Operating System Lab: CS341

LAB 4: Inter Process Communication

Course Instructor: Dr Satendra Kumar

Headers for Process Management and Communication

➢ **<stdlib.h>:** The stdlib.h header includes general-purpose utility functions for memory allocation, program control, conversions, and others.

- **Key Functions**:
  - **malloc(), calloc(), realloc(), free():** Dynamic memory allocation and deallocation.
  - **exit(), abort():** Functions for program termination.
  - **atoi(), atof(), atol(), strtol(), strtod():** Conversion functions from strings to integers, floating points, etc.
  - **rand(), srand():** Functions for generating pseudo-random numbers.
  - **system():** Executes a system command.

- **<unistd.h>:** The unistd.h header provides access to the POSIX (Portable Operating System Interface) API, including functions for system calls related to process control, file I/O, and other low-level operations.

- **Key Functions**:
  - **fork():** Creates a new process by duplicating the calling process.
    For parent : returns PID if child process is created, For child returns else if error is encountered returns - 1
  - **pipe():** Creates a unidirectional communication channel (pipe) for IPC.
    Returns 0, and the file descriptors for the read and write ends of the pipe are stored in the array passed as an argument
  - **read(), write():** Performs low-level reading from and writing to file descriptors.
    Returns the number of bytes read from the file descriptor, return 0 if end of file is reached else -1
  - **close():** Closes a file descriptor.
    On success returns 0 else return -1
  - **exec() family (execvp(), execl(), etc.):** Executes a new program, replacing the current process image.
    returns -1 if error occurs
  - **getpid(), getppid():** Retrieves the process ID and parent process ID.

    getpid() : Returns the PID of the calling process. `pid_t pid = getpid();`

    getppid() : Returns the PID of the parent process. `pid_t ppid = getppid();`

- **<sys/types.h>:** The sys/types.h header defines various data types used in system calls and other low-level operations.

- **Key Types**:

```
gid_t gid = getgid();
uid_t uid = getuid();
pid_t pid = getpid();
```

  - **pid_t:** Process ID type.
  - **uid_t, gid_t:** User ID and group ID types.
  - **off_t:** Type used for file sizes and offsets.
  - **mode_t:** Type for file permissions.
  - **key_t:** Type for IPC keys used in shared memory, semaphores, and message queues.

```
off_t offset = lseek(fd, 0, SEEK_END);  // Get the size of the file
```

```
// Read and write permissions for owner, read for group and others
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;

// Change the permissions of the file
chmod("file.txt", mode);
```

- **<sys/wait.h>:** The sys/wait.h header provides macros related to process termination and functions for waiting on child processes.

- **Key Functions**:

```
pid_t child_pid = waitpid(pid, &status, 0);
```

  - **wait():** Waits for any child process to terminate.

```
pid_t terminated_pid = waitpid(pid, &status, 0);
```

  - **waitpid():** Waits for a specific child process to terminate or change state.

The wait() system call makes a parent process wait until one of its child processes terminates. It retrieves the exit status of the terminated child process.
waitpid() : It allows a parent process to wait for a specific child process or for any child process based on the specified options.

- **<sys/ipc.h>:** The sys/ipc.h header defines the IPC (Inter-Process Communication) key structure and functions for generating keys used by System V IPC mechanisms such as shared memory, message queues, and semaphores.

- **Key Functions**:

  - **ftok():** Generates a unique key for IPC objects like shared memory, semaphores, and message queues.

```
// Generate a unique key using the file path and project ID
key_t key = ftok("/tmp/somefile", 65);
```

- **<sys/shm.h>:** The sys/shm.h header provides definitions and function prototypes for working with shared memory segments, one of the IPC mechanisms in Unix-like systems.

- **Key Functions**:
  - **shmget():** Allocates a shared memory segment or retrieves the identifier of an existing one.

```
int key= 1234 ;
int size = 64 ;  int shmflg = IPC_CREAT | 0666;
int shmid = shmget(key, size, shmflg);
```

  - **shmat():** Attaches a shared memory segment to the address space of the calling process.

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

  - **shmdt():** Detaches the shared memory segment from the address space of the calling process.
  - **shmctl():** Performs control operations on a shared memory segment, such as removing it.

- **<sys/sem.h>:** The sys/sem.h header provides definitions and function prototypes for working with semaphores, another IPC mechanism used for process synchronization.

- **Key Functions**:
  - **semget():** Allocates a semaphore set or retrieves the identifier of an existing set.
  - **semop():** Performs operations on semaphores, such as incrementing or decrementing the semaphore value (locking/unlocking).
  - **semctl():** Performs control operations on a semaphore set, such as initializing, setting, or retrieving semaphore values, and removing the semaphore set.

- **void generate_numbers(int pipe_fd[]):** This function generates an array of random numbers and sends them through a pipe to another process.

- **Parameters**:

- **pipe_fd[]:** An array of two file descriptors for the pipe. pipe_fd[1] is used to write data into the pipe.

- **write(pipe_fd[1], numbers, sizeof(numbers)):** This allows another process, which has access to the other end of the pipe (pipe_fd[0]), to read these numbers.

- **void calculate_average(int input_pipe[], int output_pipe[]):** This function reads an array of numbers from a pipe, calculates their average, and sends the average through another pipe to the next process.

- **Parameters**:
  - **input_pipe[]:** An array of two file descriptors for the input pipe, used to read data (input_pipe[0]).
  - **output_pipe[]:** An array of two file descriptors for the output pipe, used to write data (output_pipe[1]).

- **void update_counter(int input_pipe[], int shm_id, int sem_id):** This function reads the average from a pipe, updates a shared counter in shared memory, and uses a semaphore to ensure synchronized access to the shared counter.

  - **Parameters**:
    - **input_pipe[]:** An array of two file descriptors for the input pipe, used to read the average (input_pipe[0]).
    - **shm_id:** The identifier for the shared memory segment that contains the counter.
    - **sem_id:** The identifier for the semaphore set used to control access to the shared memory.

- **read(int fd, void *buf, size_t count)**: Reads count bytes from the file descriptor fd into the buffer buf.

- **shmat(int shmid, const void *shmaddr, int shmflg)**: Attaches a shared memory segment identified by shmid to the address space of the calling process.

- **shmid:** Shared memory ID.
- **shmaddr:** Address where the segment should be attached (usually NULL to let the OS choose).
- **shmflg:** Flags (usually 0 for default behavior).

- **sem_op.sem_num = 0:** This specifies which semaphore within a semaphore set you want to operate on.
- **sem_op.sem_op = -1:** Indicates a **P operation** (also known as a "wait" or "lock" operation). This operation attempts to decrement the semaphore's value by 1.
- **sem_op.sem_flg = 0:** This specifies any flags that control the operation.
- **emop(sem_id, &sem_op, 1);:** This performs the actual semaphore operation using the parameters set in the sem_op structure.

- **Parameters**:
  - **semop()** is a system call used to perform operations on a semaphore set.
  - **sem_id** is the identifier of the semaphore set, obtained earlier using semget().
  - **&sem_op** is a pointer to the sembuf structure that specifies the operation to be performed.

- **sem_op.sem_op = 1:** Indicates a **V operation** (also known as a “signal" operation). This operation attempts to increment the semaphore's value by 1.

- **int shm_id = shmget(SHM_KEY, sizeof(int), IPC_CREAT | 0666); :**Creates or accesses a shared memory segment.

  - **Parameters**:
    - **SHM_KEY -** Key to identify the shared memory.
    - **sizeof(int) -** Size of the shared memory segment (to store an integer).
    - **IPC_CREAT | 0666 -** Flags to create the segment with read/write permissions.

- **int *counter = (int *)shmat(shm_id, NULL, 0); :**Attaches the shared memory segment to the process's address space.It returns a pointer to the shared memory.

- **int sem_id = semget(SEM_KEY, 1, IPC_CREAT | 0666); :**Creates or accesses a set of semaphores.

  - **Parameters**:
    - **SEM_KEY** - Key to identify the semaphore set.
    - **1** - Number of semaphores in the set.
    - **IPC_CREAT | 0666** - Flags to create the semaphore with read/write permissions.

- **semctl(sem_id, 0, SETVAL, 1); :**Initializes the semaphore to 1, meaning it is available.

  - **Parameters**:
    - **sem_id** - Identifier of the semaphore set.
    - **0** - Index of the semaphore in the set.
    - **SETVAL** - Command to set the semaphore value.
    - **1** - The initial value to set.

- **gettimeofday(&start, NULL); :** Records the current time before the sorting begins.

- **Parameters**:
  - **&start:** A pointer to a struct timeval that will hold the time value. This structure has two fields:
    - **tv_sec:** The number of seconds since the Epoch (00:00:00 UTC, January 1, 1970).
    - **tv_usec:** The number of microseconds (millionths of a second) since the last second.
  - **NULL:** The second parameter can be used to specify the timezone, but it is typically set to NULL as the timezone is not needed.

- **gettimeofday(&end, NULL); :**Records the current time immediately after the sorting is completed.

- **Parameters**:
  - **&end:** A pointer to a struct timeval that will hold the time value after the sorting is done.
  - **NULL:** The second parameter is set to NULL as before.

- **elapsed = (end.tv_sec - start.tv_sec) * 1.0 + (end.tv_usec - start.tv_usec) / 1000000.0;** :Calculates the total time taken by the quicksort function to sort the array.

- **Parameters**:
  - end.tv_sec - start.tv_sec: Subtracts the start time's seconds from the end time's seconds, giving the difference in seconds.
  - (end.tv_usec - start.tv_usec) / 1000000.0: Subtracts the start time's microseconds from the end time's microseconds, giving the difference in microseconds, and divides by 1,000,000 to convert it to seconds.
  - elapsed: The result is stored in the variable elapsed, which represents the total elapsed time in seconds as a floating-point number.

- **Zombie Process:** A **zombie process** is a process that has completed its execution but still has an entry in the process table. This situation occurs because the process has terminated, but its parent process has not yet called wait() or waitpid() to read its exit status.
- **wait():** The wait() function makes the calling process (usually the parent process) wait until one of its child processes terminates or stops. Once a child process has terminated, wait() will return the PID of the child and also provide its termination status.

❑ **Prototype: pid_t wait(int *status);**

❖**Parameters:**

- **status**: A pointer to an integer where the exit status of the terminated child process will be stored. This status can be examined using macros like WIFEXITED(status), WEXITSTATUS(status), etc. If status is NULL, the status information is ignored.

➢ **waitpid():** It is a more flexible version of wait(). It allows the parent process to wait for a specific child process, to wait for any child process, or to use options to control the behavior of waiting.

❑ **Prototype: pid_t waitpid(pid_t pid, int *status, int options);**

❖ **Parameters:**

- **pid**: Specifies the PID of the child to wait for.
- **status**: Similar to the status parameter in wait(), this is a pointer to an integer where the exit status of the terminated child process will be stored.
- **options**: Provides additional options to control the behavior of waitpid().