Concurrent HTTP Proxy with Caching

Course Logistics

- This is the last recitation.
- Final Exam
  - Coming soon, start studying.
  - Comprehensive, slightly focused on recent material.
  - Review old exams from the course website.
- Final Review Session - Thursday
  - The lecture will be led by you.
  - Send us good questions.
  - "Please review subject x" is not a good question!
- Go to office hours this week
  - Schedule one-on-one meetings.

ProxyLab Logistics

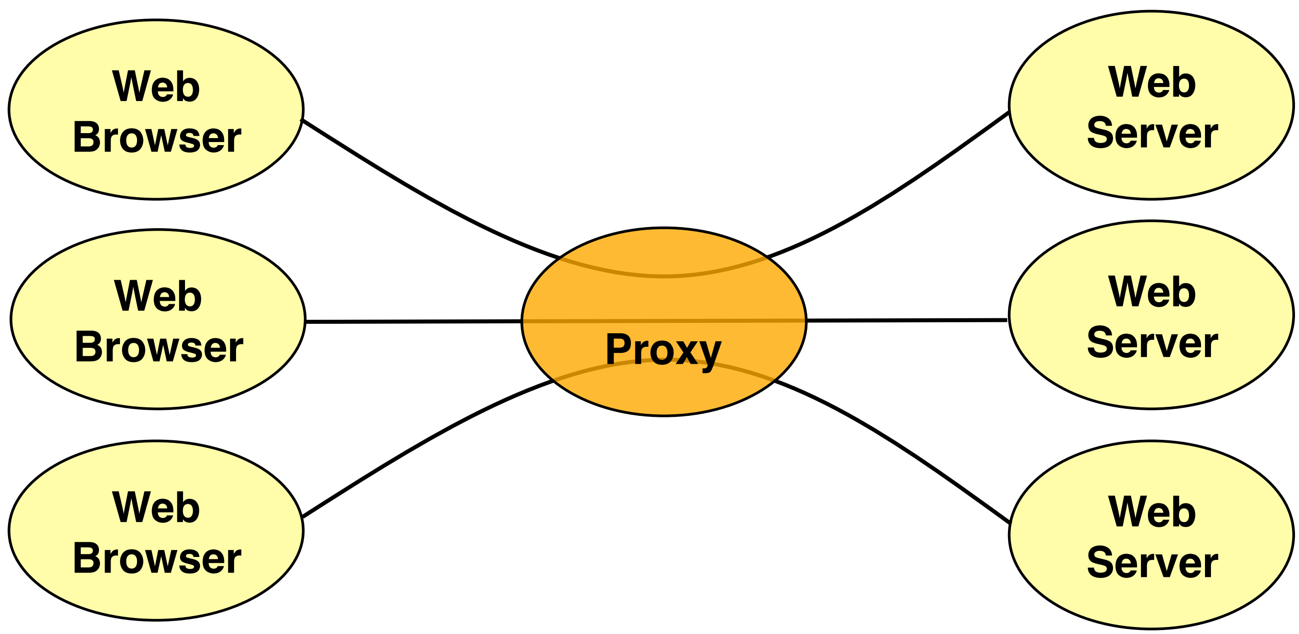
- Due Thursday, drop-dead date is Saturday
- Late Days: minimum of both partners
- Make sure both partners hand in code
- Test your proxy well
  - You may share testing ideas with classmates
  - But not testing code

Outline

- Threads
  - Review of the lecture
- Synchronization
  - Using semaphores; preview of Tue. lecture
- Caching in the proxy
- TA Evaluation Forms

Concurrent Servers

- Iterative servers can only serve one client at a time
- Concurrent servers handle multiple requests in parallel

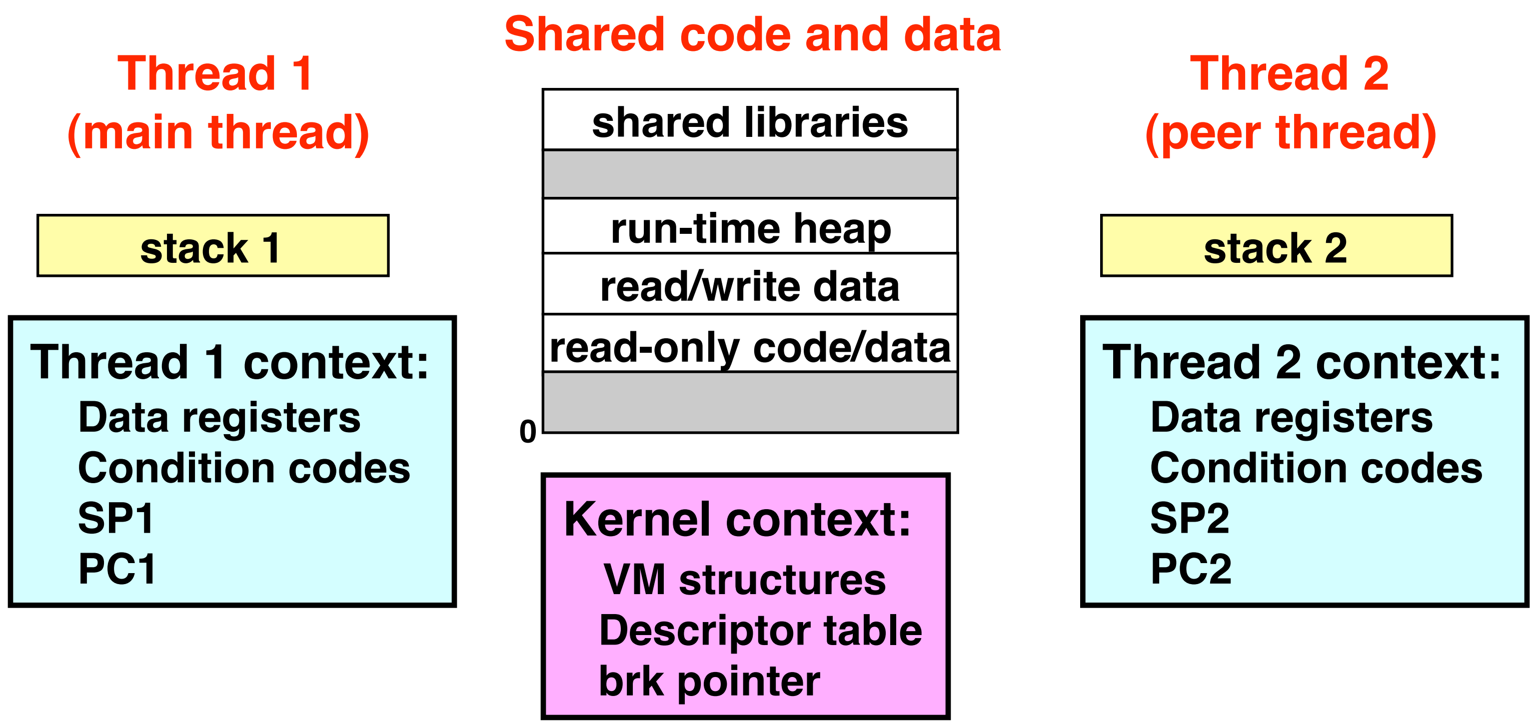


Implementing Concurrency

1. Processes
   - Fork a child process for every incoming client connection
   - Difficult to share data among child processes
2. Threads
   - Create a thread to handle every incoming client connection
   - Our focus today
3. I/O multiplexing with Unix **`select()`**
   - Use **`select()`** to notice pending socket activity
   - Manually interleave the processing of multiple open connections
   - More complex!
     - ~ implement your own app-specific thread package!

A process with Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (instruction flow)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread ID (TID)



Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow.
  - Each can run concurrently.
  - Each is context switched.

- How threads and processes are different
  - Threads share code and data, processes (typically) do not.
  - Threads are less expensive than processes.
    - Process control (creating and reaping) is twice as expensive as thread control.
    - Linux/Pentium III numbers:
      - ~20K cycles to create and reap a process.
      - ~10K cycles to create and reap a thread.

Posix Threads (pthreads)

- Creating and reaping threads
  - `pthread_create`
  - `pthread_join`
  - `pthread_detach`

- Determining your thread ID
  - `pthread_self`

- Terminating threads
  - `pthread_cancel`
  - `pthread_exit`
  - `exit` [terminates all threads]
  - `return` [terminates current thread]

Hello World, with pthreads

```
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

void *thread(void *vargp);

int main() {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

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

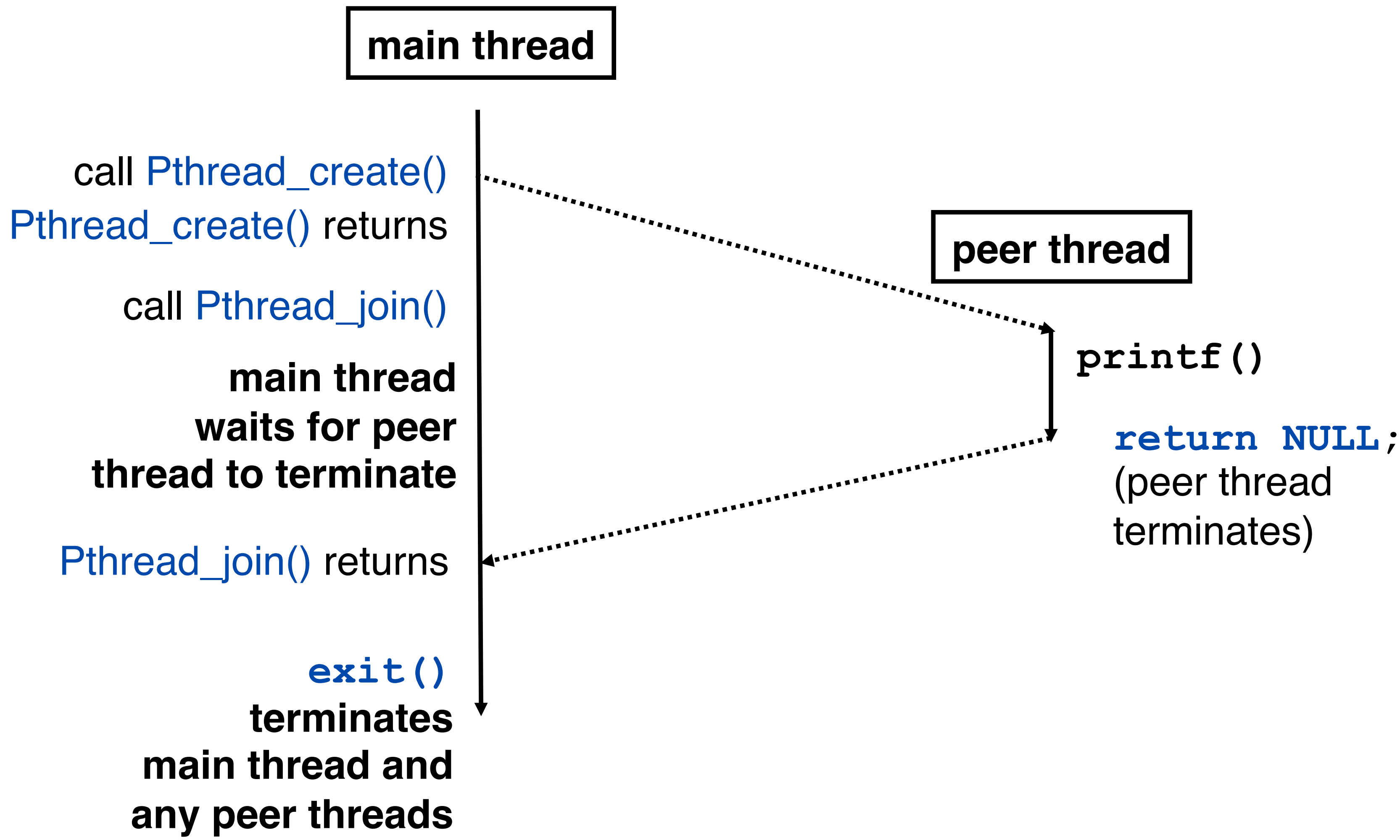
Thread attributes (usually NULL)

Thread arguments (void *p)

return value (void **p)

Upper case Pthread_xxx checks errors

Hello World, with pthreads



Thread-based Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp, port, clientlen;
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,(SA *)&clientaddr,&clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

Thread-based Echo Server

```
/* thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);

    Pthread_detach(pthread_self());
    Free(vargp);

    echo_r(connfd); /* thread-safe version of echo() */
    Close(connfd);
    return NULL;
}
```

?

`pthread_detach()` is recommended in the proxy lab

Issue 1: Detached Threads

A thread is either *joinable* or *detached*

- *Joinable* thread can be reaped or killed by other threads.
  - must be reaped (**`pthread_join`**) to free resources.
- *Detached* thread can't be reaped or killed by other threads.
  - resources are automatically reaped on termination.

- Default state is joinable.
  - **`pthread_detach(pthread_self())`** to make detached.

- *Why should we use detached threads?*
  - ***`pthread_join()`*** *blocks the calling thread*

Issue 2: Avoid Unintended Sharing

```
connfdp = Malloc(sizeof(int));
*connfdp = Accept(listenfd,(SA *)&clientaddr,&clientlen);
Pthread_create(&tid, NULL, thread, connfdp);
```

- What happens if we pass the address of connfd to the thread routine as in the following code?

```
connfd = Accept(listenfd,(SA *)&clientaddr,&clientlen);
Pthread_create(&tid, NULL, thread, (void *)&connfd);
```

Issue 3: Thread-Safe

- Easy to share data structures between threads
  - But we need to do this correctly!
- Recall the shell lab:
  - Job data structures
  - Shared between main process and signal handler
- Synchronize multiple control flows

Synchronizing with Semaphores

- Semaphores are counters for resources shared between threads
  - Non-negative integer synchronization variable
- Two operations: P(s) & V(s)
  - Atomic operations
  - P(s): [`while (s == 0) wait(); s--;`]
  - V(s): [`s++;`]
- If initial value of s == 1
  - Serves as a mutual exclusive lock

Just a very brief description
Details in the next lecture

Sharing with POSIX Semaphores

```
#include "csapp.h"
#define NITERS 1000

unsigned int cnt; /* counter */
sem_t sem;        /* semaphore */

int main() {
    pthread_t tid1, tid2;

    Sem_init(&sem, 0, 1);

    /* create 2 threads and wait */
    ......

    exit(0);
}
```

```
/* thread routine */
void *count(void *arg)
{
  int i;

  for (i=0;i<NITERS;i++){
        P(&sem);
        cnt++;
        V(&sem);
  }
  return NULL;
}
```

Thread-safety of Library Functions

- All functions in the Standard C Library are thread-safe
  - Examples: **`malloc`**, **`free`**, **`printf`**, **`scanf`**

- Most Unix system calls are thread-safe
  - with a few exceptions:

Thread-unsafe function	Reentrant version
`asctime`	`asctime_r`
`ctime`	`ctime_r`
`gethostbyaddr`	`gethostbyaddr_r`
`gethostbyname`	`gethostbyname_r`
`inet_ntoa`	`(none)`
`localtime`	`localtime_r`
`rand`	`rand_r`

Thread-unsafe Functions: Fixes

- Return a ptr to a `static` variable

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

- Fixes:
  1. Rewrite code so caller passes pointer to `struct`
     - Issue: Requires changes in caller and callee

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

Thread-unsafe Functions: Fixes

2. *Lock-and-copy*
   - Issue: Requires only simple changes in caller
   - However, caller must free memory

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

Common Hazards

- Don't hold a lock while making a system call.
- Don't hold a lock when you decide to kill a thread.
- Don't protect huge, complicated blocks of code with a mutex. Limit the amount of code that's protected: this reduces contention and improves performance.
- Be very, very careful to only lock when you DON'T have the mutex, and only unlock when you DO.

Caching

- What should you cache?
  - Complete HTTP response
    - Including headers
  - You don’t need to parse the response
    - But real proxies do. Why?

- If size(response) > MAX_OBJECT_SIZE, don’t cache

Cache Replacement

- **Least Recently Used (LRU)**
  - Evict the cache entry whose "access" timestamp is farthest into the past

- **When to evict?**
  - When you have no space!
  - Size(cache) + size(new_entry)
    > MAX_CACHE_SIZE
  - What is Size (cache)?
    - Sum of size (cache_entries)

Cache Synchronization

- A single cache is shared by all proxy threads
  - Must carefully control access to the cache
- What operations should be locked?
  - `add_cache_entry`
  - `remove_cache_entry`
  - `lookup_cache_entry`
- Remember:
  - Multiple readers can peacefully co-exist
  - But if a writer arrives, that thread MUST synchronize access with others

Summary

- Threading is a clean and efficient way to implement concurrent server
- We need to synchronize multiple threads for concurrent accesses to shared variables
  - Semaphore is one way to do this
  - Thread-safety is the difficult part of thread programming
- Common Symptoms of Concurrency Problems
  - If proxy hangs forever, you're probably forgetting to unlock somewhere
  - IF cache is getting corrupted and returning bad objects, you're probably forgetting to lock somewhere

TA Evaluation Form

- Questions on both sides
- Any comments are highly appreciated!

Thank you!