Programming for Concurrency

15-213/18-243: Introduction to Computer Systems
25th Lecture, 22 April 2010

Instructors:
Bill Nace and Gregory Kesden
Today

- Limitations of iterative servers
- Process-based concurrent servers
- Threads-based concurrent servers
- Event-based concurrent servers
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
- 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 are beyond the scope of 15-213 / 18-243
Echo Server Operation

**Client**
- `open_clientfd`
- `socket`
- `connect`
- `rio_readlineb`
- `close`

**Server**
- `open_listenfd`
- `socket`
- `bind`
- `listen`
- `accept`
- `rio_readlineb`
- `rio_writen`
- `close`

Connection request from next client

Client/Server Session
Iterative Servers

- Iterative servers process one request at a time.

```
client 1          server          client 2
  call connect    call accept    call connect
  ret connect    ret accept
  call write     read
  ret write      close
  close
```

```
client 1          server          client 2
  call connect    call accept    call connect
  ret connect    ret accept
  call write     read
  ret write      close
  close
```

```
client 1          server          client 2
  call connect    call accept    call connect
  ret connect    ret accept
  call write     read
  ret write      close
  close
```
Fundamental Flaw of Iterative Servers

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

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
- call read
- fork
- child 1
- call read

Client 2
- call connect
- ret connect
- call fgets
- fork
- child 2
- call read
- write
- close
- end read
- close
Three Basic Mechanisms for Creating Concurrent Flows

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

2. Threads
   - Kernel automatically interleaves multiple logical flows
   - All flows share the same address space

3. I/O multiplexing with `select()`
   - Programmer manually interleaves multiple logical flows
   - All flows share the same address space
   - Popular for high-performance server designs
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);

    while (1) {
        connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
        echo(connfd);
        Close(connfd);
    }
    exit(0);
}

- Accept a connection request
- Handle echo requests until client terminates
Inner Echo Loop

```c
void echo(int connfd) {
    size_t n;
    char buf[MAXLINE];
    rio_t rio;

    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        printf("server received %d bytes\n", n);
        Rio_writen(connfd, buf, n);
    }
}
```

- Server reads lines of text
- Echos them right back
  - `printf`s a status statement for each
Echo 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. Client returns from `connect`. Connection is now established between clientfd and connfd
Today

- Limitations of iterative servers
- **Process-based concurrent servers**
- Threads-based concurrent servers
- Event-based concurrent servers
Process-Based Concurrent Server

```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!) */
    }
}
```

- Server forks a separate process to deal with each client
- No communication between child processes
**Process-Based Concurrent Server (cont’d)**

- **SIGCHLD** is sent to a process when a child process terminates
- Parent must reap 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
- When child created, each has copy of `listenfd` and `connfd`
  - Parent must close `connfd`, child must close `listenfd`
Implementation Issues

- Server must reap zombie children
  - to avoid fatal memory leak
- Server must close its copy of connfd
  - Kernel keeps reference count for each socket/open file
  - After fork, reference count on connfd is 2
  - Connection will not be closed until reference count is zero
Pros and Cons of Process-Based Designs

+ Handles 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
Today

- Limitations of iterative servers
- Process-based concurrent servers
- Threads-based concurrent servers
- Event-based concurrent servers
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
Alternate View of a Process

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

**Thread context**
- 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
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
  - Each thread has its own thread id (TID)

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

Thread 2 (peer thread)
- stack 2
- Kernel context:
  - VM structures
  - Descriptor table
  - brk pointer

brk
run-time heap
read/write data
read-only code/data

SP
PC
0

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

Shared code, data, and kernel context

Process hierarchy

- P0
- P1
- foo
- bar
- sh
- sh
- sh
Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently
  - Each is context switched

- How threads and processes are different
  - Threads share code and data, processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) is twice as expensive as thread control
    - Linux/Pentium III numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles to create and reap a thread
Posix Threads (Pthreads) Interface

- **Pthreads**: Standard C interface with ~100 functions that manipulate threads
  - 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;
}

Thread attributes (usually NULL)
Thread arguments (void *p)
return value (void **p)
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 port = atoi(argv[1]);
    struct sockaddr_in clientaddr;
    int clientlen = sizeof(clientaddr);
    pthread_t tid;

    int listenfd = Open_listened(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 Server (cont)

/* thread routine */
void *echo_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 when it terminates

- **Free storage allocated to hold clientfd**
  - “Producer-Consumer” model
Threaded-Process Execution Model

- Multiple threads within single process
- Some state between them
  - File descriptors (in this example; usually more)
while (1) {
    int connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    Pthread_create(&tid, NULL, echo_thread, (void *) &connfd);
}
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, what happens if we pass the address of `connfd` to the thread routine?
    - `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
  - (next lecture)
Today

- Limitations of iterative servers
- Process-based concurrent servers
- Threads-based concurrent servers
- Event-based concurrent servers
Event-Based Concurrent Servers Using I/O Multiplexing

- Maintain a pool of connected descriptors
- Repeat the following forever:
  - Use the Unix `select()` function to block until:
    - (a) New connection request arrives on the listening descriptor
    - (b) New data arrives on an existing connected descriptor
  - If (a), add the new connection to the pool of connections
  - If (b), read any available data from the connection
    - Close connection on EOF and remove it from the pool
The select() Function

- `select()` sleeps until one or more file descriptors in the set `readset` ready for reading

```
#include <sys/select.h>

int select(int maxfdp1, fd_set *readset, NULL, NULL, NULL);
```

- `readset`
  - Opaque bit vector that indicates membership in a descriptor set
  - If bit $k$ is 1, then descriptor $k$ is a member of the descriptor set

- `maxfdp1`
  - Maximum descriptor in descriptor set plus 1
  - Tests descriptors 0, 1, 2, ..., `maxfdp1` - 1 for set membership

- `select()` returns the number of ready descriptors and sets each bit of `readset` to indicate the ready status of its corresponding descriptor
Macros for Manipulating Set Descriptors

```c
void FD_ZERO(fd_set *fdset);
  ▪ Turn off all bits in fdset

void FD_SET(fd_set *fdset);
  ▪ Turn on all bits in fdset

void FD_CLR(int fd, fd_set *fdset);
  ▪ Turn off bit fd in fdset

int FD_ISSET(int fd, *fdset);
  ▪ Is bit fd in fdset turned on?
```
Overall Structure

- **Manage Pool of Connections**
  - `listenfd`: Listen for requests from new clients
  - Active clients: Ones with a valid connection
    - file descriptors kept in `clientfd[]`
- **Use `select()` to detect activity**
  - New request on `listenfd`
  - Request by active client
- **Required Activities**
  - Adding new clients
  - Removing terminated clients
  - Echoing

### Overall Structure Table

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

- **Active**
- **Inactive**
- **Never Used**
Representing a Pool of Clients

```c
/*
 * echoservers.c - A concurrent echo server based on select
 */
#include "csapp.h"

typedef struct { /* represents a pool of connected descriptors */
    int maxfd;    /* largest descriptor in read_set */
    fd_set read_set; /* set of all active descriptors */
    fd_set ready_set; /* subset of descriptors ready for reading */
    int nready;    /* number of ready descriptors from select */
    int maxi;      /* highwater index into client array */
    int clientfd[FD_SETSIZE]; /* set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* set of active read buffers */
} pool;

int byte_cnt = 0; /* counts total bytes received by server */
```
Pool Example

- maxfd = 12
- maxi = 6
- read_set = {3,4,5,7,10,12}

listenfd

cliend

<table>
<thead>
<tr>
<th>clientfd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>
Main Loop

```c
int main(int argc, char **argv)
{
    int listenfd, connfd, clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    static pool pool;

    listenfd = Open_listenfd(argv[1]);
    init_pool(listenfd, &pool);

    while (1) {
        pool.ready_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.ready_set,
                               NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &pool.ready_set)) {
            connfd = Accept(listenfd, (SA *)&clientaddr,&clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool);
    }
}
```
Pool Initialization

/* initialize the descriptor pool */
void init_pool(int listenfd, pool *p)
{
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}
**Initial Pool**

- \( \text{maxfd} = 3 \)
- \( \text{maxi} = -1 \)
- \( \text{read\_set} = \{ 3 \} \)

<table>
<thead>
<tr>
<th>clientfd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

listenfd

\[ 3 \]
Adding Client

```c
void add_client(int connfd, pool *p) /* add connfd to pool p */
{
  int i;
  p->nready--;

  for (i = 0; i < FD_SETSIZE; i++) /* Find available slot */
    if (p->clientfd[i] < 0) {
      p->clientfd[i] = connfd;
      Rio_readinitb(&p->clientrio[i], connfd);
      FD_SET(connfd, &p->read_set); /* Add desc to read set */

      if (connfd > p->maxfd) /* Update max descriptor num */
        p->maxfd = connfd;
      if (i > p->maxi) /* Update pool high water mark */
        p->maxi = i;
      break;
    }
  if (i == FD_SETSIZE) /* Couldn't find an empty slot */
    app_error("add_client error: Too many clients");
}```
Adding Client with fd 11

- **maxfd** = 12
- **maxi** = 6
- **read_set** = \{3, 4, 5, 7, 10, 11, 12\}

<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>

| listenfd | 3 |

<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>11</td>
<td>-1</td>
<td>12</td>
<td>5</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>
Checking Clients

/* echo line from ready descs in pool p */
void check_clients(pool *p) {
  int i, connfd, n;
  char buf[MAXLINE];
  rio_t rio;

  for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
    connfd = p->clientfd[i];
    rio = p->clientrio[i];

    /* If the descriptor is ready, echo a text line from it */
    if ((connfd > 0) && (FD_ISSET(connfd, &p->read_set))) {
      p->nready--;
      if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        byte_cnt += n;
        Rio_writen(connfd, buf, n);
      } else {/* EOF detected, remove descriptor from pool */
        Close(connfd);
        FD_CLR(connfd, &p->read_set);
        p->clientfd[i] = -1;
      }
    }
  }
}
Concurrency Limitations

- Current design will hang up if partial line transmitted
- Bad to have network code that can hang up if client does something weird
  - By mistake or maliciously

- Would require more work to implement more robust version
  - Must allow each read to return only part of line, and reassemble lines within server

```c
if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
    p->nready--;
    if (n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        byte_cnt += n;
        Rio_writen(connfd, buf, n);
    }
}
```
Pro and Cons of Event-Based Designs

+ 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

- More complex code than process- or thread-based designs
- Hard to provide fine-grained concurrency
  - E.g., our example will hang up with partial lines
- Does not make use of multiple cores
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
  - Single thread – single core
Today

- Limitations of iterative servers
- Process-based concurrent servers
- Threads-based concurrent servers
- Event-based concurrent servers

Next Time:
- Synchronization