

Richard Litt



Scan QR code to add me as a friend.

Week 11 Lecture 2

XMUT-NWEN 241 - 2026 T2

Systems Programming

Richard Littauer

School of Engineering and Computer Science

Victoria University of Wellington

richard.littauer@vuw.ac.nz

Admin

- Assignment 3
  - Due date is 22 November.
- For the exercises, since we use an automated marking process, please make sure to:
  - Submit C files. In particular no Microsoft Word documents and executable files.
  - No special characters. It normally breaks the script!

Review

Q: The user types 42 and presses return. scanf("%d", &n) reads the number. What's still waiting in the input?

Q: char line[5]; fgets(line, sizeof(line), stdin); The user types hello and presses return. What's in line?

Review

Q: You run ./prog one two three. What is argc?

Q: Same command. What is argv[0]?

Q: Same command. Which argument is argv[argc - 1]?

Q: You run ./prog hello. What is argv[1][1]?

Review

Q: You run ./prog "hello world". What is argc?

Q: You run ./prog 5. Is argv[1] the number 5?

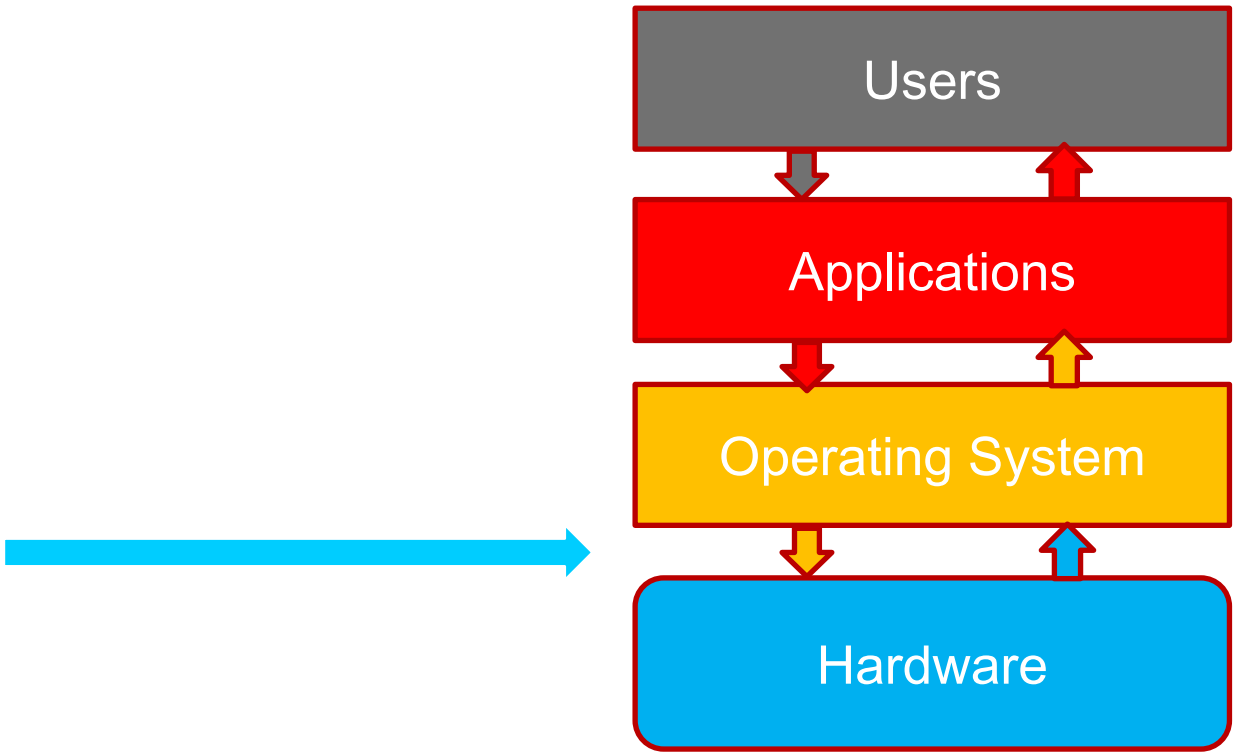
Q: You run ./prog with no arguments. What is argv[1]?

Q: In ls -l /tmp, what are argc and argv[2] for ls?

Content

- System calls
- Interprocess communication

System calls - What and Why?



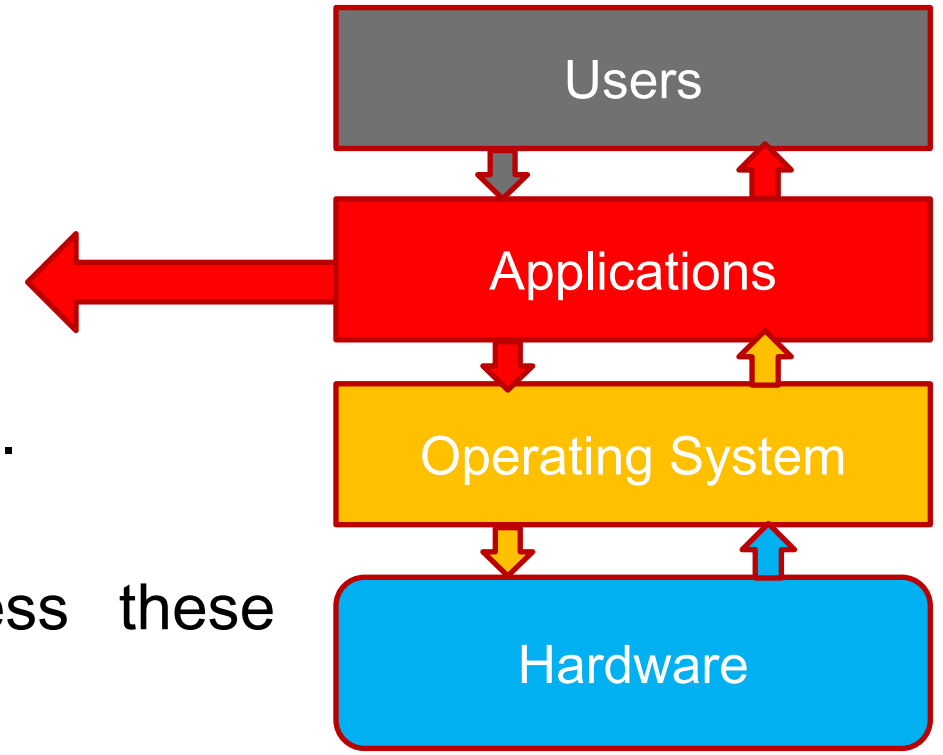
Conceptual View of a Computer System

System calls - What and Why?

- Typically needs access to system resources.

- System resources can be:
  a) physical – e.g. input devices, screen displays.
     OR
  b) Virtual – e.g. files, network connections, threads.

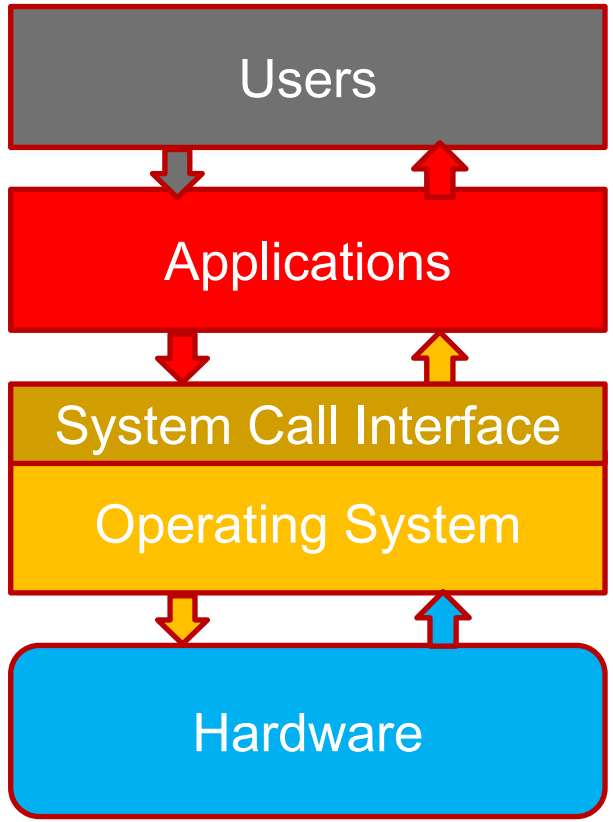
- Applications need O.S. to enable them access these resources.



Conceptual View of a Computer System

System calls - What and Why?

- Operating Systems do not allow application software to access system resources directly due to security and reliability issues.
- A program can request the services of system resources from O.S through system calls.
- System calls are function invocations made from the application into the OS in order to request some service or resource from the operating system.
- Application developers often do not have direct access to system calls but can access them through a system call API, which in turn invokes the system call.



An example of a system call usage

- Consider the following example:

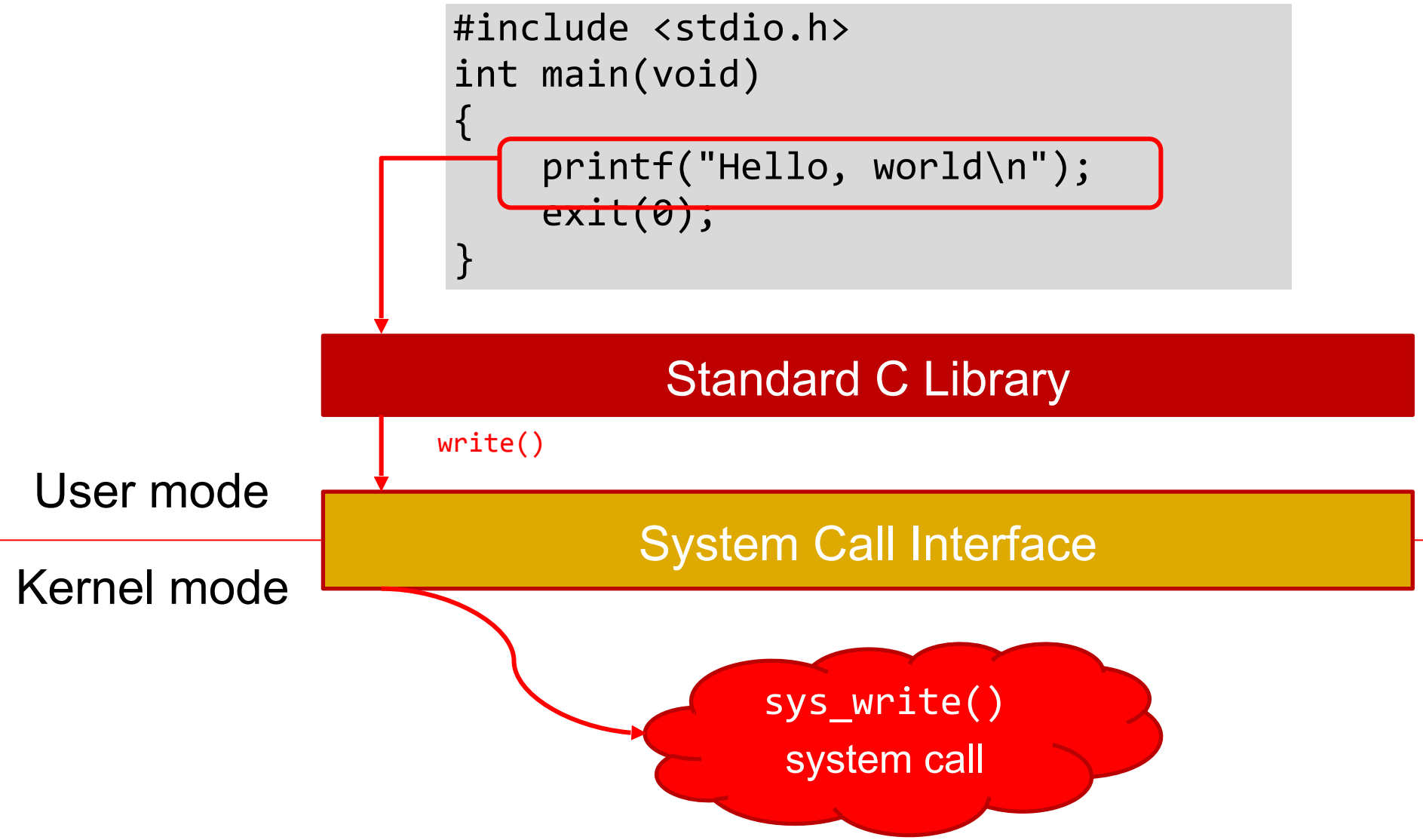
```
#include<stdio.h>

int main()
{
   printf("Hello World");
   return 0;
}
```



C Library function `printf` **"asks" the operating system to print for the calling program** by using the system call API routines

System call invocation –Example



Library functions and system calls

C library function	System call it uses
`printf`	`write`
`fopen`	`open`
`fread`	`read`
`fclose`	`close`

- The library adds the extras (like %d formatting), then makes the system call.
- In a terminal: `man 3 printf` is the library manual, `man 2 write` is the system call manual.

Demo: calling `write()`

```
#include <unistd.h>

int main(void)
{
    write(1, "Hello, world\n", 13);
    return 0;
}
```

Why it matters

- `printf` is a library function. `write` is the system call underneath it.
- 1 means the screen. 13 is the number of bytes to send.
- Printing, files, the network: they all end in a system call.

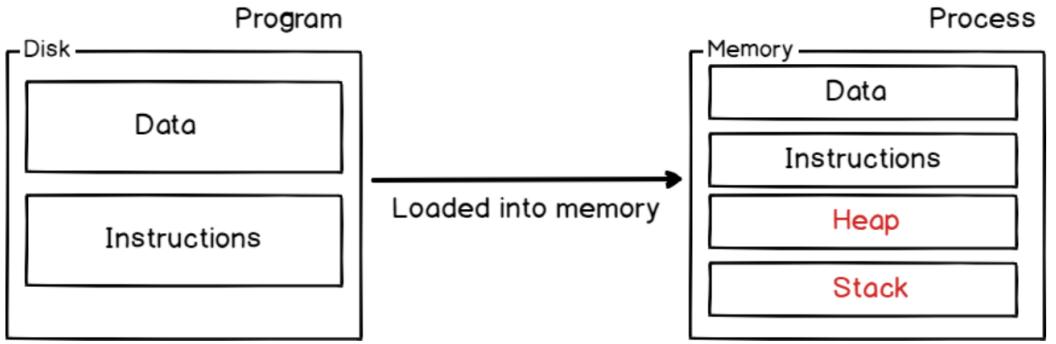
Output

```
$ ./prog
Hello, world
```

Interprocess Communication

What is a process?

- Program and process are related terms.



Program is a set of instructions to carry out a specified task	Process is a program in execution

Passive entity	Active entity
Program is a stored and does not require any other resources.	Process requires system resources such as CPU, memory, I/O etc.
Life span - longer	Life span – limited
Each time a program is run a new process is created.	

Demo: one program, many processes

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

int main(void)
{
    printf("I am process %d\n", getpid());
    return 0;
}
```

Demo: one program, many processes

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

int main(void)
{
    printf("I am process %d\n", getpid());
    return 0;
}
```

Why it matters

- The program on disk is the same both times.
- Each run is a new process, with its own process ID (PID).
- Type `ps` in a terminal to list running processes.

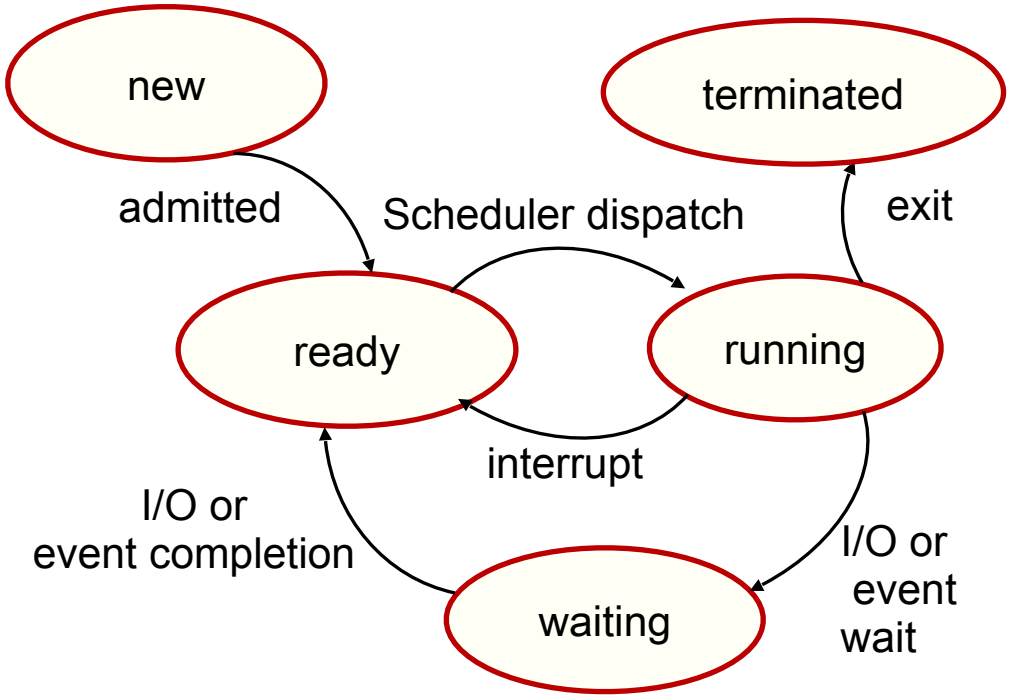
Output

```
$ ./prog
I am process 84017
$ ./prog
I am process 84040
```

Process lifecycle

As a process executes, it changes state

- new: The process is being created
- ready: The process is waiting to be assigned to a processor
- running: Instructions are being executed
- waiting: The process is waiting for some event to occur
- terminated: The process has finished execution



Other process states

- **Stopped**: the process was paused by a signal such as `SIGSTOP` or Ctrl-Z and won't run until it's continued. `ps` shows this as `T`.
- **Zombie (terminated, not reaped)**: the process has exited, but its parent hasn't collected the exit status yet. It holds a process table entry and nothing else. `ps` shows this as `Z`. You can't kill a zombie. The parent has to reap it, or the parent has to exit.
- **Uninterruptible sleep**: the process is waiting on I/O, usually disk or a network filesystem, and ignores signals until the wait finishes. `ps` shows this as `U` on macOS and `D` on Linux. It's why a stuck process sometimes can't be killed.
- **Idle**: the process is sleeping for a long time, which macOS reports as `I`. An idle shell at a prompt is usually `S` or `I`.
- **Swapped or suspended**: the memory pages have been moved to disk. This is a memory-management state, not an execution state, so it can overlap with the others.

Process management system calls

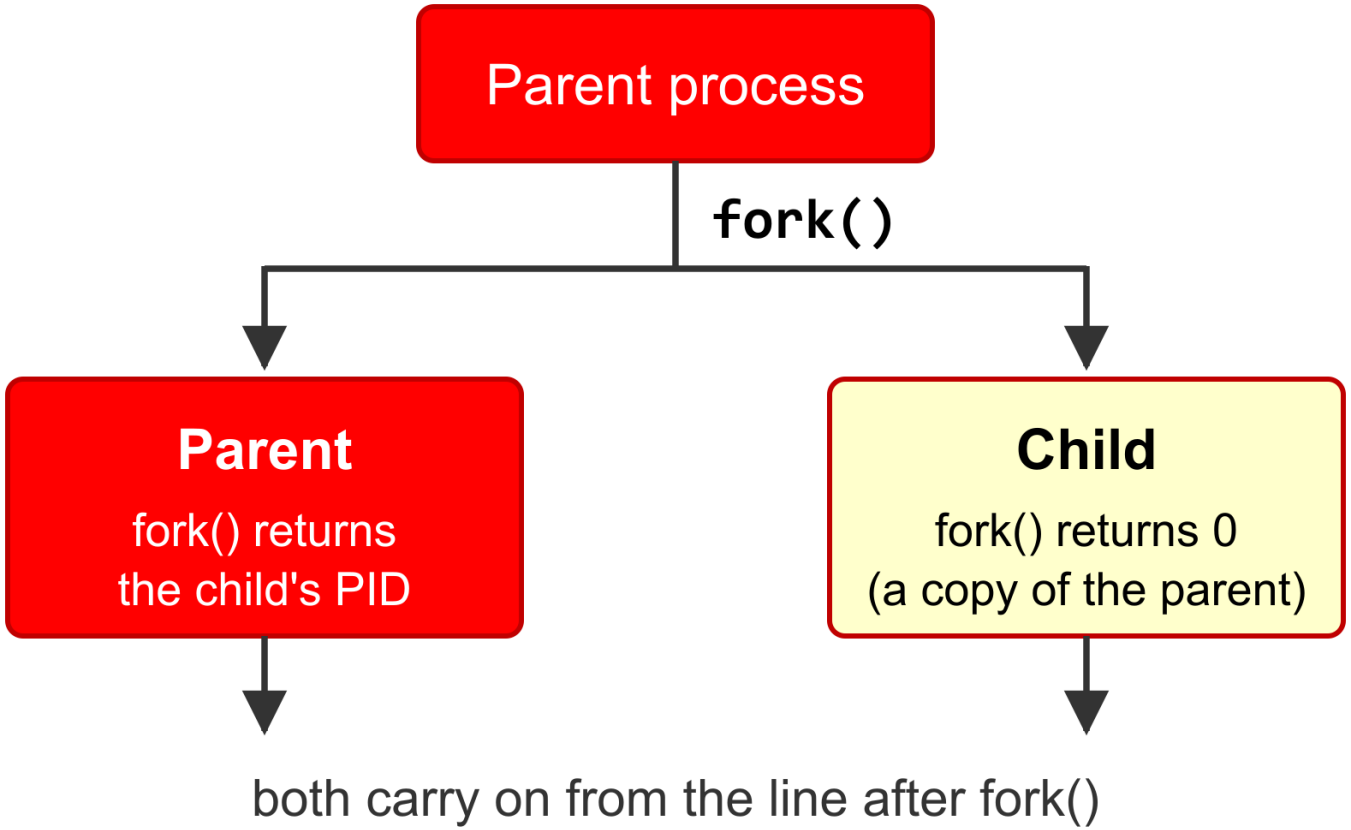
The following system calls are used for basic process management.

- `fork()`
- `exec()`

} Defined in `unistd.h`

- `wait()` } Defined in `sys/wait.h`
- `exit()` } Defined in `stdlib.h`

`fork()`: one process becomes two



- `fork()` copies the running process.
- Now there are two: the parent and the child.
- `fork` returns twice, once in each. The return value tells each one which it is.

Demo: fork

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

int main(void)
{
    printf("before fork: process %d\n", getpid());
    fork();
    printf("after fork: process %d\n", getpid());
    return 0;
}
```

Demo: fork

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

int main(void)
{
    printf("before fork: process %d\n", getpid());
    fork();
    printf("after fork: process %d\n", getpid());
    return 0;
}
```

Why it matters

- "before" prints once. "after" prints twice: two processes ran that line.
- Two different PIDs: the parent and the child.
- The order can change from run to run.

Output

```
$ ./prog
before fork: process 84157
after fork: process 84157
after fork: process 84180
```

Demo: wait and exit

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

int main(void)
{
    int pid = fork();
    if (pid == 0) {
        printf("child: working\n");
        sleep(2);
        printf("child: done\n");
        exit(0);
    }
    printf("parent: waiting for child %d\n", pid);
    wait(NULL);
    printf("parent: child finished\n");
    return 0;
}
```

Why it matters

- `fork` returns 0 in the child, so only the child runs the `if`.
- `exit(0)` ends the child: the terminated state.
- `wait(NULL)` blocks the parent until the child has finished.

Output

```
$ ./prog
parent: waiting for child 84187
child: working
child: done
parent: child finished
```

Demo: exec

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

int main(void)
{
    printf("about to run echo\n");
    execlp("echo", "echo", "\"hello world\"", NULL);
    printf("this line is never reached\n");
    return 0;
}
```

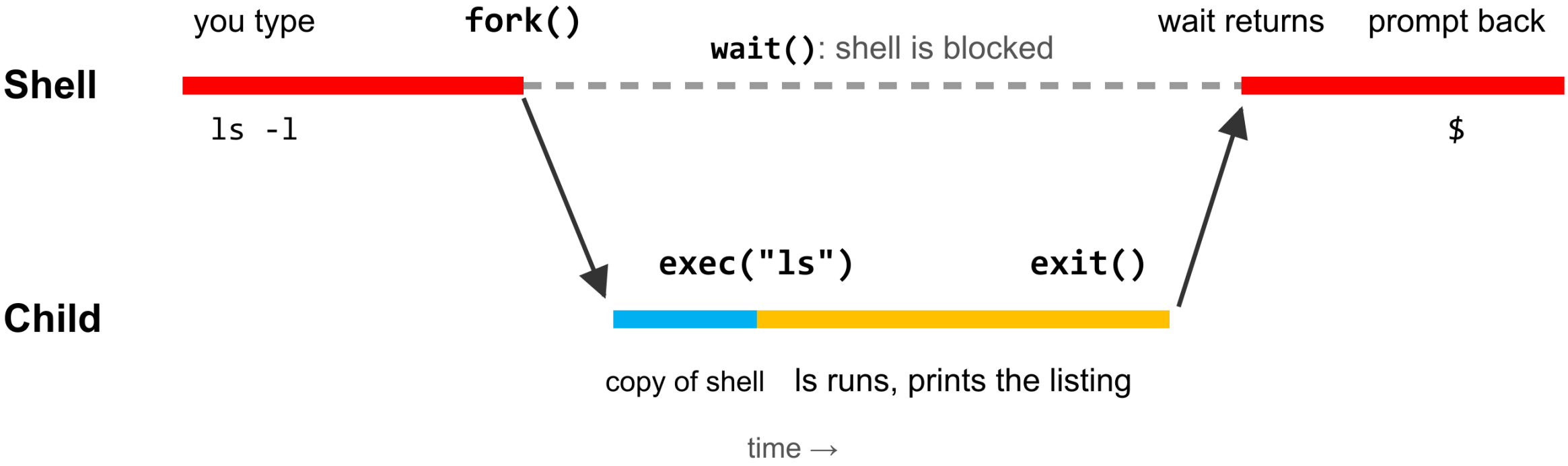
Why it matters

- `exec` replaces the program in this process with `echo`.
- The last `printf` never runs: our program is gone.
- The strings after `"echo"` become `echo`'s `argv`, like last lecture. `NULL` marks the end.

Output

```
$ ./prog
about to run echo
"hello world"
```

How your terminal runs `ls -l`



- `fork` makes the new process. `exec` puts `ls` into it.
- `wait` lets the shell sleep until `ls` has finished. Then you get the prompt back.

Process - Independent Vs Cooperating

- Independent processes: processes that don't interact with other processes
- Cooperating processes: process can affect or be affected by other processes.
- In order to co-operate processes, need to communicate
  - Inter Process Communication

Demo: each process has its own memory

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main(void)
{
    int x = 5;
    if (fork() == 0) {
        x = 99;
        printf("child:  x = %d\n", x);
        exit(0);
    }
    wait(NULL);
    printf("parent: x = %d\n", x);
    return 0;
}
```

Demo: each process has its own memory

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

int main(void)
{
    int x = 5;
    if (fork() == 0) {
        x = 99;
        printf("child:  x = %d\n", x);
        exit(0);
    }
    wait(NULL);
    printf("parent: x = %d\n", x);
    return 0;
}
```

Why it matters

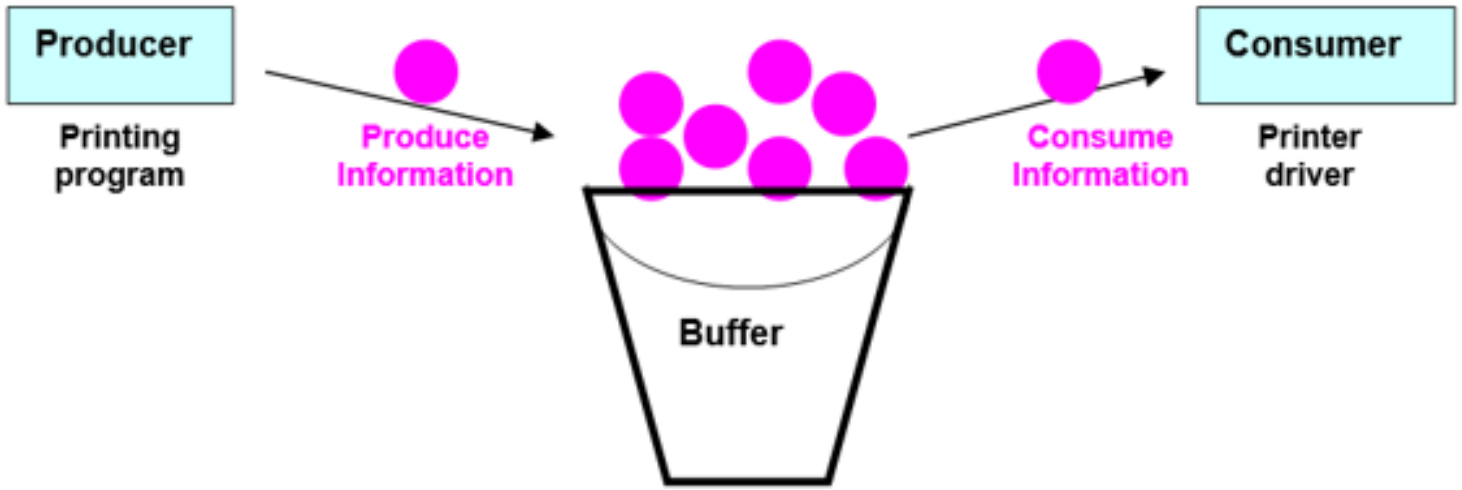
- `fork` copies the memory. It doesn't share it.
- The child changed its own copy of x.
- So processes can't talk through a variable. They need IPC.

Output

```
$ ./prog
child:  x = 99
parent: x = 5
```

Cooperating Processes

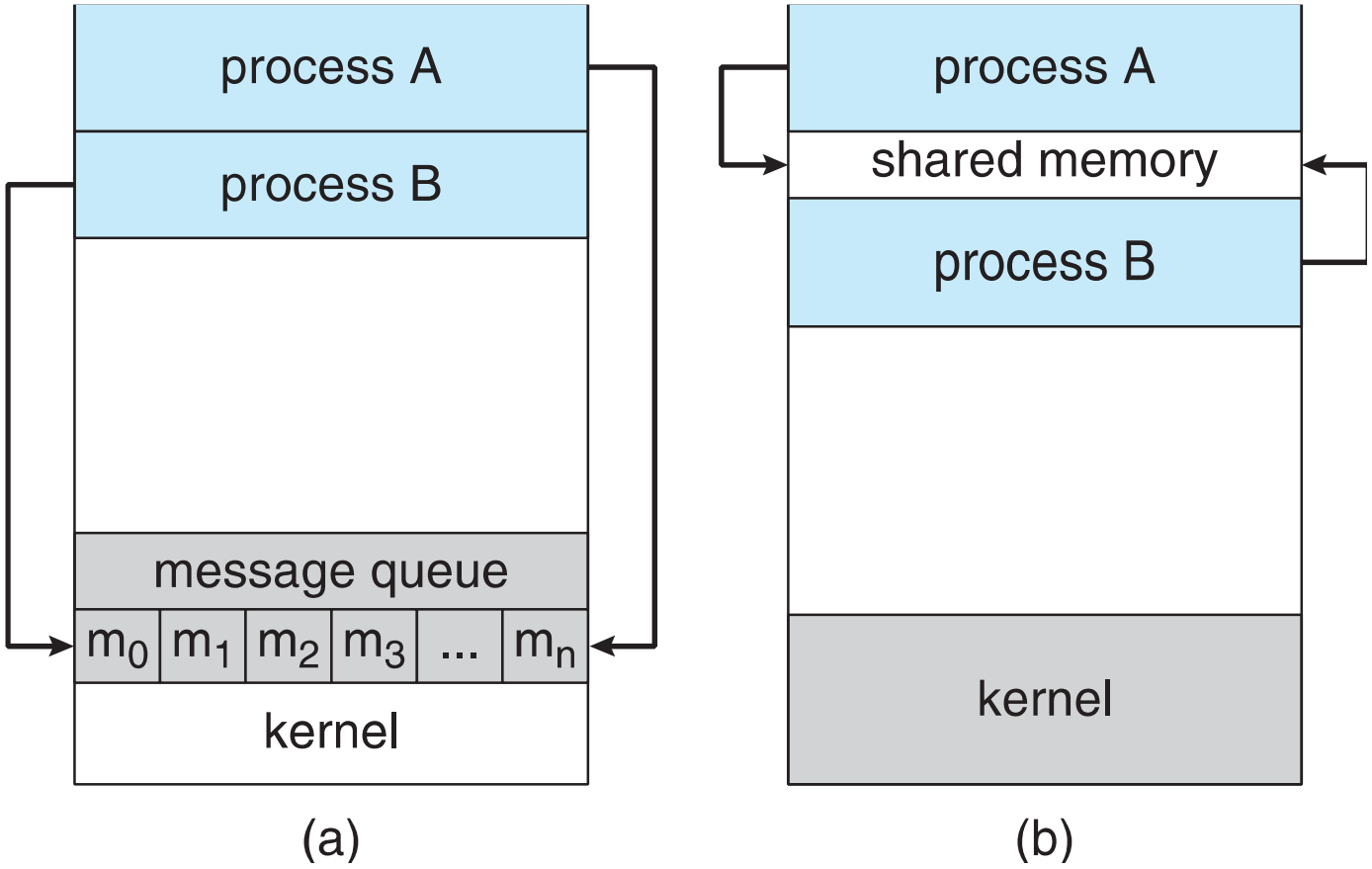
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience



- Cooperating processes can reside on same machine or in different machines (on a network).

Inter-process communication

- Cooperating processes need Inter-process communication (IPC)
- Two primary models of IPC
  - Message passing
  - Shared memory



Inter-process communication

Co-operating Process
Example: A **web browser** and a **web server**.

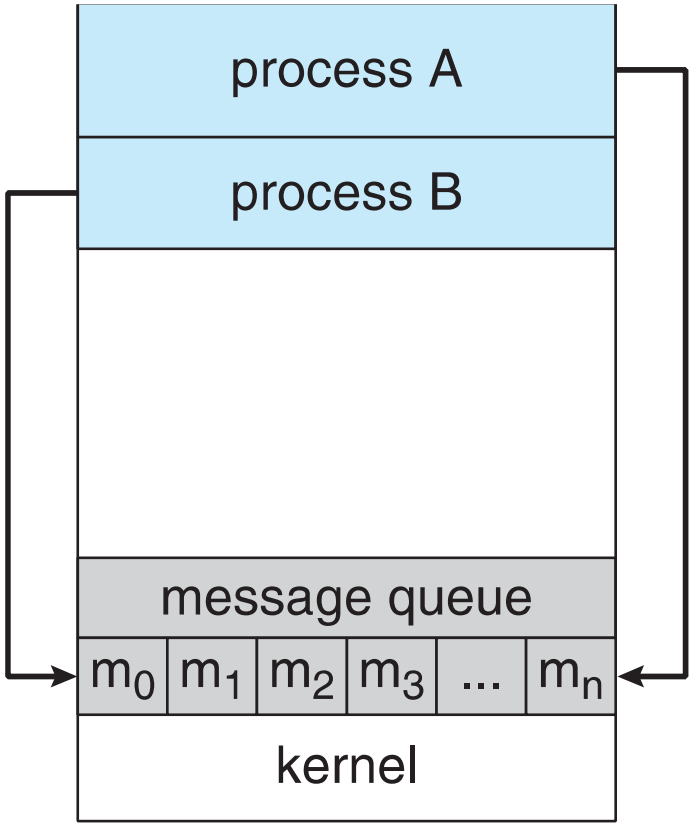
- When you open a website in your browser, your browser (client) sends a request to the server.
- The server processes the request and sends back a response (HTML, CSS, data).
- They are different processes on different machines (or could be on the same), but they must **communicate and share data** to work.
- This communication uses **Inter-Process Communication (IPC)**, often through sockets, pipes, or shared memory.

Independent Process
Example: A text editor like **Notepad** running on your computer.

- It performs its task (editing text) without depending on or interacting with other processes.
- It doesn't share data with, or receive input from, other processes.
- If another application crashes (like a music player), the text editor continues unaffected.

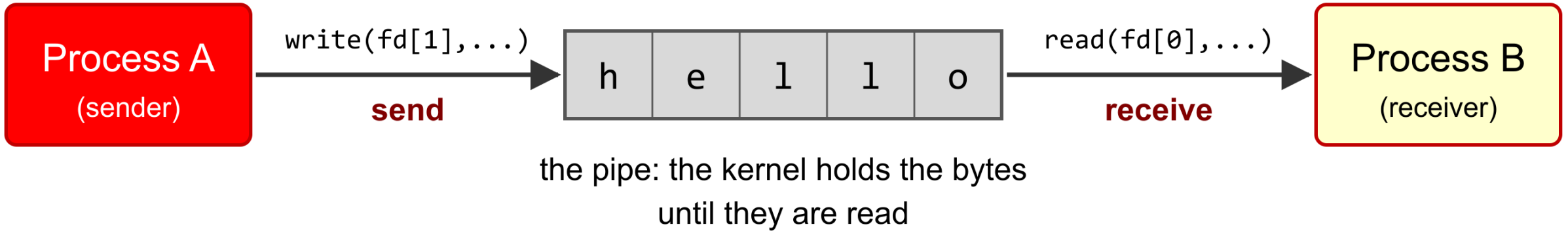
Process Type	Example	Description
Independent Process	Calculator app	Performs calculations without needing other processes.
Co-operating Process	Chat app (e.g., WhatsApp Desktop)	Sends/receives messages with server; requires IPC or network communication.

Message passing



- Processes communicate with each other without resorting to shared variables
- IPC facility provides two primitive operations:
  - `send(message)`
  - `receive(message)`
- If A and B wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

A pipe: a link between two processes



- `pipe(fd)` makes a link. `fd[1]` is the write end and `fd[0]` is the read end.
- Messages wait in the pipe until they are read: the message queue from the last slide.
- Make the pipe before `fork`. That way, both parent and child processes have access.

Demo: a pipe

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

int main(void)
{
    int fd[2];
    char buf[20];
    pipe(fd);
    if (fork() == 0) {
        printf("child: waiting for a message\n");
        int n = read(fd[0], buf, sizeof(buf));
        buf[n] = '\0';
        printf("child: got %s\n", buf);
        exit(0);
    }
    sleep(2);
    write(fd[1], "hello", 5);
    wait(NULL);
    return 0;
}
```

Why it matters

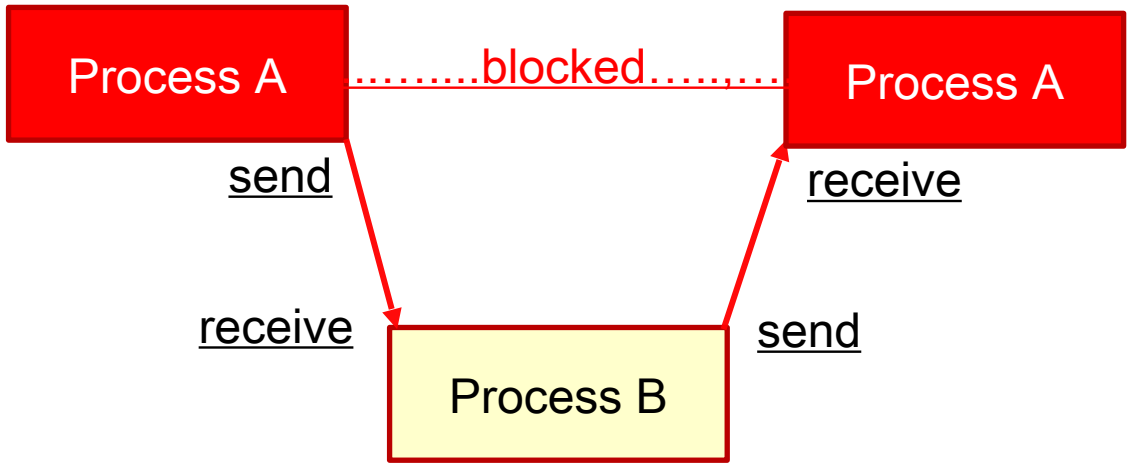
- `write` is send. `read` is receive.
- The child waits in `read` until the message arrives.

Output

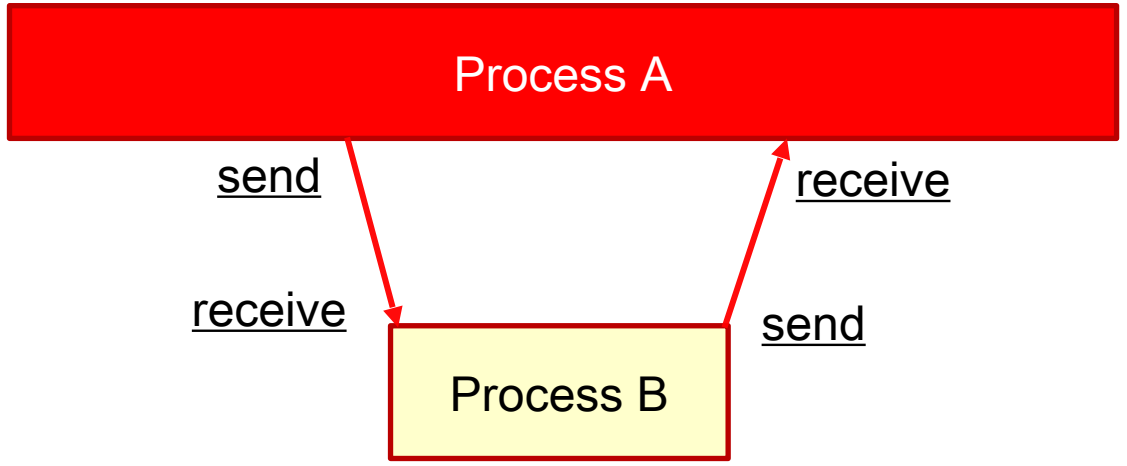
```
$ ./prog
child: waiting for a message
(2 seconds pass)
child: got hello
```

Design options - Synchronization

Blocking (Synchronous)



Non-blocking (Asynchronous)



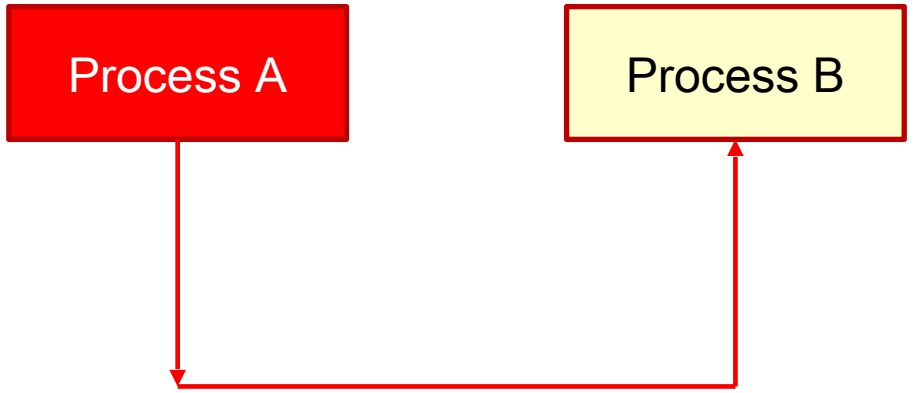
Design options - Synchronization

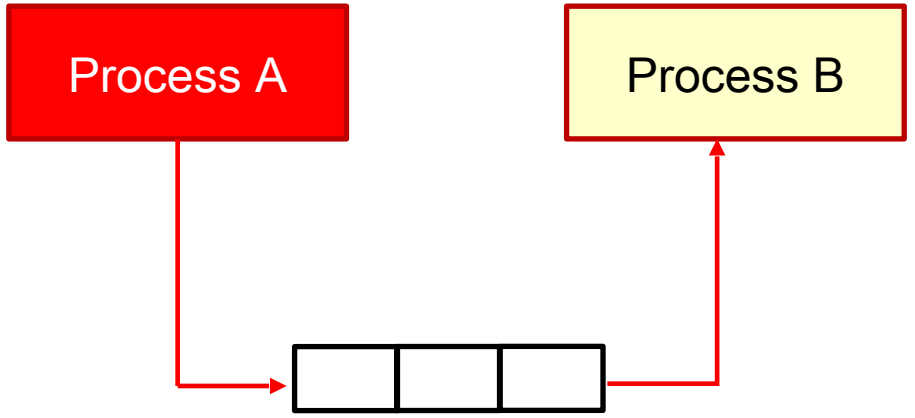
	Blocking	Non - Blocking
Send	Has the sender block until the message is received	Has the sender send the message and continue
Receive	Has the receiver block until a message is available	Has the receiver shown its willing to receive message and continue

Different combinations possible

Design options - Buffering

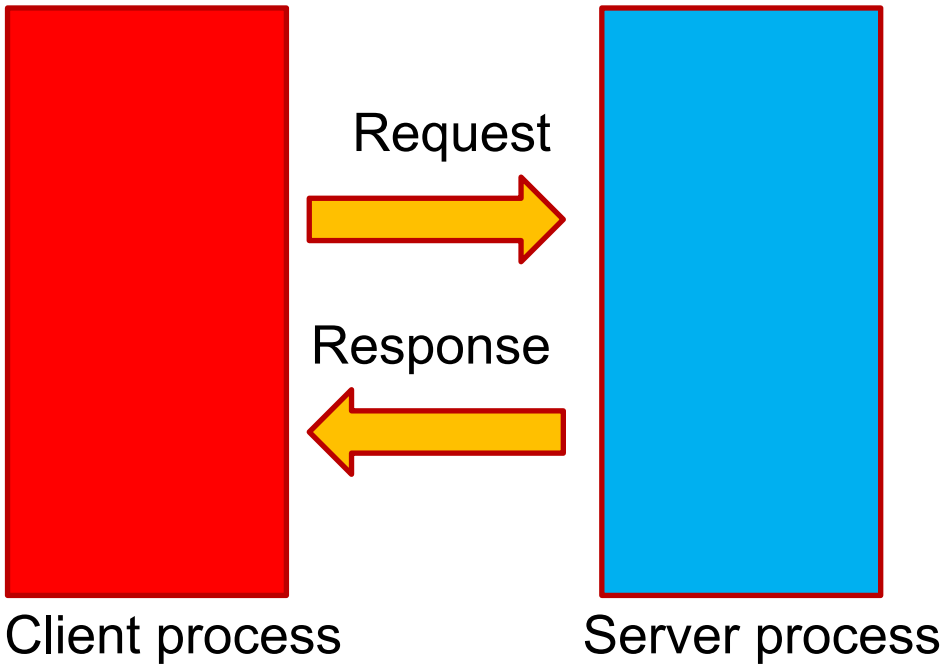
- Queue of messages attached to the link
- Implemented in one of three ways:
  - Zero capacity – 0 messages
    Sender must wait for receiver
  - Bounded capacity – finite length of n messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits





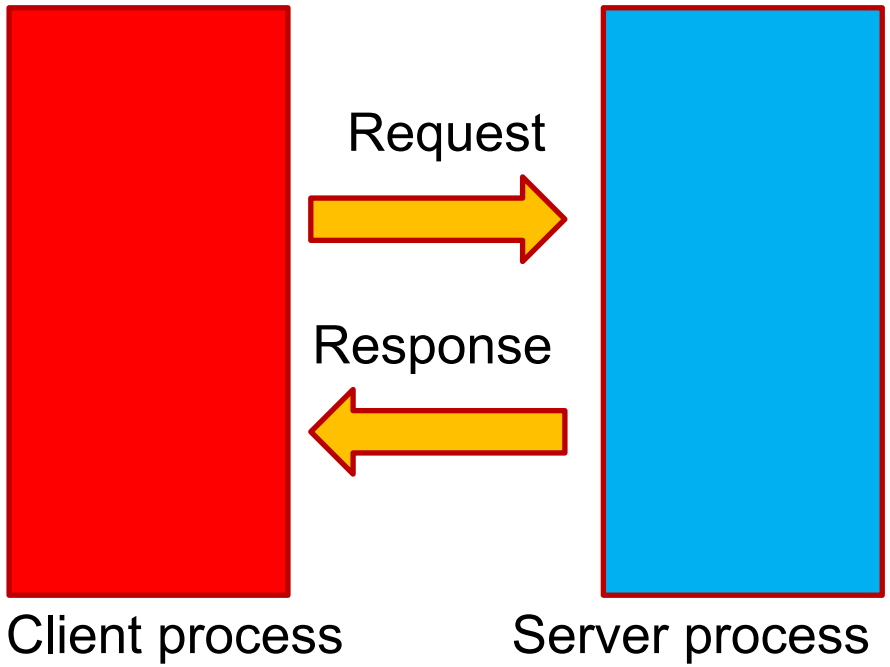
Client-server model

- Most common IPC paradigm
- Based on the producer-consumer model of process cooperation
- Client makes the request for some resource or service to the server process
- Server process handles the request and sends the response (result) back to the client



Client-server model

- Client process needs to know the existence and the address of the server
- However, the Server does not need to know the existence or address of the client prior to the connection
- Once a connection is established, both sides can send and receive information



Demo: client and server in two terminals

Terminal 1: server

```
$ mkfifo /tmp/server
$ cat /tmp/server
(waits for a client...)
hello
```

Terminal 2: client

```
$ echo hello > /tmp/server
```

- `/tmp/server` is a named pipe: a pipe with a name.
- The client has to know the server's name: its address.
- The server doesn't need to know who the client is.
- Next lecture: the address becomes an IP address and a port.

Review question 1-3

```
printf("hi\n");
fork();
printf("bye\n");
```

- What does this print? How many times is each line printed?

```
fork();
fork();
printf("hi\n");
```

- How many times is "hi" printed?

```
printf("start\n");
execlp("ls", "ls", NULL);
printf("end\n");
```

- Which lines are printed?

Review question 4

```
int x = 1;
if (fork() == 0) {
    x = x + 1;
    exit(0);
}
wait(NULL);
printf("%d\n", x);
```

- What does the parent print?

Review question 5

```
if (fork() == 0) {
    printf("child\n");
    exit(0);
}
wait(NULL);
printf("parent\n");
```

- Which line always prints first? Why?

Review question 6

```
int fd[2];
char buf[20];
pipe(fd);
if (fork() == 0) {
    read(fd[0], buf, 20);
    printf("got it\n");
    exit(0);
}
sleep(5);
write(fd[1], "hi", 2);
```

- What is the child doing during the 5 seconds? Which design option is this?

More questions: system calls (1)

- Why can't your program just send bytes to the screen itself?
- Is `printf` a system call?
- What's the difference between user mode and kernel mode?

More questions: system calls (2)

- Name a physical resource and a virtual resource.
- `write(1, "Hi\n", 3)`. What does the 1 mean, and what does the 3 mean?

More questions: processes (1)

- Program or process: which one is just a file on disk?
- You run the same program three times at once. How many processes?
- A process is waiting for you to press a key. Which state is it in?

More questions: processes (2)

- What's the difference between ready and running?
- `fork` returned 0. Are you the parent or the child?
- Three forks in a row, then `printf("hi\n")`. How many times does `hi` print?

More questions: processes (3)

- `exec` worked. What happens to the code after it in your program?
- `exec` fails, for example because the program name is misspelled. What happens?
- What does `wait` do?
- Which call ends a process?

More questions: IPC (1)

- Independent or cooperating: a calculator app?
- Independent or cooperating: a chat app?
- Name two reasons for processes to cooperate.

More questions: IPC (2)

- The two models of IPC?
- In a pipe, which end do you write to, `fd[0]` or `fd[1]`?
- A blocking send with a zero-capacity link. When does the sender carry on?

More questions: IPC (3)

- A bounded buffer is full. What happens to a blocking send?
- Which kind of buffer means the sender never waits?
- After `fork`, the child sets a global variable. Does the parent see the change?

More questions: client-server

- Who needs to know whose address first: the client or the server?
- In the two-terminal demo, which terminal was the server?
- In the two-terminal demo, what stood in for the server's address?

Next Lecture

- TCP Socket Programming