CSE 3320
Operating Systems
POSIX Threads Programming

Jia Rao

Department of Computer Science and Engineering

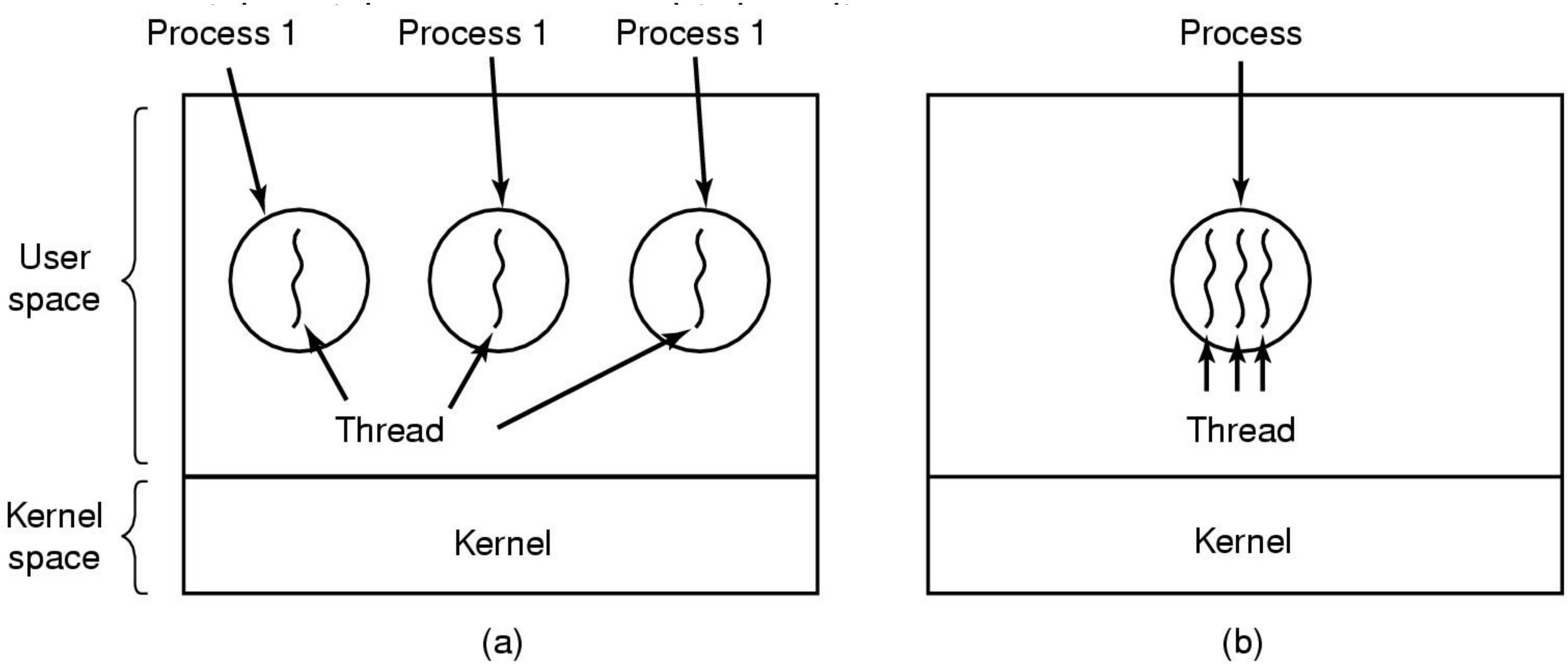
http://ranger.uta.edu/~jrao

Recap of Previous Classes

- Processes and threads
- The thread model
  - User-level thread
  - Kernel-level thread
- Mutual exclusion and critical regions
- Semaphores
- Mutexes
- Barrier

The Thread Model

- Process: for resource grouping and execution
- Thread: a finer-granularity entity for execution and parallelism



(a) Three processes each with one thread, but different address spaces

(b) One process with three threads, sharing the address space

The Thread Model (2)

- Because threads within the same process share resources
  - Changes made by one thread to shared system resources (closing a file) will be seen by all other threads
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory location is possible, and therefore requires explicit synchronization by the programmer!

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Items shared by all threads in a process   Items private to each thread

Pthreads Overview

- What are Pthreads?
  - An IEEE standardized thread programming interface (IEEE POSIX 1003.1c)
  - POSIX (Portable Operating System Interface) threads
  - Defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library

To software developer:
a thread is a “procedure” that runs independently from its main program

Why Pthreads ?

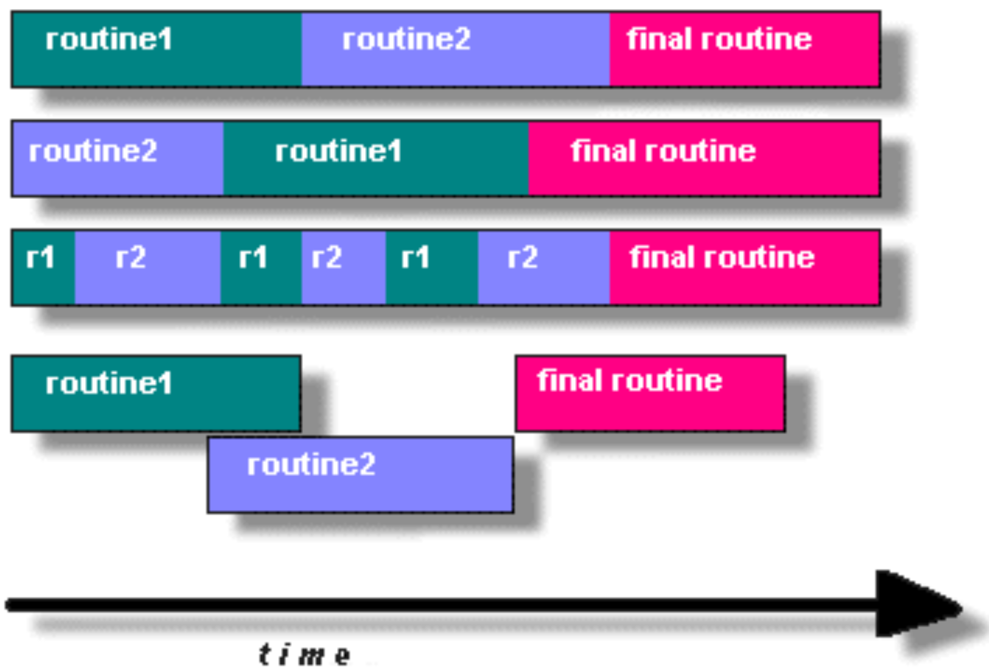
- Performance!
  - Lightweight
  - Communication
  - Overlapping CPU work with I/O; fine granularity of concurrency
  - Priority/real-time scheduling
  - Asynchronous event handling

50000 process/thread creation ! Timed in sec

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
IBM 1.9 GHz POWER5 p5-575	50.66	3.32	42.75	1.13	0.54	0.75
INTEL 2.4 GHz Xeon	23.81	3.12	8.97	1.70	0.53	0.30
INTEL 1.4 GHz Itanium 2	23.61	0.12	3.42	2.10	0.04	0.01

Design Threaded Programs

- A program must be able to be organized into discrete, independent tasks which can execute concurrently
  - E.g.: routine1 and routine2 can be interchanged, interleaved, and/or overlapped in real time
  - Thread-safeness: race conditions



The Pthreads API

- The API is defined in the ANSI/IEEE POSIX 1003.1 – 1995
  - Naming conventions: all identifiers in the library begins with pthread_
  - Three major classes of subroutines
    - Thread management, mutexes, condition variables

Routine Prefix	Functional Group
pthread_	Threads themselves and miscellaneous subroutines
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects.
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data keys

Compiling Pthreads Programs

Platform	Compiler Command	Description
IBM AIX	xlc_r / cc_r	C (ANSI / non-ANSI)
	xlC_r	C++
	xlf_r -qnosave xlf90_r -qnosave	Fortran - using IBM's Pthreads API (non-portable)
INTEL LINUX	icc -pthread	C
	icpc -pthread	C++
COMPAQ Tru64	cc -pthread	C
	cxx -pthread	C++
All Above Platforms	gcc -lpthread	GNU C
	g++ -lpthread	GNU C++
	guidec -pthread	KAI C (if installed)
	KCC -pthread	KAI C++ (if installed)

Thread Management – Creation and Termination

pthread_create (threadid, attr, start_routine, arg)

* creates a thread and makes it executable; *arg* must be passed by reference as a pointer cast of type void

pthread_exit (status)

• If main() finishes before the threads it has created, and exits with the pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes

pthread_attr_init (attr)

•Initialize the thread attribute object (other routines can then query/set attributes)

pthread_attr_destroy (attr)

• destroy the thread attribute object

Initially, your main() program comprises a single, default thread.

Pthread Argument Passing – Single Argument

Example 1 - Thread Argument Passing

This code fragment demonstrates how to pass a simple integer to each thread. The calling thread uses a unique data structure for each thread, insuring that each thread's argument remains intact throughout the program.

```
int *taskids[NUM_THREADS];

for(t=0; t<NUM_THREADS; t++)
{
   taskids[t] = (int *) malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) taskids[t]);
   ...
}
```

Source Output

Pthread Argument Passing – Multiple Arguments

Example 2 - Thread Argument Passing

This example shows how to setup/pass multiple arguments via a structure. Each thread receives a unique instance of the structure.

```
struct thread_data{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
   struct thread_data *my_data;
   ...
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   ...
}

int main (int argc, char *argv[])
{
   ...
   thread_data_array[t].thread_id = t;
   thread_data_array[t].sum = sum;
   thread_data_array[t].message = messages[t];
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &thread_data_array[t]);
   ...
}
```

Source Output

Pthread Argument Passing – Incorrect Example

Example 3 - Thread Argument Passing (Incorrect)

This example performs argument passing incorrectly. The loop which creates threads modifies the contents of the address passed as an argument, possibly before the created threads can access it.

```
int rc, t;

for(t=0; t<NUM_THREADS; t++)
{
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL, PrintHello,
        (void *) &t);
   ...
}
```

Source Output

Thread Management – Joining and Detaching

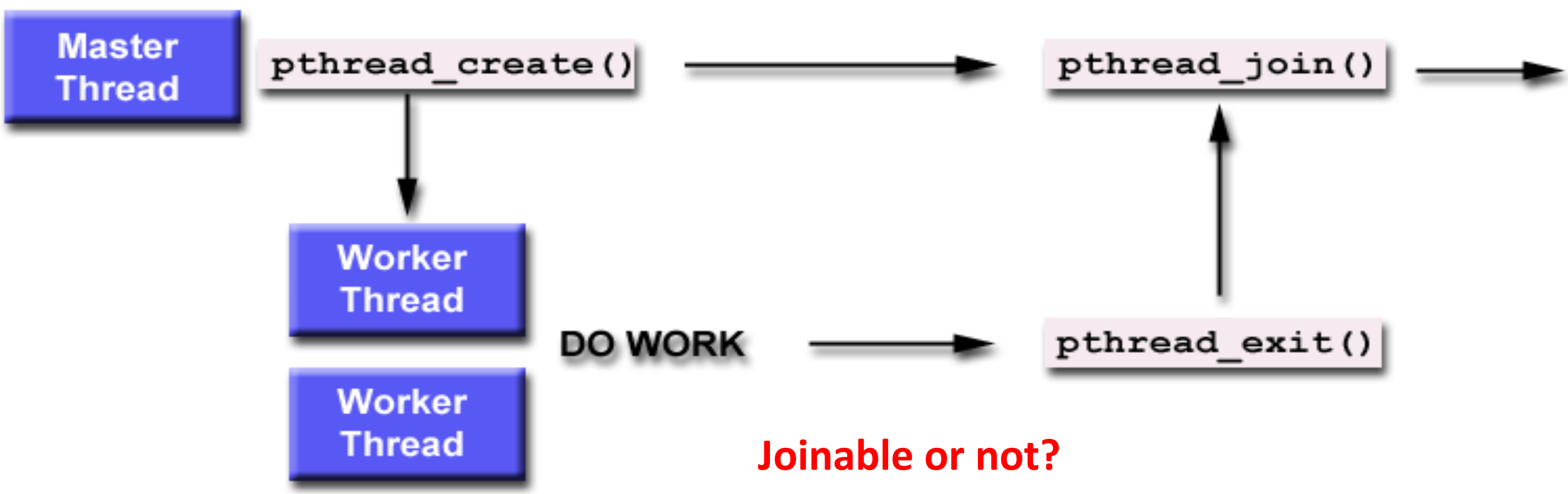
[pthread_join](#) (threadid, status)

[pthread_detech](#)(threadid, status)

[pthread_attr_setdetachstate](#)(attr, detachstate)

[pthread_attr_getdetachstate](#)(attr, detachstate)

Joining is one way to accomplish *synchronization between threads*: The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.



Joinable or not?

Thread Management – Joining and Detaching

```
int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   pthread_attr_t attr;
   int rc, t, status;

   /* Initialize and set thread detached attribute */
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

   for(t=0; t<NUM_THREADS; t++)
   {
      printf("Creating thread %d\n", t);
      rc = pthread_create(&thread[t], &attr, BusyWork, NULL);
      if (rc)
      {
         printf("ERROR; return code from pthread_create()
                is %d\n", rc);
         exit(-1);
      }
   }

   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++)
   {
      rc = pthread_join(thread[t], (void **)&status);
      if (rc)
      {
         printf("ERROR; return code from pthread_join()
                is %d\n", rc);
         exit(-1);
      }
      printf("Completed join with thread %d status= %d\n",t, status);
   }

   pthread_exit(NULL);
}
```

Thread Management – Stack Management

pthread_attr_getstacksize (attr, stacksize)

pthread_attr_setstacksize (attr, stacksize)

pthread_attr_getstackaddr (attr, stackaddr)

pthread_attr_setstackaddr (attr, stackaddr)

The POSIX standard does not indicate the size of a thread's stack, which is system implementation dependent.

Exceeding the default stack limit: program termination and/or corrupted data

Safe and portable programs should explicitly allocate enough stack for each thread; if the stack must be placed in some particular region of memory, use the last two routines

Default and maximum stack size are system-dependent.

Thread Management – Misc Routines

pthread_self () pthread_equal (thread1, thread2) pthread_yield ()
The thread identifier objects are opaque, the C equivalence operator == should not be used to compare two thread IDs against each other, or to compare a single thread ID against another value Calling thread of pthread_yield() will wait in the *run queue*.

Mutexes

- Mutex: a simplified version of the semaphores
  - a variable that can be in one of two states: unlocked or locked
  - Supports synchronization by controlling access to shared data
- A typical sequence in the use of a mutex
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex and repeats the process
  - Finally the mutex is destroyed
- Do not want to block?
  - An unblocking call with "trylock", instead of blocking "lock" call
  - It is programmer's responsibility to make sure *every thread* that needs to use a mutex to protect shared data does so

Mutex Management – Creating and Destroying Mutexes

pthread_mutex_init (mutex, attr)

pthread_mutex_destroy (mutex)

pthread_mutexattr_init (attr)

pthread_mutexattr_destroy (attr)

1. Mutex variables must be declared with type pthread_mutex_t, and must be initialized before can be used. The mutex is initially not locked.

   Statically: pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER
   Dynamically, with **pthread_mutex_init (mutex, attr)**

2. The attr object must be declared with type pthread_mutexattr_t

3. Programmers should free a mutex object that is no longer used

Mutex Management – Locking and Unlocking of Mutexes

pthread_mutex_lock (mutex)	; P(down)
pthread_mutex_trylock (mutex)	
pthread_mutexattr_unlock (mutex)	; V(up)

1. pthread_mutex_lock (mutex) is a blocking call.
2. pthread_mutex_trylock (mutex) is a non-blocking call, useful in preventing the deadlock conditions (priority-inversion problem)
3. If you use multiple mutexes, the order is important;

Monitors and Condition Variables

- Monitor: a higher-level synchronization primitive

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```

But, how processes block when they cannot proceed?

Condition variables, and two operations: *wait()* and *signal()*

Condition Variables and Mutexes

- Mutexes: support synchronization by controlling thread access to data

- Condition variables: another way for threads to synchronize
  - Allows thread to synchronize based on the actual value of data
  - Always used in conjunction with a mutex lock, why?
    - In Monitors, mutual exclusion is achieved with compiler's help which ensures at any time only one process can be active in a monitor
    - wait() and signal()

A Representative Sequence Using Condition Variables

<table>
<tr><td colspan="2">

Main Thread

Declare and initialize global data/variables which require synchronization (such as "count")
Declare and initialize a condition variable object;
Declare and initialize an associated mutex
Create threads A and B to do work

</td></tr>
<tr><td>

Thread A

1. Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
2. Lock associated mutex and check value of a global variable
3. Call *pthread_cond_wait()* to perform a blocking wait for signal from Thread-B. Note that a call to *pthread_cond_wait()* automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
4. When signalled, wake up. Mutex is automatically and atomically locked.
5. Explicitly unlock mutex
6. Continue

</td><td>

Thread B

1. Do work
2. Lock associated mutex
3. Change the value of the global variable that Thread-A is waiting upon.
4. Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A
5. Unlock mutex.
6. Continue

</td></tr>
<tr><td colspan="2">

Main Thread

Join / Continue

</td></tr>
</table>

Condition Variables – Creating and Destroying Con. Variables

pthread_cond_init (condition, attr)

pthread_cond_destroy (condition)

pthread_condattr_init (attr)

pthread_condattr_destroy (attr)

1. Mutex variables must be declared with type pthread_cond_t, and must be initialized before can be used. The mutex is initially not locked.

   Statically: pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER
   Dynamically, with pthread_cond_init (condition, attr)

2. The attr object must be declared with type pthread_condattr_t

3. Programmers should free a condition variable that is no longer used

Condition Variables – Waiting and Signaling Con. Variables

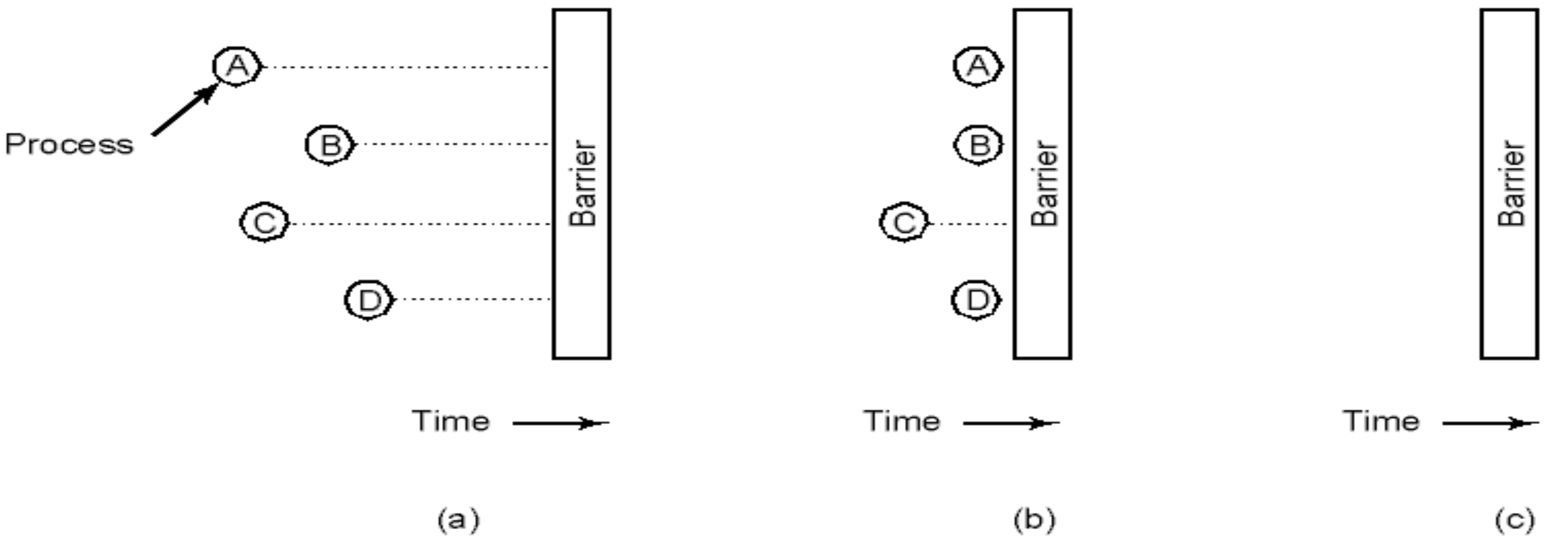
pthread_cond_wait (condition, mutex)

pthread_cond_signal (condition)

pthread_cond_broadcast (condition)

1. wait() blocks the calling thread until the specified condition is signaled. It should be called while mutex is locked
2. signal() is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked, and programmers must unlock mutex in order for wait() routine to complete
3. broadcast() is useful when multiple threads are in blocked waiting
4. wait() should come before signal() – conditions are not counters, signal would be lost if not waiting

Barriers



- Use of a barrier (for programs operate in *phases,* neither enters the next phase until all are finished with the current phase) for groups of processes to do synchronization
  (a) processes approaching a barrier
  (b) all processes but one blocked at barrier
  (c) last process arrives, all are let through

Barriers – Initializing and waiting on barriers

[pthread_barrier_init]() (barrier, attr, count)

[pthread_barrier_wait]() (barrier)

[pthread_barrier_destroy]() (barrier)

1. init() function shall allocate any resources required to use the barrier referenced by barrier and shall initialize the barrier with attr. If attr is NULL, default settings will be applied. The count argument specifies the number of threads that must call wait() before any of them successfully return from the call.

2. wait() function shall synchronize participating threads at the barrier.

3. destroy() function shall destroy the barrier and release any resources used by the barrier.

POSIX Semaphores

- Semaphores are counters for resources shared between threads. The basic operations on semaphores are: increment the counter atomically, and wait until the counter is non-null and decrement it atomically.
- The pthreads library implements POSIX 1003.1b semaphores. These should not be confused with System V semaphores (ipc, semctl and semop).
- All the semaphore functions & macros are defined in semaphore.h.
  - int sem_init (*sem_t *sem, int pshared, unsigned int value*)
  - int sem_destroy (*sem_t * sem*)
  - int sem_wait (*sem_t * sem*)
  - int sem_trywait (*sem_t * sem*)
  - int sem_post (*sem_t * sem*)
  - int sem_getvalue (*sem_t * sem, int * sval*)

Summary

- What is Pthread?
- Thread management
  - Creation and termination
  - Passing argument(s)
  - Joining and detaching
  - Stack management
- Mutex management
  - Creating and destroying
  - Locking and unlocking
- Condition variables
  - Creating and destroying
  - Waiting and signaling