exec()

- runs a new program in the current process
  - loads the new program, overwriting the current code segment with the new program's code
  - re-initializes the heap and stack

exec()

```
int execl(const char *pathname, const char *arg, ...
                /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
                /* (char  *) NULL */);
int execle(const char *pathname, const char *arg, ...
                /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
                char *const envp[]);
```

- first argument (`pathname` or `file`) is the name of the executable file to run
- `execv()` vs `execl()` – `execv` takes an array of the program's arguments (`argv`), execl takes a list of separate parameters (`arg0, arg1, arg2, …`)
  - `argv[0]`, `arg0` are the filename of the executable
- `execvp()`, `execlp()`, `execvpe()` – *p* means they use the shell's PATH environment variable to locate the executable

exec()

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

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc"
    myargs[1] = strdup("p3.c"); // arg: input file
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {              // parent goes down this path
    int rc_wait = wait(NULL);
    printf("parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p3
hello (pid:29383)
child (pid:29384)
      29     107    1030 p3.c
parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```


Motivating the API

- the separation of `fork()` and `exec()` allows the environment to be set up in the new process before the desired program is run
- an application of this is supporting redirection and pipes in the shell

```
prompt> wc p3.c > newfile.txt
```

- > redirects `stdout` to a file
- < redirects a file to `stdin`
- | sends `stdout` from one command to `stdin` for the next
  - `wc p3.c | tee newfile.txt`
    - sends the output from `wc` to `tee`, which echoes it to the screen and to `newfile.txt`

Implementing Redirection

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

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC,
         S_IRWXU);
    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

```
prompt> ./p4
prompt> cat p4.output
      32     109     846 p4.c
prompt>
```

the first available file descriptor is assigned when a file is opened – by closing `stdout` and immediately opening a new file, that new file will get the same file descriptor that had been assigned to `stdout`...and so things that think they are writing to `stdout`'s are actually now writing to `./p4.output` instead

`kill()`, `signal()`

- `kill()` sends a signal to another process
  - `SIGINT` – generated by ctrl-C (term)
  - `SIGSTOP` – pause a process to be resumed later (stop)
  - `SIGCONT` – resume a paused process (cont)
  - `SIGKILL` – terminate a process immediately (term)
  - `man 7 signal` for a complete list
- `signal()` catches the signal and handles it
  - the *default disposition* defines what happens if the receiving process doesn't catch the signal

```
Term   Default action is to terminate the process.

Ign    Default action is to ignore the signal.

Core   Default action is to terminate the process and dump core (see core(5)).

Stop   Default action is to stop the process.

Cont   Default action is to continue the process if it is currently stopped.
```

  - `SIGKILL`, `SIGSTOP` cannot be caught or ignored

`kill()`, `signal()`

- users can generally only control their own processes
- the superuser (root) can control all processes