Concurrent Programming

15-213 / 18-213: Introduction to Computer Systems
22nd Lecture, Nov. 11, 2014

Instructors:
Greg Ganger, Greg Kesden, and Dave O’Hallaron
Reminder: Iterative echo server

Client

open_clientfd

getaddrinfo

socket

connect

rio_readlineb

rio_writen

close

Server

getaddrinfo

socket

bind

listen

accept

rio_readlineb

rio_writen

close

open_listenfd

Await connection request from next client

Connection request
Iterative Servers

- Iterative servers process one request at a time.

Diagram:

Client 1:
- Connect
- Write
- Call read
- Ret read
- Close

Server:
- Accept
- Read
- Write
- Ret read
- Accept
- Read
- Write
- Close

Client 2:
- Connect
- Write
- Call read
- Ret read

Wait for server to finish with Client 1.
Fundamental Flaw of Iterative Servers

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

1. Process-based
   - Kernel automatically interleaves multiple logical flows
   - Each flow has its own private address space

2. Event-based
   - Programmer manually interleaves multiple logical flows
   - All flows share the same address space
   - Also referred to as I/O multiplexing.
   - Not covered in lecture (see your textbook)

3. Thread-based
   - Kernel automatically interleaves multiple logical flows
   - Each flow shares the same address space
   - Hybrid of of process-based and event-based.
Approach #1: Process-based Servers

- Spawn separate process for each client

User goes out to lunch

Client 1 blocks waiting for user to type in data

Child blocks waiting for data from Client 1
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);   /* Child services client */
            Close(connfd);  /* Child closes connection with client */
            exit(0);        /* Child exits */
        }
    }
    Close(connfd); /* Parent closes connected socket (important!) */
}

echoserverp.c
Process-Based Concurrent Echo Server (cont)

```c
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
```

d - Reap all zombie children
Concurrent Server: accept Illustrated

1. Server blocks in `accept`, waiting for connection request on listening descriptor `listenfd`

2. Client makes connection request by calling `connect`

3. Server returns `connfd` from `accept`. Forks child to handle client. Connection is now established between `clientfd` and `connfd`
Process-based Server Execution Model

- Each client handled by independent child process
- No shared state between them
- Both parent & child have copies of listenfd and connfd
  - Parent must close \texttt{connfd}
  - Child should close \texttt{listenfd}
Issues with Process-based Servers

- Listening server process must reap zombie children
  - to avoid fatal memory leak

- Listening server process must close its copy of connfd
  - Kernel keeps reference for each socket/open file
  - After fork, refcnt(connfd) = 2
  - Connection will not be closed until refcnt(connfd) == 0
Pros and Cons of Process-based Servers

- + Handle multiple connections concurrently
- + Clean sharing model
  - descriptors (no)
  - file tables (yes)
  - global variables (no)
- + Simple and straightforward
- – Additional overhead for process control
- – Nontrivial to share data between processes
  - Requires IPC (interprocess communication) mechanisms
    - FIFO’s (named pipes), System V shared memory and semaphores
Approach #2: Event-based Servers

- Popular approach for modern high-performance servers
  - E.g., Node.js, nginx, Tornado.

- Not covered here. See your textbook.
Approach #3: Thread-based Servers

- Very similar to approach #1 (process-based)
  - ...but using threads instead of processes
Traditional View of a Process

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

<table>
<thead>
<tr>
<th>Process context</th>
<th>Code, data, and stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program context: Data registers</td>
<td></td>
</tr>
<tr>
<td>Condition codes</td>
<td>Stack</td>
</tr>
<tr>
<td>Stack pointer (SP)</td>
<td>Shared libraries</td>
</tr>
<tr>
<td>Program counter (PC)</td>
<td>Run-time heap</td>
</tr>
<tr>
<td>Kernel context: VM structures</td>
<td></td>
</tr>
<tr>
<td>Descriptor table</td>
<td>Read/write data</td>
</tr>
<tr>
<td>brk pointer</td>
<td>Read-only code/data</td>
</tr>
</tbody>
</table>

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

Kernel context:
- VM structures
- Descriptor table
- brk pointer
Alternate View of a Process

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

Thread (main thread)

- Stack

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

Code, data, and kernel context

- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data
- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer
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
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)

Thread 1 (main thread)

<table>
<thead>
<tr>
<th>Stack 1</th>
<th>Shared code and data</th>
<th>Stack 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shared libraries</td>
<td></td>
</tr>
<tr>
<td></td>
<td>run-time heap</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read/write data</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read-only code/data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kernel context:</td>
<td>Kernel context:</td>
</tr>
<tr>
<td></td>
<td>VM structures</td>
<td>VM structures</td>
</tr>
<tr>
<td></td>
<td>Descriptor table</td>
<td>Descriptor table</td>
</tr>
<tr>
<td></td>
<td>brk pointer</td>
<td>brk pointer</td>
</tr>
</tbody>
</table>

Thread 2 (peer thread)
Logical View of Threads

- Threads associated with process form a pool of peers
  - Unlike processes which form a tree hierarchy

Threads associated with process foo

```
T1 -> T2 -> shared code, data and kernel context -> T4
T5
T3
```

Process hierarchy

```
P0
  ↓
P1
  ↓  sh  sh
  ↓
foo
  ↓
bar
```
Concurrent Threads

- Two threads are *concurrent* if their flows overlap in time
- Otherwise, they are sequential

**Examples:**
- Concurrent: A & B, A&C
- Sequential: B & C

![Diagram showing concurrency and sequentiality of threads](image)
Concurrent Thread Execution

- **Single Core Processor**
  - Simulate parallelism by time slicing

- **Multi-Core Processor**
  - Can have true parallelism

### Diagram

**Thread A**

**Thread B**

**Thread C**

**Time**

Run 3 threads on 2 cores
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 all code and data (except local stacks usually)
  - 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


Posix Threads (Pthreads) Interface

- **Pthreads**: Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads], `RET` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`
The Pthreads "hello, world" Program

/*
 * 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);
}

void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
Execution of Threaded “hello, world”

Main thread

- call Pthread_create()
  - Pthread_create() returns
- call Pthread_join()
  - Main thread waits for peer thread to terminate
- Pthread_join() returns
  - exit()
    - Terminates main thread and any peer threads

Peer thread

- printf()
- return NULL;
  - Peer thread terminates

Terminates main thread and any peer threads
Thread-Based Concurrent Echo Server

```c
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd,
                         (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

- `malloc` of connected descriptor necessary to avoid race
Thread-Based Concurrent Server (cont)

```c
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
Pthread_detach(pthread_self());
Free(vargp);
echo(connfd);
Close(connfd);
return NULL;
}
```

- Run thread in “detached” mode.
  - Runs independently of other threads
  - Reaped automatically (by kernel) when it terminates
- Free storage allocated to hold `connfd`.
- Close `connfd` (important!)
### Thread-based Server Execution Model

- Each client handled by individual peer thread
- Threads share all process state except TID
- Each thread has a separate stack for local variables
Potential Form of Unintended Sharing

```c
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, echo_thread, (void *) &connfd);
}
```

![Diagram showing potential race condition](Image)
Could this race occur?

Main

```c
int i;
for (i = 0; i < 100; i++) {
    Pthread_create(&tid, NULL,
                   thread, &i);
}
```

Thread

```c
void *thread(void *vargp)
{
    int i = *((int *)vargp);
    Pthread_detach(pthread_self());
    save_value(i);
    return NULL;
}
```

### Race Test

- If no race, then each thread would get different value of i
- Set of saved values would consist of one copy each of 0 through 99
Experimental Results

No Race

Single core laptop

Multicore server

The race can really happen!
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**
  - For example, passing pointer to main thread’s stack
    - `Pthread_create(&tid, NULL, thread, (void *)&connfd);

- **All functions called by a thread must be thread-safe**
  - (next lecture)
Pros and Cons of Thread-Based Designs

- Easy to share data structures between threads
  - e.g., logging information, file cache
- Threads are more efficient than processes

- 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 know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low
    - But nonzero!
  - Future lectures
Summary: Approaches to Concurrency

- Processes
  - Hard to share resources: Easy to avoid unintended sharing
  - High overhead in adding/removing clients

- Threads
  - Easy to share resources: Perhaps too easy
  - Medium overhead
  - Not much control over scheduling policies
  - Difficult to debug
    - Event orderings not repeatable

- I/O Multiplexing (covered in textbook)
  - Tedious and low level
  - Total control over scheduling
  - Very low overhead
  - Cannot create as fine grained a level of concurrency
  - Does not make use of multi-core
Additional slides
Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
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

- Many aspects of concurrent programming are beyond the scope of 15-213
  - but, not all 😊
#include "csapp.h"
void echo(int connfd);

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr; /* Enough room for any addr */
    char client_hostname[MAXLINE], client_port[MAXLINE];

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage); /* Important! */
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
Where Does Second Client Block?

- Second client attempts to connect to iterative server

**Client**

- `open_clientfd`
- `socket`
- `connect`
- `rio_writen`
- `rio_readlineb`

**Call to connect returns**
- Even though connection not yet accepted
- Server side TCP manager queues request
- Feature known as “TCP listen backlog”

**Call to rio_writen returns**
- Server side TCP manager buffers input data

**Call to rio_readlineb blocks**
- Server hasn’t written anything for it to read yet.