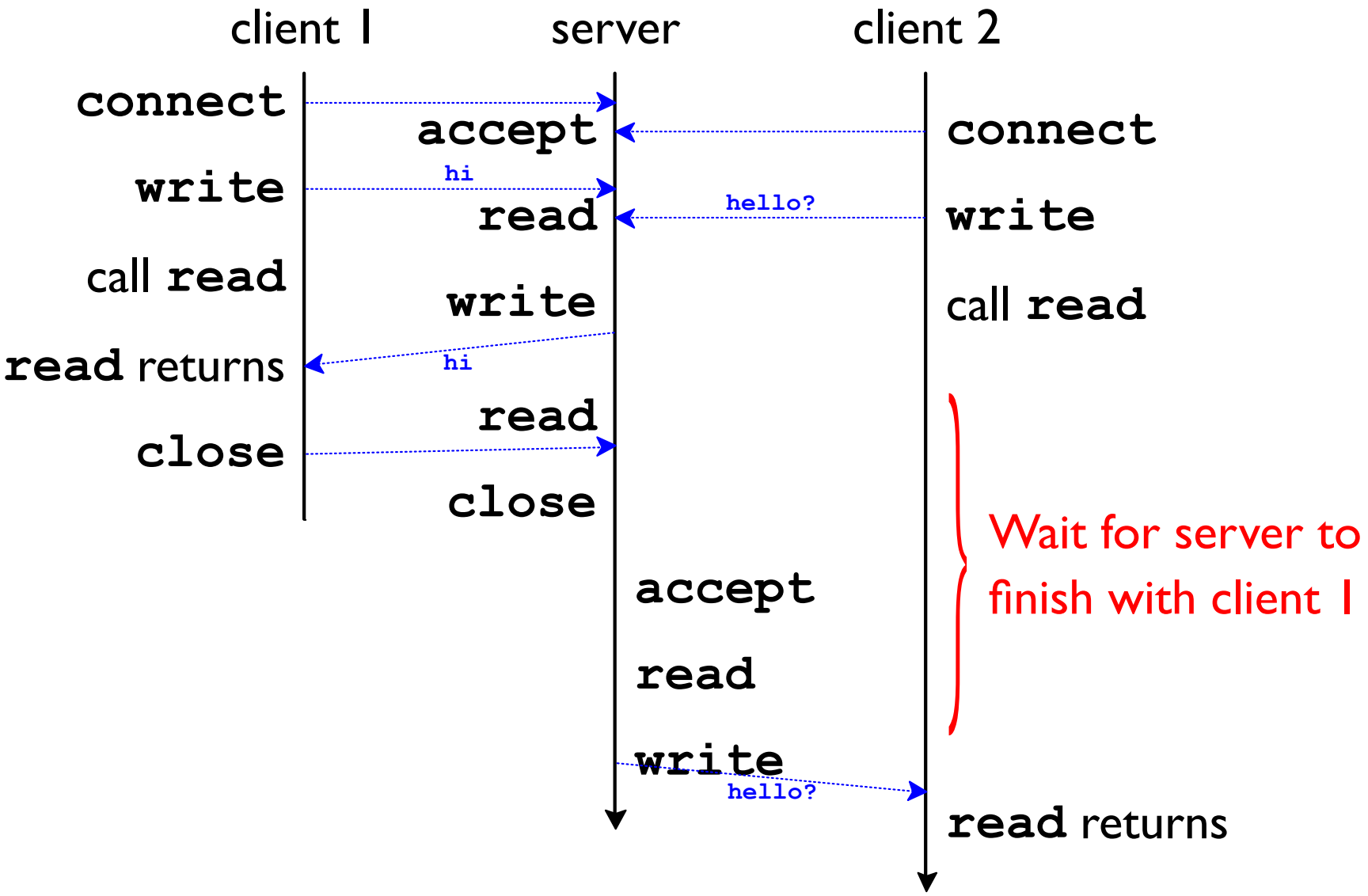
Echo Server and Multiple Clients

The current echo server operates ***sequentially***:

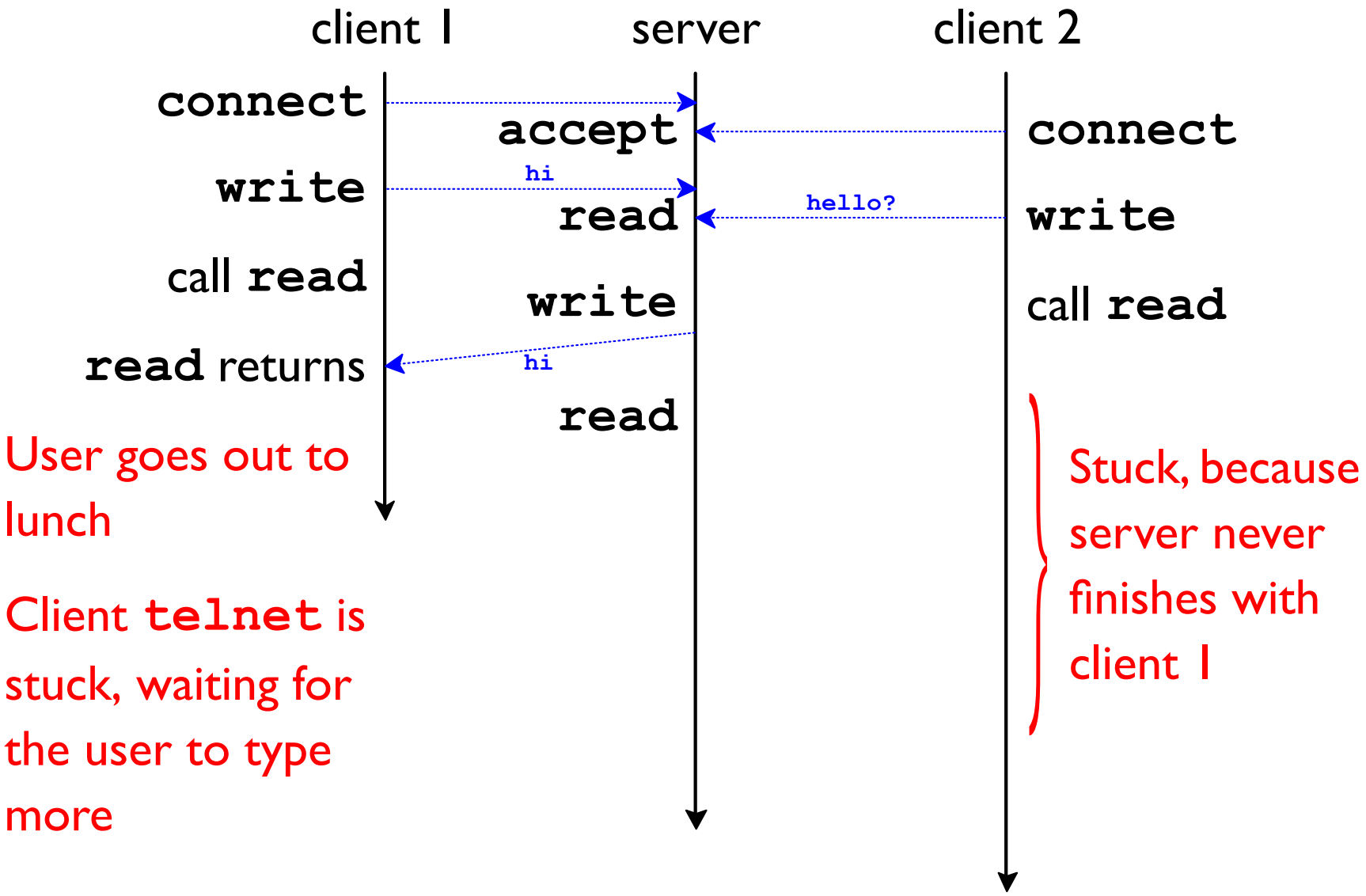
- Only one client is served at a time
- After a client sends EOF, next client can be served

TCP listener queue allows multiple clients to connect,
but only one of them receives echoes at a time

Echo Server and Multiple Clients



Echo Server and Multiple Clients

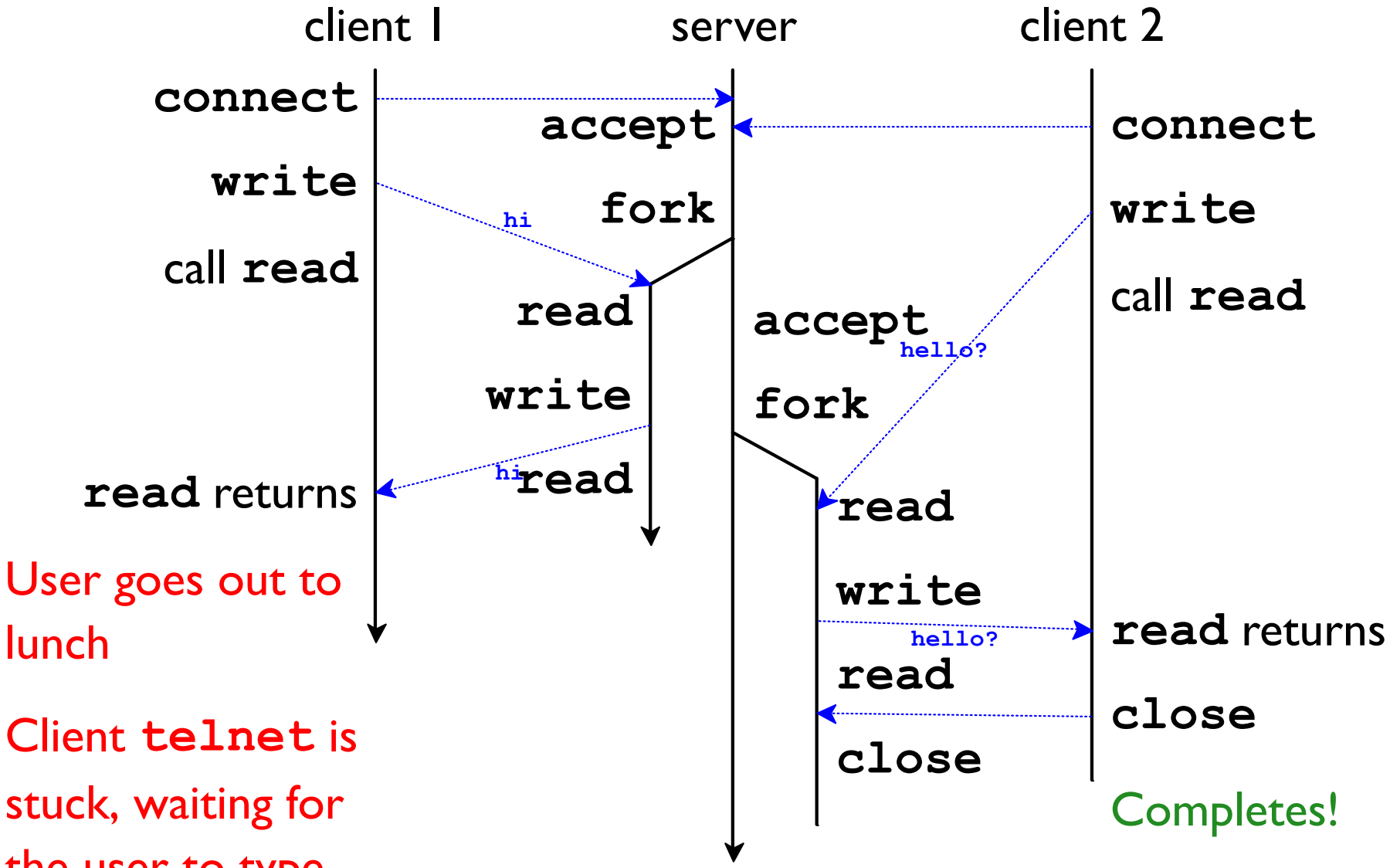


Echo Concurrency

Options for serving clients ***concurrently***:

- Per-connection processes **`fork`**
- Per-connection threads **`pthread_create`**
- Event-driven with multiplexed I/O **`select`**

Echo Concurrency by Processes



Echo Concurrency by Processes

p_echo.c

```
int main(int argc, char **argv) {
  ....
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    if (Fork() == 0) {
       Close(listenfd);
       Getnameinfo((SA *) &clientaddr, clientlen,
                   client_hostname, MAXLINE, client_port, MAXLINE, 0);
       printf("Connected to (%s, %s)\n", client_hostname, client_port);

       echo(connfd);

       Close(connfd);
       exit(0);
    }
    Close(connfd);
  }
}
.....
```

Echo Concurrency by Processes

p_echo.c

```
int main(int argc, char **argv) {
  ....
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    if (Fork() == 0) {
       Close(listenfd);
       Getnameinfo((SA *) &clientaddr, clientlen,
                   client_hostname, MAXLINE, client_port, MAXLINE, 0);
       printf("Connected to (%s, %s)\n", client_hostname, client_port);

       echo(connfd);

       Close(connfd);
       exit(0);
    }
    Close(connfd);
  }
}
.....
```

an important **Close** to avoid a leak

Echo Concurrency by Processes

p_echo.c

```
int main(int argc, char **argv) {
  ....
  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    if (Fork() == 0) {
       Close(listenfd);
       Getnameinfo((SA *) &clientaddr, clientlen,
                   client_hostname, MAXLINE, client_port, MAXLINE, 0);
       printf("Connected to (%s, %s)\n", client_hostname, client_port);

       echo(connfd);

       Close(connfd);
       exit(0);
    }
    Close(connfd);
  }
}
.....
```

still leaking PIDs — but where to **waitpid**?

Echo Concurrency by Processes

p_echo.c

```
void sigchld_handler(int sig)  {
  while (waitpid(-1, 0, WNOHANG) > 0)
    ;
}

int main(int argc, char **argv) {
  ....
  Signal(SIGCHLD, sigchld_handler);
  ....
}

....
```

Echo Concurrency by Processes

✓ Processes are great when each connection is independent

✗ Processes are not so great if connections interact

Try making `p_echo.c` track total bytes sent

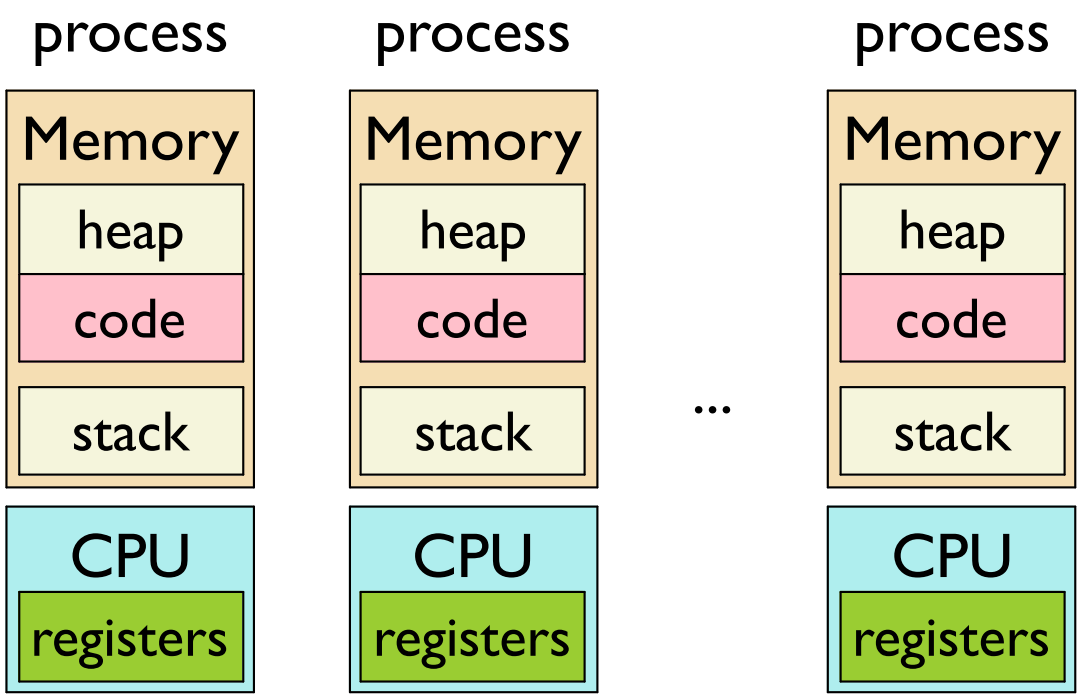
Threads

A ***thread*** is like a process in that it has its own stack, registers, and control flow

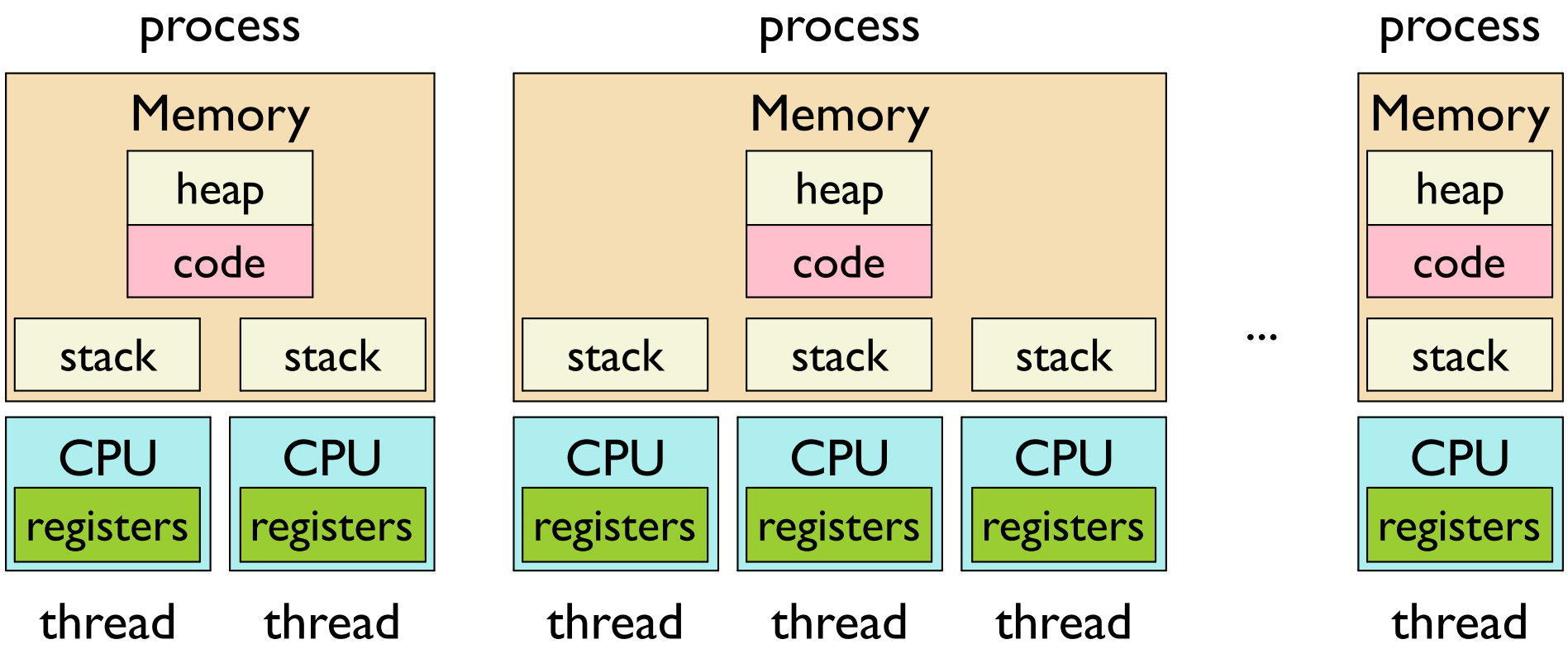
Threads within a process share a virtual address space, file descriptors, etc.

⇒ easier communication among threads

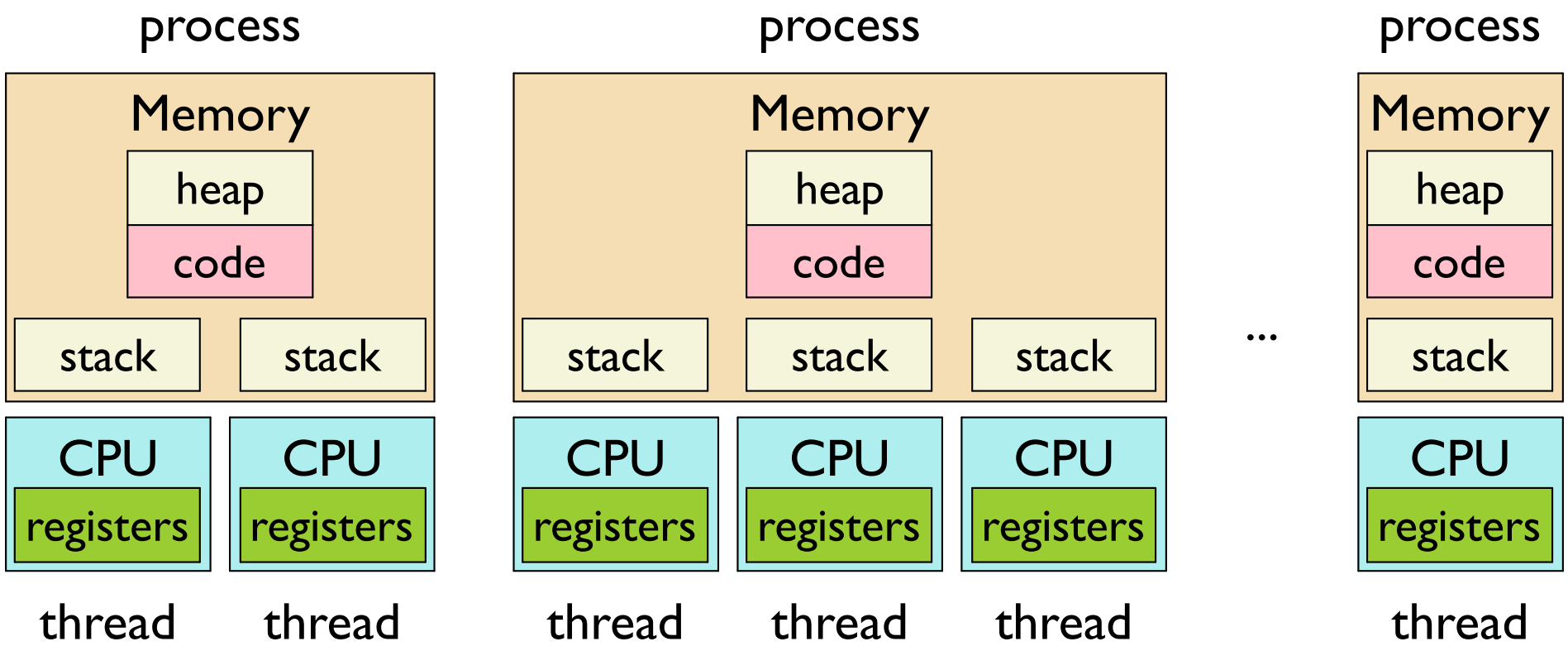
Processes: The Illusion



Threads: The Illusion



Threads: The Illusion



⇒ **`pthread_create`** cannot return twice like **`fork`**

Creating Threads

```
#include <pthread.h>

int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void *(*start_function)(void *),
                   void *start_arg);
```

Creates a new thread that calls

```
start_function(start_arg);
```

Handle to the new thread written to **`*thread`**

analogous to a PID

Options specified in **`attr`**, which can be **`NULL`**

Creating Threads

```
#include "csapp.h"

int count = 0;

void *show_var(void *name) {
  printf("%s %p\n", name, &name);
  count++;
  return NULL;
}

int main() {
  pthread_t th;
  show_var("orig");
  Pthread_create(&th, NULL, show_var, "new");
  Sleep(1);
  printf("%d\n", count);
  return 0;
}
```

Copy

gcc -pthread ....

Processes versus Threads

`pid_t`	**`pthread_t`**
`fork`	**`pthread_create`**
`waitpid`	**`pthread_join`**
`getpid`	**`pthread_self`**
`exit` or **`return`**	**`thread_exit`** or **`return`**
`kill`	**`pthread_cancel`**

Processes versus Threads

`pid_t`	`pthread_t`
`fork`	`pthread_create`
`waitpid`	`pthread_join`
`getp`	`pthrea`
`exit` or `return`	`thread_exit` or `return`
`kill`	`pthread_cancel`

from `main`

from `start_function`

Processes versus Threads

`pid_t`	**`pthread_t`**
`fork`	**`pthread_create`**
`waitpid`	**`pthread_join`**
`getpid`	**`pthread_self`**
`exit` or **`return`**	**`thread_exit`** or **`return`**
`kill`	**`pthread_cancel`**

sortof — there's also **`pthread_kill`**

Waiting for a Thread to Complete

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **result_p);
```

Waits for **`thread`** to finish

Puts the thread's return value in **`*result_p`**

Reaps the thread's identity (so **`thread`** must not be used anymore)

Waiting for a Thread to Complete

```
#include "csapp.h"

int count = 0;

void *show_var(void *name) {
  printf("%s %p\n", name, &name);
  count++;
  return name;
}

int main() {
  pthread_t th;
  void *result;
  show_var("orig");
  Pthread_create(&th, NULL, show_var, "new");
  Pthread_join(th, &result);
  printf("%s\n", result);
  return 0;
}
```

Copy

Threads are Not Hierarchical

Unlike processes, a thread doesn't have a parent or children

Any thread can **`pthread_join`** any other thread

Threads are ***peers***

The ***main thread*** is only a little special:

- It can return from **`main`** to exit the process
- Signals sent to the process go to the main thread

Threads are Not Hierarchical

```
#include "csapp.h"

void *go1(void *ignored) {
  Sleep(1);
  printf("1\n");
  return NULL;
}

void *go2(void *th) {
  Pthread_join(*(pthread_t *)th, NULL);
  printf("2\n");
  return NULL;
}

int main() {
  pthread_t one, two;
  Pthread_create(&one, NULL, go1, NULL);
  Pthread_create(&two, NULL, go2, &one);
  Pthread_join(two, NULL);
  return 0;
}
```

Copy

Threads are Not Hierarchical

```
#include "csapp.h"

void *go1(void *ignored) {
  Sleep(1);
  printf("1\n");
  return NULL;
}

void *go2(void *th) {
  Pthread_join(*(pthread_t *)th, NULL);
  printf("2\n");
  return NULL;
}

int main() {
  pthread_t one, two;
  Pthread_create(&one, NULL, go1, NULL);
  Pthread_create(&two, NULL, go2, &one);
  Pthread_join(two, NULL);
  return 0;
}
```

Copy

Prints 1 then 2

Comment out **`Pthread_join`** in **`main`**

⇒ exits without printing

Reaping Thread Identity without Waiting

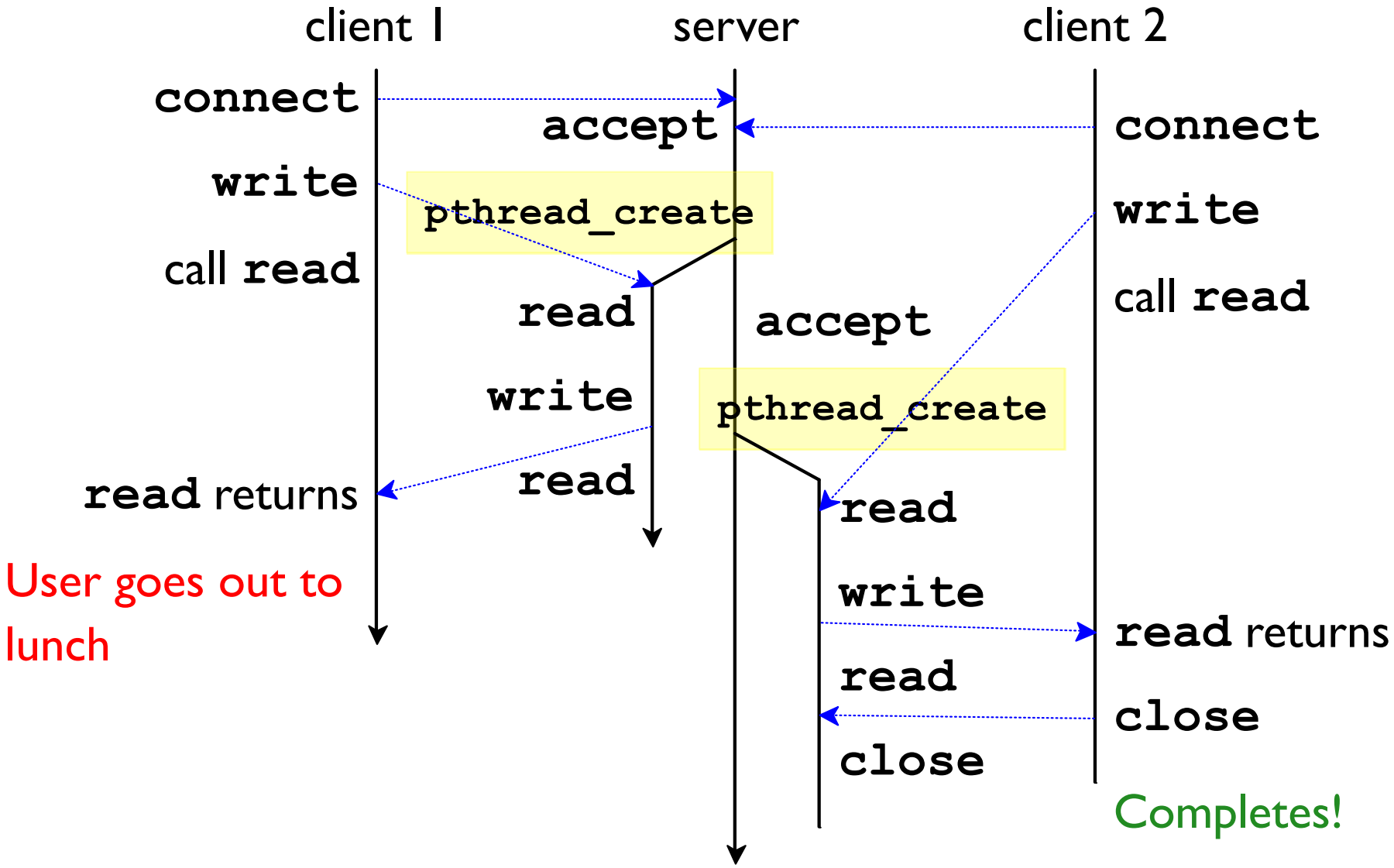
```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Reaps the thread's identity (so **`thread`** must not be used anymore), even if **`thread`** is still running

replaces using **`waitpid`** in a signal handler

Echo Concurrency by Threads



Echo Concurrency by Threads

t_echo.c

```
int main(int argc, char **argv) {
  ....

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
          client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    connfd_p = malloc(sizeof(int));
    *connfd_p = connfd;
    Pthread_create(&th, NULL, echo, connfd_p);
    Pthread_detach(th);
  }

  return 0;
}
....
```

Echo Concurrency by Threads

t_echo.c

```
int main(int argc, char **argv) {
  ....

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
          client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    connfd_p = malloc(sizeof(int));
    *connfd_p = connfd;
    Pthread_create(&th, NULL, echo, connfd_p);
    Pthread_detach(th);
  }

  return 0;
}
....
```

Don't need to wait

Don't need thread result

Echo Concurrency by Threads

t_echo.c

```
int main(int argc, char **argv) {
  ....

  listenfd = Open_listenfd(argv[1]);
  while (1) {
    socklen_t clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    Getnameinfo((SA *) &clientaddr, clientlen,
          client_hostname, MAXLINE, client_port, MAXLINE, 0);
    printf("Connected to (%s, %s)\n", client_hostname, client_port);

    connfd_p = malloc(sizeof(int));
    *connfd_p = connfd;
    Pthread_create(&th, NULL, echo, connfd_p);
    Pthread_detach(th);
  }

  return 0;
}
....
```

&connfd doesn't work

Echo Concurrency by Threads

t_echo.c

```
....
void *echo(void *connfd_p) {
  int connfd = *(int *)connfd_p;
  size_t n;
  char buf[MAXLINE];
  rio_t rio;

  free(connfd_p);

  Rio_readinitb(&rio, connfd);
  while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    printf("server received %ld bytes\n", n);
    Rio_writen(connfd, buf, n);
  }

  Close(connfd);

  return NULL;
}
```

Towards Multiplexed I/O

```
for (j = 0; j < NUM_CLIENTS; j++)
  fds[j] = Open_clientfd(hostname, port);

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  Rio_writen(fds[j], buf, strlen(buf));
}

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  amt = Rio_readn(fds[j], rbuf, strlen(buf));
  rbuf[amt] = 0;
  if (strcmp(buf, rbuf))
    app_error("didn't get expected echo");
}

for (j = 0; j < NUM_CLIENTS; j++)
  Close(fds[j]);
```

Towards Multiplexed I/O

```
for (j = 0; j < NUM_CLIENTS; j++)
  fds[j] = Open_clientfd(hostname, port);

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  Rio_writen(fds[j], buf, strlen(buf));
}

for (j = 0; j < NUM_CLIENTS; j++) {
  snprintf(buf, MAXBUF, "client%d\n", j);
  amt = Rio_readn(fds[j], rbuf, strlen(buf));
  rbuf[amt] = 0;
  if (strcmp(buf, rbuf))
    app_error("didn't get expected echo");
}

for (j = 0; j < NUM_CLIENTS; j++)
  Close(fds[j]);
```

Makes **`NUM_CLIENTS`** "concurrent" connections by explicitly spending a little time on each one

Towards Multiplexed I/O

The same idea can work for the echo server, but...

Need a way to check whether any data is available

Towards Multiplexed I/O

The same idea can work for the echo server, but...

- ~~Need a way to check whether any data is available~~
- Need a way to wait until *some* connection has data

select

```
#include <sys/select.h>

int select(int nfds,
           fd_set *readfds, fd_set *writefds, fd_set *errorfds,
           struct timeval *timeout);

void FD_ZERO(fd_set *fdset);
void FD_SET(int fd, fd_set *fdset);
int  FD_ISSET(int fd, fd_set *fdset);
```

Blocks until

- a file descriptor in **`readfds`** has input;
- a file descriptor in **`writeds`** can hold output;
- a file descriptor in **`errorfds`** has an error; or
- **`timeout`** time passes

and clears non-ready in **`readfds`**, **`writefds`**, and **`errorfds`**

Echo Concurrency by Events

e_echo.c

```
typedef struct {
  int maxfd;        /* Largest descriptor in read_set */
  fd_set read_set;  /* All active descriptors */
  fd_set ready_set; /* Subset ready for reading  */
  int nready;       /* Number of ready descriptors */
  int maxi;         /* Highwater index into client array */
  int clientfd[FD_SETSIZE];    /* Active descriptors */
  rio_t clientrio[FD_SETSIZE]; /* Active read buffers */
} pool;
```

Echo Concurrency by Events

e_echo.c

```
int main(int argc, char **argv) {
  ....
  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)) {
      clientlen = sizeof(struct sockaddr_storage);
      connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
      ....
      add_client(connfd, &pool);
    }
    check_clients(&pool);
  }
}
```

Echo Concurrency by Events

e_echo.c

```
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);
}
```

Echo Concurrency by Events

e_echo.c

```
void add_client(int connfd, pool *p) {
  int i;
  p->nready--;
  for (i = 0; i < FD_SETSIZE; i++)
    if (p->clientfd[i] < 0) {
      p->clientfd[i] = connfd;
      Rio_readinitb(&p->clientrio[i], connfd);

      FD_SET(connfd, &p->read_set);

      if (connfd > p->maxfd)
        p->maxfd = connfd;
      if (i > p->maxi)
        p->maxi = i;
      break;
    }
  if (i == FD_SETSIZE)
    app_error("add_client error: Too many clients");
}
```

Echo Concurrency by Events

e_echo.c

```
void check_clients(pool *p) {
  int i, connfd, n;
  char buf[MAXLINE];

  for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
    connfd = p->clientfd[i];
    if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
      rio_t rio = p->clientrio[i];
      p->nready--;
      if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        byte_cnt += n;
        printf("Server received %d (%d total) bytes on fd %d\n",
               n, byte_cnt, connfd);
        Rio_writen(connfd, buf, n);
      } else {
        Close(connfd);
        FD_CLR(connfd, &p->read_set);
        p->clientfd[i] = -1;
      }
    }
  }
}
```

Echo Concurrency

Per-connection processes **`fork`**

- ✓ Easy for independent
- ✗ Difficult for cooperating

Per-connection threads **`pthread_create`**

- ✓ Easy for independent or cooperating
- ✗ Maybe *too* easy for cooperating...

Event-driven with multiplexed I/O **`select`**

- ✓ Complete control of scheduling
- ✗ Manual task switching