Solutions for CP Exercise Class 5

1. By introducing a variable, *next*, indicating the smallest waketime of any waiting processes, the number of unnecessary wakeups may be considerably reduced. Using our notation:

monitor *Timer*

  var tod : $integer$:= 0;
    $next$: $integer := \infty$;
    $check$: $condition$;

  procedure $delay(interval : integer)$
    var $waketime$: $integer$;
    $waketime := tod + interval$;
    while $tod < waketime$ **do**
      if $waketime < next$ **then** $next := waketime$;
      $wait(check)$;

  procedure $tick()$
    $tod := tod + 1$;
    if $tod \geq next$ **then** $\{next := \infty;\ \ signal\_all(check)\}$

end

2. Both are used for letting a process wait, but they are indeed different:

- Semaphores may be used anywhere. Condition queues are associated with monitors and may only be used within these.
- The *wait* operation on a condition queue *atomically* realeases the monitor while putting the calling process on the queue.

  If a `P`-operation on a semaphore is made within a monitor operation, the process will just wait on the semaphore while the monitor's critical region remains locked.
- If there are no waiting processes, a `V` operation on a semaphore is remembered by incrementing the semaphore value, while signalling an empty condition queue has no effect.

3. Assuming that the variable `b` is protected by the monitor's critical region, this will lead to a deadlock since the critical region is not released during calls of `sleep()`.

4.

(a) A straightforward solution could be:

monitor *ChunkSem*

  var s : $integer$:= 0;
    $Empty$: $condition$;
    $NonEmpty$: $condition$;

procedure $P()$
  while $s = 0$ **do** $wait(NonEmpty)$;
  $s := s - 1$;
  if $s = 0$ **then** $signal(Empty)$

procedure $V()$
  while $s \neq 0$ **do** $wait(Empty)$;
  $s := s + M$;
  $signal\_all(NonEmpty)$

end

(b) For the monitor, the following safety invariant should hold:

$$I_1 \triangleq 0 \leq s \leq M$$

Provided $M \geq 1$, this readily follows from the intialization and the **while** tests.

(c) We now try to express that that no calls of the $P()$-operation are ever “forgotten”. This would be the case, if there remained processes but s was still positive. Thus we must require:

$$I_2 \triangleq waiting(NonEmpty) > 0 \Rightarrow s = 0$$

(d) If many processes are waiting on *NonEmpty* and M is small, most these processes will be unnecessarily woken up. In order to wake up only as many as can carry through the $P()$-operation, either a limited number of signals may be made or *cascaded wakeup* may be applied. Here we show the cascade solution:

monitor *ChunkSem*

  var $s : integer := 0$;
    $Empty : condition$;
    $NonEmpty : condition$;

  procedure $P()$
    while $s = 0$ **do** $wait(NonEmpty)$;
    $s := s - 1$;
    if $s > 0$ **then** $signal(NonEmpty)$
          else $signal(Empty)$

  procedure $V()$
    while $s \neq 0$ **do** $wait(Empty)$;
    $s := s + M$;
    $signal(NonEmpty)$

end

Now, the property I_2 does not necessarily hold at entry to a monitor operation, since there may be processes left on the queue while a woken process is waiting to get back to the monitor. Therefore the invariant will have to be weakened to talk about the state only when woken processes have been processed.

$$I_3 \stackrel{\Delta}{=} waiting(NonEmpty) > 0 \land woken(NonEmpty) = 0 \;\Rightarrow\; s = 0$$

[This can be formulated in a number of equivalent ways.]

5. Solution for Mon.3

A variable *sum* accumulates the contributions. To prevent new processes from interfering with the sharing of the loot, a flag *sharing* and a separate condition queue *hold* is used to hold these back while the sharing takes place.

```
monitor ShareLoot

  var count : integer := 0;
      sum : integer := 0;
      sharing : boolean := false;
      res : integer;
      c : condition;                  — Common waiting queue
      hold : condition;               — Wait for round to end

  function SYNC(v : integer) : integer
    var res : integer;
    while sharing do wait(hold);
    count := count + 1;
    sum := sum + v;
    if count = N then {sharing := true;  res := sum/N;  signal_all(c)};
    while ¬sharing do wait(c);
    count := count − 1;
    if count = 0 then {sharing := false;  sum := 0;  signal_all(hold)};
    return res

end
```

If used by more than N processes, the monitor should let them share their loot in groups of N without any interference.

The given solution works this way since the **while**-loop around *wait*(*hold*) prevents more than N to proceed to the counting phase. However, due to the *signal_all*(*hold*), the ordering of the processes may not be preserved and many processes may be woken unnecessarily. This can be improved upon by changing the last **if** statement to signal only N times:

```
    if count = 0 then { sharing := false;
                        sum := 0;
                        for i in 1..N do signal(hold)
                      };
```

6. Since both waits recheck their conditions, the solutions shown in point **5.** is robust towards *spurious wakeups*.

7. Since the two queues in the solution in point **5.** are not active at the same time, it is possible to implement the monitor in Java using the the single Java waiting queue for both waiting points.

```
class ShareLoot {

  int count = 0;
  int sum = 0;
  int res:
  boolean sharing = false;

  public synchronized int sync(int v) {
    int res;
    while (sharing) try {wait();} catch (Exception e) {};
    count++;
    sum = sum + v;
    if (count == N) { sharing = true;  res := sum/N;  notifyAll(); }
    while (! sharing) try {wait();} catch (Exception e) {};
    count--;
    if (count == 0) {
      sharing = false;  sum = 0;
      notifyAll();    // wake up any thread ready for next round
    }
    return res;
  }
}
```

As the Mon.3 solution, this one allows for more than N processes using the monitor. However, due to the lack of ordering in the Java waiting queue, a limited notification will not ensure preservation of ordering for this solution