

Processes in C

At boot time – after diagnostics of memory, disk, peripheral devices, two processes are created:

- PID 0 – swapper (also named sched or pager on different systems)
 - Controlled by kernel, handles mapping physical memory to the virtual memory on the disk.
- PID 1 – init (systemd)
- All other processes are created by `fork()`;

`fork()` system call

- was long used in concurrent programming terminology to denote a time when a new branch of some software system begins executing concurrently with other branches.
- The only way to create new processes in Unix/Linux systems.
- Creates new process and allows it to execute the same code as the creating process.
- Two processes involved:
 - Parent process, which creates a new process.
 - Child process.
- Both have equal rights to the scheduler.
- Child inherits all open descriptors, and all open files from the parent.
- This means same file can be used for communication between parent and child.

```
int main() {  
    puts("Begin fork");  
    fork();  
    puts("End fork");  
    return 0;  
}
```

- Child starts executing the same code after the `fork()` system call. The new process created by `fork()` is a copy of the current process except for the returned value.

What is the output in the code above?

Will output always be in the same sequence?

- `Fork()` returns these values:
 - Child process gets a value 0
 - Parent gets a pid of the child
 - -1 indicates an error (typical to C)

Example1. What is the output of this code? How many processes will be created?

```
int main(){
    fork();
    fork();
    printf("I am a process\n");
    return 0;
}
```

Example2. What is the output here? How many processes got created?

```
int main(){
    fork();
    fork();
    fork();
    printf("I am a process\n");
    return 0;
}
```

- Parent process and child process are running the same program, but it does not mean they are identical. OS allocates different data and states for these two processes, and the control flow of these processes can be different.

Example 3.

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}

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

Child has x = 2;
Parent has x = 0;
or:

Parent has x = 0;
Child has x = 2;

Fork Bomb

- Fork Bomb is a program which will make the system run out of memory. It will harm the system by consuming memory. The fork bomb is a form of denial-of-service (DoS) attack against a Linux-based system.
- Once a successful fork bomb has been activated in a system it may not be possible to resume normal operation without rebooting the system. The only solution to a fork bomb is to destroy all running instances of it.

```
int main()
{
    while(1)
        fork();
    return 0;
}
```

- Fork bomb consumes CPU time in the process of forking and saturates the operating system's process table.
- A basic implementation of a fork bomb is an infinite loop that repeatedly forks processes.
- The assumption is that the number of processes which can simultaneously execute is limited. When the limit is reached, the system will crash and may need to be restarted.

How to prevent a fork bomb

- Avoid the use of fork() in loop statements.
- Limit the number of user processes in /etc/security/limits.conf:

```
your_user_name hard nproc 10
```

This way user can only run 10 processes. By default the number is unlimited.

Fork bomb in bash

If you run this command, you may crash your system and may have to force-rebooting it. Also, it doesn't need root to run. If you using terminal then bash script for fork() bomb script looks like this.

```
:(){ :|: & }::
```

Syntax:

1. `:()` means you are defining a function called `:`
2. `{:|: &}` means run the function `:` and send its output to the `:` function again and run that in the background.
3. `;` command separator
4. `:` run the function now

Essentially you are creating a function that calls itself twice every call and doesn't have any way to terminate itself. It will keep doubling up until you run out of system resources.

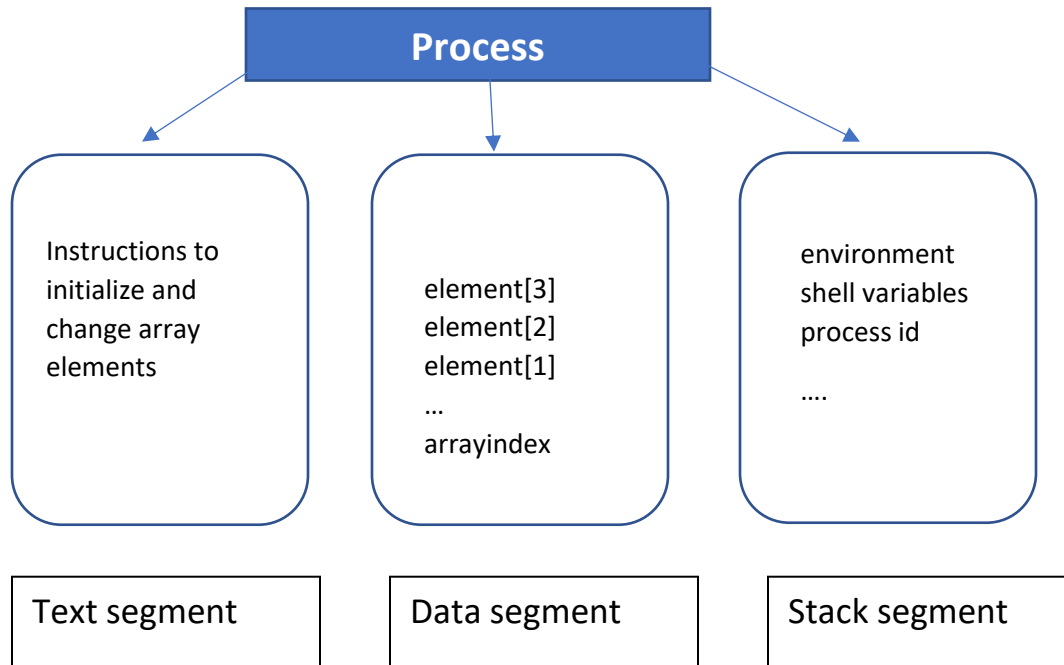
Process table

- Every process has an entry in a kernel DS called *process table*.
- An array of structures with a fixed size depending on the computer hardware.
- There is a max of how many processes can be active on the system at one time.
☹
- It can be set by the root in the file `/etc/security/limits.conf`. Read the instructions in that file before making modifications.

Segments associated with a process

- Three segments, each has its own portion of memory.
 - Text segment (code segment) – stores all executable code (that performs operation on data) for the process (machine translation of the code, library linkage, etc.)
 - Data segment – will store variable for loop control (in our example)
 - Stack segment (also called system data segment) – stores environment of the process
- Parent and child share the same text and data segment
- Stack segments are similar, but not identical, since processes are different and have different process ids.

- After `exec()` system call segments are different, since `exec()` reinitializes the text, data, and stack segments.



Initializing Process with `exec()` system call

- `exec()` overwrites the text and data segments of process with those of some other executable file.
- without explicit call to `exec()` the data segment is only modified by process itself.
- with call to `exec()` normal behavior of data segment will be modified by the process that owns the segment after data segment was init'd by the `exec` that created it.
- There are 5 `exec()` system calls.

```

int main (int argc, char * argv[]){
    if (fork() == 0) /* in child process */
        execl ("./count", "count", argv[1], 0);
    else /* in parent process */
        printf ("In parent process \n");
}

```

```
wait (NULL);  
printf ("In parent, child complete\n");  
}
```

- Parent is suspended while all processes are finished executing (ret value of -1).

Signals

- Conceptually simple way of communicating between processes.
- One process sends, another receives.
- Receiving process acts on the signal by executing signal handler, or signal catcher.
- Signal catcher is a function within the process written to have specific action.
- Signal is usually sent by the kernel, or by another process.
- Interrupts to normal processing.
- Cannot be queued.

signal() system call

- signal (signal_number, sig_action)
- signal number is an int denoting an allowable signal.
- Sig_action:
 - SIG_DFL – used to set default action for the signal.
 - SIG_IGN – ignore signal (cannot ignore the SIGKILL)
 - Pointer to a function used as a signal handler.