Semaphores

ICS432 - Fall 2008
Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

Semaphores

- We have seen
  - Locks for mutual exclusion
  - Condition Variables for synchronization
- Semaphores are unified signaling mechanisms for both mutual exclusion and synchronization
  - Low-level
  - Efficient
  - Can remove the need for counter and flag variables
- History
  - Proposed in 1968 by Dijkstra
  - Inspired by railroad semaphores:
    - Up/Down, Red/Green

Semaphore Operations

- A semaphore is a shared integer variable that is never < 0
  - Can be initialized to whatever integer value
- The semaphore provides two atomic operations
- The P operation
  - P: from Dutch "proberen", "to test"
  - Waits on a (hidden) condition variable for the variable to be > 0 and then decrements the semaphore
- The V operation
  - V: from Dutch "verhogen", "to increment"
  - Increments the semaphore
- Could be implemented with locks and condition variables, or from scratch

Types of Semaphores

- Binary Semaphore:
  - Takes values 0 and 1
  - Can be used for mutual exclusion
  - Can be used for signaling
- Counting Semaphores:
  - Takes any nonnegative value
  - Typically used to count resources and block resource consumers when all resources are busy

Critical Section with Semaphores

- Doing a critical section with a semaphore is as simple as with a lock

```
semaphore_t mutex = 1;
int shared_variable;
void worker() {
  while(1) {
    P(mutex);
    shared_variable++;
    V(mutex);
  }
}
```

Signaling Semaphores

- Another use for a binary semaphore is to signal some event
  - A thread waits for an event by calling P
  - A thread signals the event by calling V
- Example: a barrier between two threads

Thread #1

```
. . .
. . .
V(ready1);
P(ready2);
. . .
. . .
```

Thread #2

```
. . .
. . .
V(ready2);
P(ready1);
. . .
. . .
```

Global Variables

```
semaphore ready1 = 0;
semaphore ready2 = 0;
```

Signaling with Semaphores

- One can use Semaphores to do exactly what we were doing with condition variables:

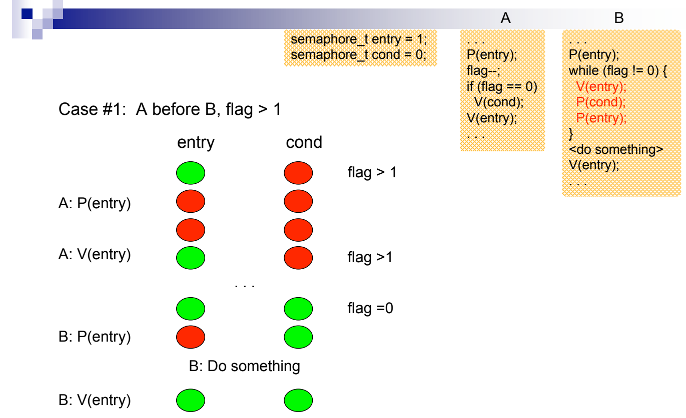
```
semaphore_t entry = 1;
semaphore_t cond = 0;
```

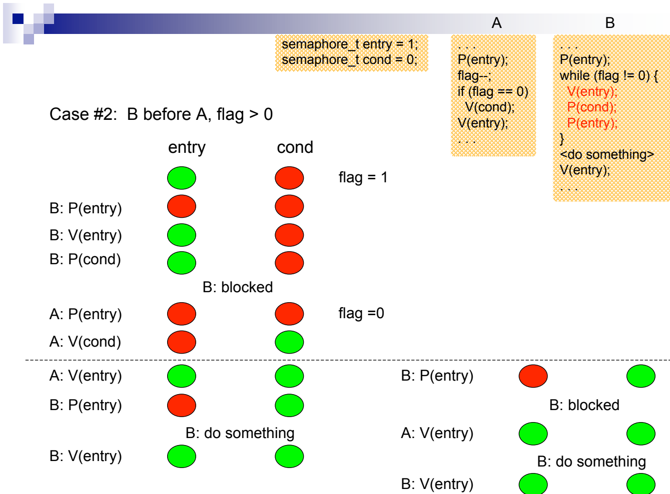
```
. . .
P(entry);
flag--;
if (flag == 0)
  V(cond);
V(entry);
. . .
```

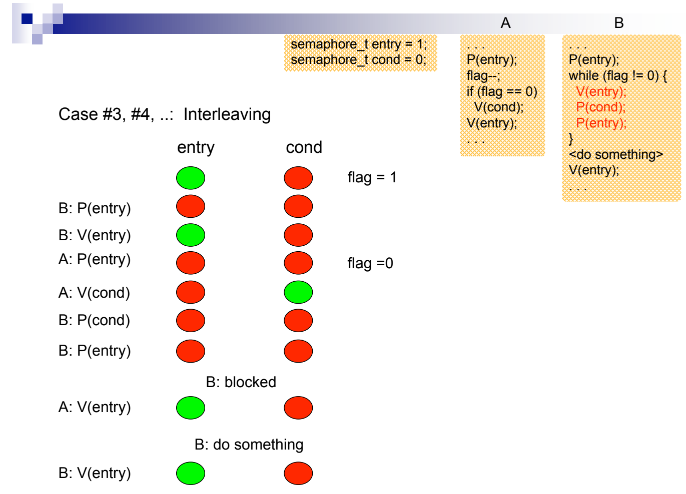
```
. . .
P(entry);
while (flag != 0) {
  V(entry);
  P(cond);
  P(entry);
}
<do something>
V(entry);
. . .
```

Equivalent to a wait() on condition variable
- release the mutex
- wait
- reacquire the mutex

In a while loop in case two such threads







Split Binary Semaphores

- A typical usage of binary semaphores is to do mutual exclusion and signaling at the same time
- Consider a specific Producer/Consumer problem
  - We have an arbitrary number of producers
  - We have an arbitrary number of consumers
  - We have a buffer that can contain a single element, consumed by consumers and produced by producers
  - Producers must be delayed until the buffer is empty
  - Consumers must be delayed until the buffer is full
- This can be easily implemented with 2 binary semaphores

Single Buffer Prod/Cons

```
semaphore_t empty = 1;
semaphore_t full = 0;
```

```
void producer() {
  while(true) {
    P(empty);
    buffer = <some value>;
    V(full);
  }
}
```

```
void consumer() {
  while(true) {
    P(full);
    consume(buffer);
    V(empty);
  }
}
```

Single Buffer Prod/Cons

```
semaphore_t empty = 1;
semaphore_t full = 0;
```

```
void producer() {
  while(true) {
    P(empty);
    buffer = <some value>;
    V(full);
  }
}
```

```
void consumer() {
  while(true) {
    P(full);
    consume(buffer);
    V(empty);
  }
}
```

- There is a simple “ping-pong” between the full and the empty semaphores
  - $0 \le full + empty \le 1$
- This ensures mutual exclusion . . .

Split Binary Semaphores

- Thread #1: P(X) <S> V(Y)
- Thread #2: P(Y) <T> V(X)
- Semaphores are initialized to (X=0,Y=1)
- They alternate between (X=0,Y=1) and (X=1,Y=0)
- Example: Starting with (X=0,Y=1)
  - Thread #1 cannot execute statement <S>
  - Thread #2 sets Y to 0
  - Thread #2 executes statement <T>
  - Thread #2 sets X to 1
  - We now have (X=1,Y=0)
  - Thread #2 cannot execute statement <T>
  - Thread #1 can execute statement <S>
  - ...

General Semaphores

- Semaphores that take values higher than 1 are typically used to control access to a limited number of resources
  - In the previous example we controlled access to a single resource
- The value of the semaphore indicates the number of free resources
  - N: all free
  - 0: none free
  - And anything in between
- Let’s look at the “bounded buffer” producer/consumer problem
  - We already did this with condition variables, but we’ll see now that with semaphores it’s a bit easier

Bounded Buffer Prod/Cons

- Problem
  - Arbitrary numbers of producers and consumers
  - The buffer can only store N elements
  - As we did before, our buffer will be a queue and we wish it to be at most N elements
- In our split binary semaphore example, mutual exclusion was enforced implicitly with the full/empty semaphores
- With General semaphores, we need an extra semaphore for mutual exclusion
- Let’s look at the code

Single Buffer Prod/Cons

```
semaphore_t notfull = n;
semaphore_t notempty = 0;
semaphore_t mutex = 1;
```

```
void producer() {
  while(true) {
    P(notfull);
    P(mutex);
    <add element to queue>
    V(mutex);
    V(notempty);
  }
}
```

```
void consumer() {
  while(true) {
    P(notempty)
    P(mutex);
    <remove element from queue>
    V(mutex);
    V(notfull);
  }
}
```

Readers/Writers

- Another classical concurrency model is the reader/writer problem
  - An example of *selective mutual exclusion*
- We have two kinds of processes:
  - Readers: read records from a database
  - Writers: read and write records from a database
- Selective mutual exclusion
  - Concurrent readers are allowed
  - A writer should access the database in mutual exclusion with all other writers and readers
- Representative of database applications
  - e.g., a Web/database server with one thread per transaction

A Naive Solution

```
semaphore_t rw = 1;
```

```
void reader() {
  while(true) {
    P(rw);
    <read from the DB>
    V(rw);
  }
}
```

```
void writer() {
  while(true) {
    P(rw);
    <write to the DB>
    V(rw);
  }
}
```

- This solution works but implements too strict a constraint
- No concurrent database access
- Loss of throughput/performance because concurrent reads should be allowed
  - In many applications, there are few writers and many readers

Reader-Preferred Solution

- One simple fix is to allow multiple readers in a “greedy” fashion:
  - There is still only a rw semaphore
  - While a reader is reading, other readers should be allowed in
  - Therefore we should have a variable, nr, keeping track of the current number of readers
  - That variable is used updated by all readers, and should be protected by a mutual exclusion semaphore
- Let’s look at the code

Reader-Preferred Solution

```
void reader() {
  while(true) {

    P(mutex);
      if (nr == 0)  P(rw);  // I am first
      nr++;
    V(mutex);

    <read from the DB>

    P(mutex);
    nr--;
    if (nr == 0) V(rw);    // I am last
    V(mutex);
  }
}
```

```
semaphore_t mutex = 1;
semaphore_t rw = 1;
int nr = 0;
```

```
void writer() {
  while(true) {
    P(rw);
    <write to the DB>
    V(rw);
  }
}
```

Reader-Preferred Solution

- The problem of the reader-preferred solution is that it is too reader-preferred
- There could be starvation of the writers
  - If there is always a reader able to read, the rw semaphore will be monopolized forever
- Turns out it’s very difficult to modify the code to make it fair between readers and writers
  - There is a classic solution that uses synchronization and the “passing the baton technique”
  - Based on a invariant condition and subtle signaling
  - Many intricate solutions presented on-line
- Let’s look at a simple but pretty good solution

Maximum number of readers

- Defining a maximum number of allowed concurrent readers simplifies the problem!
  - And most likely makes sense for most applications
- Let’s say we allow at most N concurrent active readers
- Then we can create a “resource” semaphore with initial value N
- Each reader needs to acquire one resource to be able to read
  - Therefore, N concurrent readers are allowed
- Each writer needs to acquire N resources to be able to write
  - Therefore, only one writer can be executing at a time and no readers can be executing concurrently
- Let’s look at the code

Reader/Writer

```
semaphore_t  sem = N;
```

```
void reader() {
  while(true) {
    P(sem);
    <read from the DB>
    V(sem);
  }
}
```

```
void writer() {
  while(true) {
    for (i=0; i<N; i++)
      P(sem);
    <write to the DB>
    for (i=0; i<N; i++)
      V(sem);
  }
}
```

There is still a problem...

Reader/Writer

- Deadlock!
  - One could have two writers each start acquiring resources concurrently
  - For instance
    - Writer #1 holds 1 resource
    - Writer #2 holds N-1 resources
  - They're both blocked forever
- Solution: Not allow two writers to execute the for loop of P() calls concurrently
- This can easily be done with mutual exclusion
- That is, we need another semaphore

```
void writer() {
  while(true) {
    for (i=0; i<N; i++)
      P(sem);
    <write to the DB>
    for (i=0; i<N; i++)
      V(sem);
  }
}
```

Decent Reader/Writer Solution

```
semaphore_t  sem = N;
semaphore_t  wmutex = 1;
```

```
void reader() {
  while(true) {
    P(sem);
    <read from the DB>
    V(sem);
  }
}
```

```
void writer() {
  while(true) {
    P(wmutex);
    for (i=0; i<N; i++)
      P(sem);
    V(wmutex);
    <write to the DB>
    for (i=0; i<N; i++)
      V(sem);
  }
}
```

Pthreads and Semaphores

- We have talked about Pthreads
  - mutex locks
  - condition variables
- One can implement semaphores based on the above
- But Pthreads provide a "semaphore extension"
  - sem_t semaphore
  - sem_init(&semaphore, 0, some_value);
  - sem_wait(&semaphore);
  - sem_post(&semaphore);

Pros/Cons for Semaphores

- Good
  - A single mechanism for many things
    - mutual exclusion
    - resource sharing
    - signaling/blocking
  - General enough to solve any concurrency/synchronization problem
- Bad
  - The fact that a single mechanism is used for multiple things can in fact make a program very difficult to understand
  - It's error prone
    - Forget to call V()
  - Not very modular
    - The use of a semaphore in a thread depends on its use in another thread
- A combination of locks and cond. variables is equivalent in power to semaphores
  - It's not clear which one is preferable
  - You may be seeing both in practice, depending on projects, people, languages
- In the next lecture we'll see Monitors