Exam 021025 HT2002 with solution outlines.

Question 1. (a) What is meant by *synchronous message passing* *(2p)*

(b) A busy-wait loop might be disasterous if the scheduler is unfair. Explain why. *(2p)*

(c) What do the Linda operations *in* and *out* do? *(2p)*

(d) For signal-and-continue style monitors (e.g. Java synchronised methods), explain the potential problem with code of the form

```
if (resource_not_ready) then {
    wait(available)
};
use_resource;
```

[Note for students who took the course before this fall: "signal-and -contiunue" is also called "resume-and-continue". In the psudocode, replace the instruction `wait(available)` *by the PascalFC statement* `delay(avialable)`*.]*

(2p)

Question 2. Implement a binary semaphore using a protected object. The `V` operation should do nothing if the value of the semaphore is already 1. You may write either Ada code, a PascalFC resource, or some suitable psudocode.

[Note for students who took the course before Fall 2002: the "V" operation is called "signal" in PascalFC].

(6p)

```
# Assume initialised to zero.
# MPD psudocode

protected semaphore

entry p()
proc v()   # Non blocking

protected body semaphore
  int value = 0

  entry p() when value == 0 {
    value = 0
  }

  proc v() {
    value = 1
  }
end semaphore
```

Question 3. The kernel of the UNIX operating system traditionally implements two operations called `sleep()` and `wakeup()`. Simplifying slightly, what these operations do it the following: a call to `sleep()` always blocks the calling process. A call to `wakeup()` wakes up all the processes who are currently blocked because they called `sleep()`.

Give an implementation of these operations *using message passing* (MPD op/in or Ada-style rendezvous using Ada or PascalFC) for synchronisation. *(10p)*

```
op sleeping()

process sleepserver()
{ while (true) {
  in
      sleep()
      -> forward sleeping()
  []
      wakeup()
      -> while (?sleeping > 0){receive sleeping()}
  ni
  }
}
```

Question 4. A Robot maintenance house offers minor repair and maintenance services for factory robots. The house fixes robots for two rival companies, "Vaab" and "Solvo". Robots automatically leave factory and go to the house when they need maintenance.

The question is to write a monitor (PascalFC, Java object or MPD psudocode) to simulate the house controller software which controls when robots enter the house.

Robots call the following monitor procedures:

- `Enter(...)` when they want to enter the house, and
- `Leave(...)` when they finally leave the house.

You may add your own parameters to these calls as you wish. The Enter call is potentially blocking, since you are required to ensure that:

- there are never more than 20 robots in the house at any time, and
- whenever there are robots *of both kinds* in the house, the number of *Vaab* robots is never more than double the number of *Solvo* robots, and vice versa. (This helps prevent fights between the robots from the rival companies!).

You may assume that robots who enter will eventually leave, that the scheduler is fair, and that any queues in the implementation of the programming language are FIFO. Your solution should not make robots wait unnecessarily, but does not have to be starvation free. If your solution can lead to starvation then you must explain how it can occur. *(14p)*

```
# MPD psudocode, Java style monitors
# (notify and continue)

monitor Robocop

type robotype = Enum(Vaab; Solvo)

op enter(robotype), leave(robotype)

[low(robotype):high(robotype)] int count = (0, 0)

procedure other(robotype r) returns robotype o{
if (r == Vaab) {o = Solvo} else {o = Vaab}
}

procedure dangerous(robotype r) returns bool isdanger {
  int us = count[r]
  int others = count[other(r)]

  isDanger =
     us + others == 20 || ( others > 0  and  2 * others <= us)
}

condition change

proc enter(r) {
  while (dangerous(r))
  { wait(change)}
  count[r]++
  signalAll(change)
}

proc leave(r) {
  # assume that leaving doesn't violate the constraint.
  count[r]--
  signalAll(change)
}
```

Two reasons for unfairness:
1. signalAll puts you behind other calls on the
boundary queue, so starvation for a single robot is always possible
(however unlikely) if new calls to "enter" always steal the signal.

2. If there are 3 or more Vaabs in the factory then a Solvo cannot enter. If Solvo's arrive often enough then there will be never less than 2 in the factory and so the Vaabs starve.

Question 5. (a) Explain briefly what is meant by *barrier synchronisation*. *(2p)*

(b) Show how barrier synchronisation for n processes can be implemented using semaphores. The n processes should be able to repeatedly use the barrier. Note: you may use an extra "coordinator" process if it is suitable for your approach.

(10p)

```
## One binary semaphore per process, initially zero.
# Assume each process has a unique id [1..n].
# One general semaphore for counting arrived processes, initially zero.

sem s[n] = [n]0

sem done = 0

# A process wishing to perform a barrier sync. must call arrive:

procedure arrive(int i){
   V(done) # signal arrival
   P(s[i]) # wait to be released
}

process coordinator {
   while (true){
      for [i = 1 to n] { P(done) } # wait for n procs to arrive
      for [i = 1 to n] { V(s[i]) } # release all n procs
   }
}
```