CS345
OPERATING SYSTEMS

System calls & Signals

Panagiotis Papadopoulos

panpap@csd.uoc.gr

SYSTEM CALL

When a program invokes a system call, it is interrupted and the system switches to Kernel space. The Kernel then saves the process execution context (so that it can resume the program later) and determines what is being requested. The Kernel carefully checks that the request is valid and that the process invoking the system call has enough privilege. For instance some system calls can only be called by a user with superuser privilege (often referred to as root).

If everything is good, the Kernel processes the request in Kernel Mode and can access the device drivers in charge of controlling the hardware (e.g. reading a character inputted from the keyboard). The Kernel can read and modify the data of the calling process as it has access to memory in User Space (e.g. it can copy the keyboard character into a buffer that the calling process has access to)

When the Kernel is done processing the request, it restores the process execution context that was saved when the system call was invoked, and control returns to the calling program which continues executing.
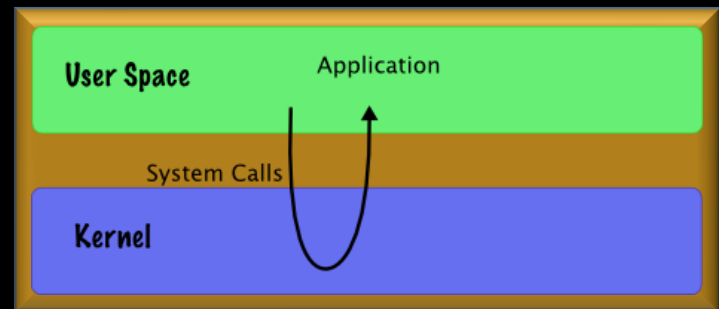
SYSTEM CALLS
FORK()

THE `FORK()` SYSTEM CALL (1/2)

- A process calling `fork()` spawns a child process.

- The child is almost an identical *clone* of the parent:

    - Program Text (segment .text)

    - Stack (ss)

    - PCB (eg. registers)

    - Data (segment .data)

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

pid_t fork(void);
```

THE `FORK()` SYSTEM CALL (2/2)

- The `fork()` is one of the those system calls, which is called once, but returns twice!

- After `fork()` both the parent and the child are executing the same program.

- pid<0: the creation of a child process was unsuccessful.

- pid==0: to the newly created child process.

- pid>0: the *process ID* of the child process, to the parent.
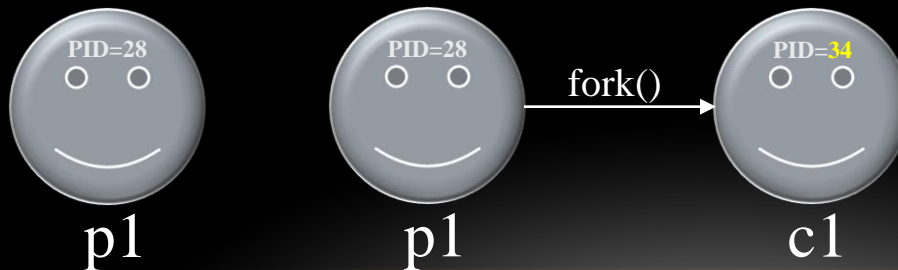
```
Consider a piece of program

...
pid_t pid = fork();
printf("PID: %d\n", pid);
...


The parent will print:
PID: 34
And the child will always print:
PID: 0
```

PID=28
p1

PID=28
p1

fork()

PID=34
c1

EXAMPLE

- Look at simpfork.c:

```
void main() {
        int i;
        printf("simpfork: pid = %d\n", getpid());
         i = fork();
         printf("Did a fork.  It returned %d.  getpid = %d.  getppid = %d\n", i, getpid(), getppid());
}
```

- When it is run, the following happens:

```
simpfork: pid = 914
Did a fork.  It returned 915.  getpid = 914.  getppid = 381
Did a fork.  It returned 0.  getpid = 915.  getppid = 914
```

When simpfork is executed, it has a pid of 914. Next it calls **fork()** creating a duplicate process with a pid of 915. The parent gains control of the CPU, and returns from **fork()** with a return value of the 915 -- this is the child's pid. It prints out this return value, its own pid, and the pid of C shell, which is still 381.

Note, there is no guarantee which process gains control of the CPU first after a **fork()**. It could be the parent, and it could be the child.

SYSTEM CALLS
EXEC()

THE `EXEC()` SYSTEM CALL (1/2)

- The `exec()` call replaces a current process' image with a new one (i.e. loads a new program within current process).

- The new image is either regular executable binary file or a shell script.

- There's not a syscall under the name `exec()`. By `exec()` we usually refer to a family of calls:
    - int execl(char *path, char *arg, ...);
    - int execv(char *path, char *argv[]);
    - int execle(char *path, char *arg, ..., char *envp[]);
    - int execve(char *path, char *argv[], char *envp[]);
    - int execlp(char *file, char *arg, ...);
    - int execvp(char *file, char *argv[]);

- Here's what l, v, e, and p mean:
    - **l** means an argument list,
    - **v** means an argument vector,
    - **e** means an environment vector, and
   - **p** means a search path.

THE `EXEC()` SYSTEM CALL (2/2)

- Upon success, `exec()` <u>never</u> returns to the caller. A successful exec replaces the current process image, so it cannot return anything to the program that made the call. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.

- Arguments passed via `exec()` appear in the `argv[]` of the `main()` function.

- As a new process is not created, the process identifier (PID) does not change, but the machine code, data, heap, and stack of the process are replaced by those of the new program.
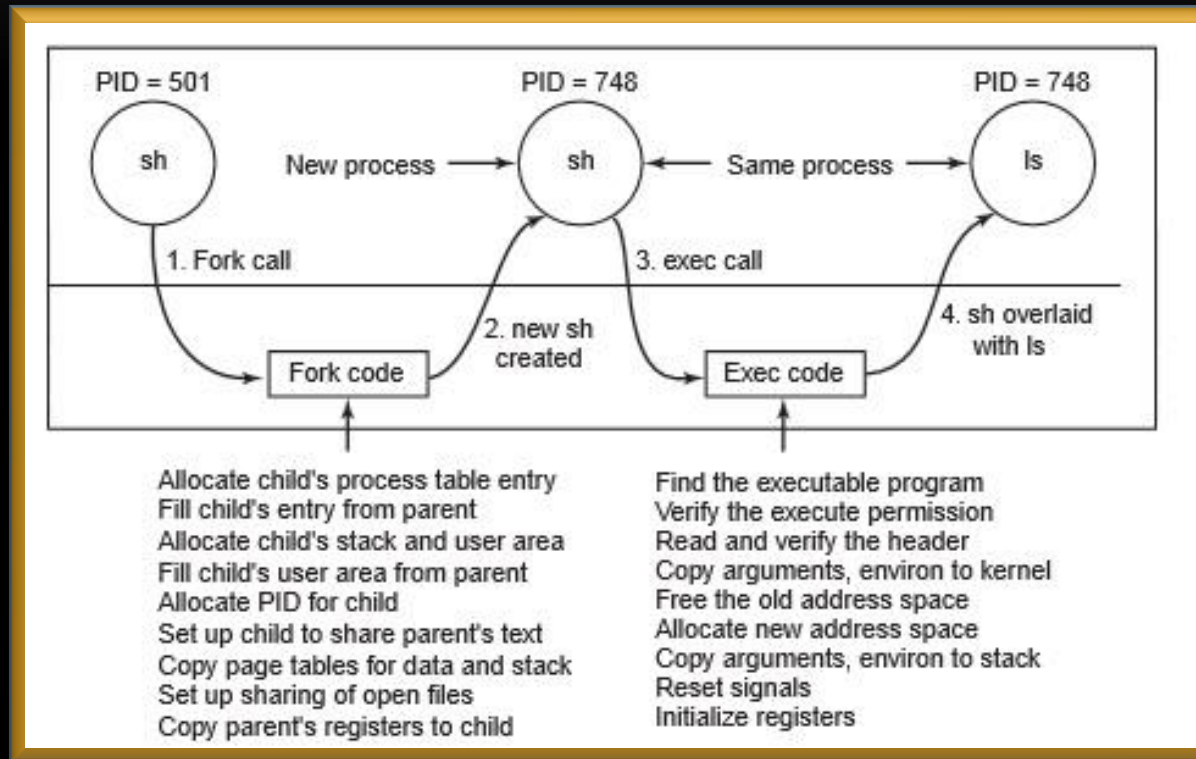
- For more info: `man 3 exec;`

PID=28

PID=28

exec()

p1   p1

Legend:

Old Program

New Program

FORK() AND EXEC() COMBINED (1/2)

- Often after doing `fork()` we want to load a new program into the child. *E.g.*: a shell.
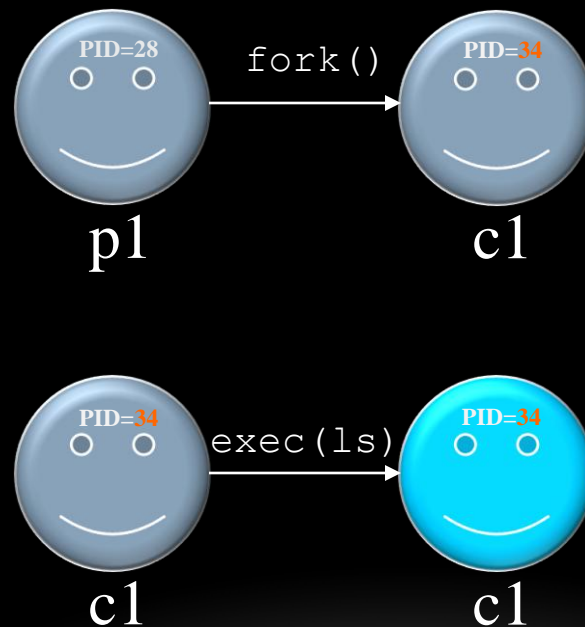
FORK() AND EXEC() COMBINED (2/2)

SYSTEM CALLS

WAIT(), PAUSE(), EXIT()

THE SYSTEM `WAIT()` CALL (1)

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate* is more appropriate terminology, but a bit less common).

- When the child process terminates, it returns an exit status to the operating system, which is then returned to the waiting parent process. The parent process then resumes execution.

- A child process that terminates but is never waited on by its parent becomes a zombie process. Such a process continues to exist as an entry in the system process table even though it is no longer an actively executing program.(*Such situations are typically handled with a special "reaper" process that locates zombies and retrieves their exit status, allowing the operating system to then de-allocate their resources.*)

THE SYSTEM `WAIT()` CALL (2)

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

- The `wait()` causes the parent to wait for any child process.

- The `waitpid()` waits for the child with specific PID.

- The return value is:

  - PID of the exited process, if no error

  - (-1) if an error has happened

THE `PAUSE()` SYSTEM CALL

```
#include <unistd.h>

int pause(void);
```

- Used to suspend process until a signal arrives

- Signal action can be the execution of a handler function or process termination

- only returns (-1) when a signal was caught and the signal-catching function returned

THE `EXIT()` SYSTEM CALL

```
#include <stdlib.h>

void exit(int status);
```

- This call gracefully terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the zombie state.

- By calling `wait()`, the parent cleans up all its zombie children.

- `exit()` specifies a return value from the program, which a parent process might want to examine as well as status of the dead process.

PROCESS STATES

- Zombie: has completed execution, still has an entry in the process table

- Orphan: parent has finished or terminated while this process is still running

- Daemon: runs as a background process, not under the direct control of an interactive user

# ZOMBIE PROCESS
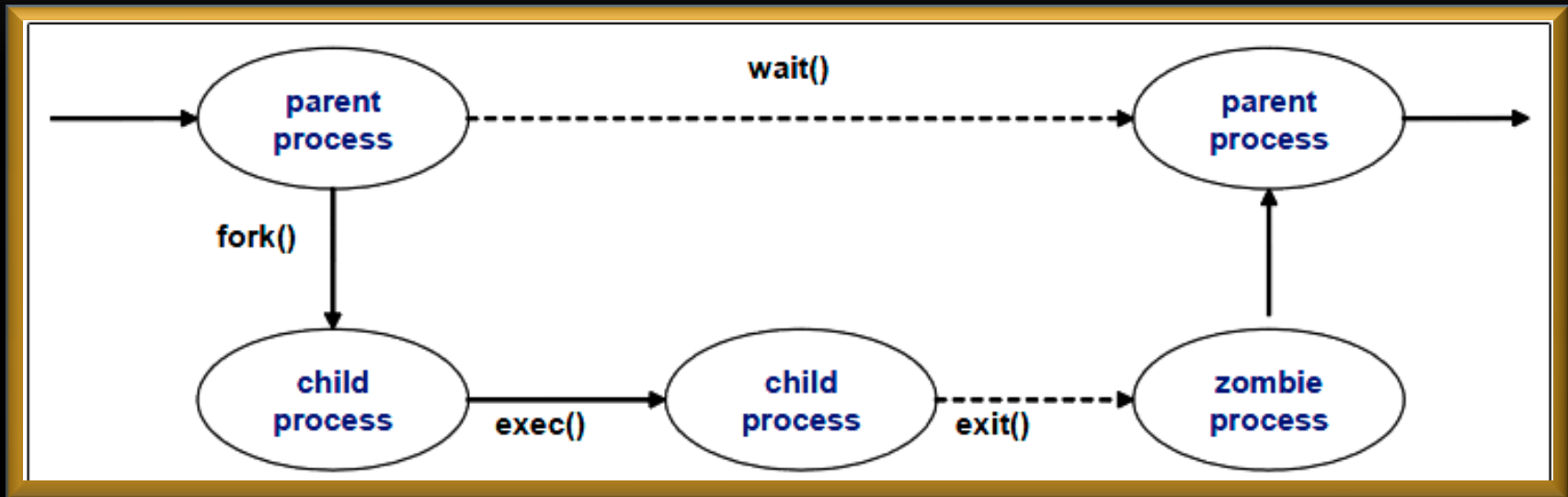
SIGNALS

PROCESS INTERACTION WITH SIGNALS

- A signal is an asynchronous event which is delivered to a process.

- Asynchronous means that the event can occur at any time

  - may be unrelated to the execution of the process

  - e.g. user types ctrl-C, or the modem hangs

- Unix supports a signal facility, looks like a software version of the interrupt subsystem of a normal CPU

- Process can send a *signal* to another - Kernel can send signal to a process (like an interrupt)

- A process can:

  - ignore/discard the signal (not possible with SIGKILL or SIGSTOP)

  - execute a signal handler function, and then possibly resume execution or terminate

  - carry out the default action for that signal
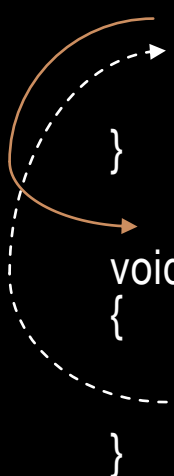
THE `SIGNAL()` SYSTEM CALL

```
#include <signal.h>

void (*signal(int sig, void (*handler)(int))) (int);
```

- The `signal()` system call installs a new signal handler for the signal with number signum. The signal handler is set to sighandler which may be a user specified function
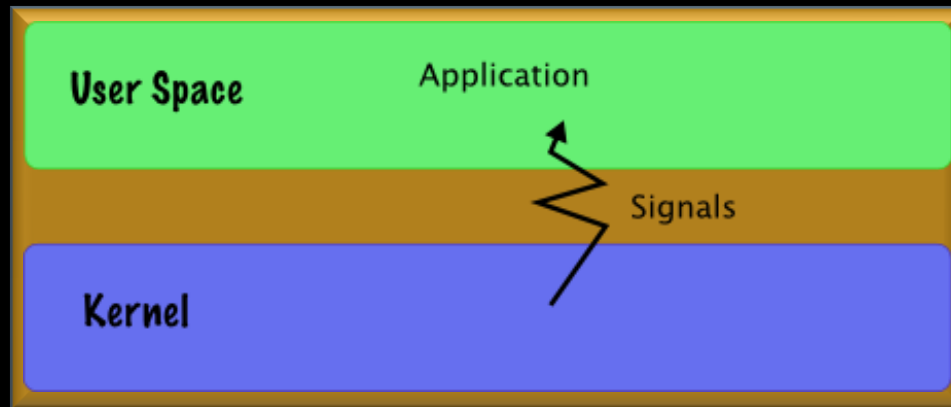
EXAMPLE

```
int main()
{
    signal( SIGINT, foo );
        :

                        /* do usual things until SIGINT */

    return 0;
}

void foo( int signo )
{
        :               /* deal with SIGINT signal */
    return;             /* return to program */
}
```

COMMON SIGNAL NAMES AND NUMBERS

Number	Name	Description	Used for
0	SIGNULL	Null	Check access to pid
1	SIGHUP	Hangup	Terminate; can be trapped
2	SIGINT	Interrupt	Terminate; can be trapped
3	SIGQUIT	Quit	Terminate with core dump; can be
9	SIGKILL	Kill	Forced termination; cannot be trapped
15	SIGTERM	Terminate	Terminate; can be trapped
24	SIGSTOP	Stop	Pause the process; cannot be trapped
25	SIGTSTP	Terminal	stop Pause the process; can be
26	SIGCONT	Continue	Run a stopped process

SENDING A SIGNAL: `KILL()` SYSTEM CALL

- `kill` command is a command that is used to send a signal in order to request the termination of the process. We typically use kill -SIGNAL PID, where you know the PID of the process.

- The `kill()` system call can be used to send any signal to any process group or process.

- `int kill( pid_t pid, int signo );`

- *pid*   *Meaning*

  - \> 0   send signal to process `pid`

  - == 0   send signal to all processes whose process group ID equals the sender's pgid. e.g. parent kills all children

  - -1   send signal to every process for which the calling process has permission to send signals

ALL TOGETHER NOW…
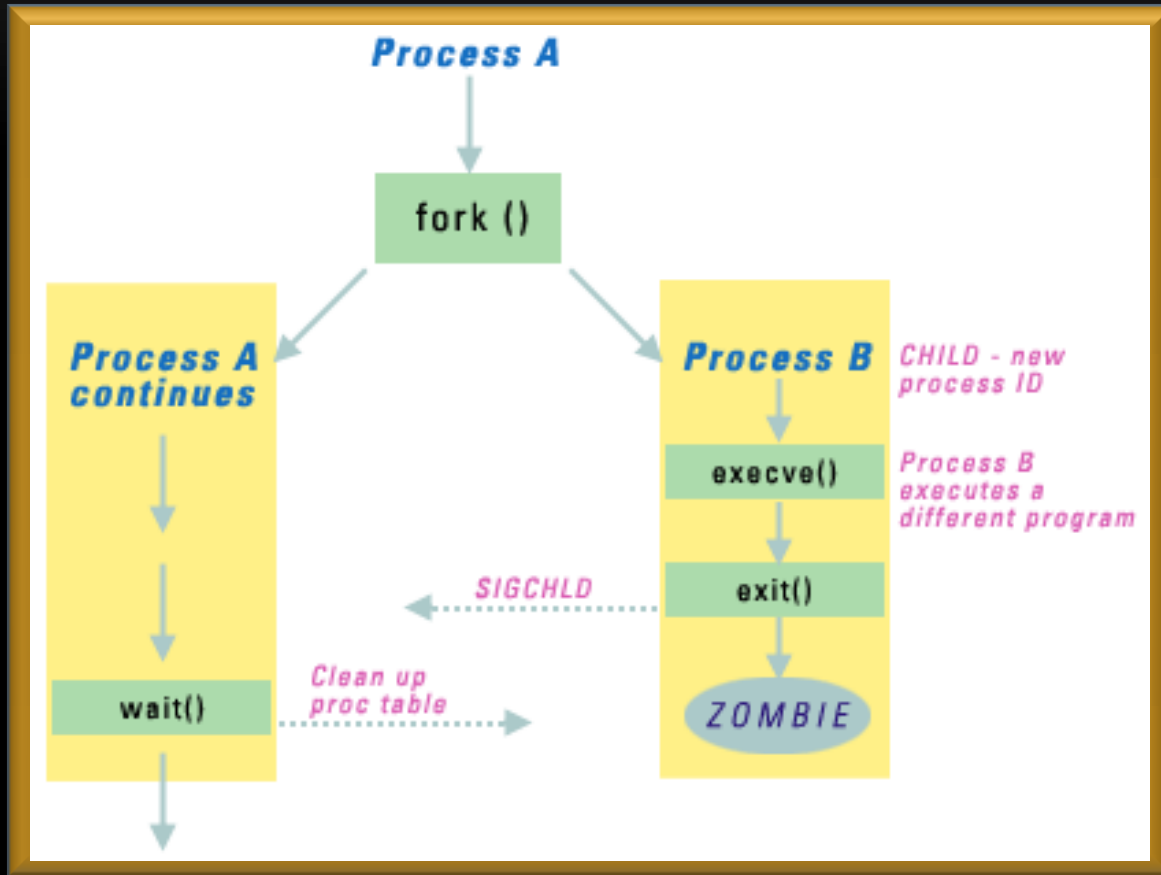
PIPES

OVERVIEW OF PIPES

- Pipes and FIFOs (also known as named pipes) provide a unidirectional interprocess communication channel

- "|" (pipe) operator between two commands directs the stdout of the first to the stdin of the second. Any of the commands may have options or arguments. Many commands use a hyphen (-) in place of a filename as an argument to indicate when the input should come from stdin rather than a file.

  e.g of pipelines:

  - command1 | command2 paramater1 | command3 parameter1 - parameter2 | command4

  - ls -l | grep key | more

CREATING PIPELINES PROGRAMMATICALLY

- Pipelines can be created under program control. The Unix pipe() system call asks the operating system to construct a unidirectional data channel that can be used for interprocess communication, a new anonymous pipe object. This results in two new, opened file descriptors in the process: the read-only end of the pipe, and the write-only end. The pipe ends appear to be normal, anonymous file descriptors, except that they have no ability to seek.

```c
void main(int argc, char *argv[]){
    int pipefd[2];
    pid_t cpid;
    char buf;
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) {                        /* Child reads from pipe */
        close(pipefd[1]);                   /* Close unused write end */
        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        exit(EXIT_SUCCESS);
    } else {                                /* Parent writes argv[1] to pipe */
        close(pipefd[0]);                   /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);                   /* Reader will see EOF */
        wait(NULL);                         /* Wait for child */
        exit(EXIT_SUCCESS);
    }
```

TIME

TIME

- `time` is a command that is used to determine the duration of execution of a particular command. It writes a message to standard error that lists timing statistics. The message includes the following information:

    - The elapsed (real) time between invocation of *command* and its termination.

    - The User CPU time, equivalent to the sum of the *tms_utime* and *tms_cutime* fields returned by the *times*() function defined in the System Interfaces volume of POSIX.1-2008 for the process in which *command* is executed.

    - The System CPU time, equivalent to the sum of the *tms_stime* and *tms_cstime* fields returned by the *times*() function for the process in which *command* is executed.

TIMES()

- `times()` gets process and waited-for child process times

- `times()` function shall fill the **tms** structure pointed to by *buffer* with time-accounting information. The **tms** structure is defined in *<sys/times.h>*.
```
clock_t times(struct tms *buffer);
struct tms {
        clock_t tms_utime;  /* user time */
        clock_t tms_stime;  /* system time */
        clock_t tms_cutime; /* user time of children */
        clock_t tms_cstime; /* system time of children */
};
```
The tms_utime field contains the CPU time spent executing instructions of the calling process. The tms_stime field contains the CPU time spent in the system while executing tasks on behalf of the calling process. The tms_cutime field contains the sum of the tms_utime and tms_cutime values for all waited-for terminated children. The tms_cstime field contains the sum of the tms_stime and tms_cstime values for all waited-for terminated children.

EXAMPLE

```
static clock_t st_time;
static clock_t en_time;
static struct tms st_cpu;
static struct tms en_cpu;

void start_clock(){
    st_time = times(&st_cpu);
}

void end_clock(char *msg){
    en_time = times(&en_cpu);
    fputs(msg,stdout);
    printf("Real Time: %jd, User Time %jd, System Time %jd\n", (intmax_t)(en_time - st_time),
              (intmax_t)(en_cpu.tms_utime - st_cpu.tms_utime),(intmax_t)(en_cpu.tms_stime - st_cpu.tms_stime));

}
```

ASSIGNMENT 1 TIPS

1. First experiment with `fork()` and `getpid(),getppid()`

2. Use simple `printf` statements to distinguish parent from child (through pid)

3. Send simple signal to child

4. Create signal handlers

5. Create logic for alternating execution

6. Read the following man pages: `fork(2), exec(3), execv(3), wait(2), waitpid(2), pipe(2), dup2(2), times(2), time(1), sh(1), bash(1), gettimeofday(2), signal(2), chdir(2), getcwd(2), getlogin(2)`

USEFUL LINKS

- http://web.eecs.utk.edu/~huangj/cs360/360/notes/Fork/lecture.html

- http://linuxprograms.wordpress.com/category/pipes/

- http://man7.org/linux/man-pages/man2/pipe.2.html

- http://man7.org/linux/man-pages/man7/signal.7.html

- http://www.cis.temple.edu/~giorgio/cis307/readings/signals.html

- http://ph7spot.com/musings/introduction-to-unix-signals-and-system-calls

- http://man7.org/linux/man-pages/man1/time.1.html

- http://unixhelp.ed.ac.uk/CGI/man-cgi?times+2

- http://www.cs.uga.edu/~eileen/1730/Notes/signals-UNIX.ppt

QUESTIONS?