# Operating systems 1, Lecture 7

Sanda-Maria AVRAM, Ph. D.

Leave a feedback

1

## Unix processes

Process = a running program that uses a system resource set (memory, processor, disk, network interface, etc.)

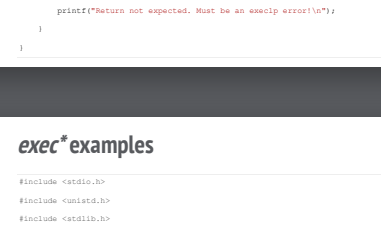
2

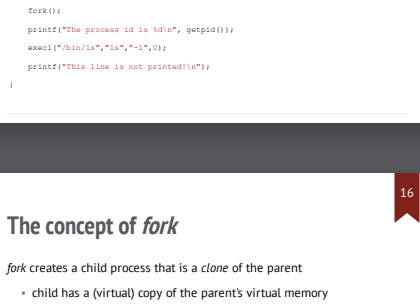
## The image of a process in memory

- **User context** – the portion of the address space accessible during execution by that process in user mode;
- **Kernel context** – maintained by the kernel and accessible only through specific system calls.

3

## The image of a process in memory




5

## Entry in the process table

- process *state*
- pointer to the *User area* and the Process *Memory Regions Table*
- *process size* in memory
- PID, PPID
- UID, EUID, GID, EGID
- process *priority* (number between 1 and 39 used for scheduling)
- *signals* sent to the process
- *time statistics* (e.g., using the processor)
- process *memory status* (if the process image is in the main memory or in the swap memory)
- *pointer to the next process* in the process queue that is in the READY state
- *event descriptors* that occurred while the process was in SLEEP state

6

## User area

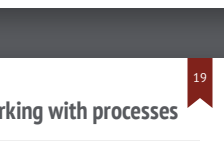
- *pointer to the entry in the process table* that corresponds to this user area
- UID, EUID, GID, EGID - used to determine process access rights
- *timers* - time spent in user mode and in kernel mode
- *vector of actions to handle signals*
- process *control terminal* - the one from which it was launched
- *return values* and possible errors after making system calls
- *the current directory* and *the root directory*
- *environmental variables*
- possible kernel *restrictions* imposed on the process (e.g., process size in memory)
- *file descriptor table* (data about all open process files)
- **umask** - access rights mask for newly created files by this process

7

## Creating a Process. System call *fork*

```
#include <unistd.h>

pid_t fork();
```



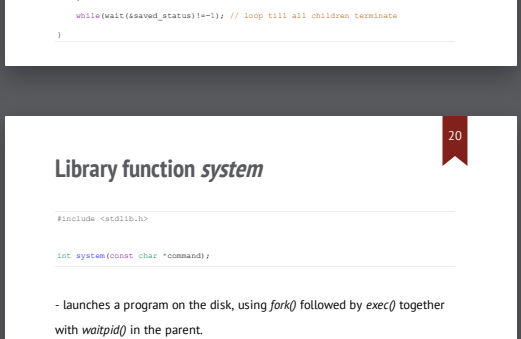

Return:

-1 - if the operation could not be performed (*err*);

0 - in child code;

pid - in parent code, where **pid** is the process identifier of the newly created child.

man fork




9

## Using *fork* - suggestions - 1

```
...
if( ( pid=fork() ) < 0) {
        perror("Error");
        exit(1);
}
if(pid==0) {
        /* child code */
        ...
        exit(0)
}
/* parent code */
...
wait(&status) ;
```

10

## Using *fork* - suggestions - 2

```
pid=fork();
switch(pid) {
    case -1:
        fprintf(stderr,"ERROR: %s\n", sys_errlist[errno]);
        exit(1);
        break;
    case 0:
        /* child code */
        break;
    default:
        /* parent code */
        break;
}
```

11

## Using *fork* - suggestions - 3

```
if (fork() == 0) {
        /* child code */
}
else {
        /* parent code */
}
```

12

## Execution of an external program. System calls *exec**

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

man 3 exec

13

## *execlp()* example

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
main() {
    pid_t pid;
    if((pid=fork())==-1) perror("fork error");
    else if(pid==0) {
        execlp("newShell", "newShell", NULL);
        printf("Return not expected. Must be an execlp error!\n");
    }
}
```

14

## *execl()* example

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

main() {
    fork();
    printf("The process id is %d\n", getpid());
    execl("/bin/ls","ls","-l",0);
    printf("This line is not printed!\n");
}
```

15

## *exec** examples

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

int main() {
    // execl("/bin/ls", "/bin/ls", "-l", NULL);
    // execlp("ls", "ls", "-l", NULL);
    // execl("/bin/ls","/bin/ls","-l","p1.c","execl.c", "fork1.c", "xx", NULL);
    execl("/bin/ls","/bin/ls","-l","*.c", NULL);
}
```

16

## The concept of *fork*

- *fork* creates a child process that is a *clone* of the parent
  - child has a (virtual) copy of the parent's virtual memory
  - child is running the same program as the parent
  - child *inherits* open file descriptors from the parent
  - child begins life with the same register values as parent
- the child process may execute a different program in its context with a separate *exec* system call

17

## System calls *wait()* and *waitpid()*

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Return: -1 - error

>0 - the process ID of the terminated child

man 2 wait

18

## *wait()* system call. Example

```
int main() {
    int i, saved_status;
    for(i=0;i<3;i++) {
        if(fork()==0) {
            // child process
            exit();
        }
    }
    while(wait(&saved_status)!=-1); // loop till all children terminate
}
```

19

## Other functions for working with processes

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

pid_t getpid(); // - returns the PID of the current process
pid_t getppid(); // - returns the PID to the parent of the current process
uid_t getuid(); // - returns the user ID that launched the current process
gid_t getgid(); // - returns the user group identifier that launched the current process
```

man getpid; man getuid; man getgid

20

## Library function *system*

```
#include <stdlib.h>

int system(const char *command);
```

- launches a program on the disk, using *fork()* followed by *exec()* together with *waitpid()* in the parent.

man system

21

## *vfork()* system call

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

pid_t vfork(void);
```

- creates a new process, just like *fork()* but does not fully copy the parent's address space. It is used in conjunction with *exec()*, and has the advantage that it does not consume the time required for copy operations that would be useless anyway if *exec()* is called immediately (however, the process will be overwritten with the program taken from the disk).

man vfork

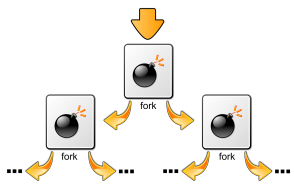
22

## Exercise

**Explain the effect of the following code sequence:**

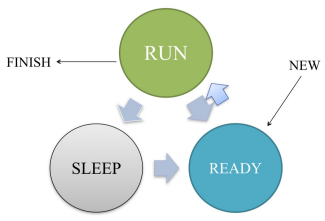
```
int i;

for (i = 1; i <= 10; i++)    fork();
```

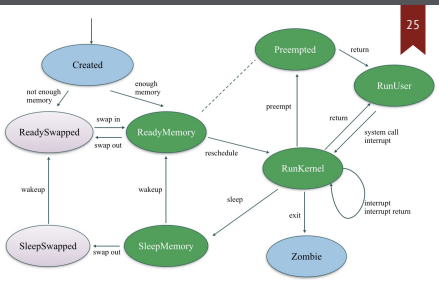





## Process states








## Process states

Created – the process is created through a *fork* system call.

ReadyMemory – the process is in queue in RAM and is ready for execution;
\
the kernel decides when it enters into execution.

ReadySwapped – the process is ready to be executed, but swapped due to memory shortage;
\
to enter the execution state must first enter the **ReadyMemory** state.

RunKernel – process instructions are executed in the kernel mode (eg system calls, interrupt handlers).

RunUser – program instructions are executed.



## Process states (cont.)

Preemted – almost identical to **ReadyMemory**. After running a system call, the process must be passed to **RunUser** or **ReadyMemory**. If there is a process with a higher priority in **ReadyMemory**, the current process is passed to **Preemted** and the process with the higher priority goes to **RunUser**, otherwise the current process passes to **RunUser**. A process in **Preemted** moves to **RunUser** when it has the higest priority.

SleepMemory – the process is in memory but can not be executed until there is an event that will get it out of this state (e.g., waits for a resource).

SleepSwapped – the process is swapped on the disk due to memory shortage, it can not be executed until an event occurs that will get it out of this state.

Zombie – the process no longer exists, has ended, but has not been removed from the process table.



# The End