Wait Syntax

pid_t wait(int *stat_loc);

Parameters:

- **stat_loc**: A pointer to an integer where information about the child process's termination status will be stored. This status can be used with macros like WIFEXITED, WIFSIGNALED, WIFSTOPPED, etc., to analyze the exit condition of the child process.

Return Value:

- On success, wait returns the **process ID (PID)** of the child process that terminated.
- On failure, it returns -1. In this case, errno will be set to indicate the error, for example:
  - ECHILD: The calling process has no children.
  - EINTR: The call was interrupted by a signal before any child terminated.

MACROS to Evaluate the Termination Status

1. WIFEXITED(status)

- **Purpose**: Checks if the child process terminated normally (i.e., by calling exit() or returning from main()).
- **Return Value**:
  - **True** (non-zero) if the child process terminated normally.
  - **False** (zero) if the child did not terminate normally.
- **Usage**: This macro is used when the parent process wants to check if the child exited with a specific status code.

Example:

```
if (WIFEXITED(status)) {
   int exit_status = WEXITSTATUS(status);  // Retrieve the exit code of the child
   printf("Child exited normally with status %d\n", exit_status);
}
```

Explanation:

- If the child process called exit(value) or returned from main(value), this macro will return true.
- You can then use **WEXITSTATUS(status)** to get the exit value of the child process.

2. WIFSIGNALED(status)

- **Purpose**: Checks if the child process was terminated by a signal (such as due to an unhandled signal or abort).
- **Return Value**:
  - **True** (non-zero) if the child process was terminated by a signal.
  - **False** (zero) if the child terminated normally or was stopped.
- **Usage**: This macro is used when the parent process wants to check if the child process was killed by a signal.

Example:

```
if (WIFSIGNALED(status)) {
    int signal_num = WTERMSIG(status);  // Retrieve the signal number that caused termination
    printf("Child terminated by signal %d\n", signal_num);
}
```

Explanation:

- If the child process was terminated by a signal, this macro will return true.
- In this case, execute **WTERMSIG(status)** to fetch the signal number that caused the termination.
- Additionally, some implementations define the macro **WCOREDUMP(status)** that returns true if a core file of the terminated process was generated

3. **WIFSTOPPED(status)**

- **Purpose**: Checks if the child process was stopped (i.e., it was paused due to receiving a signal like SIGSTOP or SIGTSTP).
- **Return Value**:
  - **True** (non-zero) if the child process is stopped (paused).
  - **False** (zero) if the child terminated normally or was terminated by a signal.
- **Usage**: This macro is used when the parent process wants to check if the child is currently stopped.
- In this case, execute **WSTOPSIG(status)** to fetch the signal number that caused the child to stop.

Example:

```
if (WIFSTOPPED(status)) {
    int signal_num = WSTOPSIG(status);  // Retrieve the signal number that caused the stop
    printf("Child stopped by signal %d\n", signal_num);
}
```

Explanation:

- If the child process was stopped (e.g., by SIGSTOP or SIGTSTP), this macro will return true.
- You can use **WSTOPSIG(status)** to retrieve the signal that caused the stop.

4. **WIFCONTINUED(status)**

Purpose:

- **WIFCONTINUED(status)** checks whether a child process has been resumed (i.e., continued) after being stopped, typically by a signal such as **SIGCONT**.

Return Value:

- **True** (non-zero) if the child process has been resumed and is no longer stopped.

- **False** (zero) if the child has not been resumed (i.e., it has either exited or is still stopped).

A Sample Code: Macro Evaluation

```
#include <stdio.h>

#include <unistd.h>  // For fork()

#include <stdlib.h>

#include <sys/wait.h>

int main(void){

pid_t childpid,pid;

int status;

pid=fork();

if(pid==0){

printf("Child Part Executed!!!! %ld\n", (long)getpid());

//sleep(100);

//int n;

//scanf("%d", &n);

exit(0);

}

else{

childpid=wait(&status);

if (childpid == -1)

perror("Failed to wait for child\n");

else if (WIFEXITED(status) && !WEXITSTATUS(status))

printf("Child %ld terminated normally\n", (long)childpid);

else if (WIFEXITED(status))

printf("Child %ld terminated with return status %d\n",(long)childpid
```

, WEXITSTATUS(status));

else if (WIFSIGNALED(status))

printf("Child %ld terminated due to uncaught signal %d\n",(long)

childpid, WTERMSIG(status));

else if (WIFSTOPPED(status))

printf("Child %ld stopped due to signal %d\n",(long)childpid,

WSTOPSIG(status));

}

return 0;}

ps –la

kill

Limitations of wait() System Call

✍It is not possible for the parent to retrieve the signal number during which the child process has stopped the execution(SIGSTOP(19), SIGTSTP(20)).

✍ Parent won't be able to get the notification when a stopped child was resumed by the delivery of a signal (SIGCONT(18),SIGCHLD(17)).

✍ Parent can only wait for first child that terminates. It is not possible to wait for a particular child.

✍ It is not possible for a non-blocking wait so that if no child has yet terminated, parent get an indication of this fact

Waitpid Syntax

pid_t waitpid(pid_t pid, int *stat_loc, int options);

Parameters:

1. **pid**:
    - Specifies which child process to wait for.
    - If pid > 0: Wait for the specific child process with that **PID**.
    - If pid == 0: Wait for any child in the same process group as the calling process.
    - If pid == -1: Wait for any child process (similar to wait).

    - If pid < -1: Wait for any child in the process group with the group ID equal to |pid| (the absolute value of pid).
2. **stat_loc**:
    - A pointer to an integer where the status of the child process will be stored. The status can be analyzed using macros like WIFEXITED, WIFSIGNALED, WIFSTOPPED, etc.
3. **options**:
    - Specifies options that modify the behavior of waitpid. Some commonly used options are:
        - options = 0 :: The waitpid waits until the child change state.
        - **WNOHANG**: The waitpid will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
        - **WUNTRACED**: The waitpid to report the status of unreported child processes that have been stopped. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process.
        - **WCONTINUED**: The waitpid returns if a stopped child has been resumed by delivery of the signal SIGCONT.

Return Value:

- **On success**:
    - Returns the **PID** of the child process that terminated (or stopped).
- **On error**:
    - Returns -1 and sets errno to indicate the error. Some possible errors include:
        - ECHILD: No child processes.
        - EINVAL: Invalid options specified.

Option in detail

1. `options = 0` (Default Behavior)

When `options = 0`, this is the default behavior of `waitpid`. It blocks (waits) until the specified child process changes its state (usually meaning the child either:

- **Terminates** (finishes execution),
- **Stops** (pauses its execution), or
- **Resumes** (if it was stopped previously).

In this case, `waitpid` doesn't return until the child process has finished or changed its state in some way. If the child process has already terminated or changed state, `waitpid` will return immediately with the status of that child.

2. `WNOHANG` (Non-blocking Behavior)

When you specify the `WNOHANG` option, `waitpid` will **not block** (pause execution of the calling process) if the specified child process hasn't changed state yet. Instead of blocking, it checks the child's state and returns immediately.

- If the child has changed state (such as finishing execution or being stopped), `waitpid` will return with that information.
- If the child is still running or hasn't changed state, `waitpid` will return `0`.

This is useful when you want to **poll** child processes without waiting indefinitely. For example, you might want to check whether a child has exited while the parent process continues doing other work.

3. `WUNTRACED` (Stopped Children)

When you use the `WUNTRACED` option, `waitpid` will report the status of any **stopped** child processes. Stopped processes are those that have been paused, typically due to receiving a signal like `SIGSTOP` or `SIGTSTP`.

- Even if a child is stopped, `waitpid` will return its status.
- The macro `WIFSTOPPED(status)` can be used to check if the child process is actually stopped.
- This is useful when you want to know if any child processes have been paused or stopped for any reason (like waiting for user input or being sent a signal).

4. `WCONTINUED` (Resumed Children)

The `WCONTINUED` option makes `waitpid` report when a **stopped child** has been **resumed** after being paused by receiving the `SIGCONT` signal. For example, a child process might be stopped, and later a `SIGCONT` signal can be sent to continue its execution.

- If a stopped child resumes, `waitpid` will return with the status information about this child.
- You can use the macro `WIFCONTINUED(status)` to check if the child has been resumed.

wait Vs waitpid a case

wait(NULL) or waitpid(-1, NULL, 0);

int status;

wait(&status) or waitpid(-1, &status, 0);

exec Family

The **exec** family of functions replaces the current process image with a new process image.

Syntax

#include <unistd.h>

extern char **environ;

int execl(const char *path, const char *arg0, ... /*, char *(0) */);

int execle (const char *path, const char *arg0, ... /*, char *(0), char *const

envp[] */);

int execlp (const char *file, const char *arg0, ... /*, char *(0) */);

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

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

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

Return Value of these functions:

return -1 if unsuccessful on successful no return to the calling process

The exec Family Series

The execl (execl, execlp and execle) functions pass the commandline arguments in an explicit list and are useful if the number of commandline arguments are known at compile time.

The **execv (execv, execvp and execve)** functions pass the commandline arguments in an argument array

The exec: l series: l - a fixed list of arguments

✍ execl(argument list)

✍ execle(argument list)

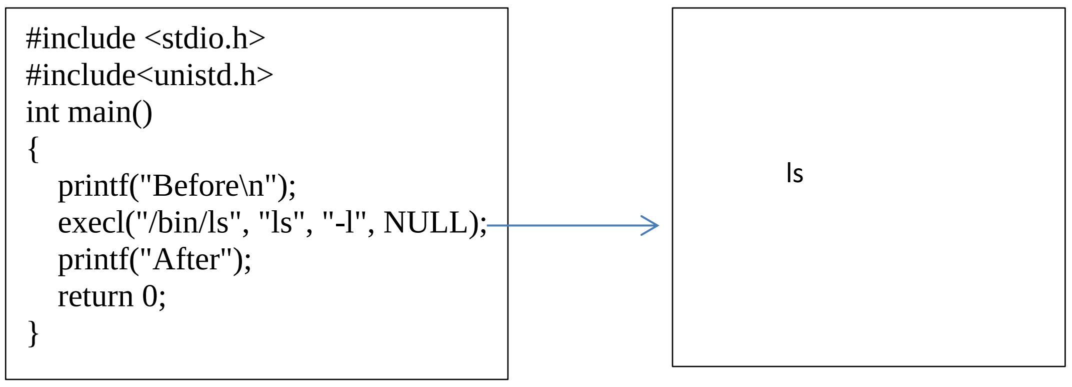
✍ execlp(argument list

✍ The exec: v series: v - a variable number of arguments

✍ execv(argument list)

✍ execve(argument list)

✍ execvp(argument list)



```
#include <stdio.h>
#include<unistd.h>
int main()
{
   printf("Before\n");
   execl("/bin/ls", "ls", "-l", NULL);
   printf("After");
   return 0;
}
```

execlp: It enables you to switch out the current process image for a different one that is specified by the given program path. This function comes in handy when you wish to **run outside programs from a C program.**

#include <unistd.h>

int execlp(const char *file, const char *arg0, ..., (char *)0);

Example :

```
#include <unistd.h>

#include <stdio.h>

  int main() {

printf("Executing ls command...\n");

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

return 0;

}
```

execle: e- environment of new process

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

int err;

err=execle("/usr/bin/wc","wc","execldemo.c",NULL,NULL);

if(err==-1){

perror("Execle Failed\n");

}

return 0;

}
```

The **execv** function takes exactly two parameters, a pathname for the executable and an argument array

Syntax:

```
#include<unistd.h>

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

Example1:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

char *cmdargs[]={"ls", "-l", NULL};

pid_t pid;

pid=fork();

if(pid==0){

execv("/bin/ls",cmdargs);

}

else{

wait(NULL);

printf("child terminate\n");

}

return 0;

}
```

Example1: (With Command line Argument)

```
#include<stdio.h>

#include<stdlib.h>
```

```
#include<unistd.h>

int main(int argc, char *argv[])

{

int i;

char *cmdarg[20];

for(i=1;argv[i]!=NULL;i++){

cmdarg[i-1]=argv[i];

}

cmdarg[i-1]=NULL;

execv("/bin/ls",cmdarg);

return 0;

}
```

To run ./a.out ls -l

execvp:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

char *cmdargs[]={"ls", "-l", NULL};

pid_t pid;

pid=fork();

if(pid==0){

execvp("ls",cmdargs);

}
else{

wait(NULL);

printf("chile terminate\n");

}

return 0;

}
```

execve:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

int main(void)

{

char *cmdargs[]={"ls", "-l", NULL};

pid_t pid;

pid=fork();

if(pid==0){

execve("/bin/ls", cmdargs, NULL);

}

else{

wait(NULL);

printf("chile terminate\n");

}

return 0;

}
```

PIPE:

A pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In UNIX Operating System, Pipes are useful for communication between related processes (inter-process communication).

It opens a pipe, which is an area of main memory that is treated as a ***"virtual file"***.

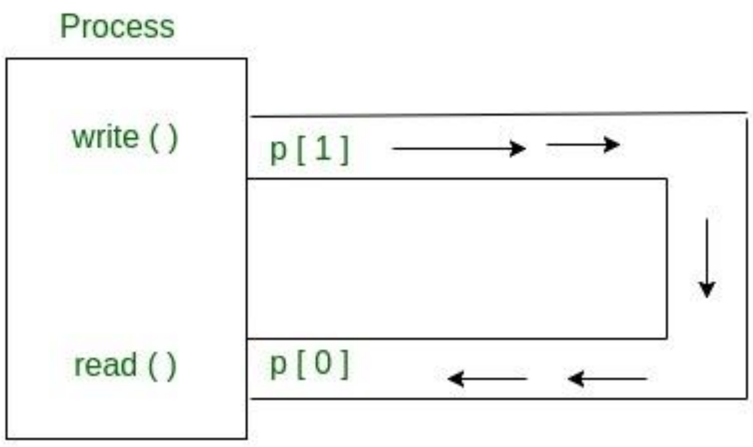
Files are designated with in C program either by file pointers or by file descriptor.

✍ **A file descriptor is an integer value that represents a file or device that is open.**

Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. If the pipe call executes successfully, the process can read from fd[0] and write to fd[1].

The pipe can be used by the creating process, as well as all its child processes, for reading and writing

read before something is written to the pipe, the process is suspended until something is written.



int pipe(int fds[2]);

Parameters :
fd[0] will be the fd(file descriptor) for the read end of pipe.
fd[1] will be the fd for the write end of pipe.
Returns : 0 on Success.
-1 on error.

What is file Descriptor

A file descriptor is an integer handle used by operating systems(like Linux) to reference an open file or other I/O resource (e.g., pipes, sockets, or devices). It's essentially a unique identifier for an open file or resource within a process.

Key Features of File Descriptors

1. **Small Integer:** File descriptors are typically small non-negative integers starting from 0.
2. **Per-Process Basis**: Each process has its own file descriptor table, so file descriptor numbers may be reused across different processes.
3. **System Calls:** File descriptors are used in various system calls like read(), write(), close(), etc., to interact with files and other I/O resources.

Standard File Descriptors

When a process starts, it is given three default file
descriptors:

File Descriptor	Name	Description
0	Standard Input	Used for input (e.g., keyboard).
1	Standard Output	Used for output (e.g., terminal).
2	Standard Error	Used for error messages (e.g., logs).

Example:

printf() sends data to stdout (file descriptor 1).
scanf() reads input from stdin (file descriptor 0).

How File Descriptors Work

When you open a file (using open(), fopen(), etc.), the operating system assigns a file descriptor to track that file. You use this descriptor in subsequent system calls to read, write, or manipulate the file. In Linux programming, the pipe() system call is used for inter-process communication (IPC). It creates a unidirectional communication channel that can pass data between processes.

A pipe has two ends:
1. Write-end: Data is written to the pipe.
2. Read-end: Data is read from the pipe.

Key Concepts of pipe()

● Unidirectional: Data flows in one direction, from write-end to read-end.

● File descriptors: The pipe() system call creates two file descriptors:
○ pipefd[0]: For reading.
○ pipefd[1]: For writing.

● Typically used between parent and child processes.

Syntax of pipe()
```
#include <unistd.h>
int pipe(int pipefd[2]);
```
● pipefd[2]: An array to hold the file descriptors.
○ pipefd[0] is for reading.
○ pipefd[1] is for writing.
● Returns 0 on success and -1 on failure.

Example: Using pipe()

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
int main() {
int pipefd[2]; // Array to hold the pipe file descriptors
pid_t pid;
char write_msg[] = "Hello from parent process!";
char read_msg[100];
// Create the pipe
if (pipe(pipefd) == -1) {
perror("pipe");
return 1;
}
// Fork a child process
pid = fork();
if (pid < 0) {
perror("fork");
return 1;
}
if (pid == 0) { // Child process
close(pipefd[1]); // Close unused write end
// Read data from the pipe
read(pipefd[0], read_msg, sizeof(read_msg));
printf("Child received: %s\n", read_msg);
close(pipefd[0]); // Close the read end
} else { // Parent process
close(pipefd[0]); // Close unused read end
// Write data to the pipe
write(pipefd[1], write_msg, strlen(write_msg) + 1);
printf("Parent sent: %s\n", write_msg);
close(pipefd[1]); // Close the write end
}
return 0;
}
```
Explanation of the Program

1. Pipe Creation:
○ pipe(pipefd) creates a pipe and assigns file descriptors
to pipefd[0] (read) and pipefd[1] (write).
2. Process Forking:
○ fork() creates a child process.
○ The parent and child process share the pipe.
3. Parent Process:
○ Closes the read end (pipefd[0]).
○ Writes a message to the write end (pipefd[1]).
4. Child Process:
○ Closes the write end (pipefd[1]).
○ Reads the message from the read end (pipefd[0]).
○ Prints the received message.
Output:

```
Parent sent: Hello from parent process!
Child received: Hello from the parent process!
```

Important Notes
1. Blocking Behavior:
○ The read() system call will block (wait) until there is data to read.
○ The write() system call will block if the pipe is full

System Call

1. write() system call is used to write to a file descriptor. In other words write() can be used to write to any file (all hardware are also referred as file in Linux) in the system but rather than specifying the file name, you need to specify its file descriptor.

Syntax:

```
#include<unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

The first parameter (fd) is the file descriptor where you want to write. The data that is to be written is specified in the second parameter. Finally, the third parameter is the total bytes that are to be written.

Example:

```
#include<unistd.h>

int main()

{

write(1,"hello\n",6); //1 is the file descriptor, "hello\n" is the data, 6 is the count of
characters in data
```

}

2. read() system call is to read from a file descriptor. The working is same as write(), the only difference is read() will read the data from file pointed to by file descriptor.

Syntax:

```
#include<unistd.h>

ssize_t read(int fd, const void *buf, size_t count);
```

Example:

```
#include<unistd.h>

int main()

{

char buff[20];

read(0,buff,10);//read 10 bytes from standard input device(keyboard), store in buffer (buff)

write(1,buff,10);//print 10 bytes from the buffer on the screen

}
```

3. open() **The open function** creates an entry in the file descriptor table that points to an entry in the system file table.

Syntax

```
    #include <sys/types.h>

    #include <sys/stat.h>

    #include <fcntl.h>

        int open(const char *pathname, int flags);

        int open(const char *pathname, int flags, mode_t mode);
```

1st parameter - name of the file that you want to open for reading/writing.

2nd parameter-the mode in which to open the file i.e., for reading or for writing. For reading from a file, the flag used is O_RDONLY, for writing O_WRONLY and for both reading and writing O_RDWR.

3rd parameter- specifies the permissions on the created file (read/write/execute).

Example 1:

```
#include<stdio.h>

#include<unistd.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<fcntl.h>

int main()

{

int n,fd;

char buff[50];

fd=open("test.txt",O_RDONLY); //opens test.txt in read mode and the file descriptor is saved in
integer fd.

printf("The file descriptor of the file is: %d\n,fd); // the value of the file descriptor is printed.

n=read(fd,buff,10);//read 10 characters from the file pointed to by file descriptor fd and save them
in buffer (buff)

write(1,buff,n); //write on the screen from the buffer

}
```

Example2:

```
#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>
```

```
int main(){

   int fileDescriptor;

   fileDescriptor = open("./NewFile", O_CREAT, S_IRWXU|S_IRWXG|S_IRWXO );

   printf("File Descriptor: %d\n", fileDescriptor);

   close(fileDescriptor);



   return 0;

}
```

4. ***close()*** :

   ```
   #include <unistd.h>
   int close(int fd);
   ```

   ```
   #include<stdio.h>
   #include<unistd.h>
   #include<sys/types.h>
   #include<sys/stat.h>
   #include<fcntl.h>
   int main()
   {
   int n, fd;
   char buff[50];
   fd=open("test.txt",O_RDONLY); //opens test.txt in read mode and the file descriptor is saved
   in integer fd.
   printf("The file descriptor of the file is: %d\n",fd); // the value of the file descriptor is
   printed.
   n=read(fd,buff,10);//read 10 characters from the file pointed to by file descriptor fd and
   save them in buffer (buff)
   write(1,buff,n); //write on the screen from the buffer
   close(fd);
   return 0;
   }
   ```

PIPE

Example 1:

```
// C program to illustrate

// pipe system call in C

#include <stdio.h>
```

```
#include <unistd.h>

#define MSGSIZE 16

char* msg1 = "hello, world #1";

char* msg2 = "hello, world #2";

char* msg3 = "hello, world #3";


int main()

{

        char inbuf[MSGSIZE];

        int p[2], i;


        if (pipe(p) < 0)

                exit(1);


        write(p[1], msg1, MSGSIZE);

        write(p[1], msg2, MSGSIZE);

        write(p[1], msg3, MSGSIZE);


        for (i = 0; i < 3; i++) {

                /* read pipe */

                read(p[0], inbuf, MSGSIZE);

                printf("% s\n", inbuf);

        }

        return 0;

}
```

Example 2: Parent and child

```
// shared by Parent and Child
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main()
{

int fd[2], n;
    char buffer[100];
    int p;
    pipe(fd);
    p=fork();
            if (p >0){
                printf("passing value to child \n");
                        write(fd[1], "hello\n", 6);

            }
            else{
                        printf("Child received data\n");
                        n=read(fd[0], buffer, 100);
                        write(1,  buffer, n);
            }
}
```

Thread

1. **Thread create**

#include<pthread.h>

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

1st parameter - the buffer which will contain the ID of the new thread, if pthread_create is successful.

2nd parameter- specifies the attributes of the thread. This parameter is generally NULL until you want to change the default settings.

3rd parameter- is the name the function which the thread will execute.

4th Parameter- is the input to the function in the third parameter.

2. **Thread wait**
   #include <pthread.h>
   int pthread_join(pthread_t thread, void **retval);

In C, the pthread_join() function is used to wait for a thread to finish executing. It is typically called in the main thread (or another thread) to block the calling thread until the specified thread has completed its execution.

 `thread: The ID of the thread you want to wait for. This is typically the value returned by pthread_create().`

 `retval: A pointer to a location where the exit status of the thread will be stored. If you don't care about the return value, you can pass NULL.`

Example:

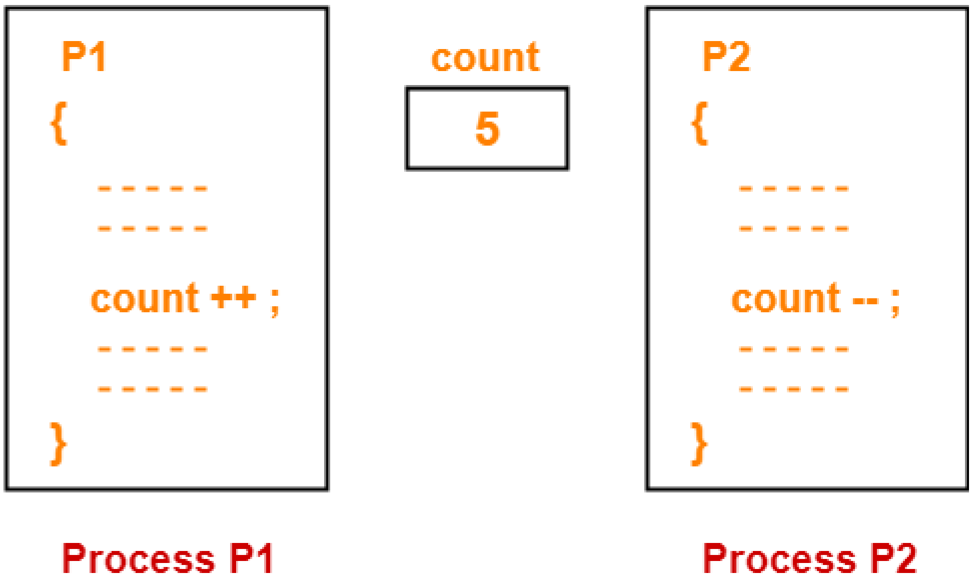
```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<pthread.h>

void *thread_function(void *arg);
```

```
 int i,j;

 int main() {

 pthread_t a_thread;  //thread declaration



 pthread_create(&a_thread, NULL, thread_function, NULL);

//thread is created

 pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line
to see the difference

 printf("Inside Main Program\n");

 for(j=20;j<25;j++)

 {

 printf("%d\n",j);

 sleep(1);

 }

 }

void *thread_function(void *arg) {

// the work to be done by the thread is defined in this function

 printf("Inside Thread\n");

 for(i=0;i<5;i++)

 {

 printf("%d\n",i);

 sleep(1);

 }

 }
```

```
$gcc Thread.c -lpthread
$./a.out
```

Race condition

A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in a critical section differs according to the order in which the threads execute.



```
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1;
int main()
{
   pthread_t thread1, thread2;
```

```
    pthread_create(&thread1, NULL, fun1, NULL);

    pthread_create(&thread2, NULL, fun2, NULL);

    pthread_join(thread1, NULL);

    pthread_join(thread2, NULL);

    printf("final value of shared is %d \n", shared);

}

void *fun1(){

    int x;

    x=shared;

    printf("Thread1 read the value of shared variable as %d \n", x);

    x++;

    printf("Local updation by Thread1 as %d \n", x);

    sleep(1);

    shared=x;

    printf("value of shared variable by thread 1 is %d \n", shared);

}

void *fun2(){

    int y;

    y=shared;

    printf("Thread2 read the value of shared variable as %d \n", y);

    y--;

    printf("Local updation by Thread2 as %d \n", y);

    sleep(1);

    shared=y;

    printf("value of shared variable by thread 2 is %d \n", shared);

}
```

SEMAPHORE

Is an integer variable which is used in mutually exclusive manner by various concurrent cooperative process to achieve synchronization.

Critical Section: When more than one processes access the same code segment that segment is known as the critical section.

In simple terms, a critical section is a group of instructions/statements or region of code that need to be executed atomically such as accessing a resource (file, input or output port, global data, etc.)

Entry section

CS

EXIT Section

There are various operations in entry and exit code such as wait and signal

P(), down, wait ----reduce by 1 value of Semaphore

V(), up, signal, post, release ------increase by 1 value of Semaphore

```
Wait(s)
{
While(s<=0);  // if true then loop continues
s=s-1;        // if false then only decrease
}
```

```
Signal(s)

{

s=s+1;

}
```

1. Unnamed Semaphore

An unnamed semaphore is used within the process that creates it. It does not have a name in the system and is identified by a pointer. Unnamed semaphores are generally created using sem_init()

1. **Declares a semaphore variable called sem: (sem_t)**

   ```
   #include <semaphore.h>
   sem_t sem;
   ```

2. **Initialize semaphore variable**

   ```
   #include int sem_init(sem_t *sem, int pshared, unsigned value);
   ```

1st arg= address of semaphore variable

2nd arg= number (0- shared between threads, Non Zero- shared between processes)

3rd arg = initial value of semaphore

Returns - returns 0 on success and -1 on error

1. **To wait on a semaphore**, use sem_wait:

   ```
   int sem_wait(sem_t *sem);
   ```

Example of use:

```
sem_wait(&sem_name);
```

2. To increment the value of a semaphore, use sem_post:

```
int sem_post(sem_t *sem);
```

Example of use:

```
sem_post(&sem_name);
```

3. **To find out the value of a semaphore, use**

   ```
   int sem_getvalue(sem_t *sem, int *valp);
   ```

- gets the current value of sem and places it in the location pointed to by `valp`

Example of use:
```
int value;
sem_getvalue(&sem_name, &value);
printf("The value of the semaphors is %d\n", value);
```

4. To destroy a semaphore, use

```
int sem_destroy(sem_t *sem);
```

- destroys the semaphore; no threads should be waiting on the semaphore if its destruction is to succeed.

Example of use:
```
sem_destroy(&sem_name);
```

```
// Semaphore

#include<pthread.h>

#include<stdio.h>

#include<unistd.h>

#include<semaphore.h>

void *fun1();

void *fun2();

int shared=1;

sem_t s;

int main()

{

   sem_init(&s, 0, 1);

   pthread_t thread1, thread2;

   pthread_create(&thread1, NULL, fun1, NULL);

   pthread_create(&thread2, NULL, fun2, NULL);

   pthread_join(thread1, NULL);
   pthread_join(thread2, NULL);
   printf("final value of shared is %d \n", shared);
}
void *fun1(){
   int x;
   sem_wait(&s);
   x=shared;
   printf("Thread1 read the value of shared variable as %d \n", x);
   x++;
   printf("Local updation by Thread1 as %d \n", x);
   sleep(1);
   shared=x;
   printf("value of shared variable by thread 1 is %d \n", shared);
   sem_post(&s);
}
void *fun2(){
   int y;
   sem_wait(&s);
   y=shared;
   printf("Thread2 read the value of shared variable as %d \n", y);
   y--;
   printf("Local updation by Thread2 as %d \n", y);
   sleep(1);
   shared=y;
   printf("value of shared variable by thread 2 is %d \n", shared);
   sem_post(&s);
}
```

Named Semaphore

A named semaphore is a semaphore that has a name and is stored in the system-wide namespace. Named semaphores are created using sem_open().

1. **sem_open** - initialize and open a named semaphore

   syntax
   #include <fcntl.h>   /* fcntl.h is a header file in the C standard library that provides the declarations for file control options and functions. It is commonly used for manipulating file descriptors and performing advanced input/output operations. */
   #include<sys/stat.h>   /* The <sys/stat.h> header file in C provides the declarations for functions and macros used to obtain and manipulate file attributes.*/
   #include<semaphore.h>

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag,
                        mode_t mode, unsigned int value);
```

RETURN== 1. successful =address of the semaphore.
2.unsuccessful= SEM_FAILED

NOTE

1st Parameter- This is a string that represents the name of the semaphore.
2nd Parameter- The oflag parameter is either 0, O_CREAT, or O_CREAT | O_EXECL
0- **opening an existing semaphore**
If the O_CREAT is specified, the sem_open requires two more parameters:
a mode parameter of type mode_t giving the permissions and
a value parameter of type unsigned giving the initial value of the semaphore

 sem_wait():

- **Blocks if the semaphore value is 0.**
- **Used to acquire the semaphore before entering a critical section.**

 sem_close():

- **Disconnects the calling process from the semaphore.**
- **Does not remove the semaphore from the system.**

 sem_unlink():

- **Removes the semaphore from the system namespace.**

`sem_post()`

- **Increments the Semaphore Value**:
  - If the semaphore count is greater than or equal to `0`, it is incremented by `1`.
  - If one or more threads or processes are waiting on the semaphore (blocked on `sem_wait`), one of them is unblocked.
-

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



const char *semName = "asdfsd";

void parent(void){
    sem_t *sem_id = sem_open(semName, O_CREAT, 0600, 0);

/*  sem_t *sem_id:
• A pointer to a semaphore object. This will be used to
  interact with the semaphore in subsequent operations
  (e.g., sem_wait, sem_post, sem_close).
    sem_open():
• This function creates or opens a named semaphore.
    semName:
• A string representing the name of the semaphore.
• It should start with a forward slash / (e.g.,
  "/my_semaphore") and should be unique in the system
  namespace.
    O_CREAT:
• A flag indicating that the semaphore should be created if
  it doesn't already exist. If the semaphore exists, it
  will open the existing semaphore.
    0600:
• The permissions for the semaphore, specified in octal
  format (similar to file permissions in Linux).
• 0600 means:
    o 6: Owner has read and write permissions.
    o 0: No permissions for the group.
    o 0: No permissions for others.
    0:
• The initial value of the semaphore. This is the starting
  count.
• In this case, the semaphore is initialized to 0, meaning
  it is locked initially, and threads or processes waiting
  on it will block until the semaphore is incremented with
  sem_post. */


    if (sem_id == SEM_FAILED){
        perror("Parent  : [sem_open] Failed\n"); return;
    }
    printf("Parent  : Wait for Child to Print\n");
    if (sem_wait(sem_id) < 0)
        printf("Parent  : [sem_wait] Failed\n");
    printf("Parent  : Child Printed! \n");

    if (sem_close(sem_id) != 0){
        perror("Parent  : [sem_close] Failed\n");
return;
    }

    if (sem_unlink(semName) < 0){
        printf("Parent  : [sem_unlink] Failed\n");
return;
    }
}

void child(void)
{
    sem_t *sem_id = sem_open(semName, O_CREAT, 0600, 0);

    if (sem_id == SEM_FAILED){
        perror("Child   : [sem_open] Failed\n"); return;
    }

    printf("Child   : I am done! Release Semaphore\n");
    if (sem_post(sem_id) < 0)
        printf("Child   : [sem_post] Failed \n");
}

int main(int argc, char *argv[])
{
    pid_t pid;
    pid = fork();

    if (pid < 0){
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (!pid){
        child();
        printf("Child   : Done with sem_open \n");
    }
    else{
        int status;
        parent();
        wait(&status);
        printf("Parent  : Done with sem_open \n");
    }

    return 0;
```

```
}
```

Output

```
Parent  : Wait for Child to Print    (after this wait loop mai chalajayega as semaphore is 0 and by that time child starts)

Child   : I am done! Release Semaphore

Child   : Done with sem_open

Parent  : Child Printed!

Parent  : Done with sem_open
```