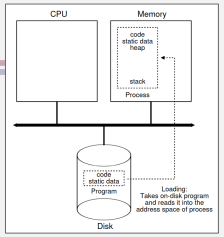
Process API

- **Create:** An operating system must include some method to create new processes. When you type a command into the shell, or double-click on an application icon, the OS is invoked to create a new process to run the program you have indicated. — `fork` `exec`
- **Destroy:** As there is an interface for process creation, systems also provide an interface to destroy processes forcefully. Of course, many processes will run and just exit by themselves when complete; when they don't, however, the user may wish to kill them, and thus an interface to halt a runaway process is quite useful. — `kill`
- **Wait:** Sometimes it is useful to wait for a process to stop running; thus some kind of waiting interface is often provided. — `wait`
- **Miscellaneous Control:** Other than killing or waiting for a process, there are sometimes other controls that are possible. For example, most operating systems provide some kind of method to suspend a process (stop it from running for a while) and then resume it (continue it running). — `kill` `signal`
- **Status:** There are usually interfaces to get some status information about a process as well, such as how long it has run for, or what state it is in.

Process Creation



Steps –

- load the program code and any static data into memory (in the address space of the process)
  - reads from disk
- allocate memory for the stack and initialize it with the parameters for main
  - includes memory for parameters, declared variables (local variables), return value, return address
- allocate memory for the heap
  - heap is for dynamically allocated memory
- set up open file descriptors
  - `stdin`, `stdout`, `stderr`

man pages

- if there are pages for *thing* in multiple sections, `man` *`thing`* finds the page in the lowest-numbered section
  - be aware of what section what you are looking for should be in, and if that's not what you get, use `man` *`section thing`* instead

```
1   Executable programs or shell commands
2   System calls (functions provided by the kernel)
3   Library calls (functions within program libraries)
4   Special files (usually found in /dev)
5   File formats and conventions, e.g. /etc/passwd
6   Games
7   Miscellaneous  (including  macro  packages  and  conventions),  e.g.  man(7),  groff(7),
    man-pages(7)
8   System administration commands (usually only for root)
9   Kernel routines [Non standard]
```


`fork()`

- creates a new process (child) that is a near-identical clone of the current one (parent)
  - same: child process has its own address space, registers, PC, etc which are a copy of the parent's
    - same PC means the child's execution begins with the return from `fork()`, not at the start of `main()`
  - same: child inherits thread-related state, open file descriptors
  - different: pid
  - different: return value from `fork()` – parent gets pid of child, child gets 0
  - for other details, `man fork`
- not deterministic – either parent or child could run next after the `fork()`
  - CPU scheduler determines which ready process runs next

fork()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p1
hello (pid:29146)
parent of 29147 (pid:29146)
child (pid:29147)
prompt>
```

`ps --forest`
to see the process hierarchy

wait(), waitpid()

- blocks the caller's process until another process has finished

```
pid_t wait(int *wstatus);

pid_t waitpid(pid_t pid, int *wstatus, int options);
```

  - `wait(NULL)` waits for *any* child process to terminate
  - `waitpid(pid,&wstatus,options)` allows specification of the process(es) to wait for (`pid`), access to status info about the waited-for child (`wstatus`), and what status change to wait for (`options`)
  - waiting for a child to terminate allows the system to release resources associated with the child after it terminates – without `wait()`, the terminated child remains in a zombie state until the parent terminates

wait()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.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());
  } 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> ./p2
hello (pid:29266)
child (pid:29267)
parent of 29267 (rc_wait:29267) (pid:29266)
prompt>
```

- ensures child's output is printed before the parent's, even if the parent runs first

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>
```