SSN COLLEGE OF ENGINEERING, KALAVAKKAM

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Operating Systems Lab – CS2257

Lab Exercise I **STUDY OF SYSTEM CALLS**

A *system call*, sometimes referred to as a *kernel call*, is a made via a software interrupt by an *active process* for a *service* performed by the kernel.

PROCESS MANAGEMENT SYSTEM CALLS

1. System Call : fork( )

Description:

System call **fork()** is used to create processes. It takes no arguments and returns a process ID. The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller. After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.

fork() returns a positive value, the ***process ID*** of the child process, to the parent. The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer.

2. System Call : exec( )

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const  char  *path,  const  char  *arg  , .., char
                                          * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Description:

The exec family of functions replaces the current process image with a new process image. Commonly a process generates a child process because it would like to transform the child process by changing the program code the child process is executing. If successful, the ***exec*** system calls do not return to the invoking program as the calling image is lost.The exec() functions replace a current process with another created according to the arguments given.

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.
- In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

Example:

```
execl("/bin/date","",NULL);  // since the second argument is the
program name,
                             // it may be null

execl("/bin/date","date",NULL);

execlp("date","date",  NULL);  //uses the PATH to find date, try:
%echo $PATH
```

3 & 4. System Call : getpid( ) , getppid( )

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

pid_t getpid(void);
pid_t getppid(void);
```

getpid() returns the process id of the current process. The process ID is a unique positive integer identification number given to the process when it begins executing.

getppid() returns the process id of the parent of the current process. The parent process forked the current child process.

5. System Call : exit( )

```
void exit(int status);
```

Description:

The C library function `exit()` calls the kernel system call `_exit()` internally. The kernel system call `_exit()` will cause the kernel to close descriptors, free memory, and perform the kernel terminating process clean-up. The C library function `exit()` call will flush I/O buffers and perform aditional clean-up before calling `_exit()` internally. The function `exit(status)` causes the executable to return "status" as the return code for `main()`. When `exit(status)` is called by a child process, it allows the parent process to examine the terminating status of the child (if it terminates first). Without this call (or a call from `main()` to `return()`) and specifying the status argument, the process will not return a value.

6. System Call : wait( )

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

pid_t wait(int *status);
```

Description:

The call wait() can be used to determine when a child process has completed it's job and finished. We can arrange for the parent process to wait untill the child finishes before continuing by calling wait(). wait() causes a parent process to pause untill one of the child processes dies or is stopped. The call returns the PID of the child process for which status information is available. This will usually be a child process which has terminated.

7. System Call : sleep ()

```
#include <unistd.h>
unsigned int sleep(unsigned int seconds);
```

Description:

sleep() makes the current process sleep until *seconds* seconds have elapsed

8. System Call : signal ()

#include <signal.h>

A **signal** is a limited form of inter-process communication
Signals are software generated interrupts that are sent to a process when a event happens

- accept the default signal action (usually death)

signal(SIGINT, SIG_DFL);

- ignore the signal

signal(SIGINT, SIG_IGN);

- install a custom signal handling function

signal(SIGINT, ourfunction);

9. System Call : kill ()

```
int  kill(pid_t pid, int sig)
```

Description:

System call **kill()** takes two arguments. The first, **pid**, is the process ID you want to send a signal to, and the second, **sig**, is the signal you want to send. Therefore, you have to find some way to know the process ID of the other party.

- If the call to **kill()** is successful, it returns 0; otherwise, the returned value is negative.
- Because of this capability, **kill()** can also be considered as a communication mechanism among processes .
- The **pid** argument can also be zero or negative to indicate that the signal should be sent to a group of processes.