CS452/652 Real-Time Programming Course Notes

Daniel M. Berry, Cheriton School of Computer Science
University of Waterloo

Inter-Process Communication (IPC)

- data transfer from task to task, i.e., communication
- inter-task control flow, i.e., synchronization

IPC Mechanisms

- semaphores, i.e., p (request) and v (release)
- monitors, i.e., a class with a semaphore ensuring that only one process at a time is inside it.
- shared memory
- sockets
- remote procedure call (RPC)
- broadcast, multicast
- asynchronous and synchronous message passing

Your Kernel

Your kernel will use synchronous, i.e., blocking, send–receive–reply message passing with variable-length messages, and will thus have primitives:

- Send
- Receive
- Reply

Send

send: (tid × msg) → replyMsg

send the message to the specified task;
wait for a reply message

int Send(int tid, char *msg, int msgLen,
char *replyBuf, int replyBufLen)

if returned value ≥ 0, then it is the actual reply length
if returned value < 0, then it is indicating an error
the replyBufLen is the maximum reply length

Receive

receive: () → (tid × msg)

wait for a task to send a message

int Receive(int *tid, char *msgBuf, int msgBufLen)

if returned value ≥ 0, then it is the actual message length
if returned value < 0, then it is indicating an error
the *tid is the identify of the sending task
the msgBufLen is the maximum message length

Reply

reply: (tid × replyMsg) → ()

reply to a previously received message

int Reply(int tid, char *replyMsg, int replyMsgLen)

if returned value ≥ 0, then it is the actual reply length
if returned value < 0, then it is indicating an error
the replyMsgLen is the maximum reply length

Blocking

Send *will* block.

Receive *may* block.

Reply *never* blocks.

Synchronous vs. Asynchronous Communication

We discuss the advantages of each.

Advantages of Asynchronous Communication

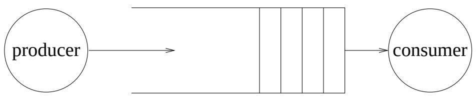
- Sender can do other work while waiting for the sent message to be received and replied to. However, is this really an advantage?
- We can set up cyclic message-passing patterns that would otherwise deadlock, e.g., a task sending a message to itself.

Advantages of Synchronous Communication

- It provides built-in synchronization that can be used to achieve any synchronization
- It is much easier to reason about synchronous communication.
- Synchronous communication can mimic asynchronous communication with the use of helper processes, that do the other work while the sending process is waiting for the sent message to be received and replied to.

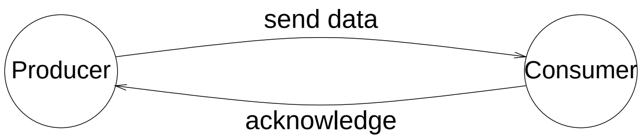
Example

Example of Blocking Send–Receive–Reply Bounded Buffer



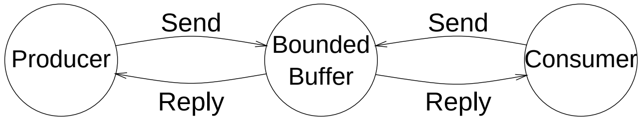
We look at a high-level view first and then the details.

High-Level Data Flow



Producer sends data and the consumer acknowledges receipt of data.

Process Structure Diagram



Implementation of send data

High-level send data is implemented by

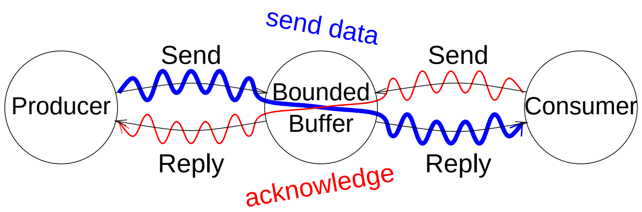
(1) a Send with data from producer to buffer task and then

(2) a Reply with the same data from buffer task to consumer, where (2) is done in response to (1’)

(1’) a Send with null data from consumer to buffer task, indicating that the consumer is ready to receive data, and then

(2’) a Reply with the same null data from buffer task to producer.

Superposition



Producer

```
Producer(){
    while(1){
        data = produceItem();
        Send(bufferTask,data,NULL); /* send data =
            Send data; receive NULL */
    }
}
```


Consumer

```
Consumer(){
    while(1){
        Send(bufferTask,NULL,data); /* acknowledge =
            Send NULL; receive data */
        consumeItem(data);
    }
}
```


BoundedBuffer

```
BoundedBuffer(){
    while (1){
        (tid,data) = Receive();
        if (tid==consumer){
            Producing
        } else if (tid==producer){
            Consuming
        }
    }
}
```


BoundedBuffer

```
BoundedBuffer(){
    while (1){
        (tid,data) = Receive();
        if (tid==consumer){
            Producing
        } else if (tid==producer){
            Consuming
        } /* ignore other processes */
    }
}
```

Producing

```
if (queueEmpty()){
    consumerWaiting = TRUE;
} else {
    data = queueRemove();
    Reply(consumer,data);
    if (producerWaiting){
        Reply(producer,NULL);
        producerWaiting = FALSE;
    }
}
```


Producing

```
if (queueEmpty()){ /*here if cons sent msg to Q that it wants data */
    consumerWaiting = TRUE;
} else { /* Q is not empty */
    data = queueRemove();
    Reply(consumer,data);
    if (producerWaiting){ /* if Q was full */
        Reply(producer,NULL); /* tell prod that Q is no longer full */
        producerWaiting = FALSE;
    }
}
```


Consuming

```
if (consumerWaiting){
    Reply(consumer,data);
    Reply(producer,NULL);
    consumerWaiting = FALSE;
} else {
    queueAdd(data);
    if (queueFull(){
        producerWaiting = TRUE;
    } else {
        Reply(producer,NULL);
    }
}
```


Consuming

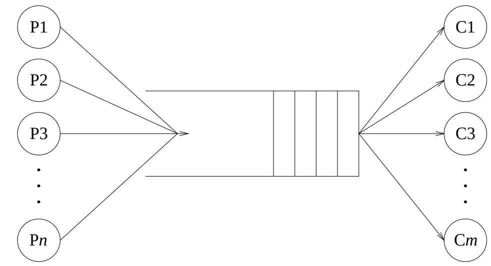
```
if (consumerWaiting){ /*here if prod sent msg to Q that it sent data */
    Reply(consumer,data); /*tell cons that it need not wait any more */
    Reply(producer,NULL); /*tell prod that Q is not full */
    consumerWaiting = FALSE;
} else { /* data arrived before request */
    queueAdd(data);
    if (queueFull(){
        producerWaiting = TRUE;
    } else {
        Reply(producer,NULL); /* tell prod that Q is not full */
    }
}
```

Size of BoundedBuffer

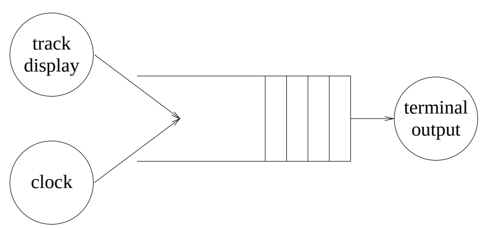
By virtue of the code, what is the maximum number of data items that can be stored in the BoundedBuffer at any time?

Generalization

n producers and *m* consumers



Example



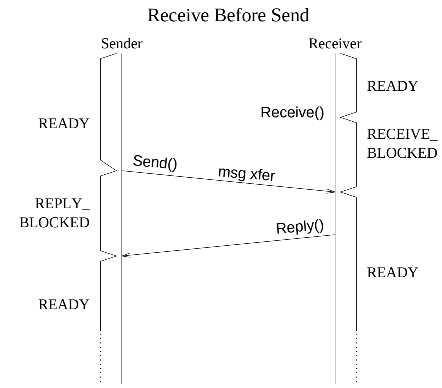
SRR Implementation

- Task States
- Transfer of Control
- State Transitions
- Implementation Suggestion
- Implementation Primitives for Kernel 2

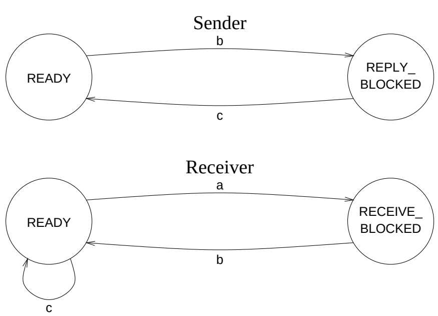
Task States

- Sender
  - SEND_BLOCKED — send to task; msg not received
  - REPLY_BLOCKED — send to task; msg received; no reply yet
- Receiver
  - RECEIVE_BLOCKED — receive issued; no send available

Transfer of Control

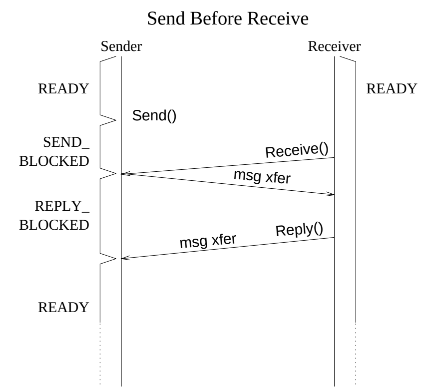


State Transitions

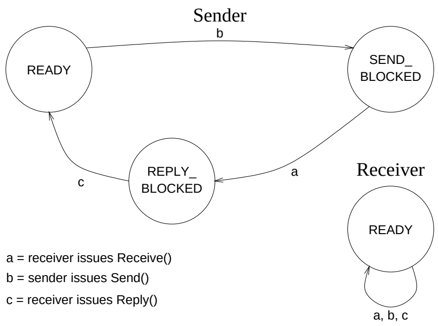


a = receiver issues Receive()
b = sender issues Send()
c = receiver issues Reply()

Transfer of Control

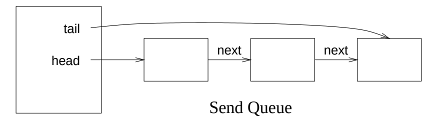


State Transitions



Implementation Suggestion

Each task descriptor should have its own queue of SEND_BLOCKED tasks.



Suggestion, Cont'd

The send queue of task *t* is a queue of tasks each of whom has issued a Send() to *t* but the message has not been transmitted because *t* has not issued a corresponding Receive().

No queue is needed for RECEIVE_BLOCKED or REPLY_BLOCKED tasks.

Why?

Message Transfer

Message transfer

- is directly from one task's address space to another's and
- is not buffered in the kernel.

Who should carry out the actual transfer?

Who Does the Transfer?

Should the kernel do it?

The Kernel?

The cost is linear in the size of the message.

So the kernel should *not* do it.

So it must be some other party, a third party.

Third Party?

The third party is either the sender or the receiver.

Which one?

How does the third party, whatever it is, get the information needed to do the transfer?

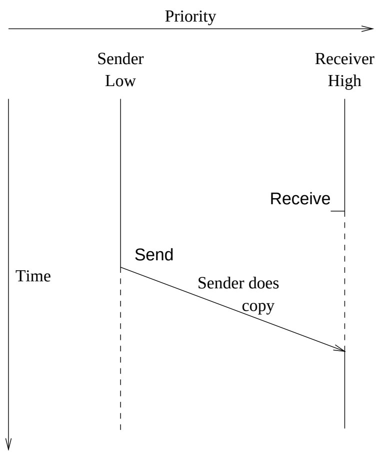
What about the priorities of the sender, the receiver, and of other tasks, if any?

Sender?

If the sender copies during the call of Send:

Suppose the receiver has a high priority and the sender has a low priority and we have a Receive before the corresponding Send.

As It Should Be

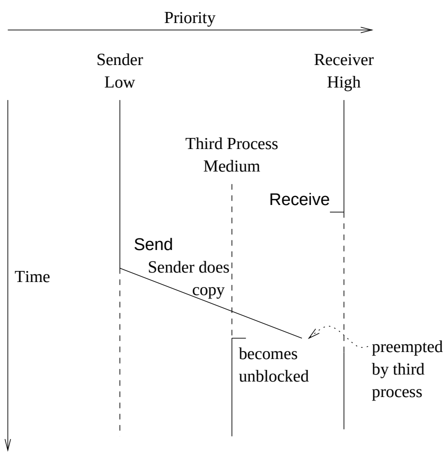


Sender?

However, there might be at least one other task, called the third task.

Suppose this third task has a middle level priority:

As It Can Be



Anomaly

The third process has higher priority than the sender and preempts the sender.

∴, the receiver, though of higher priority than the third task, never gets unblocked and never gets to run. So it is effectively pre-empted even though it is not running.

If the receiver were doing the transfer, the receiver would not be preempted.

So the receiver should do the transfer of a Send.

What If?

Suppose in the above situation, the sender is of high priority and the receiver is of low priority.

Is it correct that the receiver is doing the transfer of a Send?

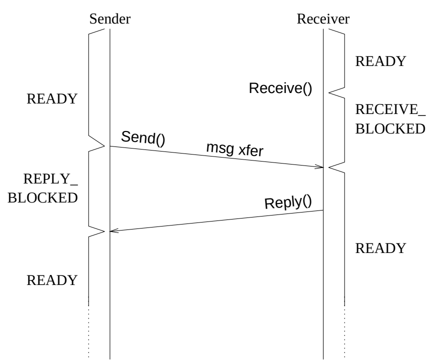
Transfer During Reply

If the sender (of Send) does the transfer during a Reply:

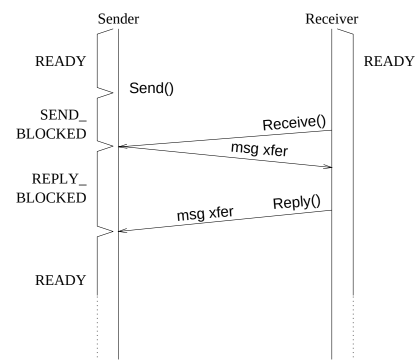
- The receiver (of Send) is not blocked.
- ∴, receiver could overwrite previously written data before the sender is finished transfering it.

If the receiver transfers on Reply, then the sender is blocked, so that no problems can arise from the sender's acting before the transfer is complete.

Receive Before Send



Send Before Receive



Information Primitives for Kernel 2

- MyParentPid()
- MyPriority()
- ValidPid(int pid)

as specified in the kernel specification.

Optional primitive

Destroy(int pid)

which kills the task named pid and reclaims the resources associated with the killed task.

Destroy Semantics

- Tasks that are SEND_BLOCKED or REPLY_BLOCKED on the task named by pid must be made READY; and an error code is returned
- The task named by pid must be removed from the ready queue.
- The storage associated with the task named by pid must be reclaimed.
- The pid itself must not be reused.
- The task descriptor named by the pid must be reusable.

Destroy Semantics, Cont'd

and last, but not least:

- You must figure out a reasonable way to deal with the MyParentPid()s of the children of the task named by the pid.

OS Services

We talk about OS services outside the kernel, and these include device drivers.

Why do they include device drivers?

Name Server

The name server provides the mapping from names to PIDs, and is used to build a name lookup service.

Required Primitives:

- RegisterAs(char *name) — register the PID of the executing process as having the given name.
- tid WhoIs(char *name) — get the PID of the process having the given name.

Optional suggested primitive: tid WaitFor(char *name) — blocking version of WhoIs.

Name Server, Cont'd

A task may register itself with more than one name.

No two tasks can register themselves with the same name.

Finding the Name Server

Either:

- provide
  whoIsNameServer()
  and
  registerAsNameServer()
  or
- Let the initial task be the name server.

Ergo, in any case the name server is a task.

NameServer

```
NameServer(){
    RegisterAsNameServer;
    while (1){
        (tid,msg) ← Receive();
        if (msg.type == WHOIS) {
            who = lookup (msg.name);
            Reply (tid,who);
        } else if (msg.type == REGISTERAS) {
            register (tid,msg.name);
            Reply (tid,NULL);
}   }   }
```


WhoIs

```
int whoIs(char *name){
    — send a msg to the NameServer
}
```

Code of RegisterAs is left as an exercise for the student ☺