Concurrent Programming

15-213: Introduction to Computer Systems
23rd Lecture, July 25, 2017

Instructor:
Brian Railing
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
Data Race
Deadlock
Deadlock

- Example from signal handlers.
- Why don’t we use printf in handlers?

```c
void catch_child(int signo) {
    printf("Child exited!\n"); // this call may reenter printf/puts! BAD! DEADLOCK!
    while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}

int main(int argc, char** argv) {
    ...
    for (i = 0; i < 1000000; i++) {
        if (fork() == 0) {
            // in child, exit immediately
            exit(0);
        } // in parent
        sprintf(buf, "Child #%d started\n", i);
        printf("%s", buf);
    }
    return 0;
}
```

Acquire lock  Receive signal

Request lock

curr \_ next
Livelock
Livelock
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 our course..
  - but, not all 😊
  - We’ll cover some of these aspects in the next few lectures.
Concurrent Programming is Hard!

It may be hard, but ...

it can be useful and sometimes necessary!
Reminder: Iterative Echo Server

Client

- socket
- connect
- rio_readlineb
- rio_written
- close

Server

- socket
- bind
- listen
- accept
- rio_readlineb
- rio_written
- close

open_clientfd

Connection request

open_listenfd

Await connection request from next client

Client / Server Session
Iterative Servers

- Iterative servers process one request at a time

![Diagram showing iterative server interactions](image)
Iterative Servers

- Iterative servers process one request at a time

![Diagram of iterative server interactions]

- Wait for server to finish with Client 1
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.
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

Allow server to handle multiple clients concurrently

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
   - Uses technique called *I/O multiplexing*

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

- Spawn separate process for each client

Client 1

- call connect

Client 1 blocks waiting for user to type in data

User goes out to lunch

- call fgets

Child 1

- call read

Child blocks waiting for data from Client 1

Server

- call accept

- ret accept

- fork

- call accept
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
Iterative Echo Server

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

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

- Accept a connection request
- Handle echo requests until client terminates
Making a Concurrent Echo Server

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

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);

        echo(connfd); /* Child services client */
        Close(connfd); /* child closes connection with client */
        exit(0);
    }
}
```
Making a Concurrent Echo Server

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

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {

            echo(connfd);    /* Child services client */
            Close(connfd);    /* Child closes connection with client */
            exit(0);          /* Child exits */

        }
    }
}
```

`echoserverp.c`
Making a Concurrent Echo Server

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

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

Why?
Making a Concurrent Echo Server

int main(int argc, char **argv)
{
  int listenfd, connfd;
  socklen_t clientlen;
  struct sockaddr_storage clientaddr;

  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!) */
}

ehoserverp.c
Process-Based Concurrent Echo Server

```c
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;
}
```

- 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 `connfd`
  - Child should close `listenfd`
Issues with Process-based Servers

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

- **Parent process must close its copy of connfd**
  - Kernel keeps reference count for each socket/open file
  - After fork, $\text{refcnt}(\text{connfd}) = 2$
  - Connection will not be closed until $\text{refcnt}(\text{connfd}) = 0$
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
  - (This example too simple to demonstrate)
Approach #2: Event-based Servers

- **Server maintains set of active connections**
  - Array of connfd’s

- **Repeat:**
  - Determine which descriptors (connfd’s or listenfd) have pending inputs
    - e.g., using `select` function
    - arrival of pending input is an *event*
  - If listenfd has input, then `accept` connection
    - and add new connfd to array
  - Service all connfd’s with pending inputs

- **Details for select-based server in book**
I/O Multiplexed Event Processing

Active Descriptors

| connfd's | 10 | 7 | 4 | -1 | -1 | 12 | 5 | -1 | -1 | -1 |

Pending Inputs

| connfd's | 10 | 7 | 4 | -1 | -1 | 12 | 5 | -1 | -1 | -1 |

Anything happened?

listenfd = 3

Read and service

listenfd = 3
Pros and Cons of Event-based Servers

- + One logical control flow and address space.
- + Can single-step with a debugger.
- + No process or thread control overhead.
  - Design of choice for high-performance Web servers and search engines. e.g., Node.js, nginx, Tornado

- – Significantly more complex to code than process- or thread-based designs.
- – Hard to provide fine-grained concurrency
  - E.g., how to deal with partial HTTP request headers
- – Cannot take advantage of multi-core
  - Single thread of control
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

**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**

- Stack
- Shared libraries
- Run-time heap
- Read/write data
- Read-only code/data

SP → Stack
brk → Shared libraries
PC → Run-time heap
0 → Read/write data
Alternate View of a Process

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

<table>
<thead>
<tr>
<th>Thread (main thread)</th>
<th>Code, data, and kernel context</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack</td>
<td>Shared libraries</td>
</tr>
<tr>
<td>SP</td>
<td>Run-time heap</td>
</tr>
<tr>
<td>Stack context:</td>
<td>Read/write data</td>
</tr>
<tr>
<td>Data registers</td>
<td>Read-only code/data</td>
</tr>
<tr>
<td>Condition codes</td>
<td></td>
</tr>
<tr>
<td>Stack pointer (SP)</td>
<td></td>
</tr>
<tr>
<td>Program counter (PC)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Kernel context:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VM structures</td>
</tr>
<tr>
<td>Descriptor table</td>
</tr>
<tr>
<td>brk pointer</td>
</tr>
</tbody>
</table>
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)  Thread 2 (peer thread)

<table>
<thead>
<tr>
<th>Stack 1</th>
<th>Stack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 1 context:</td>
<td></td>
</tr>
<tr>
<td>Data registers</td>
<td></td>
</tr>
<tr>
<td>Condition codes</td>
<td></td>
</tr>
<tr>
<td>(SP_1)</td>
<td></td>
</tr>
<tr>
<td>(PC_1)</td>
<td></td>
</tr>
<tr>
<td>Thread 2 context:</td>
<td></td>
</tr>
<tr>
<td>Data registers</td>
<td></td>
</tr>
<tr>
<td>Condition codes</td>
<td></td>
</tr>
<tr>
<td>(SP_2)</td>
<td></td>
</tr>
<tr>
<td>(PC_2)</td>
<td></td>
</tr>
</tbody>
</table>

Shared code and data

- shared libraries
- run-time heap
- read/write data
- read-only code/data

Kernel context:
- VM structures
- Descriptor table
- brk pointer
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
- T3
- T4
- T5

**Process hierarchy**

- P0
- P1
  - sh
  - foo
  - bar

**Shared code, data and kernel context**
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
Concurrent Thread Execution

Single Core Processor
- Simulate parallelism by time slicing

Multi-Core Processor
- Can have true parallelism

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)
    - 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]
    - `return` [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(int argc, char** argv)
{
    pthread_t tid;
Pthread_create(&tid, NULL, thread, NULL);
Pthread_join(tid, NULL);
return 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
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);
    }
    return 0;
}
```

- Spawn new thread for each client
- Pass it copy of connection file descriptor
- Note use of `Malloc()`! [but not `Free()`]
/* 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
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)
Potential Form of Unintended Sharing

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

Why would both copies of `vargp` point to same location?
Could this race occur?

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

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!
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

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

- **Event-based**
  - 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

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