Shell command execution

- Linux have variety of shell programs e.g. bash, csh, zsh, etc.
- The shell inputs a command from end user and then execute it.
- Two types of shell commands
  - External commands: Separate executables are available e.g. ls, cp, rm, cal, gcc, ...
    - Executed by the shell using fork() and exec().
  - Internal commands: Command logic is built in shell itself e.g. cd, exit, alias, ...
    - Handled by shell using if-else.
- Two ways of executing commands
  - Synchronous execution: Shell waits for the command (program) to complete.
    - This default way of execution.
  - Asynchronous execution: Shell doesn't wait for the command (program) to complete.
    - & This is done by suffix "&" to the command.
    - terminal> firefox &

Inter-process communication

- When multiple processes are doing related tasks, there might be requirement of communication between the processes. It may need to transfer data of a process to another or a process might need to wait for another process.
- OS provides memory protection (using MMU hardware) due to which a process cannot directly access memory of another process.
- For such requirements, OS provides ways to communicate between processes. This is called as "IPC". There are two types of IPCs.
  - Shared memory model: OS allocates some memory which is accessible to both processes (willing to communicate with each other).
  - Message passing model: A process puts data into a message and send to OS. Then OS sends that data to another process.
- Linux IPC mechanisms
  - Shared memory
  - Signals
  - Message queue
  - Pipe
  - Sockets

Signals

- OS have a set of predefined signals, which can be displayed using command
  - terminal> kill -l
- A process can send signal to another process or OS can send signal to any process.
- Information about signals.
  - terminal> man 7 signal

Send signal

- kill command is used to send signal to another process, which internally use kill() syscall.

    - terminal> kill -SIG pid

```
ps -e

kill -9 5142
# 5142 is pid of the process to be killed

kill -TERM 7273
# 7273 is pid of the process to be terminated
```

- pkill command is used to send signal to multiple processes/instances of the same program.
    - terminal> pkill -SIG programname

```
ps -e

pkill -9 chrome
# kill all chrome tabs/windows

pkill -9 java
# kill all java based applications
```

Default action/disposition

- When any signal is sent to the process, one of the following action is done by default (if signal is not handled by the process).
- Term: Terminate the process.
    - e.g. SIGKILL, SIGTERM, SIGINT, SIGHUP, SIGALRM, etc.
- Core: Abort the process with Core Dump (execution context is stored in dump file).
    - e.g. SIGSEGV.
- Stop: Stop (suspend) the process.
    - e.g. SIGSTOP.
- Cont: Continue (resume) the process.
    - e.g. SIGCONT.
- Ign: Ignore the signal.
    - e.g. SIGCHLD.

Imporant Signals

1. SIGINT (2): When CTRL+C is pressed, INT signal is sent to the foreground process.
2. SIGTERM (15): During system shutdown, OS send this signal to all processes. Process can handle this signal to close resources and get terminated.
3. SIGKILL (9): During system shutdown, OS send this signal to all processes to forcefully kill them. Process cannot handle this signal.
4. SIGSTOP (19): Pressing CTRL+S, generate this signal which suspend the foreground process. Process cannot handle this signal.
5. SIGCONT (18): Pressing CTRL+Q, generate this signal which resume suspended the process.

6. SIGSEGV (11): If process access invalid memory address (dangling pointer), OS send this signal to process causing process to get terminated. It prints error message "Segmentation Fault".
7. SIGCHLD (17): When child process is terminated, this signal is sent to the parent process. The parent process may handle this to get the exit code of the child (wait() syscall).
8. SIGHUP (1): When a terminal is closed, all processes running in that terminal and terminated due to Hang up signal.

Signal handling

- Signals are software counter part of hardware interrupts.
  - Interrupt --> Processor --> Pause the task --> IVT --> ISR --> Resume the task.
  - Signal --> Process --> Pause the execution --> Signal handler table --> Signal handler --> Resume the execution.
- To handle the signal in a process.
  - step 1: Implement signal handler function in the process.

```
void my_signal_handler(int sig) {
    // signal handling logic
}
```

  - step 2: Register signal handler in the process's signal handler table. Typically done at the beginning of the program.

```
signal(signum, my_signal_handler);
```

Signals related syscalls

kill() syscall

- kill() send signal to another proces.
  - ret = kill(pid, signum);
    - arg1: pid of the process to whom signal is to be sent.
    - arg2: signal number -- defined in signal.h
    - returns: 0 on success and -1 on failure.
  - terminal> man 2 kill --> arg

signal() syscall

- signal() is used to install signal handler in current process (signal handler table).
- When signal is received, OS calls registered signal handler.
- old_handler = signal(signum, new_handler);
  - arg1 (int): signal number of signal to be handled (except SIGKILL and SIGSTOP).
  - arg2 (fn ptr): address of signal handler function.
    - typedef void (*sighandler_t)(int);
  - returns: address of old signal handler (in the table).
- To handle a signal

- step 1: implement a signal handler function

```
void sigint_handler(int sig) {
// ...
}
```

- step 2: register the signal handler function

```
signal(SIGINT, sigint_handler);
```

sigaction() syscall

- sigaction() is extended version of signal() in Linux.
- ret = sigaction(signum, &new_sigaction, &old_sigaction);
  - arg1: signal number of the signal to be handled.
  - arg2: (in param) address of sigaction struct that hold address of signal handler function.
  - arg3: (out param) address of sigaction struct to collect address of old signal handler. If NULL, then old sigaction is not returned.
  - returns: 0 on success.
- man sigaction
  - struct sigaction
    - sa_handler -- sig handler fn -- void handler(int);
    - sa_sigaction -- sig handler fn -- void handler(int,siginfo_t*,void*);
    - sa_flags -- SA_SIGINFO to indicate that sa_sigaction handler given.
    - sa_mask
    - sa_restorer (deprecated)
- To handle signal (method1)
  - step 1: implement signal handler function

```
void sigint_handler(int sig) {
// ...
}
```

  - step 2: register signal handler function

```
struct sigaction sa;
memset(&sa, 0 , sizeof(struct sigaction));
sa.sa_handler = sigint_handler;
ret = sigaction(SIGINT, &sa, NULL);
```

sigaction() syscall

- sigaction() is extended version of signal() in Linux.
- ret = sigaction(signum, &new_sigaction, &old_sigaction);
  - arg1: signal number of the signal to be handled.
  - arg2: (in param) address of sigaction struct that hold address of signal handler function.
  - arg3: (out param) address of sigaction struct to collect address of old signal handler. If NULL, then old sigaction is not returned.
  - returns: 0 on success.
- man sigaction
- struct sigaction
  - sa_handler -- sig handler fn -- void handler(int);
  - sa_sigaction -- sig handler fn -- void handler(int,siginfo_t*,void*);
  - sa_flags -- SA_SIGINFO to indicate that sa_sigaction handler given.
  - sa_mask
  - sa_restorer (deprecated)
- To handle signal (method2)
  - step 1: implement signal handler function

```
void sigint_handler(int sig, siginfo_t *si, void *param) {
// ... si contains info about the signal e.g. si_pid (signal sender
pid)
}
```

  - step 2: register signal handler function

```
struct sigaction sa;
memset(&sa, 0 , sizeof(struct sigaction));
sa.sa_flags = SA_SIGINFO;
sa.sa_sigaction = sigint_handler;
ret = sigaction(SIGINT, &sa, NULL);
```

Signal masking sigset_t

- sigset_t represent signal mask field.
- Logically it is 64-bit field -- 1 bit for each signal.
  - sigemptyset(&set); --> make all signal bits 0
  - sigfillset(&set); --> make all signal bits 1
  - sigaddset(&set, signum); --> make signal bit 1 corresponding to given signal number
  - sigdelset(&set, signum); --> make signal bit 0 corresponding to given signal number

sigprocmask() syscall

- Change the signal mask field to mask/unmask signals of the current process (till next call to the sigprocmask()).
- sigprocmask(mode, &new_sigset, &old_sigset)
    - arg1: mode=SIG_SETMASK to set new signal mask in the PCB of current process.
    - arg2: new signal mask. Signals to be masked should be 1 and other signals should be 0.
    - arg3: old signal mask (out param).
    - returns: 0 on success.

sigsuspend() syscall

- Change the signal mask field to mask given signals and pause the execution of the process until any of the unmasked signal is received.
- Once unmasked signal is received, its handler (user-defined or default) will be executed and old signal mask will be restored. The process will continue
- further (if not terminated due to signal handler).
- sigsuspend(&set);
    - arg1: temporary signal mask field for which process will be suspended.

pause() syscall

- Wait for any signal whose handler to be executed or process to be terminated.

Assignment

1. Improve your shell program so that it should not be terminated due to SIGINT (ctrl+C).
2. Implement asynchronous execution in your shell i.e. if command suffixed with &, then shell should not wait for the child process. Ensure that process isn't left zombie.