Assignment One: System Calls in C Programming

CSCI3150 Introduction to Operating Systems, Fall 2024

Shaofeng Wu
wsf123@link.cuhk.edu.hk
Sep 19, 2024

Overview (1/1)

- **Tutorial last week**
  - Behavior of a bash shell
    - Built-in commands
    - External commands
    - Pipe
  - Code walk of asg1

All examples used in this tutorial: https://github.com/henryhxu/CSCI3150/tree/2024-Fall/tutorial/T03

- **Tutorial this week**
  - System calls in C programming
    - fork & exec
    - dup
    - pipe

Fork & Exec (1/3)

example1.c

```c
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    pid_t pid = 0;
    int status;
    int wait_return;

    for(int i=0; i<10; i++){
        /* create 10 child processes in parent process */
        if(pid == 0){
            pid = fork();
            if(pid < 0){
                printf("fork() failed\n");
                exit(-1);
            }
        }
    }

    if(pid > 0){
        printf("Hello from child %d\n",pid);
        exit(0);
    }

    /* wait for all child to terminate by checking the return value of wait */
    while((wait_return = wait(&status)) > 0){;}

    printf("All child terminated\n");

    return 0;
}
```

- **Fork & wait**
  - Function prototype
    - pid_t fork(void);
    - pid_t wait(int *wstatus);
  - fork() is used to create child processes
  - wait() is used to let the parent wait for one of its child to terminate
  - The child process created with fork() is an exact duplicate of the parent process except for several points(process id, memory lock, …)

Fork & Exec (2/3)

example2.c

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    char *file_name = "./my_env";
    char *argv[32] = {"my_env", NULL};
    char *envp[2] = { "ABC=1", NULL};

    /* execute the command */

    // execl("/bin/my_env", argv[0], NULL);
    // execlp(file_name, argv[0], NULL);
    // execle("/bin/my_env", argv[0], NULL, envp);

    // execv("/bin/my_env", argv);
    // execvp(file_name, argv);
    execvpe(file_name, argv, envp);

    printf("We just arrive at a line of code after
exec()\n");

    return 0;
}
```

- **The exec() system call family:**
  - int execl(const char *pathname, const char *arg, ... /*, (char *) NULL */);
  - int execlp(const char *file, const char *arg, ... /*, (char *) NULL */);
  - int execle(const char *pathname, const char *arg, ... /*, (char *) NULL, char *const envp[] */);
  - int execv(const char *pathname, char *const argv[]);
  - int execvp(const char *file, char *const argv[]);
  - int execvpe(const char *file, char *const argv[], char *const envp[]);

Fork & Exec (3/3)

example3.c

```c
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(){

    pid_t pid;
    pid = fork();
    if(pid == -1){
        printf("Fork error\n");
        exit(-1);
    }
    else if(pid == 0){
        /* parent */
        printf("parent is doing something\n");
        sleep(5);
        printf("parent finished its task");
    }
    else{
        /* child */
        execlp("ls","ls","-a",NULL);
    }
    return 0;
}
```

- **Combining fork() and exec()?**
  - Child process will have its own image, no longer a duplicate of its parent

Dup (1/1)

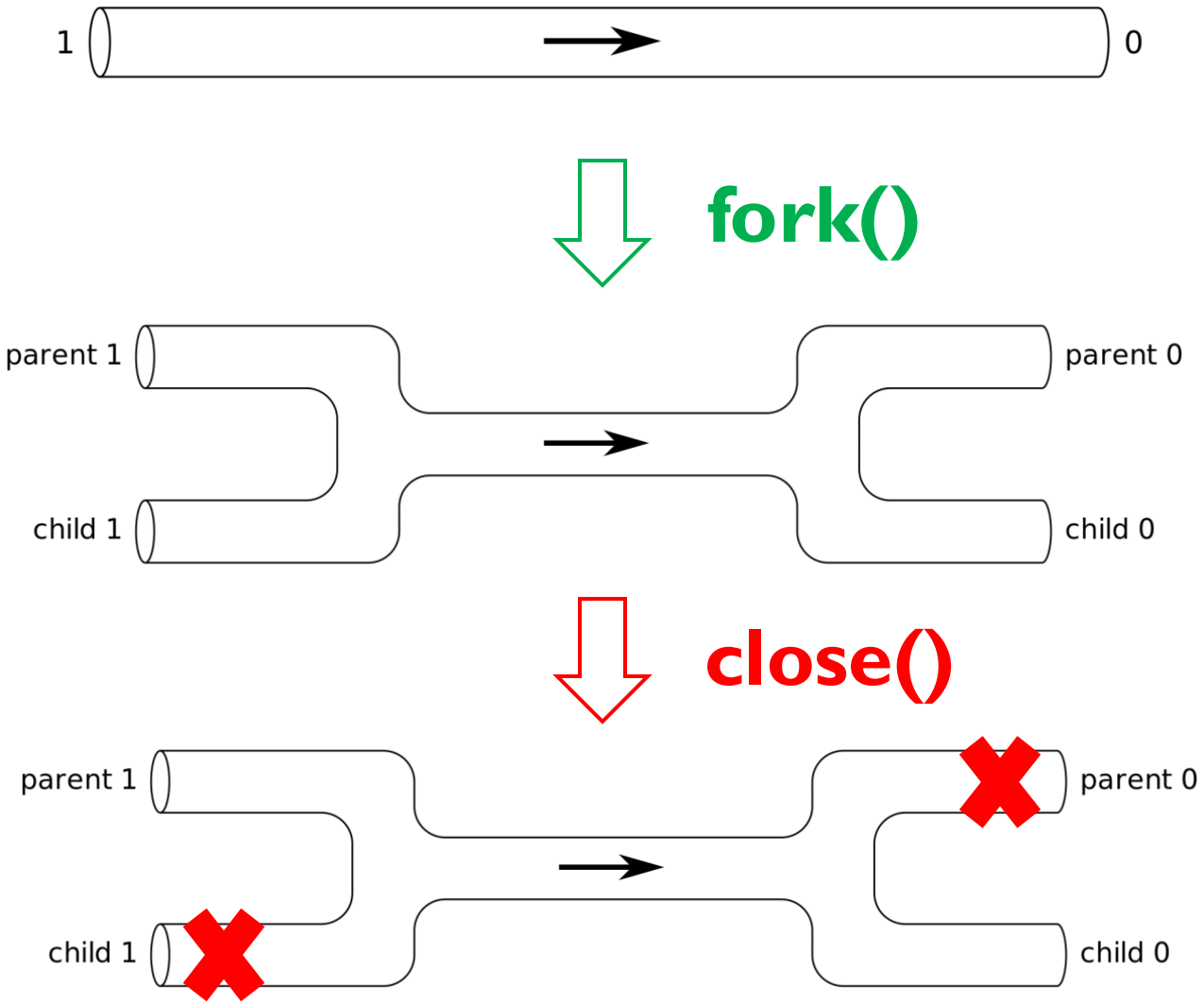
example4.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include <fcntl.h>

int main(){

    char *file_path = "./test0.txt";
    int fd = open(file_path, O_RDWR);

    /* redirect standard output to the file */
    dup2(fd,STDOUT_FILENO);

    printf("------dup test ends------\n");

    close(fd);
    return 0;
}
```

- **dup**
  - Function prototype
    - int dup(int oldfd);
    - int dup2(int oldfd, int newfd);
  - Allocates a new file descriptor that refers to the same open file description as the descriptor oldfd. For dup2, it specifically uses newfd instead of allocating a new file descriptor.
  - On success, these system calls return the new file descriptor. On error, -1 is returned.
  - Question: How to restore standard output after the dup2()?

Pipe (1/4)

- **Pipe**
  - System call for pipe creation
    - int pipe(int pipefd[2])
  - Pipe characteristics
    - Unidirectional
    - "Half-duplex"
    - Byte-stream
      - How does the reader know EOF?
    - IPC between related processes(fork())
  - Notes for correctly using pipe
    - Always close the un-used read/write end of a pipe



Pipe (2/4)

example5.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>

int main(){

    int pipe_fd[2];
    int bytes_read;
    int total_bytes_read = 0;
    char readbuf[1024];

    /* create a pipe */
    int ret = pipe(pipe_fd);
    if(ret < 0){
        printf("create pipe error\n");
        return 0;
    }

    /* write something into the pipe */
    write(pipe_fd[1], "------\npipe test1\n------\n", 25);
    write(pipe_fd[1], "pipe test2\n------\n", 18);

    /* close the write end, EOF will be added into the pipe */
    close(pipe_fd[1]);

    /* read from the pipe into a buffer until there is no content in the pipe */
    while ((bytes_read = read(pipe_fd[0], &readbuf[total_bytes_read], 1024-1-
total_bytes_read)) > 0){
        total_bytes_read += bytes_read;
    }

    /* print the content in the buffer */
    printf("total read %d\n",total_bytes_read);
    readbuf[total_bytes_read]='\0';
    printf("%s\n", readbuf);

    /* close read end of pipe */
    close(pipe_fd[0]);

}
```

- **Basic usage**
  - Function prototype
    - int pipe(int pipefd[2]);
  - Pipe() creates a pipe, a unidirectional data channel that can be used for inter-process communication.
    - pipefd[0]: read end of the pipe
    - pipefd[1]: write end of the pipe
    - On success, zero is returned. On error, -1 is returned

Pipe (3/4)

example6.c

```c
#include <assert.h>
#include <signal.h>
#include <stdbool.h>
#include <stdio.h> /* perror */
#include <stdlib.h>
#include <sys/wait.h> /* wait, sleep */
#include <unistd.h> /* fork, write */

int main() {

    pid_t pid;
    int wait_return, status;
    int pipe_fd[2];
    int bytes_read;
    int total_bytes_read = 0;
    char readbuf[1024];

    /* create a pipe */
    int ret = pipe(pipe_fd);
    if(ret < 0){
        perror("create pipe error\n");
        return 0;
    }
    pid = fork();
    if(pid == -1){
        perror("fork");
        assert(false);
    }
    else if(pid == 0){
        /* close un-unsed write end of the pipe */
        close(pipe_fd[1]);
        /* read from parent process into the buffer*/
        while ((bytes_read = read(pipe_fd[0], &readbuf[total_bytes_read], 1024-1-
total_bytes_read)) > 0){
            total_bytes_read += bytes_read;
        }
        /* print the content in the buffer */
        readbuf[total_bytes_read]='\0';
        printf("[Message from parent] %s\n", readbuf);
        exit(0);
    }
    else{
        /* close un-used read end of the pipe */
        close(pipe_fd[0]);

        // sleep(5);

        /* write something into the pipe */
        write(pipe_fd[1], "Hello child process", 20);

        /* finished writing, close write end so child sees EOF */
        close(pipe_fd[1]);

        /* wait for the child process to terminate */
        while((wait_return = wait(&status)) > 0);
    }

    return 0;
}
```

- **IPC**
  - Pipe is one of the things that will be inherited by child processes created with fork()
  - Inter process communication(IPC) can be achieved with pipe
    - pipefd[0]: read end of the pipe
    - pipefd[1]: write end of the pipe
    - On success, zero is returned. On error, -1 is returned

Pipe (4/4)

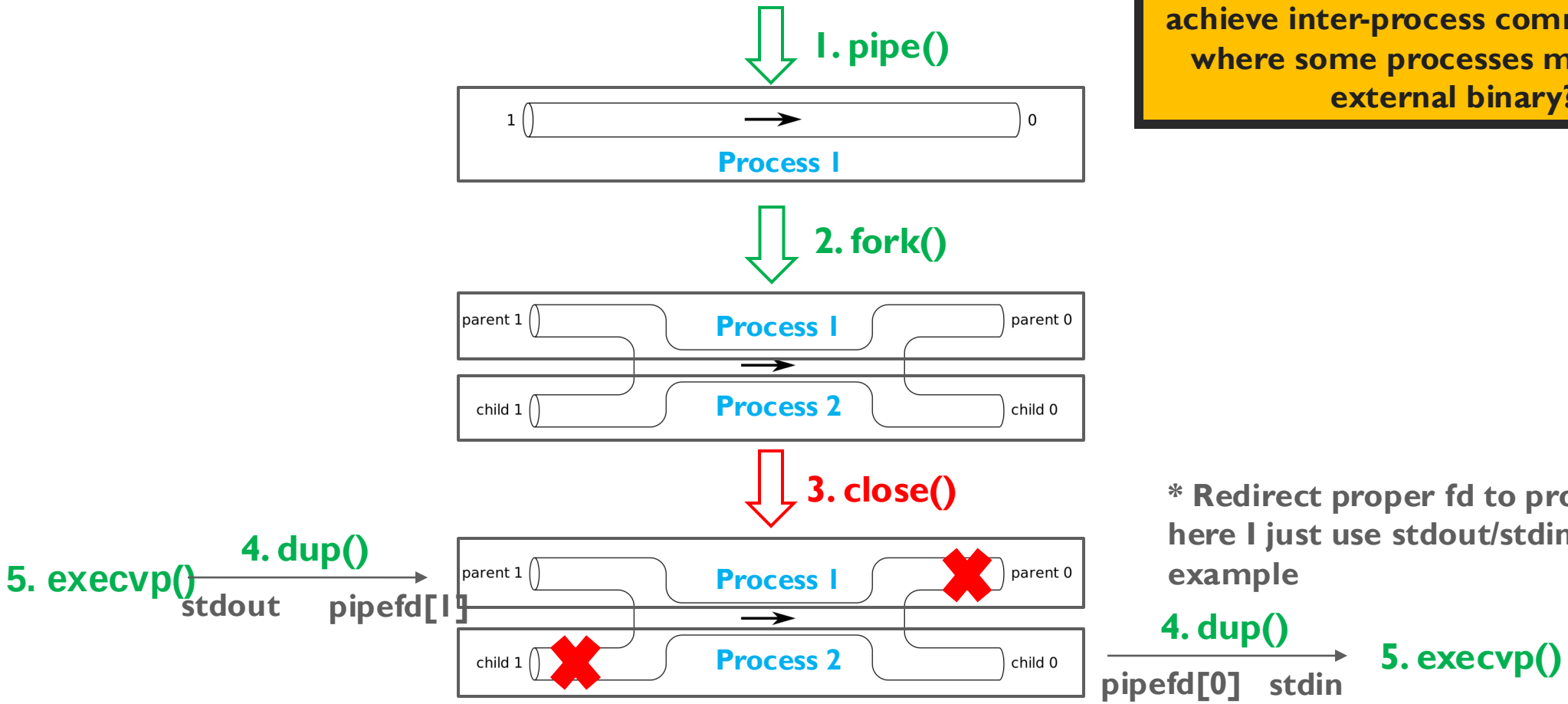
example7.c

```c
#include <assert.h>
#include <signal.h>
#include <stdio.h> /* perror */
#include <stdlib.h>
#include <sys/wait.h> /* wait, sleep */
#include <unistd.h> /* fork, write */

void signal_handler(int sig) {
    char s1[] = "SIGPIPE captured\n";
    char s2[] = "Unknown Signal\n";
    if (sig == SIGPIPE){
        write(STDOUT_FILENO, s1, sizeof(s1));
    }
    else{
        write(STDOUT_FILENO, s2, sizeof(s2));
    }
}

int main() {
    pid_t pid = getpid();
    int pipe_fd[2];

    /* create a pipe */
    int ret = pipe(pipe_fd);
    if(ret < 0){
        printf("create pipe error\n");
        return 0;
    }

    /* create a signal handler to capture signal pipe */
    signal(SIGPIPE, signal_handler);


    /* close all read end of the pipe */
    close(pipe_fd[0]);

    /* writer something to the pipe */
    write(pipe_fd[1], "test content", 12);

    return 0;
}
```

- **Pipe w/ read/write**
  - example6.c
    - If all file descriptors referring to the write end of a pipe have been closed
      - Read from pipe will see end-of-file(EOF)
    - If a process attempts to read from an empty pipe
      - Read will block until data is available
  - example7.c
    - If all file descriptors referring to the read end of a pipe have been closed
      - Write to the pipe will generate SIGPIPE for the process
    - If a process attempts to write to a full pipe
      - Write blocks until sufficient data has been read from the pipe to allow the write to complete

Summary (1/1)

How to use the above systems calls to achieve inter-process communication, where some processes may run an external binary?



* Redirect proper fd to proper fd, here I just use stdout/stdin as an example

Q & A