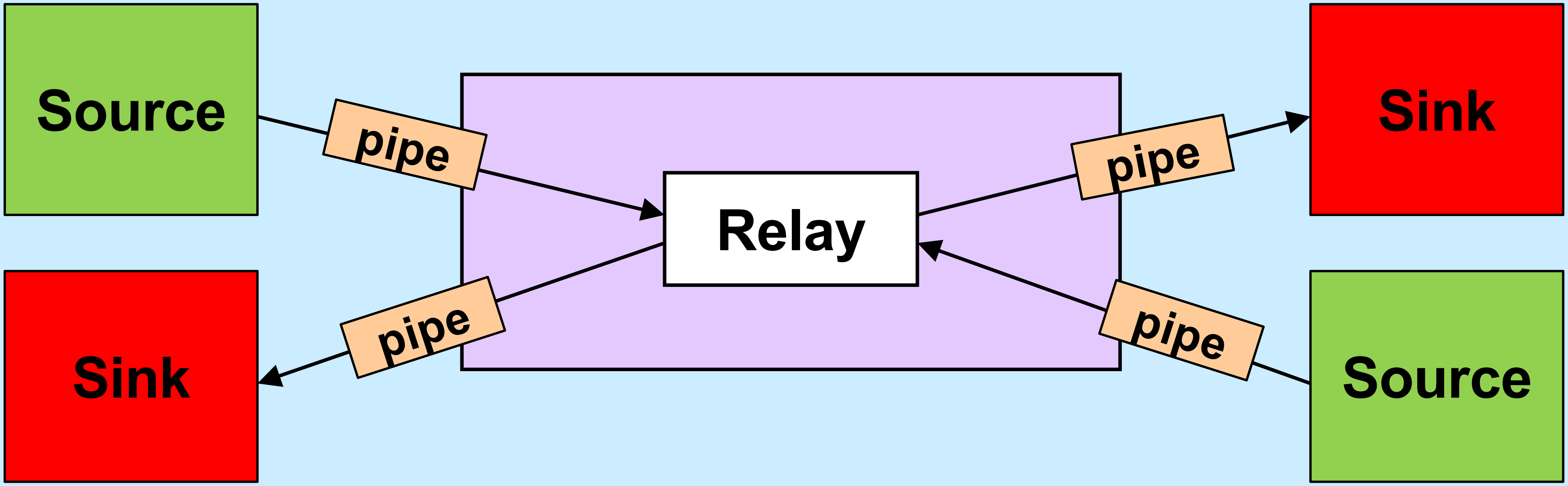
CS 33

Event-Based Programming
Multithreaded Programming I

Stream Relay



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, 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);
   }
}
```

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;
```

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)) {
  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;
}
```

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;
}
```

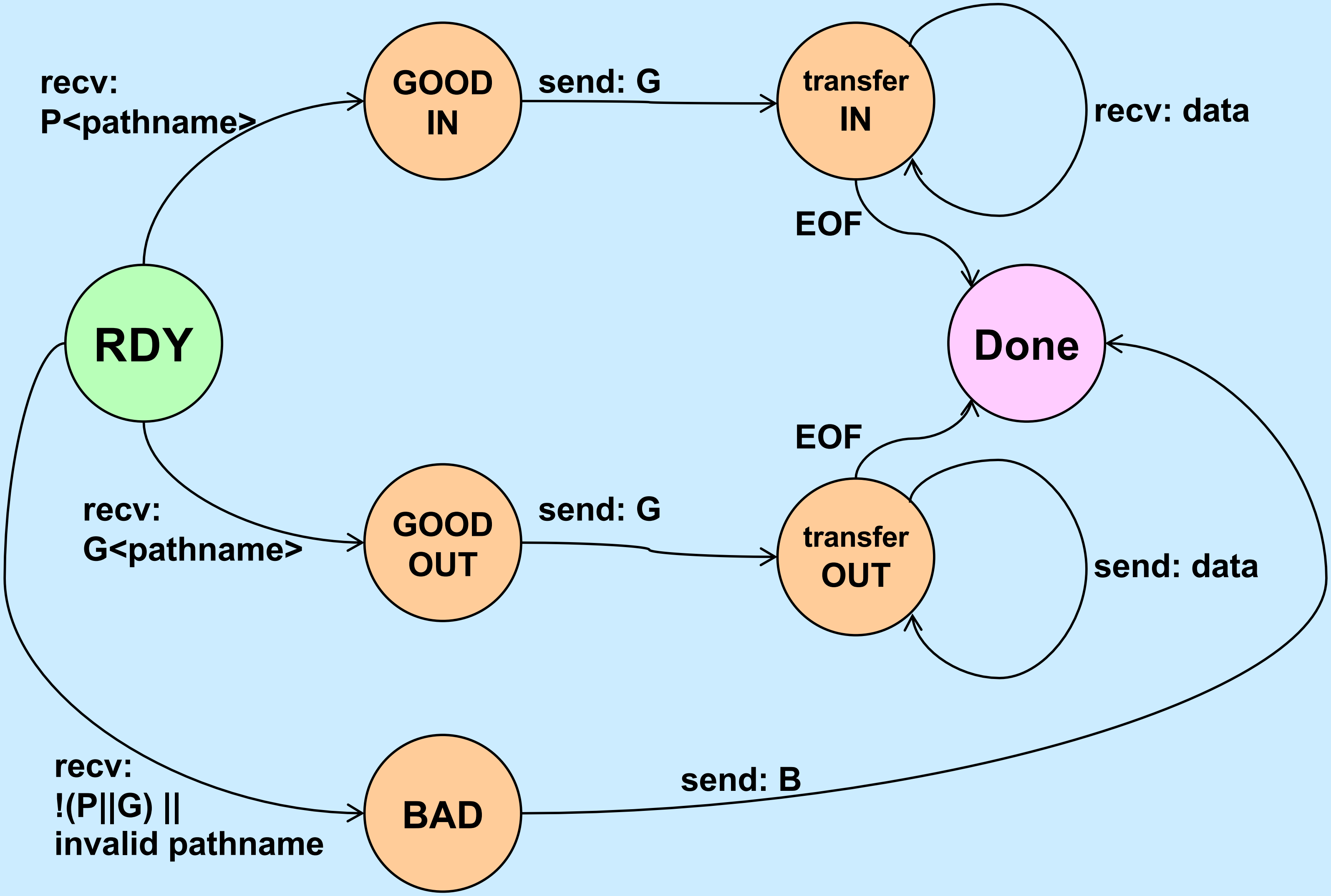
A Really Simple Protocol

- 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>

Server State Machine



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;
```

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;
```

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;
}
```

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;
}
```

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;
      }
```

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) {
```

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;
  }
}
```

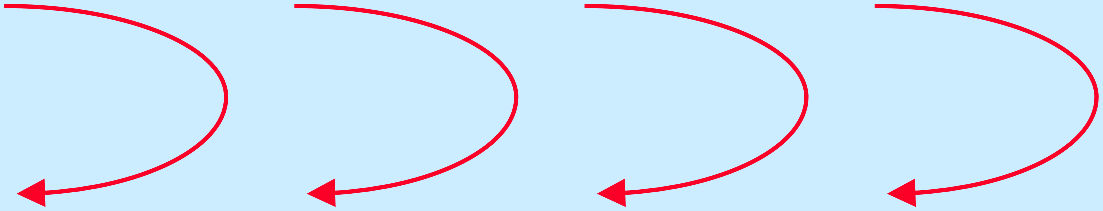
Problems

- 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

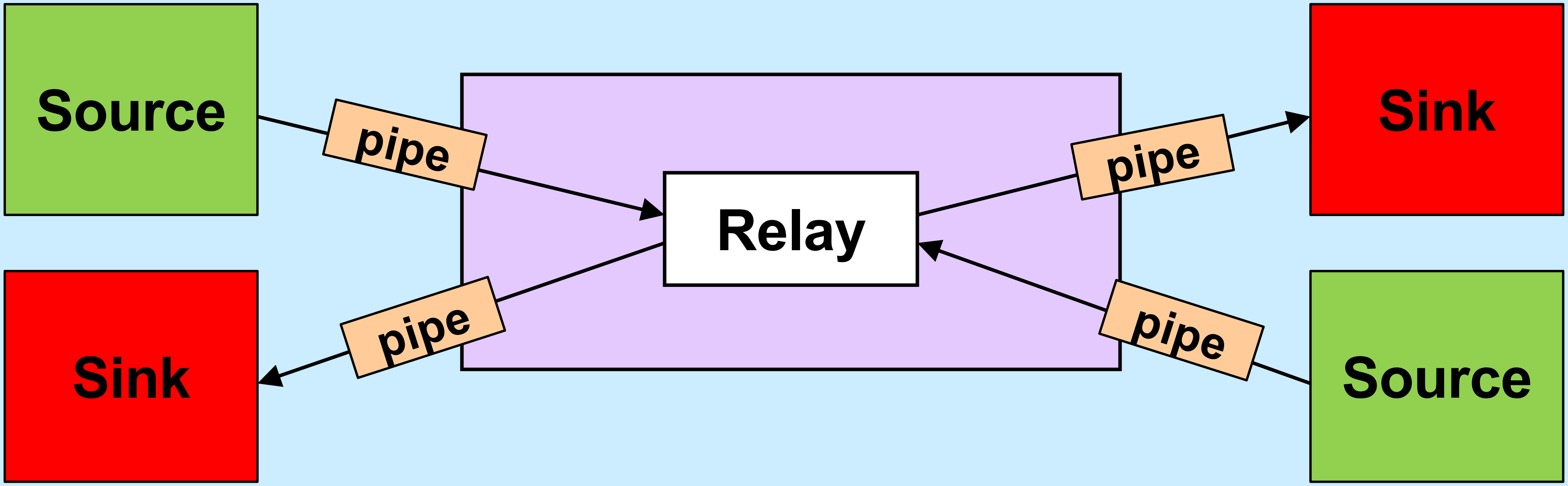
- A thread is a virtual processor
  - an independent agent executing instructions
- Multiple threads
  - multiple independent agents executing instructions

Why Threads?



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

A Simple Example



Life Without Threads

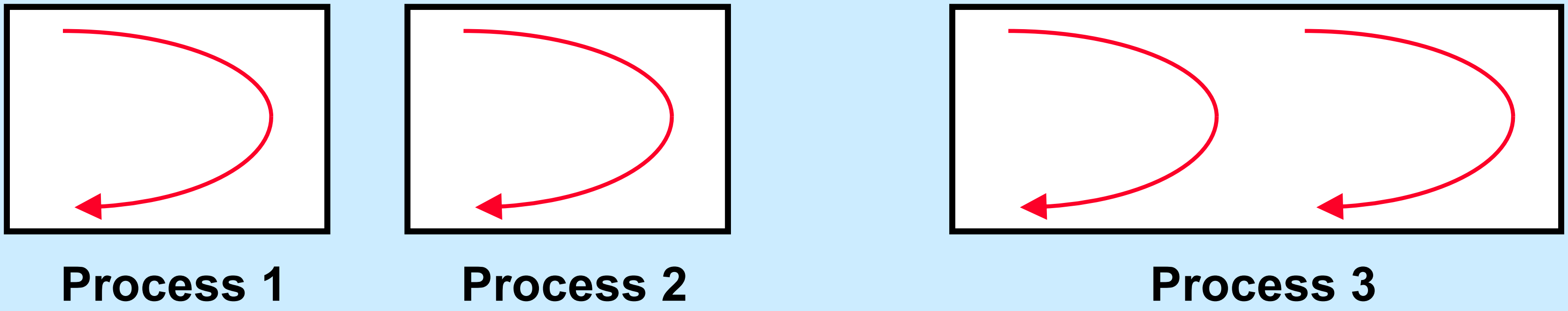
```
void relay(int left, int right) {
    fd_set rd, wr;
    int left_read = 1, right_write = 0;
    int right_read = 1, left_write = 0;
    int sizeLR, sizeRL, wret;
    char bufLR[BSIZE], bufRL[BSIZE];
    char *bufpR, *bufpL;
    int maxFD = max(left, right) + 1;

    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;
}
```

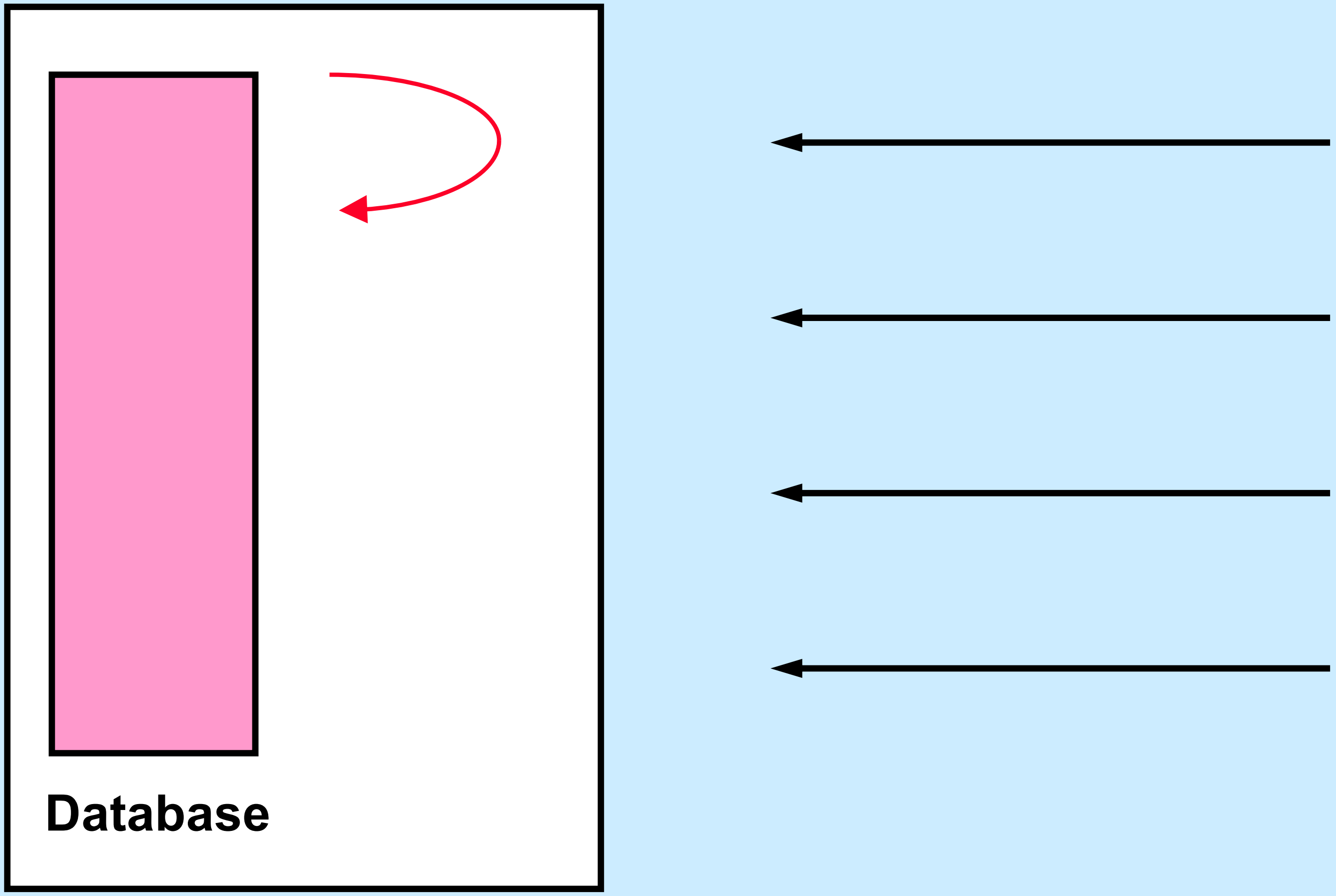
Life With Threads

```
 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



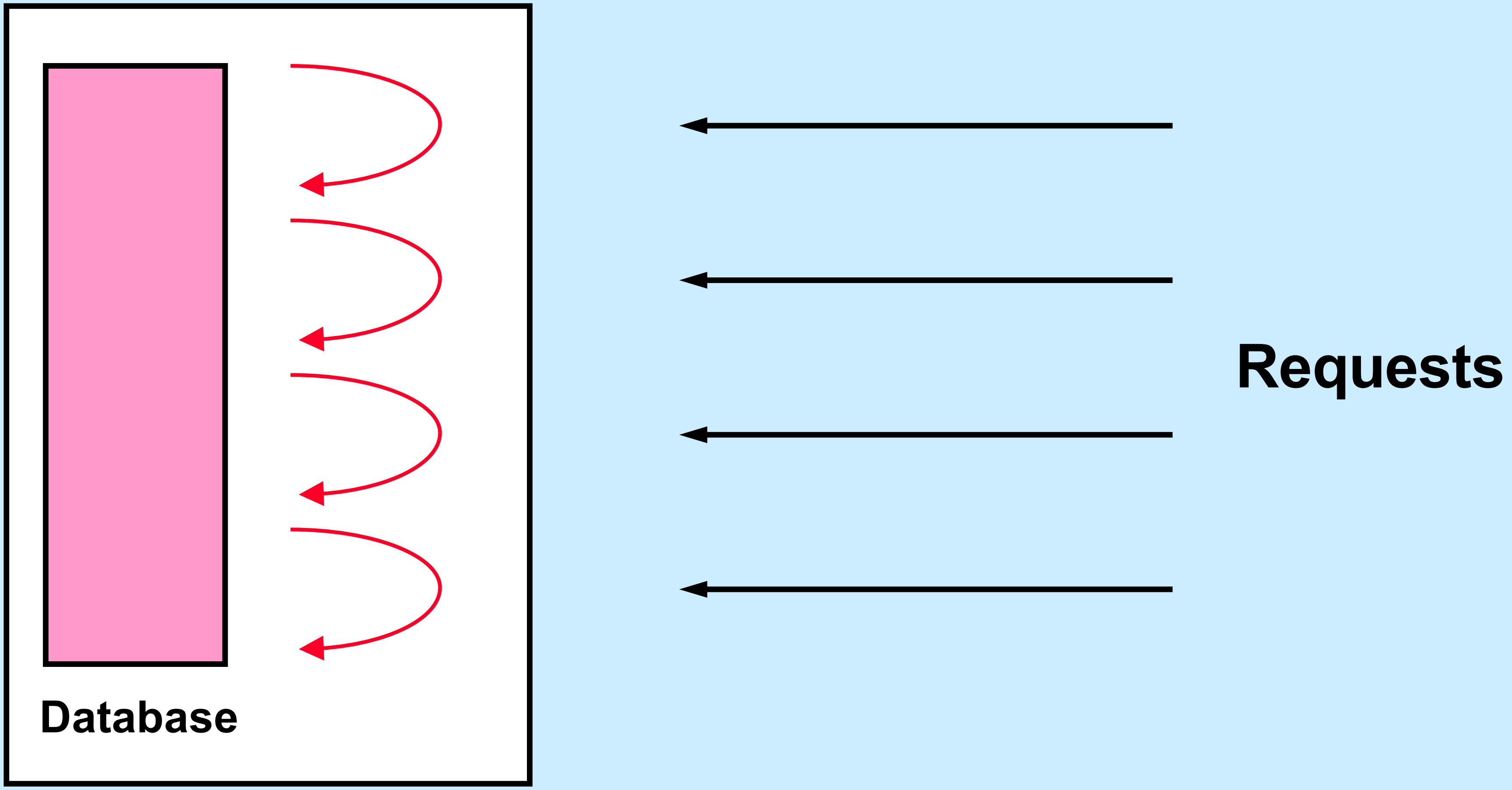
Single-Threaded Database Server



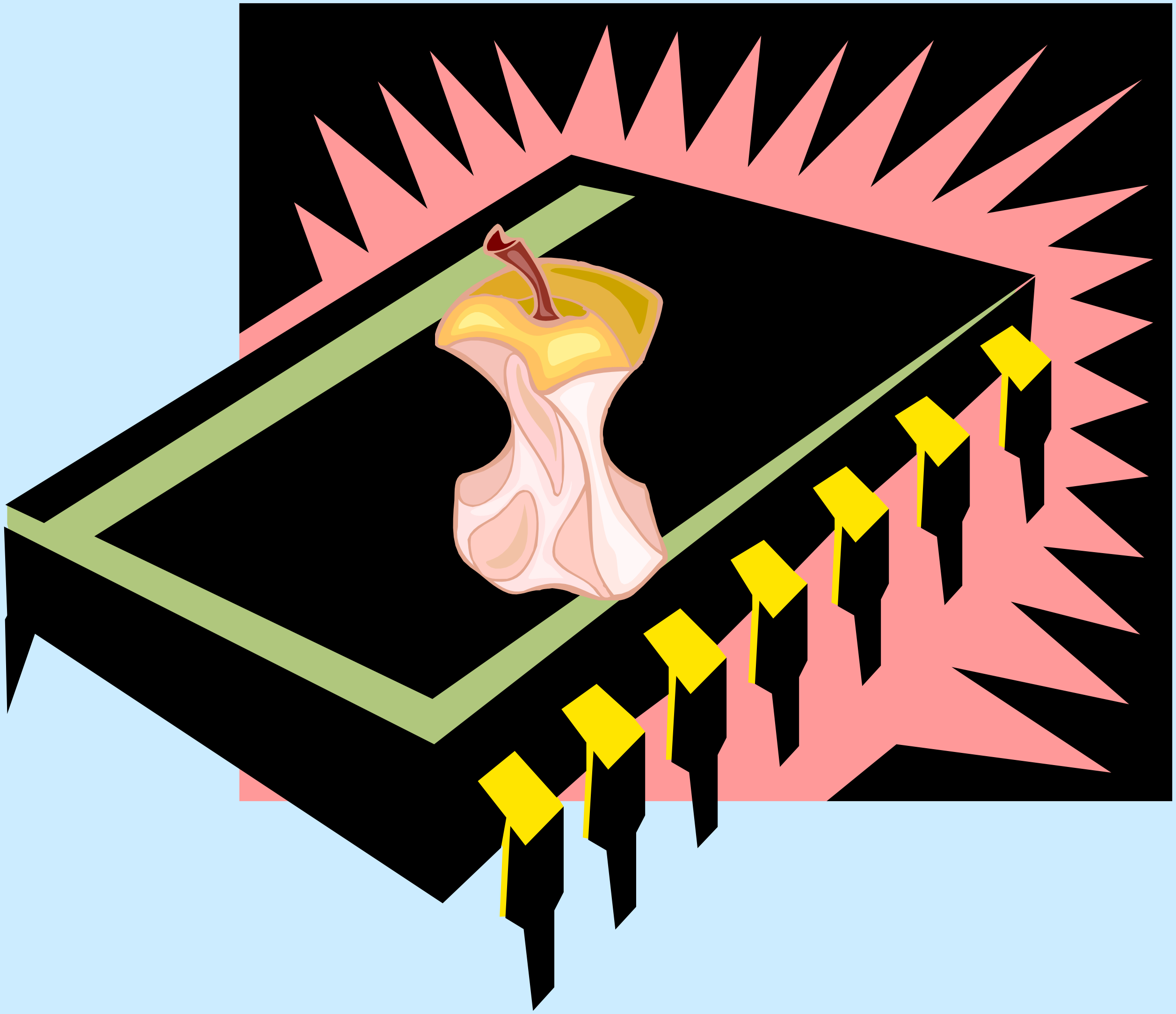
Database

Requests

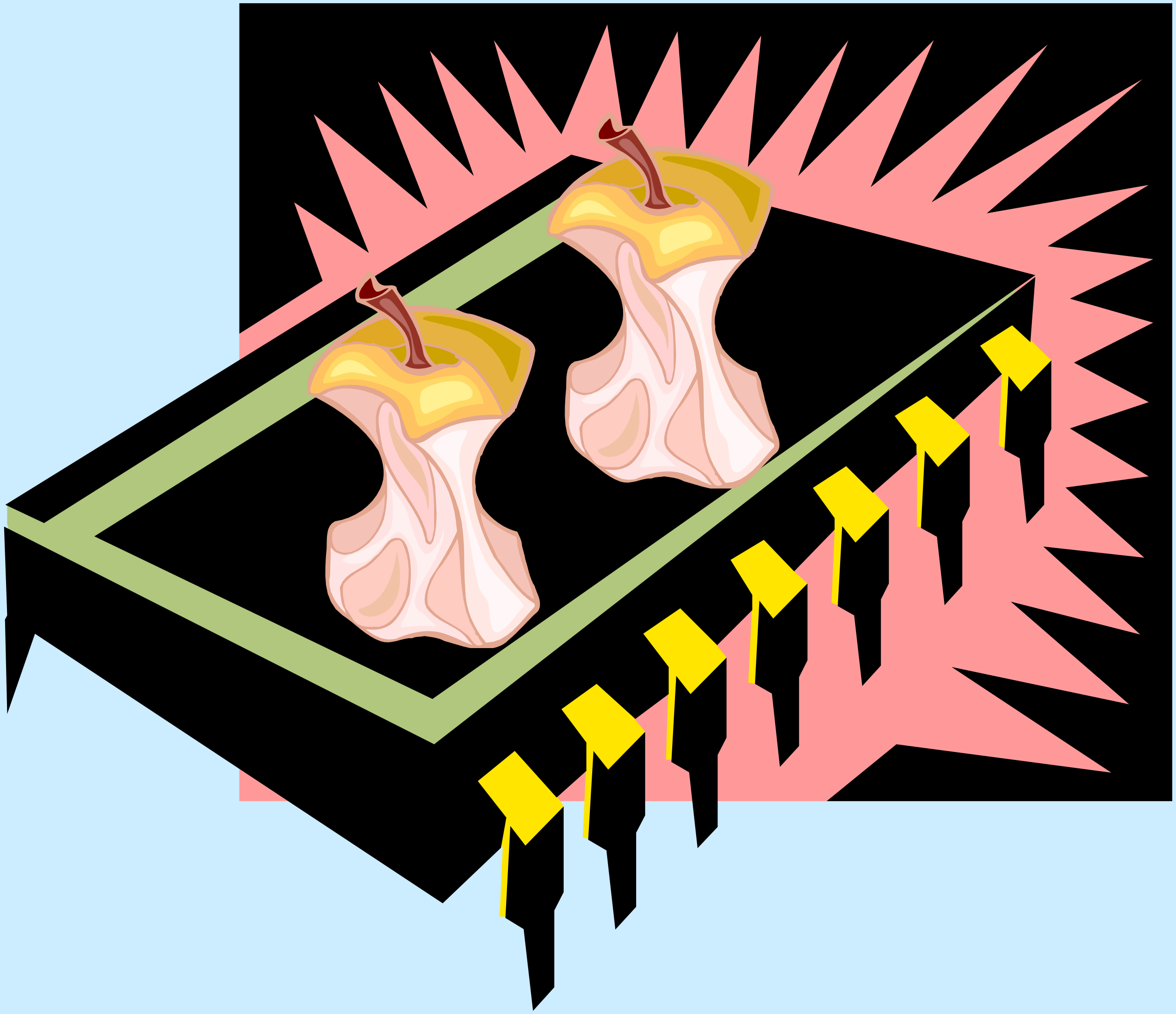
Multithreaded Database Server



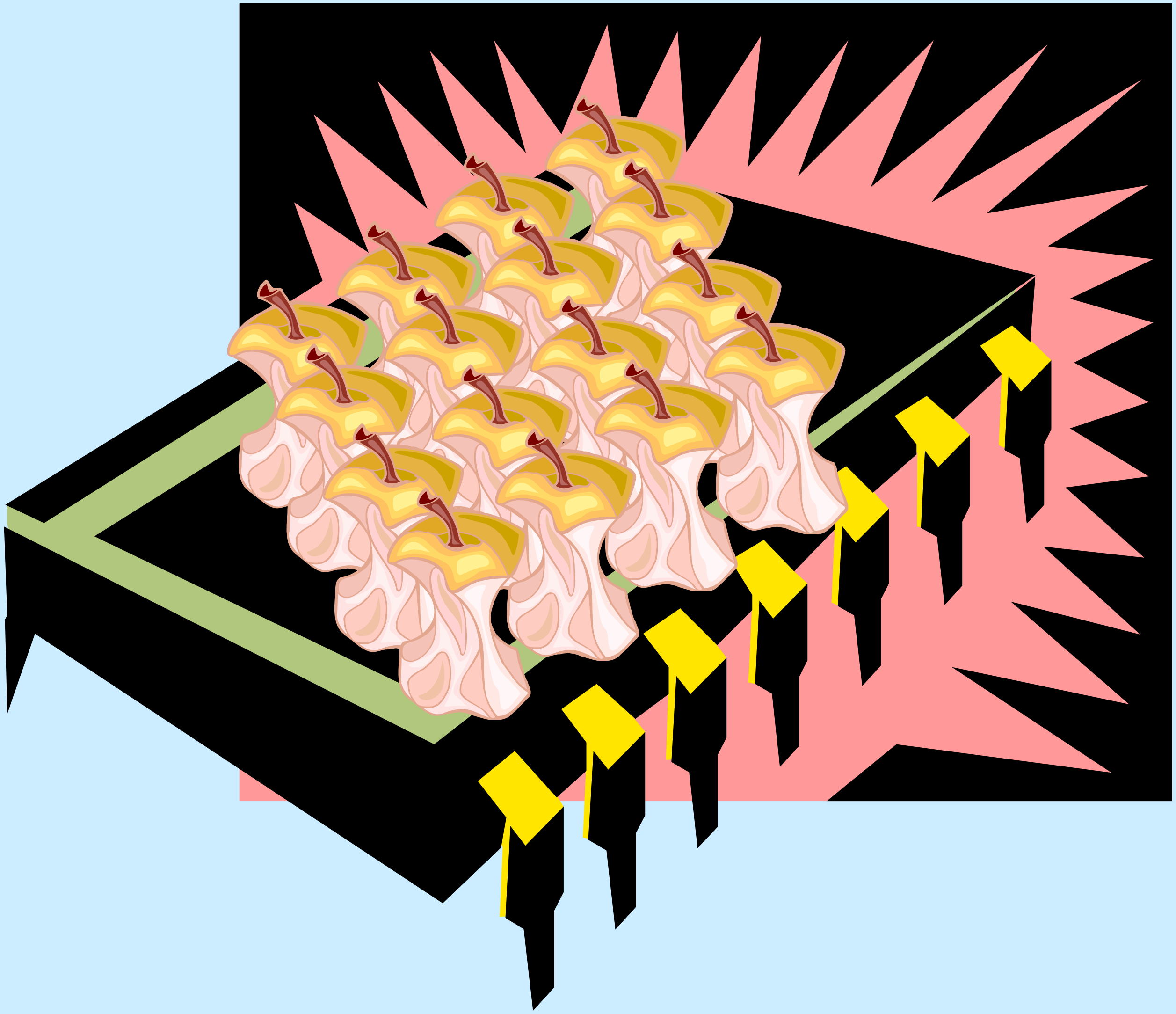
Single-Core Chips



Dual-Core Chips



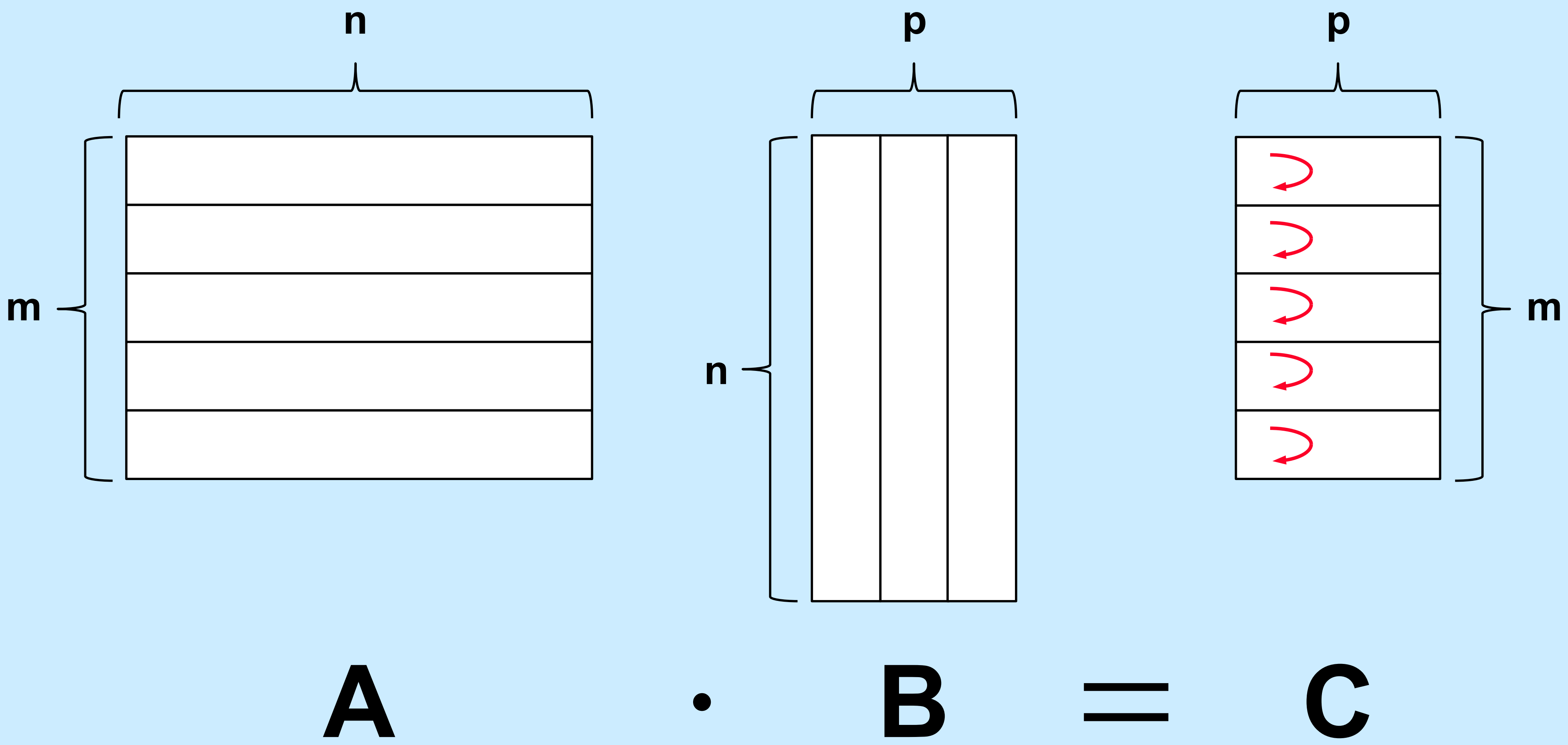
Multi-Core Chips



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 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
```

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);
}
```

Compiling It

```
% gcc –o mat mat.c -pthread
```

Termination

```
pthread_exit((void *) value);

return((void *) value);


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

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 ) {
  ...
}
```

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  ...
}
```

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);
}
```

Multiple Arguments

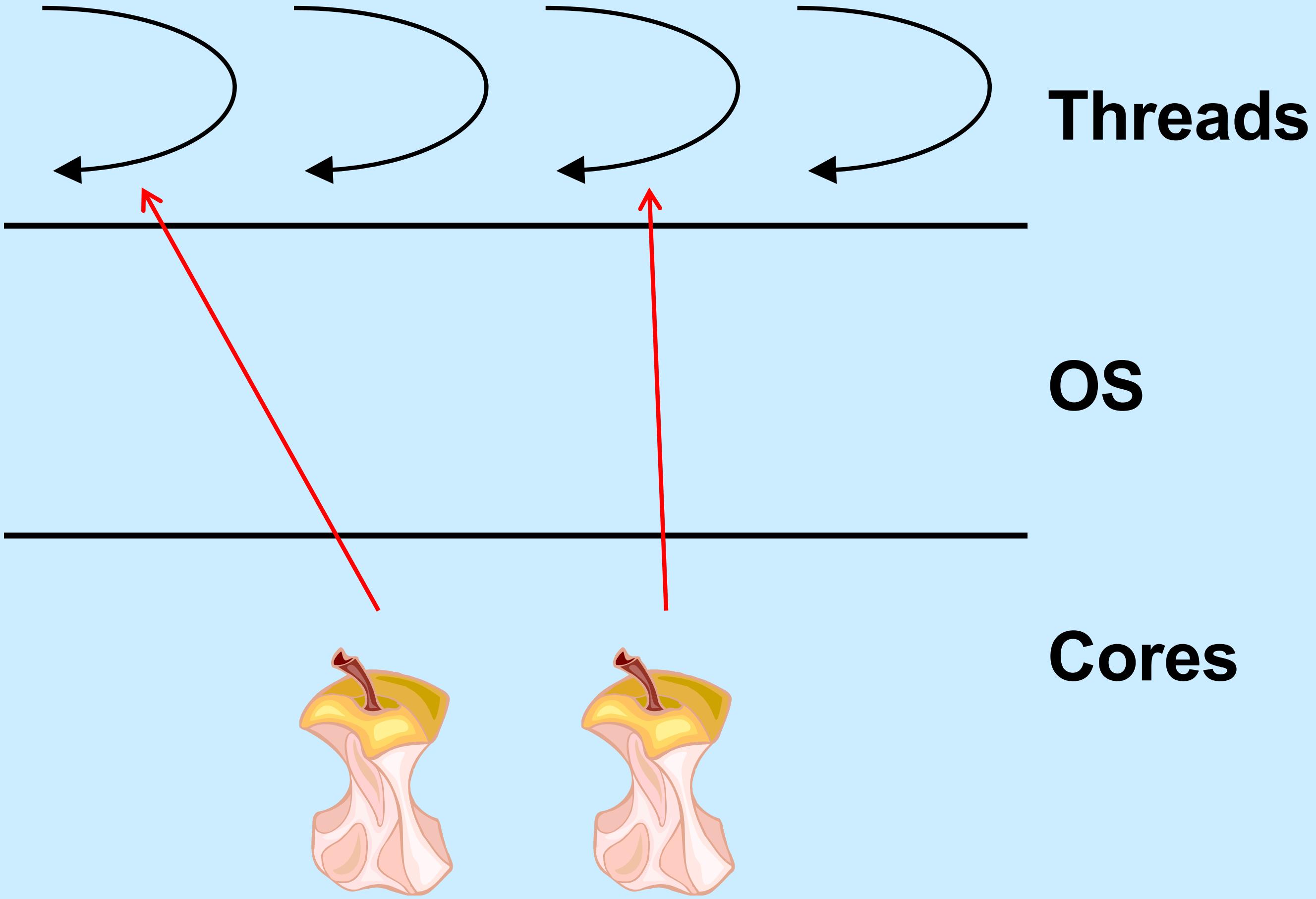
```
typedef struct args {
  int src;
  int dest;
} args_t;

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

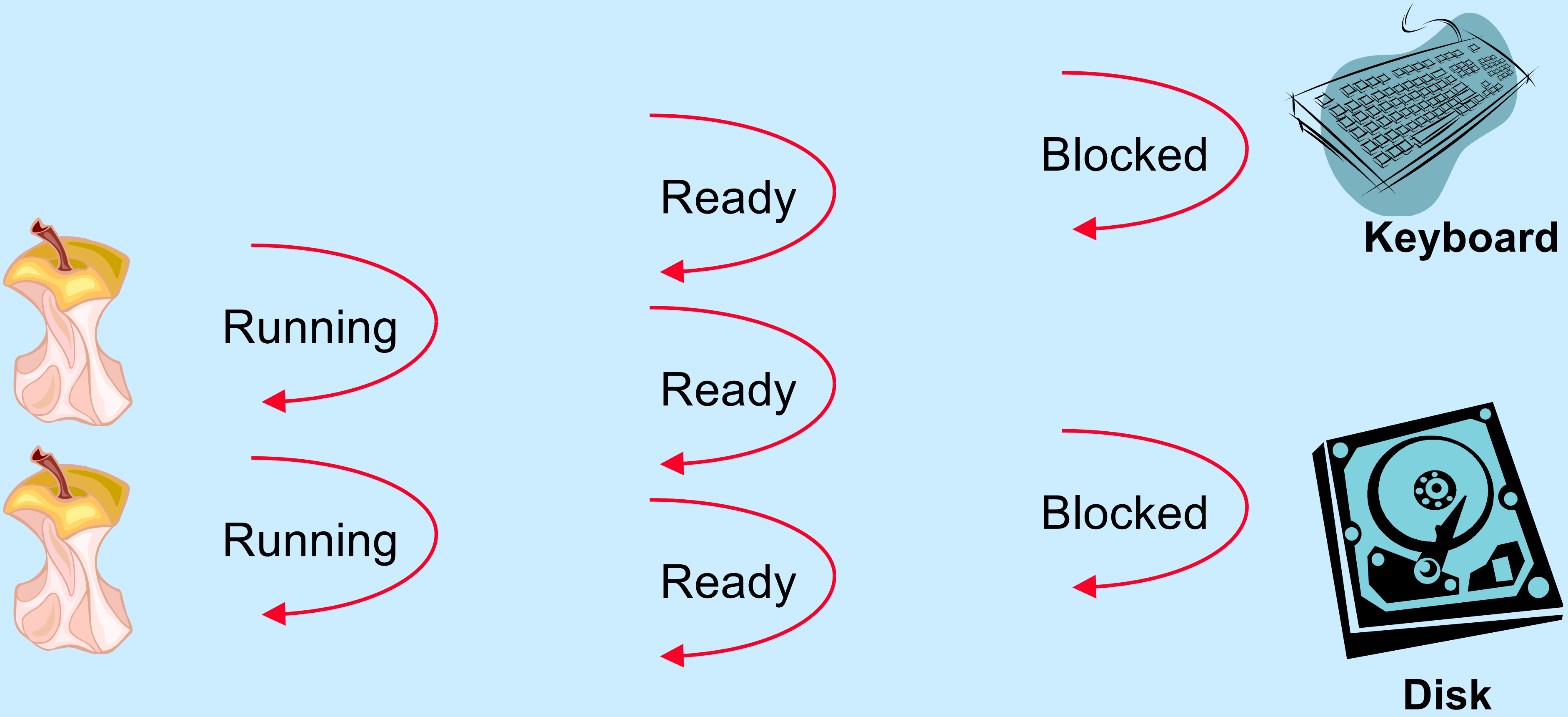
Does this work?

a) yes
b) no

Execution



Multiplexing Processors



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;
}
```

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

Problem

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

  …


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

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);

...
```

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);

...
```