Threads

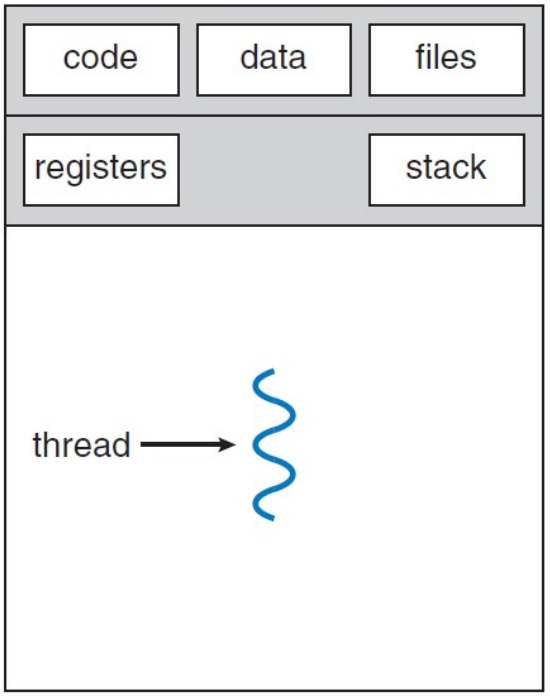
POSIX ref.: http://pubs.opengroup.org/onlinepubs/9699919799/

Threads

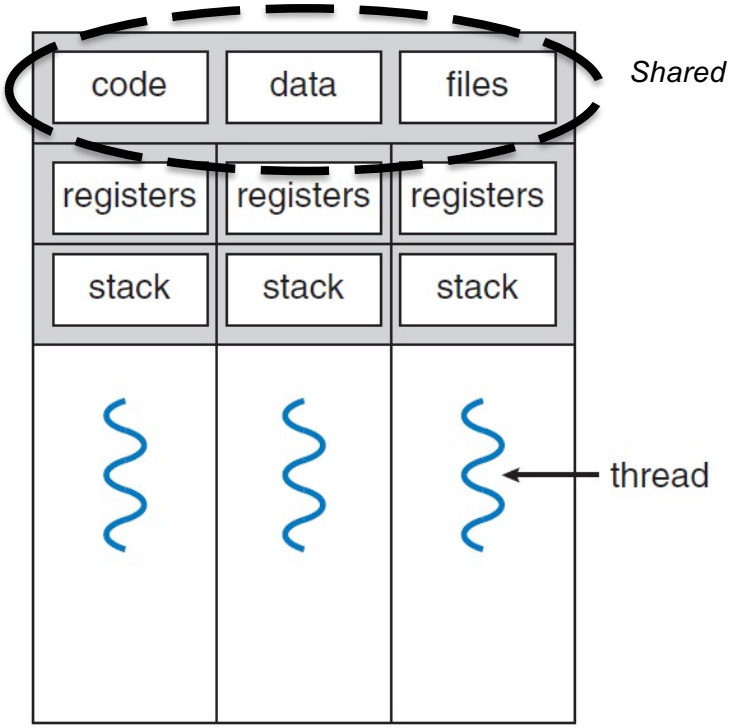
- Threads share heap, data segment, text segment and file descriptor table but have separate stacks and CPU contexts.
- Each thread have a private program counter and threads executes concurrently.
- Depending on how threads are implemented, the CPU contexts can be stored in user space or kernel space.

Threads in processes

- A thread is a basic unit of CPU utilization



Single-threaded Process



Multithreaded Process

Threads vs. Processes

How are threads and processes different?

…

Most modern applications are multithreaded

- E.g., word processor, Web browser, …
  (check with `top`!)

Most OS kernels are also multithreaded

- Process creation: slow, needs more resources
- Thread creation: faster, needs fewer resources

Switching processes is also more expensive than switching threads.

Threads/Processes speed comparison

From:

https://computing.llnl.gov/tutorials/pthreads/#Pthread

Time to create 50k processes/threads (in seconds):

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6GHz Xeon (16 cores/node)	8.1	0.1	2.9	0.5	0.2	0.3
Intel 2.8GHz Xeon (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
Intel 2.4GHz Xeon (2 cores/node)	54.9	1.5	20.8	1.6	0.7	0.9

Benefits of multiple threads

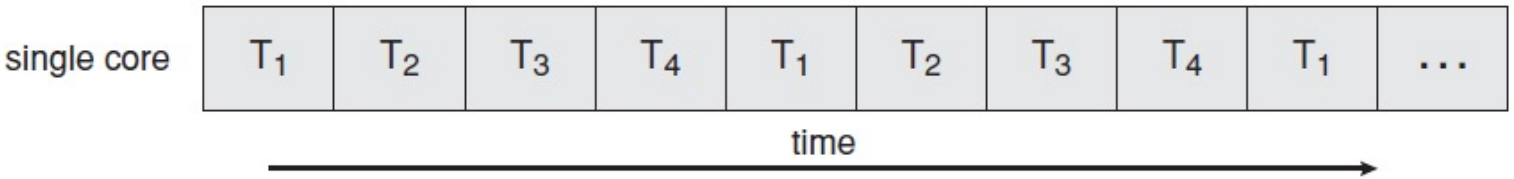
Benefits of threads:

- Responsiveness
  - If current thread is waiting for IO, or busy with a slow operation
    - Can switch to other thread
- Resource sharing
  - Threads share memory and resources by default
  - Easier code than shared memory/message passing in processes
- Economy
  - Cheaper to create/switch threads vs. processes
- Scalability
  - Threads can be running in parallel in multi-processor architecture

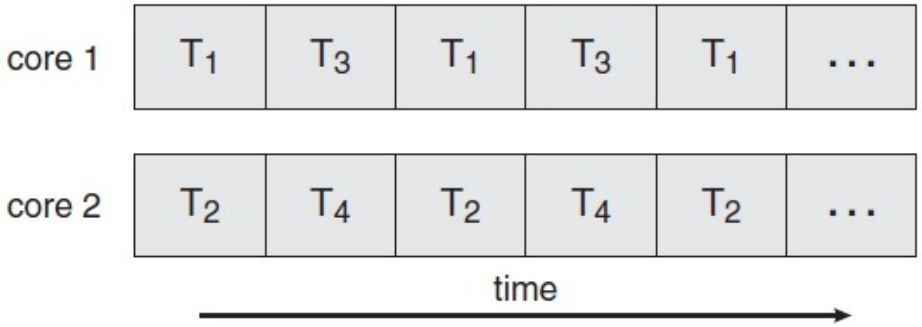
Multicore/Multithreaded Programming

- Concurrency:
  - Usually for single core system
  - One thread runs at a time
  - Switch threads to overlap idle time
- Parallelism:
  - For multi-core/multiprocessor system
  - Multiple threads run at a time, one on each core
  - Data Parallelism:
    - Each thread has a subset of the full data set
    - Each thread performs same operation on data subset
  - Task Parallelism:
    - Each thread performs different operation on data set

Multicore Execution



Concurrent execution on a single-core system



Parallel execution on a multi-core system

Non-atomic operations

- Incrementing and decrementing a variable are examples of non-atomic operations.

- The following C expression,

```
X++;
```

- Is translated by the compiler to three instructions:
  - **load** X from memory into a CPU register.
  - **increment** X and save result in a CPU register.
  - **store** result back to memory.

Race condition

- A race condition or race hazard is the behavior of system where the output is dependent on the sequence or timing of other uncontrollable events.
- It becomes a bug when events do not happen in the intended order.

- A data race occurs when two instructions from different threads access the same memory location and:
  - at least one of these accesses is a write
  - and there is no synchronization that is mandating any particular order among these accesses.

Interleaving of executing threads

Thread A (increment)				Thread B (decrement)		
OP	Operands	$t0	BALANCE	OP	Operands	$t0
			0			
lw	$t0, BALANCE	0	0			
			0	lw	$t0, BALANCE	0
addi	$t0, $t0, 1	1	0			
sw	$t0, BALANCE	1	1			
			1	addi	$t0, $t0, -1	-1
			-1	sw	$t0, BALANCE	-1

The "*Too Much Milk*" problem

Time	You	Your Roommate
3:00	Arrive Home	
3:05	Look in fridge, no milk	
3:10	Leave for grocery	
3:15		Arrive Home
3:20		Look in fridge, no milk
3:25		Leave for grocery
3:30	Arrive at grocery store	
3:35	Buy milk	
3:40	Arrive home, put milk in fridge	
3:45		Arrive at grocery store
3:50		Buy milk
3:55		Arrive home, put milk in fridge
4:00		Uh oh!

Solving the Too Much Milk Problem

Correctness properties

- Only one person buys milk
  - Safety: "nothing bad happens"
- Someone buys milk if you need to
  - Progress: "something good eventually happens"

First: use atomic loads stores as building blocks

- "Leave a note" (lock)
- "Remove a note" (unlock)
- "Don't buy milk if there's a note" (wait)

Too Much Milk: Attempt 1

- Thread A
  - if(no milk && no note)
    - leave note
    - buy milk
    - remove note
- Thread B
  - if(no milk && no note)
    - leave note
    - buy milk
    - remove note

Too Much Milk: Attempt 1

- Thread A
  - if(no milk && no note)
    - leave note
    - buy milk
    - remove note
- Thread B
  - if(no milk && no note)
    - leave note
    - buy milk
    - remove note

If both threads switch after reading the if statement, then both will buy milk.

Too Much Milk: Attempt 2

Idea: use labeled notes

- Thread A
  - leave note A
  - if (no note B)
    - if (no milk)
      - buy milk
  - remove note A

- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B

Too Much Milk: Attempt 2

Idea: use labeled notes

- Thread A
  - leave note A
  - if (no note B)
    - if (no milk)
      - buy milk
  - remove note A
- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B

If both threads switch after leaving a note, neither will buy milk.

Too Much Milk: Attempt 3

Idea: *wait* for the right note

- Thread A
  - leave note A
  - while (no note B)
    - do nothing
  - if (no milk)
    - buy milk
  - remove note A
- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B

Too Much Milk: Attempt 3

Idea: *wait* for the right note

- Thread A
  - leave note A
  - while (no note B)
    - do nothing
  - if (no milk)
    - buy milk
  - remove note A
- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B
- Possibility 1: thread A runs first, then thread B.
  - thread A will buy milk. thread B will see there is milk, and therefore not buy any more milk.

Too Much Milk: Attempt 3

Idea: *wait* for the right note

- Thread A
  - leave note A
  - while (no note B)
    - do nothing
  - if (no milk)
    - buy milk
  - remove note A
- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B
- Possibility 2: thread B runs first, then thread A.
  - thread B will buy milk. thread A will see there is milk, and therefore not buy any more milk.

Too Much Milk: Attempt 3

Idea: *wait* for the right note

- Thread A
  - leave note A
  - while (no note B)
    - do nothing
  - if (no milk)
    - buy milk
  - remove note A
- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B

- Possibility 3: Interleaved - A waits and A buys
  - thread A and B can leave a note, but thread A will wait for thread B to finish before moving on.
  - thread B will not buy milk because note A exists, and finish the thread
  - thread A will continue and buy milk.

Too Much Milk: Attempt 3

Idea: *wait* for the right note

- Thread A
  - leave note A
  - while (no note B)
    - do nothing
  - if (no milk)
    - buy milk
  - remove note A
- Thread B
  - leave note B
  - if (no note A)
    - if (no milk)
      - buy milk
  - remove note B

- Possibility 4: Interleaved - A waits and B buys
  - thread B will leave a note, and see there is no note A.
  - A can start as see there is a note B, and wait for B to finish.
  - B will continue to buy milk if there no milk, and then finish before thread A will continue.
  - thread A will not buy milk because thread B just bought milk.

Critical section

- Concurrent accesses to shared resources can lead to unexpected or erroneous behavior
- Parts of the program where the shared resource is accessed are protected.
- This protected section is the **critical section** or **critical region**.
- Typically, the critical section accesses a shared resource, such as a data structure, a peripheral device, or a network connection, that would not operate correctly in the context of multiple concurrent accesses

Mutual exclusion (mutex)

- In computer science, **mutual exclusion** is a property of **concurrency control**
- This control is instituted for the purpose of **preventing race conditions**;
  - it is the requirement that one thread of execution never enter its critical section at the same time that another concurrent thread of execution enters its own critical section.
- Often the word **mutex** is used as a short form of mutual exclusion.

User vs Kernel Threads

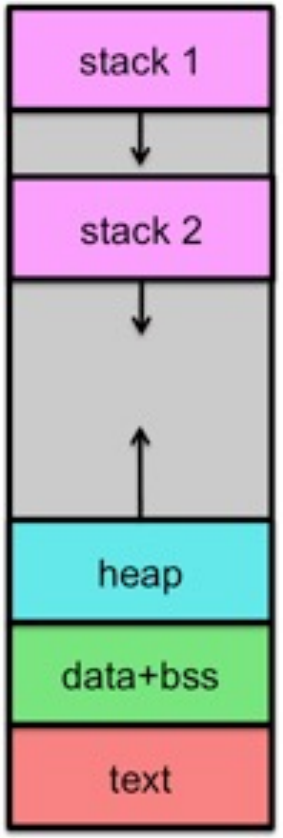
- There is the concept of
  - User Threads, that is threads that are managed entirely in the user space
  - Kernel Threads, threads that are known by and managed by the Kernel

- There are advantages and disadvantages to both. But since this is a Linux Programming course, we will ignore the user threads and focus on the kernel threads.

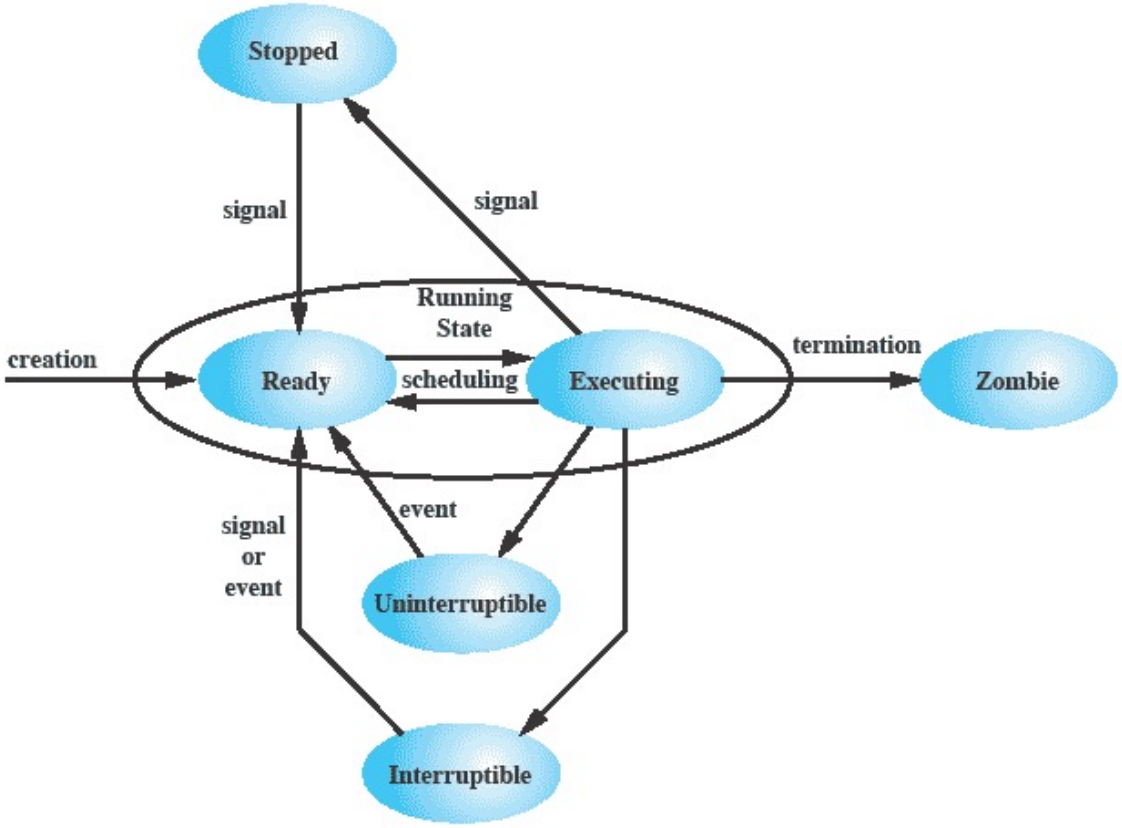
Threads

- All threads in a process share the Text, Data (initialized and uninitialized), heap, and library segments
- As well as
  - Open file descriptors
  - Signals
  - Current working directory
  - User and group IDs

- They do not share the Stack segment
- Nor do they share the CPU state (Program counter or Registers)
- They also have unique Thread IDs



Linux Process/Thread Model



Thread Example

- `pthread_t` are return structures
- `pthread_create` creates and starts a new thread

Program Code:

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

void *print_message_function( void *ptr ) {
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
    }

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);
```

Thread Example

- `pthread_join` is similar to the `wait` we used for `fork`'d processes
- But threads to NOT have a child – parent relationship any thread can wait on any other thread

Program Code:

```
/* 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);
}
```

Console Output:

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

pthread_create

Function call: **pthread_create** Arguments:

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

- thread - returns the thread id. (unsigned long int defined in bits/pthreadtypes.h)
- attr - Set to NULL if default thread attributes are used.
- void * (*start_routine) - pointer to the function to be threaded. Function has a single argument: pointer to void.
- *arg - pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

pthread_create attributes (1 of 2)

Attribute	**Meaning**	**APIs:** `pthread_attr_[...](3)`	**Values Possible**	***Linux Default***
Detach state	Create threads as joinable or detached	`pthread_attr_ [get|set]detachstate`	PTHREAD_CREATE_JOINABLE PTHREAD_CREATE_DETACHED	PTHREAD_CREATE_JOINABLE
Scheduling/contention scope	Set of threads against which we compete for resources (CPU)	`pthread_attr_ [get|set]scope`	PTHREAD_SCOPE_SYSTEM PTHREAD_SCOPE_PROCESS	PTHREAD_SCOPE_SYSTEM
Scheduling/inheritance	Determines whether scheduling attributes are inherited implicitly from calling a thread or explicitly from the attr structure	`pthread_attr_ [get|set]inheritsched`	PTHREAD_INHERIT_SCHED PTHREAD_EXPLICIT_SCHED	PTHREAD_INHERIT_SCHED
Scheduling/policy	Determines the scheduling policy of the thread being created	`pthread_attr_ [get|set]schedpolicy`	SCHED_FIFO SCHED_RR SCHED_OTHER	SCHED_OTHER

pthread_create attributes (2 of 2)

Scheduling/priority	Determines the scheduling priority of the thread being created	`pthread_attr_ [get|set]schedparam`	struct sched_param holds int sched_priority	0 (non real-time)
Stack/guard region	A guard region for the thread's stack	`pthread_attr_ [get|set]guardsize`	Stack guard region size in bytes	1 page
Stack/location, size	Query or set the thread's stack location and size	`pthread_attr_ [get|set]stack pthread_attr_ [get|set]stackaddr pthread_attr_ [get|set]stacksize`	Stack address and/or stack size, in bytes	Thread Stack Location: left to the OS Thread Stack Size: 8 MB

Thread Example

- Here we explicitly set a thread as joinable using the attribute parameter

Program Code:

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

    /* Init the thread attribute structure to defaults */
    pthread_attr_init(&attr);

    /* Create all threads as joinable */
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

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

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

Terminating a thread

- A thread terminates when:
  - It reaches the end of its start_routine
  - It calls pthread_exit
  - It is canceled by another thread calling pthread_cancel
  - The process that contains the thread terminates, for example, because of a thread calling exit, or the process receives a signal that is not handled, masked, or ignored

- Note that if a multithreaded program calls fork, only the thread that made the call will exist in the new child process. Fork does not replicate all threads.

pthread_exit

```
#include <pthread.h>
void pthread_exit(void *retval);
```

- The parameter specifies the exit status of the calling thread;
- This can be NULL
- A casted value
  - `pthread_exit((void *)1);`
- Or a pointer to a structure

Joins

- A join is performed when one wants to wait for a thread to finish.
- A thread calling routine may launch multiple threads then wait for them to finish to get the results.

- Unlike `fork` – any thread may wait for another thread, not just the "parent"

- `int pthread_join(pthread_t thread, void **retval);`
  - The first parameter to `pthread_join`, thread, is the ID of the thread to wait for.
  - The moment it terminates, the calling thread will receive, in the second parameter (a value-result style parameter), the return value from the thread that terminated—which, of course is the value passed via its `pthread_exit` call.

Mutexes

- Mutexes are used to prevent data inconsistencies due to race conditions.
- Mutexes are used for serializing shared resources.
- Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.
- One can apply a mutex to protect a segment of memory ("critical region") from other threads.
- Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

Mutex Example

- Main Code

Program Code:

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

void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;

main() {
    int rc1, rc2;
    pthread_t thread1, thread2;

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

    if ((rc1=pthread_create( &thread1, NULL, &functionC, NULL)))
        {
        printf("Thread creation failed: %d\n", rc1);
        }
```

Mutex Example

- Main code

Program Code:

```
    if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
        {
        printf("Thread creation failed: %d\n", rc2);
        }

    /* 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);

    exit(0);
    }
```

Mutex Example

- Thread code

Program Code:

```
void *functionC()
    {
    pthread_mutex_lock( &mutex1 );
    counter++;
    printf("Counter value: %d\n",counter);
    pthread_mutex_unlock( &mutex1 );
    }
```

Mutex Locks

- Two main functions:
  - `pthread_mutex_lock (pthread_mutex_t * mutex);`
  - `pthread_mutex_unlock (pthread_mutex_t * mutex);`

- The first start a critical section, the second ends it.

Mutex Locks

- In order to use a mutex lock, one must first initialize it to the unlocked state; this can be done as follows:

```
if ((ret = pthread_mutex_init(&mylock, NULL)))
    FATAL("pthread_mutex_init() failed! [%d]\n", ret);
```

- Alternatively, we could perform the initialization as a declaration, such as:

```
pthread_mutex_t mylock = PTHREAD_MUTEX_INITIALIZER;
```

- Also, once we are done, we must destroy the mutex lock(s):

```
if ((ret = pthread_mutex_destroy(&mylock)))
    FATAL("pthread_mutex_destroy() failed! [%d]\n", ret);
```

Wait on first to finish

- How to wait on finish order, rather than creation order

Program Code:

```
void *routine(void *arg)
    {
    int *fds = (int *)arg;

    pthread_t t = pthread_self();  /* Get Thread ID */

    usleep((rand()/(1.0 + RAND_MAX)) * 1000000);

    write(fds[1], &t, sizeof(t));
    }
```

Wait on first to finish

- How to wait on finish order, rather than creation order

Program Code:

```
int main() {
    int fds[2];
    srand(time(0));
    pipe(fds);
    for (int i = 0; i < 2; i++) {
        pthread_t tid;
        pthread_create(&tid, NULL, routine, fds);
        printf("created: %llu\n", (unsigned long long)tid);
        }
    for (int i = 0; i < 2; i++) {
        pthread_t tid;
        read(fds[0], &tid, sizeof(tid));  /*Block on Pipe, 1st to write, 1st done*/
        printf("joining: %llu\n", (unsigned long long)tid);
        pthread_join(tid, 0);
        }
    }
```

Daemons

- A daemon is a process that runs in the background, owned by the init process and not connected to a controlling Terminal.
- The steps to create a daemon are as follows:
  1. Call fork to create a new process, after which the parent should exit, thus creating an orphan which will be re-parented to init.
  2. The child process calls `setsid`, creating a new session and process group of which it is the sole member.
     - Consider this a way of isolating the process from any controlling terminal.
  3. Change the working directory to the root directory.
  4. Close all file descriptors and redirect stdin, stdout, and stderr (descriptors 0, 1, and 2) to /dev/null so that there is no input and all output is hidden.

- Thankfully, all of the steps can be achieved with a single function call, `daemon`