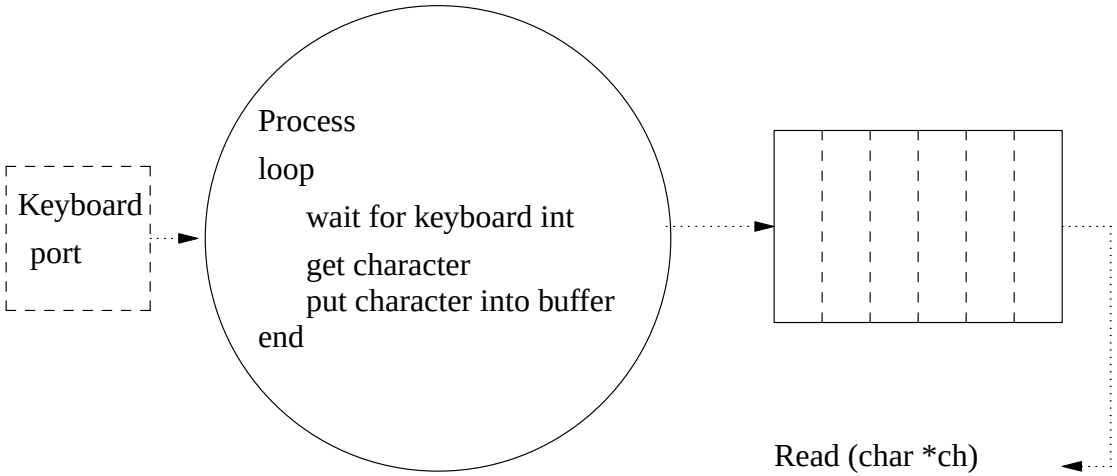
Shared buffer

- laboratory 2 implements a shared buffer



  - shared by the application process `Read (char *ch);`
  - shared by the driver process

- what are the problems with this?
  - data might be altered by both processes at the same time
  - data (buffer pointers) could become corrupt

Mutual exclusion

- require a mechanism to ensure that only one process can manipulate data at any one time
  - *mutual exclusion*

- the concepts we discuss today are *very* important for microkernels and operating systems
  - a fundamental building block

How do we implement mutual exclusion?

- simplest mechanism
  - mask processor interrupts off
  - processor cannot respond to any interrupt and therefore will execute code in sequence until it masks interrupt back on again
  - sometimes these critical sections of code are called *atomic*
  - what are this disadvantages with this approach?
  - what are this advantages with this approach?

How do we implement mutual exclusion?

- another mechanism is *semaphores*
  - essentially a binary *semaphore* is a token which can be grabbed by *only one* process at a time
  - a token is taken at the entry to the critical section and given back at the end of the critical section
  - a process can only enter once it has the token

Semaphores

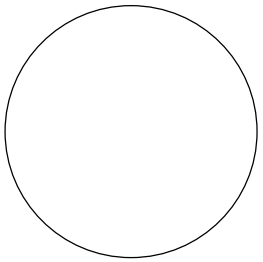
- consider the following two processes:

```
           (* Shared semaphore *)
Semaphore token;  (* initial value 1 *)

void ProcessA ()          void ProcessB ()
{                         {
   while (TRUE) {            while (TRUE) {
      ...                       ...
      Wait(Token)               Wait(Token)
      (* critical *)            (* critical *)

      Signal(Token)             Signal(Token)
      ...                       ...
   }                         }
}                         }
```

Semaphores

SEMAPHORE token 

- `Wait` gets the token
- `Signal` returns the token

Semaphores

- note that `Wait` and `Signal` are both *atomic*
- they are implemented in software with processor interrupts masked off

Semaphores

- we can express `Wait` and `Signal` in pseudo code:

```
void Wait (s)
{
   when s>0
      s--;
}

void Signal (s)
{
   s++;
}
```

Semaphores

- in our previous example the initial value of s would be 1
  - note that this is pseudo code
  - note the use of **when**

Semaphores

- we have now seen how a critical section can be achieved by using semaphore primitives `Wait` and `Signal`
- for example access to the shared buffer will be a critical section

Starting to implement a shared buffer using semaphores

```
void put (char ch)          char get (void)
{                           {

   Wait(Mutex)                 Wait(Mutex)
   (* safe to alter *)         (* safe to alter *)
   (* buffer        *)         (* buffer        *)
   place ch into buf           remove ch from buf

   Signal(Mutex)               Signal(Mutex)


                               return ch;
}                           }

     char buffer[Max];   (* global data *)
     SEMAPHORE Mutex;    (* global data *)
```

Completed implementation of a shared buffer using semaphores

```
void put (char ch)          char get (void)
{                           {
   Wait(SpaceAvailable)        Wait(ItemAvailable)
   Wait(Mutex)                 Wait(Mutex)

   (* safe to alter *)        (* safe to alter *)
   (* buffer        *)        (* buffer        *)
   place ch into buf           remove ch from buf

   Signal(Mutex)               Signal(Mutex)
   Signal(ItemAvailable)       Signal(SpaceAvailable)
                               return ch;
}                           }

     char buffer[Max];      (* global data *)
     SEMAPHORE Mutex;       (* global data *)
```

Shared buffers

- see `lab/boundedbuffer/BufferDevice.c` for a shared buffer data structure

- remember we need to solve the problem of what should happen if:
  - there is no data to take out?
  - there is no room left in the buffer?

Shared buffers

- if there is no data in the buffer and we attempt to get a datum then we should *wait* until data arrives
- if there is no space in the buffer and we attempt to put a datum then we should *wait* until space available
- we can implement this with two semaphores
  - `ItemAvailable`
  - `SpaceAvailable`

Shared buffer (continued)

- before we place an item into a buffer we must
  `Wait(SpaceAvailable)`
- before we extract an item from a buffer we must
  `Wait(ItemAvailable)`
- after we place an item into the buffer we must
  `Signal(ItemAvailable)`
- after we extract an item from the buffer we must
  `Signal(SpaceAvailable)`

Shared buffer (continued)

- what are the initial values for an empty buffer? (size 3)
  - `ItemAvailable  0`
  - `SpaceAvailable  3`

- this buffer mechanism is known as Dijkstra's bounded buffer after its author E.W. Dijkstra who discovered the algorithm in 1960s

Shared buffer (continued)

```
Wait(SpaceAvailable)     Wait(ItemAvailable)

Signal(ItemAvailable)    Signal(SpaceAvailable)
```

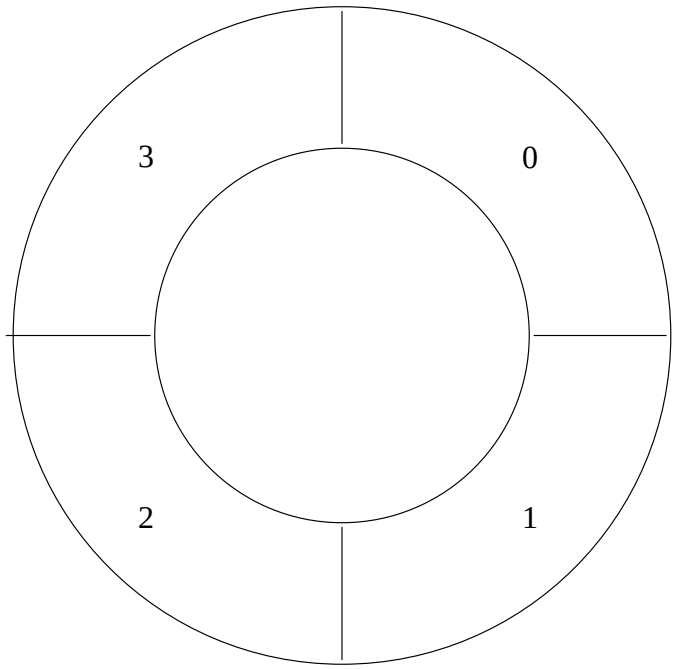
Semaphores

- the module `Executive.c` in the luk documentation exports the:
  - type `SEMAPHORE`
  - functions `Wait` and `Signal`

- you can use these functions to implement your bounded buffer

Circular buffer details

- circular buffer manipulation
  - need to put a datum in at the front
  - need to extract a datum from the end



Circular buffer details

- implement this with two indices
  - `in` and `out`
  - when we add a datum to the buffer we place it at position `in`. We then increment `in` modulo the buffer size.
  - when we extract a datum from the buffer we extract it from position `out`. We then increment `out` modulo the buffer size.

Circular Buffer Code

- when we add a datum to the circular buffer we:

```
Buf[in] = ch;
in = (in+1) % MaxBufferSize;
```

- when we extract a datum from a circular buffer we:

```
ch = Buf[out];
out = (out+1) % MaxBufferSize;
```

Circular Buffer Code (continued)

- in this laboratory tutorial you have write the procedure `InitBuffer`.
  It must perform 3 operations:
  - it must create a buffer
  - initialize the buffer pointers to correct values
  - initialize the semaphore values

- create the buffer.

Creating and initializing a buffer

- the data structure `Buffer` is a pointer type
  - you must make sure it points to something sensible!
  - to do this use the function `Storage_ALLOCATE`

-
```
Storage_ALLOCATE((void **)&b, sizeof(Buffer));
```

- initialize the buffer pointers `in` and `out`

-
```
b->in = 0;
b->out = 0;
```

Circular Buffer Code (continued)

- initialize semaphore values
  - *must use procedure* `InitSemaphore` from `Executive.c` to initialize semaphores!
- example
- ```
  b->Mutex = Executive_InitSemaphore(1, SafeStr("Mutex"));
  ```
- you must initialize the two other semaphores
  - `ItemAvailable`
  - `SpaceAvailable`
  - to their respective values `0` and `MaxBufferSize`
  ```

Device driver (revisited)

- recall that the high level description of the device driver was:

```
void DeviceDriver (void)
{
   (* mask processor *)
   (* ints off       *)
   TurnInterruptsOff ;

   SetupDevice ;
   EnableDeviceInterrupts ;
   while (TRUE) {
      WaitForInterrupt(DeviceInterrupt) ;
      ServiceDevice ;
      Store or retrieve data;
   }
}
```

- the value of `DeviceInterrupt` is 021H for our microkernel

Laboratory: bounded buffer

- once you have completed this exercise you should be able to type characters and see them appear on the screen
- two processes active on your microkernel
  - device driver `ReadDriver`
  - user code (reads `ch` and writes `ch`)
- remember that your device driver is responding to interrupts and successfully placing characters into the shared buffer
- the application process will extract characters from the buffer and display them to the screen

Self check work and extra work

- describe major data structures being used by: `BufferDevice.c`, `Scn.c`
- make a note of each module - what is its primary purpose
- if you have more time, provide a commentary on the boot code