Fork

Class Notes

Fork ()

- Fork is a part of a library called UNIX threads library.

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

- Fork creates a new process by duplicating the calling process

```
Usage: main ()
        {
                pid_t  pid;
                pid = fork();
        }
```

- Code instructions coming after fork call shall execute both in the contexts of parent and child.
- On success of the fork call, PID of the child process is returned to the parent and 0 is returned to the child. On failure -1 is returned to the parent and no child process gets created.
- It is a common practice to follow up a fork call with a conditional construct. It could check for the return value of fork and execute appropriate work in the child process.

```
Example:
        pid_t  pid;
        pid = fork();
        if (pid==0)
                printf("code running in child");
        else
                printf("code running in parent");
```

Fork implementation under Linux

- Fork API invokes system call **sys_fork (),** which invokes the kernel routine **do_fork ()** to start a new process. It does the following functions.
    1) Allocates new PCB (task_struct) instance.
    2) Map caller task_struct elements to new task_struct (except PID).
    3) Return 0.
- Child process acquires instruction of caller process and starts executing from return statement of do_fork().
- Under Linux, fork () is implemented using Copy-On –Write (COW). So the only penalty it acquires is the time and memory required to duplicate the parent's page table and to create a unique task structure for the child.
- Process created using fork can continue to run even if parent process created them is terminated.
- When a process terminates, kernel's process manager puts it in to exit state and notifies immediately to parent. When parent process signals kernel, child process in exit state shall be removed.

- **ps** tool lists out the processes in exit state with a flag <defunct>. Defunct process can be cleaned up by either of the following **methods**.
    1) Write parent code to wait for the child to terminate and read exit value on child's termination.
    2) Register a signal handler in the process context to respond to a signal SIGCHILD.

- The first method can be implemented as below

```
        pid_t pid;
        int childstatus;
        int k=10;
        pid = fork();
// parent
        wait(&childstatus);
```

The parent will wait until the child gets terminated and exit status is returned to it.

- The second method can be implemented in two ways. In one way, the signal handler waits for the child status as shown below.

Sample code:

```
        int childstatus;

        void sighand (int signum) {
                printf(" I am in sig handler \n");
                wait(&childstatus);
        }

main(){
        pid_t pid;

        struct sigaction act;
        pid = fork();
        if( pid == CHILD){
                printf(" I am in child task \n");
                sleep(10);
        }
        // parent
                else{
                act.sa_flags = SA_NOCLDSTOP;
                act.sa_handler = sighand;
                        if( sigaction(SIGCHLD,&act,NULL) == -1);
                                perror("sigaction: ");
```

The flag SA_NOCLDSTOP is used to wait for the child status and it runs the signal handler.

- The second way is not to wait for the child status and runs the default handler. This can be done using a flag SA_NOCLDWAIT.

  Sample code:

```
main(){
        pid_t pid;

        struct sigaction act;
        pid = fork();
        if( pid == CHILD){
                printf(" I am in child task \n");
                sleep(10);
        }
        // parent
                else{
                act.sa_flags = SA_NOCLDWAIT ;
                act.sa_handler = SIG_DFL;
                if( sigaction(SIGCHLD,&act,NULL) == -1);
                        perror("sigaction: ");
```

- File descriptors are shared between parent and child processes as per Copy On Write basis.
- Any attempt to close an open file descriptor either by parent or by child process will trigger a copy operation.

```
main(){
        int fd;
        char buf[2];
       pid_t pid;
        int childstatus;
        fd = open("./test",O_RDONLY);
        pid = fork();
        if( pid == CHILD){
                read(fd,buf,2);
                printf(" in child %c\n",buf[0]);
                printf(" in child %c\n",buf[1]);
                close(fd);   // COW

                }
```

```
        // parent
        else{
                wait(&childstatus);
                        read(fd,buf,2);
                printf(" in parent %c\n",buf[0]);
                printf(" in parent %c\n",buf[1]);
                close(fd);
        }
}
```

- In the above program, a test file is opened and reads both in parent and child process. In the child process after read operation, the opened file gets closed. The output of the both the process will be same as the close operation in the child creates a Copy on Write.
- Any resource acquired by the parent process after fork operation will not be available to the child process created earlier.