Inter-Process Communication

Tran Giang Son, tran-giang.son@usth.edu.vn

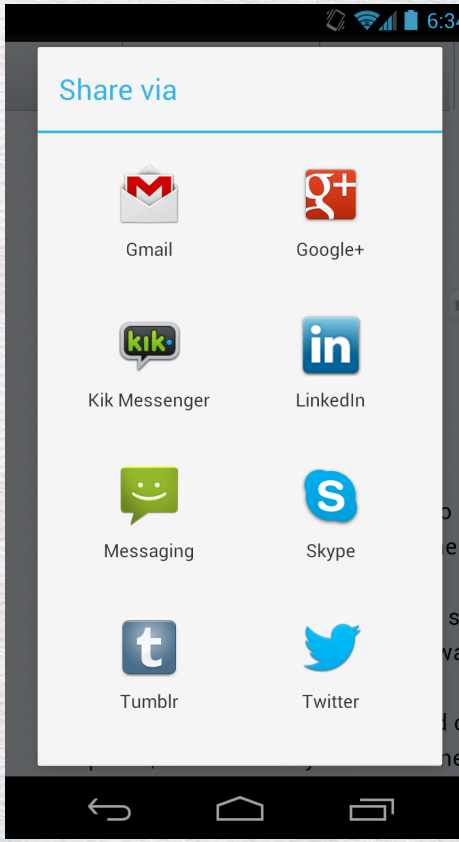
ICT Department, USTH

Introduction

What?

- A mechanism allowing processes to share data
- Examples
  - UNIX pipes
  - Android's «Share»

```
$ ps | grep zsh
437 ttys000    0:00.46 -zsh
```



Why?

- Share information
- Cooperation between processes
  - UNIX philosophy
- Modularity
- Convenience

How?

1. Signal
2. Shared memory
3. File
4. Pipe
5. Socket

Signal

(1) Signal

- What?
  - Software generated interrupts sent to a process when an event happens
    - Pause: `SIGSTOP`
    - Continue: `SIGCONT`
    - Termination: `SIGTERM`, `SIGKILL`
    - Crashed: `SIGSEV`
  - Asynchronous
  - Limited (31 signals only)
- Why?
  - Standard in UNIX
  - Early form of IPC

(1) Signal

- How?
  - Implement signal handler

```
void handler(int sig) {}
```

  - Register the signal handler

```
void (*signal(int sig, void (*func)(int)))(int);
```

(1) Signal

```c
void handler(int signal_num) {
    printf("Signal %d => ", signal_num);
    switch (signal_num) {
    case SIGTSTP:
        printf("pause\n"); break;
    case SIGINT:
    case SIGTERM:
        printf("Terminated\n");
        exit(0);
        break;
    }
}
```

```c
int main(void) {
    // ctrl z
    signal(SIGTSTP, handler);

    // ctrl c or killed
    signal(SIGINT, handler);
    signal(SIGTERM, handler);
    while (1) {
        sleep(1);
        printf(".\n");
    }
    printf("end");
    return 0;
}
```

Practical Work 5: mini shell with signal handler

- Copy your practical work 4 to a new file
  - Name it « 04.practical.work.shell.signal.c »
  - Add SIGTSTP signal handler to your shell
  - If your shell is paused with Ctrl-Z:
    - Send to your child SIGINT or SIGTERM
    - Terminate your shell
- Push your C program to corresponding forked Github repository

Shared Memory

(2) Shared memory

- What?
  - A memory region that can be accessed by different local processes
  - Permission support
- Why?
  - Fast
  - Large
  - Structured

(2) Shared memory

- Create shared memory segment

```
segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);
```

- Other processes attach to it

```
shared_memory = (char *) shmat(id, NULL, 0);
```

- They then read/write as usual

```
sprintf(shared_memory, "Writing to shared memory");
```

- After using it, detach

```
shmdt(shared_memory);
```

File

(3) File

- Open/create a file

```
FILE *fopen(const char *path, const char *mode);
```

- Read content from an opened file

```
size_t fread(void *buffer, size_t size,
    size_t nitems, FILE *restrict stream);
```

- Write content to an opened file

```
size_t fwrite(const void *ptr, size_t size,
    size_t nitems, FILE *stream);
```

- Close an opened file

```
int fclose(FILE *stream);
```

(3) File

- FILE* is a wrapper of file descriptor (int)
- A file is addressed through a descriptor
  - 0, 1 abd 2 correspond to standard input, standard output, and standard error
  - The file descriptor number is returned by the open system call

```
int open(const char *pathname, int flags);
int creat(const char *pathname, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, void *buf, size_t count);
int close(int fd)
```

Practical Work 6: mini shell with command history

- Copy your practical work 5 to a new file
  - Name it « 06.practical.work.shell.history.c »
  - Add command logger to your shell
    - Add a new line to “command.log” file for each command user inputs
  - Before quitting
    - Print all history from the file “command.log”
- Push your C program to corresponding forked Github repository

Pipe

(4) Pipe

- What?
  - FIFO mechanism to pass data
  - Output of one is input of another
  - Unidirectional
- Why?
  - Simple to use for simple communication
    - no `socket()`, `bind()`, `listen()`, `connect()`, `accept()`
  - Just like sockets...

(4) Pipe

- How?
  - `int mypipe[2];`
  - `pipe(mypipe);`
  - `mypipe[0]` is the read end
  - `mypipe[1]` is the write end
  - Data written to `mypipe[1]` (using `write()`) can be read from `mypipe[0]` (using `read()`)
  - Use **before** `fork()`

(4) Pipe

```
void doexec(void) {
    int pipefds[2];
    pipe (pipefds);
    switch (fork ()) {
        case -1: perror ("fork"); exit (1);
        case 0:
            dup2 (pipefds[1], 1);
            close(pipefds[0]);
            close(pipefds[1]);
            execvp(...);
            break;
        default:
            dup2(pipefds[0], 0);
            close(pipefds[0]);
            close(pipefds[1]);
        break;
    }
}
```

Practical Work 7: mini shell with pipe

- Copy your practical work 6 to a new file
  - Name it « 07.practical.work.shell.pipe.c »
  - Add support for IO redirection into your shell
    - e.g. `ps aux > process_list.txt`
  - Add support for pipe into your shell
    - e.g. `ps aux | grep ssh > ssh_process_list.txt`
- Push your C program to corresponding forked Github repository