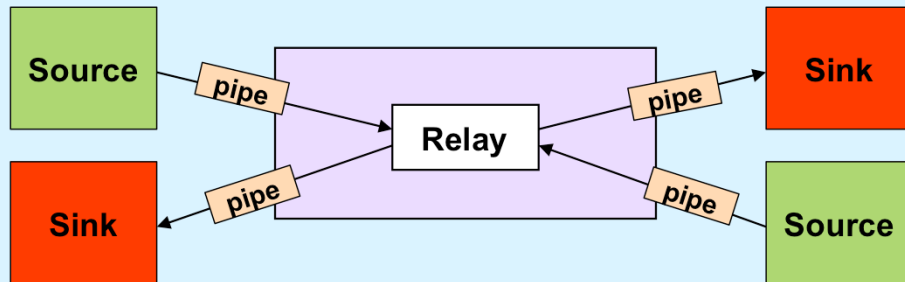


- Many things are easier to do with threads
- Many things run faster with threads

A *thread* is the abstraction of a processor — it is a *thread of control*. We are accustomed to writing single-threaded programs and to having multiple single-threaded programs running on our computers. Why does one want multiple threads running in the same program? Putting it only somewhat over-dramatically, programming with multiple threads is a powerful paradigm.

So, what is so special about this paradigm? Programming with threads is a natural means for dealing with *concurrency*. As we will see, concurrency comes up in numerous situations. A common misconception is that it is a useful concept only on multiprocessors. Threads do allow us to exploit the features of a multiprocessor, but they are equally useful on uniprocessors — in many instances a multithreaded solution to a problem is simpler to write, simpler to understand, and simpler to debug than a single-threaded solution to the same problem.

A Simple Example



For a simple example of a problem that is more easily solved with threads than without, let's look at the stream relay example from a couple lectures ago.

Life Without Threads

```
void relay(int left, int right) {
    fd_set rd, wr;
    int left_read = 1, right_write = 0;
    int right_read = 1, left_write = 0;
    int sizeLR, sizeRL, wret;
    char bufLR[BSIZE], bufRL[BSIZE];
    char *bufpR, *bufpL;
    int maxFD = max(left, right) + 1;

    while(1) {
        FD_ZERO(&rd);
        FD_ZERO(&wr);
        if (left_read)
            FD_SET(left, &rd);
        if (right_read)
            FD_SET(right, &rd);
        if (left_write)
            FD_SET(left, &wr);
        if (right_write)
            FD_SET(right, &wr);

        select(maxFD, &rd, &wr, 0, 0);

        if (FD_ISSET(left, &rd)) {
            sizeLR = read(left, bufLR, BSIZE);
            left_read = 0;
            right_write = 1;
            bufpR = bufLR;
        }
        if (FD_ISSET(right, &rd)) {
            sizeRL = read(right, bufRL, BSIZE);
            right_read = 0;
            left_write = 1;
            bufpL = bufRL;
        }
        if (FD_ISSET(right, &wr)) {
            if ((wret = write(right, bufpR, sizeLR)) == sizeLR) {
                left_read = 1; right_write = 0;
            } else {
                sizeLR -= wret; bufpR += wret;
            }
        }
        if (FD_ISSET(left, &wr)) {
            if ((wret = write(left, bufpL, sizeRL)) == sizeRL) {
                right_read = 1; left_write = 0;
            } else {
                sizeRL -= wret; bufpL += wret;
            }
        }
    }
    return 0;
}
```

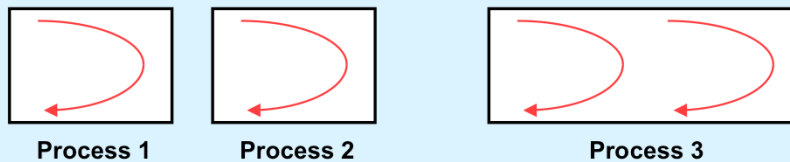
Here's the event-oriented solution we devised earlier that uses *select* (and is rather complicated).

Life With Threads

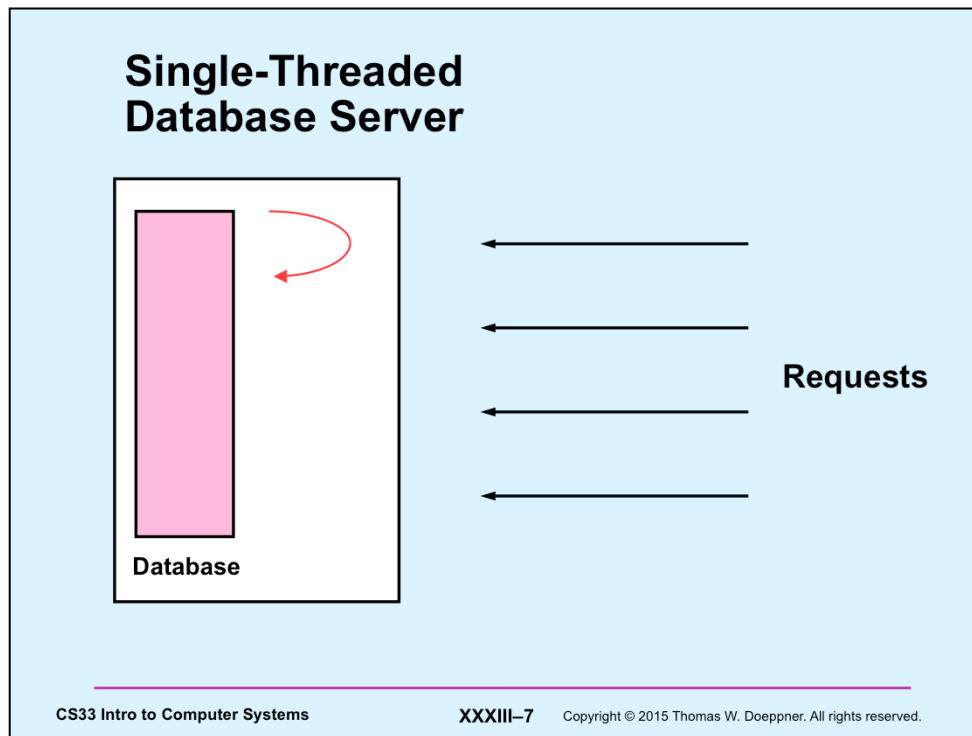
```
void copy(int source, int destination) {  
    struct args *targs = args;  
    char buf[BSIZE];  
  
    while(1) {  
        int len = read(source, buf, BSIZE);  
        write(destination, buf, len);  
    }  
}
```

Here's an essentially equivalent solution that uses threads rather than select. We've left out the code that creates the threads (we'll see that pretty soon), but what's shown is executed by each of two threads. One has source set to the left side and destination to the right side, the other vice versa.

Processes vs. Threads

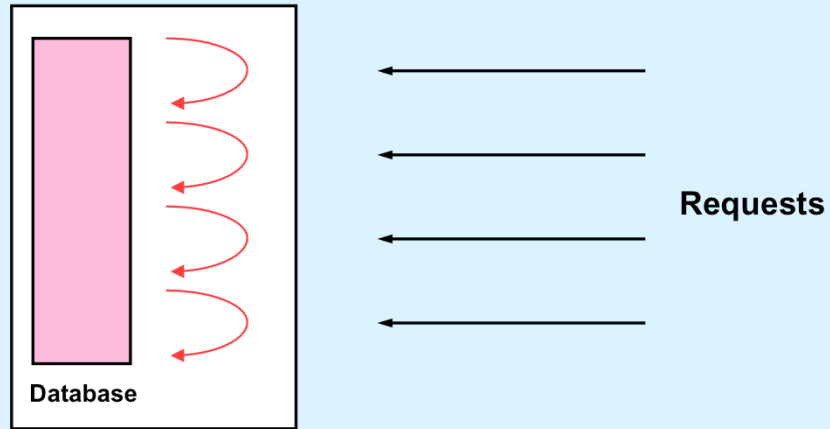


Threads provide concurrency, but so do processes. So, what is the difference between two single-threaded processes and one two-threaded process? First of all, if one process already exists, it is much cheaper to create another thread in the existing process than to create a new process. Switching between the contexts of two threads in the same process is also often cheaper than switching between the contexts of two threads in different processes. Finally, two threads in one process share everything — both address space and open files; the two can communicate without having to copy data. Though two different processes can share memory in modern Unix systems, the most common forms of interprocess communication are far more expensive.



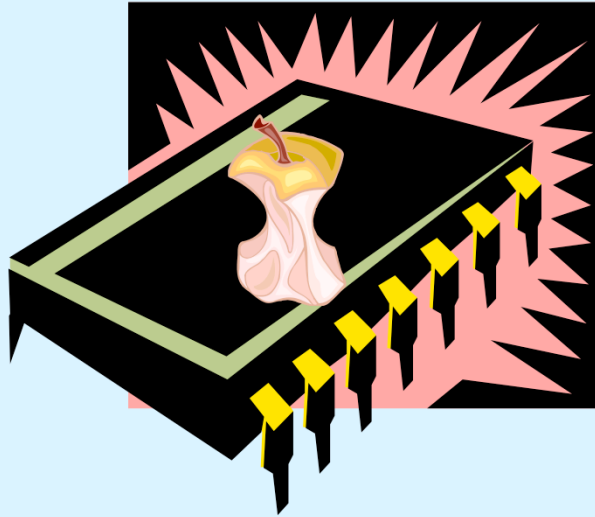
Here is another server example, a database server handling multiple clients. The single-threaded approach to dealing with these requests is to handle them sequentially or to multiplex them explicitly. The former approach would be unfair to quick requests occurring behind lengthy requests, and the latter would require fairly complex and error-prone code.

Multithreaded Database Server

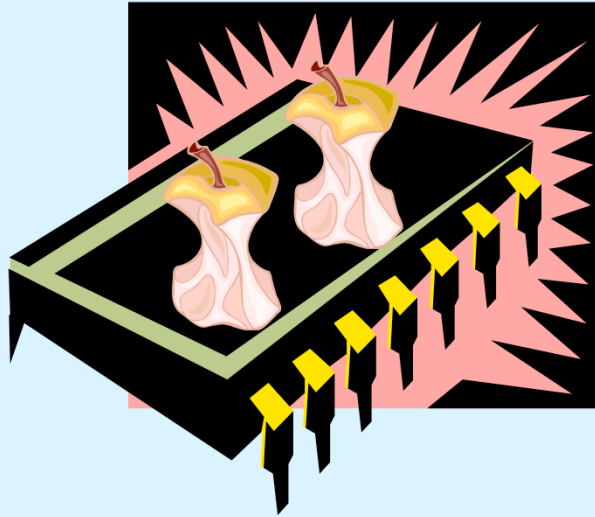


We now rearchitect our server to be multithreaded, assigning a separate thread to each request. The code is as simple as in the sequential approach and as fair as in the multiplexed approach. Some synchronization of access to the database is required, a topic we will discuss soon.

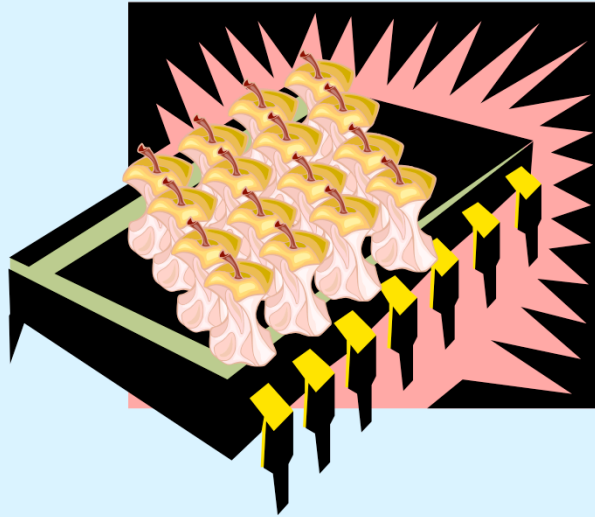
Single-Core Chips



Dual-Core Chips



Multi-Core Chips



Good News/Bad News

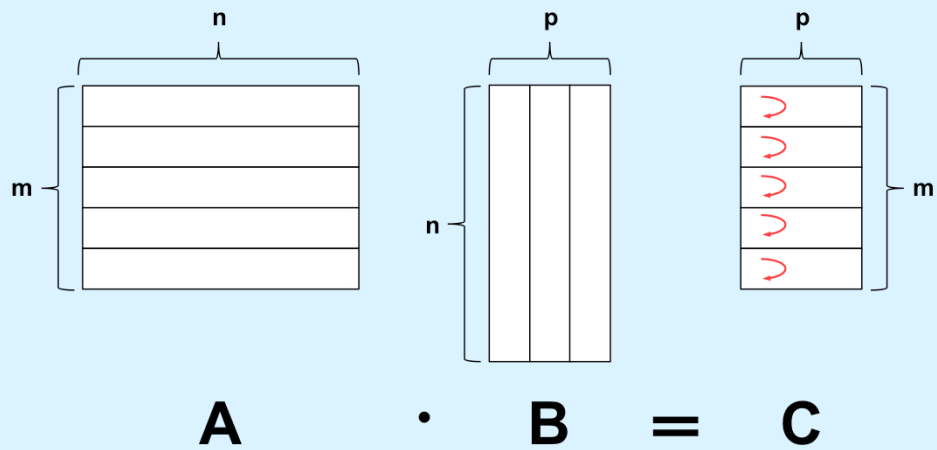
Good news

- multi-threaded programs can take advantage of multi-core chips (single-threaded programs cannot)

Bad news

- it's not easy
 - » must have parallel algorithm
 - employing at least as many threads as processors
 - threads must keep processors busy
 - doing useful work

Matrix Multiplication Revisited



Standards

- **POSIX 1003.4a → 1003.1c → 1003.1j**
- **Microsoft**
 - **Win32**

Despite the advantages of programming with threads, only relatively recently have standard APIs for multithreaded programming been developed. The most important of these APIs, at least in the Unix world, is the one developed by the group known as POSIX 1003.4a. This effort took a number of years and in the summer of '95 resulted in an approved standard, which is now known by the number 1003.1c. In 2000, the POSIX advanced realtime standard, 1003.1j, was approved. It contains a number of new features added to POSIX threads.

Microsoft, characteristically, has produced a threads package whose interface has little in common with those of the Unix world. Moreover, there are significant differences between the Microsoft and POSIX approaches — some of the constructs of one cannot be easily implemented in terms of the constructs of the other, and vice versa. Despite this, both approaches are equally useful for multithreaded programming.

Creating Threads

```
long A[M][N], B[N][P], C[M][P];
...
for (i=0; i<M; i++)    // create worker threads
    pthread_create(&thr[i], 0, matmult, i);

...

void *matmult(void *arg) {
    long i = (long) arg;
    // compute row i of the product C of A and B
    ...
}
```

To create a thread, one calls the *pthread_create* routine. This skeleton code for a server application creates a number of threads, each to handle client requests. If *pthread_create* returns successfully (i.e., returns 0), then a new thread has been created that is now executing independently of the caller. This new thread has an ID that is returned via the first parameter. The second parameter is a pointer to an *attributes structure* that defines various properties of the thread. Usually we can get by with the default properties, which we specify by supplying a null pointer (we discuss this in more detail later). The third parameter is the address of the routine in which our new thread should start its execution. The last parameter is the argument that is actually passed to the first procedure of the thread.

If *pthread_create* fails, it returns a code indicating the cause of the failure.

This example in the slide is a sketch of a multi-threaded matrix multiplication program in which we have one thread per row of the product matrix.

When Is It Done?

```
long A[M][N], B[N][P], C[M][P];
...
for (i=0; i<M; i++)    // create worker threads
    pthread_create(&thr[i], 0, matmult, i);

for (i=0; i<M; i++)    // wait for termination
    pthread_join(thr[i], 0);

printResult(C); // shouldn't do this until
                // workers have terminated
```

We'd like the first thread to be able to print the resulting produce matrix C, but it shouldn't attempt to do this until the worker threads have terminated. We have it call `pthread_join` for each of the worker threads, causing it to wait for each worker to terminate.

Termination

```
pthread_exit((void *) value);  
  
return((void *) value);  
  
pthread_join(thread, (void **) &value);
```

A thread terminates either by calling *pthread_exit* or by returning from its first procedure. In either case, it supplies a value that can be retrieved via a call (by some other thread) to *pthread_join*. The analogy to process termination and the *waitpid* system call in Unix is tempting and is correct to a certain extent — Unix’s *waitpid*, like *pthread_join*, lets one caller synchronize with the termination of another. There is one important difference, however: Unix has the notion of parent/child relationships among processes. A process may wait only for its children to terminate. No such notion of parent/child relationship is maintained with POSIX threads: one thread may wait for the termination of any other thread in the process (though some threads cannot be “joined” by any thread — see the next page). It is, however, important that *pthread_join* be called for each joinable terminated thread — since threads that have terminated but have not yet been joined continue to use up some resources, resources that will be freed once the thread has been joined. The effect of multiple threads calling *pthread_join* is “undefined” — meaning that what happens can vary from one implementation to the next.

One should be careful to distinguish between terminating a thread and terminating a process. With the latter, all the threads in the process are forcibly terminated. So, if *any* thread in a process calls *exit*, the entire process is terminated, along with its threads. Similarly, if a thread returns from *main*, this also terminates the entire process, since returning from *main* is equivalent to calling *exit*. The only thread that can legally return from *main* is the one that called it in the first place. All other threads (those that did not call *main*) certainly do not terminate the entire process when they return from their first procedures, they merely terminate themselves.

If no thread calls *exit* and no thread returns from *main*, then the process should terminate once all threads have terminated (i.e., have called *pthread_exit* or, for threads

Example (1)

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

#define M 3
#define N 4
#define P 5

long A[M][N];
long B[N][P];
long C[M][P];

void *matmult(void *);

main( ) {
    long i;
    pthread_t thr[M];
    int error;

    // initialize the matrices
    ...
}
```

In this series of slide we show the complete matrix-multiplication program.

This slide shows the necessary includes, global declarations, and the beginning of the main routine.

Example (2)

```
for (i=0; i<M; i++) { // create worker threads
    if (error = pthread_create(
        &thr[i],
        0,
        matmult,
        (void *)i)) {
        fprintf(stderr, "pthread_create: %s", strerror(error));
        exit(1);
    }
}

for (i=0; i<M; i++) // wait for workers to finish their jobs
    pthread_join(thr[i], 0)

/* print the results ... */
}
```

Here we have the remainder of *main*. It creates a number of threads, one for each row of the result matrix, waits for all of them to terminate, then prints the results (this last step is not spelled out). Note that we check for errors when calling *pthread_create*. (It is important to check for errors after calls to almost all of the pthread routines, but we normally omit it in the slides for lack of space.) For reasons discussed later, the pthread calls, unlike Unix system calls, do not return -1 if there is an error, but return the error code itself (and return zero on success). However, the text associated with error codes is matched with error codes, just as for Unix-system-call error codes.

So that the first thread is certain that all the other threads have terminated, it must call *pthread_join* on each of them.

Example (3)

```
void *matmult(void *arg) {
    long row = (long)arg;
    long col;
    long i;
    long t;

    for (col=0; col < P; col++) {
        t = 0;
        for (i=0; i<N; i++)
            t += A[row][i] * B[i][col];
        C[row][col] = t;
    }
    return(0);
}
```

Here is the code executed by each of the threads. It's pretty straightforward: it merely computes a row of the result matrix.

Note how the argument is explicitly converted from *void ** to *long*.

Compiling It

```
% gcc -o mat mat.c -pthread
```

Providing the `-pthread` flag to `gcc` is equivalent to providing all the following flags:

- `-lpthread`: include `libpthread.so` — the POSIX threads library
- `-D_REENTRANT`: defines certain things relevant to threads in `stdio.h` — we cover this later.
- `-Dotherstuff`, where “otherstuff” is a variety of flags required to get the current versions of declarations for POSIX threads in `pthread.h`.

Detached Threads

```
start_servers( ) {  
    pthread_t thread;  
    int i;  
  
    for (i=0; i<nr_of_server_threads; i++) {  
        pthread_create(&thread, 0, server, 0);  
        pthread_detach(thread);  
    }  
    ...  
}  
  
void *server(void * arg) {  
    ...  
}
```

If there is no reason to synchronize with the termination of a thread, then it is rather a nuisance to have to call *pthread_join*. Instead, one can arrange for a thread to be *detached*. Such threads “vanish” when they terminate — not only do they not need to be joined, but they cannot be joined.

Complications

```
void relay(int left, int right) {
    pthread_t LRthread, RLthread;

    pthread_create(&LRthread,
        0,
        copy,
        left, right);          // Can't do this ...
    pthread_create(&RLthread,
        0,
        copy,
        right, left);          // Can't do this ...
}
```

An obvious limitation of the *pthread_create* interface is that one can pass only a single argument to the first procedure of the new thread. In this example, we are trying to supply code for the *relay* example, but we run into a problem when we try to pass two parameters to each of the two threads.

Multiple Arguments

```
typedef struct args {
    int src;
    int dest;
} args_t;

void relay(int left, int right) {
    args_t LRargs, RLargs;
    pthread_t LRthread, RLthread;
    ...
    pthread_create(&LRthread, 0, copy, &LRargs);
    pthread_create(&RLthread, 0, copy, &RLargs);
}
```

To pass more than one argument to the first procedure of a thread, we must somehow encode multiple arguments as one. Here we pack two arguments into a structure, then pass the pointer to the structure.

Multiple Argumen

```
typedef struct args {  
    int src;  
    int dest;  
} args_t;
```

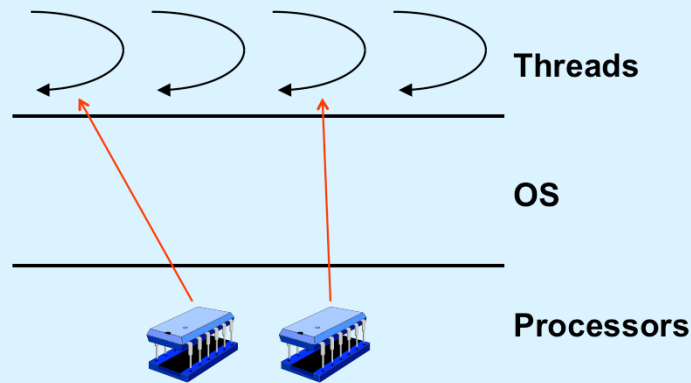
```
void relay(int left, int right) {  
    args_t LRargs, RLargs;  
    pthread_t LRthread, RLthread;  
    ...  
    pthread_create(&LRthread, 0, copy, &LRargs);  
    pthread_create(&RLthread, 0, copy, &RLargs);  
}
```

Quiz 1

Does this work?

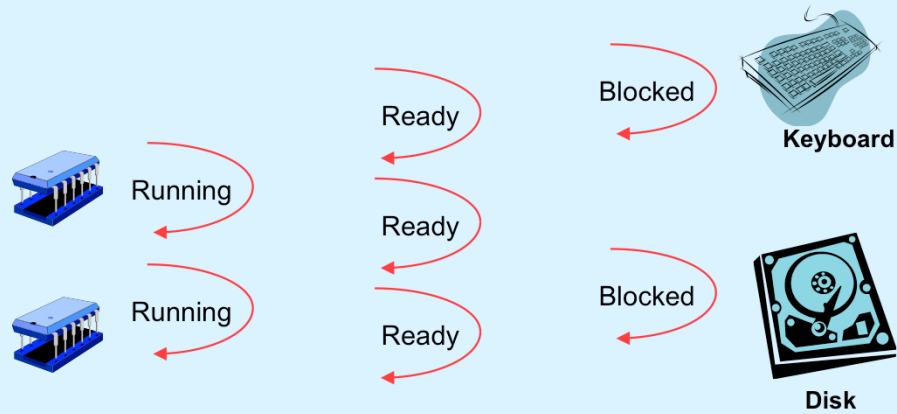
- a) yes
- b) there's a minor problem, easily dealt with
- c) there's a serious problem

Execution



The operating system is responsible for multiplexing the execution of threads on the available processors. The OS's *scheduler* is responsible for assigning threads to processors. Periodically, say every millisecond, each processor is interrupted and calls upon the OS to determine if another thread should run. If so, the current thread on the processor is preempted in favor of the next thread. Assuming all threads are treated equally, over a sufficient period of time each thread gets its fair share of available processor time. Thus, even though a system may have only one processor, all threads make progress and give the appearance of running simultaneously.

Multiplexing Processors



To be a bit more precise about scheduling, let's define some more (standard) terms. Threads are in either a *blocked* state or a *ready* state: in the former they cannot be assigned a processor, in the latter they can. The scheduler determines which ready threads should be assigned processors. Ready threads that have been assigned processors are called *running* threads.

Quiz 2

```
pthread_create(&tid, 0, tproc, (void *)1);  
pthread_create(&tid, 0, tproc, (void *)2);  
  
printf("T0\n");  
  
...  
  
void *tproc(void *arg) {  
    printf("T%d\n", (int)arg);  
    return 0;  
}
```

In which order are things printed?

- a) T0, T1, T2
- b) T1, T2, T0
- c) T2, T1, T0
- d) indeterminate

Cost of Threads

```
int main(int argc, char *argv[]) {  
    ...  
    val = niters/nthreads;  
  
    for (i=0; i<nthreads; i++)  
        pthread_create(&thread, 0, work, (void *)val);  
    pthread_exit(0);  
    return 0;  
}  
  
void *work(void *arg) {  
    long n = (long)arg; int i, j; volatile long x;  
  
    for (i=0; i<n; i++) {  
        x = 0;  
        for (j=0; j<1000; j++)  
            x = x*j;  
    }  
    return 0;  
}
```

Cost of Threads

```
int main(int argc, char *argv[]) {
    ...
    val = niters/nthreads;

    for (i=0; i<nthreads; i++)
        pthread_create(&thread, 0, work, (void *)val);
    pthread_exit(0);
    return 0;
}

void *work(void *arg) {
    long n = (long)arg; int i, j; volatile long x;

    for (i=0; i<n; i++) {
        x = 0;
        for (j=0; j<1000; j++)
            x = x*j;
    }
    return 0;
}
```

Quiz 3

This code runs in time n on a 4-core processor when $nthreads$ is 4. It runs in time p on the same processor when $nthreads$ is 400.

- a) $n \ll p$ (slower)
- b) $n \approx p$ (same speed)
- c) $n \gg p$ (faster)

Thread Attributes

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);

...

/* establish some attributes */

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

A number of properties of a thread can be specified via the *attributes* argument when the thread is created. Some of these properties are specified as part of the POSIX specification, others are left up to the implementation. By burying them inside the attributes structure, we make it straightforward to add new types of properties to threads without having to complicate the parameter list of *pthread_create*. To set up an attributes structure, one must call *pthread_attr_init*. As seen in the next slide, one then specifies certain properties, or attributes, of threads. One can then use the attributes structure as an argument to the creation of any number of threads.

Note that the attributes structure only affects the thread when it is created. Modifying an attributes structure has no effect on already-created threads, but only on threads created subsequently with this structure as the attributes argument.

Storage may be allocated as a side effect of calling *pthread_attr_init*. To ensure that it is freed, call *pthread_attr_destroy* with the attributes structure as argument. Note that if the attributes structure goes out of scope, not all storage associated with it is necessarily released — to release this storage you must call *pthread_attr_destroy*.

Stack Size

```
pthread_t thread;
pthread_attr_t thr_attr;

pthread_attr_init(&thr_attr);
pthread_attr_setstacksize(&thr_attr, 20*1024*1024);

...

pthread_create(&thread, &thr_attr, startroutine, arg);

...
```

Among the attributes that can be specified is a thread's *stack size*. The default attributes structure specifies a stack size that is probably good enough for “most” applications. How big is it? While the default stack size is not mandated by POSIX, in Linux it is two megabytes. To establish a different stack size, use the `pthread_attr_setstacksize` routine, as shown in the slide.

How large a stack is necessary? The answer, of course, is that it depends. If the stack size is too small, there is the danger that a thread will attempt to overwrite the end of its stack. There is no problem with specifying too large a stack, except that, on a 32-bit machine, one should be careful about using up too much address space (one thousand threads, each with a one-megabyte stack, use a fair portion of the address space).