BITS Pilani
Pilani Campus

Programming Shared Address Space Platforms

K Hari Babu
Department of Computer Science & Information Systems

Pthreads

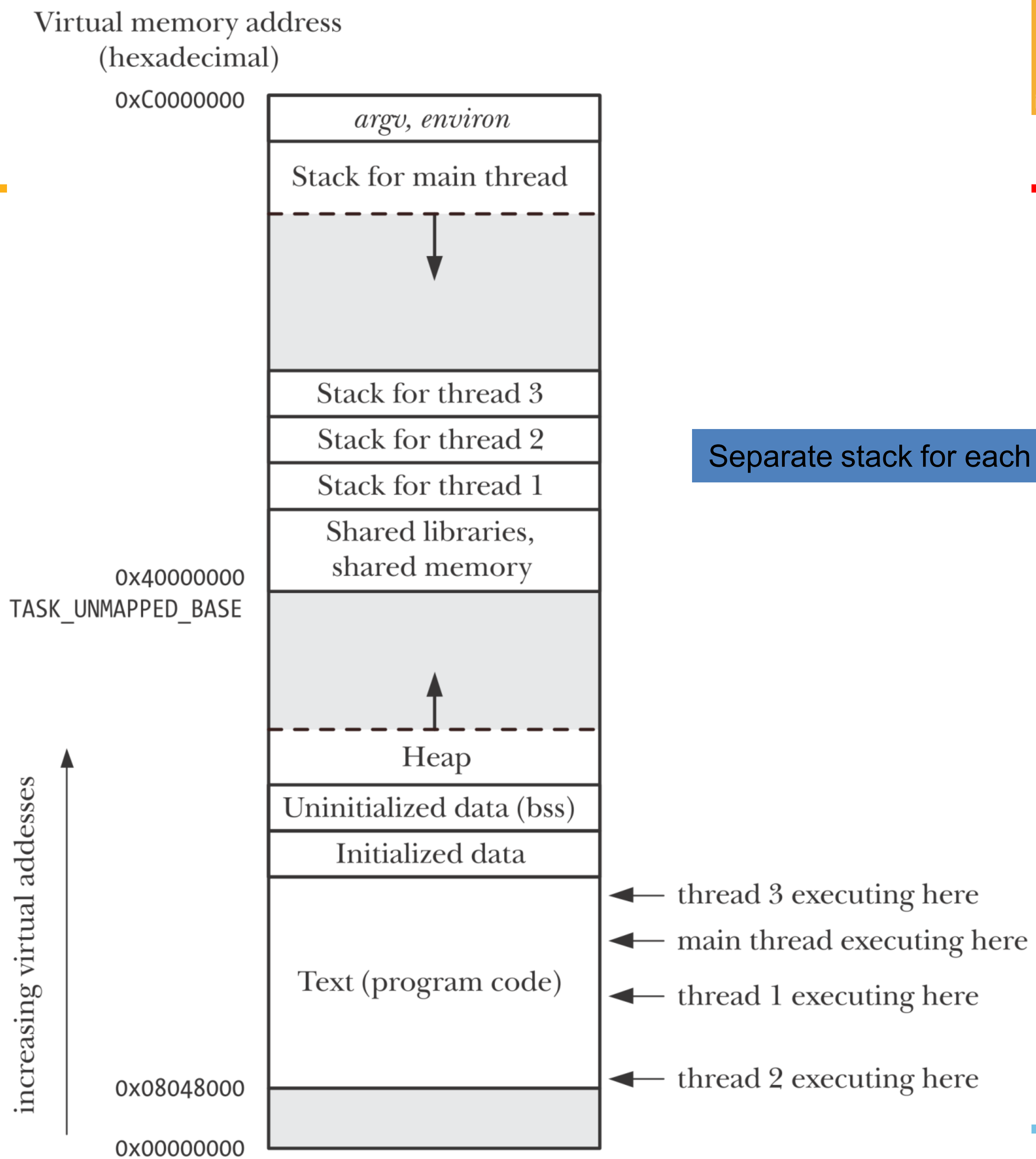
T1: ch 7

Threads

- Threads share
  - heap, data, text segments, global variables, file descriptors, CWD, user, group ids, signal handlers, signal dispositions
- Individual
  - stack, registers, program counters, *errno*, signal mask, priority
- *errno* is local to each thread.
  - Normally sys calls return -1 on error and set errno. But Pthread API calls return 0 on success and >0 on failure. They do not set errno.
    - errno is set when a sys call is directly called within a thread.
- Compiling pthread applications.

```
gcc  pthread.c -lpthread
```

  - The program is linked with the libpthread library.



Separate stack for each thread

Multiple threads run concurrently

The POSIX Thread API

- Commonly referred to as Pthreads, POSIX has emerged as the standard threads API, supported by most vendors.
- The concepts discussed here are largely independent of the API and can be used for programming with other thread APIs (NT threads, Solaris threads, Java threads, etc.) as well.

Pthreads API

- Thread Management
  - creation, detach, join, exit
    - POSIX requires that all threads are created as joinable threads
- Thread Synchronization
  - join
  - mutex (i.e. semapore with value of 1)
  - semaphore
  - condition variables

Thread Creation

- When a program is started by exec, a single thread is created, called the initial thread or main thread. Additional threads are created by *pthread_create*.

```
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t * attr,
                   void *(* start )(void *), void * arg );
//Returns 0 on success, or a positive error number on error
```

  - *start:* this is the function, the thread will start executing.
  - *thread*: this is the thread id, filled by kernel.
  - *attr*: normally NULL.
    - Each thread has numerous attributes: its priority, its initial stack size, whether it should be a daemon thread or not etc.
    - When a thread is created, we can specify these attributes by initializing a pthread_attr_t variable that overrides the default.
  - *arg*: argument to the function.

Joining with Terminated Thread

- We can wait for a given thread to terminate by calling pthread_join.
    - pthread_create is similar to fork, and pthread_join is similar to waitpid.

```
include <pthread.h>
int pthread_join(pthread_t  thread , void ** retval );
//Returns 0 on success, or a positive error number on error
```

- If *retval* is a non NULL pointer, then it receives status of a thread specified by pthread_exit().
- A thread can call join on any thread in the process. No parent-child relationship in threads.
- A non-detatched thread must be joined by some thread, otherwise it will lead to zombie thread. Resources will be held up in the kernel.

Detaching a Thread

- By default a thread is joinable. If we do not join, kernel will store the status of the thread.
- If we do not care about the status of the thread, then we can detach the thread.
  - System will automatically cleanup when the thread terminates.

```
#include <pthread.h>
int pthread_detach(pthread_t  thread );
//Returns 0 on success, or a positive error number on error
```

  - Detaching a thread doesn't make it immune to exit() in another thread or a return in the main thread.
  - pthread_detach() simply controls what happens after a thread terminates, not how or when it terminates.
- A thread can detach itself.

```
pthread_detach(pthread_self());
```

More Pthread Functions

innovate achieve lead

- pthread_self Function
  - Each thread has an ID that identifies it within a given process. The thread ID is returned by pthread_create().
  - A thread fetches this value for itself using pthread_self().

```
#include <pthread.h>
pthread_t pthread_self(void);
//Returns the thread ID of the calling thread
```

- pthread_exit Function

```
#include <pthread.h>
void pthread_exit(void * retval );
```

  - The retval argument specifies the return value for the thread.
  - Calling pthread_exit() is equivalent to performing a return in the thread's start function.
    - If the main thread calls pthread_exit() instead of calling exit() or performing a return , then the other threads continue to execute.

Computing PI

```c
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
...
pthread_t p_threads[MAX_THREADS];
pthread_attr_t attr;
pthread_attr_init (&attr);
for (i=0; i< num_threads; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
        (void *) &hits[i]);
}
for (i=0; i< num_threads; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
}
computed_pi=4.0*(double)total_hits/((double)(sample_points))
;}
```

```c
void *compute_pi (void *s) {
int seed, i, *hit_pointer;
double rand_no_x, rand_no_y;
int local_hits;
hit_pointer = (int *) s;
seed = *hit_pointer;
local_hits = 0;
for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1)
        rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1)
if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
(rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
        local_hits ++;
        seed *= i;
}
        *hit_pointer = local_hits;
        pthread_exit(0);
}
```

Protecting Accesses to Shared Variables: Mutexes

- The term critical section refers to a section of code that accesses a shared resource and whose execution should be atomic.
- Its execution should not be interrupted by another thread that simultaneously accesses the same shared resource.
- Pthreads provide mutexes for protecting critical section.
  - Each mutex has two states: locked, unlocked.
  - At most one thread may hold lock on a mutex.
  - When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex.

Critical Section Problem

```c
#define NLOOP 5000
int counter; /* incremented by threads */
void   *doit(void *);
int main(int argc, char **argv)
{
    pthread_t tidA, tidB;
    pthread_create(&tidA, NULL, &doit, NULL);
    pthread_create(&tidB, NULL, &doit, NULL);
    /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);
 exit(0);
}
```

```c
void * doit(void *vptr)
{
  int    i, val;
/*Each thread fetches, prints,
and increments the counter NLOOP times.
The value of the counter should increase monotonically.
*/
  for (i = 0; i < NLOOP; i++) {
      val = counter;
      printf("%d: %d\n", pthread_self(), val + 1);
      counter = val + 1;
  }
  return (NULL);
}
```

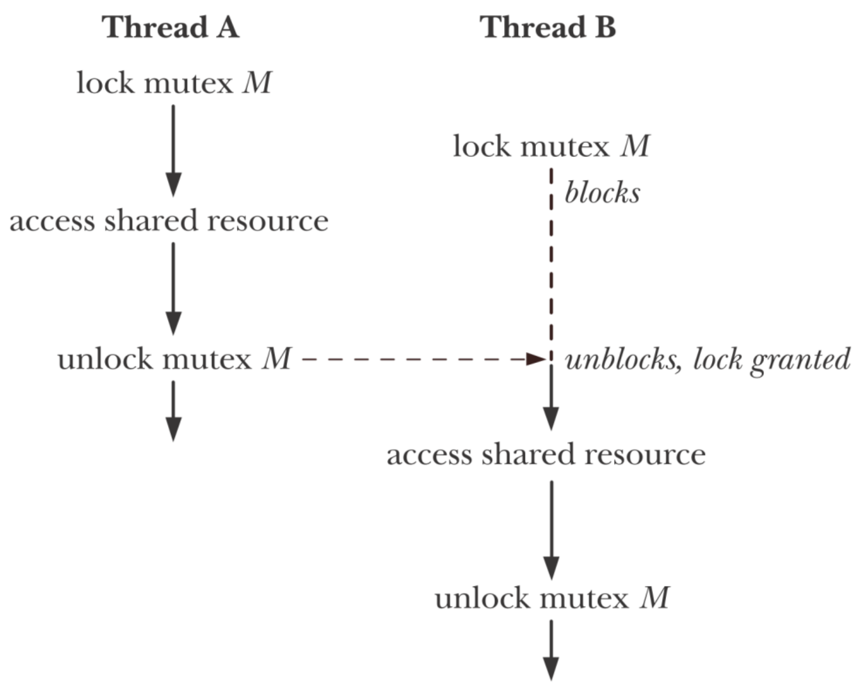
```
4: 1
4: 2
4: 3
4: 4
              continues as thread 4 executes

4: 517
4: 518
5: 518        thread 5 now executes
5: 519
5: 520
              continues as thread 5 executes

5: 926
5: 927
4: 519        thread 4 now executes; stored value is wrong
4: 520
```

Using Mutex

- Each thread employs the following protocol for accessing a resource:
  - lock the mutex for the shared resource;
  - access the shared resource; and
  - unlock the mutex.



Using Mutex

- A mutex is a variable of the type pthread_mutex_t. Mutex must always be initialized.
  - For a statically allocated mutex

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mutex );
int pthread_mutex_unlock(pthread_mutex_t * mutex );
//Both return 0 on success, or a positive error number on error
```

  - The pthread_mutex_trylock() function is the same as pthread_mutex_lock(), except that if the mutex is currently locked, pthread_mutex_trylock() fails, returning the error EBUSY.

Using Mutexes

```c
#define NLOOP 5000
int      counter;                 /* incremented by threads */
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
void    *doit(void *);
int main(int argc, char **argv)
 {
     pthread_t tidA, tidB;
     Pthread_create(&tidA, NULL, &doit, NULL);
     Pthread_create(&tidB, NULL, &doit, NULL);
         /* wait for both threads to terminate */
     Pthread_join(tidA, NULL);
     Pthread_join(tidB, NULL);
     exit(0);
 }
void *
doit(void *vptr)
{
    int      i, val;
    for (i = 0; i < NLOOP; i++) {
        pthread_mutex_lock(&counter_mutex);
        val = counter;
        printf("%d: %d\n", pthread_self(), val + 1);
        counter = val + 1;
        pthread_mutex_unlock(&counter_mutex);
    }
    return (NULL);
}
```

Condition Variables

- A mutex is fine to prevent simultaneous access to a shared variable, but we need something else to let us go to sleep waiting for some condition to occur.

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
/*producer thread*/
pthread_mutex_lock(&mtx);
avail++;/* Let consumer know another unit is available */
pthread_mutex_unlock(&mtx);
/*consumer thread*/
for (;;) {
    pthread_mutex_lock(&mtx);
        while (avail > 0) {/* Consume all available units */
            avail--;
    }
    pthread_mutex_unlock(&mtx);
}
```

- The above code works, but it wastes CPU time, because the consumer thread continually loops, checking the state of the variable *avail*. A condition variable remedies this problem.

Condition Variables

- Condition variable allows a thread to sleep (wait) until another thread notifies (signals) it that it must do something.
- A condition variable is always used in conjunction with a mutex.
- The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variable's state.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t * cond );
int pthread_cond_broadcast(pthread_cond_t * cond );
int pthread_cond_wait(pthread_cond_t * cond , pthread_mutex_t * mutex );
//All return 0 on success, or a positive error number on error
```

- Broadcast wakes up all blocked threads. Each will go through the code. Used when there is different tasks done for a particular condition.

Using Condition Variables

- Why mutex is associated with condition variable?
  - The thread locks the mutex in preparation for checking the state of the shared variable.
  - The state of the shared variable is checked.
  - If the shared variable is not in the desired state, then the thread must unlock the mutex (so that other threads can access the shared variable) before it goes to sleep on the condition variable.
    - Done atomically
  - When the thread is reawakened because the condition variable has been signaled, the mutex must once more be locked, since, typically, the thread then immediately accesses the shared variable.
- it is not possible for some other thread to acquire the mutex and signal the condition variable before the thread calling pthread_cond_wait() has blocked on the condition variable.

Using Condition Variables

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int avail = 0;
/*producer thread*/
pthread_mutex_lock(&mtx);
avail++;/*Let consumer know another unit is available */
pthread_mutex_unlock(&mtx);
pthread_cond_signal(&cond);  /* Wake sleeping consumer */
```

```
/*consumer thread*/
for (;;) {
    s = pthread_mutex_lock(&mtx);
    while (avail == 0) {/* Wait for something to consume */
        s = pthread_cond_wait(&cond, &mtx);
    }
    while (avail > 0) {/* Consume all available units */
        /* Do something with produced unit */
        avail--;
    }
    s = pthread_mutex_unlock(&mtx);
}
```

Matrix Vector Multiplication

```
/* For each row of A */
for (i = 0; i < m; i++) {
  y[i] = 0.0;
/* For each element of the row and each element of x */
for (j = 0; j < n; j++)
     y[i] += A[i][j]* x[j];
}
```

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
\mathbf{a_{i0}} & \mathbf{a_{i1}} & \cdots & \mathbf{a_{i,n-1}} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
\mathbf{x_0} \\
\hline
\mathbf{x_1} \\
\hline
\vdots \\
\hline
\mathbf{x_{n-1}} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
\mathbf{y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Matrix Vector Multiplication

- Dividing work among the threads
  - One possibility is to divide the iterations of the outer loop among the threads.
  - If there are t threads, and there are m rows, each thread gets m/t rows.
  - For thread q
    - first component: $q \times m/t$
    - and last component: $(q + 1) \times m/t - 1$
  - For 3 threads and 6 rows,

m	t	thread_number	starting_row	ending_row
6	3	0	0	1
		1	2	3
		2	4	5

Compute Pi

- Estimate pi using $$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```c
/*-------------------------------------------------------------------
 * Function:       Thread_sum
 * Purpose:        Add in the terms computed by the thread running this
 * In arg:         rank
 * Ret val:        ignored
 * Globals in:     n, thread_count
 * Global in/out:  sum
 */
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      sum += factor/(2*i+1);
   }

   return NULL;
}  /* Thread_sum */
```

Compute Pi

- Estimate pi using $$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

```
/*---------------------------------------------------------------------*/
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;

   if (my_first_i % 2 == 0)
      factor = 1.0;
   else
      factor = -1.0;

   for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
      my_sum += factor/(2*i+1);
   }
   pthread_mutex_lock(&mutex);
   sum += my_sum;
   pthread_mutex_unlock(&mutex);

   return NULL;
}  /* Thread_sum */
```

Threads and Caching

- If a processor needs to access main memory location x, rather than transferring only the content of x from main memory, a block of memory containing x is transferred from/to processor cache.
  - Such a block of memory is called a cache line or cache.
- Consider Matrix Vector multiplication example

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
\mathbf{a_{i0}} & \mathbf{a_{i1}} & \cdots & \mathbf{a_{i,n-1}} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
\mathbf{x_0} \\
\hline
\mathbf{x_1} \\
\hline
\vdots \\
\hline
\mathbf{x_{n-1}} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
\mathbf{y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Threads and Caching

- On a system with cache line or block size as 64 bytes (8 doubles), multiplying matrices of various dimensions take following times and efficiencies.

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

	Matrix Dimension					
	$8{,}000{,}000 \times 8$		8000×8000		$8 \times 8{,}000{,}000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

<u>With single thread why 8000x8000 better? 8000000x8 matrix</u>:
Entire row will fit into cache line.
Entire x vector will also fit into cache line.
Most cache misses occur in accessing y.

```
for (i = my_first_row; i <= my
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

Threads and Caching

- On a system with cache line or block size as 64 bytes (8 doubles), multiplying matrices of various dimensions take following times and efficiencies.

$$
\begin{bmatrix}
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\vdots & \vdots & & \vdots \\
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\vdots & \vdots & & \vdots \\
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1}
\end{bmatrix}
\begin{bmatrix}
x_0 \\ x_1 \\ \vdots \\ x_{n-1}
\end{bmatrix}
=
\begin{bmatrix}
y_0 \\ y_1 \\ \vdots \\ y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\ \vdots \\ y_{m-1}
\end{bmatrix}
$$

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

<u>With single thread why 8000x8000 better? 8x8000000 matrix</u>: Entire y will fit into cache line. Cache misses will occur in accessing A and x.

```
for (i = my_first_row; i <= my_la
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

Threads and Caching

- On a system with cache line or block size as 64 bytes (8 doubles), multiplying matrices of various dimensions take following times and efficiencies.

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	**Time**	**Eff.**	**Time**	**Eff.**	**Time**	**Eff.**
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

<u>Why 8x8000000 matrix has very very low Performance</u>?
Cache coherence is enforced at the cache-line level. If any core updates y, all other processors have to reload it. This has to be done 8 lakh times in every core. If t=2, it is 16 lakh times. If t =4, it is 32 lakh times.
Core 0 updates only y[0] etc yet it forces all other cores to invalidate y.

```
for (i = my_first_row; i <= my_la
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

Threads and Caching

- On a system with cache line or block size as 64 bytes (8 doubles), multiplying matrices of various dimensions take following times and efficiencies.

$$
\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
=
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}
$$

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.393	1.000	0.345	1.000	0.441	1.000
2	0.217	0.906	0.188	0.918	0.300	0.735
4	0.139	0.707	0.115	0.750	0.388	0.290

<u>This problem is called false sharing</u>
Do other dimensions also have this false sharing problem?
Yes but only with y not with A, x.
Bec. only y is updated.
Y dimension is very big either 8lac or 8000. Each thread gets a big range. T0=[0,……2lac] [2lac…4lac
… very unlikey case of invalidating

<u>Solution</u>

(i) Padding y vector with dummy elements so that one cache line contains elements of a single threa

(ii) Replace y with local variable. Later update it.

```
for (i = my_first_row; i <= my_la
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

Producer-Concumer Work Queues

- Producer creates tasks and inserts them into a work queue. The consumer threads pick up tasks from the task queue and execute them.
- The producer-consumer scenario imposes the following constraints:
  - The producer thread 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 threads should pick up tasks one at a time.

Producer-Concumer

Task_available is used for indicating a task available on the queue. This is updated by both Producer and consumer.
Lock is released in each iteration whether success or not.

```
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);
```

```
void *consumer(void *consumer_thread_data) {
    int extracted;
    struct task my_task;
    /* local data structure declarations */
    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);
    }
```

Overheads of Locking

- Locks represent serialization points since critical sections must be executed by threads one after the other.
- Encapsulating large segments of the program within locks can lead to significant performance degradation.
- It is often possible to reduce the idling overhead associated with locks using an alternate function, `pthread_mutex_trylock`.

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

`pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.

Alleviating Locking Overhead (Example)

```c
void *find_entries(void *start_pointer) {
    /* This is the thread function */
    struct database_record *next_record;
    int count;
    current_pointer = start_pointer;
    do {
        next_record = find_next_entry(current_pointer);
        count = output_record(next_record);
    } while (count < requested_number_of_records);
}
int output_record(struct database_record *record_ptr) {
    int count;
    pthread_mutex_lock(&output_count_lock);
    output_count ++;
    count = output_count;
    pthread_mutex_unlock(&output_count_lock);
    if (count <= requested_number_of_records)
        print_record(record_ptr);
    return (count);
}
```

<u>Finding k matches in a list:</u>

Each thread searches n/p entries.

If the time for a lock-update count-unlock cycle is t_1 and the time to find an entry is t_2, then the total time for satisfying the query is $(t_1 + t_2) \times n_{max}$, where n_{max} is the maximum number of entries found by any thread.

If t_1 and t_2 are comparable, then locking leads to considerable overhead.

Alleviating Locking Overhead (Example)

innovate achieve lead

```
/* rewritten output_record function */
int output_record(struct database_record *record_ptr) {
    int count;
    int lock_status;
    lock_status=pthread_mutex_trylock(&output_count_lock);
    if (lock_status == EBUSY) {
        insert_into_local_list(record_ptr);
        return(0);
    }
    else {
        count = output_count;
        output_count += number_on_local_list + 1;
        pthread_mutex_unlock(&output_count_lock);
        print_records(record_ptr, local_list,
        requested_number_of_records - count);
        return(count + number_on_local_list + 1);
    }
}
```

<u>Finding k matches in a list:</u>

Each thread searches n/p entries.

If the time for a lock-update count-unlock cycle is t_1 and the time to find an entry is t_2, then the total time for satisfying the query is $(t_1 + t_2) \times n_{max}$, where n_{max} is the maximum number of entries found by any thread.

If t_1 and t_2 are comparable, then locking leads to considerable overhead.

Producer-Consumer Using Condition Variables

Indiscriminate use of locks can result in idling overhead from blocked threads.

While the function pthread_mutex_trylock alleviates this overhead, it introduces the overhead of polling for availability of 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);
        }
    }
}
```

Producer-Consumer Using Condition Variables

We use two condition variables cond_queue_empty and cond_queue_full for specifying empty and full queues respectively.

The predicate associated with cond_queue_empty is task_available == 0, and cond_queue_full is asserted when task_available == 1.

When a thread performs a condition wait, it takes itself off the runnable list - consequently, it does not use any CPU cycles until it is woken up. This is in contrast to a mutex lock which consumes CPU cycles as it polls for the lock.

```
void *producer(void *producer_thread_data) {
int inserted;
while (!done()) {
    create_task();
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 1)
        pthread_cond_wait(&cond_queue_empty,
            &task_queue_cond_lock);
    insert_into_queue();
    task_available = 1;
    pthread_cond_signal(&cond_queue_full);
    pthread_mutex_unlock(&task_queue_cond_lock);
}
}
```

```
void *consumer(void *consumer_thread_data) {
while (!done()) {
    pthread_mutex_lock(&task_queue_cond_lock);
    while (task_available == 0)
        pthread_cond_wait(&cond_queue_full,
            &task_queue_cond_lock);
    my_task = extract_from_queue();
    task_available = 0;
    pthread_cond_signal(&cond_queue_empty);
    pthread_mutex_unlock(&task_queue_cond_lock);
    process_task(my_task);
}
}
```

Controlling Thread and Synchronization Attributes

- The Pthreads API allows a programmer to change the default attributes of entities using attributes objects.
  - 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
  - Use pthread_attr_init to create an attributes object. 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 the attrributes object using function: `pthread_mutexattr_init`.
- The function `pthread_mutexattr_settype_np` can be used for setting the type of mutex specified by the mutex attributes object.

```
pthread_mutexattr_settype_np (
pthread_mutexattr_t *attr,
int type);
```

- Here, `type` specifies the type of the mutex and can take one of:
  - `PTHREAD_MUTEX_NORMAL_NP`
  - `PTHREAD_MUTEX_RECURSIVE_NP`
  - `PTHREAD_MUTEX_ERRORCHECK_NP`

Types of Mutexes

- Pthreads supports three types of mutexes - normal, recursive, and error-check.
  - All of these locks use the same functions for locking and unlocking; however, the type of lock is determined by the lock attribute.
- A normal mutex deadlocks if a thread that already has a lock tries a second lock on it.
- A recursive mutex 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.
- An error check mutex 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.

Search Tree

- Consider the following example of a thread searching for an element in a binary tree. To ensure that other threads are not changing the tree during the search process, the thread locks the tree with a single mutex tree_lock. The search function is as follows:

If tree_lock is a normal mutex, the first recursive call to the function search_tree ends in a deadlock since a thread attempts to lock a mutex that it holds a lock on.
A recursive mutex allows a single thread to lock a mutex multiple times. Each time a thread locks the mutex, a lock counter is incremented. Each unlock decrements the counter. For any other thread to be able to successfully lock a recursive mutex, the lock counter must be zero.

```
search_tree(void *tree_ptr)
{
    struct node *node_pointer;
    node_pointer = (struct node *) tree_ptr;
    pthread_mutex_lock(&tree_lock);
    if (is_search_node(node_pointer) == 1) {
        /* solution is found here */
        print_node(node_pointer);
        pthread_mutex_unlock(&tree_lock);
        return(1);
    }
    else {
        if (tree_ptr -> left != NULL)
            search_tree((void *) tree_ptr -> left);
        if (tree_ptr -> right != NULL)
            search_tree((void *) tree_ptr -> right);
    }
    printf("Search unsuccessful\n");
    pthread_mutex_unlock(&tree_lock);
}
```

Thread Cancellation

- The pthread_cancel(*thread*) function sends a cancellation request to the thread *thread*. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its **cancelability state** and **type**.
- Cancelability State
  - A thread's cancelability state set by pthread_setcancelstate can be enabled (the default for new threads) or disabled
  - If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation. If a thread has enabled cancellation, then its **cancelability type** determines when cancellation occurs
- Cancelability Type
  - A thread's cancellation type, set by pthread_setcanceltype, may be either asynchronous or deferred (the default for new threads).
- Asynchronous cancelability means that the thread can be canceled at any time (usually immediately)
- Deferred cancelability means that cancellation will be delayed until the thread next calls a function that is a cancellation point. A list of functions that are or may be cancellation points is here: http://man7.org/linux/man-pages/man7/pthreads.7.html
- Ref: http://man7.org/linux/man-pages/man3/pthread_cancel.3.html

Composite Synchronization Constructs

- By design, Pthreads provide support for a basic set of operations.
- Higher level constructs can be built using basic synchronization constructs.
- We discuss two such constructs - read-write locks and barriers.

Read-Write Locks

- In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
- A read lock is granted when there are other threads that may already have read locks.
- If there is a write lock on the data (or if there are queued write locks), the thread performs a condition wait.
- If there are multiple threads requesting a write lock, they must perform a condition wait.
- With this description, we can design functions for read locks mylib_rwlock_rlock, write locks mylib_rwlock_wlock, and unlocking mylib_rwlock_unlock.

Read-Write Locks

- The lock data type mylib_rwlock_t holds the following:
  - a count of the number of readers,
  - the writer (a 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 mutex read_write_lock associated with the shared data structure

```c
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;
```

Lock for reading

```
void mylib_rwlock_rlock(mylib_rwlock_t *l) {
/* if there is a write lock or pending writers,
perform condition wait.. else increment count
of readers and grant read lock */
pthread_mutex_lock(&(l -> read_write_lock));
while ((l -> pending_writers > 0) || (l -> writer > 0))
pthread_cond_wait(&(l -> readers_proceed),
&(l -> read_write_lock));
l -> readers ++;
pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Locking for writing

```c
void mylib_rwlock_wlock(mylib_rwlock_t *l) {
/* if there are readers or writers, increment pending
writers count and wait. On being woken, decrement pending
writers count and increment writer count */

pthread_mutex_lock(&(l -> read_write_lock));
while ((l -> writer > 0) || (l -> readers > 0)) {
l -> pending_writers ++;
pthread_cond_wait(&(l -> writer_proceed),
&(l -> read_write_lock));
}
l -> pending_writers --;
l -> writer ++;
pthread_mutex_unlock(&(l -> read_write_lock));
}
```

Unlocking

```
void mylib_rwlock_unlock(mylib_rwlock_t *l) {
/* if there is a write lock then unlock, else if there are read locks, decrement count of read locks. If the count is 0 and there is a pending writer, let it through, else if there are pending readers, let them all go through */
pthread_mutex_lock(&(l -> read_write_lock));
if (l -> writer > 0)
l -> writer = 0;
else if (l -> readers > 0)
l -> readers --;
pthread_mutex_unlock(&(l -> read_write_lock));
if ((l -> readers == 0) && (l -> pending_writers > 0))
pthread_cond_signal(&(l -> writer_proceed));
else if (l -> readers > 0)
pthread_cond_broadcast(&(l -> readers_proceed));
}
```

Barriers

- A barrier holds a thread until all threads participating in the barrier have reached it.
- Barriers can be implemented using a counter, a mutex and a condition variable.
- A single integer is used to keep 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 (and setting the count to the number of threads) wakes up all the threads using a condition broadcast.

Barriers

```
typedef struct {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed;
    int count;
} mylib_barrier_t;
void mylib_init_barrier(mylib_barrier_t *b) {
    b -> count = 0;
    pthread_mutex_init(&(b -> count_lock), NULL);
    pthread_cond_init(&(b -> ok_to_proceed), NULL);
}
```

```
void mylib_barrier (mylib_barrier_t *b, int num_threads)
 {
    pthread_mutex_lock(&(b -> count_lock));
    b -> count ++;
    if (b -> count == num_threads) {
        b -> count = 0;
        pthread_cond_broadcast(&(b -> ok_to_proceed));
    }
else
    while (pthread_cond_wait(&(b -> ok_to_proceed),
    &(b -> count_lock)) != 0);
pthread_mutex_unlock(&(b -> count_lock));
}
```

Barriers

- The barrier described above is called a linear barrier.
- The trivial lower bound on execution time of this function is therefore O(n) for n threads.
- This implementation of a barrier can be speeded up using multiple barrier variables organized in a tree.
- We use n/2 condition variable-mutex pairs for implementing a barrier for n threads.
- At the lowest level, threads are paired up and each pair of threads shares a single condition variable-mutex pair.
- Once both threads arrive, one of the two moves on, the other one waits.
- This process repeats up the tree.
- This is also called a log barrier and its runtime grows as O(log p).

Log Barrier

- Consider an instance of a barrier with eight threads.
- Threads 0 and 1 are paired up on a single leaf node. One of these threads is designated as the representative of the pair at the next level in the tree. In the above example, thread 0 is considered the representative and it waits on the condition variable ok_to_proceed_up for thread 1 to catch up. All even numbered threads proceed to the next level in the tree. Now thread 0 is paired up with thread 2 and thread 4 with thread 6. Finally thread 0 and 4 are paired. At this point, thread 0 realizes that all threads have reached the desired barrier point and releases threads by signaling the condition ok_to_proceed_down. When all threads are released, the barrier is complete.

Log Barrier

- It is easy to see that there are n - 1 nodes in the tree for an n thread barrier.
  - Each node corresponds to two condition variables, one for releasing the thread up and one for releasing it down, one lock, and a count of number of threads reaching the node.
- The tree nodes are linearly laid out in the array mylog_logbarrier_t with the n/2 leaf nodes taking the first n/2 elements, the n/4 tree nodes at the next higher level taking the next n/4 nodes and so on.

Log Barrier

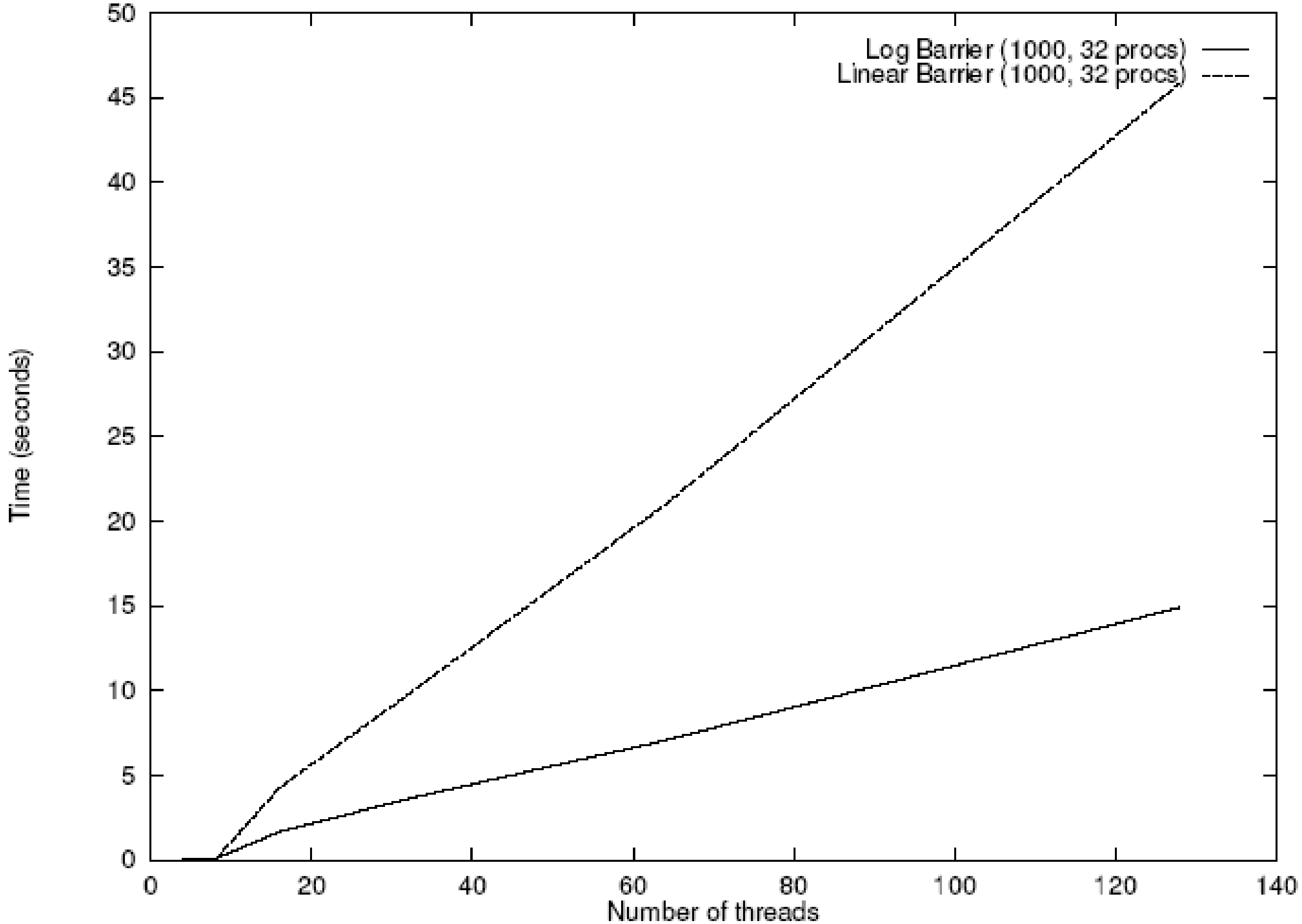
```
typedef struct barrier_node {
    pthread_mutex_t count_lock;
    pthread_cond_t ok_to_proceed_up;
    pthread_cond_t ok_to_proceed_down;
    int count;
} mylib_barrier_t_internal;
typedef struct barrier_node mylog_logbarrier_t[MAX_THREADS];
pthread_t p_threads[MAX_THREADS];
pthread_attr_t attr;

void mylib_init_barrier(mylog_logbarrier_t b) {
    int i;
    for (i = 0; i < MAX_THREADS; i++) {
        b[i].count = 0;
        pthread_mutex_init(&(b[i].count_lock), NULL);
        pthread_cond_init(&(b[i].ok_to_proceed_up), NULL);
        pthread_cond_init(&(b[i].ok_to_proceed_down), NULL);
    }
}
```

```c
void mylib_logbarrier (mylog_logbarrier_t b, int num_threads,
        int thread_id) {
    int i, base, index;      i=2;      base = 0;
    do {
        index = base + thread_id / i;
        if (thread_id % i == 0) {
            pthread_mutex_lock(&(b[index].Count_lock));
            b[index].Count ++;
            while (b[index].Count < 2)
                pthread_cond_wait(&(b[index].Ok_to_proceed_up),
                    &(b[index].Count_lock));
                pthread_mutex_unlock(&(b[index].Count_lock));
            }
            else {
                pthread_mutex_lock(&(b[index].Count_lock));
                b[index].Count ++;
                if (b[index].Count == 2)
pthread_cond_signal(&(b[index].Ok_to_proceed_up));
                while
(pthread_cond_wait(&(b[index].Ok_to_proceed_down),
                    &(b[index].Count_lock)) != 0);
                pthread_mutex_unlock(&(b[index].Count_lock));
                break;
            }
            base = base + num_threads/i;
            i=i*2;
    } while (i <= num_threads);
    i=i/2;
    for (; i > 1; i = i / 2) {
        base = base - num_threads/i;
        index = base + thread_id / i;
        pthread_mutex_lock(&(b[index].count_lock));
        b[index].count = 0;
pthread_cond_signal(&(b[index].ok_to_proceed_down));
pthread_mutex_unlock(&(b[index].count_lock));
    }
}
```

no	tid	base	i	index	count	wait on	signal
4	0	0	2	0	1	proceed_up	
4	1	0	2	0	2	proceed_down	proceed_up
4	0	2	4	2	1	proceed_up	
4	2	0	2	1	1	proceed_up	
4	3	0	2	1	2	proceed_down	proceed_up
4	2	2	4	2	2	proceed_down	proceed_up
4	0	3	8				
	outside loop						
4	0	2	4	2	0		proceed_down
4	0	0	2	0	0		proceed_down

Log Barrier



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

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You