Messages

Messages for Communication & Concurrency Control

IPC

Our Message Protocol

Mutexes & Signaling

Messages

IPC

IPC

- **Interprocess communication** (IPC): A system that allows processes to talk to each other

IPC

- **Interprocess communication** (IPC): A system that allows processes to talk to each other

What are some useful applications of IPC?

IPC

- IPC can denote communication between processes on the same computer
- However, IPC can also be used for **distributed processes**
  - Processes run on different computers
  - Connected through network
  - Sockets or messaging most common

IPC

- **Interprocess communication** (IPC): A system that allows processes to talk to each other
  - Messages/mailboxes (phase 2)
  - Signals
  - Pipes, files
  - Semaphores (phase 3)
  - Shared memory

IPC

- **Mailboxes** allow processes to send **messages** to other processes
  - Atomic messages, small payload
  - Buffering (often)
  - Blocking vs Non-blocking

IPC

- A **signal** is a software interrupt, sent from one process to another
  - OS must deliver, forces receiver into a **signal handler** function
  - Processes can block signals temporarily (analogous to disabling interrupts)
  - Typically no payload
  - “Default” behavior if no handler
    - Ignore, terminate, core dump, etc.

See **`$ man 7 signal`**

IPC

Why would one choose to use a message/mailbox over a signal for IPC?

IPC

- A **pipe** is a file-like mechanism which handles a stream of bytes, but is never stored on disk
  - `stdin,stdout,stderr`
  - `cmd1 | cmd2`
  - Usually only one reader, but many writers OK
  - Block writers when buffer fills

IPC

Example!
`/tmp/search.c`

- A **pipe** is a file-like mechanism which handles a stream of bytes, but is never stored on disk
  - `stdin,stdout,stderr`
  - `cmd1 | cmd2`
  - Usually only one reader, but many writers OK
  - Block writers when buffer fills

IPC

- **Semaphores** are integers used for counting events, block processes when insufficient resources
  - No payload
  - `P()` decrements, blocks if insufficient
  - `V()` increments, releases blocked procs
  - Useful for mutexes, resource control, buffering
  - Slide deck later!

IPC

- **Shared memory** is any time that two processes map the same writable virtual page
  - Probably map to different virtual addresses
  - Instantaneous data flow
  - Need additional mechanism if want to coordinate via blocking
  - Hard to dynamically resize
  - Example: `critical.c` and `critical_lock.c`

Activity: IPC

Two processes A and B need to achieve mutual exclusion in a critical section via a lock (mutex).

Which form of IPC should be used?

Activity: IPC

Two processes A and B need to both send and receive strings between each-other to coordinate a computation.

Which form of IPC should be used?

IPC

- **Non-blocking** functions allow a program to ask if something is available, but refuse to block if not
  - Mostly used for IPC & dist. communication
  - Great for handling multiple simultaneous connections
  - Often, confusing to use
  - Can hurt performance if you don’t have some way to sleep (spinning or polling)

Messages

Phase 2 Messaging Protocol

Our Messaging Protocol

- In our simulation (Phase 2), we will implement mailboxes and messages
- Mailboxes are created with `MboxCreate()`, and identified by integer IDs
- Each mailbox will have two limits:
  - Max bytes per message (can be 0)
  - Max buffered messages (can be 0)

Our Messaging Protocol

- In our simulation (Phase 2), we will implement mailboxes and messages
- Mailboxes are created with `MboxCreate()`, and identified by integer IDs
- Each mailbox will have two limits:
  - Max bytes per message (can be 0)
  - Max buffered messages (can be 0)

Why would we ever want either or both of these to be zero?

Our Messaging Protocol

- `MboxSend()` will "buffer" a message when there is no receiver ready to receive it
  - Each mailbox has a max (can be 0)
  - Sender will block if can't buffer
- Buffered messages must be delivered in order!!
- Blocked senders must be kept in order, too
  - Must buffer in same order they blocked
  - ***Beware priority problems!***

Our Messaging Protocol

- `MboxRecv()` will receive one message
  - Block if nothing buffered

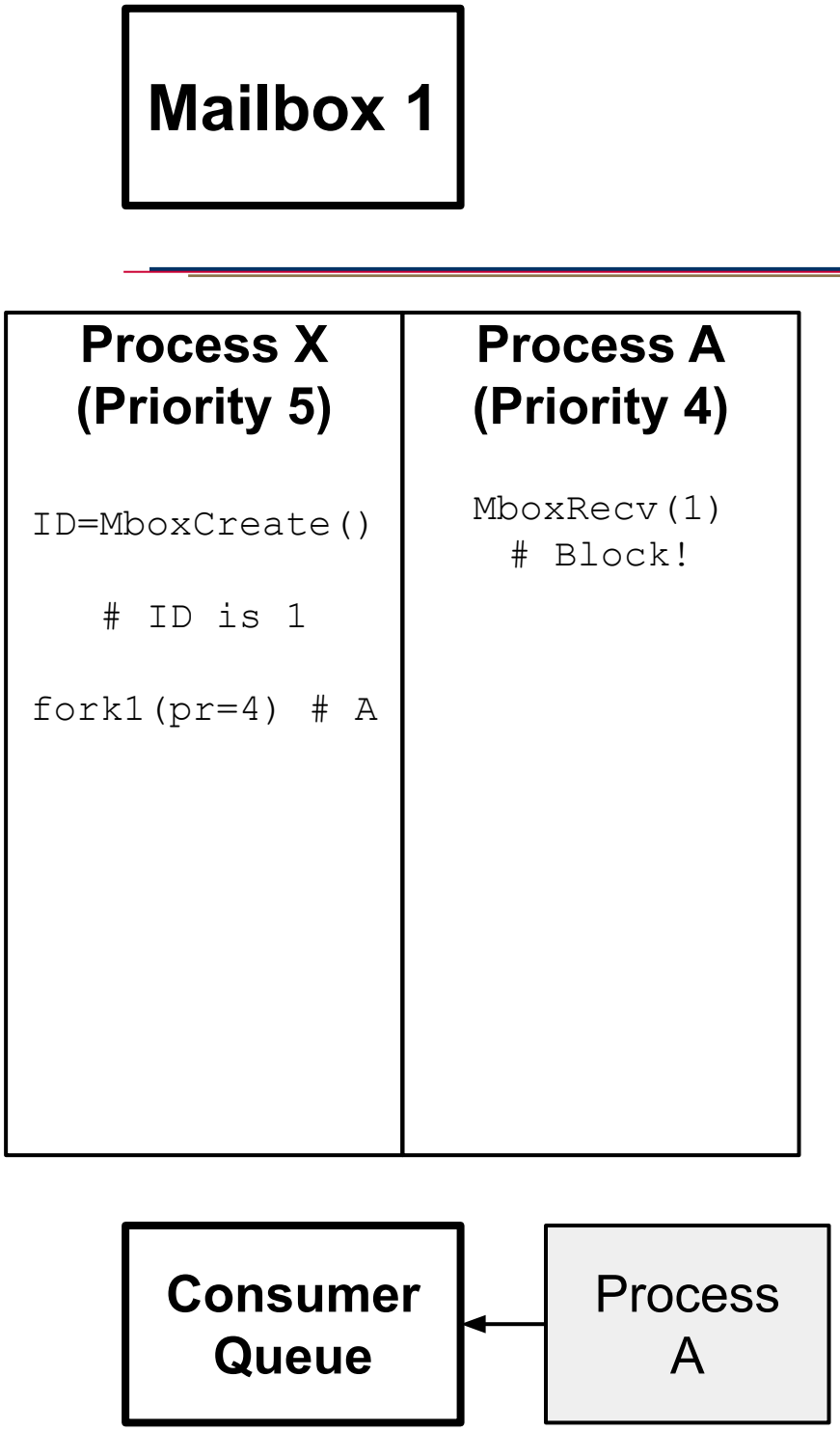
- Messages must be received in order
- Blocked receivers must be kept in order
  - ***Beware priority problems here, too!***

Process X
(Priority 5)

Mailbox 1

Process X
(Priority 5)

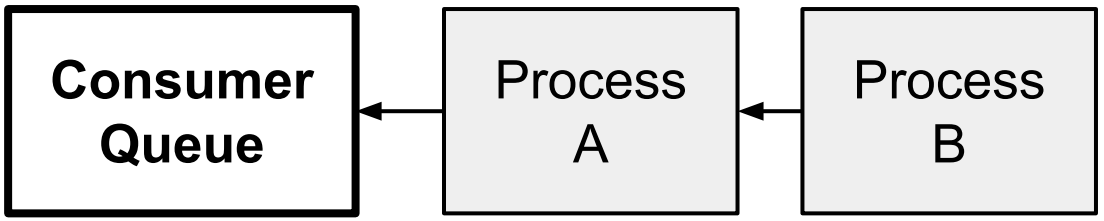
```
ID=MboxCreate()

   # ID is 1
```


Mailbox 1
Process X
(Priority 5)
ID=MboxCreate()
# ID is 1
fork1(pr=4) # A
Process A
(Priority 4)
MboxRecv(1)
# Block!
Consumer
Queue
Process
A

Mailbox 1

Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)

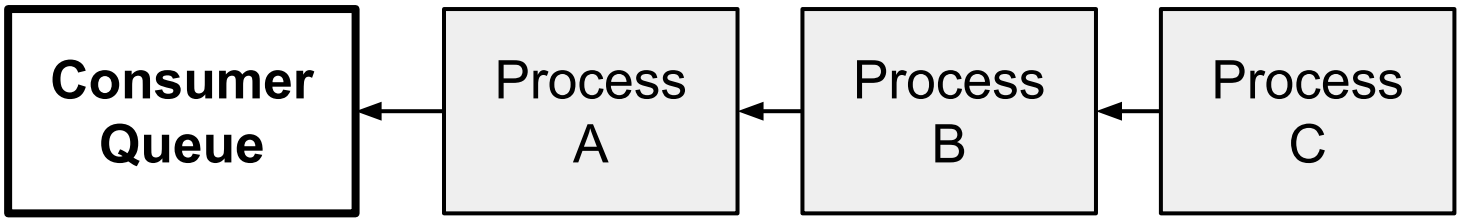
`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`



Mailbox 1

Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)

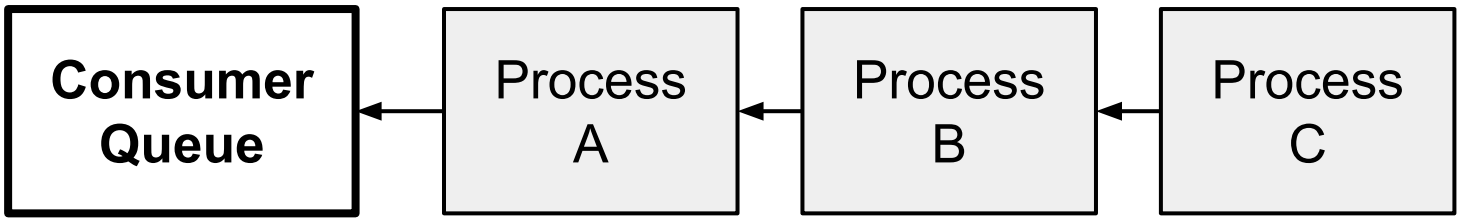
`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`
			`MboxRecv(1)`
			`# Block!`

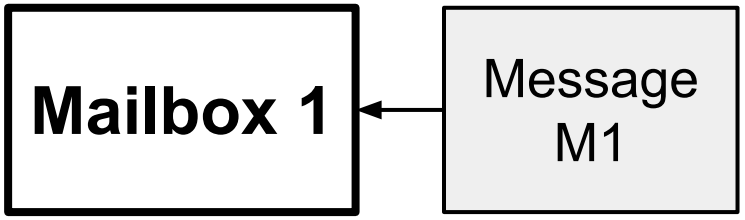


Mailbox 1

Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)	Process D (Priority 1)

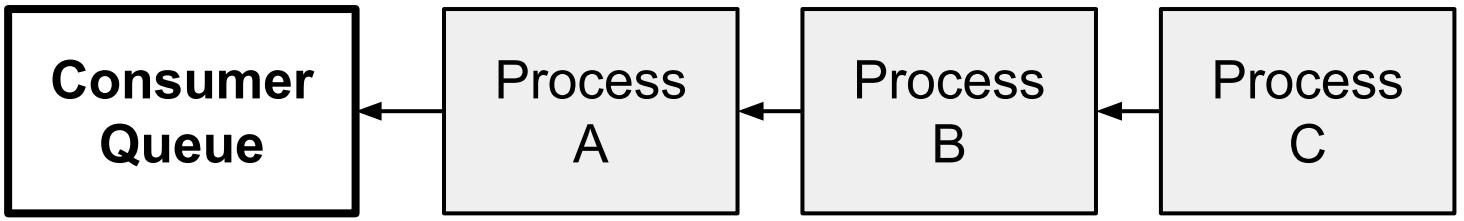
`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`
			`MboxRecv(1)`
			`# Block!`	

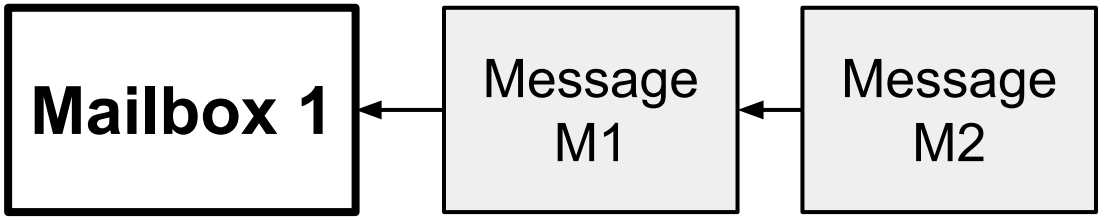




Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)	Process D (Priority 1)

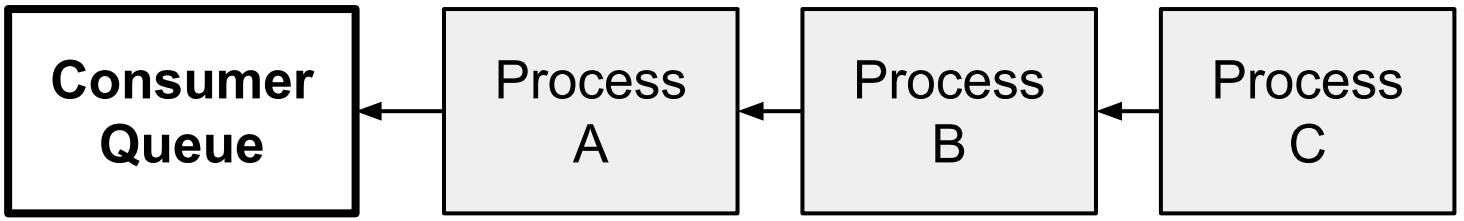
`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`
			`MboxRecv(1)`
			`# Block!`
				`MboxSend(1, M1)`

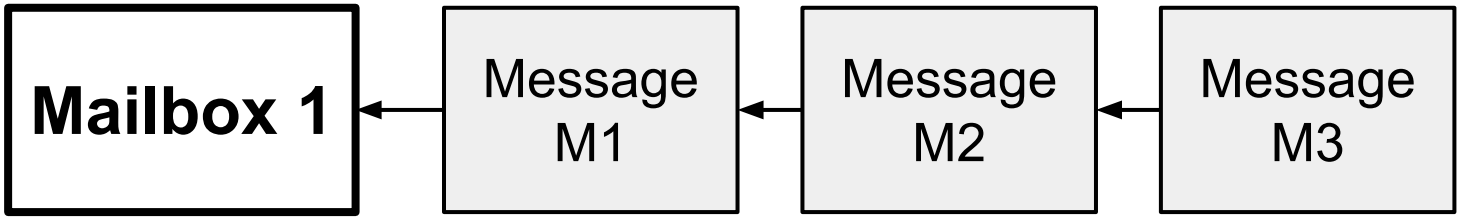




Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)	Process D (Priority 1)

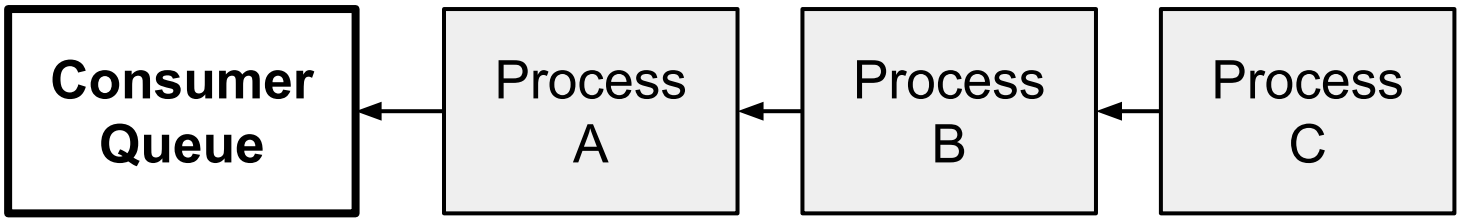
`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`
			`MboxRecv(1)`
			`# Block!`
				`MboxSend(1, M1)`
				`MboxSend(1, M2)`

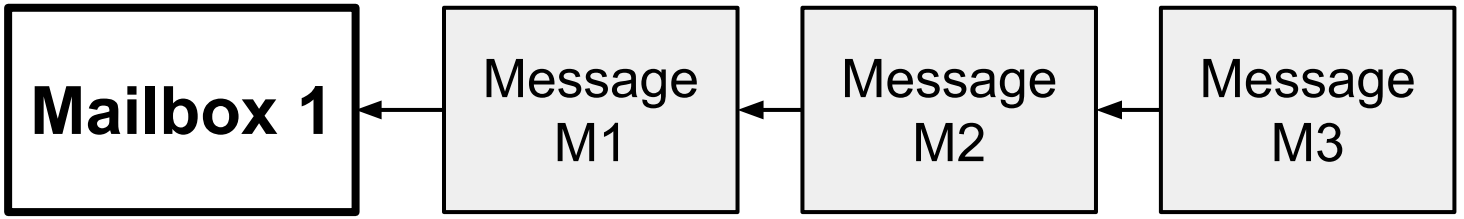




Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)	Process D (Priority 1)

`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`
			`MboxRecv(1)`
			`# Block!`
				`MboxSend(1, M1)`
				`MboxSend(1, M2)`
				`MboxSend(1, M3)`





Process X (Priority 5)

```
ID=MboxCreate()

   # ID is 1

fork1(pr=4) # A
fork1(pr=3) # B
fork1(pr=2) # C
      ....
fork1(pr=1) # D
      ....
```

Process A (Priority 4)

```
MboxRecv(1)
  # Block!
```

Process B (Priority 3)

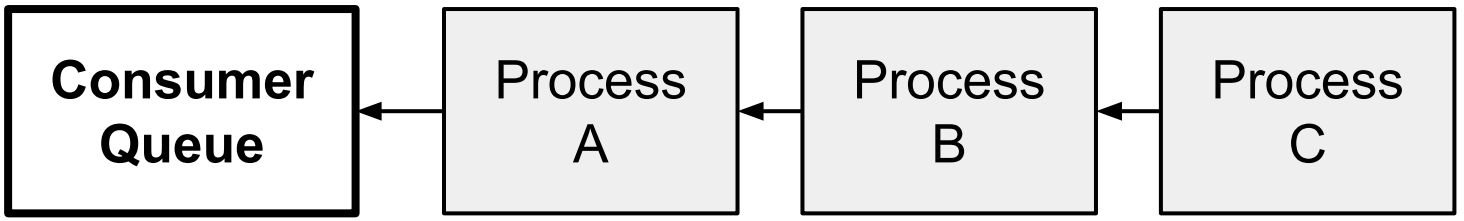
```
MboxRecv(1)
  # Block!
```

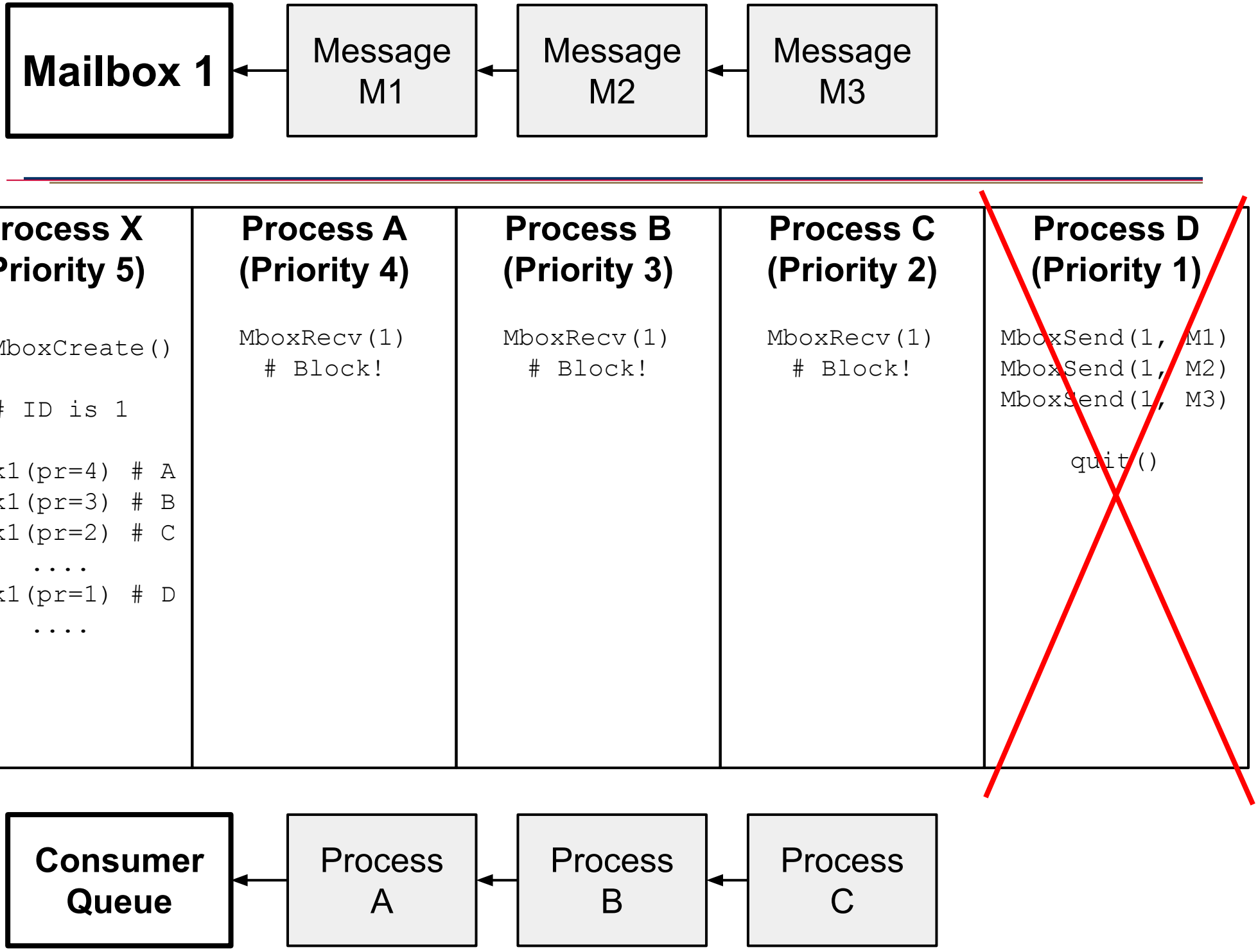
Process C (Priority 2)

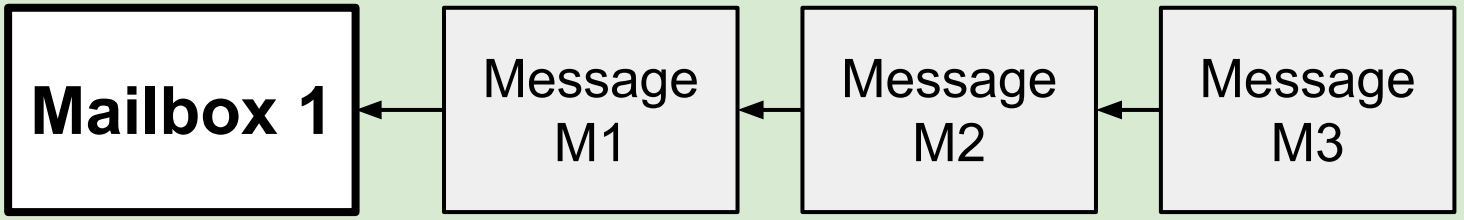
```
MboxRecv(1)
  # Block!
```

Process D (Priority 1)

```
MboxSend(1, M1)
MboxSend(1, M2)
MboxSend(1, M3)

     quit()
```




Mailbox 1
Message M1
Message M2
Message M3
Process X (Priority 5)
ID=MboxCreate()
# ID is 1
fork1(pr=4) # A
fork1(pr=3) # B
fork1(pr=2) # C
....
fork1(pr=1) # D
....
Process A (Priority 4)
MboxRecv(1)
# Block!
Process B (Priority 3)
MboxRecv(1)
# Block!
Process C (Priority 2)
MboxRecv(1)
# Block!
Process D (Priority 1)
MboxSend(1, M1)
MboxSend(1, M2)
MboxSend(1, M3)
quit()
Consumer Queue
Process A
Process B
Process C

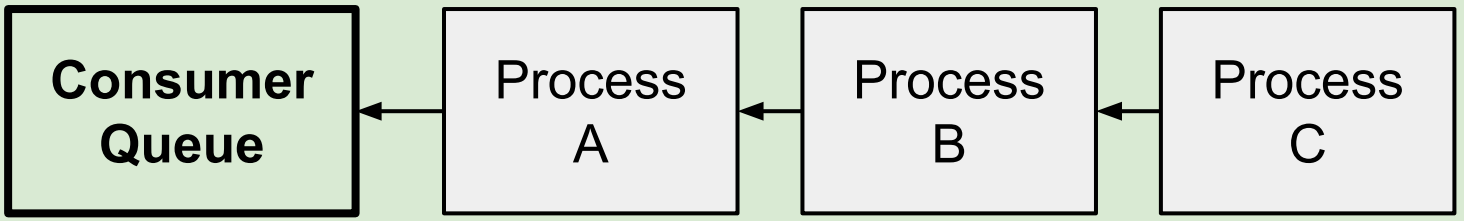


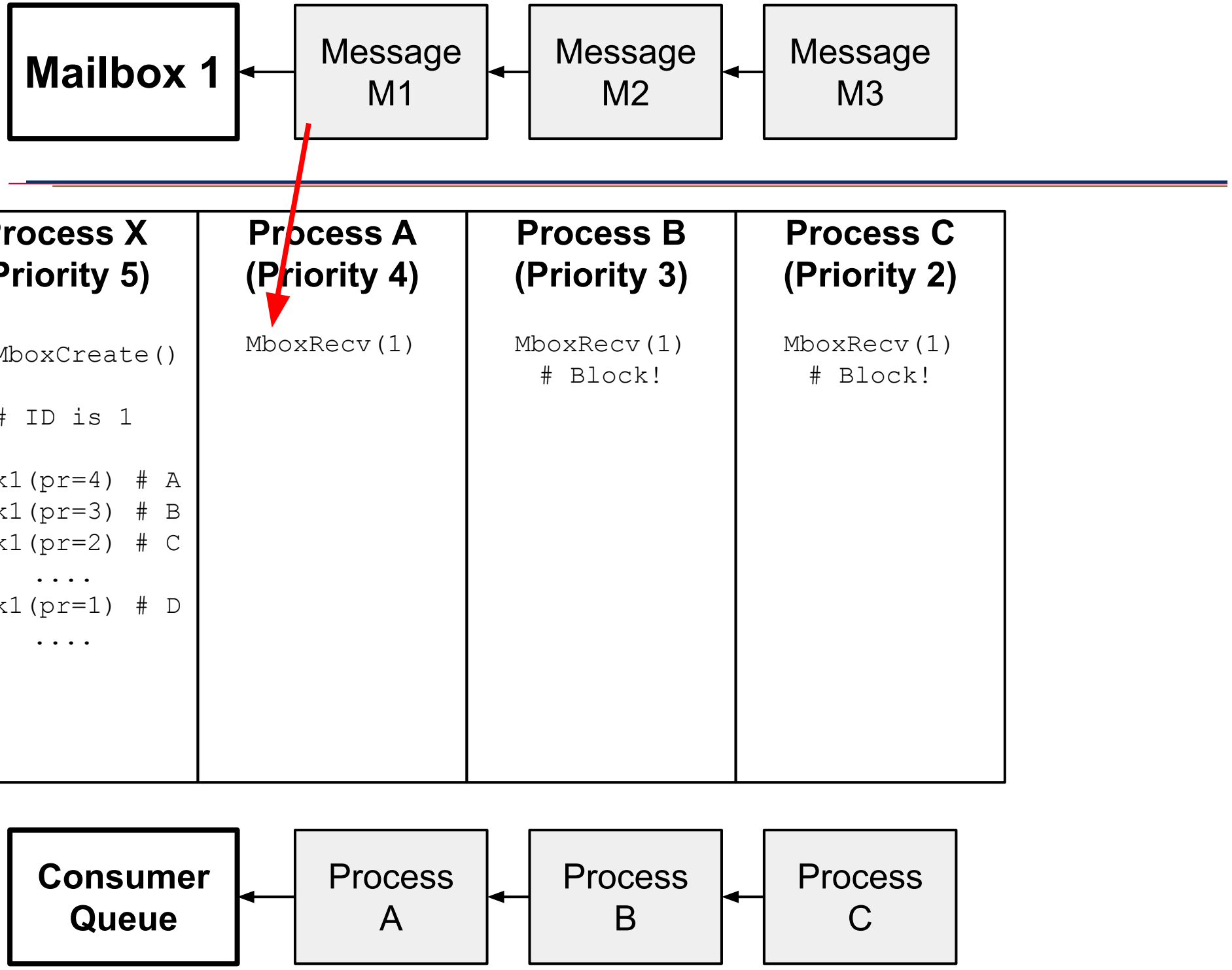
Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)

`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`
	`# Block!`
		`MboxRecv(1)`
		`# Block!`
			`MboxRecv(1)`
			`# Block!`

Who should be woken up next?

Who should receive M1?




Mailbox 1
Message M1
Message M2
Message M3
Process X (Priority 5)
ID=MboxCreate()
# ID is 1
fork1(pr=4) # A
fork1(pr=3) # B
fork1(pr=2) # C
....
fork1(pr=1) # D
....
Process A (Priority 4)
MboxRecv(1)
Process B (Priority 3)
MboxRecv(1)
# Block!
Process C (Priority 2)
MboxRecv(1)
# Block!
Consumer Queue
Process A
Process B
Process C

Process X (Priority 5)

```
ID=MboxCreate()

   # ID is 1

fork1(pr=4) # A
fork1(pr=3) # B
fork1(pr=2) # C
      ....
fork1(pr=1) # D
      ....
```

Process A (Priority 4)

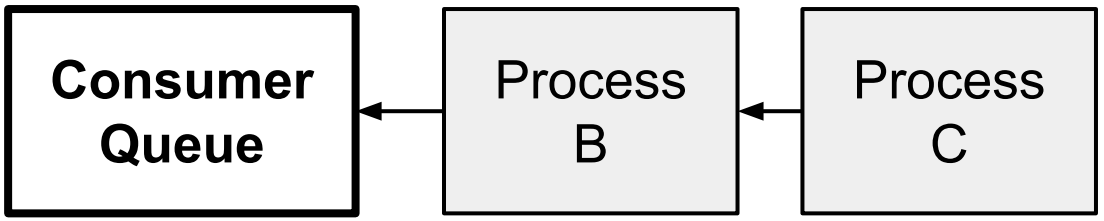
```
MboxRecv(1)
```

Process B (Priority 3)

```
MboxRecv(1)
  # Block!
```

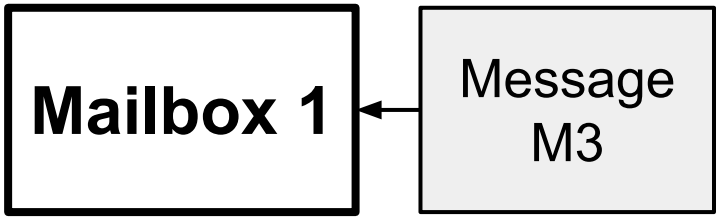
Process C (Priority 2)

```
MboxRecv(1)
  # Block!
```



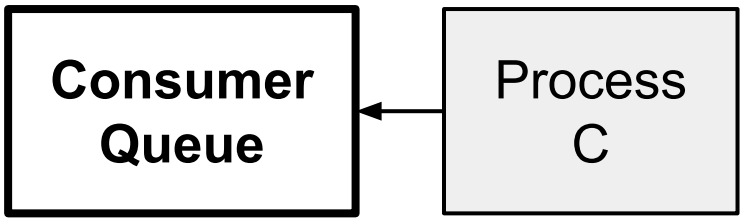
Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)

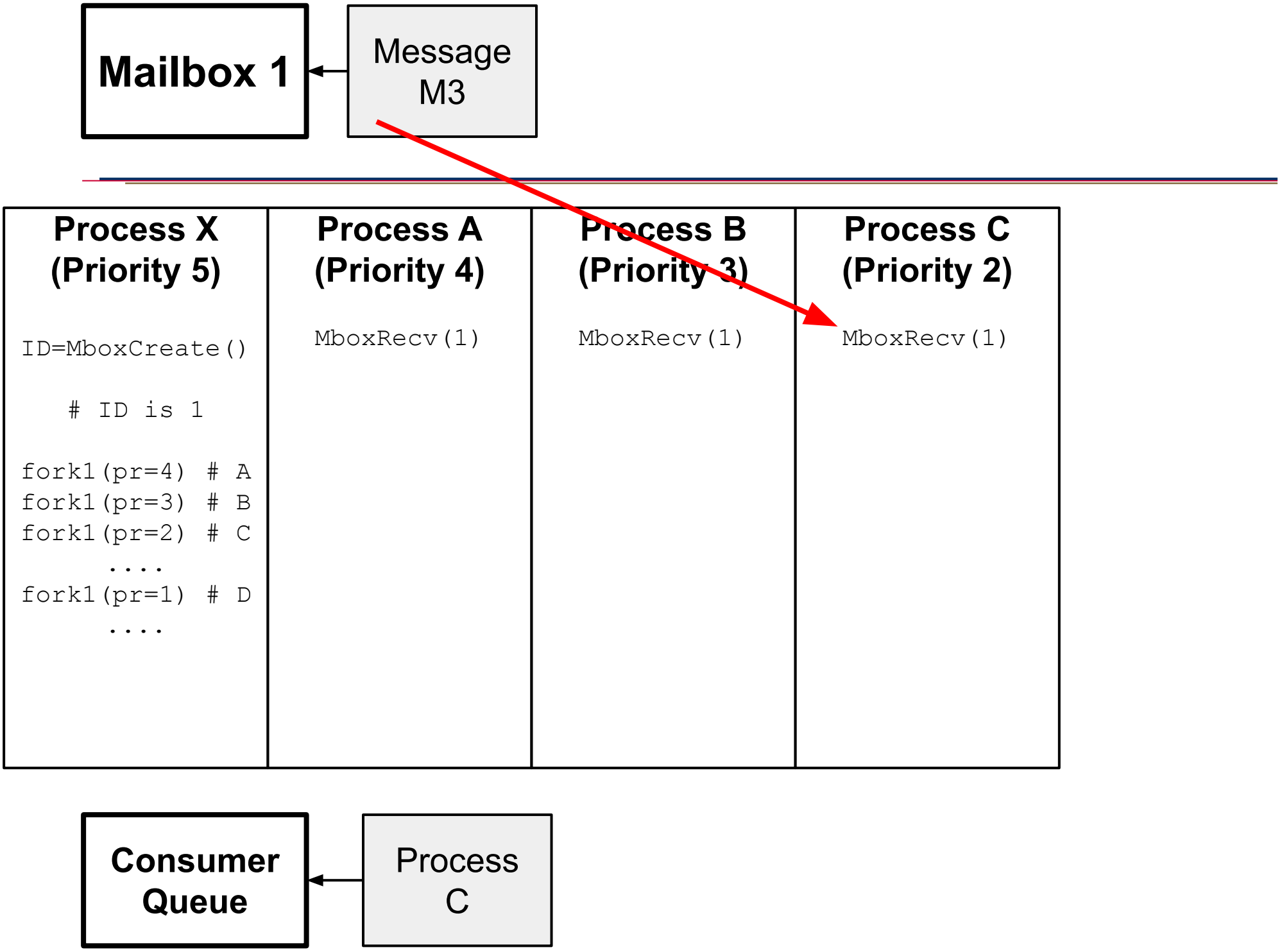
`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`	`MboxRecv(1)`	`MboxRecv(1)`
			`# Block!`

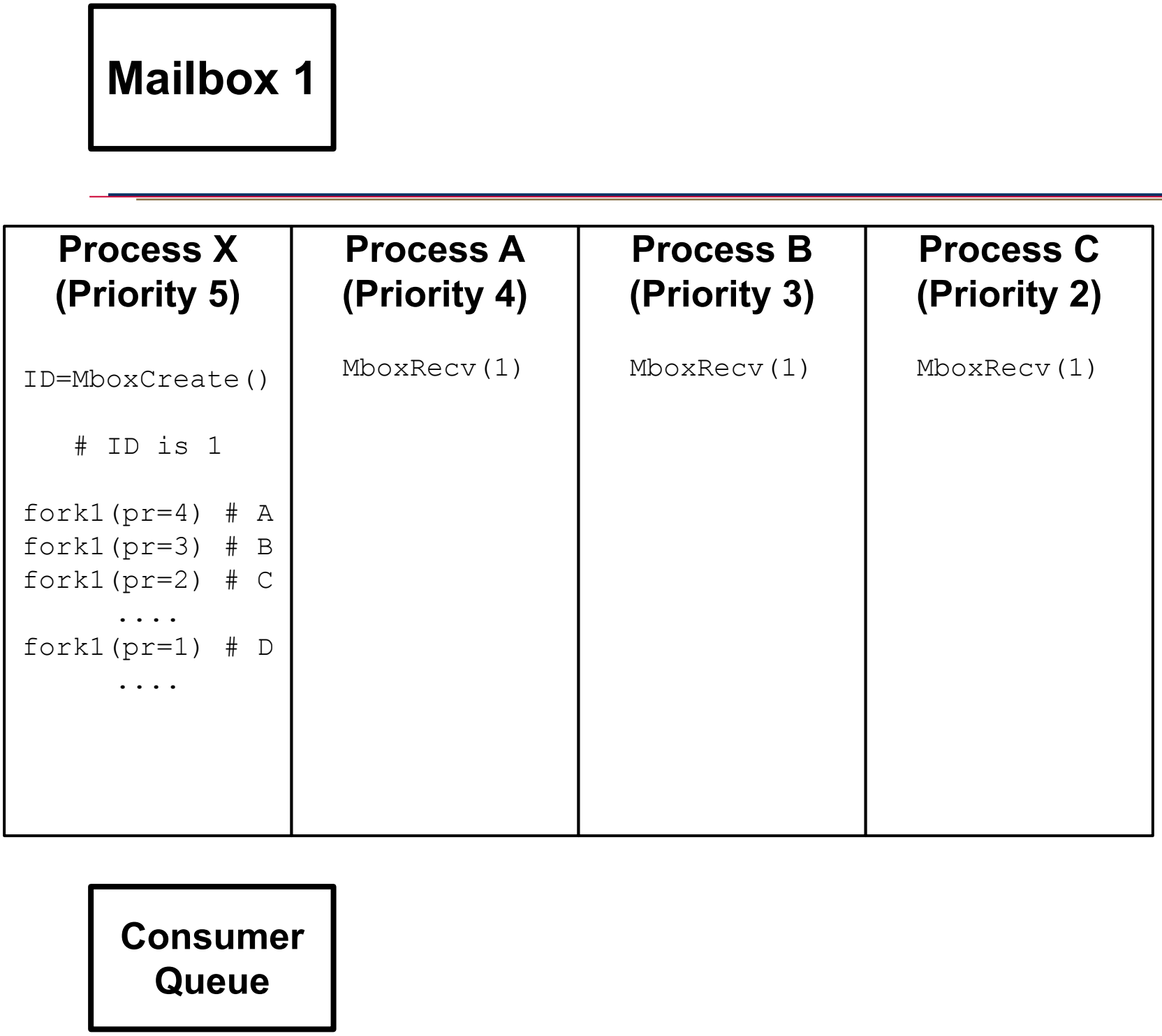


Process X (Priority 5)	Process A (Priority 4)	Process B (Priority 3)	Process C (Priority 2)

`ID=MboxCreate()`
 	
`# ID is 1`
 	
`fork1(pr=4) # A`
`fork1(pr=3) # B`
`fork1(pr=2) # C`
`....`
`fork1(pr=1) # D`
`....`
	`MboxRecv(1)`	`MboxRecv(1)`	`MboxRecv(1)`
			`# Block!`




Mailbox 1
Message M3
Process X (Priority 5)
ID=MboxCreate()
# ID is 1
fork1(pr=4) # A
fork1(pr=3) # B
fork1(pr=2) # C
....
fork1(pr=1) # D
....
Process A (Priority 4)
MboxRecv(1)
Process B (Priority 3)
MboxRecv(1)
Process C (Priority 2)
MboxRecv(1)
Consumer Queue
Process C


Mailbox 1
Process X (Priority 5)
ID=MboxCreate()
# ID is 1
fork1(pr=4) # A
fork1(pr=3) # B
fork1(pr=2) # C
....
fork1(pr=1) # D
....
Process A (Priority 4)
MboxRecv(1)
Process B (Priority 3)
MboxRecv(1)
Process C (Priority 2)
MboxRecv(1)
Consumer Queue

Our Messaging Protocol

- We have a large pool of “message buffers”
  - Global pool, shared by all mailboxes
  - Allocate when you have a message
    - In `MboxSend()`, not `MboxCreate()`
    - Over-allocation is allowed
    - Buffering can fail if no free slots
  - Free in `MboxRecv()`

Our Messaging Protocol

- `MboxCondSend(), MboxCondRecv()` are **non-blocking** versions
  - Send or recv if possible
  - Return special value if you would have blocked

Messages

Mutexes & Signaling

Mutexes & Signaling

- A mailbox can be used (abused?) as a mutex
  - Initialize with size 1
  - `Send()` to lock, `Recv()` to unlock
  - Process A calls `Send()` while in CS
    Process B calls `Send()` and blocks until A calls `Recv()`

ICA: Mutexes & Signaling

- A mailbox can be used (abused?) as a mutex
  - Initialize with size 1
  - `Send()` to lock, `Recv()` to unlock
  - Process A calls `Send()` while in CS
    Process B calls `Send()` and blocks until A calls `Recv()`

Can it be reversed? Use `Recv()` to gain lock and `Send()` to release it? Discuss!

Mutexes & Signaling

- Soon, we'll be implementing syscalls
  - For example, call to `fopen` should block after the call until the the device responds
  - In-General: Block current process until something happens

ICA: Mutexes & Signaling

- Soon, we'll be implementing syscalls
  - For example, call to `fopen` should block after the call until the the device responds
  - In-General: Block current process until something happens

How to use mailboxes for sleep & wakeup?

Nicer interface than calling Phase 1 code
How to do it?
When to allocate mailboxes?
What can be shared, or not?