Object Oriented Methodology

Week – *10*, Lecture – *2*

Shared Memory and Forking

SAURABH SRIVASTAVA
VISITING FACULTY
IIIT LUCKNOW

Communicating between Processes

Remember that we discussed that Threads can be considered as “lightweight processes”

This is because Threads can easily communicate with each other, e.g., via global variables

However, what if two processes wish to communicate with each other?

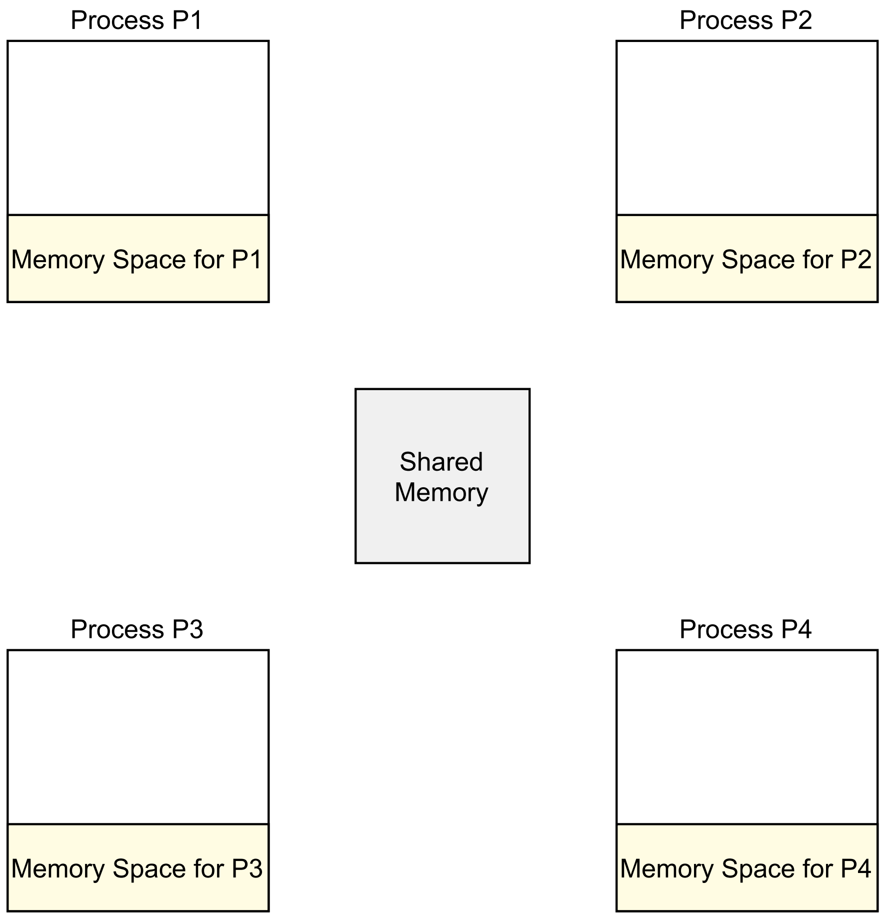
- Since they do not share any common variables, or memory blocks in general !!

The simplest way to do so, is to “create” such a common Memory block …

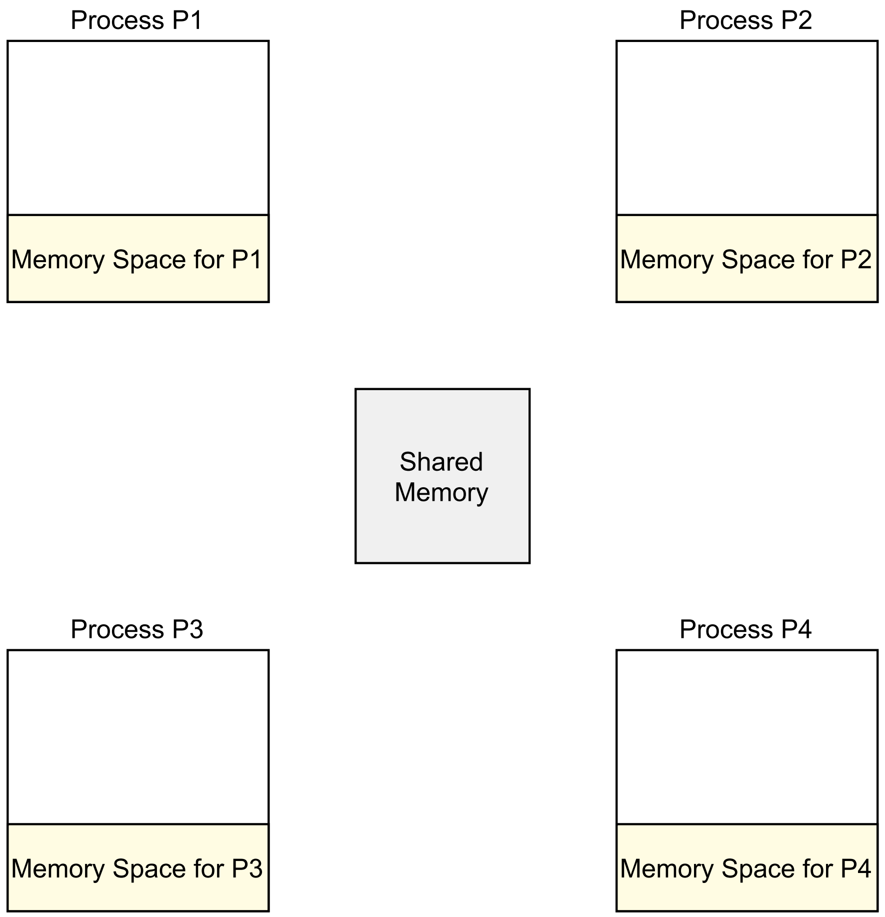
- … all the communicating processes can read to or write from this common block

This block of memory is popularly called a *Shared Memory*

Communicating through Shared Memory



Communicating through Shared Memory



A shared memory is a separate block within the main memory, which is not a part of any process's private memory space

Communicating between Processes

Remember that we discussed that Threads can be considered as "lightweight processes"

This is because Threads can easily communicate with each other, e.g., via global variables

However, what if two processes wish to communicate with each other?

- Since they do not share any common variables, or memory blocks in general !!

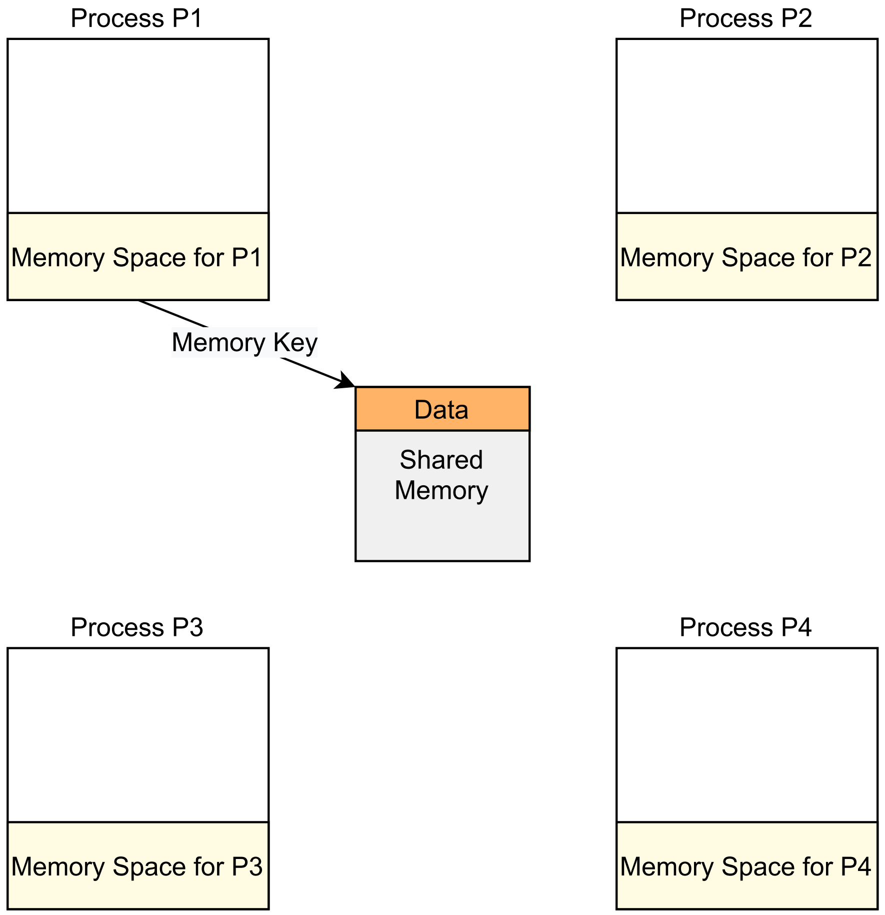
The simplest way to do so, is to "create" such a common Memory block …

- … all the communicating processes can read to or write from this common block

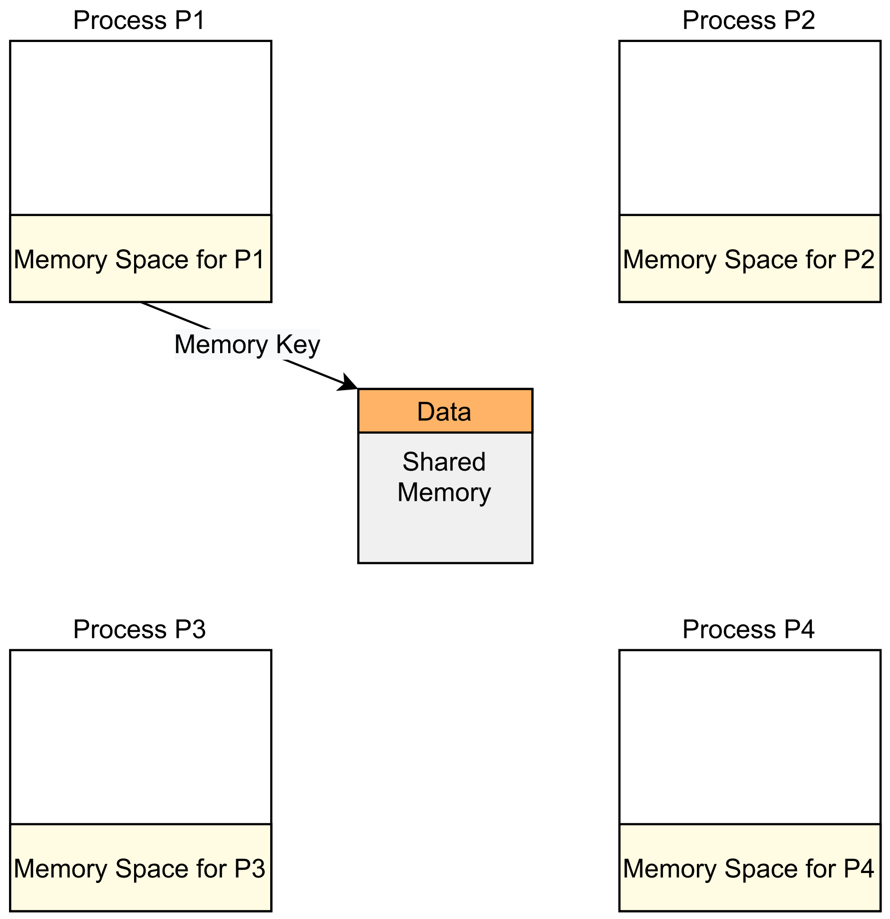
This block of memory is popularly called a *Shared Memory*

A shared memory is identified by a *key*

Communicating through Shared Memory

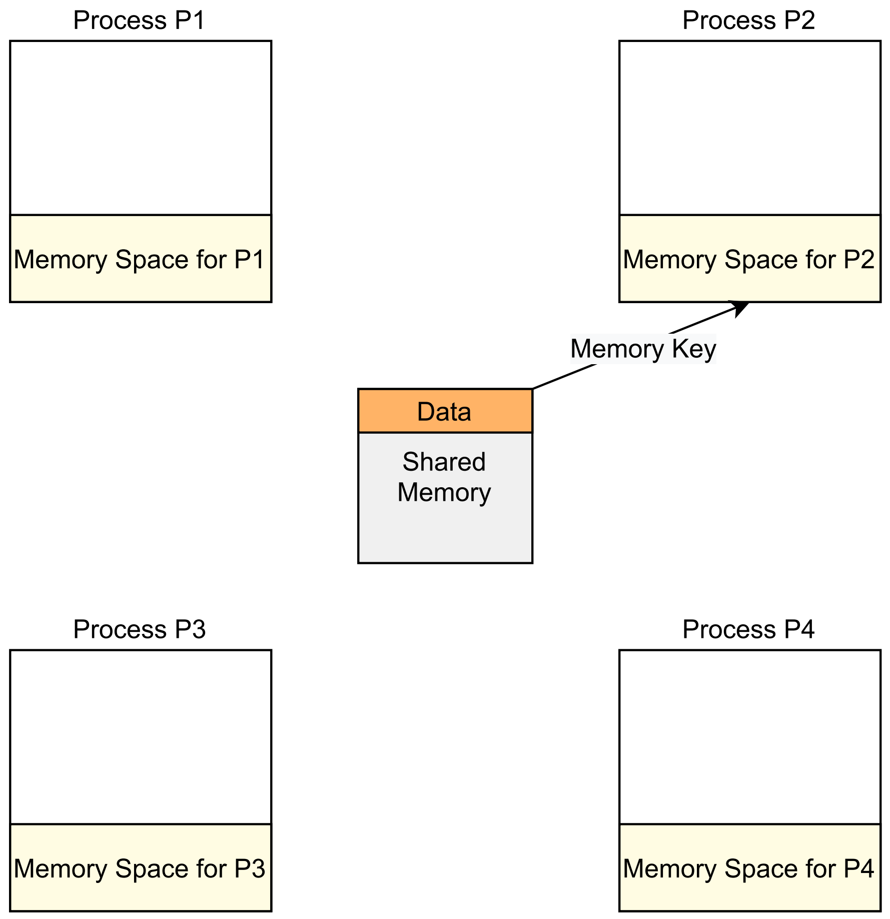


Communicating through Shared Memory



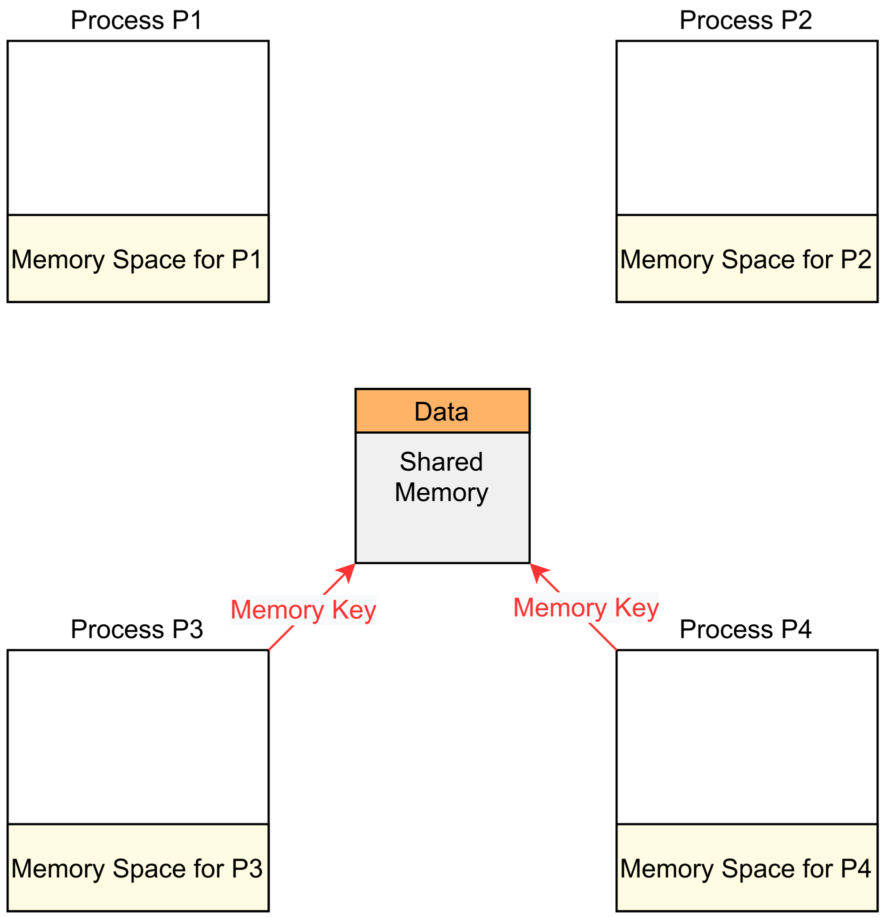
To access the shared memory, this Key is required

Communicating through Shared Memory



As long as the other communicating processes knows this key, they can access the shared memory …

Communicating through Shared Memory



… but this does mean that a process which is not authorized to use the memory, can access it if it gets the key

Using Shared Memories in code

To create, manage and remove shared memories, there is an API in C/C++

Typical steps involves in the process are

- One of the communicating processes, creates a shared memory …
- … which can be done via the `shmget()` function.
- The key for the shared memory may be generated by calling the `ftok()` function …
- … it is useful, because it turns a string (a char* to be precise) and a number into a key
- All communicating processes “attach” the memory to themselves via a call to `shmat()` …
- … The returned pointer of this call, is a pointer to the shared memory
- The pointer can be used to read from or write data to the shared memory
- All process should detach themselves from the memory, post usage by invoking `shmdt()`
- Finally, the creator or any other process, should mark the shared memory for deletion ...
- … by calling the `shmctl()` method

Creating processes from the code

We saw how a new Thread of execution can be created using the POSIX API

The major difference between a Thread and a Process is that Threads share data

- While it may seem a positive aspect, sometimes you may wish that two threads have nothing in common
- In such a case, it is better if we could create a new process instead of a new thread

The system call for this purpose is called `fork()`

- You can do so in your program by making a call to a function with the same name

A single call to `fork()` creates a single new process

- Multiple calls to `fork()` can create multiple new processes
- There is a parent-child relationship between the calling process and the newly created process

The new process is a replica of its parent …

- … i.e., its code segment has the same code as its parent, and the data segment has the same data too
- The only thing that differs between the two processes, is the returned value of the `fork()` call

Using the `fork()` system call

```
...
pid = fork();
if(pid < 0)
    cout<<"fork() failed";
else if(pid == 0)
    cout<<"This statement is executed only in the newly created process";
else
    cout<<"This statement is executed only in the parent process";
...
```

Using the `fork()` system call

```
...
pid = fork();
if(pid < 0)
    cout<<"fork() failed";
else if(pid == 0)
    cout<<"This statement is executed only in the newly created process";
else
    cout<<"This statement is executed only in the parent process";
...
```

A call to `fork()` creates a replica of the calling process

This includes the process's code and data, meaning that the child process will have the same variables (and they will have the values too) …

… with the only exception being the returned value of `fork()`, that is different in both processes

This value, can be used to create a bifurcation – between the executed code at the runtime

Using the `fork()` system call

Let us see some code now !!

```
...
pid = fork();
if(pid < 0)
    cout<<"fork() failed";
else if(pid == 0)
    cout<<"This statement is executed only in the newly created process";
else
    cout<<"This statement is executed only in the parent process";
...
```

A call to `fork()` creates a replica of the calling process

This includes the process's code and data, meaning that the child process will have the same variables (and they will have the values too) …

… with the only exception being the returned value of `fork()`, that is different in both processes

This value, can be used to create a bifurcation – between the executed code at the runtime

Homework !!

Try to use the Standard Input Stream, i.e., `cin` in the child processes created in the Family Tree example

- What do you observe? Find out the reason for your observations
- Maybe this Stackoverflow discussion can help you: https://stackoverflow.com/questions/61075110/stdcin-is-not-blocking-when-process-is-forked