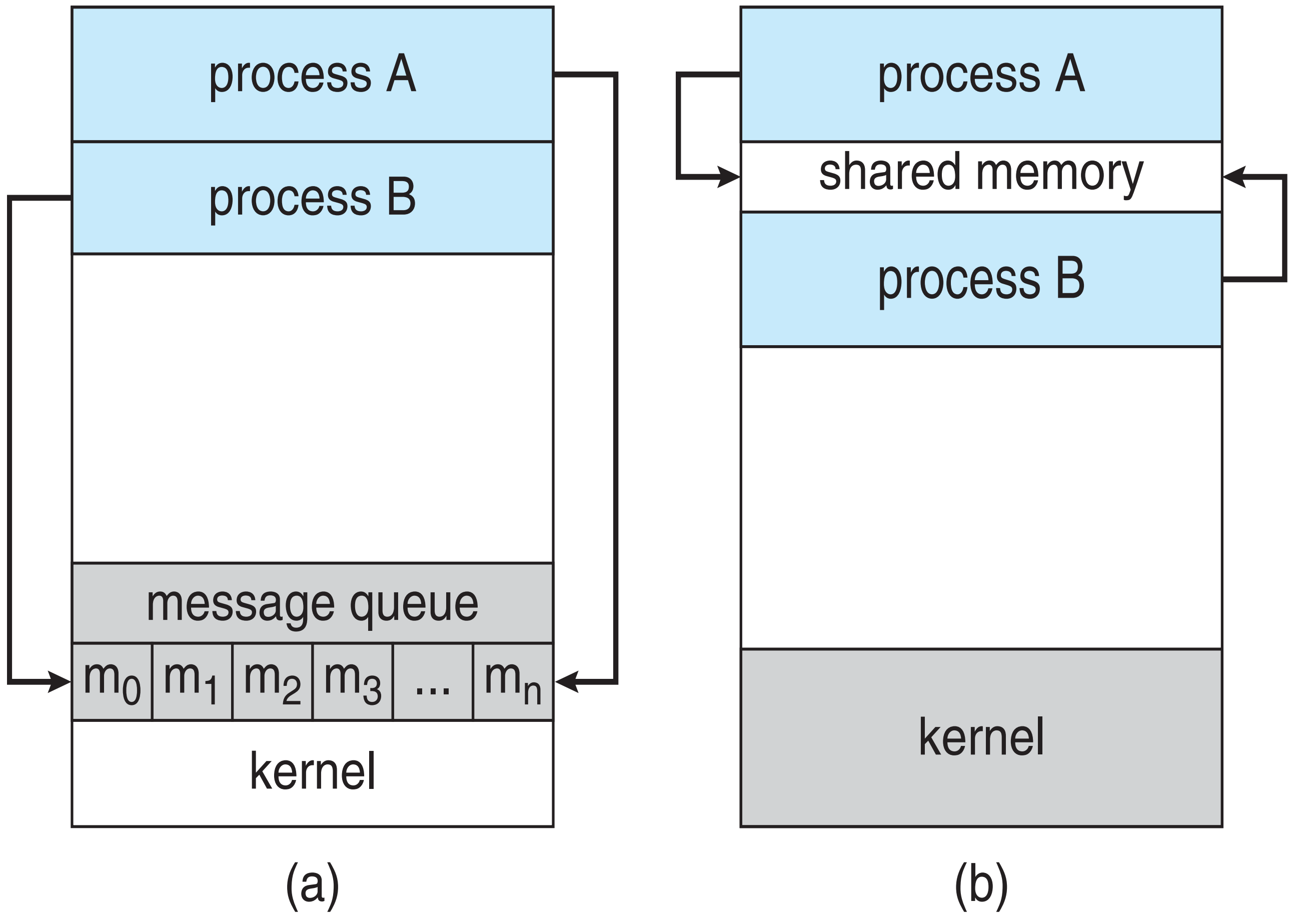
Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % BUFFER_SIZE) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out); /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Interprocess Communication – Message Passing

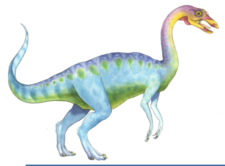
- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering



Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Synchronization (Cont.)

- Producer-consumer becomes trivial

```
  message next_produced;
  while (true) {
       /* produce an item in next produced */
  send(next_produced);
  }

message next_consumed;
while (true) {
   receive(next_consumed);

   /* consume the item in next consumed */
}
```

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Examples of IPC Systems - POSIX

- POSIX Shared Memory
  - Process first creates shared memory segment

```
shm_fd = shm_open(name, O CREAT | O RDWR, 0666);
```

  - Also used to open an existing segment to share it
  - Set the size of the object

```
ftruncate(shm fd, 4096);
```

  - Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared
memory");
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

Possible Design for Homework 4

User provides Process ID

Create the receiving Mailbox:
mq_open(ID, O_NONBLOCK | O_RDONLY | O_CREAT | O_EXCL, 0666, &attr);
Example: Process P0 has ID = /70

```
struct mq_attr attr;
    attr.mq_flags = O_NONBLOCK;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 15;
    attr.mq_curmsgs = 0;
```

Build mapping between communicating processes
Example: P0 informs P1 and P2 about its Mailbox ID

Pass mapping to a component that handles the Communications and finishes once the algorithm is over

Process P0 starts!

Heads-up working with Mailboxes

- When the program is over, don't forget to:

```
mq_close(mqMailbox);
mq_unlink(ID);
```

- POSIX message queues have kernel persistence: if not removed by mq_unlink(…), a message queue will exist until the system is shut down.
- If you do not remove it, or your program terminates unexpectedly, the created mailboxes are still there. Make sure to handle this case once you open it.
  - Always check the return value of the mq_open system call

Possible Design for Homework 4

Mailbox Manager

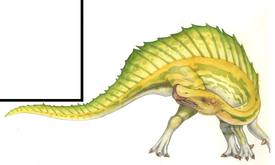
$\downarrow$

$$\begin{tikzcd}
\text{Am I P0?} \arrow[r, "\text{Yes}"] \arrow[d, "\text{No}"] & {} \\
{} &
\end{tikzcd}$$

Wait for a message to be received on the process's Mailbox(the Read-only one):
mq_receive(m_Mailbox, &msg_buf[0], 100, 0);
The call may return immediately if O_NONBLOCK has been specified, thus wrap the call into a loop

```
while(1) {
        if(m_Exit) {
            break;
        }
        msg_length = mq_receive(m_Mailbox, &msg_buf[0], 100, 0);
        if(msg_length >= 0) {
            processMsg(msg_buf);
        } else if(errno != EAGAIN){
            cout << "Error: " << strerror(errno) << endl;
            exit(-1);
        }
    }
```

$\downarrow$



Once a temperature is received, you compute some simple math function and propagate the result to other processes (write to other processes' Mailboxes).

If it's the first time, then you first open the needed mailboxes for writing

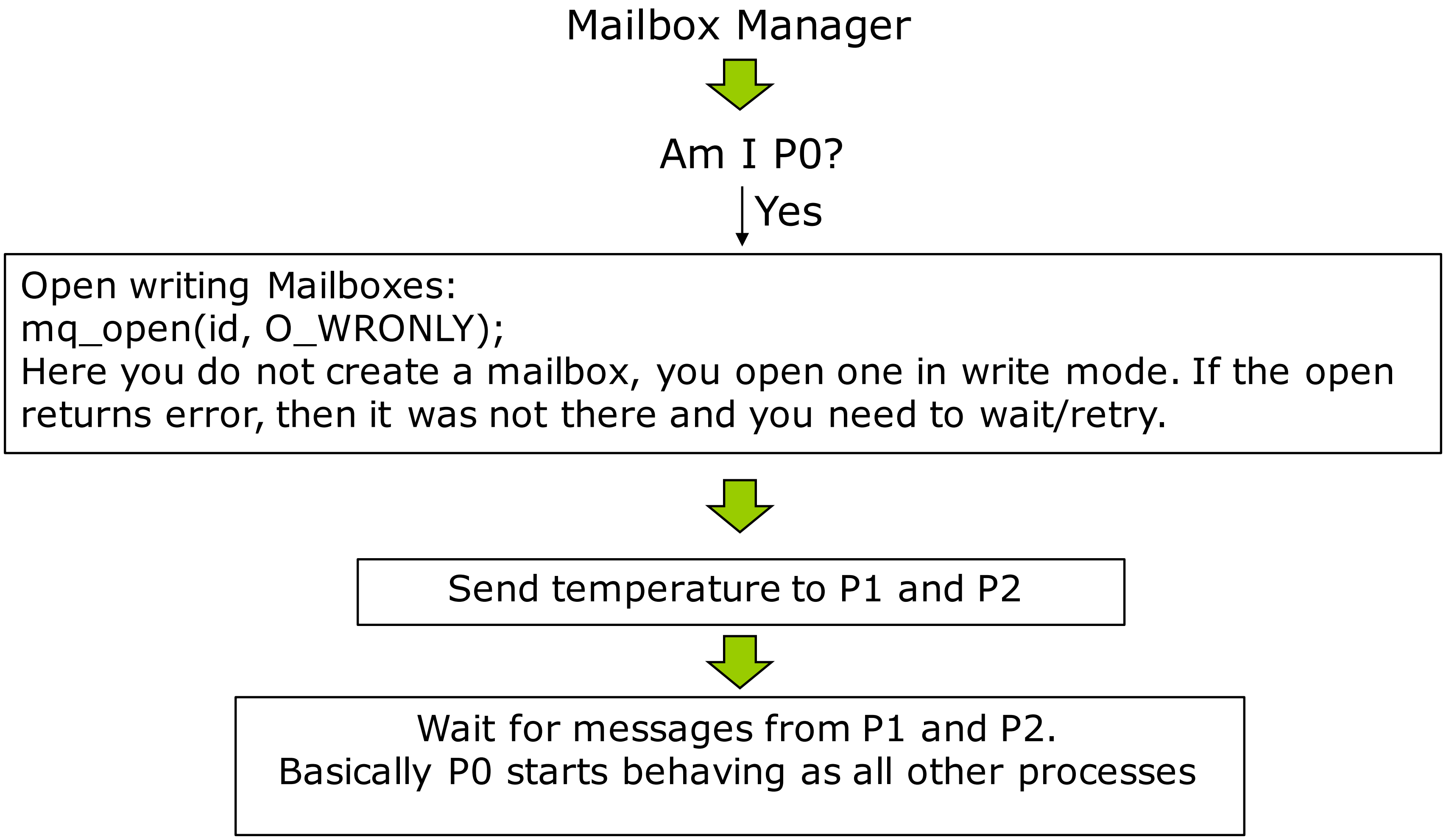
```
mailbox = mq_open(id, O_WRONLY);
```

Example: P1 writes to P0's, P3's, and P4's Mailboxes
You can use the mapping built at the beginning to know the Mailboxes each process is interested to

To send the result of the math function:

```
mq_send(dest, msg_buf, length+1, 0);
```

Possible Design for Homework 4



Mailbox Manager

Am I P0?

Yes

Open writing Mailboxes:
mq_open(id, O_WRONLY);
Here you do not create a mailbox, you open one in write mode. If the open returns error, then it was not there and you need to wait/retry.

Send temperature to P1 and P2

Wait for messages from P1 and P2.
Basically P0 starts behaving as all other processes

End of Chapter 3