Napredni operativni sistemi

UNIX, Linux i Android

Prof. dr Dragan Stojanović

Katedra za računarstvo
Elektronski fakultet u Nišu

UNIX, Linux, and Android

Chapter 10

History of UNIX and Linux

- UNICS
- PDP-11 UNIX
- Portable UNIX
- Berkeley UNIX
- Standard UNIX
- MINIX
- Linux

Linux Goals

- Simplicity, elegance, consistency
- Power and flexibility
- Lack of useless redundancy

Interfaces to Linux

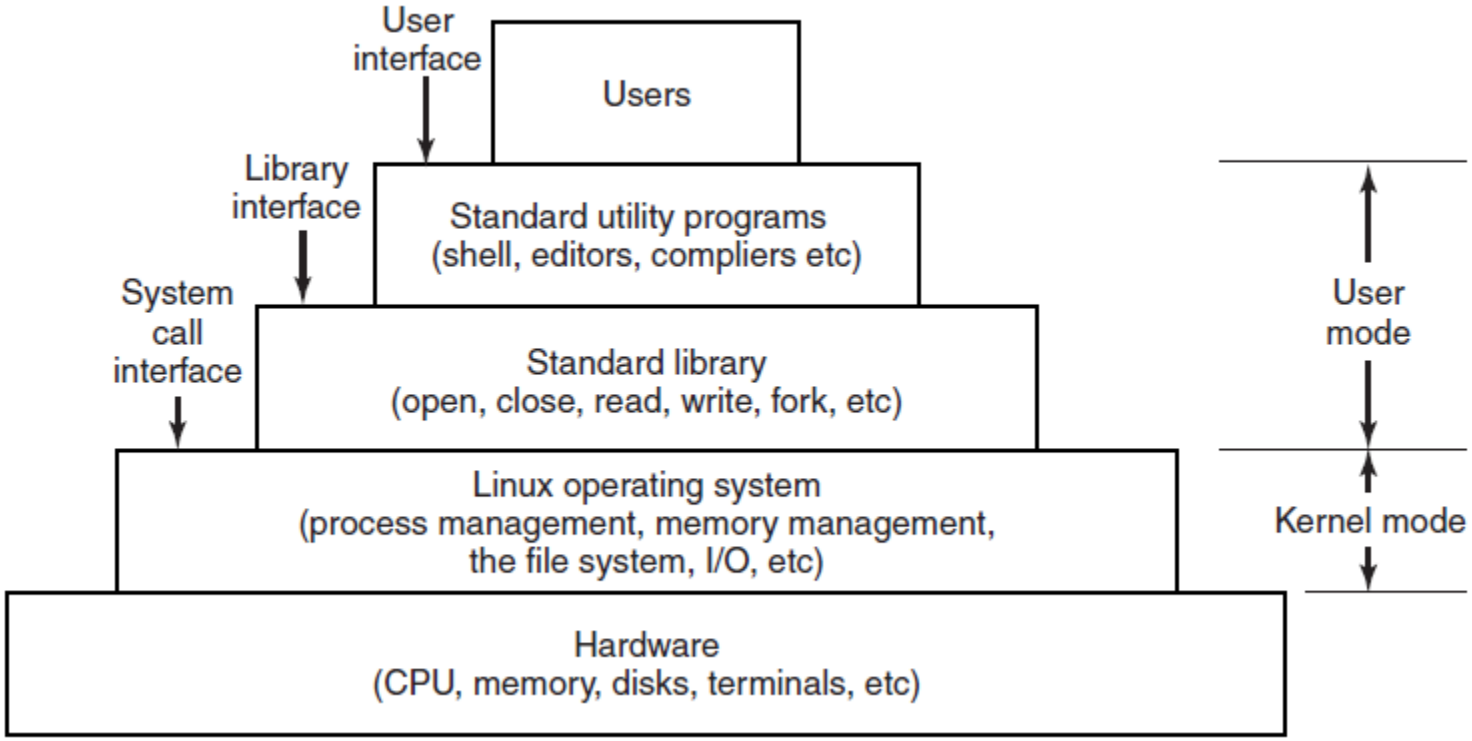

Figure 10-1. The layers in a Linux system.

The Shell

- Command line interface
  - Faster
  - More powerful
  - Easily extensible
- First word entered will be a program name
  - Commands take arguments
  - Wild card characters used
  - Filters, pipes used
- Shell scripts

Linux Utility Programs (1)

1. File and directory manipulation commands.
2. Filters.
3. Program development tools, such as editors and compilers.
4. Text processing.
5. System administration.
6. Miscellaneous.

Linux Utility Programs (2)

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
ps	List running processes
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Figure 10-2. A few of the common Linux utility programs required by POSIX.

Kernel Structure

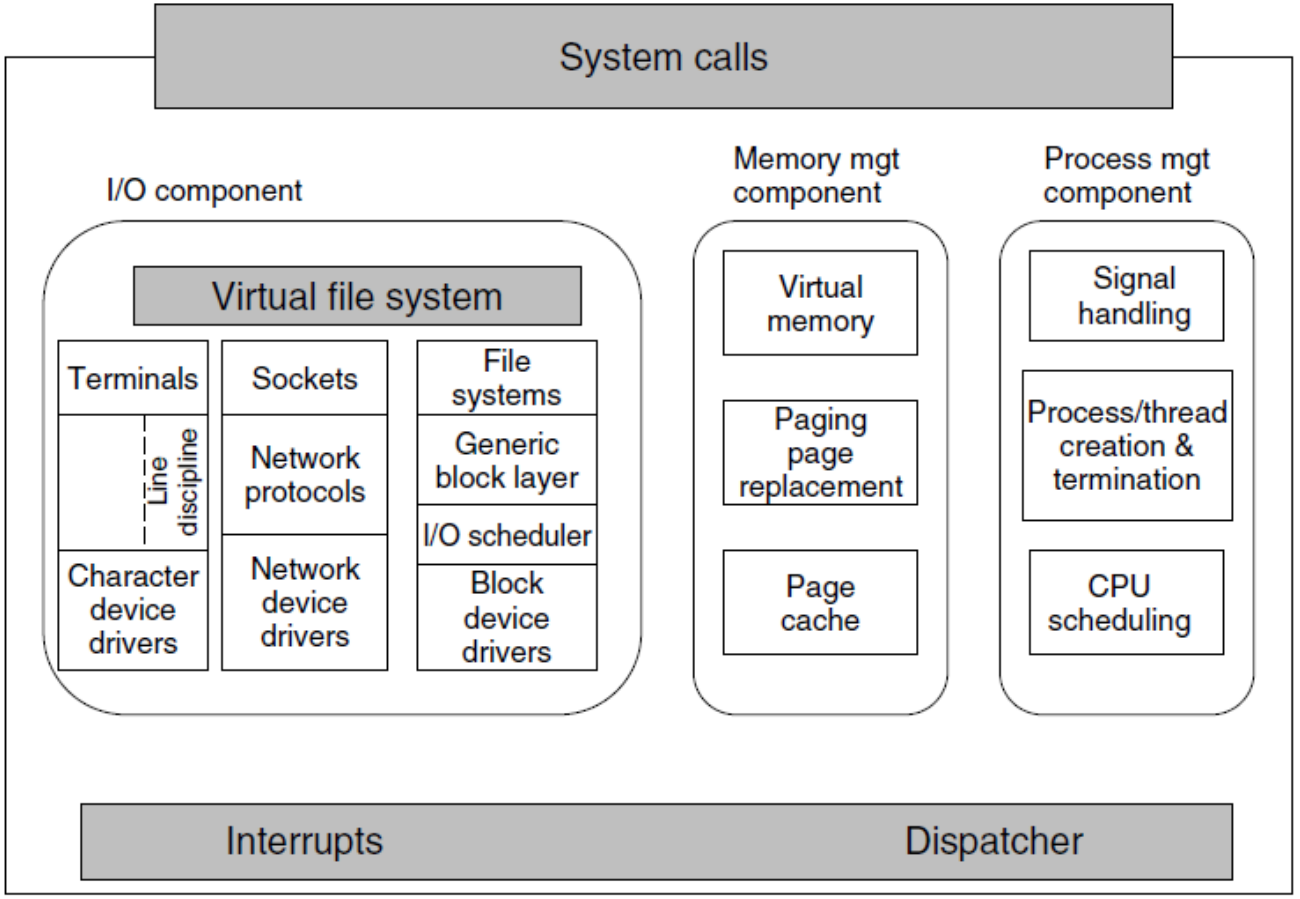

Figure 10-3. Structure of the Linux kernel

Processes: Fundamental Concepts (1)

```
pid = fork( );                  /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
        handle_error( );        /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                /* parent code goes here. /*/
} else {
                                /* child code goes here. /*/
}
```

Figure 10-4. Process creation in Linux.

Processes: Fundamental Concepts (2)

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Figure 10-5. Some of the signals required by POSIX.

Process Management System Calls in Linux (1)

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Figure 10-6. Some system calls relating to processes. The return code *s* is −1 if an error has occurred, *pid* is a process ID, and *residual* is the remaining time in the previous alarm. The parameters are what the names suggest.

Process Management System Calls in Linux (2)

```
while (TRUE) {                                  /* repeat forever /*/
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, params);          /* read input line from keyboard */

        pid = fork( );                          /* fork off a child process */
        if (pid < 0) {
                printf("Unable to fork0);       /* error condition */
                continue;                       /* repeat the loop */
        }

        if (pid != 0) {
                waitpid (–1, &status, 0);       /* parent waits for child */
        } else {
                execve(command, params, 0);     /* child does the work */
        }
}
```

Figure 10-7. A highly simplified shell.

Implementation of Processes and Threads in Linux (1)

Process descriptors

1. Scheduling parameters
2. Memory image
3. Signals
4. Machine registers
5. System call state
6. File descriptor table
7. Accounting
8. Kernel stack
9. Miscellaneous

Implementation of Processes and Threads in Linux (2)

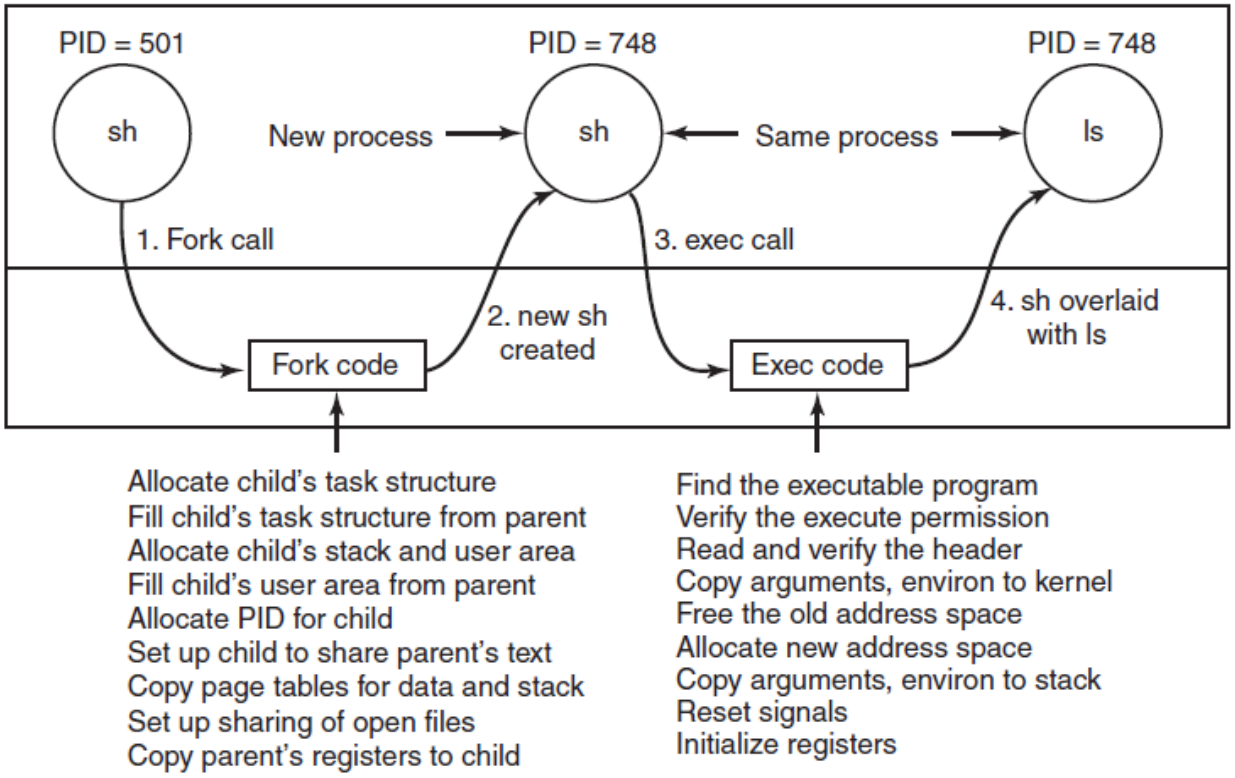

Figure 10-8. The steps in executing the command *ls* typed to the shell.

Threads in Linux

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as caller	New thread's parent is caller

Figure 10-9. Bits in the sharing flags bitmap.

Scheduling in Linux (1)

Three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Scheduling in Linux (2)

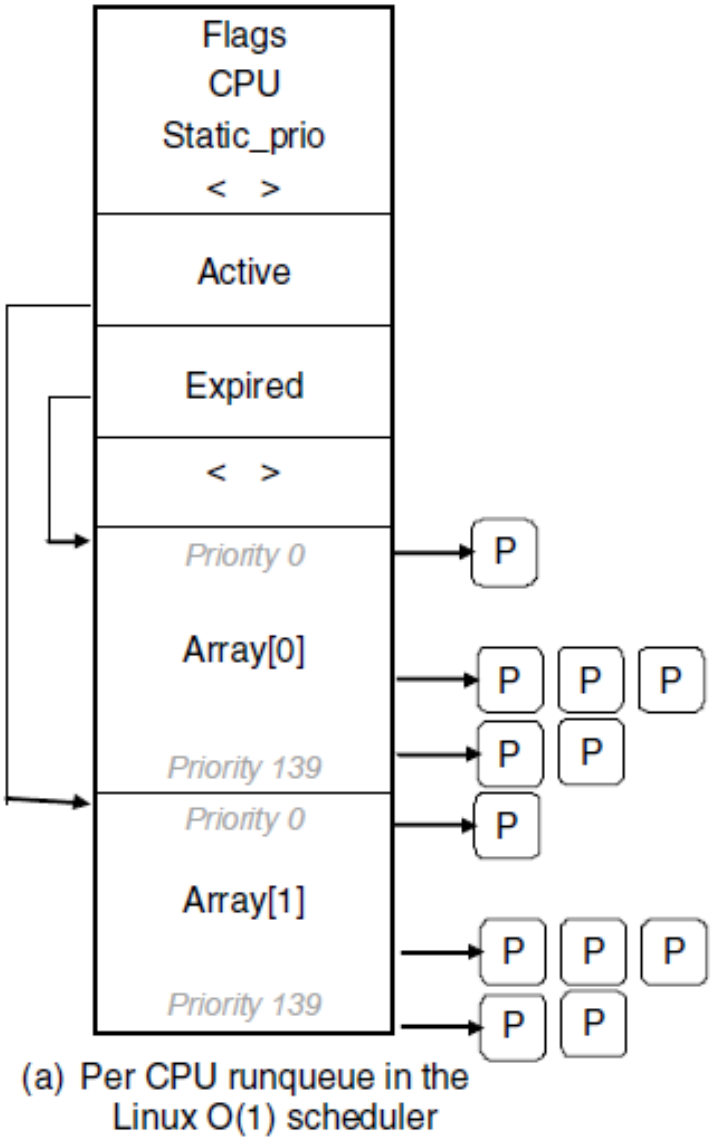

Figure 10-10. Illustration of Linux run queue data structures for (a) the Linux O(1) scheduler

Scheduling in Linux (3)

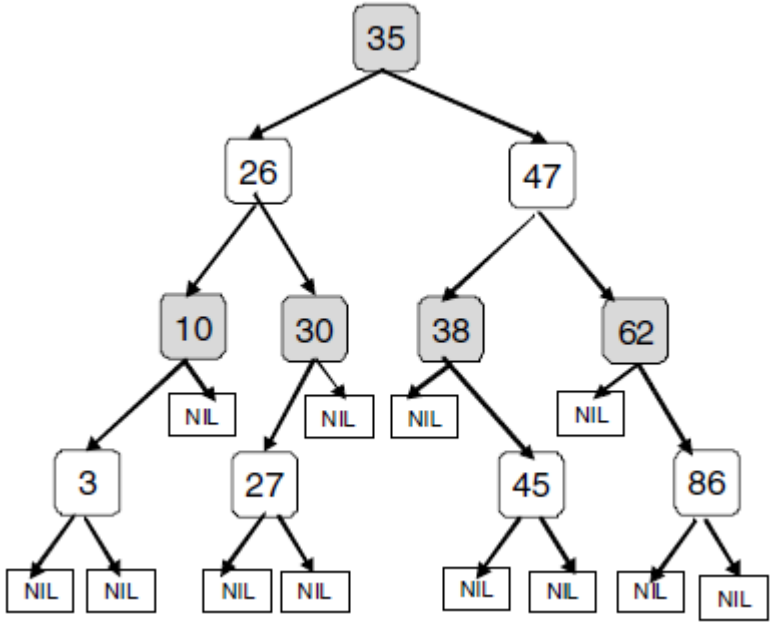

Figure 10-10. Illustration of Linux run queue data structures for (b) the Completely Fair Scheduler

Booting Linux

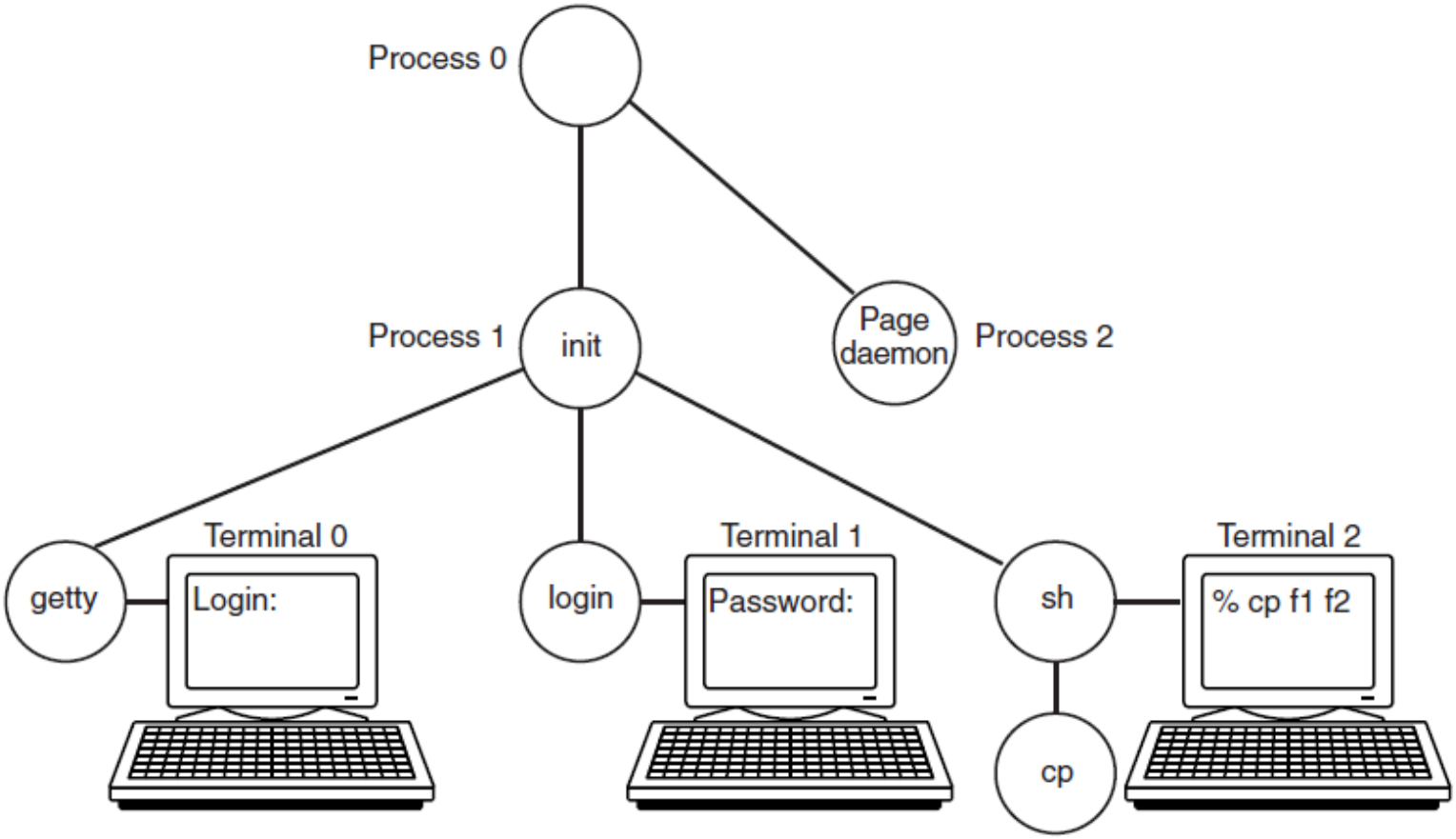

Figure 10-11. The sequence of processes used to boot some Linux systems.

Memory Management Fundamental Concepts (1)

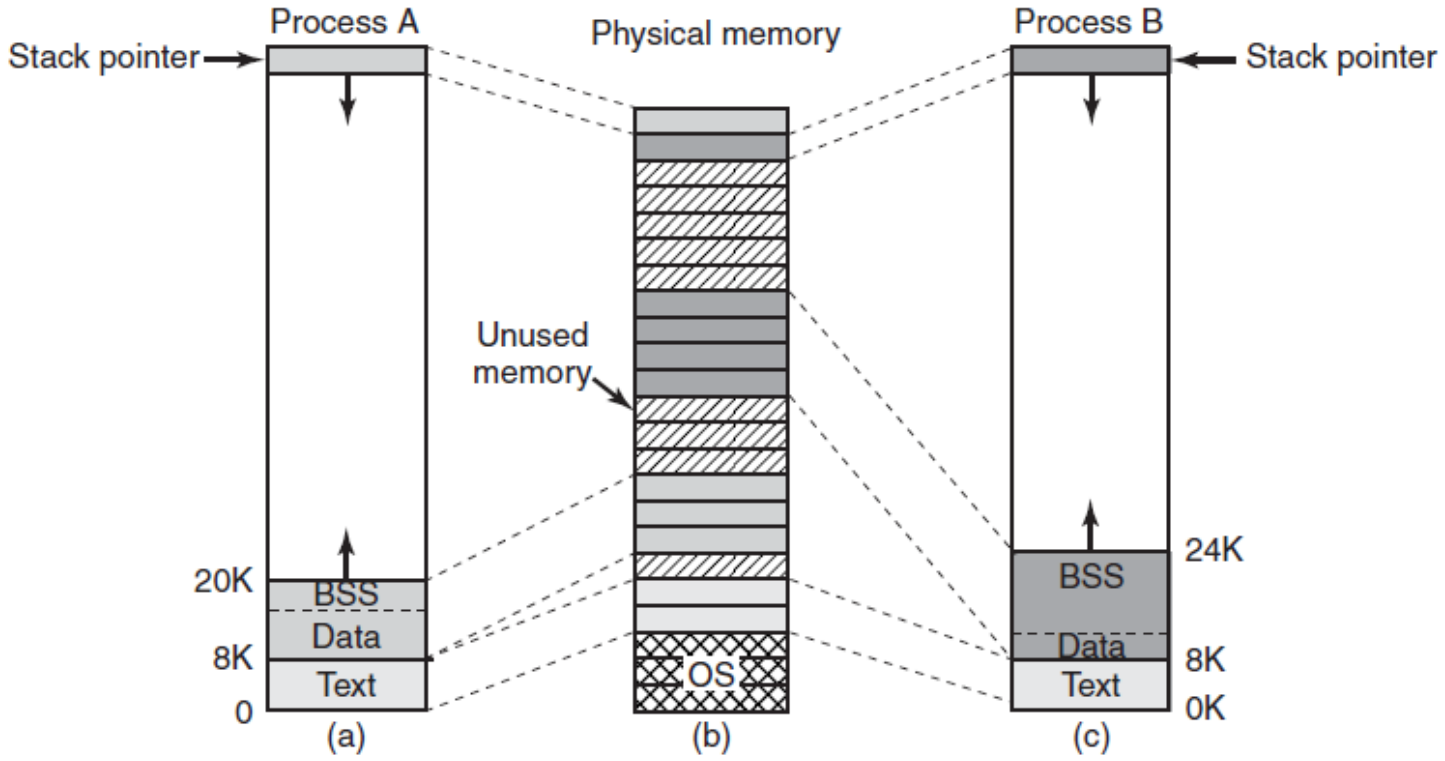

Figure 10-12. (a) Process A's virtual address space. (b) Physical memory. (c) Process B's virtual address space.

Memory Management Fundamental Concepts (2)

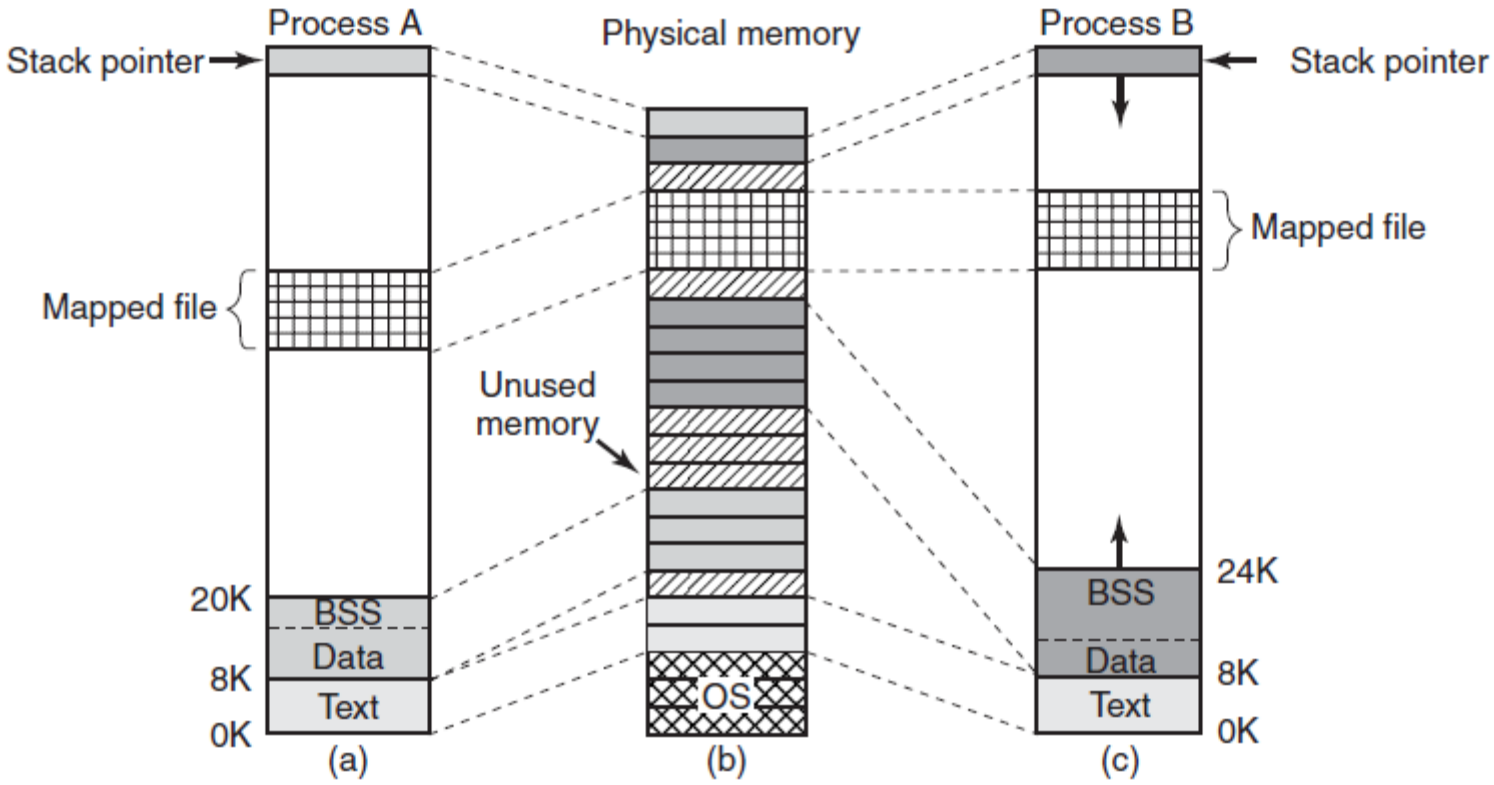

Figure 10-13. Two processes can share a mapped file.

Memory Management System Calls in Linux

System call	Description
s = brk(addr)	Change data segment size
a = mmap(addr, len, prot, flags, fd, offset)	Map a file in
s = unmap(addr, len)	Unmap a file

Figure 10-14. Some system calls relating to memory management. The return code *s* is −1 if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

Physical Memory Management (1)

Linux Memory Zones:

1. ZONE_DMA and ZONE_DMA32
2. ZONE_NORMAL
3. ZONE_HIGHMEM

Physical Memory Management (2)

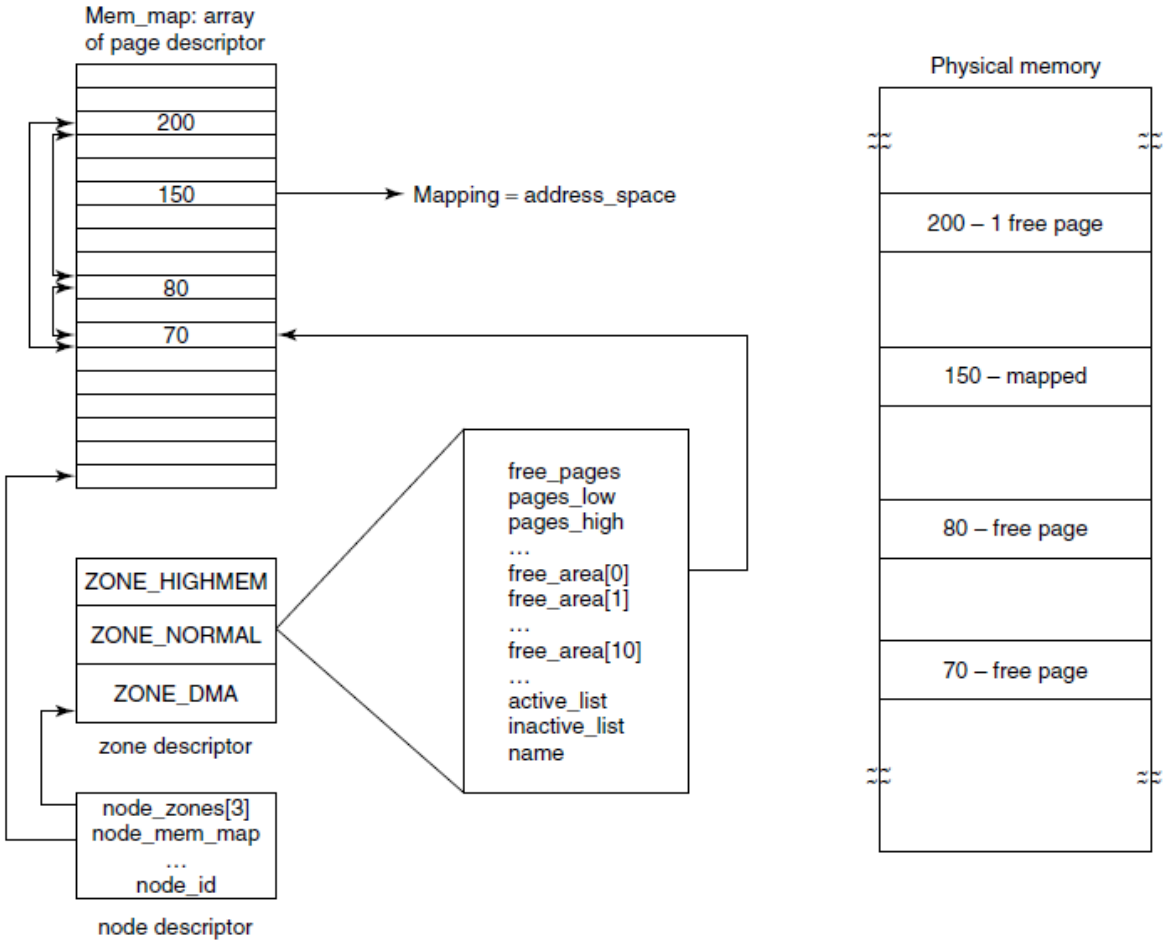

Figure 10-15. Linux main memory representation.

Physical Memory Management (3)

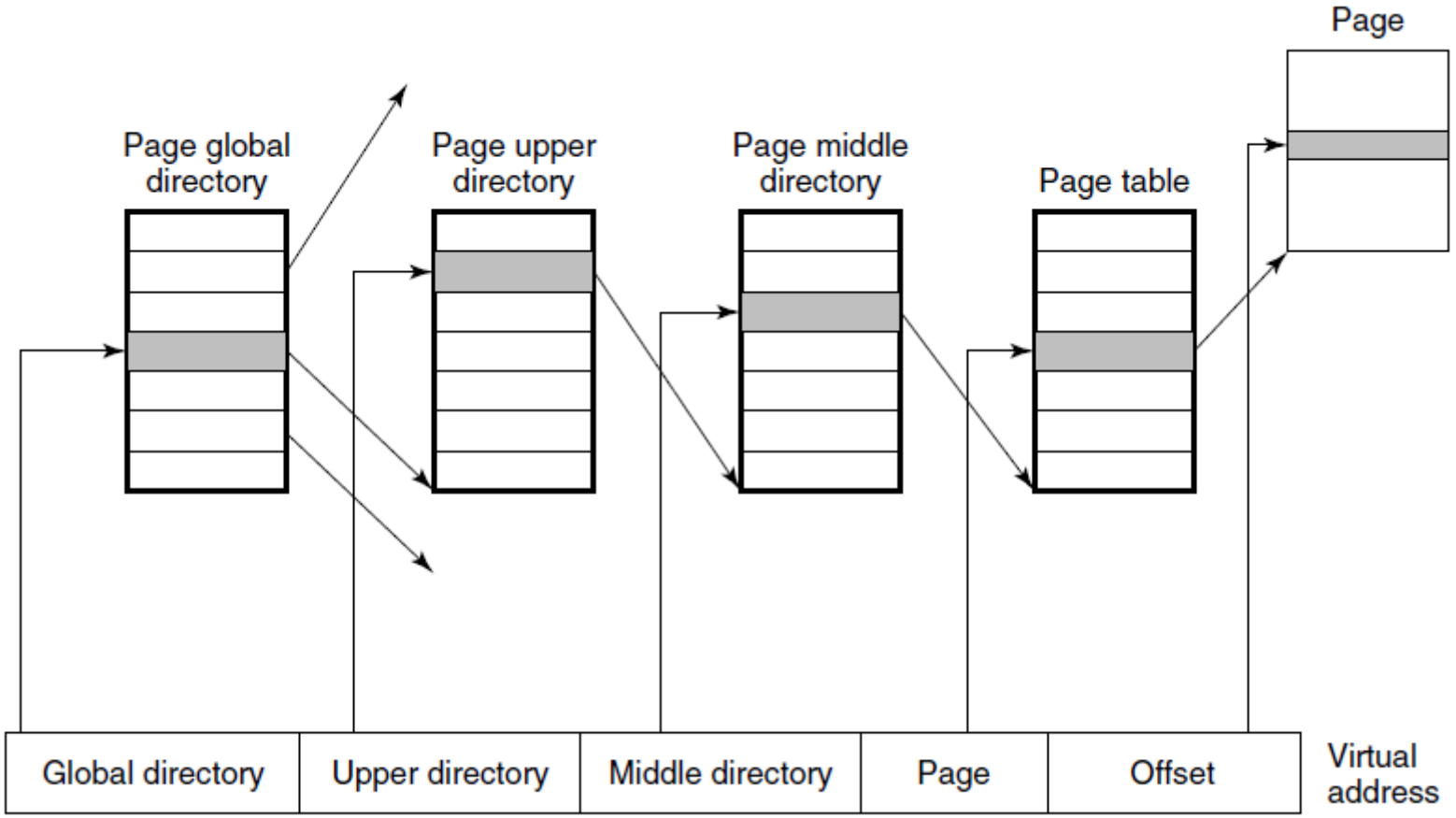

Figure 10-16. Linux uses four-level page tables.

Memory Allocation Mechanisms

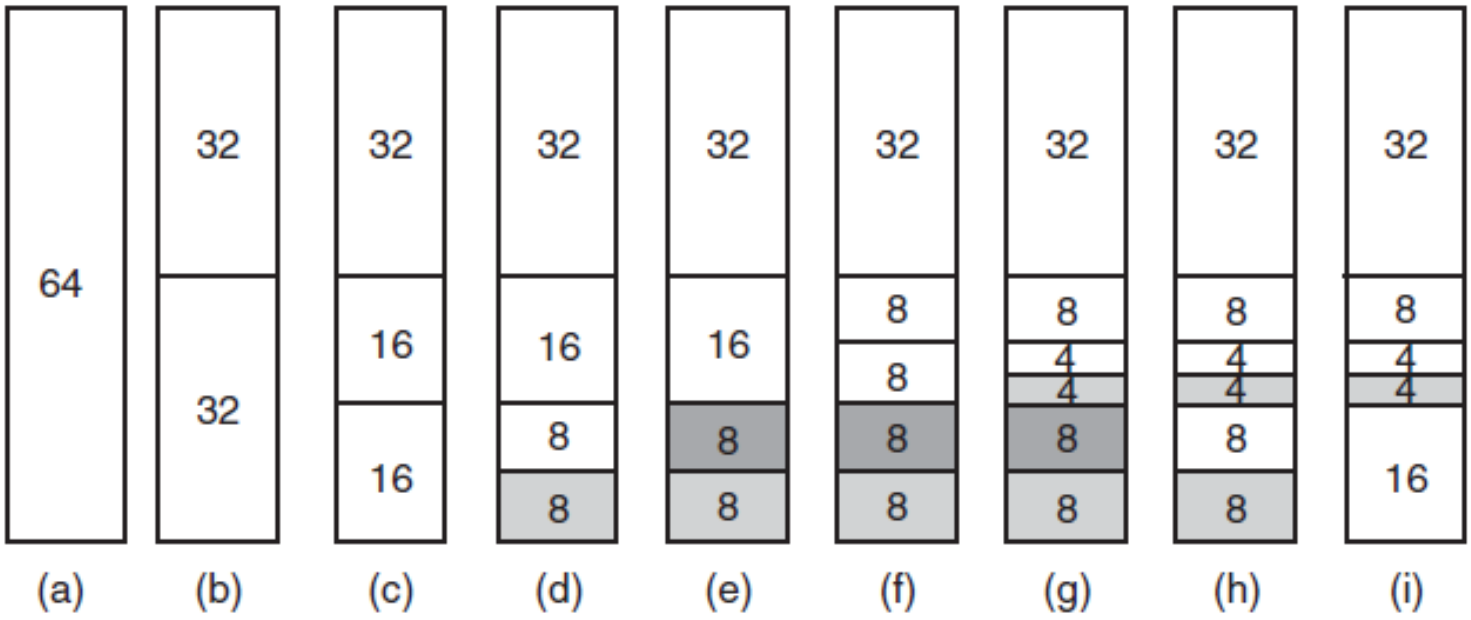

Figure 10-17. Operation of the buddy algorithm

The Page Replacement Algorithm

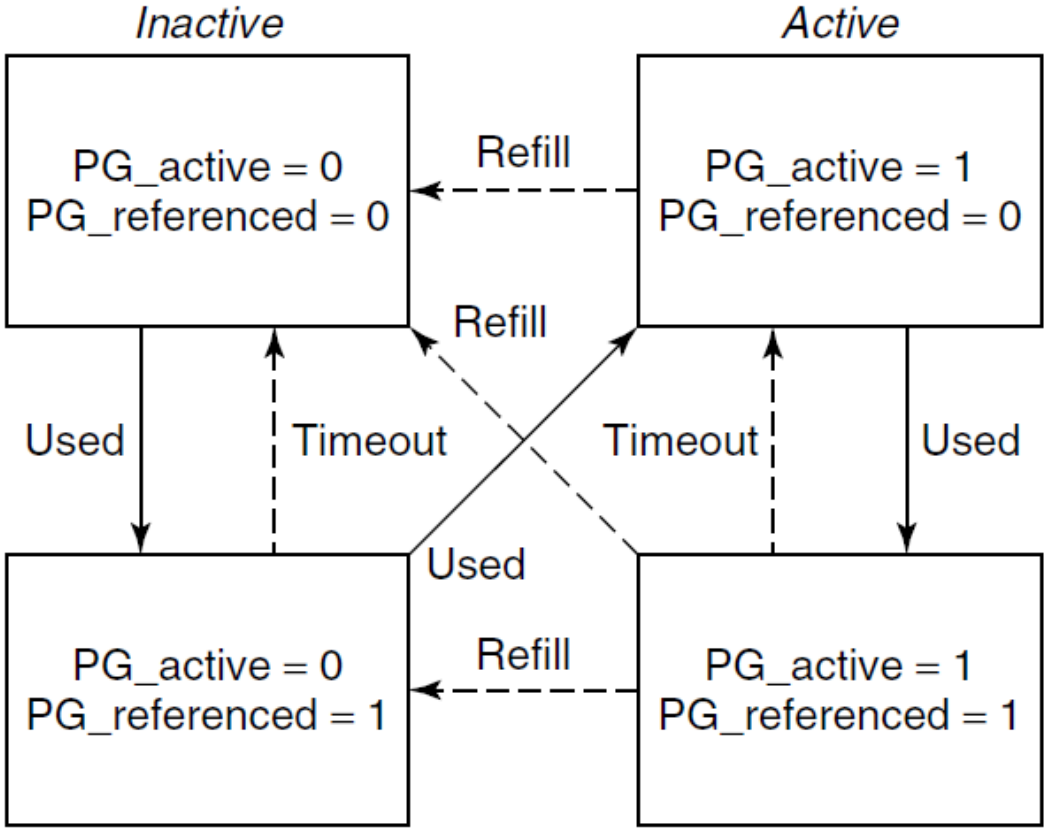

Figure 10-18. Page states considered in the page frame replacement algorithm.

I/O in Linux – Networking (1)

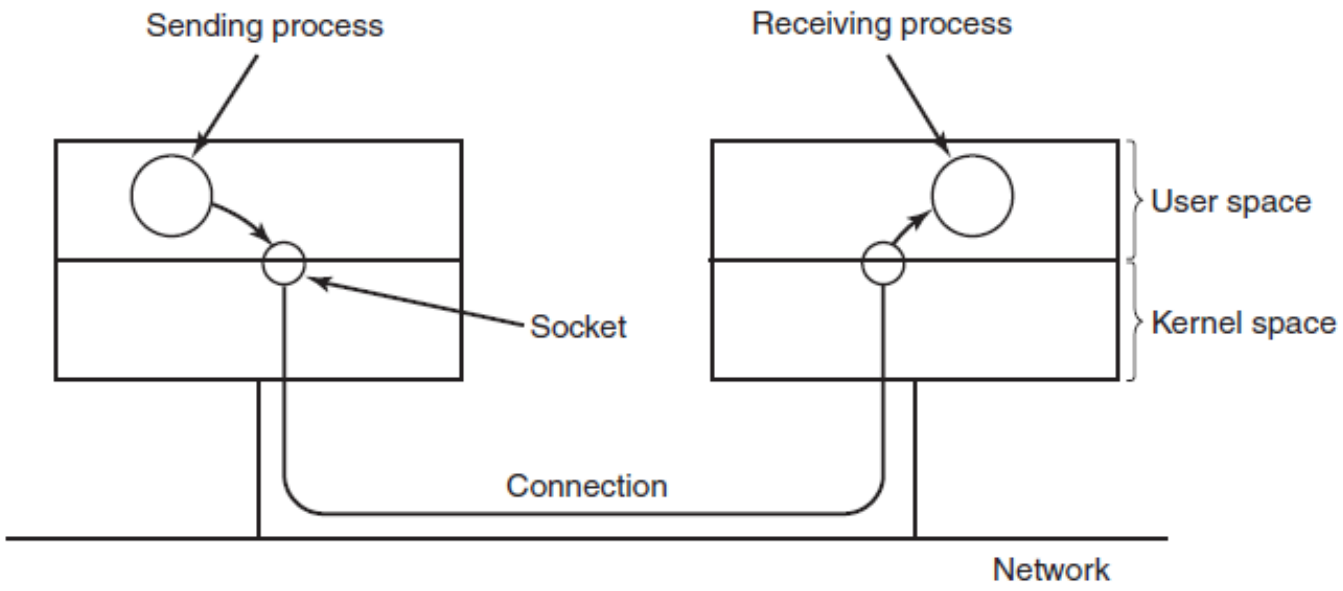

Figure 10-19. The uses of sockets for networking

I/O in Linux – Networking (2)

Common socket types:

1. Reliable connection-oriented byte stream.
2. Reliable connection-oriented packet stream.
3. Unreliable packet transmission.

Input/Output System Calls in Linux

Function call	Description
s = cfsetospeed(&termios, speed)	Set the output speed
s = cfsetispeed(&termios, speed)	Set the input speed
s = cfgetospeed(&termios, speed)	Get the output speed
s = cfgtetispeed(&termios, speed)	Get the input speed
s = tcsetattr(fd, opt, &termios)	Set the attributes
s = tcgetattr(fd, &termios)	Get the attributes

Figure 10-20. The main POSIX calls for managing the terminal.

Implementation of I/O in Linux (1)

Device	Open	Close	Read	Write	Ioctl	Other
Null	null	null	null	null	null	...
Memory	null	null	mem_read	mem_write	null	...
Keyboard	k_open	k_close	k_read	error	k_ioctl	...
Tty	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Printer	lp_open	lp_close	error	lp_write	lp_ioctl	...

Figure 10-21. Some of the file operations supported for typical character devices.

Implementation of I/O in Linux (2)

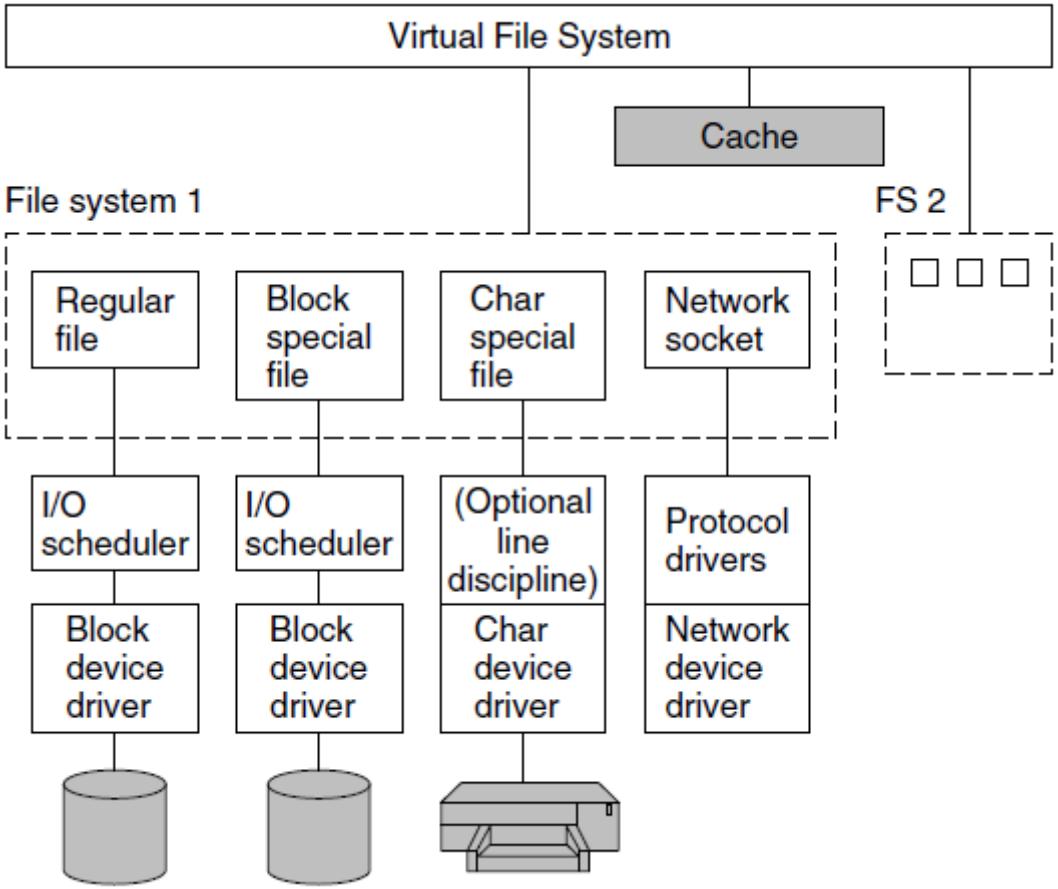

Figure 10-22. The Linux I/O system showing one file system in detail.

Linux File System: Fundamental Concepts (1)

Directory	Contents
bin	Binary (executable) programs
dev	Special files for I/O devices
etc	Miscellaneous system files
lib	Libraries
usr	User directories

Figure 10-23. Some important directories found in most Linux systems.

Linux File System: Fundamental Concepts (2)

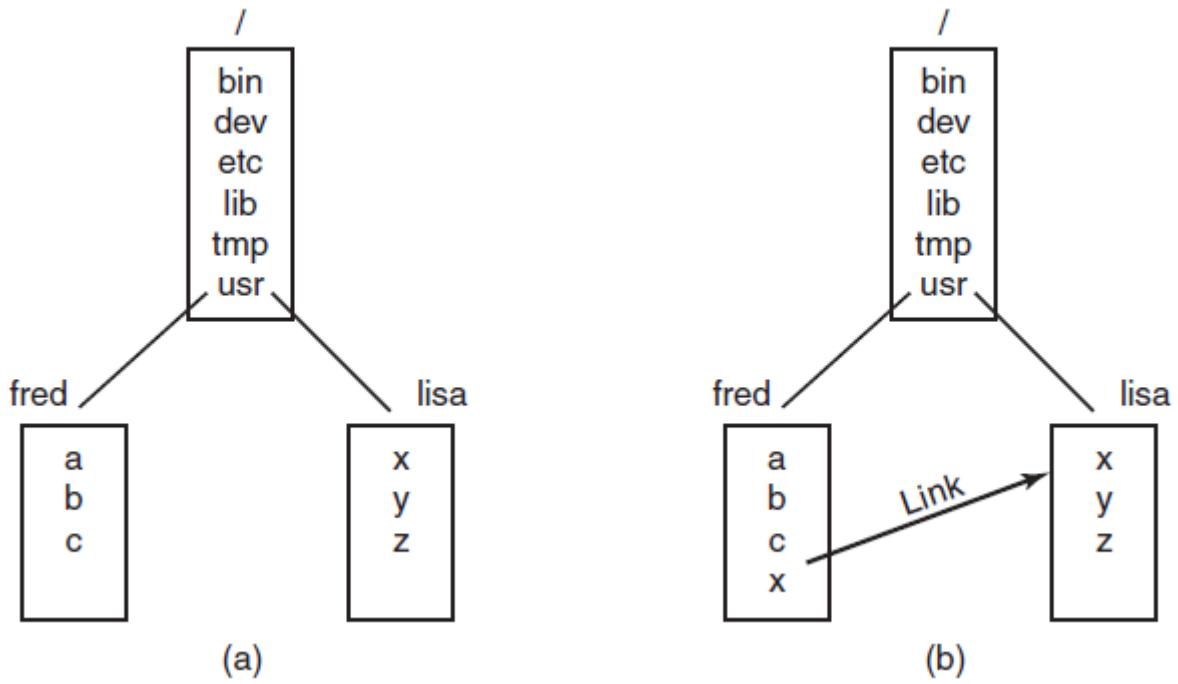

Figure 10-24. (a) Before linking. (b) After linking.

Linux File System: Fundamental Concepts (3)

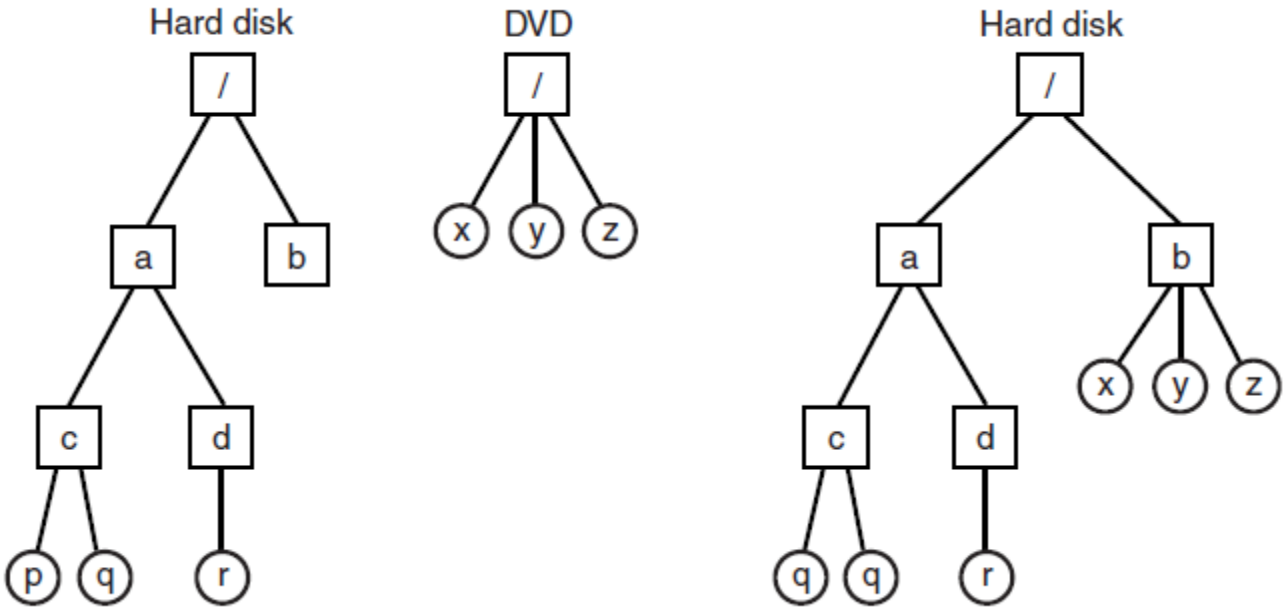

Figure 10-25. (a) Separate file systems. (b) After mounting.

Linux File System: Fundamental Concepts (4)

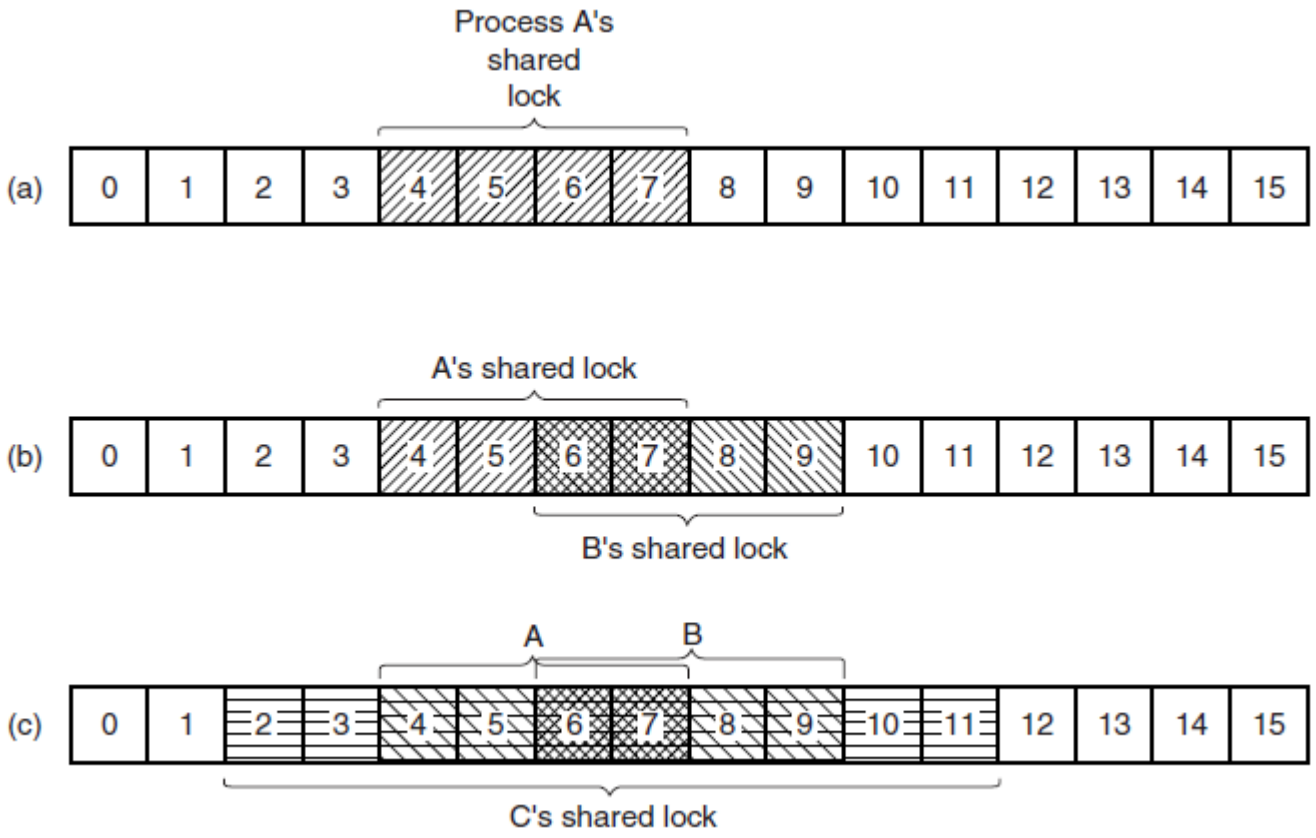

Figure 10-26. (a) A file with one lock. (b) Addition of a second lock. (c) A third lock.

File System Calls in Linux (1)

System call	Description
fd = creat(name, mode)	One way to create a new file
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
s = pipe(&fd[0])	Create a pipe
s = fcntl(fd, cmd, ...)	File locking and other operations

Figure 10-27. Some system calls relating to files. The return code *s* is *−1* if an error has occurred; *fd* is a file descriptor, and *position* is a file offset. The parameters should be self explanatory.

File System Calls in Linux (2)

Device the file is on
I-node number (which file on the device)
File mode (includes protection information)
Number of links to the file
Identity of the file's owner
Group the file belongs to
File size (in bytes)
Creation time
Time of last access
Time of last modification

Figure 10-28. The fields returned by the stat system call.

File System Calls in Linux (3)

System call	Description
s = mkdir(path, mode)	Create a new directory
s = rmdir(path)	Remove a directory
s = link(oldpath, newpath)	Create a link to an existing file
s = unlink(path)	Unlink a file
s = chdir(path)	Change the working directory
dir = opendir(path)	Open a directory for reading
s = closedir(dir)	Close a directory
dirent = readdir(dir)	Read one directory entry
rewinddir(dir)	Rewind a directory so it can be reread

Figure 10-29. Some system calls relating to directories. The return code *s* is *−1* if an error has occurred; *dir* identifies a directory stream, and *dirent* is a directory entry. The parameters should be self explanatory.

The Linux Virtual File System

Object	Description	Operation
Superblock	specific filesystem	read_inode, sync_fs
Dentry	directory entry, single component of a path	create, link
I-node	specific file	d_compare, d_delete
File	open file associated with a process	read, write

Figure 10-30. File system abstractions supported by the VFS.

The Linux Ext2 File System (1)

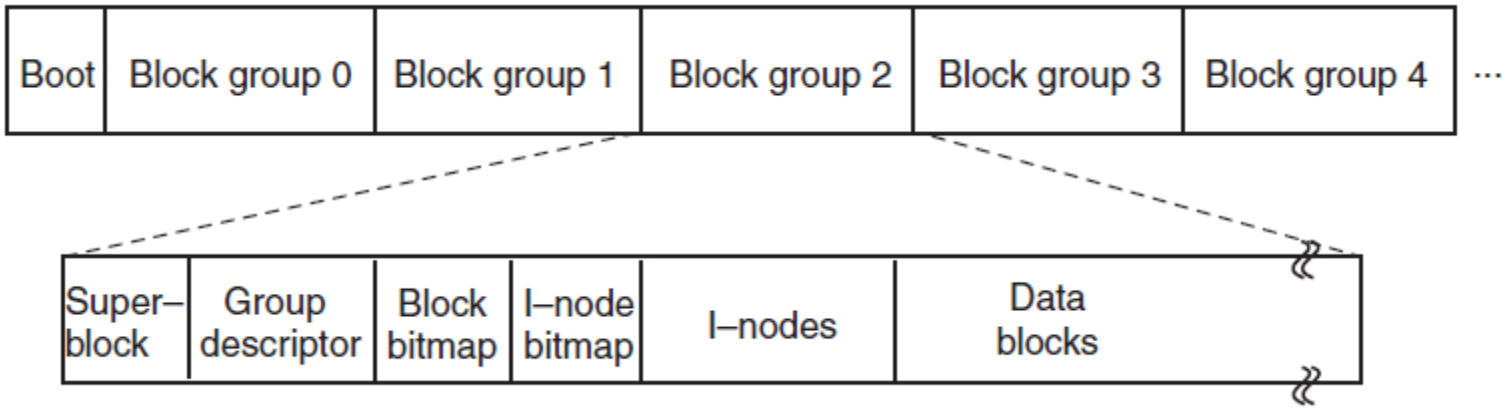

Figure 10-31. Disk layout of the Linux ext2 file system.

The Linux Ext2 File System (2)

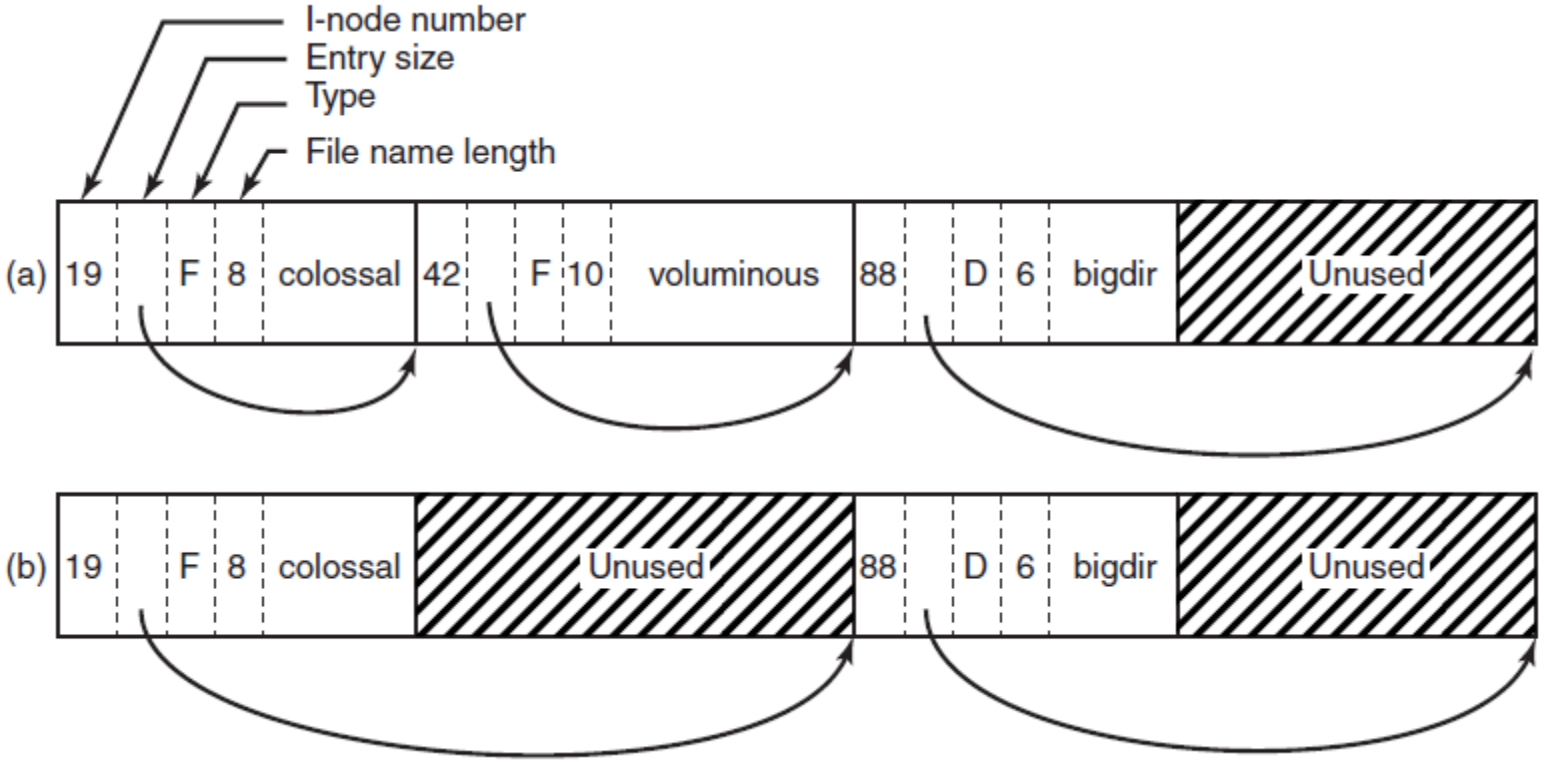

Figure 10-32. (a) A Linux directory with three files. (b) The same directory after the file voluminous has been removed.

The Linux Ext2 File System (3)

Field	Bytes	Description
Mode	2	File type, protection bits, setuid, setgid bits
Nlinks	2	Number of directory entries pointing to this i-node
Uid	2	UID of the file owner
Gid	2	GID of the file owner
Size	4	File size in bytes
Addr	60	Address of first 12 disk blocks, then 3 indirect blocks
Gen	1	Generation number (incremented every time i-node is reused)
Atime	4	Time the file was last accessed
Mtime	4	Time the file was last modified
Ctime	4	Time the i-node was last changed (except the other times)

10-33. Some fields in the i-node structure in Linux

The Linux Ext2 File System (4)

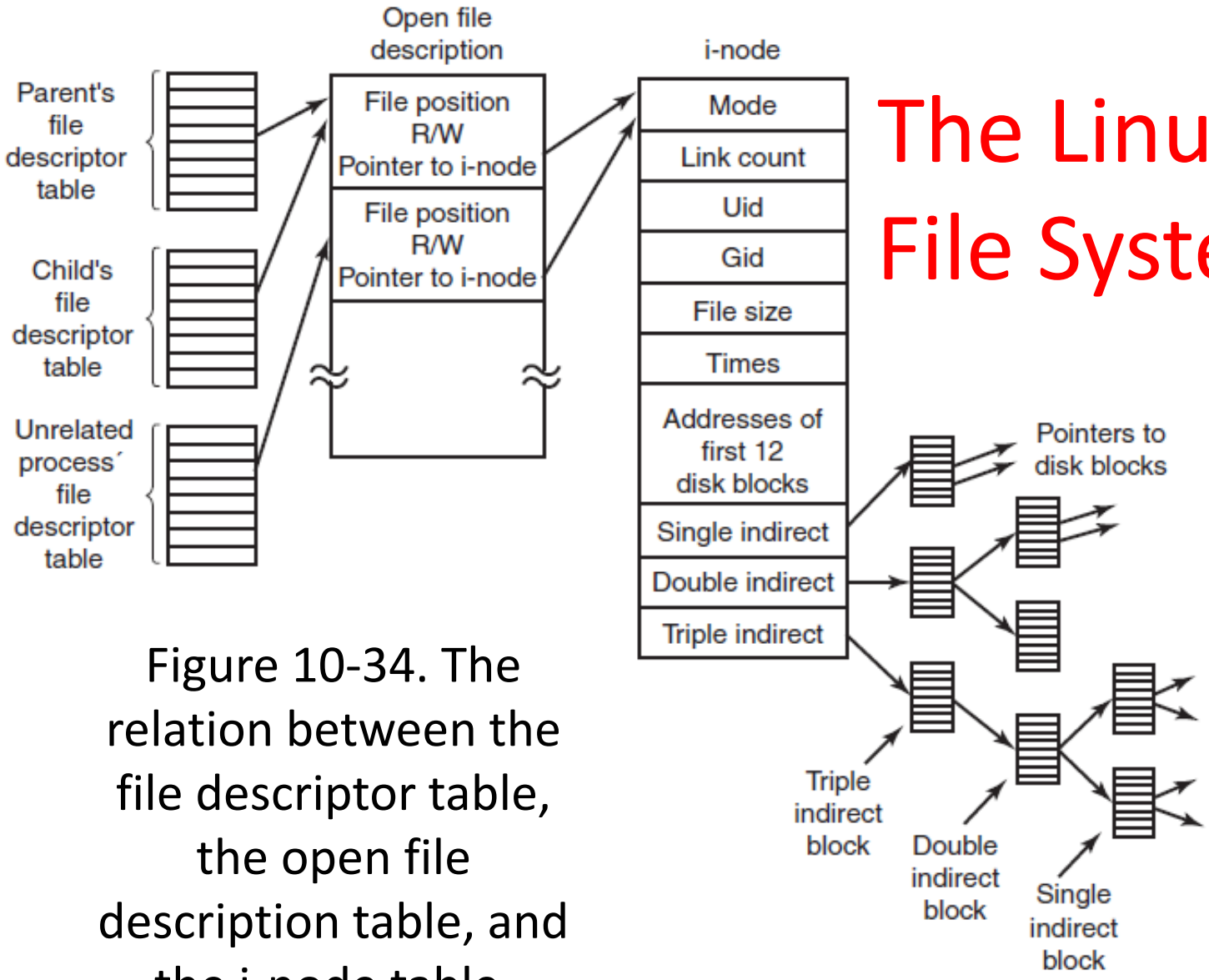

Figure 10-34. The relation between the file descriptor table, the open file description table, and the i-node table.

NFS Architecture

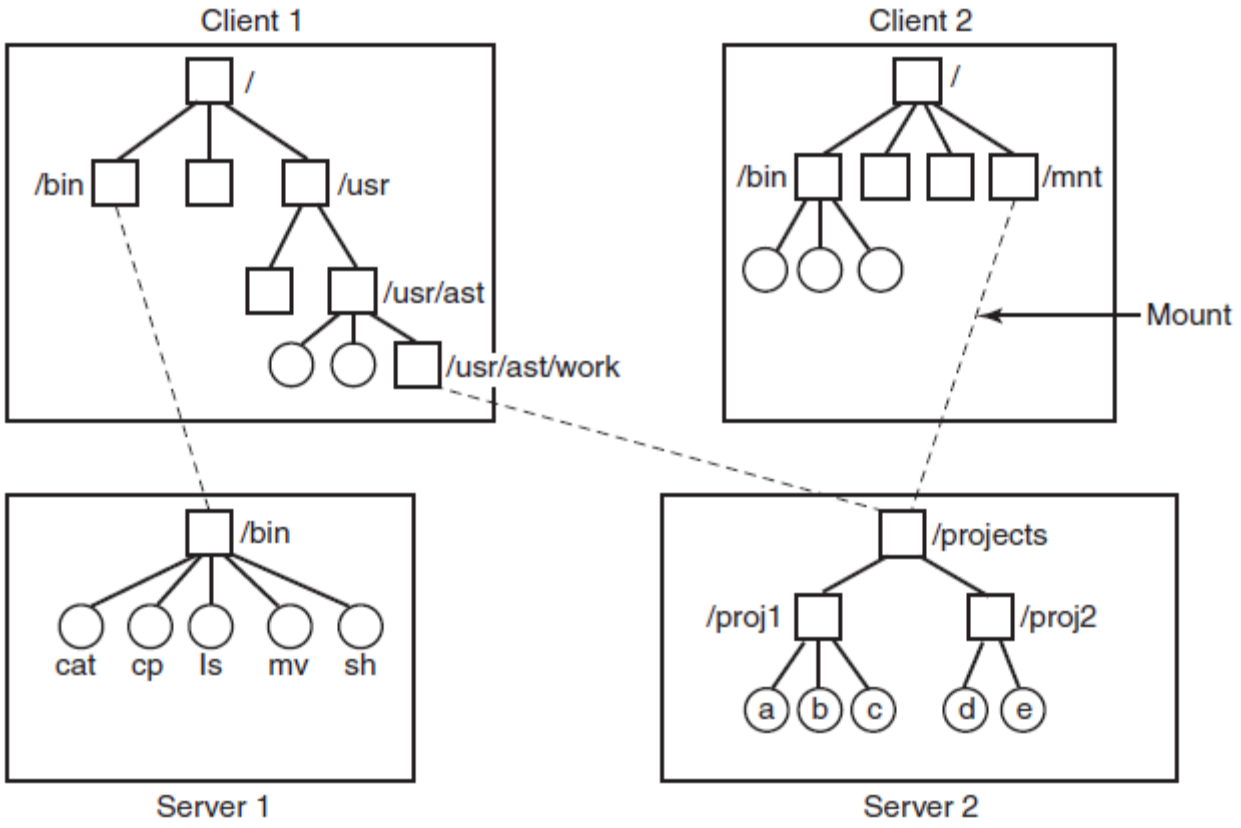

Figure 10-35. Examples of remote mounted file systems. Directories are shown as squares and files are shown as circles.

NFS Implementation

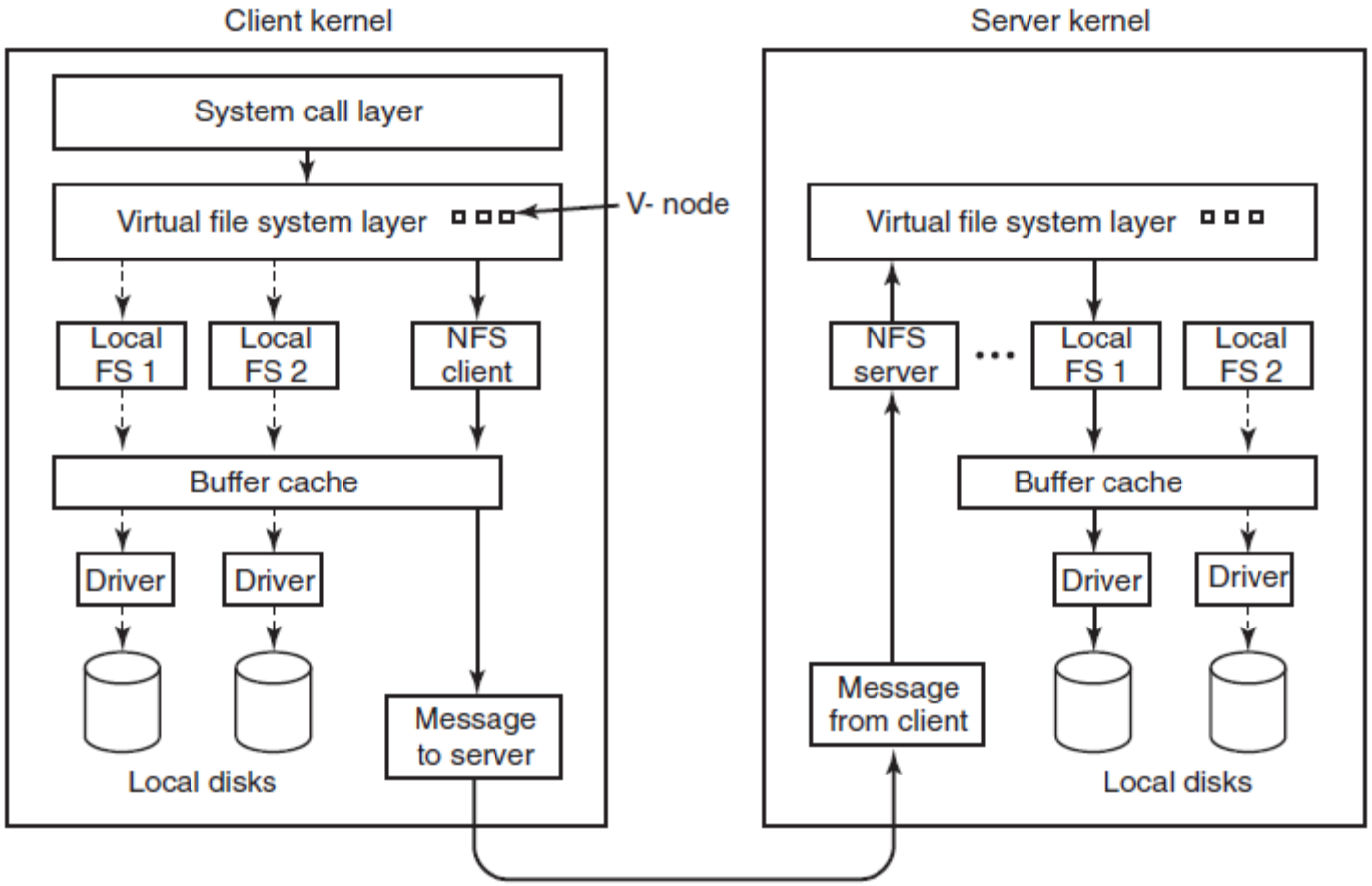

Figure 10-36. The NFS layer structure

Linux Security Fundamental Concepts

Binary	Symbolic	Allowed file accesses
111000000	rwx------	Owner can read, write, and execute
111111000	rwxrwx---	Owner and group can read, write, and execute
110100000	rw-r-----	Owner can read and write; group can read
110100100	rw-r--r--	Owner can read and write; all others can read
111101101	rwxr-xr-x	Owner can do everything, rest can read and execute
000000000	---------	Nobody has any access
000000111	------rwx	Only outsiders have access (strange, but legal)

Figure 10-37. Some example file protection modes.

Security System Calls in Linux

System call	Description
s = chmod(path, mode)	Change a file's protection mode
s = access(path, mode)	Check access using the real UID and GID
uid = getuid( )	Get the real UID
uid = geteuid( )	Get the effective UID
gid = getgid( )	Get the real GID
gid = getegid( )	Get the effective GID
s = chown(path, owner, group)	Change owner and group
s = setuid(uid)	Set the UID
s = setgid(gid)	Set the GID

Figure 10-38. Some system calls relating to security. The return code s is −1 if an error has occurred; *uid* and *gid* are the UID and GID, respectively. The parameters should be self explanatory.

Android – Design Goals

1. Open-source platform for mobile devices
2. Support 3rd party apps with robust, stable API
3. 3rd party apps compete on level playing field
4. Users need not deeply trust 3rd party apps
5. Support mobile user interaction
6. Manage app processes for users
7. Encourage apps to interoperate, collaborate
8. Full general-purpose OS

Android Architecture (1)

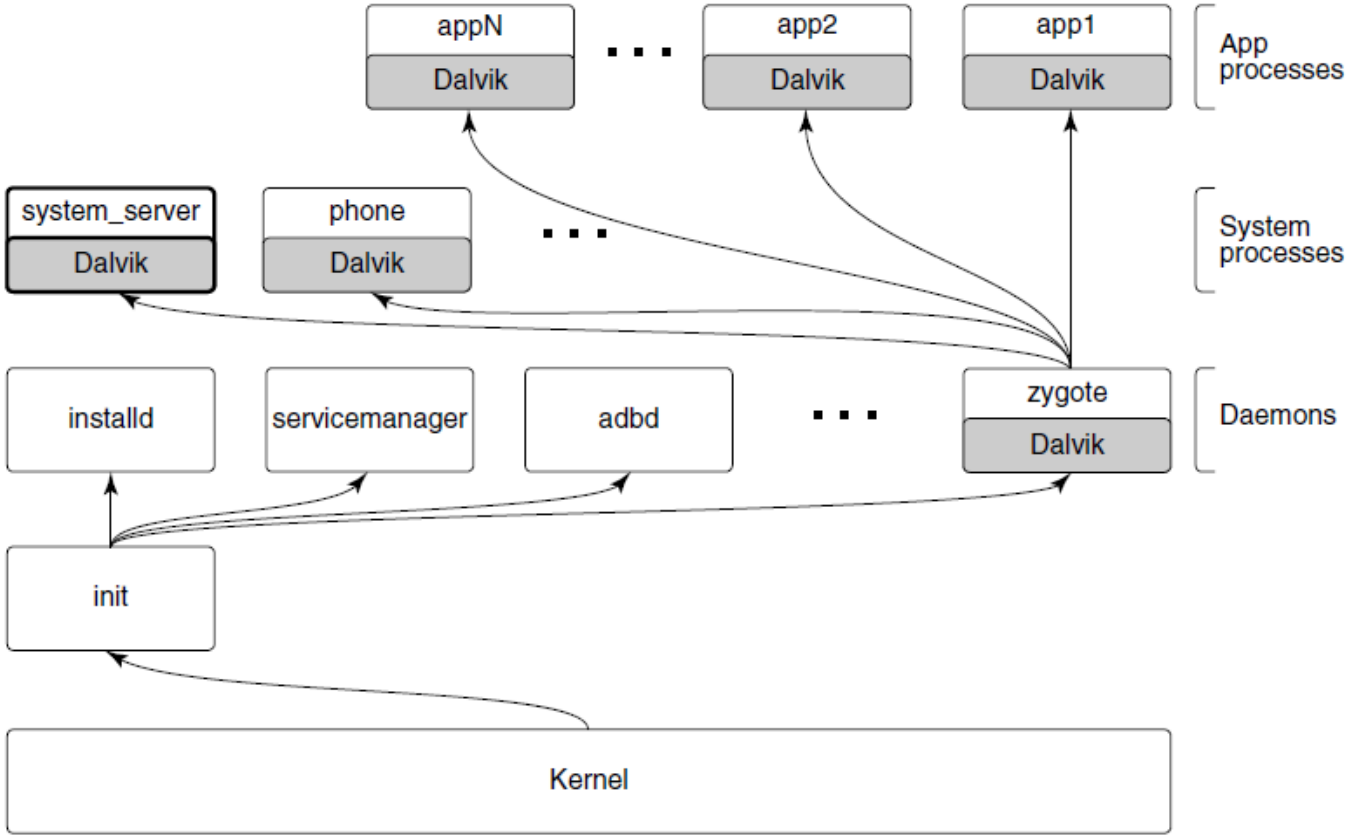

Figure 10-39. Android process hierarchy.

Android Architecture (2)

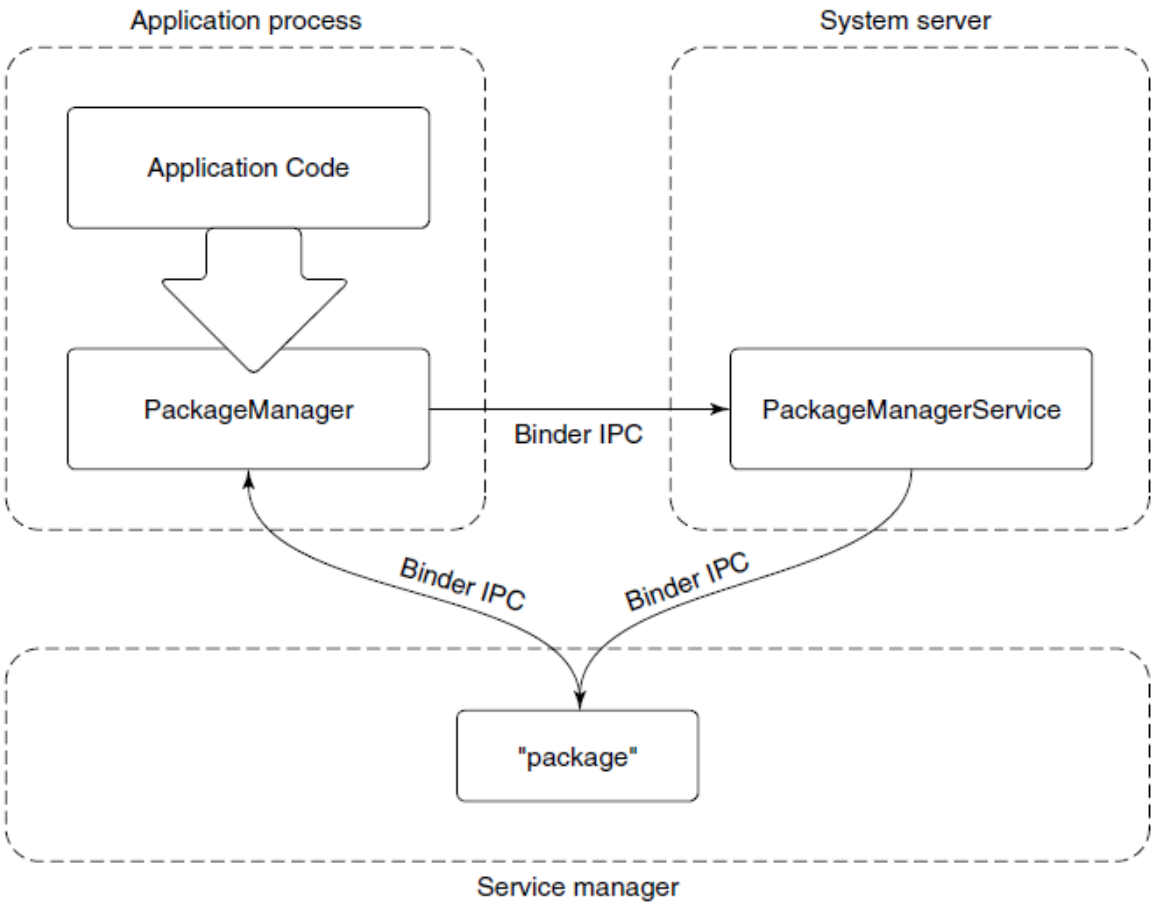

Figure 10-40. Publishing and interacting with system services.

Wake Locks

Contrast idle CPU, true sleep

1. Idle state may use more power than true sleep
2. Idle CPU can wake up at any moment if work becomes available
3. Idle CUP does not tell you that you can turn off other hardware not needed in true sleep

Dalvik

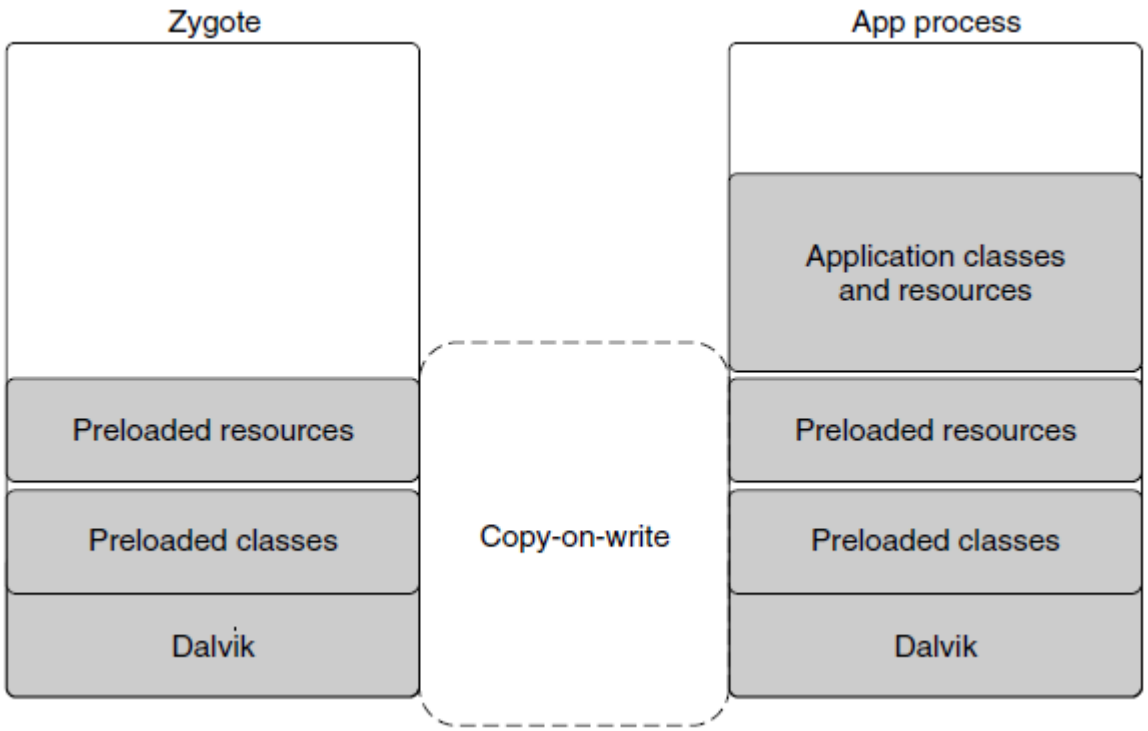

Figure 10-41. Creating a new *Dalvik* process from *zygote*.

Binder IPC

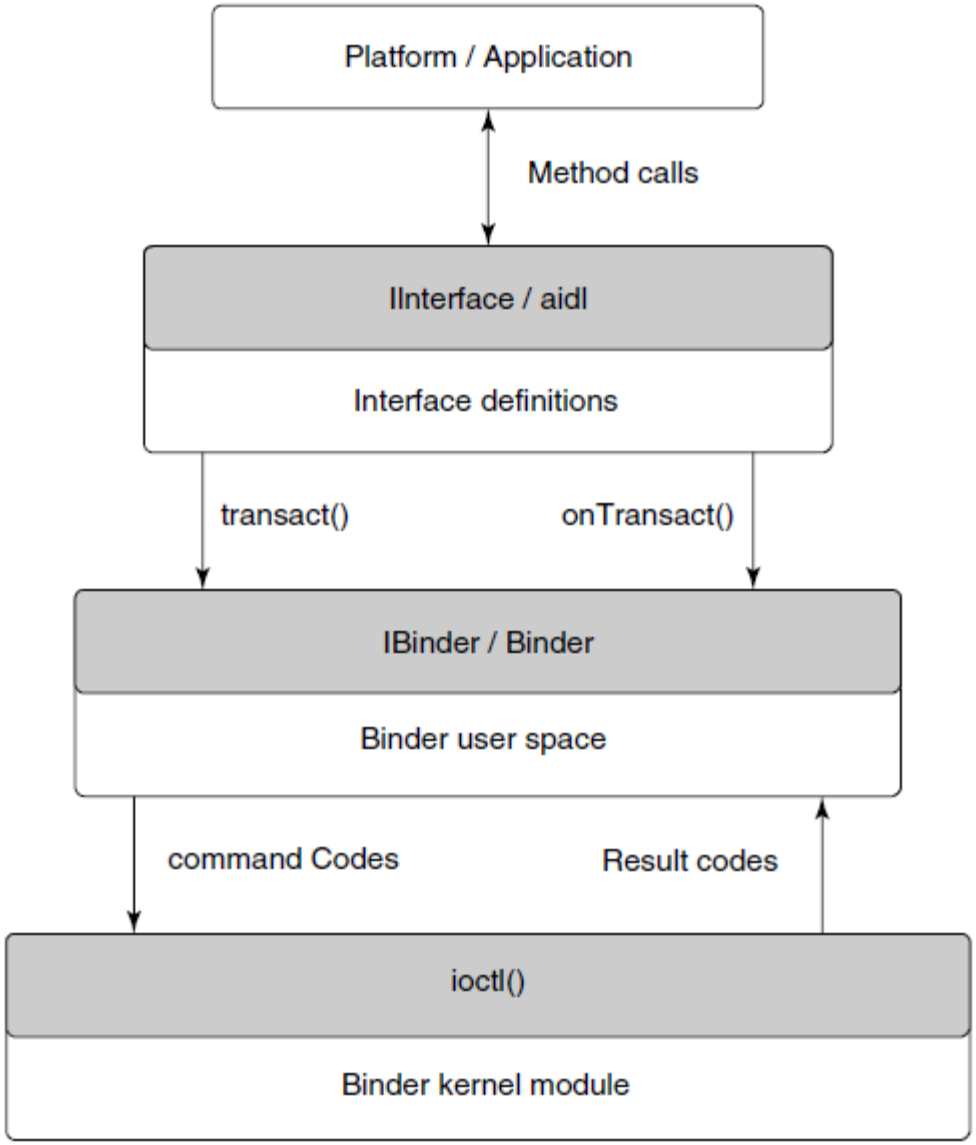

Figure 10-42. *Binder* IPC architecture.

Binder Kernel Module (1)

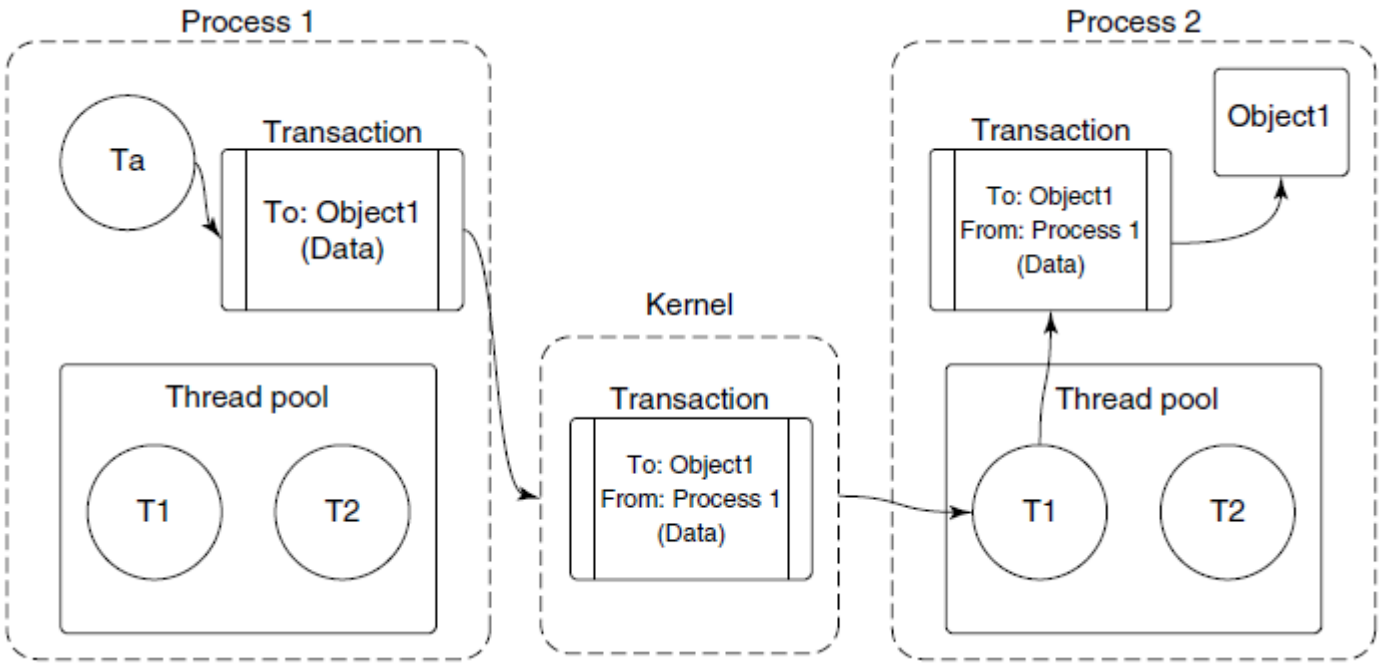

Figure 10-43. Basic Binder IPC transaction

Binder Kernel Module (2)

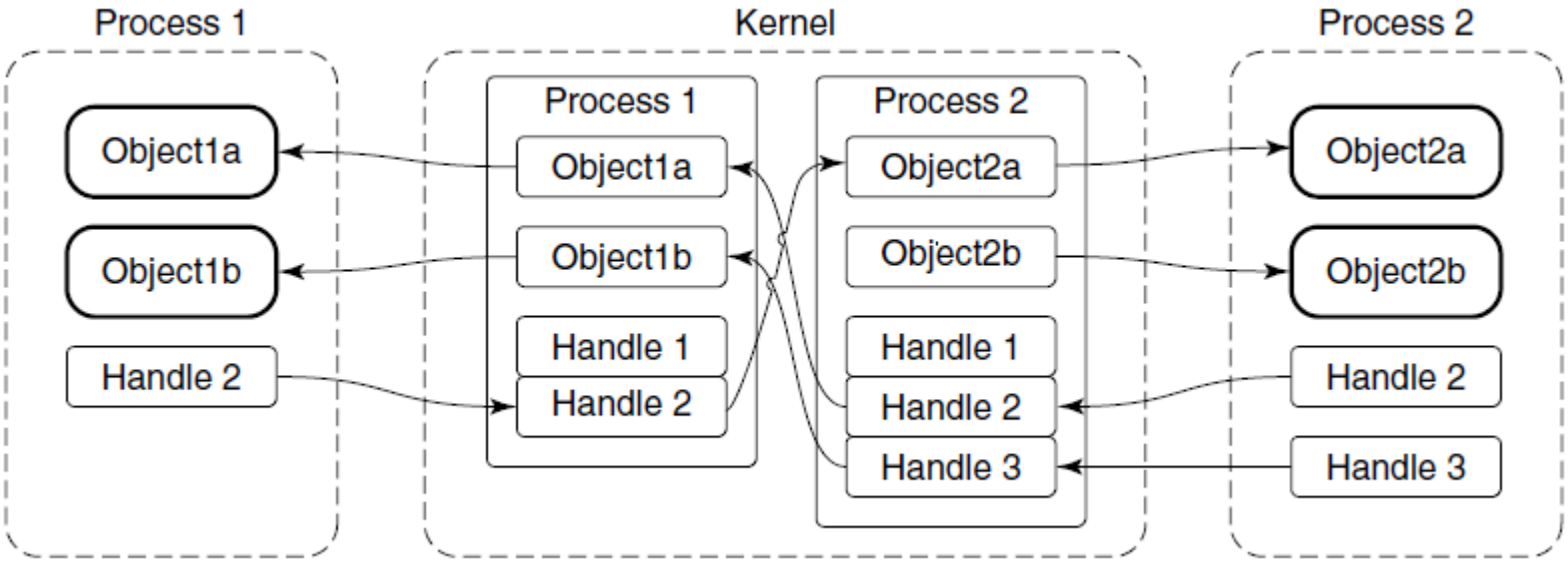

Figure 10-44. Binder cross-process object mapping.

Binder Kernel Module (3)

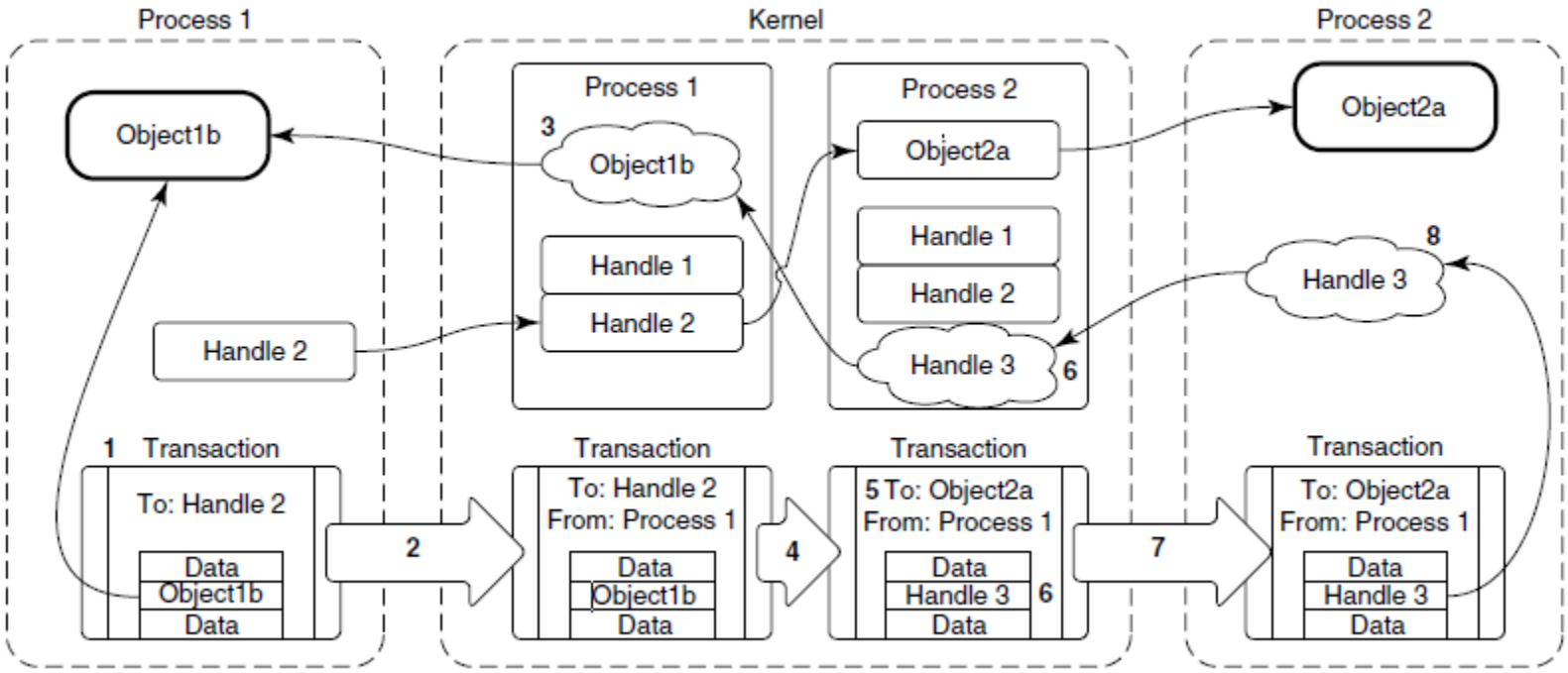

Figure 10-45. Transferring Binder objects between processes.

Binder User-Space API (1)

User-space object-oriented library

1. *IBinder* is an abstract interface for a Binder object
2. *Binder* is a concrete Binder object
3. *Parcel* is a container for reading and writing data that is in a Binder transaction.

Binder User-Space API (2)

Three classes now make it fairly easy to write IPC code:

- Subclass from *Binder*.

1. Implement *onTransact* to decode and execute incoming calls.
2. Implement corresponding code to create a *Parcel* that can be passed to that object's *transact* method.

Binder User-Space API (3)

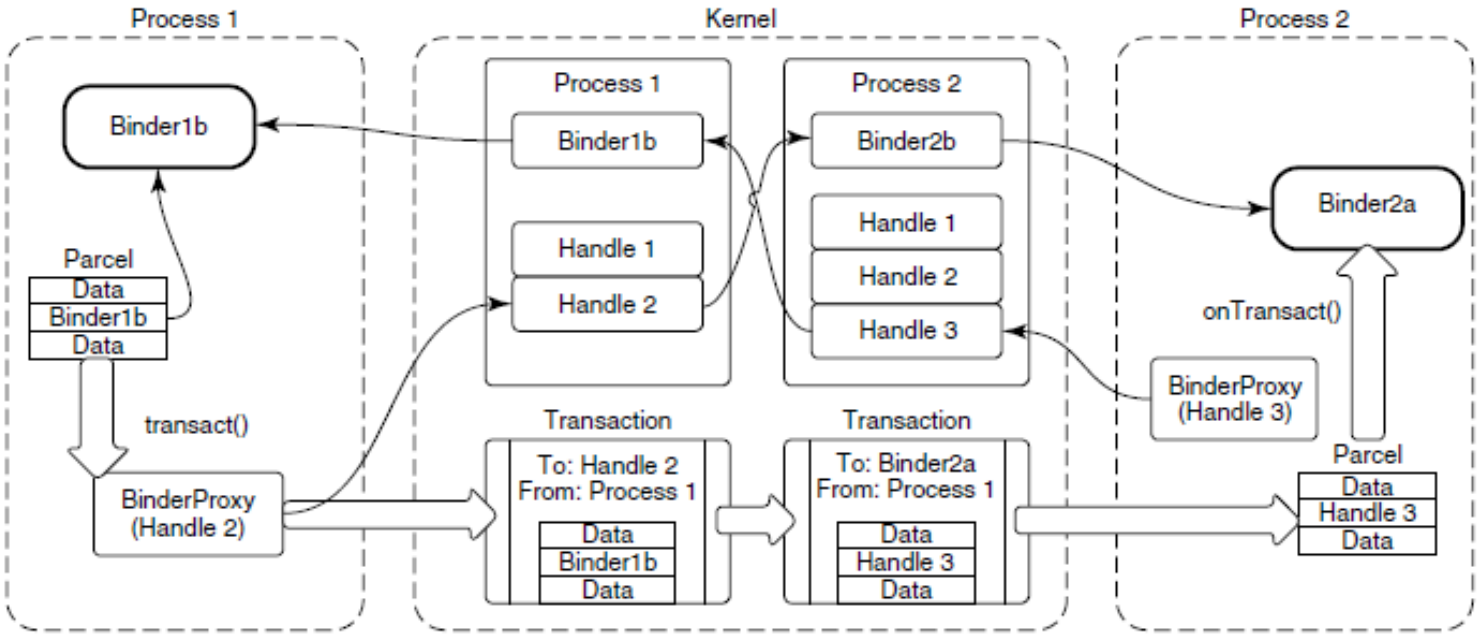

Figure 10-46. Binder user-space API.

Binder Interfaces and AIDL (1)

```
package com.example

interface IExample {
        void print(String msg);
}
```

Figure 10-47. Simple interface described in AIDL.

Binder Interfaces and AIDL (2)

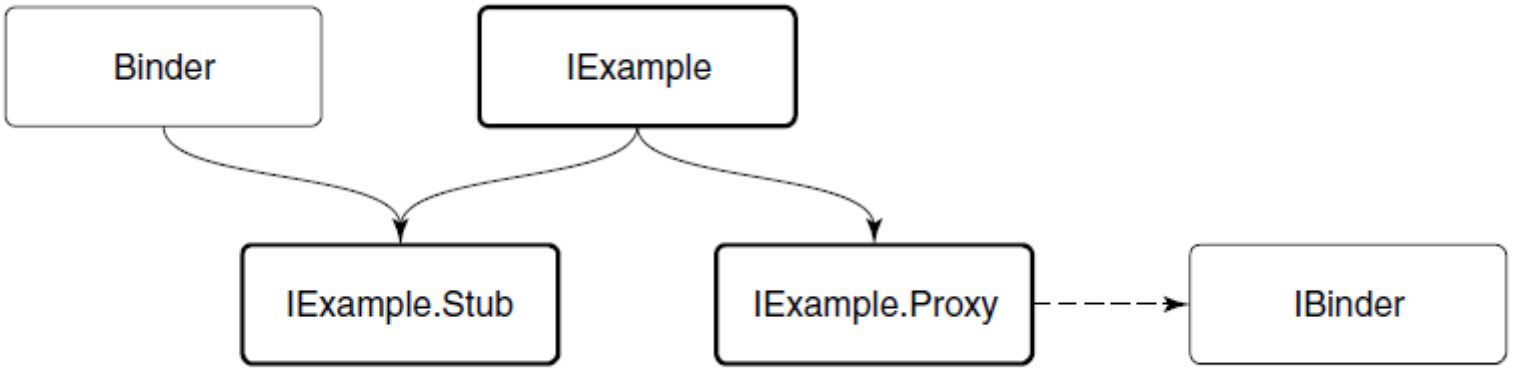

Figure 10-48. Binder interface inheritance hierarchy.

Binder Interfaces and AIDL (3)

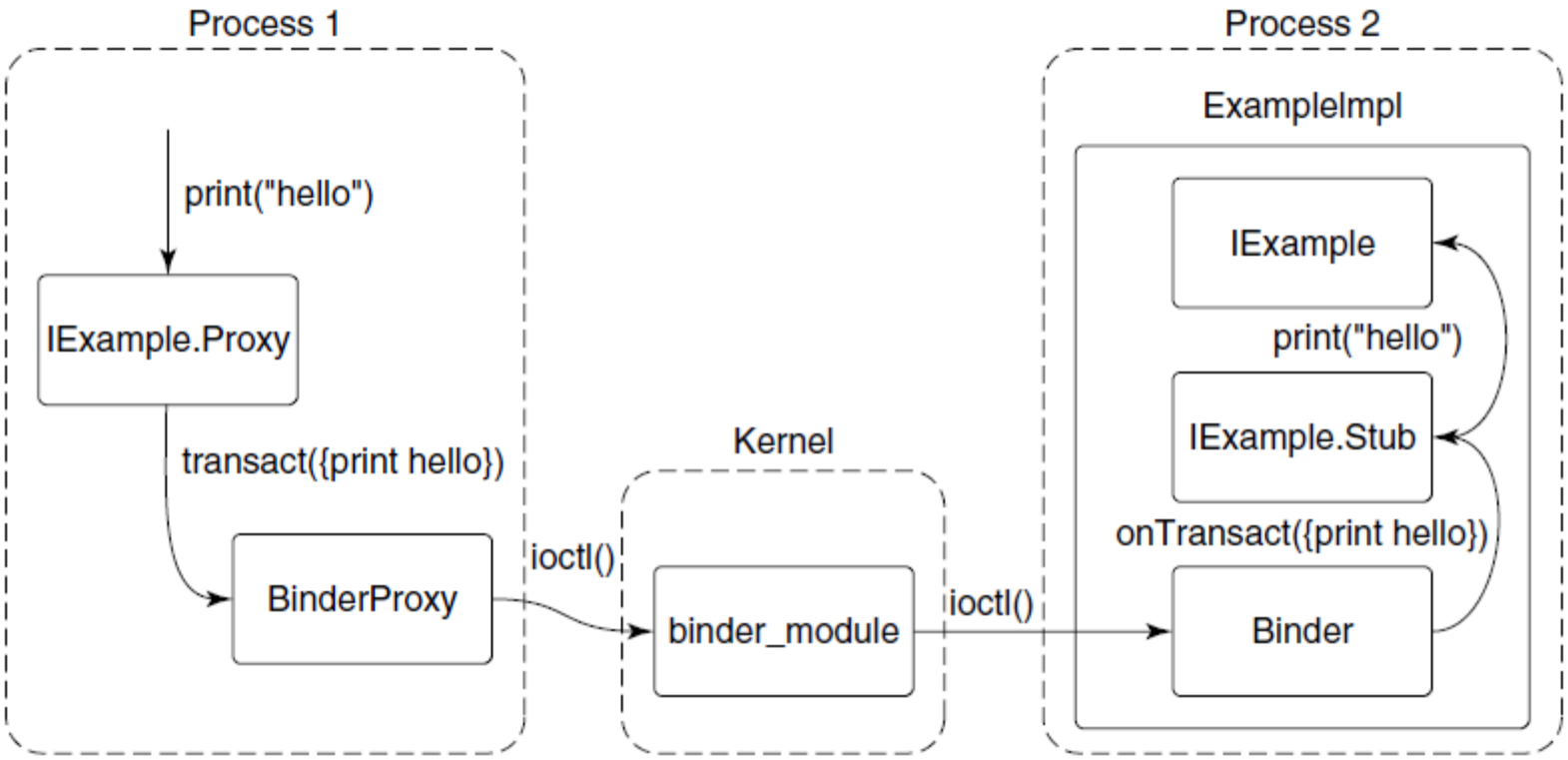

Figure 10-49. Full path of an AIDL-based Binder IPC.

Android Applications (1)

```
package android.os

interface IServiceManager {
        IBinder getService(String name);
        void addService(String name, IBinder binder);
}
```

Figure 10-50. Basic service manager AIDL interface.

Android Applications (2)

Contents of an *apk*

1. A manifest describing what the application is, what it does, and how to run it
2. Resources needed by the application
3. The code itself
4. Signing information, securely identifying the author

Android Applications (3)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.example.email">
   <application>

      <activity android:name="com.example.email.MailMainActivity">
         <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>

      <activity android:name="com.example.email.ComposeActivity">
         <intent-filter>
            <action android:name="android.intent.action.SEND" />
            <category android:name="android.intent.category.DEFAULT" />
            <data android:mimeType="*/*" />
         </intent-filter>
      </activity>

      <service android:name="com.example.email.SyncService">
```

Figure 10-51. Basic structure of AndroidManifest.xml.

Android Applications (4)

```
        </intent-filter>
    </activity>

    <service android:name="com.example.email.SyncService">
    </service>

    <receiver android:name="com.example.email.SyncControlReceiver">
      <intent-filter>
        <action android:name="android.intent.action.DEVICE_STORAGE_LOW" />
      </intent-filter>
      <intent-filter>
        <action android:name="android.intent.action.DEVICE_STORAGE_OKAY" />
      </intent-filter>
    </receiver>

    <provider android:name="com.example.email.EmailProvider"
        android:authorities="com.example.email.provider.email">
    </provider>

  </application>
</manifest>
```

Figure 10-51. Basic structure of AndroidManifest.xml.

Activities (1)

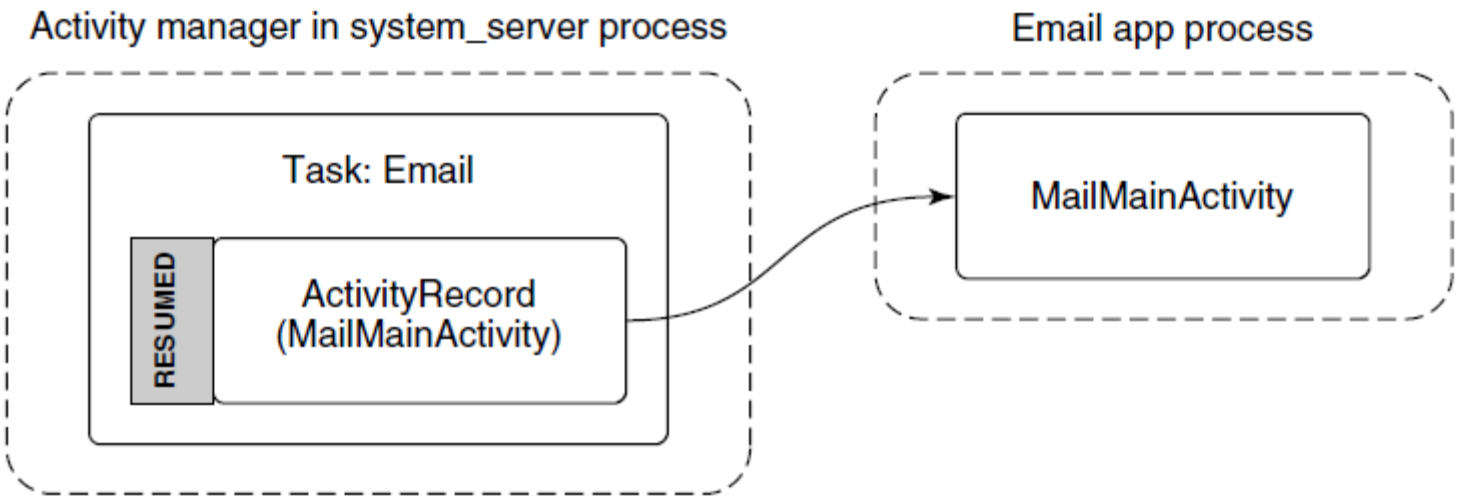

Figure 10-52. Starting an email application's main activity.

Activities (2)

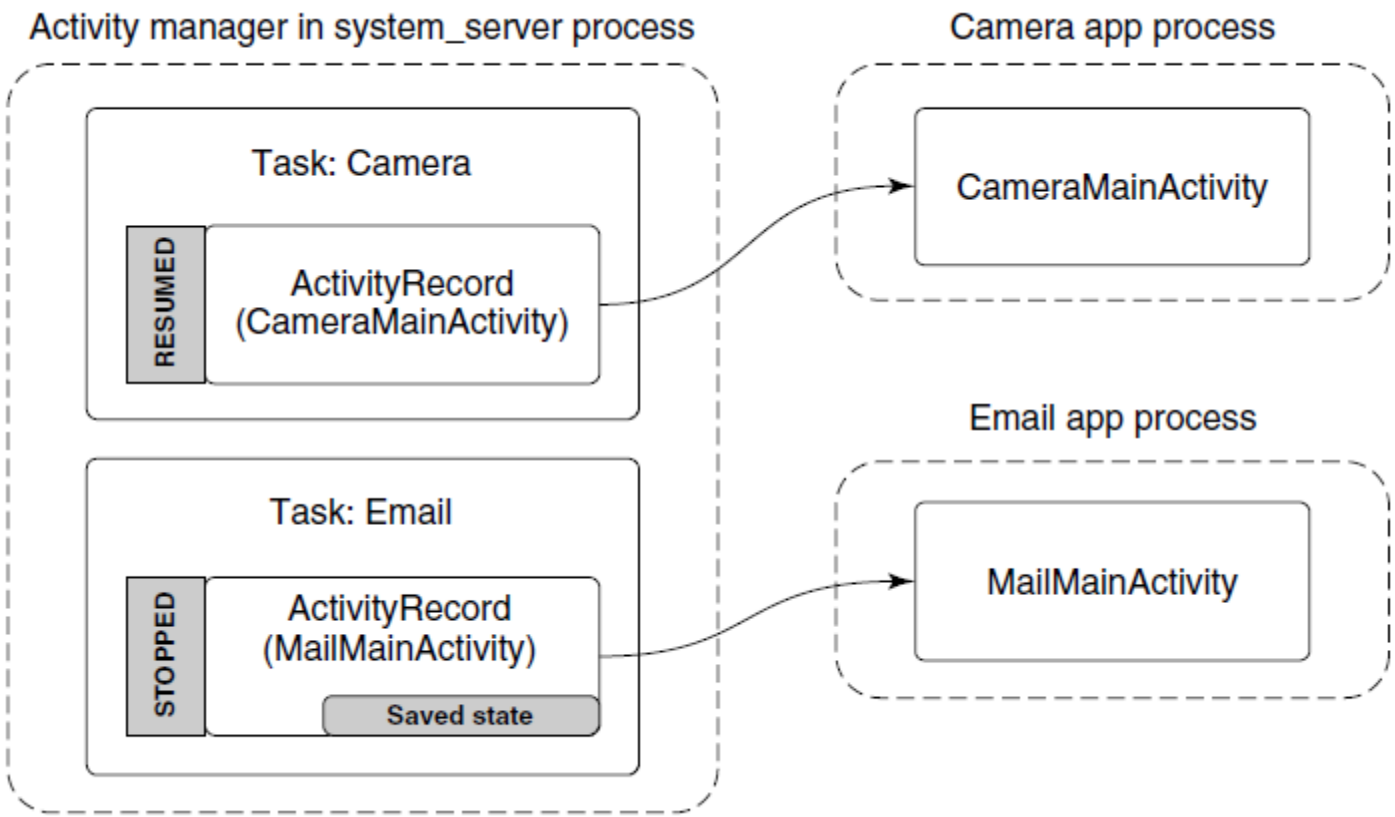

Figure 10-53. Starting the camera application after email.

Activities (3)

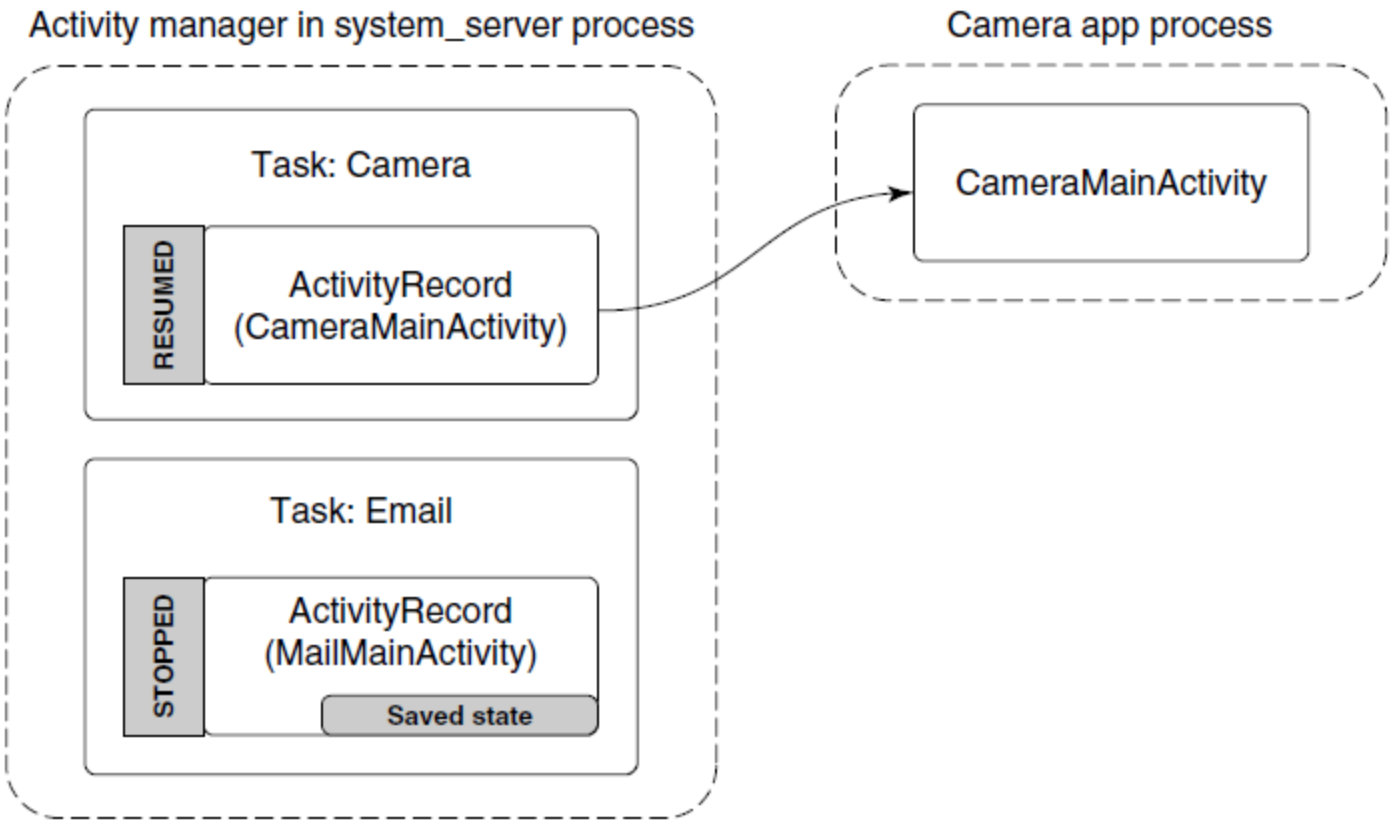

Figure 10-54. Removing the email process to reclaim RAM for the camera.

Activities (4)

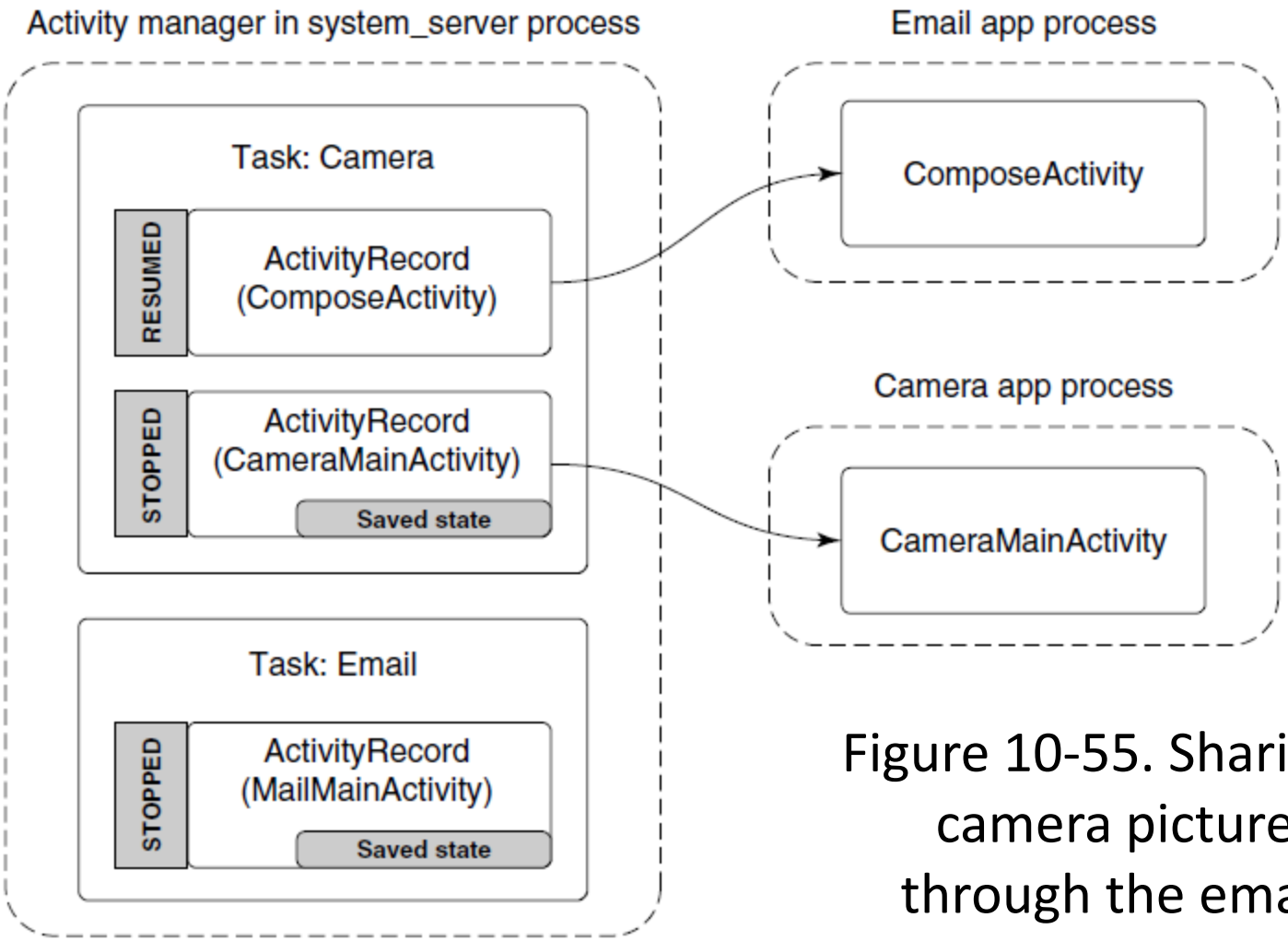

Figure 10-55. Sharing a camera picture through the email application.

Activities (5)

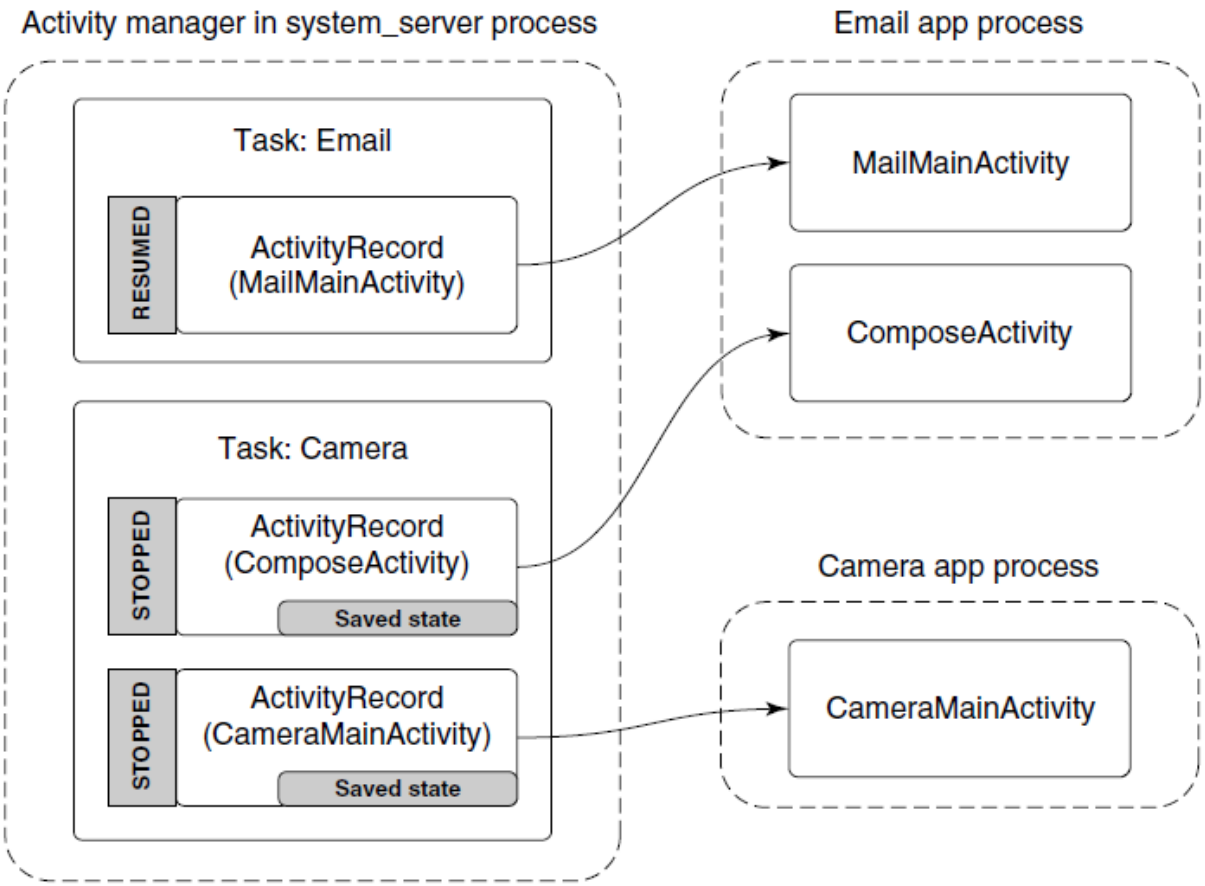

Figure 10-56. Returning to the email application.

Services (1)

A service has two distinct identities:

1. It can be a self-contained long-running background operation.
2. It can serve as a connection point for other applications or the system to perform rich interaction with the application

Services (2)

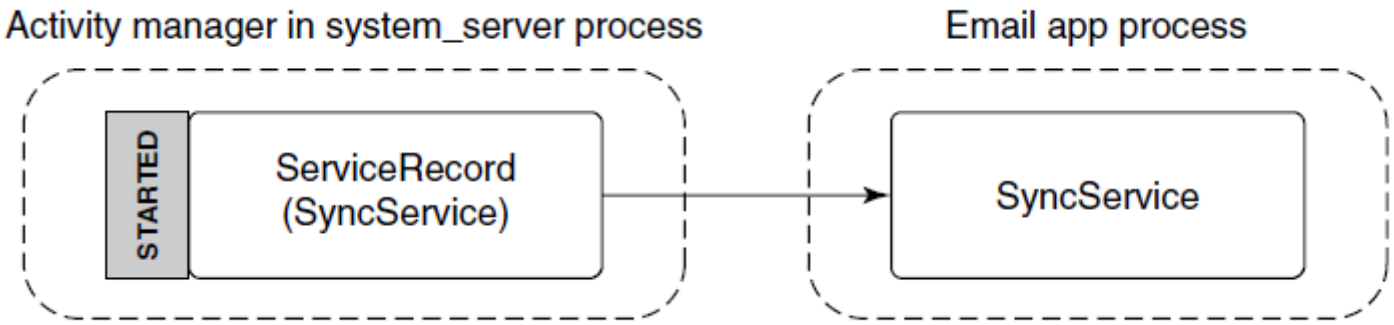

Figure 10-57. Starting an application service.

Services (3)

```
package com.example.email

interface ISyncControl {
        int getSyncInterval();
        void setSyncInterval(int seconds);
}
```

Figure 10-58. Interface for controlling a sync service's sync interval.

Services (4)

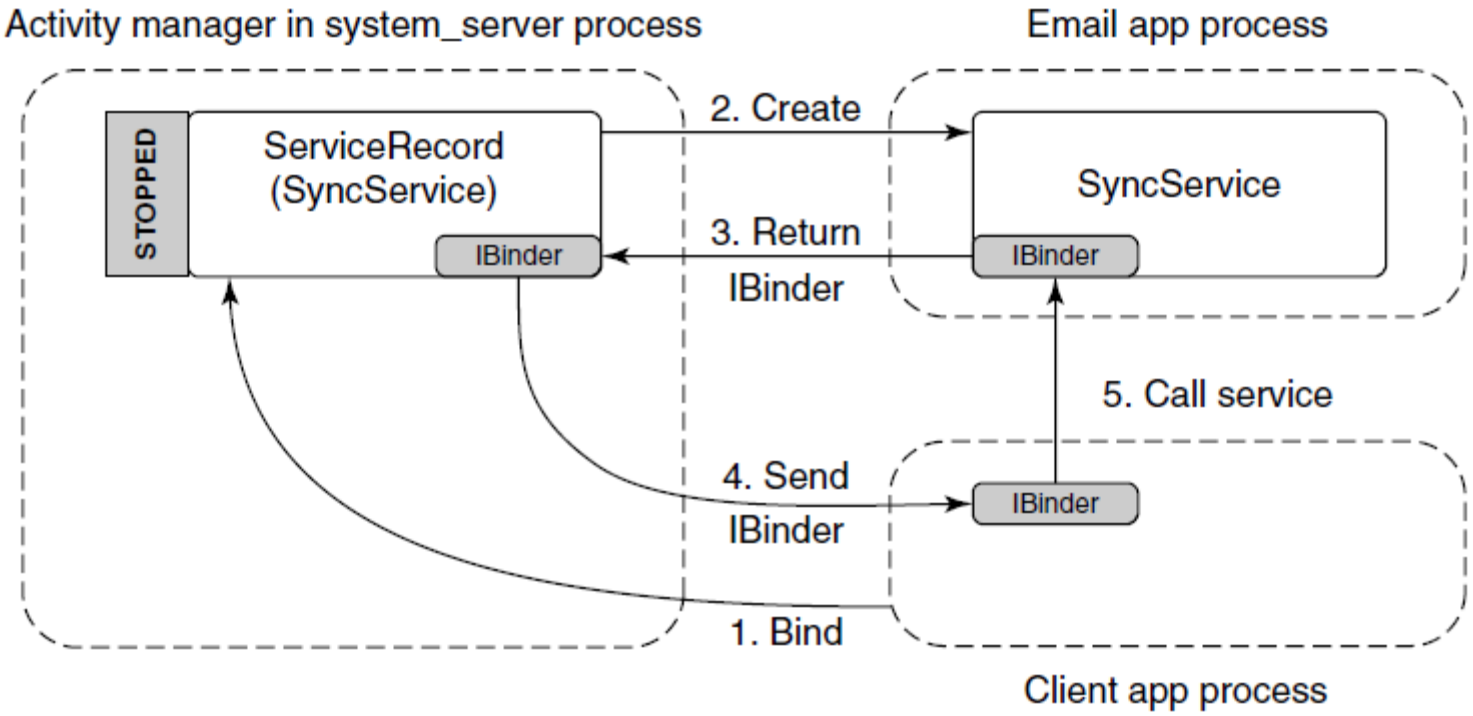

Figure 10-59. Binding to an application service.

Receivers

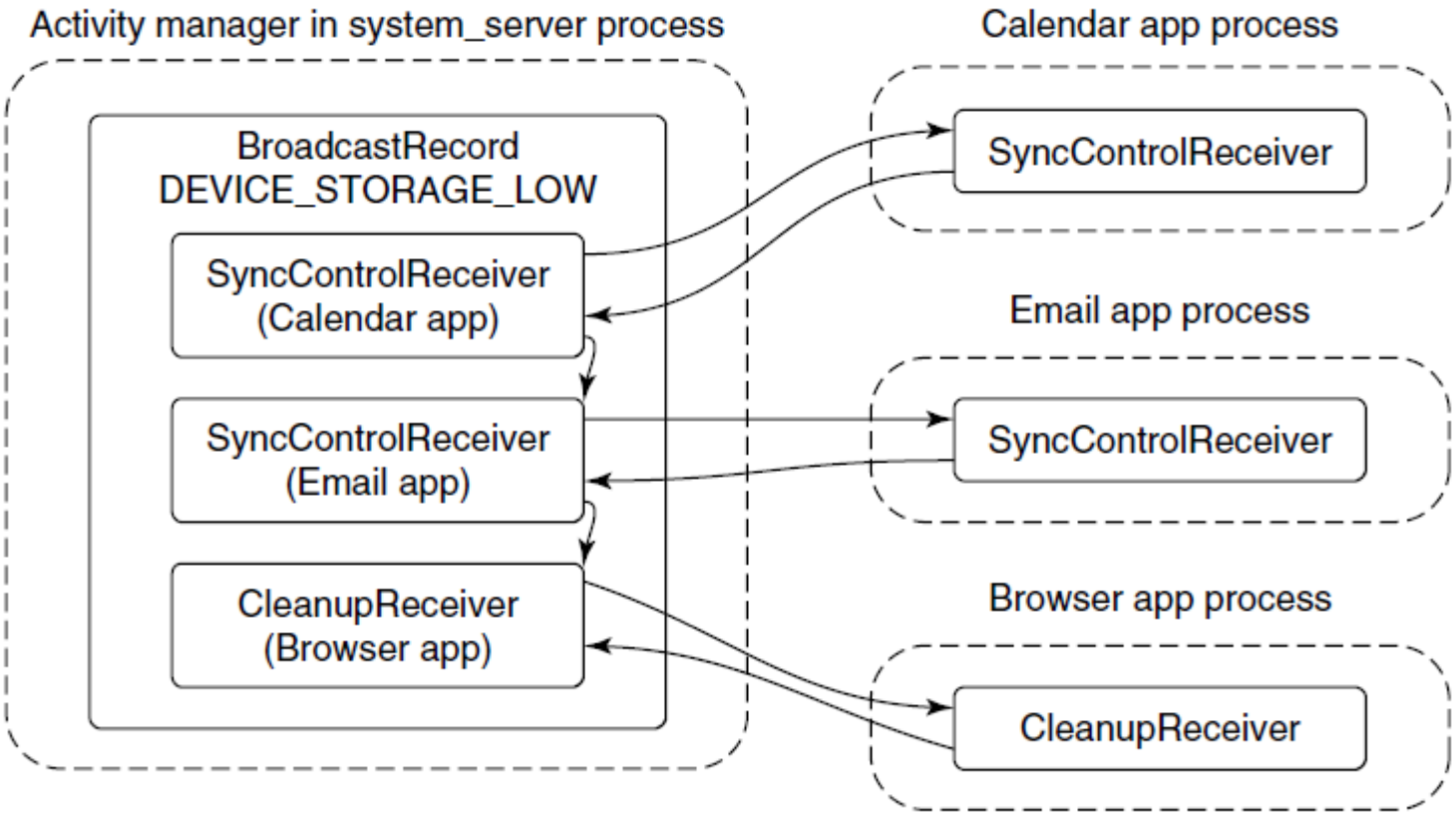

Figure 10-60. Sending a broadcast to application receivers.

Content Providers

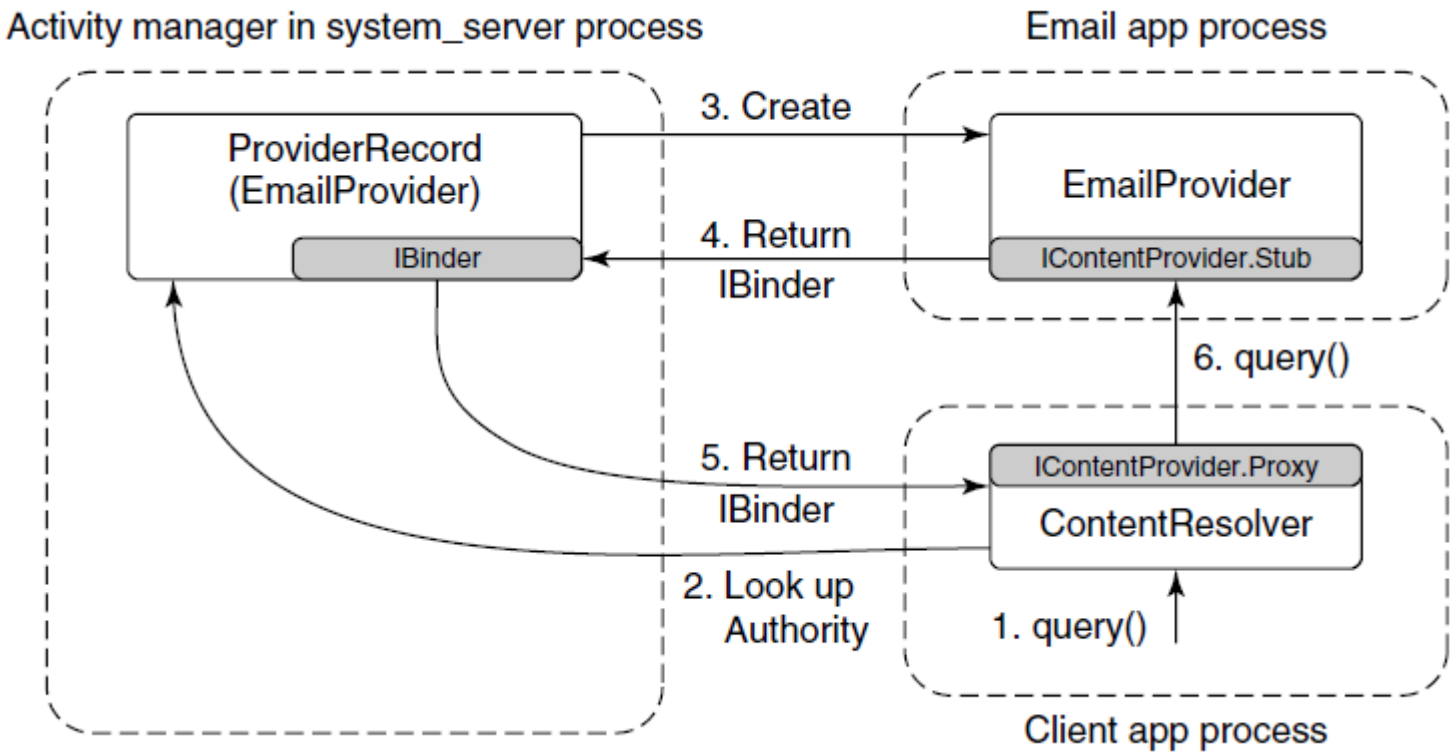

Figure 10-61. Interacting with a content provider.

Application Sandboxes Security (1)

UID	Purpose
0	Root
1000	Core system (system_server process)
1001	Telephony services
1013	Low-level media processes
2000	Command line shell access
10000–19999	Dynamically assigned application UIDs
100000	Start of secondary users

Figure 10-62. Common UID assignments in Android

Security (2)

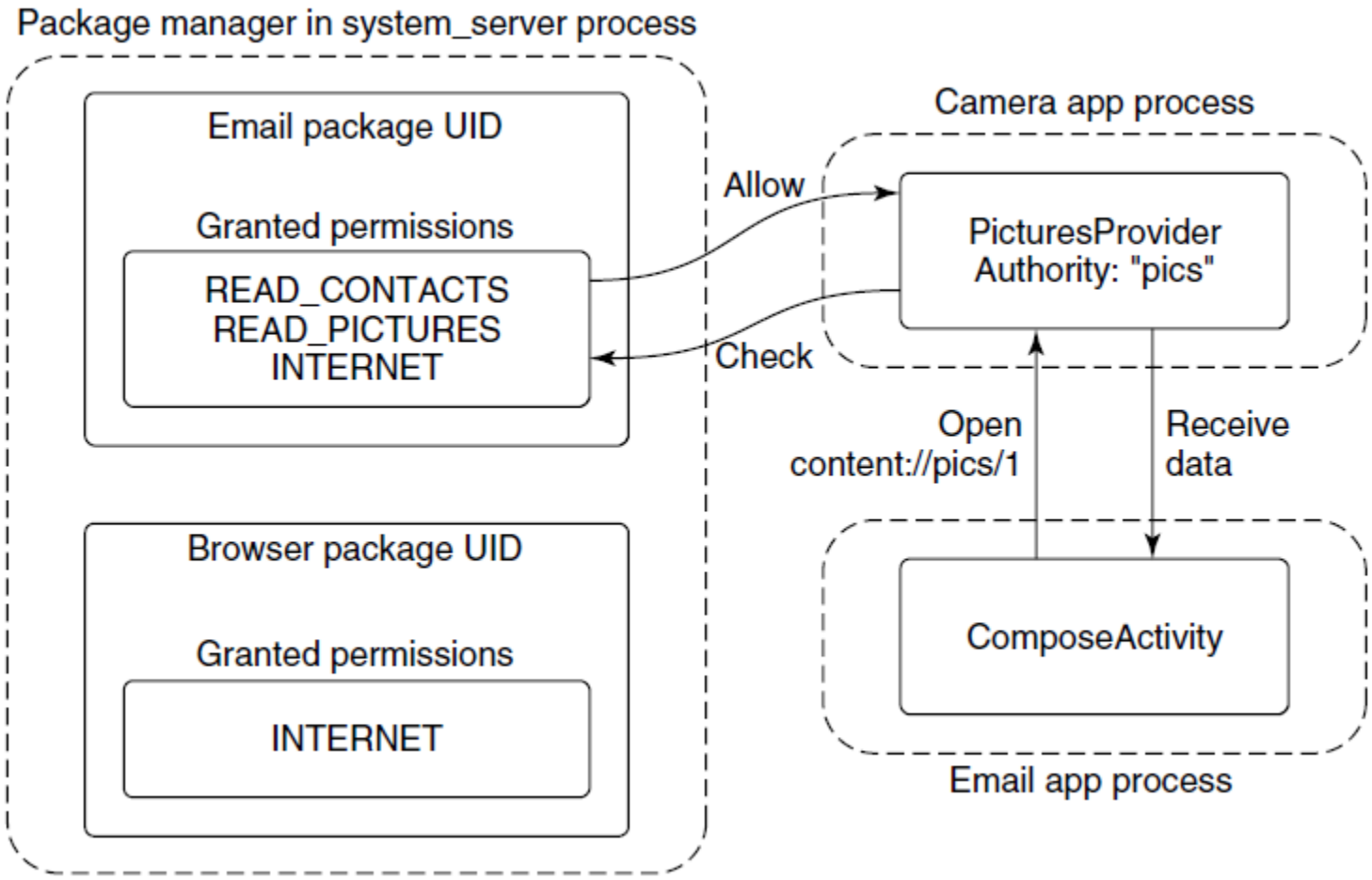

Figure 10-63. Requesting and using a permission.

Security (3)

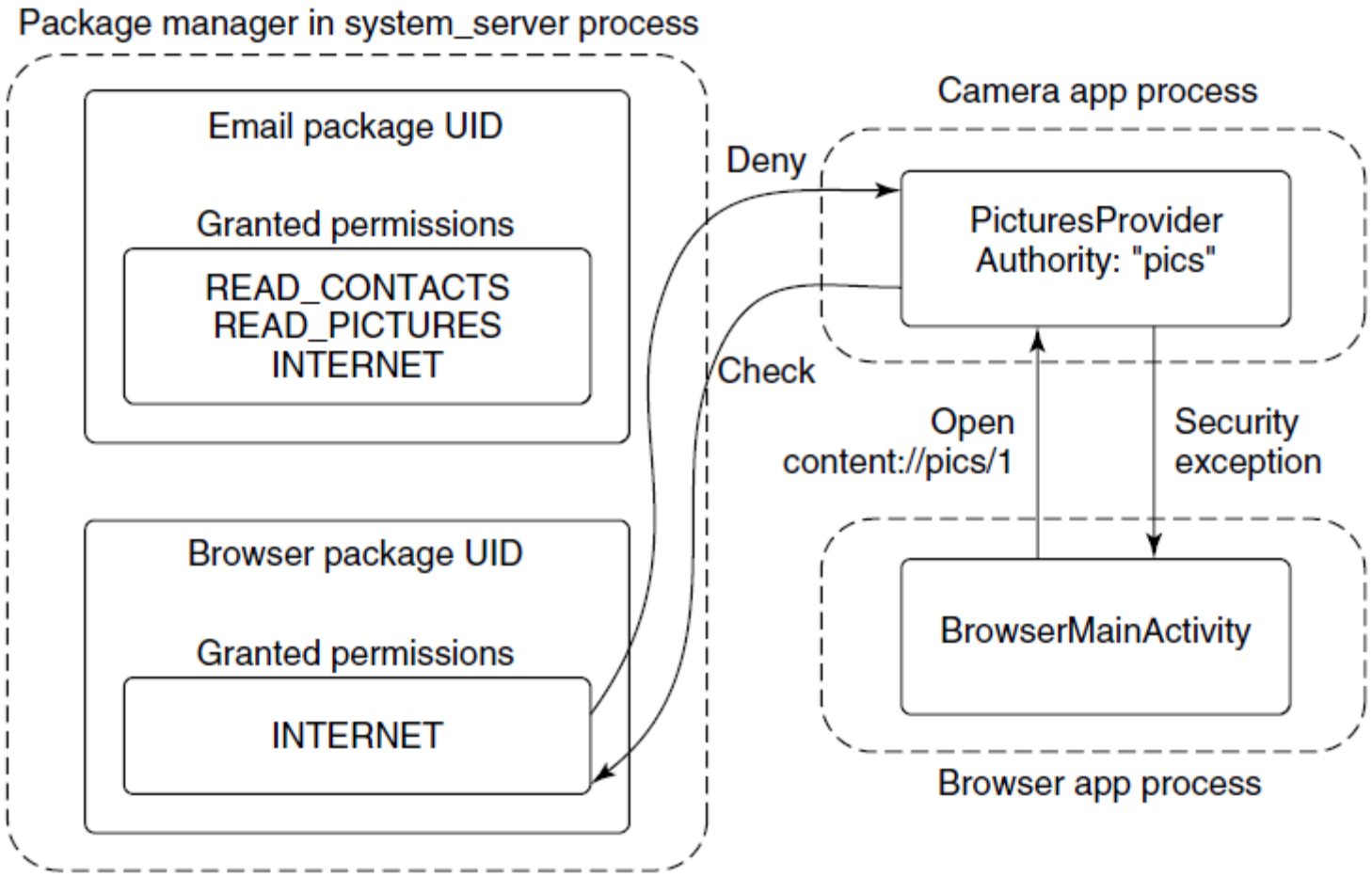

Figure 10-64. Accessing data without a permission.

Security (4)

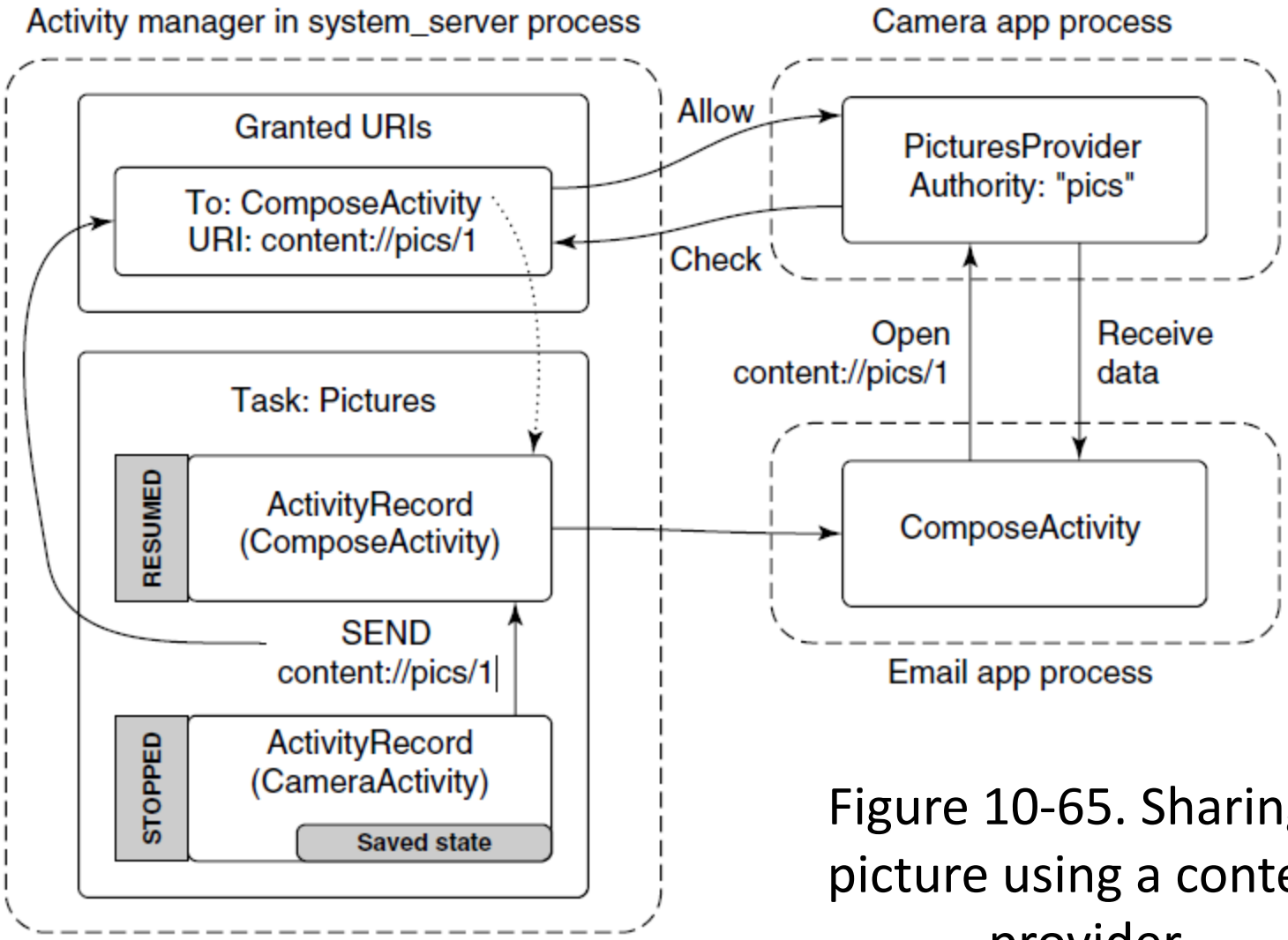

Figure 10-65. Sharing a picture using a content provider.

Security (5)

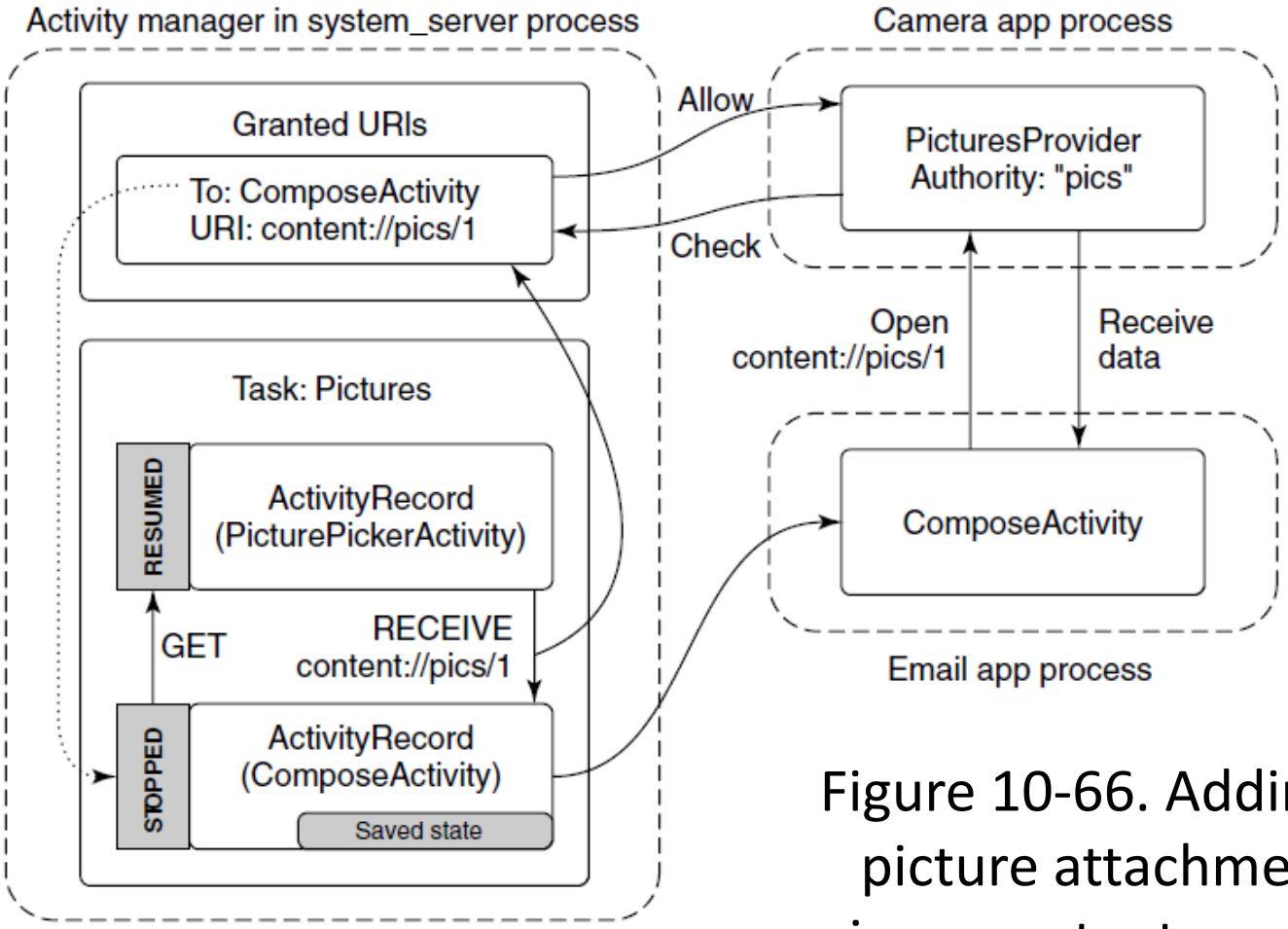

Figure 10-66. Adding a picture attachment using a content provider.

Starting Processes

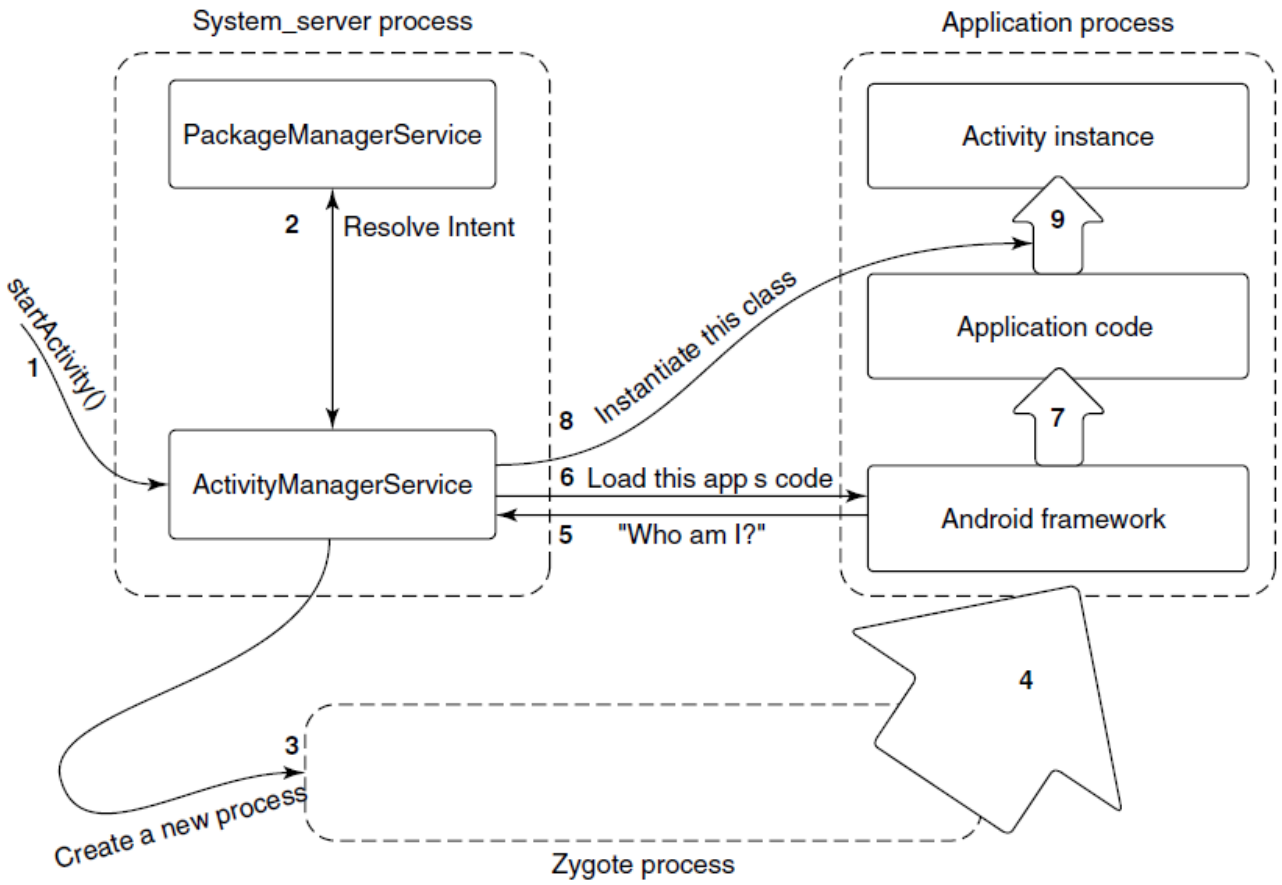

Figure 10-67. Steps in launching a new application process.

Process Lifecycle

Category	Description	oom_adj
SYSTEM	The system and daemon processes	–16
PERSISTENT	Always-running application processes	–12
FOREGROUND	Currently interacting with user	0
VISIBLE	Visible to user	1
PERCEPTIBLE	Something the user is aware of	2
SERVICE	Running background services	3
HOME	The home/launcher process	4
CACHED	Processes not in use	5

Figure 10-68. Process importance categories.

Process Dependencies (1)

Process	State	Importance
system	Core part of operating system	SYSTEM
phone	Always running for telephony stack	PERSISTENT
email	Current foreground application	FOREGROUND
camera	In use by email to load attachment	FOREGROUND
music	Running background service playing music	PERCEPTIBLE
media	In use by music app for accessing user's music	PERCEPTIBLE
download	Downloading a file for the user	SERVICE
launcher	App launcher not current in use	HOME
maps	Previously used mapping application	CACHED

Figure 10-69. Typical state of process importance

Process Dependencies (2)

Process	State	Importance
system	Core part of operating system	SYSTEM
phone	Always running for telephony stack	PERSISTENT
email	Current foreground application	FOREGROUND
music	Running background service playing music	PERCEPTIBLE
media	In-use by music app for accessing user's music	PERCEPTIBLE
download	Downloading a file for the user	SERVICE
launcher	App launcher not current in use	HOME
camera	Previously used by email	CACHED
maps	Previously used mapping application	CACHED+1

Figure 10-70. Process state after email stops using camera

End

Chapter 10