Solutions for Exercises, Week 8

1. Solution for Mon.5

(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 ≠ 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:

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

This follows from the initialization, the fact that $s = 0$ when waiting on *NonEmpty*, and the flushing of *NonEmpty*, when s is incremented.

(d) If many processes are waiting on *NonEmpty* and M is small, most of 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 taken the woken processes into account. Due to the cascade, at least one process will be woken as long as $s > 0$. This may be expressed as:

$$I_3 \triangleq waiting(NonEmpty) > 0 \Rightarrow s = 0 \vee woken(NonEmpty) > 0$$

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

For a solution using limited signalling (awakening up to M processes), the invariant should express that enough processes have been woken up:

$$I_4 \triangleq waiting(NonEmpty) > 0 \Rightarrow s \leq woken(NonEmpty)$$

(e) Since both waits recheck their conditions, the solutions shown in (d) is robust towards *spurious wakeups*. Also the invariants have been formulated with inequalities allowing for an spontaneous increase of $woken(NonEmpty)$.

(f) Since we have two waiting conditions the standard solution is to used a mixed condition queue and use a covering condition:

```
class ChunkSem {

  int s = 0;

  public synchronized void P() {
    while (s==0) try {wait();} catch (Exception e) {};
    s--;
    notifyAll();
  }

  public synchronized void V() {
    while (s!=0) try {wait();} catch (Exception e) {};
    s = s + M;
    notifyAll();
  }
}
```

However, since all calls of $P()$ are woken up when s becomes positive, only $V()$ operations can be waiting when $s > 0$. Aside: This may be formally expressed by an invariant:

$$I_5 \triangleq s > 0 \Rightarrow waiting_{P()}() = 0$$

Therefore, the signalling in $P()$ needs only be done when the condition for $V()$ is true and only has to awake a single thread.

```
public synchronized void P() {
  while (s==0) try {wait();} catch (Exception e) {};
  s--;
  if (s==0) notify();
}
```

An attempt to use the cascade solution will render both $P()$ and $V()$ calls waiting in the condition queue and hence will not work.

2. Solution for Mon.6

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** := ∞;
$check$: *condition*;

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

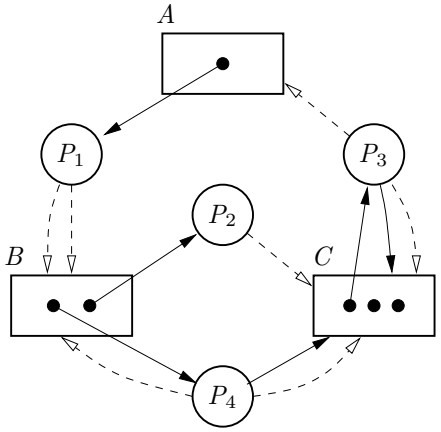
procedure *tick*()
$tod := tod + 1$;
if $tod \geq next$ **then** $\{next := \infty;$ *signal_all*(*check*)$\}$

end

Here *integer** denotes the set of integers extended with ∞ larger than any integer.

3. Solution for CP Exam December 1998, Problem 4

Question 4.1



Question 4.2

(a) Finishing processes satifying their maximum demands:

Available			*Can be finished*
A	B	C	
0	0	2	P_2
0	1	2	P_4
0	2	2	P_1
1	2	2	P_3
1	2	3	

Since a sequence exists in which all the processes can have their maximal resource demands satisfied, the situation is *safe*.

(b) Even though P_4 is granted a C-instance, the above sequence is still possible and the situation is still safe. Thus, P_4 may be granted a C-instance according the banker's algorithm.

4. Solution for Silberschatz, Galvin & Gagne Exercise 7.11

For a sytem with m inscances of a resource type, a deadlock situation is characterized by a number of processes that are requesting more instances while holding some already, but no more instances are available.

A process P_i can request more instances only if it has not yet reached is maximal claim MAX_i. The maximal number of instances n processes may have reserved without having reached their maximum claim (and thereby be able to finish) is given by:

$$\sum_{i=1}^{n}(MAX_i - 1) = (\sum_{i=1}^{n} MAX_i) - n = MAX - n$$

Thus, no deadlock can occur if this number is less than the number of available instances m:

$$MAX - n < m$$

or

$$MAX < n + m$$

It is assumed that all processes need several instances, ie. $MAX_i > 1$ for all i and, of course, that $MAX_i \leq m$.