1. Very primitive shell

Look at this very primitive shell that executes one command a time without arguments and flags with and with no support of pipelines.

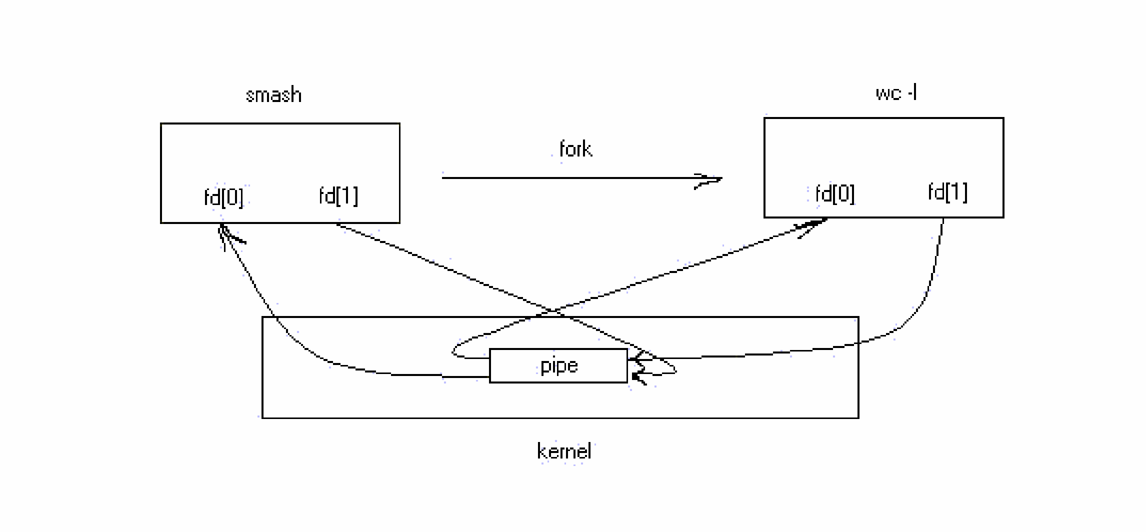
```
#include    <sys/types.h>
#include    <sys/wait.h>
#include        <unistd.h>
#include        <stdio.h>
#include        <string.h>
#include        <stdlib.h>



#define MAXLINE 2048

int
main(void)
{


      char  buf[MAXLINE];
      pid_t pid;
      int         status;


      printf("%% ");/* print prompt (printf requires %% to print %) */
      while (fgets(buf, MAXLINE, stdin) != NULL) {
            buf[strlen(buf) - 1] = 0;/* replace newline with null */

            if ( (pid = fork()) < 0)
                  fprintf(stderr,"fork error");

            else if (pid == 0) {          /* child */
                  execlp(buf, buf, (char *) 0);
                  fprintf(stderr,"couldn't execute: %s", buf);
                  exit(127);
            }

            /* parent */
            if ( (pid = waitpid(pid, &status, 0)) < 0)
                  fprintf(stderr,"waitpid error");
            printf("%% ");
      }
      exit(0);
}
```

2. Adding support of pipelines

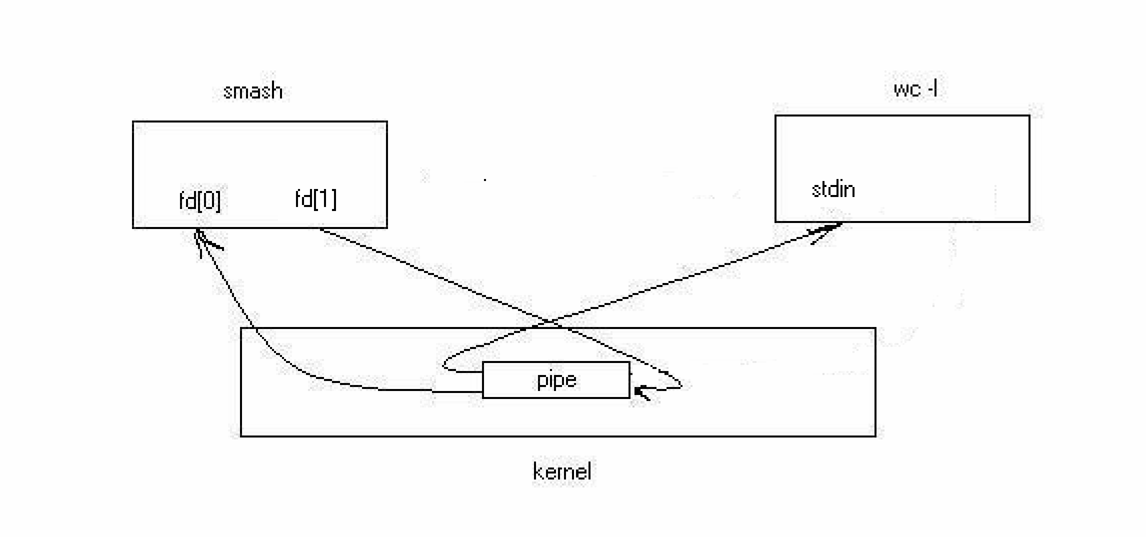
Let's see how pipelined commands are implemented on example of the following command:

ps | wc –l

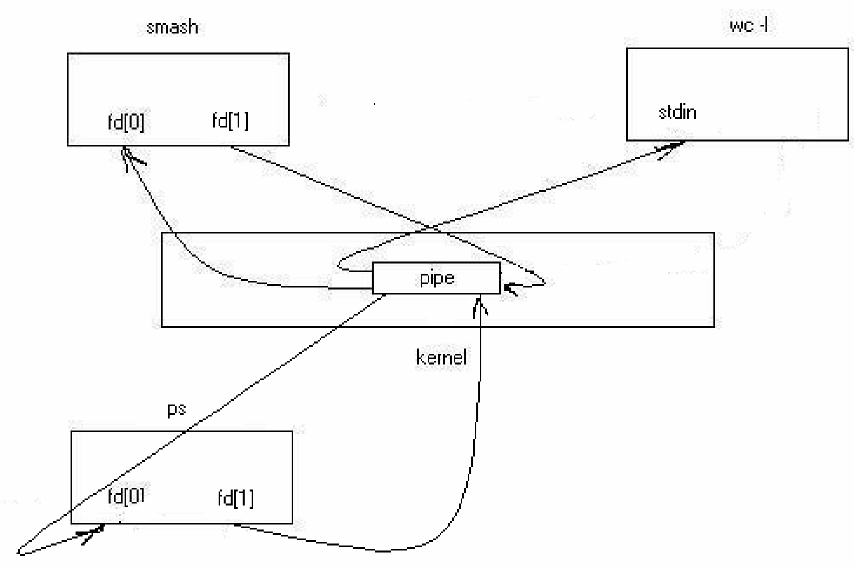
First of all shell process calls to the system call pipe and forks child process for wc:



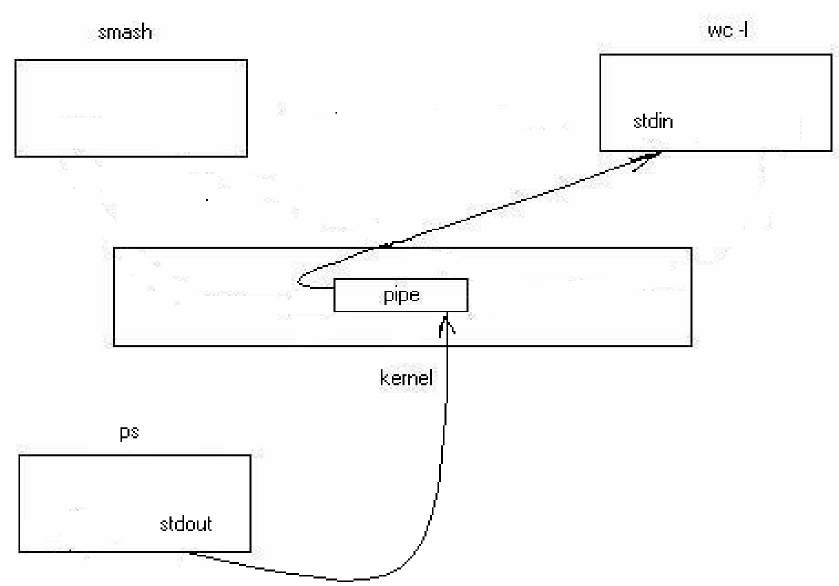
Before executing wc, shell closes non necessary file descriptors and "connects" stdin to the pipe using dup:



Now, shell forks another child for ps process:



Before executing ps, shell closes non necessary file descriptors and “connects” stout to the pipe using dup:



Following code (from dup.c) gives an example of pipe and dup system calls usage:

```
int
main(void)
{
      int n, fd[2];
      pid_t pid;
      char  line[2048];

      if (pipe(fd) < 0)
            fprintf(stderr,"pipe error");

      if ( (pid = fork()) < 0)
            fprintf(stderr,"fork error");

      else if (pid > 0) {                   /* parent */
            close(fd[0]);

            /* redirection of stdout so that stdout
                   of the process would be actualy
               written to the pipe
            */
            close(1); /* stdout */
            dup(fd[1]);
            close(fd[1]);
            write(1, "hello world\n", 12);

      } else {                      /* child */
            close(fd[1]);

            /* redirection of stdin so that stdin would
               be actualy read from pipe
            */
            close(0); /*stdin*/
            dup(fd[0]);
            close(fd[0]);
            n = read(0, line, 2048);
            write(1, line, n);
      }

      exit(0);

}
```

3 Data Structures for the Shell

Now lets complicate things and add support for job control. Our shell deals mainly with tree data structures. The jobSet type containes information about all jobs currently manages by the shell. The `job` type contains information about a job, which is a set of subprocesses linked together with pipes. The childProgram type holds information about a single subprocess. Here are the relevant data structure declarations:

```
struct jobSet {
    struct job * head;      /* head of list of running jobs */
    struct job * fg;        /* current foreground job */
};

struct job {
    int jobId;              /* job number */
    int numProgs;           /* total number of programs in job */
    int runningProgs;       /* number of programs running */
    char * text;            /* name of job */
    char * cmdBuf;          /* buffer various argv's point into */
    pid_t pgrp;             /* process group ID for the job */
    struct childProgram * progs; /* array of programs in job */
    struct job * next;      /* to track background commands */
    int stoppedProgs;       /* number of programs alive, but stopped */
};

struct childProgram {
    pid_t pid;              /* 0 if exited */
    char ** argv;           /* program name and arguments */
    int numRedirections;    /* elements in redirection array */
    glob_t globResult;      /* result of parameter globbing */
    int freeGlob;           /* should we globfree(&globResult)? */
    int isStopped;          /* is the program currently running? */
};
```

3 Initializing the Shell

When a shell enables job control, it should set itself to ignore all the job control stop signals so that it doesn't accidentally stop itself. You can do this by setting the action for all the stop signals to `SIG_IGN`.

```
    signal (SIGINT, SIG_IGN);
    signal (SIGQUIT, SIG_IGN);
    signal (SIGTSTP, SIG_IGN);
    signal (SIGTTIN, SIG_IGN);
    signal (SIGTTOU, SIG_IGN);
```

4 Launching Jobs

To create the processes in a process group, you use the same `fork` and `exec` functions described. Since there are multiple child processes involved, though, things are a little more complicated and you must be careful to do things in the right order.

As each process is forked, it should put itself in the new process group by calling `setpgid`; The first process in the new group becomes its *process group leader*, and its process ID becomes the *process group ID* for the group. .

If the job is being launched as a foreground job, the new process group also needs to be put into the foreground on the controlling terminal using `tcsetpgrp`.

The thing each child process should also do is to reset its signal actions:

```
            signal (SIGINT, SIG_DFL);
            signal (SIGQUIT, SIG_DFL);
            signal (SIGTSTP, SIG_DFL);
            signal (SIGTTIN, SIG_DFL);
            signal (SIGTTOU, SIG_DFL);
            signal (SIGCHLD, SIG_DFL);
```

Each child process should call `exec` in the normal way.

Here is the function from the shell program that is responsible for launching a command. The whole home work is about filling in 30 lines of code in this function:

```
int runCommand(struct job newJob, struct jobSet * jobList,
               int inBg) {
    struct job * job;
    char * newdir, * buf;
    int i, len;
    int nextin, nextout;
    int pipefds[2];             /* pipefd[0] is for reading */
    char * statusString;
```

```
int jobNum;

/* handle built-ins here -- we don't fork() so we can't background
   these very easily */
if (!strcmp(newJob.progs[0].argv[0], "exit")) {
    /* this should return a real exit code */
    exit(0);
} else if (!strcmp(newJob.progs[0].argv[0], "pwd")) {
    len = 50;
    buf = malloc(len);
    while (!getcwd(buf, len) && errno == ERANGE) {
        len += 50;
        buf = realloc(buf, len);
    }
    printf("%s\n", buf);
    free(buf);
    return 0;
} else if (!strcmp(newJob.progs[0].argv[0], "cd")) {
    if (!newJob.progs[0].argv[1] == 1)
        newdir = getenv("HOME");
    else
        newdir = newJob.progs[0].argv[1];
    if (chdir(newdir))
        printf("failed to change current directory: %s\n",
                strerror(errno));
    return 0;
} else if (!strcmp(newJob.progs[0].argv[0], "jobs")) {
    // FILL IN HERE
    // Scan the job list and print jobs’ status
    // using the following function
    //    printf(JOB_STATUS_FORMAT, job->jobId, statusString,
    //            job->text);
    // while statusString is one of the {Stopped, Running}


    return 0;
} else if (!strcmp(newJob.progs[0].argv[0], "fg") ||
           !strcmp(newJob.progs[0].argv[0], "bg")) {
    // FILL IN HERE
    // First of all do some syntax checking.
    // If the syntax check fails return 1
    // else find the job in the job list
    // If job not found return 1
    // If strcmp(newJob.progs[0].argv[0] == "f"
    // then put the job you found in the foreground (use tcsetpgrp)
    // Don't forget to update the fg field in jobList
    // In any case restart the processes in the job by calling
    // kill(-job->pgrp, SIGCONT). Don't forget to set isStopped = 0
    // in every proicess and stoppedProgs = 0 in the job
    return 0;
}

nextin = 0, nextout = 1;
for (i = 0; i < newJob.numProgs; i++) {
    if ((i + 1) < newJob.numProgs) {
        pipe(pipefds);
        nextout = pipefds[1];
```

```
        } else {
            nextout = 1;
        }

        if (!(newJob.progs[i].pid = fork())) {
            signal(SIGTTOU, SIG_DFL);

            if (nextin != 0) {
                dup2(nextin, 0);
                close(nextin);
            }

            if (nextout != 1) {
                dup2(nextout, 1);
                close(nextout);
            }


           signal (SIGINT, SIG_DFL);
           signal (SIGQUIT, SIG_DFL);
           signal (SIGTSTP, SIG_DFL);
           signal (SIGTTIN, SIG_DFL);
           signal (SIGTTOU, SIG_DFL);
           signal (SIGCHLD, SIG_DFL);

            execvp(newJob.progs[i].argv[0], newJob.progs[i].argv);
            fprintf(stderr, "exec() of %s failed: %s\n",
                    newJob.progs[i].argv[0],
                    strerror(errno));
            exit(1);
        }

        /* put our child in the process group whose leader is the
           first process in this pipe */
        setpgid(newJob.progs[i].pid, newJob.progs[0].pid);

        if (nextin != 0) close(nextin);
        if (nextout != 1) close(nextout);

        /* If there isn't another process, nextin is garbage
           but it doesn't matter */
        nextin = pipefds[0];
    }

    newJob.pgrp = newJob.progs[0].pid;

    /* find the ID for the job to use */
    newJob.jobId = 1;
    for (job = jobList->head; job; job = job->next)
        if (job->jobId >= newJob.jobId)
            newJob.jobId = job->jobId + 1;

    /* add the job to the list of running jobs */
    if (!jobList->head) {
        job = jobList->head = malloc(sizeof(*job));
    } else {
        for (job = jobList->head; job->next; job = job->next);
```

```
        job->next = malloc(sizeof(*job));
        job = job->next;
    }

    *job = newJob;
    job->next = NULL;
    job->runningProgs = job->numProgs;
    job->stoppedProgs = 0;

    if (inBg) {
        /* we don't wait for background jobs to return -- append it
           to the list of backgrounded jobs and leave it alone */

        printf("[%d] %d\n", job->jobId,
               newJob.progs[newJob.numProgs - 1].pid);
    } else {
        jobList->fg = job;

        /* move the new process group into the foreground */

        if (tcsetpgrp(0, newJob.pgrp))
            perror("tcsetpgrp");
    }

    return 0;
}
```

5 Stopped and Terminated Jobs

When a foreground process is launched, the shell must block until all of the processes in that job have either terminated or stopped. It can do this by calling the `waitpid` function; Using the `WUNTRACED` option so that status is reported for processes that stop as well as processes that terminate.

The shell must also check on the status of background jobs so that it can report terminated and stopped jobs to the user; this can be done by calling `waitpid` with the `WNOHANG` option. A good place to put a such a check for terminated and stopped jobs is just before prompting for a new command.

The shell can also receive asynchronous notification that there is status information available for a child process by establishing a handler for `SIGCHLD` signals. Implementing asynchronous notification requires a tackle with race conditions and can be implemented as a bonus.

Why asynchronous notification requires a dealing with a race conditions? In the shell program, the `SIGCHLD` signal is normally ignored. This is to avoid reentrancy problems involving the global data structures the shell manipulates. But at specific times when the shell is not using these data structures—such as when it is waiting for input on the terminal—it makes sense to enable a handler for `SIGCHLD`. The same function that is used to do the synchronous status checks (`checkJobs`, in our case) can also be called from within this handler.

Here are the parts of the shell program that deal with checking the status of jobs and reporting the information to the user:

```
/* Checks to see if any background processes have exited -- if they
   have, figure out why and see if a job has completed */
void checkJobs(struct jobSet * jobList) {
    struct job * job;
    pid_t childpid;
    int status;
    int progNum;

    while ((childpid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0)
{
        for (job = jobList->head; job; job = job->next) {
            progNum = 0;
            while (progNum < job->numProgs &&
                        job->progs[progNum].pid != childpid)
                progNum++;
            if (progNum < job->numProgs) break;
        }

        if (WIFEXITED(status) || WIFSIGNALED(status)) {
            /* child exited */
            job->runningProgs--;
```

```
            job->progs[progNum].pid = 0;

            if (!job->runningProgs) {
                printf(JOB_STATUS_FORMAT, job->jobId, "Done", job-
>text);
                removeJob(jobList, job);
            }
        } else {
            /* child stopped */
            job->stoppedProgs++;
            job->progs[progNum].isStopped = 1;

            if (job->stoppedProgs == job->numProgs) {
                printf(JOB_STATUS_FORMAT, job->jobId, "Stopped", job-
>text);
            }
        }
    }

    if (childpid == -1 && errno != ECHILD)
        perror("waitpid");
}
```

6. Summary

This assignment is not difficult. After all you have to do is to fill in 30 lines of the code in runCommand function according to the exact pseudo code.

To make the assignment difficult you can also implement asynchronous notification for 10 points bonus using system calls signal, sigemptyset, sigfillset, sigaddset, sigprocmask, sigsuspend. Hint: use sigprocmask to block/unblock signals in critical sections. Figure out which signals has to be blocked/unblocked.