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
  - dispatcher and team
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
Remember

• System may time-slice your thread

• You should assume that a thread could be switched at any time ... your program should still work
  • A program that fails 1 out of $10^{100}$ runs is buggy

• This will make your code much more portable
Pthread: Creation

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

```c
#include <pthread.h>

int pthread_create(pthread_t *thread,
                    pthread_attr_t *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 -lpthread

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

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

• Most POSIX thread calls returns 0 upon success, nonzero otherwise

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

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

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

```c
void *thread_fn (void *arg) {
    printf (“%d”, *((int *)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

```c
void main (){  
    pthread_t tl;
    int x = 1;

    pthread_create (&tl, NULL, thread_fn,(void*)&x);
    x = 2;
...
}
```
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 a `void*` when thread returns/finishes
  • be careful with return value

  • what must be true of the return value?—not a stack value! WHY?

• You can also explicitly exit elsewhere
  
  ```c
  #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, NULL);
```

- Cancellation can be controlled
- See `pthread_setcancel{state | type}`

```c
pthread_cancelstate (PTHREAD_CANCEL_DISABLE, NULL)
```

- Using `state`, thread can control if it is cancellable ... (it is, by default)
- Using `type`, control when a thread may be cancelled
  - anytime, at a blocking point
Lab #3
Joining threads

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

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

thread_return is the exit value of the thread
    from return or pthread_exit

Note: unlike wait() have to name the thread in question

For fun, try a pthread_join (pthread_self())

    what happened and why?
Yield

• To yield a thread:
  • gives up the CPU -- HINT

```
int pthread_yield () ;
```

• Suspend (block)/Resume (unblock)
  • Posix doesn’t have these explicitly
  • Other thread packages do
  • We can achieve this with synchronization
    • E.g. locks
Pthread example

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

void *pmf (void *msg) {
    char *message;
    message = (char*) msg;
    fprintf (stderr, "%s", message);
    return 0; }
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);}

<threadtest>
Parameters

• What is the problem with this?

```c
void main (){ 
    pthread_t tl; 
    int i; 

    for (i=0; i<MAX; i++) 
        pthread_create (&tl, NULL, 
                        thread_fn, (void*)&i); 

    ... 
}
```
Fix

int args[count];

for (int i = 0; i < count; i++) {
    args[i] = i;
    pthread_create(&p[i], NULL, thread_fn, (void*) &args[i]);
}
Detaching threads

#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
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
Mixing Concepts

• Fork + threads = ?
  • behavior is unspecified
  • seems to work—why?
  • `<threadfork>`

• Signals + threads = ?
  • process-level signal can interrupt any thread
  • however, it is possible that if you call `sigaction/signal` within a thread function, it may get directed to “that” thread
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 (just in case)
POSIX thread safety (cont’d)

• Most standard libraries have thread-safe code

  • Variables (even global and static ones) *may* get moved into the thread context

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

• Errno:

  ```c
  #define errno  __errno (thread_ID)
  ```

• Shared data-structures like heap code (in `malloc`) *are* protected by locks
Pthread attributes

• Things you can change include:

• Stack size

• Scheduling attributes
Default policy?

• Time-slicing
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

Can also change parameters of scheduling policy

```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_t attr;
struct sched_param param;
pthread_attr_init (&attr);
pthread_attr_getschedparam (&attr, &param);
param.sched_priority = priority; //int; ^high
pthread_attr_setschedparam (&attr, &param);
```
A Word about Scheduling

• It is a HINT and may be ignored
Exam #2 Coverage

• Material since last exam

• IPC: shared memory, message-passing, issues, programming interfaces

• Signals—how are they used?
  • Dealing with them
  • Issues—reentrancy, races
  • Programming models
Exam (cont’d)

• Threads—what good are they?
  • issues: models, implementations, race conditions, pitfalls, programming
  • alternative ways to get concurrency

• Synchronization—what is it?
  • this is discussed on Tuesday
  • locks, programming
Sample

• Short answer:
  • What is a race condition?
  • List two main differences between Unix Pipes and Unix Message Queues?

• Longer
  • Write a multithreaded program to do X?
  • Use signals (e.g. ALARMS) to do Y?
Study

• Lab 2, 3
• Lecture notes
• Read the book - write multithreaded programs
Exam (cont’d)

• Closed book
  • 40% short answer
  • 60% programming (3 questions)
    • IPC
    • signals
    • threads/locks

• We’ll provide the APIs
Next time

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

• Chapter 13 R & R

• On-line as of tomorrow

• Best of luck on the exam!