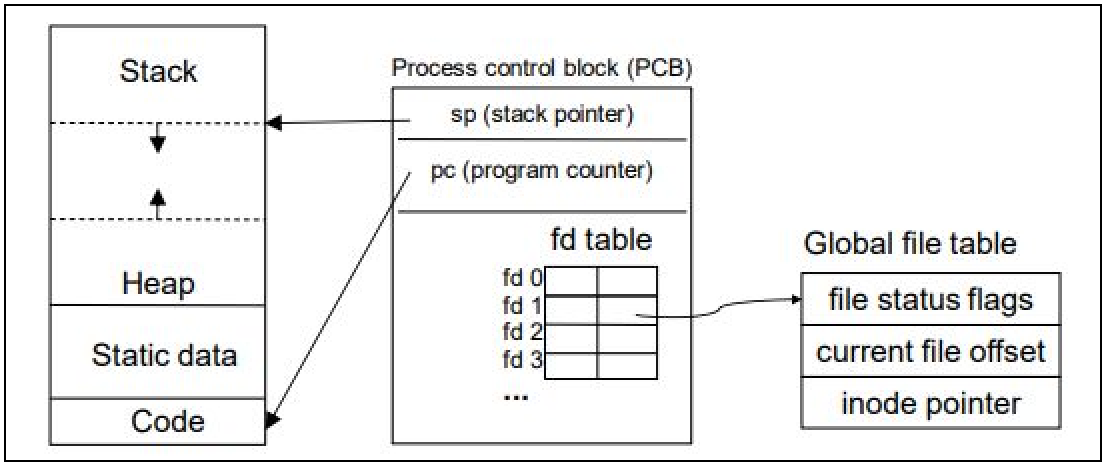
CSCB09 Week 8 Notes

1. **Parallel programs:**
   - Hardware is not getting that much faster anymore. Therefore, we can use more hardware to solve a problem.
     I.e. Multiple processors/cores and multiple machines.
   - Processes often need to communicate. This means that processes of a parallel program need to exchange data and/or synchronize. However, if you use fork, the parent and the child cannot use the same variables as each process has a separate copy of each variable.
   - We can use pipes to solve this problem.
2. **Pipes:**
   - Pipes are a one-way (half-duplex) communication channel.
   - Pipes are a communication medium between two or more related or interrelated processes.
   - It can be either within one process or a communication between the child and the parent processes.
   - Communication can also be multi-level such as communication between the parent, the child and the grand-child, etc.
   - Pipes are buffers managed by the OS.
   - Processes use low-level file descriptors for pipe operations.
   - Communication is achieved by one process writing into the pipe and other reading from the pipe. To achieve the pipe system call, create two files, one to write into the file and another to read from the file.
3. **I/O mechanisms in C:**
   - So far we have used **file pointers** (regular files).
     I.e. FILE *file
   - For pipes, we need to use **file descriptors (fd)**, which are low level.
   - A file descriptor is an integer that uniquely identifies an open file in a computer's operating system.
   - When a process makes a successful request to open a file, the kernel returns a file descriptor which points to an entry in the kernel's global file table. The file table entry contains information such as the inode of the file, byte offset, and the access restrictions for that data stream (read-only, write-only, etc).
   - A **file descriptor table (fd table)** is a collection of file descriptors in which elements are pointers to the file table's entries.
   - A unique file descriptor table is provided in the operating system for each process.
   - Each process's fd table has all the open files of that process.
   - The global fd table has all the open files system-wide.

CSCB09 Week 8 Notes



- The fd table is preserved if you exec.
- If you fork, a child gets a copy of parent's fd table, so it will have same files open. Both the parent's and the child's fd will point to same entry in the global file table, so they see the same offset.
- **Note:** File descriptors created after calling fork are not shared.
- All programs automatically have three files open:
    - FILE *stdin
    - FILE *stdout
    - FILE *stderr;
- These are known as the **Standard File Descriptors**.

	Stdio name	File descriptor
Standard input	stdin	0 (STDIN_FILENO)
Standard output	stdout	1 (STDOUT_FILENO)
Standard error	stderr	2 (STDERR_FILENO)

- A useful system call to get the corresponding fd for a FILE object is: **int fileno(FILE *stream);**
- The operations for file descriptors are open, close, read and write.
- **Open:**
    - Used to open the file for reading, writing or both. This returns a file descriptor upon success or -1 upon failure.
    - Syntax: **int open(const char *pathname, int flags);**
    - **Pathname** is the path to the file that you want to use. Use an absolute path that begins with "/", if you are not working in same directory as the file. Use a relative path if you are working in same directory as the file.
    - **Flags:**
        - **O_RDONLY:** Opens the file for read-only.
        - **O_WRONLY:** Opens the file for write-only.
        - **O_RDWR:** Opens the file for both reading and writing.

    - **Note:** To use the open function, you must include the following:
        - **#include<sys/types.h>**
        - **#include<sys/stat.h>**
        - **#include <fcntl.h>**
- **Read:**
    - This reads count bytes of input into the memory area indicted by buf from the file indicated by fd. This returns the number of bytes read on success, 0 on reaching the end of file and -1 on error.
    - Syntax: **ssize_t read(int fd, void *buf, size_t count);**
    - **buf:**
        - The buffer (Memory Area) that stores the data when you read it.
        - buf needs to point to a valid memory location with size not smaller than count.
    - **count:**
        - This is the length of the buffer.
          I.e. This is the number of bytes requested to be read.
        - **Note:** Sometimes, the number of bytes requested to be read is not the same as the actual number of bytes read.
- **Write:**
    - This writes count bytes from buf to the file indicated by fd. If cnt is zero, write simply returns 0 without attempting any other action. Furthermore, write will overwrite the contents of the file, if there are any. This returns the number of bytes written on success and -1 on error.
    - Syntax: **ssize_t write(int fd, const void *buf, size_t count);**
    - **buf:**
        - The buffer that stores the data being written from.
        - buf needs to point to a valid memory location with size not smaller than count.
    - **count:**
        - The length of the buffer.
          I.e. This is the number of bytes requested to be written.
        - **Note:** Sometimes, the number of bytes requested to be written is not the same as the actual number of bytes written.
- **Close:**
    - Closes the file pointed by fd.
    - This returns 0 on success and -1 on failure.
    - Syntax: **int close(int fd);**

4. **Programming With Pipes:**
    - You need to use **#include<unistd.h>**.
    - Syntax: **int pipe(int pipefd[2]);**
    - You pass a pointer to two integers (i.e. an array or a malloc of two ints) and pipe fills it with two newly opened FDs.

I.e. pipe creates an OS internal system buffer and two file descriptors, one for reading and one for writing.

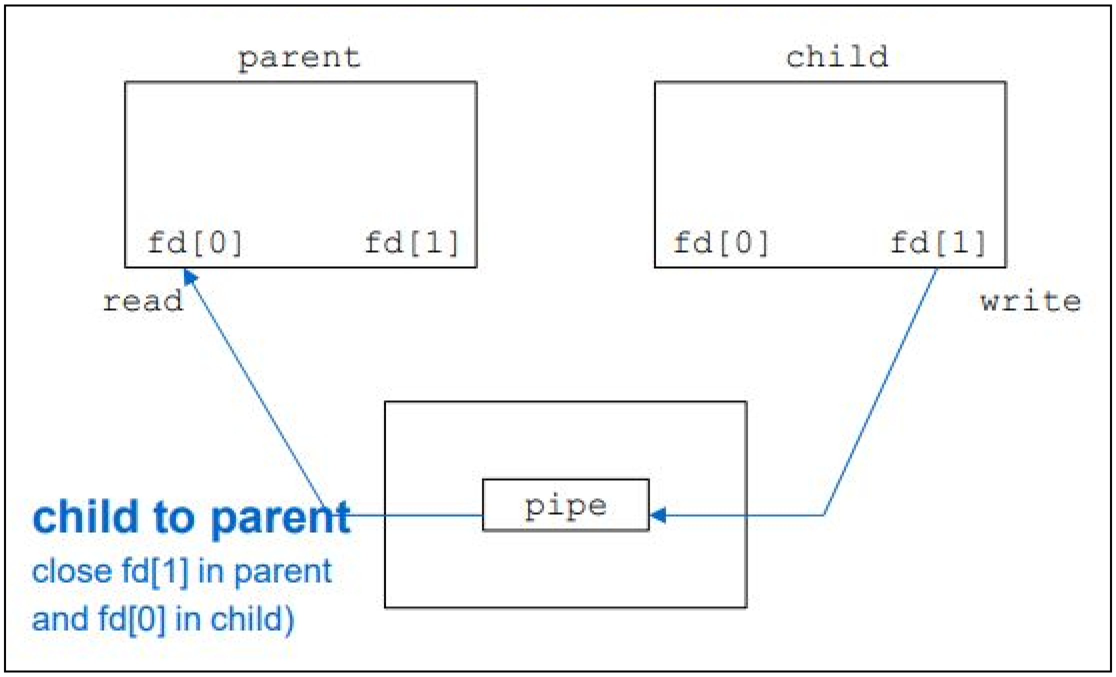
- pipefd[0] is used for reading.
- pipefd[1] is used for writing.
- Whatever is written into pipefd[1] can be read from pipefd[0].
- Returns 0 on success and -1 on error. To know the cause of failure, check with errno or perror.
- Read blocks until data is available in the pipe. If the writing end is closed, read detects EOF and returns 0.
- All open pipes (and other FDs) are closed when a process exits.
- In general, each process will read or write a pipe (not both). You should close the end you are not using.
  I.e. Before reading: close pipefd[1].
  Before writing: close pipefd[0].
- If you don't close the end you are not using, your code might hang.
- E.g.

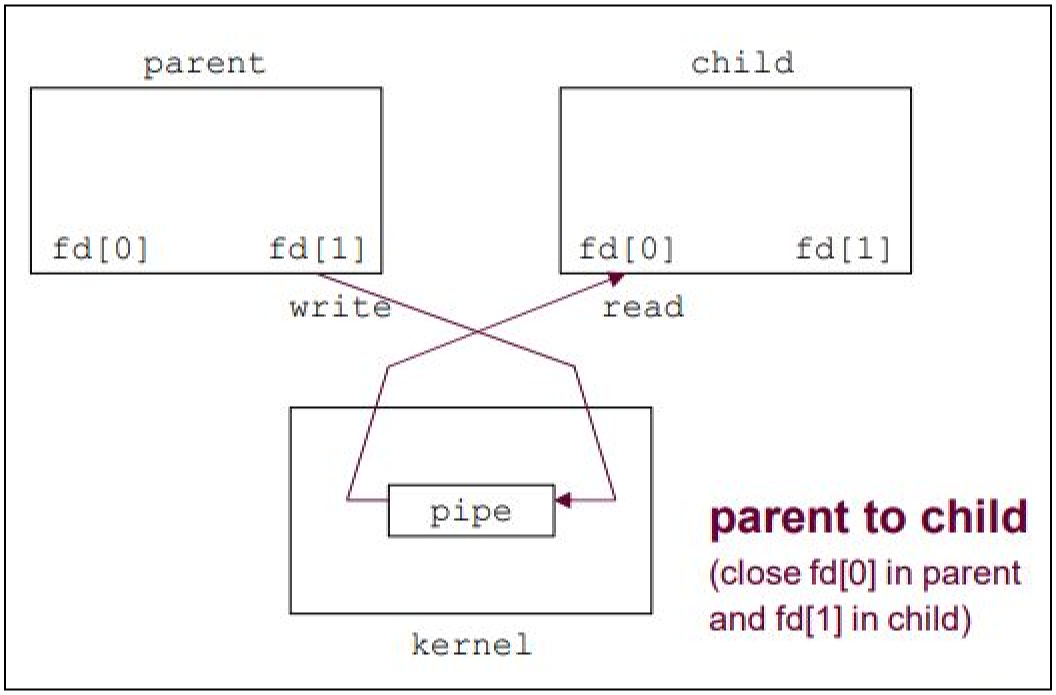
```
if (fork == 0){
   write(p[1], msg, msgsize);
}else{
   while((nbytes=read(p[0], buf, bufsize)) > 0){
      write(STDOUT_FILENO, buf, nbytes);
   }
}
```

This code will hang because parent wants to read until EOF and write. However, because p[1] wasn't closed, read is always expecting more data, so it blocks.

```
if (fork == 0){
   write(p[1], msg, msgsize);
}else{
  close(p[1]);
   while((nbytes=read(p[0], buf, bufsize)) > 0){
      write(STDOUT_FILENO, buf, nbytes);
   }
}
```

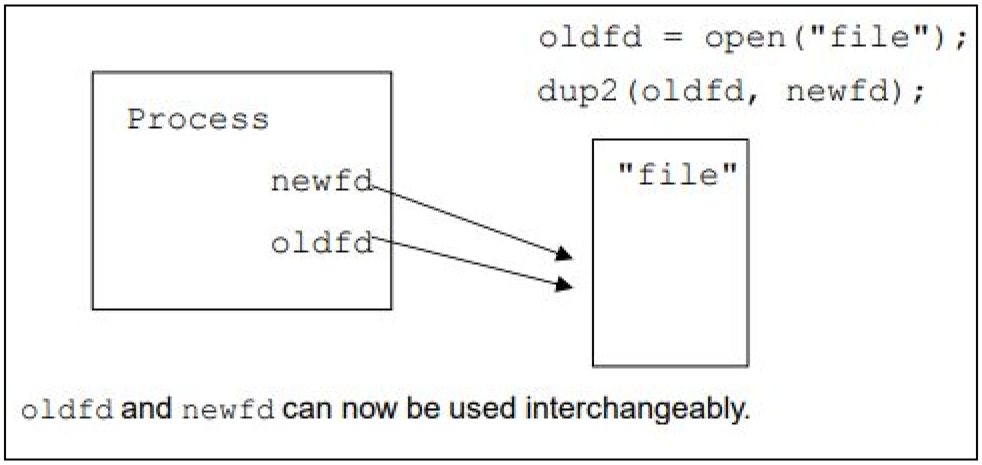
Because we closed p[1] before reading, now the parent won't hang.





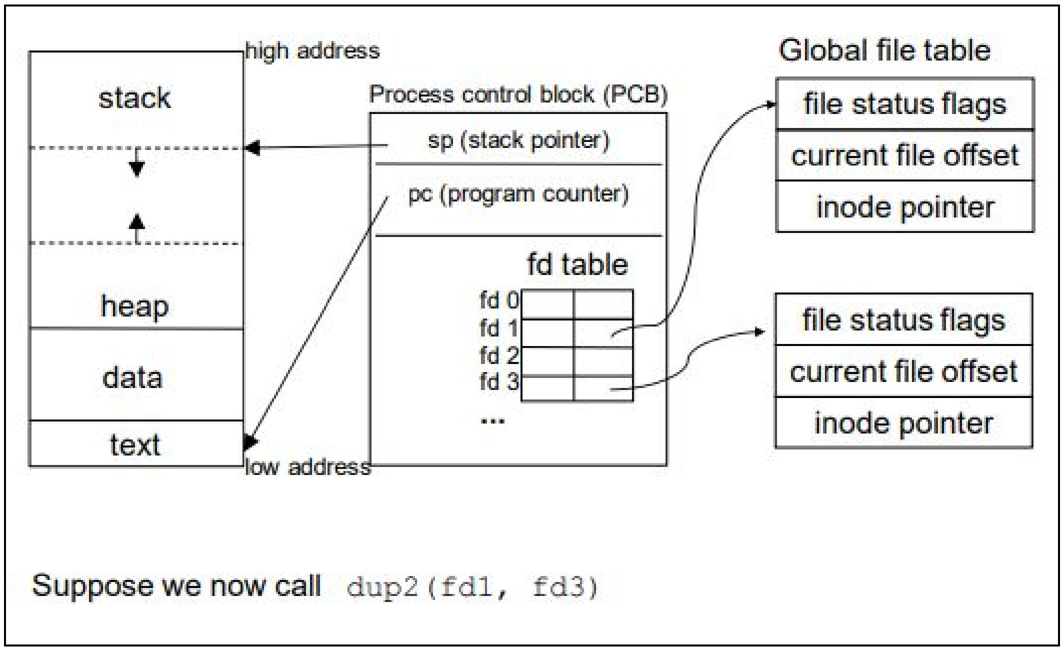
-
- If you try to write to a pipe and no one has reading end open, you get a signal (SIGPIPE) that will terminate program.
- If you write to a pipe and the buffer is full, write blocks until space frees up.
- If you read from a pipe and nobody has a writing end open, read returns EOF.
- If you read from a pipe and there is no data read blocks.

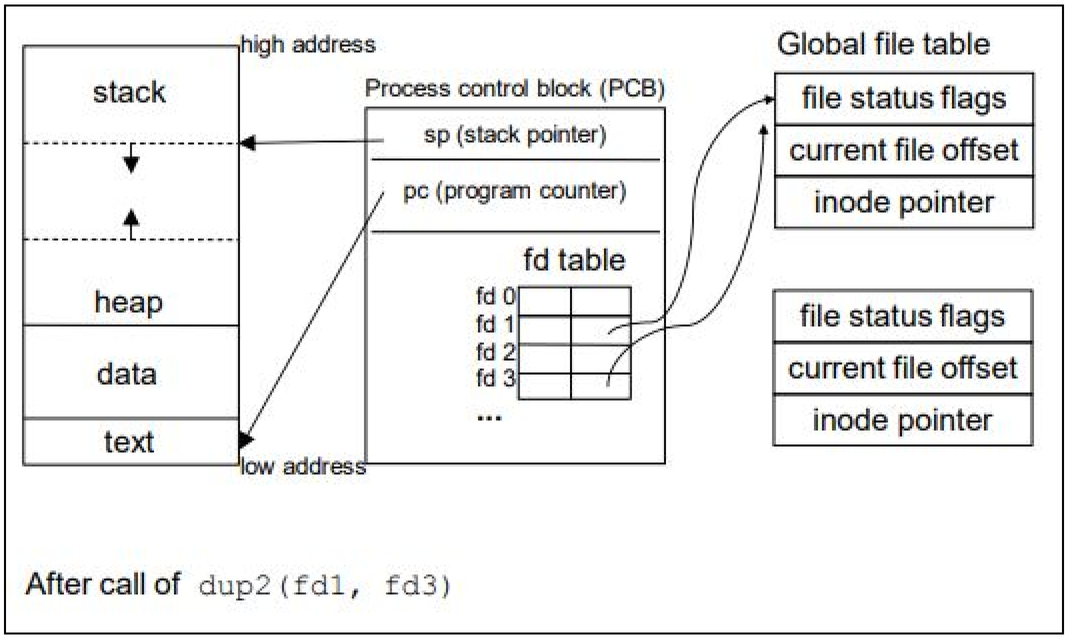
5. **<u>Dup2:</u>**
    - Syntax: **int dup2(oldfd, newfd);**
    - dup2 duplicates the file descriptor of oldfd into newfd.
    - If the descriptor newfd was previously open, it is silently closed before being reused.
    - If oldfd is not a valid file descriptor, then the call fails, and newfd is not closed.
    - If oldfd is a valid file descriptor, and newfd has the same value as oldfd, then dup2() does nothing, and returns newfd.
    - Note: You must include **#include<unistd.h>** to use dup2.
    - On success, dup2 returns a new file descriptor, newfd, that has the following in common with the original:
        - Same open file or pipe.
        - Same file pointer (both file descriptors share one file pointer).
        - Same access mode (read, write, or read/write).
    - On failure, dup2 returns -1 and errno is set accordingly.
    - dup2 is used to redirect input and output. It works similar to ">" and "<" from Unix commands.
    - E.g.


```
oldfd = open("file");
dup2(oldfd, newfd);
```

oldfd and newfd can now be used interchangeably.

- E.g.


high address
stack
heap
data
text
low address
Process control block (PCB)
sp (stack pointer)
pc (program counter)
fd table
fd 0
fd 1
fd 2
fd 3
...
Global file table
file status flags
current file offset
inode pointer
file status flags
current file offset
inode pointer
Suppose we now call dup2(fd1, fd3)


high address
stack
heap
data
text
low address
Process control block (PCB)
sp (stack pointer)
pc (program counter)
fd table
fd 0
fd 1
fd 2
fd 3
...
Global file table
file status flags
current file offset
inode pointer
file status flags
current file offset
inode pointer
After call of dup2(fd1, fd3)