Shared Memory Programming with Pthreads

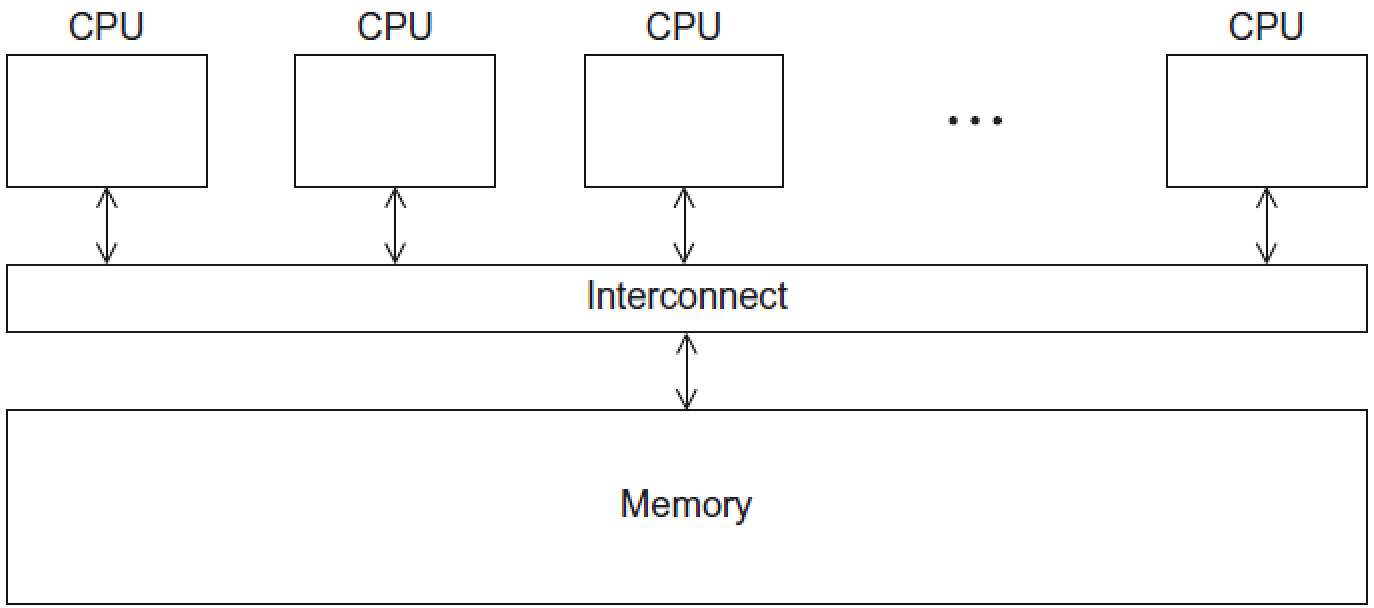
Pacheco. Chapter 4

T. Yang. UCSB CS140. Spring 2014

Outline

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

Shared Memory Architecture

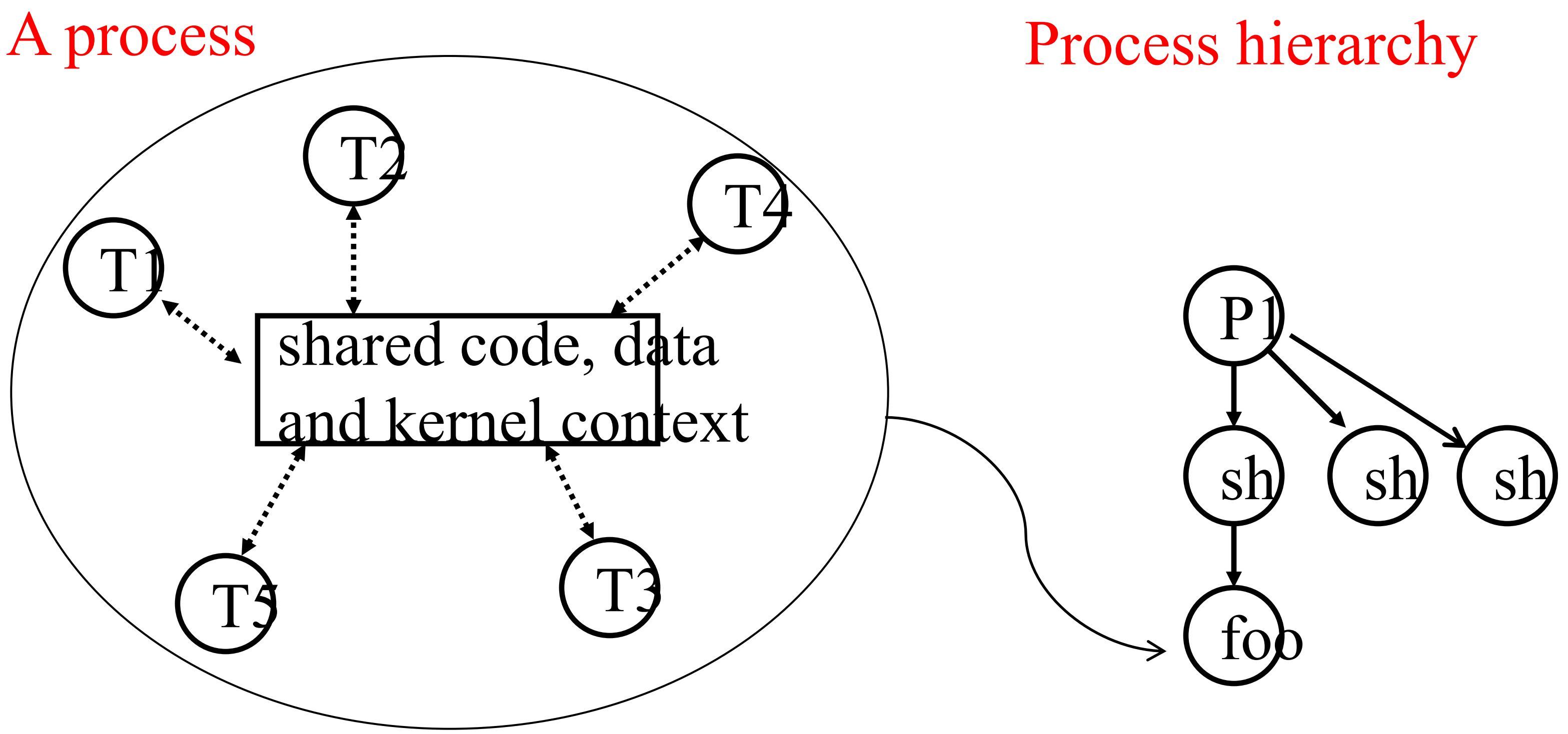


Processes and Threads

- **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**

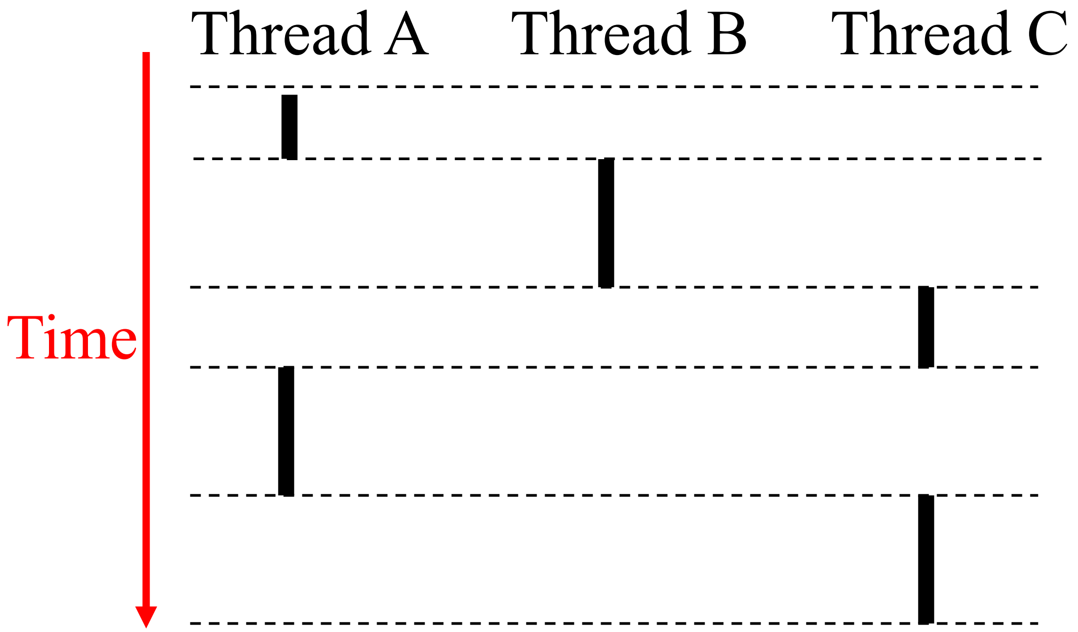


Concurrent Thread Execution

- **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:

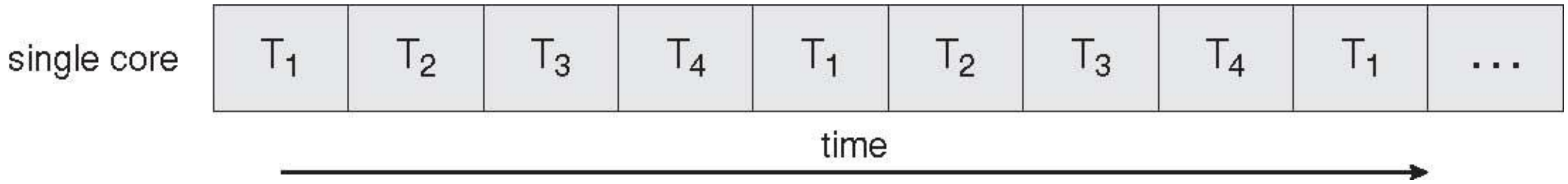
    A & B, A&C
  - Sequential:

    B & C

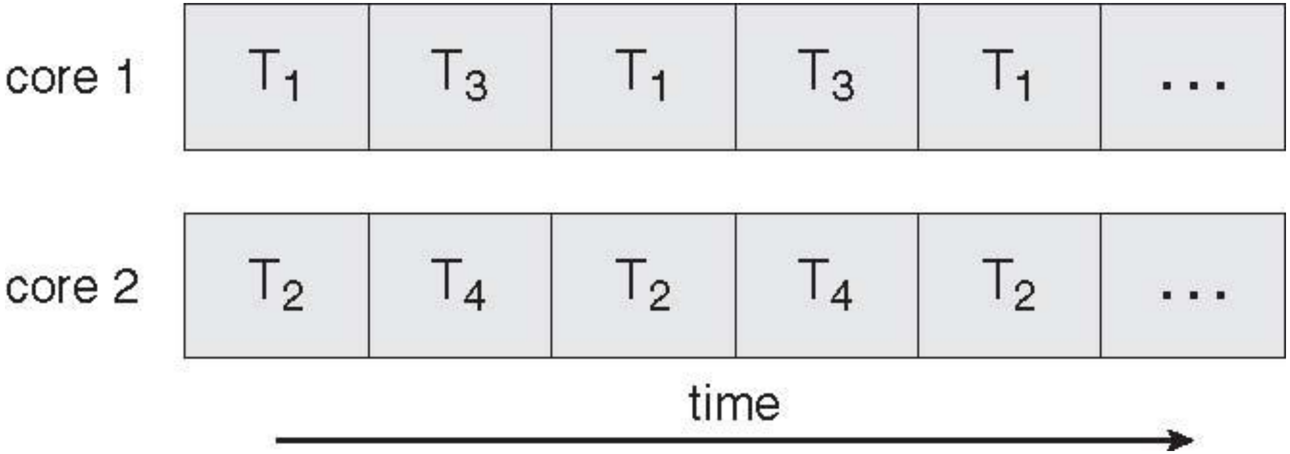


Execution Flow on one-core or multi-core systems

Concurrent execution on a single core system

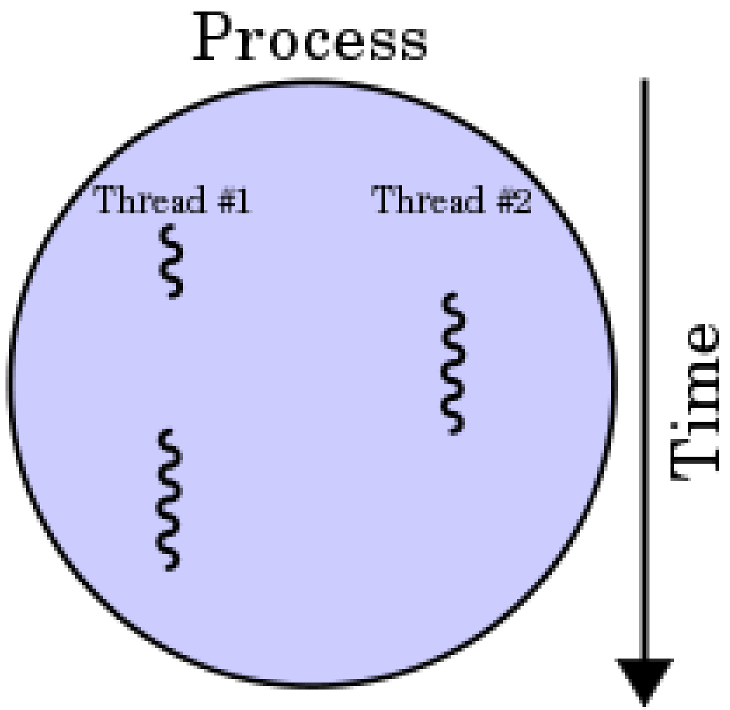


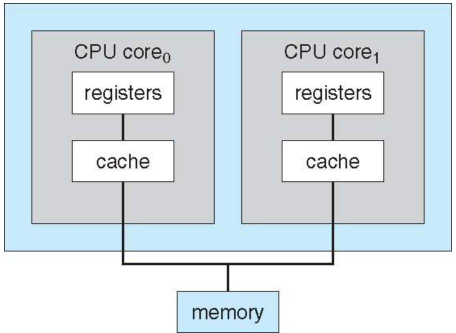
Parallel execution on a multi-core system



Benefits of multi-threading

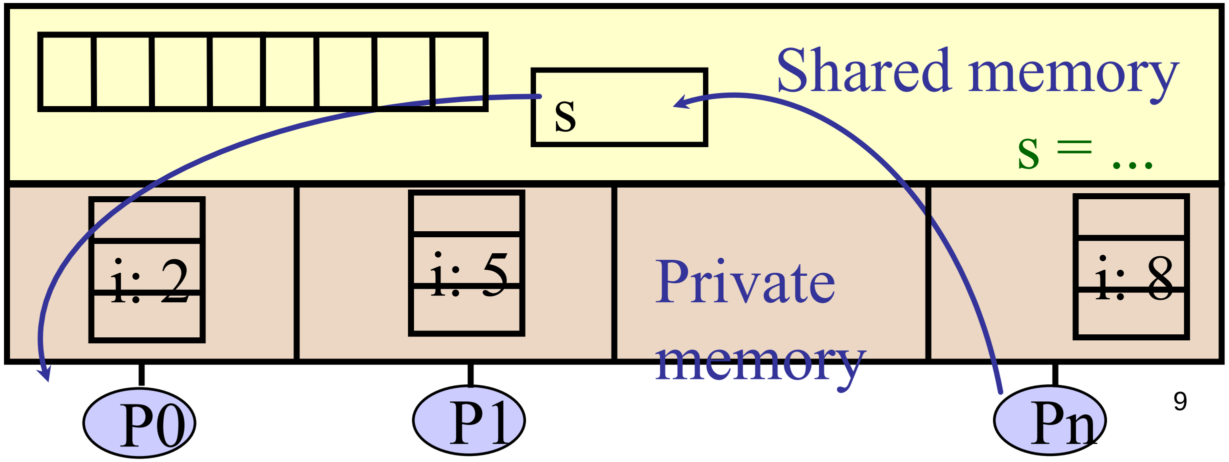
- **Responsiveness**
- **Resource Sharing**
  - Shared memory
- **Economy**
- **Scalability**
  - Explore multi-core CPUs





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

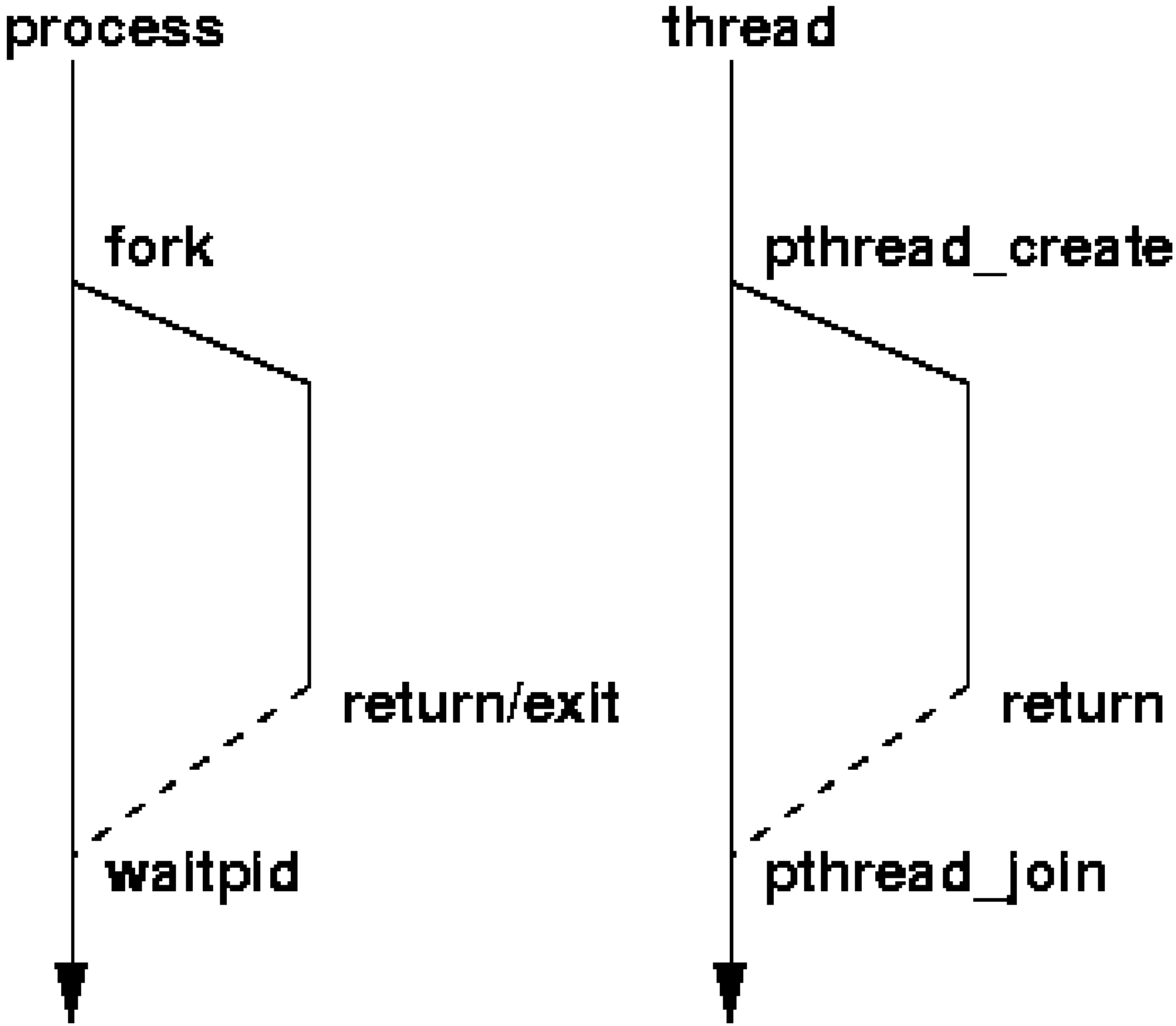


Shared Memory Programming

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

Creation of Unix processes vs. Pthreads



C function for starting a thread

pthread.h

pthread_t

One object for each thread.

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

pthread_t objects

- **Opaque**
- The actual data that they store is system-specific.
- Their data members aren’t directly accessible to user code.
- However, the Pthreads standard guarantees that a pthread_t object does store enough information to uniquely identify the thread with which it’s associated.

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.

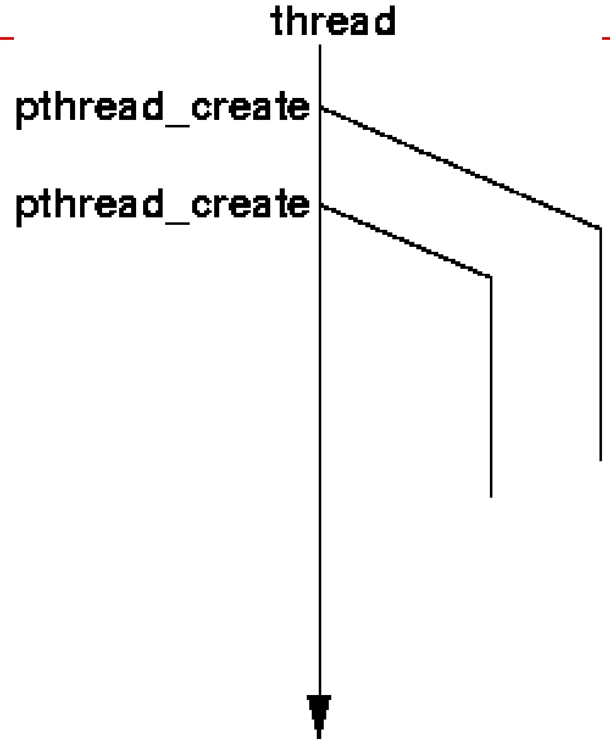
Wait for Completion of Threads

```
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.**

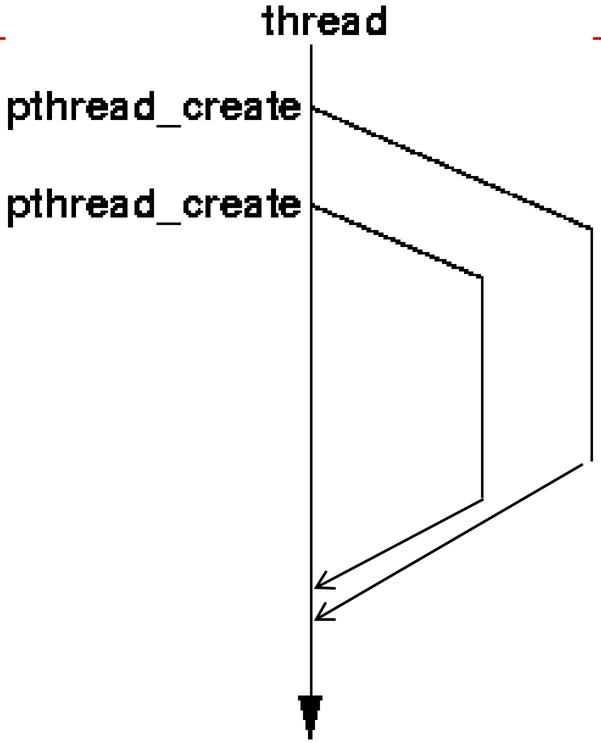
Example of Pthreads




```
#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("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);
}
```



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`

Textbook Hello World example

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

/* Global variable:  accessible to all threads */
int thread_count;

void *Hello(void* rank);  /* Thread function */

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;

   /* Get number of threads from command line */
   thread_count = strtol(argv[1], NULL, 10);

   thread_handles = malloc (thread_count*sizeof(pthread_t));
```

declares the various Pthreads functions, constants, types, etc.

Hello World! (2)

```
for (thread = 0; thread < thread_count; thread++)
   pthread_create(&thread_handles[thread], NULL,
       Hello, (void*) thread);

printf("Hello from the main thread\n");

for (thread = 0; thread < thread_count; thread++)
   pthread_join(thread_handles[thread], NULL);

free(thread_handles);
return 0;
}  /* main */
```

Hello World! (3)

```
void *Hello(void* rank) {
   long my_rank = (long) rank;  /* Use long in case of 64-bit system */

   printf("Hello from thread %ld of %d\n", my_rank, thread_count);

   return NULL;
}  /* Hello */
```

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   <number of threads>

. / pth_hello 1

Hello from the main thread
Hello from thread 0 of 1

. / pth_hello 4

Hello from the main thread
Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

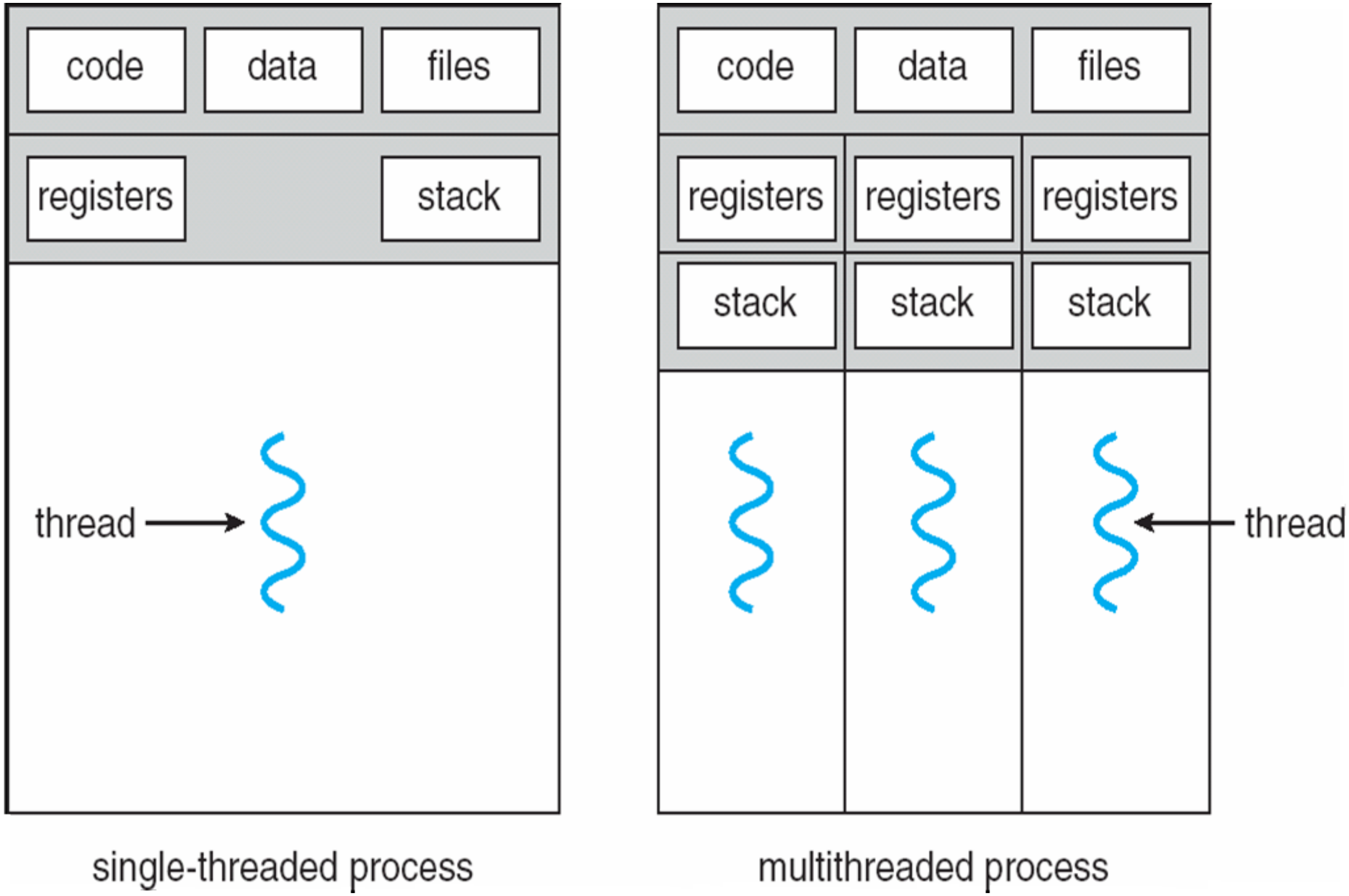
Issues in Threads vs. Processes

- **Shared variables as global variables exist in threads**
  - Can introduce subtle and confusing bugs!
  - Limit use of global variables to situations in which they're really needed.

- **Starting threads**
  - Processes in MPI are usually started by a script.
  - In Pthreads the threads are started by the program executable.

Difference between Single and Multithreaded Processes

Shared memory access for code/data
Separate control flow -> separate stack/registers



Matrix-Vector Multiplication with Pthreads

Textbook 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

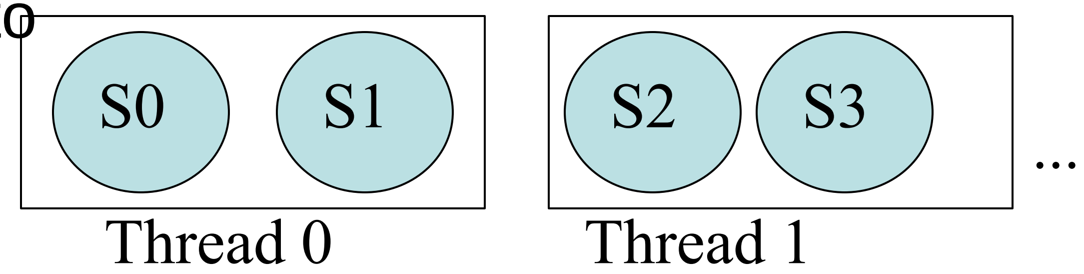
- Task partitioning

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];
```

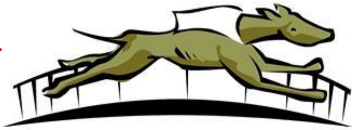
Pthread code for thread with ID rank

```
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

CRITICAL SECTIONS

Data Race Example



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

Thread 0

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

Thread 1

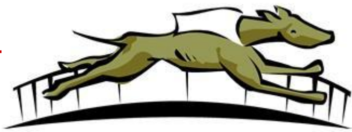
```
for i = n/2, n-1
    s = s + f(A[i])
```

- 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**
5. **Barriers**

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.

Application Pthread Code: Estimating π

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

```
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;
```

Mapping for a multi-core machine

- Two thread distribution

Divide computation to 2 threads or more using block mapping. For example, n=20

Thread 0:	Thread 1:
Iterations 0, 1, 2, .., 9	Iterations 10, 11, 12, .., 19

- No of threads = thread_count
- No of iterations per thread   my_n= n/ thread_count
  - Assume it is an integer?
- Load assigned to my thread:
  - First iteration: my_n * my_rank
  - Last iteration: First iteration + my_n -1

A thread function for computing π

```
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;

   if (my_first_i % 2 == 0)  /* my_first_i is even */
      factor = 1.0;
   else  /* my_first_i is odd */
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

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

Unprotected critical section.

Running results with 1 thread and 2 threads

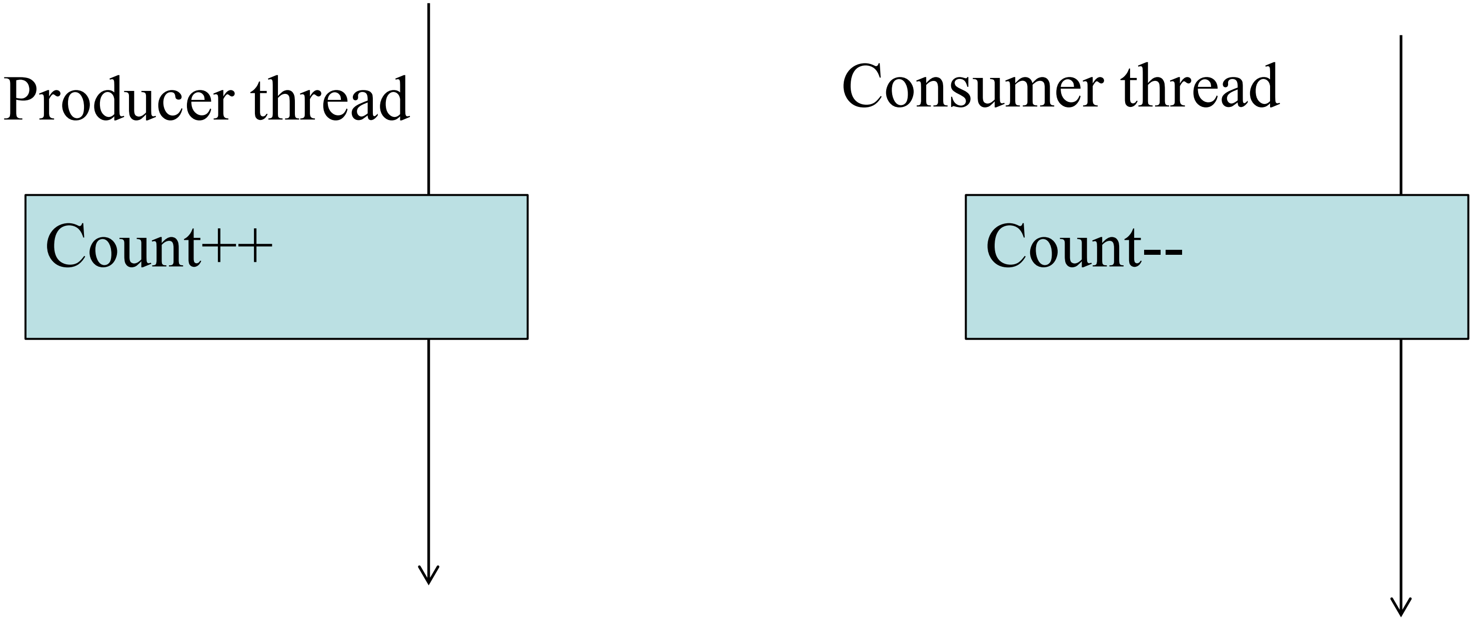
	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

As n becomes larger,

- The one thread result becomes more accurate, gaining more correct digits
- The two-thread result is getting worse or strange

Race Conditions: Example

Count=5



Is count still 5?

Race Conditions: Example

Count=5

Producer thread

Count++:

    register1 = count
    register1 = register1 + 1
    count = register1

Consumer thread

Count—:

    register2 = count
    register2 = register2 - 1
    count = register2

Is count still 5?

Race Conditions: Example

Count=5

Producer thread

Count++:
register1 = count
register1 = register1 + 1

count = register1

Consumer thread

Count—:

register2 = count
register2 = register2 - 1

count = register2

Is count still 5?

Race Condition

- "count = 5" initially:

S0: producer execute register1 = count {register1 = 5}
S1: producer execute register1 = register1 + 1 {register1 = 6}
S2: consumer execute register2 = count {register2 = 5}
S3: consumer execute register2 = register2 - 1 {register2 = 4}
S4: producer execute count = register1 {count = 6 }
S5: consumer execute count = register2 {count = 4}

Busy-Waiting

- **A thread repeatedly tests a condition, but, effectively, does no useful work until the condition has the appropriate value.**

- **Beware of optimizing compilers, though!**

```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

flag initialized to 0 by main thread

Pthreads global sum with busy-waiting

```
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;

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

   for (i = my_first_i; i <
      while (flag != my_ran
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

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

sum is a shared global variable. Can we transform code and minimize thread interaction on this variable?

Global sum with local sum variable/busy waiting (1)

```
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor, my_sum = 0.0;
   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;

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

Global sum with local sum variable/busy waiting

```
   for (i = my_first_i;                                  )
      my_sum += factor/(2*i

   while (flag != my_rank);
   sum += my_sum;
   flag = (flag+1) % thread_count;

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

my_sum is a local variable, not shared.

Still have to contribute my_sum at the end to the global sum variable.

Mutexes (Locks)

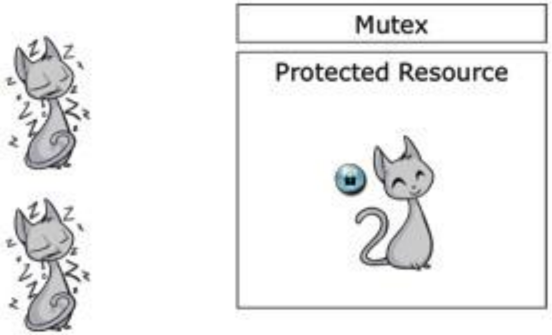
Acquire mutex lock

Critical section

Unlock/Release mutex

- Code structure
- 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.



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;
```

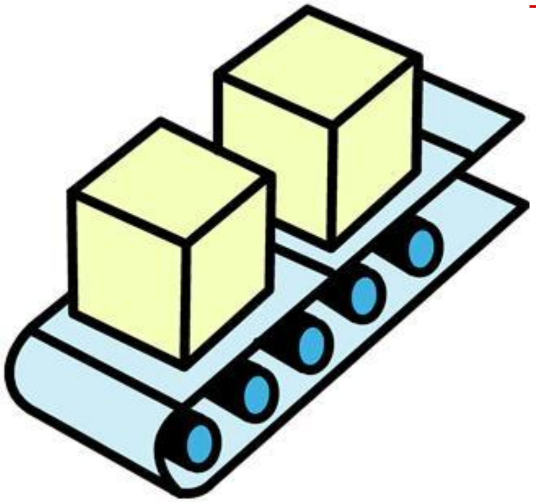
Global sum function that uses a mutex (2)

```
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 */
```

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

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

Run-times (in seconds) of π programs using n = 108 terms on a system with two four-core processors.



Producer-consumer Synchronization and Semaphores

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.

Problems with a mutex solution in multiplying many matrices

product_mat= A*B*C

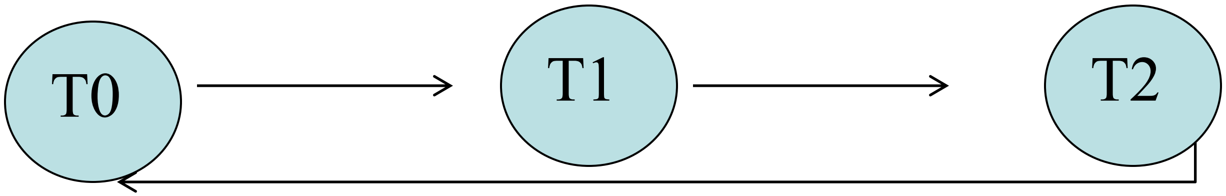
Out of order multiplication → product_mat= A*C*B

That is wrong

```
/* n and product_matrix are shared and initialized by the main thread *
/* product_matrix is initialized to be the                            *
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);
   Multiply_matrix(product_mat, my_mat);
   pthread_mutex_unlock(&mutex);
   Free_matrix(&my_mat);
   return NULL;
}  /* Thread_work */
```

The order of multiplication is not defined

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?

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
   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: Generalization from mutex locks

- Semaphore *S* – integer variable
  - Initial value can be negative or positive
- 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.'

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 */);
```

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]);
```

Typical Producer-Consumer Flow in Using a Semaphore

- Thread 1: //Consumer

sem_wait(s);

// condition is satisfied

Consume an item

- Thread 2: // Producer

  Produce an item

  sem_post(s);

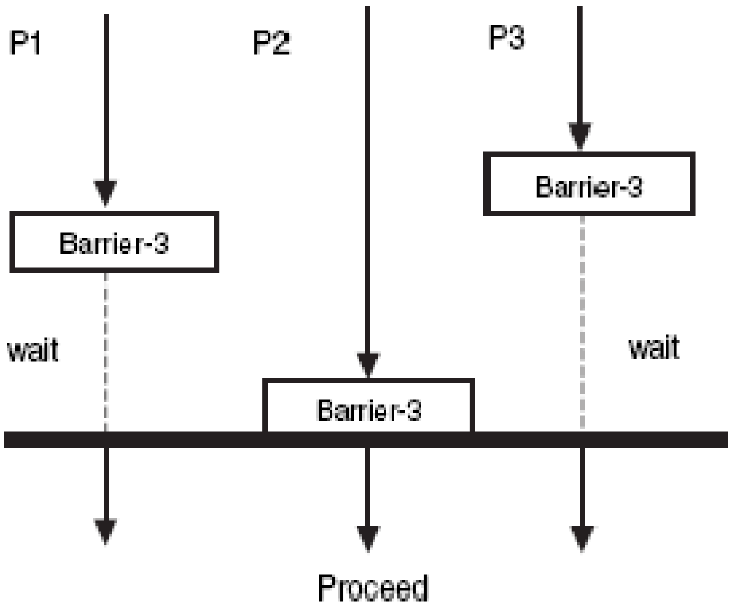
- What does initial value s mean?
  – s=0?
  – s=2?
  – s=-2

BARRIERS AND CONDITION VARIABLES

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.



Application: Start timing of all threads at a fixed point.

```
/* Shared */
double elapsed_time;
. . .
/* Private */
double my_start, my_finish, my_elapsed;
. . .
Synchronize threads;
Store current time in my_start;
/* Execute timed code */
. . .
Store current time in my_finish;
my_elapsed = my_finish − my_start;

elapsed = Maximum of my_elapsed values;
```

Using barriers for debugging

```
point in program we want to reach;
barrier;
if (my_rank == 0) {
   printf("All threads reached this point\n");
   fflush(stdout);
}
```

Implement a barrier with busy-waiting and a mutex

- A shared counter as # of threads waiting in this point.

```
/* 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);
   . . .
}
```

Need one counter variable for each instance of the barrier, otherwise problems are likely to occur.

Implementing a barrier with semaphores

```
/* Shared variables */
int counter;                  0 */
sem_t count_sem;           /* In
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);
   }
```

Protect counter

Wait all threads to come

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

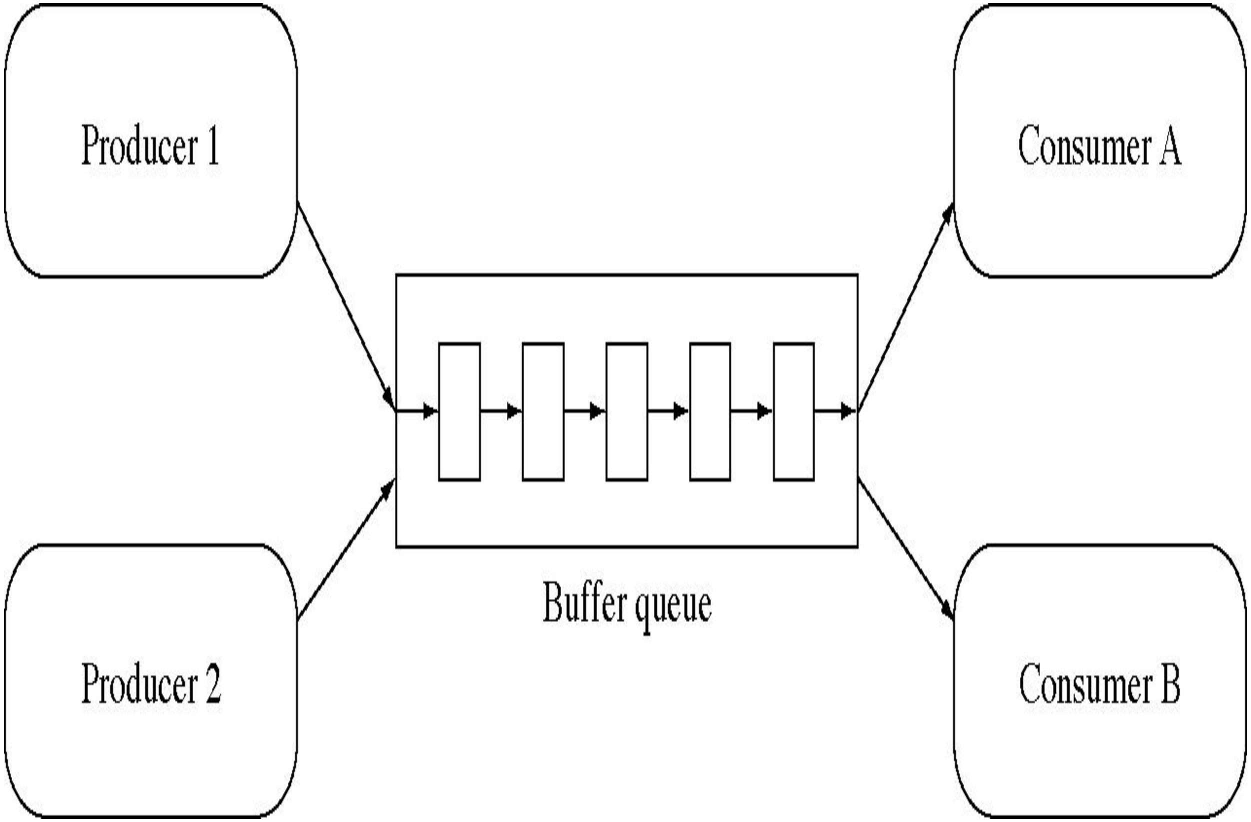
```
Mutex_lock(mutex);
  While (condition is not satisfied)
          Cond_Wait(mutex, cond);
   Critical Section;
Mutex_unlock(mutex)
```

- 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

Implementing a barrier with condition variables

Text book p.180

```
/* Shared */
int counter = 0;
pthread_mutex_t mutex;
pthread_cond_t cond_var;
. . .
void* Thread_work(. . .) {
    . . .
    /* Barrier */
    pthread_mutex_lock(&mutex);
    counter++;
    if (counter == thread_count) {
        counter = 0;
        pthread_cond_broadcast(&cond_var);
    } else {
        while (pthread_cond_wait(&cond_var, &mutex) != 0);
    }
    pthread_mutex_unlock(&mutex);
    . . .
}
```

Concluding Remarks (1)

- A thread in shared-memory programming is analogous to a process in distributed memory programming.
  - However, a thread is often lighter-weight
- In Pthreads programs, all the threads have access to global variables, while local variables usually are private to the thread running the function.
- When multiple threads access a shared resource without controlling, 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

Concluding Remarks (2)

- **Busy-waiting can be used for critical sections with a flag variable and a while-loop**
  - It can waste CPU cycles, & may be unreliable
- **A mutex arrange for mutually exclusive access to a critical section.**
- **Semaphore & Condition variables**
  - more powerful synchronization primitives.
- A barrier is a point in a program at which the threads block until all of the threads have reached it.