Inter-Process Communications (IPC)

ICS332

Operating Systems

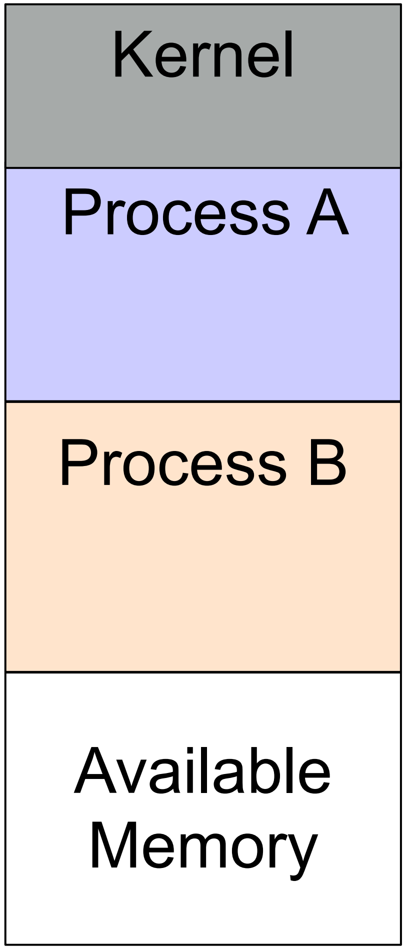
Henri Casanova (henric@hawaii.edu)

Communicating Processes?

- So far we have seen independent processes
  - Each process runs code independently
  - Parents are aware of their children, and children are aware of their parents, but they do not interact
    - Besides the ability to wait for a child to terminate and to kill another process
- But often we need processes to cooperate
  - To share information (e.g., access to common data)
  - To speed up computation (e.g., to use multiple cores concurrently)
  - Because it's convenient (e.g., some applications are naturally implemented as sets of interacting processes)
- But, processes cannot see each other's address spaces!
- In general, the means of communication between cooperating processes is called Inter-Process Communication (IPC)

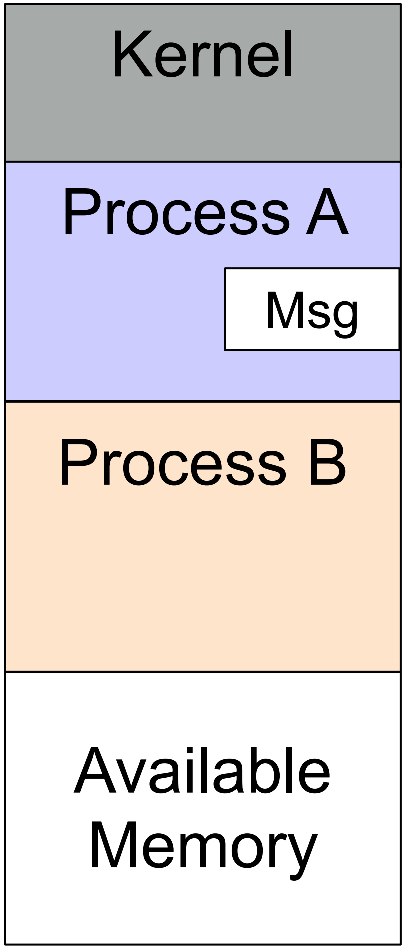
Communication Models

- Process A needs to communicate with Process B



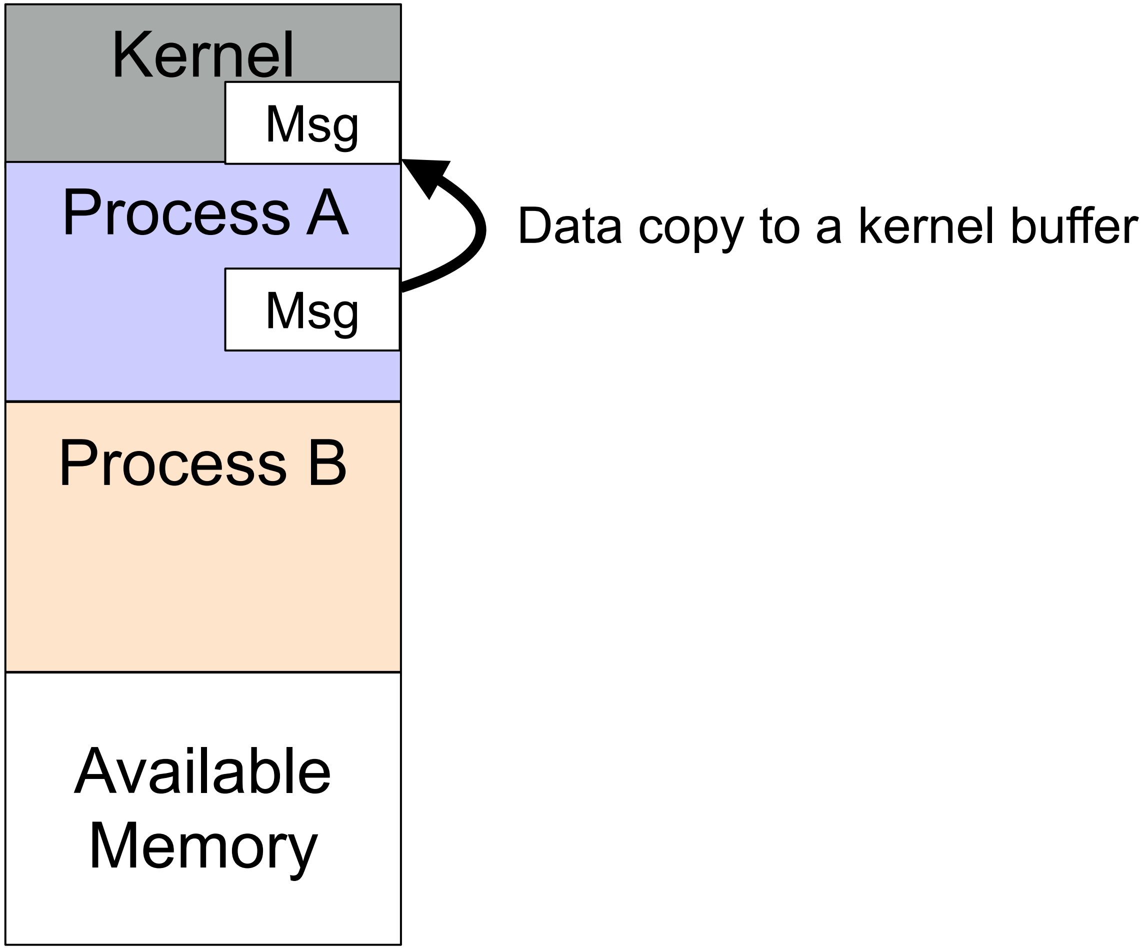
Message Passing

- Option #1: Message Passing



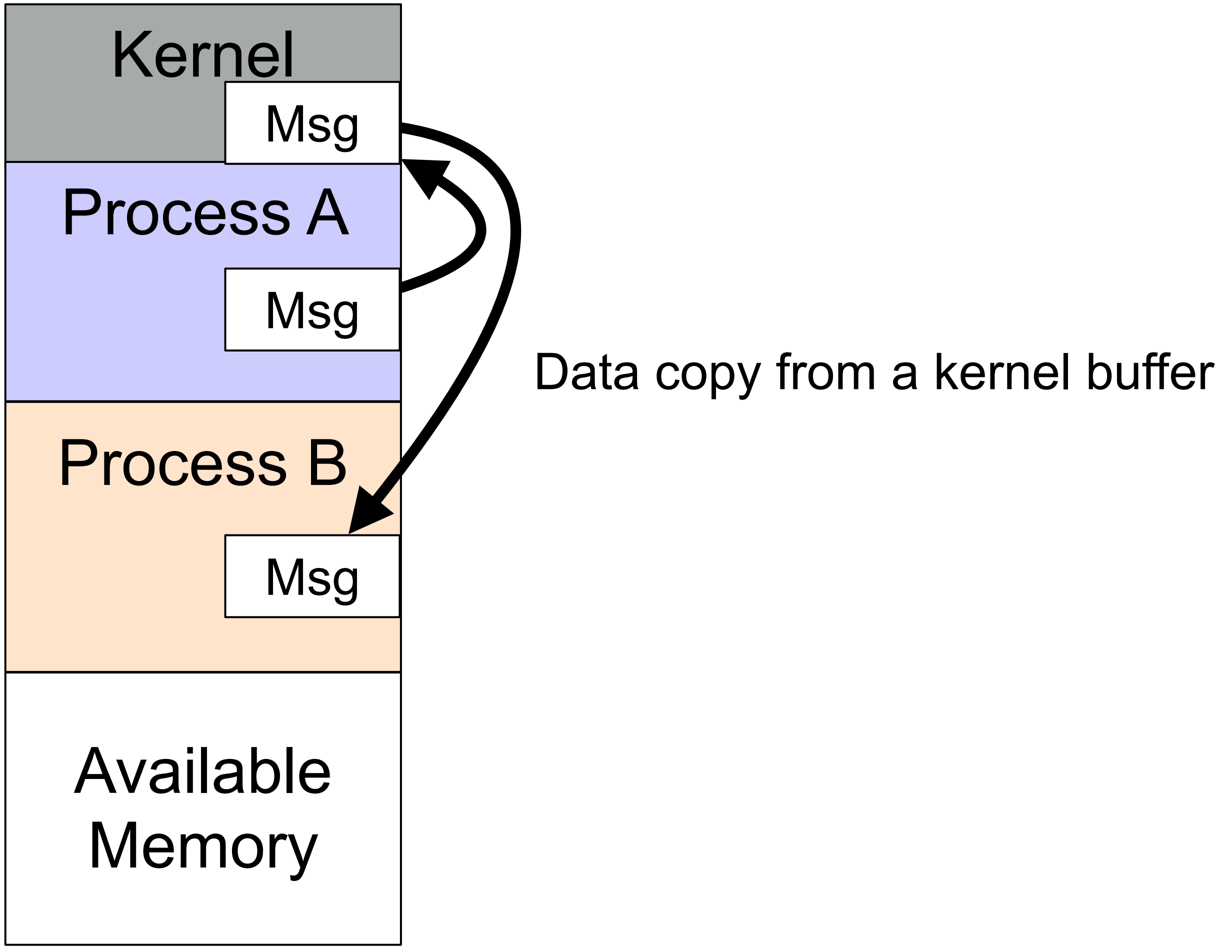
Message Passing

- Option #1: Message Passing



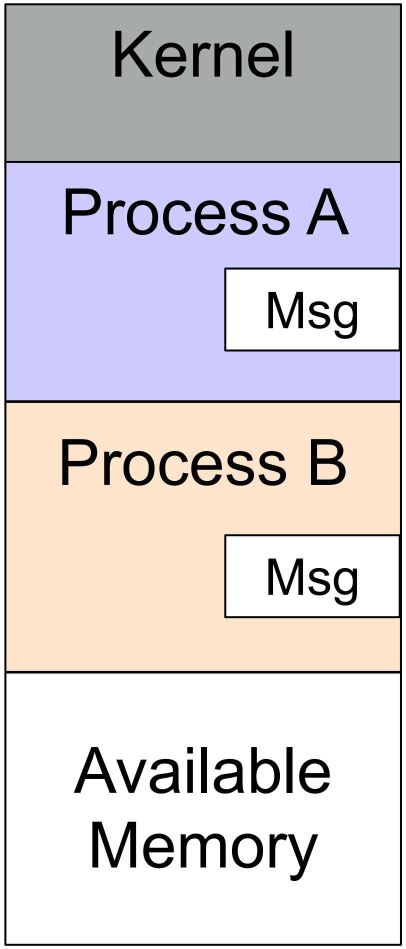
Message Passing

- Option #1: Message Passing



Message Passing

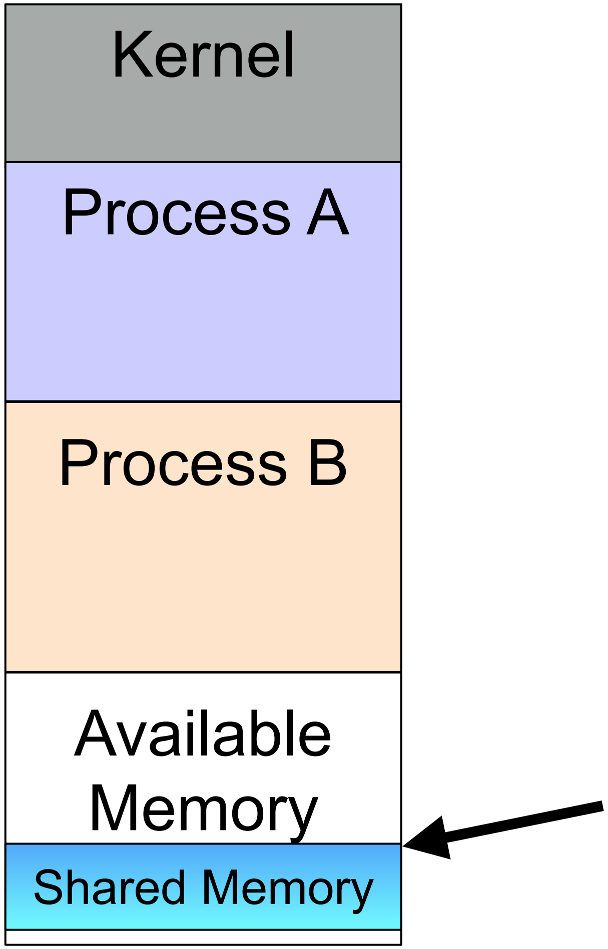
- Option #1: Message Passing



Process B now has the message in its address space

Message Passing

- Option #2: Shared Memory



A zone of memory that "belongs" to both processes's address space, so that each can read/write at will it it and the other can "see" it all

Pros and Cons

- Message Passing
  - 😀 Simple to implement in the kernel
  - 😡 Limited by kernel size: small messages
  - 😡 One syscall per operation (send / receive): high overhead
  - 😡 Cumbersome for users as code can be hard to read with sends/receives everywhere
- Shared memory
  - 😡 Not as easy to implement in the kernel (stay tuned…)
  - 😀 Large messages allowed
  - 😀 Low overhead: a few syscalls to set it up, and then no kernel involvement thereafter
  - 😀 Convenient for users (after setup, just normal memory reads/writes)
  - 😡 Violates the principle of memory protection between processes, which can lead to horrible bugs

Message Passing

- All OSes provide several IPC abstractions and API
  - And so do many user-level libraries
- In your careers you will have to define abstraction and APIs for all kinds of purposes
- Abstraction and API design choices often seem innocuous but can have huge impact
  - Good choices can lead to awesome success, bad choices can lead to abject failures/rewrites
- Making good Abstraction/API choices is hard:
  - Sufficiently expressive (can users do anything they might want to do with it?)
  - Sufficiently convenient (can users do what they want easily?)
  - Not too hard for you to implement/maintain/evolve
- **Pedagogic challenge:** Conveying to college students how important/crucial this is, when it all seems like a bunch of pointless nitpicking
  - You wouldn't believe the number of hours spent daily on minuscule API details in the software industry
  - Because you haven't yet experienced the above "snowball effect" of your poorly designed Abstractions/API

POSIX Message Queue

- A standard message passing scheme supported by UNIX-like systems are POSIX Message Queues
  - There is a message queue “object” that has a name, a maximum message size, and a maximum number of messages in the queue
  - Both processes create their own queue object using the same name (meaning they both have a reference to the same queue)
  - The queue object supports send/receive operations
- This Abstraction/API makes several design choices
  - One option called “direct communication” would have been “I am process A and I send a message to process B”, which requires that process B is created/known when A does the send
  - Instead, this API uses “indirect communication” by using a message queue object, which is more flexible
- Just for kicks let’s look at a hello world example…

POSIX MQ Hello World

```
pid_t pid = fork();

if (pid) { // parent

  mqd_t queue = mq_open("mq", O_CREAT | O_WRONLY, 0664, NULL);
  char msg[MSG_SIZE] = "Hello!";
  mq_send(queue, msg, MSG_SIZE, 1);
  waitpid(pid, NULL, 0);
  mq_close(queue);
  mq_unlink(MQ_NAME);

} else { // child

  mqd_t queue = mq_open("mq", O_CREAT | O_RDONLY, 0664, NULL);
  char msg[MSG_SIZE];
  mq_receive(queue, msg, MSG_SIZE, NULL);
  mq_close(queue);
  mq_unlink(MQ_NAME);

}
```

- Let's look at and run the real/full code in posix_mq_example.c
- **Conceptually** this is just like network communication, but within a machine
- There are MANY abstractions/implementations of message passing for all kinds of scenarios/purposes, each with slight differences

POSIX Shared Memory Segments

- Like there is a POSIX MQ API, there is a POSIX SHM (Shared Memory) API
- The abstraction is that of a “shared memory segment” with a simple API
- One process can create a shared memory segment
- Multiple processes can then attach it to their address spaces
  - Bye bye memory protection
  - It’s the processes’ (i.e., the developer’s) responsibility to make sure that processes are not stepping on each other’s toes
- Once the setup is done, the OS is not involved
  - What happens in shared memory stays in shared memory
- At some point, the shared memory segment is freed by the requester
- Let’s look at a Hello World example…

POSIX SHM Hello World

```
int segment_id = shmget(IPC_PRIVATE, 10*sizeof(char), SHM_R | SHM_W);

pid = fork();
if (pid) { // parent

  char *shared_memory = (char *)shmat(segment_id, NULL, 0);
  sprintf(shared_memory, "hello");
  waitpid(pid, NULL, 0);
  shmdt(shared_memory);
  shmctl(segment_id, IPC_RMID, NULL);

} else { // child

  char *shared_memory = (char *)shmat(segment_id, NULL, 0);
  fprintf(stdout,"Child: read '%s' in SHM\n", shared_memory);
  shmdt(shared_memory);

}
```

- Let’s look at and run the real/full code in posix_shm_example.c

POSIX SHM Hello World

```
int segment_id = shmget(IPC_PRIVATE, 10*sizeof(char), SHM_R | SHM_W);

pid = fork()
if (pid) { /

   char *shar
   sprintf(sh
   waitpid(pi
   shmdt(shar
   shmctl(seg

} else { //

   char *shared_memory = (char *)shmat(segment_id, NULL, 0);
   fprintf(stdout,"Child: read '%s' in SHM\n", shared_memory);
   shmdt(shared_memory);

}
```

> Note that the child needs the **`segment_id`**. In this case, we're ok because **`shmget()`** is called before **`fork()`**. But if the child was a different program (e.g., after an **`exec()`**), then the **`segment_id`** would need to be communicated to the child (e.g., via message passing!!)

- Let's look at and run the real/full code in posix_shm_example.c

The IPC Zoo

- There are many IPC abstractions that fall into the message passing or the shared memory category, or blur the lines
  - Signals, sockets, message queues, pipes, shared memory segments, files, …
- Several abstractions share common characteristics but have a few key differences (e.g., a message queue and a socket)
- There is a distinction between the abstraction that's exposed by the API and the implementation of this API
- In fact, many abstractions can be implemented on top of others
  - message queues on top of shared memory segments
  - message queues on top of files
  - message queues on top of sockets
  - shared memory segments on top of message passing
  - …
- Some implementations are only for IPCs within a machine, some implementations are also for across machines over a network
- Let's now talk about a very, very commonplace abstraction: pipes

Pipes

- One of the most ancient, yet simple, useful, and powerful IPC mechanism provided by OSes is typically called pipes
- Before we get into pipes, we need to take a little detour about UNIX file descriptors and output redirection...

stdin, stdout, stderr

- In UNIX, every process comes with 3 already opened “files”
  - Not real files, but in UNIX “everything looks like a file” by design
- These files, or streams, are:
  - stdin: the standard input stream
  - stdout: the standard output stream
  - stderr: the standard error stream
- You’ve encountered these when developing code in all languages (C/C++, Java, Python, etc.)
  - e.g., printf() writes to stdout
- Each file in UNIX is associated to an integer file descriptor
  - An index into some “this process’ open files” table
- By convention, the file descriptors for each standard stream are (see `/usr/include/unistd.h`):
  - stdin: STDIN_FILENO = **0**
  - stdout: STDOUT_FILENO = **1**
  - stderr: STDERR_FILENO = **2**

Re-directing output

- Perhaps some of you have wondered how come something like **`ls > file.txt`** can work?
- After all, **`ls`** has code that looks like:

```
fprintf(stdout, "%s", filename);
```

- So how can this code magically knows to write to a file instead of to stdout when I put a “>” on the command line???
- This is one of the famous UNIX “tricks”
- In UNIX, when I open a new file, this file gets the first available file descriptor number
- So, if I close stdout, and open a file right after, this file will have file descriptor 1
- Therefore, **`printf()`** will write to it as if it were stdout
  - Because **`fprintf(stdout, …)`** really is just **`fprintf(1, ...)`**
  - And I don’t need to change the code of **`ls`** at all!!!
- Let’s see an example program…

Output Redirect Example

Example program fragment

```
...
pid_t pid = fork();
if (!pid) { // child
  // close stdout
  close(1);
  // open a new file, which gets file descriptor 1
  FILE *file = fopen("/tmp/stuff", "w");
  // exec the "ls -la" program
  char* const arguments[] = {"ls", "-la", NULL};
  execv("ls", arguments);
}
...
```

- This program will run `ls -la` and write its output to file `/tmp/stuff`
- Let's look at output_redirect_example1.c

What if I opened the file before calling fork()?

- In the previous example, the sequence of operation is:
  - Close stdout
  - Open a new file, which then gets file descriptor 1
- What if I have already opened the file and it has some other file descriptor?
- This is why the **`dup()`** syscall is there: file descriptor duplication!
  - Essentially, **`dup()`** allows you to say “Create another file descriptor for an existing opened file”, and it will always pick to lowest unused descriptor number
  - The **`fileno()`** library call returns the descriptor of an open file
- So the sequence is:
  - **`FILE *some file = fopen(....);`**
  - **`close(1);`**
  - **`dup(fileno(some file));`**
- After this sequence, writing to file descriptor 1 writes to the file instead!
- Let’s see a simple example again...

Another Output Redirect Example

Example program fragment

```
...
FILE *file = fopen("/tmp/stuff", "w");

pid_t pid = fork();
if (!pid) { // child
  // close stdout
  close(1);
  // duplicate the file's file descriptor
  dup(fileno(file));
  // exec the "ls -la" program
  char* const arguments[] = {"ls", "-la", NULL};
  execv("ls", arguments);
}
...
```

- This program will run `ls -la` and write its output to file `/tmp/stuff`
- Let's look at output_redirect_example2.c

UNIX Pipes

- A pipe is a simple IPC mechanism between two processes
- One can create a pipe so that process A can write to it and process B reads from it
- Available in the shell with the | symbol: the output of a process becomes the input of other(s)
  - Just like a file indirection, but to another process' input stream
- Example: Count the files whose names contain foo but not bar in the /tmp directory
  - List all files in /tmp: **`find /tmp -type f`**
  - Keep those with foo: **`grep foo`**
  - Remove those with bar: **`grep -v bar`**
  - Count the lines that remain: **`wc -l`**

Putting everything together: **`find /tmp -type f | grep foo | grep -v bar | wc -l`**

popen(): fork() with a pipe!

- Very convenient library functions are **`popen()`** and **`pclose()`**
- Sounds like "pipe open" and "pipe close", but it's MUCH more than that
- **`popen()`** does:
  - Creates a (bi-directional) pipe, and we have to specify whether we're going to read ("r") or write ("w") to it
  - Forks and execs a child process (e.g., "ls -a")
  - Returns the pipe, which is in fact a file (FILE *)
  - Both the parent and the child can "talk" through the pipe!
- **`pclose()`** does:
  - Waits for the child process to complete
  - Closes the pipe
- These are implemented with several system calls: **`fork`**, **`waitpid`**, **pipe** (which creates a pipe), **`close`**, **`open`**, **`dup`**
- Re-implementing **`popen/pclose`** would be a bit too much here, but let's just see an example program that uses it...

popen() / pclose() Example

Example program fragment

```
// fork/exec a child process and get a pipe to READ from
FILE *pipe = popen("/usr/bin/ls -la", "r");

// Get lines of output from the pipe, which is just a FILE *,
// until EOF is reached
char buffer[2048];
while (fgets(buffer, 2048, pipe)) {
  fprintf(stderr,"LINE: %s", buffer);
}

// Wait for the child process to terminate
pclose(pipe);
```

- This program prints all the output produced by `ls -la`
- Almost all languages provide something like this: Python's subprocess module, Java's ProcessBuilder class, etc.
- Let's look at and run popen_example1.c
- And then let's look at and run popen_example2.c, which opens a pipe to **write** to

Higher-Level IPC?

- What we've seen so far are IPC abstractions for processes to exchange raw bytes
- With that one can do everything, since the bytes can be encoded/interpreted in arbitrary ways
- Often IPC is used to ask another process to do something for us and send us back the result
- This is conceptually like calling a method/ function on the other process
- A powerful abstraction has been proposed to do this more easily than with just byte messages: Remote Procedure Call (RPC)

RPC

- RPC provides a procedure invocation abstraction across processes (and actually across machines)
- A client invokes a procedure in another process (almost) as it would invoke it directly itself
- RPC has a lot of usages, of course for client-server applications (and microkernels!)
- The "magic" is performed through a client stub (one stub for each RPC):
  - Marshal the parameters (converts structured data to bytes)
  - Send the data over to the server
  - Wait for the server's answer
  - Unmarshal the returned values (convert bytes to structured data)
- A lot of different implementations exist... including in Java

Java Remote Method Invocation (RMI)

- RPC in Java: Remote Method Invocation (RMI)
- A process in a JVM can invoke a method of an object living in another JVM
- Marshalling/Unmarshalling of data is performed by the JVM
  - Each object must be from a class that implements the java.io.Serializable interface
- RMI hides all the gory details of RPC/IPC
- See this Java RMI Tutorial for more info
- We'll come back to RMI later…

Main Takeaways

- Two kinds of mechanisms for processes to communicate:
  - **Message Passing:** Within the kernel Space
  - **Shared Memory:** Outside the kernel Space
- Both kinds of mechanisms are implemented in all mainstream OS and **many** variants and abstractions exist
  - Message Queues, Shared Memory Segments, Files, Signals, Sockets, Pipes, RPC
- UNIX Pipes and various output redirections mechanisms (to files, to the parent process)
- Concept of RPC

Conclusion

- The line between message passing and shared memory is often blurred by abstractions, and abstractions of one kind can be implemented on top of abstractions of the other kind
  - For instance, it would be easy to implement a "message passing" pipe abstraction using a "shared memory" implementation
- Let's look at Optional Homework Assignment #3