Introduction to Computer Systems
15-213/18-243 Fall 2010
November 15th, 2010

Threading and Thread Safety
Overview

- News
- Threading
  - Basics
  - Thread Lifecycle
- Thread Safety
  - Race Conditions
  - Synchronization Techniques
- Proxy Lab
News

- **Proxy** due Tuesday Nov 23\textsuperscript{rd} at 11:59pm

- **DBUG info session**: Saturday Nov 20\textsuperscript{th}, 12-2PM in GHC 4401.
Threading
Multi-Threaded process

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

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

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

Shared resources:

Kernel context:
- VM structures
- Descriptor table

Private Address Space:
- shared libraries
- run-time heap
- writable data
- read-only data
- code
- 0
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

- `pthread_exit()` only terminates the current thread, NOT the process

- `exit()` terminates ALL the threads in the process, i.e., the process itself
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
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
Connection Establishment Functions

- **Server Sockets**
  - `socket(...)`
  - `bind(...)`
  - `listen(...)`
  - `accept(...)`
  - `close(...)`

- **Client Sockets**
  - `socket(...)`
  - `connect(...)`
  - `close(...)`
socket(domain, type, protocol)

int sock_fd= socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);

- **domain** – Protocol Family to use
  - PF_INET is the IPv4 family of protocols

- **type** – Type of protocol to use
  - SOCK_STREAM suggests a steady data stream with guaranteed in-order delivery

- **protocol** – Specific protocol to use
  - IPPROTO_TCP suggests to use TCP (stream-based socket protocol)
bind(sock_fd, my_addr, addrlen)

```c
struct sockaddr_in sock_addr;
memset(&sock_addr, 0, sizeof(sockaddr));
sockaddr.sin_family = AF_INET;
sockaddr.sin_addr.s_addr = INADDR_ANY;
sockaddr.sin_port = htons(listenPort);
err = bind(sock_fd, (struct sockaddr*) sock_addr, sizeof(sockaddr));
```

- **sock_fd**—file descriptor of socket
- **my_addr**—address to which to bind
- **addrlen**—size (in bytes) of address struct
listen(sock_fd, backlog)

err = listen(sock_fd, MAX_WAITING_CONNECTIONS);

- sock_fd—socket on which to listen
- backlog—Maximum size of list of waiting connections
accept(sock_fd, addr, addrlen)

struct sockaddr_in client_addr;
socklen_t my_addr_len = sizeof(client_addr);
client_fd = accept(listener_fd, &client_addr, &my_addr_len);

- sock_fd—listening socket from which to accept connection
- addr—pointer to sockaddrstruct to hold client address
- addrlen—pointer to length of addr that is overwritten with actual length of connection
connect(sock_fd, addr, addrlen)

struct sockaddr_in remote_addr;  
/* initialize remote_addr*/
err = connect(listener_fd, &remote_addr, sizeof(remote_addr));

- sock_fd—socket to connect to
- addr—pointer to sockaddrstruct that holds remote address
- addrlen—length of addrthat is overwritten with actual length of connection
Socket Communication Functions

- `send(sock_fd, buf, buf_len, flags)`
- `recv(sock_fd, buf, max_len, flags)`

Like `read` and `write`, but takes flags. Check man page to use flags.
Proxy Lab

- 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

- Understand what is robust about the rio package
  - Behavior of network sockets

- You may use select, but it will be a lot more work than threads.
What is a proxy?

1) Client → Proxy
   GET http://www.google.com/ HTTP/1.0

2) Proxy → Server
   GET http://www.google.com/ HTTP/1.0

3) Proxy ← Server
   HTTP/1.0 200 OK

4) Client ← Proxy
   HTTP/1.0 200 OK
What is a Caching Proxy

1) Client

GET http://www.google.com/ HTTP/1.0

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

2) Client

HTTP/1.0 200 OK

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?
- Remember you are submitting your files in a compressed folder and therefore edits in your copy of csapp.c/csapp.h should be submitted as well.
**Gethostbyname**

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

```c
/* $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

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

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

```c
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?
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?
On Testing Your Caching Proxy…

- DBUG
- Set up your browser.
Questions?