CS3281 / CS5281

Synchronization Basics

CS3281 / CS5281

Spring 2024

**Some lecture slides borrowed and adapted from CMU's "Computer Systems: A Programmer's Perspective"*

Review

- Threads enable concurrency in an application
  - Example: update a progress bar while simultaneously doing background work
- Processes also enable concurrency
  - But independent processes do not share an address space
- Linux implements threads as processes that share certain resources
  - Threads of the same process share almost everything except stacks

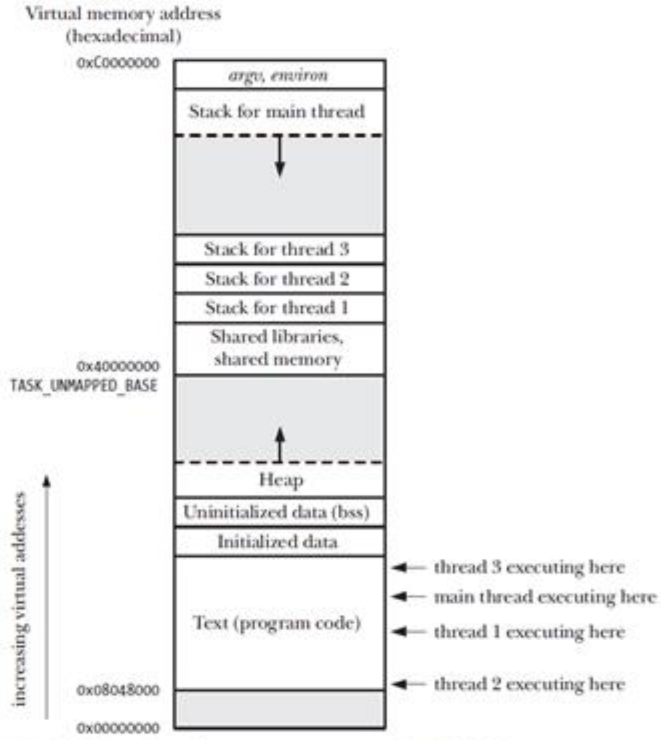


Figure 29-1: Four threads executing in a process (Linux/x86-32)

POSIX Thread API

- Linux exposes threads via the POSIX thread, or pthread, API
- Basic functions:
  - pthread_create() -- create a thread
  - pthread_join() -- join (“wait for”) a terminated thread
  - pthread_exit() -- terminate the calling thread (does not cause the whole program to terminate)
- Other functions:
  - pthread_attr_init() -- specify “attributes” for the thread, e.g., stack size, is the thread detached
  - pthread_self() -- get a “handle” to a pthread
  - pthread_cancel() -- cancel a thread
  - pthread_kill() -- send a signal to a thread
  - pthread_detach() -- detach a thread: automatically free its resources when it’s done
  - pthread_equal() -- compare two threads for equality

Items not Shared Between Threads

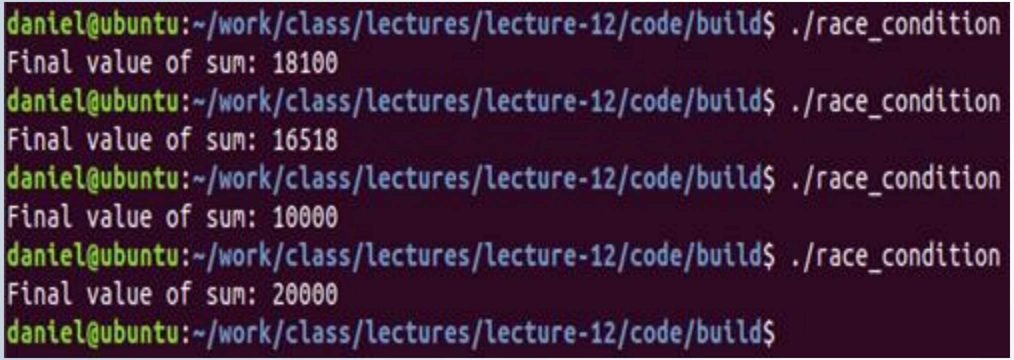
- Threads do not share their stacks
- They also do not share:
  - thread ID
  - signal mask: set of signals whose delivery is currently blocked
  - thread-specific data: allows function to have separate data for each thread
  - alternate signal stack (sigaltstack()): a location to use for a signal handler's stack frame
  - the errno variable: global integer variable that identifies error when a system call fails
  - floating-point environment (see fenv(3)): how floating point rounding/exceptions are handled
  - real-time scheduling policy and priority
  - CPU affinity (Linux-specific): which CPU (or core) thread executes on
  - capabilities (Linux-specific): allow processes to perform *some* privileged operations

Race Conditions

- Race condition: a situation where the result produced by multiple threads (or processes) operating on shared resources depends in an unexpected way on the relative order in which the processes gain access to the CPU(s)
- What causes race conditions?
  - In a nutshell: non-deterministic scheduling and interleaving of thread/process execution

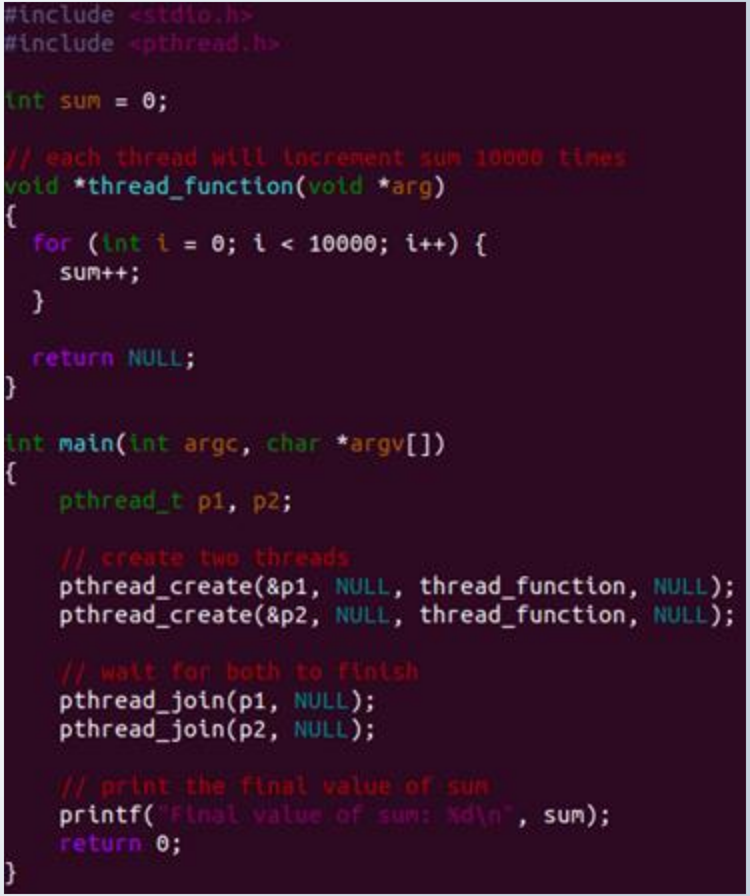
Simple Example to Demonstrate

- Consider race_condition.c:
- Running it yields different results each time:



```
daniel@ubuntu:~/work/class/lectures/lecture-12/code/build$ ./race_condition
Final value of sum: 18100
daniel@ubuntu:~/work/class/lectures/lecture-12/code/build$ ./race_condition
Final value of sum: 16518
daniel@ubuntu:~/work/class/lectures/lecture-12/code/build$ ./race_condition
Final value of sum: 10000
daniel@ubuntu:~/work/class/lectures/lecture-12/code/build$ ./race_condition
Final value of sum: 20000
daniel@ubuntu:~/work/class/lectures/lecture-12/code/build$
```

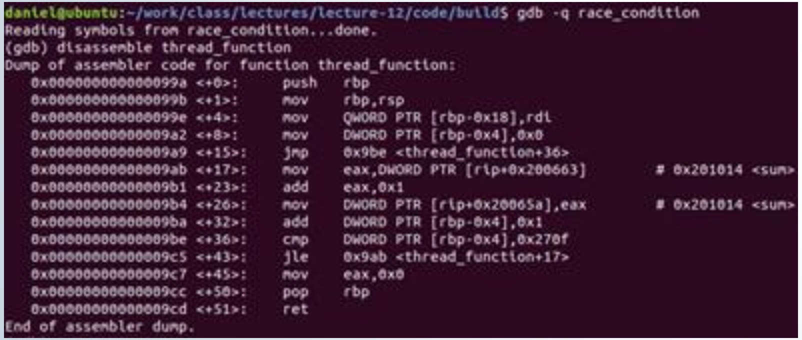
- What causes this behavior?



```
#include <stdio.h>
#include <pthread.h>

int sum = 0;

// each thread will increment sum 10000 times
void *thread_function(void *arg)
{
  for (int i = 0; i < 10000; i++) {
    sum++;
  }

  return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;

    // create two threads
    pthread_create(&p1, NULL, thread_function, NULL);
    pthread_create(&p2, NULL, thread_function, NULL);

    // wait for both to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    // print the final value of sum
    printf("Final value of sum: %d\n", sum);
    return 0;
}
```

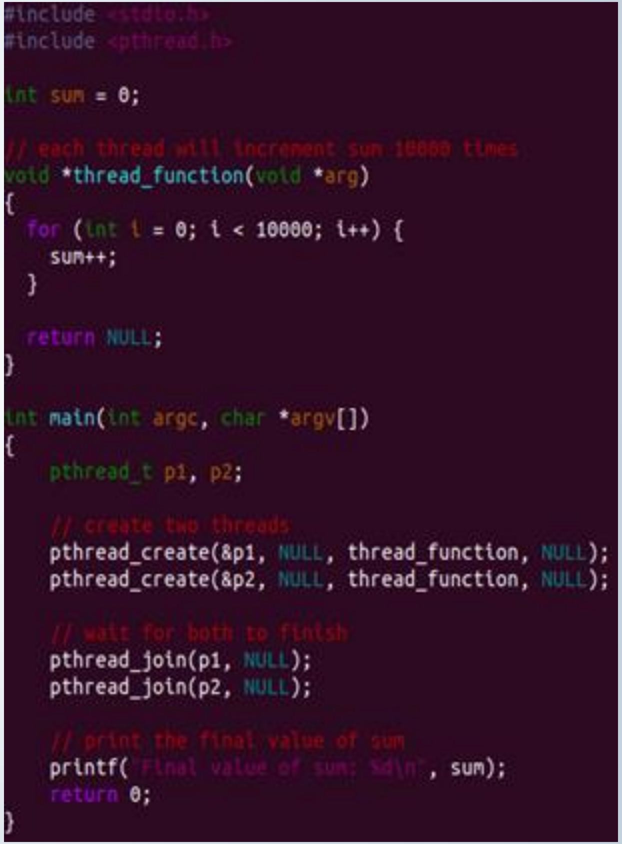
Simple Example (cont.)

- This race condition is caused by the fact that sum++ is implemented as three non-atomic instructions:
  - `mov eax, DWORD PTR [rip+0x200663]`
  - `add eax, 0x1`
  - `mov DWORD PTR [rip+0x20065a], eax`



```
daniel@ubuntu:~/work/class/lectures/lecture-12/code/build$ gdb -q race_condition
Reading symbols from race_condition...done.
(gdb) disassemble thread_function
Dump of assembler code for function thread_function:
   0x000000000000099a <+0>:     push   rbp
   0x000000000000099b <+1>:     mov    rbp,rsp
   0x000000000000099e <+4>:     mov    QWORD PTR [rbp-0x18],rdi
   0x00000000000009a2 <+8>:     mov    DWORD PTR [rbp-0x4],0x0
   0x00000000000009a9 <+15>:    jmp    0x9be <thread_function+36>
   0x00000000000009ab <+17>:    mov    eax,DWORD PTR [rip+0x200663]        # 0x201014 <sum>
   0x00000000000009b1 <+23>:    add    eax,0x1
   0x00000000000009b4 <+26>:    mov    DWORD PTR [rip+0x20065a],eax        # 0x201014 <sum>
   0x00000000000009ba <+32>:    add    DWORD PTR [rbp-0x4],0x1
   0x00000000000009be <+36>:    cmp    DWORD PTR [rbp-0x4],0x270f
   0x00000000000009c5 <+43>:    jle    0x9ab <thread_function+17>
   0x00000000000009c7 <+45>:    mov    eax,0x0
   0x00000000000009cc <+50>:    pop    rbp
   0x00000000000009cd <+51>:    ret
End of assembler dump.
```

- Key point: the OS can switch to a different process after any of these instructions!



```c
#include <stdio.h>
#include <pthread.h>

int sum = 0;

// each thread will increment sum 10000 times
void *thread_function(void *arg)
{
  for (int i = 0; i < 10000; i++) {
    sum++;
  }

  return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t p1, p2;

    // create two threads
    pthread_create(&p1, NULL, thread_function, NULL);
    pthread_create(&p2, NULL, thread_function, NULL);

    // wait for both to finish
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);

    // print the final value of sum
    printf("Final value of sum: %d\n", sum);
    return 0;
}
```

Locks

- The typical way to prevent race conditions is to use *locks*. The idea is:
  - Obtain lock
  - Perform critical section
  - Release lock
- Question: how do we build locks?
  - We need at least two operations:
    - Obtain lock
    - Release lock
- How do we implement these operations?

Implementing Locks

- Simple implementation: use an integer variable to represent the lock
  - 0: lock is free
  - 1: lock is taken
- Releasing the lock is simple
  - Set the value to 0! Guaranteed to be atomic (by the architecture)
- But what about obtaining the lock?
  - Check if the current value is 0
  - If current value is 0, then set it to 1
  - But that’s at least two atomic operations!

Atomic Instructions to the Rescue

- Modern hardware has special instructions as the building blocks for locks. Many general kinds:
  - Fetch and op: e.g., fetch memory and increment it
  - Exchange/Test and Set: set the value of memory and return the old value
  - Compare and swap: check value and swap if it matches and return if swapped
  - Load Reserve/Store Conditional: two operations used together to ensure atomicity
- Special instructions implement the corresponding logic *atomically. e.g.,*
  - x86: xchg, cmpxchg, xadd, bts, …
  - RISC-V: AMOSWAP, LR/SC
- Often these operations are invoked through compiler primitives rather than assembly instructions
- Let's use these to build a *spin-lock*

Using xchg() to Build a Spin Lock

- A spin-lock is a lock that just keeps trying to obtain the lock until it succeeds
- Let’s use the xchg instruction to build a spin-lock:

```
void init(int *lock) {
  // 0 = free, 1 = taken
  *lock = 0;
}

void lock(int *lock) {
  while (xchg(lock, 1) == 1)
    ; // spin-wait (do nothing)
}

void unlock(int *lock) {
  *lock = 0;
}
```

// xchg does two things atomically:
// 1. Set the value of *lock to 1
// 2. Returns the previous value of *lock

Critical Sections

- More complex operations can’t be implemented by a single atomic operation
- Need way to enforce a piece of code being functionally atomic
- Critical section: piece of code that accesses a shared resource and whose execution should be atomic
  - In other words, it shouldn’t be concurrent with another thread that also accesses the resource
- Must be careful that multiple threads don’t perform simultaneous updates

Shared Variables in Threaded C Programs

- Question: Which variables in a threaded C program are shared?
  - The answer is not as simple as "*global variables are shared*" and "*stack variables are private*"

- *Def:* A variable `x` is *shared* if and only if multiple threads reference some instance of `x`

- Requires answers to the following questions:
  - What is the memory model for threads?
  - How are instances of variables mapped to memory?
  - How many threads might reference each of these instances?

Threaded 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, PC, condition codes, and GP 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:
  - Register values are truly separate and protected, but…
  - Any thread can read and write the stack of any other thread

The mismatch between the conceptual and operation model is a source of confusion and errors

Example Program to Illustrate Sharing

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

int main()
{
    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;
}
```

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

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 automatic 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

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

int main()
{
    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

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

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

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

```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;
}
```

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`			
`cnt`			
`i.m`			
`msgs.m`			
`myid.p0`			
`myid.p1`			

- Answer: A variable `x` is shared iff some instance of `x` is referenced by more than one thread. 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.

Improper Synchronization: badcnt.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);
}
```

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

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

Asm code for thread i

```
      movq   (%rdi), %rcx
      testq  %rcx,%rcx
      jle    .L2
      movl   $0, %eax
```

H_i : Head

```
.L3:
      movq   cnt(%rip),%rdx
      addq   $1, %rdx
      movq   %rdx, cnt(%rip)
```

L_i : Load `cnt`

U_i : Update `cnt`

S_i : Store `cnt`

```
      addq   $1, %rax
      cmpq   %rcx, %rax
      jne    .L3
.L2:
```

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

🟩 Thread 1 critical section

🟥 Thread 2 critical section

OK

Concurrent Execution (cont.)

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

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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!

Concurrent Execution (cont.)

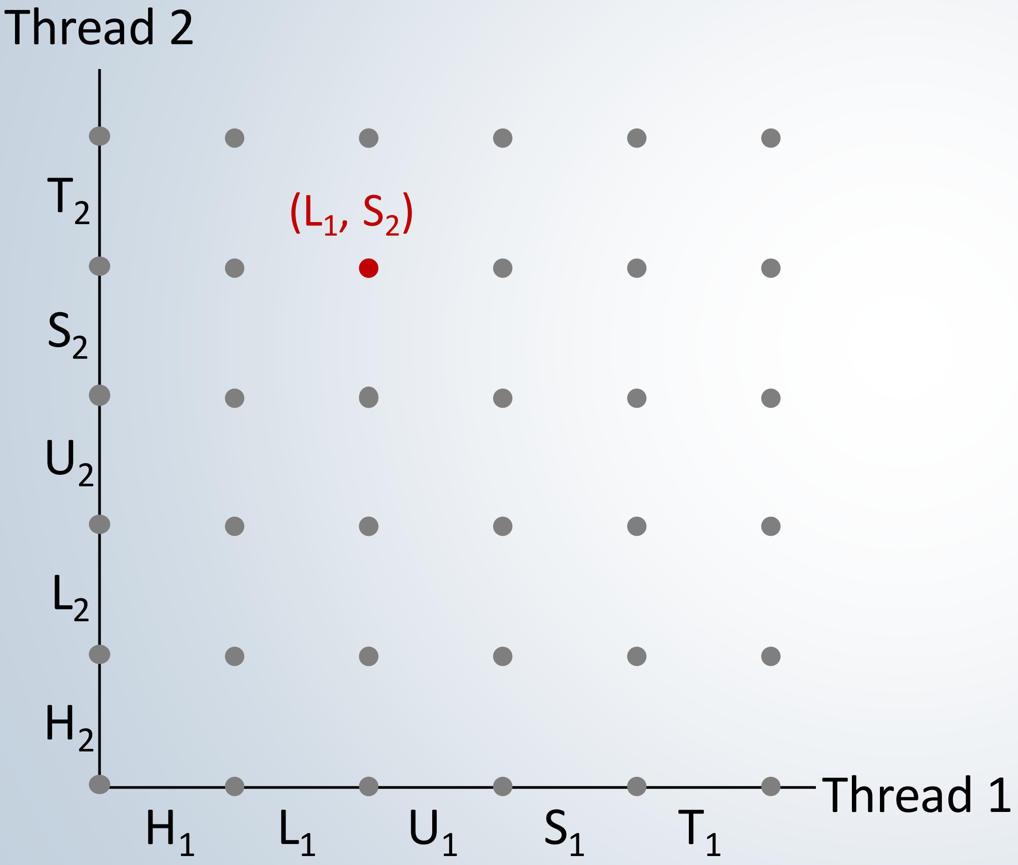
- How about this ordering?

i (thread)	instr_i	$\%\text{rdx}_1$	$\%\text{rdx}_2$	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		1
2	T_2		1	1

Oops!

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

Progress Graphs



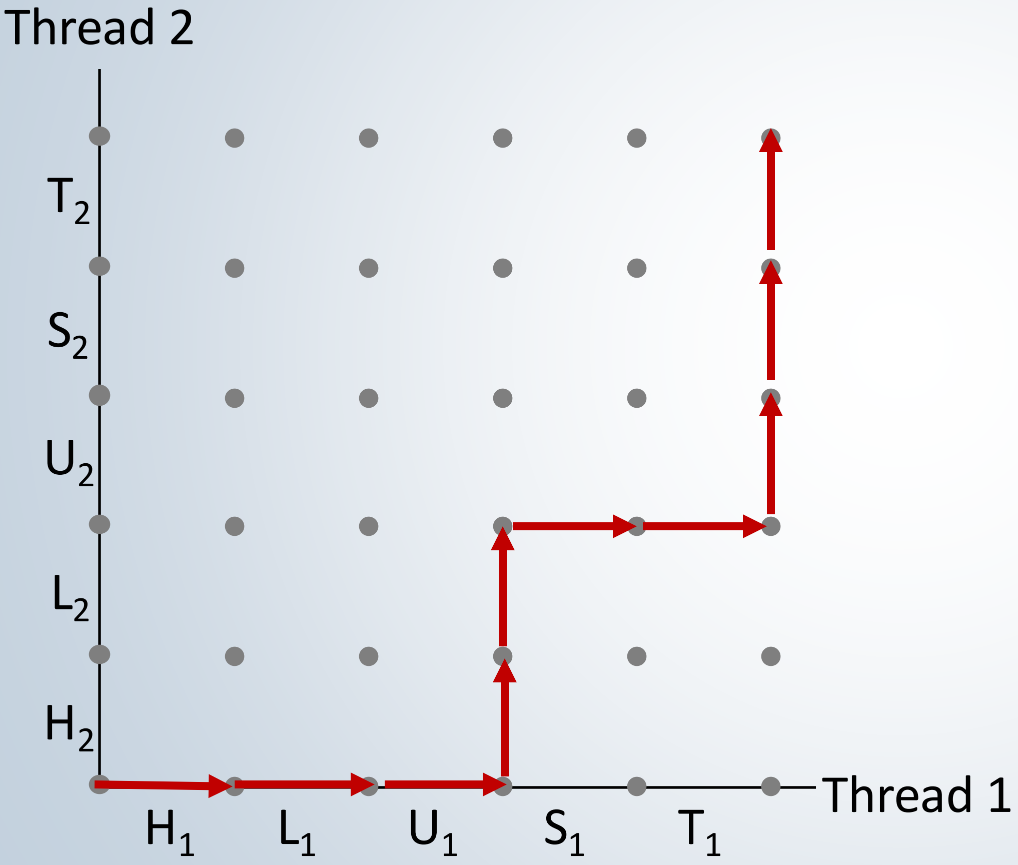
A *progress graph* depicts the discrete *execution state space* of concurrent threads.

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

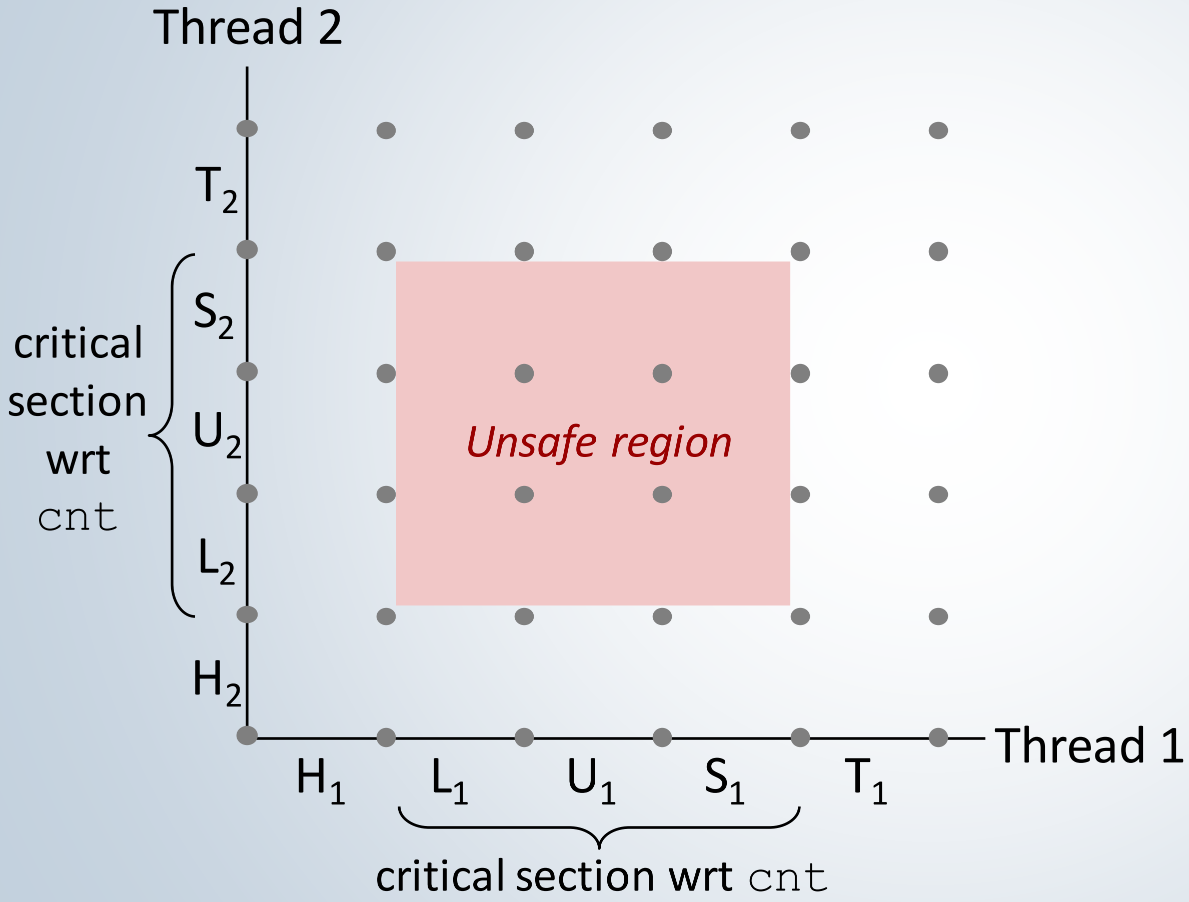


A *trajectory* is a sequence of legal state transitions that describes one possible concurrent execution of the threads.

Example:

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

Critical Sections and Unsafe Regions

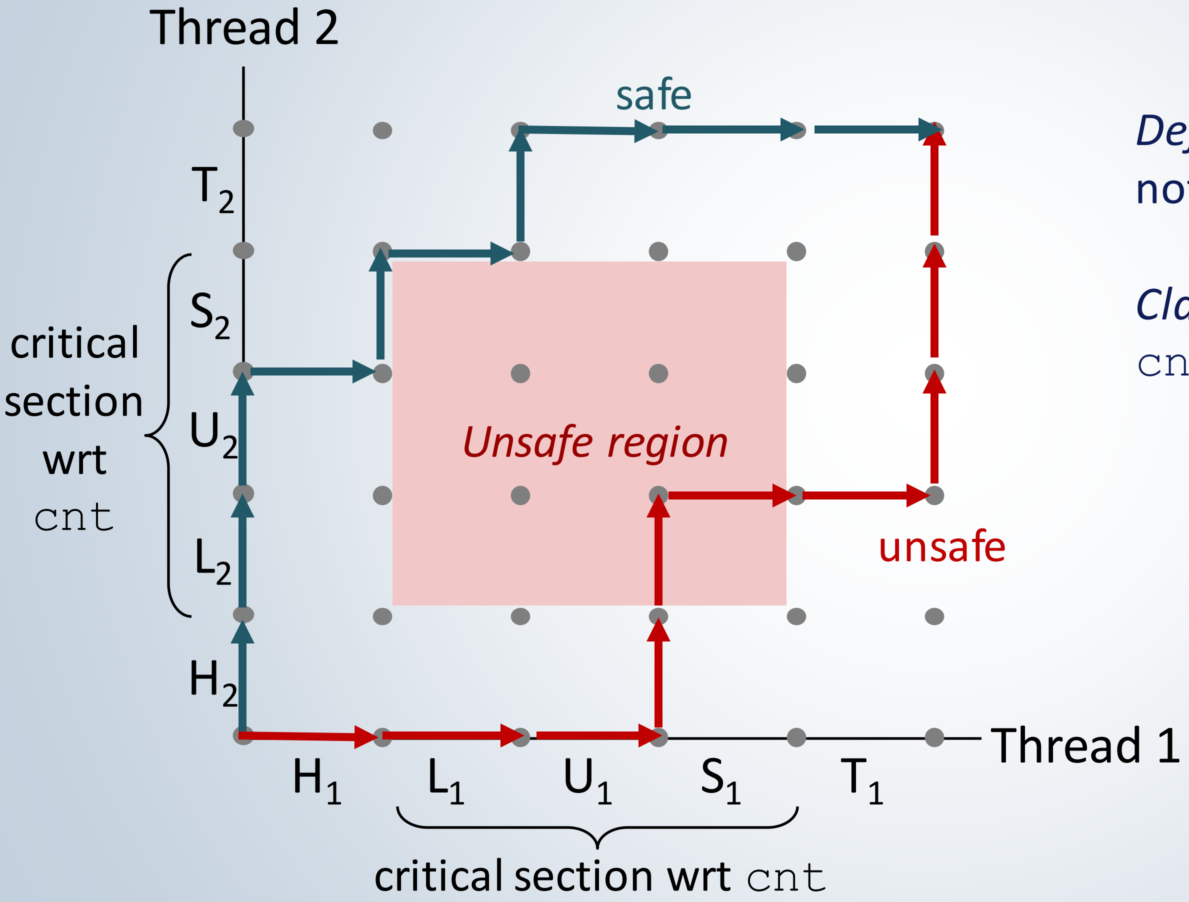


L, U, and S form a *critical section* with respect to the shared variable `cnt`

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

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:
  - Semaphores (Edsger Dijkstra)
- Other approaches
  - Mutex and condition variables (Pthreads)
  - Monitors (Java)

Semaphores

- ***Semaphore:*** non-negative global integer synchronization variable. Manipulated by *P* and *V* operations.
- P(s)
  - If *s* is nonzero, then decrement *s* by 1 and return immediately.
    - Test and decrement operations occur atomically (indivisibly)
  - If *s* is zero, then suspend thread until *s* becomes nonzero and the thread is restarted by a V operation.
  - After restarting, the P operation decrements *s* and returns control to the caller.
- ***V(s):***
  - Increment *s* by 1.
    - Increment operation occurs atomically
  - If there are any threads blocked in a P operation waiting for *s* to become non-zero, then restart exactly one of those threads, which then completes its P operation by decrementing *s*.

- Semaphore invariant: *(s >= 0)*

C Semaphore Operations

Pthreads functions:

```
#include <semaphore.h>

int sem_init(sem_t *s, 0, unsigned int val);} /* s = val */

int sem_wait(sem_t *s);  /* P(s) */
int sem_post(sem_t *s);  /* V(s) */
```

CS:APP wrapper functions:

```
#include "csapp.h”

void P(sem_t *s); /* Wrapper function for sem_wait */
void V(sem_t *s); /* Wrapper function for sem_post */
```

Improper Synchronization: badcnt.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);
}
```

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

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

   return NULL;
}
```

How can we fix this using semaphores?

Using Semaphores for Mutual Exclusion

- Basic idea:
  - Associate a unique semaphore *mutex*, initially 1, with each shared variable (or related set of shared variables).
  - Surround corresponding critical sections with *P(mutex)* and *V(mutex)* operations.

- Terminology:
  - *Binary semaphore*: semaphore whose value is always 0 or 1
  - *Mutex:* binary semaphore used for mutual exclusion
    - P operation: "locking" the mutex
    - V operation: "unlocking" or "releasing" the mutex
    - *"Holding"* a mutex: locked and not yet unlocked.
  - *Counting semaphore*: used as a counter for set of available resources.

Proper Synchronization: goodcnt.c

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

```
 volatile long cnt = 0;  /* Counter */
sem_t mutex;           /* Semaphore that protects cnt */
Sem_init(&mutex, 0, 1); /* mutex = 1 */
```

- **Surround** critical section with *P* and *V*:

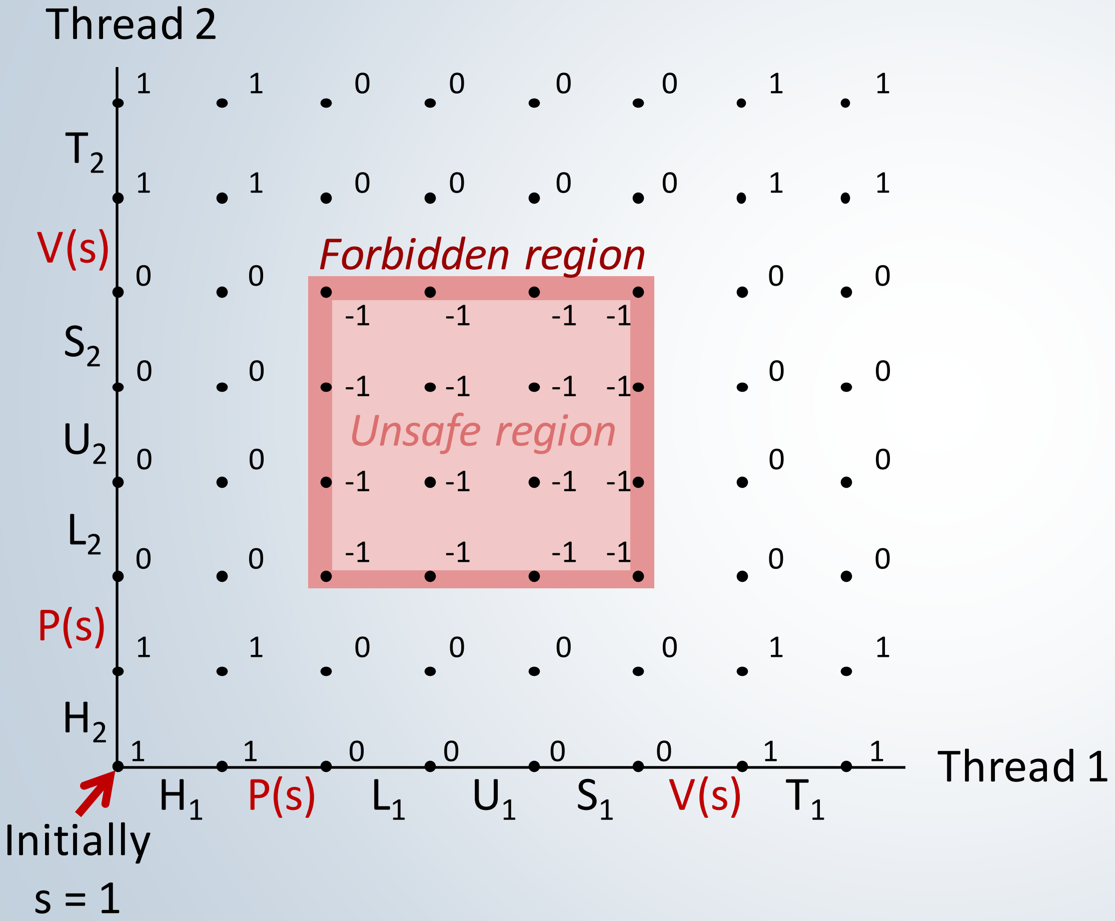
```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

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

Warning: It’s orders of magnitude slower than `badcnt.c`.

Why Mutexes Work



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 that cannot be entered by any trajectory.

Summary

- Programmers need a clear model of how variables are shared by threads.
- Variables shared by multiple threads must be protected to ensure mutually exclusive access.
- Semaphores are a fundamental mechanism for enforcing mutual exclusion.