Multi Threading

Operating System Lab
Spring 2015

Roya Razmnoush
CE & IT Department, Amirkabir University of Technology

Introduction

Why threads?!

- Code is often written in a *serialized* (or sequential) fashion. What is meant by the term serialized? code is executed sequentially, one after the next in a monolithic fashion, without regard to possibly more available processors the program could exploit. Often, there are potential parts of a program where performance can be improved through the use of threads.

- With increasing popularity of machines with symmetric multiprocessing (largely due in part to the rise of multicore processors), programming with threads is a valuable skill set worth learning.

Thread concept

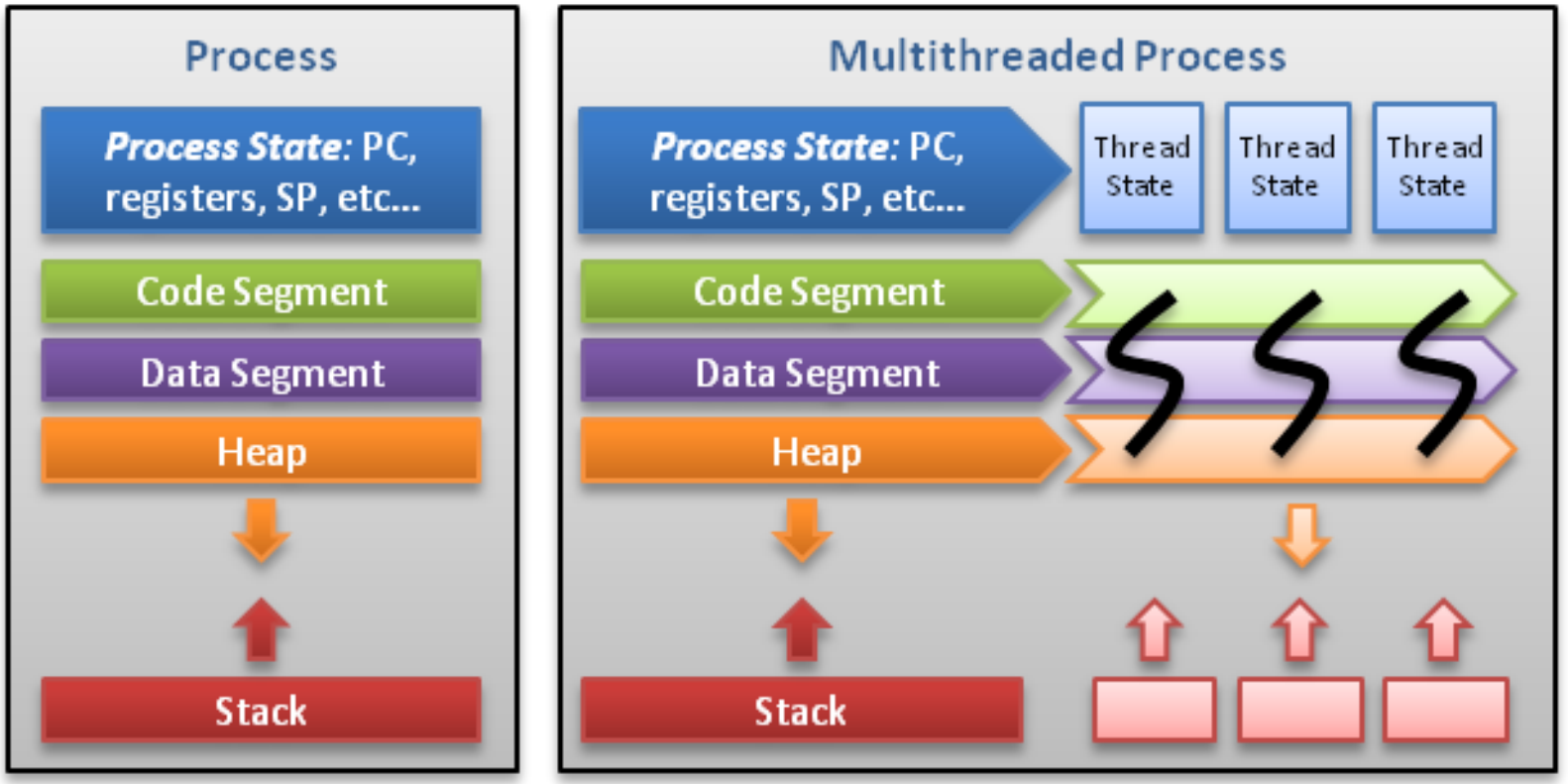
Thread vs. Process

Process vs. Thread

- A computer program becomes a **process** when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a processor or a set of processors. A process description in memory contains vital information such as the program counter which keeps track of the current position in the program (i.e. which instruction is currently being executed), registers, variable stores, file handles, signals, and so forth.

- A **thread** is a sequence of such instructions within a program that can be executed independently of other code.

Process vs. Thread



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

Process vs. Thread

- Threads are within the *same process address space*, thus, much of the information present in the memory description of the process can be shared across threads.

- Some information cannot be replicated, such as the stack (stack pointer to a different memory area per thread), registers and thread-specific data. This information sufficies to allow threads to be scheduled independently of the program's main thread and possibly one or more other threads within the program.

Threads need OS support

Explicit operating system support is required to run multithreaded programs. Fortunately, most modern operating systems support threads such as Linux (via NPTL), BSD variants, Mac OS X, Windows, Solaris, AIX, HP-UX, etc. Operating systems may use different mechanisms to implement multithreading support.

Mutual exclusion

mutex

Protecting Shared Resources

- Threads may operate on disparate data, but often threads may have to touch the same data. It is unsafe to allow concurrent access to such data or resources without some *mechanism that defines a protocol for safe access*! Threads must be explicitly instructed to block when other threads may be potentially accessing the same resources.

Mutual exclusion

- Mutual exclusion is the method of *serializing access* to shared resources. You do not want a thread to be modifying a variable that is already in the process of being modified by another thread! Another scenario is a dirty read where the value is in the process of being updated and another thread reads an old value.

mutex

- Mutual exclusion allows the programmer to create a defined protocol for serializing access to shared data or resources.
- Logically, a **mutex** is a lock that one can virtually attach to some resource.
- If a thread wishes to modify or read a value from a shared resource, the thread must first gain the lock. Once it has the lock it may do what it wants with the shared resource without concerns of other threads accessing the shared resource because other threads will have to wait.
- Once the thread finishes using the shared resource, it unlocks the mutex, which allows other threads to access the resource. This is a protocol that serializes access to the shared resource.

mutex

- As an analogy, you can think of a mutex as a safe with only one key (for a standard mutex case), and the resource it is protecting lies within the safe. Only one person can have the key to the chest at any time, therefore, is the only person allowed to look or modify the contents of the chest at the time it holds the key.

- The code between the lock and unlock calls to the mutex, is referred to as a **critical section**. Minimizing time spent in the critical section allows for greater concurrency because it potentially reduces the amount of time other threads must wait to gain the lock. Therefore, it is important for a thread programmer to minimize critical sections if possible.

Mutex types

There are different types of locks other than the standard simple blocking kind.

- **Recursive**: allows a thread holding the lock to acquire the same lock again which may be necessary for recursive algorithms.
- **Queuing**: allows for *fairness* in lock acquisition by providing FIFO ordering to the arrival of lock requests. Such mutexes may be slower due to increased overhead and the possibility of having to wake threads next in line that may be sleeping.
- **Reader/Writer**: allows for multiple readers to acquire the lock simultaneously. If existing readers have the lock, a writer request on the lock will block until all readers have given up the lock. This can lead to writer starvation.
- **Scoped**.

Depending upon the thread library or interface being used, only a subset of the additional types of locks may be available. POSIX pthreads allows recursive and reader/writer style locks.

Potential Traps with Mutexes

- An important problem associated with mutexes is the possibility of **deadlock**. A program can deadlock if two (or more) threads have stopped execution or are spinning permanently. For example, a simple deadlock situation: thread 1 locks lock A, thread 2 locks lock B, thread 1 wants lock B and thread 2 wants lock A. Instant deadlock. You can prevent this from happening by making sure threads acquire locks in an agreed order (i.e. preservation of **lock ordering**). Deadlock can also happen if threads do not unlock mutexes properly.
- A **race condition** is when non-deterministic behavior results from threads accessing shared data or resources without following a defined synchronization protocol for serializing such access. This can result in erroneous outcomes that cause failure or inconsistent behavior making race conditions particularly difficult to debug. In addition to incorrectly synchronized access to shared resources, library calls outside of your program's control are common culprits. Make sure you take steps within your program to enforce serial access to shared file descriptors and other external resources.
- Another problem with mutexes is that contention for a mutex can lead to **priority inversion**. A higher priority thread can wait behind a lower priority thread if the lower priority thread holds a lock for which the higher priority thread is waiting. This can be eliminated/reduced by limiting the number of shared mutexes between different priority threads.

race condition

- When ***two or more*** concurrently running threads access a ***shared*** data item and the final result depends on ***the order of execution***, we have a ***race condition***. Let us take a look at an example. Suppose we have two threads **A** and **B** as shown below. Thread **A** increases the shared variable Count by 1 and thread **B** decreases Count by 1. Since Count is shared by two threads, we know it should be protected by a mutex lock or a binary semaphore.

Synchronization

Join, conditional variables, barriers, spin locks, semaphores

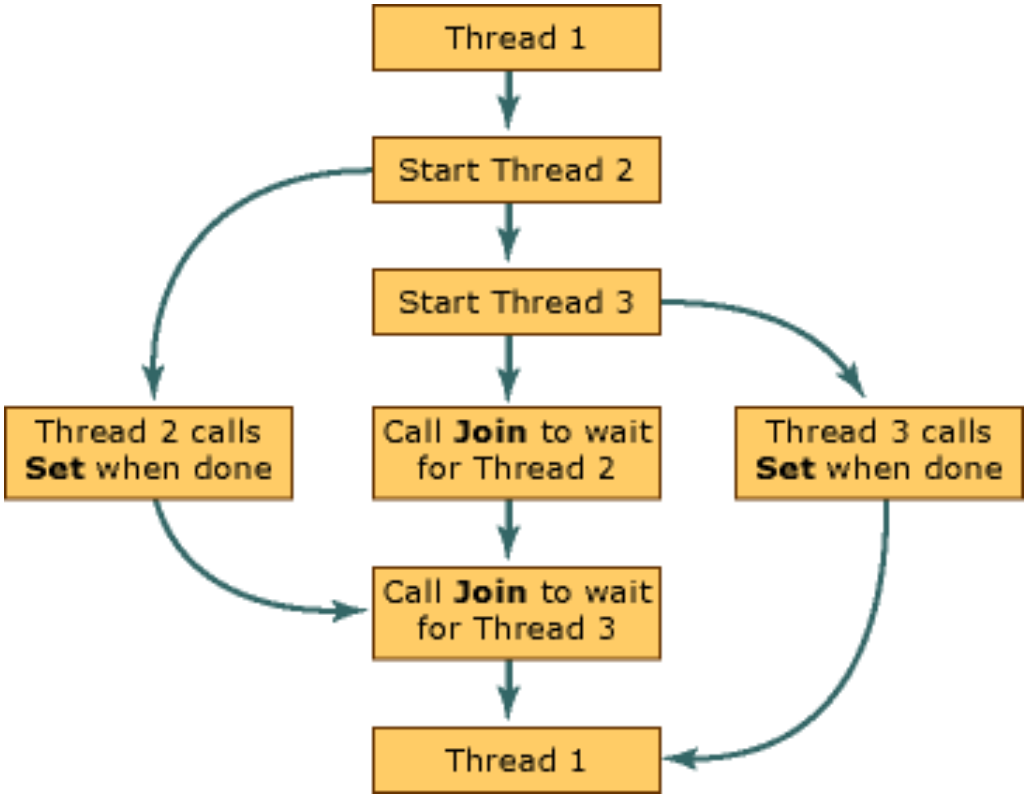
Thread Synchronization Primitives

- mutexes are one way of synchronizing access to shared resources. There are other mechanisms available for not only coordinating access to resources but synchronizing threads.

Join

- A thread join is a protocol to allow the programmer to *collect* all relevant threads at a logical synchronization point. For example, in fork-join parallelism, threads are spawned to tackle parallel tasks and then join back up to the main thread after completing their respective tasks (thus performing an implicit barrier at the join point). Note that a thread that executes a join has terminated execution of their respective thread function.

Join

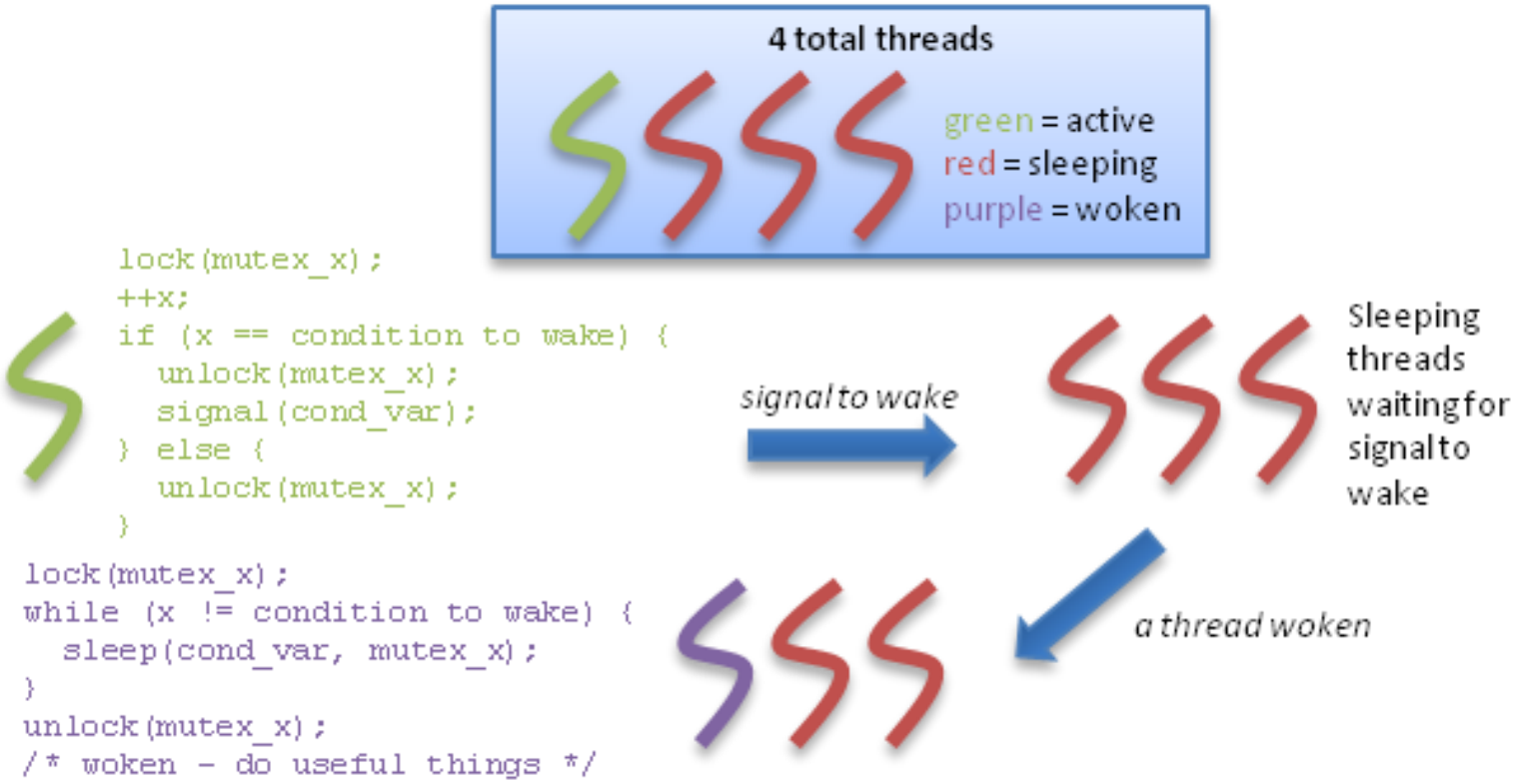


The Thread.Join method is useful for determining if a thread has completed before starting another task. The Join method waits a specified amount of time for a thread to end. If the thread ends before the timeout, Join returns True; otherwise it returns False

Condition Variables

- Condition variables allow threads to synchronize to a value of a shared resource. Typically, condition variables are used as a notification system between threads.
- For example, you could have a counter that once reaching a certain count, you would like for a thread to activate. The thread (or threads) that activates once the counter reaches the limit would *wait* on the condition variable. Active threads *signal* on this condition variable to notify other threads waiting/sleeping on this condition variable; thus causing a waiting thread to wake. You can also use a *broadcast* mechanism if you want to signal *all* threads waiting on the condition variable to wakeup.
- When waiting on condition variables, the wait should be inside a loop, not in a simple if statement because of **spurious wakeups**. You are not guaranteed that if a thread wakes up, it is the result of a signal or a broadcast call.

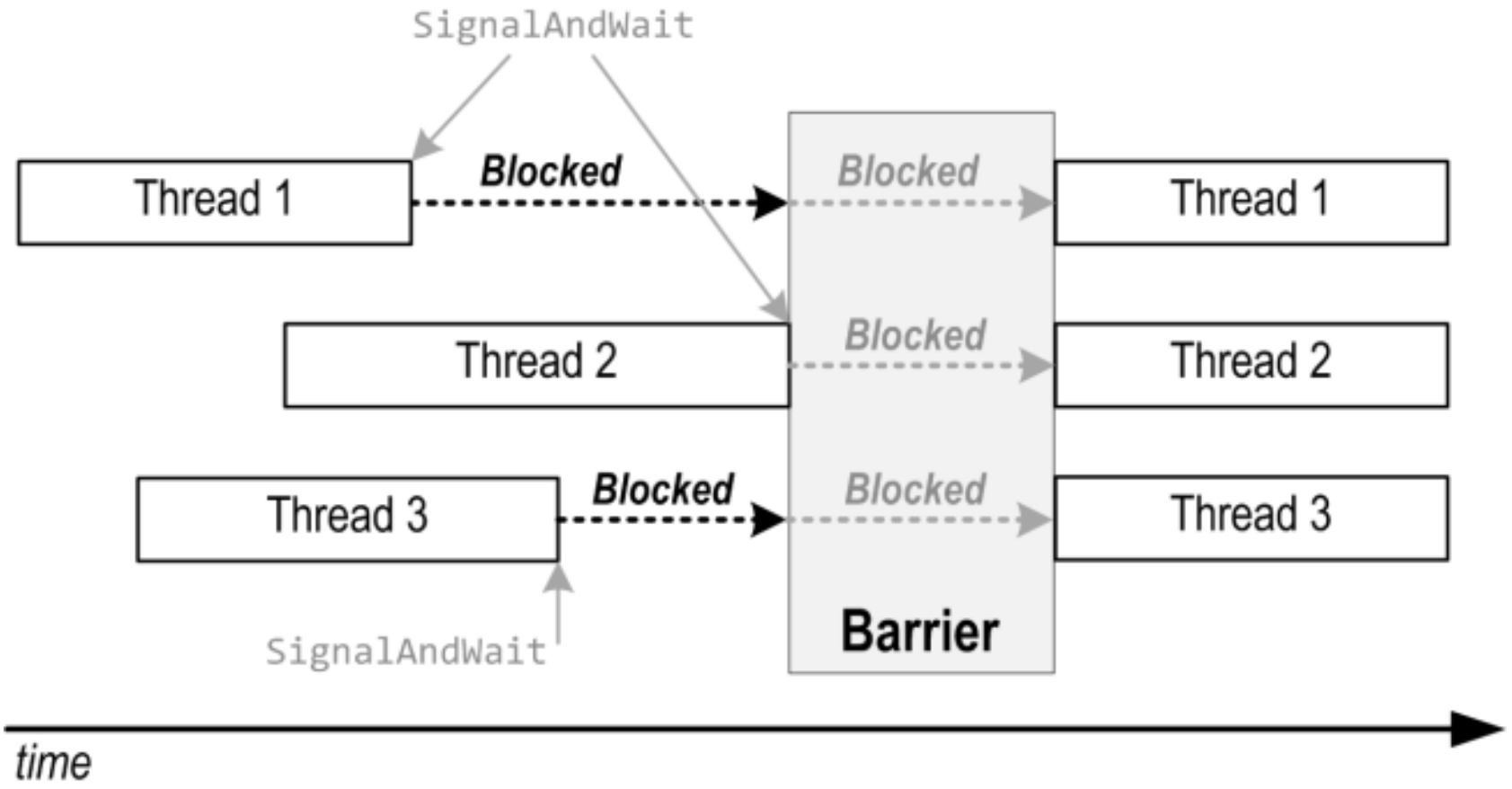
Condition Variables



Barriers

- Barriers are a method to synchronize a set of threads at some point in time by having all participating threads in the barrier wait until all threads have called the said barrier function. This, in essence, blocks all threads participating in the barrier until the slowest participating thread reaches the barrier call.

Barriers


SignalAndWait
Thread 1
Blocked
Blocked
Thread 1
Thread 2
Blocked
Thread 2
Thread 3
Blocked
Blocked
Thread 3
SignalAndWait
Barrier
time

Spin locks

- Spinlocks are locks which *spin* on mutexes. Spinning refers to continuously polling until a condition has been met. In the case of spinlocks, if a thread cannot obtain the mutex, it will keep polling the lock until it is free. The advantage of a spinlock is that the thread is kept active and does not enter a sleep-wait for a mutex to become available, thus can perform better in certain cases than typical blocking-sleep-wait style mutexes. Mutexes which are heavily contended are poor candidates for spinlocks.

Semaphores

- Semaphores are another type of synchronization primitive that come in two flavors: binary and counting. Binary semaphores act much like simple mutexes, while counting semaphores can behave as *recursive mutexes*. Counting semaphores can be initialized to any arbitrary value which should depend on how many resources you have available for that particular shared data. Many threads can obtain the lock simultaneously until the limit is reached. This is referred to as *lock depth*.
- Semaphores are more common in multiprocess programming (i.e. it's usually used as a synch primitive between processes).

Semaphore

- Semaphores are a synchronization primitive.
- To obtain a shared resource:
  - Test the semaphore that controls the resource.
  - If the value is positive the process can use the resource. The process decrements the value by 1.
  - If the value is 0, the process goes to sleep until the value is greater than 0.

pthread library

Using pthread library to implement mutual exclusion

POSIX standards

Portable Operating System Interface for UNIX

- POSIX is a computer industry operating system standard that every major version of UNIX complied with. In other words, if your operating system was POSIX-compliant, it was UNIX.

- The POSIX family of related standards is being developed by PASC (IEEE's Portable Application Standards Committee, www.pasc.org). A comprehensive FAQ on POSIX, including many links, appears at
  www.opengroup.org/austin/papers/posix_faq.html

- POSIX stands for Portable Operating System Interface, and its aim is to provide a standard level of UNIX compatibility. If any element running on Linux or any other UNIX version is POSIX compliant, it will run without problems on any flavor of UNIX. The POSIX standard is not just limited to Linux and UNIX operating systems; most versions of Windows (up to and including Vista) are POSIX compliant.

✓ Every file on every POSIX-compliant file system needs an inode to store its administrative information.

✓ POSIX standard 1003.2 describes shell functionality. The Bourne Again Shell pro vides the features that match the requirements of this POSIX standard. Efforts are under way to make the Bourne Again Shell fully comply with the POSIX standard. In the meantime, if you invoke bash with the ––posix option, the behavior of the Bourne Again Shell will more closely match the POSIX requirements.

POSIX pthreads

- POSIX pthreads is a particular threading implementation, The pthread library can be found on almost any modern POSIX-compliant OS (and even under Windows)

- Pthreads concepts such as thread scheduling classes, thread-specific data, thread canceling, handling signals and reader/writer locks are not covered here.

Preliminaries

- Before we begin, there are a few required steps you need to take before starting any pthreads coding:
- Add #include <pthread.h> to your source file(s).
- If you are using gcc, you can simply specify -pthread which will set all proper defines and link-time libraries. On other compilers, you may have to define _REENTRANT and link against -lpthread.
- *Optional:* some compilers may require defining _POSIX_PTHREAD_SEMANTICS for certain function calls like sigwait().

Creating pthreads

- A pthread is represented by the type pthread_t. To create a thread, the following function is available:

```
1  int pthread_create(pthread_t *thread, pthread_attr_t *attr,
2                     void *(*start_routine)(void *), void *arg);
```

- Let's digest the arguments required for pthread_create():

1) pthread_t *thread: the actual thread object that contains pthread id
2) pthread_attr_t *attr: attributes to apply to this thread
3) void *(*start_routine)(void *): the function this thread executes
4) void *arg: arguments to pass to thread function above

Two other important thread functions

- pthread_exit() terminates the thread and provides the pointer *value_ptr available to any pthread_join() call.
- pthread_join() suspends the calling thread to wait for successful termination of the thread specified as the first argument pthread_t thread with an optional *value_ptr data passed from the terminating thread's call to pthread_exit().

```
1  void pthread_exit(void *value_ptr);
2  int pthread_join(pthread_t thread, void **value_ptr);
3
4  /* ignore me: needed so underscores above do not get clipped off */
```

Example1

- This program creates NUM_THREADS threads and prints their respective user-assigned thread id. The first thing to notice is the call to pthread_create() in the main function. The syntax of the third and fourth argument are particularly important. Notice that the thr_func is the name of the thread function, while the fourth argument is the argument passed to said function. Here we are passing a thread function argument that we created as a thread_data_t struct. Of course, you can pass simple data types as pointers if that is all that is needed, or NULL if no arguments are required. However, it is good practice to be able to pass arguments of arbitrary type and size, and is thus illustrated for this purpose.

Example1

- A few things to mention:

- Make sure you check the return values for all important functions.
- The second argument to pthread_create() is NULL indicating to create a thread with default attributes. The defaults vary depend upon the system and pthread implementation.
- Notice that we have broken apart the pthread_join() from the pthread_create(). Why is it that you should not integrate the pthread_join() in to the thread creation loop?
- Although not explicitly required to call pthread_exit() at the end of the thread function, it is good practice to do so, as you may have the need to return some arbitrary data back to the caller via pthread_join().

Pthread Attributes

- Threads can be assigned various thread attributes at the time of thread creation. This is controlled through the second argument to pthread_create(). You must first pass the pthread_attr_t variable through:

```
1  int pthread_attr_init(pthread_attr_t *attr);
2
3  /* ignore me: needed so underscores above do not get clipped off */
```

Pthread Attributes

- Some attributes that can be set are:

```
1  int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
2  int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize);
3  int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched);
4  int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
5  int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
6  int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
7  int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
8  int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
9
10 /* ignore me: needed so underscores above do not get clipped off */
```

- Attributes can be retrieved via complimentary get functions. Consult the man pages for the effect of each of these attributes

Pthread mutexes

- pthread mutexes are created through the following function:

```
1 | int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
2 |
3 | /* ignore me: needed so underscores above do not get clipped off */
```

- The pthread_mutex_init() function requires a pthread_mutex_t variable to operate on as the first argument. Attributes for the mutex can be given through the second parameter. To specify default attributes, pass NULL as the second parameter. Alternatively, mutexes can be initialized to default values through a convenient macro rather than a function call:

```
1 | pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Here a mutex object named lock is initialized to the default pthread mutex values.

Pthread mutexes

- To perform mutex locking and unlocking, the pthreads provides the following functions:

```
1  int pthread_mutex_lock(pthread_mutex_t *mutex);
2  int pthread_mutex_trylock(pthread_mutex_t *mutex);
3  int pthread_mutex_unlock(pthread_mutex_t *mutex);
4
5  /* ignore me: needed so underscores above do not get clipped off */
```

- Each of these calls requires a reference to the mutex object. The difference between the lock and trylock calls is that lock is blocking and trylock is non-blocking and will return immediately even if gaining the mutex lock has failed due to it already being held/locked. It is absolutely essential to check the return value of the trylock call to determine if the mutex has been successfully acquired or not. If it has not, then the error code EBUSY will be returned.

Example2

In the example 2 code, we add some shared data called shared_x and ensure serialized access to this variable through a mutex named lock_x. Within the thr_func() we call pthread_mutex_lock() before reading or modifying the shared data. Note that we continue to maintain the lock even through the printf() function call as releasing the lock before this and printing can lead to inconsistent results in the output. Recall that the code in-between the lock and unlock calls is called a critical section. Critical sections should be minimized for increased concurrency.

Miscellaneous

- Here are some suggestions and issues you should consider when using pthreads:

1) You should check all return values for important pthread function calls!
2) Sometimes it is desirable for a thread not to terminate (e.g., a server with a worker thread pool). This can be solved by placing the thread code in an infinite loop and using condition variables. Of course, there needs to be some terminating condition(s) to the infinite loop (i.e., break when it is deemed necessary).

- Additional useful pthread calls:

1) pthread_kill() can be used to deliver signals to specific threads.
2) pthread_self() returns a handle on the calling thread.
3) pthread_equal() compares for equality between two pthread ids
4) pthread_once() can be used to ensure that an initializing function within a thread is only run once.
5) There are many more useful functions in the pthread library.