Concurrent Programming

Kai Shen

1

Concurrent Programming is Hard!

- The human mind tends to be sequential
  - Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
- The notion of time is often misleading
  - Time isn't always fine-grained enough
  - Time across machines or CPUs are not well synchronized

2

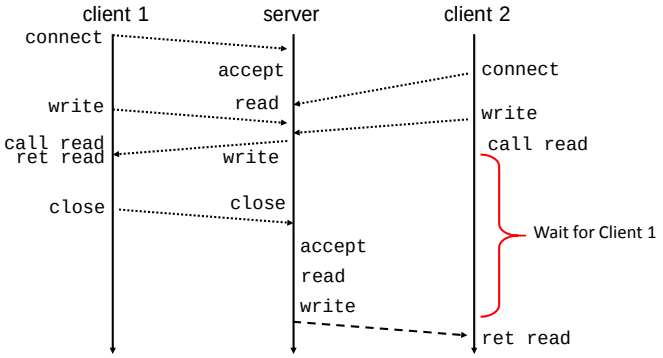
Concurrent Programming is Hard!

- Classical problem classes of concurrent programs:
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Livelock / Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line

3

Iterative Servers

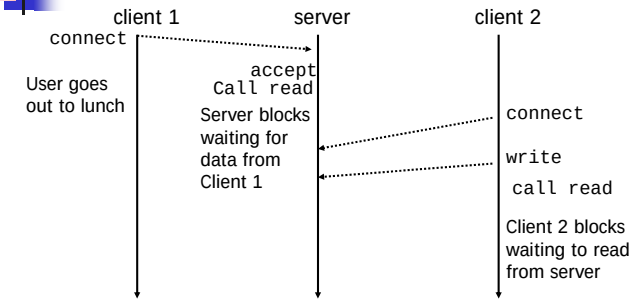
- Iterative servers process one request at a time



- When does client 2 block (on connect or read)?

4

Iterative Server: Slow Response from One Client

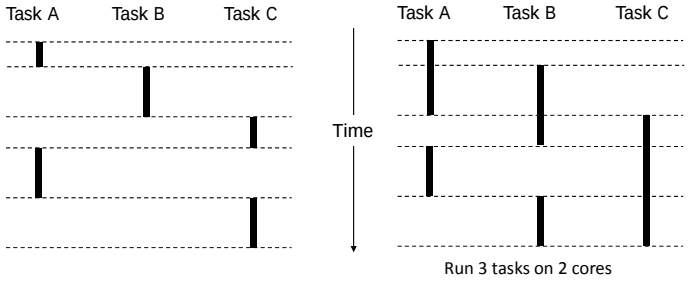


- Solution: use *concurrent servers* instead
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time

5

What is Concurrency?

- Single Core Processor
  - The blocking of one control flow does not block others
- Multi-Core Processor
  - Truly running at the same time



6

Creating Concurrent Flows

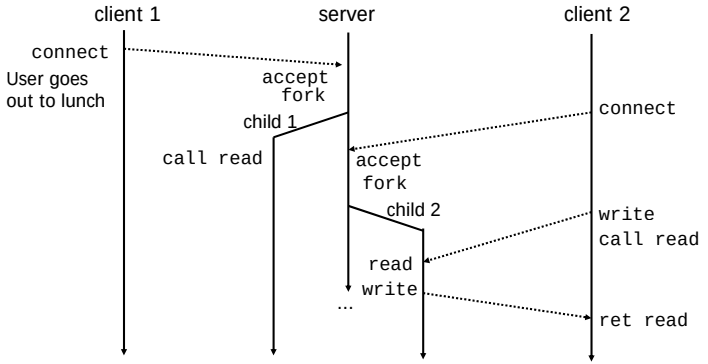
Allow server to handle multiple clients simultaneously

1. Processes
   - The OS automatically interleaves multiple logical flows
   - Each flow has its own private address space
2. Threads
   - The system (OS or not) automatically interleaves multiple logical flows
   - Each flow shares the same address space
3. I/O multiplexing with **`select()`**

7

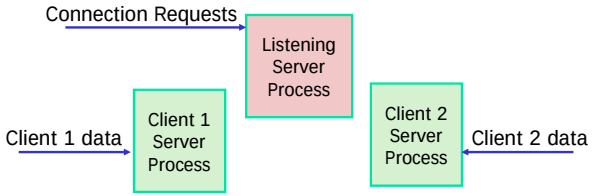
Concurrent Servers: Multiple Processes

- Spawn separate process for each client



8

Socket Connections In Process-Based Concurrent Server



- Both parent & child have copies of listenfd and connfd after forking
  - Properly close them?

9

Closing the Sockets

Fork separate process for each client

```
for (;;) {
    connfd = accept(listenfd, … …);
    if (fork() == 0) {
        close(listenfd); /* Child closes its listening socket */
        … … … …         /* Real work in child */
        close(connfd);   /* Child closes connection with client */
        exit(0);         /* Child exits */
    }
    else {
        close(connfd);   /* Parent closes connected socket */
    }
}
```

What if we don't close sockets properly?

- ⇨ Resource leaking, a critical issue for long-running server.
- ⇨ Is some close more important than others?

10

Reaping Processes for Completed Requests

Fork separate process for each client

```
signal(SIGCHLD, sigchld_handler);

for (;;) {
    connfd = accept(listenfd, … …);
    if (fork() == 0) {
        close(listenfd); /* Child closes its listening socket */
        … … … …         /* Real work in child */
        close(connfd);   /* Child closes connection with client */
        exit(0);         /* Child exits */
    }
    else {
        close(connfd);   /* Parent closes connected socket */
    }
}
```

`sigchld_handler` must reap zombie children to avoid resource leaking

11

Additional Issues of Process-Based Concurrent Server

- Each process has its own memory space
  - Pro or con?
- Significant overhead for process management
  - Process pooling can help.

12

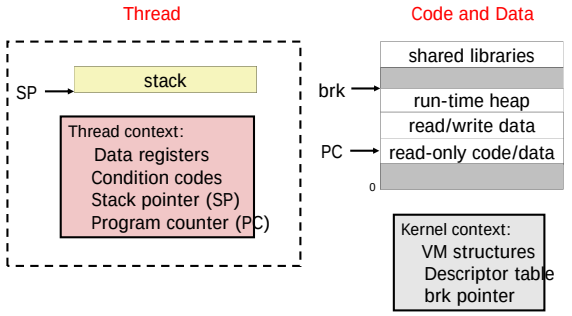
Approach #2: Multiple Threads

- Very similar to approach #1 (multiple processes)
  - But, with threads instead of processes

13

View of A Process

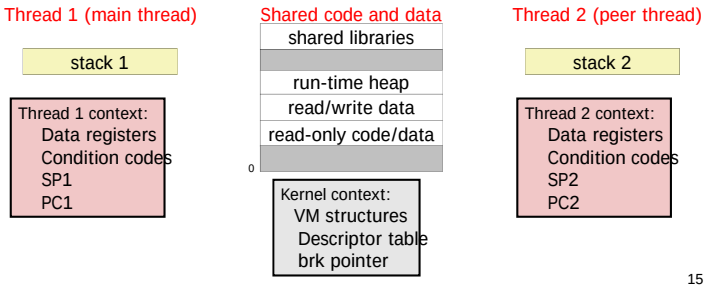
- Process = thread + code, data, and kernel context



14

A Process With Multiple Threads

- Multiple threads can be associated with a process
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
    - Share common virtual address space (inc. stacks)



15

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 do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) is 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

16

Posix Threads (Pthreads) Interface

- *Pthreads:* Standard interface for thread programming
  - 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_cond_init`
    - `pthread_cond_wait`
    - `Pthread_cond_signal`

17

Thread-Based Concurrent Server

```
for (;;) {
    int *connfdp = malloc(sizeof(int));
    *connfdp = accept(listenfd, … …);
    pthread_create(NULL, NULL, my_thread, connfdp);
}
```

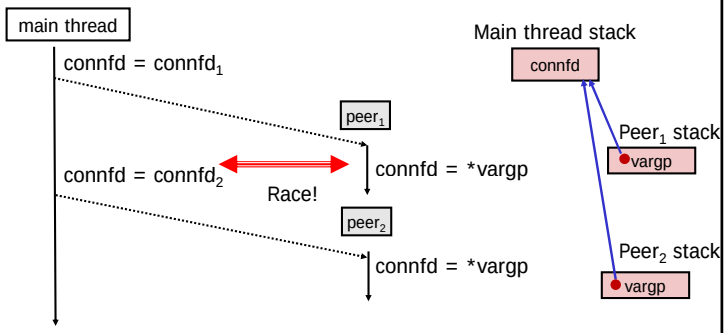
```
void *my_thread(void *vargp) {
    int connfd = *((int *)vargp);
    pthread_detach(pthread_self());
    free(vargp);
    … … … …;   /* your work */
    close(connfd);
    return NULL;
}
```

- Run thread in "detached" mode
  - Runs independently of other threads, reaped when it terminates
- Pass connection file descriptor in heap space
  - Ugly! Note use of malloc()/free() in different contexts!

18

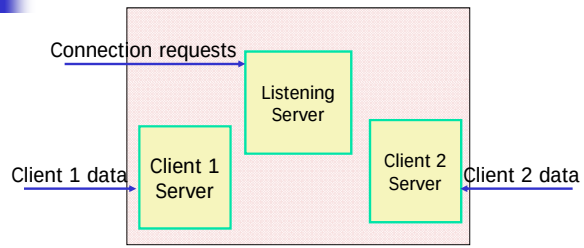
Potential Form of Unintended Sharing

```
for (;;) {
    connfd = accept(listenfd, … …);
    pthread_create(NULL, NULL, my_thread, (void *) &connfd);
}
```



19

Threaded Execution Model



- Multiple threads within single process
- Share state between them
  - Memory
  - File descriptors (don't close a socket twice!)

20

Issues With Thread-Based Servers

- Must run "detached" to avoid memory leak.
  - At any point in time, a thread is either *joinable* or *detached.*
  - *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.
- Must be careful to avoid unintended sharing.
- All functions called by a thread must be *thread-safe*
  - `gethostbyname`
  - `gethostbyname_r`

21

Pros and Cons of Thread-Based Designs

+ Threads are more efficient than processes.

+ Easy to share data structures between threads
  - e.g., logging information, file cache.

– Unintentional sharing can introduce subtle and hard-to-reproduce errors!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads.
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!
  - Topic of "synchronization" will be discussed in next class

– Lack of fault isolation

22

Which form of concurrency to use?

- Apache web server uses process-based concurrency.
- You can try either.
- Yet some high-performance servers that do safe things use neither processes or threads

⇨ Event-based concurrent servers using I/O multiplexing
  - A single thread of control
  - It repeatedly waits on an array of file descriptors (often sockets) and processes any that has available data
    - listening socket and all active connection sockets, …
    - `select()` system call
  - The processing handler must not block. If need to, register an event to wait on and then return to main loop
  - Most efficient on single-core machine

23

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

24