Exceptional Control Flow: Processes and Signals (Cont.)

B&O Readings: 8.4-8.8
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Safe Signal Handling

- **Proceed with caution: signal-handling is a tricky business**
  - The handler executes concurrently with the main program.
  - Shared data structures can become corrupted.
  - Can have nested signal handlers.
  - Certain library functions are ***NOT SAFE*** to invoke within a signal handler!

Guidelines for Writing Safe Handlers

- **G0: Keep your handlers as simple as possible**
  - e.g., Set a global flag and return
- **G1: Call only async-signal-safe functions in your handlers**
  - **`printf`**, **`sprintf`**, **`malloc`**, and **`exit`** are not safe!
- **G2: Save and restore `errno` on entry and exit**
  - So that this or other handlers don't overwrite the value of **`errno`** used by the application code
- **G3: Protect accesses to shared data structures by temporarily blocking all signals.**
  - To prevent possible corruption
- **G4: Declare global variables as `volatile`**
  - To prevent compiler from storing them in a register
- **G5: Declare global flags (integer) as `sig_atomic_t`**
  - Individual read / write will be atomic (e.g. flag = 1, not flag++)

Async-Signal-Safety

- **Function is *async-signal-safe* if either reentrant (e.g., all variables stored on stack frame, CS:APP3e 12.7.2) or non-interruptible by signals.**

- **Posix guarantees 117 functions to be async-signal-safe**
  - Source: "`man 7 signal`"
  - Popular functions on the list:
    - `_exit, write, wait, waitpid, sleep, kill`
  - Popular functions that are **not** on the list:
    - `printf`, `sprintf`, `malloc, exit`
    - Unfortunate fact: `write` is the only async-signal-safe output function

Safely Generating Formatted Output

- **Use the reentrant SIO (Safe I/O library) from `csapp.c` in your handlers.**
  - `ssize_t sio_puts(char s[]) /* Put string */`
  - `ssize_t sio_putl(long v)    /* Put long */`
  - `void sio_error(char s[])    /* Put msg & exit */`

```
void sigint_handler(int sig) /* Safe SIGINT handler */
{
    Sio_puts("So you think you can stop the bomb with ctrl-
c, do you?\n");
    sleep(2);
    Sio_puts("Well...");
    sleep(1);
    Sio_puts("OK. :-)\n");
    _exit(0);
}
```

sigintsafe.c

Why Does the Execution Hang?

```c
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) ; /* Parent spins */
}
```

forks.c

```
linux> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
(hang)
```

Buggy Code

```c
int ccount = 0;
void child_handler(int sig) {
    int olderrno = errno;
    pid_t pid;
    if ((pid = wait(NULL)) < 0)
        Sio_error("wait error");
    ccount--;
    Sio_puts("Handler reaped child ");
    Sio_putl((long)pid);
    Sio_puts(" \n");
    sleep(1);
    errno = olderrno;
}

void fork14() {
    pid_t pid[N];
    int i;
    ccount = N;
    Signal(SIGCHLD, child_handler);

    for (i = 0; i < N; i++) {
        if ((pid[i] = Fork()) == 0) {
            Sleep(1);
            exit(0);  /* Child exits */
        }
    }
    while (ccount > 0) ; /* Parent spins */
}
```

forks.c

- **Pending signals are not queued**
  - For each signal type, one bit indicates whether or not signal is pending…
  - …thus at most one pending signal of any particular type.
- **You can't use signals to count events!**

```
linux> ./forks 14
Handler reaped child 23240
Handler reaped child 23241
(hang)
```

Correct Signal Handling

- **Must wait for all terminated child processes**
  - Put `wait` in a loop to reap all terminated children

```
void child_handler2(int sig)
{
    int olderrno = errno;
    pid_t pid;
    while ((pid = waitpid(-1, NULL, WNOHANG)) > 0) {
        ccount--;
        Sio_puts("Handler reaped child ");
        Sio_putl((long)pid);
        Sio_puts(" \n");
    }
    if (errno != ECHILD)
        Sio_error("wait error");
    errno = olderrno;
}
```

```
linux> ./forks 15
Handler reaped child 23246
Handler reaped child 23247
Handler reaped child 23248
Handler reaped child 23249
Handler reaped child 23250
linux>
```

Blocking and Unblocking Signals

- **Why might we want to block / unblock signals?**
- **Implicit blocking mechanism**
  - Kernel blocks any pending signals of type currently being handled.
  - E.g., A SIGINT handler can't be interrupted by another SIGINT
- **Explicit blocking and unblocking mechanism**
  - ```
    sigprocmask(int how, const sigset_t *set,
                sigset_t *oldset);
    ```
- **Supporting functions**
  - `sigemptyset` – Create empty set
  - `sigfillset` – Add every signal number to set
  - `sigaddset` – Add signal number to set
  - `sigdelset` – Delete signal number from set
    ```

Temporarily Blocking Signals

```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);

:    /* Code region that will not be interrupted by SIGINT */

/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

Example of a Simple Shell

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    char cmdline[MAXLINE];

    Signal(SIGCHLD, handler); /* Install SIGCHLD handler */
    initjobs(); /* Initialize the job list */
    Sigfillset(&mask_all);

    while (1) {
        get_cmd(cmdline, MAXLINE, stdin);
        if ((pid = Fork()) == 0) { /* Child */
            Execve(cmdline, argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* In parent */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL); /* Restore */
    }
}
```

procmask1.c

Example of a Simple Shell

```
void handler(int sig) {  /* SIGCHLD handler */
    int olderrno = errno;
    sigset_t mask_all, prev_all;
    pid_t pid;

    Sigfillset(&mask_all);
    while ((pid = waitpid(-1, NULL, 0)) > 0) { /* Reap child */
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all);
        deletejob(pid); /* Delete the child from the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL);
    }
    if (errno != ECHILD)
        Sio_error("waitpid error");
    errno = olderrno;
}
```
procmask1.c

- By blocking the signals, we make sure that accesses to the job data structure is synchronized.

Example of a Simple Shell (Buggy)

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    char cmdline[MAXLINE];

    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler); /* Install SIGCHLD handler */
    initjobs(); /* Initialize the job list */

    while (1) {
        get_cmd(cmdline, MAXLINE, stdin);
         if ((pid = Fork()) == 0) { /* Child */
             Execve(cmdline, argv, NULL);
         }
         Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* In parent */
         addjob(pid);  /* Add the child to the job list */
         Sigprocmask(SIG_SETMASK, &prev_all, NULL); /* Restore */
    }
}
```

procmask1.c

- **Ooops. This code is buggy. Where is the bug?**

Example of a Simple Shell (Buggy)

```c
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, prev_all;
    char cmdline[MAXLINE];

    Sigfillset(&mask_all);
    Signal(SIGCHLD, handler); /* Install SIGCHLD handler */
    initjobs(); /* Initialize the job list */

    while (1) {
        get_cmd(cmdline, MAXLINE, stdin);
        if ((pid = Fork()) == 0) { /* Child */
            Execve(cmdline, argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, &prev_all); /* In parent */
        addjob(pid);  /* Add the child to the job list */
        Sigprocmask(SIG_SETMASK, &prev_all, NULL); /* Restore */
    }
}
```

procmask1.c

- **There is a subtle synchronization error: what if the child finishes and the handler invoked before the addjob is called?**

Corrected Shell Program without Race

```
int main(int argc, char **argv)
{
    int pid;
    sigset_t mask_all, mask_one, prev_one;

    Sigfillset(&mask_all);
    Sigemptyset(&mask_one);
    Sigaddset(&mask_one, SIGCHLD);
    Signal(SIGCHLD, handler);
    initjobs(); /* Initialize the job list */

    while (1) {
        Sigprocmask(SIG_BLOCK, &mask_one, &prev_one); /* Block SIGCHLD */
        if ((pid = Fork()) == 0) { /* Child process */
            Sigprocmask(SIG_SETMASK, &prev_one, NULL); /* Unblock SIGCHLD */
            Execve("/bin/date", argv, NULL);
        }
        Sigprocmask(SIG_BLOCK, &mask_all, NULL); /* In
        addjob(pid);  /* Add the child to the job
        Sigprocmask(SIG_SETMASK, &prev_one, NULL);
    }
}
```

See other examples in textbook 8.5!

Block signals before fork!

Remember to unblock signals in child.

procmask2.c

What's the Output of This Program?

```
pid_t pid;
int counter = 2;
void handler1(int sig) {
    counter = counter - 1;
    Sio_put(counter);
    fflush(stdout);
    exit(0);
}
int main() {
    Signal(SIGUSR1, handler1);
    printf("%d", counter);
    fflush(stdout);
    if ((pid = fork()) == 0) {
        while(1) {};
    }
    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}
```

What's the Output of This Program?

```
pid_t pid;
int counter = 2;
void handler1(int sig) {
    counter = counter - 1;
    Sio_put(counter);
    fflush(stdout);
    exit(0);
}
int main() {
    Signal(SIGUSR1, handler1);
    printf("%d", counter);
    fflush(stdout);
    if ((pid = fork()) == 0) {
        while(1) {};
    }
    kill(pid, SIGUSR1);
    waitpid(-1, NULL, 0);
    counter = counter + 1;
    printf("%d", counter);
    exit(0);
}
```

Answer: 213

Concurrent Programming ~~and Synchronization~~

B&O Readings: Loosely based on Chp. 12 (12.1, 12.3-12.5)
CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Note: these slides were originally created by Markus Püschel at Carnegie Mellon University

Today

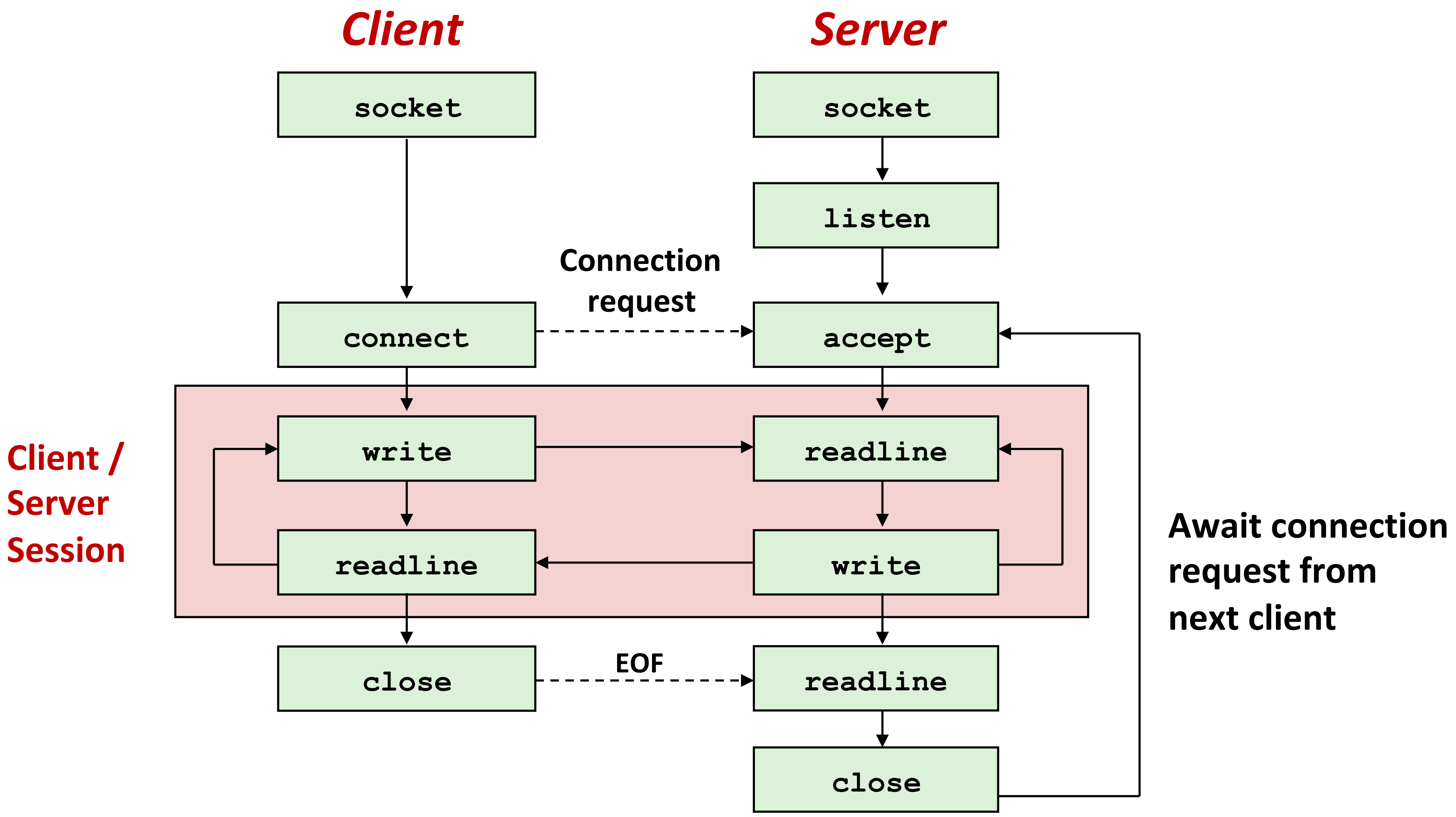
- **Threads: a mechanism for concurrency**
- **Synchronization in threaded programs**

Different Forms of Concurrency

- **Recall: logical control flows are *concurrent* if they overlap in time.**

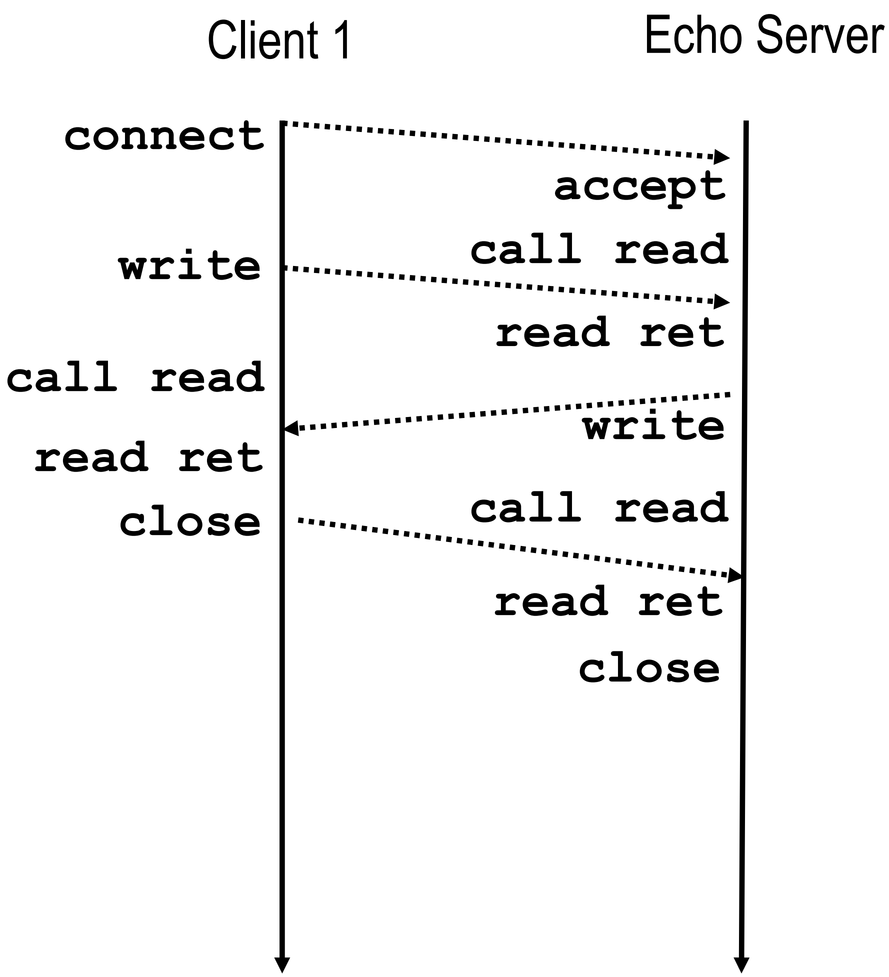
- **Ex 1: concurrency among different applications: concurrent processes running different applications on your desktop.**
- **Ex 2: concurrency within a single application: signal handler and the main logical control of your shell program.**

Motivation for Concurrent Programming: Iterative Echo Server



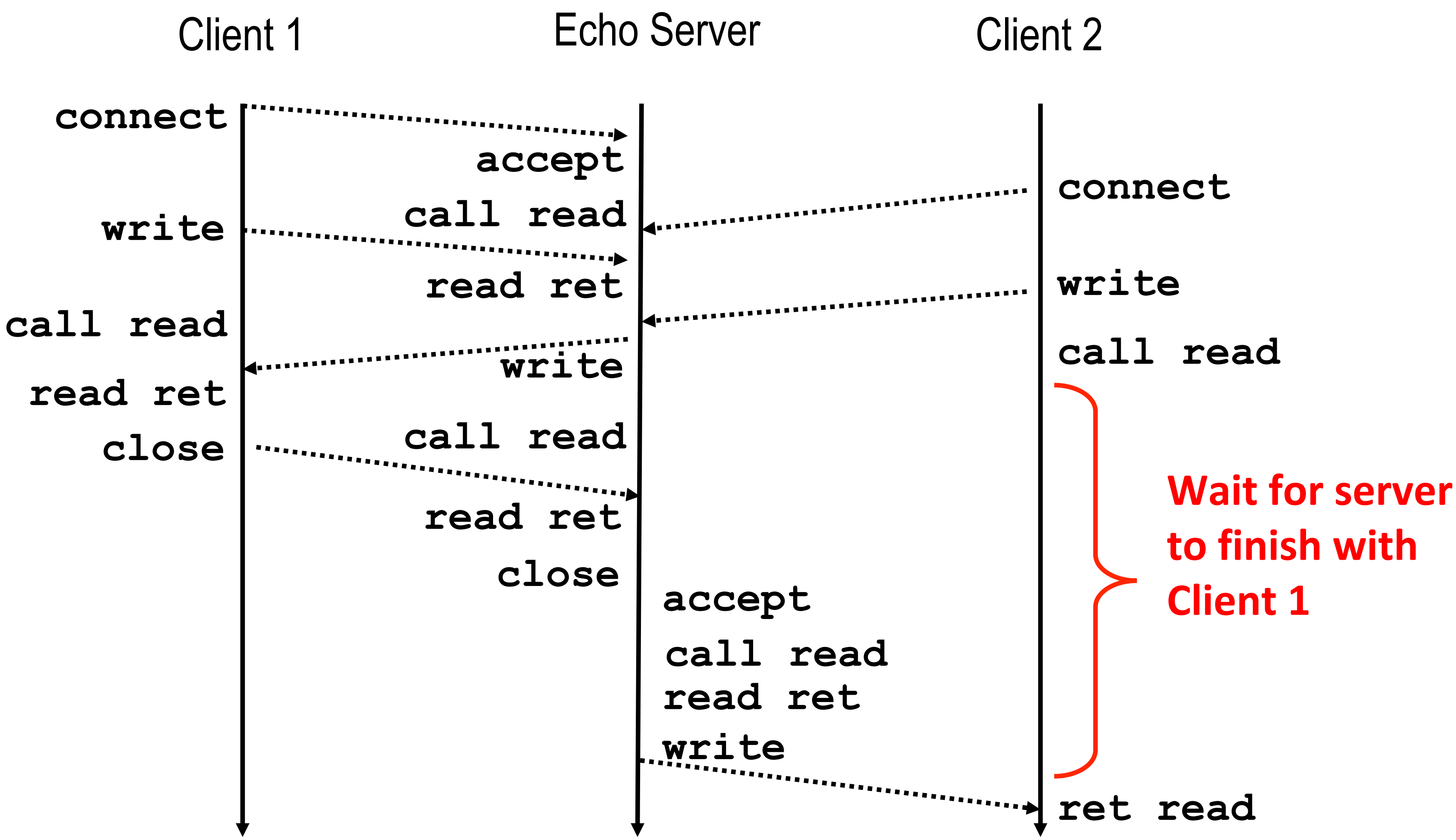
An Iterative Echo Server

- **Iterative servers process one request at a time**



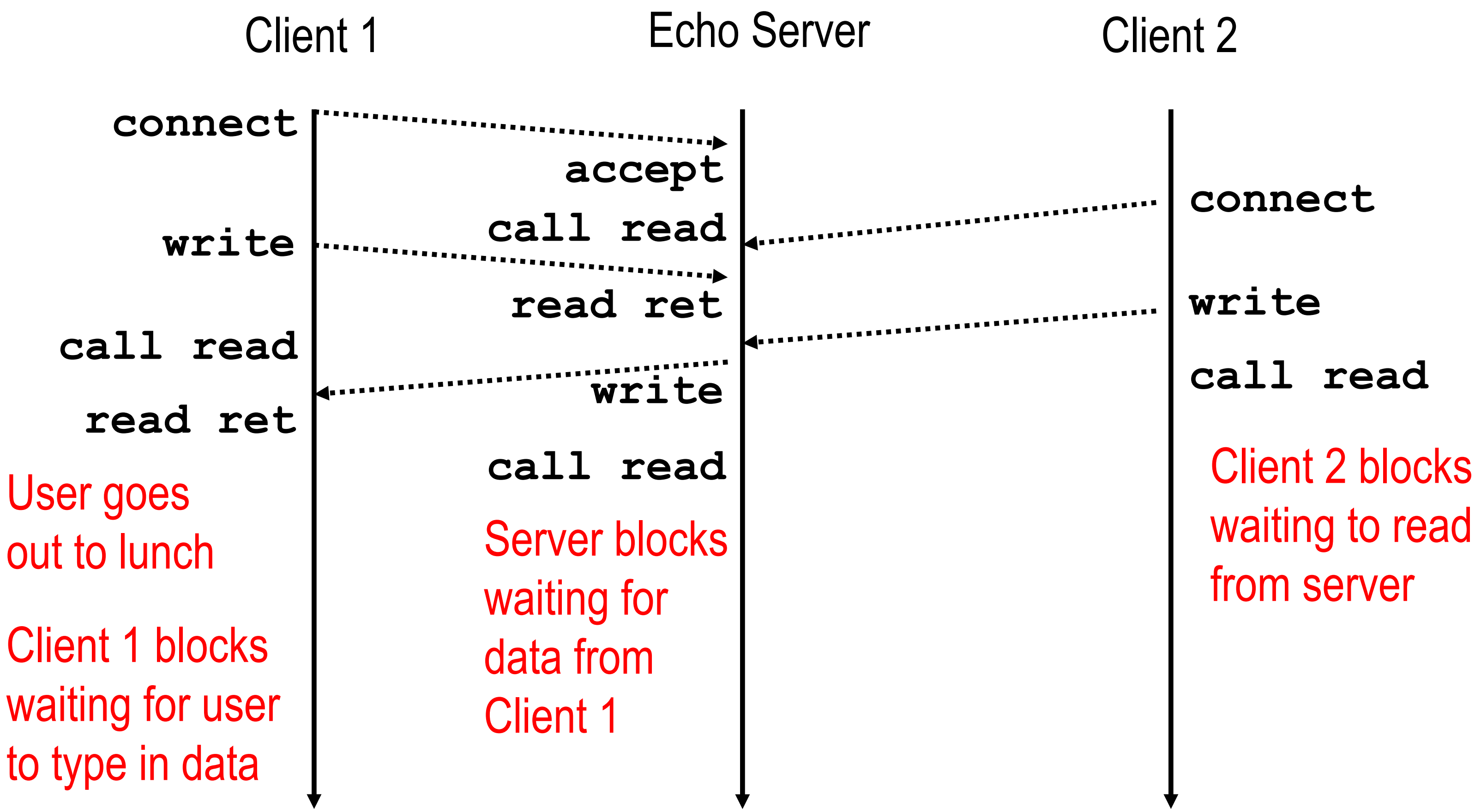
An Iterative Echo Server

- Iterative servers process one request at a time



Fundamental Flaw of An Iterative Server

- **Iterative servers process one request at a time**



- **Solution: use a *concurrent server*, which uses multiple concurrent flows to serve multiple clients at the same time.**

Approaches for Writing A Concurrent Server

Allow server to handle multiple clients concurrently

1. Process-based

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

2. Event-based (we won't cover this)

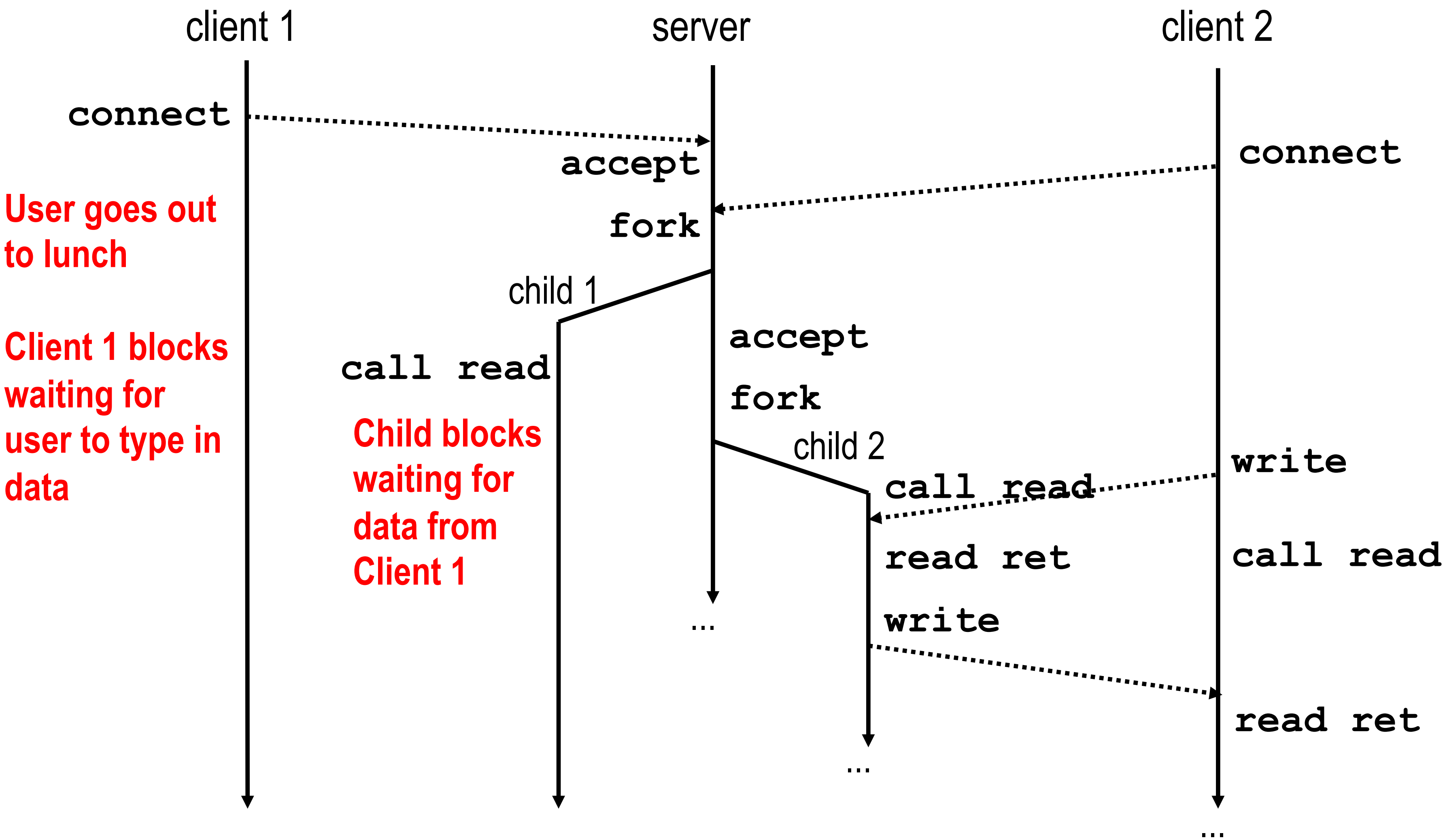
- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Uses technique called *I/O multiplexing*.

3. Thread-based

- Kernel automatically interleaves multiple logical flows
- Each flow shares the same address space
- Hybrid of of process-based and event-based.

Approach #1: Process-based Servers

- Spawn separate process for each client



Pros and Cons of Process-based Servers

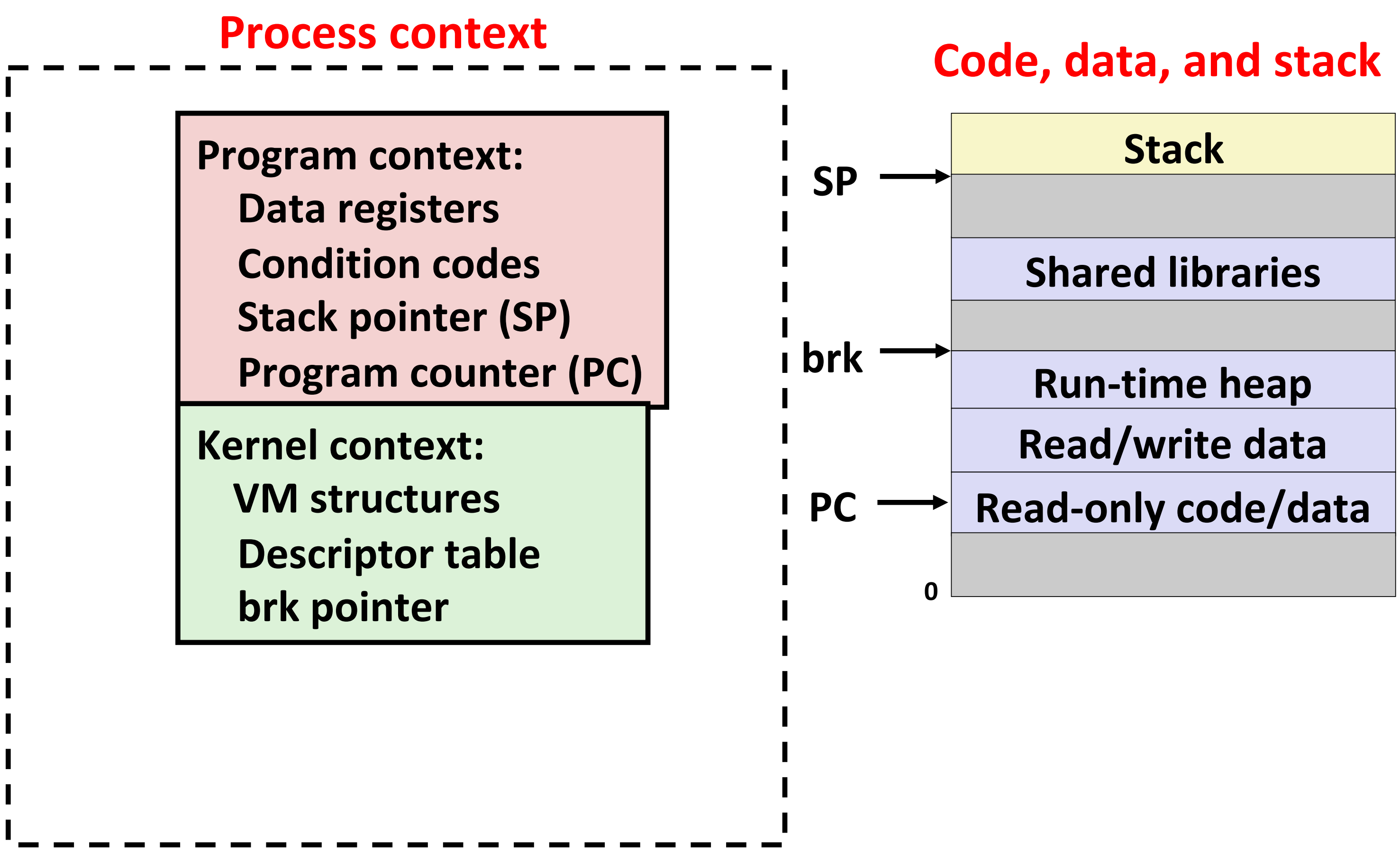
- **+ Handle multiple connections concurrently**
- **+ Clean sharing model**
  - separate address space
- **+ Simple and straightforward**
- **– Additional overhead for process control**
- **– Nontrivial to share data between processes**
  - Requires IPC (interprocess communication) mechanisms
    - Pipes, signals, explicitly share memory via mmap … etc.

Approach #3: Thread-based Servers

- **Very similar to approach #1 (process-based)**
  - …but using threads instead of processes

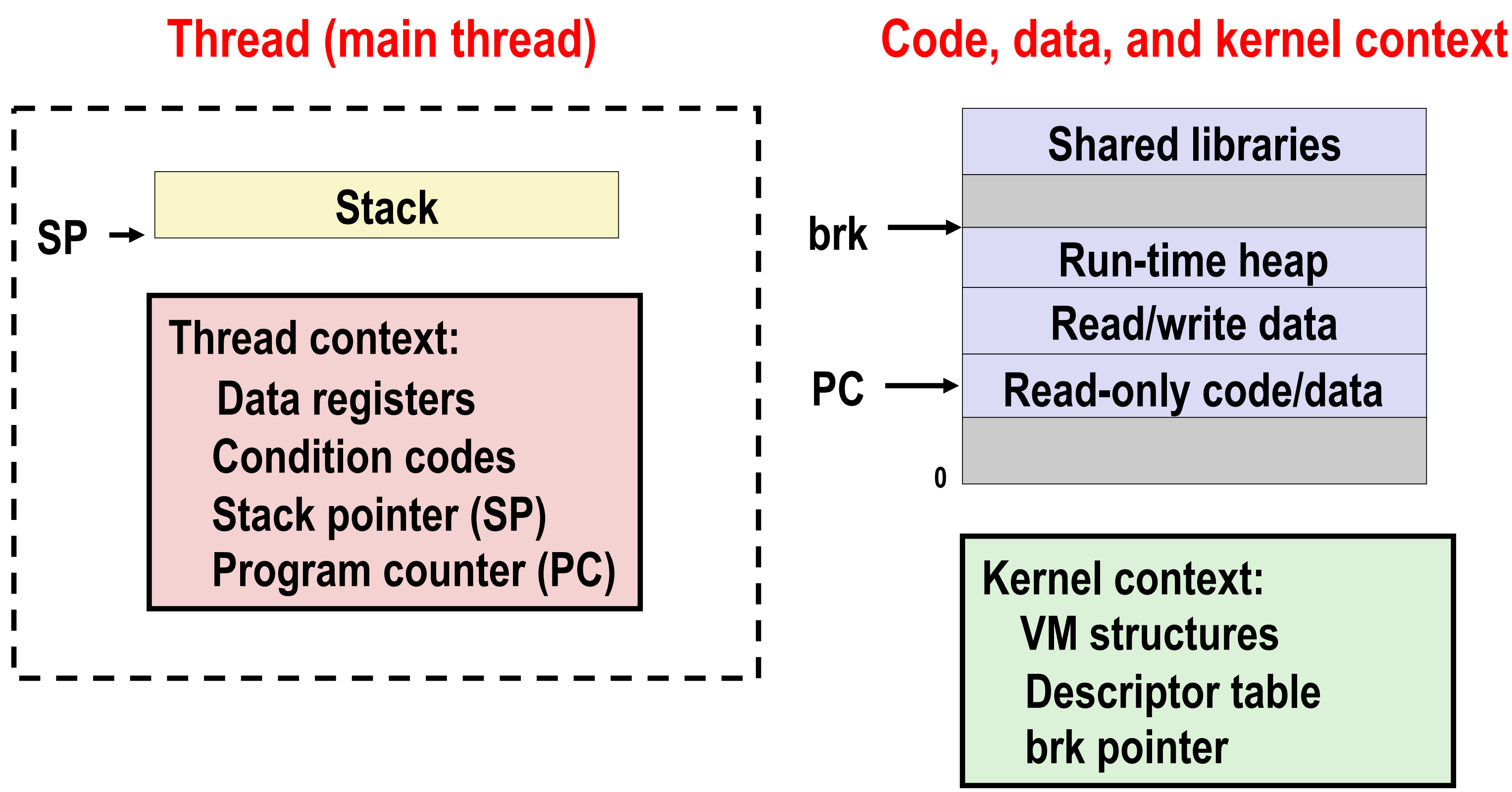
Traditional View of a Process

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



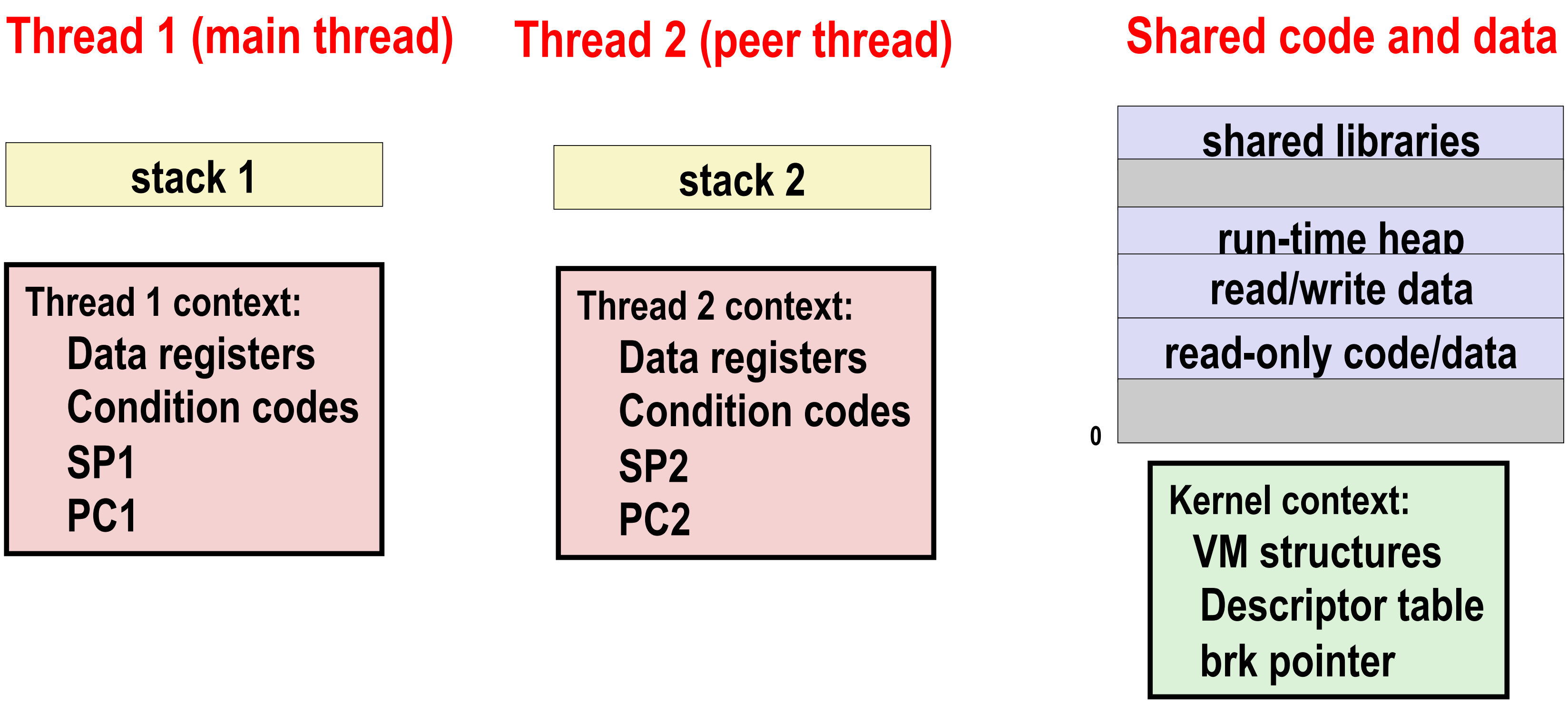
Alternate View of a Process

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



A Process With Multiple Threads

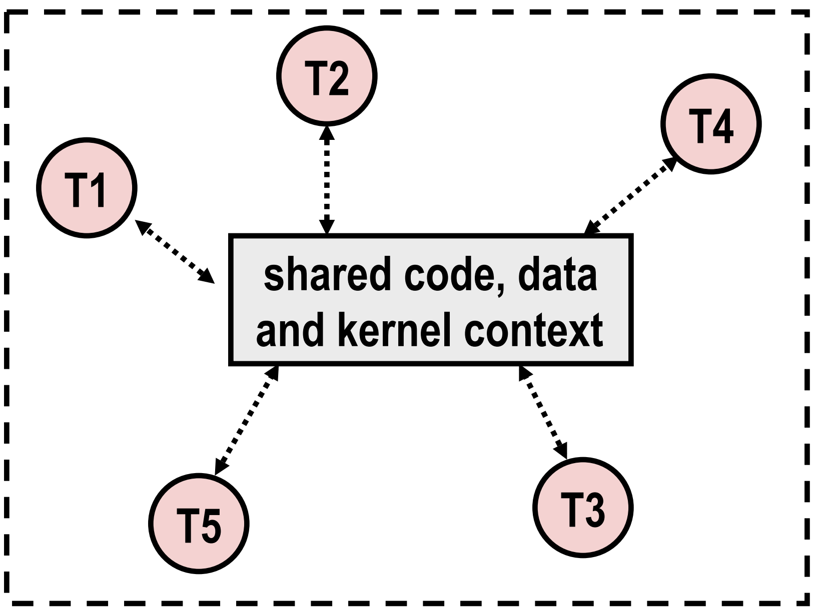
- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id (TID)
  - All threads share the same code, data, and kernel context



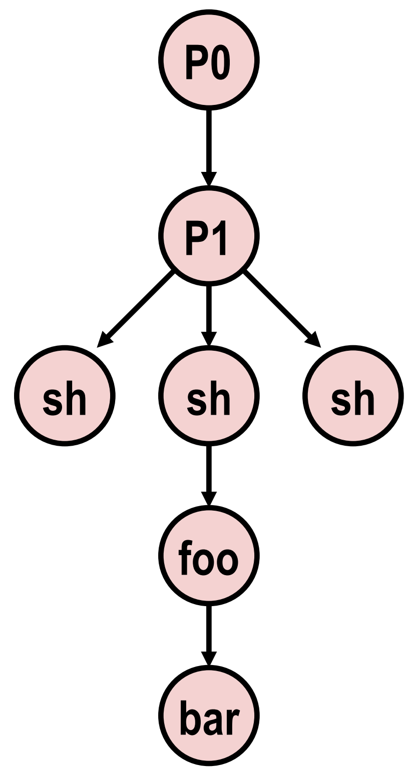
Logical View of Threads

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

Threads associated with process foo



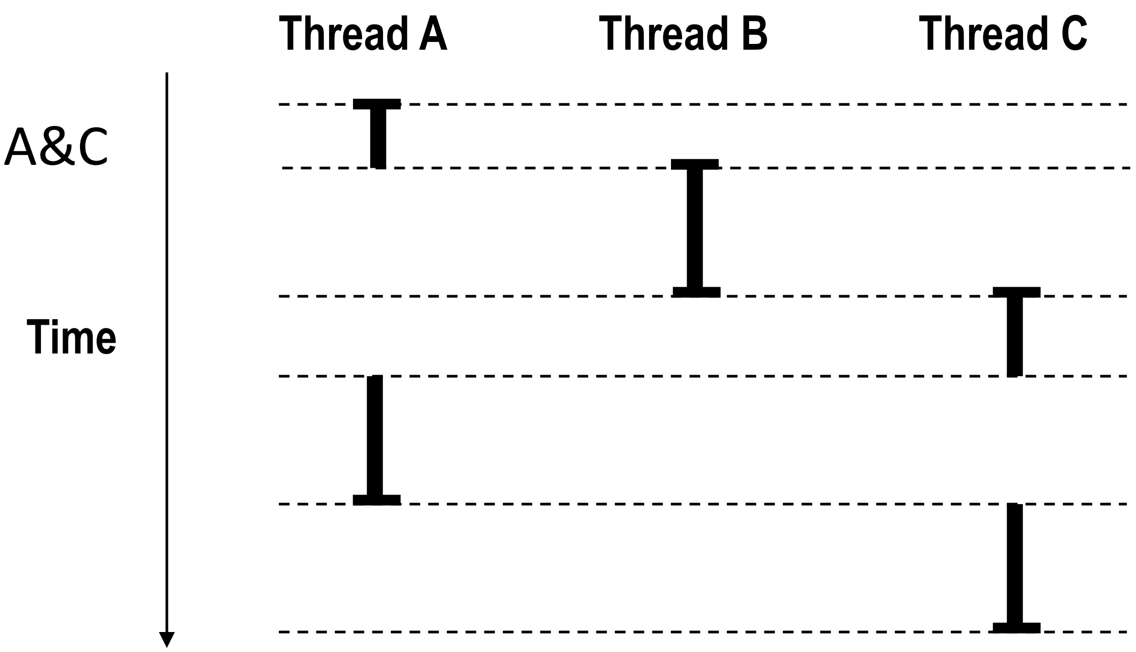
Process hierarchy



Concurrent Threads

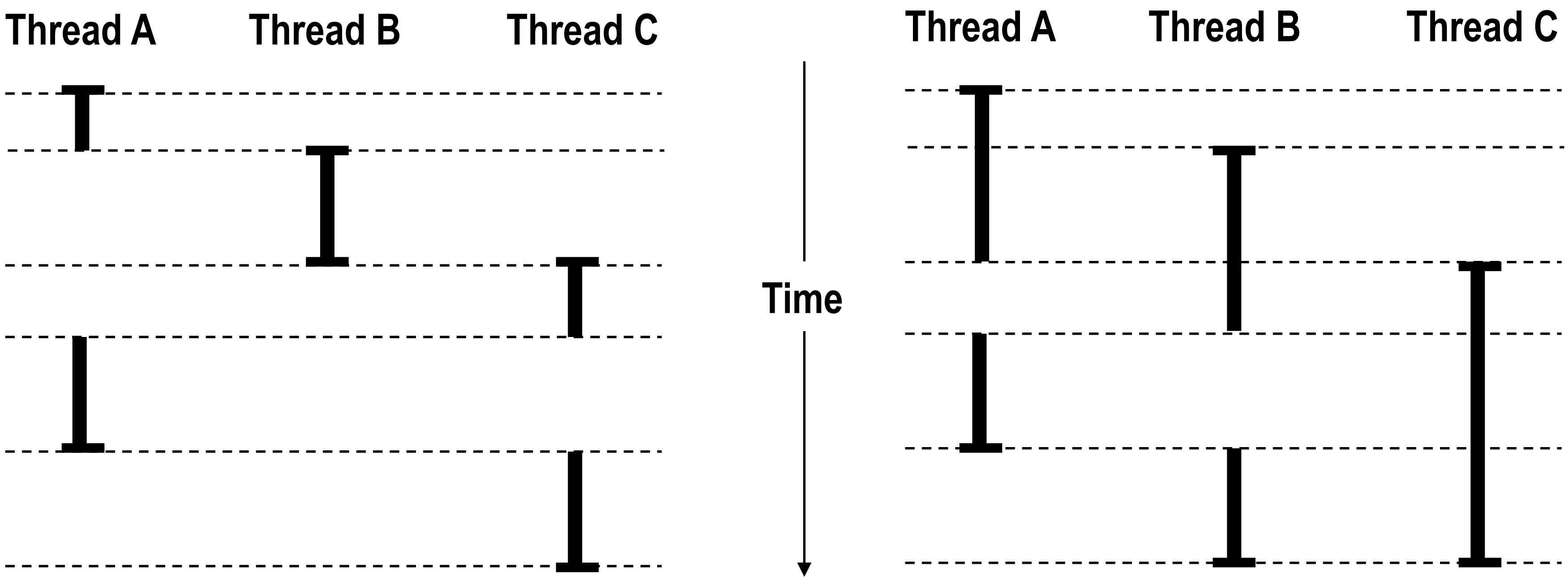
- **Two threads are *concurrent* if their flows overlap in time**
- **Otherwise, they are sequential**

- **Examples:**
  - Concurrent: A & B, A&C
  - Sequential: B & C



Concurrent Thread Execution

- **Single Core Processor**
  - Simulate concurrency by time slicing
- **Multi-Core Processor**
  - Can have true concurrency



Run 3 threads on 2 cores

Threads vs. Processes

- **How threads and processes are similar**
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- **How threads and processes are different**
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not
  - Threads are somewhat less expensive than processes
    - Process control (creating and reaping) twice as expensive as thread control
    - Linux numbers:
      - ~20K cycles to create and reap a process
      - ~10K cycles (or less) 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`

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);
}
```

hello.c

Thread ID

Thread attributes (usually NULL)

Thread routine

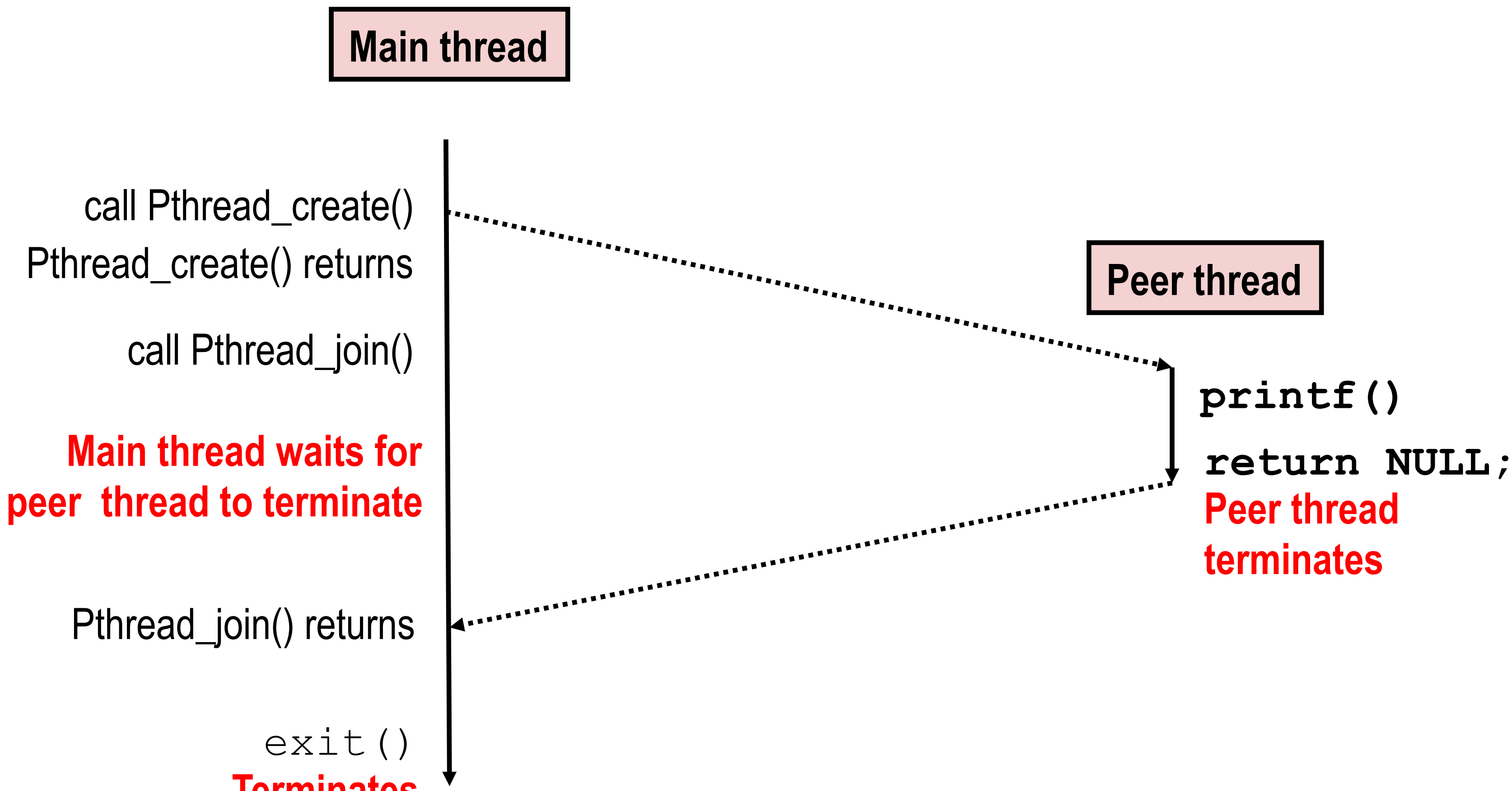
*Thread arguments (void *p)*

*Return value (void **p)*

```c
void *thread(void *vargp) /* thread routine */
{
    printf("Hello, world!\n");
    return NULL;
}
```

hello.c

Execution of Threaded "hello, world"



Note: **`pthread_join`** is akin to reaping the thread; resources associated with the thread is not freed until it's reaped.

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
  - Hard to know which data shared & which private
  - Hard to detect by testing
    - Probability of bad race outcome very low, but nonzero!
  - All functions called by threads must be thread-safe.

Concurrent Programming is Hard!

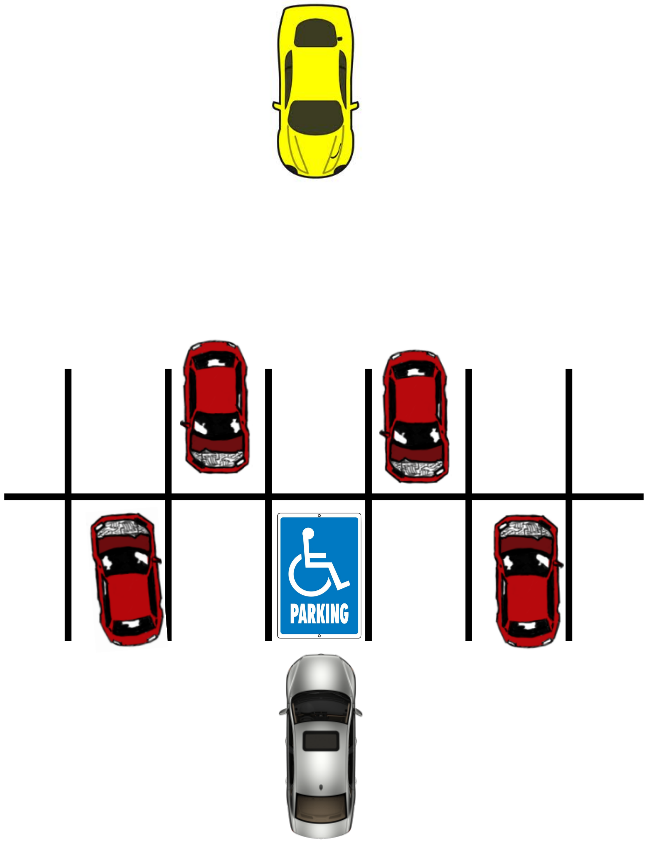
- **Need to reason about concurrent events occurring at the same time.**
  - Events interact through shared state.
  - Reasoning about the correctness of your code involves reasoning about all possible interleaving of events.
    - Ex: job list in your shell program
  - The human mind tends to be sequential.

- **Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible**

Classical problems of concurrent programs

- ***Races:***
  - improper coordination when accessing shared resources concurrently
  - outcome depends on arbitrary scheduling decisions
  - Example: who gets the last seat on the airplane?
- ***Deadlock:***
  - improper resource allocation prevents forward progress
  - Example: traffic gridlock
- ***Livelock / Starvation / Fairness*:**
  - external events and/or system scheduling decisions can prevent sub-task progress
  - Example: people always jump in front of you in line

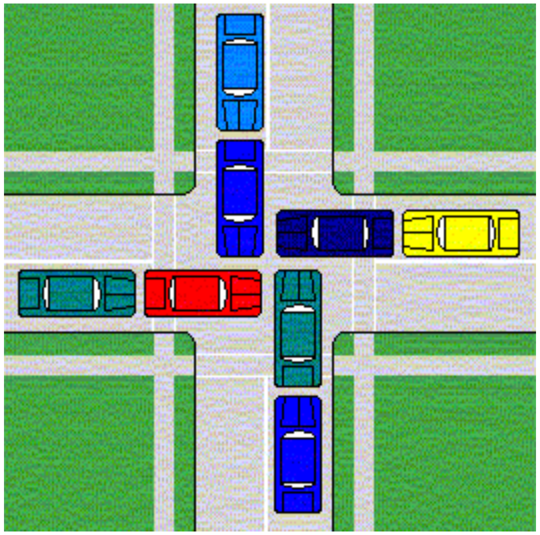
Data Race



Classical problems of concurrent programs

- ***Races:***
  - improper coordination when accessing shared resources concurrently
  - outcome depends on arbitrary scheduling decisions
  - Example: who gets the last seat on the airplane?
- ***Deadlock:***
  - improper resource allocation prevents forward progress
  - Example: traffic gridlock
- ***Livelock / Starvation / Fairness*:**
  - external events and/or system scheduling decisions can prevent sub-task progress
  - Example: people always jump in front of you in line

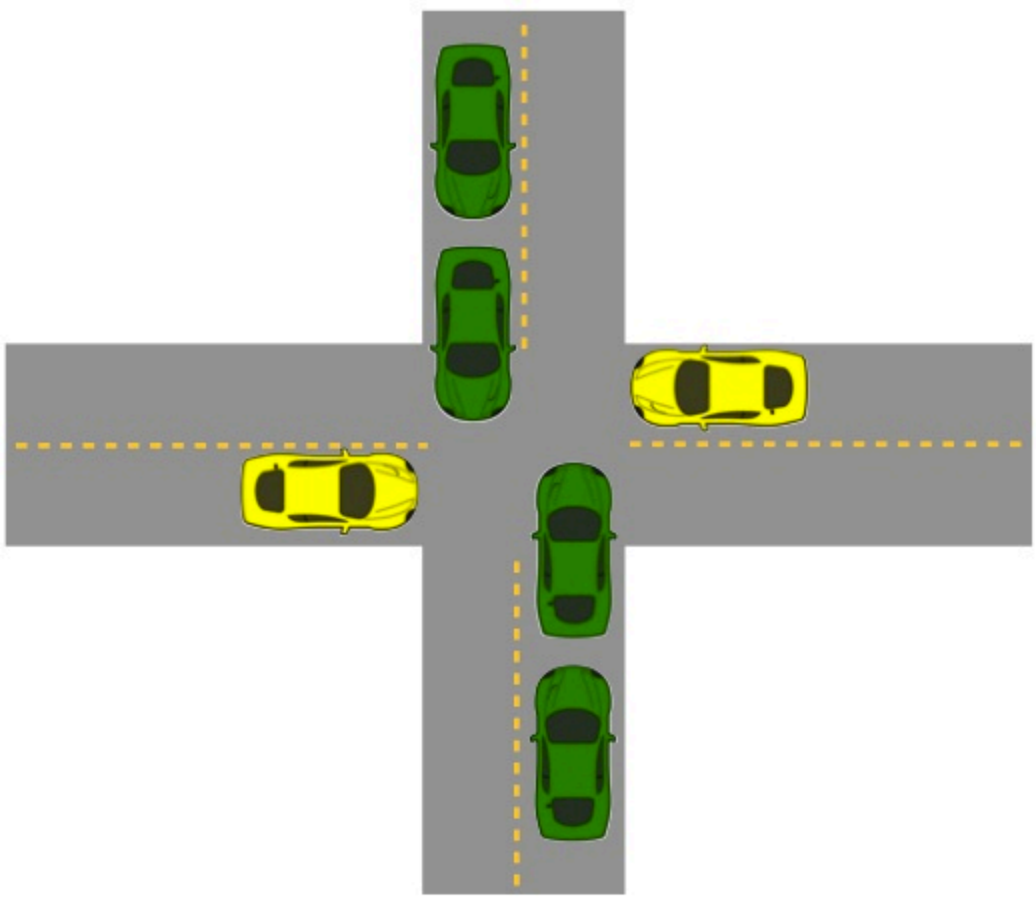
Deadlock



Classical problems of concurrent programs

- ***Races:***
  - improper coordination when accessing shared resources concurrently
  - outcome depends on arbitrary scheduling decisions
  - Example: who gets the last seat on the airplane?
- ***Deadlock:***
  - improper resource allocation prevents forward progress
  - Example: traffic gridlock
- ***Livelock / Starvation / Fairness*:**
  - external events and/or system scheduling decisions can prevent sub-task progress
  - Example: people always jump in front of you in line

Starvation



- Yellow must yield to green
- Continuous stream of green cars
- Overall system makes progress, but some individuals wait indefinitely

Classical problems of concurrent programs

- ***Races:***
  - improper coordination when accessing shared resources concurrently
  - outcome depends on arbitrary scheduling decisions
  - Example: who gets the last seat on the airplane?
- ***Deadlock:***
  - improper resource allocation prevents forward progress
  - Example: traffic gridlock
- ***Livelock / Starvation / Fairness*:**
  - external events and/or system scheduling decisions can prevent sub-task progress
  - Example: people always jump in front of you in line

Many aspects of concurrent programming are beyond the scope of our course...