CSI 402 – Lecture 13
(Unix – Process Related System Calls)
**Ref:** Chapter 5 of [HGS].

**Process:**

- A program in execution.

- Several processes are executed concurrently by the scheduler.

- Each process has a unique ID (called process ID or pid).

- When a process P is created, there is a parent process for P.  
  *(Note: Process with pid zero is its own parent.)*
Useful Shell commands:

- ps: Gives the list of processes that are currently running.
- kill: Command to kill one or more processes.

Example: Suppose the ps command shows that processes with IDs 1274 and 1297 are running. To kill these processes, the shell command is the following:

```
% kill -9 1274 1297
```
System calls for obtaining pid:

- **Prototypes:**
  
  ```c
  pid_t getpid (void)
  pid_t getppid (void)
  ```

- **Headers:** `<sys/types.h>` and `<unistd.h>`.

- The type `pid_t` is usually unsigned long.

- **getpid:** Returns the pid of the process.

- **getppid:** Returns parent’s pid.

- No error exit for either function.
Sample Program:

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

int main(void) {
    printf("Pid = %ld\n", getpid());
    printf("Parent’s Pid = %ld\n", getppid());
    return 0;
} /* End of main. */
System call fork:

- **Prototype:** `pid_t fork (void)`
- **Headers:** `<sys/types.h>` and `<unistd.h>`.
- Creates a new process by copying the parent's memory image.
- **Both** processes continue to execute after the call to `fork`.
- Returns zero to child and the pid of the child to the parent.
- Another system call (one from the family of `exec` system calls) is used to make parent and child execute different programs.
- Returns -1 in case of failure.
(Bad) Example:

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

int x = 0; fork();
x = 1;
```

After the call to `fork`, there are two *independent* processes.

Each process has its own location for the variable `x`.

**Better Example:** Handout 13.1.
**Failure of fork call:**

- The total number of processes in the system exceeds a preset limit or the total number of processes for the user exceeds a preset limit.

- No child process is created when `fork` fails.

- The value of `errno` is `EAGAIN`. 
System calls **getuid** and **geteuid**:

- **Prototypes:**
  
  ```c
  uid_t getuid (void)
  uid_t geteuid (void)
  ```

- **Headers:** `<sys/types.h>` and `<unistd.h>`.

- In addition to pid, each process has a (real) user id and an **effective** user id.

- **getuid:** Returns the (real) user id of the process.

- **geteuid:** Returns effective user id of the process. *(Recall: Setuid bit for executables.)*

- No error exit for either function.
Additional notes about processes:

- A child process *inherits* parent’s privileges and resources such as files.

- The child process competes for the CPU along with the parent.

- There are situations where the parent *waits* for the child to complete.

  **Common example:** Shell

  - Each shell command, except the `cd` command, is executed by a child process of the shell.
System Call \texttt{wait}:

- **Prototype:** \texttt{pid_t \ wait (int *estatus)}
- **Headers:** \texttt{<sys/types.h>} and \texttt{<sys/wait.h>}
- Causes the caller to wait until some child terminates: one form of \textit{synchronization}.
- Normally, returns the pid of the child that terminated.
- If no child is waiting, the call returns \texttt{-1} and \texttt{errno} has the value \texttt{ECHILD}.
- If \texttt{estatus} is \texttt{NULL}, it is ignored; otherwise, the exit status of terminating child is returned in \texttt{*estatus}. (The exit status is \texttt{0} if the child terminated normally; nonzero otherwise.)

**Program Example:** Handout 13.2.
System call *waitpid*:

- **Prototype:**

  ```c
  pid_t waitpid (pid_t pid, int *estatus, int options)
  ```

- **Headers:** Same as `wait`.

- Causes the caller to wait until the child with id given by `pid` terminates.

- If `pid` is -1 and `options` is 0, then `waitpid` behaves exactly like `wait`.

- Most common value for `options` is `WNOHANG`. In that case, if the specified child is still running, the call returns 0 and the caller does *not* wait.

- Helpful when the parent process wants to perform some actions while the specified child is running.
Reading Assignment: Program example on page 107 of [HGS].

Two Special Processes:

- The swapper process: Pid = 0; the swapper is its own parent.
- The init process: Pid = 1; its parent is the swapper process.

Orphan Process:

- A process which is still running but whose parent has terminated.
- Doesn’t stay an orphan for too long.
- Orphan processes are “adopted” by the init process.
**Zombie Process:**

- Dictionary meaning of “zombie”: One who seems more dead than alive.

- In Unix, a **zombie** process is one which has terminated *before* its parent had a chance to wait for it.

**Example Code Segment:**

```c
if ((cid = fork()) == 0) {
    -- code for child --
}
else {
    -- parent with many lines of code--
    c = wait(&status);
}
```

**Note:** Child may exit before parent reaches the `wait` call; that is, the child may become a zombie process.
System Calls for Processes (continued)

**Why are zombies bad?**

- Kernel maintains a process table, with one entry per process. (The size of this table is the maximum number of processes allowed in the system.)

- When a process P terminates, the exit status of P must be conveyed to P’s parent.

- The parent may be going through a long program before waiting for the child.

- So, some information about process P must be kept in the process table even though P can’t execute anymore.

- The process table entry given to P can’t be given to another process until P is “completely dead” (i.e., the exit status of P has been given to P’s parent).
The **exec family of system calls:**

- Used in conjunction with `fork` to create processes executing different code.

- Traditional way: Child executes an appropriate `exec` call.

- Two sets of calls: `execl` and `execv`.

  - `execl`: Used when the command line arguments are known at compile time and can be passed as a list.

  - `execv`: Used to pass command line arguments as an array (similar to `argv[]`).

- Commonly used forms: `execlp` and `execvp`.

- The ’p’ suffix indicates that the call will search the directories in the PATH environment variable.
System calls `execlp` and `execvp`:

- **Prototypes:**
  
  ```c
  int execlp( const char *prog, const char *arg0, ..., const char *argn, NULL )
  
  int execvp( const char *prog, const char *argv[] )
  ```

- **Header:** `<unistd.h>`.

- Note that `execlp` has a *variable* number of arguments; the NULL pointer indicates the end of the list.

- Using exec is *different* from usual function calls; in particular, a call to exec should *not* return if there are no errors.

**Program Examples:** Handouts 13.3 and 13.4.