Semaphores

Godmar Back

Virginia Tech

October 8, 2024

Coordinating Threads

- A common task is for multiple threads to coordinate regarding the occurrence of events, e.g:
  - "has thread A computed a value thread B wants to use?"
- Sometimes called a "ordering," "precedence," or "scheduling constraint", e.g. "Must do Y after X".

Note:

Although the participating threads usually share state, this is a different problem from coordinating access to such state (for which mutual exclusion is a solution)

Coordinating Threads

A solution to this problem should be

- Correct (thread B should never miss the event that thread A has computed its value)
- Efficient
  - Not waste resources (i.e., busy-waiting)
  - Does not induce unnecessary delay (e.g. not rely on thread B periodically polling)

Semaphores provide a first solution to this problem.

VIRGINIA TECH.

Semaphores

Semaphores are an abstraction introduced by Edsger Dijkstra [2] in the 1960's. A semaphore is an ADT that encapsulates a counter S and provides 2 operations for it:

- P(S) aka "down" or "wait" – if counter value is greater than zero, decrement it. Otherwise, block until it becomes greater than zero, then decrement it.
- V(S) aka "up" or "signal" or "post" – increment the counter's value, and (if necessary) ensure that any threads blocked in a P(S) operation are unblocked.
- Programmer chooses initial value V_i

Semaphore Invariant

$V_i + |U| - |D| \geq 0$ where $|U|$ and $|D|$ are the numbers of completed up- and down operations, respectively. ("Semaphore doesn't go negative")

Producer

```c
int coin_flip;
sem_t coin_flip_done;   // semaphore for thread 1 to signal coin flip
// requires sem_init(&coin_flip_done, 0, 0) to give initial value 0

static void * thread1(void *_)
{
    coin_flip = rand() % 2;
    sem_post(&coin_flip_done);  // raise semaphore, increment, 'up'

    printf("Thread 1: flipped coin %d\n", coin_flip);
}
```

Consumer

```c
static void * thread2(void *_)
{
    // wait until semaphore is raised, then decrement, 'down'
    sem_wait(&coin_flip_done);
    printf("Thread 2: flipped coin %d\n", coin_flip);
}
```

Discussion

- Code works now matter the relative order in which threads arrive at the `sem_wait` and `sem_post` calls.
- Semaphores can be used to solve a number of classical synchronization problems, see [3] for examples.
- Semaphores are a more general synchronization device: a "binary semaphore," which can only take the values 0 and 1, can be used to solve the mutual exclusion problem.
- Can be generalized to represent acquisition/release of up to N resources by setting initial value to N.
- Using semaphores for mutual exclusion is not however recommended, mutexes (e.g., `pthread_mutex_t`) should be used instead [1].

VIRGINIA TECH

Using a semaphore for mutual exclusion

```
sem_t S;
sem_init(&S, 0, /*initial value=*/ 1);

void lock_acquire()
{    // try to decrement, wait if 0
     sem_wait (S);
}

void lock_release()
{    // increment (wake up waiters if any)
     sem_post(S);
}
```

How do I tell what a semaphore is used for?

Look at the initial value: if 1, the semaphore represents a mutex. If 0, the semaphore is used for ordering.

References

[1] Bryan Cantrill and Jeff Bonwick.
Real-world concurrency.
Queue, 6(5):16–25, September 2008.

[2] Edsger W. Dijkstra.
Ewd-74: Over seinpalen.
E.W. Dijkstra Archive. transcription, n.d.

[3] Allen B. Downey.
The Little Book of Semaphores.
available online.