Introduction to Unix System Programming

Introduction to Unix System Programming

- **PROCESS MANAGEMENT**

- When a process duplicates,
the parent and child processes are virtually identical
(except for aspects like PIDs, PPIDs, and runtimes);

the child's code, data, and stack are a copy of the parent's,
and the processes even continue to execute the same code.

- A child process may replace its code with that of another executable file, there by differentiating itself from its parent.

- When "init" starts executing, it quickly duplicates several times.

Each of the duplicate child processes then replaces its code from the executable file called "getty",
which is responsible for handling user logins.

Introduction to Unix System Programming

- **PROCESS MANAGEMENT**

Name	Function
fork	duplicates a process
getpid	obtains a process' ID number
getppid	obtains a parent process'ID number
exit	terminates a process
wait	waits for a child process
exec…	replaces the code, data, and stack of a process.

fork()

- **Creating a New Process: fork()**

- A process may duplicate itself by using "fork()", which works like this:

System Call: pid_t **fork**(void)

"fork()" causes a process to duplicate.
The child process is an almost-exact duplicate of the original parent process;

it inherits a copy of its parent's code, data, stack, open file descriptors, and signal table.

the parent and child processes have different process ID numbers and parent process ID numbers.

If "fork()" succeeds, it returns the PID of the child to the parent process and returns a value of 0 to the child process.

fork()

- **PROCESS MANAGEMENT**

- A process may obtain its own process ID and parent process ID numbers
  by using the "getpid()" and "getppid()" system calls, respectively.

- Here's a synopsis of these system calls:

System Call : pid_t **getpid**(void)
pid_t **getppid**(void)

"getpid()" and "getppid()" return a process'ID number and parent process' ID number, respectively.

The parent process ID number of PID 1 (i.e., "init") is 1.

-fork() has no argument.
-It returns > 0 to parent as successful creation of child and < 0 when error occurs.
-It returns 0 to child process.

fork()

- **PROCESS MANAGEMENT**

```
#include <stdio.h>
 main()
 { int  pid;
   printf("I'm the original process with PID %d and PPID %d. \n",
        getpid(), getppid() );
   pid = fork();   /* Duplicate.  Child and parent continue from here */
   if ( pid!= 0 )  /* pid is non-zero, so I must be the parent */
     {
       printf("I'm the parent process with PID %d and PPID %d. \n",
                   getpid(), getppid() );
       printf("My child's PID is %d \n", pid );
     }
   else  /* pid is zero, so I must be the child */
     {
       printf("I'm the child process with PID %d and PPID %d. \n",
          getpid(), getppid() );
     }
```

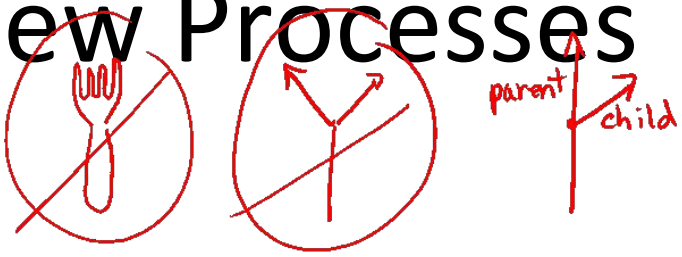
fork()

- **PROCESS MANAGEMENT**

```
    printf("PID %d terminates. \n", getpid() );  /* Both processes */
                                                 /* execute this */
}
```

```
$ myfork          ---> run the program.
I'm the original process with PID 13292 and PPID 13273.
I'm the parent process with PID 13292 and PPID 13273.
My child's PID is 13293.
I'm the child process with PID 13293 and PPID 13292.
PID 13293 terminates.  ---> child terminates.
PID 13292 terminates.  ---> parent terminates.
$ _
```

`fork`: Creating New Processes



returns a PID

- **`pid_t`** `fork(`**`void`**`)`
  - Creates a new "child" process that is *identical* to the calling "parent" process, including all state (memory, registers, etc.)
  - Returns 0 to the child process
  - Returns child's process ID (PID) to the parent process
- Child is *almost* identical to parent:
  - Child gets an identical (but separate) copy of the parent's virtual address space
  - Child has a different PID than the parent

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

parent gets child's PID

child gets 0

- `fork` is unique (and often confusing) because it is called once but returns "twice"

Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

pid = Y

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

pid = 0

Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

pid = Y

hello from parent

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

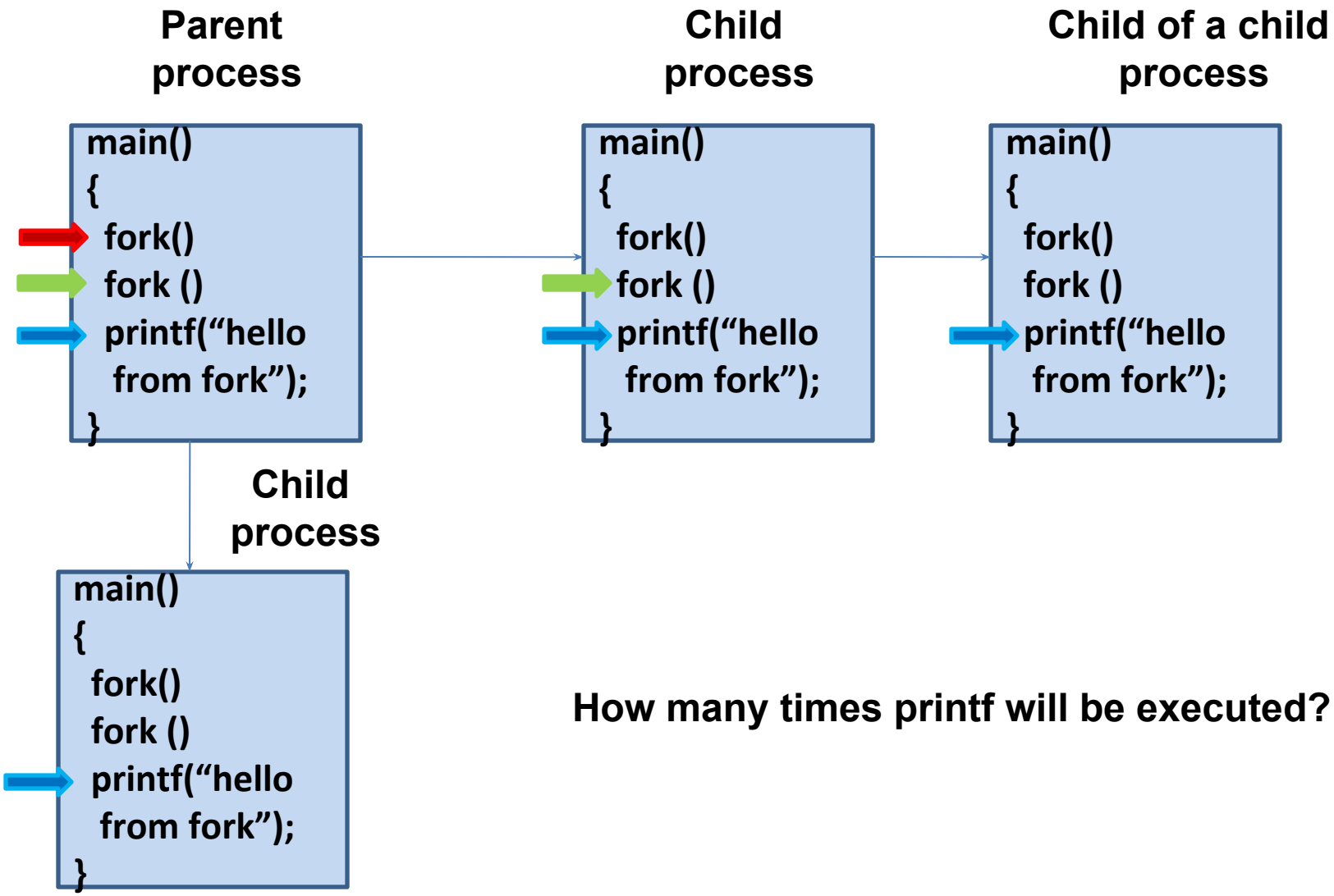
```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from
parent\n");
}
```

pid = 0

hello from child

Which one appears first?

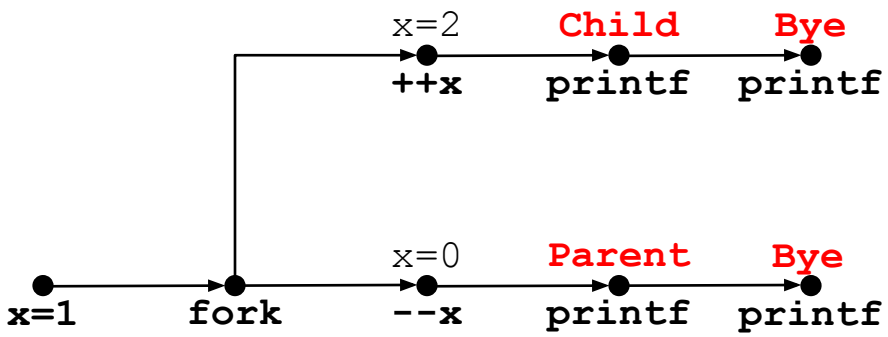
Understanding fork



How many times printf will be executed?

Fork Example: Possible Output

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
    printf("Child has x = %d\n", ++x);
    else
    printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```



- The 'C' odyssey unix -The Open Boundless C
- Meeta Gandhi, Tilak Shetty, Rajiv Shah

Lab Exercise

Write a program to create 4 processes: parent process and its child process which perform various tasks :

- Parent process count the frequency of a number
- 1st child calculate the sum of even numbers in an array
- 2nd child find total even number(s) in a given array
- 3rd child sort the array

- Main()
- { int fp, pid;
- Char chr=‘A’;
- pid= fork();
- If (pid==0)
- { fp=
- }