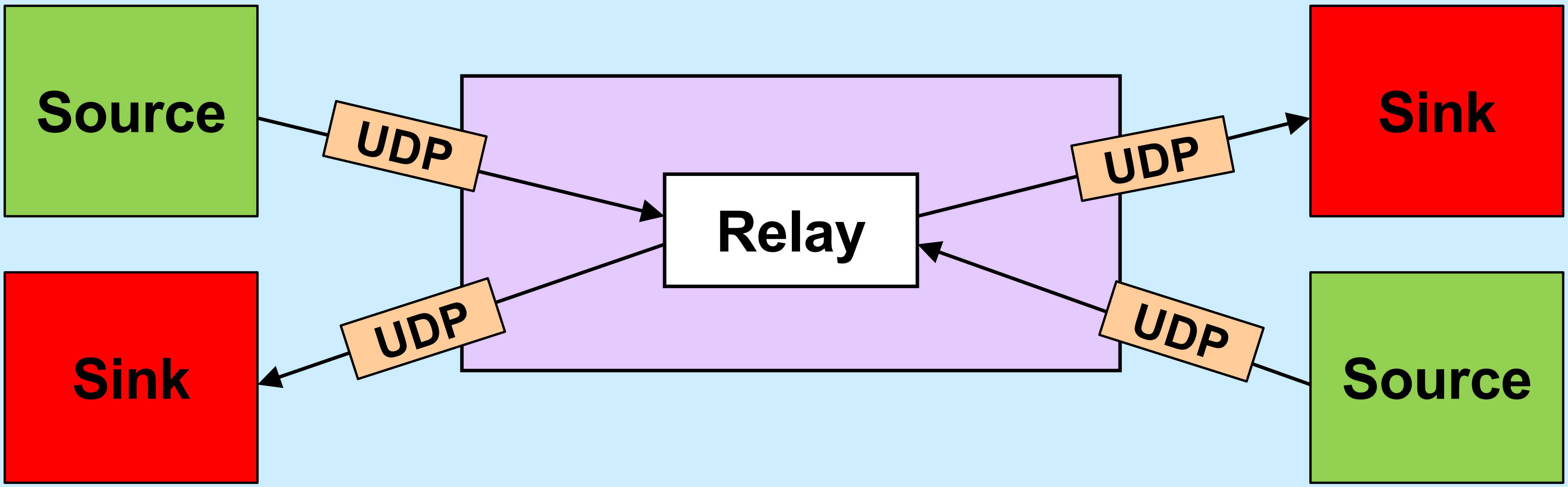
CS 33

Event-Based Programming

Stream Relay



Solution?

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

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
);
```

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));
   }
}
```

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;
```


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);
```

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;
}
```

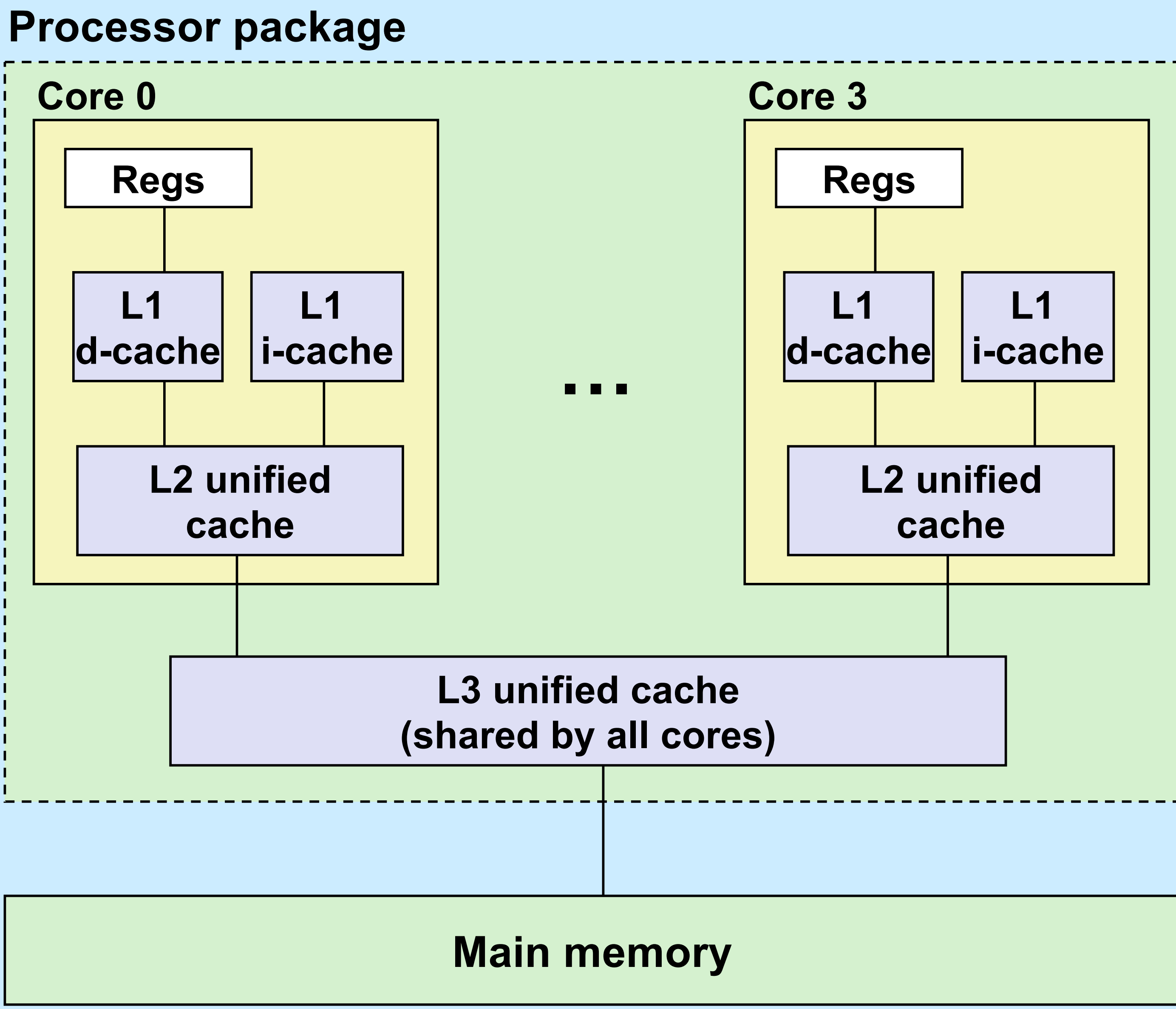
Relay (4)

```
    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;
}
```

CS 33

Caching and Program Optimization

Intel Core i5 and i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches

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

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?

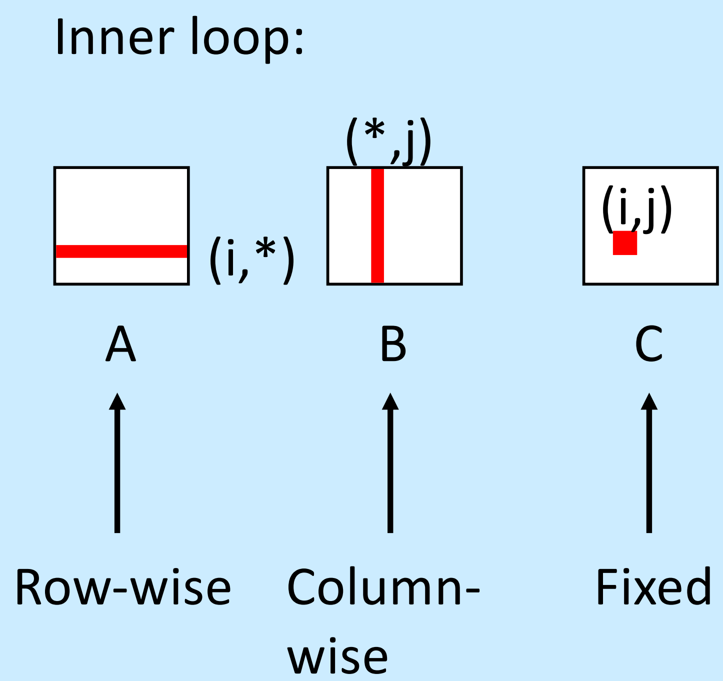
a) half as fast
b) roughly the same speed
c) twice as fast
d) four times as fast

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**

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

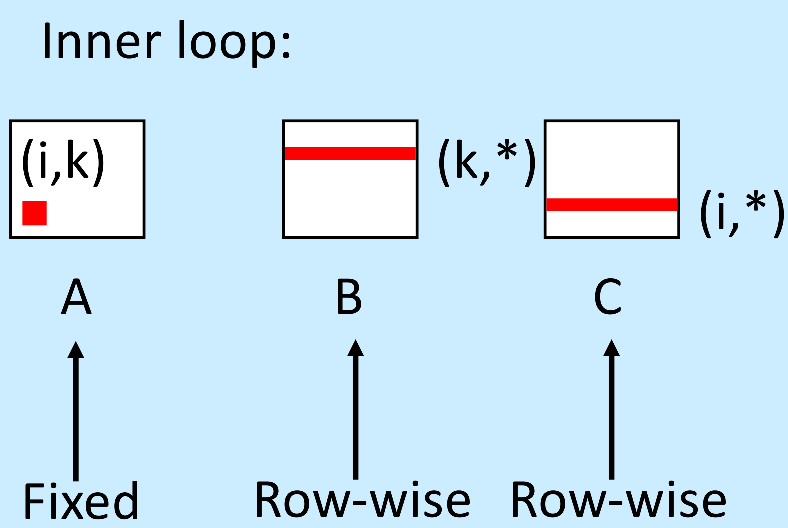


Misses per inner loop iteration:

A	B	C
0.125	1.0	0.0

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];
  }
}
```



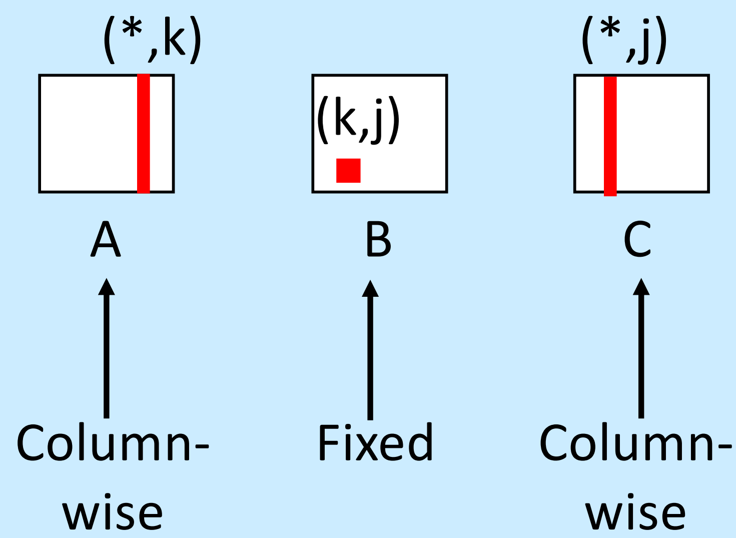
Misses per inner loop iteration:

A	B	C
0.0	0.125	0.125

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

Summary of Matrix Multiplication

```
for (i=0; i<n; i++)
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
  }
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = **1.125**

```
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];
  }
```

kij (& ikj):
- 2 loads, 1 store
- misses/iter = **0.25**

```
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;
 }
```

jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

In Real Life ...

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

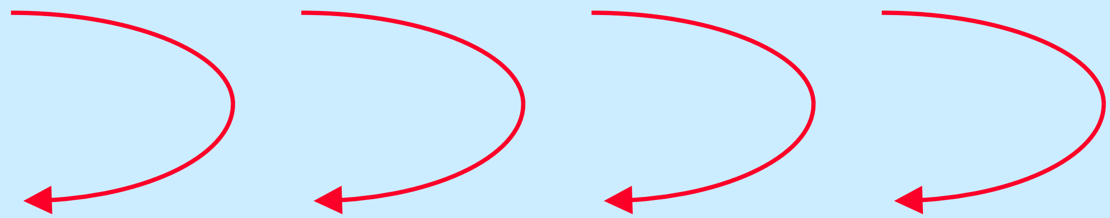
CS 33

Multithreaded Programming I

Multithreaded Programming

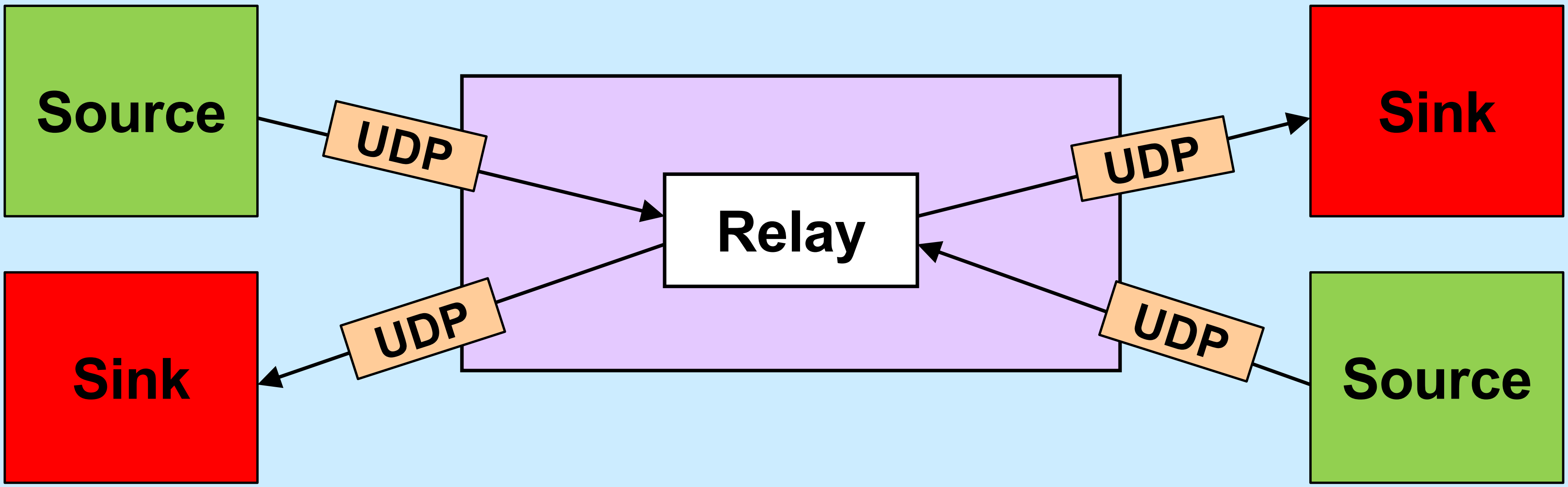
- A thread is a virtual processor
  - an independent agent executing instructions
- Multiple threads
  - multiple independent agents executing instructions

Why Threads?



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

A Simple Example

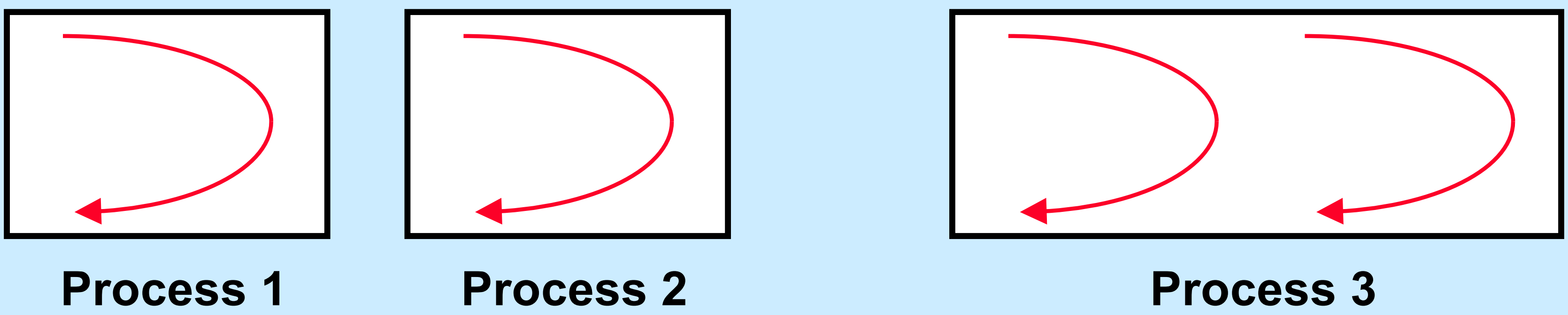


Life With Threads

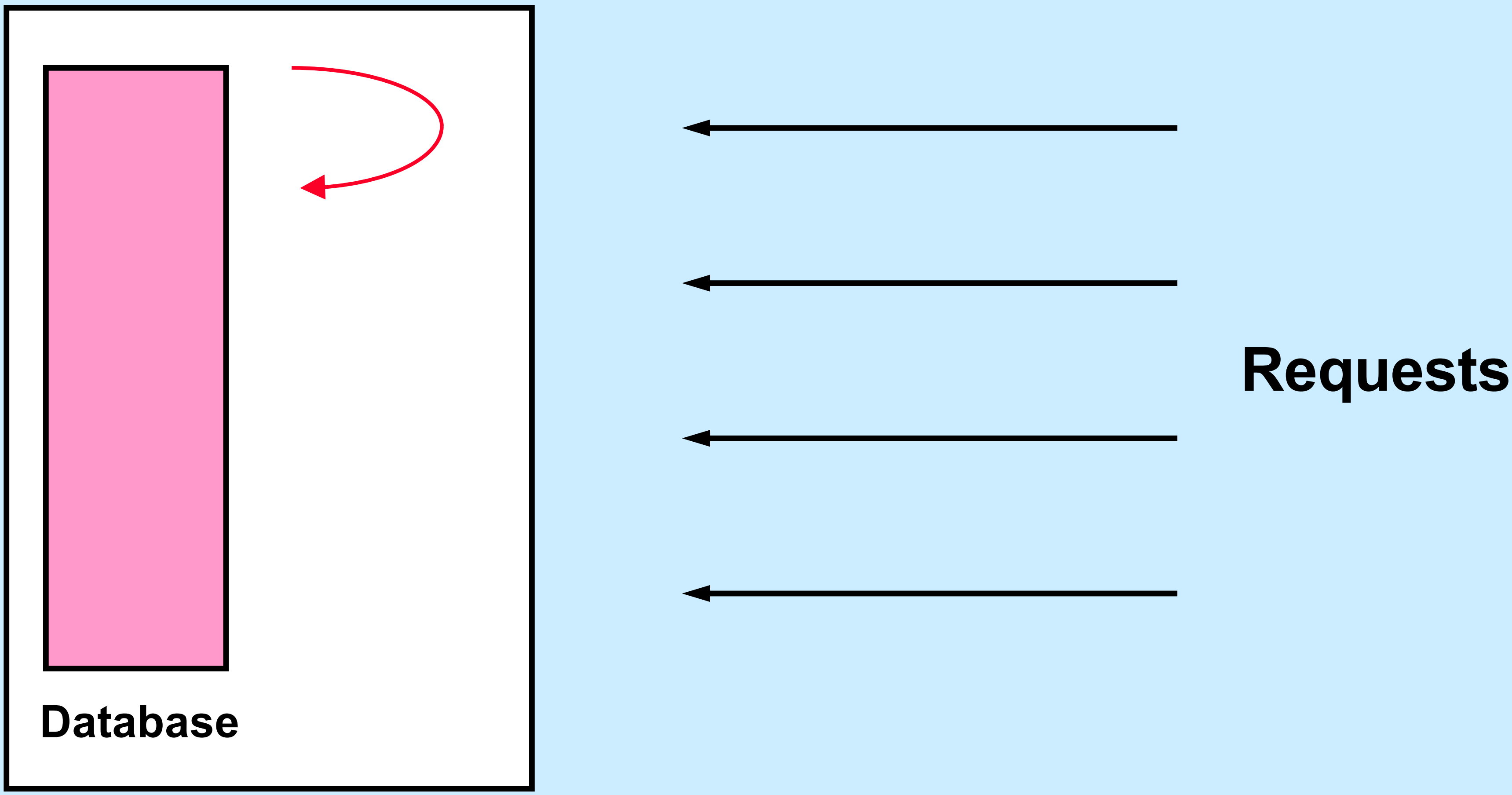
```
void copy(int source, int destination) {
  struct args *targs = args;
  char buf[BSIZE];

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

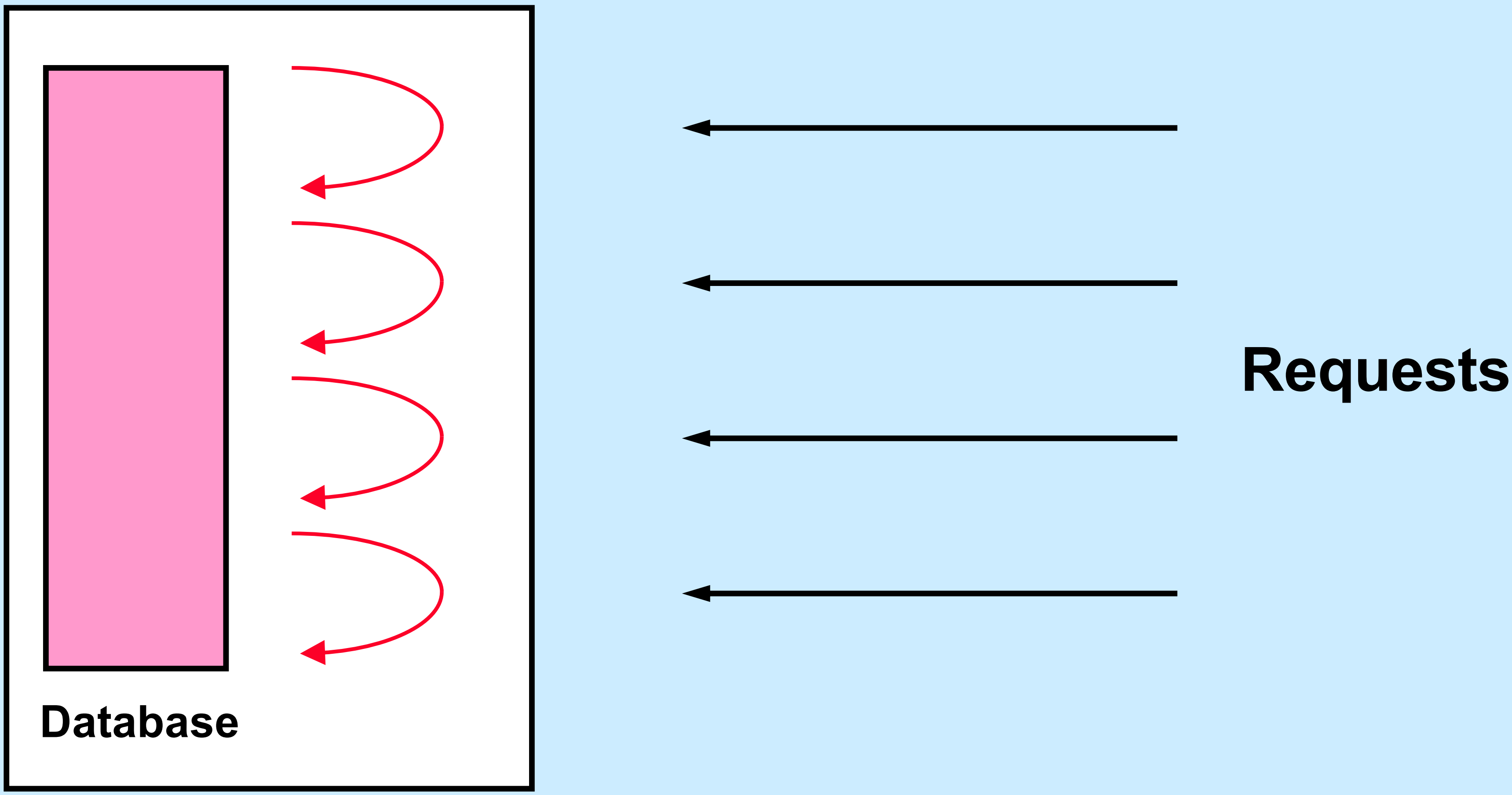

Processes vs. Threads



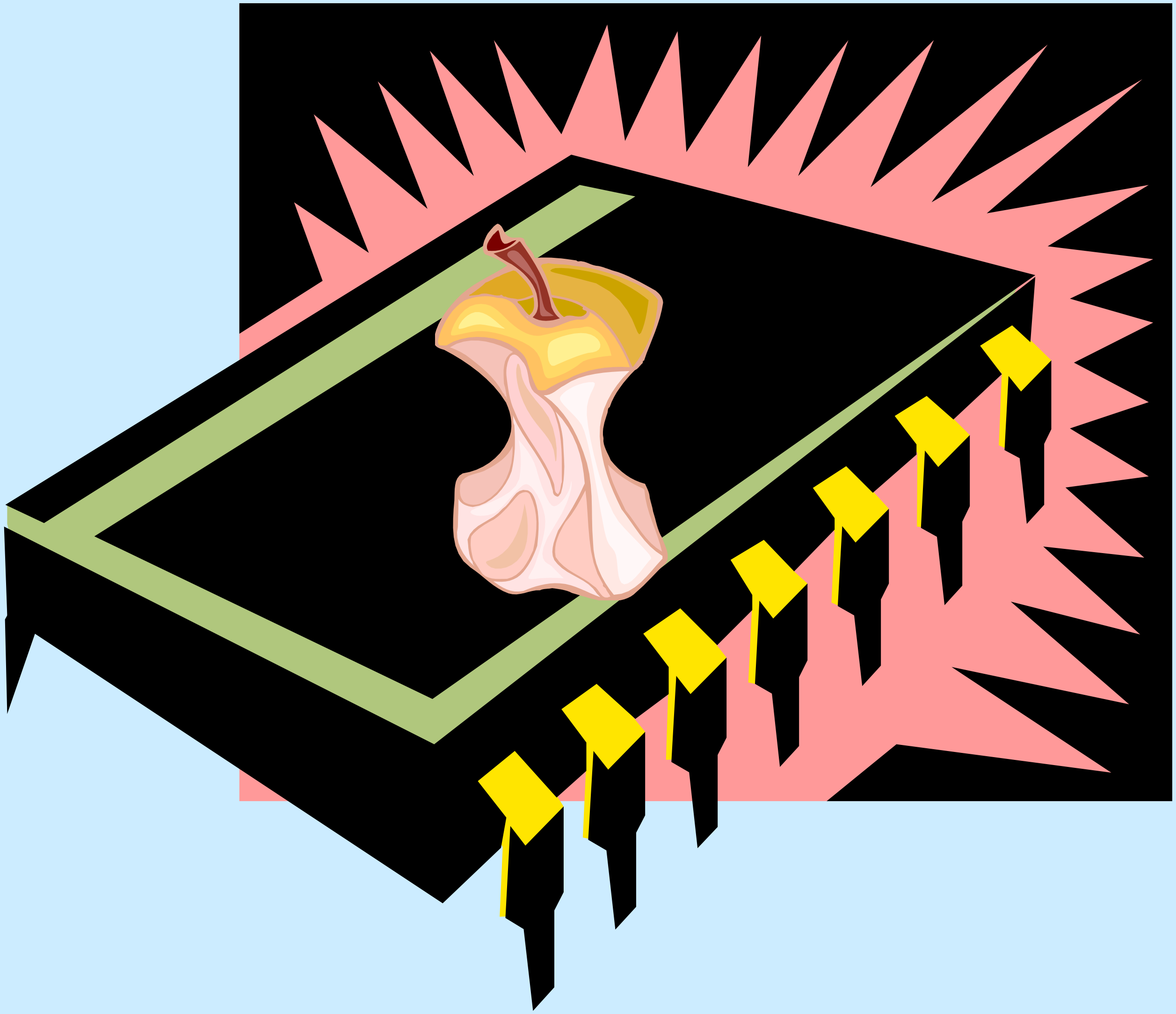
Single-Threaded Database Server



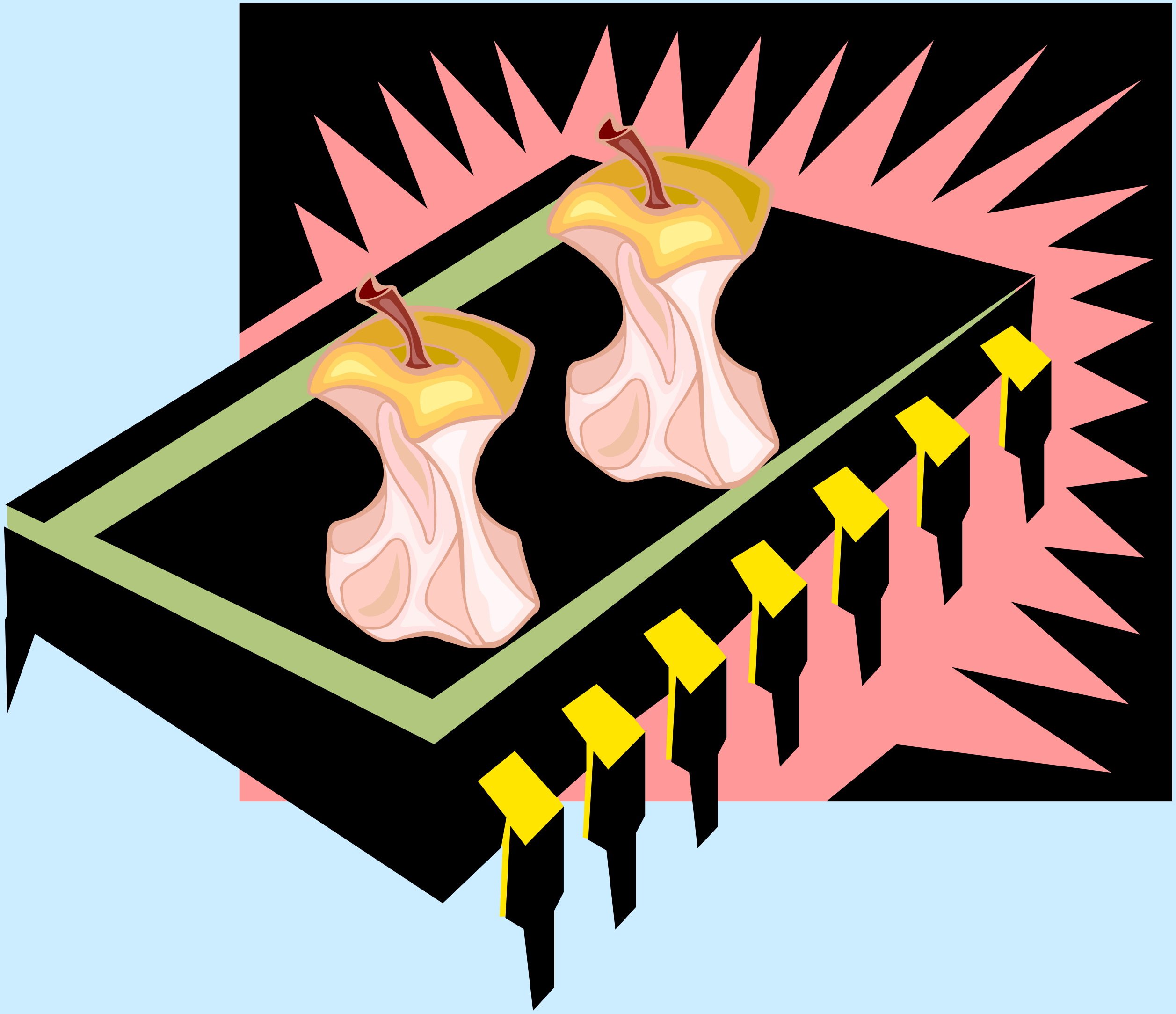
Multithreaded Database Server



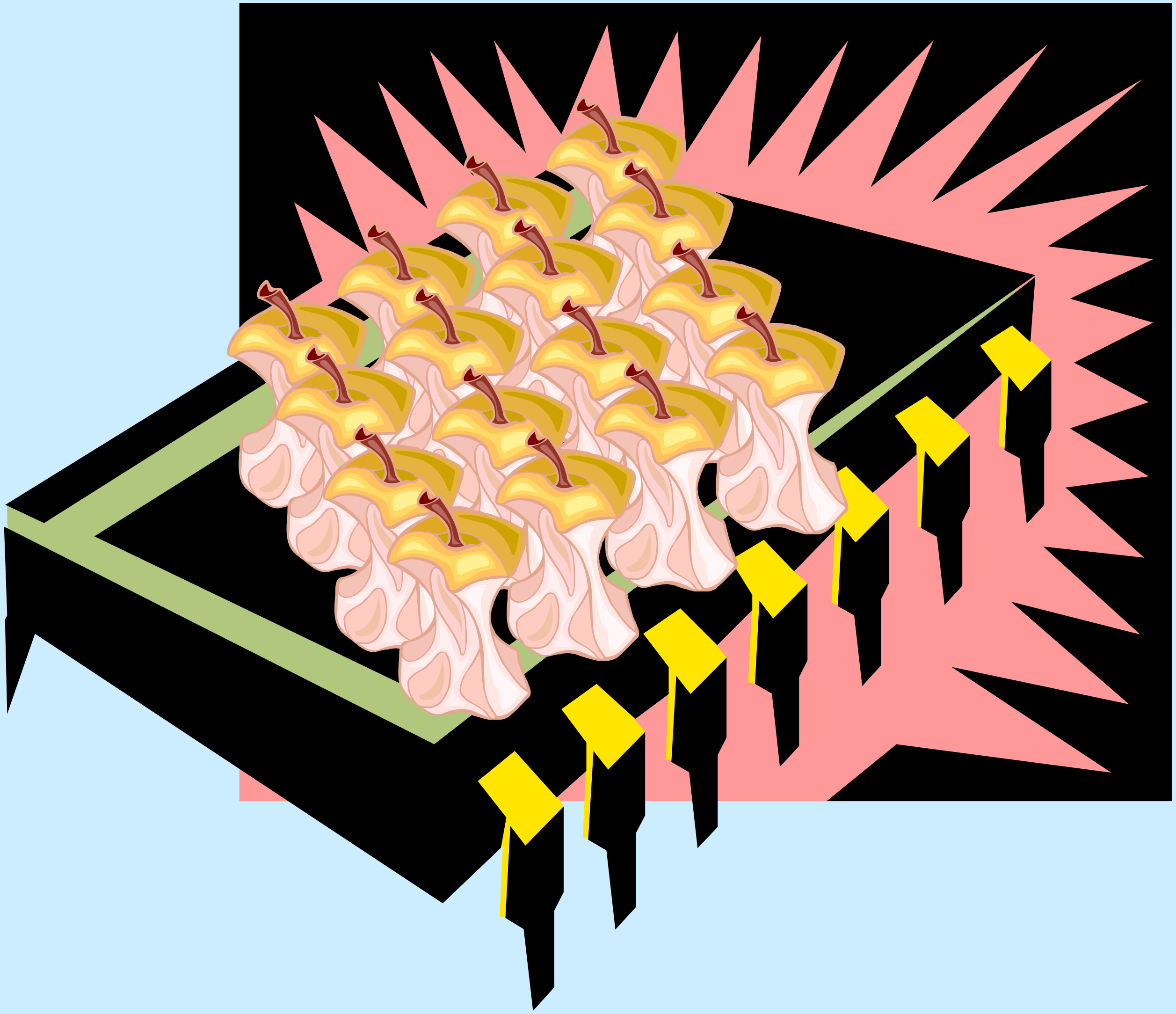
Single-Core Chips



Dual-Core Chips



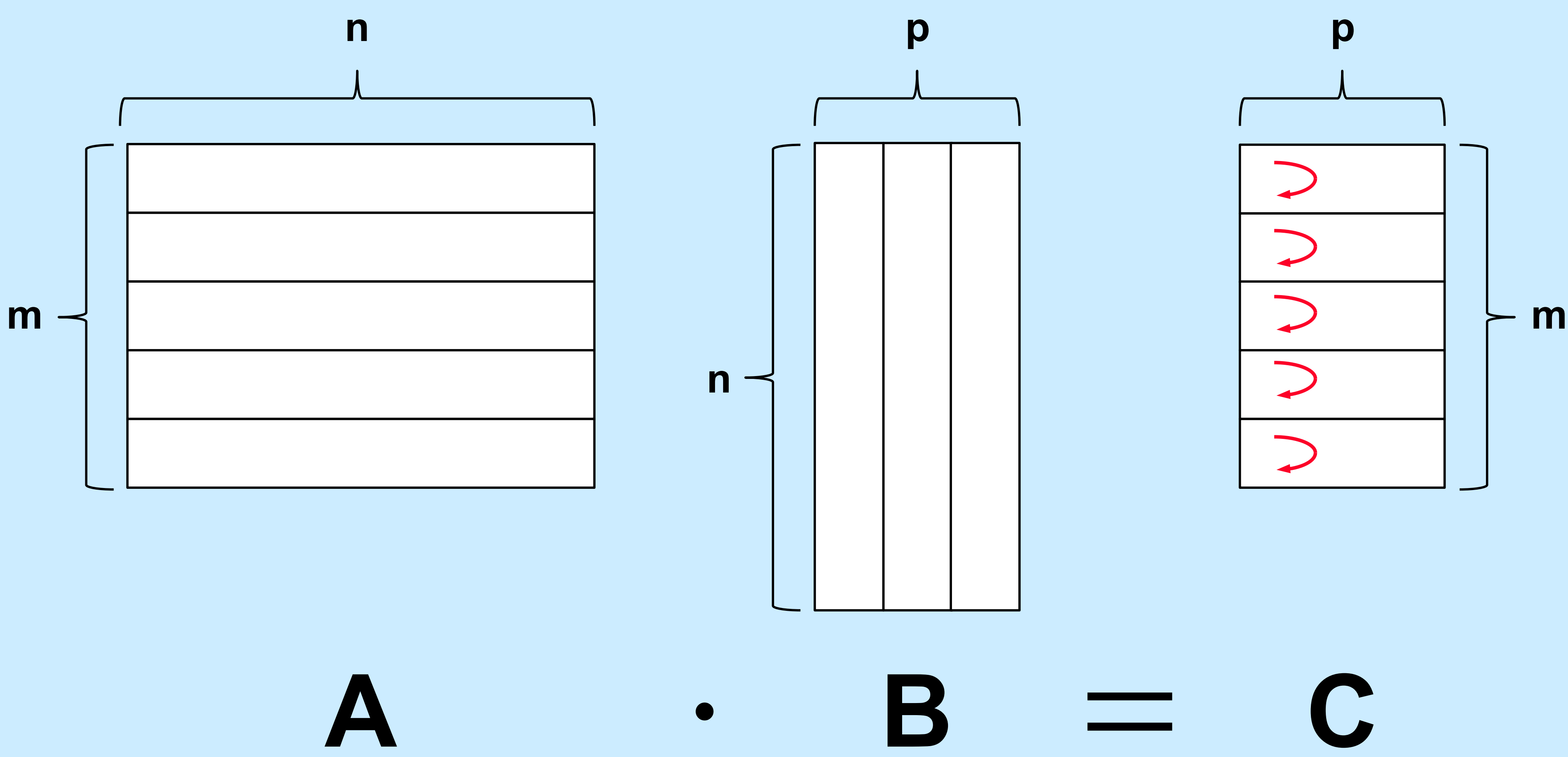
Multi-Core Chips



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
  - Win32/64

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
  ...
}
```

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
```

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
  ...
```

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  ... */
}
```


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);
}
```

