PROCESS IDENTIFIERS

- Every process has a unique process ID, a non negative integer
- Special processes : process ID 0 scheduler process also known as swapper process ID 1 init process init process never dies ,it's a normal user process run with super user privilege process ID 2 pagedaemon

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid (void);
pid_t getppid (void);
uid_t getuid (void);
uid_t geteuid (void);
gid_t getgid (void);
gid_t getegid (void);
```

Fork function

- The only way a new process is created by UNIX kernel is when an existing process calls the fork function

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void);
```

- The new process created by fork is called child process
- The function is called once but returns twice
- The return value in the child is 0
- The return value in parent is the process ID of the new child
- The child is a copy of parent

- Child gets a copy of parents text, data , heap and stack
- Instead of completely copying we can use COW copy on write technique

```
#include<sys/types.h>
#include "ourhdr.h"
int  glob = 6;
/* external variable in initialized data */
char          buf[ ] = "a write to stdout\n";
int main(void)
{
      int       var;
      /* automatic variable on the stack */
      pid_t    pid;
      var = 88;
      if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
            err_sys("write error");
          printf("before fork\n");
      if ( (pid = fork()) < 0)
            err_sys("fork error");
      else if (pid == 0)
      {                          /* child */
            glob++;              /* modify variables */
            var++;
      }
else
            sleep(2);
      /* parent */
      printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
      exit(0);
}
```

Output

```
$ ./a.out
a write to stdout
before fork
pid = 430, glob = 7, var = 89        child's variables were changed
pid = 429, glob = 6, var = 88        parent's copy was not changed
$ ./a.out > temp.out
$ cat temp.out
a write to stdout
before fork
pid = 432, glob = 7, var = 89
before fork
pid = 431, glob = 6, var = 88
```

file sharing

- Fork creates a duplicate copy of the file descriptors opened by parent
- There are two ways of handling descriptors after fork

1. The parent waits for the child to complete
2. After fork the parent closes all descriptors that it doesn't need and the does the same thing

Besides open files the other properties inherited by child are

- Real user ID, group ID, effective user ID, effective group ID
- Supplementary group ID
- Process group ID
- Session ID
- Controlling terminal
- set-user-ID and set-group-ID
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors

- Environment
- Attached shared memory segments
- Resource limits

The difference between the parent and child

- The return value of fork
- The process ID
- Parent process ID
- The values of tms_utime , tms_stime , tms_cutime , tms_ustime is 0 for child
- file locks set by parent are not inherited by child
- Pending alrams are cleared for the child
- The set of pending signals for the child is set to empty set

■ The functions of fork

1. A process can duplicate itself so that parent and child can each execute different sections of code
2. A process can execute a different program

vfork

- It is same as fork
- It is intended to create a new process when the purpose of new process is to exec a new program
- The child runs in the same address space as parent until it calls either exec or exit
- vfork guarantees that the child runs first , until the child calls exec or exit

```
int glob = 6;
    /* external variable in initialized data */
    int main(void)
    {
            int     var;
```

```
        /* automatic variable on the stack */
                pid_t   pid;
                var = 88;
                printf("before vfork\n");
         if ( (pid = vfork()) < 0)
                err_sys("vfork error");
        else if (pid == 0) {                /* child */
                glob++;
        /* modify parent's variables */
                var++;
                _exit(0);                   /* child terminates */
        }
        /* parent */
        printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
        exit(0);
}
```

Source : http://elearningatria.files.wordpress.com/2013/10/cse-iv-unix-and-shell-programming-10cs44-notes.pdf