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 the existing processes
  - Creating process is called parent
  - Created process is called 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
```

- Each process can create/spawn child processes, and each process has parent except for the top-most process.

- Who and when creates top-most process?
  - init/systemd is the top-most process with PID=1. It's created directly by the kernel.
  - All other processes are either started directly by init/systemd or by one of its child processes.

Process Creation & Termination

- Spawn a new process via fork() system call
  - A child process get a copy of the address space of its 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;
pid=fork();
if (pid== 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

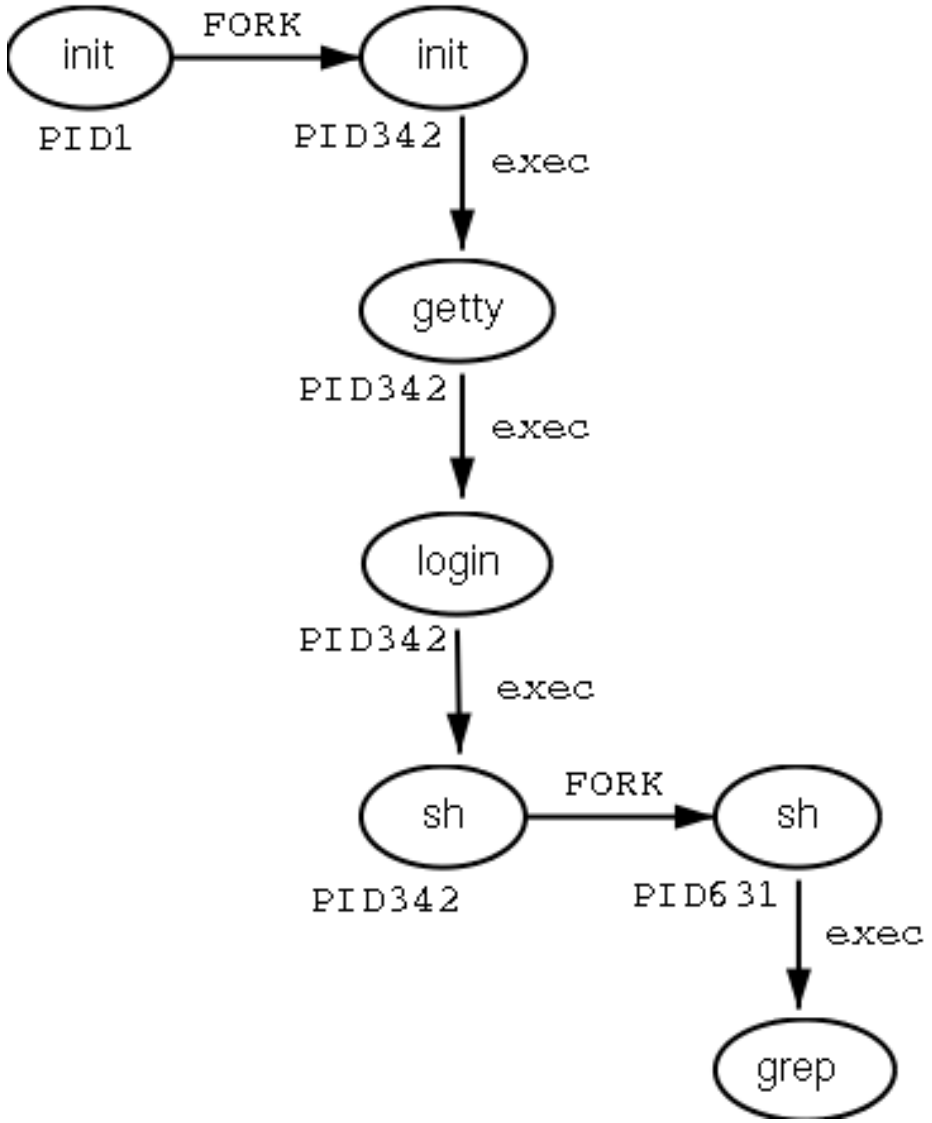
- Spawn a new process via execve() system call
  - execve() is called to let the new process overwrite the calling process totally
  - The address space of the calling process is reinitialized
  - There is no return from a successful call to execve()
    - ✓ return -1, if execve() fails
  - An execve()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

Process Creation & Termination

- Example of process creations

  - ✓ fork() creates a child process with a different PID.

  - ✓ execve() overwrites a process with the same PID.



Process Creation & Termination

❑ Synopsis/Syntax of execve()

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

- ✓ use "man -s 2 execve" to see more details
- ✓ Both argv and envp must end with 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 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;
```

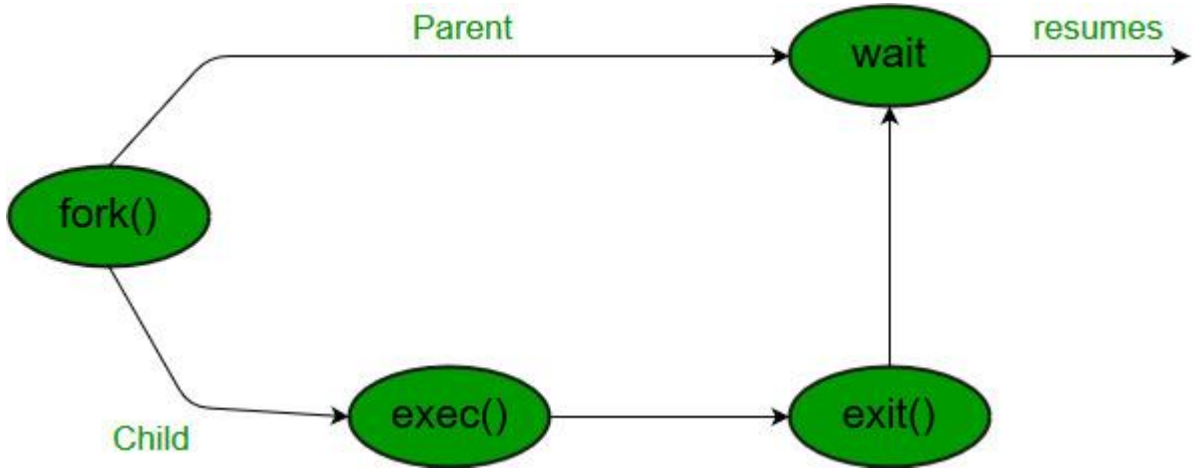
Output:

```
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 in order to let its child process to be execute.
- 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) in main();
  - ✓ It receives a signal (from the OS or another process) whose action is to terminate.



Process Creation & Termination

- Wait for termination of a process via wait() system call
  - If only one child process was created, then wait() suspends the parent and returns process ID of the child process once it is terminated.
  - If more than one child processes were created, then wait() suspends the parent process until one of its child processes terminates.
  - If the 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;
}
```

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

int main()
{
   pid_t pid1=fork();
   pid_t pid2=fork();
   if (pid2==0)
   {
        printf("Child is running\n");
   }
   else
   {
        printf("Parent is running\n");
        wait(NULL);
        printf("Child has been terminated\n");
   }
   return 0;
}
```

Process Creation & Termination

❑ Example of wait()

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

int main()
{
    pid_t pid1=fork();
    pid_t pid2=fork();
    if (pid2==0)
    {
        printf("Child is running\n");
    }
    else
    {
        printf("Parent is running\n");
        wait(NULL);
        printf("Child has been terminated\n");
    }
    return 0;
}
```

Output:

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

Process Creation & Termination

❑ Child status information in wait(*status)

- Status indicates the termination info of the child process reported by wait.
- Termination info: encode both the termination status of the process as well as the reason of termination.
- In order to find the termination info, WIF… macros is applied. For example,
  1. WIFEXITED(status): returns true if child is terminated by exit().
     WEXITSTATUS(status): return the parameter of exit() if WIFEXITED(status) is true.
  2. WIFSIGNALED(status): returns true if the child process was terminated by a signal.
     WTERMSIG(status): gives the index of the terminating signal if WIFSIGNALED(status) is true.
- 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 waitpid()

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

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

  - options:
    1) If 0, it means no option, i.e., the parent has to wait for the termination of the target child process. Return: target child process PID.
    2) If WNOHANG, it means parent does not wait if the target child process does not terminate. (not block parent process). Return: target child process PID if it is terminated, or 0 if it is not terminated.
    3) If WUNTRACED, waitpid() also returns if the target child process has stopped.
    4) If WCONTINUED, waitpid() also returns if the stopped child process has been resumed by delivery of SIGCONT.
  - Find more info about waitpid() 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

```
1. /* waitpid.c */
2. #include <sys/types.h>
3. #include <sys/wait.h>
4. #include <unistd.h>
5. #include <stdio.h>
6.
7. int main()
8. {
9.     pid_t pc, pr;
10.
11.    pc = fork();
12.    if ( pc < 0 )
13.    {
14.        printf("fork error\n");
15.        exit(1);
16.    }
17.    else if ( pc == 0 )
18.    {
19.        sleep(10);
20.        exit(0);
21.    }
22.    else
23.    {
24.        do {
25.            pr = waitpid(pc, NULL, WNOHANG);
26.            if ( pr == 0 )
27.            {
28.                printf("No child exit\n");
29.                sleep(1);
30.            }
31.           }while (pr == 0 );
32.        if ( pr == pc )
33.            printf("successfully get child %d\n", pr);
34.        else
35.            printf("wait child error\n");
36.    }
37.    return 0;
38. }
```

```
$ ./waitpid
No child exit
No child exit
No child exit
No child exit
No child exit
No child exit
No child exit
No child exit
No child exit
No child exit
successfully get child 4607
```

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 as its PCB are still maintained until its parent calls wait().
- A zombie process is a process that has terminated, but its parent has not yet called wait().
- 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.

```
int main()
{
    // Fork returns process id
    // in parent process
    pid_t child_pid = fork();

    // Parent process
    if (child_pid > 0)
        sleep(50);

    // Child process
    else
        exit(0);

    return 0;
}
```

Process Creation & Termination

Zombie process

- In the Ubuntu OS system, if a process that is still existed in the system, but its parent has terminated and has not called wait(). This process is also considered a zombie process.

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

int main ()
{
   pid_t child_pid;
   child_pid = fork();
   if (child_pid > 0) {
     printf ("child process ID %d\n",child_pid);
     exit(0); }
  else {
     printf("I am the child process before sleep.\n");
     sleep(20);
     printf("I am the child process after sleep.\n");
    exit (0);}
}
```

Process Creation & Termination

- Zombie process
  - In the Ubuntu OS system, if a process that is still existed in the system, but its parent has terminated and has not called wait(). This process is also considered a zombie process.

```
shaun       5971  0.0  0.2 581628  2428 ?        Sl   10:05   0:00 /usr/libexec/gnome-shell-calendar-server
shaun       5977  0.0  0.2 464156  2448 ?        Ssl  10:05   0:00 /usr/libexec/evolution-source-registry
shaun       5985  0.0  0.3 707748  3652 ?        Ssl  10:05   0:00 /usr/libexec/evolution-calendar-factory
shaun       5995  0.0  0.2 156200  2840 ?        Sl   10:05   0:00 /usr/libexec/dconf-service
shaun       6000  0.0  0.3 681664  3692 ?        Ssl  10:05   0:00 /usr/libexec/evolution-addressbook-factory
shaun       6018  0.0  0.3 317548  3304 ?        Sl   10:05   0:00 /usr/libexec/gvfsd-trash --spawner :1.3 /org/gtk/gvfs/exec_sp
shaun       6023  0.0  0.1 2598772 1912 ?        Sl   10:05   0:00 /usr/bin/gjs /usr/share/gnome-shell/org.gnome.Shell.Notificat
shaun       6036  0.0  0.2 313920  2584 ?        Ssl  10:05   0:00 /usr/libexec/gsd-a11y-settings
shaun       6038  0.0  0.6 570392  6936 ?        Ssl  10:05   0:00 /usr/libexec/gsd-color
shaun       6040  0.0  0.2 374364  2588 ?        Ssl  10:05   0:00 /usr/libexec/gsd-datetime
shaun       6041  0.0  0.2 314164  2968 ?        Ssl  10:05   0:00 /usr/libexec/gsd-housekeeping
shaun       6042  0.0  0.5 348352  5300 ?        Ssl  10:05   0:00 /usr/libexec/gsd-keyboard
shaun       6046  0.0  0.9 690548  9096 ?        Ssl  10:05   0:00 /usr/libexec/gsd-media-keys
shaun       6047  0.0  0.6 348772  6140 ?        Ssl  10:05   0:00 /usr/libexec/gsd-power
shaun       6051  0.0  0.3 248100  2996 ?        Ssl  10:05   0:00 /usr/libexec/gsd-print-notifications
shaun       6053  0.0  0.3 457100  3060 ?        Ssl  10:05   0:00 /usr/libexec/gsd-rfkill
shaun       6054  0.0  0.2 235508  2468 ?        Ssl  10:05   0:00 /usr/libexec/gsd-screensaver-proxy
shaun       6057  0.0  0.3 469040  3640 ?        Ssl  10:05   0:00 /usr/libexec/gsd-sharing
shaun       6060  0.0  0.2 318080  2536 ?        Ssl  10:05   0:00 /usr/libexec/gsd-smartcard
shaun       6063  0.0  0.2 321784  2472 ?        Ssl  10:05   0:00 /usr/libexec/gsd-sound
shaun       6065  0.0  0.2 387884  2464 ?        Ssl  10:05   0:00 /usr/libexec/gsd-usb-protection
shaun       6073  0.0  0.4 274188  4908 ?        Ssl  10:05   0:00 /usr/libexec/gsd-wacom
shaun       6075  0.0  0.2 318248  2308 ?        Ssl  10:05   0:00 /usr/libexec/gsd-wwan
shaun       6079  0.0  0.6 348996  6156 ?        Ssl  10:05   0:00 /usr/libexec/gsd-xsettings
shaun       6093  0.0  0.3 789816  3780 ?        Sl   10:05   0:00 /usr/libexec/evolution-data-server/evolution-alarm-notify
shaun       6113  0.0  0.1 231800  1352 ?        Sl   10:05   0:00 /usr/libexec/gsd-disk-utility-notify
shaun       6145  0.0  0.2 342180  2420 ?        Sl   10:05   0:00 /usr/libexec/gsd-printer
shaun       6191  0.0  0.1 166548  1668 ?        Sl   10:05   0:00 /usr/libexec/ibus-engine-simple
shaun       6254  0.0  0.3 162296  3456 ?        Ssl  10:06   0:00 /usr/libexec/gvfsd-metadata
shaun       6257  0.0  1.4 496532 14404 ?        Sl   10:06   0:00 update-notifier
shaun       6293  0.3  3.4 821124 34036 ?        Rsl  10:06   0:07 /usr/libexec/gnome-terminal-server
shaun       6311  0.0  0.3  10612  3196 pts/0    Ss+  10:06   0:00 bash
shaun       6456  0.2  3.2 991712 32448 ?        Sl   10:07   0:05 /usr/bin/nautilus --gapplication-service
shaun       6498  0.0  0.3 317740  3312 ?        Sl   10:09   0:00 /usr/libexec/gvfsd-network --spawner :1.3 /org/gtk/gvfs/exec_
shaun       6504  0.0  0.6 450336  6048 ?        Sl   10:09   0:00 /usr/libexec/gvfsd-smb-browse --spawner :1.3 /org/gtk/gvfs/ex
root        7287  0.0  0.0      0     0 ?        I<   10:10   0:00 [xfsalloc]
root        7288  0.0  0.0      0     0 ?        I<   10:10   0:00 [xfs_mru_cache]
root        7292  0.0  0.0      0     0 ?        S    10:10   0:00 [jfsIO]
root        7293  0.0  0.0      0     0 ?        S    10:10   0:00 [jfsCommit]
root        7294  0.0  0.0      0     0 ?        S    10:10   0:00 [jfsSync]
shaun       8060  0.2  4.9 820392 49232 ?        Sl   10:11   0:03 /usr/bin/gedit --gapplication-service
shaun       8362  0.0  0.4  10612  4756 pts/1    Ss   10:13   0:00 bash
root        8435  0.0  0.0      0     0 ?        I    10:30   0:00 [kworker/u2:1-events_unbound]
shaun       8465  0.0  0.0   2496    72 pts/0    S    10:36   0:00 ./Zombie
shaun       8466  0.0  0.3  11496  3172 pts/1    R+   10:36   0:00 ps aux
shaun@shaun-VirtualBox:~$
```

Zombie process with PID=8465

```
Zombie.c:16:2: warning: incompatible implicit declaration of built-in function
'exit'
   16 |  exit (0);
      |  ^~~~
Zombie.c:16:2: note: include '<stdlib.h>' or provide a declaration of 'exit'
shaun@shaun-VirtualBox:~/OS$ ./Zombie
chirld process ID 8465
shaun@shaun-VirtualBox:~/OS$ I am the child process before sleep.
I am the child process after sleep.
```

Process Creation & Termination

How to kill a zombie process

- Solution_1: a child process will automatically send a SIGCHLD signal to the parent process after its termination. Once receiving the signal, the parent process could execute the wait() system call and clean up its zombie children.

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

- Solution_2: kill 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.

Process Creation & Termination

More about Unix/Linux Process's Attributes

- PID
  - ✓ each process has a non-negative, system-unique # as its PID
  - ✓ some fixed process IDs, 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 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 calling process.

Process Creation & Termination

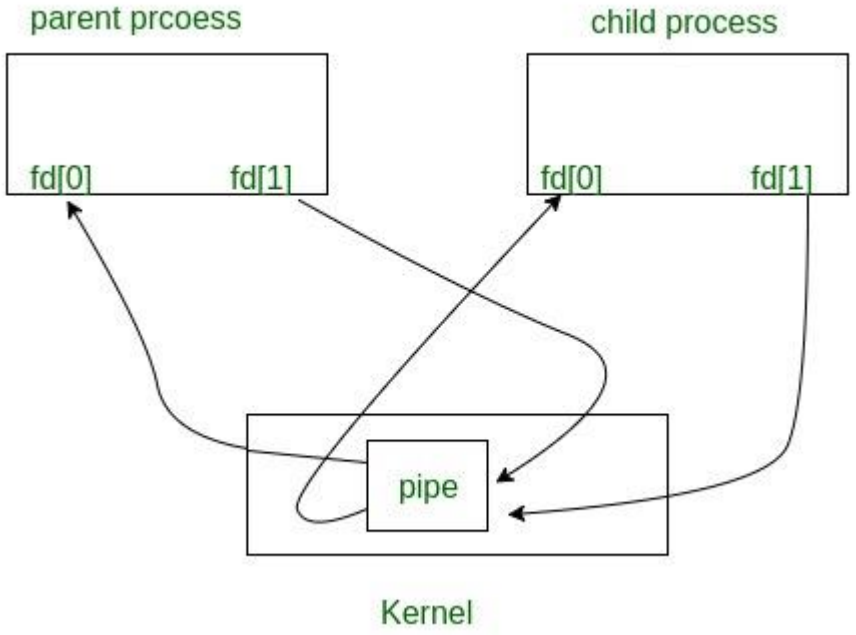
❑ More about Unix/Linux Process's Attributes

- ➢ Session ID (SID)
  - ✓ session = a set of process groups sharing a control terminal.
  - ✓ when a process is created, it becomes a member of its parent's session.
  - ✓ 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().
  - ✓ start a new session by calling setsid(void) (return the new SID)

❑ More about Unix/Linux Process's Attributes

➢ pipe() system call

- ✓ 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.
- ✓ If a process attempts to read from an empty pipe, the process is block.



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
```