Process management

1. fork - Link 1, Link 2
    - `pid_t fork(void);`
    - fork() creates a new process by duplicating the calling process. It returns a negative value when the function fails to create a child process. On successful duplication of a process, the PID of the child process is returned in the parent, and 0 is returned in the child process.
2. wait(NULL)
    - wait(NULL) will block the parent process until any of its children has finished. If the child terminates before the parent process reaches wait(NULL), then the child process turns to a zombie process until its parent waits on it and it is released from memory.
3. execlp
    - `int execlp(const char *file, const char *arg, ...)`
    - The file argument is the path name of an executable file to be executed. arg is the string we want to appear as argv[0] in the executable. By convention, argv[0] is just the executable file name; normally, it's set to the same as the file. The ... are now the additional arguments to give to the executable.
4. fopen - Link 1
    - `FILE *fopen(const char *pathname, const char *mode`
    - The `fopen()` function opens the file whose name is the string pointed to by pathname and associates a stream with it.
5. fclose - Link 1
    - `int fclose(FILE *stream`
    - The `fclose()` function flushes the stream pointed to by the stream (writing any buffered output data using `fflush()`) and closes the underlying file descriptor.
6. fgetc - Link 1
    - `int fgetc(FILE *pointer)`
    - fgetc() is used to obtain input from a file, a single character at a time. This function returns the ASCII code of the character read by the function. `pointer` is a pointer to a FILE object that identifies the stream on which the operation is to be performed.

Pipes

1. pipe - Link 1
    - `int pipe(int pipefd[2])`
    - pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe.

2. write - Link 1
    - `ssize_t write(int fd, const void buf[.count], size_t count)`
    - write() writes up to *count* bytes from the buffer starting at *buf* to the file referred to by the file descriptor *fd*. On success, the number of bytes written is returned. On error, -1 is returned, and errno is set to indicate the error.
3. read - Link 1
    - `ssize_t read(int fd, const void buf[.count], size_t count)`
    - read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. On success, the number of bytes read is returned (zero indicates the end of the file), and the file position is advanced by this number.
4. close - Link 1, Link 2(Highly recommended read)
    - `int close(int fd)`
    - close() closes a file descriptor so that it no longer refers to any file and may be reused. close() returns zero on success. On error, -1 is returned, and errno is set to indicate the error.
5. execlp -
    - `int execlp(const char *file, const char *arg, ...)`
    - The file argument is the path name of an executable file to be executed. arg is the string we want to appear as argv[0] in the executable. By convention, argv[0] is just the executable file name; normally, it's set to the same as the file. The ... are now the additional arguments to give to the executable.
6. dup2 - Link 1, Link 2
    - `int dup(int oldfd), int dup2(int oldfd, int newfd)`
    - The dup() system call allocates a new file descriptor that refers to the same open file description as the descriptor oldfd. The dup2() system call performs the same task as dup(), but instead of using the lowest-numbered unused file descriptor, it uses the file descriptor number specified in newfd. In other words, the file descriptor newfd is adjusted to now refer to the same open file description as oldfd.
7. wait - Link 1,
    - `pid_t wait(int *_Nullable wstatus)`
    - Kindly go through this

Message queues

1. ftok - Link 1
    - `key_t ftok(const char *pathname, int proj_id)`
    - The ftok() function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file)and the least significant 8 bits of *proj_id* (which must be nonzero) to generate a *key_t* type System V IPC key. The resulting value is the same for all pathnames that name the same file when the same value of *proj_id* is used.
2. msgget - Link 1
    - `int msgget(key_t key, int msgflg)`

- The msgget() system call returns the message queue identifier associated with the value of the *key* argument. It may be used to obtain the identifier of a previously created message queue or create a new one. The *msgflag* field can be modified using bitwise operations to use the function differently. For example, `msgget(key, PERMS | IPC_CREAT)` will create a new message queue with the id *key* if it already does not exist.

3. msgrcv - Link 1(Highly recommended)
   - `ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg)`
   - An appropriate explanation regarding the function can be found in the link mentioned above.
4. msgsend - Link 1(Highly recommended)
   - `int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg)`
   - An appropriate explanation regarding the function can be found in the link mentioned above.
5. fgets - Link 1
   - `char *fgets (char *str, int n, FILE *stream)`
   - *str* is a pointer to an array of chars where the string read is copied. *n* is the maximum number of characters copied into *str*(including the terminating null character). **stream* is a pointer to a FILE object that identifies an input stream. The fgets() function returns a pointer to the string where the input is stored.
6. msgctl - Link 1, Link 2(Highly recommended)
   - `int msgctl(int msqid, int cmd, struct msqid_ds *buf)`
   - The *msgctl*() function shall provide message control operations as specified by *cmd.* Refer to Link 2, as mentioned above, for more information on what each value of *cmd* would do. Information about *buf* can be found in Link 1, but the field can be kept NULL or 0 depending on the purpose for which the *msgctl()* function is used.
7. sprintf - Link 1, Link 2
   - `int sprintf(char *str, const char *format, ...)`
   - sprintf stands for "string print." In C programming language, it is a file-handling function that sends formatted output to the string. Instead of printing on the console, the *sprintf()* function stores the output on the char buffer specified in sprintf.

Shared Memory

1. shmget - Link 1(Highly recommended)
   - `int shmget(key_t key, size_t size, int shmflg)`
   - The shmget() system call returns the shared memory identifier associated with the value of the *key* argument. It may be used either to obtain the identifier of a previously created shared memory segment or to create a new one with size equal to the value of *size* rounded up to a multiple of PAGE_SIZE. The *shmflg* field can be

modified using bitwise operations to use the function in different ways. For example, shmget(SHM_KEY, BUF_SIZE, 0644 | IPC_CREAT)
Will create a new shared memory segment with id SHM_KEY if it already does not exist.

2. shmat - Link 1
    - `void *shmat(int shmid, const void *shmaddr, int shmflg)`
    - The *shmat*() function attaches the shared memory segment associated with the shared memory identifier specified by *shmid* to the address space of the calling process. *shmflg* is used to determine the operation to be performed by the *shmat()* if *shmaddr* is not null or for reading.
3. shmdt - Link 1
    - `int shmdt(const void *shmaddr)`
    - The *shmdt*() function detaches the shared memory segment located at the address specified by *shmaddr* from the address space of the calling process.
4. shmtcl - Link 1, Link 2(Highly recommended)
    - `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`
    - The *shmctl*() function provides a variety of shared memory control operations as specified by *cmd*. Refer to Link 2, as mentioned above, for more information on what each value of *cmd* would do. Information about *buf* can be found in Link 1, but the field can be kept NULL or 0 depending on the purpose for which the *shmtcl()* function is used.
5. ftok - Link 1
    - `key_t ftok(const char *pathname, int proj_id)`
    - The ftok() function uses the identity of the file named by the given *pathname* (which must refer to an existing, accessible file)and the least significant 8 bits of *proj_id* (which must be nonzero) to generate a *key_t* type System V IPC key. The resulting value is the same for all pathnames that name the same file when the same value of *proj_id* is used.
6. strcpy - Link 1
    - `char* strcpy(char* destination, const char* source)`
    - The strcpy() function copies the string pointed by *source* (including the null character) to the *destination*. The *strcpy()* function also returns the copied string.
7. fgets - Link 1
    - `char *fgets (char *str, int n, FILE *stream)`
    - *str* is a pointer to an array of chars where the string read is copied. *n* is the maximum number of characters to be copied into *str*(including the terminating null character). **stream* is a pointer to a FILE object that identifies an input stream. The fgets() function returns a pointer to the string where the input is stored.