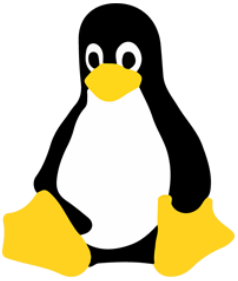


CS416: Systems Programming

Process Related Systems Call

(System, Fork, EXEC)

Most of the slides in this lecture are either from or adapted from the slides provided by Dr. Ahmad Barghash

Creating processes

- **Method (1) using *system***
  - The system function in the standard C library provides an easy way to execute a command from within a program, much as if the command had been typed into a shell.

```
#include <stdlib.h>
int main ()
{
        int return_value;
        return_value = system (“ls -l /”);
        return return_value;
}
```

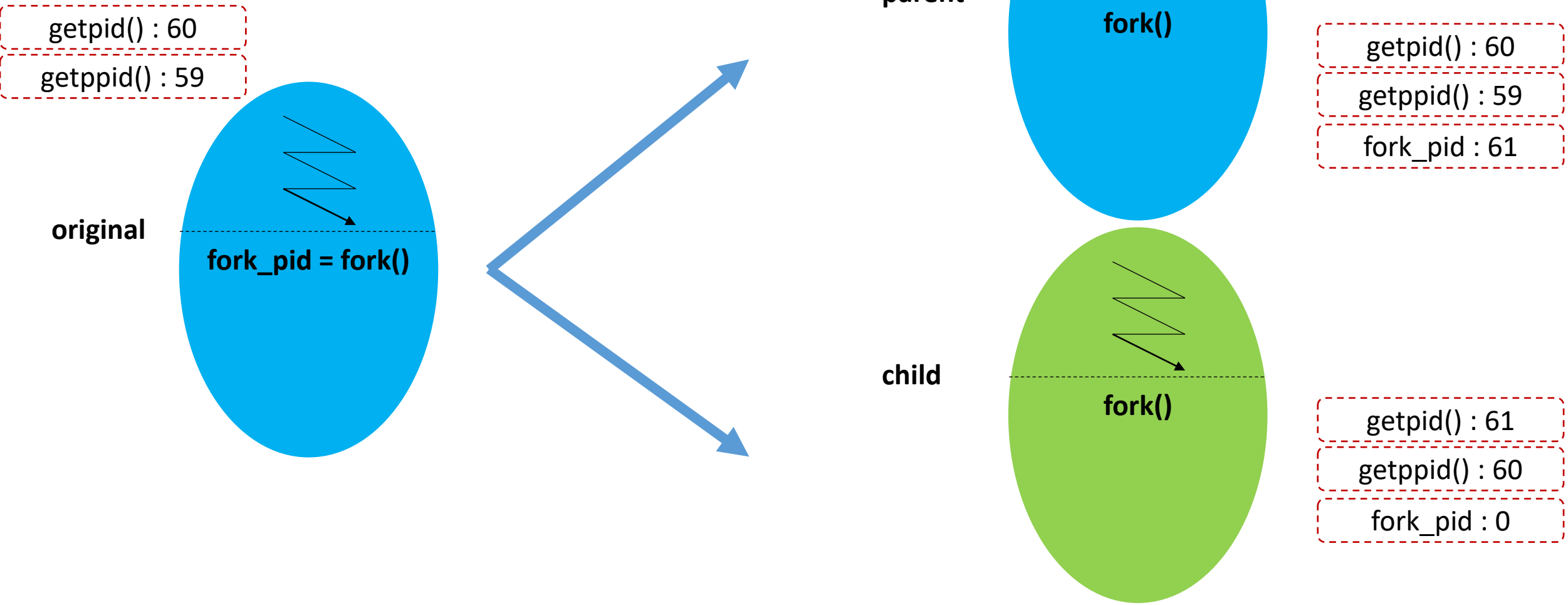
Creating processes

- **Method (2) using *fork* and *exec***
  - Linux provides one function, **fork**, that makes a child process that is an exact copy of its parent process
  - Linux provides another set of functions, the **exec** family, that causes a particular process to cease being an instance of one program and to instead become an instance of another program
  - To **spawn** a new process, you first use **fork** to make a copy of the current process. Then you use **exec** to transform one of these processes into an instance of the program you want to spawn

int fork(void)

- **Description:** create a child process
- **Returns:** process ID of the new process

int fork(void)



fork() - Example

```
./main
the main program process ID is 117
the main program parent process ID is 6
**************Parent Process************
process ID is 117
parent process ID is 6
the child's process ID is 118
**************Child Process************
process ID is 118
parent process ID is 117
logical ID of the process based on the fork function is 0
```

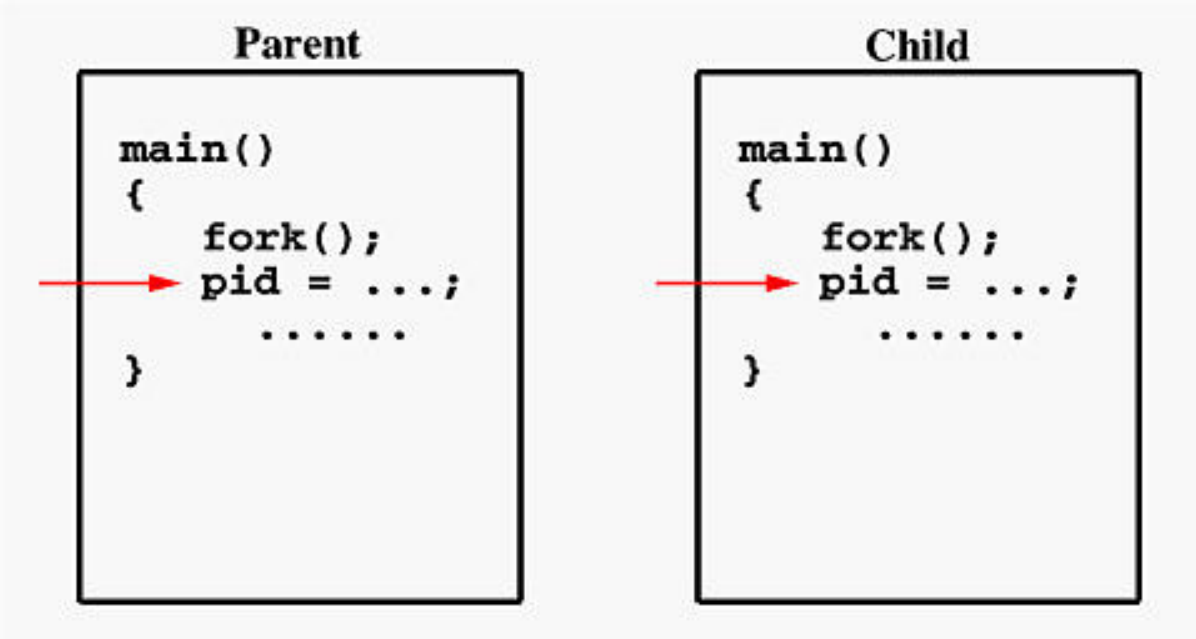
```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

int main (void)
{
  pid_t fork_pid;
  printf ("the main program process ID is %d\n", (int) getpid ());
  printf ("the main program parent process ID is %d\n", (int) getppid ());
  fork_pid = fork ();

  if (fork_pid != 0)
  {
    printf ("**************Parent Process************\n");
    printf ("process ID is %d\n", (int) getpid ());
    printf ("parent process ID is %d\n", (int) getppid ());
    printf ("the child's process ID is %d\n", (int) fork_pid);
    sleep(10);
  }
  else{
    wait(NULL);
    printf ("**************Child Process************\n");
    printf ("process ID is %d\n", (int) getpid ());
    printf ("parent process ID is %d\n", (int) getppid ());
    printf ("logical ID of the process based on the fork function is %d\n", (int) fork_pid);
  }
  return 0;
}
```

Contd.

- System call **fork()** is used to create processes.
- It takes no arguments and returns a process ID.
- The purpose of **fork()** is to create a ***new*** process, which becomes the *child* process of the caller.
- After a new child process is created, ***both*** processes will execute the next instruction following the ***fork()*** system call.



Distinguish parent from child processes

Simply check the returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent.

The returned process ID is of type **pid_t** defined in **sys/types.h**. Normally, the process ID is an integer. Moreover, a process can use function **getpid()** to retrieve the process ID assigned to this process.

Another example

```
#include  <stdio.h>
#include  <sys/types.h>
#include <unistd.h>
#define   MAX_COUNT  3
void  ChildProcess(void);    /* child process prototype  */
void  ParentProcess(void);   /* parent process prototype */
```

```
int  main(void)
{
    pid_t  pid;
    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
    return 0;
}
void  ChildProcess(void)
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("   This line is from child, value = %d\n", i);
    printf("   *** Child process is done ***\n");
}
void  ParentProcess(void)
{
    int   i;
    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n", i);
    printf("*** Parent is done ***\n");
}
```

Previous Example - Sample Output

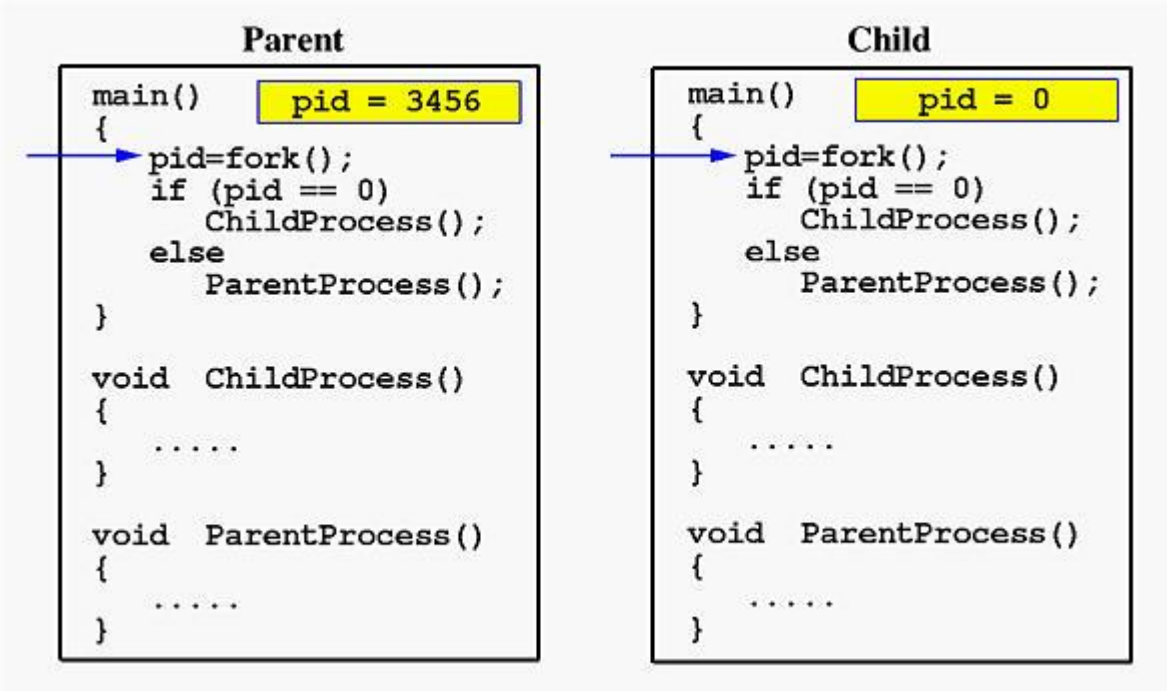
```
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
*** Parent is done ***
   This line is from child, value = 1
   This line is from child, value = 2
   This line is from child, value = 3
   *** Child process is done ***
```

```
This line is from parent, value = 1
   This line is from child, value = 1
   This line is from child, value = 2
   This line is from child, value = 3
   *** Child process is done ***
This line is from parent, value = 2
This line is from parent, value = 3
*** Parent is done ***
```

```
This line is from parent, value = 1
This line is from parent, value = 2
   This line is from child, value = 1
   This line is from child, value = 2
   This line is from child, value = 3
   *** Child process is done ***
This line is from parent, value = 3
*** Parent is done ***
```

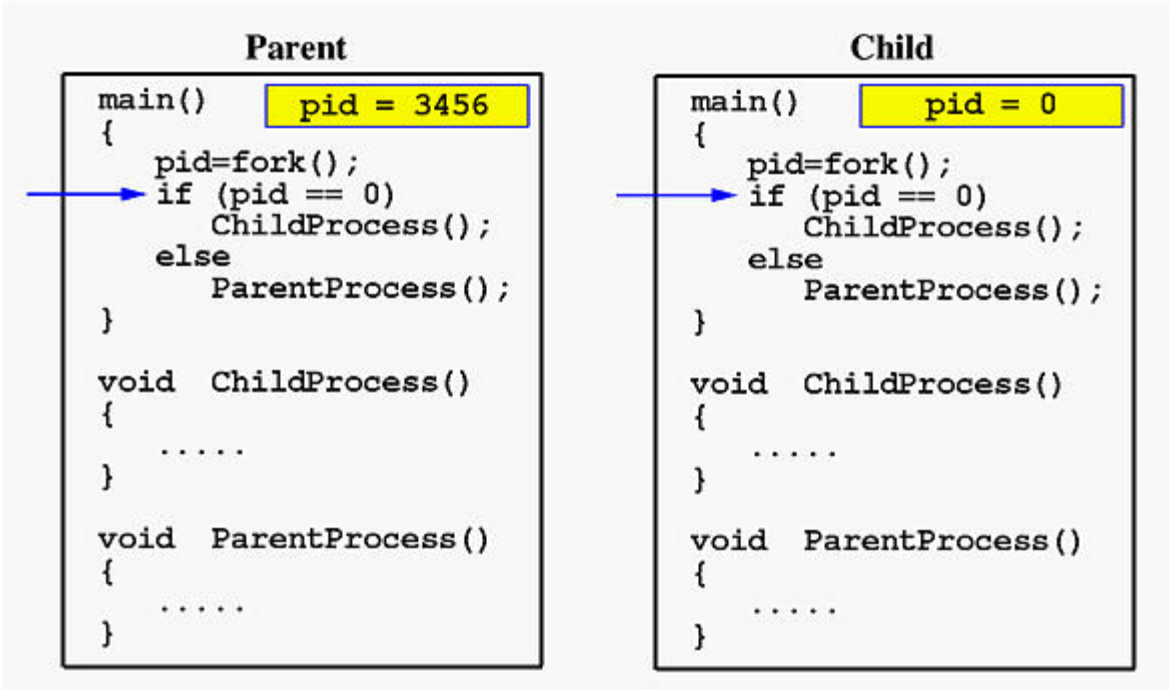
```
This line is from parent, value = 1
This line is from parent, value = 2
This line is from parent, value = 3
*** Parent is done ***
```

How does it work

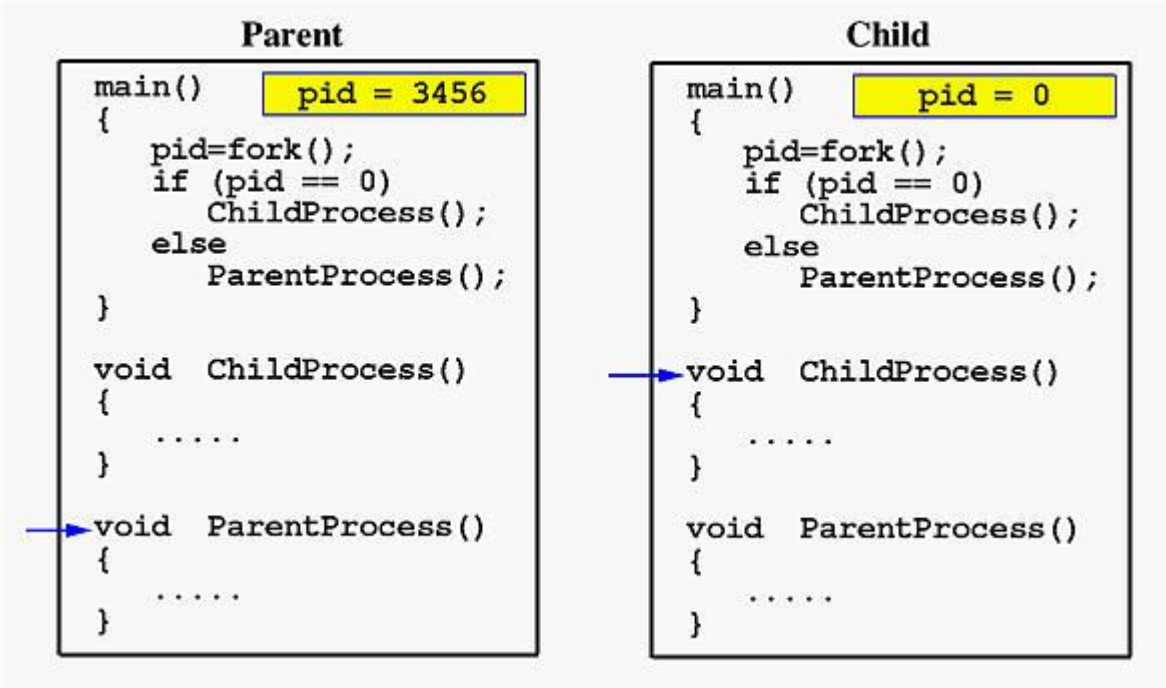


The execution depends on the fork returned values

Till this point there is no difference



How does it work



Based on the fork returned vale, one function is chosen. If both functions has an output command (like printf), the outputs in the child and parent processes might intersect

One extra example

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

int main(int argc, char **argv)
{
    printf("--beginning of program\n");

    int counter = 0;
    pid_t pid = fork();

    if (pid == 0)
    {
        // child process
        int i = 0;
        for (; i < 5; ++i)
        {
            printf("child process: counter=%d\n", ++counter);
        }
    }
    else if (pid > 0)
    {
        // parent process
        int j = 0;
        for (; j < 5; ++j)
        {
            printf("parent process: counter=%d\n", ++counter);
        }
    }
    else
    {
        // fork failed
        printf("fork() failed!\n");
        return 1;
    }
    printf("--end of program--\n");
    return 0;
}
```

```
--beginning of program
parent process: counter=1
parent process: counter=2
parent process: counter=3
parent process: counter=4
parent process: counter=5
--end of program--
child process: counter=1
child process: counter=2
child process: counter=3
child process: counter=4
child process: counter=5
--end of program--
```

Change the program running in the process

The exec ……. function replaces the program running in a process with another program.

The exec family has many similar functions that vary in the way they are called

1. Functions with the letter **p** in their names (execvp and execlp) accept a program name and search for it in the current execution path. No p in the name means you need to provide the full path of the program.
2. Functions with the letter **v** in their names (execv, execvp, and execve) accept an argument list for the program as a null terminated array of pointers to strings
3. Functions with the letter **e** in their names (execve and execle) accept an additional argument, an array of environment variables.

More about exec

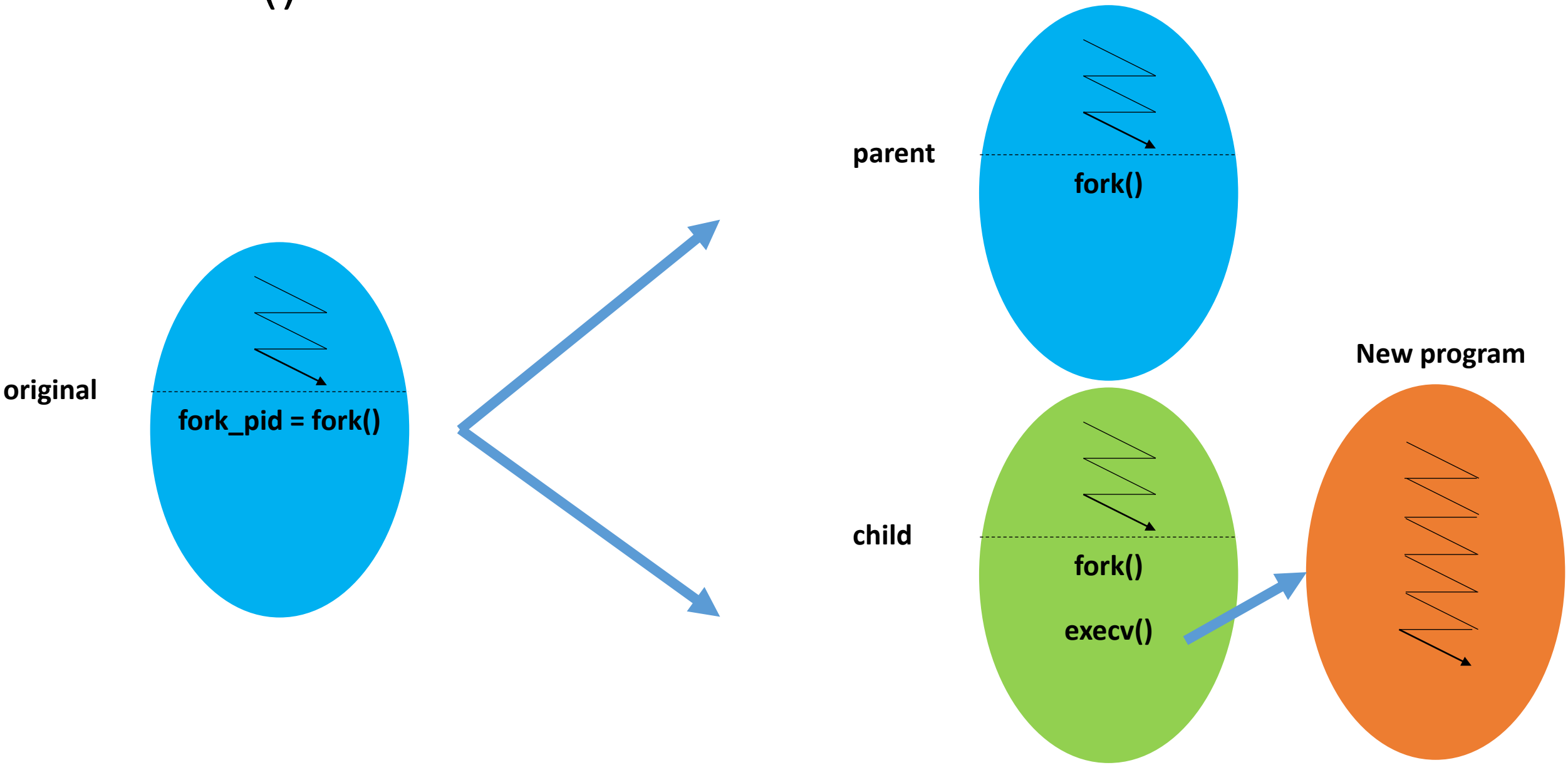
- Exec family never returns unless an error occurs
  - Use the function fprintf
  - Need the library stdlib.h

- Programmer should pass the name of the program as the first element of the argument list

```
int execv(char *progName, char *argv[])
```

- description: load program in process address space
- returns: -1 if failed

execv()


original
fork_pid = fork()
parent
fork()
child
fork()
execv()
New program

prog1.c

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

int main(void) {
  printf("Child Process: Hello -- %d\n", getpid());
  return 0;
}
```

>> gcc prog1.c -o childProg

>> ./childProg

```
~/WildMammothMicroscope$ gcc prog1.c -o childProg
~/WildMammothMicroscope$ ./childProg
Child Process: Hello -- 1014
~/WildMammothMicroscope$
```

prog2.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
int main()
{
  char* arg_list[]={"./childProg", NULL};
  pid_t child_pid;
  child_pid=fork();
  if(child_pid!=0)
  {
    printf("Parent Process: Hello -- %d\n", getpid());
    sleep(5); //sleep for 5 seconds
  }
  else{
    printf("Child Process: Before being replaced with another program -- %d\n", getpid());
    execvp(arg_list[0],arg_list);
    printf("Child Process: After being replaced with another program\n");
    abort();
  }
  printf("Done with the main prgram\n");
  return 0;
}
```

>> gcc prog2.c -o parentProg

>> ./parentProg

```
~/WildMammothMicroscope$ gcc prog2.c -o parentProg
~/WildMammothMicroscope$ ./parentProg
Parent Process: Hello -- 1528
Child Process: Before being replaced with another program -- 1529
Child Process: Hello -- 1529
Done with the main prgram
~/WildMammothMicroscope$
```

prog1.c

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

int main(void) {
  printf("Child Process: Hello -- %d\n", getpid());
  return 0;
}
```

>> gcc prog1.c -o childProg

>> ./childProg

```
~/WildMammothMicroscope$ gcc prog1.c -o childProg
~/WildMammothMicroscope$ ./childProg
Child Process: Hello -- 1014
~/WildMammothMicroscope$
```

prog2.c

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
int main()
{
  char* arg_list[]={"./NotExistingProg ", NULL};
  pid_t child_pid;
  child_pid=fork();
  if(child_pid!=0)
  {
    printf("Parent Process: Hello -- %d\n", getpid());
    sleep(5); //sleep for 5 seconds
  }
  else{
    printf("Child Process: Before being replaced with another program -- %d\n", getpid());
    execvp(arg_list[0],arg_list);
    printf("Child Process: After being replaced with another program\n");
    abort();
  }
  printf("Done with the main prgram\n");
  return 0;
}
```

>> gcc prog2.c -o parentProg

>> ./parentProg

```
~/WildMammothMicroscope$ gcc prog2.c -o parentProg
~/WildMammothMicroscope$ ./parentProg
Parent Process: Hello -- 2161
Child Process: Before being replaced with another program -- 2162
Child Process: After being replaced with another program
Done with the main prgram
~/WildMammothMicroscope$
```

fork and exec (Create spawn function for Linux)

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
void spawn(char* program, char** arg_list)
{
        pid_t child_pid;
        child_pid=fork();
        if(child_pid!=0)
        {
                printf("Run\n");
                printf("      Diff.\n");
                printf("                    Things\n");
                sleep(5); //sleep for 5 seconds
        }
        else{
                execvp(program,arg_list);
                fprintf(stderr,"an error occured in execvp\n");
                abort();
        }
}

int main()
{
        char* arg_list[]={"ls","-l","/", NULL};
        spawn(arg_list[0], arg_list);

        printf("Done with the main prgram\n");
        return 0;
}
```

- The Linux “ls” command allows you to view a list of the files and folders in a given directory.
- The -l flag instructs Linux to print out a list of files with detailed descriptions.
- A forward slash “/” is used to refer to the root directory.

Correct and false runs

```
> ./main
Run
    Diff.
        Things
total 88
drwxr-xr-x   1 root   root     4096 Feb 29 21:32 bin
drwxr-xr-x   2 root   root     4096 Apr 24  2018 boot
drwxr-xr-x   1 runner runner   4096 Dec  4 19:14 config
drwxr-xr-x   5 root   root      340 Feb 29 21:32 dev
drwxr-xr-x   1 root   root     4096 Feb 29 21:32 etc
drwxr-xr-x   3 root   root     4096 Nov 21 00:03 hom
drwxr-xr-x   1 root   root     4096 Nov 21 00:01 home
dr-xr-xr-x   4 nobody nogroup  4096 Feb 29 21:32 io
drwxr-xr-x   1 root   root     4096 Nov 21 00:01 lib
drwxr-xr-x   2 root   root     4096 Nov 20 23:50 lib32
drwxr-xr-x   2 root   root     4096 Oct 29 21:25 lib64
drwxr-xr-x   2 root   root     4096 Oct 29 21:25 media
drwxr-xr-x   2 root   root     4096 Oct 29 21:25 mnt
drwxr-xr-x   1 root   root     4096 Dec  4 19:14 opt
dr-xr-xr-x 777 nobody nogroup     0 Feb 29 21:32 proc
drwx------   1 root   root     4096 Dec  4 19:13 root
drwxr-xr-x   1 root   root     4096 Dec  4 19:13 run
drwxr-xr-x   1 root   root     4096 Dec  4 19:14 run_dir
drwxr-xr-x   1 root   root     4096 Feb 29 21:32 sbin
drwxr-xr-x   2 root   root     4096 Oct 29 21:25 srv
dr-xr-xr-x  13 nobody nogroup     0 Feb 29 10:25 sys
drwxrwxrwt   1 root   root     4096 Feb 29 22:46 tmp
drwxr-xr-x   1 root   root     4096 Nov 21 00:15 usr
drwxr-xr-x   1 root   root     4096 Nov 21 00:01 var
Done with the main prgram
>
```

```
> ./main
Run
    Diff.
        Things
an error occured in execvp
Done with the main prgram
>
```

execvp() vs. execl()

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
int main()
{
  pid_t child_pid;
  child_pid=fork();
  if(child_pid!=0){
     printf("Run\n");
     printf("  Diff.\n");
     printf("    Things\n");
     sleep(5); //sleep for 5 seconds
  } else{
     char* arg_list[]={"ls","-l","/", NULL};
     execvp(arg_list[0],arg_list);
     fprintf(stderr,"an error occured in execvp\n");
      abort();
  }
  printf("Done with the main prgram\n");
  return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
int main()
{
  pid_t child_pid;
  child_pid=fork();
  if(child_pid!=0){
     printf("Run\n");
     printf("  Diff.\n");
     printf("    Things\n");
     sleep(5); //sleep for 5 seconds
  } else{
     execl("/bin/ls","ls", "-l", "/", NULL);
     fprintf(stderr,"an error occured in execvp\n");
     abort();
  }
   printf("Done with the main prgram\n");
  return 0;
}
```

Linux vs. Windows

- The DOS and windows API contain the *spawn* family of functions. This family takes the name of a program and creates a new process instance of it
- Linux does not contain a single function to perform the same task.
- Linux uses *fork* and *exec.*
  - *fork* makes the new process
  - *exec* transforms the process instance into the needed program

```
int spawnl(int mode, char *path, char *arg0, ...);
int spawnle(int mode, char *path, char *arg0, ..., char ** envp);
int spawnlp(int mode, char *path, char *arg0, ...);
int spawnlpe(int mode, char *path, char *arg0, ..., char ** envp);
int spawnv(int mode, char *path, char **argv);
int spawnve(int mode, char *path, char **argv, char ** envp);
int spawnvp(int mode, char *path, char **argv);
int spawnvpe(int mode, char *path, char **argv, char ** envp);
```

Name	Notes
e	An array of pointers to environment arguments is explicitly passed to the child process.
l	Command line arguments are passed individually to the function.
p	Uses the PATH argument variable to find the file to be executed.
v	Command line arguments are passed to the function as an array of pointers.

Reference: https://en.wikipedia.org/wiki/Spawn_(computing)