Reading Material

Command Line arguments:

Command-line arguments are a way to pass data to the program. Command-line arguments are passed to the main function. Suppose we want to pass two integer numbers to the main function of an executable program called a.out. On the terminal write the following line:

```
./a.out 1 22
```

./a.out is the usual method of running an executable via the terminal. Here 1 and 22 are the numbers we passed as command-line arguments to the program. These arguments are passed to the primary function. In order for the main function to be able to accept the arguments, we have to change the signature of the primary function as follows:

```
int main(int argc, char *arg[]);
```

argc is the counter. It tells how many arguments have been passed. arg is the character pointer to our arguments. argc in this case will not be equal to 2, but it will be equal to 3. This is because the name ./a.out is also passed as command line argument. At index 0 of arg we have ./a.out; at index 1 we have 1; and at index 2 we have 22. Here 1 and 22 are in the form of character string, we have to convert them to integers by using a function atoi.

Suppose we want to add the passed numbers and print the sum on the screen:

```
cout << atoi(arg[1]) + atoi(arg[2]);
```

Fork() system call:

The fork() system call is used in Unix-like operating systems to create a new process. It is an essential function in process management, and it allows the creation of a child process. Here's a breakdown of how fork() works and its key details:

1.
    a. fork() creates a new process by duplicating the calling (parent) process.
    b. The new process created is called the child process.
    c. The child process gets a copy of the parent process's memory, file descriptors, and other resources.
2. Return value of fork():
    a. fork() returns a positive value to the parent process. This value is the PID (Process ID) of the newly created child.
    b. fork() returns zero (0) to the child process.
    c. If fork() fails (due to system limits), it returns -1 to both the parent and child.

3. Working:
    a. The parent and child processes run concurrently after the fork().
    b. Both processes will execute the code that follows the fork() call, but they will have different return values for fork().
    c. Memory and resources are copied, but modifications in one process (parent or child) do not affect the other, as memory is duplicated.
4. Waiting for child process:
    a. The parent process can wait for the child process to finish using wait() or waitpid(). This helps avoid zombie processes (terminated children that still occupy process table slots).
    b. The child process can exit with an exit status using the exit() system call. The parent can collect this status.
    c. Wait and return example
    d. if (pid > 0)

```
{
 int status;
if ( waitpid(pid, &status, 0) == -1 )
{
perror("waitpid() failed");
exit(EXIT_FAILURE);
}

 if ( WIFEXITED(status) ) {
int es = WEXITSTATUS(status);
 printf("Exit status was %d\n", es);
 }
 }
```

5. Example Code:
    a.

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

int main() {
    pid_t pid;

    pid = fork();  // Create a new child process

    if (pid == -1) {
        perror("Fork failed");  // Handle error if fork() fails
        return 1;
    }
    if (pid == 0) {
              printf("Child process, PID = %d\n", getpid());
    } else {
       printf("Parent process, PID = %d, Child PID = %d\n",
getpid(), pid);
    }

    return 0;
}
```

File descriptors:

When you open a file, the operating system creates an entry to represent that file and store the information about that opened file. So if there are 100 files opened in your OS then there will be 100 entries in the OS (somewhere in the kernel). These entries are represented by integers like (...100, 101, 102....). This entry number is the file descriptor. It is just an integer number that uniquely represents an opened file for the process. If your process opens 10 files then your Process table will have 10 entries for file descriptors.

Lets understand this with an example:

```
#include<stdio.h>
#include<conio.h>
main()
{
  FILE *fp;
  char ch;
  fp = fopen("hello.txt", "w");
  printf("Enter data");
  while( (ch = getchar()) != EOF) {
   putc(ch,fp);
  }
  fclose(fp);
  fp = fopen("hello.txt", "r");

  while( (ch = getc(fp)! = EOF)
    printf("%c",ch);

  fclose(fp);
}
```

Code explanation with respect to file descriptors:

1. fp = fopen("hello.txt", "w");
    a. `fopen` opens the file `hello.txt` for writing (`"w"` mode).
    b. internally, the operating system assigns a file descriptor to this file and associates it with the `FILE` pointer `fp`.

```
 2. while ((ch = getchar()) != EOF) {
        putc(ch, fp);
    }
```

    a. The file descriptor associated with `fp` is used to manage the write operations.

3. fclose(fp);
    a. `fclose` closes the file, releasing the file descriptor and flushing any buffered output.
4. fp = fopen("hello.txt", "r"); // opening the file for reading
    a. `fopen` opens the file `hello.txt` for reading (`"r"` mode).
    b. A new file descriptor is assigned to manage the read operations.
5. fclose(fp);
    a. `fclose` closes the file, releasing the file descriptor.

Note:
Here File pointer is used for file descriptor which works with fopen and fclose, if you want to use int fd you have to use fcntl library which provides open and close functions file opening and closing files.

Exec system call and exec family:

The exec family of functions replaces the current running process image with a new process image. It can be used to run a C program by using another C program. It comes under the header file **unistd.h**. They do not create a new process but rather replace the current process with a new one.
exec type system calls allow a process to run any program files, which include a binary executable or a shell script. Many members of the exec family are shown below with examples. They differ in how the program and arguments are provided.

1. **execl**: Executes a program, passing the arguments as a list of parameters.

```
int execl(const char *path, const char *arg, ...);
```

Example:

```
execl("/bin/ls", "ls", "-l", NULL);
```

This call executes the `ls -l` command.

2. **execlp**: Similar to execl, but searches for the program in the directories listed in the PATH environment variable.

```
int execlp(const char *file, const char *arg, ...);
```

Example:

```
execlp("ls", "ls", "-l", NULL);
```

Here, `ls` is found in one of the directories listed in `PATH`.

3. **execle**: Similar to execl, but also allows specifying the environment for the new program.

```
int execle(const char *path, const char *arg, ..., char *const envp[]);
```

example:

```
char *env[] = {"HOME=/usr/home", "PATH=/bin:/usr/bin", NULL};
execle("/bin/ls", "ls", "-l", NULL, env);
```

4. **execv**: Executes a program, passing the arguments as an array of strings. Takes the path of the executable and an array of arguments (where the last element is `NULL`).

```
int execv(const char *path, char *const argv[]);
```

Example:

```
char *args[] = {"ls", "-l", NULL};
execv("/bin/ls", args);
```

5. **execve**:Similar to `execv`, but it allows specifying both the argument array and the environment array.

```
int execve(const char *path, char *const argv[], char *const envp[]);
```

Example:

```
char *args[] = {"ls", "-l", NULL};
char *env[] = {"HOME=/usr/home", "PATH=/bin:/usr/bin", NULL};
execve("/bin/ls", args, env);
```

6. **execvp:** Similar to `execv`, but searches for the program in the directories listed in the `PATH` environment variable.

```
int execvp(const char *file, char *const argv[]);
```

Example:

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

Read write permissions:

In Linux systems, file permissions are a fundamental aspect of security and access control. They determine who can read, write, or execute a file or directory. Here's a explanation of read and write permissions:

Each file and directory in Linux has three types of permissions for three categories of users:

1. Owner (User): The user who owns the file.
2. Group: A group of users who share the same permissions.
3. Others: All other users on the system.

Types of Permissions:

1. Read (r):
   - Files: Allows viewing or copying the file's contents.
   - Directories: Allows listing the contents of the directory.
2. Write (w):
   - Files: Allows modifying the file's contents.
   - Directories: Allows adding, removing, or renaming files within the directory.
3. Execute (x):
   - Files: Allows executing the file if it is a script or a binary.
   - Directories: Allows entering the directory and accessing its contents.

You can view the permissions of a file or directory using the ls -l command:

```
$ ls -l
-rw-r--r-- 1 user group 1234 Sep 12 12:00 example.txt
```

In this example:

- -rw-r--r-- represents the permissions.
  - r (read), w (write), and x (execute) are the permissions.
  - The first character indicates the file type (- for a regular file, d for a directory).
  - The next three characters (rw-) are the owner's permissions.
  - The next three characters (r--) are the group's permissions.
  - The last three characters (r--) are the permissions for others.

Changing the permissions:
You can change the permissions of a file or directory using the chmod command. There are two ways to specify permissions: symbolic mode and numeric mode.

```
$ chmod u+rwx,g+rx,o+r example.txt
```

- u (user/owner), g (group), o (others), and a (all).
- + (add), - (remove), = (set exactly).

<u>Numeric mode:</u>
Permissions can also be represented by numbers:

- Read (r) = 4
- Write (w) = 2
- Execute (x) = 1

Combine these values to set permissions:

```
$ chmod 755 example.txt
```

755 means:

  - Owner: 7 (read, write, execute: 4+2+1)
  - Group: 5 (read, execute: 4+1)
  - Others: 5 (read, execute: 4+1)

Inter process communication:

Pipes:

Libraries:
#include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

Ordinary pipes allow two processes to communicate in standard producer consumer fashion: the producer writes to one end of the pipe (the write-end) and the consumer reads from the other end (the read-end). As a result, ordinary pipes are unidirectional, allowing only one-way communication. If two-way communication is required, two pipes must be used with each pipe sending data in a different direction.
References: Operating System concepts Page no 142 section 3.6.3.1

A pipe has a read end and a write end. Data written to the write end of a pipe can be read from the read end of the pipe.

Creation: Pipes are created using system calls. In UNIX-like systems, the pipe() system call is used.

```
int fds[2];
pipe(fds);
```

- fds[0]: File descriptor for the read end of the pipe.
- fds[1]: File descriptor for the write end of the pipe.
- Returns 0 on success and -1 on error.

Communication:

- Writing to the Pipe: One process writes data to the write end of the pipe.
- Reading from the Pipe: Another process reads data from the read end of the pipe.
- Synchronization: If a process tries to read from an empty pipe, it will be suspended until data is written to the pipe.

Example:

```
#include <string.h>

int main() {
    int fds[2];
    char buffer[100];
    pipe(fds);

    if (fork() == 0) {
        close(fds[0]); // Close read end
        char msg[] = "Data from child";
        write(fds[1], msg, strlen(msg) + 1);
        close(fds[1]); // Close write end
    } else {  // Parent process
        wait(NULL);
        close(fds[1]); // Close write end
        read(fds[0], buffer, sizeof(buffer));
        printf("Received: %s\n", buffer);
        close(fds[0]); // Close read end
    }

    return 0;
}
```

FIFO:

A named pipe, also known as a FIFO (First In, First Out), is a special type of file used for inter-process communication (IPC). Unlike unnamed pipes, which are temporary and exist only as long as the process that created them is running, named pipes persist in the filesystem and can be accessed by multiple processes for reading and writing.

Creating the Named Pipe: The mkfifo() function creates a named pipe with the specified path and permissions.

Example:

```
#include <sys/stat.h>
#include <string.h>

#define FIFO_PATH "/tmp/myfifo"

void writer_process() {
    int fd;
    char message[] = "named pipe!";

    // Open the named pipe for writing
    fd = open(FIFO_PATH, O_WRONLY);
    if (fd == -1) {
        perror("open");
        exit(-1);
    }

    // Write the message to the named pipe
    write(fd, message, strlen(message) + 1);
    printf("Message sent: %s\n", message);
    close(fd);
}

int main() {
    // Create the named pipe
    mkfifo(FIFO_PATH, 0666);
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork");
        exit(-1);
    } else if (pid == 0) {
        writer_process();
    } else {
        wait(NULL);
        int fd;
        char buffer[100];
        // Open the named pipe for reading
        fd = open(FIFO_PATH, O_RDONLY);
        if (fd == -1) {
            perror("open");
            exit(-1);
        }

        // Read the message from the named pipe
        read(fd, buffer, sizeof(buffer));
        printf("Message received: %s\n", buffer);
        close(fd);
        // Remove the named pipe
        unlink(FIFO_PATH);
    }

    return 0;
}
```

Dup system call:

The dup system call in Unix-like operating systems is used to duplicate a file descriptor. This means it creates a new file descriptor that refers to the same open file description as the original one. The new file descriptor is the lowest-numbered available descriptor.

```
int new_fd = dup(old_fd);
```

- oldfd: The file descriptor to be duplicated.
- Returns: The new file descriptor on success, or -1 on error.

Dup2 system call:

The dup2 system call duplicates the file descriptor oldfd to newfd, allowing you to specify a particular file descriptor number for the duplication. If newfd is already in use, it is closed before the duplication occurs.
This is particularly useful for redirecting standard input/output or managing file descriptors in a controlled manner.

```
int dup2(int oldfd, int newfd);
```

- oldfd: The file descriptor to be duplicated.
- newfd: The file descriptor to which oldfd should be duplicated.
- Returns: The new file descriptor on success, or -1 on error.

```
int file_desc = open("file.txt", O_WRONLY | O_APPEND);
dup2(file_desc, STDOUT_FILENO); // Redirects stdout to file.txt
```

printf("%d\n", someData); // Write the data to stdout (redirected to output.txt)

Grep command:

grep is a powerful command-line utility in Unix/Linux systems used for searching text using patterns.

```
grep [options] pattern [file...]
```

- pattern: The regular expression or string to search for.
- file: The file(s) to search within. If no file is specified, `grep` searches the standard input.

Common Options:

- `-i`: Ignore case distinctions.
- `-v`: Invert the match, showing lines that do not match the pattern.
- `-c`: Count the number of matching lines.
- `-n`: Show line numbers with output lines.
- `-r`: Recursively search directories.

grep "hello" file.txt // This command searches for the string “hello” in `file.txt`.
grep -i "hello" file.txt // This command counts the number of lines containing “hello” in file.txt.
grep -r "hello" /path/to/directory //This command searches for “hello” in all files within the specified directory and its subdirectories.

```
grep -in "error" server.log
```

pipe (|) command is a powerful feature that allows you to connect the output of one command directly to the input of another command. This enables you to chain commands together, creating complex workflows from simple commands.

Syntax: command1 | command2

```
ls | grep "txt"
```

This command lists the contents of the current directory (ls) and then filters the output to show only lines containing the string “txt” using the grep command.

```
cat file.txt | sort | uniq
```

This command reads the contents of file.txt, sorts the lines, and then removes duplicate lines using the uniq command.

Threads:

https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html
Read the first example in the above given link.

Types of Thread
Threads are implemented in following two ways −

- User Level Threads − User managed threads.
- Kernel Level Threads − Operating System managed threads acting on kernel, an operating system core.

1) Pthread_create
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t
*attr, void *(*start_routine) (void *), void *arg);

Compile and link with -pthread.
The pthread_create() function starts a new thread in the calling process. The new thread starts execution by invoking start_routine(); arg is passed as the sole argument of start_routine().

On success, pthread_create() returns 0; on error, it returns an
error number, and the contents of *thread are undefined.

2) Pthread_join
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);

Compile and link with -pthread.

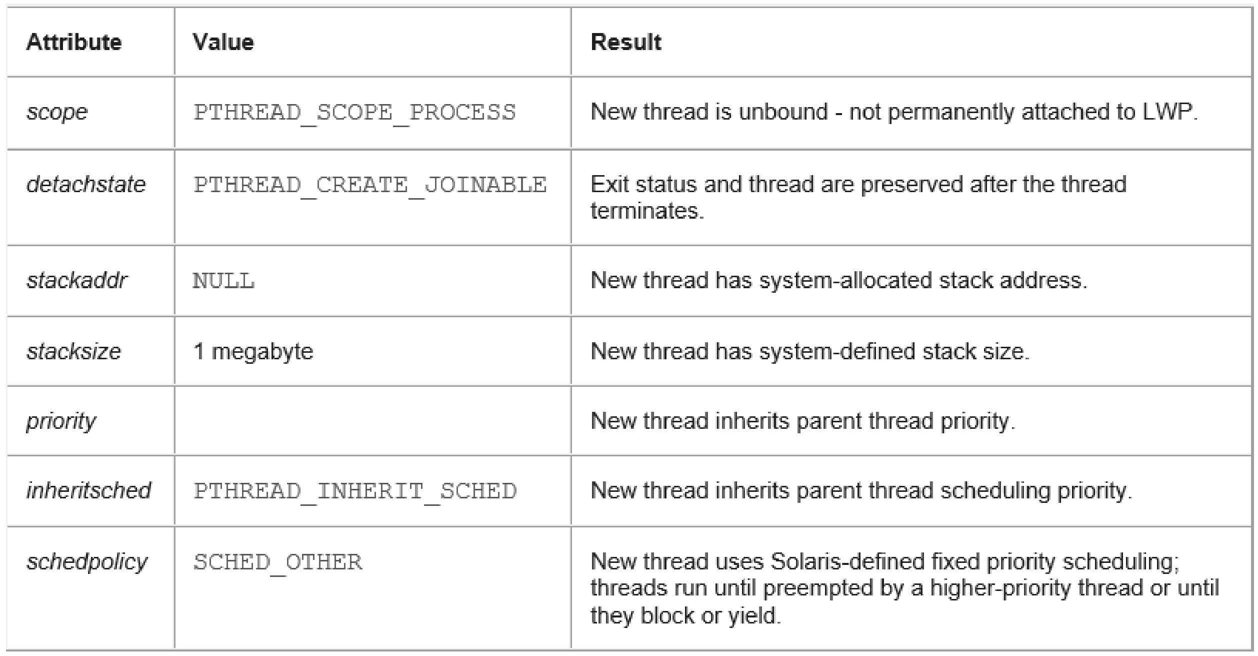
The pthread_join() function waits for the thread specified by thread to terminate. If that thread has already terminated, then pthread_join() returns immediately. The thread specified by thread must be joinable.

3) Pthread_exit
#include <pthread.h>

void pthread_exit(void *retval);
Compile and link with -pthread.

The pthread_exit() function terminates the calling thread and returns a value via retval that (if the thread is joinable) is available to another thread in the same process that calls **pthread_join(3)** This call always succeeds and does not return anything in the calling process.

Command to see thread ID:
cd /proc/pid
ls task
Every thread has a set of attributes.
Default set of attributes: Table 3-1 Default Attribute Values



Attribute	Value	Result
scope	`PTHREAD_SCOPE_PROCESS`	New thread is unbound - not permanently attached to LWP.
detachstate	`PTHREAD_CREATE_JOINABLE`	Exit status and thread are preserved after the thread terminates.
stackaddr	`NULL`	New thread has system-allocated stack address.
stacksize	1 megabyte	New thread has system-defined stack size.
priority		New thread inherits parent thread priority.
inheritsched	`PTHREAD_INHERIT_SCHED`	New thread inherits parent thread scheduling priority.
schedpolicy	`SCHED_OTHER`	New thread uses Solaris-defined fixed priority scheduling; threads run until preempted by a higher-priority thread or until they block or yield.

Race Condition in multithreading:

Race conditions occur when two computer program processes, or threads, attempt to access the same resource at the same time and cause problems in the system. Race conditions are considered a common issue for multithreaded applications.
The situation where processes using shared memory do their own work at the wrong time and the data is corrupted is called a race condition. The main purpose is to ensure that the processes work in the correct order. For example, which process will be run first, which one will

take the value there and how it will change are the best examples of race conditions. Race, as the word suggests, is the competition of processes with each other. It is entirely our responsibility which process will run at what time.

Shared Memory:

Shared memory is a memory management technique used in operating systems that allows multiple processes to access a common memory space. This shared memory region enables fast data communication between processes since they can directly read and write to this shared area without needing to use slower inter-process communication (IPC) methods like pipes or message queues.

How It Works

- When a process wants to share data, it requests a block of memory from the operating system to be designated as "shared."
- This shared block is then available to other processes, which can also request to connect to this block.
- Once connected, processes can read from and write to this memory block as if it's a part of their own memory space.

Creating Shared Memory

- In most operating systems, shared memory is created using system functions. For example, in UNIX-based systems, the function `shmget()` creates a shared memory segment.
- This segment gets a unique ID, kind of like a Wi-Fi password that's shared among all the processes that need access.

Connecting to Shared Memory

- Each process that wants to access this shared space must "attach" to it. In UNIX, they do this with `shmat()`, which means "shared memory attach."
- When a process attaches to the shared memory, the OS connects this shared area to the process's memory, letting it use the shared memory like any other part of its memory.

Reading and Writing to Shared Memory

- Once connected, each process can directly read from and write to this memory.
- For instance, one process can write data to this memory space, and the other can immediately read it—just like one person writing on a whiteboard and another reading from it right away.

The Need for Synchronization

- Because multiple processes can access shared memory at once, they can sometimes get in each other's way if they try to read or write at the same time. This is called a *race condition*.
- **Example**: Imagine two people trying to write on the same part of a whiteboard at the same time; their messages would get mixed up.
- To prevent this, processes use synchronization tools like **mutexes** (locks) or **semaphores** to coordinate access.

Removing Shared Memory

- When processes are done using the shared memory, they detach from it, similar to disconnecting from a network.
- Finally, the operating system can remove the shared memory block, freeing up the memory for other uses.

Real-Life Example of Shared Memory

- **Database Systems**: In database applications, many processes may need access to the same data, like user records. Instead of copying data between them, they can share a memory area that holds this data. This speeds up database transactions since all processes can look at the same data in real-time.
- **Graphics Rendering**: In multimedia applications, such as video games or 3D rendering software, shared memory allows fast communication between processes that handle different parts of the graphics pipeline. This is crucial for real-time performance.

Functions

1. ftok

The ftok function in UNIX-like operating systems is used to generate a unique key (identifier) that multiple processes can use to access the same shared memory segment.

```
key_t key = ftok(char *filename, 0);
```

2. shmget

```
int id = shmget(key_t key, int size, int flags);
```

// int shmid = shmget(key, 1024, 0666 | IPC_CREAT);

- Allocates a shared memory segment.
- Key is the key associated with the shared memory segment you want.
- Size is the size in bytes of the shared memory segment you want allocated.
- Memory is allocated in pages, so chances are you will probably get a little more memory than you wanted.

- Flags indicate how you want the segment created and its access permissions.
- The general rule is just to use 0666 | IPC_CREAT | IPC_EXCL if the caller is making a new segment.
- If the caller wants to use an existing share region, simply pass 0 in the flag.

<u>RETURN VALUES</u>
- Upon successful completion, shmget() returns the positive integer identifier of a shared memory
- segment.
- Otherwise, -1 is returned

<u>Shmget will fail if:</u>
1. Size specified is greater than the size of the previously existing segment. Size specified is less than the system imposed minimum, or greater than the system imposed maximum.
2. No shared memory segment was found matching the key, and IPC_CREAT was not specified.
3. The kernel was unable to allocate enough memory to satisfy the request.
4. IPC_CREAT and IPC_EXCL were specified, and a shared memory segment corresponding to the key already exists.

3. shmat

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- Maps a shared memory segment onto your process's address space.
- shmid is the id as returned by shmget() of the shared memory segment you wish to attach.
- Addr is the address where you want to attach the shared memory. For simplicity, we will pass NULL.
- NULL means that the kernel itself will decide where to attach it to address space of the process.

<u>RETURN VALUES</u>
- Upon success, shmat() returns the address where the segment is attached;
- Otherwise, -1 is returned and errno is set to indicate the error.

<u>Shmat() will fail if:</u>
1. No shared memory segment was found corresponding to the given id.

4. shmdt

```
int shmdt(void *addr);
```

- This system call is used to detach a shared memory region from the process's address space.
- Addr is the address of the shared memory

<u>RETURN VALUES</u>

On success, shmdt() returns 0; on error -1 is returned shmdt will fail if:

1. The address passed to it does not correspond to a shared region.

5. shmctl

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- Delete shared memory region
- shmctl(shmid, IPC_RMID, NULL);
- shmctl() performs the control operation specified by cmd on the System V shared memory segment whose identifier is given in shmid.
- For Deletion, we will use IPC_RMID flag.
- IPC_RMID marks the segment to be destroyed.
- The segment will actually be destroyed only after the last process detaches it (The caller must be the owner or creator of the segment, or be privileged).
- The buf argument is ignored.

Return Value:
For IPC_RMID operation, 0 is returned on success; else -1 is returned.

Example:

Process 1:

- Creates a shared memory area.
- Writes text (passed as a command-line argument) to the shared memory.
- Waits for 10 seconds to allow Process 2 to read the data.
- Unlinks and deletes the shared memory area after 10 seconds.

Process 2:

- Attaches to the shared memory area created by Process 1.
- Reads the text from shared memory and prints it.
- Unlinks itself from the shared memory.

Process1:

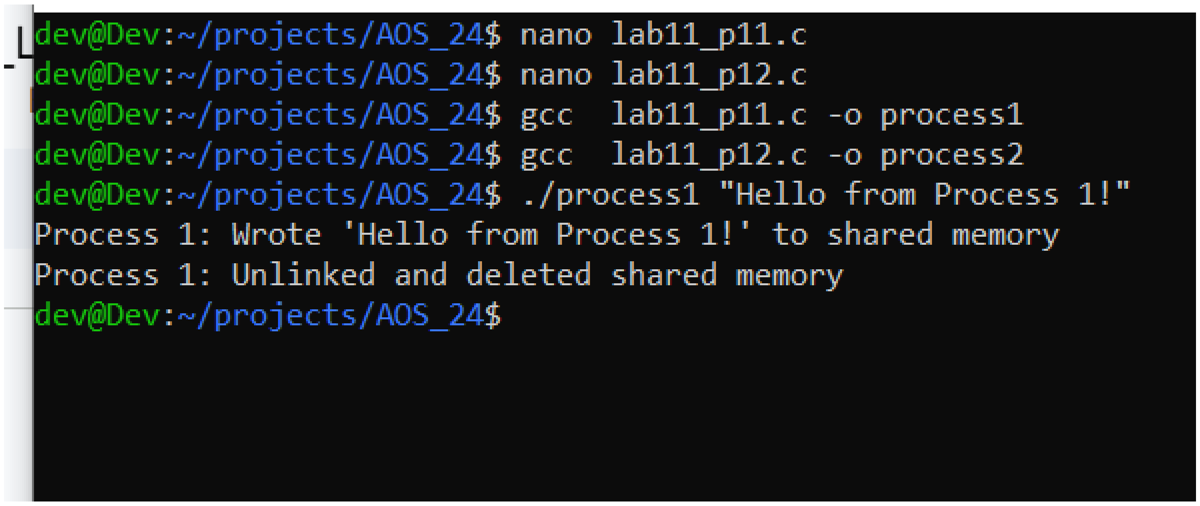
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024 //Size of shared memory, value can be directly passed
```

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <text_to_send>\n", argv[0]);
        exit(1);
    }

    // Generate a unique key for shared memory
    key_t key = ftok("shmfile", 65);

    // Create a shared memory segment
    int shmid = shmget(key, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach to the shared memory
    char *data = (char *) shmat(shmid, NULL, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    // Write text to shared memory
    strncpy(data, argv[1], SHM_SIZE);
    printf("Process 1: Wrote '%s' to shared memory\n", data);

    // Wait for 10 seconds before unlinking and deleting shared memory
    sleep(10);

    // Detach from shared memory
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    // Delete the shared memory segment
    if (shmctl(shmid, IPC_RMID, NULL) == -1) {
        perror("shmctl");
        exit(1);
    }
    printf("Process 1: Unlinked and deleted shared memory\n");

    return 0;
}
```

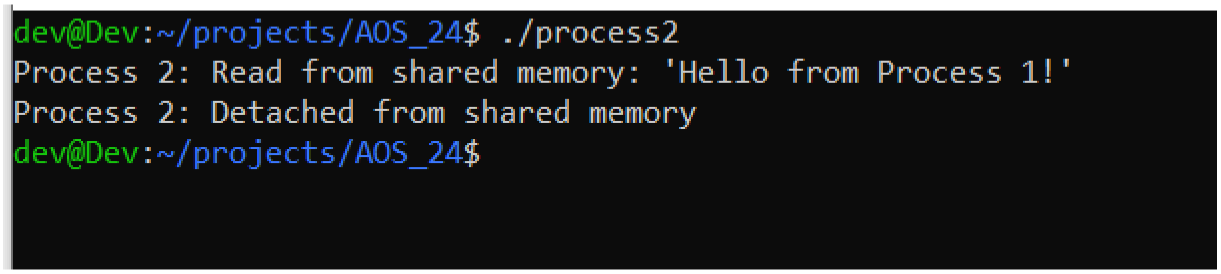
Process2:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  // Size of shared memory

int main() {
    // Generate a unique key for shared memory (must match Process 1)
    key_t key = ftok("shmfile", 65);

    // Access the shared memory segment
    int shmid = shmget(key, SHM_SIZE, 0666);
    if (shmid == -1) {
        perror("shmget");
        exit(1);
    }

    // Attach to the shared memory
    char *data = (char *) shmat(shmid, NULL, 0);
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }

    // Read and print the text from shared memory
    printf("Process 2: Read from shared memory: '%s'\n", data);

    // Detach from the shared memory
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }

    printf("Process 2: Detached from shared memory\n");
    return 0;
}
```

Output:



```
dev@Dev:~/projects/AOS_24$ nano lab11_p11.c
dev@Dev:~/projects/AOS_24$ nano lab11_p12.c
dev@Dev:~/projects/AOS_24$ gcc  lab11_p11.c -o process1
dev@Dev:~/projects/AOS_24$ gcc  lab11_p12.c -o process2
dev@Dev:~/projects/AOS_24$ ./process1 "Hello from Process 1!"
Process 1: Wrote 'Hello from Process 1!' to shared memory
Process 1: Unlinked and deleted shared memory
dev@Dev:~/projects/AOS_24$
```



```
dev@Dev:~/projects/AOS_24$ ./process2
Process 2: Read from shared memory: 'Hello from Process 1!'
Process 2: Detached from shared memory
dev@Dev:~/projects/AOS_24$
```

Semaphores and Mutexes:

Mutexes and semaphores are synchronization primitives used in concurrent programming to manage access to shared resources. They help prevent race conditions and ensure proper execution order among threads.

Thread synchronization:

Mutex:

A **mutex** (short for "mutual exclusion") is a lock that allows only one thread to access a shared resource at a time. It is commonly used to protect critical sections in multi-threaded applications.

You need to include the pthread library.

Steps to Use a Mutex

- Declare a pthread_mutex_t variable.
- Initialize the mutex using pthread_mutex_init().
- Lock the mutex using pthread_mutex_lock().
- Unlock the mutex using pthread_mutex_unlock().
- Destroy the mutex using pthread_mutex_destroy() when it's no longer needed.

Example:
Two threads (t1 and t2) increment a shared counter.
The mutex ensures that only one thread modifies the counter at a time.

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

pthread_mutex_t lock;
int counter = 0;

void* increment_counter(void* arg) {
    pthread_mutex_lock(&lock); // Lock the mutex
    counter++;
    printf("Counter: %d\n", counter);
    pthread_mutex_unlock(&lock); // Unlock the mutex
    return NULL;
}

int main() {
    pthread_t t1, t2;

    pthread_mutex_init(&lock, NULL); // Initialize the mutex

    pthread_create(&t1, NULL, increment_counter, NULL);
    pthread_create(&t2, NULL, increment_counter, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    pthread_mutex_destroy(&lock); // Destroy the mutex
    return 0;
}
```

Semaphore:

A **semaphore** is a synchronization mechanism that allows a fixed number of threads to access a shared resource simultaneously.

- **Binary Semaphore**: Works like a mutex (only allows one thread at a time).
- **Counting Semaphore**: Allows multiple threads (up to a specified limit) to access a resource.

Manual for semaphore => https://man7.org/linux/man-pages/man3/sem_init.3.html

How to Use a Semaphore

You need to include semaphore.h and pthread.h.
Steps to Use a Semaphore

- Declare a sem_t variable.
- Initialize it using sem_init().
- Decrement (wait) using sem_wait().
- Increment (signal) using sem_post().
- Destroy using sem_destroy() when finished.

Example:

- The semaphore is initialized with 2, so at most two threads can be inside the critical section at the same time.
- sem_wait() ensures that only available slots are used.
- sem_post() releases the slot for another thread.

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

sem_t sem;

void* worker(void* arg) {
    sem_wait(&sem); // Wait (decrement the semaphore)
    printf("Thread %ld is in critical section\n", (long)arg);
    sleep(1);
    printf("Thread %ld is leaving critical section\n", (long)arg);
    sem_post(&sem); // Signal (increment the semaphore)
    return NULL;
}

int main() {
    pthread_t threads[5];

    sem_init(&sem, 0, 2); // Initialize semaphore to allow 2 threads at a
time

    for (long i = 0; i < 5; i++) {
        pthread_create(&threads[i], NULL, worker, (void*)i);
    }

    for (int i = 0; i < 5; i++) {
        pthread_join(threads[i], NULL);
    }

    sem_destroy(&sem); // Destroy semaphore
    return 0;
}
```

Process Synchronization:

Named semaphores:

- Are system-wide semaphores visible by a name (like files).
- Can be accessed by unrelated processes (not just parent-child).
- Created using sem_open("/name", flags, mode, initial_value).
- Managed by the kernel and persist (briefly) until explicitly removed with sem_unlink.

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

#define SEM_PING "/sem_ping"
#define SEM_PONG "/sem_pong"
#define ITERATIONS 20
```

```c
int main() {
    // Create two named semaphores
    sem_t *sem_ping = sem_open(SEM_PING, O_CREAT | O_EXCL, 0666, 1); //
ping starts first
    sem_t *sem_pong = sem_open(SEM_PONG, O_CREAT | O_EXCL, 0666, 0); //
pong waits

    // Clean up if semaphores already existed
    if (sem_ping == SEM_FAILED || sem_pong == SEM_FAILED) {
        sem_unlink(SEM_PING);
        sem_unlink(SEM_PONG);
        sem_ping = sem_open(SEM_PING, O_CREAT | O_EXCL, 0666, 1);
        sem_pong = sem_open(SEM_PONG, O_CREAT | O_EXCL, 0666, 0);
    }

    pid_t ping_pid = fork();
    if (ping_pid == 0) {
        for (int i = 0; i < ITERATIONS; i++) {
            sem_wait(sem_ping);
            printf("ping\n");
            fflush(stdout);
            sem_post(sem_pong);
        }
        exit(0);
    }

    pid_t pong_pid = fork();
    if (pong_pid == 0) {
        for (int i = 0; i < ITERATIONS; i++) {
            sem_wait(sem_pong);
            printf("pong\n");
            fflush(stdout);
            sem_post(sem_ping);
        }
        exit(0);
    }

    waitpid(ping_pid, NULL, 0);
    waitpid(pong_pid, NULL, 0);

    sem_close(sem_ping);
    sem_close(sem_pong);
    sem_unlink(SEM_PING);
    sem_unlink(SEM_PONG);

    return 0;
}
```

Sem_int with mmap

sem_init with mmap() to create an unnamed semaphore in shared memory

- Uses fork() to create two child processes.
- Synchronizes them via an unnamed semaphore stored in shared memory.
- Alternates printing "ping" and "pong" exactly 20 times each.
- No named semaphores, no shared memory file descriptors — **all anonymous in memory**

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

#define ITERATIONS 20

int main() {
    // Create unnamed semaphore in shared memory
    sem_t *sem_ping = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
                           MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    sem_t *sem_pong = mmap(NULL, sizeof(sem_t), PROT_READ | PROT_WRITE,
                           MAP_SHARED | MAP_ANONYMOUS, -1, 0);

    if (sem_ping == MAP_FAILED || sem_pong == MAP_FAILED) {
        perror("mmap");
        exit(1);
    }

    // Initialize semaphores: ping starts with 1, pong waits with 0
    sem_init(sem_ping, 1, 1); // 1 = shared between processes
    sem_init(sem_pong, 1, 0);

    pid_t ping_pid = fork();
    if (ping_pid == 0) {
        for (int i = 0; i < ITERATIONS; i++) {
            sem_wait(sem_ping);
            printf("ping\n");
            fflush(stdout);
            sem_post(sem_pong);
        }
        exit(0);
    }

    pid_t pong_pid = fork();
    if (pong_pid == 0) {
        for (int i = 0; i < ITERATIONS; i++) {
            sem_wait(sem_pong);
            printf("pong\n");
            fflush(stdout);
            sem_post(sem_ping);
        }
        exit(0);
    }

    waitpid(ping_pid, NULL, 0);
    waitpid(pong_pid, NULL, 0);

    sem_destroy(sem_ping);
    sem_destroy(sem_pong);
    munmap(sem_ping, sizeof(sem_t));
    munmap(sem_pong, sizeof(sem_t));

    return 0;
}
```

Here we create an anonymous shared memory region. That memory:

- Is only shared between parent and children created using fork().
- Cannot be accessed by totally separate processes (like separate binaries/files run independently).
- There's no persistent name or file backing it — once the process ends, the memory vanishes.

So this only works because all processes are created from one executable using fork().

Shared Memory & Semaphores

In this example shared data is not actually used, we can print ping, pong alternatively without it. I just added this to show how we can work with shared memory and semaphores.

Shared data: shared_def.h

```
#ifndef SHARED_DEFS_H
#define SHARED_DEFS_H

#define SHM_NAME "/shm_pingpong"
#define SEM_PING "/sem_ping"
#define SEM_PONG "/sem_pong"
#define ITERATIONS 20

typedef struct {
    int counter;
} shared_data_t;

#endif
```

Ping.c

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/stat.h>
#include "shared_defs.h"

int main() {
    // Create or open shared memory
    int shm_fd = shm_open(SHM_NAME, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, sizeof(shared_data_t));
    shared_data_t *shared_data = mmap(NULL, sizeof(shared_data_t),
PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    // Initialize counter only if it's the first run
    shared_data->counter = 0;

```

```c
    sem_t *sem_ping = sem_open(SEM_PING, O_CREAT, 0666, 1);
    sem_t *sem_pong = sem_open(SEM_PONG, O_CREAT, 0666, 0);

    for (int i = 0; i < ITERATIONS; i++) {
        sem_wait(sem_ping);
        printf("ping\n");
        fflush(stdout);
        shared_data->counter++;
        sem_post(sem_pong);
    }

    munmap(shared_data, sizeof(shared_data_t));
    close(shm_fd);
    sem_close(sem_ping);
    sem_close(sem_pong);

    return 0;
}
```

Pong.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <semaphore.h>
#include <sys/stat.h>
#include "shared_defs.h"

int main() {
    // Open existing shared memory
    int shm_fd = shm_open(SHM_NAME, O_RDWR, 0666);
    shared_data_t *shared_data = mmap(NULL, sizeof(shared_data_t),
PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    sem_t *sem_ping = sem_open(SEM_PING, 0);
    sem_t *sem_pong = sem_open(SEM_PONG, 0);

    for (int i = 0; i < ITERATIONS; i++) {
        sem_wait(sem_pong);
        printf("pong\n");
        fflush(stdout);
        shared_data->counter++;
        sem_post(sem_ping);
    }

    munmap(shared_data, sizeof(shared_data_t));
    close(shm_fd);
    sem_close(sem_ping);
    sem_close(sem_pong);

    if (shared_data->counter >= ITERATIONS * 2) {
        shm_unlink(SHM_NAME);
        sem_unlink(SEM_PING);
        sem_unlink(SEM_PONG);
    }

    return 0;
}
```