

Inter-process Communication

Didem Unat

Lecture 5

COMP304 - Operating Systems (OS)

Outline

- Last Lecture: Process Management
 - Process State
 - Context Switch
 - Process Creation and Termination
- Today: Inter-Process Communication (IPC)
 - Cooperating Processes
 - Direct Communication
 - Indirect Communication
 - IPC on Unix, Mac and Windows
 - Pipes

Quiz Question

- Each process has its own process control block
 - True or False?
- From waiting state, a process can only enter into _____.
 - A) running state
 - B) ready state
 - C) new state
 - D) terminated state

Question

```
#include <stdio.h>
#include <unistd.h>

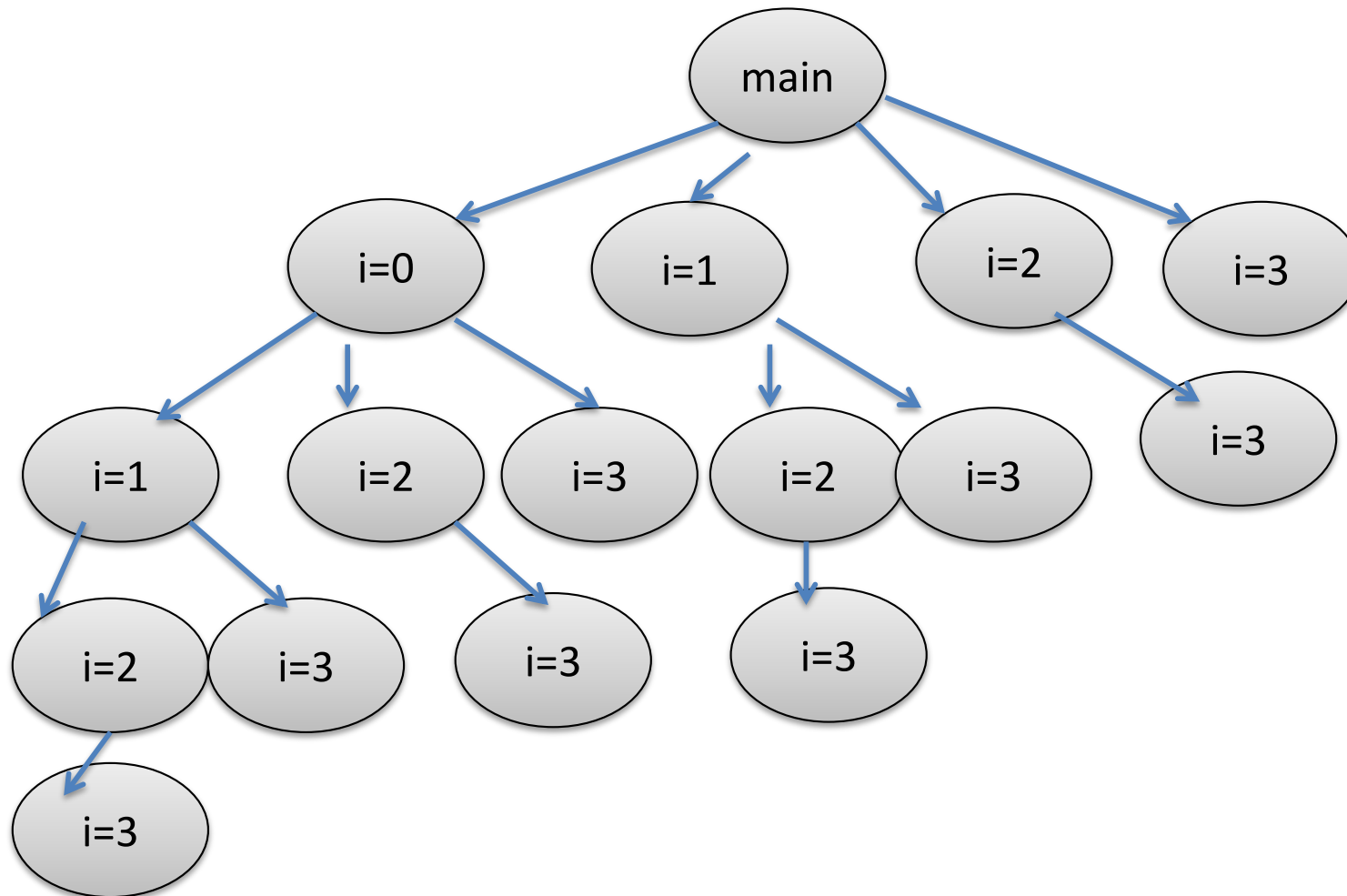
int main()
{
    int i;
    for(i=0; i < 4 ; i++)
        fork();

    printf("PID %d\n", getpid());
    return 0;
}
```

Including the initial parent process,
How many processes are created?

Draw a process tree starting from
the initial parent process as the root!

Process Tree for Question

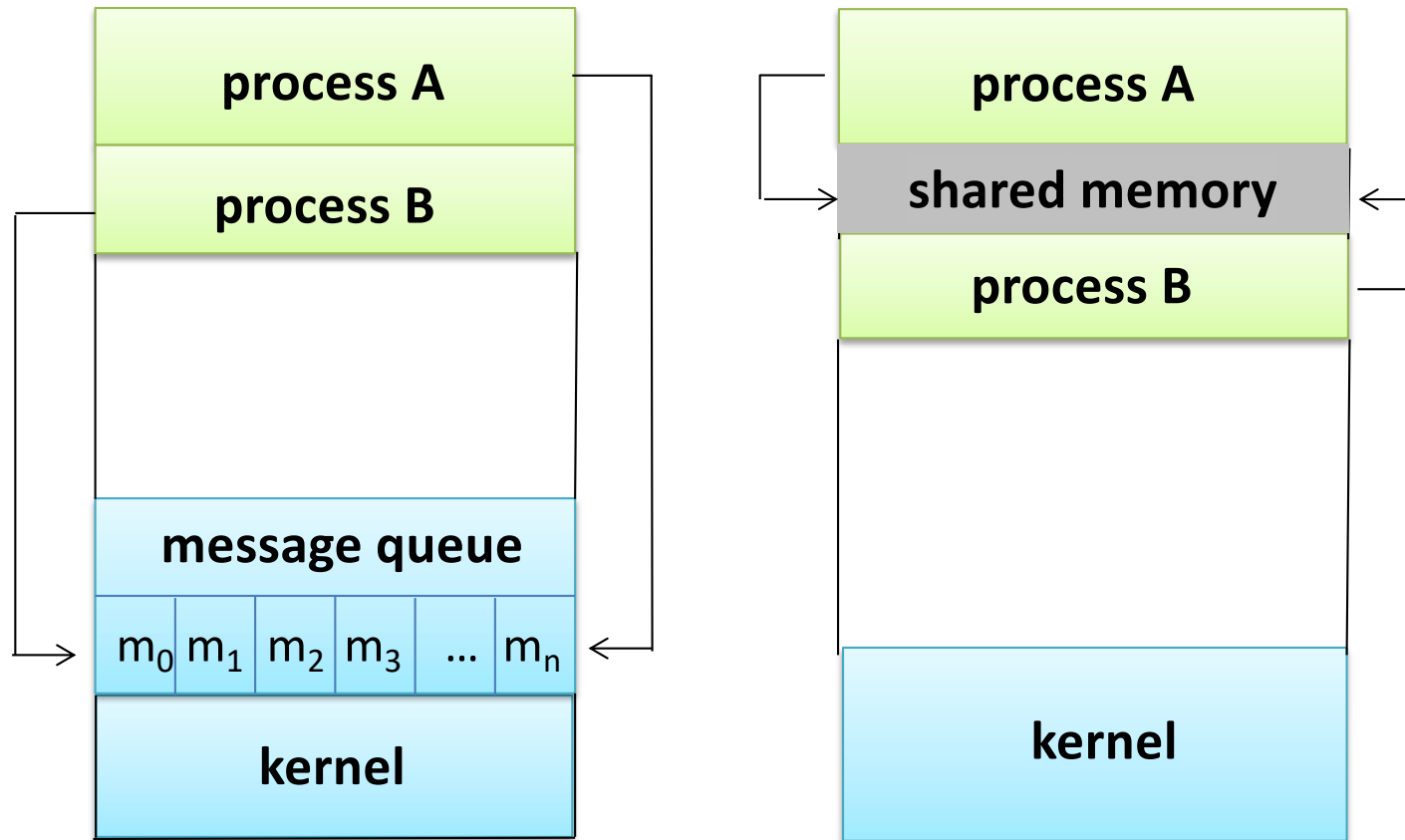


Inter-process Communication (IPC)

- *An independent* process cannot affect or be affected by the execution of another process.
- *Cooperating* processes can affect or be affected by the execution of other processes
- Cooperating processes need methods for **inter-process communication (IPC)**
- There are two models of IPC
 - **Shared memory**
 - **Message passing**

Two Models of Communication

Message Passing vs Shared Memory



- Message passing requires the message of A to be copied to a buffer and copied to process B's memory – thus it is slightly slower but safer

Why support IPC?

There are several reasons for supporting IPC

- Sharing information
 - for example, web servers use IPC to share web documents and media with users through a web browser
- Distributing work across systems
 - for example, Wikipedia uses multiple servers that communicate with one another using IPC to process user requests
- Separating privilege
 - for example, network systems are separated into layers based on privileges to minimize the risk of attacks. These layers communicate with one another using encrypted IPC
- Processes within the same computer or across computers use similar techniques for communication

Message Passing

- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
 - **send**(*message*) – message size fixed or variable
 - **receive**(*message*)
- If P and Q wish to communicate, they need to:
 - establish a **communication link** between them
 - exchange messages via send/receive
- Implementation of communication link
 - Direct or indirect,
 - Synchronous or asynchronous,
 - Automatic or explicit buffering

Direct Communication

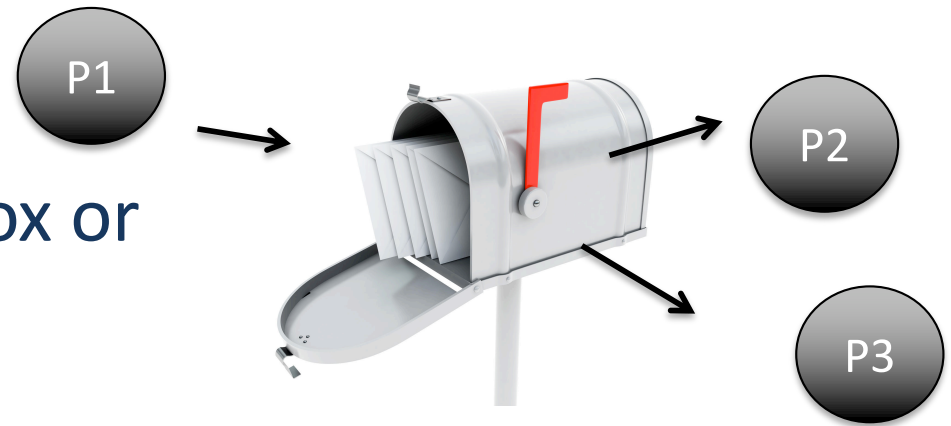
- Processes must name each other explicitly:
 - **send** (*P*, *message*) – send a message to process P
 - **receive**(*Q*, *message*) – receive a message from process Q
- Properties of communication link
 - Links are established automatically
 - A link is associated with exactly one pair of communicating 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



- A process may own a mailbox or
- OS provides operations to
 - create a new mailbox
 - 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*

Blocking or Nonblocking ?

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
 - **Blocking send** has the sender block until the message is received
 - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
 - **Non-blocking send** has the sender send the message and continue
 - **Non-blocking receive** has the receiver receive a valid message or null



Blocking or
non-blocking?



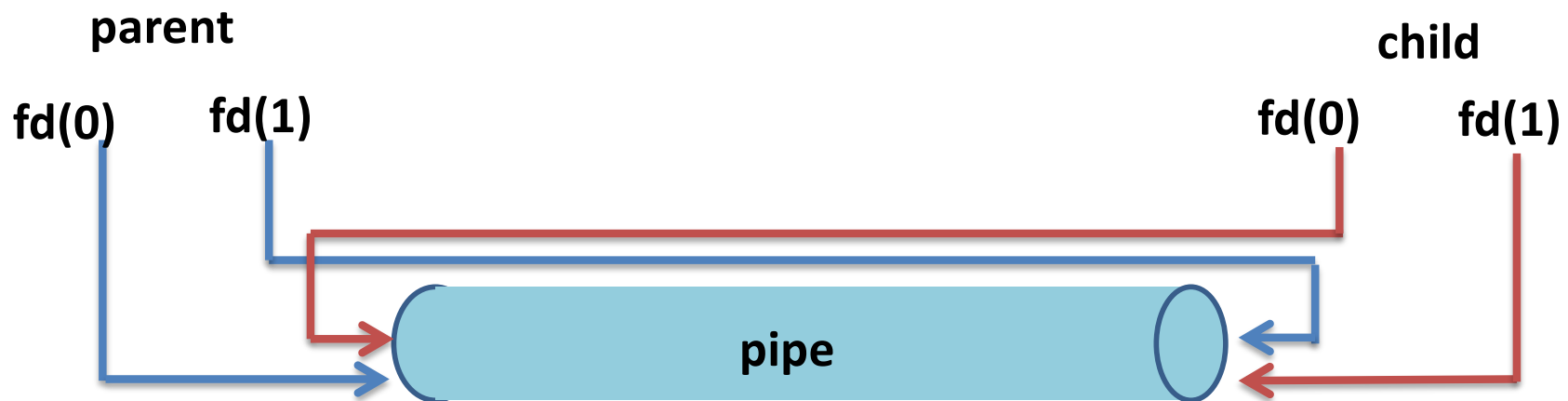
Blocking or
non-blocking?

Pipes

- Acts as a conduit allowing two processes to communicate
 - Ordinary Pipes
 - Named Pipes
- **Issues**
 - Is communication unidirectional or bidirectional?
 - Must there exist a relationship (i.e. *parent-child*) between the communicating processes?
 - Can the pipes be used over a network?

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore **unidirectional**
- Require parent-child relationship between communicating processes



`int fd[]` file descriptors: `fd[0]` is the read-end of the pipe, and `fd[1]` is the write-end.

An Example of Ordinary Pipes

- Powerful command for I/O redirection
- Connects multiple commands together
- With pipes, the standard output of one command is fed into the standard input of another.

```
bash$> ls -l | less
```

```
bash$> history | less
```


Named Pipes

- Named Pipes are more powerful than ordinary pipes.
- Communication is bidirectional.
- No parent-child relationship is necessary between the communicating processes.
- Several processes can use the named pipe for communication.
- Provided on both UNIX and Windows systems.
- An example here:
 - <https://www.geeksforgeeks.org/named-pipe-fifo-example-c-program/>

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, SIZE);
```

- Memory-mapped the file

```
ptr = mmap(start, length, PROT_WRITE, MAP_SHARED, shm_fd, offset);
```

- Now the process could write to the shared memory

```
sprintf(ptr, "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 object */
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;
```

```
}
```

Need to compile with `-lrt` flag

Create a shared memory segment

Memory-mapped file

Writing into the shared memory object

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 object */
    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;
}
```

Create a shared memory segment for readonly

Memory-mapped file for reading

Read from the shared memory object

Examples of IPC Systems - Mach

- Mach communication is message based
 - Even system calls are messages
 - Each task gets two mailboxes at creation- Kernel and Notify
 - Only three system calls needed for message transfer
msg_send(), msg_receive(), msg_rpc()
 - Mailboxes needed for communication, created via
port_allocate()
 - Send and receive are flexible, for example four options if mailbox full:
 - Wait indefinitely
 - Wait at most n milliseconds
 - Return immediately
 - Temporarily cache a message

Examples of IPC Systems – Windows

- Message-passing centric via **advanced local procedure call (LPC)** facility
 - Only works between processes on the same system
 - Uses ports (like mailboxes) to establish and maintain communication channels
 - Communication works as follows:
 - The client opens a handle to the subsystem's **connection port** object.
 - The client sends a connection request.
 - The server creates two private **communication ports** and returns the handle to one of them to the client.
 - The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Reading

- Read Chapter 3.4-3.7
 - Excluding client-server communication
- Acknowledgments
 - These slides are adapted from
 - Öznur Özkasap (Koç University)
 - Operating System and Concepts (9th edition) Wiley