Processes and Process System Calls in UNIX

- The process is the smallest unit of computation and the only active entity in the system.
- Each process is associated with (or runs) a single program and has a single thread of control.
- System calls for process management in UNIX are:

FORK, EXEC, WAIT, EXIT

(1) The FORK System Call

(man fork)

- FORK is the only way to create a new process in UNIX
- FORK creates an exact copy of the original process. (*Parent and child*)
- After the FORK system call has completed, the two processes continue executing from the point where they both returned from FORK.
- The two processes can distinguish who is the parent (original process) and who is the child (the new process) by testing the return value from FORK.
- The return value to the parent is the process ID of the child process (a positive number) and the return value to the child is zero.
- When FORK fails it returns a value of **-1**

getpid() a process can know its pid / man getpid */*

(2) The EXEC System Call

(man execv or man execl … etc) (man execve)

- A process can replace the program that it is currently running with a new program using the EXEC system call.
- In the most general case, EXEC has three parameters:
  - the name of the program to be executed
  - a pointer to the argument array
  - a pointer to the environment array
- Various library routines, including *execl*, *execv*, *execle*, and *execve* are provided to allow the parameters to be omitted or specified in various ways.
- If EXEC is successful, the text, data, and stack segments are replaced in the new program. Only the user area remains the same.


```
int execl (path, arg0, arg1, ..., argn, (char *) 0)
int execv (path, argv)
int execle(path, arg0, arg1,..., argn, (char *)0, envp)
int execve (path, argv, envp)
```

path

-- is pointer to a string containing the full or relative path name of an executable file

arg0, ..., argn

-- are pointers to strings that contain parameters to be passed to the new program. These values are placed in the **argv[]** argument of **main()** of the new program. The number of these arguments are placed in the **argc** argument to **main()**. The list of arguments must be terminated by a zero.

argv[]

-- is the array of pointers containing the same string pointers as **arg0, ..., argn**. The last element of **argv[]** must contain a zero address.

envp[]

-- is the array of new environment variables that you can pass to the new program. The values in this array are copied to the **envp[]** argument of **main()** of the new program. The last element of **envp[]** must contain a zero address.

5

(3)The WAIT System Call

(man –s 2 wait)

- A process can execute the WAIT system call to wait for one of its children (any one) to terminate. (*wait till child terminates*)

- WAIT has one argument, the address of the variable that will be set to the child's exit status (normal or abnormal termination and exit value)

- When WAIT is called, the following takes place:

  - If there are no outstanding children, it immediately returns with **-1** and **errno==ECHILD**

  - If the process has a zombie child, it returns with the process-ID and the exit-code of an arbitrarily chosen zombie child.

  - If there exists a child process that has not yet terminated, the calling process goes to sleep until a child terminates. At which time it returns with the process-ID and the exit-code of the terminated child.

6

If while waiting (at interruptable priority) it caches a signal, then it will return with a value of **-1** and **errno==EINTR**

- A process is a **zombie** for the period of time between its termination and the time the parent does a WAIT on it.

You can wait for a process to finish and then continue.

pid_t **waitpid**(pid_t pid, int *stat_loc, int options);

7

(4) The EXIT System Call

(man –s 3 exit)

- Processes terminate by executing the **EXIT** system call.

- An exiting process enters **zombie** state, relinquishing its resources and context except for its slot in the process table.

- You can specify a status value as an argument to **EXIT** that ranges in value from 0 to 255. By convention, a 0 value would indicate that the program terminated normally; values between 1 and 255 indicate that the program terminated because of some error condition:

  EXIT (status)

- The exit status is made available to the argument of the parent process's **WAIT** systems call. The value returned to the parent must be shifted right eight bits for it to be the same as the argument to **EXIT**.

8

A simplified shell illustrating FORK, WAIT, and EXEC:

```
while ( TRUE )
  {
        read_command (command, parameters);

        if (FORK() != 0 )
          {
                // parent does some work…
                WAIT (status);  /** parent waits */
                // after child finishes parent does more work
          }
        else
          {
        EXECVE (command, parameters, 0);  /** child works */
          }
  }
```


Example: ***cp xfile yfile***

- The shell process forks a new process (child)

- Child locates and executes program 'cp' and passes
  it the necessary parameters about the files to be copied.

- The main program of cp contains the declaration

    main (argc, argv, envp)

- argc gets 3, the number of items on the command
- argv[0] gets "cp"
- argv[1] gets "xfile"
- argv[2] gets "yfile"

Another example:

Client-server:
Server: - parent process accepts a connection
- forks a child
- the child handles the client.

Ex.
waitpid(xpid, statloc, opt)

the three parameters:
- pid of the child to wait for, or –1 for any child
- address variable to be set to child's exist code status
- option: for example whether caller terminates…etc.

Interprocess Communication

UNIX and POSIX IPC mechanisms for processes running on the same host:

1. *Pipes* and *FIFOs*
   (parent-child relationship between processes→pipe)
2. System V *message queues* (early 1980s)
   Posix *message queues* (1003.1b-1993)
3. *Remote Procedure Calls* (*RPCs*) (mid 1980s)

Forms of Synchronization in Unix

- *Record locking* (Unix early 1980s; Posix.1-1988)
- System V *shared memory* and *semaphores* (early 80s)
  Posix *shared memory* and *semaphores* (1003.1b-1993)
- Posix *mutexes* and *condition variables* (1003.1c-1995)
  -- Usually provide synchronization between threads
  -- Can provide synchronization between processes only if the mutex is stored in memory segment shared by the processes (e.g. in shared memory)
- *Read-write locks* (posix standard)

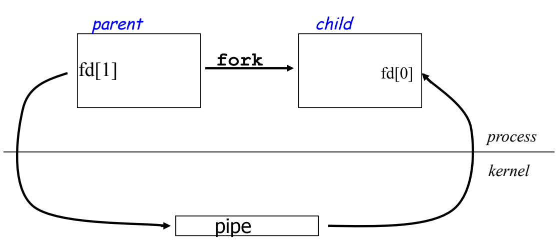
Three ways to share information between Unix processes:

1. through files in the file system,
2. through kernel address space (e.g. *pipes*, System V *message queues* and *semaphores*),
3. through *shared memory* (after set up, shared memory accesses do not involve kernel.)

Persistence of IPC Objects:

1. *Process-persistence* - remains in existence until last process closes it. e.g., pipes and FIFOs
2. *Kernel-persistence* - remains in existence until either the kernel reboots or it is explicitly deleted.
   e.g., System V IPCs.
3. *File system-persistence* - remains in existence until it is explicitly removed. e.g., Posix IPCs can optionally be implemented as such.

Example: **pipes**



pipe created by: int pipe(fd[2]); returns two integers fd[0], fd[1]
ex.: $who | sort $who | grep john

```
/********************************
 simple program: Unix pipes
*********************************/
main()
{
 int    pipefd[2], n;
 char   buff[100];

 if( pipe(pipefd) < 0 )
    printf("\n Error: pipe error\n");

 printf("\n read fd = %d, write fd=%d\n",pipefd[0],pipefd[1]);
 if( write(pipefd[1],"Hello World\n",12) != 12 )
     printf("\n pipe write error\n");

  if( (n = read(pipefd[0],buff, sizeof(buff))) <= 0 )
     printf("\n pipe read error\n");

  write(1, buff, n);   /** fd 1 = stdout **/
  exit(0);
}
```

```
read fd = 3, write fd=4
Hello World
```

17

FIFO

- First-In-First-Out
- Can be used between unrelated processes.
- One-way (half-duplex) flow of data

mkfifo(fifox): create a fifo

Then use standard I/O 'open' function to open a fifo for reading or writing (either read-only, or write-only)

open(fifox,….)

FIFO Example:

```
#include     "unpipc.h"
#define      FIFO1 "/tmp/fifo.1"
#define      FIFO2 "/tmp/fifo.2"
void  client(int, int), server(int, int);
```

18

```
int main(int argc, char **argv)
{   int         readfd, writefd;
    pid_t       childpid;
         /* create two FIFOs; OK if they already exist */
    if ((mkfifo(FIFO1, FILE_MODE) < 0) && (errno != EEXIST))
                err_sys("can't create %s", FIFO1);
    if ((mkfifo(FIFO2, FILE_MODE) < 0) && (errno != EEXIST)) {
                unlink(FIFO1);
                err_sys("can't create %s", FIFO2);
    }
    if ( (childpid = Fork()) == 0) {  /* child */
                readfd = Open(FIFO1, O_RDONLY, 0);
                writefd = Open(FIFO2, O_WRONLY, 0);
                server(readfd, writefd);
                exit(0);
        }
                /* parent */
        writefd = Open(FIFO1, O_WRONLY, 0);
        readfd = Open(FIFO2, O_RDONLY, 0);
        client(readfd, writefd);
        Waitpid(childpid, NULL, 0);   /* wait for child to terminate */
        Close(readfd);
        Close(writefd);
        Unlink(FIFO1);
        Unlink(FIFO2);
        exit(0);
}
```

19

Type of IPC	Persistence
Pipe, FIFO	Process
Posix mutex	Process
Posix condition variable	Process
Posix read-write lock	Process
Fcntl record locking	Process
Posix message queue	Kernel
Posix shared memory	Kernel
Posix named semaphore	Kernel
Posix memory-based semaphore	Process
System V message queue	Kernel
System V shared memory	Kernel
System V semaphore	Kernel
TCP socket	Kernel
UDP socket	Kernel
Unix domain socket	Kernel

Ex. Writing data to a file is file system persistent, but it is

20