15-213

"The course that gives CMU its Zip!"

Concurrent Servers

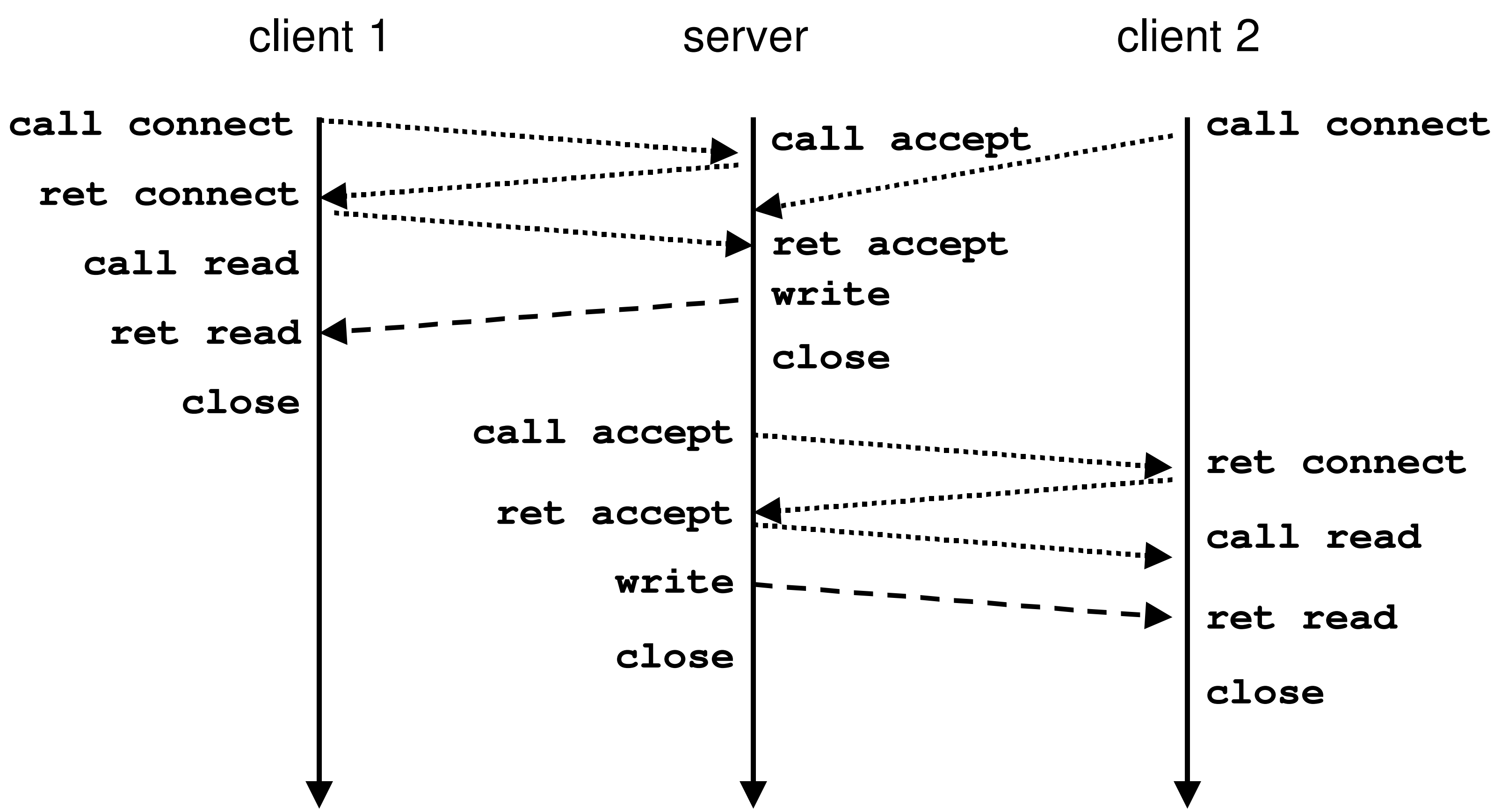
December 4, 2001

Topics

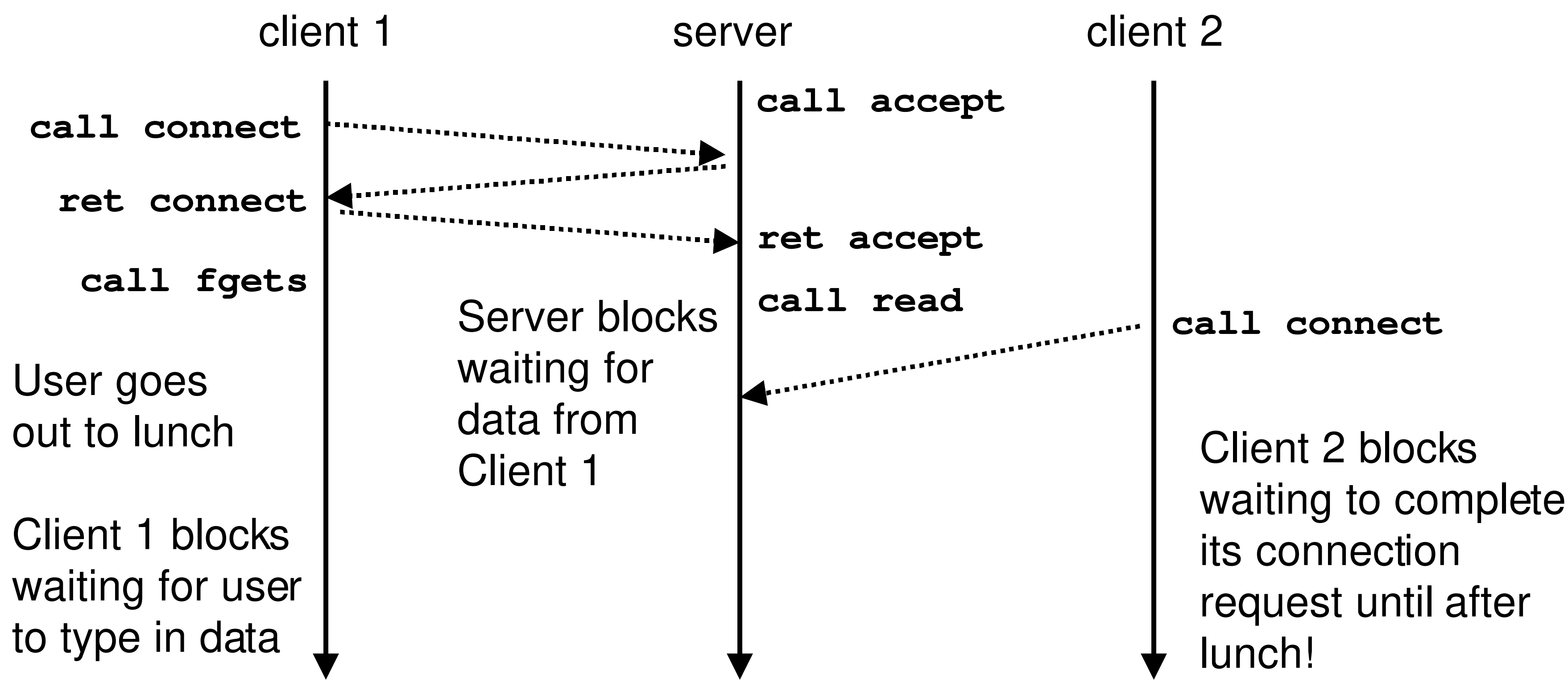
- Limitations of iterative servers
- Process-based concurrent servers
- Threads-based concurrent servers
- Event-based concurrent servers

Iterative servers

Iterative servers process one request at a time.



The fundamental flaw of iterative servers

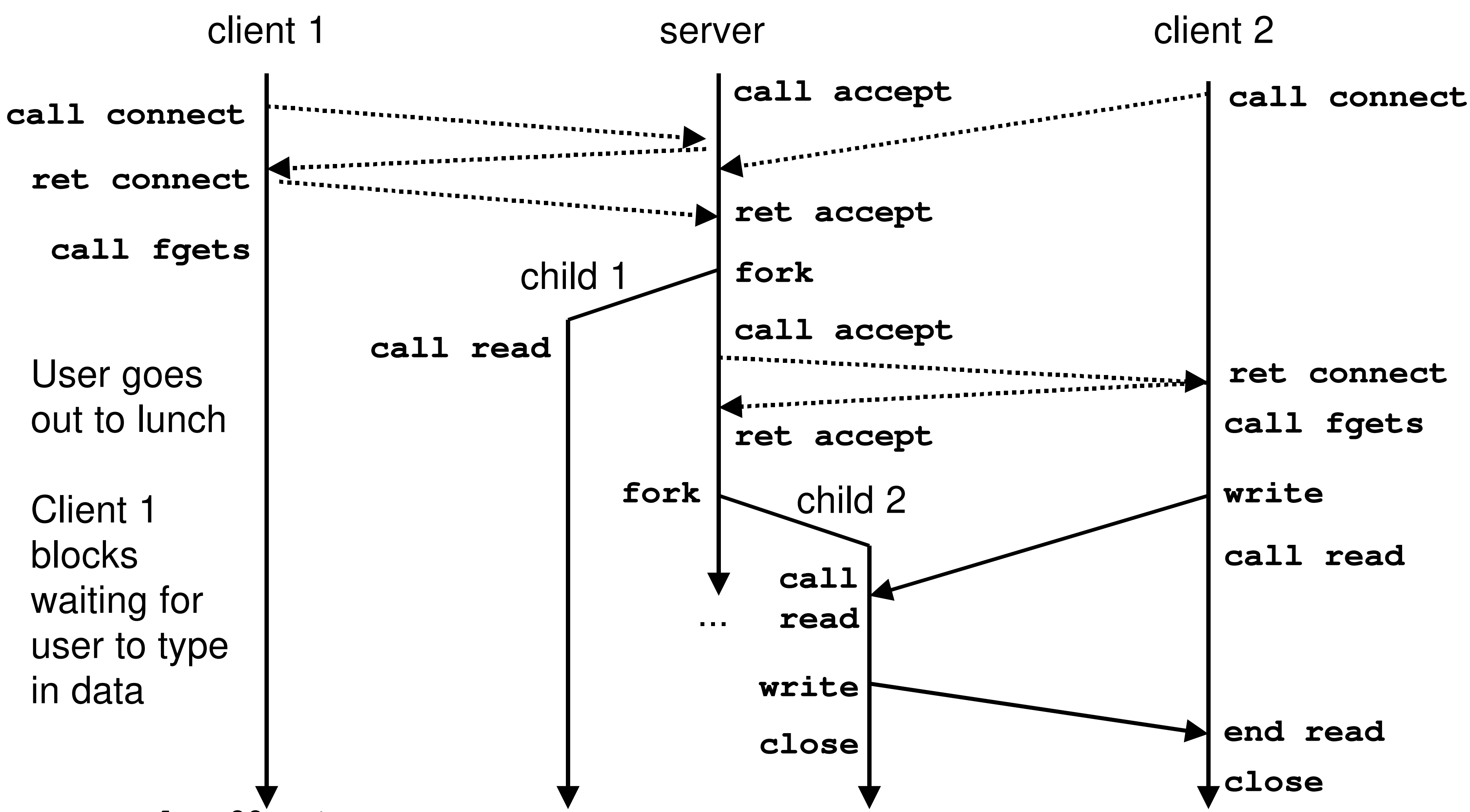


Solution: use *concurrent servers* instead.

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.

Concurrent servers

Concurrent servers handle multiple requests concurrently.



Three basic mechanisms for creating concurrent flows

1. Processes

- Kernel provides multiple control flows with separate address spaces.
- Standard Unix process control and signals.

2. Threads

- Kernel provides multiple control flows (threads) running in one process.
  - Each thread has its own stack and register values.
  - All threads share the same address space and open files.
- POSIX threads (Pthreads) interface.

3. I/O multiplexing with `select()`

- Manually interleave the processing of multiple open connections.
- Use Unix `select()` function to notice pending socket activity.
- Form of manual, application-level concurrency.
- Popular for high-performance server designs.

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_listenfd(portno);
```

Process-based concurrent server (cont)

```
 Signal(SIGCHLD, 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 */
      echo(connfd);    /* 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)

```
/* 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 reap zombie children

- to avoid fatal memory leak.

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

- Kernel keeps reference for each socket.
- After fork, `refcnt(connfd) = 2`.
- Connection will not be closed until `refcnt(connfd)=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

Threads provide more efficient flows with easier sharing of data between the flows

Traditional view of a process

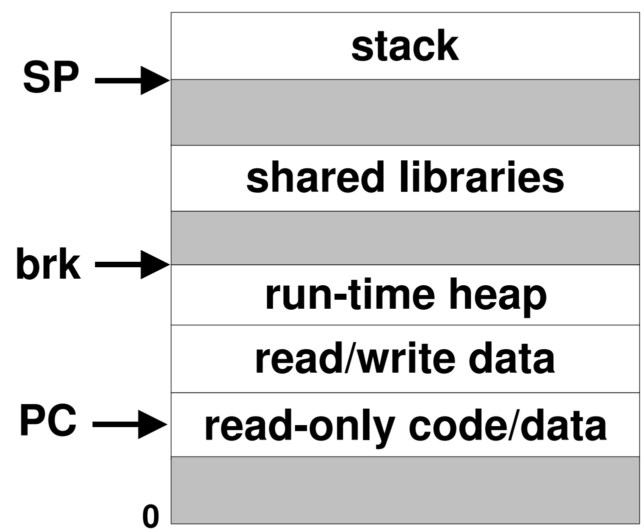
Process = process context + code, data, and stack

Process context

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

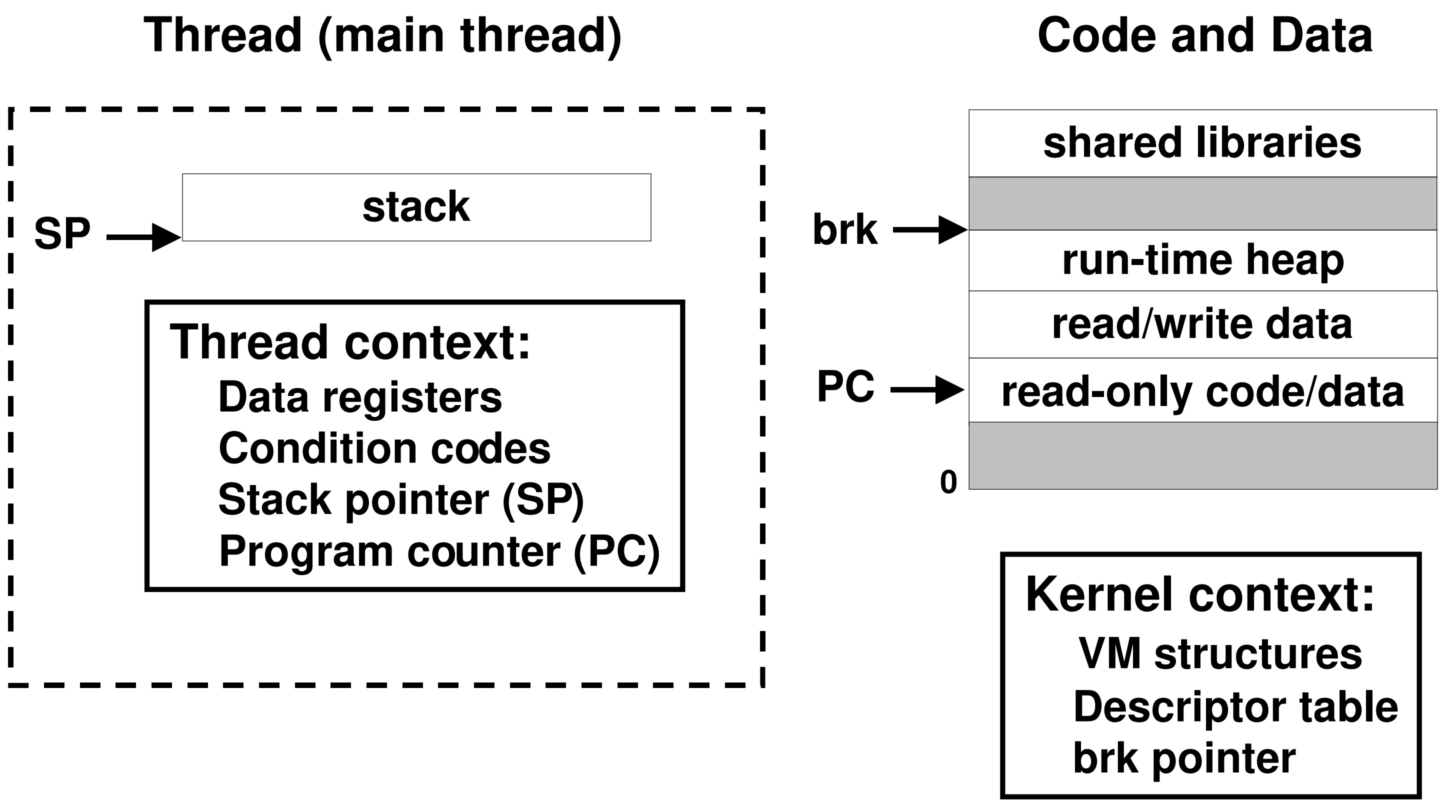
Kernel context:
- VM structures
- Descriptor table
- brk pointer

Code, data, and stack



Alternate view of a process

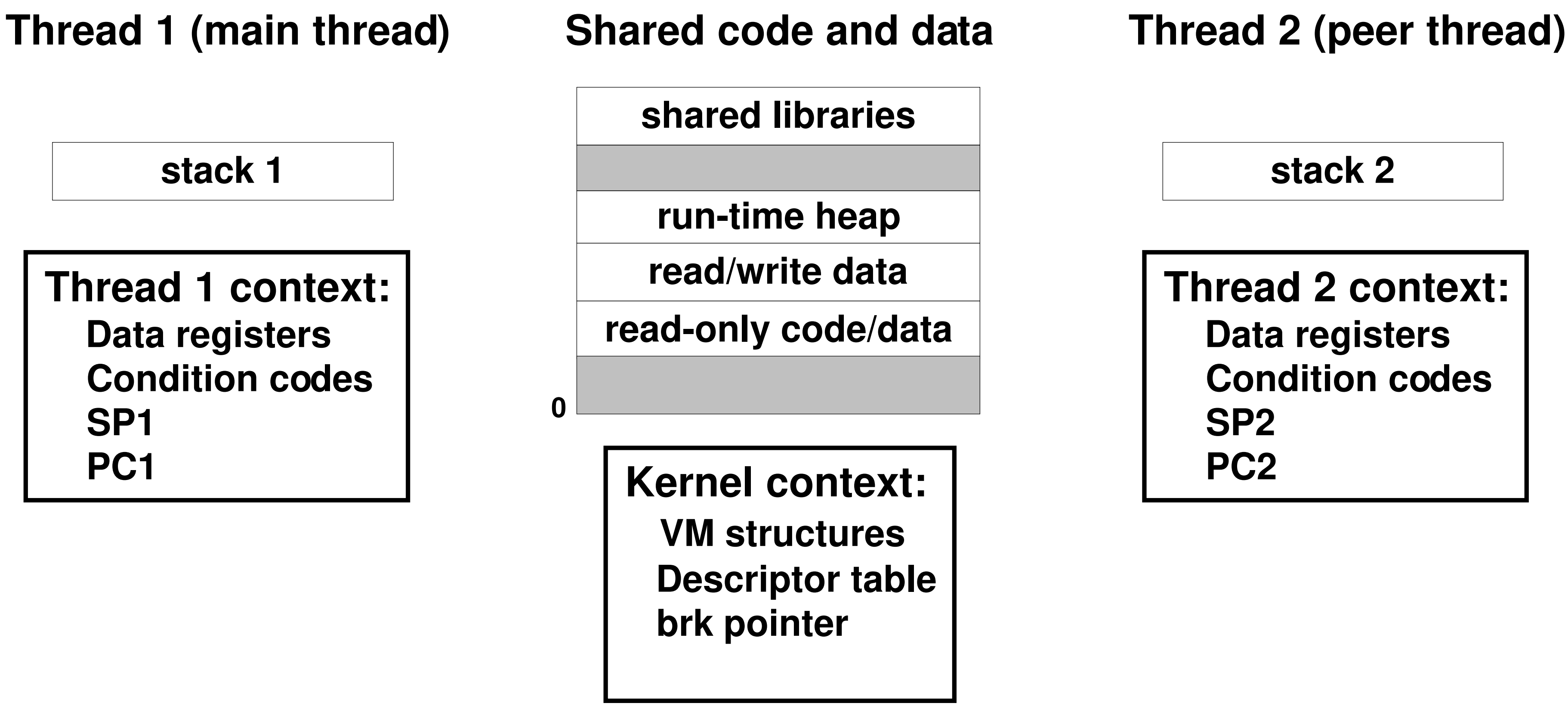
Process = thread + code, data, and kernel context



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)

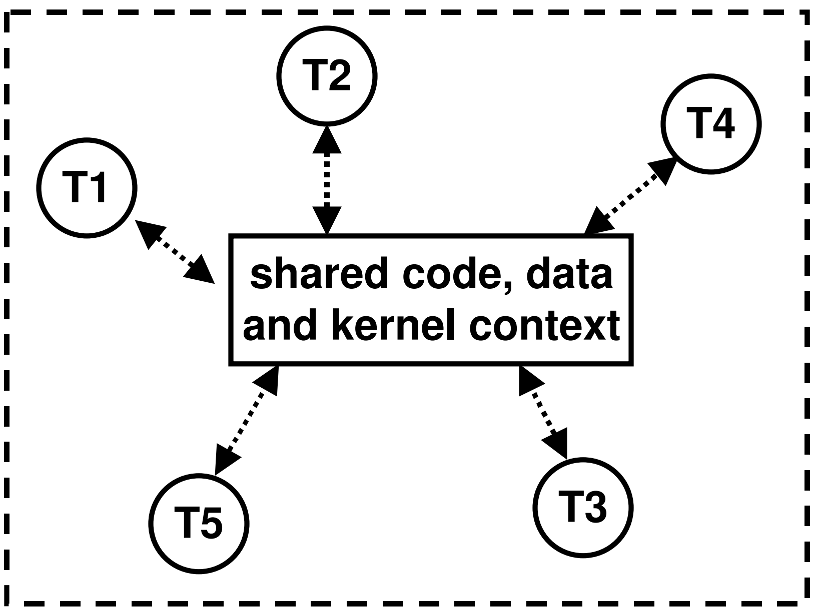


Logical view of threads

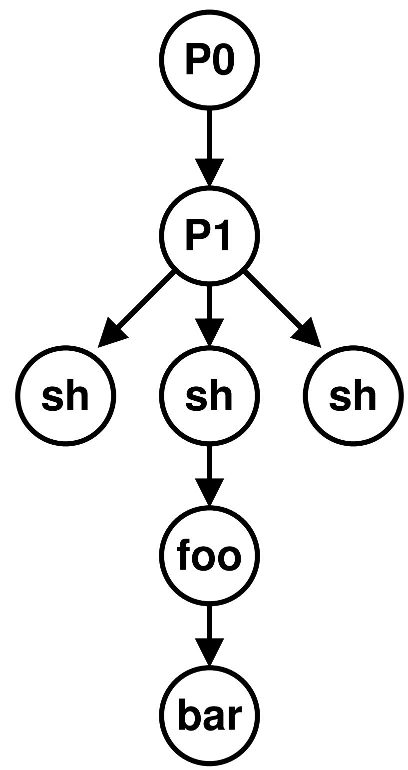
Threads associated with a process form a pool of peers.

- Unlike processes which form a tree hierarchy

Threads associated with process foo



Process hierarchy



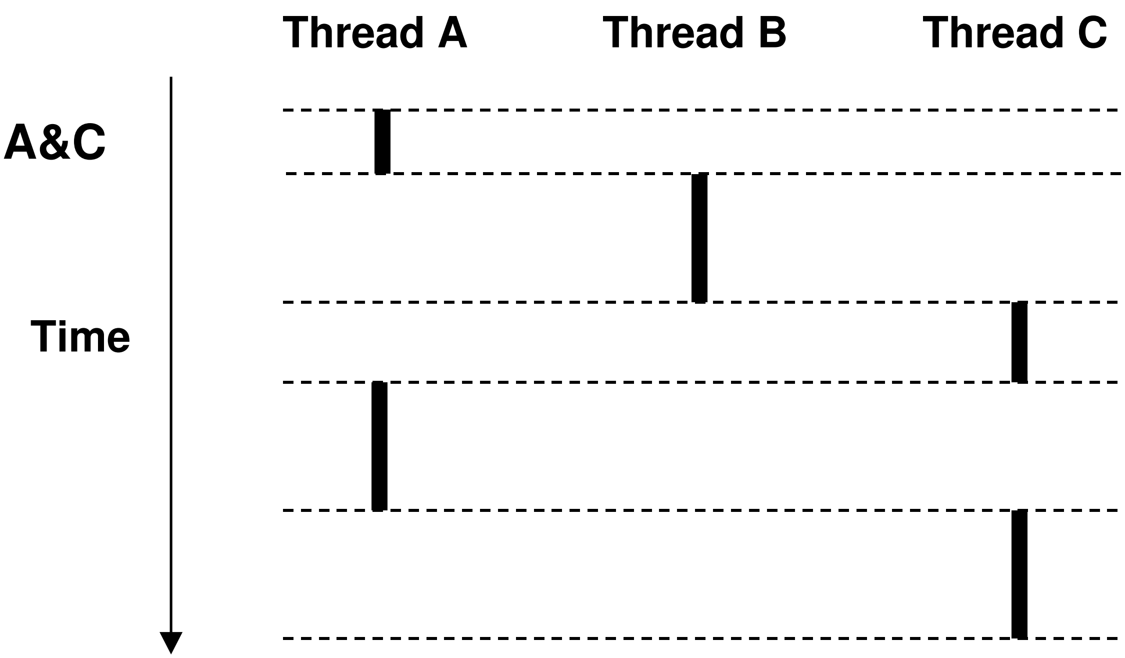
Concurrent thread execution

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**



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] , `ret` [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

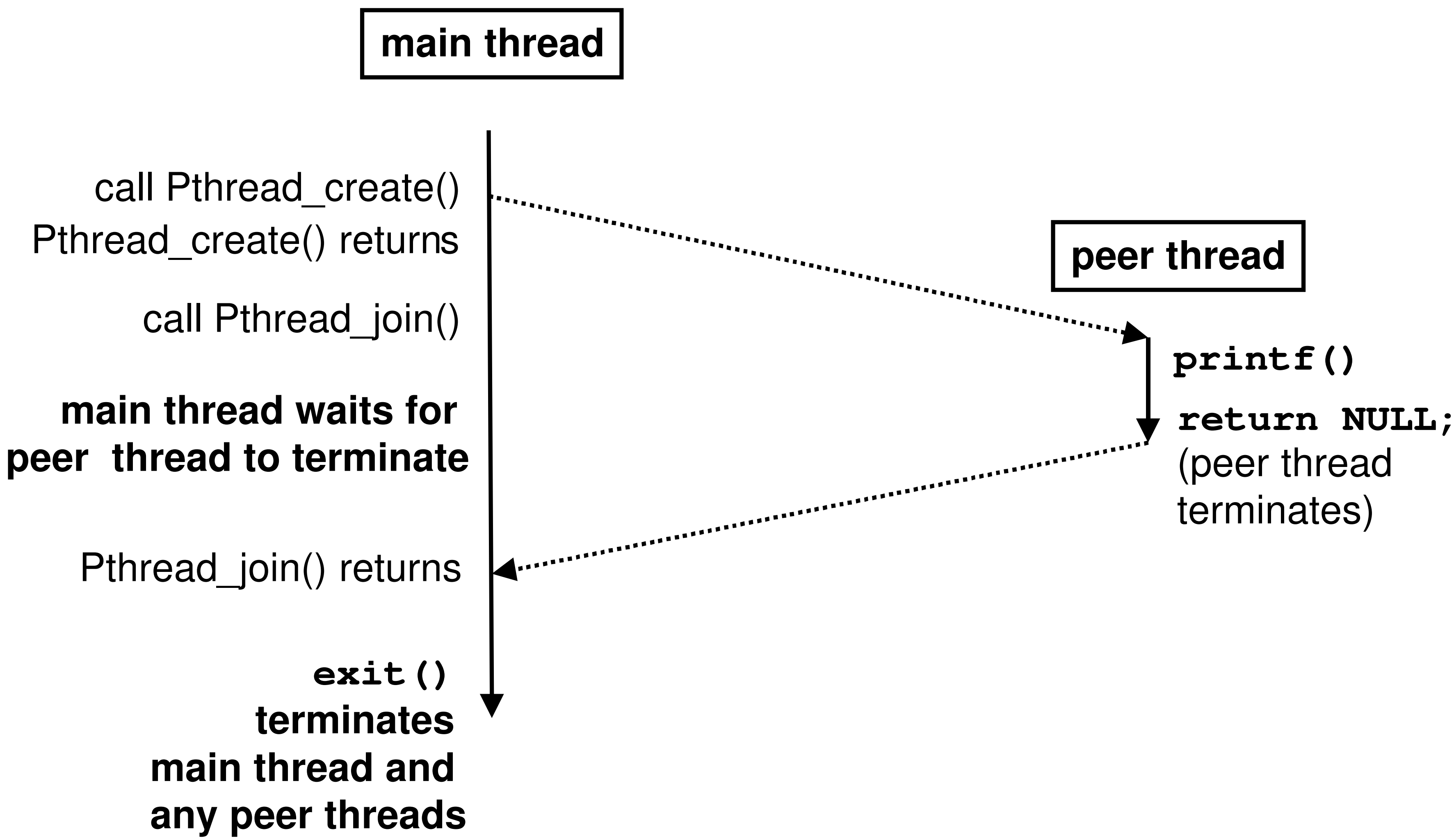
```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"

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;
}
```

Thread attributes (usually NULL)

*Thread arguments (void *p)*

*return value (void **p)*

Execution of "hello, world"



Thread-based concurrent echo server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp, 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);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

Thread-based concurrent server (cont)

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

    Pthread_detach(pthread_self());
    Free(vargp);

    echo_r(connfd); /* reentrant version of echo() */
    Close(connfd);
    return NULL;
}
```

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 *)&connfd);`

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)*

Event-based concurrent servers

An event-based approach to concurrency:

- 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.

Writing an event-based server is akin to implementing your own application-specific threads package.

`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 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);           /* 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 (cont)

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 (cont)

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);
     Readn(connfd, buf, BUFSIZE);
     Writen(connfd, buf, strlen(buf));
     Close(connfd);
  }
} /* while */
```

Event-based concurrent echo server

```
/* echoservers.c - A concurrent echo server based on select */
#include "csapp.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 */
```

Event-based concurrent server (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_listenfd(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);
```

Event-based concurrent server (cont)

```
/* main server loop */
while (1) {
  rset = allset;

  /* Wait until one or more descriptors are ready to read */
  nready = Select(maxfd+1, &rset, NULL, NULL, NULL);
  ...
```

Event-based concurrent server (cont)

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

    /* 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;
  }
```

Event-based concurrent server (cont)

```
    /* PART II: check the pool of connected descriptors for
       client data to read */
    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) */

} /* main */
```

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?