



西北工业大学



# Parallel Computing

Programming with PThreads





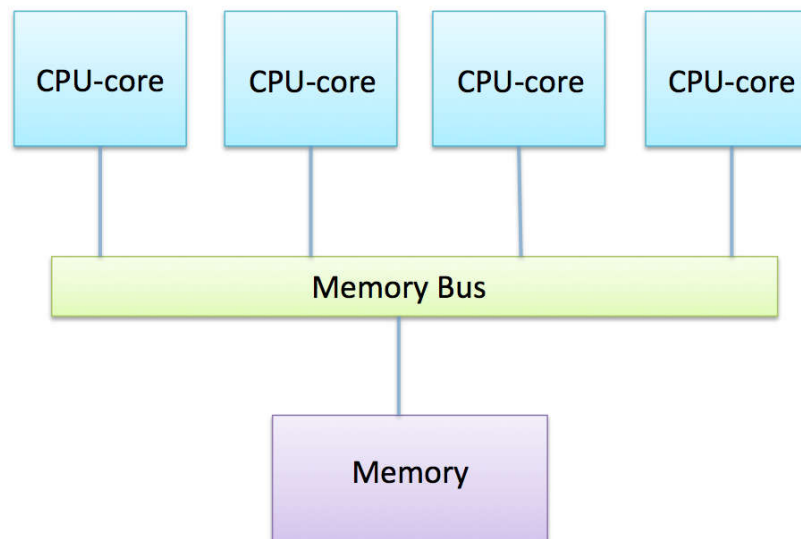
# Outline

---

1. Shared Memory Programming: Overview
2. Parallel Programming with PThreads
  - Threads and Processes
  - POSIX Pthreads API
  - Critical section & thread synchronization
  - False Sharing
3. Summary

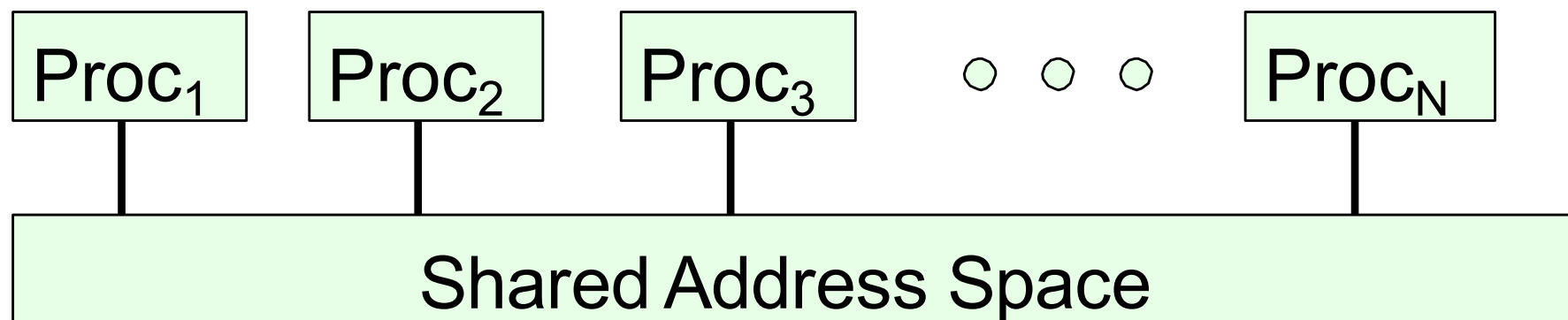


# Shared Memory Hardware

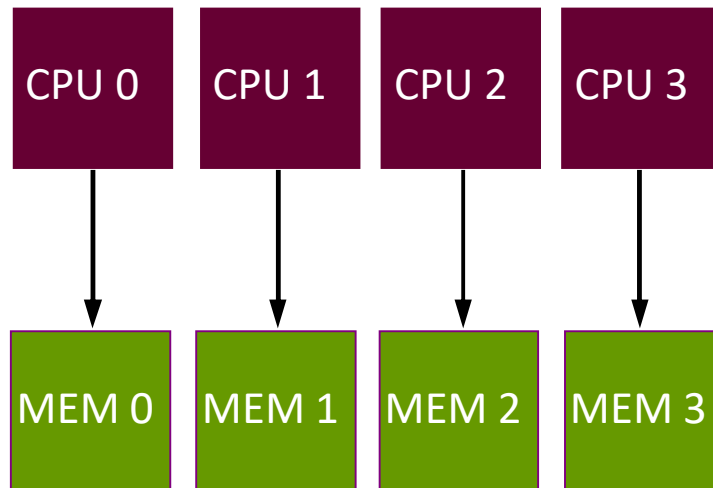


# Shared memory Computers

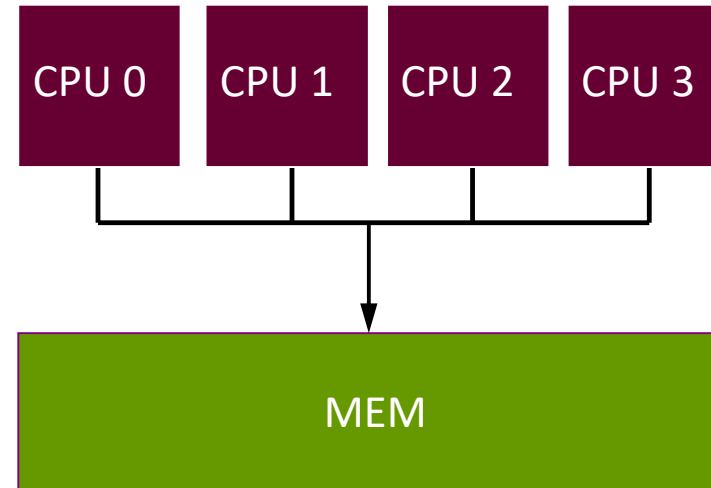
- **Shared memory computer** : any computer composed of multiple processing elements that share an address space. Two Classes:
  - **Symmetric multiprocessor (SMP)**: a shared address space with “equal-time” access for each processor, and the OS treats every processor the same way.
  - **Non Uniform address space multiprocessor (NUMA)**: different memory regions have different access costs ... think of memory segmented into “Near” and “Far” memory.



# Shared vs. Distributed Memory



distributed



shared

**distributed memory** refers to a multiprocessor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors.

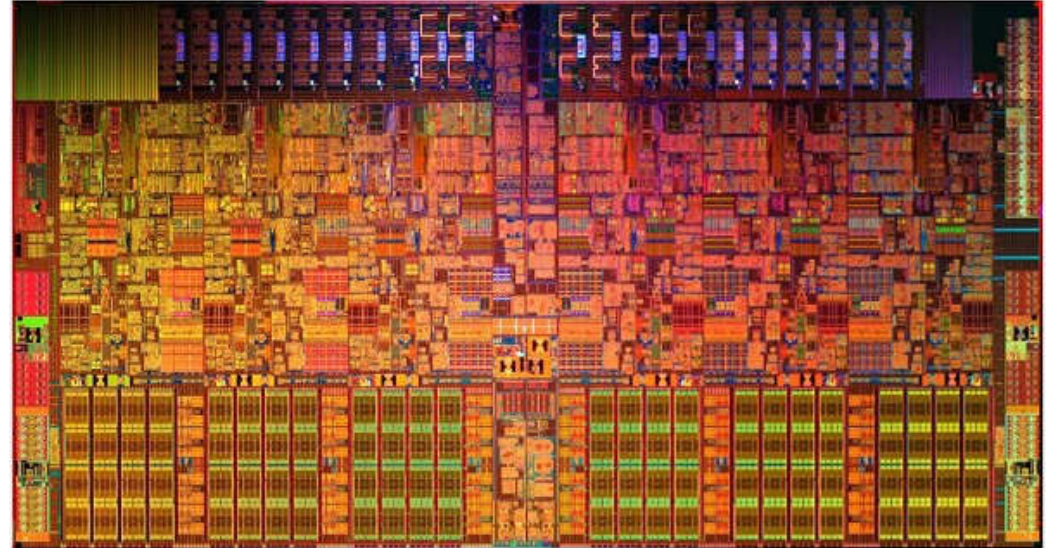
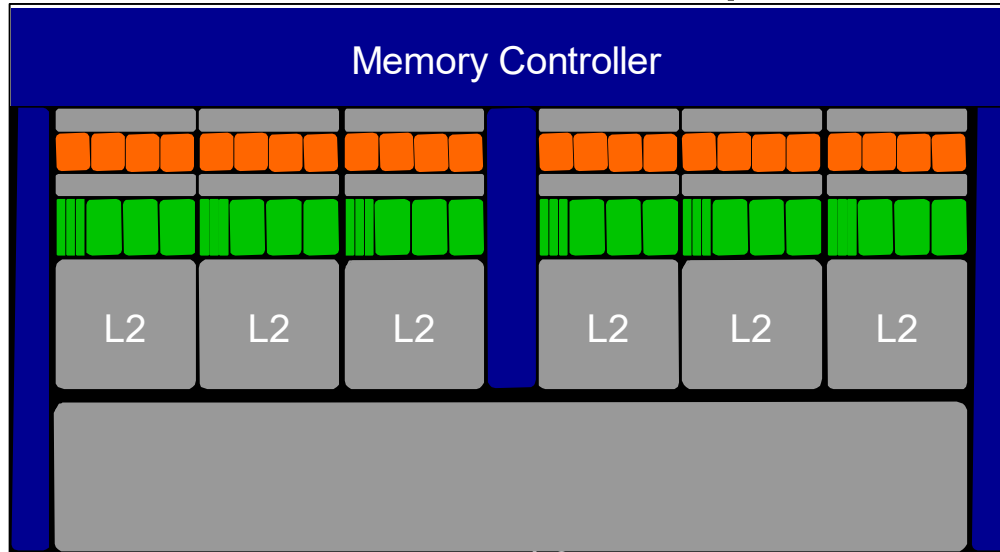
**a shared memory** multiprocessor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.





# Shared memory machines: SMP

Intel® Core™ i7-970 processor: Often called an SMP, but is it?



- 6 cores, 2-way multithreaded, 6-wide superscalar, quad-issue, 4-wide SIMD (on 3 of 6 pipelines)
- 4.5 KB (6 x 768 B) “Architectural” Registers, 192 KB (6 x 32 KB) L1 Cache, 1.5 MB (6 x 256 KB) L2 cache, 12 MB L3 Cache
- MESIF Cache Coherence, Processor Consistency Model
- 1.17 Billion Transistors on 32 nm process @ 2.6 GHz

Cache hierarchy means different processors have different costs to access different address ranges .... It's NUMA

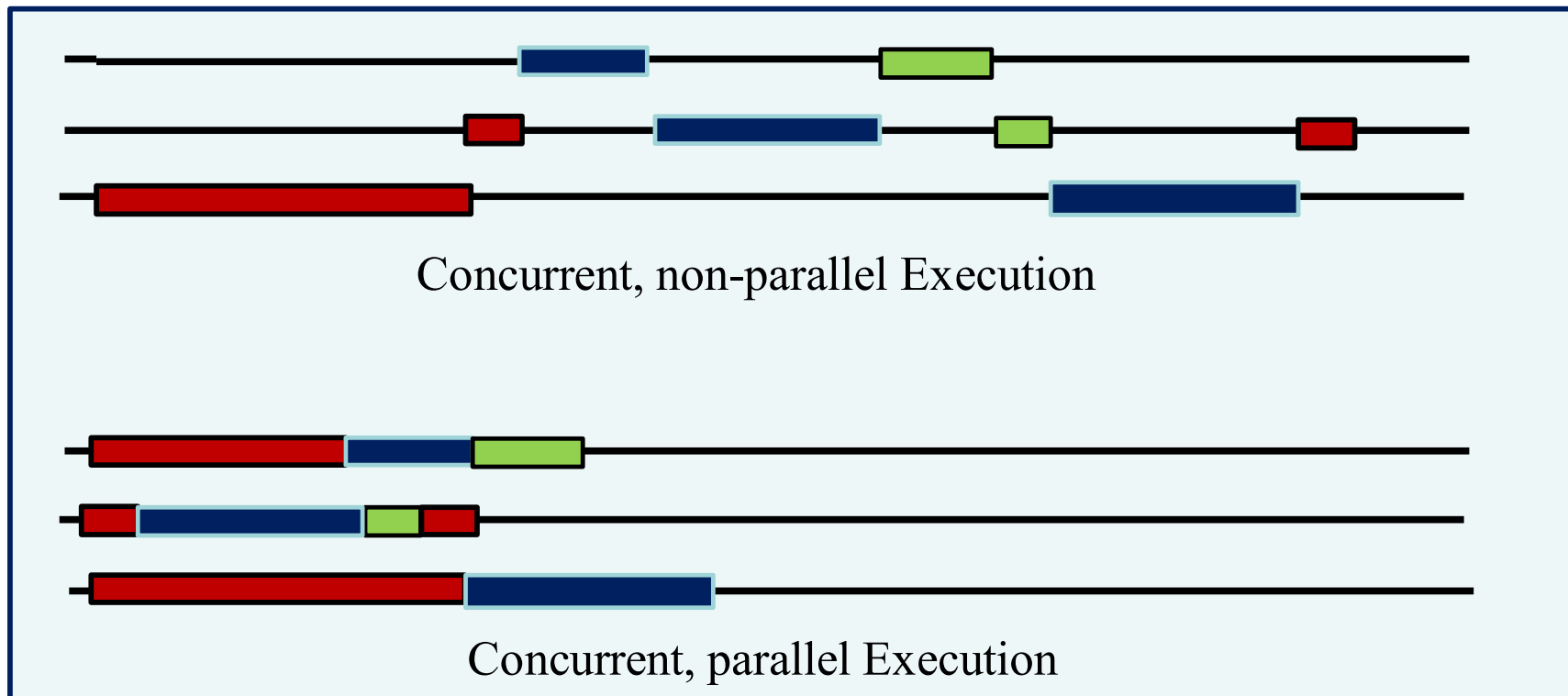


# Parallel Programming with Threads



# Concurrency vs. Parallelism

- Two important definitions:
  - **Concurrency**: A condition of a system in which multiple tasks are *logically* active at one time.
  - **Parallelism**: A condition of a system in which multiple tasks are actually active at one time.





# Concurrent vs. Parallel applications

- We distinguish between two classes of applications that exploit the concurrency in a problem:

Concurrent application: An application for which computations **logically** execute simultaneously due to the semantics of the application.

- The problem is fundamentally concurrent.

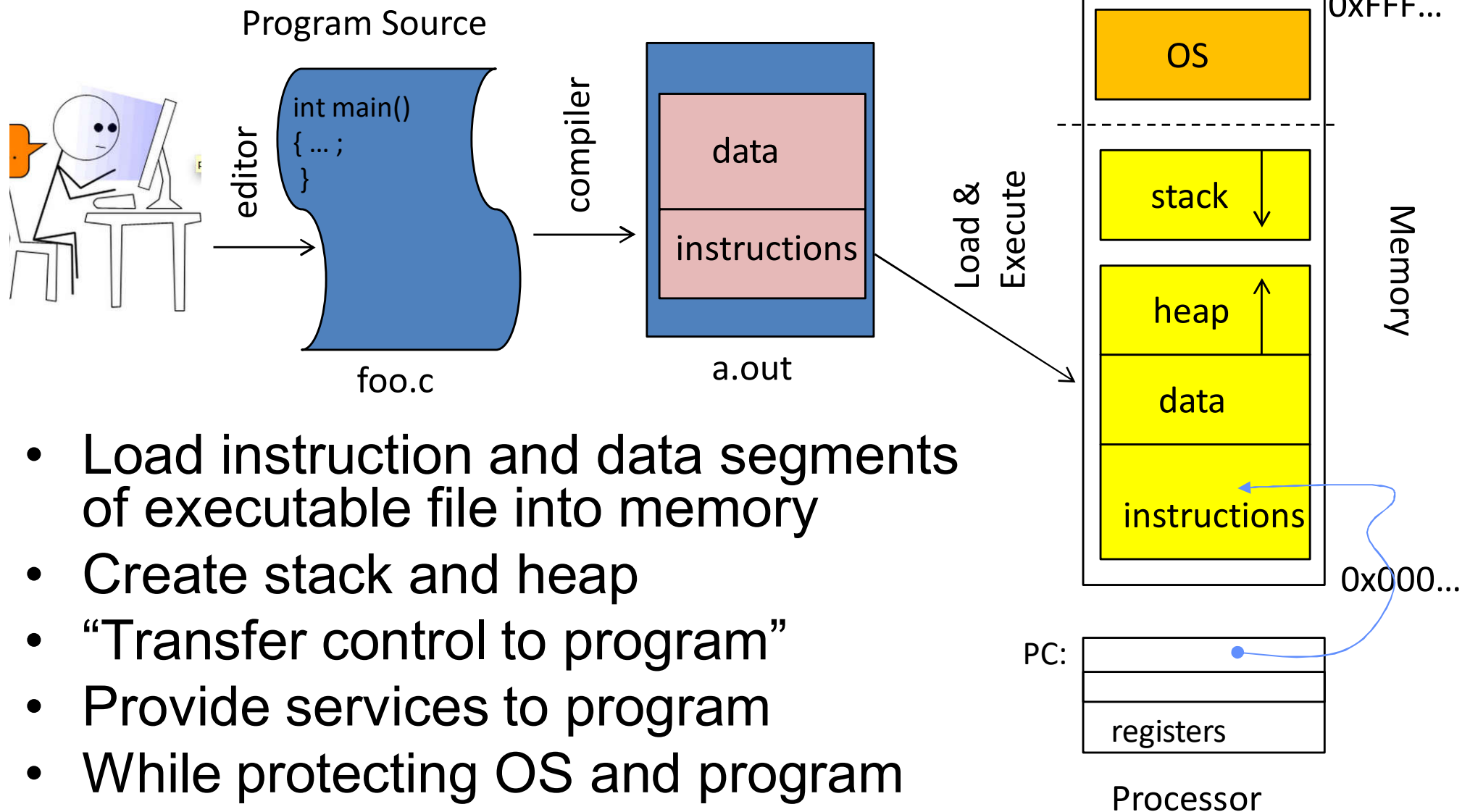
Parallel application: An application for which the computations **actually** execute simultaneously in order to complete a problem in less time.

- The problem doesn't inherently require concurrency ... you can state it sequentially.



# Run Programs

Executable





# What is a Process?

---

- A process contains all the information needed to execute the program
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap
- Each process has its own address space

# What is a thread?

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process
- Each thread has its own run time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

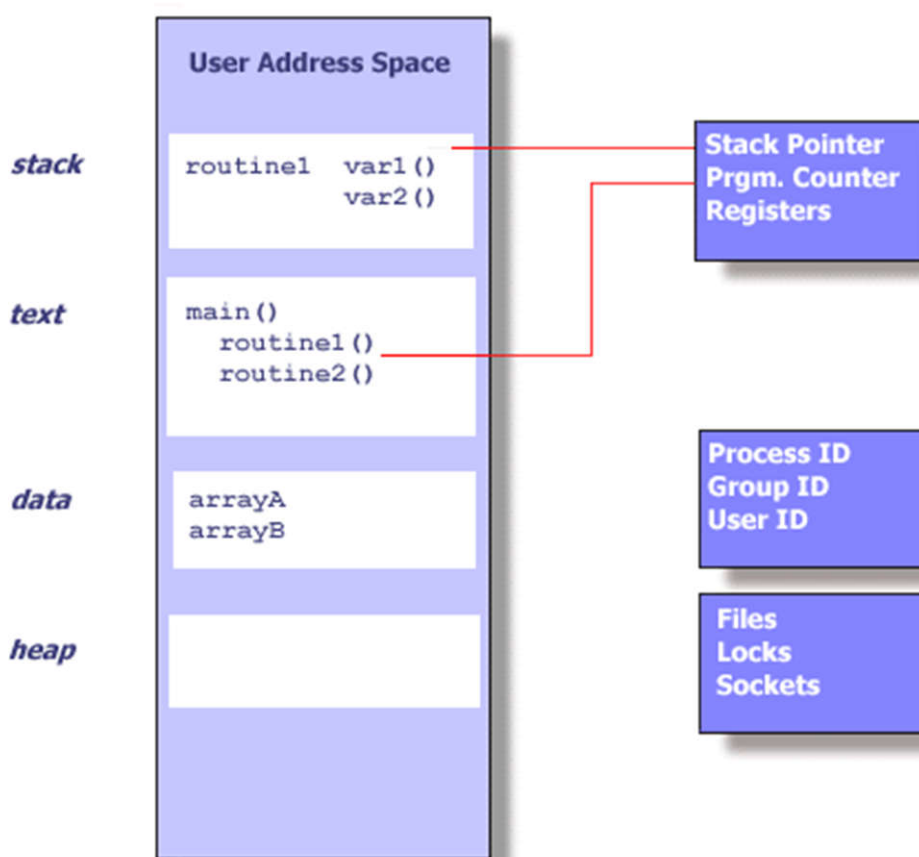
# Process vs. Thread

- **A process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
- **A thread** of execution is the smallest unit of processing that can be scheduled by an operating system.
- Differences between threads and processes:
  - A thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory.
  - The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
  - Different processes do not share these resources.

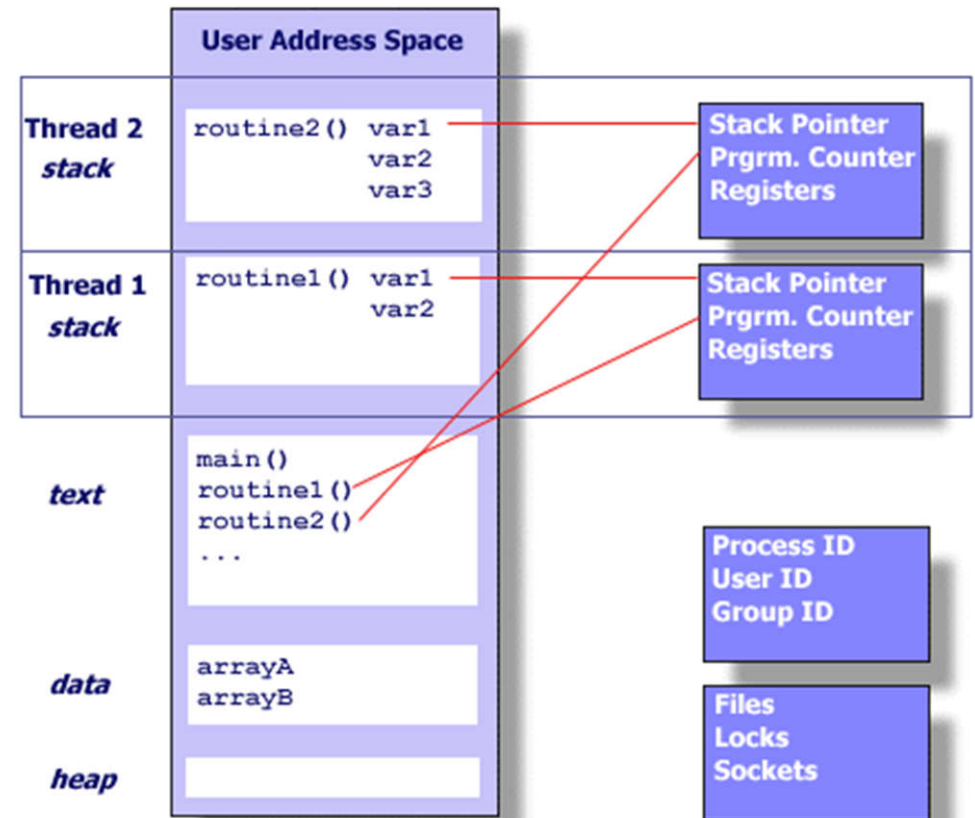
[http://en.wikipedia.org/wiki/Process\\_\(computing\)](http://en.wikipedia.org/wiki/Process_(computing))



# Programming shared memory computers



Single Process

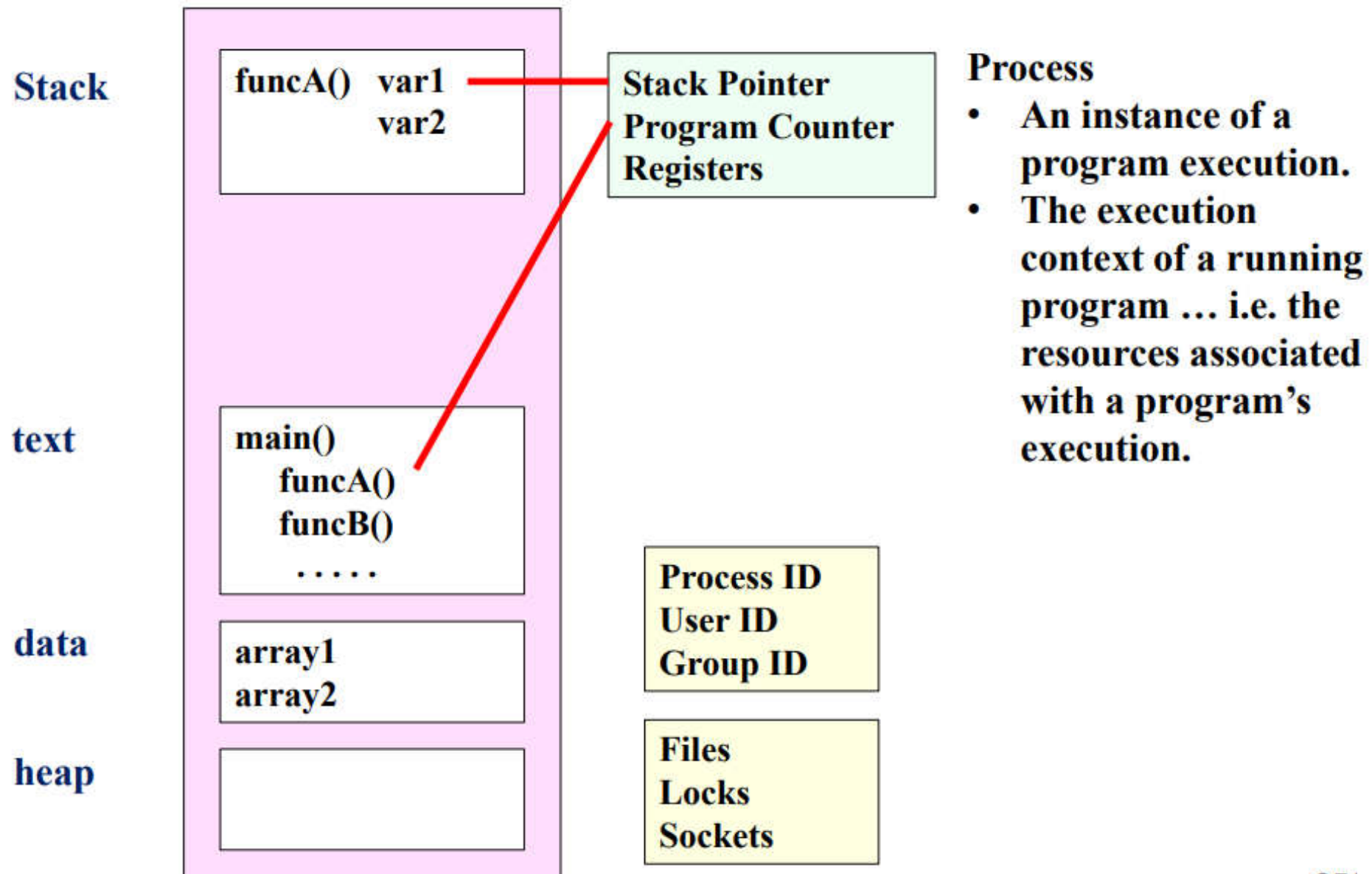


Single Process with 2 threads



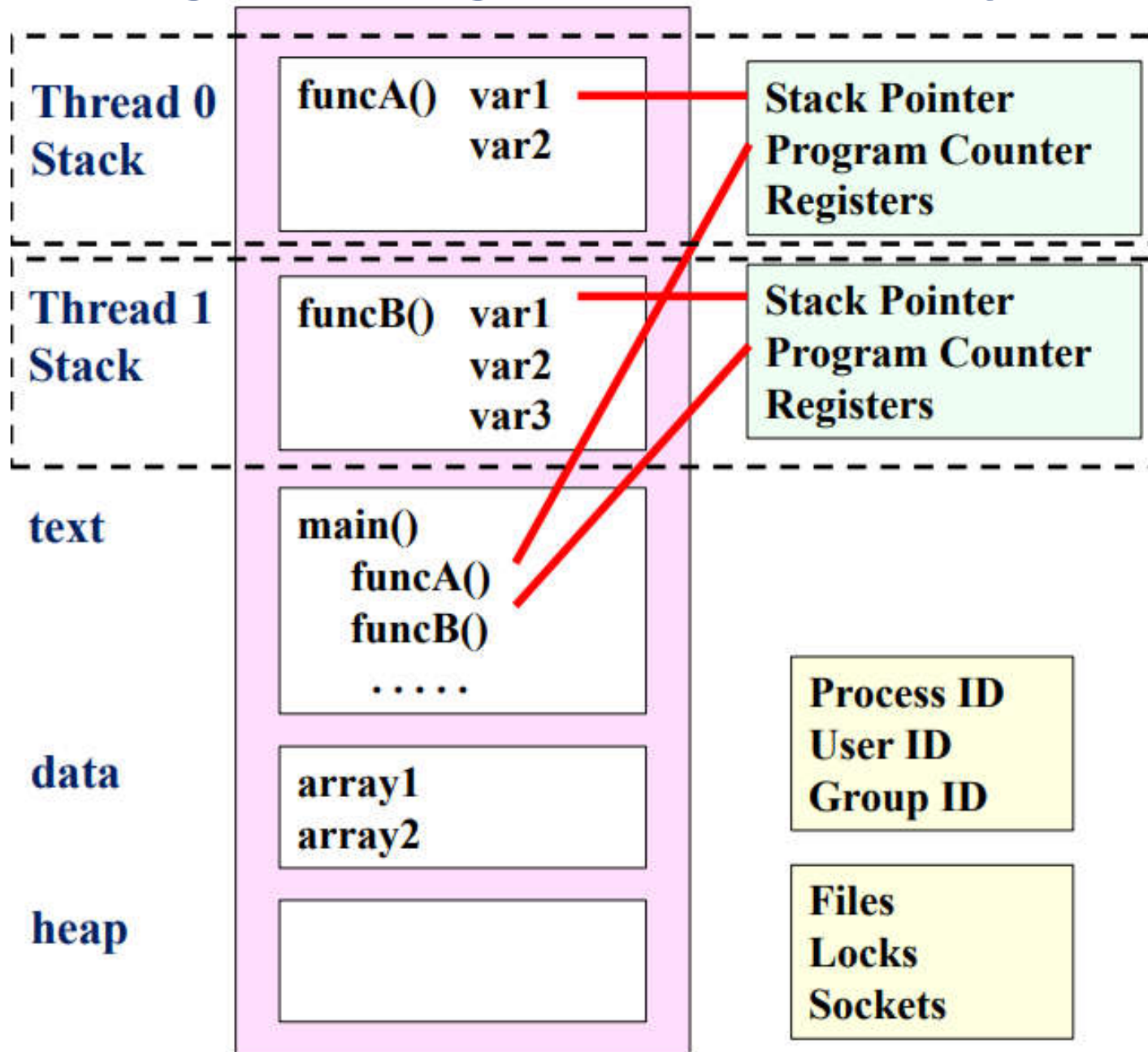


# Programming shared memory computers





# Programming shared memory computers

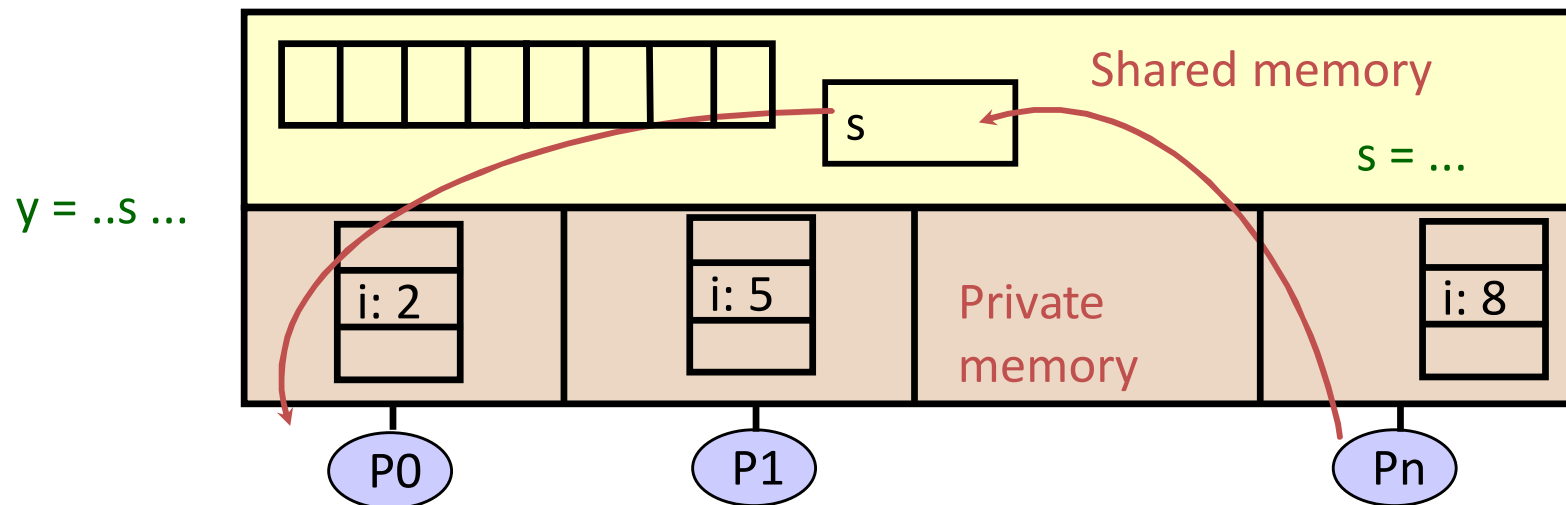


## Threads:

- Threads are "light weight processes"
- Threads share Process state among multiple threads ... this greatly reduces the cost of switching context.

# Programming Model 1: Shared Memory

- Program is a collection of threads of control.
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of **private variables**, e.g., local stack variables
- Also a set of **shared variables**, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate **implicitly** by writing and reading shared variables.
  - Threads coordinate by **synchronizing** on shared variables



# Shared Memory Programming

## Several Thread Libraries/systems

- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - <http://www.openMP.org>
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C “ilk”
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language



# Parallel Programming using Pthreads



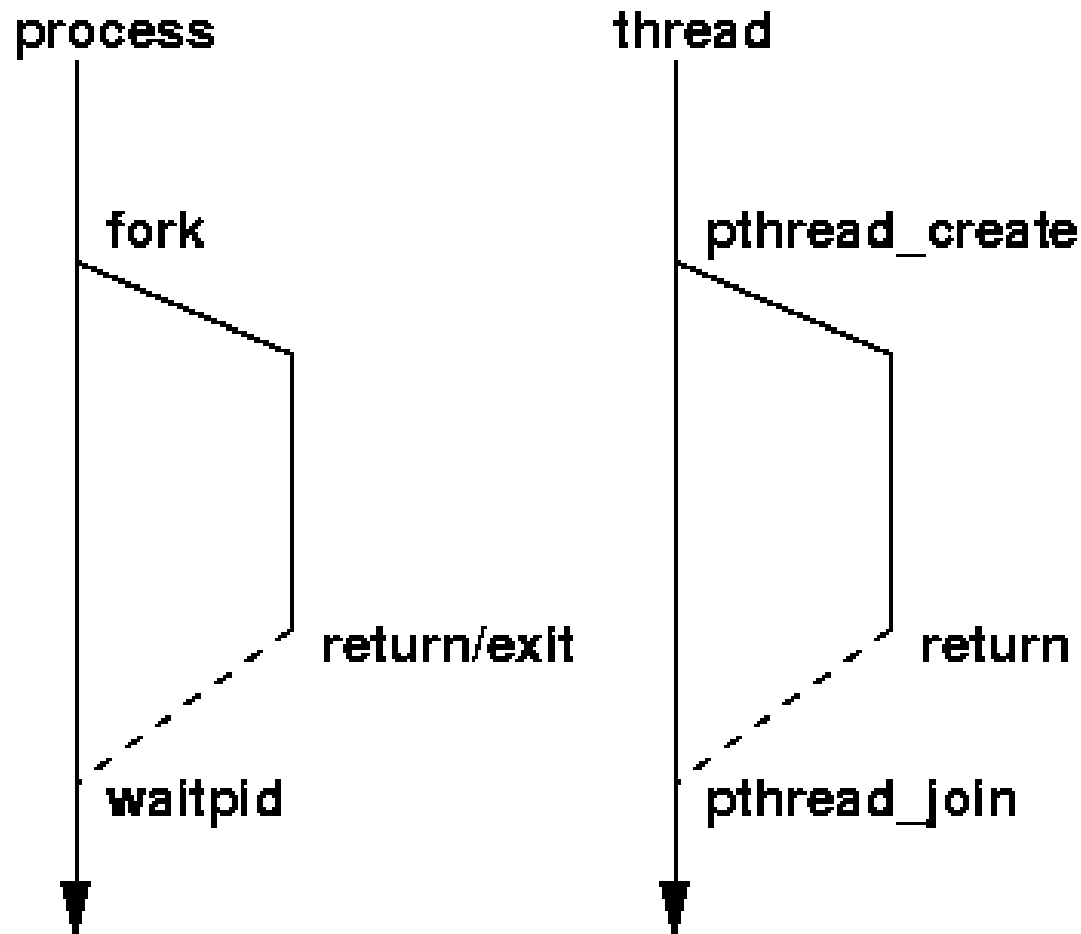
# Overview of POSIX Threads

- POSIX: *Portable Operating System Interface for UNIX*
  - Interface to Operating System utilities
- PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - In CSIL, compile a c program with `gcc -lpthread`
- PThreads contain support for
  - Creating parallelism and synchronization
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread





# Creation of Unix processes vs. Pthreads





# C function for starting a thread

pthread.h



pthread\_t



One object for  
each thread.

Allocate before calling.

```
int pthread_create (  
    pthread_t* thread_p          /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p                  /* in */  
);
```

The function that the thread is to  
run.

Pointer to the argument that should  
be passed to the function *start\_routine*.



# A closer look (1)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

We won't be using, so we just pass NULL.

Allocate before calling.



## A closer look (2)

```
int pthread_create (  
    pthread_t* thread_p /* out */,  
    const pthread_attr_t* attr_p /* in */,  
    void* (*start_routine) ( void ) /* in */,  
    void* arg_p /* in */ );
```

Pointer to the argument that should  
be passed to the function *start\_routine*.

The function that the thread is to run.



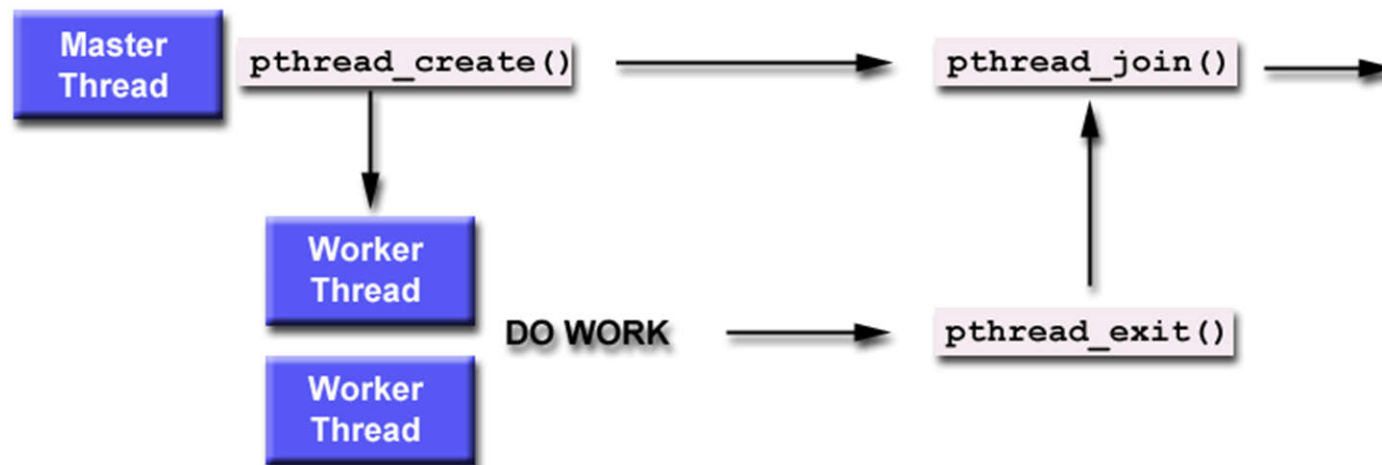
# Function started by pthread\_create

- Prototype:  
`void* thread_function ( void* args_p ) ;`
- Void\* can be cast to any pointer type in C.
- So args\_p can point to a list containing one or more values needed by thread\_function.
- Similarly, the return value of thread\_function can point to a list of one or more values.

# Wait for Completion of Threads

```
pthread_join(pthread_t *thread, void **result);
```

- Wait for specified thread to finish. Place exit value into \*result.
- We call the function `pthread_join` once for each thread.
- A single call to `pthread_join` will wait for the thread associated with the `pthread_t` object to complete.

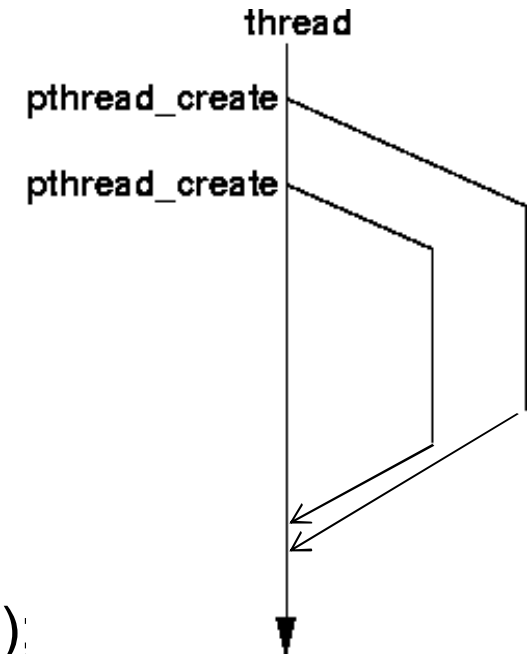




# Example of Pthreads with join

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
    printf("Thread%d: Hello World!\n", id);
}

void main (){
    pthread_t thread0, thread1;
    pthread_create(&thread0, NULL, PrintHello, (void *) 0);
    pthread_create(&thread1, NULL, PrintHello, (void *) 1);
    pthread_join(thread0, NULL);
    pthread_join(thread1, NULL);
}
```



**Compiling:**

gcc -g -Wall -o pth\_hello pth\_hello . c -lpthread

# Some More Pthread Functions

- `pthread_yield()` ;
  - Informs the scheduler that the thread is willing to yield
- `pthread_exit(void *value)` ;
  - Exit thread and pass value to joining thread (if exists)

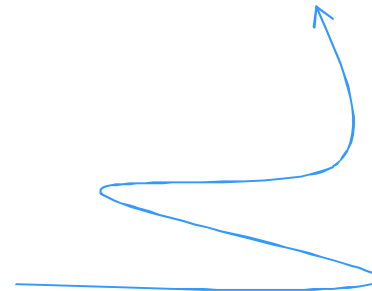
Others:

- `pthread_t me; me = pthread_self()` ;
  - Allows a pthread to obtain its own identifier  
`pthread_t thread;`
- Synchronizing access to shared variables
  - `pthread_mutex_init, pthread_mutex_[un]lock`
  - `pthread_cond_init, pthread_cond_[timed]wait`

# Compiling a Pthread program

```
gcc -g -Wall -o pthread_hello pthread_hello.c -lpthread
```

link in the Pthreads library



Homework: accumulate number 1 to 100.

1. Use two threads to accumulate 1 to 49 and 50 to 100 respectively.
2. Then add the results that return from threads

```
for(int i=1;i<=100;i++){  
    result = i+ result;  
}
```



# Matrix-Vector Multiplication with Pthreads

- Definition: Let  $A$  be an  $[m \times n]$  matrix, and  $x$  be a vector of an  $[n \times 1]$ , then  $y$  will be a vector with the dimensions  $[m \times 1]$ .

Then 
$$y_j = \sum_{t=1}^m a_{it}x_t = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{m-1,1}b_{m-1}$$

$$\begin{bmatrix} a_{00} & \cdots & a_{0j} & \cdots & a_{0,n-1} \\ \vdots & & \vdots & & \vdots \\ \mathbf{a_{i0}} & \cdots & \mathbf{a_{ij}} & \cdots & \mathbf{a_{i,n-1}} \\ \vdots & & \vdots & & \vdots \\ a_{m-1,0} & \cdots & a_{m-1,j} & \cdots & a_{m-1,n-1} \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ \mathbf{y_j} \\ \vdots \\ y_{m-1} \end{bmatrix}$$

$a_{00}$	$a_{01}$	$\cdots$	$a_{0,n-1}$
$a_{10}$	$a_{11}$	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{i0}$	$a_{i1}$	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

$x_0$
$x_1$
$\vdots$
$x_{n-1}$

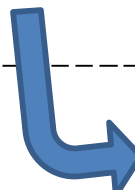
 $=$ 

$y_0$
$y_1$
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$
$\vdots$
$y_{m-1}$

# Serial Pseudo-code

pseudo-code for a *serial* program for matrix-vector multiplication might look like this

```
/* For each row of A */  
for (i=0; i<m; i++){  
    y[i] = 0.0;  
    /* For each element of the row and each element of x */  
    for (j=0; j<n; j++){  
        y[j] += A[i][j]*x[j];  
    }  
}
```


$$y_i = \sum_{j=0}^{n-1} a_{ij}x_j$$

# Using 3 Pthread, 6 elements

- suppose that  $m = n = 6$
- the number of threads is 3

Thread	Components of $y$
0	$y[0], y[1]$
1	$y[2], y[3]$
2	$y[4], y[5]$

Thread 0

```
y[0] = 0.0;  
for (j = 0; j < n; j++)  
    y[0] += A[0][j]* x[j];
```



general case

```
y[i] = 0.0;  
for (j = 0; j < n; j++)  
    y[i] += A[i][j]*x[j];
```





# Pthreads matrix-vector multiplication

- assume that both  $m$  and  $n$  are evenly divisible by  $t(\text{thread\_count})$
- Divide the  $A$  by row
- each thread gets  $m/t$  components

```
void* Pth_mat_vect(void* rank) {
    long my_rank = (long) rank;
    int i, j;
    int local_m = m/thread_count;
    int my_first_row = my_rank*local_m;
    int my_last_row = (my_rank+1)*local_m - 1;

    for (i = my_first_row; i <= my_last_row; i++) {
        y[i] = 0.0;
        for (j = 0; j < n; j++)
            y[i] += A[i][j]*x[j];
    }

    return NULL;
} /* Pth_mat_vect */
```

Pthreads matrix-vector multiplication



- More Straightforward because of shared memory
- Code only reads shared arrays ( $A$ ,  $x$ ), so no contention associated
- with shared updates of same memory location
- No thread communication



## Critical section & thread synchronization



## Estimating $\pi$ using n terms of A Maclaurin series: Serial Code

- estimate the value of  $\pi$

$$\pi = 4 \left( 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right).$$

The following *serial* code uses this formula:

```
double factor = 1.0;
double sum = 0.0;
for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
}
pi = 4.0*sum;
```

# POSIX Threads: Pacheco pthd pi.c

- First attempt:  
Parallelize similar to the way we did matrix-vector multiplication:  
Divide iterates in the *for loop* and make *sum* a shared variable.

```
*-----  
* Function:      Thread_sum,   Purpose: Add in the terms computed by the thread running this  
* In arg:       rank  
* Ret val:      ignored  
* Globals in:   n, thread_count  
* Global in/out: sum  
*/  
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i, my_n = n/thread_count, my_first_i = my_n*my_rank, my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0)  
        factor = 1.0;  
    else  
        factor = -1.0;  
  
    for (i = my_first_i; i < my_last_i; i++, factor = -factor)  
        sum += factor/(2*i+1);  
  
    return NULL;  
} /* Thread_sum */
```



# Program run with 2 threads, dual core processor

	<i>n</i>			
	$10^5$	$10^6$	$10^7$	$10^8$
$\pi$	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

- For two threads, as  $n$  " accuracy of  $\pi$  "
- But, as # threads " accuracy of  $\pi$  #







# Simplify the issue

- Suppose that we have two threads
- private variable **y** and shared variable **x**

Thread function

```
y = Compute(my_rank);  
x = x + y;
```

Suppose we use two threads

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute()	Started by main thread
3	Assign y = 1	Call Compute()
4	Put x=0 and y=1 into registers	Assign y = 2
5	Add 0 and 1	Put x=0 and y=2 into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

# Definition

- Race condition
  - Several threads access and manipulate the same data **concurrently**
  - Outcomes of the execution **depends** on the **order** in which the access take place
- How to remove **Race Condition**?
  - **Serial execution**

## Synchronization Solutions

1. Busy waiting
2. Mutex (lock)
3. Semaphore
4. Conditional Variables

# Critical section problem

- Consider system of  $n$  threads  $\{t_0, t_1, \dots, t_{n-1}\}$
- Each thread has **critical section** segment of code
  - thread may be changing common variables, updating table, writing file, etc
  - When one thread in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each thread **must ask permission** to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**



# Critical section

- General structure of thread  $T_i$

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

# Requirements to solutions

- **Mutual exclusion**

- If thread  $T_i$  is executing in its critical section, then **no other threads** can be executing in their critical sections

- **Progress**

- If no thread is executing in its critical section and there exist some threads that wish to enter their critical section, then the **selection of the threads** that will enter the critical section next **cannot be postponed indefinitely**

- **Bounded waiting**

- A **bound** must exist on the **number of times that other threads are allowed** to enter their critical sections after a thread has made a request to enter its critical section and before that request is granted
  - Assume that each thread executes at a nonzero speed
  - No assumption concerning **relative speed** of the  $n$  threads

# Busy Waiting

- **Busy waiting** is a technique in which a thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.

A simple approach is the use of **a flag variable**.

```
1  y = Compute(my_rank);  
2  while (flag != my_rank);  
3  x = x + y;  
4  flag++;
```

**Cons:** Waste CPU resource. Sometime not safe with compiler optimization.



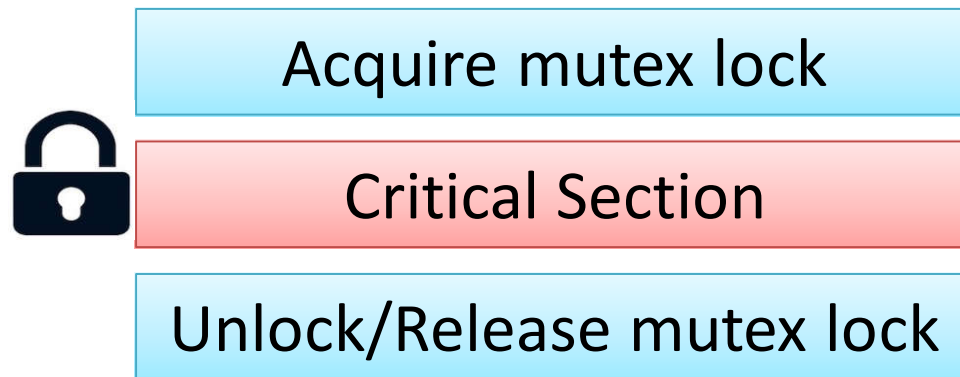


# Pthreads global sum with busy-waiting

```
1 void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8
9     if (my_first_i % 2 == 0)
10         factor = 1.0;
11     else
12         factor = -1.0;
13
14     for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
15         while (flag != my_rank);
16         sum += factor/(2*i+1);
17         flag = (flag+1) % thread_count;
18     }
19
20     return NULL;
21 } /* Thread_sum */
```

# Mutex

- Mutex is an abbreviation of *mutual exclusion*
- Mutex (*mutual exclusion*) is a special type of variable used to restrict access to a critical section to a single thread at a time.

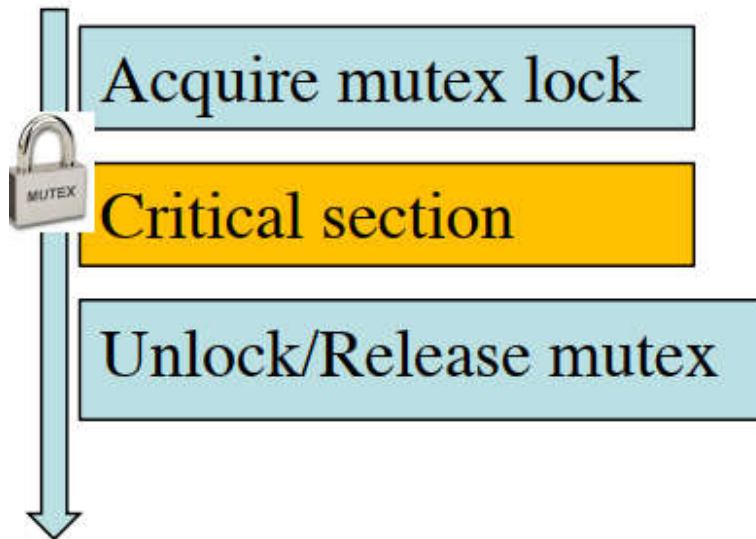


- When A thread waits on a mutex/lock, CPU resource can be used by others.
- Only thread that has acquired the lock can release this lock

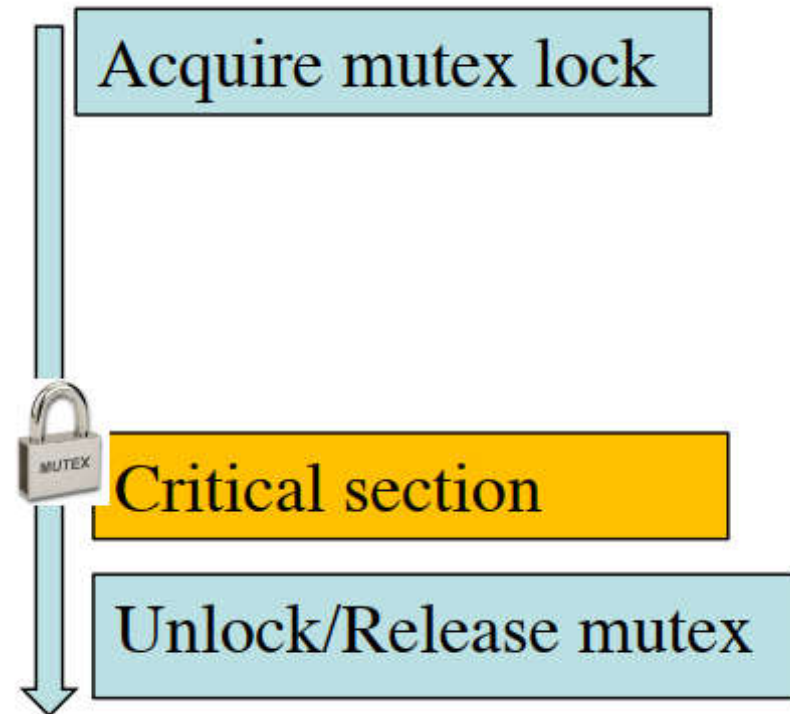


# Execution example with 2 threads

Thread 1



Thread 2



# Mutexes in Pthreads

- A special type for mutexes: `pthread_mutex_t`.

```
int pthread_mutex_init(  
    pthread_mutex_t*      mutex_p    /* out */,  
    const pthread_mutexattr_t* attr_p /* in */);
```

- To gain access to a critical section, call

```
int pthread_mutex_lock(pthread_mutex_t* mutex_p /* in/out */);
```

- To release mutex lock

```
int pthread_mutex_unlock(pthread_mutex_t* mutex_p /* in/out */);
```

- When finishing use of a mutex, call

```
int pthread_mutex_destroy(pthread_mutex_t* mutex_p /* in/out */);
```





# Global sum function that uses a mutex

```
1 void* Thread_sum(void* rank) {
2     long my_rank = (long) rank;
3     double factor;
4     long long i;
5     long long my_n = n/thread_count;
6     long long my_first_i = my_n*my_rank;
7     long long my_last_i = my_first_i + my_n;
8     double my_sum = 0.0;
9
10    if (my_first_i % 2 == 0)
11        factor = 1.0;
12    else
13        factor = -1.0;
14
15    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
16        my_sum += factor/(2*i+1);
17    }
18    pthread_mutex_lock(&mutex);
19    sum += my_sum;
20    pthread_mutex_unlock(&mutex);
21
22    return NULL;
23 }
```

# performance

- the ratio of the run-time of the single-threaded program with the multithreaded program is equal to the number of threads as long as the number of threads is no greater than the number of cores
- use busy-waiting, performance can degrade if there are more threads than cores

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread\_count},$$

**Table 4.1** Run-Times (in Seconds) of  $\pi$  Programs Using  $n = 10^8$  Terms on a System with Two Four-Core Processors

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38



# Semaphore

- A semaphore is **a variable or abstract data type** used to control access to a common resource by **multiple threads** and avoid critical section problems in a concurrent system.
- Semaphores can be thought of as a special type of ***unsigned int S***

Can only be accessed /modified via **two (atomic) operations** with the following semantics

```
wait (S) { //also called P()  
    while S <= 0 wait in a queue;  
    S--;  
}
```

```
post(S) { //also called V()  
    S++;  
    Wake up a thread that waits in the queue.  
}
```

# Syntax of Pthreads semaphore functions

```
#include <semaphore.h>
```

→ Semaphores are not part of Pthreads;  
you need to add this.

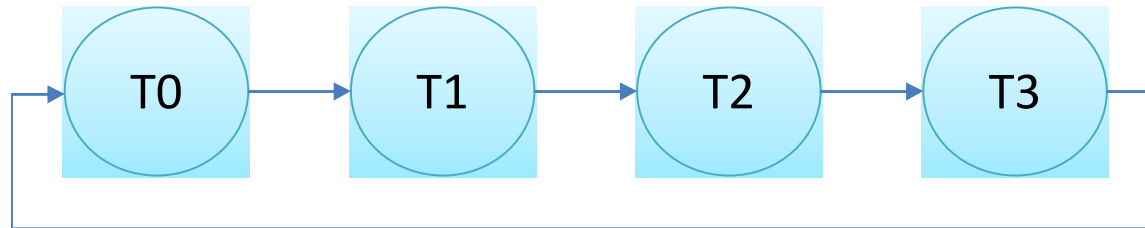
- Init a semaphore

```
int sem_init(  
    sem_t*      semaphore_p    /* out */,  
    int         shared         /* in  */,  
    unsigned    initial_val    /* in  */);
```

- semaphore functions

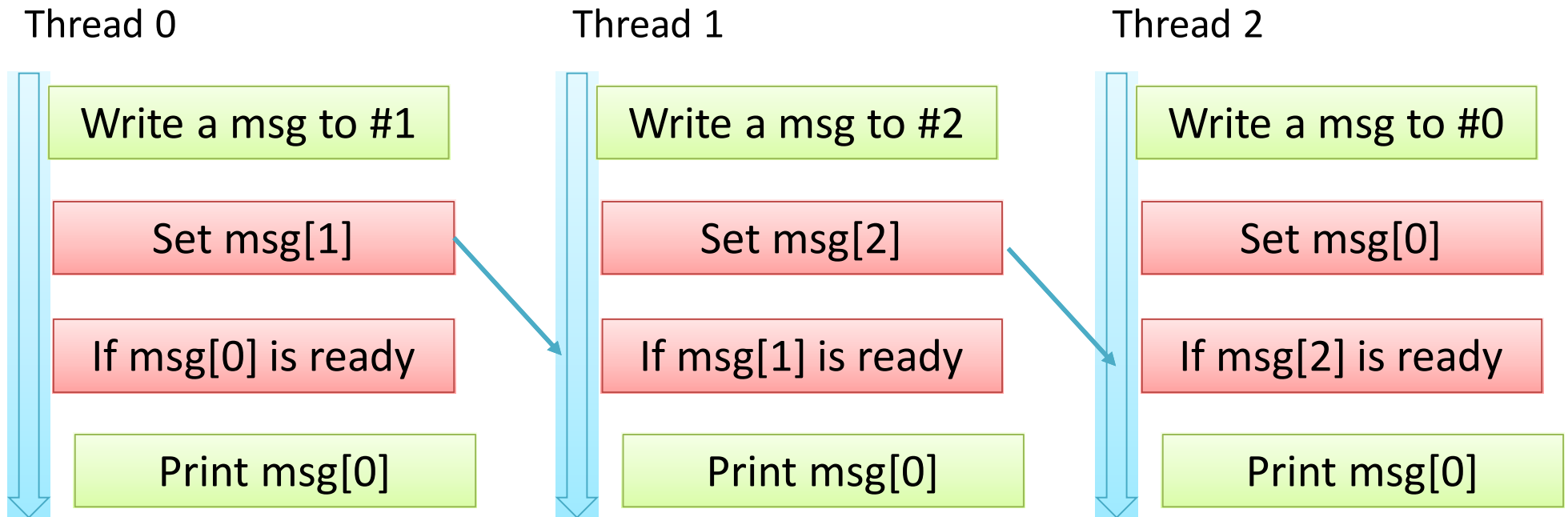
```
int sem_destroy(sem_t*      semaphore_p    /* in/out */);  
int sem_post(sem_t*        semaphore_p    /* in/out */);  
int sem_wait(sem_t*        semaphore_p    /* in/out */);
```

# Producer-Consumer Example



- Thread  $x$  produces a message for Thread  $x+1$ .
  - Last thread produces a message for thread 0.
- Each thread prints a message sent from its source.
- Will there be null messages printed?
  - A consumer thread prints its source message before this message is produced.
  - How to avoid that?

# Flag-based Synchronization with 3 threads



To make sure a message is received/printed, use busy waiting.

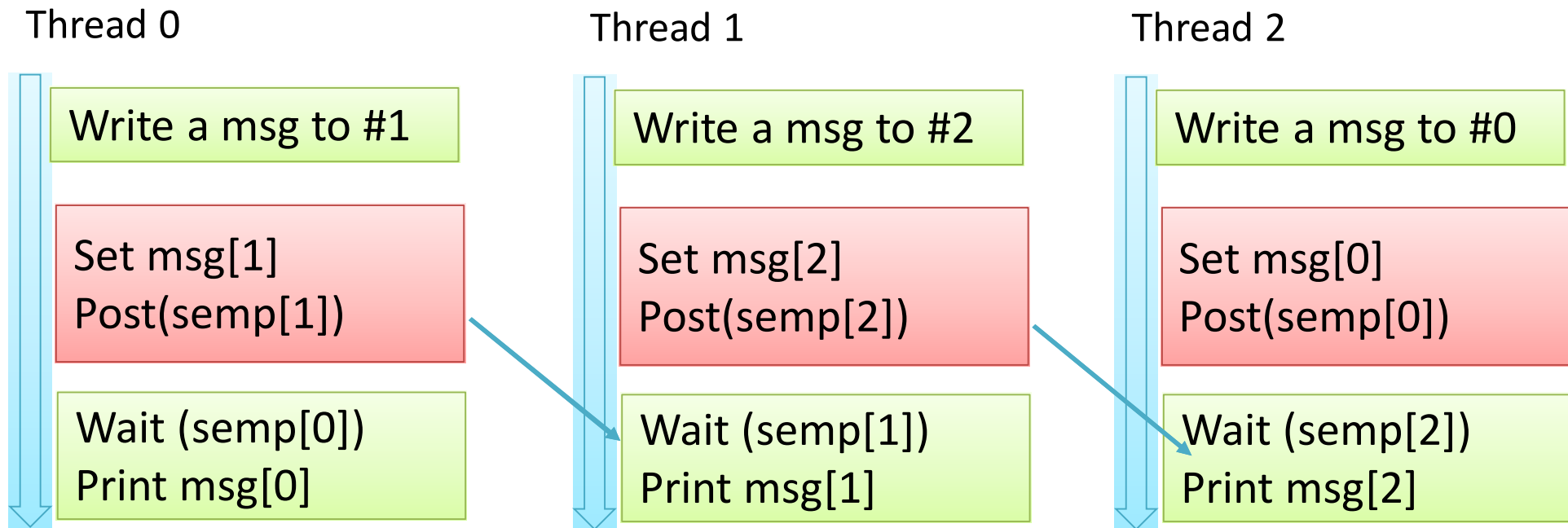


# code

```
1  /* messages has type char**. It's allocated in main. */
2  /* Each entry is set to NULL in main. */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      long source = (my_rank + thread_count - 1) % thread_count;
7      char* my_msg = malloc(MSG_MAX*sizeof(char));
8
9      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
10     messages[dest] = my_msg;
11
12
13     while (messages[my_rank] == NULL);
14     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16
17     return NULL;
18 }
```



# Semaphore Synchronization with 3 threads



Use semaphore to notify destination threads.





# code

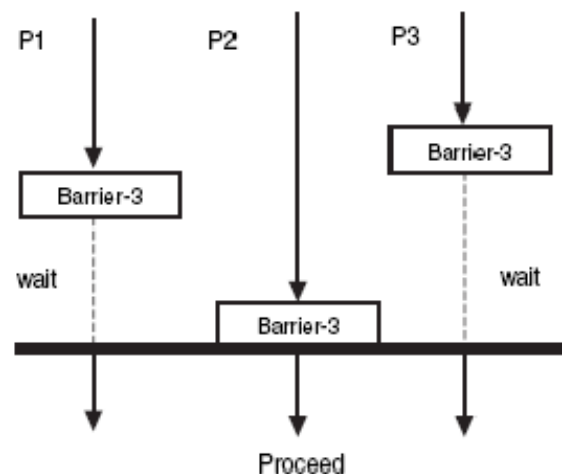
The main thread can initialize all of the **semaphores to 0**

```
1  /* messages is allocated and initialized to NULL in main */
2  /* semaphores is allocated and initialized to 0 (locked) in
   main */
3  void* Send_msg(void* rank) {
4      long my_rank = (long) rank;
5      long dest = (my_rank + 1) % thread_count;
6      char* my_msg = malloc(MSG_MAX*sizeof(char));
7
8      sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
9      messages[dest] = my_msg;
10     sem_post(&semaphores[dest])
        /* ‘‘Unlock’’ the semaphore of dest */
11
12     /* Wait for our semaphore to be unlocked */
13     sem_wait(&semaphores[my_rank]);
14     printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
15
16     return NULL;
17 } /* Send_msg */
```

Using semaphores so that threads can send messages

# Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.
- Availability:
  - No barrier provided by Pthreads library and needs



# Barriers using Busy-waiting and a mutex

- Implementing a barrier using busy-waiting and a mutex
  - a counter indicate how many threads has entered the critical section
  - If all threads hit the critical section, threads can leave

```
/* Shared and initialized by the main thread */
int counter; /* Initialize to 0 */
int thread_count;
pthread_mutex_t barrier_mutex;
. . .

void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&barrier_mutex);
    counter++;
    pthread_mutex_unlock(&barrier_mutex);
    while (counter < thread_count);
    . . .
}
```

Thread arrival  
counter

condition for  
busy waiting

Cons:

waste CPU cycles when threads are in the busy-wait loop

# Barriers using semaphores

- Using two semaphores:
  - *count sem* protects the counter,
  - *barrier sem* block threads that have entered the barrier.

lock at  
beginning

```
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier_sem;  /* Initialize to 0 */
. . .
void* Thread_work(...) {
    . . .
    /* Barrier */
    sem_wait(&count_sem);
    if (counter == thread_count-1) {
        counter = 0;
        sem_post(&count_sem);
        for (j = 0; j < thread_count-1; j++)
            sem_post(&barrier_sem);
    } else {
        counter++;
        sem_post(&count_sem);
        sem_wait(&barrier_sem);
    }
    . . .
}
```

Unlock at beginning





# Pthread Barriers

POSIX threads specifies a synchronization object called a barrier, along with barrier functions. The functions create the barrier, specifying the number of threads that are synchronizing on the barrier.

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

```
int pthread_barrier_init (pthread_barrier_t *barrier,  
                        const pthread_barrierattr_t *attr,  
                        unsigned int count);
```

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```



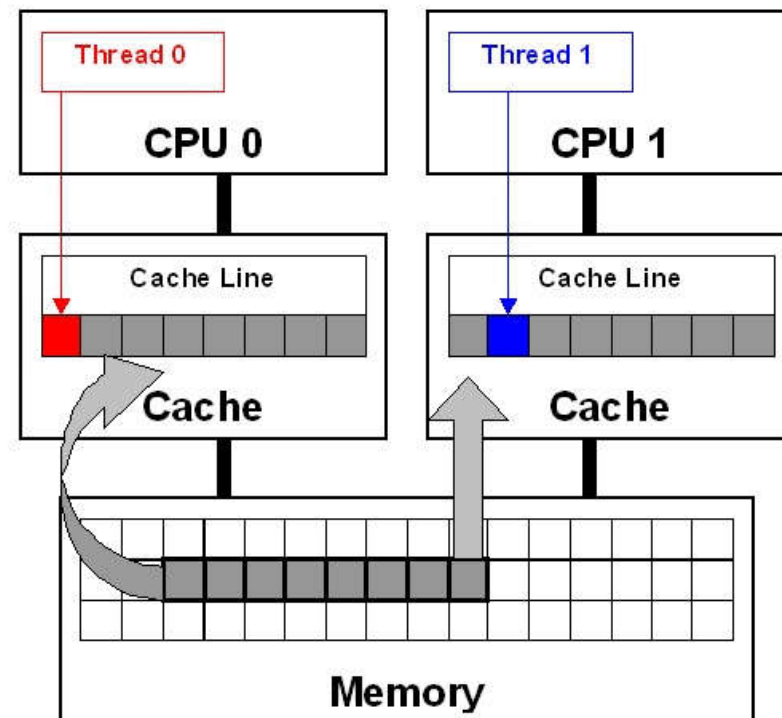
## Issues with Threads: False Sharing, Deadlocks, Thread-safety



# Problem: False Sharing

- Occurs when two or more threads/cores access different data in same cache line, and at least one of them writes.
  - Leads to ping-pong effect

This circumstance is called false sharing because each thread is not actually sharing access to the same variable.



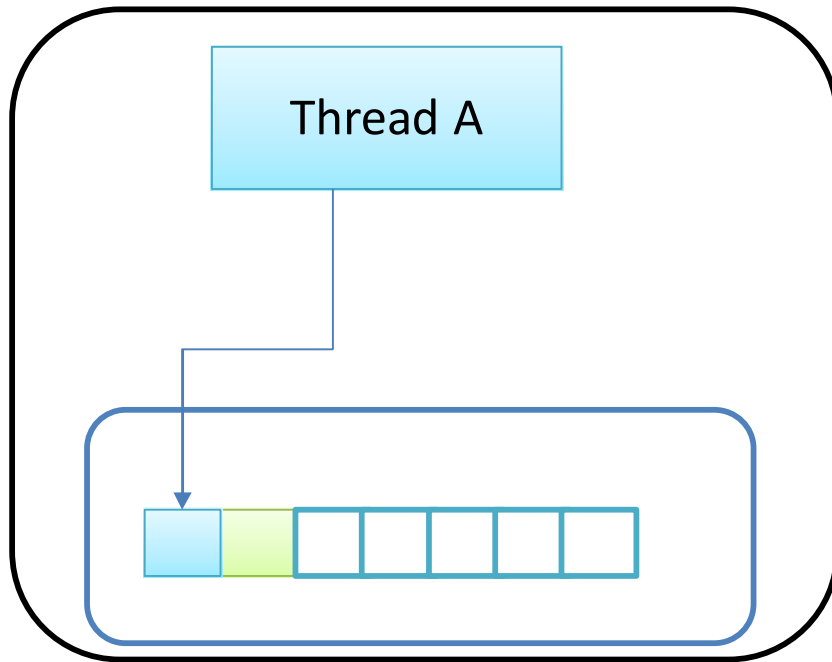


```
y[i] += A[i][j]*x[j];
```

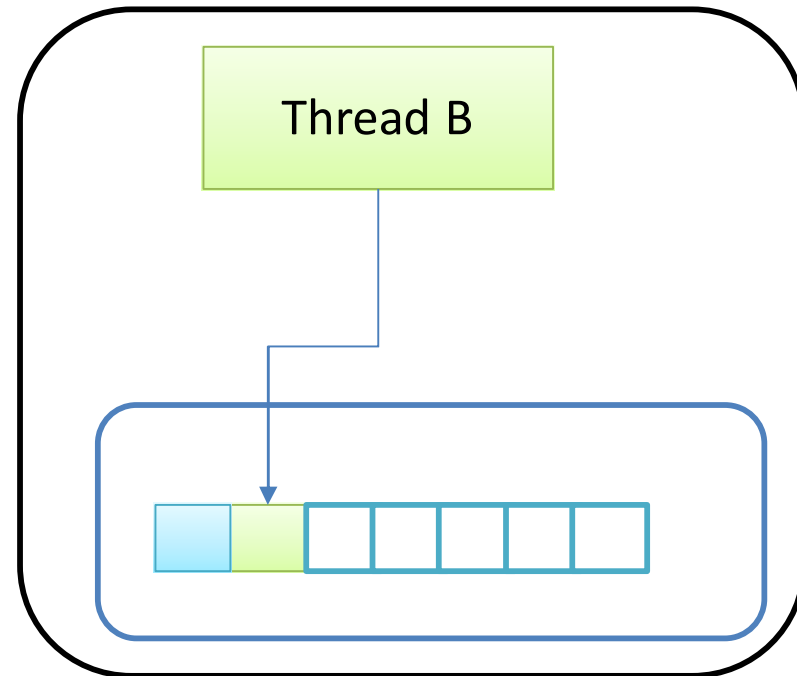
$$\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1} \end{bmatrix}$$
[illegible]



Core 0



Core 1



`y[5996], y[5997], y[5998], y[5999]`

`y[6000], y[6001], y[6002], y[6003]`



# Questions?

---

