POPEN (Pipe Open)

1. If you need to run a program or any general shell command from C code, and either read its output or send data to its standard input, you can use **popen**.
2. Write a program that generates the "99 bottles of beer" song, and displays it in a pager like less.

```
int main(int argc, char** argv) {
    int i;
    FILE* fp;

    fp = popen("less", "w");
    for(i=99; i>0; i--) {
        fprintf(fp, "%d bottles of beer on the wall,\n", i);
        fprintf(fp, "    %d bottles of beer.\n", i);
        fprintf(fp, "If one of those bottles should happen to fall,\n");
        fprintf(fp, "    %d bottles of beer on the wall.\n\n", i-1);
    }
    pclose(fp);
    return 0;
}
```

3. Write a C program that displays the user with the most processes in the system, and how many processes he or she
has. Use a Shell command to find the data.

```
int main(int argc, char** argv) {
    int n;
    char u[64];
    FILE* fp;

    fp = popen("ps -ef | awk '{print $1}' | sort | uniq -c | sort -nr | head -n 1", "r");
    fscanf(fp, "%d %s", &n, u);
    pclose(fp);
    printf("%s: %d\n", u, n);
    return 0;
}
```

UNIX PROCESS FILE DESCRIPTOR MANIPULATION WITH DUP() AND DUP2()

1. The goal of this section is to see behind the scenes of input/output redirections, and eventually be able to write in C the equivalent of the command ps -ef | grep -E "^root" | awk '{print $2}'
2. Every process in a Unix-like operating system has a **file descriptor table** that is just a small table (array) that keeps track of all the **open files** (and other things like pipes or sockets) that the process is using.

   Example :

```
int main(int argc, char**argv) {
    int fd, myfifo, pa[2], pb[2];

    fd = open("a.txt", O_RDWR);
    myfifo = open("myfifo", O_RDONLY);
    pipe(pa);
    pipe(pb);

    return 0;
}
```

The file descriptor table will look as follows

Index	Value
0	Handle for reading from the console (standard input)
1	Handle for writing to the console (standard output)
2	Handle for writing to the console (standard error)
3	Handle for reading/writing to file a.txt. Variable `fd` has the value of the index, i.e. 3
4	Handle for reading from FIFO myfifo. Variable `myfifo` has the value of the index, i.e. 4
5	Handle for reading from the first PIPE created. Variable `pa[0]` has the value of the index, i.e. 5
6	Handle for writing to the first PIPE created. Variable `pa[1]` has the value of the index, i.e. 6
7	Handle for reading from the second PIPE created. Variable `pb[0]` has the value of the index, i.e. 7
8	Handle for writing to the second PIPE created. Variable `pb[1]` has the value of the index, i.e. 8

4. The file descriptor table can be manipulated using the system calls **dup()** and **dup2()**.
   a. **int dup(int oldfd)** makes a copy of the handle at index **oldfd** to a new entry, and returns the index of the entry just created. For example, adding line **int x = dup(myfifo)** to the program above, just before the **return**, will result in the following line being added to the file descriptor table, and variable **x** will have the value 9.

9	Handle for reading from FIFO myfifo.

   b. **int dup2(int oldfd, int newfd)** makes a copy of the handle at index **oldfd** to index **newfd**, overwriting the existing value (it also closes it silently before overwriting it). For example, adding line **dup2(p[0], 0)** to the program above, just before the **return**, will result in line 0 containing the handle for reading from the first PIPE created. Consequently, any reading from the standard input will use the first PIPE created because by using dup2(p[0], 0), you're telling the grep process: 'Don't read from the keyborad (stdin = 0) anymore; instead, read from the read end of PIPE'.

0	Handle for reading from the first PIPE created.

5. Let's implement in C a program that does the equivalent of running the command ps -ef | grep -E "^root" | awk '{print $2}', by running the three programs and transferring the data between them using pipes.

```
int main(int argc, char** argv) {
  int p2g[2], g2a[2];

  pipe(p2g); pipe(g2a);
  if(fork() == 0) {
    close(p2g[0]);close(g2a[0]);close(g2a[1]);
    dup2(p2g[1], 1);
    execlp("ps", "ps", "-ef", NULL);
    exit(1);
  }
  if(fork() == 0) {
    close(p2g[1]); close(g2a[0]);
    dup2(p2g[0], 0); dup2(g2a[1], 1);
    execlp("grep", "grep", "-E", "^root", NULL);
    exit(1);
  }
  if(fork() == 0) {
    close(p2g[0]);close(p2g[1]);close(g2a[1]);
    dup2(g2a[0], 0);
    execlp("awk", "awk", "{print $2}", NULL);
    exit(1);
  }

  close(p2g[0]); close(p2g[1]);
  close(g2a[0]); close(g2a[1]);

  wait(0); wait(0); wait(0);

  return 0;
}
```

6. How do you undo a **dup2** call? Use **dup** to make a copy of the entry that will be overwritten, and when you want to reset it, use again **dup2** with the copy.

```
int x = dup(1);
dup2(p[1], 1);
....
dup2(x, 1);
```

UNIX IPC SHARED MEMORY

1. UNIX IPC is a set of mechanisms (other than pipe and FIFO) for communicating among processes (IPC = Inter-Process Communication)
   a. **Semaphore** : synchronization mechanism
   b. **Message queues** : message-based communication
   c. **Shared memory** : communication through a common memory region

   Just like FIFOs, IPCs allow any two processes to communicate through them, but unlike FIFOs, IPCs are not files on disk. They are identified by a number that is unique in the system, this number is chosen by the developer and provided to all processes that need to communicate through that specific IPC.
2. IPC cleanup
   a. IPCs are persistent structures in the operating system
   b. The operating system imposes limits for the size and number of IPCs
   c. If not cleaup properly, the system will refuse to allow the creation of new IPCs
   d. You can see all existing IPCs using the command **ipcs**
   e. You can delete an IPC using command **ipcrm**, as long as you have permissions on that IPC

```
> ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x58df7f12 0          rares      644        100        0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

> ipcrm -M 0x58df7f12
```

3. Shared memory API
   a. Create or get a handle to an existing shared memory segment: **shmget**
   b. Attach - map the shared memory segment to a pointer in your process: **shmat**
   c. Detach - unmap the shared memory segment from you pointer: **shmdt**
   d. Control (configure, delete, etc.) the shared memory segment: **shmctl**