p494

MR

2 Comparative Programming languages 2004

(a) Briefly describe the concept of coroutines as provided in BCPL and outline the effect of the library functions `createco(f, size)`, `deleteco(cptr)`, `callco(cptr, val)`, and `cowait(val)`. [6 marks]

(b) Discuss the relative merits of BCPL coroutines to the use of threads such as those provided in Java. [6 marks]

(c) Outline the overall design and organisation of a BCPL program to perform discrete event simulation using coroutines to implement the simulated activities. Concentrate on the design of the simulation event loop, the organisation of the priority queue and what functions you would provide to simplify the implementation of the activities. It would probably be sensible to adopt a programming style similar to that used in Simula 67. You should hold simulated time as a global (integer) variable. [8 marks]

ANSWER NOTES:

(a) BCPL coroutines are bookwork. Points to mention:

A coroutine requires its own runtime stack whose base contains some system information. Coroutines all run in the same address space. They are not pre-emptive, only giving up control when they wish to. When control passes from one coroutine to another a value is also passed. Createco creates a coroutine with a given main function and stack size. Initially leaving it suspended in the cowait call in:

```
c := f(cowait(c)) REPEAT
```

Deleteco will delete a specified coroutine, callco will transfer control to a specified coroutine passing a specified value. The current coroutine is first suspended. The called coroutine has a parent link set to point to the calling coroutine. This is used by cowait to identify which coroutine to pass control to.

(b) Threads are pre-emptive and can run simultaneous on multi-processor machine. Their scheduling is more costly involving complete register dumps and the use of priorities. Like coroutines they run in the same and so can access shared memory but unlike coroutine must usual use sychronisation primitive to ensure access and modification of shared variables is safe. This adds to the cost and complexity of their use.

(c) The priority queue can be logically a list of items containing a time, a coroutine and a value to pass to the coroutine when is is activated. The items are in increasing time order. The event loop dequeues the first item from the priority queue sets the global time variable to its and calls the specified coroutine using callco. The priority queue could be organised as a (heapsort style) heap so that the cost of insertion and removal is O(log n) rather than typically O(n). It would be useful to provide functions such as waitfor(ticks) to suspend a coroutine for a specified number of ticks of simulated time, waituntil(time) to wait until a specified simulated time, die() to cause the current coroutine to commit suicide. Random number generators for various statistical distributions should be provided. There should be a mechanism to cause the event loop to know when to stop. If could use a global flag that is set to FALSE to stop the loop. An event could be placed in the priority queue to fire at the simulated time that the simulation should stop. Its coroutine should set the flag to FALSE thus causing the simulation to cease.