Chris Bohlman

CS422: HW 04

Problem 1: Exercise 7.2, page 353

Consider a filter process **Partition** having the following specifications. **Partition** receives unsorted values from one input channel **in** and sends the values it receives to one of two output channels, **out1** or **out2**. **Partition** uses the first value **v** it receives to partition the input values into two sets. It sends all values less than or equal to **v** to **out1**; it sends all values greater than **v** to **out2**. Finally, **Partition** sends **v** to **out1** and sends a sentinel EOS to both **out1** and **out2**. The end of the input stream is marked by a sentinel EOS.

a.) Develop an implementation of **Partition** in pseudo-code. State predicates specifying the contents of the channels, then develop the body of Partition.

```
chan in, out1, out2


process Partition {
        var v = 0;
        bool readInitialValue = false;
        bool reading = true;
        while (reading) {                       // While process is still reading from input streams
                receive(value);
                if (value == EOS) {             // If sentinel EOS is read in
                        reading = false;
                        send toOut1(v);
                        send toOut1(EOS);
                        send toOut2(EOS);
                }
                else {                                          // Continue reading values
                        receive(value);
                        if (readInitialValue) {                 // If initial value has been read in
                                if (value <= v) {       // If read value is less than or equal to v
                                        send toOut1(v);
                                } else {                // If read value is greater than v
                                        Send toOut2(v);
                                }
                        } else {                                // If initial value has not been read in
                                v = value;
                                readInitialValue = true;
                        }
                }
        }
}
```

b.) Show how to construct a sorting network out of **Partition** processes. Assume that there are **n** input values and that **n** is a power of **2**. In the worst case, how big does the network have to be? (This is not a good way to do sorting; this is just an exercise!)

Have a Partition process for every single value of n.
Have the first process start as normal and send values across channels
Have next partition process accept values from out1 first, then accept values from out2. This will ensure values are received while partitioned correctly.
Continue this until last Partition process finishes.

Worst case network size = n processes
I know this is a god-awful solution but it's what I got.

Problem 2: Dining Philosophers:
Develop a solution to the Dining Philosophers problem using RPC. Each philosopher, when ready to eat, will call:
chop1 = getChop(myId);
chop2 = getChop(myId);
... eat for a little while ...
releaseChop(myId, chop1);
releaseChop(myId, chop2);
The server should only give philosophers the two chopsticks that are nearest to them (as in the standard description of the Dining Philosophers problem). In addition, the server should insure that deadlock will not happen.

Synchronization code not shown

```
int numPhilsophers; //Assumed to be known by server
chop chopsticks[n];

proc getChop(int myId) {
        if (myId != numPhilosophers -1) {
                if (chop[myId].owner == myId) {                 //First chopstick already gotten
                        while (chop[(myId+1)%numPhilosophers].free != true) {
                                …implement mechanism for getting the chopstick wanted when
                                it's free…
                        }
                        chop[(myId+1)%numPhilosophers].free == false;
                        chop[(myId+1)%numPhilosophers].owner = myId
                        return chop[(myId+1)%numPhilosophers];
                } else {
                        while (chop[myId].free != true) {
                                …implement mechanism for getting the chopstick wanted when
                                it's free…
                        }
                        chop[myId].free == false;
                        chop[myId].owner = myId
                        return chop[myId];

        } else {                                                // Get other chopstick first
                if (chop[(myId+1)%numPhilosphers].owner == myId) {
                        while (chop[myId].free != true) {
                                …implement mechanism for getting the chopstick wanted when
                                it's free…
                        }
                        chop[myId].free == false;
                        chop[myId].owner = myId
                        return chop[myId];
```

```
        } else {
                while (chop[(myId+1)%numPhilosophers].free != true) {
                        …implement mechanism for getting the chopstick wanted when
                        it's free…
                }
                chop[(myId+1)%numPhilosophers].free == false;
                chop[(myId+1)%numPhilosophers].owner = myId
                return chop[(myId+1)%numPhilosophers];
        }
}

proc releaseChop(int myId, Chopstick chop) {
        chop.free = true;
        chop.ower = NULL;
        // signal any other waiting philosophers
}
```

Problem 3: Exercise 7.7, page 354

Develop an implementation of a time-server process. The server provides two operations that can be called by client processes: one to get the time of day and one to delay for a specified interval. In addition, the time server receives periodic "tick" messages from a clock interrupt handler. Also show the client interface to the time server for the time of day and delay operations.

```
Module TimeServer
        op get_time() returns int;
        op delay(int interval);

body
        int time_of_day = 0;
        sem delay = 1;
        queue delay_times;
        sem delay_proccess[n] = {1};

proc get_time() {
        return time_of_day;
}

Proc delay(int interval) {
        P(delay);
        Int waketime = tod + interval;
        Delay_times.enqueue(waketime, clientId);
        V(delay);
        P(delay_process[clientId]);
}

Process clock {
        While(true) {
                P(delay);
                time_of_day += 1
                While (time_of_day >= delay_times.peek()) {
                        Object = Delay_times.dequeue();
                        V(delay_process[object.clientId]);
                }
                V(delay);
        }
}
```

Clients can call get_time as well as delay and be thread safe.

Problem 4: Exercise 7.9 part a.), page 355

Two kinds of processes, **A**'s and **B**'s, enter a room. An **A** process cannot leave until it meets two **B** processes, and a **B** process cannot leave until it meets one **A** process. Each kind of process leaves the room — without meeting any other processes — once it has met the required number of other processes.

a.) Develop a server process to implement this synchronization. Show how **A** and **B** processes interact with the server. Assume asynchronous send and synchronous receive.

```
Process Server {
        Queue valuesA;
        Queue valuesB;
        While (true) {
                Receive(process);
                If (process.type == A) {
                        valuesA.enqueue(process);
                } else {
                        valuesB.enqueue(process);
                }
                If (valuesA.size() > 0 && valuesB.size()%2 == 0) {
                        //Processes have met in a room
                        ProcessA_dequeue = valuesA.dequeue();
                        ProcessB_dequeue1 = valuesB.dequeue();
                        ProcessB_dequeue2 = valuesB.dequeue();
                        ProcessB_dequeue1.send(ProcessA_dequeue);
                        ProcessB_dequeue2.send(ProcessA_dequeue);
                        ProcessA_dequeue.send(ProcessB_dequeue1);
                        ProcessA_dequeue.send(ProcessB_dequeue2);
                }
        }
}
```