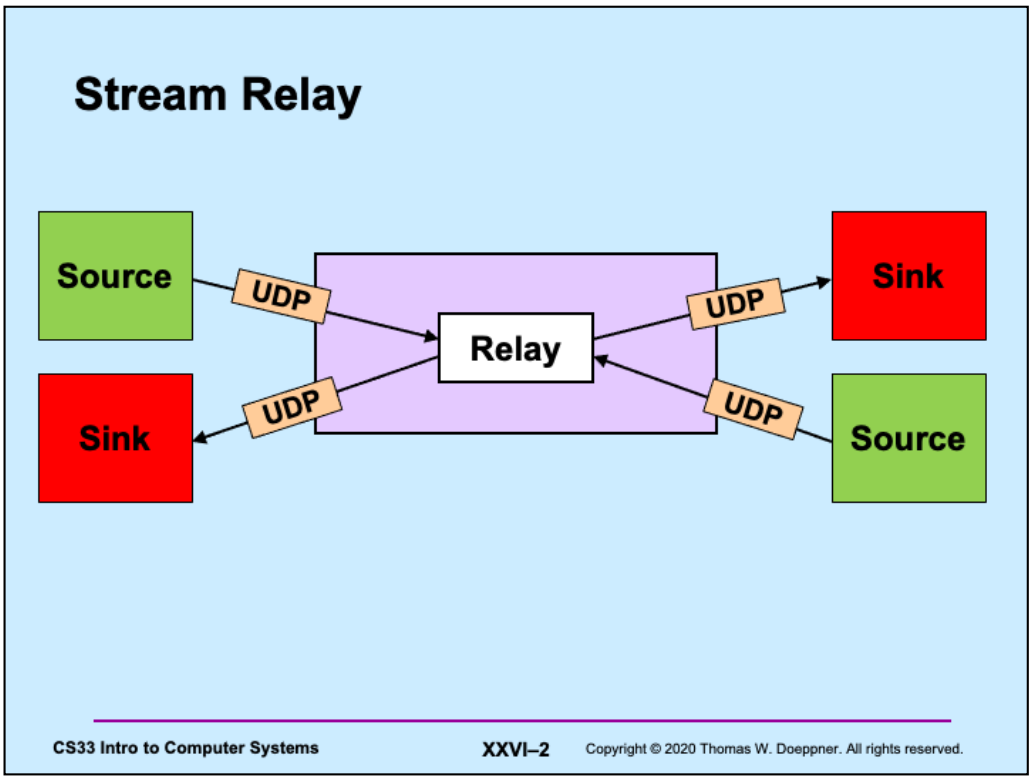
CS 33

Event-Based Programming

Copyright © 2020 Thomas W. Doeppner. All rights reserved.



We look 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 we examine a simple relay: we want to write a program that takes data received via UDP from a source on the left and forwards it (via UDP) to a sink on the right. At the same time it's taking data received from a source on the right and forwards it (via UDP) to a sink on the left.

Solution?

```
while(…) {
   size = read(left, buf, sizeof(buf));
   write(right, buf, size);
   size = read(right, buf, sizeof(buf));
   write(left, buf, size);
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

This solution is probably not what we'd want, since it strictly alternates between processing the data stream in one direction and then the other.

Note that to simply the slides a bit, even though we're using UDP, we'll use read and write system calls – the source and destination are assumed in each case.

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
);
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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, sizeof(message_t));
      if (FD_ISSET(right, &rd))
         read(right, bufRL, sizeof(message_t));
      if (FD_ISSET(right, &wr))
         write(right, bufLR, sizeof(message_t));
      if (FD_ISSET(left, &rd))
         write(left, bufRL, sizeof(message_t));
   }
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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 be included in the set, the last checks to see if a particular file descriptor is included in the set.

This sketch doesn't quite work because it doesn't take into account the fact that we have limited buffer space: we can't read two messages in a row from one side without writing the first to the other side before reading the second. Furthermore, even though select may say it's possible to write to either the left or the right side, we can't do so until we're read in some data from the other side. Also, the fd_sets that are select's arguments are modified on return from select to indicate if it's now possible to read or write on the associated file descriptor. Thus if, on return from select, it's not possible to use that file descriptor, its associated bit will be zero. We need to explicitly set it to one for the next call so that select knows we're still interested.

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;
  message_t bufLR;
  message_t bufRL;
  int maxFD = max(left, right) + 1;
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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);
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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)) {
  read(left, bufLR, sizeof(message_t));
  left_read = 0;
  right_write = 1;
}
if (FD_ISSET(right, &rd)) {
  read(right, bufRL, sizeof(message_t));
  right_read = 0;
  left_write = 1;
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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)) {
      write(right, bufLR, sizeof(message_t));
      left_read = 1;
      right_write = 0;
    }
    if (FD_ISSET(left, &wr)) {
      write(left, bufRL, sizeof(message_t));
      right_read = 1;
      left_write = 0;
    }
  }
  return 0;
}
```

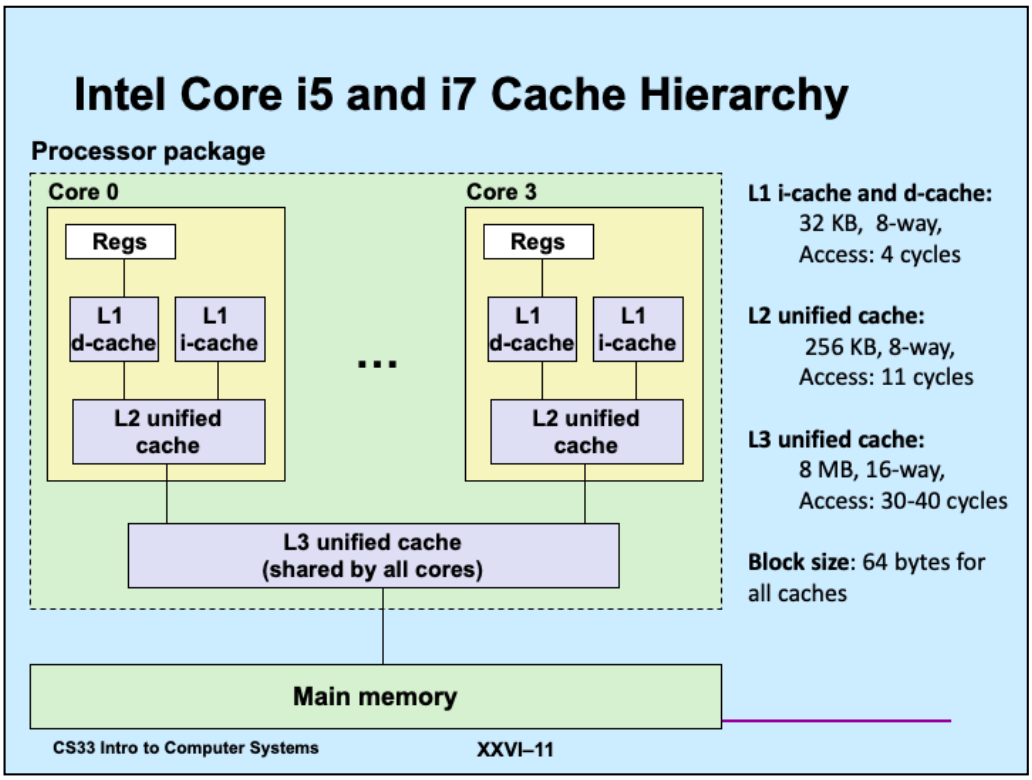
 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Similarly for writing: if we've written something to one side, we have nothing more to write to that side, but are now interested in reading from the other side.

CS 33

Caching and Program Optimization

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.



Supplied by CMU.

The L3 cache is known as the *last-level cache* (LLC) in the Intel documentation.

One concern is whether what's contained in, say, the L1 cache is also contained in the L2 cache. if so, caching is said to be *inclusive*. If what's contained in the L1 cache is definitely not contained in the L2 cache, caching is said to be *exclusive*. An advantage of exclusive caches is that the total cache capacity is the sum of the sizes of each of the levels, whereas for inclusive caches, the total capacity is just that of the largest. An advantage of inclusive caches is that what's been brought into the cache hierarchy by one core is available to the other core.

AMD processors tend to have exclusive caches; Intel processors tend to have inclusive caches.

We will explain what is meant by 8-way and 16-way caches in a subsequent lecture, if we have time. For the purposes of this lecture, the distinction is not important.

Accessing Memory

- Program references memory (load)
  - if not in cache (*cache miss*), data is requested from RAM
    - » fetched in units of 64 bytes
      - aligned to 64-byte boundaries (low-order 6 bits of address are zeroes)
    - » if memory accessed sequentially, data is pre-fetched
    - » data stored in cache (in 64-byte *cache lines*)
      - stays there until space must be re-used (least recently used is kicked out first)
  - if in cache (*cache hit*) no access to RAM needed
- Program modifies memory (store)
  - data modified in cache
  - eventually written to RAM in 64-byte units

Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Quiz 1

The previous slide said that 64 bytes of memory from contiguous locations are transferred at a time. Suppose we have memory that transfers 128 contiguous bytes in the same amount of time. If we have a program that reads memory one byte at a time from random (but valid) memory locations, how much faster will it run with the new memory system than with the old?

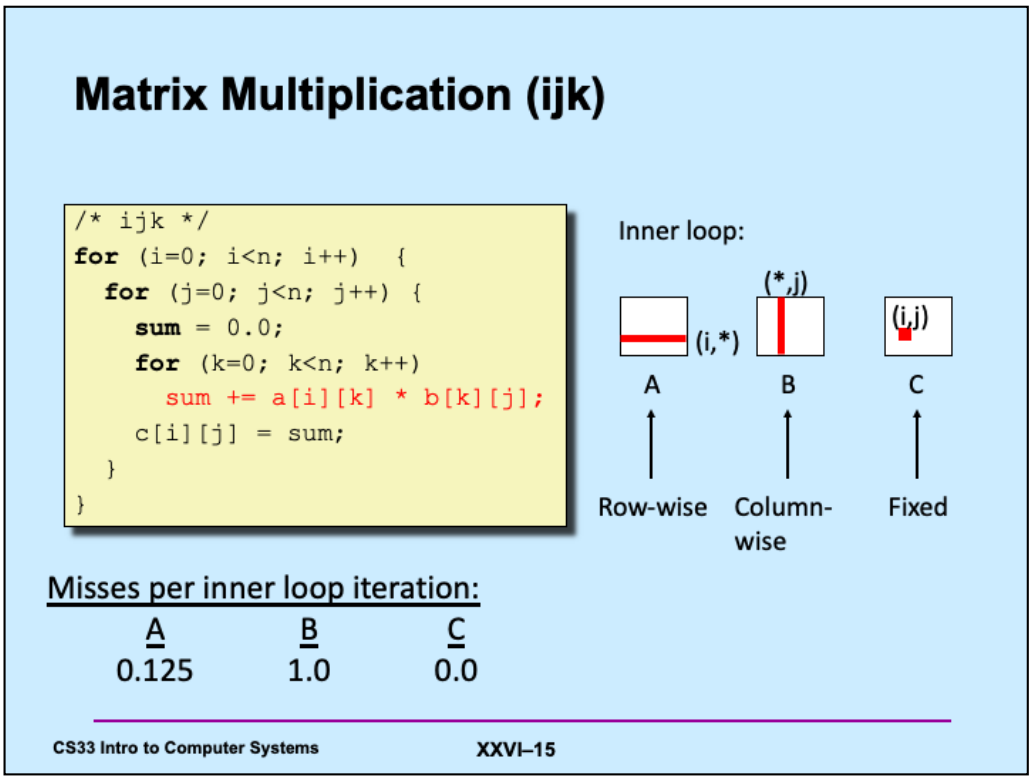
a) half as fast
b) roughly the same speed
c) twice as fast
d) four times as fast

Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Layout of C Matrices in Memory

- C matrices allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < n; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - data fetched from RAM in 64-byte units
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - accesses distant elements
  - if array element is 8 bytes, 56 bytes (out of 64) are not used
    - » effective throughput reduced by factor of 8

Adapted from a slide supplied by CMU.

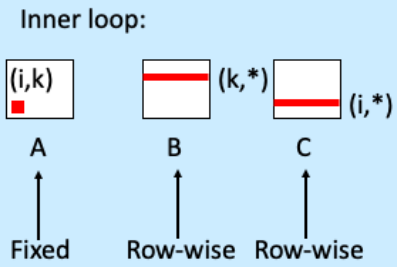


Supplied by CMU.

Assume we are multiplying arrays of doubles, thus each element is eight bytes long, and thus a cache line holds eight matrix elements.

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



Misses per inner loop iteration:

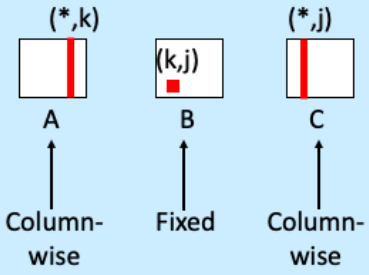
A	B	C
0.0	0.125	0.125

Supplied by CMU.

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

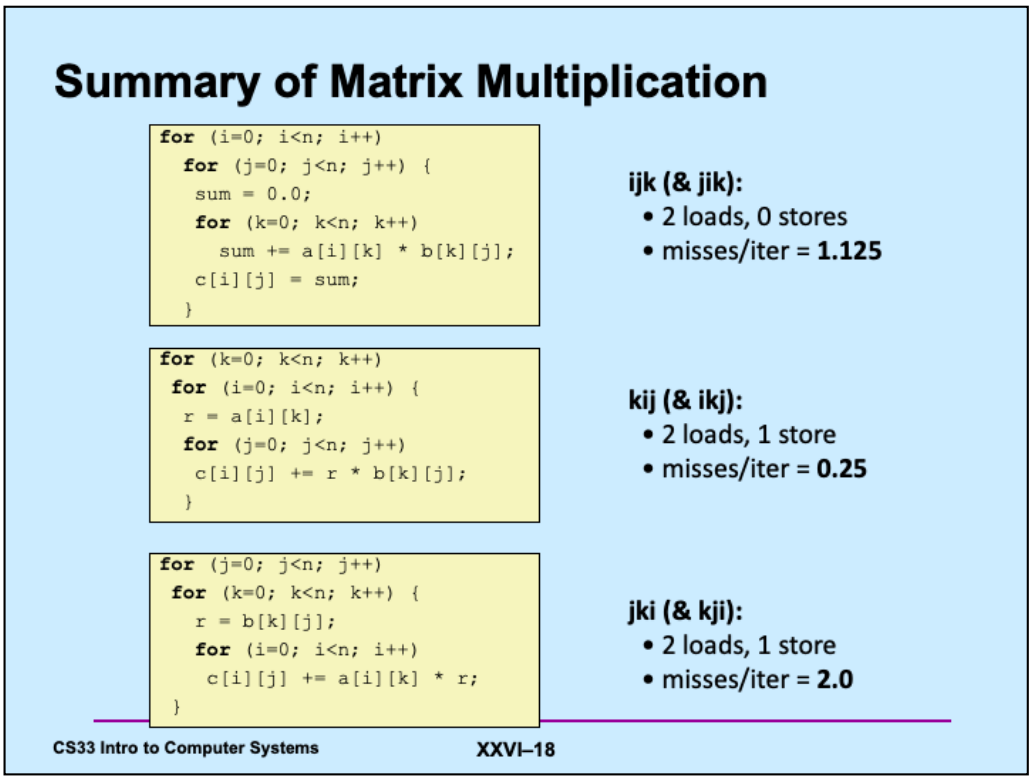
Inner loop:



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Supplied by CMU.



Supplied by CMU.

In Real Life ...

- Multiply two 1024x1024 matrices of doubles on sunlab machines
  - ijk
    - » 4.185 seconds
  - kij
    - » 0.798 seconds
  - jki
    - » 11.488 seconds

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

CS 33

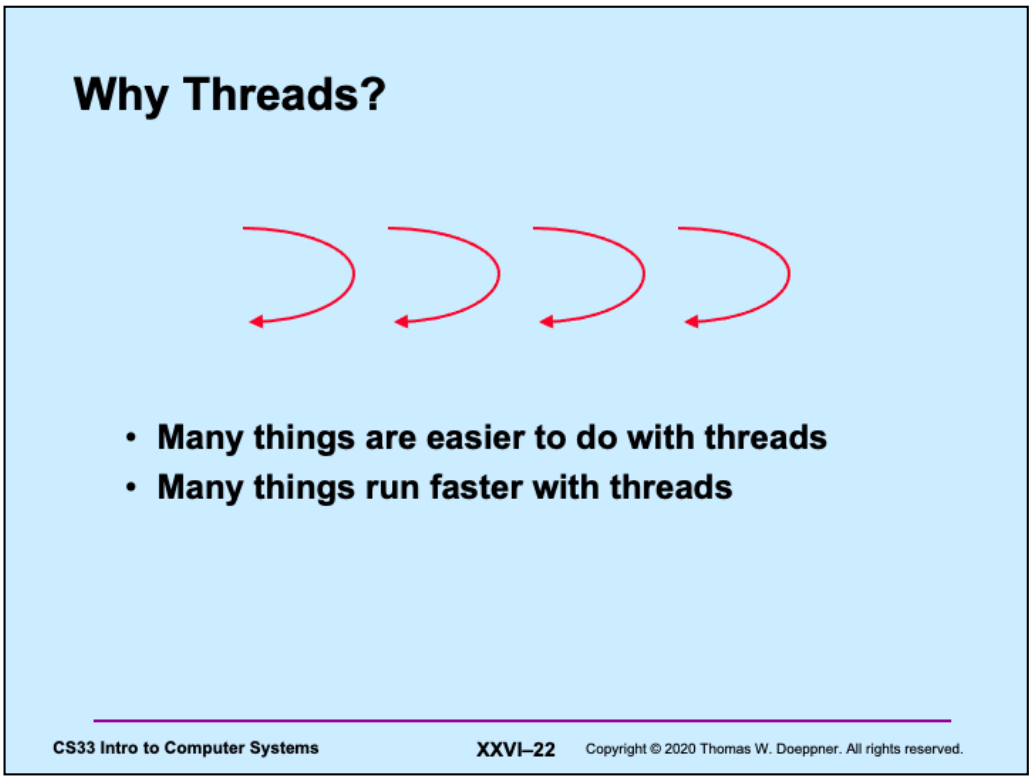
Multithreaded Programming I

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Multithreaded Programming

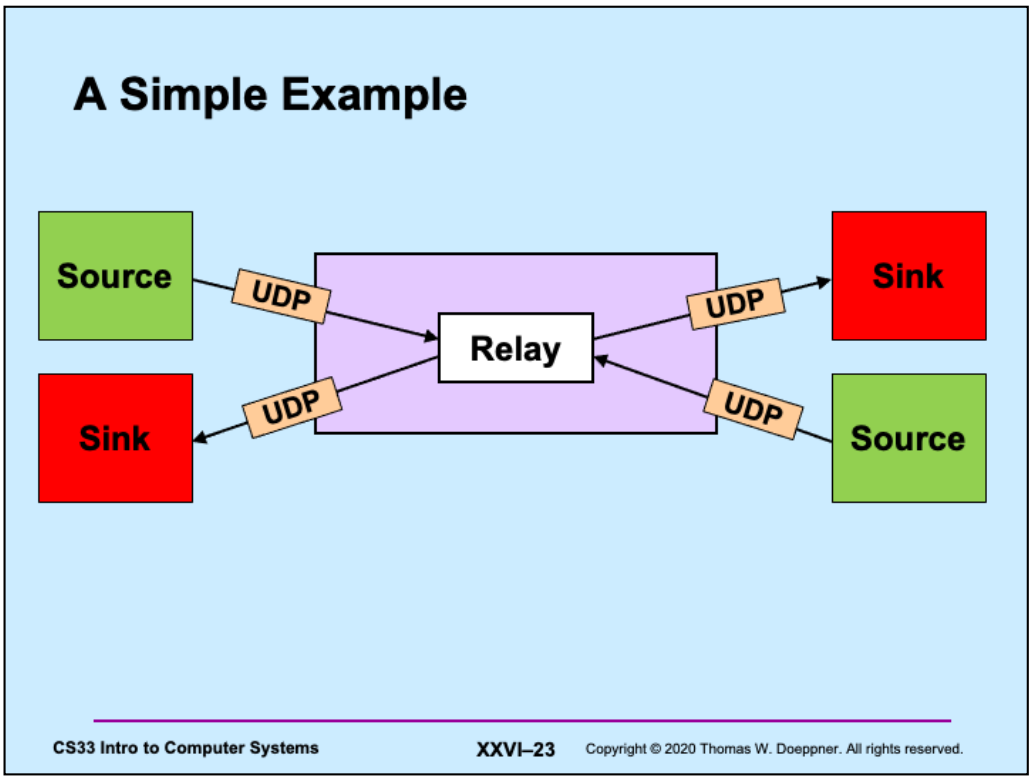
- A thread is a virtual processor
  - an independent agent executing instructions
- Multiple threads
  - multiple independent agents executing instructions

Copyright © 2020 Thomas W. Doeppner. All rights reserved.



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.



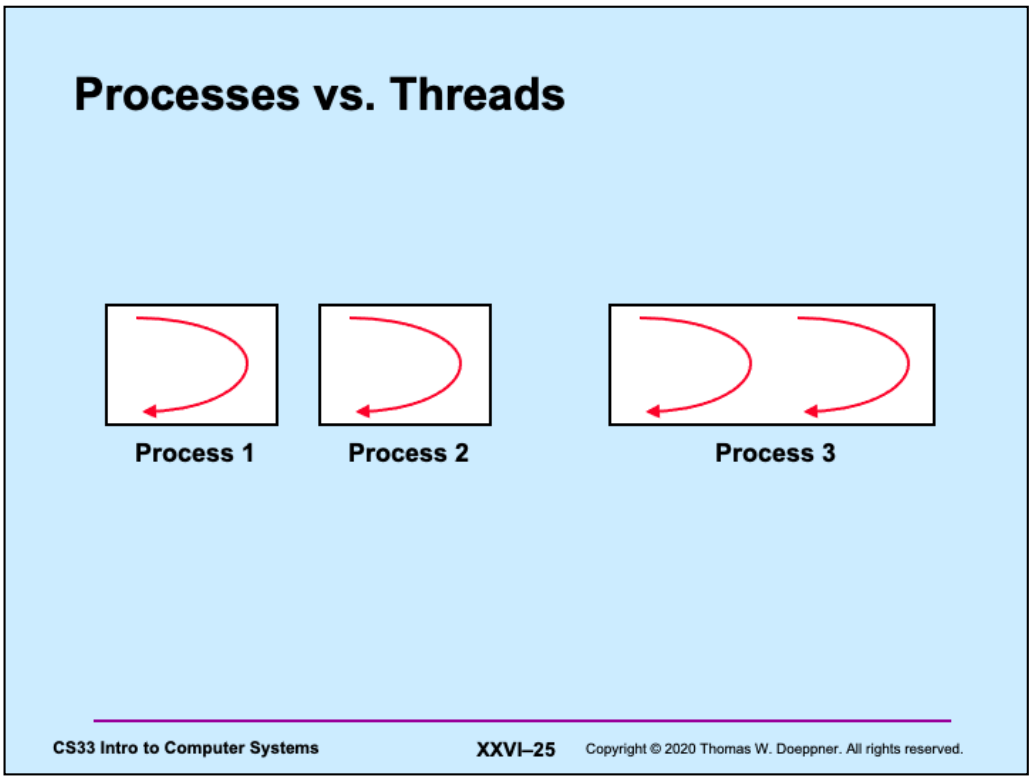
For a simple example of a problem that is more easily solved with threads than without, let's take another look at the stream relay example.

Life With Threads

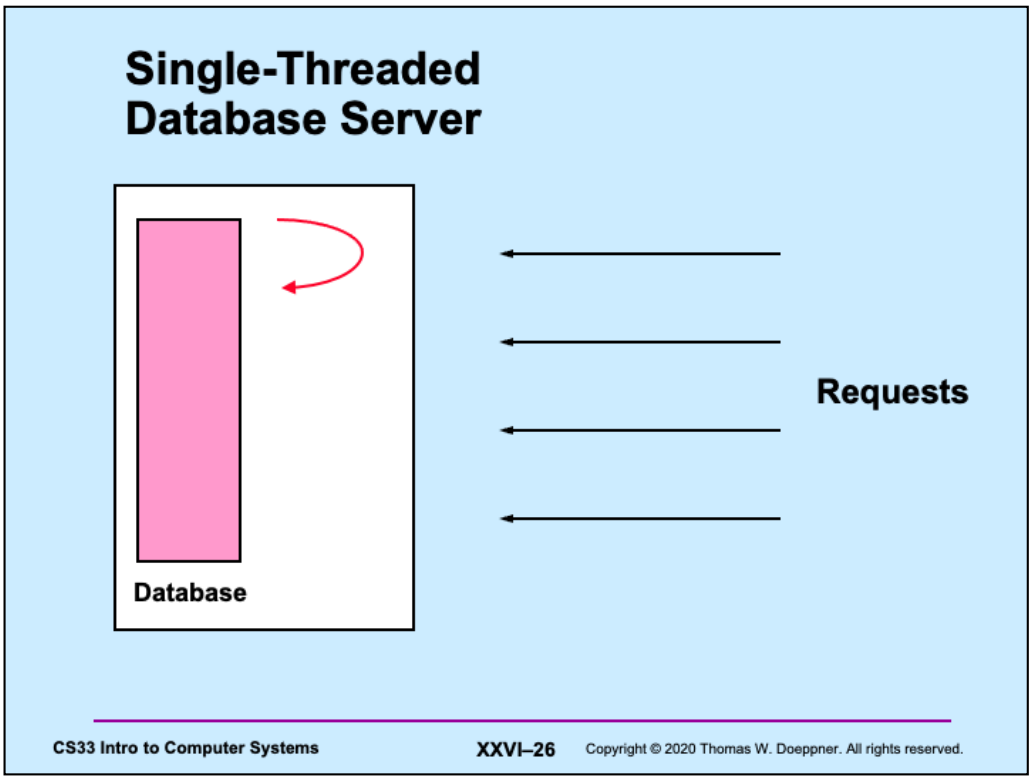
```
void copy(int source, int destination) {
  struct args *targs = args;
  char buf[BSIZE];

  while(1) {
    int len = read(source, buf, BSIZE);
    write(destination, buf, len);
  }
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

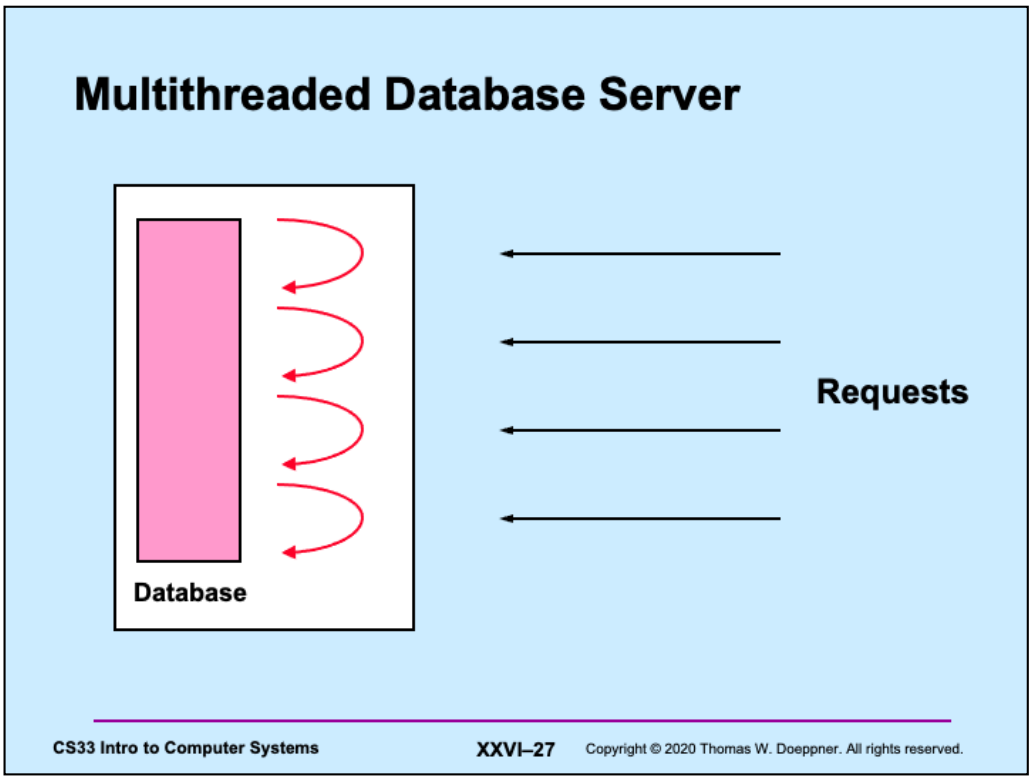
Here's an essentially equivalent solution to the one we saw previously that uses threads rather than select. We've left out the code that creates the threads (we'll see that pretty soon), but what's shown is executed by each of two threads. One has source set to the left side and destination to the right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

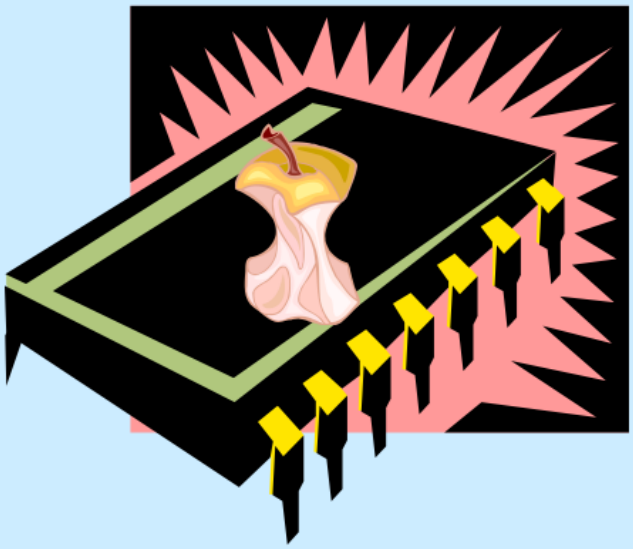


Here is another server example, a database server handling multiple clients. The single-threaded approach to dealing with these requests is to handle them sequentially or to multiplex them explicitly. The former approach would be unfair to quick requests occurring behind lengthy requests, and the latter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



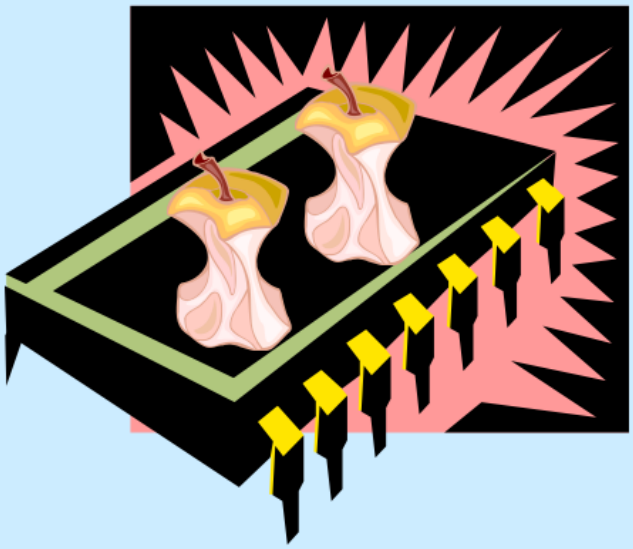
We now rearchitect our server to be multithreaded, assigning a separate thread to each request. The code is as simple as in the sequential approach and as fair as in the multiplexed approach. Some synchronization of access to the database is required, a topic we will discuss soon.

Single-Core Chips



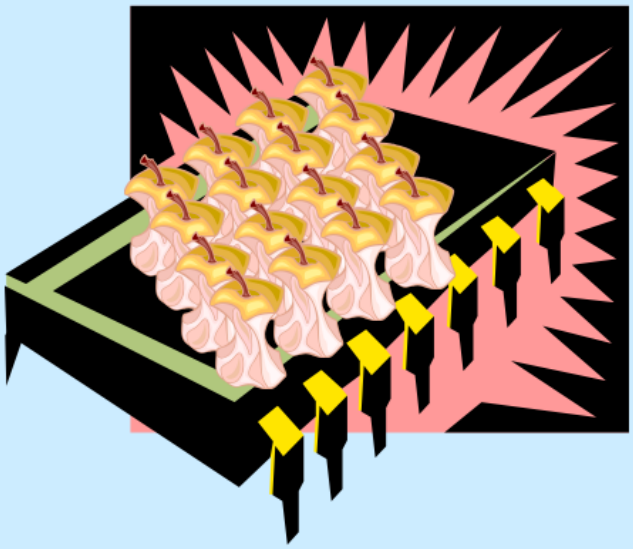
 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Dual-Core Chips



 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Multi-Core Chips



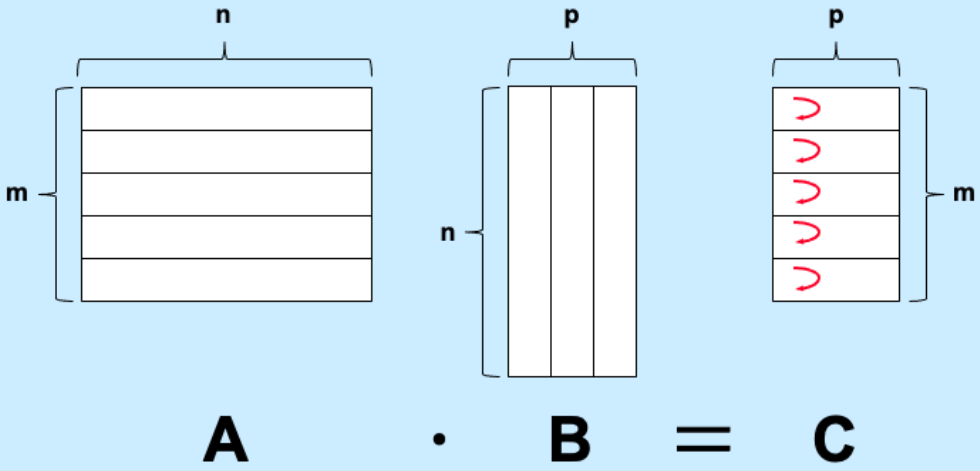
 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Good News/Bad News

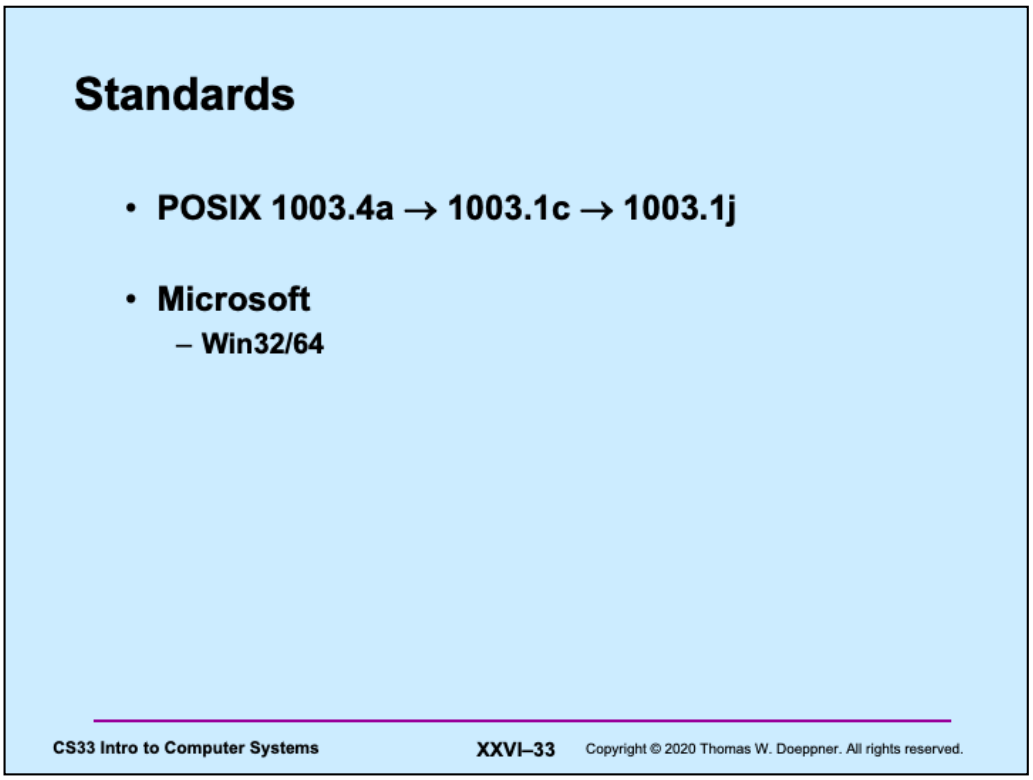
- ☺ Good news
  - multi-threaded programs can take advantage of multi-core chips (single-threaded programs cannot)
- ☹ Bad news
  - it's not easy
    - » must have parallel algorithm
      - employing at least as many threads as processors
      - threads must keep processors busy
        - doing useful work

Copyright © 2020 Thomas W. Doeppner. All rights reserved.

Matrix Multiplication Revisited



Copyright © 2020 Thomas W. Doeppner. All rights reserved.



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
  ...
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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 Finished?

```
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
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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
  ...
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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  ... */
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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);
}
```

 Copyright © 2020 Thomas W. Doeppner. All rights reserved.

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?