Semaphores

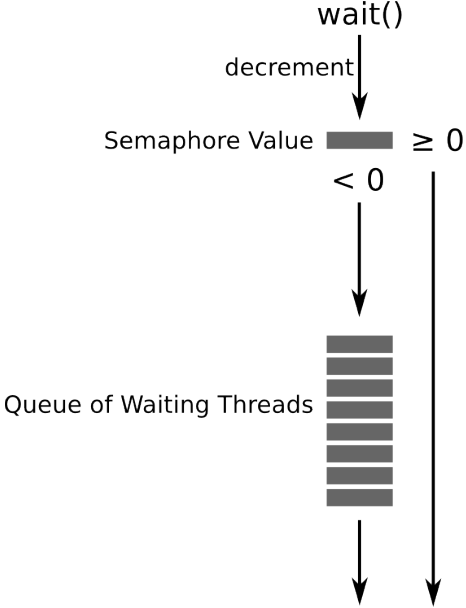
Daniel Zappala

CS 360 Internet Programming
Brigham Young University

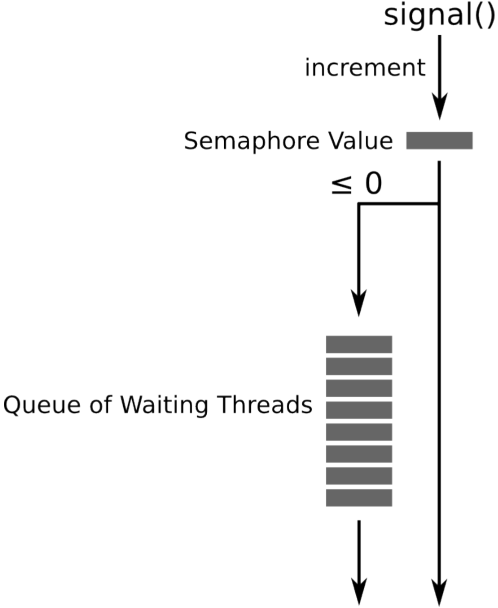
Semaphores

- semaphore is a shared variable maintained by OS
  - contains an integer and a queue
  - value initialized $>= 0$
- `wait(s)`: wait for a signal on semaphore *s*
  - decrements semaphore, blocks if value < 0
  - if blocked, process put on the queue, suspends until signal is sent
- `signal(s)`: transmit a signal to semaphore *s*
  - increments semaphore
  - if value $<= 0$ then unblock someone
- `wait()` and `signal()` are atomic operations and cannot be interrupted

wait()


wait()
decrement
Semaphore Value
≥ 0
< 0
Queue of Waiting Threads

signal()


signal()
increment
Semaphore Value
≤ 0
Queue of Waiting Threads

Types of Sempahores

- binary semaphore
  - only one process at a time may be in the critical section
- counting semaphore
  - a fixed number of processes > 0 may be in the critical section
- OS determines whether processes are released from queue in FIFO order or otherwise; usually FIFO in order to prevent starvation

Using Semaphores

```
semaphore s = 1;

void thread(int i) {
    while (true) {
        wait(s);
        /* critical section */
        signal(s);
       /* remainder */
    }
}
```

- semaphore protects critical section
- can set s to > 1 to let more than one process in the critical section
  - $s >= 0$: number that can enter
  - $s < 0$: number that are waiting

POSIX Semaphores

POSIX Semphores

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
```

- `sem_init()`: sets initial value of semaphore; `pshared = 0` indicates semaphore is local to the process
- `sem_wait()`: suspends process until semaphore is > 0, then decrements semaphore
- `sem_trywait()`: returns EAGAIN if semaphore count is $= 0$
- `sem_post()`: increments semaphore, may cause another thread to wake from `sem_wait()`

Example Code

- see example code `semaphore.cc`

▸ GitHub

Producer Consumer

Producer Consumer Problem

- one or more producers are generating data and placing them in a buffer
- one or more consumers are taking items out of the buffer
- only one producer or consumer may access the buffer at any time

Producer Consumer

```
vector buffer;
append(item) {
  buffer.append(item);
}
take() {
  return buffer.remove();
}
```

producer:

```
while (true) {
  item = produce();
  append(item);
}
```

consumer:

```
while (True) {
  item = take();
  consume(item);
}
```

Producer Consumer with Infinite Buffer

```
sem_t s, n;
sem_init(&s,0,1);
sem_init(&n,0,0);
```

producer:

```
while (True) {
  item = produce();
  sem_wait(&s);
  append(item);
  sem_post(&s);
  sem_post(&n);
}
```

consumer:

```
while (True) {
    sem_wait(&n);
    sem_wait(&s);
    item = take();
    sem_post(&s);
    consume(item);
}
```

Looking at the Code ...

1. *What is the purpose of semaphore s?*
2. *What is the purpose of semaphore n?*
3. *Why is semaphore s initialized to 1 but semaphore n is initialized to 0?*
4. *Why can the producer signal n every time an item is added to the buffer ?*
5. *Can the producer swap the signals for n and s?*
6. *Can the consumer swap the waits for n and s?*

Important Insights

- two purposes for semaphores
  - mutual exclusion: semaphore s controls access to critical section
  - signalling: semaphore n coordinates when the buffer is empty: consumer waits if buffer is empty, producer signals when buffer becomes non-empty
- avoid race conditions
  - *item* keeps a local copy of the data protected by the semaphore so that it can be accessed later
  - reduces amount of processing inside the critical section

Important Insights

- *n*: semaphore value is number of items in buffer
  - if $n == 0$, consumer must wait
  - can swap `sem_post(&n);` and `sem_post(&s);` in producer and be OK
  - can't swap `sem_wait(&n);` and `sem_wait(&s);` in consumer: otherwise consumer enters and then waits and deadlocks the producer!
- ordering of semaphore operations is important

Producer Consumer with Finite Buffer

```
sem_t s, n, e;
sem_init(&s,0,1);
sem_init(&n,0,0);
sem_init(&e,0,BUFFER_SIZE);
```

producer:

```
while (True) {
    produce();
    sem_wait(&e);
    sem_wait(&s);
    append();
    sem_post(&s);
    sem_post(&n);
}
```

consumer:

```
while (True) {
    sem_wait(&n);
    sem_wait(&s);
    take();
    sem_post(&s);
    sem_post(&e);
    consume();
}
```

Looking at the Code ...

1. *What is the difference between semaphore e and semaphore n?*

Thread-Safe Classes

Organizing Semaphores

- difficult to get semaphores right
  - match wait and signal
  - put in right order
  - scattered throughout code
- put them in a class, with the data structures they use
  - private data structures, public methods
  - any object calling this class is thread-safe

Thread-Safe Classes

```
class Buffer {
  public:
    append(item) {
      sem_wait(&e);
      sem_wait(&s);
      buffer.append(item);
      sem_post(&s);
      sem_post(&n);
    };
    take() {
      sem_wait(&n);
      sem_wait(&s);
      item = buffer.remove();
      sem_post(&s);
      sem_post(&e);
      return item;
    };

  private:
    vector buffer;
};
```