PROCESS MANAGEMENT:

Every process in UNIX has
- some code
- some data
- a stack
- a unique process ID (PID)

When UNIX starts (boots), there is only one process, called init, with process ID = 1.

The only way to create a new process is to duplicate an existing process; So init is the ancestor of all subsequent processes.

Initially, init duplicates (forks) several times and each child process replaces its code (execs) with the code of the executable getty which is responsible for user logins.

It is very common for a parent process to suspend itself until one of its child processes terminates. For example:

Consider the way a shell process executes a utility in the foreground:

- The shell first duplicates itself using fork()
- The child shell process replaces its code with that of the utility using exec(). (Differentiates)
- The parent process waits for the child process to terminate using wait()
- When the child process terminates using exit(), the parent process is informed (using a signal) and the parent process presents the user with the shell prompt to accept the next command.

System calls:

fork(), getpid(), getppid(), exit(), wait(), exec()

pit_t fork(void)

causes a process to duplicate. The child process is almost exactly a duplicate of the parent process; it inherits a copy of its parent's code, data, stack, open file descriptors, and signal tables. The only difference is in the process IDs (pid) and parent process IDs (ppid)

If fork() succeeds, it returns a value of 0 to the child process and the PID of the child process to the parent.

If fork() fails, it returns a -1 to the parent process and the child process is not created.

```
pid_t getpid(void)
pid_t getppid(void)
```

return the process' ID and parent process ID respectively.
They always succeed. (The PPID value for process with ID=1 is 1)

```
#include <stdio.h>
main () {
 int pid;
 printf ("I'm the original process with PID %d and PPID %d.\n",
         getpid (), getppid ());
 pid = fork (); /* Duplicate. Child and parent continue from here */
 if (pid != 0) /* pid is non-zero, so I must be the parent */ {
   printf ("I'm the parent process with PID %d and PPID %d.\n",
            getpid (), getppid ());
   printf ("My child's PID is %d\n", pid);
 }
 else /* pid is zero, so I must be the child */ {
   printf ("I'm the child process with PID %d and PPID %d.\n",
            getpid (), getppid ());
 }
 printf ("PID %d terminates.\n", getpid () ); /* Both processes execute this */
}
```

```
$ myfork
I'm the original process with PID 639 and PPID 416.
I'm the parent process with PID 639 and PPID 416.
My child's PID is 640
PID 639 terminates.
I'm the child process with PID 640 and PPID 1.
PID 640 terminates.
$
```

Orphan Processes: If a parent process terminates before its child terminates,
the child process is automatically adopted by the init process;

```
#include <stdio.h>
main () {
 int pid;
 printf ("I'm the original process with PID %d and PPID %d.\n",
         getpid (), getppid ());
 pid = fork (); /* Duplicate. Child and parent continue from here */
 if (pid != 0) /* Branch based on return value from fork () */ {
   /* pid is non-zero, so I must be the parent */
   printf ("I'm the parent process with PID %d and PPID %d.\n", getpid (), getppid ());
   printf ("My child's PID is %d\n", pid);
 }
 else {
   /* pid is zero, so I must be the child */
   sleep (5); /* Make sure that the parent terminates first */
   printf ("I'm the child process with PID %d and PPID %d.\n", getpid (), getppid ());
 }
 printf ("PID %d terminates.\n", getpid () ); /* Both processes execute this */
}
```

```
$ orphan
I'm the original process with PID 680 and PPID 416.
I'm the parent process with PID 680 and PPID 416.
My child's PID is 681
PID 680 terminates.
$ I'm the child process with PID 681 and PPID 1.
PID 681 terminates.
```

Terminating a Process: exit()

```
void exit(int status)
```

closes all of a process' file descriptors, deallocates its code, data, and stack; then terminates the process.

When a child process terminates, it sends its parent a SIGCHLD signal and waits for its termination code (status) to be accepted. (only lower 8 bits of status are used; so value 0-255).

A process which is waiting for its parent process to accept its return code is called a zombie process.

A parent accepts a child's termination code by executing wait()

The kernel makes sure that all of a terminating process' children are adopted by init by setting their PPID's to 1;

init always accepts its children's termination code.

exit() never returns

```
% cat myexit.c
#include <stdio.h>
main() {
  printf("I am going to exit with status code 42\n");
  exit(42);
}

% myexit
I am going to exit with status code 42
% echo $status
42
```

Zombie Process:

A process that cannot leave the system (even if it has exit-ed) until its parent process accepts its termination code.
(If parent process is dead; init adopts process and accepts code).
If the parent process is alive but is unwilling to accept the child's termination code (because it never executes wait());
the child process will remain a zombie process.

Zombie processes do not take up system resources but do use up one entry in the system's fixed-size process table. Too many zombie processes is a problem.

```
#include <stdio.h>
main () {
 int pid;
  pid = fork (); /* Duplicate */
 if (pid != 0) /* Branch based on return value from fork () */ {
     while (1)  /* Never terminate, and never execute a wait () */
       sleep (1000);
 }
 else {
   exit (42); /* Exit with a silly number */
 }
}

$ zombie &
[1] 684
$ ps
  684  p4 S    0:00 zombie
  685  p4 Z    0:00 (zombie <zombie>)
  686  p4 R    0:00 ps
$ kill 684
[1]+  Terminated              zombie
$ ps
  688  p4 R    0:00 ps
$
```

Waiting for a Child: wait()

```
pid_t wait(int *status)
```

causes a process to suspend until one of its child processes terminates. A successful call to wait() returns the PID of the child process that terminated abd places a status code into status that is encoded as follows:

- if the rightmost byte of status is zero, the leftmost byte contains the low 8 bits of the value returned by the child's exit() or return() call

- if the rightmost byte of status is non-zero, the rightmost 7 bits are equal to the Signal Number that caused the child to terminate and the last bit is set to 1 if the child core dumped.

If a process executes a wait() and has no children, wait() returns immediately with a value of -1

If a process executes a wait() and one or more of its children are already zombies, wait() returns immediately with the status of one of the zombies.

```
#include <stdio.h>
main () {
 int pid, status, childPid;
 printf ("I'm the parent process and my PID is %d\n", getpid ());
 pid = fork (); /* Duplicate */
 if (pid != 0) /* Branch based on return value from fork () */ {
     printf ("I'm the parent process with PID %d and PPID %d\n",
              getpid (), getppid ());
     childPid = wait (&status); /* Wait for a child to terminate. */
     printf ("A child with PID %d terminated with exit code %d\n",
             childPid, status >> 8);
 }
 else {
     printf ("I'm the child process with PID %d and PPID %d\n",
              getpid (), getppid ());
     exit (42); /* Exit with a silly number */
 }
  printf ("PID %d terminates\n", getpid () );
}

$ mywait
I'm the parent process and my PID is 695
I'm the parent process with PID 695 and PPID 418
I'm the child process with PID 696 and PPID 695
A child with PID 696 terminated with exit code 42
PID 695 terminates
```

Differentiating a Process: exec()

A process may replace its current code, data, and stack with those of another executable by using one of the exec() family of system calls.

When a process executes exec() its PID and PPID stay the same; only the code that the process is executing changes.

```
int execl(const char* path, const char* arg0, ...,
                              const char* argn, NULL)
int execv(const char* path, const char* argv[])
```

The following two use $PATH variable to find the executable:

```
int execlp(const char* path, const char* arg0, ...,
                               const char* argn, NULL)
int execvp(const char* path, const char* argv[])
```

```
path: executable (relative or absolute path for execl/execv
                  name of executable for execlp/execvp)

argi or argv[i]:
  ith command line argument for executable (arg0: name of executable)

If executable is not found, -1 is returned otherwise the calling process
replaces its code, data, and stack with those of the executable and
starts executing the new code.

A successful exec() never returns.
```

```
#include <stdio.h>
main () {
 printf ("I'm process %d and I'm about to exec an ls -l\n", getpid ());
 execl ("/bin/ls", "ls", "-l", NULL); /* Execute ls */
 printf ("This line should never be executed\n");
}
```

```
$ myexec
I'm process 710 and I'm about to exec an ls -l
total 38
-rw-rw-r--   1 raj      raj            187 Jul 22 20:24 alarm.c
-rw-rw-r--   1 raj      raj            226 Jul 22 20:22 background.c
-rw-rw-r--   1 raj      raj            284 Jul 22 20:22 mychdir.c
-rw-rw-r--   1 raj      raj           2058 Jul 22 20:23 vount.c
-rwxrwxr-x   1 raj      raj           4174 Jul 24 12:08 zombie
-rw-rw-r--   1 raj      raj            298 Jul 22 20:20 zombie.c
```

Changing Directories: chdir()

Every process has a current working directry that is used when processing a relative path name.

A child process inherits the current working directory from its parent; e. when a utility is run from a shell, the utility process inherits the shell's current working directory.

```
int chdir(const char* pathname)
```

sets a process' current working directory to pathname. The process must have execute permission from the directory for chdir() to succeed.

chdir() returns -1 if it fails; 0 if it succeeds

```
#include <stdio.h>
main () {
 system ("pwd"); /* Display current working directory */
 chdir ("/"); /* Change working directory to root directory */
 system ("pwd"); /* Display new working directory */
 chdir ("/home/naveen"); /* Change again */
 system ("pwd"); /* Display again */
}

$ mychdir
/home/raj/3320/ch12
/
/home/naveen
$
```

Changing Priorities: nice()

Child process inherits the priority of the parent process and can change the priority using nice()

```
int nice(int delta)
```

adds delta to the priority (legal values of priority -20 to +19)
Only super user can change to negative priority.

Smaller the priority value, faster the process will run.

Getting and Setting User and Group Ids:

```
uid_t getuid()              uid_t setuid(uid_t id)
uid_t geteuid()             uid_t seteuid(uid_t id)
gid_t getgid()              uid_t setgid(gid_t id)
gid_t getegid()             uid_t setegid(gid_t id)
```

The get functions always succeed; The set methods will succeed only when executed by super user or if id equals real or effective id of the process.

Program which uses fork() and exec() to execute a program in the background.

The original process creates a child process to exec the specified executable and terminates.

The orphaned child process is adopted by init.

```
#include <stdio.h>
main (int argc, char* argv[]) {
 if (fork () == 0) /* Child */ {
     execvp (argv[1], &argv[1]); /* Execute other program */
     fprintf (stderr, "Could not execute %s\n", argv[1]);
 }
}
```

```
$ background gcc mywait.c
```