Concurrent Programming

Instructor: Adam C. Champion, Ph.D.
CSE 2431: Introduction to Operating Systems
Reading: Sections 12.1–12.3, [CSAPP]
Concurrent Programming is Hard! (1)

• 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! (2)

• 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 this class
Iterative Echo Server

Client

socket()

connect()

write()

read()

close()

Server

socket()

bind()

listen()

accept()

read()

write()

close()

open_clientfd()

open_listenfd()

Await connection request from next client

Client / Server Session

Connection request
Iterative Servers

- Iterative servers process one request at a time.

```
Client 1  Server  Client 2
connect() accept() connect()
write()   read()   write()
read()    write()  Call read()
Return from read()  write()  Call read()
close()   close()  Return from read()
Call read() accept()
Return from read()
Wait for Client 1
```
Where Does Second Client Block?

- Second client attempts to connect to iterative server
  - Call to `connect()` returns
    - Even though connection not yet accepted
    - Server side TCP manager queues request
    - Feature known as “TCP listen backlog”
  - Call to `write()` returns
    - Server side TCP manager buffers input data
  - Call to `read()` 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
Creating Concurrent Flows

Concurrent lets server handle multiple clients at same time

• Processes
  – Kernel automatically interleaves multiple logical flows
  – Each flow has its own private address space

• Threads
  – Kernel automatically interleaves multiple logical flows
  – Each flow shares the same address space

• I/O multiplexing with `select()`
  – Programmer manually interleaves multiple logical flows
  – All flows share the same address space
  – Relies on lower-level system abstractions
Concurrent Servers: Multiple Processes

- Spawn separate process for each client

Client 1

Call connect()
Return from connect()  
Call fgets()

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server

Call accept()
Return from accept()

Call read()

Child 1

Fork()

Child 2

Call connect()

Call fgets()

Write()

Fork()

Call read()

End read()

Close()
Review: Iterative Echo Server

```c
int main(int argc, char **argv) {
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);

    listenfd = Open_listenfd(port); // socket(), bind(), listen()
    while (1) {
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}
```

- Accept a connection request
- Handle echo requests until client terminates
int main(int argc, char **argv) {
    int listenfd, connfd;
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);
    
    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(port);
    while (1) {
        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!) */
    }
}

Fork separate process for each client
Does not allow any communication between different client handlers
Process-Based Concurrent Server (2)

- Signal handler called on each `fork()` (SIGCHLD)
- Reaps all zombie children

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

- Each client handled by independent process
- No shared state between them
- Both parent and child have copies of listenfd, connfd
  - Parent must close connfd
  - Child must close listenfd
Concurrent Server: `accept()` Illustrated

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

2. Client makes connection request by calling and blocking in `connect()`

3. Server returns `connfd` from `accept()`. Forks child to handle client. Client returns from `connect()`. Connection is now established between `clientfd` and `connfd`
Implementation Must-dos With Process-Based Designs

• Listening server process must reap zombie children to avoid fatal memory leaks

• 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`
### View from Server’s TCP Manager

Client 1  Client 2  Server

```
srv> ./echoserverp 15213
```

```
c11> ./echoclient greatwhite.ics.cs.cmu.edu 15213
```

```
srv> connected to (128.2.192.34), port 50437
```

```
c12> ./echoclient greatwhite.ics.cs.cmu.edu 15213
```

```
srv> connected to (128.2.205.225), port 41656
```

<table>
<thead>
<tr>
<th>Connection</th>
<th>Host IP Address</th>
<th>Port</th>
<th>Host IP Address</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listening</td>
<td>---</td>
<td>---</td>
<td>128.2.220.10</td>
<td>15213</td>
</tr>
<tr>
<td>cl1</td>
<td>128.2.192.34</td>
<td>50437</td>
<td>128.2.220.10</td>
<td>15213</td>
</tr>
<tr>
<td>cl2</td>
<td>128.2.205.225</td>
<td>41656</td>
<td>128.2.220.10</td>
<td>15213</td>
</tr>
</tbody>
</table>
View from Server’s TCP Manager

<table>
<thead>
<tr>
<th>Connection</th>
<th>Host</th>
<th>Port</th>
<th>Host</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Listening</td>
<td>---</td>
<td>---</td>
<td>128.2.220.10</td>
<td>15213</td>
</tr>
<tr>
<td>cl1</td>
<td>128.2.192.34</td>
<td>50437</td>
<td>128.2.220.10</td>
<td>15213</td>
</tr>
<tr>
<td>cl2</td>
<td>128.2.205.225</td>
<td>41656</td>
<td>128.2.220.10</td>
<td>15213</td>
</tr>
</tbody>
</table>

- Port Demultiplexing
  - TCP manager maintains separate stream for each connection
    - Each represented to application program as socket
    - New connections directed to listening socket
    - Data from clients directed to one of the connection sockets
Pros, Cons of Process-Based Designs

+ 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 (inter-process communication) mechanisms: FIFOs (named pipes), System V shared memory, and semaphores
Approach #2: Multiple Threads

- Very similar to approach #1 (multiple processes)
  - But with threads instead of processes
Traditional View of a Process

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

Process context

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

Kernel context:
- VM structures
- Descriptor table
- brk pointer

SP → stack
brk → run-time heap
PC → read-only code/data
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)
    - Each thread has its own thread id (TID)

**Thread 1 (main thread)**
- Thread 1 context:
  - Data registers
  - Condition codes
  - SP1
  - PC1

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

**Kernel context:**
- VM structures
- Descriptor table
- brk pointer

**Thread 2 (peer thread)**
- Thread 2 context:
  - Data registers
  - Condition codes
  - SP2
  - PC2

- stack 1
- stack 2
Logical View of Threads

• Threads associated with process form a pool of peers
  – Unlike processes that form a tree hierarchy

Threads associated with process foo

Process hierarchy
Thread Execution

- Single Core Processor
  - Simulate concurrency by time slicing

- Multi-Core Processor
  - Can have true concurrency

Run 3 threads on 2 cores
Logical Concurrency

• Two threads are (logically) concurrent if their flows overlap in time
• Otherwise, they are sequential

• Examples:
  – Concurrent: A & B, A & C
  – Sequential: B & C
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 (typically) do not
  – Threads are somewhat less expensive than processes
    • Process control (creating, reaping) is 2× 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()`
    - `pthread_cond_init()`
    - `pthread_cond_[timed]wait()`
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);
}

/* thread routine */
void *thread(void *vargp) {
    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

(1)

```c
int main(int argc, char **argv) {
    int port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);
    pthread_t tid;

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

– Spawn new thread for each client
– Pass it copy of connection file descriptor
– Note use of `Malloc()`!
  • Without corresponding `Free()`
Thread-Based Concurrent Echo Server

(2)

- Run thread in “detached” mode
  - Runs independently of other threads
  - Reaped when it terminates
- Free storage allocated to hold clientfd
  - “Producer-Consumer” model

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

- Multiple threads within single process
- Some state between them
  - File descriptors
Potential Form of Unintended Sharing

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

• 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.

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;
}
```
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/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
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 are shared and which are private
  – Hard to detect by testing
    • Probability of bad race outcome very low
    • But nonzero!
Event-Based Concurrent Servers Using I/O Multiplexing

• Use library functions to construct scheduler within single process
• Server maintains set of active connections
  – Array of connfd’s
• Repeat:
  – Determine which connections have pending inputs
  – If listenfd has input, then accept connection
    • Add new connfd to array
  – Service all connfd’s with pending inputs
• Uses the `select()` system call
• Details in book
I/O Multiplexed Event Processing

Active Descriptors

<table>
<thead>
<tr>
<th>Clientfd</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>12</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Pending Inputs

<table>
<thead>
<tr>
<th>Clientfd</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>-1</td>
<td>-1</td>
<td>12</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

Active Descriptors Pending Inputs

- Active
  -listenfd = 3

- Inactive
  -listenfd = 3
Pros and Cons of I/O Multiplexing

• + One logical control flow.
• + Can single-step with a debugger.
• + No process or thread control overhead.
  – Design of choice for high-performance Web servers and search engines.
• – Significantly more complex to code than process- or thread-based designs.
• – Hard to provide fine-grained concurrency
  – E.g., our example will hang up with partial lines.
• – Cannot take advantage of multi-core
  – Single thread of control
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
  – 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