Project 1 | CPE 453
This project must be done **individually**.

Scheduling

Scheduling is one of the principal components of operating systems. In this assignment, you will implement your own user-level scheduler to handle simple scheduling operations. You will be using process management system calls that you learned about in CPE 357.

Program: *schedule*

The name of the program is *schedule*. It launches a series of child processes and then allows each to run for a given period (the quantum) before stopping that process and allowing the next process to run. When a process concludes, it is taken out of circulation. This scheme, known as "round robin scheduling", repeats until there are no more processes running. Your program has to support many processes, each having 0 to 10 arguments. However, the number "10" should not be a magic number. For sanity's sake, you should #define in your header file: MAX_PROCESSES which should be over 100. MAX_ARGUMENTS can be 10.

The command line execution for this program looks like this:

```
schedule quantum [prog 1 [args] [: prog 2 [args] [: prog3 [args] [: … ]]]]
```

The first argument, quantum, is the number of **milliseconds** a process is allowed to run before its turn is over and it must be stopped. After that, come the command lines for each child, separated by colons. The colon delimiting the command lines will be surrounded by space, i.e. it will have one space before and one space after it. For example

```
# schedule 1000 foo a : bar b
```

should schedule two processes: *foo* and *bar*. Each process is also run with a command line argument: *a*, and *b* respectively. However, this syntax:

```
# schedule 1000 foo a: bar b
```

does not have a space before the colon, so your program should schedule only one process: foo, with command line arguments "a:", "bar", and "b".

Round Robin scheduling

The program will follow round robin scheduling. This means the original processes each take a turn in the original (user supplied) order. The sequence is never altered, although processes can drop out of it.

Detailed example

`Two` takes an integer argument, n, and prints n, n times, in a field $8n$ characters wide, sleeping for one second in-between. Two runs are shown in Figure 1. In the first one, the quantum is large enough for each program to finish, but in the second, it is not.

```
% schedule 10000 two 1 : two 2 : two 3
        1
                2
                2
                        3
                        3
                        3
% schedule 500 two 1 : two 2 : two 3
        1
                2
                        3
                2
                        3
                        3
%
```

Figure 1: Effect on the schedule of changing the quantum

Program Requirements

Your program must

- be named ***schedule*** (binary executable)
- use fork() to run all the processes in the order presented on the command line.
- allow each process to run until either the quantum expires or it terminates or it suspends itself.
- maintain the original round-robin scheduling order as processes terminate.
- only give quanta to live processes. If a process drops out, it should not be scheduled any more.
- respond immediately to process terminations or self-suspensions. A process should not be forced to take its full quantum, and the next process should not have to wait for the quantum to expire.
- give each process the opportunity to have a full quantum, regardless of how the previous process gave up its quantum.
- continue scheduling processes until there are no more processes left.
- dynamically allocate and deallocate resources for each process.
- use wait() or (waitpid()) for each child at an appropriate time; you don't want to leave the system swarming with zombies.
- not busy-wait. It can't just do an infinite poll like a "for(;;);" loop polling for data or signals. If it has nothing to do, it should not be executing anything. You may use:
    - interval timer (setitimer() with **ITIMER_REAL**) to keep track of when quanta have expired.
    - pause(2).

Helpful hints

The system calls in the following functions (see next page) could be useful for this assignment.

Hint on waitpid(): Usually wait() and waitpid() block and return only after a child process has been terminated. if you run waitpid() with the WUNTRACED flag, it will also return when a process has been stopped. You should also recall that whenever a process stops or terminates its parent receives a SIGCHLD signal.

More hints:

- A process can be stopped by sending it SIGSTOP signal. A stopped process can be restarted with a SIGCONT signal.
- Remember that catching a signal interrupts "long" system calls like wait() and pause() and causes them to return.
- Develop incrementally. Make sure you can launch and reap all of your children properly before attempting to schedule them.
- Think carefully before starting. This is not a large or complicated program, but there are some design subtleties that are best discovered before coding the majority of your

program.

- In particular, think about the problem of initially creating ***and stopping*** each child. Right after you run fork() a new process is created and starts executing immediately. But you must not allow it to run until it can be scheduled.
- Create test executables to test with the scheduler.

`pid_t fork(2)`	creates a new process that is an exact image of the current one

`int execl(3)`
`int execlp(3)`
`int execle(3)`
`int execv(3)`
`int execve(2)`
`int execvp(3)`
	known collectively as "the execs", this family of functions overwrites the current process with a new program. For this assignment, look closely at `execvp()`.

`int kill(2)`	sends a signal to a process
`void pause(2)`	suspends execution until a signal is received.
`int raise(3)`	sends a signal to the current process. It is equivalent to `kill(getpid(), sig);`
`getitimer(2)` `setitimer(2)`	for getting and setting the system interval timer
`sigaction(2)`	the POSIX reliable interface for signal handling
`sigprocmask(2)`	for manipulating the blocking or unblocking of particular signals

`sigemptyset(3)`
`sigfillset(3)`
`sigaddset(3)`
`sigdelset(3)`
`sigismember(3)`
	for manipulating signal sets used by `sigaction(2)` and `sigprocmask(2)`

`strtol(3)`	convert a string to an integer

`pid_t wait(2)`
`pid_t`
`waitpid(2)`
	waits for a child process to change status (terminate or stop)

Figure 2: Some potentially useful system calls and library functions

Assignment

Turn in this assignment on Canvas. Name of the submitted file must be **schedule.tar.gz**

What to turn in

1. Your source files (.c, .h, etc.)
2. A make file (called Makefile) that will build schedule on the department Unix machines.
3. A README.txt file with:
    a. Your names on the very first line
    b. Special instructions for the program
    c. If your program does not work properly, state why and what is missing
    d. Anything else you want me to consider while grading

Using tar/gzip

Your files must be tar-ed and gzipped. Only one file should be submitted, and that is your tar.gz file containing everything in one directory (do not tar the directory itself). **Do not include any binary files in your submission**. But do include your make file. Your tar.gz file must be named **schedule.tar.gz**. Also, **no external libraries** may be used. Any standard libraries that have been installed on Unix2.csc.calpoly.edu are OK.

Let's say you have four files you want to submit to this assignment: schedule.c, schedule.h, README and Makefile Then create schedule.tar.gz :

```
# tar -cf schedule.tar schedule.c schedule.h Makefile README.txt
# gzip schedule.tar
```