Threads & Posix Threads (pthreads)

Z. Cihan TAYŞİ

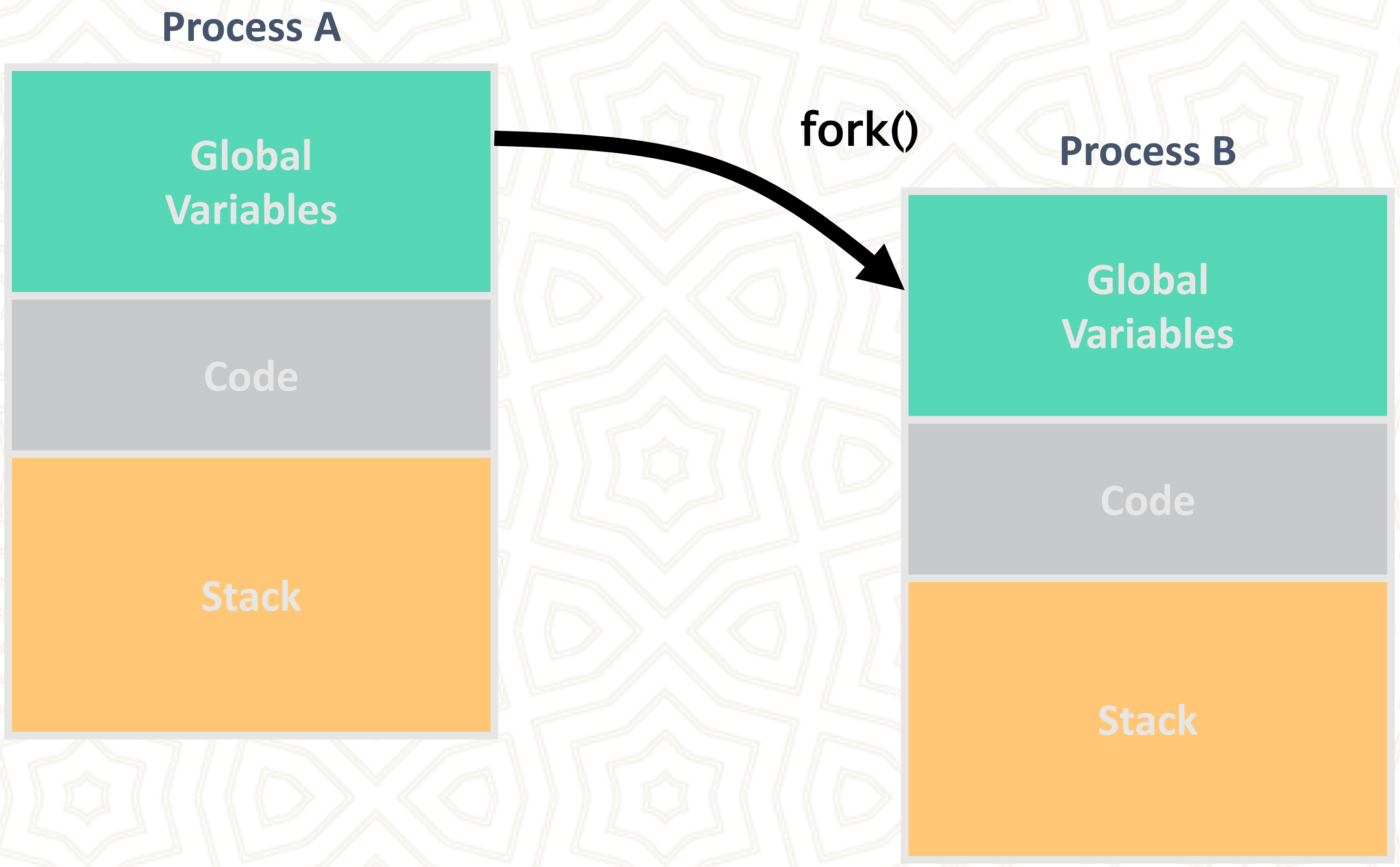
Outline

- Threads vs. Processes
- Multiple threads
- Thread-specific resources
- Posix threads
- Detached vs attached

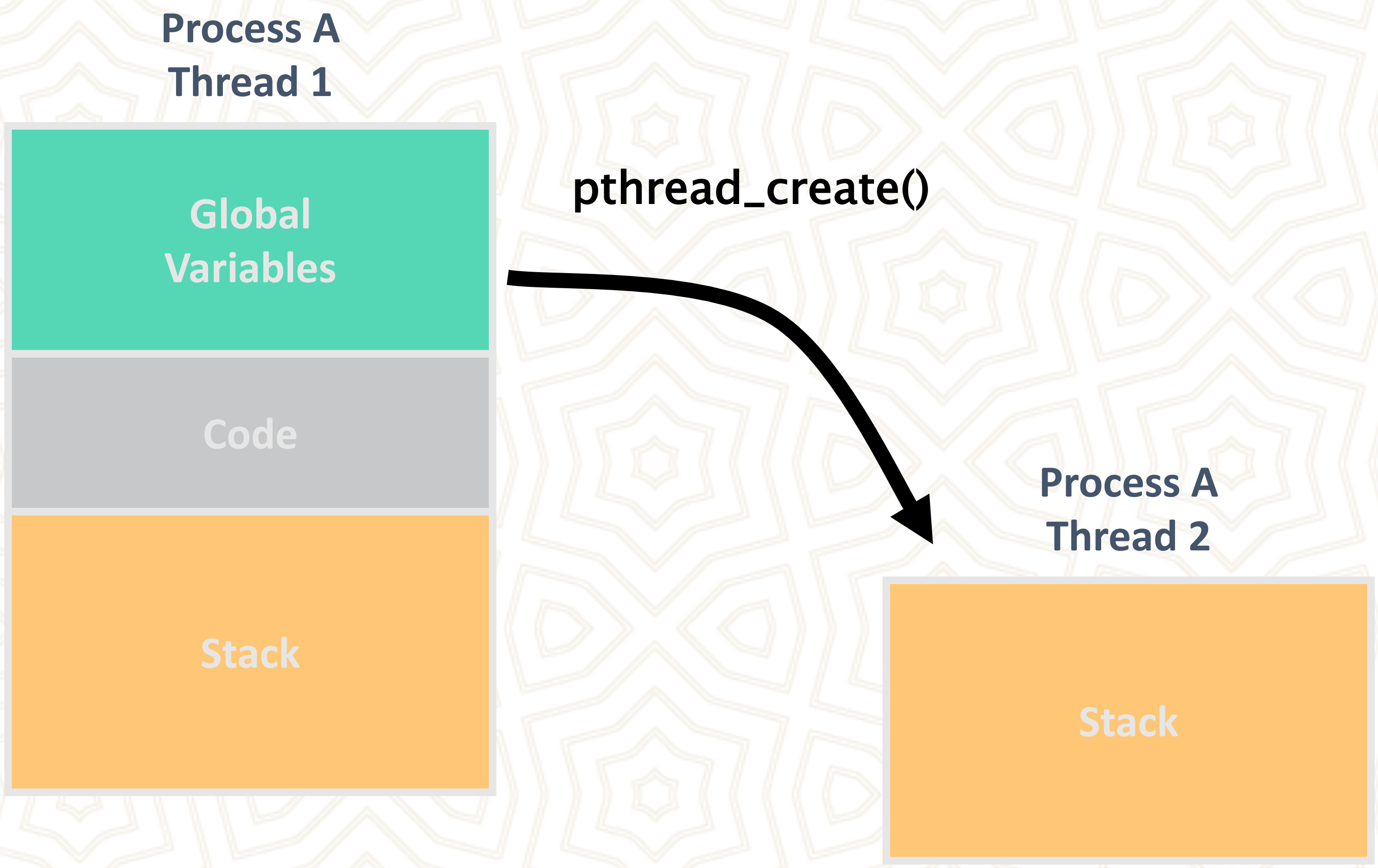
Threads vs. Processes

- Creation of a new process using fork is expensive
  - time & memory
- A thread does not require lots of memory or startup time.
  - sometimes called a lightweight process

The fork() system call



pthread_create()



Multiple Threads

- Each process can include many threads.

- All threads of a process share:
  - memory (program code and global data)
  - open file/socket descriptors
  - signal handlers and signal dispositions
  - working environment (current directory, user ID, etc.)

Thread-specific Resources

- Each thread has its own
  - Thread ID
  - Stack, Registers, Program Counter
  - **errno** (if not - **errno** would be useless!)

- Threads within the same process can communicate using shared memory.
  - ***Must be done carefully***

Posix Threads

- We will focus on Posix Threads - most widely supported threads programming API.

- you need to link with “**-lpthread**”

- On many systems this also forces the compiler to link in re-entrant libraries (instead of plain vanilla C libraries).

Thread Creation

- pthread_create(

```
        pthread_t *tid,
        const pthread_attr_t *attr,
        void *(*func)(void *),
        void *arg);
```

- **func** is the function to be called.
  - when func() returns the thread is terminated.

`pthread_create()`

- The return value is 0 for OK.
  - **positive error number on error.**

- Does not set errno !!!

- Thread ID is returned in **tid**

Thread IDs

- Each thread has a unique ID, a thread can find out it's ID by calling **pthread_self()**.

- Thread IDs are of type pthread_t which is usually an unsigned int. When debugging, it's often useful to do something like this:
  - **printf("Thread %u:\n",pthread_self());**

Thread Arguments

- When **func()** is called the value **arg** specified in the call to **pthread_create()** is passed as a parameter.

- **func** can have **only 1** parameter, and it can't be larger than the size of a **void ***.

Thread Arguments (cont.)

- Complex parameters can be passed by creating a structure and passing the address of the structure.

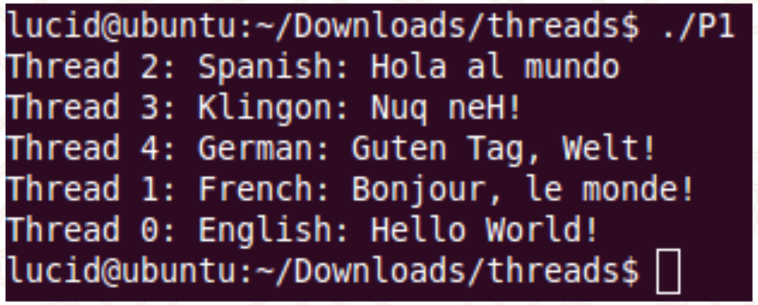
- The structure can't be a local variable (of the function calling **pthread_create**)!!
  - threads have different stacks!

A Simple pthread Example

pthread1.c

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

  pthread_t threads[NUM_THREADS];     // Thread identifiers
  int i, rc, *taskid[NUM_THREADS];    // Id numbers for each thread

  // Initialize the salutations for each thread
  messages[0] = "English: Hello World!";
  messages[1] = "French: Bonjour, le monde!";
  messages[2] = "Spanish: Hola al mundo";
  messages[3] = "Klingon: Nuq neH!";
  messages[4] = "German: Guten Tag, Welt!";

  for(i=0; i<NUM_THREADS; i++) {
    // Allocte an array for arguments to the threads
    taskid[i] = (int *) malloc(sizeof(int));
    *taskid[i] = i;
    // Create a thread with its argument in taskid[i]
    rc = pthread_create(&threads[i], NULL, PrintHello, (void *) taskid[i]);
    if (rc) { // Check for errors
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(0);
}
```

```c
void *PrintHello(void *threadid) {
    int *myarg;
    sleep(1);                        // Sleep for a second
    myarg = (int *) threadid;        // Get own id from the argument
    printf("Thread %d: %s\n", *myarg, messages[*myarg]);
    pthread_exit(NULL);
}
```



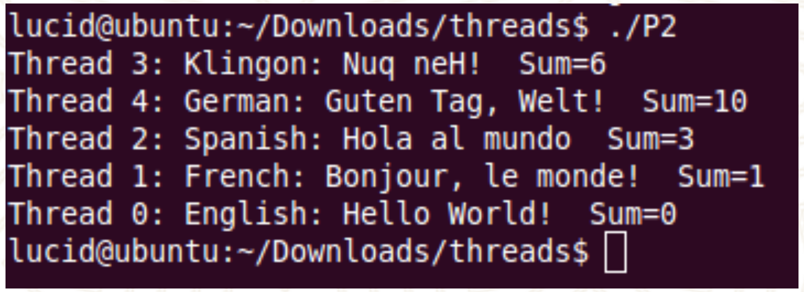
```
lucid@ubuntu:~/Downloads/threads$ ./P1
Thread 2: Spanish: Hola al mundo
Thread 3: Klingon: Nuq neH!
Thread 4: German: Guten Tag, Welt!
Thread 1: French: Bonjour, le monde!
Thread 0: English: Hello World!
lucid@ubuntu:~/Downloads/threads$
```

A Not So Simple pthread Example

pthread2.c

```c
int main(int argc, char *argv[]) {
  pthread_t threads[NUM_THREADS];
  int rc, i, sum;

  sum=0;
  messages[0] = "English: Hello World!";
  messages[1] = "French: Bonjour, le monde!";
  messages[2] = "Spanish: Hola al mundo";
  messages[3] = "Klingon: Nuq neH!";
  messages[4] = "German: Guten Tag, Welt!";

  for(i=0; i<NUM_THREADS; i++) {
    // Initialize arguments to a thread
    sum = sum + i;
    thread_data_array[i].thread_id = i;
    thread_data_array[i].sum = sum;
    thread_data_array[i].message = messages[i];
    // Create a thread
    rc = pthread_create(&threads[i], NULL, PrintHello, &thread_data_array[i]);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

```c
void *PrintHello(void *threadarg) {
  int myid, sum;
  char *hello_msg;
  struct thread_data *my_data;

  sleep(1);
  my_data = (struct thread_data *) threadarg;
  myid = my_data->thread_id;
  sum = my_data->sum;
  hello_msg = my_data->message;
  printf("Thread %d: %s  Sum=%d\n", myid, hello_msg, sum);
  pthread_exit(NULL);
}
```



```
lucid@ubuntu:~/Downloads/threads$ ./P2
Thread 3: Klingon: Nuq neH!  Sum=6
Thread 4: German: Guten Tag, Welt!  Sum=10
Thread 2: Spanish: Hola al mundo  Sum=3
Thread 1: French: Bonjour, le monde!  Sum=1
Thread 0: English: Hello World!  Sum=0
lucid@ubuntu:~/Downloads/threads$ 
```

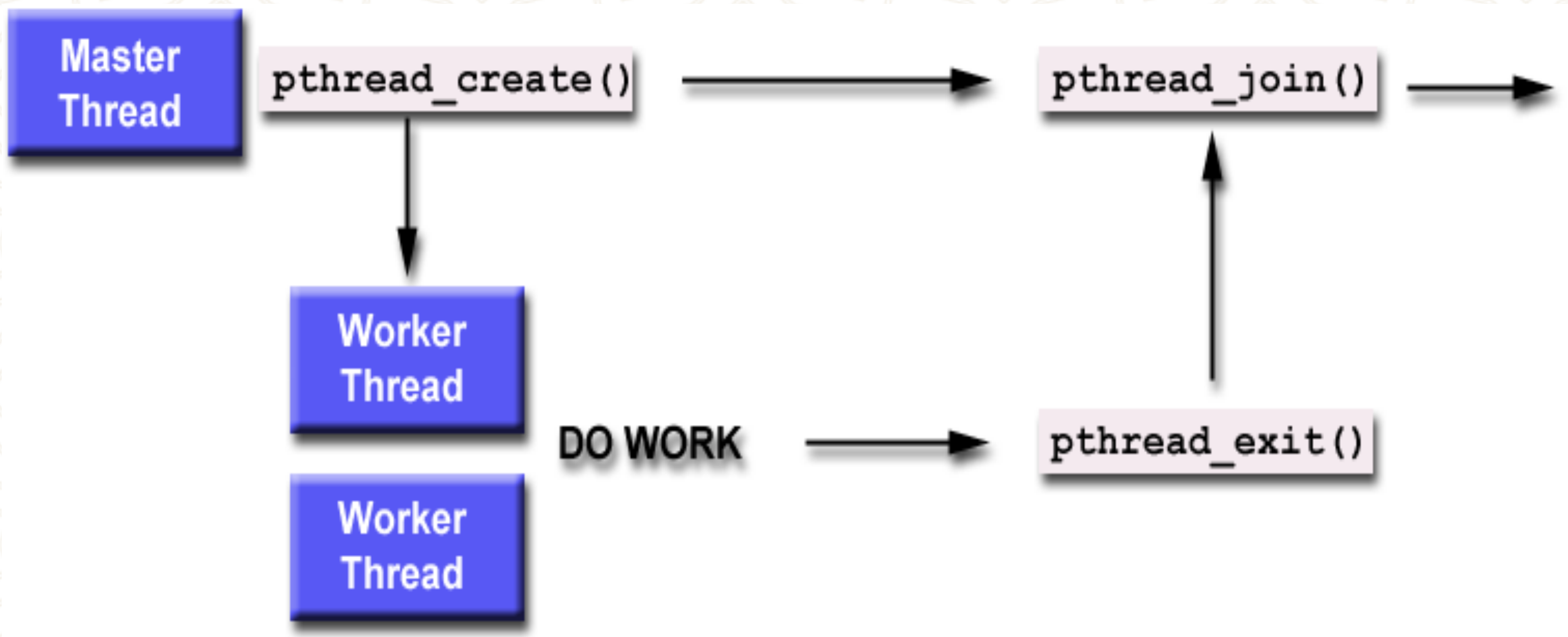
Thread Lifespan

- Once a thread is created, it starts executing the function **func()** specified in the call to **pthread_create()**.
- If **func()** returns, the thread is terminated.
- A thread can also be terminated by calling **pthread_exit().**
- If **main()** returns or any thread calls exit() all threads are terminated.

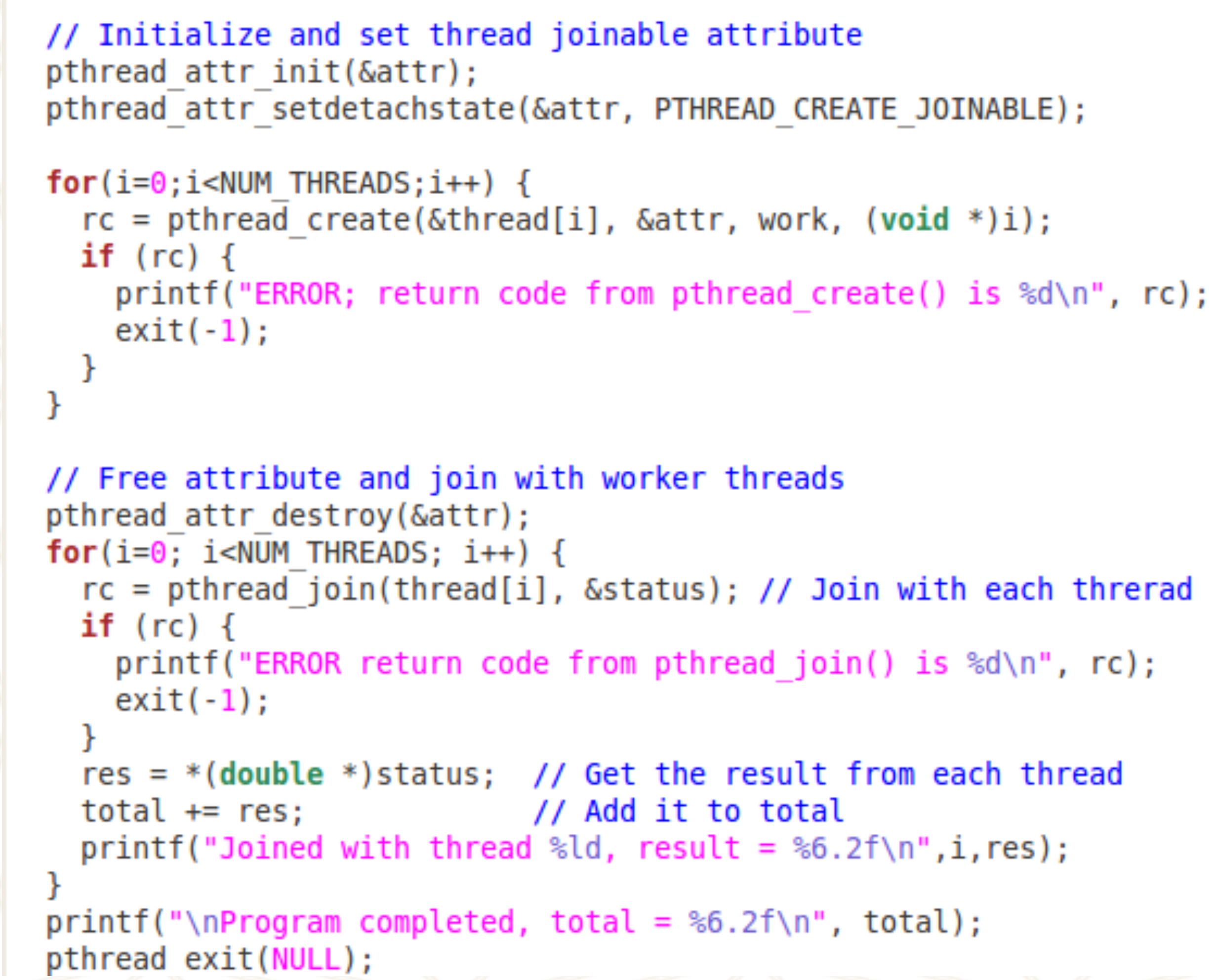
Detached vs. Joinable

- Each thread can be either **joinable** or **detached**.
- **Joinable:** on thread termination the thread ID and exit status are saved by the OS.
- **Detached:** on termination all thread resources are released by the OS. A detached thread cannot be joined.

Detached vs. Joinable (Contd.)

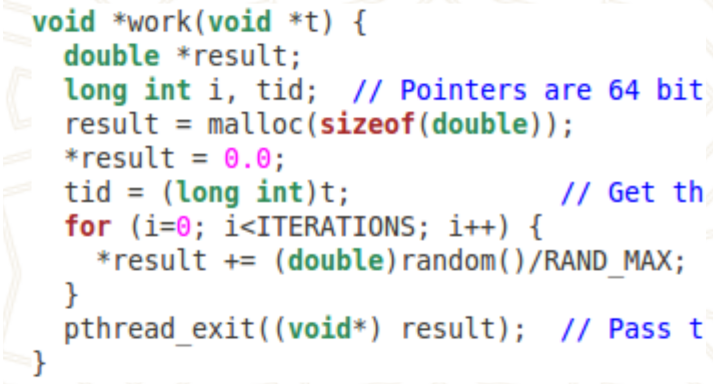


Joinable Thread Example

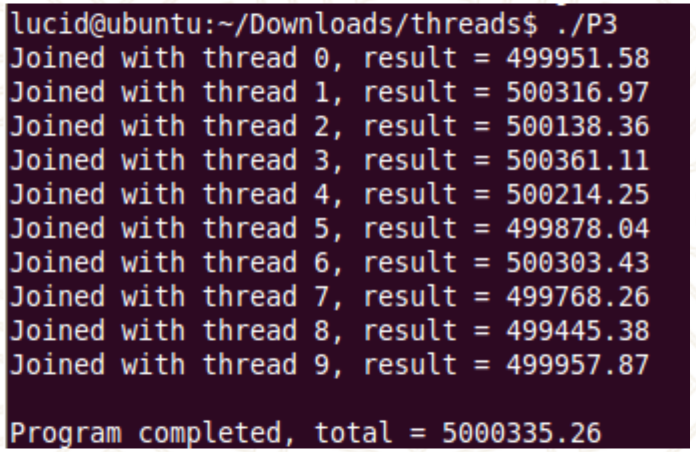


```
// Initialize and set thread joinable attribute
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUM_THREADS;i++) {
  rc = pthread_create(&thread[i], &attr, work, (void *)i);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
  }
}

// Free attribute and join with worker threads
pthread_attr_destroy(&attr);
for(i=0; i<NUM_THREADS; i++) {
  rc = pthread_join(thread[i], &status); // Join with each threrad
  if (rc) {
    printf("ERROR return code from pthread_join() is %d\n", rc);
    exit(-1);
  }
  res = *(double *)status;  // Get the result from each thread
  total += res;             // Add it to total
  printf("Joined with thread %ld, result = %6.2f\n",i,res);
}
printf("\nProgram completed, total = %6.2f\n", total);
pthread_exit(NULL);
```

pthread3.c

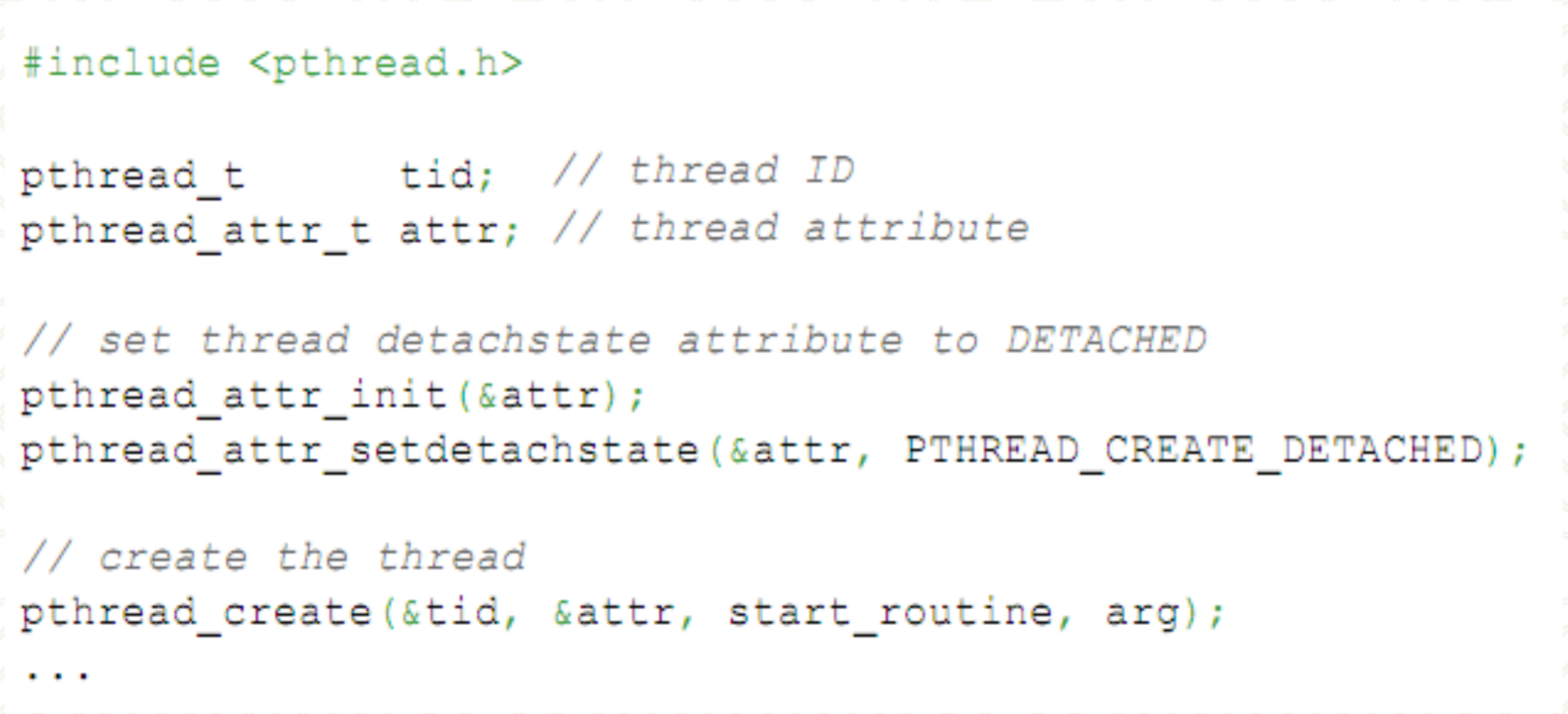


```
void *work(void *t) {
  double *result;
  long int i, tid;  // Pointers are 64 bit
  result = malloc(sizeof(double));
  *result = 0.0;
  tid = (long int)t;             // Get th
  for (i=0; i<ITERATIONS; i++) {
    *result += (double)random()/RAND_MAX;
  }
  pthread_exit((void*) result);  // Pass t
}
```



```
lucid@ubuntu:~/Downloads/threads$ ./P3
Joined with thread 0, result = 499951.58
Joined with thread 1, result = 500316.97
Joined with thread 2, result = 500138.36
Joined with thread 3, result = 500361.11
Joined with thread 4, result = 500214.25
Joined with thread 5, result = 499878.04
Joined with thread 6, result = 500303.43
Joined with thread 7, result = 499768.26
Joined with thread 8, result = 499445.38
Joined with thread 9, result = 499957.87

Program completed, total = 5000335.26
```

Howto detach

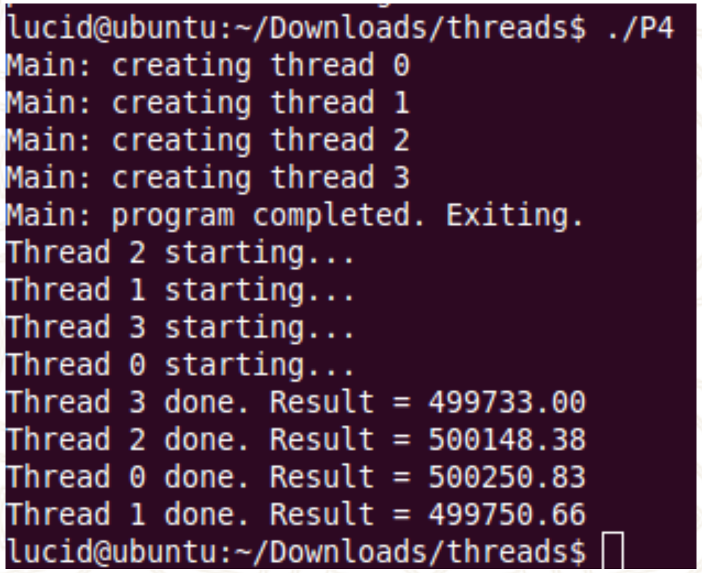


```
#include <pthread.h>

pthread_t      tid;  // thread ID
pthread_attr_t attr; // thread attribute

// set thread detachstate attribute to DETACHED
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

// create the thread
pthread_create(&tid, &attr, start_routine, arg);
...
```

Detached Thread Example

```c
int main(int argc, char *argv[]) {
  pthread_t thread[NUM_THREADS];
  pthread_attr_t attr;
  int rc;
  long int t;

  // Initialize and set thread detached attribute
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

  for(t=0;t<NUM_THREADS;t++) {
    printf("Main: creating thread %ld\n", t);
    rc = pthread_create(&thread[t], &attr, BusyWork, (void *)t);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }

  // We're done with the attribute object, so we can destroy it
  pthread_attr_destroy(&attr);

  printf("Main: program completed. Exiting.\n");
  pthread_exit(NULL);
}
```

```c
void *BusyWork(void *t) {
  long int i, tid;
  double maxval, result=0.0;
  tid = (long int)t;
  printf("Thread %ld starting...\n",tid);
  for (i=0; i<ITERATIONS; i++) {
    result += (double)random()/RAND_MAX;
  }
  printf("Thread %ld done. Result = %6.2f\n",tid, result);
}
```



```
lucid@ubuntu:~/Downloads/threads$ ./P4
Main: creating thread 0
Main: creating thread 1
Main: creating thread 2
Main: creating thread 3
Main: program completed. Exiting.
Thread 2 starting...
Thread 1 starting...
Thread 3 starting...
Thread 0 starting...
Thread 3 done. Result = 499733.00
Thread 2 done. Result = 500148.38
Thread 0 done. Result = 500250.83
Thread 1 done. Result = 499750.66
lucid@ubuntu:~/Downloads/threads$
```

Shared Global Variables

- Possible problems
  - Global variables

- Avoiding problems

- Synchronization Methods
  - Mutexes
  - Condition variables

Possible problems

- **Sharing global variables is dangerous**
    - two threads may attempt to modify the same variable at the same time.

- Just because you don't see a problem when running your code doesn't mean it can't and won't happen!!!!

Avoiding problems

- pthreads includes support for **Mutual Exclusion** primitives that can be used to protect against this problem.

- The general idea is to **lock** something before accessing global variables and to unlock as soon as you are done.

- **Shared socket descriptors** should be treated as **global variables**!!!

Mutexes

- A global variable of type pthread_mutex_t is required:

- **pthread_mutex_t** counter_mtx **=**
  PTHREAD_MUTEX_INITIALIZER;

- Initialization to PTHREAD_MUTEX_INITIALIZER

is required for a static variable!

Lock & Unlock

- To lock use:
  - **pthread_mutex_lock(pthread_mutex_t &);**
- To unlock use:
  - **pthread_mutex_unlock(pthread_mutex_t &);**
- Both functions are blocking!

Mutex Example

- A simple program to compute dot product of two vectors
  - a and b
- Main data is globally available to all threads
  - **dotdata** structure
    - dotdata.a
    - dotdata.b
  - Each threads works on a different part of it
- The sum is stored **dotdata.sum**
  - protected by a **mutex** !!

Condition Variables

- **pthreads** support **condition variables**, which allow one thread to wait (sleep) for an event generated by any other thread.

- This allows us to avoid the **busy waiting** problem.

- **pthread_cond_t** foo = **PTHREAD_COND_INITIALIZER;**

Condition Variables (cont.)

- A condition variable is always used with mutex.

- **pthread_cond_wait(pthread_cond_t *cptr,**
  pthread_mutex_t *mptr);

- **pthread_cond_signal(pthread_cond_t *cptr);**

don't let the word signal confuse you -
this has nothing to do with Unix signals

Condition Variable Example

- The main thread creates three threads.
  - Two of those threads increment a "count" variable,
  - The third thread watches the value of "count".

- When "**count**" reaches a predefined limit,
  - the waiting thread is signaled by one of the incrementing threads.

Summary

- Threads are awesome, but dangerous. You have to pay attention to details or it's easy to end up with code that is incorrect (doesn't always work, or hangs in deadlock).
- Posix threads provides support for mutual exclusion, condition variables and thread-specific data.
- IHOP serves breakfast 24 hours a day!

References

- http://ube.ege.edu.tr/~cinsdiki/UBI511.html
- man pages