
WELCOME
TO
15-213
14-513
18-213
18-613
15-513

Concurrent Programming

18-213/18-613: Introduction to Computer Systems
21st Lecture, August 20, 2023

Outline

- **Concurrency**
- Concurrency Hazards
- Processes Reminder
- Threads
- Sharing
- Reasoning about Sharing
- Mutual Exclusion

Concurrency

- We've played a bit with "Fork bombs"
- They were "hard" to sort out, right?
- Why?

- The reason is a phenomenon known as *concurrency*
- Today, we are going to explore this phenomenon and look at another model for implementing it

- For a quick one-liner, *concurrency* is the overlapping of activities in time, whether through parallelism or interleaving (turn taking)
  - It is to be distinguished from sequentiality, i.e. in series or one-after-the-other

Sequential thinking in a concurrent world

- **Think about the world around you**
  - Consider all of the different events occurring at exactly the same time.
  - Consider all of the different events that interleave over time, e.g. many classes meet in the same room at regular intervals, but other classes use this space at the other times

- **Now, think about how you describe complex situations**
  - Break them down into individual activities
  - Preview the activities
  - Describe each one, one at a time.
  - Describe the interactions among the activities

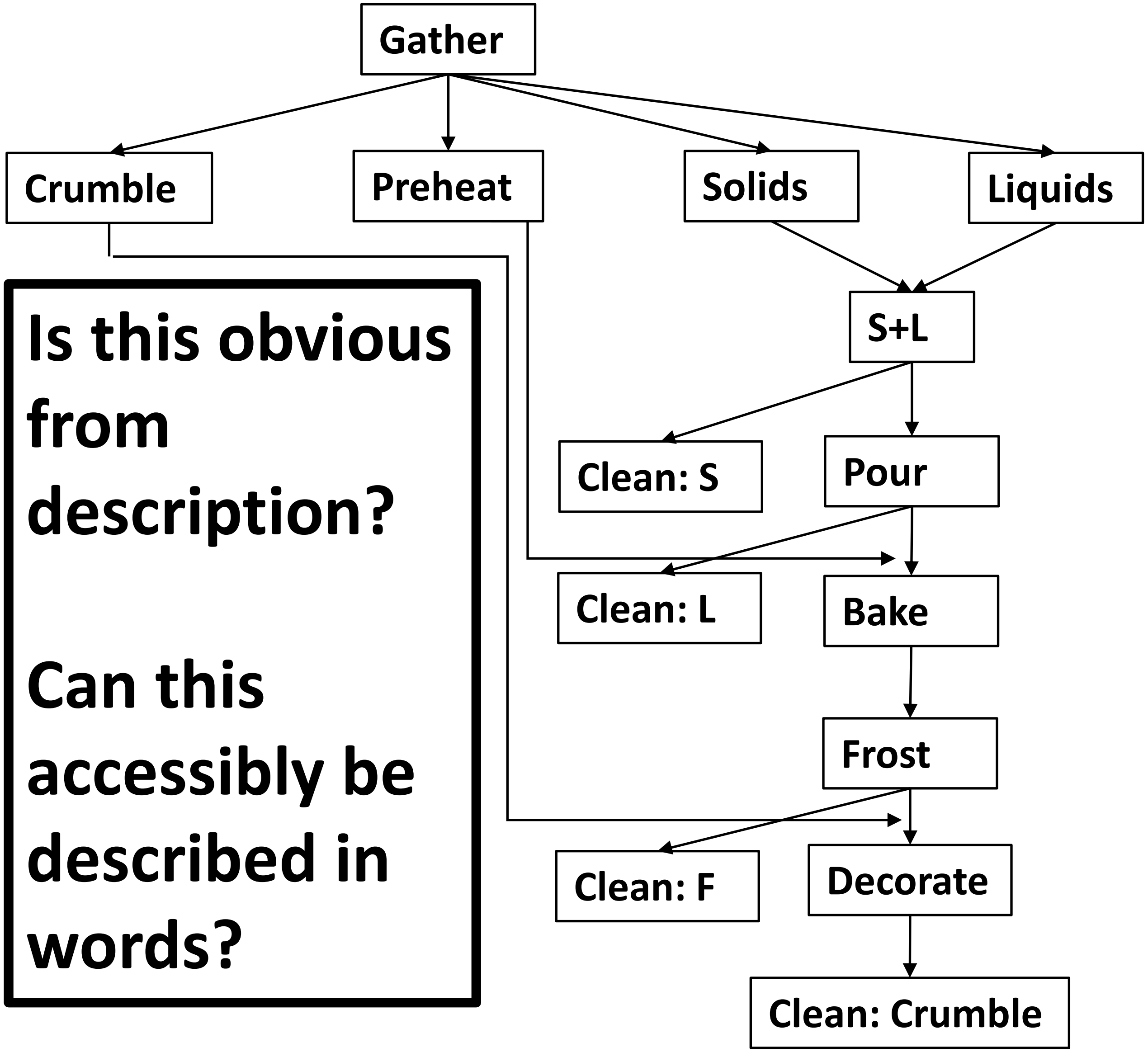
A concurrent world described sequentially

- To bake a cake, one needs to gather ingredients, preheat the oven, mix the solids, mix the liquids, mix the solids and the liquids together, crumble the cookies for the topping, pour the cake into the pan, bake the cake, frost the cake, decorate with the crumbled cookies, and then clean up.
  - Does it matter if the solids are mixed together before the liquids are mixed together [Nope]
  - Does it matter when the cookies are crumbled to long as it is before they are used [Nope]
  - Can the frosting be applied before the cake is baked? [Nope]
  - Can cleanup be done before the cake is baked [Some of it]

How many cooks can we use (and when)?

- **Gather**: Gather ingredients (Don't start unless we have all)
- **Preheat:** Preheat the oven
- **Solids**: Mix the solids
- **Liquids:** Mix the liquids
- **S+L:** Mix the solids and the liquids together
- **Crumble:** Crumble the cookies for the topping
- **Pour:** Pour the cake into the pan
- **Bake:** Bake the cake
- **Frost:** Frost the cake
- **Decorate:** Decorate with the crumbled cookies
- **Clean:** Clean up

How many cooks can we use (and when)?



Is this obvious from description?

Can this accessibly be described in words?

How many ingredients?
One person can get each?

4

1

2

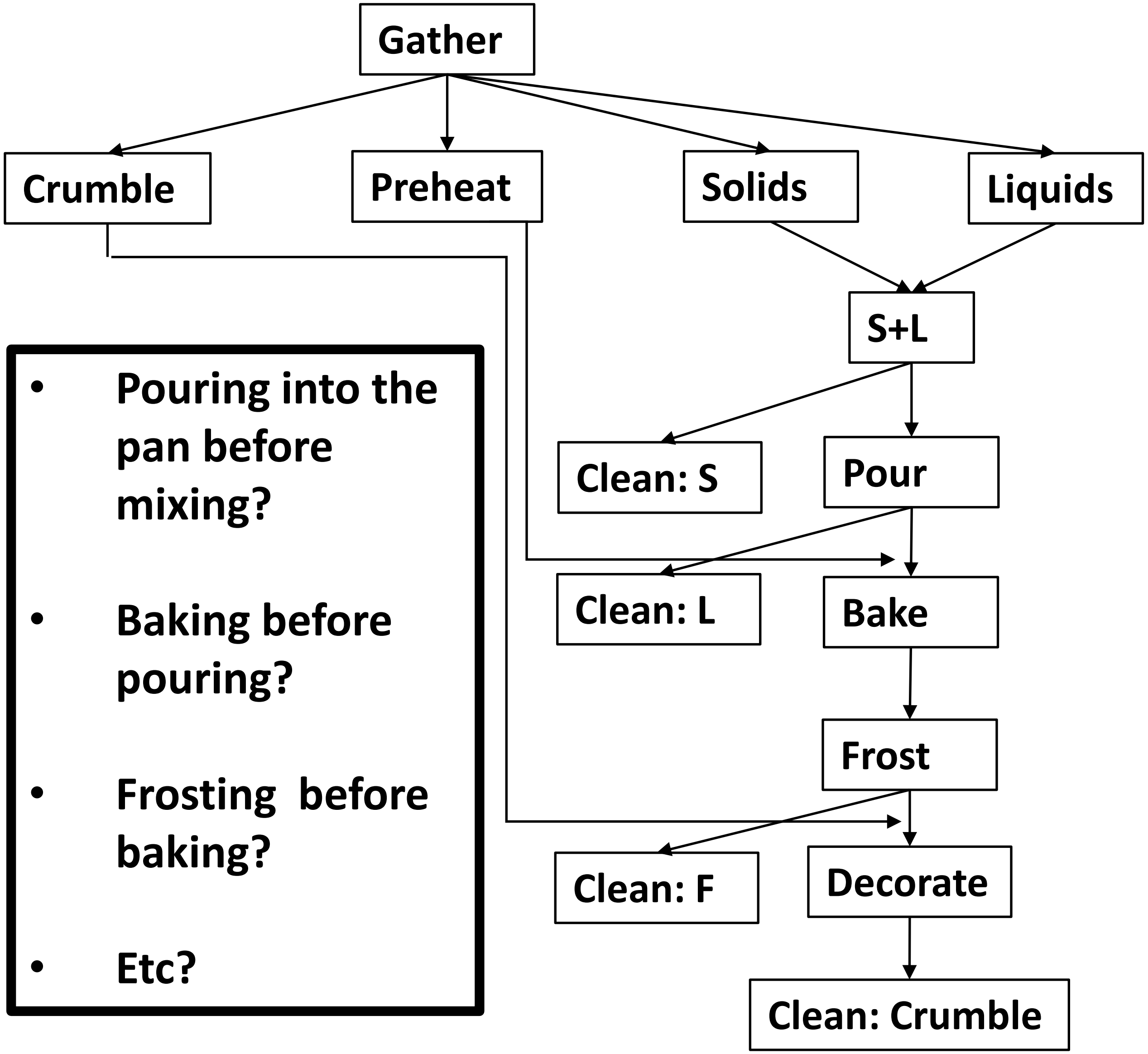
2

1? More? How many knives?
"Knife wars" in the container?

2

1

What happens if some things get out of order?



- Pouring into the pan before mixing?
- Baking before pouring?
- Frosting before baking?
- Etc?

How many ingredients?
One person can get each?

4

1

2

2

1? More? How many knives?
"Knife wars" in the container?

2

1

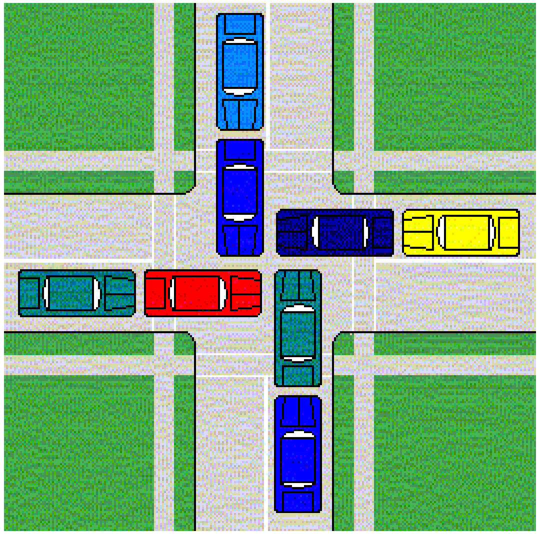
Concurrent Programming is Hard!

- 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

Outline

- Concurrency
- **Concurrency Hazards**
- Processes Reminder
- Threads
- Sharing
- Reasoning about Sharing
- Mutual Exclusion

What can go wrong? Deadlock



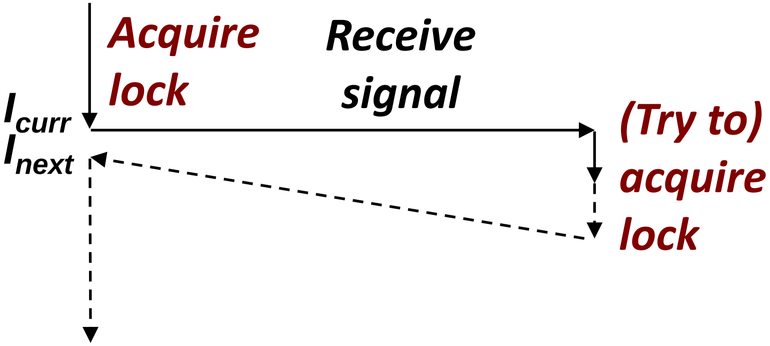
Key characteristic: Circular wait

Deadlock

- Example from signal handlers.
- Why don't we use printf in handlers?

```
void catch_child(int signo) {
   printf("Child exited!\n"); // this call may reenter printf/puts! BAD!  DEADLOCK!
   while (waitpid(-1, NULL, WNOHANG) > 0) continue;  // reap all children
}
```

- Printf code:
  - Acquire lock
  - Do something
  - Release lock



- What if signal handler interrupts call to printf?

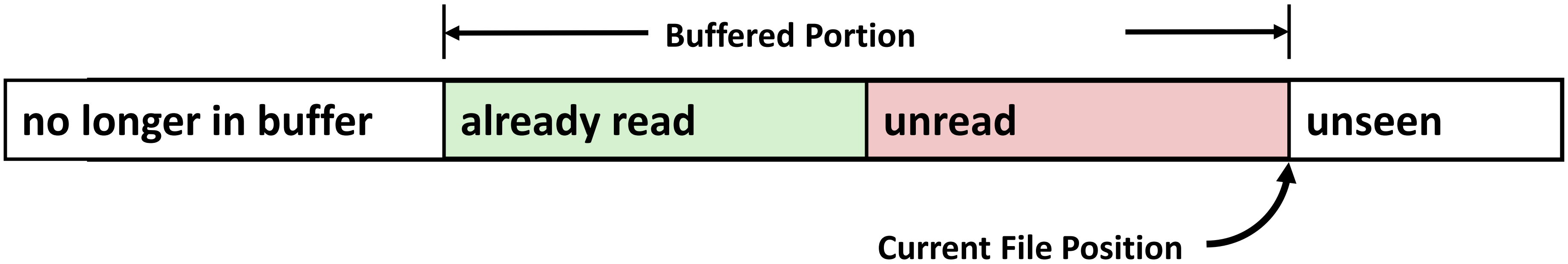
Testing Printf Deadlock

```
void catch_child(int signo) {
   printf("Child exited!\n"); // this call may reenter printf/puts! BAD!  DEADLOCK!
   while (waitpid(-1, NULL, WNOHANG) > 0) continue; // reap all children
}

int main(int argc, char** argv) {
  char buf[MAXLINE];
  int i;

  if (signal(SIGCHLD, catch_child) == SIG_ERR)
    unix_error("signal error");

  for (i = 0; i < 1000000; i++) {
    if (fork() == 0) {
      exit(0); // in child, exit immediately
    }
    // in parent
    sprintf(buf, "Child #%d started\n", i);
    printf("%s", buf);
  }
  return 0;
}
```

```
Child #0 started
Child #1 started
Child #2 started
Child #3 started
Child exited!
Child #4 started
Child exited!
Child #5 started
   .
   .
   .
Child #5888 started
Child #5889 started
```

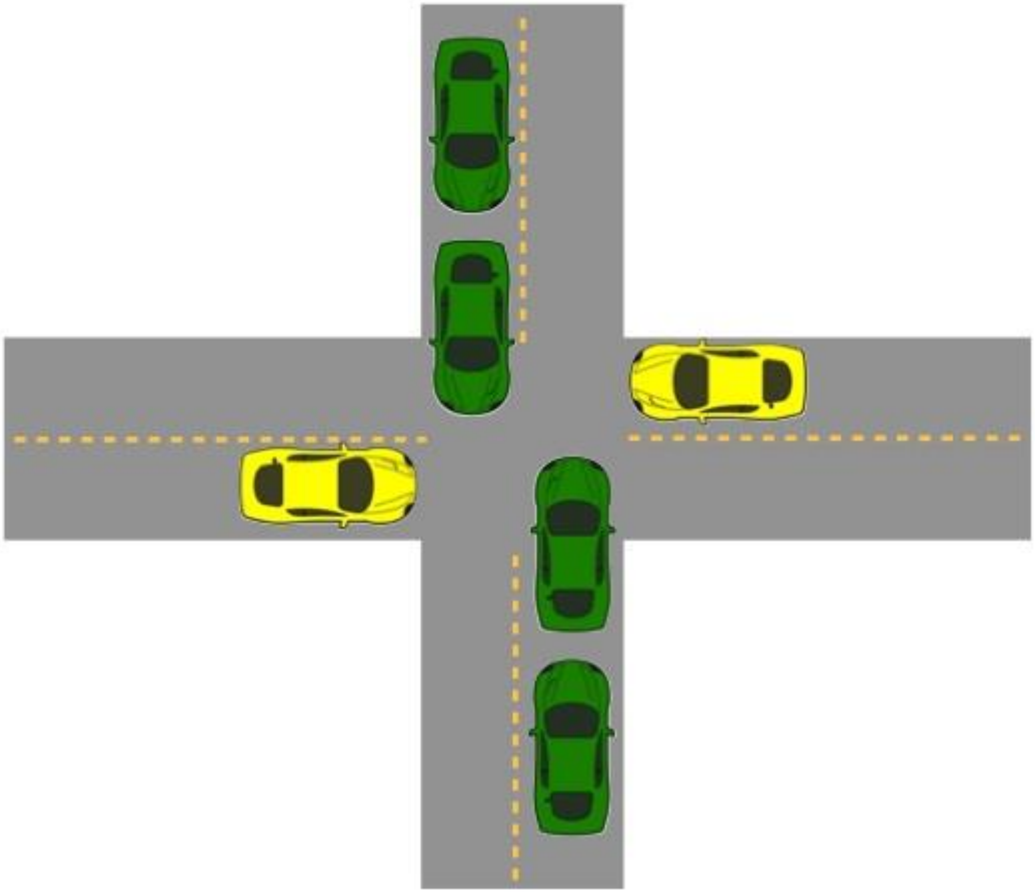
Why Does Printf require Locks?

- Printf (and fprintf, sprintf) implement *buffered* I/O



- Require locks to access to shared buffers

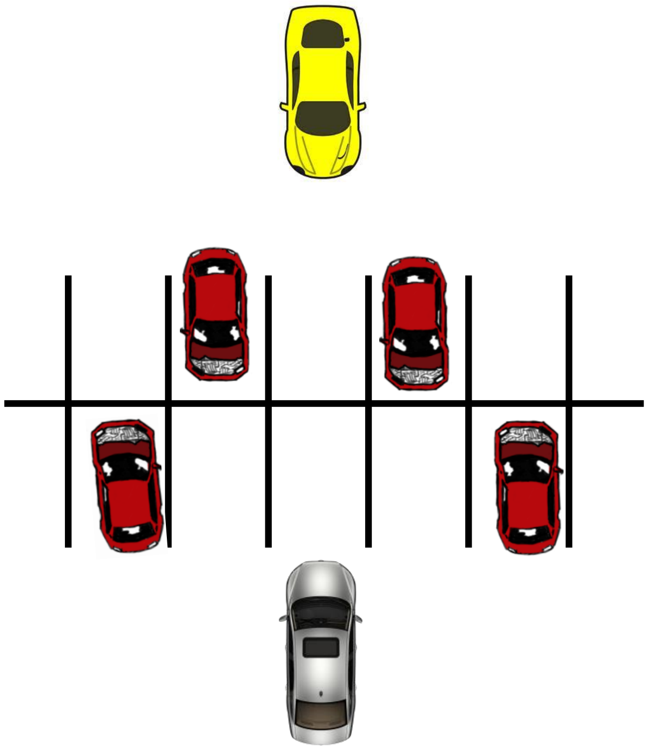
Starvation



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

Sometimes starvation is okay: If the fire trucks get to the fire in time to put it out, it is okay if the gawkers go home without "getting to see" it. Priority can cause starvation and that may be okay, sometimes, and not other times.

Data Race



If a collision occurs, and if not, which car gets the space, depends purely on timing. This isn't something the programmer specifies. It is arbitrary in the sense that it is impacted by many details that escape consideration and can vary from run to run.

Concurrent Programming is Hard!

- **Classical problem classes of concurrent programs:**
  - ***Deadlock:*** improper resource allocation prevents forward progress
    - Example: traffic gridlock
  - ***Starvation / Fairness***: external events and/or system scheduling decisions can prevent sub-task progress
    - Example: people always jump in front of you in line
  - ***Races:*** outcome depends on arbitrary scheduling decisions elsewhere in the system
    - Example: who gets the last seat on the airplane?
- **Many aspects of concurrent programming are beyond the scope of our course…**
  - but, not all ☺
  - We'll cover some of these aspects in the next few lectures.

Concurrent Programming is Hard!

It may be hard, but …

it can be useful and sometimes necessary!

Outline

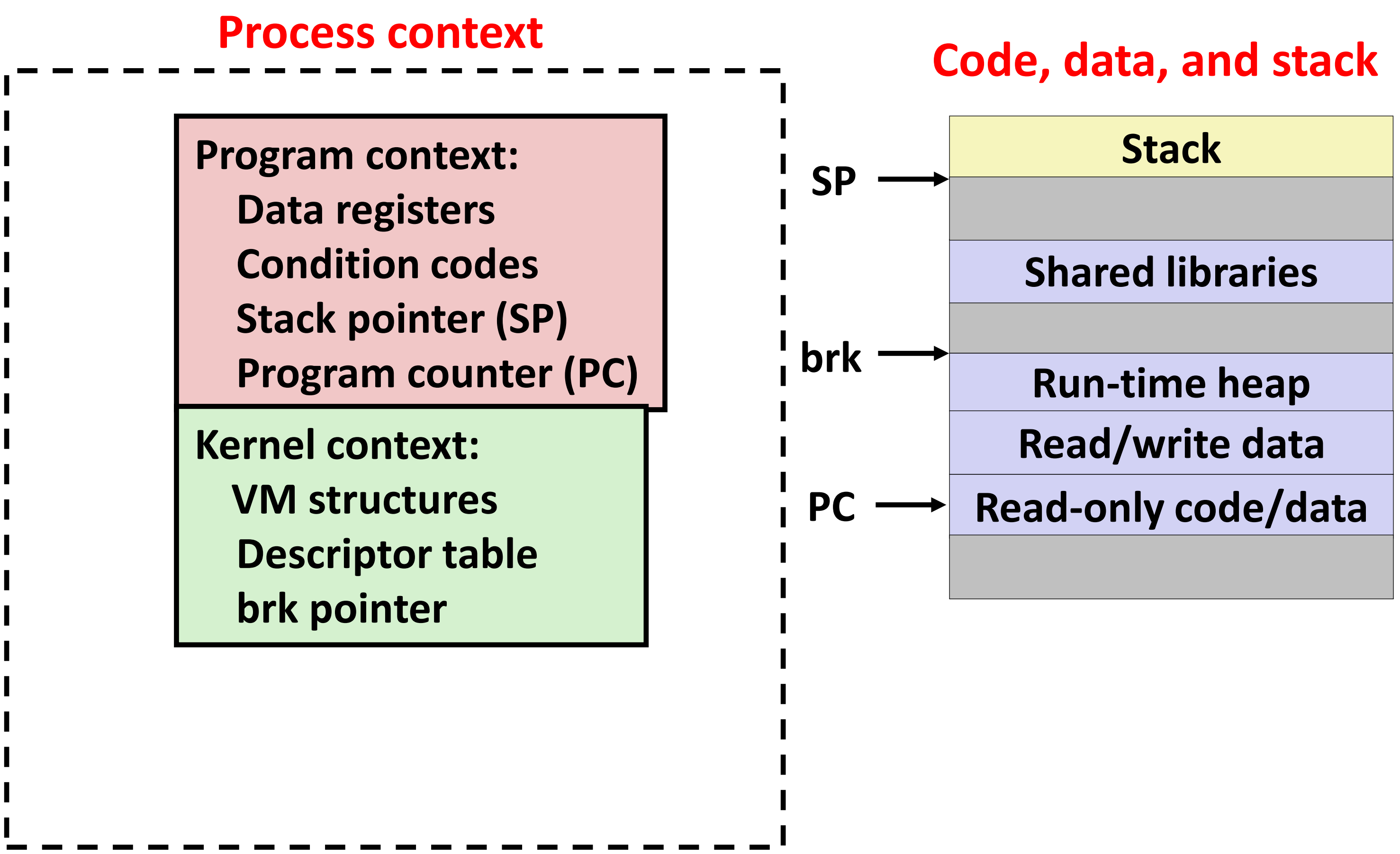
- Concurrency
- Concurrency Hazards
- **Processes Reminder**
- Threads
- Sharing
- Reasoning about Sharing
- Mutual Exclusion

Models for concurrency

- **We've already seen that processes can run concurrently**
  - Fork bombs and process graphs!
  - And, we've already seen that, when concurrent processes interact, the resulting executions can have constraints and degrees of freedom
  - The freedom can make results non-deterministic, unless we are careful
- **Each process in our model contained a full set of resources, a.k.a. contexts:**
  - Register context (general purpose registers)
  - Execution context (%rip)
  - Function call context (stack space and %esp register)
  - VM context (page table and area struct)
  - File context (file descriptor array)
  - Signal context (pending set, blocked set, handlers)
- **Painful interactions occur at resources outside of these contexts**
  - Files, keyboard, screen, network, etc.
  - Think about the confusion of what the various fork bombs would do

The Familiar: A Traditional Process

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

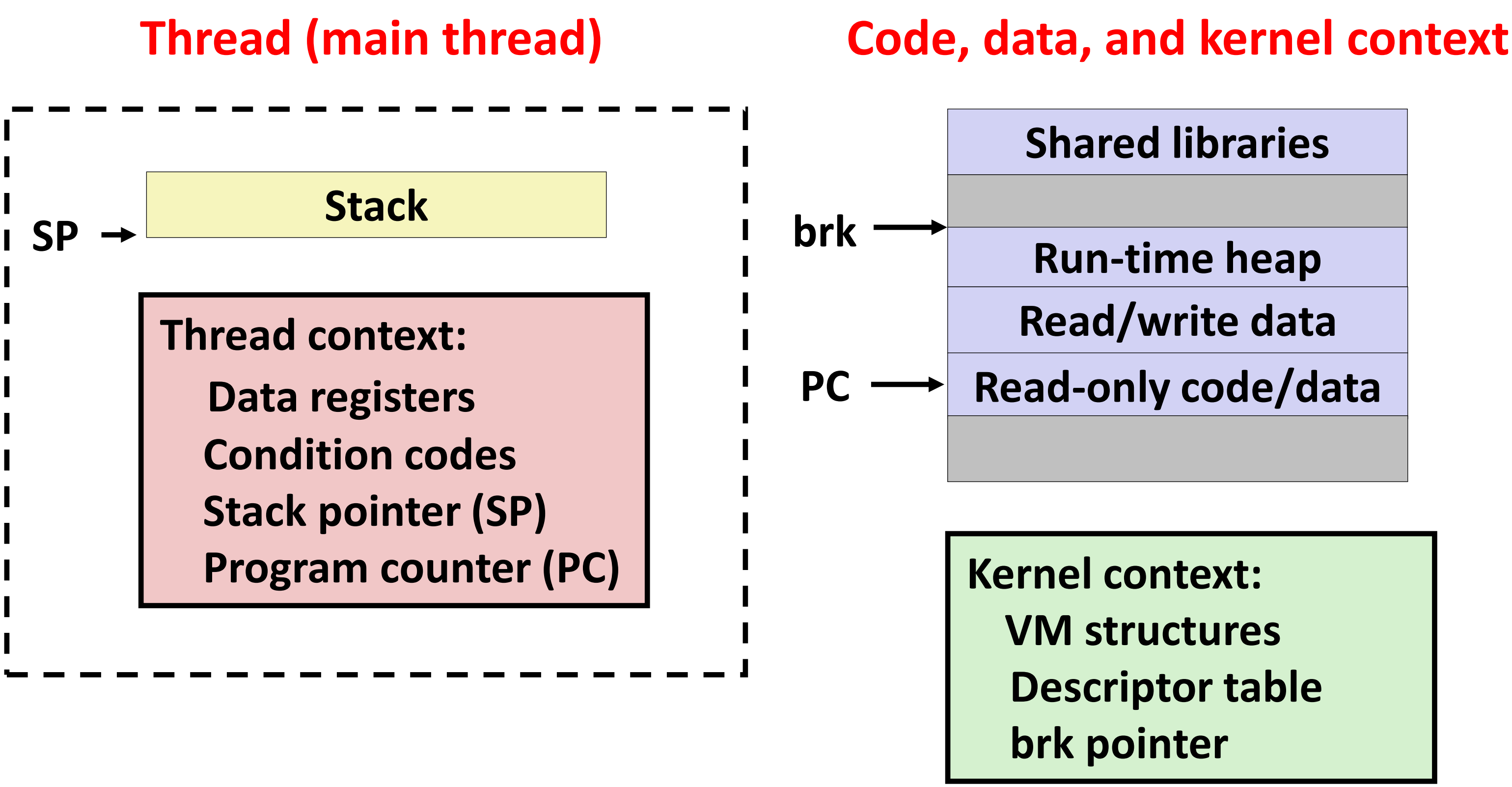


Outline

- Concurrency
- Concurrency Hazards
- Processes Reminder
- **Threads**
- Sharing
- Reasoning about Sharing
- Mutual Exclusion

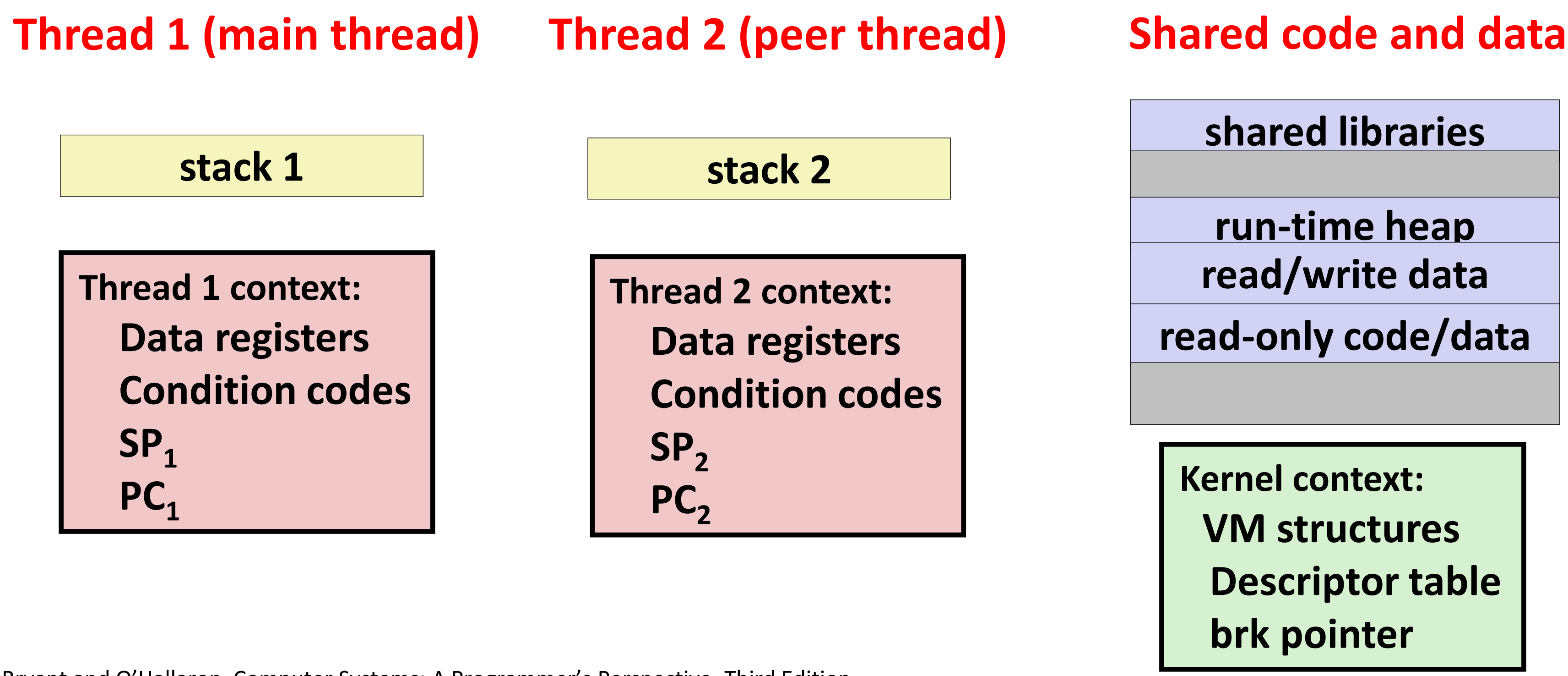
Alternate View of a Process: Separate the activity from the resources

- **Process = thread + code, data, and kernel context**
  - A thread represents an activity that uses resources in the broader whole process and whole world contexts.



A Process With Multiple Threads

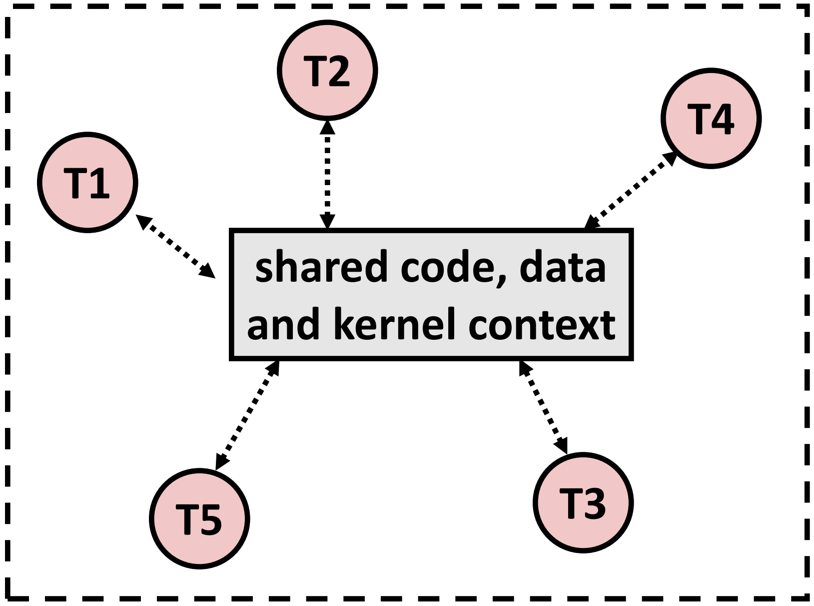
- **Multiple threads can be associated with a process**
  - Each thread has its own logical control flow
  - Each thread shares the same code, data, and kernel context
  - Each thread has its own stack for local variables
    - but not protected from other threads
  - Each thread has its own thread id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



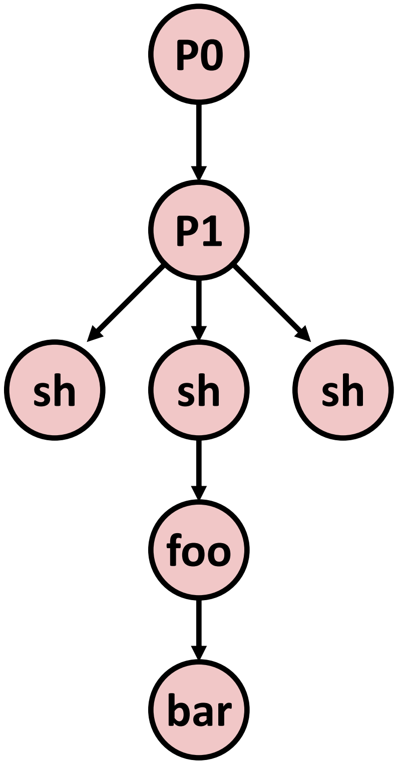
Logical View of Threads

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

Threads associated with process foo



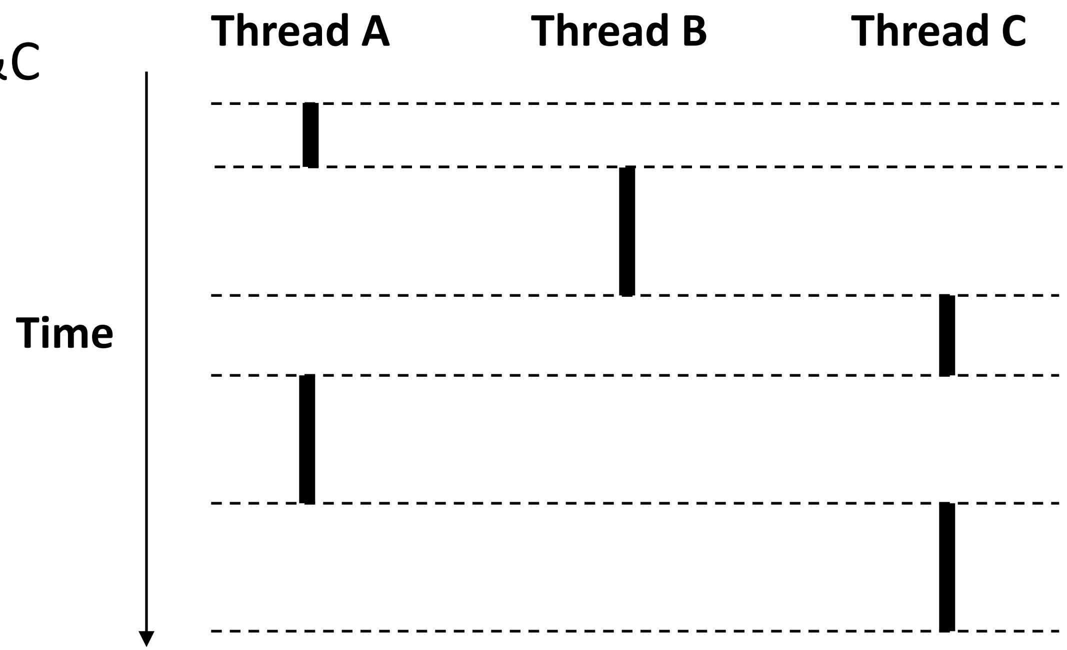
Process hierarchy



Concurrent Threads

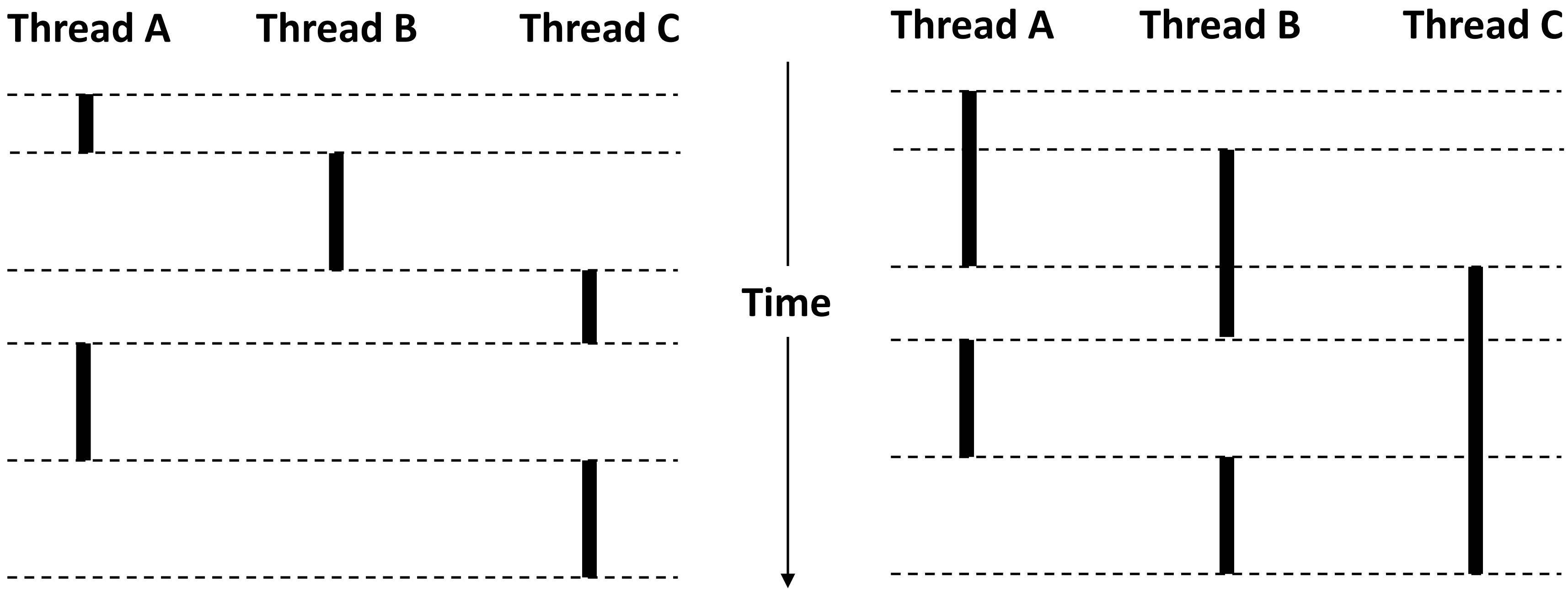
- **Two threads are *concurrent* if their flows overlap in time**
- **Otherwise, they are sequential**

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



Concurrent Thread Execution

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

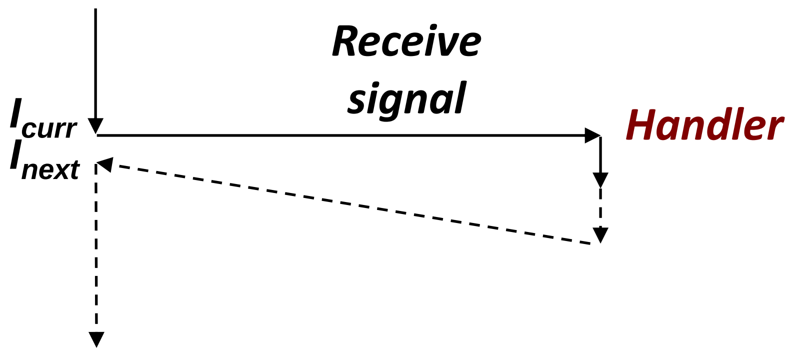


Run 3 threads on 2 cores

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

Threads vs. Signals



- **Signal handler shares state with regular program**
  - Including stack
- **Signal handler interrupts normal program execution**
  - Unexpected procedure call
  - Returns to regular execution stream
  - *Not* a peer
- **Limited forms of synchronization**
  - Main program can block / unblock signals
  - Main program can pause for signal

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`

The Pthreads "hello, world" Program

```c
/*
 * hello.c - Pthreads "hello, world" program
 */
#include "csapp.h"
void *thread(void *vargp);

int main(int argc, char** argv)
{
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    return 0;
}
```

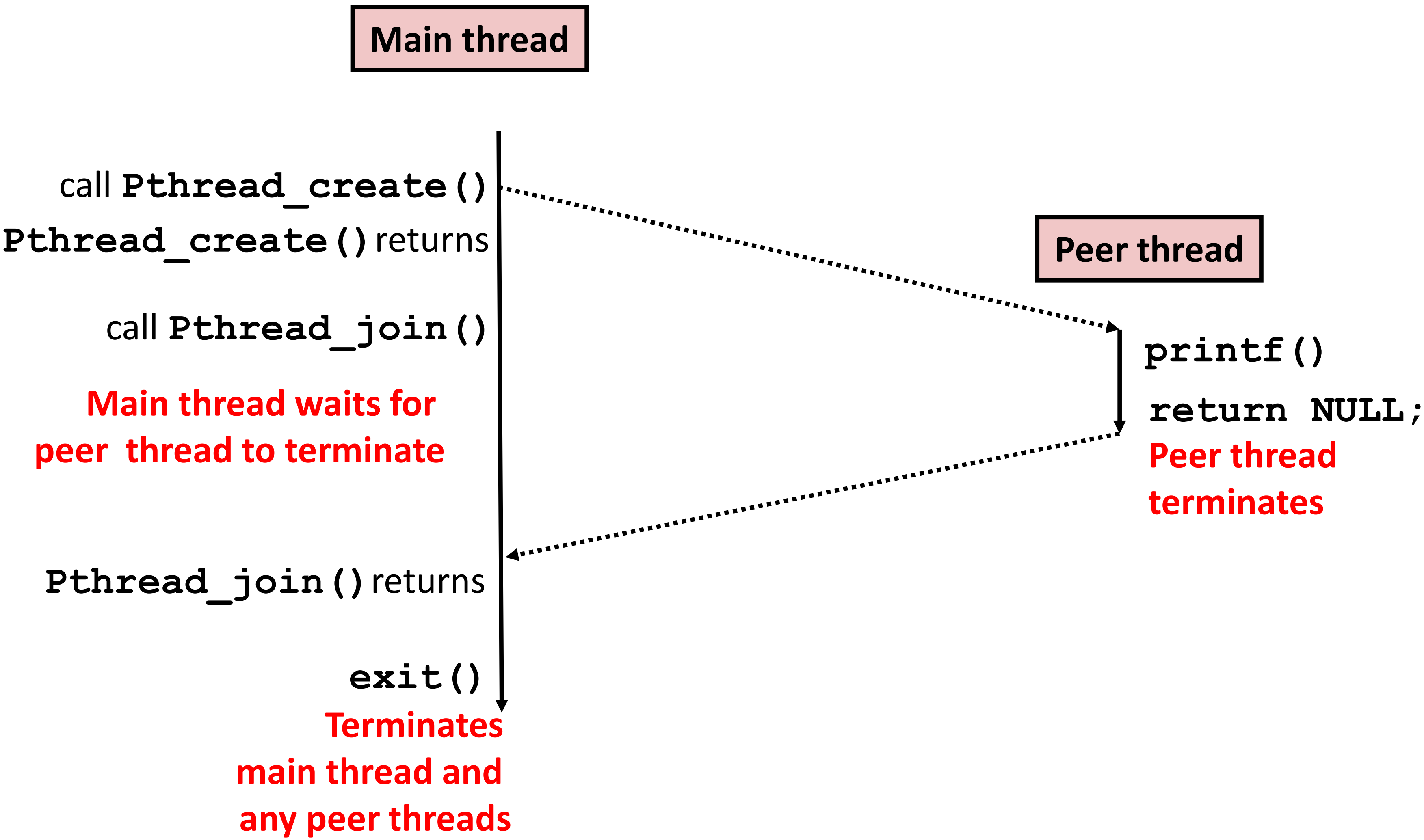
hello.c

- *Thread ID* (&tid)
- *Thread attributes (usually NULL)* (first NULL in Pthread_create)
- *Thread routine* (thread)
- *Thread arguments (void *p)* (last NULL in Pthread_create)
- *Return value (void **p)* (NULL in Pthread_join)

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

hello.c

Execution of Threaded "hello, world"



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!
  - Future lectures

Summary: 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

Outline

- Concurrency
- Concurrency Hazards
- Processes Reminder
- Threads
- **Sharing**
- Reasoning about Sharing
- Mutual Exclusion

What happens here?

Main

```
int i;
for (i = 0; i < 100; i++) {
  Pthread_create(&tid, NULL,
                 thread, &i);
}
```

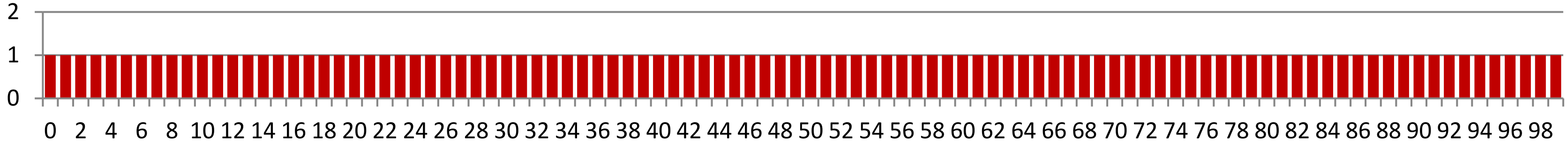
Thread

```
void *thread(void *vargp)
{
  int i = *((int *)vargp);
  Pthread_detach(pthread_self());
  save_value(i);
  return NULL;
}
```

- **Race Test**
  - If no race, then each thread would get different value of **`i`**
  - Set of saved values would consist of one copy each of 0 through 99

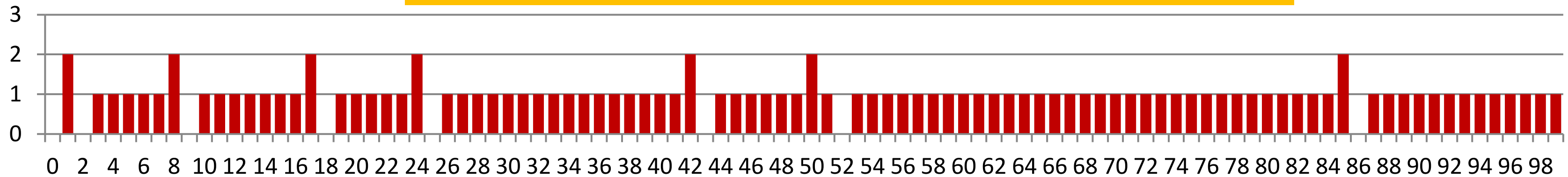
Ut-Oh: Experimental Results

No Race



Single core laptop

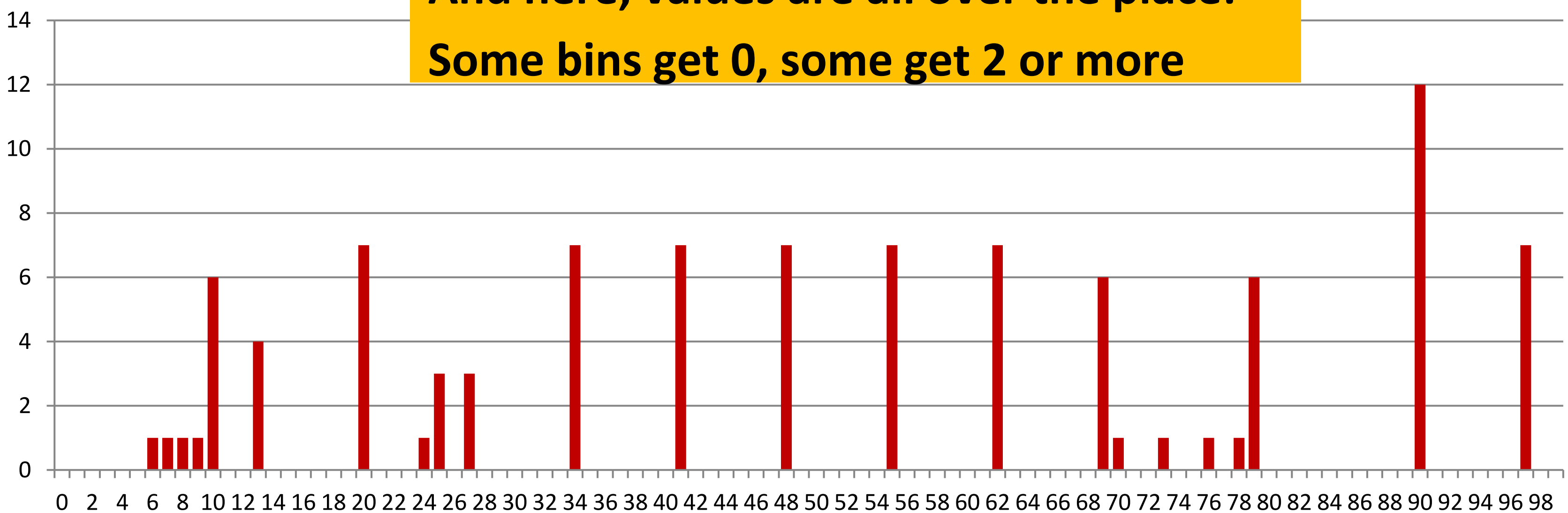
For each "0" there is some later "2" here



Multicore server

And here, values are all over the place:

Some bins get 0, some get 2 or more



Outline

- Concurrency
- Concurrency Hazards
- Processes Reminder
- Threads
- Sharing
- **Reasoning about Sharing**
- Mutual Exclusion

Sharing: A More Involved Example

```c
char **ptr;  /* global var */

int main(int argc, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

```c
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

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

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

A common, but inelegant way to pass a single argument to a thread routine

Mapping Variable Instances to Memory

- **Global variables**
  - *Def:* Variable declared outside of a function
  - **Virtual memory contains exactly one instance of any global variable**

- **Local variables**
  - *Def:* Variable declared inside function without `static` attribute
  - **Each thread stack contains one instance of each local variable**

- **Local static variables**
  - *Def:* Variable declared inside function with the `static` attribute
  - **Virtual memory contains exactly one instance of any local static variable.**

Mapping Variable Instances to Memory

Global var: 1 instance (ptr [data])

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

Notation: instance of msgs in main

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

int main(int main, char *argv[])
{
    long i;
    pthread_t tid;
    char *msgs[2] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < 2; i++)
        Pthread_create(&tid,
            NULL,
            thread,
            (void *)i);
    Pthread_exit(NULL);
}
```

sharing.c

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

```
void *thread(void *vargp)
{
    long myid = (long)vargp;
    static int cnt = 0;

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

Local static var: 1 instance (cnt [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
`cnt`	no	yes	yes
`i.m`	yes	no	no
`msgs.m`	yes	yes	yes
`myid.p0`	no	yes	no
`myid.p1`	no	no	yes

```
char **ptr;  /* global var */
int main(int main, char *argv[]) {
  long i; pthread_t tid;
  char *msgs[2] = {"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)
{
  long myid = (long)vargp;
  static int cnt = 0;

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

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
`cnt`	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`, `cnt`, and `msgs` are shared
  - `i` and `myid` are *not* shared

Synchronizing Threads

- **Shared variables are handy…**
- **…but introduce the possibility of nasty *synchronization* errors.**

badcnt.c: Improper Synchronization

```c
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```c
/* Thread routine */
void *thread(void *vargp)
{
    long j, niters =
                *((long *)vargp);

    for (j = 0; j < niters; j++)
        cnt++;

  return NULL;
}
```

```
linux> ./badcnt 10000
OK cnt=20000
linux> ./badcnt 10000
BOOM! cnt=13051
linux>
```

cnt should equal 20,000.

What went wrong?

Assembly Code for Counter Loop

C code for counter loop in thread i

```
for (j = 0; j < niters; j++)
    cnt++;
```

Asm code for thread i

```
    movq  (%rdi), %rcx
    testq %rcx,%rcx
    jle   .L2
    movl  $0, %eax
.L3:
    movq  cnt(%rip),%rdx
    addq  $1, %rdx
    movq  %rdx, cnt(%rip)
    addq  $1, %rax
    cmpq  %rcx, %rax
    jne   .L3
.L2:
```

H_i : Head

L_i : Load cnt

U_i : Update cnt

S_i : Store cnt

T_i : Tail

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
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	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

Note: One of many possible interleavings

OK

**For now. In reality, on x86 even non-sequentially consistent interleavings are possible*

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
  - $\%rdx_i$ is the content of %rdx in thread i's context

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	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

Thread 1 critical section (green)

Thread 2 critical section (blue)

OK

Concurrent Execution (cont)

- **Incorrect ordering: two threads increment the counter, but the result is 1 instead of 2**

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	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!

(badcnt will print "BOOM!")

Concurrent Execution (cont)

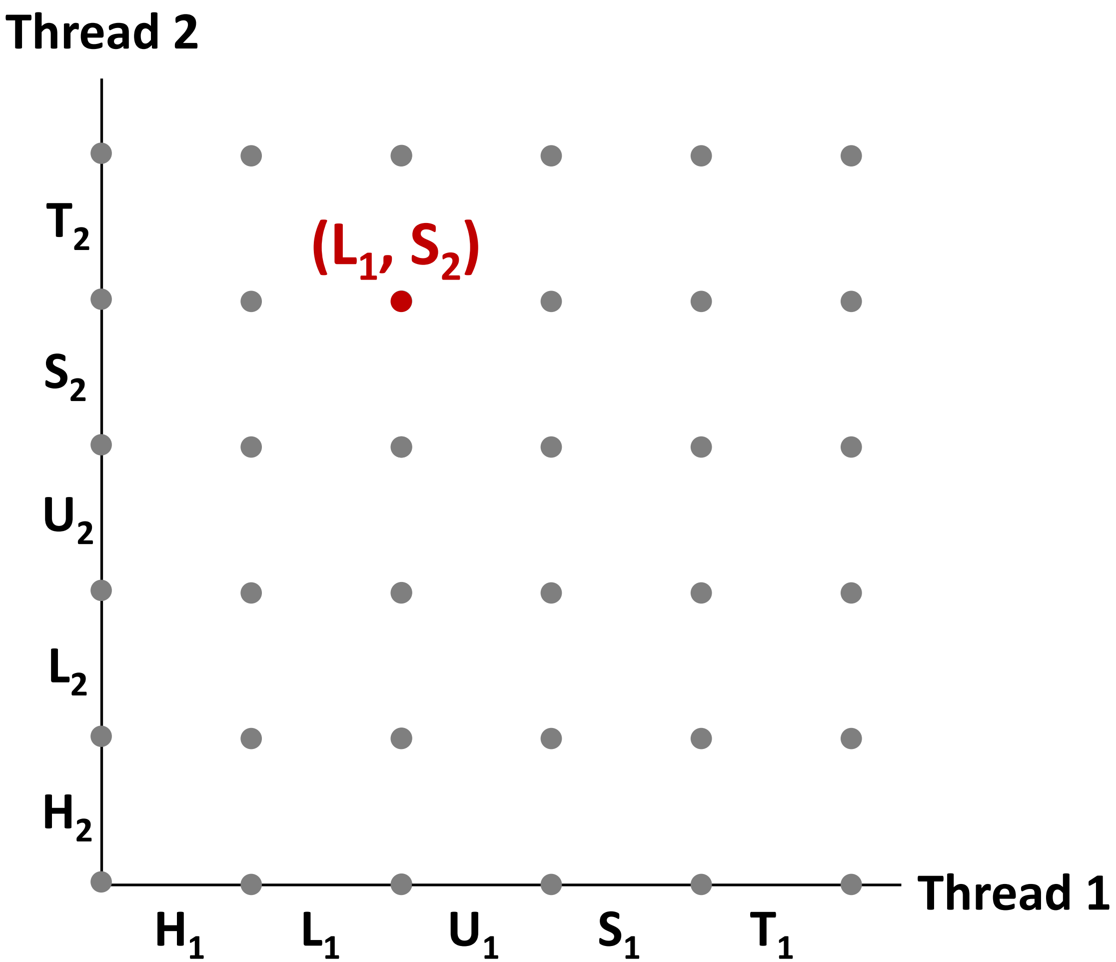
- How about this ordering?

i (thread)	$instr_i$	$\%rdx_1$	$\%rdx_2$	cnt
1	H_1			0
1	L_1	0		
2	H_2			
2	L_2		0	
2	U_2		1	
2	S_2		1	1
1	U_1	1		
1	S_1	1		1
1	T_1			1
2	T_2			1

Oops again!

- We can analyze the behavior using a *progress graph*

Progress Graphs



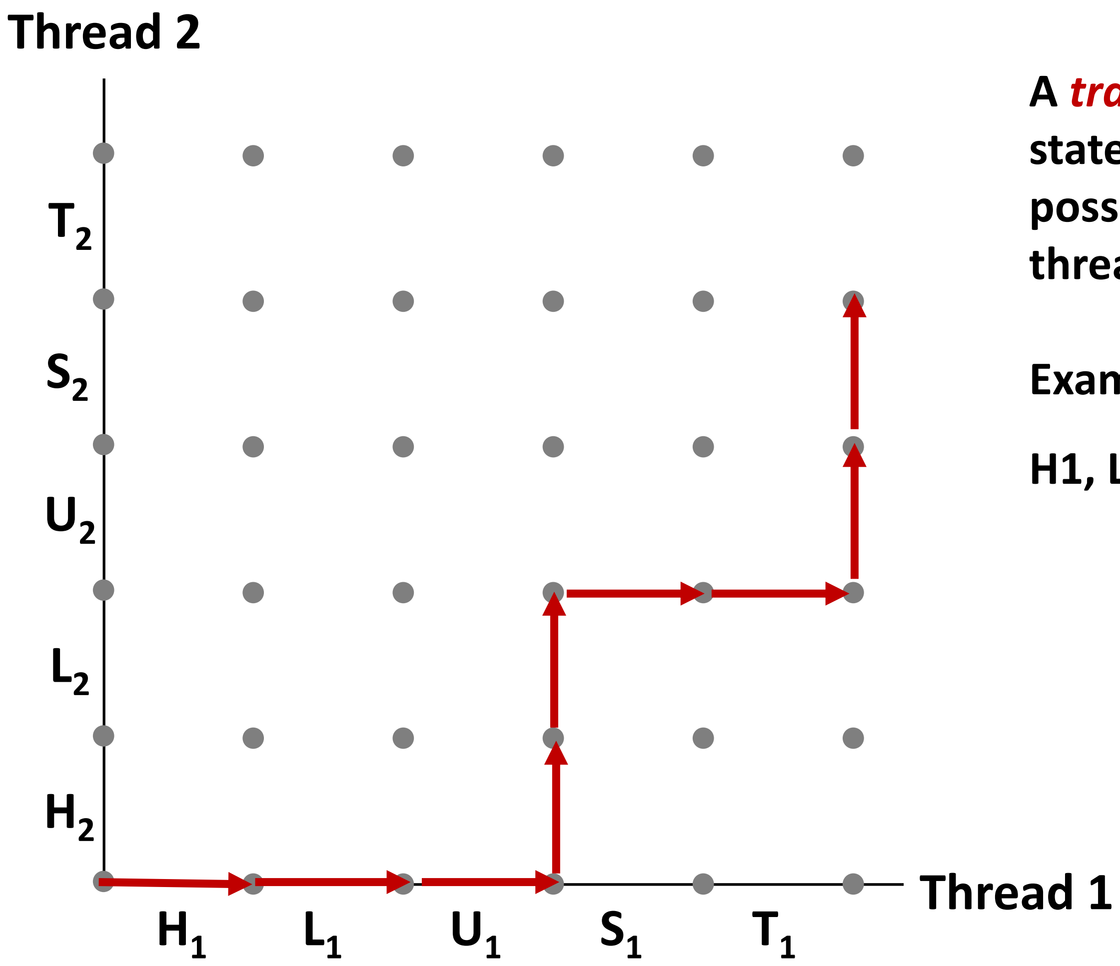
A ***progress graph*** depicts the discrete ***execution state space*** of concurrent threads.

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

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

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

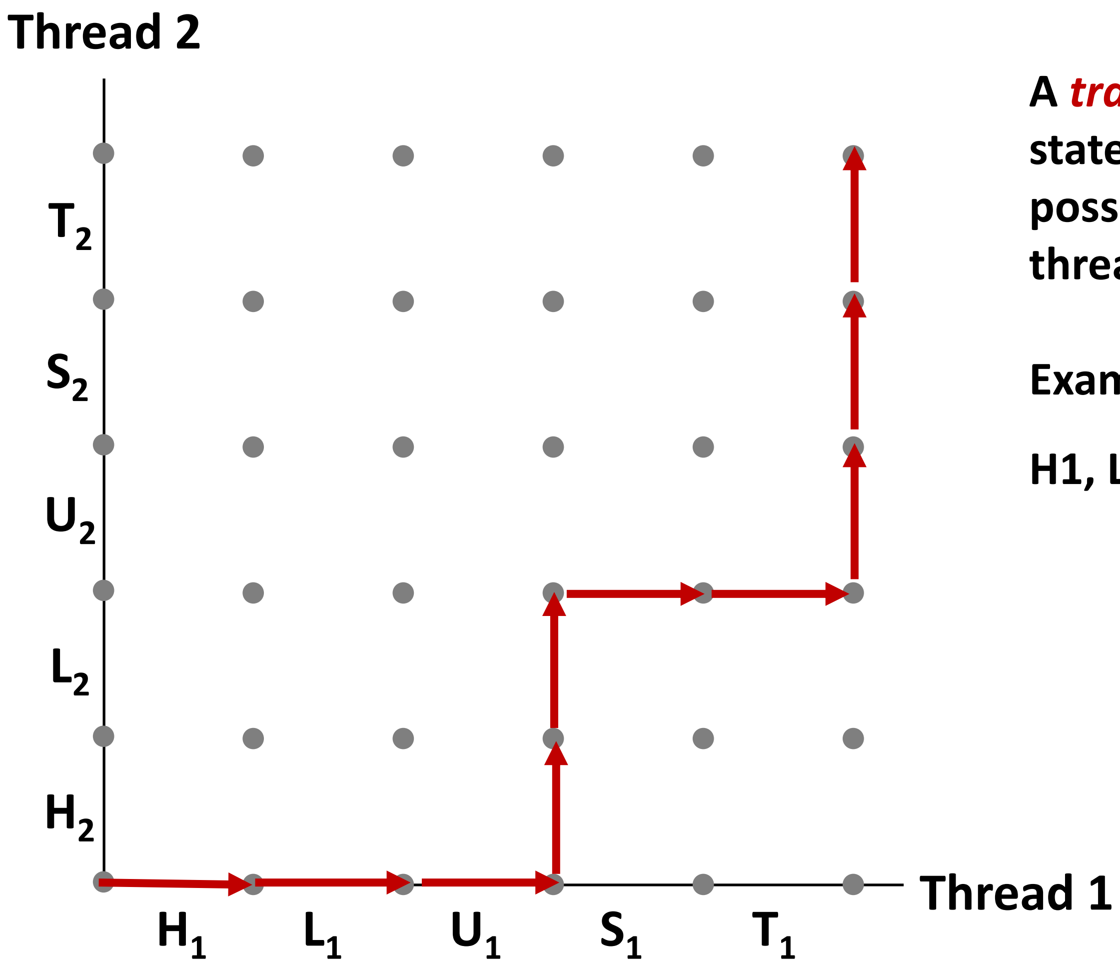


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Trajectories in Progress Graphs

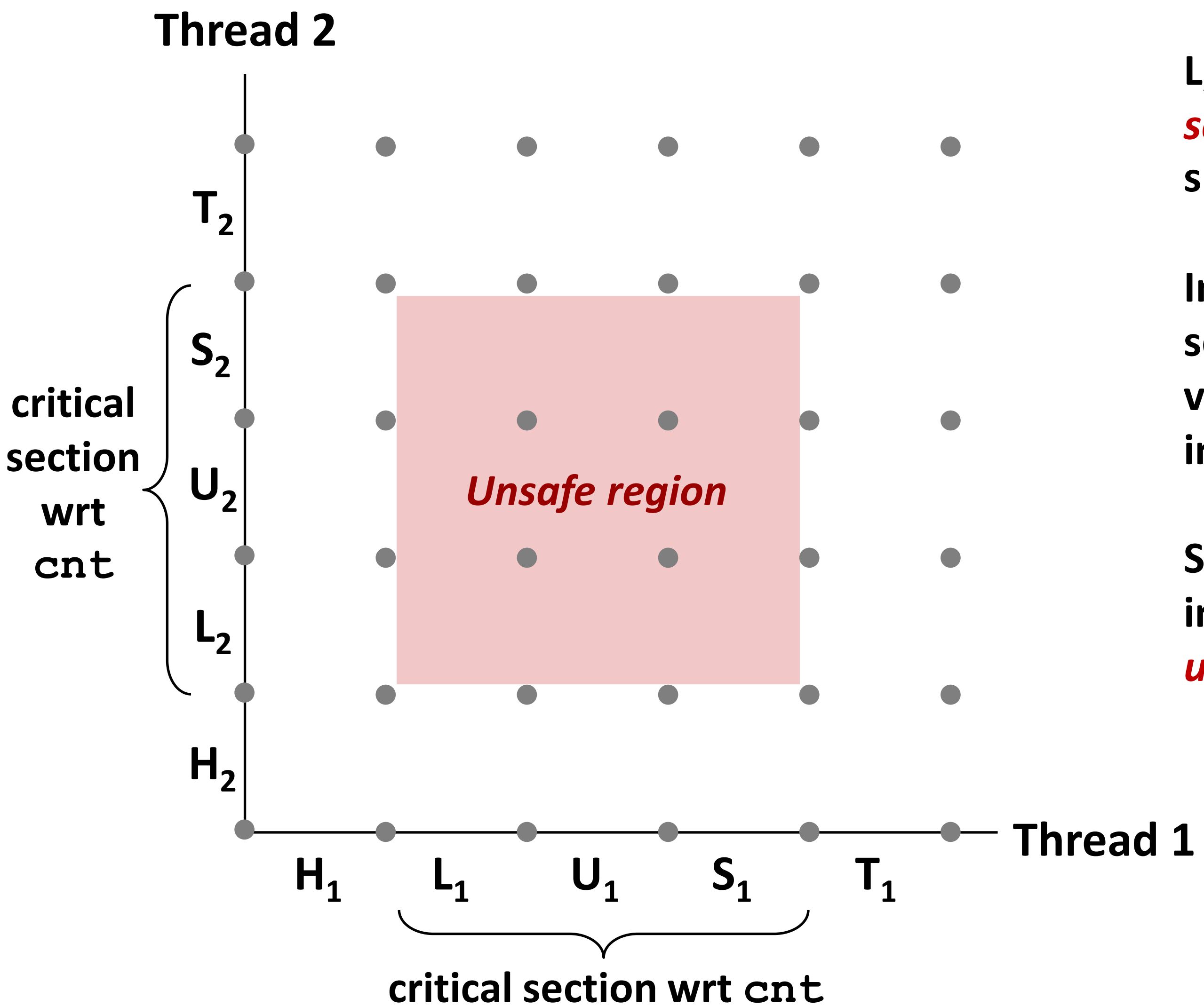


A ***trajectory*** is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Critical Sections and Unsafe Regions

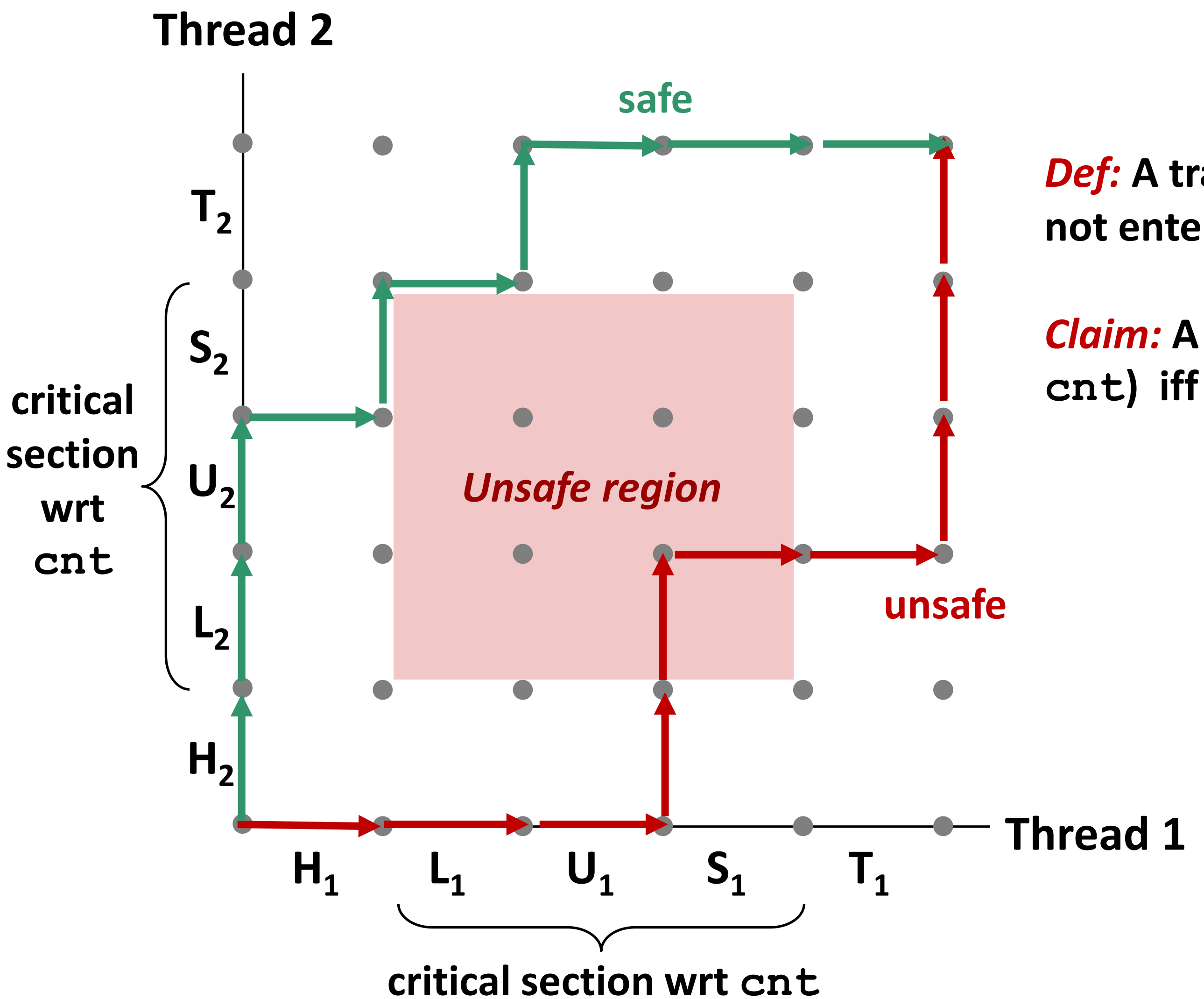


L, U, and S form a ***critical section*** with respect to the shared variable `cnt`

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

Sets of states where such interleaving occurs form ***unsafe regions***

Critical Sections and Unsafe Regions



Def: A trajectory is ***safe*** iff it does not enter any unsafe region

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

badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long j, niters =
              *((long *)vargp);

    for (j = 0; j < niters; j++)
        cnt++;

    return NULL;
}
```

Variable	main	thread1	thread2
cnt	yes*	yes	yes
niters.m	yes	yes	yes
tid1.m	yes	no	no
j.1	no	yes	no
j.2	no	no	yes
niters.1	no	yes	no
niters.2	no	no	yes

Outline

- Concurrency
- Concurrency Hazards
- Processes Reminder
- Threads
- Sharing
- Reasoning about Sharing
- **Disciplining Access and Mutual Exclusion**

Enforcing Mutual Exclusion

- ***Question:*** **How can we guarantee a safe trajectory?**

- **Answer: We must *synchronize* the execution of the threads so that they can never have an unsafe trajectory.**
  - i.e., need to guarantee ***mutually exclusive access*** for each critical section.

- **Classic solution:**
  - Mutex (pthreads)
  - Semaphores (Edsger Dijkstra)

- **Other approaches (out of our scope)**
  - Condition variables (pthreads)
  - Monitors (Java)

MUTual EXclusion (mutex)

- ***Mutex***: **boolean synchronization variable**
- **enum {locked = 0, unlocked = 1}**
- **lock(m)**
  - If the mutex is currently not locked, lock it and return
  - Otherwise, wait (spinning, yielding, etc) and retry
- **unlock(m)**
  - Update the mutex state to unlocked

MUTual EXclusion (mutex)

- *Mutex*: boolean synchronization variable *

- **Swap(*a, b)**

  [t = *a; *a = b; return t;]

  // Notation: what's inside the brackets [ ] is indivisible (a.k.a. atomic)

  // by the magic of hardware / OS

- **Lock(m):**

  while (swap(&m->state, locked) == locked) ;

- **Unlock(m):**

  m->state = unlocked;

**For now. In reality, many other implementations and design choices (c.f., 15-410, 418, etc).*

badcnt.c: Improper Synchronization

```
/* Global shared variable */
volatile long cnt = 0; /* Counter */

int main(int argc, char **argv)
{
    long niters;
    pthread_t tid1, tid2;

    niters = atoi(argv[1]);
    Pthread_create(&tid1, NULL,
        thread, &niters);
    Pthread_create(&tid2, NULL,
        thread, &niters);
    Pthread_join(tid1, NULL);
    Pthread_join(tid2, NULL);

    /* Check result */
    if (cnt != (2 * niters))
        printf("BOOM! cnt=%ld\n", cnt);
    else
        printf("OK cnt=%ld\n", cnt);
    exit(0);
}
```

badcnt.c

```
/* Thread routine */
void *thread(void *vargp)
{
    long j, niters =
              *((long *)vargp);

    for (j = 0; j < niters; j++)
        cnt++;

    return NULL;
}
```

How can we fix this using synchronization?

goodmcnt.c: Mutex Synchronization

- Define and initialize a mutex for the shared variable cnt:

```
volatile long cnt = 0;  /* Counter */
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL); // No special attributes
```

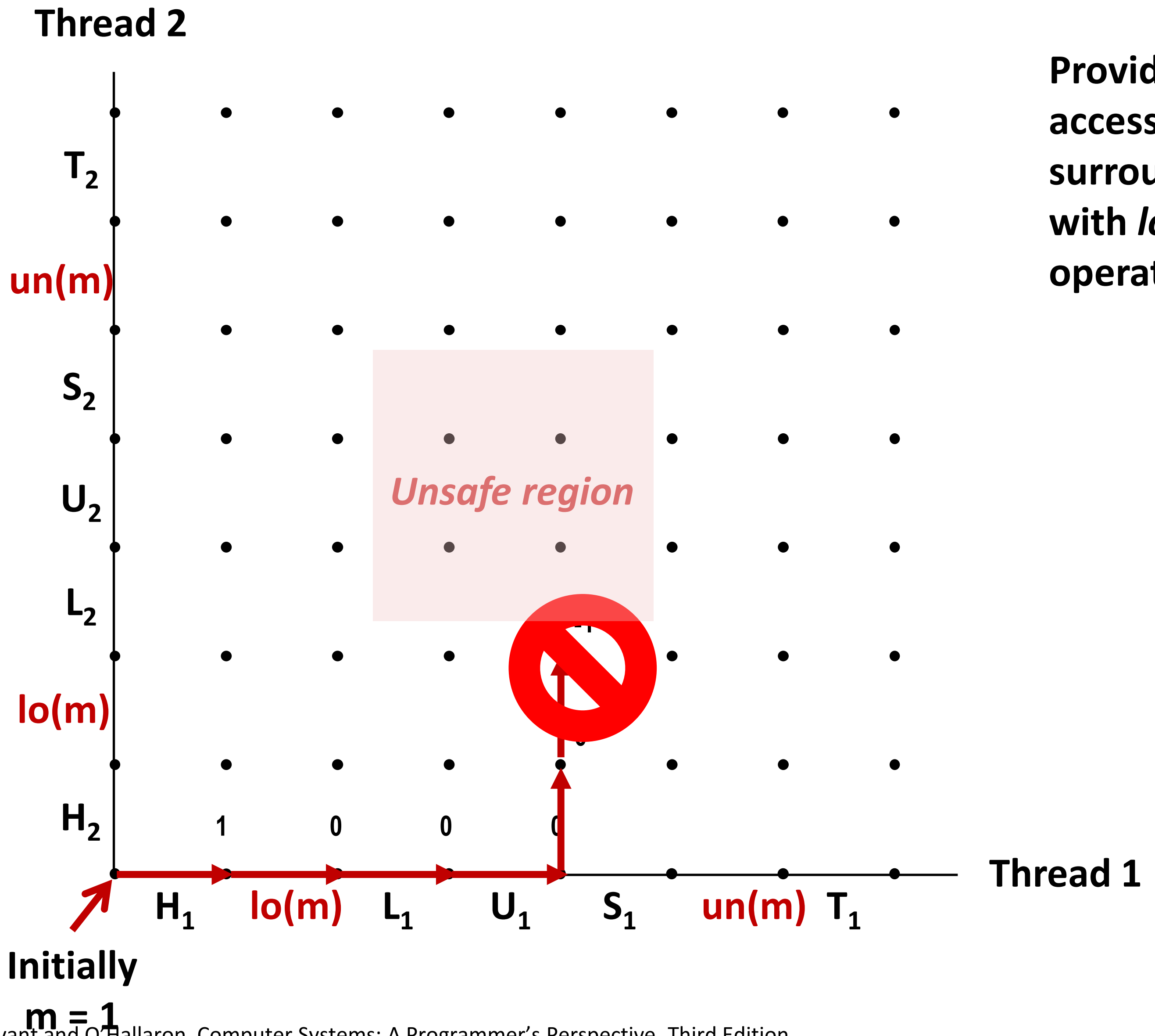
- Surround critical section with *lock* and *unlock*:

```
for (i = 0; i < niters; i++) {
    pthread_mutex_lock(&mutex);
    cnt++;
    pthread_mutex_unlock(&mutex);
}
```

```
linux> ./goodmcnt 10000
OK cnt=20000
linux> ./goodmcnt 10000
OK cnt=20000
```

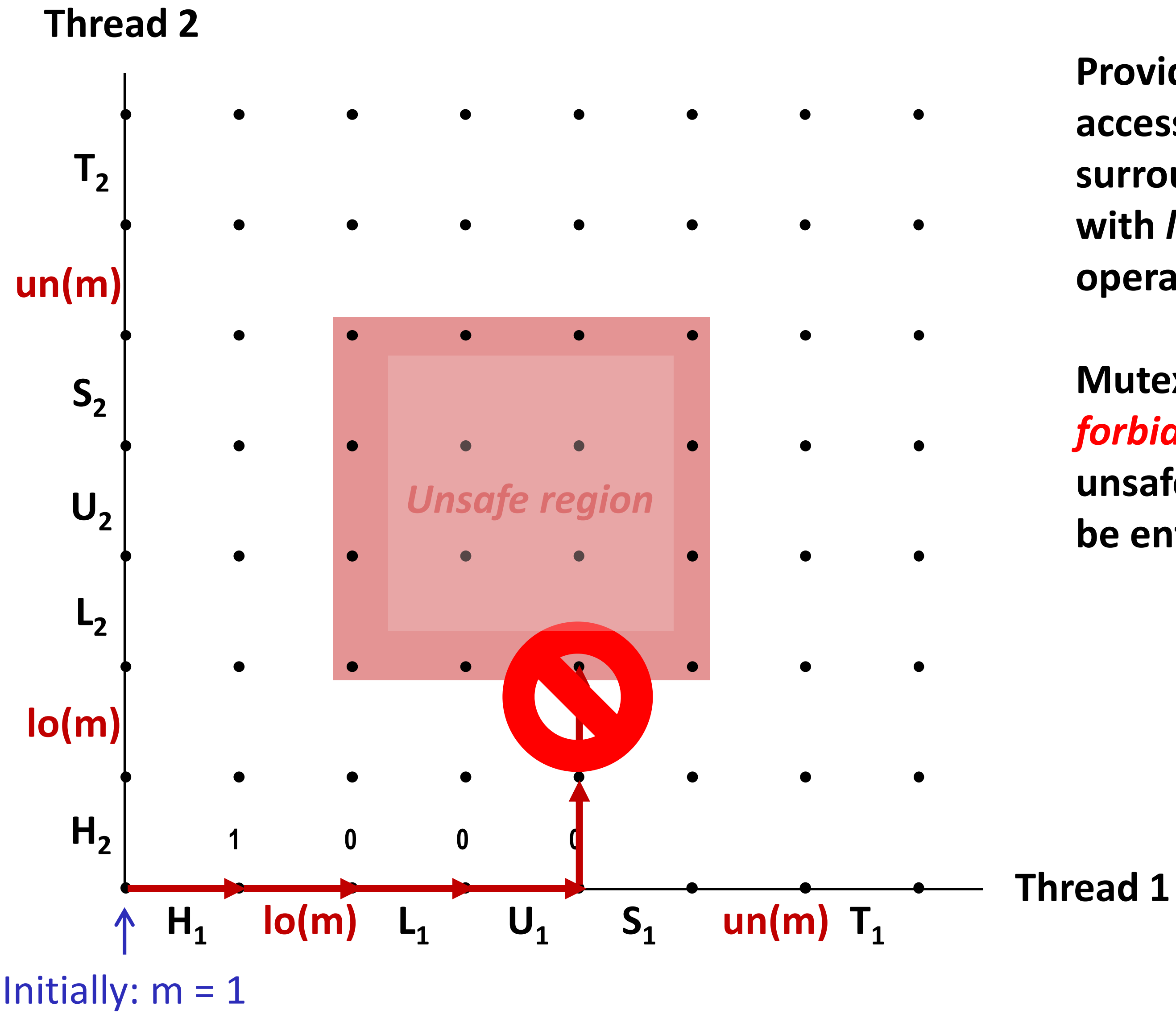
Function	badcnt	goodmcnt
Time (ms) niters = 10^6	12.0	214.0
Slowdown	1.0	17.8

Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

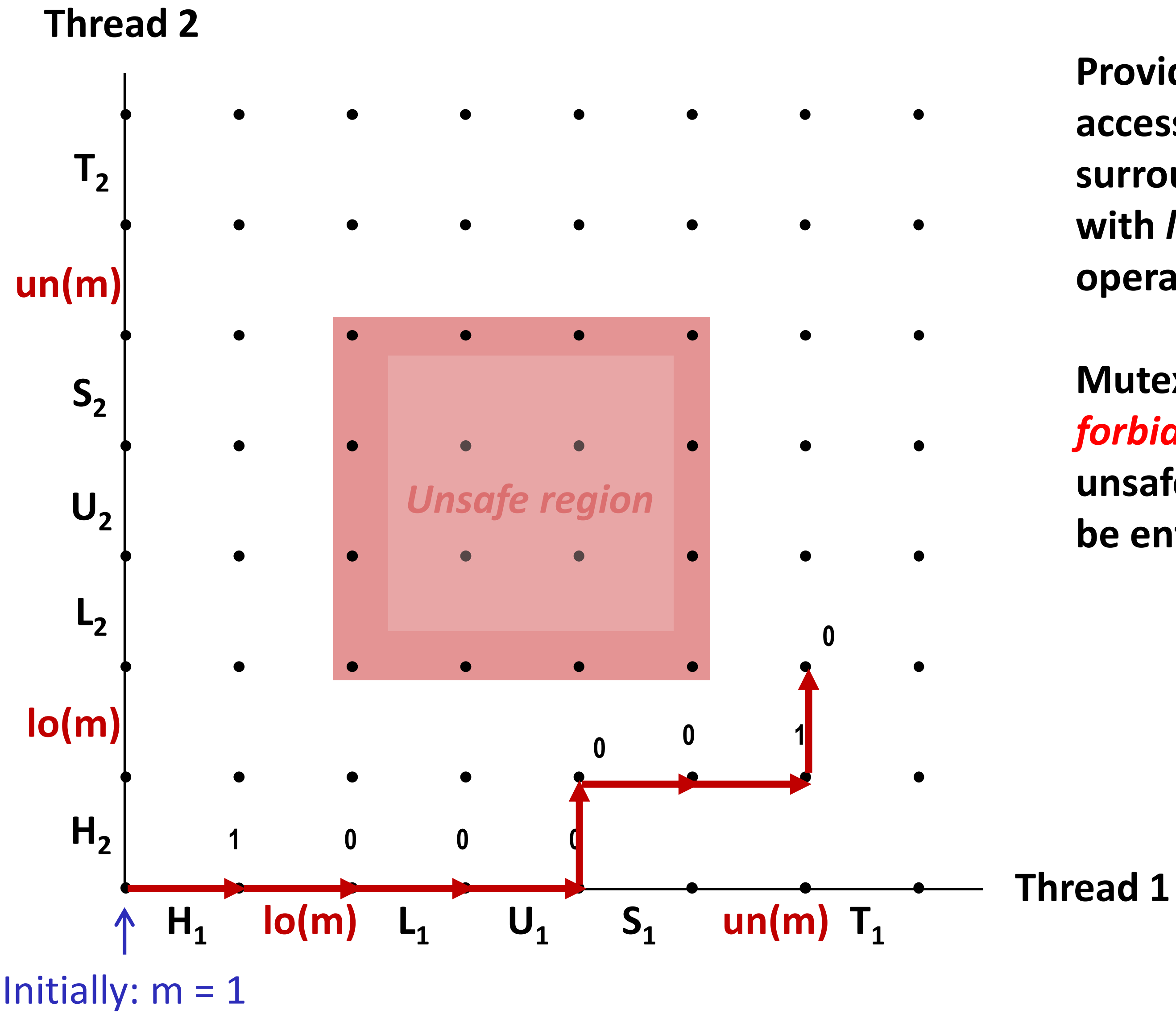
Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

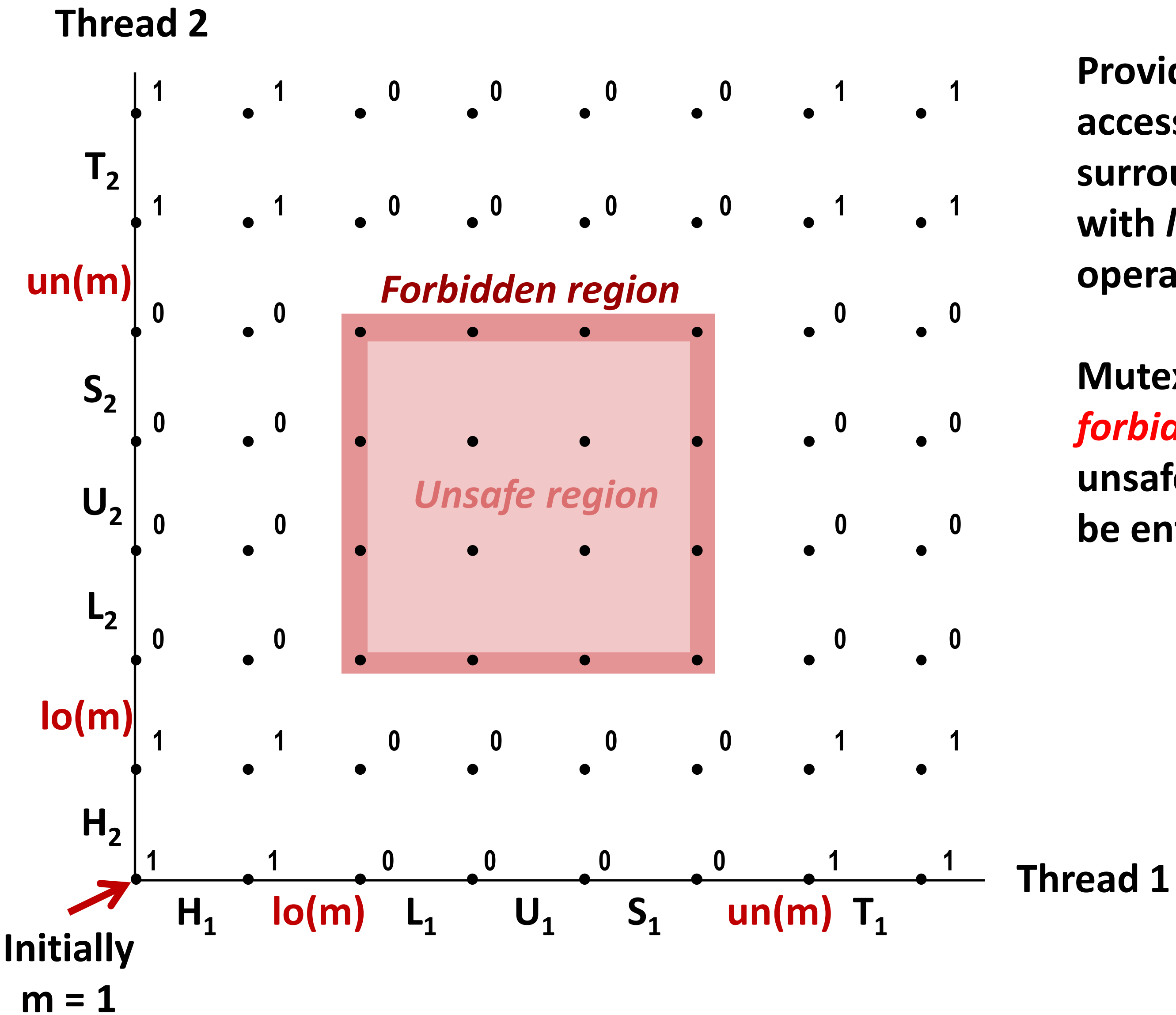
Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

Why Mutexes Work



Provide mutually exclusive access to shared variable by surrounding critical section with *lock* and *unlock* operations

Mutex invariant creates a *forbidden region* that encloses unsafe region and that cannot be entered by any trajectory.

Summary: Managing Races

- Identify the resources
- Identify the shared resources
- Identify the *critical resources*, i.e. the resources that are shared in a way that is not naturally safe
- Discipline the use of the critical resources to ensure that they are used safely
  - Augment the *critical sections* of code i.e. the code that makes otherwise unsafe use of the critical resources to enforce the safe discipline.
  - Mutual exclusion, a.k.a. “At most one (concurrent user)” is a very common discipline that is straight-forward to enforce

Aside/Extra: Multiplexed Event Processing

- **Concurrency can also be managed by taking explicit control over the scheduling and avoiding bad schedules**
  - This approach does not require a new abstraction for work, i.e. it doesn't require threads, etc.
  - It is "old school", but still used in microcontrollers and other austere environments without threads.
- **Server maintains set of active fd connections**
  - Array of connfd's
- **Loop:**
  - Determine which descriptors (connfd's or listenfd) have pending inputs
    - e.g., using `select` function
    - arrival of pending input is an *event*
  - If listenfd has input, then `accept` connection and add new connfd to array
  - Service all connfd's with pending inputs
- **Details for select-based server in book**

I/O Multiplexed Event Processing