Assignment 2

CIS3110/Operating Systems/W25
updated: January 29, 2025

CODING ASSIGNMENT

1 Modelling a Token Ring using Semaphores

This exercise will give you some experience working with semaphores, and designing systems to avoid deadlock.

2 Objective

Write a C program that uses multiple Unix processes communicating via a shared memory region to simulate a Token Ring network.

A Token Ring is one of the two most common types of Local Area Network. It consists of network of nodes (machines) cabled together in a ring topology. A token (small data packet) is passed around the ring to govern which node is given control of the ring.

When a node wishes to transmit a data packet on the ring, it must wait until the token arrives and then it grabs the token and transmits one or more data packets to the next machine. When done transmitting, the node passes the now-available token along. As it does this, it must accept the various bytes of data from its own packet (which will be returning on their cycle around the ring) and discard them.

When a node receives a packet it normally retransmits it to the next node on the ring. The only exceptions to this are

- when the packet is the token *and* the node wishes to transmit data packet(s) or
- when the packet is one the *current node placed on the ring*

When a node receives a data packet, the node must check to see if it is the node the packet is addressed to and, if so, saves it as a received packet *as it retransmits it*.

A token is just a small data packet, but with a flag (usually just one bit) near the beginning set to indicate it is a token instead of a normal data packet.

The basic algorithm for a node on the ring is shown here:

3 Basic Algorithm

When a packet is received:

- if the packet is the “available” token
  - if the node wishes to transmit data
    - send the data packet(s)
    - note that the “available” token needs to be passed along after the last bytes of the packet
  - else
    - send the token as is
- else
  - if packet is from this node
    - discard packet

* if token needs to be passed along
  - send token
* else
  - retransmit the data packet, noting if it was to this node

4 Notes on the Algorithm

Normally the token flag is a single bit near the beginning of a packet and a node retransmits the packet as it receives it with minimal delay, flipping the token flag bit to "grab" the token and transmit a data packet. This is done to limit the amount of memory required in the retransmission hardware.

For your simulation, you may retransmit data to the next node a byte at a time, instead of worrying about individual bits.

We will therefore use a whole byte to represent the token. Be sure to use a byte value that you will not send within an ASCII message!

Although the above algorithm gives you an outline of a node's actions, it does not describe the order that data has to be handled. The nodes must handle one byte at a time for send and receive. Keeping separate state variables for send and receive that described where in a packet they were (`TOKEN_FLAG`, `TO`, `FROM`, `LEN`, and `DATA`) will be useful.

You may assume the data packet is structured as follows:

```
struct data_pkt {
    char token_flag;        /* set to 0x01 for a token, 0x00 otherwise */
    char to;                /* set to the destination node # */
    char from;              /* set to the sender's node # */
    unsigned char length;   /* Data length 1<-->250 bytes */
    char data[250];         /* Up to 250 bytes of data */
};
```

For the simulation, **fork off** `N_NODES` **child processes**, one to simulate each of the `N_NODES` nodes, implementing the above algorithm. The "`struct data_pkt`" data/token packets should be passed between the nodes using **a single byte in the shared memory region, one for each edge of the ring** (*i.e.;* `N_NODES` interconnects). Semaphores should be used to co-ordinate sending/reception of the data/token packet(s).

The node processes should forward a byte received as soon as possible to the next node, except for the case where they are receiving a data packet they sent, which should not be forwarded, but should be replaced by the token.

If a node tries to transmit too much data before doing a receive, it is very easy to get a deadlock situation. A non-optimal solution will receive a byte for every byte sent. This is a simpler implementation that you should do first. At most 1.5 marks out of 10 will be assigned to getting it to send more aggressively, so only do that if you want the challenge and have the time.

Due to the ring topology, node *k* can only receive packets from node `(k-1)% N_NODES` and send to node `(k+1)% N_NODES`. The token has a data length of 0 and the to and from fields are ignored in it.

Somehow, you will also have to introduce a token to the ring. You may want to do this by having the process simulating node `0` do this when it started up.

Each of the `N_NODES` children will have to somehow know their node number, which should go from `[0 ... N_NODES -1]`. Along with the `N_NODES` **one-byte shared memory buffers**, there should also be a "`struct data_pkt`" in the shared memory area **for each node** which will be filled in by the parent process to indicate that that node has a packet to send. When sent, the node should **notify the parent process that it was sent via semaphore(s)**. The data packet has from 1 to 250 data bytes in it, as indicated by the "`length`" field. The node processes **should also increment sent/received counters in the shared data region**, that will be printed out by the parent process at the end of the simulation run.

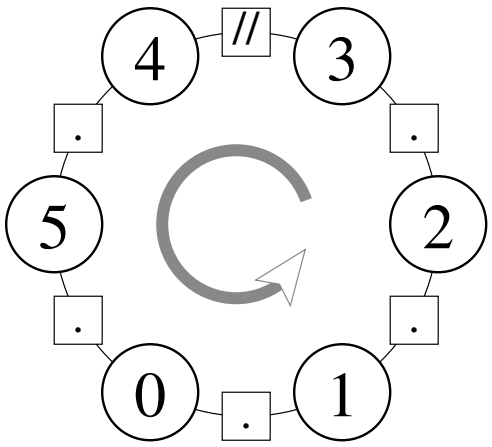

Figure 1: Empty Ring – A ring shown with alternating round and square nodes. A circular arrow is shown in the centre indicating traversal of the ring in a counter-clockwise direction. The nodes are numbered beginning at Node 0, and ending at Node 5. The boxes between the nodes all have a single . character in them with the exeption of the box between nodes 3 and 4, which has the mark //.

The parent process should generate packets for nodes to send at random and they should be of random size, in the range [1...250] bytes. At most there can be one outstanding packet to be sent for each node and the simulation should end after SIM_PACKETS packets have been successfully transmitted on the ring, with all children terminated. At the end, it should print out the summary sent/received statistics for all the child processes.

There is code provided in /home/courses/cis3110/assignments/CIS3110-W25-A2-code to get you started, plus you will find instructive examples in the /home/courses/cis3110/examples/semaphore_process_demo.c and /home/courses/cis3110/examples/semaphore_demo2 programs.

5 Processing States

The figure below shows various states as a message containing the string “data” as its value moves around the ring.

- *Figure 1: Empty Ring* An empty token ring. The “available” token (marked with “//”) has just passed node 3. “Empty” regions of the ring are shown with “.” characters.
- *Figure 2: Begin Send* Node 3 has a four-byte message to send to node 1, and has just begun to send by switching the token value from “//” to “++” and has attached the first byte of the packet (destination address 1).
- *Figure 3: Mid Send* Node 3 is midway through sending the message, and the first bytes are arriving at node 1, which has just discovered that a message is intended for it, but does not yet know who it is from, or how long it will be.
- *Figure 4: End Send* Node 3 has sent the last byte of the message, and so sends the “available” token (“//”) so the next message (likely from another node) can be attached. Bytes arriving at node 3 during the sending must be part of the message that node 3 has just been sending, so they are thrown away, until the token comes around again.

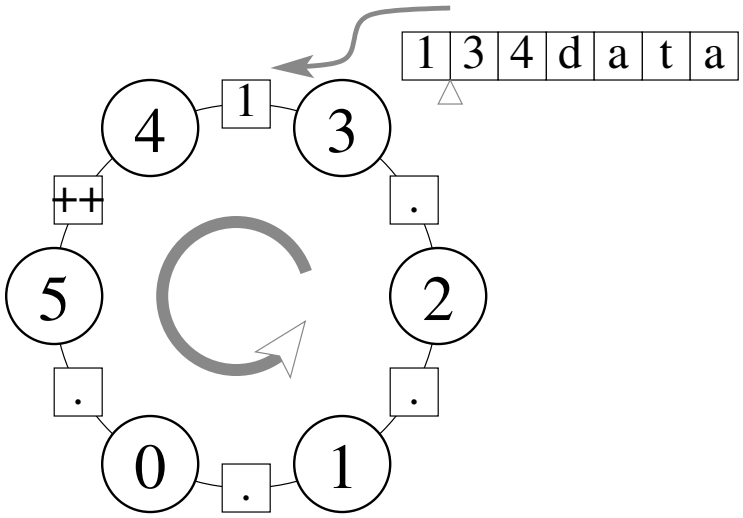

Figure 2: Begin Send – The same ring structure is shown. A buffer of input symbols containing “1 3 4 d a t a” is shown with an arrow leading from this past node 3 and into the box previously marked // that lies between node 3 and 4. This box is now labelled “1”. A triangle marker is shown under the buffer of input symbols immediately after the “1” symbol in the buffer The box between nodes 4 and 5 is now labelled ++.

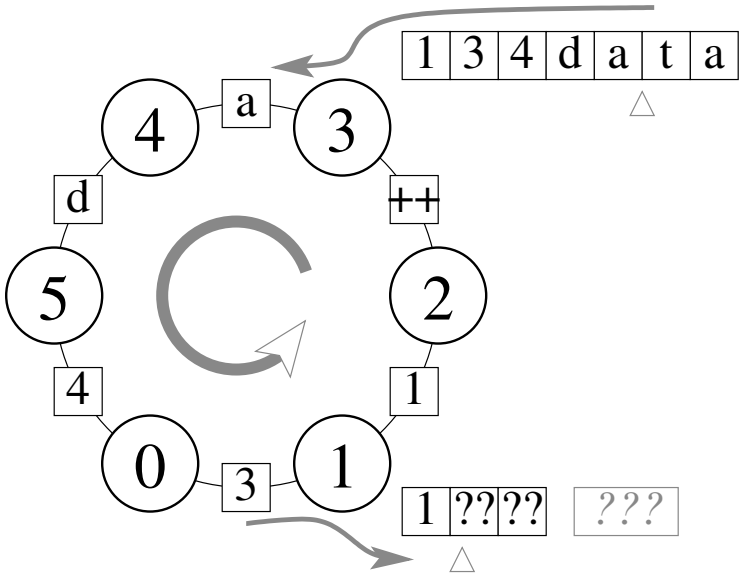

Figure 3: Mid Send – The same ring structure is shown. The triangle marking the location within the buffer of symbols is now positioned after the first “a”, with two symbols (“t” and “a”) after it. The arrow connecting the buffer to the ring structure points at the same box on the ring as before. This box now contains an “a”, and the boxes as we proceed counter-clockwise around the ring read “d”, “4”, “3”, “1” and “++”. The “++” symbol is in the box between nodes 2 and 3. A second arrow leads from the ring into an output buffer. This arrow originates in the box between nodes 0 and 1 and leads to a buffer showing a “1” in the first postion, followed by buffer locations filled with “??” symbols. A triangle marker is shown immediately after the first position.

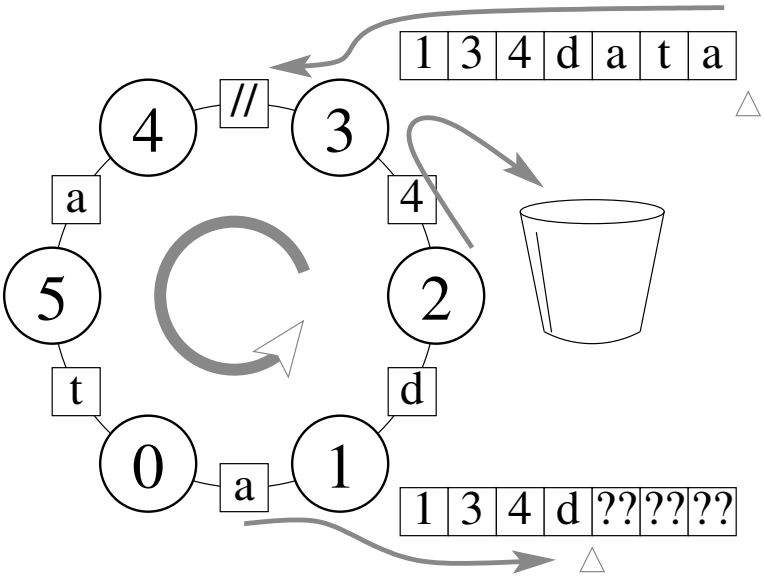

Figure 4: End Send – The same ring structure is shown. The triangle marker is now shown at the end (after the data) of the input buffer. The box between nodes 4 and 3 into which the arrow from the input buffer leads is marked //. The last symbols from the input buffer remain in the boxes of the ring, in counter-clockwise order from the '//' box reading "a", "t", "a", "d" and "4". A new arrow leads out from the box containing the "4" into a picture of an empty bucket. The output arrow from the box between nodes 0 and 1 leads into the output buffer which now contains "1", "3", "4", and "d", after which the remaining positions still contain "??". The triangle marker indicates the position after the "d".

6 Submission Requirements

The assignment should be handed in as a single `tar(1)` file, through CourseLink.

Be sure that:

- *your name* is in a comment at the beginning of the program
- you have included a functional `makefile`
- your submission contains a brief discussion describing your program design in a file entitled `README.md` or `README.txt`

Be absolutely sure that your code compiles and runs on `linux.socs.uoguelph.ca`.

There are several system dependencies that will differ having to do with header files and the usage of some of the low level semaphore primitives that will mean that if you develop your code elsewhere and hand that in without checking it is unlikely to compile and run properly on our grading platform. Whatever you hand in is what we will grade, so please make sure that you are handing in working code.