Threads

- A thread is an execution stream within a process
- A sequence of control within a process
- A sub-process that shares an address space with the parent process
- An execution state of a process

Difference between for system call and creation of new threads :

- When a process executes a **fork call**, a **new copy of the process is created** with its own variables and its own PID. This new process is scheduled independently, and (in general) executes almost independently of the process that created it.

- When we create a **new thread** in a process, the new thread of execution gets its own stack (and hence local variables) but shares global variables, file descriptors, signal handlers, and its current directory state with the process that created it.

What makes a process?

- Address space
- Privileges
- Resources
- Code Segment
- Execution State : PC, SP, registers
- Can separate the execution state from the rest of the process

Processes and Threads

- A thread is bound to a **single** process
- Processes can have **many** threads
- Each process has at least one thread
- A thread **shares** the following with the process that created it
    - PID
    - Address space
    - File descriptors
    - Signal Handlers
    - Privileges
- A thread has its own set of local variables

Threads

- Creation
- Joining
- Terminating

Why Use Threads?

Advantages of threads over forked processes

- Fork is expensive.
- Threads are **lightweight**
    - Not a big deal in linux
    - Thread creation is about 10-100 times faster than process creation.
- **Concurrent execution** with shared data
    - GUI
        - Service display during computation
        - Edit text while doing a word count

- DB servers : the requirements for **locking** and data consistency cause the different processes to be very **difficult** . This can be done much more **easily** with multiple threads than with multiple processes.

- Multimedia applications : **animation**

- More control over execution and better to utilize the hardware resources available

Because **threads** are relatively lightweight, **multi-threaded applications are preferred** over multi-process applications in many situations

- Web servers
- Almost all client-server applications that need to service more than one request at a time
- The operating system mutex (mutual execution)
- Scientific applications on distributed systems
- Applications that do network or socket I/O

Thread Drawbacks

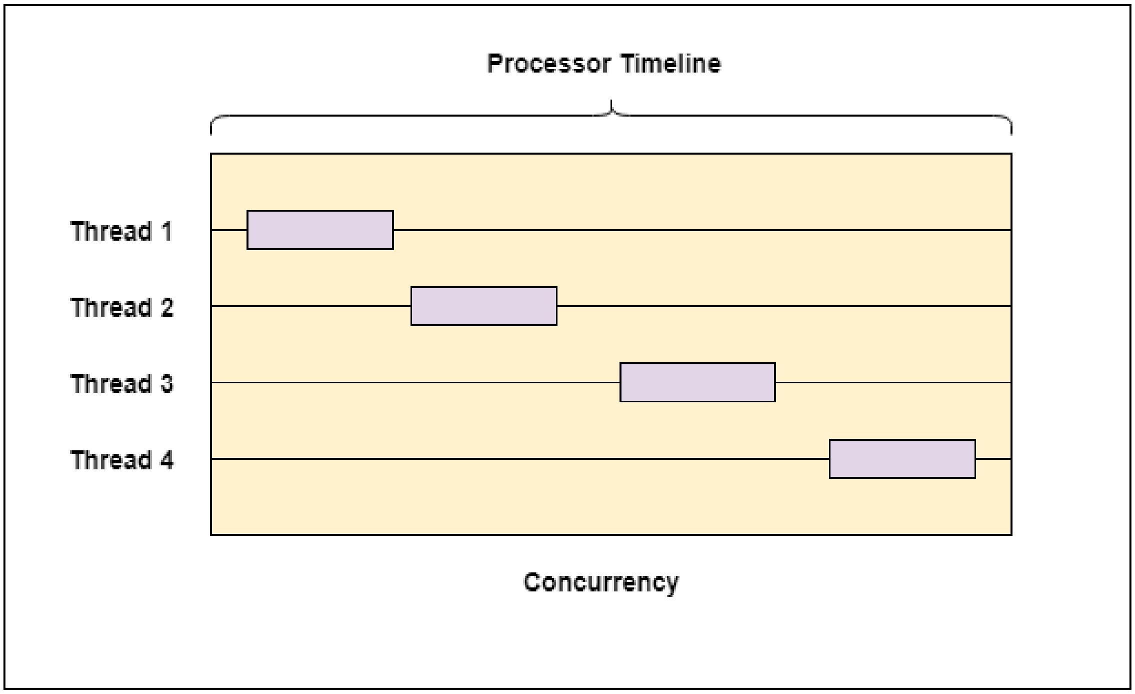
- **Hard to program**
  - Mostly for experts and wizards
- **Debugging** a multithreaded program is much, much harder than debugging a single-threaded one, because the interactions between the threads are very hard to control
- **Synchronization**
  - Must coordinate access to shared data with locks
  - If we forget a lock we end up with corrupt data
- **Deadlock**
  - Circular dependences in locks
  - A waits for B to finish, B waits for A to finish

Note :

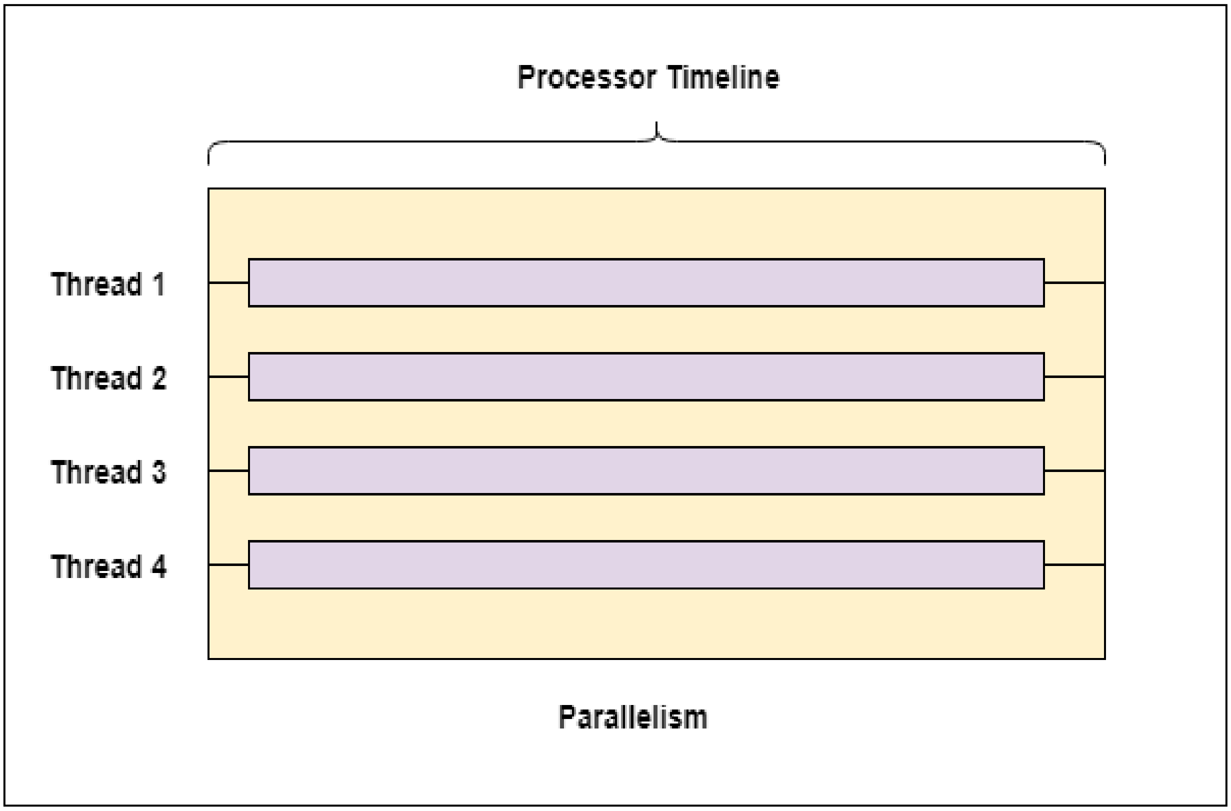
A program that splits a large calculation into two and runs the two parts as different threads will **not necessarily run more quickly on a single processor machine**, unless the calculation truly allows multiple parts to be calculated simultaneously and the machine it is executing on has multiple processor cores to support true multiprocessing.

Concurrency vs Parallelism

- Concurrency **does not imply** parallelism
- Parallelism means **that multiple processes or threads are making progress in parallel**. This means that the threads are executing at the same time. Need **more than one CPU** (or core) for parallelism
- Concurrency means **that multiple processes or threads are making progress concurrently**. While **only one** thread is executed **at a time** by the CPU. Benefit from concurrency is realized when we overlap computation with I/O.



All the four threads are running concurrently. However, only one of them can be scheduled on a processor at a time.

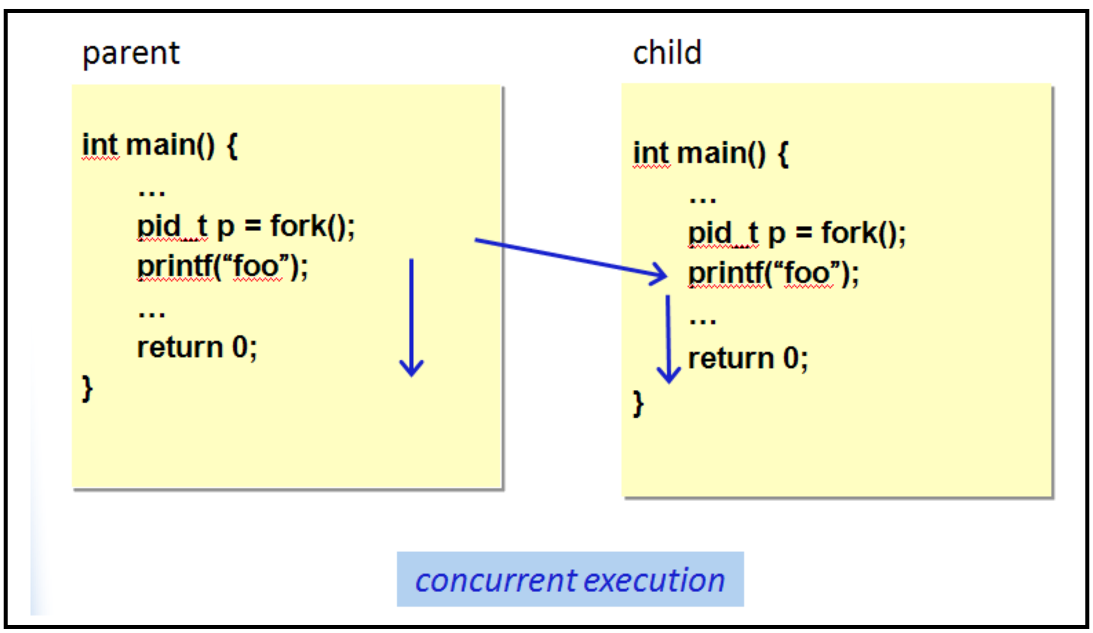


All the four threads are running in parallel i.e. they are executing at the same time.

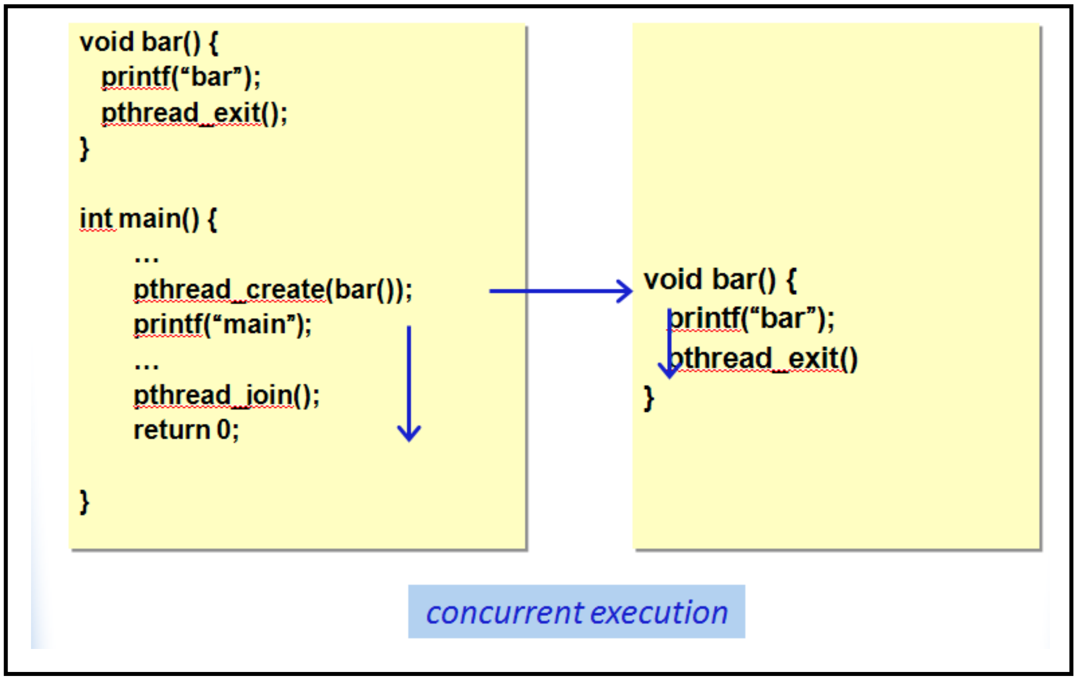
Thread Functions

pthread_create() /* equivalent to fork */
pthread_join() /* equivalent to wait */
pthread_exit() /* equivalent to exit */

fork() Control



Thread Control



pthread() and fork() Execution Differences

- In **fork()** both parent and child start executing the very next statement following the **fork()**
- In **pthread_create()** the new thread starts executing code from a function specified as an argument
- The parent/main thread executes the statement following the **pthread_create()** call

Creating a Thread

Header

#include<pthread.h>

Prototype

```
int pthread_create(pthread_t*  thread,
                   pthread_attr_t*  attr,
                   void*  (*start_routine) (void*),
                   void* arg);
```

- **pthread_create()** **creates** a new thread that executes concurrently with the calling thread

- **First argument**
  - A handle on the newly created thread
  - **Declare a pthread_t variable** and pass its address
  - Need this because no PID

- **Second argument**
  - A set of attributes for finer control over the new thread
  - In most cases, we will pass **NULL**

- **Third argument is a pointer to a user-defined function**
    - The function takes one argument of type void *
        - *implies, we can pass a single argument of any type*
    - The function returns void *
        - *implies, we can return any type*

- **The newly created thread will start executing the first statement in the user-defined function**

- **The thread terminates when the function terminates**

pthread_create() return value

- If **pthread_create()** is successful
    - A handle for the newly created thread is stored in the location pointed to by the first argument
    - **0 is returned**

- If **pthread_create()** fails
    - a non-zero error code is returned
    - possible values
        - EAGAIN
            - Exceeded max thread count
        - EINVAL
            - Invalid attribute

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

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()

void *myThreadFun(void *vargp) {

        printf("Printing from My Thread Function \n");
        return NULL;
}

int main() {
        pthread_t    thread_id;
        printf("Before Creating Thread\n");
        pthread_create(&thread_id, NULL, &myThreadFun, NULL);
        printf("After Thread is Done\n");
        exit(0);
}
```

Sample run

husain-gholooms-macbook:~ husaingholoom$ **gcc -lpthread ztest0.c**
husain-gholooms-macbook:~ husaingholoom$ ./a.out

Before Creating Thread
Printing from My Thread Function
After Thread is Done

husain-gholooms-macbook:~ husaingholoom$

Example

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

/*thread function definition*/
void* threadFunction(void* args) {
	while (1) {
		printf("I am threadFunction.\n");
	}
}
int main() {
	/*creating thread id*/
	pthread_t id;
	int ret;

	/*creating thread*/
	ret = pthread_create(&id, NULL, &threadFunction, NULL );
	if (ret == 0) {
		printf("Thread created successfully.\n");
	} else {
		printf("Thread not created.\n");
		return 0; /*return from main*/
	}

	while (1) {
		printf("I am main function.\n");
	}

	return 0;
}
```

Sample run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out

Thread created successfully.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
I am main function.
I am threadFunction.
```

.

.

Joining a thread

Header

#include<pthread.h>

Prototype

```
int pthread_join (pthread_t thread,
                  void **thread_return);
```

- Notion of a thread join is very similar to **wait()**

- **pthread_join()** suspends the execution of the calling thread until the thread identified by the argument thread, terminates

    - blocking operation

- The return value (exit status) is stored in the location pointed to by **thread_return**

Example - 1

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()

void *myThreadFun(void *vargp) {
	sleep(1);
	printf("Printing from My Thread Function \n");
	return NULL;
}

int main() {
	pthread_t thread_id;
	printf("Before Creating Thread\n");
	pthread_create(&thread_id, NULL, myThreadFun, NULL);
	pthread_join(thread_id, NULL);
	printf("After Thread is Done\n");
	exit(0);
}
```

Sample run

husain-gholooms-macbook:~ husaingholoom$ **gcc -lpthread ztest0.c**
husain-gholooms-macbook:~ husaingholoom$ ./a.out

Before Creating Thread
Printing from My Thread Function
After Thread is Done

husain-gholooms-macbook:~ husaingholoom$

Example - 2

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

void *print_message_function(void *ptr);

main() {
      pthread_t thread1, thread2;
      char *message1 = "Thread 1";
      char *message2 = "Thread 2";
      int iret1, iret2;

/* Create independent threads each of which will execute function */

      iret1 = pthread_create(&thread1, NULL, print_message_function,
                  (void*) message1);
      iret2 = pthread_create(&thread2, NULL, print_message_function,
                  (void*) message2);

/* Wait till threads are complete before main continues. Unless we   */
/* wait we run the risk of executing an exit which will terminate    */
/* the process and all threads before the threads have completed.    */

      pthread_join(thread1, NULL );
      pthread_join(thread2, NULL );

      printf("Thread 1 returns: %d\n", iret1);
      printf("Thread 2 returns: %d\n", iret2);
      exit(0);
}

void *print_message_function(void *ptr) {
      char *message;
      message = (char *) ptr;
      int i = 0;
      for (i = 0; i < 3; i++)
            printf("%s \n", message);
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

```
Thread 1
Thread 1
Thread 1
Thread 2
Thread 2
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

husain-gholooms-macbook:~ husaingholoom$

Exiting a Thread

Header

#include<pthread.h>

Prototype

void pthread_exit (void *retval)

- **pthread_exit()** **terminates** the execution of the current thread
- semantics of **pthread_exit()** is similar to **exit()**
- should use it with threads created with **pthread_creat()**
    - on some systems, called implicitly

void pthread_exit (void *retval) is

- A pointer to a **variable of any type is** passed as an argument to **pthread_exit()**
- This value is returned to the caller
- Do not pass pointers to **local variables. This** could lead to elusive bugs
- You can register exit handlers using a mechanism that is similar to **atexit()**

- Won't cover this in class
- Do a man on
    - **pthread_cleanup_push**
    - **pthread_cleanup_pop**

Terminating a Thread

Header

#include<pthread.h>

Prototype

int pthread_cancel (pthread_t thread)

- **pthread_cancel() sends a cancellation request** to the thread denoted by the thread argument

- like signals, cancel **requests are asynchronous**

- **Do not need** parent-child relationship

- The **only requirement** is that we have a **handle** on the thread

- Cancellation is very **similar to terminating** a process using a **signal**

- Depending on its settings, **a target thread can**
  - **Ignore** the request
  - **Honor** it immediately
  - **Defer** it till it reaches a cancellation point

Example Using Cancel

```
// C program to demonstrates cancellation of another thread
// using thread id

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <pthread.h>

// To Count

int counter = 0;

// for temporary thread which will be
// store thread id of second thread

pthread_t    tmp_thread;

// thread_one  call func

void* func(void* p)
{
      while (1) {

            printf("\nthread number one\n");
            sleep(1); // sleep 1 second
            counter++;
```

```
        // for exiting if counter == 2
        if (counter == 2) {

            // for cancel thread_two
            pthread_cancel(tmp_thread);

            // for exit from thread_one
            pthread_exit(NULL);
        }
    }
}
```

// thread_two call func2

```
void* func2(void* p)
{
    // store thread_two id to tmp_thread

    tmp_thread = pthread_self();    // pthread_self() gets
                                    //  the ID of the current
                                    //  thread.
    while (1) {
        printf("\nthread Number two\n");
        sleep(1); // sleep 1 second
    }
}
```

```
// Driver code

int main()
{
    // declare two thread
    pthread_t   thread_one,   thread_two;

    // create thread_one
    pthread_create(&thread_one, NULL, func, NULL);

    // create thread_two
    pthread_create(&thread_two, NULL, func2, NULL);

    // waiting for when thread_one is completed
    pthread_join(thread_one, NULL);

    // waiting for when thread_two is completed
    pthread_join(thread_two, NULL);

     printf("\n\n... Main End \n\n");
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

thread number one

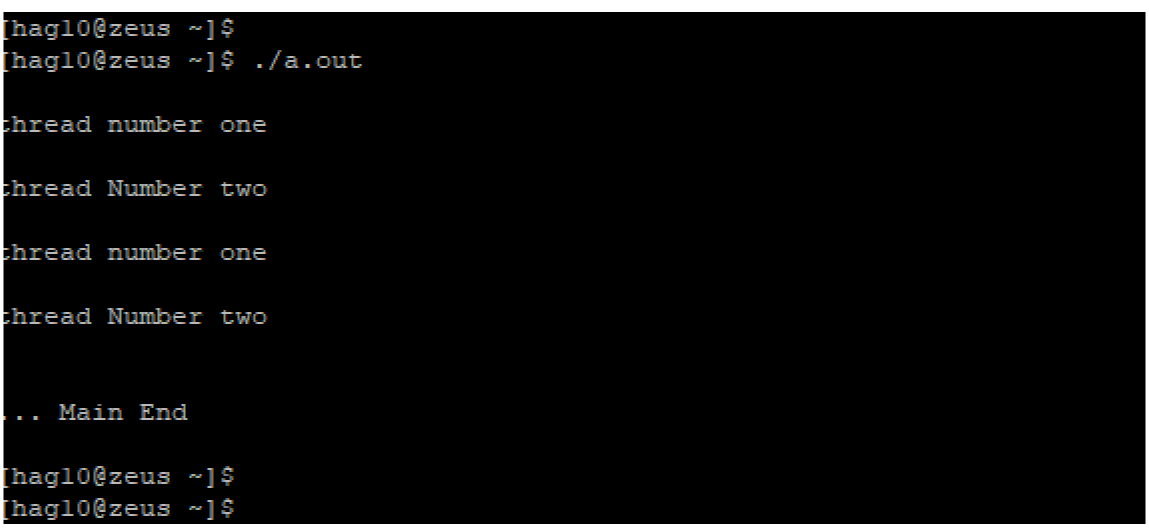
thread Number two

thread number one

thread Number two

... Main End

husain-gholooms-macbook:~ husaingholoom$



Example using exit

```
#include <pthread.h>
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <errno.h>

#include <unistd.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);
       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);
 }
```

Sample run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out

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

husain-gholooms-macbook:~ husaingholoom$
```

Or

Sample run

```
husain-gholooms-macbook:~ husaingholoom$ ./a.out

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

husain-gholooms-macbook:~ husaingholoom$
```

```
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
Hello World! It's me, thread #0!
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!
[hag10@zeus ~]$
[hag10@zeus ~]$
```

Passing Data To Threads

- All arguments must be passed by reference and cast to (void *)

- For multiple arguments
  - Create a structure that contains all of the arguments
  - Pass a pointer to that structure in pthread_create()

- Of course **globals** are always an option
  - generally want to **avoid** this

Passing Data To Threads : Examples

Passing an int

```
int i = 42;
pthread_create(..., my_func, (void *)&i);
```

Passing a C-string:

```
char *str = "foobar";
pthread_create(..., my_func, (void *)str);
```

Passing an array

```
int arr[100];
pthread_create(..., my_func, (void *)arr);
```

Passing Arguments To Threads : Safety

- Given the **non-deterministic execution schedule of threads** , need to be careful about passing data values to newly-created threads
  - Ensure all passed data is thread **safe**
  - **Cannot be changed by other threads** or changed only with mutual exclusion locks

Example

- This program creates a single extra thread
- Shows that it is sharing variables with the original thread
- And gets the new thread to return a result to the original thread.

Example : Mutlithread

```
/* Includes */
#include <unistd.h>          /* Symbolic Constants */
#include <sys/types.h>       /* Primitive System Data Types */
#include <errno.h>           /* Errors */
#include <stdio.h>           /* Input/Output */
#include <stdlib.h>          /* General Utilities */
#include <pthread.h>         /* POSIX Threads */
#include <string.h>          /* String handling */

/* prototype for thread routine */

void print_message_function ( void *ptr );

/* struct to hold data to be passed to a thread.
  this shows how multiple data items can be passed to a thread
*/

typedef  struct   str_thdata     //  for multiple values
{
   int      thread_no;
   char    message[100];
}   thdata;
```

```
int main()
{
   pthread_t   thread1,   thread2;  /* thread variables */
   thdata   data1,   data2;              /* structs to be passed */
                                        /*  to  threads */

/* initialize data to pass to thread 1 */
   data1.thread_no = 1;
   strcpy(data1.message, "Hello!");

/* initialize data to pass to thread 2 */
   data2.thread_no = 2;
   strcpy(data2.message, "Hi!");

/* create threads 1 and 2 */

pthread_create (&thread1, NULL, (void *)
        &print_message_function, (void *) &data1);
pthread_create (&thread2, NULL, (void *)
       &print_message_function, (void *) &data2);
```

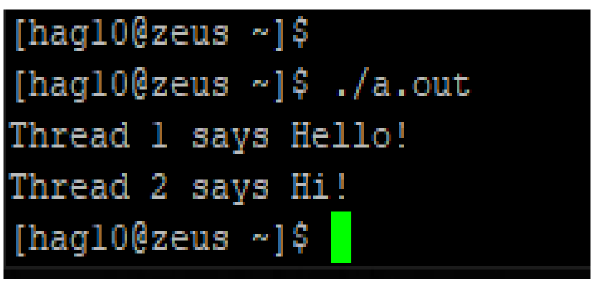
```
/* Main block now waits for both threads to terminate,
   before it exits  If main block exits, both threads exit,
   even if the threads  have not  finished their work  */

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);

/* exit */

   exit(0);

} /*  end of main() */
```

```
/***   print_message_function is used as the start routine
        for the threads used   it accepts a void pointer     **/

void print_message_function ( void *ptr )
{
   thdata  *data;
   data = (thdata *) ptr;  /* type cast to a pointer to thdata */

   /* do the work */
   printf("Thread %d says %s \n", data->thread_no,
          data->message);

   pthread_exit(0); /* exit */

}

/* print_message_function ( void *ptr ) */
```

Sample Run

192:thread husaingholoom$ ./a.out

Thread 1 says Hello!
Thread 2 says Hi!
192:thread husaingholoom$



Thread Synchronization

Two methods

- **Mutexes**
- **Semaphores**

Mutex

- A mutex is a MUTual EXclusion **device**
- **Useful for**
  - **Protecting shared data structures from concurrent modifications**
  - **Implementing critical sections**
- **Mutex** Comes from the **OS** world
  - Process scheduling and I/O consistency
- Mutes has **become a user-level tool for thread synchronization**
  - APIs allow us to create and manipulate mutexes in software
  - Kernel does a lot of the work behind the curtains
    - We won't cover that in this class

Mutex Semantics

- Code enclosed in a mutex can be executed by only one thread at a time

```
void *thread_func(void *arg) {
  ... // unguarded code
      BEGIN MUTEX
       i++;
      END MUTEX
  ... // unguarded code
}
```

- Data Structures enclosed within a mutex can be **accessed by only one thread at a time**
- For a thread to execute the code within a mutex, the thread needs to acquire a **lock on the mutex**
- If the mutex is already locked by another thread, then the thread must **wait** for it to be unlocked
- Once a **lock has been acquired** a thread executes the code enclosed in the mutex and then **releases** the lock
- This mechanism ensures in-order modifications of data structures within code sections enclosed in a mutex

Mutex code template

```
#include<pthread.h>

...

int foo;

pthread_mutex_t   myMutex;      /* needs to be visible to
                                   all threads in contention

pthread_mutex_lock(&myMutex);       /* at any point only one thread is
/ * code that modifies foo */          going to execute this code */
pthread_mutex_unlock(&myMutex);
```

5 Steps to Using a mutex

1. Declare a mutex object
2. Initialize mutex
3. Acquire lock on mutex
4. Release lock on mutex
5. Destroy mutex

Declaring a mutex

Header

#include<pthread.h>

Declaration

pthread_mutex_t   mutex;

- Needs to be visible to all threads that want to acquire a lock on this mutex
- Typically declared as global

Initializing a mutex

Header

#include <pthread.h>

Prototype

int pthread_mutex_init (pthread_mutex_t   *mutex,

const pthread_mutexattr_t   *mutexattr);

- Use this function to **initialize** mutexes that are allocated dynamically
- For statically allocated mutexes, use **PTHREAD_MUTEX_INITIALIZER**
- **pthread_mutex_init()** **initializes the mutex** object pointed to by mutex according to the mutex attributes specified in **mutexattr**
- If **mutexattr** is **NULL**, default attributes are used instead
- Example attribute
  - *Allow multiple concurrent locks*
- **We won't deal with mutex attributes in this course**
  - We will still need to initialize the mutex

Acquiring a mutex Lock

Header

#include <pthread.h>

Prototype

int pthread_mutex_lock (pthread_mutex_t
***mutex));**

- **pthread_mutex_lock()** locks the given mutex

- If the mutex is currently unlocked, it becomes locked and owned by the calling thread, and **pthread_mutex_lock()** returns immediately

- If the mutex is already locked by another thread, **pthread_mutex_lock()** suspends the calling thread until the mutex is unlocked

Releasing a mutex Lock

Header

#include <pthread.h>

Prototype

int pthread_mutex_unlock (pthread_mutex_t *mutex);

- **pthread_mutex_unlock()** **unlocks** the given mutex

- The mutex is assumed to be locked and owned by the calling thread on entrance to

  pthread_mutex_unlock()

Destroying a mutex

Header

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

Prototype

```
int pthread_mutex_destroy (pthread_mutex_t
                            *mutex);
```

- **pthread_mutex_destroy()** destroys a mutex, which must not be reused until it is reinitialized
- This function always returns 0

Summary

- Synchronization is a mechanism that allows us **to run parts of a parallel or concurrent program sequentially**
- Required in most parallel applications to **preserve data dependence**
- Thread synchronization is usually data centric
- A **mutex** is one method of synchronizing threads
    - Code enclosed in a mutex **is guaranteed to be executed by one thread** at a time
    - The **programmer** is responsible for setting up the mutexes for preserving the semantics of the program

Example : Without mutex

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

pthread_t   tid[2];
int counter = 0;

void* trythis(void *arg) {
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < 100000000; i++)
        ;
    printf("\n Job %d has finished\n", counter);

    return NULL;
}

int main(void) {
    int i = 0;
    int error;

    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL,
        &trythis, NULL);

        if (error != 0)
            printf("\nThread can't be created : [%s]",
                  strerror(error));
        i++;
    }
```

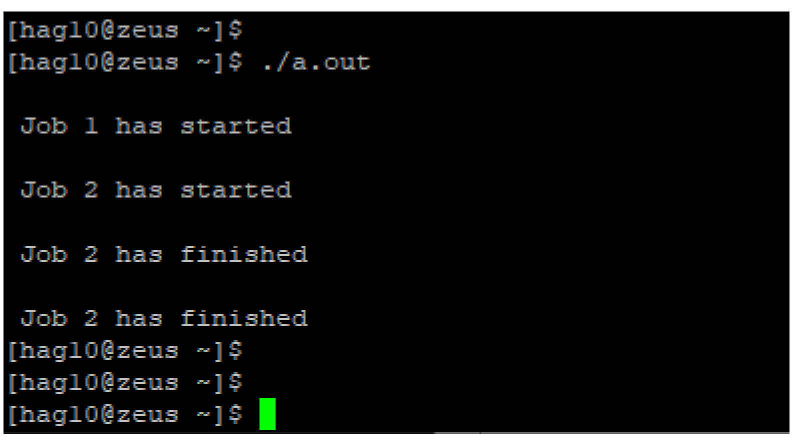
```
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

Job 1 has started
Job 2 has started
Job 2 has finished
Job 2 has finished

husain-gholooms-macbook:~ husaingholoom$



Example : With mutex

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

pthread_t tid[2];
int counter = 0;

pthread_mutex_t lock;

void* trythis(void *arg) {
     pthread_mutex_lock(&lock);
     unsigned long i = 0;
     counter += 1;
     printf("\n Job %d has started\n", counter);
     for (i = 0; i < 100000000; i++)
          ;
     printf("\n Job %d has finished\n", counter);

     pthread_mutex_unlock(&lock);
     return NULL;
}

int main(void) {
     int i = 0;
     int error;

     if (pthread_mutex_init(&lock, NULL) != 0) {
          printf("\n mutex init has failed\n");
          return 1;
     }
```

```
    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL,
        &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]",
                    strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Sample Run

husain-gholooms-macbook:~ husaingholoom$ ./a.out

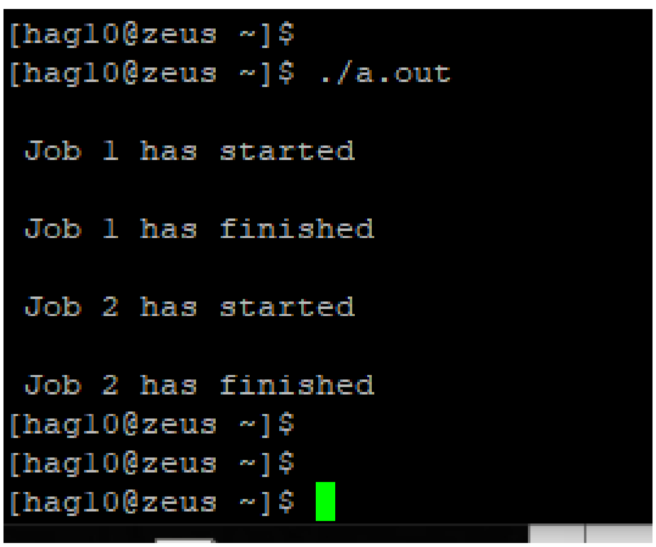
Job 1 has started

Job 1 has finished

Job 2 has started

Job 2 has finished

husain-gholooms-macbook:~ husaingholoom$



Efficient Synchronization

- The thread that wants to acquire a lock needs to ***constantly*** check the state of the mutex
  - *this type of check is sometimes referred to as* ***a busy-wait loop***
- When synchronizing threads, generally want to **avoid** busy-wait loops
- pthreads provide two functions that allow us to avoid busy-wait loops
  - **pthread_cond_wait();**
  - **pthread_cond_signal();**

- These functions **work in pairs** and are implemented based on conditional variables (a la semaphores)

pthread_cond_wait()

Header

#include <pthread.h>

Prototype

```
pthread_cond_wait(thread_cond_t   *restrict cond,
          pthread_mutex_t   *restrict mutex);
```

- **pthread_cond_wait()** atomically unlocks the mutex argument and waits on the cond argument
- returns after it receives the cond signal
- before returning control to the calling function, re-acquires the mutex

pthread_cond_signal()

Header

#include <pthread.h>

Prototype

pthread_cond_signal(pthread_cond_t   *cond);

- **pthread_cond_signal()** unblocks one thread waiting for the condition variable cond

Pthread : lock – unlock - wait and signal

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

// Declaration of thread condition variable
pthread_cond_t  cond1 = PTHREAD_COND_INITIALIZER;

pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

int done = 1;

void* foo() {

    pthread_mutex_lock(&lock);
    if (done == 1) {

        // let's wait on condition variable cond1
        done = 2;
        printf("Waiting on condition variable
                cond1\n");
        pthread_cond_wait(&cond1, &lock);
    } else {

        // Let's signal condition variable cond1
        printf("Signaling condition variable
                cond1\n");
        pthread_cond_signal(&cond1);
    }
```

```
    // release lock
    pthread_mutex_unlock(&lock);

    printf("Returning thread\n");

    return NULL ;
}

// Driver code

int main() {
    pthread_t tid1, tid2;

    // Create thread 1
    pthread_create(&tid1, NULL, foo, NULL );

    // sleep for 1 sec so that thread 1
    // would get a chance to run first
    sleep(1);

    // Create thread 2
    pthread_create(&tid2, NULL, foo, NULL );

    // wait for the completion of thread 2
    pthread_join(tid2, NULL );

    return 0;
}
```

Sample Run

```
Waiting on condition variable cond1
Signaling condition variable cond1
Returning thread
Returning thread
```

```
[hag10@zeus ~]$
[hag10@zeus ~]$ ./a.out
Waiting on condition variable  cond1
Signaling condition variable   cond1
Returning thread
Returning thread
[hag10@zeus ~]$
```

Deadlock

- Recall that one definition of an operating system is a resource allocator.

- There are many resources that can be allocated to only one process at a time, and several operating system features allows this, such as mutexes, semaphores or file locks.

- Sometimes a process has to reserve more than one resource. For example, a process which copies files from one tape to another generally requires two tape drives.

- In general, resources allocated to a process are not preemptable; this means that once a resource has been allocated to a process, there is **no** simple mechanism by which the system can take the resource back from the process unless the process voluntarily gives it up or the system administrator kills the process.

- This can lead to a situation called ***deadlock***. A set of processes or threads is deadlocked when each process or thread is waiting for a resource to be freed which is controlled by another process.

Here is an example of a situation where deadlock can occur.

```
Mutex M1, M2;

/* Thread 1 */
while (1) {
   NonCriticalSection()
   Mutex_lock(&M1);
   Mutex_lock(&M2);
   CriticalSection();
   Mutex_unlock(&M2);
   Mutex_unlock(&M1);
}

/* Thread 2 */
while (1) {
   NonCriticalSection()
   Mutex_lock(&M2);
   Mutex_lock(&M1);
   CriticalSection();
   Mutex_unlock(&M1);
   Mutex_unlock(&M2);
}
```

Suppose thread 1 is running and locks M1, but before it can lock M2, it is interrupted.

Thread 2 starts running; it locks M2, when it tries to obtain and lock M1, it is blocked because M1 is already locked (by thread 1).

Eventually thread 1 starts running again, and it tries to obtain and lock M2, but it is blocked because M2 is already locked by thread 2.

Both threads are blocked; each is waiting for an event which will never occur.

In general , In order for deadlock to occur, four conditions must be true.

- **Mutual exclusion** - Each resource is either currently allocated to exactly one process or it is available. (Two processes cannot simultaneously control the same resource or be in their critical section).

- **Hold and Wait** - processes currently holding resources can request new resources

- **No preemption** - Once a process holds a resource, it cannot be taken away by another process or the kernel.

- **Circular wait** - Each process is waiting to obtain a resource which is held by another process.

Example Of a deadlock

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

pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutex2 = PTHREAD_MUTEX_INITIALIZER;


// These two functions will run concurrently.
void* print_i(void *ptr) {
  pthread_mutex_lock(&mutex1);
  pthread_mutex_lock(&mutex2);
  printf("I am in i");
  pthread_mutex_unlock(&mutex2);
  pthread_mutex_unlock(&mutex1);
}

void* print_j(void *ptr) {
  pthread_mutex_lock(&mutex2);
  pthread_mutex_lock(&mutex1);
  printf("I am in j");
  pthread_mutex_unlock(&mutex1);
  pthread_mutex_unlock(&mutex2);
}

int main() {
  pthread_t t1, t2;
  int iret1 = pthread_create(&t1, NULL, print_i, NULL);
  int iret2 = pthread_create(&t2, NULL, print_j, NULL);

  while(1){}
  exit(0); //never reached.
}
```