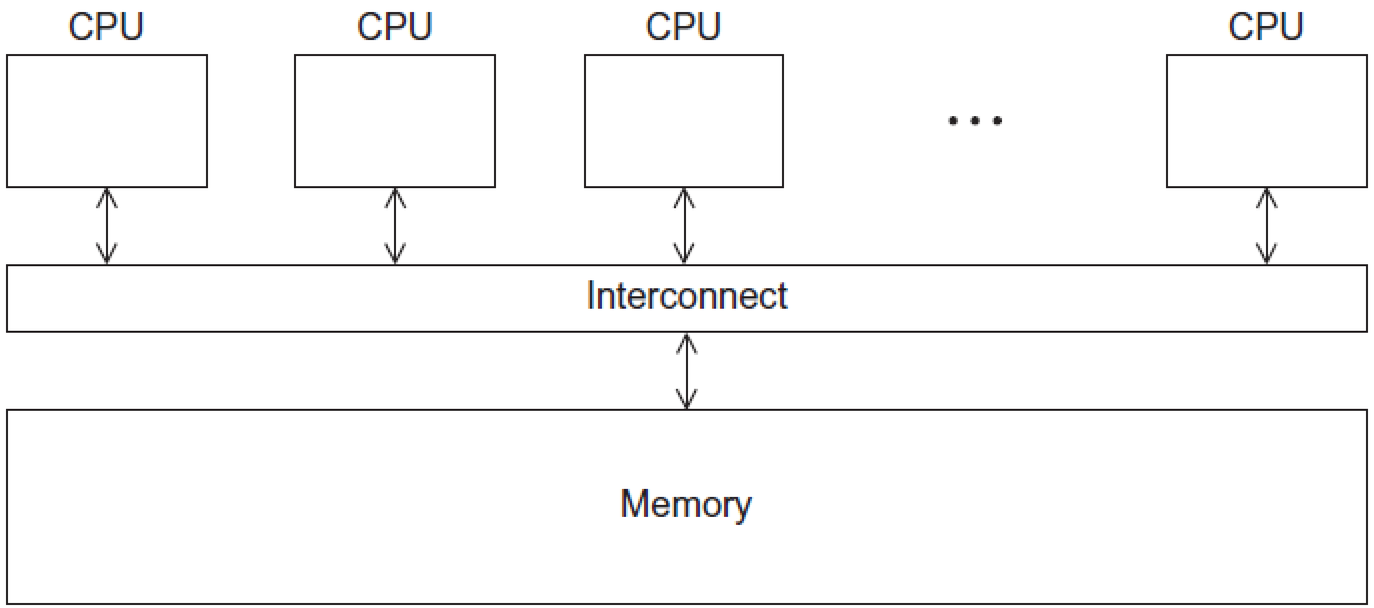
Shared Memory Programming with Pthreads

T. Yang. UCSB CS240A. Spring 2016

Outline

- **Shared memory programming: Overview**
- **POSIX pthreads**
- **Critical section & thread synchronization.**
  - Mutexes.
  - Producer-consumer synchronization and semaphores.
  - Barriers and condition variables.
  - Read-write locks.
- **Thread safety.**

Shared Memory Architecture

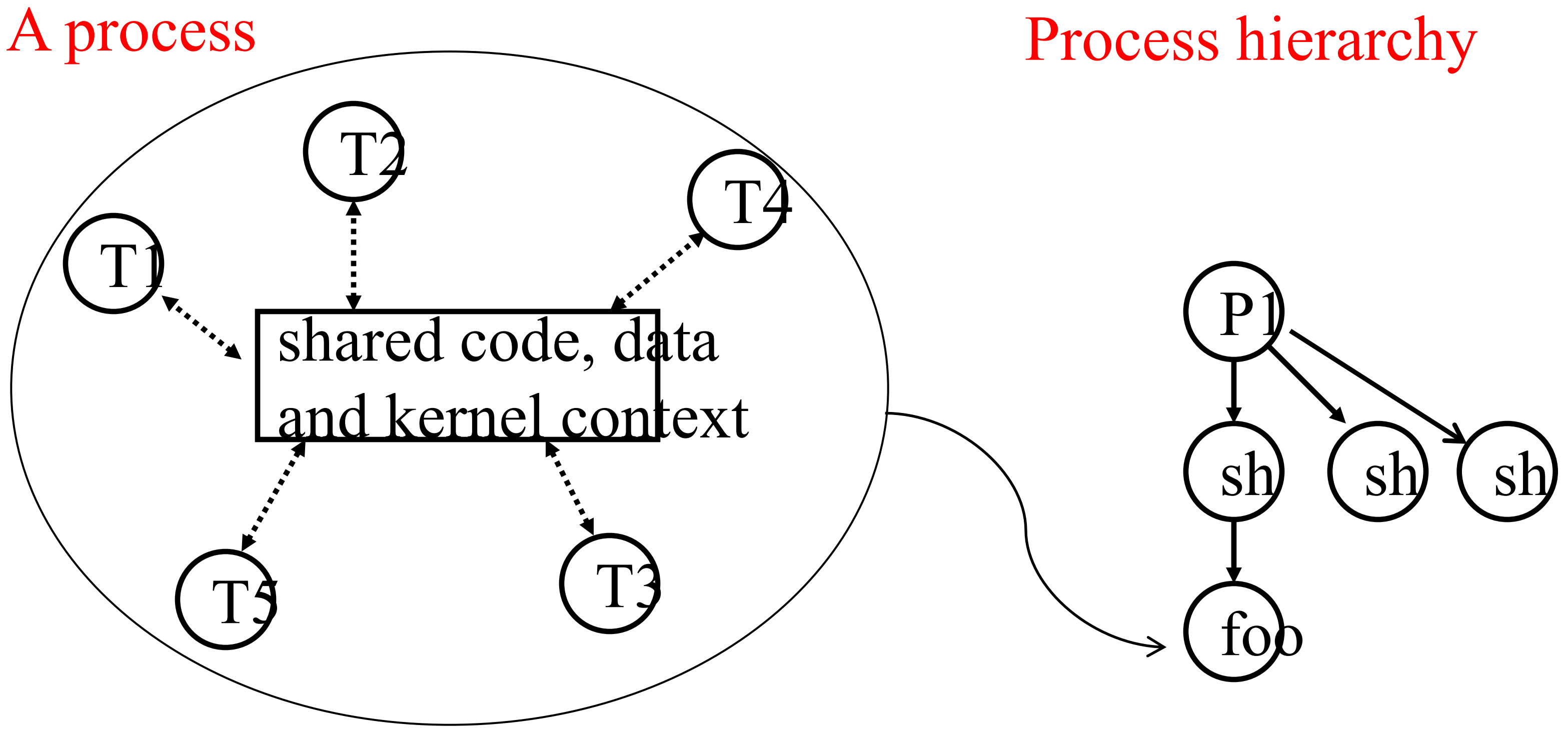


Processes and Threads

- A process is an instance of a running (or suspended) program.
- Threads are analogous to a “light-weight” process.
- In a shared memory program a single process may have multiple threads of control.

Logical View of Threads

- **Threads are created within a process**



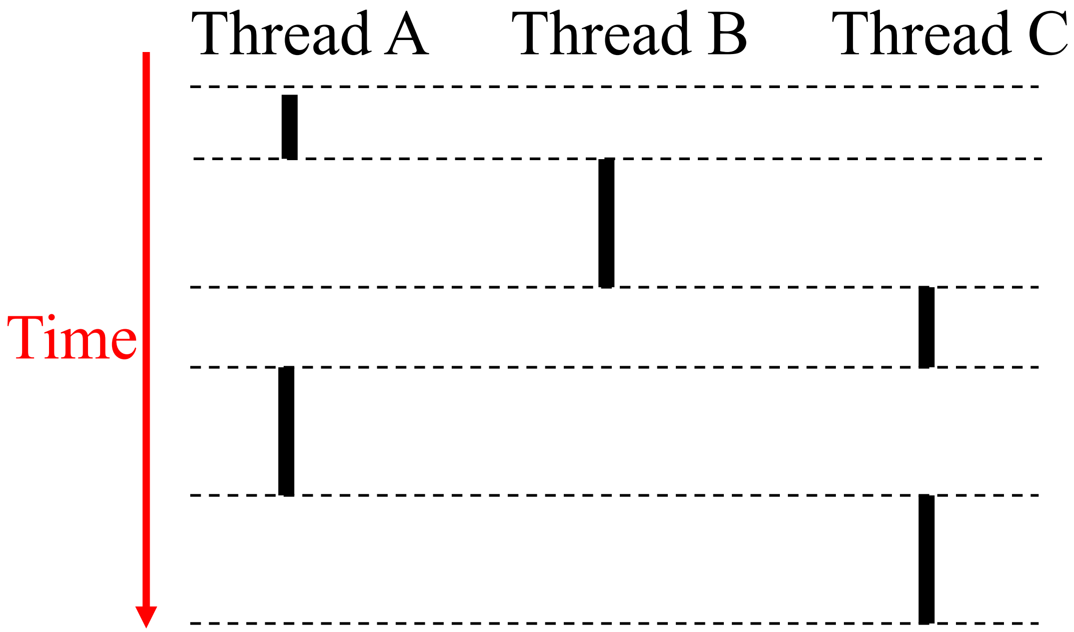
Concurrent Thread Execution

- **Two threads run concurrently if their logical flows overlap in time**
- **Otherwise, they are sequential (we'll see that processes have a similar rule)**
- **Examples:**
  - Concurrent:

  A & B, A&C

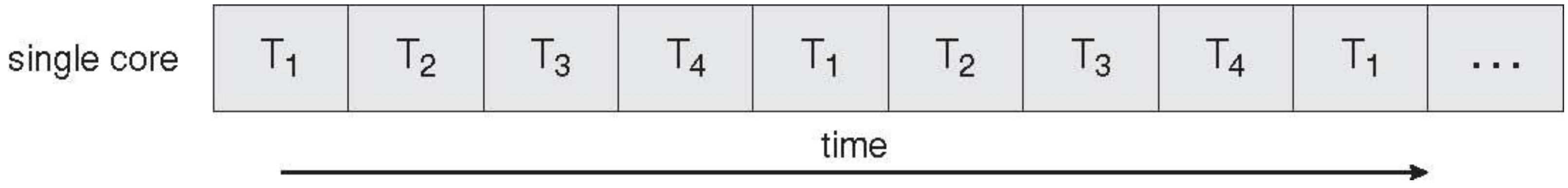
  - Sequential:

  B & C

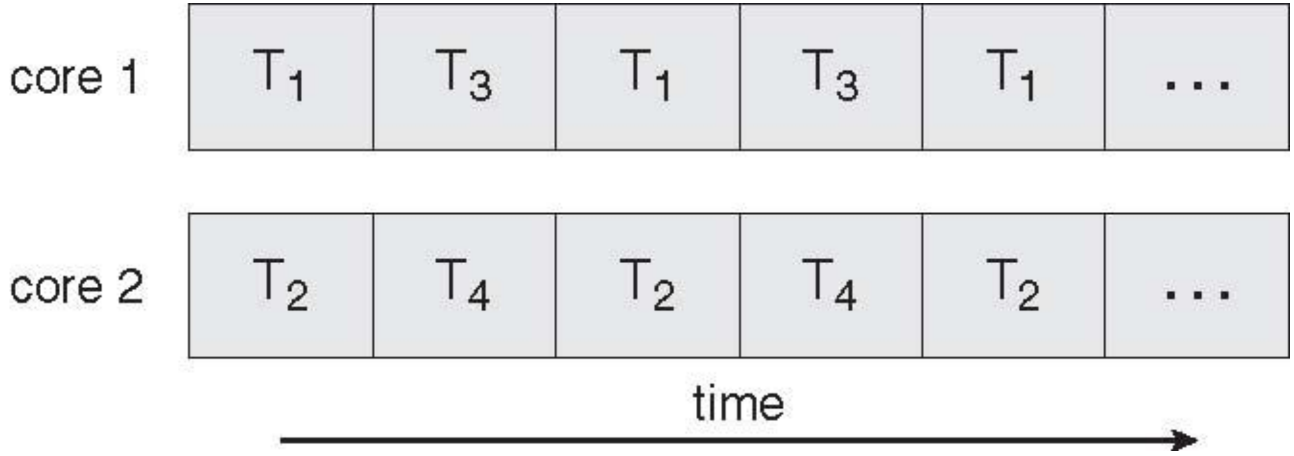


Execution Flow on one-core or multi-core systems

Concurrent execution on a single core system

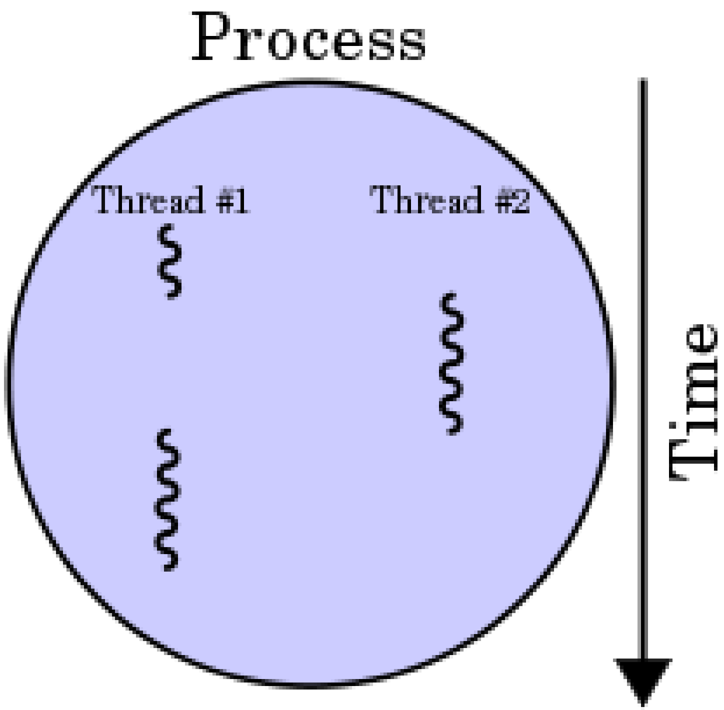


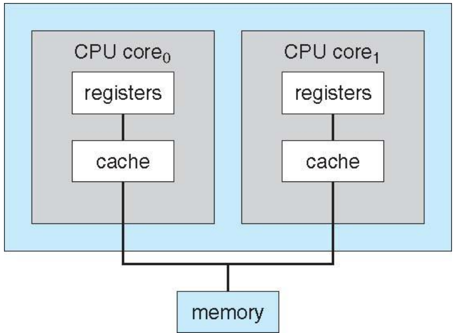
Parallel execution on a multi-core system



Benefits of multi-threading

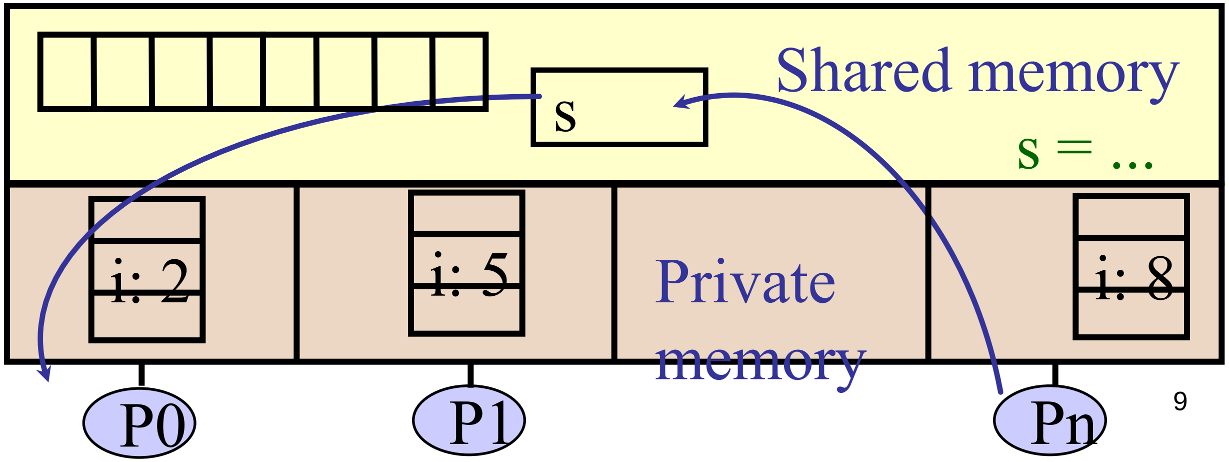
- **Responsiveness**
- **Resource Sharing**
  - Shared memory
- **Economy**
- **Scalability**
  - Explore multi-core CPUs





Thread Programming with Shared Memory

- **Program is a collection of threads of control.**
  - Can be created dynamically
- **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

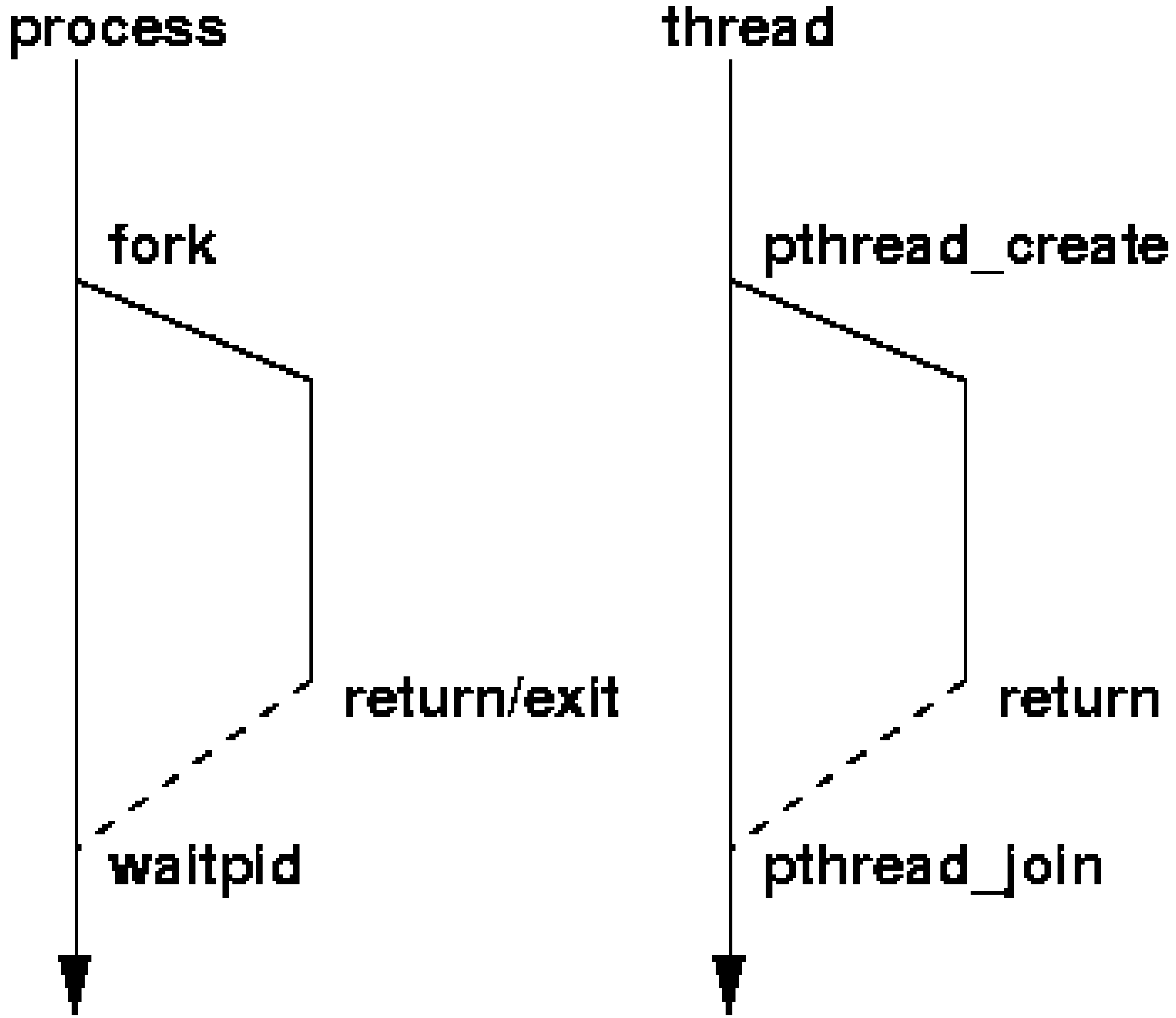


Shared Memory Programming

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
- **Java Threads**
- **TBB: Thread Building Blocks**
  - Intel
- **CILK: Language of the C "ilk"**
  - Lightweight threads embedded into C

Creation of Unix processes vs. Pthreads



C function for starting a thread

pthread.h

One object for each thread.

pthread_t

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

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 <u>before</u> 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

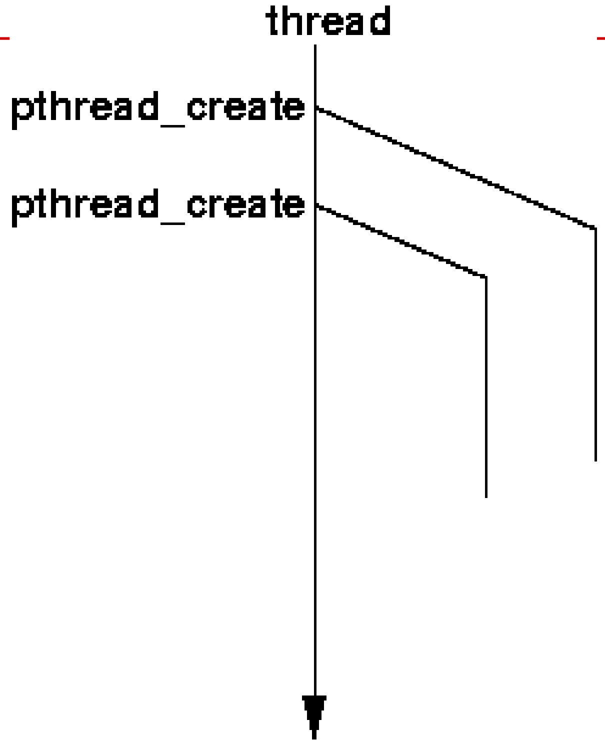
```
pthread_join(pthread_t *thread, void
   **result);
```

- 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.**

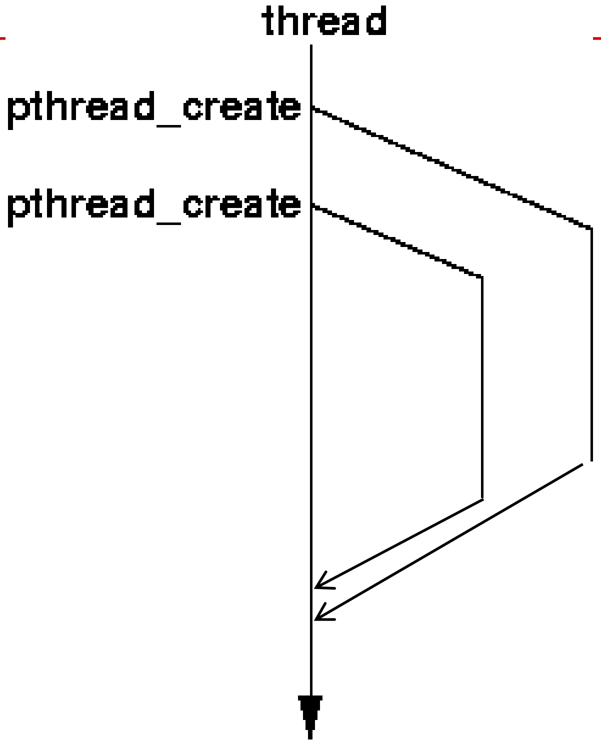
Example of Pthreads

```
#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);
}
```



Example of Pthreads with join

```
#include <pthread.h>
#include <stdio.h>
void *PrintHello(void * id){
   printf("Hello from thread %d\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);
}
```



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 pth_hello pth_hello . c −lpthread
```

link in the Pthreads library

Running a Pthreads program

. / pth_hello

Hello from thread 1

Hello from thread 0

. / pth_hello

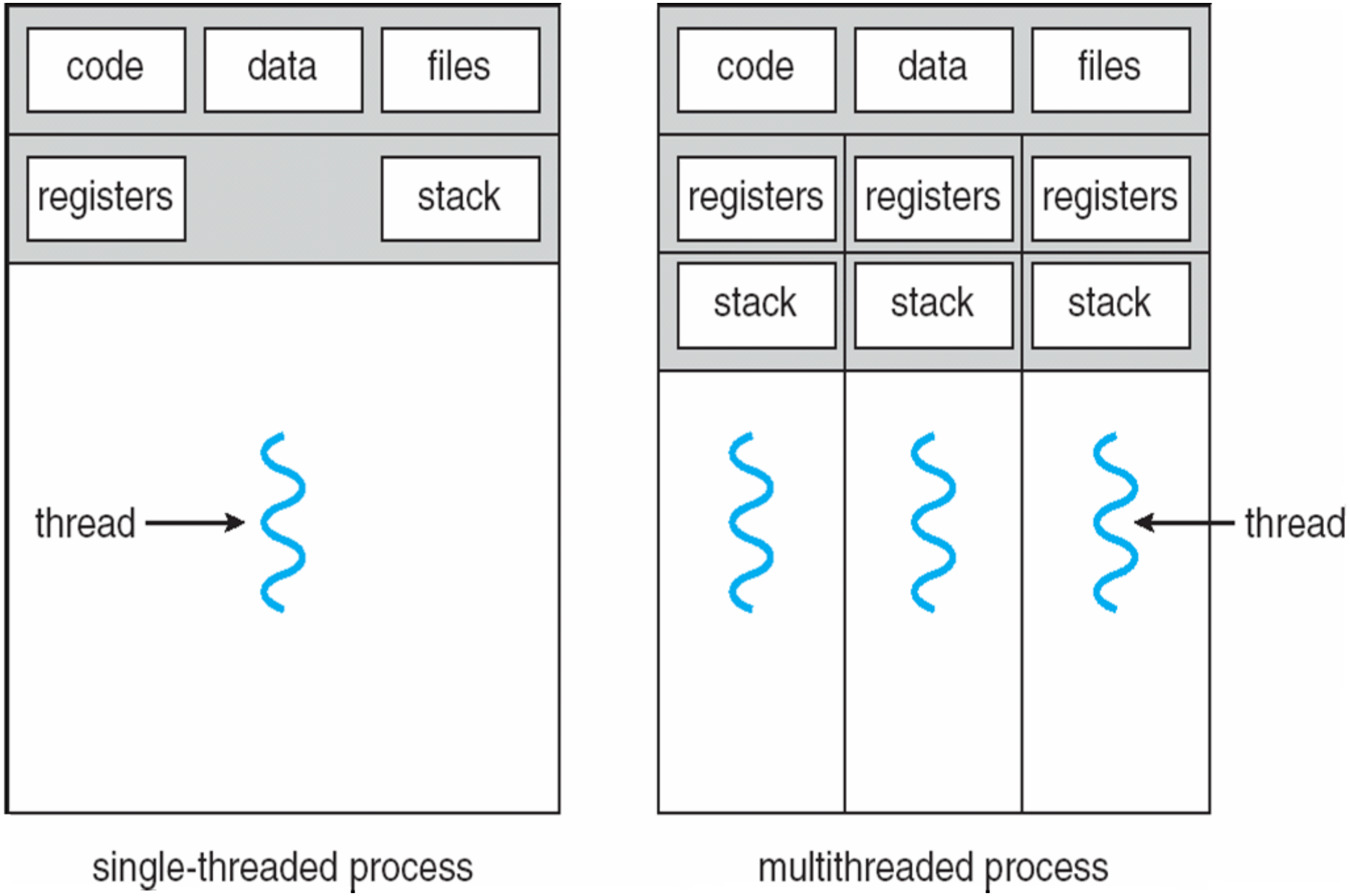
Hello from thread 0

Hello from thread 1

Difference between Single and Multithreaded Processes

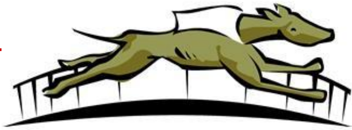
Shared memory access for code/data

Separate control flow -> separate stack/registers



CRITICAL SECTIONS

Data Race Example



```
static int s = 0;
```

Thread 0

```
for i = 0, n/2-1
    s = s + f(A[i])
```

Thread 1

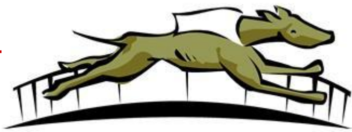
```
for i = n/2, n-1
    s = s + f(A[i])
```

- Also called critical section problem.
- A race condition or data race occurs when:
  - two processors (or two threads) access the same variable, and at least one does a write.
  - The accesses are concurrent (not synchronized) so they could happen simultaneously

Synchronization Solutions

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

Example of Busy Waiting



```
static int s = 0;
static int flag=0
```

Thread 0

```
  int temp, my_rank
  for i = 0, n/2-1
      temp0=f(A[i])
      while flag!=my_rank;
      s = s + temp0
      flag= (flag+1) %2
```

Thread 1

```
  int temp, my_rank
  for i = n/2, n-1
      temp=f(A[i])
      while flag!=my_rank;
      s = s + temp
      flag= (flag+1) %2
```

- A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.
- Weakness: Waste CPU resource. Sometime not safe with compiler optimization.

Mutexes (Locks)

- Code structure

Acquire mutex lock

Critical section

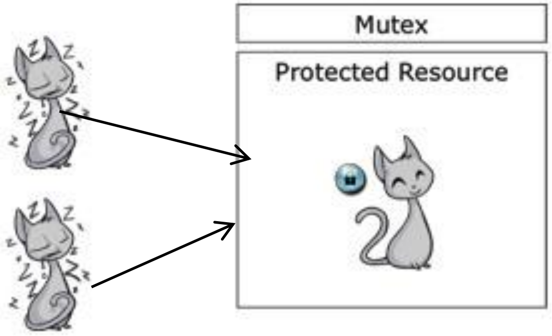
Unlock/Release mutex

- Mutex (mutual exclusion) is a special type of variable used to restrict access to a critical section to a single thread at a time.
- guarantee that one thread "excludes" all other threads while it executes the critical section.
- When A thread waits on a mutex/lock,

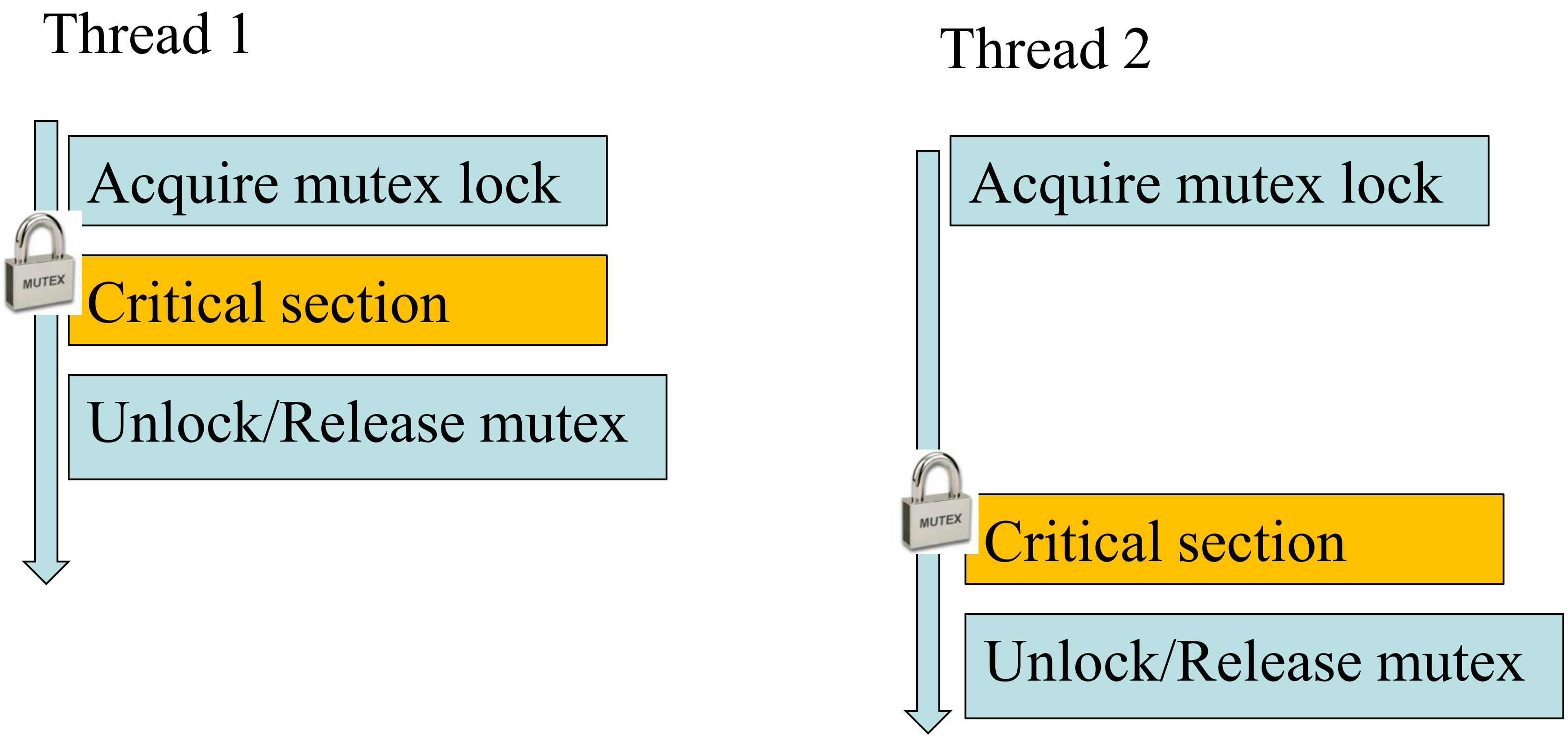
CPU resource can be used by others.

- Only thread that has acquired the lock

can release this lock



Execution example with 2 threads



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

```
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) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;
```

Global sum function that uses a mutex (2)

```
   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Semaphore: Generalization from mutex locks

- Semaphore *S* – integer variable
- 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.

    }

'I think Lassie is trying to tell us something, ma.'

Why Semaphores?

Synchronization	Functionality/weakness
Busy waiting	Spinning for a condition. Waste resource. Not safe
Mutex lock	Support code with simple mutual exclusion
Semaphore	Handle more complex signal-based synchronization

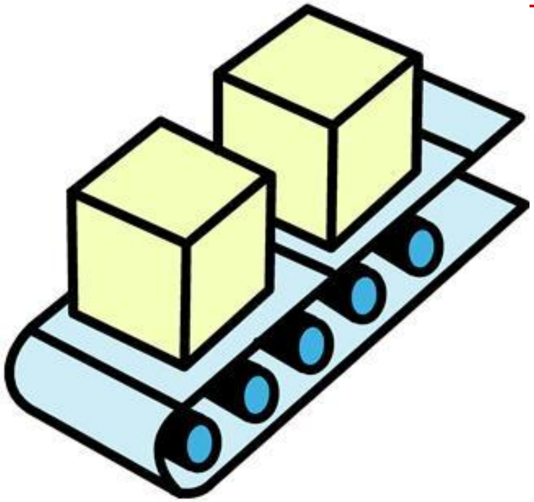
'I think Lassie is trying to tell us something, ma.'

- **Examples of complex synchronization**
  - Allow a resource to be shared among multiple threads.
    - Mutex: no more than 1 thread for one protected region.
  - Allow a thread waiting for a condition after a signal
    - E.g. Control the access order of threads entering the critical section.
    - For mutexes, the order is left to chance and the system.

Syntax of Pthread semaphore functions

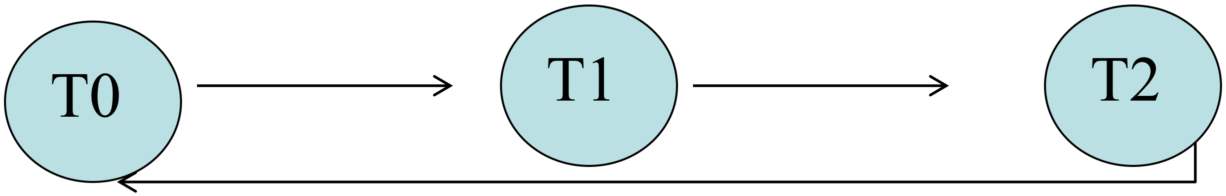
Semaphores are not part of Pthreads; you need to add this.

```
#include <semaphore.h>

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


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 */);
```



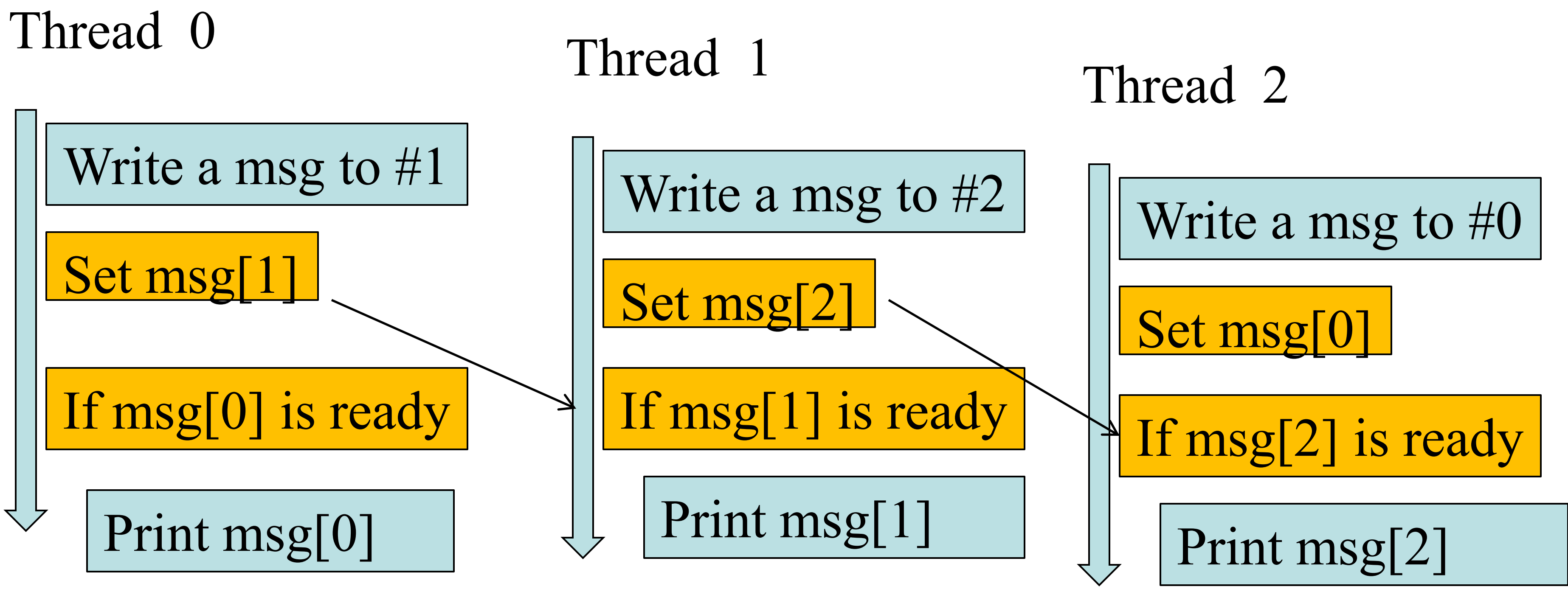
Producer-consumer Synchronization and Semaphores

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.

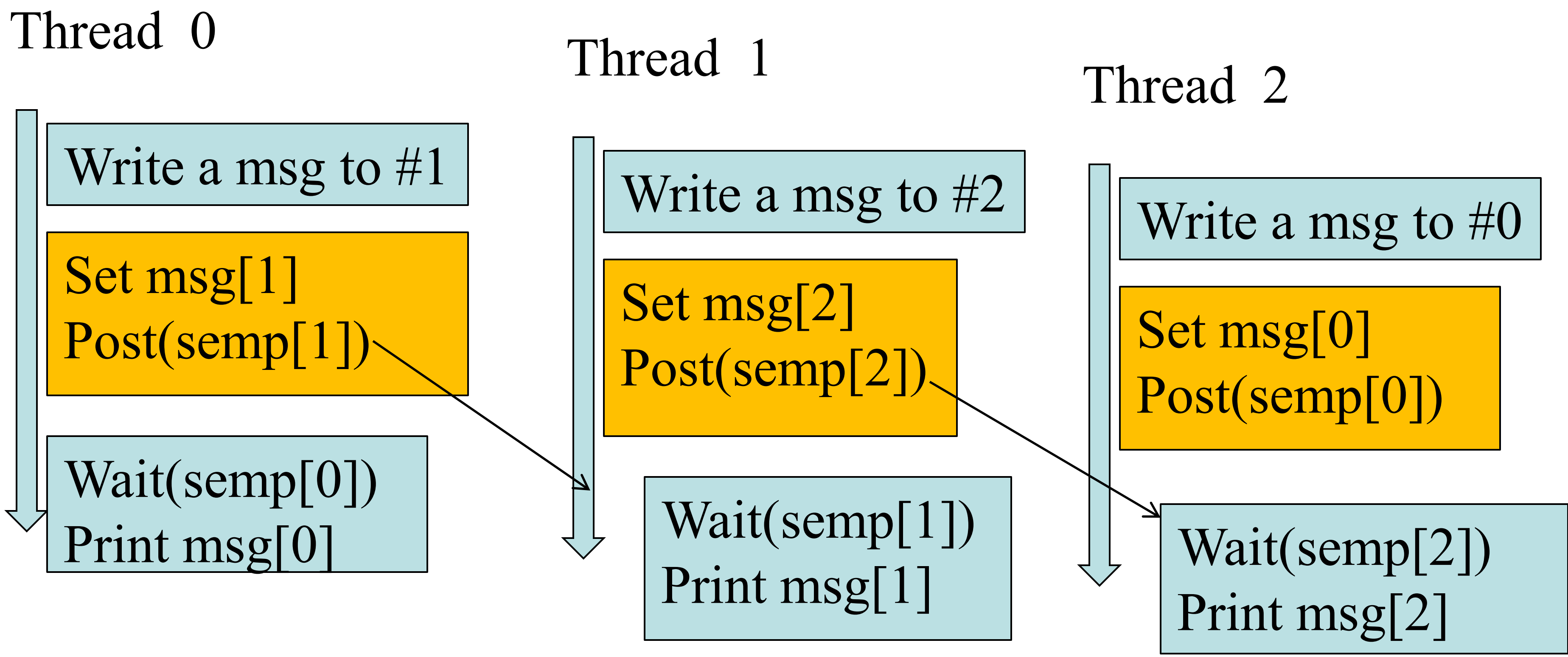
First attempt at sending messages using pthreads

```
/* messages has type char**. It's allocated in main. */
/* Each entry is set to NULL in main.                 */
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   long source = (my_rank + thread_count - 1) % thread_count;
   char* my_msg = malloc(MSG_MAX*

   sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
   messages[dest] = my_msg;

   if (messages[my_rank] != NULL)
      printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
   else
      printf("Thread %ld > No message from %ld\n", my_rank, source);

   return NULL;
}  /* Send_msg */
```

Produce a message for a destination thread

Consume a message

Semaphore Synchronization with 3 threads



Message sending with semaphores

```
sprintf(my_msg, "Hello to %ld from %ld", dest, my_rank);
messages[dest] = my_msg;

sem_post(&semaphores[dest]);
                                        /* signal the dest thread*/
sem_wait(&semaphores[my_rank]);
        /* Wait until the source message is created */

printf("Thread %ld > %s\n", my_rank,
   messages[my_rank]);
```

READERS-WRITERS PROBLEM

Synchronization Example for Readers-Writers Problem

- **A data set is shared among a number of concurrent threads.**
  - Readers – only read the data set; they do **not** perform any updates
  - Writers – can both read and write
- **Requirement:**
  - allow multiple readers to read at the same time.
  - Only one writer can access the shared data at the same time.
- **Reader/writer access permission table:**

	Reader	Writer
Reader	OK	No
Writer	NO	No

Readers-Writers (First try with 1 mutex lock)

- **writer**

```
do {
        mutex_lock(w);
        //    writing is performed
        mutex_unlock(w);
} while (TRUE);
```

- **Reader**

```
     do {
        mutex_lock(w);
        //    reading is performed
        mutex_unlock(w);
} while (TRUE);
```

	Reader	Writer
Reader	?	?
Writer	?	?

Readers-Writers (First try with 1 mutex lock)

- **writer**

```
do {
        mutex_lock(w);
        //    writing is performed
        mutex_unlock(w);
} while (TRUE);
```

- **Reader**

```
do {
        mutex_lock(w);
        //    reading is performed
        mutex_unlock(w);
} while (TRUE);
```

	Reader	Writer
Reader	no	no
Writer	no	no

2nd try using a lock + readcount

- **writer**

```
do {
        mutex_lock(w);// Use writer mutex lock
        //    writing is performed
        mutex_unlock(w);
} while (TRUE);
```

- **Reader**

```
do {
        readcount++; // add a reader counter.
        if(readcount==1) mutex_lock(w);
        //    reading is performed
        readcount--;
          if(readcount==0)  mutex_unlock(w);
} while (TRUE);
```

Readers-Writers Problem with semaphone

- **Shared Data**
  - Data set
  - Lock mutex (to protect readcount)
  - Semaphore wrt initialized to 1 (to synchronize between readers/writers)
  - Integer readcount initialized to 0

Readers-Writers Problem

- **A writer**

```
do {
    sem_wait(wrt) ;  //semaphore wrt

   // writing is performed

    sem_post(wrt) ;  //
} while (TRUE);
```

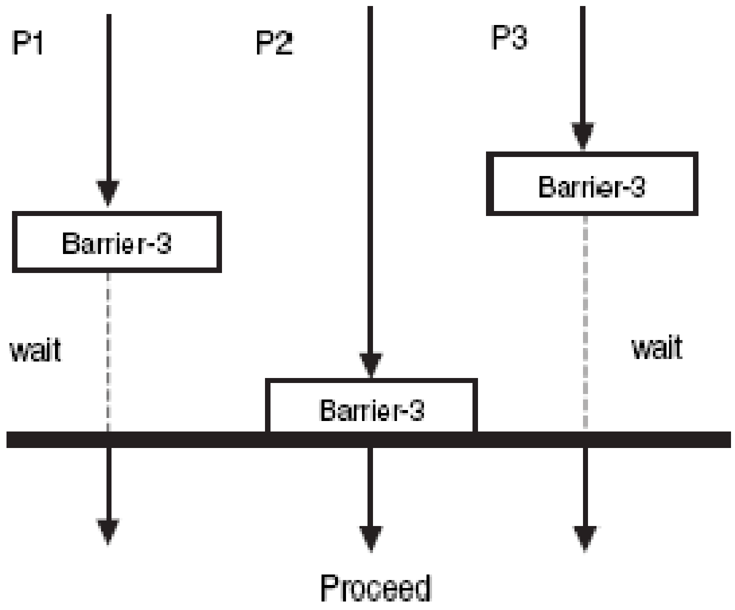
Readers-Writers Problem (Cont.)

- **Reader**

```
do {
        mutex_lock(mutex);
        readcount ++ ;
        if (readcount == 1)
                sem_wait(wrt);  //check if anybody is writing
        mutex_unlock(mutex)

        // reading is performed

        mutex_lock(mutex);
        readcount  - - ;
        if (readcount  == 0)
                sem_post(wrt) ;  //writing is allowed now
        nlock(mutex) ;
    } while (TRUE);
```

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 a custom implementation
  - Barrier is implicit in OpenMP and available in MPI.



Condition Variables

- Why?
- More programming primitives to simplify code for synchronization of threads

Synchronization	Functionality
Busy waiting	Spinning for a condition. Waste resource. Not safe
Mutex lock	Support code with simple mutual exclusion
Semaphore	Signal-based synchronization. Allow sharing (not wait unless semaphore=0)
Barrier	Rendezvous-based synchronization
Condition variables	More complex synchronization: Let threads wait until a user-defined condition becomes true

Synchronization Primitive: Condition Variables

- Used together with a lock
- One can specify more general waiting condition compared to semaphores.
- A thread is blocked when condition is no true:
  - placed in a waiting queue, yielding CPU resource to somebody else.
  - Wake up until receiving a signal

Pthread synchronization: Condition variables

```
int status;     pthread_condition_t cond;

const pthread_condattr_t attr;

pthread_mutex mutex;

status = pthread_cond_init(&cond,&attr);

status = pthread_cond_destroy(&cond);

status = pthread_cond_wait(&cond,&mutex);
```

-wait in a queue until somebody wakes up. Then the mutex is reacquired.

```
status = pthread_cond_signal(&cond);
```

- wake up one waiting thread.

```
status = pthread_cond_broadcast(&cond);
```

- wake up all waiting threads in that condition

How to Use Condition Variables: Typical Flow

- Thread 1: //try to get into critical section and wait for the condition

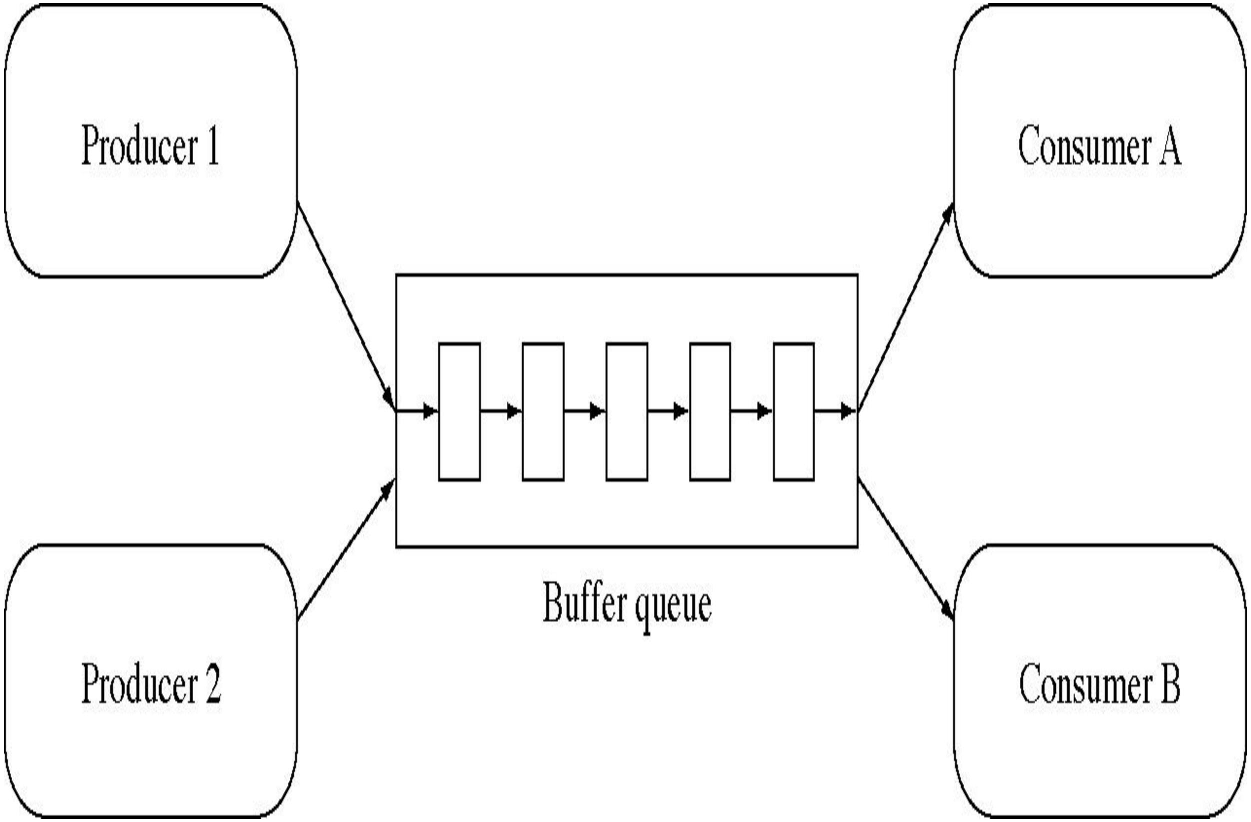
```
Mutex_lock(mutex);
  While (condition is not satisfied)
        Cond_Wait(mutex, cond);
    Critical Section;
Mutex_unlock(mutex)
```

- Thread 2: // Try to create the condition.

```
    Mutex_lock(mutex);
    When condition can satisfy,  Signal(cond);
    Mutex_unlock(mutex);
```

Condition variables for in producer-consumer problem with unbounded buffer

Producer deposits data in a buffer for others to consume



First version for consumer-producer problem with unbounded buffer

- int avail=0;   // # of data items available for consumption
- Consumer thread:

```
while (avail <=0); //wait
 Consume next item;  avail = avail-1;
```

- *Producer thread:*

```
Produce next item;  avail = avail+1;
//notify an item is available
```

Condition Variables for consumer-producer problem with unbounded buffer

- int avail=0; // # of data items available for consumption
- Pthread mutex m and condition cond;
- Consumer thread:

```
multex_lock(&m)
 while (avail <=0) Cond_Wait(&cond, &m);
  Consume next item;  avail = avail-1;
 mutex_unlock(&mutex)
```

- *Producer thread:*

```
mutex_lock(&m);
Produce next item;  availl = avail+1;
Cond_signal(&cond); //notify an item is available
mutex_unlock(&m);
```

When to use condition broadcast?

- When waking up one thread to run is not sufficient.
- Example: concurrent malloc()/free() for allocation and deallocation of objects with non-uniform sizes.

Running trace of malloc()/free()

- Initially 10 bytes are free.
- m() stands for malloc(). f() for free()

Thread 1:	Thread 2:	Thread 3:
m(10) – succ	m(5) – wait	m(5) – wait
f(10) –broadcast		
	Resume m(5)-succ	
		Resume m(5)-succ
m(7) – wait		
		m(3) –wait
	f(5) –broadcast	
Resume m(7)-wait		Resume m(3)-succ

Time

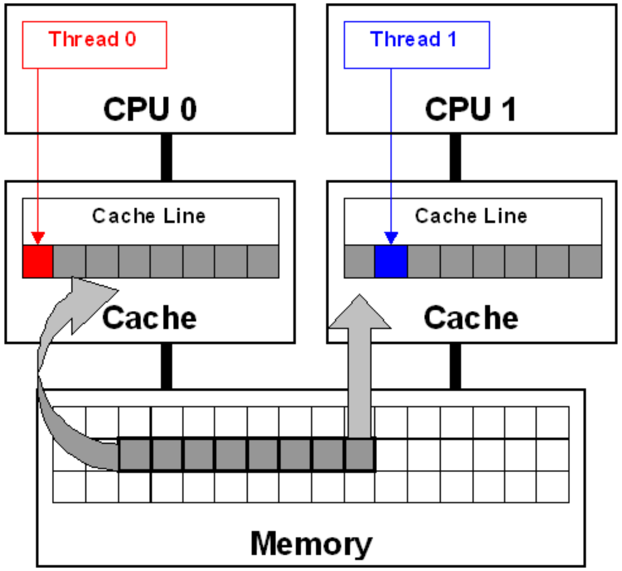
Issues with Threads: False Sharing, Deadlocks, Thread-safety

Problem: False Sharing

- **Occurs when two or more processors/cores access different data in same cache line, and at least one of them writes.**
  - Leads to ping-pong effect.
- **Let's assume we parallelize code with p=2:**

```
for( i=0; i<n; i++ )
    a[i] = b[i];
```

  - Each array element takes 8 bytes
  - Cache line has 64 bytes (8 numbers)

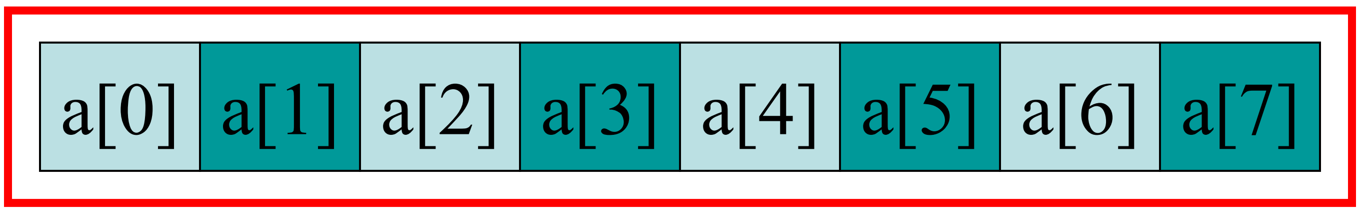


False Sharing: Example (2 of 3)

Execute this program in two processors

```
for( i=0; i<n; i++ )
      a[i] = b[i];
```

cache line



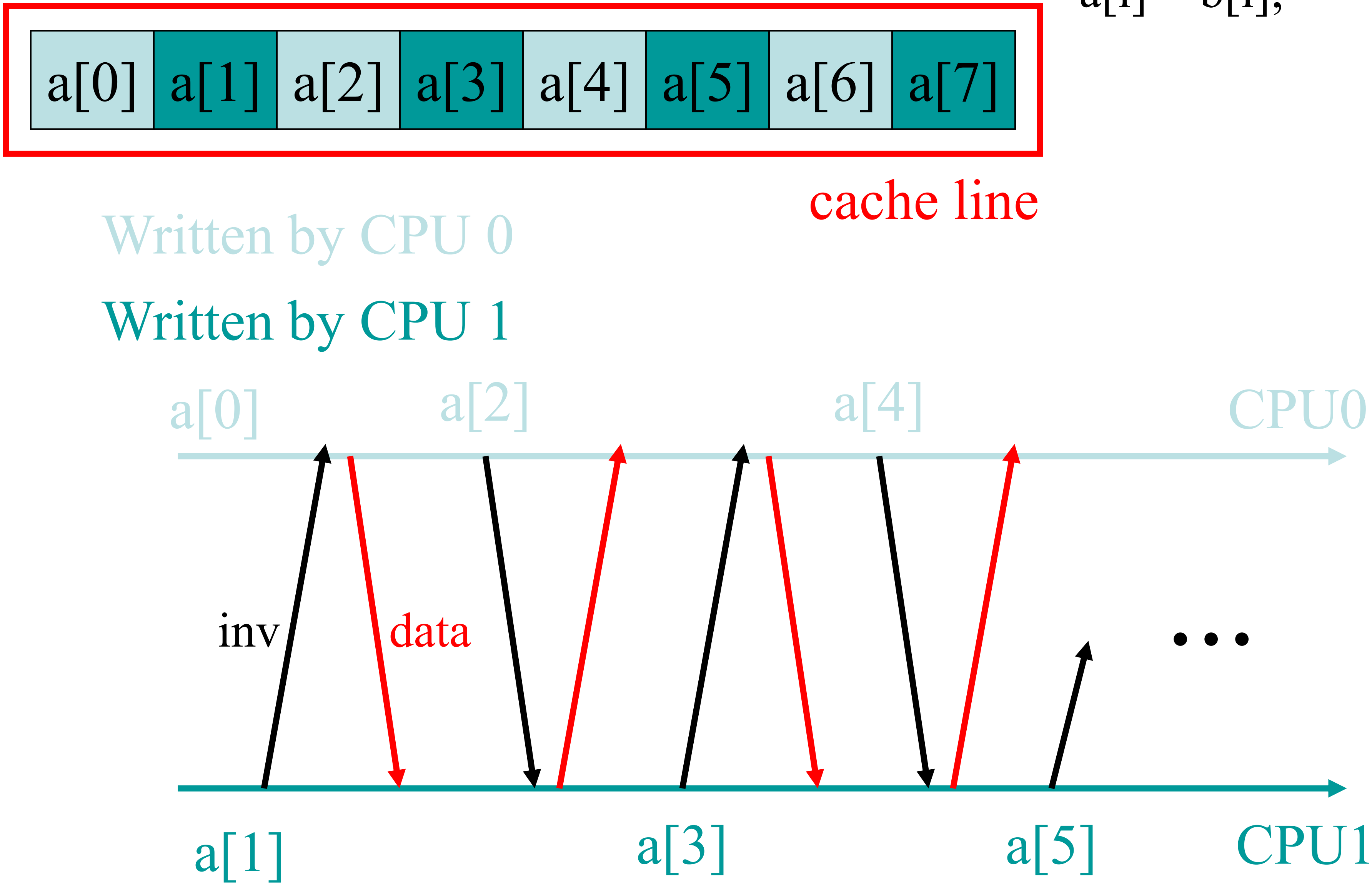
Written by CPU 0

Written by CPU 1

False Sharing: Example

Two CPUs execute:

```
for( i=0; i<n; i++ )
    a[i] = b[i];
```



Matrix-Vector Multiplication with Pthreads

Parallel programming book by Pacheco book P.159-162

Sequential code

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1*1+2*2+3*3 \\ 4*1+5*2+6*3 \\ 7*1+8*2+9*3 \end{pmatrix} = \begin{pmatrix} 14 \\ 32 \\ 50 \end{pmatrix}$$

```
/* 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[i] += A[i][j]* x[j];
}
```

$$\begin{array}{|c|c|c|c|} \hline a_{00} & a_{01} & \cdots & a_{0,n-1} \\ \hline a_{10} & a_{11} & \cdots & a_{1,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\ \hline \vdots & \vdots & & \vdots \\ \hline a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\ \hline \end{array} \quad \begin{array}{|c|} \hline x_0 \\ x_1 \\ \vdots \\ x_{n-1} \\ \hline \end{array} = \begin{array}{|c|} \hline y_0 \\ \hline y_1 \\ \hline \vdots \\ \hline y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \hline \vdots \\ \hline y_{m-1} \\ \hline \end{array}$$

Block Mapping for Matrix-Vector Multiplication

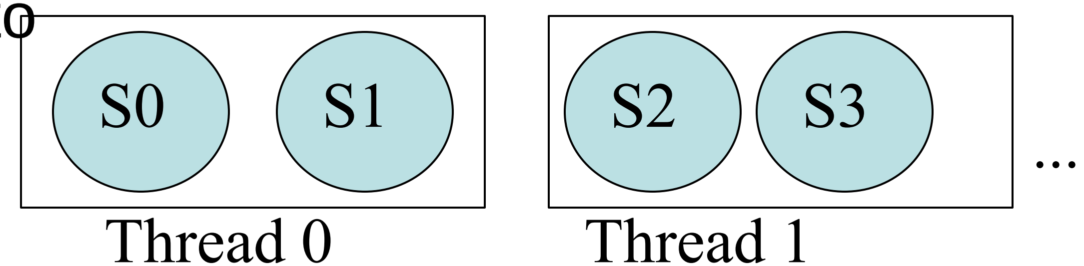
- Task partitioning

  For (i=0; i<m; i=i+1)

```
Task Si for Row i
y[i]=0;
For (j=0; j<n; j=j+1)
      y[i]=y[i] +a[i][j]*x[j]
```

Task graph

Mapping to threads



Using 3 Pthreads for 6 Rows: 2 row per thread

Thread	Components of y	
0	y[0], y[1]	→ S0, S1
1	y[2], y[3]	→ S2, S3
2	y[4], y[5]	→ S4,S5

Code for S0

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

Code for Si

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

Pthread code for thread with ID rank

i-th thread calls Pth_mat_vect(&i)
m is # of rows in this matrix A.
n is # of columns in this matrix A.
local_m is # of rows handled by this thread.

```
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 */
```

Task Si

Impact of false sharing on performance of matrix-vector multiplication

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

(times are in seconds)

Why is performance of 8x8,000,000 matrix bad?

How to fix that?

Deadlock and Starvation

- **Deadlock – ** two or more threads are waiting indefinitely for an event that can be only caused by one of these waiting threads
- **Starvation – ** indefinite blocking (in a waiting queue forever).
  - Let S and Q be two mutex locks:

P_0	P_1
Lock(S);	Lock(Q);
Lock(Q);	Lock(S);
.	.
.	.
.	.
Unlock(Q);	Unlock(S);
Unlock(S);	Unlock(Q);

Deadlock Avoidance

- **Order the locks and always acquire the locks in that order.**
- **Eliminate circular waiting**
  - :

P_0	P_1
Lock(S);	Lock(S);
Lock(Q);	Lock(Q);
.	.
.	.
.	.
Unlock(Q);	Unlock(Q);
Unlock(S);	Unlock(S);

Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.
- When you program your own functions, you know if they are safe to be called by multiple threads or not.
- You may forget to check if system library functions used are thread-safe.
  - Unsafe function: strtok()from C string.h library
  - Other example.
    - The random number generator random in stdlib.h.
    - The time conversion function localtime in time.h.

Concluding Remarks

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
  - However, a thread is often lighter-weight than a full-fledged process.
- When multiple threads access a shared resource without controling, it may result in an error: we have a race condition.
  - A critical section is a block of code that updates a shared resource that can only be updated by one thread at a time
  - Mutex, semaphore, condition variables
- Issues: false sharing, deadlock, thread safety