

## CSI 400: Lecture 07

# Static and Dynamic Entities Compared Process Creation

```
main(){  
    int C = 38;  
    while(C--){cout << "Hi" << endl;}  
}
```

This program has **ONE static** instance of a certain “output statement”. This static statement is merely a mark on a screen.

When this program is compiled, linked, loaded into a process’s virtual memory and that process runs, the computer executes **38 dynamic instances** of the same output operation (assuming the program runs to completion without errors.)

Similarly a **system call** in the static sense is a kind of operation or subroutine whose use can be coded in programs. System calls are coded by “syscall” type machine instructions (such as `int 0x80` on `x386/Linux`) that cause interrupts.

A system call in the dynamic sense is the action of a process executing a system call instruction at a particular time.

The `Unix fork()` system call creates a new process. Each new process is created by the OS (kernel) as the effect of a `fork()` executed by some already existing process, except for the first process or for forks that fail.

Typically, a process is created by a system call for that purpose.

- Under Unix, `fork()` does it. (Linux has the more general `clone()`.)
- A process created by a system call must have a **parent**: *the process that called that system call!* Parenthood defines the **tree of processes**.
- How did the first process get created? *Special kernel code!*
- Unix innovation (late 70's): `fork()` initializes the new process's virtual memory with a *COPY* of the parent process's virtual memory.
- Other variations:
  - Executable file is an argument to the process creator system call.
  - Arguments to the process creator call specify *which resources are shared* (e.g., `Linux clone()`): one resource is *virtual memory!*, so `clone()` can create a new thread.)Alternative: Parent and child separately release some resources and request others, after the child is created.
- Parent waits until child runs and exits (`Unix system()` function), parent blocks; **OR**
- Parent and child execute concurrently.

How do ordinary people get processes created for them?

1. When you log in, the “log in system” creates a process that runs an interactive shell program.

Interactive shells can be text terminal (Albany’s Unix-based CS major courses, MS-DOS “DOS shell”) or visual (MS Windows, Macintosh, Unix/Linux desktops, etc.)

2. The shell scans, checks, gathers arguments (e.g. command line args.), opens named files for redirection, etc., and then,
3. The shell calls the system call(s) to create a new process to run the program somehow specified by you.

4. The shell might **block** until your program’s process finishes (or it receives a signal, typically caused by special characters like `control-c` or `control-z`,
  - Typically, your process interacts with you via the terminal while the shell is blocked (your process is running in the **foreground**).

5. or, the shell continues to execute *concurrently* with your process.
  - Under Unix shells, you can make this happen by commanding the job run **in the background**, by ending the command with `&`, e.g.,

```
$ emacs myproject.cpp &      You can give new commands now,  
$ g++ myproject.cpp  
$ a.out
```

Interaction with your program.... **Your PROCESS, actually!**

```
$      Edit out bugs, save myproject.cpp file  
$ g++ myproject.cpp  
$ a.out
```

Using a shell to make the computer run several of your programs concurrently:

```
$ g++ -o part1 part1.cpp  
$ part1 &  
$ g++ -o part2 part2.cpp  
$ part2 &  
$ part1 &
```

```
#include <iostream>
#include <sys/types.h>
#include <unistd.h>
using namespace std;
main(int argc, char *argv[])
{
    cout << "Process with pid=" << getpid() << " ";
    cout << "running " << argv[0] << " ";
    if(argc > 1)
        cout << "on argument " << argv[1] << " ";
    cout << "is done!" << endl;
    return 88;
}
```

```
[seth@knowledge L07]$ part1 HAPPY & part1 DAYS & part1 ARE & part2  
HERE & part2 "so lets have some fun"
```

```
[3] 3729
```

```
[4] 3730
```

```
Process with pid=3729 running part1 on argument HAPPY is done!
```

```
[5] 3731
```

```
[6] 3732
```

```
Process with pid=3731 running part1 on argument ARE is done!
```

```
Process with pid=3732 running part2 on argument HERE is done!
```

```
Process with pid=3733 running part2 on argument so lets have some  
fun is done!
```

```
[3] Exit 88
```

```
part1 HAPPY
```

```
[5] - Exit 88
```

```
part1 ARE
```

```
[6] + Exit 88
```

```
part2 HERE
```

```
[seth@knowledge L07]$ Process with pid=3730 running part1 on argum  
ent DAYS is done!
```

```
[4] + Exit 88
```

```
part1 DAYS
```

```
[seth@knowledge L07]$
```

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    while(1) {
        cout << argv[0];
        if(argc > 1) cout << " on " << argv[1];
        cout << endl;
    }
    return 0;
}
```

University at Albany Computer Science Dept.

## Unix fork and exec **READ**: Haviland Chapter 5.

```
#include <stdio.h>           //Example from AOS Fig. 4.8
#include <sys/types.h>
#include <unistd.h>
void main (int argc, char *argv[])
{
    pid_t pid;
    /* fork another process */
    pid = fork(); /* fork() never has an argument! */
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execlp("/bin/ls", "ls", NULL);
        /* NEVER RETURNS!!! */
    }
    else { /* parent process */
        wait(NULL);
        printf("Child Complete");
        exit(0);
        /* NEVER RETURNS */
    }
}
```