Concurrent Programming: *Synchronizing threads:* 2. *Semaphores*

Semaphores

- ***Semaphore:*** **non-negative *global (shared)* integer synchronization variable. Manipulated by atomic operations *P* and *V*.**

Semaphores

- ***Semaphore:*** **non-negative *global (shared)* integer synchronization variable. Manipulated by atomic operations *P* and *V*.**
- **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.

Semaphores

- ***Semaphore:*** **non-negative *global (shared)* integer synchronization variable. Manipulated by atomic operations *P* and *V*.**
- **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*.

Semaphores

- ***Semaphore:*** **non-negative *global (shared)* integer synchronization variable. Manipulated by atomic operations *P* and *V*.**
- **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)***

Semaphores

- ***Semaphore:*** **non-negative *global (shared)* integer synchronization variable. Manipulated by atomic operations *P* and *V*.**
- **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)***
- **Proberen/Verhogen, Procure/Vacate, Acquire/Release, Wait/Signal**

C Semaphore Operations

Pthreads functions, implemented in libc, using CPU-dependent instructions and syscall `futex` (since 2015)

```
#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) */
```

https://developers.redhat.com/blog/2015/01/28/recent-improvements-to-concurrent-code-in-glibc/

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 */
```

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 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.

goodcnt.c: Proper Synchronization

- 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

```
$ ./goodcnt 10000
OK cnt=20000
$ ./goodcnt 10000
OK cnt=20000
$
```

goodcnt.c: Proper Synchronization

- 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

```
$ ./goodcnt 10000
OK cnt=20000
$ ./goodcnt 10000
OK cnt=20000
$
```

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

goodcnt.c: Proper Synchronization

- 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*:

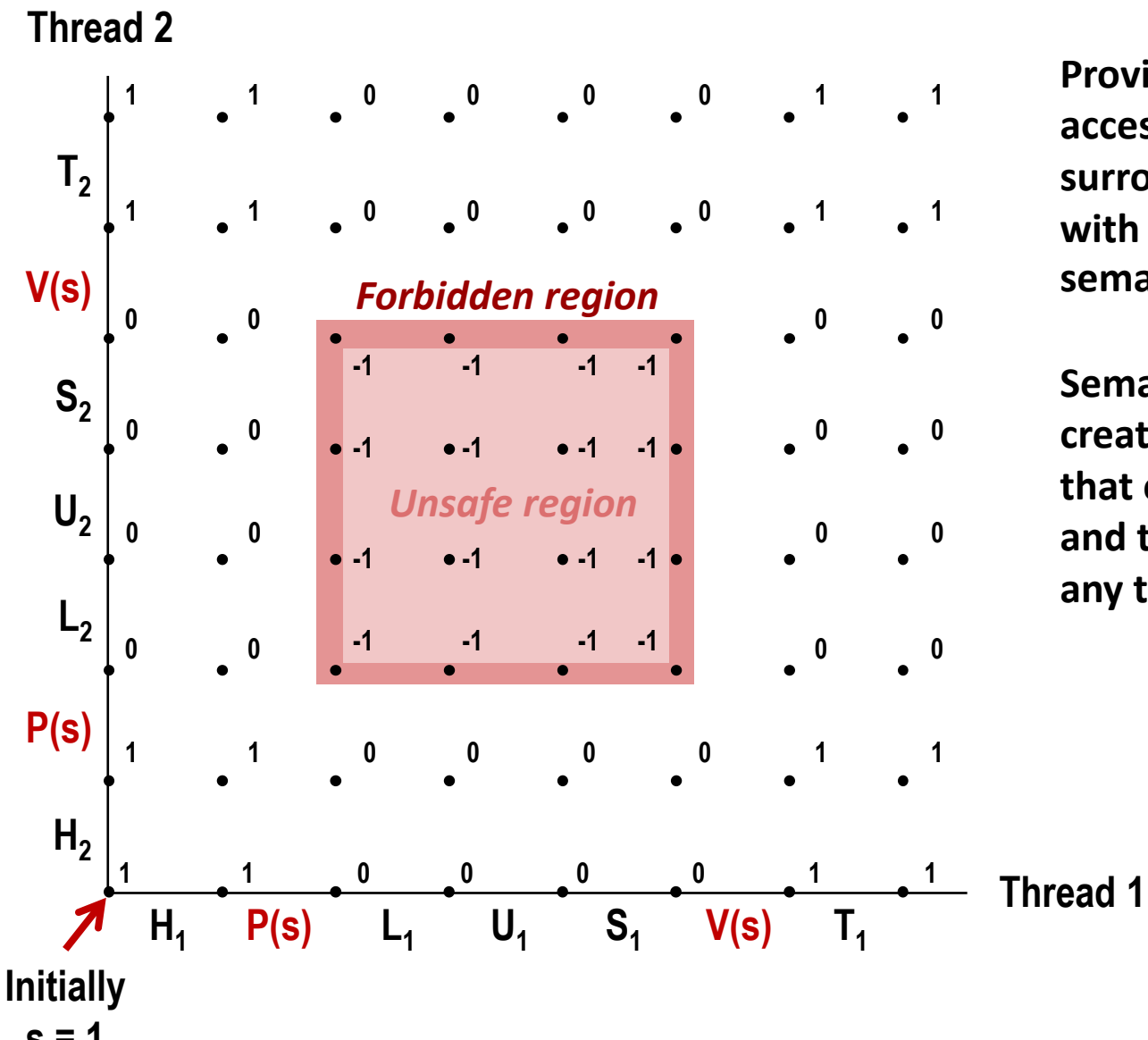
```
for (i = 0; i < niters; i++) {
    P(&mutex);
    cnt++;
    V(&mutex);
 }
```

goodcnt.c

```
$ ./goodcnt 10000
OK cnt=20000
$ ./goodcnt 10000
OK cnt=20000
$
```

Warning: It's orders of magnitude slower than `badcnt.c`. Better put P, V around loop?

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.

`volatile` considered useless (or harmful)

```
volatile long cnt = 0; /* Counter */
```

- `volatile` on global variables gives a sense of safety
- …but *only* gives a hint to compiler to sync variable back to memory after each line of code (costly!)

`volatile` considered useless (or harmful)

```
volatile long cnt = 0; /* Counter */
```

- `volatile` on global variables gives a sense of safety
- …but *only* gives a hint to compiler to sync variable back to memory after each line of code (costly!)
- ***Race conditions still very much possible!***

`volatile` considered useless (or harmful)

```
volatile long cnt = 0; /* Counter */
```

- `volatile` on global variables gives a sense of safety
- …but *only* gives a hint to compiler to sync variable back to memory after each line of code (costly!)
- ***Race conditions still very much possible!***
- We do not want `cnt` to end up in a register…
- …but it won't!

```
{
  my_library_call ();  /* Can't assume won't modify cnt */
  ...
}
```