Concurrent Servers

CS 475

Echo Server Operation

Client
- socket
- open_clientfd
- connect
- rio_readlineb
- rio_writen
- close

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

Connection request
Await connection request from next client
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 (approach #1): Multiple Processes

Concurrent servers handle multiple requests concurrently

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

Child 1
- call read
- fork

Child 2
- call read
- write
- close

Client 2
- call connect
- ret connect
- call fgets

- 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
   - Each flow shares 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
Review: Sequential Echo 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

---

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 Must-dos With Process-Based Designs

Listening server process must reap zombie children
  - to avoid fatal memory leak

Listening server process must close its copy of connfd
  - Kernel keeps reference for each socket/open file
  - After fork, refcnt(connfd) = 2
  - Connection will not be closed until refcnt(connfd) == 0

Pros and Cons of Process-Based Designs

+ Handle 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
Approach #2: Multiple Threads

Very similar to approach #1 (multiple processes)
- but, with threads instead of processes

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 (inc. stacks)
- Each thread has its own thread id (TID)

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

Thread 2 (peer thread)
- Stack 2
- Thread 2 context:
  - Data registers
  - Condition codes
  - SP2
  - PC2

Kernel context:
- VM structures
- Descriptor table
- Brk pointer
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 (in this example; usually more)

Potential Form of Unintended Sharing

```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?
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 connd to the thread routine?
  - pthread_create(&tid, NULL, thread, (void *)&connd);

All functions called by a thread must be thread-safe

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
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

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

clientfd

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ...
|---|---|---|---|---|---|---|---|---|---|---
| -1 | 10 | 4 | -1 | -1 | 12 | 5 | -1 | -1 | -1 | Never Used

Active

Inactive

Manage Pool of Connections
- listenfd: Listen for requests from new clients
- Active clients: Ones with a valid connection

Use select to detect activity
- New request on listenfd
- Request by active client

Required Activities
- Adding new clients
- Removing terminated clients
- Echoing
Representing Pool of Clients

/*
 * 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 */
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 \}

```
listenfd = 3

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

<table>
<thead>
<tr>
<th></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>clf</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
</tbody>
</table>

... (continues)

### 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);
    }
}
```
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**

<table>
<thead>
<tr>
<th>clientfd</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9 10 11</td>
</tr>
</tbody>
</table>

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

- **Active**
- **Inactive**
- **Active**
- **Never Used**
Checking Clients

```c
void check_clients(pool *p) { /* echo line from ready descs in 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->ready_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;
            }
        }
    }
}
```

Concurrences Limitations

- Current design will get stuck if partial line transmitted
- Bad to have network code that can get stuck 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