Summary Page: Signals & Processes in C
CISC 220, fall 2012

Types of Signals:

name	default action	notes
SIGALRM	terminates process	Used by Linux "alarm clock" timer
SIGCHLD	ignored	Sent by system when a child process terminates or stops
SIGCONT	re-starts stopped process	Sent by system when stopped process re-starts
SIGINT	terminates process	Sent by system when user hits control-C
SIGKILL	terminates process	Programs can't "catch" SIGKILL.
SIGSTOP	stops process	a stopped process can be re-started later Programs can't "catch" SIGSTOP.
SIGTERM	terminates process	
SIGTSTP	stops process	sent by system when user hits control-Z.
SIGUSR1	terminates process	Not used by system; user programs may use for any purpose
SIGUSR2	terminates process	Not used by system; user programs may use for any purpose

All signals except `SIGKILL` and `SIGSTOP` can be caught or ignored by user programs.

Sending Signals Using Bash:

`kill –signal pid`

`signal` should be a signal name or number. The initial "`SIG`" may be left off signal names.

`pid`: a process id or job id

Examples:

`kill –SIGSTOP 4432` sends a `SIGSTOP` signal to process 4432.

`kill –INT %2` sends a `SIGINT` signal to job number 2

If no signal is specified, sends a `SIGTERM`.

`kill –l` (lowercase L) prints a list of all the signal names and numbers, if you're interested.

Setting Up a Signal Catcher in a C Program:

`signal` function establishes a catcher for a particular signal type

```
void signal(int signum, void (*catcher) (int));
```

Predefined catchers:

`SIG_IGN`: ignore the signal

`SIG_DFL`: use the default action for the signal

Waiting For Signal:

```
pause(); /* suspends until you receive a signal */
```

Sending a Signal To Yourself:

```
int raise(int signal);
```

Using the Alarm Clock:

```
int alarm(int secs);
/* generates a SIGALRM in that many seconds. */
/* alarm(0) turns off the alarm clock */
```

Sending a Signal To Another Process:

```
kill(int pid, int signal);
```

Observing Processes From bash:

`jobs -l`: show your jobs with pid numbers
`ps`: info about your own processes
`ps a`: info about every process on computer coming from a terminal
`ps -e`: info about **every** process on computer (terminal or not)
add "-f" to any "ps" command: full listing format
`top`: interactive display of processes on computer

fork: Splits the current process into two concurrent processes.

```
pid_t pid = fork(); /* pid_t is an integer type */
```

For child process, result of fork is zero. For parent process, result of fork is the process id of the child. A negative result means error – couldn't create a new process.

"exec" functions. general: Each of these functions runs another program or script. This program or script takes over the current process and doesn't return. If an exec function returns, it means there was an error in calling the other program. You can't use wildcards, I/O redirection, or other shell features; the arguments are passed directly to the program.

execl: Takes a variable number of parameters:

- path name of the program to run
- "base name" of the program (without the directory name)
- first argument to the program
- second argument to the program
- ....
- last argument to the program
- NULL (to mark the end of the list of parameters)

Example (runs "`ls –l –F`")

```
execl("/bin/ls", "ls", "-l", "-F", NULL);
fprintf(stderr, "error: execl returned\n");
exit(1);
```

execv: Takes two parameters:

- path name of the program to run
- an array of strings, consisting of:
  - the "base name" of the program
  - arguments for the program
  - a NULL at the end

The second parameter will be passed to the program as the `argv` array.

Example (runs "`ls –l –F`")

```
char *args[4];
args[0] = "/bin/ls";
args[1] = "-l";
args[2] = "-F";
args[3] = NULL;
execv("/bin/ls", args);
fprintf(stderr, "error: execv returned\n");
exit(1);
```

execlp and execvp: Just like `execl` and `execv`, but the first parameter doesn't need to be a full name; the function will search your PATH.

Important note about using the "exec" functions: These functions cause the system to execute the named program with the arguments as given. Bash is not involved, so things like output redirection and references to shell variables will not be interpreted correctly.

wait: Waits until any child of the current process exits. Return value is the process id of the child that exited. Parameter is a pointer to an integer, which will get the exit status of the child.

Example:

```
int status;
pid_t child_pid = wait(&status);
if (status == 0)
   printf("child process %d was successful!\n", child_pid);
 else
   printf("child process %d exited with status = %d\n",
     child_pid, status);
```

If you don't care about the child's exit status, call `wait(NULL).`

waitpid: Waits for a child to exit, or checks on its status without waiting. Parameters are:

- the process id of the child to wait for (or -1, meaning any child)
- pointer to an integer, which will get the exit status of the child
- an int containing options. Useful values are:
  - 0: no special options; wait for child to exit
  - WNOHANG: Check without waiting. If the child process is still running, the function will return a value of zero immediately.
- return value is the process id of the child if it has exited, or zero if the child is still running and we're using the WNOHANG options.

`waitpid(-1,&status,0)` is equivalent to `wait(&status)`.

Libraries: This topic involves some new libraries:

`<sys/types.h>`:
    `pid_t` (the integer type for process ids)

`<unistd.h>`:
    `fork`
    the "exec" functions
    `sleep`
    `pause`
    `getpid`
    `getppid`

`<sys/wait.h>`:
    `wait`
    `waitpid`
    `WNOHANG`

`<signal.h>`:
    names for the signals
    `signal`
    `kill`
    `SIG_IGN`
    `SIG_DFL`