Process Management

- An executable program for a CPU architecture consists of a sequence of instructions acceptable to the CPU. Executable programs reside on a media.
- A Process is an executable program in execution.
  - Program loaded into memory area called process address space.
  - Address space -- Set of memory locations accessible to the loaded program. Typically, a process has its own address space.
  - CPU fetches instructions from the address space to execute them.
  - The data part of the process is also stored in the address space.
  - A pointer to the next instruction to be executed is maintained.
- OS terminology for a process - task.
- UNIX Operating Systems are multi-tasking:
  - Multiple programs are loaded into memory and CPU is switched among them.

Process Management

- UNIX Operating Systems are multi-user:
  - Programs of multiple users are loaded in memory.
- UNIX Operating Systems are time sharing:
  - CPU switching is done so that user interactive programs generate quick responses to the users.
- Multiple processes can be created simultaneously from a single program.
  - Multiple vi processes, each created from /usr/bin/vi, can be executing concurrently on behalf of more than one user. The file /usr/bin/vi it self will be owned by root, but has rest of the world execute permission.
- Each process is one executing instance of the program from which it is created.

Process Identifiers

Identifier	System call to fetch ID
Process ID	**pid_t getpid(void)**
Parent process ID	**pid_t getppid(void)**
Real user ID	**uid_t getuid(void);**
Effective user ID	**uid_t geteuid(void);**
Real group ID	**gid_t getgid(void);**
Effective group ID	**gid_t getegid(void);**
Process group ID	**pid_t getpgrp(void);**
Supplementary group IDs	**int getgroups()**

Process Creation

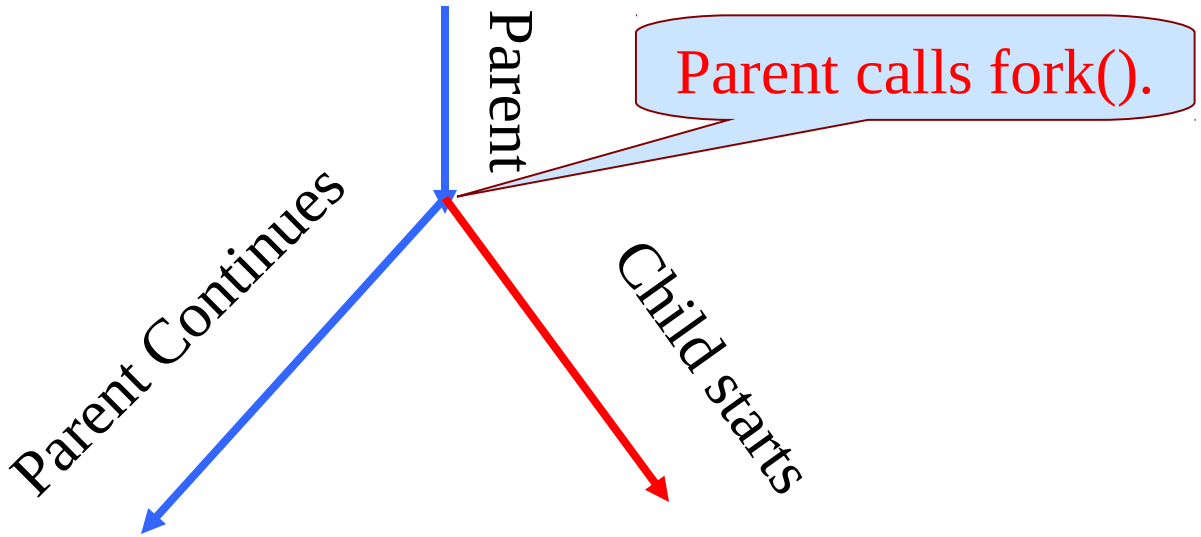
- A process can be created (only by) another process.
- First user process is hand crafted by kernel after it is loaded by boot strapping.
- Referred to as init process and has PID = 1.
- The init process is responsible for initializing (bringing up) the system.
- The lineage of every user process can be traced to init.

Process Creation

pid_t fork(void);

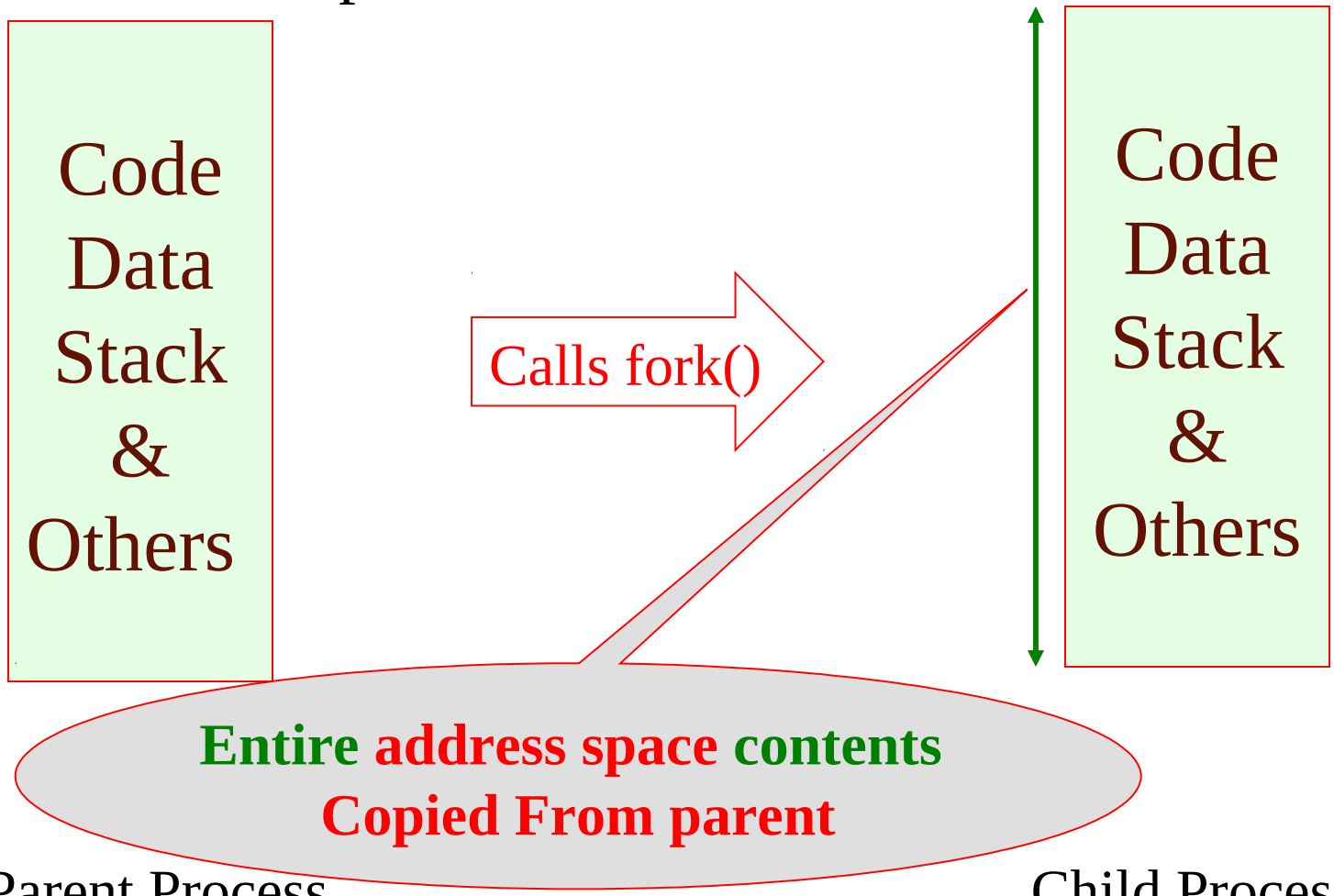
- Will create a new child process with the calling process being the parent.
- By the time fork() returns, both parent and child process will be eligible to run.

A CPU can run only one process at a time.



Process Creation

- Parent process's address space will be copied to the address space of the child.



Process Creation

- Child's address space is an exact replica of parent's. This includes code, data and stack.
- Child Starts execution as if fork() has just returned.
- Return value of fork():
  - In parent -- new Child's Process ID.
  - In Child -- Zero

```
short X= 100;
main(){
int Y=20;
X++, Y--;
pid_t fork_ret = fork();`
printf( "pid= %d\n", fork_ret );
printf("X=%d,Y=%d\n", X, Y);
}
```

Parent calls fork()

```
short X= 100;
main(){
int Y=20;
X++, Y--;
pid_t fork_ret = fork();
printf( "pid= %d\n", fork_ret );
printf("X=%d,Y=%d\n", X, Y);
}
```

Child starts at assignment.

Different Execution Paths

```
main(){
pid_t fork_ret = fork();

if (fork_ret < 0)
fprintf(stderr, “Error forking\n”);

if (fork_ret >  0) {
    printf( “Parent Process:\n”);
    printf( “PID=%d\n”, getpid());
    }

if (fork_ret ==  0) {
    printf( “Child Process:\n”);
    printf( “PID=%d\n”, getpid());
    }
}
```

```
main(){
pid_t fork_ret = fork();

if (fork_ret < 0)
fprintf(stderr, “Error forking\n”);

if (fork_ret >  0) {
    printf( “Parent Process:\n”);
    printf( “PID=%d\n”, getpid());
    }

if (fork_ret ==  0) {
    printf( “Child Process:\n”);
    printf( “PID=%d\n”, getpid());
    }
}
```


Logic Option

Different Execution Paths

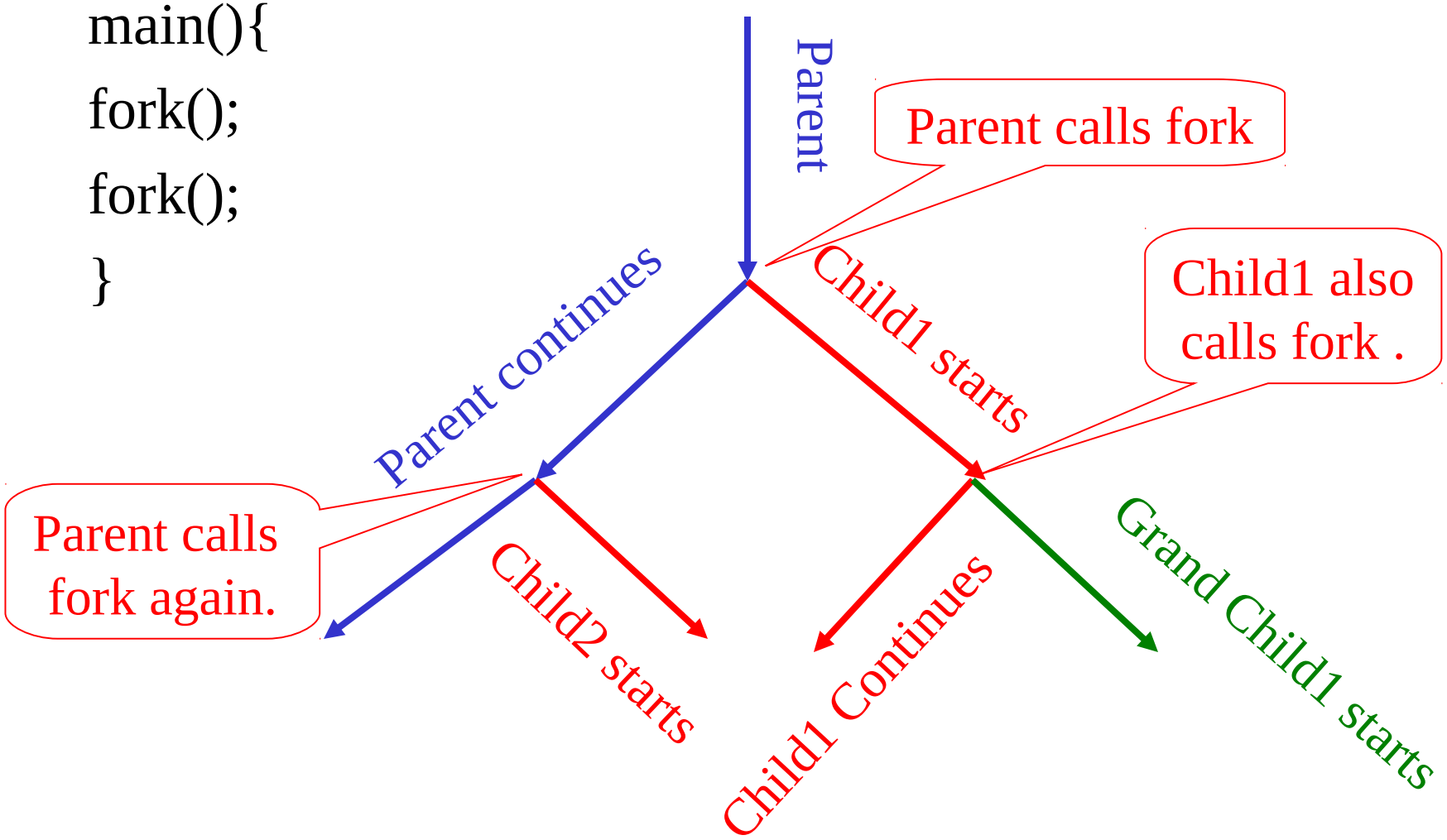
```
main(){
pid_t fork_ret = fork();

pid_t pid= getpid();
int Count = 40;

if (fork_ret >  0) {
    printf( “Parent pid=%d:\n”, pid);
    printf( “Count =%d\n”, ++Count);
    }

if (fork_ret ==  0) {
    printf( “Child pid:%d\n”, pid);
    printf( “Count=%d\n”,--Count);
    }
}
```

```
main(){
pid_t fork_ret = fork();

pid_t pid= getpid();
int Count = 40;

if (fork_ret >  0) {
    printf( “Parent pid=%d:\n”, pid);
    printf( “Count =%d\n”,++Count);
        }

if (fork_ret ==  0) {
    printf( “Child pid:%d\n”, pid);
    printf( “Count=%d\n”,--Count);
    }
}
```

Logic Option

How many Processes created?

```
main(){
fork();
fork();
}
```



FDT duplication on fork

```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

if (fork_ret >  0) { /* Parent */
read(fd1, buf, 80);
close(fd1);
int fd2=open("f2",O_RDONLY);
    }/* Only FDT Entry Freed.*/

if (fork_ret ==  0) { /* Child */
write(fd1, buf, 60);
int fd3=open("f3",O_WRONLY);
    }
}
```

FDT duplication on fork

FDT of Parent before fork

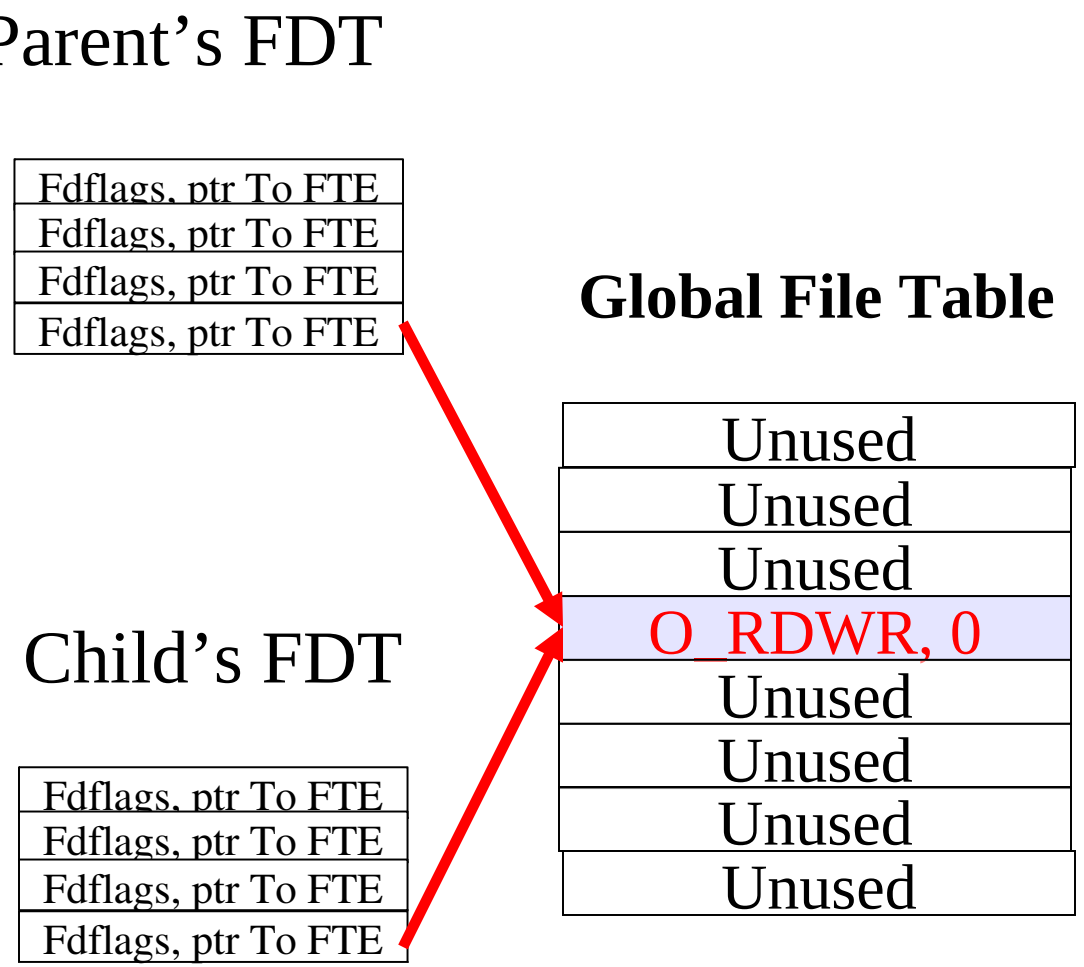
0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FD Flags, Pointer to FTE of file “f1”

FDT of child on creation - Duplicated from Parent

0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FD Flags, Pointer to FTE of file “f1”

FDT duplication on fork

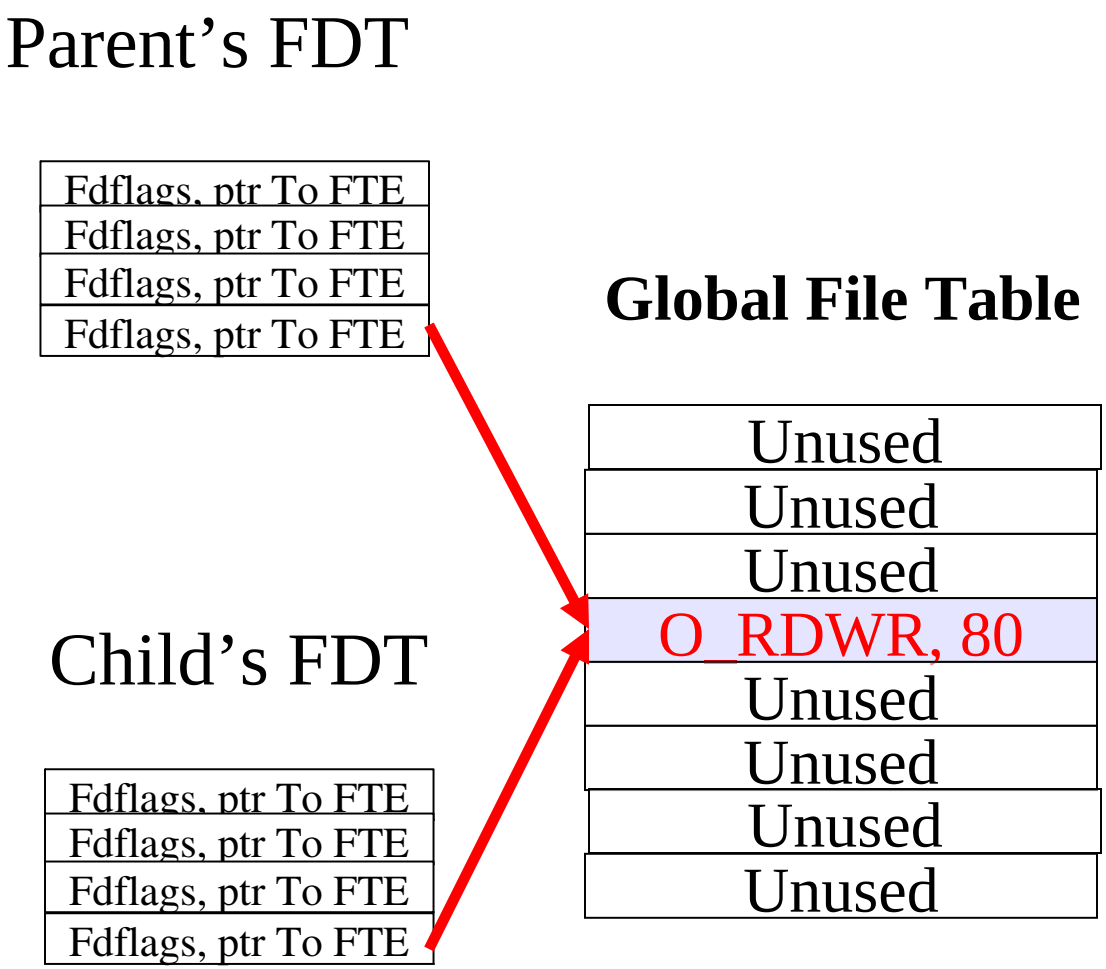
```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();
```




```
}
```

FDT duplication on fork

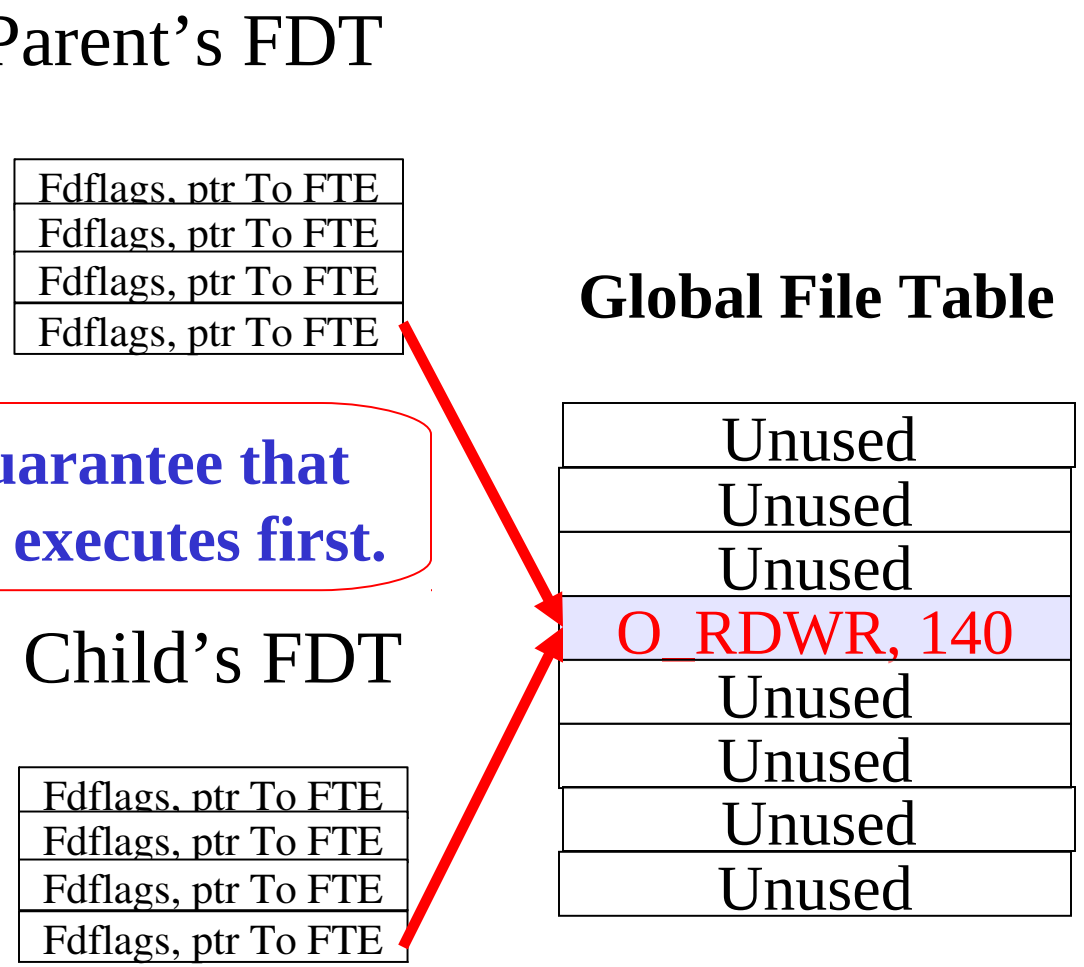
```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

if (fork_ret >  0) { /* Parent */
read(fd1, buf, 80);
    }




}
```



Logic Option

FDT duplication on fork

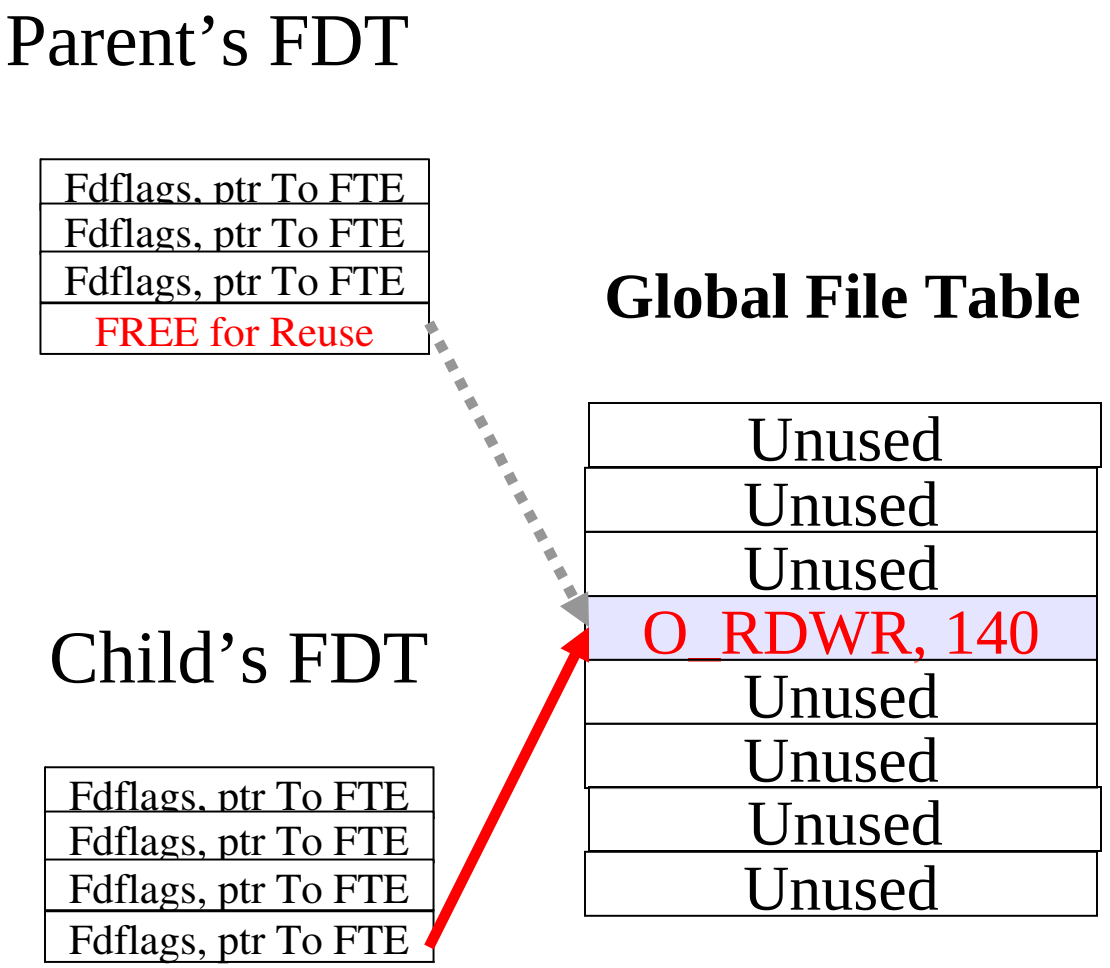
```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

if (fork_ret >  0) { /* Parent */
read(fd1, buf, 80);
    }

if (fork_ret ==  0) { /* Child */
write(fd1, buf, 60);
    }

}
```



Logic Option

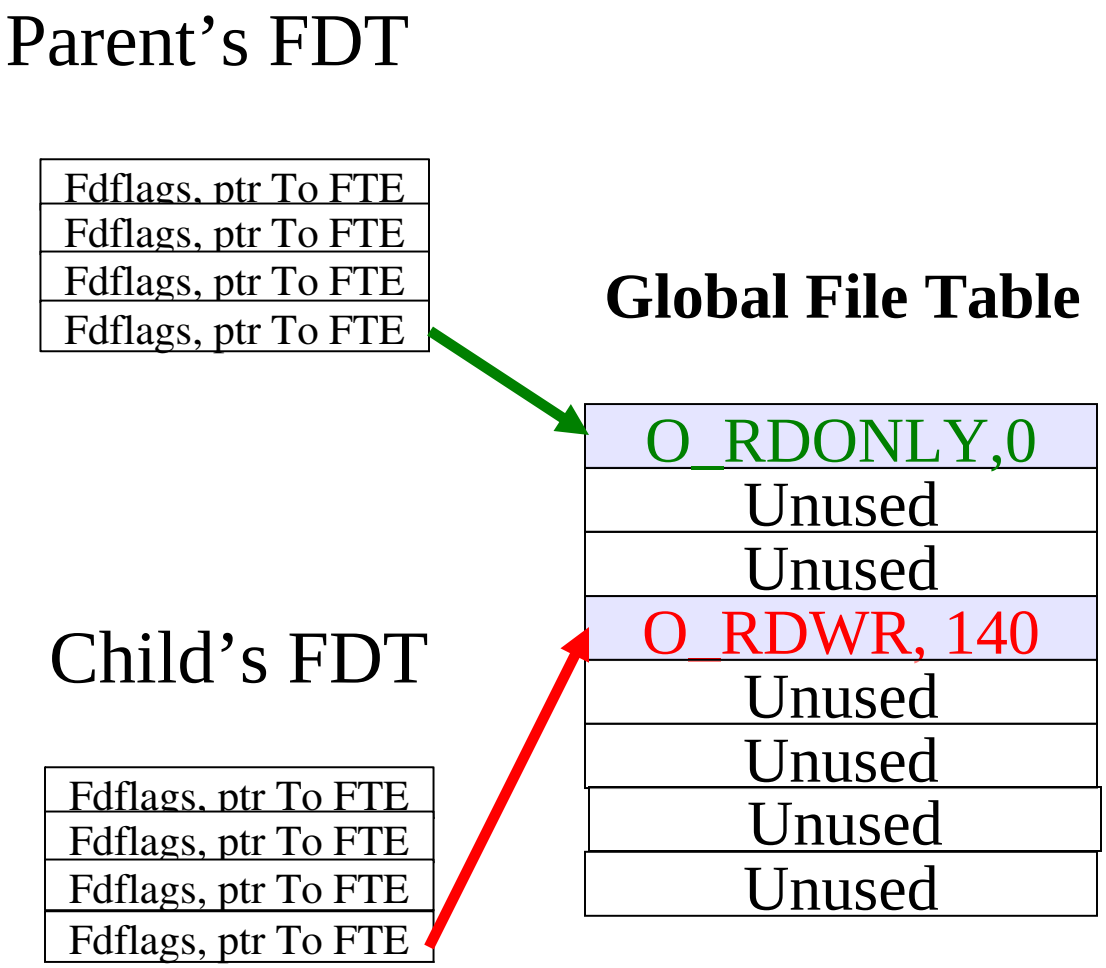
FDT duplication on fork

```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

if (fork_ret >  0) { /* Parent */
read(fd1, buf, 80);
close(fd1);
}/* Only FDT Entry Freed.*/

if (fork_ret ==  0) { /* Child */
write(fd1, buf, 60);
    }

}
```



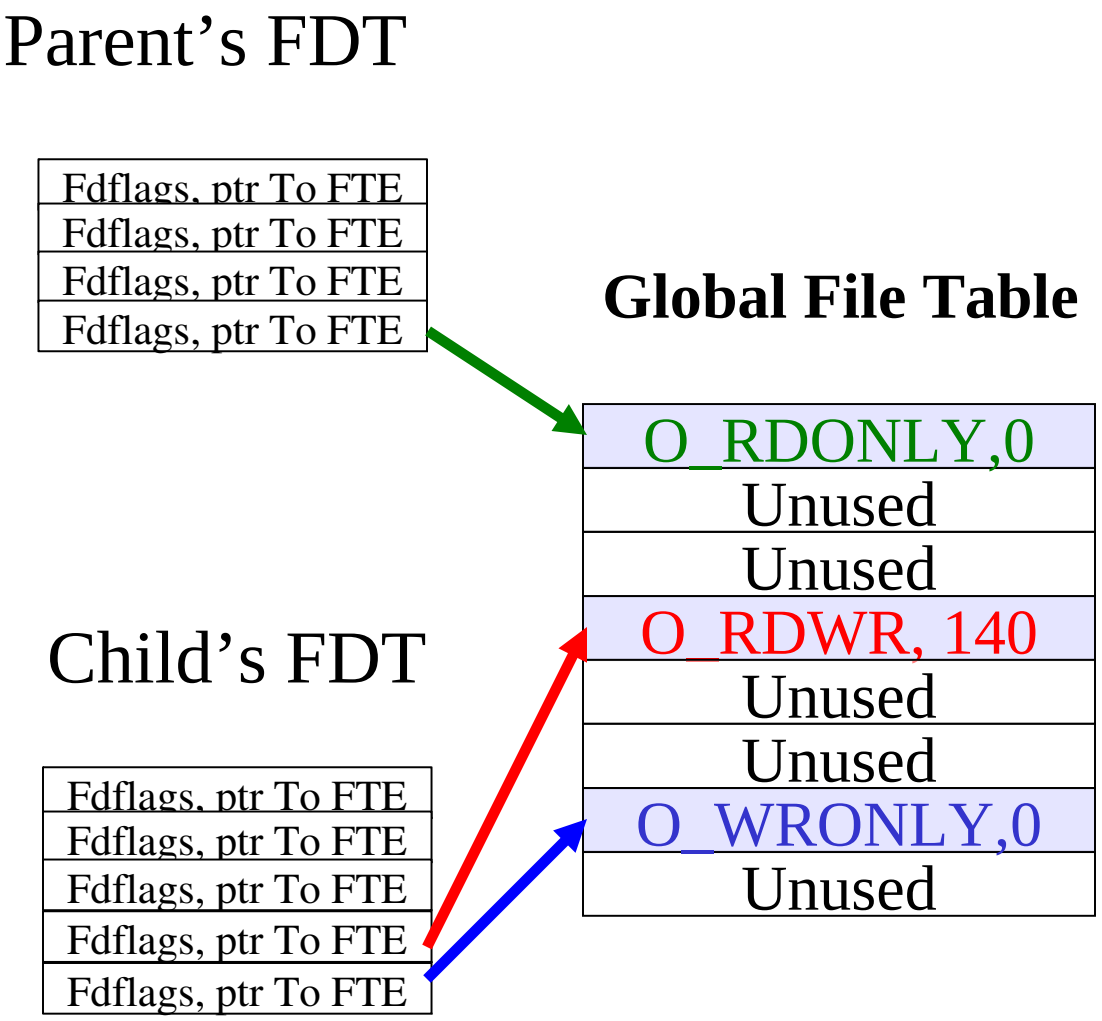
FDT duplication on fork

```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

if (fork_ret >  0) { /* Parent */
read(fd1, buf, 80);
close(fd1);
int fd2=open("f2",O_RDONLY);
}              /* fd2 only in parent */
if (fork_ret ==  0) { /* Child */
write(fd1, buf, 60);
    }

}
```



FDT duplication on fork

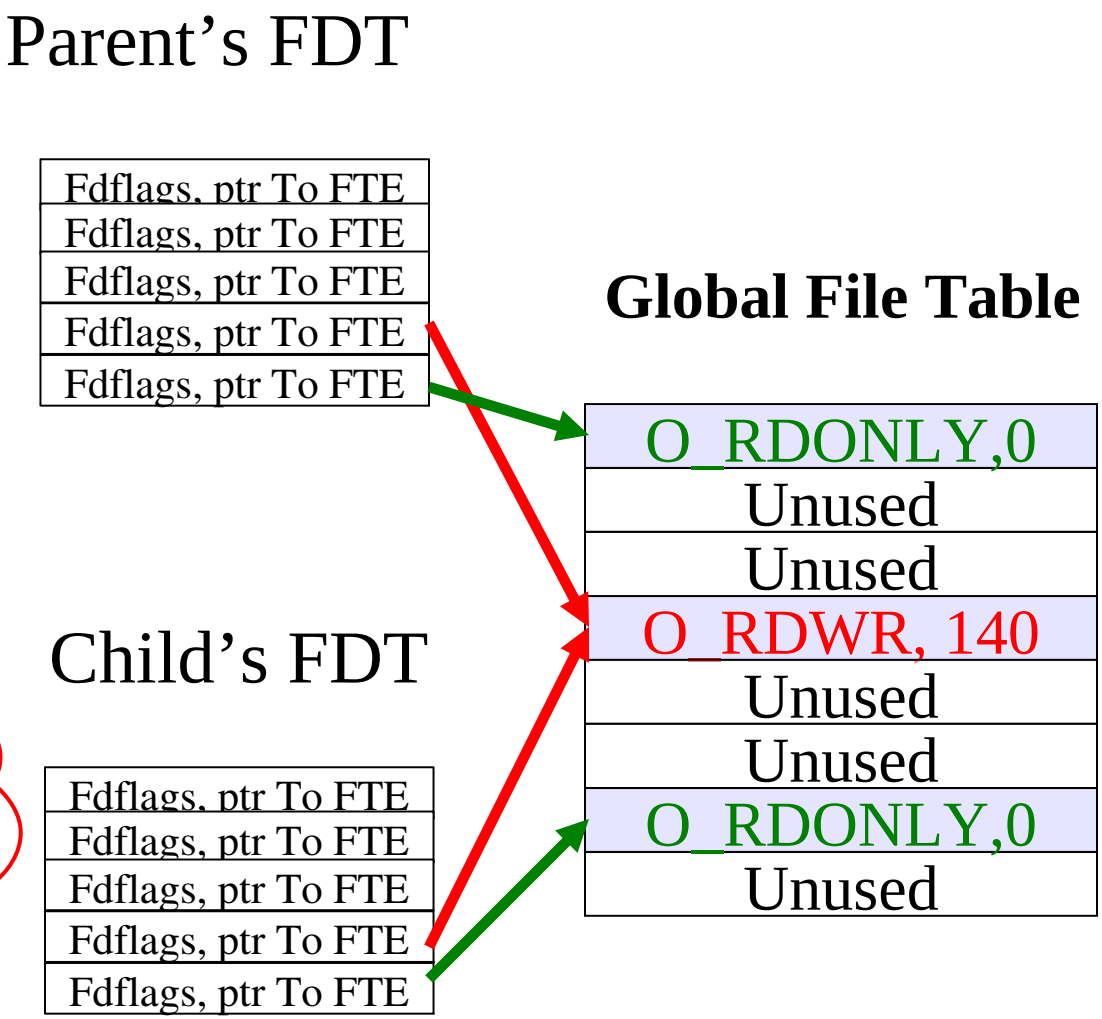
```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

if (fork_ret >  0) { /* Parent */
read(fd1, buf, 80);
close(fd1);
int fd2=open("f2",O_RDONLY);
}              /* fd2 only in parent */
if (fork_ret ==  0) { /* Child */
write(fd1, buf, 60);
int fd3=open("f3",O_WRONLY);
}              /* fd3 only in Child */
}
```



FDT duplication on fork

```
main(){
char buf[256];
int fd1=open("f1",O_RDWR);
pid_t fork_ret = fork();

int fd2=open("f2",O_RDONLY);

}
```

File descriptor(s) in which process?



FDT: Parent and child Open

FDT of Parent After Opening f2.

0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FD Flags, Pointer to FTE of file “f1”
4	FD Flags, Pointer to FTE of file “f2”

FDT of child After Opening f3.

0	FD Flags, Pointer to FTE of stdin device.
1	FD Flags, Pointer to FTE of stdout device.
2	FD Flags, Pointer to FTE of stderr device.
3	FD Flags, Pointer to FTE of file “f1”
4	FD Flags, Pointer to FTE of file “f3”

Process Termination

- Normal Termination
  - **Calls exit()**
  - **Calls _exit();**
  - **Returns from main()**
- Abnormal termination
  - **Process terminated due to a signal.**

    SIGTERM, SIGUSR1, SIGKILL
  - **The job of the process is partially completed.**
- Process Suspension
  - **Process suspended by a signal.**

    SIGSTOP, SIGTTIO
  - **Execution can be resumed by SIGCONT signal.**

Normal Termination

- void _exit(int status);
  - **System call that terminates the process normally.**
  - **Closes all open descriptors.**
  - **Status % 256 is the exit status of the process.**

- void exit(int status);
  - **Library call.**
  - **Flushes all open streams and closes them.**
  - **Status % 256 is the exit status.**
  - **Calls _exit() with LSB of status.**

- return (status); from main().
  - **Equivalent to calling exit(status);**

Normal Termination ?

```
main() {
int fd = open ( “f1”, O_RDWR):
if( fd < 0) {
    fprintf(stderr, “ Error opening f1\n”);
    _exit(10);
}
```

- Normal termination -- Process terminates itself.
- Abnormal termination -- If the process is terminated by an external signal before it completes it execution.

Process Termination status

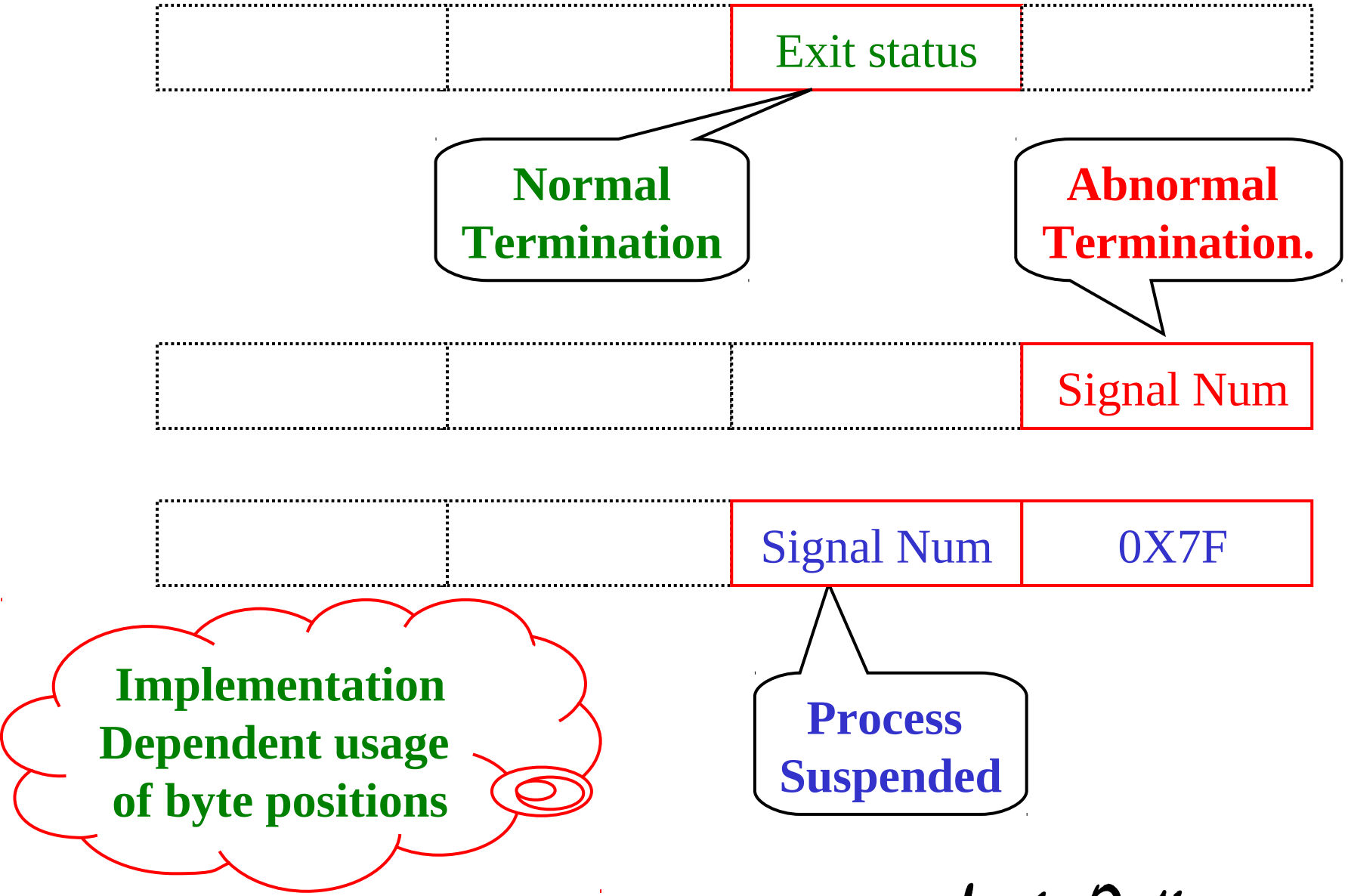
- When a process terminates, the kernel registers its termination status.
- Termination status is generally of integral type.
- Termination status embeds within it the process's termination type and the reason for termination.
  - Exits Status - Normal Termination
  - Signal Num - Abnormal Termination
  - Sigan Num - Process Suspension

Retrieving Termination Status

- pid_t wait(int *status_ptr);

- Parent process can retrieve the termination status of a child with wait() system call.
- The terminated child's PID is returned by wait.
- The terminated child's termination status is stored in the location pointed at by status_ptr.

Termination Status - Embeds….

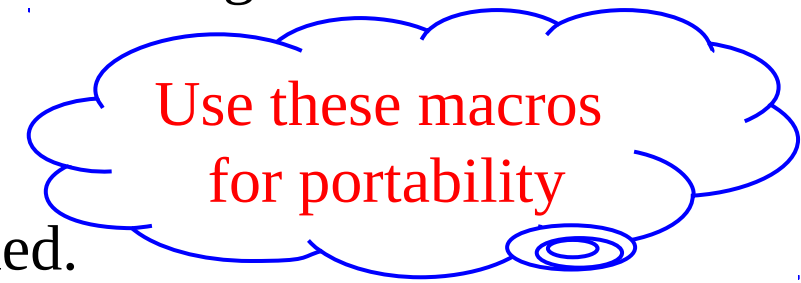


Termination Status -- Study

- WIFEXITED(*status_ptr)
  - True if child process had normal termination.
  - WEXITSTATUS(* status_ptr) - Gives exit status.

- WIFSIGNALED(* status_ptr)
  - True if child process had abnormal termination.
  - WTERMSIG(* status_ptr) - Gives signal number.

- WIFSTOPPED(* status_ptr)
  - True if child process was suspended.
  - WSTOPSIG(* status_ptr) - Gives signal number.
  - Only with WUNTRACED option of waitpid().



Retrieving Termination Status

```
main(){
if(fork()>){
    int s;
    pid_t pid=wait(&s);
    if(WIFEXITED(s))
        printf("Child Process ID=%d,\
                Exit status=%d\n",
            pid, WEXITSTATUS(s));
}
else{
    _exit(10);
}
```

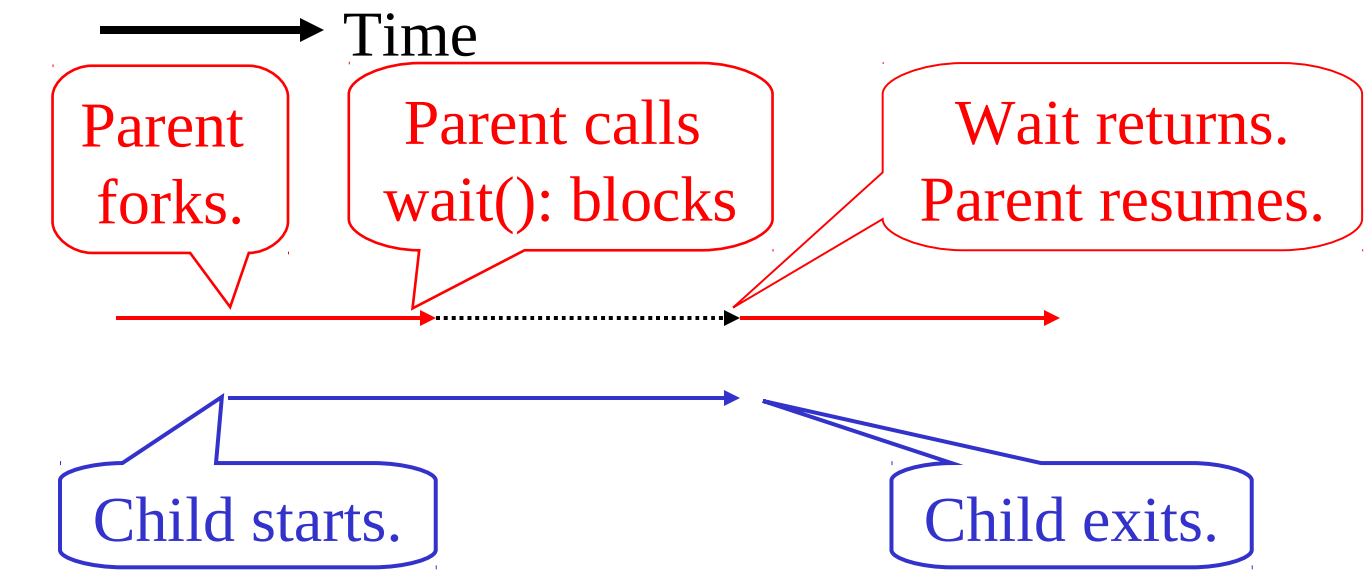
Termination Status of child with embedded Exit status

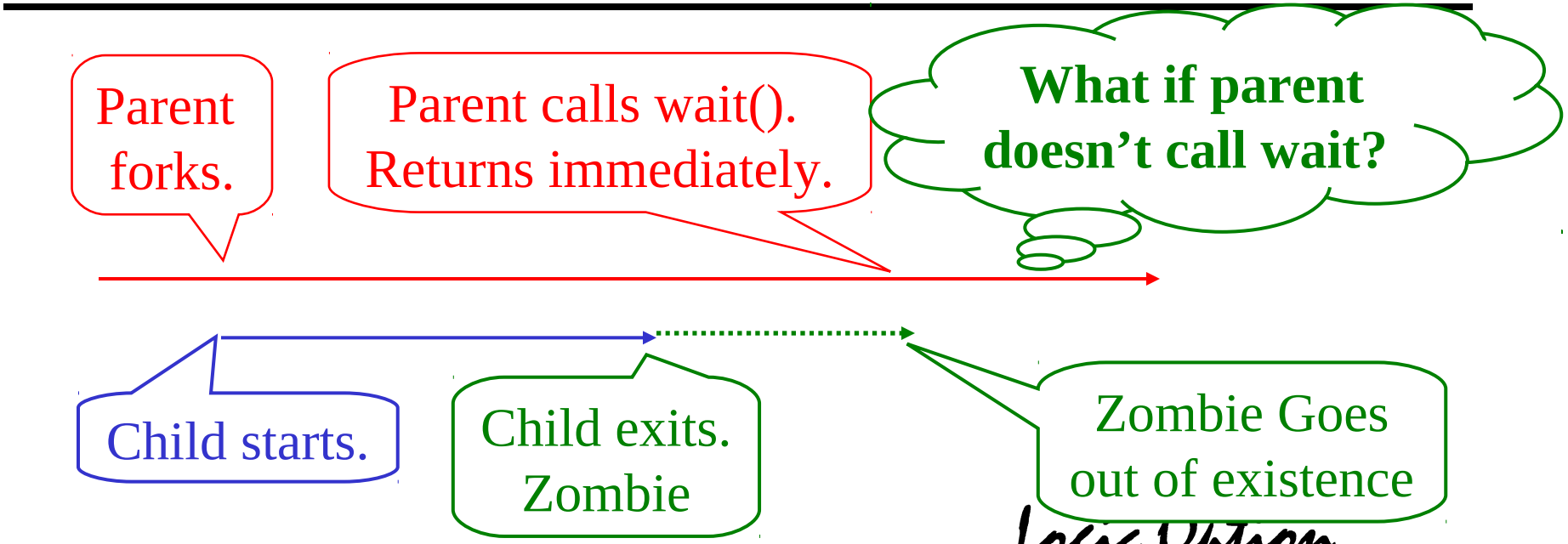
KERNEL

Child's Exit Status

Logic Option

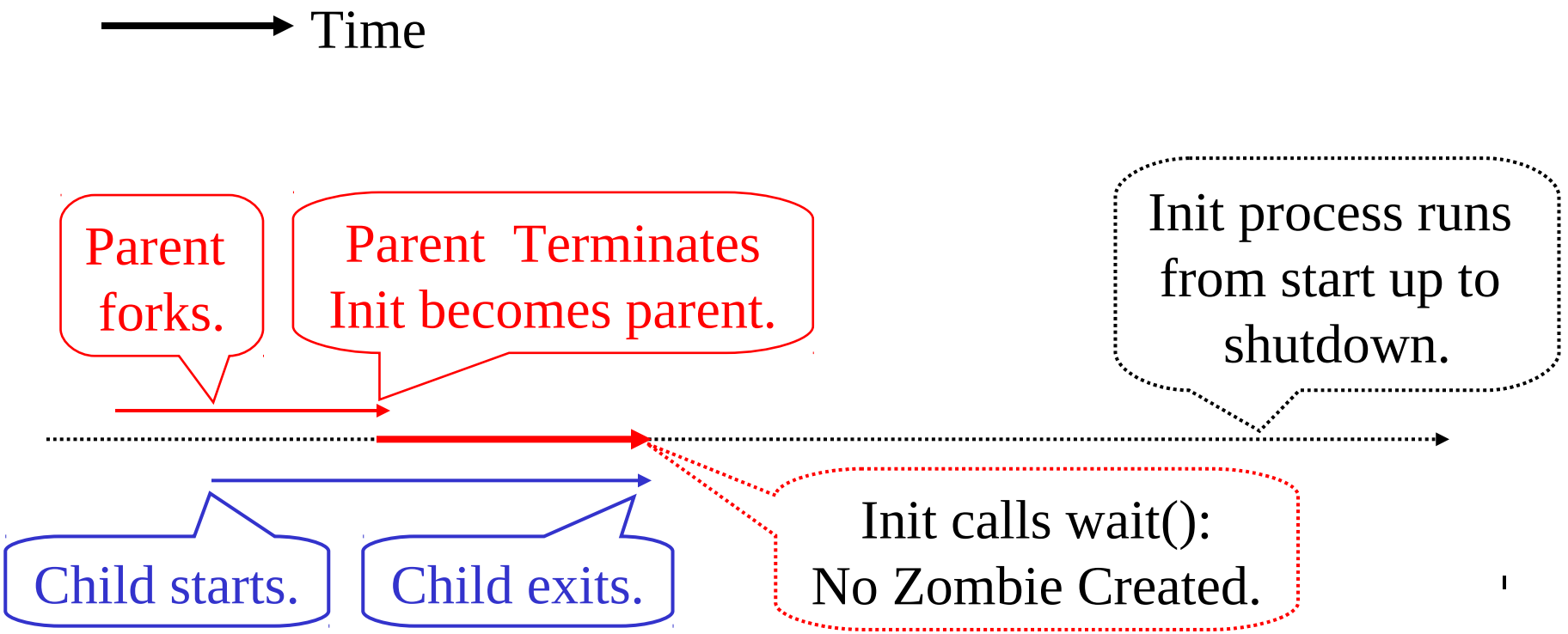
Child Terminates before Parent



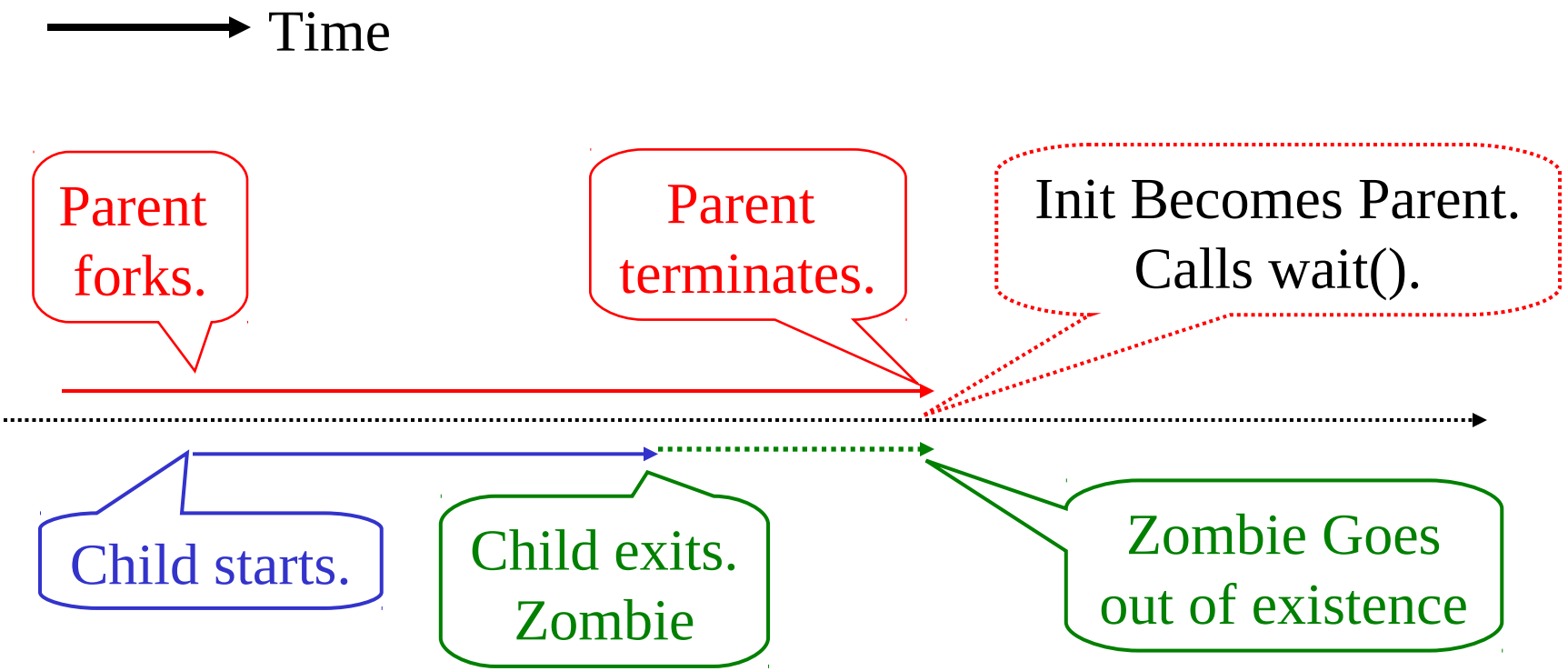


Logic Option

Parent Terminates before Child

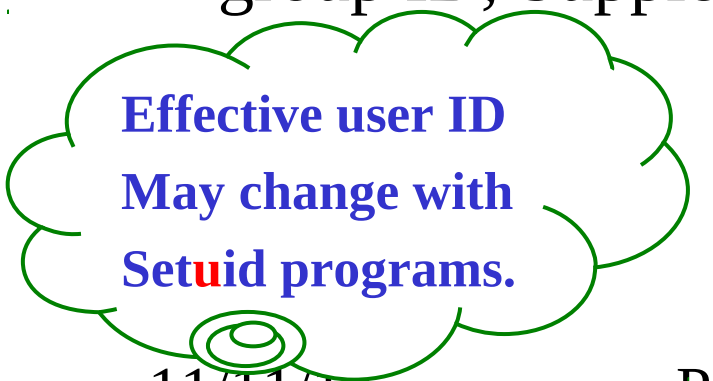


Zombies Revisited.



Running executables

- Typically an executable is loaded into the current process's address space.
- As a result the current process will start running the code from the executable.
- The old code of the program that loaded the executable will no longer execute.
- Loading executable will not result in any change to PID, PPID, Real user ID, Real group Id, Process group ID, Supplementary group Ids.

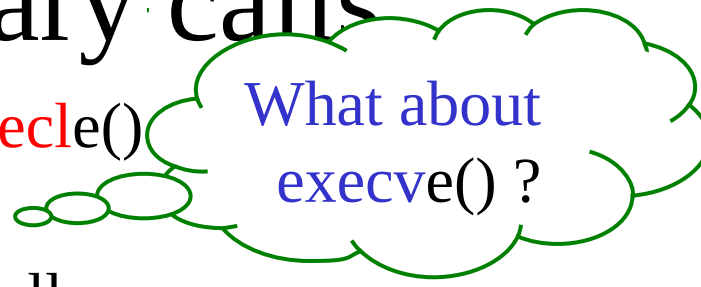


Logic Option

Exec family of library calls

- The library calls execl(), execlp(), execle()
  execv(), execvp()

  Constitute the exec family of library calls.
- The exec class of calls replace the calling process's address space with a new executable file.




```
main()
{
printf("Before execl()\n");
execl ("/home/kumar/myprog", "myprog", (char *) 0);
/* The above calls loads the executable /home/kumar/myprog
   into the current process and starts executing it's main().
   Exec functions never return on successful execution. */
printf("This printf() should not be executed.\n");
}
```

Execl

- int execl(const char *path, const char *arg, ...);

  execl ("/home/kumar/myprog",

  "myprog", "arg1", "arg2", "arg3", (char *) 0);

- path - pathname of the file to be executed in the current process.
- The program myprog will start executing in the current process. No change in PID or PPID.
- Assuming c Source for myprog , execution starts in main() of myprog . The command line arguments would be :

"myprog", "arg1", "arg2", "arg3", (char *) 0

argv[0] argv[1] argv[2] argv[3] argv[4]

Conventionally,Program name.
Any string allowed.

Logic Option

Execlp

- int execlp(const char *file, const char *arg, ...);

  execlp (“ls",

  “ls", “-l", “/etc/passwd", (char *) 0);

- file - filename of the file to be executed in the current process.
- The Shell variable PATH will be used to locate a file with name “ls” that has execute permission.
- If found, that executable will start executing in the current process. No change in PID or PPID.
- The command line arguments would be :

“ls", “-l", “/etc/passwd", (char *) 0

argv[0] argv[1] argv[2] argv[3]

Execv

- int execv(const char *path, char *const argv[]);

  char * a[]={ “ls", “-l", “/etc/passwd", (char *) 0};

  execv (“/bin/ls",a);

- path – Path of filename of the file to be executed in the current process.
- Uses a vector of arguments instead of a list.
- Last element of the vector should be a null pointer.

“ls",	“-l",	“/etc/passwd",	(char *) 0
argv[0]	argv[1]	argv[2]	argv[3]

Execvp

- int execvp(const char *file, char *const argv[]);

  char * a[]={ “ls", “-l", “/etc/passwd", (char *) 0};

  execvp (“ls",a);
- file – Filename of the file to be executed in the current process.
- Uses a vector of arguments instead of a list.
- Last element of the vector should be a null pointer.

“ls",	“-l",	“/etc/passwd",	(char *) 0
argv[0]	argv[1]	argv[2]	argv[3]

Combining fork and exec

```
if( (pid = fork ()) >0)
{
 wait( &status); /* Parent */
 if (WIFEXITED (status))
      printf ("Child terminated Normally.
}
else
{execl (  “/bin/ls",     /* child */
  "ls", "-l", "/etc/passwd", (char *) 0);


printf ("IF THIS IS PRINTED, \
                  THERE WAS AN ERROR\n");
}
```

Redirecting standard devices..

```
if( (pid = fork ()) >0)
{ wait( &status); /* Parent */ }
else
{
 int fd = open(“f1”,
              O_CREAT|O_WRONLY|O_TRUNC);
 close(1);
 dup(fd);
 execl (  “/bin/ls",     /* child */
  "ls", "-l", "/etc/passwd", (char *) 0);

}
```