Fork() System Call and Processes multi-taskings

Programs and Processes

- Program: Executable binary (code and static data)
- Process: A program loaded into memory
  - Program (executable binary with data and text section)
  - Execution state (heap, stack, and processor registers)

Program

```
int foo() {
    return 0;
}

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

Process

```
int foo() {
    return 0;
}

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

Heap

Stack

Registers

fork()

- A system call that creates a new process identical to the calling one
  - Makes a copy of text, data, stack, and heap
  - Starts executing on that new copy
- Uses of fork()
  - To create a parallel program with multiple processes (E.g. Web server forks a process on each HTTP request)
  - To launch a new program using exec() family of functions (E.g. Linux shell forks an ‘ls’ process)

fork() example

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  int seq=0;
  if(fork()==0)
  {
    printf("Child! Seq=%d\n", ++seq);
  }
  else
  {
    printf("Parent! Seq=%d\n", ++seq);
  }
  printf("Both! Seq=%d\n", ++seq);
  return 0;
}
```

```
>> ./a.out
Parent! Seq=1
Both! Seq=2
Child! Seq=1
Both! Seq=2
```

- Differentiate child and parent using return value of fork()
  - "Child" process return value is 0
  - "Parent" process gets child's process id number
- **Copies** execution state (not shares)
  - Child copies stack variable seq onto its own stack
  - Child / parent has own copy of seq

How do Parent / Child Run in Parallel?

- Obvious answer: by running on different processors on a multiprocessor system
- What if it's a uniprocessor system?
- What if other processors are already busy?
- Answer: by context switching (running each process in short time slots)

Spawning a New Program

- Combination of fork() and exec(…)
  - fork(): Clone current process
  - exec(…): copy new program on top of current process
- Exec(...) family of functions
  - execl, execlp, execle, execv, execve, execvp
  - User space wrappers for the **execve** system call
  - Also called the “program loader”
    - Loads in text and data sections of a binary executable
    - Links in any shared objects and perform relocations
    - Sets up stack and starts executing
  - What Linux shell calls when launching a program

execvp() example

```
#include <stdio.h>
#include <unistd.h>
int main()
{
  if(fork()==0)
  {
    char *args[3] ={"ls", "-al", NULL};
    execvp(args[0], args);
    // DOESNOTGETHERE
  }
  else
  {
    printf("Parent!\n");
  }
  printf("Only parent!\n");
  return 0;
}
```

```
>> ./a.out
Parent!
Only parent!
drwx------ 4 wahn UNKNOWN1     4096 Oct 21 08:13 .
drwxr-xr-x 10 wahn UNKNOWN1    2048 Oct 21 08:13 ..
-rwxr-xr-x 1 wahn UNKNOWN1     6743 Oct 21 08:12 a.out
```

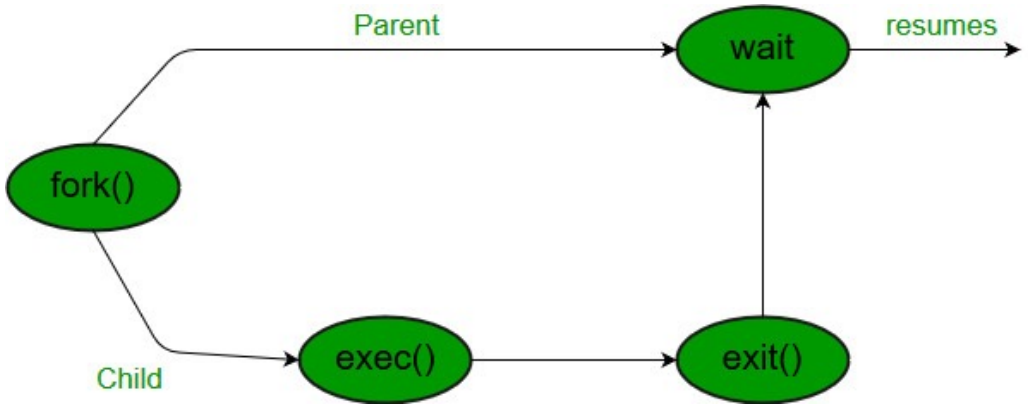
- execvp never returns since memory is overwritten using another program image

Wait() function to synchronize parent -> child terminaison

A call to wait() blocks the calling process until one of its child processes exits.
After child process terminates, parent ***continues*** its execution after wait system call instruction.

Child process may terminate due to any of these:

- It calls exit()
- It receives a terminating signal (from the OS or another process)
- It returns (an int) from main



Syntax in c language:
#include #include // take one argument status and returns // a process ID of dead children.
pid_t wait(int *stat_loc);