

## Last Time: Web Stuff

- Web follows the client / server model
- Web transactions are in HTTP
  - A protocol layered on top of TCP/IP
  - Requests / responses are in ASCII
  - Content may be static or dynamic
- Proxies sit between clients and servers
- Also: Course evaluations are all online now, please go fill out an evaluation for 4400

1

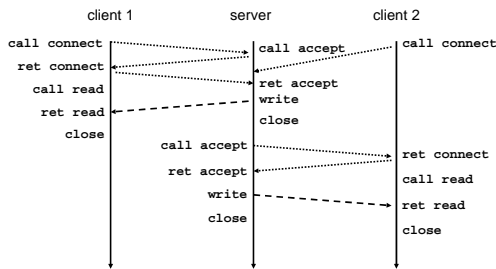
## Today: Concurrent Servers

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

2

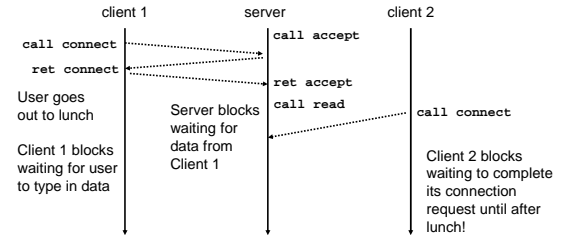
## Iterative Servers

- Iterative servers process one request at a time.



3

## Fundamental Flaw of Iterative Servers

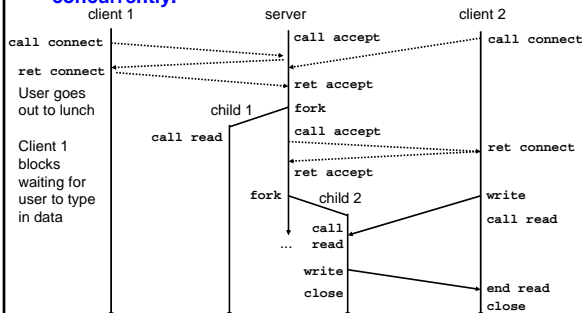


- Solution: use *concurrent servers* instead.
  - Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.

4

## Concurrent Servers

- Concurrent servers handle multiple requests concurrently.



5

## Three Basic Mechanisms for Creating Concurrency in a Server

1. Processes
  - Kernel automatically interleaves multiple logical flows
    - » Processes are kernel scheduled entities
  - Each flow has its own private address space
2. Threads
  - Kernel automatically interleaves multiple logical flows
    - » Threads are kernel scheduled entities
  - Each flow shares the same address space
3. Events – I/O multiplexing with `select()`
  - User manually interleaves multiple logical flows
    - » I/O flows are not kernel scheduled
  - Each flow shares the same address space
  - Popular for high-performance server designs

6

## Process-Based Concurrent Server

```
/*
 * echoserv.c - A concurrent echo server based on processes
 * Usage: echoserv <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);
```

7

## 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! */
}
```

Note after fork both child and server are waiting in accept for the same listening and connected sockets

8

## 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;
}
```

9

## 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 child processes
  - to avoid fatal memory leak
- Server must close its copy of `connfd`.
  - Kernel keeps reference count for each socket
  - After fork, `refcnt(connfd) = 2`
  - Connection will not be closed until `refcnt(connfd) = 0`

10

## Child `connfd` Closure

- Does the child process need to `Close(connfd)`?
- No
  - Kernel will close all fds on child termination

11

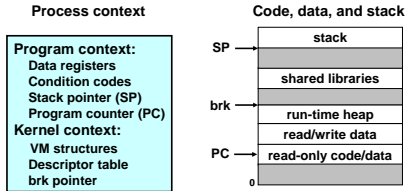
## Pros and Cons of Process-Based Designs

- + 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
  - FIFOs (named pipes), System V shared memory, and semaphores
- *Threads and I/O multiplexing provide more control with less overhead...*

12

## Traditional View of a Process

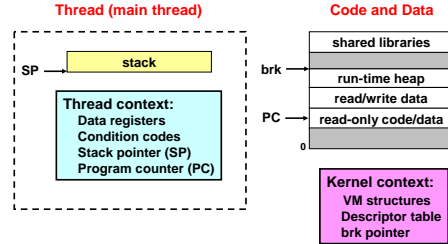
- Process = process context + code, data, and stack



13

## Alternate View of a Process

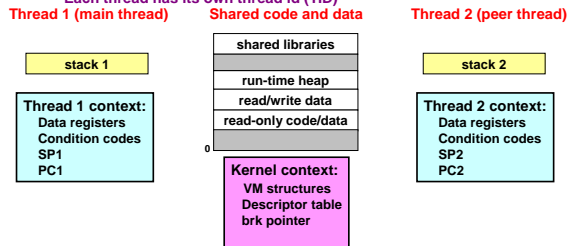
- Process = thread + code, data, and kernel context



14

## A Process With Multiple Threads

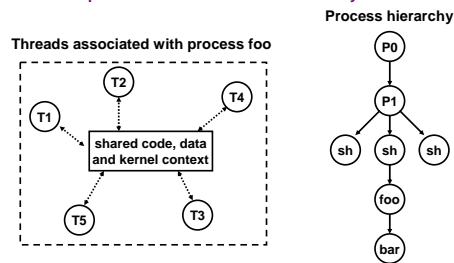
- Multiple threads can be associated with a process
  - Each thread has its own logical control flow (sequence of PC values)
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own thread id (TID)



15

## Logical View of Threads

- Threads associated with a process form a pool of peers.
  - Unlike processes which form a tree hierarchy



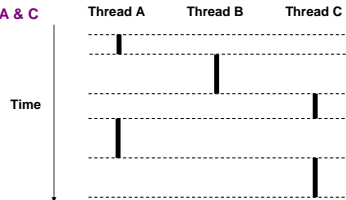
16

## 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



17

## 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

18

## POSIX Threads (Pthreads) Interface

- **Pthreads: Standard interface for ~60 functions that manipulate threads from C programs.**
  - **Creating and reaping threads.** `#include <pthread.h>`
    - » `pthread_create(pointer to handle, attributes, start routine,..)`
    - » `pthread_join \/* wait until a thread terminates *\`
  - **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`

} more on these next lecture

19

## The Pthreads "hello, world" Program

```

/*
 * 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;
}

```

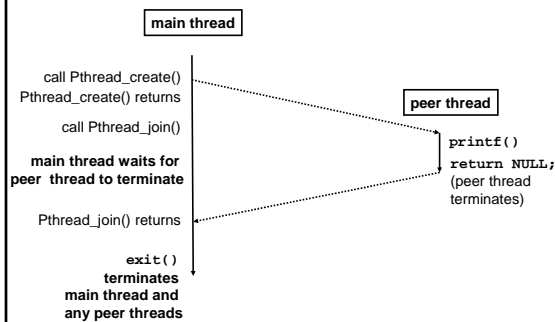
Thread attributes  
(usually NULL)

Thread arguments  
(void \*p)

return value  
(void \*\*p)

20

## Execution of Threaded "hello, world"



21

## 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);
    }
}

```

22

## Thread-Based Concurrent Server (cont)

```

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

    Pthread_detach(pthread_self()); /*next slide*/
    Free(vargp);

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

```

23

## 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)

24

## 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!
  - Note: can't gdb across multiple contexts
    - » threads & processes
    - » can't watch overall execution – EEK!
  - The ease with which data can be shared is both the greatest strength and the greatest weakness of threads.
  - (next lecture)

25

## Thread Details

- Threads may be implemented in the kernel or in user space
  - User-level threads are usually faster and more scalable
  - But, often suffer from compatibility problems
- Most modern systems give you kernel threads by default
  - Linux, Win XP, FreeBSD, etc.
  - But, all of these systems make user threads available since they're useful sometimes

26

## I/O Concurrency – The Basic Idea

- Each process can have many open fd's
  - Each fd provides access to a flow
  - Flows are inherently concurrent
    - » They might be interleaved into common streams
    - » Separate streams are by nature independent
- These flows can be explicitly managed using `select()`
- `select()`
  - Several flavors – w/ and w/o timeout and/or sigmask
    - » Timeouts allow non-blocking behavior – return if nothing has happened
  - Watches 3 independent sets of fds
    - » Read set – return when a read will not block
    - » Write set – return when a write will not block
    - » Exception set – look for exceptions

27

## Event-Based Concurrent Servers Using I/O Multiplexing

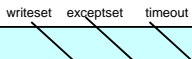
- Maintain a pool of connected descriptors
- Repeat the following forever:
  - Use the Unix `select` function to block until:
    - » (a) New connection request arrives on the listening descriptor
    - » (b) New data arrives on an existing connected descriptor
  - If (a), add the new connection to the pool of connections
  - If (b), read any available data from the connection
    - » Close connection on EOF and remove it from the pool

28

## The `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.

29

## Select idea

- On call
  - `read_set` specifies which flows you care about
  - Barring timeout
    - » On any change return
- On return
  - `read_set` specifies which flows can be read
    - » E.g. which flows had the appropriate event
  - Return value is the # of flows that had something happen on them
    - » 0 indicates nothing to be done
    - » >1 → # of 1's in the `read_set`
- Note
  - The same bit vector is used as a global variable
  - → a need for manipulation macros to deal with the bit vector

30

## 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?

31

## 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);
}
```

32

## 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);
    }
}
```

33

## 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);
    Rio_readn(connfd, buf, BUFSIZE);
    Rio_writen(connfd, buf, strlen(buf));
    Close(connfd);
} /* while */
```

34

## Event-based Concurrent Echo Server

```
/*
 * 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 */
```

35

## Event-based Concurrent Server (cont)

```
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.read_set = pool.read_set;
        pool.nready = Select(pool.maxfd+1, &pool.read_set,
                             NULL, NULL, NULL);

        if (FD_ISSET(listenfd, &pool.read_set)) {
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }
        check_clients(&pool);
    }
}
```

36

## Event-based Concurrent Server (cont)

```
/* 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);
}

```

37

## Event-based Concurrent Server (cont)

```
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");
}

```

38

## Event-based Concurrent Server (cont)

```
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->read_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;
            }
        }
    }
}

```

39

## 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.
  - Metric: # of instructions executed per network event
  - More events → more instructions →
- If the process blocks for some reason, everything the server is doing hangs up
  - Why would the process block?

40

## Summary

- **Processes:**
  - Easy to program, strong isolation provided by OS
  - High overhead, hard to share
- **Threads**
  - Somewhat easy to program, good performance, easy sharing
- **Events**
  - Best performance, easy sharing
  - Difficult programming model – concurrency implemented by you

41