Concurrent Programming

December 1, 2006

Topics

- Limitations of iterative servers
- Process-based concurrent servers
- Event-based concurrent servers
- Threads-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 elsewhere in the system
  - Example: who gets the last seat on the airplane?
- Deadlock: improper resource allocation prevents forward progress
  - Example: traffic gridlock
- Lifelock / 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

Echo Server Operation

Iterative Servers

Iterative servers process one request at a time.
Fundamental Flaw of Iterative Servers

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

Concurrent Servers: Multiple Processes

Concurrent servers handle multiple requests concurrently.

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.
- Each flow shares the same address space.

3. I/O multiplexing with select()
- User manually interleaves multiple logical flows.
- Each flow shares the same address space.
- Popular for high-performance server designs.

Review: Sequential 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);
    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

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`.

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!) */
    }
}
```

- Fork separate process for each client
- Does not allow any communication between different client handlers

Process-Based Concurrent Server (cont)

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

- Reap all zombie children
Process Execution Model

- Each client handled by independent process
- No shared state between them
- When child created, each have copies of listenfd and connfd
  - Parent must close connfd, child must close listenfd

Implementation Issues With Process-Based Designs

Server must reap zombie children
- to avoid fatal memory leak.

Server must close its copy of connfd.
- Kernel keeps reference for each socket.
- After fork, refcnt(connfd) = 2.
- Connection will not be closed until refcnt(connfd) = 0.

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

Traditional View of a Process

Process = process context + code, data, and stack

- 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

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)

Logical View of Threads

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

Concurrent Thread Execution

Two threads run concurrently (are concurrent) if their logical 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.
- 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 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

```c
/*
 * 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);
    return 0;
}

/* thread routine */
void *thread(void *vargp) {
    printf("Hello, world!\n");
    return NULL;
}
```

Execution of Threaded "hello, world"

- call `Pthread_create()`
  - `Pthread_create()` returns
- call `Pthread_join()`
  - `Pthread_join()` returns
  - `exit()` terminates main thread and any peer threads

Main thread waits for peer thread to terminate

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_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 Server (cont)

```c
/* 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

Process Execution Model

- Multiple threads within single process
- Some state between them
  - File descriptors

Potential Form of Unintended Sharing

```c
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)

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 (max FD_SETSIZE bits) 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(int fd, fd_set *fdset);
   // Turn on bit fd 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

- **listenfd**: Listen for requests from new clients
- **Active clients**: Ones with a valid connection
- **New request on listenfd**
- **Request by active client**
- **Use select to detect activity**
- **Adding new clients**
- **Removing terminated clients**
- **Echoing**

Representing 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

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

```c
/* 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

```
maxfd = 3
maxi = -1
read_set = { 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**

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

**Concurrent 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

**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.
- 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.
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