CSci 4061
Introduction to Operating Systems

(Threads-POSIX)
Thread Overview

• Thread advantages-
  • modularity, concurrency
  • sharing, cheap

• Sharing is a double edged sword
  • race conditions, failure

• Implementation models
  • User, kernel, hybrid

• Programming models
How do I program them?
General Thread Operations

- **Create/Fork**
  - Allocate memory for stack, perform bookkeeping
  - Parent thread creates child threads
  - Associates function to execute
  - Returns an id

- **Destroy/Cancel**
  - Release memory (or recycle), perform bookkeeping

- **Suspend (e.g. Sleep)**
  - ->blocked, Resume->unblock, yield->deschedule

- **Wait/Join**
  - Wait for something, e.g. child finishing
Pthread: Creation

- Creating a thread is like a combination of fork() and exec()

  
  # include <pthread.h>
  int pthread_create(pthread_t *thread,
                 pthread_att_r *attr,
                 void * (*function)(void *),
                 void *arg);

- thread is the returned thread ID, attr is an attribute set
- function is the function to be called with arg

Compile/Link with -D_REENTRANT-lpthreads

*************may not get linker errors!!
Pthreads: Creation (cont’d)

• The thread stays in the system until its function returns (or is cancelled/killed)
  • At that point the thread is finished

• Most POSIX thread calls returns ) upon success, nonzero otherwise

• POSIX thread functions return an error code - they do not set errno!

• Thread states: running, blocked, ready
• K ready threads, 1 is running (single core)
Parameters and return values

- When you start a thread, you pass its function a pointer to an arg

Void *thread_fn (void*arg) {...}
  - arg is a void* so you can cast it to whatever you need
  - when pthread_create () returns thread_fn may not be running....yet

void main (){  
  pthread_t tl;
  int x;

  pthread_create (&tl, NULL, thread_fn, (void*)&x);
  ...
}
Thread identity

- Threads are identified by the value type `pthread_t`

```c
#include <pthread.h>
pthread_t pthread_self (void);
    pthread_self () returns the identity of the calling thread

int pthread_equal
    (pthread_t tl, pthread_t t2);
```
Thread Termination

• The thread function returns `void*` when thread returns/finishes
  • Be careful with return value
  • What must be true of the return value?—not a stack value

• You can also explicitly exit elsewhere
  
  ```
  #include <pthread.h>

  void pthread_exit (void *return_value);
  
  return and pthread exit are the same, except in the main thread (where return ends the process)
  
  exit/abort will terminate the process if called from any thread
  ```
Thread Cancellation

• Cancel a thread when it is a good time to “stop”
  • Done from the “outside”, e.g. parent
  • Make a cancellation request

```c
#include <pthread.h>
void pthread_cancel (pthread_t thread);
```

• Cancellation can be controlled

  • See `pthread_setcancel {state | type}`

  • Using state, thread can control if it is cancellable ... (it is, by default)
  • Using type, a thread may be cancelled (e.g. anytime, at a blocking point, ...)

• PTHREAD_CANCEL_{ENABLE/DISABLE}
Joining threads

Joining a thread is analogous to waiting/blocking for a child process to complete

```c
#include <pthread.h>
Int pthread_join (pthread_t th,
                    void **thread_return);
```

thread_return is the exit value of the thread
Note: unlike wait() have to name the thread in question

For fun, try a pthread_join (pthread_self())

what happened and why?
Pthread example

#include <pthread.h>
#include <stdio.h>
void *pmf (void *msg) {
    char *message;
    message = (char*) msg;
    fprintf (stderr, “%s”, message);
    return 0;
}
Pthread example (cont’d)

```c
int main (){
    pthread_t t1, t2;
    char *message1 = "Hello";
    char *message2 = "World";
    pthread_create (&t1, NULL, pmf,
        (void*) message1);
    pthread_create (&t2, NULL, pmf,
        (void*) message2);
    pthread_join (t1, NULL); // block until t1 finishes
    pthread_join (t2, NULL); // block until t2 finishes
    exit (1);}
```

Run it
Detaching threads

```c
#include <pthread.h>
int pthread_detach (pthread_t thread);
```

- A detached thread cannot be joined — it will just go away when it exits
- You cannot detach a thread if some other thread is joining it
- Good style and practice: should either detach or join every thread
- For joinable threads, its resources are not released until `join` is performed
Pthread attributes

- Examples
- Stack size
- Scheduling attributes
Pthread attributes

- You can set thread attributes before you start the thread.

```c
#include <pthread.h>

//init attr to default values
int pthread_attr_init (pthread_attr_t *attr);

int pthread_attr_setschedpolicy (pthread_attr_t *attr, int policy);

int pthread_attr_getschedpolicy (pthread_attr_t *attr, int *policy);

pthread_attr_t 2nd arg passed to pthread_create.
Pthread scheduling

- Can set policy to SCHED_FIFO, SCHED_RR, SCHED_OTHER

SCHED_OTHER is usually pre-emptive

Change parameters of scheduling policy
- You can set thread attributes before you start the thread

```c
#include <sched.h>
pthread_attr_getschedparam (pthread_attr_t *attr, 
                           struct sched_param*param);
pthread_attr_setschedparam (pthread_attr_t *attr, 
                           struct sched_param*param);

pthread_attr_getschedparam (attr, &param);
param.sched_priority = priority;  //int; ^ high
pthread_attr_setschedparam (attr &param);
```
Thread Implementations

- POSIX threads are implemented by a user-level library
  - May be pure user-level
  - Can be exploit kernel threads if available
  - Behavior can vary slightly
POSIX thread safety

• All threads see the same global environment
• Thread safety is an issue — globals and static data, heap data
• Any library function that is async-signal-safe is thread-safe
  (see man pages)

• Compile with -D_REENTRANT
POSIX thread safety (cont’d)

• Each thread gets a copy of non-local variables (e.g. errno)

  • Variables (even static ones) get moved into the thread context

    #define errno ___errno (thread_ID)

• Shared data-structures like heap code (in malloc) are protected by locks
Next time

• One of the drawbacks with threads ... synchronization!

• Chapter 13 R & R

• Have a great weekend

• Good luck with lab #3