



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

Shared Memory Programming Using POSIX Threads

Instructor: Haidar M. Harmanani

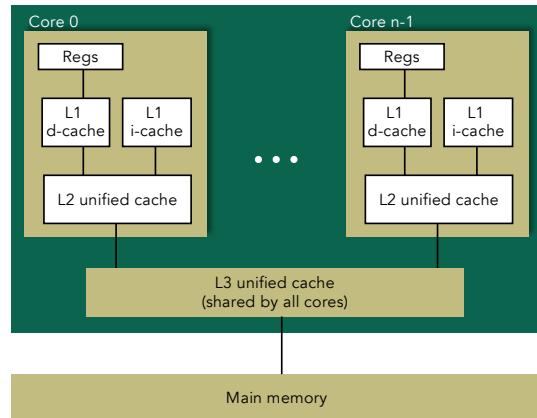
Spring 2019

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

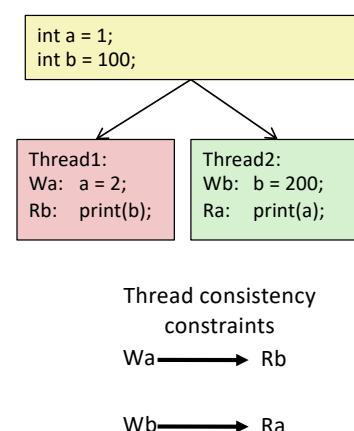
Multicore Processor

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



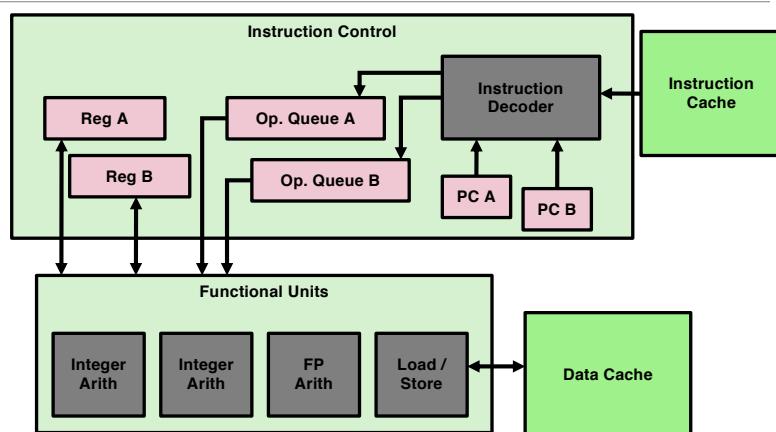
Memory Consistency

- 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

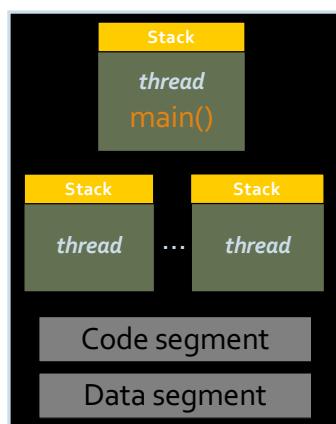


Hyperthreading

- Replicate enough instruction control to process K instruction streams
- K copies of all registers
- Share functional units

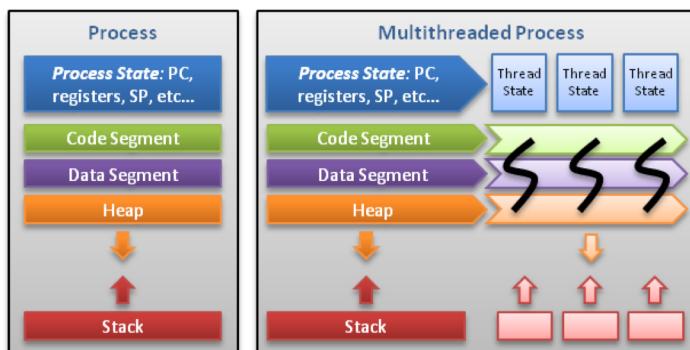


Processes and Threads



- 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

Processes and Threads



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.

© Alfred Park, <http://randu.org/tutorials/threads>

Review of Unix Processes

Unix Processes

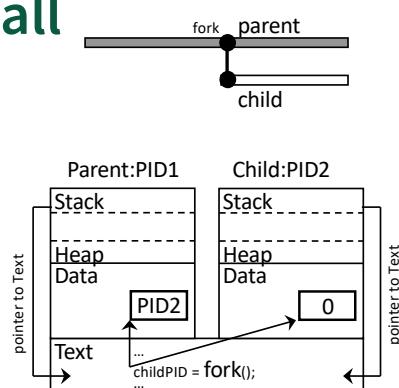
- A UNIX process is created by the operating system, and requires a fair amount of “overhead” including information about:
 - Process ID
 - Environment
 - Working directory.
 - Program instructions
 - Registers
 - Stack
 - Heap
 - File descriptors
 - Signal actions
 - Shared libraries
 - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

```
#include <unistd.h>
pid_t fork();
```

- The `fork()` system call creates the copy of process that is executing.
- It is the only way in which a new process is created in UNIX.
- Returns:
 - for Parent process,
 - 0 - for Child process,
 - 1 - in case of error (errno indicates the error).
- Fork reasons:
 - Make a copy of itself to do another task simultaneously
 - Execute another program (See `exec` system call)

```
pid_t childPID; /* typedef int pid_t */
...
childPID = fork();
if (childPID < 0)
{
    perror("fork failed");
    exit(1);
}
else if (childPID > 0) /* This is Parent */
{
    /* Parent processing */
    exit(0);
}
else /* childPID == 0 */ /* This is Child */
{
    /* Child processing */
    exit(0);
}
```

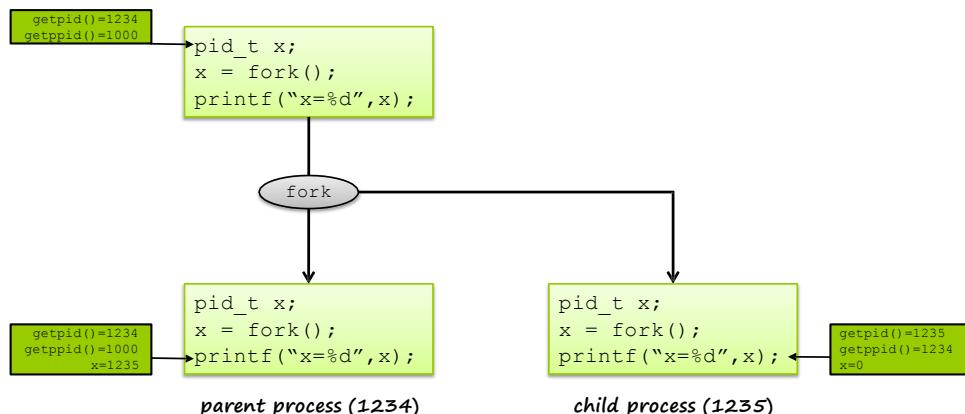
Fork System Call



Child Process has Shared Text and Copied Data.
It differs from Parent Process by following attributes:

- PID (new)
- Parent PID (= PID of Parent Process)
- Own copies of Parent file descriptors
- Time left until alarm signal reset to 0

The fork() System Call [2/3]



The Fork of Death!

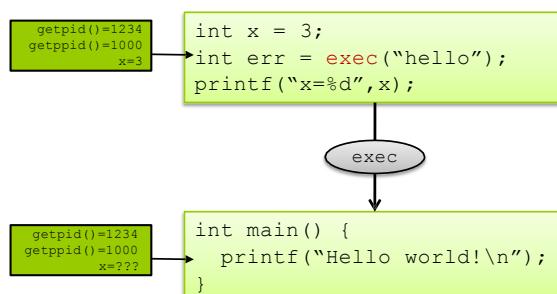
- You must be careful when using fork()!!
 - Very easy to create very harmful programs

```
while (1) {
    fork();
}
```

- Finite number of processes allowed
 - PIDs are 16-bit integers, maximum 65536 processes!
- Administrators typically take precautions
 - Limited quota on the number of processes per user
 - Try this at home ☺

The exec() System Call [1/4]

- exec() allows a process to switch from one program to another
 - Code/Data for that process are destroyed
 - Environment variables are kept the same
 - File descriptors are kept the same
 - New program's code is loaded, then started (from the beginning)
 - There is no way to return to the previously executed program!



The exec() System Call [2/4]

- fork() and exec() could be used separately
 - Imagine examples when this could happen?
- But most commonly they are used together:

```

if (fork()==0)      /* child */
{
    exec("hello"); /* load & execute new program */
    perror("Error calling exec()!\n");
    exit(1);
}
else /* parent */
{
    ...
}
  
```

Waiting for Children Processes [1/2]

- To block until some child process completes:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

- wait() waits for **any child** to complete
 - Also handles an already completed (zombie) child → returns instantly
 - Returns the PID of the completed child
 - status indicates the process' return status

Waiting for Children Processes [2/2]

- waitpid() gives you more control:
 - pid specifies which child to wait for (-1 means any)
 - option=WNOHANG makes the call return immediately, if no child has already completed (otherwise use option=0)

```
void sig_chld(int sig) {
    pid_t pid;
    int stat;
    while ( (pid=waitpid(-1, &stat, WNOHANG)) > 0 ) {
        printf("Child %d exited with status %d\n", pid, stat);
    }
    signal(sig, sig_chld);
}
```

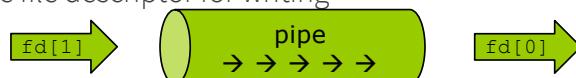
- Example: a SIGCHLD signal handler
- Question: Why is the while statement necessary?

Pipes [1/3]

- A pipe is a unidirectional communication channel between processes
 - Write data on one end – Read it at the other end
 - Bidirectional communication? Use TWO pipes.
- Creating a pipe

```
#include <unistd.h>
int pipe(int fd[2]);
```

- The return parameters are:
 - fd[0] is the file descriptor for reading
 - fd[1] is the file descriptor for writing



Pipes [2/3]

- Pipes are often used in combination with fork()

```
int main() {
    int pid, fd[2];
    char buf[64];

    if (pipe(fd)<0) exit(1);

    pid = fork();
    if (pid==0)          /* child */
    {
        close(fd[0]);   /* close reader */
        write(fd[1],"hello, world!",14);
    }
    else {              /* parent */
        close(fd[1]);   /* close writer */
        if (read(fd[0],buf,64)>0)
            printf("Received: %s\n", buf);
        waitpid(pid,NULL,0);
    }
}
```

Pipes [3/3]

- Pipes are used for example for shell commands like:
`sort foo | uniq | wc`
- Pipes can only link processes which have a common ancestor
 - Because children processes inherit the file descriptors from their parent
- What happens when two processes with no common ancestor want to communicate?
 - Named pipes (also called FIFO)
 - A special file behaves like a pipe
 - Assuming both processes agree on the pipe's filename, they can
- communicate
 - `mkfifo()` for creating a named pipe
 - `open()`, `read()`, `write()`, `close()`

Shared Memory

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
```

- Shared memory segments must be created, then attached to a process to be usable

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- Must detach and destroy them once done

```
int shmdt(const void *shmaddr);
```

Example: Shell Skeleton

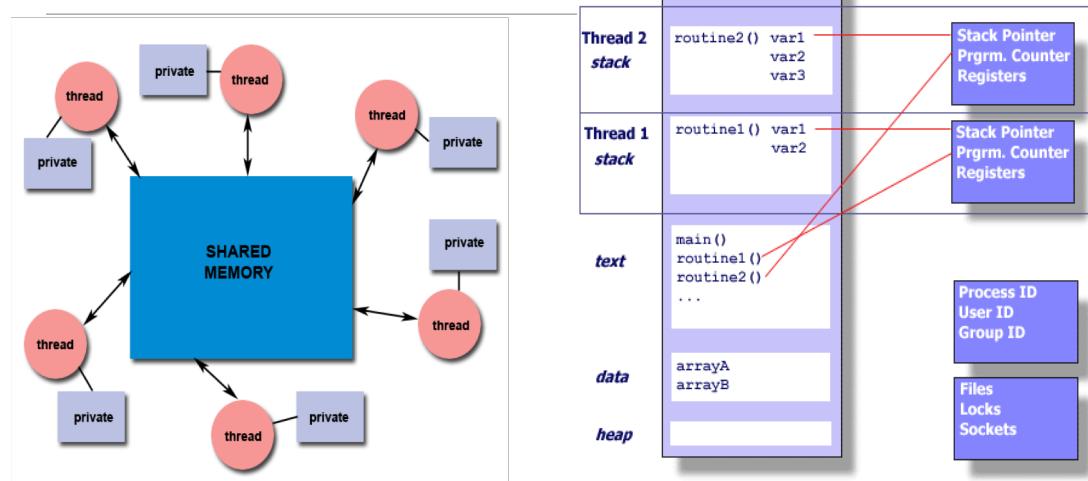
```
while (1) // Infinite
    print_prompt();
    read_command(command, parameters);
    if (fork()) {
        //Parent
        wait(&status);
    }
    else {
        execve(command, parameters, NULL);
    }
}
```

Pthreads

Shared Memory Model

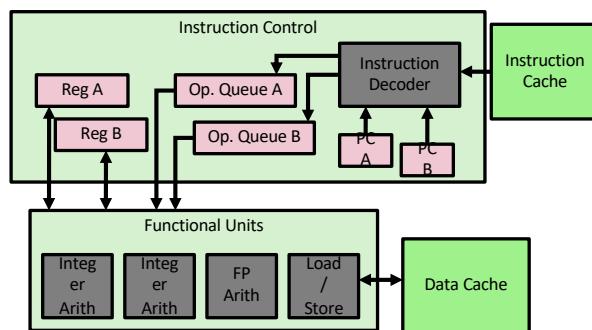
- All threads have equal priority read/write access to the same global (shared) memory.
- Threads also have their own private data.
 - Local variables for instance.
- Programmers are responsible for synchronizing any write access (protection) to globally shared data.

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.

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

Threads and Thread Attributes

Thread Attributes

```

pthread_attr_t attr;           // Declare a thread attribute
pthread_t tid;
pthread_attr_init(&attr);      // Initialize a thread attribute
pthread_attr_setdetachstate(&attr,    // Set the thread attribute
                           PTHREAD_CREATE_UNDETACHED);
pthread_create(&tid, &attr, start_func, NULL); // Use the attribute
                                                // to create a thread
pthread_join(tid, NULL);
pthread_attr_destroy(&attr);      // Destroy the thread attribute

```

Notes:

There are many other thread attributes. See a POSIX Threads manual for details.

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(pthread_t *tid, const pthread_attr_t *attr, void *(*function)(void *), void *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.

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 Creates?

```
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
```

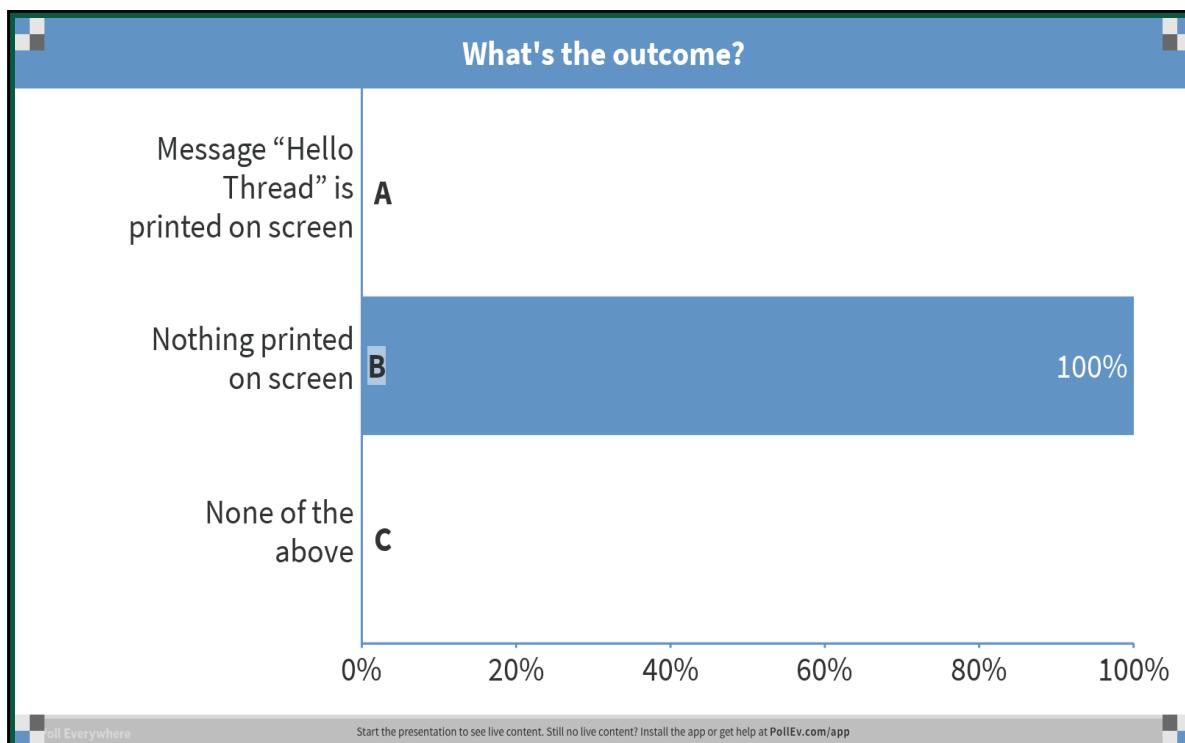
## Example: Thread Creation

```
#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);**

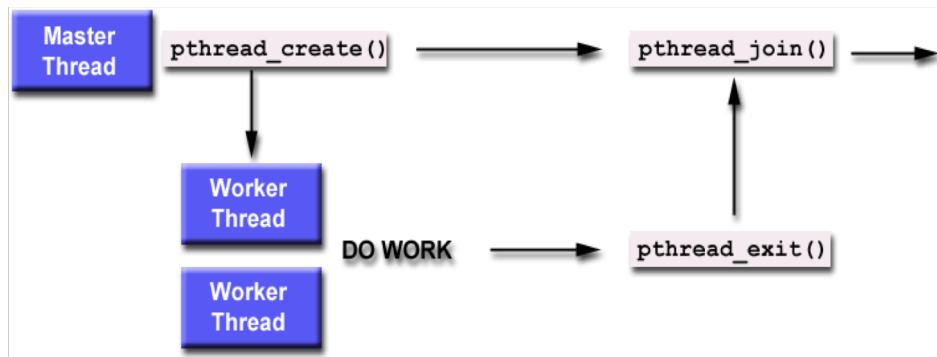
- `pthread_join` will block until the thread associated with the `pthread_t` handle has terminated.
  - There is no single function that can join multiple threads.
- The second parameter returns a pointer to a value from the thread being joined.
- `pthread_join()` can be used to wait for one thread to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

## **`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(
 void *status // 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.

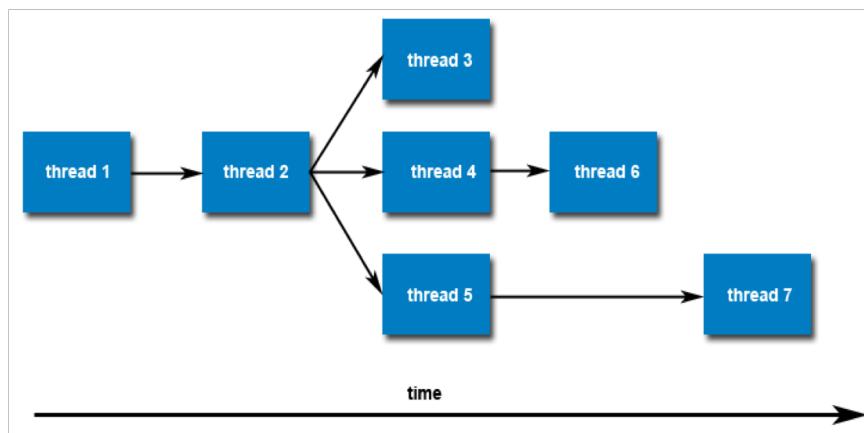
**Return value:**

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);
}

```

## Summation Example

- Sum numbers 0, ..., N-1
  - Should add up to  $(N-1) \cdot N / 2$
- Partition into K ranges
  - $\lfloor N/K \rfloor$  values each
  - Accumulate leftover values serially
- Method #1: All threads update single global variable
  - 1A: No synchronization
  - 1B: Synchronize with pthread semaphore
  - 1C: Synchronize with pthread mutex
    - o “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;
}

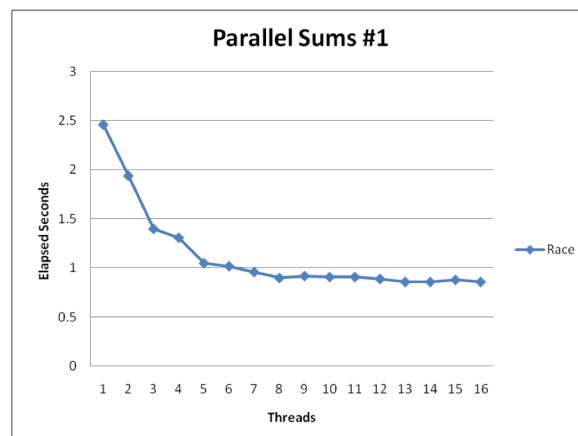
```

## Race Conditions

- Concurrent access of same variable by multiple threads
  - Read/Write conflict
  - Write/Write conflict
- Most common error in concurrent programs
- May not be apparent at all times

## Race Condition

- $N = 2^{30}$
- Best speedup = 2.86X
- Gets wrong answer when > 1 thread!



## 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;

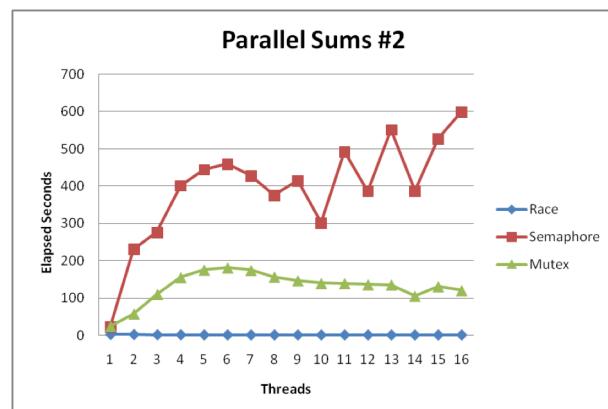
 for (i = start; i < end; i++) {
 sem_wait(&semaphore);
 global_sum += i;
 sem_post(&semaphore);
 }
 return NULL;
}
```

### 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

```

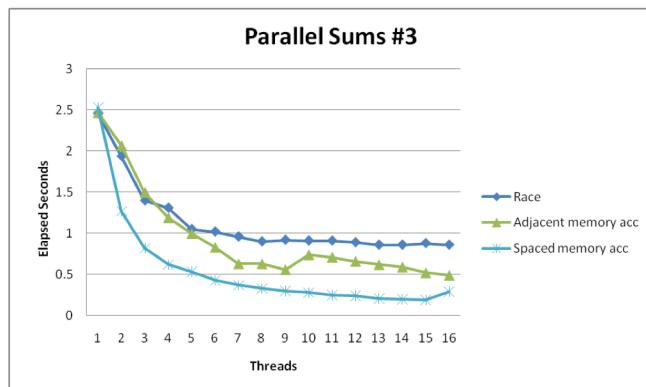
void *sum_global(void *vargp)
{
 int myid = *((int *)vargp);
 size_t start = myid * nelems_per_thread;
 size_t end = start + nelems_per_thread;
 size_t i;

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

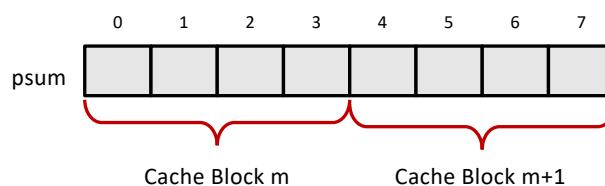
```

## 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?



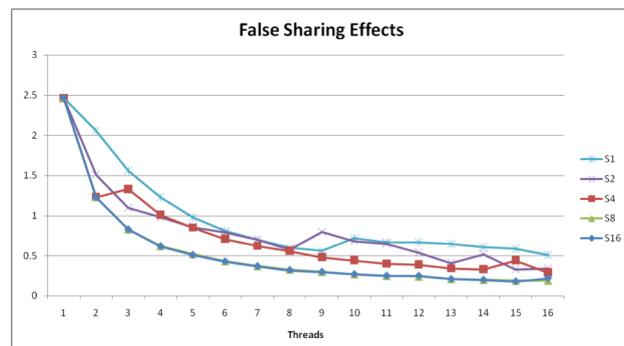
## False Sharing



- 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

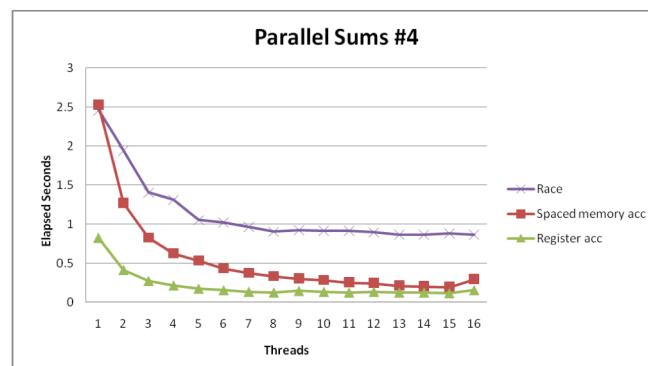


## Thread Function: Register Accumulation

```
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



## Computing $\pi$

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

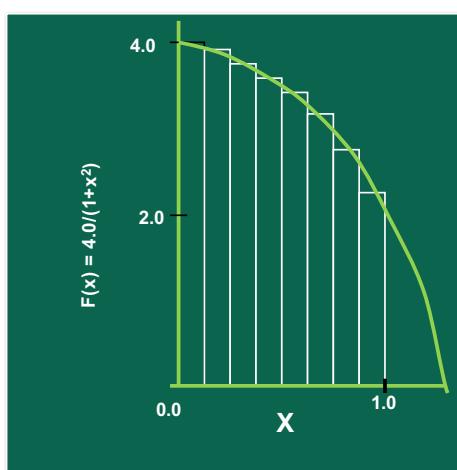
Mathematically, we know that:

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

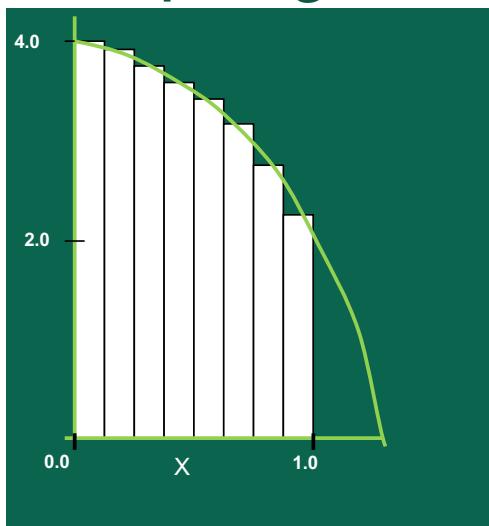
We can approximate the integral as a sum of rectangles:

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

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



## Computing $\pi$



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

## Computing $\pi$ : A Serial PI Program

```

static long num_steps = 1000000;
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;
}

```

## Assg 2: Computing $\pi$ (Due Next Week)

- 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;

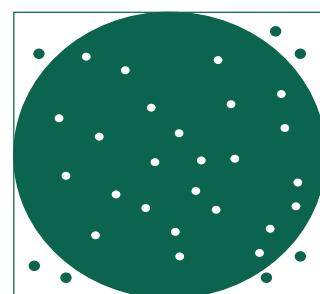
void main()
{ int i;
 double x;

 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 3: Computing $\pi$ and Compare!

```
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: 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);
}

```

Spring 2019

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

81



```

#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);
 }

 /* 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);
}

```

Spring 2019

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

82



## How to check whether a child thread has completed?

```

1 void *child(void *arg) {
2 printf("child\n");
3 // XXX how to indicate we are done?
4 return NULL;
5 }
6
7 int main(int argc, char *argv[]) {
8 printf("parent: begin\n");
9 pthread_t c;
10 Pthread_create(&c, NULL, child, NULL); // create child
11 // XXX how to wait for child?
12 printf("parent: end\n");
13 return 0;

```

## Possible Solution: Problem?

```

1 volatile int done = 0;
2
3 void *child(void *arg) {
4 printf("child\n");
5 done = 1;
6 return NULL;
7 }
8
9 int main(int argc, char *argv[]) {
10 printf("parent: begin\n");
11 pthread_t c;
12 Pthread_create(&c, NULL, child, NULL); // create child
13 while (done == 0)
14 ; // spin
15 printf("parent: end\n");
16 return 0;
17 }

```

## 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's 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 `cond` 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 initialized**  
**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
```

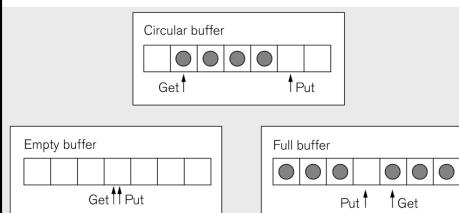
**Arguments:**

- 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



```

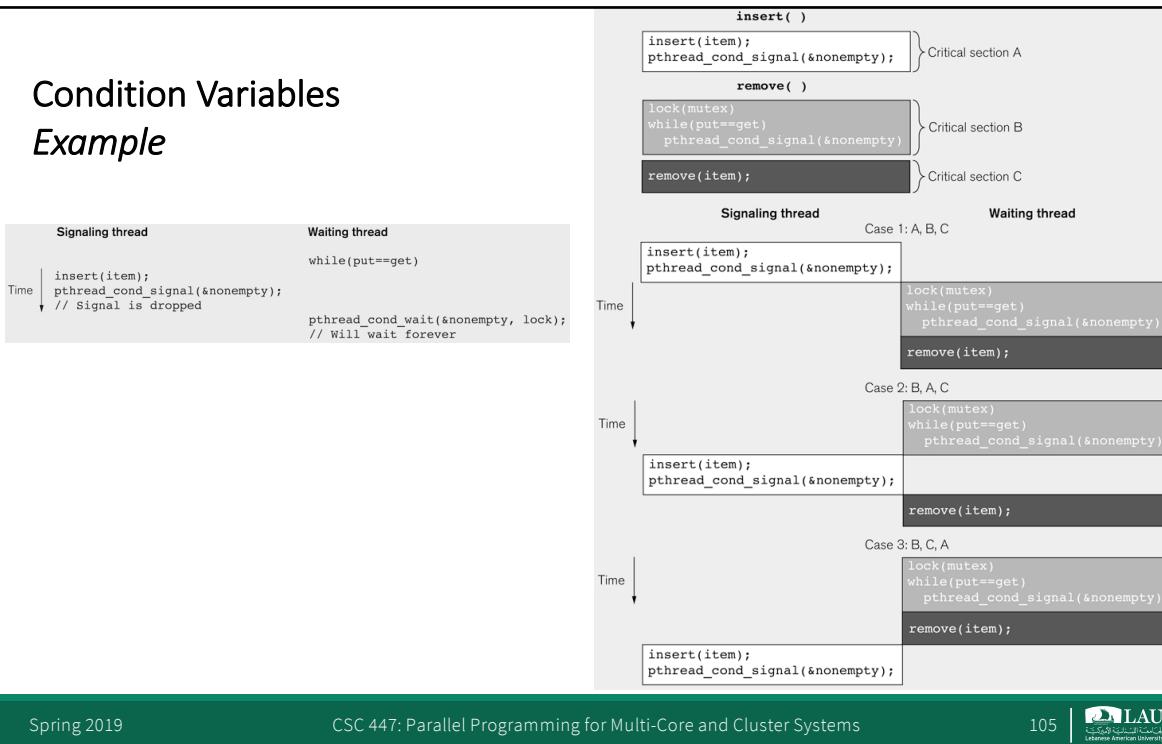
1 pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3 pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4 Item buffer[SIZE];
5 int put=0; // Buff index for next insert
6 int get=0; // Buff index for next remove
7
8 void insert(Item x) // Producer thread
9 {
10 pthread_mutex_lock(&lock);
11 while((put>get&&(put-get)==SIZE-1)|| // While buffer is
12 (put<get&&(put+get)==SIZE-1)) // full
13 {
14 pthread_cond_wait(&nonfull, &lock);
15 }
16 buffer[put]=x;
17 put=(put+1)%SIZE;
18 pthread_cond_signal(&nonempty);
19 pthread_mutex_unlock(&lock);
20 }
21
22 Item remove() // Consumer thread
23 {
24 Item x;
25 pthread_mutex_lock(&lock);
26 while(put==get) // While buffer is empty
27 {
28 pthread_cond_wait(&nonempty, &lock);
29 }
30 x=buffer[get];
31 get=(get+1)%SIZE;
32 pthread_cond_signal(&nonfull);
33 pthread_mutex_unlock(&lock);
34 return x;
35 }
```

## Use of Mutex and Signals

```

1 pthread_mutex_t lock=PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t nonempty=PTHREAD_COND_INITIALIZER;
3 pthread_cond_t nonfull=PTHREAD_COND_INITIALIZER;
4 Item buffer[SIZE];
5 int put=0; // Buff index for next insert
6 int get=0; // Buff index for next remove
7
8 void insert(Item x) // Producer thread
9 {
10 pthread_mutex_lock(&lock);
11 while((put>get&&(put-get)==SIZE-1)|| // While buffer is
12 (put<get&&(put+get)==SIZE-1)) // full
13 {
14 pthread_cond_wait(&nonfull, &lock);
15 }
16 buffer[put]=x;
17 put=(put+1)%SIZE;
18 pthread_cond_signal(&nonempty);
19 pthread_mutex_unlock(&lock);
20 }
21
22 Item remove() // Consumer thread
23 {
24 Item x;
25 pthread_mutex_lock(&lock);
26 while(put==get) // While buffer is empty
27 {
28 pthread_cond_wait(&nonempty, &lock);
29 }
30 x=buffer[get];
31 get=(get+1)%SIZE;
32 pthread_cond_signal(&nonfull);
33 pthread_mutex_unlock(&lock);
34 return x;
35 }
```

## 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

# Quick Sort

A More Interesting Example

Spring 2019

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

107



## A More Interesting Example

- Sort set of N random numbers
- Multiple possible algorithms
  - Use parallel version of quicksort
- Sequential quicksort of set of values X
  - Choose “pivot” p from X
  - Rearrange X into
    - L: Values  $\leq p$
    - R: Values  $\geq p$
  - Recursively sort L to get L'
  - Recursively sort R to get R'
  - Return L' : p : R'

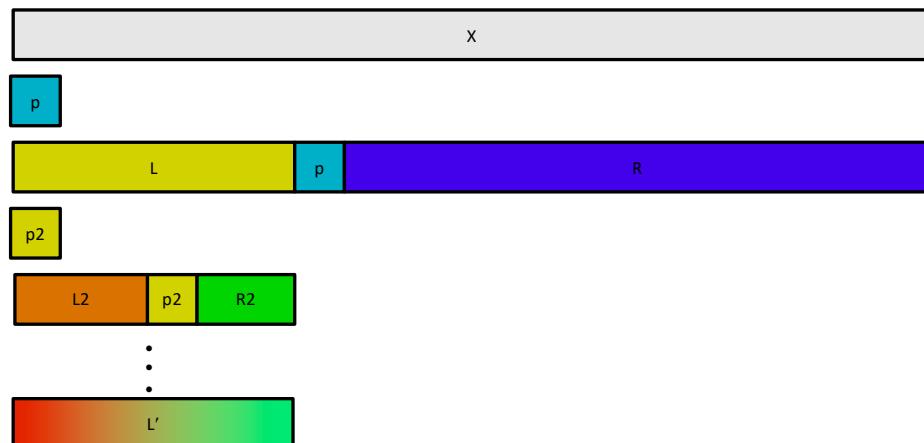
Spring 2019

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

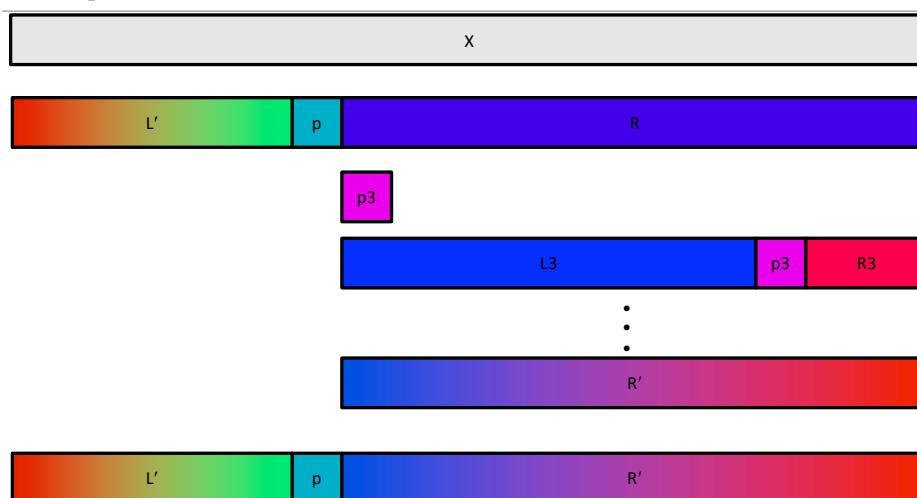
108



## Sequential Quicksort Visualized



## Sequential Quicksort Visualized



## Sequential Quicksort Code

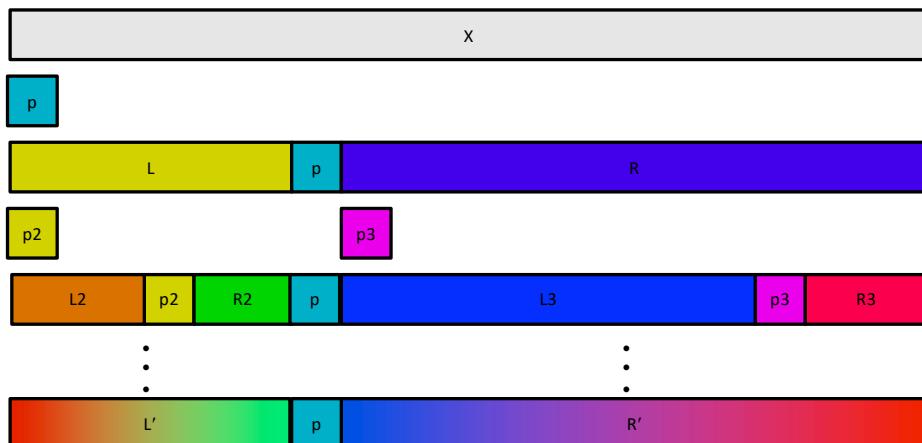
- Sort  $nele$  elements starting at base
  - Recursively sort L or R if has more than one element

```
void qsort_serial(data_t *base, size_t nele) {
 if (nele <= 1)
 return;
 if (nele == 2) {
 if (base[0] > base[1])
 swap(base, base+1);
 return;
 }
 /* Partition returns index of pivot */
 size_t m = partition(base, nele);
 if (m > 1)
 qsort_serial(base, m);
 if (nele-1 > m+1)
 qsort_serial(base+m+1, nele-m-1);
}
```

## Parallel Quicksort

- Parallel quicksort of set of values X
  - If  $N \leq N_{thresh}$ , do sequential quicksort
  - Else
    - Choose “pivot”  $p$  from X
    - Rearrange X into
      - L: Values  $\leq p$
      - R: Values  $\geq p$
    - Recursively spawn separate threads
      - Sort L to get  $L'$
      - Sort R to get  $R'$
    - Return  $L' : p : R'$
- Degree of parallelism
  - Top-level partition: none
  - Second-level partition:  $2X$
  - ...

## Parallel Quicksort Visualized



## Parallel Quicksort Data Structures

- Data associated with each sorting task
  - **base:** Array start
  - **nele:** Number of elements
  - **tid:** Thread ID
- Generate list of tasks
  - Must protect by mutex

```
/* Structure that defines sorting task */
typedef struct {
 data_t *base;
 size_t nele;
 pthread_t tid;
} sort_task_t;

volatile int ntasks = 0;
volatile int ctasks = 0;
sort_task_t **tasks = NULL;
sem_t tmutex;
```

## Parallel Quicksort Initialization

```
static void init_task(size_t nele) {
 ctasks = 64;
 tasks = (sort_task_t **) Calloc(ctasks, sizeof(sort_task_t *));
 ntasks = 0;
 Sem_init(&tmutex, 0, 1);
 nele_max_serial = nele / serial_fraction;
}
```

- Task queue dynamically allocated
- Set Nthresh = N/F:
  - N Total number of elements
  - F Serial fraction
    - o Fraction of total size at which shift to sequential quicksort

## Parallel Quicksort: Accessing Task Queue

```
static sort_task_t *new_task(data_t *base, size_t nele) {
 P(&tmutex);
 if (ntasks == ctasks) {
 ctasks *= 2;
 tasks = (sort_task_t **) Realloc(tasks, ctasks * sizeof(sort_task_t *));
 }
 int idx = ntasks++;
 sort_task_t *t = (sort_task_t *) Malloc(sizeof(sort_task_t));
 tasks[idx] = t;
 V(&tmutex);
 t->base = base;
 t->nele = nele;
 t->tid = (pthread_t) 0;
 return t;
}
```

- Dynamically expand by doubling queue length
  - Generate task structure dynamically (consumed when reap thread)
- Must protect all accesses to queue & ntasks by mutex

## Parallel Quicksort: Top-Level Function

```
void tqsrt(data_t *base, size_t nele) {
 int i;
 init_task(nele);
 tqsrt_helper(base, nele);
 for (i = 0; i < get_ntasks(); i++) {
 P(&tmutex);
 sort_task_t *t = tasks[i];
 V(&tmutex);
 Pthread_join(t->tid, NULL);
 free((void *) t);
 }
}
```

- Actual sorting done by tqsrt\_helper
- Must reap all of the spawned threads
  - All accesses to task queue & ntasks guarded by mutex

## Parallel Quicksort: Recursive function

```
void tqsrt_helper(data_t *base, size_t nele) {
 if (nele <= nele_max_serial) {
 /* Use sequential sort */
 qsort_serial(base, nele);
 return;
 }
 sort_task_t *t = new_task(base, nele);
 Pthread_create(&t->tid, NULL, sort_thread, (void *) t);
}
```

- If below Nthresh, call sequential quicksort
- Otherwise create sorting task

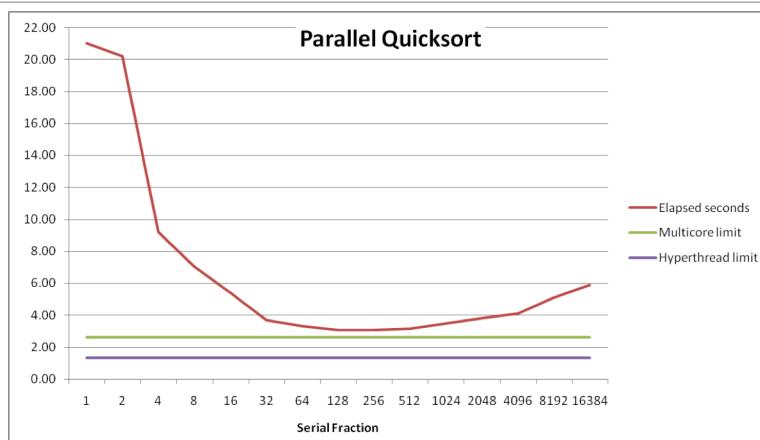
## Parallel Quicksort: Sorting Task Function

```
static void *sort_thread(void *vargp) {
 sort_task_t *t = (sort_task_t *) vargp;
 data_t *base = t->base;
 size_t nele = t->nele;
 size_t m = partition(base, nele);
 if (m > 1)
 tqsort_helper(base, m);
 if (nele-1 > m+1)
 tqsort_helper(base+m+1, nele-m-1);
 return NULL;
}
```

- Same idea as sequential quicksort

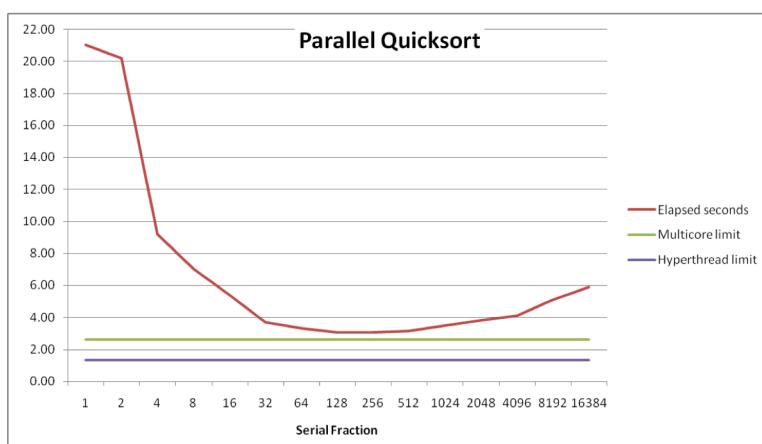
## Parallel Quicksort Performance

- Sort  $2^{37}$  (134,217,728) random values
- Best speedup = 6.84X



## Parallel Quicksort Performance

- Good performance over wide range of fraction values
  - F too small: Not enough parallelism
  - F too large: Thread overhead + run out of thread memory



## Implementation Subtleties

- Task set data structure
  - Array of structs
 

```
sort_task_t *tasks;
```

    - new\_task returns pointer or integer index
  - Array of pointers to structs
 

```
sort_task_t **tasks;
```

    - new\_task dynamically allocates struct and returns pointer
- Reaping threads
  - Can we be sure the program won't terminate prematurely?

## Amdahl's Law & Parallel Quicksort

- Sequential bottleneck
  - Top-level partition: No speedup
  - Second level:  $\leq 2X$  speedup
  - $k^{\text{th}}$  level:  $\leq 2^{k-1}X$  speedup
- Implications
  - Good performance for small-scale parallelism
  - Would need to parallelize partitioning step to get large-scale parallelism
    - Parallel Sorting by Regular Sampling
      - H. Shi & J. Schaeffer, J. Parallel & Distributed Computing, 1992

## Lessons Learned

- Must have strategy
  - Partition into K independent parts
  - Divide-and-conquer
- Inner loops must be synchronization free
  - Synchronization operations very expensive
- Watch out for hardware artifacts
  - Sharing and false sharing of global data
- You can do it!
  - Achieving modest levels of parallelism is not difficult

## Additional Notes

Spring 2019

CSC 447: Parallel Programming for Multi-Core and Cluster Systems 125

## Producer/Consumer using pthreads

Spring 2019

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

126

```

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

/* Producer/consumer program illustrating conditional variables */

/* Size of shared buffer */
#define BUF_SIZE 3

int buffer[BUF_SIZE]; /*shared buffer */
int add=0; /* place to add next element */
int rem=0; /* place to remove next element */
int num=0; /* number elements in buffer */
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER; /* mutex lock for buffer */
pthread_cond_t c_cons=PTHREAD_COND_INITIALIZER; /* consumer waits on this cond var */
pthread_cond_t c_prod=PTHREAD_COND_INITIALIZER; /* producer waits on this cond var */

void *producer(void *param);
void *consumer(void *param);

```

```

main (int argc, char *argv[])
{
 pthread_t tid1, tid2; /* thread identifiers */
 int i;

 /* create the threads; may be any number, in general */
 if (pthread_create(&tid1,NULL,producer,NULL) != 0) {
 fprintf (stderr, "Unable to create producer thread\n");
 exit (1);
 }
 if (pthread_create(&tid2,NULL,consumer,NULL) != 0) {
 fprintf (stderr, "Unable to create consumer thread\n");
 exit (1);
 }
 /* wait for created thread to exit */
 pthread_join(tid1,NULL);
 pthread_join(tid2,NULL);
 printf ("Parent quitting\n");
}

```

```

/* Produce value(s) */
void *producer(void *param)
{
 int i;
 for (i=1; i<=20; i++) {
 /* Insert into buffer */
 pthread_mutex_lock (&m);
 if (num > BUF_SIZE) exit(1); /* overflow */
 while (num == BUF_SIZE) /* block if buffer is full */
 pthread_cond_wait (&c_prod, &m);
 /* if executing here, buffer not full so add element */
 buffer[add] = i;
 add = (add+1) % BUF_SIZE;
 num++;
 pthread_mutex_unlock (&m);
 pthread_cond_signal (&c_cons);
 printf ("producer: inserted %d\n", i); fflush (stdout);
 }
 printf ("producer quiting\n"); fflush (stdout);
}

```

```

/* Consume value(s); Note the consumer never terminates */
void *consumer(void *param)
{
 int i;
 while (1) {
 pthread_mutex_lock (&m);
 if (num < 0) exit(1); /* underflow */
 while (num == 0) /* block if buffer empty */
 pthread_cond_wait (&c_cons, &m);
 /* if executing here, buffer not empty so remove element */
 i = buffer[rem];
 rem = (rem+1) % BUF_SIZE;
 num--;
 pthread_mutex_unlock (&m);
 pthread_cond_signal (&c_prod);
 printf ("Consume value %d\n", i); fflush(stdout);
 }
}

```

# Semaphores

Spring 2019

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

131



## Semaphores

- Synchronization object that keeps a count
  - Represent the number of available resources
  - Formalized by Edsgar Dijkstra
- Two operations on semaphores
  - Wait [P(s)]: Thread waits until  $s > 0$ , then  $s = s - 1$
  - Post [V(s)]:  $s = s + 1$

Spring 2019

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

132



## POSIX Semaphores

- Not part of the POSIX Threads specification
  - Defined in POSIX.1b
- Check for system support of semaphores before use
  - Is **\_POSIX\_SEMAPORES** defined in <unistd.h>?
  - If available, threads within a process can use semaphores
  - Use header file <semaphore.h>
- New type
  - **sem\_t**
    - the semaphore variable

### **sem\_init**

```
int sem_init(sem, pshared, value);
```

**sem\_t \*sem**

- counting semaphore to be initialized

**int pshared**

- if non-zero, semaphore can be shared across processes

**unsigned int value**

- initial value of semaphore

## sem\_init Explained

- Initializes the semaphore object
- If **pshared** is zero, semaphore can only be used by threads within the calling process
  - If non-zero, semaphore can be used between processes
- The value parameter sets the initial semaphore count value

**EINVAL – sem is not valid semaphore**  
**EPERM – process lacks appropriate privilege**  
**ENOSYS – semaphores not supported**

## sem\_wait

```
int sem_wait(sem);
```

**sem\_t \*sem**

- counting semaphore to decrement or wait

## sem\_wait Explained

- If semaphore count is greater than zero
  - Decrement count by one (1)
  - Proceed with code following
- Else, if semaphore count is zero
  - Thread blocks until value is greater than zero

**EINVAL** – sem is not valid semaphore  
**EDEADLK** – deadlock condition was detected  
**ENOSYS** – semaphores not supported

## sem\_post

```
int sem_post(sem);
```

**sem\_t \*sem**

- counting semaphore to be incremented

## sem\_post Explained

- Post a wakeup to semaphore
- If one or more threads are waiting, release one
  - Otherwise, increment semaphore count by one (1)

**EINVAL – sem is not valid semaphore**  
**ENOSYS – semaphores not supported**

## Semaphore Uses

- Control access to limited size data structures
  - Queues, stacks, deques
  - Use count to enumerate available elements
- Throttle number of active threads within a region
- Binary semaphore [0,1] can act as mutex

## Semaphore Cautions

- No ownership of semaphore
- Any thread can release a semaphore, not just the last thread to wait
  - Use good programming practice to avoid
- No concept of abandoned semaphore
  - If thread terminates before post, semaphore increment may be lost
  - Deadlock

## Example: Semaphore as Mutex

- Main thread opens input file, waits for thread termination
- Threads will
  - Read line from input file
  - Count all five letter words in line

## Example: Main

```

sem_t hSem1, hSem2;
FILE *fd;
int fiveLetterCount = 0;

main() {
 pthread_t hThread[NUMTHREADS];

 sem_init(&hSem1, 0, 1); // Binary semaphore
 sem_init(&hSem2, 0, 1); // Binary semaphore

 fd = fopen("InFile", "r"); // Open file for read

 for (int i = 0; i < NUMTHREADS; i++)
 pthread_create(&hThread[i], NULL, CountFives, NULL);
 for (int i = 0; i < NUMTHREADS; i++)
 pthread_join(hThread[i], NULL);

 fclose(fd);

 printf("Number of five letter words is %d\n", fiveLetterCount);
}

```

## Example: Semaphores

```

void * CountFives(void *arg) {
 int bDone = 0;
 char inLine[132]; int lCount = 0;

 while (!bDone)
 {
 sem_wait(*hSem1); // access to input
 bDone = (GetNextLine(fd, inLine) == EOF);
 sem_post(*hSem1);
 if (!bDone)
 if (lCount = GetFiveLetterWordCount(inLine)) {
 sem_wait(*hSem2); // update global
 fiveLetterCount += lCount;
 sem_post(*hSem2);
 }
 }
}

```

## Thread-specific Data

- Another means for “local” storage

**`pthread_key_create`**

- Create thread-specific key for all threads

**`pthread_setspecific`**

- Associate thread-specific value with given key

**`pthread_getspecific`**

- Return current data value associated with key

**`pthread_key_delete`**

- Delete a data key

## Thread Attribute Functions

**`pthread_attr_init`**

- Initialize attribute object to default settings

**`pthread_attr_destroy`**

- Delete attribute object

**`pthread_{get|set}detachstate`**

- Return or set thread’s detach state

**`pthread_{get|set}stackaddr`**

- Return or set the stack address of thread

**`pthread_{get|set}stacksize`**

- Return or set the stack size of thread

## Mutex Attribute Functions

- **`pthread_mutexattr_init`**
  - Initialize mutex attribute object to defaults
- **`pthread_mutexattr_destroy`**
  - Delete mutex attribute object
- **`pthread_mutexattr_{get|set}pshared`**
  - Return or set whether mutex is shared between processes

## Condition Attribute Functions

- **`pthread_condattr_init`**
  - Initialize condition variable attribute object
- **`pthread_condattr_destroy`**
  - Destroy condition variable attribute object
- **`pthread_condattr_{get|set}pshared`**
  - Return or set whether condition variable is shared between processes