CS 33

Network Programming (2)

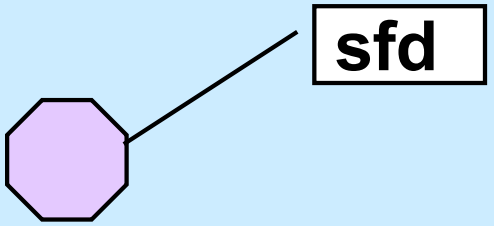
Reliable Communication

- The promise …
  - what is sent is received
  - order is preserved
- Set-up is required
  - two parties agree to communicate
  - within the implementation of the protocol:
    - » each side keeps track of what is sent, what is received
    - » received data is acknowledged
    - » unack'd data is re-sent
- The standard scenario
  - server receives connection requests
  - client makes connection requests

Streams in the Inet Domain (1)

- **Server steps**
  - **1) create socket**

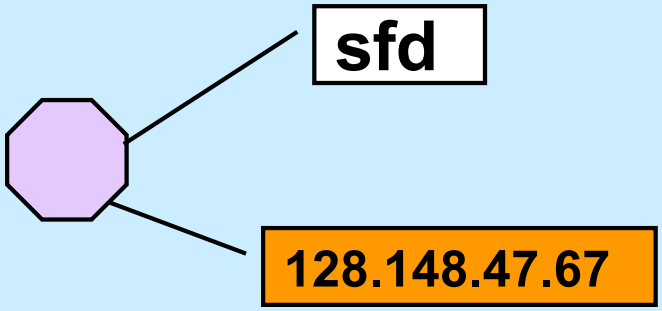
```
sfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (2)

- **Server steps**
  - **2) bind name to socket**

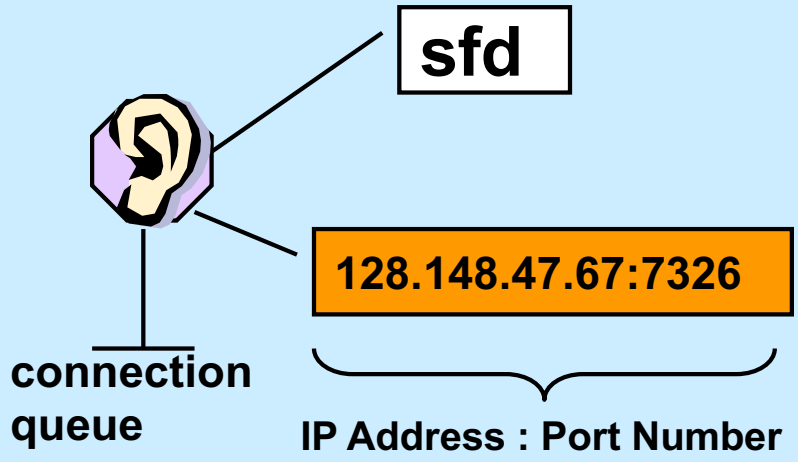
```
bind(sfd,
   (struct sockaddr *)&my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (3)

- **Server steps**
  - **3) put socket in "listening mode"**

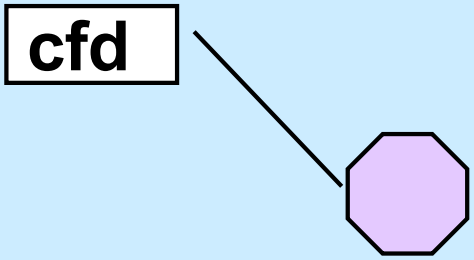
```
int listen(int sfd, int MaxQueueLength);
```



Streams in the Inet Domain (4)

- **Client steps**
  - **1) create socket**

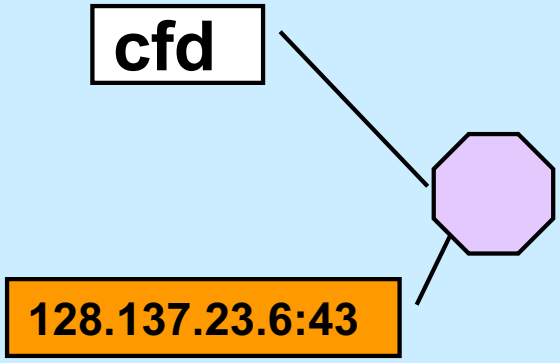
```
cfd = socket(AF_INET, SOCK_STREAM, 0);
```



Streams in the Inet Domain (5)

- **Client steps**
  - **2) bind name to socket**

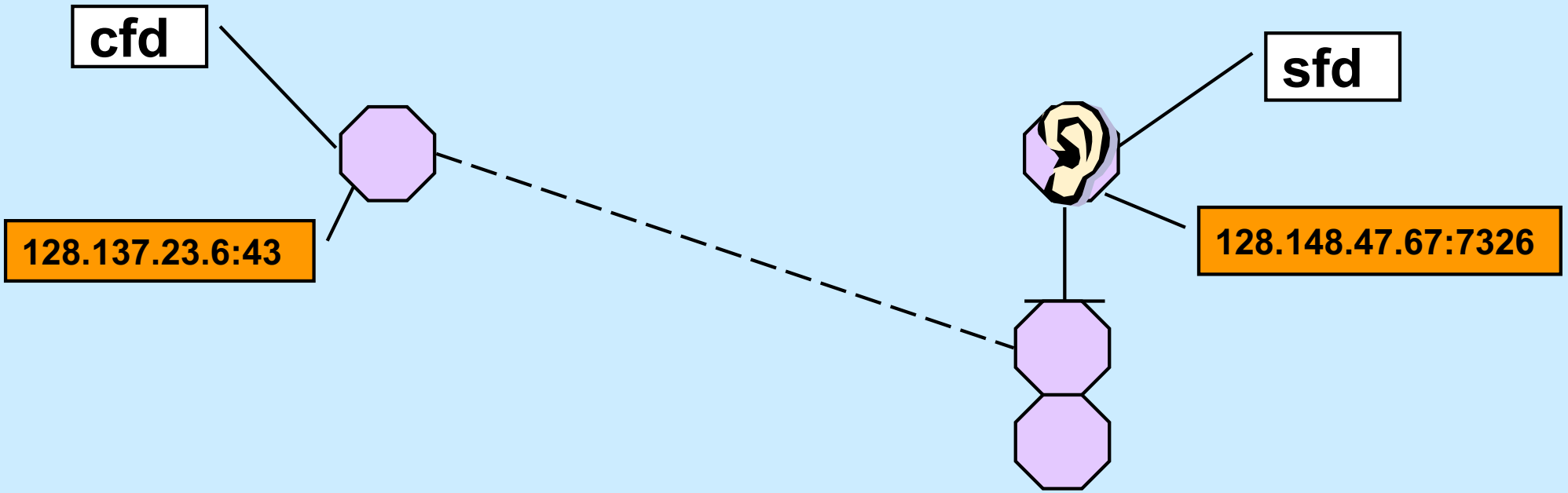
```
bind(cfd,
   (struct sockaddr *)&my_addr, sizeof(my_addr));
```



Streams in the Inet Domain (6)

- **Client steps**
  - **3) connect to server**

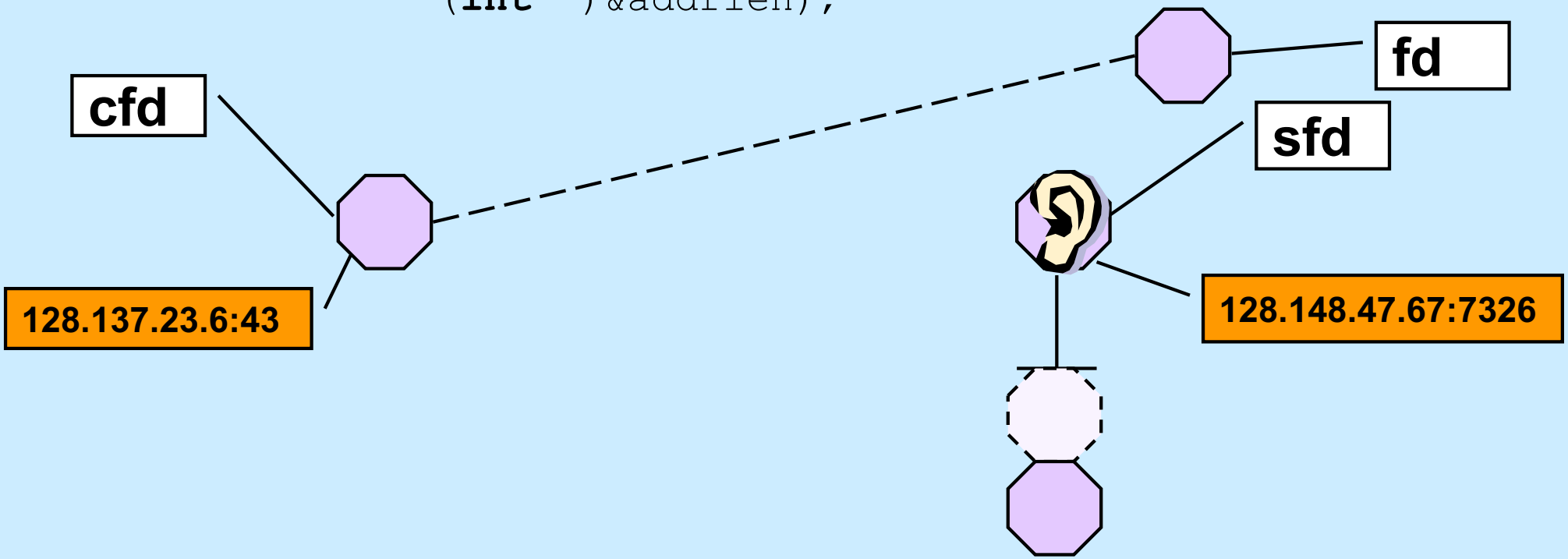
```
connect(cfd, (struct sockaddr *)&server_addr,
    sizeof(server_addr));
```



Streams in the Inet Domain (7)

- **Server steps**
  - **4) accept connection**

```
fd = accept((int)sfd, (struct sockaddr *)addr,
     (int *)&addrlen);
```



TCP Server (1)

```
int main(int argc, char *argv[ ]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: port\n");
        exit(1);
    }

    int lsocket;
    struct addrinfo tcp_hints;
    struct addrinfo *result;
```

TCP Server (2)

```
memset(&tcp_hints, 0, sizeof(tcp_hints));
tcp_hints.ai_family = AF_INET;
tcp_hints.ai_socktype = SOCK_STREAM;
tcp_hints.ai_flags = AI_PASSIVE;

int err;
if ((err = getaddrinfo(NULL, argv[1], &tcp_hints,
      &result)) != 0) {
    fprintf(stderr,"%s\n", gai_strerror(err));
    exit(1);
}
```

TCP Server (3)

```
struct addrinfo *r;
for (r = result; r != NULL; r = r->ai_next) {
    if ((lsocket =
          socket(r->ai_family, r->ai_socktype,
          r->ai_protocol)) < 0) {
        continue;
    }
    if (bind(lsocket, r->ai_addr, r->ai_addrlen) >= 0) {
        break;
    }
    close(lsocket);
}
```

TCP Server (4)

```
if (r == NULL) {
    fprintf(stderr, "Could not find local interface %s\n");
    exit(1);
}
freeaddrinfo(result);


if (listen(lsocket, 50) < 0) {
    perror("listen");
    exit(1);
}
```

TCP Server (5)

```
while (1) {
    int csock;
    struct sockaddr client_addr;
    int client_len = sizeof(client_addr);

    csock = accept(lsocket, &client_addr, &client_len);
    if (csock == -1) {
        perror("accept");
        exit(1);
    }
```

TCP Server (6)

```
char host_name[256];
char serv_name[256];
int err;
if ((err = getnameinfo(&client_addr,
      client_len, host_name, sizeof(host_name),
      serv_name, sizeof(serv_name), 0))) {
    fprintf(stderr, "%s/n", gai_strerror(err));
    exit(1);
}
printf("received connection from %s port %s\n",
      host_name, serv_name);
```

TCP Server (7)

```
        switch (fork()) {
        case -1:
            perror("fork");
            exit(1);
        case 0:
            serve(csock);
            exit(0);
        default:
            close(csock);
            break;
        }
    }
    return 0;
}
```

TCP Server (8)

```
void serve(int fd) {
    char buf[1024];
    int count;

    while ((count = read(fd, buf, 1024)) > 0) {
        write(1, buf, count);
    }
    if (count == -1) {
        perror("read");
        exit(1);
    }
    printf("connection terminated\n");
}
```

TCP Client (1)

```
int main(int argc, char *argv[]) {
    int s;
    int sock;
    struct addrinfo hints;
    struct addrinfo *result;
    struct addrinfo *rp;
    char buf[1024];

    if (argc != 3) {
        fprintf(stderr, "Usage: tcpClient host port\n");
        exit(1);
    }
```


TCP Client (2)

```
memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

if ((s=getaddrinfo(argv[1], argv[2], &hints, &result))
      != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(1);
}
```

TCP Client (3)

```
for (rp = result; rp != NULL; rp = rp->ai_next) {
    if ((sock = socket(rp->ai_family, rp->ai_socktype,
          rp->ai_protocol)) < 0) {
        continue;
    }
    if (connect(sock, rp->ai_addr, rp->ai_addrlen) >= 0) {
        break;
    }
    close(sock);
}
```

TCP Client (4)

```
if (rp == NULL) {
    fprintf(stderr, "Could not connect to %s\n", argv[1]);
    exit(1);
}
freeaddrinfo(result);
```

TCP Client (5)

```
    while(fgets(buf, 1024, stdin) != 0) {
        if (write(sock, buf, strlen(buf)) < 0) {
            perror("write");
            exit(1);
        }
    }
    return 0;
}
```

Quiz 1

The previous slide contains

```
write(sock, buf, strlen(buf))
```

If data is lost and must be retransmitted

a) **write returns an error so the caller can retransmit the data.**

b) **nothing happens as far as the application code is concerned, the data is retransmitted automatically.**

c) **the receiving application has to tell the sending application to retransmit.**

Quiz 2

A previous slide contains

```
write(sock, buf, strlen(buf))
```

We lose the connection to the other party (perhaps a network cable is cut).

a) write returns an error so the caller can reconnect, if desired.

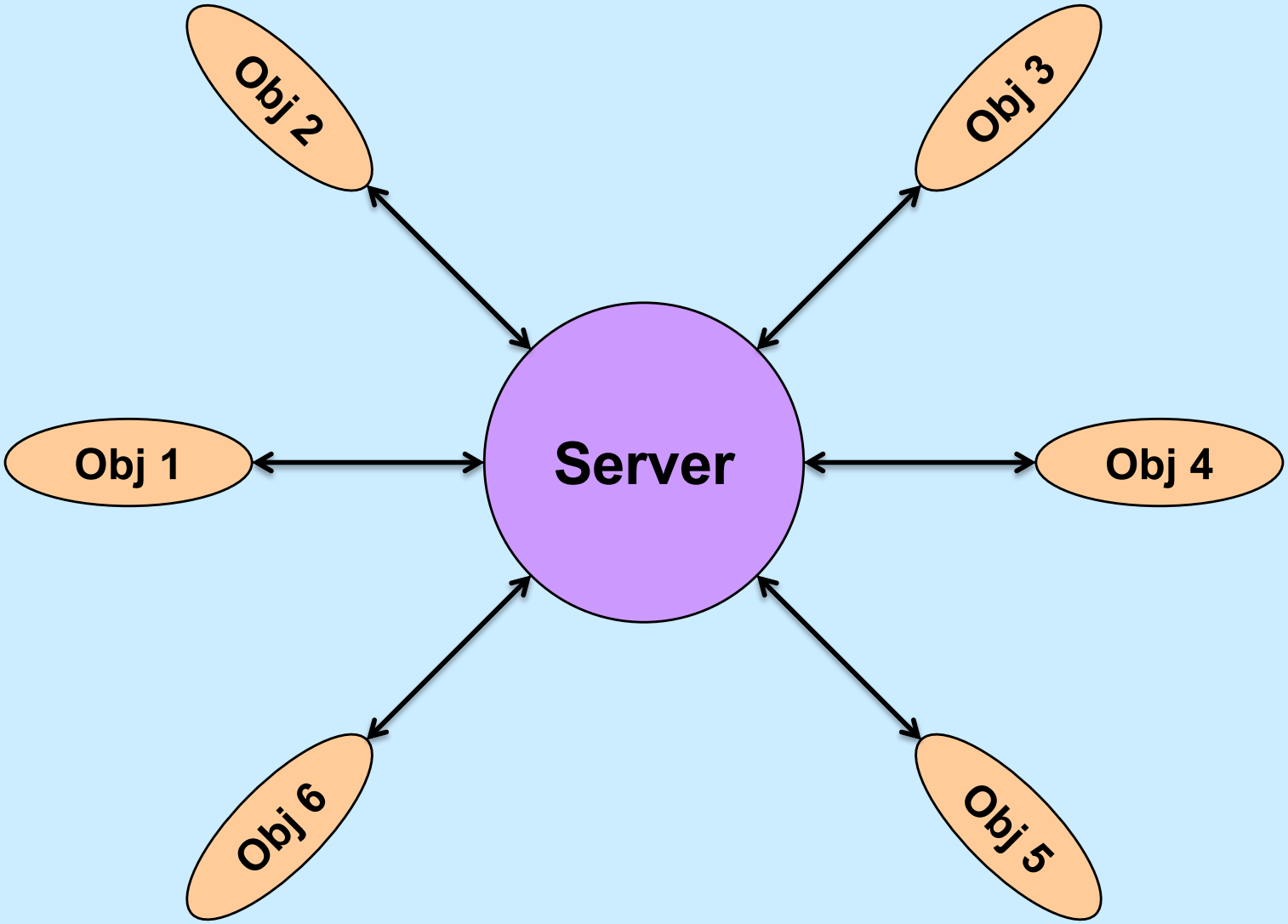
b) nothing happens as far as the application code is concerned, the connection is reestablished automatically.

c) the receiving application has to tell the sending application to reconnect.

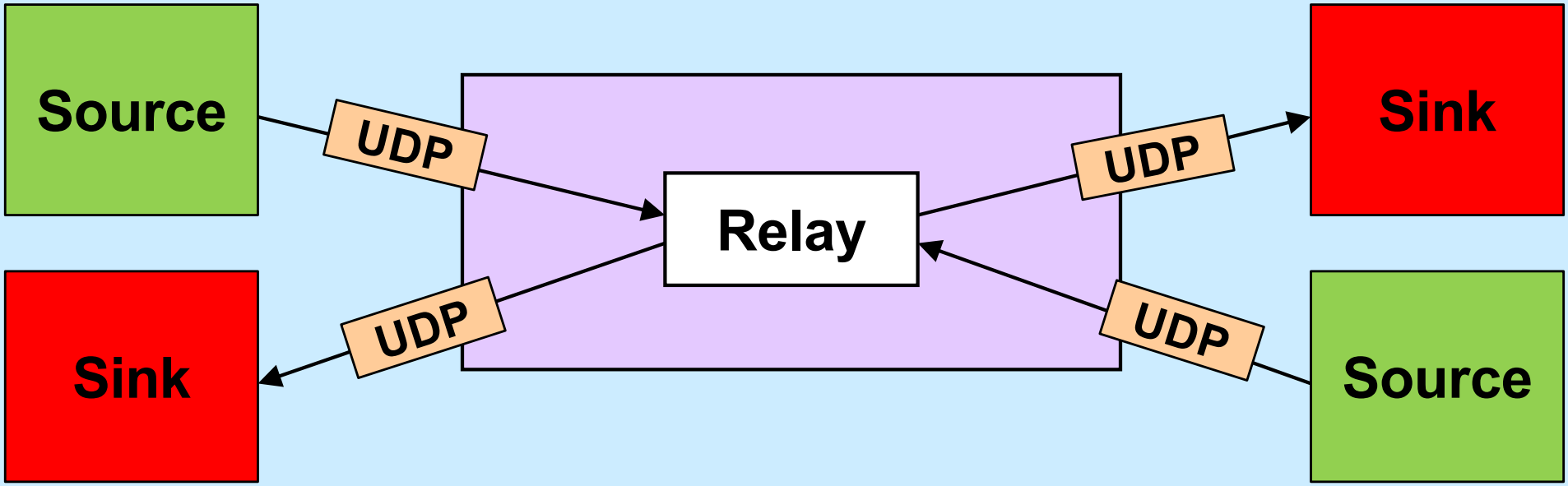
CS 33

Event-Based Programming

Event Handling



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;
}
```

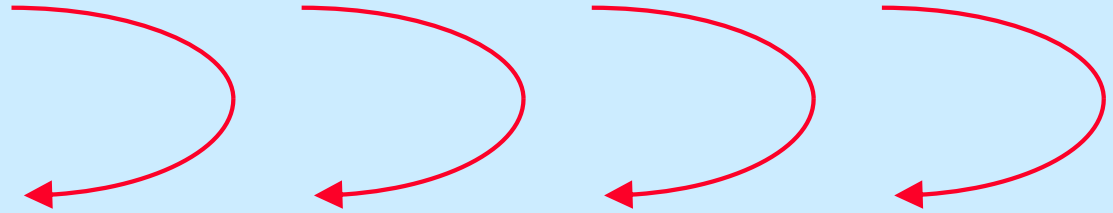
CS 33

Multithreaded Programming (1)

Multithreaded Programming

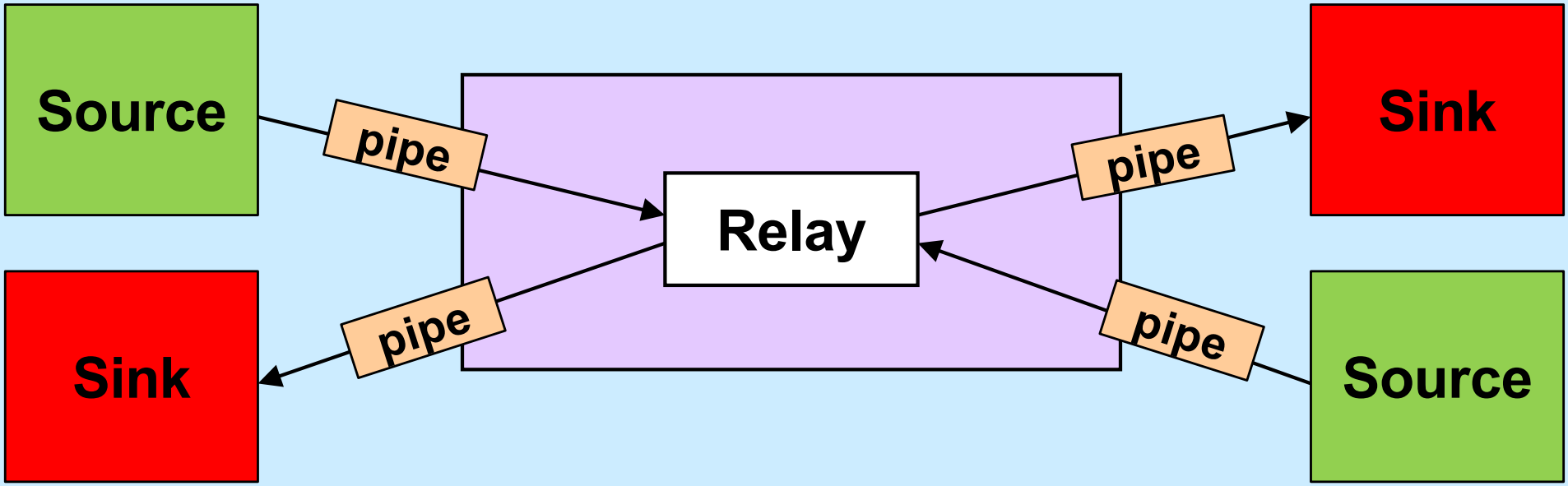
- A thread is a virtual processor
  - an independent agent executing instructions
- Multiple threads
  - multiple independent agents executing instructions

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);
  }
}
```

Quiz 3

The multi-threaded program, compared to the single-threaded program that uses *select*, is

a) always faster

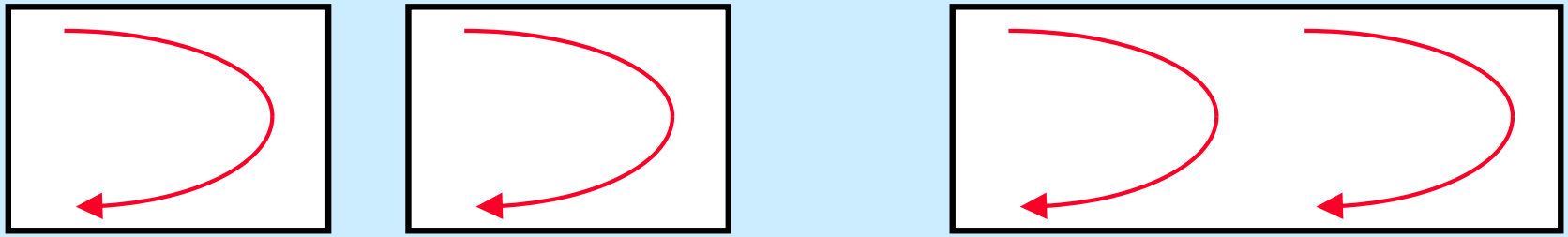
b) always faster if there is more than one processor

c) about the same for one processor; faster for more than one processor

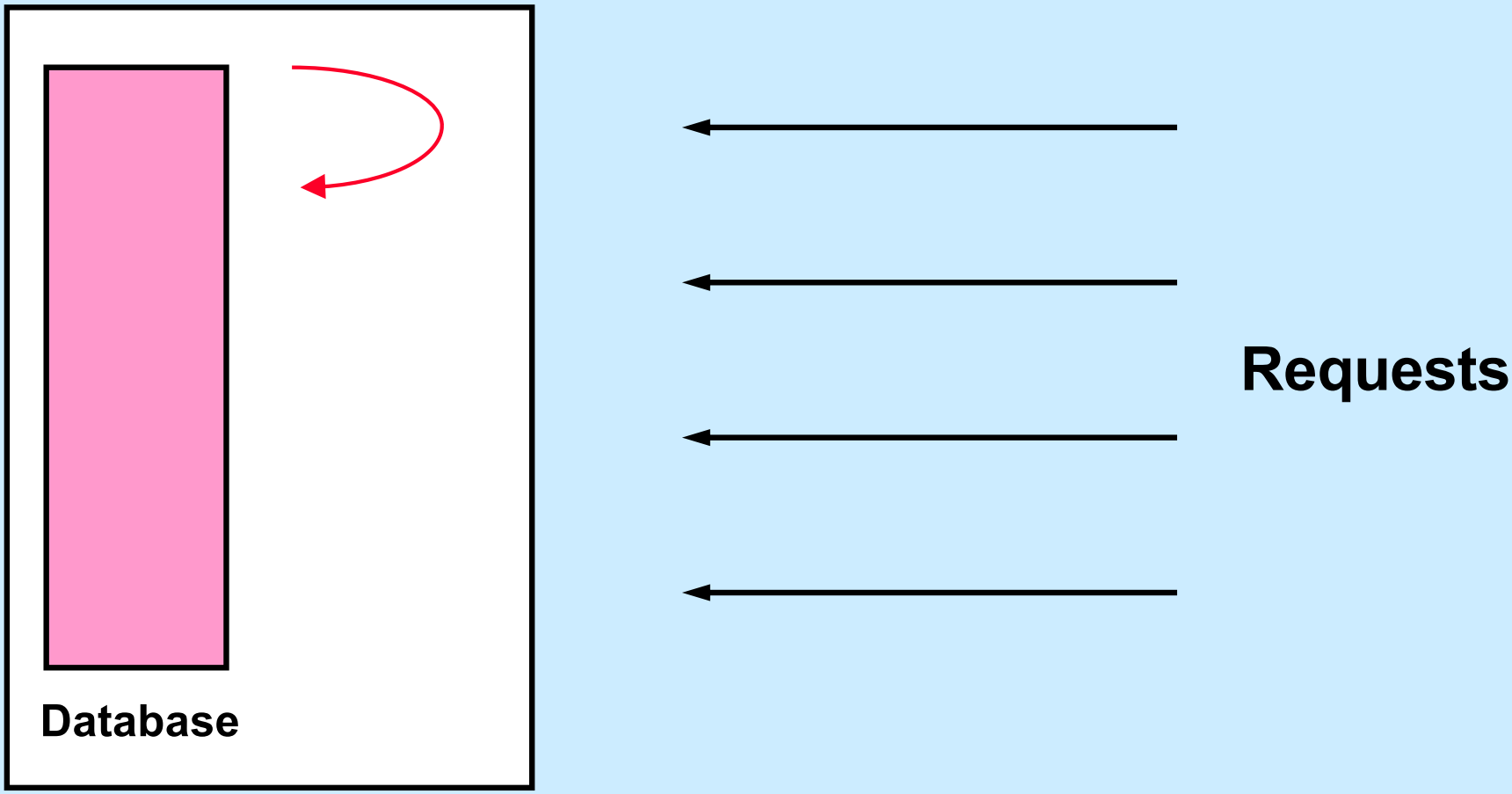
d) often slower

e) always slower

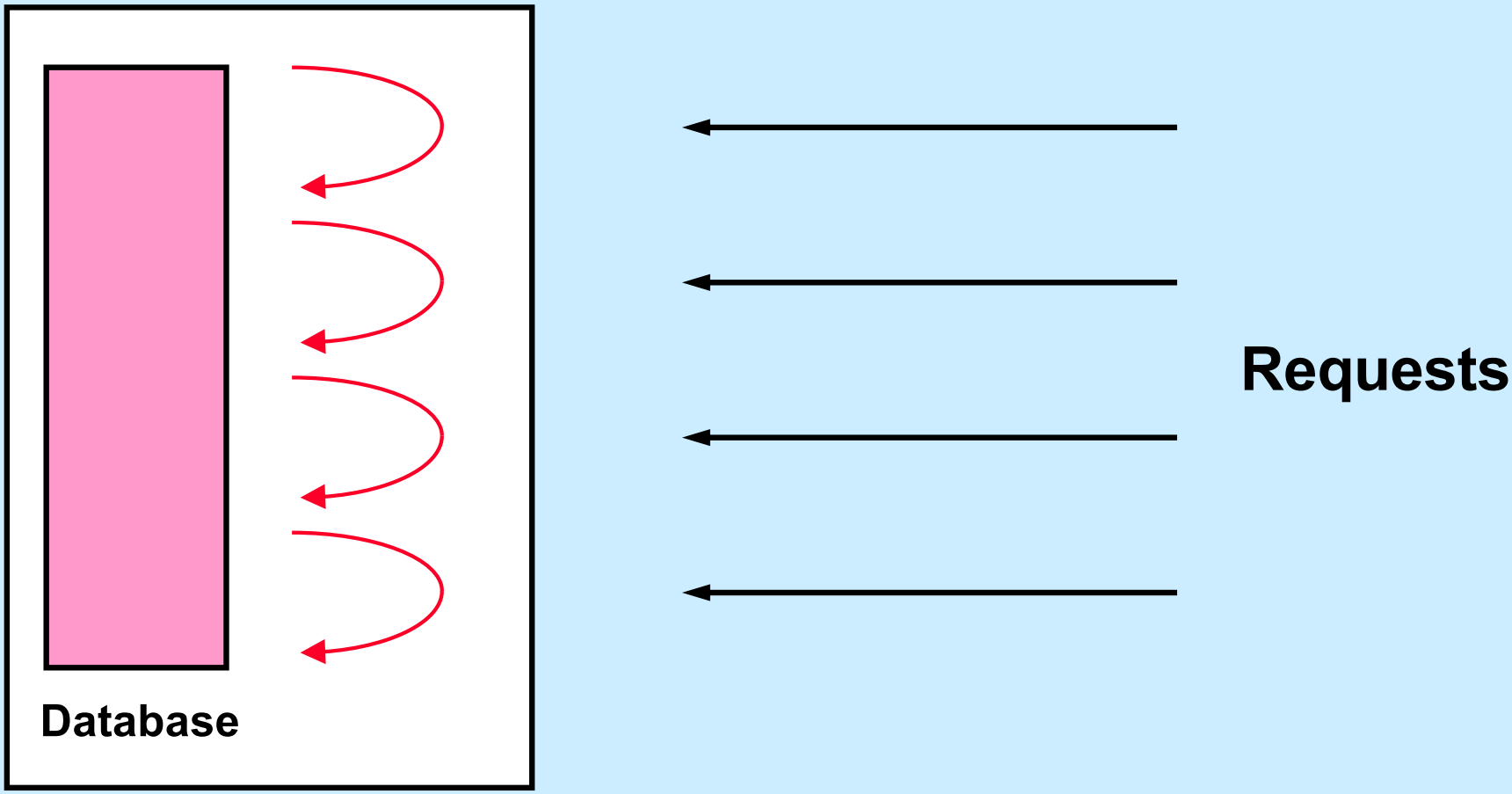
Processes vs. Threads



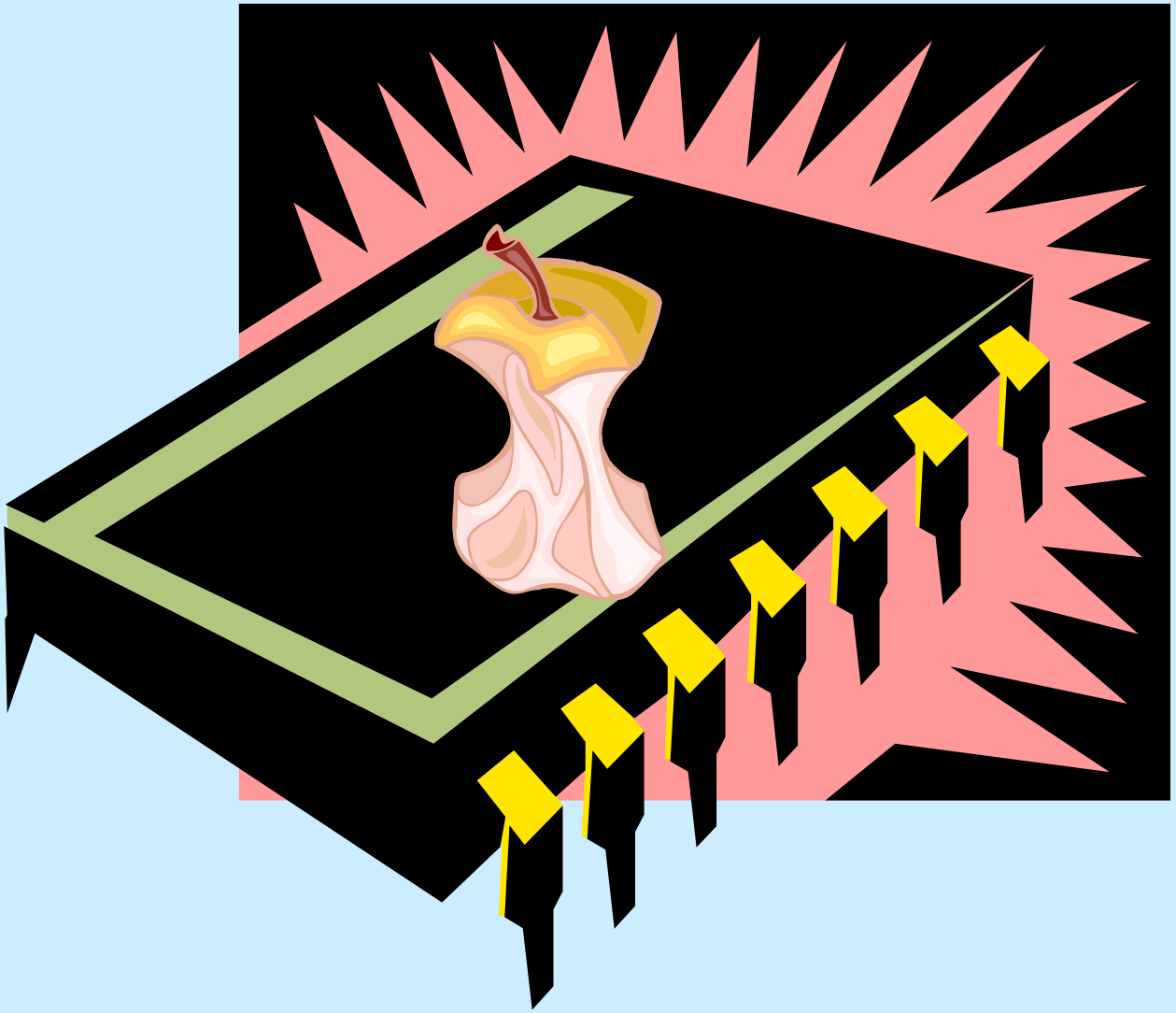
Single-Threaded Database Server



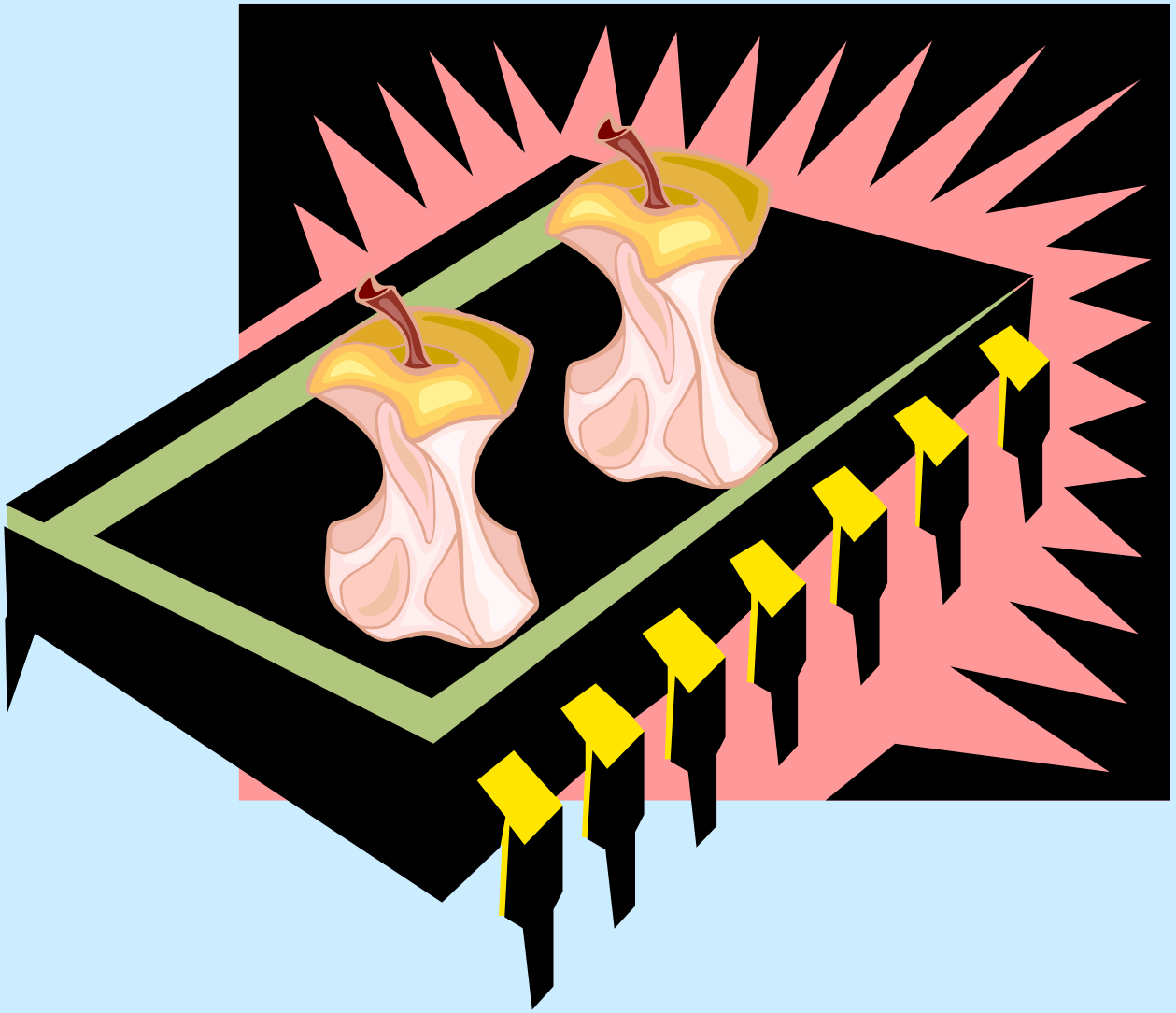
Multithreaded Database Server



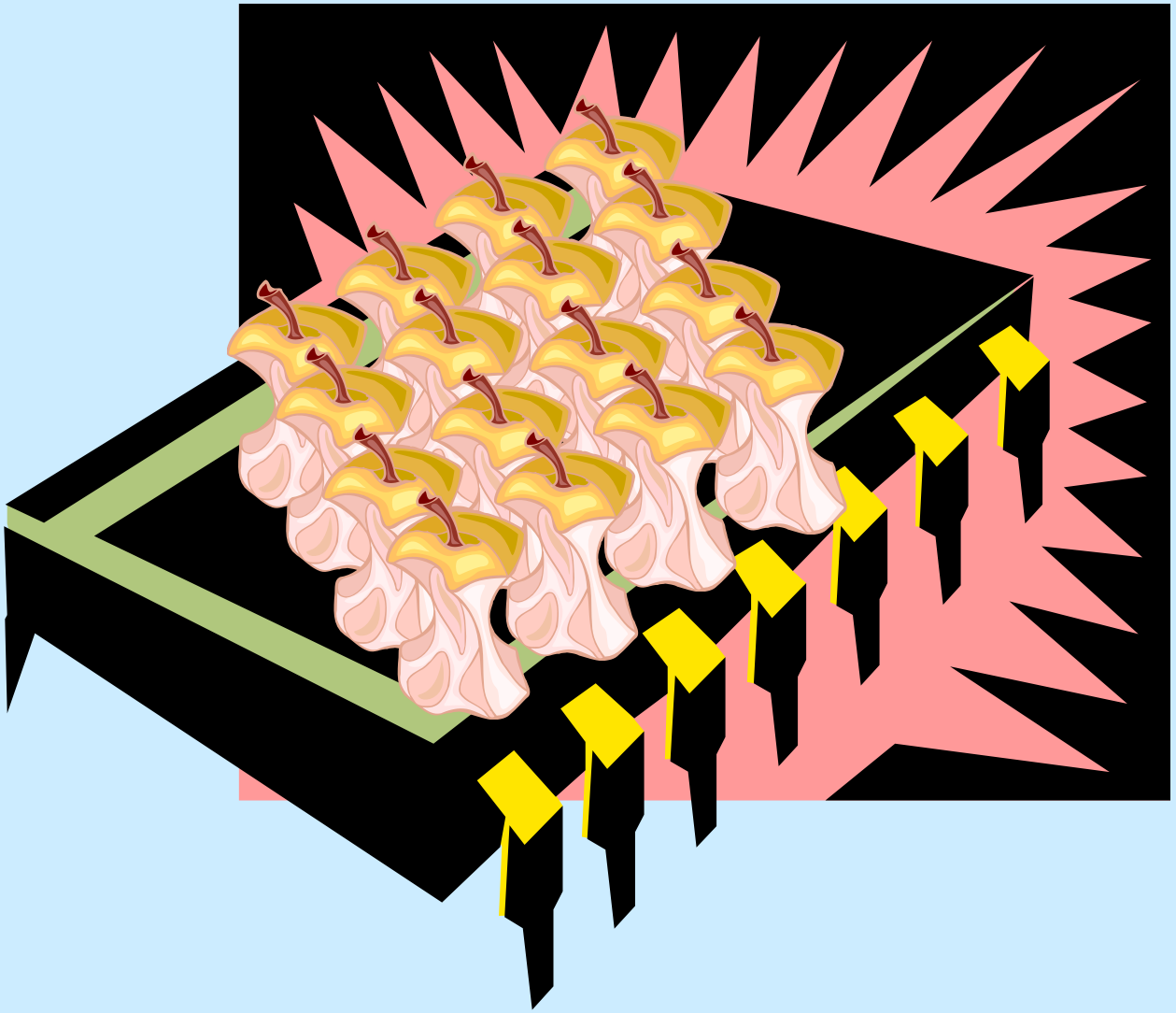
Single-Core Chips



Dual-Core Chips



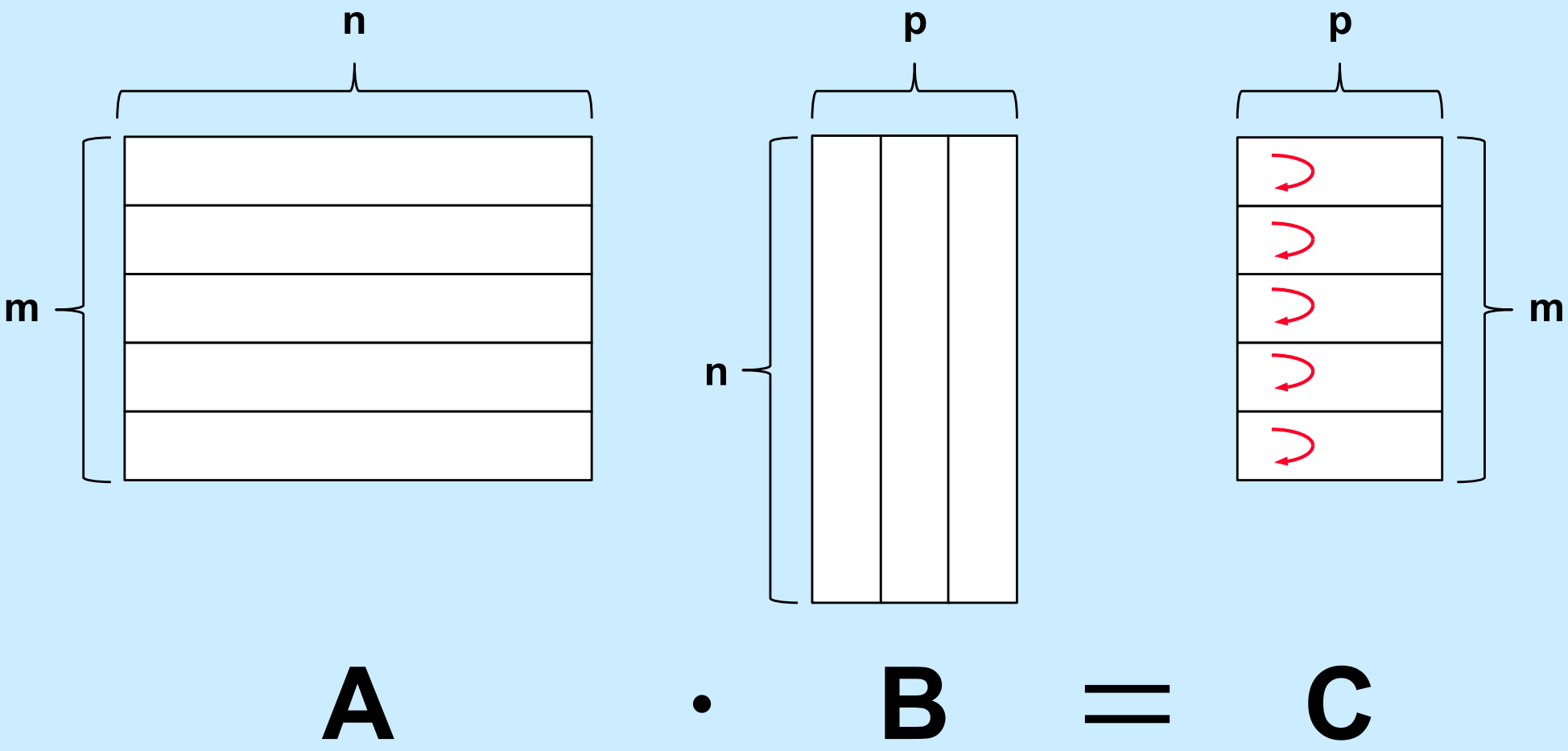
Multi-Core Chips



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

