fork + exec + dup

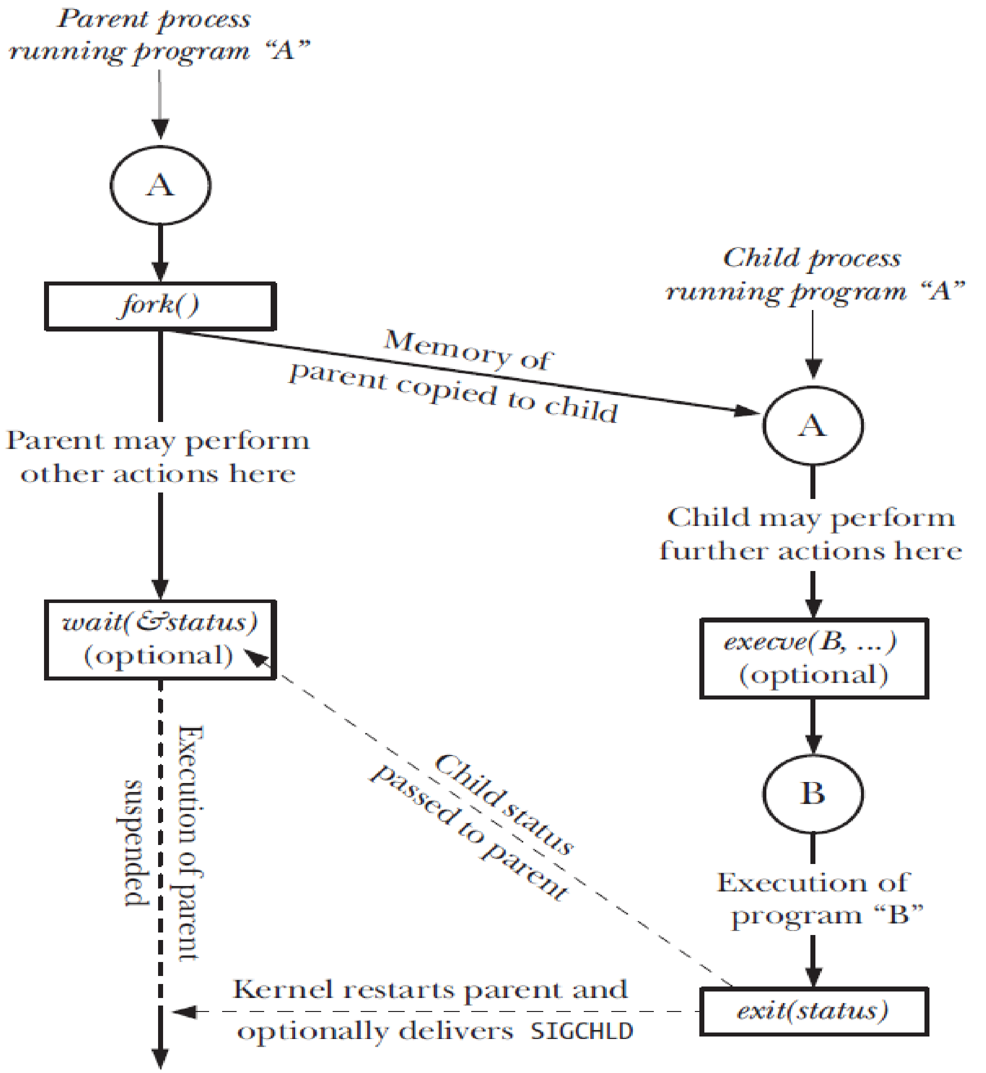
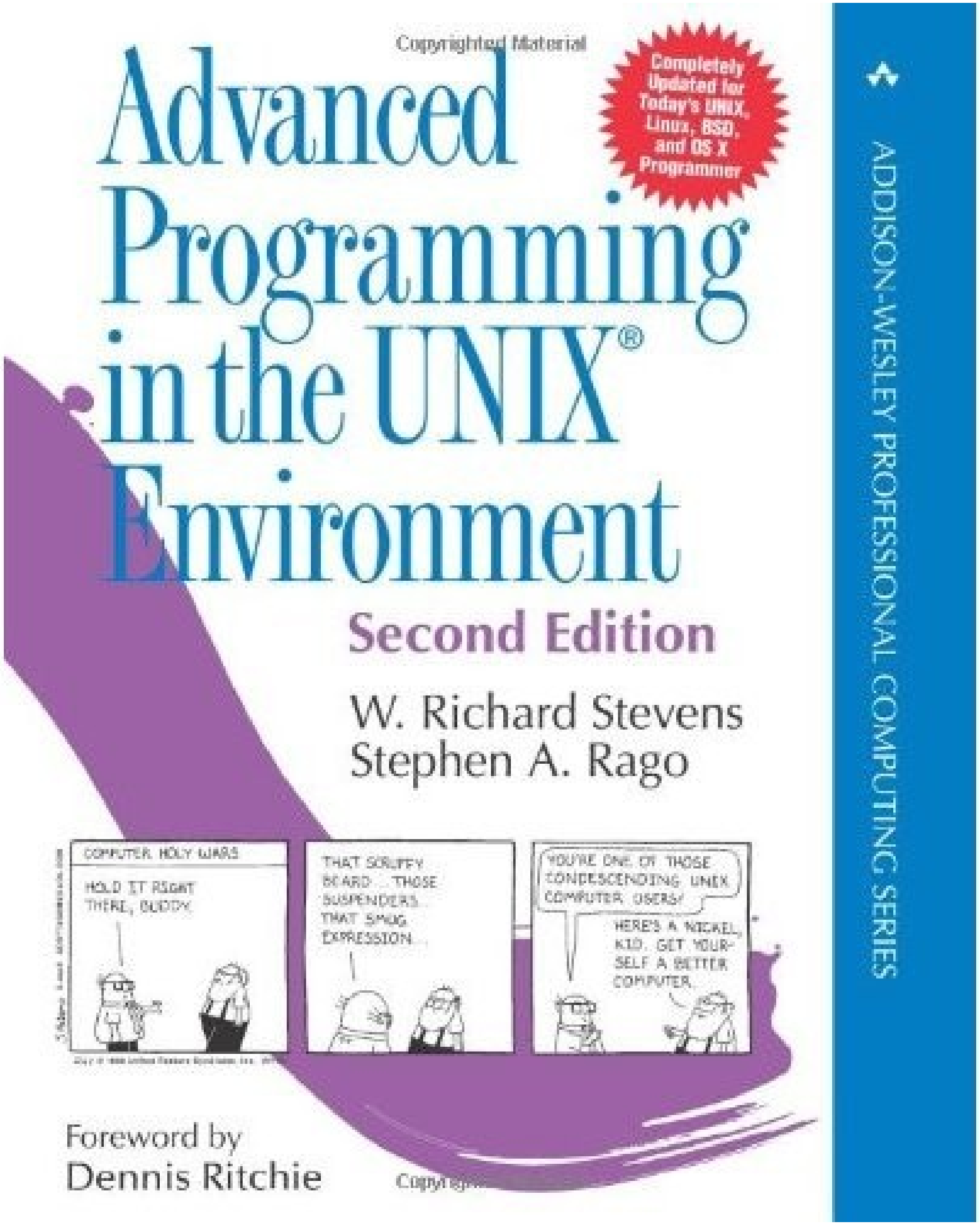
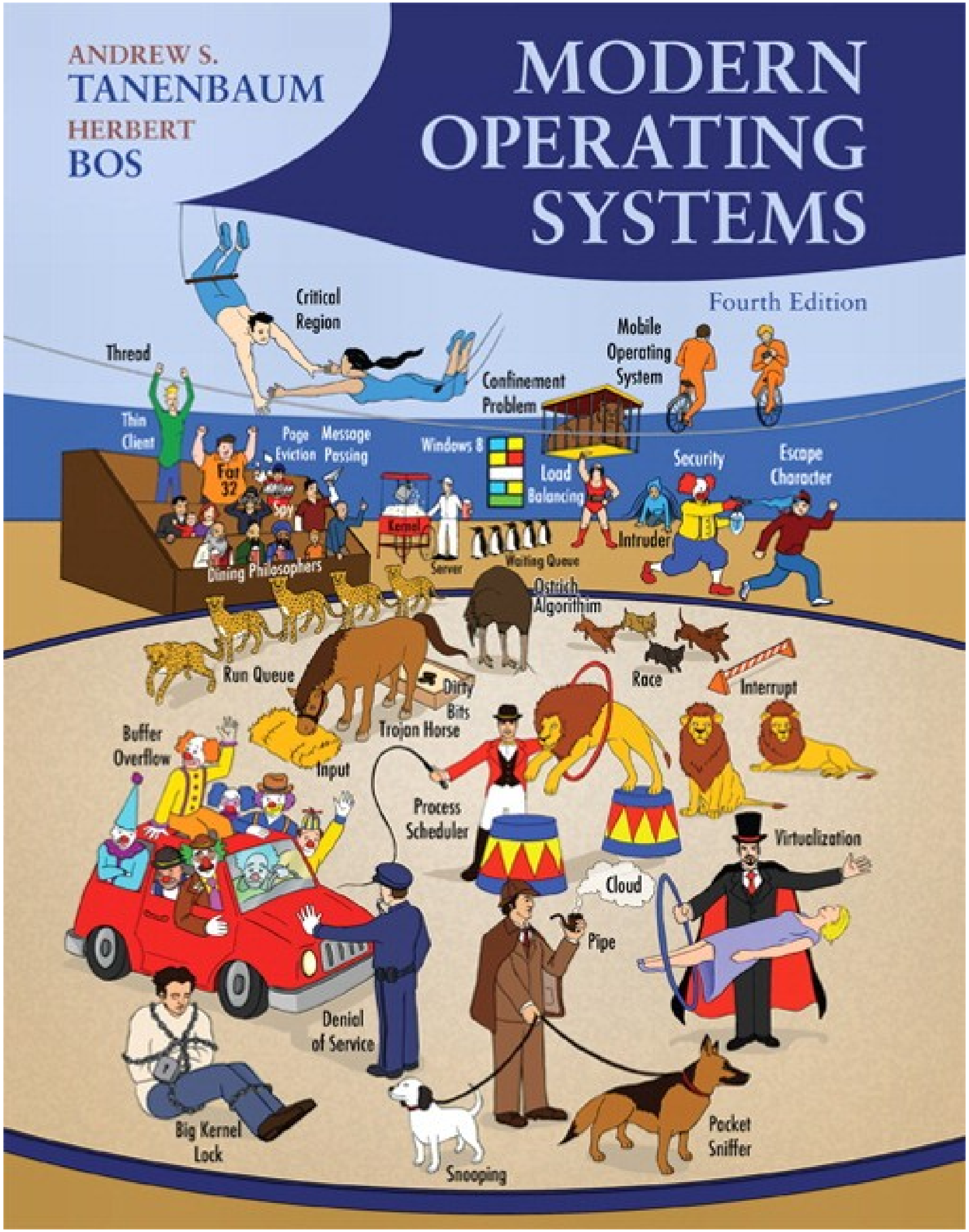
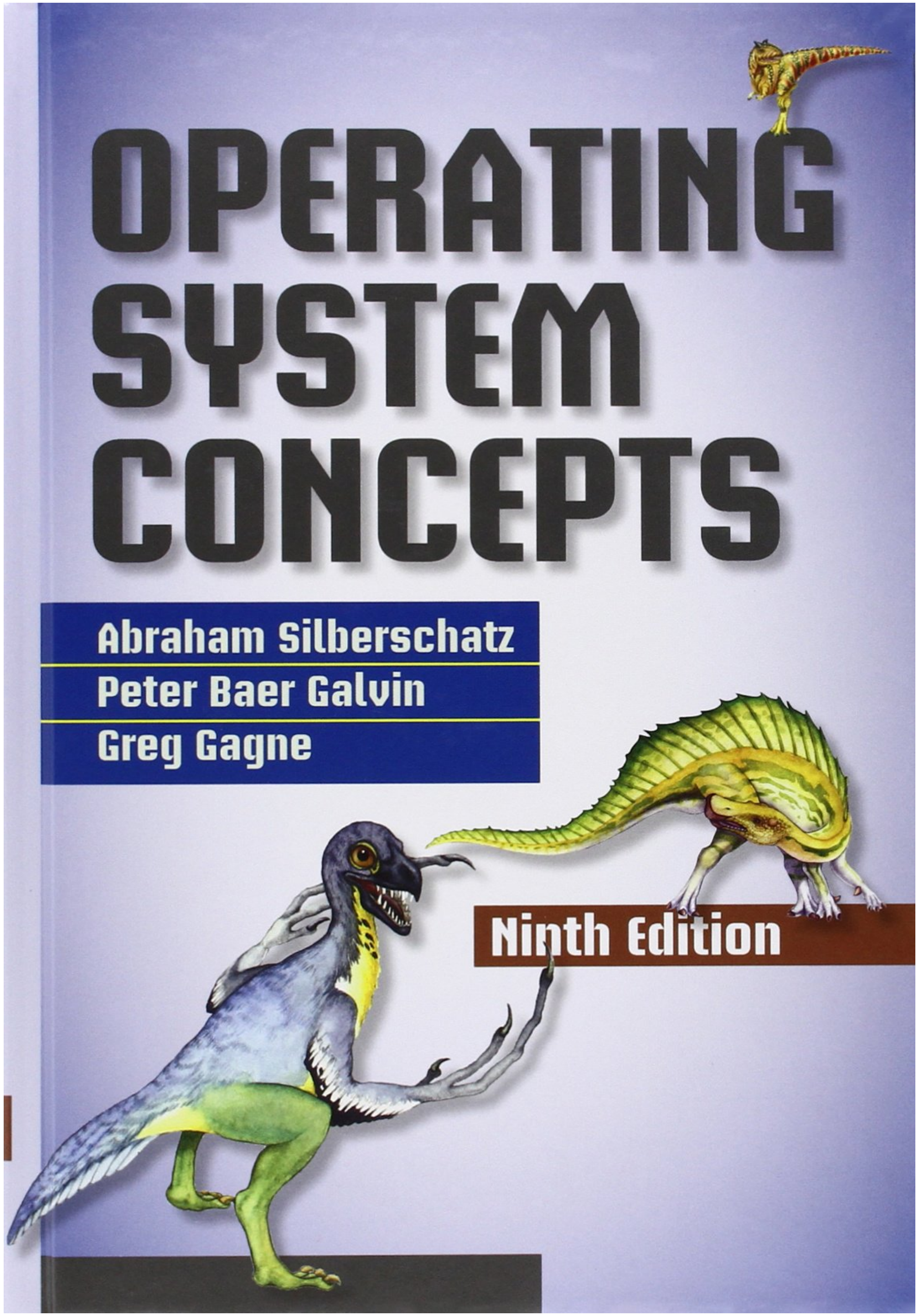

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

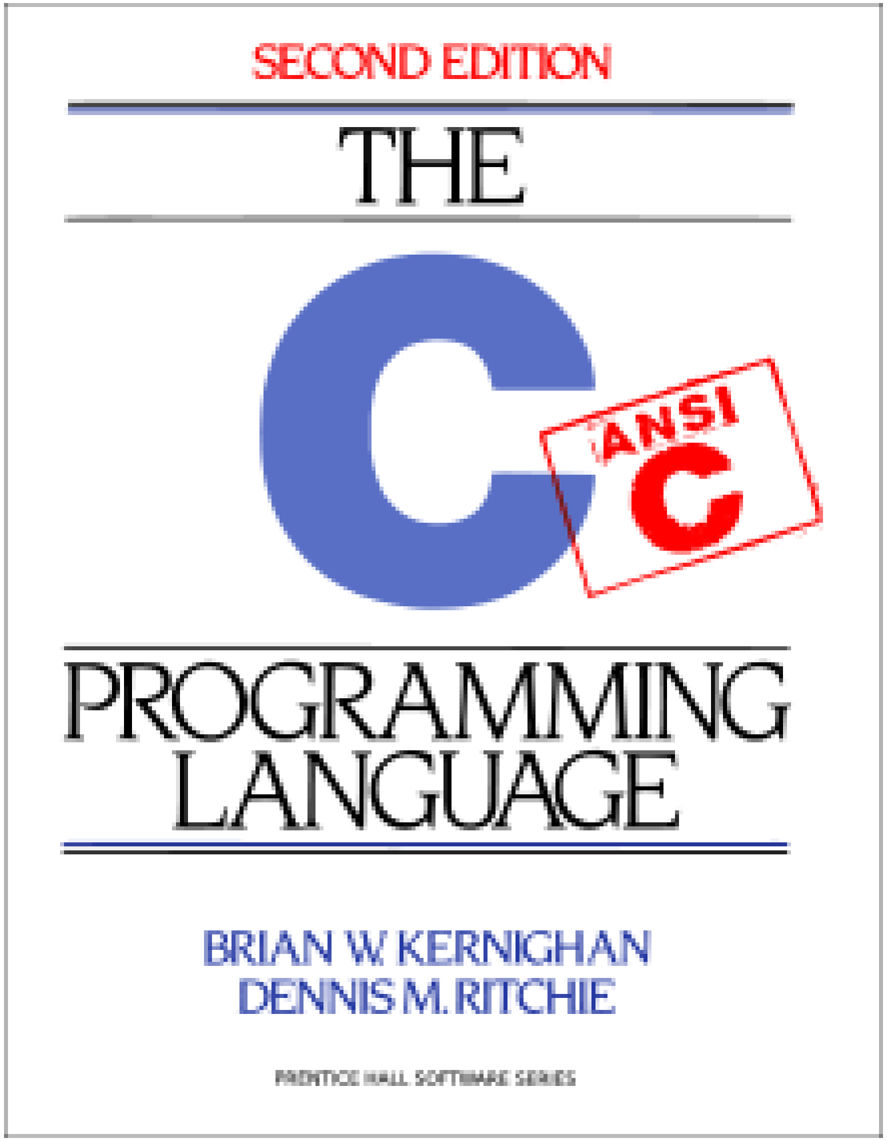
Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*


Copyrighted Material
Advanced Programming in the UNIX® Environment
Completely Updated for Today's UNIX, Linux, BSD, and OS X Programmer
Second Edition
W. Richard Stevens
Stephen A. Rago
COMPUTER HOLY WARS
HOLD IT RIGHT THERE, BUDDY.
THAT SCRUFFY BEARD... THOSE SUSPENDERS... THAT SMUG EXPRESSION...
YOU'RE ONE OF THOSE CONDESCENDING UNIX COMPUTER USERS!
HERE'S A NICKEL, KID. GET YOURSELF A BETTER COMPUTER.
Foreword by
Dennis Ritchie
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES



ANDREW S. TANENBAUM
HERBERT BOS
MODERN OPERATING SYSTEMS
Fourth Edition
Critical Region
Thread
Mobile Operating System
Confinement Problem
Thin Client
Page Eviction
Message Passing
Fat 32
Spy
Windows 8
Kernel
Load Balancing
Security
Escape Character
Dining Philosophers
Server
Waiting Queue
Intruder
Ostrich Algorithim
Run Queue
Dirty Bits
Race
Interrupt
Buffer Overflow
Trojan Horse
Input
Process Scheduler
Virtualization
Cloud
Pipe
Denial of Service
Big Kernel Lock
Snooping
Packet Sniffer


OPERATING SYSTEM CONCEPTS
Abraham Silberschatz
Peter Baer Galvin
Greg Gagne
Ninth Edition



SECOND EDITION

THE

C

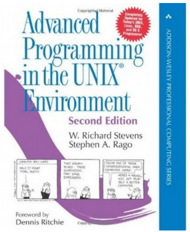
ANSI C

PROGRAMMING LANGUAGE

BRIAN W. KERNIGHAN
DENNIS M. RITCHIE

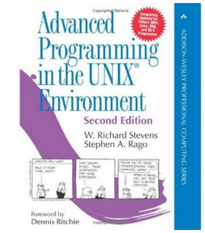
PRENTICE HALL SOFTWARE SERIES

W. Richard Stevens (APUE/2E), Fig. 8.1


Advanced Programming in the UNIX Environment
Second Edition
W. Richard Stevens
Stephen A. Rago
Foreword by Dennis Ritchie

```
#include "apue.h"
int     glob = 6;       /* external variable in initialized data */
char    buf[] = "a write to stdout\n";
int
main(void)
{
    int       var;      /* automatic variable on the stack */
    pid_t     pid;
    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
        err_sys("write error");
    printf("before fork\n");    /* we don't flush stdout */
    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid == 0) {      /* child */
        glob++;                 /* modify variables */
        var++;
    } else {
        sleep(2);               /* parent */
    }
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

W. Richard Stevens (APUE/2E), Fig. 8.1



```
$ ./a.out
a write to stdout                  Note 1
before fork                        Note 2
pid = 430, glob = 7, var = 89      child's variables were changed
pid = 429, glob = 6, var = 88      parent's copy was not changed

$ ./a.out > temp.out

$ cat temp.out
a write to stdout                  Note 3
before fork                        Note 4
pid = 432, glob = 7, var = 89      Note 5
before fork                        Note 6
pid = 431, glob = 6, var = 88      Note 7
```

Notes 1 & 3: the function write is not buffered.
Note 2: standard output (in standard I/O library) is line buffered if it's connected to a terminal device. The standard output buffer is flushed by the newline.
Notes 4-7: standard output (redirected to a file) is fully buffered. The printf (before the fork) is called once, but the line remains in the buffer when fork is called. This buffer is then copied into the child when the parent's data space is copied to the child. Both the parent and the child now have a standard I/O buffer with this line in it. The second printf (right before the exit), just appends its data to the existing buffer. When each process terminates, its copy of the buffer is finally flushed.

The basic I/O functions `open`, `read`, `write`, `close` are called unbuffered I/O functions because each `read` or `write` invokes a system call into the kernel.

The goal of the buffering provided by the standard I/O library is to use the minimum number of `read` and `write` calls.

Three types of buffering are provided:

1. Fully buffered. In this case, actual I/O takes place when the standard I/O buffer is filled. The term *flush* describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines, such as when a buffer fills, or we can call the function `fflush` to flush a stream.

2. Line buffered. In this case, the standard I/O library performs I/O when a newline character is encountered on input or output.

3. Unbuffered. The standard I/O library does not buffer the characters.

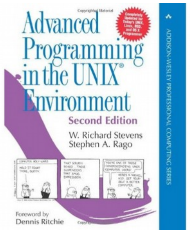
ISO C requires the following buffering characteristics:

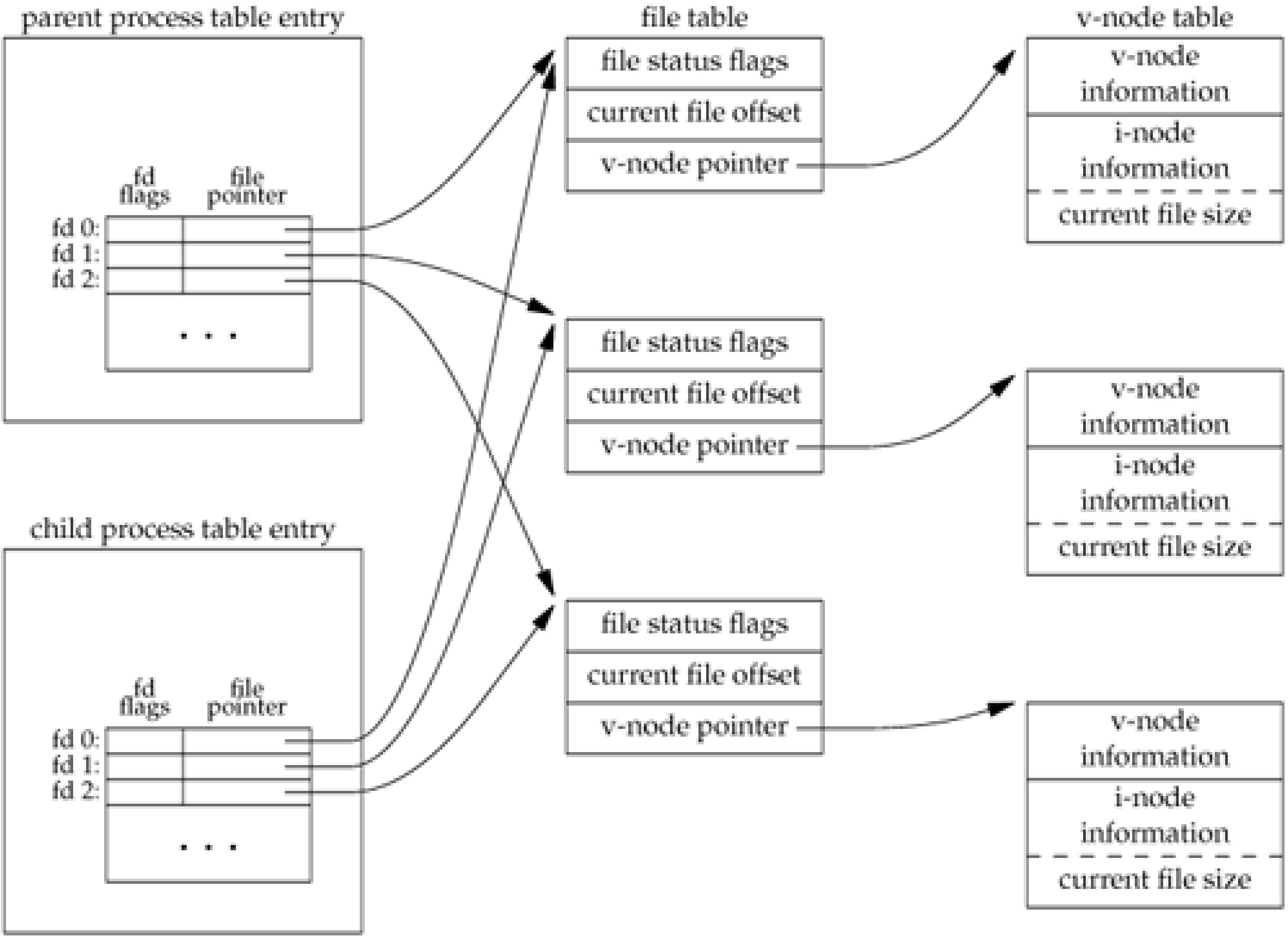
- Standard input and standard output are fully buffered, if and only if they do not refer to an interactive device.
- Standard error is never fully buffered.

Most implementations default to the following types of buffering:

- Standard error is always unbuffered.
- All other streams are line buffered if they refer to a terminal device; otherwise, they are fully buffered.

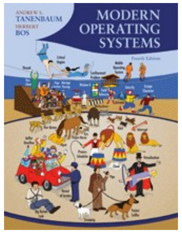
APUE, Fig. 8.2. Sharing of open files between parent and child after fork



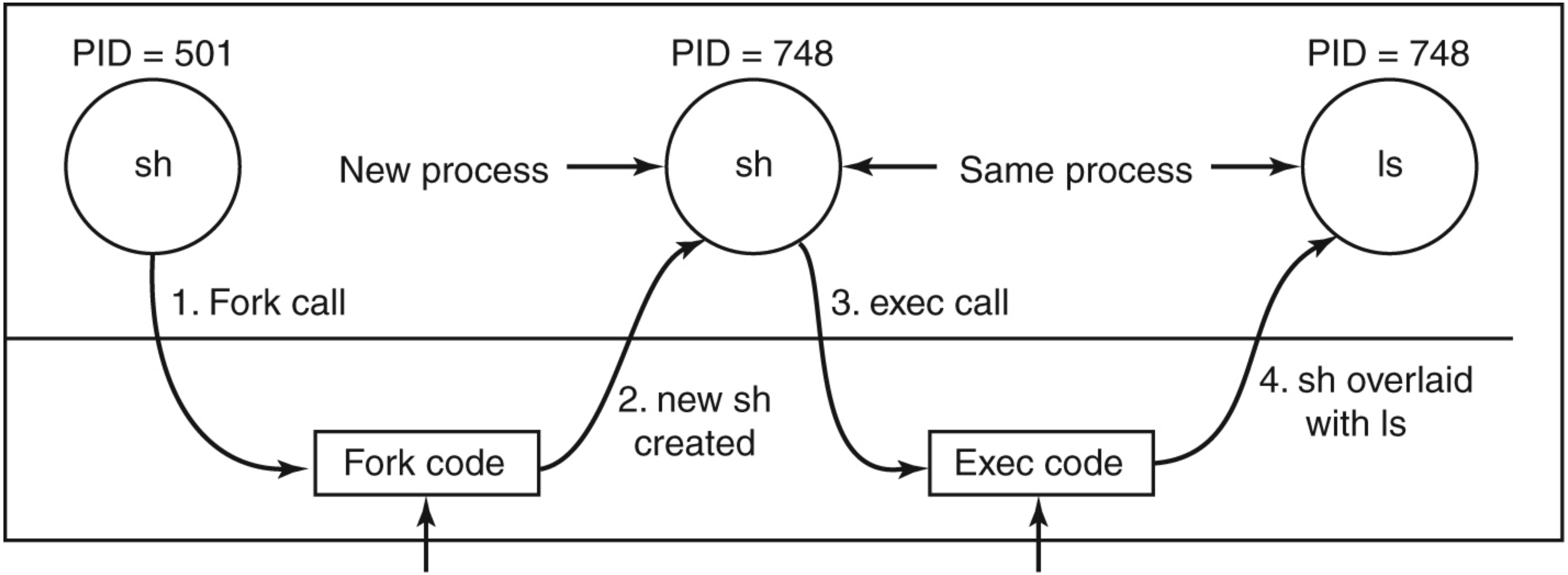


MOS4E, Fig. 1.19. A stripped-down shell

```
#define TRUE 1

while (TRUE) {                                  /* repeat forever */
      type_prompt( );                           /* display prompt on the screen */
      read_command(command, parameters);        /* read input from terminal */

      if (fork( ) != 0) {                       /* fork off child process */
          /* Parent code. */
          waitpid(−1, &status, 0);              /* wait for child to exit */
      } else {
          /* Child code. */
          execve(command, parameters, 0);       /* execute command */
      }
}
```



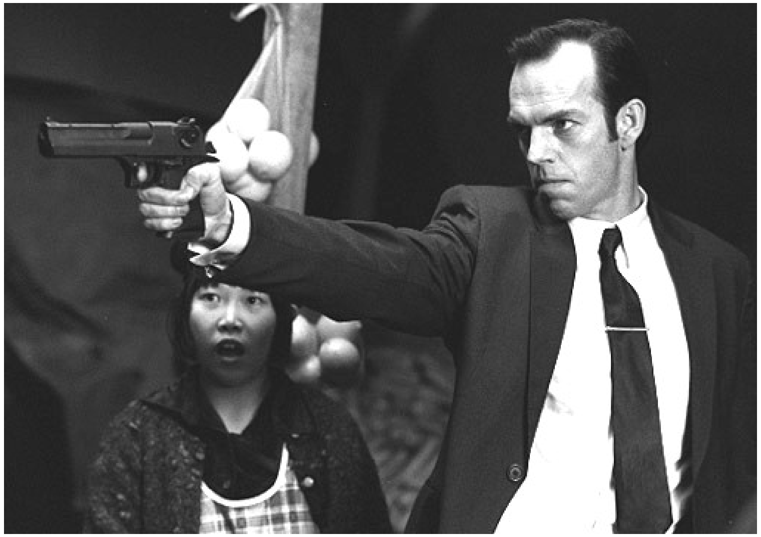
MOS4E, Fig. 10.8. The steps in executing the command *ls* typed to the shell

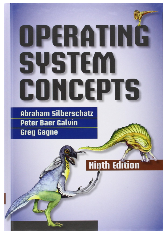


Allocate child's task structure
Fill child's task structure from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

Si quitar la libertad de hacer exec ...



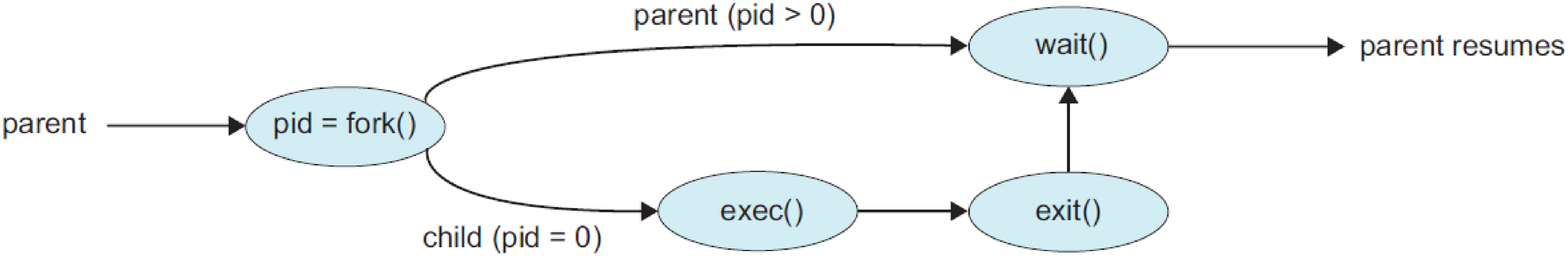


OPERATING SYSTEM CONCEPTS

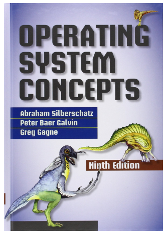
Abraham Silberschatz
Peter Baer Galvin
Greg Gagne

Ninth Edition

OSC/9E, Fig. 3.10. Process creation using the fork() system call

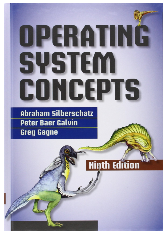


OSC/9E,Fig.3.11.Creating a separate process using the Win32 API.



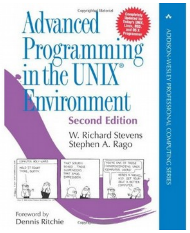
```
#include <stdio.h>
#include <windows.h>

int main(VOID)
{
  STARTUPINFO si;
  PROCESS_INFORMATION pi;

  /* allocate memory */
  ZeroMemory(&si, sizeof(si));
  si.cb = sizeof(si);
  ZeroMemory(&pi, sizeof(pi));

  /* create child process */
  if (!CreateProcess(NULL,                     /* use command line */
    "C:\\WINDOWS\\system32\\mspaint.exe",     /* command */
    NULL,                                     /* don't inherit process handle */
    NULL,                                     /* don't inherit thread handle */
    FALSE,                                    /* disable handle inheritance */
    0,                                        /* no creation flags */
    ...
...
    NULL,                                       /* use parent's environment block */
    NULL,                                       /* use parent's existing directory */
    &si,
    &pi))
  {
    fprintf(stderr, "Create Process Failed");
    return -1;
  }
  /* parent will wait for the child to complete */
  WaitForSingleObject(pi.hProcess, INFINITE);
  printf("Child Complete");

  /* close handles */
  CloseHandle(pi.hProcess);
  CloseHandle(pi.hThread);
}
```

OSC/9E, Fig. 3.9. Process creation



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t  pid;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
           fprintf(stderr, "Fork Failed");
           return 1;
    }
    else if (pid == 0) { /* child process */
           execlp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
           /* parent will wait for the child to complete */
           wait (NULL);
           printf ("Child Complete");
    }
    return 0;
}
```

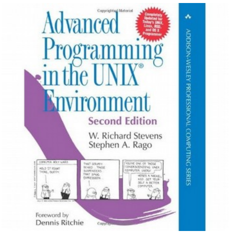
W. Richard Stevens (APUE/2E), 8.10 exec Functions



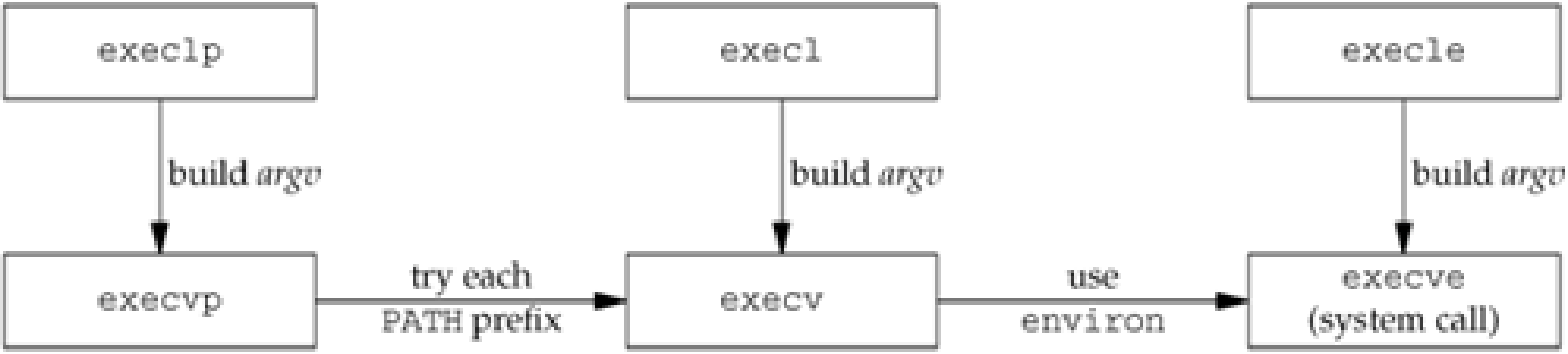
```
#include <unistd.h>


int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv []);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0,  char *const envp[] */ );

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

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv []);


                         All six return: -1 on error, no return on success
```

W. Richard Stevens (APUE/2E), Fig. 8.14 Differences among the six exec functions

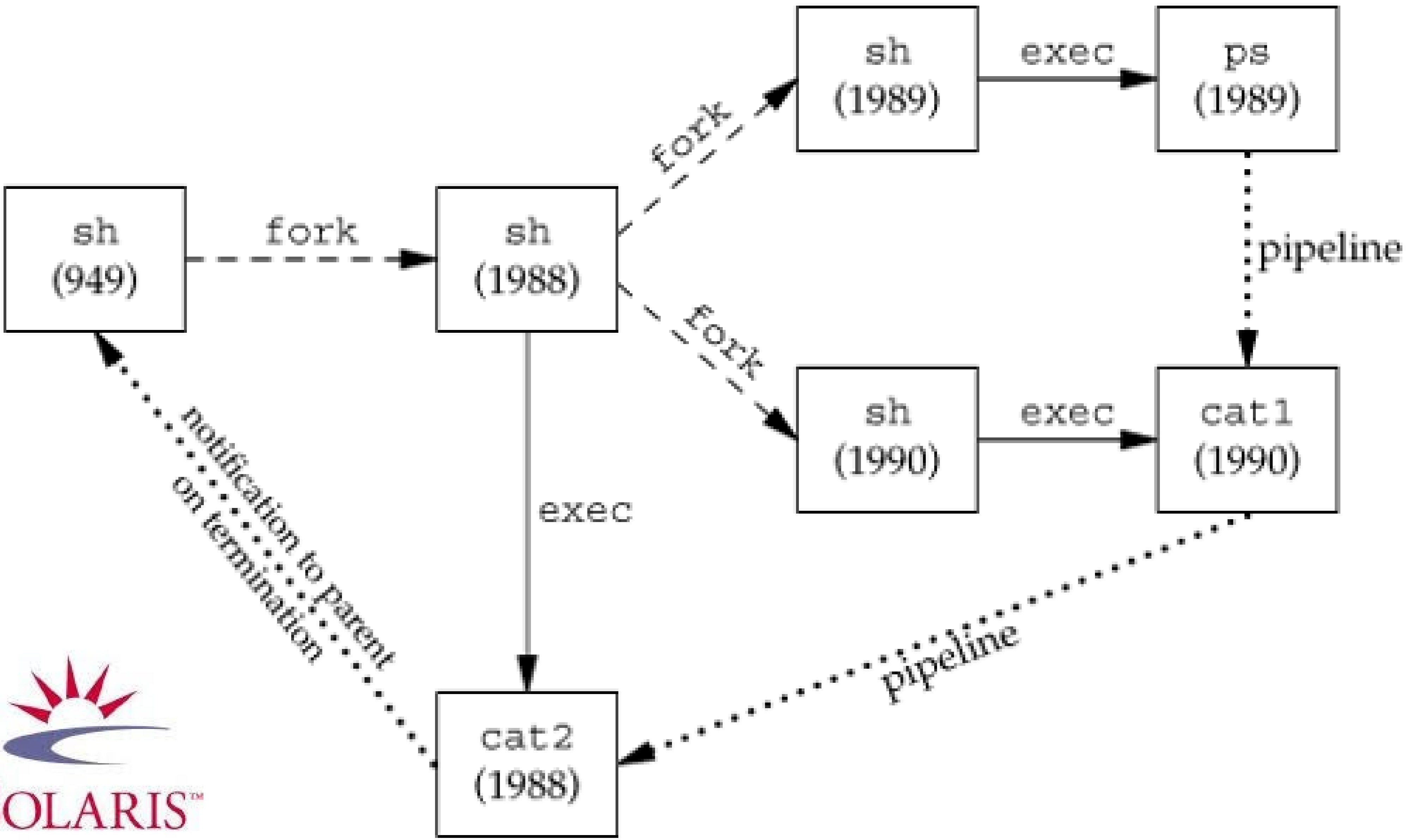
Function	*pathname*	*filename*	Arg list	*argv* [ ]	environ	*envp* [ ]
execl	●		●		●	
execlp		●	●		●	
execle	●		●			●
execv	●			●	●	
execvp		●		●	●	
execve	●			●		●
(letter in name)		p	l	v		e



APUE/2E, Fig. 8.15. Relationship of the six exec function

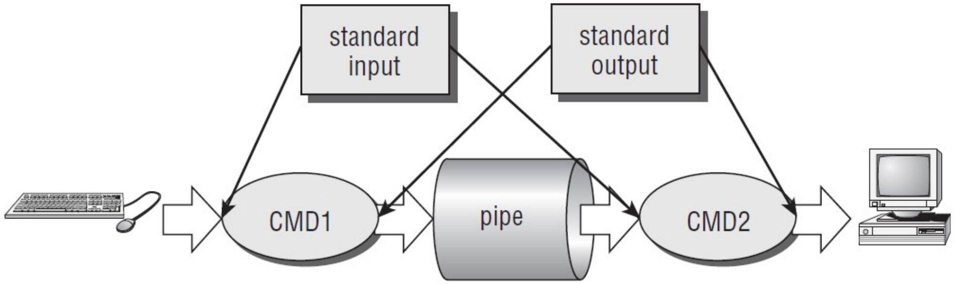


APUE/2E, Fig.9.9. (Solaris 9) Processes in the pipeline ps | cat1 | cat2 when invoked by Bourne shell



```
$ echo $SHELL
/bin/bash

$ ls -l /bin/bash
-rwxr-xr-x 1 root root 1113504 abr  4 13:30 /bin/bash

$ which cat
/bin/cat

$ tee < /bin/cat cat1 > cat2

$ ls -l cat?
-rw-rw-r-- 1 vk vk 35064 set  9 13:22 cat1
-rw-rw-r-- 1 vk vk 35064 set  9 13:22 cat2

$ chmod a+x cat?

$ ls -l cat?
-rwxrwxr-x 1 vk vk 35064 set  9 13:22 cat1
-rwxrwxr-x 1 vk vk 35064 set  9 13:22 cat2
```

```
$ echo $$
17737

$ ps -o pid,ppid,pgid,sid,comm | ./cat1 | ./cat2
  PID  PPID  PGID   SID COMMAND
17737 16240 17737 17737 bash
18250 17737 18250 17737 ps
18251 17737 18250 17737 cat1
18252 17737 18250 17737 cat2
```



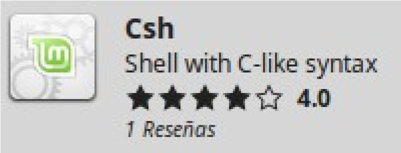
BLP4E

```
$ which sh
/bin/sh

$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 ago  6 20:02 /bin/sh -> dash
```

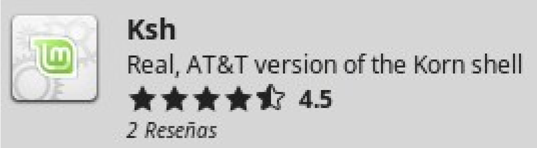
Debian Almquist Shell

```
$ which dash
/bin/dash

$ ls -l /bin/dash
-rwxr-xr-x 1 root root 121432 ene 25  2018 /bin/dash

$ dash

$ ps -o pid,ppid,pgid,sid,comm | ./cat1 | ./cat2
  PID  PPID  PGID   SID COMMAND
17737 19839 17737 17737 bash
18781 17737 18781 17737 dash
18814 18781 18814 17737 ps
18815 18781 18814 17737 cat1
18816 18781 18814 17737 cat2
```

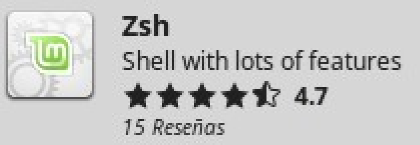
```
$ echo $0
dash

$ ps $$
  PID TTY      STAT   TIME COMMAND
18781 pts/5    S      0:00 dash

$ cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash

$ exit

$ echo $0
bash
```



Csh
Shell with C-like syntax
★★★★☆ 4.0
1 Reseñas

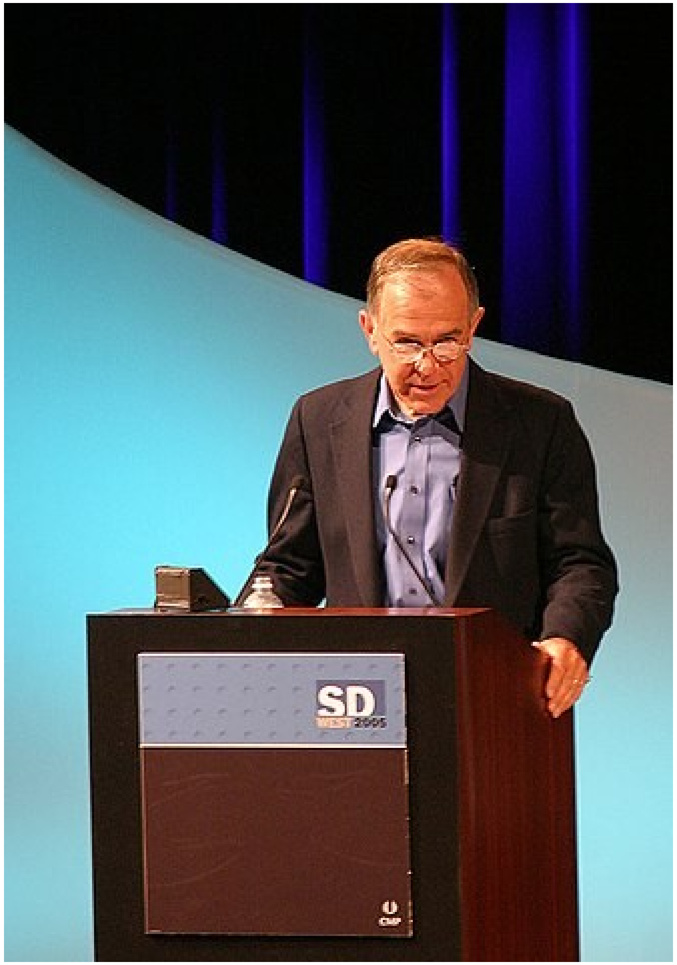


Ksh
Real, AT&T version of the Korn shell
★★★★⯪ 4.5
2 Reseñas



Zsh
Shell with lots of features
★★★★⯪ 4.7
15 Reseñas

...



Steve Bourne

```
$ date
dom set  9 13:45:45 -05 2018

$ date > foo

$ ls -l foo
-rw-rw-r-- 1 vk vk 29 set  9 13:46 foo

$ cat foo
dom set  9 13:46:39 -05 2018

$ > foo

$ ls -l foo
-rw-rw-r-- 1 vk vk 0 set  9 13:50 foo
```

Example 4.35 (USP-CCT, Ch.4)

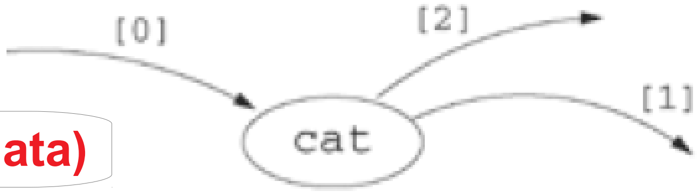
```
$ man cat
CAT(1)                          User Commands                         CAT(1)

NAME
       cat - concatenate files and print on the standard output

SYNOPSIS
       cat [OPTION]... [FILE]...

DESCRIPTION
       Concatenate FILE(s) to standard output.

       With no FILE, or when FILE is -, read standard input.
...
```

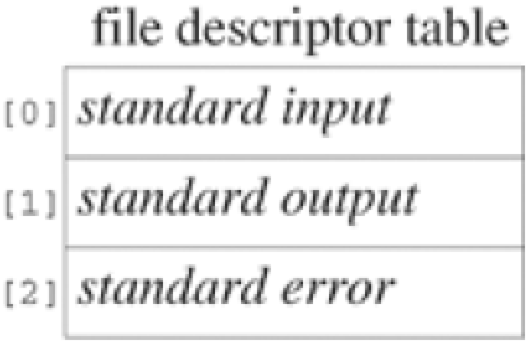
```
$ cat
Hola
Hola
$
```

from standard input line by line

to standard output

Ctrl-D (end-of-data)





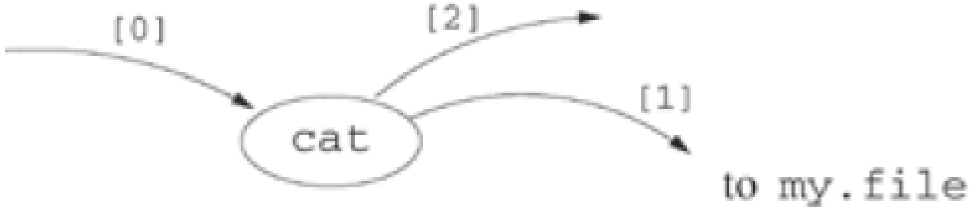
file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*

Example 4.35 (USP-CCT, Ch.4)

```
$ cat > my.file
Hola
$
```

After redirection by shell and before the cat-process execution



file descriptor table

[0]	*standard input*
[1]	*write to* my.file
[2]	*standard error*

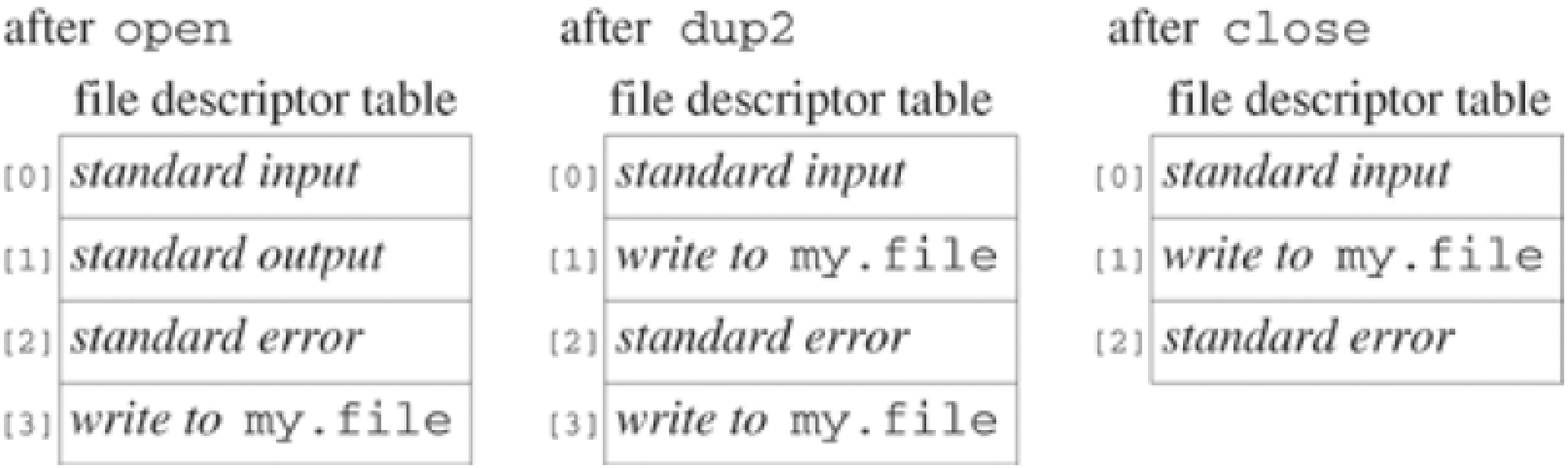
```
$ ls -l my.file
-rw-rw-r-- 1 vk vk 5 set  9 14:50 my.file
```

Program 4.18 (USP-CCT, Ch.4)

```
$ cat -n usp_prog4.18.c | expand
     1  #include <fcntl.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  #define CREATE_FLAGS (O_WRONLY|O_CREAT|O_APPEND)
     6  #define CREATE_MODE (S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH) /* rw-r--r-- */
     7  int
     8  main(void) {
     9      int fd;
    10
    11      fd = open("my.file",CREATE_FLAGS,CREATE_MODE);
    12      if (fd == -1) {
    13          perror("Failed to create my.file");
    14          return 1;
    15      }
    16      if (dup2(fd,STDOUT_FILENO) == -1) {
    17          perror("Failed to redirect standard output");
    18          return 2;
    19      }
    20      if (close(fd) == -1) {
    21          perror("Failed to close the file");
    22          return 3;
    23      }
    24      if (write(STDOUT_FILENO,"Ok",2) == -1) {
    25          perror("Failed in writing to file");
    26          return 4;
    27      }
    28      return 0;
    29  }
```

Program 4.18 (USP-CCT, Ch.4)

```
$ gcc -o usp_prog4.18 usp_prog4.18.c

$ ./usp_prog4.18

$ ls -l my.file
-rw-r--r-- 1 vk vk 2 set  9 15:43 my.file

$ cat my.file
Ok$
```



after open

	file descriptor table
[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	*write to* my.file

after dup2

	file descriptor table
[0]	*standard input*
[1]	*write to* my.file
[2]	*standard error*
[3]	*write to* my.file

after close

	file descriptor table
[0]	*standard input*
[1]	*write to* my.file
[2]	*standard error*

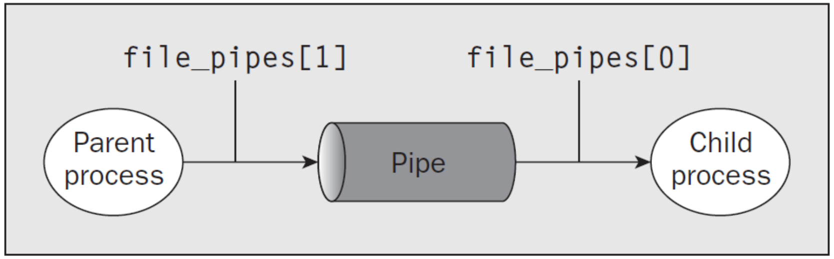


Figure 13-2

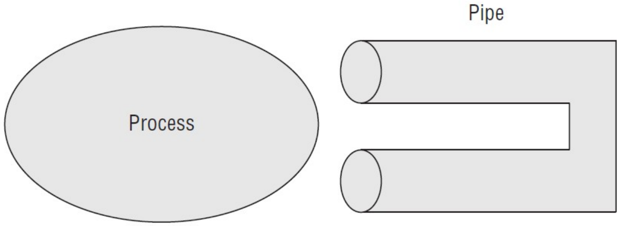


Figure 13-3

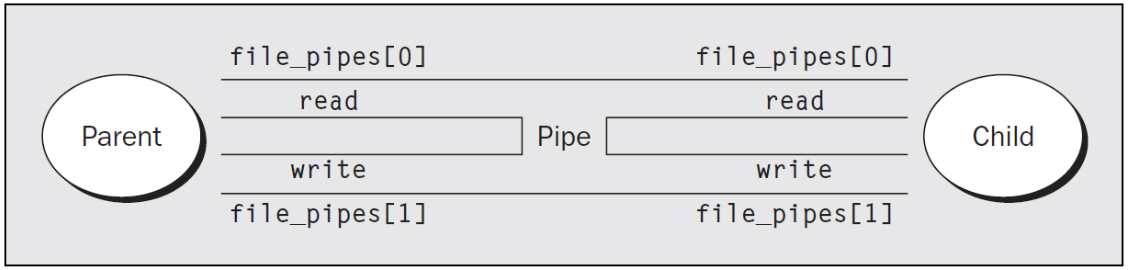


Figure 13-4

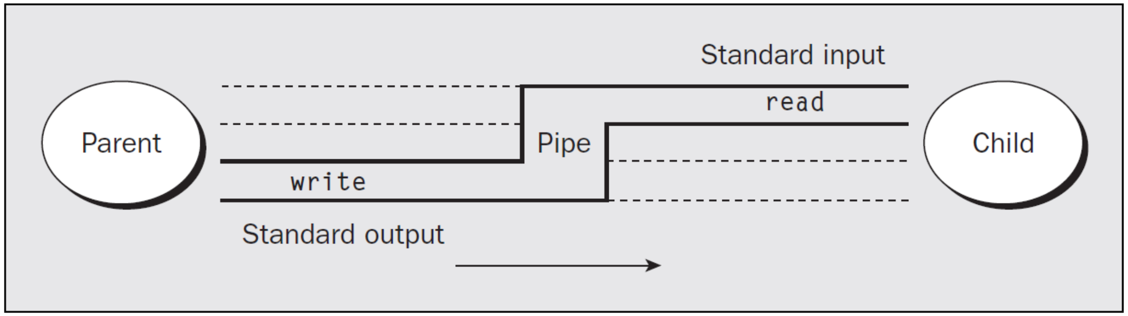


Figure 13-5

BLP4E

Preparación de un *pipe* entre dos procesos

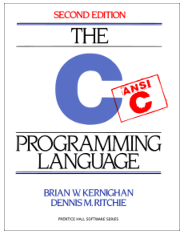
File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

```
#define STD_INPUT 0
#define STD_OUTPUT 1

pipeline(process1, process2)
char *process1, *process2;
{
    int fd[2];

    pipe(&fd[0]);
    if (fork() != 0) {       /* El proceso padre */
        close(fd[0]); close(STD_OUTPUT); dup(fd[1]);
        close(fd[1]);
        execl(process1, process1, 0);
    } else {                 /* El proceso hijo ejecuta esta parte */
        close(fd[1]); close(STD_INPUT); dup(fd[0]);
        close(fd[0]);
        execl(process2, process2, 0);
    }
}
```

```
dup2(fd[1],STD_OUTPUT);
```

```
dup2(fd[0],STD_INPUT);
```

Brian W. Kernighan, Dennis M. Ritchie, El Lenguaje de Programación C, Segunda edición, Sección 7.1 del Capítulo 7

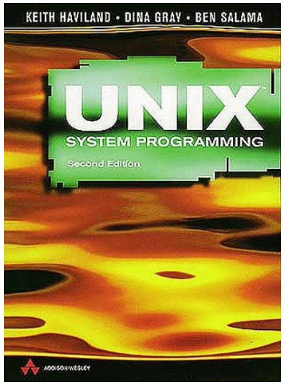


SECOND EDITION
THE
C
ANSI C
PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

```
$ cat capital.c
#include <stdio.h>
#include <ctype.h>
#include <fcntl.h>

#define STD_IN  0

int main(int argc, char *argv[])
{
  int c, fd;

/* si está dado el parámetro, lo consideramos como el nombre del archivo, lo abrimos para
   obtener su descriptor. Consideramos este archivo como la entrada estándar. */

  if (argc == 2) { fd = open(argv[1], O_RDONLY); close(STD_IN); dup(fd); }

  while ((c = getchar()) != EOF) putchar(toupper(c));
  return 0;
}

$ capital < capital.c      # using standard input
$ capital capital.c
$ cat mcap.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
  int np, i, status;
  pid_t childpid;

  if (argc == 1) { printf("El formato es: %s <lista-de-archivos>\n", argv[0]);
                   exit(1); }

  np = argc - 1;        /* número de procesos a crear */
  for ( i=1; i <= np; i++ ) {
    if ( (childpid=fork()) == -1 ) { perror("fork fallo...\n"); exit(2); }
    else if ( childpid == 0 ) { /* es el hijo */
             execl("./capital", "capital", argv[i], NULL);
             perror("execl fallo...\n");
             exit(3);
         }
  }
  while (np--) wait(&status);
  return 0;
}
$ mcap capital.c capital.c capital.c
$ cat mcapsort.c
...
  int np, i, status, fd[2];
...
  pipe(fd);
  np = argc - 1;        /* número de procesos a crear */

  for (...) {
    if ( (childpid=fork()) == -1 ) { perror("fork fallo...\n"); exit(2); }
    else if ( childpid == 0 ) { /* es el hijo */
             dup2(fd[1], STDOUT_FILENO); close(fd[0]); close(fd[1]);
             execl("./capital", "capital", argv[i], NULL);
             perror("execl fallo...\n"); exit(3);
         }
  }
  while (np--) wait(&status);
  if ( (childpid=fork()) == -1 ) { perror("fork para sort fallo...\n"); exit(4); }
  else if ( childpid == 0 ) { /* el hijo para sort */
           dup2(fd[0], STDIN_FILENO); close(fd[0]); close(fd[1]);
           execl("/usr/bin/sort", "sort", NULL);
           perror("execl para sort fallo...\n");
           exit(5);
       }
  close(fd[0]); close(fd[1]); /* cerrar la tuberia */
  wait(&status); /* esperar la terminacion de sort */
  return 0;
}
```

Examen 1 (2000-1), pregunta 1:

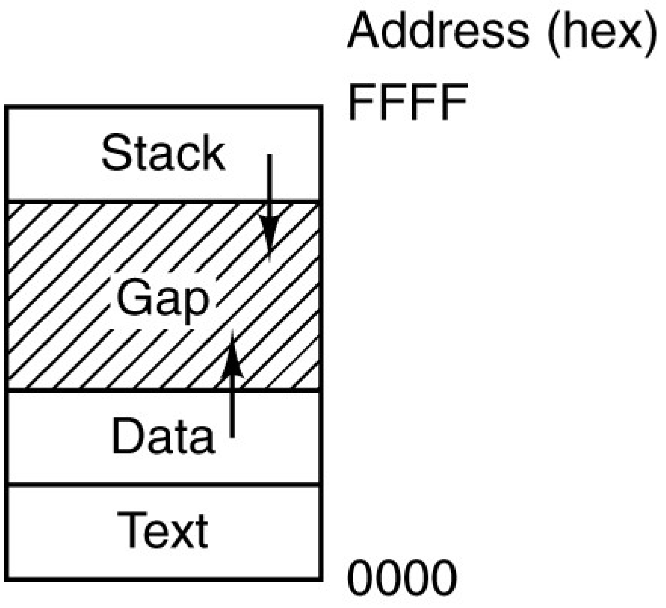
Cuando este programa termina satisfactoriamente, muestra una propiedad importante de la implementación de una tubería. ¿Cuál es esta propiedad y cuál sería el mensaje faltante en la función `alrm_action()`? (Keith Haviland, Ben Salama. *UNIX System Programming*, AW, 1997)



```
#include <signal.h>

int count;
int alrm_action();

main()
{
    int p[2];
    char c = ’x’;

    if (pipe(p) < 0) { perror(”pipe call”); exit(1); }
    signal(SIGALRM, alrm_action);
    for (count = 0;;) {
        alarm(20);          /* unsigned int alarm(unsigned int seconds) */
        write(p[1],&c,1);
        alarm(0); /* Alarm requests are not stacked; successive calls reset the alarm clock. */
                  /* If the argument is 0, any alarm request is canceled. */
        if ((++count % 1024) == 0)
            printf(”%d characters in pipe\n”, count);
    }
}
alrm_action()
{
    printf(...);
    exit(0);
}
```

Examen 1 (2001-1), pregunta 3:

¿Cuál será el resultado de ejecución del siguiente programa?

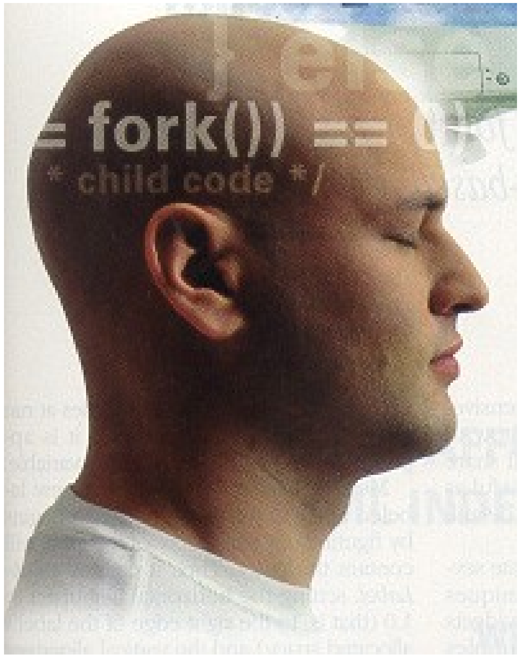


```
$ cat pointerex.c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
{
    int pid;
    int value=5;
    int *pval = &value;

    if ( (pid = fork() ) != 0) {
        printf("parent: value = %d, *(pval) = %d\n", value, *pval);
        value = 7;
        printf("parent: value = %d, *(pval) = %d\n", value, *pval);
    } else {
        value = 8;
        printf("child: value = %d, *(pval) = %d\n", value, *pval);
        value = 9;
        printf("child: value = %d, *(pval) = %d\n", value, *pval);
  }
}
$ gcc pointerex.c
$ a.out
...
```

Examen 1 (2002-2), pregunta 4:

¿Cuál serán los resultados de ejecución del siguiente programa?

```
$ cat exam1-2002-2.c
#include <stdio.h>
#include <stlib.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    int i, n, status;
    char str_n[8];

    if (argc == 1) exit(1);
    n = atoi(argv[1]);
    sprintf(str_n, "%d", --n);
    if (!n) execl("/bin/ps","ps","-l",NULL);
    if ( fork() && fork() )
        for (i=0; i<2; i++) wait(&status);
    else execl("./exam1", "exam1", str_n, NULL);
    return 0;
}
$ gcc -o exam1 exam1-2002-2.c
$ exam1
...
$ exam1 1
...
$ exam1 2
...
$ exam1 4
...
```



Examen 1 (2003-1), pregunta 4:

¿Cuál serán los resultados de ejecución del siguiente programa?

```
#include <stdio.h>
#include <unistd.h>


int main(void)
{
    int status;

    if ( (fork() && fork()) || (fork() && fork()) || fork() ) {
        while( wait(&status) != -1 );
        printf("pid = %d, ppid = %d\n", getpid(), getppid());
    }
    else {
        printf("pid = %d, ppid = %d\n", getpid(), getppid());
        sleep(1);
        execl("/bin/ps", "ps", "-l", NULL);
    }
    return 0;
}
```

M.S. Escher, *Waterfall*

Examen 2 (2003-2), pregunta 1:

¿Cuál serán los resultados de ejecución del siguiente programa?

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
    int i=0, status;

    fprintf(stderr, "pid = %ld, ppid = %ld.\n",
            (long)getpid(), (long)getppid());
    while ( (!fork() || !fork()) && i<2 ) {
        fprintf(stderr, "pid = %ld, ppid = %ld: %d\n",
                (long)getpid(), (long)getppid(), i);
        if ( i % 2 ) break;
        i++;
    }
    sleep(1);
    while(waitpid(&status) != -1);
    fprintf(stderr, "pid = %ld, ppid = %ld: terminated.\n",
            (long)getpid(), (long)getppid());
    return 0;
}
```

Examen 1 (2004-1), pregunta 1:

¿Cuál serán los resultados de ejecución del siguiente programa?

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

int main(void)
{
    pid_t great_grandfather, ppid, bill, bill_vol2;
    int status, pipefd[2];

    pipe(pipefd);
    dup2(pipefd[0],STDIN_FILENO); dup2(pipefd[1],STDOUT_FILENO);
    close(pipefd[0]); close(pipefd[1]);
    fprintf(stderr, "pid = %ld, ppid = %ld.\n",
            (long)(great_grandfather=getpid()), (long)getppid());
    if ( (!fork() || fork()) && (fork() == !(bill=fork())) ) {
        fprintf(stderr, "pid = %ld, ppid = %ld: true and bill = %ld\n",
                (long)getpid(), (long)(ppid=getppid()), (long)bill);
        sleep(1);
        if (ppid != great_grandfather) {
            write(1, &bill, sizeof(ppid));
            dup2(STDERR_FILENO,STDOUT_FILENO);
            execl("/bin/ps", "ps", "-l", NULL);
        } else {
        ...
        ...
        } else {
            bill_vol2 = bill;
            read(0, &bill, sizeof(ppid));
            fprintf(stderr, "pid = %ld, ppid = %ld: father of %ld did ps\n",
                    (long)getpid(), (long)getppid(), bill);
            kill(bill, SIGTERM);
            fprintf(stderr, "pid = %ld: terminated.\n", bill);
            kill(bill_vol2, SIGTERM);
            fprintf(stderr, "pid = %ld: terminated.\n", bill_vol2);
            fprintf(stderr, "pid = %ld, ppid = %ld: finished.\n",
                    (long)getpid(), (long)getppid());
            exit(0);
        }
    }
    close(0); close(1);
    sleep(3);
    while (waitpid(-1, &status, 0) != -1);
    fprintf(stderr, "pid = %ld, ppid = %ld: finished.\n",
            (long)getpid(), (long)getppid());
    return 0;
}
```