KOÇ UNIVERSITY

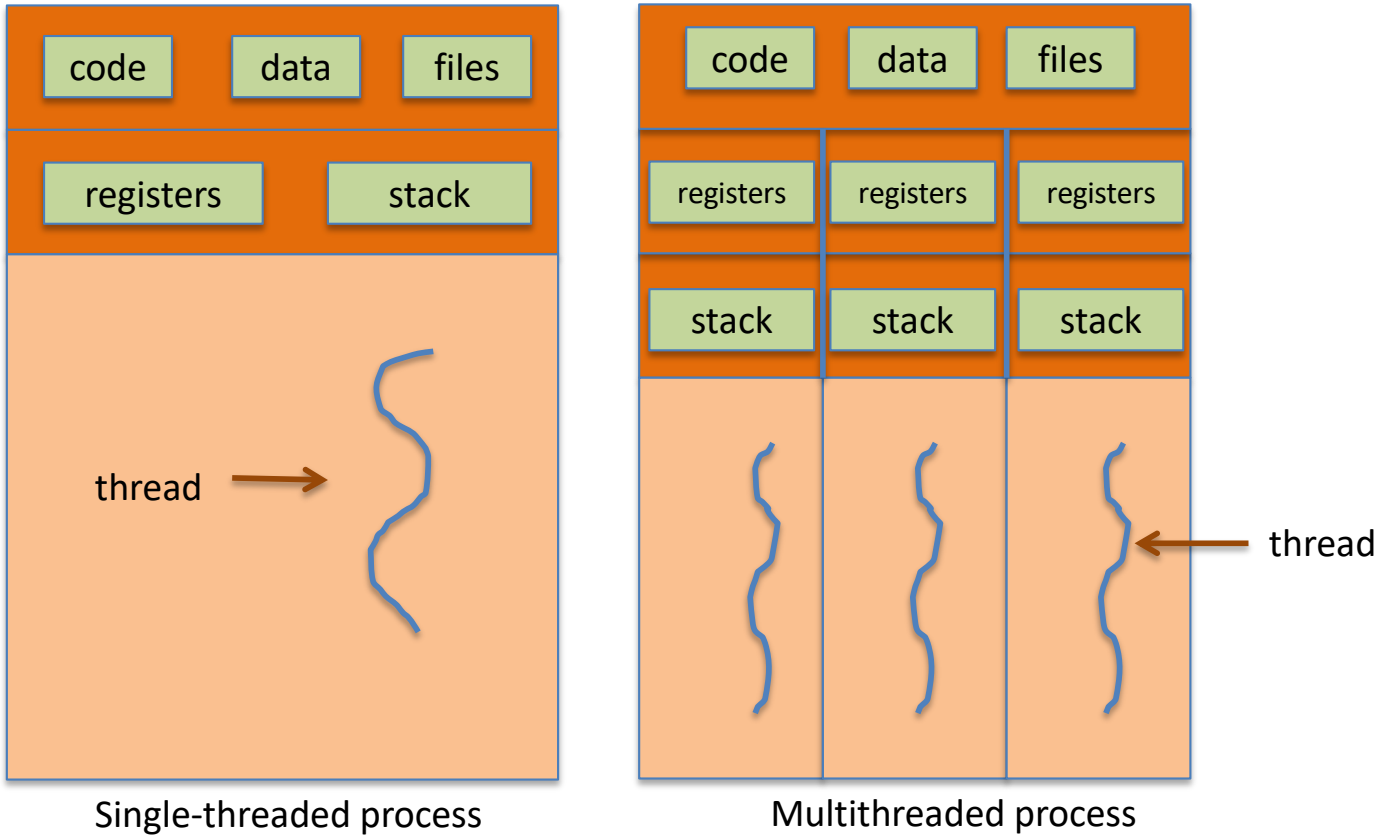
POSIX Threads

Didem Unat

Lecture 14

COMP304 - Operating Systems (OS)

Single vs Multithreaded Process



- A thread has an ID, a program counter, a register set, and a stack
- Shares the code section, data section and OS resources (e.g. files) with other threads within the same process

POSIX Threads API (Pthreads)

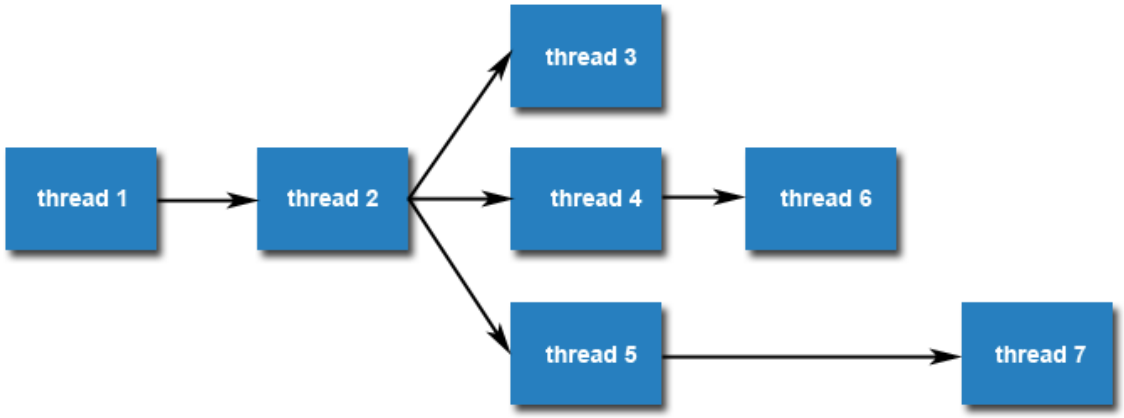
- Pthreads is the POSIX (Portable Operating System Interface for Unix) Thread Library
  - a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems
  - Solaris, Linux, Mac OS X.
- Is this a Kernel or User Level Thread API?

POSIX Threads API

- Functions of pthreads API provide:
  - Thread management:
    - Creation/termination of threads
    - Set/Query thread attributes
  - Mutexes, semaphores
  - Condition variables
- All identifiers in the thread library begin with pthread_

Creating Threads

- Initially, a 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.
- The maximum number of threads that may be created by a process is implementation dependent.
  - Programs that attempt to exceed the limit can fail or produce wrong results.
- Threads can create other threads (but there is no hierarchy)



Pthread_create()

- Forking Pthreads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (* start_routine)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id, &thread_attribute,
                         &thread_func, &func_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - standard default values obtained by passing a NULL pointer
- thread_func the function to be run (takes and returns void*)
- func_arg an argument can be passed to thread_fun when it starts
- errorcode will be set to nonzero if the create operation fails

Thread Creation

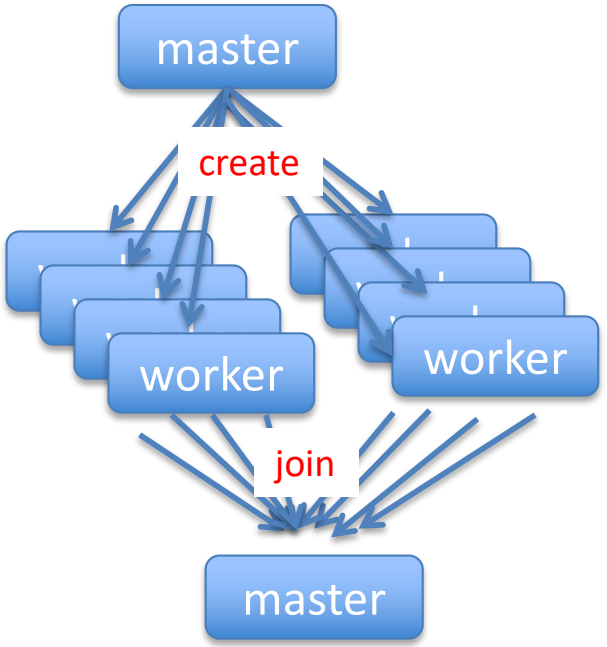
- Each thread executes a specific function, thread_func
  - For the program to perform different work in different threads, the arguments passed at thread creation distinguish the thread's "id" and any other unique features of the thread.

- After a thread is created, various attributes of it can be set
  - Priority of the thread
  - Stack size
  - Its scheduling policy

Simple Example

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

int main() {

  pthread_t threads[8];
  int tn;

  for(tn=0; tn<8; tn++) {
    pthread_create(&threads[tn], NULL, ParFun, NULL);
  }

  for(tn=0; tn<8 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

declares the various Pthreads functions, constants, types, etc.



- This code creates 8 threads that execute the function "ParFun".
- What happens to the master while workers are executing?
  - Does master become one of the workers?

Thread Termination

- **Pthread_exit()**
  - A thread returns from its starting routine by default, similar to a process terminating when it reaches to end of main
- **Pthread_cancel()**
  - Thread is cancelled by another thread via this call
- **Pthread_join(. . .)**
  - From Unix specification: “suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated.”

- After the last thread in a process terminates, the process terminates by calling exit()
- If the entire process is terminated, then all its threads will terminate

“Hello World”

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

int thread_count=16; //accessible by all threads

void *hello(void* rank); //thread function

int main() {
  pthread_t threads[16]; //thread handles

  int tn;
  for(tn=0; tn<16; tn++) {

    pthread_create(&threads[tn], NULL, hello,(void*)tn);

  }
  for(tn=0; tn<16 ; tn++) {

    pthread_join(threads[tn], NULL);

  }

  return 0;
}
```

Wait for thread completion

Start function to execute

Arguments to function

e.g. Thread ID

“Hello World”

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

int thread_count=16; //global var, accessible by all threads

void *hello(void* rank); //thread function

int main() {
  pthread_t threads[16]; //thread handles

  int tn;
  for(tn=0; tn<16; tn++) {

    pthread_create(&threads[tn], NULL, hello,(void*)tn);

  }

  pthread_exit(NULL);



  return 0;
}
```

Prevents the process to die before its threads are done!

main()/master will block and be kept alive to support the threads it created

``Hello World”

```
void *hello(void* id){

   int my_id = (int) id; //typecasting

   printf(“Hello from thread %d of %d\n”, my_id,
                                          thread_count);

   pthread_exit(0);
}
```

- By using thread id, different execution for each thread is possible
- How is possible for a thread to access thread_count?

Compiling a Pthread Program

```
gcc  −o pth_hello pth_hello.c −lpthread
```

The Pthreads API is only available on POSIX systems
— Linux, MacOS X, Solaris, HPUX, …

Because threads share resources:

- Changes made by one thread to shared system resources (such as closing a file) will be seen by all other threads.
- Two pointers having the same value point to the same data.
  - Because of the **shared address space**
- Reading and writing to the same memory locations is possible, and therefore requires **explicit synchronization** by the programmer.

Shared Data

- Variables declared outside of ‘main’ are **global variables**
  - Those variables are shared by all threads

- Object allocated on the **heap** may be shared if pointer is passed as an argument to the thread function

```
char *message = "Hello World!\n";
pthread_create( &thread1, NULL, (void*)&print_fun, (void*)message);
```

- Variables on the **stack are private** (locally defined variables)
  - Passing pointer to these around to other threads can cause problems

Thread Synchronization

- Need to protect the shared data and synchronize threads
- Pthread provides several ways to synchronize threads:
- Mutexes (Locks)
- Semaphores
- Condition Variables
- Barriers
  - Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.

Mutex (Locks) in Pthreads

```
#include <pthread.h>
pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

//create a mutex
pthread_mutex_init(&mymutex, NULL);

//use it
pthread_mutex_lock(&mymutex);
//mutually excluded code region
pthread_mutex_unlock(&mymutex);

//destroy a mutex
pthread_mutex_destroy(&mymutex);
```

```
pthread_mutex_lock(&our_lock);
counter++
pthread_mutex_unlock(&our_lock);
```

Condition Variables

- While **mutex**es implement synchronization by controlling thread access to data, condition variables allow threads to synchronize based upon the actual value of data.
  - Without condition variables, the programmer would need to have threads continually polling to check (possibly in a critical section) if the condition is met.

- A condition variable is always used in conjunction with a mutex lock.
- Set/query condition variable attributes

See an example usage:
https://hpc-tutorials.llnl.gov/posix/example_using_cond_vars/

Condition Variable Code Example

See the handout

1. The main thread creates three threads.
2. Two of those threads increment a "count" variable, while the third thread watches the value of "count".
3. When "count” reaches a predefined limit, the waiting thread is signaled by one of the incrementing threads.
4. The waiting thread "awakens" and then modifies count. The program continues until the incrementing threads reach TCOUNT.
5. The main program prints the final value of count.

Reference link:

https://hpc-tutorials.llnl.gov/posix/example_using_cond_vars/

Condition Variables (cont.)

- Main Thread
  - Declare and initialize global data/variables which require synchronization (such as "count")
  - Declare and initialize a condition variable object
  - Declare and initialize an associated mutex
  - Create threads A and B to do work

- Thread A
  - Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
  - Lock associated mutex and check value of a global variable
  - Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.
  - When signaled, wake up. Mutex is automatically and atomically locked.
  - Explicitly unlock mutex
  - Continue

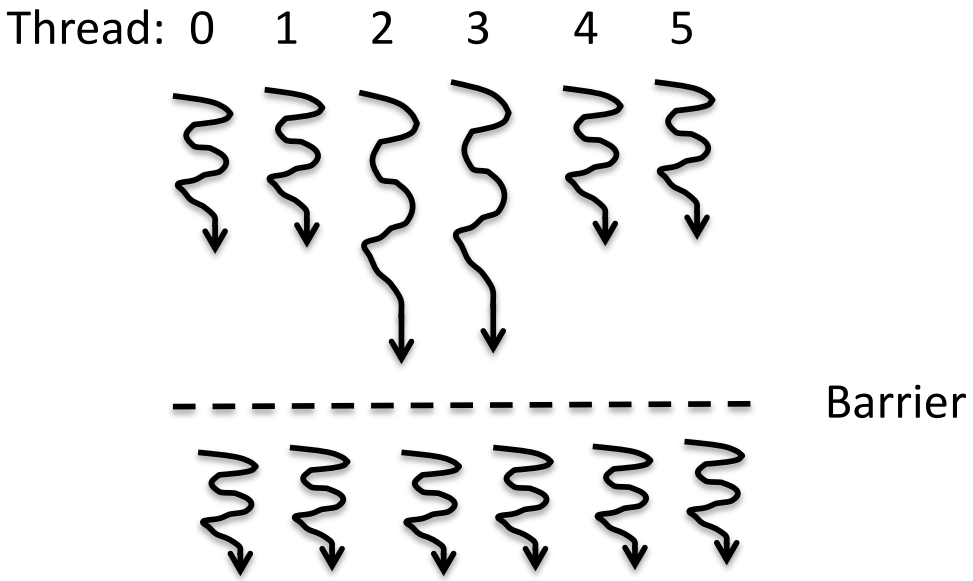
- Thread B
  - Do work
  - Lock associated mutex
  - Change the value of the global variable that Thread-A is waiting upon.
  - Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
  - Unlock mutex.
  - Continue

Semaphores in Pthreads

- Functions defined in semaphore.h:
- A semaphore is represented by a sem_t type.
- sem_init: for initializing semaphore
- sem_wait: for waiting on a semaphore
- sem_post: for signaling on a semaphore
- sem_destroy: for deallocating a semaphore if you no longer need it

Barriers

- Synchronizing the threads to make sure that they all are at the same point in a program is called a barrier.
- No thread can cross the barrier until all the threads have reached it.



Even though threads 2 and 3 reached barrier, they will wait for others to arrive. Then all threads cross the barrier point together.

Barriers in Pthreads

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
pthread_barrier_t b;
pthread_barrier_init(&b,NULL,3);
```

- The second argument specifies an object attribute; using NULL yields the default attributes.
- To wait at a barrier, a process executes:

```
pthread_barrier_wait(&b);
```

- To destroy a barrier

```
pthread_barrier_destroy(&b);
```

Thread Scheduling

- Threads can be scheduled by the operating system and run as independent entities
- Many-to-one and many-to-many models, thread library schedules user-level threads
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process
- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD_SCOPE_PROCESS schedules threads using PCS scheduling
  - PTHREAD_SCOPE_SYSTEM schedules threads using SCS scheduling
- Can be limited by OS – Linux and Mac OS X only allow PTHREAD_SCOPE_SYSTEM

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[]) {

   int i, scope, policy;
   pthread_t tid[NUM THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* first inquire on the current scope */
   if (pthread_attr_getscope(&attr, &scope) != 0)
      fprintf(stderr, "Unable to get scheduling scope\n");
   else {
      if (scope == PTHREAD_SCOPE_PROCESS)
         printf("PTHREAD SCOPE PROCESS");
      else if (scope == PTHREAD_SCOPE_SYSTEM)
         printf("PTHREAD SCOPE SYSTEM");
      else
         fprintf(stderr, "Illegal scope value.\n");
   }
```

Pthread Scheduling API (cont.)

```
/* get the current scheduling policy */
  if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
     fprintf(stderr, "Unable to get policy.\n");
  else {
     if (policy == SCHED_OTHER) printf("SCHED OTHER\n");
     else if (policy == SCHED_RR) printf("SCHED RR\n");
     else if (policy == SCHED_FIFO) printf("SCHED FIFO\n");
  }

/* set the scheduling policy - FIFO, RR, or OTHER */
  if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0)
     fprintf(stderr, "Unable to set policy.\n");
...
```

Pthread Scheduling API (cont.)

```
   /* set the scheduling algorithm to PCS or SCS */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
   /* create the threads */
   for (i = 0; i < NUM THREADS; i++)
      pthread_create(&tid[i],&attr,run_method,NULL);
   /* now join on each thread */
   for (i = 0; i < NUM THREADS; i++)
      pthread_join(tid[i], NULL);
}
/* Each thread will begin control in this function */
void *run_method(void *param)
{
   /* do some work ... */
   pthread_exit(0);
}
```

Summary of Pthreads

- Pthreads are user-level threads for POSIX systems
  - Familiar language for most programmers, particularly for systems people
  - Ability to shared data is convenient
  - Various supports for synchronization
- Pthread Tutorial
  - https://hpc-tutorials.llnl.gov/posix/example_using_cond_vars/
- Reading from Book
  - 4.1, 4.3, 4.6 , 4.7
- Acknowledgements
  - These slides are adapted from
    - Öznur Özkasap (Koç University)
    - Operating System and Concepts (9th edition) Wiley
    - https://computing.llnl.gov/tutorials/pthreads/