Processes and Fork

Processes

- In modern operating systems a ton of processes can be running in the background.
- To see the processes running in linux use:


```
(base) nawiebe@DESKTOP-1TUR0IR:~$ top
```

```
top - 08:50:37 up 5 days, 14:43,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   4 total,   1 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7.7 us,  2.8 sy,  0.0 ni, 89.3 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :  16305.4 total,   4715.4 free,  11366.0 used,    224.0 buff/cache
MiB Swap:  49152.0 total,  48449.7 free,    702.3 used.   4808.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0    9352    488    316 S   0.0   0.0   2:49.02 init
24050 root      20   0    9352    232    184 S   0.0   0.0   0:00.00 init
24051 nawiebe   20   0   18344   3948   3840 S   0.0   0.0   0:00.21 bash
24148 nawiebe   20   0   18924   2144   1528 R   0.0   0.0   0:00.03 top
```

What does this mean?

- Each task has a Process ID (PID) that uniquely identifies it.
- A task is sleeping if it is waiting for a signal from a child process.


```
top - 08:50:37 up 5 days, 14:43,  0 users,  load average: 0.52, 0.58, 0.59
Tasks:   4 total,   1 running,   3 sleeping,   0 stopped,   0 zombie
%Cpu(s):  7.7 us,  2.8 sy,  0.0 ni, 89.3 id,  0.0 wa,  0.2 hi,  0.0 si,  0.0 st
MiB Mem :  16305.4 total,   4715.4 free,  11366.0 used,    224.0 buff/cache
MiB Swap:  49152.0 total,  48449.7 free,    702.3 used.   4808.8 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
    1 root      20   0    9352    488    316 S   0.0   0.0   2:49.02 init
24050 root      20   0    9352    232    184 S   0.0   0.0   0:00.00 init
24051 nawiebe   20   0   18344   3948   3840 S   0.0   0.0   0:00.21 bash
24148 nawiebe   20   0   18924   2144   1528 R   0.0   0.0   0:00.03 top
```

Calling Linux System

- Functionality in `#include<unistd.h>`
- Important types in: `#include<sys/types.h>`

- Important Commands:
  - getpid -> returns a PID for the calling process as type pid_t (signed int)
  - execlp -> swaps process with a program on the linux PATH
  - fork -> generates a new process that has an independent copy of the stack.

Example: Get PID

PID.c

```c
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int main()
{
  pid_t PID=getpid();
  printf("PID is %d\n",PID);

}
```

Makes a system call that returns the PID of the calling process.


```
(base) nawiebe@DESKTOP-ITUK0IR:~/demos/fork$ ./PID
PID is 24239
```

Example: Running GetPID externally with execlp

- execlp runs the program PID replacing existing process immediately.

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<errno.h>
int main()
{
  char *progName = "./PID";
  char *arg = "";
  printf("Calling Process Has ID %d\n",getpid());
  fflush(stdout); //Flushes standard output immediately to screen.
  int err=execlp(progName,arg,NULL);
  printf("Call Complete\n");
}
```

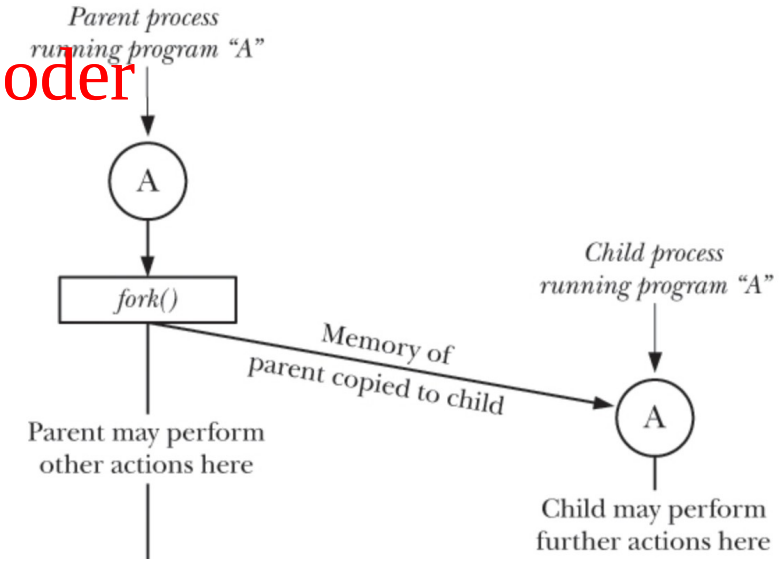

- What do you think prints?

```
(base) nawiebe@DESKTOP-ITUK0IR:~/demos/fork$ ./execTest
Calling Process Has ID 24274
PID is 24274
```

Fork this Process

- exec runs a program but doesn't create a new process.
- This isn't ideal in multi-threaded environments.
- Often what you want to do is start a new process, fork does this.
- The code starts for the child at exactly the point where fork was called
- The only difference in memory is in the return of fork:

  0 = child
  PID of child = parent
  negative = fail



The first forking example

```c
#include<unistd.h>
#include<stdio.h>
int main()
{
  printf("My Process ID is %d\n",getpid());
  int x=fork();
  if (x==0){
    printf("I'm a child and my PID is %d\n",getpid());
  }
  else if(x>0){
    printf("I'm a parent and my PID is %d\n",getpid());
    printf("My child's PID is %d\n",x);
  }
  else
    {
      printf("Oh no something bad happened\n");
    }
}
```



```
(base) nawiebe@DESKTOP-ITUK0IR:~/demos/fork$ ./fork
My Process ID is 24309
I'm a parent and my PID is 24309
I'm a child and my PID is 24310
My child's PID is 24310
```

Race Conditions

```
(base) nawiebe@DESKTOP-ITUK0IR:~/demos/fork$ ./fork
My Process ID is 24309
I'm a parent and my PID is 24309
I'm a child and my PID is 24310
My child's PID is 24310
```

- The order the forked processes are executed is not deterministic.
- Most times the parent process will be given priority over the child but that isn't guaranteed.
- In this example, the result is benign, but what if they both wrote to a file?
- This situation is known as a race condition and we'll worry about dealing with it next lecture.