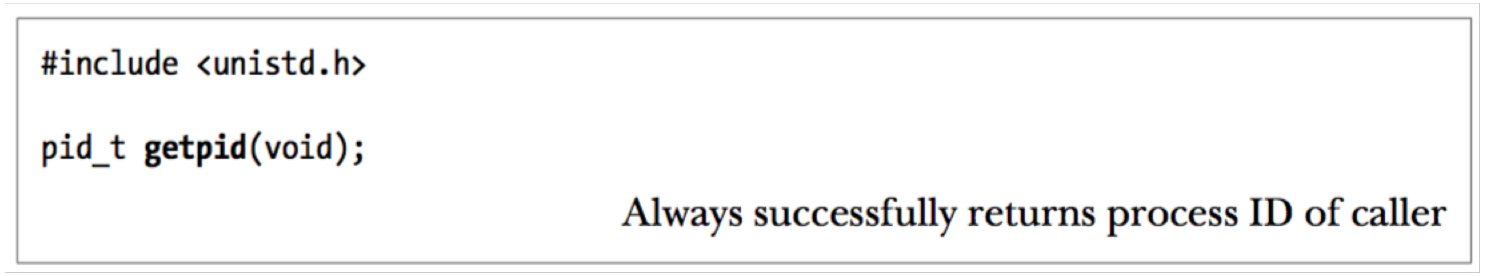
CS 3113

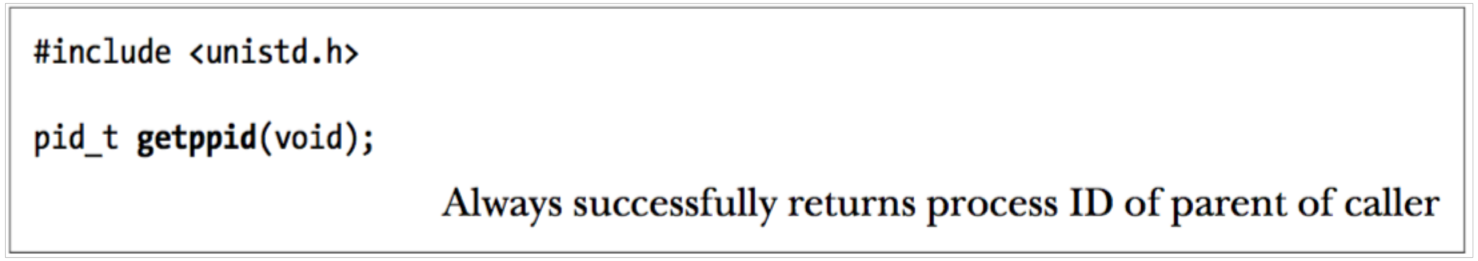
Processes

Get pid



```
#include <unistd.h>

pid_t getpid(void);
```

Always successfully returns process ID of caller



```
#include <unistd.h>

pid_t getppid(void);
```

Always successfully returns process ID of parent of caller

Virtual Memory

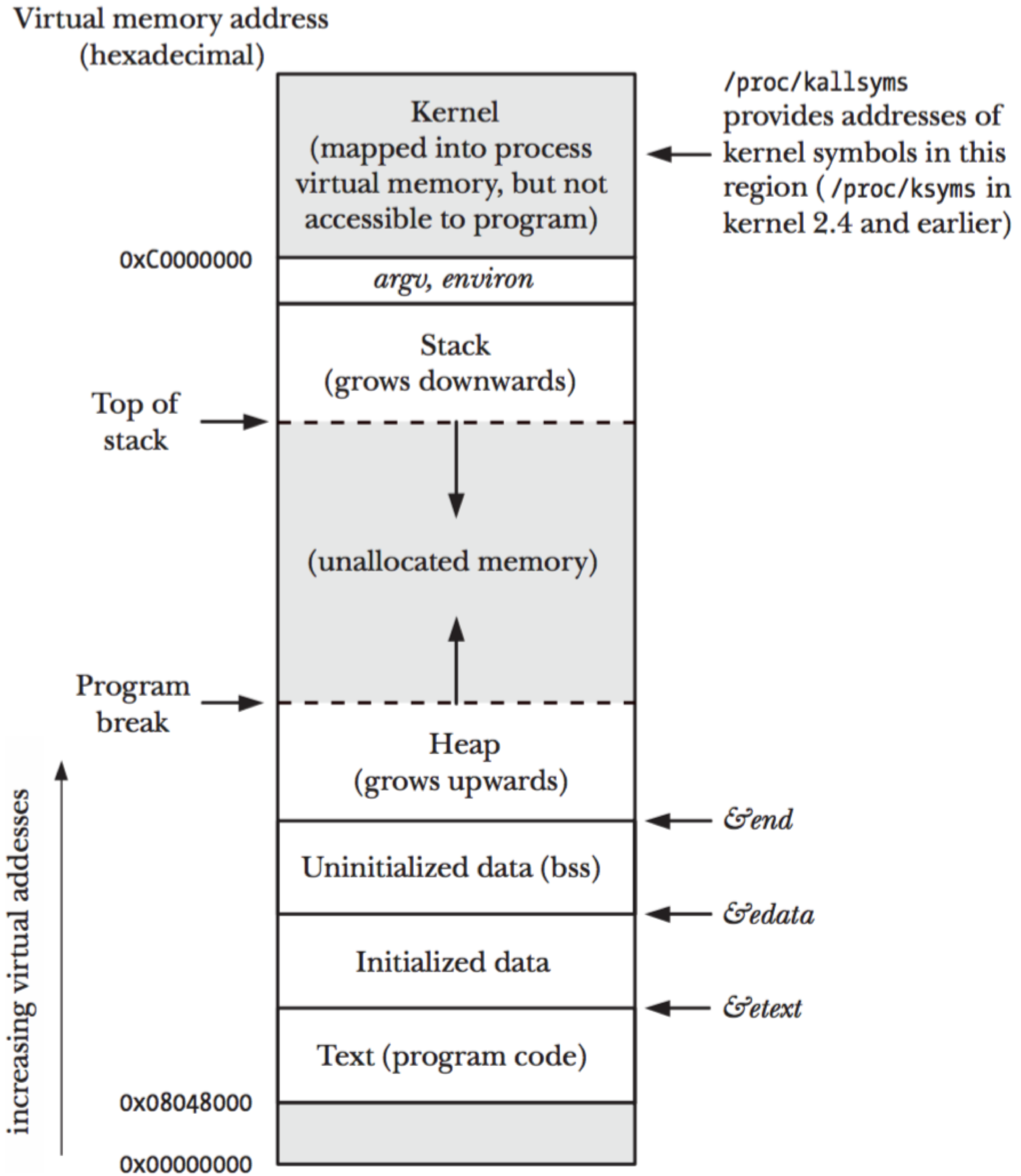
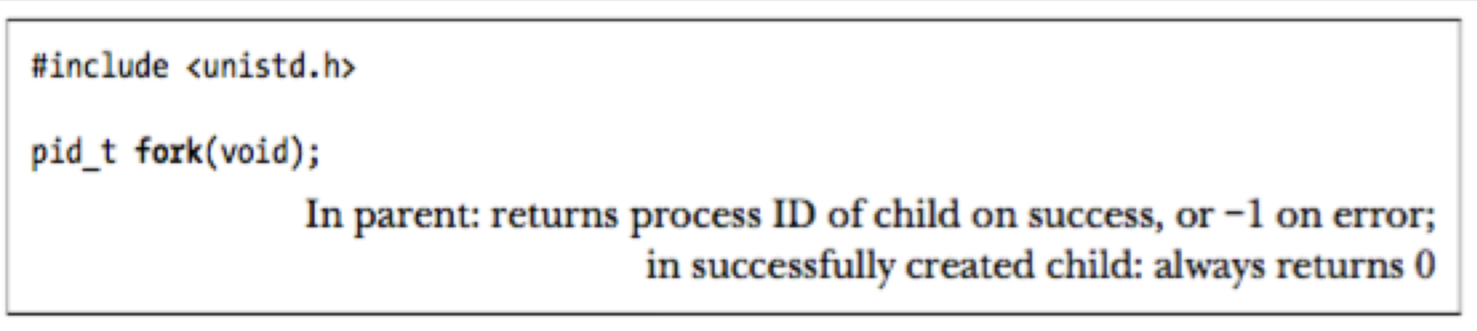


Figure 6-1: Typical memory layout of a process on Linux/x86-32

fork

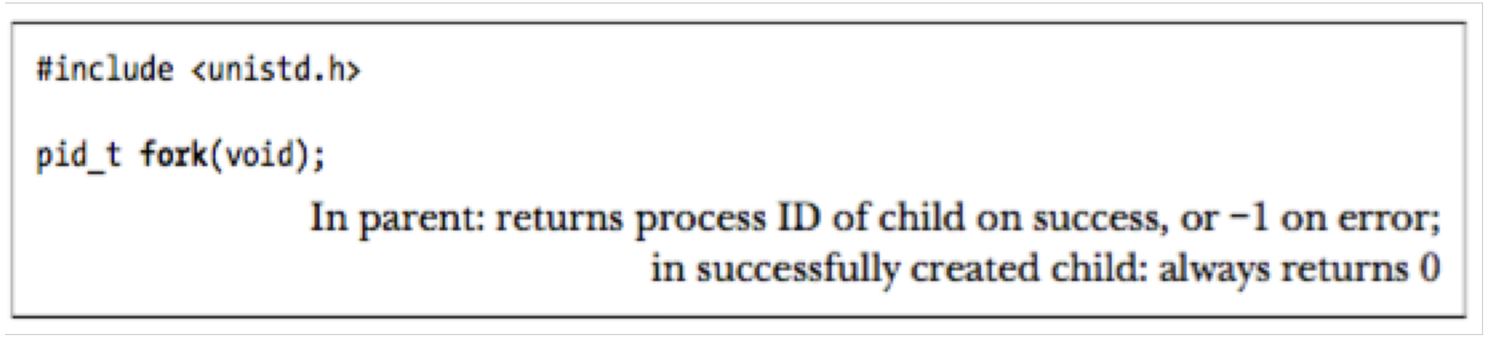


```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or –1 on error;
in successfully created child: always returns 0

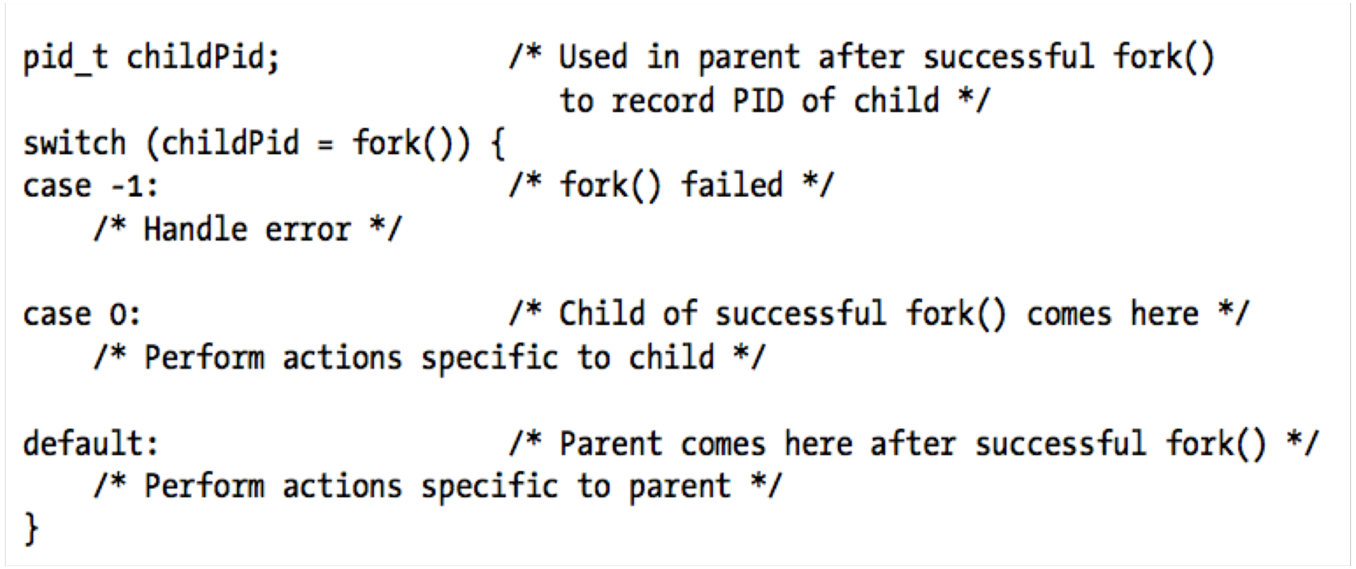
Kerrisk Chapter 24

fork



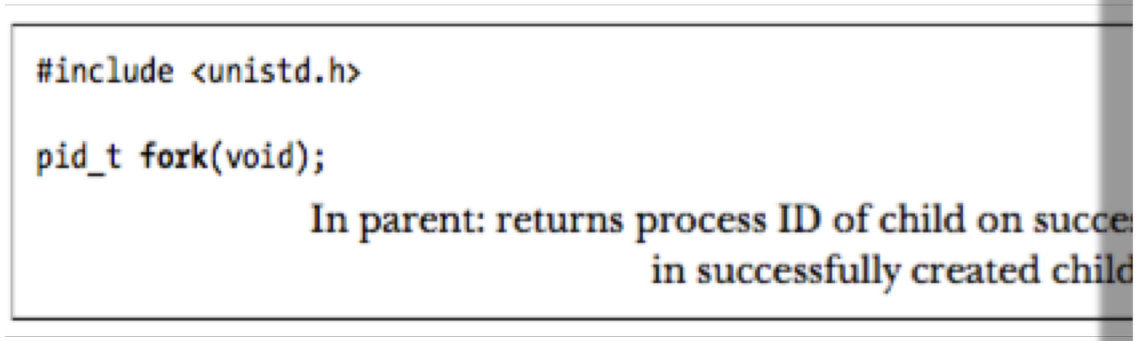
```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on success, or −1 on error;
in successfully created child: always returns 0



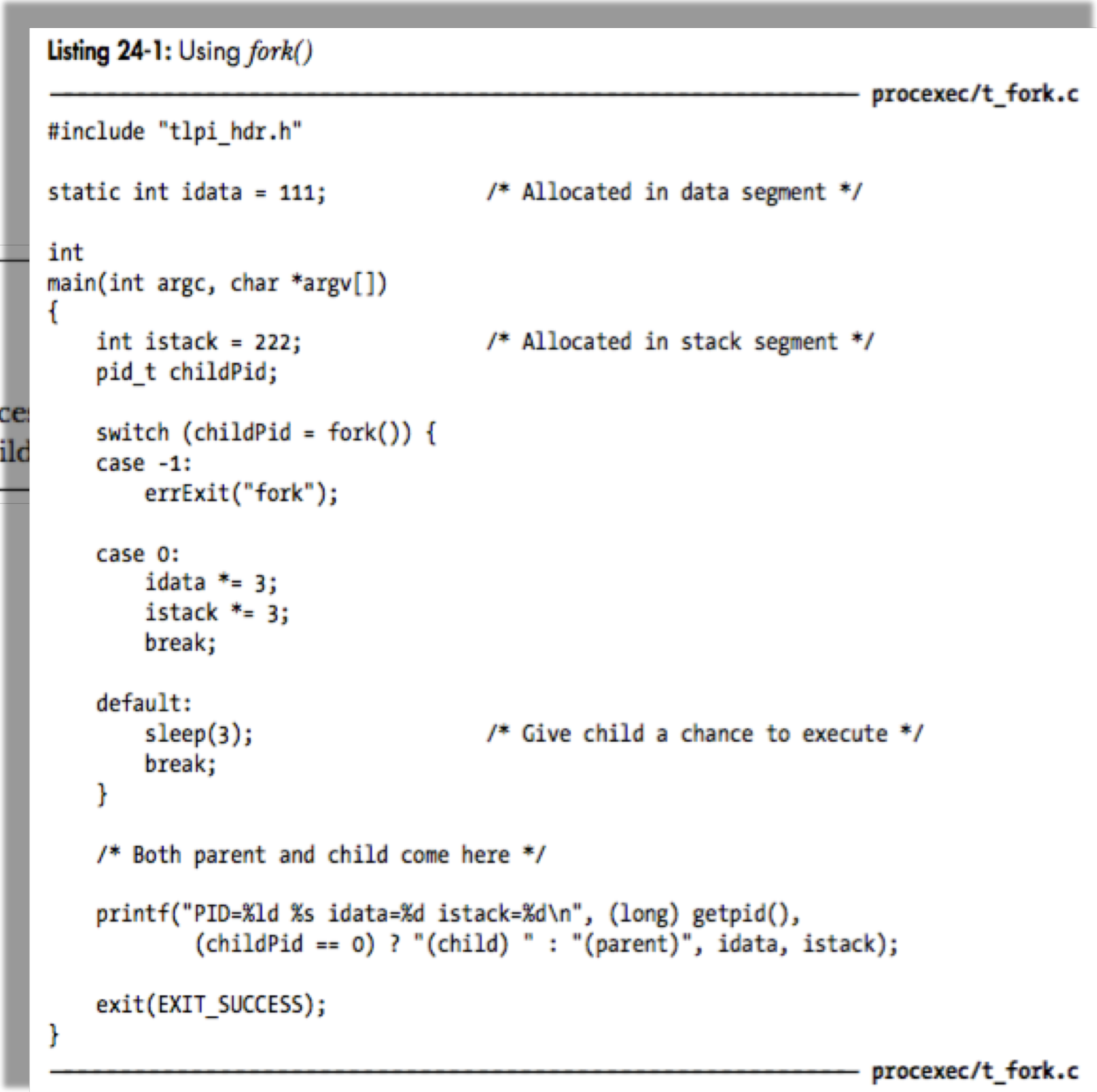
```
pid_t childPid;             /* Used in parent after successful fork()
                               to record PID of child */
switch (childPid = fork()) {
case -1:                    /* fork() failed */
    /* Handle error */

case 0:                     /* Child of successful fork() comes here */
    /* Perform actions specific to child */

default:                    /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```

fork



```
#include <unistd.h>

pid_t fork(void);
```

In parent: returns process ID of child on succe
in successfully created child



Listing 24-1: Using *fork()*

procexec/t_fork.c

```
#include "tlpi_hdr.h"

static int idata = 111;             /* Allocated in data segment */

int
main(int argc, char *argv[])
{
    int istack = 222;               /* Allocated in stack segment */
    pid_t childPid;

    switch (childPid = fork()) {
    case -1:
        errExit("fork");

    case 0:
        idata *= 3;
        istack *= 3;
        break;

    default:
        sleep(3);                   /* Give child a chance to execute */
        break;
    }

    /* Both parent and child come here */

    printf("PID=%ld %s idata=%d istack=%d\n", (long) getpid(),
            (childPid == 0) ? "(child) " : "(parent)", idata, istack);

    exit(EXIT_SUCCESS);
}
```

procexec/t_fork.c

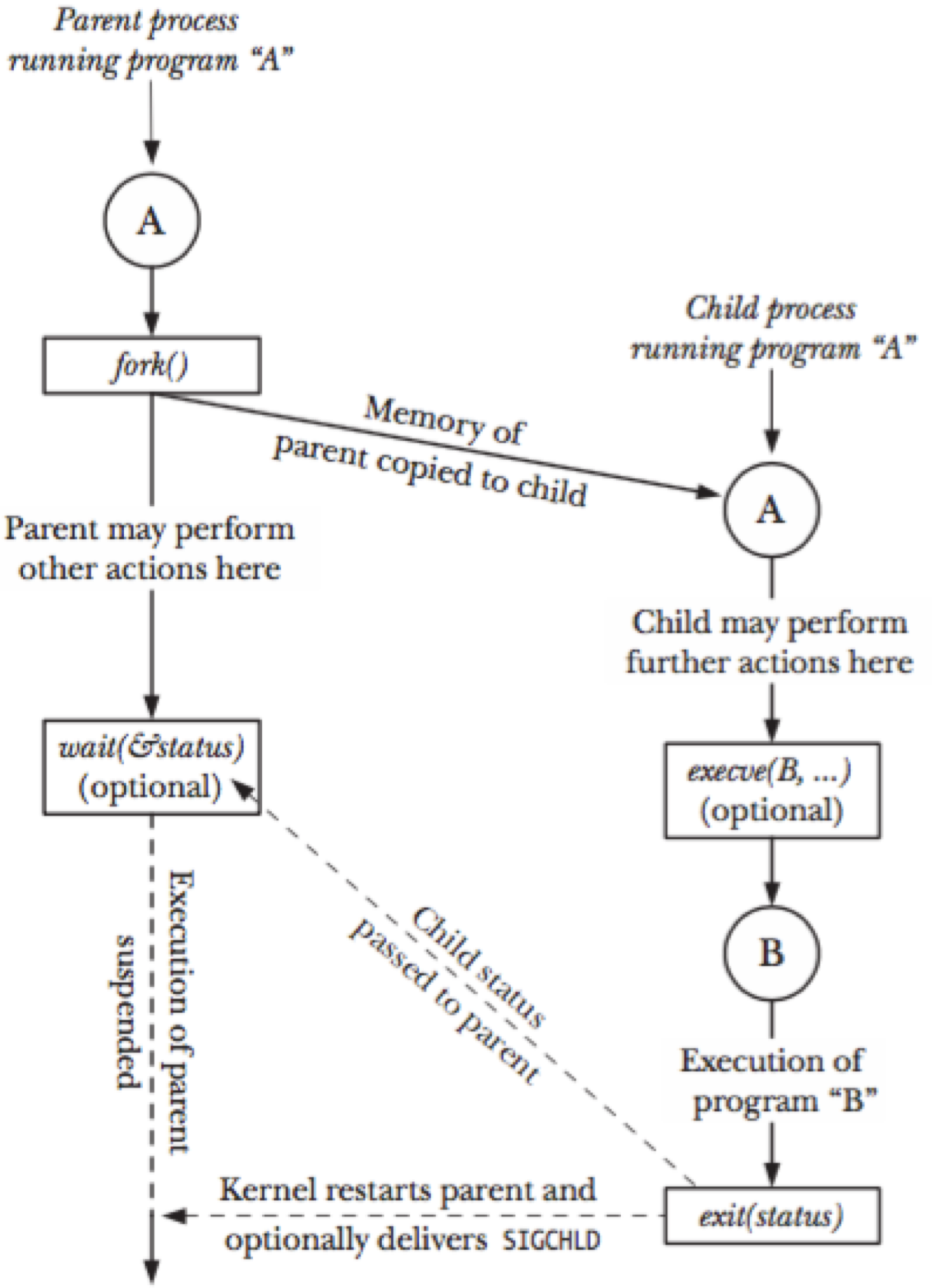

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent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

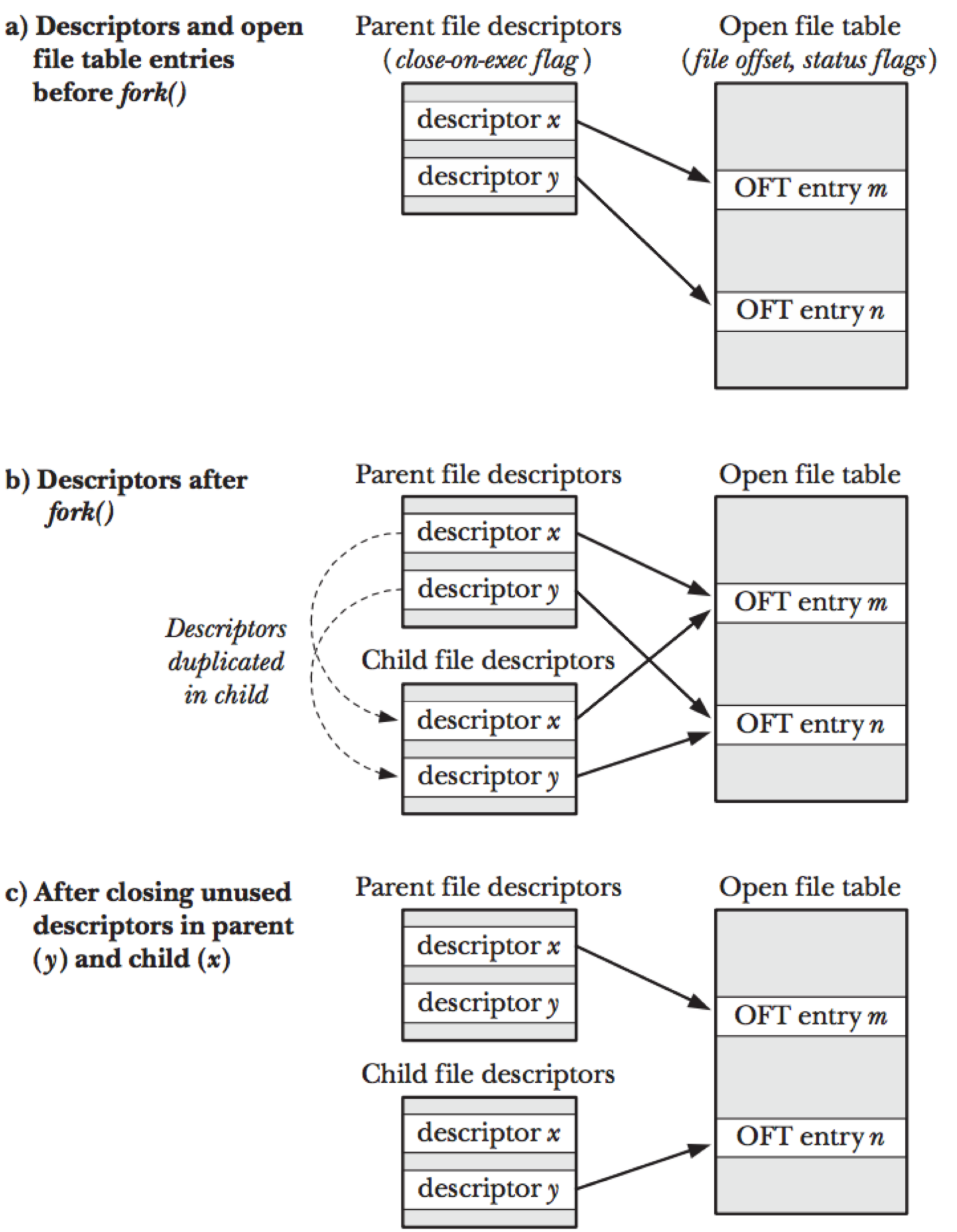


Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

Copy on write memory

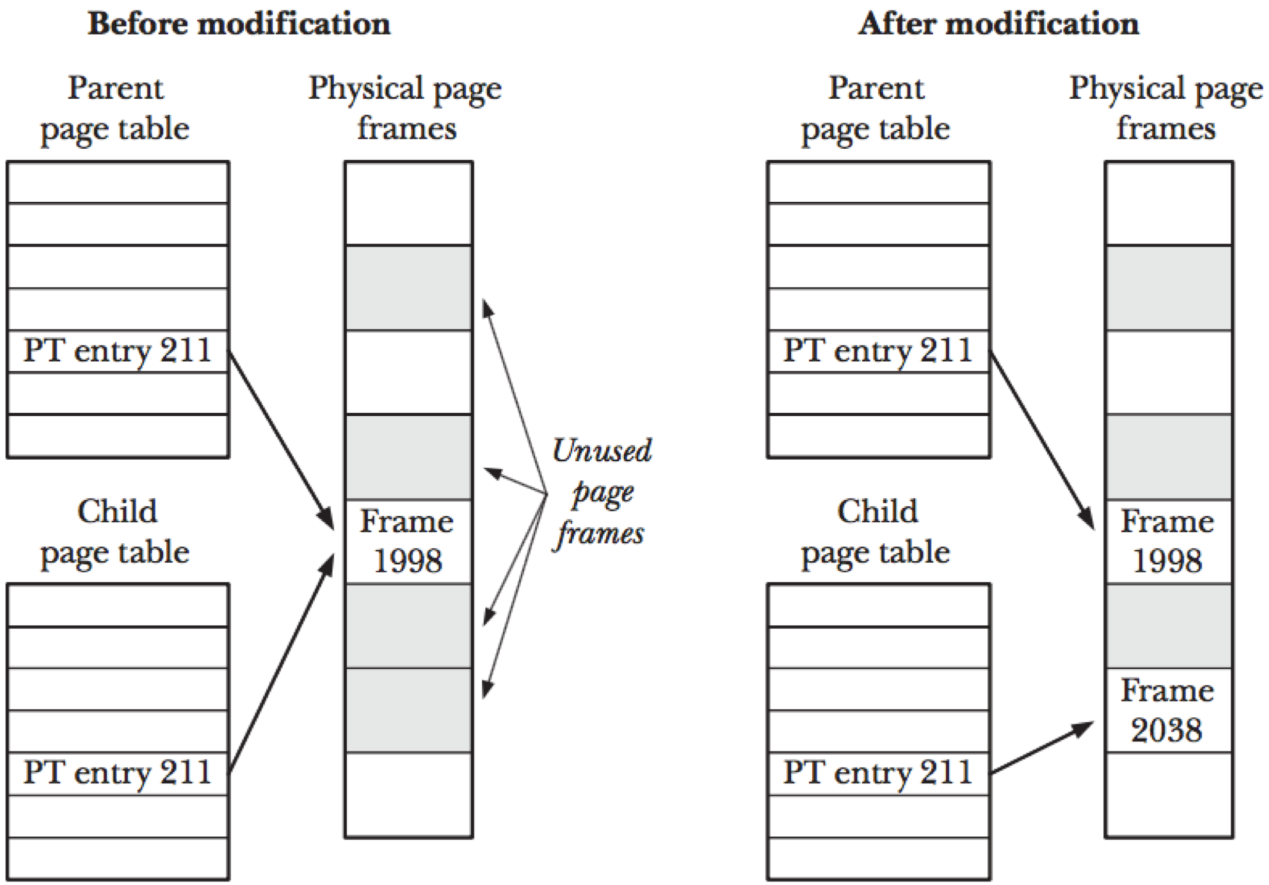
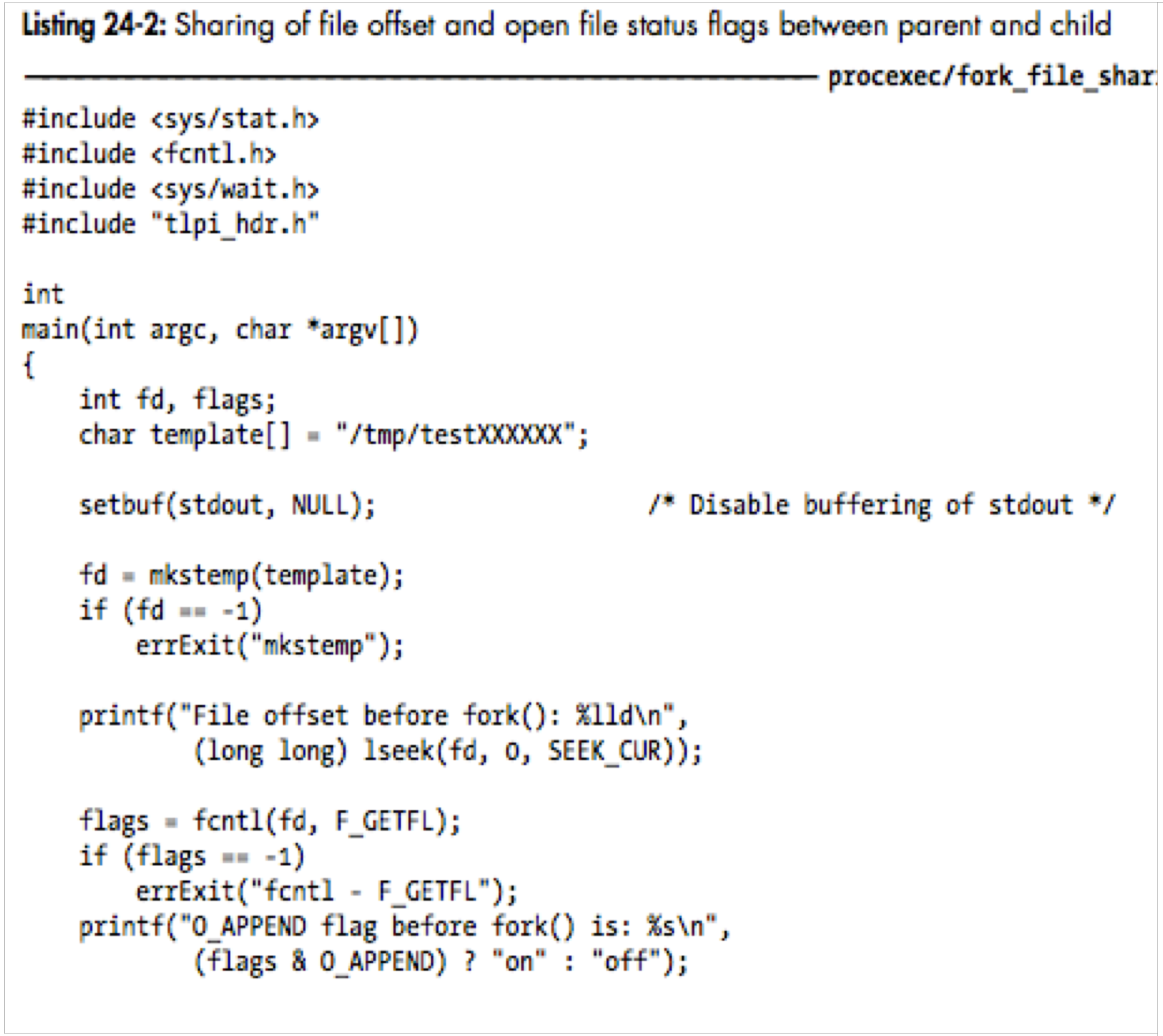


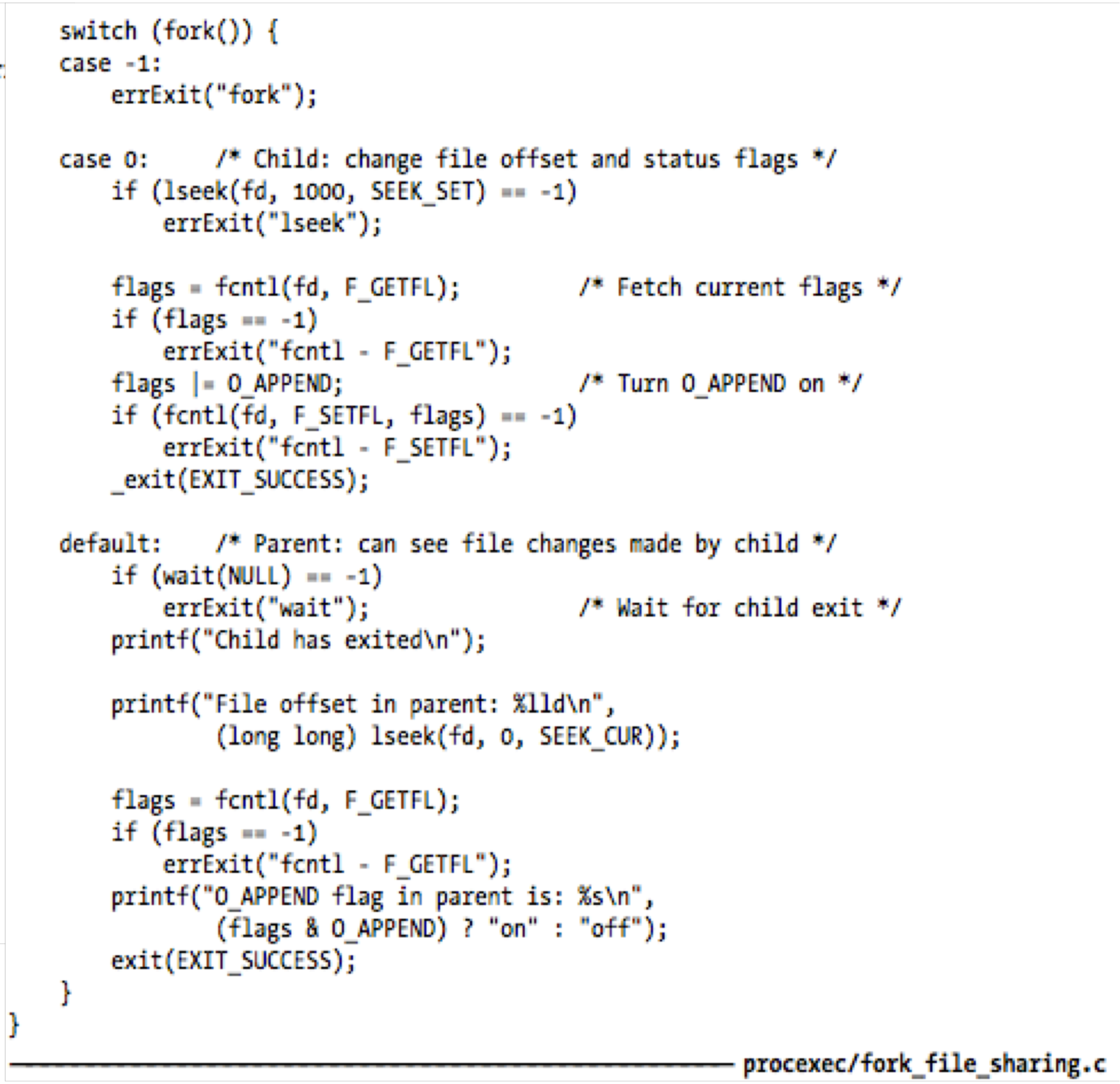
Figure 24-3: Page tables before and after modification of a shared copy-on-write page



Listing 24-2: Sharing of file offset and open file status flags between parent and child

procexec/fork_file_shar

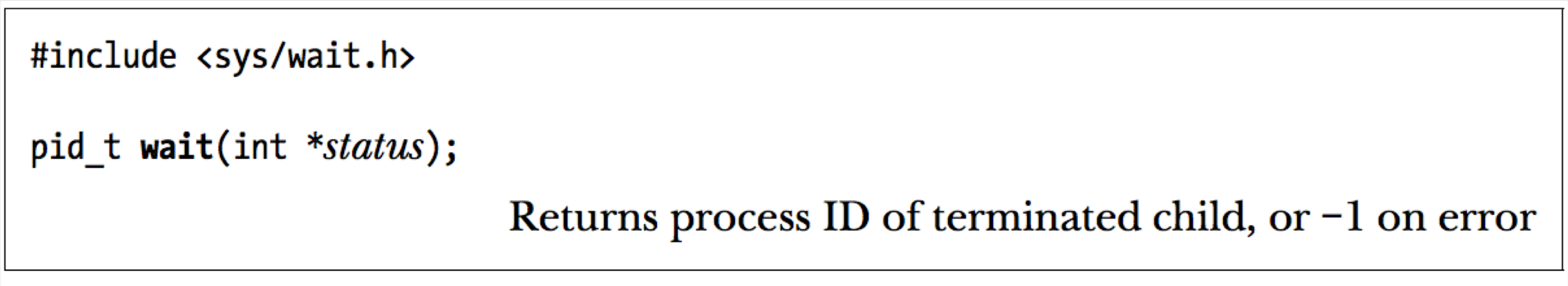
```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd, flags;
    char template[] = "/tmp/testXXXXXX";

    setbuf(stdout, NULL);                   /* Disable buffering of stdout */

    fd = mkstemp(template);
    if (fd == -1)
        errExit("mkstemp");

    printf("File offset before fork(): %lld\n",
            (long long) lseek(fd, 0, SEEK_CUR));

    flags = fcntl(fd, F_GETFL);
    if (flags == -1)
        errExit("fcntl - F_GETFL");
    printf("O_APPEND flag before fork() is: %s\n",
            (flags & O_APPEND) ? "on" : "off");
```



```
    switch (fork()) {
    case -1:
        errExit("fork");

    case 0:     /* Child: change file offset and status flags */
        if (lseek(fd, 1000, SEEK_SET) == -1)
            errExit("lseek");

        flags = fcntl(fd, F_GETFL);         /* Fetch current flags */
        if (flags == -1)
            errExit("fcntl - F_GETFL");
        flags |= O_APPEND;                  /* Turn O_APPEND on */
        if (fcntl(fd, F_SETFL, flags) == -1)
            errExit("fcntl - F_SETFL");
        _exit(EXIT_SUCCESS);

    default:    /* Parent: can see file changes made by child */
        if (wait(NULL) == -1)
            errExit("wait");                /* Wait for child exit */
        printf("Child has exited\n");

        printf("File offset in parent: %lld\n",
                (long long) lseek(fd, 0, SEEK_CUR));

        flags = fcntl(fd, F_GETFL);
        if (flags == -1)
            errExit("fcntl - F_GETFL");
        printf("O_APPEND flag in parent is: %s\n",
                (flags & O_APPEND) ? "on" : "off");
        exit(EXIT_SUCCESS);
    }
}
```

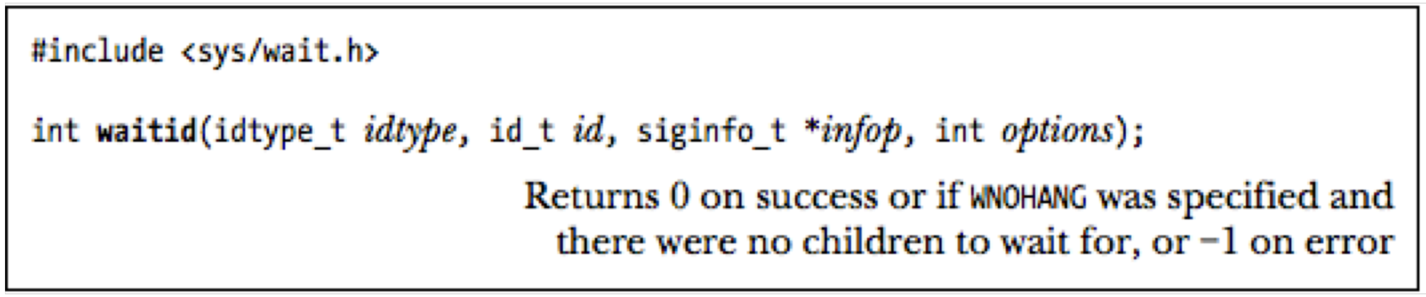
procexec/fork_file_sharing.c

wait()



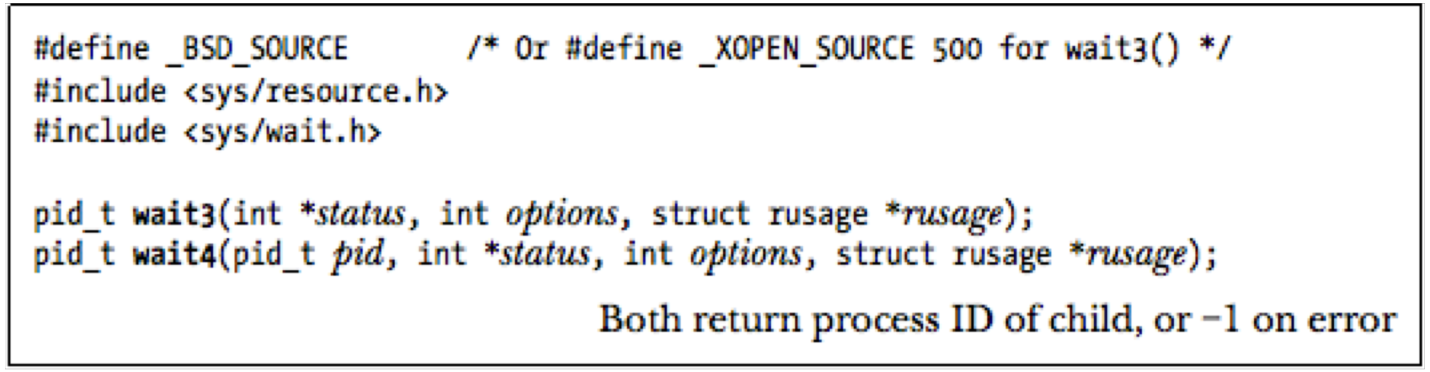
```
#include <sys/wait.h>

pid_t wait(int *status);
```

Returns process ID of terminated child, or −1 on error



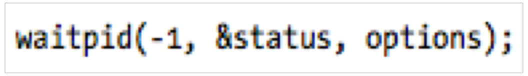
```
#include <sys/wait.h>

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Returns 0 on success or if WNOHANG was specified and there were no children to wait for, or −1 on error

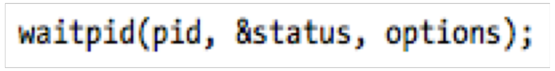


```
#define _BSD_SOURCE       /* Or #define _XOPEN_SOURCE 500 for wait3() */
#include <sys/resource.h>
#include <sys/wait.h>

pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

Both return process ID of child, or −1 on error



```
waitpid(-1, &status, options);
```



```
waitpid(pid, &status, options);
```

Wait *status*

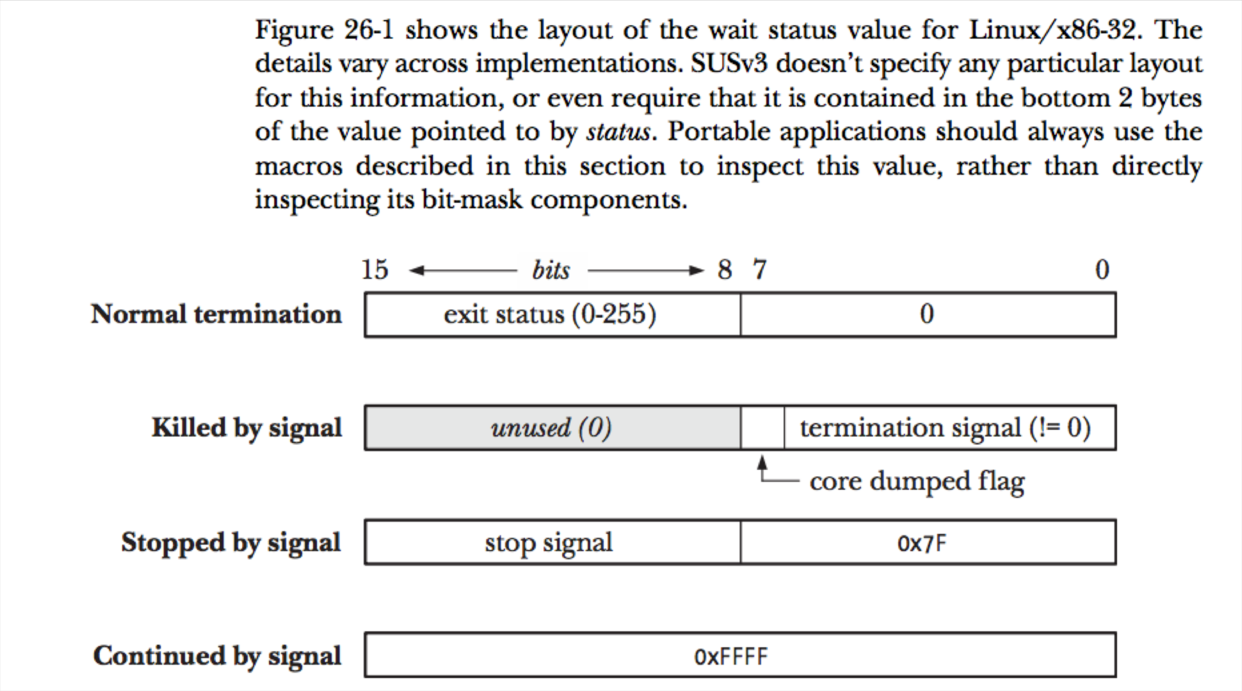


Figure 26-1 shows the layout of the wait status value for Linux/x86-32. The details vary across implementations. SUSv3 doesn't specify any particular layout for this information, or even require that it is contained in the bottom 2 bytes of the value pointed to by *status*. Portable applications should always use the macros described in this section to inspect this value, rather than directly inspecting its bit-mask components.

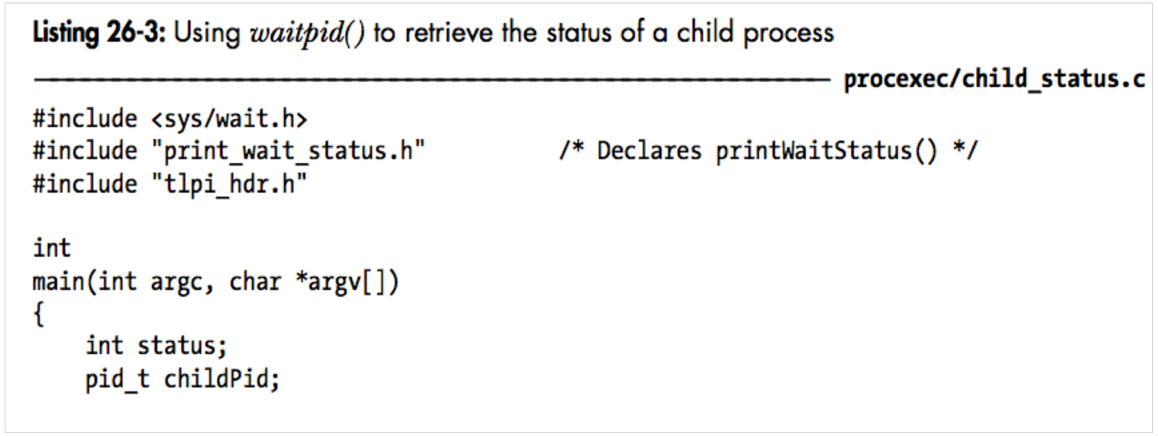
	15 ← *bits* → 8	7 0
Normal termination	exit status (0-255)	0
Killed by signal	*unused (0)*	core dumped flag; termination signal (!= 0)
Stopped by signal	stop signal	0x7F
Continued by signal	0xFFFF	

Listing 26-1: Creating and waiting for multiple children

procexec/multi_wait.c

```
#include <sys/wait.h>
#include <time.h>
#include "curr_time.h"                  /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numDead;       /* Number of children so far waited for */
    pid_t childPid;    /* PID of waited for child */
    int j;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);               /* Disable buffering of stdout */

    for (j = 1; j < argc; j++) {        /* Create one child for each argument */
        switch (fork()) {
        case -1:
            errExit("fork");

        case 0:                         /* Child sleeps for a while then exits */
            printf("[%s] child %d started with PID %ld, sleeping %s "
                    "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
            _exit(EXIT_SUCCESS);

        default:                        /* Parent just continues around loop */
            break;
        }
    }

    numDead = 0;
    for (;;) {                          /* Parent waits for each child to exit */
        childPid = wait(NULL);
        if (childPid == -1) {
            if (errno == ECHILD) {
                printf("No more children - bye!\n");
                exit(EXIT_SUCCESS);
            } else {                    /* Some other (unexpected) error */
                errExit("wait");
            }
        }

        numDead++;
        printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
                currTime("%T"), (long) childPid, numDead);
    }
}
```

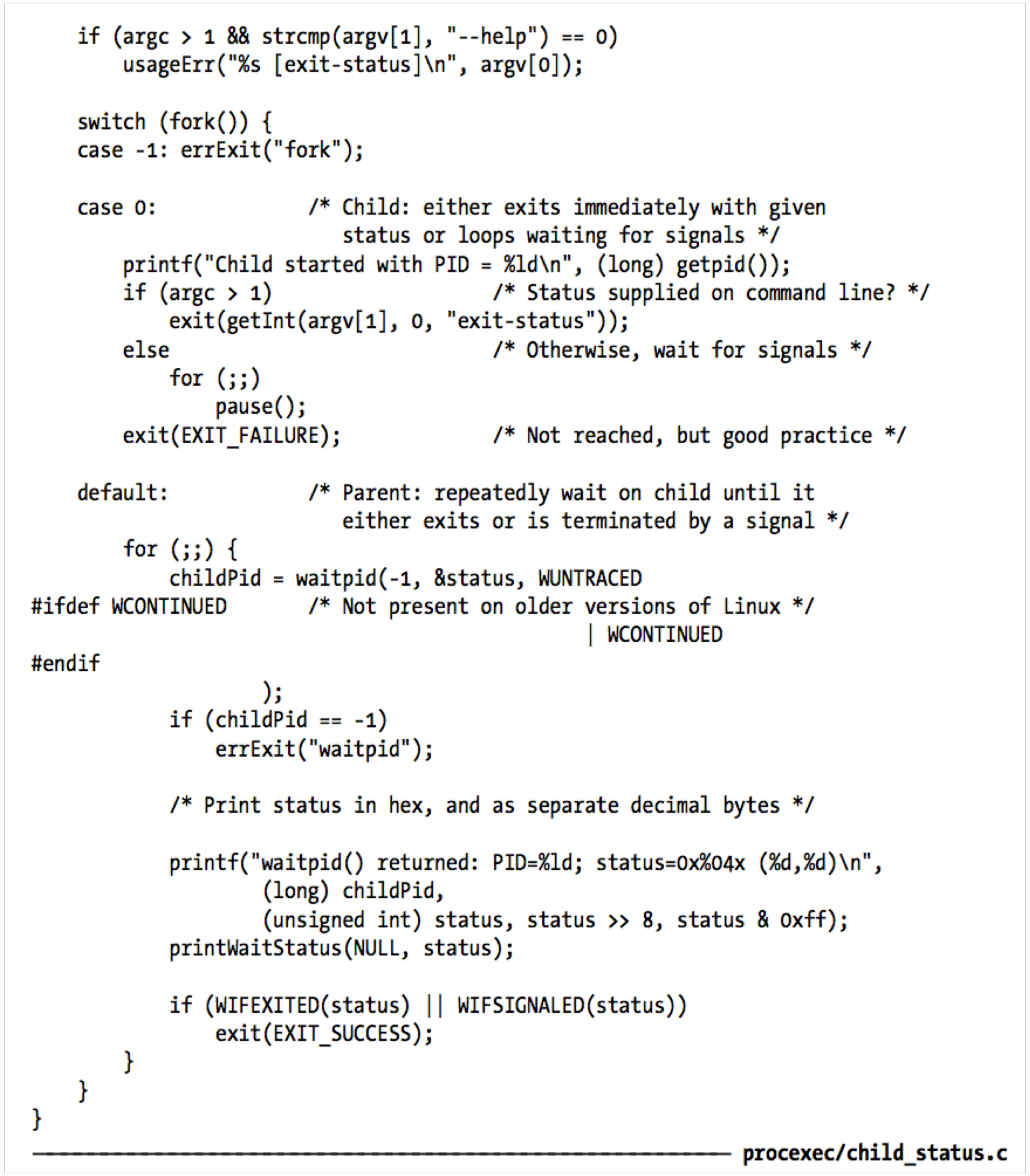
procexec/multi_wait.c



Listing 26-3: Using *waitpid()* to retrieve the status of a child process

––– **procexec/child_status.c**

```
#include <sys/wait.h>
#include "print_wait_status.h"          /* Declares printWaitStatus() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int status;
    pid_t childPid;
```

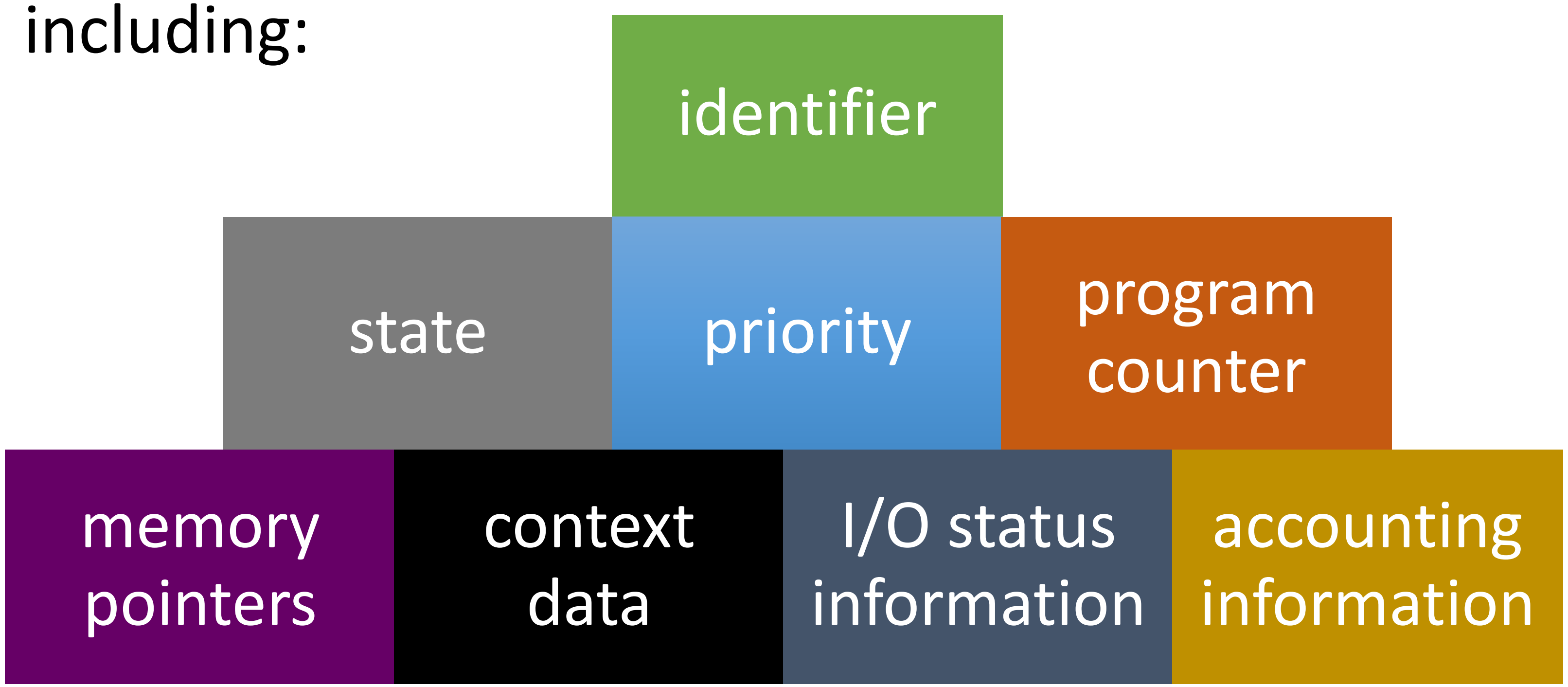


```
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [exit-status]\n", argv[0]);

    switch (fork()) {
    case -1: errExit("fork");

    case 0:             /* Child: either exits immediately with given
                           status or loops waiting for signals */
        printf("Child started with PID = %ld\n", (long) getpid());
        if (argc > 1)                   /* Status supplied on command line? */
            exit(getInt(argv[1], 0, "exit-status"));
        else                            /* Otherwise, wait for signals */
            for (;;)
                pause();
        exit(EXIT_FAILURE);             /* Not reached, but good practice */

    default:            /* Parent: repeatedly wait on child until it
                           either exits or is terminated by a signal */
        for (;;) {
            childPid = waitpid(-1, &status, WUNTRACED
#ifdef WCONTINUED       /* Not present on older versions of Linux */
                                                | WCONTINUED
#endif
                    );
            if (childPid == -1)
                errExit("waitpid");

            /* Print status in hex, and as separate decimal bytes */

            printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                    (long) childPid,
                    (unsigned int) status, status >> 8, status & 0xff);
            printWaitStatus(NULL, status);

            if (WIFEXITED(status) || WIFSIGNALED(status))
                exit(EXIT_SUCCESS);
        }
    }
}
```

––– **procexec/child_status.c**

Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:



identifier

state

priority

program counter

memory pointers

context data

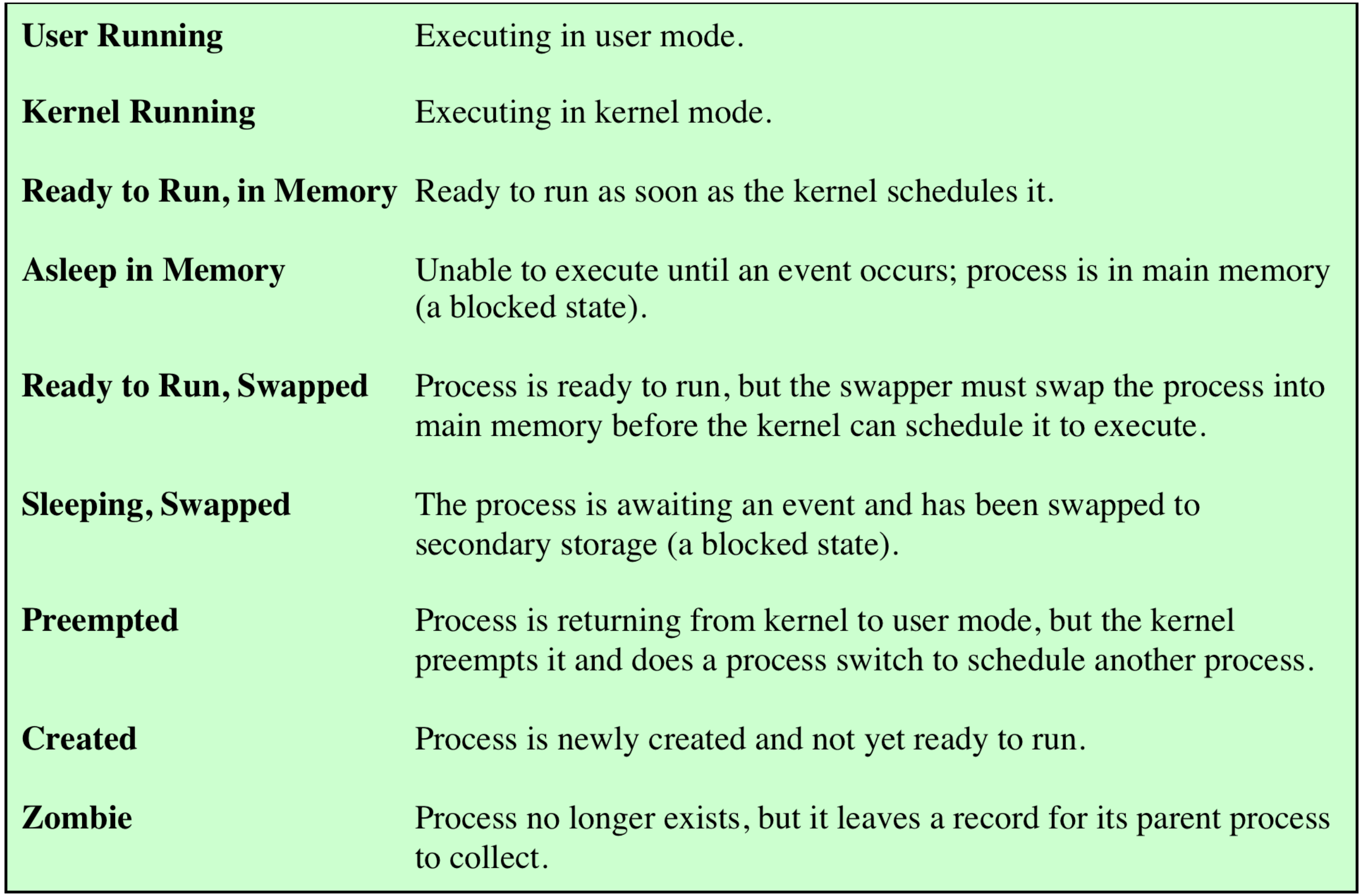
I/O status information

accounting information

Unix SVR4

- Uses the model where most of the OS executes within the environment of a user process
- System processes run in kernel mode
  - executes operating system code to perform administrative and housekeeping functions
- User Processes
  - operate in user mode to execute user programs and utilities
  - operate in kernel mode to execute instructions that belong to the kernel
  - enter kernel mode by issuing a system call, when an exception is generated, or when an interrupt occurs

Table 3.9 UNIX Process States



User Running	Executing in user mode.
Kernel Running	Executing in kernel mode.
Ready to Run, in Memory	Ready to run as soon as the kernel schedules it.
Asleep in Memory	Unable to execute until an event occurs; process is in main memory (a blocked state).
Ready to Run, Swapped	Process is ready to run, but the swapper must swap the process into main memory before the kernel can schedule it to execute.
Sleeping, Swapped	The process is awaiting an event and has been swapped to secondary storage (a blocked state).
Preempted	Process is returning from kernel to user mode, but the kernel preempts it and does a process switch to schedule another process.
Created	Process is newly created and not yet ready to run.
Zombie	Process no longer exists, but it leaves a record for its parent process to collect.

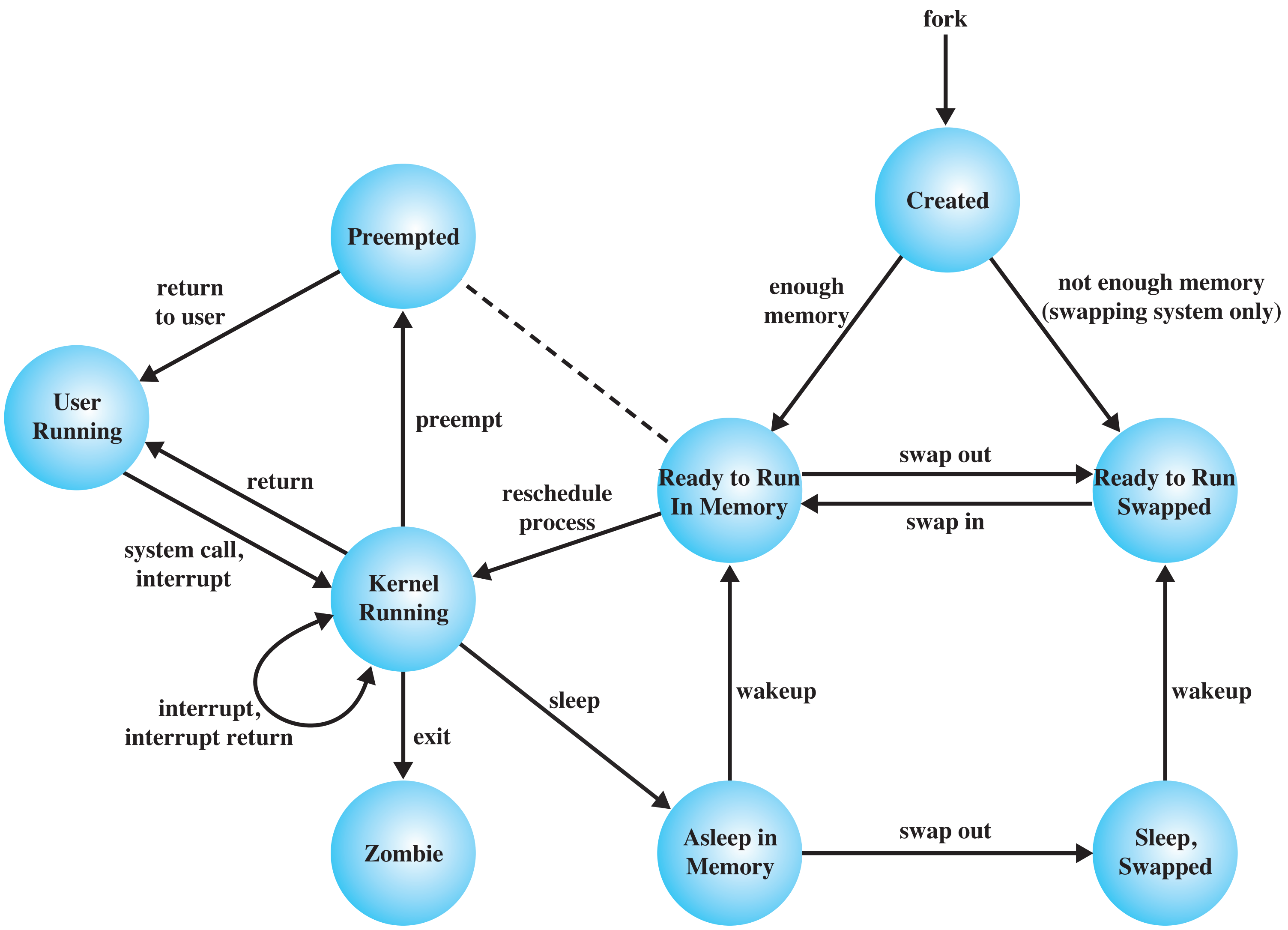
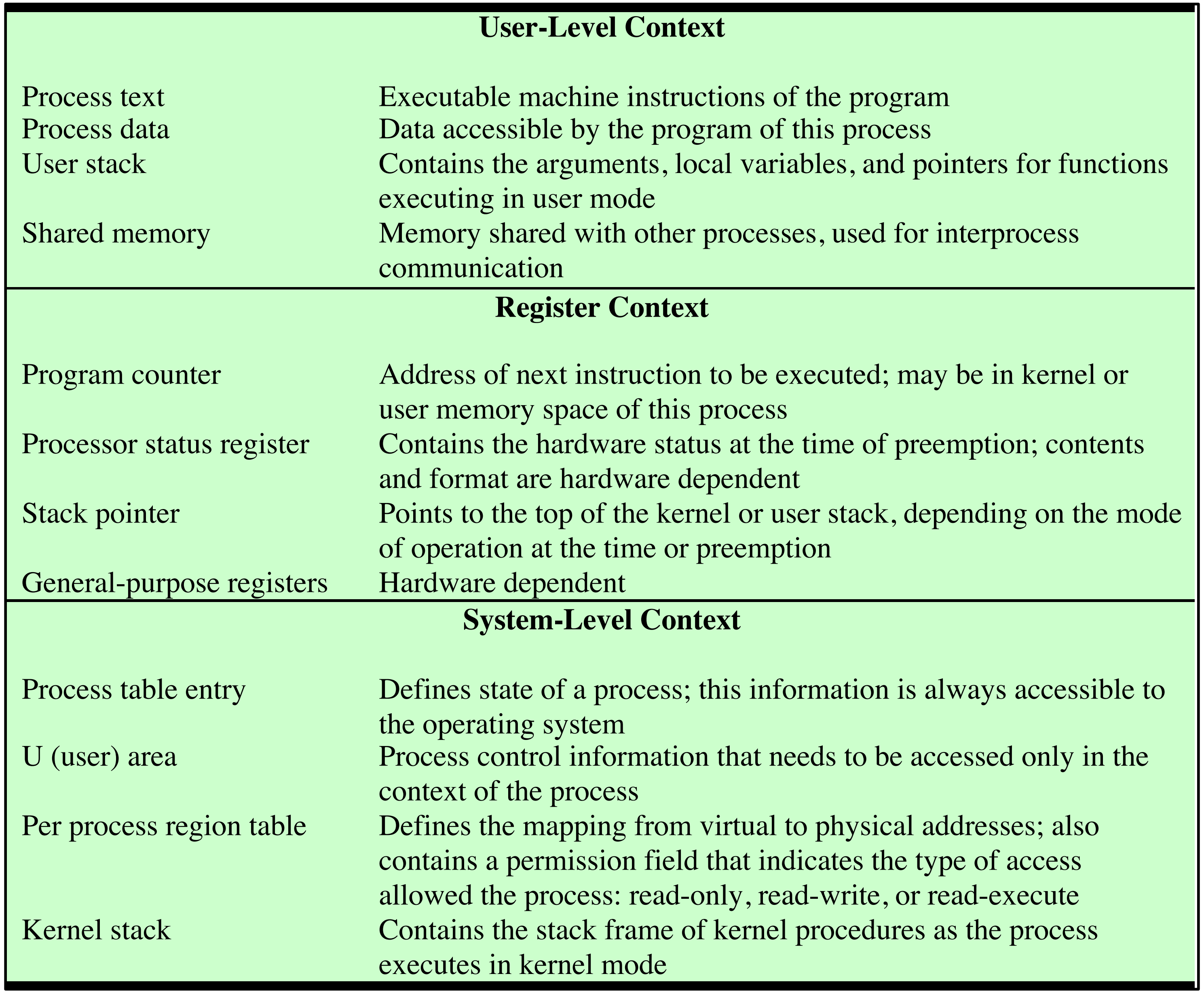


Figure 3.17 UNIX Process State Transition Diagram

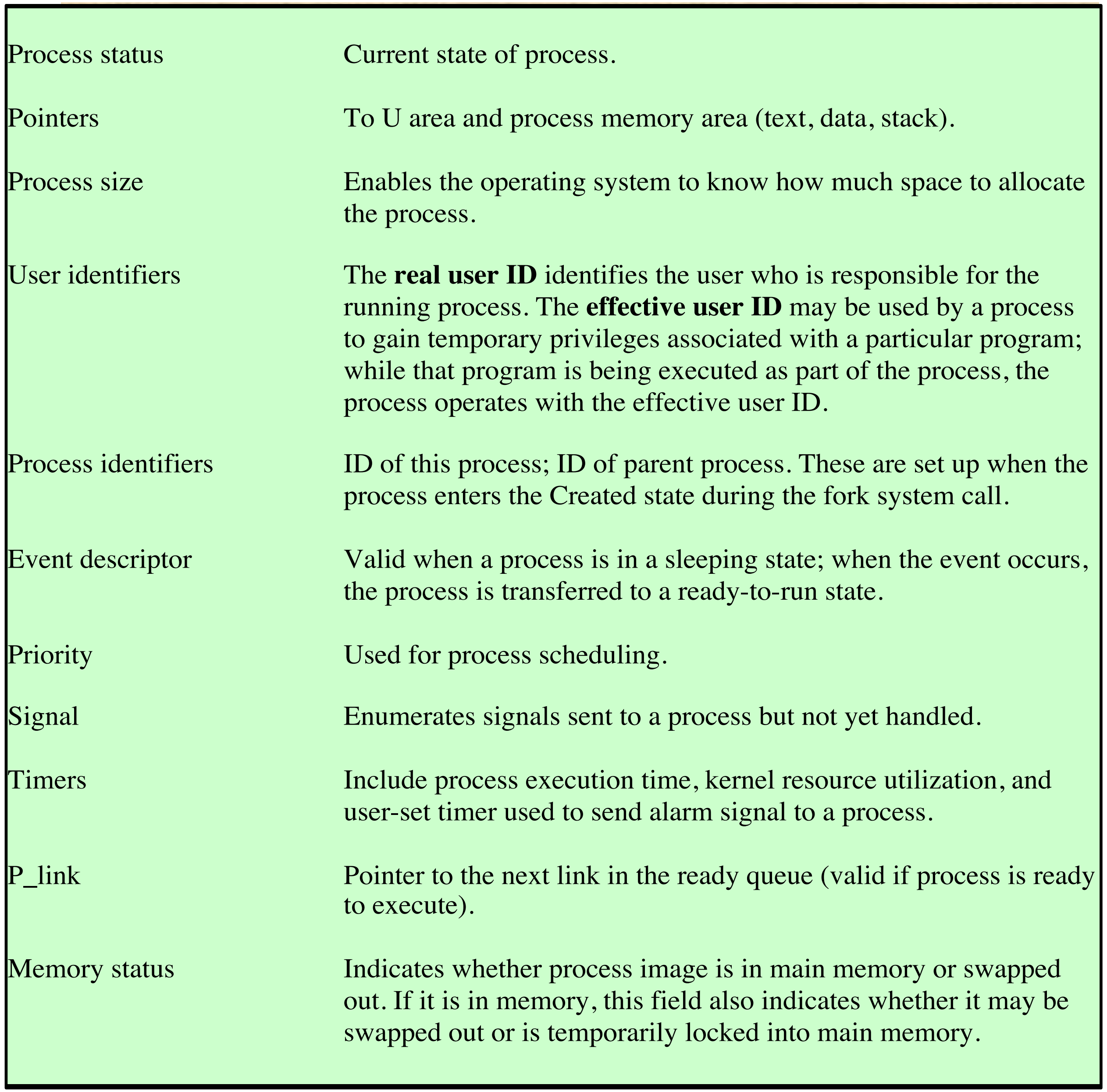
Table 3.10 UNIX Process Image



User-Level Context	
Process text	Executable machine instructions of the program
Process data	Data accessible by the program of this process
User stack	Contains the arguments, local variables, and pointers for functions executing in user mode
Shared memory	Memory shared with other processes, used for interprocess communication
Register Context	
Program counter	Address of next instruction to be executed; may be in kernel or user memory space of this process
Processor status register	Contains the hardware status at the time of preemption; contents and format are hardware dependent
Stack pointer	Points to the top of the kernel or user stack, depending on the mode of operation at the time or preemption
General-purpose registers	Hardware dependent
System-Level Context	
Process table entry	Defines state of a process; this information is always accessible to the operating system
U (user) area	Process control information that needs to be accessed only in the context of the process
Per process region table	Defines the mapping from virtual to physical addresses; also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute
Kernel stack	Contains the stack frame of kernel procedures as the process executes in kernel mode

(Table is located on page 144 in the textbook)

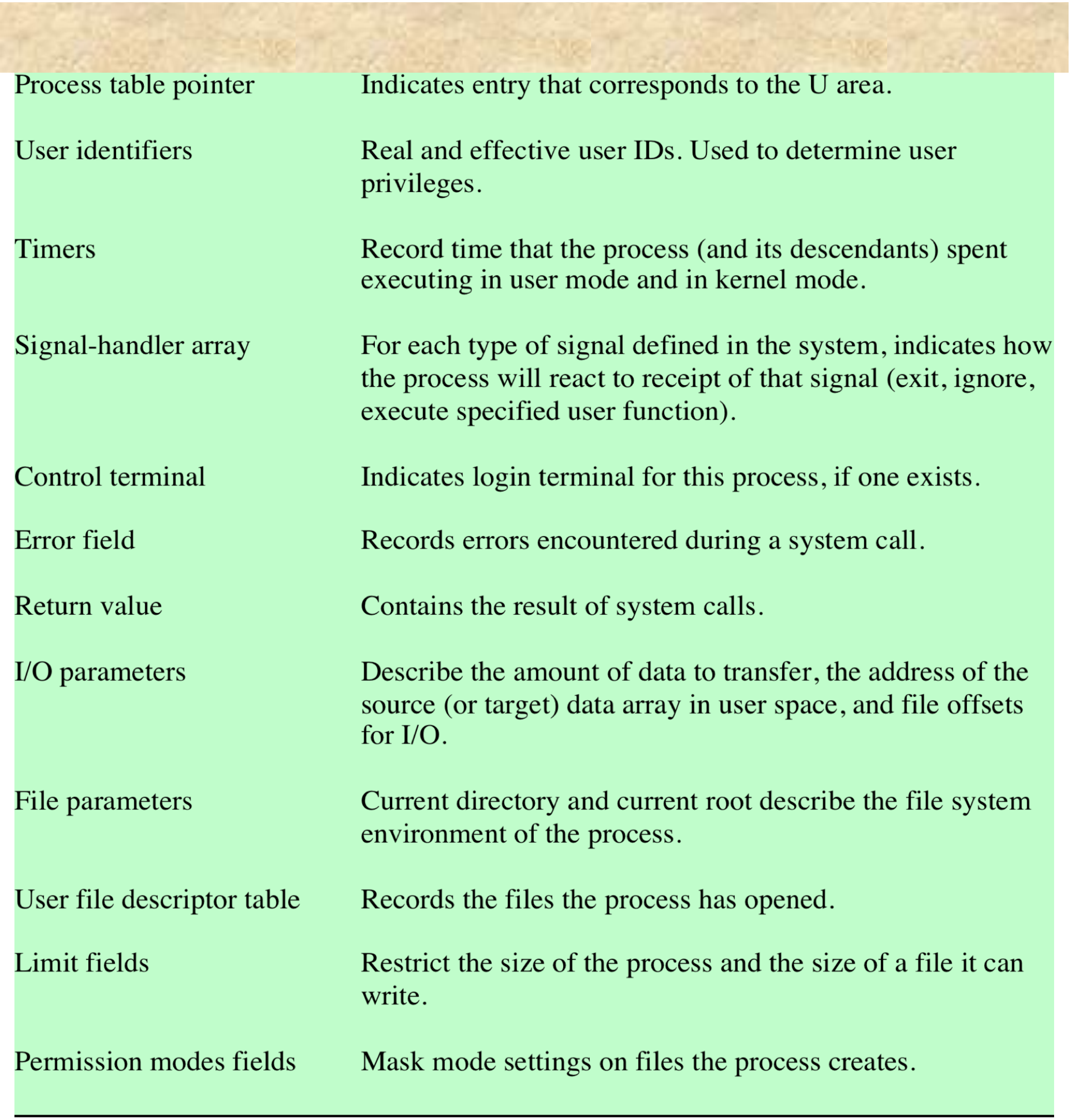
Table 3.11 UNIX Process Table Entry



Process status	Current state of process.
Pointers	To U area and process memory area (text, data, stack).
Process size	Enables the operating system to know how much space to allocate the process.
User identifiers	The **real user ID** identifies the user who is responsible for the running process. The **effective user ID** may be used by a process to gain temporary privileges associated with a particular program; while that program is being executed as part of the process, the process operates with the effective user ID.
Process identifiers	ID of this process; ID of parent process. These are set up when the process enters the Created state during the fork system call.
Event descriptor	Valid when a process is in a sleeping state; when the event occurs, the process is transferred to a ready-to-run state.
Priority	Used for process scheduling.
Signal	Enumerates signals sent to a process but not yet handled.
Timers	Include process execution time, kernel resource utilization, and user-set timer used to send alarm signal to a process.
P_link	Pointer to the next link in the ready queue (valid if process is ready to execute).
Memory status	Indicates whether process image is in main memory or swapped out. If it is in memory, this field also indicates whether it may be swapped out or is temporarily locked into main memory.

(Table is located on page 145 in the textbook)

Table 3.12
UNIX U Area

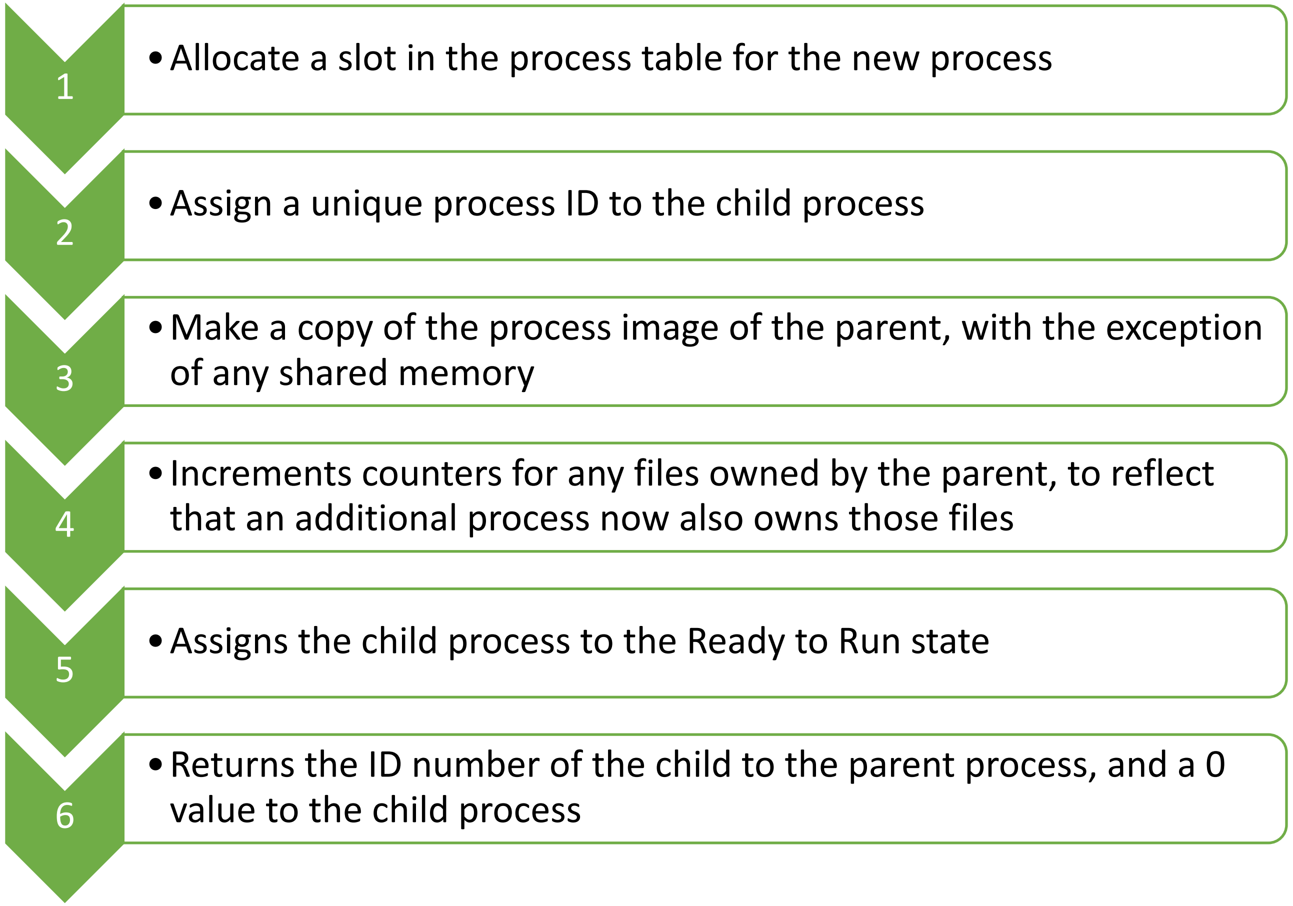


Process table pointer	Indicates entry that corresponds to the U area.
User identifiers	Real and effective user IDs. Used to determine user privileges.
Timers	Record time that the process (and its descendants) spent executing in user mode and in kernel mode.
Signal-handler array	For each type of signal defined in the system, indicates how the process will react to receipt of that signal (exit, ignore, execute specified user function).
Control terminal	Indicates login terminal for this process, if one exists.
Error field	Records errors encountered during a system call.
Return value	Contains the result of system calls.
I/O parameters	Describe the amount of data to transfer, the address of the source (or target) data array in user space, and file offsets for I/O.
File parameters	Current directory and current root describe the file system environment of the process.
User file descriptor table	Records the files the process has opened.
Limit fields	Restrict the size of the process and the size of a file it can write.
Permission modes fields	Mask mode settings on files the process creates.

(Table is located on page 146 in the textbook)

Process Creation

- Process creation is by means of the kernel system call, fork( )
- This causes the OS, in Kernel Mode, to:



1
- Allocate a slot in the process table for the new process

2
- Assign a unique process ID to the child process

3
- Make a copy of the process image of the parent, with the exception of any shared memory

4
- Increments counters for any files owned by the parent, to reflect that an additional process now also owns those files

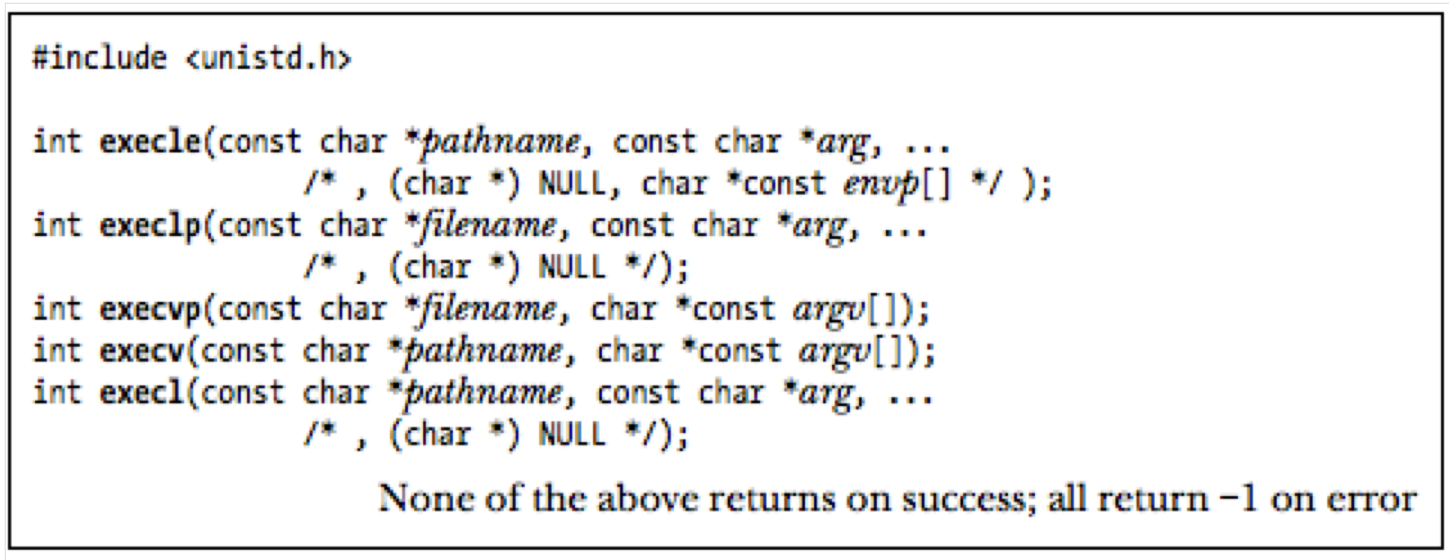
5
- Assigns the child process to the Ready to Run state

6
- Returns the ID number of the child to the parent process, and a 0 value to the child process

After Creation

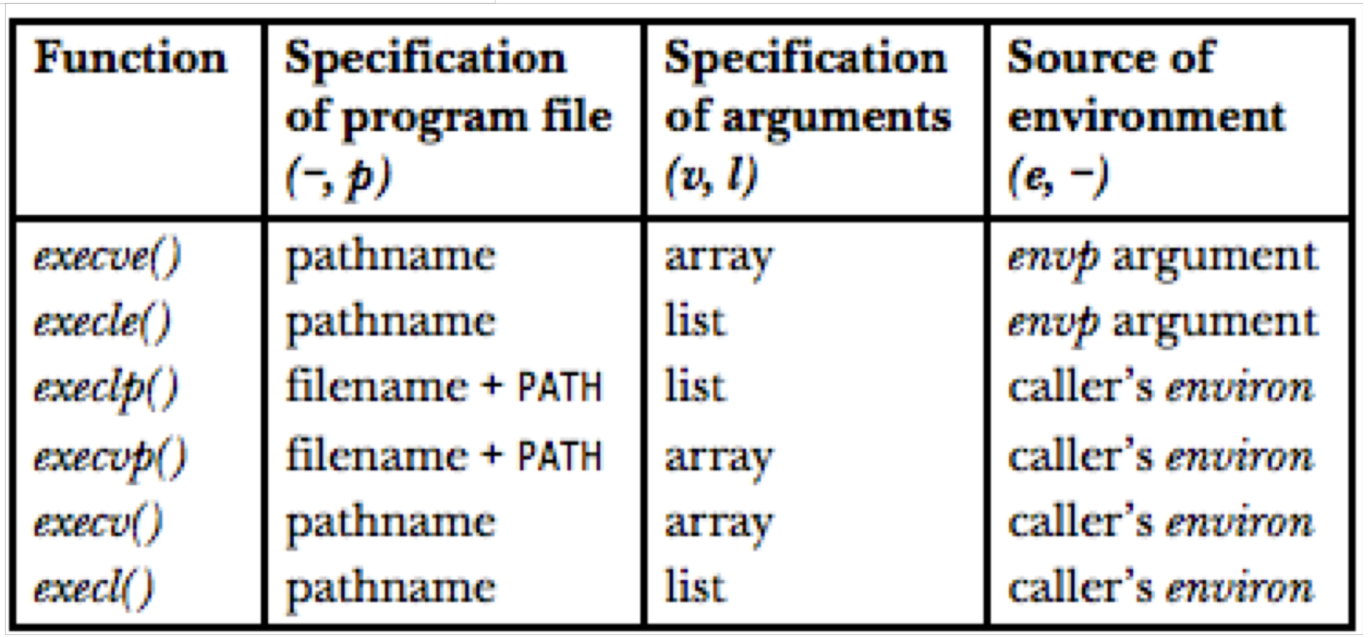
- After creating the process the Kernel can do one of the following, as part of the dispatcher routine:
  - stay in the parent process
  - transfer control to the child process
  - transfer control to another process

Exec()



```
#include <unistd.h>

int execle(const char *pathname, const char *arg, ...
                /* , (char *) NULL, char *const envp[] */ );
int execlp(const char *filename, const char *arg, ...
                /* , (char *) NULL */);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, ...
                /* , (char *) NULL */);
```

None of the above returns on success; all return –1 on error



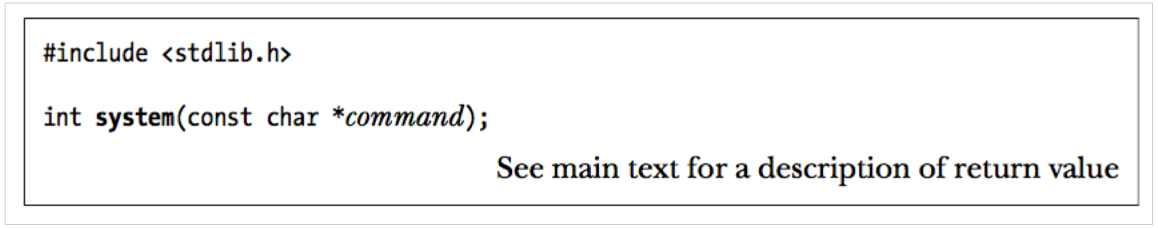
Function	**Specification of program file (–, *p*)**	**Specification of arguments (*v*, *l*)**	**Source of environment (*e*, –)**
execve()	pathname	array	*envp* argument
execle()	pathname	list	*envp* argument
execlp()	filename + PATH	list	caller's *environ*
execvp()	filename + PATH	array	caller's *environ*
execv()	pathname	array	caller's *environ*
execl()	pathname	list	caller's *environ*

Examples…

Linux find

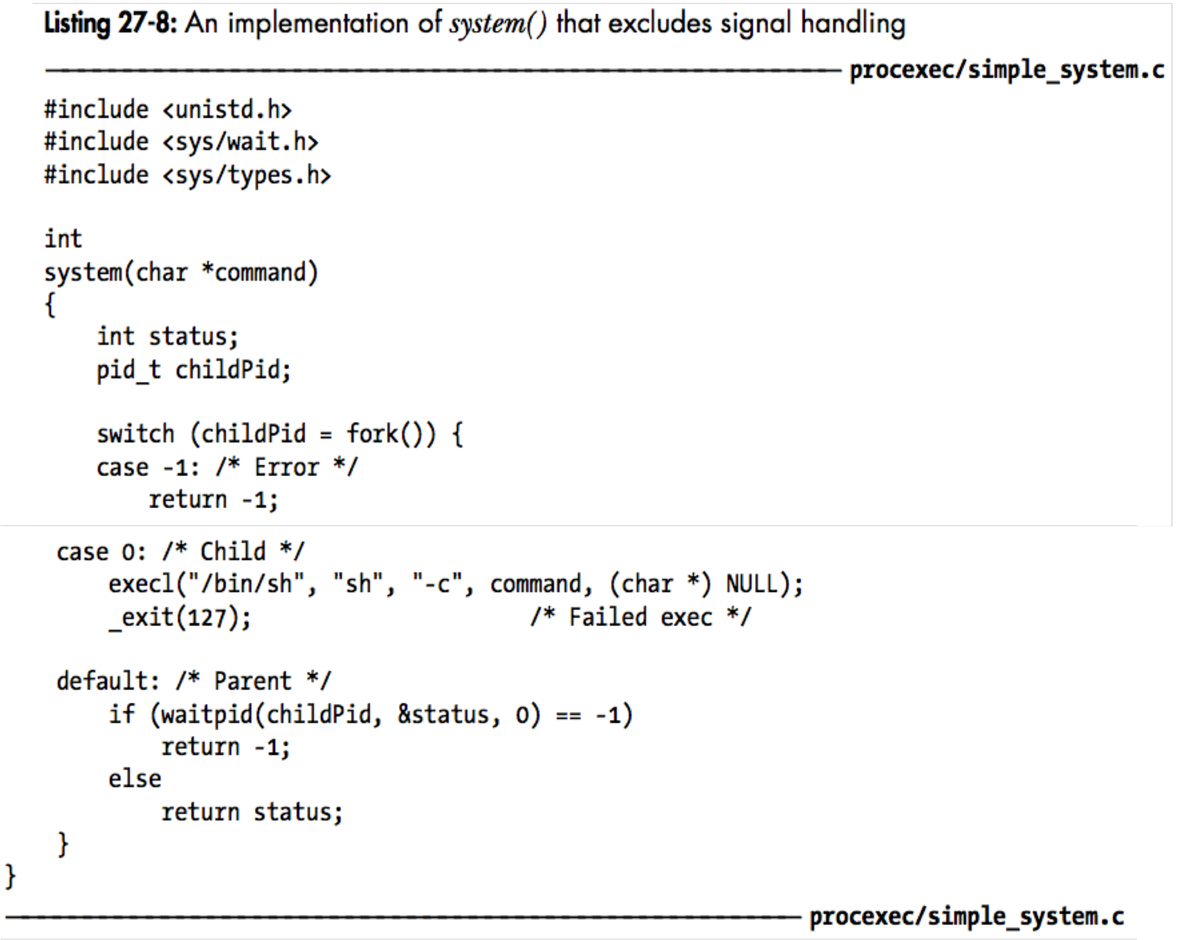
Man find

system



```
#include <stdlib.h>

int system(const char *command);
```

See main text for a description of return value



Listing 27-8: An implementation of *system()* that excludes signal handling

procexec/simple_system.c

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

int
system(char *command)
{
    int status;
    pid_t childPid;

    switch (childPid = fork()) {
    case -1: /* Error */
        return -1;

    case 0: /* Child */
        execl("/bin/sh", "sh", "-c", command, (char *) NULL);
        _exit(127);                     /* Failed exec */

    default: /* Parent */
        if (waitpid(childPid, &status, 0) == -1)
            return -1;
        else
            return status;
    }
}
```

procexec/simple_system.c

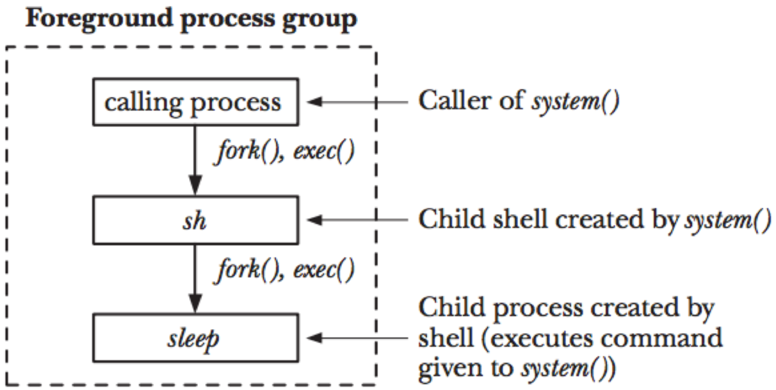

Foreground process group
calling process
Caller of system()
fork(), exec()
sh
Child shell created by system()
fork(), exec()
sleep
Child process created by shell (executes command given to system())

Figure 27-2: Arrangement of processes during execution of *system("sleep 20")*