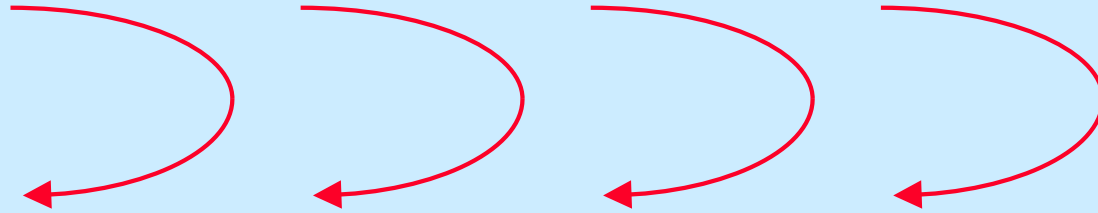


# CS 33

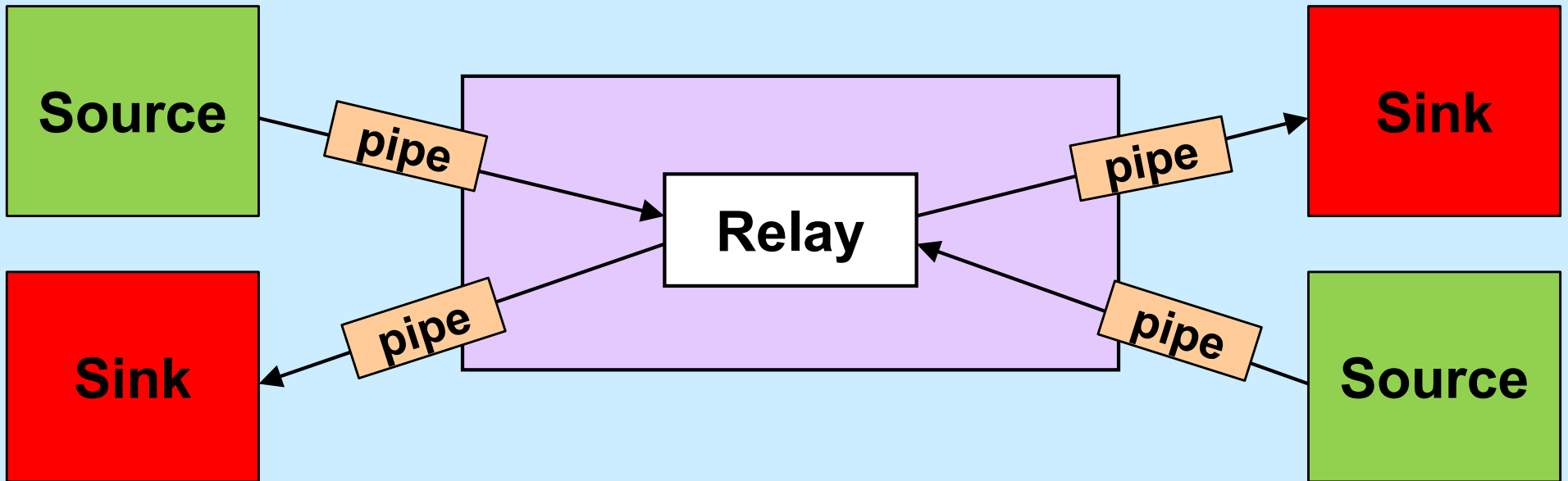
## Multithreaded Programming I

# 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);

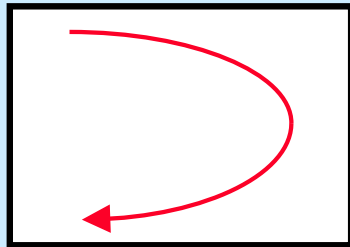
        select(maxFD, &rd, &wr, 0, 0);

        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;
}
```

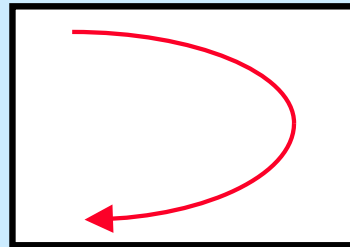
# Life With Threads

```
void copy(int source, int destination) {  
    struct args *targs = args;  
    char buf[BFSIZE];  
  
    while(1) {  
        int len = read(source, buf, BFSIZE);  
        write(destination, buf, len);  
    }  
}
```

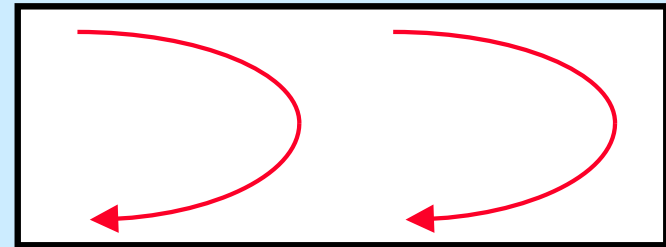
# Processes vs. Threads



Process 1

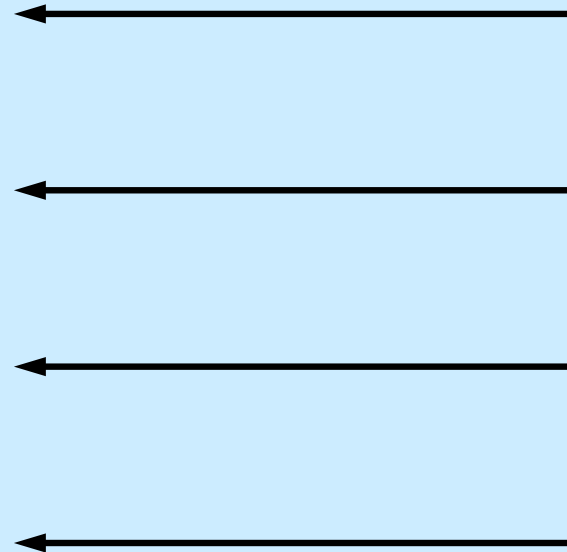
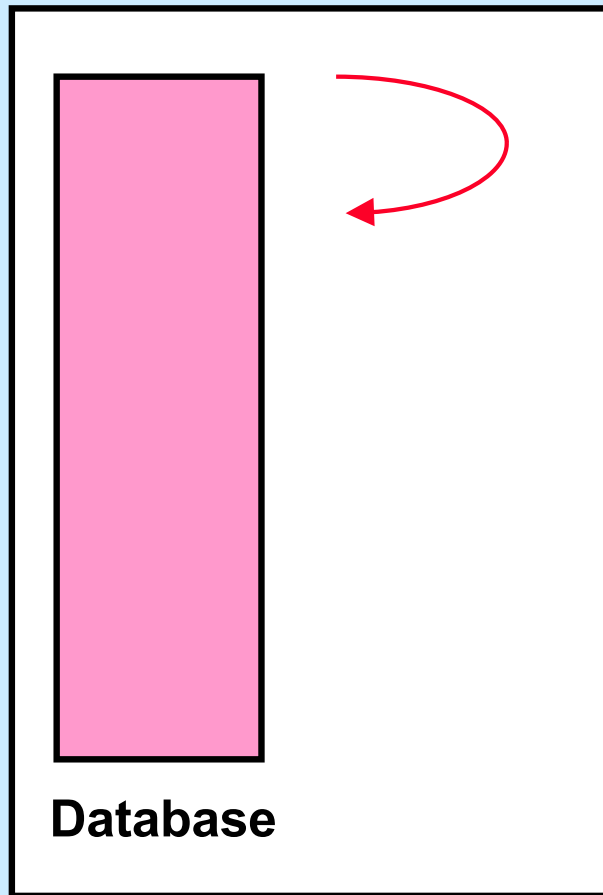


Process 2



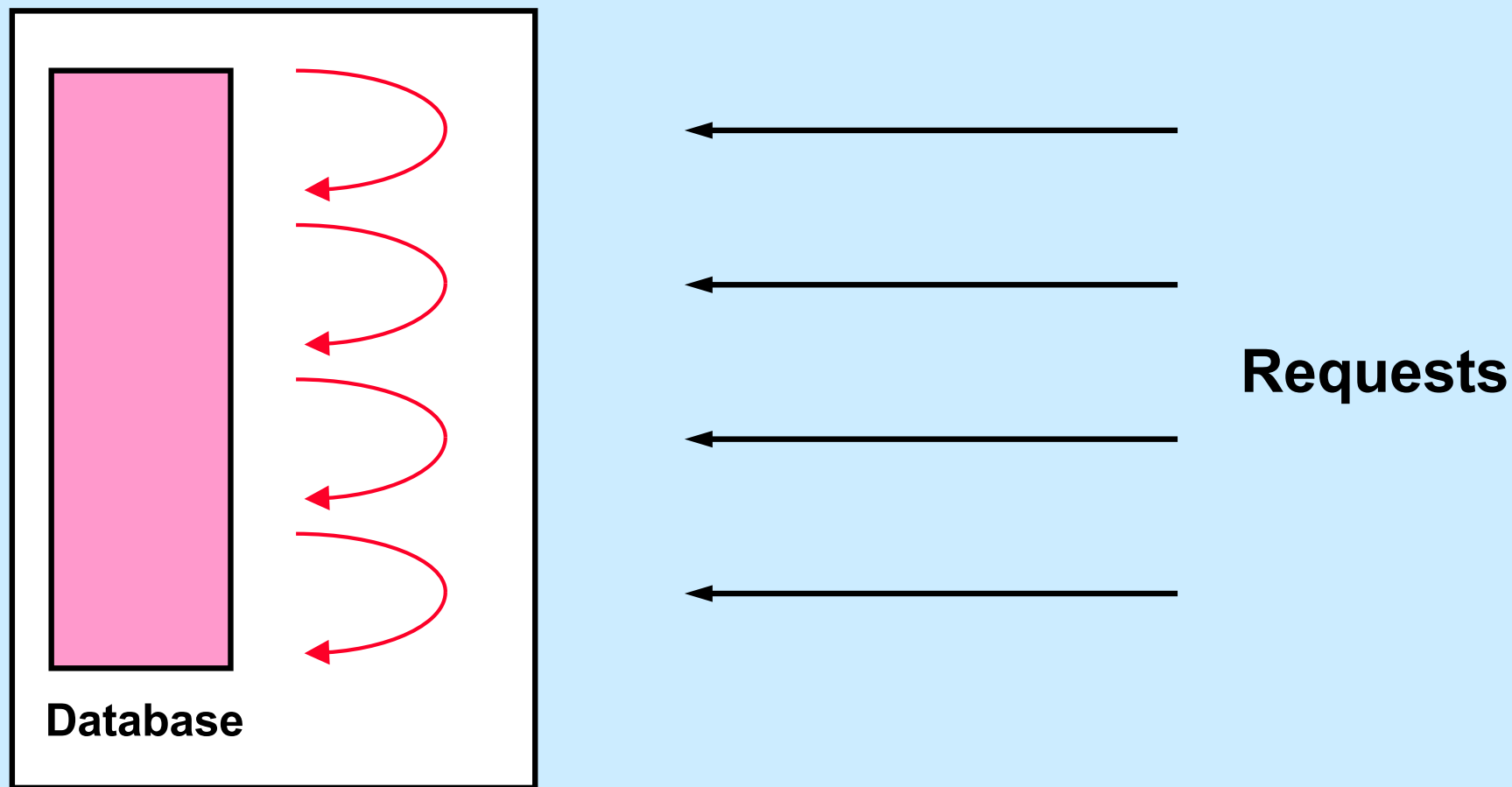
Process 3

# Single-Threaded Database Server



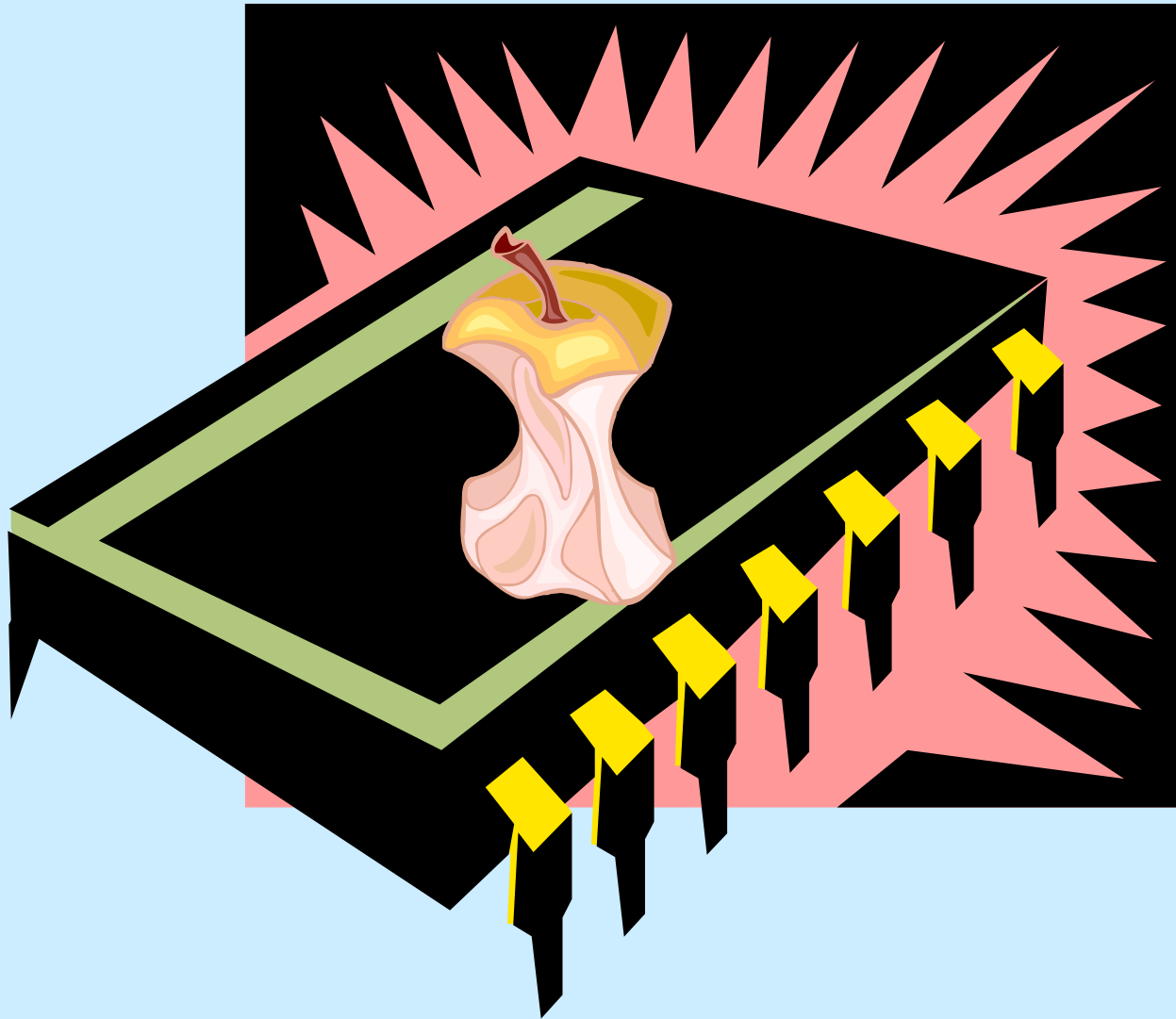
**Requests**

# Multithreaded Database Server

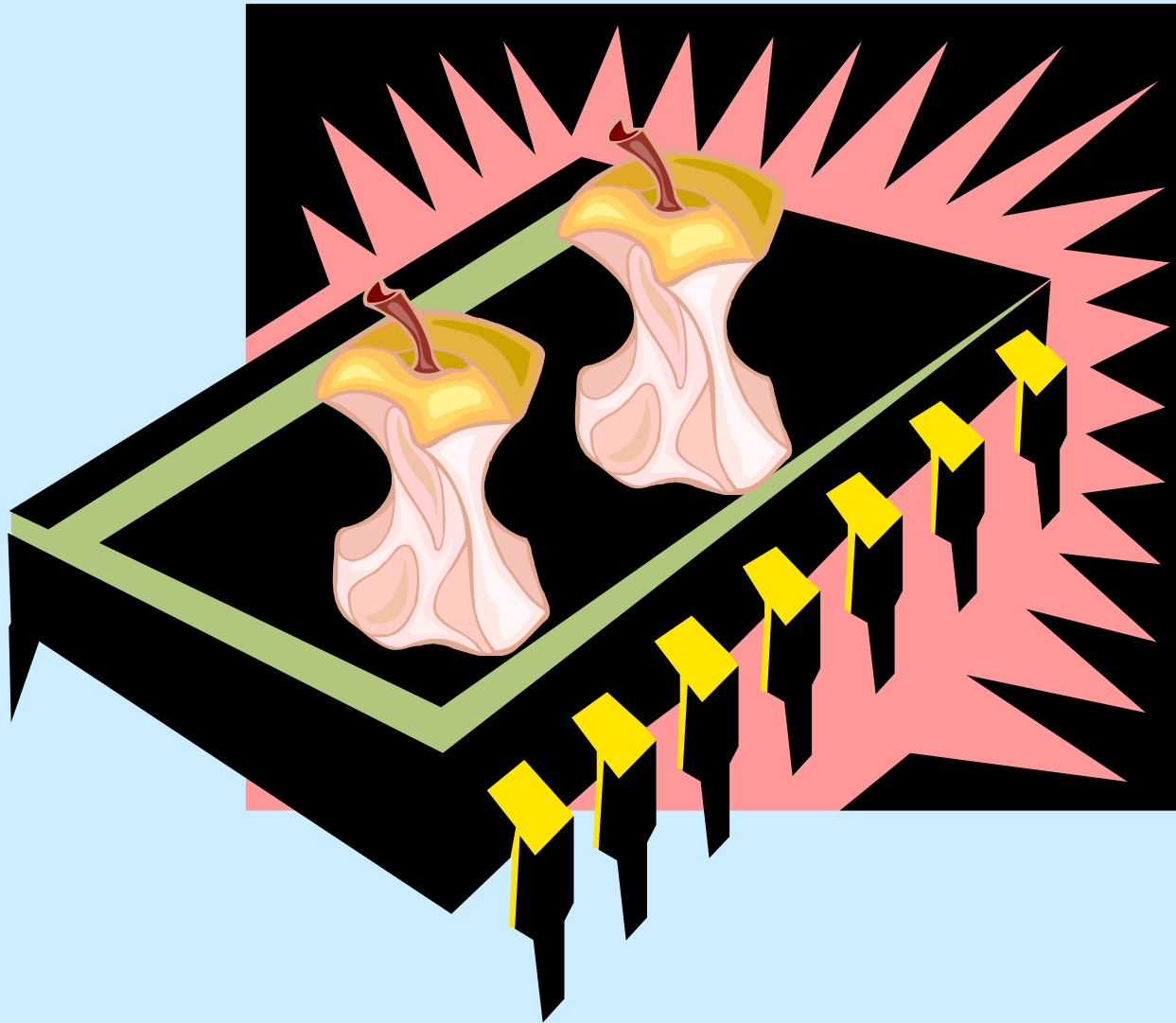




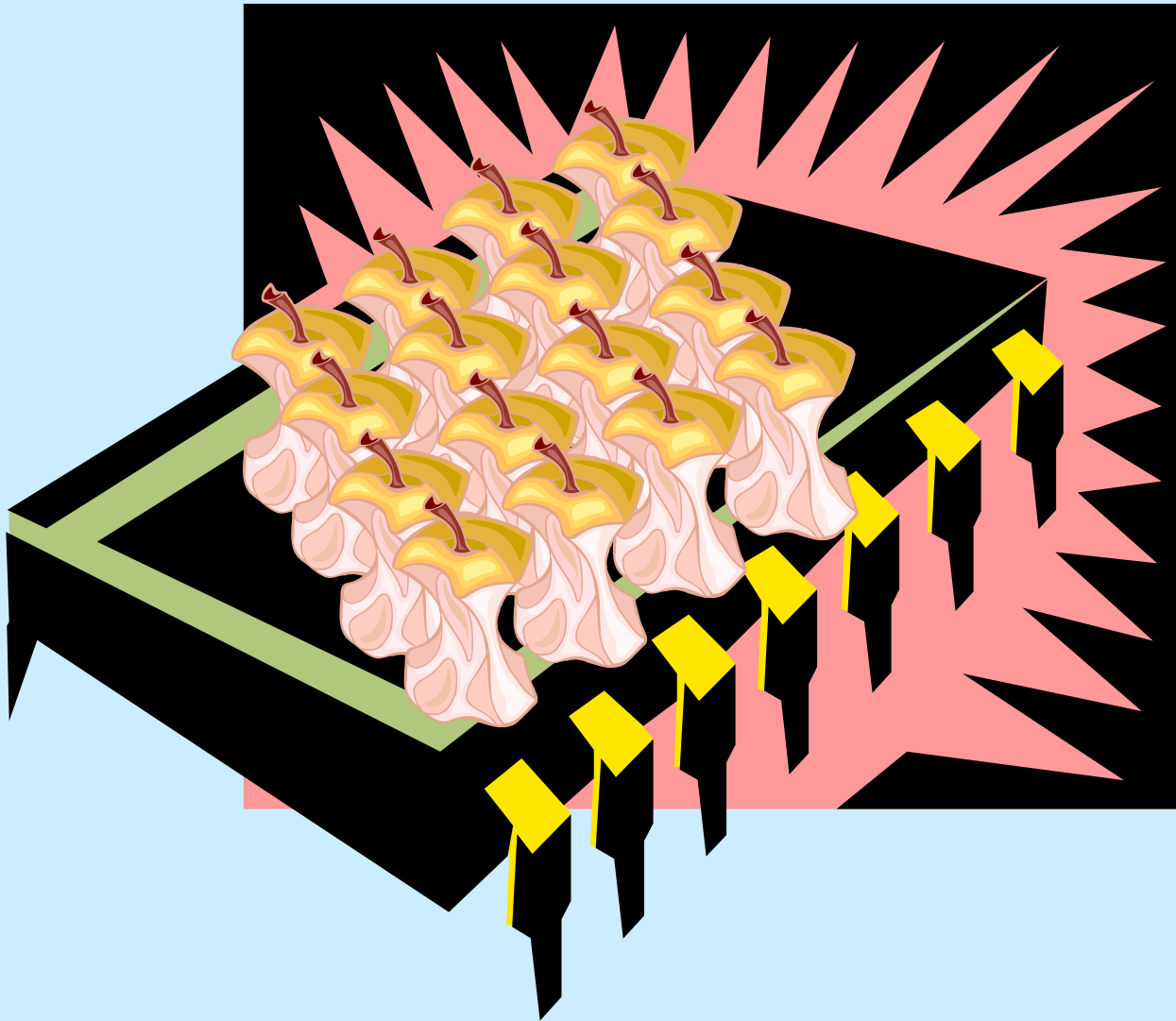
# Single-Core Chips



# Dual-Core Chips



# Multi-Core Chips



# Good News/Bad News

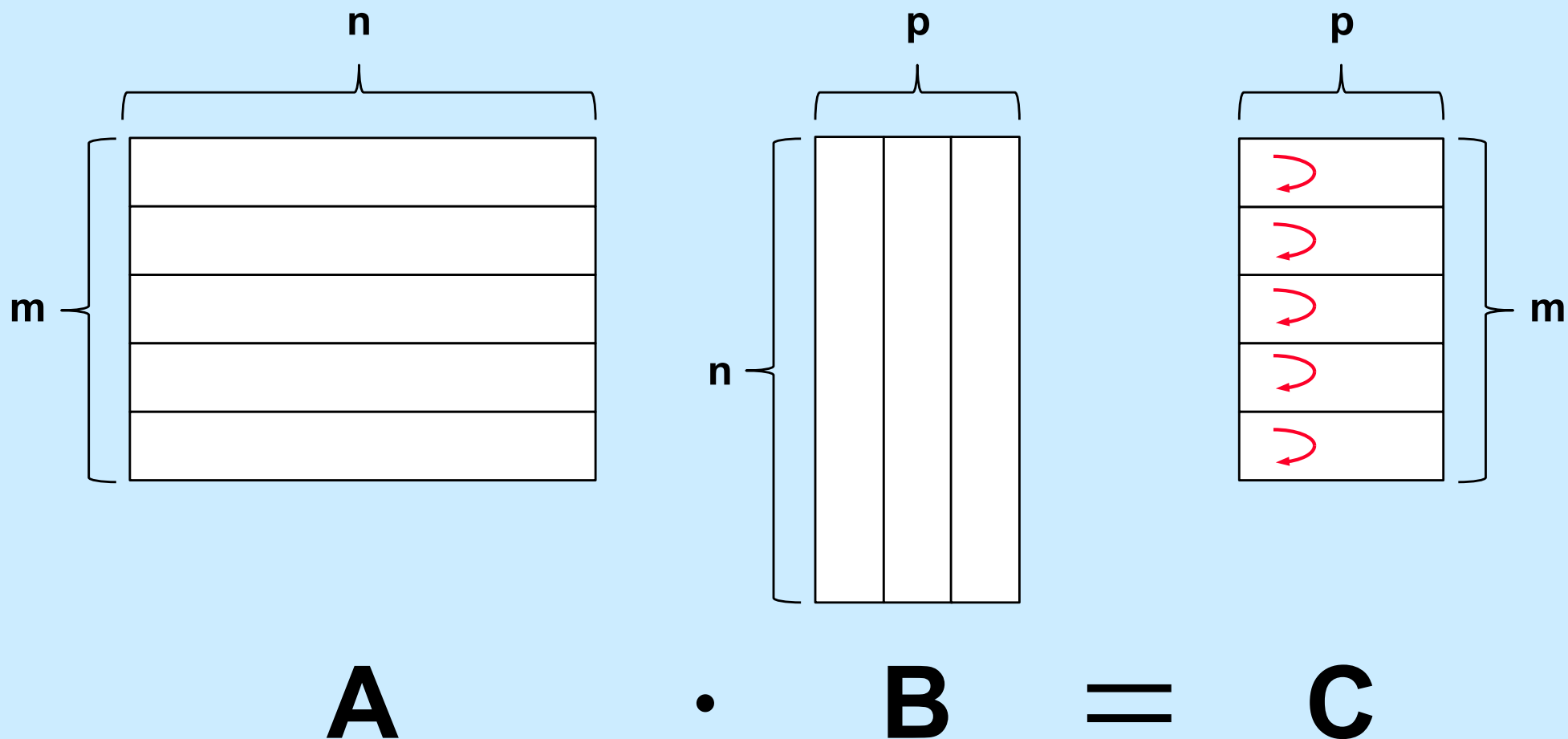
## Good news

- 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;
```

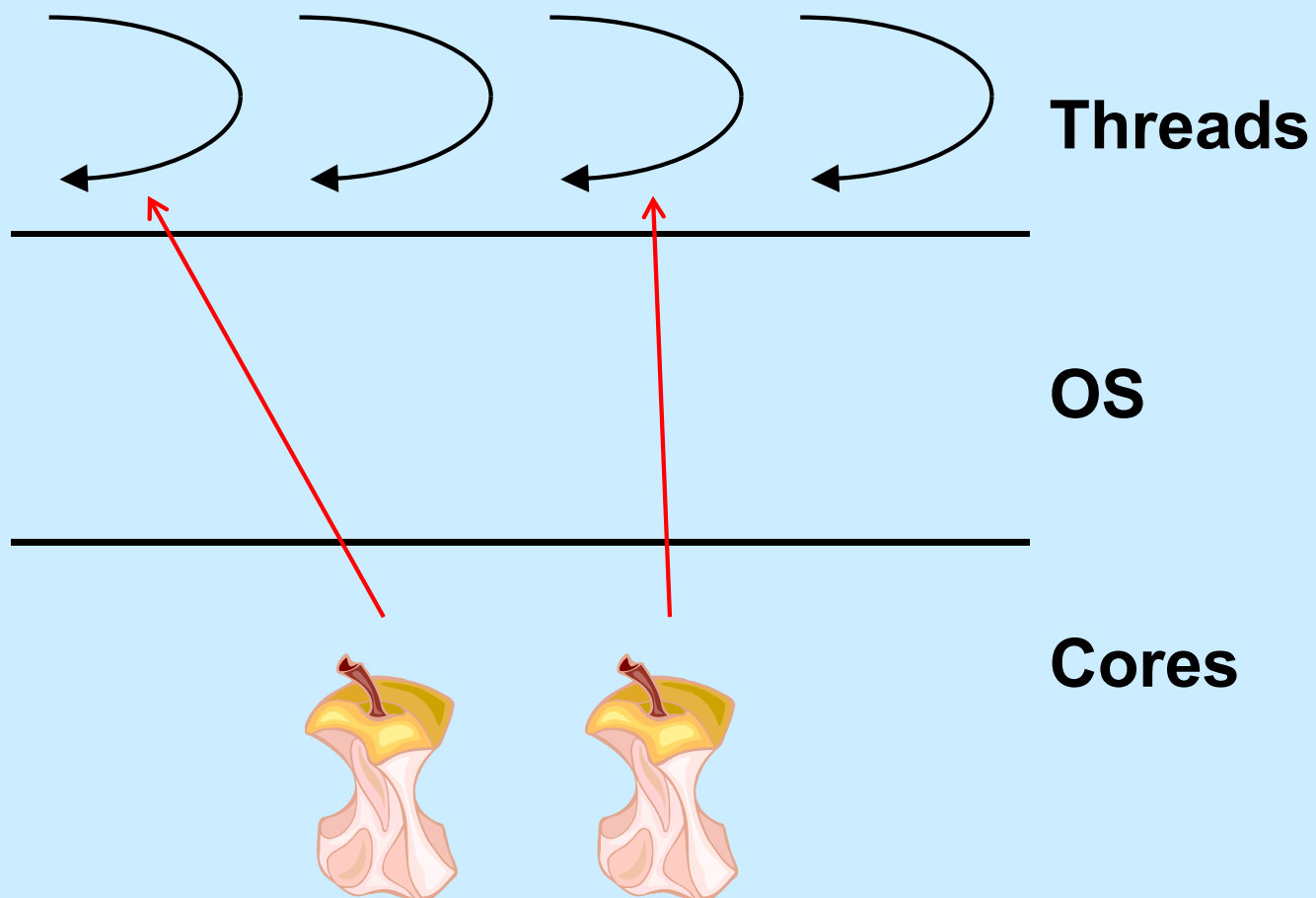
```
void relay(int left, int right) {  
    args_t LRargs, RLargs;  
    pthread_t LRthread, RLthread;  
    ...  
    pthread_create(&LRthread, 0, copy, &LRargs);  
    pthread_create(&RLthread, 0, copy, &RLargs);  
}
```

## Quiz 1

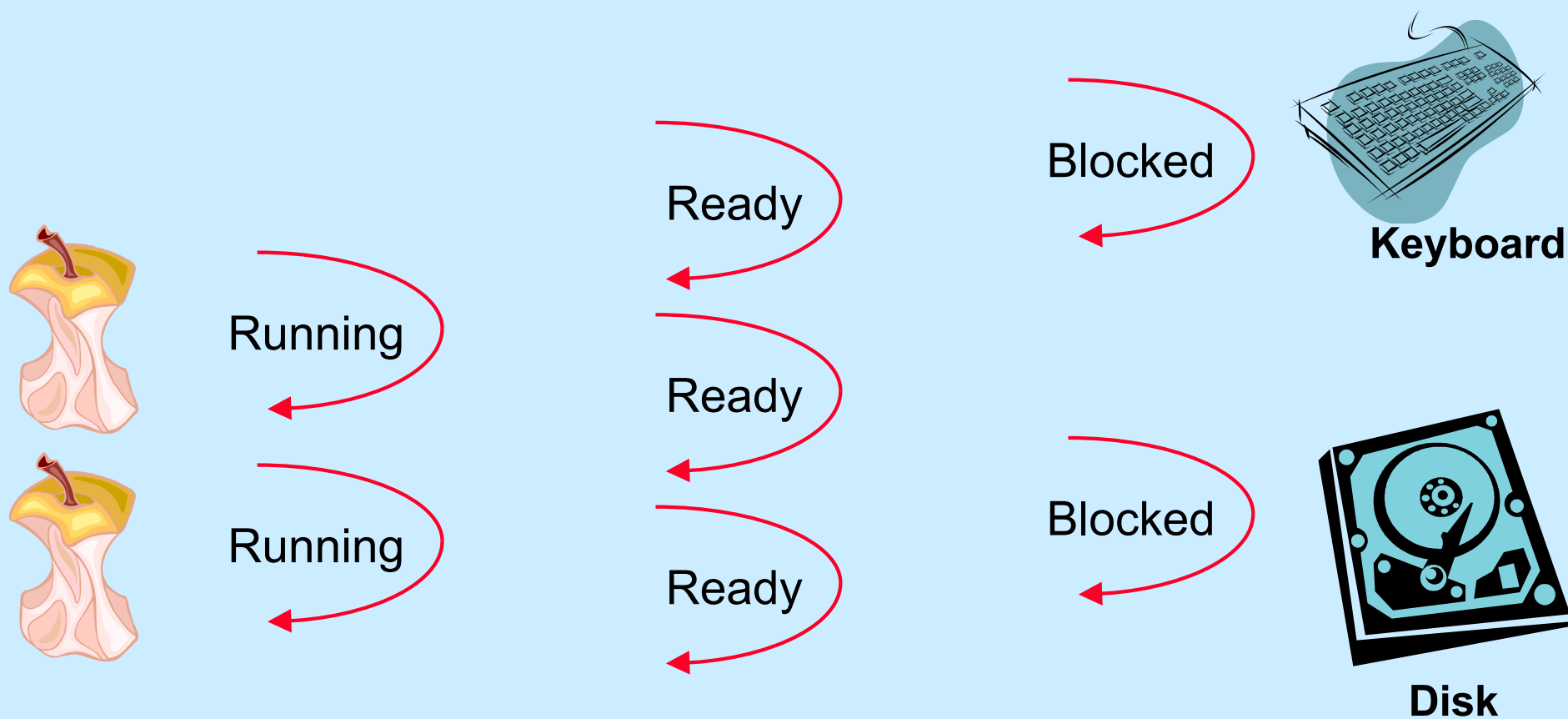
Does this work?

- 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%d\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 4. 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)

# 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, 20*1024*1024);  
  
...  
  
pthread_create(&thread, &thr_attr, startroutine, arg);  
  
...
```