

Semaphores and a shared buffer

- recall our previous example from last week which had two processes
 - one process calls `put` and another process calls `get`
- both operate on a shared buffer
 - we use a semaphore called `Mutex` to protect the buffer

Semaphores and a shared buffer

- ```

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

```

## Semaphores and a shared buffer

- what happens if a process calls `get` before a process calls `put`?
- there is no character to take from the buffer
  - there is no data to return
- what happens if a process keeps calling `put` and no process calls `get`
  - potentially the buffer will be overrun

## Semaphores and a shared buffer

- both cases can be fixed by using two additional semaphores
- if there is no character in the buffer and we call `get` then we should *wait* until data arrives
- if there is no space in the buffer and we attempt to put a character into the buffer then we should *wait* until space becomes available

## Semaphores and a shared buffer

- we can implement this with two semaphores, which we will declare as
  - `itemAvailable`
  - `spaceAvailable`

## Semaphores and a shared buffer

- before we place a character into a buffer we must `wait(spaceAvailable)`
- before we extract a character 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)`

## Semaphores and a shared buffer

- what are their initial values for an empty buffer?
  - for simplicity let us assume the buffer can hold four characters
  - `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

## 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 *)
      
```

Completed implementation of a shared buffer using semaphores

- if one process keeps calling `put` and another process calls `get` we see that both processes are synchronising against taking data from an empty buffer and also from putting data into a full buffer

Readers and writers problem and semaphores

- another common classic problem in operating systems is solving the readers/writers problem
- here the problem is defined as some common resource needs to be protected such that
 - multiple readers can read from the resource simultaneously
 - only one writer can write to the resource at a time
 - a writer must wait for all readers to finish reading before it can alter the resource

Readers and writers problem and semaphores

- how to solve this with the minimal amount of semaphores?
- this problem is common among databases or game servers
- we use a `mutex` semaphore to protect the other data structures used in our lock
- we use another semaphore `writers` to queue multiple writers trying to access the shared resource
- we use an integer count to count the number of readers reading from the resource `readcount`

Readers and writers problem and semaphores

- the writer processes can be implemented by:

```
writers = semaphore (value = 1)

while True:
    ...
    wait(writers)
    # the process can now write to the shared resource
    signal(writers)
    ...
```

Readers and writers problem and semaphores

- the reader process can be implemented by:

Readers and writers problem and semaphores

- ```

mutex = semaphore (value = 1)
readcount = 0

while True:
 ...
 wait(mutex)
 readcount = readcount+1
 if readcount == 1: # first reader waits as a writer
 wait(writers)
 signal(mutex)
 # reader can read the shared resource
 wait(mutex)
 readcount = readcount-1
 if readcount == 0: # last reader signals as a writer
 signal(writers)
 signal(mutex)
 ...

```

## Interprocess communication: Message passing

- message passing is another form of Interprocess Communication
- it allows processes to communicate and to synchronise their actions without sharing the same address space
- a message passing facility provides at least two operations
  - `send(message)` and `receive(message)`
- some message passing libraries allow for variable sized data to be sent/received and other allow a fixed amount of data to be send/received
  - tradoffs between complexity of implementation of the library and complexity of the user program

## Interprocess communication: Message passing

- the message passing libraries also may be further complicated by how a process addresses another process
- consider
  - ```

send(P, message) # send a message to process P
received(Q, message) # receive a message from process Q
          
```
- we describe these primitives as having symmetry in addressing
 - that is both processes need to know the name of the other to receive and send a message

Interprocess communication: Message passing

- other library implementations might use asymmetric naming for process addressing, consider:

- ```
send(P, message) # sends a message to process P
receive(id, message) # receive a message from any process
id will contain the process's name
```

## Conclusion

- we have seen how semaphores can be used to solve some classic computer science problems
  - readers/writers and shared buffer
- we have explored the message passing paradigm