Fork System Call

- If `fork()` succeeds it returns the child PID to the parent and returns 0 to the child.
- If `fork()` fails, it returns -1 to the parent (no child is created) and sets `errno`.
- A program almost always uses this difference to do different things in the parent and child processes.
- Failure occurs when the limit of processes that can be created is reached.
- `pid_t` data type represents process identifiers.
- Other calls:
  - `pid_t getpid()` - returns the PID of calling process.
  - `Pid_t getppid()` - returns the PID of parent process.
int main ()
{
    pid_t pid;
    int status = 0;
    pid = fork();
    if (pid < 0)
        perror("fork()");
    else if (pid > 0) {
        /* parent */
        printf("I am parent\n");
        wait(0);
    } else {
        /* child */
        printf("I am child\n");
    }
    return 0;
}

The `fork` system call returns twice: it returns a zero to the child and the child process ID (pid) to the parent.

The `perror` function produces a message on the standard error output describing the last error encountered during a call to a system or library function (man page)

The `wait` function is used to terminate the parent process when the child terminates

pid is zero which indicates a child process

Example 1: fork1.c; fork1_pid.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t pid;
    int i;
    sum = 0;
    pid = fork();
    if( pid > 0 )
    {
        /* parent */
        for( i=0; i < 10; i++ )
            sum = sum + i;
        printf( "parent: sum is %d\n", sum);
        wait(0)
    }
    else
    {
        /* child */
        for( i=0; i < 10; i++ )
            sum = sum - i;
        printf( "child: sum is %d\n", sum);
    }
    return (0);
}

Example 2: fork2.c
fork() Example 2

- What is the value of `sum` in the parent and child processes after `pid = fork()`?
  - 0

- What is the value of `sum` in the parent and child processes at the `print` statements?
  - parent: `sum` is 45
  - child: `sum` is -45
fork() Example 2

- Remember that \texttt{sum} was 0 before the fork took place.
- When the fork took place the process was duplicated which means that a copy is made of each variable; \texttt{sum} was duplicated.
- Since \texttt{sum} was 0 just before the fork then \texttt{sum} is 0 right after the fork in both the child and parent processes.
fork() Example 3

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

int main()
{
    pid_t pid;
    int i;
    pid = fork();
    if( pid > 0 )
    {
        /* parent */
        for( i=0; i < 10; i++ )
            printf("\t\t\tPARENT %d\n", i);

        wait(0);
    }
    else
    {
        /* child */
        for( i=0; i < 10; i++ )
            printf("CHILD %d\n", i);
        return 0;
    }
}

Example 2: fork3.c

What is the possible output?
fork () Example 2: Possible Output

PARENT 0
PARENT 1
PARENT 2
PARENT 3
PARENT 4
PARENT 5
PARENT 6
PARENT 7
PARENT 8
PARENT 9

CHILD 0
CHILD 1
CHILD 2
CHILD 3
CHILD 4
CHILD 5
CHILD 6
CHILD 7
CHILD 8
CHILD 9
fork () Example 2: Possible Output

PARENT 0
PARENT 1
PARENT 2
PARENT 3
PARENT 4
PARENT 5
PARENT 6

CHILD 0
CHILD 1
CHILD 2
CHILD 3
CHILD 4
CHILD 5
CHILD 6
CHILD 7
CHILD 8
CHILD 9

Lots of possible outputs!!
Execution

- Processes get a share of the CPU before giving it up to give another process a turn.

- The switching between the parent and child depends on many factors:
  - machine load, system process scheduling

- Output interleaving is **nondeterministic**
  - Cannot determine output by looking at code.
How many Processes are Created by this Program?

```c
#include <stdio.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    fork();
    fork();
}
```
Process File Descriptor Table

- Every process has a process file descriptor table
- Each entry represents something that can be read from or written to e.g.,
  - file
  - Screen
  - pipe (later)
The OS maintains a system file descriptor table in the OS's memory space. Every open file in the system has an entry in this table. One file can be associated with many entries in this table (if opened by many processes). These entries contain among other things: the file descriptor's permissions, # links, the file offset which is moved by read and write. An entry is removed when 0 links point to it.
Assume that there was something like this in a program:

```c
FILE *in_file;
in_file = fopen("list.txt", "r");
```
Fork and Files

- In a fork what gets copied is the file descriptor table;
- Thus the parent and child point to the same entry in the system file table.
Fork and Files

Parent File Descriptor table

0  stdin
1  stdout
2  stderr
3  in_file

Child File Descriptor table

0  stdin
1  stdout
2  stderr
3  in_file

System file table

File info e.g., read offset
Fork and Files

- Open a file before a fork
  - The child process gets a copy of its parent's file descriptor table.
  - The child and parent share a file offset pointer because the open came before the fork.

- Open a file after a fork
  - Assume that parent and child each open a file after the fork
  - They get their own entries in the System File Descriptor table
    - This implies that the file position information is different
Question

Suppose that foobar.txt consists of the 6 ASCII characters foobar. Then what is the output of the following program?

```c
int main()
{
    FILE *fd1, *fd2;
    char c;

    fd1 = fopen("foobar.txt", "r");
    fd2 = fopen("foobar.txt", "r");
    fscanf(fd1, "%c", &c);
    fscanf(fd2, "%c", &c);
    printf("c = %c\n", c);
}
```
The descriptors \texttt{fd1} and \texttt{fd2} each have their own open file table entry, so each descriptor has its own file position for \texttt{foobar.txt}. Thus, the read from \texttt{fd2} reads the first byte of \texttt{foobar.txt}, and the output is
\begin{verbatim}
c = f
\end{verbatim}
and not
\begin{verbatim}
c = o
\end{verbatim}
Question

Example code: fd_1.c
int main()
{
    FILE *fd1, *fd2;
    char c;
    pid_t pid;

    fd1 = fopen("foobartxt", "r");
    fd2 = fopen("foobartxt", "r");

    pid = fork();
    if (pid > 0){
        fscanf(fd1, "%c", &c);
        printf("parent: c = %c\n", c);
    }else if (pid == 0) {
        fscanf(fd2, "%c", &c);
        printf("child: c = %c\n", c);
    }
}

Example code: fd_fork_1.c
Fork and Files

- Be careful
- It is much better to open files after forks.
- Even so you need to be careful if there is writing
  - This often requires that a process has mutual exclusive access during the write process (more later in the course)
Wait

- Parents waits for a child (system call)
  - Blocks until a child terminates
  - Returns pid of the child process
  - Returns -1 if no child process exists (already exited)
  - status
    - `#include <sys/types.h>`
    - `#include <sys/wait.h>`
    - `pid_t wait(int *status)`

- Parent waits for a specific child to terminate
  - `pid_t waitpid(pid_t pid, int *status, int options)`
int main ()
{
    pid_t pid;
    int status = 0;
    pid = fork();
    if (pid < 0)
        perror("fork()");
    if (pid > 0) {
        /* parent */
        printf("I am parent\n");
        pid = wait(&status);
    } else {
        /* child */
        printf("I am child\n");
        exit(status);
    }
}
More about Process Operations

- In Unix-based systems, a hierarchy of processes is formed.
- In Unix, we can obtain a listing of processes by using the `ps` command.
- `ps -el` will list complete information for all processes.
The term `exec` refers to a family of functions where each of the functions replace a process’s program (the one calling one of the exec functions) with a new loaded program.

- A call to a function from `exec` loads a binary file into memory (destroying the memory image of the program calling it).
- The new program starts executing from the beginning (where `main` begins).
- On success, `exec` never returns; on failure, `exec` returns `-1`.

The different versions are different primarily in the way parameters are passed.
Exec

The exec family consists of these functions: `execvp`, `execlp`, `execv`, `execve`, `execl`, `execlp`, `execle`, `execve`

- Functions with `p` in their name (`execvp`, `execlp`) search for the program in the path indicated by the PATH environment variable; functions without `p` must be given full path.
- Functions with `v` in their name (`execv`, `execvp`, `execve`) differ from functions with `l` (`execl`, `execlp`, `execle`) in the way arguments are passed.
- Functions with `e` accept array of environment variables.
Versions of exec

Versions of exec offered by C library:

- `int execl( const char *path, const char *arg, ... );`
- `int execlp( const char *file, const char *arg, ... );`
- `int execle( const char *path, const char *arg, ..., char *const envp[] );`
- `int execv( const char *path, char *const argv[] );`
- `int execvp( const char *file, char *const argv[] );`
- `int execve( const char *filename, char *const argv [], char *const envp[] );`

Exec Example

Program A:
    int i = 5;
    printf("%d\n",i);
    execl("B", "B", NULL);
    printf("%d\n",i);

Program B:
    main()
    {
        printf("hello\n");
    }

- What is the output of program A?
  - 5
  - hello

- Why is it not this?
  - 5
  - hello
  - 5

- The `execl` command replaces the instructions in the process with instructions for program B. It starts at the first instruction (starts at `main`)

Example code: exec1.c
Program A:
```c
int i = 5;
prog_argv[0] = "B";
prog_argv[1] = NULL;
printf("%d\n",i);
execv(prog_argv[0],
     prog_argv);
printf("%d\n",i);
```

Program B:
```c
main()
{
    printf("hello\n");
}
```

- Same functionality as the program on the previous slide
- Used `execv` instead of `execl`
- `execv` uses an array to pass arguments
- `execl` uses a list to pass arguments

Example code: exec2.c
Exec Example

```c
int main(int argc, char *argv[]) {
    int i = 5;
    int status;

    status = execlp("ls", "ls", "-l", NULL);
    if (status != 0) {
        perror("you goofed\n");
        printf("errno is %d\n", errno);
    }
    printf("%d\n", i);
}
```

- In this example, note that the command is
  - `ls -l`
- Each argument is in the list

Example code: exec3.c
Exec Example

```c
int main(int argc, char *argv[])
{
    char *prog1_argv[4];
    int i = 5;

    prog1_argv[0] = "ls";
    prog1_argv[1] = "-l";
    prog1_argv[2] = NULL;

    execvp(prog1_argv[0],
            prog1_argv);

    ...
    printf("%d\n",i);
}
```

- Same example as that on the previous side but `execvp` is used which requires an array
Fork and Exec

- Child process may choose to execute some other program than the parent by using one of the exec calls.
- Exec overlays a new program on the existing process.
- Child will not return to the old program unless exec fails. This is an important point to remember.
Example code: exec4.c