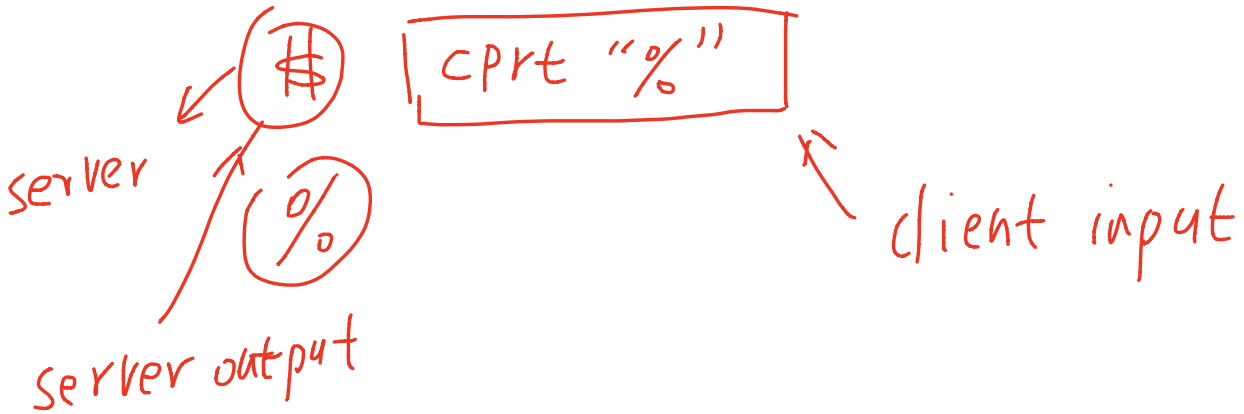
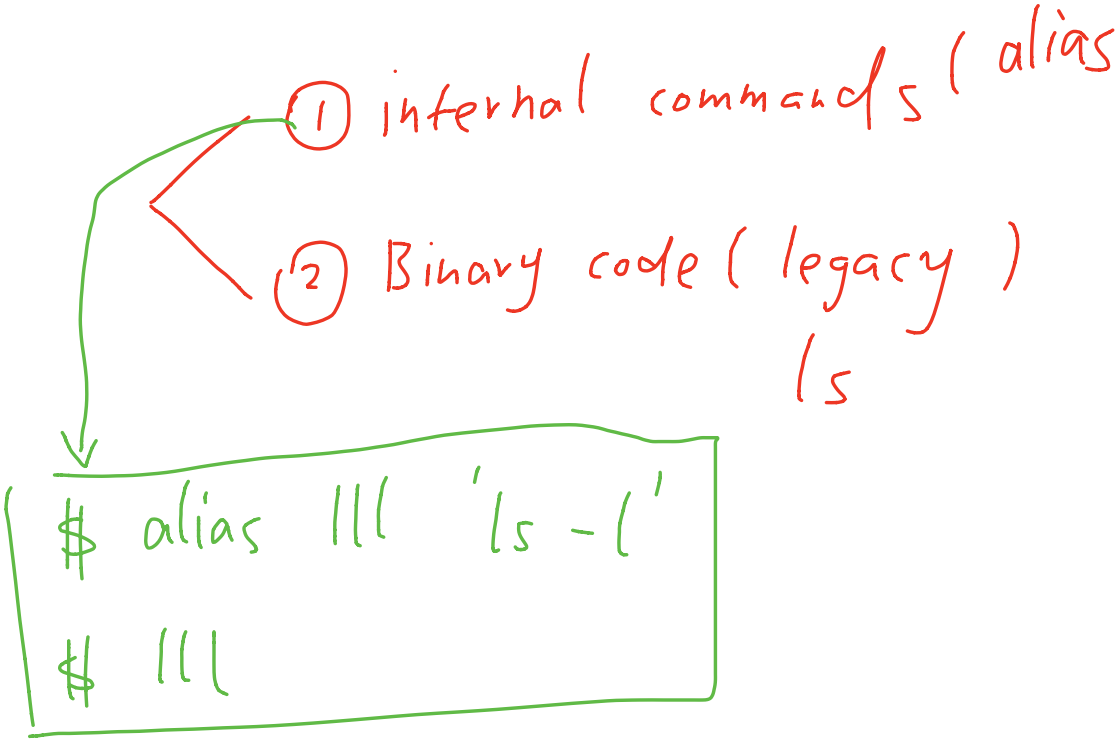




interpreting

execlp( ) → no need to spirify the path

execl( ) → have to spirify the path

```
$ ls -l -a
```

command-line arguments

execlp("ls", "ls", NULL)

command, command

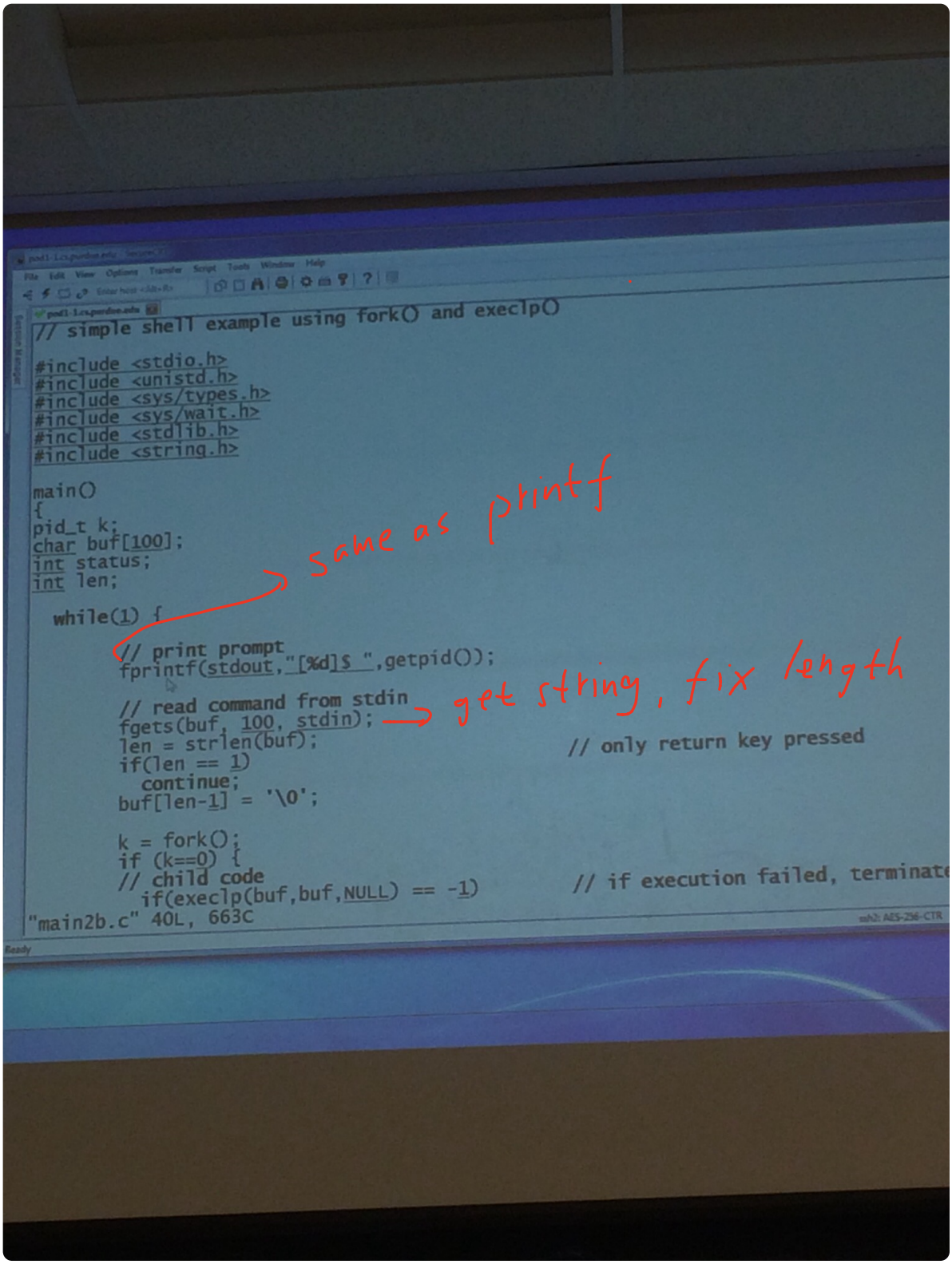
no command arguments

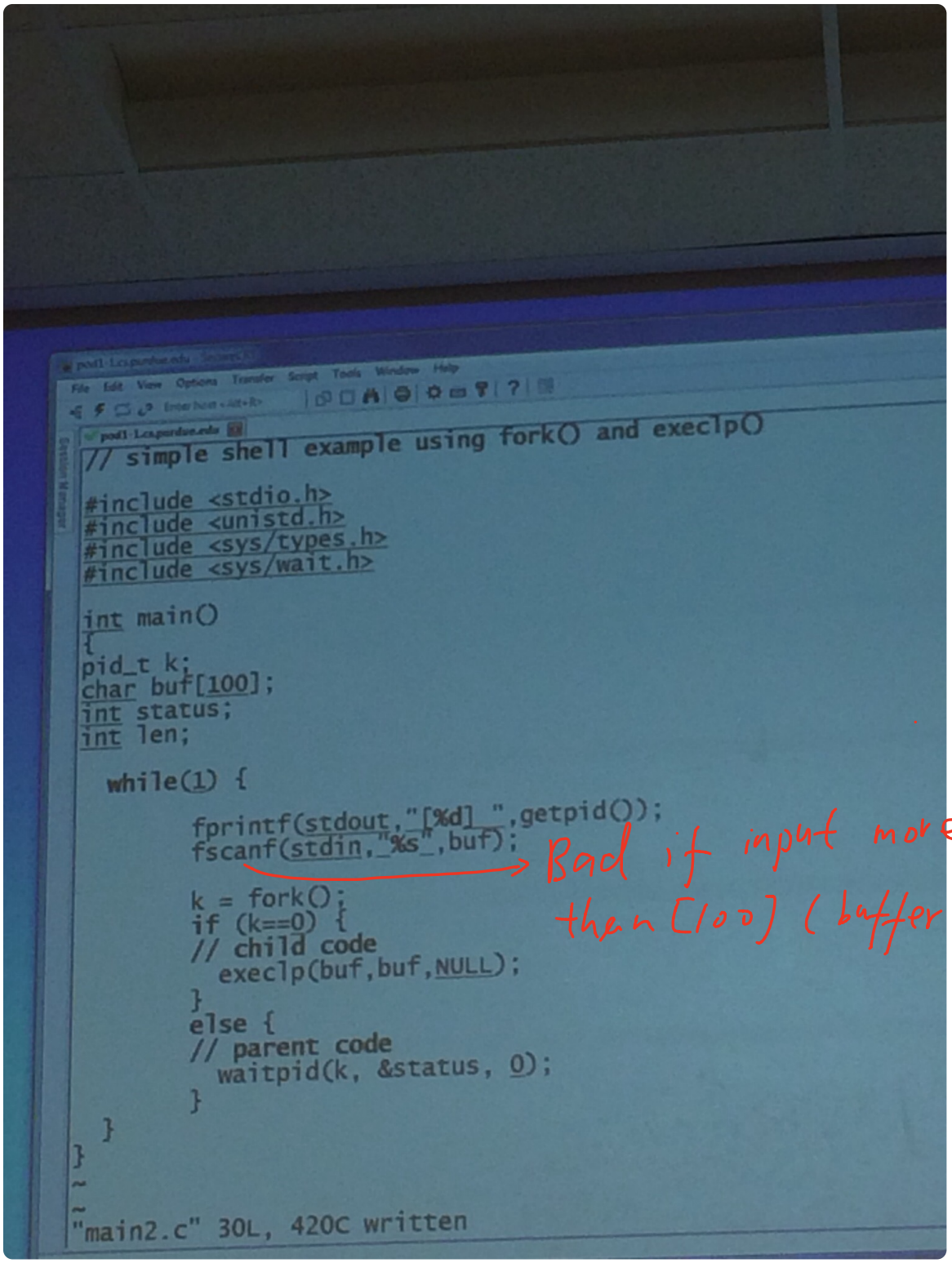
execl( ) same

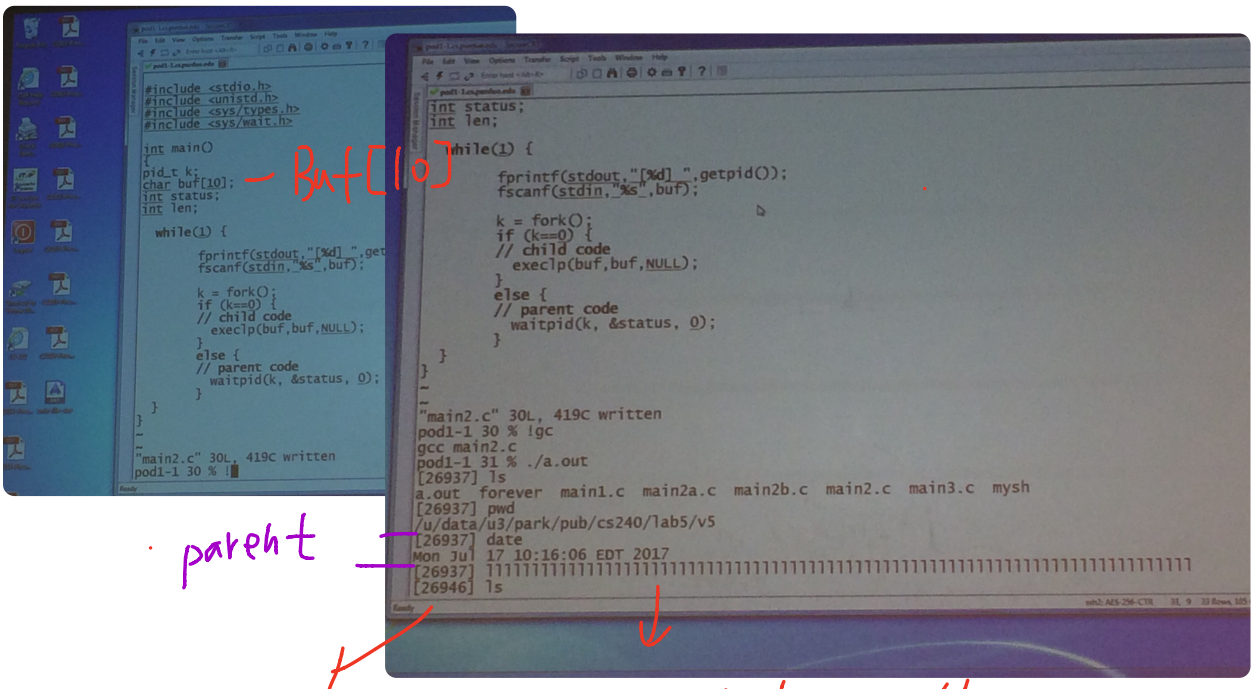
```
$ ls > file
```

shell internal command

```
$ ls &
```


// simple shell example using fork() and execlp()
#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(execlp(buf,buf,NULL) == -1)
"main2b.c" 40L, 663C
same as printf
get string, fix length
// only return key pressed
// if execution failed, terminate


// simple shell example using fork() and execlp()
#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);
k = fork();
if (k==0) {
// child code
execlp(buf,buf,NULL);
}
else {
// parent code
waitpid(k, &status, 0);
}
}
}
"main2.c" 30L, 420C written
Bad if input more than [100] (buffer size)



Buf[10]

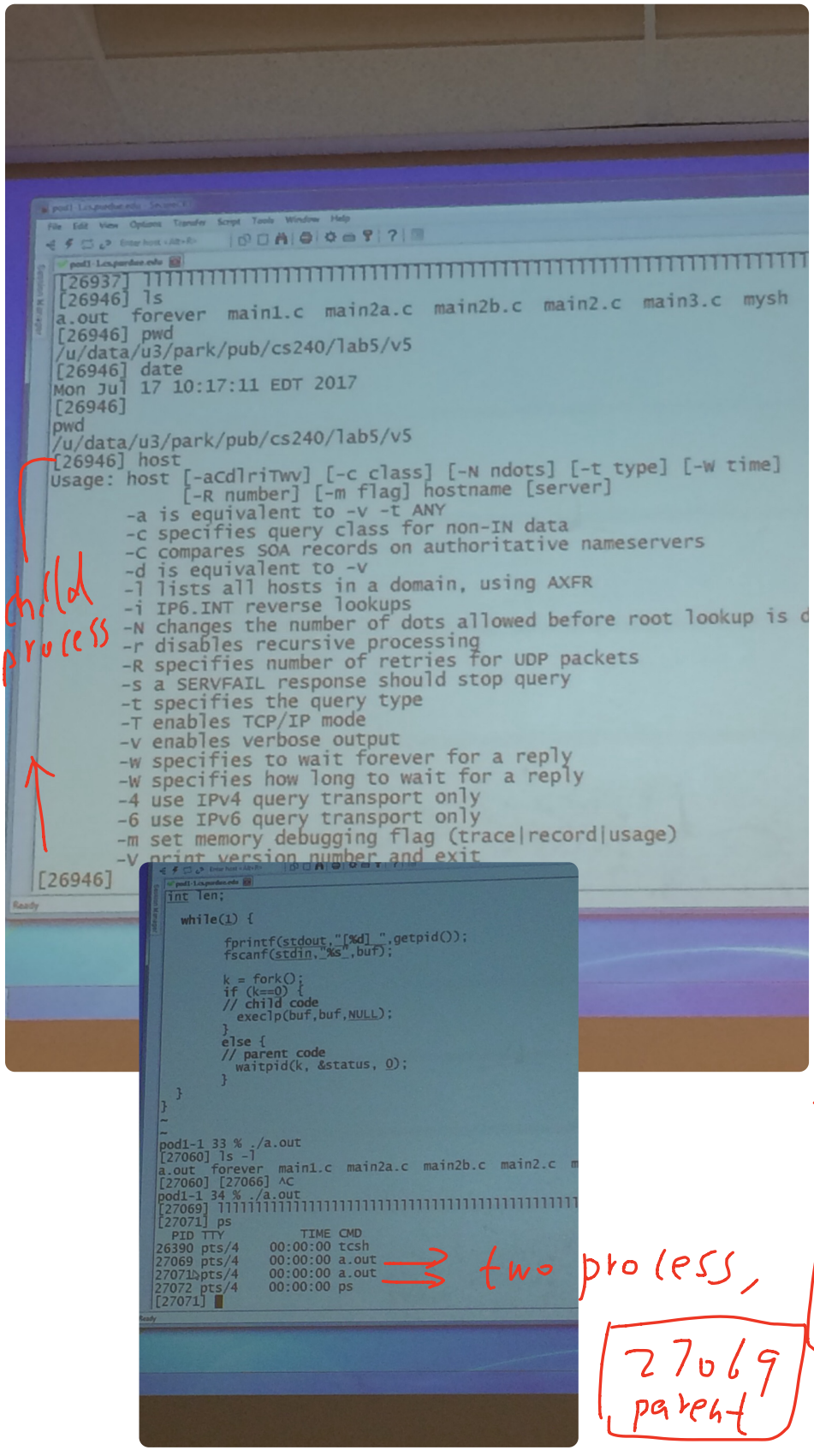
parent

in child process,

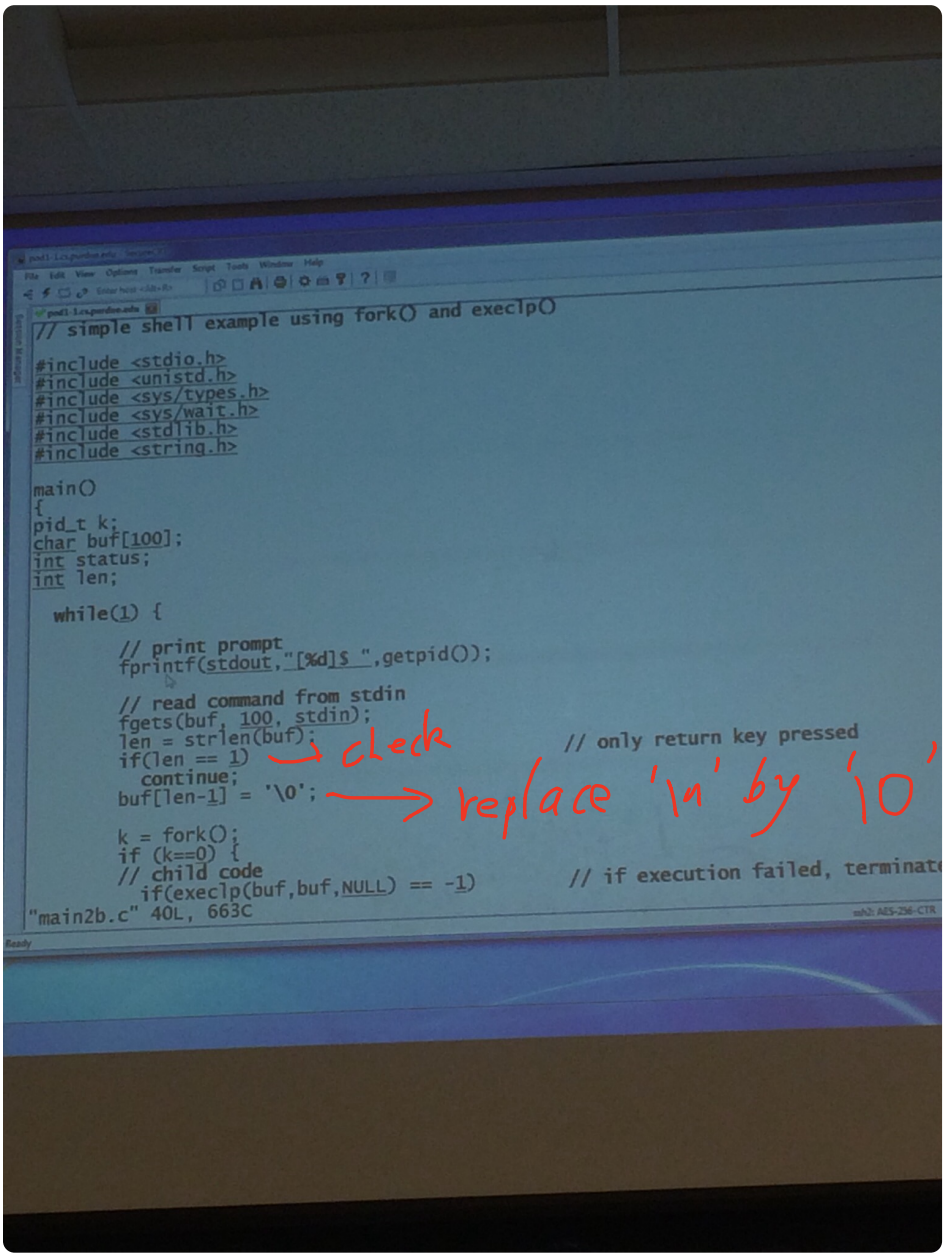
Command longer than [10]

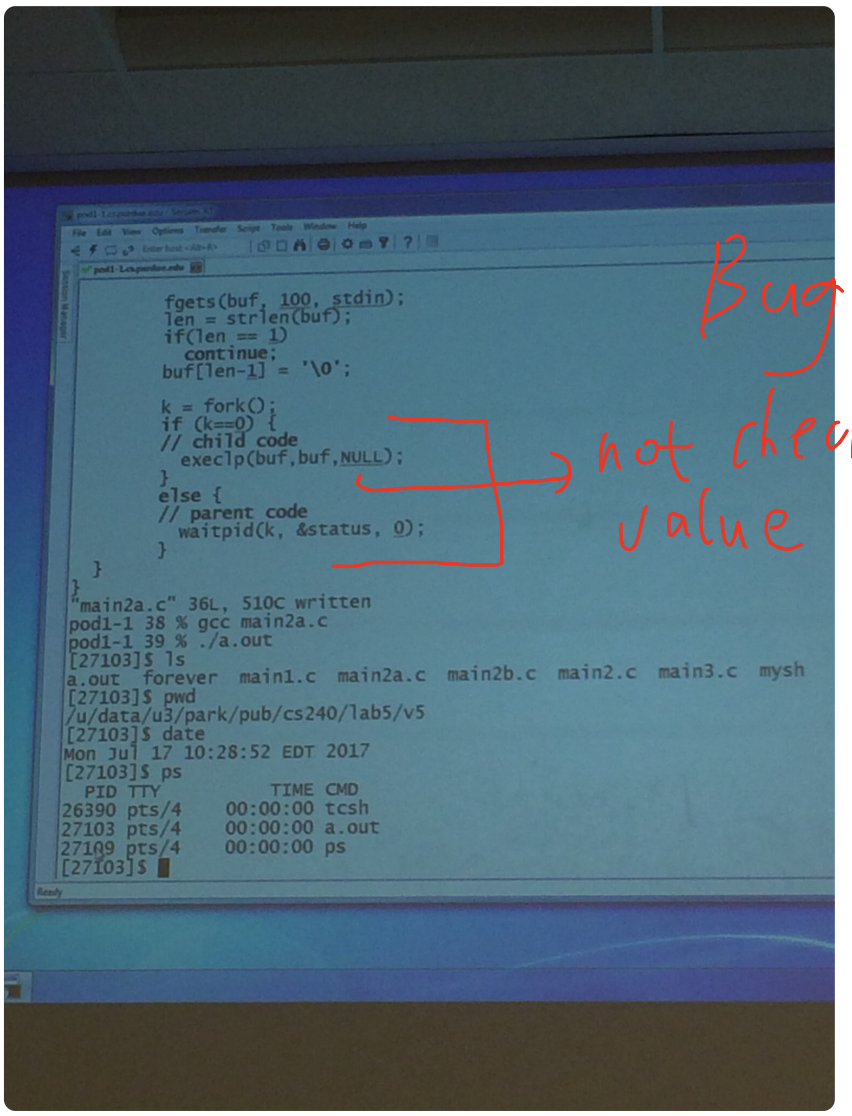
execlp() run fails, child process go back to while(1), print the prompt, which is child process PID,

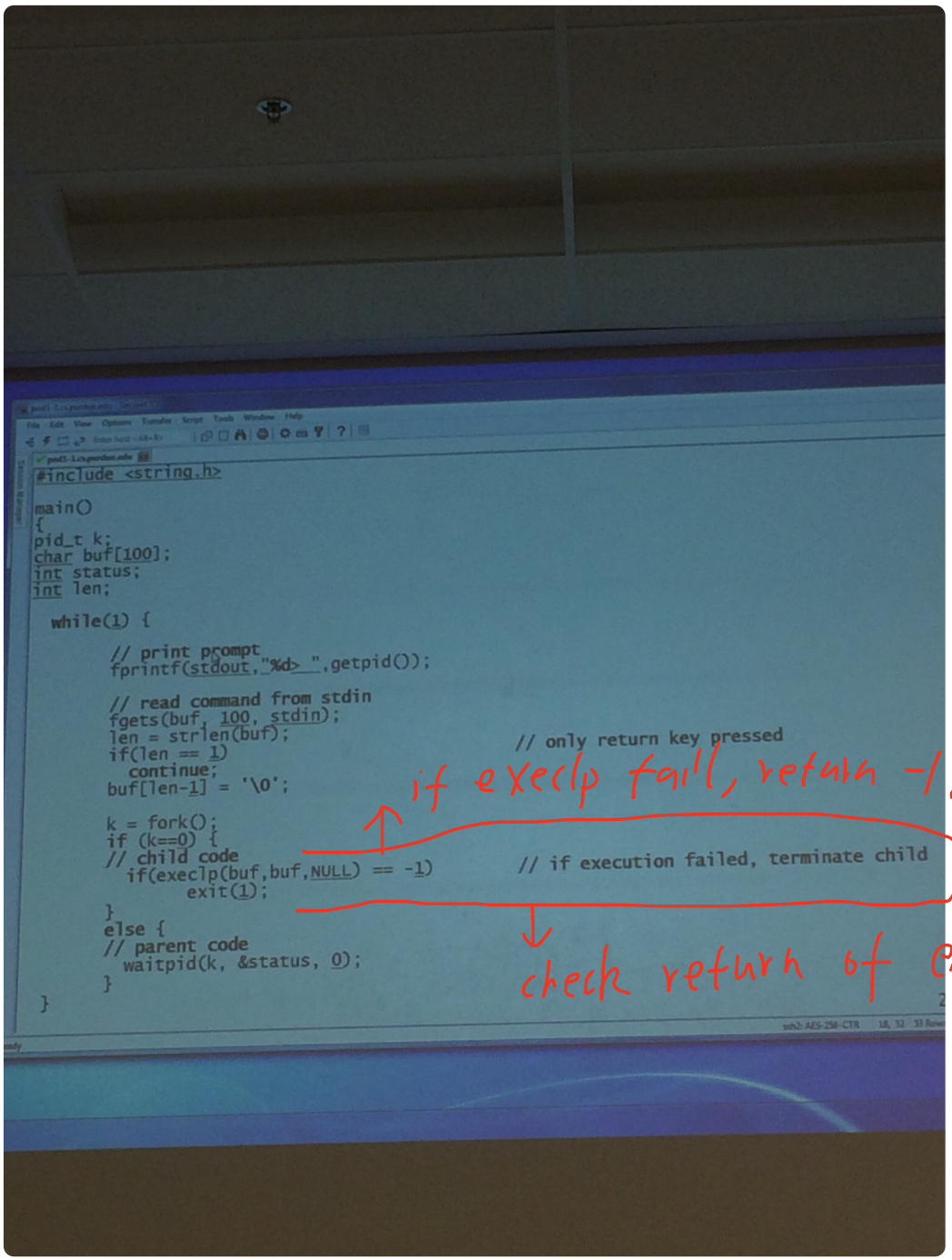
parent still hang on, waiting for the child process to terminate

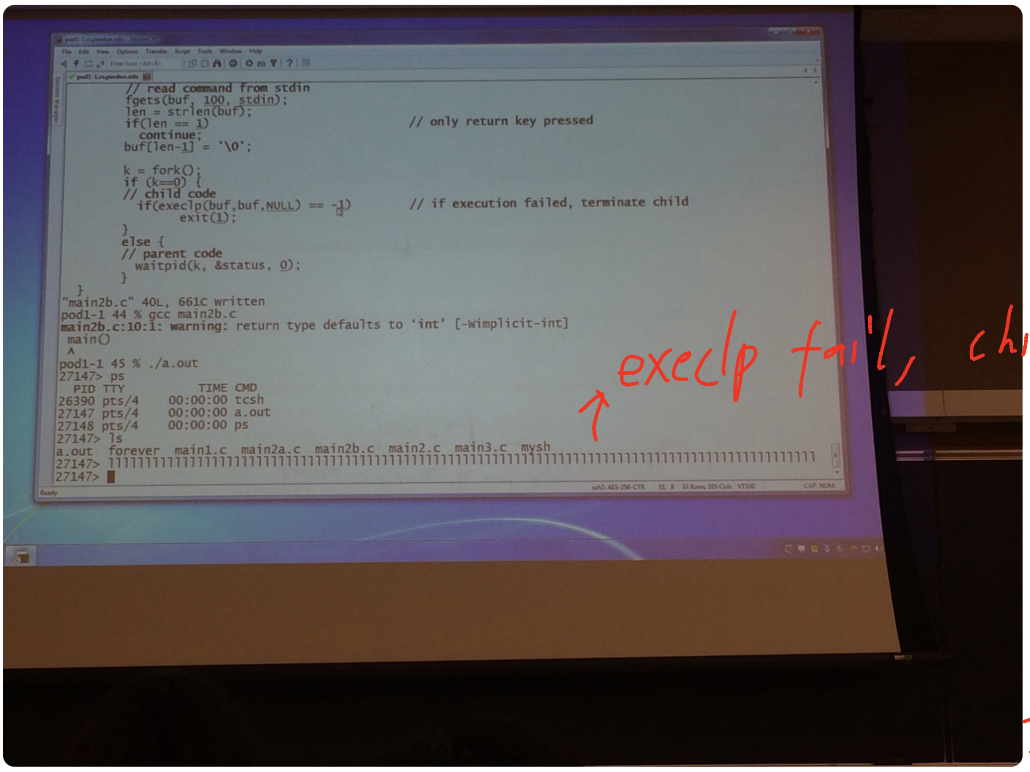


the child prog own shell now, if we use ctrl-c, we will kill the child process, because the signal will send to the owner.


// simple shell example using fork() and execlp()
#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(execlp(buf,buf,NULL) == -1)
// only return key pressed
check
replace '\n' by '\0'
// if execution failed, terminat
"main2b.c" 40L, 663C
Ready


fgets(buf, 100, stdin);
len = strlen(buf);
if(len == 1)
continue;
buf[len-1] = '\0';
k = fork();
if (k==0) {
// child code
execlp(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 retun 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;
buf[len-1] = '\0';
// only return key pressed
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 execlp fail, return -1.
check return of execlp



↑ execlp fail, child process exit(1)

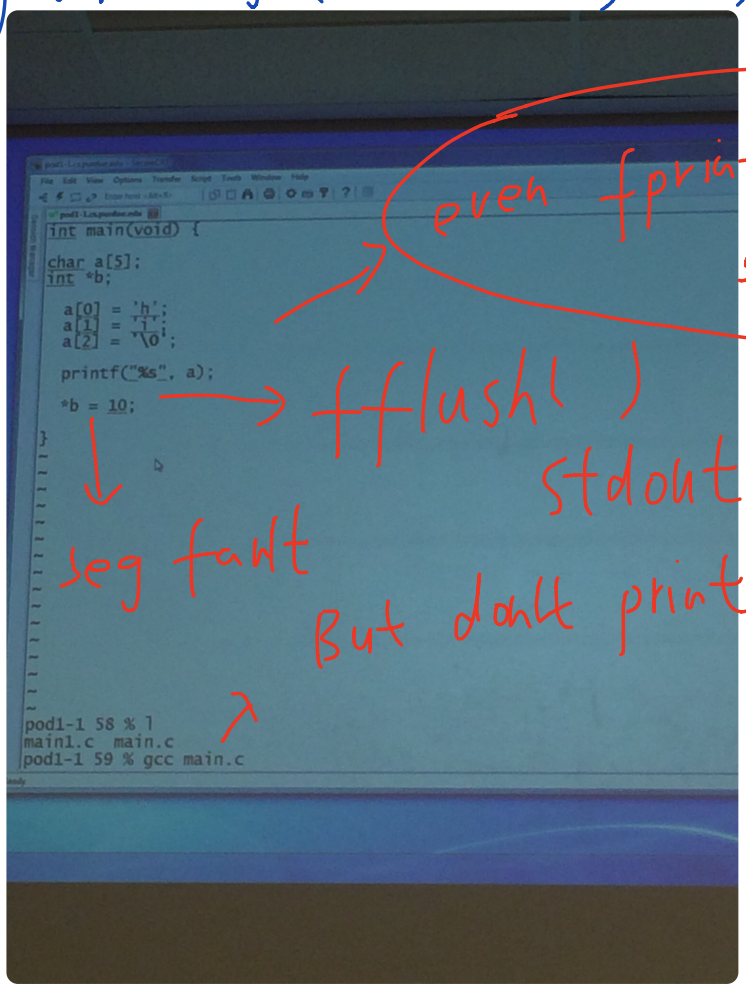
parent process waiting for child,

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

↑

VS (lab6)

work fprintf(stderr, ,) unbuffered



even fprintf(stdout,) same, Don't work

fflush( ) stdout

seg fault

But don't print "Hi"

fflush(stdout)

fflush(stderr)

Inside printf()

use some spare as buffer

in RAM

to temprorly stored writing

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

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

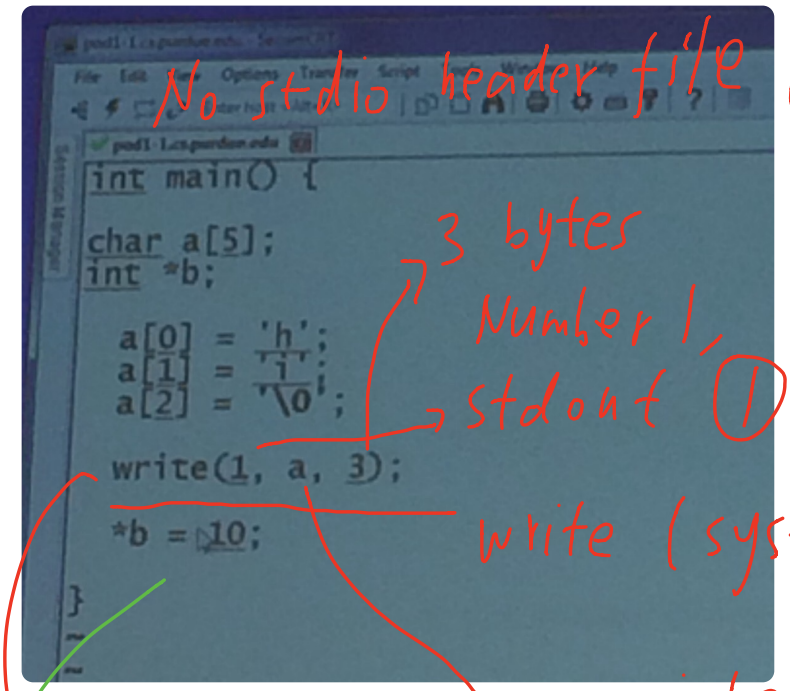
Combine these function togethes, and then do I/O.

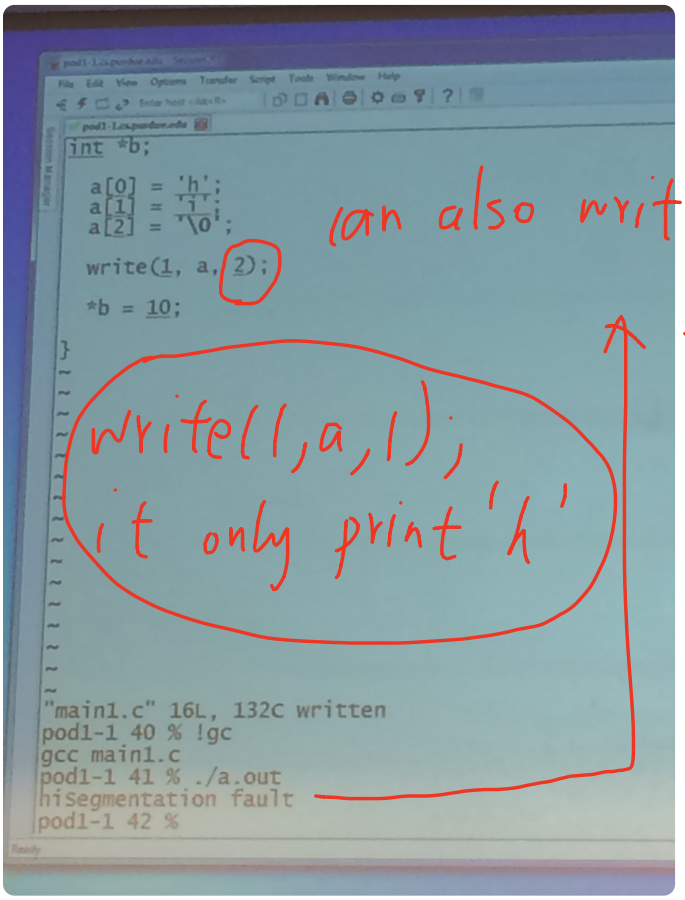
But, when Seg fault, the buffer not

cleaned. divid by zero, . . .

exit(1) will clean the buffer.

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





can also write 2 bytes

Same Result