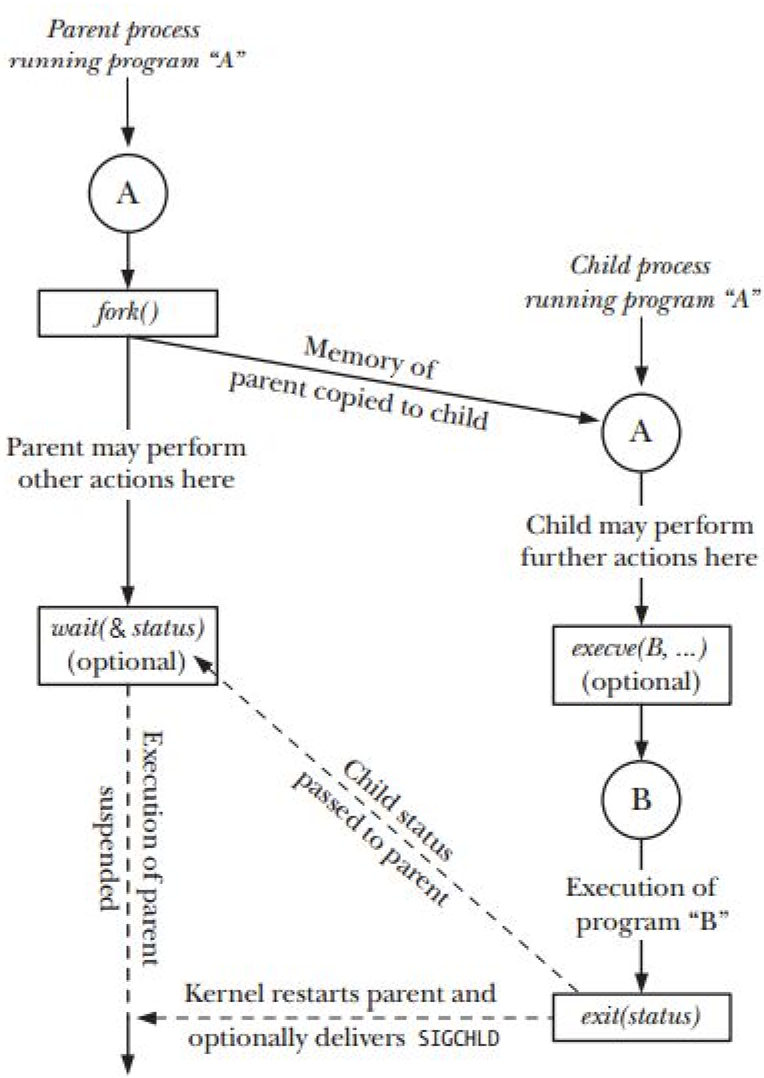
Advanced Operating System (CS G623)
First Semester 2019-2020

Objective:

1. To get familiarized with fork & pipe system call
2. Working of fork & pipe using C-Programing

Fork System Call



- Fork system call used for creates a new process, which is called the child *process*, which runs concurrently with process (which process called system call fork) and this process is called *parent process*. After a new child process created, both processes will execute the next instruction following the fork() system call. A child process uses the same PC(program counter), same CPU registers, same open files which use in the parent process.
- It takes no parameters and returns an integer value. Below are different values returned by fork().
- ***Negative Value***: creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process.
- The child is a copy of the parent. The child gets a copy of the parent's data section, heap, and stack. Memory is copied not shared.
- The parent and the child share the text segment.

```
#include <unistd.h>
pid_t fork(void);
/*Return process ID of child on success, or -1 on error; in
successfully created child : always return 0*/
```

- Within the code of the program, child and parent can be distinguished by the return value of fork().

- In parent return value > 0
- In child return value == 0

- In general, we never know whether the child starts executing before the parent or vice versa.

- To synchronize child and parent, some form of interprocess communication is required.

Fork: Practical Codes

The following program prints the some details of a process in which it is running.

1.

```
#include <unistd.h>
main () {
      printf ("I am running in a process whose details are as
follows\n");
      printf("process id (pid) = %d, parent process id(ppid) = %d,
user id(uid) = %d\n",getpid (), getppid (), getuid ());
}
```

2.Process Creation: a process is created by fork() system call. Consider the following program**.**

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
int glob = 6; //global variable
int main () {
      int var;
      pid_t pid;
      var = 88;
      printf ("Before fork\n");

      if ((pid = fork ()) < 0)
            perror ("fork");
      else if (pid == 0) {
            glob++;
            var++;
```

```
            printf ("pid = %d, glob=%d, var=%d\n", getpid (), glob,
      var);
            exit (0);
      }

      else {
            printf ("pid = %d, glob=%d, var=%d\n", getpid (), glob,
var);
            exit (0);
      }
}
```

3. wait and waitpid()

We will use the following program to understand wait() and waitpid() calls. Run the following program and observe the result of synchronization using wait().

File Name: wait.c

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
main () {
    int i = 0, j = 0;
    pid_t ret;
    int status;
    ret = fork ();

    if (ret == 0) {
        for (i = 0; i < 5000; i++)
            printf ("Child: %d\n", i);
        printf ("Child ends\n");
    } else {
        wait (&status);
        printf ("Parent resumes.\n");
        for (j = 0; j < 5000; j++)
            printf ("Parent: %d\n", j);
    }
```

```
}
```

4.What is the output of the code ?

a)

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

main()
{
    int val = 5;
    if(fork())
        wait(&val);

    val++;
    printf("%d\n", val);
    return val;
}
```

b)

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>

int
main()
{
    int pid1,pid2;
    printf("FIRST\n");
    pid1=fork();
```

```
    if(pid1==0) {
        printf("SECOND \n");
        pid2=fork();
        printf("SECOND \n");
    } else {
        printf("THIRD\n");
    }
}
```

i) How many total processes are created if the above code runs?
ii) How many times each of the strings “FIRST”,”SECOND” and “THIRD” in the above code printed?

c)
Given the following piece of code

```
main(int argc, char ** argv) {
    forkme(4);
}

void forkme(int n) {
    if(n > 0) {
        fork();
        forkme(n-1);

    }
}
```

If the above piece of code runs, how many processes are created?

Pipes

- We can think of pipe as piece of plumbing that allows data to flow from one process to another.

- A pipe is byte stream. No boundaries maintained between two writes of sender process.
- Pipes are unidirectional.
- Data can travel in only one direction.
- One end is used for reading and the other for writing.
- The pipe() system call creates a new pipe.
- Successful call return two file descriptors.
- Filedes[0] for read end and filedes[1] for write end.
- Normally pipe is used for communication between two processes. So fork() follows pipe() system call.

```
#include <unistd.h>
int pipe(int filedes[2]);
/*return 0 on success, -1 on error*/
```

– If there is a need for both parent and child to read and write data, then,
  - Using single pipe leads to race conditions. Can be avoided using some synchronizations mechanism.
  - Simpler is to use to two pipes, one in each direction.

- This may lead to a deadlock situation. Both parent and child blocked in reading but there is no data in the pipes.

- Not only parent and child but any two processes having a common ancestor can use pipe provided that common ancestor has created the pipe.

- Process reading from pipe closes write end of the pipe. Why?
  - While reading from pipe an EOF is encountered only if there are no more write ends open.
  - If not closed, the read may block indefinitely waiting.

- Process writing to pipe closes read end of the pipe. Why?

- If a process tries to write to a pipe for which there is no read end open, then kernel generates SIGPIPE signal. This signal has default action, terminate process.
- If the process doesn't close read end, process will still be able to write to the pipe, once full it will indefinitely blocked waiting for someone to read the pipe.

Pipes: Practical

1) **File Name. pipe.c**

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

#define MSGSIZE 16
main ()
{
  int i;
  char *msg = "How are you?";
  char inbuff[MSGSIZE];
  int p[2];
  pid_t ret;
  pipe (p);
  ret = fork ();
  if (ret > 0)
    {
      i = 0;
      while (i < 10)
        {
          write (p[1], msg, MSGSIZE);
          //sleep (2);
          read (p[0], inbuff, MSGSIZE);
          printf ("Parent: %s\n", inbuff);
          i++;
        }
    exit(1);
    }
```

```
  else
    {
      i = 0;
      while (i < 10)
        {
         sleep (1);
          read (p[0], inbuff, MSGSIZE);
          printf ("Child: %s\n", inbuff);
          write (p[1], "i am fine", strlen ("i am fine"));
          i++;
        }
    }
  exit (0);
}
```

2. File Name: filter.c

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
void
err_sys (char *str)
{
  perror (str);
  exit (-1);
}

int
main (void)
{
  int c;

  while ((c = getchar ()) != EOF)
    {
      if (islower (c))
     c = toupper (c);
      if (putchar (c) == EOF)
     err_sys ("output error");
      if (c == '\n')
```

```
      fflush (stdout);
    }
  exit (0);
}
```

3. File Name: parent.c

```
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>

void err_sys(char* str)
{
     perror(str);
          exit(-1);
          }

#define MAXLINE 80
  int
main (void)
{
  char line[MAXLINE];
  FILE *fpin;

  if ((fpin = popen ("./filter", "r")) == NULL)
    err_sys ("popen error");
  for (;;)
    {
      fputs ("prompt> ", stdout);
      fflush (stdout);
      if (fgets (line, MAXLINE, fpin) == NULL)    /* read from pipe */
     break;
      if (fputs (line, stdout) == EOF)
     err_sys ("fputs error to pipe");
    }
  if (pclose (fpin) == -1)
    err_sys ("pclose error");
  putchar ('\n');
```

```
  exit (0);
}
```