**Environment**

- setting environment for program execution

MY\_VAR=’myVar’

export MY\_VAR

  - retrieving environment from within program (e.g., from within C) : getenv()

File descriptors (per-process) / file descriptions (system-wide)

  - stdin, stdout, stderr

* File descriptor 0: sdin, 1:stout, 2:stderr

  - pipe(): **creates a pipe**, a unidirectional data channel that can be used for inter-process communication.

- dup2() : Overwrites the oldfd (file descriptor) with the newfd. Closes one file descriptor and points it at the same file description as another file description.

  - close(): closes a file descriptor, no longer can be used

  - fileno(): returns the **file descriptor** associated with *stream*. If return -1, set **errno** to indicate errors

- fopen() : file open with "w" as writing and "r" as reading

Shell usage

  - set environment from shell

* env VAR1="value" + set | grep TEST\_VAR

  - jobs / job control

|  |  |  |  |
| --- | --- | --- | --- |
|  | Background: there can only be at most 1 | Foreground: if no "&" operator is found, then the shell runs the command as a *foreground* job. | suspend |

control-z: reverse your last action

 - **echo: displaying lines of text or string which are passed as arguments on the command line**

* **Cat:** reads files sequentially, writing them to the standard output
* **Grep:** searches the named input FILEs
* **Errno**: **global integer variable**. A valid error numbers are all positive numbers. Errno = 0 at set up. And errno = -1 ( no errors)

Processes: A process is an instance of a running program.

  - fork():

* Make child the same as parents by create an identical copy of the open file descriptors associated with the parent process, except for PID.
* **fork() returns 0 to the child proces**s and **returns the process ID of the child process to the parent process.**
* Otherwise, -1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

    - concurrency, concurrent execution

* Two processes run concurrently (are concurrent) if their flows overlap in time

  - execve(): overwritten memory and program counter

+ but maintains the same PID for new program

  - **wait() / waitpid():** returns because the status of a child process is available

* If  pid = -1, **status is requested for any child process**.  
  If  pid > 0, **specifies the process ID of the child process** for which status is requested.
* If  pid = 0 status is requested for any child process whose process group ID is equal to that of the calling process.
* If pid < -1, status is requested for any child process whose process group ID is equal to the absolute value of pid.

  synchronization : At some point we need to put wait for the child to process

* When a child process terminates before the parent has called **wait**, the kernel retains some information about the process, such as its exit status, to enable its parent to call wait later.

orphaned processes ("daemons")

* getpid() -> because it will be passed the pid() of the parent
* A process of running code under new management
* In what case do we care about managing children?
  + When we have too many children and since this is a long live process and we do not clean up children

  calling with WNOHANG flag::

* return process id of the child
* Only read child processes that already done, not going to wait any longer
* **Return value** of -1 : no children at all, 0: no children to be reap at the time
* **If(waitpid(…Wnohang)) > 0)** -> there is some children ready to be reap, no waiting involve

    - zombies/reaping : half alive half deads

  - process groups, setpgid()/getpgid(), how a shell handles groups with signals, etc.

**setpgid(): Change process group of a process**

Stop and kill signals are sent to groups. The shell needs to **split off children into a different group** so it doesn’t get killed/stopped when it passes on the kill/stop signals

WIFEXITED(status). Returns true if the child terminated normally

WEXITSTATUS(status). Returns the exit status terminated child. Defined if WIFEXITED() returned true.

WIFSIGNALED(status). Returns true if the child process terminated be- cause of a signal that was not caught.

WTERMSIG(status). Returns the number of the signal that caused the child process to terminate. This status is only defined if WIFSIGNALED() returned true.

WIFSTOPPED(status). Returns true if the child that caused the return is currently stopped.

WSTOPSIG(status). Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED() returned true.

WIFCONTINUED(status). Returns true if the child process was restarted by receipt of a SIGCONT signal.

**Exceptions handler**: kernel mode. We run in **user mode** until hit kernal.

  - asynchronous vs. synchronous: happen immediately

* Calling open() on a file - sync
* Bad instruction - sync : cause immediately fail
* Segmentation fault - sync
* Ctrl-c at keyboard : async can happen anytime
* Page fault - async : cant know when it might happen - memory stuff
* Hardware error : async - because it something we code, not something expected to happen
* Network I/O: sync

  - traps, faults, aborts

* **Traps: intentional**

Examples: system calls, breakpoint traps, special instructions

Returns control to “next” instruction

* **Faults: Unintentional but possibly recoverable**

Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions

Either re-executes faulting (“current”) instruction or aborts

* **Aborts : unintentional and unrecoverable**

Examples: illegal instruction, parity error, machine check Aborts current program

  - system calls: Each x86-64 system call has a unique ID number

User calls: open(filename, options)

Calls \_\_open function, which invokes system call instruction

**Signals**

  - Signal blocking: There are 2 bit arrays (Received, and blocked) that handle signals

  - Sending vs. receiving a signal

* **Kernel *sends* (delivers) a signal to a *destination process***
* Reasons for kernal sending signals

Kernel detected **divide-by-zero (SIGFPE)** or the termination of a child process (SIGCHLD)

The **kill** system call to explicitly request the kernel to send a signal to the destination process

When a signal is received the appropriate bit is set to 1

  - kill() system call, use with negative vs. positive pid

*pid* > 0, then signal *sig* is sent to the process with the ID specified by *pid*.

*pid* = 0, then *sig* is sent to every process in the process group of the calling process.

*pid* equals -1, then *sig* is sent to every process for which the calling process has permission to send signals, except for process 1 (*init*).

  - Some familiarity with **SIGCHLD(to ignore), SIGINT(control c to terminate), SIGTSTP(control-z) , SIGCONT, SIGTERM(terminate), SIGKILL(terminate and cant be overwritten )**

  - Signals associated with ctrl-c and ctrl-z

SIGCHLD: A child has terminated

SIGINT: A ‘would you please terminate’ signal (ctrl-c keyboard interrupt, can be caught) SIGTSTP: Program is to stop running

SIGCONT: Program is set to running

SIGTERM: ‘I really really want you to terminate’ signal (can be caught/ignored) SIGKILL: You dead son (CANNOT BE CAUGHT/IGNORED)

  - Default actions, overriding with handlers or with SIG\_DFL (default) or SIG\_IGN (ignore).

* Sigaction() - override the stuff

  - Signal blocking with sigprocmask() - the act of setting the mask block vector for one or more signal, delay the receive of the signal while something might be send. It's a temporary thing

**Quiz note**

* When -1 is passed as the pid (first) argument to waitpid(): The wait set consists of all the parent's child processes.
* A signal can be sent to every process in a given process group.
* Each signal type has a **default action**, which include any of the following: the process terminates, the process ignores the signal , the process stops (suspends) until restarted by a SIGCONT signal
* If 0 is passed as the pid (first) argument to the function kill(), then the signal specified as the sig (second) argument is every process in the process group of the calling process, including the process itself.
* What is the default action for SIGINT (interrupt)? - control c is terminate
* Which signal cannot be overridden? -  **SIGKILL and SIGSTOP**