CSCI 1730 Systems Programming

Threads, and

other IPC:

Shared Memory, and Message Queus

Threads: Questions

- How is a thread different from a process?
- Why are threads useful?
- How can POSIX threads be useful?
- What are problems with threads?

- Resources:
- https://computing.llnl.gov/tutorials/pthreads/

Review: What is a Process?

A process is a program in execution…

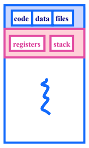
A thread have

(1) an execution stream and

(2) a context

- **Execution stream**
  - » stream of instructions
  - » sequential sequence of instructions
  - » 1“thread” of control
- **Process ‘context’ (Review)**
  - » Everything needed to run (restart) the process …
  - » **Registers**
    - – program counter, stack pointer, general purpose…
  - » **Address space**
    - – Everything the process can access in memory
    - – Heap, stack, code

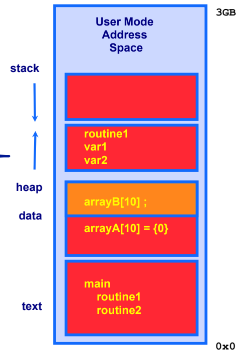
Running on a thread



```
{nike:maria:27} size task2
   text    data     bss     dec     hex filename
   1040     484      16    1540     604 task2
```

Review: What Makes up a Process?

- **Program code (text)**
  - » Compiled version of the text
- **Data (cannot be shared)**
  - » global variables
    - – Uninitialized (BSS segment) sometimes listed separately.
    - – Initialized
- **Process stack (scopes)**
  - » function parameters
  - » return addresses
  - » local variables and functions
- **<<Shared Libraries >>**
- **Heap: Dynamic memory (alloc)**
- **OS Resources, environment**
  - » open files, sockets
  - » Credential for security
- **Registers**
  - » program counter, stack pointer



address space are the shared resources of a(ll) thread(s) in a program

What are are problem’s with processes?

- **How do processes (*independent* memory space) *communicate*?**
  - » **Complicated/Not really that simple (seen it, tried it – and you have too):**
    - – Message Passing:
      - Remote machine (send and receive): Sockets
      - Local machine via message queues
        - » http://beej.us/guide/bgipc/output/html/multipage/mq.html
    - – Pipes
    - – Signal
    - – Shared Memory: Set up a shared memory area
      - http://beej.us/guide/bgipc/output/html/multipage/shm.html
  - » **Slow/Overhead: All of the methods above add some kernel overhead lowering performance**
    - – Process Creation is heavy weight
      - Allocate space/heavy weight

Processes versus Threads

Solution: A thread is a “lightweight process” (LWP)

- **An execution stream that shares an address space**
  - » Overcome data flow over a file descriptor
  - » Overcome setting up `tighter memory’ space
- **Multiple threads within a single process**

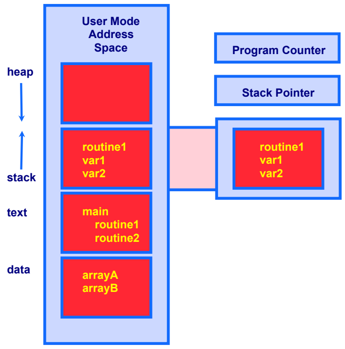
```
main()
    {
    i = 55;
    fork();
    // what is i
```

Examples:

- **Two processes (copies of each other) examining memory address `0xffe84264` see different values (i.e., different contents)**
  - » same frame of reference
- **Two threads examining memory address `0xffe84264` see same value (i.e., same contents)**
- **Illustrate: `ctest/i-threading.c, ctest/i-process.c`**

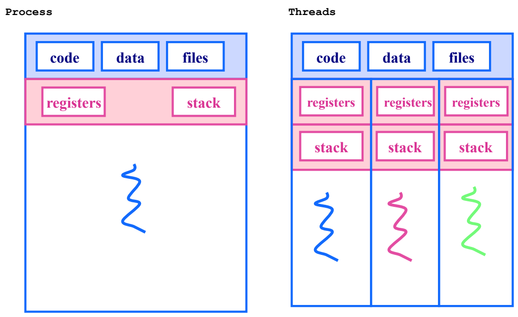
What Makes up a Thread?

- Own stack (necessary?)
- Own registers (necessary?)
  - » Own program counter
  - » Own stack pointer
- State (running, sleeping)
- Signal mask



address space are the shared resources of a(ll) thread(s) in a program

Single and Multithreaded Process



Why Support Threads?

- *Divide large task across several cooperative threads*
- Multi-threaded task has many performance benefits

- Examples:
  - » Web Server: create threads to:
    - – Get network message from client
    - – Get URL data from disk
    - – Compose response
    - – Send a response
  - » Word processor: create threads to:
    - – Display graphics
    - – Read keystrokes from users
    - – Perform spelling and grammar checking in background

Why Threads instead of a Processes?

- Advantages of Threads:
  - » Thread operations cheaper than corresponding process operations
    - – In terms of: Creation, termination, (context) switching
  - » IPC cheap through shared memory
    - – No need to invoke kernel to communicate between threads
- Disadvantages of Threads:
  - » True Concurrent programming is a challenge (what does this mean? True concurrency?)
  - » Synchronization between threads needed to use shared variables (more on this later – this is HARD).

Why are Threads Challenging? `pthread1` Example: Output?

```
main()
{
   pthread_t t1, t2;
   char *msg1 = “Thread 1”; char *msg2 = “Thread 2”;
   int ret1, ret2;
   ret1 = pthread_create( &t1, NULL, print_fn, (void *) msg1 );
   ret2 = pthread_create( &t2, NULL, print_fn, (void *) msg2 );
   if( ret1 || ret2 )
   {
        fprintf(stderr, “ERROR: pthread_created failed.\n”);
        exit(1);
   }
   pthread_join( t1, NULL );
   pthread_join( t2, NULL );
   printf( “Thread 1 and thread 2 complete.\n” );
}
void print_fn(void *ptr)
{
   printf(“%s\n”, (char *)ptr);
}
```

Why are Threads Challenging?

- Example: Transfer `$50.00` between two accounts and output the total balance of the accounts:

`M` = Balance in Maria’s account (begin `$100`)

`T` = Balance in Tucker’s account (begin `$50`)

`B` = Total balance

- Tasks:

```
T = 50, M = 100
M = M - $50.00
T = T + $50.00
B = M + T
```

```
Idea: on distributing
the  tasks:
(1)One thread debits
  and credits
(2) The other Totals
Does that work?
```

Why are Threads Challenging?

- Tasks:

```
T = 50, M = 100
M = M - $50.00   } One thread debits
T = T + $50.00   } & credits
B = M + T        } One thread totals
```

M = M - $50.00	M = M - $50.00	B = M + T
T = T + $50.00	B = M + T	M = M - $50.00
B = M + T	T = T + $50.00	T = T + $50.00
B = $150	B = $100	B = $150

Common Programming Models

- Manager/worker
  - » Single manager handles input and assigns work to the worker threads
- Producer/consumer
  - » Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- Pipeline
  - » Task is divided into series of subtasks, each of which is handled in series by a different thread

Thread Support

- Three approaches to provide thread support
  - » User-level threads (Pthreads)
  - » Kernel-level threads (not cover)
    - – Kernel manages the threads (avoids blocking)
  - » Hybrids

Latencies

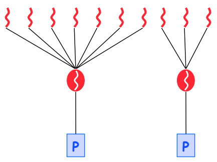
- Comparing user-level threads, kernel threads, and processes
- Thread/Process Creation Cost:
  - » Evaluate –with Null fork: the time to create, schedule, execute, and complete the entity that invokes the null procedure
- Thread/Process Synchronization Cost:
  - » Evaluate – with Signal-Wait: the time for an entity to signal a waiting entity and then wait on a condition (overhead of synchronization)

Procedure call = 7 us Kernel Trap = 17 us	User Level Threads	Kernel Level Threads	Processes
Null fork	34	948	11,300
Signal-wait	37	441	1,840

30X,12X

User-Level Threads

- Many-to-one thread mapping
  - » Implemented by user-level runtime libraries
    - – Create, schedule, synchronize threads at user-level, state in user level space
  - » OS is not aware of user-level threads
    - – OS thinks each process contains only a single thread of control



- Advantages
  - » Does not require OS support; Portable
  - » Can tune scheduling policy to meet application (user level) demands
  - » Lower overhead thread operations since no system calls
- Disadvantages
  - » Cannot leverage multiprocessors (no true parallelism)
  - » Entire process blocks when one thread blocks

POSIX Pthreads

- P-threads is a standard set of C library functions for multithreaded programming
  - » IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
- Programs must include the file `pthread.h`
- Programs must be linked with the pthread library (`-lpthread`)
  - » Done by default by some gcc's (e.g., on Mac OS X)

Pthread: Code Base

The subroutines which comprise the Pthreads API can be informally grouped into Two major groups:

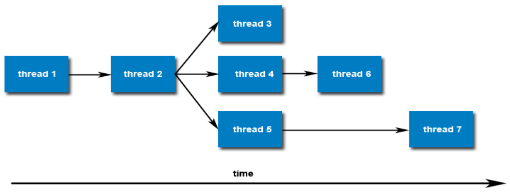
- **Thread management**: Routines that work directly on threads
- **Synchronization**:
  - **Mutexes**: Routines that deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion”
  - **Locks**: Routines that manage read/write locks and barriers
  - **Condition variables**: Routines that address communications between threads that share a mutex.

Thread Management

- Creating and Terminating Threads
- Passing Arguments to Threads
- Joining and Detaching Threads
- Setting Thread Attributes
- Miscellaneous Routines

Creating and Terminating Threads

- `pthread_create(thread,attr,start_routine,arg)`
- `pthread_exit(status)`
- pthread_join(tid, 0);



- Initially, main() has a single thread
- New threads created via pthread_create
  - Max # threads are platform dependent
- Threads are peers, and may create other threads.
  - No implied hierarchy or dependency between threads.

Pthread Create

```
#include <pthread.h>

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

- **thr**: Will contain the newly created thread’s id. Must be passed by reference
- **attr**: Give the attributes that this thread will have. Use NULL for the default ones.
- **start_routine**: The name of the function that the thread will run. Must have a void pointer as its return and parameters values
- **arg**: The argument for the function that will be the body of the Pthreads

Pointers of the type void can reference *ANY* type of data, but they *CANNOT* be used in any type of operations that reads or writes its data without a cast

Terminating Threads

- thread returns from its starting routine
- Thread calls pthread_exit()
- Thread is canceled by another thread via the pthread_cancel routine.
- The entire process is terminated due to a call to either the exec or exit subroutines.
- main() finishes … before the threads that it created.

- *void* pthread_exit(void **arg*);
  - » This function will indicate the end of a pthread and the returning value will be put in *arg*
- *pthread_t* pthread_self(*void*)
  - » Returns the id of the calling thread. Returns a pthread_t type which is usually an integer type variable

"Hello World" Example

```
#include <pthread.h>          // +stlib, stio.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);
       }  }
pthread_exit(NULL); /* Last thing that main() should do */
}
```

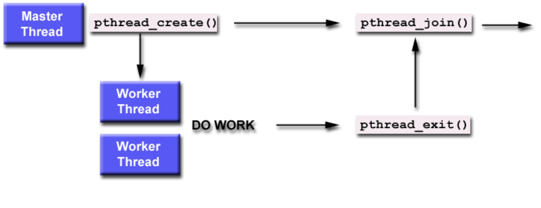
```
{ingrid:547}  01-hello
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #0!
Hello World! It's me, thread #1!
In main: creating thread 4
Hello World! It's me, thread #2!
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

Passing Arguments to Threads

- ***Single Argument Passing***
  - » Cast its value as a void * (a tricky pass by value)
  - » Cast its address as a void pointer (pass by reference).
    - – The value that the address is pointing should NOT change between Pthreads creation
- ***Multiple Argument Passing***
  - » Heterogonous: Create an structure with all the desired arguments and pass an element of that structure as a void pointer.
  - » Homogenous: Create an array and then cast it as a void pointer

- 02-hello_arg1.c (single argument)
- 02-hello_arg2.c (struct non-homogenous)

Thread Joining



- Analogous to wait()
- "Coarse grained" synchronization b/c threads.
- Blocks calling thread until the thread with "id" terminates.
- A joining thread can match one pthread_join() call.
  - – It is a logical error to attempt multiple joins on the same thread.
- A thread is either joinable or detached (can never be joined).

Joinable or Detached?

- If a thread requires joining, consider explicitly creating it as joinable.
  - » This provides portability as not all implementations may create threads as joinable by default.
- If you know in advance that a thread will never need to join with another thread,
  - » consider creating it in a detached state.
  - » Some system resources may be able to be freed.

Example

- i-threading.c

Synchronization

```
int pthread_mutex_init(pthread_mutex_t * mutex,
                              pthread_mutexattr_t *attr);
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

- Stands for Mutual Exclusion
- Serializes access to some critical region of code or data
- Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.

Synchronization

- http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html#SYNCHRONIZATION
  - » Locks. Mutex.

- *void* pthread_yield ()

Processes vs. Threads

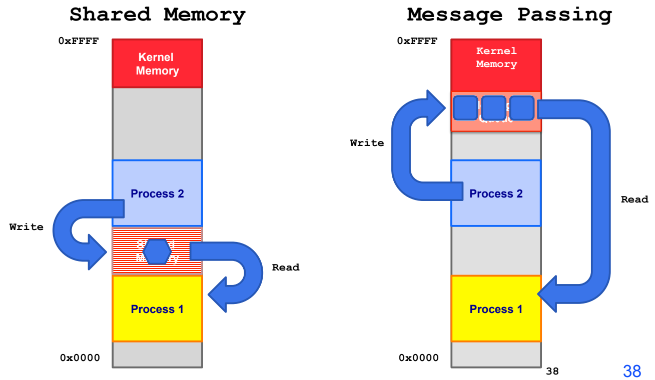
- Threads are better if:
  - » You need to create new ones quickly, on-the-fly
  - » You need to share lots of state
- Processes are better if:
  - » You want protection
    - – One process that crashes or freezes doesn’t impact the others
  - » You need high security
    - – Only way to move state is through well-defined, sanitized message passing interface

- https://computing.llnl.gov/tutorials/pthreads/#CreatingThreads

Design: Threading Issues: fork() & exec()

- fork()
  - Duplicate all threads?
  - Duplicate only the thread that performs the fork
  - Resulting new process is single threaded?
  - -> solution provide two different forks (mfork)
- exec()
  - Replaces the process - including all threads?
  - If exec is after fork then replacing all threads is unnecessary.

Other IPC Mechanisms



IPC: Shared Memory

- Processes
  - Each process has private address space
  - Explicitly set up shared memory segment within each address space
- Threads
  - Always share address space (use heap for shared data), don't need to set up shared space already there.
- Advantages
  - Fast and easy to share data
- Disadvantages
  - Must *synchronize* data accesses; error prone (later)

ex-mem.c

Shared Memory API

- shmget () – creates, allocate a shared memory page
- shmat() – map the memory page into the processes address space
  - Now you can read/write the page using a pointer
- shmdt () – remove/detaches a shared page
  - Processes with open references may still access the page
- shmctl() – ipc control, destroy it.

POSIX Shared Memory

- A variation of mapped memory.
- Uses shm_open() to open the shared memory object (instead of calling open()) and
- shm_unlink() to close and delete the object (instead of calling close() which does not remove the object).
- The options in shm_open() are substantially fewer than the number of options provided in open().

IPC: Message Passing (also for threads, similar to processes)

- Message passing most commonly used between processes
  - Explicitly pass data between sender (src) + receiver (destination)
  - Example: Unix pipes, Message Queues
- Advantages:
  - Makes sharing explicit
  - Improves modularity (narrow interface)
  - Does not require trust between sender and receiver
- Disadvantages:
  - Performance overhead to copy messages
- Issues:
  - How to name source and destination?
    - One process, set of processes, or mailbox (port)
  - Does sending process wait (I.e., block) for receiver?
    - Blocking: Slows down sender
    - Non-blocking: Requires buffering between sender and receiver

- OpenMP
- Pthreads:
  - » http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html
  - » https://computing.llnl.gov/tutorials/pthreads/
  - » http://www.dirjournal.com/library/posix-threads.php
  - » https://www.sourceware.org/gdb/current/onlinedocs/gdb/Threads.html#Threads
  - » http://www.mit.edu/people/proven/IAP_2000/index.html
- Advanced IPC (Shared Memory, Message Queues, Memory Mapped Files)
  - » http://beej.us/guide/bgipc/output/html/singlepage/bgipc.html