CS 240 Programming in C

Process Control

Nov 1, 2022

Process

- Processes are the primitive units for allocation of system resources. Each process has its own address space and (usually) one thread of control. A process executes a program; you can have multiple processes executing the same program, but each process has its own copy of the program within its own address space and executes it independently of the other copies.
- Processes are organized hierarchically. Each process has a parent process which explicitly arranged to create it. The processes created by a given parent are called its child processes. A child inherits many of its attributes from the parent process.
- There are three distinct operations involved: creating a new child process, causing the new process to execute a program, and coordinating the completion of the child process with the original program.

Properties of Process

- Creation of each process requires separate system calls for each process.
- It is an isolated execution entity and does not share data and information.
- Processes use the IPC(Inter-Process Communication) mechanism for communication that significantly increases the number of system calls.
- Process management takes more system calls.
- A process has its stack, heap memory with memory, and data map.

Process Identifier

- In computing, the process identifier (a.k.a. process ID or PID) is a number used by most operating system kernels—such as those of Unix, macOS and Windows—to uniquely identify an active process.
- When a program is called, a process is created and a process ID is issued. The process ID is given by the function getpid() defined in <unistd.h>.

Unix commands

```
ps              List current process
kill            kill process
ps aux          list all processes
```

Background Process

A long running program can be controlled by introducing a process running in the background. For example, we can run a C program in the background by typing

Command

```
./program1 &
```

Background processes continues to run even when you logout. You can check the status of a background process by typing > ps at the login.

Thread

- Thread is an execution unit that is part of a process. A process can have multiple threads, all executing at the same time. It is a unit of execution in concurrent programming. A thread is lightweight and can be managed independently by a scheduler. It helps you to improve the application performance using parallelism.

Properties of Thread

- Single system call can create more than one thread
- Threads share data and information.
- Threads shares instruction, global, and heap regions. However, it has its register and stack.
- Thread management consumes very few, or no system calls because of communication between threads that can be achieved using shared memory.

Key Difference Between Process and Thread

- Process means a program is in execution, whereas thread means a segment of a process.
- A Process is not Lightweight, whereas Threads are Lightweight.
- A Process takes more time to terminate, and the thread takes less time to terminate.
- Process takes more time for creation, whereas Thread takes less time for creation.
- Process likely takes more time for context switching whereas as Threads takes less time for context switching.
- A Process is mostly isolated, whereas Threads share memory.
- Process does not share data, and Threads share data with each other.

Running a command

- The easy way to run another program is to use the system function. This function does all the work of running a subprogram, but it doesn't give you much control over the details: you have to wait until the subprogram terminates before you can do anything else.
- Function: int system (const char *command)
- If the command argument is a null pointer, a return value of zero indicates that no command processor is available.
- The system function is declared in the header file 'stdlib.h'.

Process Identification

The pid_t data type represents process IDs.

- The getpid function returns the process ID of the current process.
- The getppid function returns the process ID of the parent of the current process.

Your program should include the header files ‘unistd.h’ and ‘sys/types.h’ to use these functions.

Creating a Process

The fork function is the primitive for creating a process. It is declared in the header file 'unistd.h'.

- The child process has its own unique process ID.
- If the operation is successful, there are then both parent and child processes and both see fork return, but with different values: it returns a value of 0 in the child process and returns the child's process ID in the parent process.
- If process creation failed, fork returns a value of -1 in the parent process. The following errno error conditions are defined for fork:
  - EAGAIN - There aren't enough system resources to create another process, or the user already has too many processes running.
  - ENOMEM - The process requires more space than the system can supply.

Executing a File

For executing a file as a process image, **exec** family of functions can be used. These functions make a child process, that executes a new program after it has been forked. It is declared in the header file 'unistd.h'.

- The exec call replaces the entire current contents of the process with a new program. It loads the program into the current process space and runs it from the entry point.

```
char *args[] = {"ps", "-a", NULL};
execvp("ps", args);
```

Exec functions

- execl: execute a program
- execlp: execute a program in the path
- execle: execute a program with environment
- execv: execute a program with arguments
- execvp: execute a program with arguments in the path
- execvpe: execute a program with arguments and environment in the path

Process completion

Function

```
pid_t waitpid (pid_t pid, int *status-ptr, int options)
```

The waitpid function is used to request status information from a child process whose process ID is pid. Normally, the calling process is suspended until the child process makes status information available by terminating.

- Other values for the pid argument have special interpretations. A value of -1 or WAIT_ANY requests status information for any child process; a value of 0 or WAIT_MYPGRP requests information for any child process in the same process group as the calling process;
- If status information for a child process is available immediately, this function returns immediately without waiting. If more than one eligible child process has status information available, one of them is chosen randomly, and its status is returned immediately.

Process related functions

- system()
- fork()
- exec functions
- waitpid()
- getpid()
- getppid()
- exit()

Inter Process Communication

A process can be of two types:

- Independent process.
- Co-operating process.

There are many situations when co-operative nature can be utilized for increasing computational speed, convenience, and modularity.
Inter-process communication (IPC) is a mechanism that allows processes to communicate with each other and synchronize their actions. Processes can communicate with each other through:

- Shared Memory
- Message passing
- Sockets
- Pipes / Named Pipes
- Semaphores

Inter Process Communication

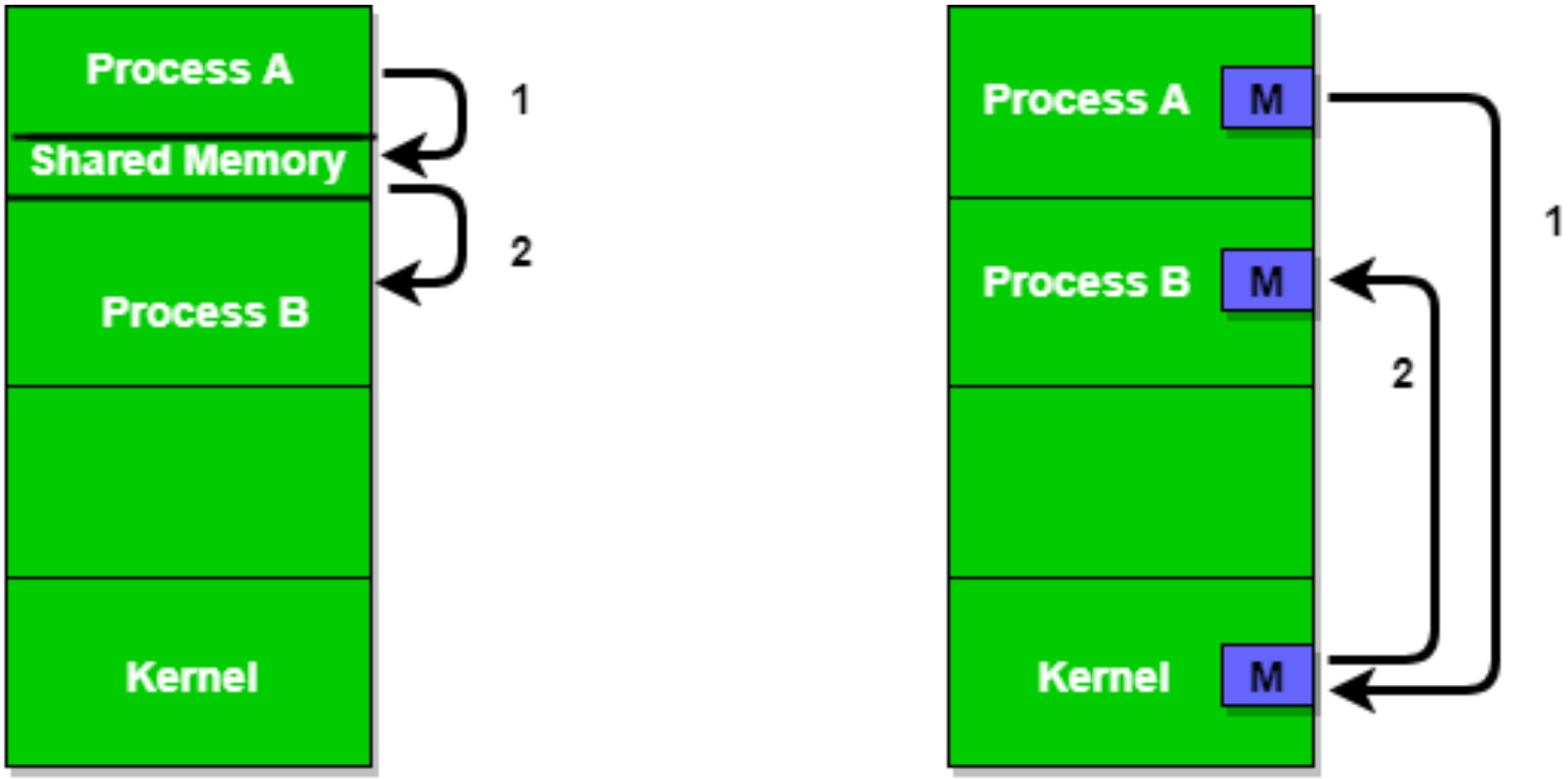

Figure 1 - Shared Memory and Message Passing

Shared Memory

The problem with pipes, fifo and message queue – is that for two process to exchange information. The information has to go through the kernel.

- Server reads from the input file.
- The server writes this data in a message using either a pipe, fifo or message queue.
- The client reads the data from the IPC channel,again requiring the data to be copied from kernel's IPC buffer to the client's buffer.
- Finally the data is copied from the client's buffer.

Shared Memory

- ftok(): is use to generate a unique key.
- shmget(): upon successful completion, shmget() returns an identifier for the shared memory segment.
- shmat(): Before you can use a shared memory segment, you have to attach the process to it using shmat().
- shmdt(): When you're done with the shared memory segment, your program should detach itself from it using shmdt().
- shmctl(): when you detach from shared memory, it is not destroyed. So, to destroy shmctl() is used.

Shared Memory: Sample Program

Let us consider the following sample program.

- Create two processes, one is for writing into the shared memory (shm_write.c) and another is for reading from the shared memory (shm_read.c)
- The program performs writing into the shared memory by write process (shm_write.c) and reading from the shared memory by reading process (shm_read.c)
- In the shared memory, the writing process, creates a shared memory of size 1K (and flags) and attaches the shared memory
- The write process writes 5 times the Alphabets from 'A' to 'E' each of 1023 bytes into the shared memory. Last byte signifies the end of buffer

Shared Memory: Sample Program

- Read process would read from the shared memory and write to the standard output.
- Reading and writing process actions are performed simultaneously
- After completion of writing, the write process updates to indicate completion of writing into the shared memory (with complete variable in struct shmseg)
- Reading process performs reading from the shared memory and displays on the output until it gets indication of write process completion (complete variable in struct shmseg)
- Performs reading and writing process for a few times for simplication and also in order to avoid infinite loops and complicating the program

Demo

Sample Programs

Resource Links

- https://ftp.gnu.org/old-gnu/Manuals/glibc-2.2.3/html_chapter/libc_26.html
- https://www.guru99.com/difference-between-process-and-thread.html
- https://www.geeksforgeeks.org/inter-process-communication-ipc/