Semaphore Example II: Producer/Consumer

Work queue shared among threads is example **producer/consumer** problem

Aka **bounded buffer problem**

1 or more threads write to one end of queue/buffer

1 or more threads "read from" other end of queue/buffer — but it's really a write operation because items are removed

Producer/Consumer Problem, II

Potential concurrency: if queue is long enough, can add & delete simultaneously

Potential problems:

1. Multiple simultaneous adds

2. Multiple simultaneous deletes

3. Stop consumer when buffer becomes empty, then re-start consumer when item becomes available

4. Stop producer when buffer becomes full, then re-start producer when space becomes available

#1 and #2 solved using mutual exclusion

#3 and #4 require threads to "signal" (i.e., coordinate with) each other

Producer/Consumer Problem, III

```
/* semaphore for mutual exclusion of critical section */
int mutex = 1;

/* these semaphores are "condition variables" */
int spaces = buffer_size;
int items = 0;

Producer:
    P(spaces);
    P(mutex);
    <add to buffer>
    V(mutex);
    V(items);

Consumer:
    P(items);
    P(mutex);
    <remove from buffer>
    V(mutex);
    V(spaces);
```

Producer/Consumer Problem, IV

Condition variables indicate the state of (i.e., “the condition of”) the buffer – hence the name “condition variables”

For this problem, programmer must think hard about where to place P, V – above solution is far from obvious

Producer/Consumer Notes, I

Q: Why is `mutex` semaphore needed?

A: For mutual exclusion in case there is >1 producer or >1 consumer

Q: Why are TWO condition variable semaphores needed?

A: P & V affect the value of only one semaphore variable. Need to wakeup sleeping threads when items become available OR when space becomes available. So need ONE semaphore variable that will wakeup sleeping consumers when items become available AND ANOTHER that will wakeup sleeping producers when space becomes available.

Producer/Consumer Notes, II

Q: Why are P/V for spaces & items placed OUTSIDE critical section?

A: Otherwise might deadlock

Consider what could happen if code were instead:

```
Producer:
    P(mutex);
    P(spaces);
       ...
    V(items)
    V(mutex);

Consumer:
    P(mutex);
    P(items);
       ...
    V(spaces)
    V(mutex);
```

Producer/Consumer Notes, III

This could happen if buffer were empty:

1. Consumer executes P(mutex) ... passes thru

2. Consumer executes P(items) ... blocks waiting for Producer's V(items)

3. Producer executes P(mutex) ... blocks

Now both threads are blocked

Understanding the Solution, I

V operations provide wakeup “signaling” to waiting threads:

```
/* producer */
P(mutex);
<add to buffer>
V(mutex);
<signal to threads waiting for items:
           there is another item>


/* consumer */
P(mutex);
<remove from buffer>
V(mutex);
<signal to threads waiting for space:
           there is space for another item>
```

Understanding the Solution, II

P operations cause threads to wait when conditions are not yet right to proceed:

```
/* producer */
<if (no space)
     then wait until space available>
P(mutex);
<add to buffer>
V(mutex);
<signal to threads waiting for items:
              there is another item>


/* consumer */
<if (no items)
     then wait until items available>
P(mutex);
<remove from buffer>
V(mutex);
<signal to threads waiting for space:
              there is more space>
```

Semaphore Summary

Semaphores accomplish TWO purposes:

1. Limiting access (P)

2. “Signaling” some sleeper when access becomes permissible (V)

Semaphores often awkward: must translate real synchronization condition into increment/decrement of integer

E.g., “`if (no space) then wait until space available`” is obvious, “`P(spaces)`” is less obvious

Implementation

How special instructions might be used to implement Linux semaphores: pseudocode for `sem_wait`:

```
struct semaphore_internals_t {
    int lock;      // lock
    int value;     // semaphore's value
    <linked list of waiting threads>
};
typedef semaphore_internals_t sem_t;
int sem_wait(sem_t *sem) {
    <loop using special instruction to get lock>
    if (sem->value > 0) {
        sem->value = sem->value - 1;
        <drop lock>
        return 0;
    } else if (sem->value == 0) {
        <thread places itself on wait list>
        <drop lock>
        <thread resumes from this point when awakened>
        <loop using special instruction to get lock>
        sem->value = sem->value - 1;
        <drop lock>
        return 0;
    }
}
```