CME 213

SPRING 2012-2013

Eric Darve

Stanford University

PTHREADS

PROGRAMMING USING THREADS

- **This is the most basic approach for programming in parallel.**
- **Pthreads: POSIX threads. This is a standard to implement threads on UNIX systems. It is based on the C programming language.**
- **The other approach is based on OpenMP.**

THE BASICS

- **Include the header file:**

```
<pthread.h>
```

- **Compile using:**

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

LET'S DIVE IN

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

void *PrintHello(void *threadid){
   long 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);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
   }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Stanford University

OUTPUT

```
In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
Hello World! It's me, thread #2!
In main: creating thread 3
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

THREAD CREATION

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*routine)(void*),
    void *arg)
```

- **thread** thread identifier
- **routine** function that will be executed by the generated thread
- **arg** pointer to the argument value with which the thread function **routine()** will be executed
- **attr** use **NULL** for the time being

THREAD TERMINATION
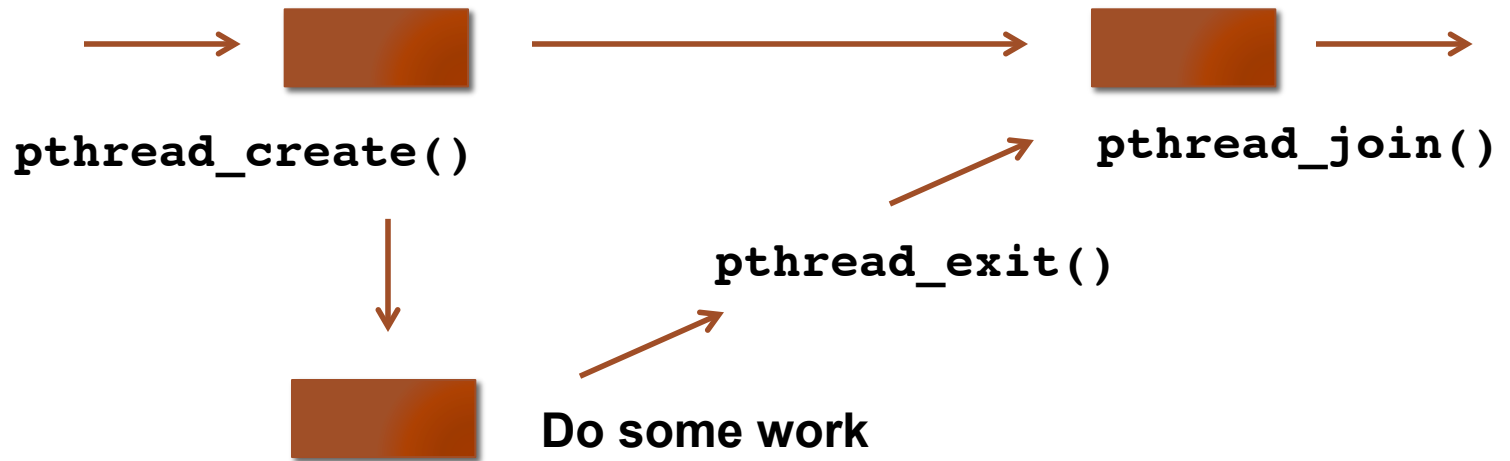
A thread terminates when:

1. **Thread reaches the end of its thread function, i.e., returns.**

2. **Thread calls**

```
void pthread_exit(void *valuep)
```

Note:

- **Upon termination, a thread releases its runtime stack.**
- **Therefore the pointer should point to: 1) a global variable, or 2) a dynamically allocated variable.**

Stanford University

JOINING THREADS

pthread_create()

pthread_join()

pthread_exit()

Do some work
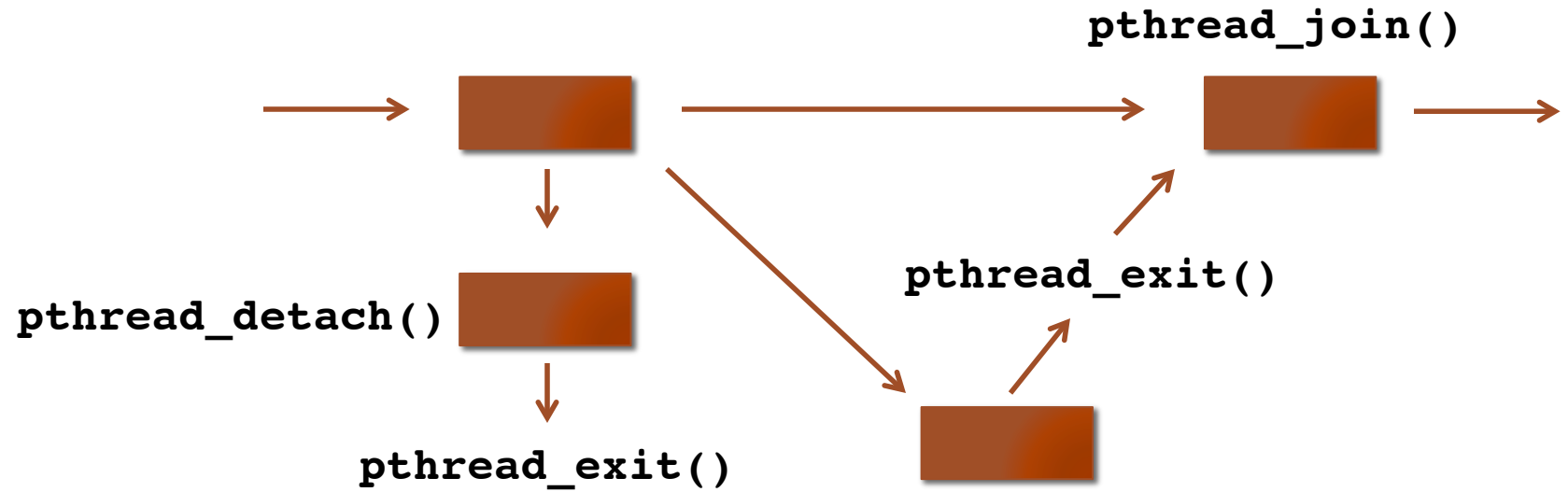
`int pthread_join( pthread_t thread, void **valuep )`

- **Calling thread waits for thread to terminate.**
- **`pthread_join` is used to synchronize threads.**
- **`valuep` memory address where the return value of thread is stored.**

DETACHING A THREAD

pthread_join()

pthread_detach()

pthread_exit()

pthread_exit()

Stanford University

DETACHING THREADS

- **A thread that terminates needs to hold some internal data structure until `pthread_join()` is called.**
- **What happens if `pthread_join()` is never called?**
  - Basically, synchronization is not needed. We launched a thread; it does its work but we don't need to know when it terminates.
- **In that case, the internal data structure is never released. This can be a problem in programs that create and terminate a lot of threads.**

Solution:

```
int pthread_detach(pthread_t thread)
```

- **This function notifies the runtime system that the internal data structure of this thread can be freed as soon as the thread terminates.**
- **`pthread_join()` returns an error in that case.**

Stanford University

EXAMPLE: MATRIX MATRIX PRODUCT

MC = MA * MB

```c
#include <pthread.h>

typedef struct {
  int size, row, column;
  double (*MA)[8], (*MB)[8], (*MC)[8];
} matrix_type_t;

void *thread_mult (void *w) {
  matrix_type_t *work = (matrix_type_t *) w;
  int i, row = work->row, column = work->column;
  work -> MC[row][column] = 0;
  for (i=0; i < work->size; i++)
    work->MC[row][column] += work->MA[row][i] * work->MB[i][column];
  return NULL;
}

int main() {
  int row, column, size = 8, i;
  double MA[8][8], MB[8][8], MC[8][8];
  matrix_type_t *work;
  pthread_t thread[8*8];
  for (row=0; row<size; row++)
    for (column=0; column<size; column++) {
      work = (matrix_type_t *) malloc (sizeof (matrix_type_t));
      work->size = size;
      work->row = row;
      work->column = column;
      work->MA = MA; work->MB = MB; work->MC = MC;
      pthread_create (&(thread[column + row*8]), NULL,
                       thread_mult, (void *) work);
    }

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

ford University

THREAD COORDINATION

MUTEXES

THE RISKS OF MULTI-THREADED PROGRAMMING

- Let us assume that a well-known bank company has asked you to implement a multi-threaded code to perform bank transactions.

- You start with the modest goal of allowing deposits.

- Clients deposit money and the amount gets credited to their accounts.

- As a result of having multiple threads running concurrently the following can happen:

Stanford University

A PARALLEL BANK DEPOSIT

Thread 0	Thread 1	Balance
Client requests a deposit	Client requests a deposit	$1000
Check current balance = $1000		
	Check current balance = $1000	
Ask for deposit amount = $100	Ask for deposit amount = $300	
	Compute new balance = $1300	
Compute new balance = $1100	Write new balance to account	$1300
Write new balance to account		$1100

Race condition

- **Although the correct balance should be $1,400, it is $1,100.**
- **The problem is that many operations "take time" and can be "interrupted" by other threads attempting to modify the same data.**
- **This is called a race condition: the final result depends on the precise order in which the instructions are executed.**
- **Unless thread 0 completes its update before thread 1 (or vice versa) we get an incorrect result.**
- **This issue is addressed using mutexes (mutex): mutual exclusion.**
- **They ensure that certain common pieces of data are accessed and modified by a single thread.**

Stanford University

MUTEX

- **A mutex can only be in two states: locked or unlocked.**
- **Once a thread locks a mutex:**
  - Other threads attempting to lock the same mutex are blocked
  - Only the thread that initially locked the mutex has the ability to unlock it.
- **This allows to protect regions of code.**

Stanford University

TYPICAL USAGE

A typical sequence in the use of a mutex is as follows:
- **Create and initialize a mutex variable**
- **Several threads attempt to lock the mutex**
- **Only one succeeds and that thread owns the mutex**
- **The owner thread performs some set of actions**
- **The owner unlocks the mutex**
- **Another thread acquires the mutex and repeats the process**

Performing a dot product in parallel

- This is the simplest example that uses mutexes.
- All threads calculate a portion of the dot product using local variables.
- A final reduction (+) is made to a shared variable.
- This update is protected using a mutex.

Stanford University

HEAD OF FILE

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

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

/* Define globally accessible variables and a mutex */
DOTDATA dotstr;
pthread_mutex_t mutexsum;
```

Stanford University

```c
int main (int argc, char *argv[]) {
    int numthrds = 4;
    int veclen = 100000;
    long i;
    double *a, *b;
    pthread_t *thread;

    /* Assign storage and initialize values */
    thread = (pthread_t*) malloc(numthrds*sizeof(pthread_t));

    a = (double*) malloc (numthrds*veclen*sizeof(double));
    b = (double*) malloc (numthrds*veclen*sizeof(double));
    …

    dotstr.veclen = veclen;
    dotstr.a = a; dotstr.b = b;
    dotstr.sum = 0.0;

    pthread_mutex_init(&mutexsum, NULL);

    /* Create threads to perform the dotproduct */
    for (i=0; i<numthrds; i++)
        pthread_create(&thread[i], NULL, dotprod, (void *)i);

    /* Wait on the other threads */
    for (i=0; i<numthrds; i++) pthread_join(thread[i], NULL);

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

Stanford University

```c
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];

    /*
     Lock a mutex prior to updating the value in the shared
     structure, and unlock it upon updating.
     */
    pthread_mutex_lock(&mutexsum);
    dotstr.sum += mysum;
    printf("Thread %d did %8d to %8d: mysum = %7.3f global sum = %7.3f\n",
            (int)offset,start,end,mysum,dotstr.sum);
    pthread_mutex_unlock(&mutexsum);

    pthread_exit(NULL);
}
```

Stanford University

OUTPUT

```
Thread 2 did    200000 to    300000: mysum =    0.405 global sum =    0.405
Thread 1 did    100000 to    200000: mysum =    0.693 global sum =    1.099
Thread 3 did    300000 to    400000: mysum =    0.288 global sum =    1.386
Thread 0 did         0 to    100000: mysum =   12.090 global sum =   13.476
Sum =   13.476437
```

Stanford University

```
int pthread_mutex init (pthread_mutex t *mutex,
                   const pthread_mutexattr_t *attr)
```
Initialization of mutex; choose `NULL` for `attr`.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex)
```
Destruction of mutex

```
int pthread_mutex_lock (pthread_mutex_t *mutex)
```
Locks a mutex; blocks if another thread has locked this mutex and owns it.

```
int pthread_mutex_unlock (pthread_mutex_t *mutex)
```
Unlocks mutex; after unlocking, other threads get a chance to lock the mutex.

Stanford University