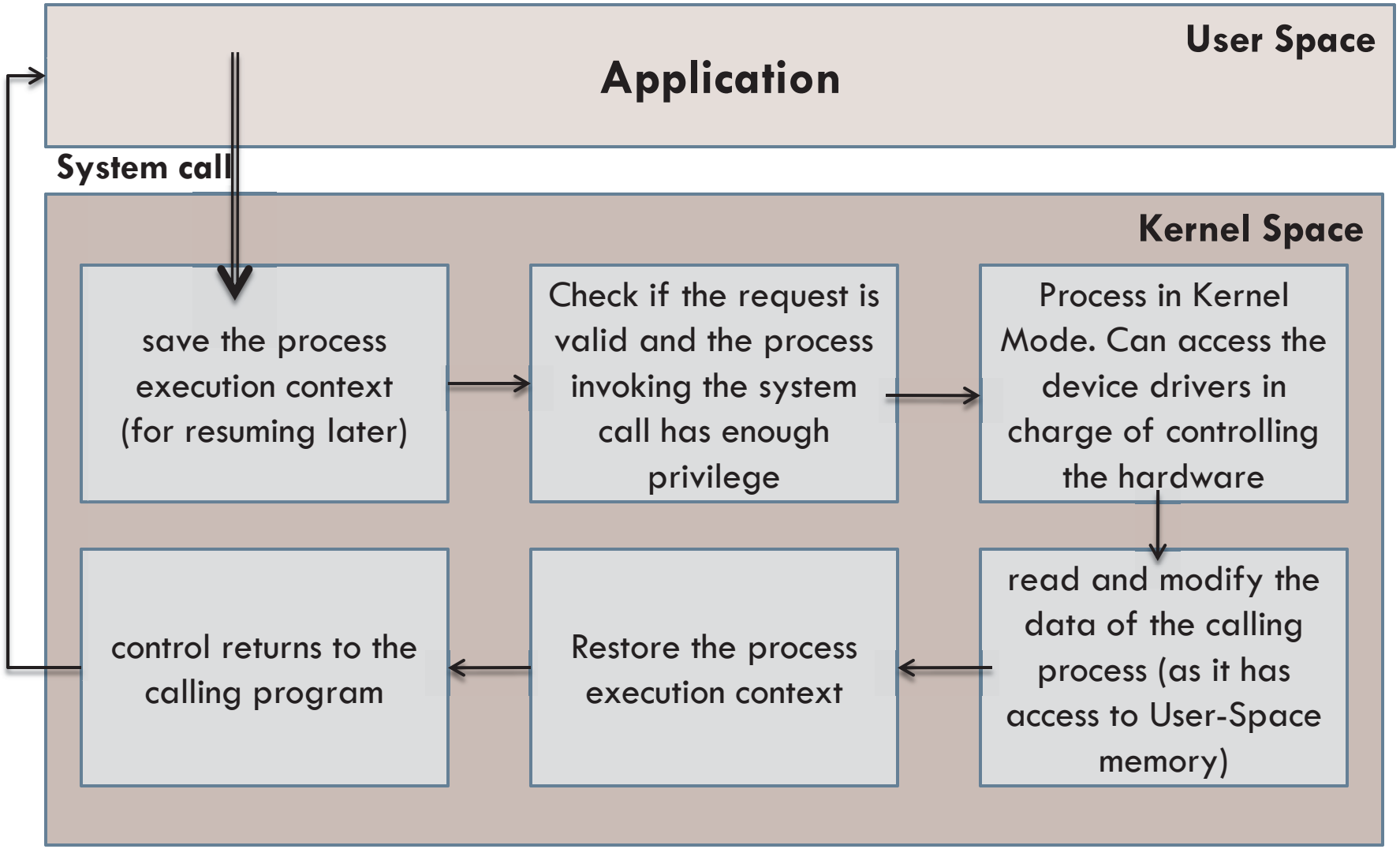
CS-345 | ASSIGNMENT 1

Implementation of simple C shell : "cs345sh"

System Calls

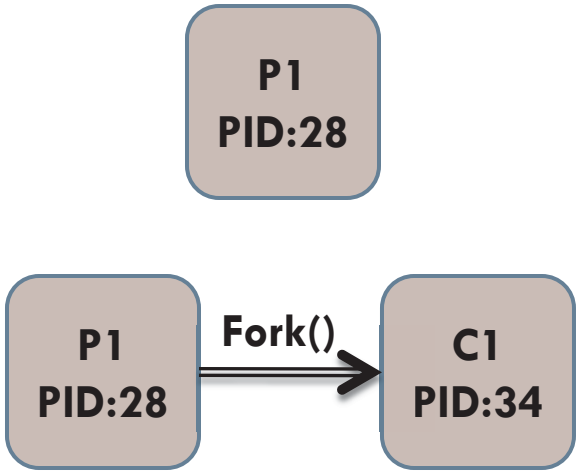


The “fork()” system call

- 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)
- The `fork()` is called once, but returns twice!
- After `fork()` both the parent and the child are executing the same program.

The “fork()” system call - PID

- pid<0: the creation of a child process was unsuccessful.
- pid==0: the newly created child.
- pid>0: the *process ID* of the child process passes 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
```

"fork()" Example

```
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());
}
```

Returns:

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 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.

The "exec()" System Call

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

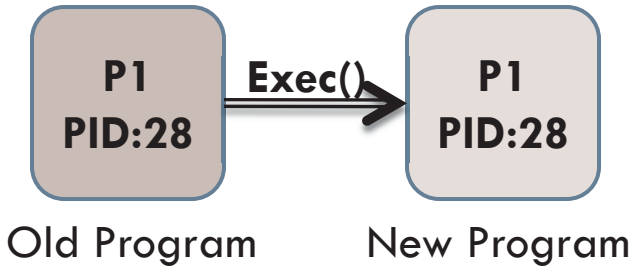
- The new image is either regular executable **binary file** or a **shell script**.

- 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[]);

Where l=argument list, v=argument vector, e=environmental vector, and p=search path.

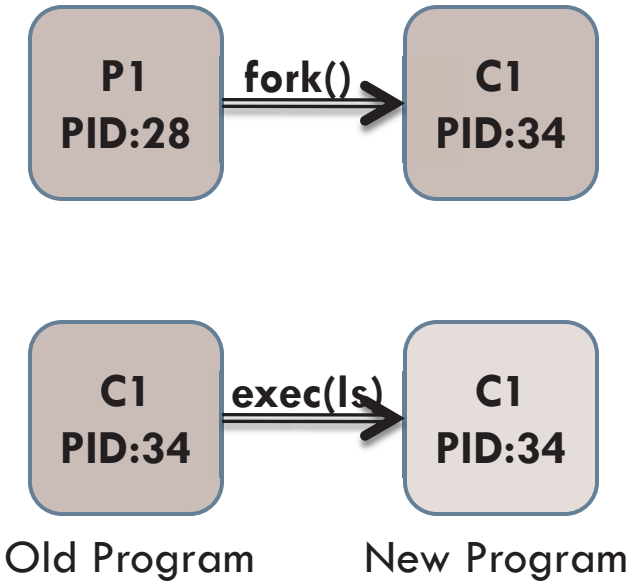
The "exec()" System Call

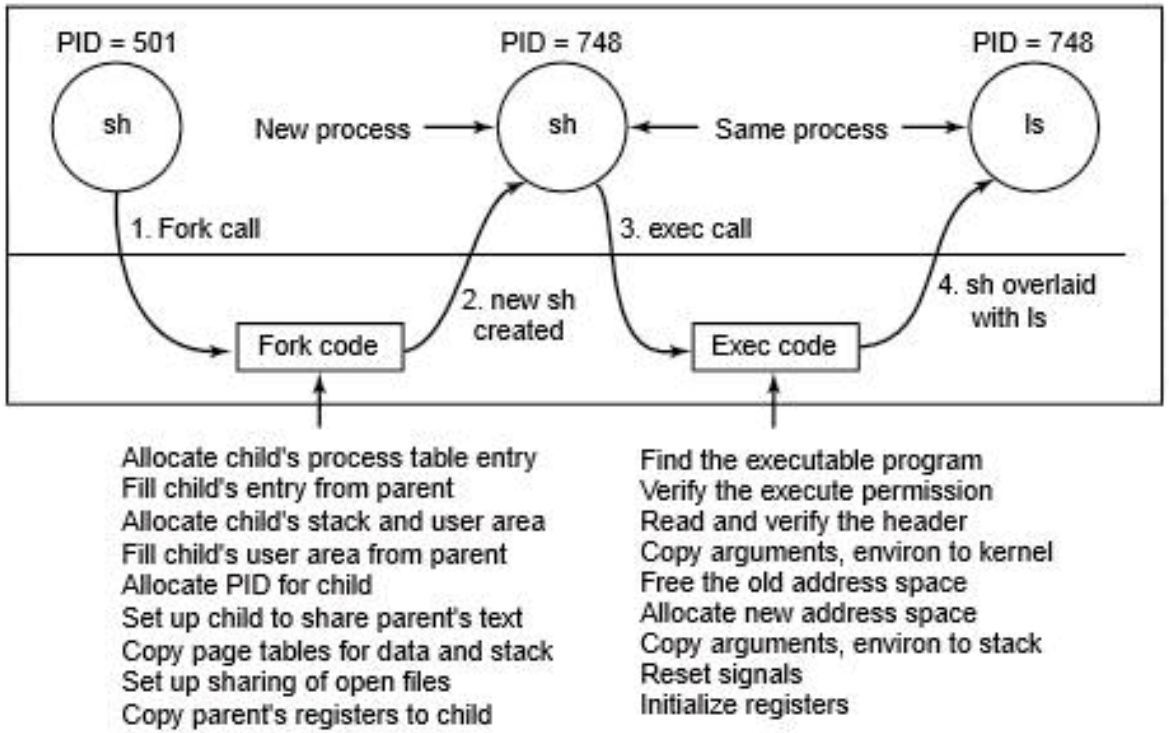
- Upon success, `exec()` never returns to the caller. It replaces the current process image, so it cannot return anything to the program that made the call. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
- Arguments passed via `exec()` 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;`



“fork()” and “exec()” combined

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





The “wait()” system call

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate).*
- When the child process dies, 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 dies 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.

The “wait()” system call

```
#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.
  - pid: pid of (child) process that the calling process waits for.
  - status: a pointer to the location where status information for the terminating process is to be stored.
  - options: specifies optional actions.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

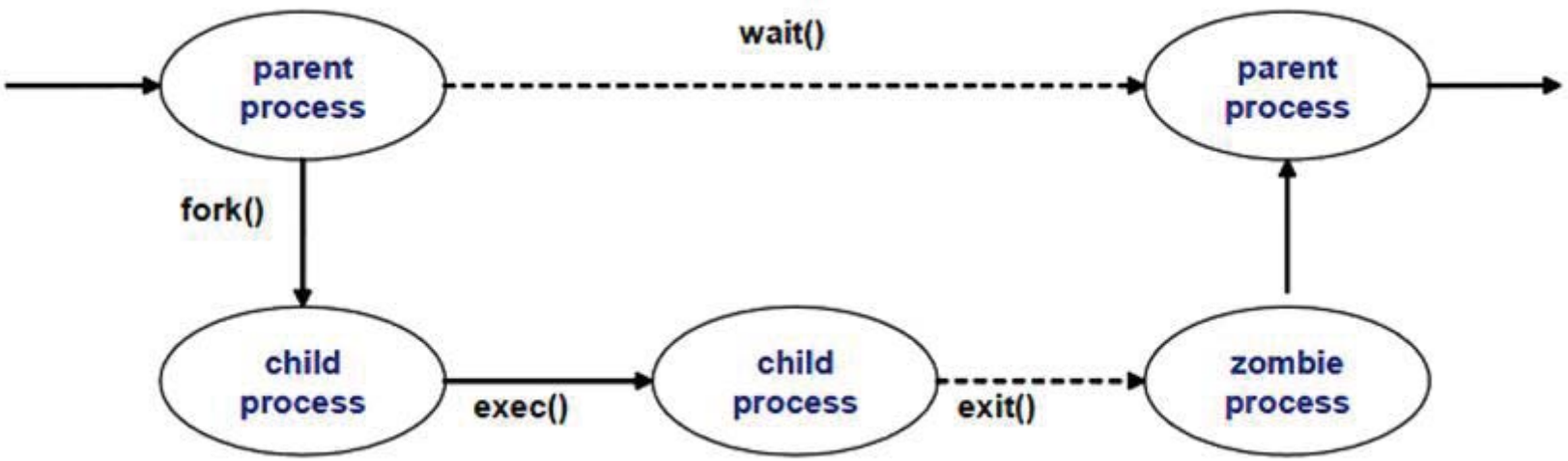
The “exit()” system call

- 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.
- When the child process dies, an exit status is returned to the operating system and a signal is sent to the parent process. The exit status can then be retrieved by the parent process via the ***wait*** system call.

The 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

A zombie process



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

Programming Pipelines

- 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.

```
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);
    }
```

SIGNALS

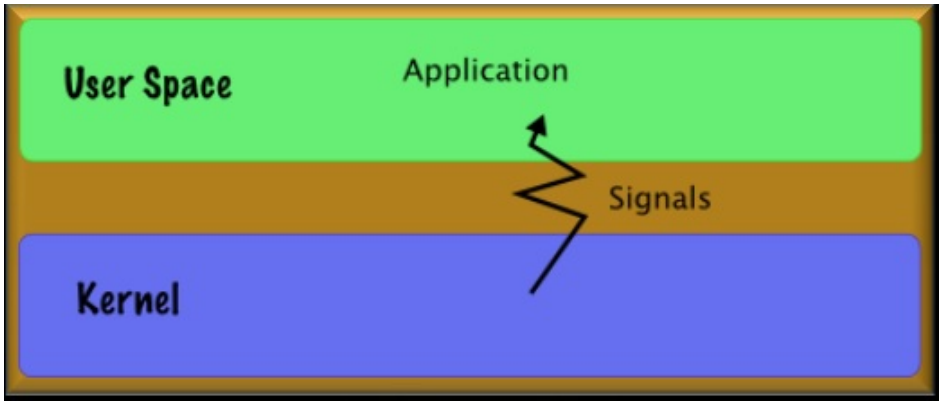
- A signal is an asynchronous event which is delivered to a process.
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types ctrl-C, or the modem hangs
- Unix supports a signal facility, looks like a software version of the interrupt 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(intsig, 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

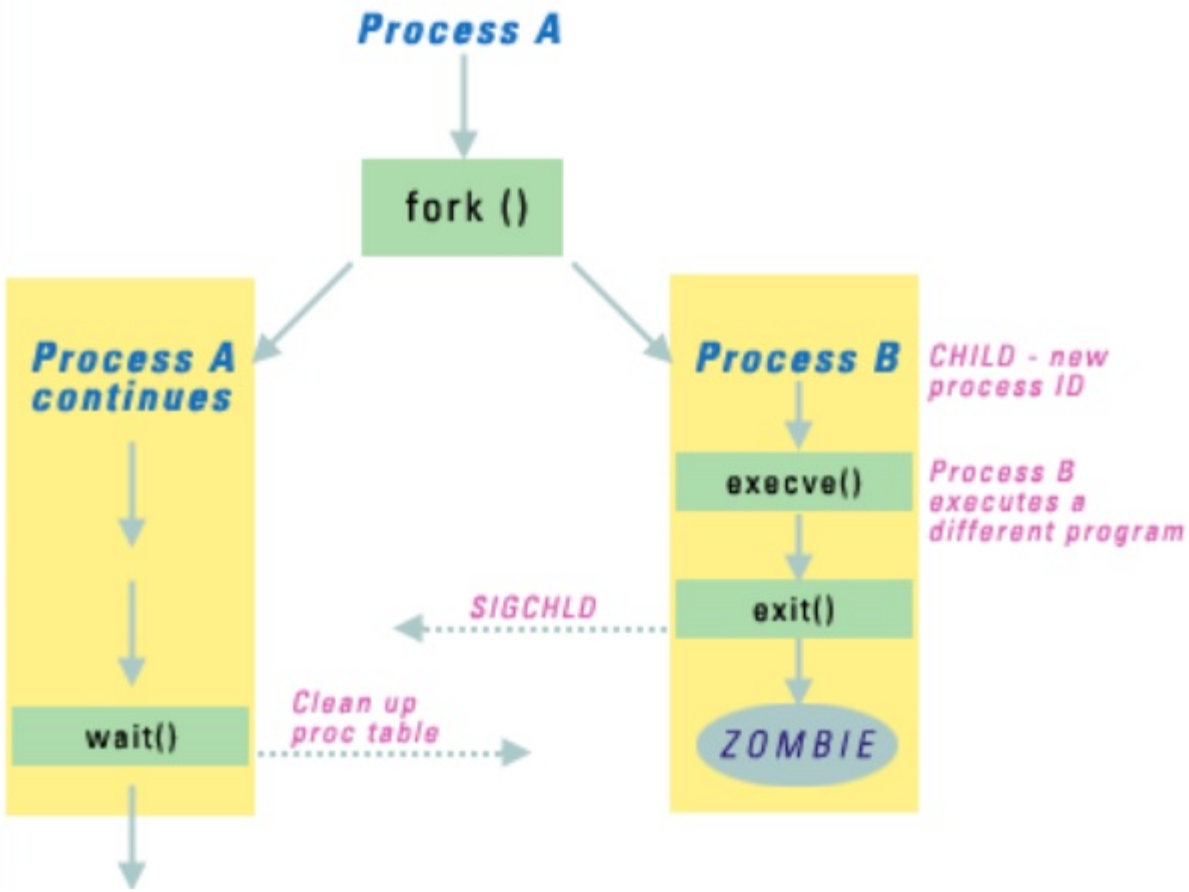
```
int main()
{
    signal( SIGINT, foo );
    …                       /*  do usual things until SIGINT */
    return 0;
}

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

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, here 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



JOB CONTROL

- Job control refers to the ability to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point.

- The shell keeps a table of currently executing jobs, which may be listed with the **jobs** command.

ASSIGNMENT 1

- Το shell θα διαβάζει εντολές από τον χρήστη και θα τις εκτελεί.
- Η έξοδος μιας εντολής θα μπορεί να δίνεται σαν είσοδος σε μια άλλη εντολή που υπάρχει στην ίδια γραμμή εντολών και διαχωρίζονται με το σύμβολο "|" μεταξύ τους.
- Ο χρήστης θα μπορεί να κάνει έλεγχο διεργασιών αναστέλοντας (suspend) και συνεχίζοντας (resume) την εκτέλεση της μιας διεργασίας είτε στο προσκήνιο είτε στο παρασκήνιο.

TIPS

- First experiment with fork() and getpid(), getppid()
- Use simple printf statements to distinguish parent from child (through pid)
- Send simple signal to child
- Create signal handlers

Useful links

- http://web.eecs.utk.edu/~huangj/cs360/360/notes/Fork/lecture.html
- https://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://cis-linux1.temple.edu/~giorgio/cis307/readings/signals.html
- http://ph7spot.com/musings/introduction-to-unix-signals-and-system-calls
- www.cs.uga.edu/~eileen/1730/Notes/signals-UNIX.ppt