ECE437/CS481

M02B: PROCESSES & THREADS

PROCESSES CREATION

Chapter 3.3

Xiang Sun

The University of New Mexico

Process Creation & Termination

- New processes are created by existing processes
  - creator is called the parent
  - created process is called the child
  - Linux: do ps, look for PID field

```
home@VirtualBox:~$ ps ux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
home       1114  0.0  0.8  46548  8512 ?        Ssl  Sep03   0:00 gnome-sess
home       1151  0.0  0.0   3856   140 ?        Ss   Sep03   0:00 /usr/bin/s
home       1154  0.0  0.0   3748   484 ?        S    Sep03   0:00 /usr/bin/d
home       1155  0.1  0.2   6656  3036 ?        Ss   Sep03   0:18 //bin/dbus
home       1157  0.0  0.2   9148  2368 ?        S    Sep03   0:00 /usr/lib/g
home       1162  0.0  0.2  31588  2296 ?        Ssl  Sep03   0:00 /usr/lib/g
home       1174  0.0  1.4 132472 14884 ?        Sl   Sep03   0:03 /usr/lib/g
```

- What and when creates the first process?
  - init is the first process with PID=1. It is started directly by the kernel.
  - All other programs are either started directly by init or by one of its child processes.
  - The entire process of starting the system and shutting it down is maintained by init.

Process Creation & Termination

- Create a new process via fork () system call
  - child process get a copy of the address space of the parent process
  - both processes continue execution at the instruction after the fork()
  - If fork() succeeds:
    - ✓ return 0 to the child
    - ✓ return child's PID to the parent

    } The fork() system call "returns twice"
  - If fork() fails:
    - ✓ no child process is created, and it returns -1 in the parent process

Process Creation & Termination

Synopsis/Syntax of fork()

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Template for using fork()

```
pid_t pid;
if ((pid=fork()) == 0) {
/* put code for child here */ }
else if (pid < 0) {
/* fork failed, put error handling here */ }
else {
/* fork successful; put remaining code for parent here */ }
```

Process Creation & Termination

❑ Example of fork()

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

int main()
{
pid_t pid = fork();
if (pid == 0)
        printf("\tI'm the child process, and I received %d\n", pid);

else
        printf("I'm the parent process, and my child PID is %d\n", pid);
return 0;
}
```

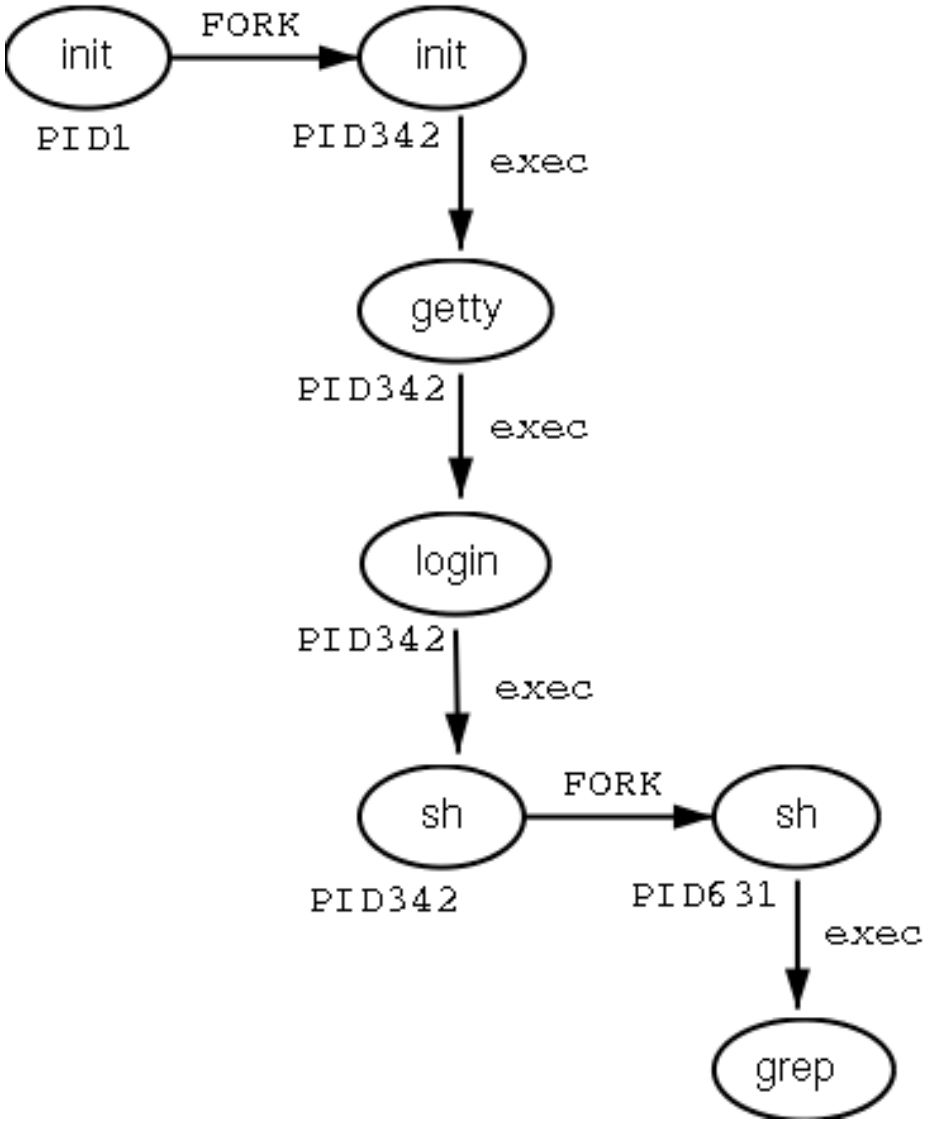
Output:

```
I'm the parent process, and my child PID is 2172
I'm the child process, and I received 0
```

Process Creation & Termination

- Execute a new program via execve() system call
  - execve() is called to let the new program overwrite the calling program totally
  - the address space of the calling process is reinitialized
  - there is no return from a successful call to exec
    - ✓ return -1, if execve() fails
  - an execve() call often follows fork() to create a new process that runs another program.
    - ✓ Process A (the parent process) calls fork() to create a child Process B;
    - ✓ Process B immediately calls execve() to run a new program.

Process Creation & Termination



Process Creation & Termination

❑ Synopsis/Syntax of execve()

```
#include <unistd.h>
int execve(const char *filename, char *const argv[], ..., char *const *envp[]);
```

- ✓ use "man -s 2 execve" to see more details
- ✓ Both argv and envp must be terminated by a null pointer.

Process Creation & Termination

❑ Example of execve()

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int
main(int argc, char **argv) {
        char *args[] = {"/bin/ls", "-m", 0};/* each element represents a command line argument */
        char *env[] = { 0 };    /* leave the environment list null */
        printf("About to run /bin/ls\n");
        execve("/bin/ls", args, env);
        perror("execve");       /* if we get here, execve failed */
        exit(1);
}
```

Output:

```
shaun@shaun-VirtualBox:~$ ./test1
About to run /bin/ls
Desktop, Documents, Downloads, Music, Pictures, Public, Templates, Videos,
a.out, examples.desktop, hello, hello.c, test.c, test1, test1.c
```

Process Creation & Termination

- Terminate a process via kill() system call
  - a process may terminate other process if it has such a privilege
  - terminate another process by kill() system call, which sends a signal to a process specified by PID.

- Synopsis/Syntax of kill()

```
#include <unistd.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Use "man -s 2 kill" to see more details

Number	Name (short name)	Description	Used for
0	SIGNULL (NULL)	Null	Check access to pid
1	SIGHUP (HUP)	Hangup	Terminate; can be trapped
2	SIGINT (INT)	Interrupt	Terminate; can be trapped
3	SIGQUIT (QUIT)	Quit	Terminate with core dump; can be trapped
9	SIGKILL (KILL)	Kill	Forced termination; cannot be trapped
15	SIGTERM (TERM)	Terminate	Terminate; can be trapped
24	SIGSTOP (STOP)	Stop	Pause the process; cannot be trapped. This is default if signal not provided to kill command.
25	SIGTSTP (STP)	Terminal	Stop/pause the process; can be trapped
26	SIGCONT (CONT)	Continue	Run a stopped process

Process Creation & Termination

❑ Example of kill()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

int main(void){
    pid_t retVal;
    retVal = fork();
    if(retVal > 0){
        int i = 0;
        while(i++ < 5){
            printf("in the parent process.\n");
            sleep(1);
        }
        //kill the child process
        kill(retVal, SIGKILL);

    } else if (retVal == 0){
        int i = 0;
        //will not ever get to 15, because
        //the parent process will kill it
        while(i++ < 15){
            printf("In the child process.\n");
            sleep(1);
        }
    } else {
        //something bad happened.
        printf("Something bad happened.");
        exit(EXIT_FAILURE);
    }
    return 0;
```

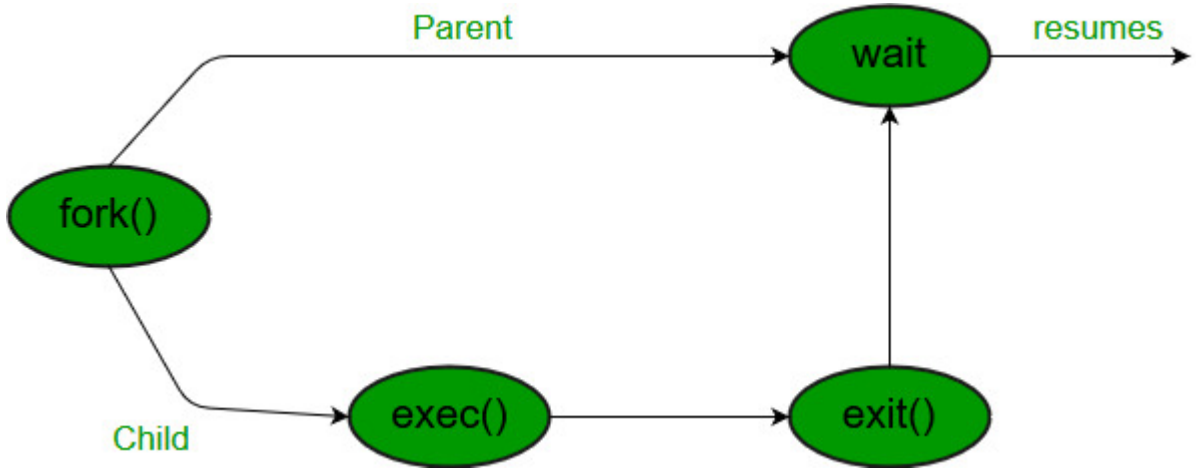
Output:

```
shaun@shaun-VirtualBox:~$ ./test_wait
in the parent process.
In the child process.
in the parent process.
In the child process.
in the parent process.
In the child process.
in the parent process.
In the child process.
in the parent process.
In the child process.
```

Process Creation & Termination

❑ Wait for termination of a process via wait() system call

- A parent process can temporarily suspend its execution while one of its child process is running.
- After child process terminates, parent continues its execution after waiting system call instruction.
- Child process may terminate due to any of these:
  - ✓ It calls exit();
  - ✓ It returns (an int) from main
  - ✓ It receives a signal (from the OS or another process) whose default action is to terminate.



Process Creation & Termination

- Wait for termination of a process via wait() system call
  - If only one child process is terminated, then wait() returns process ID of the terminated child process.
  - If more than one child processes are terminated, then wait() reap any arbitrarily child and return a process ID of that child process.
  - If any process has no child process, then wait() returns -1.

- Synopsis/Syntax of wait()

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

pid_t wait(int &status);
```

Process Creation & Termination

❑ Example of wait()

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

int main()
{
    pid_t pid=fork();
    if (fork()== 0)
        printf("Child is running\n");
    else
    {
        printf("Parant is running\n");
        wait(NULL);
        printf("Child has been terminated\n");
    }

    return 0;
}
```

Output:

```
shaun@shaun-VirtualBox:~$ ./test_wait1
Parant is running
Child is running
Child has been terminated
```

Process Creation & Termination

❑ Child status information in wait(&status)

- Parent process can see the termination info of the child reported by wait.
- For finding the termination info we use WIF… macros

1. WIFEXITED(status):returns true if child exited normally
   WEXITSTATUS(status): return code when child exits.

2. WIFSIGNALED(status): returns true if the child process was terminated by a signal.
   WTERMSIG(status): gives the number of the terminating signal.

3. WIFSTOPPED(status): returns true if the child process was stopped by delivery of a signal
   WSTOPSIG(status): gives the number of the stop signal.

- Find more info about WIF… macros via http://man7.org/linux/man-pages/man2/waitpid.2.html

Process Creation & Termination

❑ Example of wait(&status)

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

void waitexample()
{
    int stat;
    if (fork() == 0)
        exit(1);
    else
        wait(&stat);
    if (WIFEXITED(stat))
        printf("Exit status: %d\n", WEXITSTATUS(stat));
    else if (WIFSIGNALED(stat))
        psignal(WTERMSIG(stat), "Exit signal");
}


int main()
{
    waitexample();
    return 0;
}
```

Output:

```
shaun@shaun-VirtualBox:~$ ./test_wait2
Exit status: 1
```

Process Creation & Termination

- Wait for termination of a specific process via waitpid() system call
- Synopsis/Syntax of wait()

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

pid_t waitpid(child_pid, &status, options);
```

  - Options Parameter
    1) If 0, it means no option, i.e., parent has to wait for terminates child.
    2) If WNOHANG, it means parent does not wait if child does not terminate. (not block parent process)
    3) If WIMTRACED, waitpid() also return if a child has stopped.
    4) If WCONTINUED, waitpid() also return if a stopped child has been resumed by delivery of SIGCONT.
  - Find more info about the child_pid and options parameter via http://man7.org/linux/man-pages/man2/waitpid.2.html

Process Creation & Termination

❑ Example of waitpid()

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

void waitexample()
{
    int i, stat;
    pid_t pid[5];
    for (i=0; i<5; i++)
    {
        if ((pid[i] = fork()) == 0)
        {
            sleep(1);
            exit(100 + i);
        }
    }

    // Using waitpid() and printing exit status
    // of children.
    for (i=0; i<5; i++)
    {
        pid_t cpid = waitpid(pid[i], &stat, 0);
        if (WIFEXITED(stat))
            printf("Child %d terminated with status: %d\n",
                   cpid, WEXITSTATUS(stat));
    }
}

int main()
{
    waitexample();
    return 0;
}
```

Output:

```
shaun@shaun-VirtualBox:~$ ./test_waitpid
Child 2701 terminated with status: 100
Child 2702 terminated with status: 101
Child 2703 terminated with status: 102
Child 2704 terminated with status: 103
Child 2705 terminated with status: 104
```

Process Creation & Termination

❑ Zombie process

- When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table as well its PCB must remain there until the parent calls wait(), because the process table contains the process's exit status.

- Zombie process: A process that has terminated, but whose parent has not yet called wait().

- Once the parent calls wait(), the process ID of the zombie process and its entry in the process table are released.

- Zombie processes only occupy few system resources (to store their PCBs). However, each zombie process retains its process ID (PID), and total number of PIDs is limited.
  - ✓ Consuming all the available PID pool, thus preventing creating new processes.

Process Creation & Termination

❑ How to kill a zombie process

- Solution_1: sending the SIGCHLD signal to the parent process. This signal tells the parent process to execute the wait() system call and clean up its zombie children.

  kill -s SIGCHLD pid

  Assumption: the parent process should be programmed properly (i.e., invoke wait()) when SIGCHILD signals are received.

- Solution_2: killing the zombie's parent process.
  - ✓ All its child process (including zombie process) become orphan processes.
  - ✓ Init process (pid=1) becomes the new parent to orphan processes.
  - ✓ Init process periodically executes the wait() system call to clean zombie process.
  - ✓ [option] Restart the previous parent process.

Process Creation & Termination

More about Unix/Linux Process's Attributes

- Process PID
  - ✓ each process has a non-negative, system-unique # as its PID
  - ✓ some fixed processs ID, such as PID=1, init process
  - ✓ obtain its own process ID by calling getpid(void)
  - ✓ obtain its parent process ID by calling getppid(void)

Process Creation & Termination

❑ More about Unix/Linux Process's Attributes

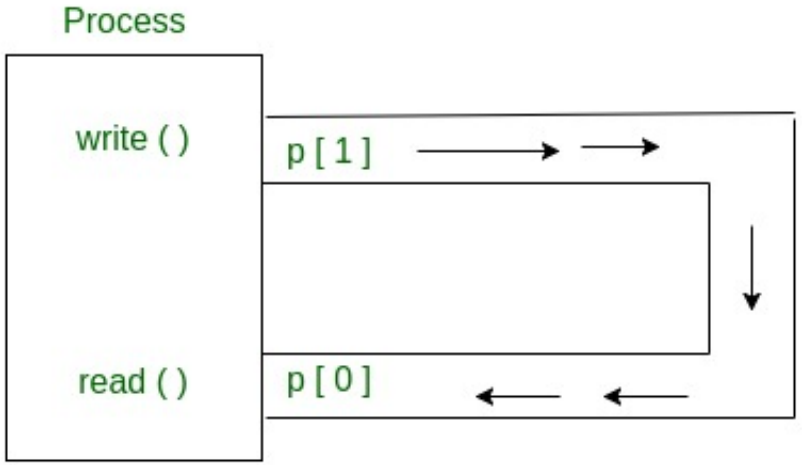
➢ Process group ID (GID)

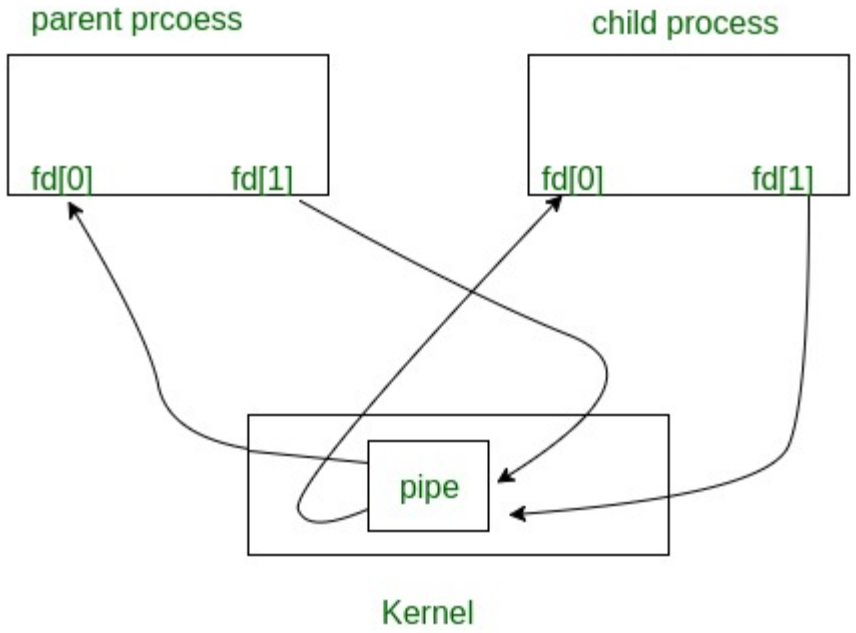
- ✓ process group == a set of related processes
  process group is used to control the distribution of signals.
- ✓ when the process is created, it becomes a member of the process group of its parent.
- ✓ process group ID (GID)=PID of the process group leader (normally, the first member of the process group)
- ✓ a process finds the ID of its process group using the system call getpgrp()
- ✓ a process finds another process p's GID using getpgid(p)
- ✓ put a process into a process group using setpgid(pid, pgid)
- ✓ create a new process group with process group leader pid using setpgid(pid, 0)
- ✓ setpgrp()=setpgid(0, 0), which is to create a new process with process group leader equaling to the current process.

❑ More about Unix/Linux Process's Attributes

➢ pipe() system call

- ✓ a pipe is a connection among multiple processes.
- ✓ pipe is one-way communication (one process write to the pipe, and the other process reads from the pipe). If a pipe is created, there is an area of main memory that is treated as a "virtual file". One process can write to this "virtual file" and another related process can read from it later on.





Process Creation & Termination

❑ Example of process group and pipe()

```c
#define _POSIX_SOURCE
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>

main() {
  pid_t pid;
  int p1[2], p2[2];
  char c='?';

  if (pipe(p1) != 0)
    perror("pipe() #1 error");
  else if (pipe(p2) != 0)
    perror("pipe() #2 error");
  else
    if ((pid = fork()) == 0) {
      printf("child's process group id is %d\n", (int) getpgrp());
      write(p2[1], &c, 1);
      read(p1[0], &c, 1);
      puts("child is waiting for parent to complete task");
      printf("child's process group id is now %d\n", (int) getpgrp());
      exit(0);
    }
    else {
      printf("parent's process group id is %d\n", (int) getpgrp());
      read(p2[0], &c, 1);
      printf("parent is performing setpgid() on pid %d\n", (int) pid);
      if (setpgid(pid, 0) != 0)
        perror("setpgid() error");
      write(p1[1], &c, 1);
      printf("parent's process group id is now %d\n", (int) getpgrp());
      sleep(5);
    }
}
```

```
shaun@shaun-VirtualBox:~$ ./test_setpgrp
parent's process group id is 2722
child's process group id is 2722
parent is performing setpgid() on pid 2723
parent's process group id is now 2722
child is waiting for parent to complete task
child's process group id is now 2723
```

Process Creation & Termination

❑ More about Unix/Linux Process's Attributes

- ➢ Session ID (SID)

  ✓ session = a set of process groups sharing a control terminal

  --Linux is a multi-user system. Usually, when a user log in the system, a new session is created. When the user log out, the session will be terminated.

  ✓ when the process is created, it becomes a member of the session of its parent

  ✓ the session ID of a session = the process ID of the first member of the session

  ✓ a process finds the ID of its session using the system call getsid().

  ✓ obtain its own process ID by calling getsid()

  ✓ start a new session by calling setsid(void) (return the new SID)