

interpreting

`execp()` → no need to specify the path

`exec()` → have to specify the path

# ls -l -a  
↓  
command-line arguments

`execp ("ls", "ls", NULL)`  
↓    ↓  
command, command  
no command arguments

`exec( . . . )` same

# ls > file

# ls & ↑  
↳ shell internal command

```
// simple shell example using fork() and execvp()
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>

main()
{
pid_t k;
char buf[100];
int status;
int len;
while(1) {
    // print prompt
    fprintf(stdout,"%d$ ",getpid());
    // read command from stdin
    fgets(buf, 100, stdin);
    len = strlen(buf);
    if(len == 1)
        continue;
    buf[len-1] = '\0';
    k = fork();
    if (k==0) {
        // child code
        if(execvp(buf,buf,NULL) == -1) // if execution failed, terminate
            "main2b.c" 40L, 663C
    }
}
}
```

same as printf

get string, fix length

```
// simple shell example using fork() and execvp()
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
pid_t k;
char buf[100];
int status;
int len;

while(1) {
    fprintf(stdout, "[%d] ", getpid());
    fscanf(stdin, "%s", buf);
    Bad if input more
    than [100] (buffer size)
    k = fork();
    if (k==0) {
        // child code
        execvp(buf, buf, NULL);
    }
    else {
        // parent code
        waitpid(k, &status, 0);
    }
}
~  
~  
"main2.c" 30L, 420C written
```

in child process, command longer than [10]

execvp() run fails, child process  
go back to while(), print the prompt,  
which is child process PID,

parent still hangs on, waiting for  
the child process to terminate

```
[26937] ls
[26946] ls
a.out forever main1.c main2a.c main2b.c main2.c main3.c mysh
[26946] pwd
/u/data/u3/park/pub/cs240/lab5/v5
[26946] date
Mon Jul 17 10:17:11 EDT 2017
[26946]
pwd
/u/data/u3/park/pub/cs240/lab5/v5
[26946] host
Usage: host [-acdlritwv] [-c class] [-N ndots] [-t type] [-w time]
           [-R number] [-m flag] hostname [server]
      -a is equivalent to -v -t ANY
      -c specifies query class for non-IN data
      -C compares SOA records on authoritative nameservers
      -d is equivalent to -v
      -l lists all hosts in a domain, using AXFR
      -i IP6.INT reverse lookups
      -N changes the number of dots allowed before root lookup is done
      -r disables recursive processing
      -R specifies number of retries for UDP packets
      -s a SERVFAIL response should stop query
      -t specifies the query type
      -T enables TCP/IP mode
      -v enables verbose output
      -w specifies to wait forever for a reply
      -W specifies how long to wait for a reply
      -4 use IPv4 query transport only
      -6 use IPv6 query transport only
      -m set memory debugging flag (trace|record|usage)
      -V print version number, and exit
[26946]
```

the child process  
own shell  
now,  
if we use  
ctrl-c,  
we will kill  
the child  
process,  
because the  
signal will  
send to  
the owner.

```

graph LR
    P["27069  
parent"] --- C["27071  
child"]

```

The diagram illustrates a parent-child relationship between two processes. A box labeled "27069 parent" is connected by a line to another box labeled "27071 child". The word "parent" is written below the ID in the box, and "child" is written below the ID in the box.

```
// simple shell example using fork() and execvp()
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <string.h>

main()
{
    pid_t k;
    char buf[100];
    int status;
    int len;
    while(1) {
        // print prompt
        fprintf(stdout,"%d$ ",getpid());
        // read command from stdin
        fgets(buf, 100, stdin);
        len = strlen(buf);
        if(len == 1) → check           // only return key pressed
            continue;
        buf[len-1] = '\0'; → replace '\n' by '\0'
        k = fork();
        if (k==0) {
            // child code
            if(execvp(buf,buf,NULL) == -1)      // if execution failed, terminate
    "main2b.c" 40L, 663C
```

pod1-Lx-parkins:~/Scratches/K/

```
fgets(buf, 100, stdin);
len = strlen(buf);
if(len == 1)
    continue;
buf[len-1] = '\0';

k = fork();
if (k==0) {
    // child code
    execvp(buf,buf,NULL);
}
else {
    // parent code
    waitpid(k, &status, 0);
}

"main2a.c" 36L, 510C written
pod1-1 38 % gcc main2a.c
pod1-1 39 % ./a.out
[27103]$ ls
a.out forever main1.c main2a.c main2b.c main2.c main3.c mysh
[27103]$ pwd
/u/data/u3/park/pub/cs240/lab5/v5
[27103]$ date
Mon Jul 17 10:28:52 EDT 2017
[27103]$ ps
  PID TTY          TIME CMD
26390 pts/4    00:00:00 tcsh
27103 pts/4    00:00:00 a.out
27109 pts/4    00:00:00 ps
[27103]$
```

Bug  
not check return value

```
#include <string.h>

main()
{
    pid_t k;
    char buf[100];
    int status;
    int len;
    while(1) {
        // print prompt
        fprintf(stdout, "%d> ", getpid());
        // read command from stdin
        fgets(buf, 100, stdin);
        len = strlen(buf);
        if(len == 1)
            continue; // only return key pressed
        buf[len-1] = '\0';
        k = fork();
        if (k==0) {
            // child code
            if(execlp(buf,buf,NULL) == -1)
                exit(1); // if execution failed, terminate child
        }
        else {
            // parent code
            waitpid(k, &status, 0);
        }
    }
}
```

if execvp fail, return -1.

check return of execlp

A photograph of a computer monitor showing a terminal window with a command-line interface. The screen displays the following text:

```
sub1:AV5-296-CTR 13 8 33 Rows 105 Cols CAP NUM
```

Handwritten red text is overlaid on the right side of the image, explaining the state of processes:

↑ execvp fail, child process  
exit(1),  
parent process  
waiting  
for  
child,

The handwritten text corresponds to the state shown in the terminal window: the child process has failed (exit code 1) and the parent process is waiting for it.

when child process exit with 1, parent  
got noticed , go back to while(1)  
continue

↑  
VS ((ab b))

(work) `fprintf(stderr, ,)` unbuffered

A screenshot of a terminal window titled "pod1-Linuxfundamentals". The code in the window is:

```
int main(void) {
    char a[5];
    int *b;
    a[0] = 'h';
    a[1] = 'i';
    a[2] = '\0';
    printf("%s", a);
    *b = 10;
}
```

The terminal output shows:

```
pod1-1 58 % 1
main.c main.c
pod1-1 59 % gcc main.c

```

Handwritten annotations in red:

- An arrow points from the word "unbuffered" to the line "printf(stderr, ,)".
- A large red circle encloses the entire terminal window.
- Red text "even `fprintf(stdout, )`" is written above the terminal window.
- Red text "same, Doh!" is written next to the circle.
- An arrow points from the word "seg fault" to the terminal output.
- An arrow points from the word "fflush()" to the line "fflush(stdout)" in the handwritten notes below.
- Red text "But don't print 'Hi'" is written below the terminal output.

`fflush(stdout)`

`fflush(stderr)`

Inside printf()

use some space as buffer  
in RAM

to temporarily store writing

---

Writing/Reading from disk is super slow, CPU have to wait I/O

```
printf("hi");  
C printf("hello");
```

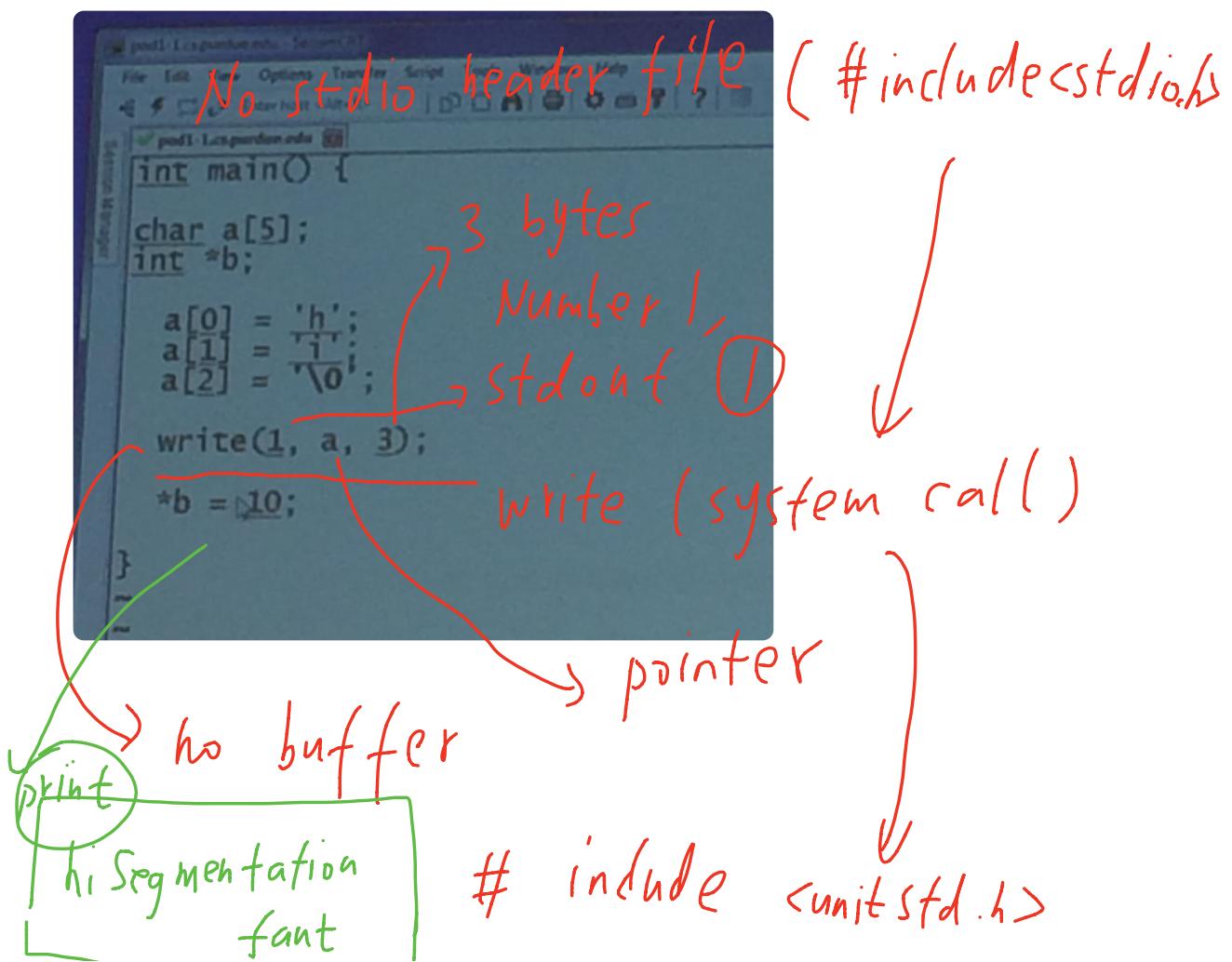
Combine these function together, and then do I/O.

But, when Seg fault, the buffer not

cleaked. divid by zero, . . .

exit() will clean the buffer.

fflush(stdout) will force to print  
the buffer to screen.



A screenshot of a terminal window titled "pod1-1: main1.c". The window shows the following code:

```
int *b;
a[0] = 'h';
a[1] = '\n';
a[2] = '\0';
write(1, a, 2);
*b = 10;
}
~"main1.c" 16L, 132C written
pod1-1 40 % !gc
gcc main1.c
pod1-1 41 % ./a.out
Segmentation fault
pod1-1 42 %
```

The code defines an array `a` with three elements: `a[0]` is set to 'h', `a[1]` is set to '\n', and `a[2]` is set to '\0'. The `write` function is called with arguments 1, `a`, and 2. The variable `*b` is initialized to 10. The terminal output shows the compilation of `main1.c` and the execution of the resulting binary, which results in a segmentation fault.

Handwritten annotations in red:

- "can also write 2 bytes" is written above the `write` call.
- "Same Result" is written below the terminal output.
- A large red oval encloses the `write(1, a, 1);` line and the text "it only print 'h'" below it.
- A red arrow points from the right side of the terminal window towards the handwritten annotations.

can also write 2 bytes  
Same Result

write(1, a, 1);  
it only print 'h'