CPE 460 Operating System Design

Hey Process, Can we chat?

Ahmed Tamrawi

Hey Process! Can we communicate?

*Processes executing concurrently in the operating system may be either **independent** processes or **cooperating** processes*

Why do we need cooperating processes?

- Information sharing
- Computation speedup
- Modularity
- Convenience

A **process is independent** if it cannot affect or be affected by the other processes executing in the system.

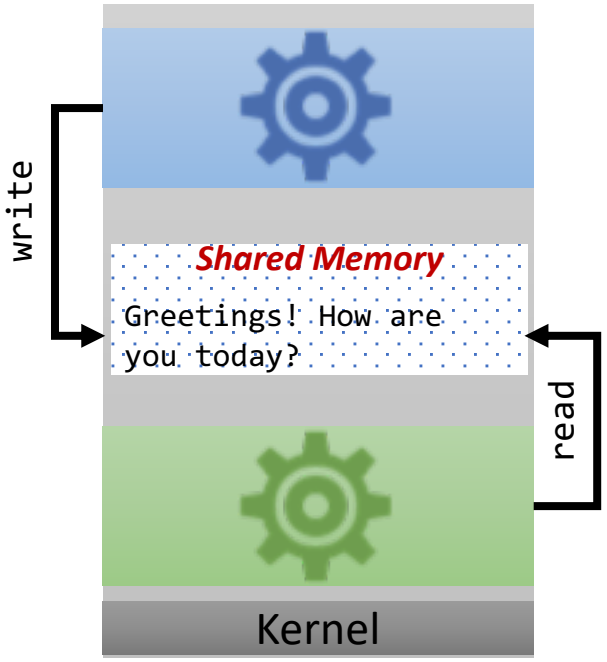
Any process that does not share data with any other process is independent

A **process is cooperating** if it can affect or be affected by the other processes executing in the system.

Any process that shares data with other processes is a cooperating process

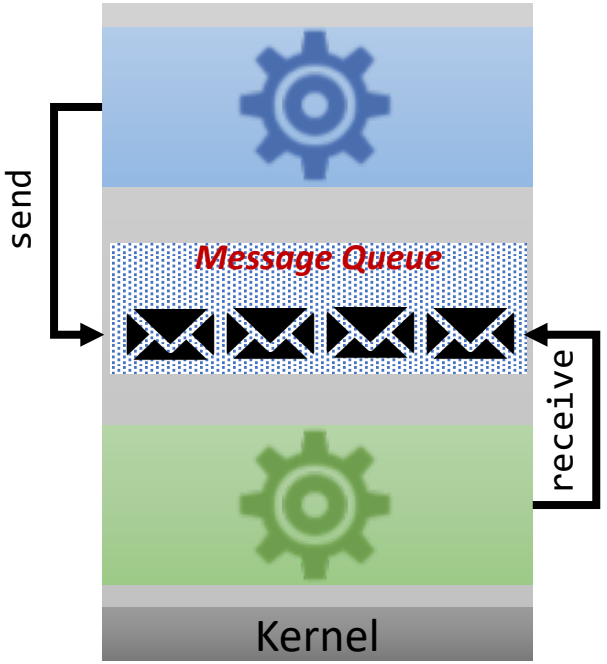
Cooperating processes need interprocess communication (IPC)

The operating system provides multiple mechanisms that allow processes to exchange data and information



Speed

Many Implementations



Shared Memory

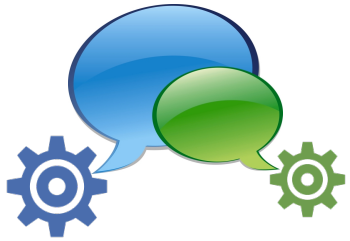
A region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region

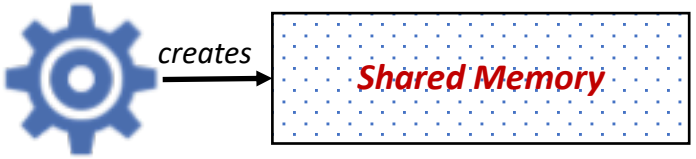
Message Passing

Communication takes place by means of messages exchanged between the cooperating processes

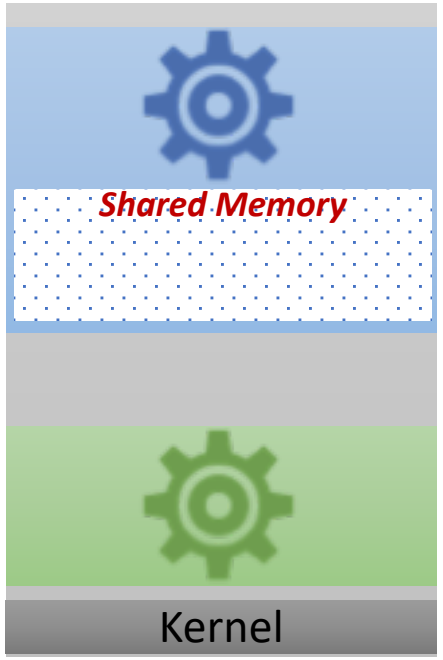
Shared Memory

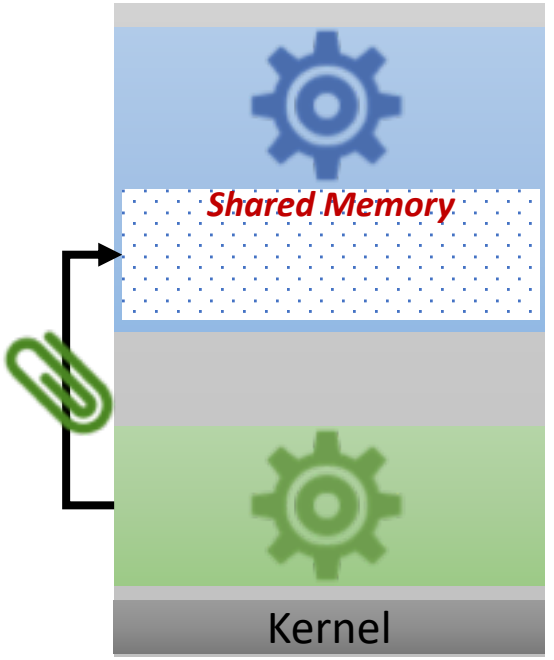
An area of memory shared among the processes that wish to communicate

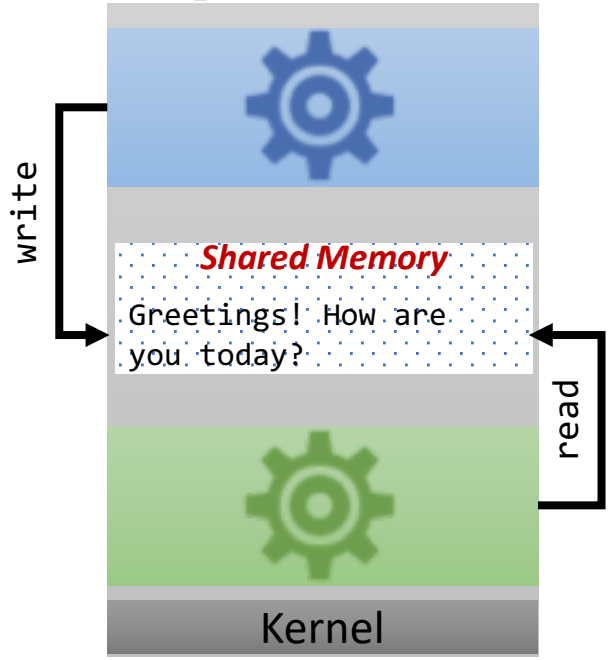




Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment.







OS **prevents** one process from accessing another process's memory

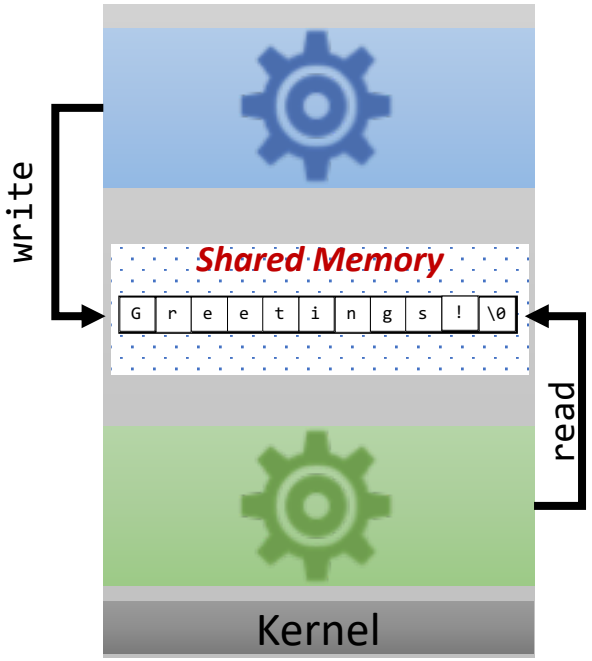
Shared Memory

An area of memory shared among the processes that wish to communicate

Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

```
while(1){
    if(buffer[0] == '\0'){
        sprintf(buffer, "Greetings!");
    }else{
        // do nothing
    }
}
```

```
while(1){
    if(buffer[0] == '\0'){
        // do nothing
    }else{
        printf(buffer);
    }
}
```



Bounded-Buffer

assumes that there is a fixed buffer size

char* buffer[10]

G	r	e	e	t	i	n	g	s	!	\0

The consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

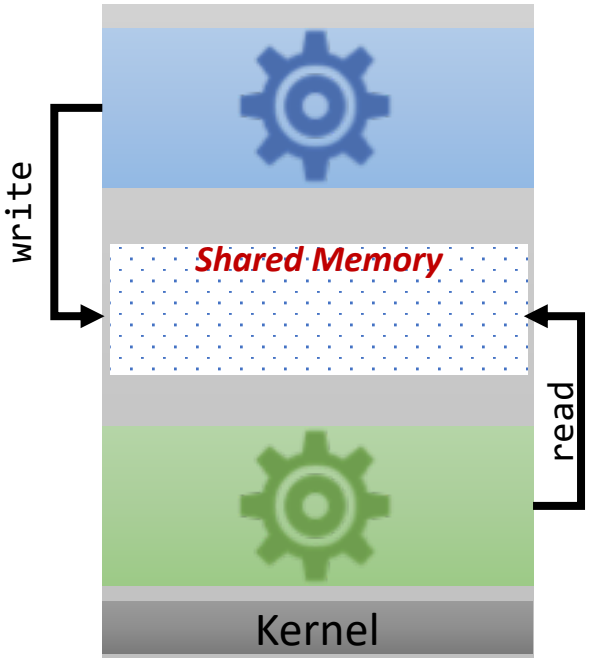
Shared Memory

An area of memory shared among the processes that wish to communicate

Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory

```
while(1){
    /* produce new entry */
    char * newEntry = ...;
    list.add(newEntry);
    in++;
}
```

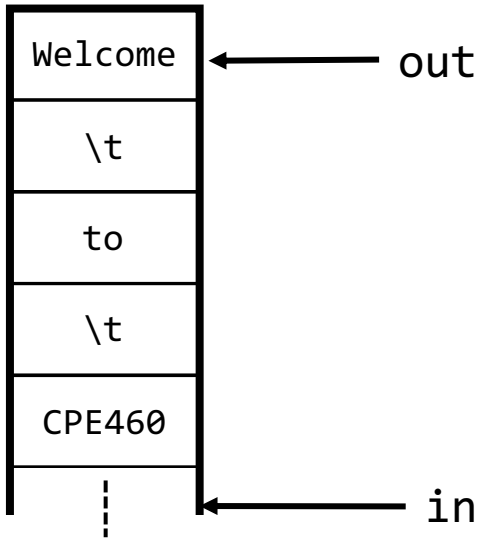
```
while(1){
    if(in == out){
        // do nothing
    }else{
        char *entry = list.get(out);
        out++;
        printf(entry);
    }
}
```



Unbounded-Buffer

places no practical limit on the size of the buffer

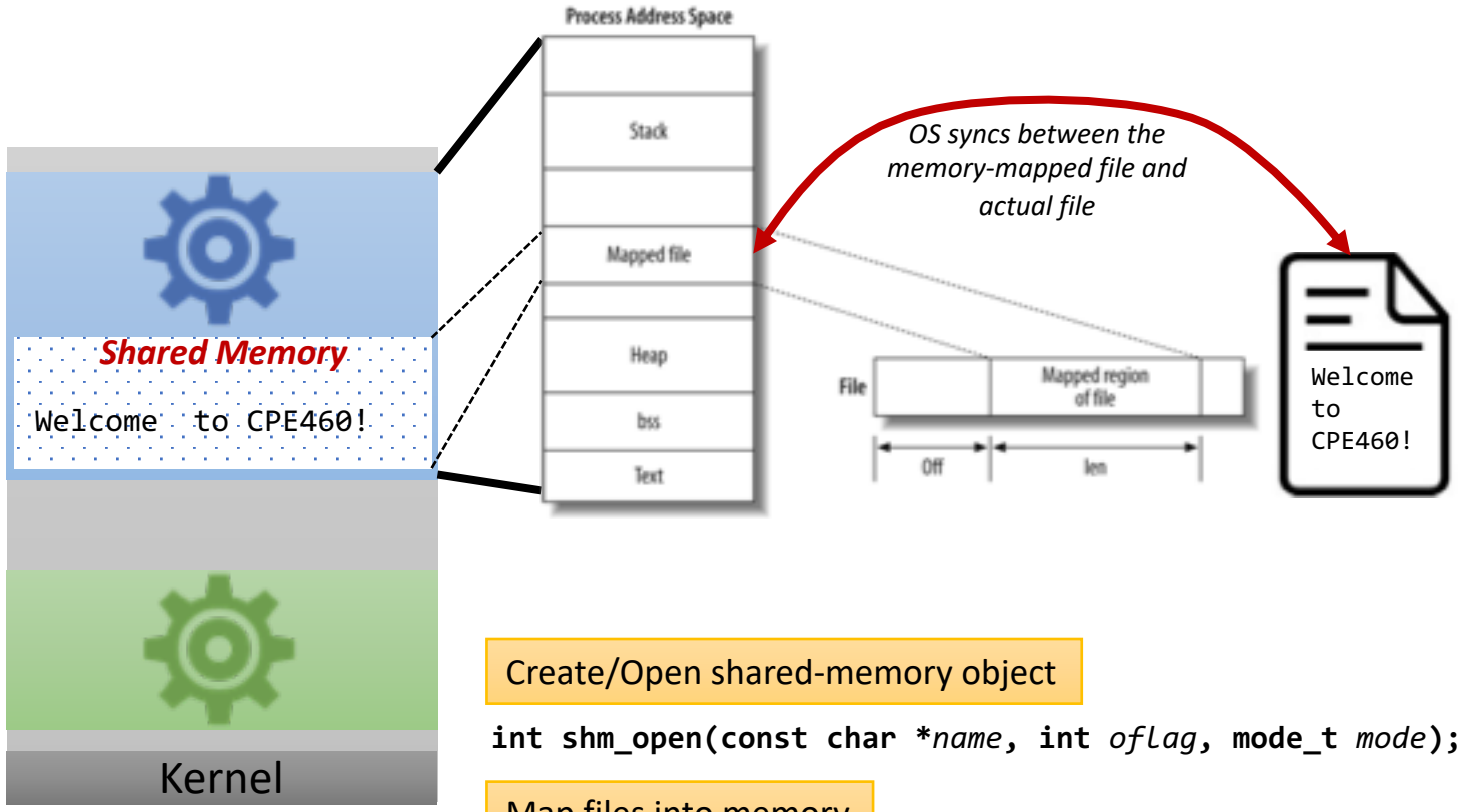
```
List<char*> list;
int in = 0;
int out = 0;
```



The consumer may have to wait for new items, but the producer can always produce new items.

POSIX Shared Memory

*POSIX shared memory is organized using **memory-mapped files**, which associate the region of shared memory with a file*



Create/Open shared-memory object

```
int shm_open(const char *name, int oflag, mode_t mode);
```

Map files into memory

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

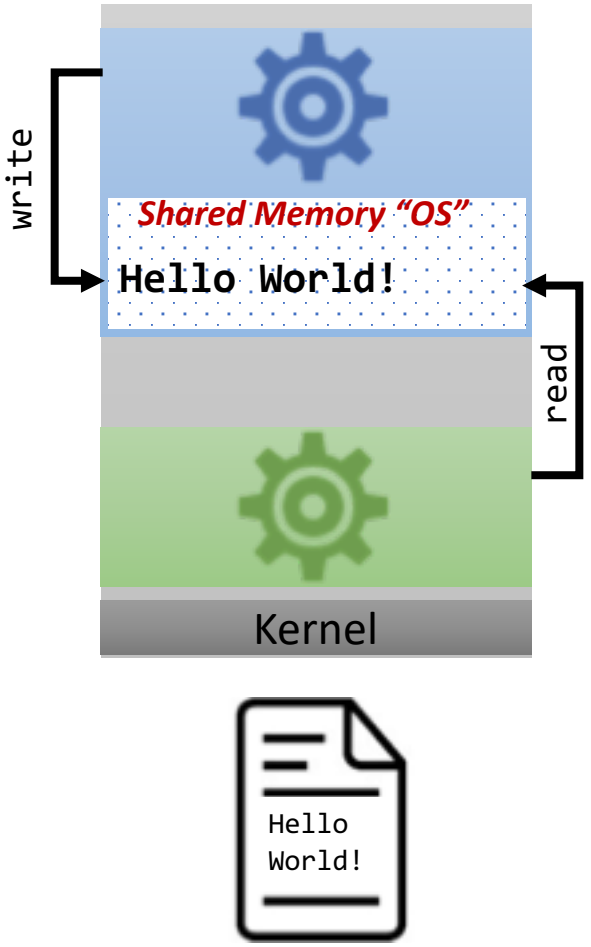
https://github.com/torvalds/linux/blob/master/ipc/shm.c

https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s03.html

POSIX Shared Memory

```c
#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;
}
```

Producer

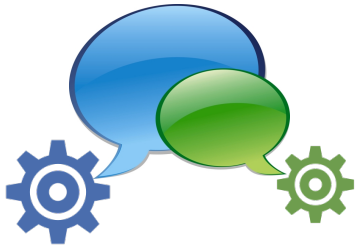



```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

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;
}
```

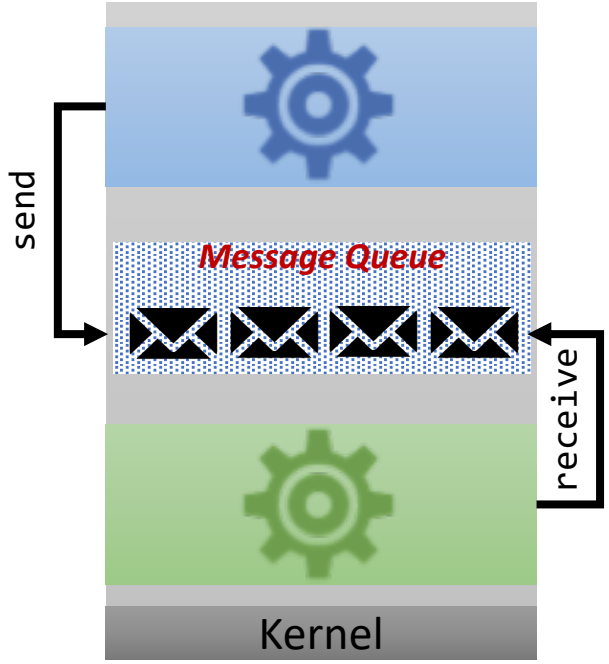
Consumer

Message Passing

Communication takes place by means of messages exchanged between the cooperating processes



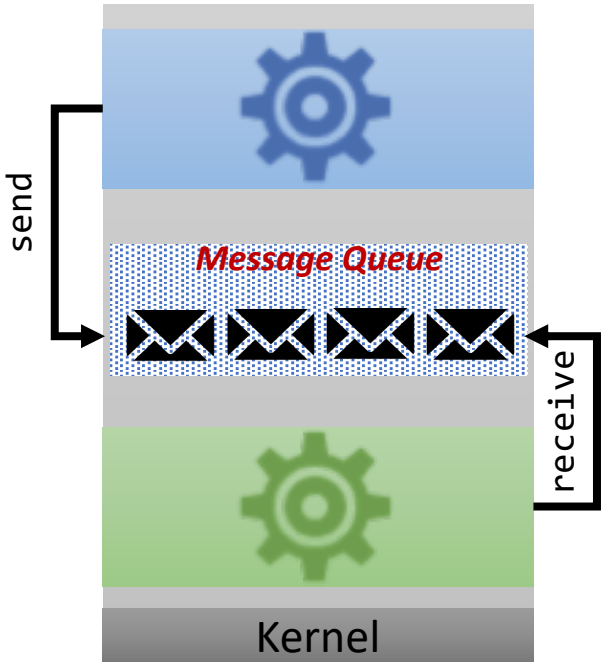
1. Establish a communication link between them
2. Exchange messages via `send/receive`



Implementation issues:

- How are communication 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 a link unidirectional or bi-directional?

Message Passing Communication



Communication Link

Direct Communication

Processes must name each other explicitly

`send(`⚙, ✉`)` OR `receive(`⚙, ✉`)`

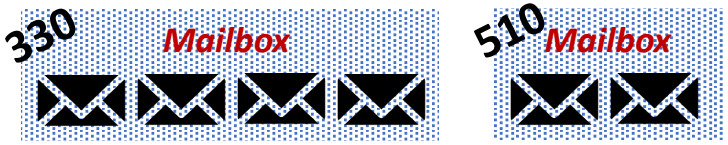
Properties of communication link

- 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

Hard-Coding & Less Desirable

Indirect Communication

Messages are directed and received from mailboxes (ports)

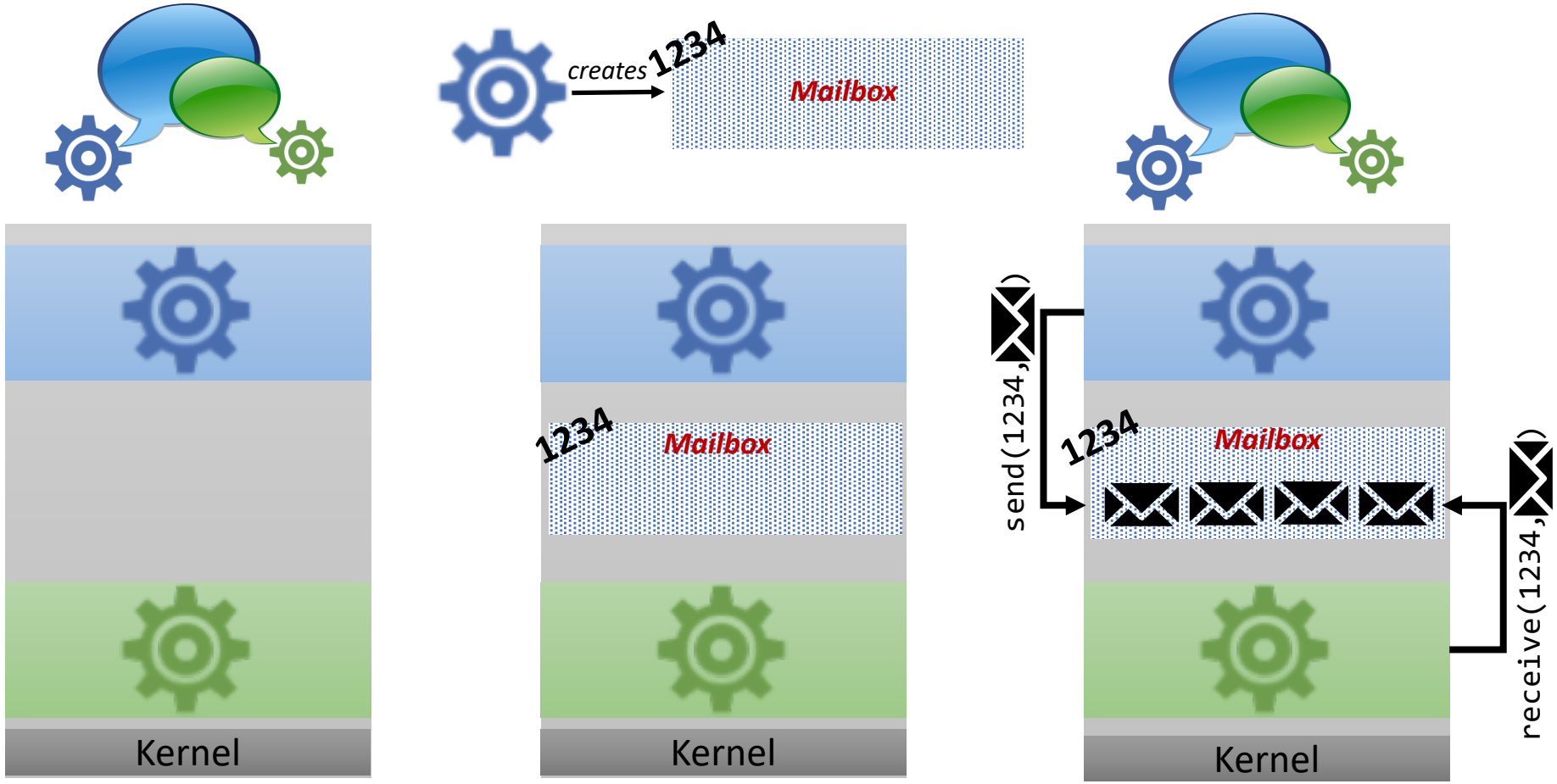


Properties of communication link

- Link are 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

Flexible & More Desirable

Message Passing Operations



Message Passing Synchronization

Blocking (Synchronous) Message Passing

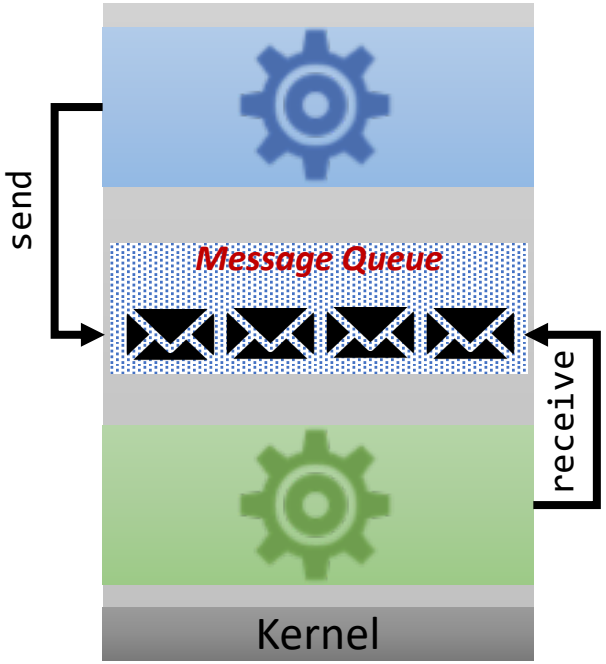
`send` - *the sender is blocked until the message is received*
`receive` - *the receiver is blocked until a message is available*

Non-blocking (Asynchronous) Message Passing

`send` - *the sender sends the message and continue*
`receive` - *the receiver receives: a valid message, or Null message*

If both send and receive are blocking, we have a **rendezvous**

Message Passing Buffering



Zero Capacity
no messages are queued on a link. Sender must wait for receiver (rendezvous)

Unbounded capacity
infinite length. Sender never waits

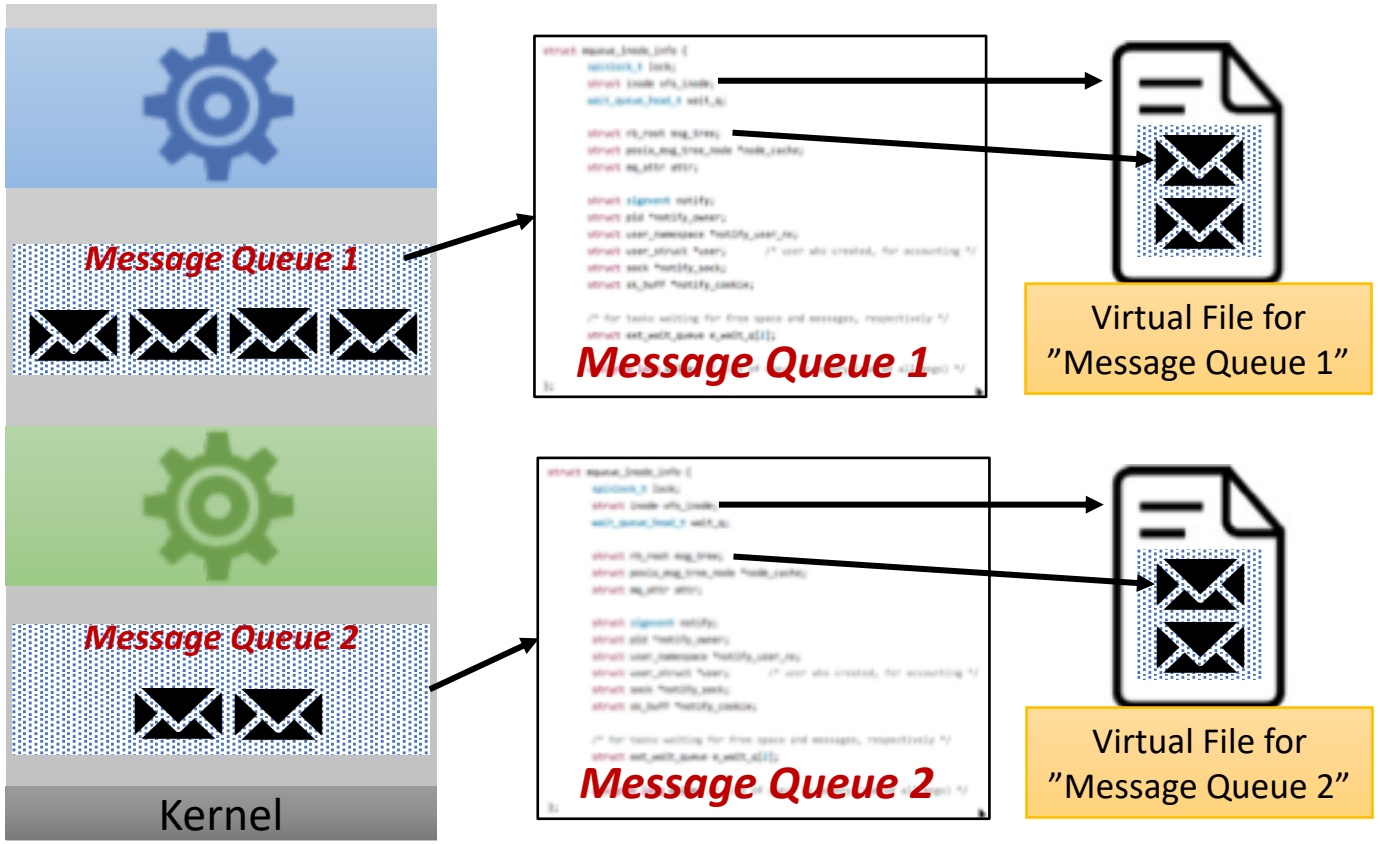
Bounded Capacity
Finite length messages Sender must wait if link full

POSIX Message Queues

*POSIX message queues is organized using **virtual file system**, and each message queue is pointed to by an* `mqueue_inode_info` *data structure*

Virtual File System

`/proc/sys/fs/mqueue/`

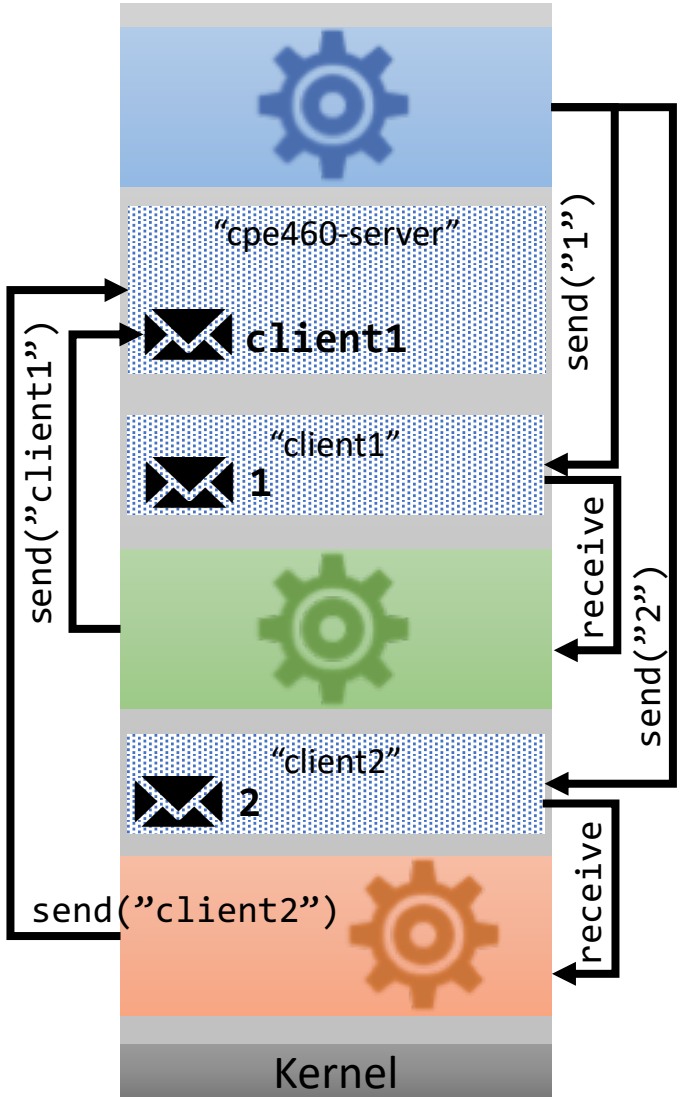


https://github.com/torvalds/linux/blob/master/ipc/mqueue.c

POSIX Message Queues

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

int main (int argc, char **argv)
{
    mqd_t qd_server, qd_client;   // queue descriptors
    long token_number = 1; // next token to be given to client

    printf ("Server: Hello, World!\n");

    struct mq_attr attr;

    attr.mq_flags = 0;
    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 256;
    attr.mq_curmsgs = 0;

    if ((qd_server = mq_open ("/cpe460-server", O_RDONLY | O_CREAT, 0660, &attr)) == -1) {
        perror ("Server: mq_open (server)");
        exit (1);
    }
    char in_buffer [300];
    char out_buffer [300];

    while (1) {
        // get the oldest message with highest priority
        if (mq_receive (qd_server, in_buffer, 300, NULL) == -1) {
            perror ("Server: mq_receive");
            exit (1);
        }

        printf ("Server: message received.\n");

        // send reply message to client

        if ((qd_client = mq_open (in_buffer, O_WRONLY)) == 1) {
            perror ("Server: Not able to open client queue");
            continue;
        }

        sprintf (out_buffer, "%ld", token_number);

        if (mq_send (qd_client, out_buffer, strlen (out_buffer), 0) == -1) {
            perror ("Server: Not able to send message to client");
            continue;
        }

        printf ("Server: response sent to client.\n");
        token_number++;
    }
}
```

Producer



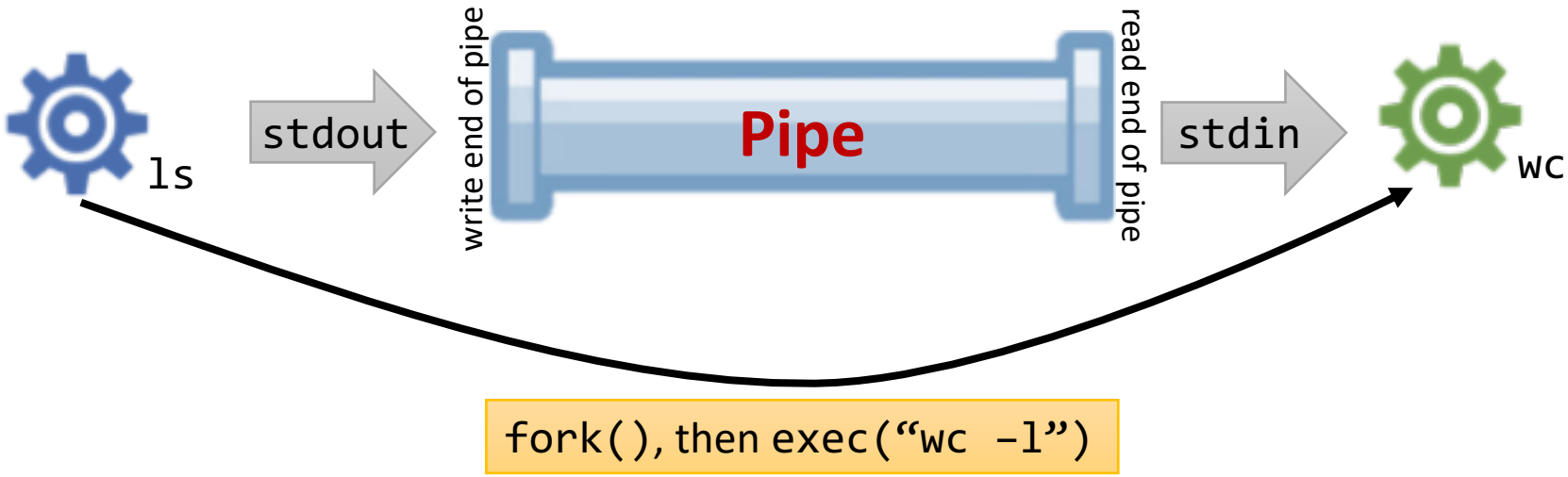

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <mqueue.h>

int main (int argc, char **argv)
{
    char client_queue_name [64];
    mqd_t qd_server, qd_client;   // queue descriptors
    // create the client queue for receiving messages from server
    sprintf (client_queue_name, "/sp-example-client-%d", getpid ());

    struct mq_attr attr;

    attr.mq_flags = 0; attr.mq_maxmsg = 10; attr.mq_msgsize = 256; attr.mq_curmsgs = 0;

    if ((qd_client = mq_open (client_queue_name, O_RDONLY | O_CREAT, 0660, &attr)) == -1) {
        perror ("Client: mq_open (client)");
        exit (1);
    }

    if ((qd_server = mq_open ("/cpe460-server", O_WRONLY)) == -1) {
        perror ("Client: mq_open (server)");
        exit (1);
    }

    char in_buffer [300]; char temp_buf [10];
    printf ("Ask for a token (Press <ENTER>): ");

    while (fgets (temp_buf, 2, stdin)) {
        // send message to server
        if (mq_send (qd_server, client_queue_name, strlen (client_queue_name), 0) == -1) {
            perror ("Client: Not able to send message to server");
            continue;
        }
        // receive response from server
        if (mq_receive (qd_client, in_buffer, 300, NULL) == -1) {
            perror ("Client: mq_receive");
            exit (1);
        }
        // display token received from server
        printf ("Client: Token received from server: %s\n\n", in_buffer);
        printf ("Ask for a token (Press ): ");
    }
    if (mq_close (qd_client) == -1) {
        perror ("Client: mq_close");
        exit (1);
    }

    if (mq_unlink (client_queue_name) == -1) {
        perror ("Client: mq_unlink");
        exit (1);
    }
    printf ("Client: bye\n");
}
```

Consumer

https://www.softprayog.in/programming/interprocess-communication-using-posix-message-queues-in-linux

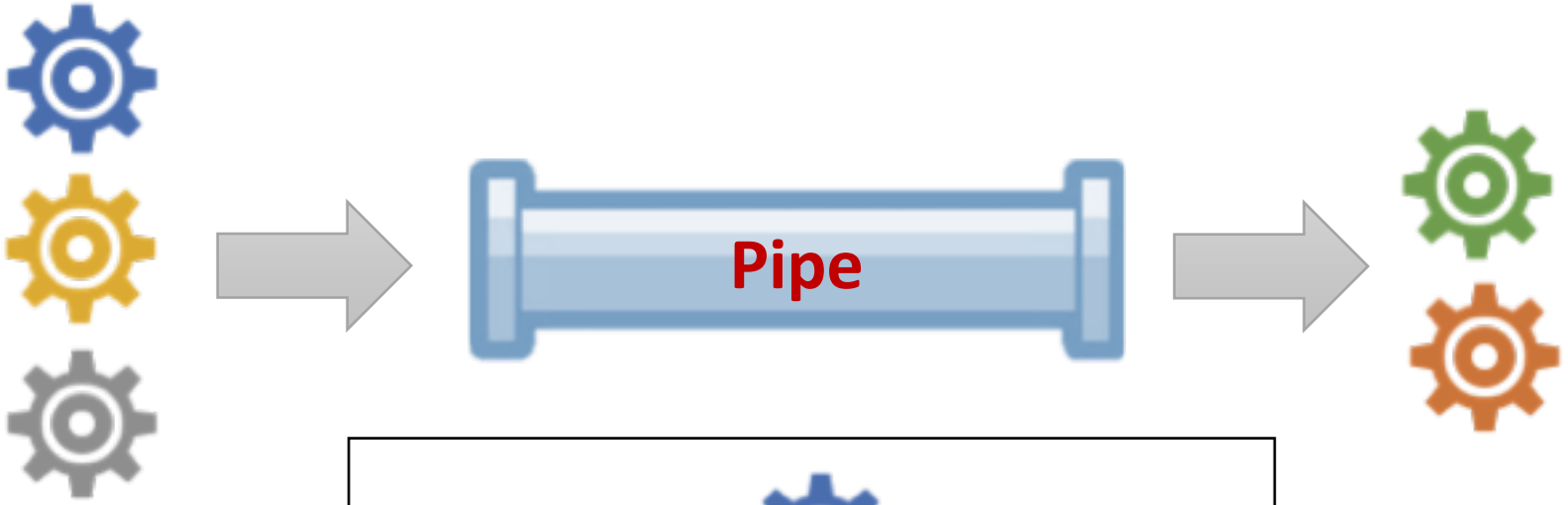
```
$ ls | wc -l
```

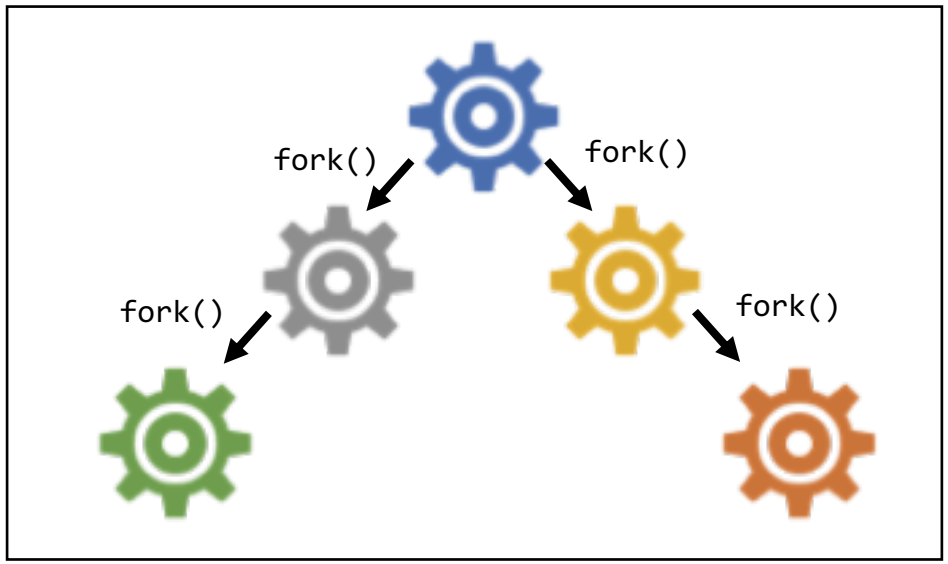
Counts the number of files in a directory



Pipes

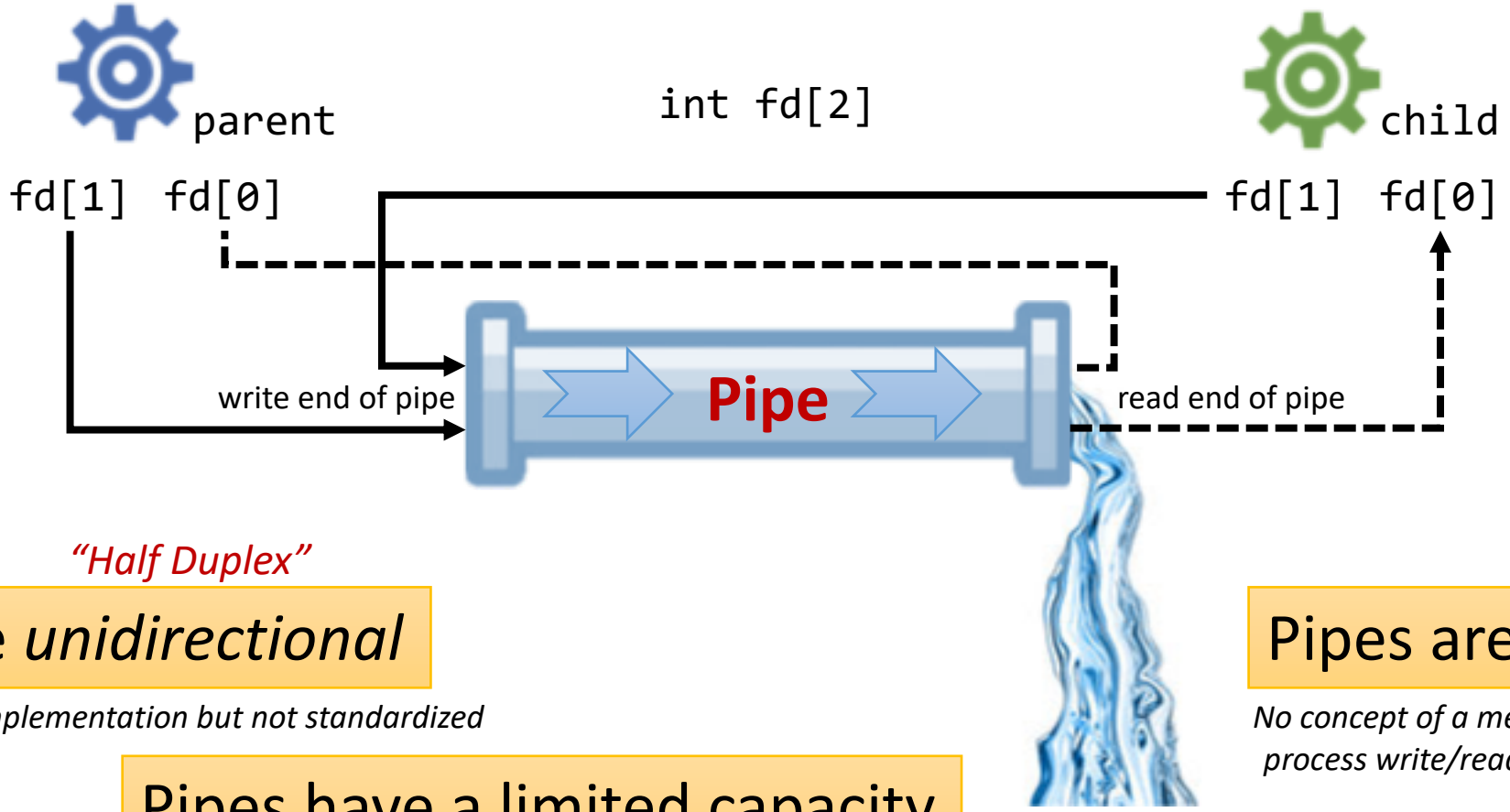
*A **pipe** acts as a conduit allowing **related processes** to communicate*





Pipes

*A conduit allowing **related processes** to communicate*



"Half Duplex"

Pipes are *unidirectional*

There are bidirectional implementation but not standardized

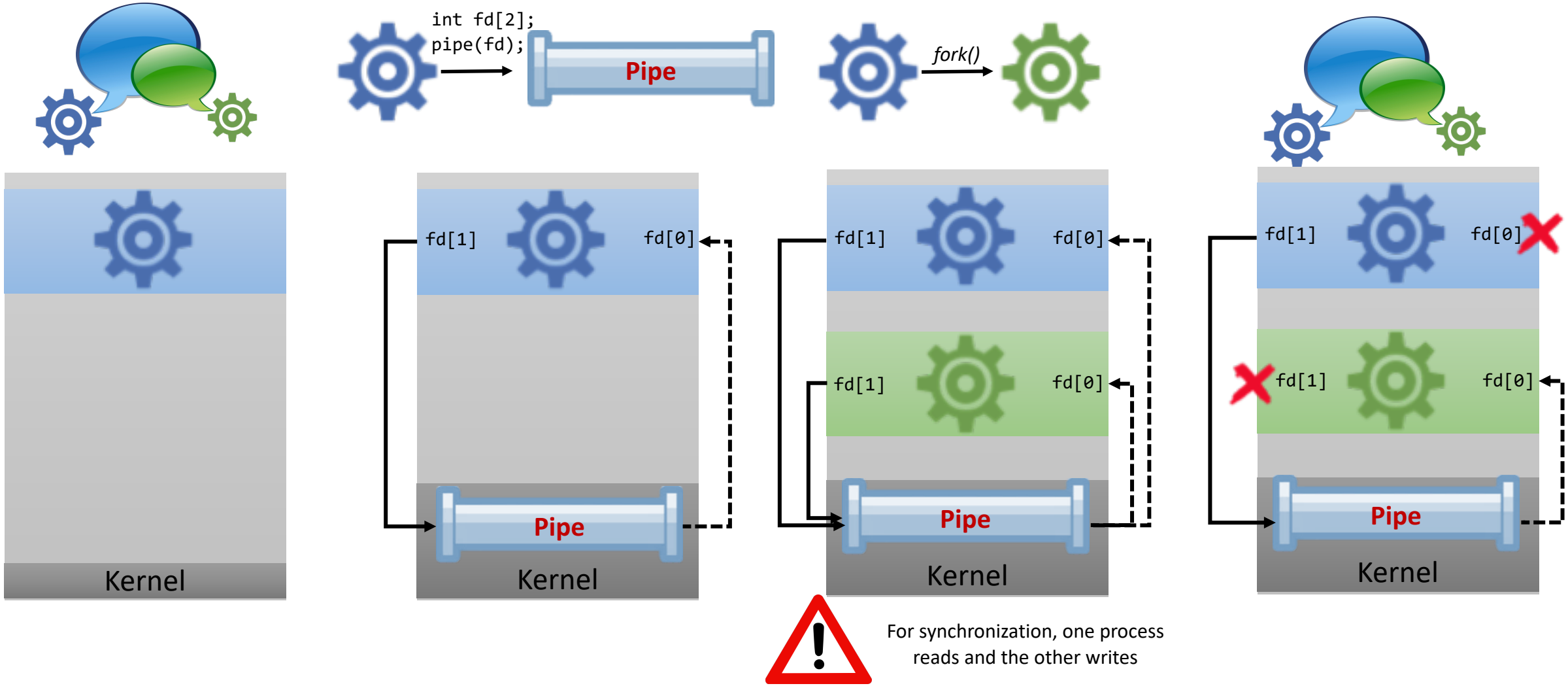
"Full Duplex"

Pipes have a limited capacity

Pipes are *byte stream*

No concept of a message, the writing/reading process write/reads block of data of any size

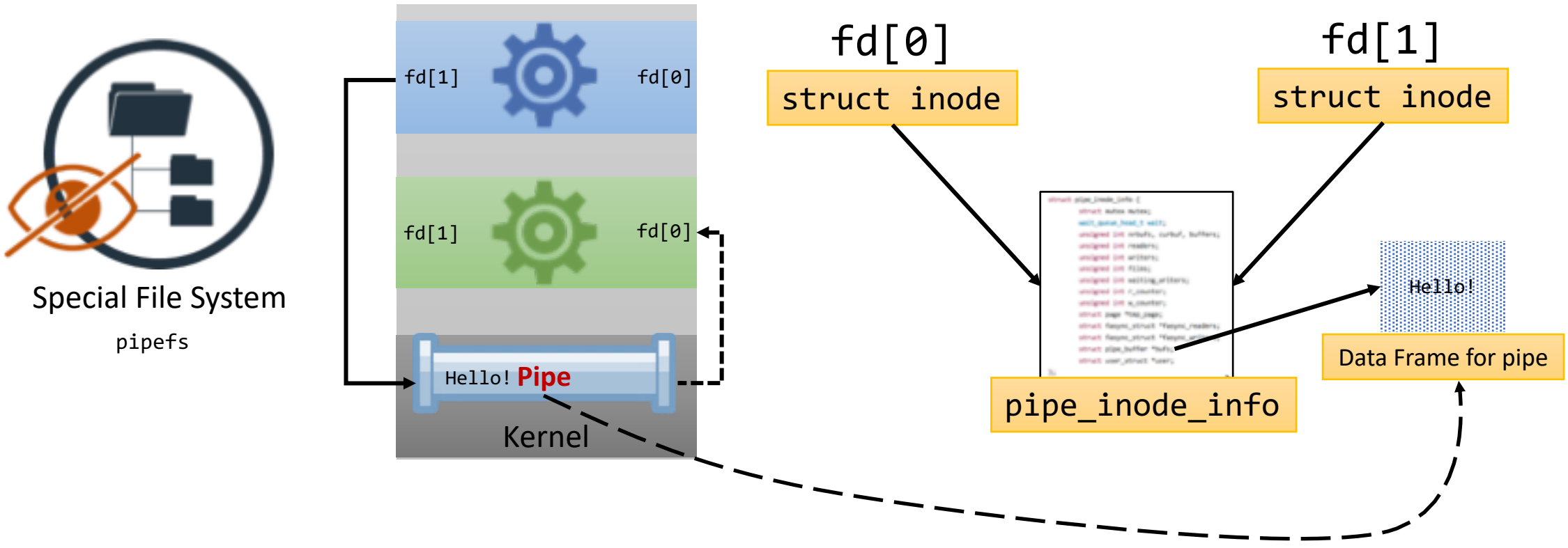
Pipes



For synchronization, one process reads and the other writes

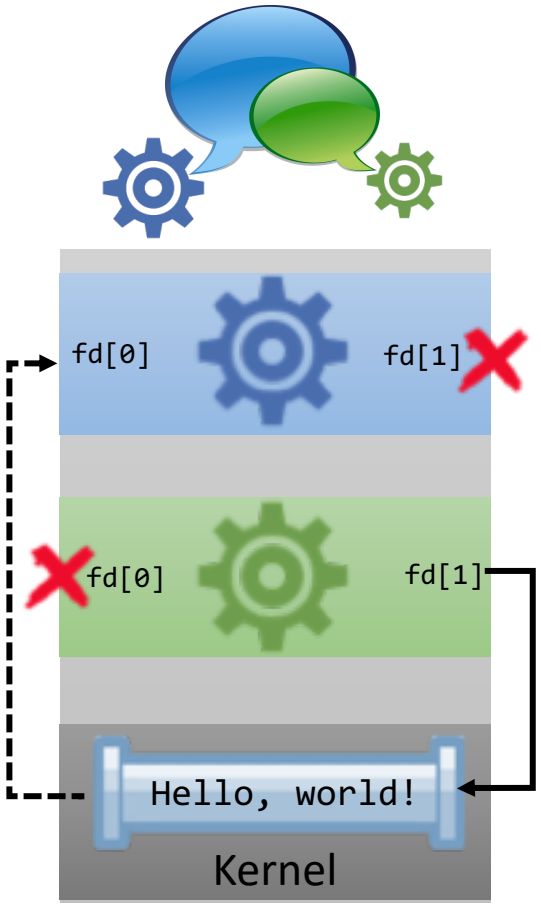
Pipes

*Ordinary pipes is organized using **special file system** "not visible to user", and each pipe is pointed to by a* `pipe_inode_info` *data structure*



https://github.com/torvalds/linux/blob/master/include/linux/fs.h#L553
https://github.com/torvalds/linux/blob/master/fs/pipe.c
https://books.google.com/books?id=LhQ7BAAAQBAJ&printsec=frontcover#v=onepage&q&f=true

Pipes



Child

Parent

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

int main(void){
    int fd[2], nbytes;
    pid_t childpid;
    char string[] = "Hello, world!\n";
    char readbuffer[80];

    /* create pipe */
    pipe(fd);

    if((childpid = fork()) == -1){
        /* Error forking a new child */
        perror("fork");
        exit(1);
    }

    if(childpid == 0){
        /* Child process closes up input side of pipe */
        close(fd[0]);

        /* Send "string" through the output side of pipe */
        write(fd[1], string, (strlen(string)+1));
        exit(0);
    }else{
        /* Parent process closes up output side of pipe */
        close(fd[1]);

        /* Read in a string from the pipe */
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
    }
    return(0);
}
```

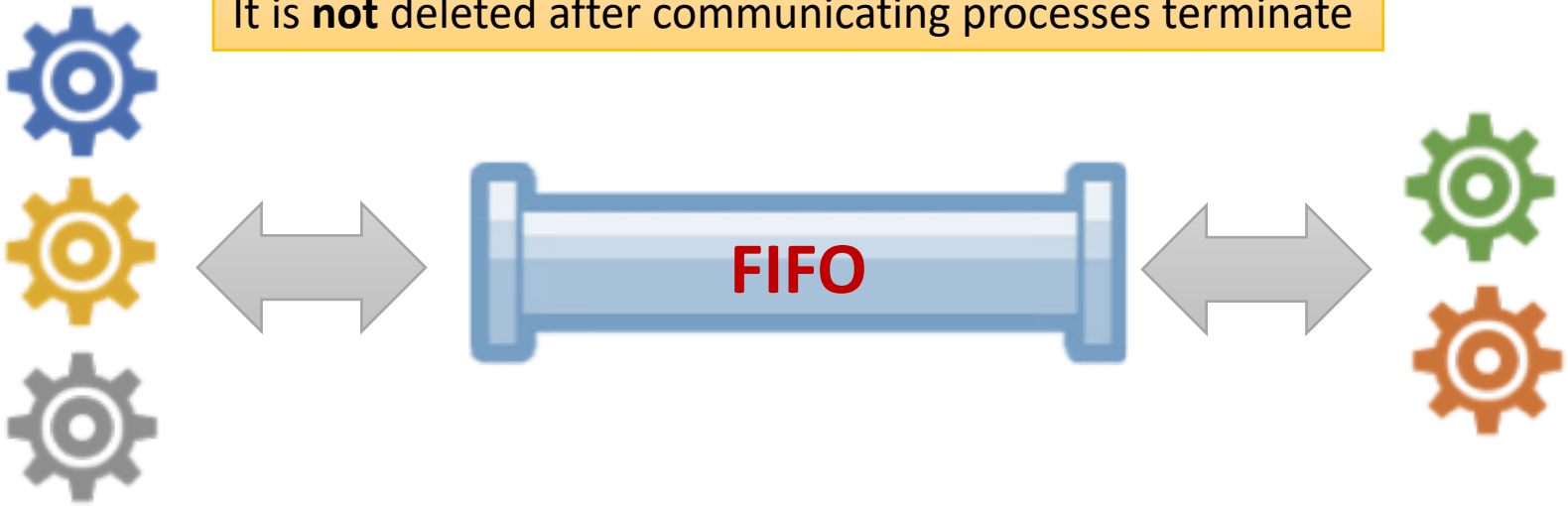
https://www.softprayog.in/programming/interprocess-communication-using-posix-message-queues-in-linux

Named Pipes “FIFO”

*A **named pipe** acts as a **bidirectional** conduit allowing **processes** to communicate*

No parent-child relationship required

It is **not** deleted after communicating processes terminate



http://en.wikipedia.org/wiki/Named_pipe

http://stackoverflow.com/questions/2784500/how-to-send-a-simple-string-between-two-programs-using-pipes

つづく