PThreads

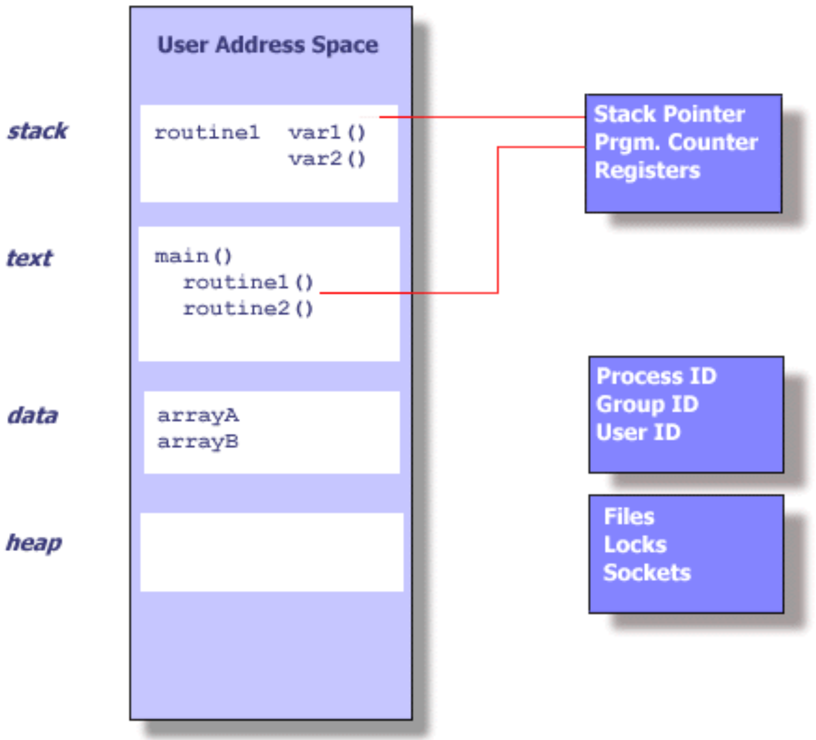
Thanks to Professor Sam Midkiff for providing his ECE 563 slides from which these slides are derived.

https://engineering.purdue.edu/~smidkiff/ece563/slides/PThreads.pdf

Professor Midkiff's slides were derived from the LLNL tutorial. For a great resource on pthreads see: https://computing.llnl.gov/tutorials/pthreads/

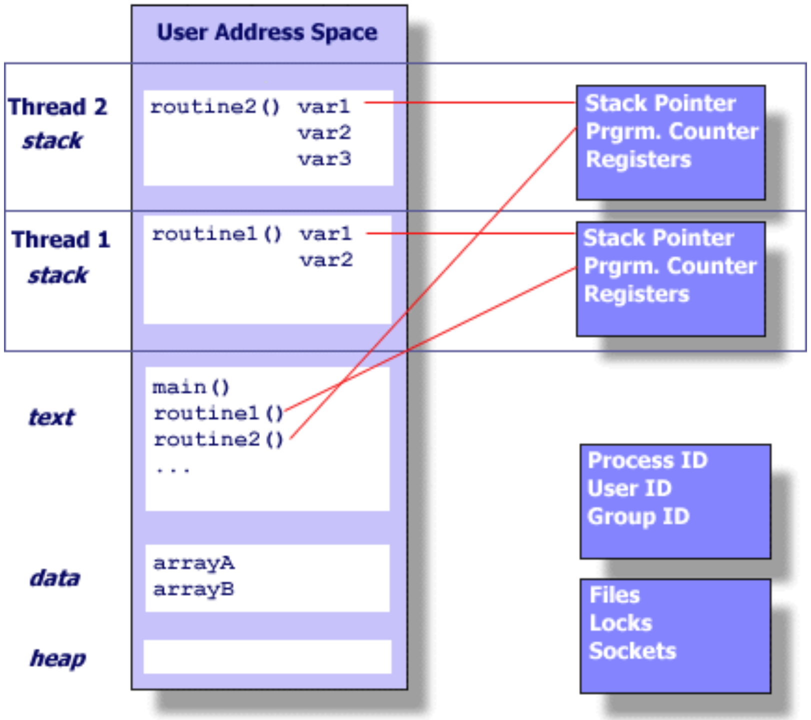
Processes and threads

- Understanding what a thread means knowing th relationship between a process and a thread. A process is created by the operating system.
  - Processes contain information about progra resources and program execution state, including:
    - Process ID, process group ID, user ID, and group ID, address space
    - Environment, working directory
    - Program instructions, registers, stack, heap
    - File descriptors, inter-process communication tools (such as message queues, pipes, semaphores, or shared memory), signal actions
    - Shared libraries



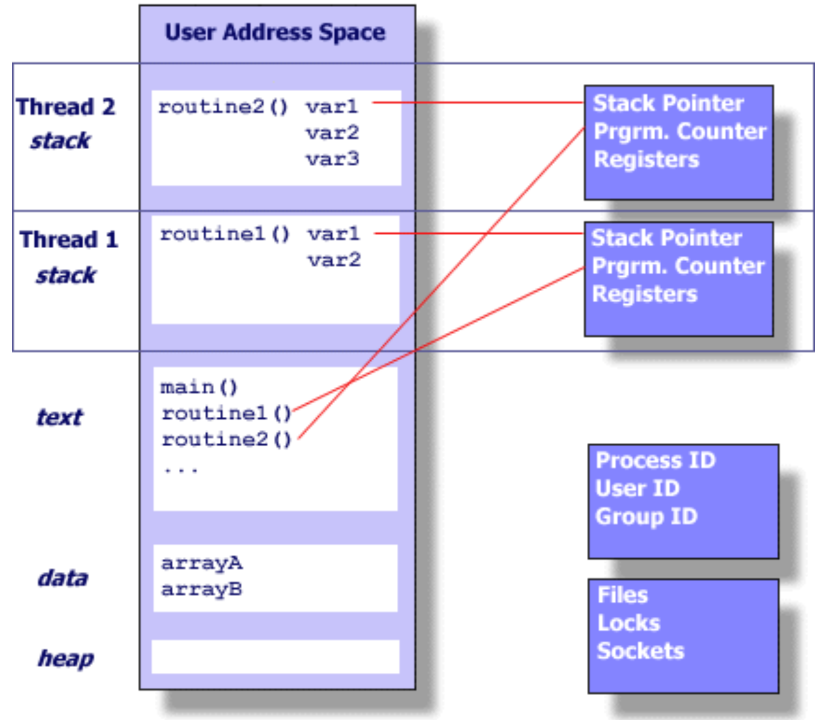
Processes and threads, cont.

- Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities within a process



Processes and threads, cont.

- A thread can possess an independent flow of control and be schedulable because it maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.



Processes and threads, cont.

- A process can have multiple threads, all of which share the resources within a process and all of which execute within the same address space
- Within a multi-threaded program, there are at any time multiple points of execution

Processes and threads, cont.

- Because threads within the same process 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
  - Reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer

What are Pthreads?

- Historically, hardware vendors implemented their own proprietary versions of threads.
  - Standardization required for portable multi-threaded programming
  - For Unix, this interface specified by the IEEE POSIX 1003.1c standard (1995).
    - Implementations of this standard are called POSIX threads, or Pthreads.
    - Most hardware vendors now offer Pthreads in addition to their proprietary API's
    - Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library

Creating threads

- **pthread_create** (thread, attr, start_routine, arg)
- This routine creates a new thread and makes it executable. Typically, threads are first created from within main() inside a single process.
  - *start_routine* is the C routine that the thread will execute once it is created. A single argument may be passed to *start_routine* via *arg* as a void pointer.
  - The *attr* parameter is used to set thread attributes. Can be an object, or NULL for the default values. (For Lab2, you only need to set to NULL)
  - The maximum number of threads that may be created by a process is implementation dependent.

Terminating threads

- Multiple methods in general. For our purposes:
  - Recommended for Lab 2: The thread returns from its starting routine (the main routine for the initial thread)
  - Another option: The thread makes a call to the pthread_exit subroutine
    - Returning from starting routine is equivalent to calling pthread_exit with the value supplied in the return statement

- **pthread_join** (threadId, status)
- The pthread_join() subroutine blocks the calling thread until the specified *threadId* thread terminates
- The programmer is able to obtain the target thread's termination return status if specified in: pthread_exit(), in the *status* parameter
  - Or value when returning from the thread’s starting_routine

Protecting access to memory locations shared by threads

Example code:

Global variable:
"balance": how much money left in a bank account

Two threads:
- Thread 1 is withdrawing money
- Thread 2 is depositing money

Operations occur concurrently.
Both threads access the global variable "balance"
Need to be careful to protect accesses to the variable.

Protecting access to memory locations shared by threads

Global variable: “balance”: how much money left in a bank account
Two separate threads execute withdrawalFn, and depositFn

```
void * withdrawalFn(void * arg){
        int curr_balance = balance;
        curr_balance = curr_balance - withdrawal_amount;
        sleep(1);
        balance = curr_balance;

        return NULL;
}
void * depositFn(void * arg){
        int curr_balance = balance;
        curr_balance = curr_balance + deposit_amount;
        sleep(1);
        balance = curr_balance;

        return NULL;
}
```

Example race condition that can occur

Thread 1 (deposit)	Thread 2 (withdraw)	Balance
Read balance: $1000		**$1000**
	Read balance: $1000	**$1000**
	Withdraw $200	**$1000**
Deposit $200		**$1000**
Update balance $1000+$200		**$1200**
	Update balance $1000-$200	**$800**

Using locks

- **pthread_mutex_lock** (mutex)
  - Acquire lock if available
  - Otherwise wait until lock is available
- pthread_mutex_unlock (mutex)
  - Release the lock to be acquired by another pthread_mutex_lock call
  - Cannot make assumptions about which thread aquire the lock next

Protecting access to shared variables using locks

```
void * withdrawalFnLock(void * arg){
        pthread_mutex_lock(&lock);
        int curr_balance = balance;
        curr_balance = curr_balance - withdrawal_amount;
        sleep(1);
        balance = curr_balance;
        pthread_mutex_unlock(&lock);

        return NULL;
}
void * depositFnLock(void * arg){
       pthread_mutex_lock(&lock);
        int curr_balance = balance;
        curr_balance = curr_balance + deposit_amount;
       sleep(1);
        balance = curr_balance;
        pthread_mutex_unlock(&lock);

        return NULL;
}
```

Example execution with locks

Thread 1 (deposit)	Thread 2 (withdraw)	Balance
	GetLock()	**$1000**
	Read balance: $1000	**$1000**
	Withdraw $200	**$1000**
	Update balance $1000-$200	**$800**
	ReleaseLock()	**$800**
GetLock()		$800
Read balance: $800		**$800**
Deposit $200		$800
Update balance $800+$200		$1000
ReleaseLock()		$1000

Creating threads

- **pthread_create** (thread, attr, start_routine, arg)
- This routine creates a new thread and makes it executable. Typically, threads are first created from within main() inside a single process.
  - *start_routine* is the C routine that the thread will execute once it is created. A single argument may be passed to *start_routine* via *arg* as a void pointer.
  - The *attr* parameter is used to set thread attributes. Can be an object, or NULL for the default values. (For Lab2, you only need to set to NULL)
  - The maximum number of threads that may be created by a process is implementation dependent.

time

Thread 0	Thread k
t = 0;	
pthread_create(..., f, t);	
t = 1	*thread spawn*
	f(t);
pthread_create(..., f, t);	
	x = t;
t = 2	

What is the value of t that is used in this call to f?
The value is indeterminate.

Passing arguments to a thread

- Thread startup is non-deterministic
- It is implementation dependent
- If we do not know when a thread will start, how do we pass data to the thread knowing it will have the right value at startup time?
  - Don’t pass data as arguments that can be changed by another thread
  - In general, use a separate instance of a data structure for each thread.

Passing data to a thread (a simple integer)

```
int *taskids[NUM_THREADS];
for(t=0;t < NUM_THREADS;t++) {
   taskids[t] = (int *)
                 malloc(sizeof(int));
   *taskids[t] = t;
   printf("Creating thread %d\n", t);
   rc = pthread_create(&threads[t], NULL,
                       PrintHello,
                       (void *) &t);
   …
}
```