15-213
"The course that gives CMU its Zip!"

Concurrent Programming
November 20, 2003

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 decisions elsewhere in the system
  - Deadlock: improper resource allocation prevents forward progress
  - Lifelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress
- Many aspects of concurrent programming are beyond the scope of 15-213

Iterative Servers
Iterative servers process one request at a time.

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 interleave multiple logical flows.
   - Each flow has its own private address space.

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

3. Threads
   - Kernel automatically interleave multiple logical flows.
   - Each flow shares the same address space.
   - Hybrid of processes and I/O multiplexing!

---

Process-Based Concurrent Server

```c
/*
 * echoserver.c - A concurrent echo server based on processes
 * Usage: echoserver <port>
 *
 * include <sys.h>
 * define SIZE 1024
 * void echo(int connfd);
 * void handler(int sig);
 * int main(int argc, char **argv) {
 *     int listenfd, connfd;
 *     int portno;
 *     struct sockaddr_in clientaddr;
 *     int clientlen = sizeof(struct sockaddr_in);
 *     if (argc != 2) {
 *         fprintf(stderr, "usage: %s <port>\n", argv[0]);
 *         exit(0);
 *     }
 *     portno = atoi(argv[1]);
 *     listenfd = open_listenfd(portno);
 *     echo(connfd, &clientaddr, &clientlen);
 *     while (1) {
 *         connfd = Accept(listenfd, &clientaddr, &clientlen);
 *         if (Fork() == 0) {
 *             Close(listenfd); /* child closes its listening socket */
 *             echo(connfd); /* child reads and echoes input line */
 *             Close(connfd); /* child is done with this client */
 *             exit(0); /* child exits */
 *         }
 *         Close(connfd); /* parent must close connected socket */
 *     }
 * }
 */
```
Process-Based Concurrent Server (cont)

```c
/* handler - reaps children as they terminate */
void handler(int sig)
{
    pid_t pid;
    int status;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
        return;
}
```

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

I/O multiplexing provides more control with less overhead...

Implementation Issues With Process-Based Designs

Server should restart accept call if it is interrupted by a transfer of control to the SIGCHLD handler

- Not necessary for systems with POSIX signal handling.
  - Our signal wrapper tells kernel to automatically restart accept.

- Required for portability on some older Unix systems.

Server must reap zombie children

- to avoid fatal memory leak.

Server must close its copy of confd.

- Kernel keeps reference for each socket.
- Afterfork, refcnt (confd) = 2.
- Connection will not be closed until refcnt (confd) = 0.

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` are ready for reading.

```
#include <sys/select.h>
int select(int maxfdpl, 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.

`maxfdpl` - Maximum descriptor in descriptor set plus 1.
- Tests descriptors 0, 1, 2, ..., `maxfdpl` - 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**

- `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?

---

**select Example**

```c
/* main loop: wait for connection request or stdin command.
 * If connection request, then echo input line
 * and close connection. If stdin command, then process.
 */
printf("server > ");
fflush(stdout);
while (notdone) {
    * select: check if the user typed something to stdin or
    * if a connection request arrived.
    FD_ZERO(&readfds);
    FD_SET(listenfd, &readfds); /* add socket fd */
    FD_SET(0, &readfds);        /* add stdin fd (0) */
    Select(&listenfd, &readfds, NULL, NULL, NULL);
```

---

**select Example (cont)**

First we check for a pending event on stdin.

```c
/* if the user has typed a command, process it */
if (FD_ISSET(0, &readfds)) {
    fgets(buf, BUFSIZE, stdin);
    switch (buf[0]) {
    case 'c': /* print the connection count */
        printf("Received %d conn. requests so far.\n", connectcnt);
        printf("server > ");
        fflush(stdout);
        break;
    case 'q': /* terminate the server */
        notdone = true;
        break;
    default: /* bad input */
        printf("ERROR: unknown command\n");
        printf("server > ");
        fflush(stdout);
    }
}  
```
**select Example (cont)**

Next we check for a pending connection request.

```c
/* if a connection request has arrived, process it */
if (FD_ISSET(listenfd, &readfds)) {
    connfd = Accept(listenfd, (struct sockaddr *) &clientaddr, &clientlen);
    connectcnt++;
    memcpy(buf, BUFSIZE);
    Ric_read(connfd, buf, BUFSIZE);
    Ric_write(connfd, buf, strlen(buf));
    Close(connfd);
} while */
```

---

**Event-based Concurrent Echo Server**

```c
/* echoservers.c - A concurrent echo server based on select */
#include "caspp.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 */
    size_t clientcnt[FD_SETSIZE]; /* size of descriptors */
} pool;

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

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

    listenfd = Open_listenfd(argc[1]);
    init_pool(listenfd, 4pool);

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

        if (FD_ISSET(listenfd, pool.ready_set)) {
            connfd = Accept(listenfd, (struct sockaddr *) &clientaddr, &clientlen);
            add_client(connfd, pool);
        }
        check_clients(pool);
    }
    } /* main */
```

---

**Event-based Concurrent Server (cont)**

```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_SET(listenfd, &p->read_set);
    FD_SET(listenfd, &p->write_set);
    FD_SET(listenfd, &p->except_set);
}
```
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.
- Can be vulnerable to denial of service attack
  - How?

Threads provide a middle ground between processes and I/O multiplexing...

Traditional View of a Process

Process = process context + code, data, and stack
Alternate View of a Process

Process = thread + code, data, and kernel context

Thread (main thread)  Code and Data

- stack
- Thread context: Data registers, Condition codes, Stack pointer (SP), Program counter (PC)
- Kernel context: VM structures, Descriptor table, brk pointer

- shared libraries
- run-time heap
- read/write data
- 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 (sequence of PC values)
- Each thread shares the same code, data, and kernel context
- Each thread has its own thread id (TID)

Thread 1 (main thread)  Shared code and data  Thread 2 (peer thread)

- stack 1
- Thread 1 context: Data registers, Condition codes, SP1, PC1
- Kernel context: VM structures, Descriptor table, brk pointer

- stack 2
- Thread 2 context: Data registers, Condition codes, SP2, PC2

Logical View of Threads

Threads associated with a 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], _exit [terminates current thread]
- Synchronizing access to shared variables
  - pthread_mutex_init
  - pthread_mutex_unlock
  - pthread_mutex_init
  - pthread_mutex_timedwait

The Pthreads "hello, world" Program

```c
/* hello.c - Pthreads "hello, world" program */
#include "csapp.h"

void *thread(void *argp);

int main(int argc, char *argv[])
{
    pthread_t tid;
    pthread_create(&tid, NULL, thread, NULL);
    pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *argp)
{
    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() returns
  - _exit() returns NULL; (peer thread terminates)
Thread-Based Concurrent Echo Server

```c
int main(int argc, char **argv)
{
    int listenfd, *connfdp, port, clientlen;
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if (argc != 2) {
        printf(stderr, "usage: %s <port\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        clientaddr = malloc(sizeof(int));
        *clientaddr = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        pthread_create(&tid, NULL, thread, (void *)connfdp);
    }
}
```

Thread-Based Concurrent Server (cont)

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

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
  - 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
  - *(void *)connfdp;

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)