Processes

CISC 3320 — CUNY Brooklyn College
Lecture Notes

Processes

In this chapter, we will:

1. Define what a process is and describe how a process' data is stored in memory.
2. Explain how a process gets to run and the OS elements involved in this decision.
3. Describe what operations could be done with processes, from launching to quitting.
4. List the various, commonly used methods for interprocess communication (IPC).

Processes

Programs are files passively stored on disk and contain **binary code**, which are machine language instructions that can run on the computer. Programs that are large and significant are referred to as **applications**.

Once the binary code of a program is copied to memory in order to start executing, we call such code a **process**.

- That is, a process is an instance of a program that is executing. A program can have several process instances (since you could run the same program several times simultaneously. E.g.: running 2 instances of the Calculator app on Windows simultaneously.)
- The notion of process also includes the memory sections associated with it, the files that it opens, the user who ran the program, and the threads that the process created. More on threads in the next topic.
- In batch systems, processes are referred to as **jobs**, and so both terms can be used interchangeably.

Every process has a unique identifier: its **process ID** (or **pid**.)

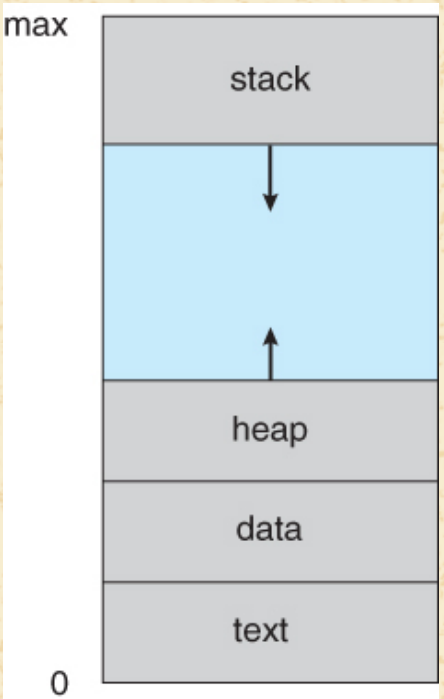
Processes

The memory chunk that a process controls contains several sections:

- The **text** section, in which the binary code of the program (the process's instructions) is stored. This section is fixed in size (depending on how long the binary code is.)
- The **data** section, in which global and static variables are stored. It has a fixed size, too. (***Why so?***)
- The **stack** section, in which local variables, such as those defined inside functions, are stored, and
- The **heap** section, in which variables that were dynamically allocated are stored. E.g.: when you use the `new` operator in java, your variable (or object) is stored on the heap.

Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other since the size of the stack or heap could change over the process's execution. If they should ever meet, then either a stack overflow error will occur, or else a call to `new` or `malloc` (the function the language C uses to allocate memory) will fail due to insufficient memory available.

Processes



A process in memory. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Processes

Any questions?

Getting to Run: Process States

Every process exists in 1 of 5 states at any given time: (Some OS have more states besides these 5.)

1. **New**: After a user issued a command to run a program, the data of the program, including the binary code, is being copied to memory to create the process.
2. **Ready**: All the data needed for the process to run is ready, so the process is in the line to start executing on a CPU.
3. **Running**: The operating system assigned a CPU to the process, so the process's instructions are fetched by the CPU and get executed. Only one process can run on one CPU at any given moment.
4. **Waiting**: The process might have issued an I/O request so it paused its execution until the I/O is done, or it is waiting for some event (e.g., until a process that it created is terminated.) At this stage, the process is removed from the CPU on which it was running until it continues running again, but the process's data is still in memory.
5. **Terminated**: The process is on the way to being removed from memory, either since all its instructions were successfully executed, or some critical error that occurred prevents it from continuing running.

Getting to Run: Process States

A process can transition from one state to another, as the following diagram illustrates:

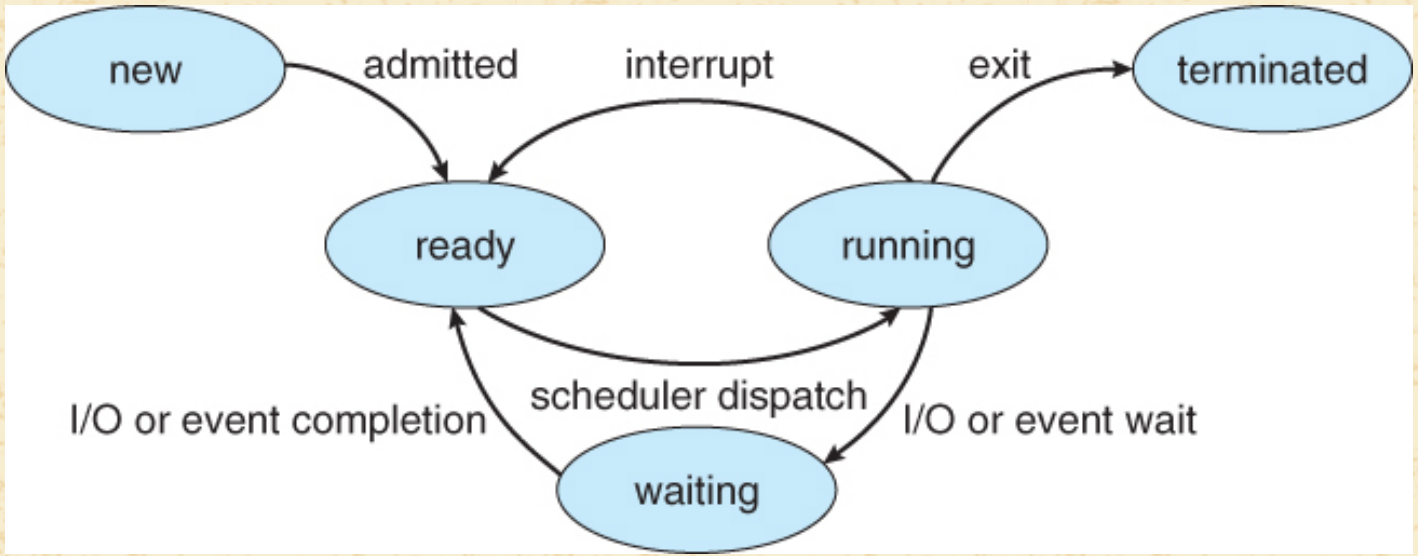


Diagram of process state. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

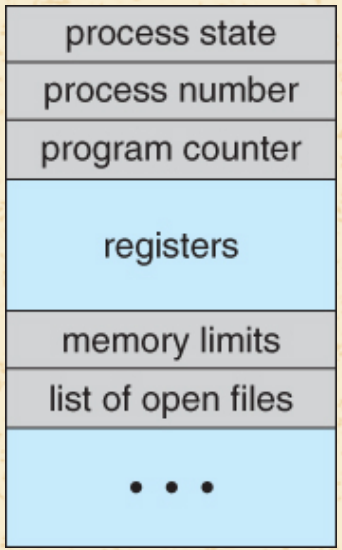
Getting to Run: The Process Control Block

The OS must also keep track of the data that each process needs to be able to execute. The operating system stores these 'metadata' in a kernel data structure called the **Process Control Block** (**PCB**), also called **Task Control Block**. Since the data in the PCB is sensitive, this data structure is stored in the part of memory that only the kernel has permission to access (= kernel space). These metadata include:

1. The **process id** and the current **process state** (one of the 5 we saw earlier.)
2. The **program counter**, which contains the address to the next instruction that should be executed.
3. Copies of the **CPU registers**, which contain the data that was processed when the process was running.
4. **CPU-scheduling information**, such as the process's priority and pointers to scheduling queues.
5. **Memory-management information**, such as page tables or segment tables.
6. **Accounting information**, such as the amount of CPU and real time that the process has used.
7. **I/O Status information** like the I/O devices that the process uses and the list of open files.

Getting to Run: The Process Control Block

The Process Control Block would be of the following structure:



The Process Control Block (PCB). Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Getting to Run: Process Scheduling

Just as the operating system needs to decide the order in which I/O tasks occur on an I/O device, it also needs to decide the order in which processes run on a CPU. Such a decision is called **process scheduling**, and it is made by the **process scheduler** software of the OS. The reason that the OS needs to make this decision is that only one process can run on a single CPU at any given time.

On devices that have multiple CPUs, several programs can run in parallel (each on one CPU.)

The ultimate purposes of performing process scheduling are:

1. To ensure that the CPU is always busy (that some program runs on the CPU at any given moment.)
2. To allow the running processes to provide responses to users within a reasonable time (imagine the insanity of waiting for 2 hours for your Java program to run just because a high-quality computer game is also running.)

Getting to Run: Process Scheduling

The operating system is using several **queues** to store the information about processes:

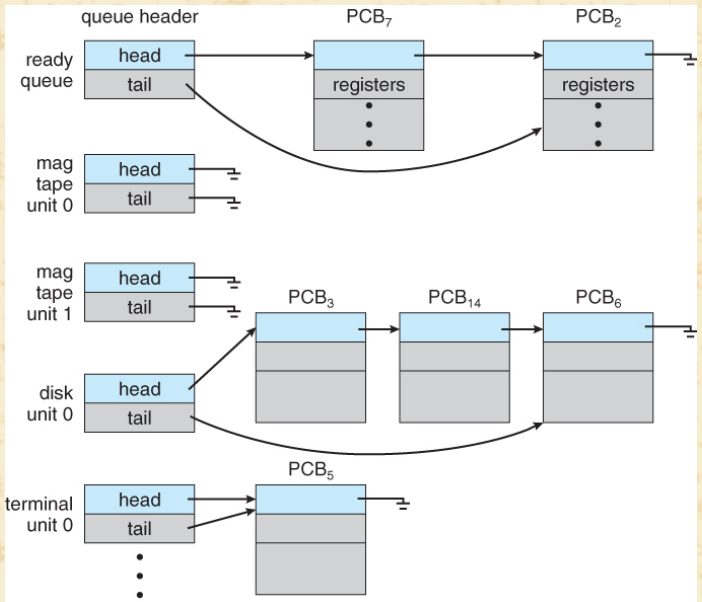
1. All the processes are stored in the **job queue**.
2. Processes that are ready to run (have the 'ready' status) are stored in the **ready queue**.
3. Processes that are in a waiting stage (have the 'waiting' status) are stored in a **wait queue** (there might be several wait queues, each for a different waiting purpose/cause.)
4. Processes waiting for an I/O device are kept in a **device queue**, which is a type of a wait queue. The OS maintains a device queue for each device.

These queues are usually implemented as linked lists, and each node contains the corresponding process's PCB.

Process scheduling can be neatly represented with the help of a **queueing diagram**, which contains the ready queue and several waiting queues.

Getting to Run: Process Scheduling

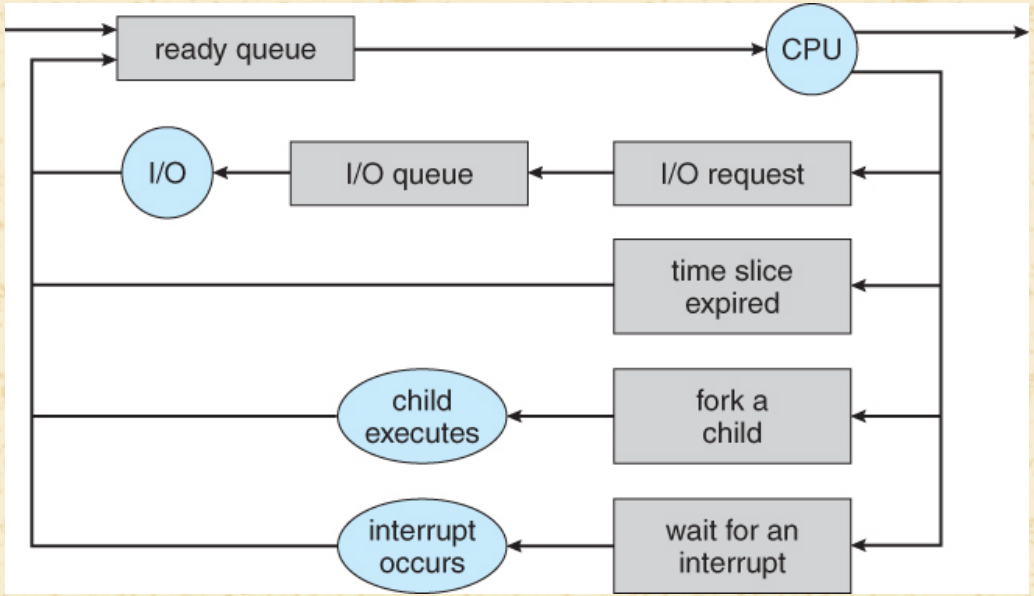
A depiction of the ready queue (top) and a few typical device queues:



The ready queue and various I/O device queues. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Getting to Run: Process Scheduling

A queueing diagram. Circles are the resources serving the queues, and arrows indicate the transition of processes:



Queueing-diagram representation of process scheduling. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Getting to Run: Process Scheduling

Any questions?

Context Switch

When the execution of one process's instructions on the CPU must pause due to an interrupt that occurred, the OS must save the current execution state of the process before it handles the interrupt because the OS needs to run the interrupt handler software on the CPU.

In such a **state-save** action, the operating system copies the process's data from the CPU to memory. When the OS is ready to resume the execution of the process on the CPU, it performs a **state-restore** action, in which the process's data is copied from memory back to the CPU.

The same also happens when we remove one process from the CPU in favor of another process that will run there (remember that the OS does this often to create the sense that processes run simultaneously.) The procedure of saving the state of one process, removing it from the CPU, and then restoring the state of a different process on the CPU is called a **context switch**. Context switches happen extremely frequently, so the operating system performs them very fast.

Context Switch

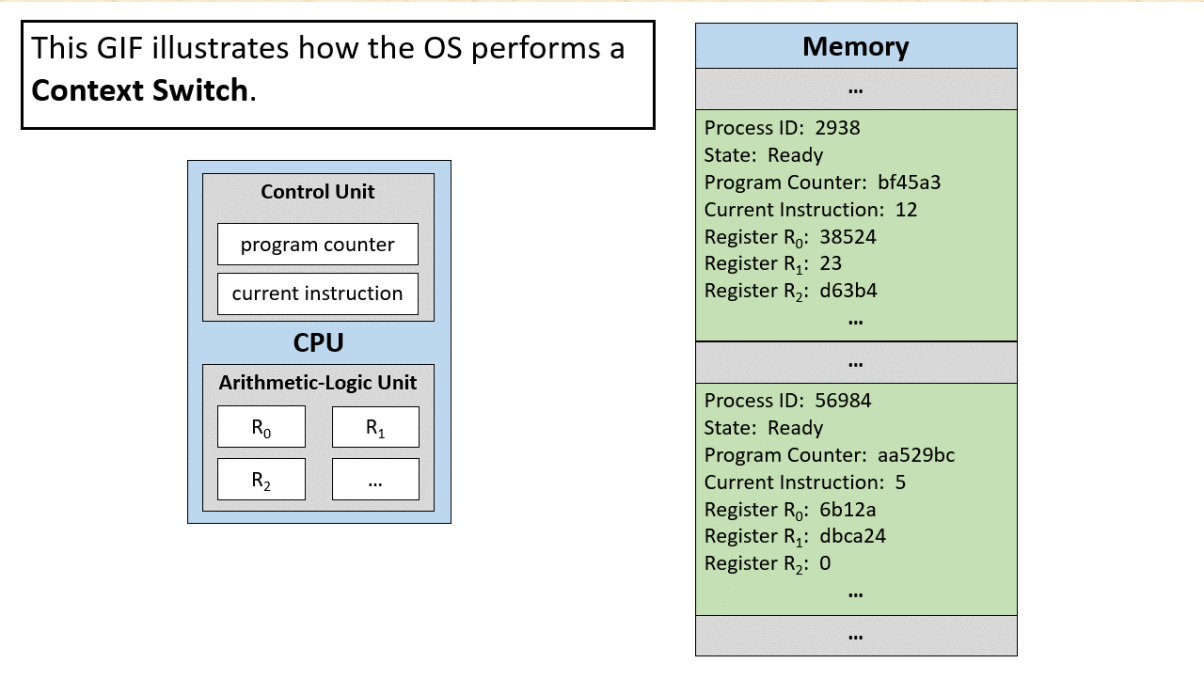


Diagram of a context switch. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Context Switch

Any questions?

Operations on Processes

Just as living things do, processes also have life cycles and family relations. A running process can control other processes via **process operations** (or functions.) Each operation that we call upon a process will either change the phase in its life cycle or modify its family.

1. A running process can create (also: **spawn** or **fork**) a new process.
    - In such a case, the original process is called the **parent** process, and the created one is referred to as the **child** process.
    - Every created process will be given a unique ID. In addition, each process will also store its parent's ID.
    - Well, if a process can create another process, who then is the first process that runs on the computer? This is usually the process scheduler program, and it is given the ID of 0.
    - The process scheduler then launches a process called **`init`** (or **`systemd`** in newer Linux versions) with the ID of 1. **`init`** then proceeds to create all the other processes on the computer, thereby becoming the Great Parent of all processes.

Operations on Processes

1. ◦ Some of the processes that **`init`** launches are daemons.

(Image taken from belloflostsouls.net.)

- A **daemon** is a process that the user cannot control directly through a terminal, and therefore is a process running in the background. Daemon processes are essential for the reliable activity of modern OSs.
- As a rule, if the name of a process ends with the letter ‘d’, this is most likely a daemon process. Example: **`sshd`** is a daemon process controlling the OpenSSH server; its purpose is to listen to incoming SSH connection requests and establish such connections.

◦ Following is a tree diagram showing an example of the process family on a computer. Notice that **`init`**, the ultimate parent, is at the tree’s root.

Operations on Processes

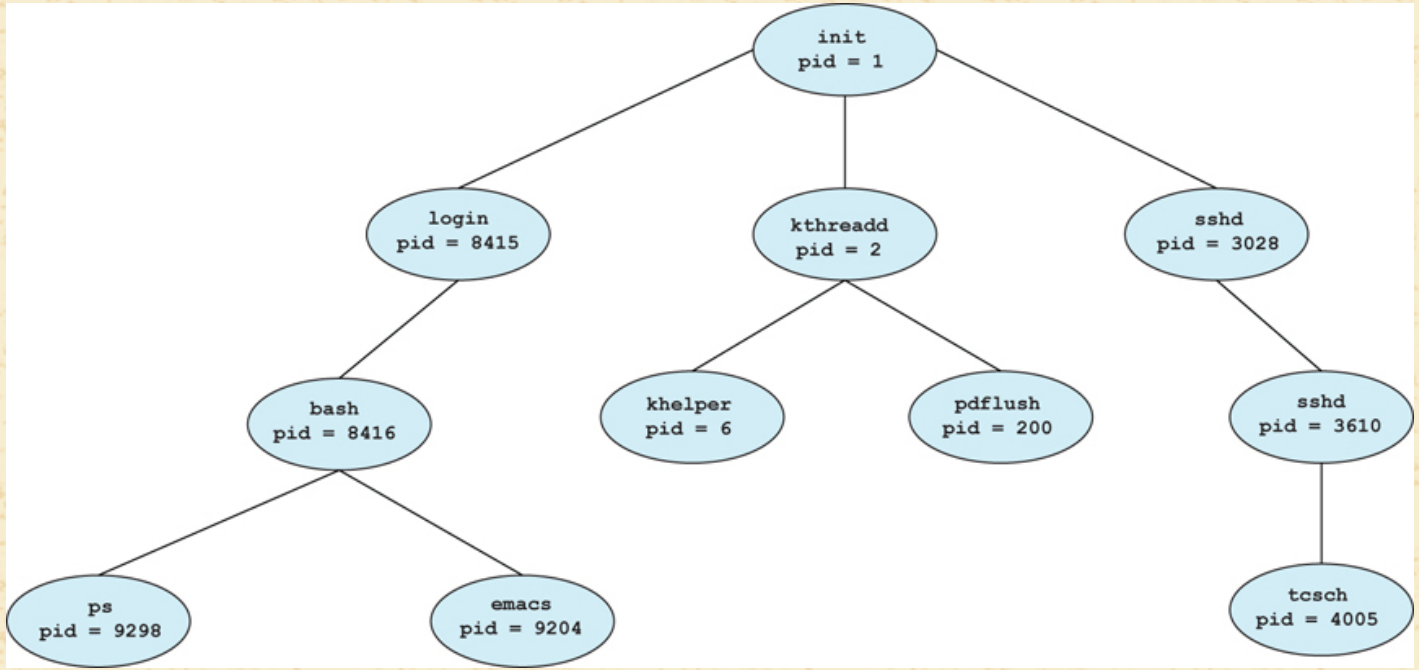
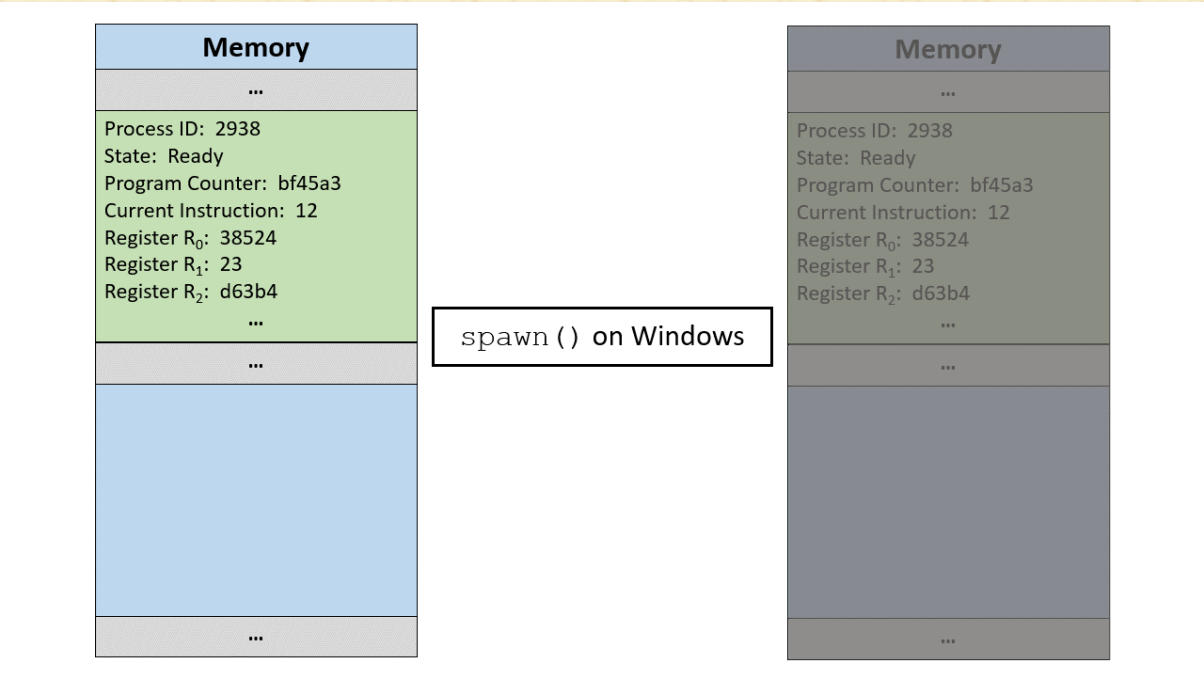


Diagram of process hierarchy. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Operations on Processes

1. ◦ What exactly happens inside the OS when a process is created? That depends on the operating system:
   i. On Windows, when a process calls the **`spawn()`** system call, the OS will load a program into memory by copying its code and initializing the process control block of the new process.
   ii. On Linux, the above is done in 2 separate steps:
      1. First, by issuing the **`fork()`** system call, the OS creates a child that is *identical* to the parent process.
      2. Then, by calling **`exec()`**, one of which arguments is a path to a program (binary) file, the child process asks the OS to replace its current binary code with the code of another program so that it might NOT be identical to the parent anymore.
   ◦ The diagram on the following slide shows how Windows (left) and Linux (right) let a process create a child.

Operations on Processes



Creating a New Process in Windows (left) and Linux/Mac (right). Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

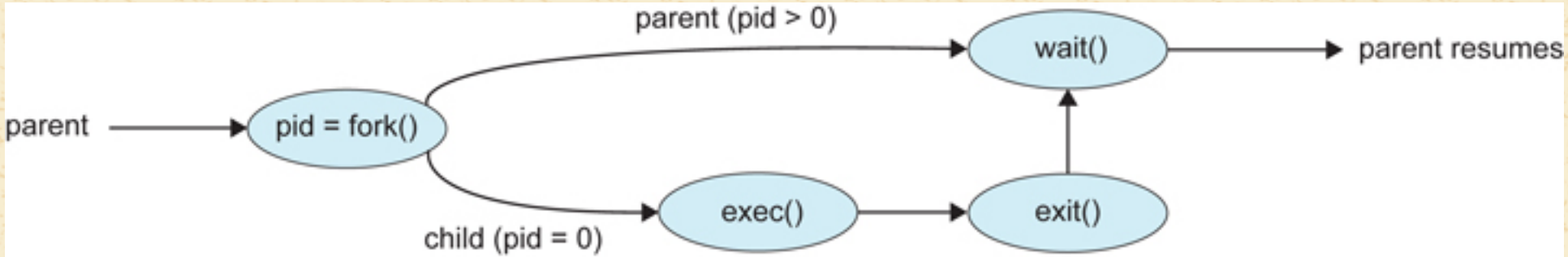
Operations on Processes

2. The parent process might issue a **`wait()`** system call, which will **block** (= pause) the parent process from executing until the child process terminates.
   - In some implementations, the `wait()` system call returns some statistical information on the child's running and termination (normal or abnormal.)
   - Since a process can have several children, you could use a variation of the wait function, **`waitpid()`** (or **`WaitForSingleObject()`** in Windows,) which will wait for a specific child to terminate.
3. Finally, a process is terminated when it calls the **`exit()`** system call.
   - When we return from the main function, the `exit()` system call is issued.
   - `exit()` returns an integer to its caller. As such, the parent process will get this integer when a child terminates via the status info returned from the `wait()` function.
   - Zero (0) usually indicates normal termination (success) and a non-zero indicates abnormal termination (failure).

Operations on Processes

3. ◦ Other ways in which a process can terminate are:
   i. When there are insufficient resources for the process to work with.
   ii. When a process ‘kill’ signal or another type of irrecoverable interrupt occurs.
   iii. When an illegal or faulty instruction was executed. E.g.: when we have a stack overflow or when we attempt to access an address that hasn’t been rightfully allocated by the program.
   ◦ If a parent didn’t wait for the child to terminate, which resulted in the parent terminating before the child, the still-running child process is called an **orphan**.
   ◦ When a child finished running and doesn't yet release its resources (e.g., allocated array in memory,) but the parent doesn’t wait for it, the child is called a **zombie**.
   ◦ To eliminate orphans and zombies, the `init` process will, from time to time, check upon the running processes and will adopt all orphan and zombie processes to terminate them.

Operations on Processes



The life cycle of a child process. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Operations on Processes

To illustrate how a process creates a child process, we created two Java programs:

- `Parent.java`, which you can find at https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3320/lecture_notes/topic_04/Parent.java, and
- `Child.java`, which you can find at https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3320/lecture_notes/topic_04/Child.java.

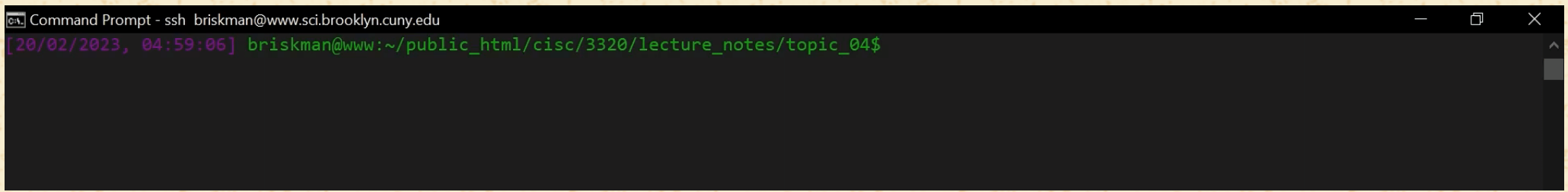
`Child.java` is a simple, short program that prints "Hello, World!" to the console. `Parent.java` compiles the `Child.java` program and executes it. After the `Child.java` program returns, `Parent.java` prints the output that `Child.java` produced (a.k.a, "Hello, World!") to the screen.

The programs were tested and executed on Windows and Linux, and should also work on Mac.

Operations on Processes

Before compiling and running the programs, ensure that (1) Java is installed on your computer and is added to the PATH environmental variable, and (2) Both `Parent.java` and `Child.java` were downloaded to the same folder. You can check that Java is on PATH by opening the command-line or terminal on your computer, and then typing "java". If you see instructions on how to use Java, then Java is installed and is on PATH.

Then, open the command-line or terminal on your computer, compile just the parent program using: `javac Parent.java`, and then run the parent with: `java Parent`.



Compiling and running the Parent process, which executes the Child process. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Operations on Processes

Any questions?

Inter-Process Communication

Process A and process B are called **Independent Processes** if the execution/output of process A doesn't affect the execution/output of process B and vise versa. On the other hand, **Cooperating Processes** are those whose execution/output can affect or be affected by this of other processes.

Reasons that we might be interested in having two processes cooperating with each other are:

1. **Information Sharing**: Several processes might need to access the same file/content.
2. **Computation speedup**: If we divide a problem into smaller pieces, and assign each process a task/piece, we might solve the problem faster with several processes than with one process.
3. **Modularity**: Some applications may benefit from division into modules, each controlled by a process.
4. **Convenience**: Dividing a major activity into smaller, inter-connected activities is convenient for the user.

Cooperating processes require some type of **inter-process communication**, abbreviated as **IPC**, which is most commonly one of two types: Shared Memory systems or Message Passing systems.

Inter-Process Communication

The video below is a beautiful, 5-min introduction to inter-process communication:

https://www.youtube.com/watch?v=W0BX6geRCDQ

IPC: Shared Memory

When two or more processes communicate via **shared memory**, each of these processes can read from and write to the same, shared chunk of main memory. [Some part of each process's address space, though, isn't shared with other processes.]

- Usually, the shared memory chunk initially belongs to one of the processes. To establish memory-sharing, other processes issue system calls to receive access permissions to this chunk of memory.
- Once the permission is received, this memory chunk becomes a part of the issuer's address space as well.
- Shared Memory is faster once it is set up, because no system calls are required once the communication was established, and memory access occurs at normal memory speeds.
- Therefore, shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.
- A classical example of this IPC method is the **Producer-Consumer** scenario, shown on Topic 9's slide 4.

IPC: Message Passing

When two or more processes communicate via **message passing**, each process can send messages to other processes and receive messages therefrom. The operating system actively behaves as the middleperson in passing the messages.

- Message passing requires system calls for every message transfer, and is therefore slower than memory sharing.
- However, it is simpler to set up and works well across multiple computers.
- Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.
- These systems must, at a minimum, support system calls for "sending a message" and "receiving a message".
- A message passing system must decide on (1) whether the sender must know the name of the receiver to which a message it to be sent, (2) whether sending or receiving messages can be blocking or non-blocking, and (3) how many messages are allowed to be stored by the OS temporarily before the receiver reads them.

IPC: Message Passing

Below is an illustration (pseudocode) of the Producer-Consumer programs using message passing:

```
message next_produced;

while (true) {
     /* produce an item in next_produced */

     send(next_produced);
}
```

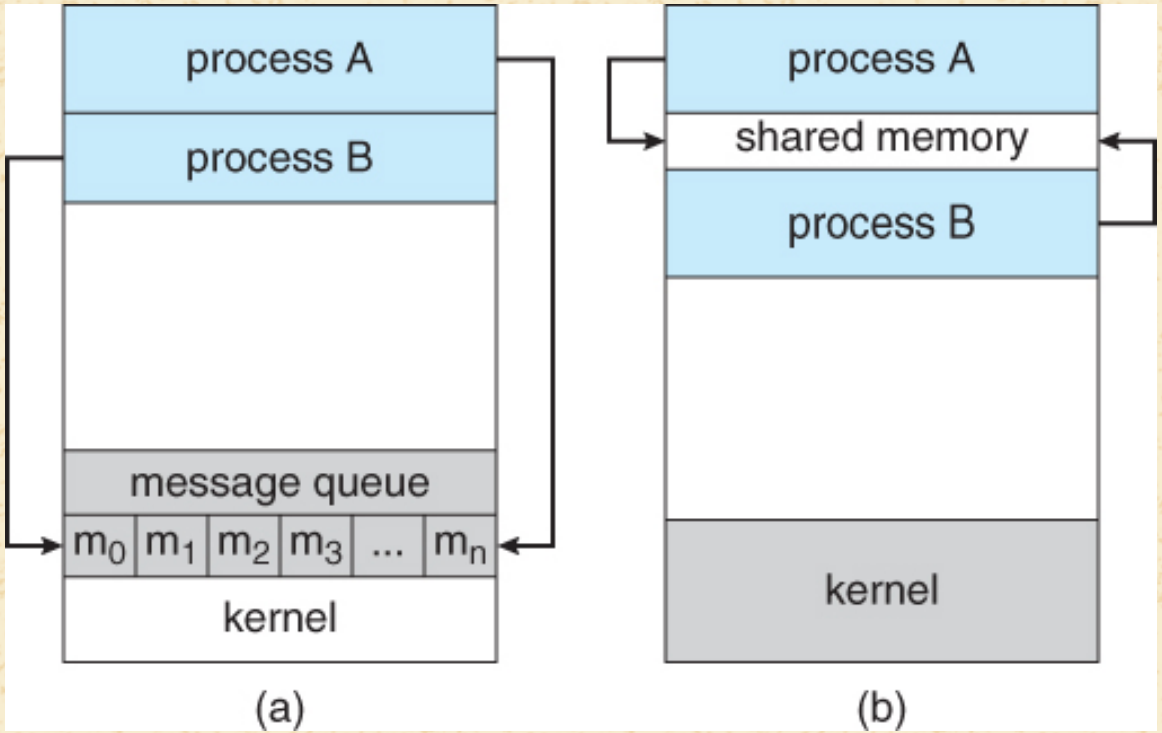
Figure 3.15 The producer process using message passing.

```
message next_consumed;

while (true) {
     receive(next_consumed);

     /* consume the item in next_consumed */
}
```

Figure 3.16 The consumer process using message passing.

Producer-Consumer programs using message passing. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Message Passing vs. Shared Memory: Comparison



Communications models: (a) Message passing. (b) Shared memory. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

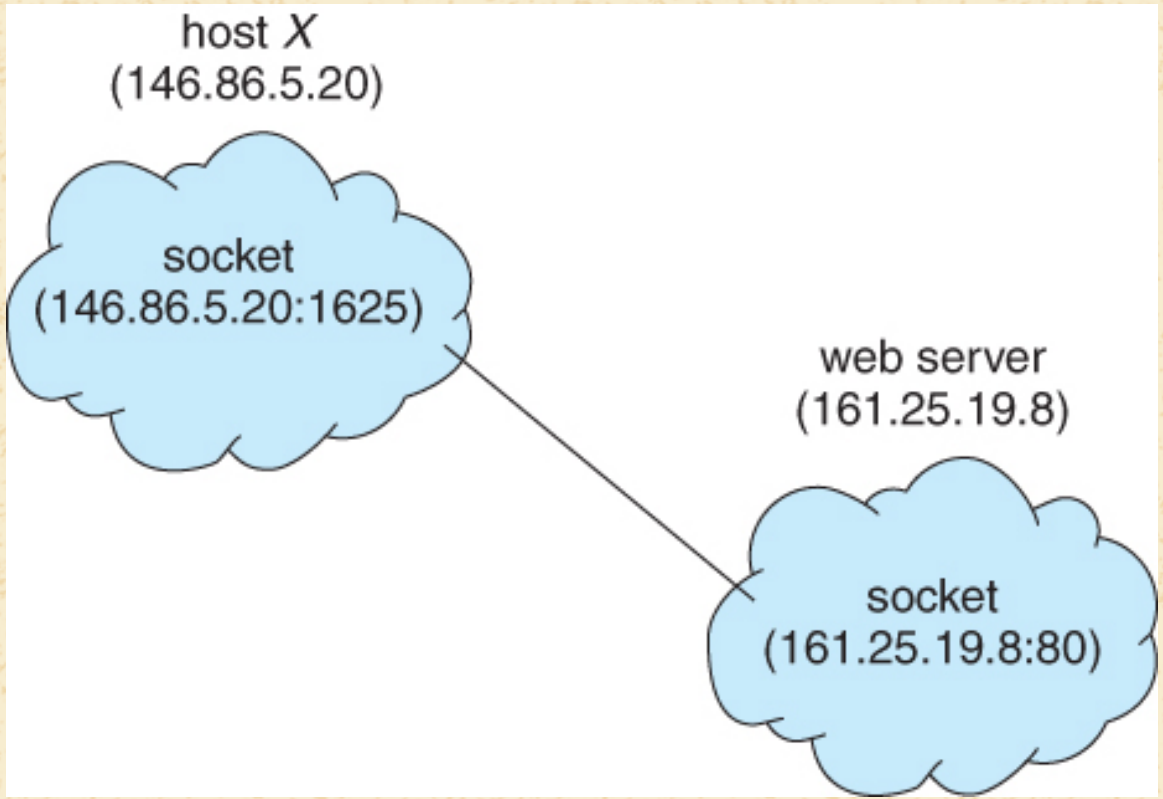
Sockets

A **socket** is one endpoint of a two-way communication link between two programs running on the network (this definition is taken from Oracle.) This isn't only about communication between programs on different devices: programs running on the same device may also communicate via a socket. Sockets are a type of Message Passing.

Each socket consists of two elements: an **IP address** and a **port**. An IP address (stands for **I**nternet **P**rotocol) is a structure consisting of 4 or 6 numbers separated by dots or colons (e.g., 192.0.2.1 or 2001:0db8:85a3:0000:0000:8a2e:0370:7334) that identifies a network of devices. A port is an integer between 1 to 65535 that identifies a specific device within that network.

When some program A wants to communicate with program B, it calls socket library functions. Program A passes the IP address and the port number of program B to the functions. For the communication to start, program B must call socket functions on its end, too, indicating program A's IP address and port number. The programs aren't even required to be written in the same language!

Sockets



An example of communication using sockets. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Sockets

A communication channel via sockets may take one of two major forms:

1. **Connection-oriented**: simulates a telephone connection. All the data that is sent down the connection are guaranteed to arrive in good condition at the other end and to be delivered to the receiving process in the order in which they were sent. Such a mechanism verifies all packets sent, re-sends packets if necessary, and arranges the received packets in the proper order before delivering them to the receiving process. Performing all these checks consumes extra time. Example of a mechanism: **TCP** (= **T**ransmission **C**ontrol **P**rotocol.)
2. **Connectionless**: simulates individual telegrams. There is no guarantee that any particular datum will get through undamaged (if at all) and no guarantee that the data will get delivered in any particular order. There may even be duplicate copies of the data delivered, depending on how the intermediary connections are configured. Such transmissions are much faster connection-oriented ones, but apps must implement their own error-checking and recovery procedures. Example of a mechanism: **UDP** (= **U**ser **D**atagram **P**rotocol.)

Remote Procedure Calls

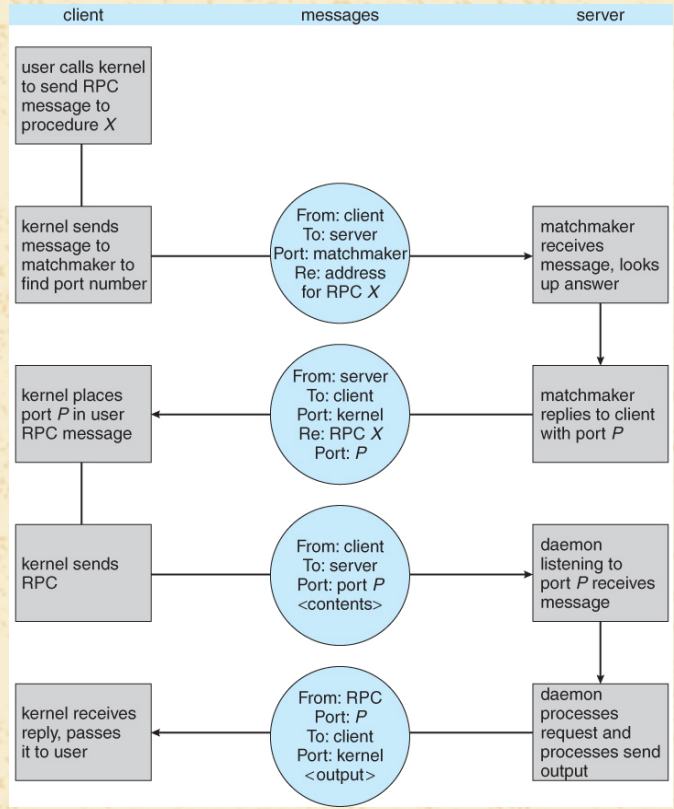
A **Remote Procedure Call** (**RPC**) is a function that executes on another computer.

Here is how it works: when a program on device A wants a value to be calculated on device B, the program sends a message to device B, along with parameter values. Device B then executes the requested function using the parameters it received from device A's program. Finally, device B sends the results of the function back to the program on device A. As such, RPC is another type of Message Passing.

Since the two devices might differ in their architecture and features, before an RPC request is issued, both devices should first agree upon the format of the data that is sent back and forth (to avoid confusion and misinterpretation.)

An example of RPC is a networked file system. Messages are passed from device A to device B to read, write, delete, rename, or check the status of data or files stored on device B, similarly to how device A would access its own disks.

Remote Procedure Calls



How an RPC executes. Taken from Bell, John T. "Processes." University of Illinois, Chicago.

Pipes

A **pipe** is a simple type of channel of communication between two processes.

UNIX/Linux and Windows systems implement pipes, usually as files.

To start communicating, both processes call functions that create the pipe(s). Then, a process sends data through a pipe by issuing a system call similar to `write()`, while the other process accesses this data by calling `read()`. Since the data exchange is moderated by the OS (via the system calls,) a pipe is also a Message Passing mechanism.

Pipes can be implemented as one-directional or bidirectional. A bidirectional pipe allows data to be sent back and forth. In addition, some pipes can be used only by processes on the same computer, while other pipe types can be used by those running on two different devices attached to a network.

Two general types of pipes are commonplace: Ordinary Pipes and FIFOs.

Ordinary Pipes

Ordinary pipes (**Anonymous pipes** on Windows) are one-directional. Data is written into the pipe on one end and then is read from the pipe at the other end. (When the two processes need to communicate back and forth, then a second pipe is created.) Such pipes can be used only between parent and child processes.

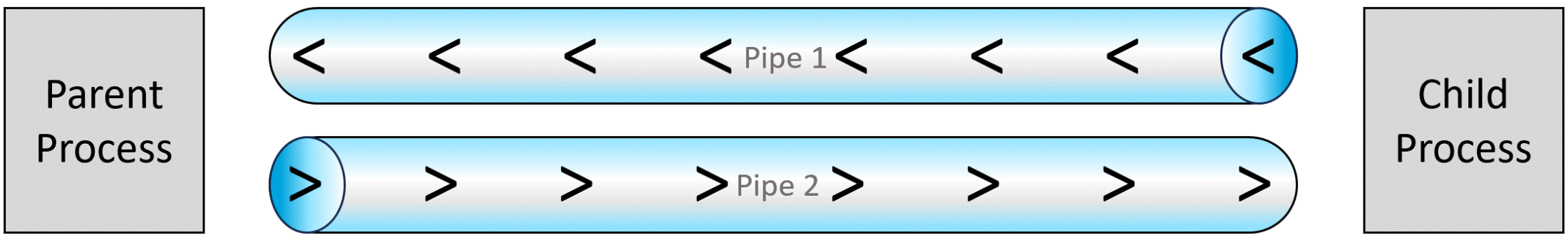
On UNIX/Linux, an ordinary pipe is created via the `pipe()` system call, while a Windows process calls `CreatePipe()` to create one.

Ordinary pipes are only accessible within the process that created them. That is, if a process that terminates hasn't closed the pipes, the pipe is anyway 'destroyed' once the process terminates.

Typically, a parent creates the pipes before forking a child. Since a child process always inherits open files from its parent on UNIX/Linux, including the pipe file(s), a channel of communication is then established. On Windows, the programmer must specify that the child has to inherit the pipes.

Ordinary Pipes

Here is an illustration of how data is exchanged between a parent process and a child process via ordinary pipes:



Two ordinary pipes allow message passing between a parent and a child. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

FIFOs

FIFOs (**Named pipes** on Windows), unlike ordinary pipes, are:

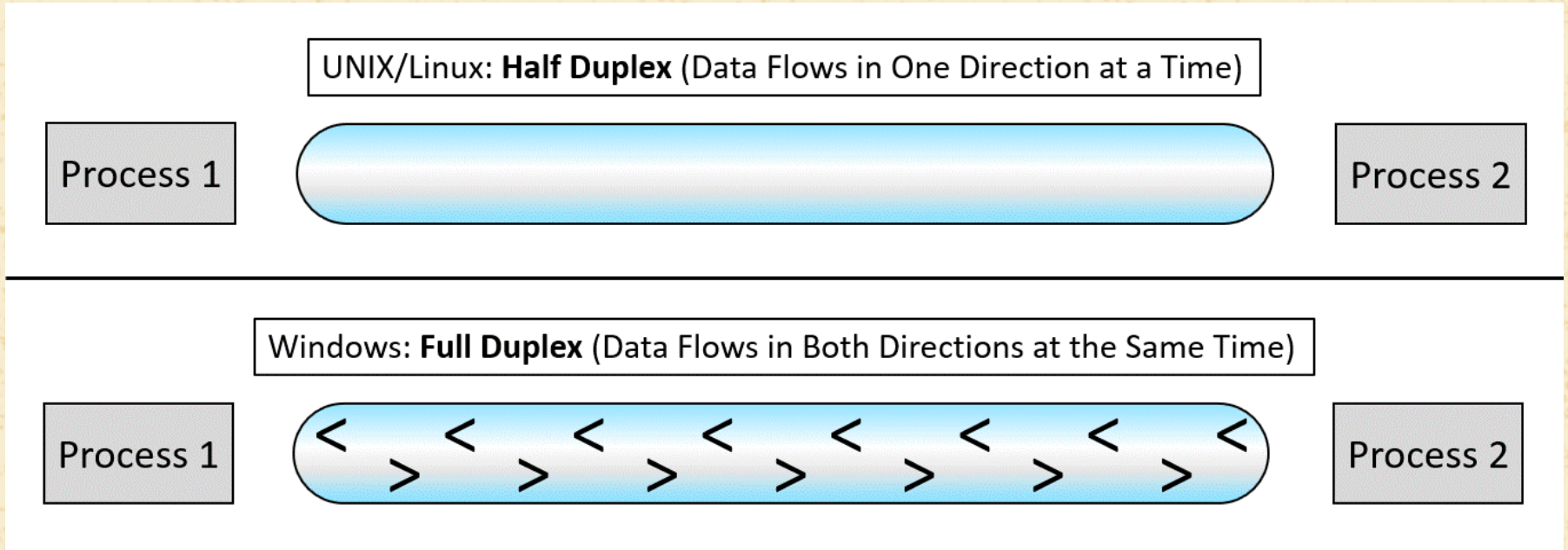
- Bi-directional (data can flow in a FIFO in either direction,)
- Can connect processes that aren't in a parent-child relationship,
- May keep existing even after the process that created them terminates, and
- Can connect more than just two processes.

On UNIX/Linux, one creates a FIFO using the `mkfifo()` system call. On Windows, it is created using the `CreateNamedPipe()` system call.

Moreover, on UNIX/Linux, a FIFO can operate between processes on the same machine only, while Windows supports Named pipes between processes on different machines, too.

FIFOs

One more difference between UNIX/Linux and Windows is that only Windows supports full duplex FIFOs:



Half duplex FIFOs (UNIX/Linux) vs. full duplex FIFOs (Windows). Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Inter-Process Communication: Summary

A summary of what we covered on IPC. Miriam Briskman, CC BY-ND 4.0.

Note: click the GIF amination above to open and play it in a browser window.

Inter-Process Communication

Any questions?

Processes: Sources

Bell, John T. "Processes." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/3_Processes.html

Chapter 5: The Process (pp. 84-96).

Wienand, Ian (2019). Computer Science from the Bottom Up. This work is licensed under the Creative Commons Attribution-ShareAlike3.0 Unported License. URL: https://www.bottomupcs.com/csbu.pdf