Semaphores

COS 417: Operating Systems

Spring 2025, Princeton University

Wait, *another* **synchronizitation primitive?**

Wait, *another* synchrononiziation primitive?

Yes, another one. Deal with it.

Wait, *another* **synchroniziation primitive?**

Yes, another one. Deal with it.

Book: “As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems.”

Wait, *another* **synchroniziation primitive?**

Yes, another one. Deal with it.

Book: “As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems.”

Well… no! We’ve seen: we can build CVs from mutexes, and mutexes from atomic integer instructions.

Wait, *another* **synchroniziation primitive?**

Yes, another one. Deal with it.

Book: “As we know now, one needs both locks and condition variables to solve a broad range of relevant and interesting concurrency problems.”

Well… no! We’ve seen: we can build CVs from mutexes, and mutexes from atomic integer instructions.

But remember, we’re dealing with abstractions here…

Musing on Abstractions

An unnecessary abstractions is a terrible tragedy. Necessary if:

- Allows system to implement more efficiently than application
- Allows portability
- Help programmers reason about correctness more easily
  - But this one can be done in a library!

Different synchronization abstractions serve all three.

Semaphore Interface

```c
// Initialize a semaphore with initial value `value`
void sem_init(sem_t *s, unsigned int value);

// Decrement the semaphore's value, waiting first value is `0`.
void sem_wait(sem_t *s, unsigned int value);

// Increment the semaphore's value
void sem_post(sem_t *s, unsigned int value);
```

Semaphore Interface

```
// Initialize a semaphore with initial value `value`
void sem_init(sem_t *s, unsigned int value);

// Decrement the semaphore's value, waiting first value is `0`.
void sem_wait(sem_t *s, unsigned int value);

// Increment the semaphore's value
void sem_post(sem_t *s, unsigned int value);
```

Invariants:

- Semaphore value is never negative
- # `waits` returned <= # `posts` returned + initial value

Example: A Resource Pool

Assume ~~a spherical cow~~ an atomic queue…

```
typdef struct {
  sem_t s; queue r;
} pool;

void release(pool *wp, void *w)
{
  atomic_enqueue(&wp->r, w);
  sem_post(&wp->s);
}
```

```
void init_pool(pool *wp) {
    sem_init(&wp->s, 0);
}

void *acquire(pool *wp) {
  sem_wait(&wp->s);
  return
    atomic_dequeue(&wp->r);
}
```

Example: Resource Pool

Using a semaphore gave us:

- A simple implementation that’s easy to reason about
- Implementation works regardless of *how* system implements semaphores

Example: Resource Pool

Using a semaphore gave us:

- A simple implementation that's easy to reason about
- Implementation works regardless of *how* system implements semaphores

But, can we implement this as a library?

Example: Resource Pool

Using a semaphore gave us:

- A simple implementation that’s easy to reason about
- Implementation works regardless of *how* system implements semaphores

But, can we implement this as a library?

Can we implement this as a library without sacrificing portability?

Semaphore imlemented with a Mutex

```
typedef struct {
  mutex_t m;
  int v;
} mysem_t;

void mysem_post(mysem_t *s) {
  mutex_lock(&s->m);
  s->v++;
  mutex_unlock(&s->m);
}
```

- *Almost* Linux kernel impl.
  - Using a spinlock for a mutex
  - Plus some magic startdust

```
void mysem_wait(mysem_t *s) {
  while(1) {
    mutex_lock(&s->m);
    if (s->v <= 0) {
      mutex_unlock(&s->m);
      continue;
    } else {
      s->v--;
      mutex_unlock(&s->m);
      break;
    }
  }
}
```

Semaphore imlemented with a Mutex

```
void mysem_post(mysem_t *s) {
  mutex_lock(&s->m);
  s->v++;
  mutex_unlock(&s->m);
}
void mysem_wait(mysem_t *s) {
  while(1) {
    mutex_lock(&s->m);
    if (s->v <= 0) {
      mutex_unlock(&s->m);
      continue;
    } else {
      s->v--;
      mutex_unlock(&s->m); break;
    }
  }
}
```

- Is this efficient?

Semaphore imlemented with a Mutex

```
void mysem_post(mysem_t *s) {
  mutex_lock(&s->m);
  s->v++;
  mutex_unlock(&s->m);
}
void mysem_wait(mysem_t *s) {
  while(1) {
    sleep(1);
    mutex_lock(&s->m);
    if (s->v <= 0) {
      mutex_unlock(&s->m);
      continue;
    } else {
      s->v--;
      mutex_unlock(&s->m); break;
    }
  }
}
```

- Is this efficient?
- What about this?

Semaphore imlemented with a Mutex + CV

```
typedef struct {
  mutex_t m;
  cond_t c;
  int v;
} mysem_t;
```

- Is this efficient?
-
-

```
void mysem_post(mysem_t *s) {
  mutex_lock(&s->m);
  s->v++;
  cond_signal(&s->c);
  mutex_unlock(&s->m);
}

void mysem_wait(mysem_t *s) {
    mutex_lock(&s->m);
    while (s->v <= 0) {
      cond_wait(&s->c, &s->m);
    }
    s->v--;
    mutex_unlock(&s->m);
}
```

Semaphore imlemented with a Mutex + CV

```
typedef struct {
  mutex_t m;
  cond_t c;
  int v;
} mysem_t;
```

- Is this efficient?
- Is this *fair*?
-

```
void mysem_post(mysem_t *s) {
  mutex_lock(&s->m);
  s->v++;
  cond_signal(&s->c);
  mutex_unlock(&s->m);
}

void mysem_wait(mysem_t *s) {
    mutex_lock(&s->m);
    while (s->v <= 0) {
      cond_wait(&s->c, &s->m);
    }
    s->v--;
    mutex_unlock(&s->m);
}
```

Semaphore imlemented with a Mutex + CV

```
typedef struct {
  mutex_t m;
  cond_t c;
  int v;
} mysem_t;
```

- Is this efficient?
- Is this *fair*?
- pthreads implementation

```
void mysem_post(mysem_t *s) {
  mutex_lock(&s->m);
  s->v++;
  cond_signal(&s->c);
  mutex_unlock(&s->m);
}

void mysem_wait(mysem_t *s) {
    mutex_lock(&s->m);
    while (s->v <= 0) {
      cond_wait(&s->c, &s->m);
    }
    s->v--;
    mutex_unlock(&s->m);
}
```

Let’s implement a CV using a semaphore!

- Never do this at home
- We’ll probably get it wrong

Let's implement a CV using a semaphore!

- Never do this at home
- We'll probably get it wrong

Take 10 minutes to think about this.

What should our data structure look like?