CSci 4061
Introduction to Operating Systems

(Threads-POSIX)
Chapter 12)

Pthread: Creation

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

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

often NULL (input arg)

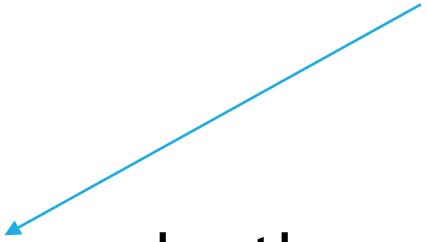
- `thread` is the returned thread ID, `attr` is an attribute set
- `function` is the function to be called with `arg`

Compile/Link with -lpthread

Older systems also recommend compiler flag: -D_REENTRANT

Pthreads: Creation (cont'd)

- The thread stays in the system until its function returns/exits (or it is `cancel`led/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`

```
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

```
void main (){
    pthread_t t1;
    int x = 1;

    pthread_create (&t1, NULL, thread_fn,(void*)&x);
    x = 2;
    …
}
```

main is the parent thread; t1 is the child

Thread identity

- Threads are identified by the value type `pthread_t`

```
#include <pthread.h>
pthread_t pthread_self ();
```

`pthread_self ()` returns the identity of the calling thread

```
int pthread_equal
    (pthread_t t1, 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

```
#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

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

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

```
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

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 `pthread_join (pthread_self(), NULL)`

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)

```
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);}
```

Parameters

- What is the problem with this?

```
void main (){
  pthread_t t[MAX];
  int i;

  for (i=0; i<MAX; i++)
    pthread_create (&t[i], NULL,
                    thread_fn, (void*)&i);
  …
}
```

```
void *thread_fn (void *msg) {
  int message;
  message = *(int*) msg;
  fprintf (stderr, "%d\n", message);
  return 0;
```

Fix

```
int args[count];


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

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

Yield Question

- T1 and T2 each call a block code concurrently
- Are there race conditions? Way to test?

Code_block:

```
C-instr1
C-instr2
C-instr3
…
C-instrn
```

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 exploit kernel threads if available
  - Behavior can vary slightly
  - Linux: supports kernel-level

POSIX thread safety

- All threads see the same global environment
- Thread safety is an issue — globals and static data, heap data
- Linux library functions/system calls are marked *thread-safe* if they are
  - see man pages

Pthread attributes

- Things you can change include:
- Stack size
- Scheduling attributes

Default scheduling policy?

- Time-slicing

Can set policy to SCHED_FIFO, SCHED_RR, SCHED_OTHER (just a hint)

Next time

- One of the drawbacks with threads … synchronization!

- Chapter 13 R & R