Threads

Process

- What Is a Process?:
  - A program on execution (or suspended)
  - What the OS is paying attention to
    - Processor registers
    - Memory used by the task
      - includes the executable code,
      - process-specific data (input and output),
      - stack
      - heap
      - Registers (PC,…)

Thread

- Smallest unit of processing that can be scheduled by an operating system.
- Is also called “light weight process”. In most cases, a thread is contained inside a process. They share almost everything from the process but the stack and the PC. Multiple threads can exist within the same process and share resources:
  - Memory
    - Instructions (code)
    - Context (the values that its variables reference at any given moment).

Threads

- Threads are lightweight processes
- Each process can execute several threads
  - The threads execute <u>independently</u>
  - Threads <u>share</u> the global variables, programs code, heap (malloc'd memory), and OS resources
  - Each thread has its <u>own stack</u> and follow its <u>own execution flow</u>

Thread

- single processor, **multithreading** occurs by time-division multiplexing: the processor switches between different threads.
- multiprocessor, the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.

Thread

- In this chapter we use POSIX threads, Pthreads.
- POSIX is a standard for Unix like OS
- Pthreads is not a programming language; it is a library that can be linked with C programs

Threads

- In practice
  - ➢ Main program creates threads
    - By specifying an entry point function and an argument.
  - ➢ The main program and each created thread run independently.
    - They share global variables.
    - They do not share local variables.

Threads

- Hello world.
- Main function creates several threads . Each thread prints message and quits

Compilation:

```
gcc –g –Wall –o pth_hello pth_hello.c –lpthread
```

Run:

```
./pth_hello <number of threads>
```

Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/*global variable accesible to all threads*/
long threads_count;
void* Hello(void* rank);

int main(int argc, char* argv[]){
  long thread;
  pthread_t* thread_handles;
  //get number of threads
  threads_count=strtol(argv[1], NULL, 10);
  thread_handles=malloc(threads_count*sizeof(pthread_t));
  for(thread=0; thread<threads_count; thread++)
     pthread_create(&thread_handles[thread], NULL, Hello, (void*) thread);
  printf("Hello from the main thread\n");
  for(thread=0; thread<threads_count; thread++)
     pthread_join(thread_handles[thread], NULL);
   free(thread_handles);
   return 0;
}
void* Hello(void* rank){
    long my_rank = (long) rank;
    printf("hello from thread %ld of %ld\n", my_rank, threads_count);
    return NULL;
}
```

Threads

In Pthreads programmers do not control where the threads are run.

Thread placement is controlled by OS

Threads

- Example of functions to handle threads
  - Creation
  - Exit
  - Cancellation
  - Synchronization

Threads

- Example
  - Alternating threads
    - Creates 3 threads
    - Let the system execute them in a round-robin fashion
    - Wait for them to finish at main

Threads

```
int
main ( )
{
    int   i;

    for (i = 0; i < 3; i++)
            create thread to execute function loop with parameter i

    wait for each thread to finish
}
```

Threads

```
void
loop (int n)
{
        int i;


         for (i = 0; i < 20; i++)
        {
                printf("Thread %d\n", n);
                sleep (1); //cause the calling thread to be suspended from                                                  //
       execution until either the number of realtime                                          //seconds specified by
       the argument seconds has                                            //elapsed or a signal is delivered to the calling
                                                         //thread and its action is to invoke a signal-catching
                     //function or to terminate the process.
                                       //The suspension time may be longer than requested                                  //
       due to the scheduling of other system activity
        }
}
```

pthread library

- We will use pthread library
  - Main information
    - man pthread
  - There are man pages for specific functions
  - Include ".h" file as shown in the man page:

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

  - Compile using –lpthread

```
gcc main.c –o myprogram -lpthread
```

pthread library

Operations

```
#include <pthread.h>

pthread_t thread;

int pthread_create (
                pthread_t *restrict thread,           // thread id
                const pthread_attr_t *restrict attr,  // attributes
                void *(*start_routine)(void *),                // function
                void *restrict arg);                                   // argument
//0 is returned on success, nonzero otherwise
```

Note: C99, The restrict keyword says that for the lifetime of the pointer, only it or a value directly derived from it (such as pointer + 1) will be used to access the object to which it points.

pthread library

Operations

void pthread_exit (void *value_ptr);

//inside a thread function, a thread can terminate itself by returning from the thread function or by calling pthread_exit

int pthread_cancel(pthread_t thread);

//stop a thread currently on execution

int pthread_join (pthread_t thread, void **value_ptr);

// suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. When a pthread_join() returns successfully, the target thread has been terminated. Note: makes one thread wait until the other one finish

pthread library

- Example using pthreads
  - Alternating threads
    - Creates 3 threads
    - Let the system execute them in a round-robin fashion
    - Wait for them to finish at main

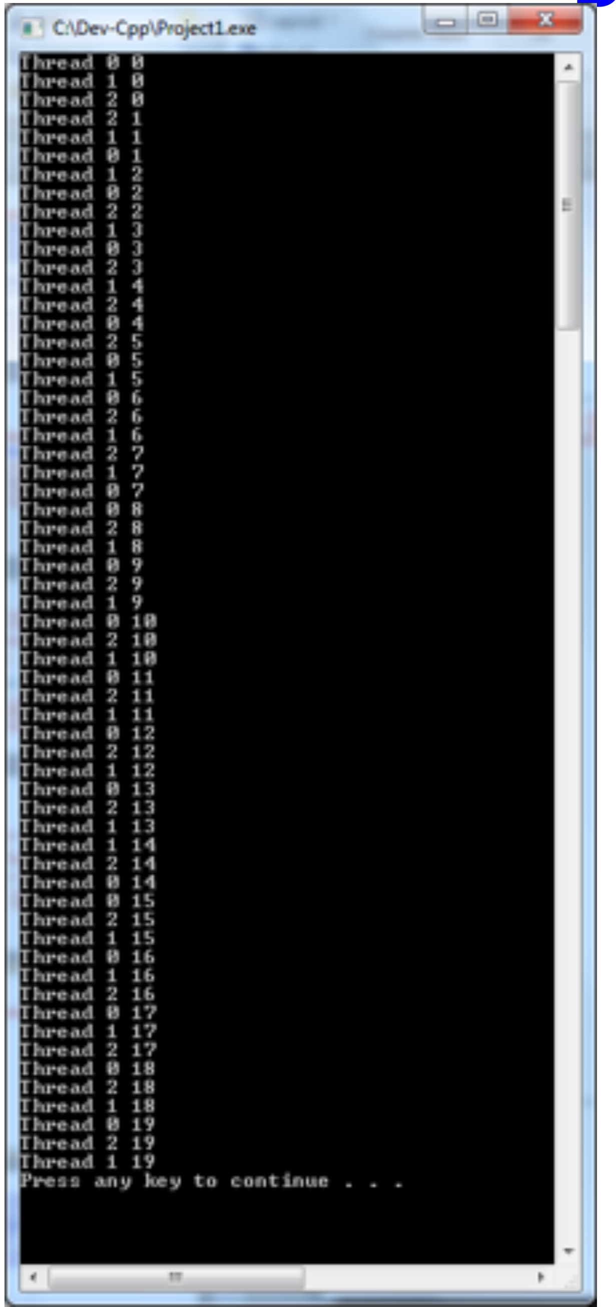
pthread library

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

void *loopThread (void *arg);

int main()
{
      int                    i;
      pthread_t          thr[3];
      for (i = 0; i < 3; i++)
            pthread_create (&thr[i], NULL, loopThread, (void *) i);
      for (i = 0; i < 3; i++)
            pthread_join (thr[i], NULL);
     return 0;
}
```

pthread library

```
void *
loopThread (void *arg)
{
        int i;
        int threadNo = (int) arg;

        for (i = 0; i < 20; i++)
        {
                printf("Thread %d %d\n", threadNo,i);
                sleep (1);
        }
        return NULL;
}
```

pthread library



```
Thread 0 0
Thread 1 0
Thread 2 0
Thread 2 1
Thread 1 1
Thread 0 1
Thread 1 2
Thread 0 2
Thread 2 2
Thread 1 3
Thread 0 3
Thread 2 3
Thread 1 4
Thread 2 4
Thread 0 4
Thread 2 5
Thread 0 5
Thread 1 5
Thread 0 6
Thread 2 6
Thread 1 6
Thread 2 7
Thread 1 7
Thread 0 7
Thread 0 8
Thread 2 8
Thread 1 8
Thread 0 9
Thread 2 9
Thread 1 9
Thread 0 10
Thread 2 10
Thread 1 10
Thread 0 11
Thread 2 11
Thread 1 11
Thread 0 12
Thread 2 12
Thread 1 12
Thread 0 13
Thread 2 13
Thread 1 13
Thread 1 14
Thread 2 14
Thread 0 14
Thread 0 15
Thread 2 15
Thread 1 15
Thread 0 16
Thread 1 16
Thread 2 16
Thread 0 17
Thread 1 17
Thread 2 17
Thread 0 18
Thread 2 18
Thread 1 18
Thread 0 19
Thread 2 19
Thread 1 19
Press any key to continue . . .
```

NO pthread_join

pthread library

```
#include <stdio.h>
#include <pthread.h>
/* This is our thread function. */
void *threadFunc(void *arg) {
        char *str;
        int i = 0;
        str=(char*)arg;
        while(i < 10 ) {
                usleep(1);
                printf("threadFunc says: %s\n",str);
                ++i;
        }
        return NULL;
}
```

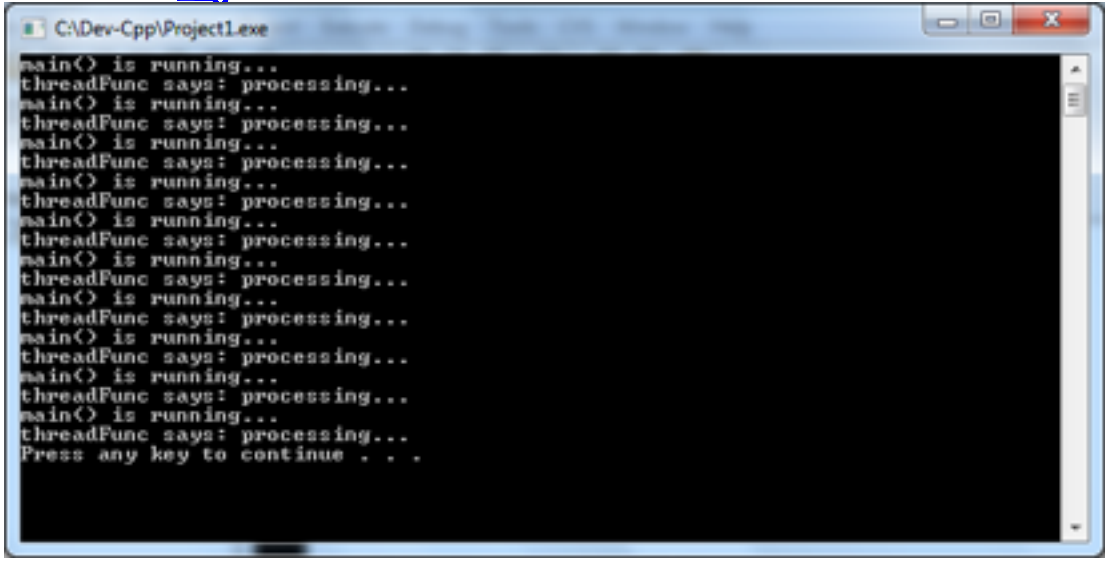
pthread library

```
int main(void) {
		pthread_t pth; // this is our thread identifier
		int i = 0; /* Create worker thread */
pthread_create(&pth,NULL,threadFunc,"processing...");
		/* wait for our thread to finish before continuing */		pthread_join(pth,
NULL);
		while(i < 10 ) {
				usleep(1);
				printf("main() is running...\n"); ++i;
		}
		return 0;
}
```

This will create a lot of text from thread function and then a bunch of text from main()

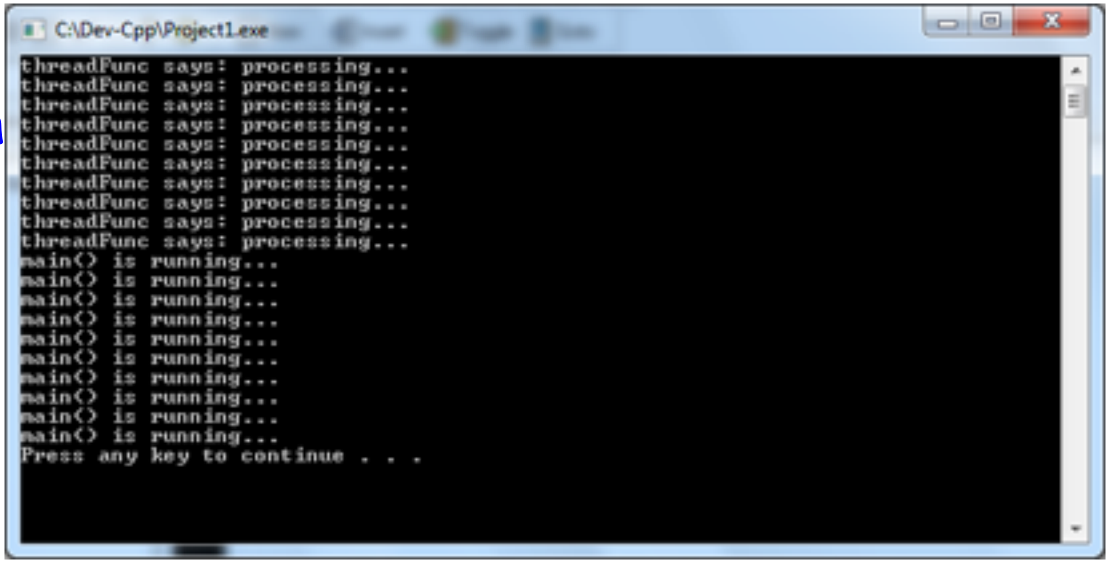
pthread library

NO pthread_join:



```
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
main() is running...
threadFunc says: processing...
Press any key to continue . . .
```

With



```
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
threadFunc says: processing...
main() is running...
main() is running...
main() is running...
main() is running...
main() is running...
main() is running...
main() is running...
main() is running...
main() is running...
main() is running...
Press any key to continue . . .
```

Performance considerations

- Thread creation is expensive
- Each thread gets his own stack
- Lock contention requires skills to keep as many threads as possible running

Matrix Vector multiplication

$$\begin{pmatrix} A_{00} & \cdots & A_{0n} \\ \vdots & \ddots & \vdots \\ A_{m0} & \cdots & A_{mn} \end{pmatrix} * \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

Code without using multithreading

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

Matrix Vector multiplication

Parallelize the problem by dividing work among threads

$$\begin{pmatrix} A_{00} & \cdots & A_{0n} \\ \vdots & \ddots & \vdots \\ A_{m0} & \cdots & A_{mn} \end{pmatrix} * \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

Lets create a thread to calculate each of the row operations so for example thread 0 will calculate the following

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

Matrix Vector multiplication

$$\begin{pmatrix} A_{00} & \cdots & A_{0n} \\ \vdots & \ddots & \vdots \\ A_{m0} & \cdots & A_{mn} \end{pmatrix} * \begin{bmatrix} x_0 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_0 \\ \vdots \\ y_n \end{bmatrix}$$

Lets create a thread to calculate each of the row operations so we will create i threads and each calculates.

Each thread needs to see values of x, so x[] should be global.

Same for A and y they need to be global

Assuming we can create one thread per row of the matrix:

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

Matrix Vector multiplication

```
void* p_matvec(void * id)
{
        int threadID=(int) id;
          int i,j;

        y[threadID]=0.0;
        for(j=0;j<nCOL;j++)
      y[threadID]+=A[threadID][j]*x[j];
        return NULL;
}
```

pthread library

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define nROW 5
#define nCOL 5
int A[nROW][nCOL]= {{1,2,3,4,5},{1,9,3,4,5},{1,2,10,4,5},
{1,2,3,4,22}, {31,2,3,4,5}};
int x[nCOL]={7,7,8,5,7};
int y[nCOL];
```

pthread library

```
int main(void) {
            pthread_t pth[nROW]; // this is our thread identifier
            int i;
            for(i=0;i<nROW;i++)
                pthread_create(&pth[i],NULL, p_matvec,(void *)i);
            /* wait for our thread to finish before continuing */
            for(i=0;i<nROW;i++)
                 pthread_join(pth[i], NULL);
            for(i=0;i<nROW;i++)
                        printf("%d \n", y[i]);
            system("pause");
            return 0;
}
```

pthread library

If we can not create a thread per row, and we only have thread_count threads then:

```
void* p_matvec(void * id)
{
        int threadID=(int) id;
                int i,j;
                int local_m=m/thread_count;
                int first_row= threadID *local_m;
                int last_row=first_row+local_m;
                for(i=my_first_row; i< last_row;i++){
            y[i]=0.0;
            for(j=0;j<nCOL;j++)
                    y[i]+=A[i][j]*x[j];
                }
        return NULL;
}
```

Calculate the value of π using Series

Matrix-vector multiplication was a very easy code because the shared memory locations were accessed in a highly desirable way.

After initialization all variables (except y were accessed only read)
And only one thread makes changes to an individual location of y.

Calculate the value of π using Series

$\Pi=4(1-1/3+1/5-1/7+\ldots+(-1)^{n}1/(2n+1)$

```
double factor=1.0;
double sum=0.0;

for(i=0;i<n;i++, factor=-factor){
        sum+=factor/(2*i+1);
}
pi=4*sum;
```

Threads

- Need synchronization
- Solution:
  - Locks
  - Conditional variables
  - Semaphores

Threads Basic Concepts:

- Multithreaded applications often require synchronization objects. These objects are used to protect memory from being modified by multiple threads at the same time, which might make the data incorrect.

Threads

- Mutex. A mutex is like a lock. A thread can lock it, and then any subsequent attempt to lock it, by the same thread or any other, will cause the attempting thread to block until the mutex is unlocked.
    - For example, imagine a very large linked list. If one thread deletes a node at the same time that another thread is trying to walk the list, it is possible for the walking thread to fall off the list, so to speak, if the node is deleted or changed.
- Technically speaking, only the thread that locks a mutex can unlock it, but sometimes operating systems will allow any thread to unlock it.
    - Doing this is, of course, a Bad Idea.

Locking

- Sharing variables
  - Requires mutual exclusion
  - Lock/unlock to avoid a race condition
- Have one lock for each independent critical region

Locking

```
//No synchronization -->> BAD!!

int count = 0;

void *update ( )
{
        int    i;

        while (i = 0; i < 1000; i++)
        {
            count++;
        }
}
```

```
//Using lock

int count = 0;

void *update ( )
{
        int    i;

        while (i = 0; i < 1000; i++)
        {
                lock
                count++;
                unlock
        }
}
```

Threads

- Semaphore. A semaphore is like a mutex that counts instead of locks. If it reaches zero, the next attempt to access the semaphore will block until someone else increases it. This is useful for resource management when there is more than one resource, or if two separate threads are using the same resource in coordination.

  Unlike mutexes, semaphores are designed to allow multiple threads to up and down them all at once. If you create a semaphore with a count of 1, it will act just like a mutex, with the ability to allow other threads to unlock it.

Threads

- Semaphore.
- Functions:

```
int sem_init(sem_t* semaphore_p, int shared,
unsigned initial_val);
int sem_destroy(sem_t* semaphore_p);
int sem_post(sem_t* semaphore_p);
int sem_wait(sem_t* semaphore_p);
```

Threads

- The third and final structure is the thread itself. More specifically, thread identifiers. These are useful for getting certain threads to wait for other threads, or for getting threads to tell other threads interesting things.

- code protected by mutexes and semaphores as **Critical Sections**.

pthread -- mutex lock

Functions

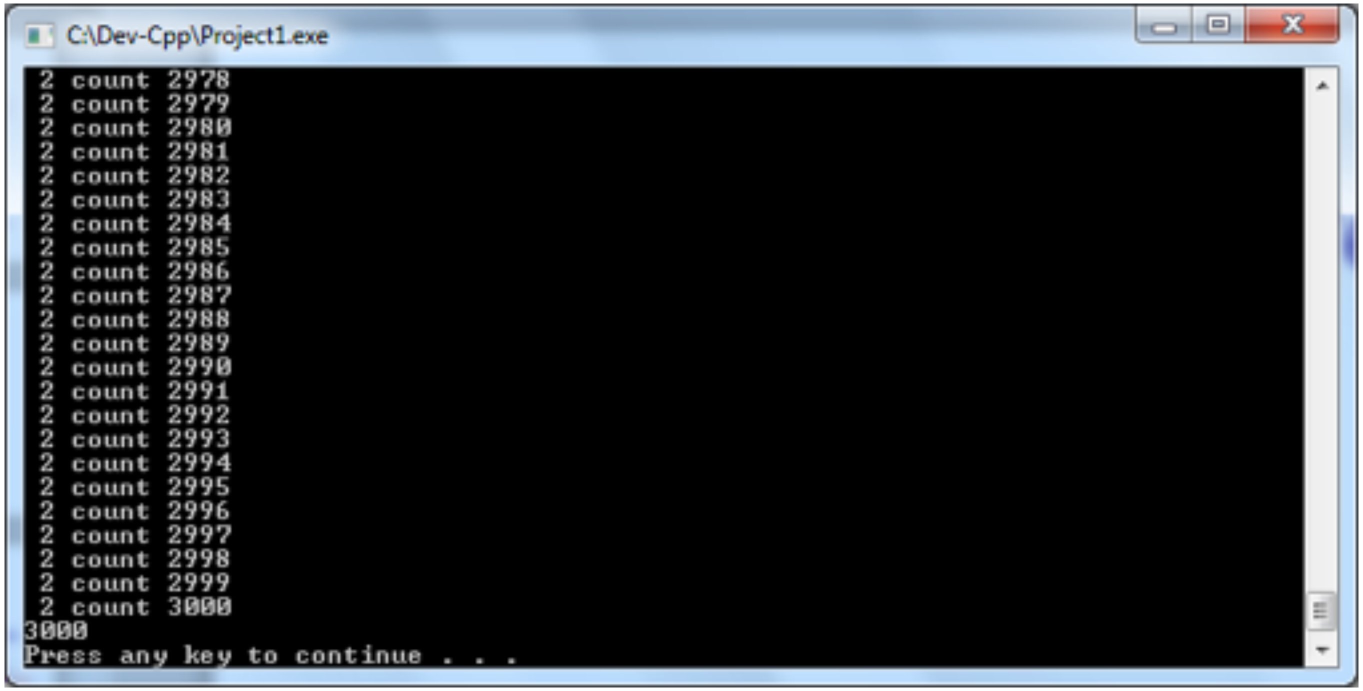
```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
        // macro initializes the static mutex mutex

INIT
int pthread_mutex_init (pthread_mutex_t *restrict mutex,
                                   const pthread_mutexattr_t *restrict attr);
LOCK
int pthread_mutex_lock (pthread_mutex_t *mutex);

UNLOCK
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

pthread -- mutex lock

```
//No synchronization -->> BAD!!

int count = 0;

void *update ( )
{
        int     i;

        while (i = 0; i < 1000; i++)
        {
            count++;
        }
}
```

```
//Using mutex lock
//pthread_mutex_init called in main

int count = 0;
pthread_mutex_t mutex =
     PTHREAD_MUTEX_INITIALIZER;

void *update ( )
{
      int     i;

      for (i = 0; i < 1000; i++)
      {
           pthread_mutex_lock (&mutex);
           count++;
           pthread_mutex_unlock (&mutex);
      }
```

pthread -- mutex lock



```
 2 count 2978
 2 count 2979
 2 count 2980
 2 count 2981
 2 count 2982
 2 count 2983
 2 count 2984
 2 count 2985
 2 count 2986
 2 count 2987
 2 count 2988
 2 count 2989
 2 count 2990
 2 count 2991
 2 count 2992
 2 count 2993
 2 count 2994
 2 count 2995
 2 count 2996
 2 count 2997
 2 count 2998
 2 count 2999
 2 count 3000
3000
Press any key to continue . . .
```

Threads -- Deadlocks

- Danger!
  - When one or more threads are waiting for one another and no thread can proceed
- Example
  - One thread is waiting for itself
  - Two threads are waiting for each other
  - Several threads are waiting for one another in a cycle

Calculate the value of π using Series

With multithreading:

```
void* thread_sum(void *id){
        long myid=(int)id;
        int i;
        double my_n=n/thread_count;
        double first_i=my_n*myid; double last_i=first_i+my_n;
        if(first_i % 2==0) factor=1.0;
        else factor=-1.0;
         for(i=first_i;i<last_i;i++, factor=-factor){
                pthread_mutex_lock(&mutex);
                sum+=factor/(2*i+1);
                pthread_mutex_unlock(&mutex);
        }
        pi=4*sum;
        return NULL;
}
```

Calculate the value of π using Series

The problem is that the previous code runs slower than the non multhithreaded one why?

Sol:

```
void* thread_sum(void *id){
            long myid=(int)id;
            int i; double my_sum=0.0;
            double my_n=n/thread_count;
            double first_i=my_n*myid; double last_i=first_i+my_n;
            if(first_i % 2==0) factor=1.0;
            else factor=-1.0;
             for(i=first_i;i<last_i;i++, factor=-factor){
                        my_sum+=factor/(2*i+1);
            }
            pthread_mutex_lock(&mutex);
            sum+=my_sum;
            pthread_mutex_unlock(&mutex);
            sum=4*sum; //for more efficiency do this just one on main()
            return NULL;
}
```

Time Performance Evaluation

```
#include <time.h>

clock_t start, end;
double elapsed;
start=clock();
....
....
end=clock(); //only counts CPU time used

elapsed=(end-start)/(double)CLOCKS_PER_SEC;
```

Time Performance Evaluation

```
int gettimeofday(struct timeval *tv, struct timezone *tz);
```

Obtains current time since the Epoch (January 1, 1970)

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

and gives the number of seconds and microseconds since the Epoch The tz argument is a struct timezone:

```
struct timezone {
    int tz_minuteswest;     /* minutes west of Greenwich */
    int tz_dsttime;         /* type of DST correction */
};
```

Time Performance Evaluation

- **time.c**

```
#include <time.h>
#include <sys/time.h>
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char **argv)
{
      if (argc < 2) {
                printf("USAGE: %s loop-iterations\n", argv[0]);
                return 1; }
      int iterations = atoi(argv[1]);
      struct timeval start, end;
      gettimeofday(&start, NULL);
      for (int i = 0; i < iterations; i++) { }
      gettimeofday(&end, NULL);
      printf("%ld\n", ((end.tv_sec * 1000000 + end.tv_usec) - (start.tv_sec * 1000000 + start.tv_usec)));
      return 0;
}
```

Time Performance Evaluation

RDTSC – Time stamp counter

```
typedef unsigned long long ticks;
static __inline__ ticks getticks(void) {
  unsigned a, d;
  asm("cpuid");
  asm volatile("rdtsc" : "=a" (a), "=d" (d));

  return (((ticks)a) | (((ticks)d) << 32));
}
int main() {
  ticks tick1, tick2;
  tick1 = getticks();
  sleep(10);
  tick2 = getticks();
  printf("CPU speed: %.3f GHz\n",
                   (tick2-tick1)/10.0/1000000000.0);
  return 0;
}
```

Producer-Consumer

The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue.

The producer's job is to generate a piece of data, put it into the buffer.

The consumer is consuming the data one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

Producer-Consumer with buffer size 10

```
#include <sys/time.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#define SIZE 10

pthread_mutex_t region_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t space_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t data_available = PTHREAD_COND_INITIALIZER;

int b[SIZE];
int size = 0;  /* number of full elements */
int front,rear=0;  /* queue */

main()  {
  pthread_t producer_thread;
  pthread_t consumer_thread;
  void *producer();
  void *consumer();


    pthread_create(&consumer_thread,NULL,consumer,NULL);
    pthread_create(&producer_thread,NULL,producer,NULL);
    pthread_join(consumer_thread,NULL); //join prod also

}
```

Producer-Consumer with buffer size 10

```
void add_buffer(int i){
  b[rear] = i;
  rear = (rear+1) % SIZE;
  size++;
}

int get_buffer(){
  int v;
  v = b[front];
  front= (front+1) % SIZE;
  size--;
  return v ;
}
```

Producer-Consumer with buffer size 10

```
void *producer()
{
int i = 0;
while (1) {
  pthread_mutex_lock(&region_mutex);
  if (size == SIZE) {
     pthread_cond_wait(&space_available,&region_mutex);
  }
  add_buffer(i);
  pthread_cond_signal(&data_available);
  pthread_mutex_unlock(&region_mutex);
  i = i + 1;
}
pthread_exit(NULL);
}
```

Producer-Consumer with buffer size 10

```
void *consumer()
{
int i,v;
for (i=0;i<100;i++) {
   pthread_mutex_lock(&region_mutex);
   if (size == 0) {
       pthread_cond_wait(&data_available,&region_mutex);
   }
   v = get_buffer();
   pthread_cond_signal(&space_available);
   pthread_mutex_unlock(&region_mutex);
   printf("got %d  ",v);
}
pthread_exit(NULL);
}
```

Linked List

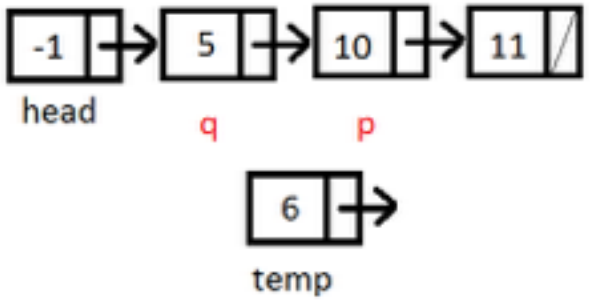
Example: sorted linked list of int

pointer head_p pointing to first element

functions: insert, delete, member

```
struct list_node_s{
        int data;
        struct list_node_s* next;
}
```

Ordered Linked Lists



```
void insert (int number, struct list_node_s** head_p) {
     struct list_node_s* curr_p=*head_p;
     struct list_node_s* prev_p=*NULL;
     struct list_node_s* temp_p;
    while (current_p !=  NULL  &&  current_p->data < number){
          prev_p=curr_p;
          curr_p = curr_p->next;
    }
    if(curr_p==NULL || curr_p->data> value) {
          temp_p=malloc(sizeof(struct list_node_s)); temp->data=number;
          temp_p->next=curr_p;
          if (pred_p==NULL) //new first node
                    *head_p=temp_p;
          else
                    pred_p->next=temp_p;
          return 1;
     } else return 0; //value already in list
}
```

Ordered Linked Lists

```
void delete (int number, struct list_node_s** head_p) {
       struct list_node_s* curr_p=*head_p;
       struct list_node_s* prev_p=*NULL;
       while (current_p !=  NULL  &&  current_p->data < number){
                prev_p=curr_p;
                curr_p = curr_p->next;
      }
      if(curr_p==NULL || curr_p->data == value) {
                if (pred_p==NULL) {//delete first node
                                 *head_p=curr_p->next;
                                 free(curr_p);{
                else{
                                 pred_p->next=curr_p->next;
                                 free(curr_p);}
                return 1;
       } else return 0; //value already in list
}
```

Ordered Linked Lists

```
void member (int number, struct list_node_s** head_p) {
     struct list_node_s* curr_p=*head_p;
     while (current_p !=  NULL  &&  current_p->data < number){
           curr_p = curr_p->next;
    }
    if(curr_p==NULL || curr_p->data> value)
           return 0;
     else
           return 1;
}
```

Read-Write Locks
Multithreaded Linked List Implementations

Define head_p in global memory

Problems:

multiple threads can read same memory location:

no problem so multiple threads can execute member

Insert and delete read but also write to memory:

so we may have problems executing any function while Insert, Delete is going on

One solution: lock the list any time that a thread tries to access it

so each of the three list function calls will be protected by a mutex

```
pthread_mutex_lock(&list_mutex);
member(value);
pthread_mutex_unlock(&list_mutex);
```

Note: This solution may be acceptable if most of the time we do insert and delete not member

Issues

- We're serializing access to the list.
- If the vast majority of our operations are calls to Member, we'll fail to exploit this opportunity for parallelism.
- On the other hand, if most of our operations are calls to Insert and Delete, then this may be the best solution since we'll need to serialize access to the list for most of the operations, and this solution will certainly be easy to implement.

Read-Write Locks
Multithreaded Linked List Implementations

Better solution (fain-grained locking) but complex to implement:

Instead of locking the entire list lock only individual nodes

```
struct list_node_s{
        int data;
        struct list_node_s* next;
        pthread_mutex_t mutex;
        }
```

Each time we try to access the node we first lock the mutex associated with the node

Implementation of Member with one mutex per list node (1)

```
int  Member(int value) {
   struct list_node_s* temp_p;

   pthread_mutex_lock(&head_p_mutex);
   temp_p = head_p;
   while (temp_p != NULL && temp_p->data < value) {
      if (temp_p->next != NULL)
         pthread_mutex_lock(&(temp_p->next->mutex));
      if (temp_p == head_p)
         pthread_mutex_unlock(&head_p_mutex);
      pthread_mutex_unlock(&(temp_p->mutex));
      temp_p = temp_p->next;
   }
```

Implementation of Member with one mutex per list node (2)

```
    if (temp_p == NULL || temp_p->data > value) {
       if (temp_p == head_p)
          pthread_mutex_unlock(&head_p_mutex);
       if (temp_p != NULL)
          pthread_mutex_unlock(&(temp_p->mutex));
       return 0;
    } else {
       if (temp_p == head_p)
          pthread_mutex_unlock(&head_p_mutex);
       pthread_mutex_unlock(&(temp_p->mutex));
       return 1;
    }
}   /* Member */
```

Pthreads Read-Write Locks

- Neither of our multi-threaded linked lists exploits the potential for simultaneous access to any node by threads that are executing Member.
- The first solution only allows one thread to access the entire list at any instant.
- The second only allows one thread to access any given node at any instant.

Read-Write Locks
Multithreaded Linked List Implementations

Better solution read-write locks:

mutex with two function (read-lock, write-lock)

Multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function

```
            pthread_rwlock_rdlock(&rwlock);
            member(value);
            pthread_rwlock_unlock(&rwlock);
…
            pthread_rwlock_wrlock(&rwlock);
            insert(value);
            pthread_rwlock_unlock(&rwlock);
```

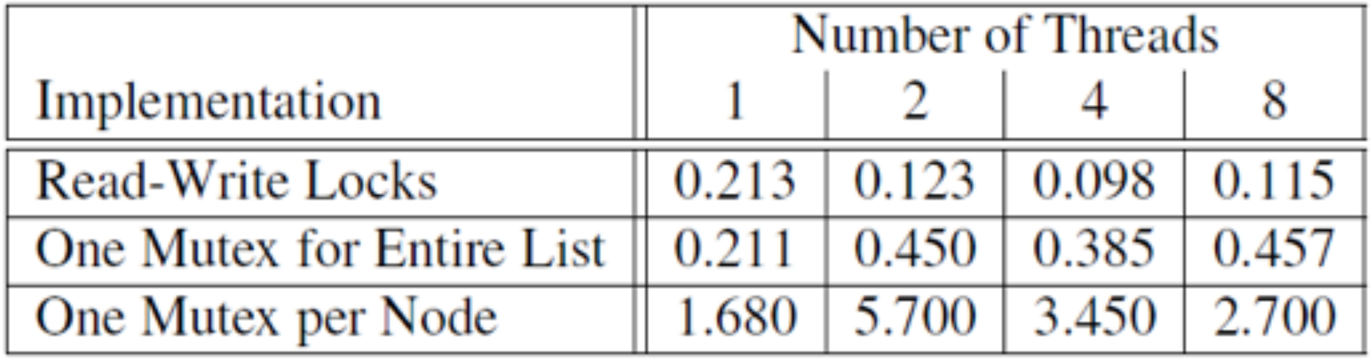
Pthreads Read-Write Locks

- A read-write lock is somewhat like a mutex except that it provides two lock functions.
- The first lock function locks the read-write lock for reading, while the second locks it for writing.

Pthreads Read-Write Locks

- So multiple threads can simultaneously obtain the lock by calling the read-lock function, while only one thread can obtain the lock by calling the write-lock function.
- If any thread owns the lock for writing, any threads that want to obtain the lock for reading or writing will block in their respective locking functions.

Linked List Performance



Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

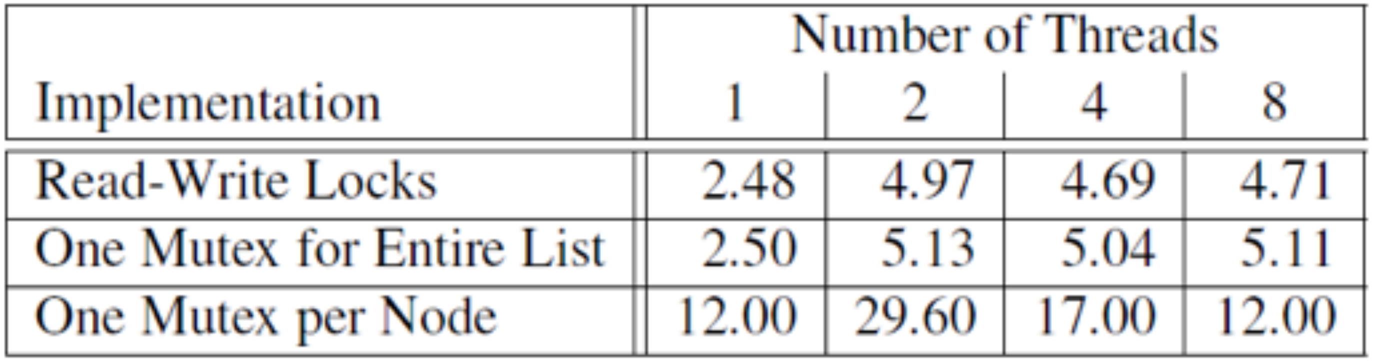
100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

Linked List Performance



	Number of Threads			
Implementation	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

100,000 ops/thread

80% Member

10% Insert

10% Delete