# 15-213

*"The course that gives CMU its Zip!"*

# Concurrent Programming
# November 18, 2008

**Topics**

- Event-based concurrent servers
- Shared variables
- The need for synchronization
- Synchronizing with semaphores

`lecture-23.ppt`

# 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 automatically interleaves multiple logical flows
- Each flow shares the same address space

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

- Programmer manually interleaves multiple logical flows
- All flows share the same address space
- Popular for high-performance server designs

# Appr. #3: 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

```
#include <sys/select.h>

int select(int maxfdp1, fd_set *readset, NULL, NULL, NULL);
```

**`readset`**

- Opaque bit vector (max FD_SETSIZE bits) that indicates membership in a *descriptor set*
- If bit k is 1, then descriptor k is a member of the descriptor set

**`maxfdp1`**

- Maximum descriptor in descriptor set plus 1
- Tests descriptors 0, 1, 2, ..., maxfdp1 - 1 for set membership

`select()` returns the number of ready descriptors and sets each bit of `readset` to indicate the ready status of its corresponding descriptor

# Macros for Manipulating Set Descriptors

```
void FD_ZERO(fd_set *fdset);
```

- **Turn off all bits in `fdset`**

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

- **Turn on bit `fd` in `fdset`**

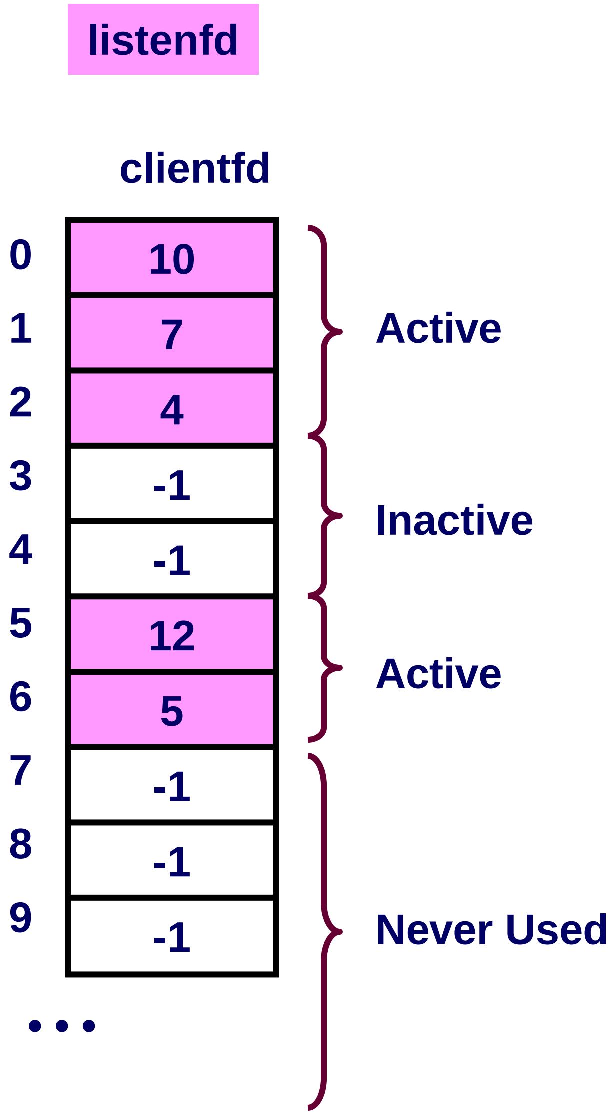
```
void FD_CLR(int fd, fd_set *fdset);
```

- **Turn off bit `fd` in `fdset`**

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

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

# Overall Structure




## Manage Pool of Connections

- listenfd: Listen for requests from new clients
- Active clients: Ones with a valid connection

## Use select to detect activity

- New request on listenfd
- Request by active client

## Required Activities

- Adding new clients
- Removing terminated clients
- Echoing

# Representing Pool of Clients

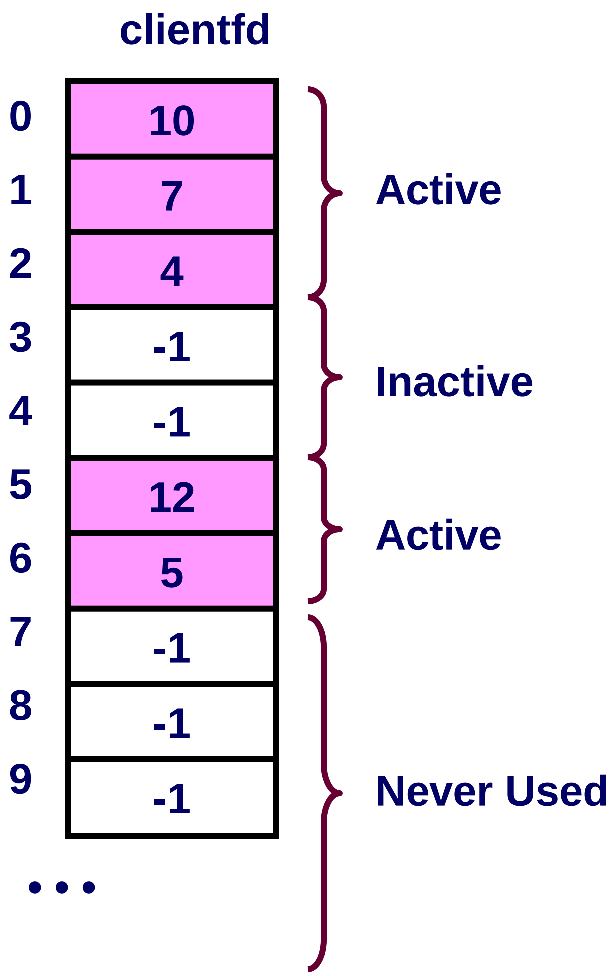
```
/*
 * echoservers.c - A concurrent echo server based on select
 */
#include "csapp.h"

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

listenfd = 3




- maxfd = 12
- maxi = 6
- read_set = { 3, 4, 5, 7, 10, 12 }

# 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

listenfd = 3

clientfd

| | clientfd |
|---|---|
| 0 | -1 |
| 1 | -1 |
| 2 | -1 |
| 3 | -1 |
| 4 | -1 |
| 5 | -1 |
| 6 | -1 |
| 7 | -1 |
| 8 | -1 |
| 9 | -1 |
| • • • | |

Never Used

- maxfd = 3
- maxi = -1
- read_set = { 3 }

# 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);
    }
}
```

# 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

listenfd = 3

- maxfd = 12
- maxi = 6
- read_set = { 3, 4, 5, 7, 10, 11, 12 }

| | clientfd | |
|---|---|---|
| 0 | 10 | |
| 1 | 7 | Active |
| 2 | 4 | |
| 3 | 11 | Inactive |
| 4 | -1 | |
| 5 | 12 | Active |
| 6 | 5 | |
| 7 | -1 | |
| 8 | -1 | |
| 9 | -1 | Never Used |
| • • • | | |

# Checking Clients

```
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;
            }
        }
    }
}
```

# 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);
    }
}
```

Does not return until complete line received

- Current design will gets stuck if partial line transmitted
- Bad to have network code that can get stuck 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

# 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

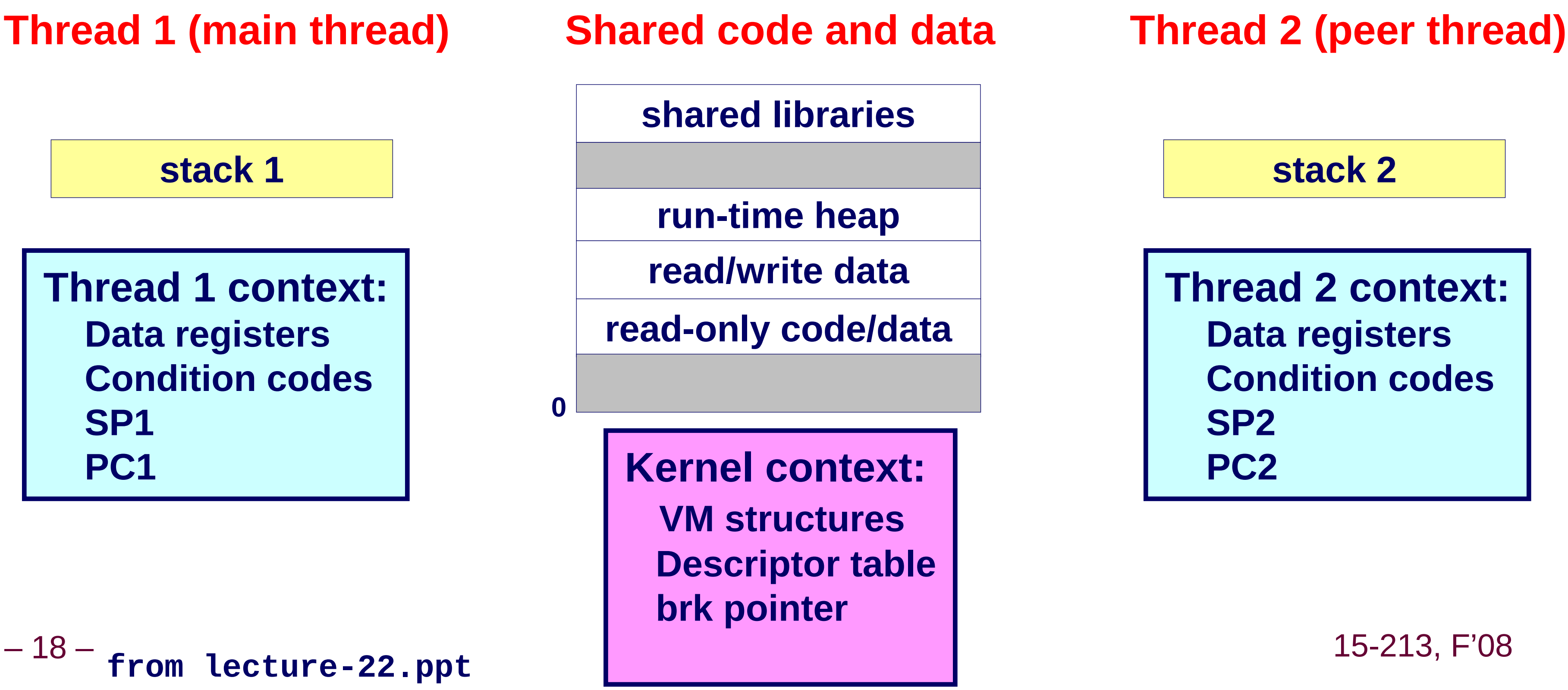
- Significantly more complex to code than process- or thread-based designs

- Hard to provide fine-grained concurrency
  - E.g., our example will hang up with partial lines

# A Process With Multiple Threads

## Multiple threads can be associated with a process

- Each thread has its own logical control flow
- Each thread shares the same code, data, and kernel context
  - Share common virtual address space
- Each thread has its own thread id (TID)






`from lecture-22.ppt`

# 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)


`from lecture-22.ppt`

# Shared Variables in Threaded C 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 each memory instance?
- How many threads might reference each of these instances?

# Threads Memory Model

**Conceptual model:**

- **Multiple threads run within the context of a single process**
- **Each thread has its own separate thread context**
  - **Thread ID, stack, stack pointer, program counter, condition codes, 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 handlers**

**Operationally, this model is not strictly enforced:**

- **While register values are truly separate and protected....**
- **Any thread can read and write the stack of any other thread**

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

# Example of Threads Accessing Another Thread's Stack

```
char **ptr;  /* global */

int main()
{
    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);
}
```

```
/* thread routine */
void *thread(void *vargp)
{
    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 Variables to Mem. Instances

***Global var*: 1 instance (`ptr` [data])**

***Local automatic vars*: 1 instance (`i.m, msgs.m` )**

```
char **ptr;  /* global */

int main()
{
    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);
}
```

***Local automatic var:* 2 instances (**
**`myid.p0`[peer thread 0's stack],**
**`myid.p1`[peer thread 1's stack]**
**)**

```
/* thread routine */
void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int svar = 0;

    printf("[%d]: %s (svar=%d)\n",
         myid, ptr[myid], ++svar);
}
```

***Local static var*: 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: An Improperly Synchronized Threaded Program

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

int main() {
    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);
}
```

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

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

linux> ./badcnt
BOOM! cnt=198261801

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

**cnt should be equal to 200,000,000. What went wrong?!**

# Assembly Code for Counter Loop

**C code for counter loop**

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

**Corresponding asm 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 are incorrect!**

- $I_i$ denotes that thread i executes instruction I
- $\%eax_i$ is the contents 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)

**Incorrect 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!**

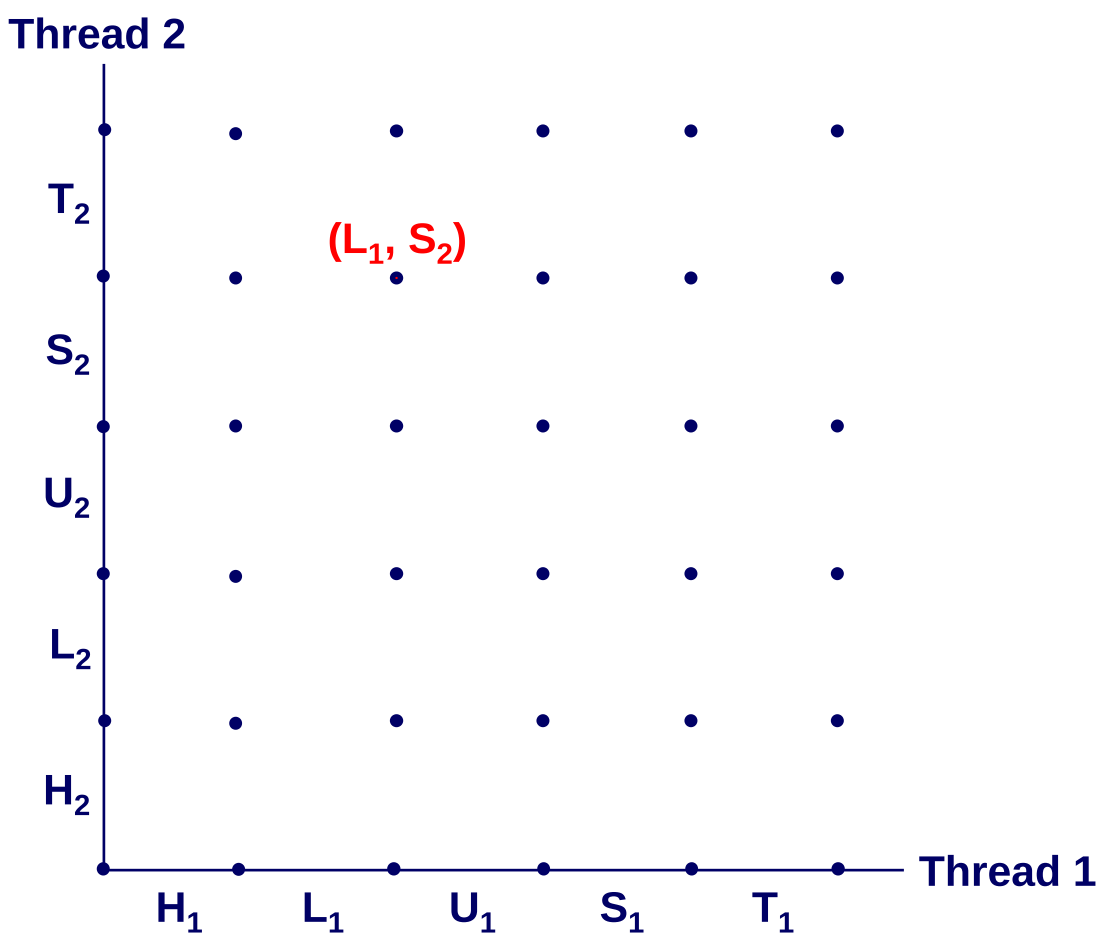
# Concurrent Execution (cont)

## 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 clarify our understanding of concurrent execution with the help of the *progress graph***

# Progress Graphs



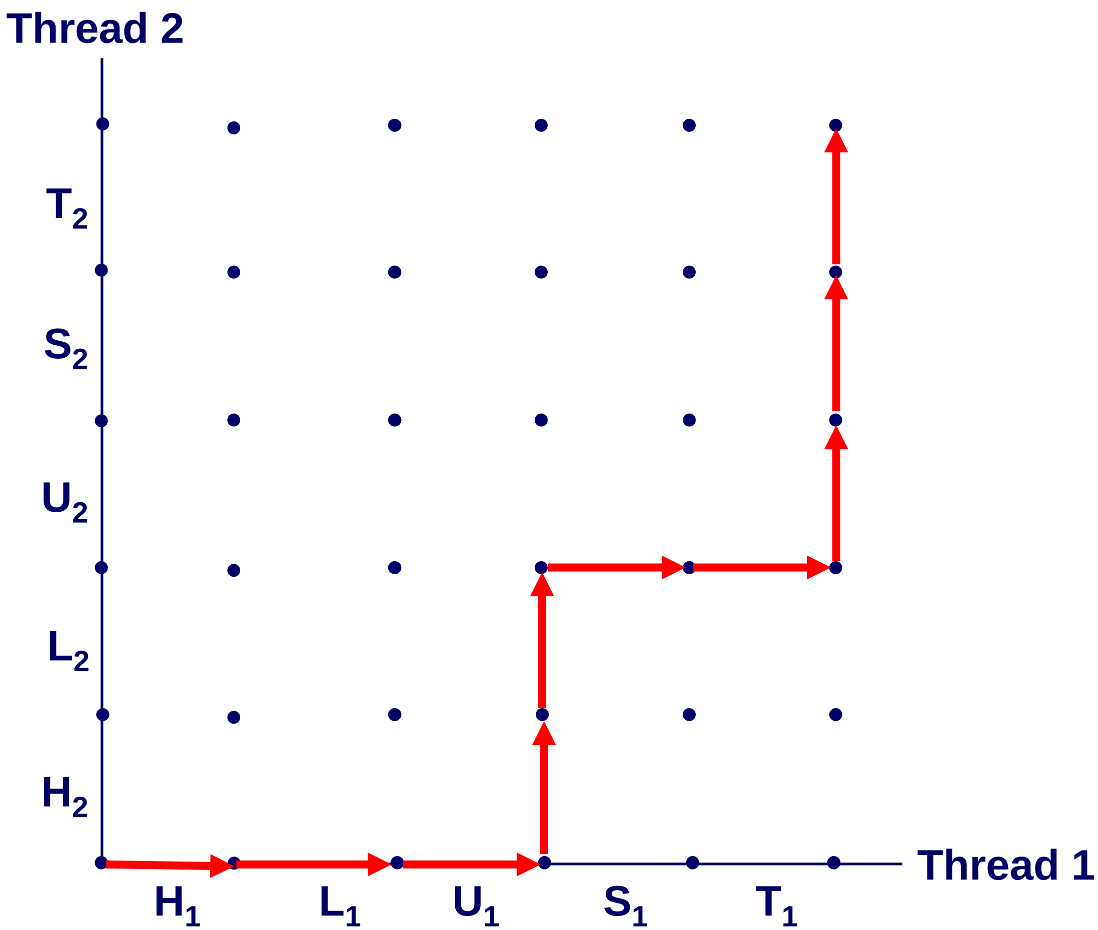

A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

Each point corresponds to a possible ***execution state*** $(Inst_1, Inst_2)$.

E.g., $(L_1, S_2)$ denotes state where thread 1 has completed $L_1$ and thread 2 has completed $S_2$.

# Trajectories in Progress Graphs

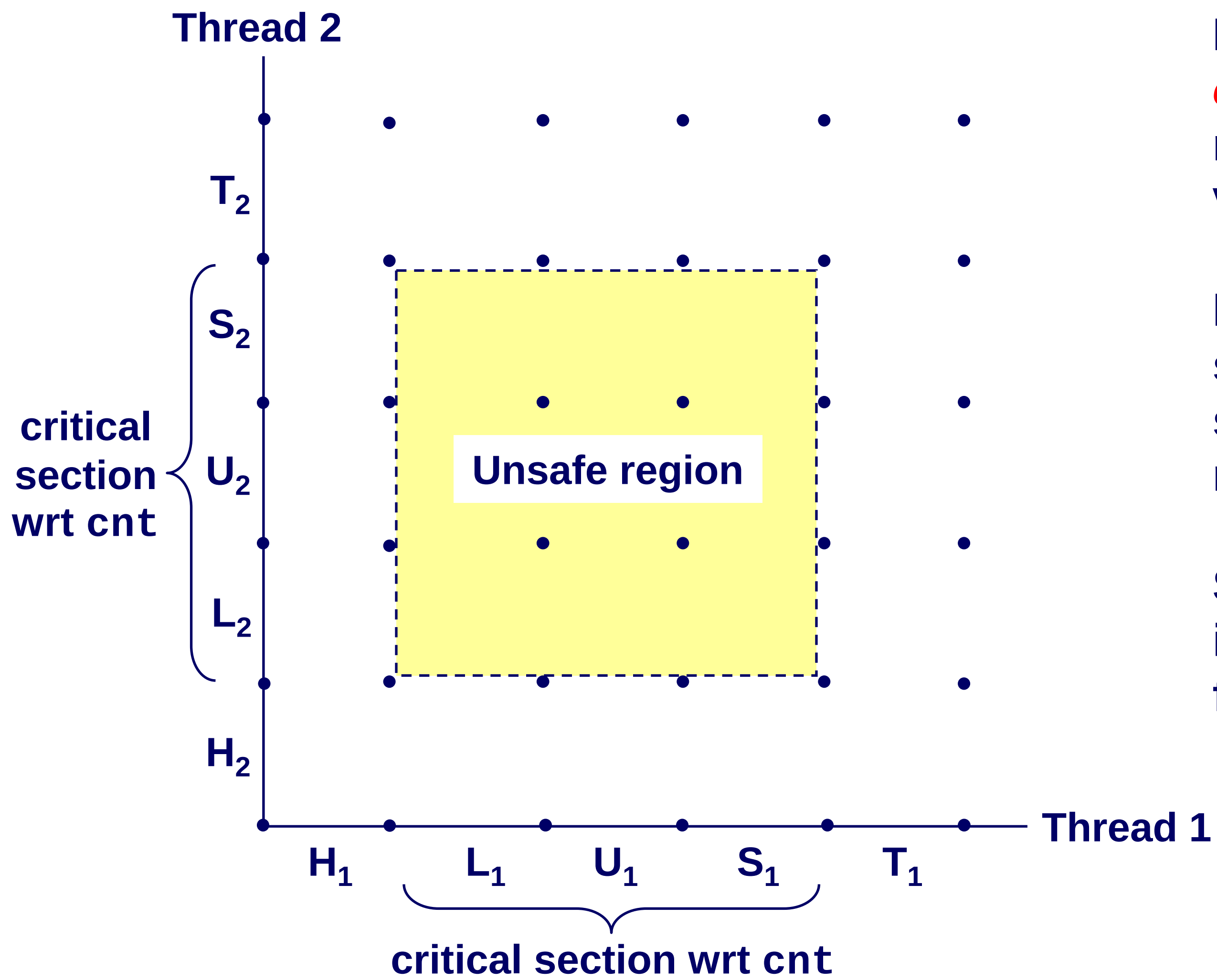



A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

**Example:**

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

# Critical Sections and Unsafe Regions

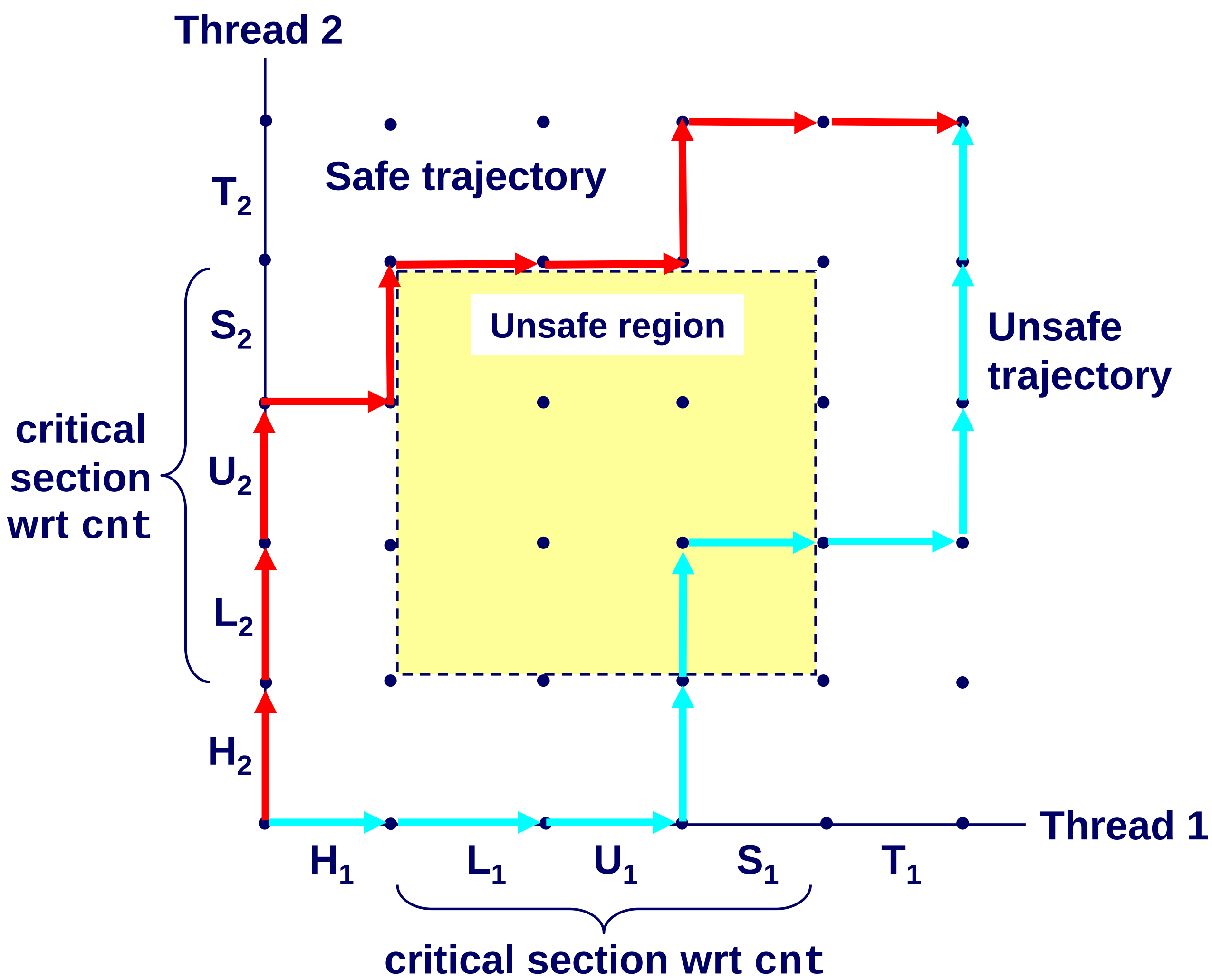



L, U, and S form a ***critical section*** with respect to the shared variable `cnt`.

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.

# Semaphores

***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

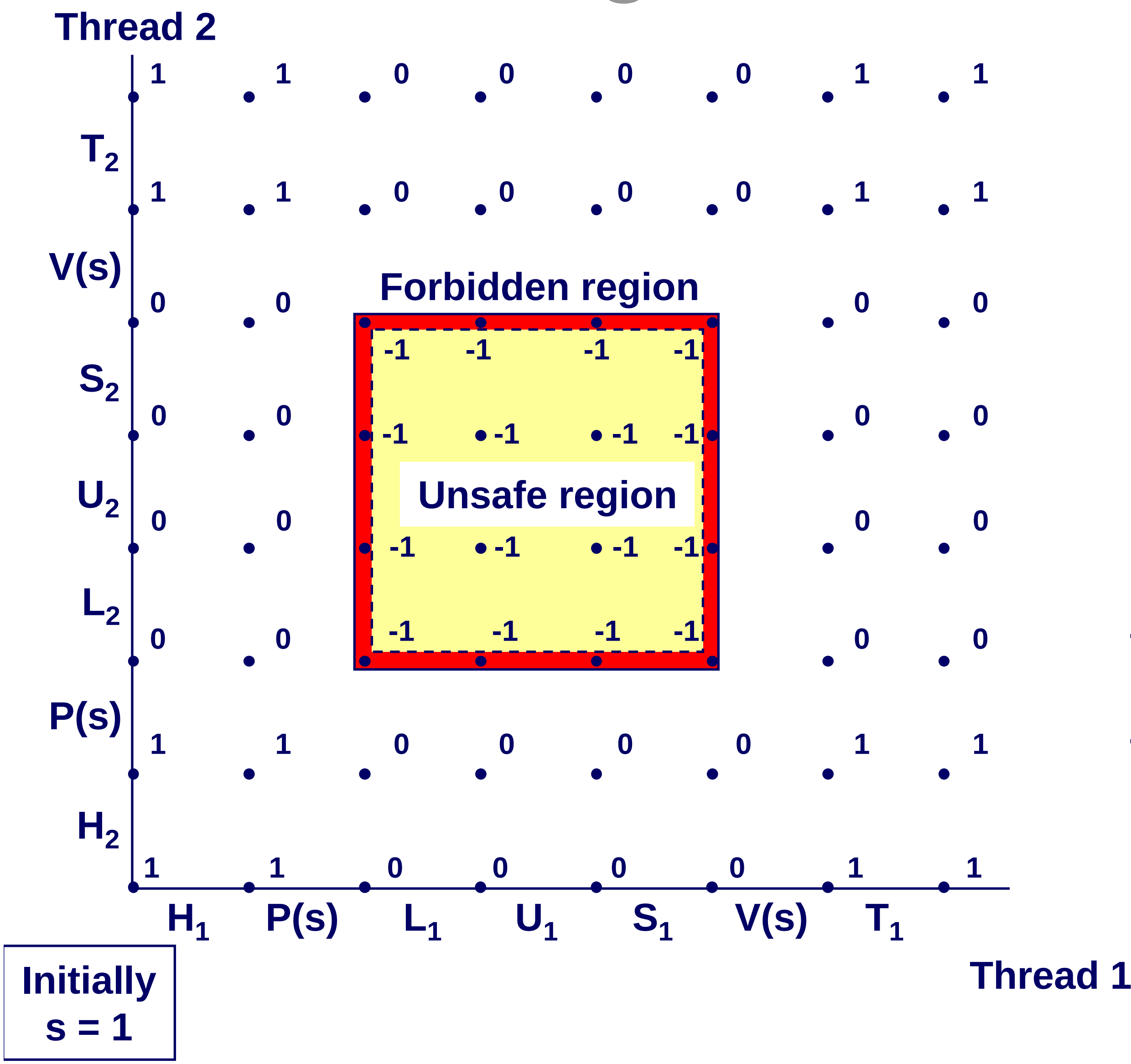
Here is how we would use P and V operations to synchronize the threads that update `cnt`.

```
/* Semaphore s is initially 1 */

/* Thread routine */
void *count(void *arg)
{
    int i;

    for (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.**