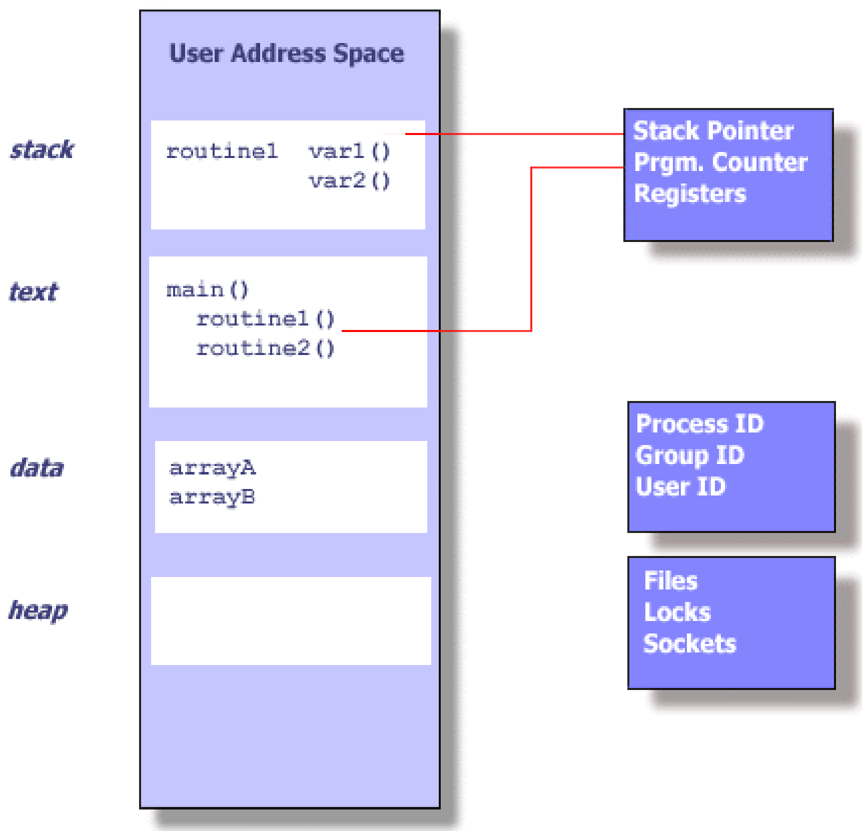
Threads Programming in Linux

What is a Process ?

- Fundamental to almost all operating systems
- = program *in execution*
- Seperate **address space**
- program counter, stack pointer, hardware registers



What is a Process ?

- Processes contain **lots of information**:
  - Process ID, process group ID, user ID, and group ID
  - Environment
  - Working Directory
  - Program Instructions
  - Registers
  - Stack
  - Heap
  - File Descriptors
  - Signal Actions
  - Shared Libraries
  - Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory)

What is a Process ?

- OS alternate between processes
  - run process on CPU
  - clock interrupt happens
  - save **process state**
    - registers (PC, SP, numeric)
    - memory map (address space)
    - memory (core image) → possibly swapped to disk
    - → ***process table (PCB)***
  - **continue some other process**

Scheduling

Context Switch

Remember Unix process creation: fork()

```
#include <sys/types.h>
#include <unistd.h>
int v = 42;
int main (void)
{
   pid_t pid;
   if ((pid = fork()) < 0) {
      perror("fork");
      exit(1);
   } else if (pid == 0) {
      printf("child %d of parent %d\n", getpid(), getppid());
      v++;
      printf(“child:%d \n”,v);
   }
   else {
     sleep(10);
     printf(“child:%d \n”,v);
  }
  return 0;
}
```

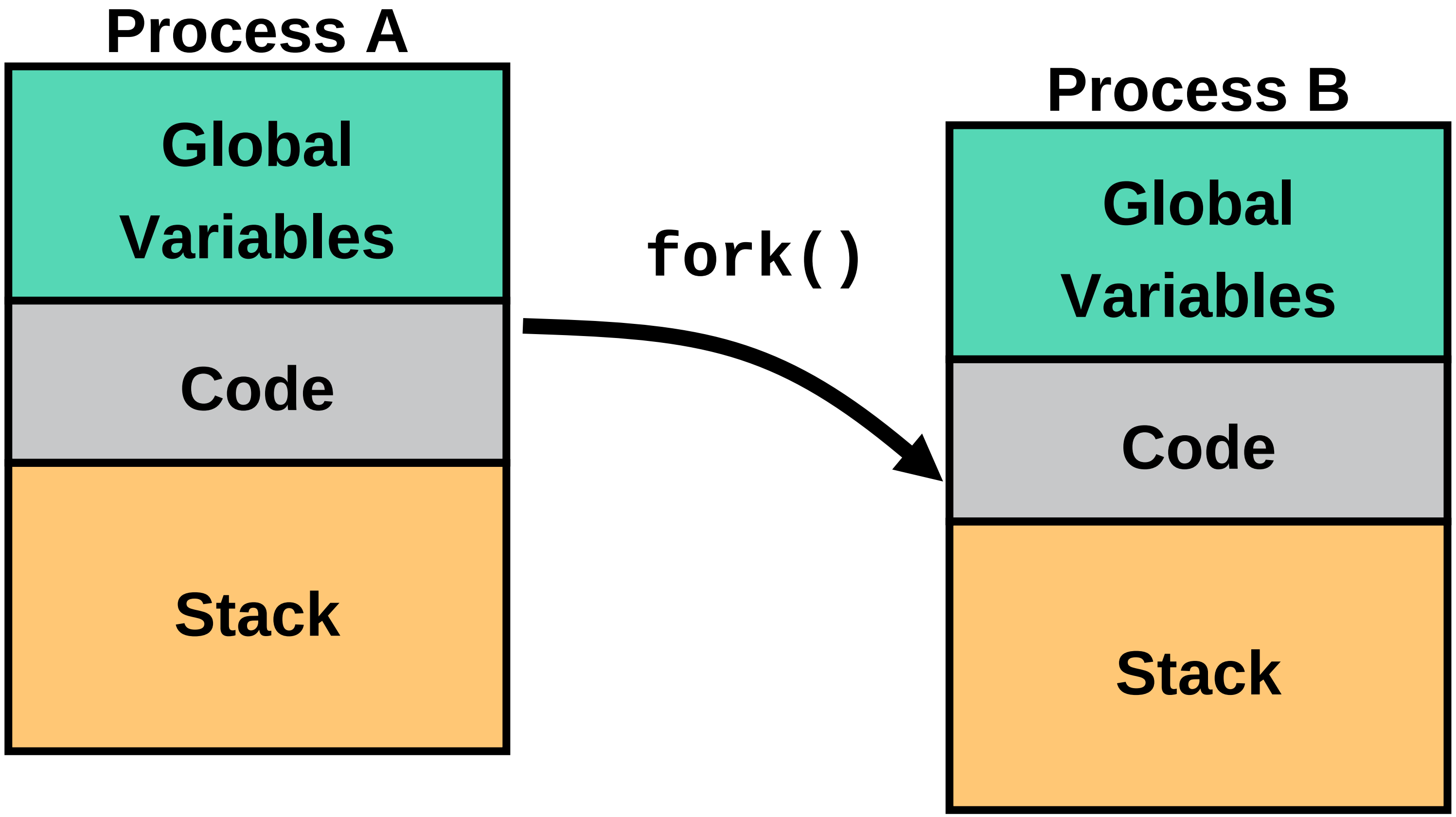
Global variable

Parent process forked child process here

Child process executes here

Parent process executes here

fork()

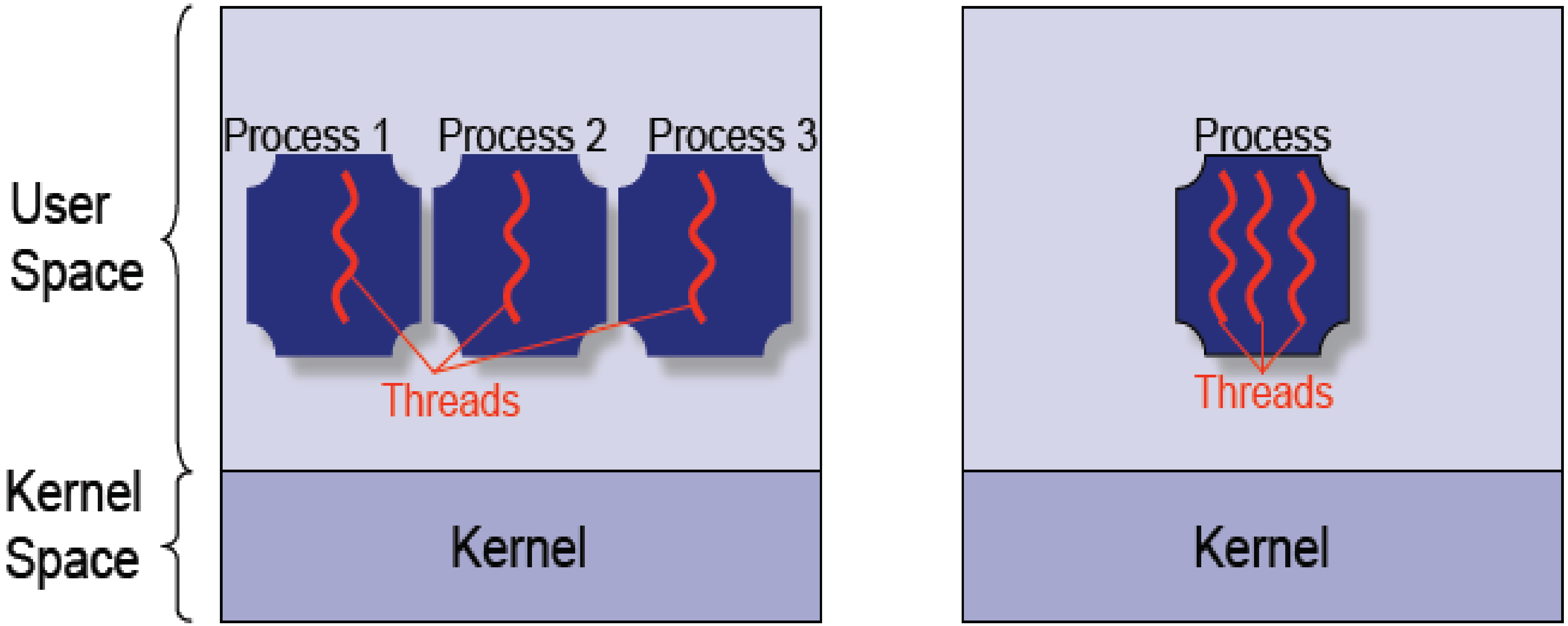


Creation of a new process using **fork** is *expensive* (time & memory).

Threads

- A thread (sometimes called a ***lightweight process***) does not require lots of memory or startup time.
- We can see processes as **unit of allocation**. A process groups resources together.
  - Resources, privileges, open files, allocated memory...
- We can see threads as **unit of execution**. They execute on allocated resources.
  - PC, SP, registers...
- Each process **may include many threads**
- Each thread **belongs to one process** (executes inside a process)

Threads



Each process has one memory space and one thread of execution

A process with one memory space and three threads of execution.

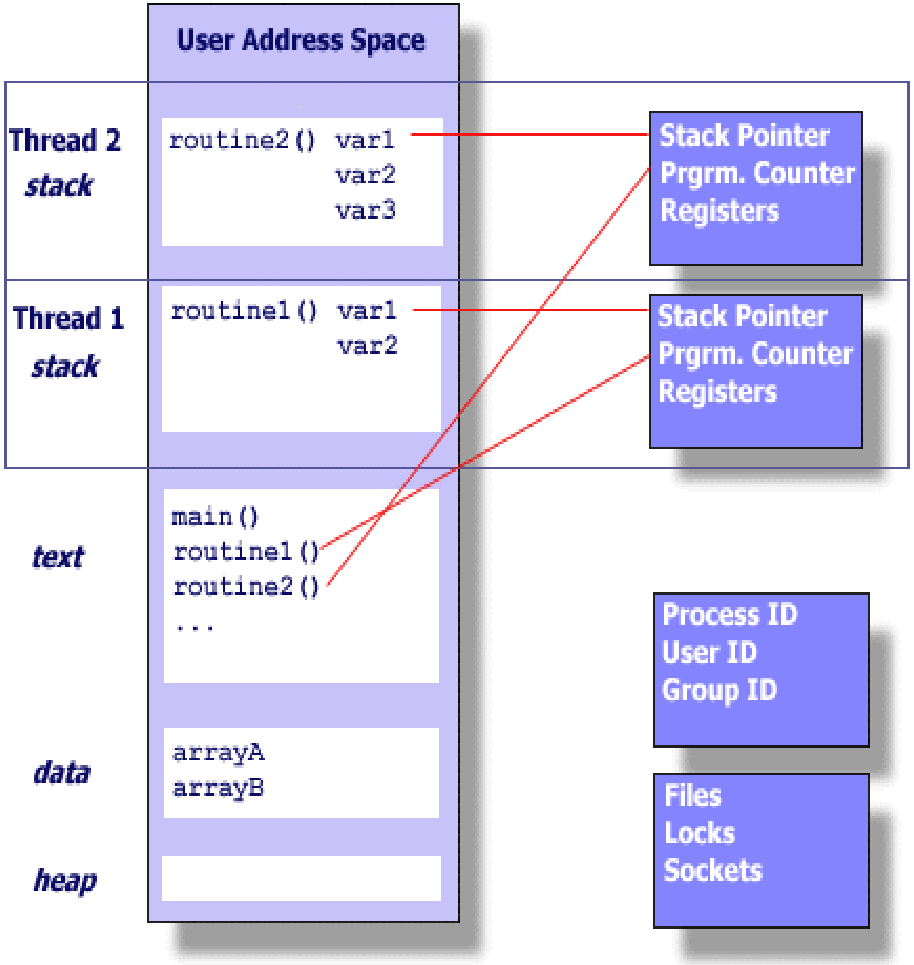
Thread-Specific Resources

Each thread has it's own **hardware execution state**:

- Thread ID (integer)
- Stack (SP), Registers, Program Counter (PC)

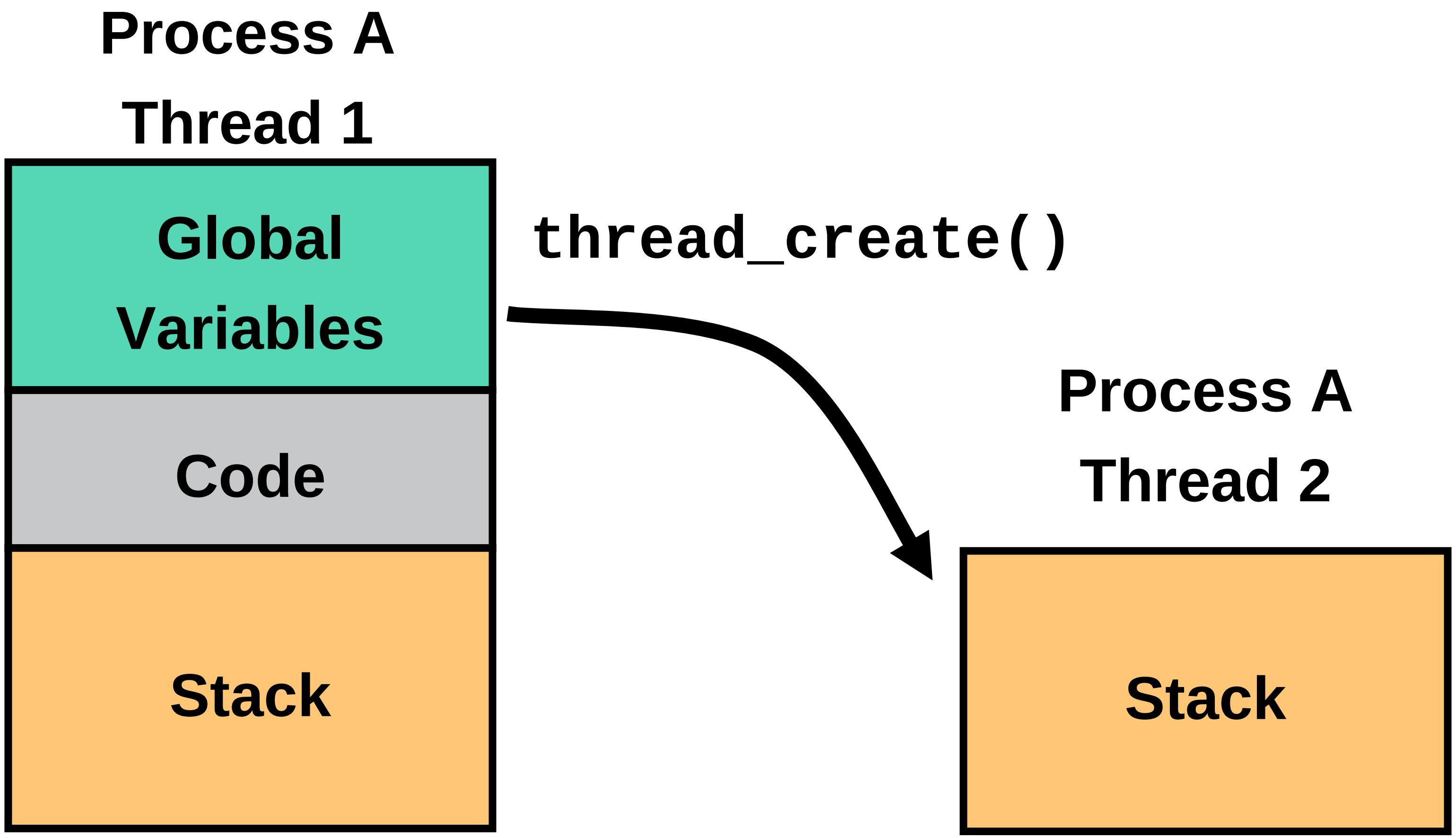
Each thread share following with **its process**

- Same code and data (**address space**)
- The same privilege
- The same resources
- Global variables
- Open files
- Signals and signal handlers ...



Threads share **the same memory (address space)** together with their process but have **separate execution context**.

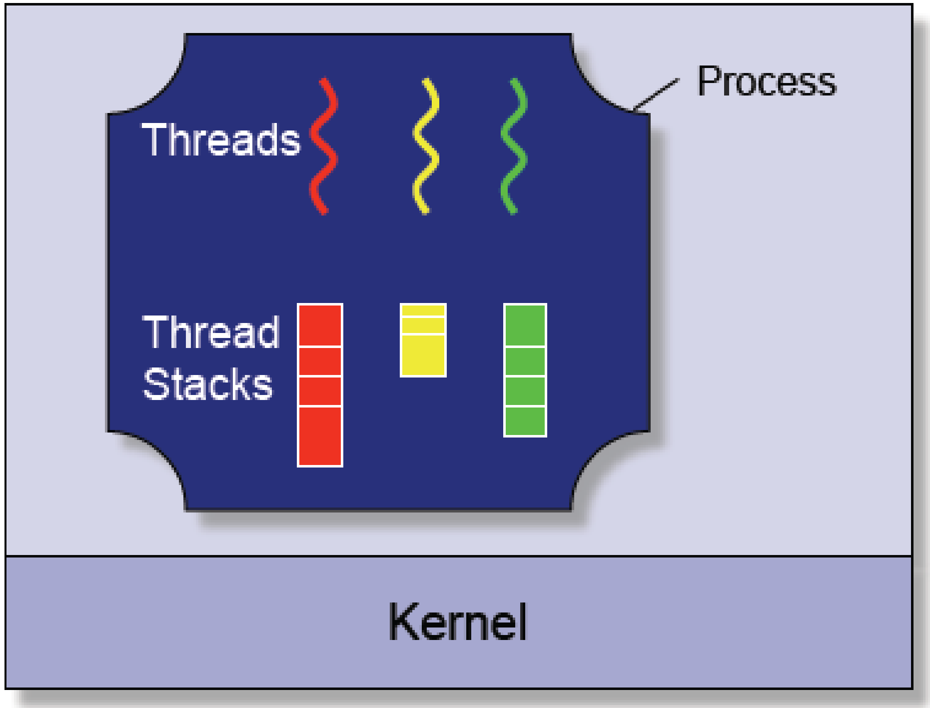
Thread Creation



Each thread has its own stack to store its local variables.

Multiple processes vs Multiple threads

- Multiple Threads
  - Creation
    - **Much faster and easy**
  - Context switching
    - **Much faster and easy**
  - Inter-thread Communication
    - **Much faster and easy**
  - Protection
    - **Threads are not protected from each other**



Why use threads?

• Because threads have minimal internal state, it takes less time to create a thread than a process (10x speedup in UNIX).
• It takes less time to terminate a thread.
• It takes less time to switch to a different thread.
• A multi-threaded process is much cheaper than multiple (redundant) processes.

Examples of Using Threads

• Threads are useful for any application with multiple tasks that can be run with separate threads of control.

• A Word processor may have separate threads for:

– User input

– Spell and grammar check

– displaying graphics

– document layout

• A web server may spawn a thread for each client

– Can serve clients concurrently with multiple threads.

– It takes less overhead to use multiple threads than to use multiple processes.

Examples of multithreaded programs

- **Most modern OS kernels**

– Internally concurrent because have to deal with concurrent requests by multiple users
– But no protection needed within kernel

- **Database Servers**

– Access to shared data by many concurrent users
– Also background utility processing must be done

- **Parallel Programming (More than one physical CPU)**

– Split program into multiple threads for parallelism. This is called Multiprocessing

What is a POSIX thread?

- An IEEE standardized way of using threads on a given system.
- Systems that support POSIX threads include:
  - Unix
  - Linux
  - Mac OS X
  - Windows (through Cygwin, etc)

POSIX Threads API

#include <pthread.h> → C/C++ programs

***Linux** - you need to link with "**-lpthread**" library*

```
gcc -lpthread main.c -o main.out
```

Some Posix Thread API functions:

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

void **pthread_exit**(void *retval);

int **pthread_join** (pthread_t tid, void **thread_return);

pthread_t **pthread_self** ();

Creating a New Thread

the thread handle gets stored here

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

returns a 0 if successful otherwise an errno!

the function to begin executing on this thread. It must take a void * as an argument and return a void *. When the function returns, the thread exits.

argument given to the thread function. usually a pointer to a struct or an array

thread attributes, usually NULL.

Getting Current Thread Handle (ID)

Occasionally, **pthread_self** useful for a sequence of code to determine which thread is running it.

```
pthread_t pthread_self ( );
```

returns thread handle of
the caller thread.

Terminating a Thread

A thread exits when the thread function returns. Alternatively, the thread can call the **pthread_exit** function.

```
void pthread_exit(void *retval);
```

this must point to data
that exists after the thread
terminates!

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

#define NUM_THREADS     5

void *HelloWorld(void *threadid)
{
   int tid;
   tid = (int)threadid;
   printf("Hello World! It's me, thread #%d!\n", tid);
   pthread_exit(0);
}

int main(int argc, char *argv[])
{
        pthread_t threads[NUM_THREADS];

        int i;

        for(i=0;i<NUM_THREADS;i++)
        {
                pthread_create(&threads[i], 0, HelloWorld, (void *)i);
        }

        pthread_exit(0);
        return 0;
}
```

Cast (void *) to integer (since void * is a pointer and pointer is nothing other than 32 bit integer value.)

Start of thread execution

Thread handles

Handle of the created thread

Address of start function

Argument to the function

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

#define NUM_THREADS     8

char *messages[NUM_THREADS];

void *HelloWorld(void *threadid)
{
	int id;

	sleep(1);
	id = (int) threadid;

	printf("Thread %d: %s\n", id, messages[id]);

	pthread_exit(0);
}
```

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

	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!";
	messages[5] = "Russian: Zdravstvytye, mir!";
	messages[6] = "Japan: Sekai e konnichiwa!";
	messages[7] = "Latin: Orbis, te saluto!";

	for(i=0;i<NUM_THREADS;i++)
	{
		pthread_create(&threads[i], NULL,
		HelloWorld, (void *)i);
	}

	pthread_exit(0);

	return 0;
}
```

Fill global data

Create threads

Global data

Access and print global data

Passing Data to Threads

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

The thread argument is of type **void*** . This allows us to pass a lot of data to a thread by passing a pointer to a struct or an array of data with casting it to the void * pointer.

```
struct data_t
{
   int  thread_id;
   int x;
};
...
struct data_t thread_data;
thread_data.thread_id = 0;
thread_data.x = 1;
...
```

```
pthread_create(th,NULL,start, (void *) &thread_data);
```

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

#define NUM_THREADS     8

char *messages[NUM_THREADS];

struct thread_data
{
   int  thread_id;
   int  sum;
   char *message;
};

struct thread_data thread_data_array[NUM_THREADS];
```

Structure for passing data to the thread

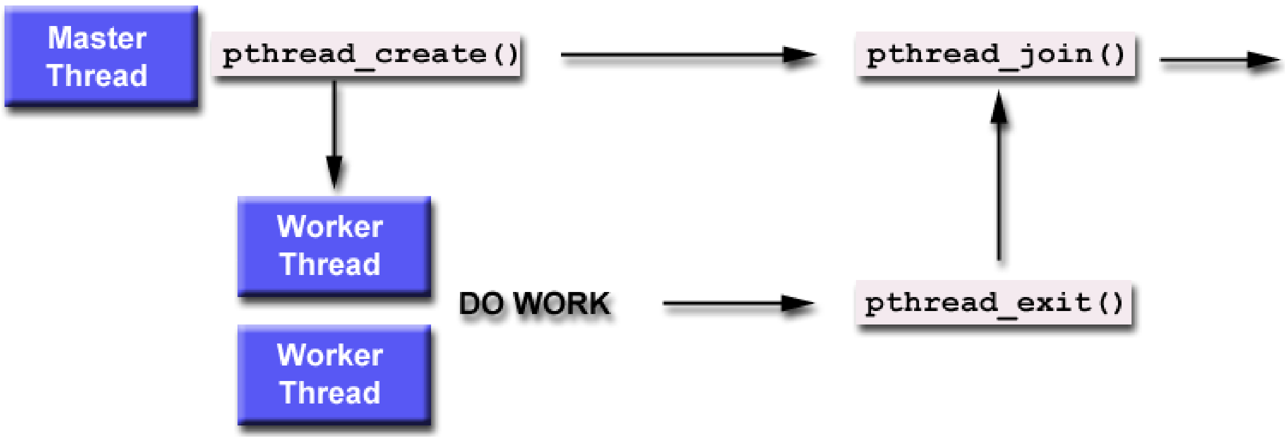
Global array that holds arguments to the threads

```
int main(int argc, char *argv[])
{
	pthread_t threads[NUM_THREADS];
	int i,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!";
	messages[5] = "Russian: Zdravstvytye, mir!";
	messages[6] = "Japan: Sekai e konnichiwa!";
	messages[7] = "Latin: Orbis, te saluto!";

	for(i=0;i<NUM_THREADS;i++)
	{
		sum = sum + i;
		thread_data_array[i].thread_id = i;
		thread_data_array[i].sum = sum;
		thread_data_array[i].message = messages[i];

		pthread_create(&threads[i], NULL, HelloWorld, (void *)&thread_data_array[i]);
	}

	pthread_exit(0);
}
```

Fill data structure to pass to the thread function (fill ith element of the global array)

Pass the ith element of the global array to the thread start function

Waiting for a Thread



int **pthread_join** (pthread_t tid, void **thread_return);

0 on success
or error code.

the thread to wait for

pointer to the return value.
NULL if there is nothing to return.

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

#define NUM_THREADS     3

void *BusyWork(void *null)
{
   int i;
   double result=0.0;
   for (i=0; i<1000000; i++)
   {
     result = result + (double)random();
   }
   printf("Thread result = %e\n",result);
   pthread_exit(0);
}
```

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

        for(i=0;i<NUM_THREADS;i++)
        {
                pthread_create(&thread[i],0, BusyWork, NULL);
        }

        for(i=0;i<NUM_THREADS;i++)
        {
                pthread_join(thread[i],0);
        }

        printf("Completed join with thread %d\n",i);

   pthread_exit(0);
}
```

Main thread is waiting other threads to finish

All created threads finished here

Create threads

pthread_yield Subroutine

- **Purpose**:
  Forces the calling thread to relinquish use of its processor

- **NOTE: Also is common to Use sched_yield, from unistd.h, instead.**

- **Library**:
  Threads Library (`libpthreads.a`)

- **Syntax**:
```
#include <pthread.h>
void pthread_yield ()
```

- **Description**:
  The pthread_yield subroutine forces the calling thread to relinquish use of its processor, and to wait in the run queue before it is scheduled again

  If the run queue is empty when the pthread_yield subroutine is called, the calling thread is immediately rescheduled.

Shared Global Variables

```
int counter=0;
....
void *start(void *arg) {
      counter++;
      printf("Thread %u is number %d\n", pthread_self(),counter);
}
main() {
      int i; pthread_t tid;
     for (i=0;i<10;i++)
        pthread_create(&tid,NULL,start,NULL);
}
```

Threads share all global variables

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

Threads must be **synchronized** to avoid problems on shared global variables.

Mutex

- ✓A synchronization variable that is used to synchronize access on shared memory regions. It has 2 **blocking** operations on it:
  - ✓Lock(**mutex**): an *atomic* operation that locks mutex. If the mutex is already locked, it blocks the caller thread.
  - ✓Unlock(**mutex**): an *atomic* operation that unlocks mutex. If there are any waiting threads, this functions unblocks one of them.

- ✓Permit only one thread to execute a section of the code that access shared variables (shared memory regions : critical sections)
- ✓The general idea is to ***lock*** it before accessing global variables and to ***unlock*** as soon as we are done.

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 1000
```

```
mutex = 1
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 1000
```

```
mutex = 1
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 1000
```

```
mutex = 0
```

Mutex

A1 statement	B1 statement
A2 mutex.wait()	B2 mutex.wait()
A3 wolski.balance = wolski balance - 200	B3 wolski.balance = wolski balance - 200
A4 mutex.signal()	B4 mutex.signal()
A5 statement	B5 statement

```
balance = 800
```

```
mutex = 0
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 800
```

```
mutex = 0
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 800
mutex = -1
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 800
```

```
mutex = -1
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 800
```

```
mutex = 0
```

Mutex

A1 statement	B1 statement
A2 mutex.wait()	B2 mutex.wait()
A3 wolski.balance = wolski balance - 200	B3 wolski.balance = wolski balance - 200
A4 mutex.signal()	B4 mutex.signal()
A5 statement	B5 statement

`balance = 800`

`mutex = 0`

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 800
```

```
mutex = 0
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 600
```

```
mutex = 0
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

```
balance = 600
```

```
mutex = 1
```

Mutex

A1 statement
A2 mutex.wait()
A3 wolski.balance =
wolski balance - 200
A4 mutex.signal()
A5 statement

B1 statement
B2 mutex.wait()
B3 wolski.balance =
wolski balance - 200
B4 mutex.signal()
B5 statement

`balance = 600`

`mutex = 1`

Pthread Mutex API

- **pthreads** includes support for *Mutual Exclusion* primitives that can be used to synchronize threads.

Posix Mutex API functions:

int **pthread_mutex_init**(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);

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

int **pthread_mutex_unlock**(pthread_mutex_t *mutex);

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

Global mutex variable that will be used for synchronization

```
#incluc
#incluc
#incluc

pthread_mutex_t mutex;
int x = 0;
```

Shared global variable

```
void *MyThread(void *arg)
{
   char *sbName;

   sbName = (char *)arg
   pthread_mutex_lock(&mutex);
         x = x + 1;
  pthread_mutex_unlock(&mutex);
   printf("X = %d in Thread %s\n", x, sbName);
}
```

Lock mutex before changing global variable. This will block the caller thread if the mutex is locked.

Unlock mutex after changing global variable

```
int main()
{
   pthread_t idA, idB;

   if (pthread_mutex_init(&mutex, NULL) < 0)
   {
       perror("pthread_mutex_init");
       exit(1);
   }
   if (pthread_create(&idA, NULL, MyThread, (void *)"A") != 0)
   {
     perror("pthread_create");
     exit(1);
   }

   if (pthread_create(&idB, NULL, MyThread, (void *)"B") != 0)
   {
      perror("pthread_create");
      exit(1);
    }
   pthread_join(idA, NULL);
   pthread_join(idB, NULL);

   pthread_mutex_destroy(&mutex);
}
```

Initialize mutex before using it

Destroy mutex after using it

Semaphores

- ✓ A synchronization variable that takes on positive integer values. It has 2 **blocking** operations on it:
  - ✓ Wait(**semaphore**): an *atomic* operation that waits for semaphore to become positive, then **decrements** it by 1. (blocks caller if semaphore is negative)
  - ✓ Signal(**semaphore**): an *atomic* operation that **increments** semaphore by 1. (if the value of the semafore is negative, this function unblocks one of the waiting threads)

- ✓ Permit a limited number of threads to execute a section of the code
- ✓ Similar to mutexes
- ✓ Generally used for signalling existence of an event or action

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.
.
.
A1 statement
A2 sem.signal()
.
.
.

.
.
.
B1 sem.wait()
B2 statement
.
.
.

sem = 0

Signalling

.
.
.
A1 statement
A2 sem.signal()
.
.
.

.
.
.
B1 sem.wait()
B2 statement
.
.
.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = -1

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = -1

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = -1

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

Signalling

.	.
.	.
.	.
A1 statement	B1 sem.wait()
A2 sem.signal()	B2 statement
.	.
.	.
.	.

sem = 0

#include < semaphore.h> → C/C++ programs

Semaphore API functions:

int **sem_init**(sem_t *sem, int pshared, unsigned int value);

Semaphore object

Generally 0

Gives an initial value to the semaphore

int **sem_post**(sem_t *sem);

atomically increases the value of a semaphore by 1

int **sem_wait**(sem_t *sem);

atomically decreases the value of a semaphore by 1

int **sem_destroy**(sem_t *sem);

Frees resources allocated for semaphore

```
#include <pthread.h>
#include <semaphore.h>
void *thread_function( void *arg );
sem_t semaphore;
int main()
{
      int tmp;
      tmp = sem_init( &semaphore, 0, 0 );
      pthread_create( &thread[i], NULL, thread_function, NULL );
      while ( still_has_something_to_do() )
      {
        sem_post( &semaphore );
      }
      pthread_join( thread[i], NULL );
      sem_destroy( &semaphore );
      return 0;
}
```

Global semaphore variable

initialize semaphore

signal semaphore

destroy semaphore

```
void *thread_function( void *arg )
{
  sem_wait( &semaphore );
    ...
  pthread_exit( NULL );
}
```

wait semaphore

```c
#include <stdio.h>      /* Input/Output */
#include <stdlib.h>     /* General Utilities */
#include <pthread.h>    /* POSIX Threads */
#include <semaphore.h>  /* Semaphore */

#define BUFFER_SIZE 5

int headPos = BUFFER_SIZE - 1;
int endPos = 0;
int Buffer[BUFFER_SIZE];

sem_t producerSem;
sem_t consumerSem;
pthread_mutex_t mutex;
```

Global buffer to store produced values

Mutex that will be used to access shared Buffer

```c
int main()
{
    int i[2];
    pthread_t thread_a;
    pthread_t thread_b;

    pthread_mutex_init(&mutex, NULL);

    pthread_create (&thread_a, 0, producer, (void *)0);
    pthread_create (&thread_b, 0, consumer, (void *)0);

    pthread_join(thread_a, NULL);
    pthread_join(thread_b, NULL);

    /* exit */
    exit(0);
}
```

Init mutex that will be used to access shared Buffer

Producer semaphore used for the notification from consumer thread

Consumer semaphore used for the notification from producer thread

```
void *producer(void *arg)
{
   int itemCount = 10;

   sem_init(&producerSem, 1,BUFFER_SIZE);

   while(itemCount)
   {
      sem_wait(&producerSem);

      pthread_mutex_lock(&mutex);

      Buffer[endPos] = 1;

      printf("Producer Thread : Buffer[%d] = 1 \n",endPos);
      endPos =  (endPos + 1) % BUFFER_SIZE;
      itemCount--;

      pthread_mutex_unlock(&mutex);

      sem_post(&consumerSem);
   }

   pthread_exit(0);


   return 0;
}
```

Init semaphore with BUFFER_SIZE

Wait notification from consumer thread

Enter critical section

Access shared memory and produce value

Exit critical section

Notify consumer thread

```
void *consumer(void *arg)
{
   int itemCount = 10;

   sem_init(&producerSem, 0,0);

   while(itemCount)
   {
      sem_wait(&consumerSem);

      pthread_mutex_lock(&mutex);

      Buffer[headPos] = 0;

      printf("Consumer Thread : Buffer[%d] = 0 \n",headPos);
      headPos =  (headPos + 1) % BUFFER_SIZE;
      itemCount--;

      pthread_mutex_unlock(&mutex);

      sem_post(&producerSem);
   }

   pthread_exit(0);

   return 0;
}
```

İnit semaphore with 0

Wait notification from consumer thread

Enter critical section

Access shared memory and consume value

Exit critical section

Notify producer thread

References

- https://computing.llnl.gov/tutorials/pthreads/
- http://students.cs.byu.edu/~cs345ta/