Multiple producers and consumers

Interestingly, the producer-consumer code presented previously works if there is only one producer and one consumer thread. However, if there are multiple producers or multiple consumers, this code fails. For example, if two producers try to produce to a full queue, both of them may start waiting on `not_full`. If a consumer consumes one element, and calls `notify`, both producers will wakeup. Each of them will, in turn, try to acquire `qlock` and produce an element. However, the consumer only consumed one element, and the two producers will produce two elements, causing a queue overflow!

The problem is that the producer threads, do not check the condition after waking up from a `wait()`, and simply assume that the condition is true. This problem can be fixed by requiring the threads to check the condition again after returning from `wait`. This can be done by replacing the `if` condition with a `while` condition, as follows:

```
char queue[MAX];  //global
int head = 0, tail = 0; //global
struct cv not_full, not_empty;
struct lock qlock;

void produce(char data) {
  acquire(&qlock);
  while ((head + 1) % MAX  ==  tail) {
    wait(&not_full, &qlock);
  }
  queue[head] = data;
  head = (head + 1) % MAX;
  notify(&not_full);
  release(&qlock);
}

char consume(void) {
  acquire(&qlock);
  while (tail == head) {
    wait(&not_empty, &qlock);
  }
  e = queue[tail];
  tail = (tail + 1) % MAX;
  notify(&not_empty);
  release(&qlock);
  return e;
}
```

A thread that has just woken up from `wait()` will again check the condition (because of the `while` construct). If the condition is true, it will go back to sleep (by another call to `wait()`). If the condition is false, it will fall through and perform its operation. This code is correct, even for multiple producers and multiple consumers.

In general, usage of condition variables has the following pattern:

```
struct lock mutex;
struct cv condition_variable;

acquire(&mutex);
while (!condition) {
  wait(&condition_variable, &mutex);
}
...
release(&mutex);
```

Here, the `condition_variable` is associated with the `condition` becoming true. The mutex variable is a lock used for mutual exclusion.

Semaphores

A common pattern in programming, as also seen in the producer-consumer example, is counting of resources. To simplify such patterns, Dijkstra proposed an abstraction called semaphores in 1965.

So far, we have studied locks, which are used for mutual exclusion, and condition variables, that are associated with conditions. Semaphores are usually used for "counting". Here are the semantics of a semaphore.

Semantics

A semaphore is defined by a stateful type (`struct sema;`) and three functions:

```
void sema_init(struct sema *, int val);
void P(struct sema *);   // also called wait()
void V(struct sema *);   // also called signal()
```

The names 'P' and 'V' stand for the dutch names of these operations. The semaphore type maintains an integer state, called `count`, which is a non-

negative counter.

The `sema_init` function initializes the counter to `val`, which must be a non-negative integer.

The `P` function decrements the counter by one, if it is strictly positive. If the counter is 0 (before the decrement operation), then the `P` function waits for the counter to become positive (greater than zero), before decrementing it.

The `V` function increments the counter by one. Also, it wakes up one process that is waiting on the semaphore to become positive.

Most importantly, all these operations, `sema_init`, `P`, and `V` are atomic with respect to each other.

Here is a sample implementation of a semaphore using locks and condition variables:

```
struct sema {
  int count;
  struct lock lock;
  struct cv cv;
};

void sema_init(struct sema *sema, int val) {
  sema->count = val;
  lock_init(&sema->lock);
  cv_init(&sema->cv);
}

void P(struct sema *sema) {
  acquire(&sema->lock);
  while (sema->count == 0) {
    wait(&sema->cv, &sema->lock);
  }
  sema->count--;
  release(&sema->lock);
}

void V(struct sema *sema) {
  acquire(&sema->lock);
  sema->count++;
  if (sema->count == 1) {
    notify(&sema->cv);
  }
  release(&sema->lock);
}
```

Notice that a semaphore maintains state. This is unlike a condition variable which maintains no state. For condition variables, the effect of a `notify()` operation is to wakeup any currently waiting threads. If a thread has not yet gone to sleep, `notify()` will have no effect. On the other hand, the `V()` operation (that is somewhat analogous to `notify()`) will increment the counter, even if no thread is currently waiting inside the `P()` operation (which is somewhat analogous to `wait()`). Let's see the implications of this using concrete examples.

Producer-consumer example with semaphores

The producer consumer problem involves ensuring that a producer does not produce to a full queue, and a consumer does not consume from an empty queue. This can be done by using two counters, one for the number of filled slots in the queue (`nchars`), and another for the number of empty slots in the queue (`nholes`). We use one semaphore for each counter, as follows.

```
char q[MAX];
int head = 0, tail = 0;
struct sema nchars, nholes;
struct lock mutex;

sema_init(&nchars, 0);
sema_init(&nholes, MAX);

void produce(char c) {
  P(&nholes);
  acquire(&mutex);

  //produce an element.

  release(&mutex);
  V(&nchars);
}

char consume(void) {
  P(&nchars);
  acquire(&mutex);

  //consume an element.
```

```
  release(&mutex);
  V(&nholes);
}
```

Notice that this code is much simpler, as it subsumes much of the counting and waiting logic within `P()` and `V()` functions. Because this pattern is common, semaphores are a convenient abstraction.

Resource allocation with semaphores

Semaphores are also a convenient abstraction for performing resource allocation. For example, if you have multiple threads that want to access a resource, and you only have N copies of that resource (e.g., N printers), then you could use a semaphore to restrict the number of concurrent accesses to N.

Here is an example where multiple threads may call the `print()` function, but you want to restrict the maximum number of concurrent prints to N.

```
sema_init(&nprinters, N);

void print(void) {
  P(&nprinters);
  //print command
  V(&nprinters);
}
```

This will ensure that the system has at most N simultaneous print requests. Also, as print requests get completed, they allow other print requests to be admitted.

Locks as semaphores

Locks can be trivially implemented as semaphores by initializing it to 1.

```
struct lock {
  struct sema sema;
};

void lock_init(struct lock *l) {
  sema_init(&l->mutex, 1);
}

void acquire(struct lock *l) {
  P(&l->sema);
}

void release(struct lock *l) {
  V(&l->sema);
}
```

Scheduling with semaphores

Semaphores are also used for enforcing thread schedules. Consider the following example, where one thread computes `x`, a second thread computes `y`, a third thread computes `z = f(x, y)`, and finally a fourth thread prints `z`. You may want to enforce an execution order (schedule) such that the computation of `z` happens only after the computation of `x` and `y`. Similarly, the print statement should execute only after the computation of `z`. This can be achieved by using semaphores `sx`, `sy`, and `sz`, initialized to zero. A one value in these semaphore counters will be used to indicate that the corresponding value has been computed.

```
thread1() {
  ....
  x = ...;
  V(sx);
}

thread2() {
  ....
  y = ...;
  V(sy);
}

thread3() {
  ....
  P(sx);
  P(sy);
  z = f(x, y);
  V(sz);
}

thread4() {
```

```
  P(sz);
  print z;
}
```

Monitors

All our abstractions for synchronization so far (locks, condition variables, semaphores) are implemented as types and functions on them. It is the responsibility of the programmer to ensure that the type objects are correctly used --- for example, a programmer must ensure that a lock is properly initialized before it is used; she should also ensure that a lock acquisition is always appropriately bracketed with a lock release. Notice that this is completely independent of the language in which the program is written.

Because concurrent programs are quite common, some languages provide intrinsic support for handling concurrency. One example of such support is a *monitor*. A monitor is defined as a type construct (e.g., class), which may declare private shared-memory objects and routines that access those objects. The monitor enforces mutual exclusion between the monitor routines, by using an "implicit lock". Here is an example of the producer-consumer queue implemented as a monitor:

```
monitor Q {
 char buf[MAX];
  int head = 0, tail = 0;

  void produce(char c) {
  while ((head + 1) % MAX == 0) {
      wait(&not_full);
    }
  q[head] = c;
    head = (head + 1) % MAX;
    notify(&not_empty);
  }

 char consume(void) {
    ...
  }
}
```

Notice that because the monitor guarantees mutual exclusion between its routines (only one thread can be inside a monitor routine at any time), we did not have to use a mutex ourselves. Instead, the compiler will implicitly generate a mutex for us. One way to implement monitors, is for the compiler to instrument each function with `acquire()` and `release()` calls on the implicit lock, before and after the routine respectively. Also, notice that the `wait()` function does not need the second argument; instead the second argument is automatically understood to be the monitor's implicit lock, i.e., a thread calling wait() will internally release the monitor's implicit lock before going to sleep.

Monitors make it simpler to write concurrent programs, and avoid common bugs like unbracketed acquires or releases. In Java, the "synchronized" keyword can be used to specify a monitor.