CONCURRENCY

Introduction: What is a thread?

- on board

multiple "programs" executing within the SAME address space!

usually cooperating to achieve some task, or independent related tasks

- e.g., parallel program, web/db server

thus, each thread has:

- its own private set of registers

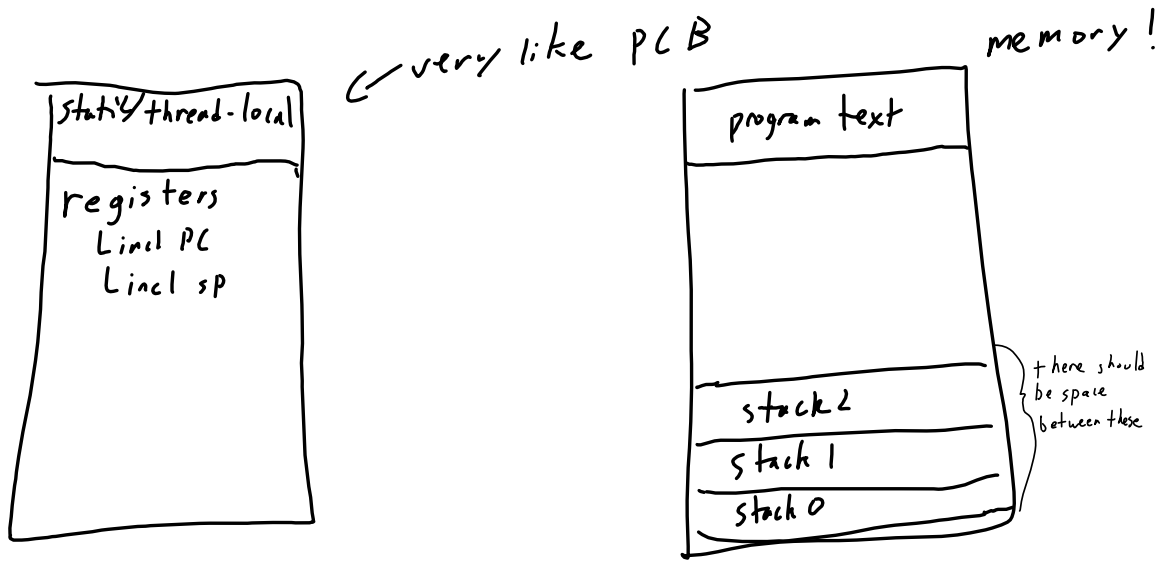
- its own program counter

- its own stack (and stack-pointer/base-pointer)

but shares

- rest of address space (heap, static global, code section too)

Control structures: the thread control block (TCB)



What makes thread programming hard?

main-thread-0 (no locks)

Examine in detail: imagine a trace of main-thread-0; how is "++" implemented?
assembly on the board: load <addr>; add 1; store <addr>

Why programs get tricky: SHARED DATA

REAL PROBLEM: uncontrolled scheduling (interrupts at any time)

- remember our timer-based scheduler?? This is where it hurts!

- note: this is NOT a problem if we're in cooperative scheduling with predictable yields.

lots of definitions:

- program is not deterministic (indeterminate)

- critical section (where the bug can be)

- race condition (what the bug is)

- need mutual exclusion (one-at-a-time)

(turn non-deterministic code into quasi-deterministic code)

main-thread-1 (fine-grained locks)

need synch primitives

main-thread-2 (coarse-grained locks)

need synch primitives but be careful

main-thread-3 (implement locks try #1: test-and-set)

just run it

(what is the problem?)

main-thread-4 (implement locks try #2: x86 xchg)

how to build a lock using special hardware?

(how to use xchg?)

main-thread-5 (implement locks try #2: x86 xchg + spinlock implementation)

this is how

objdump -d to look at it

Conclusions:

Why in OS class?

threads are basic OS primitive

OS itself is a concurrent program!

code fixed (1)

we just init a lock and lock the exact increment

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

int max;
volatile int balance = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *
mythread(void *arg)
{
    char *letter = arg;
    //cpubind();
    printf("%s: begin\n", letter);
    int i;
    for (i = 0; i < max; i++) {
        Pthread_mutex_lock(&lock);
        balance++;
        Pthread_mutex_unlock(&lock);
    }
    printf("%s: done\n", letter);
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [balance = %d]\n", balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
           balance, max*2);
    return 0;
}
```

code fix (2)

we move the lock to outside the loop, and cpubind()

```
#include <stdio.h>
#include "mythreads.h"
#include "mythreads-2.h"
#include <stdlib.h>
#include <pthread.h>

int max;
volatile int balance = 0;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *
mythread(void *arg)
{
    char *letter = arg;
    cpubind();
    printf("%s: begin\n", letter);
    int i;
    Pthread_mutex_lock(&lock);
    for (i = 0; i < max; i++) {
        balance++;
    }
    Pthread_mutex_unlock(&lock);
    printf("%s: done\n", letter);
    return NULL;
}

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: main-first <loopcount>\n");
        exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: begin [balance = %d]\n", balance);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");
    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done\n [balance: %d]\n [should: %d]\n",
           balance, max*2);
    return 0;
}
```

in the exchange-based one

```
SpinLock(volatile unsigned int *lock) {
  while (xchg(lock, 1) == 1)
 // spin!
}

SpinUnlock(volatile unsigned int *lock) {
    xchg(lock, 0);
}
```