High Performance Computing - Session 3

Shared Memory Programming with pthreads

Jayanti Prasad

http://www.iucaa.ernet.in/ jayanti/

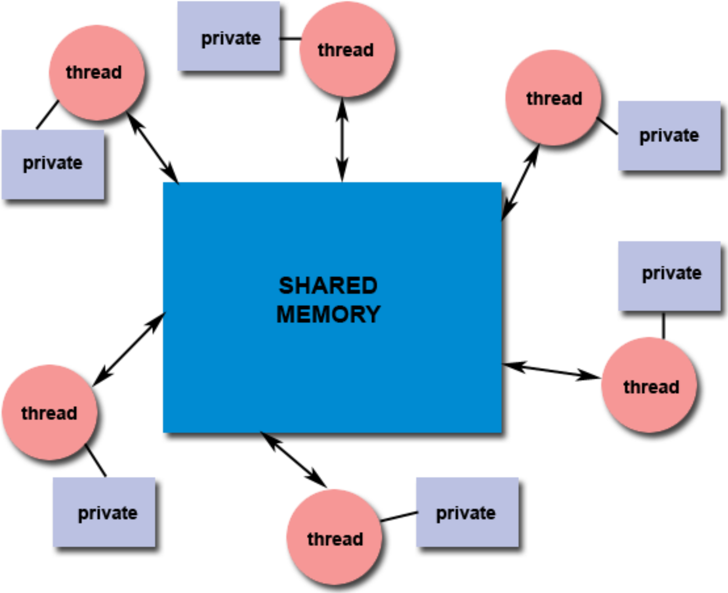
Inter-University Centre for Astronomy & Astrophysics
Pune, India (411007)

February 01, 2012

- Why we should bother about threads ?
  1. At present almost all desktops/laptops come with multi-core processors which can be fully exploited only by multi-thread programming.
  2. Multi-thread programming is much more efficient than multi-processors programing (MPI) due to faster communication (P-M : 30 GBPS, P-P: 5 GBPS).
  3. Multi-GPU systems and clusters with multi-core processors can be used more efficiently by incorporating multi-thread programming (hybrid programming).
- Why `pthreads` ?
  1. `pthreads` provide better control on threads.
  2. `pthreads` are more natural to a Linux system with C programming.
  3. `pthreads` are very light and versatile.

Plan of the Talk

- Introduction
  - Processes
  - Threads
- Posix threads
  - What is `pthreads`
  - Creating and joining `pthreads`
  - Race condition and Mutex locks
- Examples
- Problems


private
thread
thread
thread
private
private
SHARED
MEMORY
private
thread
thread
private
thread
private
thread
private

Linux Processes

- A process on a Linux system is an "object" through which the resources used by a program like memory, processor time and IO are managed and monitored.
- Processes are building blocks of any Linux system and can run in the "kernel" space or in the "user space".
- A process consists of an address space (a set of memory pages) and a set of data structures.
- The address space of a process contains the code and libraries that the process is executing,the processs variables, its stacks, and various extra information needed by the kernel while the process is running.
- Some of the common Linux command to monitor and manage processes are `ps -ef`, `top`, `strace`, `kill` etc.

Process

Program = Instruction + data
Process = Program in action

Processs address contains

- The current status of the process (sleeping, stopped, runnable, etc.)
- The execution priority of the process
- Information about the resources the process has used
- Information about the files and network ports the process has opened
- The processs signal mask (a record of which signals are blocked)
- The owner of the process

- In a Linux/Unix system processes execute asynchronously (independent from each other) even when there is only one processor.
- Processes are created using **fork** command on a Linux/Unix system.
- Processes can be killed by **kill -9** command on a Linux/Unix system.

Threads

- Threads are *light weight* sub-processes within a process (examples). For example, when we open a new tab in Internet browser we launch a new thread.
- Threads inherit many attributes of a the parent process (such as the processs address space).
- Multiple threads can execute concurrently (may be not at the same time) within a single process under a model called multi-threading.
- In a multi-thread process the processor can switch execution resources between threads, resulting in concurrent execution.
- Concurrency on a single processor (core) system indicates that more than one thread is making progress, but the threads are not actually running simultaneously.
- On a multi-core system each thread in the process can run concurrently on a separate core i.e., true parallelism.

What is pthreads ?

- pthreads (Portable Operating System Interface Threads) is an Application Programming Interface (API) or library, which can be used for shared memory/address space programming.
- pthreads library provides more than one hundred functions to manage threads, however, a very few (less than 10) are in general commonly used.
- When a thread is created, a new thread of control is added to the current process.
- Every process has at least one thread of control, in the program's main() routine.
- Each thread in the process runs simultaneously, and has access to the calling process's global data.
- In addition each thread has its own private attributes and call stack.

Managing pthreads

- To create a new thread, a running thread calls the pthread_create() function, and passes a pointer to a function for the new thread to run.
- One argument for the new thread's function can also be passed, along with thread attributes.
- The execution of a thread begins with the successful return from the pthread_create() function.
- The thread ends when the function that was called with the thread completes normally.
- A thread can also be terminated if the thread calls a pthread_exit() routine, or if any other thread calls pthread_cancel() to explicitly terminate that thread.
- When two or more concurrently running threads access a shared data item and the final result depends on the order of execution, we have a race condition.

Creating pthreads

```c
#include <pthread.h>

pthread_t thread[num_threads];

void *foo(void *);
void *arg;

int i;

for(i=0; i < num_threads; i++){
  pthread_create(&thread[i], NULL,foo,arg);
}
```

Compilation

gcc program.c -lpthread -lm

Joining threads

```
#include <pthread.h>

pthread_t thread[num_threads];

void *status;
/* waiting to join thread "tid" with status */

for(i=0; i < num_threads; i++){
    pthread_join(thread[i], &status);
}

/* waiting to join thread "tid" without status */

for(i=0; i < num_threads; i++){
  pthread_join(thread[i], NULL);
}
```

Race condition

Simultaneous updates lead to race conditions. Suppose two threads both execute `i++`. In machine code, this single statement becomes several operations:

Thread 1	Thread 2
`r1 = i`	`r2 = i`
`r1 += 1`	`r2 += 1`
`i = r1`	`i = r2`

If `i` starts at 0, what is the value of `i` after execution?

Race condition

Simultaneous updates lead to race conditions.

Thread 1	Thread 2
`r1 = i`	
`r1 += 1`	
`i = r1`	
	`r2 = i`
	`r2 += 1`
	`i = r2`

Final value of `i` is 2.

Race condition

Simultaneous updates lead to race conditions.

Thread 1	Thread 2
`r1 = i`	
`r1 += 1`	
	`r2 = i`
	`r2 += 1`
`i = r1`	
	`i = r2`

Final value of `i` is 1. The first update to `i` is lost.

Mutex Lock

```
#include <pthread.h>

pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);

counter++;

pthread_mutex_unlock(&mutex);
```

Program 1 : hello_world1_pthreads.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h> // must be included

//function which will be executed by every thread in parallel
void *tfunc(void *arg){ // one argument can be passed
  long    my_id = (long) arg;
  fprintf(stdout," Hello World from  %ld\n",my_id);
}

int main(int argc, char *argv[]){
  int num_threads;
  long i;
  pthread_t *t; // thread object
  if(argc < 2){
    fprintf(stderr,"./hello_world <# threads>\n");
    return(-1);
  }
  num_threads=atoi(argv[1]); // # of threads
  t = (pthread_t *)malloc(num_threads*sizeof(pthread_t));
  // create threads
  for(i=0; i < num_threads; i++)
    pthread_create(&t[i],NULL,tfunc,(void *)i);

  //join threads
  for(i=0; i < num_threads; i++)
    pthread_join(t[i],NULL) ;

  return(0);
}
```

Program 2 : hello_world2_pthreads.c

```
/* This structure can be used to send data to the function
  executed by every thread in  parallel */

typedef struct{
  int thread_id;
  char msg[100];
} thread_data;

// this function still done not return anything

void *tfunc(void *arg){
  thread_data *p = (thread_data *)arg; // p is a pointer to a structure
  int my_id = p->thread_id;
  fprintf(stdout," %s from  %d\n",p->msg,my_id);
}

int main(int argc, char *argv[]){
   pthread_t *t;
   thread_data *q; // this is an array

   q = (thread_data *)malloc(num_threads*sizeof(thread_data));

   for(i=0; i < num_threads; i++){
     q[i].thread_id = i;
     sprintf(q[i].msg,"Hello World");
     pthread_create(&t[i],NULL,tfunc,(void *)(q+i));
   }

  // join thread

}
```

Program 3 : return_pthreads.c

```c
void* child_thread( void * param ){
 long id, jd;
 id = (long )param;
 jd = id *id;
 return (void *)jd;
}

int main(int argc, char *argv[]){
   pthread_t *thread;
  long i, *return_value;

  thread = (pthread_t *)malloc(num_threads*sizeof(pthread_t));
  return_value=(long *)malloc(num_threads*sizeof(long ));

  for (i=0; i<num_threads; i++ )
    pthread_create(&thread[i],NULL,&child_thread,(void*)i );

  for (i=0; i<num_threads ; i++ )  {
    pthread_join(thread[i], (void**)&return_value[i] );
    printf( "input = %ld output=%ld  \n",i, return_value[i] );
  }
}
```

Program 4 : summation1_pthreads.c

```c
void* child_thread(void * param ){
  long id,i,p,y;
  id = (long )param;
  p = n/num_threads;
  y = 0;
  for(i=id*p; i <(id+1)*p; i++)
    y+=x[i];
  return (void *)y;
}

int main(int argc, char *argv[]){
  pthread_t *thread;
  long i,sum,y;

  thread = (pthread_t *)malloc(num_threads*sizeof(pthread_t));
  x=(long *)malloc(n*sizeof(long ));

  for(i=0; i < n; i++)
    x[i] = (long) i;

  for (i=0; i<num_threads; i++ )
    pthread_create(&thread[i],0,&child_thread,(void*)i );

  sum = 0;
  for (i=0; i<num_threads ; i++ )  {
    pthread_join(thread[i], (void**)&y );
    sum+=y;
  }
  printf("sum=%ld\n",sum);
}
```

Program 5 : summation2_pthreads.c

```
float *x;
/*---------------------------------------*/
typedef struct{
  int thread_id;  int chunk; float data;
} thread_data;
/*---------------------------------------*/
void *tfunc(void *arg){
  thread_data *p = (thread_data *)arg;
  long  id = p->thread_id;
  int i, np  = p->chunk;  float y = 0.0;
  for(i=id*np; i <(id+1)*np; i++)
    y+=x[i];

  p->data = y;
  // this is how the pointer to a structure is feed.
  return (void *)p;
}
/*---------------------------------------*/
int main(int argc, char *argv[]){
  pthread_t *t;  thread_data *q;
  np = (int) (num_points/num_threads);
      /*---- Create threads -----*/
  for(i=0; i < num_threads; i++){
    q[i].thread_id = i; q[i].chunk = np;
    pthread_create(&t[i],NULL,tfunc,(void *)(q+i));
  }
      /*------Join threads -------*/
  for(sum=0.0,i=0; i < num_threads; i++){
    pthread_join(t[i], (void**)&q[i] );
    sum+=q[i].data;
  }
 }
```

Program 6 : mutex1_pthreads.c

```c

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0

void *functionC(){
  pthread_mutex_lock(&mutex1 );
  counter++;
  printf("Counter value: %d\n",counter);
  pthread_mutex_unlock( &mutex1 );
}

int main(int argc, char *argv[]){


   for(l=0; l < num_threads; l++)
     pthread_create(&thread[l],NULL,&functionC,NULL);

   for(l=0; l < num_threads; l++)
     pthread_join(thread[l],NULL);

}
```

All programs

1. hello_world1_pthreads.c
2. hello_world2_pthreads.c
3. return_pthreads.c
4. summation1_pthreads.c
5. summation2_pthreads.c
6. mutex1_pthreads.c
7. mutex2_pthreads.c
8. create_fork.c
9. create_pthreads.c
10. stack_pthreads.c
11. scalar_prod_pthreads.c
12. compute_pi_pthreads.c
13. nbody_kernal_pthreads.c

Exercise

1. Compute the value of π by numerically integrating $1/(1+x^2)$ between limits $[0-1]$ in parallel using `pthreads` . You need to split the limit of integration and send the sub-limits to threads which return the answer for that limit.
2. Write a parallel program using `pthreads` for finding the number of prime number up to some number n and compare the performance with the `OpenMP` program for the same (you can modify the `OpenMP` program which was provided in `OpenMP` session).
3. Write a parallel program for matrix multiplication using `pthreads` .

Thank You !