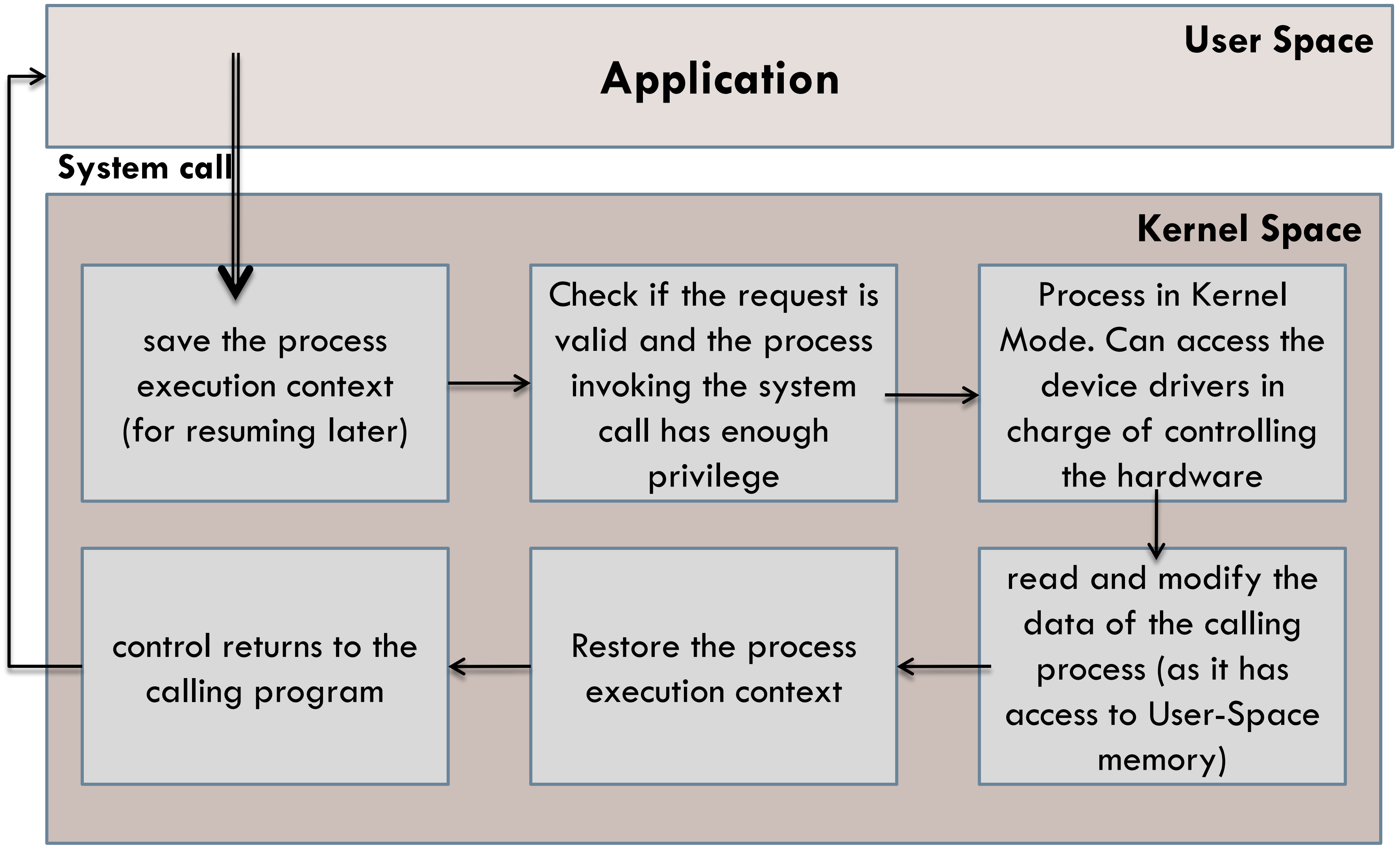
ASSIGNMENT 1

Implementation of Linux C shell : "csd_sh"

System Calls

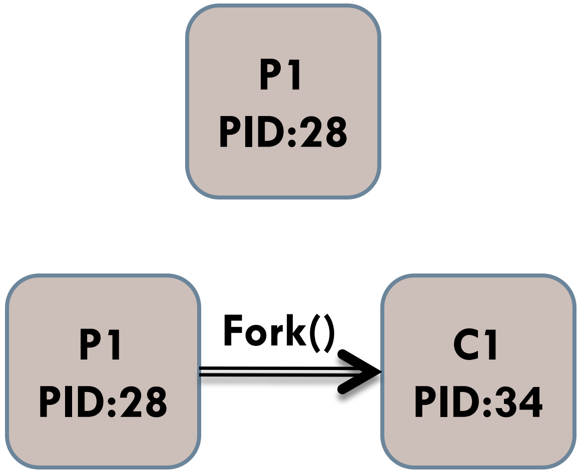

User Space
Application
System call
Kernel Space
save the process execution context (for resuming later)
Check if the request is valid and the process invoking the system call has enough privilege
Process in Kernel Mode. Can access the device drivers in charge of controlling the hardware
read and modify the data of the calling process (as it has access to User-Space memory)
Restore the process execution context
control returns to the calling program

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).

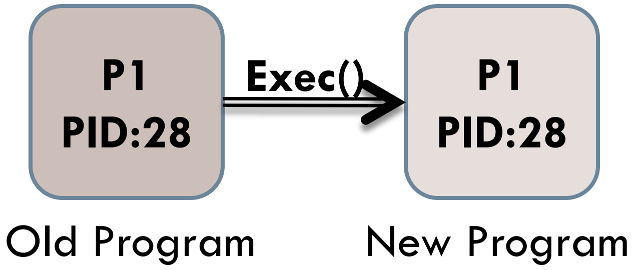
- The new image is either regular executable **binary file** or a **shell script.**

- 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.

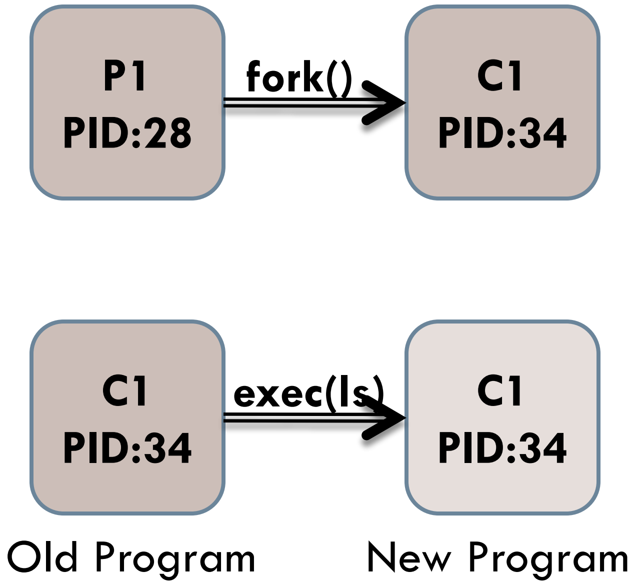
The "exec()" System Call

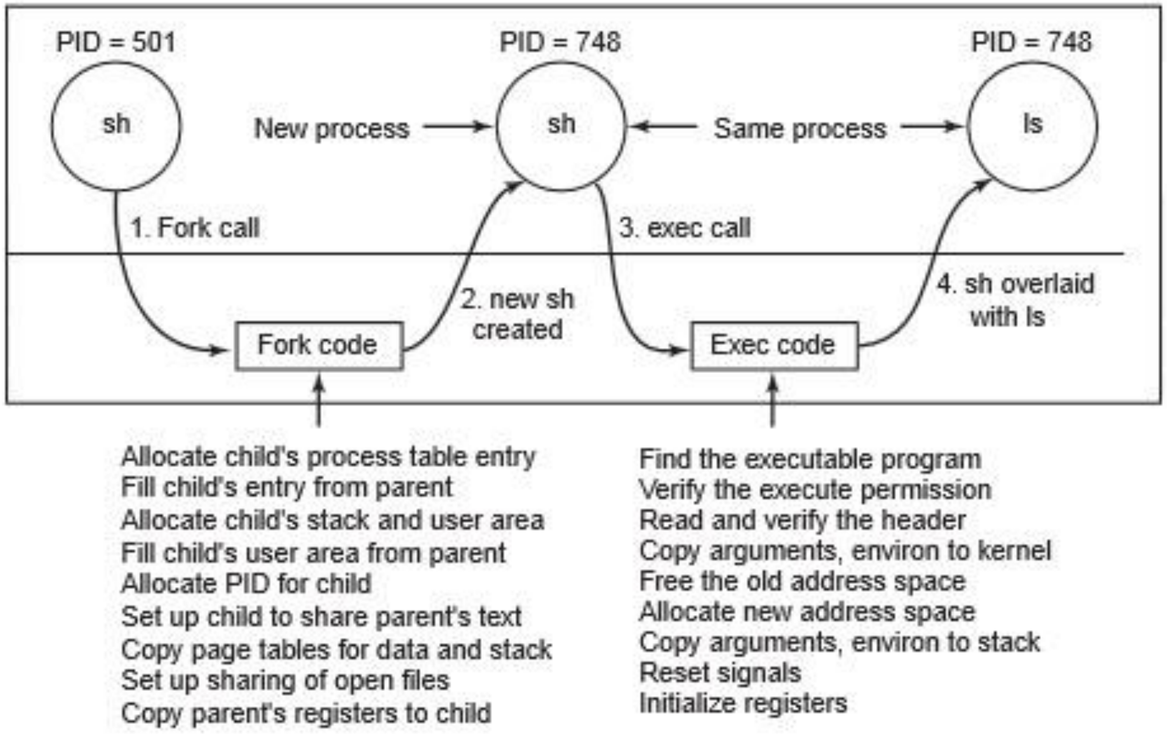
- Upon success, `exec()` never returns to the caller. It replaces the current process image, so it cannot return anything to the program that made the call. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
- Arguments passed via `exec()`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





Allocate child's process table entry
Fill child's entry from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

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

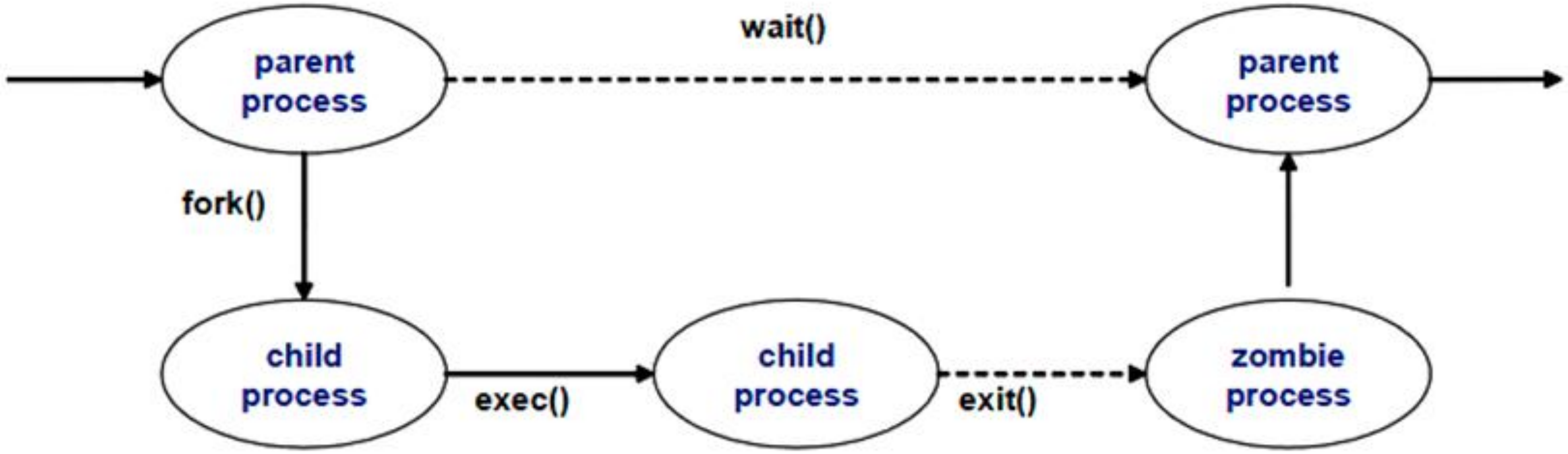
The “exit()” system call

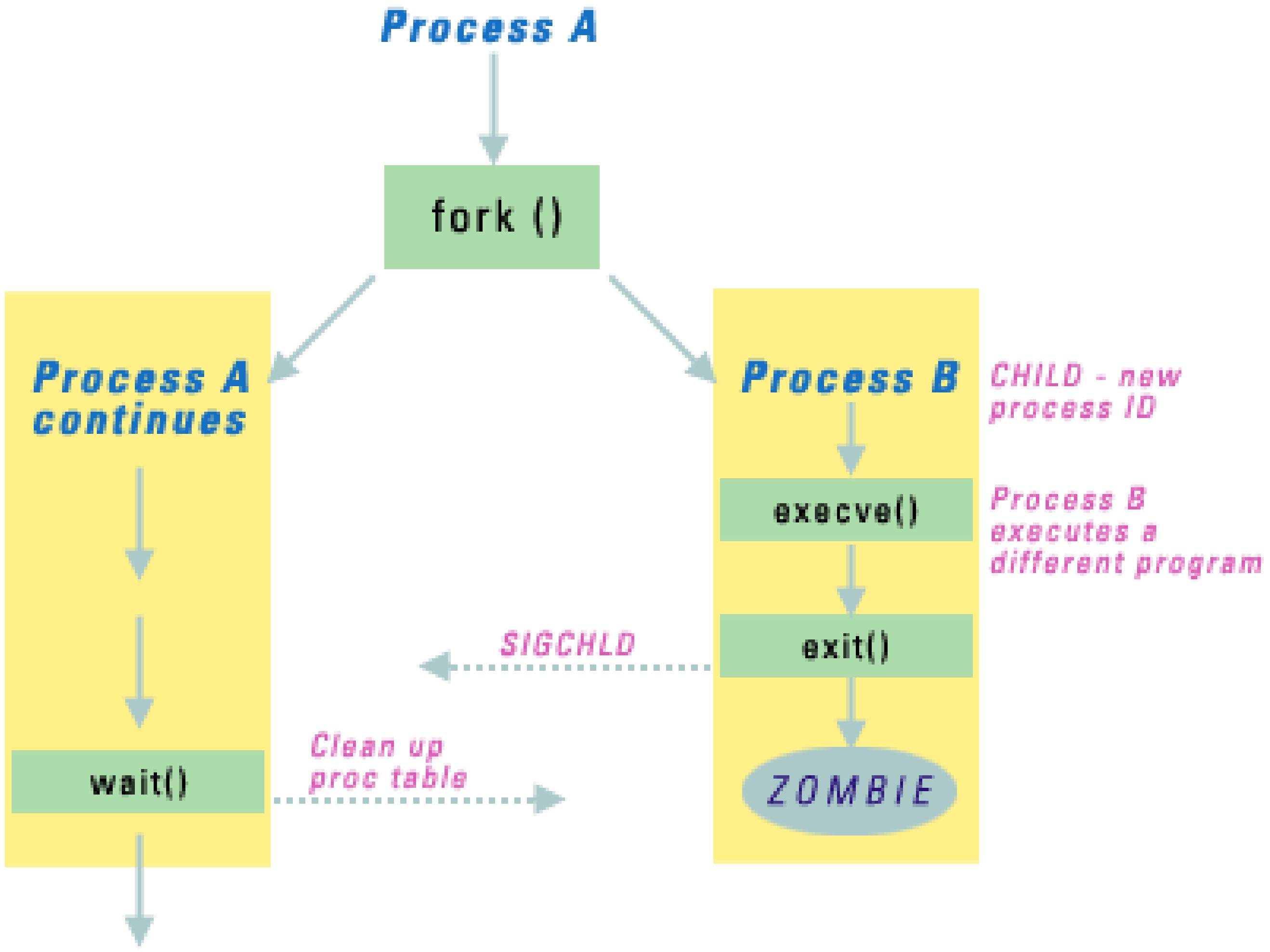
- This call **gracefully** terminates process execution. Gracefully means it does clean up and release of resources, and puts the process into the **zombie state.**
- By calling `wait()`, the parent cleans up all its zombie children.
- When the child process dies, an exit status is returned to the operating 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




Process A
fork ()
Process A continues
wait()
Clean up proc table
Process B
CHILD - new process ID
execve()
Process B executes a different program
exit()
SIGCHLD
ZOMBIE

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);
    }
```

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 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
- It fills 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 */
};
```

Times() 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: %d, User Time %d, System Time %d\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

- Το shell θα διαβάζει εντολές από τον χρήστη και θα τις εκτελεί.
- Ο χρήστης του **csd_sh** θα μπορεί να κατευθύνει την έξοδο/είσοδο μιας εντολής σε ένα αρχείο χρησιμοποιώντας το σύμβολο **'>'** ή αντίστοιχα **'<'**

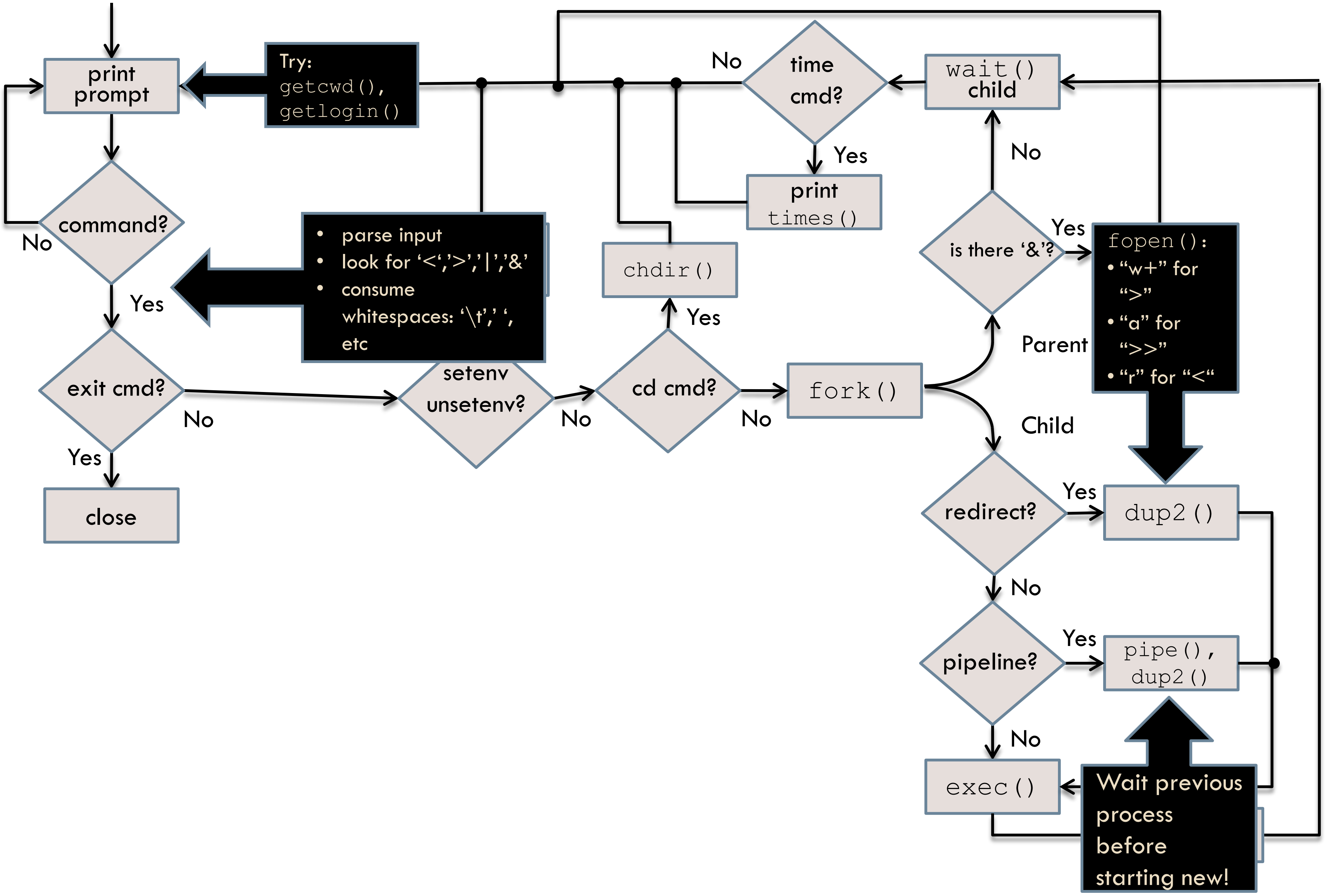
  (user@csd_sh /dir/# ls -l my_files/ > output_file) (see *dup2()*).
- Η έξοδος μιας εντολής θα μπορεί να δίνεται σαν είσοδος σε μια άλλη εντολή που υπάρχει στην ίδια γραμμή εντολών και διαχωρίζονται με το σύμβολο **"/"** . (user@csd_sh /dir/# ps axl | grep zombie) (see *pipe()*).
- **cd** (see chdir())
- **setenv/unsetenv** (see setenv() and unsetenv())
- **csdTime** (see *gettimeofday()* and *times()*)
- **exit**

Tips

1. First experiment with `fork()` and `getpid()`, `getppid()`
2. Use simple `printf` statements to distinguish parent from child (through pid)
3. Create logic for alternating execution
4. 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)`, `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/man1/time.1.html
- http://unixhelp.ed.ac.uk/CGI/man-cgi?times+2


print prompt
Try:
getcwd(),
getlogin()
command?
No
Yes
parse input
look for '<','>','|','&'
consume whitespaces: '\t',' ', etc
exit cmd?
Yes
close
No
setenv unsetenv?
No
cd cmd?
Yes
chdir()
No
fork()
Parent
is there '&'?
Yes
fopen():
• "w+" for ">"
• "a" for ">>"
• "r" for "<"
No
wait() child
time cmd?
No
Yes
print times()
Child
redirect?
Yes
dup2()
No
pipeline?
Yes
pipe(), dup2()
No
exec()
Wait previous process before starting new!