CSci 4061 Introduction to Operating Systems

Recitation 6
Process Management and Pipe
10/16/2017
TA- Shalini Pandey

A Process

- What is a process in Unix?
  - A process is a program in execution
  - Difference between a process and program
- Different states
  - READY
  - RUNNING
  - WAITING

The all_ids.c example

- How are different processes identified?
  - – Process ids

- ‘pid’ - Process id of a process.
- ‘ppid’ - Process id of the parent process.

all_ids.c

```
printf("I am a process with an id %ld\n", (long)getpid());
printf("The Id of my parent is %ld\n", (long)getppid());
printf("My real user id is %ld\n", (long)getuid());
printf("My effective user id is %ld\n", (long)geteuid());
printf("My real group id is %ld\n", (long)getgid());
printf("My effective group id is %ld\n", (long)getegid());
```

Process pool

- How do you find out what processes your system is running currently?
- *ps –a*
- *man ps*
- *Options*
  - *-e -a -f*

fork()

- fork() – Creates a new process
- If fork() fails, it returns -1 and sets a errno to EAGAIN
- If fork() succeeds, it returns 0 to the child and the child’s pid to the parent.
- Potential pitfalls:
  - duplicate memory (file pointers) can provide intermixed output.

Exercise Problem 1

Problem Statement: In fork_ex.c modify the code such that when child process is running it prints "I am a child with id %process id of child%" while the parent prints "I am a parent with id %process id of parent%"

Resource file: fork_ex.c

fork_ex.c

```
pid_t childpid;

childpid = fork();
if (childpid == -1)
{
        perror("fork() failed");
        return 1;
}
if (childpid == 0)
        printf("I am a child with id %ld\n", (long)getpid());
else
        printf("I am a parent with id %ld\n", (long)getpid());
return 0;
```

wait()

- When a process creates a child, both parent and child proceed execution from the point of *fork()*
- The parent can execute *wait()* or *waitpid()* to block until the child executes
- wait() : waits for the termination of one of the children
- waitpid() : waits for the termination for specified child process

waitpid()

- man waitpid and see the options available.
    1. WUNTRACED
    2. WNOHANG
    3. WCONTINUED

wait_ex.c

```
pid_t childpid;
pid_t waitreturn;
int status;
childpid = fork();

if(childpid==-1)
{
        perror("fork");
        exit(0);
}
else if(childpid==0){
        printf("I am a child\n");
        exit(3);
}
else {
        waitreturn = wait(&status);
        if(WIFEXITED(status)) {
                printf("child exited with status %d\n",WEXITSTATUS(status));
        }
}
```

Exercise Problem 2

Problem Statement: Modify process_fan.c so that it waits for the second child.

Resources: process_fan.c.

process_fan.c

```
int i;
int n = 4;
int waitstat;
pid_t childpid;
pid_t second_childpid;
for ( i=0;  i < n; i++ )  {

        childpid = fork();
         if(i==1)
              second_childpid = childpid;

          if ( childpid == 0){
            /* I just created a child */
            break;
      }
}

waitpid(second_childpid, &waitstat, 0);
printf( "Process-ID: %-8ld, Parent-Process-ID: %-8ld\n",  (long)getpid(), (long)getppid() );
```

Exercise Problem 3

Problem Statement: Use waitpid and WNOHANG to modify wait_ex.c so that parent does not wait for the child process to finish.

Resource file: wait_ex.c

wait_ex.c

```
    pid_t childpid;
    pid_t waitreturn;
    int status;
    childpid = fork();
    if(childpid==-1)
    {
        perror("fork");
        exit(0);
    }
    else if(childpid==0){
        printf("I am a child\n");
        exit(3);
    }
    else {
        waitreturn = waitpid(childpid, &status, WNOHANG);
        if(WIFEXITED(status)) {
            printf("child exited with status %d\n",WEXITSTATUS(status));
        }
    }
}
```

exec()

- exec – execute a shell command or program
- Six of them – *execl, execlp* and *execle* form one family while *execv, execvp* and *execve* form the other
- *man* them all – On your own time!

Exercise Problem 3

Problem statement:

- *man* execl
- Modify execl_ex.c such that the child executes ps -af command.
- Try both execl and execlp commands for this task.

execl_ex.c

```
pid_t childpid;
 childpid = fork();
 if(childpid==-1){
        perror("Failed to fork");
        return 1;
 }
 //child code
 if(childpid == 0){
        execl("/bin/ps", "ps", "-af",NULL);
       execlp( "ps", "-a”,”f",NULL);
        perror("child failed to exec all_ids");
        return 1;
 }
 if(childpid != wait(NULL)){
        perror("parent failed to wait due to signal or error");
        return 1;

 }
```

kill.c

```
pid_t childpid;

 childpid = fork();
if (childpid == -1)
{
        perror("fork() failed");
        return 1;
}
 printf("childpid == %d\n",childpid);
if (childpid == 0){
        printf("I am a child with id %ld\n", (long)getpid());
        printf("Terminating process %ld \n", (long)getpid());
        kill((long)getpid(), 9);
}else{
        printf("I am a parent with id %ld\n", (long)getpid());
        printf("Terminating process %ld \n", (long)getpid());
        kill((long)getpid(), 9);
}
```

PIPES

pipe()

- pipe – create descriptor pair for interprocess communication
- A pipe is a unidirectional communication channel between UNIX processes. By this we mean that a pipe can be written to on one end and read from at the other.
- pipe() - creates a pipe and place two file descriptors, one each into the arguments fildes[0] and fildes[1], that refer to the open file descriptions for the read and write ends of the pipe.