Exam Technical Details

- When: Thursday, October 11th, in class.
- Seats will be assigned
- One page of notes are allowed.
    - 8.5x11 paper (double-sided is fine). Typed or handwritten. No Magnification instruments.
- No electronic devices.
    - Including calculators, watches, iwatches, phones, laptops, Tamagotchis, tablets, ...
- Multiple Choice
- Can grade as you exit the exam.
- Contact drc.ou.edu for appropriate accommodations (drc@ou.edu).

Operating System

- A program that controls the execution of application programs
- An interface between applications and hardware

Main objectives of an OS:

- Convenience
- Efficiency
- Ability to evolve

The Operating System as Resource Manager

- The OS is responsible for controlling the use of a computer's resources, such as I/O, main and secondary memory, and processor execution time

Modes of Operation

User Mode

- User program executes in user mode
- Certain areas of memory are protected from user access
- Certain instructions may not be executed

Kernel Mode

- Monitor executes in kernel mode
- Privileged instructions may be executed
- Protected areas of memory may be accessed

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

Percent CPU Utilization $= \frac{1}{31} = 0.032 = 3.2\%$

Figure 2.4 System Utilization Example

Process

- Fundamental to the structure of operating systems

A *process* can be defined as:

- A program in execution
- An instance of a running program
- The entity that can be assigned to, and executed on, a processor
- A unit of activity characterized by a single sequential thread of execution, a current state, and an associated set of system resources

Process Management

- The entire state of the process at any instant is contained in its context
- New features can be designed and incorporated into the OS by expanding the context to include any new information needed to support the feature

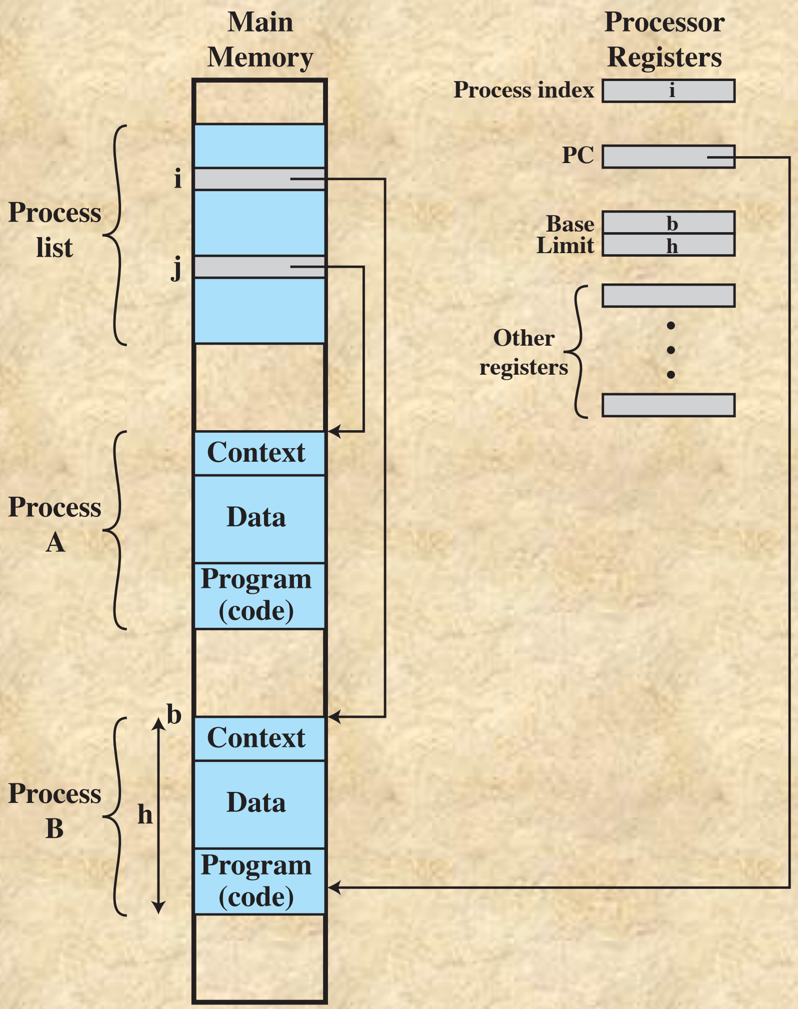


Figure 2.8 Typical Process Implementation

Memory Management

- The OS has five principal storage management responsibilities:

- Process isolation
- Automatic allocation and management
- Support of modular programming
- Protection and access control
- Long-term storage

Linux Signals

SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

Table 2.6 Some Linux Signals

Filesystem related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor
Process related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
ptrace	Provides a means by which a parent process my observe and control the execution of another process, and examine and change its core image and registers.
Scheduling related	
sched_getparam	Sets the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max	Returns the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler	Sets both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval	Writes into the timespec structure pointed to by the parameter `tp` the round robin time quantum for the process `pid`.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Table 2.7 Some Linux System Calls (page 1 of 2)

Interprocess Communication (IPC) related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl	Performs the control operation specified by cmd on the semaphore set semid.
semop	Performs operations on selected members of the semaphore set semid.
shmat	Attaches the shared memory segment identified by shmid to the data segment of the calling process.
shmctl	Allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment.
Socket (networking) related	
bind	Assigns the local IP address and port for a socket. Returns 0 for success and –1 for error.
connect	Establishes a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Returns local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Sets the options on a socket
Miscellaneous	
fsync	Copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
time	Returns the time in seconds since January 1, 1970.
vhangup	Simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.

Table 2.7 Some Linux System Calls (page 2 of 2)

Atomicity

- There are many situations where access to a common resource involves a sequence of operations that cannot be interrupted.
  - We want to treat these operations **atomically** (i.e., that they cannot be broken apart)
- These are called **critical sections**

- Because this issue comes up in many different ways in an OS, we will find a range of context-specific solutions to this problem

File Descriptors vs File Pointers

- File descriptor:
  - int type that references a table of open streams
  - Can reference files, pipes or sockets (more on the middle soon; latter is for inter-process communication)
  - Access through system calls: open(), read(), write(), close() …
- File pointer
  - FILE type defined in stdio.h (it is a struct)
  - Includes the file descriptor, but adds buffering and other features
  - Access through the stdio library: fopen(), fread(), fwrite(), fclose(), fprintf(), fscanf()
  - When working with files, this is the preferred interface

Flushing Streams

- Because FILE streams are buffered, a `fprintf()` does not necessarily affect the file immediately
- Instead, the bytes are dropped into a buffer; at some point the library will decide to move the bytes from the buffer to the file
- `fflush(fp)` will immediately force all bytes in the buffer to the file

File Descriptors to Files (or Streams): Three Levels of Representation

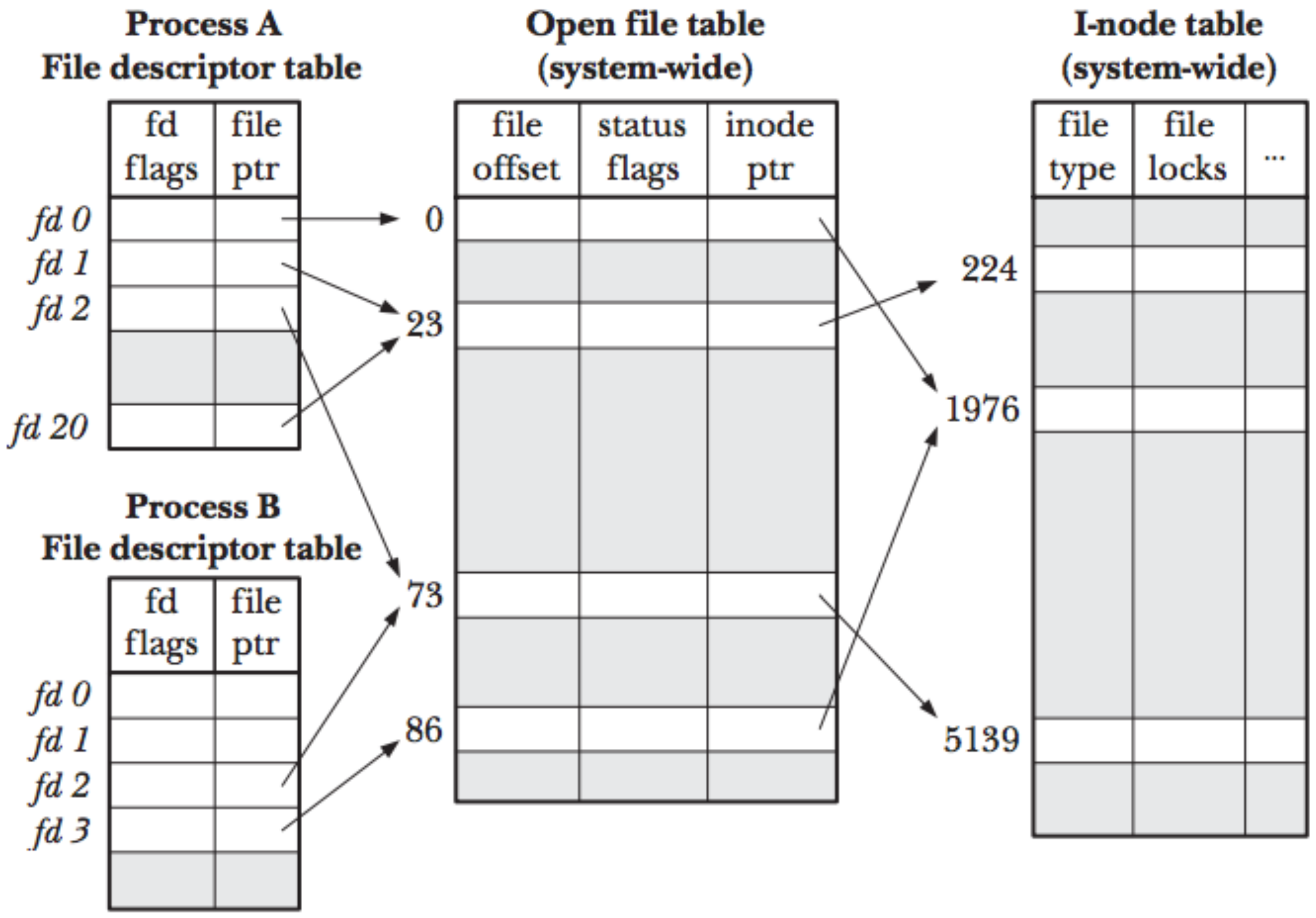


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

Copying a File Descriptor

- In some cases, it is useful for a process to be able to refer to the same file/stream using two different file descriptors
  - For example, if we want output written to both stdout and stderr to appear on stderr
- Allocate the first available fd & configure it to point to the same resource as oldfd:

```
newfd = dup(oldfd)
```

- Close newfd (if it is open) and allocate it to point to oldfd:

```
newfd = dup2(oldfd, newfd)
```

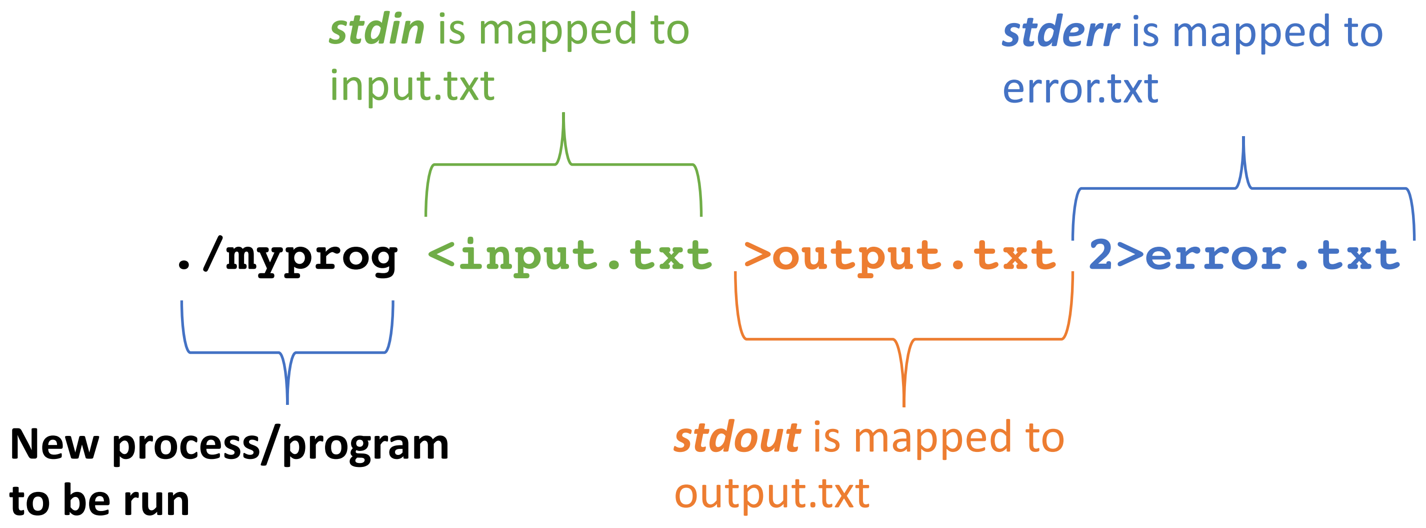
Key I/O System Calls

fd = open(pathname, flags, mode)	| opens the file identified by *pathname*, returning a file descriptor.
numread = read(fd, buffer, count)	| reads at most *count* bytes from the open file referred to by *fd* and stores them in *buffer*.
numwritten = write(fd, buffer, count)	| writes up to *count* bytes from *buffer* to the open file referred to by *fd*.
status = close(fd)	| is called after all I/O has been completed, in order to release the file descriptor *fd* and its associated kernel resources.

Standard File Descriptors

- When a shell program is run, these descriptors are copied from the terminal to the running program.
- I/O redirection may modify this assignment.
- IDEs may map output to stderr to a red color
- POSIX names are available in `<unistd.h>`

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*



```
./myprog <input.txt >output.txt 2>error.txt
```

stdin is mapped to input.txt
stderr is mapped to error.txt
New process/program to be run
stdout is mapped to output.txt

Universality of I/O

same four system calls—*open()*, *read()*, *write()*, and *close()*—are used to perform I/O on **all** types of files.

```
$ ./copy test test.old            Copy a regular file
$ ./copy a.txt /dev/tty           Copy a regular file to this terminal
$ ./copy /dev/tty b.txt           Copy input from this terminal to a regular file
$ ./copy /dev/pts/16 /dev/tty     Copy input from another terminal
```

File offset

| Also called *read- write offset* or *pointer*

| the kernel records a *file offset* for **each open file**.

| The first byte of the file is at offset 0.

| The file offset is set to point to the start of the file when the file is opened and is automatically adjusted by each subsequent call to *read()* or *write()*

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Returns new file offset if successful, or –1 on error

```c
lseek(fd, 0, SEEK_CUR); /* Returns current cursor loc of without change */
lseek(fd, 0, SEEK_SET); /* Start of file */
lseek(fd, 0, SEEK_END); /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END); /* Last byte of file */
lseek(fd, -10, SEEK_CUR); /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END); /* 10001 bytes past last byte of file */
```

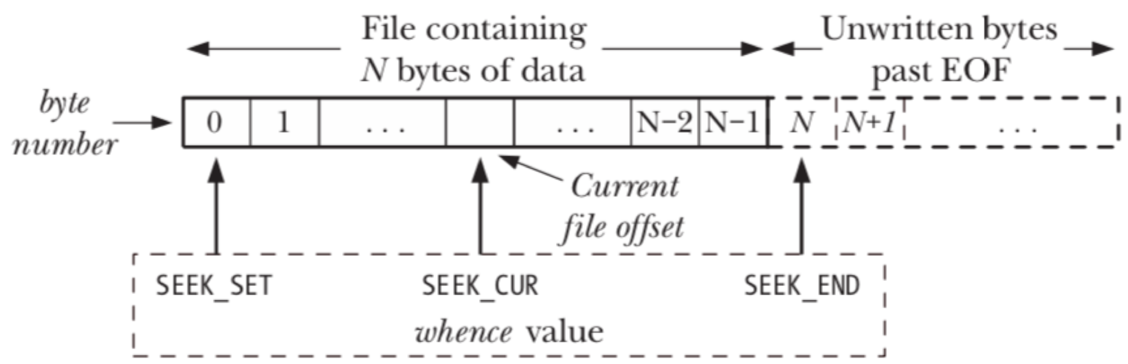


Figure 4-1: Interpreting the *whence* argument of *lseek()*

Generating an Executable File

C File (.c)

Compiler: translate from human readable to machine-specific code

Object File (.o)

- Intermediate machine-specific representation of just what is in a C file

Linker: bring together multiple object files so that all functions are known

Executable (no extension)

Our First Makefile

```
# The top rule is executed by default
all: hello

# Other rules are invoked as necessary

# Rule for creating the hello executable
hello: hello.c
	gcc hello.c -o hello
```

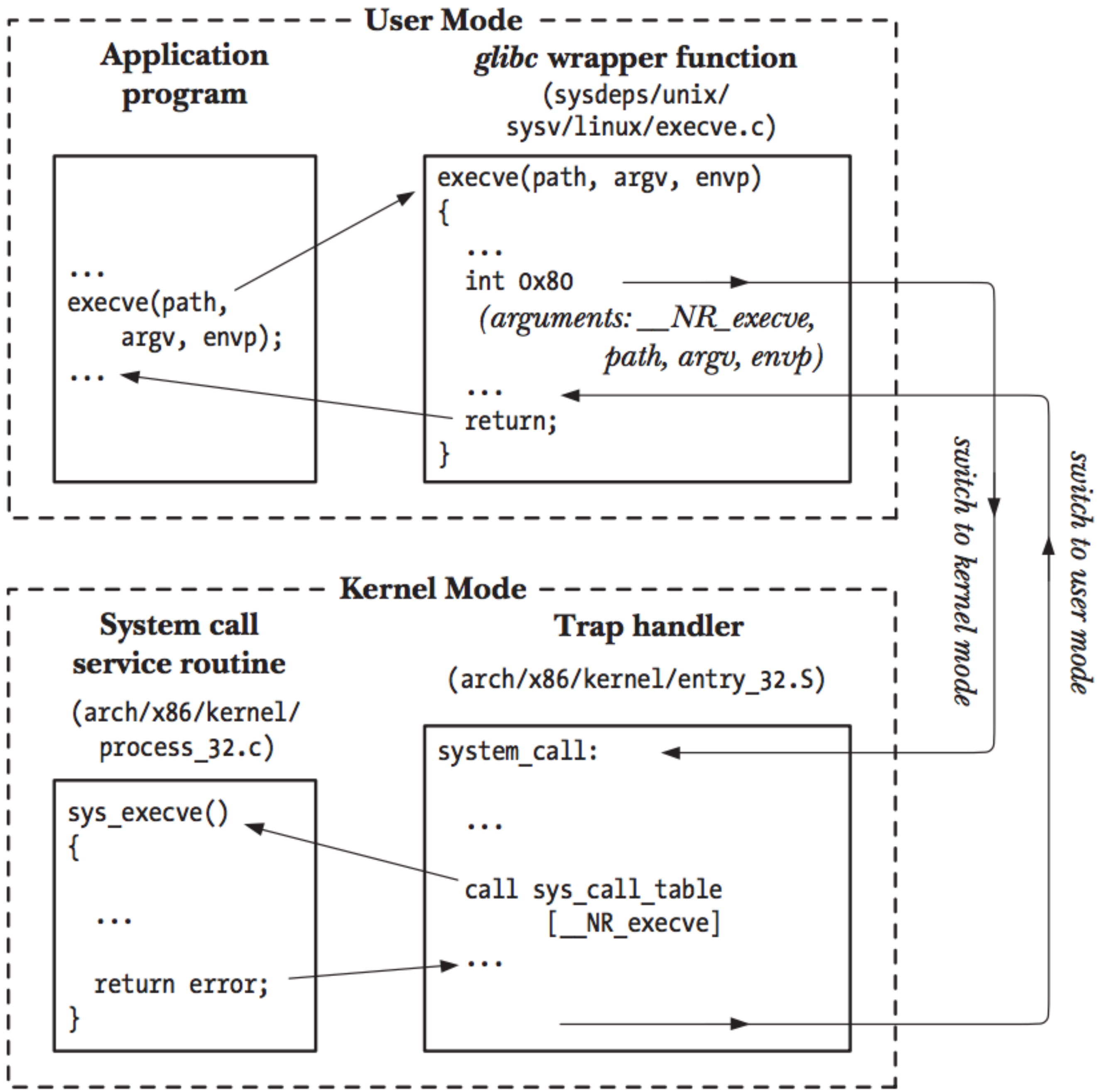
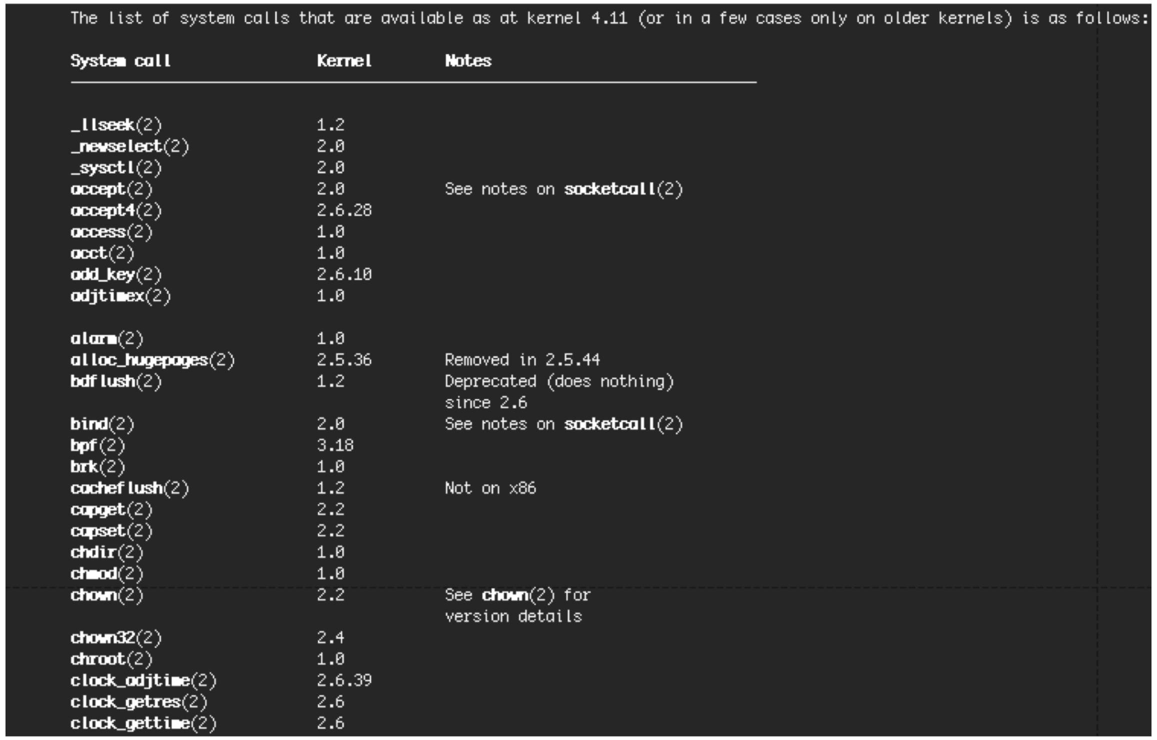


Figure 3-1: Steps in the execution of a system call

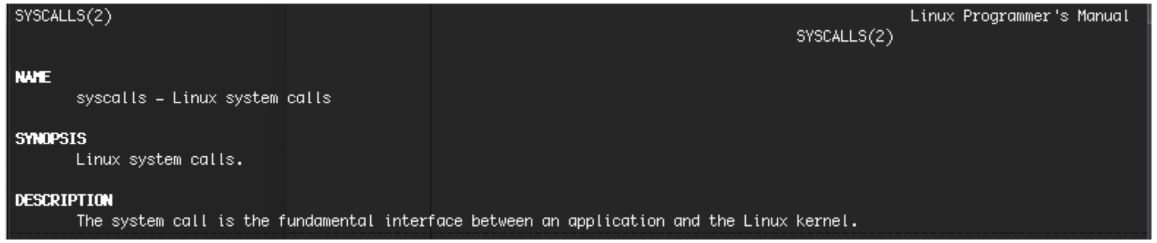


```
       The list of system calls that are available as at kernel 4.11 (or in a few cases only on older kernels) is as follows:

       System call                Kernel        Notes
       ───────────────────────────────────────────────────────────

       _llseek(2)                 1.2
       _newselect(2)              2.0
       _sysctl(2)                 2.0
       accept(2)                  2.0           See notes on socketcall(2)
       accept4(2)                 2.6.28
       access(2)                  1.0
       acct(2)                    1.0
       add_key(2)                 2.6.10
       adjtimex(2)                1.0

       alarm(2)                   1.0
       alloc_hugepages(2)         2.5.36        Removed in 2.5.44
       bdflush(2)                 1.2           Deprecated (does nothing)
                                                since 2.6
       bind(2)                    2.0           See notes on socketcall(2)
       bpf(2)                     3.18
       brk(2)                     1.0
       cacheflush(2)              1.2           Not on x86
       capget(2)                  2.2
       capset(2)                  2.2
       chdir(2)                   1.0
       chmod(2)                   1.0
       chown(2)                   2.2           See chown(2) for
                                                version details
       chown32(2)                 2.4
       chroot(2)                  1.0
       clock_adjtime(2)           2.6.39
       clock_getres(2)            2.6
       clock_gettime(2)           2.6
```



```
SYSCALLS(2)                                                             Linux Programmer's Manual
                                                                        SYSCALLS(2)

NAME
       syscalls - Linux system calls

SYNOPSIS
       Linux system calls.

DESCRIPTION
       The system call is the fundamental interface between an application and the Linux kernel.
```

Man syscalls

Mount Points

- We would like to provide a file system abstraction that makes it appear as though all of the storage resources live within one common directory tree (starting from /)
- Linux solution: provide a way to virtually make a file system appear as though it is a directory with the root directory

To the instance…

Create a new file system in a file:

```
dd if=/dev/zero of=~/myfile bs=512 count=4096
mkfs.ext3 ~/myfile
sudo mkdir /myfs
sudo mount ~fagg/myfile /myfs
```

Unmount the new file system:

```
sudo umount /myfs
```

- Note: not allowed if the fs is being accessed at that instant

Process Elements

- Two essential elements of a process are:

Program code

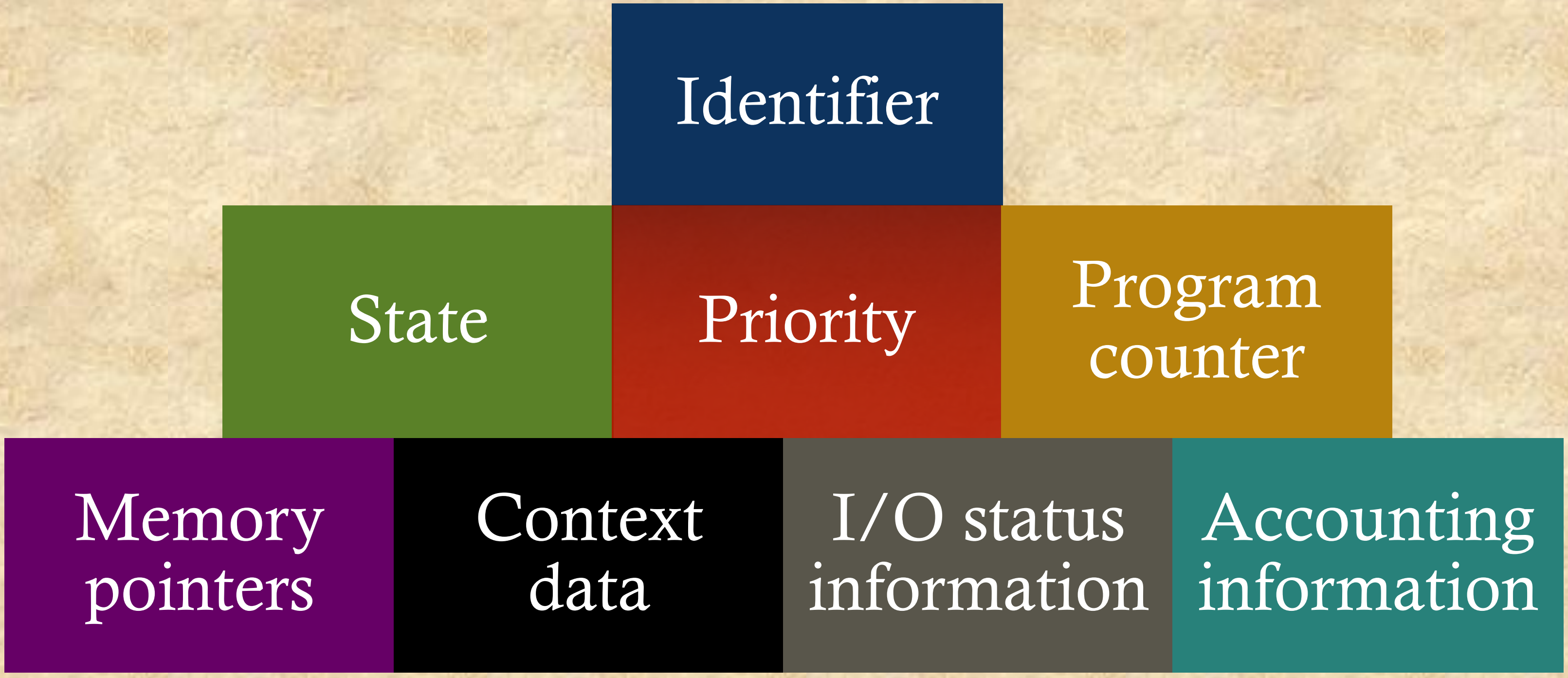
- which may be shared with other processes that are executing the same program

A set of data associated with that code

- when the processor begins to execute the program code, we refer to this executing entity as a ***process***

Process Elements

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



Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

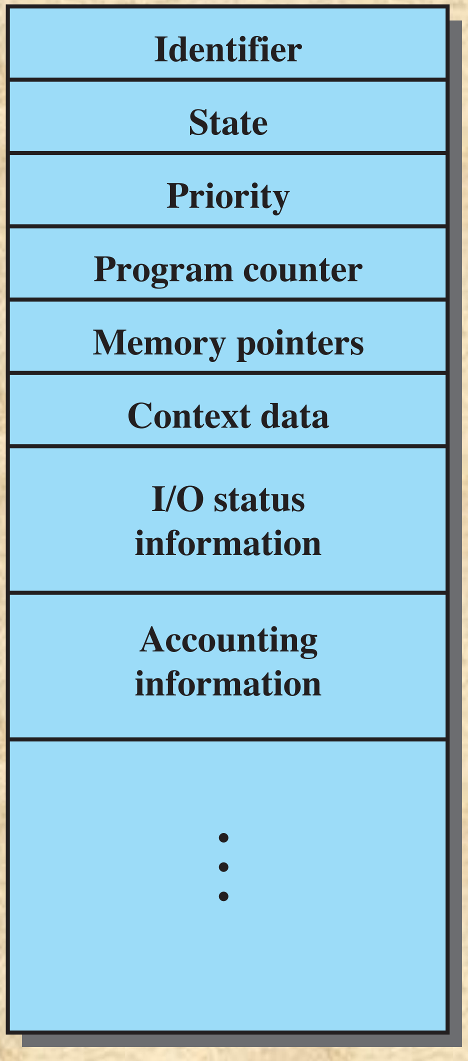


Figure 3.1 Simplified Process Control Block

Table 3.1 Reasons for Process Creation

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Table 3.2

Reasons for Process Termination

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

(Table is located on page 111
in the textbook)

Five-State Process Model

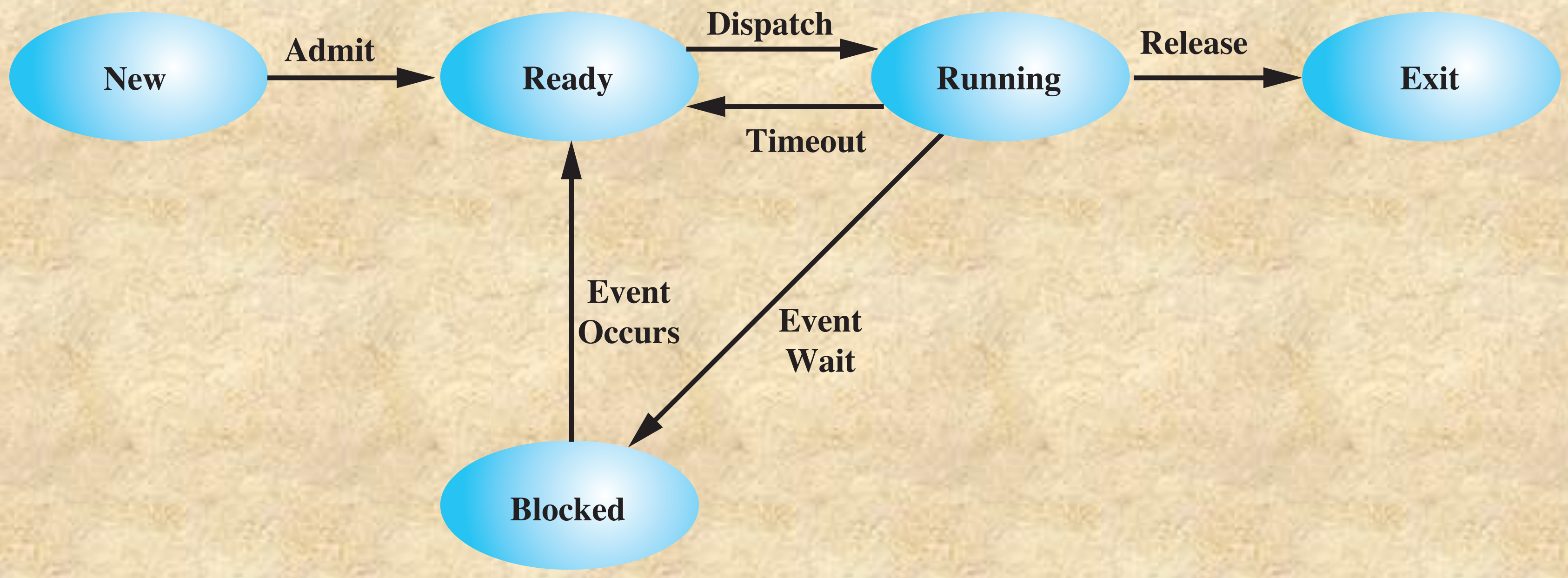
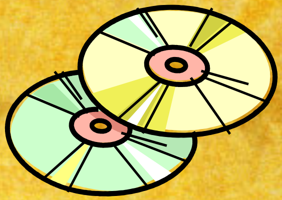
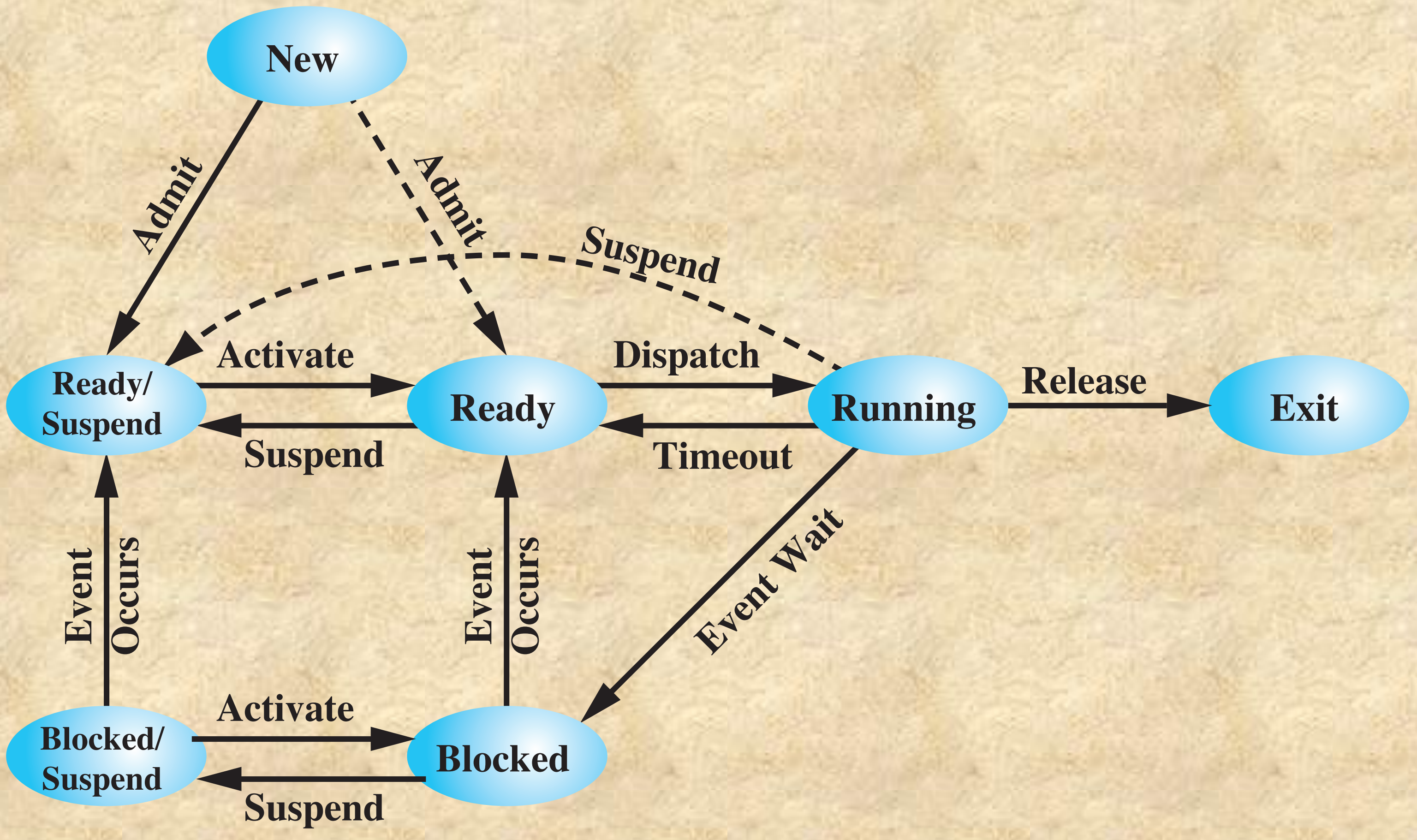


Figure 3.6 Five-State Process Model



Suspended Processes

- Swapping
  - Involves moving part of all of a process from main memory to disk
  - When none of the processes in main memory is in the Ready state, the OS swaps one of the blocked processes out on to disk into a suspend queue
    - This is a queue of existing processes that have been temporarily kicked out of main memory, or suspended
    - The OS then brings in another process from the suspend queue or it honors a new-process request
    - Execution then continues with the newly arrived process
  - Swapping, however, is an I/O operation and therefore there is the potential for making the problem worse, not better. Because disk I/O is generally the fastest I/O on a system, swapping will usually enhance performance



(b) With Two Suspend States

Figure 3.9 Process State Transition Diagram with Suspend States

Table 3.4
Typical Elements of a Process Image

User Data

The modifiable part of the user space. May include program data, a user stack area, and programs that may be modified.

User Program

The program to be executed.

Stack

Each process has one or more last-in-first-out (LIFO) stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls.

Process Control Block

Data needed by the OS to control the process (see Table 3.5).

Table 3.5 Typical Elements of a Process Control Block

(page 1 of 2)

Process Identification

Identifiers

Numeric identifiers that may be stored with the process control block include

- Identifier of this process
- Identifier of the process that created this process (parent process)
- User identifier

Processor State Information

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode. Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.

Control and Status Registers

These are a variety of processor registers that are employed to control the operation of the processor. These include

- **Program counter:** Contains the address of the next instruction to be fetched
- **Condition codes:** Result of the most recent arithmetic or logical operation (e.g., sign, zero, carry, equal, overflow)
- **Status information:** Includes interrupt enabled/disabled flags, execution mode

Stack Pointers

Each process has one or more last-in-first-out (LIFO) system stacks associated with it. A stack is used to store parameters and calling addresses for procedure and system calls. The stack pointer points to the top of the stack.

(Table is located on page 125 in the textbook)

Table 3.5 Typical Elements of a Process Control Block

(page 2 of 2)

Process Control Information

Scheduling and State Information

This is information that is needed by the operating system to perform its scheduling function. Typical items of information:

- **Process state***:* Defines the readiness of the process to be scheduled for execution (e.g., running, ready, waiting, halted).
- **Priority:** One or more fields may be used to describe the scheduling priority of the process. In some systems, several values are required (e.g., default, current, highest-allowable)
- **Scheduling-related information:** This will depend on the scheduling algorithm used. Examples are the amount of time that the process has been waiting and the amount of time that the process executed the last time it was running.
- **Event:** Identity of event the process is awaiting before it can be resumed.

Data Structuring

A process may be linked to other process in a queue, ring, or some other structure. For example, all processes in a waiting state for a particular priority level may be linked in a queue. A process may exhibit a parent-child (creator-created) relationship with another process. The process control block may contain pointers to other processes to support these structures.

Interprocess Communication

Various flags, signals, and messages may be associated with communication between two independent processes. Some or all of this information may be maintained in the process control block.

Process Privileges

Processes are granted privileges in terms of the memory that may be accessed and the types of instructions that may be executed. In addition, privileges may apply to the use of system utilities and services.

Memory Management

This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.

Resource Ownership and Utilization

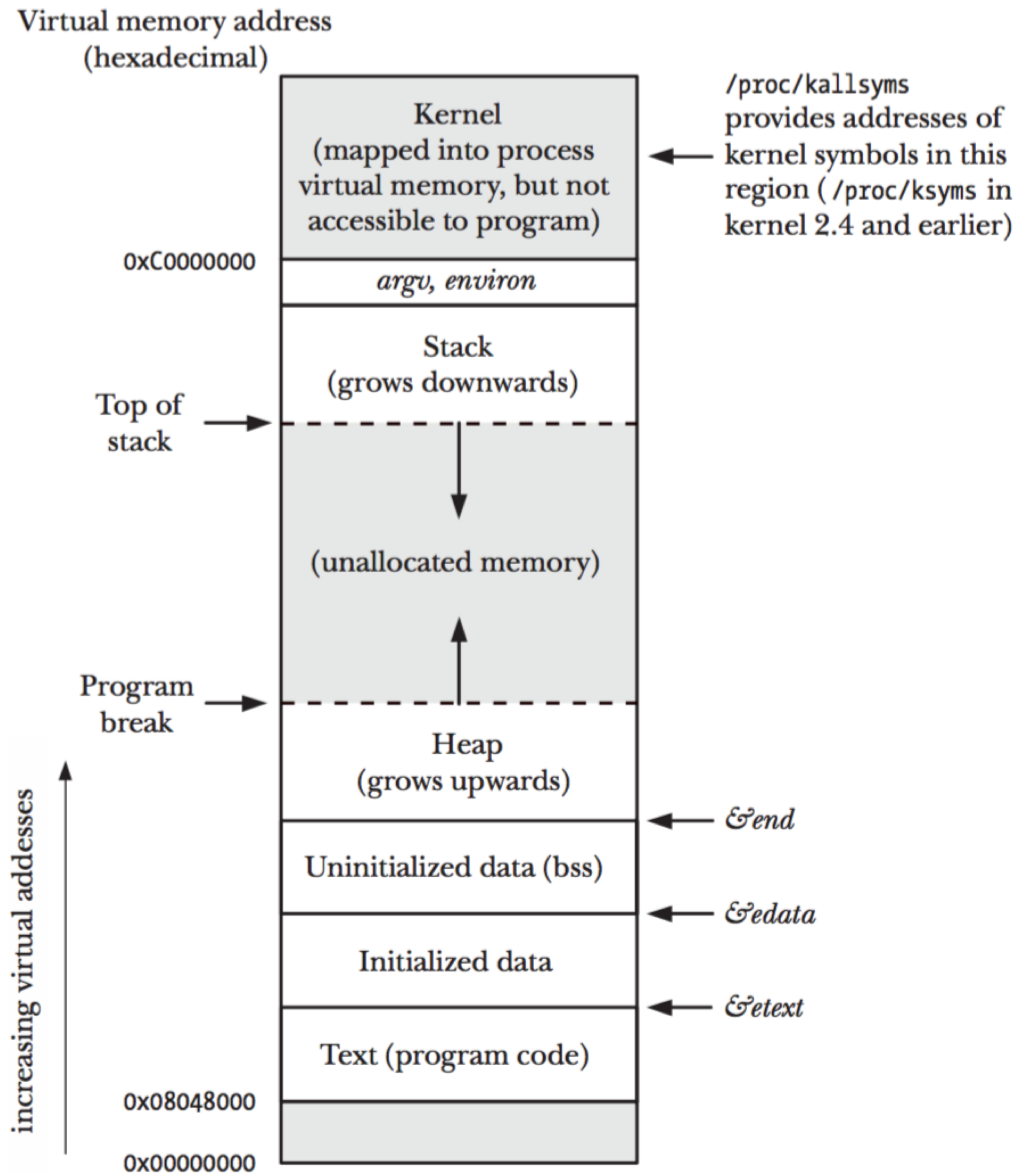
Resources controlled by the process may be indicated, such as opened files. A history of utilization of the processor or other resources may also be included; this information may be needed by the scheduler.

(Table is located
on page 125 in the textbook)

Role of the Process Control Block

- The most important data structure in an OS
  - Contains all of the information about a process that is needed by the OS
  - Blocks are read and/or modified by virtually every module in the OS
  - Defines the state of the OS
- Difficulty is not access, but protection
  - A bug in a single routine could damage process control blocks, which could destroy the system's ability to manage the affected processes
  - A design change in the structure or semantics of the process control block could affect a number of modules in the OS

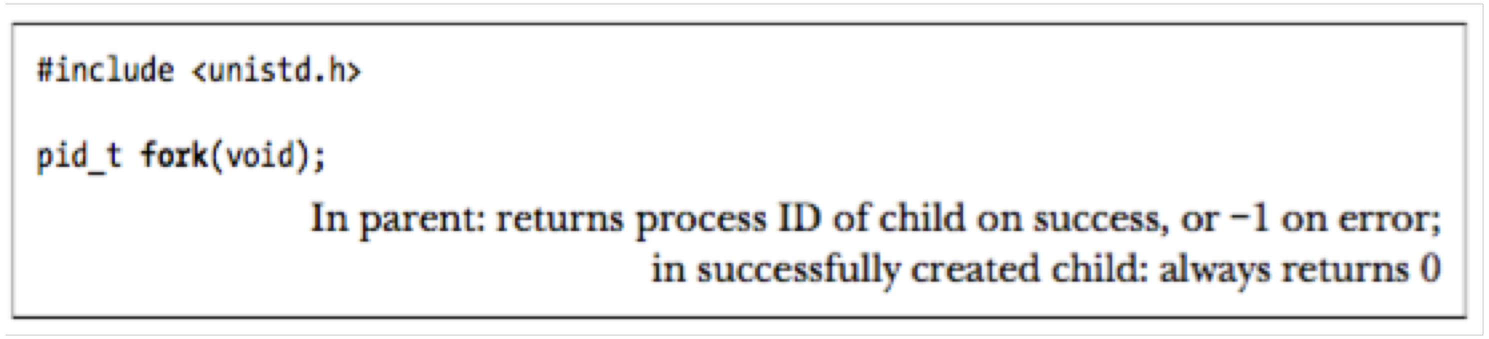
Virtual Memory



Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addresses

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

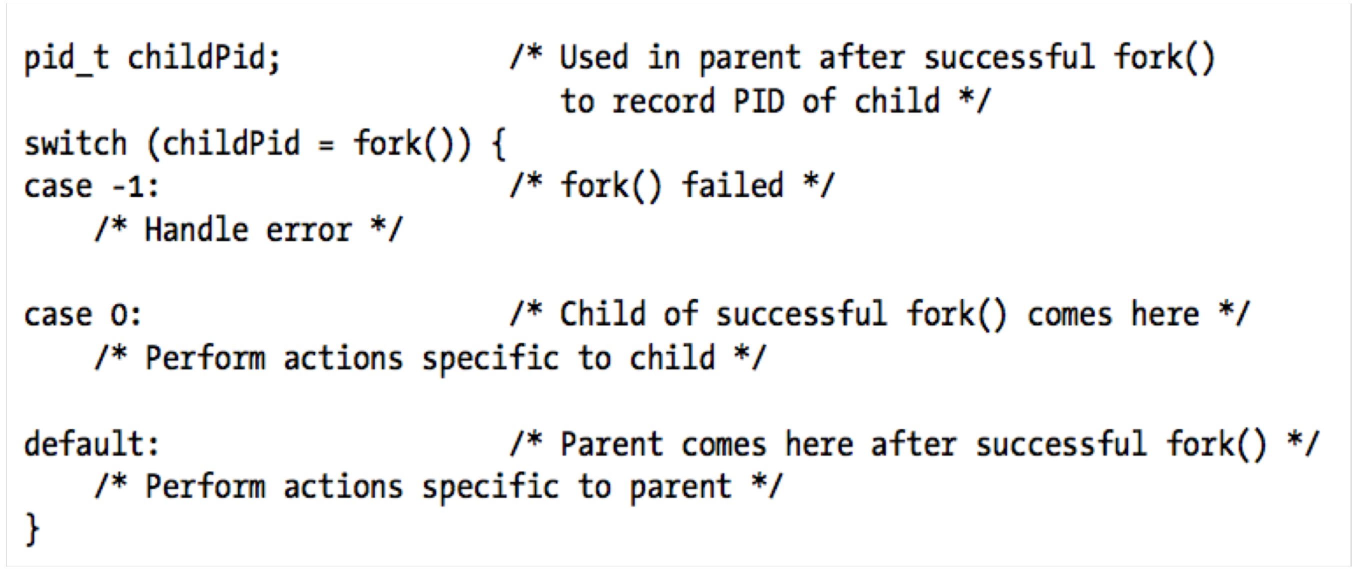
fork



```
#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



```
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 */
}
```

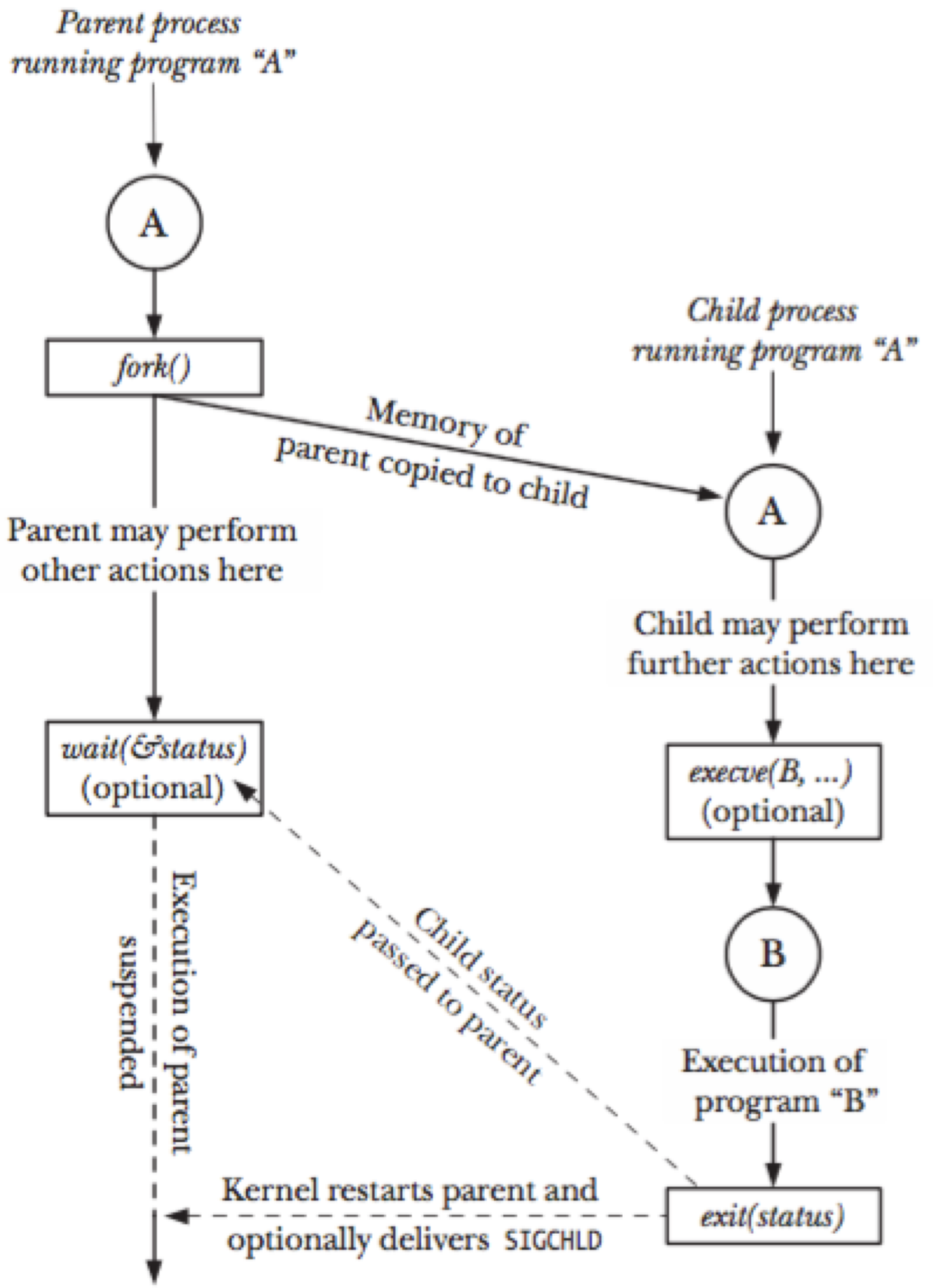

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent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

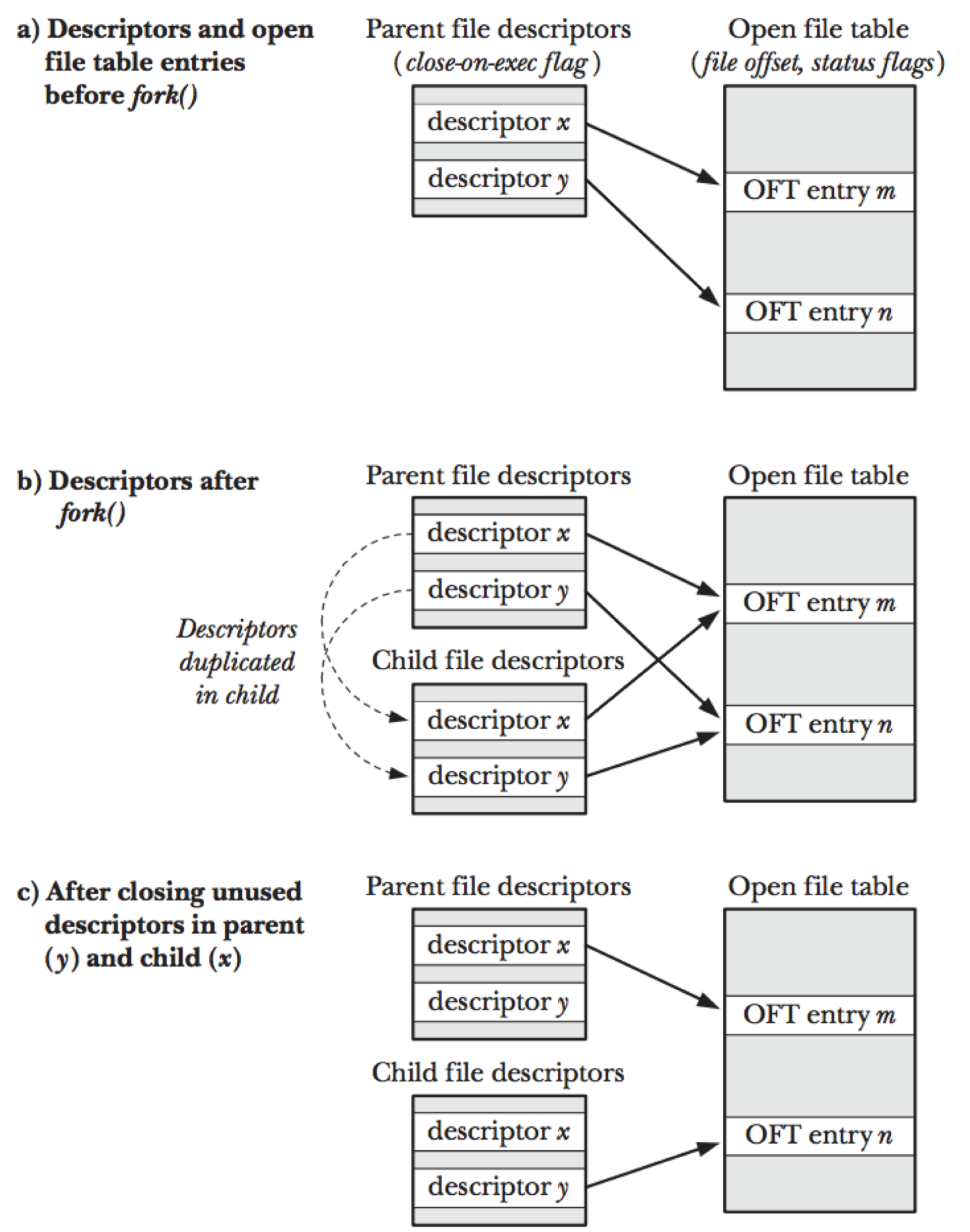


Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

Copy on write memory

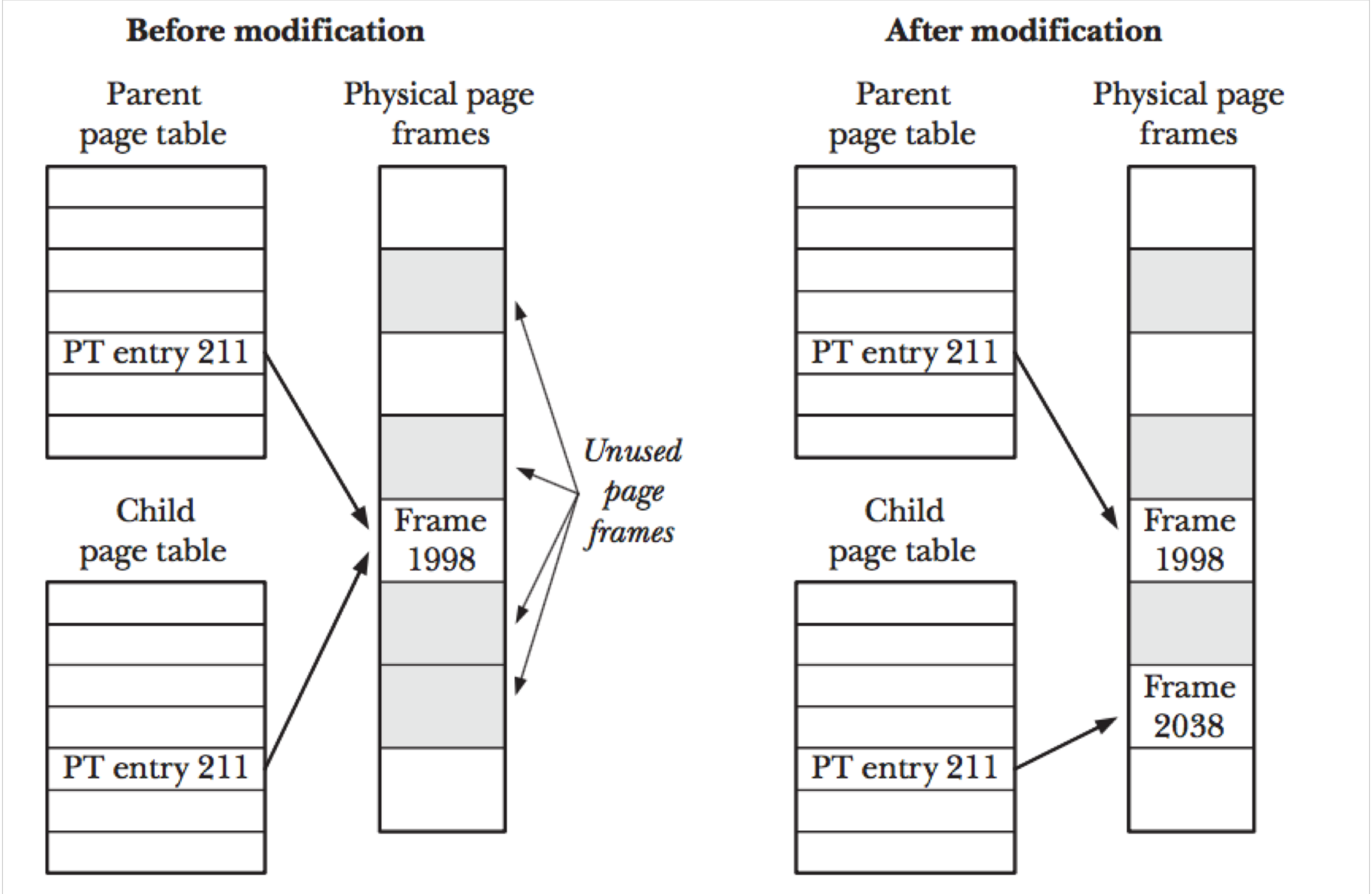


Figure 24-3: Page tables before and after modification of a shared copy-on-write page

```
#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);
```