Lecture 08: Programming with PThreads: PThreads basics, Mutual Exclusion and Locks, and Examples

CSCE 790: Parallel Programming Models for Multicore and Manycore Processors

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

OpenMP: Worksharing Constructs

Sequential code

```
for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel shared (a, b)

{

      int id, i, Nthrds, istart, iend;
      id = omp_get_thread_num();
      Nthrds = omp_get_num_threads();
      istart = id * N / Nthrds;
      iend = (id+1) * N / Nthrds;
      for(i=istart;i<iend;i++)   { a[i] = a[i] + b[i]; }
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel shared (a, b) private (i)
#pragma omp for schedule(static)
      for(i=0;i<N;i++)   { a[i] = a[i] + b[i]; }
```

PThreads

- Processing element abstraction for software
  - PThreads
  - OpenMP/Cilk/others runtime use PThreads for their implementation

- The foundation of parallelism from computer system

- Topic Overview
  - Thread basics and the POSIX Thread API
  - Thread creation, termination and joining
  - Thread safety
  - Synchronization primitives in PThreads

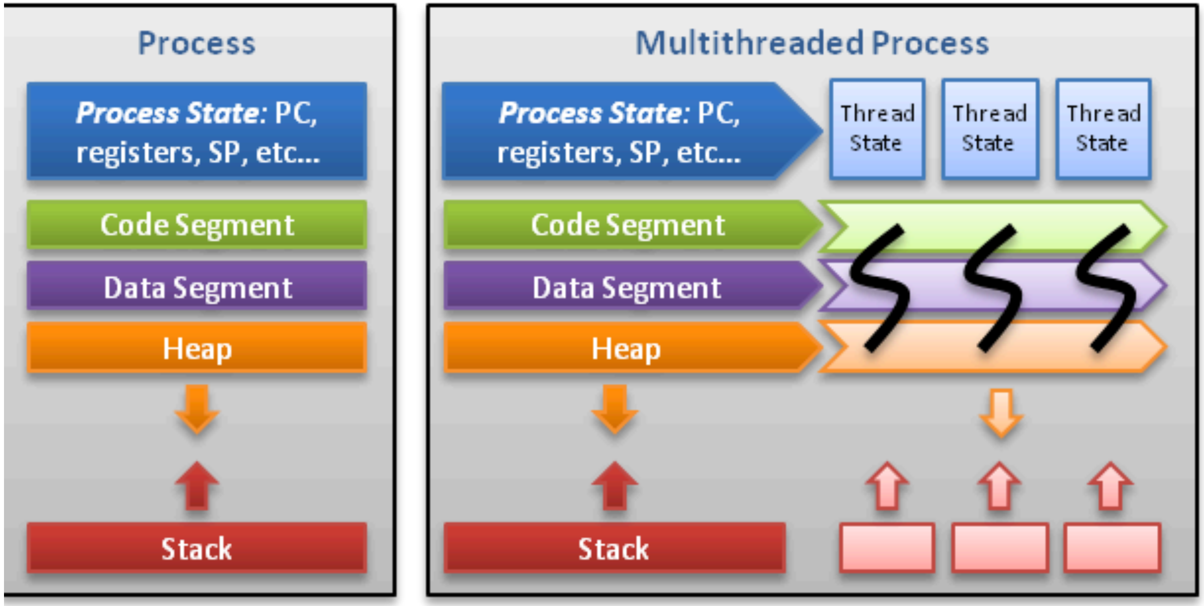
OS Review: Processes

- **processes** contain information about program resources and program execution state, including:
  - Process ID, process group ID, user ID, and group 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).

- When we run a program, a process is created
  - E.g. ./a.out, ./axpy, etc
  - fork () system call

Threads

- Threads use, and exist within, the process resources
- Scheduled and run as independent entities
- Duplicate only the bare essential resources that enable them to exist as executable code

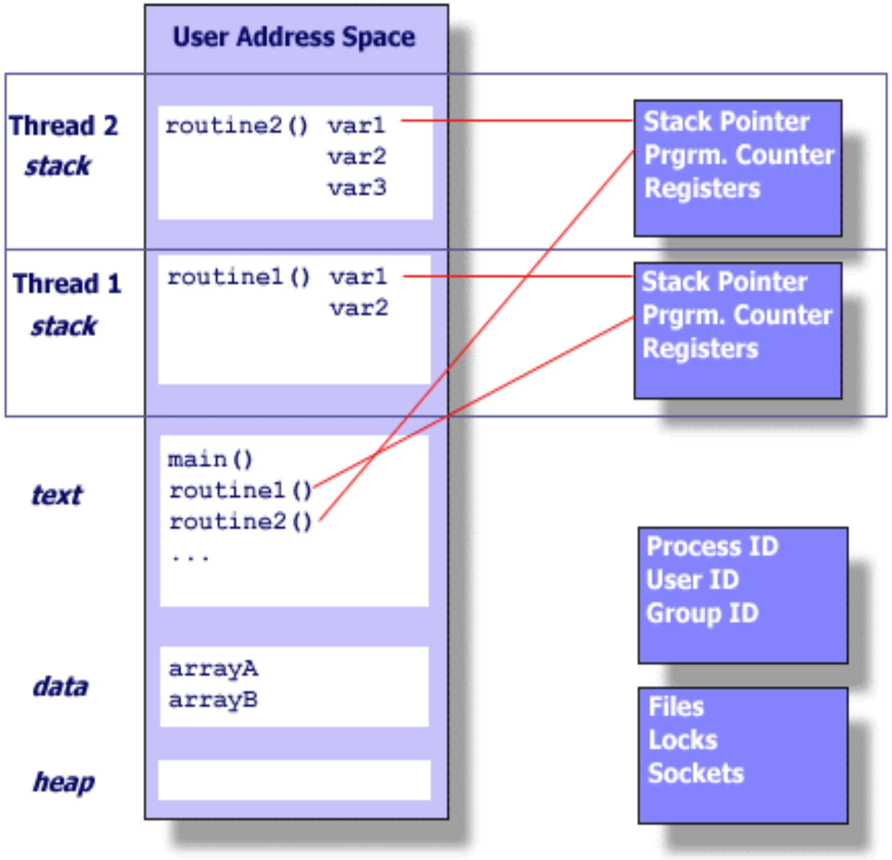


Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

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

Threads

- A thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.
- Multiple threads share the process resources
- A thread dies if the process dies
- "lightweight" for creating and terminating threads that for processes

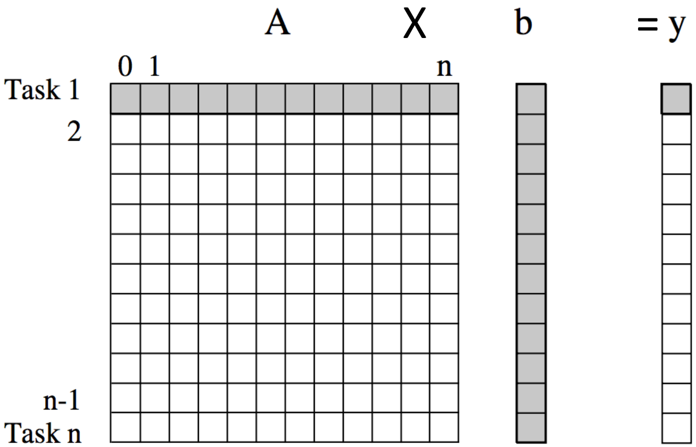


What is a Thread in Real

- OS view
  - An independent stream of instructions that can be scheduled to run by the OS.
- **Software developer view**
  - A "**procedure**" that runs independently from the main program
    - Imagine multiple such procedures of main run simultaneously and/or independently
  - Sequential program: a single stream of instructions in a program.
  - Multi-threaded program: a program with multiple streams
    - Multiple threads are needed to use multiple cores/CPUs

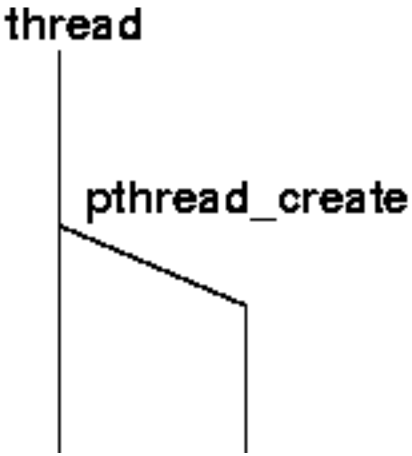
Thread as “function instance”

A *thread* is a single stream of control in the flow of a program:



```
for (i = 0; i < n; i++)
  y[i] = dot_product(row(A, i),b);
```

```
for (i = 0; i < n; i++)
   y[i] = create_thread(dot_product(row(A, i), b));
```



- think of the thread as an instance of a function that returns before the function has finished executing.

POSIX threads (PThreads)

- Threads used to implement parallelism in shared memory multiprocessor systems, such as SMPs
- Historically, hardware vendors have implemented their own proprietary versions of threads
  - Portability a concern for software developers.
- For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard.
  - Implementations that adhere to this standard are referred to as **POSIX threads**

The POSIX Thread API

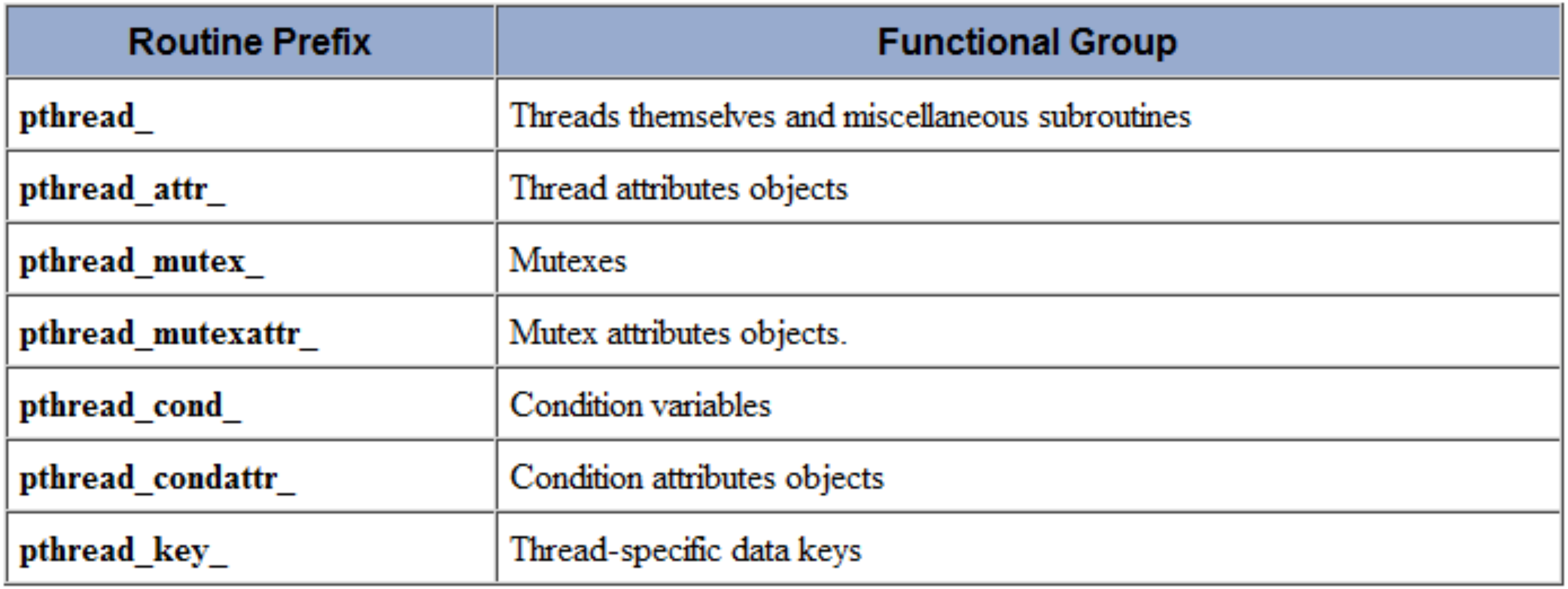
- Commonly referred to as PThreads, POSIX has emerged as the standard threads API, supported by most vendors.
  - Implemented with a pthread.h header/include file and a thread library

- Functionalities
  - Thread management, e.g. creation and joining
  - Thread synchronization primitives
    - Mutex
    - Condition variables
    - Reader/writer locks
    - Pthread barrier
  - Thread-specific data

- The concepts discussed here are largely independent of the API
  - Applied to other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

PThread API

- `#include <pthread.h>`



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

- **`gcc -lpthread`**

Thread Creation

- Initially, main() program comprises a single, default thread
  - All other threads must be explicitly created

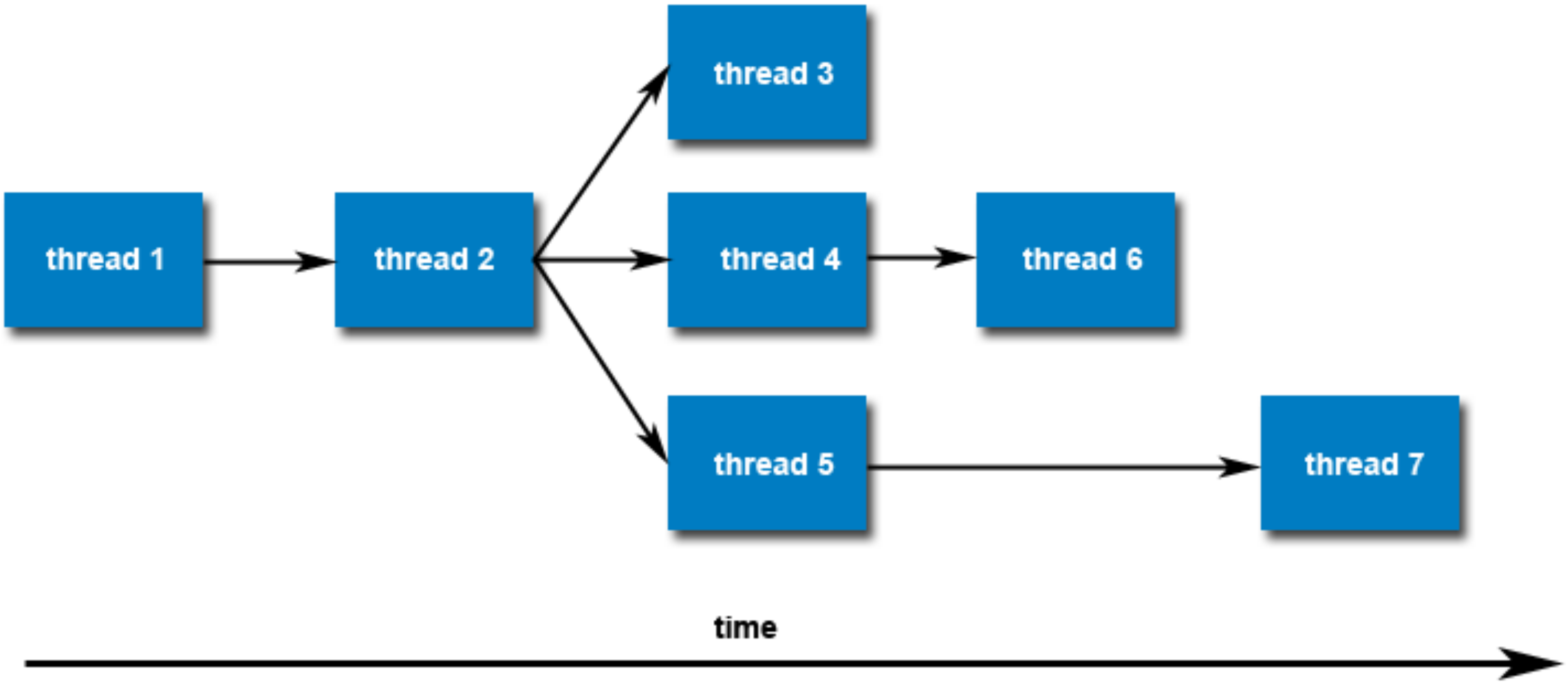
```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine)(void *),
    void * arg);
```

- **thread**: An *opaque*, unique identifier for the new thread returned by the subroutine
- **attr**: An *opaque* attribute object that may be used to set thread attributes
  You can specify a thread attributes object, or NULL for the default values
- **start_routine**: the C routine that the thread will execute once it is created
- **arg:** A single argument that may be passed to *start_routine*. It must be passed by reference as a pointer cast of type void. NULL may be used if no argument is to be passed.

Opaque object: A letter is an opaque object to the mailman, and sender and receiver know the information.

Thread Creation

- **pthread_create** creates a new thread and makes it executable, i.e. run immediately in theory
  - can be called any number of times from anywhere within your code
- Once created, threads are peers, and may create other threads
- There is no implied hierarchy or dependency between threads



Example 1: pthread_create

```
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *thread_id) {
    long tid = (long)thread_id;
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    long t;

    for(t=0;t<NUM_THREADS;t++) {
       printf("In main: creating thread %ld\n", t);
       int rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t );
       if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
       }
    }
  pthread_exit(NULL);
}
```

```
One possible output:

In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #0!
In main: creating thread 4
Hello World! It's me, thread #1!
Hello World! It's me, thread #3!
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
```

Terminating Threads

- **`pthread_exit`** is used to explicitly exit a thread
  - Called after a thread has completed its work and is no longer required to exist
- If `main()` finishes before the threads it has created
  - If exits with pthread_exit(), the other threads will continue to execute
  - Otherwise, they will be automatically terminated when `main()` finishes
- The programmer may optionally specify a termination *status*, which is stored as a void pointer for any thread that may join the calling thread
- Cleanup: the `pthread_exit()` routine does not close files
  - Any files opened inside the thread will remain open after the thread is terminated

Thread Attribute

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

- Attribute contains details about
  - whether scheduling policy is inherited or explicit
  - scheduling policy, scheduling priority
  - stack size, stack guard region size

- `pthread_attr_init` and `pthread_attr_destroy` are used to initialize/destroy the thread attribute object
- Other routines are then used to query/set specific attributes in the thread attribute object

Passing Arguments to Threads

- The **`pthread_create()`** routine permits the programmer to pass **one** argument to the thread start routine
- For cases where multiple arguments must be passed:
  - Create a structure which contains all of the arguments
  - Then pass a pointer to the object of that structure in the `pthread_create()` routine.
  - All arguments must be passed by reference and cast to (void *)

- Make sure that all passed data is thread safe: data racing
  - it can not be changed by other threads
  - It can be changed in a determinant way
    - Thread coordination

Example 2: Argument Passing

```
#include <pthread.h>
#define NUM_THREADS 8

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

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg) {
   int taskid;
   char *hello_msg;

   sleep(1);
   struct thread_data *my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   hello_msg = my_data->message;
   printf("Thread %d: %s\n", taskid, hello_msg);
   pthread_exit(NULL);
}
```

Example 2: Argument Passing

```
int main(int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int t;
   char *messages[NUM_THREADS];
   messages[0] = "English: Hello World!";
   messages[1] = "French: Bonjour, le monde!";
   messages[2] = "Spanish: Hola al mundo";
   messages[3] = "Klingon: Nuq neH!";
   messages[4] = "German: Guten Tag, Welt!";
   messages[5] = "Russian: Zdravstvytye, mir!";
   messages[6] = "Japan: Sekai e konnichiwa!";
   messages[7] = "Latin: Orbis, te saluto!";

   for(t=0;t<NUM_THREADS;t++) {
      struct thread_data * thread_arg = &thread_data_array[t];
      thread_arg->thread_id = t;
      thread_arg->message = messages[t];
      pthread_create(&threads[t], NULL, PrintHello, (void *) thread_arg);
   }
   pthread_exit(NULL);
}
```

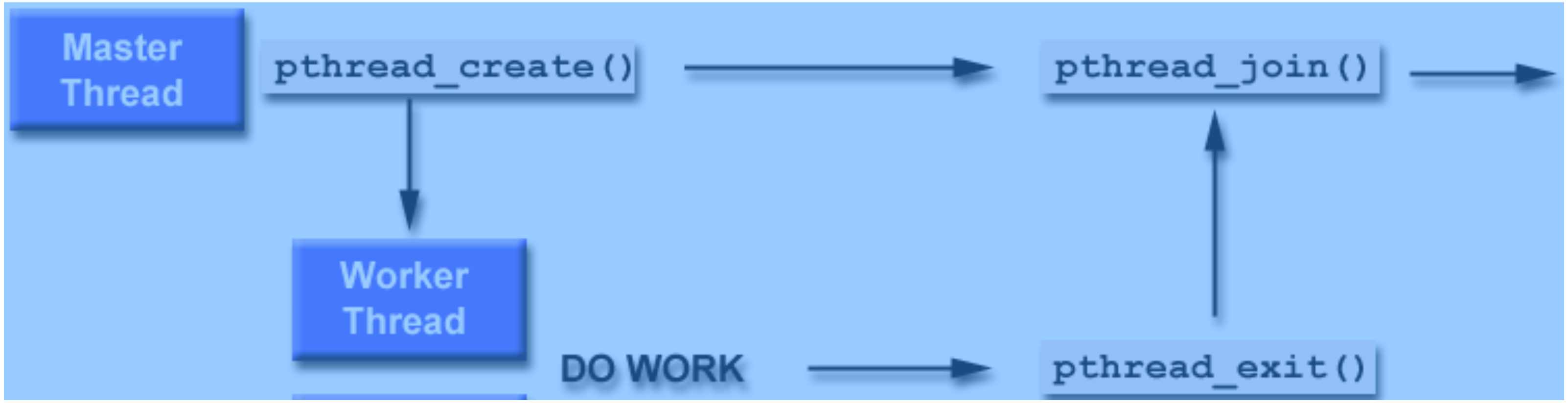
```
Thread 3: Klingon: Nuq neH!
Thread 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 2: Spanish: Hola al mundo
Thread 5: Russian: Zdravstvytye, mir!
Thread 4: German: Guten Tag, Welt!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
```

Wait for Thread Termination

Suspend execution of calling thread until thread terminates

```
#include <pthread.h>
int pthread_join(
   pthread_t thread,
    void **value_ptr);
```

- **thread:** the joining thread
- **value_ptr**: ptr to location for return code a terminating thread passes to pthread_exit



- It is a logical error to attempt simultaneous multiple joins on the same thread

Example 3: PThreads Joining

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

void *BusyWork(void *t) {
    int i;
    long tid = (long)t;
    double result=0.0;
    printf("Thread %ld starting...\n",tid);
    for (i=0; i<1000000; i++)  {
        result = result + sin(i) * tan(i);
    }
    printf("Thread %ld done. Result = %e\n",tid, result);
    pthread_exit((void*) t);
}
```

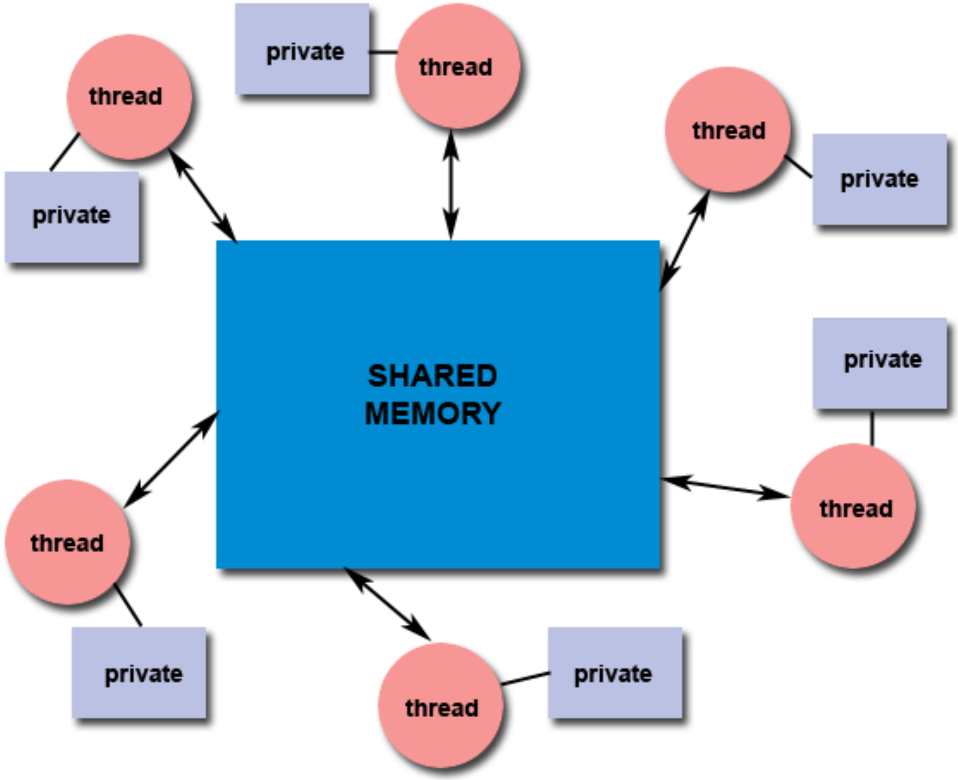
Example 3: PThreads joining

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

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

   for(t=0; t<NUM_THREADS; t++) {
      printf("Main: creating thread %ld\n", t);
      pthread_create(&thread[t], &attr, BusyWork, (v
   }
   /* Free attribute and wait for the other threads */
   pthread_attr_destroy(&attr);
   for(t=0; t<NUM_THREADS; t++) {
      pthread_join(thread[t], &status);
      printf(“Main: joined with thread %ld, status: %ld\n", t, (long)status);
   }
   printf("Main: program completed. Exiting.\n");
   pthread_exit(NULL);
}
```

```
Main: creating thread 0
Main: creating thread 1
Thread 0 starting...
Main: creating thread 2
Thread 1 starting...
Main: creating thread 3
Thread 2 starting...
Thread 3 starting...
Thread 1 done. Result = -3.153838e+06
Thread 0 done. Result = -3.153838e+06
Main: joined with thread 0, status: 0
Main: joined with thread 1, status: 1
Thread 2 done. Result = -3.153838e+06
Main: joined with thread 2, status: 2
Thread 3 done. Result = -3.153838e+06
Main: joined with thread 3, status: 3
Main: program completed. Exiting.
```

Shared Memory and Threads

- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access (protecting) globally shared data.



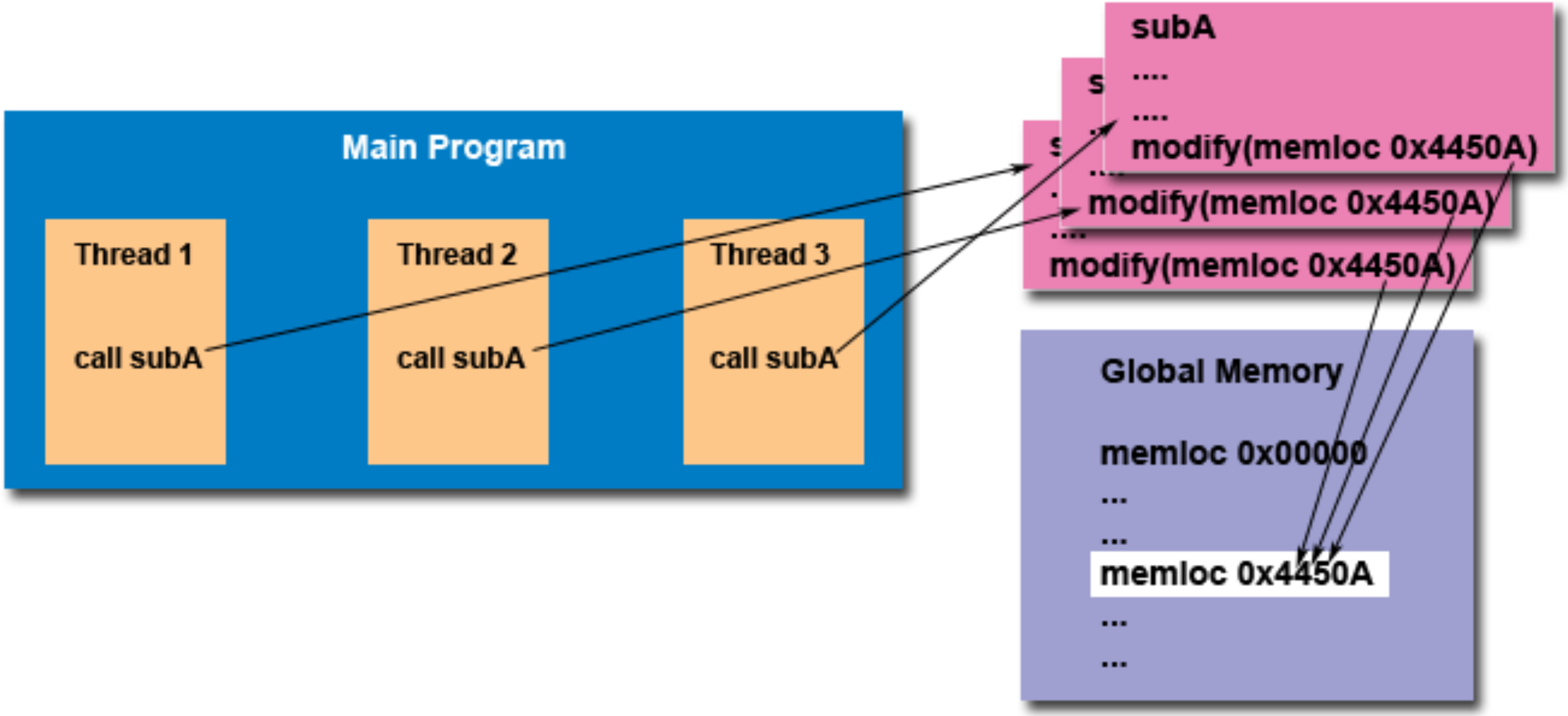
Thread Consequences

- Shared State!
  - Accidental changes to global variables can be fatal.
  - Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads
  - Two pointers having the same value point to the same data
  - Reading and writing to the same memory locations is possible
  - Therefore requires explicit synchronization by the programmer
- Many library functions are not thread-safe
  - Library Functions that return pointers to static internal memory. E.g. gethostbyname()
- Lack of robustness
  - Crash in one thread will crash the entire process

Thread-safeness

- Thread-safeness: in a nutshell, refers an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions
- Example: an application creates several threads, each of which makes a call to the same library routine:
  - This library routine accesses/modifies a global structure or location in memory.
  - As each thread calls this routine it is possible that they may try to modify this global structure/memory location at the same time.
  - If the routine does not employ some sort of synchronization constructs to prevent data corruption, then it is not thread-safe.

Thread-safeness



Thread-safeness

The implication to users of external library routines:

- If you aren't 100% certain the routine is thread-safe, then you take your chances with problems that could arise.

- **Recommendation**
  - Be careful if your application uses libraries or other objects that don't explicitly guarantee thread-safeness.
  - When in doubt, assume that they are not thread-safe until proven otherwise
  - This can be done by "serializing" the calls to the uncertain routine, etc.

Example 4: Data Racing

```c
#include <pthread.h>
#define NUM_THREADS 5

void *PrintHello(void *thread_id) { /* thread func */
    long tid = *((long*)thread_id);
    printf("Hello World! It's me, thread #%ld!\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];
    long t;
    for(t=0;t<NUM_THREADS;t++) {
       printf("In main: creating thread %ld\n", t);
       int rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t );
       if (rc) {
            printf("ERROR; return code from pthread_create() is %d\n", rc);
            exit(-1);
       }
    }
  pthread_exit(NULL);
}
```

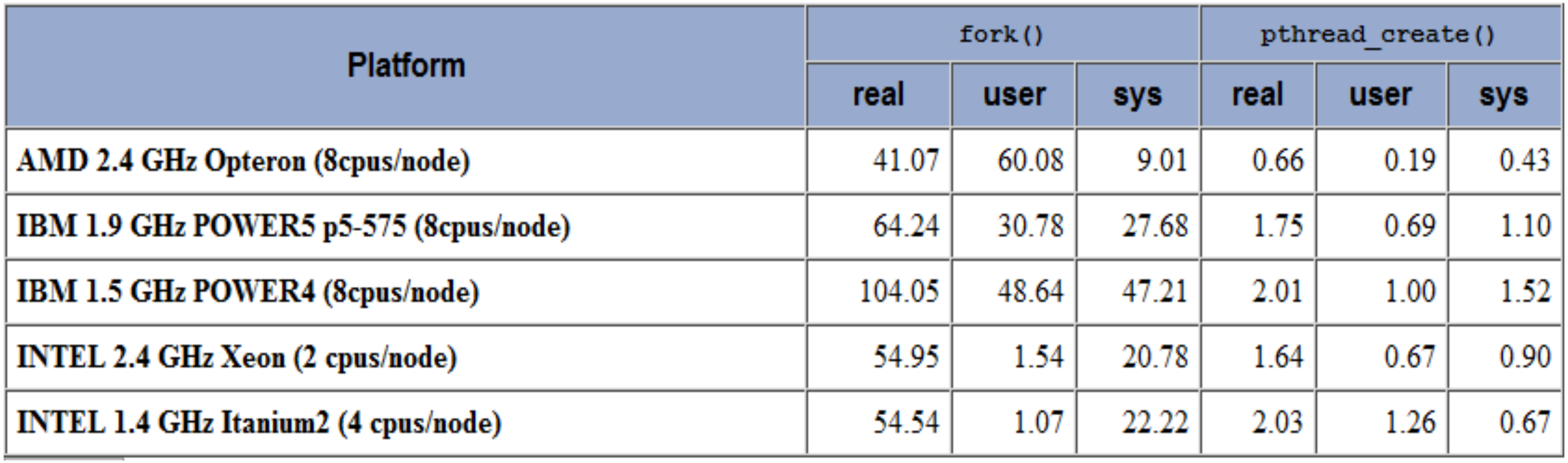
```
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #3!
Hello World! It's me, thread #3!
Hello World! It's me, thread #3!
In main: creating thread 4
Hello World! It's me, thread #4!
Hello World! It's me, thread #5!
```

Why PThreads (not processes)?

- The primary motivation
  - To realize potential program performance gains
- Compared to the cost of creating and managing a process
  - A thread can be created with much less OS overhead
- Managing threads requires fewer system resources than managing processes
- All threads within a process share the same address space
- Inter-thread communication is more efficient and, in many cases, easier to use than inter-process communication

pthread_create vs fork

- Timing results for the **fork()** subroutine and the **pthreads_create()** subroutine
  - Timings reflect 50,000 process/thread creations
  - units are in seconds
  - no optimization flags



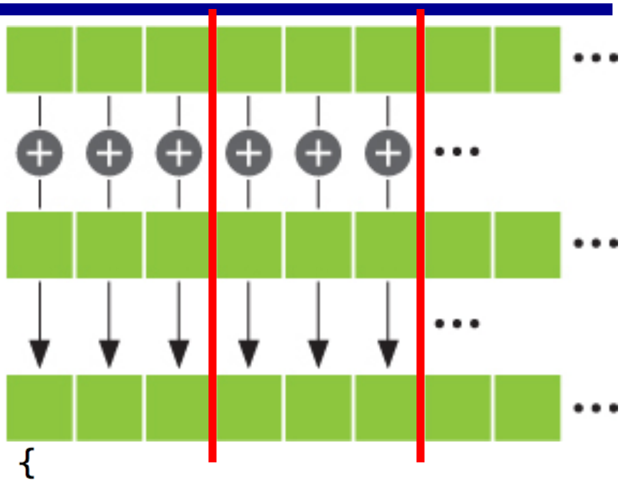
Platform	`fork()`			`pthread_create()`		
	real	user	sys	real	user	sys
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.10
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1.00	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.90
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Why pthreads

- Potential performance gains and practical advantages over non-threaded applications:
  - Overlapping CPU work with I/O
    - For example, a program may have sections where it is performing a long I/O operation
    - While one thread is waiting for an I/O system call to complete, CPU intensive work can be performed by other threads.
- Priority/real-time scheduling
  - Tasks which are more important can be scheduled to supersede or interrupt lower priority tasks.
- Asynchronous event handling
  - Tasks which service events of indeterminate frequency and duration can be interleaved
  - For example, a web server can both transfer data from previous requests and manage the arrival of new requests.

AXPY with PThreads

- y = α·x + y
  - x and y are vectors of size N
    - In C, x[N], y[N]
  - α is scalar
- Decomposition and mapping to pthreads



```
void dist(int tid, int N, int num_tasks, int *Nt, int *start) {
    int remain = N % num_tasks;
    int esize = N / num_tasks;
    if (tid < remain) { /* each of the first remain task has one more element */
        *Nt = esize + 1;
        *start = *Nt * tid;
    } else {
        *Nt = esize;
        *start = esize * tid + remain;
    }
}
void axpy_dist(int N, REAL Y[], REAL X[], REAL a, int num_tasks) {
    int tid;
    for (tid = 0; tid < num_tasks; tid++) {
        int Nt, start;
        dist(tid, N, num_tasks, &Nt, &start);
        axpy_base_sub(start, Nt, N, Y, X, a);
    }
}
```

A task will be mapped to a pthread

AXPY with PThreads

```
struct axpy_dist_pthread_data {
    int Nt;
    int start;
    int N;
    REAL *Y;
    REAL *X;
    REAL a;
};

void * axpy_thread_func(void * axpy_thread_arg) {
    struct axpy_dist_pthread_data * arg = (struct axpy_dist_pthread_data *) axpy_thread_arg;
    axpy_base_sub(arg->start, arg->Nt, arg->N, arg->Y, arg->X, arg->a);
    pthread_exit(NULL);
}

void axpy_dist_pthread(int N, REAL Y[], REAL X[], REAL a, int num_tasks) {
    struct axpy_dist_pthread_data pthread_data_array[num_tasks];
    pthread_t task_threads[num_tasks];
    int tid;
    for (tid = 0; tid < num_tasks; tid++) {
        int Nt, start;
        dist(tid, N, num_tasks, &Nt, &start);
        struct axpy_dist_pthread_data *task_data = &pthread_data_array[tid];
        task_data->start = start;
        task_data->Nt = Nt;
        task_data->a = a;
        task_data->X = X;
        task_data->Y = Y;
        task_data->N = N;

        pthread_create(&task_threads[tid], NULL, axpy_thread_func, (void*)task_data);
    }

    for (tid = 0; tid < num_tasks; tid++) {
        pthread_join(task_threads[tid], NULL);
    }
}
```

Data Racing in a Multithread Program

Consider:

```
/* each thread to update shared variable
   best_cost */

if (my_cost < best_cost)
   best_cost = my_cost;
```

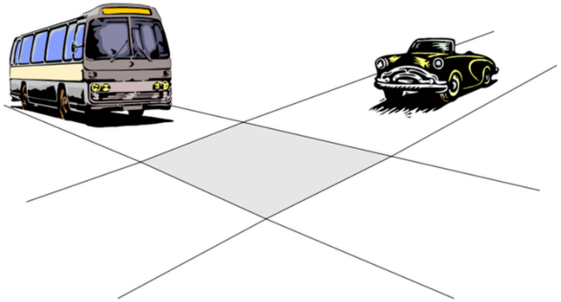
- two threads,
- the initial value of `best_cost` is 100,
- the values of `my_cost` are 50 and 75 for threads t1 and t2

T1	T2
`if (my_cost (50) < best_cost)`	`if (my_cost (75) < best_cost)`
`best_cost = my_cost;`	`best_cost = my_cost;`

- The value of `best_cost` could be 50 or 75!
- The value 75 does not correspond to any serialization of the two threads.

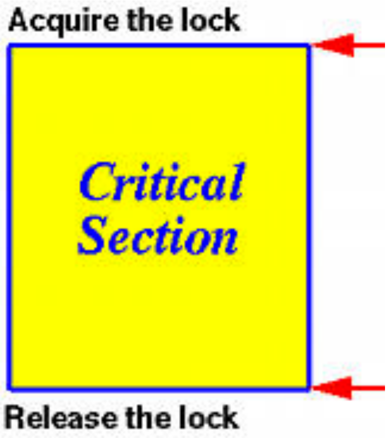
Critical Section and Mutual Exclusion

- Critical section = a segment that must be executed by only one thread at any time



```
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Mutex locks protect critical sections in Pthreads
  - locked and unlocked
  - At any point of time, only one thread can acquire a mutex lock
- Using mutex locks
  - request lock before executing critical section
  - enter critical section when lock granted
  - release lock when leaving critical section



Mutual Exclusion using Pthread Mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex_lock);
int pthread_mutex_unlock (pthread_mutex_t *mutex_lock);
int pthread_mutex_init (pthread_mutex_t   *mutex_lock,
    const pthread_mutexattr_t *lock_attr);
```

```
pthread_mutex_t cost_lock;
int main() {
      ...
   pthread_mutex_init(&cost_lock, NULL);
   pthread_create(&thhandle, NULL, find_best, ...)
      ...
}
void *find_best(void *list_ptr) {
   ...
   pthread_mutex_lock(&cost_lock); // enter CS
   if (my_cost < best_cost)
      best_cost = my_cost;
   pthread_mutex_unlock(&cost_lock); // leave CS
}
```

Critical Section

pthread_mutex_lock blocks the calling thread if another thread holds the lock

When **pthread_mutex_lock** call returns

1. Mutex is locked, enter CS
2. Any other locking attempt (call to thread_mutex_lock) will cause the blocking of the calling thread

When **pthread_mutex_unlock** returns

1. Mutex is unlocked, leave CS
2. One thread who blocks on thread_mutex_lock call will acquire the lock and enter CS

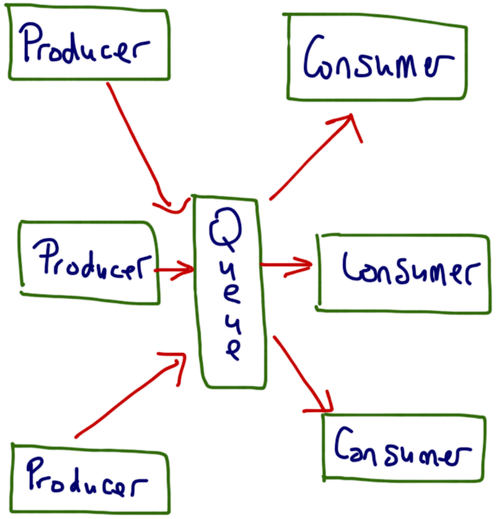
Producer-Consumer Using Locks

Constrains:

- The producer threads
  - must not overwrite the shared buffer when the previous task has not been picked up by a consumer thread.
- The consumer threads
  - must not pick up tasks until there is something present in the shared data structure.
  - Individual consumer thread should pick up tasks one at a time

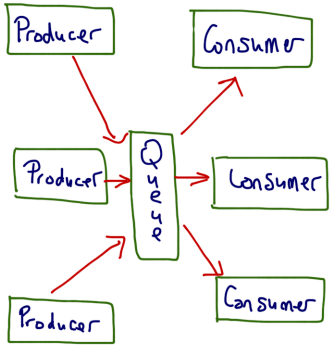
Contention:

- Between producers
- Between consumers
- Between producers and consumers



Producer-Consumer Using Locks

```
pthread_mutex_t task_queue_lock;
int task_available;
main() {
    ....
    task_available = 0;
    pthread_mutex_init(&task_queue_lock, NULL);
    ....
}
```



```
void *producer(void *producer_thread_data) {
    ....
    while (!done()) {
        inserted = 0;
        create_task(&my_task);
        while (inserted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 0) {
                insert_into_queue(my_task);
                task_available = 1; inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

Critical Section

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    while (!done()) {
        extracted = 0;
        while (extracted == 0) {
            pthread_mutex_lock(&task_queue_lock);
            if (task_available == 1) {
                extract_from_queue(&my_task);
                task_available = 0; extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
}
```

Note the purpose of inserted and extraced variables

Three Types of Mutexes

- Normal
  - Deadlocks if a thread already has a lock and tries a second lock on it.
- Recursive
  - Allows a single thread to lock a mutex as many times as it wants.
    - It simply increments a count on the number of locks.
  - A lock is relinquished by a thread when the count becomes zero.
- Error check
  - Reports an error when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case).

- The type of the mutex can be set in the attributes object before it is passed at time of initialization
  - pthread_mutex_attr_init

Overheads of Locking

- Locks enforce serialization
  - Thread must execute critical sections one after another
- Large critical sections can lead to significant performance degradation.
- Reduce the blocking overhead associated with locks using:

```
int pthread_mutex_trylock (
        pthread_mutex_t *mutex_lock);
```

- acquire lock if available
- return EBUSY if not available
- enables a thread to do something else if lock unavailable

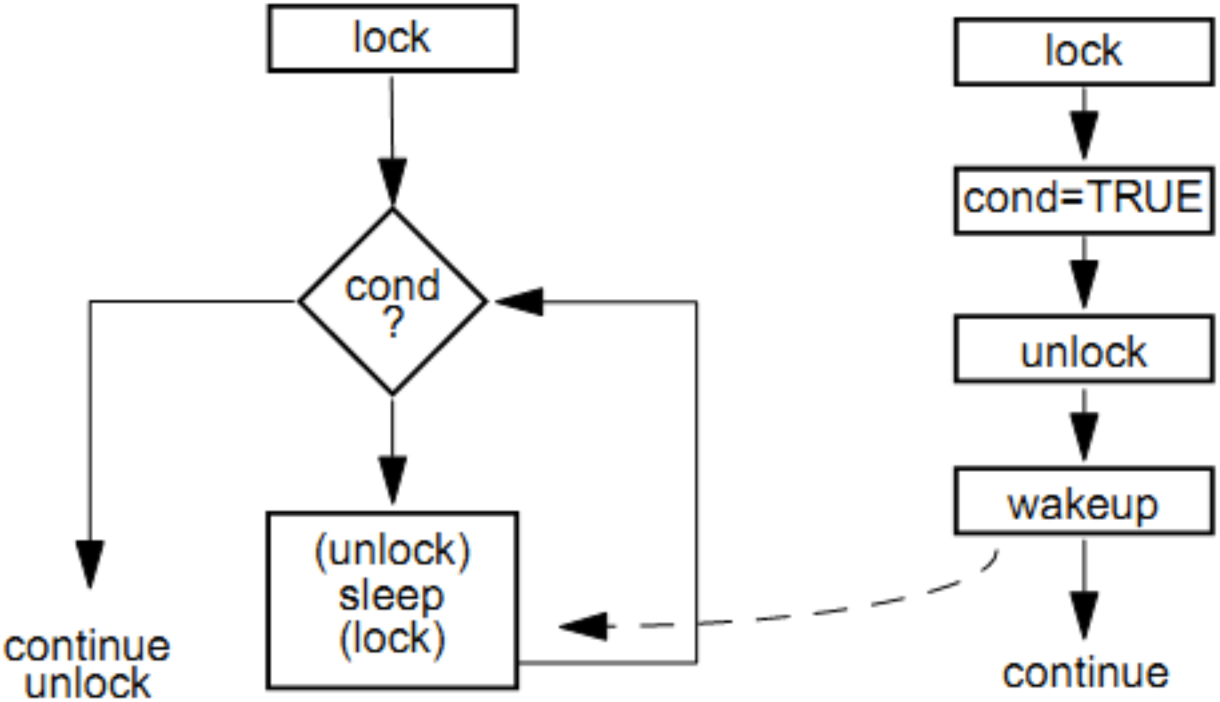
- pthread **trylock** typically much faster than **lock** on certain systems
  - It does not have to deal with queues associated with locks for multiple threads waiting on the lock.

Condition Variables for Synchronization

A condition variable: associated with a predicate and a mutex

– A sync variable for a condition, e.g. mybalance > 500

- A thread can block itself until a condition becomes true
  - Wh e it
- Whe another thre l other thre
- A co ciated with it.
  - A t



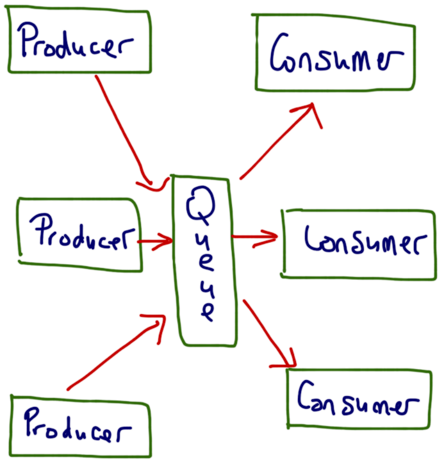
Using a Condition Variable

Condition Variables for Synchronization

```
/* the opaque data structure */
pthread_cond_t

/* initialization and destroying */
int pthread_cond_init(pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);

/* block and release lock until a condition is true */
int pthread_cond_wait(pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
pthread_mutex_t *mutex, const struct timespec *wtime);

/* signal one or all waiting threads that condition is true */
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Producer-Consumer Using Condition Variables

```
pthread_cond_t cond_queue_empty, cond_queue_full;
pthread_mutex_t task_queue_cond_lock;
int task_available;
/* other data structures here */

main() {
  /* declarations and initializations */
  task_available = 0;
  pthread_cond_init(&cond_queue_empty, NULL);
  pthread_cond_init(&cond_queue_full, NULL);
  pthread_mutex_init(&task_queue_cond_lock, NULL);
  /* create and join producer and consumer threads */
}
```



- Two conditions:
  - Queue is full: (task_available == 1) ← `cond_queue_full`
  - Queue is empty: (task_available == 0) ← `cond_queue_empty`
- A mutex for protecting accessing the queue (CS): `task_queue_cond_lock`

Producer-Consumer Using Condition Variables

```
void *producer(void *producer_thread_data) {
  int inserted;
  while (!done()) {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);

    while (task_available == 1)
1     pthread_cond_wait(&cond_queue_empty,
         &task_queue_cond_lock);
    insert_into_queue();
2   task_available = 1;            CS
3   pthread_cond_signal(&cond_queue_full);

    pthread_mutex_unlock(&task_queue_cond_lock);
  }
}
```

Release mutex (unlock) when blocked/wait

Acquire mutex (lock) when awaken

Producer:

1. **Wait for queue to become empty, notified by consumer through cond_queue_empty**
2. **insert into queue**
3. **Signal consumer through cond_queue_full**

Producer-Consumer Using Condition Variables

```
   void *consumer(void *consumer_thread_data) {
     while (!done()) {
     pthread_mutex_lock(&task_queue_cond_lock);

1    while (task_available == 0)
       pthread_cond_wait(&cond_queue_full,
         &task_queue_cond_lock);

2      my_task = extract_from_queue();
       task_available = 0;

3      pthread_cond_signal(&cond_queue_empty);

       pthread_mutex_unlock(&task_queue_cond_lock);
       process_task(my_task);
     }
   }
```

Release mutex (unlock) when blocked/wait

Acquire mutex (lock) when awaken

Consumer:

1. **Wait for queue to become full, notified by producer through cond_queue_full**
2. **Extract task from queue**
3. **Signal producer through cond_queue_empty**

Thread and Synchronization Attributes

- Three major objects
  - pthread_t
  - pthread_mutex_t
  - pthread_cond_t
- Default attributes when being created/initialized
  - NULL
- An attributes object is a data-structure that describes entity (thread, mutex, condition variable) properties.
  - Once these properties are set, the attributes object can be passed to the method initializing the entity.
  - Enhances modularity, readability, and ease of modification.

Attributes Objects for Threads

- Initialize an attribute objects using **`pthread_attr_init`**
- Individual properties associated with the attributes object can be changed using the following functions:

`pthread_attr_setdetachstate`,
`pthread_attr_setguardsize_np`,
`pthread_attr_setstacksize`,
`pthread_attr_setinheritsched`,
`pthread_attr_setschedpolicy`, and
`pthread_attr_setschedparam`

Attributes Objects for Mutexes

- Initialize an attributes object using function: `pthread_mutexattr_init`.
- **pthread_mutexattr_settype_np** for setting the mutex type

  pthread_mutexattr_settype_np (pthread_mutexattr_t *attr,int type);
- Specific types:
  - PTHREAD_MUTEX_NORMAL_NP
  - PTHREAD_MUTEX_RECURSIVE_NP
  - PTHREAD_MUTEX_ERRORCHECK_NP

Attributes Objects for Condition Variable

- Initialize an attribute object using `pthread_condattr_init`
- **int pthread_condattr_setpshared(pthread_condattr_t *cattr, int pshared)** to specifies the scope of a condition variable to either process private (intraprocess) or system wide (interprocess) via pshared
  - PTHREAD_PROCESS_SHARED
  - PTHREAD_PROCESS_PRIVATE

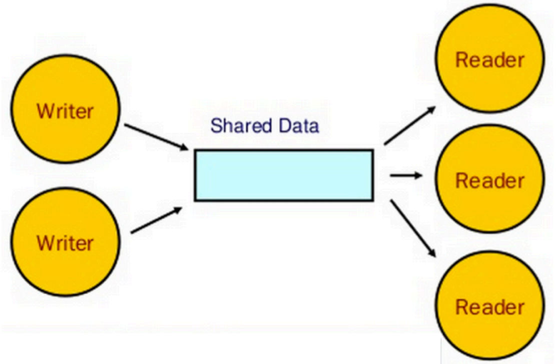
Composite Synchronization Constructs

- Pthread **Mutex** and **Condition Variables** are two basic sync operations.
- Higher level constructs can be built using basic constructs.
  - Read-write locks
  - Barriers

- Pthread has its corresponding implementation
  - pthread_rwlock_t
  - pthread_barrier_t

- We will discuss our own implementations

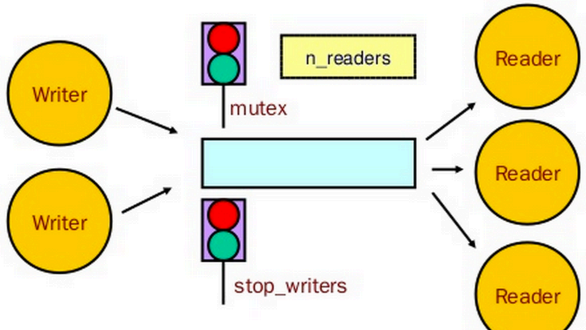
Read-Write Locks



- Concurrent access to data structure:
  - Read frequently but
  - Written infrequently
- Behavior:
  - Concurrent read: A read request is granted when there are other reads or no write (pending write request).
  - Exclusive write: A write request is granted only if there is no write or pending write request, or reads.
- Interfaces:
  - The rw lock data structure: **struct `mylib_rwlock_t`**
  - Read lock: **mylib_rwlock_rlock**
  - write lock: **mylib_rwlock_wlock**
  - Unlock: **mylib_rwlock_unlock**.

Read-Write Locks

- Two types of mutual exclusions
  - 0/1 mutex for protecting access to write
  - Counter mutex (semaphore) for counting read access
- Component sketch
  - a count of the number of readers,
  - 0/1 integer specifying whether a writer is present,
  - a condition variable readers_proceed that is signaled when readers can proceed,
  - a condition variable writer_proceed that is signaled when one of the writers can proceed,
  - a count pending_writers of pending writers, and
  - a pthread_mutex_t read_write_lock associated with the shared data structure



Read-Write Locks

```
typedef struct {
  int readers;
  int writer;
  pthread_cond_t readers_proceed;
  pthread_cond_t writer_proceed;
  int pending_writers;
  pthread_mutex_t read_write_lock;
} mylib_rwlock_t;


void mylib_rwlock_init (mylib_rwlock_t *l) {
  l->readers=0; l->writer=0; l->pending_writers=0;
  pthread_mutex_init(&(l->read_write_lock), NULL);
  pthread_cond_init(&(l->readers_proceed), NULL);
  pthread_cond_init(&(l->writer_proceed), NULL);
}
```

Read-Write Locks

```
  void mylib_rwlock_rlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l->read_write_lock));

    while ((l->pending_writers > 0) || (l->writer > 0))
1     pthread_cond_wait(&(l->readers_proceed),
      &(l->read_write_lock));

2   l->readers ++;

    pthread_mutex_unlock(&(l->read_write_lock));
  }
```

Reader lock:

1. if there is a write or pending writers, perform condition wait,
2. else increment count of readers and grant read lock

Read-Write Locks

```
  void mylib_rwlock_wlock(mylib_rwlock_t *l) {
     pthread_mutex_lock(&(l->read_write_lock));
     l->pending_writers ++;

     while ((l->writer > 0) || (l->readers > 0)) {
1       pthread_cond_wait(&(l->writer_proceed),
            &(l->read_write_lock));
     }

     l->pending_writers --;
2    l->writer ++;

     pthread_mutex_unlock(&(l->read_write_lock));
  }
```

Writer lock:

1. **If there are readers or writers, increment pending writers count and wait.**
2. **On being woken, decrement pending writers count and increment writer count**

Read-Write Locks

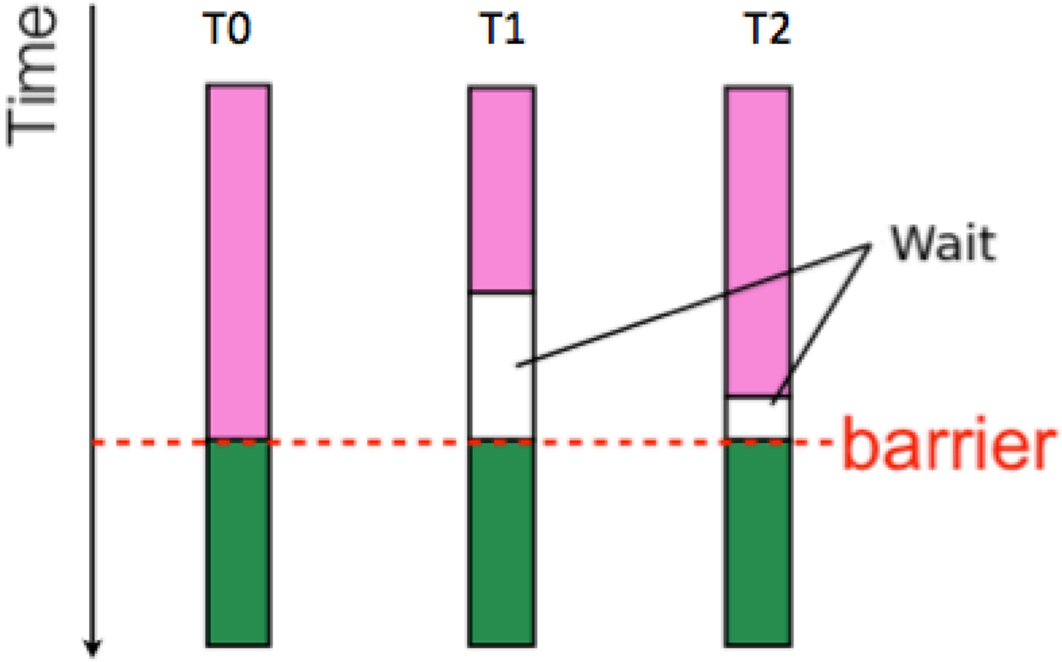
```
  void mylib_rwlock_unlock(mylib_rwlock_t *l) {
    pthread_mutex_lock(&(l->read_write_lock));
1   if (l->writer > 0) /* only writer */
      l->writer = 0;
2   else if (l->readers > 0) /* only reader */
      l->readers --;
    pthread_mutex_unlock(&(l->read_write_lock));

3   if ((l->readers == 0) && (l->pending_writers > 0))
      pthread_cond_signal(&(l->writer_proceed));
4   else if (l->readers > 0)
      pthread_cond_broadcast(&(l->readers_proceed));
  }
```

Reader/Writer unlock:

1. If there is a write lock then unlock
2. If there are read locks, decrement count of read locks.
3. If the read count becomes 0 and there is a pending writer, notify writer
4. Otherwise if there are pending readers, let them all go through

Barrier

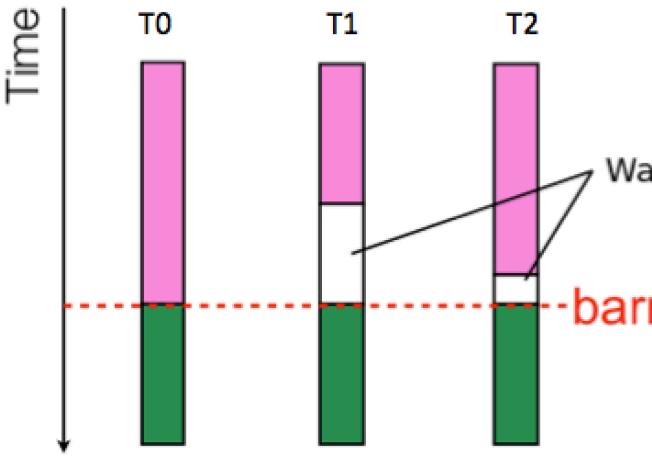
- A barrier holds one or multiple threads until all threads participating in the barrier have reached the barrier point



Barrier

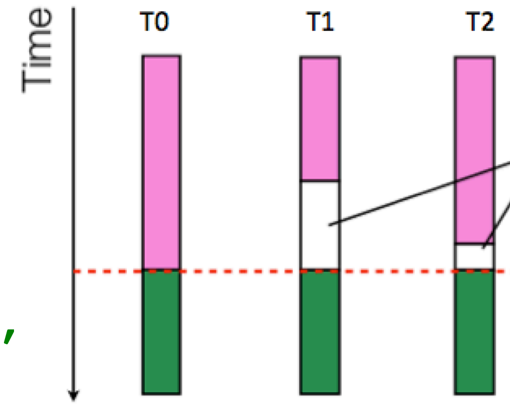
- Needs **a counter**, **a mutex** and **a condition variable**
  - The counter keeps track of the number of threads that have reached the barrier.
    - If the count is less than the total number of threads, the threads execute a condition wait.
  - The last thread entering (master) wakes up all the threads using a condition broadcast.

```
typedef struct {
  int count;
  pthread_mutex_t count_lock;
  pthread_cond_t ok_to_proceed;
} mylib_barrier_t;

void mylib_barrier_init(mylib_barrier_t *b) {
  b->count = 0;
  pthread_mutex_init(&(b->count_lock), NULL);
  pthread_cond_init(&(b->ok_to_proceed), NULL);
}
```



Barriers

```
void mylib_barrier (mylib_barrier_t *b, int num_threads) {
  pthread_mutex_lock(&(b->count_lock));

1 b->count ++;
  if (b->count == num_threads) {
2   b->count = 0;
    pthread_cond_broadcast(&(b->ok_to_proceed));
  } else
3   while (pthread_cond_wait(&(b->ok_to_proceed),
      &(b->count_lock)) != 0);

  pthread_mutex_unlock(&(b->count_lock));
}
```

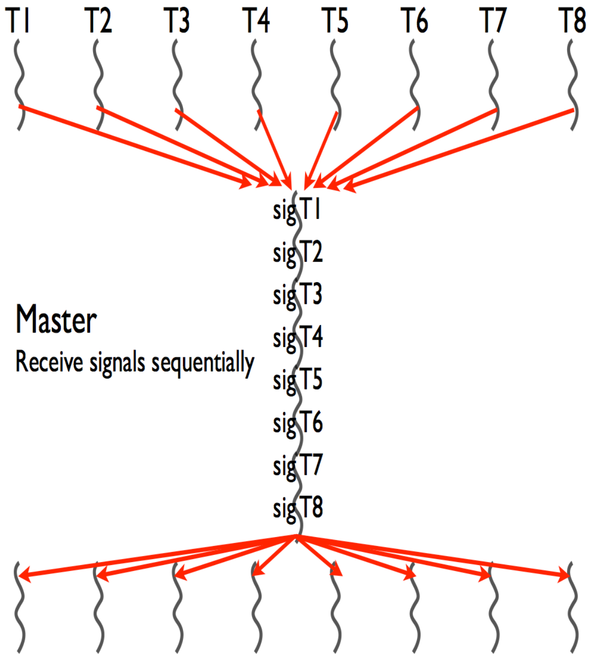


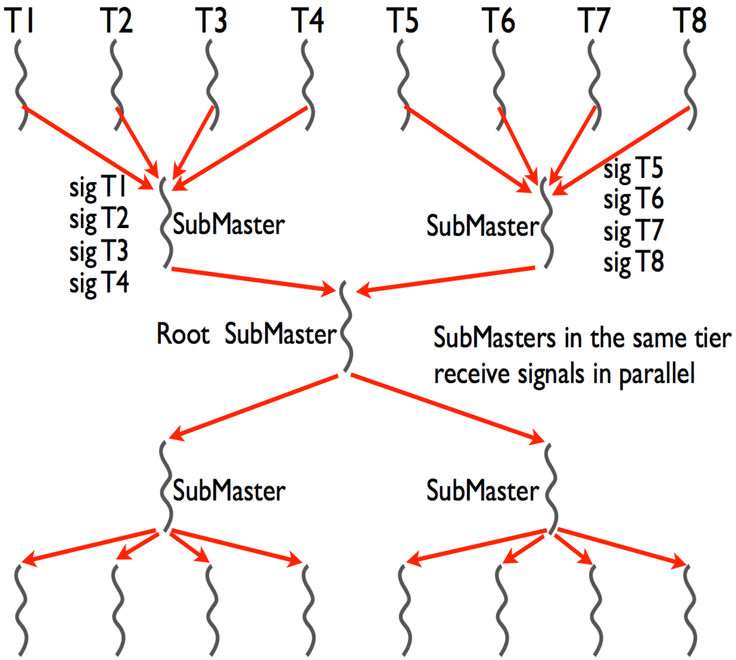
Barrier

1. Each thread increments the counter and check whether all reach
2. The thread (master) who detect that all reaches signal others to proceed
3. If not all reach, the thread waits

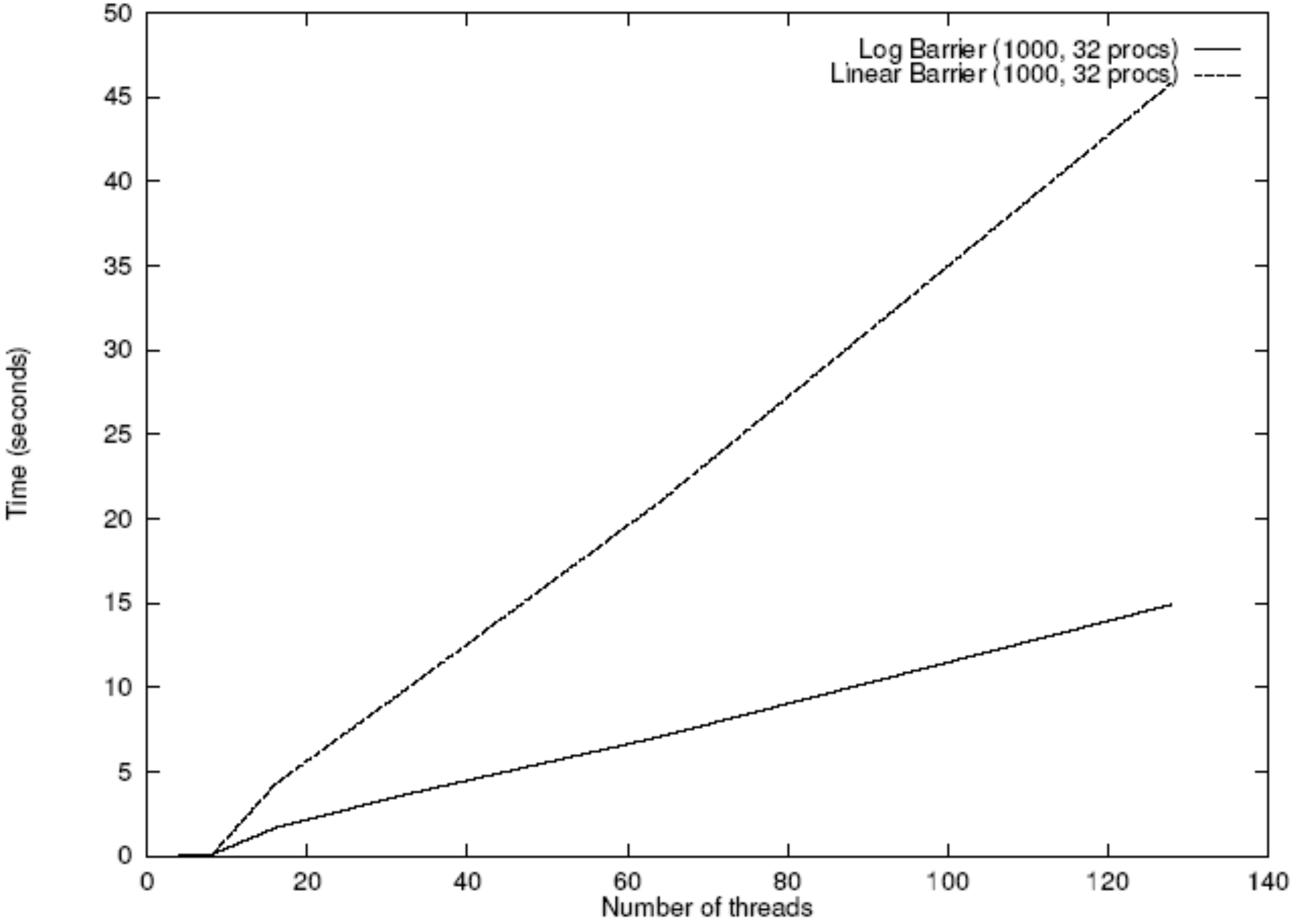
Flat/Linear vs Tree/Log Barrier

- Linear/Flat barrier.
  - O(n) for n thread
  - A single master to collect information of all threads and notify them to continue
- Tree/Log barrier
  - Organize threads in a tree logically
  - Multiple submaster to collect and notify
  - Runtime grows as O(log p).





Barrier



- Execution time of 1000 sequential and logarithmic barriers as a function of number of threads on a 32 processor SGI Origin 2000.

References

- Adapted from slides "Programming Shared Address Space Platforms" by Ananth Grama. Bradford Nichols, Dick Buttlar, Jacqueline Proulx Farrell.
- "Pthreads Programming: A POSIX Standard for Better Multiprocessing." O'Reilly Media, 1996.
- Chapter 7. "Introduction to Parallel Computing" by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. Addison Wesley, 2003

- Other pthread topics
  - int pthread_key_create(pthread_key_t *key, void (*destroy)(void *))
  - int pthread_setspecific(pthread_key_t key, const void *value)
  - void *pthread_getspecific(pthread_key_t key)