Introduction to Computer Systems
15-213/18-243, spring 2009
Lecture, Apr. 23rd

Instructor: Nathan D. Mickulicz
Lecture Outline

- Why do we *need* concurrency?

- How do we *create* concurrency?

- What are some *hazards* associated with concurrent programming?

- I/O Multiplexing and Thread Pools
Lecture Outline

- Why do we need concurrency?

- How do we create concurrency?

- What are some hazards associated with concurrent programming?

- I/O Multiplexing and Thread Pools
Iterative Servers

- Consider the client/server design below:
Iterative Servers

- Let's look at the message flow in the *common use case*:
Iterative Servers

- Does anyone see the problem yet?
Iterative Servers

- Does anyone see the problem yet?
Iterative Servers

- Only one request is served at once!

- Slow clients can slow down the server!

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server blocks waiting for data from Client 1

Client 2 blocks waiting to complete its connection request until after lunch!
Iterative Servers

- Concurrency to the rescue!

Client 1
- call connect
- ret connect
- call fgets

User goes out to lunch

Client 1 blocks waiting for user to type in data

Server
- call accept
- ret accept
- fork
- call accept
- ret accept
- fork

Child 1
- call read

Child 2
- call
- ... read
- write
- close
- end read
- close

Client 2
- call connect
- ret connect
- call fgets
- write
- call read
- close
Lecture Outline

- Why do we need concurrency?

- How do we create concurrency?

- What are some hazards associated with concurrent programming?

- I/O Multiplexing and Thread Pools
In the Beginning...

...there was fork().
Concurreny with fork()

- Clean, straightforward code.

```c
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!) */
}
```
Concurrency with fork()

- Each connection is handled in its own *process*.
  - Clients can execute concurrently!

- File descriptors are shared, memory is *not*.
  - Programming is easy.
  - Communication between child processes is *NOT*.

- Big overhead for process management!
  - A moderately-loaded web server could easily generate 1000 concurrent processes!
Concurrency with fork()

- Impractical for serious concurrent programming.

- **Idea:** multiple, concurrent *threads of execution* inside the *same process*.
  - Communication between threads is as simple as writing to memory.
  - Avoids copying memory and other process management.
Adding Threads to Processes

- The traditional view of a process is the process context and code/data/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**

- **SP** → stack
- **brk** → run-time heap
- **PC** → read-only code/data
- **0** → read/write data
- **0** → shared libraries
Adding Threads to Processes

- Same objects, different organization.
Adding Threads to Processes

- Now we can duplicate the thread while sharing the other data.

Thread 1 (main thread)
- stack 1

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)
- stack 2

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2
“Ok, it seems people are experimenting with threads. That's nice, I had been starting to give up on it completely (I was originally hoping for a threaded X server, but that never happened so now I have to hope that somebody else comes up with a "worthwhile" threaded project).”

- Linus Torvalds
  May 3, 1996
POSIX Threads (pthreads)

- A standard library of functions which allow you to manage threads from C programs.
  - Creating/reaping threads
    - `pthread_create()`
    - `pthread_join()`
    - `pthread_detach()`
  - Determining thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
  - Synchronization objects
POSIX Threads “Hello, World!”

/*
 * 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;
}
POSIX Threads “Hello, World!”

- How is this actually executed?

```
main thread

! Pthread_create() returns

! call Pthread_create()

! main thread waits for peer thread to terminate

! call Pthread_join()

! Pthread_join() returns

! exit() terminates main thread and any peer threads

peer thread

printf()

return NULL;

(peer thread terminates)
```
Concurrent Server with Threads

- The beginnings of a simple Echo server.

```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 a new thread for each client
- Pass in the file descriptor
Concurrent Server with Threads

- The thread procedure for the Echo server.

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

- Detaches from parent – cleanup done on exit.
- Echoes the input back to the client.
- Closes the file descriptor.
So, what was up with that weird malloc/free thing?

- Isn't that a waste of time to run the allocator?
- Just pass in the file descriptor as an argument!
  - But the argument is a pointer...and we wouldn't want to cast integers to pointers!
- Okay, then just pass in a reference to the stack variable!
  - *Hmm...*
Lecture Outline

- Why do we need concurrency?
- How do we create concurrency?
- What are some hazards associated with concurrent programming?
- I/O Multiplexing and Thread Pools
Race Conditions

Here's a small threaded program...

```c
/* shared */
volatile unsigned int cnt = 0;
#define NITERS 100000000

int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                     count, NULL);
    Pthread_create(&tid2, NULL,
                     count, NULL);

    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}

/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```
Race Conditions

...with a really big problem.

```c
/* shared */
volatile unsigned int cnt = 0;
#define NITERS 100000000

int main() {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL,
                   count, NULL);
    Pthread_create(&tid2, NULL,
                   count, NULL);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}

/* thread routine */
void *count(void *arg) {
    int i;
    for (i=0; i<NITERS; i++)
        cnt++;
    return NULL;
}
```

```
linux> ./badcnt
BOOM! cnt=198841183

linux> ./badcnt
BOOM! cnt=198261801

linux> ./badcnt
BOOM! cnt=198269672

cnt should be equal to 200,000,000. What went wrong?!
```
Race Conditions

- Let's dissect the code.

**C code for counter loop**

```c
for (i=0; i<NITERS; i++)
    cnt++;
```

**Corresponding asm code**

```asm
.L9:
    movl -4(%ebp),%eax
    cmpl $99999999,%eax
    jle .L12
    jmp .L10

.L12:
    movl cnt,%eax    # Load
    leal 1(%eax),%edx # Update
    movl %edx,cnt    # Store

.L11:
    movl -4(%ebp),%eax
    leal 1(%eax),%edx
    movl %edx,-4(%ebp)
    jmp .L9

.L10:
```

- Head ($H_i$)
- Load cnt ($L_i$)
- Update cnt ($U_i$)
- Store cnt ($S_i$)
- Tail ($T_i$)
# Race Conditions

- Many different orders of execution, or *interleaving of threads*, are possible...

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr\textsubscript{i}</th>
<th>%eax\textsubscript{1}</th>
<th>%eax\textsubscript{2}</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H\textsubscript{1}</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L\textsubscript{1}</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U\textsubscript{1}</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S\textsubscript{1}</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>H\textsubscript{2}</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>L\textsubscript{2}</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U\textsubscript{2}</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S\textsubscript{2}</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>T\textsubscript{2}</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>T\textsubscript{1}</td>
<td>1</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>
Race Conditions

- ...but *not all are correct!*

<table>
<thead>
<tr>
<th>i (thread)</th>
<th>instr</th>
<th>%eax(_1)</th>
<th>%eax(_2)</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H(_1)</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>L(_1)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U(_1)</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>H(_2)</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>L(_2)</td>
<td>-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>S(_1)</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>T(_1)</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>U(_2)</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>S(_2)</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>T(_2)</td>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Race Conditions

- How to fix these problems?
  - It's not easy!
    - #1 source of concurrent programming headaches.
  - The main technique is to synchronize the threads.
    - Don't allow the threads to run concurrently while modifying shared state!
    - This is a topic for next week.
  - Lets go back to a more familiar context...
Stack-Sharing Hazards

“Just pass a reference to the stack variable!”

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

Why would both copies of vargp point to same location?
Pros and Cons of Threading

**Pros**
- Easy to share data between threads
  - Logging information, file cache, ...
- Threads much more efficient than processes

**Cons**
- Much harder to write programs correctly!!!
  - Unintentional data sharing can introduce subtle and hard-to-reproduce errors!
  - There are ways to deal with this – covered in the next lecture.
Summary of Concurrency

- **Processes**
  - Hard to share resources, easy to avoid unintended sharing.
  - High overhead in adding/removing clients.

- **Threads**
  - Easy to share resources – sometimes too easy!
  - Medium overhead.
  - Not much control over when threads are run.
  - Difficult to debug because thread interleaving is not repeatable!

- **I/O Multiplexing**
  - Next lecture.
Questions

- ???