CONCURRENCY MODEL
int main (int argc, char *argv[]) {
...
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    bzero((char *)&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = htonl(INADDR_ANY);
    saddr.sin_port = htons(port);
    bind(listenfd, (struct sockaddr *)&saddr, sizeof(saddr));

    listen(listenfd, 5);
    while (1) {
        connfd = accept(listenfd, (struct sockaddr *)&caddr, &clen);
        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf ("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }
        close(connfd);
    }
}
Iterative Model

- One request at a time

Client 1
- call connect
- ret connect
- call read
- ret read
- close

Server
- call accept
- ret accept
- write
- close

Client 2
- call connect
- ret connect
- call read
- ret read
- close
- write
- close
Only One Client at a Time

- Solution: use concurrent servers instead

- Use multiple concurrent flows to serve multiple clients at the same time.

User goes out to lunch.

Client 1 blocks waiting for user to type in data.

Server blocks waiting for data from Client 1.

Client 2 blocks waiting to complete its connection request until after lunch!
Creating Concurrent Flows

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

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

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

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
- fork
- call accept
- ret accept
- call fgets

child 1

- call read

child 2

- fork
- call read
- write
- close

Client 2

- call connect
- ret connect
- call fgets
- write
- call read
- end read
- close
int main (int argc, char *argv[]) {
    ... 
    while (1) {
        connfd = accept (listenfd, (struct sockaddr *)&caddr, &caddrlen));

        while ((n = read(connfd, buf, MAXLINE)) > 0) {
            printf ("got %d bytes from client.\n", n);
            write(connfd, buf, n);
        }

        close(connfd);
    }
}
int main (int argc, char *argv[]) 
{
  ...
  signal (SIGCHLD, handler);

  while (1) {
    connfd = accept (listenfd, (struct sockaddr *)&caddr, &caddrlen);
    if (fork() == 0) {
      close(listenfd);
      while ((n = read(connfd, buf, MAXLINE)) > 0) {
        printf ("got %d bytes from client." , n);
        write(connfd, buf, n);
      }
      close(connfd);
      exit(0);
    }
    close(connfd);
  }
}

void handler(int sig) {
  pid_t pid;
  int stat;
  while ((pid = waitpid(-1, &stat, WNOHANG)) > 0);
  return;
}
Implementation Issues

- Servers should restart `accept()` if it is interrupted by a transfer of control to SIGCHLD handler
  - Not necessary for systems with 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 for each socket
  - After `fork()`, ref. count of `connfd` becomes 2
  - Connection will not be closed until ref. count becomes 0
Process-based Designs

- **Pros**
  - Handles multiple connections concurrently
  - Easy to implement and clear to understand sharing model
  - Simple and straightforward

- **Cons**
  - Additional overhead for process control
    - Process creation and termination
    - Process switching
  - Nontrivial to share data between processes
    - Requires IPC (Inter-Process Communication) mechanisms
      - FIFO, shared memory, and semaphores
I/O Multiplexing

- Event-based concurrent servers
  - Maintain a pool of connected descriptors
  - Repeat followings forever
    - Use `select()` system call to block until
      - (a) New connection request arrives on the listening descriptor
      - (b) New data arrives on an existing connected descriptor
    - If (a), add new connection to connection descriptor pool
    - If (b), read any available data from connection
      - Close connection on EOF and remove it from pool.
- I/O multiplexing provides more control with less overhead
Select on Sockets

- **Prototype**
  ```c
  #include <sys/select.h>
  int select(int nfds, fd_set *readfds, fd_set *writelfds, 
  fd_set *exceptfds, struct timeval *timeout);
  ```

- `select()` returns number of FDs contained in the three returned descriptor sets

- `select()` blocks for timeout waiting for a file descriptor to become ready

- FDs in `readfds` will be tested whether they have something to read

- FDs in `writelfds` will be tested whether they are available for writing
Select on Sockets

- FDs in `exceptfds` will be watched for exceptions
- Three given FD sets will be changed accordingly
  - Only FDs in `ready` will be included in the result sets
Macros for FD Sets

- 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, fd_set *fdset)`
    - Is bit `fd` in `fdset` turned on?
typedef struct {
    int maxfd;       // largest descriptor in read_set
    int nready;     // number of ready desc. from select
    fd_set read_set;  // set of all active descriptors
    fd_set ready_set;  // subset of desc. ready for reading
} pool;

int main (int argc, char *argv[])
{
    int listenfd, connfd, val;
    pool p;

    ...
    listenfd = ...       // socket(), bind(), listen()

    // initialize pool
    p.maxfd = listenfd;
    FD_ZERO(&p.read_set);
    FD_SET(listenfd, &p.read_set);
while (1) {
    p.ready_set = p.read_set;
    p.nready = select(p.maxfd+1, &p.ready_set, NULL, NULL, NULL);

    if (FD_ISSET(listenfd, &p.ready_set)) {
        connfd = accept (listenfd, (struct sockaddr *)&caddr, &caddrlen));
        FD_SET(connfd, &p.read_set);
        if (connfd > p.maxfd) p.maxfd = connfd;
        p.nready--;
    }
    check_clients (listenfd, &p);
}
}
void check_clients (int listenfd, pool *p) {
    int s, n;
    char buf[MAXLINE];
    for (s = 0; s < p->maxfd+1 && p->nready > 0; s++) {
        if (s == listenfd) continue;
        if (FD_ISSET(s, &p->read_set) && FD_ISSET(s, &p->ready_set)) {
            p->nready--;
            if ((n = read(s, buf, MAXLINE)) > 0)
                write(s, buf, n);
            if (n == 0) { // EOF
                close(s);
                FD_CLR(s, &p->read_set);
                if (s == p->maxfd) {
                    p->maxfd--;
                    while (!FD_ISSET(p->maxfd, &p->read_set)) p->maxfd--;
                }
            }
        }
    }
}
Event-based Designs

- **Pros**
  - 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

- **Cons**
  - Significantly more complex to code than process- or thread-based designs
  - Can be vulnerable to Denial-of-Service attack