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

Concurrent Servers
April 29, 2003

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

Tips for Completing Lab 7
- Get the latest copies of csapp.c and driver.pl
- Use Rio library, but for https, don't use Rio_readn, Rio_readnb or Rio_readlineb.
- Ignore SIGPIPE signals by installing “ignore” handler signal(SIGPIPE,SIG_IGN)
- When EOF detected while reading server socket, send EOF to client using shutdown(clientfd,1), and vice versa.
- Pass all request headers received from client on to server.
- For https, must read client and server sockets simultaneously.

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**

Concurrent servers handle multiple requests concurrently.

User goes out to lunch

Client 1 blocks waiting for user to type in data

---

**Three Basic Mechanisms for Creating Concurrent Flows**

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

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

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

---

**Process-Based Concurrent Server**

```c
/* echoserverp.c - A concurrent echo server based on processes
 * Usage: echoserverp <port>
 */
#include <ics.h>
define KFSIZE 1024

#include <ics.h>
define KFSIZE 1024
void echo(int connfd);
void echo(int connfd, int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "usage: \%s <port>\n", argv[0]);
  exit(0);
  }
}

int main(int argc, char **argv) {
  printf("%s\n", argv[0]);
  exit(0);
  }
  portno = atoi(argv[1]);
  listenfd = open_listenfd(portno);
```

---

**Process-Based Concurrent Server (cont)**

```c
Signal(SIGCHILD, handler); /* parent must reap children! */

/* main server loop */
while (1) {
  connfd = Accept(listenfd, (struct sockaddr *) &clientaddr, &clientlen);
  if (Fork() == 0) {
    Close(listenfd); /* child closes its listening socket */
    if (Read(connfd) == 0) {
      /* 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 stat;
    while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
        ;
    return;
}
```

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 restart**
- to avoid fatal memory leak.

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

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

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.

```c
#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?
```

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+1, &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, BUF_SIZE, 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 = 0;
    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++;
    bzero(buf, BUFSIZE);
    Rio_readn(connfd, buf, BUFSIZE);
    Rio_writev(connfd, buf, strlen(buf));
    Close(connfd);
}
} /* while */
```

Event-based Concurrent Echo Server

/*
 * 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; /* set of descriptors ready for reading */
    fd_set writable_set; /* set of descriptors ready for writing */
    int nready; /* number of ready descriptors from select */
    int maxi; /* highwater index into client array */
    int clientfd[FDS_MAX]; /* set of active descriptors */
    rio_t *clientrio[FDS_MAX]; /* set of active read buffers */
} pool;

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

Event-based Concurrent Server (cont)

```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);
    }
}
```

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

```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(4p->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");
}
```

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

<table>
<thead>
<tr>
<th>Process context</th>
<th>Code, data, and stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program context:</td>
<td></td>
</tr>
<tr>
<td>Data registers</td>
<td></td>
</tr>
<tr>
<td>Condition codes</td>
<td></td>
</tr>
<tr>
<td>Stack pointer (SP)</td>
<td></td>
</tr>
<tr>
<td>Program counter (PC)</td>
<td></td>
</tr>
<tr>
<td>Kernel context:</td>
<td></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>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td></td>
</tr>
<tr>
<td>shared libraries</td>
<td></td>
</tr>
<tr>
<td>run-time heap</td>
<td></td>
</tr>
<tr>
<td>read/write data</td>
<td></td>
</tr>
<tr>
<td>read-only code/data</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Alternate View of a Process

Process = thread + code, data, and kernel context

<table>
<thead>
<tr>
<th>Thread (main thread)</th>
<th>Code and Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>stack</td>
</tr>
<tr>
<td>brk</td>
<td>run-time heap</td>
</tr>
<tr>
<td></td>
<td>read/write data</td>
</tr>
<tr>
<td></td>
<td>read-only code/data</td>
</tr>
</tbody>
</table>

Thread context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernl 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 (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
- Stack 1
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

Thread 2 (peer thread):
- Stack 2
  - shared libraries
  - run-time heap
  - read/write data
  - read-only code/data

Logical View of Threads

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

Threads associated with process foo:
- T1
- T2
- T3
- T4

Process hierarchy:
- P0
- P1
- sh
- sh
- sh
- foo
- bar

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

Time

Thread A

Thread B

Thread 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_[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);
    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

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

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    listenfd = open_listenfd(port);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfp = malloc(clientlen);
        *connfp = accept(listenfd, (SA *) &clientaddr, &clientlen);
        pthread_create(&tid, NULL, thread, connfp);
    }
}
```

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 *)connfp);

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)