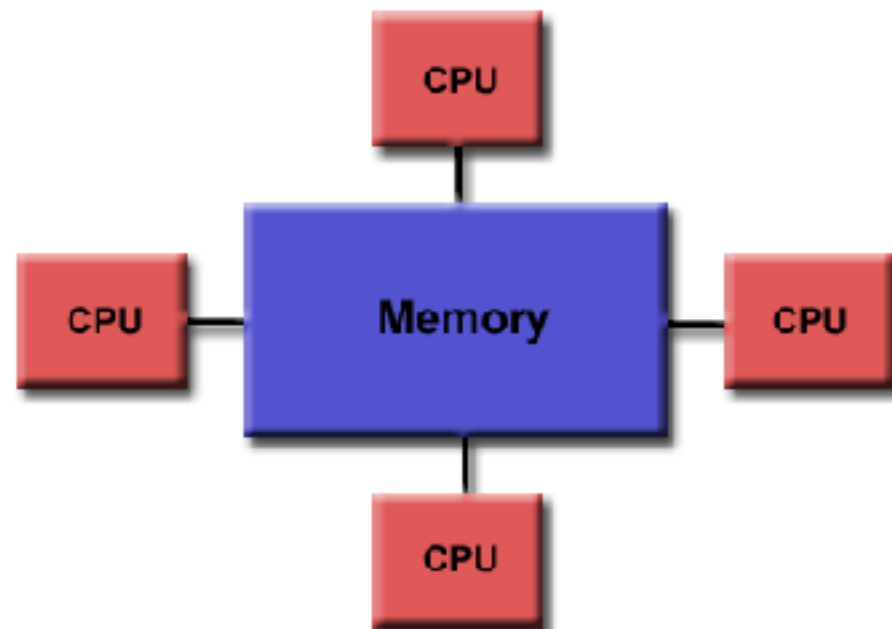


Shared memory parallelization

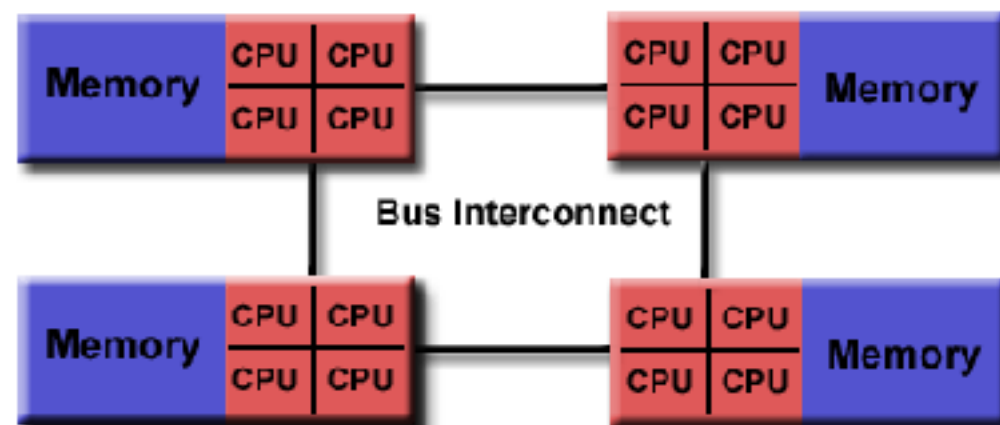


shared memory vs. distributed memory

shared memory

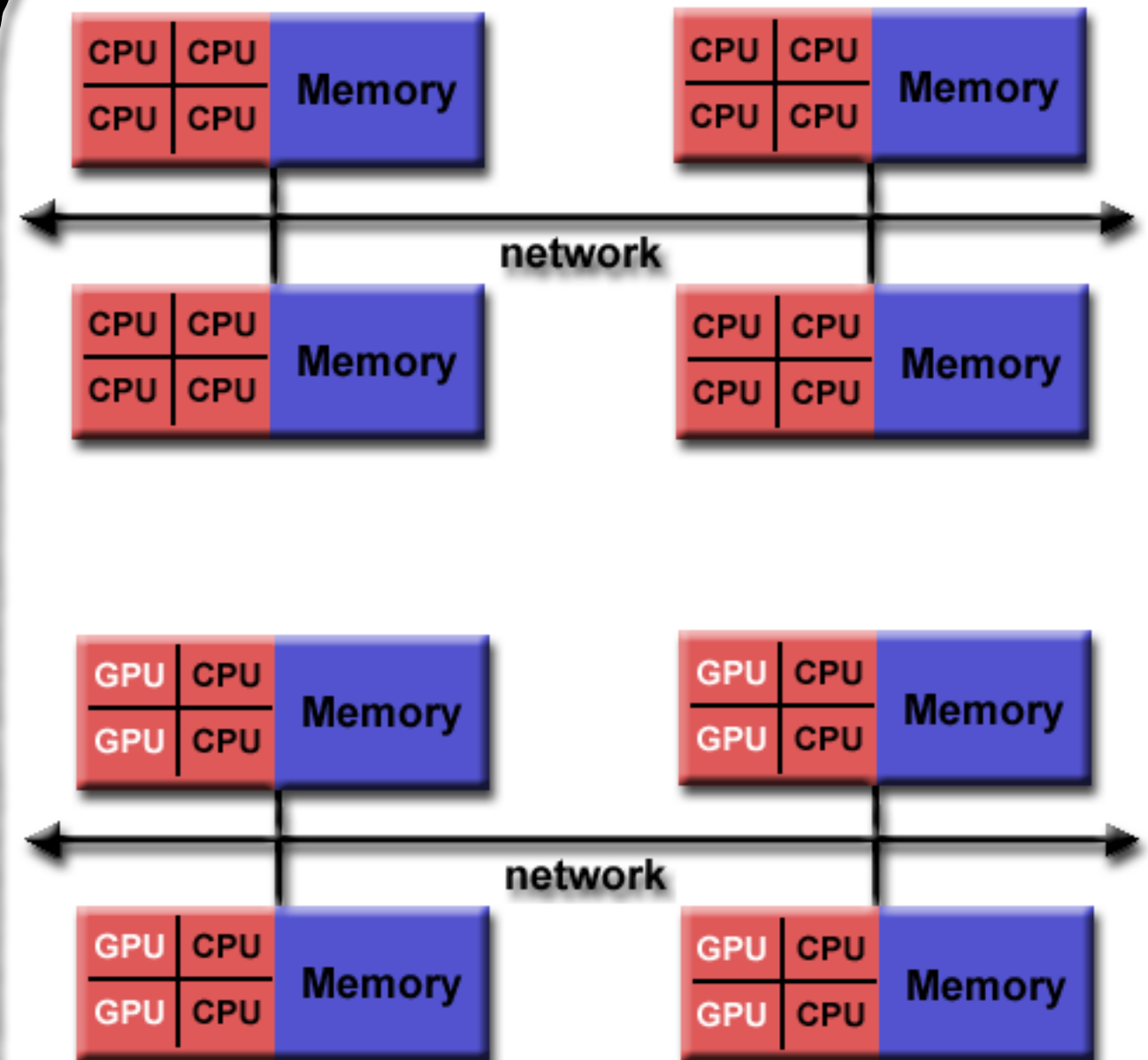


Shared Memory (UMA)



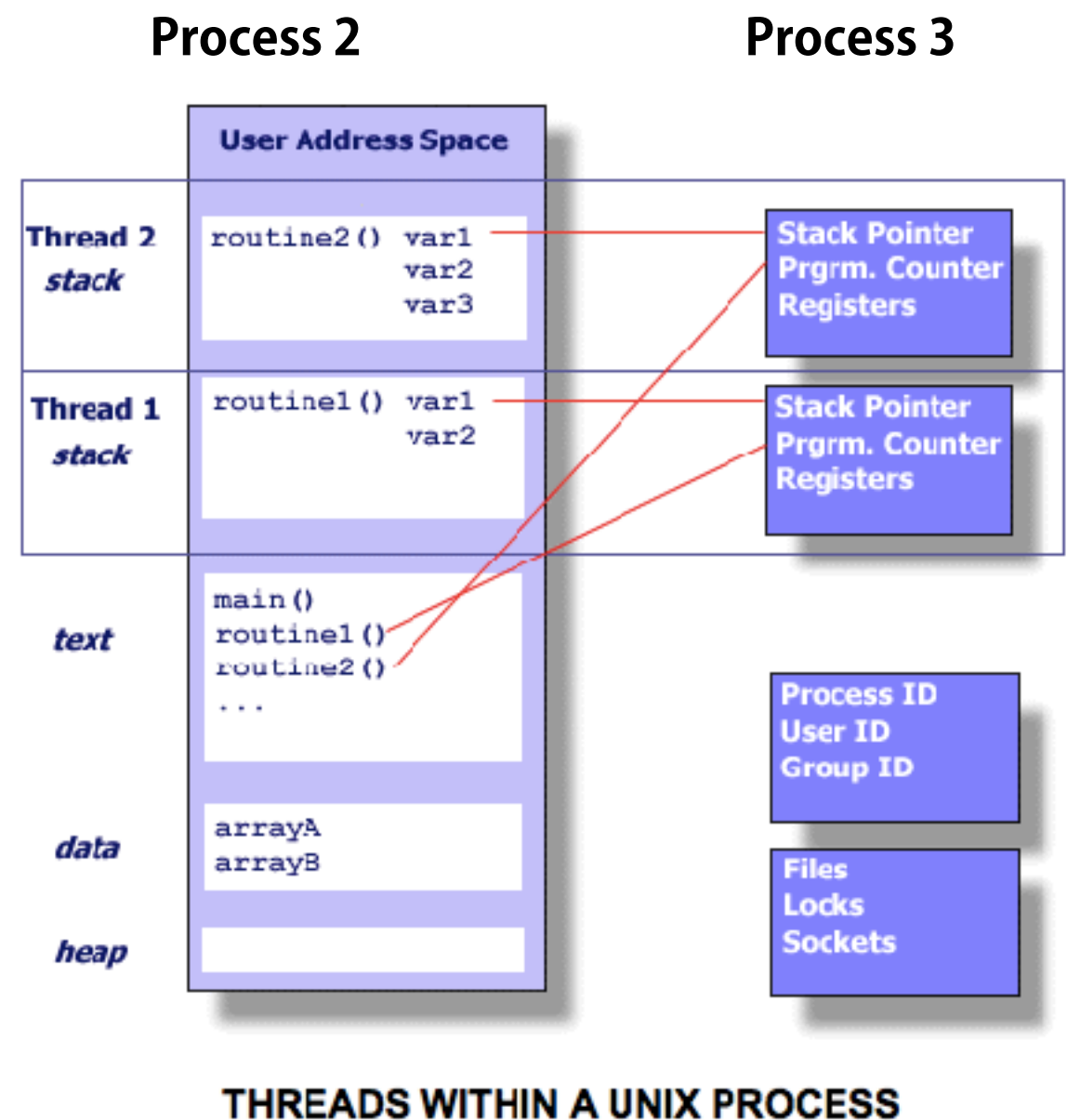
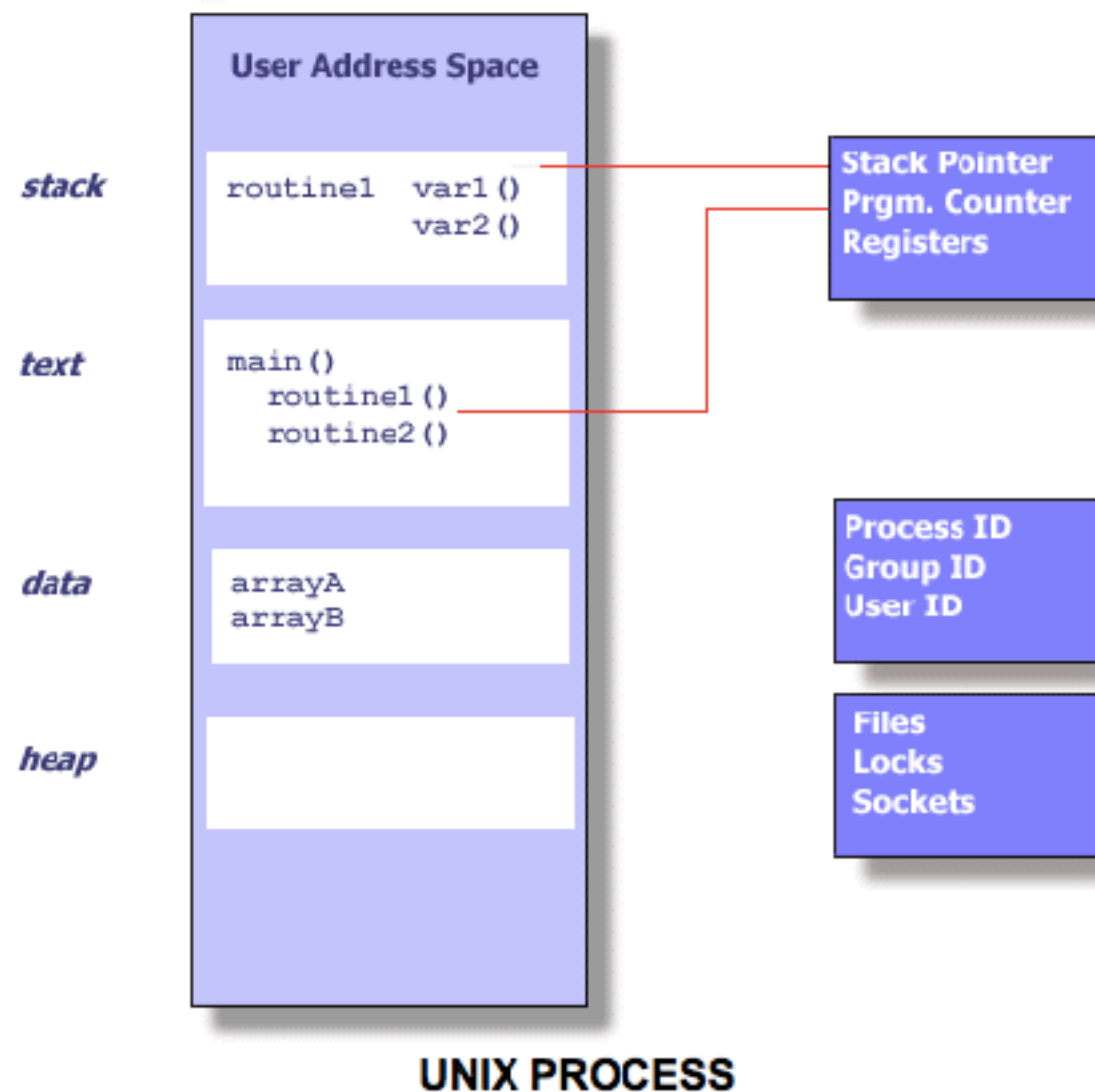
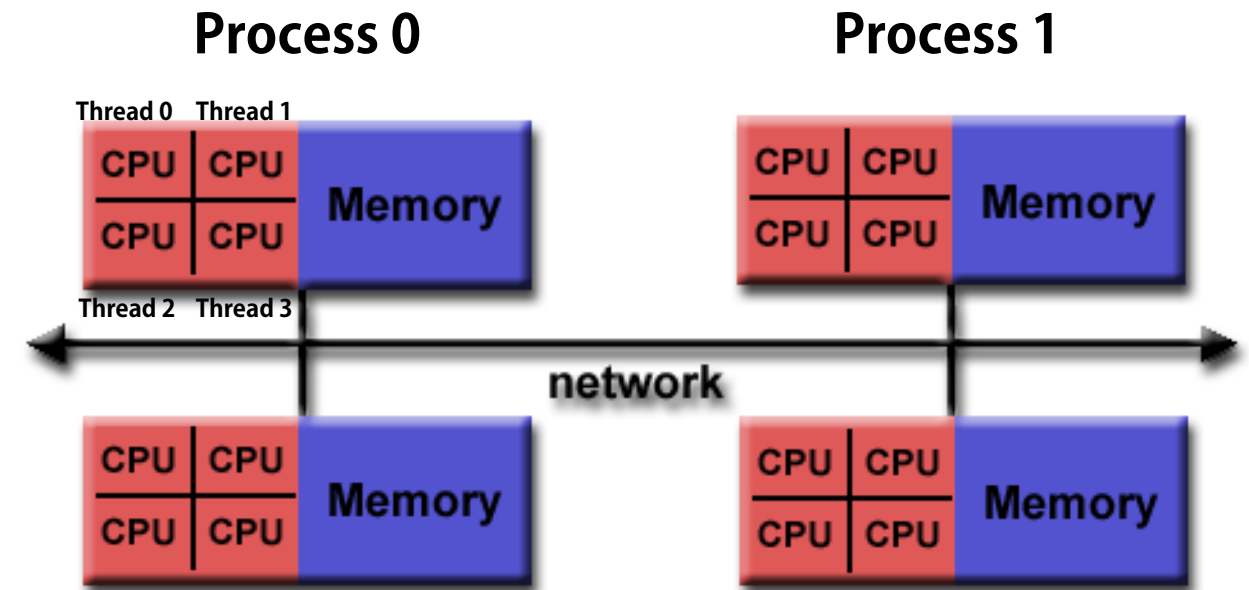
Shared Memory (NUMA)

distributed memory



process vs. thread

Per process items	Per thread items
-----	-----
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



pthread vs. OpenMP

pthread

- POSIX standard
- Library functions
- Explicit fork and join
- Explicit synchronization
- Explicit locks

```
#include <pthread.h>
#include <stdio.h>

void* print(void*) {
    static int t=0;
    printf("%d\n", t++);
}

int main() {
    for(int i=0; i<10; i++) {
        pthread_t thread;
        pthread_create(&thread, NULL, print, NULL);
    }
    pthread_exit(NULL);
}
```

g++ -pthread example.cpp

OpenMP

- Industry standard
- Compiler directives
- Explicit fork and join
- Implicit synchronization
- Implicit locks

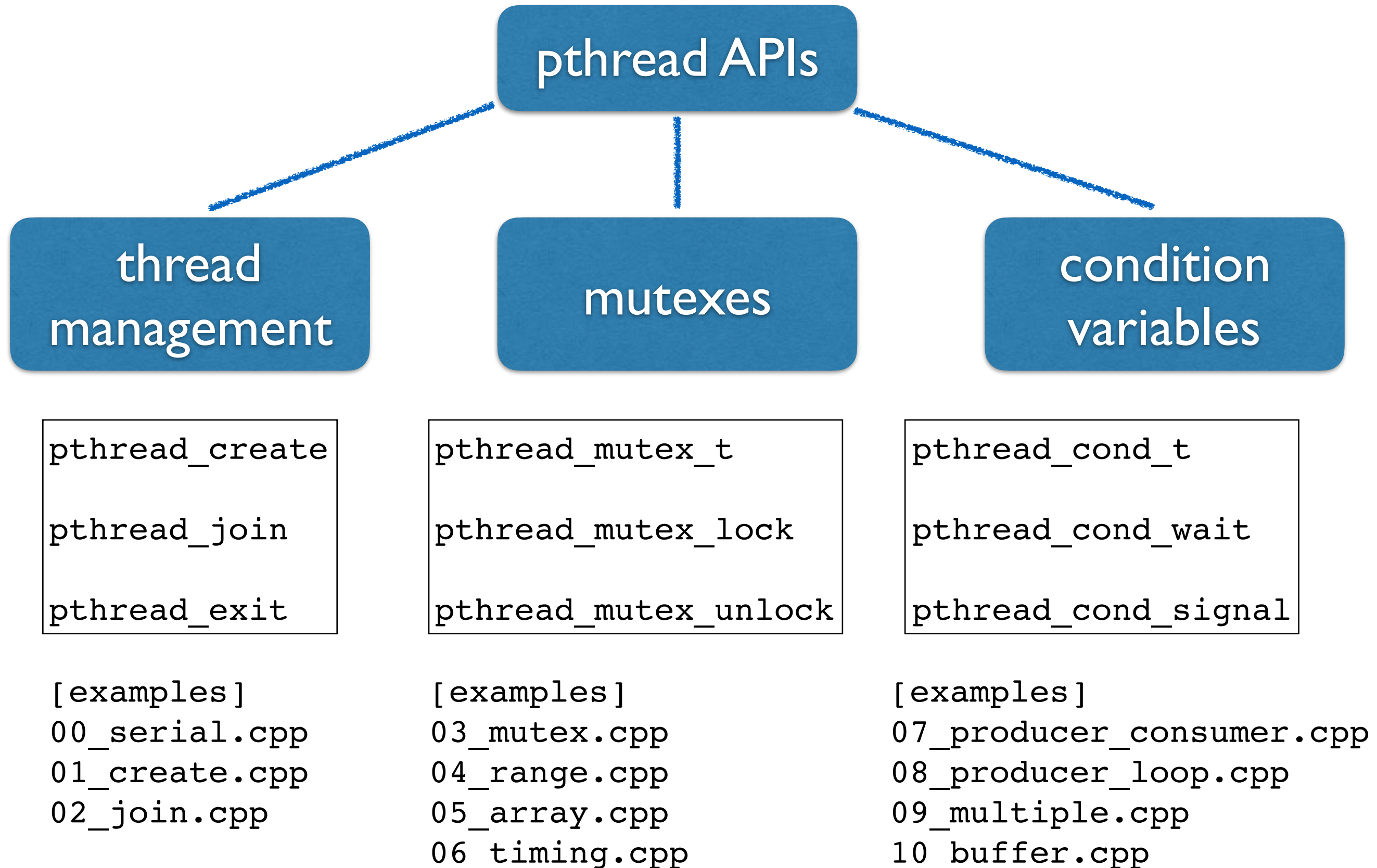
```
#include <stdio.h>

void print() {
    static int t=0;
    printf("%d\n", t++);
}

int main() {
    #pragma omp parallel for
    for(int i=0; i<10; i++) {
        print();
    }
}
```

g++ -fopenmp example.cpp

pthread (POSIX threads)



Thread management

00_serial.cpp

```
#include <stdio.h>

void print() {
    static int t=0;
    printf("%d\n", t++);
}

int main() {
    for(int i=0; i<10; i++) {
        print();
    }
}
```

First let's review what a static variable is.
What will this code print?

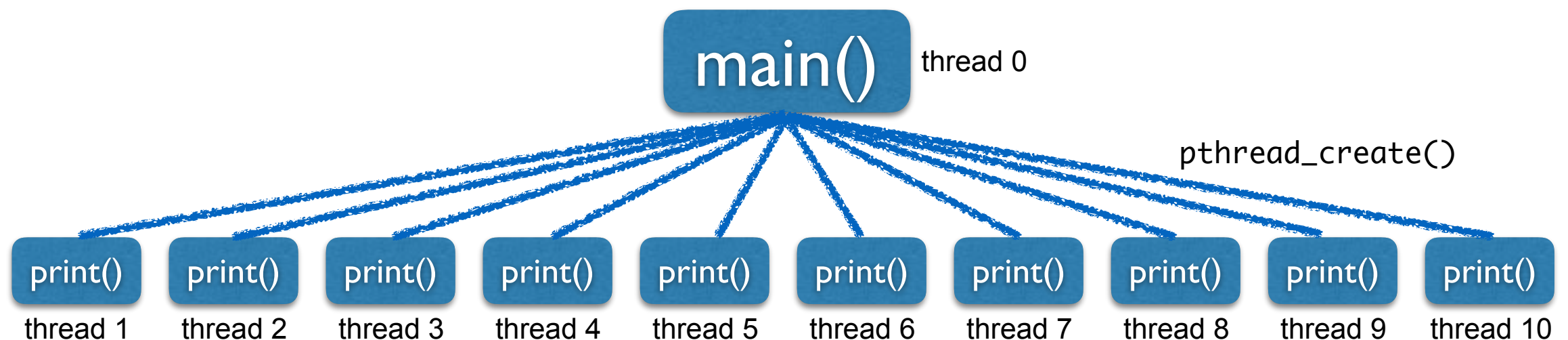
01_create.cpp

```
#include <pthread.h>
#include <stdio.h>

void* print(void*) {
    static int t=0;
    printf("%d\n", t++);
}

int main() {
    for(int i=0; i<10; i++) {
        pthread_t thread;
        pthread_create(&thread, NULL, print, NULL);
    }
    pthread_exit(NULL);
}
```

This code will create 10 threads and print from each of them.
Compile with "g++ -pthread 01_create.cpp"



Thread management

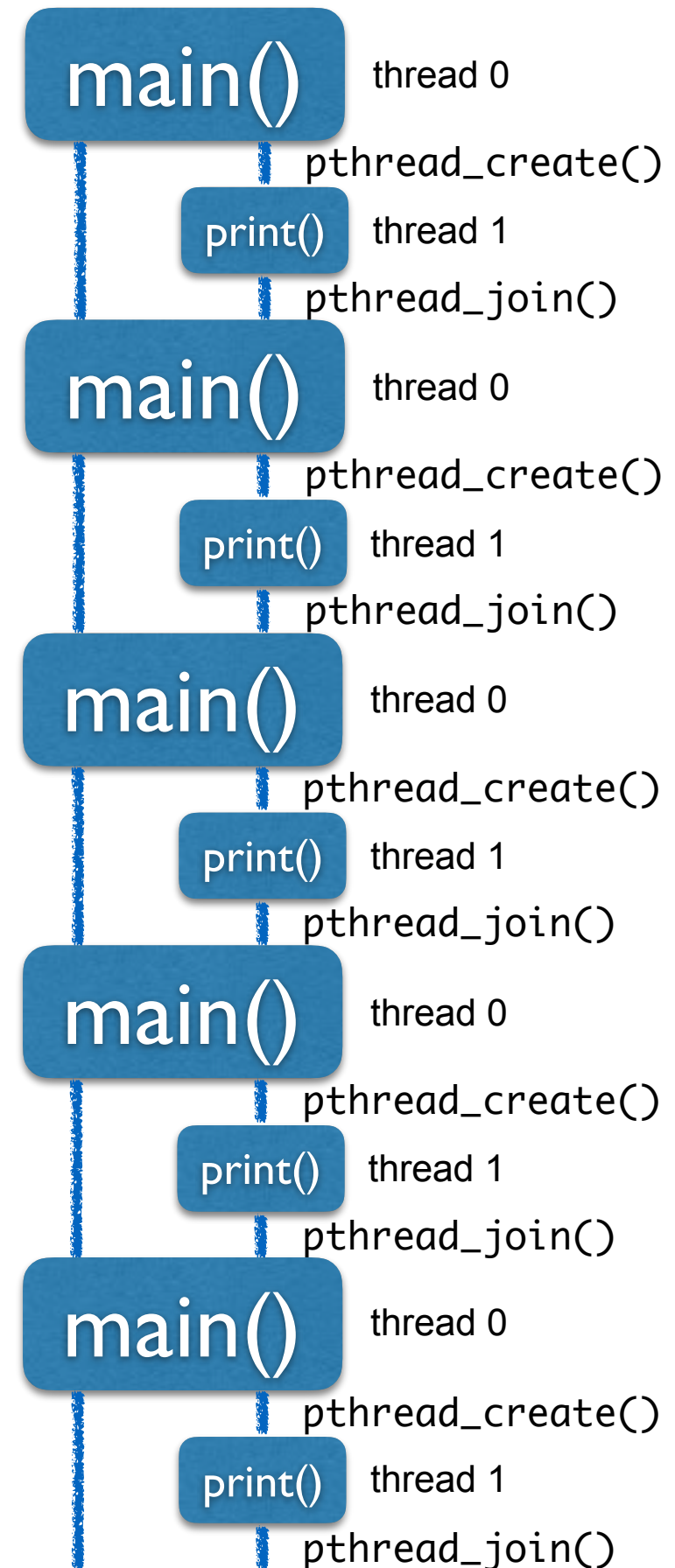
02_join.cpp

```
#include <pthread.h>
#include <stdio.h>

void* print(void*) {
    static int t=0;
    printf("%d\n", t++);
}

int main() {
    for(int i=0; i<10; i++) {
        pthread_t thread;
        pthread_create(&thread, NULL, print, NULL);
        pthread_join(thread, NULL);
    }
    pthread_exit(NULL);
}
```

What happens when we do a `pthread_join` in the loop?
Why is the output different from the previous code?



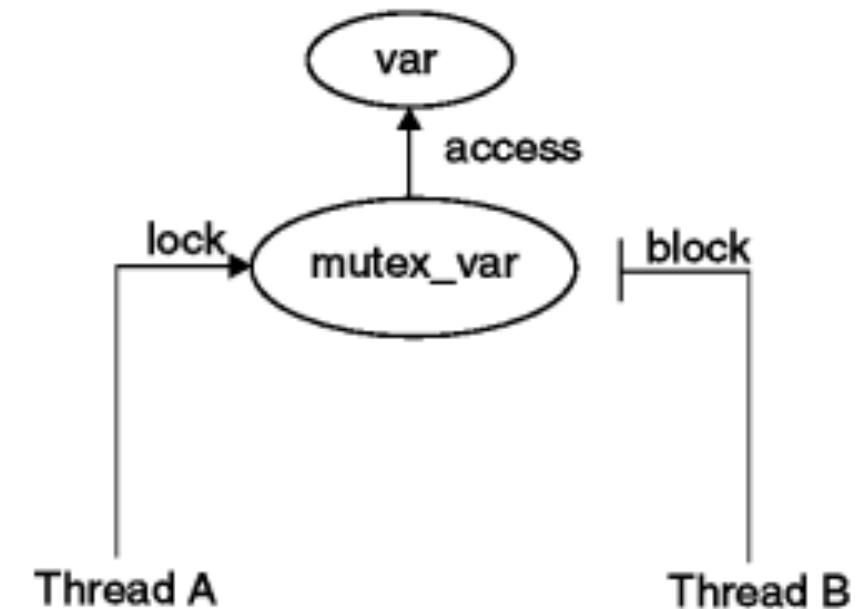
Mutexes

03_mutex.cpp

```
#include <pthread.h>
#include <stdio.h>

void* print(void*) {
    static int t=0;
    static pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&mutex);
    t++;
    pthread_mutex_unlock(&mutex);
    printf("%d\n", t-1);
}

int main() {
    for(int i=0; i<10; i++) {
        pthread_t thread;
        pthread_create(&thread, NULL, print, NULL);
    }
    pthread_exit(NULL);
}
```



A mutex can be used for mutually exclusive updates of `t++`.
This means only one thread can read/write at any given time.
Why is the output always in order?

Mutexes

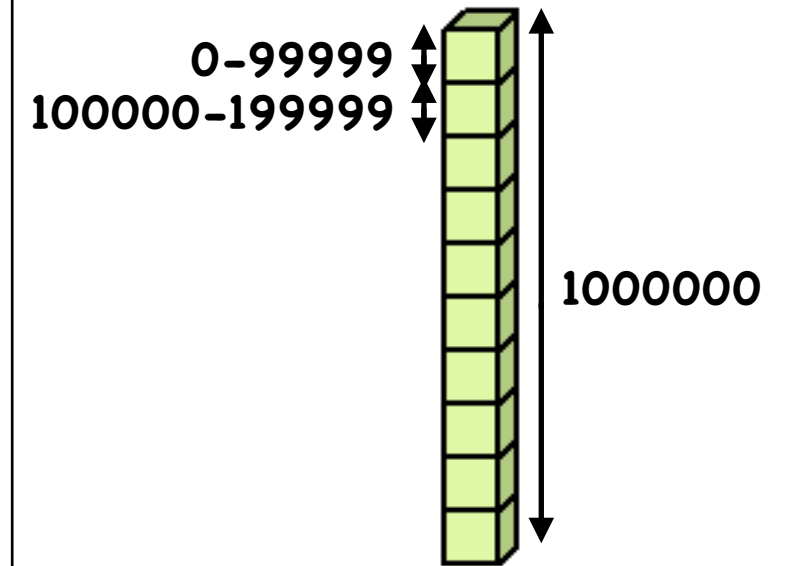
04_range.cpp

```
include <pthread.h>
#include <stdio.h>

const int size=1000000;

void* print(void*) {
    static int t=0;
    static pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&mutex);
    t++;
    pthread_mutex_unlock(&mutex);
    int ibegin = (t-1)*size/10;
    int iend = t*size/10;
    printf("thread %d, range %d - %d\n", t-1, ibegin, iend-1);
}

int main() {
    for(int i=0; i<10; i++) {
        pthread_t thread;
        pthread_create(&thread, NULL, print, NULL);
    }
    pthread_exit(NULL);
}
```



Now let us calculate a range of numbers for each thread.

This will be used in the next example to loop over subsets of an array.

Mutexes

05_array.cpp

```
#include <pthread.h>
#include <stdio.h>

const size_t size=1000000;
static size_t sum=0;

void* print(void* arg) {
    static int t=0;
    static pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&mutex);
    t++;
    pthread_mutex_unlock(&mutex);
    size_t ibegin = (t-1)*size/10;
    size_t iend = t*size/10;
    printf("thread %d, range %ld - %ld\n", t-1, ibegin, iend-1);
    size_t *a = (size_t*)arg;
    pthread_mutex_lock(&mutex);
    for (int i=ibegin; i<iend; i++) sum+=a[i];
    pthread_mutex_unlock(&mutex);
}

int main() {
    size_t *a = new size_t [size];
    for (size_t i=0; i<size; i++) a[i] = 1;
    pthread_t thread[10];
    for(int i=0; i<10; i++) {
        pthread_create(&thread[i], NULL, print, (void*)a);
    }
    printf("sum = %ld\n", sum);
    for(int i=0; i<10; i++) {
        pthread_join(thread[i], NULL);
    }
    printf("sum = %ld\n", sum);
    delete[] a;
    pthread_exit(NULL);
}
```

**This code calculates the sum of all elements of an array.
See how the sum is only correct after all the threads are joined.**

06_timing.cpp

Not showing the code here because it's too large.

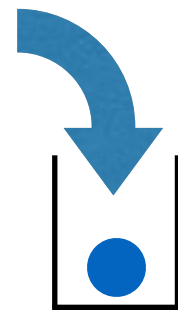
**This code measures the execution time of
the multi-threaded array summation.**

**Change the parameter "nthreads" and see
how the execution time changes.**

Producer consumer model

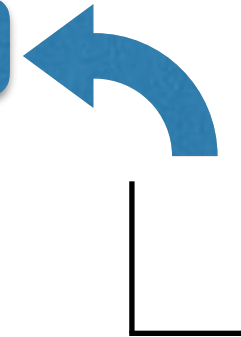
07_producer_consumer.cpp

producer thread

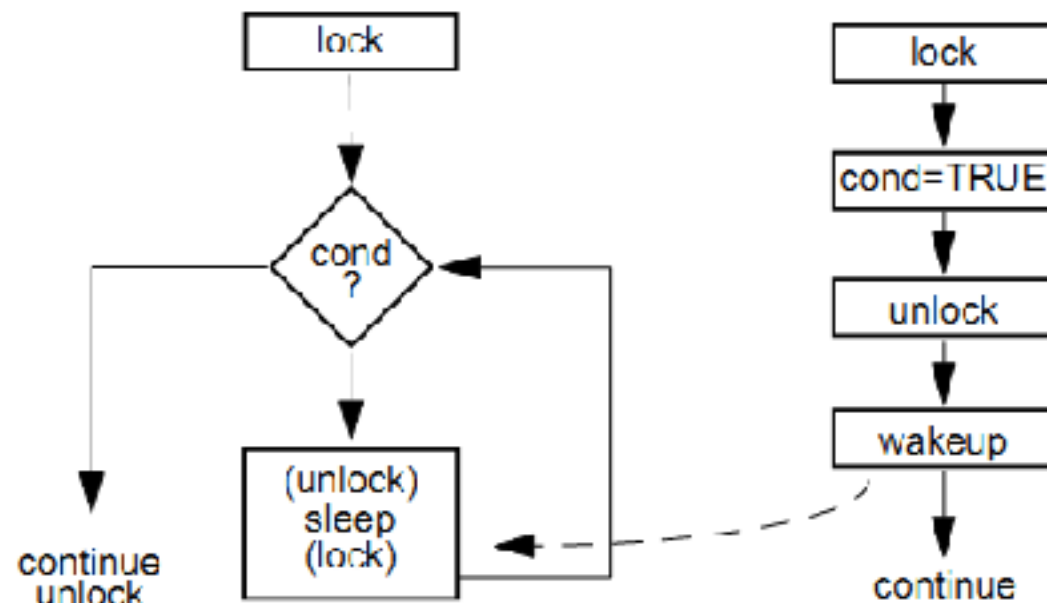


```
void *producer(void *arg) {  
    pthread_mutex_lock(&mutex);  
    put();  
    pthread_cond_signal(&fill);  
    pthread_mutex_unlock(&mutex);  
    return NULL;  
}  
  
void put() {  
    value=1;  
}
```

consumer thread



```
void *consumer(void *arg) {  
    int tmp=0;  
    pthread_mutex_lock(&mutex);  
    while (value == 0) {  
        pthread_cond_wait(&fill, &mutex);  
    }  
    get();  
    pthread_mutex_unlock(&mutex);  
    return NULL;  
}  
  
void get() {  
    value=0;  
}
```



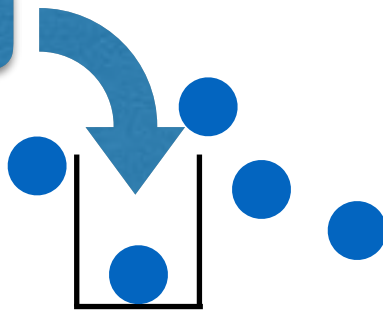
Using a Condition Variable

What happens if the value is empty?
This is where we need condition variables.

Producer consumer model

08_producer_loop.cpp

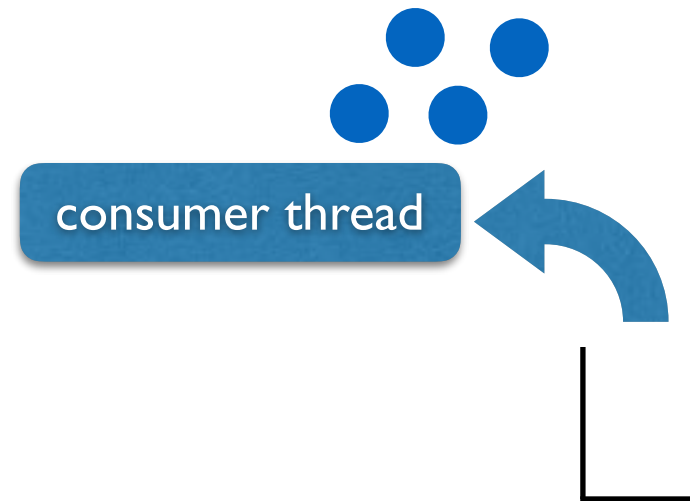
producer thread



```
void *producer(void *arg) {
    for (int i=0; i<loops; i++) {
        pthread_mutex_lock(&mutex);
        while (value != 0) {
            pthread_cond_wait(&empty, &mutex);
        }
        put(i);
        pthread_cond_signal(&fill);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Now let us consider the case where the producer puts multiple values. This requires an “end” flag to indicate the last value. We use a conditional variable to signal if full.

consumer thread

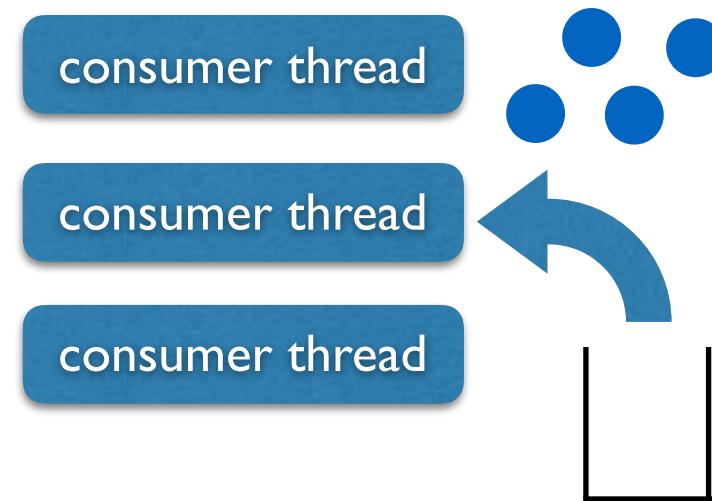
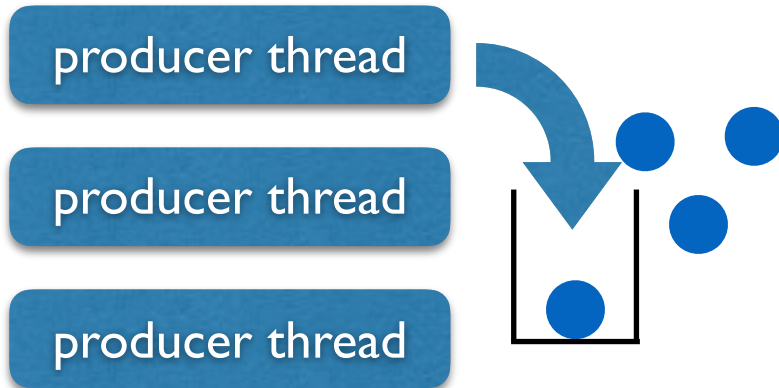


```
void *consumer(void *arg) {
    int tmp=0;
    while (tmp != end) {
        pthread_mutex_lock(&mutex);
        while (value == 0) {
            pthread_cond_wait(&fill, &mutex);
        }
        tmp=get();
        pthread_cond_signal(&empty);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

The consumer loops until it reaches the “end” value. We use a conditional variable to signal if empty.

Producer consumer model

09_multiple.cpp



```
for (int i=0; i<producers; i++) {
    pthread_create(&pid[i], NULL, producer, (void *) thread_id);
    thread_id++;
}
for (int i=0; i<consumers; i++) {
    pthread_create(&cid[i], NULL, consumer, (void *) thread_id);
    thread_id++;
}
for (int i=0; i<producers; i++) {
    pthread_join(pid[i], NULL);
}
for (int i=0; i<consumers; i++) {
    pthread_mutex_lock(&mutex);
    while (value != 0)
        pthread_cond_wait(&empty, &mutex);
    put(end);
    pthread_cond_signal(&full);
    pthread_mutex_unlock(&mutex);
}
for (int i=0; i<consumers; i++) {
    pthread_join(cid[i], NULL);
}
```

Many threads are created for the producer and consumer.

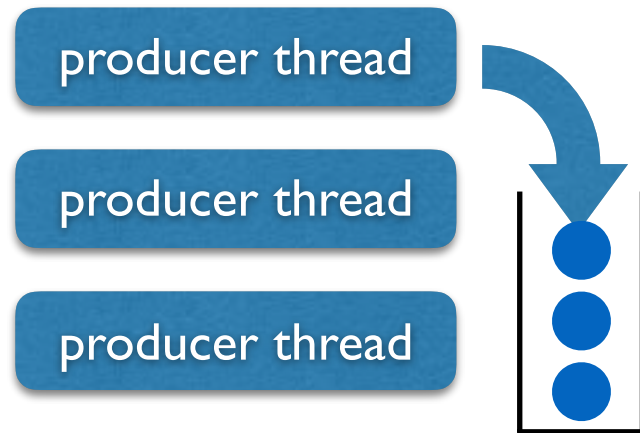
Since only one value can fit in the buffer conditional variables are used to check for full or empty buffers.

If all the producer threads have been joined and the value is not empty, then the "end" flag is inserted.

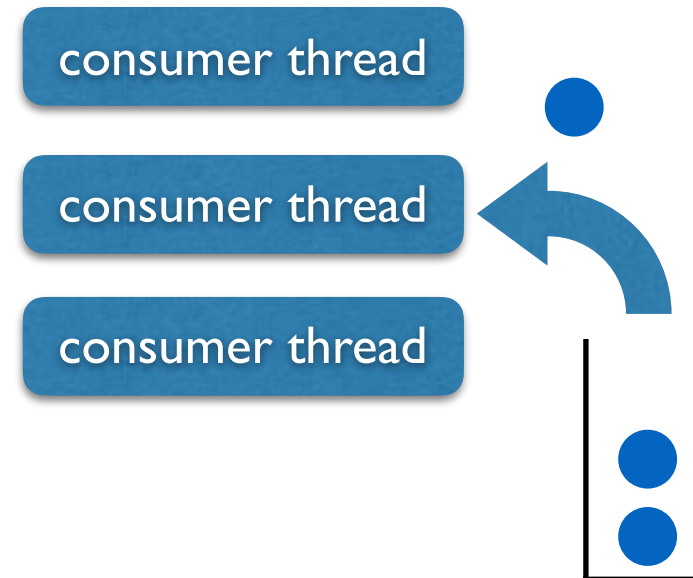
Finally the consumer threads are joined.

Producer consumer model

10_queue.cpp



```
void put(int value) {  
    queue[end_p]=value;  
    end_p=(end_p + 1) % max_queue;  
    count++;  
}  
  
int get() {  
    int tmp=queue[begin_p];  
    queue[begin_p]=0;  
    begin_p=(begin_p + 1) % max_queue;  
    count--;  
    return tmp;  
}
```



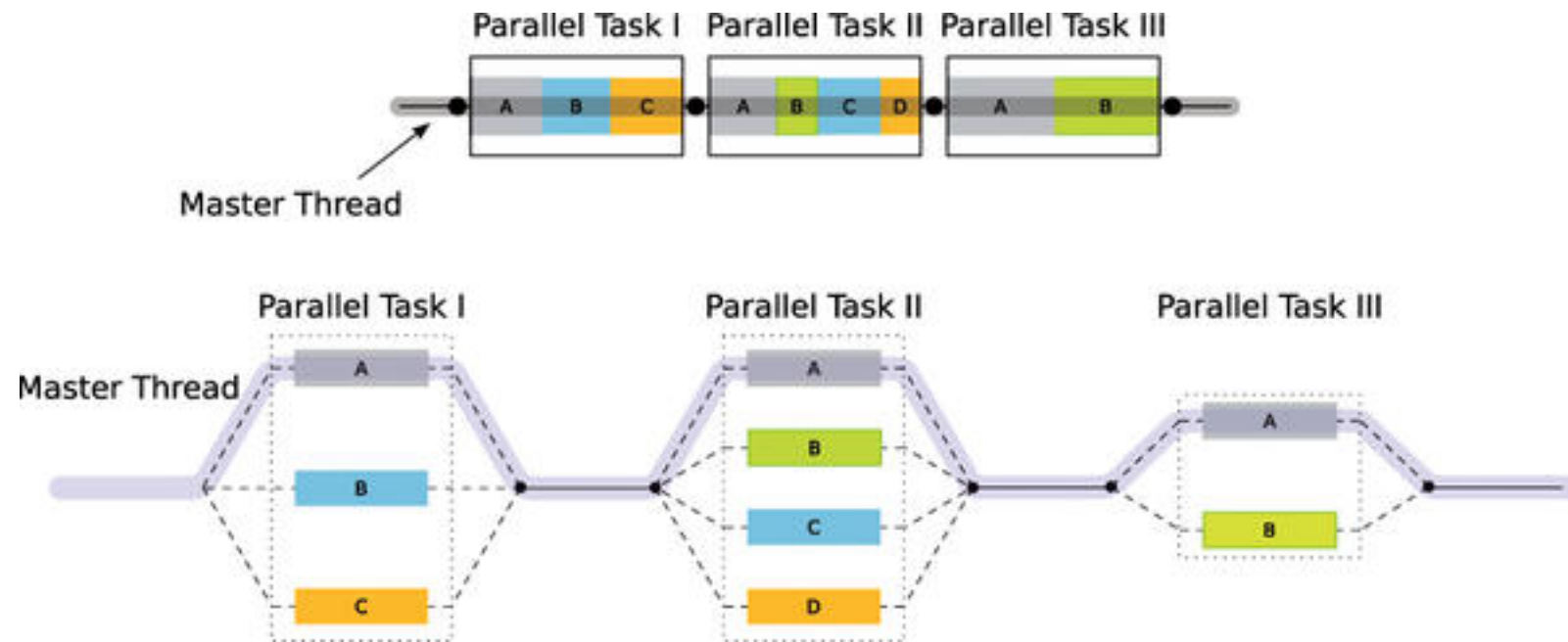
Finally, we enlarge the queue to hold many variables.

The producer and consumer put and get the data by FIFO.

We need both "begin" and "end" pointers for the queue.

**We also introduce a "count" for the queue
and signal "full" if count == max_queue
and signal "empty" if count == 0.**

OpenMP



```
#pragma omp parallel for
for (int i=0; i<n; i++) {
    b[i] = -a[i];
}
#pragma omp parallel for
for (int i=0; i<n; i++) {
    c[i] = 2*b[i]+a[i];
}
```

OpenMP: parallel for

```
#include <cstdlib>
#include <cstdio>
#include <sys/time.h>
int main(int argc, char ** argv) {
    struct timeval tic, toc;
    int n = atoi(argv[1]);
    double * a = new double [n];
    double * b = new double [n];
    gettimeofday(&tic, NULL);
#pragma omp parallel for
    for (int i=1; i<n; i++) {
        b[i] = (a[i] + a[i-1]) / 2.0;
    }
    gettimeofday(&toc, NULL);
    printf("%lf s\n",toc.tv_sec-tic.tv_sec+(toc.tv_usec-tic.tv_usec)*1e-6);
    delete[] a;
    delete[] b;
}
```

```
>g++ -fopenmp step01.cpp
>./a.out 1000000
```


OpenMP: barrier

```
#include <stdio>
#include <omp.h>
int main() {
    int x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x );
        }
    }
    #pragma omp barrier
    if (omp_get_thread_num() == 0) {
        printf("2: Thread# %d: x = %d\n", omp_get_thread_num(), x );
    } else {
        printf("3: Thread# %d: x = %d\n", omp_get_thread_num(), x );
    }
}
}
```

OpenMP: nowait

```
#include <cmath>
#include <cstdlib>
#include <stdio>
#include <sys/time.h>
int main(int argc, char ** argv) {
    struct timeval tic, toc;
    int n = atoi(argv[1]);
    double * a = new double [n];
    double * b = new double [n];
    double * y = new double [n];
    double * z = new double [n];
    gettimeofday(&tic, NULL);
#pragma omp parallel
    {
#pragma omp for nowait
        for (int i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;
#pragma omp for nowait
        for (int i=0; i<n; i++)
            y[i] = sqrt(z[i]);
    }
    gettimeofday(&toc, NULL);
    printf("%lf s\n", toc.tv_sec-tic.tv_sec+(toc.tv_usec-tic.tv_usec)*1e-6);
    delete[] a;
    delete[] b;
    delete[] y;
    delete[] z;
}
```

OpenMP: lastprivate

```
#include <stdio>
#include <omp.h>
int main() {
    int jlast, klast;
#pragma omp parallel
    {
#pragma omp for collapse(2) lastprivate(jlast, klast)
        for (int k=1; k<=2; k++) {
            for (int j=1; j<=3; j++) {
                jlast = j;
                klast = k;
            }
        }
#pragma omp single
        printf("%d %d\n", klast, jlast);
    }
}
```

OpenMP: sections

```
#include <stdio>
#include <omp.h>
int main() {
    int section_count = 0;
    omp_set_dynamic(0);
    omp_set_num_threads(2);
#pragma omp parallel
#pragma omp sections
    {
#pragma omp section
    {
        section_count++;
        printf("section_count %d\n",section_count);
    }
#pragma omp section
    {
        section_count++;
        printf("section_count %d\n",section_count);
    }
    }
}
```

OpenMP: Fibonacci

```
#include <stdlib.h>
#include <stdio.h>
int fib(int n) {
    int i,j;
    if (n<2) return n;
#pragma omp task shared(i)
    i = fib(n-1);
#pragma omp task shared(j)
    j = fib(n-2);
#pragma omp taskwait
    return i+j;
}

int main(int argc, char ** argv) {
    int n = atoi(argv[1]);
    printf("%d\n",fib(n));
}
```

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

OpenMP: depend

```
#include <stdio>
int main() {
    int x = 1;
    #pragma omp task shared(x) depend(out: x)
    x = 2;
    #pragma omp task shared(x) depend(in: x)
    printf("x + 1 = %d\n", x+1);
    #pragma omp task shared(x) depend(in: x)
    printf("x + 2 = %d\n", x+2);
}
```

OpenMP: atomic

```
#include <stdio>
int main() {
    float x[10];
    int index[1000];
    for (int i=0; i<1000; i++) {
        index[i] = i % 10;
    }
    for (int i=0; i<10; i++)
        x[i] = 0.0;
#pragma omp parallel for shared(x, index)
    for (int i=0; i<1000; i++) {
#pragma omp atomic update
        x[index[i]]++;
    }
    for (int i=0; i<10; i++)
        printf("%d %f\n",i, x[i]);
}
```

OpenMP: ordered

```
#include <stdio>
int main() {
#pragma omp parallel for ordered schedule(dynamic)
    for (int i=0; i<100; i+=5) {
#pragma omp ordered
        printf("%d\n",i);
    }
}
```

OpenMP: threadprivate

```
#include <stdio>
int counter = 0;
#pragma omp threadprivate(counter)

int main() {
#pragma omp parallel for
    for (int i=0; i<100; i++) {
        counter++;
    }
    printf("%d\n",counter);
}
```