Inter-Process Communication

Objectives

- Motivate the need for Inter-Process Communication
- Introduce a simple send/receive/reply message passing paradigm
- Show how to implement this paradigm

Current state of affairs

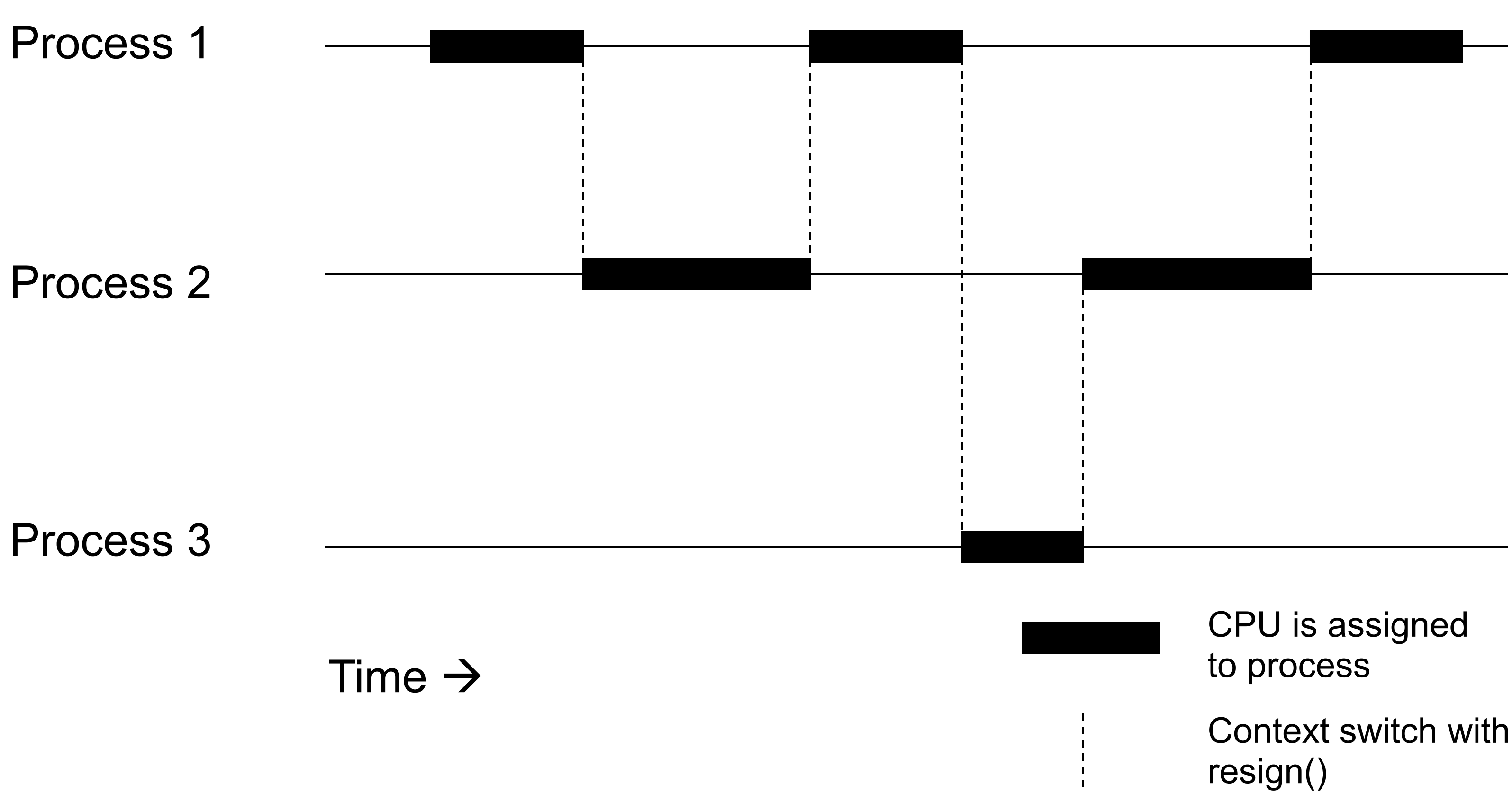
Status quo:

– We can create arbitrary number of processes (up to a maximum of 20)

– TOS is non-preemptive, i.e., context switch only happens explicitly by calling `resign()`

– Processes are independent of each other, i.e., no synchronization between processes.

Context Switch in TOS



Cooperating Processes

- Processes are not isolated but work together. E.g.
  - Process for managing the file system
  - Process for managing the keyboard
  - Process implementing the application logic
- Possible scenario:
  - user shell (e.g. bash) sends a message to the keyboard process
  - user shell “waits” until user has typed a command
  - user shell interprets command and sends appropriate instructions to the file system
- What does “wait” mean? Answer: process is taken off the ready queue because it has nothing to do

Inter Process Communication (IPC)

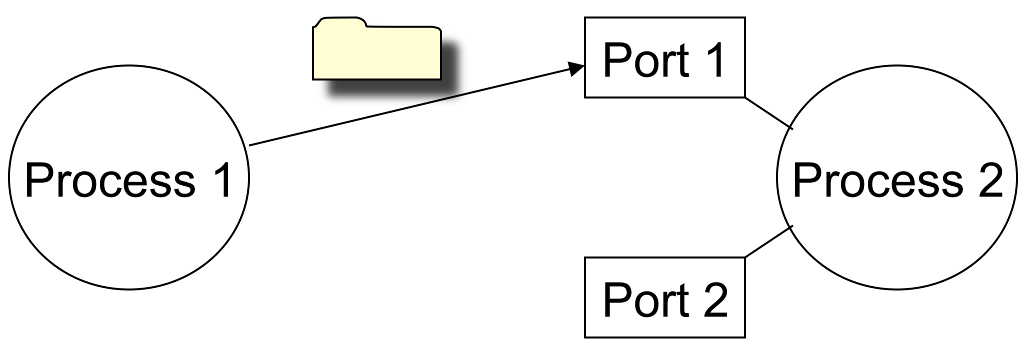
- What is missing?
  - Synchronization mechanisms to coordinate interactions between processes
  - Ability to react to hardware interrupts
- Solution:
  - A communication mechanism between processes, also called *Inter-Process Communication*.

IPC in TOS

- TOS implements IPC through a set of message passing API.
- One process can send a message to another process.
- A message is simply a void-pointer `(void *)`. Remember that all TOS processes share the same address space. Sender and receiver have to agree what the void-pointer is actually pointing to.
- Apart from sending the message, the sender is blocked until the message has been delivered to the receiver.
- This is called the *rendezvous point*, because it is the point in time where sender and receiver meet.

Ports

- Messages are sent to ports; not processes.
- A port resembles a mailbox where messages are delivered.
- A port is owned by exactly one process.
- A process can own several ports.
- A port is defined through type `PORT_DEF` in `~/tos/include/kernel.h`



Port Data Structure

- TOS maintains an array of `MAX_PORTS` ports (defined in `kernel.h`)
- `magic`: magic cookie initialized to `MAGIC_PORT`
- `used`: if this port is available
- `open`: if this port is open
- `owner`: pointer to the process that owns this port
- `next`: all ports owned by the same process are in a single linked list

```
typedef struct _PORT_DEF {
  unsigned  magic;
  unsigned  used;
  unsigned  open;
  PROCESS   owner;
  PROCESS   blocked_list_head;
  PROCESS   blocked_list_tail;
  struct _PORT_DEF *next;
} PORT_DEF;

typedef PORT_DEF* PORT;
```

IPC in TOS

- When sending a message, we may want a process to wait (or *block*)
- Two ways to send a message in TOS:
  - `message()`: sender is blocked until the receiver gets the message
  - `send()`: sender is blocked until the receiver gets the message **and** calls `reply()`

Port functions in TOS

- Port functions are implemented in file `~/tos/kernel/ipc.c`
- `typedef PORT_DEF *PORT;`
- Functions:
  - `PORT create_port()`
    Creates a new port. The owner of the new port will be the calling process `(active_proc)`. The return value of `create_port()` is the newly created port. The port is initially open.
  - `PORT create_new_port (PROCESS proc)`
    Creates a new port. The owner of the new port will be the process identified by `proc`. The return value of `create_port()` is the newly created port. The port is initially open.
  - `void open_port (PORT port)`
    Opens a port. Only messages sent to an open port are delivered to the receiver.
  - `void close_port (PORT port)`
    Closes a port. Messages can still be sent to a closed port, but they are not delivered to the receiver. If a port is closed, all incoming messages are queued.

IPC functions in TOS

- IPC functions are implemented in file `~/tos/kernel/ipc.c`
- Functions:
  - `void send (PORT dest_port, void* data)`
    Sends a synchronous message to the port `dest_port`. The receiver will be passed the void-pointer `data`. The sender is blocked until the receiver replies to the sender.
  - `void message (PORT dest_port, void* data)`
    Sends a synchronous message to the port `dest_port`. The receiver will be passed the void-pointer `data`. The sender is unblocked after the receiver has received the message.
  - `void* receive (PROCESS* sender)`
    Receives a message. If no message is pending for this process, the process becomes received blocked. This function returns the void-pointer passed by the sender and modifies argument `sender` to point to the PCB-entry of the sender.
  - `void reply (PROCESS sender)`
    The receiver replies to a sender. The receiver must have previously received a message from the sender and the sender must be reply blocked.

create_process() - Revisited

- New TOS processes can be created via `create_process()`
- Signature:
```
PORT create_process(void (*func) (PROCESS, PARAM),
                    int prio, PARAM param, char* name)
```
- A previous slide said that `create_process()` should return a NULL pointer as the result.
- This needs to be changed (you will have to modify your implementation for `create_process()`)
- As part of creating a new process, the newly created process should be given a port.
- Use `create_new_port()` to create a port for the new process.
- Save the pointer to this first port in `PCB.first_port`
- Also return the pointer to this first port as the result of `create_process()`

Process States

- When a process is off the ready queue, it is waiting for some event to happen
- To distinguish what the process is waiting for, the process can be in one of different states

State	Description
STATE_READY	This is the only state in which the process is on the ready queue, ready to run
STATE_SEND_BLOCKED	Process executed `send`(), but the receiver is not ready to receive the next message
STATE_REPLY_BLOCKED	Process executed `send`() and the receiver has received the message, but not yet replied
STATE_RECEIVE_BLOCKED	Process executed `receive`(), but no messages are pending
STATE_MESSAGE_BLOCKED	Process executed `message`(), but receiver is not ready to receive the message

Using IPC – Scenario 1

- In the following we show two different scenarios for using the IPC API.
- In scenario 1, the Boot Process creates the Receiver Process.
- Assumptions:
  - These are the only processes in the system.
  - Both processes have priority 1.
- Boot Process calls `send()`. Since the receiver is not ready to receive a message, the sender will become *send blocked* (`STATE_SEND_BLOCKED`).
- When the receiver calls `receive()`, the pending message will be delivered immediately (receiver is not blocked). The sender will remain off the ready queue, but change to state *reply blocked* (`STATE_REPLY_BLOCKED`).
- When the receiver replies via `reply()`, the sender is put back onto the ready queue. When the receiver calls `resign()` subsequently, the Boot Process is scheduled again.

Using IPC – Scenario 1
The Receiver

```
void receiver_process (PROCESS self, PARAM param)
{
    PROCESS sender;
    int* data_from_sender;

    kprintf ("Location C\n");
    data_from_sender = (int*) receive (&sender);
    kprintf ("Received: %d\n“, *data_from_sender);
    reply (sender);
    kprintf ("Location D\n");
    while (1);
}
```

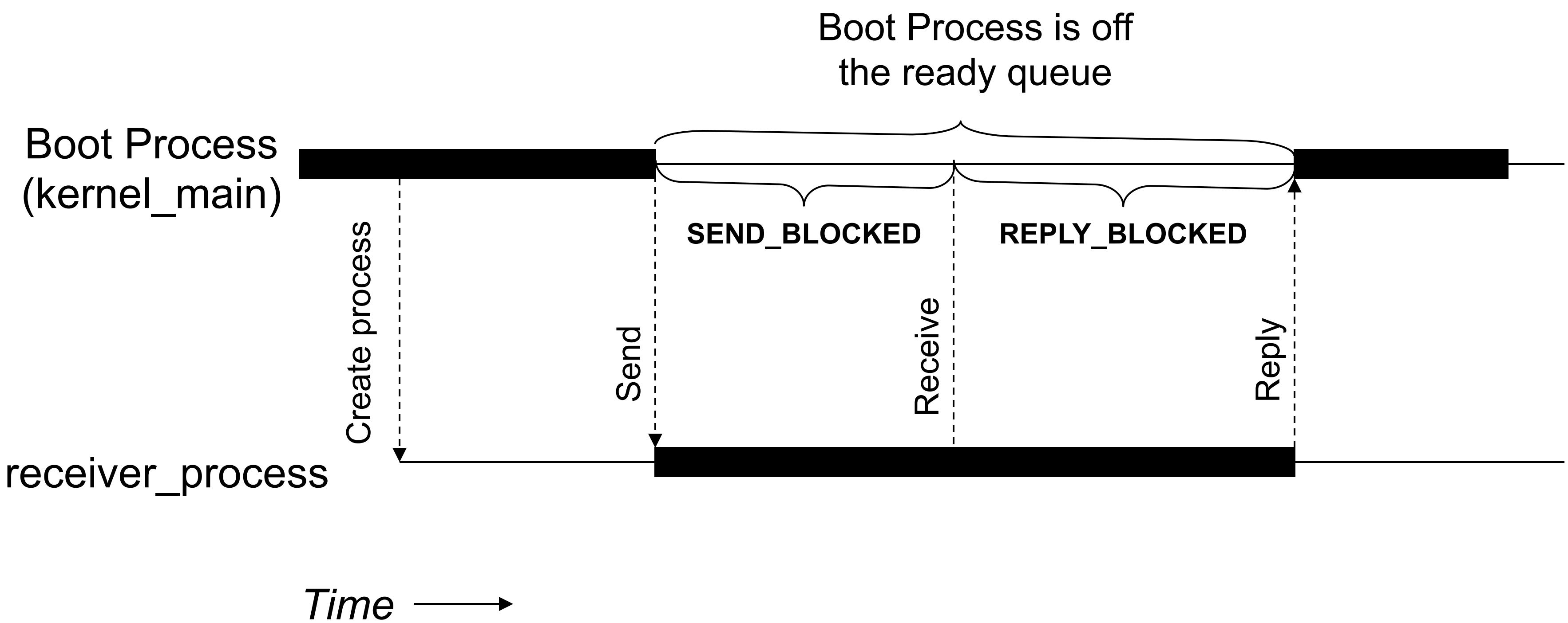
Using IPC – Scenario 1
The Sender

```
void kernel_main()
{
    PORT receiver_port;
    int data = 42;

    init_process();
    init_dispatcher();
    init_ipc();
    receiver_port = create_process (receiver_process,
                                    1, 0, "Receiver");
    kprintf ("Location A\n");
    send (receiver_port, &data);
    kprintf ("Location B\n");
    while (1);
}
```

Output:

```
Location A
Location C
Received: 42
Location B
```

Using IPC – Scenario 1
Time Diagram



Using IPC – Scenario 2

- For scenario 2 we make the same assumptions as for scenario 1.
- The only difference between scenario 1 and scenario 2 is that the Boot Process calls `resign()` after creating the Receiver Process. Otherwise the implementation is unchanged.
- After this call to `resign()`, the Receiver Process is scheduled.
- The Receiver Process calls `receive()`, but there is no message pending. The receiver will be taken off the ready queue and it becomes *receive blocked* (`STATE_RECEIVE_BLOCKED`).
- Scheduler switches back to the Boot Process.
- Boot Process calls `send()`. Since the receiver is waiting for a message, it will be put back onto the ready queue. Since the Boot Process still waits for a reply, it will be taken off the ready queue and becomes *reply blocked* (`STATE_REPLY_BLOCKED`).
- Receiver Process resumes execution after `receive()`.
- When the receiver replies via `reply()`, the sender is put back onto the ready queue. When the receiver calls `resign()` subsequently, the Boot Process is scheduled again.

Using IPC – Scenario 2
The Sender

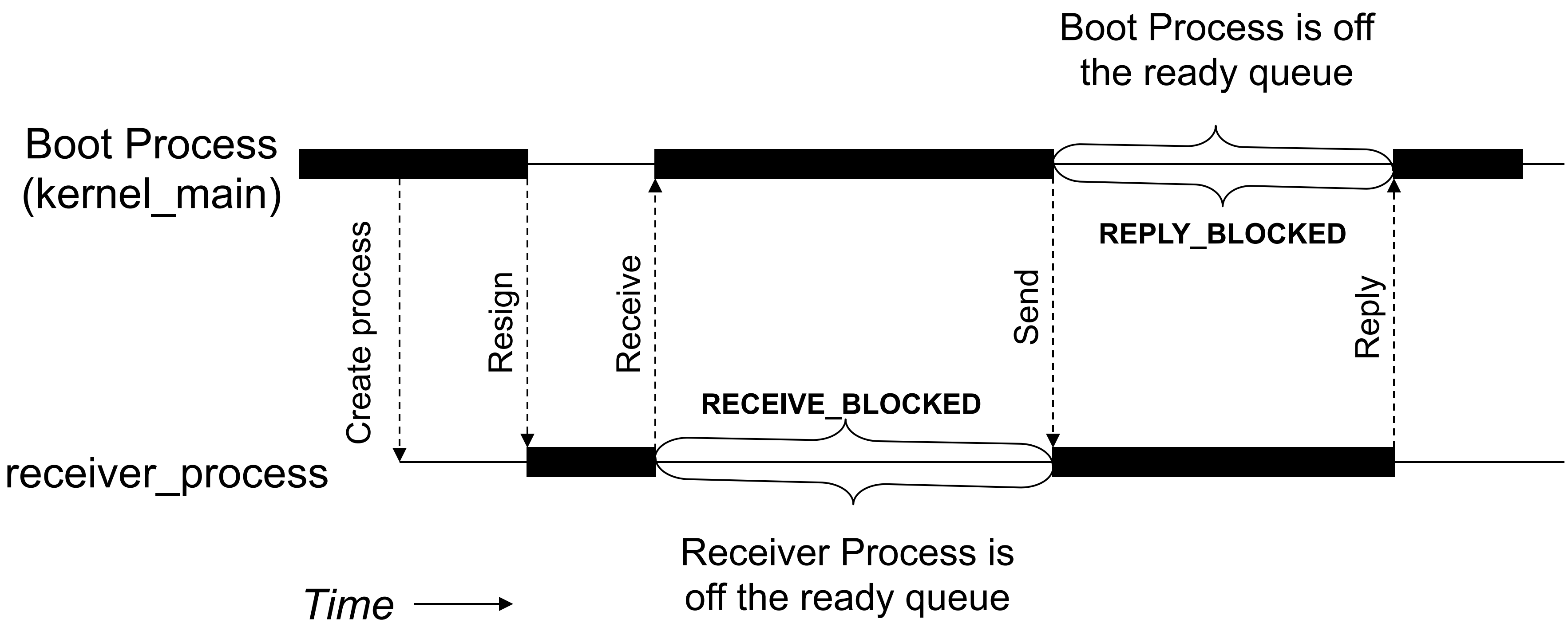
```
void kernel_main()
{
    PORT receiver_port;
    int data = 42;

    init_process();
    init_dispatcher();
    init_ipc();
    receiver_port = create_process (receiver_process,
                                    1, 0, "Receiver");
    resign();
    kprintf ("Location A\n");
    send (receiver_port, &data);
    kprintf ("Location B\n");
    while (1);
}
```

Added ⇨ (points to `resign();`)

Output:

Location C Location A Received: 42 Location B

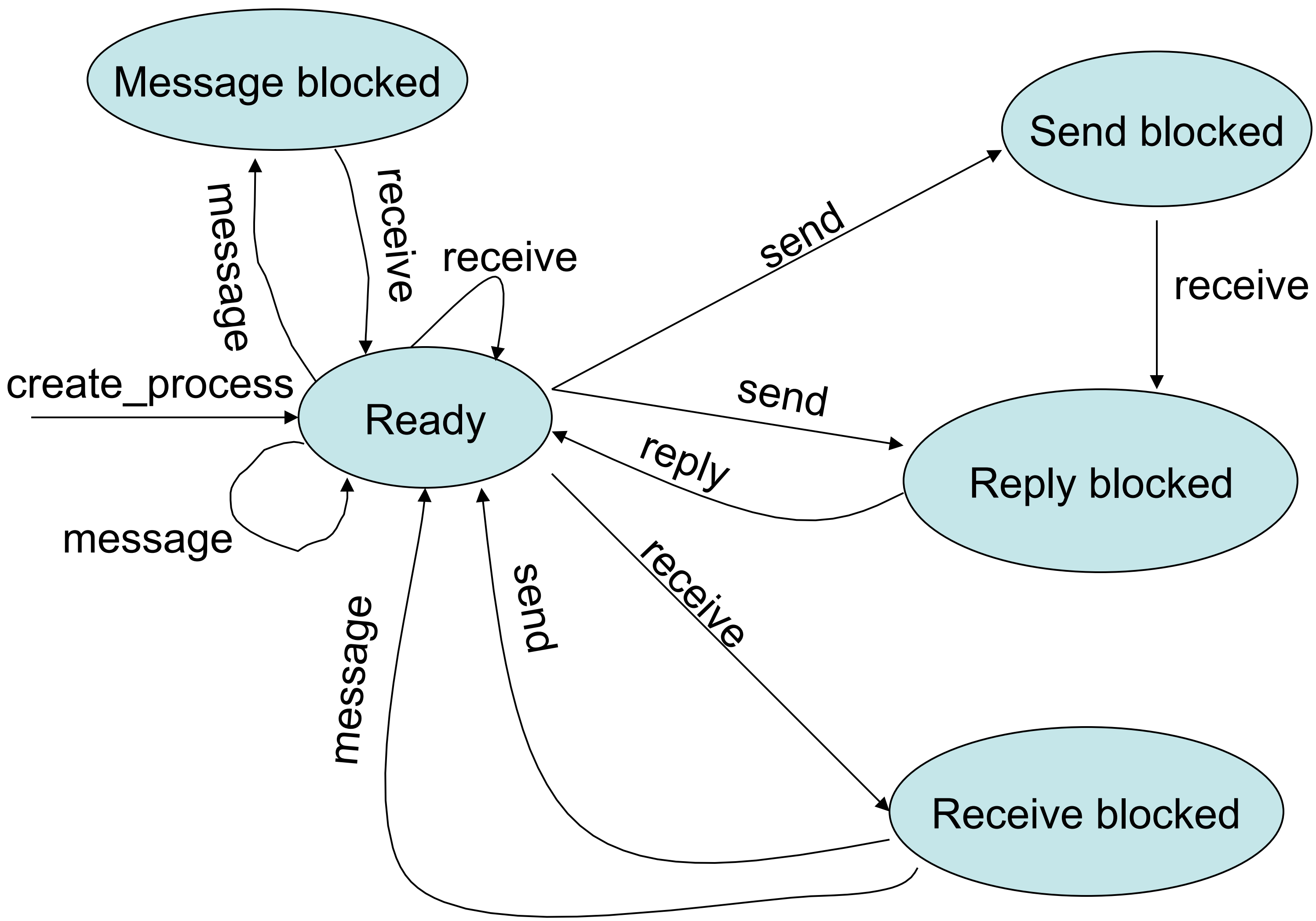
Using IPC – Scenario 2 Time Diagram



print_process()

- The process states explained in the previous scenarios are defined in `~/tos/include/kernel.h`
- Remember `print_process()`? Make sure it knows about those new process states!

State Diagram



Send Blocked List (1)

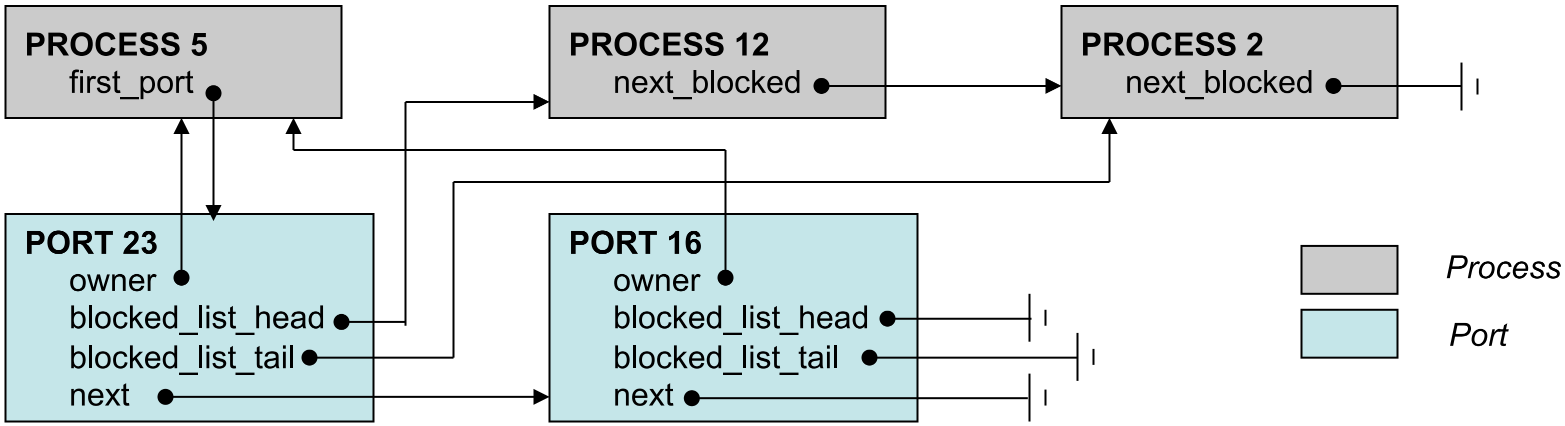
- When a process sends a message, but the receiver is not STATE_RECEIVE_BLOCKED, the sender will become STATE_SEND_BLOCKED (or STATE_MESSAGE_BLOCKED)
- Several processes might be STATE_SEND_BLOCKED on the same receiver process
- When the receiver eventually executes a receive(), one of the STATE_SEND_BLOCKED processes will deliver its message and become STATE_REPLY_BLOCKED
- Problem: how does a receiver process know that there are sender processes waiting to deliver a message to it?
- Solution: there is a *send blocked list* for each port. Processes on this list try to deliver a message to the receiver process.

Send Blocked List (2)

- The send blocked list is a single-linked list:
  - Head: `PORT_DEF.blocked_list_head`
  - Tail: `PORT_DEF.blocked_list_tail`
  - Link to next node: `PCB.next_blocked`
- The tail to the list is maintained in order to efficiently add new processes to the end of the list (why the end?)

```
typedef struct {
    /* ... */
    PROCESS blocked_list_head;
    PROCESS blocked_list_tail;
    /* ... */
} PORT_DEF;

typedef struct {
    /* ... */
    PROCESS next_blocked;
    /* ... */
} PCB;
```

Send Blocked List (3)



- Process 5 owns ports 23 and 16.
- Processes 12 and 2 tried to send a message to process 5 via port 23, but were send blocked. There are no messages pending at port 16.
- Next time process 5 executes a `receive()`, it will receive message from process 12. After delivering the message, process 12 is taken off the send blocked list. Process 12 will then become reply blocked.
- New processes are always added to the end of the send blocked list to ensure fairness.

Pseudo Code for send()

```
send ()
{
    if (receiver is received blocked and port is open) {
       Change receiver to STATE_READY;
       Change to STATE_REPLY_BLOCKED;
    } else {
       Get on the send blocked list of the port;
       Change to STATE_SEND_BLOCKED;
    }
}
```

Pseudo Code for message()

```
message ()
{
    if (receiver is receive blocked and port is open) {
       Change receiver to STATE_READY;
    } else {
       Get on the send blocked list of the port;
       Change to STATE_MESSAGE_BLOCKED;
    }
}
```

Pseudo Code for receive()

```
receive ()
{
    if (send blocked list is not empty) {
       sender = first process on the send blocked list;
       if (sender is STATE_MESSAGE_BLOCKED)
           Change state of sender to STATE_READY;
       if (sender is STATE_SEND_BLOCKED)
           Change state of sender to STATE_REPLY_BLOCKED;
    } else {
       Change to STATE_RECEIVED_BLOCKED;
    }
}
```

Scanning the send blocked list

- One of the things that `receive()` has to do is to see if there are any processes on its send blocked list
- Since a process can own several ports, `receive()` uses the following algorithm to scan its ports:

```
PORT p = active_proc->first_port;
while (p != NULL) {
    if (p->open && p->blocked_list_head != NULL)
        // Found a process on the send blocked list
    p = p->next;
}
// Send blocked list empty. No messages pending.
```

- Note that this algorithm does not guarantee fairness among several ports!

Pseudo Code for reply()

```
reply ()
{
    Add the process replied to back to the ready queue;
    resign();
}
```

Parameter Passing

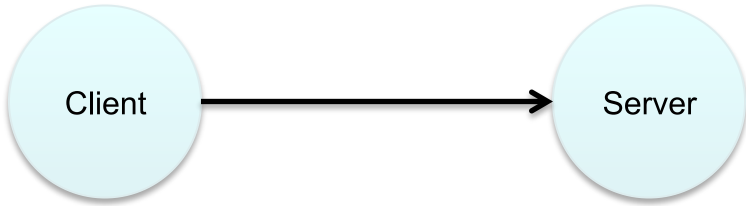
- When processes are added to the ready queue, they are typically woken up in the middle of `send()` or `receive()`.
- It is sometimes necessary to pass the input parameters to `send()` to another process.
- This is accomplished by temporarily storing those parameters in the PCB.

```
typedef struct {
    /* ... */
    PROCESS  param_proc;
    void*    param_data;
    /* ... */
} PCB;
```

Assignment 5

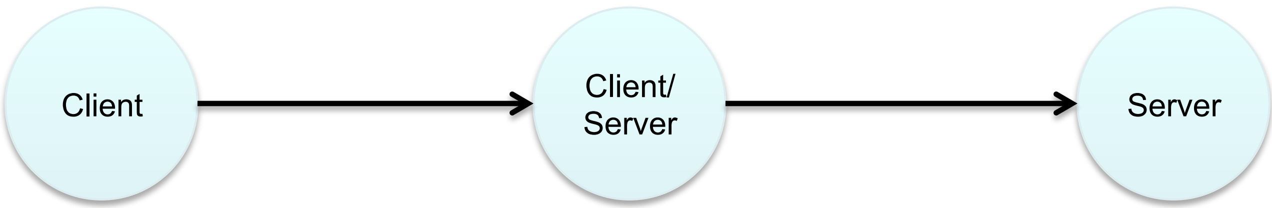
- Implement the functions located in `~/tos/kernel/ipc.c`:
  - `create_port()`
  - `create_new_port()`
  - `open_port()`
  - `close_port()`
  - `send()`
  - `message()`
  - `receive()`
  - `reply()`
- Test cases: `test_ipc_[1-6]`

Collaboration Patterns



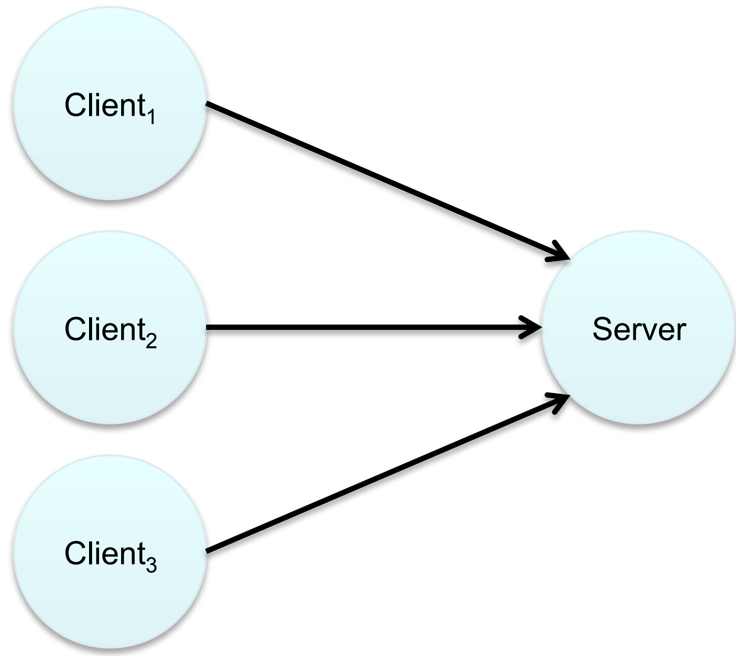
- TOS's IPC allows for different collaboration patterns that define how processes interact with one another.
- Simplest case (see above diagram): one client (executing send()/message()) and one server (executing receive()/reply()).
- It does not matter if client sends message first or receiver first tries to receive message. Client and server are synchronized at the *rendezvous point*. For that reason this form of IPC implements *synchronous communication*.

Delegation



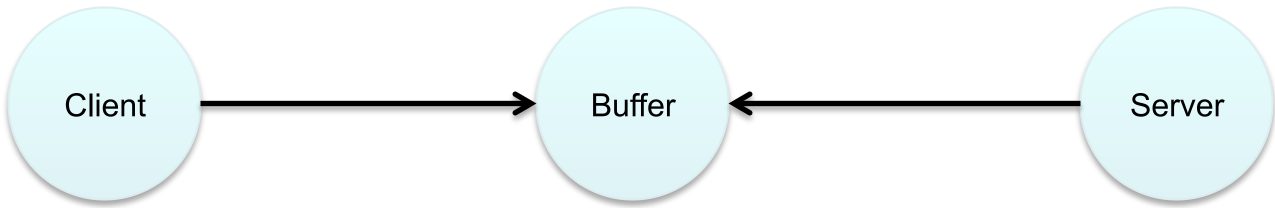
- Client and server are roles.
- A process can be both in the role of a client and a server at different points in times.
- E.g., a process could break up a job into multiple sub-jobs and delegate each to a different process.

Worker Process



- Multiple clients contact same server.
- E.g., worker process (server) implements a file system. Clients perform file I/O operations.
- Server will only process one request at a time (i.e., receive() will only ever return one message; other clients remain on the send blocked list)
- IPC will synchronize among several clients.
- Server encapsulates a shared resource (e.g., file system, printer, etc)

Asynchronous Communication

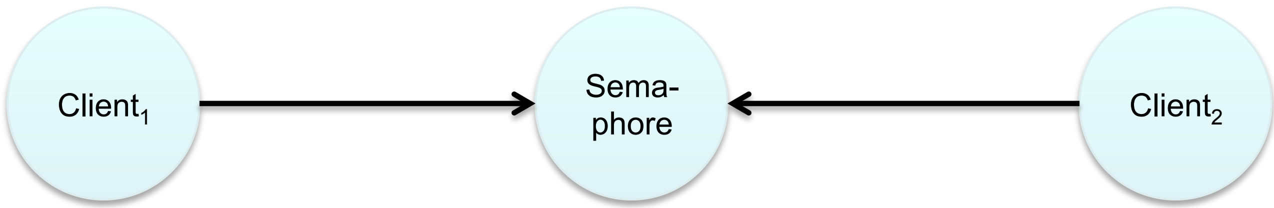


- Since client and server need a rendezvous point to deliver a message, either one will have to be blocked until the other is ready.
- This is called *synchronous communication*.
- *Asynchronous communication* can be achieved by adding a special buffer process and reversing the roles of send/receive/reply for the server.
- In the diagram, Client can send the Server a message via the Buffer Process.
- Buffer Process will only use receive() and reply() but never send()!
- Server will use send() to request the next message.
- Note that Buffer Process may need to buffer messages if one process is sending faster than the other receives.
- Bounded buffer: throttle sender.

Pseudo Code for Buffer Process

```
Buffer messages = [];
while (1) {
    msg = receive();
    if (msg is from client) {
        messages.add(msg);
        reply;
        continue;
    }
    if (msg is from server) {
        if (messages == []) {
            reply with empty message;
        } else {
            nextMsg = messages.dequeue();
            reply(nextMsg);
        }
    }
}
```

Mutual Exclusion



- Replies do not have to be sent to clients in the same order in which their messages were delivered.
- This is called *out-of-order replies*.
- This can be used to implement mutual exclusion via a special Semaphore Process.
- Semaphore Process will only ever call receive() and reply().
- A client requesting entry to the critical section sends an acquire message to the Semaphore Process.
- If entry is granted, Semaphore Process will reply.
- Other clients who wish to enter the critical section will be kept reply blocked.
- When a client exits the critical section, it sends a release message to the Semaphore Process who then replies to another client that waits for entry.

Pseudo Code for Semaphore Process

```
PROCESS waiting = [];
bool process_in_critical_section = false;
while (1) {
    msg = receive();
    if (msg is of type acquire) {
        if (process_in_critical_section) {
            waiting.add(process that sent msg);
        } else {
            process_in_critical_section = true;
            reply(process that sent msg);
        }
    }
    if (msg is of type release) {
        if (waiting == []) {
            process_in_critical_section = false;
        } else {
            next_process = waiting.dequeue();
            reply(next_process);
        }
    }
}
```