Creating process in UNIX/LINUX

- Use fork() system call to create new processes
- The process making the system call is the parent
- The new process is called the child process
- Child process receives a copy of the address space of the parent
- The two processes are distinguished by the value returned
- A value of 0 is returned to the child and the ID of the child process is returned to the parent
- Both processes continue execution at the instruction after the fork()

Example 1

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
void main(){
   int   pid;
   pid = fork();
   switch (pid) {
   case -1 : printf(“ fork failed”);
     exit(-1);
   case 0 : printf(“I am the child, ID = %d\n”, getpid());
     exit(0);
   default : printf(“I am the parent, ID = %d\n”, getpid());
   }
}
```

Example 2

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
void main(){
   int   i, n;
   pid_t   pid;
   for (i = 0; i < n; i++)
          if (pid = fork() == 0)
   break;
          printf("My process ID = %d\n and my parent's ID = %d\n",    getpid(),  getppid());
}
```

- This can be used to create n children of a parent

wait() System Call

- The process making the wait() system call waits until its child completes or stops or until the caller receives a signal
- The *wait* returns immediately if the process has no child
- If the child terminates, the value returned is id of the child and is greater than 0
- wait() takes one argument which is an integer pointer and it stores the return status of the child

Example

```
#include   <stdio.h>
#include   <sys/wait.h>
#include   <sys/types.h>
#include   <unistd.h>
void main(){
   pid_t   pid;
   int   status;
   pid = fork();
   switch (pid) {
   case -1 : printf(“ fork failed”);
     exit(-1);
   case 0 : printf(“I am the child, ID = %d\n”, getpid());
     exit(0);
   default : if (wait (&status) != pid)
   printf(“A signal has interrupted the wait);
                  else
   printf(“I am the parent, ID = %d and child ID =%d\n”, getpid());
   }
}
```

exec() System Call

- fork() creates a duplicate of the calling process
- May require child to work on a different program
- exec() system call allows a new process to take the place of the calling process
- The new process starts executing at its main() function
- Usually use a combination of fork()-exec() sequence to let child execute a different program while the parent continues with the original program

Variations of exec()

- Six different functions
  - Differentiates the way command line arguments are given and the name of the program file is specified
- The six functions are:
  - execl()
    - List arguments with the call; list ends with (char *)0
  - execv()
    - Place arguments in a vector

Variations of exec()

- execle()
  - Same as execl() with additional argument that specifies the environment of the new program. In exec() forms in which environment is not provided the new program takes the environment of the parent
- execlp()
  - Same as execl(), but uses the PATH environment variable to search for the executable.

Variations of exec()

- execvp()
  - Same as execv(); uses the PATH environment variable to search for the executable
- execve()
  - Same as execv(); takes additional variable for the environment of the new program

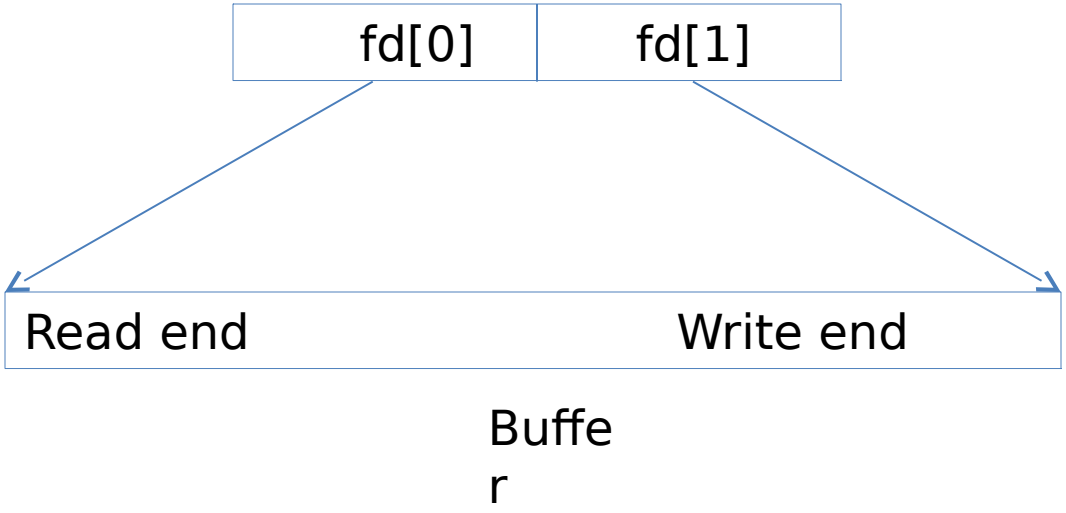
Example

```
#include        <stdio.h>
#include        <sys/types.h>
#include        <unistd.h>
#include        <stdlib.h>
#include        <sys/wait.h>
void main(){
        pid_t pid;
        int    status;
        pid = fork();
        switch (pid) {
        case -1 : printf(“ fork failed”);
                  exit(-1);
        case  0 :   printf(“I am the child, ID = %d\n”, getpid());
                execl(“/usr/bin/ls”, “ls”, -l, NULL);
        default : if (wait (&status) != pid)
                        printf(“A signal has interrupted the wait);
                         else
                        printf(“I am the parent, ID = %d and child ID =%d\n”, getpid());
        }
}
```

pipe() System Call

- Used for communications between parent and child
- The pipe(fd) system call creates a communication buffer, where fd[] is an integer array of two elements
- It returns two file descriptors fd[0] and fd[1].
- To read from the file, use fd[0]
- To write to the file, use fd[1]
- Use read() system call to read from the buffer
- Use write() system call to write to the buffer
- Buffer is organized as a FIFO queue

pipe() system call buffer



read()/write() system calls

- read() and write() system calls are defined as follows:

  ssize_t read(int fd, void *buf, size_t nbyte)

  read() requests to read nbyte bytes from the file pointed to by fd and placed in buf. buf should be large enough to hold nbyte bytes; returns number of bytes actually read

  ssize_t write(int fd, const void *buf, size_t nbyte)

  write() requests to write nbyte bytes from buf to the file with descriptor fd; returns the number of bytes actually written

Example

- *#include <errno.h>*
- *#include <stdio.h>*
- *#include <unistd.h>*
- *#include <stdlib.h>*
- *#include <sys/types.h>*
- *#define bufsize 20*

- *int fildes[2];*
- *char buffer[bufsize];*

- *void error(void)*
- *{*
- *   printf("unable to create a new process - job terminated \n");*
- *   fflush(stdout);*
- *}*

Example contd.

```
void child(void)
{
        int pm, i;
        printf("child starts working.\n");
        close(fildes[1]);
        for (i = 0; i<5; i++)
        {
              printf("in child i = %d \n", i);
              pm = read(fildes[0],&buffer,20);
              printf ("child received %d chars from parent\n",pm);
              printf ("message received is: %s ",buffer);
              fflush(stdout);
              sleep(1);
        }
       exit(0);
}
```

Example contd.

```
void parent(void)
{
   int i, pm;
        printf("parent starts working.\n");
        close(fildes[0]);
        for (i = 0; i < 5 ; i++)
            {
              switch(i){
              case 0:
            pm = write(fildes[1], "first message \n", 20);
                     printf("characters written  by parent = %d \n", pm);
            fflush(stdout);
                 break;
```

Example contd.

```
case 1:
   pm = write(fildes[1], "second message \n", 20);
   printf("characters written by parent = %d \n", pm);
   fflush(stdout);
break;
case 2:
   pm = write(fildes[1], "third message \n", 20);
   printf("characters written by parent = %d \n", pm);
   fflush(stdout);
break;
```

Example contd.

```
case 3:
      pm = write(fildes[1], "fourth message \n", 20);
      printf("characters written  by parent = %d \n", pm);
      fflush(stdout);
break;
case 4:
      pm = write(fildes[1], "fifth message \n", 20);
      printf("characters written  by parent = %d \n", pm);
      fflush(stdout);
            }
   }
       wait(NULL);
       printf("parent terminating\n");
       exit(0);
}
```

Example contd.

```
int main(void)
{
    int pp, pid;

    pp = pipe (fildes);
    printf("pipe opened with pp = %d \n", pp);
    fflush(stdout);
    pid = fork();
    if (pid == 0)
        child();
    else if (pid == -1)
                error();
           else  parent();
}
```

Pipe() System call

- pipe() creates a unidirectional communication buffer
- To have a two-way communication, need to make two pipe() system calls
- For read(), end-of-file condition for pipes occurs only when pipe is empty and there are no more processes with write descriptors open to the pipe. Need to close all descriptors not needed
- If a pipe is empty and write descriptors to the pipe are open, the process will wait