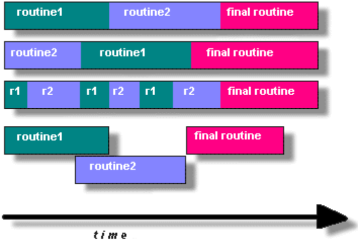
COMP20200 Unix Programming

Lecture 12

CS, University College Dublin, Ireland

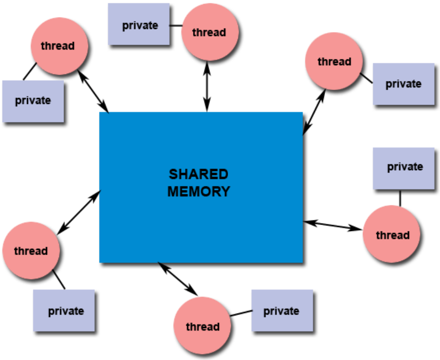
Designing Multi-threaded Programs

- Challenges affecting all parallel programs:
  - What type of parallel programming model to use?
  - Problem partitioning
  - Load balancing
  - Communications
  - Data dependencies
  - Synchronization and race conditions
  - Program complexity, Programmer effort/costs/time
- Program must be organized into discrete, independent tasks which can execute concurrently



Shared Memory Model:

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



Thread-safeness

- If an application can execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions it is thread safe.
- Eg. Application creates several threads, each makes a call to the same library routine.
    - If this library routine accesses global memory and does not employ synchronization constructs, then it is not thread-safe.

Creating and Terminating Threads

- `pthread_create (thread,attr,start_routine,arg)`
- `pthread_exit (status)`
- `pthread_cancel (thread)`
- `pthread_attr_init (attr)`
- `pthread_attr_destroy (attr)`

Creating Threads

- Initially main() program comprises a single, default thread.
- All other threads must be explicitly created by the programmer.
- `pthread_create` creates a new thread and makes it executable.
- Can be called any number of times from anywhere within your code.
- Compiling Threaded Programs with gcc: `gcc -pthread ...`

Skeleton example

```c
#include <pthread.h>
void* threadfunction(void* arg) {
  int* incoming = (int*)arg;
  (*incoming)++;
  return NULL;
}
int main(void) {
  pthread_t threadID;
  void* exitstatus;
  int value;
  value=42;
  pthread_create(&threadID,NULL,threadfunction,&value);
  pthread_join(threadID,&exitstatus);
  return 0;
}
```

Returning results from threads

```c
void *threadfunction(void *) {
  int *code;
  code=malloc(sizeof(int));
  // Set the value of (*code) -- code[0]

  return (void*)code;
}
```

Terminating Threads

- Several ways a thread may be terminated:
  - The thread returns normally from its starting routine. It's work is done.
  - The thread makes a call to the pthread_exit subroutine - whether its work is done or not.
  - The thread is canceled by another thread via the pthread_cancel routine – not recommended
  - The entire process is terminated due to making a call to either the exec() or exit()
  - If `main()` finishes first, without calling pthread_exit explicitly itself
- If main() finishes before the threads it spawned, all of the threads will terminate.
  - Explicitly calling pthread_exit() will block main() until all threads are finished.

Pthread Creation and Termination

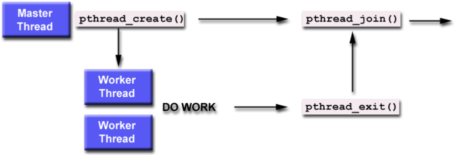
```
#include <pthread.h>  #define NUM_THREADS 5

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

int main (int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc; long t;
  for(t=0; t<NUM_THREADS; t++){
    printf("In main: creating thread %ld\n", t);
    //Not good -- open to race condition
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *)&t);
    if (rc){
      printf("ERROR; return code from pthread_create() is %d\n", rc
          );
      exit(-1);
    } }
  exit(0); }
```

Joining

- "Joining" is one way to accomplish synchronization between threads
- The pthread_join() subroutine blocks the calling thread until the specified threadid thread terminates.
- The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit().
- A joining thread can match one pthread_join() call. It is a logical error to attempt multiple joins on the same thread.
- Two other synchronization methods, mutexes and condition variables, will be discussed later.



Application of `pthreads`: Dot product

- Multiplication of 2 vectors (also known as scalar or inner product)
- $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_n]$
  $a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + ... + a_n b_n$
- In parallel, partition into chunks.
  - Eg. Vector size 1000 and 4 worker threads:
  - Main thread creates the 4 worker threads.
    - Thread 0 computes: $S_{local} = a_0 b_0 + a_1 b_1 + ... + a_{250} b_{250}$
    - ...
    - Thread 3 computes: $S_{local} = a_{750} b_{750} + a_{750} b_{750} + ... + a_{999} b_{999}$
  - Each worker:
    - calls a mutex lock
    - does $S_{global} = S_{global} + S_{local}$
    - calls a mutex unlock
  - Main thread joins all worker threads and outputs answer.

Application of pthreads: Dot product

- Code on next 4 slides
- Here is output:

```
Thread 1 did 250 to 499:  mysum=250.000000 global sum=250.000000
Thread 0 did   0 to 249:  mysum=250.000000 global sum=500.000000
Thread 3 did 750 to 999:  mysum=250.000000 global sum=750.000000
Thread 2 did 500 to 749:  mysum=250.000000 global sum=1000.000000
Sum =  1000.000000
```

Dot product 1 of 4

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

typedef struct {
  double      *a;
  double      *b;
  double     sum;
  int     veclen;
} DOTDATA;

#define NUMTHRDS 4
#define VECLEN 250
DOTDATA dotstr;
pthread_t callThd[NUMTHRDS];
pthread_mutex_t mutexsum;
```

Dot product 2 of 4

```
void *dotprod(void *arg){
  int i, start, end, len;
  long offset; double mysum, *x, *y;

  offset = *((long*)arg);

  len = dotstr.veclen;
  start = offset*len;
  end   = start + len;
  x = dotstr.a;
  y = dotstr.b;

  mysum = 0;
  for (i=start; i<end ; i++) {
    mysum += (x[i] * y[i]);
  }

  pthread_mutex_lock (&mutexsum);
  dotstr.sum += mysum;
  printf("Thread %ld did %d to %d:  mysum=%f global sum=%f\n",
      offset,start,end-1,mysum,dotstr.sum);
  pthread_mutex_unlock (&mutexsum);

  pthread_exit(NULL); }
```

Dot product 3 of 4

```
int main(int argc, char *argv[]){
  long i; double *a, *b;

  a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
  b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

  for (i = 0; i < VECLEN*NUMTHRDS; i++) {
    a[i] = 1;
    b[i] = a[i];
  }

  dotstr.veclen = VECLEN;
  dotstr.a = a;
  dotstr.b = b;
  dotstr.sum = 0;

  pthread_mutex_init(&mutexsum, NULL);

  pthread_attr_t attr;
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
```

Dot product 4 of 4

```
  //open to race condition
  //for(i=0;i<NUMTHRDS;i++) {
  //  pthread_create(&callThd[i], &attr, dotprod, (void *)&i);
  //}
  long *off;
  off=calloc(NUMTHRDS, sizeof(long));
  for(i=0;i<NUMTHRDS;i++,off[i]=i) { //safe
    pthread_create(&callThd[i], &attr, dotprod, off+i);
  }

  pthread_attr_destroy(&attr);

  for(i=0;i<NUMTHRDS;i++)
    pthread_join(callThd[i], NULL);

  printf ("Sum =  %f \n", dotstr.sum);
  free (a); free (b);
  pthread_mutex_destroy(&mutexsum);
  pthread_exit(NULL);
}
```