Threading

15-213/18-213: Introduction to Computer Systems
14th Recitation, Nov 28, 2011

Adrian Trejo Nuñez (atrejo)
PH 125C 3:30p-4:30p
Today

- Threads
- Thread safety
- Proxy
Reminder

- Proxylab is due on 11:59p, Sunday Dec 4
- Sign for your partner on Autolab if you haven't already.
Threads

• What is a thread?
  • Registers
  • Stack
  • Stack pointer
  • Program counter

• Then a process is just a thread along with code, data, and kernel context
  • Processes can have more than one thread though
Why Use Threads?

- Concurrency
- Easy sharing of data structures and variables
- Cheaper than processes
  - Roughly half as many CPU cycles needed
POSIX Threads Interface

- Creating and reaping threads
  - pthread_create
  - pthread_join
- Determining your thread ID
  - pthread_self
- Terminating threads
  - pthread_exit
  - exit (kills all threads associated with process)
  - return (kills current thread)
include "csapp.h"

void *thread(void *vargp);

int main(void) {
    pthread_t tid;
    int i;
    for (i = 0; i < 42; ++i) {
        pthread_create(&tid, NULL, thread, NULL);
        pthread_join(tid, NULL);
    }
    return 0;
}

void *thread(void *vargp) {
    puts("Hello world!");
    return NULL;
}
Joinable vs. Detached

- Joinable threads need to be reaped by other threads to free up memory resources
  - `pthread_join`
- Detached threads are automatically reaped when they terminate
  - `pthread_detach(tid)`
  - `pthread_detach(pthread_self())`
- Default state is joinable
Thread Safety

- Each thread has its own logical control flow, but not its own set of data like a process.
- If we want to use threads to write concurrent programs, we will need to be careful with our data.
Race Conditions

- Occur when your correctness depends on one thread reaching point $x$ in its control flow before another thread reaches point $y$
  - Global variables
  - Threads dependent on conditions
Race Condition

- global++;
- Think of as:
  1. Load value of global into register
  2. Add one to register
  3. Store new value in address of global
- We don't want threads to interweave
  1. 1-2-3-1-2-3
- But they might...
  1. 1-2-1-2-3-3
Safety

- Need to synchronize threads so that any critical region has at most one thread in it
- Use semaphores for this synchronization
Semaphores

- Non-negative global integer synchronization variable

- Can do two operations on it
  - \( P(s) \rightarrow \text{while } (s == 0) \text{ wait(); } s--; \)
  - \( V(s) \rightarrow s++; \)

- Only one \( P \) or \( V \) operation can modify \( s \)
  - When while loop in \( P \) terminates, only that \( P \) can decrement \( s \)
POSIX Semaphore Interface

- Creating and destroying a semaphore
  - `sem_init`
  - `sem_destroy`
- Modifying a semaphore's value
  - `sem_wait // P`
  - `sem_post // V`
Safe Multi-threading

#include "csapp.h"

static volatile int global = 0;
static sem_t mutex;

int main(void) {
    pthread_t tid1, tid2;
    sem_init(&mutex, 0, 1);
    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    if (global == 10000)
        return 0;
    return -1;
}

void *thread(void *vargp) {
    int i;
    for (i = 0; i < 5000; ++i) {
        sem_wait(&mutex);
        global++;
        global++;
    }
    sem_post(&mutex);
    return NULL;
}
Proxy

- Your proxy needs to handle concurrent requests
- Writeup suggests to spawn thread for every request
- All of those threads will try to access and modify your proxy's cache
- Make sure you have no race conditions!
  - Can also use pthread_mutex_t instead of sem_t