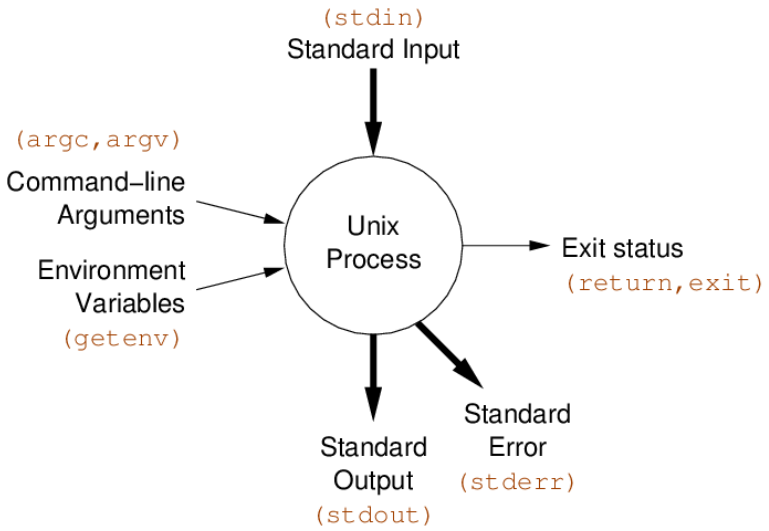
Unix Processes

A Unix process executes in this environment



Standard Streams

- Stream is is a sequence of bytes
- **stdio.h** define three streams
- **stdin** standard input stream
  scanf, getchar read from stdin
- **stdout** standard output stream
  printf, putchar write to stdout
- **stderr** standard error stream
  by convention used for error messages

I/O direction

- If program run from terminal: stdin, stdout, stderr connected to terminal.
- Unix shells allow you to re-direct stdin, stdout and stderr.
- Run **a.out** with stdin coming from file **data.txt**

```
$ ./a.out <data.txt
```

- Run **a.out** with stdout going to (overwriting) file **output.txt**

```
$ ./a.out >output.txt
```

- Run **a.out** with stdout appended to file **output.txt**

```
$ ./a.out >>output.txt
```

- Run **a.out** with stderr going to file **errors.txt**

```
$ ./a.out 2>errors.txt
```

Unix Pipes

- Unix shells allow you to connect stdout of one program to stdin of another program
- Run **a.out** with stdout going to stdin of **wc**

```
$ ./a.out | wc -l
```

- wc counts chars, words and lines in its stdin
- Unix other useful programs (filters) designed to be run like wc
  E.g.: grep, cut, sort, uniq
- covered in detail in COMP2041

Using stderr for error Messages

- **fprintf** allows you to specify stream to print to
- For example:

```
fprintf(stderr, "error: can not open %s\n", argv[1]);
```

- *printf* actually just calls *fprintf* specifying stdout

```
fprintf(stdout, ...);
```

- Best if error messages written to stderr, so users sees them even if stdout is directed to a file.
- Common for stderr to be redirected separately to a log file for system programs.

Accessing Files

- **FILE *f = fopen(char *filename, char *mode)**
  Create a stream to read from or write to file
  returns NULL if open fails
- **int fclose(FILE *f)**
  finish operations on a file
  fclose called automatically on program exit
  Beware: some output may be cached until **fclose** called
- **int fgetc(FILE *f)**
  Read a single character from a stream
  returns EOF if no character available
- **int fputc(int, FILE *f)**
  Writes a single character to a stream
- **int fscanf(FILE *f, ...)**
  scanf from stream; returns number of values read
- **int fprintf(FILE *f, ...)**
  printf to specified stream

Opening a File

```
FILE *fp = fopen("filename.txt", "w");
```

- `fopen` - opens a file
- `parameter 1` - the name of the file to be opened
- `parameter 2` - the mode in which to open the file
- `return value` - a pointer to the file which has been opened. This pointer is then used to reference the opened file for operations such as reading, writing, and closing of the file. Return value will be **NULL** if file can not be opened.

fopen *mode* parameter

- `"r"`
  - Opens an existing text file for reading purpose.
- `"w"`
  - Opens a text file for writing.
  - If it does not exist, then a new file is created.
  - Starts writing from the beginning of the file.
- `"a"`
  - Opens a text file for writing in appending mode.
  - If it does not exist, then a new file is created.
  - Starts writing at the end of the existing file contents.
- `"r+" "w+" "a+"`
  - Opens a file for both reading and writing.
  - `w+` truncates the file length to zero if it exists, while `a+` starts reading from the start of the file and writing at the end of the existing file contents.

Writing to a File

```
fputs("the text I want to write in the file\n", fp);
```

or

```
fprintf(fp, "the text I want to write in the file\n");
```

or

```
int c = '!';
fputc(c, fp);
```

Reading from a File

```
char line[MAX_LINE_LENGTH];
if (fgets(line, MAX_LINE_LENGTH, fp) != NULL) {
    printf("read line\n");
} else {
    printf("Could not read a line\n");
}
```

or

```
int c;
c = fgetc(fp);
if (c != EOF) {
    printf("read a '%d'\n", c);
} else {
    printf("Could not read a character\n");
}
```