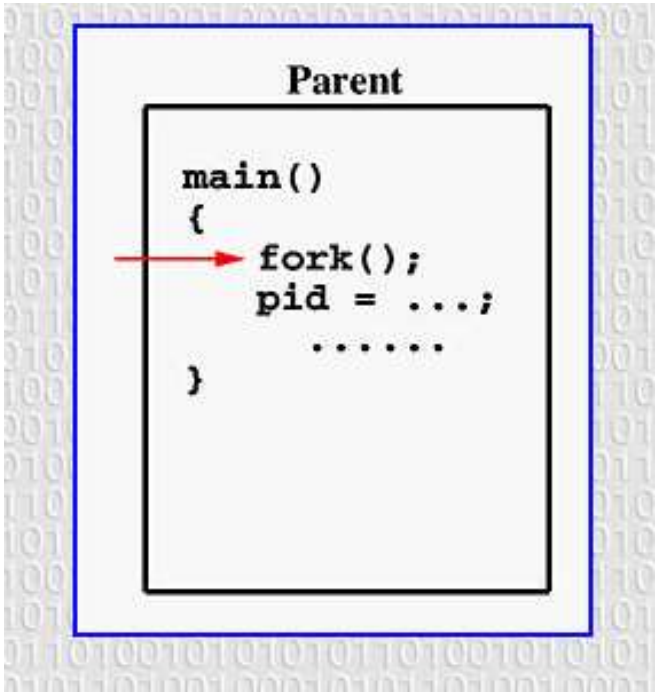
# The fork() System Call

- 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.

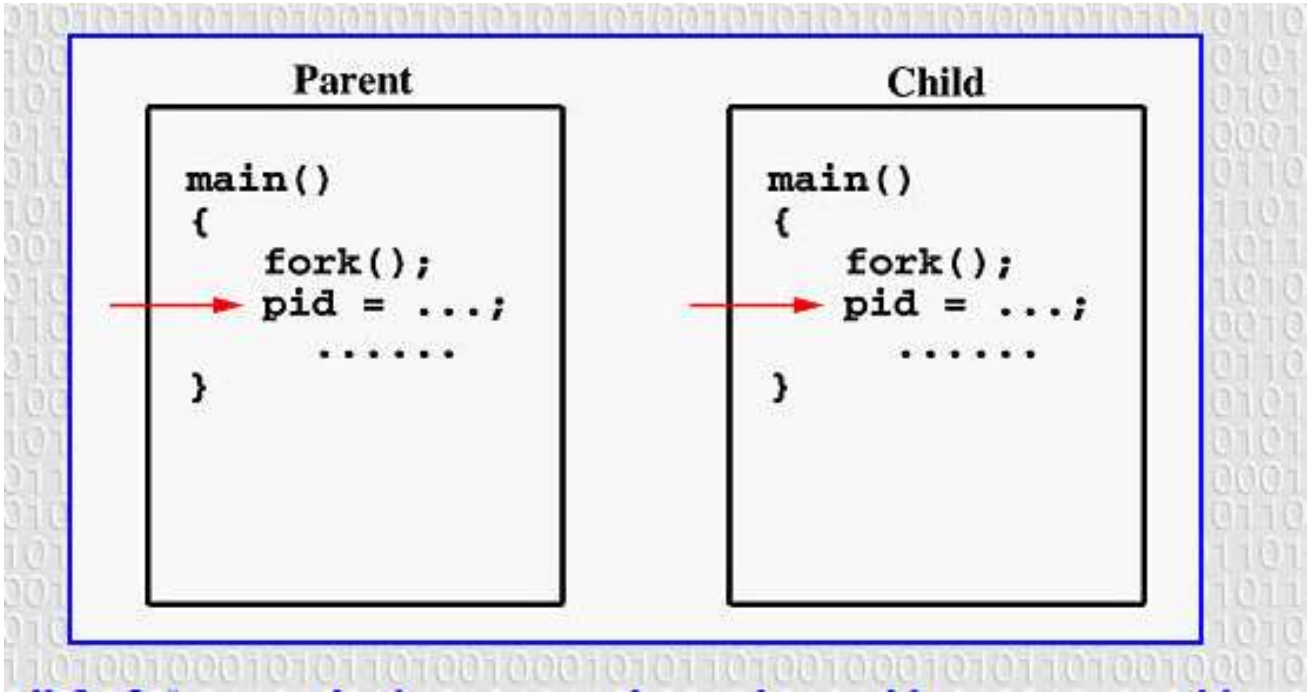
- Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():
  - If fork() returns a negative value, the creation of a child process was unsuccessful.
  - fork() returns a zero to the newly created child process.
  - fork() returns a positive 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.
- 
- Therefore, after the system call to fork(), a simple test can tell which process is the child.
- Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.

```
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
```


Parent
main()
{
fork();
pid = ...;
.....
}

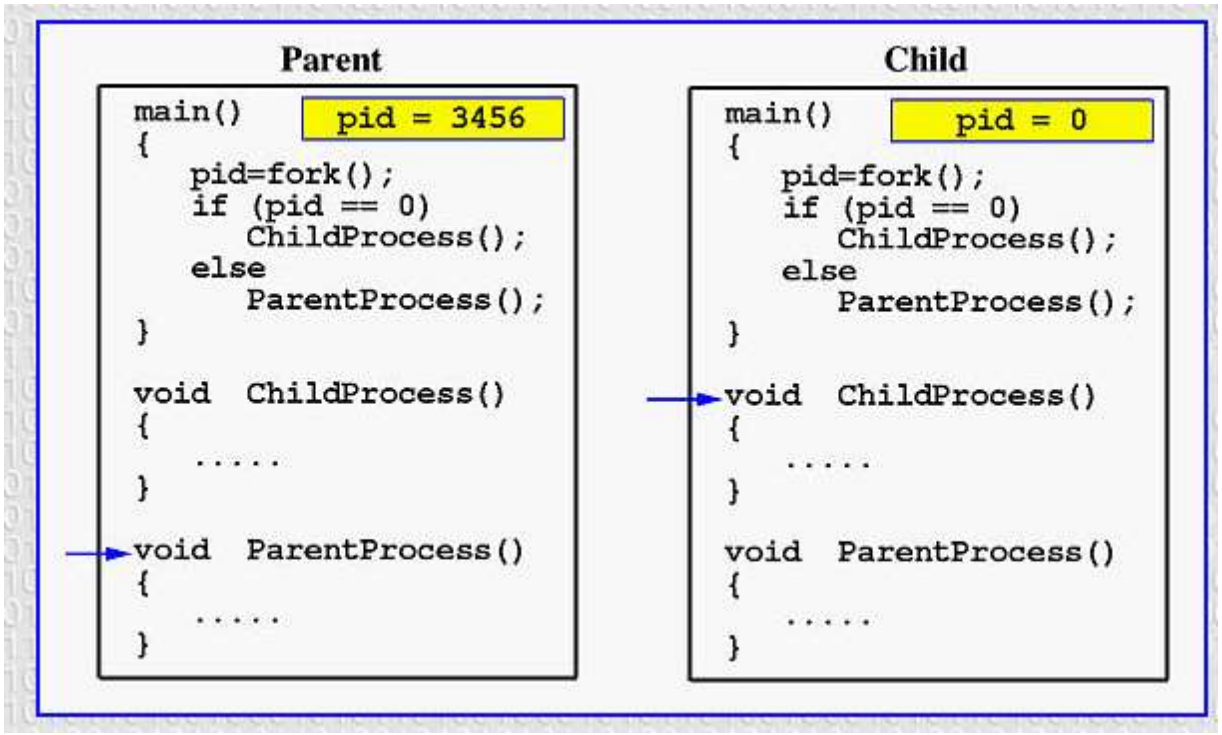
- If the call to fork() is executed successfully, Unix will
- make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the fork() call.
-


Parent
main()
{
fork();
pid = ...;
.....
}
Child
main()
{
fork();
pid = ...;
.....
}

- Those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others.

# What is the reason of using write rather than printf?

- It is because printf() is "buffered," meaning printf() will group the output of a process together. While buffering the output for the parent process, the child may also use printf to print out some information, which will also be buffered.
- As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result.
- Worse, the output from the two processes may be mixed in strange ways.
- To overcome this problem, you may consider to use the "unbuffered" write.


Parent
Child
main()
pid = 3456
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}
main()
pid = 0
{
pid=fork();
if (pid == 0)
ChildProcess();
else
ParentProcess();
}
void ChildProcess()
{
.....
}
void ParentProcess()
{
.....
}

```
#include <stdio.h>
#include <sys/types.h>i
int main(){
fork();
fork();
fork();
 printf("hello\n");    return 0;}
```

Output:

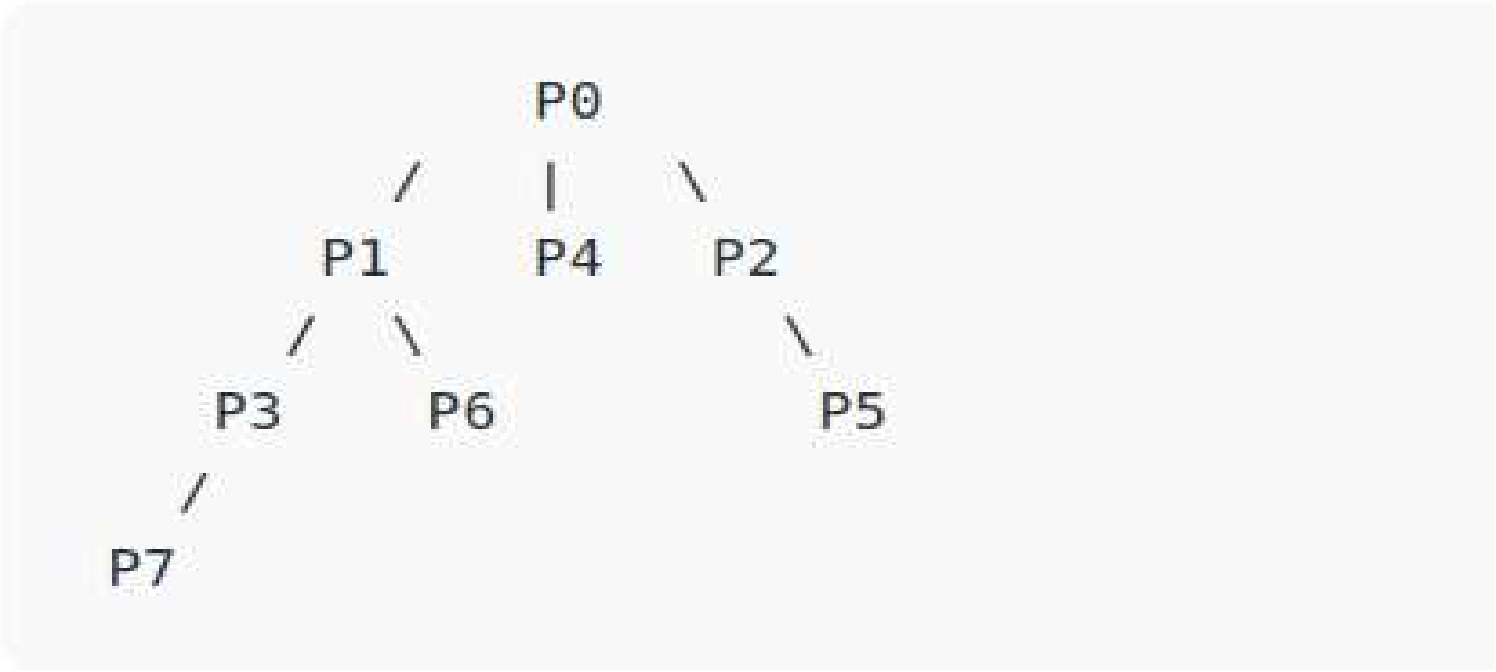
```
hello
hello
hello
hello
hello
hello
hello
hello
```

The main process: P0

Processes created by the 1st fork: P1

Processes created by the 2nd fork: P2, P3

Processes created by the 3rd fork: P4, P5, P6, P7



# grep

- Grep is a useful command to search for matching patterns in a file.
- grep is short for **"global regular expression print".**
- If you are a system admin who needs to scrape through log files or a developer trying to find certain occurrences in the code file, then grep is a powerful command to use.

- int execv(const char* path, char *const argv[])
- execv(path,argv) causes the current process to abandon the program that it is running and start running the program in file path. Parameter argv is the argument vector for the command, with a null pointer at the end. It is an array of strings.
- Normally, execv does not return, since the current program is thrown away, and another program is started instead. In the event of an error (such as file path not being found), execv will return. Any return of execv indicates an error.

- Example: Cause the current program to be replaced by a program running command ls -l -F.
-
-
- char* argv[4];
- argv[0] = "ls";
- argv[1] = "-l";
- argv[2] = "-F";
- argv[3] = NULL;
- execv(argv[0], argv);
-
-
- pid_t wait(int *status)
- wait(v) waits for a child process of the current process to terminate. When one does, wait stores the termination status of the terminated child (the value returned by main) into variable status, and returns the process number of the terminated child process.
-
- If status is NULL, then the status value will not be stored.
-

# Wait() system call

- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then terminated the child remains in a "zombie" state (see NOTES below).

- The value of pid can be:
- Tag Description
- < -1 meaning wait for any child process whose process group ID is equal to the absolute value of pid.
- -1 meaning wait for any child process.
- 0 meaning wait for any child process whose process group ID is equal to that of the calling process.
- > 0 meaning wait for the child whose process ID is equal to the value of pid.

- #include <sys/wait.h>
- int wait(int *stat_loc);
-

stat_loc
A pointer to an integer where the wait function will return the status of the child process. If the wait function returns because the status of a child process is available, the return value will equal the process identifier (ID) of the exiting process. For this, if the value of stat_loc is not NULL, information is stored in the location pointed to by stat_loc. If status is returned from a terminated child process that returned a value of 0, the value stored at the location pointed to by stat_loc will be 0. If the return is greater than 0, you can evaluate this information using the following macros:
WEXITSTATUS
WIFEXITED
WIFSIGNALED
WTERMSIG.

# opendir(), readdir() and closedir()

- In Linux, DIR and dirent are related to directory handling.
- 
- **DIR** is a type that represents a directory stream, which is an ordered sequence of directory entries. It is typically used in conjunction with functions such as opendir() and closedir() to open and close directories and retrieve the contents of a directory.
- 
- **dirent** is a structure that represents a directory entry. It contains information such as the name of the file or directory, its size, and its type. It is used in conjunction with functions such as readdir() to read the contents of a directory stream.

- #include <dirent.h>
-
- DIR *opendir(const char *name);

- The function takes a single argument, which is a string that specifies the name of the directory to open. The directory name can be either an absolute path or a path relative to the current working directory.
- 
- The function returns a pointer to a DIR structure, which represents the directory stream. If the function fails to open the directory for any reason, it returns a **null** pointer and sets the global variable **errno** to indicate the specific error that occurred.

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

int main() {
    DIR *dir;

    dir = opendir(".");
    if (dir == NULL) {
        perror("opendir");
        return 1;
    }

    printf("Directory opened successfully!\n");

    closedir(dir);
    return 0;
}
```

- In this example, we call **opendir()** with the string **"."**, which represents the current working directory.
- We then check if the function returns a null pointer to indicate an error.
- If the directory was opened successfully, we print a message to indicate that, and then close the directory stream using **closedir()**.
-

# closedir()

- In Linux, the closedir() system call is used to close a directory stream that was previously opened with opendir(). Here's the syntax of the closedir() system call:
- **#include <dirent.h>**
- 
- **int closedir(DIR *dirp);**

- The closedir() system call takes one argument:
- 
- **dirp:** a pointer to a DIR structure representing the directory stream to close.
- 
- The closedir() system call returns 0 if successful, or -1 to indicate an error.
- 

- The **dirent** structure is used in Linux to represent a directory entry. Here's the definition of the dirent structure:
- **struct dirent {**
- **ino_t d_ino; /* inode number */**
- **off_t d_off; /* offset to the next dirent */**
- **unsigned short d_reclen; /* length of this record */**
- **unsigned char d_type; /* type of file; not supported by all file system types */**
- **char d_name[256]; /* filename */**
- **};**

- The members of the structure are:
- 
- **d_ino:** the inode number of the file, which uniquely identifies it within the file system.
- **d_off:** the offset from the beginning of the directory stream to the next dirent structure.
- **d_reclen:** the length of this dirent structure. Note that this value may not be the same as the size of the dirent structure itself, due to alignment considerations.
- **d_type:** the type of the file. This field is not supported by all file system types, and on some systems, it may always be set to DT_UNKNOWN.
- **d_name:** the null-terminated filename, with a maximum length of 255 characters.

- The **dirent** structure is typically used in conjunction with functions such as **readdir()** to read the contents of a directory stream. The **readdir() function returns a pointer to a dirent structure** containing information about the next directory entry, or a null pointer to indicate the end of the directory stream.

# Open(), read(), write()

- In Linux, the open() system call is used to open a file and obtain a file descriptor, which is a unique integer value that identifies the file. Here's the syntax of the open() system call:
- **#include <sys/types.h>**
- **#include <sys/stat.h>**
- **#include <fcntl.h>**
-
- **int open(const char *pathname, int flags);**
- **int open(const char *pathname, int flags, mode_t mode);**

- **The open() system call takes two or three arguments:**
- 
- **pathname:** a string that specifies the name of the file to open. This can be either an absolute path or a path relative to the current working directory.
- **flags:** a set of flags that specify the mode in which to open the file. These can include the **O_RDONLY** flag to open the file for reading, **O_WRONLY** to open the file for writing, or **O_RDWR** to open the file for both reading and writing. Other flags include **O_CREAT** to create the file if it does not exist, **O_TRUNC** to truncate the file to zero length if it already exists, and **O_APPEND** to append data to the end of the file.
- **mode (optional):** a set of permissions that specify the access rights for the file, if it needs to be created. This argument is only used when the O_CREAT flag is specified. The mode argument is typically specified as an octal value, such as **0644.**

- The open() system call returns a **non-negative integer** file descriptor if successful,
- or a **negative** value to indicate an error.
- If the file cannot be opened for any reason, the function sets the global variable **errno** to indicate the specific error that occurred.

# read()

- In Linux, the read() system call is used to read data from a file descriptor. Here's the syntax of the read() system call:
- **#include <unistd.h>**
-
- **ssize_t read(int fd, void *buf, size_t count);**
-
-

- The read() system call takes three arguments:
- 
- **fd:** an integer file descriptor that specifies the file to read from.
- **buf:** a pointer to a buffer that is used to store the data read from the file.
- **count:** an integer that specifies the maximum number of bytes to read from the file.
- 
- The **read()** system call returns a non-negative integer that specifies the number of bytes read if successful, or a negative value to indicate an error. If the function returns zero, it means that the end of the file has been reached.
- 
- 

# write()

- In Linux, the write() system call is used to write data to a file descriptor. Here's the syntax of the write() system call:
- **#include <unistd.h>**
- 
- **ssize_t write(int fd, const void *buf, size_t count);**
- 

- The write() system call takes three arguments:
- 
- **fd**: an integer file descriptor that specifies the file to write to.
- **buf:** a pointer to a buffer that contains the data to write to the file.
- **count**: an integer that specifies the number of bytes to write to the file.
- 
- The write() system call returns a non-negative integer that specifies the number of bytes written if successful, or a negative value to indicate an error.

# cretae()

- In Linux, the **creat()** system call is used to create a new file with a specified name and set of permissions. Here's the syntax of the creat() system call:
- **#include <sys/types.h>**
- **#include <sys/stat.h>**
- **#include <fcntl.h>**
-
- **int creat(const char *pathname, mode_t mode);**
-

- The creat() system call takes two arguments:
-
- **pathname:** a string that specifies the name of the file to create. This can be either an absolute path or a path relative to the current working directory.
- **mode**: a set of permissions that specify the access rights for the file. The mode argument is typically specified as an octal value, such as 0644.
-
-