Concurrent Programming is Hard!

- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
- Classical problem classes of concurrent programs:
  - Races: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - Deadlock: improper resource allocation prevents forward progress
  - Lifelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress
- Many aspects of concurrent programming are beyond the scope of 15-213

Iterative Servers

Iterative servers process one request at a time.

- Client 1 blocks waiting for user to type in data
- Client 2 blocks waiting to complete its connection request until after lunch!
- Server blocks waiting for data from Client 1

Solution: use concurrent servers instead.

- Concurrent servers use multiple concurrent flows to serve multiple clients at the same time.
Concurrent Servers: Multiple Processes

Concurrent servers handle multiple requests concurrently.

User goes out to lunch

Client 1 blocks waiting for user to type in data

Three Basic Mechanisms for Creating Concurrent Flows

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

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

3. Threads
   - Kernel automatically interleaves multiple logical flows.
   - Each flow shares the same address space.
   - Hybrid of processes and I/O multiplexing!

Process-Based Concurrent Server

```c
#include <ics.h>
#define BUFSIZE 1024

void echo(int connfd);
void handler(int sig);

int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>
        exit(0);
    }

    portno = atoi(argv[1]);
    listenfd = open_listenedfd(portno);
}
```

Process-Based Concurrent Server (cont)

```c
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)

```c
/* 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

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.
The select Function

select() sleeps until one or more file descriptors in the set readset are ready for reading.

```c
#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

```c
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

```c
/*
 * main loop: wait for connection request or stdin command.
 * If connection request, then echo input line and close connection. If stdin command, then process.
 */

int main() {
    int listenfd;
    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);
    while (notdone) {
        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);
        FD_ISSET(0, &readfds);    /* 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);
                    break;
            }
        }
    }
    return 0;
}
```

select Example (cont)

First we check for a pending event on stdin.
select Example (cont)

Next we check for a pending connection request.

```c
/* 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 */
```

Event-based Concurrent Echo Server

```c
/* 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 */
```

Event-based Concurrent Server (cont)

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

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

Event-based Concurrent Server (cont)

```c
/* 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);
}
```

```c
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;
```
Event-based Concurrent Server (cont)

```c
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;
        } /* Couldn't find an empty slot */
    app_error("add_client error: Too many clients");
}
```

Event-based Concurrent Server (cont)

```c
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->ready_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;
            }
        } else {/* Descriptor not ready, keep as is */
            break;
        }
    }
}
```

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?

Threads provide a middle ground between processes and I/O multiplexing...

Traditional View of a Process

Process = process context + code, data, and stack

```plaintext
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
- stack
- shared libraries
- run-time heap
- read/write data
- read-only code/data
```
Alternate View of a Process

Process = thread + code, data, and kernel context

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

Code and Data
- shared libraries
- run-time heap
- read/write data
- read-only code/data

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

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)

Thread 1 (main thread)
- stack 1
- read-only code/data
- stack 1
- read/write data

Thread 1 context:
- Data registers
- Condition codes
- SP1
- PC1

Thread 2 (peer thread)
- stack 2
- read-only code/data

Thread 2 context:
- Data registers
- Condition codes
- SP2
- PC2

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
- T1
- T2
- T3
- T4
- T5

Process hierarchy
- P0
- P1
- sh
- sh
- sh
- foo
- sh
- bar

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 (next lecture)
  - pthread_mutex_init
  - pthread_mutex_unlock
  - pthread_cond_init
  - pthread_cond_timedwait

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;
}
```

Execution of Threaded “hello, world”

- main thread
  - call Pthread_create()
  - Pthread_create() returns
  - call Pthread_join()
  - Pthread_join() returns
  - exit() terminates main thread and any peer threads

- peer thread
  - printf() return NULL;
  - (peer thread terminates)
Thread-Based Concurrent Echo Server

```c
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>
        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)

```c
* 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)*