

CS 33

Multithreaded Programming (2)

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)

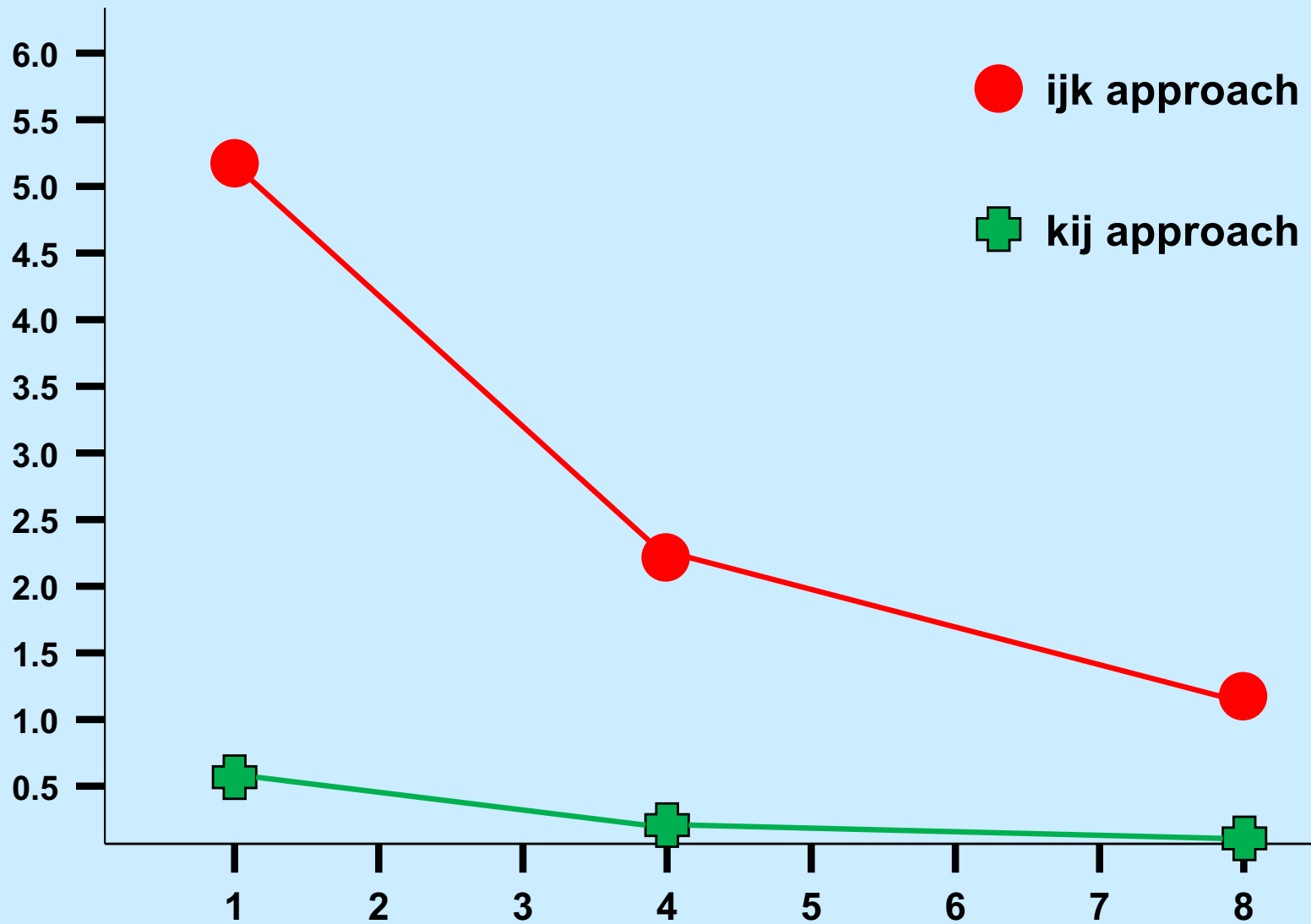
```
void *matmult(void *arg) {
    long row = (long) arg;
    long col;
    long i;
    long t;

    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);
}
```

Compiling It

```
% gcc -o mat mat.c -pthread
```

Performance



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;
```

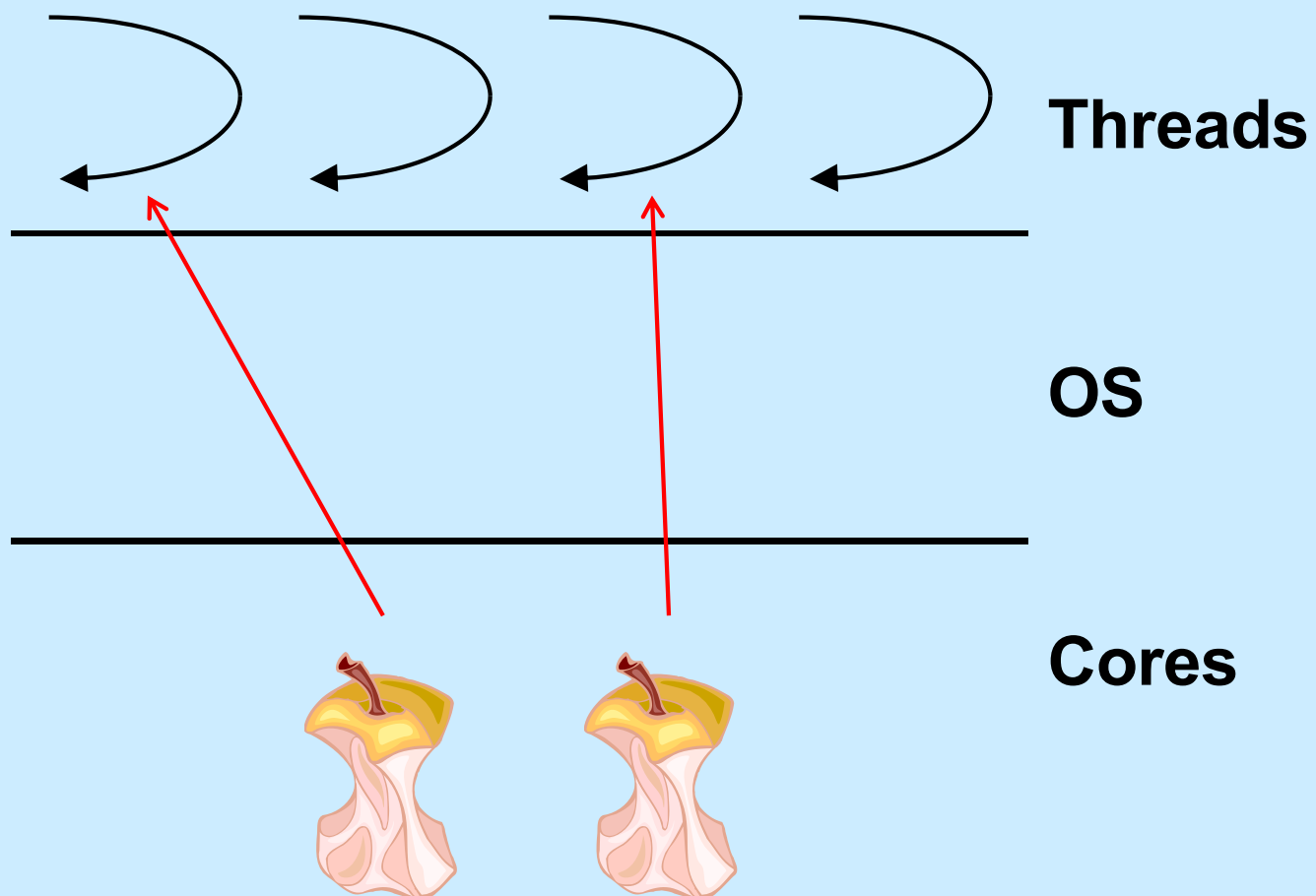
```
void relay(int left, int right) {  
    args_t LRargs, RLargs;  
    pthread_t LRthread, RLthread;  
    ...  
    pthread_create(&LRthread, 0, copy, &LRargs);  
    pthread_create(&RLthread, 0, copy, &RLargs);  
}
```

Quiz 1

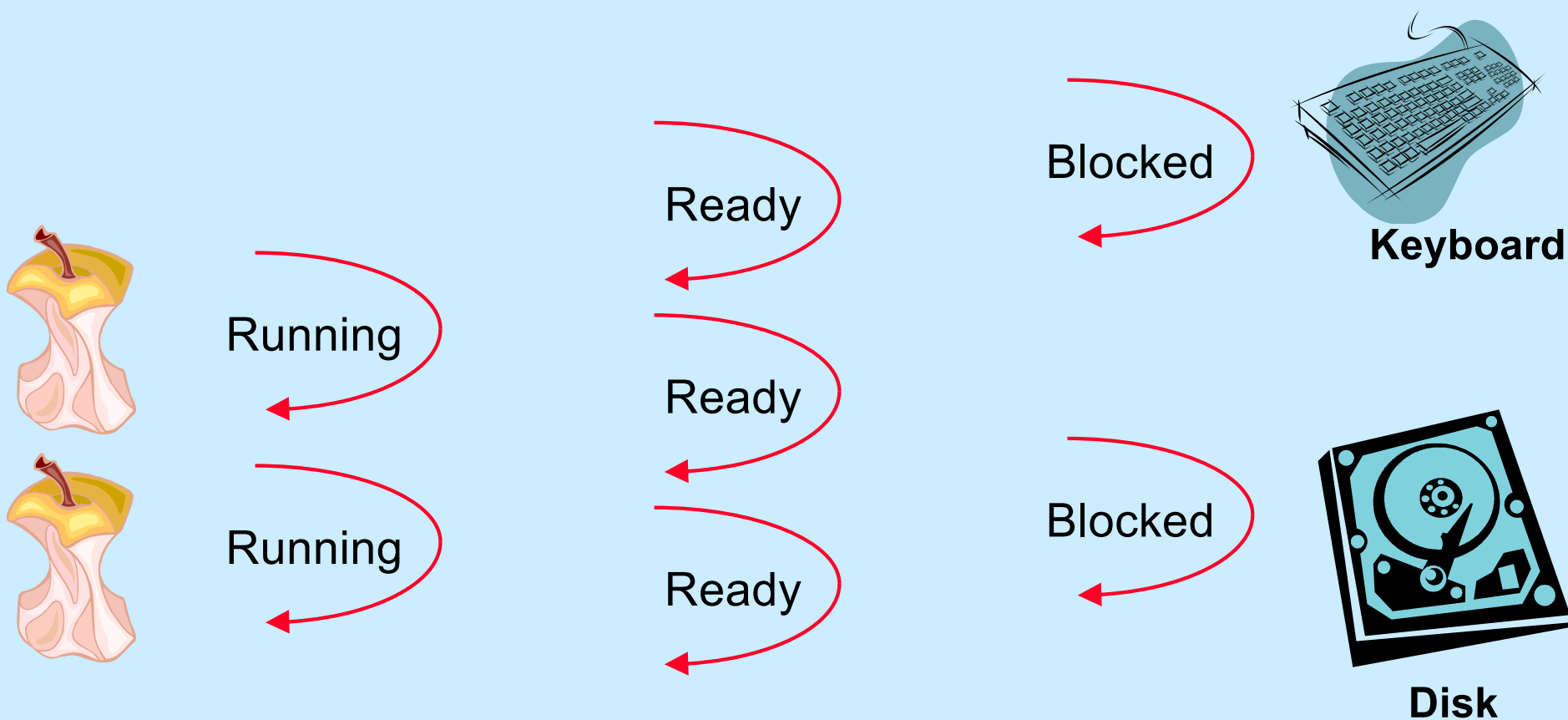
Does this work?

- a) no
- b) yes
- c) it depends upon the word size

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) indeterminate
- b) T2, T1, T0
- c) T0, T1, T2
- d) T1, T2, T0

Cost of Threads

```
void *work(long n) {  
    volatile long x=2;  
  
    for (long i=0; i<n; i++) {  
        long oldx = x;  
        x *= x;  
        x /= oldx;  
    }  
    return 0;  
}
```

Cost of Threads

```
int main(int argc, char *argv[]) {  
    long nthreads = atol(argv[1]);  
    long iterations = atol(argv[2]);  
    long val = iterations/nthreads;  
  
    for (long i=0; i<nthreads; i++)  
        pthread_create(&thread, 0, work,  
            (void *)val);  
    pthread_exit(0);  
    return 0;  
}
```

Cost of Threads

```
void *work(long n) {  
    volatile long x=2;  
  
    for (long i=0; i<n; i++) {  
        long oldx = x;  
        x *= x;  
        x /= oldx;  
    }  
    return 0;  
}
```

Not a Quiz

This code runs in time n on a 6-core processor when $nthreads$ is 6. It runs in time p on the same processor when $nthreads$ is 1000.

- 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);  
  
...
```

Mutual Exclusion



Threads and Mutual Exclusion

Thread 1:

```
x = x+1;  
/*  
    movl x,%eax  
    incr %eax  
    movl %eax,x  
*/
```

Thread 2:

```
x = x+1;  
/*  
    movl x,%eax  
    incr %eax  
    movl %eax,x  
*/
```


Quiz 3

Suppose the following code is compiled by gcc. Will it still be the case that x's value might not be incremented by 2?

- a) yes
- b) no

Thread 1:

```
x = x+1;  
/*  
  incr x  
*/
```

Thread 2:

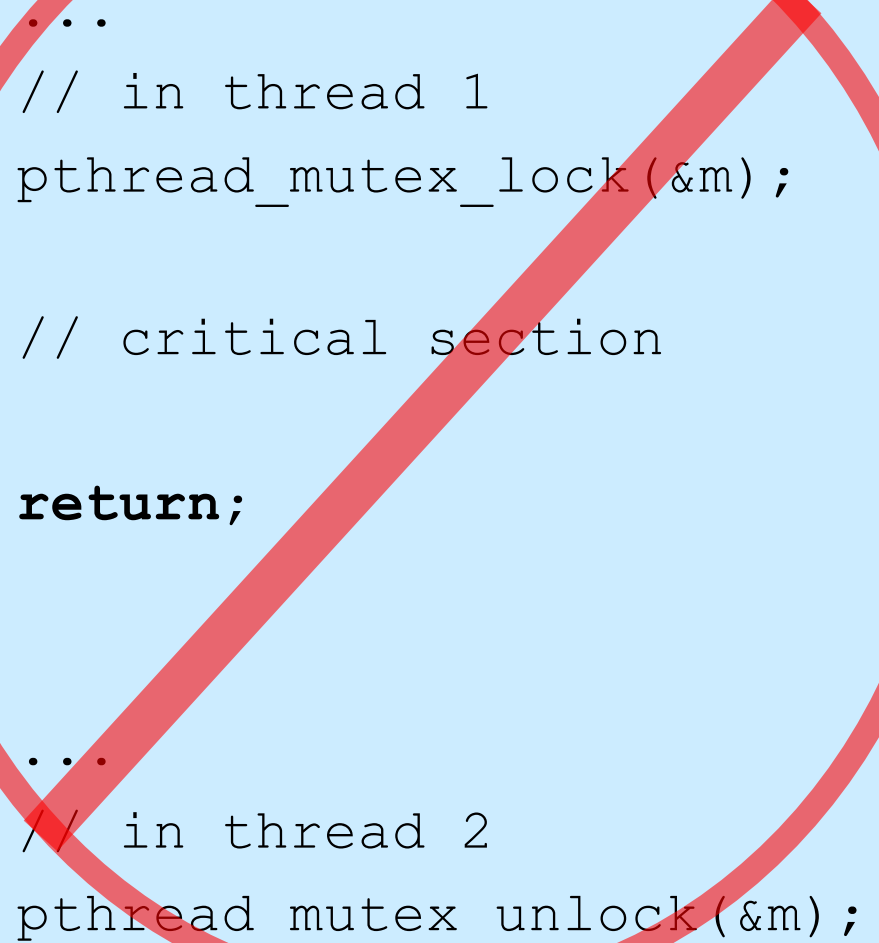
```
x = x+1;  
/*  
  incr x  
*/
```

POSIX Threads Mutual Exclusion

```
pthread_mutex_t m =  
    PTHREAD_MUTEX_INITIALIZER;  
    // shared by both threads  
int x; // ditto  
  
pthread_mutex_lock(&m);  
  
x = x+1;  
  
pthread_mutex_unlock(&m);
```

Correct Usage

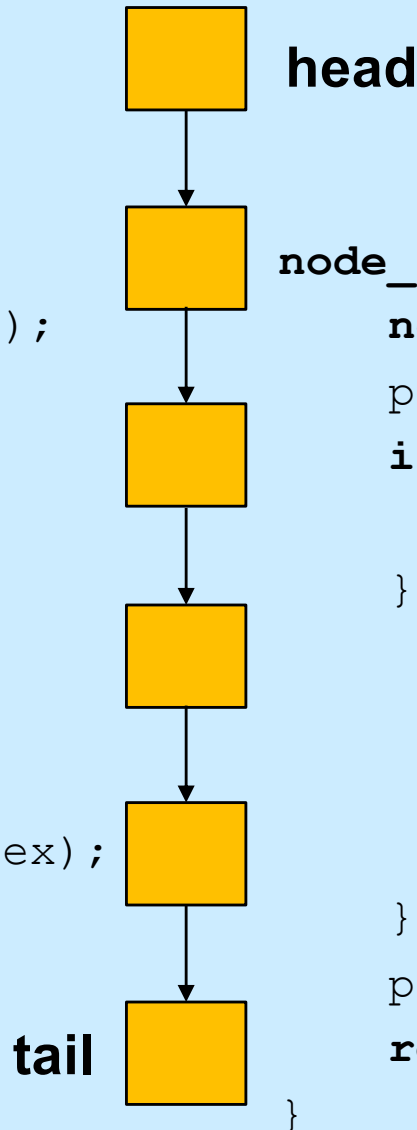
```
pthread_mutex_lock(&m);  
  
// critical section  
  
pthread_mutex_unlock(&m);
```



```
...  
// in thread 1  
pthread_mutex_lock(&m);  
  
// critical section  
  
return;  
  
...  
// in thread 2  
pthread_mutex_unlock(&m);
```

A Queue

```
void enqueue(node_t *item) {  
    pthread_mutex_lock(&mutex);  
    item->next = NULL;  
    if (tail == NULL) {  
        head = item;  
        tail = item;  
    } else {  
        tail->next = item;  
    }  
    pthread_mutex_unlock(&mutex);  
}
```



```
node_t *dequeue() {  
    node_t *ret;  
    pthread_mutex_lock(&mutex);  
    if (head == NULL) {  
        ret = NULL;  
    } else {  
        ret = head;  
        head = head->next;  
        if (head == NULL)  
            tail = NULL;  
    }  
    pthread_mutex_unlock(&mutex);  
    return ret;  
}
```

Removing a Freelist Block

```
void pull_from_freelist(fblock_t *fbp) {  
    ...  
    fbp->blink->flink = fbp->flink;  
    fbp->flink->blink = fbp->blink;  
    ...  
}
```

Parallelizing It

- **Coarse grained**
 - one mutex for the heap
 - threads lock the mutex before doing any operation
 - unlock it afterwards
 - only one thread at a time
- **Fine grained**
 - one mutex for each block
 - threads lock mutexes of only the blocks they are using
 - multiple threads at a time

Removing a Freelist Block: Coarse Grained

```
void pull_from_freelist(fblock_t *fbp) {  
    pthread_mutex_lock(&flist_mutex);  
  
    ...  
    fbp->blink->flink = fbp->flink;  
    fbp->flink->blink = fbp->blink;  
  
    ...  
    pthread_mutex_unlock(&flist_mutex);  
}
```

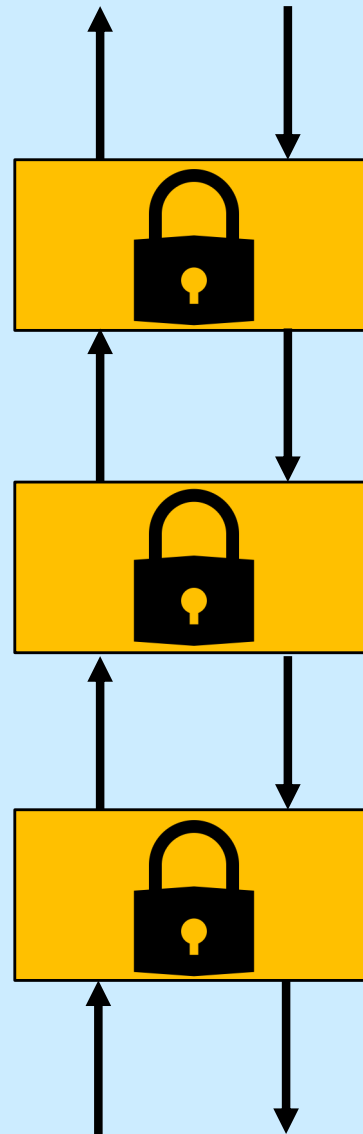
Removing a Freelist Block: Fine Grained (1)

```
void pull_from_freelist(fblock_t *fbp) {  
    pthread_mutex_lock(&fpp->mutex);  
  
    ...  
    fbp->blink->flink = fbp->flink;  
    fbp->flink->blink = fbp->blink;  
  
    ...  
    pthread_mutex_unlock(&fpp->mutex);  
}
```

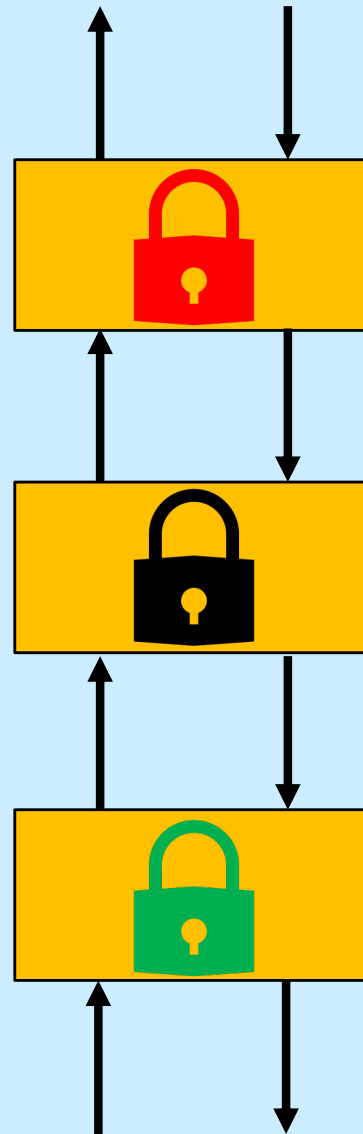

Removing a Freelist Block: Fine Grained (2)

```
void pull_from_freelist(fblock_t *fbp) {
    pthread_mutex_lock(&fpp->mutex);
    ...
    pthread_mutex_lock(&fpp->blink->mutex);
    fbp->blink->flink = fbp->flink;
    pthread_mutex_lock(&fpp->flink->mutex);
    fbp->flink->blink = fbp->blink;
    ...
    pthread_mutex_unlock(&fpp->blink->mutex);
    pthread_mutex_unlock(&fpp->flink->mutex);
    pthread_mutex_unlock(&fpp->mutex);
}
```

Multiple Pulls



Multiple Pulls



Taking Multiple Locks

```
func1( ) {  
    pthread_mutex_lock(&m1);  
    /* use object 1 */  
    pthread_mutex_lock(&m2);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m2);  
    pthread_mutex_unlock(&m1);  
}
```

```
func2( ) {  
    pthread_mutex_lock(&m2);  
    /* use object 2 */  
    pthread_mutex_lock(&m1);  
    /* use objects 1 and 2 */  
    pthread_mutex_unlock(&m1);  
    pthread_mutex_unlock(&m2);  
}
```