15-213

"The course that gives CMU its Zip!"

Concurrent Servers

December 7, 2000

Topics

- **Baseline iterative server**
- **Process-based concurrent server**
- **Threads-based concurrent server**
- **`select`-based concurrent server**

Error-handling sockets wrappers

To simplify our code, we will use error handling wrappers of the form:

```
int Accept(int s, struct sockaddr *addr, int *addrlen) {
  int rc = accept(s, addr, addrlen);
  if (rc < 0)
    unix_error("Accept");
  return rc;
}
```

```
void unix_error(char *msg) {
  printf("%s: %s\n", msg, strerror(errno));
  exit(0);
};
```


Echo client revisited

```
/*
 * echoclient.c - A simple connection-based echo client
 * usage: echoclient <host> <port>
 */
#include <ics.h>
#define BUFSIZE 1024

int main(int argc, char **argv) {
    int sockfd;                   /* client socket */
    struct sockaddr_in serveraddr; /* server socket addr struct */
    struct hostent *server;       /* server's DNS entry */
    char *hostname;               /* server's domain name */
    int portno;                   /* server's port number */
    char buf[BUFSIZE];

    /* check command line arguments */
    if (argc != 3) {
       fprintf(stderr,"usage: %s <hostname> <port>\n", argv[0]);
       exit(0);
    }
    hostname = argv[1];
    portno = atoi(argv[2]);
```


Echo client (cont)

```
    /* create the socket */
    sockfd = Socket(AF_INET, SOCK_STREAM, 0);

    /* initialize the server's socket address struct */
    server = Gethostbyname(hostname);
    bzero((char *) &serveraddr, sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&serveraddr.sin_addr.s_addr, server->h_length);
    serveraddr.sin_port = htons(portno);

    /* request a connection to the server */
    Connect(sockfd, (struct sockaddr *)&serveraddr,
            sizeof(serveraddr));
```

Echo client (cont)

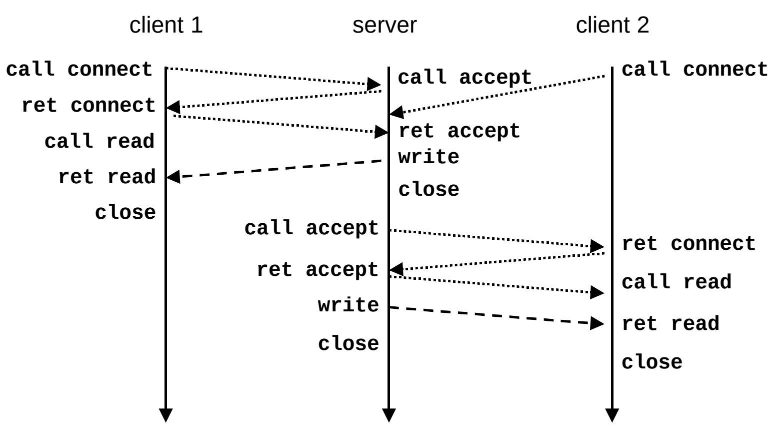
```
    /* get a message line from the user */
    printf("Please enter msg: ");
    bzero(buf, BUFSIZE);
    fgets(buf, BUFSIZE, stdin);

    /* send message line to server and read its echo */
    Write(sockfd, buf, strlen(buf));
    bzero(buf, BUFSIZE);
    Read(sockfd, buf, BUFSIZE);
    printf("Echo from server: %s", buf);
    Close(sockfd);
    exit(0);
}
```


open_streamsock helper function

```
int open_streamsock(int portno) {
  int listenfd, optval = 1;
  struct sockaddr_in serveraddr;

  /* create a socket descriptor */
  listenfd = Socket(AF_INET, SOCK_STREAM, 0);
  Setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR,
             (const void *)&optval , sizeof(int));

  /* accept requests to (any IP addr, portno) */
  bzero((char *) &serveraddr, sizeof(serveraddr));
  serveraddr.sin_family = AF_INET;
  serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serveraddr.sin_port = htons((unsigned short)portno);
  Bind(listenfd, (struct sockaddr *) &serveraddr sizeof(serveraddr));

  /* Make it a listening socket ready to accept conn requests */
  Listen(listenfd, 5);
  return listenfd;
}
```


Iterative servers

Iterative servers process one request at a time.



Iterative echo server

```
/*
 * echoserveri.c - iterative echo server
 * Usage: echoserveri <port>
 */
#include <ics.h>
#define BUFSIZE 1024
void echo(int connfd);

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

  /* check command line args */
  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n",
            argv[0]);
    exit(0);
  }
  portno = atoi(argv[1]);
```

Iterative echo server (cont)

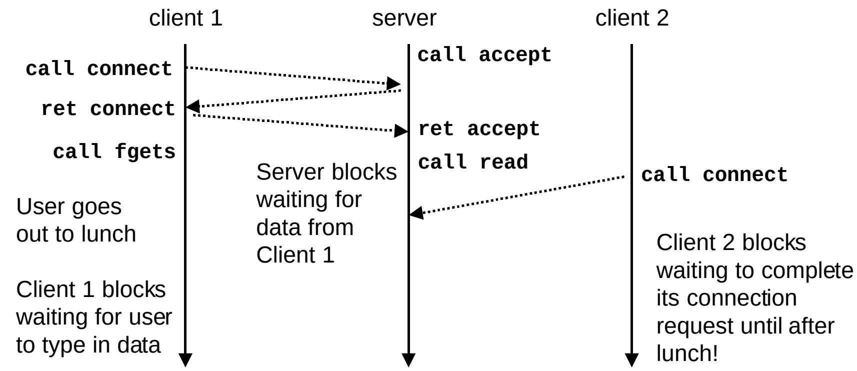
```
/* open the listening socket */
  listenfd = open_streamsock(portno);

  /* main server loop */
  while (1) {
    connfd = Accept(listenfd,
                   (struct sockaddr *) &clientaddr, &clientlen);
    echo(connfd);
    Close(connfd);
  }
}

/* echo - read and echo a line from a client connection */
void echo(int connfd) {
  int n;
  char buf[BUFSIZE];

  bzero(buf, BUFSIZE);
  n = Read(connfd, buf, BUFSIZE);
  printf("server received %d bytes: %s", n, buf);
  Write(connfd, buf, strlen(buf));
}
```


Pros and cons of iterative servers

+ simple

- can process only one request at a time

- **one slow client can hold up thousands of others**
- **Example: echo clients and server**



Concurrent servers

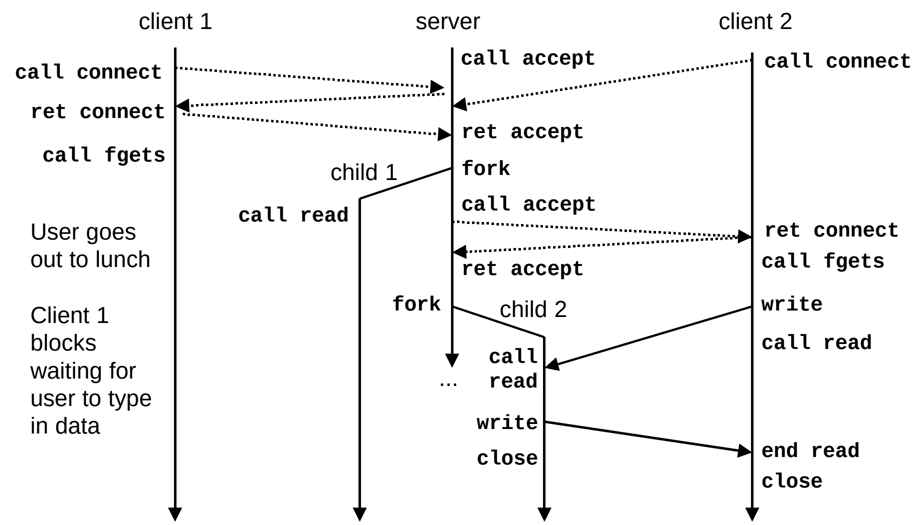
Concurrent servers process multiple requests concurrently.

- **The basic idea is to use multiple control flows to handle multiple requests.**

Example concurrent server designs:

- **Fork a new child process for each request.**
- **Create a new thread for each request.**
- **Pre-fork a pool of child processes to handle requests. *(not discussed)***
- **Pre-create a pool of threads to handle requests. *(not discussed)***
- **Manually interleave the processing for multiple open connections.**
  - **Uses Linux `select()` function to notice pending socket activity**
  - **Form of application-level concurrency**

Example: Concurrent echo server



Process-based concurrent server

```
/*
 * echoserverp.c - A concurrent echo server based on processes
 * Usage: echoserverp <port>
 */
#include <ics.h>
#define BUFSIZE 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_streamsock(portno);
```


Process-based server (cont)

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

  /* main server loop */
  while (1) {
    /* for complete portability, must restart if interrupted by */
    /* call to SIGCHLD handler */
    if ((connfd = accept(listenfd, (struct sockaddr *) &clientaddr,
                        &clientlen)) < 0) {
       if (errno == EINTR)
          continue; /* go back */
       else
          unix_error("accept");

    if (Fork() == 0) {
      Close(listenfd); /* child closes its listening socket */
      echo(connfd);    /* child reads and echos input line */
      Close(connfd);   /* child is done with this client */
      exit(0);         /* child exits */
    }
    Close(connfd); /* parent must close connected socket! */
  }
}
```


Reaping zombie children

```
/* handler - reaps children as they terminate */
void handler(int sig) {
  pid_t pid;
  int stat;

  while ((pid = waitpid(-1, &stat, WNOHANG)) > 0)
    ;
  return;
}
```

Question: Why is the call to `waitpid` in a loop?

Issues with process-based design

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

- not necessary for systems such as Linux that support Posix signal handling.
- required for portability on some older Unix systems.

Server must reap zombie children

- to avoid fatal memory leak.

Server must `close` its copy of `connfd`.

- kernel keeps reference count of descriptors that point to each socket.
- after fork, `refcnt(connfd)=2`.
- Connection will not be closed until `refcnt(connfd)=0`.

Pros and cons of process-based design

- + handles multiple connections concurrently
- + clean sharing model
  - descriptors (yes)
  - global variables (no)
- + simple and straightforward
- - nontrivial to shared data between processes
  - requires IPC (interprocess communication mechanisms)
    - FIFO's
    - System V shared memory
    - System V semaphores
- - additional overhead for process control

Threads-based server

```
/*
 * echoservert2.c - A concurrent echo server using threads
 * Usage: echoservert2 <port>
 */
#include <ics.h>
#define BUFSIZE 1024
void echo(int connfd);
void *thread(void *vargp);

int main(int argc, char **argv) {
  int listenfd, *connfdp;
  int portno;
  struct sockaddr_in clientaddr;
  int clientlen = sizeof(struct sockaddr_in);
  pthread_t tid;

  /* check command line args */
  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(0);
  }
  portno = atoi(argv[1]);
```


Threads-based server (cont)

```
 /* open the listening socket */
  listenfd = open_streamsock(portno);

  /* main server loop */
  while (1) {
    connfdp = Malloc(sizeof(int));
    *connfdp = Accept(listenfd,
                      (struct sockaddr *) &clientaddr, &clientlen);

    Pthread_create(&tid, NULL, thread, (void *)connfdp);
  }
}
```


Threads-based server (cont)

```
/* thread - thread routine */
void *thread(void *vargp) {
  int connfd;

  /* run detached to avoid a memory leak */
  Pthread_detach(pthread_self());

  connfd = *((int *)vargp);
  Free(vargp);

  echo(connfd);
  Close(connfd);
  return NULL;
}
```

Issues with threads-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);`

Pros and cons of thread-based design

+ Arguably the simplest option

- No reaping zombies
- No signal handling

+ 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 race conditions between threads.

- malloc an argument struct for each thread and pass ptr to struct to thread routine.
- Keep globals to a minimum.
- If a thread references a global variable:
  - protect it with a semaphore or a mutex or
  - think carefully about whether unprotected is safe:
    - » e.g., one writer thread, multiple readers is OK.

`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 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?

`select` example

```
/*
 * main loop: wait for connection request or stdin command.
 * If connection request, then echo input line
 * and close connection. If 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);          /* initialize the fd set */
   FD_SET(listenfd, &readfds); /* add socket fd */
   FD_SET(0, &readfds);        /* add stdin fd (0) */
   Select(listenfd+1, &readfds, NULL, NULL, NULL);
```


`select` example

First we check for a pending event on stdin.

```
  /* 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 = 0;
        break;
     default: /* bad input */
        printf("ERROR: unknown command\n");
        printf("server> ");
        fflush(stdout);
     }
  }
```


`select` example

Next we check for a pending connection request.

```
  /* if a connection request has arrived, process it */
  if (FD_ISSET(listenfd, &readfds)) {
     connfd = Accept(listenfd,
                     (struct sockaddr *) &clientaddr, &clientlen);
     connectcnt++;

     bzero(buf, BUFSIZE);
     Read(connfd, buf, BUFSIZE);
     Write(connfd, buf, strlen(buf));
     Close(connfd);
  }
} /* while */
```


I/O multiplexing with `select`

```
/*
 * echoservers.c - A concurrent echo server based on select
 * Usage: echoservers <port>
 */
#include <ics.h>
#define BUFSIZE 1024
void echo(int connfd);

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

  fd_set allset; /* descriptor set for select */
  fd_set rset;   /* copy of allset for select */
  int maxfd;     /* max descriptor value for select */

  int client[FD_SETSIZE]; /* pool of connected descriptors */
  int maxi;      /* highwater index into client pool */
  int nready;    /* number of ready descriptors from select */
  int i, sockfd; /* misc */
```

I/O multiplexing with `select` (cont)

```
/* check command line args */
if (argc != 2) {
  fprintf(stderr, "usage: %s <port>\n", argv[0]);
  exit(0);
}
portno = atoi(argv[1]);

/* open the listening socket */
listenfd = open_streamsock(portno);

/* initialize the pool of active client connections */
maxi = -1;
maxfd = listenfd;
for (i=0; i< FD_SETSIZE; i++)
  client[i] = -1;
FD_ZERO(&allset);
FD_SET(listenfd, &allset);
```


I/O multiplexing with `select` (cont)

```
/* main server loop */
while (1) {
  rset = allset;
  nready = Select(maxfd+1, &rset, NULL, NULL, NULL);

  /* PART I: add a new connected descriptor to the pool */
  if (FD_ISSET(listenfd, &rset)) {
    connfd = Accept(listenfd, (struct_sockaddr *)
                    &clientaddr, &clientlen);
    nready--;
```


I/O multiplexing with `select` (cont)

```
    /* update the client pool */
    for (i=0; i<FD_SETSIZE; i++)
      if (client[i] < 0) {
         client[i] = connfd;
         break;
      }
      if (i == FD_SETSIZE)
         app_error("Too many clients\n");


    /* update the read descriptor set */
    FD_SET(connfd, &allset);
    if (connfd > maxfd)
       maxfd = connfd;
    if (i > maxi)
       maxi = i;

  } /* if (FD_ISSET(listenfd, &rset) */
```


I/O multiplexing with `select` (cont)

```
    /* PART II: check the pool of connected descs for client data */
    for (i=0; (i<=maxi) && (nready > 0); i++) {
      sockfd = client[i];
      if ((sockfd > 0) && (FD_ISSET(sockfd, &rset))) {
        echo(sockfd);
        Close(sockfd);
        FD_CLR(sockfd, &allset);
        client[i] = -1;
        nready--;
      }
    } /* for */
  } /* while(1) */
}
```

Pro and cons of `select`-based design

+ one logical control flow.
+ can single step with a debugger.
+ no process or thread control overhead.

- significantly more complex to code initially than process or thread designs.
- vulnerable to denial of service attack
  - How?