Introduction to Computer Systems
15-213/18-243 Spring 2012
April 16th, 2012

Concurrent Programming / Proxy Lab

Julian Shun

Announcements

- **Proxy Lab due April 26th**
- **May work with a partner**
  - Register on autolab
- **Start now!**

Overview

- **Processes vs Threads**
- **Threading**
  - Basics
  - Thread Lifecycle
- **Thread Safety (Preview)**
  - Race Conditions
  - Synchronization Techniques
- **Proxy Lab**

Threads vs. Processes

- **How threads and processes are similar?**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched

Threads vs. Processes

- **How threads and processes are similar?**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different?**

Threads vs. Processes

- **How threads and processes are similar?**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different?**
  - Threads share code and some data
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) to create and reap a thread

Traditional View of a Process

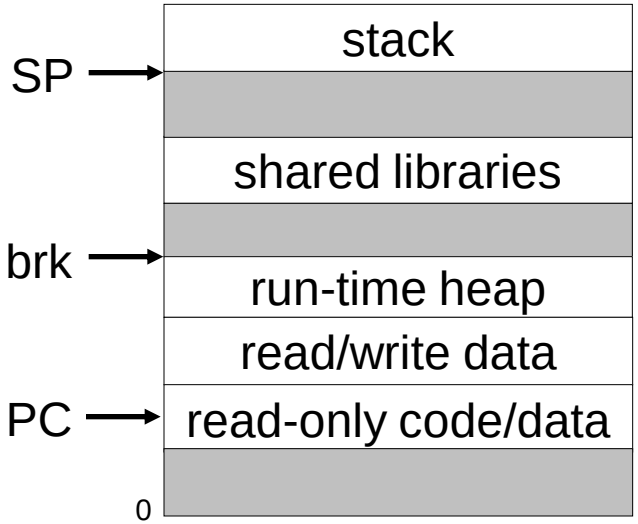
- **Process = process context + code, data, and stack**

Process context

Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

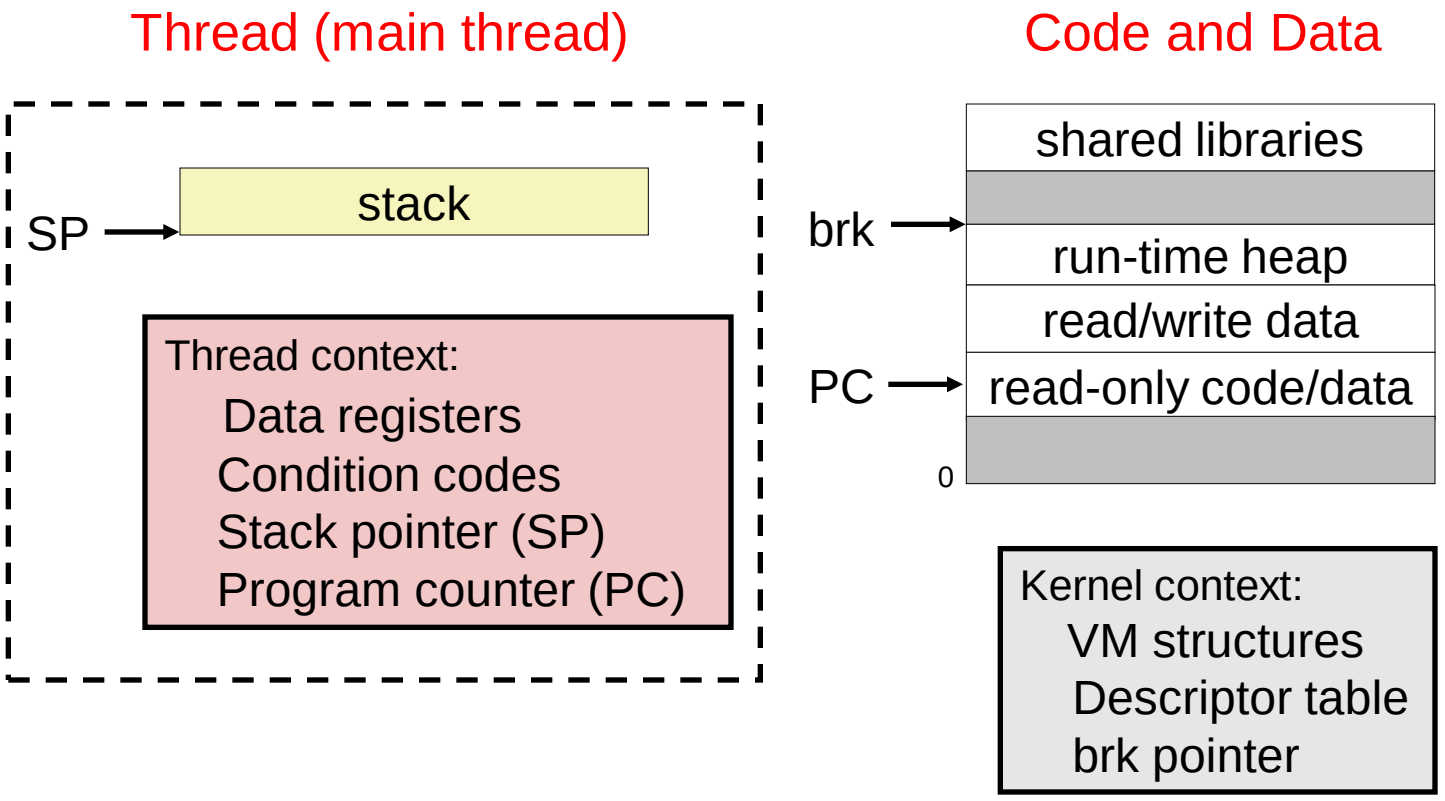
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



Alternate View of a Process

- **Process = thread + code, data, and kernel context**



Multi-Threaded process

Thread 1

stack 1

Thread 1 context:
- Data registers
- Condition codes
- SP-1
- PC-1

Thread 2

stack 2

Thread 2 context:
- Data registers
- Condition codes
- SP-2
- PC-2

...

Thread N

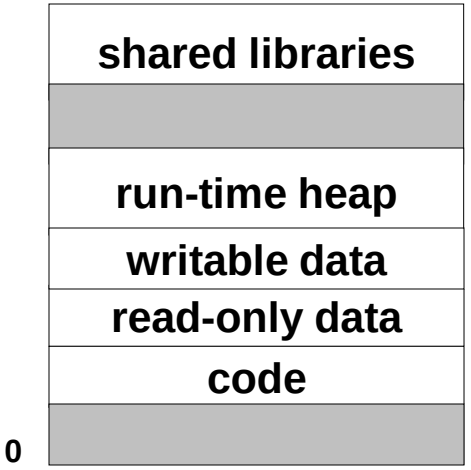
stack N

Thread N context:
- Data registers
- Condition codes
- SP-N
- PC-N

Shared resources:

Kernel context:
- VM structures
- Descriptor table

Private Address Space



Posix Threads (Pthreads) Interface

- **Standard interface for ~60 functions**
  - Creating and reaping threads.
    - pthread_create
    - pthread_join
  - Determining your thread ID
    - pthread_self
  - Terminating threads
    - pthread_cancel
    - pthread_exit
  - Synchronizing access to shared variables
    - pthread_mutex_init
    - pthread_mutex_[un]lock
    - pthread_rwlock_init
    - pthread_rwlock_[wr]rdlock

Multi-threaded Hello World

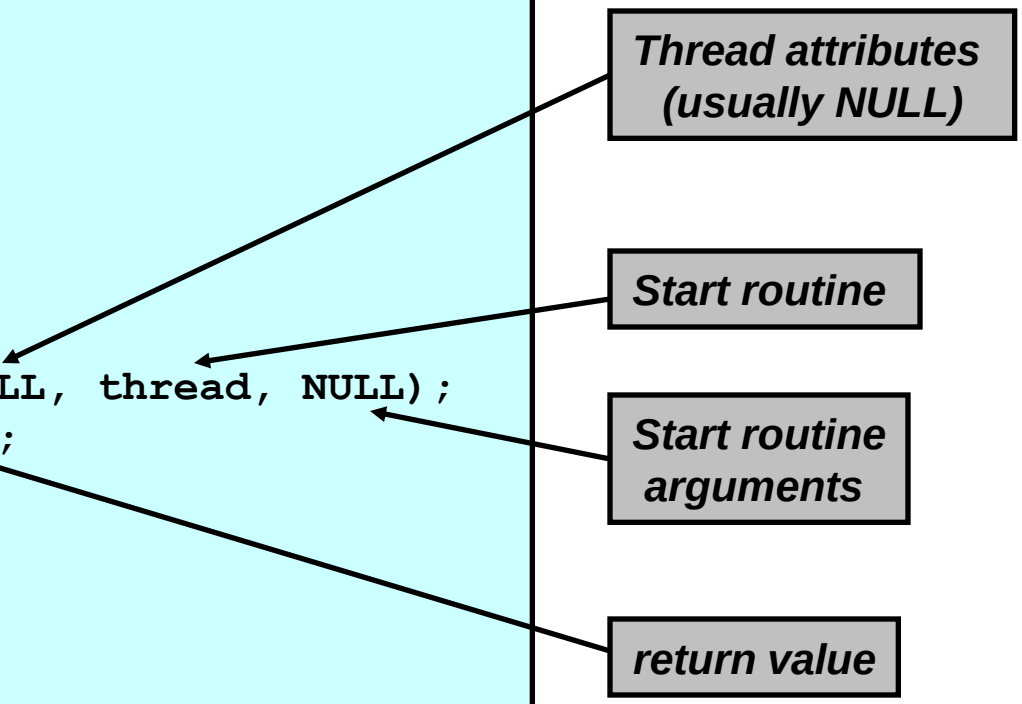
```
/* hello.c - Pthreads "hello, world" program */

#include "csapp.h"

void *thread(void *vargp);

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

/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
```



Exiting a process and thread

- A thread terminates *implicitly* when its top-level thread routine returns
- A thread terminates *explicitly* by calling pthread_exit(NULL)
- pthread_exit(NULL) only terminates the current thread, NOT the process
- exit() terminates ALL the threads in the process, i.e., the process itself
- pthread_cancel(tid) terminates the thread with id equal to tid

Joinable & Detached Threads

- **Joinable thread can be reaped and killed by other threads**
  - must be reaped (with pthread_join) to free memory resources.
- **Detached thread cannot be reaped or killed by other threads**
  - resources are automatically reaped on termination.
- **Default state is joinable**
  - use pthread_detach(pthread_self()) to make detached.

Thread Safety (Preview)

Race condition

- **A race occurs when the correctness of a program depends on one thread reaching point x in its control flow before another thread reaches point y.**
  - Access to shared variables and data structures
  - Threads dependent on a condition
- **Use synchronization to avoid race conditions**
- **Ways to do synchronization**
  - Semaphores
  - Mutex
  - Read-write locks

Synchronization

- **Semaphore**
  - Restricts the number of threads that can access a shared resource
- **Mutex**
  - Special case of semaphore that restricts access to one thread
- **Read-write locks**
  - Multiple readers allowed
  - Single writer allowed
  - No readers allowed when writer is present

Semaphore

- **Classic solution: Dijkstra's P and V operations on semaphores.**

- **Semaphore: non-negative integer synchronization variable.**
  - P(s): [while (s == 0) wait(); s--;]
  - V(s): [s++;]
  - OS guarantees that operations between brackets [ ] are executed indivisibly.
  - Only one P or V operation at a time can modify s.
  - Semaphore invariant: (s >= 0)
  - Initialize s to the number of simultaneous threads allowed

POSIX synchronization functions

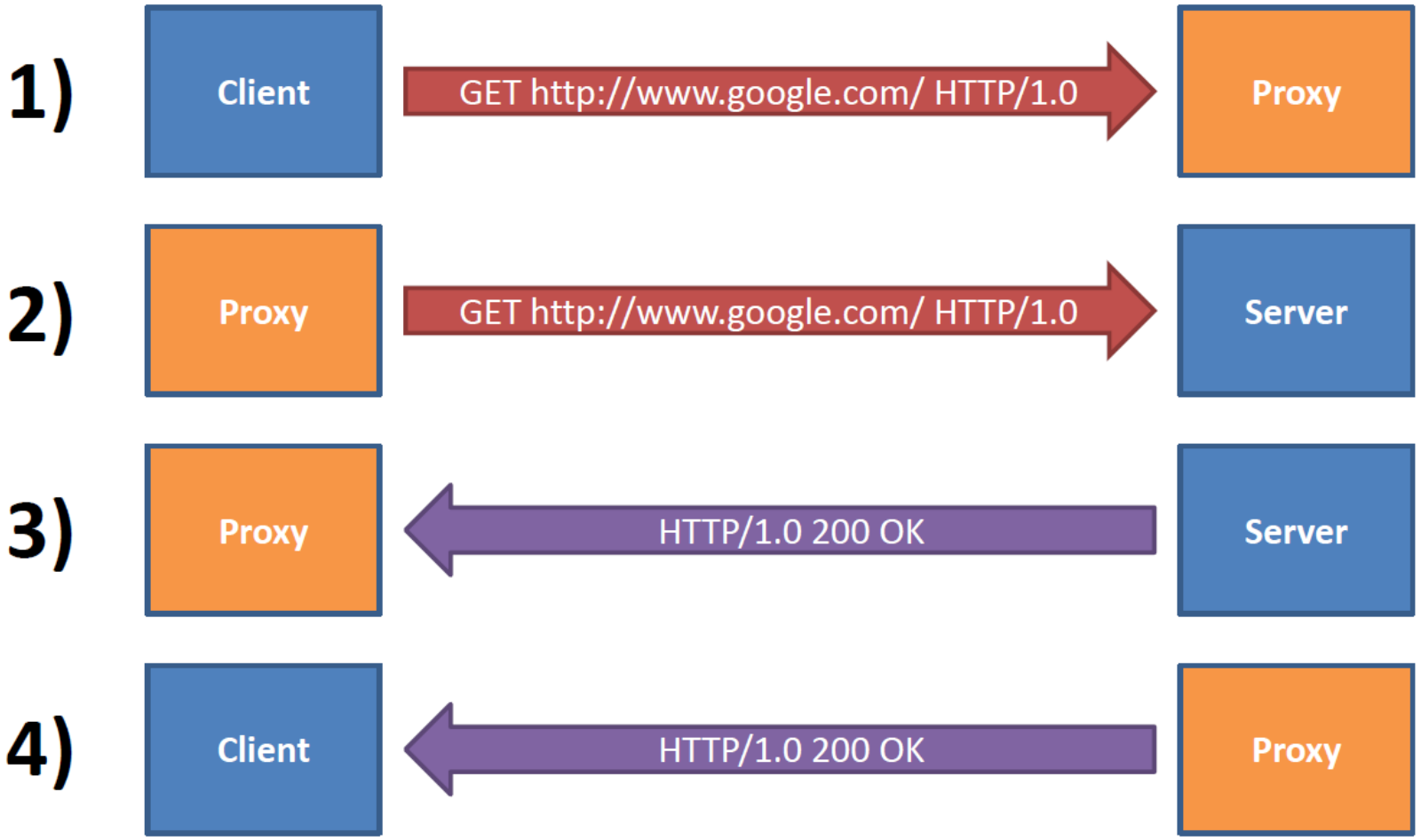
- **Semaphores**
  - sem_init
  - sem_wait
  - sem_post

- **Read-write locks**
  - pthread_rwlock_init
  - pthread_rwlock_rdlock
  - Pthread_rwlock_wrlock

Proxy Lab -- due April 26th

- **May work with partner (register on autolab)**
- **Read the writeup very carefully!**
- **Graceful error handling**
  - Proxy should not exit once it has finished initialization
- **Document design decisions**
- **Code organization**
  - Break proxy into multiple functions
- **Complete lab in three stages**
  - Basic sequential proxy
  - Handling concurrent requests
  - Caching web objects
- **Understand what is robust about the rio package**
  - Behavior of network sockets

What is a proxy?



What is a Caching Proxy

1)

The Proxy has already serviced a request for http://www.google.com/ and has stored the result.

2)

The Proxy simply responds with the stored result for http://www.google.com/. The Client is unaware that it has not communicated with the google.com server directly.

Important Notes on ProxyLab

RIO Package

- Provided for you in csapp.c
- The rio package has a very strict method for dealing with error. Should your proxy use the same method?
- Note: remember to account for binary data when doing I/O

Gethostbyname

This is the wrapper in csapp.c. What could go wrong?

```
/* $begin gethostbyname */
struct hostent *Gethostbyname(const char *name) {
        struct hostent *p;
        if ((p = gethostbyname(name)) == NULL)
        dns_error("Gethostbyname error");     return p;
}
 /* $end gethostbyname */
```

Thread-Unsafe Functions (cont)

- Returning a ptr to a `static` variable
- Fixes:
  - 1. Rewrite code so caller passes pointer to `struct`
    - Issue: Requires changes in caller and callee
  - 2. *Lock-and-copy*
    - Issue: Requires only simple changes in caller (and none in callee)
    - However, caller must free memory

```
struct hostent
*gethostbyname(char name)
{
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

```
hostp = Malloc(...));
gethostbyname_r(name, hostp);
```

```
struct hostent
*gethostbyname_ts(char *name)
{
  struct hostent *q = Malloc(...);
  struct hostent *p;
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;   /* copy */
  V(&mutex);
  return q;
}
```

Alternative (Better) Solution

- As you know from writing malloc, many things happen behind the scenes when malloc/free are called. This includes overhead of both time and space.
- What might be a better solution?

Alternative (Better) Solution

- As you know from writing malloc, many things happen behind the scenes when malloc/free are called. This includes overhead of both time and space.
- What might be a better solution?
- Declare a variable on the stack and pass in a pointer to that variable.
- Why is this still ok?
- Why is it better?

Testing

- **No driver program to evaluate correctness**
- **Test simple pages at the beginning and more complicated ones as your proxy improves**
- **Not all pages will work!**
  - Only need to handle GET requests
- **Use port_for_user.pl to pick a reasonable port number**
- **Useful tool: netcat**
  - Client: nc <host> <port>
  - Server nc –l <port>
- **Other tools: curl, thttpd**
- **See writeup for details**

Evaluation

- 30 points – basic proxy operation
- 30 points – handling concurrent requests
- 30 points – caching
- 10 points – style
- Half of correctness score determined by private autograder and other half determined by manual grading
- Handin: “make submit” and upload the output, proxylab.tar.gz, to autolab

Questions?