Concurrency

Alan L. Cox
alc@rice.edu

Some slides adapted from CMU 15.213 slides

Objectives

Be able to apply three techniques for achieving concurrency

- **Process**
- **Thread**
- **Event-driven**

Be able to analyze the correctness of concurrent programs

- **Thread-safety**
- **Race conditions**
- **Deadlocks**

Be able to use synchronization to correctly share data

Concurrency

Perform multiple tasks at once (actually or logically)

- Hardware exception handlers
- Processes
- Unix signal handlers
- etc.

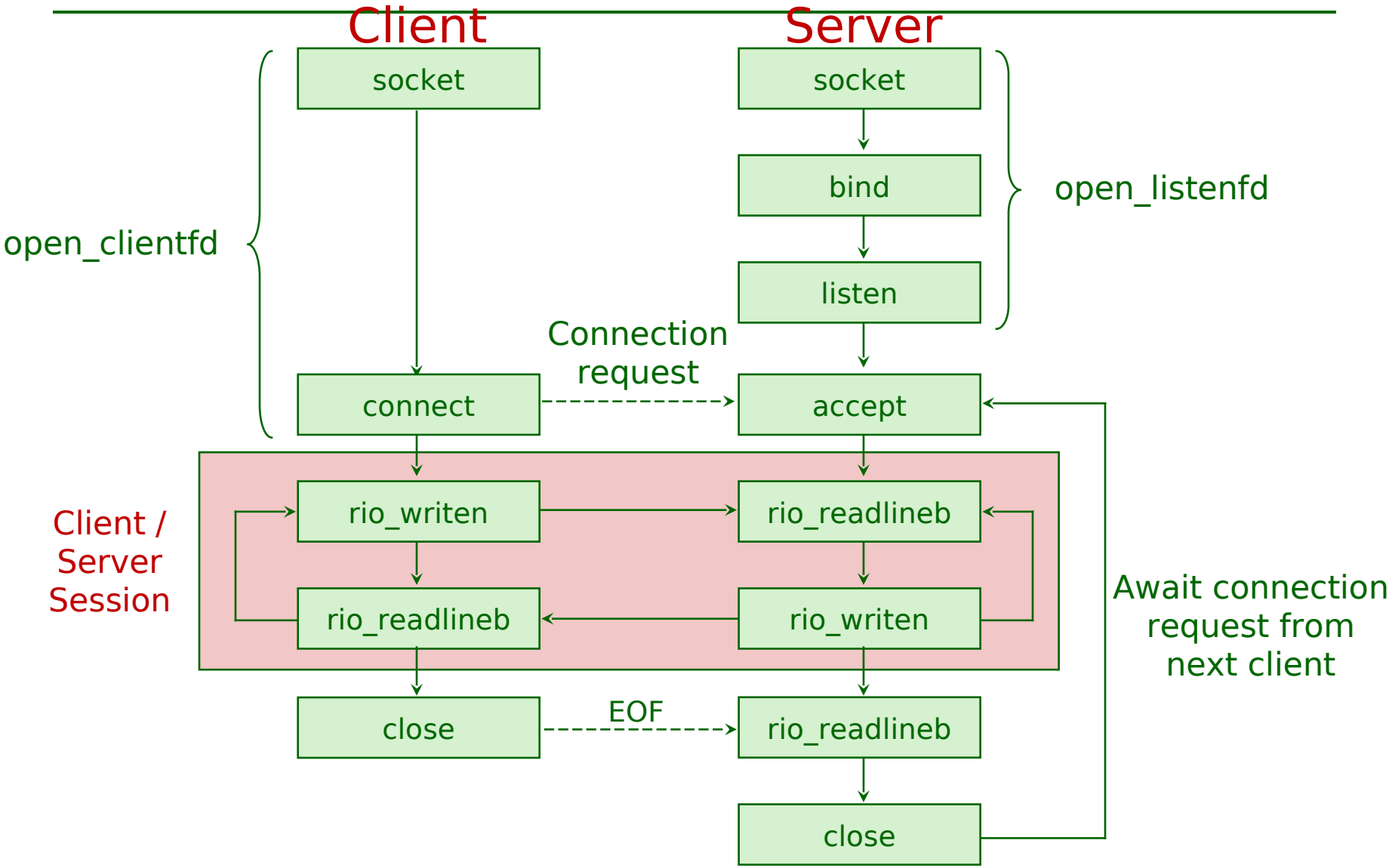
Useful in a variety of application-level contexts

- Computation on multiprocessors
- Accessing slow I/O devices
  - Similar to kernel overlapping disk transfers with other processes
- Interacting with humans
  - Modern windowing systems concurrently handle user inputs
- Reducing latency by deferring work
  - e.g., perform heap coalescing in the background
- Servicing multiple network clients

Concurrent Programming is Hard!

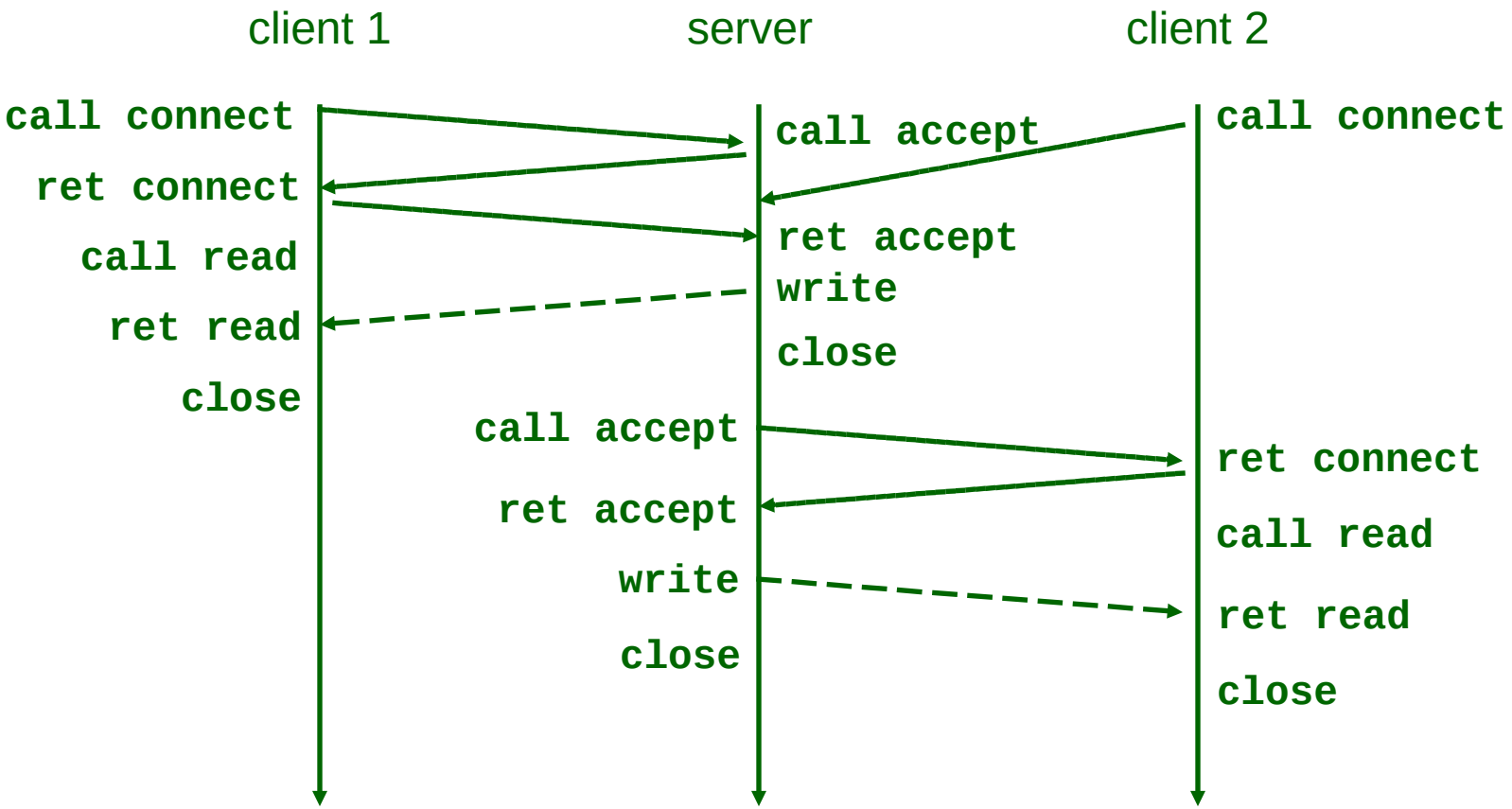
- The human mind tends to be sequential
- The notion of time is often misleading
- Thinking about all possible sequences of events in a computer system is at least error prone and frequently impossible
- Classical problem classes of concurrent programs:
  - Races: outcome depends on arbitrary scheduling decisions elsewhere in the system
  - Deadlock: improper resource allocation prevents forward progress
  - Livelock / Starvation / Fairness: external events and/or system scheduling decisions can prevent sub-task progress
- Many aspects of concurrent programming are beyond the scope of COMP 321

Echo Server Operation

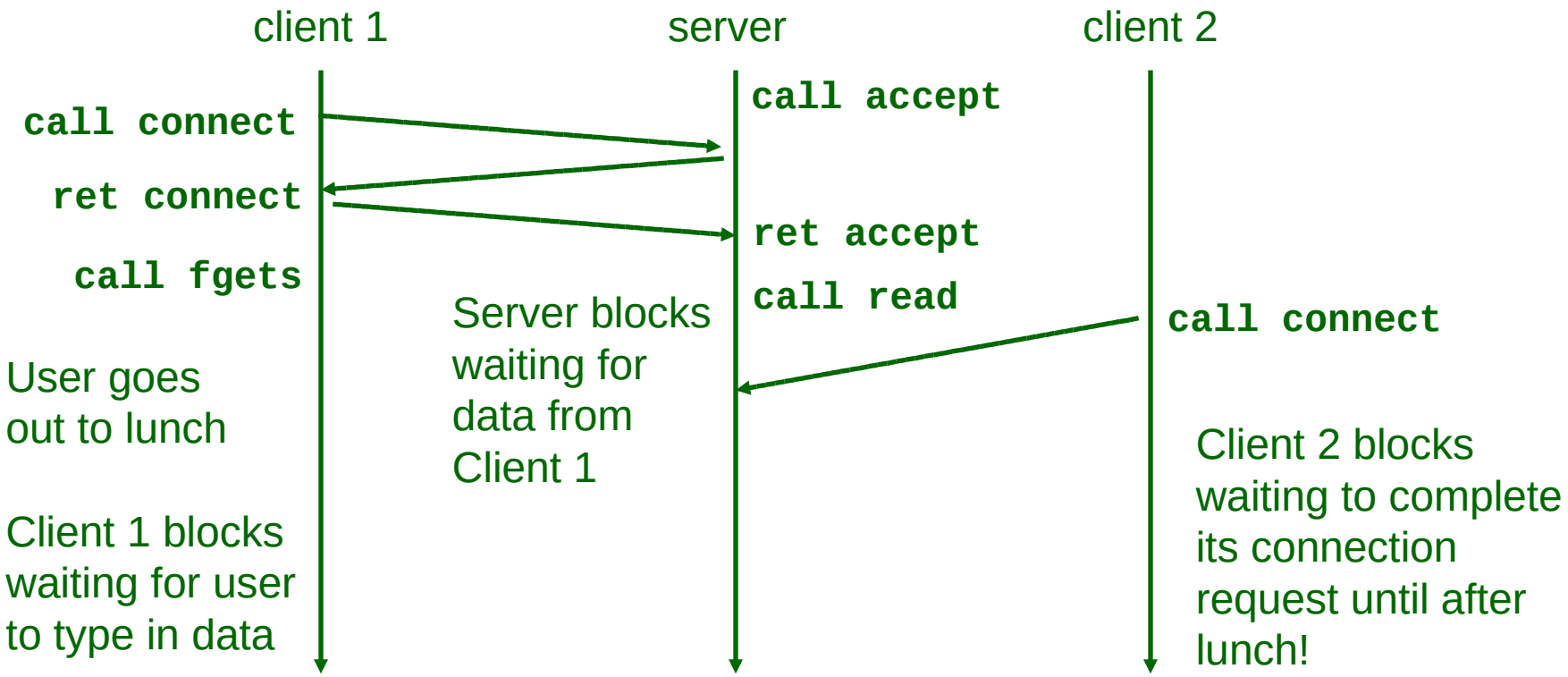


Iterative Servers

Iterative servers process one request at a time



Fundamental Flaw of Iterative Servers

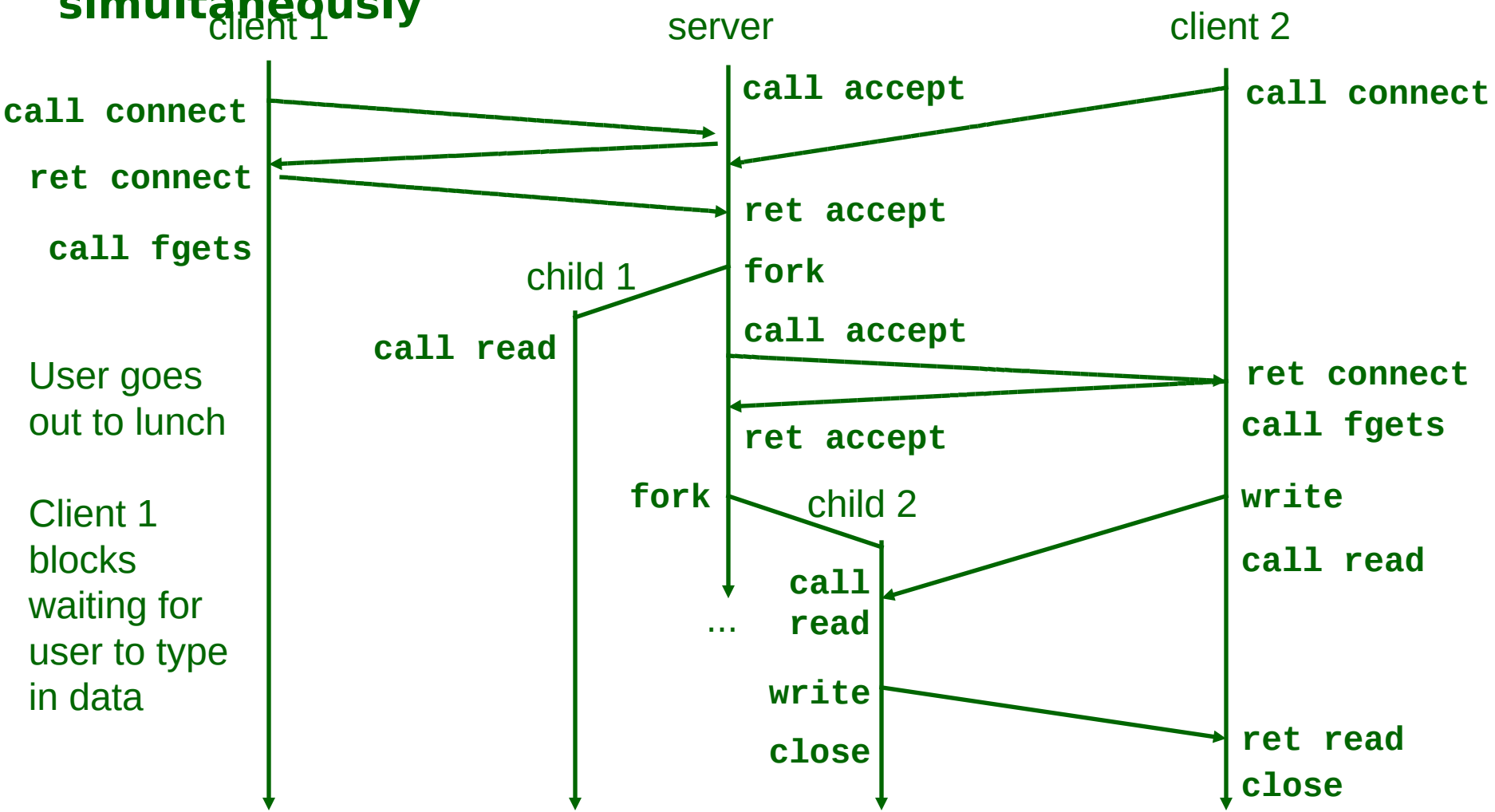


Solution: use concurrent servers instead

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

Concurrent Servers: Multiple Processes

Concurrent servers handle multiple requests simultaneously



Three Basic Mechanisms for Creating Concurrent Flows

1. Processes

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

2. Threads

- Kernel (or thread library) automatically interleaves multiple logical flows
- Each flow shares the same address space

3. I/O multiplexing

- User manually interleaves multiple logical flows
- Each flow shares the same address space
- Popular idea for high-performance server designs

Process-Based Concurrent Server

```
/* echoserverp.c - A concurrent echo server based on processes
 * Usage: echoserverp <port> */
#include <csapp.h>

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

int main(int argc, char **argv) {
  int listenfd, connfd;
  struct sockaddr_in clientaddr;
  socklen_t clientlen = sizeof(struct sockaddr_in);

  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(0);
  }

  listenfd = Open_listenfd(argv[1]);
  Signal(SIGCHLD, sigchld_handler); // Parent must reap children!
```

Process-Based Concurrent Server (cont)

```
  // Main server process runs forever!
  while (true) {
    connfd = Accept(listenfd, (SA *)&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!
  }
}
```

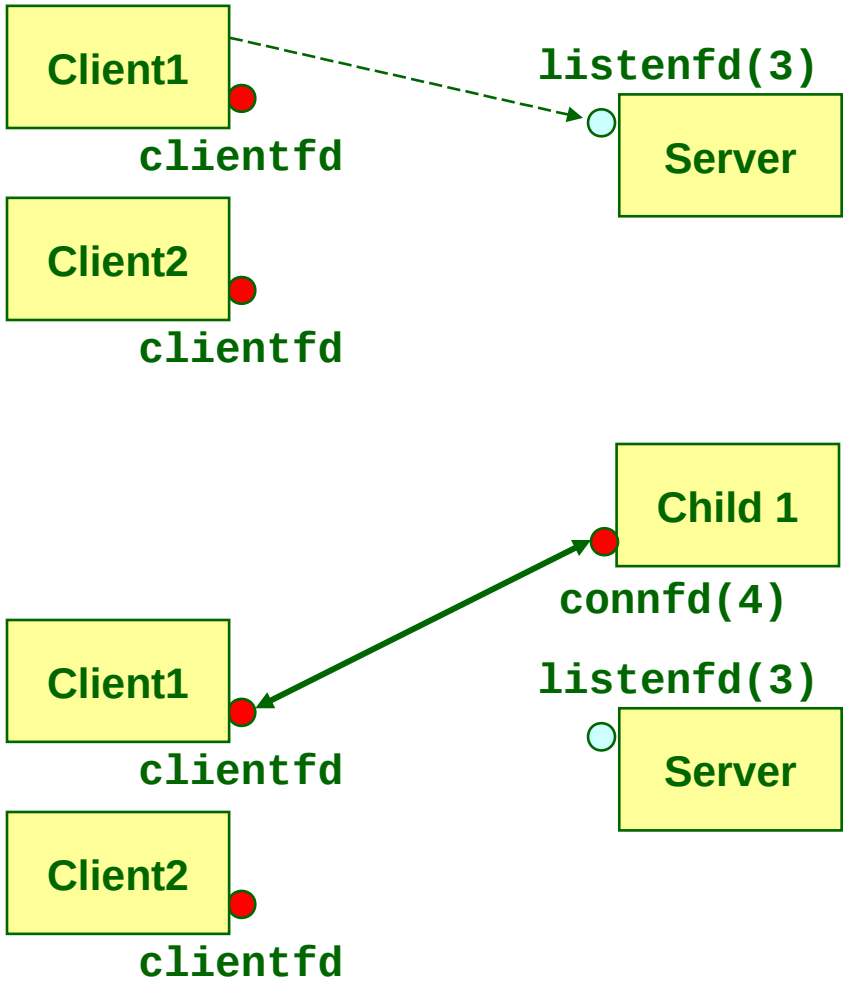
Sequential Main Server Loop

```
  while (true) {
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    echo(connfd);
    Close(connfd);
  }
```

Process-Based Concurrent Server (cont)

```
/*
 * handler - Reaps children as they terminate.
 */
void
handler(int sig)
{
  pid_t pid;

  while ((pid = waitpid(-1, 0, WNOHANG)) > 0)
    ;
  return;
}
```

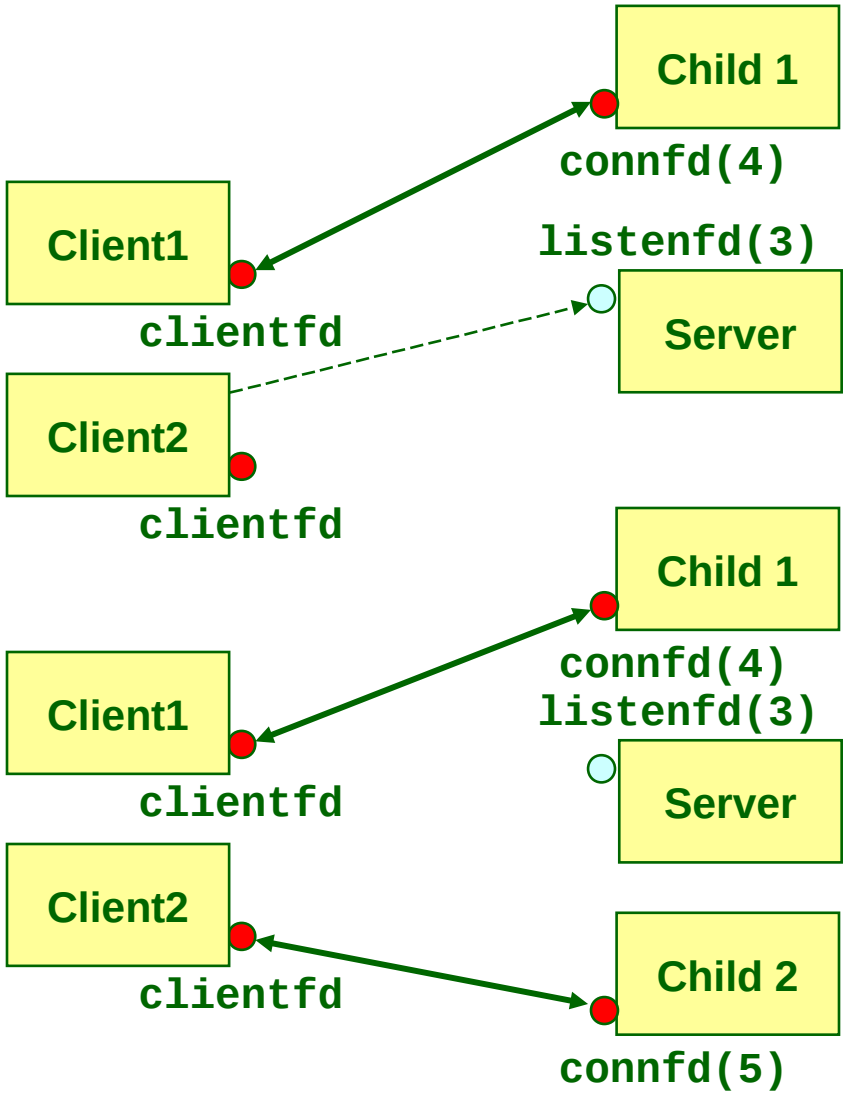
Process-based Server Illustrated



1. Server accepts a connection request from Client1

2. Server forks a child process to service Client1

Process-based Server Illustrated



3. Server accepts another connection request (from Client2)

4. Server forks another child process to service Client2

Implementation Issues

- **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**
    - **`Signal` wrapper tells kernel to automatically restart `accept`**
  - **Required for portability on some Unix systems**
- **Server must reap zombie children**
  - **Avoids fatal resource (process) 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**

Pros/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
    - Named pipes, shared memory, and semaphores

Road map

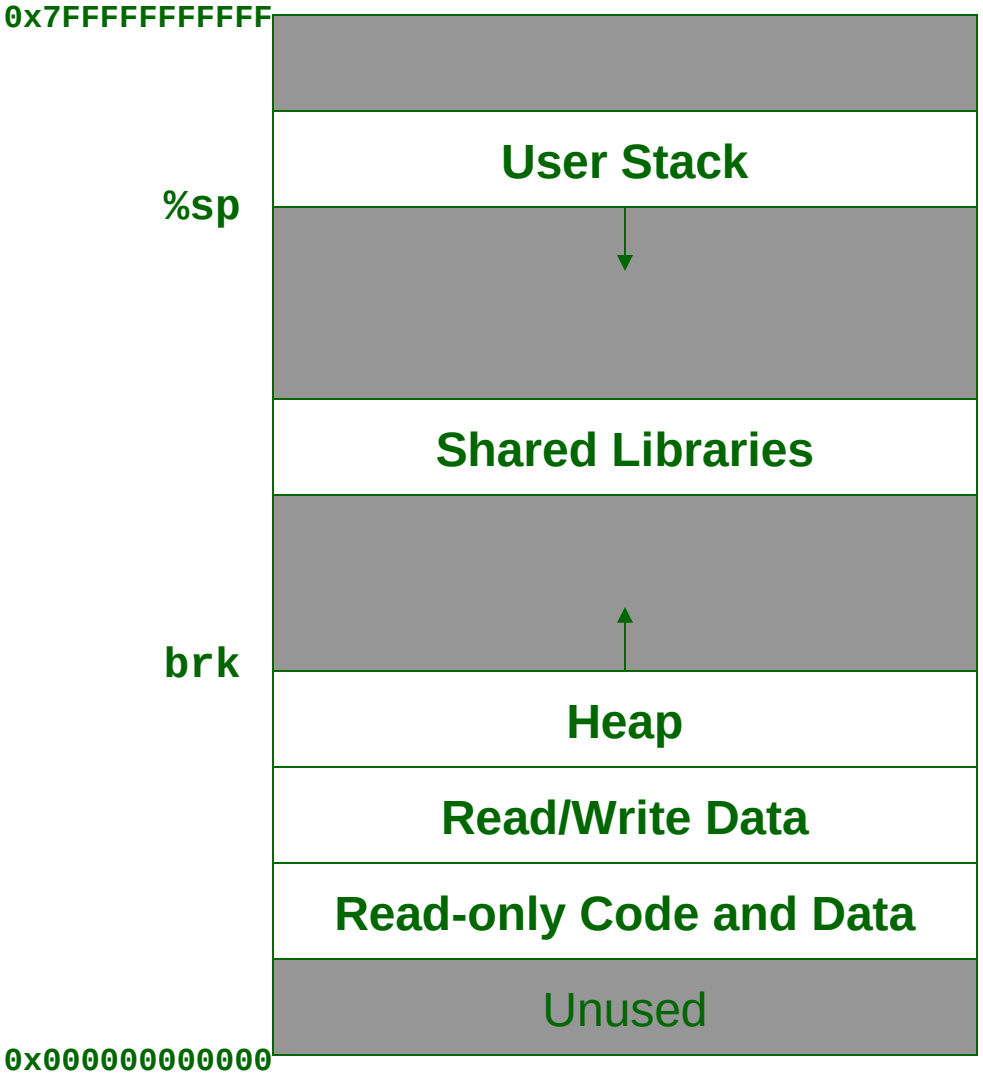
Process-based concurrency

Thread-based concurrency

Safe sharing using semaphore

Event-driven concurrency

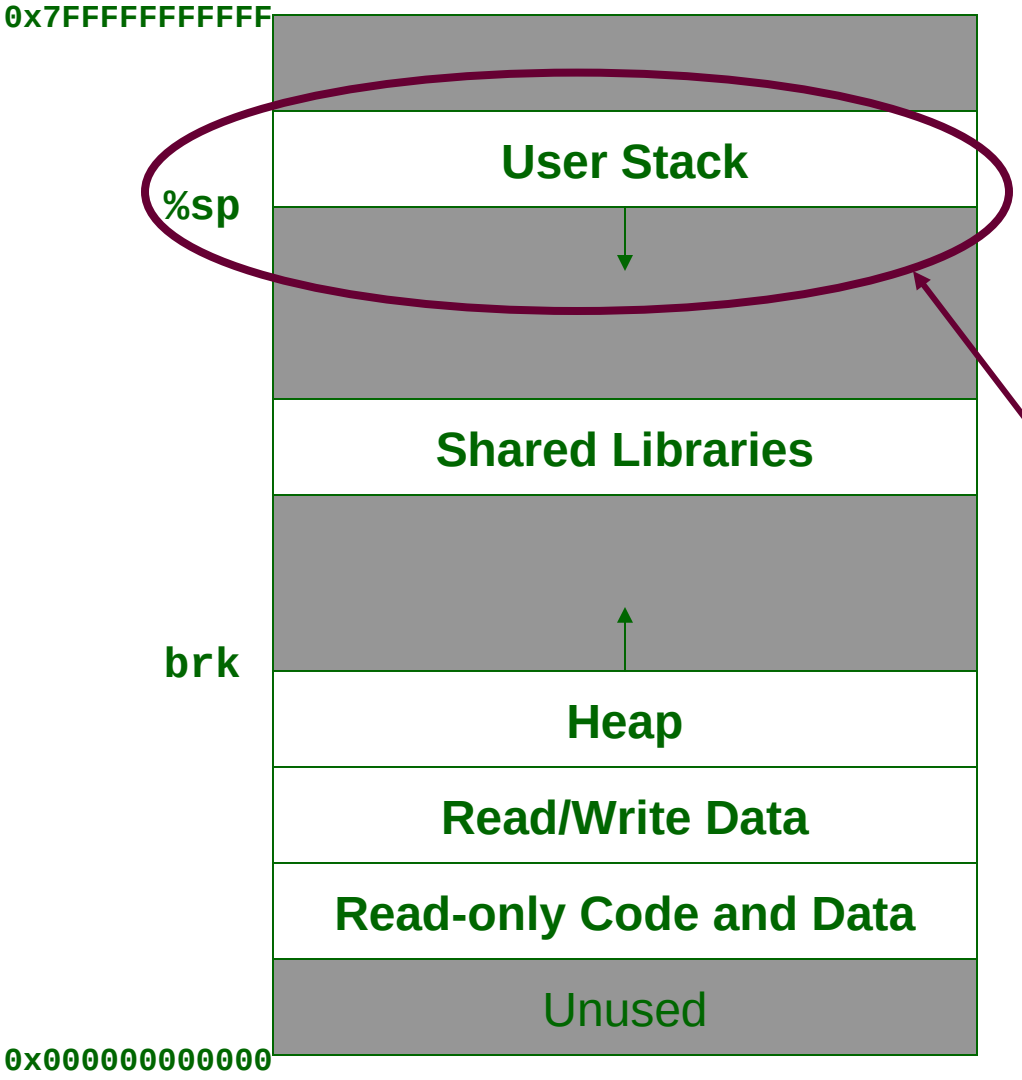
Traditional View of a Process



Process context

- Virtual memory (code/data/stack)
- Registers
- Program counter
- Stack pointer
- brk pointer
- File descriptors

Alternate View of a Process



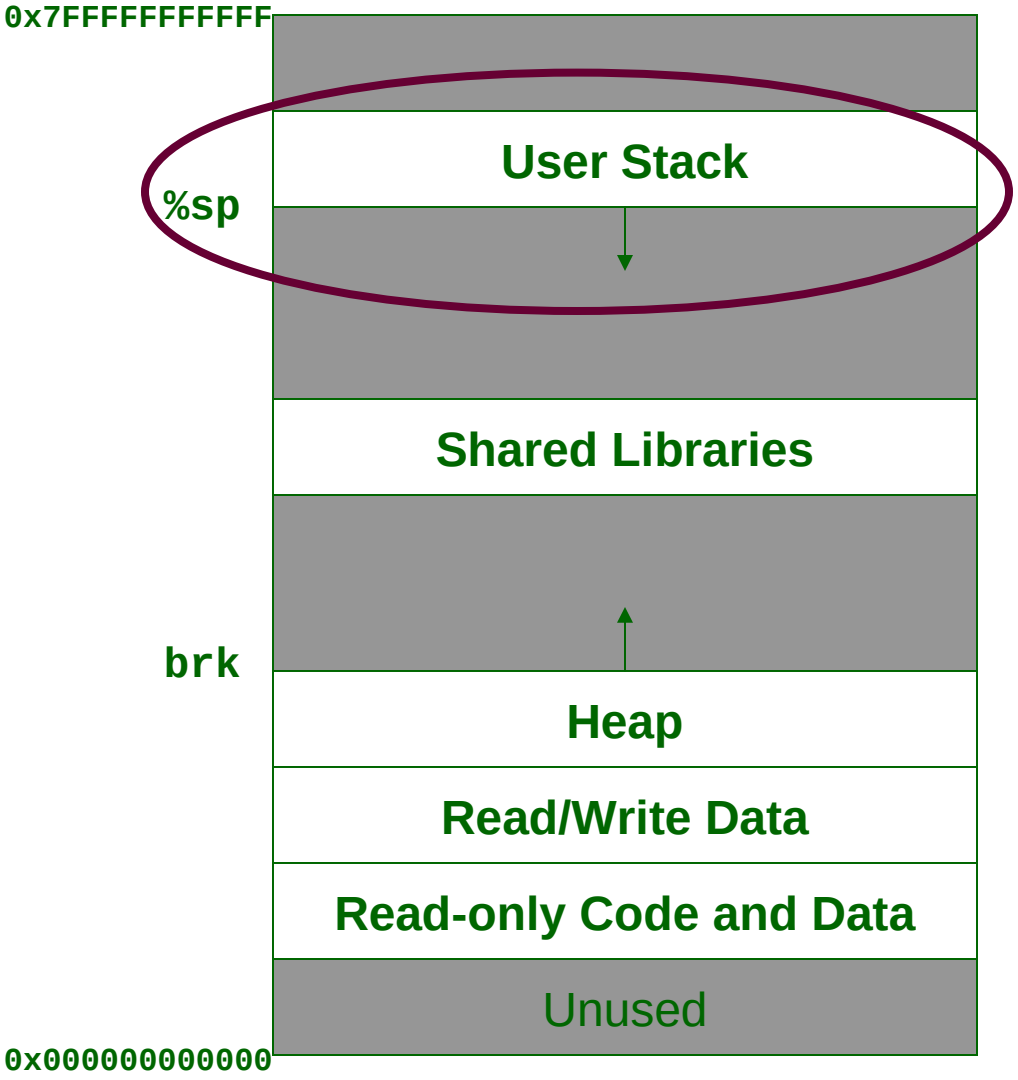
Process context

- Virtual memory (code/data)
- brk pointer
- File descriptors

Thread context

- Registers
- Program counter
- Stack
- Stack pointer

A Process with Multiple Threads



Multiple threads can be associated with a process

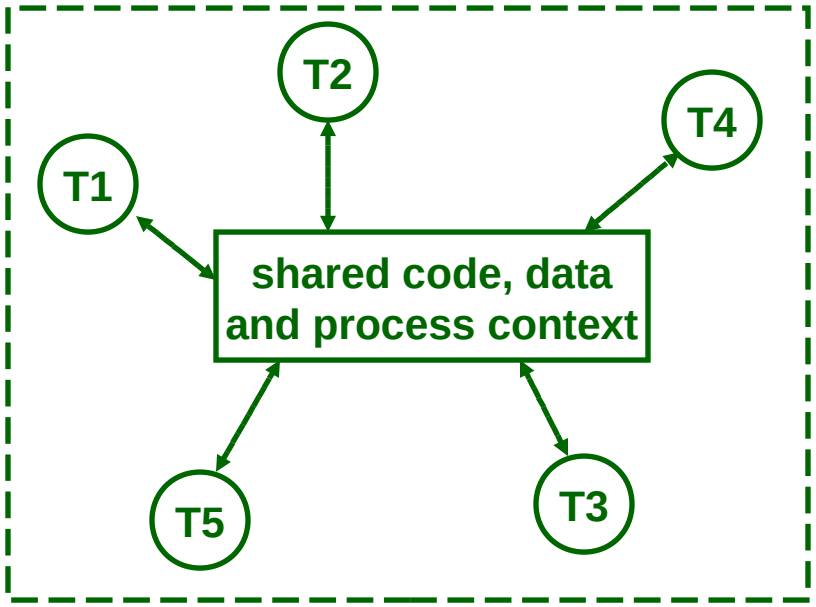
- **Each thread shares the same process context**
  - **Code**
  - **Data**
  - **brk pointer**
  - **File descriptors**
- **Each thread has its own context**
  - **Registers**
  - **Program counter**
  - **Stack**
  - **Stack pointer**
- **Each thread has its own logical control flow (sequence of PC values)**
- **Each thread has its own thread id (TID)**

Logical View of Threads

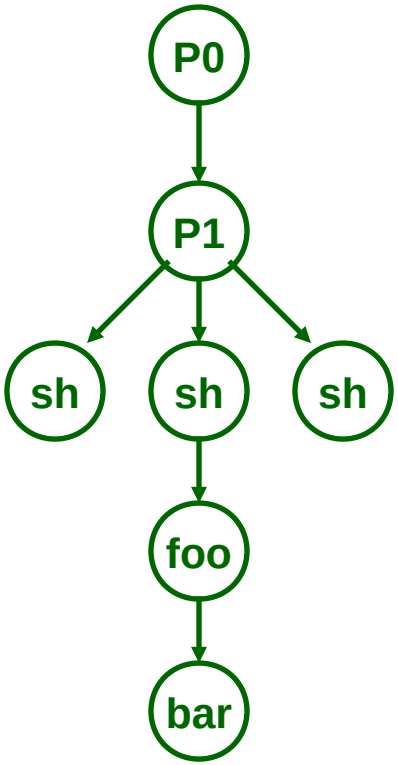
Threads associated with a process form a pool of peers

- Unlike processes which form a tree hierarchy

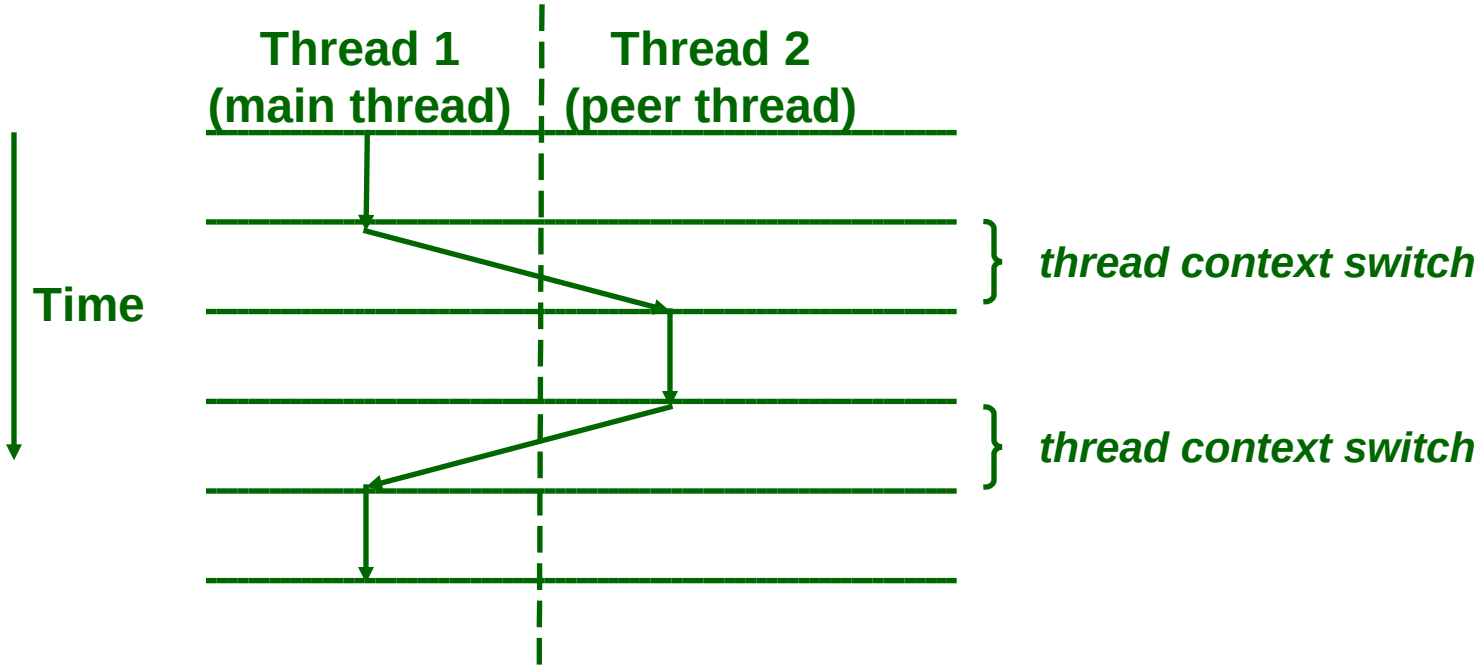
Threads associated with process foo



Process hierarchy



Thread Execution Model



Thread context switch

- **Cheaper than process context switch (less state)**
- **Main thread is simply the initial thread to run in a process**
  - **Threads are peers (no parent/child relationship)**

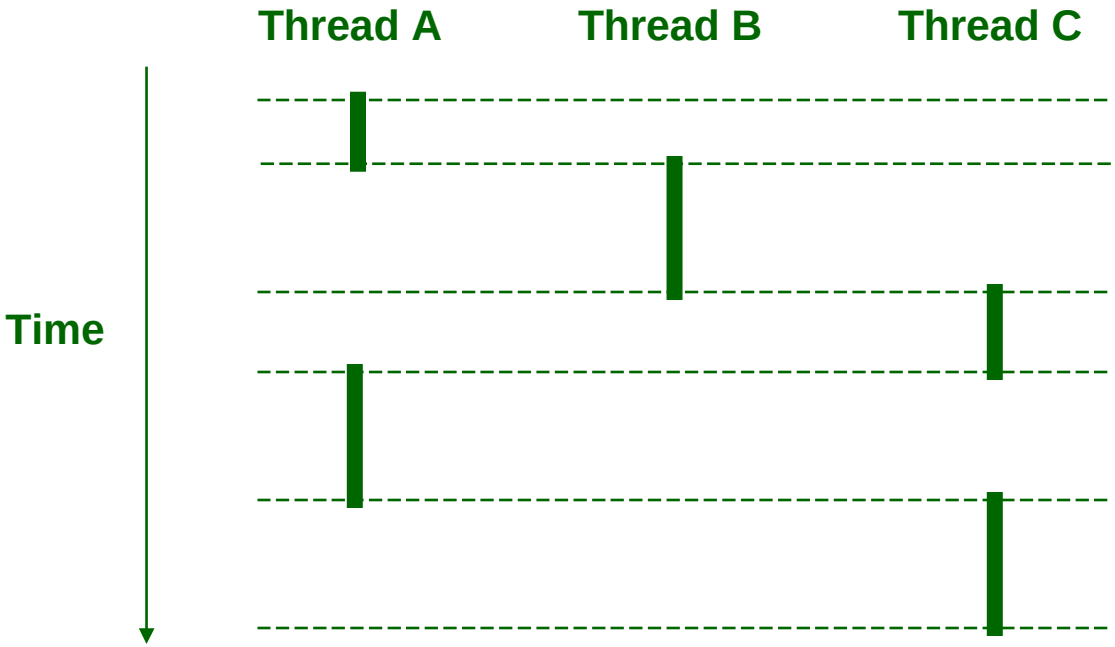
Threads can be scheduled by the kernel or by a user-level thread library (depending on OS and library)

Concurrent Thread Execution

Two threads run concurrently (are concurrent) if their logical flows overlap in time

Otherwise, they are sequential

Concurrent: A & B, A & C
Sequential: B & C



Threads vs. Processes

Similarities

- Each has its own logical control flow
- Each can run concurrently
- Each is context switched

Differences

- Threads share code and data, processes (typically) do not
- Threads are less expensive than processes
  - Process control (creating and reaping) is 2-5 times as expensive as thread control

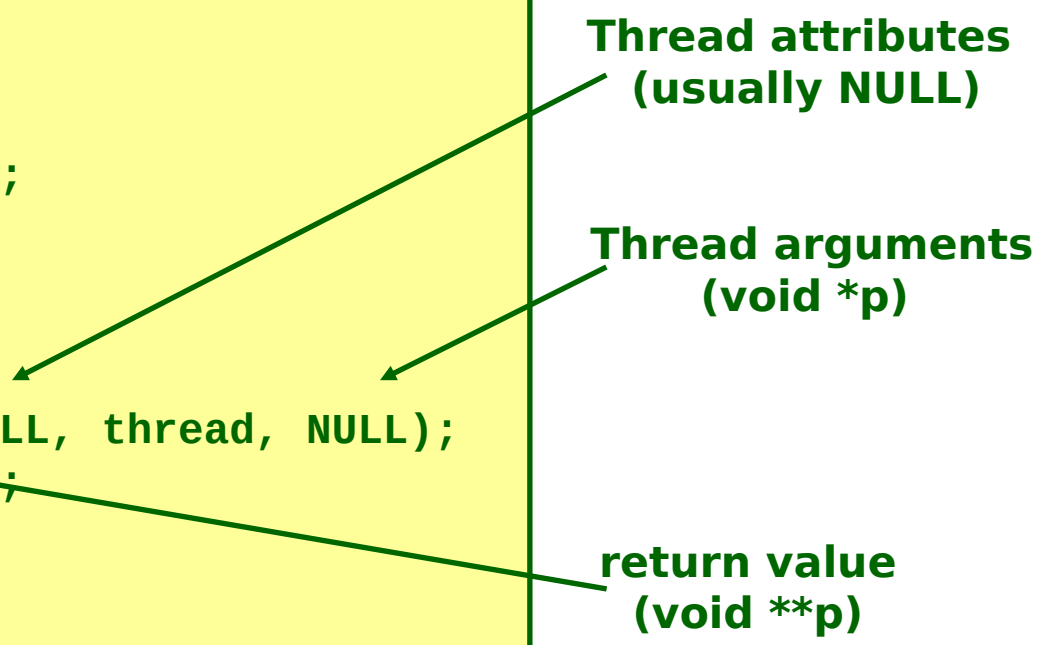
Posix Threads (Pthreads) Interface

Pthreads: Standard interface for ~60 functions that manipulate threads from C programs

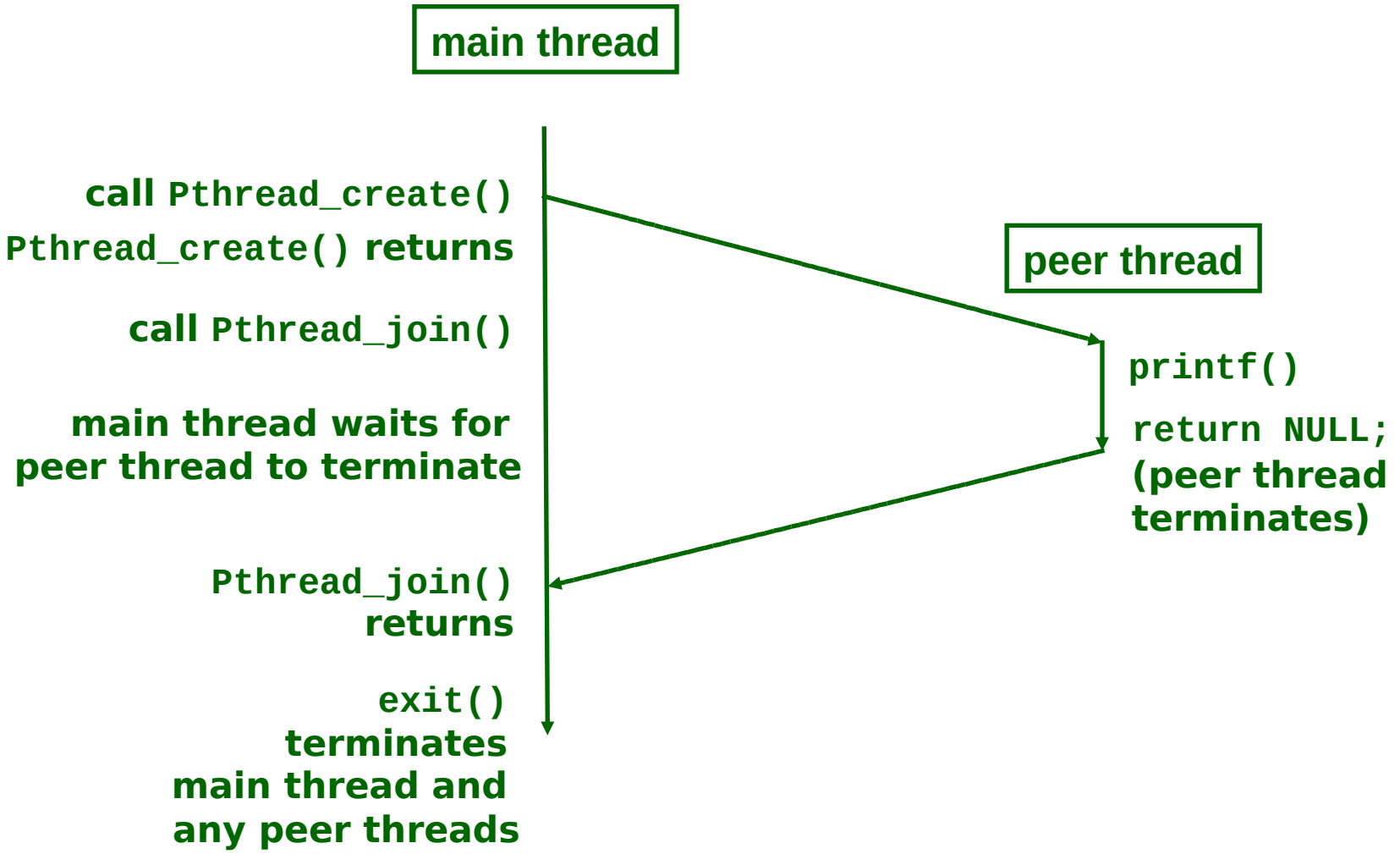
- **Creating and reaping threads**
  - `pthread_create`
  - `pthread_join`
- **Determining your thread ID**
  - `pthread_self`
- **Terminating threads**
  - `pthread_cancel`
  - `pthread_exit`
  - `exit` **[terminates all threads]**
  - `return` **[terminates current thread]**
- **Synchronizing access to shared variables**
  - `pthread_mutex_init`
  - `pthread_mutex_[un]lock`
  - `pthread_cond_init`
  - `pthread_cond_[timed]wait`

The Pthreads "hello, world" Program

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

void *thread(void *vargp);

int main(void) {
  pthread_t tid;

  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}

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



Execution of Threaded “hello, world”



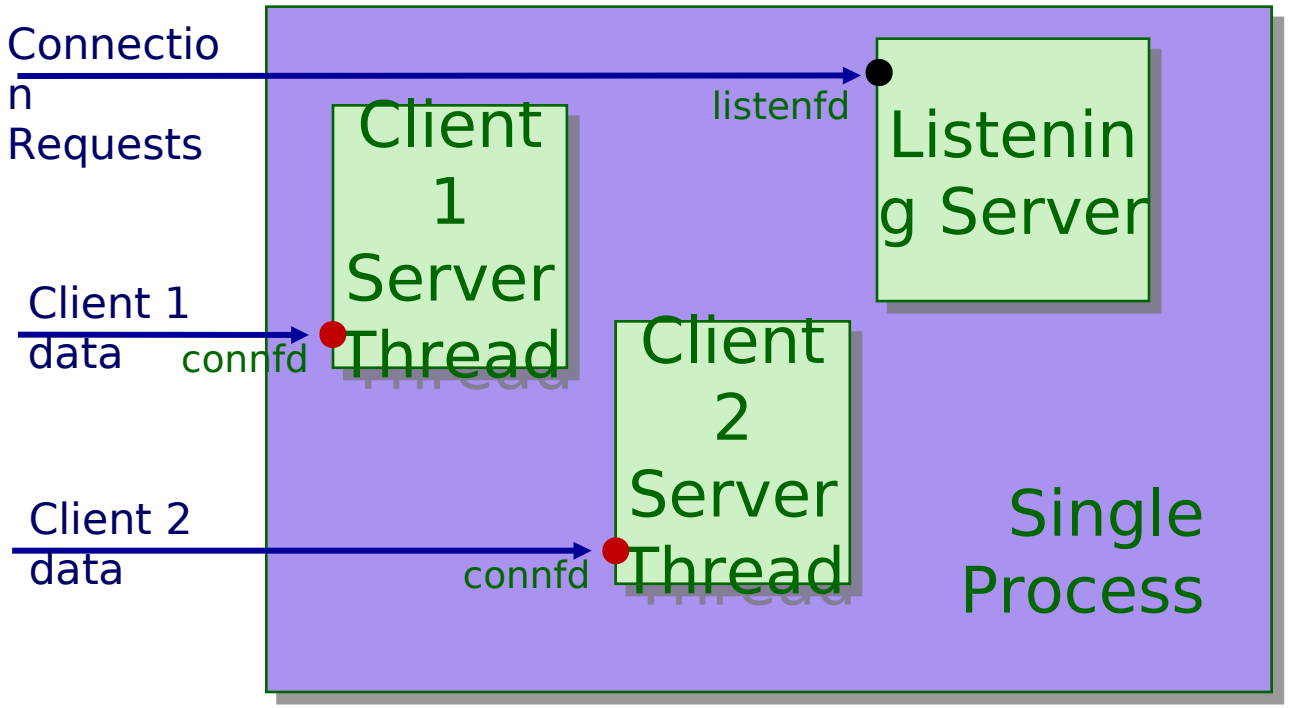
Thread-Based Concurrent Echo Server

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    pthread_t tid;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }

    listenfd = Open_listenfd(argv[1]);
    while (true) {
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

Thread-Based Concurrent Server (cont)

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

    Pthread_detach(Pthread_self());
    Free(vargp);

    echo(connfd);
    Close(connfd);
    return (NULL);
}
```

Threaded-Process Execution Model



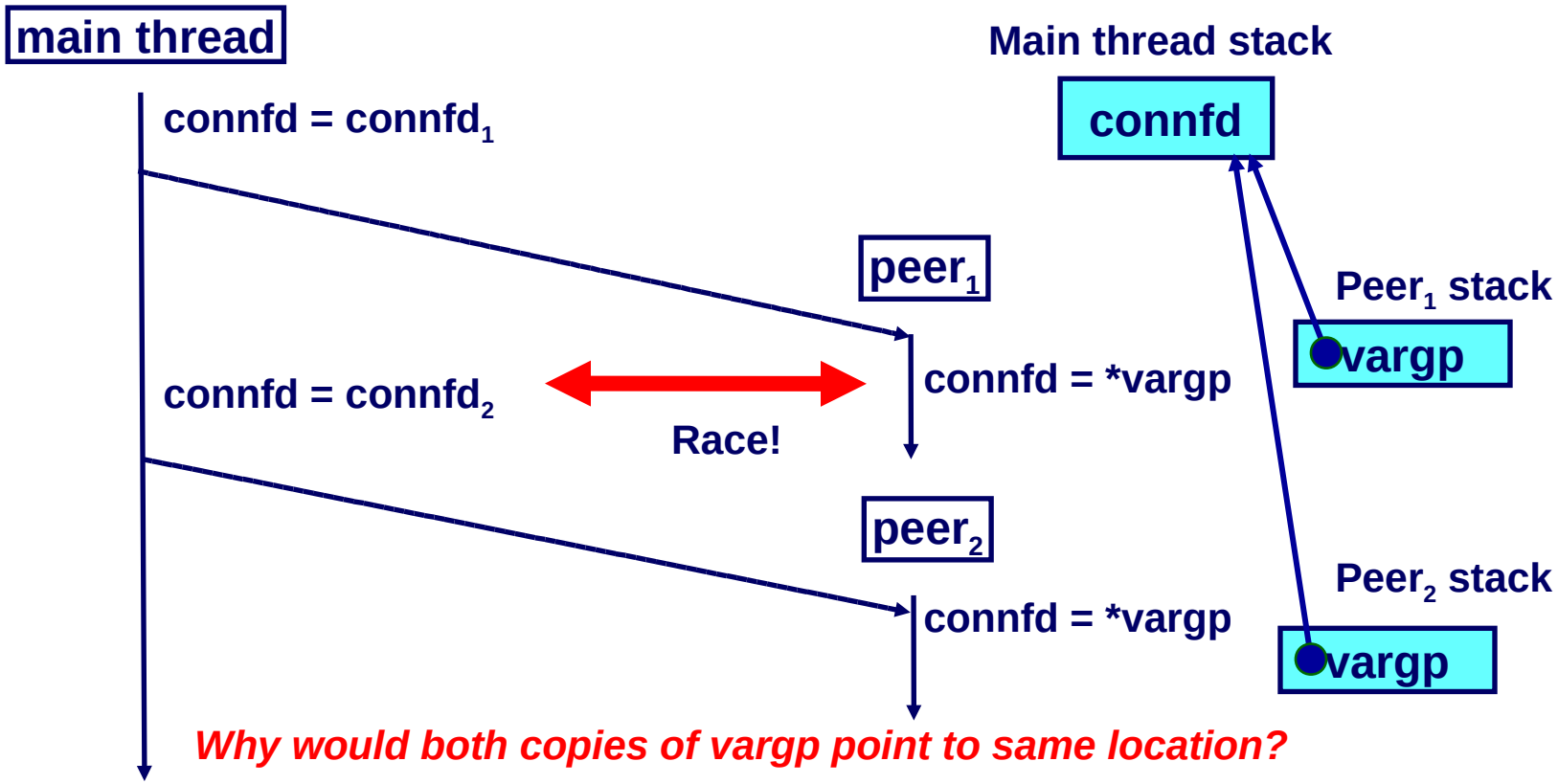
Multiple threads within single process

Some shared state between them

- **File descriptors (in this example; usually more)**

Potential Form of Unintended Sharing

```
  while (true) {
    int connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    Pthread_create(&tid, NULL, echo_thread, &connfd);
  }
}
```



Issues With Thread-Based Servers

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 (no parent, any thread can reap)**
- **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, &connfd);`

All functions called by a thread must be thread-safe

Pros/Cons of Thread-Based Designs

+ Easy to share data 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

Road map

Process-based concurrency

Thread-based concurrency

Safe data sharing using semaphore

Event-driven concurrency

Shared Variables in Threaded Programs

Question: Which variables in a threaded C program are shared variables?

- The answer is not as simple as “global variables are shared” and “stack variables are private”

Requires answers to the following questions:

- What is the memory model for threads?
- How are variables mapped to memory instances?
- How many threads reference each of these instances?

Threads Memory Model

Conceptual model:

- Each thread runs in the context of a process
- Each thread has its own separate thread context
  - Thread ID, stack, stack pointer, program counter, and general purpose registers
- All threads share the remaining process context
  - Code, data, heap, and shared library segments of the process virtual address space
  - Open files and installed signal handlers

Operationally, this model is not strictly enforced:

- Register values are truly separate and protected
- But, any thread can read and write the stack of any other thread!

Mismatch between the conceptual and operational model is a source of confusion and errors

Accessing Another Thread's Stack

```
char **ptr;  // global

int main(void) {
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp) { // thread routine
    int myid = (int)vargp;
    static int svar = 0;
    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++svar);
}
```

Peer threads access main thread's stack indirectly through global `ptr` variable

Mapping Variable to Memory Instances

```
char **ptr;  // global

int main(void) {
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };
    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp) { // thread routine
    int myid = (int) vargp;
    static int svar = 0;
    printf("[%d]: %s (svar=%d)\n", myid, ptr[myid], ++svar);
}
```

Global variable – 1 instance (ptr [data])

Local variable – 1 instance (msgs.m [mt stack])

Local variable – 2 instances (myid.p0 [pt0 stack], myid.p1 [pt1 stack])

Local variable – 1 instance (svar [data])

Shared Variable Analysis

Which variables are shared?

Variable instance	Referenced by main thread?	Referenced by peer thread 0?	Referenced by peer thread 1?
`ptr`	yes	yes	yes
`svar`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

Answer: A variable x is shared iff multiple threads reference at least one instance of x

Thus:

- `ptr`, `svar`, **and** `msgs` **are shared**
- `i` **and** `myid` **are not shared**

badcnt.c: Improper Synchronization

```
#define NITERS 100000000
unsigned int cnt = 0; // shared

int main(void) {
    pthread_t tid1, tid2;
    Pthread_create(&tid1, NULL, count, NULL);
    Pthread_create(&tid2, NULL, count, NULL);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);
    if (cnt != (unsigned)NITERS*2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}

void *count(void *arg) { // thread routine
    for (int i = 0; i < NITERS; i++)
        cnt++;
    return (NULL);
}
```

```
machine1> ./badcnt
BOOM! cnt=198841183

machine1> ./badcnt
BOOM! cnt=198261801

machine1> ./badcnt
BOOM! cnt=198269672
```

```
machine2> ./badcnt
OK cnt=200000000

machine2> ./badcnt
OK cnt=200000000

machine2> ./badcnt
OK cnt=200000000
```

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (i = 0; i < NITERS; i++)
    cnt++;
```

Corresponding assembly code

Head (H_i)

```
.L9:
        movl -4(%ebp),%eax
        cmpl $99999999,%eax
        jle .L12
        jmp .L10
```

Load cnt (L_i)
Update cnt (U_i)
Store cnt (S_i)

```
.L12:
        movl cnt,%eax       # Load
        leal 1(%eax),%edx   # Update
        movl %edx,cnt       # Store
```

Tail (T_i)

```
.L11:
        movl -4(%ebp),%eax
        leal 1(%eax),%edx
        movl %edx,-4(%ebp)
        jmp .L9
```

```
.L10:
```

Concurrent Execution

Key idea: **In general, any sequentially consistent interleaving is possible, but some give an unexpected result!**

- **I_i denotes that thread i executes instruction I**
- **$\%eax_i$ is the content of %eax in thread i's context**

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt	
1	H_1	-	-	0	
1	L_1	0	-	0	
1	U_1	1	-	0	
1	S_1	1	-	1	
2	H_2	-	-	1	
2	L_2	-	1	1	
2	U_2	-	2	1	
2	S_2	-	2	2	
2	T_2	-	2	2	
1	T_1	1	-	2	*OK*

Concurrent Execution (cont)

Another ordering: two threads increment the counter, but the result is 1 instead of 2

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1	-	-	0
1	L_1	0	-	0
1	U_1	1	-	0
2	H_2	-	-	0
2	L_2	-	0	0
1	S_1	1	-	1
1	T_1	1	-	1
2	U_2	-	1	1
2	S_2	-	1	1
2	T_2	-	1	1

Oops!

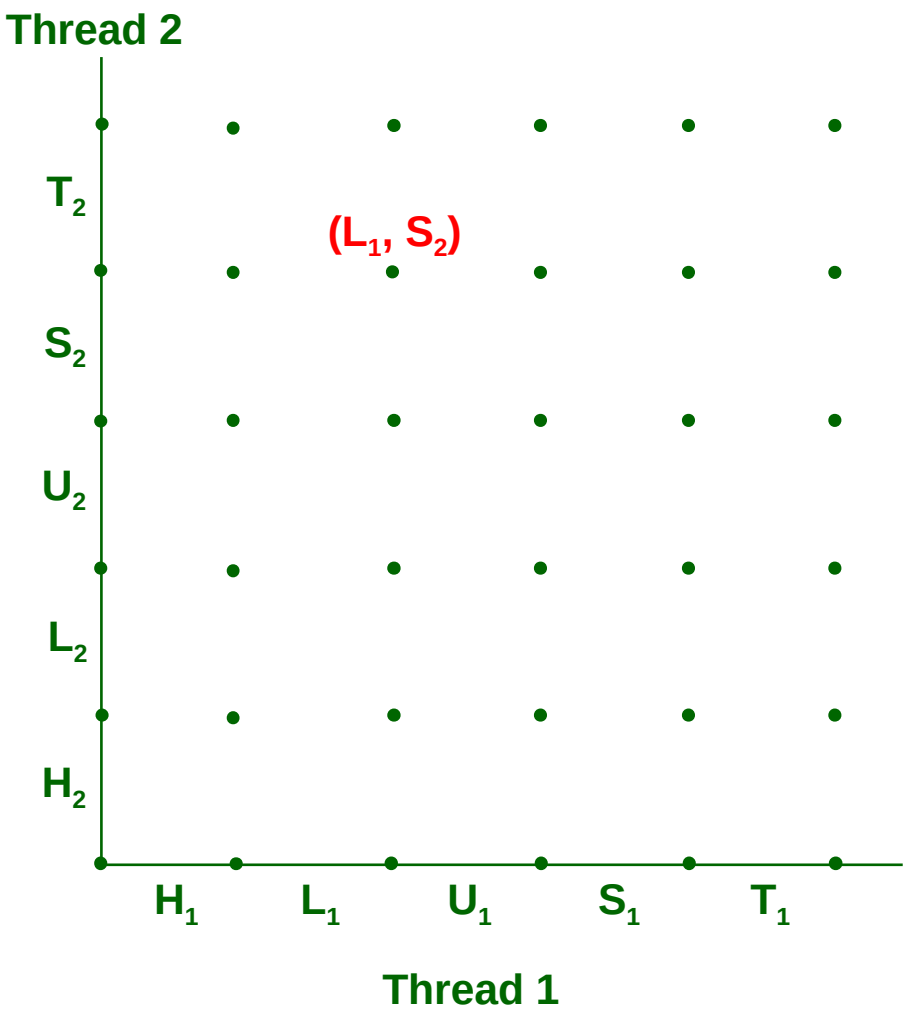
Concurrent Execution (cont)

How about this ordering?

i (thread)	$instr_i$	$\%eax_1$	$\%eax_2$	cnt
1	H_1			
1	L_1			
2	H_2			
2	L_2			
2	U_2			
2	S_2			
1	U_1			
1	S_1			
1	T_1			
2	T_2			

We can analyze the behavior using a *progress graph*

Progress Graphs



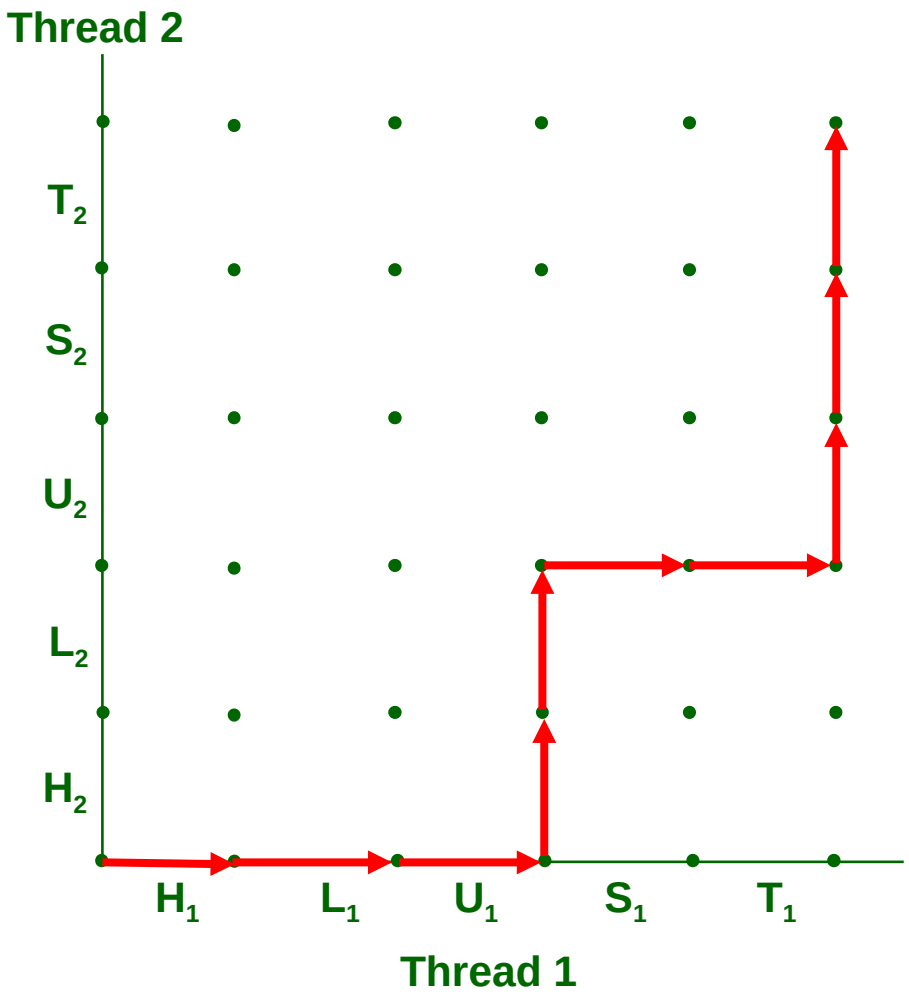
A progress graph depicts the discrete execution state space of concurrent threads

Each axis corresponds to the sequential order of instructions in a thread

Each point corresponds to a possible execution state (Inst1, Inst2)

E.g., (L1, S2) denotes state where thread 1 has completed L1 and thread 2 has completed S2

Trajectories in Progress Graphs

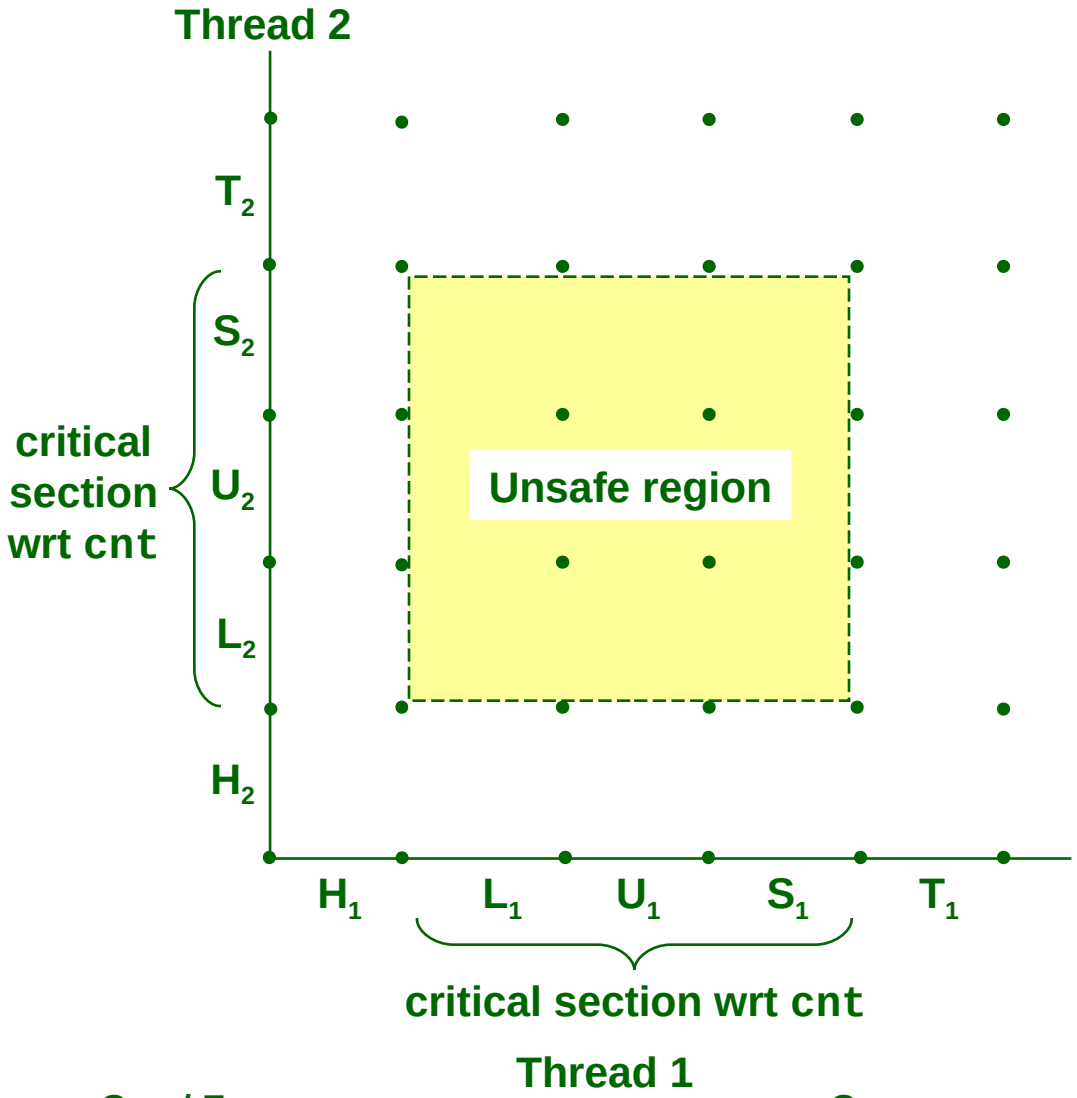


A trajectory is a sequence of legal state transitions that describes one possible concurrent execution of the threads

Example:

H1, L1, U1, H2, L2, S1, T1, U2, S2, T2

Critical Sections and Unsafe Regions

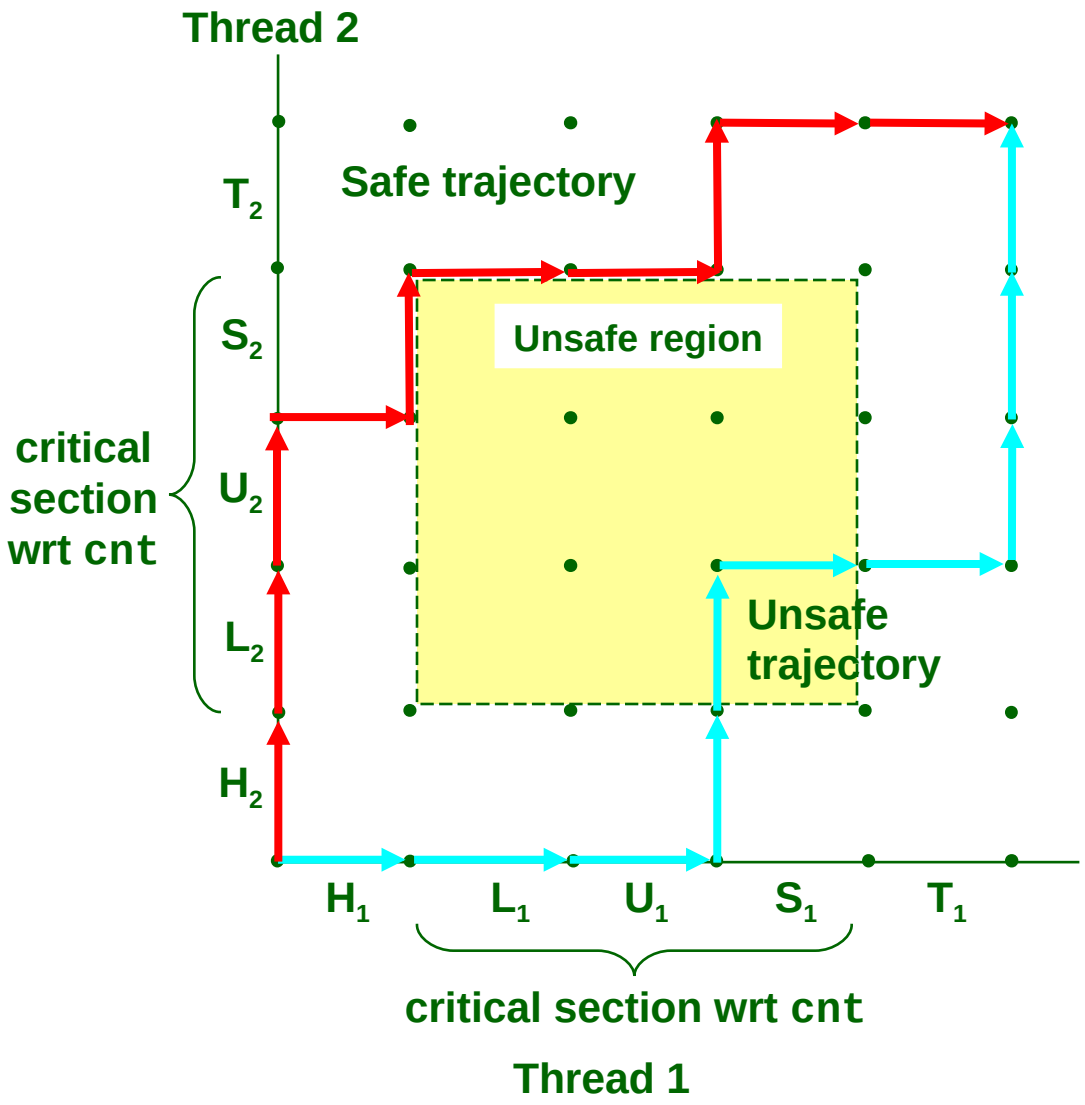


L, U, and S form a critical section with respect to the shared variable cnt

Instructions in critical sections (wrt to some shared variable) should not be interleaved

Sets of states where such interleaving occurs form unsafe regions

Safe and Unsafe Trajectories



Def: A trajectory is safe iff it doesn't touch any part of an unsafe region

Claim: A trajectory is correct (wrt `cnt`) iff it is safe

Semaphore

"A visual signaling apparatus with flags, lights, or mechanically moving arms, as one used on a railroad." - thefreedictionary.com

Question: How can we guarantee a safe trajectory?

- We must synchronize the threads so that they never enter an unsafe state

Classic solution: Dijkstra's P and V operations on semaphores.

- semaphore: non-negative integer synchronization variable.
  - P(s): [while (s == 0) wait(); s--;]
    - Dutch for "Proberen" (test)
  - V(s): [s++;]
    - Dutch for "Verhogen" (increment)
- OS guarantees that operations between brackets [ ] are executed indivisibly
  - Only one P or V operation at a time can modify s
  - When while loop in P terminates, only that P can decrement s

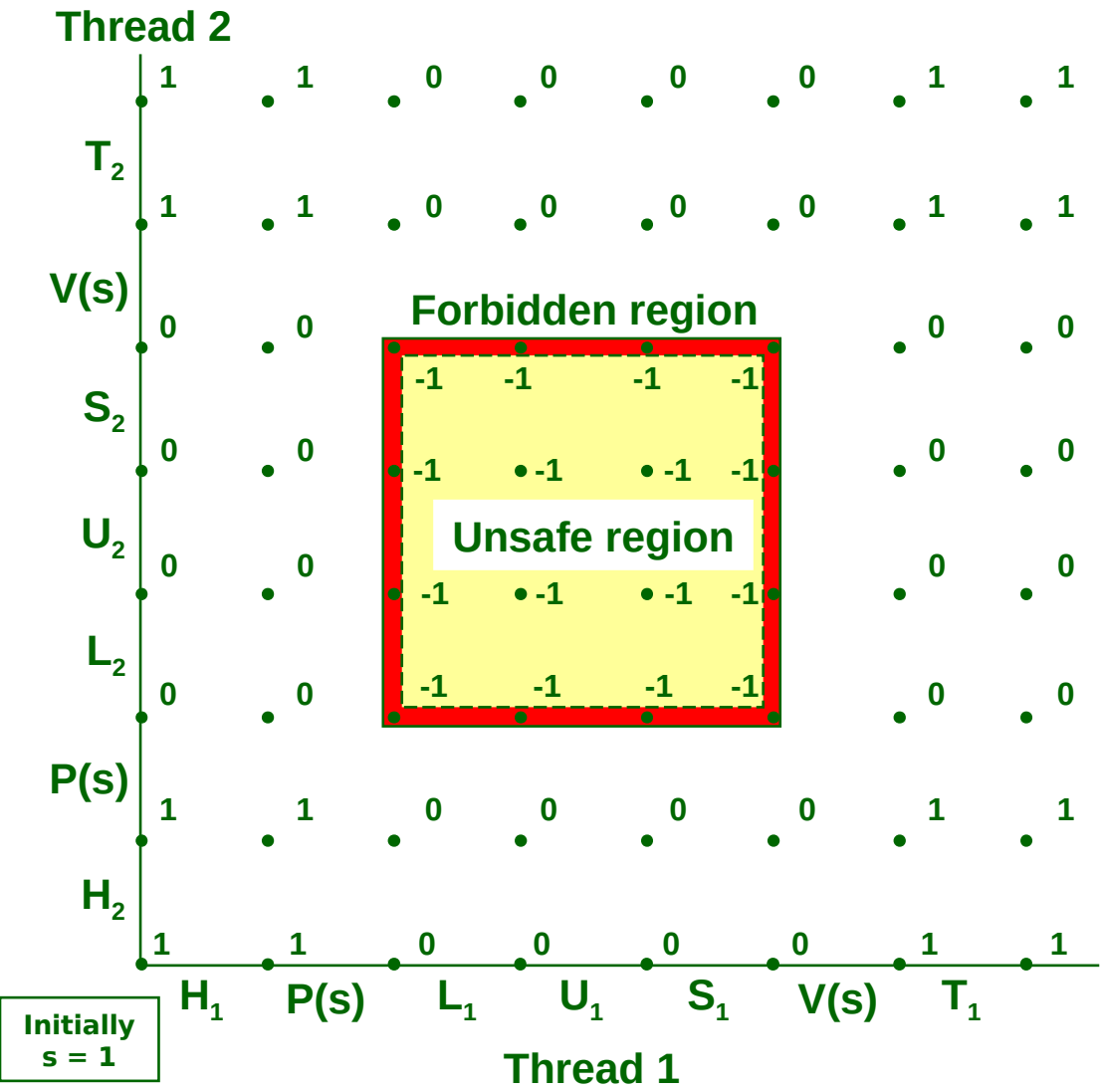
Semaphore invariant: (s >= 0)

Safe Sharing with Semaphores

Here is how we would use P and V operations to synchronize the threads that update `cnt`

```
// Semaphore “s” is initially 1.

// Thread routine
void *count(void *arg)
{
    for (int i = 0; i < NITERS; i++) {
        P(s);
        cnt++;
        V(s);
    }
    return (NULL);
}
```

Safe Sharing With Semaphores



Provide mutually exclusive access to shared variable by surrounding critical section with P and V operations on semaphore s (initially set to 1)

Semaphore invariant creates a forbidden region that encloses unsafe region and is never touched by any trajectory

POSIX Semaphores

```
/* Initialize semaphore sem to value */
/* pshared=0 if thread, pshared=1 if process (shared memory) */
void Sem_init(sem_t *sem, int pshared, unsigned int value) {
  if (sem_init(sem, pshared, value) < 0)
    unix_error("Sem_init");
}

/* P operation on semaphore sem */
void P(sem_t *sem) {
  if (sem_wait(sem))
    unix_error("P");
}

/* V operation on semaphore sem */
void V(sem_t *sem) {
  if (sem_post(sem))
    unix_error("V");
}
```

goodcnt.c Sharing With POSIX Semaphores

```
unsigned int cnt; // counter
sem_t sem;        // semaphore

int main(void) {
    Sem_init(&sem, 0, 1); // sem=1
    // Create 2 threads and wait; code omitted.
    if (cnt != (unsigned)NITERS * 2)
        printf("BOOM! cnt=%d\n", cnt);
    else
        printf("OK cnt=%d\n", cnt);
}

void *count(void *arg) { // thread routine
    for (int i = 0; i < NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return (NULL);
}
```

Crucial concept: Thread Safety

Functions called from a thread (without external synchronization) must be *thread-safe*

- **Meaning: it must always produce correct results when called repeatedly from multiple concurrent threads**

Some examples of thread-unsafe functions:

- **Failing to protect shared variables**
- **Relying on persistent state across invocations**
- **Returning a pointer to a static variable**
- **Calling thread-unsafe functions**

Thread-Unsafe Functions (Class 1)

Failing to protect shared variables

- **Fix: Use P and V semaphore operations**
- **Example: `goodcnt.c`**
- **Issue: Synchronization operations will slow down code**
  - **e.g., `badcnt` requires 0.5s, `goodcnt` requires 7.9s**

Thread-Unsafe Functions (Class 2)

Relying on persistent state across multiple function invocations

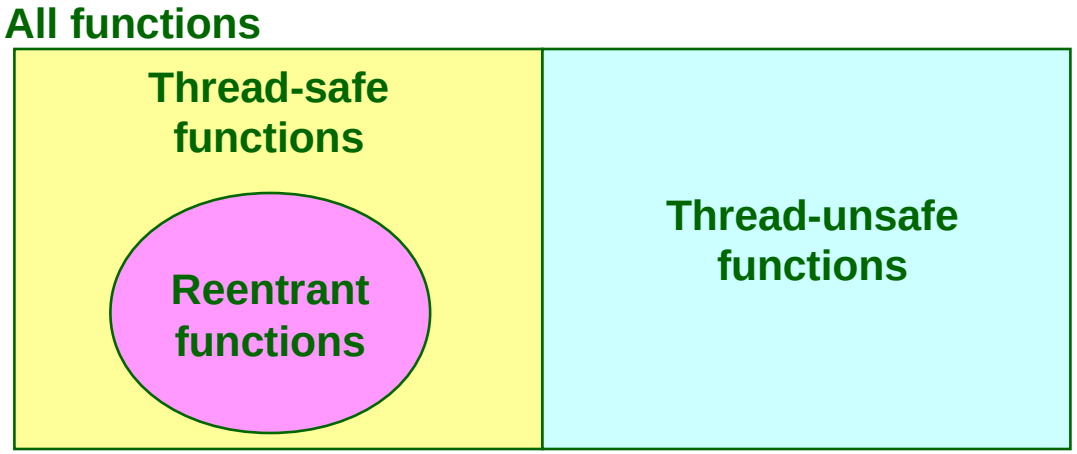
- **Example: Random number generator (RNG) that relies on static state**

```
/* rand: return pseudo-random integer on 0..32767 */
static unsigned int next = 1;
int rand(void)
{
    next = next*1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}

/* srand: set seed for rand() */
void srand(unsigned int seed)
{
    next = seed;
}
```

Reentrant Functions

A function is reentrant iff it accesses no shared variables when called from multiple threads

- **Reentrant functions are a proper subset of the set of thread-safe functions**



Making Thread-Safe RNG

Pass state as part of argument

- and, thereby, eliminate static state

```
/* rand - return pseudo-random integer on 0..32767 */

int rand_r(int *nextp)
{
    *nextp = *nextp*1103515245 + 12345;
    return (unsigned int)(*nextp/65536) % 32768;
}
```

Consequence: programmer using rand must maintain seed

Thread-Unsafe Functions (Class 3)

Returning a ptr to a `static` variable

```
struct hostent
*gethostbyname(char name)
{
  static struct hostent h;
  <contact DNS and fill in h>
  return &h;
}
```

Fixes:

- **1. Rewrite code so caller passes pointer to `struct`**
  - **Issue: Requires changes in caller and callee**
- **2. *Lock-and-copy***
  - **Issue: Requires only simple changes in caller (and none in callee)**
  - **However, caller must free memory**

```
hostp = Malloc(...);
gethostbyname_r(name, hostp);
```

```
struct hostent
*gethostbyname_ts(char *name)
{
  struct hostent *q = Malloc(...);
  struct hostent *p;
  P(&mutex); /* lock */
  p = gethostbyname(name);
  *q = *p;   /* copy */
  V(&mutex);
  return q;
}
```

Thread-Unsafe Functions (Class 4)

Calling thread-unsafe functions

- **Calling one thread-unsafe function makes the entire function that calls it thread-unsafe**

- **Fix: Modify the function so it calls only thread-safe functions ☺**

Thread-Safe Library Functions

All functions in the Standard C Library (at the back of your K&R text) are thread-safe

- **Examples:** `malloc`, `free`, `printf`, `scanf`

Most Unix system calls are thread-safe, with a few exceptions:

Thread-unsafe function		Reentrant version
`asctime`	3	`asctime_r`
`ctime`	3	`ctime_r`
`gethostbyaddr`	3	`gethostbyaddr_r`
`gethostbyname`	3	`gethostbyname_r`
`inet_ntoa`	3	(none)
`localtime`	3	`localtime_r`
`rand`	2	`rand_r`

One worry: races

A *race* occurs when correctness of the program depends on one thread reaching point x before another thread reaches point y

```
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return (NULL);
}
```

Race Elimination

Make sure don't have unintended sharing of state

```
/* a threaded program with a race */
int main() {
    pthread_t tid[N];
    int i;
    for (i = 0; i < N; i++) {
        int *valp = Malloc(sizeof(int));
        *valp = i;
        Pthread_create(&tid[i], NULL, thread, valp);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}
/* thread routine */
void *thread(void *vargp) {
    int myid = *(int *)vargp;
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return (NULL);
}
```

Another worry: Deadlock

Processes wait for condition that will never be true

Typical Scenario

- **Processes 1 and 2 needs two resources (A and B) to proceed**
- **Process 1 acquires A, waits for B**
- **Process 2 acquires B, waits for A**
- **Both will wait forever!**

Deadlocking With POSIX Semaphores

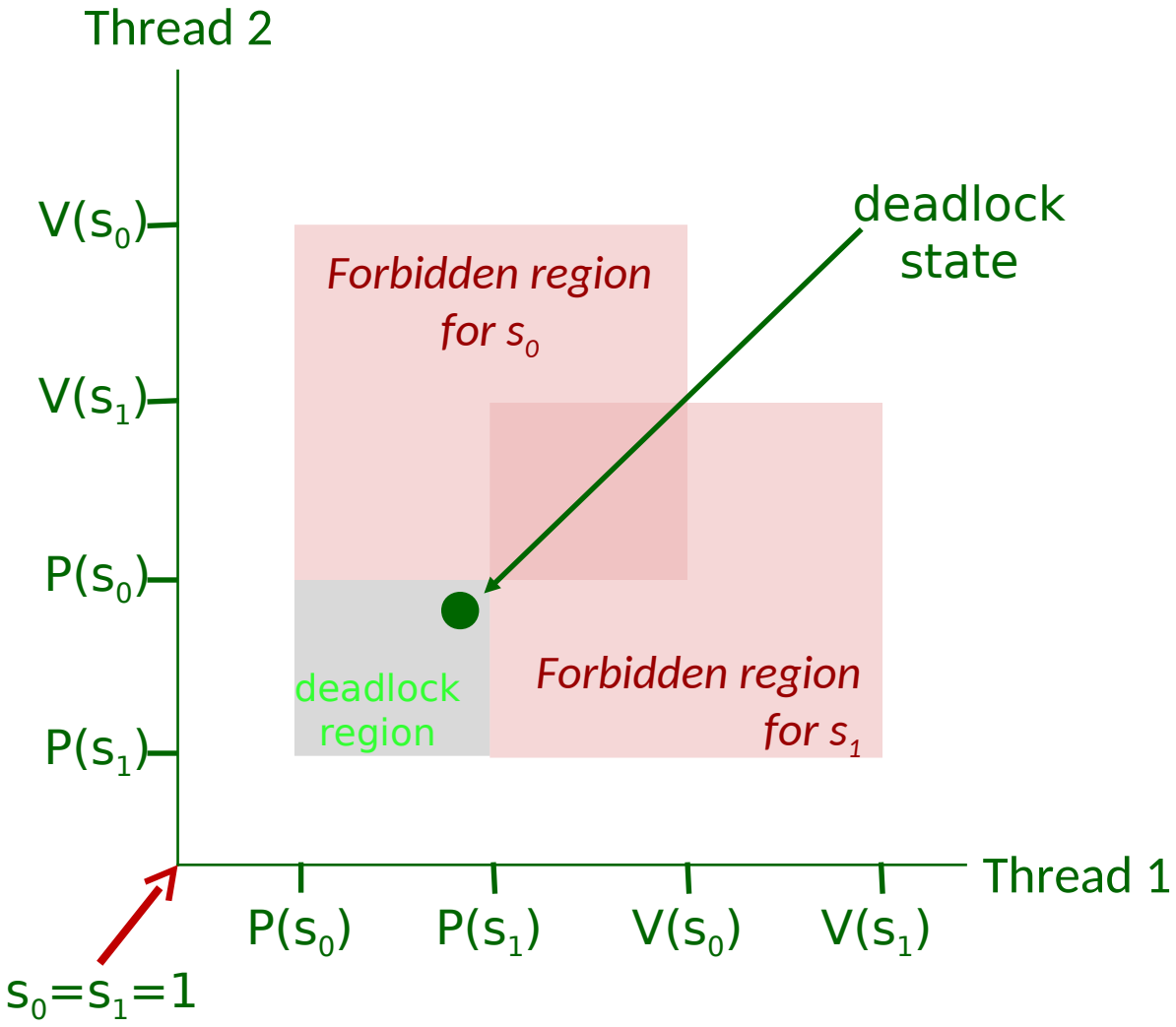
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[id]); P(&mutex[1-id]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
P(s_0);
P(s_1);
cnt++;
V(s_0);
V(s_1);

Tid[1]:
P(s_1);
P(s_0);
cnt++;
V(s_1);
V(s_0);

Deadlock Visualized in Progress Graph



Locking introduces the potential for *deadlock:* waiting for a condition that will never be true

Any trajectory that enters the *deadlock region* will eventually reach the *deadlock state*, waiting for either s_0 or s_1 to become nonzero

Other trajectories luck out and skirt the deadlock region

Unfortunate fact: deadlock is often non-deterministic

Avoiding Deadlock

Acquire shared resources in same order

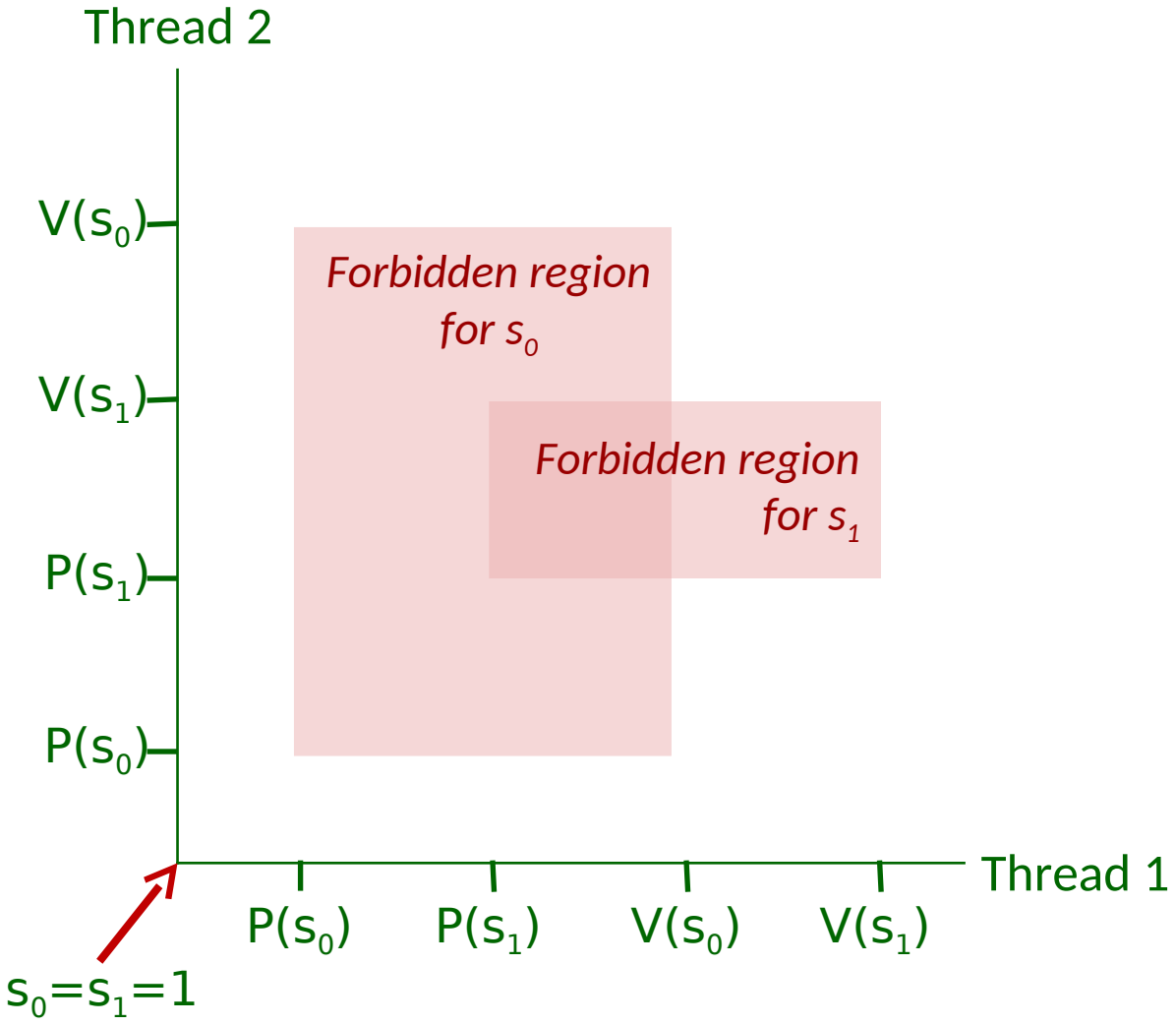
```
int main()
{
    pthread_t tid[2];
    Sem_init(&mutex[0], 0, 1);  /* mutex[0] = 1 */
    Sem_init(&mutex[1], 0, 1);  /* mutex[1] = 1 */
    Pthread_create(&tid[0], NULL, count, (void*) 0);
    Pthread_create(&tid[1], NULL, count, (void*) 1);
    Pthread_join(tid[0], NULL);
    Pthread_join(tid[1], NULL);
    printf("cnt=%d\n", cnt);
    exit(0);
}
```

```
void *count(void *vargp)
{
    int i;
    int id = (int) vargp;
    for (i = 0; i < NITERS; i++) {
        P(&mutex[0]); P(&mutex[1]);
        cnt++;
        V(&mutex[id]); V(&mutex[1-id]);
    }
    return NULL;
}
```

Tid[0]:
P(s_0);
P(s_1);
cnt++;
V(s_0);
V(s_1);

Tid[1]:
P(s_0);
P(s_1);
cnt++;
V(s_1);
V(s_0);

Avoided Deadlock in Progress Graph

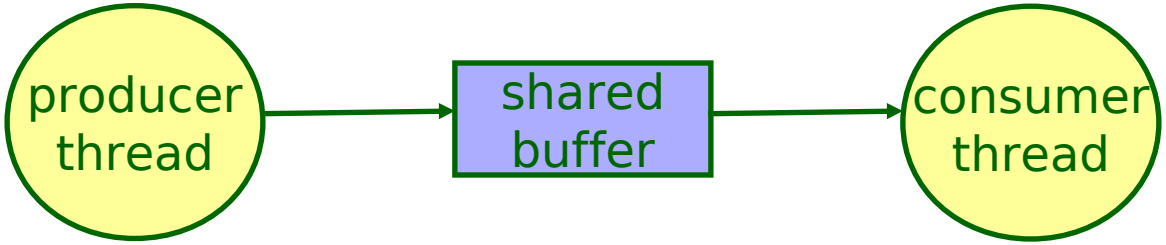


No way for trajectory to get stuck

Processes acquire locks in same order

Order in which locks released immaterial

Notifying With Semaphores



Common synchronization pattern:

- **Producer waits for slot, inserts item in buffer, and notifies consumer**
- **Consumer waits for item, removes it from buffer, and notifies producer**

Examples

- **Multimedia processing:**
  - **Producer creates MPEG video frames, consumer renders them**
- **Event-driven graphical user interfaces**
  - **Producer detects mouse clicks, mouse movements, and keyboard hits and inserts corresponding events in buffer**
  - **Consumer retrieves events from buffer and paints the display**

Producer-Consumer on a Buffer That Holds One Item

```
/* buf1.c - producer-consumer
on 1-element buffer */
#include “csapp.h”

#define NITERS 5

void *producer(void *arg);
void *consumer(void *arg);

struct {
  int buf; /* shared var */
  sem_t full; /* sems */
  sem_t empty;
} shared;
```

```
int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* initialize the semaphores */
  Sem_init(&shared.empty, 0, 1);
  Sem_init(&shared.full,  0, 0);

  /* create threads and wait */
  Pthread_create(&tid_producer, NULL,
                 producer, NULL);
  Pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  exit(0);
}
```

Producer-Consumer (cont)

Initially: empty = 1, full = 0

```
/* producer thread */
void *producer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* produce item */
    item = i;
    printf("produced %d\n",
            item);

    /* write item to buf */
    P(&shared.empty);
    shared.buf = item;
    V(&shared.full);
  }
  return (NULL);
}
```

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* read item from buf */
    P(&shared.full);
    item = shared.buf;
    V(&shared.empty);

    /* consume item */
    printf("consumed %d\n", item);
  }
  return (NULL);
}
```

Counting with Semaphores

Remember, it's a non-negative integer

- So, values greater than 1 are legal

```
/* thing_5 and thing_3 */
#include "csapp.h"

sem_t five;
sem_t three;

void *five_times(void *arg);
void *three_times(void *arg);
```

```
int main() {
  pthread_t tid_five, tid_three;

  /* initialize the semaphores */
  Sem_init(&five, 0, 5);
  Sem_init(&three,  0, 3);

  /* create threads and wait */
  Pthread_create(&tid_five, NULL,
                 five_times, NULL);
  Pthread_create(&tid_three, NULL,
                 three_times, NULL);
  .
  .
  .
}
```

Counting with semaphores (cont)

Initially: five = 5, three = 3

```
/* thing_5() thread */
void *five_times(void *arg) {
  int i;

  while (1) {
    for (i=0; i<5; i++) {
      /* wait & thing_5() */
      P(&five);
      thing_5();
    }
    V(&three);
    V(&three);
    V(&three);
  }
  return NULL;
}
```

```
/* thing_3() thread */
void *three_times(void *arg) {
  int i;

  while (1) {
    for (i=0; i<3; i++) {
      /* wait & thing_3() */
      P(&three);
      thing_3();
    }
    V(&five);
    V(&five);
    V(&five);
    V(&five);
    V(&five);
  }
  return NULL;
}
```

Producer-Consumer on a Buffer That Holds More than One Item

```
/* buf1.c - producer-consumer
on 1-element buffer */
#include “csapp.h”

#define NITERS 5
#define NITEMS 7

void *producer(void *arg);
void *consumer(void *arg);

struct {
  void *buf[NITEMS];
  int cnt;
  sem_t full; /* sems */
  sem_t empty;
  sem_t mutex;
} shared;
```

```
int main() {
  pthread_t tid_producer;
  pthread_t tid_consumer;

  /* initialization */
  Sem_init(&shared.empty, 0, NITEMS);
  Sem_init(&shared.full,  0, 0);
  Sem_init(&shared.mutex,  0, 1);
  shared.cnt = 0;

  /* create threads and wait */
  Pthread_create(&tid_producer, NULL,
                 producer, NULL);
  Pthread_create(&tid_consumer, NULL,
                 consumer, NULL);
  Pthread_join(tid_producer, NULL);
  Pthread_join(tid_consumer, NULL);

  exit(0);
}
```

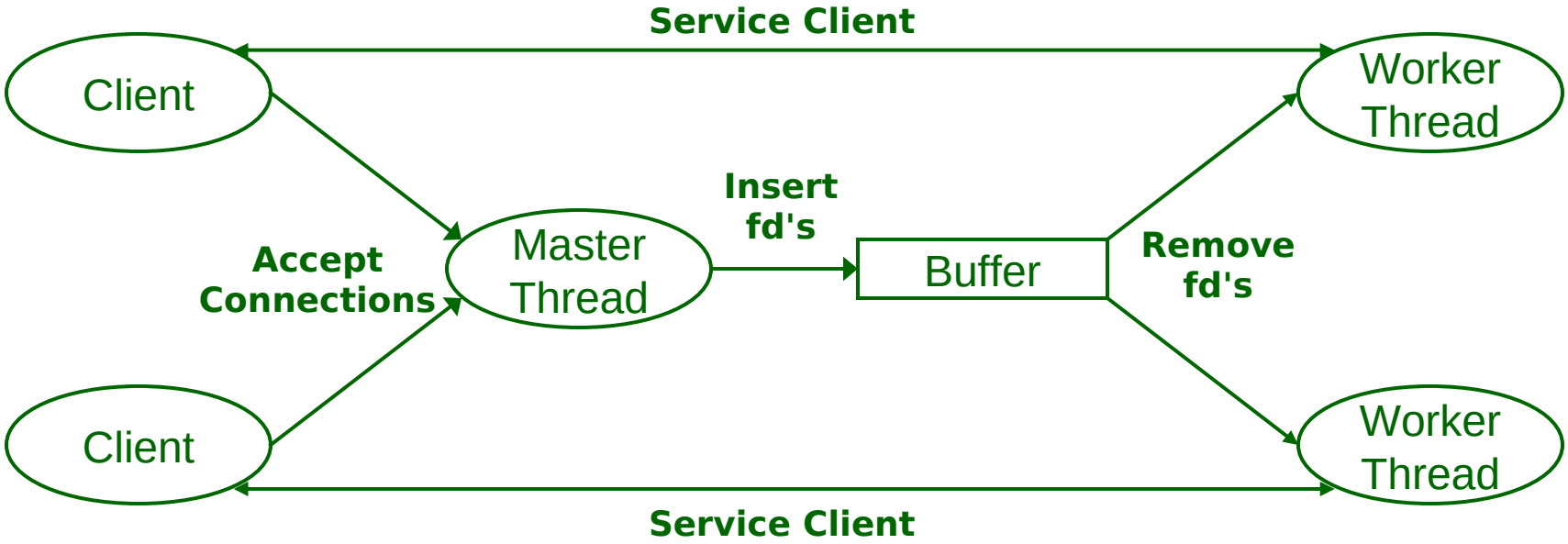
Producer-Consumer (cont)

Initially: empty = all slot, full = no slots, e.g. 0

```
/* producer thread */
void *producer(void *arg) {
  int i;

  for (i=0; i<NITERS; i++) {
    /* write item to buf */
    P(&shared.empty);
    P(&shared.mutex);

    shared.buf[shared.cnt++] =
        produceItem();

    V(&shared.mutex);
    V(&shared.full);
  }
  return NULL;
}
```

```
/* consumer thread */
void *consumer(void *arg) {
  int i, item;

  for (i=0; i<NITERS; i++) {
    /* read item from buf */
    P(&shared.full);
    P(&shared.mutex);

    item=shared.buf[shared.cnt--];

    V(&shared.mutex);
    V(&shared.empty);

    /* consume item */
    printf("consumed %d\n“, item);
  }
  return NULL;
}
```

Prethreaded Concurrent Servers



Expensive to create/reap threads

- **Create a pool of threads in advance**

What if there are not enough threads?

- **Buffer connected file descriptors**

Must synchronize transfer of file descriptors

Threads Summary

Threads provide another mechanism for writing concurrent programs

Threads are growing in popularity

- **Somewhat cheaper than processes**
- **Easy to share data between threads**

However, the ease of sharing has a cost

- **Easy to introduce subtle synchronization errors**
- **Tread carefully with threads!**

For more info:

- **D. Butenhof, “Programming with Posix Threads”, Addison-Wesley, 1997.**

Beware of Optimizing Compilers!

Code From Book

```
#define NITERS 100000000

/* shared counter variable */
unsigned int cnt = 0;

/* thread routine */
void *count(void *arg)
{
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}
```

- Global variable `cnt` shared between threads
- Multiple threads could be trying to update within their iterations

Generated Code

```
movl    cnt, %ecx
movl    $99999999, %eax
.L6:
   leal  1(%ecx), %edx
   decl  %eax
   movl  %edx, %ecx
   jns   .L6
   movl  %edx, cnt
```

- Compiler moved access to `cnt` out of loop
- Only shared accesses to `cnt` occur before loop (read) or after (write)
- What are possible program outcomes?

Controlling Optimizing Compilers!

Revised Book Code

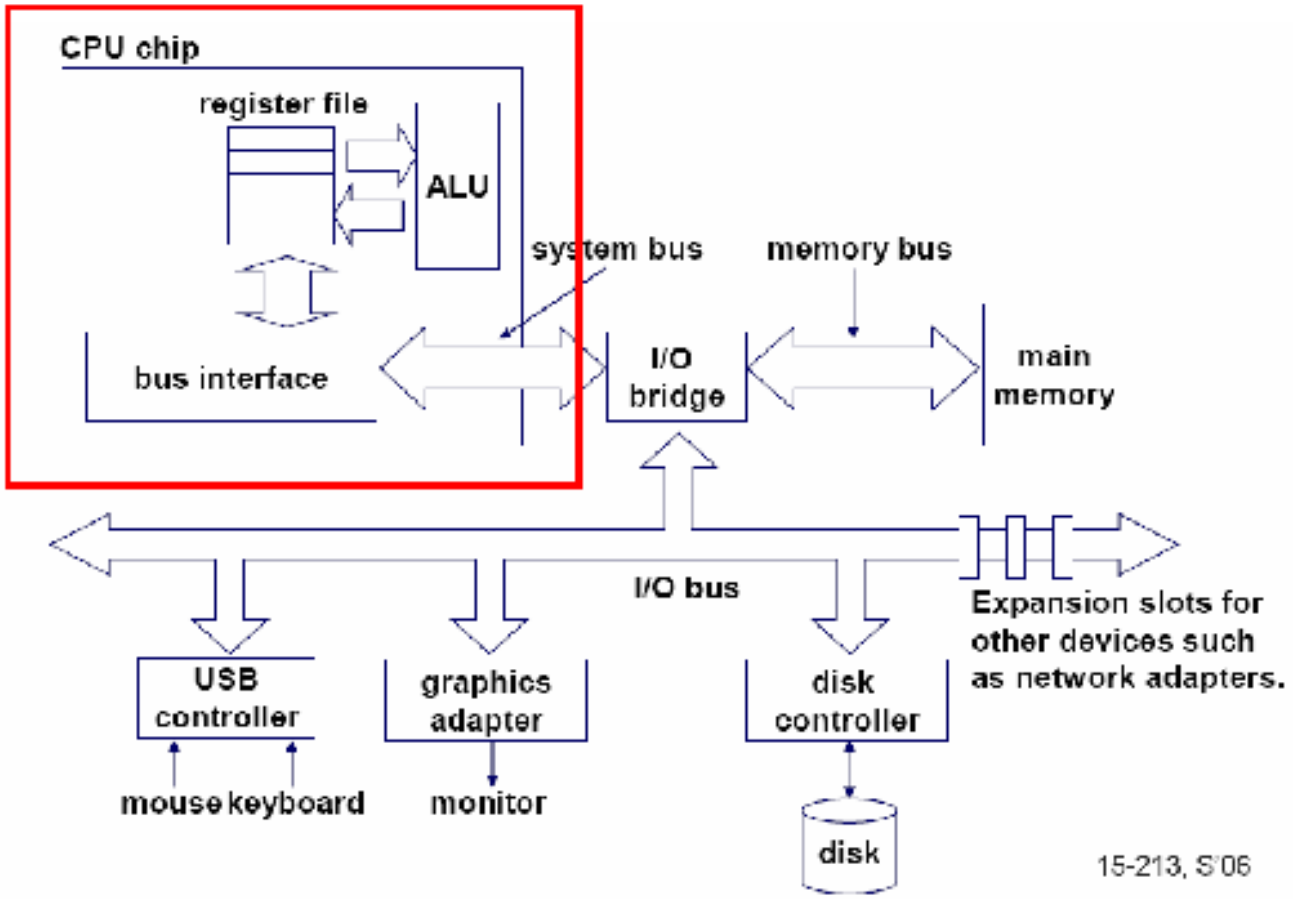
```
#define NITERS 100000000

/* shared counter variable */
volatile unsigned int cnt = 0;

/* thread routine */
void *count(void *arg)
{
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}
```

Generated Code

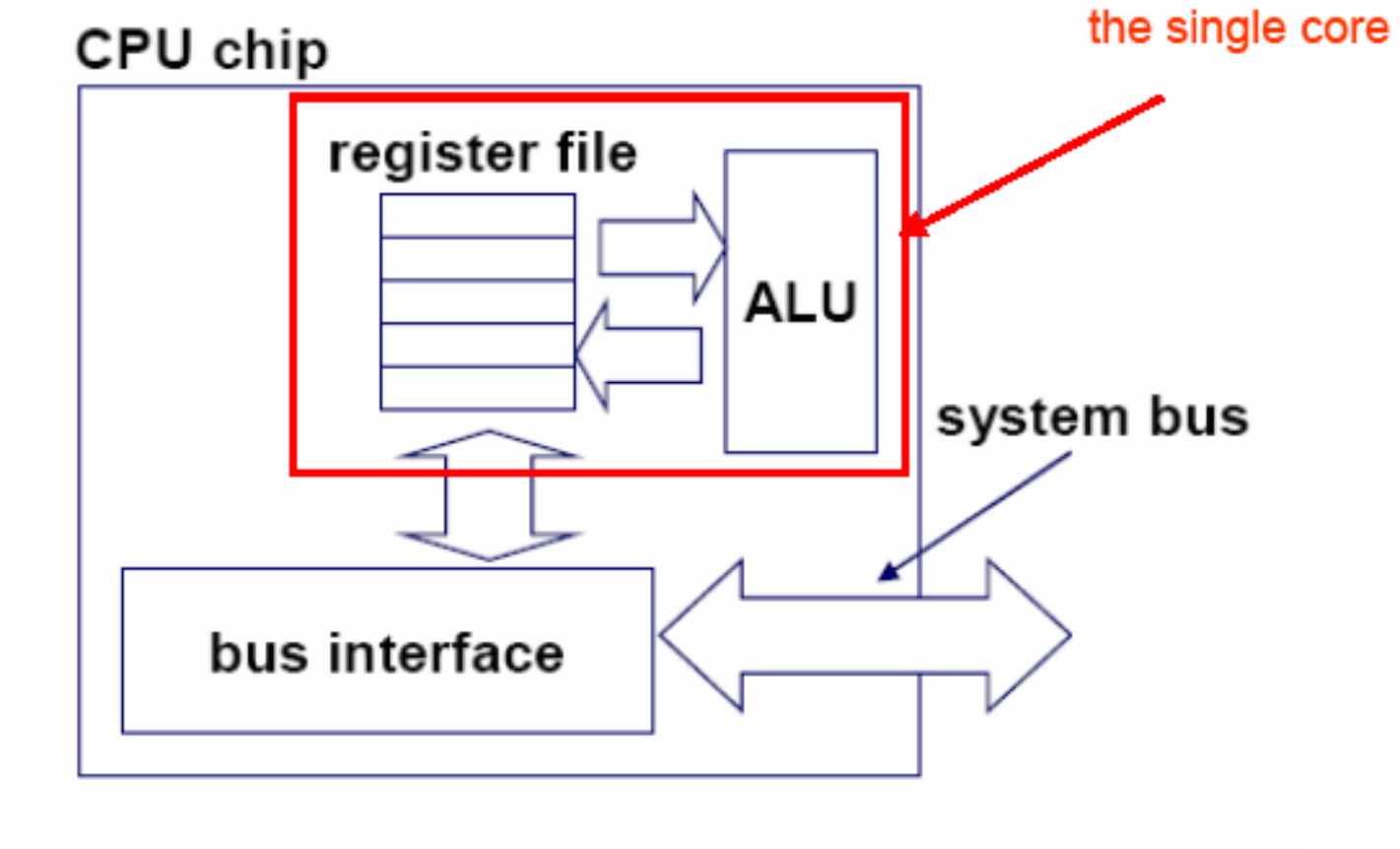
```
   movl $99999999, %edx
.L15:
   movl cnt, %eax
   incl %eax
   decl %edx
   movl %eax, cnt
   jns  .L15
```

- Declaring variable as volatile forces it to be kept in memory
- Shared variable read and written each iteration

Single-core computer

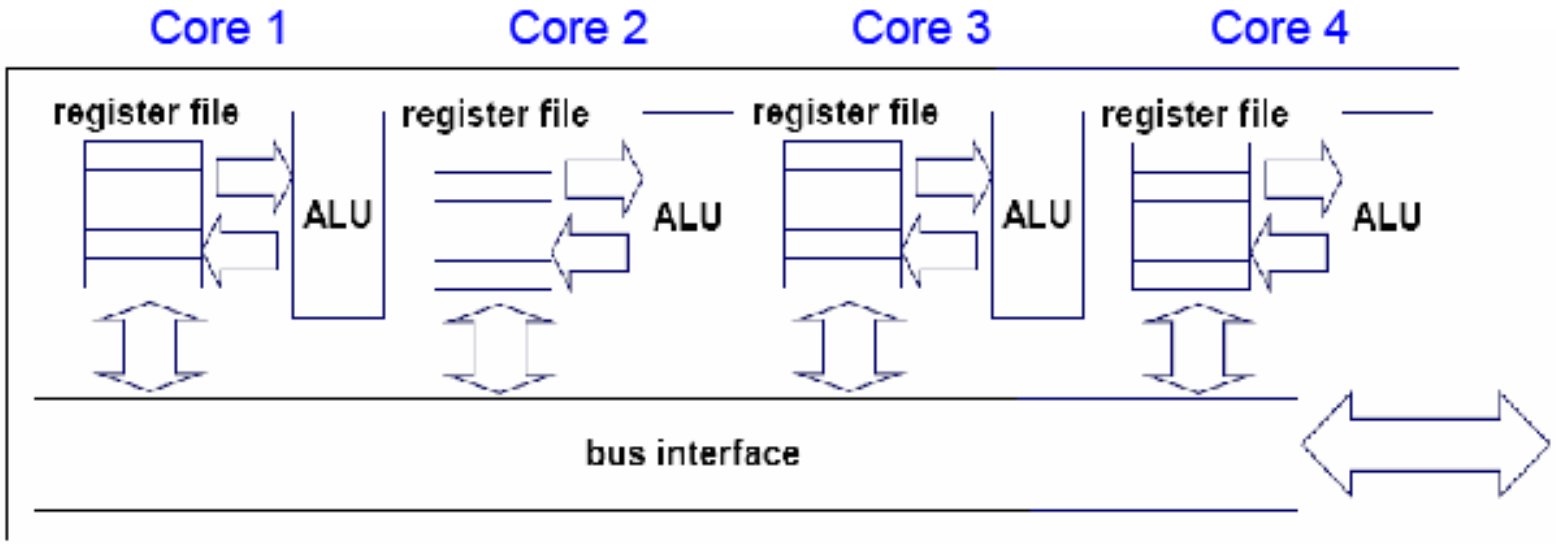


Single-core CPU chip



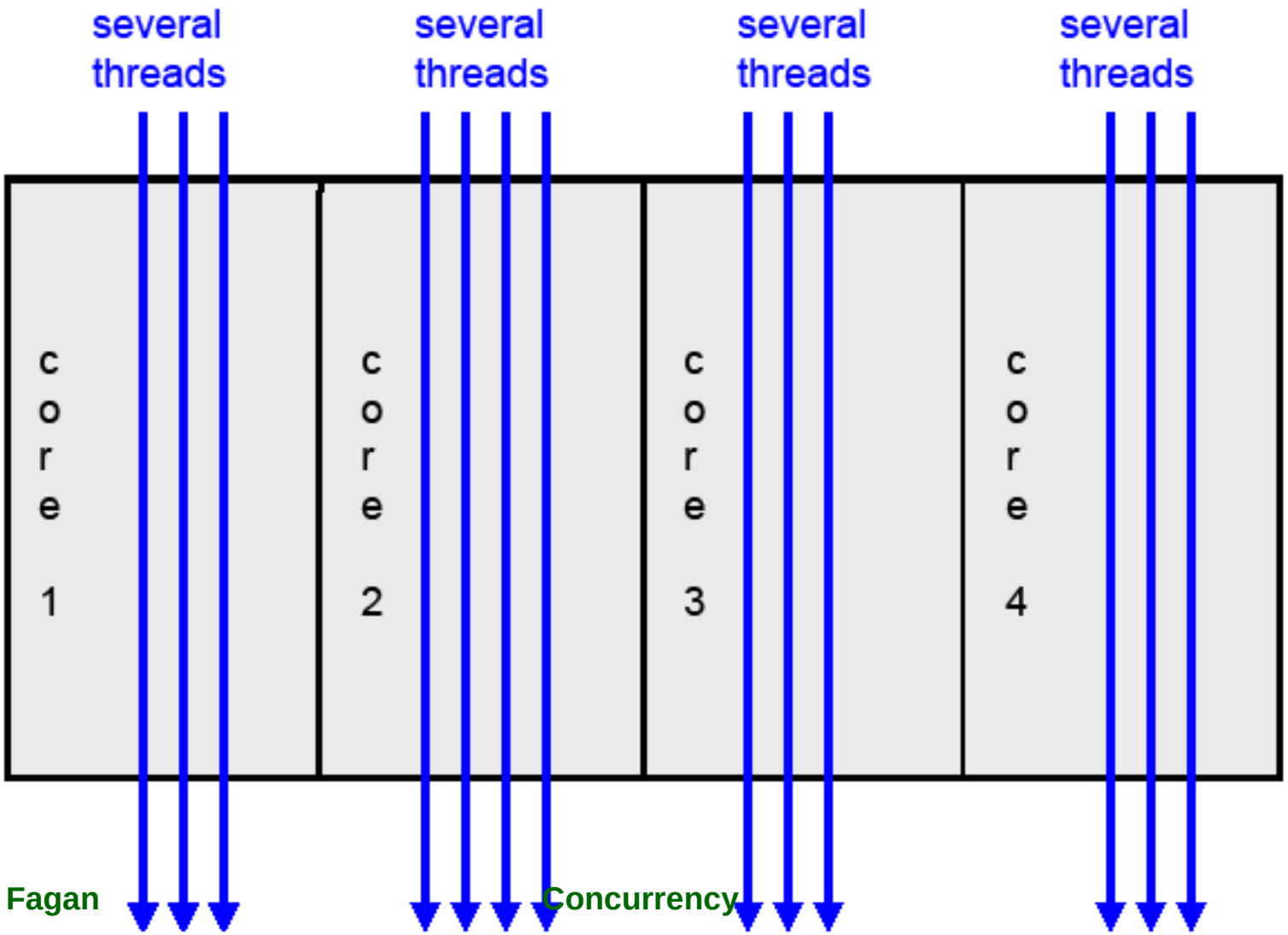
Multi-core architectures

- This lecture is about a new trend in computer architecture:
  Replicate multiple processor cores on a single die.



Multi-core CPU chip

Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction With the Operating System

- OS perceives each core as a separate processor
- OS scheduler maps threads/processesto different cores
- Most major OS support multi-core today:
  - Windows, Linux, Mac OS X, …

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning

- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

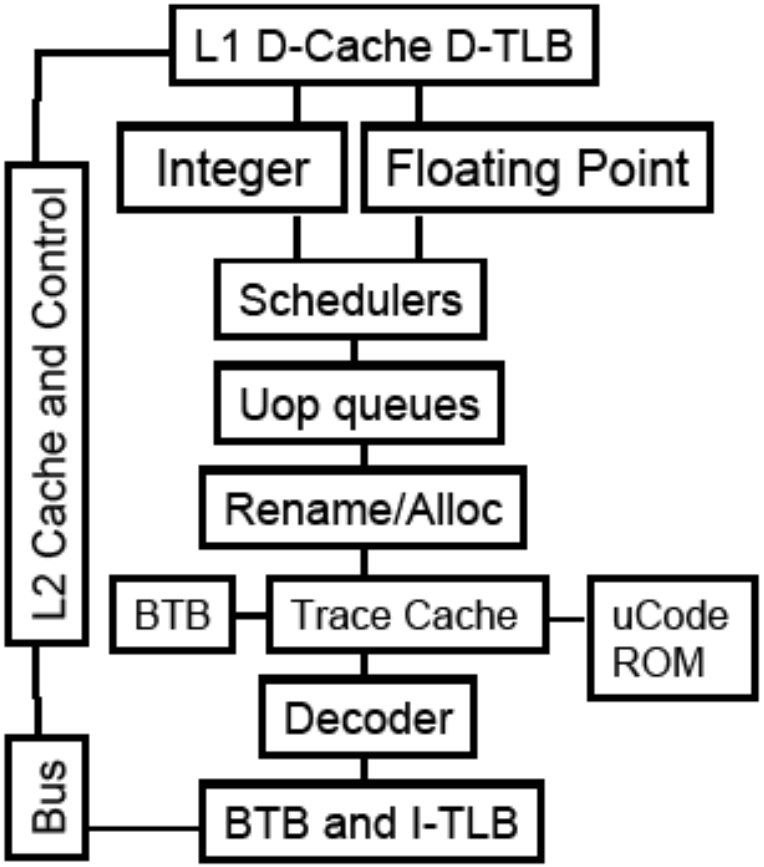
- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

A technique complementary to multi-core:

Simultaneous multithreading

- Problem addressed: The processor pipeline can get stalled:
  - Waiting for the result of a long floating point (or integer) operation
  - Waiting for data to arrive from memory

Other execution units wait unused



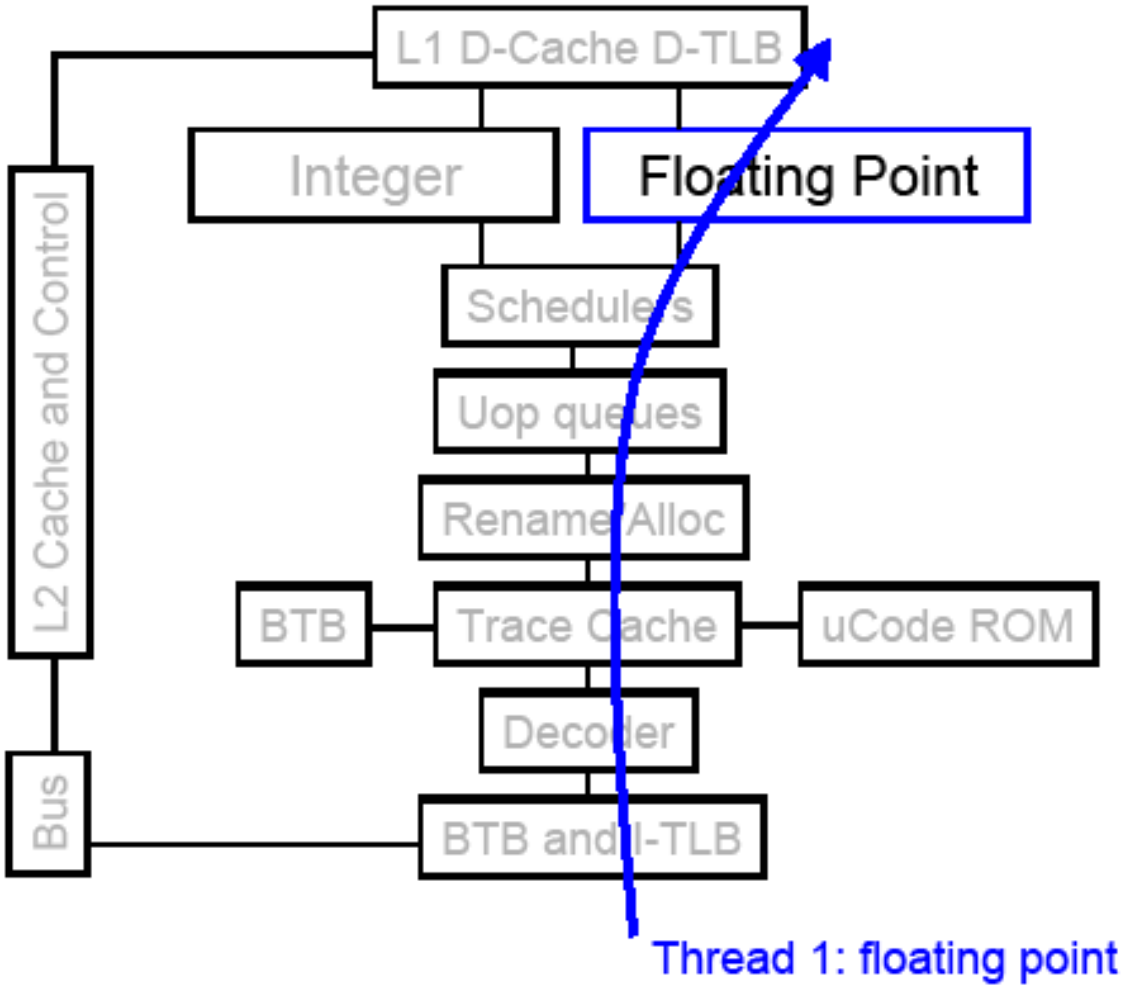
Source: Intel

Simultaneous multithreading (SMT)

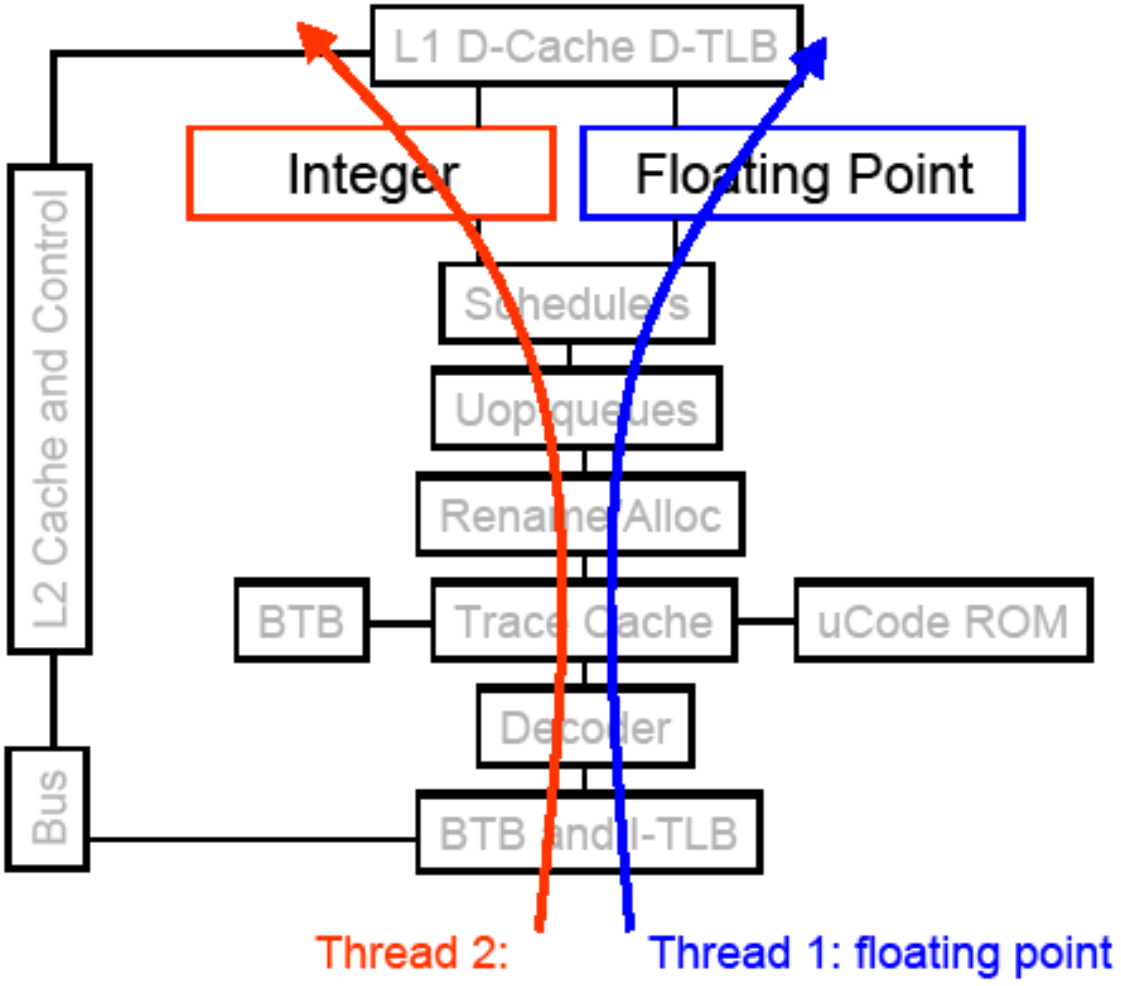
- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core

- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

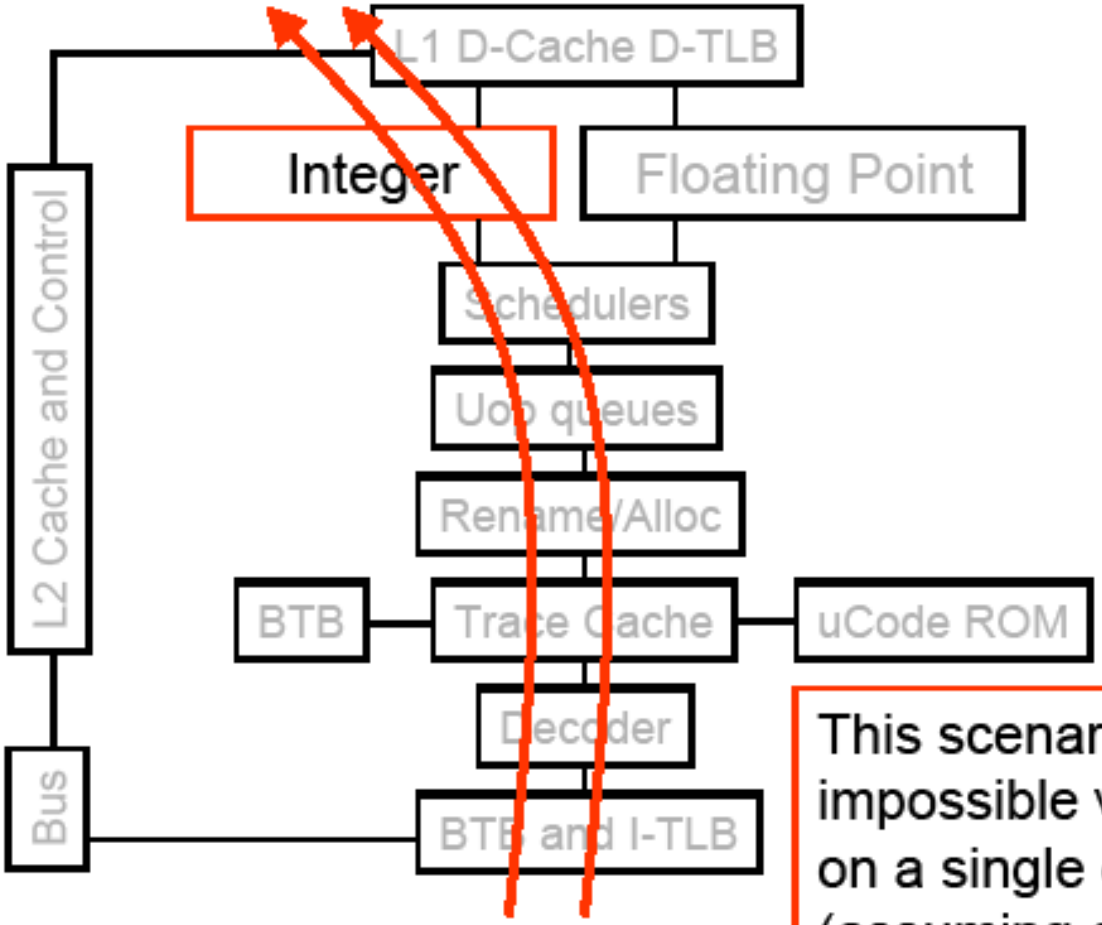
Without SMT, only a single thread can run at any given time



SMT processor: both threads can run concurrently

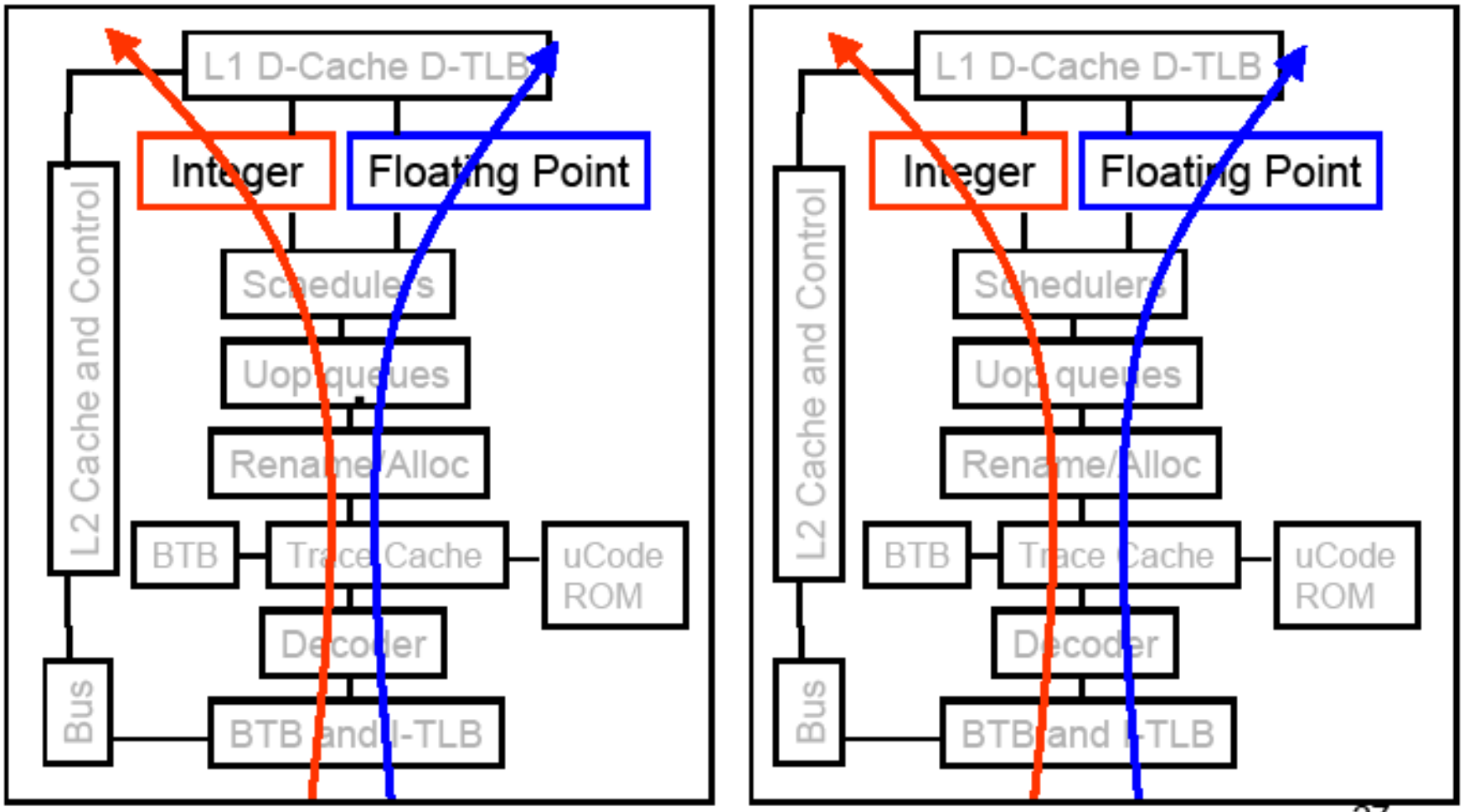


But: Can't simultaneously use the same functional unit



This scenario is impossible with SMT on a single core (assuming a single integer unit)

SMT Dual-core: all four threads can run concurrently



Comparison: multi-core vs SMT

- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

Road map

Process-based concurrency

Thread-based concurrency

Safe sharing using semaphore

Event-driven concurrency

Event-Based Concurrent Servers

I/O multiplexing

Maintain a pool of connected descriptors

Repeat the following forever:

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

Can wait for input from local I/O (standard input) and remote I/O (socket) simultaneously!

Event-Based Concurrent I/O

How to deal with multiple I/O operations concurrently?

- For example: wait for a keyboard input, a mouse click and input from a network connection

Select system call

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

Poll system call (same idea, different implementation)

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);

struct pollfd { int fd;           /* file descriptor */
                short events;     /* requested events */
                short revents;    /* returned events */
              };
```

Other mechanisms are also available

- /dev/poll (Solaris), /dev/epoll (Linux)
- kqueue (FreeBSD)
- Posix real-time signals + sigtimedwait()
- Native Posix Threads Library (NPTL)

The `select` Function

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

```
#include <sys/time.h>
int select(int nfds, 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**

`nfds`

- **Maximum descriptor value + 1 in the set**
- **Tests descriptors 0, 1, 2, ..., `nfds` - 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`

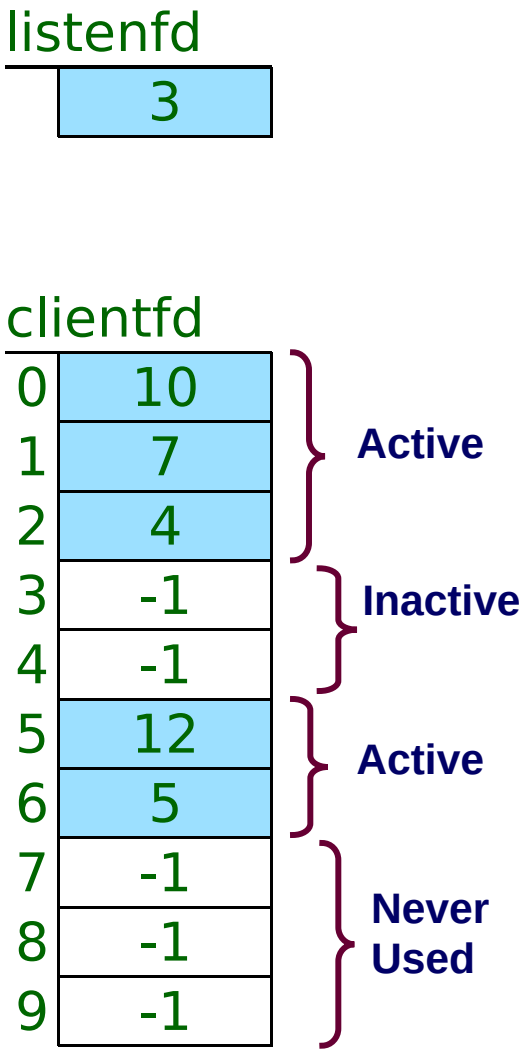
```
void FD_CLR(int fd, fd_set *fdset);
```

- Turn off bit `fd` in `fdset`

```
int FD_ISSET(int fd, *fdset);
```

- Is bit `fd` in `fdset` turned on?

Concurrent echo server with select



Representing a Pool of Clients

```
/*
 * 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 */
```

Pool Example

maxfd = 12

maxi = 6

read_set = {3,4,5,7,10,12}

listenfd

3

clientfd

0	10
1	7
2	4
3	-1
4	-1
5	12
6	5
7	-1
8	-1
9	-1

Main Loop

```
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);
  }
}
```

Pool Initialization

```
/* 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);
}
```

Initial Pool

maxfd = 3

maxi = -1

read_set = { 3 }

listenfd

3

clientfd

0	-1
1	-1
2	-1
3	-1
4	-1
5	-1
6	-1
7	-1
8	-1
9	-1

Adding Client

```
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");
}
```

Adding Client with fd 11

maxfd = 12

maxi = 6

read_set = {3,4,5,7,10,11,12}

listenfd

3

clientfd

0	10
1	7
2	4
3	-1
4	-1
5	12
6	5
7	-1
8	-1
9	-1

clientfd

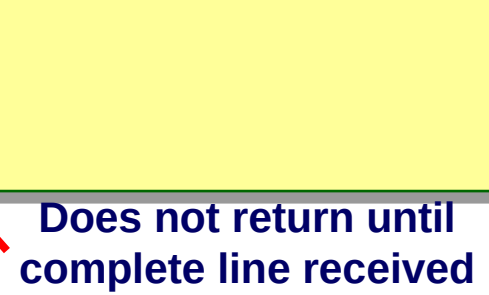
0	10
1	7
2	4
3	11
4	-1
5	12
6	5
7	-1
8	-1
9	-1

Checking Clients

```
/* echo line from ready descs in pool p */
void check_clients(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;
      }
    }
  }
}
```

Concurrency Limitations

```
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);
    }
}
```



- Current design will hang up if partial line transmitted
- Bad to have network code that can hang up if client does something weird
  - By mistake or maliciously
- Would require more work to implement more robust version
  - Must allow each read to return only part of line, and reassemble lines within server

Pros and Cons of Event-Based Designs

+ One logical control flow
+ Can single-step with a debugger
+ No process or thread control overhead

- More complex code than process- or thread-based designs
- Harder to provide fine-grained concurrency
  - E.g., our naïve design will hang up with partial lines
- Does not make use of multiple cores

Approaches to Concurrency

Process-based

- Hard to share resources: Easy to avoid unintended sharing
- High overhead in adding/removing clients

Thread-based

- Easy to share resources: Perhaps too easy
- Medium overhead
- Not much control over scheduling policies
- Difficult to debug
  - Event orderings not repeatable

Event-based

- Tedious and low level
- Total control over scheduling
- Very low overhead
- Cannot create as fine grained a level of concurrency
- S

High-performance Web servers and search engines often combine multiple mechanisms

Next Time

TBD