# Exceptional Control Flow II (Process Operations)

# Outline

- Process id
- Exiting Processes
- Creating Processes
- Reaping Child Processes
- Putting Processes to Sleep
- Loading and Running Programs

# Obtaining Process ID's

- **`Process ID`**
  - PID
  - Each process has a unique positive PID

- **`Getpid()`**
  - Returns the PID of the calling process
- **`Getppid()`**
  - Returns the PID of its parent
    - The process that created the calling process

# Obtaining Process ID's

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void);
pid_t getppid(void);
```

returns: PID of either the caller or the parent

- The **getpid** and **getppid** routines return an integer value of type **pid_t**
- **pid_t**
  - Defined in **types.h** as an **int** on Linux systems

# Exit Function

```
#include <stdlib.h>
void exit(int status);
```

this function does not return

- The **exit** function terminates the process with an **exit status** of status.

# 如何创建和初始化进程?

- 建一个空的数据结构，再填充每个域
- 复制（内核 init 进程是所有进程的祖先， pid=1 ）
  - fork()
    - 创建新进程（子进程），由父进程复制而来
    - 两个进程都从 fork 返回
      - 父进程返回创建的子进程 PID
      - 子进程返回 0
  - exec(..)
    - 参数传进来一个 program ，更换当前进程的 code 和 data ，执行传进来的 program 指令

# Fork Function

- A parent process
  - creates a new running child process
  - by calling the **fork** function

```
#include <unistd.h>
#include <sys/types.h>
pid_t fork(void);
```

Returns: 0 to child, PID of child to parent, -1 on error

# Fork Function

- 新创建的子进程
  - 几乎，但不是完全地，与父进程相同
- 父进程和新创建的子进程有不同的 PID

# Fork Function

- 子进程
  - 得到一份父进程用户层虚拟机地址空间的完全拷贝
    - including the text, data, and bss segments, heap, and user stack.
  - 同时也得到父进程已打开的文件描述符（ file descriptors ）的完全拷贝
    - 意味着子进程可以直接读，写父进程中已经打开的任何文件

# Fork Function

- Called once
  - In the parent process
- Returns twice:
  - In the parent process
    - Return the PID of the child
  - In the newly created child process.
    - Return 0
- The return value provides an unambiguous way
  - whether the program is executing in the parent or the child.

# Fork Function

```
1 #include "csapp.h"
2
3 int main()
4 {
5         pid_t pid;
6         int x = 1;
7
8         pid = Fork();
9         if (pid == 0) { /* child */
10                printf("child : x=%d\n", ++x);
11                exit(0);
12        }
13
14        /* parent */
15        printf("parent: x=%d\n", --x);
16        exit(0);
17 }
```

# Fork Function

- Call once, return twice.
  - As mentioned before
  - This is fairly straightforward for programs that create a single child.
  - Programs with multiple instances of **`fork`**
    - can be confusing
    - need to be reasoned about carefully

# Fork Function

- Concurrent execution
  - The parent and the child are separate processes that run concurrently.
  - The instructions in their logical control flows can be interleaved by the kernel in an arbitrary way.
  - We can never make assumptions about the interleaving of the instructions in different processes.

# Fork Function

- Duplicate but separate address spaces
  - Immediately after the **`fork`** function returned in each process, the address space of each process is identical.
    - Local variable **`x`** has a value of 1 in both the parent and the child when the **`fork`** function returns in line 8.

# Fork Function

- Duplicate but separate address spaces.
  - The parent and the child are separate processes
    - they each have their own private address spaces.
    - Any subsequent changes that a parent or child makes to **x**
      - private
      - not reflected in the memory of the other process.

# Fork Function

- Duplicate but separate address spaces
  - The variable **`x`** has different values in the parent and child when they call their respective **`printf`** statements. (++x/--x)
- Shared files
  - Like **`stdout`**
  - Communication between child and parent

# Fork Function

```
                                    prompt> ./p1
                                    hello world (pid:29146)
1   #include <stdio.h>              hello, I am parent of 29147 (pid:29146)
2   #include <stdlib.h>             hello, I am child (pid:29147)
3   #include <unistd.h>             prompt>
4
5   int
6   main(int argc, char *argv[])
7   {
8       printf("hello world (pid:%d)\n", (int) getpid());
9       int rc = fork();
10      if (rc < 0) {           // fork failed; exit
11          fprintf(stderr, "fork failed\n");
12          exit(1);
13      } else if (rc == 0) { // child (new process)
14          printf("hello, I am child (pid:%d)\n", (int) getpid());
15      } else {                // parent goes down this path (main)
16          printf("hello, I am parent of %d (pid:%d)\n",
17                  rc, (int) getpid());
18      }
19      return 0;
20  }
```

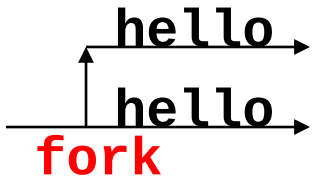
# Fork Function

```
prompt> ./p1
hello world (pid:29146)
hello, I am child (pid:29147)
hello, I am parent of 29147 (pid:29146)
prompt>
```

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <sys/wait.h>
5
6   int
7   main(int argc, char *argv[])
8   {
9       printf("hello world (pid:%d)\n", (int) getpid());
10      int rc = fork();
11      if (rc < 0) {          // fork failed; exit
12          fprintf(stderr, "fork failed\n");
13          exit(1);
14      } else if (rc == 0) { // child (new process)
15          printf("hello, I am child (pid:%d)\n", (int) getpid());
16      } else {               // parent goes down this path (main)
17          int wc = wait(NULL);
18          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
19                  rc, wc, (int) getpid());
20      }
21      return 0;
22  }
```

# Fork Function

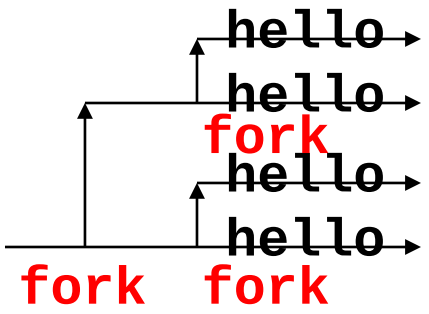
```
1 #include "csapp.h"
2
3 int main()
4 {
5         Fork();
6         printf("hello!\n");
7         exit(0);
8 }
```




(a) Calls fork once.  (b) Prints two output lines.

# Fork Function

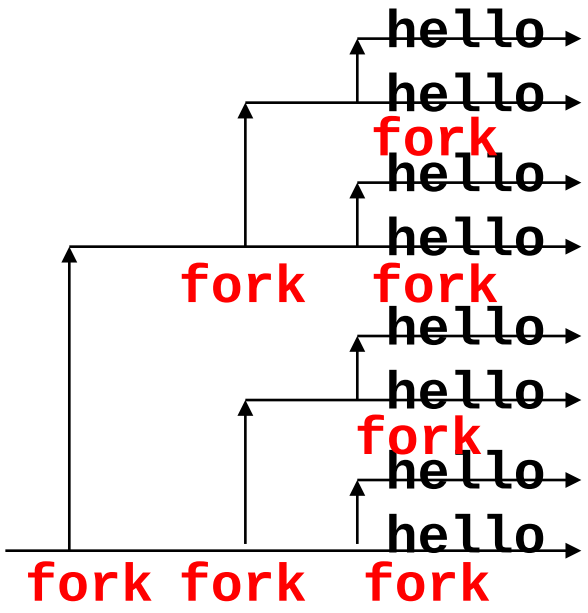
```
1 #include "csapp.h"
2
3 int main()
4 {
5         Fork();
6         Fork();
7         printf("hello!\n");
8         exit(0);
9 }
```




(c) Calls fork twice.  (d) Prints four output lines.

# Fork Function

```
1 #include "csapp.h"
2
3 int main()
4 {
5         Fork();
6         Fork();
7         Fork();
8         printf("hello!\n");
9         exit(0);
10 }
```




(e) Calls fork **three** times.  (f) Prints **eight** output lines.

# 课堂练习

*code/ecf/forkprob0.c*

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int x = 1;
6
7       if (Fork() == 0)
8           printf("printf1: x=%d\n", ++x);
9       printf("printf2: x=%d\n", --x);
10      exit(0);
11  }
```

*code/ecf/forkprob0.c*

A. 子进程的输出是什么？

B. 父进程的输出是什么？

**C. 共多少种输出可能？（假设每行一定完整输出）**

# 课堂练习

这个程序会输出多少个“hello”输出行？

*code/ecf/forkprob1.c*

```
1   #include "csapp.h"
2
3   int main()
4   {
5       int i;
6
7       for (i = 0; i < 2; i++)
8           Fork();
9       printf("hello\n");
10      exit(0);
11  }
```

*code/ecf/forkprob1.c*

# 课堂练习

这个程序会输出多少个“hello”输出行？

*code/ecf/forkprob4.c*

```
1   #include "csapp.h"
2
3   void doit()
4   {
5       Fork();
6       Fork();
7       printf("hello\n");
8       return;
9   }
10
11  int main()
12  {
13      doit();
14      printf("hello\n");
15      exit(0);
16  }
```

*code/ecf/forkprob4.c*

# Zombie

- Kernel 并不会在进程终止后立刻将其清理掉
- 已终止的进程一般会保持在 terminated 状态，直到被其 parent 收割（reaped）

# Zombie

- When the parent reaps the terminated child
  - 内核首先把 child 的 exit status 发送给 parent
  - 然后抛弃已经终止的进程
- A terminated process that has not yet been reaped is called a zombie.

# Zombie

- If the parent process
  - terminates without reaping its zombie children,
  - the kernel arranges for the **`init`** process to reap them.
- The **`init`** process
  - has a PID of 1
  - and is created by the kernel during system initialization.

# Zombie

- 长期运行的程序（例如 shells 或 servers），应该总是收割他们的 zombie children
- 即使 zombie 不在运行，没有消耗 CPU 资源，他们也消耗了系统的内存资源

# 课堂练习

- 进程阻塞的原因不包括 _____ 。
  - A. 时间片切换
  - B. 等待 I/O
  - C. 进程 sleep
  - D. 等待解锁

# Wait_pid Function

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

Returns: PID of child if OK, 0 (if WNOHANG) or -1 on error

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **pid > 0**
  - The wait set is the singleton child process
    - whose process ID is equal to **pid**
- **pid = -1**
  - The wait set consists of all of the parent's child processes

# Wait_pid Function

- The **waitpid** function
  - If the calling process has no children
    - returns –1
    - sets **errno** to **ECHILD**
  - If interrupted by a signal
    - returns –1
    - sets **errno** to **EINTR**

>>

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options = 0
  - waitpid 函数的调用进程进入挂起状态，直到它的 wait-set 中的一个 child process 终止了；
  - 如果 wait-set 中的一个进程在进行 waitpid 调用的时候已经终止了，那么 waitpid 立刻返回；
  - Waitpid 返回导致 waitpid 函数返回的、已经终止的 child process 的 PID
  - 已中止的 child process 被从系统中删除，释放其资源

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options
  - WNOHAN: 如果 wait-set 中的任何子进程都还没有终止，那么立即返回，返回值为 0
  - 如果在等待子进程终止的过程的同时，还希望做些有用的工作，这个选项会有用。

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options
  - WUNTRACED: 将调用 waitpid 的进程挂起，直到 wait-set 中的一个子进程已经变成已终止或者被停止。返回已终止或已停止的子进程 PID 。

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options
  - WCONTINUED: 将调用 waitpid 的进程挂起，直到 wait-set 中的一个正在运行的子进程终止，或者等待集合中一个被停止的进程收到 SIGCONT 信号重新开始。

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- options
  - WUNTRACED | WNOHANG：立刻返回。如果等待集合中的进程都没有被停止或终止，则返回 0；如果有一个停止或终止，则返回该进程的 PID 。

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- Checking the exit status of a reaped child
  - The **status** argument is non-NULL
  - **waitpid** encodes status information about the child
    - that caused the return in the **status** argument.
  - The **wait.h** include file defines several macros
    - for interpreting the **status** argument

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **WIFEXITED(status)**
  - Returns true if the child terminated normally
    - via a call to **exit** or a return.
- **WEXITSTATUS(status)**
  - Returns the exit status of a normally terminated child.
  - This status is only defined if **WIFEXITED** returned true.

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **WIFSIGNALED(status)**
  - Returns true if the child process terminated because of a signal that was not caught
- **WTERMSIG(status)**
  - Returns the number of the signal that caused the child process to terminate.
  - This status is only defined if **WIFSIGNALED** returned true.

# Wait_pid Function

```
pid_t waitpid(pid_t pid, int *status, int options);
```

- **WIFSTOPPED(status)**
  - Returns true if the child that caused the return is currently stopped.
- **WSTOPSIG(status)**
  - Returns the number of the signal that caused the child to stop.
  - This status is only defined if **WIFSTOPPED** returned true.

# Wait_pid Function (Nondeterministic)

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6   int status, i;
7   pid_t pid;
8
9   /* Parent creates N children */
10  for (i = 0; i < N; i++)
11    if ((pid = Fork()) == 0) /* child */
12      exit(100+i);
13
```

# Wait_pid Function (Nondeterministic)

```
14  /* Parent reaps N chds. in no particular order */
15  while ((pid = waitpid(-1, &status, 0)) > 0) {
16    if (WIFEXITED(status))
17      printf("child %d terminated normally with exit
18            status=%d\n", pid, WEXITSTATUS(status));
19    else
20      printf("child %d terminated abnormally\n", pid);
21  }
22
23  /* The only normal term. is if there no more chds. */
24  if (errno != ECHILD)
25    unix_error("waitpid error");
26
27  exit(0);
28}
```

Recall P33

# Wait_pid Function (Nondeterministic)

```
unix>./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

- There is no reaping order, every one is equally correct
  - Nondeterministic behavior for currency

# Wait_pid Function (Deterministic)

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6   int status, i;
7   pid_t pid[N+1], retpid;
8
9   /* Parent creates N children */
10  for (i = 0; i < N; i++)
11    if ((pid[i] = Fork()) == 0) /* Child */
12      exit(100+i);
13
14  /* Parent reaps N children in order */
15  i = 0;
```

# Wait_pid Function (Deterministic)

```
15  while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
16    if (WIFEXITED(status))
18      printf("child %d terminated normally with exit
19            status=%d\n", retpid, WEXITSTATUS(status));
20    else
21      printf("child %d terminated abnormally\n", retpid);
22  }
23
24  /* The only normal term. is if there are no more chds. */
25  if (errno != ECHILD)
26    unix_error("waitpid error");
27
28  exit(0);
29}
```

# wait

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
pid_t wait(int *status);
```

Returns: PID of child if OK, 0 (if WNOHANG) or -1 on error

- wait(&status) 是 waitpid 的简化版，相当于
  - waitpid(-1, &status, 0);

# 课堂练习

- Consider the following program:

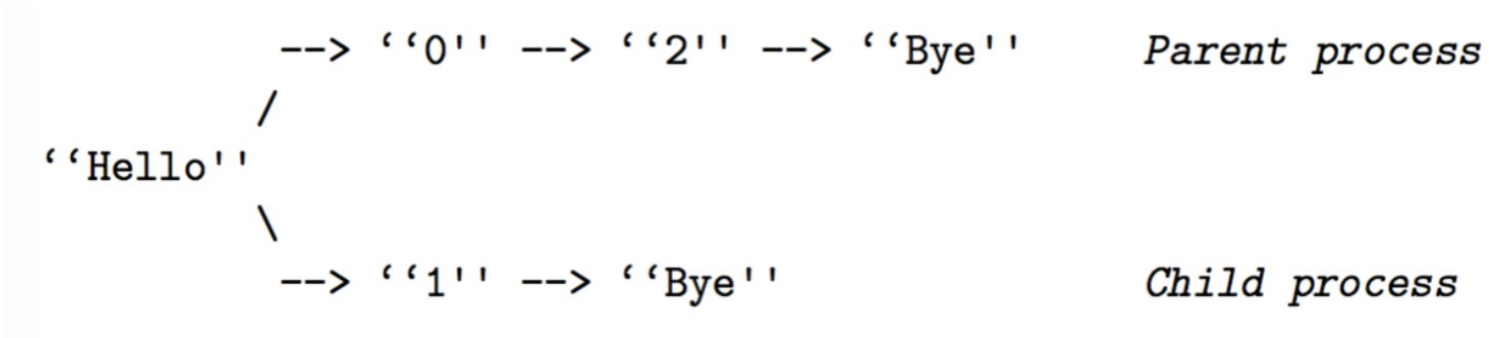
```
int main() {
        int status;
        pid_t pid;
        printf("Hello\n");
        pid = Fork();
        printf("%d\n", !pid);
        if(pid!=0){
                if (waitpid(-1, &status, 0) > 0) {
                        if (WIFEXITED(status) != 0)
                                printf("%d\n", WEXITSTATUS(status)); } }
        printf("Bye\n");
        exit(2);
}
```

**A. How many output lines does this program generate?**
**B. What is one possible ordering of these output**

# 练习答案

A. Each time we run this program, it generates six output lines.

B. The ordering of the output lines will vary from system to system, depending on the how the kernel interleaves the instructions of the parent and the child. In general, any topological sort of the following graph is a valid ordering:



# Putting Process to Sleep

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
                                    Returns: seconds left to sleep
int pause(void);
                                    Always returns -1
```

- sleep
  - 挂起一个进程一段时间;
  - 如果请求的时间量已经到了，则返回 0 ；
  - 否则返回剩下还要休眠的秒数
    - 如果 sleep 函数被一个信号中断而过早返回时，会没有休眠足够的描述而提前返回。

# Putting Process to Sleep

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
                                    Returns: seconds left to sleep
int pause(void);
                                                Always returns -1
```

- pause
  - 让调用函数休眠，直到该进程收到一个信号

# 课堂练习

- Write a wrapper function for sleep, called snooze, with the following interface:

  unsigned int snooze(unsigned int secs);
- The snooze function behaves exactly as the sleep function, except that it prints a message describing how long the process actually slept:
- *Slept for 4 of 5 secs.*
- *（考虑到sleep 可能收到信号提前返回）*

# 练习答案

```
unsigned int snooze(unsigned int secs) {
        unsigned int rc = sleep(secs);
        printf("Slept for %u of %u secs.\n", secs
- rc, secs);
        return rc;
}
```

# 父进程与子进程

- 父进程等待子进程结束
  - Waitpid()
- 子进程等待父进程结束
  - 轮询（polling）
    - While (getppid() != 1)
      - sleep(1);
    - getppid() 返回父进程 PID，永远成功返回
      - PID=1: ？
  - 或者采用效率更高的信号机制

# 竞争条件

- 子进程和父进程各输出一个字符串
- static void charatatime(char *);
- int main(void) {
  - pid_t pid;
  - if((pid=fork()) < 0) {
    - err_sys(“fork error”);
  - }else if (pid == 0) {
    - charatatime(“output form child\n”);
  - }else {
    - charatatime(“output form parant\n”);
  - }
  - exit(0);
- }

```
// char at a time
Static void charatatime(char
*str) {
        Char *ptr;
        Int c;
        Setbuf(stdout, NULL);
        For(ptr=str; (c=*ptr++)!=0;)
                Putc(c, stdout);
}
```

**Setbuf(): 三种 buffer 方式: unbuffered, block buffered, and line buffered**

http://man7.org/linux/man-pages/man3/setbuf.3.html

# 竞争条件

- 输出结果：
  - #./a.out
  - ooutput from child
  - utput from parent
  - #./a.out
  - output from child
  - output from parent

**为什么?**
**怎么改进程序？子 -> 父；父 -> 子**
**假设有以下宏**
**TELL_PARENT(pid_t)**
**WAIT_PARENT()：阻塞**
**TELL_CHILD(pid_t)**
**WAIT_CHILD()：阻塞**
**来互相通信**

# 竞争条件

- 先子后父
- int main(void) {
  - pid_t pid;
  - if((pid=fork()) < 0) {
    - err_sys(“fork error”);
  - }else if (pid == 0) {
    - charatatime(“output form child\n”);
    - TELL_PARANT(getppid());
  - }else {
    - WAIT_CHILD();
    - charatatime(“output form parant\n”);
  - }
  - exit(0);
- }

# 竞争条件

- 先父后子
- int main(void) {
  - pid_t pid;
  - if((pid=fork()) < 0) {
    - err_sys(“fork error”);
  - }else if (pid == 0) {
    - WAIT_PARENT();
    - charatatime(“output form child\n”);
  - }else {
    - charatatime(“output form parant\n”);
    - TELL_CHILD(pid);
  - }
  - exit(0);
- }

# 竞争条件

- 对于前面先父后子的程序，用 $./a.out 输出一次是正确的。但是用下列方式执行多次，输出就不正确了。
- $./a.out;./a.out;./a.out
- output from parent
- ooutput from parent
- ouotput from child
- put from parent
- output from child
- utput from child
- 原因是什么？怎么样才能改正错误？如果先子后父，会不会有上述问题?

# 竞争条件

- 原因：
  - Shell 一行执行多个命令，当父进程结束，就开始执行下一个命令，不管子进程是否执行完。
  - 因此 child 的数据会与 parent 的随机交叉
  - 先子后父，不会交叉

# 竞争条件

- int main(void) {
  - pid_t pid;
  - if((pid=fork()) < 0) {
    - err_sys(“fork error”);
  - }else if (pid == 0) {
    - WAIT_PARENT();
    - charatatime(“output form child\n”);
    - TELL_PARENT(getppid());
  - }else {
    - charatatime(“output form parant\n”);
    - TELL_CHILD(pid);
    - WAIT_CHILD();
  - }
  - exit(0);
- }

# Loading and Running

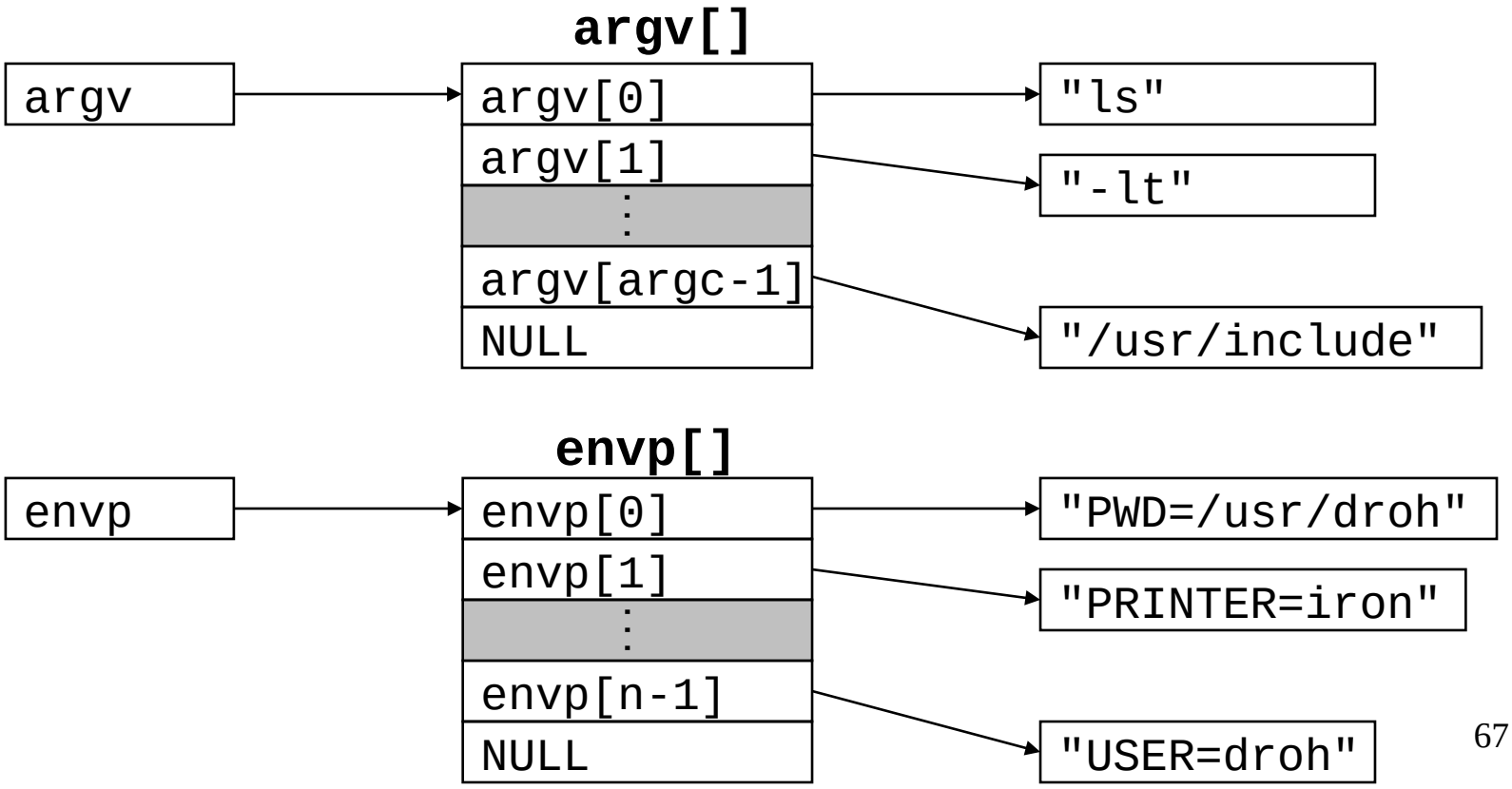
```
#include <unistd.h>
int execve(const char *filename, const char *argv[],

           const char *envp[]);
```

does not return if OK, returns -1 on error

- Loads and runs
  - the executable object file **`filename`**
  - with the argument list **`argv`**
  - the environment variable list **`envp`**.
  - returns to the calling program only if there is an error
    - such as not being able to find **`filename`**.
- The **`Execve`** is called once and never returns

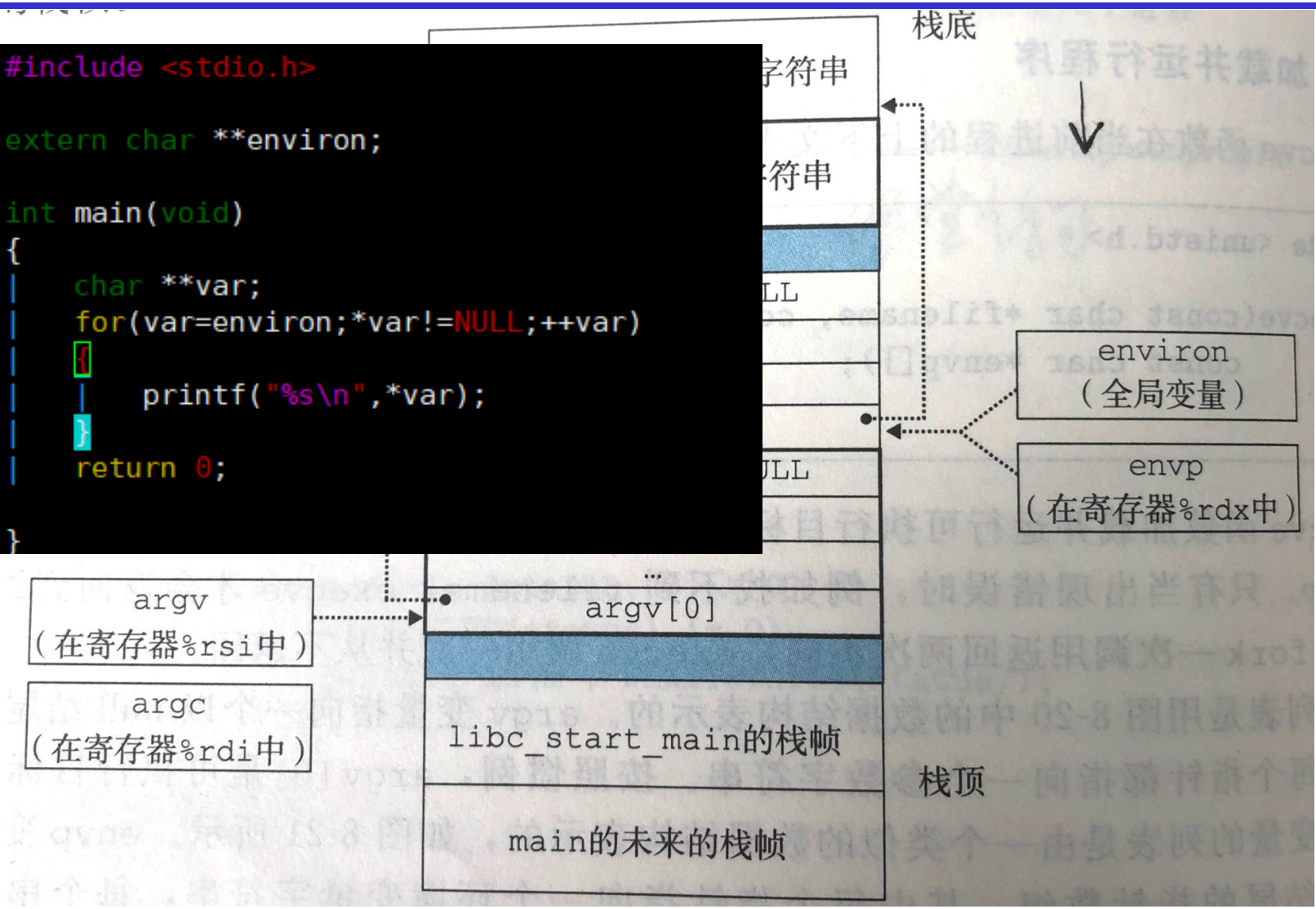
# Loading and Running

```
int execve(const char *filename, const char *argv[],
           const char *envp[]);
```



# The user stack when a new program starts

```c
#include <stdio.h>

extern char **environ;

int main(void)
{
    char **var;
    for(var=environ;*var!=NULL;++var)
    {
        printf("%s\n",*var);
    }
    return 0;

}
```



# Loading and Running

```
#include <unistd.h>
char *getenv(const char *name);
```

Returns: ptr to name if exists, NULL if no match.

```
int setenv(const char *name, const char *newvalue,
           int overwrite);
```

Returns: 0 on success, -1 on error.

```
void unsetenv(const char *name);
```

Returns: nothing.

# Exec

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <unistd.h>
4   #include <string.h>
5   #include <sys/wait.h>
6
7   int
8   main(int argc, char *argv[])
9   {
10      printf("hello world (pid:%d)\n", (int) getpid());
11      int rc = fork();
12      if (rc < 0) {          // fork failed; exit
13          fprintf(stderr, "fork failed\n");
14          exit(1);
15      } else if (rc == 0) { // child (new process)
16          printf("hello, I am child (pid:%d)\n", (int) getpid());
17          char *myargs[3];
18          myargs[0] = strdup("wc");   // program: "wc" (word count)
19          myargs[1] = strdup("p3.c"); // argument: file to count
20          myargs[2] = NULL;           // marks end of array
21          execvp(myargs[0], myargs);  // runs word count
22          printf("this shouldn't print out");
23      } else {                // parent goes down this path (main)
24          int wc = wait(NULL);
25          printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
26                  rc, wc, (int) getpid());
27      }
28      return 0;
29  }
```

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
prompt>
```

wc 统计文件的行数、字数、字节数
一个字被定义为由空白、跳格或换行字符分隔的字符串

# Why? Motivating the API

- 为什么采用 fork 和 exec 的组合来创建进程?
  - 因为用来写 Unix Shell
    - Shell 进程（父进程）先 fork 出一个子进程
    - 然后子进程 exec 执行用户输入的命令 / 程序
    - 父进程 wait ，直到子进程结束（前端模式）
    - 父进程不 wait ，可以继续输入其他命令（后端模式 & ）
  - 而且能够解决问题

- 原 Lab1：写一个自己的 Shell 程序，核心就是 fork 和 exec

# Unix Shell

- An interactive application-level program that
  - runs other program on behalf of the user
- Variants: sh, csh, tcsh, ksh, bash
- Performs a sequence of read/evaluate steps and terminate
  - Read: reads a command line from the user
  - Evaluate: parses the command line and runs programs on behalf of the user

```
/* The main routine for a simple shell program */
1  #include "csapp.h"
2  #define MAXARGS 128
3
4  /* function prototypes */
5  void eval(char*cmdline);
6  int parseline(const char *cmdline, char **argv);
7  int builtin_command(char **argv);
8
```

```
9  int main()
10 {
11   char cmdline[MAXLINE]; /* command line */
12
13   while (1) {
14     /* read */
15     printf("> ");
16     Fgets(cmdline, MAXLINE, stdin);
17     if (feof(stdin))
18       exit(0);
19
20     /* evaluate */
21     eval(cmdline);
22   }
23 }
```

```
1  /* eval - evaluate a command line */
2  void eval(char *cmdline)
3  {
4    char *argv[MAXARGS]; /* argv for execve() */
5    char buf[MAXLINE]; /* holds modified cmd line */
6    int bg; /* should the job run in bg or fg? */
7    pid_t pid; /* process id */
8
9    strcpy(buf, cmdline);
10   bg = parseline(buf, argv);
11   if (argv[0] == NULL)
12     return; /* ignore empty lines */
13
```

# Parsing command line

- Parse the space separated command-line arguments
- Builds the **`argv`** vector
  - which will eventually be passed to **`execve`**

```
1  /* parse the cmd line and build the argv array */
2  int parseline(const char *cmdline, char **argv)
3  {
4    char *delim; /* first space delimiter */
5    int argc; /* number of args */
6    int bg; /* background job? */
7
8    buf[strlen(buf)-1] = ' ';
     /* replace trailing '\n' with space */
12   while (*buf && (*buf == ' ')) /* ignore spaces */
13     buf++;
14
```

```
12      /* build the argv list */
13      argc = 0;
14      while ((delim = strchr(buf, ' '))) {
15        argv[argc++] = buf;
16        *delim = '\0'; /* replace space with '\0'*/
17        buf = delim + 1;
18        while (*buf && (*buf == ' ')) /* ignore spaces */
19          buf++;
20      }
21      argv[argc] = NULL; /* set the end of argv list */
22
23      if (argc == 0) /* ignore blank line */
24        return 1;
25
26      /* should the job run in the background? */
27      if ((bg = (*argv[argc-1] == '&')) != 0)
28        argv[--argc] = NULL;
29      return bg;
30  }
```

# Parsing command line

- The first argument is assumed to be
  - either the name of a **built-in** shell command
    - that is interpreted immediately
  - or an **executable object file**
    - that will be loaded and run in the context of a new child process

```
14    if (!builtin_command(argv)) {
15      if ((pid = Fork()) == 0) { /* execute in child */
16        if (execve(argv[0], argv, environ) < 0) {
17          printf("%s: Command not found.\n", argv[0]);
18          exit(0);
19        }
20      }
21
```

```
14    if (!builtin_command(argv)) {
15      if ((pid = Fork()) == 0) { /* execute in child */
16        if (execve(argv[0], argv, environ) < 0) {
17          printf("%s: Command not found.\n", argv[0]);
18          exit(0);
19        }
20      }
21
```

---

```
34 /* if 1st arg is a builtin command,
      run it and return true */
35 int builtin_command(char **argv)
36 {
37   if (!strcmp(argv[0], "quit")) /* quit command */
38     exit(0);
39   if (!strcmp(argv[0], "&")) /* ignore singleton & */
40     return 1;
41   return 0; /* not a builtin command */
42 }
```

# Foreground and Background

- If the last argument is a “&” character
  - parseline returns 1
  - indicating the program should be executed in the background

  (the shell returns to the top of the loop and waits for the next command)
- Otherwise
  - parseline returns 0
  - indicating the program should be run in the foreground

  (the shell waits for it to complete)

```
14    if (!builtin_command(argv)) {
15      if ((pid = Fork()) == 0) { /* execute in child */
16        if (execve(argv[0], argv, environ) < 0) {
17          printf("%s: Command not found.\n", argv[0]);
18          exit(0);
19        }
20      }
21
22      /* parent waits for foreground job to terminate */
23      if (!bg) { /* foreground */
24        int status;
25        if (waitpid(pid, &status, 0) < 0)
26          unix_error("waitfg: waitpid error");
27      }
28      else
29        printf("%d %s", pid, cmdline);
30    }
31    return;
32  }
```

# homework2

- 提交时间：一周内
- obe.ruc.edu.cn 上传电子版