Chapter 1. UNIX System Overview

朱金辉

csjhzhu@scut.edu.cn

华南理工大学软件学院

1. Introduction

- Tour of UNIX from a programmer's perspective
- Brief descriptions and examples of terms and concepts
- Overview of services provided by UNIX

2. Unix Architecture

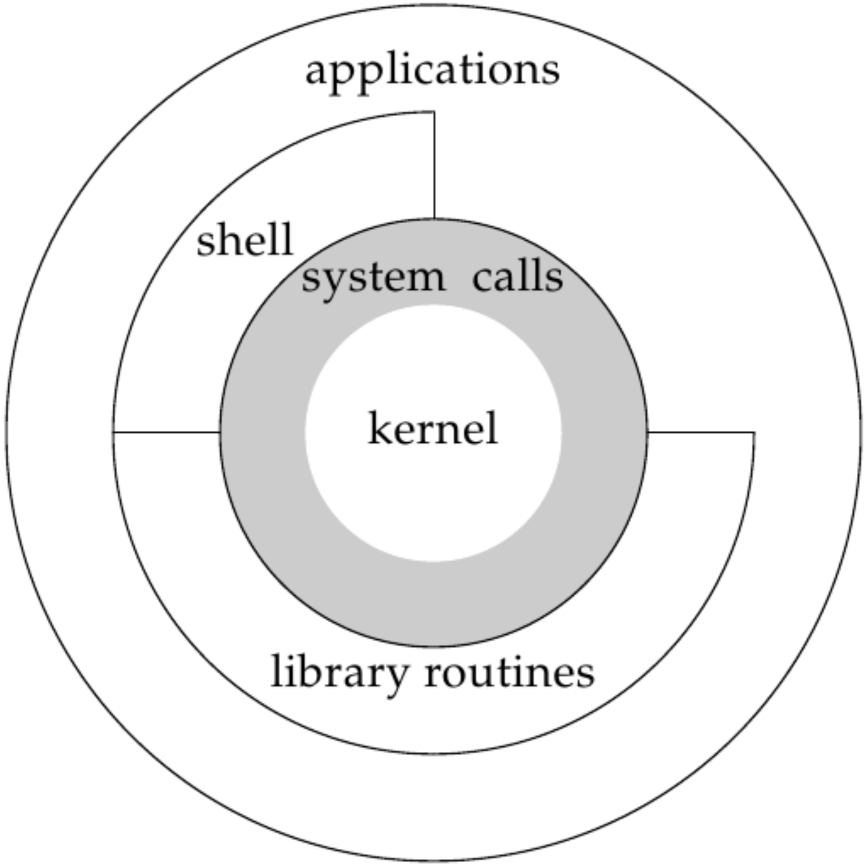


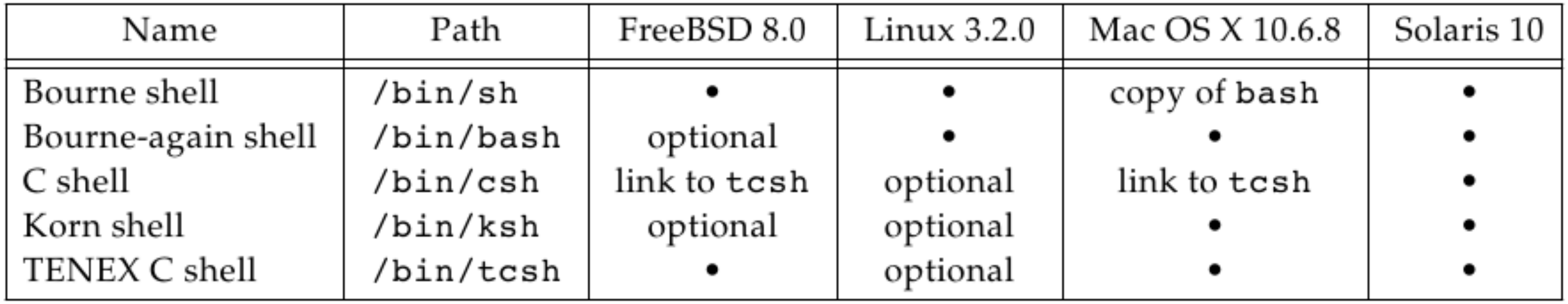
Figure 1.1 Architecture of the UNIX operating system

3. Logging In

- Login name
- Password
- /etc/passwd
  - name,
  - encrypted password or “x” (password is in /etc/shadow),
  - numeric user ID,
  - numeric group ID,
  - real name,
  - home directory,
  - shell program

Logging In

- Shells: a command line interpreter that reads user input and executes commands



Name	Path	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
Bourne shell	`/bin/sh`	•	•	copy of `bash`	•
Bourne-again shell	`/bin/bash`	optional	•	•	•
C shell	`/bin/csh`	link to `tcsh`	optional	link to `tcsh`	•
Korn shell	`/bin/ksh`	optional	optional	•	•
TENEX C shell	`/bin/tcsh`	•	optional	•	•

Figure 1.2 Common shells used on UNIX systems

4. Files and Directories

- Filesystem: hierarchical arrangement of directories and files
- Root directory: /
- File attributes: type, size, owner, permissions, last modification time, …
- stat(), fstat(): return file attribute struct

Files and Directories

- Filename
  - Chars not allowed: (/) and (NULL)
- Two filenames automatically created whenever a new dir is created:
  - . (dot) current directory
  - .. (dot-dot) parent directory
- What is .. in root directory (/)?

Files and Directories

- Pathname
  - A sequence of zero or more filenames, separated by slashes (/), and optionally starting with a slash
- Absolute pathname
- Relative pathname

Program 1.3:
(bare bones implementation of ls command)

```
#include          “apue.h"
#include          <dirent.h>

int main(int argc, char *argv[]){
    DIR           *dp;
    struct dirent  *dirp;

    if (argc != 2)
        err_quit(“usage: ls directory_name");

    if ( (dp = opendir(argv[1])) == NULL)
        err_sys("can't open %s", argv[1]);
    while ( (dirp = readdir(dp)) != NULL)
        printf("%s\n", dirp->d_name);
    closedir(dp);
    exit(0);
}
```

Program 1.3

- Edit and save in myls.c

```
cc myls.c     (output: a.out)
./a.out /dev  (output: . .. ………)
$ ./a.out /etc/ssl/private
can’t open /etc/ssl/private: Permission denied
a.out /dev/tty
can’t open /dev/tty: Not a directory
```

5. Input and Output

- File Descriptors
  - small nonnegative integers that kernel uses to identify files being accessed by a process
- Standard Input
- Standard Output
- Standard Error

Input and Output

- ls
  - stdin, stdout, stderr: ☾ terminal
- ls > myfile.abc
  - Stdout: myfile.abc
- How to redirect **stderr** to a file?
- How to redirect **stdin** from a file?
- Unbuffered I/O
  - open(), read(), write(), lseek(), close()

Program 1.4: stdin stdout

```
#include     "apue.h"
#define      BUFFSIZE    4096
int main(void) {
  int        n;
  char       buf[BUFFSIZE];
  while ( (n = read(STDIN_FILENO, buf, BUFFSIZE)) > 0)
      if (write(STDOUT_FILENO, buf, n) != n)
            err_sys("write error");
  if (n < 0)
      err_sys("read error");
  exit(0);
}
```

Standard I/O

- A buffered interface
- No need to worry about BUFFSIZE
- Deal with “lines of input”
  - fgets() reads an entire line
  - read() reads a specified # of bytes
- printf() (#include <stdio.h>)

Program 1.5:
stdin stdout using standard I/O

```
#include "apue.h"
int main(void) {
  int        c;
  while ( (c = getc(stdin)) != EOF)
     if (putc(c, stdout) == EOF)
          err_sys("output error");
  if (ferror(stdin))
     err_sys("input error");
  exit(0);
}
```

6. Programs and Processes

- Program: an executable file in disk
- Process: an executing instance of a program
- Process also called "task" by some OS
- Unique nonnegative integer identifier for each process (pid)

Program 1.6: process ID

```
#include "apue.h"

int main(void) {
  printf("hello world from process
  ID %d\n", getpid());
  exit(0);
}
```

Process Control

- Three functions
  - fork()
  - exec(): 6 variants
  - waitpid()

Program 1.7: exec stdin cmds

```
#include "apue.h"
#include <sys/wait.h>
int
main(void)
{
    char buf[MAXLINE];    /* from apue.h */
    pid_tpid;
    int              status;

    printf("%% ");           /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
         if (buf[strlen(buf) - 1] == '\n')
                 buf[strlen(buf) - 1] = 0; /* replace newline with null */

         if ((pid = fork()) < 0) {
                 err_sys("fork error");
         } else if (pid == 0) {              /* child */
                 execlp(buf, buf, (char *)0);
                 err_ret("couldn't execute: %s", buf);
                 exit(127);
         }

         /* parent */
         if ((pid = waitpid(pid, &status, 0)) < 0)
                 err_sys("waitpid error");
         printf("%% ");
    }
    exit(0);
}
```

Threads

- All threads within a process share the same address space, file descriptors, stacks, and process related attributes.
- Each thread executes on its own stack.
- Threads are identified by IDs.

7. Error Handling

- Negative return value when error occurs
- #include <errno.h>
- errno variable
  - never cleared if error does not occur
  - never set to 0 by any function

Error Handling (contd)

- 2 functions for printing error messages:

```
#include <string.h>
char *strerror(int errnum);
```

```
#include <stdio.h>
void perror(const char *msg);
```

- strerror() returns a string
- perror() outputs "msg: <error_msg>"

Program 1.8: use of error func

```
#include        <errno.h>
#include        "apue.h"

int main(int argc, char *argv[]) {
    fprintf(stderr, "EACCES: %s\n", strerror(EACCES));

    errno = ENOENT;
    perror(argv[0]);

    exit(0);
}
```

Program 1.8: results

```
$ a.out
EACCES: Permission denied
a.out: No such file or directory
```

- (argv[0] passed as arg to perror())

8. User Identification

- User ID: numeric identifier of a user
- Group ID: numeric identifier of a group

```
#include  "apue.h"

int main(void) {
     printf("uid = %d, gid = %d\n",
          getuid(), getgid());
     exit(0);
}
```

9. Signals

- A technique to notify a process that some condition has occurred
- E.g.: divide by zero ☾ SIGFPE
- Process response to a signal
  - Ignore the signal, OR
  - Let the default action occur, OR
  - Provide a function to handle the signal.

Signals Example: shell2.c

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

static void            sig_int(int);          /* our signal-catching function */
int main(void) {
    char   buf[MAXLINE];
    pid_t  pid;
    int    status;
    if (signal(SIGINT, sig_int) == SIG_ERR)
           err_sys("signal error");
    printf("%% "); /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
           buf[strlen(buf) - 1] = 0;/* replace newline with null*/
           if ( (pid = fork()) < 0)
                     err_sys("fork error");
```

Signals (contd)

```
        else if (pid == 0) {                    /* child */
                execlp(buf, buf, (char *) 0);
                err_ret("couldn't execute: %s", buf);
                exit(127);
        }
        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)
                err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
void sig_int(int signo) {
    printf("interrupt\n%% ");
}
```

10. Time Values

- Two different time values
- **Calendar time**: #seconds since the Epoch, which is 00:00:00 Jan 1, 1970, Coordinated Universal Time (UTC).
- **Process time**: measures CPU resources used by a process, in clock ticks, which is 50, 60, or 100 ticks per second.

Time Values (contd)

- Execution time of a process has 3 values:
- **clock time**: total amount of time from process start to finish
- **user CPU time**: CPU time due to user instructions in a process
- **system CPU time**: CPU time due to kernel activities on behalf of the process

Time Values (contd)

- To measure process execution time, use the “time” command as follows:

```
time ls > /dev/null
real     0m19.81s
user     0m0.43s
sys      0m4.53s
```

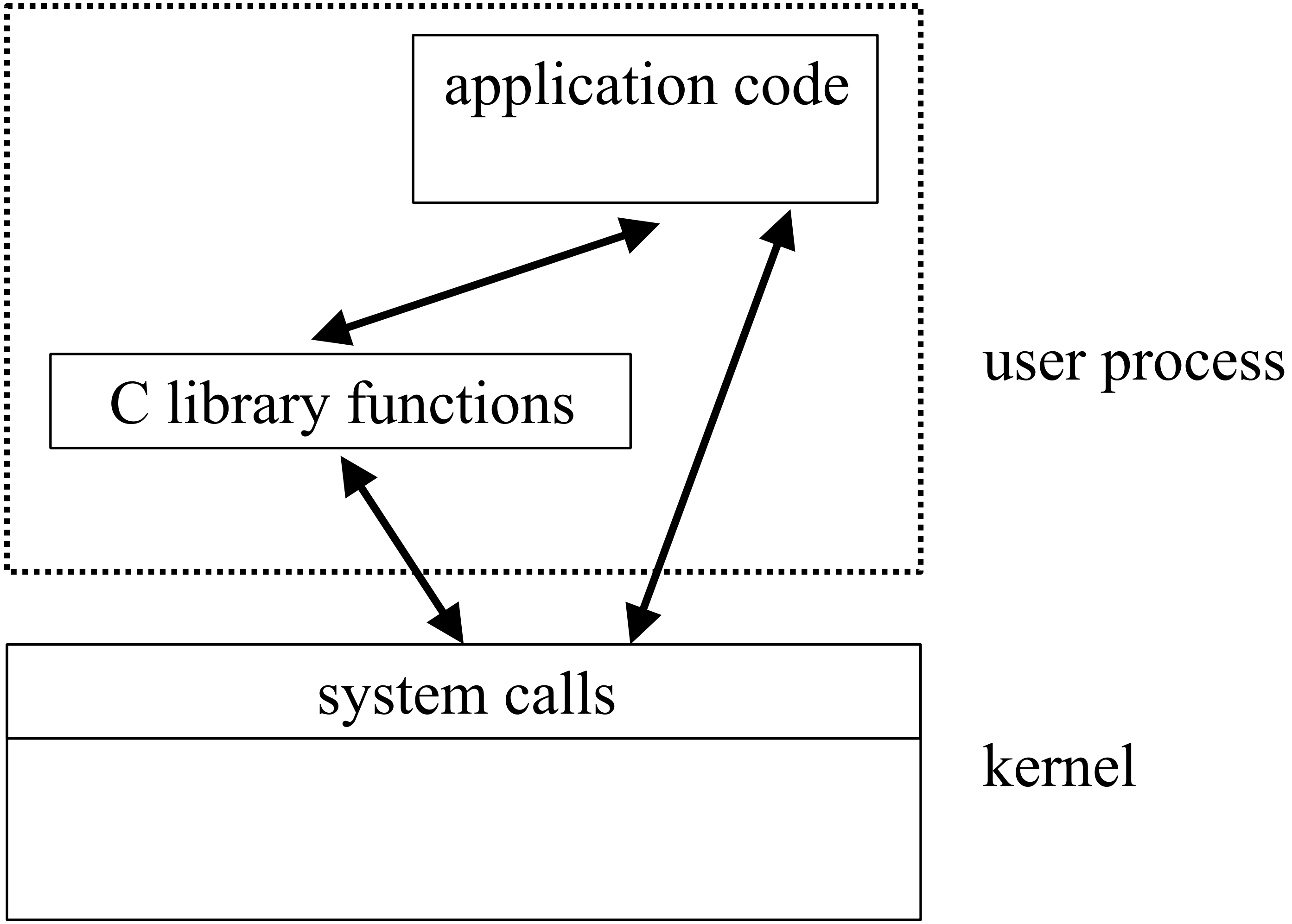
11. System Calls & Library Functions

- **System Calls**:
  Entry points into an OS kernel
- Cannot be changed by user
- A function of the same name in the standard C library
- User just calls those C functions whenever system calls are needed

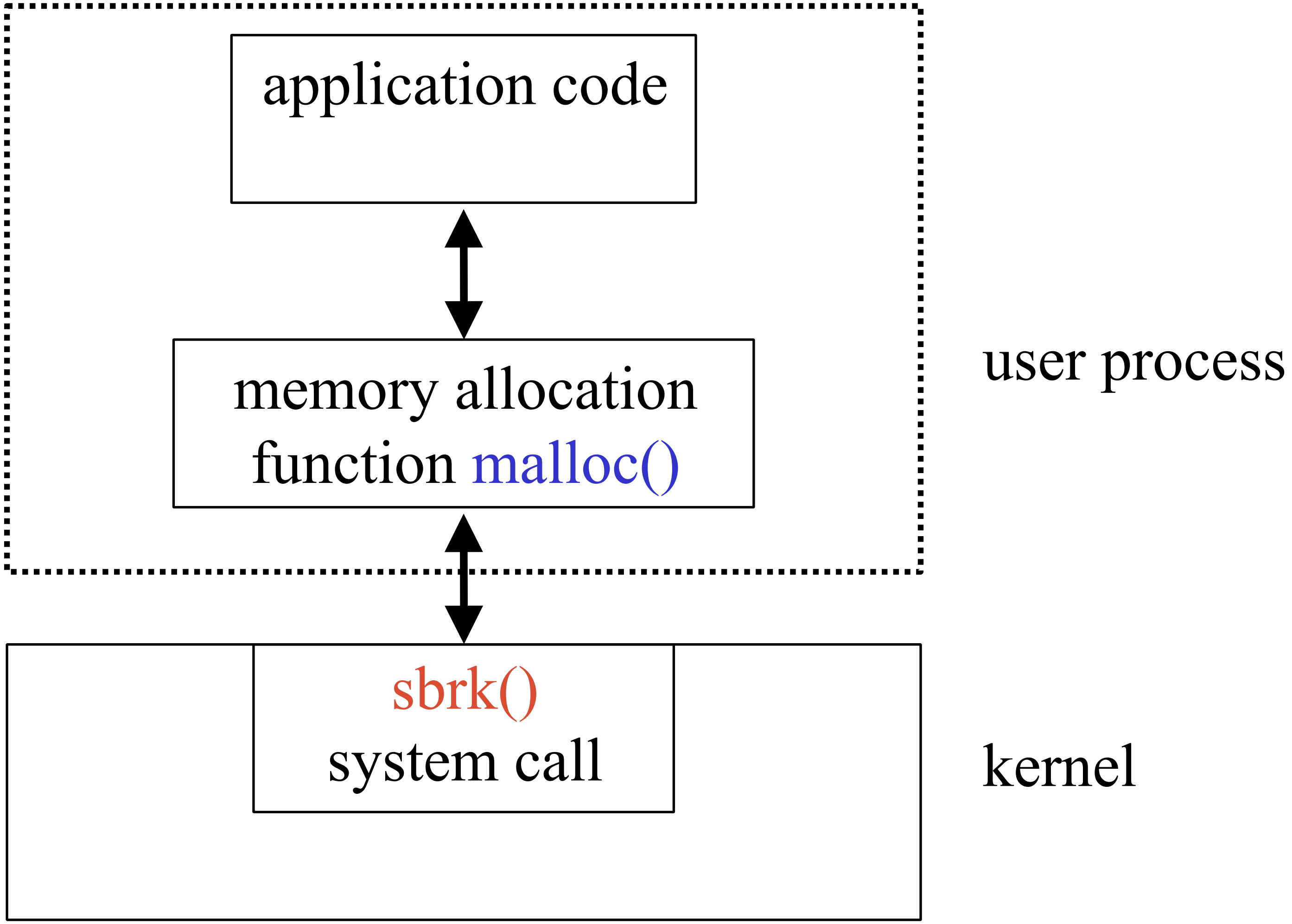
System Calls & Library Functions

- **Library Functions**: not entry points into kernel, just functions, but they may invoke one or more system calls
  - E.g.: printf() invokes write() system call
  - E.g.: strcpy(), atoi(): do not invoke any system call
- Implementor view: fundamental diff
- Programmer view: no critical difference

System Calls & Library Functions

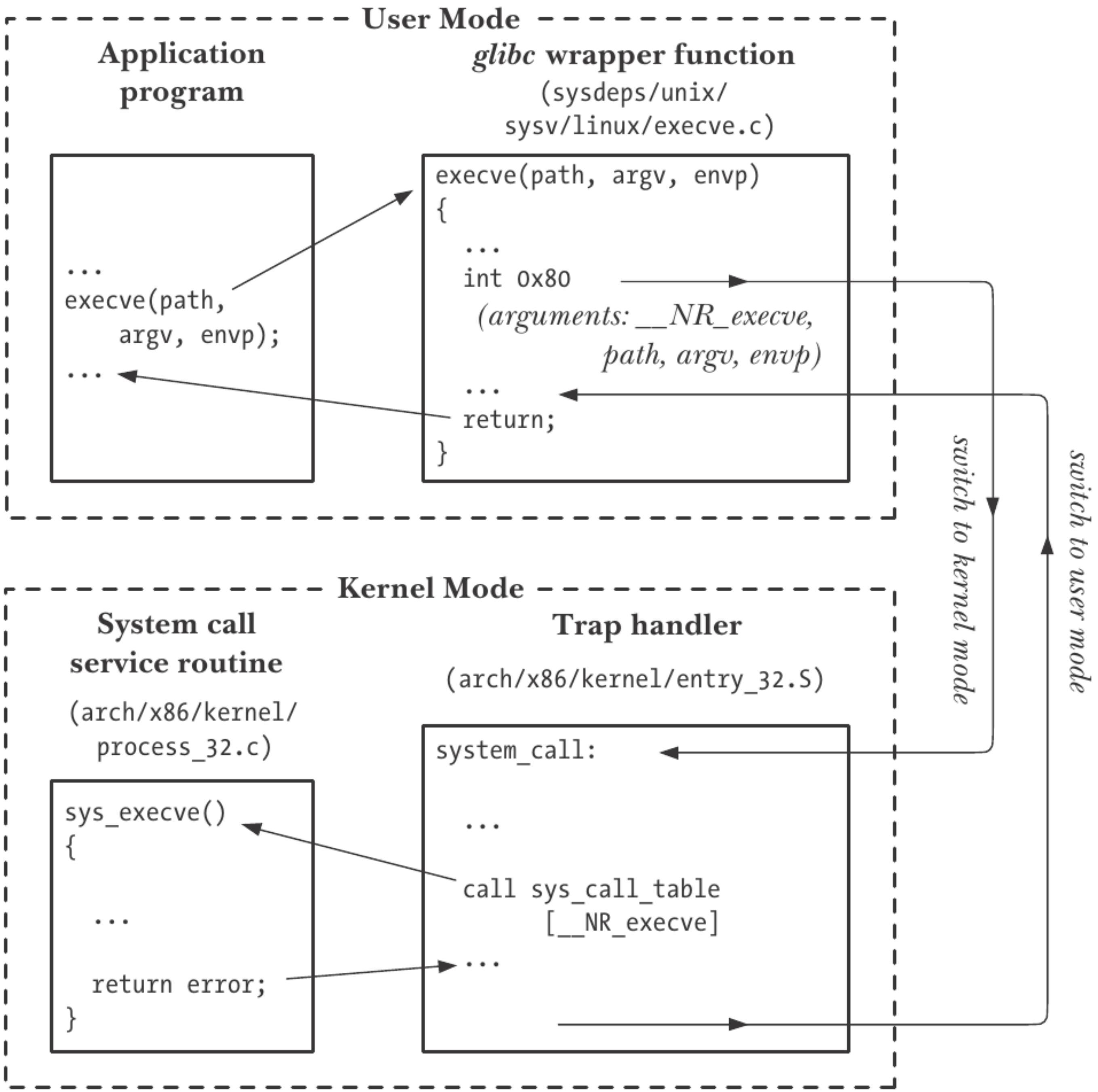


System Calls & Library Functions Example



Example

- ➢execve
- ➢glibc



User Mode
Application program
glibc wrapper function
(sysdeps/unix/sysv/linux/execve.c)
Kernel Mode
System call service routine
(arch/x86/kernel/process_32.c)
Trap handler
(arch/x86/kernel/entry_32.S)
switch to kernel mode
switch to user mode

```
...
execve(path,
    argv, envp);
...
```

```
execve(path, argv, envp)
{
  ...
  int 0x80
   (arguments: __NR_execve,
               path, argv, envp)
  ...
  return;
}
```

```
sys_execve()
{

  ...

  return error;
}
```

```
system_call:

  ...

  call sys_call_table
          [__NR_execve]
  ...
```