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 IDs

- **Process ID (PID)**
  - 每个进程有一个唯一的正整数 PID

```
#include <unistd.h>
#include <sys/types.h>
pid_t getpid(void);  // 获取当前进程的 pid
pid_t getppid(void); // 获取当前进程的父进程的 pid
```

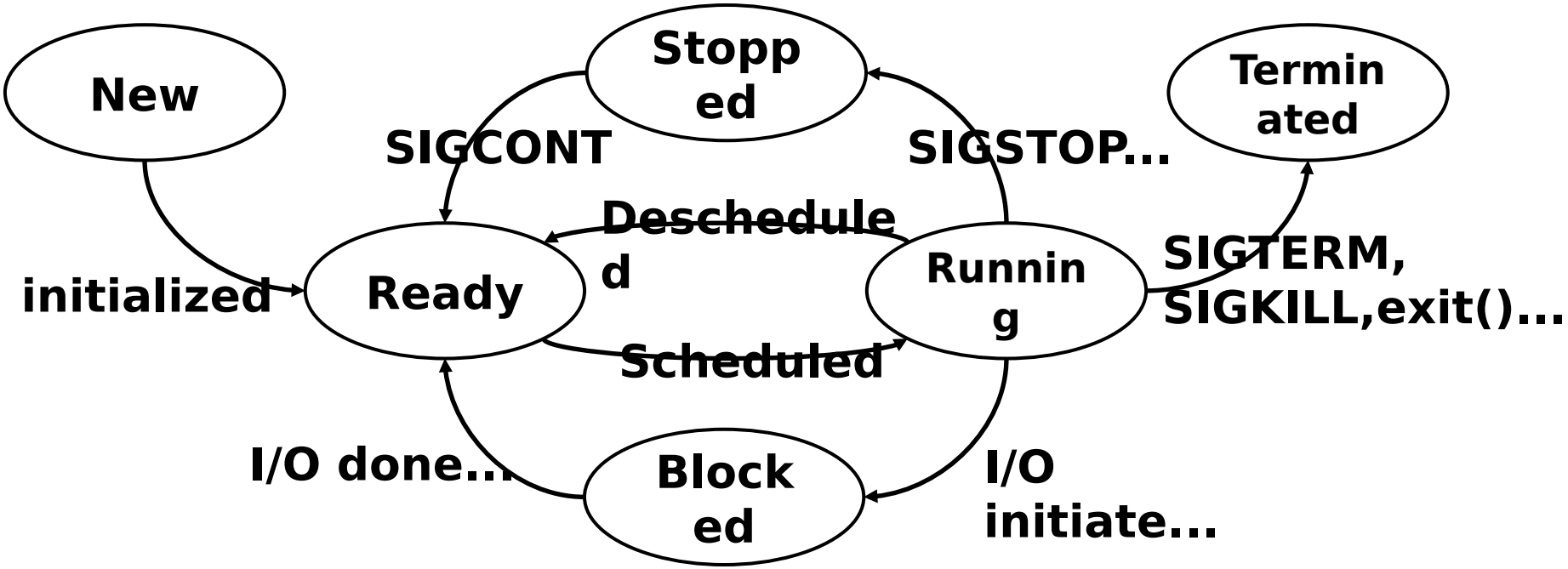
- **pid_t**
  - **在 types.h 中定义**
  - Linux 中就是 int

States of a Process

- New (新建)： 进程正在初始化
- Running (运行)： 进程正在 CPU 上执行
- Ready (就绪)
  - 进程等待被执行，且迟早会被调度执行
- Stopped (暂停)
  - 进程暂停执行，且永远都不会被调度 (除非转为 Ready)
- Blocked (阻塞)
  - 进程等待外部事件 (如 I/O 完成) 而停止执行，且永远都不会被调度 (除非事件完成、转为 Ready)
- Terminated (终止)： 进程永久停止执行

States of a process

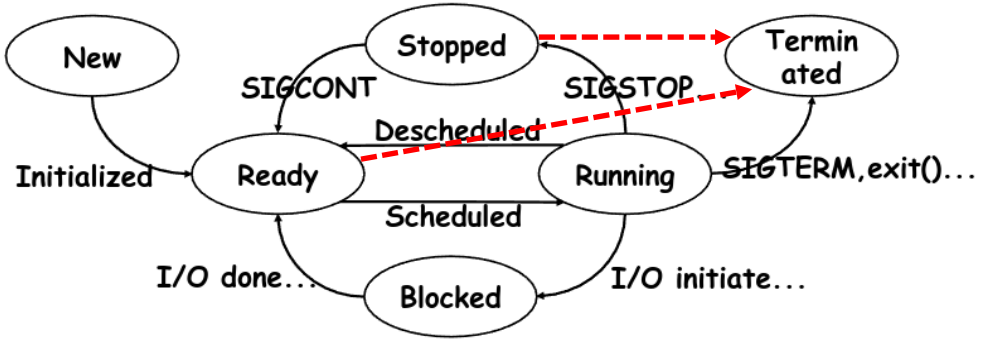
- 进程的状态转移



SIGSTOP、 SIGCONT、 SIGTERM 等是 Signal
Signal:OS 提供的一种 software interrupt 机制，之后课程会讲

States of a process

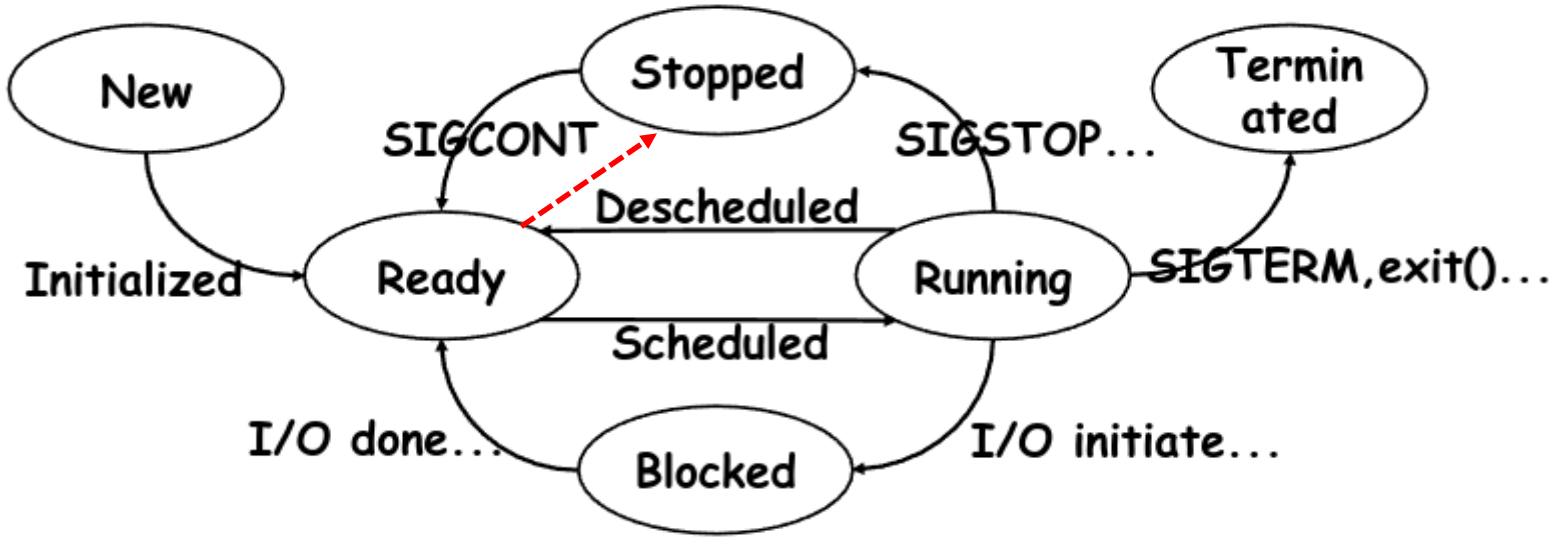
- 终止 (terminate) 意味着回收进程的一切资源
- 进程如何被终止
  - 接收到一个可以终止进程的 signal，如 SIGTERM
  - 从 main 函数返回
  - 调用 **exit** 系统调用
  - 父进程被终止
  - OS 被关闭



Running 状态之外的进程可以被终止吗?

States of a process

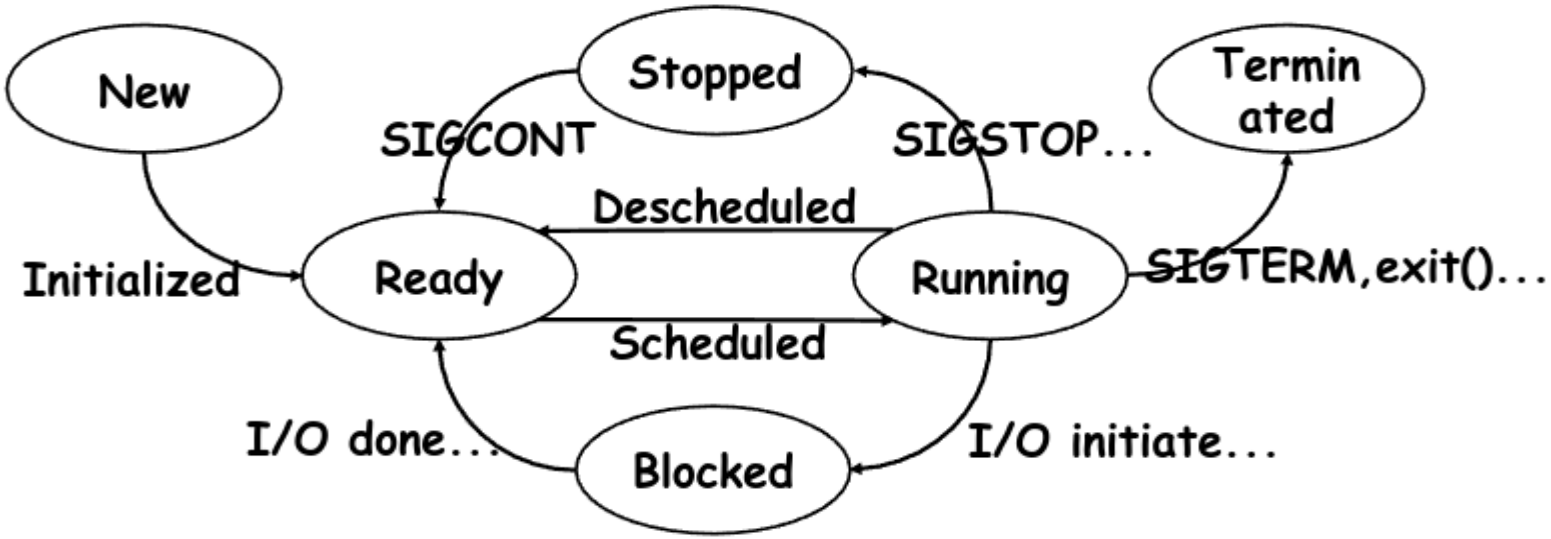
- Stop 意味着暂停执行和调度，除非收到 SIGCONT signal



课堂练习： Running 状态之外的进程可以被 Stop 吗?

States of a process

- 进程不能直接从 Blocked 或 Stopped 状态进入 Running 状态
- 必须先进入 Ready 状态，等待 OS 调度
- 否则进程就可以不进入 ready 队列，这等于跳过了 OS 的进



States of a process

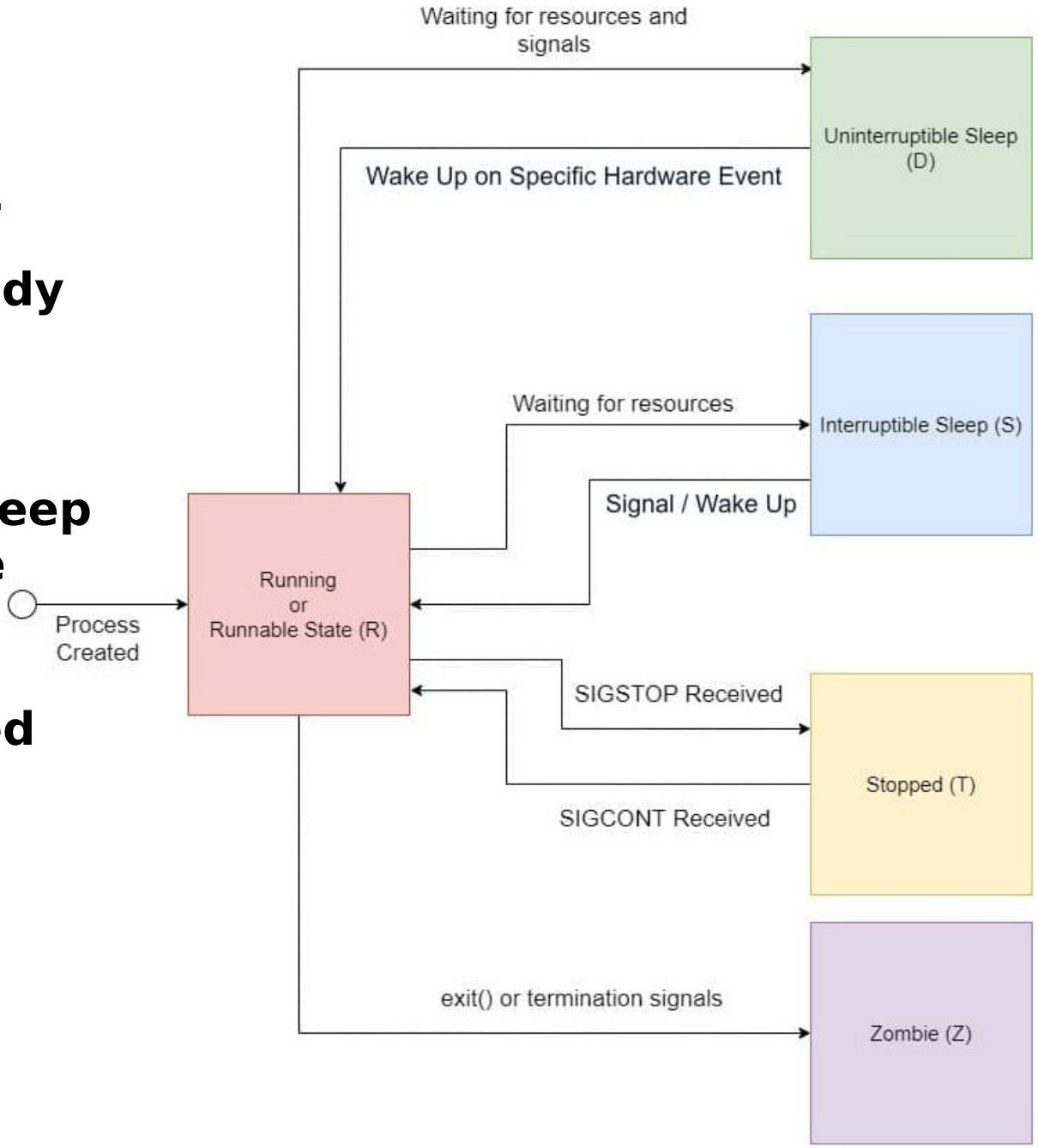
- 进程的状态转移

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Linux 进程状态

R: Running or Ready
T: Stopped
Z: Zombie (terminating)
S: Interruptible Sleep
D: Uninterruptible Sleep

S 和 D 相当于 Blocked
Linux 4.14 之后，内核线程有一个额外的状态 I (idle)，相当于用户进程的 S 状态



Linux 进程状态

- 用 top 、 ps 等命令查看进程状态

```
top - 22:58:04 up 9 days, 15 min,  1 user,  load average: 2.26, 2.55, 2.26
Tasks: 355 total,   5 running, 350 sleeping,   0 stopped,   0 zombie
%Cpu(s): 30.7 us, 10.6 sy,  0.2 ni, 56.8 id,  1.6 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem :  15892.1 total,    873.8 free,   8227.9 used,   6790.4 buff/cache
MiB Swap:   2048.0 total,      0.2 free,   2047.8 used.   5454.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 767419 hank      20   0 1398.7g 759912 191120 R  49.2   4.7  20:28.09 chrome
 110148 hank      20   0   33.4g 183740 115332 R  27.5   1.1 136:46.57 chrome
 109450 hank      20   0 5303048 210960  52720 S  13.4   1.3 140:27.72 gnome-shell
 603122 hank      20   0 1496348  88676  46680 S  10.5   0.5 304:51.01 wpsoffice
 109127 hank      20   0 1569316  90932  36376 S  10.2   0.6 111:46.39 Xorg
 203285 hank      20   0 8564208   2.3g 202704 S   9.8  14.6   1139:36 idea
 110102 hank      20   0   32.9g 464756 231432 R   8.9   2.9 215:45.87 chrome
 108819 root      20   0  230728   3728   3456 S   5.6   0.0 802:08.08 pgyvpn_svr
 110627 hank      20   0 2000992  48788  18384 S   3.0   0.3  77:02.44 gnome-control-c
 615808 hank      20   0 1392.0g 232280 123016 S   2.6   1.4  97:13.28 chrome
 703759 hank      20   0 3136368 575252 231316 S   2.6   3.5   8:12.98 wpp
 110152 hank      20   0   32.4g 110272  72828 S   1.6   0.7  43:56.32 chrome
 392515 hank      20   0 1363640  16948   6492 S   1.6   0.1 121:25.29 feishu
 779752 hank      20   0   17112   4608   3584 R   1.6   0.0   0:05.72 top
 203375 hank      20   0    5928   2176   1403 S   1.3   0.0 139:36.07 fsnotifier
 614426 hank      20   0 1396.9g 221768 109692 S   1.3   1.4   7:38.06 chrome
   1325 mysql     20   0 2441348   9420   4864 S   1.0   0.1 137:54.45 mysqld
   1053 root      20   0  184192   7296   6912 S   0.7   0.0 140:35.68 ECAgent
 109935 hank      39  19  652360  25020  12172 S   0.7   0.2   0:09.98 tracker-miner-f
 204533 hank      20   0 4195384 224856   5988 S   0.7   1.4  58:08.66 java
 392426 hank      20   0 7239132 213072  46392 S   0.7   1.3  35:30.57 feishu
 767095 hank      20   0 1398.7g 384108 156420 S   0.7   2.4   0:53.43 chrome
 775825 root      20   0       0      0      0 I   0.7   0.0   0:09.60 kworker/1:0-i915-unordered
     24 root      20   0       0      0      0 S   0.3   0.0   6:55.77 ksoftirqd/1
    400 root     -51   0       0      0      0 S   0.3   0.0  12:49.69 irq/133-iwlwifi
    740 message+  20   0   11352   5632   2944 S   0.3   0.0  14:38.74 dbus-daemon
    745 root      20   0  264816   9980   7932 S   0.3   0.1  16:25.33 NetworkManager
   1036 root      20   0    9244   4608   4096 S   0.3   0.0  28:43.47 EasyMonitor
   2478 root      20   0   80936   4608   4224 S   0.3   0.0  63:18.84 pgyvpn_oraysl
```

Thread

- 在同一个进程空间的多个指令流（控制流）
- 共享很多进程资源，变量可以互相访问
- 切换开销较小
- 将在“并行”部分详细介绍

- Linux 上的线程是通过共享地址空间的进程实现的，线程相当于 lightweight process ，被内核调度

Linux Thread 的两种实现

- pthread (POSIX Thread) 是 Unix-like 系统的标准线程接口，但 Linux 对其实现并不完善

- LinuxThreads
  - Linux 2.4 (2000 年) 之前的实现
  - 对 pthread 的兼容较差， getpid() 返回不同的 pid
- NPTL (Native POSIX Threads Library)
  - Linux 2.4 及以后的实现
  - 对 pthread 兼容更好，采用 thread group 将统一进程的线程放在同一组， getpid() 返回父进程的 pid

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

- 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

- 父进程和新创建的子进程有不同的 PID
- 子进程
  - 得到一份父进程虚拟机地址空间的完全拷贝
    - 包括 text, data, bss segments, heap，和 user stack
    - **只是地址空间的拷贝，不是内容的拷贝**
    - **父子进程对内存的修改采用 copy-on-write，互相不可见**
  - 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

- 调用一次
  - In the parent process
- 返回两次
  - 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

- Concurrent execution
  - 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

```
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

- Duplicate but separate address spaces
  - Local variable **x** has a value of 1 in both the parent and the child when the **fork** function returns in line 8
  - Any subsequent changes that a parent or child makes to **x are** private
    - not reflected in the memory of the other process

Fork

- Shared files
  - Like **`stdout`**
  - Communication between child and parent

Fork

```
                                    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  }
```

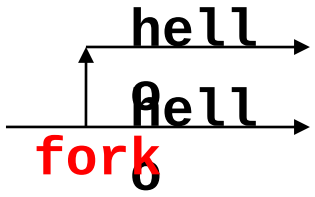
Fork

```
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

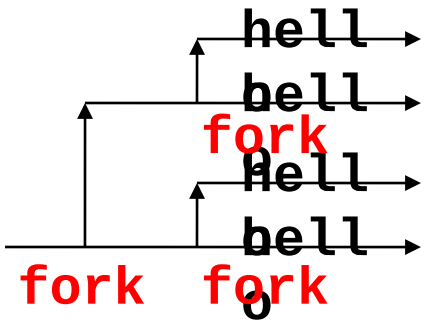
```
1 #include "csapp.h"
2
3 int main()
4 {
5         fork();
6         printf("hello!\n");
7         exit(0);
8 }
```



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

Fork

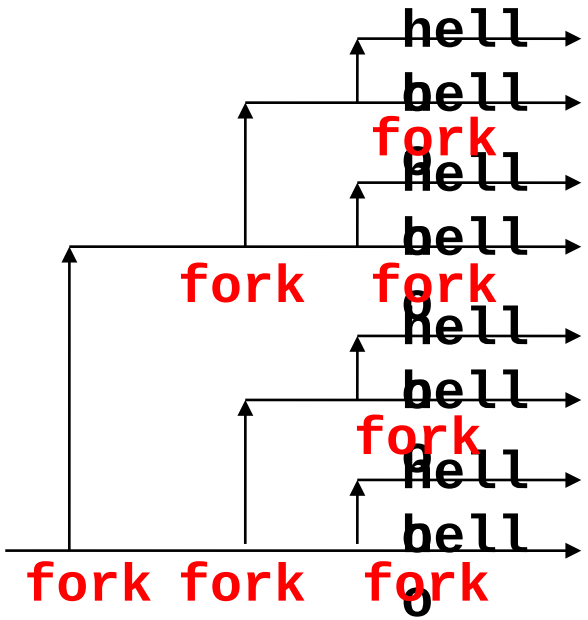
```
1 #include "csapp.h"
2
3 int main()
4 {
5         fork();
6         fork();
7         printf("hello!\n");
8         exit(0);
9 }
```



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

Fork

```
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

- 父进程收割已终止的子进程时
  - 内核首先把子进程的 exit status 发送给父进程
  - 然后抛弃已经终止的进程
- 已终止但未被收割的进程称为 zombie 进程

Zombie

- 如果父进程退出时没有收割 zombie 子进程
  - **这些 zombie 将由 init 进程负责收割**
- **init** process
  - PID 为 1
  - 由内核在启动时创建
  - 除非 OS 关闭，否则永远不会终止

Zombie

- 长期运行的程序（例如 shells 或 servers），应该总是收割他们的 zombie 子进程
- 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

进程如何等待子进程终止?

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

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

Wait_pid

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

Wait-set:

- **pid > 0**
  - 等待 pid 为该值的子进程终止并被收割
- **pid = -1**、
  - 等待该进程的所有子进程终止并被收割

Wait_pid Function

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

- options = 0
  - waitpid 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 返回已经终止的 child process 的 PID

Wait_pid Function

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

- options=WNOHANG
  - 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
  - Default option 仅当子进程终止时返回

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 wait-set 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);
```

- 用于查看被收割子进程的退出状态
  - **status** 为 non-NULL
  - **wait.h** 中定义了一些帮助解析 status 的宏

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

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

- **WIFCONTINUED(status)**
  - Returns true if the child was restarted by receipt of a SIGCONT signal

Wait_pid

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

- Error conditions
  - 如果当前进程没有子进程
    - 返回－1，并设置 **errno 为 ECHILD**
  - 如果该函数的执行被 signal 中断
    - 返回－1 ，并设置 **errno 为 EINTR**

>>

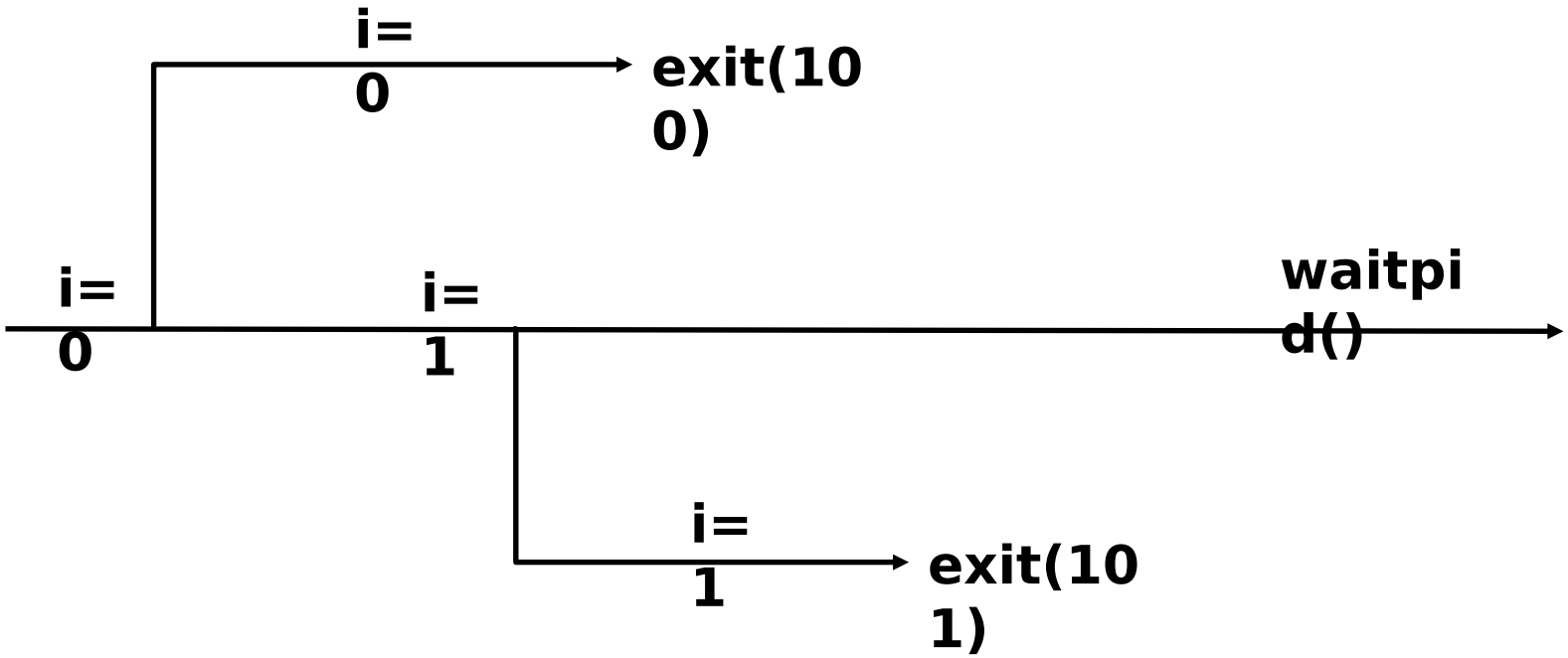
Wait_pid (Nondeterministic)

```
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 (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}
```

Wait_pid (Nondeterministic)



Wait_pid (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 order is correct
  - Nondeterministic behavior for concurrency

Wait_pid (Deterministic)

```
1 #include "csapp.h"
2 #define N 2
3
4 int main()
5 {
6   int status, i;
7   pid_t pid[N], 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 (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 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);
}
```

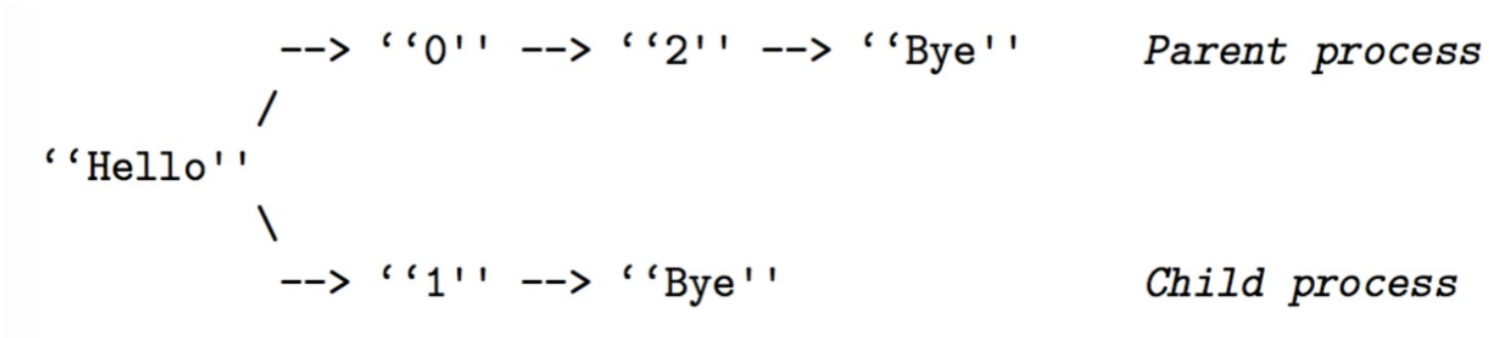
A. How many output lines does this program generate?

B. What is one possible ordering of these output lines?

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，会没有休眠足够的时间而提前返回

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
  - 让调用函数的进程休眠，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

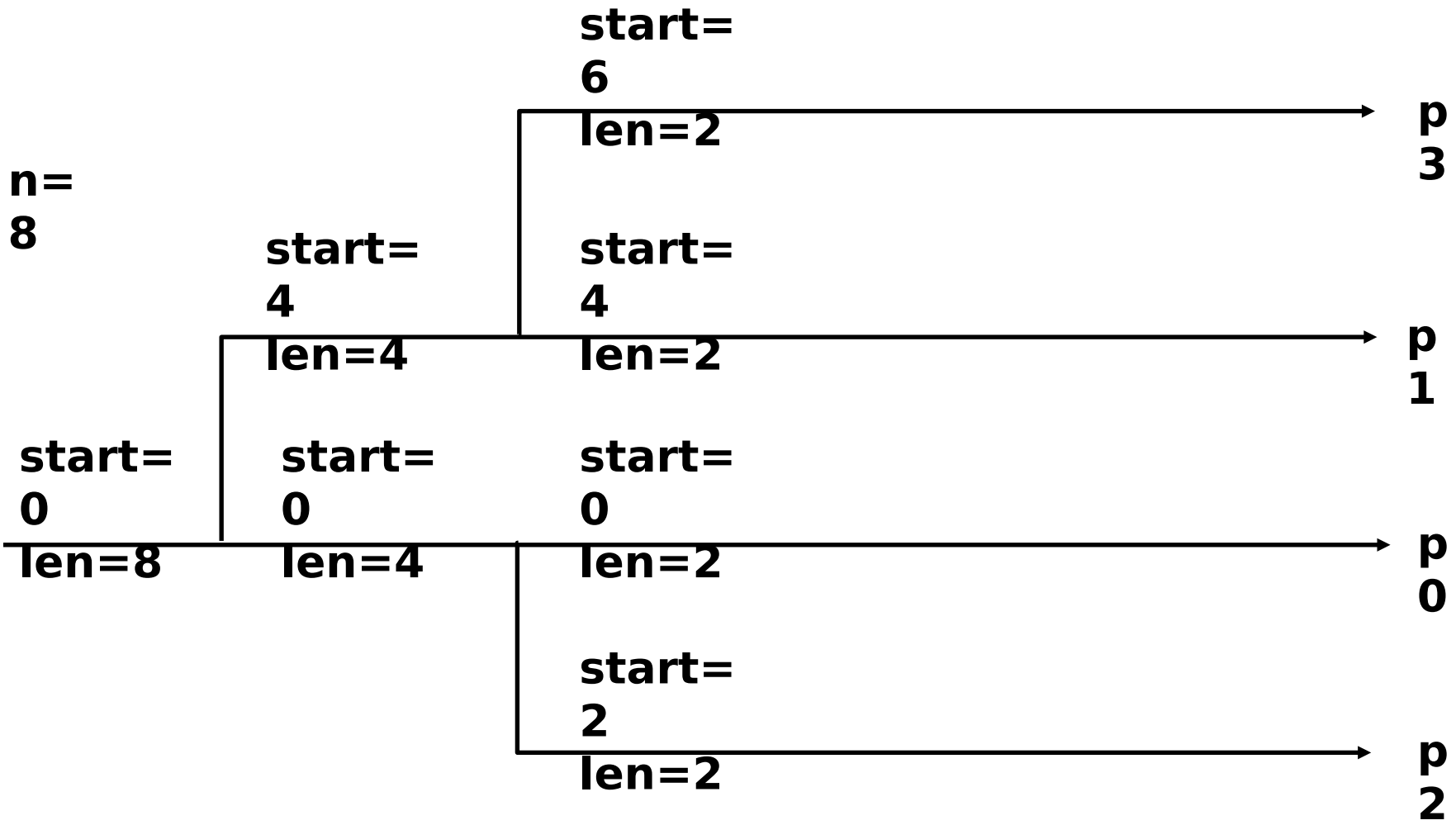
```
unsigned int snooze(unsigned int secs) {
        unsigned int rc = sleep(secs);
        printf("Slept for %u of %u secs.\n", secs
- rc, secs);
        return rc;
}
```

课堂练习

- 一个进程初始化一个长度为 n 的整形数组 arr，然后创建一个子进程，父进程和子进程再各自创建一个进程（共 4 个进程），每个进程打印这个整形数组的 1/4 （不重复，且将整个数组打印完成，不需要考虑打印顺序；假设 n 能够被 4 整除）。
- 请写出代码 / 伪代码

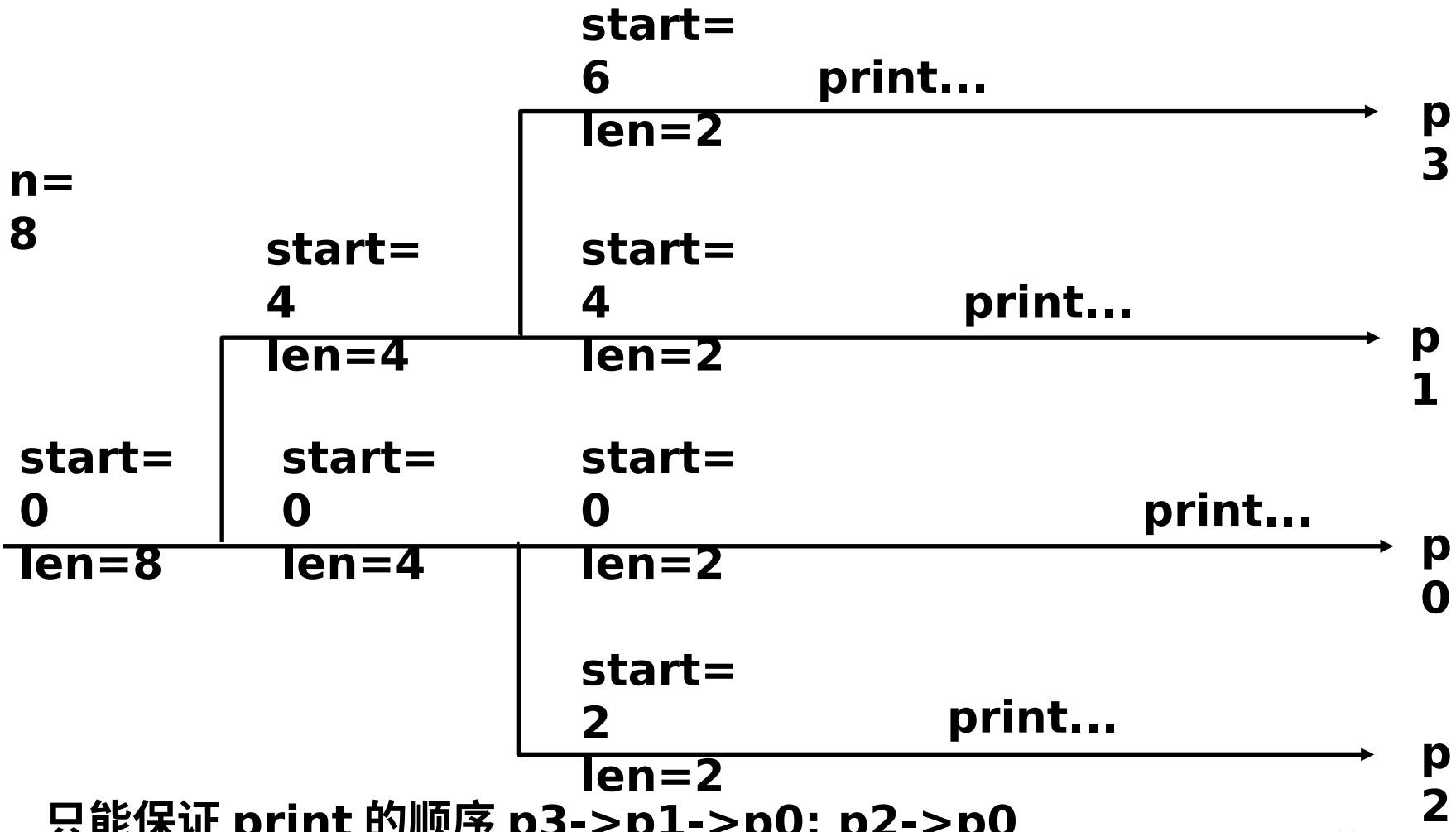
练习答案

每次 fork 后:
子进程中 start+=len/2;len/=2;
父进程中 len/=2



如果要控制打印顺序呢?

练习答案



只能保证 print 的顺序 p3->p1->p0; p2->p0
如何解决?

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);
- }

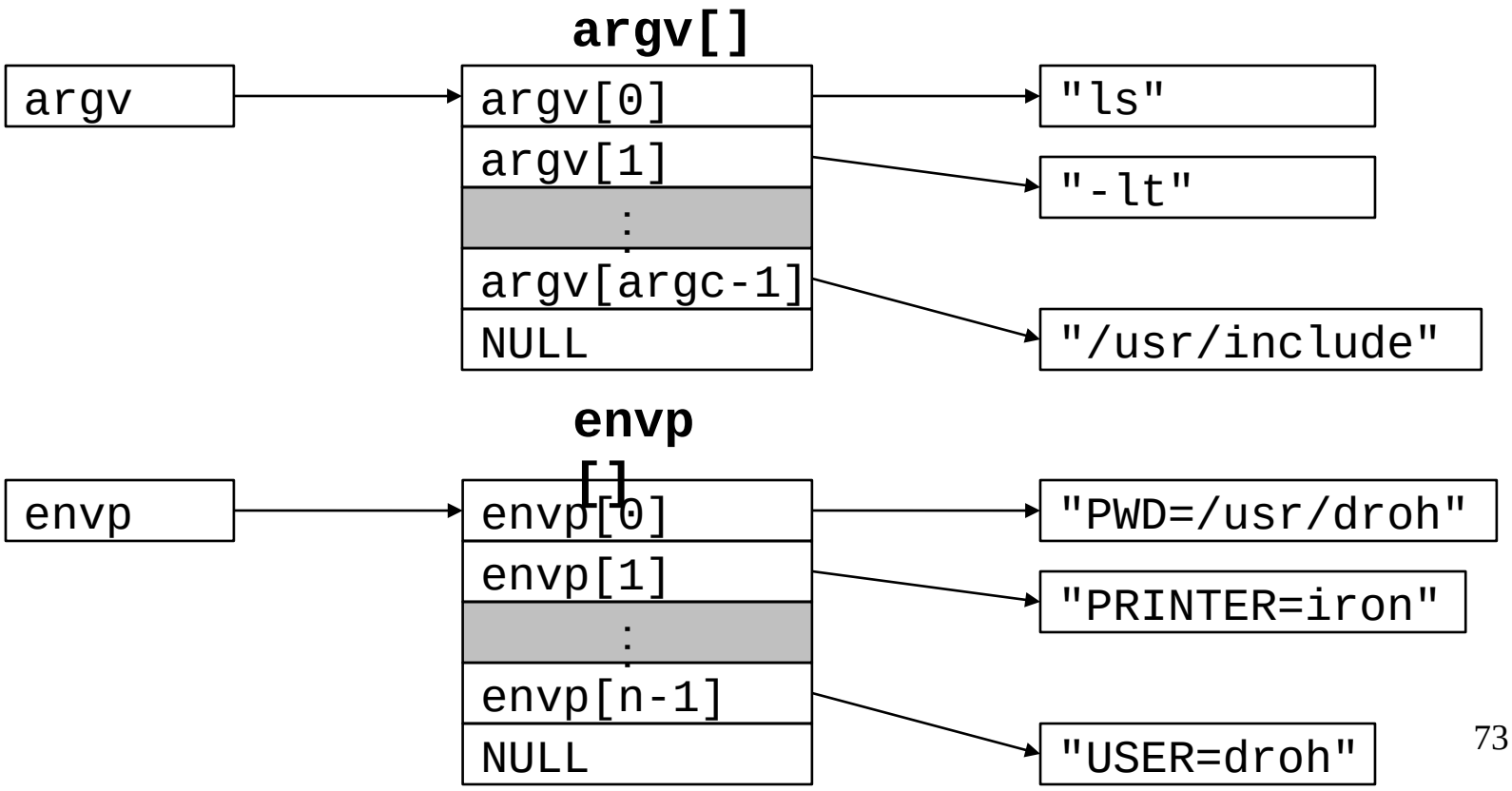
Loading and Running

```
#include <unistd.h>
int execve(const char *filename, const char *argv[],
           const char *envp[]);
```
does not return if OK, returns -1 on error

- 在当前进程中装载一个新程序并运行:
  - 可执行文件 **filename**
  - argument list **argv**
  - environment variable list **envp**.
  - 只有当出错时才返回
    - 例如无法找到 **filename**.
- The **Execve** is called once and never returns

Loading and Running

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



Loading and Running

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

execve 重新 set-up user stack 并将控制流转到一个新的 main 函数：
int main(int argc, char **argv, char **envp);

修改 environment variables

```
#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 family funcs

```
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>
```

```
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  }
```

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?
  - 在 shell 和 web server 等程序中很常见
  - 在 Unix Shell 程序中
    - 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 & ）

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

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

Foreground and Background

- If the last argument is a “&” character
  - 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
  - indicating the program should be run in the foreground

  (the shell waits for it to complete)

homework 1 & 2

- 提交时间：一周内
- obe.ruc.edu.cn 上传电子版