Universitetet i Oslo
Institutt for Informatikk
Research group for Reliable Systems (PSY)
Silvia Lizeth Tapia Tarifa, Riccardo Sieve

INF 5170: Models of Concurrency

Fall 2026 — **Group Session 4** — 23.09.2026

Topic: Channels and Message Passing

Exercise 1 (Partition filter) ([1, Exercise 7.2a]) Consider a filter process `Partition` having the following specifications. `Partition` receives unsorted integer 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` and 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`.

Develop an implementation of `Partition`. First, give predicates specifying the contents of the channels, then develop the body of `Partition`.

Solution: [of Exercise 1] Concerning the predicate: A good inspiration can be the filter we developed in the lecture. The filter here handles, as the one in the lecture, a *finite* input stream that makes it slightly simpler as we can formulate a "post-condition" (but the splitting would also work on infinite streams).

Let n be the number of proper values sent, including `v`, but excluding `EOS`. Let "in" be the list of values sent over channel `in`, and let in$[i]$ be the i's value of in. Similarly, let "out1" and "out2" be the list of values sent over the channels `out1` and `out2`, and let out1$[i]$, respectively out2$[i]$, be the i'th value of out1, respectively out2. Let m be the number of proper values (excluding `EOS`) sent to `out1`. Then the following predicate holds for the channels *after termination of the program.*

$$\begin{aligned}&\texttt{empty(in)} \wedge \text{in}[1] = \texttt{v} \wedge \text{out1}[m] = \texttt{v} \wedge \text{out1}[m+1] = \texttt{EOS} \wedge \text{out2}[(n-m)+1] = \texttt{EOS}\\ &\wedge \forall 1 \leq i \leq m,\ \forall 1 \leq j \leq (n-m)\colon\ \text{out1}[i] < \text{out2}[j]\\ &\wedge \forall 1 \leq i \leq n : (\exists j 1 \leq j \leq m\colon\ \text{in}[i] = \text{out1}[j]) \vee (\exists 1 \leq j \leq (n-m)\colon\ \text{in}[i] = \text{out2}[j])\end{aligned}$$

The solution is implemented in Listing 1. This solution fails if the stream is empty or just contains EOS. Proper defensive programming would need to check for those conditions and react appropriately.

Exercise 2 (Readers/writers & server with asynchronous message passing) ([1, Exercise 7.6]) Consider the readers/writers problem defined in the lecture: There are reader and writer processes that share access to a database/shared data structure. Readers only read from the database and writers write to the database. A writer needs exclusive access to the database, so no other reader or writer should access the database at the same time. On the other hand, many readers can access the database at the same time.

Develop a server process to implement access to the database that the readers and writers access. Show the reader and writer interfaces to the server. The processes should interact using asynchronous message passing.

Listing 1: Partition filter

```
1  chan in(int), out1(int), out2(int);
2
3  process Partition {
4    int v, curVal;          # the first value and the current value
5    receive in(v);
6    receive in(curVal);
7
8    while (curVal ≠ EOS){
9      if (curVal ≤ v)
10       send out1(curVal);
11     else
12       send out2(curVal);
13     receive in(curVal);
14   }
15   send out1(v);
16   send out1(EOS);
17   send out2(EOS);
18 }
```

Solution: [of Exercise 2] For solving this exercise, we can take inspiration from the monitors solution to the readers/writers problem presented in the lecture. In the monitor solution, we used 4 procedures, 2 for reading, and 2 for writing. For the same reasons, we use 4 different "classes" of messages here, two client kinds and two operation kinds, see Listing 2.

As discussed in the lecture, there is one reply channel *per client*. However, there is one shared channel for requests, where each client sends their id for identification. Being a "server" to the different kinds of requests, the process is implemented as a big while-loop, reading one channel message after the other; that is a standard design. Also standard is that the body of the while loop is a big case construct, making a distinction between all kinds of message that are possible. Here we have 4 cases, represented by the nested if-then-else construct.

Another point to consider are the internal data structures. As we learnt from the monitors (and semaphores, etc.), not only must the "server" make processes wait, but it should also treat the waiting processes "fairly" when the waiting period is over. That is, it should treat the processes in a first-come-first-serve (FCFS) manner.

Making a client "wait" is simple, we just make sure that the client, after sending a message, waits on receiving an *acknowledgment* message. That is the very general pattern of turning asynchronous communication into synchronous communication: sending (non-blocking) and immediately waiting to receive an acknowledgment (receiving is blocking). For waking the process up again (by sending the acknowledgment at the appropriate time), we need to implement the fair FCFS behavior ourselves with appropriate data structures; in this case a queue. Besides that, we keep track of the numbers of readers and writers.

The solution could be shorter, but this one is "fair" (in the sense mentioned). One property of readers/writers, namely that a constant flow of readers may prevent writers forever from writing is not achieved in the proposed solution.

Exercise 3 (Savings account) ([1, Exercise 7.8]) A savings account is shared by several people. Each person may deposit or withdraw funds from the account. The current balance in the account is the sum of all deposits to date minus the sum of all withdrawals to date. The balance must never become negative.

Develop a server to solve this problem and show the client interface to the server. Clients make two kinds of requests: one to deposit `amount` dollars and one to withdraw `amount` dollars. The withdraw operation must delay until there are sufficient funds. Assume that `amount` is positive. (Remember that we had the same problem for monitors.)

Listing 2: Readers/writers with asynchronous message passing

```
type client_kind = enum(READER, WRITER);
type op_kind = enum(REQUEST, RELEASE);
chan request(int clientID, client_kind clkind, op_kind opkind);
chan reply[n] (bool reply);

process rwserver{
  queue readers, writers;
  int nr := 0, nw := 0;      # number of readers/writers
  int clientID, clientID2;
  client_kind clkind;
  op_kind opkind;

  while (true){
    receive request(clientID, clkind, opkind);
    if (opkind = REQUEST){
      if (clkind = READER){
        if (nw = 0 and empty(writers)){
          nr++; send reply[clientID](true);}
        else{insert(readers, clientID);}
      }
      else{              # clkind = WRITER
        if(nr = 0 and nw = 0){
          nw++; send reply[clientID](true);}
        else{insert(writers, clientID);}
      }
    }
    else{                # opkind = RELEASE
      if(clkind = READER){
        nr := nr - 1;
        if(nr = 0 and !empty(writers)){
          remove(writers, clientID2);
          nw++;
          send reply[clientID2](true);
        }
      }
      else{              # clkind = WRITER
        nw := nw - 1;
        if(!empty(readers)){
          while(!empty(readers)){
            remove(readers, clientID2);
            nr++;
            send reply[clientID2](true);
          }
        }
        else if (!empty(writers)){
          remove(writers, clientID2);
          nw++;
          send reply[clientID2](true);
        }
      }
    }
  }
}
process client[i=1 to n]{
  client_kind clkind;
  bool reply;

  while(true){
    send request(i, clkind, REQUEST);
    receive reply[i](reply)      # wait for reply from server
    # use the server
    send request(i, clkind, RELEASE);
  }
}
```

Solution: [of Exercise 3] The first to fix is the pattern of the solution and, connected to that, the interface (both are not independent). Since we are in the message passing world, we are dealing with channels, so the interface will be the channels that the "account server" offers, see Listing 3.

Listing 3: Savings account

```
type opKind = enum(DEPOSIT, WITHDRAW);
chan request(int clientID, opKind opkind, int amount);
chan reply[n](bool reply);
bool done := false;

process accountManager{
  queue pending;
  int clientID, clientID2, amount;
  opKind opkind;
  int balance := 0;

  while(true){
    receive request(clientID, opkind, amount);
    if(opkind = WITHDRAW){
      if(balance >= amount){
        balance := balance − amount;
        send reply[clientID](true);}
      else{insert(pending, (clientID, amount));}}
      else{                                          # opkind = DEPOSIT
      balance := balance + amount;
      while(!empty(pending) and !done){
        remove(pending, (clientID2, amount));
        if(balance < amount){
          insertFirst(pending, (clientID2,amount));
          done := true}
        else{
          balance := balance − amount;
          send reply[clientID2](true);}
      }
      done := false;
    }
  }
}

process client[i=1 to n]{
  int amount;
  opKind opkind;
  bool reply;

  while(true){
    send request(i, opkind, amount);
    receive reply[i](reply)      # wait for reply from server
    # money was deposited or withdrawn
  }
}
```

As always, one has a range of options how many channels one wants. The first question we should ask is: Which communication directions are there? The client(s) send to the server, the question is: Is there a need for back-communication? Generally, there can be two reasons for back channels: *communication* (giving back a return value) and synchronization (the client waits for acknowledgment that it is safe to proceed). If one gives back a value, then that most often combines communication and synchronization. The problem here has both (and we have only asynchronous channels).

The general design of the server resembles closely to the one readers/writers from Exercise 2 on this sheet.

Exercise 4 (Printers) ([1, Exercise 7.10]) Suppose a computer center has two printers, `A` and `B`, that are similar but not identical. Three kinds of client processes use the printers: those that must use `A`, those that must use `B`, and those that can use either `A` or `B`.

Develop code that each kind of client executes to request and release a printer, and develop a server process to allocate the printers. Your solution should be fair assuming that a client using a printer eventually releases it.

Solution: [of Exercise 4] The overall design is more or less the same as for the previous exercises. We have basically 3 "synchronization needs", one for each client. We may therefore add three queues, see Listing 4.

References

[1] G. R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

Listing 4: Printers

```
type opKind = enum(REQUEST, RELEASE);
type prNames = (A, B, BOTH);
chan request(int clientID, opKind opkind, prNames printer);
chan reply[n] (prNames printer);

process printerAccess{
  queue waitA, waitB, waitAB;                    # 3 conditions, 3 queues
  prNames chosenWhenA := A, chosenWhenB := B;  # which queue was chosen the
                                                 # last time a printer was ready
  int clientID, clientID2;
  opKind opkind;
  prNames printer;
  bool availA := true, availB := true;           # are the printers available?

  while(true){
    receive request(clientID, opkind, printer);
      if(opkind = REQUEST and printer = A){
        if(availA) {availA := false; reply[clientID](A); }
        else {insert(waitA, clientID);}
      }
      else if(opkind = REQUEST and printer = B){
        if(availB) {availB := false; reply[clientID](B); }
        else {insert(waitB, clientID);}
      }
      else if(opkind = REQUEST and printer = BOTH){
          if(availA) {availA := false; reply[clientID](A); } # bias towards A
          else if(availB) {availB := false; reply[clientID](B); }
          else {insert(waitAB, clientID);}
      }
      else if(opkind = RELEASE and printer = A){
        if(empty(waitA) and empty(waitAB)){availA := true;}
        else if((chosenWhenA = BOTH and !empty(waitA)) or empty(waitAB)){
          chosenWhenA := A; remove(waitA, clientID2);
          send reply[clientID2](A);
        }
        else{
          chosenWhenA := BOTH; remove(waitAB, clientID2);
          send reply[clientID '](A);
        }
      }
      else if(opkind = RELEASE and printer = B){
        if(empty(waitB) and empty(waitAB)){availB = true;}
        else if((chosenWhenB = BOTH and !empty(waitB)) or empty(waitAB)){
          chosenWhenB := B; remove(waitB, clientID2);
          send reply[clientID2](B);
        }
        else{
          chosenWhenB = BOTH; remove(waitAB, clientID2);
          send reply[clientID2](B);
        }
      }
  }
}
process client[i=1 to n]{
  prNames possiblePrinter, usePrinter;
  bool reply;

  while(true){
    send request(i, REQUEST, possiblePrinter);
    receive reply[i](usePrinter)      # wait for reply from server
    # use the printer
    send request(i, RELEASE, usePrinter);
  }
}
```