Parallel Programming

Shared-Memory Programming with Pthreads

(Part I)

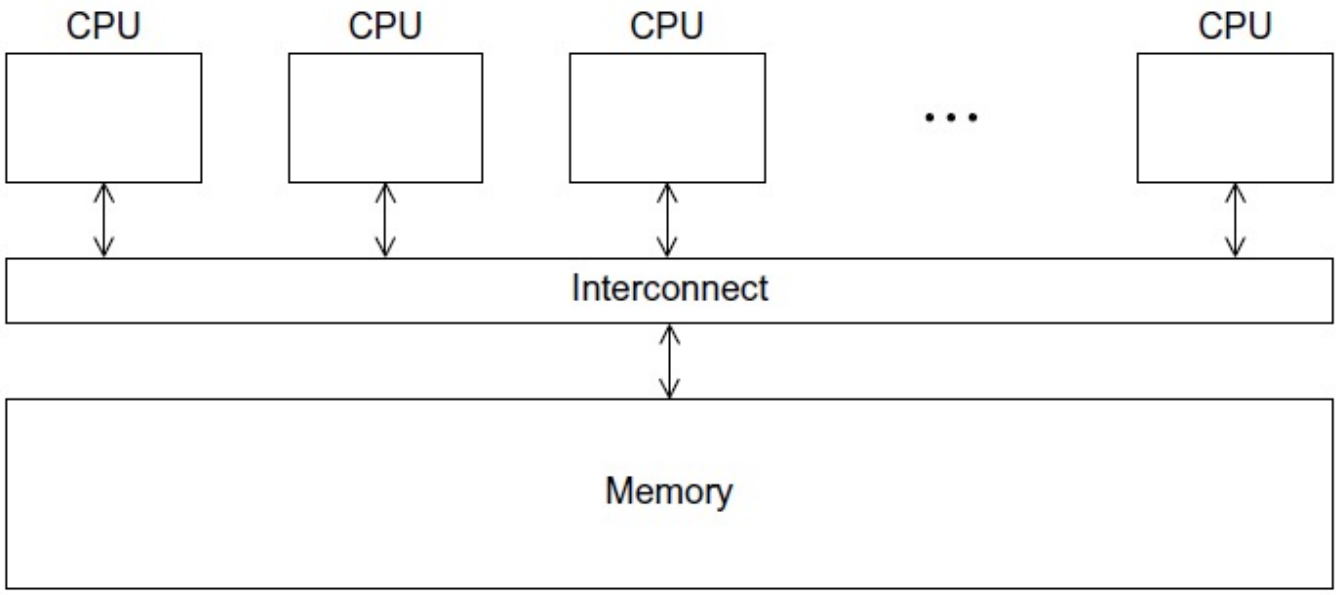
Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Shared-Memory System

- Recall that from a programmer's point of view a shared-memory system is one in which all the cores can access all the memory locations



Issues to be addressed

- Learn how to synchronize threads
- Learn how to put a thread “to sleep” until a condition has occurred
- See that there are some circumstances in which it may at first seem that a critical section must be quite large
- See that the use of cache memories can actually cause a shared-memory program to run more slowly
- See that functions that “maintain state” between successive calls can cause inconsistent or even incorrect results

Outline

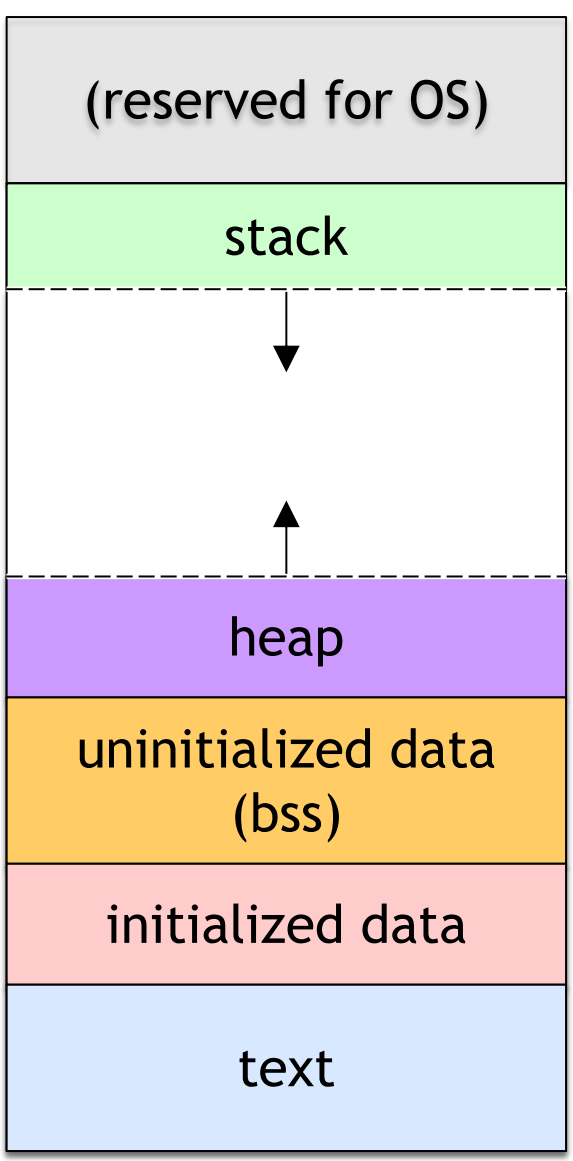
- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Processes

- A process is an instance of a running program. In addition to its executable, it consists of the following:
  - A block of memory for the stack
  - A block of memory for the heap
  - Descriptors of resources that the system has allocated for the process
    - E.g., file descriptors
  - Security information
    - E.g., which hardware/software resources the process can access
  - Information about the state of the process
    - E.g., PC, registers

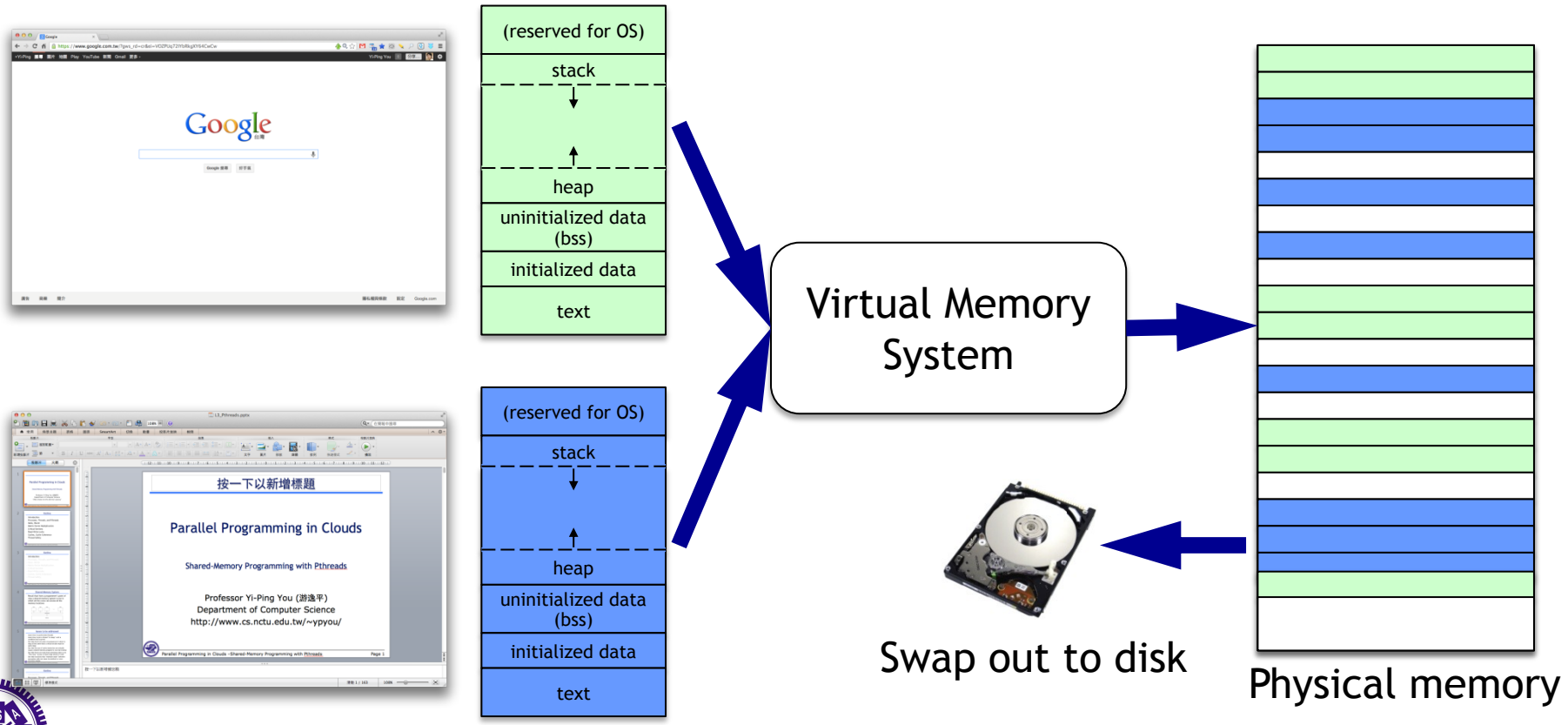
Memory Layout of C Programs

```
char *string = "hello";
int iSize;

char* foo(int x) {
  static int y;
  char *p;
  iSize = 8;
  p = (char *)malloc(iSize);
  return p;
}

int main() {
  int z;
  ...
}
```



Virtual Memory

- OS also gives every application the illusion of having 4 GB of memory (on 32-bit machines)!
  - Each process has its own address space
  - In reality, physical RAM is split across multiple applications



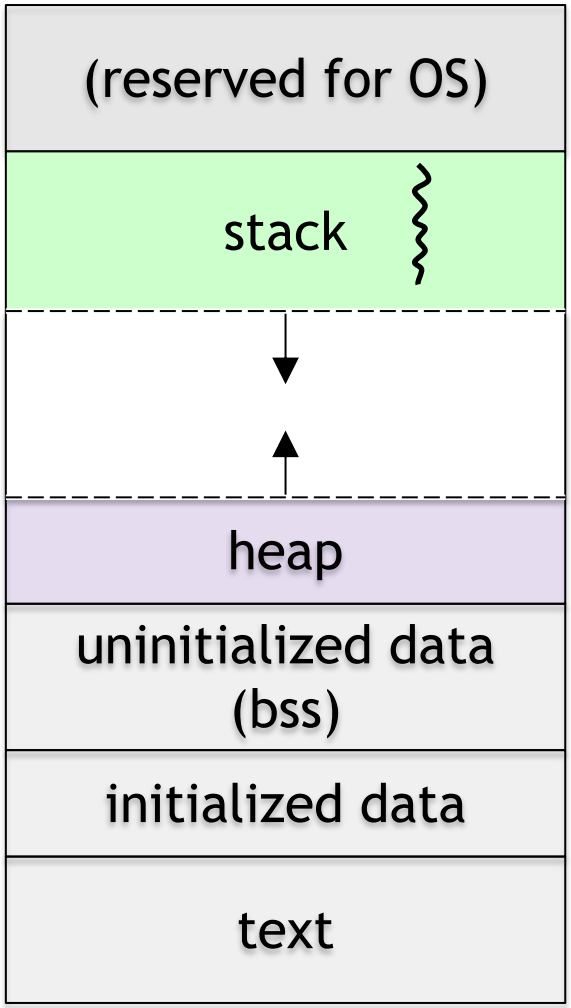
Processes (Cont'd)

- In most systems, by default, a process' memory blocks are private
- Another process can't directly access the memory of a process unless the operating system intervenes
- This is even more crucial in a multiuser environment
  - One user's processes shouldn't be allowed access to the memory of another user's processes
- However, this isn't what we want when we're running shared-memory programs
  - At a minimum, we'd like certain variables to be available to multiple processes

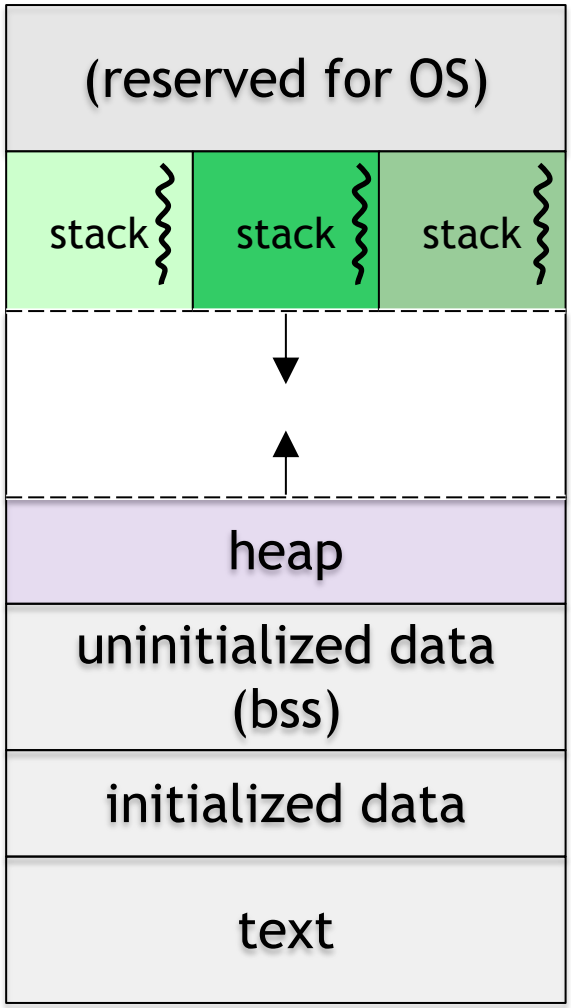
Threads

- **Thread**, comes from the concept of "thread of control"
- A thread of control is just a sequence of statements in a program
- The term suggests a stream of control in a single process, and in a shared-memory program a single *process* may have multiple *threads* of control

Memory Layout of Multi-threaded Programs



Single-threaded process

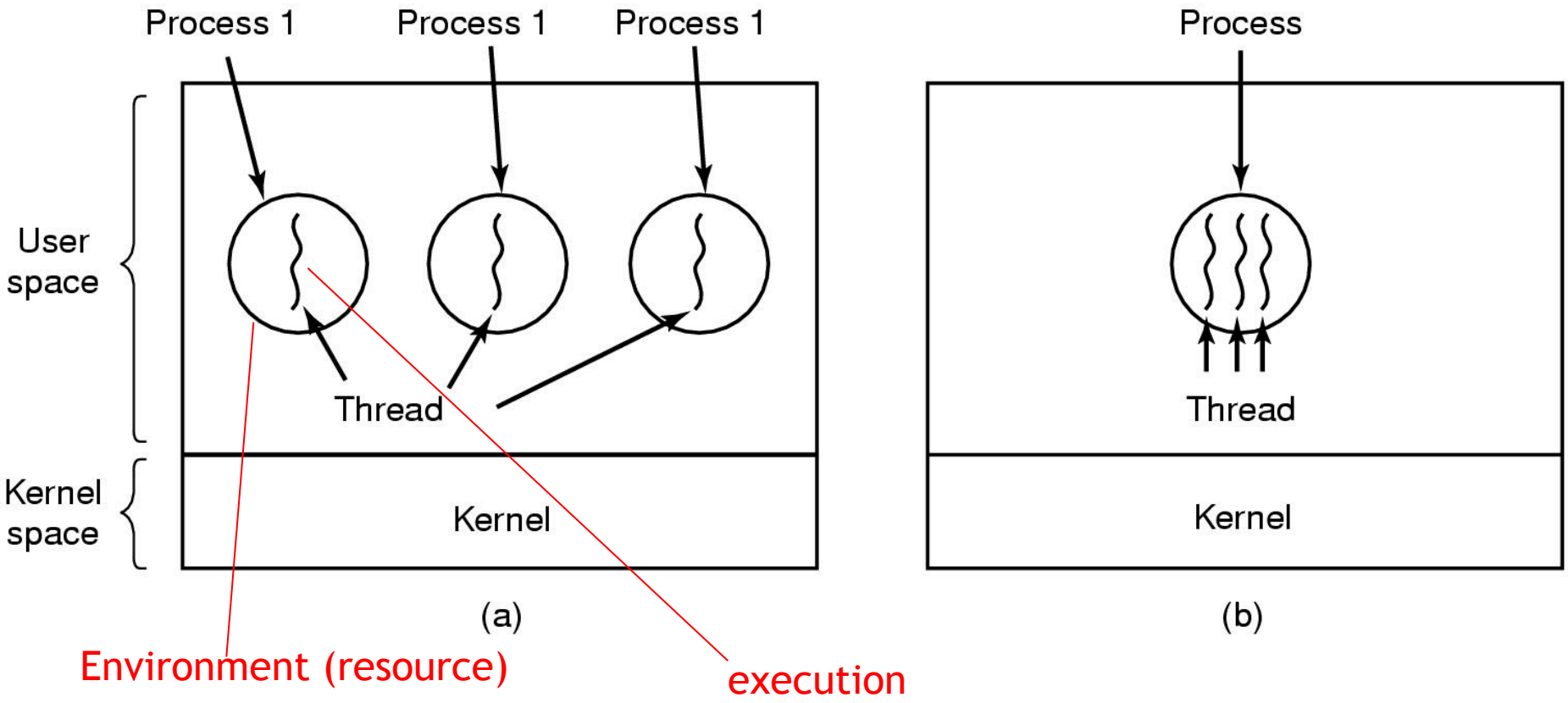


Multi-threaded process

Each thread has its own context and thus its own stack

Shared memory among threads

Threads: Lightweight Processes



(a) Three processes each with one thread
(b) One process with three threads

Processes vs Threads

Property	Processes created with fork	Threads of a process	Ordinary function calls
Variables	get copies of all variables	share static data/heap	share static data/heap
IDs	get new process IDs	share the same process ID but have unique thread ID	share the same process ID (and thread ID)
Communication	Must explicitly communicate, e.g.pipes or use small integer return value	May communicate with return value or shared variables if done carefully	May communicate with return value or shared variables (don't have to be careful)
Parallelism (one CPU)	Concurrent	Concurrent	Sequential
Parallelism (multiple CPUs)	May be executed simultaneously	Kernel threads may be executed simultaneously	Sequential

Pthreads

- Pthreads is not a programming language (such as C or Java)
- Pthreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Originally IEEE POSIX 1003.1c
  - Should be relatively uniform across UNIX-like OS platforms
    - Actually, it's available for many different OSs, including Windows, MacOS, and, of course, Linux, BSD and Solaris
- Pthreads is an interface. How it's implemented depends on the platform
  - E.g., Linux uses kernel threads; Windows uses Win32 threads, etc.

Pthreads (Cont'd)

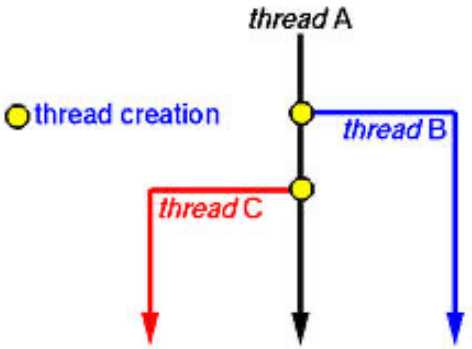
- If you have a single-/multi-core system, you can have many threads in your system, and the OS schedule your threads
- Pthreads specifies a *library* that can be linked with C programs
  - Since Pthreads is a C library, it can, in principle, be used in C++ programs
  - However, C++11 and its standard library vastly improves concurrent programming
    - It may make sense to use it rather than Pthreads if you're writing C++ programs

Pthread Operations

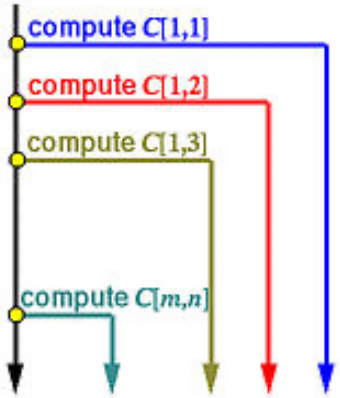
POSIX function	Description
`pthread_create`	create a thread
`pthread_detach`	set thread to release resources
`pthread_equal`	test two thread IDs for equality
`pthread_exit`	exit a thread without exiting process
`pthread_kill`	send a signal to a thread
`pthread_join`	wait for a thread
`pthread_self`	find out own thread ID

Creation of Threads

- `pthread_create()`
  - Creates a new thread to execute a specified thread function
    - The parent thread continues to run
    - New threads may be created any time during the program by any thread
  - Parent thread may pass data to its child thread during creation
- Thread ID
  - Each thread in a process is identified by a thread ID.
  - When referring to thread IDs in C or C++ programs, use the type `pthread_t`




```
for (int i = 1; i <= m; i++)
   for (int j = 1; j <= n; j++)
      create a thread to compute C[i, j];
```



Source: http://www.cs.mtu.edu/~shene/NSF-3/e-Book/FUNDAMENTALS/thread-management.html

The Syntax of `pthread_create` (1/2)

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

- The first argument is a pointer to the appropriate `pthread_t` object
  - The object is not allocated by the call to `pthread_create`; it must be allocated *before* the call
- The `attr` argument points to a `pthread_attr_t` structure whose contents are used at thread creation time to determine attributes for the new thread
  - If `attr` is `NULL`, then the thread is created with default attributes

The Syntax of `pthread_create` (2/2)

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

- The third argument is the function that the thread is to run
- The last argument is a pointer to the argument that should be passed to the function `start_routine`

`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

Example: Creation of 2 Threads (1/2)

```
/* The main program.  */

int main ()
{
  pthread_t thread1_id;
  pthread_t thread2_id;
  struct char_print_parms thread1_args;
  struct char_print_parms thread2_args;

  /* Create a new thread to print 30,000 'x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20,000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  return 0;
}
```

Define two thread IDs

Arguments to pass to thread functions

Setting up arguments for thread 1

Create a thread to run `char_print()`, with arguments

Parent thread exit immediately!

Create a thread to run `char_print()`, with arguments

Source: Advanced Linux Programming by CodeSourcery LLC, http://www.advancedlinuxprogramming.com/

Example: Creation of 2 Threads (2/2)

```
void* char_print (void* parameters)
{
  /* Cast the cookie pointer to the right type.  */
  struct char_print_parms* p = (struct char_print_parms*) parameters;
  int i;

  for (i = 0; i < p->count; ++i)
    fputc (p->character, stderr);
  return NULL;
}
```

Local variables (`p`, `i`) within thread function are private, i.e. only this thread see them

- This illustrates Single-Program Multiple Data (SPMD)
- 30000 “x” and 20000 “o” mixed in a random order
- Two threads execute independently in MIMD fashion, not SIMD
- Data dependencies exist!
  - Both threads call `fputc()` to print characters to the same device `stderr`
  - The contention occurs at OS level and it serializes execution

Source: Advanced Linux Programming by CodeSourcery LLC, http://www.advancedlinuxprogramming.com/

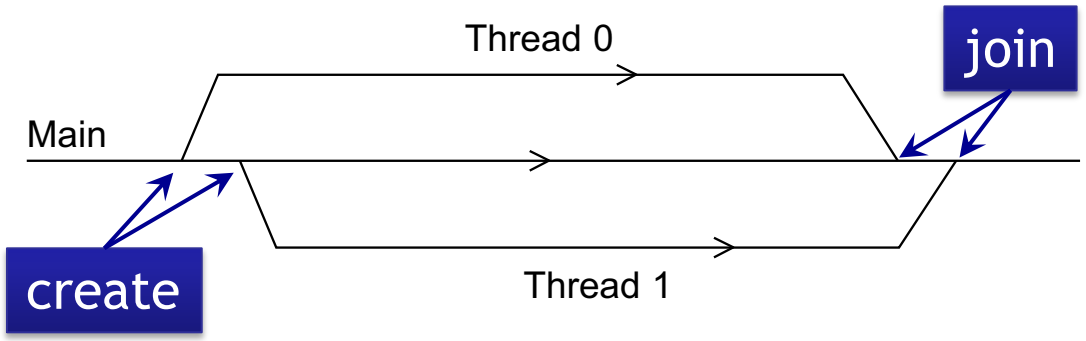
Parallel Threads Execution

- "Creation of 2 Threads" example has a serious bug!
  - Bugs in threads are usually hard to find because programmers are not used to think parallel
- Where is the bug?
  - The main thread creates the thread parameter structures (`thread1_args` and `thread2_args`) as local variables, and then passes pointers to these structures to the threads it creates
  - What happens when `main()` finishes before any of its child threads are done?
    - The memory containing `thread1_args` and `thread2_args` will be deallocated while the other two threads are still accessing it
- Solution
  - `main()` need to join child threads before it exits

The Syntax of `pthread_join`

```
int pthread_join(
      pthread_t  thread      /* in  */,
      void**     ret_val_p   /* out */);
```

- Wait for the thread associated with the `pthread_t` object to complete
- The second argument (address of a void * variable) can be used to receive any return value computed by the thread

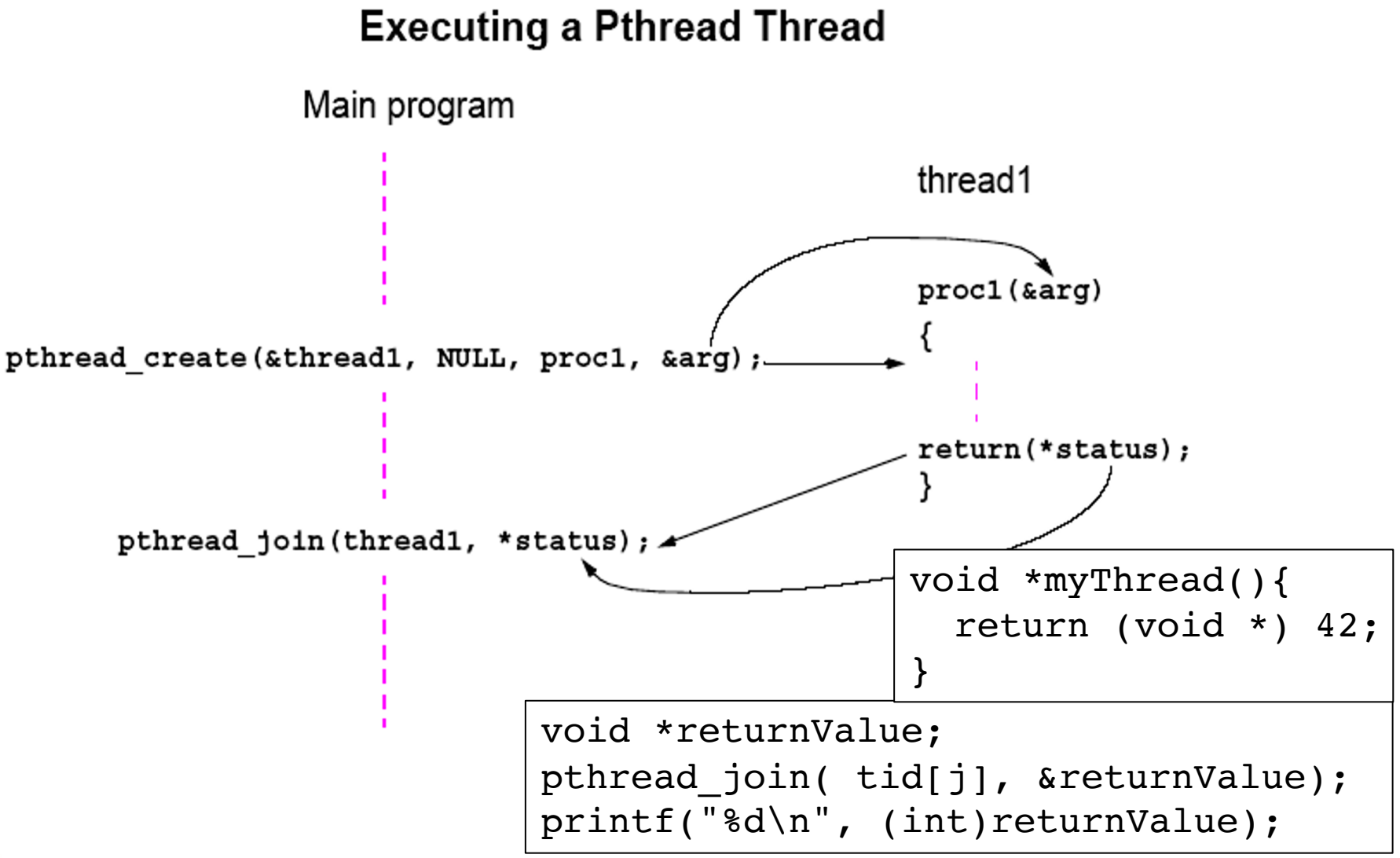


Joining Threads

```
  /* Create a new thread to print 30,000 x's.  */
  thread1_args.character = 'x';
  thread1_args.count = 30000;
  pthread_create (&thread1_id, NULL, &char_print, &thread1_args);

  /* Create a new thread to print 20,000 o's.  */
  thread2_args.character = 'o';
  thread2_args.count = 20000;
  pthread_create (&thread2_id, NULL, &char_print, &thread2_args);

  /* Make sure the first thread has finished.  */
  pthread_join (thread1_id, NULL);
  /* Make sure the second thread has finished.  */
  pthread_join (thread2_id, NULL);

  /* Now we can safely return.  */
  return 0;
}
```

Source: Advanced Linux Programming by CodeSourcery LLC, http://www.advancedlinuxprogramming.com/

Creating and Joining Threads



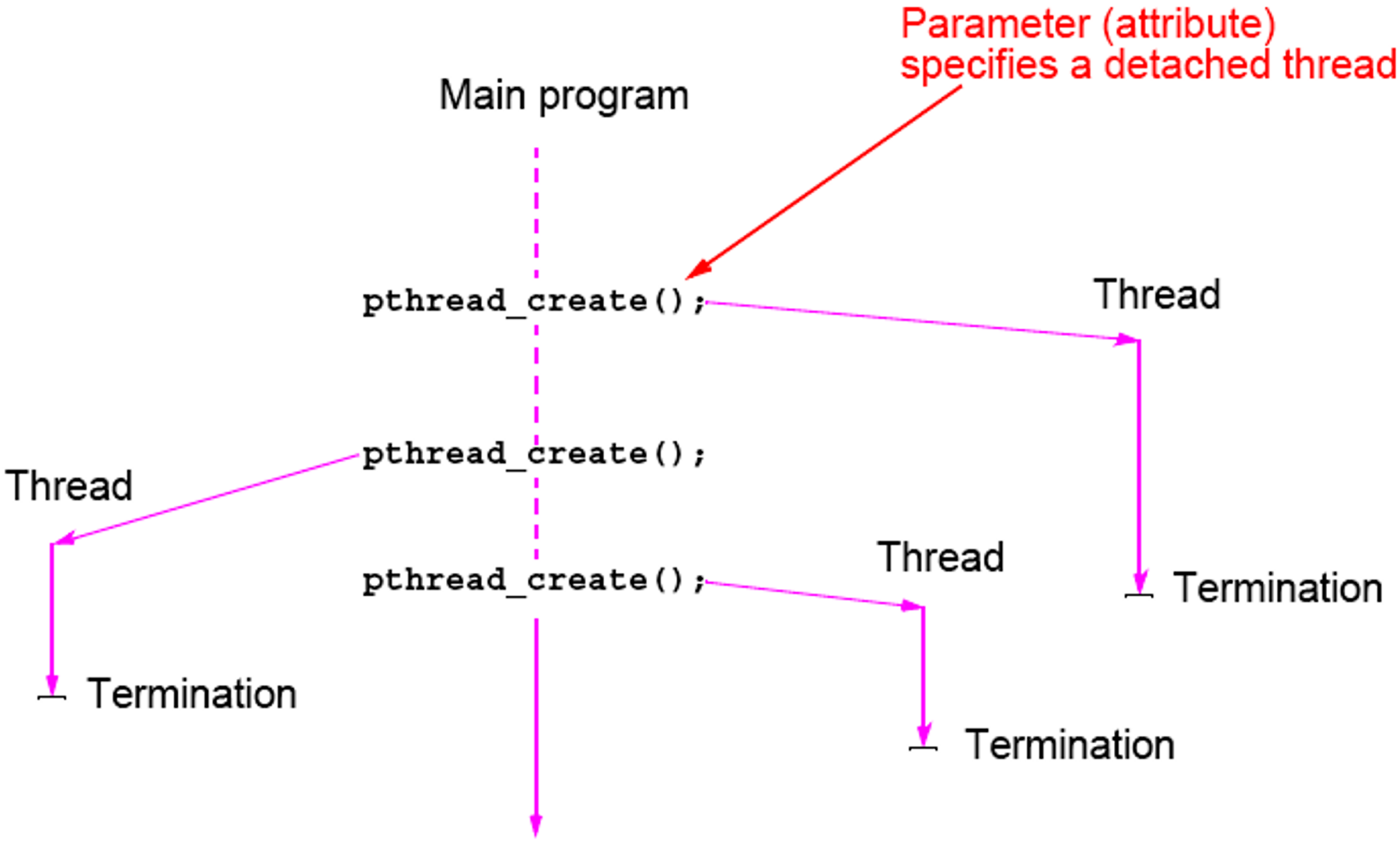
Termination of Threads

- Normally a thread exits in one of two ways
- Implicit: Normally returning from the thread function
  - The return value from the thread function is returned to the parent
- Explicit: A thread can exit explicitly by calling `pthread_exit(status)`
  - May be called from within the thread function or from some other function called directly or indirectly by the thread function
  - The argument (`status`) to `pthread_exit` is the thread's return value
- Terminating a thread from another thread
  - `pthread_cancel(pthread_t tid)`

Detached Threads

- It may be that thread is not bothered when a thread it creates terminates and then a join not needed
- Threads not joined are called detached threads
- When detached threads terminate, they are destroyed and their resource released

Detached Threads (Cont'd)



Thread Attributes

- Thread attributes provide a mechanism for fine-tuning the behavior of individual threads
  - Detached state is one thread attribute which is not joinable and will release the resources
  - The other available attributes are primarily for real-time programming
- A thread may be created as a
  - **joinable** thread (the default)
    - When it terminates, its exit state is still kept in the system until another thread calls `pthread_join()` to obtain its return value
  - **detached** thread
    - Its exit state is cleaned up automatically when it terminates
    - Another thread may not synchronize on its completion by using `pthread_join` or obtain its return value

Setting Attrributes to Detached State

```
int main ()
{
  pthread_attr_t attr;
  pthread_t thread;

  pthread_attr_init (&attr);
  pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
  pthread_create (&thread, &attr, &thread_function, NULL);
  pthread_attr_destroy (&attr);

  /* Do work here...  */

  /* No need to join the second thread.  */
  return 0;
}
```

Define a variable to store thread attributes

Initialize the attribute variable

Set it to detached state

Child thread will be a detached thread

No need for attribute variable → destroy it

Source: Advanced Linux Programming by CodeSourcery LLC, http://www.advancedlinuxprogramming.com/

Thread Attribute Functions

- Basic management
  - Create (initialize with default values) and destroy a thread attribute object
    - E.g., `pthread_attr_init()` and `pthread_attr_destroy()`
- Specifying stack information
  - Get and set the stack size in the attribute object for new threads
    - E.g., `pthread_attr_getstacksize()` and `pthread_attr_setstacksize()`
- Thread scheduling attributes
  - Get and set the value of the schedule policy attribute of a thread attributes object
    - E.g., `pthread_attr_getschedpolicy()` and `pthread_attr_setschedpolicy()`
- Detachable or joinable (undetachable)
  - Control whether the thread is created in the joinable state
    - E.g., `pthread_attr_getdetachstate()` and `pthread_attr_setdetachstate()`

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

A Pthreads "Hello World!"

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <pthread.h>
4
5  /* Global variab
6  int thread_count;
7
8  void* Hello(void* rank);  /* Thread function
9
10 int main(int argc, char* argv[]) {
11    long       thread;  /* Use long in case of a 64-bit system */
12    pthread_t* thread_handles;
13
14    /* Get number of threads from command line */
15    thread_count = strtol(argv[1], NULL, 10);
16
17    thread_handles = malloc (thread_count*sizeof(pthread_t));
18
19    for (thread = 0; thread < thread_count; thread++)
20       pthread_create(&thread_handles[thread], NULL,
21          Hello, (void*) thread);
22
23    printf("Hello from the main thread\n");
24
25    for (thread = 0; thread < thread_count; thread++)
26       pthread_join(thread_handles[thread], NULL);
27
28    free(thread_handles);
29    return 0;
30 }  /* main */
```

```
32 void* Hello(void* rank) {
33    long my_rank = (long) rank;
         /* Use long in case of 64-bit system */
35    printf("Hello from thread %ld of %d\n", my_rank,
         thread_count);
36
37    return NULL;
38 }  /* Hello */
```

Must be included to use Pthread functions

Global (shared) variable

Allocate storage for one **pthread_t** object for each thread

Used for storing thread-specific information

Create a thread

Wait for the thread associated with the **pthread_t** object to complete

Compilation

- The program is compiled like an ordinary C program, with the possible exception that we may need to link in the Pthreads library:
  - `$ gcc -o pth_hello pth_hello.c -lpthread`
- The `-lpthread` tells the compiler that we want to link in the Pthreads library

Execution

- To run the program, we just type:
  - `$ ./pth_hello <number of threads>`
- For example, to run the program with one thread, we type:
  - `$ ./pth_hello 1`
- The output will look something like this:

```
Hello from the main thread
Hello from thread 0 of 1
```

Execution (Cont'd)

- To run the program with four threads, we type:
  - `$ ./pth_hello 4`
- The output will look something like this:

```
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
```

  - In single core system, output will be in order

Starting the Threads

- There is no technical reason for each thread to run the same function; we could have one thread run hello, another run goodbye
- Each thread will run the same thread function, but we'll obtain the effect of different thread functions by branching within a thread

Running the Threads

- The thread that's running the main function is sometimes called the **main thread**
- After starting the threads, it prints the message:
  - `Hello from the main thread`
- In the meantime, the threads started by the calls to `pthread_create` are also running
- In Pthreads, the programmer doesn't directly control where the threads are run
  - There's no argument in `pthread_create` saying which core should run which thread
  - Thread placement is controlled by the operating system

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Matrix-Vector Multiplication

- Let's take a look at writing a Pthreads matrix-vector multiplication program

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

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}

a: an $m \times n$ matrix

Matrix-Vector Multiplication: 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[i] += A[i][j]* x[j];
}
```

Parallelizing the Matrix Multiplication

- Using data parallelism
  - Dividing the iterations of the outer loop among the threads (assuming $m = n = 6$, $t = 3$)

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];
```

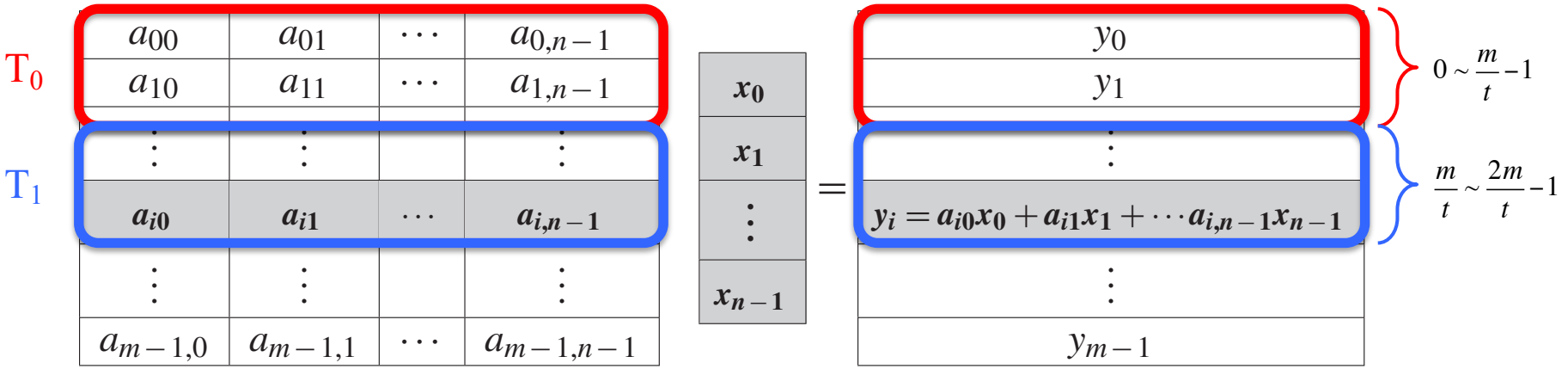
general case

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

- At a minimum, **x** should be shared
- Each thread only needs to access its assigned rows of **A** and assigned components of **y**

Parallelizing the Matrix Multiplication

- Thread 0 gets the first m/t, thread 1 gets the next m/t, and so on



$$\mathrm{T}_q: \quad q \times \frac{m}{t} \sim (q+1) \times \frac{m}{t} - 1$$

Matrix-Vector Multiplication: Parallel Pseudo-Code (1/2)

- Let's also make **A** and **y** shared
  - Although this might seem to violate our principle that we should only make variables global that need to be global

```
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*n+j]*x[j];
   }

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

- All of the variables—except **y**—are only read by the threads
- There are no attempts by two (or more) threads to modify any single component of **y**

What happens when multiple threads update a single memory location?

Matrix-Vector Multiplication: Parallel Pseudo-Code (2/2)

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

/* Global variables */
int     thread_count;
int     m, n;
double* A;
double* x;
double* y;

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   thread_count = atoi(argv[1]);
   thread_handles = malloc(thread_count*sizeof(pthread_t));

   A = malloc(m*n*sizeof(double));
   x = malloc(n*sizeof(double));
   y = malloc(m*sizeof(double));

   Read_matrix("Enter the matrix", A, m, n);
   Read_vector("Enter the vector", x, n);

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

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

   return 0;
}  /* main */
```

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Calculating Pi

- Let's look at an example: Leibniz's Formula
- Let's try to estimate the value of pi. There are lots of different formulas we could use. One of the simplest is:

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

```
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;
```

Parallelizing Calculation of Pi

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

long thread_count;
long long n;
double sum;

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

   thread_count = strtol(argv[1], NULL, 10);
   n = strtoll(argv[2], NULL, 10);
   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));

   sum = 0.0;
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Thread_sum, (void*)thread);

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

   sum = 4.0*sum;

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

Global Sum Function

$$\pi = 4\left(\underbrace{1 - \frac{1}{3} + \frac{1}{5}}_{T_0} - \underbrace{\frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1}}_{T_1} + \underbrace{\cdots}_{T_n}\right)$$

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

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

Peculiar Results due to Race Conditions

- If we run the Pthreads program with two threads and *n* is relatively small, the results of the Pthreads program are in agreement with the serial sum program
- As *n* gets larger, we start getting some peculiar results
  - For example, with a dual-core processor we get the following results:
  - **+= operator is not atomatic**

	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

- It matters if multiple threads try to update a single shared variable
  - We have a **race condition**

An Example of Race Condition

- Suppose two threads run the following code

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

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`

Critical Section

- It's a block of code that updates a shared resource that can only be updated by one thread at a time

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

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

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Busy-Waiting

- A simple approach that doesn't involve any new concepts is the use of a flag variable
- Suppose flag is a shared `int` that is set to 0 by the main thread

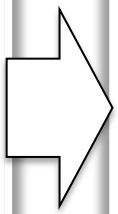
```
y = Compute(my_rank);
while (flag != my_rank);
x = x + y;
flag++;
```

- Thread 1 cannot enter the critical section until thread 0 has completed the execution of `flag++`

Potential Problem Caused by Optimizing Compilers

- An optimizing compiler might therefore determine that the program would make better use of registers if the order of the statements were switched

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



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

- This defeats the purpose of the busy-wait loop
  - The simplest solution to this problem is to turn compiler optimizations off when we use busy-waiting

Parallelizing Calculation of Pi Using Busy Waiting

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

long thread_count;
long long n;
int flag;
double sum;

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

   thread_count = strtol(argv[1], NULL, 10);
   n = strtoll(argv[2], NULL, 10);
   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));

   sum = 0.0;
   flag = 0;
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Thread_sum, (void*)thread);

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

   sum = 4.0*sum;

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

Global Sum Function Using Busy Waiting (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;

   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) {
      while (flag != my_rank);
      sum += factor/(2*i+1);
      flag = (flag+1) % thread_count;
   }

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

Global Sum Function Using Busy Waiting (2)

```
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;

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

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

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

Busy-Waiting is Not Great

- We can immediately see that busy-waiting is not an ideal solution to the problem of controlling access to a critical section
- Since thread 1 will execute the test over and over until thread 0 executes `flag++`, if thread 0 is delayed, thread 1 will simply "spin" on the test, eating up CPU cycles
- There are much better solutions
  - Mutexes
  - Semaphores

Why Busy-Wait Is Slower?

- Suppose we have only two cores

Table 4.2 Possible Sequence of Events with Busy-Waiting and More Threads than Cores

Time	flag	Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
0	0	crit sect	busy-wait	susp	susp	susp
1	1	terminate	crit sect	susp	busy-wait	susp
2	2	—	terminate	susp	busy-wait	busy-wait
⋮	⋮			⋮	⋮	⋮
?	2	—	—	crit sect	susp	busy-wait

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Mutexes

- **Mutex** is an abbreviation of *mutual exclusion*
- A mutex is a special type of variable that, together with a couple of special functions, can be used to restrict access to a critical section to a single thread at a time
- The Pthreads standard includes a special type for mutexes: `pthread_mutex_t`
- A variable of type `pthread_mutex_t` needs to be initialized by the system before it's used

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

pthread_mutex_t

- To gain access to a critical section, a thread calls:

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

- When a thread is finished executing the code in a critical section, it should call:

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

- When a Pthreads program finishes using a mutex, it should call:

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

Parallelizing Calculation of Pi Using a Mutex

```
#include <pthread.h>

long thread_count;
long long n;
double sum;
pthread_mutex_t mutex;

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

   thread_count = strtol(argv[1], NULL, 10);
   n = strtoll(argv[2], NULL, 10);
   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
   pthread_mutex_init(&mutex, NULL);
   sum = 0.0;

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

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

   sum = 4.0*sum;

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

Global Sum Function Using a Mutex

```
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;

   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 */
```

Notice that the first thread to call `pthread_mutex_lock` will be the first to execute the code in the critical section. Subsequent accesses will be scheduled by the system.

Busy-Wait vs Mutex: Performance

- We don't see much difference in the overall run-time when the programs are run with fewer threads than cores
  - Each thread only enters the critical section once
- When we use busy-waiting, performance can degrade if there are more threads than cores

Table 4.1 Run-Times (in Seconds) of π 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

Busy-Waiting vs Mutexes: Adaptability

- Busy-waiting has the property by which we know, in advance, the order in which the threads will execute the code in the critical section: thread 0 is first, then thread 1, then thread 2, and so on
- With mutexes, the order in which the threads execute the critical section is left to chance and the system
- Since addition is commutative, this doesn't matter in our program for estimating Pi
  - What if we also want to control the order in which the threads execute the code in the critical section?

Example: Matrix Multiplication

- Suppose each thread generates an $n \times n$ matrix, and we want to multiply the matrices together in thread-rank order
- Matrix multiplication is not commutative

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix},$$

$$\mathbf{AB} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} = \begin{pmatrix} a\alpha + b\gamma & a\beta + b\delta \\ c\alpha + d\gamma & c\beta + d\delta \end{pmatrix},$$

$$\mathbf{BA} = \begin{pmatrix} \alpha & \beta \\ \gamma & \delta \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} \alpha a + \beta c & \alpha b + \beta d \\ \gamma a + \delta c & \gamma b + \delta d \end{pmatrix}.$$

Mutexes Not Working for Matrix Multiplication

- Since matrix multiplication isn't commutative, our mutex solution would have problems

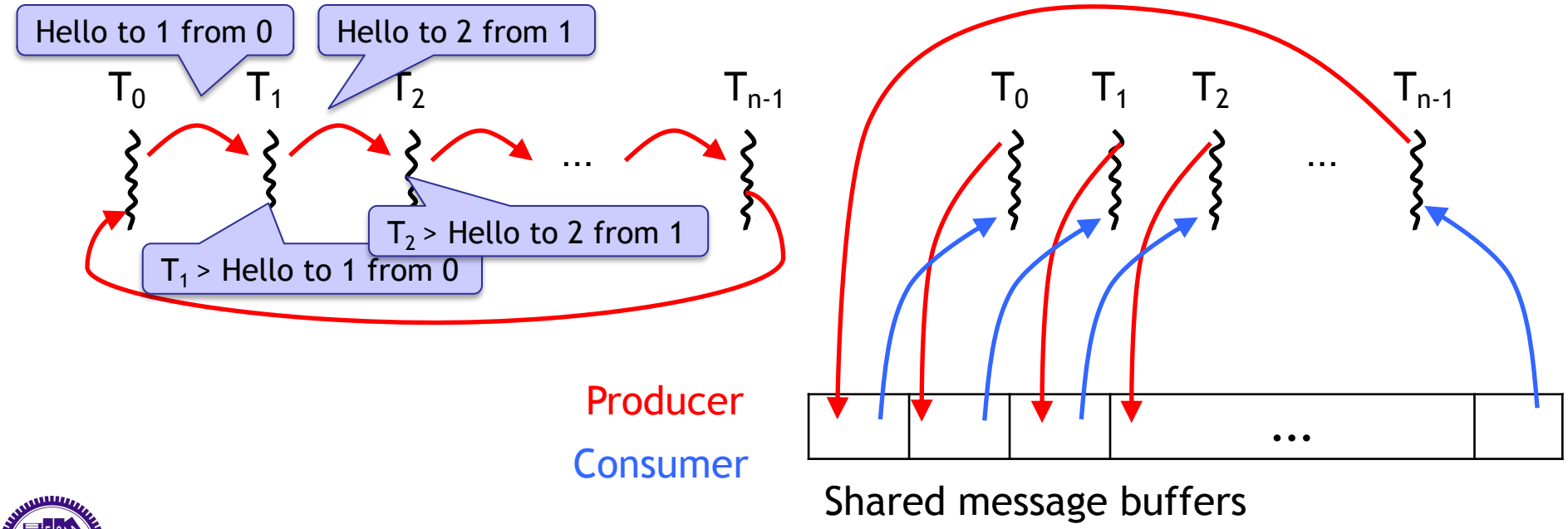
```
/* n and product_matrix are shared and initialized by the main
thread */
/* product_matrix is initialized to be the identity matrix  */
void* Thread_work(void* rank) {
   long my_rank = (long) rank;
   matrix_t my_mat = Allocate_matrix(n);
   Generate_matrix(my_mat);
   pthread_mutex_lock(&mutex);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

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Another Example: Producer-Consumer Problem

- Producers have information to send to the consumers
  - Each thread "sends a message" to another thread and "receives a message" from yet another thread



Producer-Consumer Problem Using Pthreads (1/2)

```
...
int thread_count;
char** messages;

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   ... // processing argv[]

   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
   messages = (char**) malloc(thread_count*sizeof(char*));
   for (thread = 0; thread < thread_count; thread++)
      messages[thread] = NULL;

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

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

   for (thread = 0; thread < thread_count; thread++)
      free(messages[thread]);
   free(messages);

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

Producer-Consumer Problem Using Pthreads (2/2)

```
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 = (char*) malloc(MSG_MAX*sizeof(char));

   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 */
```

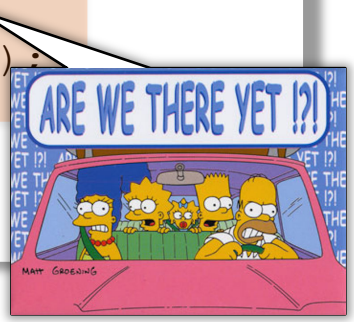
- Some of the messages are never received
  - E.g., thread 0, which is started first, will typically finish before thread *n*-1 has copied the message into the messages array

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Producer-Consumer Problem Using Busy-Wait

```
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 = (char*) malloc(MSG_MAX*sizeof(char));

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

   while (messages[my rank] == NULL);
   printf("Thread %ld > %s\n", my_rank, messages[my_rank]);

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



- Of course, this solution would have the same problems that any busy-waiting solution has, so we'd prefer a different approach

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Try to Use Mutexes (1/3)

```
. . .
messages[dest] = my_msg;
Notify thread dest that it can proceed;

Await notification from thread source
printf("Thread %ld > %s\n", my_rank, messages[my_rank]);
. . .
```

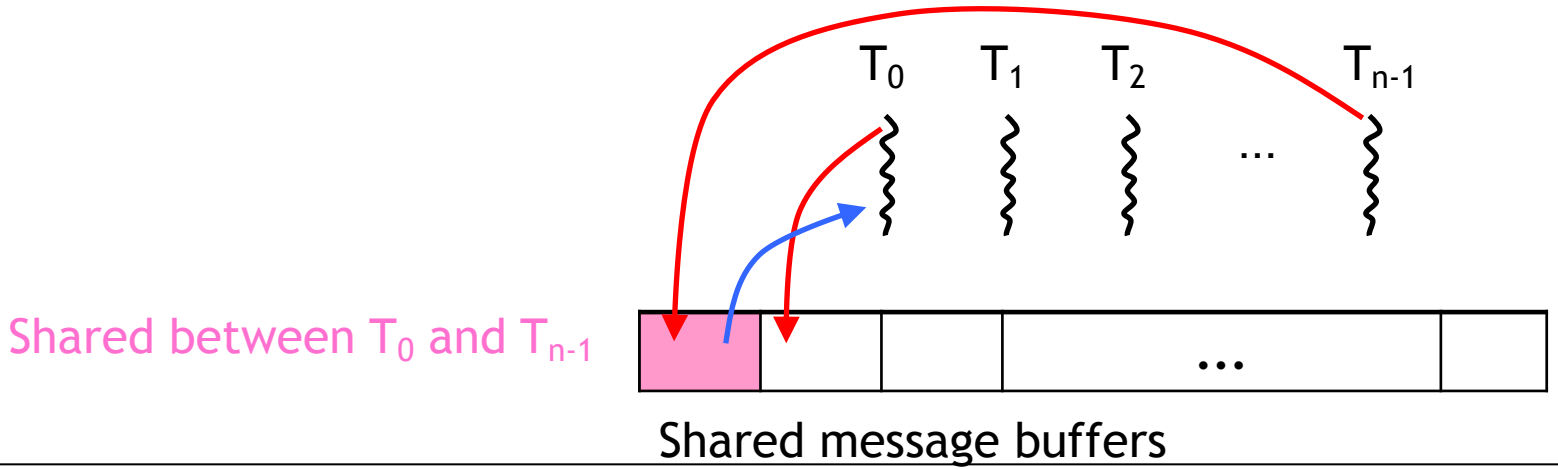
Producer

unlock

lock

Consumer

- Each buffer requires a lock
- We might try calling `pthread_mutex_unlock` to "notify" the thread with rank `dest`



Try to Use Mutexes (2/3)

- However, mutexes are initialized to be *unlocked*, so we'd need to add a call *before* initializing `messages[dest]` to lock the mutex

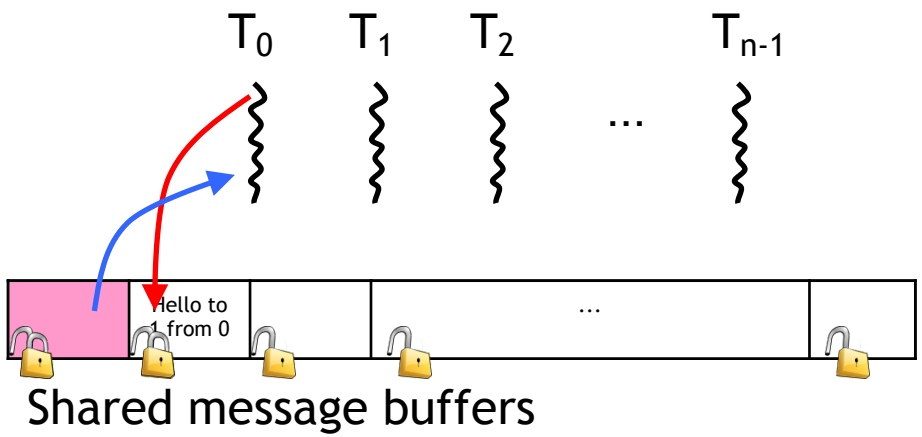
```
...
pthread_mutex_lock(mutex[dest]);
...
messages[dest] = my msg;
pthread_mutex_unlock(mutex[dest]);
...
pthread_mutex_lock(mutex[my rank]);
printf("Thread %ld > %s\n", my rank, messages[my rank]);
...
```

- This will be a problem since we don't know when the threads will reach the calls to `pthread_mutex_lock`

Try to Use Mutexes (3/3)

T_{n-1} ...
T_0 `pthread_mutex_lock(mutex[dest]);`
...
`messages[dest] = my msg;`
T_0 `pthread_mutex_unlock(mutex[dest]);`
...
T_0 `pthread_mutex_lock(mutex[my rank]);`
`printf("Thread %ld > %s\n", my rank, messages[my rank]);`
...

- T_0 gets so far ahead of T_n that it reaches the second call to `pthread_mutex_lock` before T_n reaches the first call to `pthread_mutex_lock`
- T_0 will acquire the lock and continue to dereferencing a null pointer

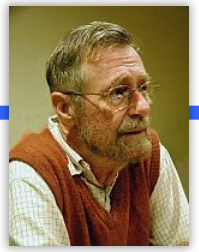


Shared message buffers

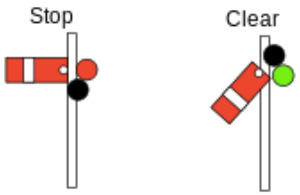
Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Semaphores



- Edsger Dijkstra - 1965
- A *semaphore* is a data structure consisting of a counter and a task descriptor queue
- Semaphores can be used to implement guards on the code that accesses shared data structures
- Semaphores have only two operations, *wait* and *release(post)* (originally called *P* and *V* by Dijkstra)
- The name is taken from the mechanical device that railroads use to control which train can use a track



Pthread Semaphores

- POSIX also provides a somewhat different means of controlling access to critical sections: **semaphores**
- Semaphores can be thought of as a special type of **unsigned int**, so they can take on the values 0, 1, 2, ...
- In most cases, we'll only be interested in using them when they take on the values 0 and 1
  - Called binary semaphores
  - 0: a locked mutex
  - 1: an unlocked mutex

`sem_wait` and `sem_post`

Right before entering a parking lot

- Before the critical section you want to protect, you place a call to the function `sem_wait`
  - A thread that executes `sem_wait` will block if the semaphore is 0
  - If the semaphore is nonzero, it will *decrement* the semaphore and proceed

Right after leaving a parking lot

- After executing the code in the critical section, a thread calls `sem_post`
  - It *increments* the semaphore, and a thread waiting in `sem_wait` can proceed

Semaphores vs Mutexes

- For our current purposes, the crucial difference between semaphores and mutexes is that there is no ownership associated with a semaphore
- The main thread can initialize all of the semaphores to 0—that is, “locked,” and
  - then any thread can execute a `sem_post` on any of the semaphores, and,
  - similarly, any thread can execute `sem_wait` on any of the semaphores

Syntax of `sem_*`

```
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 */);
```

- The second argument to `sem_init`:
  - If zero, this semaphore may only be shared between threads in the same process
  - If nonzero, the semaphore can be shared between processes

Producer-Consumer Problem Using Semaphores (1/2)

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

int thread_count;
char** messages;
sem_t* semaphores;

int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   ... // processing argv[]

   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
   messages = (char**) malloc(thread_count*sizeof(char*));
   semaphores = malloc(thread_count*sizeof(sem_t));
   for (thread = 0; thread < thread_count; thread++) {
      messages[thread] = NULL;
      sem_init(&semaphores[thread], 0, 0);
   }

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

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

   for (thread = 0; thread < thread_count; thread++) {
      free(messages[thread]);
      sem_destroy(&semaphores[thread]);
   }
   free(messages);
   free(semaphores);
   free(thread_handles);
   return 0;
}  /* main */
```

Producer-Consumer Problem Using Semaphores (2/2)

```
void *Send_msg(void* rank) {
   long my_rank = (long) rank;
   long dest = (my_rank + 1) % thread_count;
   char* my_msg = (char*) malloc(MSG_MAX*sizeof(char));

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

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

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

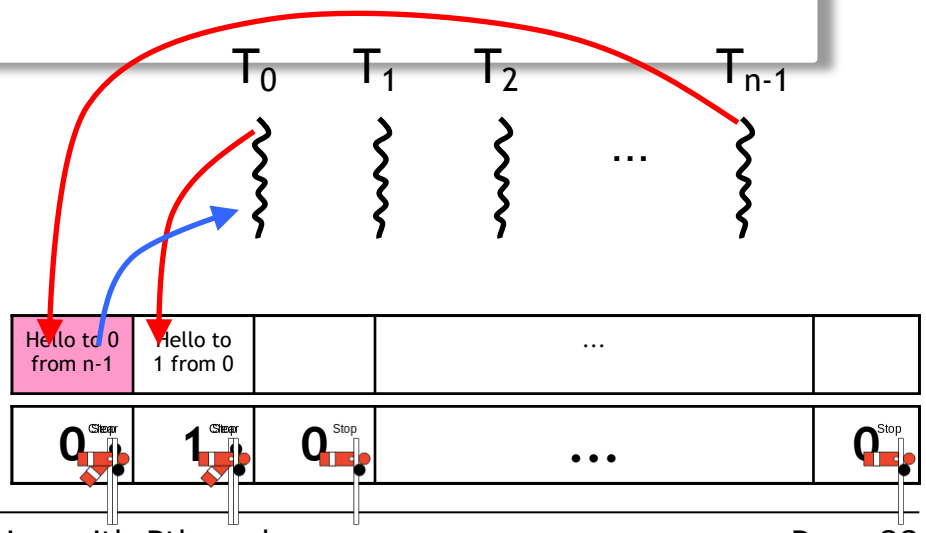
T_{n-1}

T_0

T_0 T_{n-1}

T_0

T_0



Producer-Consumer Synchronization

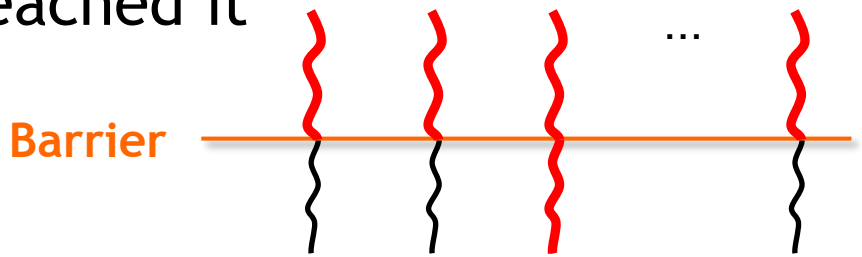
- The problem didn't involve a critical section
  - No *competition synchronization*
- Rather, thread `my_rank` couldn't proceed until thread source had finished creating the message
- This type of synchronization, when a thread can't proceed until another thread has taken some action, is sometimes called **producer-consumer synchronization**
  - *Cooperation synchronization*

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Barriers

- Let's take a look at another problem in shared-memory programming
  - Synchronizing the threads by making sure that they all are at the same point in a program
  - Such a point of synchronization is called a **barrier** because no thread can proceed beyond the barrier until all the threads have reached it



- Example
  - Timing some part of a multithreaded program
    - We'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, that is, the "slowest" thread

Example: Timing a Multithreaded Program

- Example
  - Timing some part of a multithreaded program
    - We'd like for all the threads to start the timed code at the same instant, and then report the time taken by the last thread to finish, that is, the "slowest" thread

```
/* 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;
```

Make sure that all of the threads will record `my_start` at approximately the same time

Another Use of Barriers: Debugging

- It can be very difficult to determine *where* an error is occurring in a parallel program
  - Tedious approach: we can, of course, have each thread print a message indicating which point it's reached in the program
  - Barrier approach:

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

Implementations of Barriers

- Many implementations of Pthreads don't provide barriers, so if our code is to be portable, we need to develop our own implementation
  - Using busy-waiting and a mutex
  - Using semaphores
  - Using a condition variable

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

A Barrier Using Busy-Waiting and a Mutex

- Use a shared counter protected by the mutex
  - When the counter indicates that every thread has entered the critical section, threads can leave a busy-wait loop

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

A Barrier Using Busy-Waiting and a Mutex: Issues

- Of course, this implementation will have the same problems that our other busy-wait codes had
  - Wasting CPU cycles when threads are in the busy-wait loop
- What happens if we want to implement a second barrier and we try to reuse the counter?

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

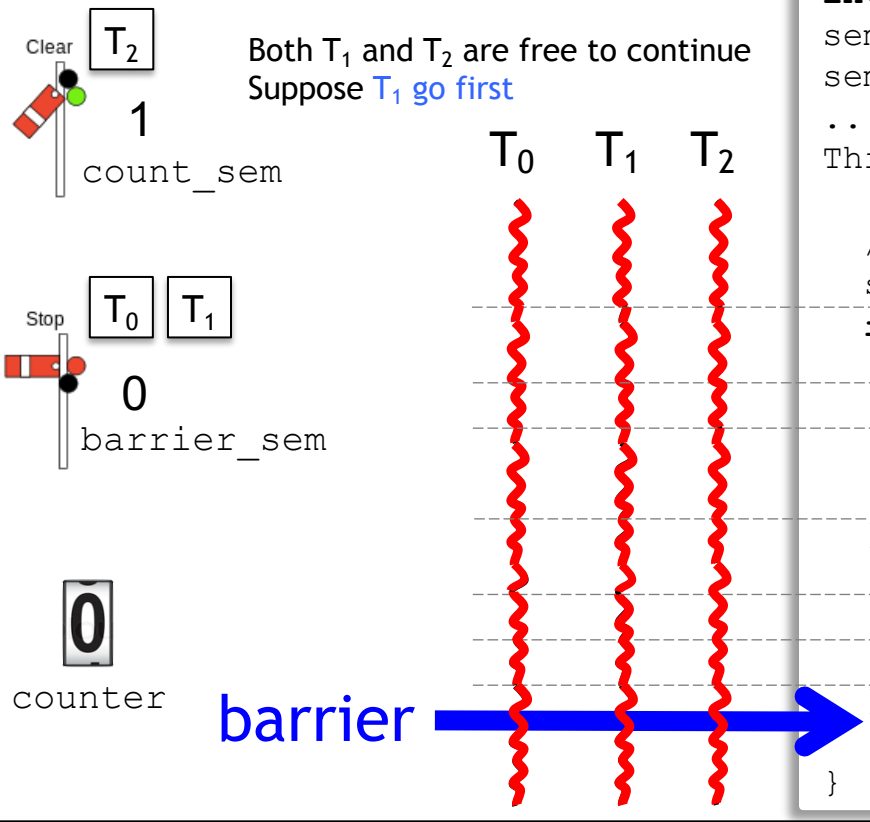
- We must "somehow" reset `counter` after the first barrier completes
  - If the last thread to enter the loop tries to reset it,
    - some thread in the busy-wait may never see the fact that `counter == thread_count`
  - If some thread tries to reset `counter` after the barrier,
    - some other thread may enter the second barrier before `counter` is reset and its increment to the counter will be lost

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

A Barrier Using Semaphores (1/2)

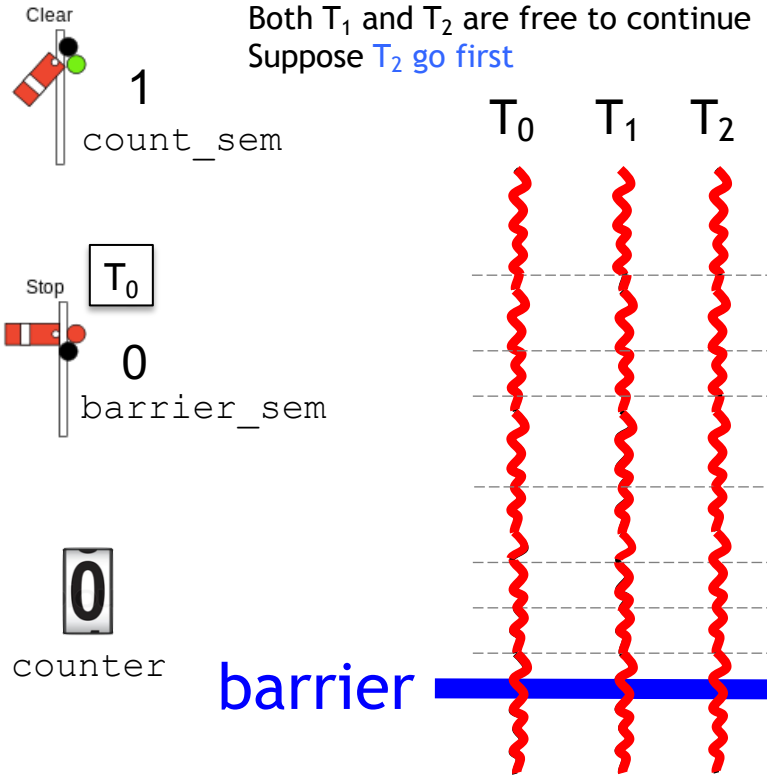
- Use two semaphores
  - `count_sem` protects the counter
    - Initialized to 1 (clear)
  - `barrier_sem` blocks threads that have entered the barrier
    - Initialized to 0 (stop)




```
/* Shared variables */
int counter;        /* Initialize to 0 */
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier sem; /* Initialize to 0 */
...
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);
  }
  ...
}
```

A Barrier Using Semaphores (2/2)

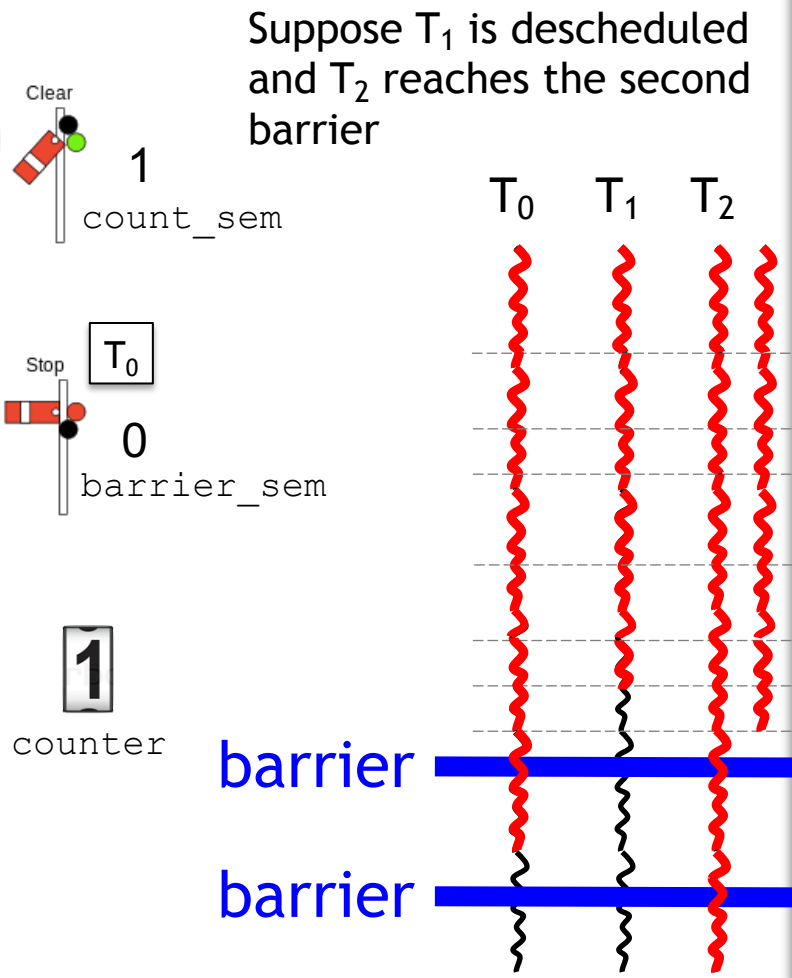
- It doesn't matter if T_2 races ahead and executes multiple calls to `sem_post` before a thread can be unblocked from `sem_wait(&barrier_sem)`




```
/* Shared variables */
int counter;        /* Initialize to 0 */
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier sem; /* Initialize to 0 */
...
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);
  }
  ...
}
```

Barriers Using Semaphores

- `counter` and `count_sem` can be reused
- However, reusing `barrier_sem` results in a race condition




```
/* Shared variables */
int counter;        /* Initialize to 0 */
sem_t count_sem;    /* Initialize to 1 */
sem_t barrier sem; /* Initialize to 0 */
...
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);
  }
  ...

  ...
}
```

Outline

- Introduction
- Processes, Threads, and Pthreads
- Hello World!
- Matrix-Vector Multiplication
- Calculation of Pi
  - Using Busy-Wait
  - Using a Mutex
- Producer-Consumer Synchronization
  - Using Busy-Wait
  - Using Mutexes
  - Usng Semaphores
- Barriers
  - Using Busy-Wait and a Mutex
  - Using Semaphores
  - Using a Condition Variable

Condition Variables

- A somewhat better approach to creating a barrier in Pthreads is provided by *condition variables*
- A **condition variable** is a data object that allows a thread to suspend execution until a certain event or *condition* occurs
  - When the event or condition occurs, another thread can *signal* the thread to "wake up"
  - A condition variable is *always* associated with a mutex

Condition Variables

- Typically, condition variables are used in constructs similar to this pseudo code:

```
lock mutex;
if condition has occurred
   signal thread(s);
else {
   unlock the mutex and block;
   /* when thread is unblocked, mutex is relocked */
}
unlock mutex;
```

- Condition variables in Pthreads have type `pthread_cond_t`

Signaling and Blocking Threads

```
int pthread_cond_signal(pthread_cond_t*  cond_var_p /* in/out */);
```

- Unblock *one* of the blocked threads

```
int pthread_cond_broadcast(pthread_cond_t* cond_var_p /* in/out */);
```

- Unblock *all* of the blocked threads

```
int pthread_cond_wait(
      pthread_cond_t*    cond_var_p   /* in/out */,
      pthread_mutex_t*   mutex_p      /* in/out */);
```

```
pthread_mutex_unlock(&mutex_p);
wait_on_signal(&cond_var_p);
pthread_mutex_lock(&mutex_p);
```

- Unlock the mutex referred to by `mutex_p` and cause the executing thread to block until it is unblocked by another thread's call to `pthread_cond_signal` or `pthread_cond_broadcast`
- When the thread is unblocked, it reacquires the mutex

A Barrier Using a Condition Variable

```
/* 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 - 1) {
      counter = 0;
      pthread_cond_broadcast(&cond_var);
   } else {
      while (pthread_cond_wait(&cond_var, &mutex) != 0);
   }
   pthread_mutex_unlock(&mutex);
   ...
}
```

- The call to pthread_cond_wait is usually placed in a **while** loop
  - If the thread is unblocked by some event other than a call to `pthread_cond_signal` or `pthread_cond_broadcast`, then the return value of `pthread_cond_wait` will be nonzero, and the unblocked thread will call `pthread_cond_wait` again

Initialization and Destroy

- Like mutexes and semaphores, condition variables should be initialized and destroyed

```
int pthread_cond_init(
      pthread_cond_t*              cond_p       /* out */,
      const pthread_condattr_t*    cond_attr_p  /* in  */);
```

```
int pthread_cond_destroy(pthread_cond_t*  cond_p  /* in/out */);
```

Pthreads Barrier

- Open Group, the standards group that is continuing to develop the POSIX standard, does define a barrier interface for Pthreads
  - However, as we noted earlier, it is not universally available

```
int pthread_barrier_init(
      pthread_barrier_t*             barrier_p  /* out */,
      const pthread_barrierattr_t*   attr_p     /* in  */,
      unsigned                       count      /* in  */);
```

```
int pthread_barrier_wait(
      pthread_barrier_t*  barrier_p  /* in/out */);
```

```
int pthread_barrier_destroy(
      pthread_barrier_t*  barrier_p  /* in/out */);
```