Additional Operations on Condition Variables

Some implementations offer additional operations on condition variables, which simplify some types of algorithms. For example, given that we have a condition variable `c`:

```
empty(c)
```

This function returns true if there are no processes waiting on `c`.

With wait and signal defined, delayed processes are normally awakened in the order in which they delayed, in other words, in first-in, first-out order. Some implementations offer a priority `wait`, where processes are placed in the queue according to some priority. The statement has the form

```
wait(c,rank)
```

where `rank` is an integer value representing a priority, the lower the value of `rank`, the higher the priority. This allows a simple implementation of the shortest job next algorithm to allocate a single resource:

```
monitor SJN {
   bool free = true;
   Condition turn;

   void request(int time) {
      if (free) free=false;
      else wait(turn, time);
   }

   void release(void) {
      if (empty(c)) free=true;
      else signal(turn);
   }
}
```

With this priority `wait`, it is sometimes useful to know the rank of the first process in the queue. Assuming that the queue is not empty and all the wait statements are priority wait statements,

```
minrank(c)
```

is a function that returns the rank of the first process delayed on `c`. If the queue associated with c is `empty`, the value of `minrank` is typically useless.

Typically in environments that support Signal and Continue semantics, there is an operation that allows a process to awaken all those waiting on a condition. This is called

```
signal_all(c)
```

Each awakened processes continues execution in the monitor at some future time. Of course, mutual exclusion is still guaranteed by the monitor. The effect of `signal_all` is the same as

```
while (! empty(c)) signal(c);
```

Examples

Here is an example where we implement a timer monitor. This is a utility that is often provided in operating systems to enable users to to do such things as periodically execute commands.

Our monitor has two operations, `delay(interval)` that processes call to wait `interval` time units. Another operation will be `tick()`, which we will assume is called by a process that is periodically awakened by a hardware timer. There could be other operations defined as well, for example to return the current time.

```
monitor timer {
   int tod=0;                        // Time Of Day, simply a counter
   Condition check                   // signaled when tod increases

   void delay(int interval) {
      int wake_time = tod + interval;
      while(wake_time > tod) wait(check);
   }

   void tick(void) {
      tod++;
      signal_all(check);
   }
}
```

We see a sample usage of `signal_all`, which forces each process to wake up, and check to see if it is time to continue, or go back to sleep. Of course in this particular case, this not very efficient, especially if processes are calling delay to wait for long periods of time.

By using priority `wait`, however, we can transform this solution into one that is equally simple, yet efficient. Essentially, we don't need to wake up each process every time, and we don't want to wake them up in the order in which they arrive:

```
monitor timer {
   int tod=0;                        // Time Of Day, simply a counter
   Condition check;                  // signaled when tod increases

   void delay(int interval) {
      int wake_time = tod + interval;
      if (wake_time > tod) wait(check, wake_time);
   }

   void tick(void) {
      tod++;
      while (! empty(check) && minrank(check) <= tod)
         signal(check);
   }
}
```

Here the use of `minrank` helps tremendously.

Implementing Monitors with semaphores

To implement a monitor with semaphores, we need to provide mutual exclusion and synchronization via condition variables.

What do we need to implement a monitor ? We will need a semaphore for mutual exclusion; for each condition variable in the monitor, one semaphore to block processes that need to wait for the condition, and a counter in order to know if there are processes waiting on the condition.

shared variables:

```
Semaphore e = 1
Semaphore c = 0
int nc = 0
```

Once we have these variables, we need to specify how to use them in order correctly implement a monitor.