0.1 Monitor

A Monitor is another powerful concurrent primitive. This essay will also use a simplified monitor from [TODO:reference] , which is described in the next paragraph.

Figure 1: Description

Usually, code (protected by the monitor) is wrapped inside a synchronized block. But before the process runs code inside the synchronized block, the process needs to check if others have obtained the monitor lock. Also, an exiting process needs to release the monitor lock so that other processes can continue. In Figure ref, Edge 1 represents a process entering the synchronized block, and edge 6 describes the transition of a process exiting the synchronized block. Inside the synchronized block, the process may call notify, notifyAll, wait function. A running process calling wait blocks execution and waits to be notified. When a running process notifies a waiting process, the waiting process becomes runnable and waits to obtain the monitor lock. Note that the waiting process can only continue execution after the current process exits its synchronized block, unlike that a process waiting to acquire the semaphore can immediately continue after another process raises the semaphore. All above transitions are represented by edge 2,3,4,5 in diagram ref, respectively.

In the CSP implementation of monitors, the monitor process does most of the work. Most monitor methods are implemented simply by sending an event to the monitor process, except for the wait. Recall that a running process calling wait first waits for notification and reobtains the monitor lock. So after sending the Wait event, the process waits for a notify event or a spurious wakeup event. Finally, the waiting process remains for a WaitEnter event, which gives the process the monitor lock. In practice, the wait function is often guarded with a while loop to prevent spurious wakeup. Again CSPM does not have a built-in while loop, and a generic while statement is implemented in continuation pass style. (Maybe leave this continuation pass style while loop in appendix?)

The monitor process has two states. When there is no running process, the monitor process can allow a free process to enter its synchronized block by synchronizing on a waitEnter event with the process. When there is a running process, the server process should respond to method calls from the running process, but not allow a process to obtain the monitor lock and proceed to its synchronized block. In either state, the monitor process can spuriously wake up a waiting process.

There is a corner case to highlight in the CSP implementation. In Java, if a running process calls notify method when there is no waiting processing, the function call does nothing. To make sure that the running process is not blocked in this case, when the running process calls notify method, the process send a notify to the server. Then if there are process waiting to be notified, the server process sends a waitNotify event to the waiting process.