CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared Memory Programming Using POSIX Threads

Instructor: Haidar M. Harmanani

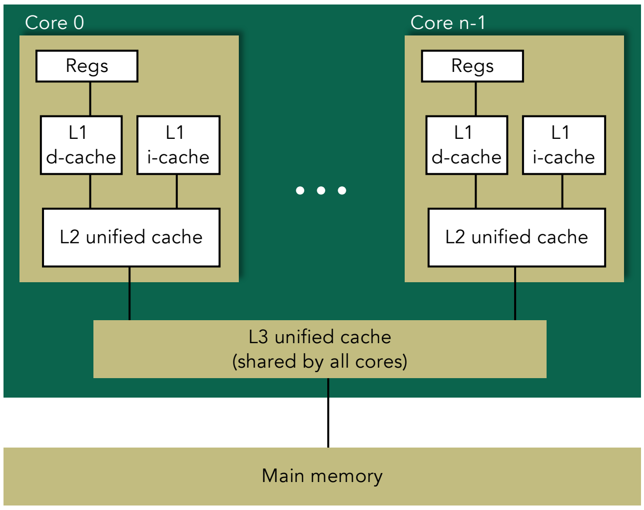
Spring 2020

Multicore vs. Manycore

- A multicore processor is a single computing component with two or more “independent” processors (called "cores”)
- A many-core processor is a multi-core processor (probably heterogeneous) in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient.
  - Tilera TILE-GX processor family with 16 to 100 identical processor cores.
  - Intel Knights Corner with 50 cores

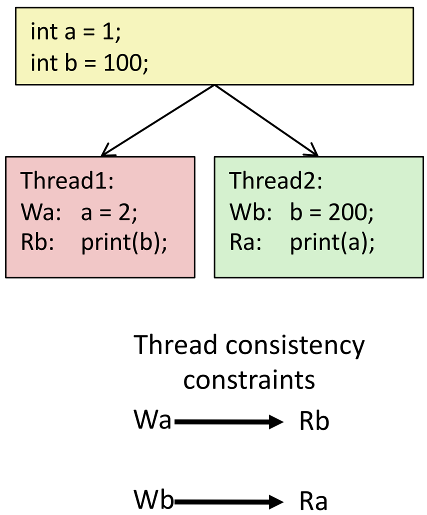
Multicore Processor

- Intel Nehalem Processor
  - Multiple processors operating with coherent view of memory

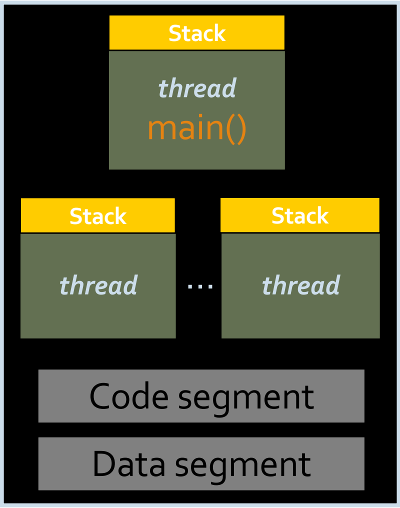


Memory Consistency

- What are the possible values printed?
  - Depends on memory consistency model
  - Abstract model of how hardware handles concurrent accesses
- Sequential consistency
  - Overall effect consistent with each individual thread
  - Otherwise, arbitrary interleaving



Processes and Threads

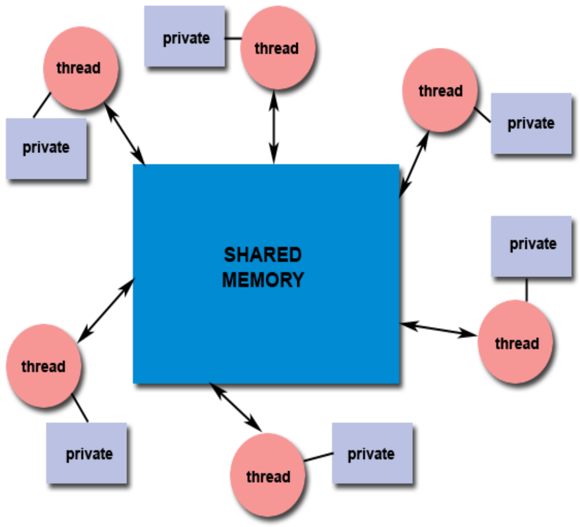


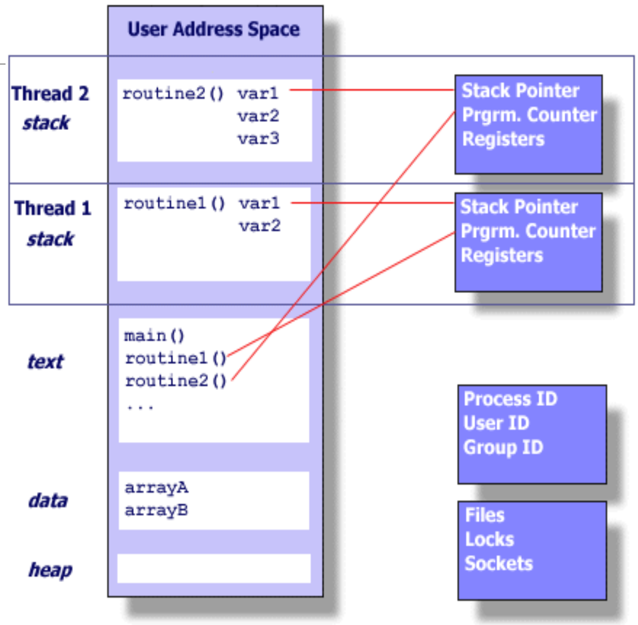
- Modern operating systems load programs as processes
  - Resource holder
  - Execution
- A process starts executing at its entry point as a thread
- Threads can create other threads within the process
  – Each thread gets its own stack
- All threads within a process share code & data segments

Shared Memory Model

- All threads have equal priority read/write access to the same global (shared) memory.
- Threads also have their own private data.
  – Local variables for instance.
- Programmers are responsible for synchronizing any write access (protection) to globally shared data.

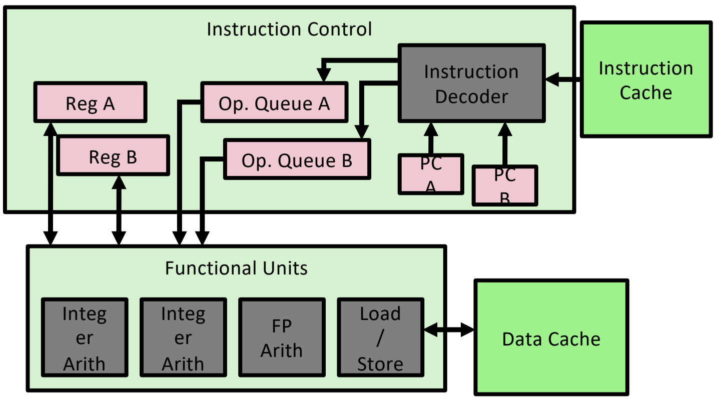
Threads





Hyperthreading

- Replicate enough instruction control to process K instruction streams
  - K copies of all registers
  - Share functional units
- Hyper-threading (HT - Intel) is the ability, of the hardware and the system, to schedule and run two threads or processes simultaneously on the same processor core



Designing Threaded Programs

- Several common models for threaded programs exist:
  - *Manager/worker*
    - A single thread, the *manager* assigns work to other threads, the *workers*.
    - The manager handles all input and parcels out work to the other tasks. At least two forms of the manager/worker model are common: static worker pool and dynamic worker pool.
  - *Pipeline*
    - A task is broken into a series of suboperations, each of which is handled in series, but concurrently, by a different thread.
  - *Peer*
    - similar to the manager/worker model, but after the main thread creates other threads, it participates in the work.

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Pthreads

What are Pthreads?

- IEEE POSIX 1003.1c standard
- pthreads routines be grouped in the following categories
  - *Thread Management*: Routines to create, terminate, and manage the threads.
  - *Mutexes*: Routines for synchronization
  - *Condition Variables*: Routines for communications between threads that share a mutex.
  - *Synchronization*: Routines for the management of read/write locks and barriers.
- All identifiers in the threads library begin with pthread_

Why Pthreads?

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.8 GHz Xeon 5660 (12 cores)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores)	12.5	1	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus)	104.5	48.6	47.2	2.1	1	1.5
INTEL 2.4 GHz Xeon (2 cpus)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus)	54.5	1.1	22.2	2	1.2	0.6

Latest: Intel Xeon Platinum 8176 (28 cores, 56 threads)

Preliminaries

- All major thread libraries on Unix systems are Pthreads-compatible
- Include pthread.h in the main file
- Compile program with `–lpthread`
  - `gcc -o test test.c -lpthread`
  - may not report compilation errors otherwise but calls will fail
  - The MacOS has dropped the need for the inclusion of `-lpthread`
  - Check your OS's requirement!
- Good idea to check return values on common functions

The Pthreads API

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
pthread_rwlock_	Read/write locks
pthread_barrier_	Synchronization barriers

Creating Threads

- Identify portions of code to thread
- Encapsulate code into function
  - If code is already a function, a driver function may need to be written to coordinate work of multiple threads
- Add **`pthread_create()`** call to assign thread(s) to execute function

pthread_create

- **`int pthread_create(tid, attr, function, arg);`**
  - **`pthread_t *tid`**
    - Handle of created thread
  - **`const pthread_attr_t *attr`**
    - attributes of thread to be created
    - You can specify a thread attributes object, or NULL for the default values.
  - **`void *(*function)(void *)`**
    - The C routine that the thread will execute once it is created
  - **`void *arg`**
    - single argument to function
    - NULL may be used if no argument is to be passed.

Example: pthread_create

`pthread_create(&threads[t], NULL, HelloWorld, (void *) t)`

- Spawn a thread running the function
- Thread handle returned via `pthread_t` structure
  - Specify NULL to use default attributes
- Single argument sent to function
  - If no arguments to function, specify NULL
- Check error codes!

EAGAIN - insufficient resources to create thread
EINVAL - invalid attribute

How Many Threads Can One Create?

```
rey:~ haidar$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
file size               (blocks, -f) unlimited
max locked memory       (kbytes, -l) unlimited
max memory size         (kbytes, -m) unlimited
open files                      (-n) 256
pipe size            (512 bytes, -p) 1
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1418
virtual memory          (kbytes, -v) unlimited
```

What is the Outcome of the following code?

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

void *hello ()
{
   printf("Hello Thread\n");
}

main() {
       pthread_t tid;

          pthread_create(&tid, NULL, hello, NULL);
}
```


Example: Thread Creation

- Two possible outcomes
  - Message "Hello Thread" is printed on screen
  - Nothing printed on screen.
    - Why?
      - Main thread is the process and when the process ends, all threads are cancelled, too.
      - Thus, if the `pthread_create` call returns before the OS has had the time to set up the thread and begin execution, the thread will die a premature death when the process ends.
      - Need some method of waiting for a thread to finish

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

void *hello ()
{
   printf("Hello Thread\n");
}

main() {
       pthread_t tid;

          pthread_create(&tid, NULL, hello, NULL);
}
```

Waiting for a Thread

int pthread_join(tid, val_ptr);

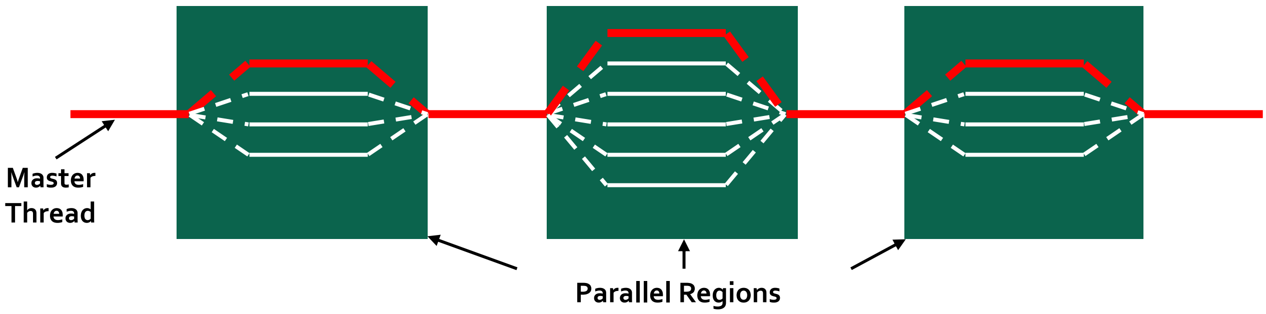
- `pthread_join` will block until the thread associated with the `pthread_t` handle has terminated.
  - There is no single function that can join multiple threads.
- The second parameter returns a pointer to a value from the thread being joined.
- `pthread_join()` can be used to wait for one thread to terminate.

`pthread_t tid`
  - handle of *joinable* thread

`void **val_ptr`
  - exit value returned by joined thread

Fork-Join Execution Model

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed
  - Parallelism is added incrementally: that is, the sequential program evolves into a parallel program

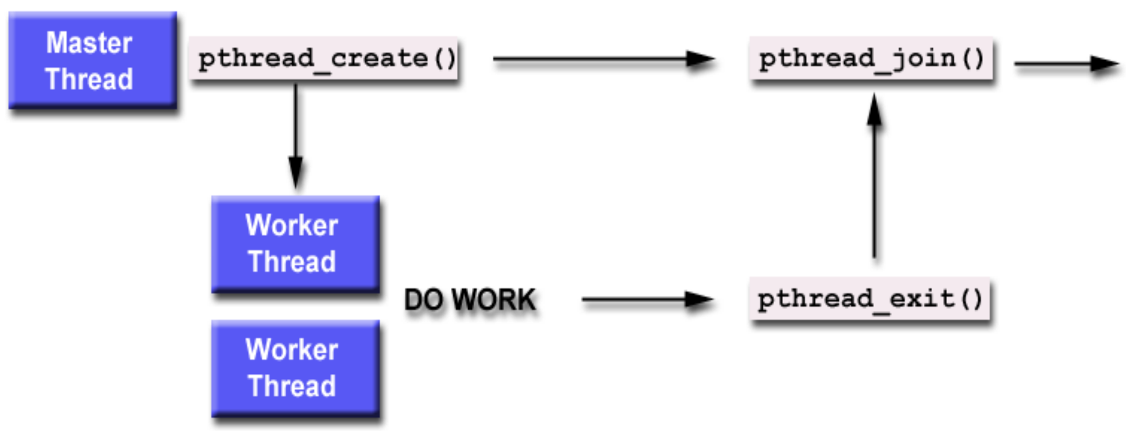


pthread_join

- Calling thread waits for thread with handle `tid` to terminate
  - Can't do multiple `pthread_joins` on the same thread
  - Thread must be *joinable*
- Exit value is returned from joined thread
  - Type returned is `(void *)`
  - Use `NULL` if no return value expected

ESRCH - thread (pthread_t) not found
EINVAL - thread (pthread_t) not joinable

pthread_join



A Better Hello Threads…

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 8
void* hello(void* threadID) {
    long id = (long) threadID;
    printf("Hello World, this is thread %ld\n", id);
    return NULL;
}
int main(int argc, char argv[]) {
    long t;
    pthread_t thread_handles[NUM_THREADS];
    for(t=0 ; t<NUM_THREADS; t++)
        pthread_create(&thread_handles[t], NULL, hello, (void *) t);
    printf("Hello World, this is the main thread\n");
    for(t=0; t<NUM_THREADS; t++)
        pthread_join(thread_handles[t], NULL);
    return 0;
}
```


Sample Execution Runs

```
yoda:~ haidar$ ./a.out
Hello World, this is thread 0
Hello World, this is thread 1
Hello World, this is thread 2
Hello World, this is thread 3
Hello World, this is thread 4
Hello World, this is thread 5
Hello World, this is the main
thread
Hello World, this is thread 7
Hello World, this is thread 6
```

```
yoda:~ haidar$ ./a.out
Hello World, this is thread 0
Hello World, this is thread 1
Hello World, this is thread 2
Hello World, this is thread 3
Hello World, this is thread 4
Hello World, this is the main
thread
Hello World, this is thread 5
Hello World, this is thread 7
Hello World, this is thread 6
```

Thread States

- pthreads threads have two states
  – *joinable* and *detached*
- Threads are joinable by default
  – Resources are kept until **`pthread_join`**
  – Can be reset with attributes or API call
- Detached threads cannot be joined
  – Resources can be reclaimed at termination
  – Cannot reset to be *joinable*

Example: Multiple Threads Creation

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS        5
void *PrintHello(void *threadid)
{
    long tid;
    tid = (long) threadid;
    printf("Hello World! I am thread #%ld!\n", tid);
    pthread_exit(NULL);
}
```

```
int main(int argc, char *argv[])
{
    pthread_t threads[NUM_THREADS];
    int rc;
    long t;
    for(t=0;t<NUM_THREADS;t++){
        rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
       if (rc){
            printf("ERROR; return code from
                pthread_create() is %d\n", rc);
        }
      }
}
```

pthread_exit

```
void pthread_exit()

void pthread_exit(                         // terminate a thread
    void *status                           // completion status
);
```

Arguments:

The completion status of the thread that has exited. This pointer value is available to other threads.

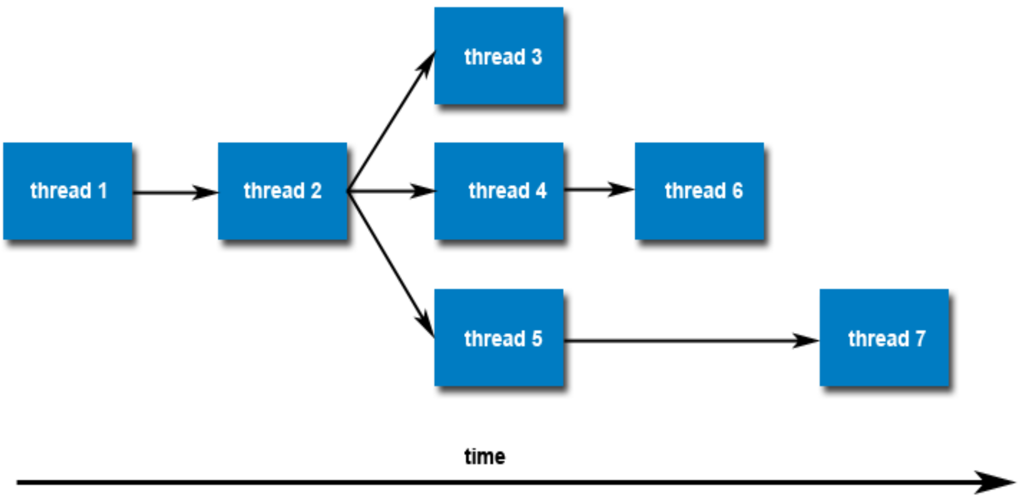
Return value:

None.

Notes:

When a thread exits by simply returning from the start routine, the thread's completion status is set to the start routine's return value.

No implied hierarchy or dependency between threads



Example: Multiple Threads with Joins

```
#include <stdio.h>
#include <pthread.h>
#define NUM_THREADS 4

void *hello () {
       printf("Hello Thread\n");
}

main() {
       pthread_t tid[NUM_THREADS];
       for (int i = 0; i < NUM_THREADS; i++)
               pthread_create(&tid[i], NULL, hello, NULL);

       for (int i = 0; i < NUM_THREADS; i++)
               pthread_join(tid[i], NULL);
}
```


Home Lab 3

Parallel Summation

Parallel Summation Problem

- Sum numbers 0, …, N-1
- Partition into K ranges
  - ⌊N/K⌋ values each
  - Accumulate leftover values serially
- Method #1: All threads update single global variable
- Report your numbers based on the following approaches [Home]:
  - 1A: No synchronization
  - 1B: Synchronize with **`pthread`** semaphore
  - 1C: Synchronize with **`pthread_mutex`**
    - "Binary" semaphore. Only values 0 & 1

Accumulating in Single Global Variable: Declarations

```
typedef unsigned long data_t;
/* Single accumulator */
volatile data_t global_sum;

/* Mutex & semaphore for global sum */
sem_t semaphore;
pthread_mutex_t mutex;

/* Number of elements summed by each thread */
size_t nelems_per_thread;

/* Keep track of thread IDs */
pthread_t tid[MAXTHREADS];

/* Identify each thread */
int myid[MAXTHREADS];
```

Accumulating in Single Global Variable: Operation

```
nelems_per_thread = nelems / nthreads;
/* Set global value */
global_sum = 0;

/* Create threads and wait for them to finish */
for (i = 0; i < nthreads; i++) {
    myid[i] = i;
    Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
}
for (i = 0; i < nthreads; i++)
    Pthread_join(tid[i], NULL);

result = global_sum;

/* Add leftover elements */
for (e = nthreads * nelems_per_thread; e < nelems; e++)
     result += e;
```


Thread Function: No Synchronization

```
void *sum_race(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        global_sum += i;
    }
    return NULL;
}
```

How to Avoid Data Races

- Scope variables to be local to threads
  - Variables declared within threaded functions
  - Allocate on thread's stack
  - Thread Local Storage (TLS)
- Control shared access with critical regions
  - Mutual exclusion and synchronization
  - Lock, semaphore, condition variable, critical section, mutex…

pthread's Mutex

- Simple, flexible, and efficient
- Enables correct programming structures for avoiding race conditions
- `Mutex` variables must be declared with type **`pthread_mutex_t`**, and must be initialized before they can be used
- Attributes are set using **`pthread_mutexattr_t`**
- The `mutex` is initially unlocked.

Initializing `mutex` Variables

- Two ways:
  - Statically, when it is declared:
    - pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;
  - Dynamically, with the pthread_mutex_init() routine.
    - Permits setting mutex object attributes, *attr*.

pthread_mutex_init

```
int pthread_mutex_init( mutex, attr );
```

`pthread_mutex_t *mutex`
– mutex to be initialized

`const pthread_mutexattr_t *attr`
– attributes to be given to mutex

- The Pthreads standard defines three optional mutex attributes:
  - Protocol: Specifies the protocol used to prevent priority inversions for a mutex.
  - Prioceiling: Specifies the priority ceiling of a mutex.
  - Process-shared: Specifies the process sharing of a mutex.

Alternate Initialization

- Can also use the static initializer
  PTHREAD_MUTEX_INITIALIZER

```
pthread_mutex_t mtx1 = PTHREAD_MUTEX_INITIALIZER;
```

  – Uses default attributes
- Programmer must always pay attention to mutex scope
  – Must be visible to threads

pthread_mutex_lock

```
int pthread_mutex_lock( mutex );
```

`pthread_mutex_t *mutex`
- mutex to attempt to lock

- Used by a thread to acquire a lock on the specified mutex variable
  – If mutex is locked by another thread, calling thread is blocked
- Mutex is held by calling thread until unlocked
  – Mutex lock/unlock must be paired or deadlock occurs

`EINVAL` - mutex is invalid
`EDEADLK` - calling thread already owns mutex

pthread_mutex_trylock

- Attempt to lock a mutex.
- If the mutex is already locked, the routine will return immediately with a "busy" error code.
- This routine may be useful in preventing deadlock conditions, as in a priority-inversion situation.

pthread_mutex_unlock

```
int pthread_mutex_unlock( mutex );
```

`pthread_mutex_t *mutex`
– mutex to be unlocked

`EINVAL` - **mutex is invalid**
`EPERM` - **calling thread does not own mutex**

Freeing `mutex` Objects and Attributes

- Used to free a **`mutex`** object which is no longer needed
- **`pthread_mutexattr_init()`** and **`pthread_mutexattr_destroy()`**
  - Create and destroy mutex attribute objects respectively
- **`pthread_mutex_destroy()`**
  - Used to free a mutex object which is no longer needed.

More on Mutexes

Acquiring and Releasing Mutexes

```
int pthread_mutex_lock(              // Lock a mutex
  pthread_mutex_t *mutex);
int pthread_mutex_unlock(            // Unlock a mutex
  pthread_mutex_t *mutex);
int pthread_mutex_trylock(           // Nonblocking lock
  pthread_mutex_t *mutex);
```

Arguments:

Each function takes the address of a mutex variable.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

The `pthread_mutex_trylock()` routine attempts to acquire a mutex but will not block. This routine returns the POSIX Threads constant `EBUSY` if the mutex is locked.

More on Mutexes

Dynamically Allocated Mutexes

```
pthread_mutex_t *lock;                    // Declare a pointer to a lock
lock=(pthread_mutex_lock_t *) malloc(sizeof(pthread_mutex_t));
pthread_mutex_init(lock, NULL);
    /*
     * Code that uses this lock.
     */
pthread_mutex_destroy(lock);
free(lock);
```


Thread Function: Semaphore / Mutex

Semaphore

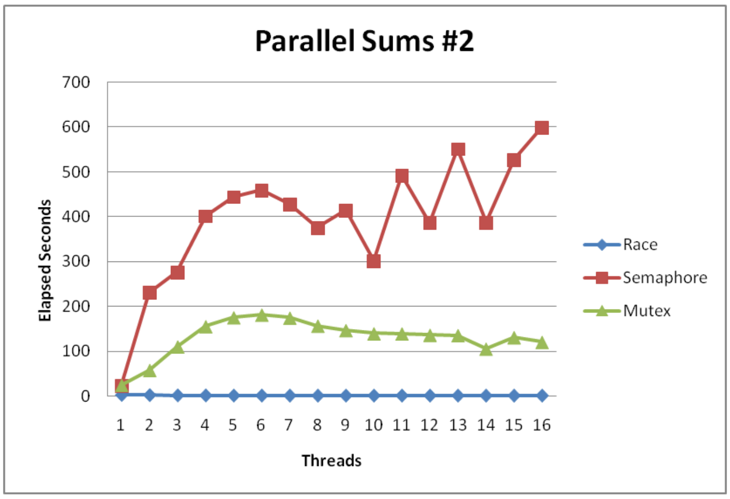
```
void *sum_sem(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    for (i = start; i < end; i++) {
        sem_wait(&semaphore);
        global_sum += i;
        sem_post(&semaphore);
    }
    return NULL;
}
```

Mutex

```
pthread_mutex_lock(&mutex);
global_sum += i;
pthread_mutex_unlock(&mutex);
```

Semaphore / Mutex Performance

- Terrible Performance
  - 2.5 seconds ➔ ~10 minutes
- Mutex 3X faster than semaphore
- Clearly, neither is successful



Separate Accumulation

- Method #2: Each thread accumulates into separate variable
  - 2A: Accumulate in contiguous array elements
  - 2B: Accumulate in spaced-apart array elements
  - 2C: Accumulate in registers

```
/* Partial sum computed by each thread */
data_t psum[MAXTHREADS*MAXSPACING];
/* Spacing between accumulators */
size_t spacing = 1;
```

Separate Accumulation: Operation

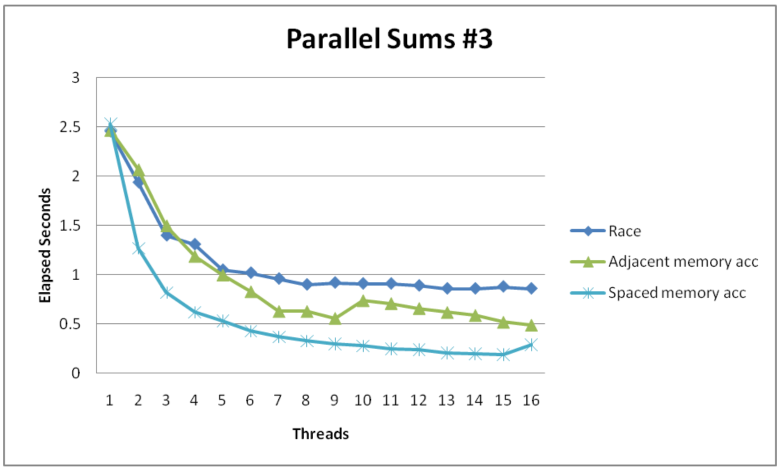
```
    nelems_per_thread = nelems / nthreads;

    /* Create threads and wait for them to finish */
    for (i = 0; i < nthreads; i++) {
        myid[i] = i;
        psum[i*spacing] = 0;
        Pthread_create(&tid[i], NULL, thread_fun, &myid[i]);
    }
    for (i = 0; i < nthreads; i++)
        Pthread_join(tid[i], NULL);

    result = 0;
    /* Add up the partial sums computed by each thread */
    for (i = 0; i < nthreads; i++)
        result += psum[i*spacing];
    /* Add leftover elements */
    for (e = nthreads * nelems_per_thread; e < nelems; e++)
        result += e;
```


Thread Function: Memory Accumulation

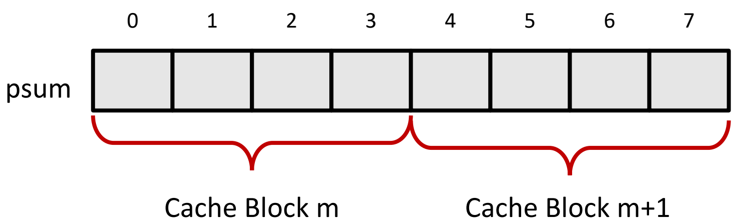
```
void *sum_global(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;

    size_t index = myid*spacing;
    psum[index] = 0;
    for (i = start; i < end; i++) {
        psum[index] += i;
    }
    return NULL;
}
```

Memory Accumulation Performance

- Clear threading advantage
  - Adjacent speedup: 5 X
  - Spaced-apart speedup: 13.3 X (Only observed speedup > 8)
- Why does spacing the accumulators apart matter?



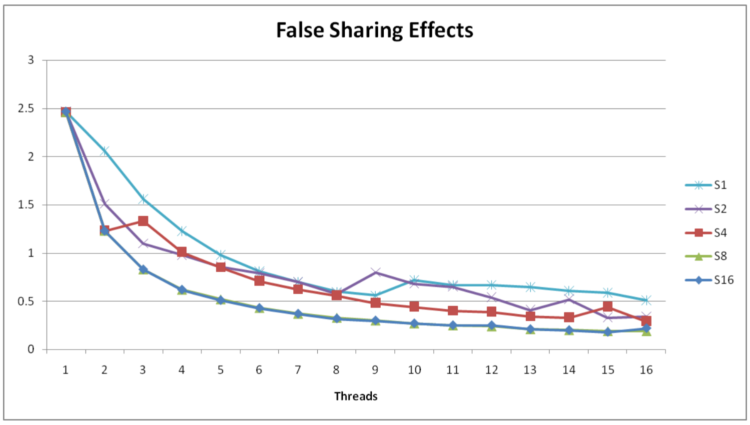
False Sharing



- Coherency maintained on cache blocks
- To update psum[i], thread i must have exclusive access
  - Threads sharing common cache block will keep fighting each other for access to block

False Sharing Performance

- Best spaced-apart performance 2.8 X better than best adjacent
- Demonstrates cache block size = 64
  - 8-byte values
  - No benefit increasing spacing beyond 8

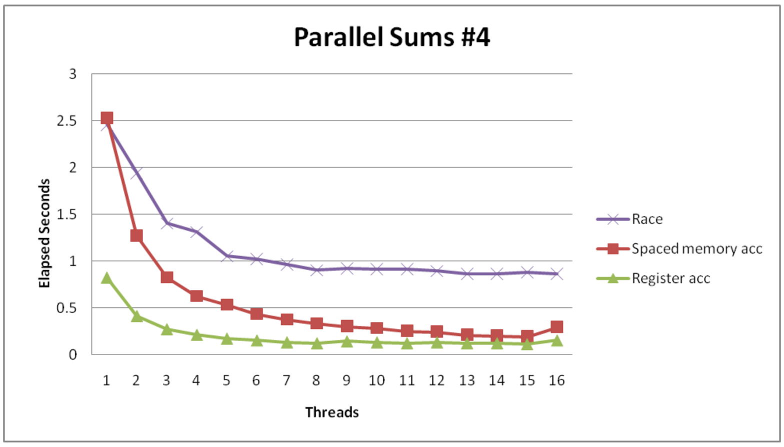


Thread Function: Register Accumulation

```
void *sum_local(void *vargp)
{
    int myid = *((int *)vargp);
    size_t start = myid * nelems_per_thread;
    size_t end = start + nelems_per_thread;
    size_t i;
    size_t index = myid*spacing;
    data_t sum = 0;
    for (i = start; i < end; i++) {
        sum += i;
    }
    psum[index] = sum;    return NULL;
}
```

Register Accumulation Performance

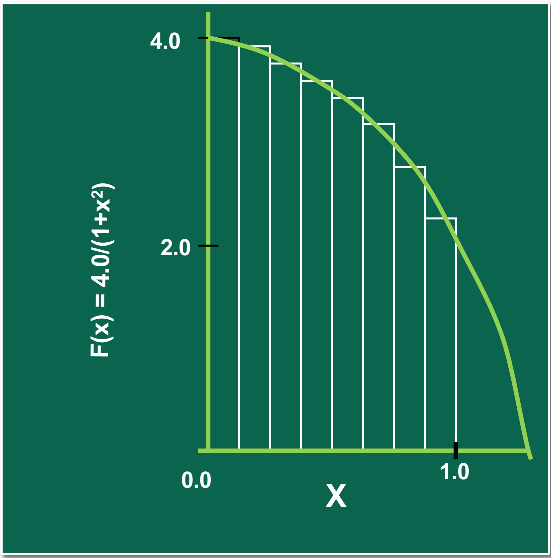
- Clear threading advantage
  - Speedup = 7.5 X
- 2X better than fastest memory accumulation



Examples

Example 1: Computing π

$$\tan^{-1}(x) = \int_0^x \frac{dt}{1+t^2}$$



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

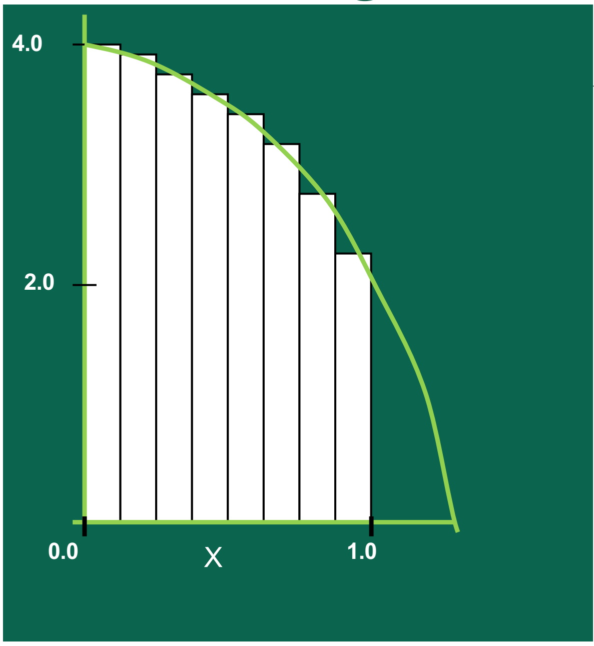
Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Example 1: Computing π

$$\tan^{-1}(x) = \int_0^x \frac{dt}{1+t^2}$$

```
static long num_steps = 100000;
double step;
void main ()
{
   int i;
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

Computing π



$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

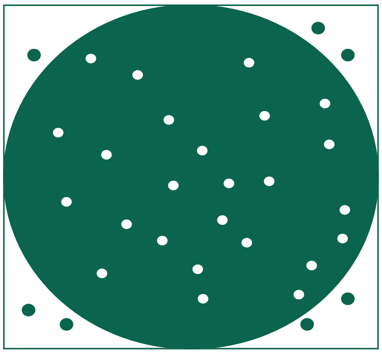
Assg 2: Computing π

- Parallelize the numerical integration code using POSIX* Threads
- How can the loop iterations be divided among the threads?
- What variables can be local?
- What variables need to be visible to all threads?

```
static long num_steps=100000;
double step, pi;

void main()
{   int i;
    double x;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        pi += 4.0/(1.0 + x*x);
    }
    pi *= step;
    printf("Pi = %f\n",pi);
}
```

Assg 2: Computing π (Continue)

```
int inTheCircle=0, numtrials=1000000;
double x, y;

for (i=0; i<numTrials; i++){
   x = rand();
   y = rand();
   if((x*x + y*y) < 1.0)
      inTheCircle++;
   }
pi = 4*inTheCircle/numTrials;
```



Example 2: Dot Product

```
void *dotprod(void *arg)
 {

    int i, start, end, len ;
    long offset;
    double mysum, *x, *y;
     offset = (long)arg;

    len = dotstr.veclen;
    start = offset*len;
    end   = start + len;
    x = dotstr.a;
    y = dotstr.b;
    mysum = 0;

   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }

    pthread_mutex_lock (&mutexsum);
    dotstr.sum += mysum;
    pthread_mutex_unlock (&mutexsum);

    pthread_exit((void*) 0);
 }
```

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

/* Example program creating thread to compute square of
 value */

int value;      /* thread stores result here */
void *my_thread(void *param);        /* the thread */

main (int argc, char *argv[])
{
 pthread_t tid;           /* thread identifier */
 int retcode;

 /* check input parameters */
 if (argc != 2) {
    fprintf (stderr, "usage: a.out <integer value>\n");
    exit(0);
 }
```

```c
 /* create the thread */
 retcode = pthread_create(&tid,NULL,my_thread,argv[1]);
 if (retcode != 0) {
          fprintf (stderr, "Unable to create thread\n");
          exit (1);
 }

 /* wait for created thread to exit */
 pthread_join(tid,NULL);
 printf ("I am the parent: Square = %d\n", value);
} //main

/* The thread will begin control in this function */
void *my_thread(void *param)
{
 int i = atoi (param);

 printf ("I am the child, passed value %d\n", i);
 value = i * i;

 /* next line is not really necessary */
 pthread_exit(0);
}
```


Condition Variables

How to check whether a child thread has completed?

```
void  *child(void  *arg) {
  printf("child\n");
  // XXX  how  to indicate we  are done?
  return  NULL;
}

int  main(int  argc,  char  *argv[])  {
  printf("parent: begin\n");
  pthread_t c;
  Pthread_create(&c,  NULL,  child,  NULL);  //  create  child
  //  XXX  how  to  wait  for  child?
  printf("parent: end\n");
  return  0;
```


Possible Solution: Problem?

```
volatile int done  =  0;

void  *child(void  *arg) {
  printf("child\n");
  done  =  1;
  return  NULL;
}

int  main(int  argc,  char  *argv[])  {
  printf("parent: begin\n");
  pthread_t  c;
  pthread_create(&c,  NULL,  child,  NULL);  //  create  child
  while  (done  ==  0)
    ; // spin
  printf("parent: end\n");
  return  0;
}
```

The Crux

HOW TO WAIT FOR A CONDITION?
In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. The simple approach, of just spinning until the condition becomes true, is grossly inefficient and wastes CPU cycles, and in some cases, can be incorrect. Thus, how should a thread wait for a condition?

Condition Variables

- There are cases where a thread wishes to check whether a condition is true before continuing its execution
- Condition variables provide another mechanism for threads to synchronize.
  - mutex'es implement synchronization by controlling thread access to data
  - Condition variables allow threads to synchronize based on the actual value of data.

Condition Variables

- A condition variable is an explicit queue that threads can put themselves on when some state of execution is not as desired
  - Without condition variables, the programmer would need to have threads continually polling (possibly in a critical section), to check if the condition is met.
    - Resource consuming
- The idea goes back to Dijkstra's use of "private semaphores"
- A condition variable is always used in conjunction with a **`mutex`** lock.

Condition Variables

- A condition variable (cv) allows a thread to block itself until a specified condition becomes true.
- When a thread executes **`pthread_cond_wait(cv)`**, it is blocked until another thread executes **`pthread_cond_signal(cv)`** or **`pthread_cond_broadcast(cv)`**.
  - pthread_cond_signal() is used to unblock one of the threads blocked waiting on the condition variable.
  - pthread_cond_broadcast() is used to unblock all the threads blocked waiting on the condition variable.
- If no threads are waiting on the condition variable, then a pthread-cond_signal() or pthreadcond_broadcast() will have no effect.

Condition Variables

- Use a **`while`** loop instead of an **`if`** statement to check the waited for condition in order to alleviate the following problems:
  - If several threads are waiting for the same wake up signal, they will take turns acquiring the mutex, and any one of them can then modify the condition they all waited for.
  - If the thread received the signal in error due to a program bug
  - The Pthreads library is permitted to issue spurious wake ups to a waiting thread without violating the standard.
- Mutex is unlocked
  - Allows other threads to acquire lock
  - When signal arrives, mutex will be reacquired before **`pthread_cond_wait`** returns

Main Thread
- Declare and initialize global data/variables that 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

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

Thread B
- Do work
- Lock associated mutex
- Change the value of the global variable that ***Thread A*** is waiting upon.
- Check value of the global ***Thread A*** wait variable. If it fulfills the desired condition, **signal** ***Thread A***.
- Unlock mutex.
- Continue

Main Thread
Join / Continue

Problem

- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

```
int main (int argc, char *argv[])
 {
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i=0; i<NUM_THREADS; i++) {
     pthread_join(threads[i], NULL);
  }
  printf ("Main(): Waited on %d  threads. Done.\n", NUM_THREADS);

  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
  pthread_cond_destroy(&count_threshold_cv);
  pthread_exit(NULL);
 }
```


Problem

- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

```
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
 {
   int i;
   long my_id = (long)t;

   for (i=0; i<TCOUNT; i++) {
     pthread_mutex_lock(&count_mutex);
     count++;

     if (count == COUNT_LIMIT) {
       pthread_cond_signal(&count_threshold_cv);
       printf("inc_count(): thread %ld, count = %d  Threshold reached.\n", my_id, count);
       }
     printf("inc_count(): thread %ld, count = %d, unlocking mutex\n", my_id, count);
     pthread_mutex_unlock(&count_mutex);

       sleep(1);
     }
   pthread_exit(NULL);
 }
```

Problem

- Two of the threads perform work and update a "count" variable.
- The third thread waits until the count variable reaches a specified value.

```
void *watch_count(void *t)
 {
   long my_id = (long)t;

   printf("Starting watch_count(): thread %ld\n", my_id);

  /*
   Lock mutex and wait for signal.  Note that the pthread_cond_wait routine will automatically and
   atomically unlock mutex while it waits. Also, note that if COUNT_LIMIT is reached before this
   routine is run by   the waiting thread, the loop will be skipped to prevent pthread_cond_wait
   from never returning.
  */
   pthread_mutex_lock(&count_mutex);
   while (count<COUNT_LIMIT) {
     pthread_cond_wait(&count_threshold_cv, &count_mutex);
     printf("watch_count(): thread %ld Condition signal received.\n", my_id);
     count += 125;
     printf("watch_count(): thread %ld count now = %d.\n", my_id, count);
    }
   pthread_mutex_unlock(&count_mutex);
   pthread_exit(NULL);
 }
```


Condition Variables: Declaration

- Condition variables must be declared with type `pthread_cond_t`, and must be initialized before they can be used.
  - `pthread cond t c` ← declares **c** as a condition variable
- Two ways to initialize a condition variable:
  - Statically, when it is declared. For example:
    `pthread_cond_t myconvar = PTHREAD_COND_INITIALIZER;`
  - Dynamically, with the `pthread_cond_init()` routine.
    - The ID of the created condition variable is returned to the calling thread through the condition parameter.
    - This method permits setting condition variable object attributes, `attr`.

pthread_cond_init

```
int pthread_cond_init( cond, attr );
```

`pthread_cond_t *cond`
- condition variable to be initialized

`const pthread_condattr_t *attr`
- attributes to be given to condition variable

`ENOMEM` - insufficient memory for mutex
`EAGAIN` - insufficient resources (other than memory)
`EBUSY` - condition variable already intialized
`EINVAL` - `attr` is invalid

Alternate Initialization

- Can also use the static initializer PTHREAD_COND_INITIALIZER

```
pthread_cond_t cond1 = PTHREAD_COND_INITIALIZER;
```

  – Uses default attributes
- Programmer must always pay attention to condition (and mutex) scope
  – Must be visible to threads

Condition Variables: Usage

- A condition variable has two operations associated with it
  - `wait()`
  - `signal()`
- The **`wait()`** call is executed when a thread wishes to put itself to sleep
- The **`signal()`** call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition

```
pthread_cond_wait(pthread_cond_t  *c,  pthread_mutex_t  *m);
pthread_cond_signal(pthread_cond_t *c);
```


Condition Variable and Mutex

- Mutex is associated with condition variables
  - Protects evaluation of the conditional expression
  - Prevents race condition between signaling thread and threads waiting on condition variable
- While the thread is waiting on a condition variable, the mutex is automatically unlocked, and when the thread is signaled, the mutex is automatically locked again

`pthread_cond_signal` Explained

- Signal condition variable, wake one waiting thread
- If no threads waiting, no action taken
  - Signal is not saved for future threads
- Signaling thread need not have mutex
  - May be more efficient
  - Problem may occur if thread priorities used

`EINVAL` **- cond is invalid**

`pthread_cond_broadcast` Explained

- Wake all threads waiting on condition variable
- If no threads waiting, no action taken
  - Broadcast is not saved for future threads
- Signaling thread need not have mutex

`EINVAL` **- cond is invalid**

Condition Variables: Summary

pthread_cond_signal()

```
int pthread_cond_signal(
  pthread_cond_t *cond);             // Condition to signal
int pthread_cond_broadcast(
  pthread_cond_t *cond);             // Condition to signal
```

Arguments:

A condition variable to signal.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Notes:

- These routines have no effect if there are no threads waiting on `cond`. In particular, there is no memory of the signal when a later call is made to `pthread_cond_wait()`.
- The `pthread_cond_signal()` routine may wake up more than one thread, but only one of these threads will hold the protecting mutex.
- The `pthread_cond_broadcast()` routine wakes up all waiting threads. Only one awakened thread will hold the protecting mutex.

Condition Variables: Summary

pthread_cond_wait()

```
int pthread_cond_wait(
  pthread_cond_t *cond,              // Condition to wait on
  pthread_mutex_t *mutex);           // Protecting mutex
int pthread_cond_timedwait(
  pthread_cond_t *cond,
  pthread_mutex_t *mutex,
  const struct timespec *abstime);   // Time-out value
```

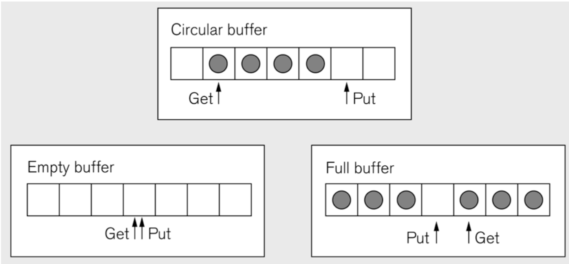
Arguments:

- A condition variable to wait on.
- A mutex that protects access to the condition variable. The mutex is released before the thread blocks, and these two actions occur atomically. When this thread is later unblocked, the mutex is reacquired on behalf of this thread.

Return value:

0 if successful. Error code from `<errno.h>` otherwise.

Condition Variables
Example

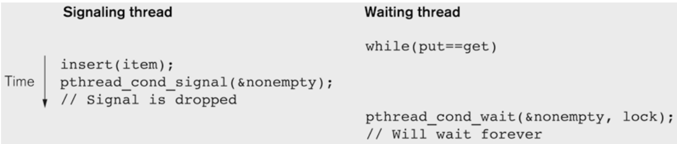


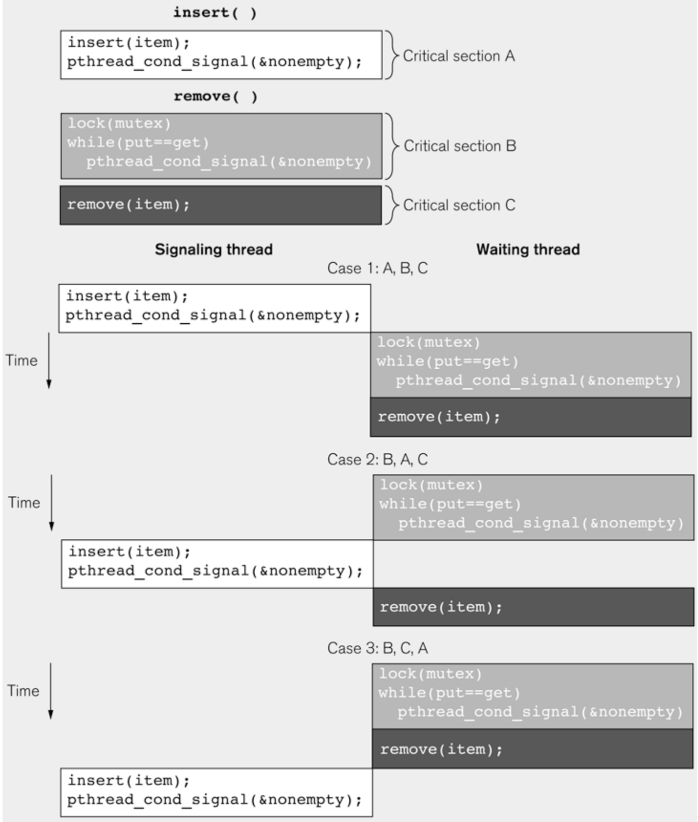

```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
Item buffer[SIZE];
int put=0;                          // Buff index for next insert
int get=0;                          // Buff index for next remove

void insert(Item x)                        // Producer thread
{
  pthread_mutex_lock(&lock);
  while((put>get&&(put-get)==SIZE-1)||     // While buffer is
         (put<get&&(put+get)==SIZE-1))     // full
  {
    pthread_cond_wait(&nonfull, &lock);
  }
  buffer[put]=x;
  put=(put+1)%SIZE;
  pthread_cond_signal(&nonempty);
  pthread_mutex_unlock(&lock);
}

Item remove()                              // Consumer thread
{
  Item x;
  pthread_mutex_lock(&lock);
  while(put==get)                          // While buffer is empty
  {
    pthread_cond_wait(&nonempty, &lock);
  }
  x=buffer[get];
  get=(get+1)%SIZE;
  pthread_cond_signal(&nonfull);
  pthread_mutex_unlock(&lock);
  return x;
}
```


Use of Mutex and Signals


```
pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
Item buffer[SIZE];
int put=0;                          // Buff index for next insert
int get=0;                          // Buff index for next remove

void insert(Item x)                        // Producer thread
{
  pthread_mutex_lock(&lock);
  while((put>get&&(put-get)==SIZE-1)||     // While buffer is
         (put<get&&(put+get)==SIZE-1))     // full
  {
    pthread_cond_wait(&nonfull, &lock);
  }
  buffer[put]=x;
  put=(put+1)%SIZE;
  pthread_cond_signal(&nonempty);
  pthread_mutex_unlock(&lock);
}

Item remove()                              // Consumer thread
{
  Item x;
  pthread_mutex_lock(&lock);
  while(put==get)                          // While buffer is empty
  {
    pthread_cond_wait(&nonempty, &lock);
  }
  x=buffer[get];
  get=(get+1)%SIZE;
  pthread_cond_signal(&nonfull);
  pthread_mutex_unlock(&lock);
  return x;
}
```

Condition Variables
Example





Summary

- Create threads to execute work encapsulated within functions
- Coordinate shared access between threads to avoid race conditions
  - Local storage to avoid conflicts
  - Synchronization objects to organize use