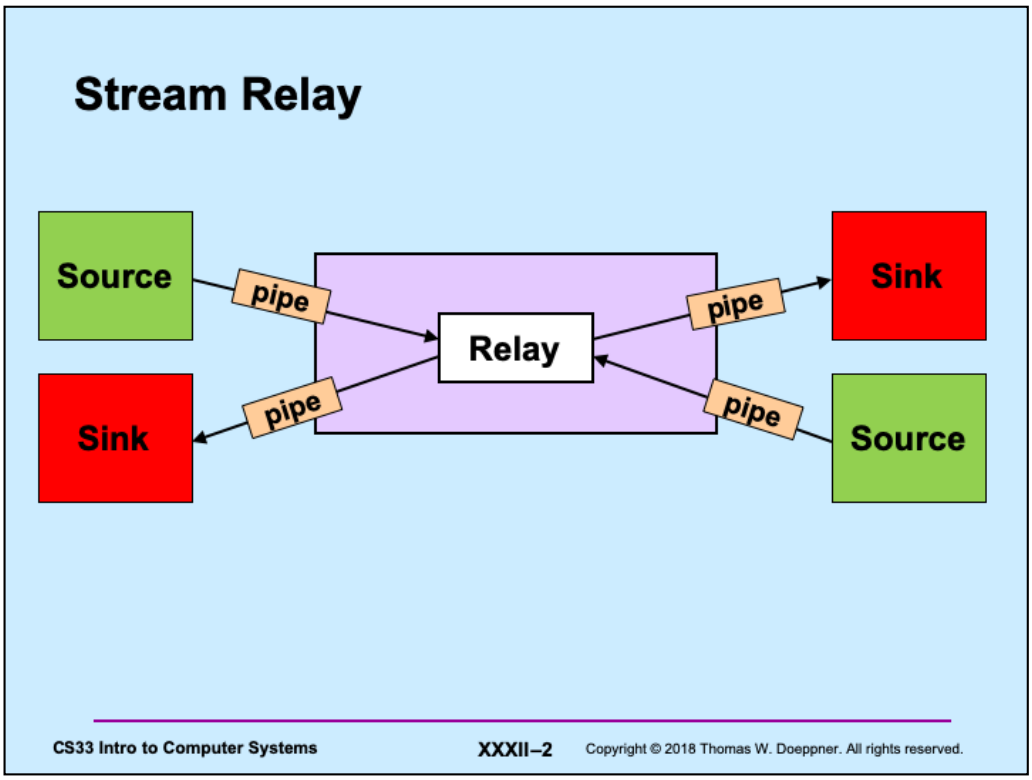
CS 33

Event-Based Programming
Multithreaded Programming I



We start by looking at what's known as *event-based programming*: we write code that responds to events coming from a number of sources. As a simple example, before we use the approach in a networking example, we examine a simple relay: we want to write a program that takes data, via a pipe, from the left source and sends it, via a pipe, to the right sink. At the same time it takes data from the right source and sends it to the left sink.

Select System Call

```
int select(
  int nfds,          // size of fd_sets
  fd_set *readfds,   // descriptors of interest
                     // for reading
  fd_set *writefds,  // descriptors of interest
                     // for writing
  fd_set *excpfds,   // descriptors of interest
                     // for exceptional events
  struct timeval *timeout
                     // max time to wait
);
```


The select system call operates on three sets of file descriptors: one of fie descriptors we're interested in reading from, one of file descriptors we're interested in writing to, and one of file descriptors that might have exceptional conditions pending (we haven't covered any examples of such things – they come up as a peculiar feature of TCP known as out-of-band data, which is beyond the scope of this course). A call to select waits until at least one of the file descriptors in the given sets has something of interest. In particular, for a file descriptor in the read set, it's possible to read data from it; for a file descriptor in the write set, it's possible to write data to it. The nfds parameter indicates the maximum file descriptor number in any of the sets. The timeout parameter may be used to limit how long select waits. If set to zero, select waits indefinitely.

Relay Sketch

```
void relay(int left, int right) {
   fd_set rd, wr;
   int maxFD = max(left, right) + 1;
   FD_ZERO(&rd); FD_SET(left, &rd); FD_SET(right, &rd);
   FD_ZERO(&wr); FD_SET(left, &wr); FD_SET(right, &wr);
   while (1) {
      select(maxFD, &rd, &wr, 0, 0);
      if (FD_ISSET(left, &rd))
         read(left, bufLR, BSIZE);
      if (FD_ISSET(right, &rd))
         read(right, bufRL, BSIZE);
      if (FD_ISSET(right, &wr))
         write(right, bufLR, BSIZE);
      if (FD_ISSET(left, &rd))
         write(left, bufRL, BSIZE);
   }
}
```


Here a simplified version of a program to handle the relay problem using *select*. An *fd_set* is a data type that represents a set of file descriptors. FD_ZERO, FD_SET, and FD_ISSET are macros for working with fd_sets; the first makes such a set represent the null set, the second sets a particular file descriptor to included in the set, the last checks to see if a particular file descriptor is included in the set.

Relay (1)

```
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;
```


This and the next three slides give a more complete version of the relay program.

Initially our program is prepared to read from either the left or the right side, but it's not prepared to write, since it doesn't have anything to write. The variables *left_read* and *right_read* are set to one to indicate that we want to read from the left and right sides. The variables *right_write* and *left_write* are set to zero to indicate that we don't yet want to write to either side.

Relay (2)

```
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);
```


We set up the fd_sets *rd* and *ur* to indicate what we are interested in reading from and writing to (initially we have no interest in writing, but are interested in reading from either side).

Relay (3)

```
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 there is something to read from the left side, we read it. Having read it, we're temporarily not interested in reading anything further from the left side, but now want to write to the right side.

In a similar fashion, if there is something to read from the right side, we read it.

Relay (4)

```
    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;
}
```


Writing is a bit more complicated, since the outgoing pipe might not have room for everything we have to write, but just some of it. Thus we must pay attention to what write returns. If everything has been written, then we can go back to reading from the other side, but if not, we continue trying to write.

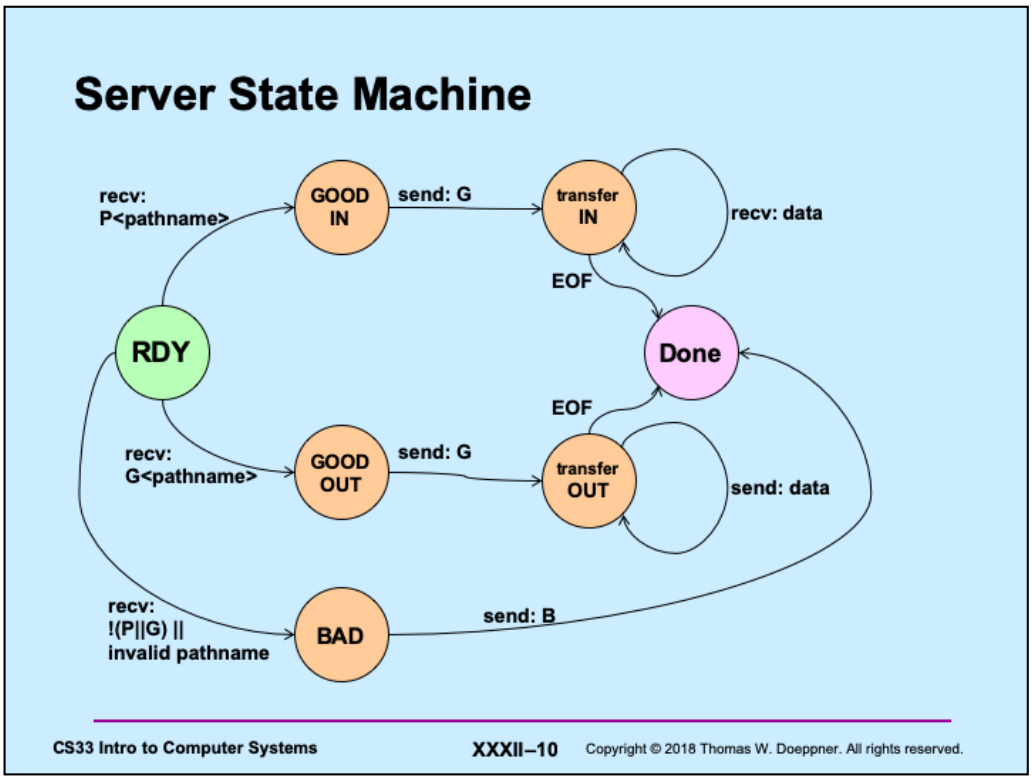
A Really Simple Protocol

- **Transfer a file**
  - **layered on top of TCP**
    - » **reliable**
    - » **indicates if connection is closed**
- **To send a file**

  P<null-terminated pathname><contents of file>
- **To retrieve a file**

  G<null-terminated pathname>

Now we use the event paradigm to implement a simple file-transfer program.



We design the protocol in terms of a simple state machine. The server will be dealing concurrently with many clients and will maintain a state machine for each client.

Keeping Track of State

```
typedef struct client {
  int fd;     // file descriptor of local file being transferred
  int size;   // size of out-going data in buffer
  char buf[BSIZE];
  enum state {RDY, BAD, GOOD, TRANSFER} state;
  /*
     states:
        RDY: ready to receive client's command (P or G)
        BAD: client's command was bad, sending B response + error msg
        GOOD: client's command was good, sending G response
        TRANSFER: transferring data
    */
  enum dir {IN, OUT} dir;
  /*
     IN: client has issued P command
     OUT: client has issued G command
   */
} client_t;
```


Note the use of the *enum* data type. Variables of this type have a finite set of possible values, as given in the declaration.

Rather than have separate GOOD-IN, GOOD-OUT, TRANSFER-IN, and TRANSFER-OUT states, we have two state variables, one which keeps track of the direction.

Keeping Track of Clients

```
client_t clients[MAX_CLIENTS];
for (i=0; i < MAX_CLIENTS; i++)
  clients[i].fd = -1; // illegal value

listen(lsock, max_queue_len);
fd_set rd, wr;
FD_ZERO(&rd);
FD_SET(lsock, &rd);
FD_ZERO(&wr);

fd_set trd = rd;
fd_set twr = wr;
```


Each client of our server is represented by a separate *client_t* structure. We allocate an array of them and will refer to client's *client_t* structure by the file descriptor of the socket used to communicate with it. Thus if we're using a socket whose file descriptor is *sfd* to communicate with a client, then the client's state is in *clients[sfd]*.

We set up the sets of read and write file descriptors we'll be using with select. Initially, the only file descriptor of interest is lsock, which we put in listening mode. We can then use *select* to determine if it's possible to issue an *accept* system call on lsock without having to wait.

Main Server Loop

```
while(1) {
  select(maxfd, &trd, &twr, 0, 0);
  if (FD_ISSET(lsock, &trd)) {
    // a new connection
    new_client(lsock);
  }
  for (i=lsock+1; i<maxfd; i++) {
    if (FD_ISSET(i, &trd)) {
      // ready to read
      read_event(i);
    }
    if (FD_ISSET(i, &twr)) {
      // ready to write
      write_event(i);
    }
  }
  trd = rd; twr = wr;
}
```


lsock is the file descriptor for the "listening-mode" socket on which the server is waiting for connections. Our server may be handling multiple clients; each will be communicating with the server via a separate connected socket. These sockets have file descriptors greater than *lsock*. Note that *trd*, *twr*, *rd* and *wr* are all of type *fd_set*. *rd* and *wr* are initialized so that *rd* contains just the file descriptor for the listening socket and *wr* is empty. *trd* and *twr* are copied from *rd* and *wr* respectively before the loop is entered. *rd* and *wr* are global variables that are modified by *new_client*, *read_event*, and *write_event*.

New Client

```
// Accept a new connection on listening socket
// fd. Return the connected file descriptor

int new_client(int fd) {
  int cfd = accept(fd, 0, 0);
  clients[cfd].state = RDY;
  FD_SET(cfd, &rd);
  return cfd;
}
```


When the server gets a new client, the state machine for that client is initialized in the RDY state, and the file descriptor for the socket used to communicate with the client is added to the set of read file descriptors.

Read Event (1)

```
// File descriptor fd is ready to be read. Read it, then handle
// the input
void read_event(int fd) {
  client_t *c = &clients[fd];
  int ret = read(fd, c->buf, BSIZE);
  switch (c->state) {
  case RDY:
    if (c->buf[0] == 'G') {
      // GET request (to fetch a file)
      c->dir = OUT;
      if ((c->fd = open(&c->buf[1], O_RDONLY)) == -1) {
        // open failed; send negative response and error message
        c->state = BAD;
        c->buf[0] = 'B';
        strncpy(&c->buf[1], strerror(errno), BSIZE-2);
        c->buf[BSIZE-1] = 0;
        c->size = strlen(c->buf)+1;
      }
```


When the server gets a read event, the file descriptor causing it is used to identify the client. What happens next depends on the state of the client.

Read Event (2)

```
    else {
      // open succeeded; send positive response
      c->state = GOOD;
      c->size = 1;
      c->buf[0] = 'G';
    }
    // prepare to send response to client
    FD_SET(fd, &wr);
    FD_CLR(fd, &rd);
    break;
  }
```


Read Event (3)

```
    if (c->buf[0] == 'P') {
      // PUT request (to create a file)
      c->dir = IN;
      if ((c->fd = open(&c->buf[1],
          O_RDWR|O_CREAT|O_TRUNC, 0666)) == -1) {
          // open failed; send negative response and error message
          ...
    } else {
        // open succeeded; send positive response
        ...
    }
    // prepare to send response to client
    FD_SET(fd, &wr);
    FD_CLR(fd, &rd);
    break;
  }
```


Read Event (4)

```
case TRANSFER:
  // should be in midst of receiving file contents from client
  if (ret == 0) {
    // eof: all done
    close(c->fd);
    close(fd);
    FD_CLR(fd, &rd);
    break;
  }
  if (write(c->fd, c->buf, ret) == -1) {
    // write to file failed: terminate connection to client
    ...
    break;
  }
  // continue to read more data from client
  break;
}
```


Write Event (1)

```
// File descriptor fd is ready to be written to. Write to it, then,
// depending on current state, prepare for the next action.
void write_event(int fd) {
  client_t *c = &clients[fd];
  int ret = write(fd, c->buf, c->size);
  if (ret == -1) {
    // couldn't write to client; terminate connection
    close(c->fd);
    close(fd);
    FD_CLR(fd, &wr);
    c->fd = -1;
    perror("write to client");
    return;
  }
  switch (c->state) {
```


As with handling a read event, when the server has a write event, it uses the file descriptor to determine which client to write to. (A write event indicates that it's now possible to write to the client.)

Write Event (2)

```
case BAD:
  // finished sending error message; now terminate client connection
  close(c->fd);
  close(fd);
  FD_CLR(fd, &wr);
  c->fd = -1;
  break;
```


Write Event (3)

```
case GOOD:
  c->state = TRANSFER;
  if (c->dir == IN) {
    // finished response to PUT request
    FD_SET(fd, &rd);
    FD_CLR(fd, &wr);
    break;
  }
  // otherwise finished response to GET request, so proceed
  // to read file and start transfer out
  // fd should remain in wr
```

Write Event (4)

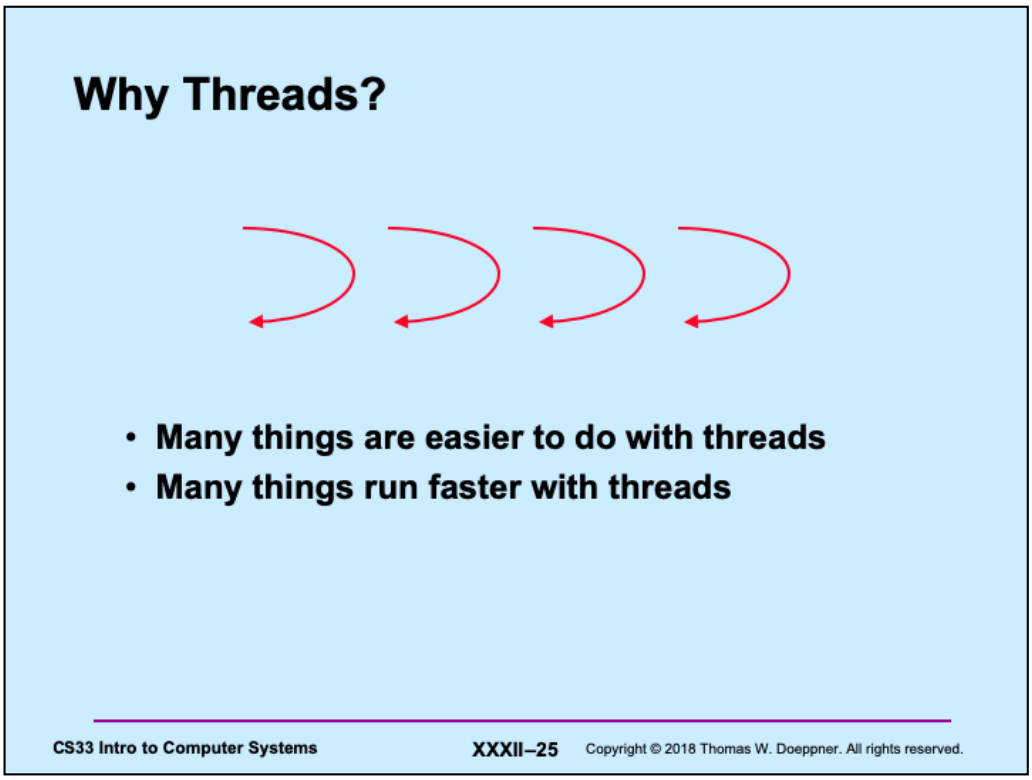
```
  case TRANSFER:
    // should be in midst of transferring file contents to client
    if ((c->size = read(c->fd, c->buf, BSIZE)) == -1) {
      ...
      break;
    } else if (c->size == 0) {
      // no more file to transfer; terminate client connection
      close(c->fd);
      close(fd);
      FD_CLR(fd, &wr);
      c->fd = -1;
      break;
    }
    // continue to write more data to client
    break;
  }
}
```


Problems

- Works fine as long as the protocol is followed correctly
  - can client (malicious or incompetent) cause server to misbehave?
- How can the server limit the number of clients?
- How does server limit file access?

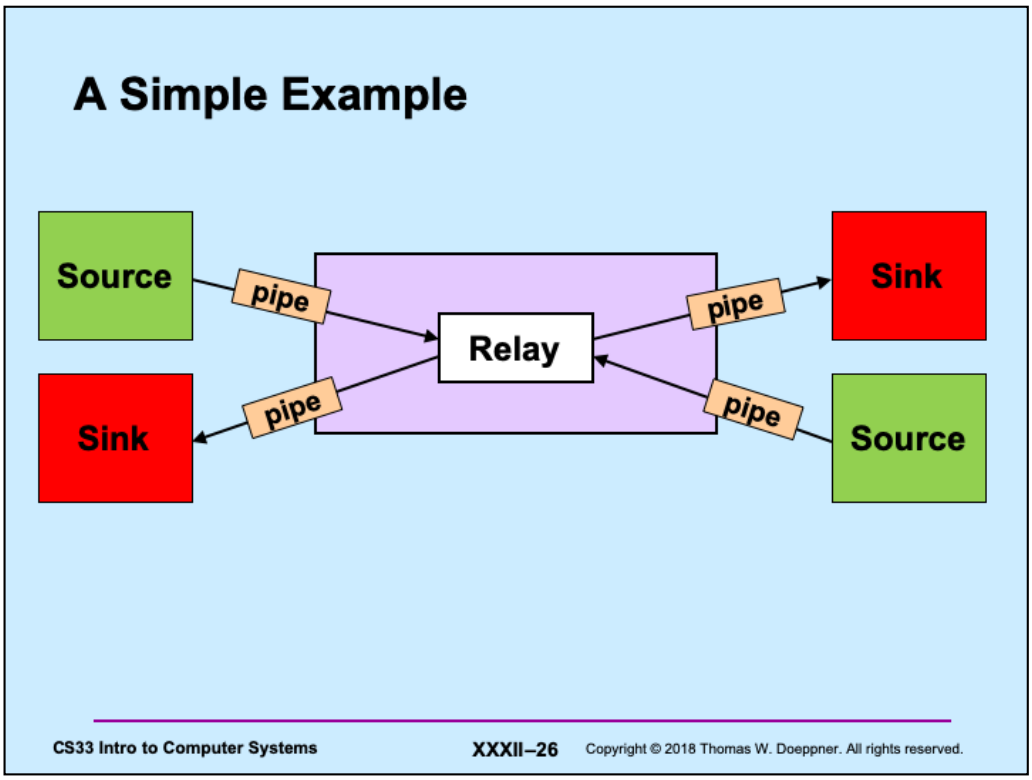
Multithreaded Programming

- A thread is a virtual processor
  - an independent agent executing instructions
- Multiple threads
  - multiple independent agents executing instructions



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.



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 the previous lecture.

Life Without Threads

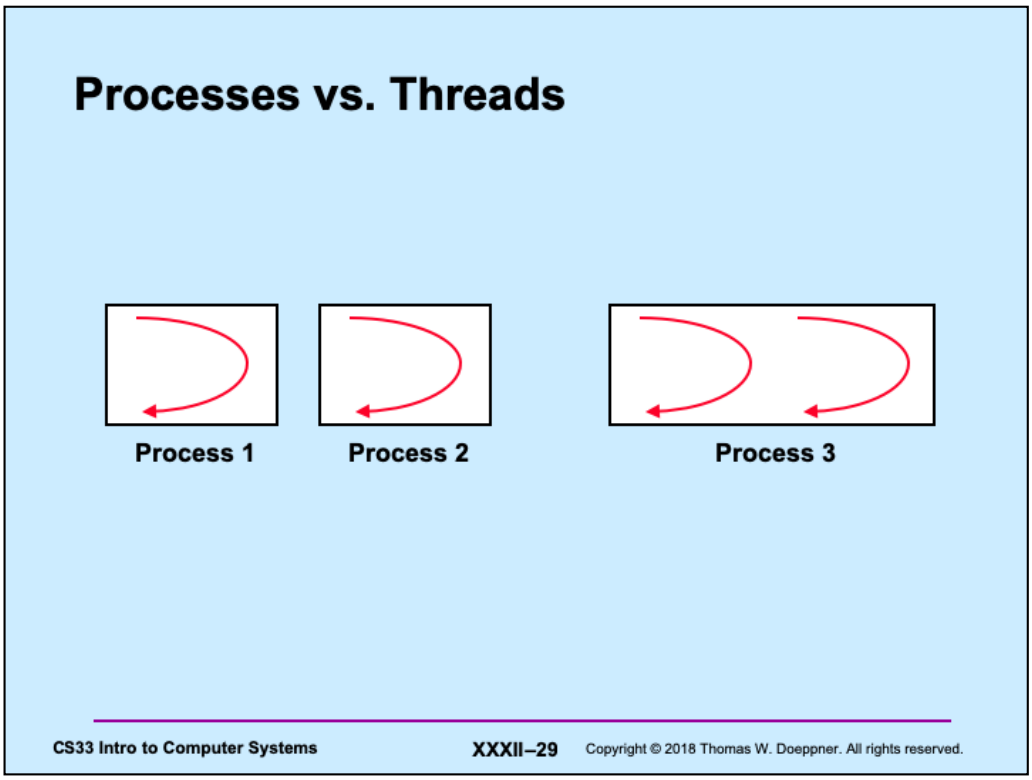
```
void relay(int left, int right) {
    fd_set rd, wr;
    int left_read = 1, right_write = 0;
    int right_read = 1, left_write = 0;
    int sizeLR, sizeRL, wret;
    char bufLR[BSIZE], bufRL[BSIZE];
    char *bufpR, *bufpL;
    int maxFD = max(left, right) + 1;

    fcntl(left, F_SETFL, O_NONBLOCK);
    fcntl(right, F_SETFL, O_NONBLOCK);

    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).

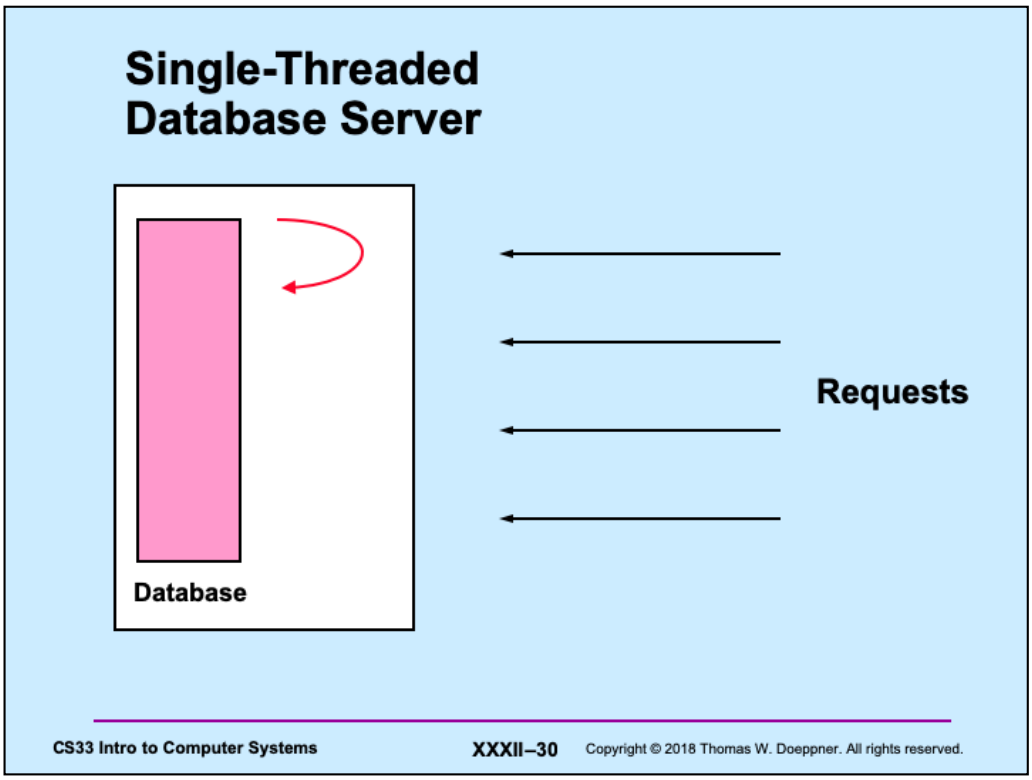
Life With Threads

```
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);
 }
}
```

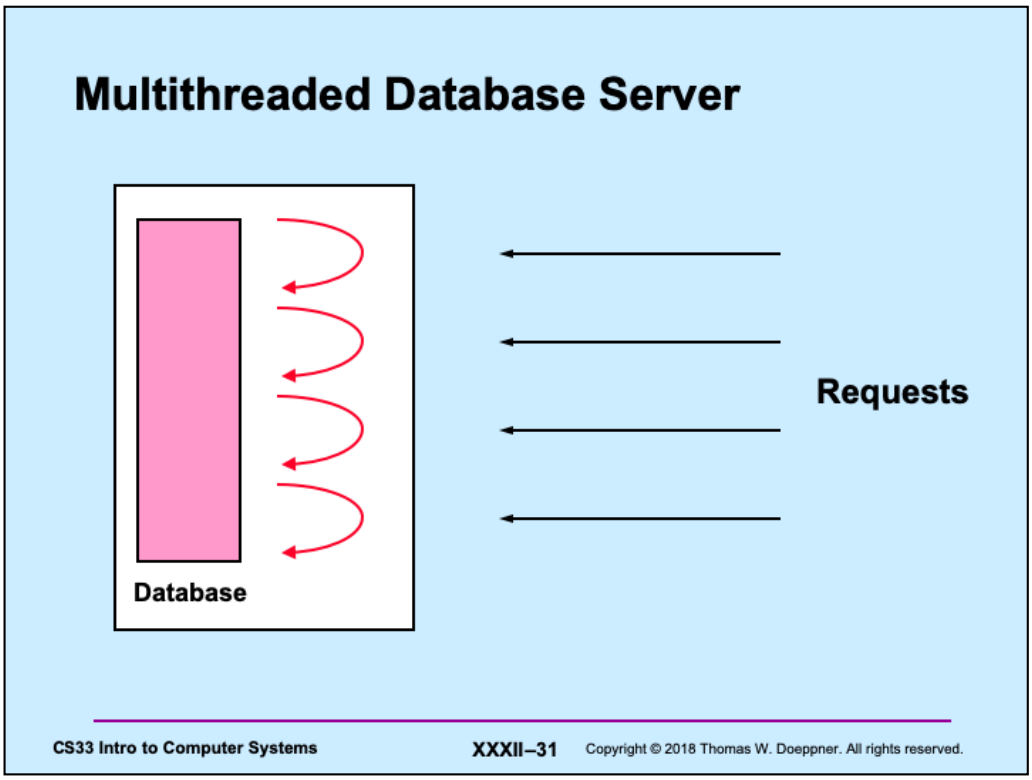

Here's an essentially equivalent solution that uses threads rather than select. We've left out the code that creates the threads (we'll see that pretty soon), but what's shown is executed by each of two threads. One has source set to the left side and destination to the right side, the other vice versa.



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.

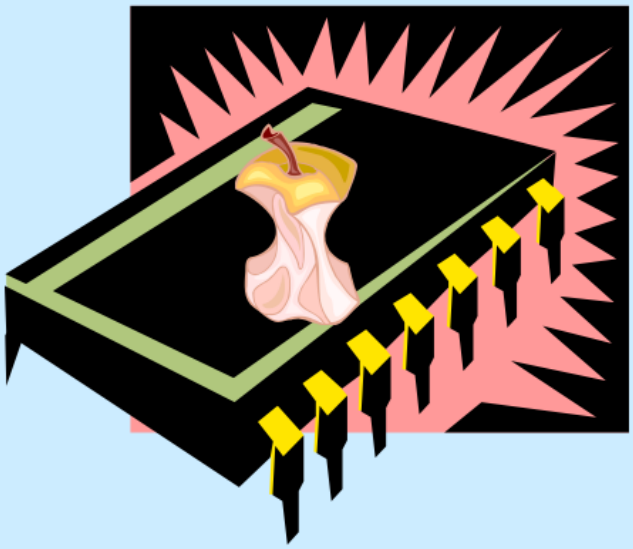


Here is another server example, a database server handling multiple clients. The single-threaded approach to dealing with these requests is to handle them sequentially or to multiplex them explicitly. The former approach would be unfair to quick requests occurring behind lengthy requests, and the latter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

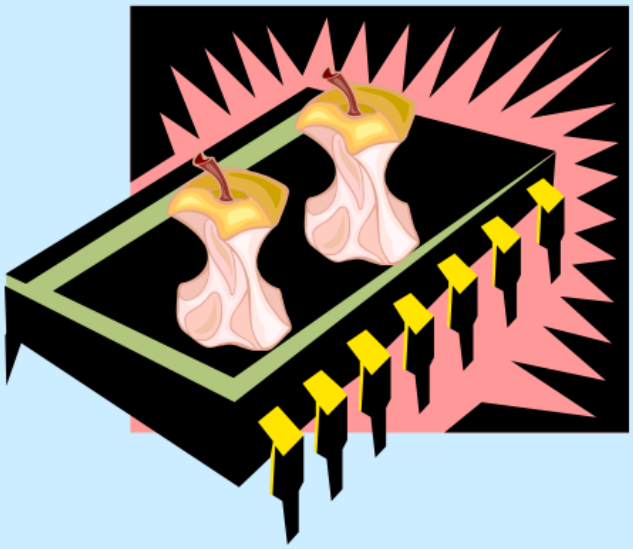


We now rearchitect our server to be multithreaded, assigning a separate thread to each request. The code is as simple as in the sequential approach and as fair as in the multiplexed approach. Some synchronization of access to the database is required, a topic we will discuss soon.

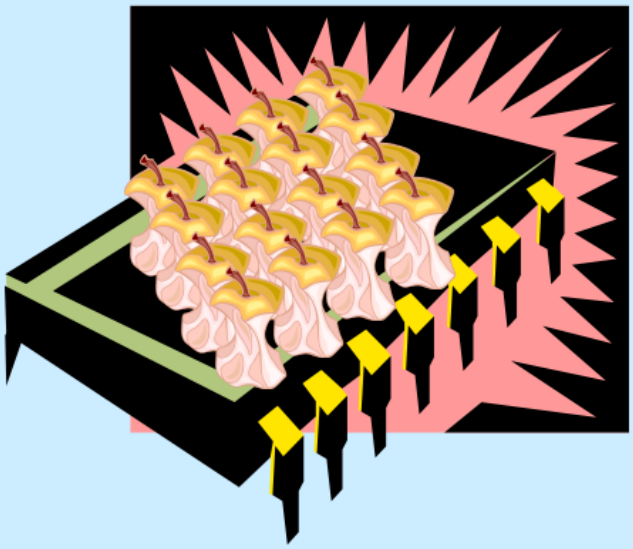
Single-Core Chips



Dual-Core Chips



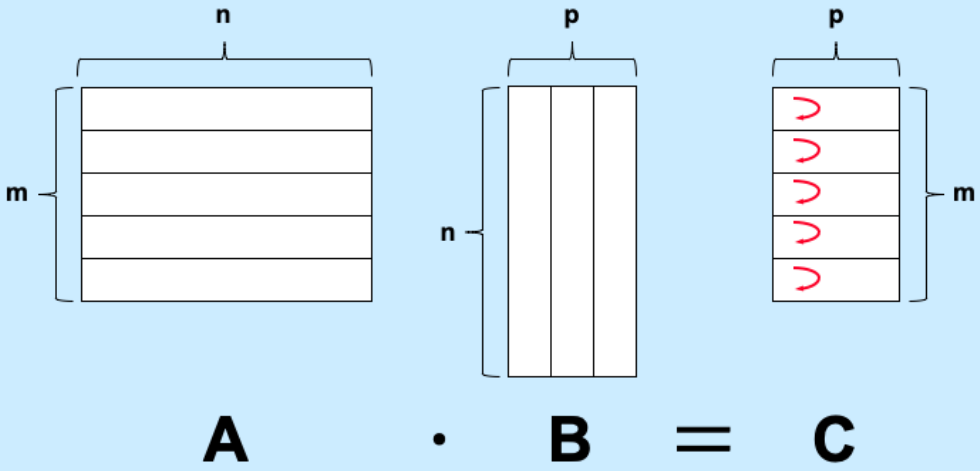
Multi-Core Chips

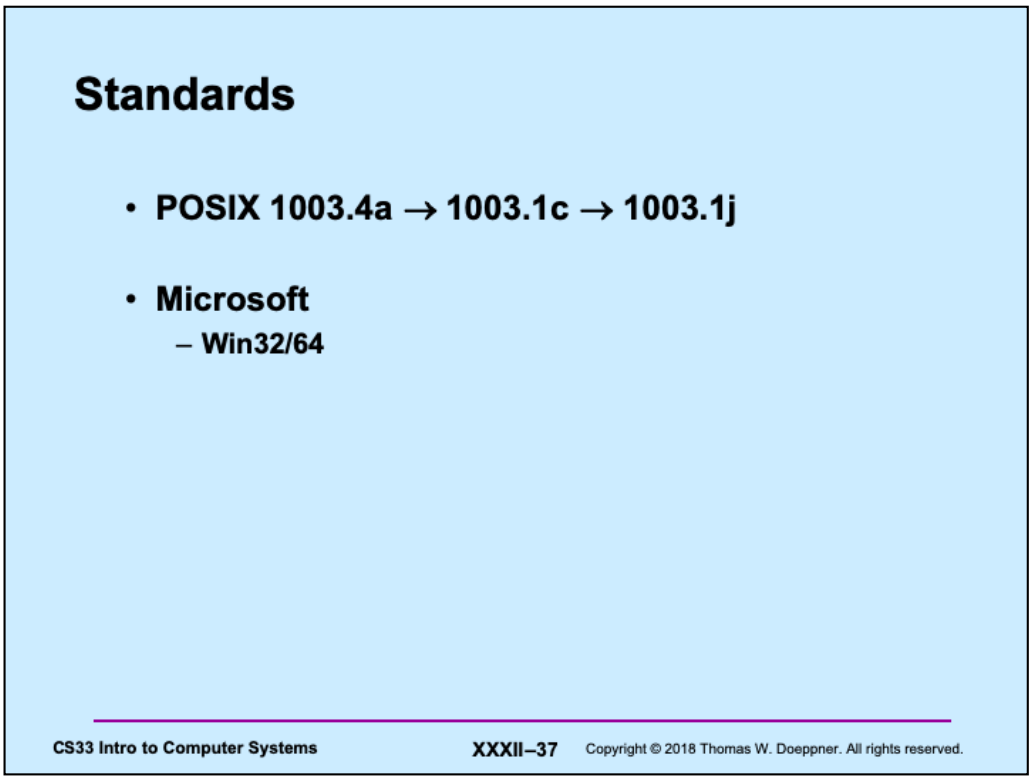


Good News/Bad News

- ☺ 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





Despite the long-known 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 additional 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.

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.

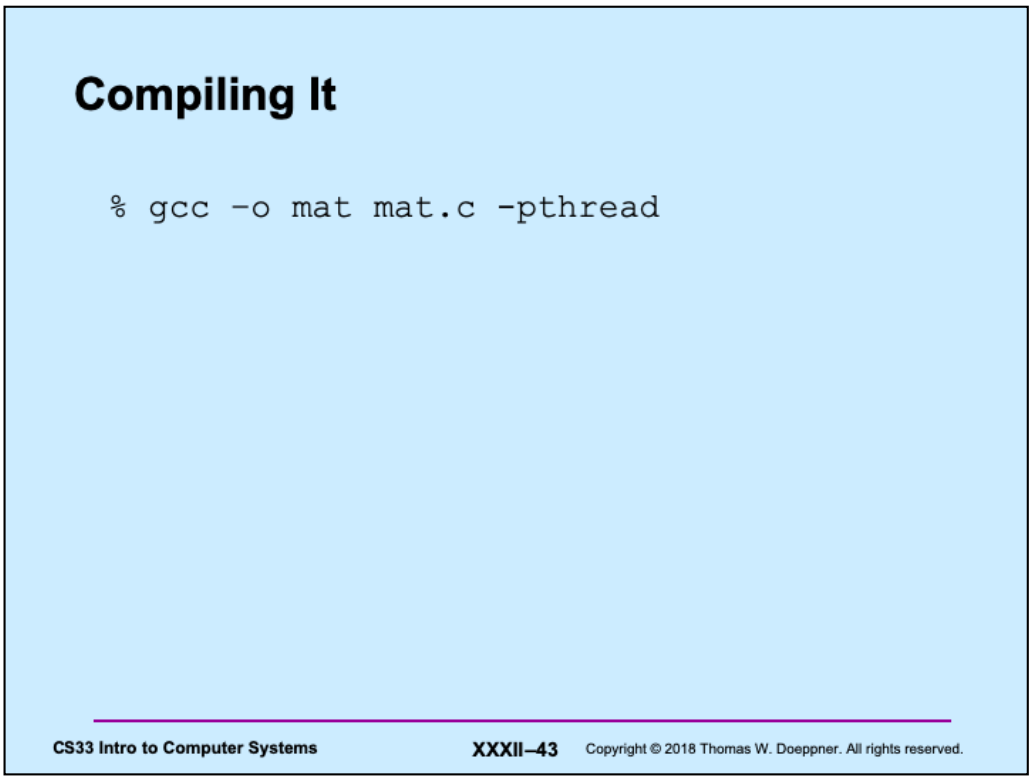
Example (3)

```
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.

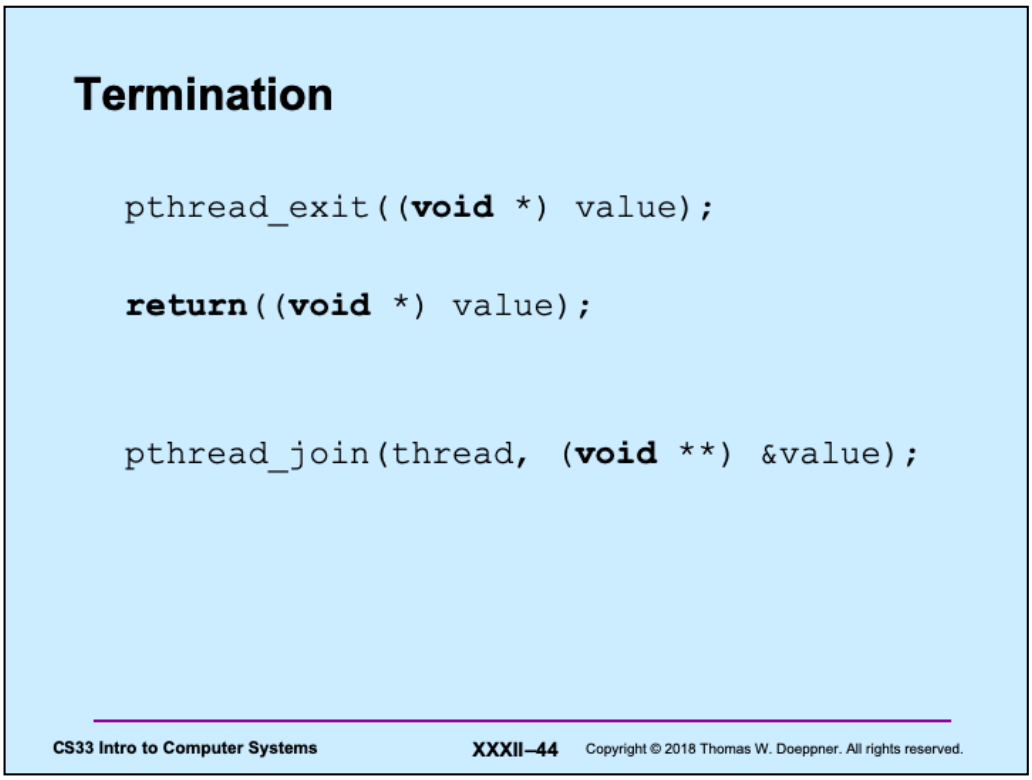
Note how the argument is explicitly converted from *void** to *long*.

This code does not make optimal use of the cache. How can it be restructured so it does?



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.



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

other than the first one, have returned from their first procedure). If the first thread calls *pthread_exit*, it self-destructs, but does not cause the process to terminate (unless no other threads are extant).

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 Arguments

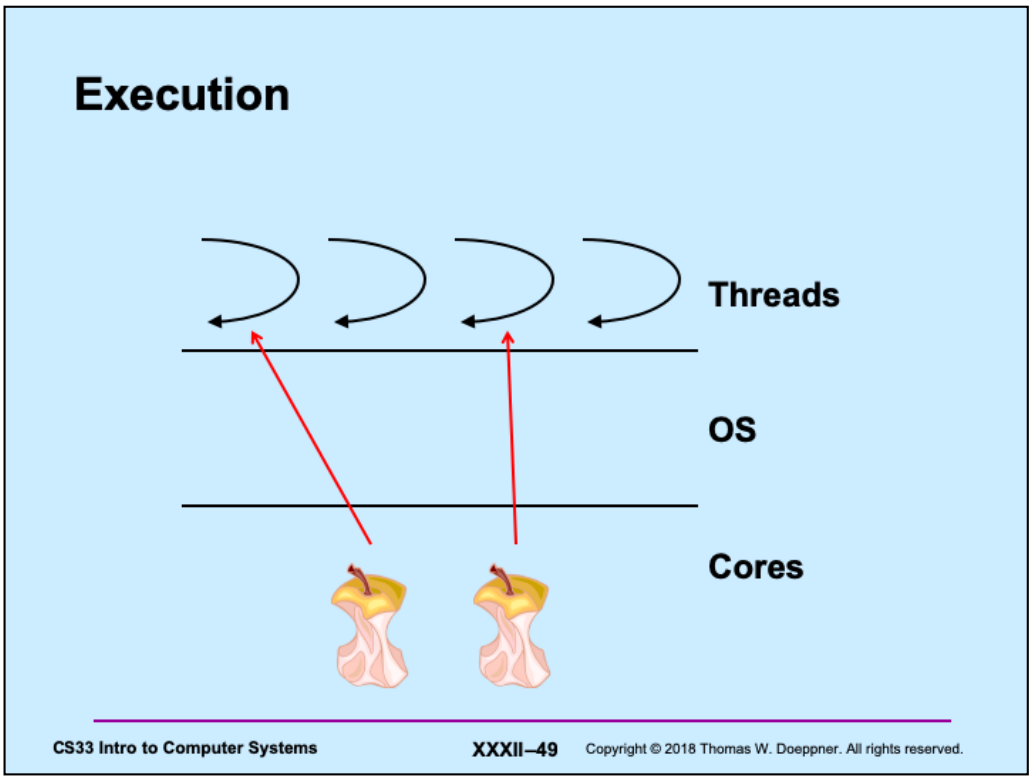
```
typedef struct args {
  int src;
  int dest;
} args_t;

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);
}
```

Quiz 1

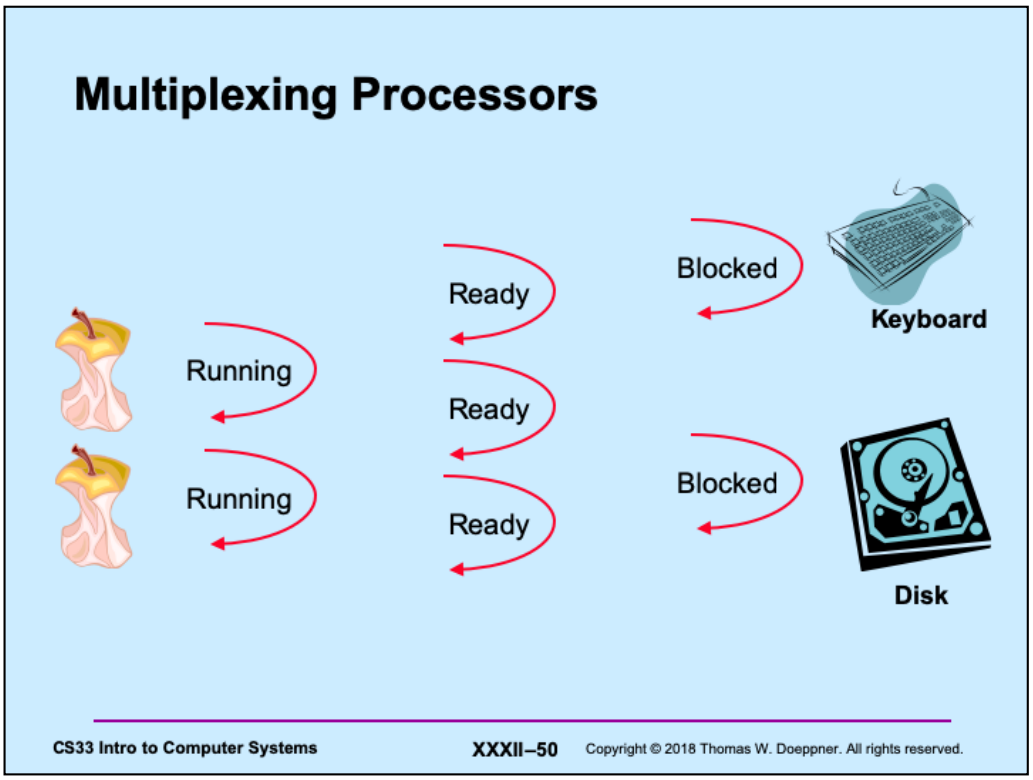
Does this work?

a) **yes**

b) **no**



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 processor cores. Periodically, say every millisecond, each processor is core and calls upon the OS to determine if another thread should run. If so, the current thread on the core 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 core, all threads make progress and give the appearance of running simultaneously.



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 core, in the latter they can. The scheduler determines which ready threads should be assigned cores. Ready threads that have been assigned cores 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%dl\n", (long)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;
}
```


While it's not clear what the function work actually does, its purpose is simply to occupy a processor for a fair amount of time, time directly proportional to its argument N. The idea behind this code is that we compute how long it takes one thread to compute work(N). We then compute how long it takes for M threads to each compute work(N/M). The total amount of computation done by these M threads is the same is done by one thread calling work(N). If we run this on a one-core computer, then the ratio of the time for M threads each computing work(N/M) to the time of one thread computing work(N) is the the overhead of running M threads.

Similarly, if we run this on a P-core processor, the ratio of the time for PM threads each computing work(N/PM) to the time of P threads each computing work(N/P) is the overhead of running PM threads on a P-core processor.

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 8. It runs in time *p* on the same processor when *nthreads* is 400.

a) ***n* << *p* (slower)**
b) ***n* ≈ *p* (same speed)**
c) ***n* >> *p* (faster)**

Problem

```
  pthread_create(&thread, 0, start, 0);

  …


void *start(void *arg) {
  long BigArray[128*1024*1024];
  …
  return 0;
}
```


Here we are creating a thread that has a very large local variable that, of course, is allocated on the thread's stack. How can we be sure that the thread's stack is actually big enough? As it turns out, the default stack size for threads in Linux is two megabytes.

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, 130*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).

What happens if a thread uses more stack space than was allotted to it? It would probably clobber memory holding another thread's stack, which could lead to some rather difficult to debug problems. To guard against such happenings, The lowest-address page of a thread's stack (recall that stacks grow downwards) is made inaccessible, meaning that any reference to it will generate a fault. Thus if the thread references just beyond its allotted stack, there will be a fault which, though not good, makes it clear that this thread has exceeded its stack space.