Unit II -Programming Shared Address Space Platforms:

- ✓ Thread
  - ✓ Basics
  - ✓ The POSIX Thread API,
  - ✓ Thread Creation and Termination,
  - ✓ Synchronization Primitives in Pthreads,
  - ✓ Controlling Thread and Synchronization Attributes,
  - ✓ Thread Cancellation, Composite Synchronization Constructs.

Distributed memory systems

Recap

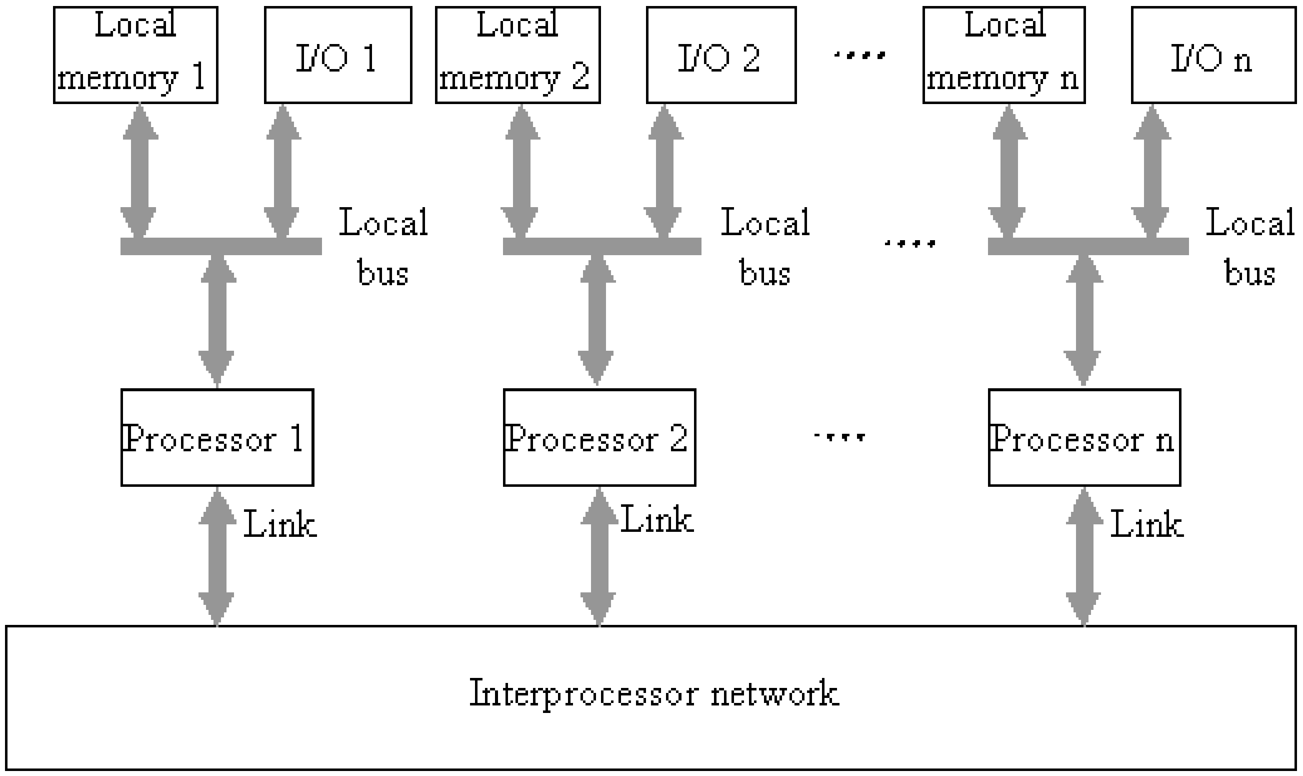


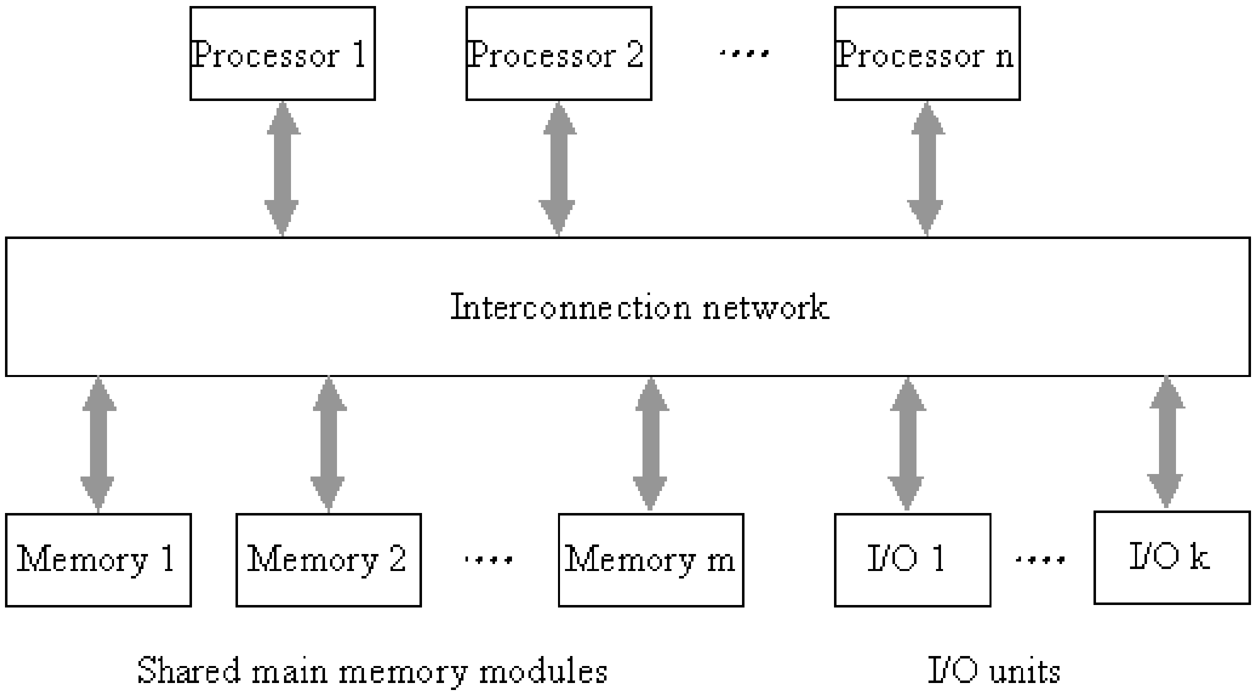
Fig: A multiprocessor system with a distributed memory (loosely coupled system)

Distributed memory

- Each processor has its own private memory.
- Computational tasks can only operate on local data,
- if remote data is required, the computational task must communicate with one or more remote processors. Communication through the **message passing**.

Recap

Shared memory systems



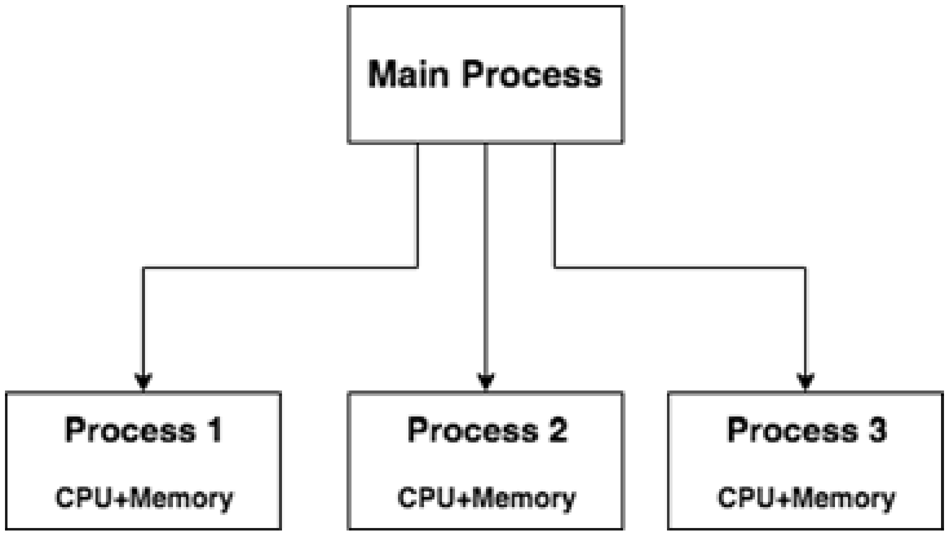
all processors can access all the main memory address space.

Shared variables access in the main memory

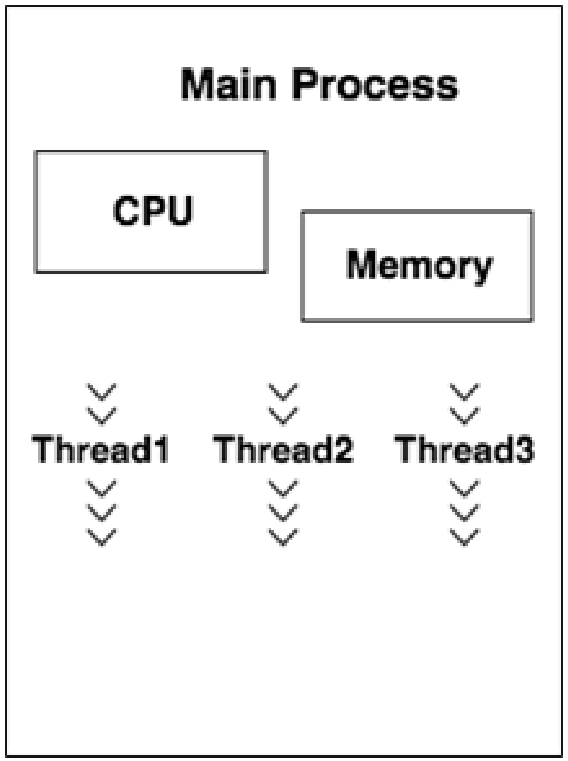
Fetching instructions for execution in processors is also done from a shared memory

A multiprocessor system with shared memory (tightly coupled system)

Multiprocessing



Multithreading



Threaded Programming Models

- Library-based models —
  - all data is shared, unless otherwise specified
  - Examples: Pthreads, Intel Threading Building Blocks, Java Concurrency, Boost, Microsoft .Net Task Parallel Library •
- Directive-based models
  - e.g., OpenMP —shared and private data —
  - Thread creation and synchronization
  - Programming languages —
    - Cilk Plus (Intel, GCC) —CUDA (NVIDIA) —Habanero-Java (Rice/Georgia Tech)

Parallel Programming

1. Synchronization between concurrent tasks
2. Communication of intermediate results

Shared address space architectures -

- Implicitly specified since some (or all) of the memory is accessible to all the processors.
- **Threads** assume that all memory is global

Thread Basics

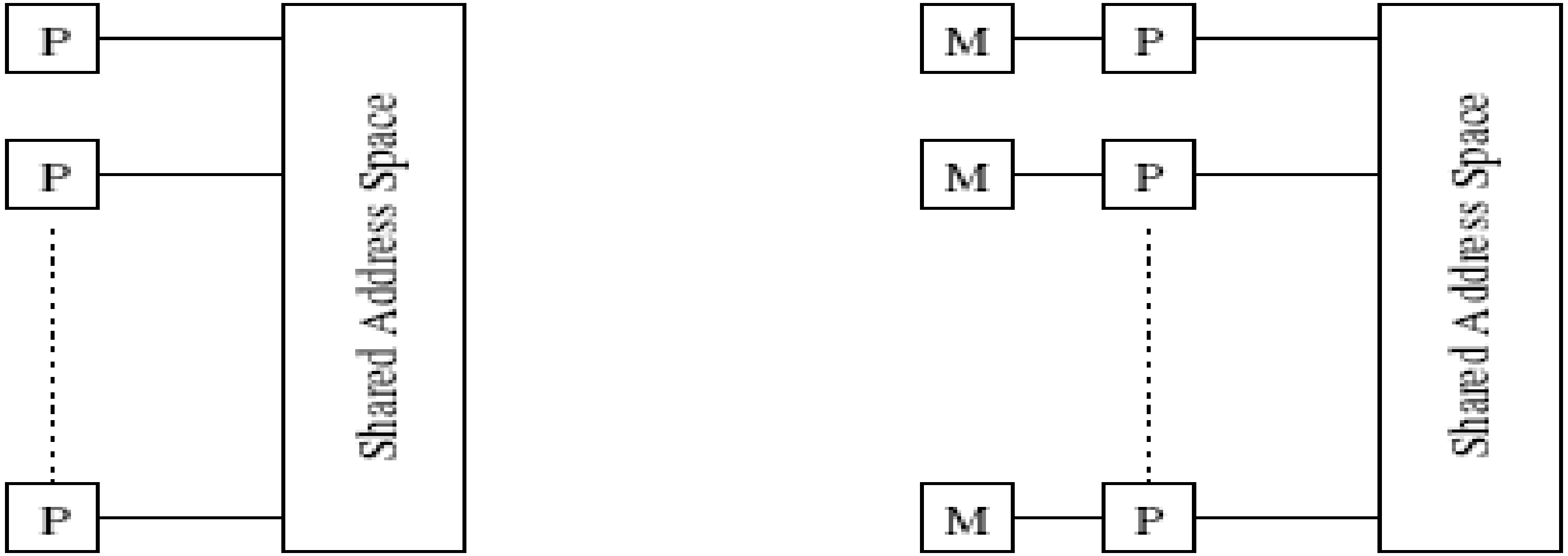
- A ***thread is a single stream of control in the flow of a program and it is a basic unit of CPU utilization.***

Thread

- •Thread ID
- •Program counter
- •Register
- •setStack

Shares resources with other threads within the same process

Thread Basics



- The logical machine model of a thread-based programming paradigm.

Explain the advantages of threaded programming models.

Why Threads?

- Threads provide software portability.
- Inherent support for latency hiding.
- Scheduling and load balancing.
- Ease of programming and widespread use.

Software Portability: Covered by standardized threading libraries like Pthreads.

Inherent support for latency hiding: Covered by responsiveness when one thread blocks.

Scheduling and load balancing: Covered by the kernel's ability to dispatch threads to different cores.

Ease of programming and widespread use: Covered by direct shared access and dominant use on shared-memory systems.

Advantage,Threaded Programming Model,Message Passing Model
1. Resource Sharing & Communication,"Efficient Communication: Threa
within a process share the same address space (memory, files).
Communication is fast, easy, and direct via shared variables.","Explicit
Communication: Processes have separate address spaces. Communicati
requires explicit data copying via mechanisms like sockets or pipes,
introducing overhead."
2. Economy & Overhead,"Low Overhead: Threads are ""lightweight"" an
require fewer system resources for creation and context switching, as th
OS doesn't have to switch the entire memory map.","High Overhead:
Processes are ""heavyweight,"" demanding more resources for creation
context switching, which slows down execution."
3. Latency Hiding,"Inherent Support for Latency Hiding: If one thread is
blocked (e.g., waiting for I/O), another thread in the same process can ru
immediately. This prevents the application from stalling and improves
responsiveness.",Latency hiding is less straightforward and requires
managing multiple fully independent processes.
4. Scheduling & Load Balancing,"Effective Scalability: The OS kernel
efficiently schedules threads across multiple CPU cores, enabling true
parallelism and maximizing hardware utilization for automatic load
balancing.","Requires the OS to manage and schedule separate,
independent entities (processes)."
5. Portability & Ease of Use,Software Portability: Standards like Pthreads
make concurrent code portable across different operating systems. Ease
Programming is often simpler for tasks that naturally share data.,"Portab
can sometimes be a concern, and programming explicit send/receive log
can add complexity."

Product of two dense matrices of size *n x n.*

```
• for (row = 0; row < n; row++)
      for (column = 0; column < n; column++)
          c[row][column] =
          dot_product( get_row(a, row),
                       get_col(b, col));
```

can be transformed to:

```
for (row = 0; row < n; row++)
  for (column = 0; column < n; column++)
          c[row][column] =
          create_thread( dot_product(get_row(a,
row),                                get_col(b,
col)));
```

The POSIX Thread API

- Number of vendors provide vendor-specific thread APIs
- Pthreads-IEEE specifies a standard 1003.1c-1995, POSIX API
- POSIX has emerged as the standard threads API
- Other thread API: NT threads, Solaris threads, Java threads, etc

Thread Basics: Creation and Termination

- Pthreads provides two basic functions for specifying concurrency in a program:

```
#include <pthread.h>
int pthread_create (
   pthread_t *thread_handle, const pthread_attr_t *attribute,
   void * (*thread_function)(void *),
   void *arg);
int pthread_join (
   pthread_t thread,
   void **ptr);
```

- The function pthread_create invokes function thread_function as a thread

Create thread

- int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*thread_function)(void *), void *arg);

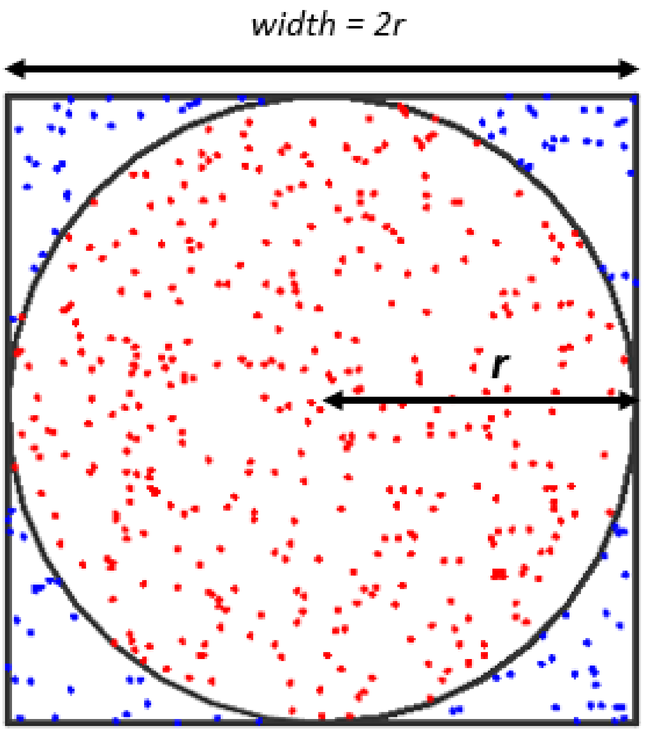
- 1st arg – pointer to the identifier of the created thread
- 2nd arg – thread attributes. If null, then the thread is created with default attributes
- 3rd arg – pointer to the function the thread will execute
- 4th arg – the argument of the executed function
- returns 0 for success

Waiting threads

int pthread_join(pthread_t thread, void **thread_return)

- main thread will wait for daughter thread *thread* to finish
- 1st arg – the thread to wait for
- 2nd arg – pointer to a pointer to the return value from the thread
- returns 0 for success
- threads should always be joined; otherwise, a thread might keep on running even when the main thread has already terminated

Estimating Pi using the Monte Carlo Method



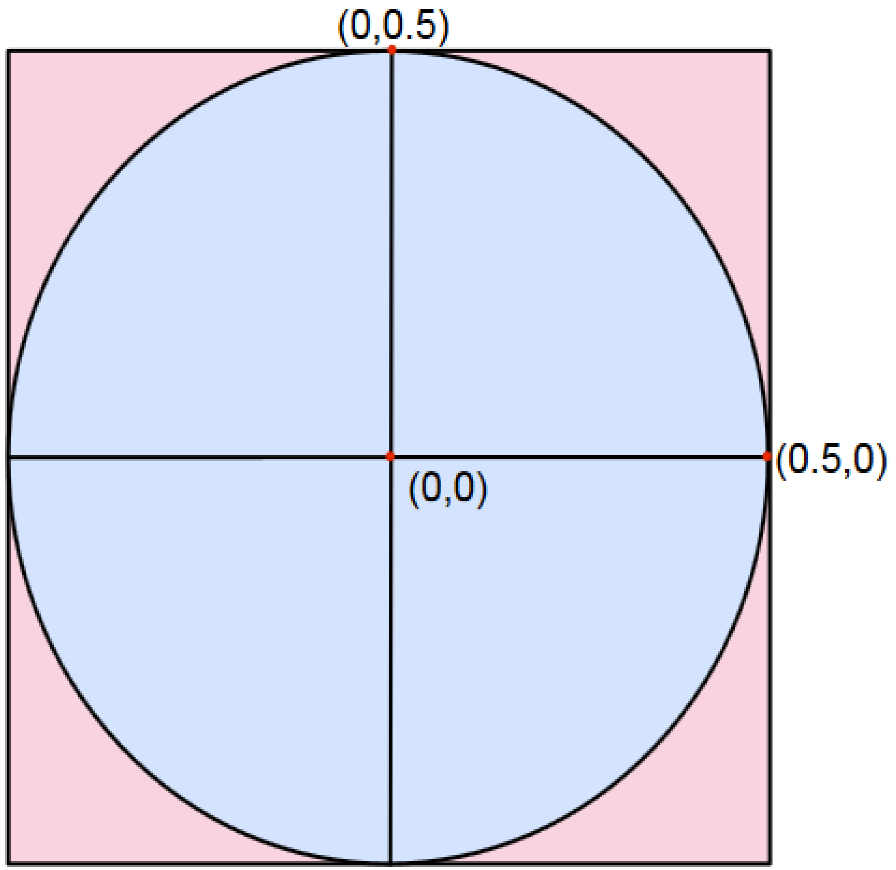
The area of the circle is πr^2,
The area of the square is width^2 = $(2r)^2 = 4r^2$.
If we divide the area of the circle, by the area of the square we get $\pi/4$.

$\pi \approx 4$ x (number of points in the circle / total number of points)

Running Example: Monte Carlo Estimation of Pi

Approximate Pi

- generate random points with x, y $\in$ [-0.5, 0.5]
- test if point inside the circle, i.e., $x^2 + y^2 < (0.5)^2$
- ratio of circle to square = $\pi r^2 / 4r^2 = \pi / 4$
- $\pi \approx 4$ * (number of points inside the circle) / (number of points total)



Thread Basics: Creation and Termination (Example)

```
#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];
     }
     ...
}
```

Thread Basics: Creation and Termination (Example)

```
#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];
     }
     ...
}
```

```c
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
    ...
    pthread_t p_threads[NUM_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];
    }
```

default attributes

thread function

thread argument

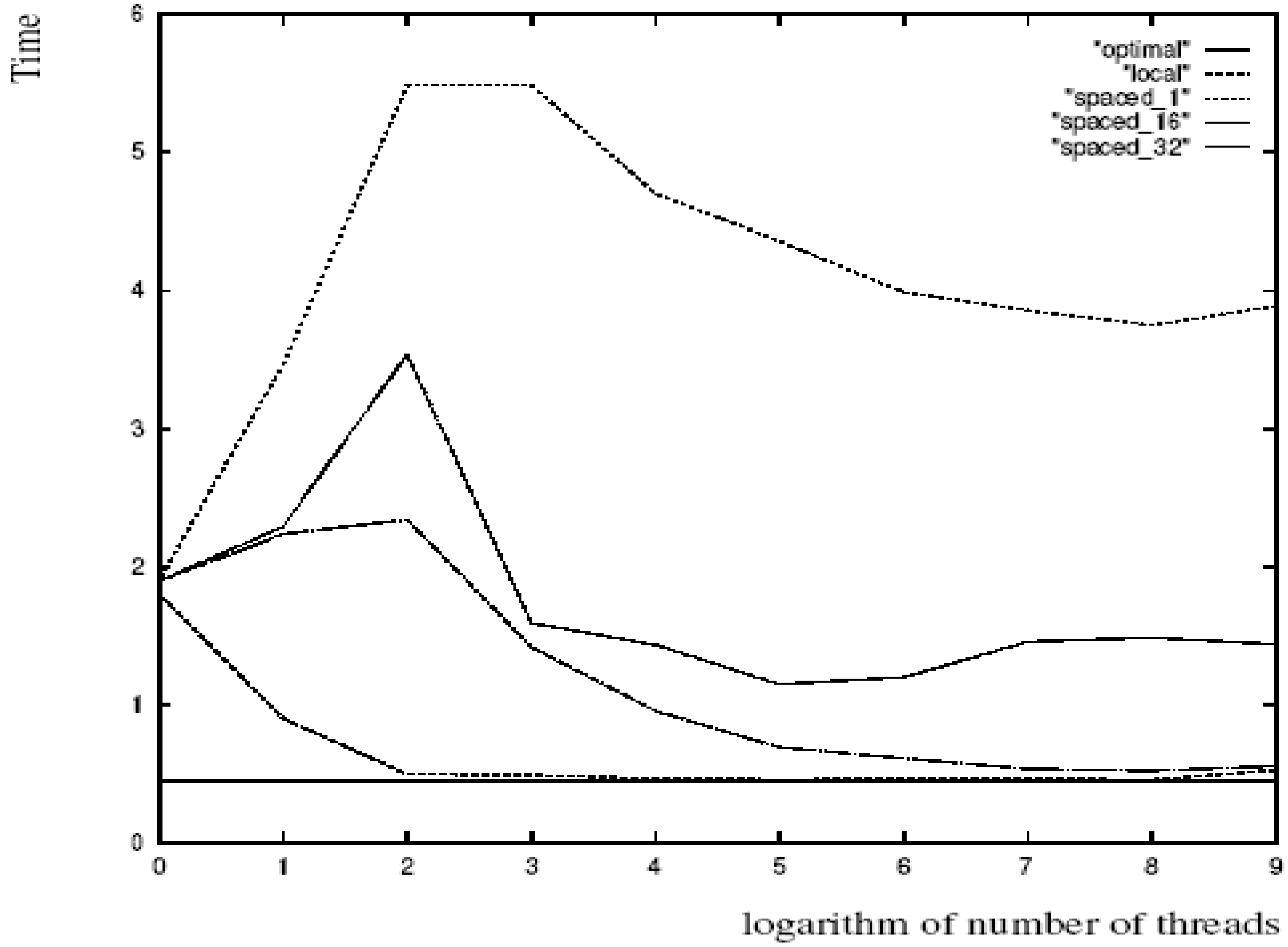
Thread Basics: Creation and Termination (Example)

```
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);
}
```

Programming and Performance Notes

- Note the use of the function `rand_r` (instead of superior random number generators such as `drand48`).
- Executing this on a 4-processor SGI Origin, we observe a 3.91 fold speedup at 32 threads. This corresponds to a parallel efficiency of 0.98!
- We can also modify the program slightly to observe the effect of false-sharing.
- The program can also be used to assess the secondary cache line size.

Programming and Performance Notes



- Execution time of the `compute_pi` program.

Discuss the synchronization primitives in pthreads.

Synchronization Primitives in Pthreads

Mutual Exclusion for Shared Variables

- Tasks work together to manipulate data and accomplish a given task.
- When multiple threads attempt to manipulate the same data item the results can often be incoherent if proper care is not taken to synchronize them
- Much of the effort associated with writing correct threaded programs is spent on synchronizing concurrent threads with respect to their data accesses or scheduling

The variable my_cost is thread-local and best_cost is a global variable shared by all threads.

```
/* each thread tries to update variable best_cost as follows */
if (my_cost < best_cost)
    best_cost = my_cost;
```

- Assume that there are two threads,
- The initial value of best_cost is 100,
- The values of my_cost are 50 and 75 at threads t1 and t2, respectively.
- If both threads execute the condition inside the if statement concurrently, then both threads enter the then part of the statement.
- Depending on which thread executes first, the value of best_cost at the end could be either 50 or 75.

There are two problems here:

1. non-deterministic nature of the result;
2. more importantly, the value 75 of best_cost is inconsistent in the sense that no serialization of the two threads can possibly yield this result.

Result of the computation depends on the race between competing threads

- Critical segment- segment that must be executed by only one thread at any time.
- Threaded APIs provide support for implementing critical sections and atomic operations using

mutex-locks

Mutual Exclusion

- Critical segments in Pthreads are implemented using mutex locks.
- Mutex-locks have two states:
  - locked and unlocked
  - At any point of time, only one thread can lock a mutex lock. A lock is an atomic operation.
- A thread entering a critical segment first tries to get a lock. It goes ahead when the lock is granted.

- Mutex locks enforce mutual exclusion in Pthreads
  - mutex lock states: locked and unlocked
  - only one thread can lock a mutex lock at any particular time
- Using mutex locks
  - request lock before executing critical secti
  - enter critical section when lock granted
  - release lock when leaving critical section

created by
`pthread_mutex_attr_init`
specifies mutex type

Mutual Exclusion

The Pthreads API provides the following functions for handling mutex-locks:

```
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);
```

Mutual Exclusion

- We can now write our previously incorrect code segment as:

```
pthread_mutex_t minimum_value_lock;
...
main() {
    ....
    pthread_mutex_init(&minimum_value_lock, NULL);
    ....
}
void *find_min(void *list_ptr) {
    ....
    pthread_mutex_lock(&minimum_value_lock);
    if (my_min < minimum_value)
    minimum_value = my_min;
    /* and unlock the mutex */
    pthread_mutex_unlock(&minimum_value_lock);
}
```

function to initialize a mutex-lock to its unlocked state

- **If the mutex-lock is already locked, the calling thread blocks**
- **otherwise the mutex-lock is locked and the calling** thread returns

- a thread must unlock the mutex-lock
- **If it does not do so, no other thread will be able to enter this section,**
- **(typically resulting deadlock.)**

Producer-Consumer Using Locks

Constraints

- 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.

```
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 (work_available == 0) {
                consumer_work = my_task; work_available = 1;
                inserted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
    }
}
```

critical section

```c
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 (work_available == 1) {
                my_task = consumer_work;
                work_available = 0;
                extracted = 1;
            }
            pthread_mutex_unlock(&task_queue_lock);
        }
        process_task(my_task);
    }
```

critical section

Mutex Types

```
pthread_mutex_init(&minimum_value_lock, NULL);
```

Mutex Types

- Normal
  - —thread deadlocks if tries to lock a mutex it already has locked
- Recursive
  - — single thread may lock a mutex as many times as it wants – increments a count on the number of locks —thread relinquishes lock when mutex count becomes zero
- Errorcheck
  - —report error when a thread tries to lock a mutex it already locked —report error if a thread unlocks a mutex locked by another

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 s
  - since it does not have to deal with queues associated with locks for multiple threads waiting on the lock.
  - enables a thread to do something else if a lock is unavailable

Alleviating Locking Overhead (Example)

```
/* Finding k matches in a list */
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);
}
```

Alleviating Locking Overhead (Example)

```
/* 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);
    }
}
```

- ✓ Thread
  - ✓ Basics
  - ✓ The POSIX Thread API,
  - ✓ Thread Creation and Termination,
  - ✓ Synchronization Primitives in Pthreads,
  - ✓ Controlling Thread and Synchronization Attributes,
  - ✓ Thread Cancellation, Composite Synchronization Constructs.

- ✓ Thread
  - ✓ Basics
  - ✓ The POSIX Thread API,
  - ✓ Thread Creation and Termination,
  - ✓ Synchronization Primitives in Pthreads,
  - ✓ Controlling Thread and Synchronization Attributes,
  - ✓ Thread Cancellation, Composite Synchronization Constructs.

Condition Variables for Synchronization

- A condition variable is a data object used for synchronizing threads
- This variable allows a thread to block itself until specified data reaches a predefined state
- Always use condition variables together with a mutex lock.

The shared variable task_available must become 1 before the consumer threads can be signaled.
The boolean condition task_available == 1 is referred to as a predicate.

- The condition variables to atomically block threads until a particular condition is true.

Condition Variables for Synchronization

- If the predicate is not true, the thread waits on the condition variable associated with the predicate using the function `pthread_cond_wait`.
- **int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);**
  - A call to this function blocks the execution of the thread until it receives a signal from another thread

int pthread_cond_signal(pthread_cond_t *cond);

1. Unblocks at least one thread that is currently waiting on the condition variablecond.
2. The producer then relinquishes its lock on mutex by explicitly calling pthread_mutex_unlock
3. allowing one of the blocked consumer threads to consume the task

int pthread_cond_broadcast(pthread_cond_t *cond);

wake all threads that are waiting on the condition variable

- Using a condition variable —
  - thread can block itself until a condition becomes true
    - thread locks a mutex
    - tests a predicate defined on a shared variable if predicate is false, then wait on the condition variable waiting on condition variable unlocks associated mutex
  - when some thread makes a predicate true
    - Thread can signal the condition variable to either wake one waiting thread wake all waiting threads
    - –when thread releases the mutex, it is passed to first waiter

Condition Variables for Synchronization

int pthread_cond_init (pthread_cond_t *cond, const pthread_condattr_t *attr);

Initializes a condition variable (pointed to by cond) whose
Attributes are defined in the attribute object attr

int pthread_cond_destroy(pthread_cond_t *cond);

If at some point in a program a condition variable is no longer required,

**int pthread_cond_timedwait (pthread_cond_t *cond, pthread_mutex_t *mutex,
const struct timespec *abstime);**

1. Thread can perform a wait on a condition variable until a specified
time expires.
2. At this point, the thread wakes up by itself if it does not receive a signal or a
broadcast.

Write a program of producer consumer using conditional variables.

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_init();
    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 */
}
```

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)
            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);
    }
}
```

Producer-Consumer Using Condition Variables

```
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.

Int pthread_attr_init (pthread_attr_t *attr);

initializes the attributes object attr to the default values

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`

Thread Cancellation

int pthread_cancel (pthread_t thread);

- **When a call to this function is made, a cancellation is sent to the specified Thread**
- **The function returns a 0 on successful completion.**

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.

What are read-write locks? Discuss about its role in controlling and synchronizing the threads.

Read-Write Locks

- In many applications, a data structure is read frequently but written infrequently. For such applications, we should use read-write locks.
  - multiple reads can proceed without any coherence problems. However, writes must be serialized.
- 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

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 = l -> writer = 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) {
    /* 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));
}
```

Read-Write Locks

```
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));
}
```

Read-Write Locks

```
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));
}
```

With syntax explain the barrier and critical directives.

Barriers

- As in MPI, 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);
}
```

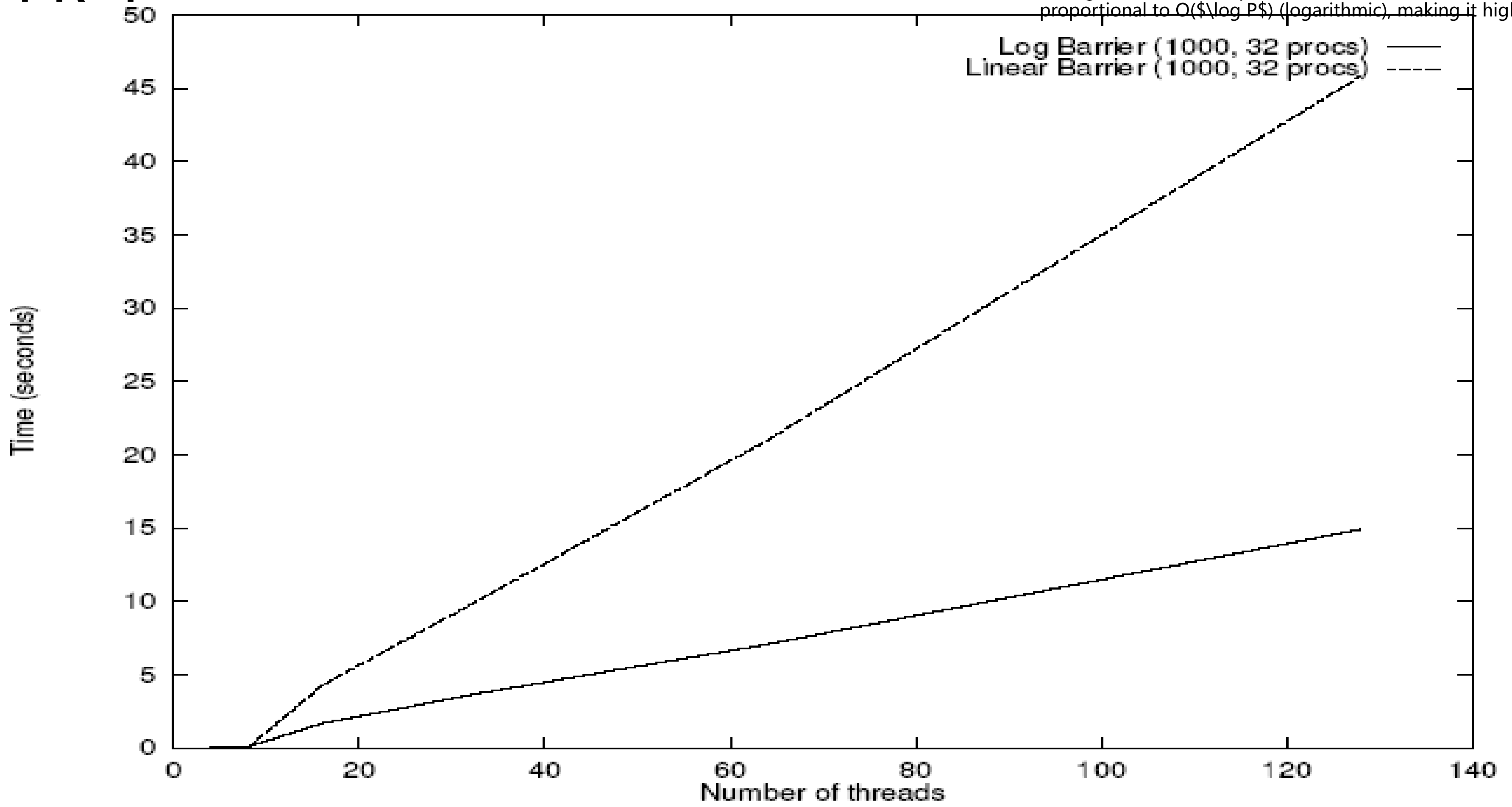
Barriers

```
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));
}
```

Barrier

Differentiate between log barrier and linear barrier.

A. Linear Barrier (Simple/Centralized Barrier)Mechanism: Threads arrive and atomically increment a single counter. When the counter equals the number of threads (P), the last thread signals all ot waiting threads to proceed (often via a flag or condition variable).Structure: Centralized (all thread single point).Scalability: Poor. The atomic increment/decrement of the central counter becomes a contention bottleneck for a large number of threads (P). The synchronization time is O(P) (lineai to serialization.B. Log Barrier (Tree/Butterfly Barrier)Mechanism: Threads synchronize in stages, for a tree or butterfly pattern. A thread only needs to wait for $\log_2 P$ partners before proceeding next stage.Structure: Distributed/Tree-based.Scalability: Excellent. Since the synchronization is han in stages that combine pairs of threads, the contention is reduced. The synchronization time is proportional to O($\log P$) (logarithmic), making it highly scalable for large thread counts.



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

Implement DAXPY loop using pthread(32threads)

4. **DAXPY Loop:**
The daxpy loop is the core of the benchmark. This loop is used to measure the performance. By using this loop we can observe how fast a certain machine can execute. Daxpy loop multiplies a vector by a scalar and adds it to another vector.
D stands for Double precision, A is a scalar value, X and Y are one-dimensional vectors of size 216 each, P stands for Plus. The operation to be completed in one iteration i.e `X[i] = a*X[i] + Y[i]`.

CS72-HPC-assignment A-1: Pthreads and OpenMp

```
void daxpy(double y[], double a, double x[], int n)
    {
    int i;
    for (i = 0; i &lt; n; i++)
    y[i] = a*x[i] + y[i];
    }
```