Intro to System Calls

Dr. Chokchai (Box) Leangsuksun

Louisiana Tech University

Outline

- Introduction to System Call
- Anatomy of system calls
- Some useful system calls
  - Process control
  - File management
  - Information maintenance
  - Communications

System Calls

- Programming interface to the services provided by the OS
- Typically written and used in a high-level language (C or C ++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
-

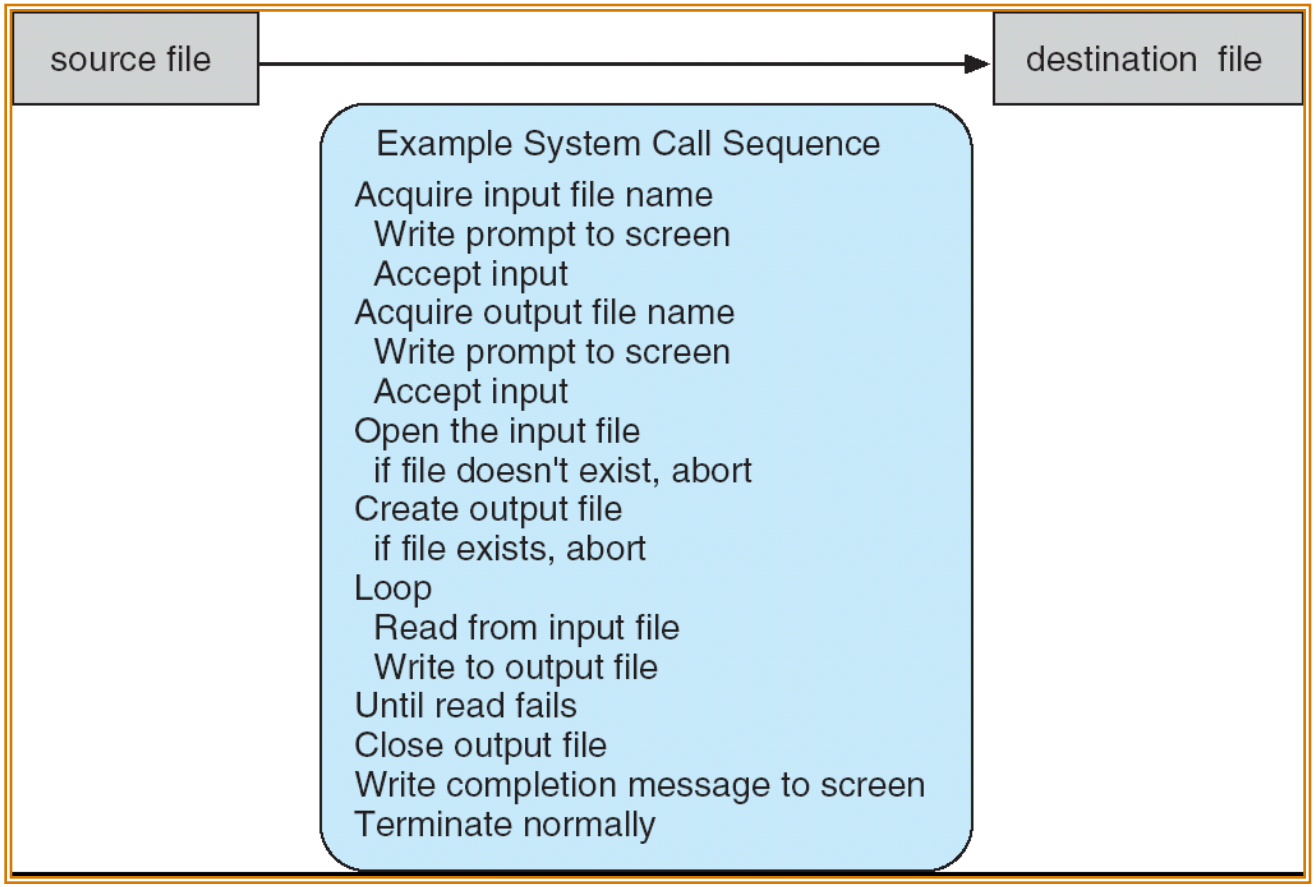
(Note that the system-call names used throughout this text are generic)

Linux/Unix System Calls

- Linux/UNIX has about 60 system calls
- The most calls are written in C.
- can be accessed from C programs.
- There are similar system programs that provide similar system call features/services
- Basic I/0
- Process control (creation, termination, execution)
- File operations and permission
- System status

Example of System Calls

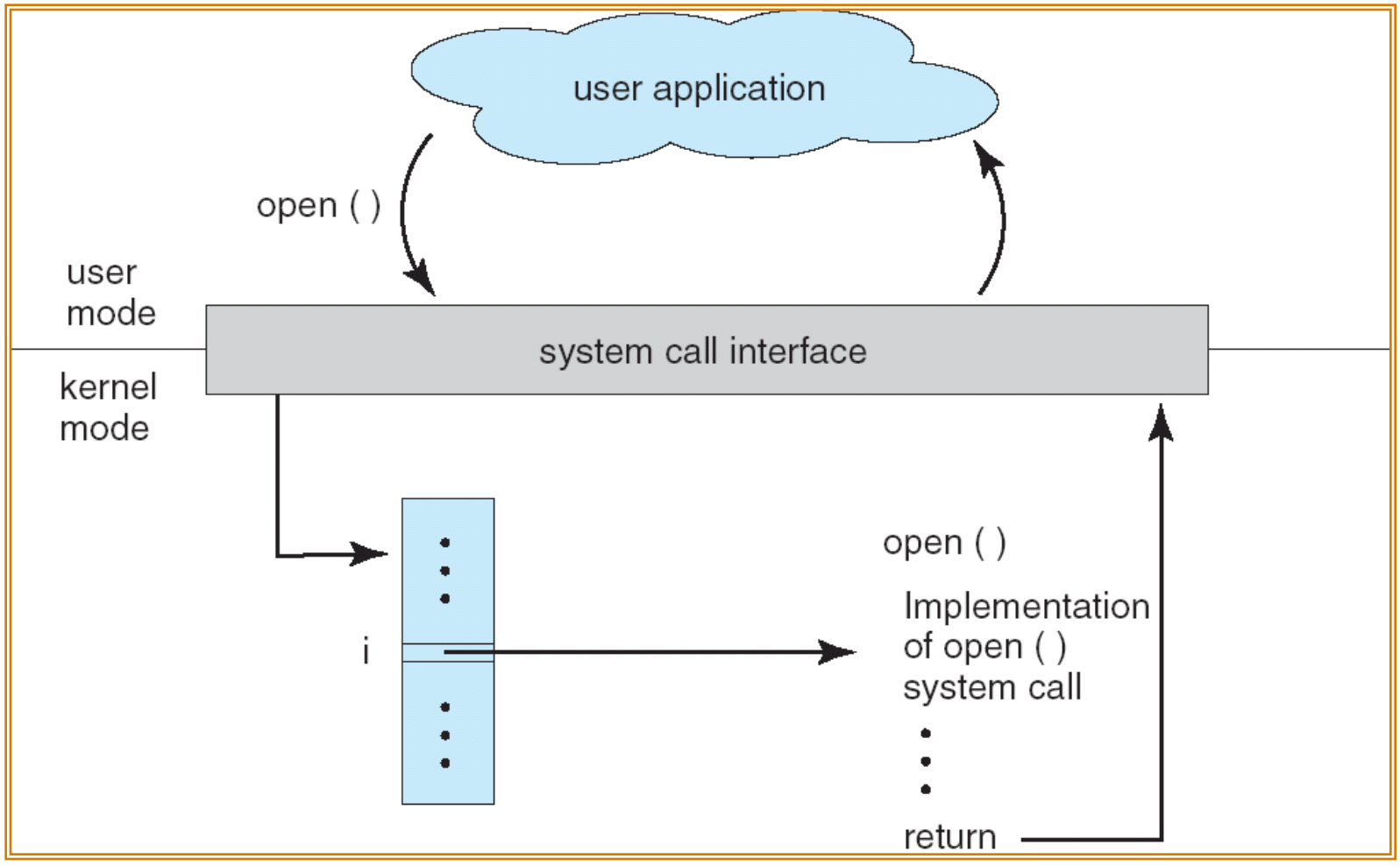
- System call sequence to copy the contents of one file to another file



System Call Implementation

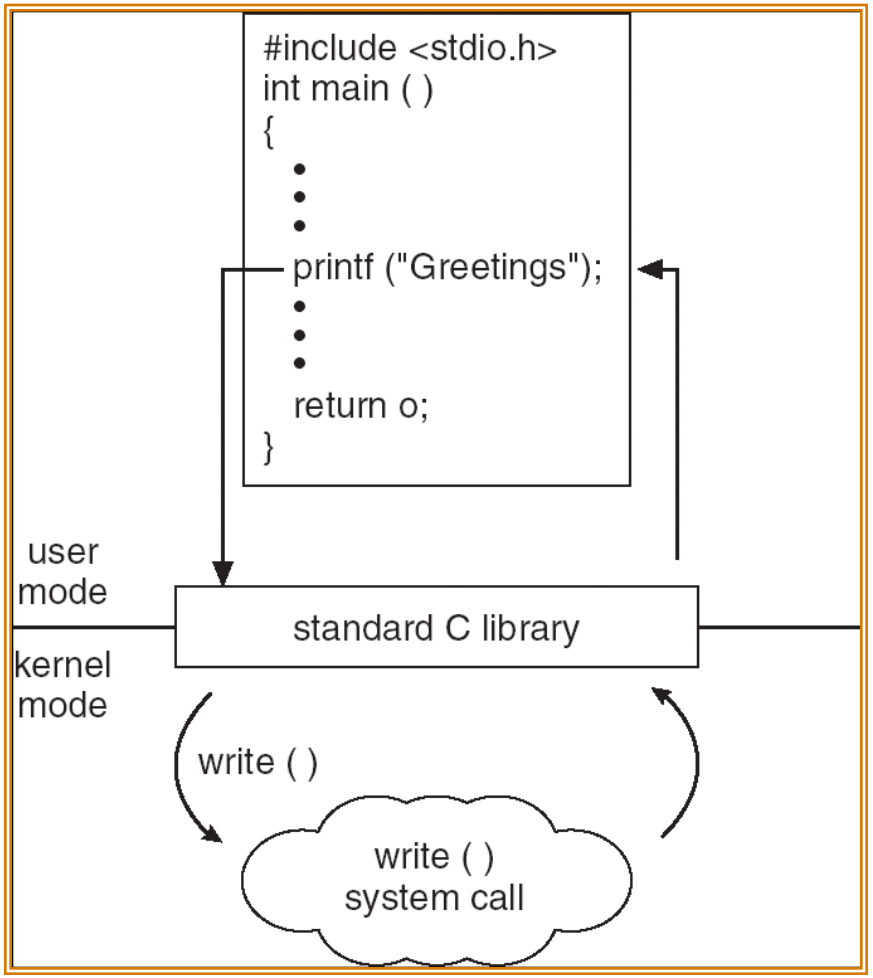
- Typically, a number associated with each system call
  - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)

API – System Call – OS Relationship



Standard C Library Example

- C program invoking printf() library call, which calls write() system call

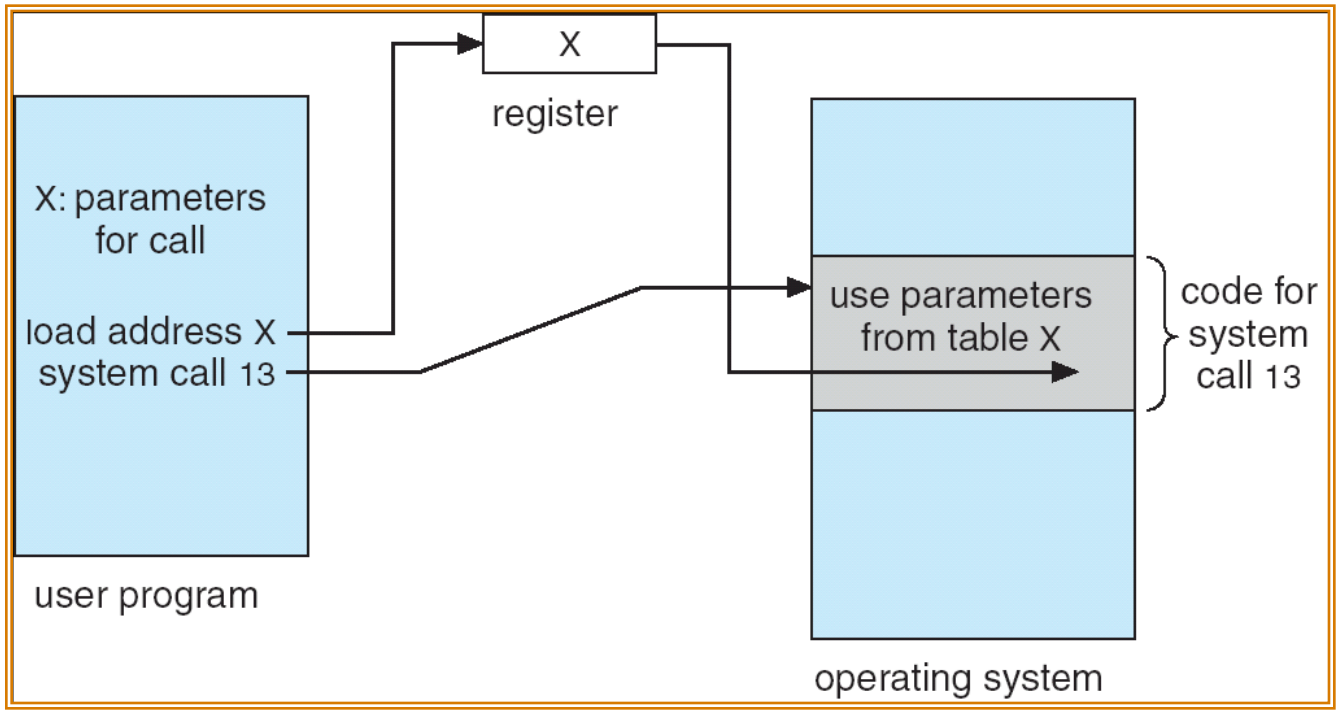


System Call Parameter Passing

- Often, more information is required than simply identity of desired system call
  - Exact type and amount of information vary according to OS and call
- Three general methods used to pass parameters to the OS
  1. Simplest: pass the parameters in *registers*
     - In some cases, may be more parameters than registers
  2. Parameters stored in a *block,* or table, in memory, and address of block passed as a parameter in a register
     - This approach taken by Linux and Solaris
  3. Parameters placed, or *pushed,* onto the *stack* by the program and *popped* off the stack by the operating system

  - Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table



Types of System Calls

- Process control
- File management
- Device management
- Information maintenance
- Communications

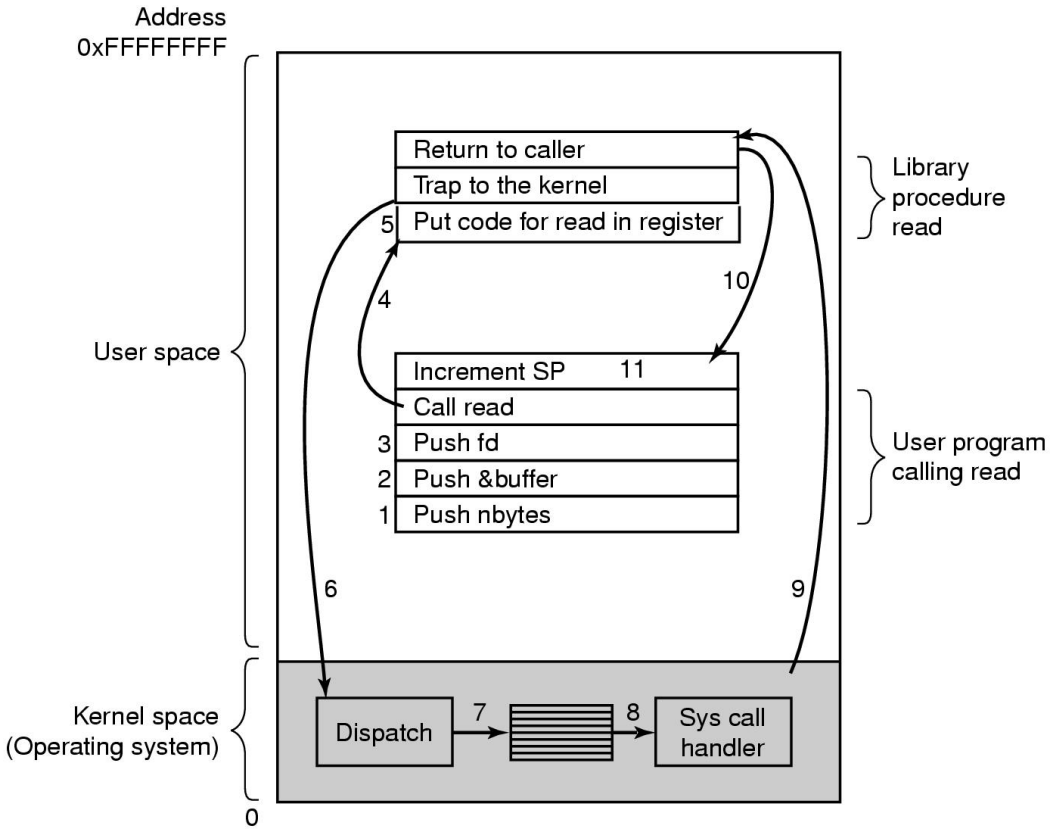
Linux System Calls

- System calls are low level functions the operating system makes available to applications via a defined API (Application Programming Interface)
- System calls represent the *interface* the kernel presents to user applications.
- In Linux all low-level I/O is done by reading and writing file handles, regardless of what particular peripheral device is being accessed—a tape, a socket, even your terminal, they are all *files*.
- Low level I/O is performed by making *system calls*.

Anatomy of a Linux System Call

- A System Call is an explicit request to the kernel made via a software interrupt.
- The interrupt call '0x80' call to a system call handler (sometimes called the "call gate").
- The system call handler in turns calls the system call interrupt service routine (ISR).
- To perform Linux system calls we have to do following:
  - Put the system call number in `EAX` register.
  - Set up the arguments to the system call in `EBX,ECX, etc.`
  - call the relevant interrupt (`for Linux, 80h).`
  - The result is usually returned in `EAX.`

Sample of how a System Call is made




```
read (fd, buffer, nbytes)
```

POSIX APIs vs. System Calls

- An ***application programmer interface*** is a ***function definition*** that specifies how to obtain a given service.
- A ***system call*** is an explicit request to the kernel made via a ***software interrupt***.

From a Wrapper Routine to a System Call

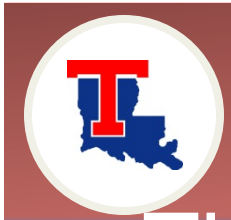
- Unix systems include several ***libraries of functions*** that provide **API**s to programmers.
- Some of the **API**s defined by the `libc` standard **C** library refer to ***wrapper routines*** (routines whose only purpose is to issue a ***system call***).
- Usually, each system call has a corresponding wrapper routine, which defines the **API** that application programs should employ.

APIs and System Calls

- An **API** does not necessarily correspond to a specific system call.
  - First of all, the **API** could offer its services directly in User Mode. (For something abstract such as math functions, there may be no reason to make system calls.)
  - Second, a single **API** function could make several system calls.
  - Moreover, several **API** functions could make the same system call, but wrap extra functionality around it.

Example of Different APIs Issuing the Same System Call

- In Linux, the `malloc( )` , `calloc( )` , and `free( )` **API**s are implemented in the `libc` library.
- The code in this library keeps track of the allocation and deallocation requests and uses the `brk( )` system call to enlarge or shrink the ***process heap***.
  - P.S.: See the section "Managing the Heap" in Chapter 9.



The Return Value of a Wrapper Routine

- Most wrapper routines return an integer value, whose meaning depends on the corresponding system call.
- A return value of **-1** usually indicates that the ***kernel*** was unable to satisfy the process request.
- A failure in the ***system call handler*** may be caused by
  - invalid parameters
  - a lack of available resources
  - hardware problems, and so on.
- The specific ***error code*** is contained in the `errno` variable, which is defined in the `libc` library.

Sample of System Calls

Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

File Management

File management

Call	Description
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

Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Miscellaneous Tasks

Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

System Calls (5)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Some Win32 API calls

Process Management

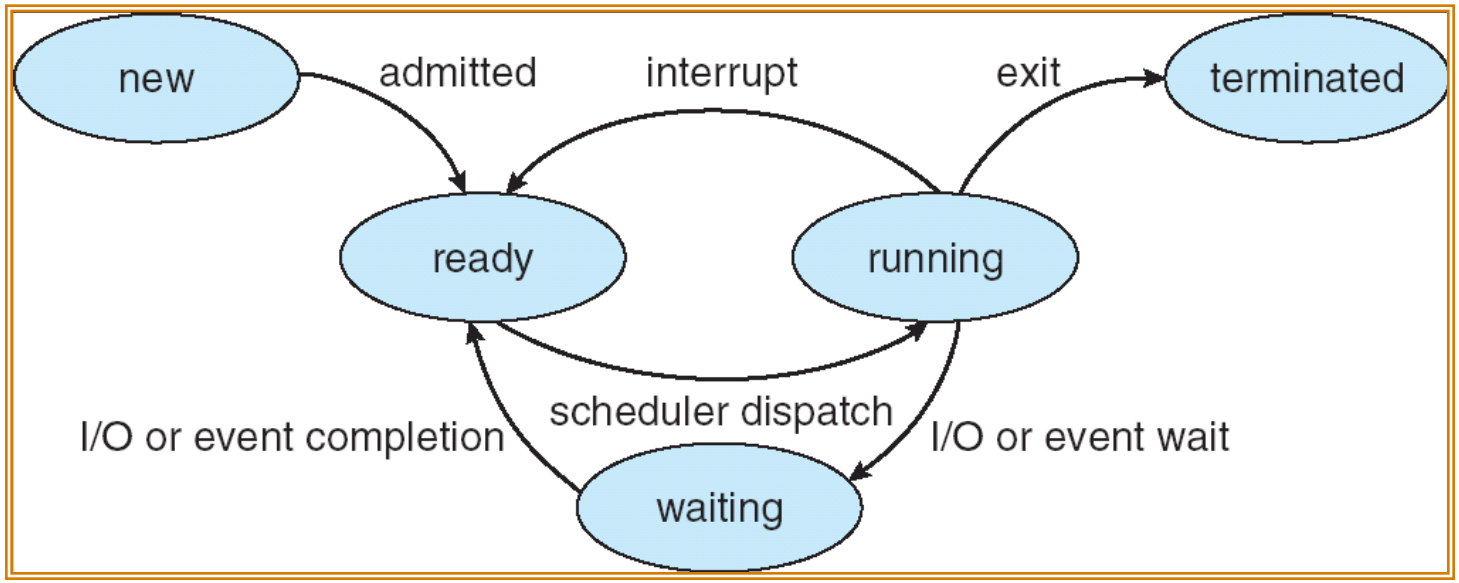
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program
  - program counter
  - stack
  - data section
- In Linux, how do you see process info and what can you see?

Process State

- As a process executes, it changes *state*
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a process
  - **terminated**: The process has finished execution

Diagram of Process State

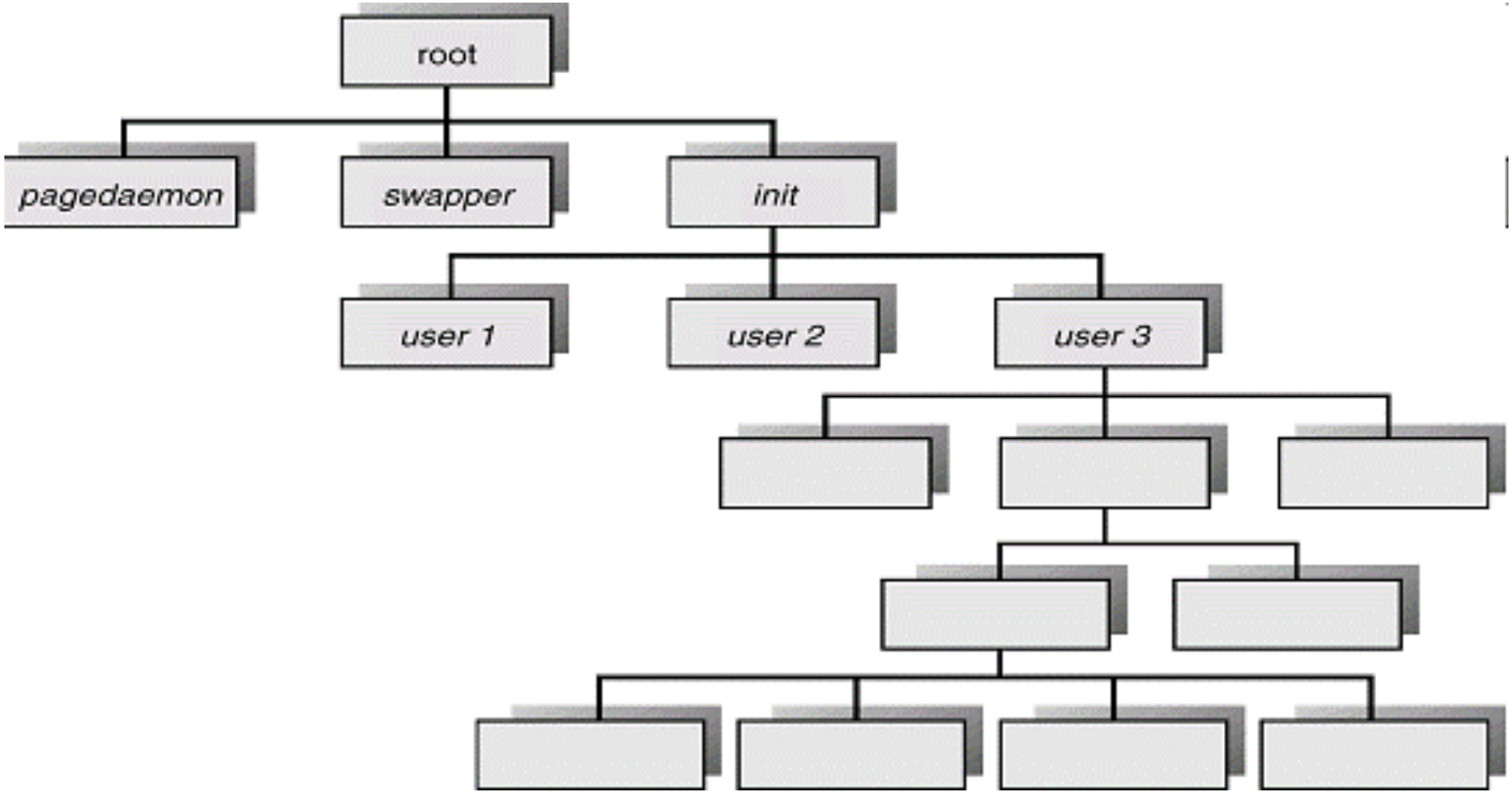


Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Resource sharing
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution
  - Parent and children execute concurrently
  - Parent waits until children terminate

- **fork()** system call creates new process

Linux Process



fork(): Creating a Process

Include File(s)	<sys/ types.h> <unistd.h>	Manual Section	2
Summary	`pid_t fork ( void );`		
Return	Success	Failure	Sets errno
	0 in child, child process ID in the parent	-1	Yes

fork(): Creating a Process

- The fork system call does not take an argument.
- If the fork system call fails, it returns a −1 and sets the value in errno to indicate one of the error conditions.

fork(): errono

#	Constant	perror Message	Explanation
11	EAGAIN	Resource temporarily unavailable	The operating system was unable to allocate sufficient memory to copy the parent's page table information and allocate a task structure for the child.
12	ENOMEM	Cannot allocate memory	Insufficient swap space available to generate another process.

Example of a fork system call

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

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

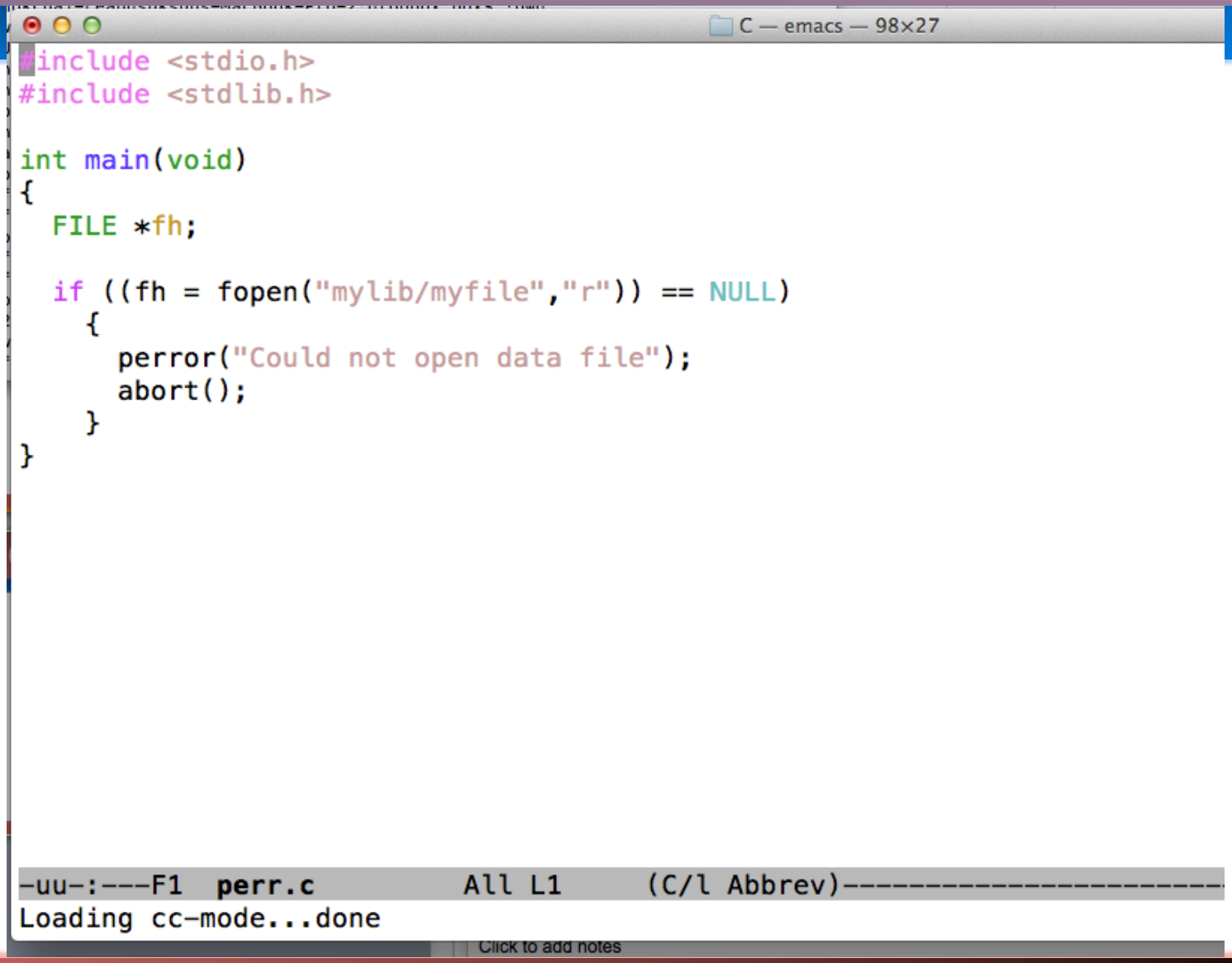
Show sample of execlp & fork

Managing Failures

Include File(s)	`<stdio.h>`	Manual Section	3
Summary	**void perror (const char *s);** writes a description of the current error to stderr		
Return	**Success**	**Failure**	**Sets errno**

Managing Failures

- The summary table for perror indicates the header file <stdio.h> must be included if we want to use perror.
- The perror library function takes a single argument:
  - The argument is a pointer to a character string constant (i.e., const char *).
- The perror library function does not return a value (as indicated by the data type void) and will not modify errno if it itself fails.


C — emacs — 98×27
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
FILE *fh;
if ((fh = fopen("mylib/myfile","r")) == NULL)
{
perror("Could not open data file");
abort();
}
}
-uu-:---F1 perr.c All L1 (C/l Abbrev)----------
Loading cc-mode...done
Click to add notes

Managing Failures

Include File(s)	`<stdio.h>`
	`<errno.h>`
	`<string.h>`
		Manual Section	3

Summary	**char *strerror(int errnum);** get a human-readable string for the error number		
Return	**Success**	**Failure**	**Sets errno**
	Reference to error message		

Managing Failures

- The strerror function maps the integer errnum argument (which is usually the errno value) to an error message and returns a reference to the message.
- The error message generated by strerror should be the same as the message generated by perror.
- If needed, additional text can be appended to the string returned by strerror.

C — emacs — 98×27

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

extern int errno ;

int main ()
{
  FILE * pFile;
  pFile = fopen ("unexist.ent","rb");
  if (pFile == NULL)
    {
      printf( "Error Value is : %d, Error String is  %s\n", errno,  strerror(errno) );
    }
  else
    fclose (pFile);
  return 0;
}
```

```
-uu-:---F1  strerr.c       All L1     (C/l Abbrev)----------------------------------
Loading cc-mode...done
```

A series of execxx functions

- **exec*xx*()** system call used after a **fork** to replace the process' memory space with a new program
- ```
  extern char **environ;
  int execl(const char *path, const char *arg0, ... /
  *, (char *)0 */);
  int execle(const char *path, const char *arg0, ... /
  *,
          (char *)0, char *const envp[]*/);
  int execlp(const char *file, const char *arg0, ... /
  *, (char *)0 */);
  int execv(const char *path, char *const argv[]);
  int execve(const char *path, char *const argv[],
  char *const envp[]);
  int execvp(const char *file, char *const argv[]);
  ```
-
  ```

Using of exec*xx* calls

- **Using execl()**
- The following example executes the *ls* command, specifying the pathname of the executable (**/bin/ls**) and using arguments supplied directly to the command to produce single-column output.

```
#include <unistd.h>

int ret; ... ret = execl ("/bin/ls", "ls", "-1", (char *)0);
```

- **Using execle()**
- The following example is similar to Using execl(). In addition, it specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>
 int ret; char *env[] = { "HOME=/usr/home",
"LOGNAME=home", (char *)0 };
...

 ret = execle ("/bin/ls", "ls", "-l", (char *)0,
env);
```

- **Using execlp()**
- The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable.

```
#include <unistd.h>
 int ret;

...

ret = execlp ("ls", "ls", "-l", (char *)0);
```

- **Using execvp()**
- The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable, and passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>
 int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execvp ("ls", cmd);
```

execlp

- The execlp library function is used when the number of arguments to be passed to the program to be executed is known in advance.

(Table 3.3)

execlp

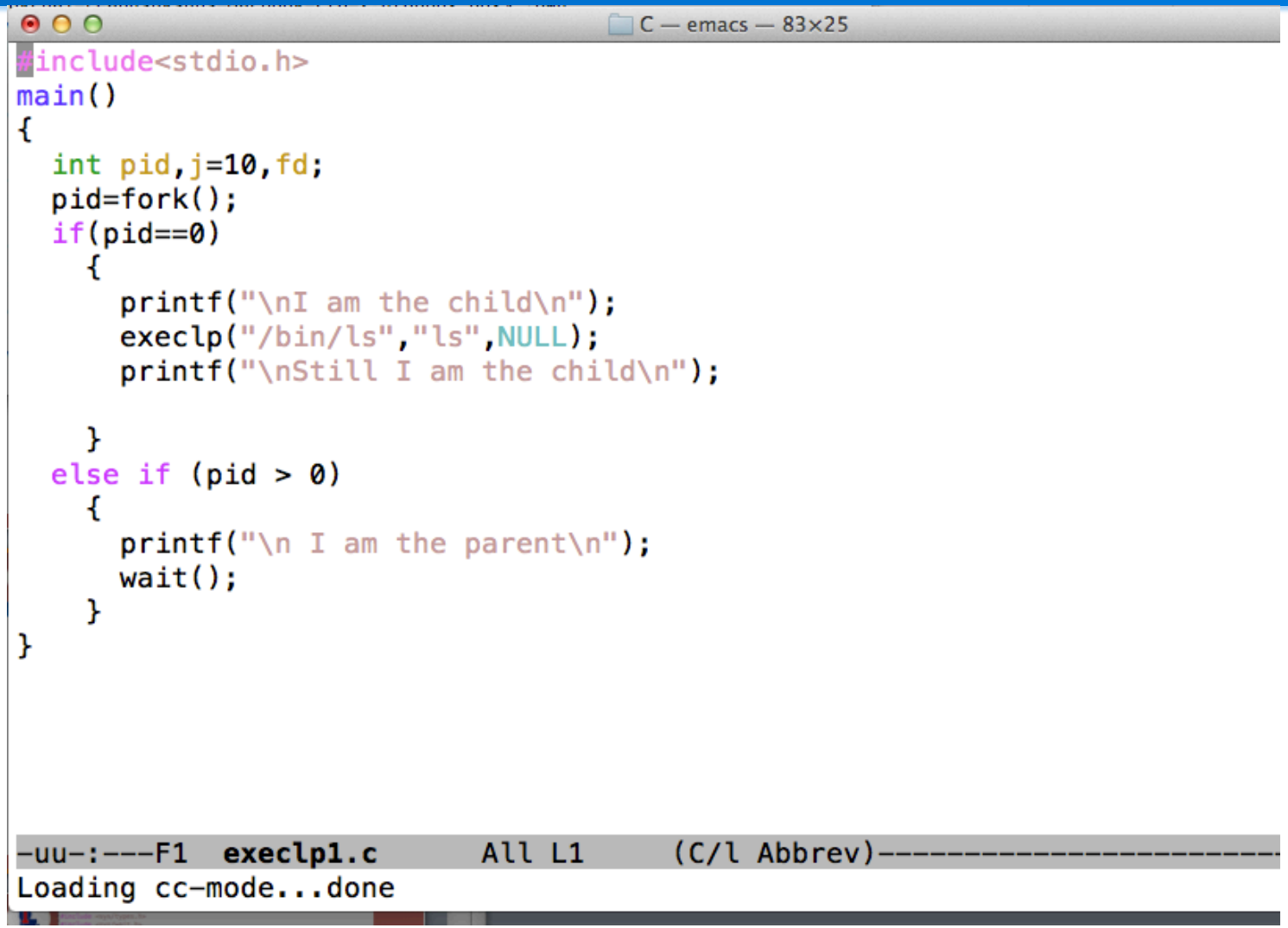
Include File(s)	<unistd.h> extern char *environ;	Manual Section	3
Summary	int execlp(const char *file,const char *arg, . . .);		
Return	Success	Failure	Sets errno
	Does not return	-1	Yes

Table 3.3. Summary of the execlp Library Function.

execlp

- The initial argument, file, is a pointer to the file that contains the program code to be executed.
- For the execlp call to be successful, the file referenced must be found and be marked as executable.
- If the call fails, it returns a −1 and sets errno to indicate the error.

(Table 3.4).


C — emacs — 83×25
#include<stdio.h>
main()
{
int pid,j=10,fd;
pid=fork();
if(pid==0)
{
printf("\nI am the child\n");
execlp("/bin/ls","ls",NULL);
printf("\nStill I am the child\n");
}
else if (pid > 0)
{
printf("\n I am the parent\n");
wait();
}
}
-uu-:---F1 execlp1.c All L1 (C/l Abbrev)----------
Loading cc-mode...done

execlp

#	Constant	perror Message	Explanation
1	EPERM	Operation not permitted	• The process is being traced, the user is not the superuser, and the file has an SUID or SGID bit set. • The file system is mounted nosuid, the user is not the superuser, and the file has an SUID or SGID bit set.
2	ENOENT	No such file or directory	One or more parts of path to new process file does not exist (or is NULL).
4	EINTR	Interrupted system call	Signal was caught during the system call.
5	EIO	Input/output error	
7	E2BIG	Argument list too long	New process argument list plus exported shell variables exceed the system limits.

. Table 3.4. exec Error Messages

execlp

#	Constant	perror Message	Explanation
8	ENOEXEC	Exec format error	New process file is not in a recognized format.
11	EAGAIN	Resource temporarily unavailable	Total system memory while reading raw I/O is temporarily insufficient.
12	ENOMEM	Cannot allocate memory	New process memory requirements exceed system limits.
13	EACCES	Permission denied	• Search permission denied on part of file path. • The new file to process is not an ordinary file. • No execute permission on the new file to process.
14	EFAULT	Bad address	path references an illegal address.
20	ENOTDIR	Not a directory	Part of the specified path is not a directory.

. Table 3.4. exec Error Messages

execlp

#	Constant	perror Message	Explanation
21	EISDIR	Is a directory	An ELF interpreter was a directory.
22	EINVAL	Invalid argument	An ELF executable had more than one interpreter.
24	EMFILE	Too many open files	Process has exceeded the maximum number of files open.
26	ETXTBSY	Text file busy	More than one process has the executable open for writing.
36	ENAMETOOLONG	File name too long	The path value exceeds system path/file name length.
40	ELOOP	Too many levels of symbolic links	The perror message says it all.
67	ENOLINK	Link has been severed	The path value references a remote system that is no longer active.

. Table 3.4. exec Error Messages

execlp

#	Constant	perror Message	Explanation
72	EMULTIHOP	Multihop attempted	The path value requires multiple hops to remote systems, but file system does not allow it.
80	ELIBBAD	Accessing a corrupted shared library	An ELF interpreter was not in a recognized format.

. Table 3.4. exec Error Messages

execvp

- If the number of arguments for the program to be executed is dynamic, then the execvp call can be used.

(Table 3.5).

execvp

Include File(s)	<unistd.h> <extern char *environ;	Manual Section	3
Summary	Int execvp(const char *file, char *const argv[]);		
Return	Success	Failure	Sets errno
	Does not return	-1	Yes

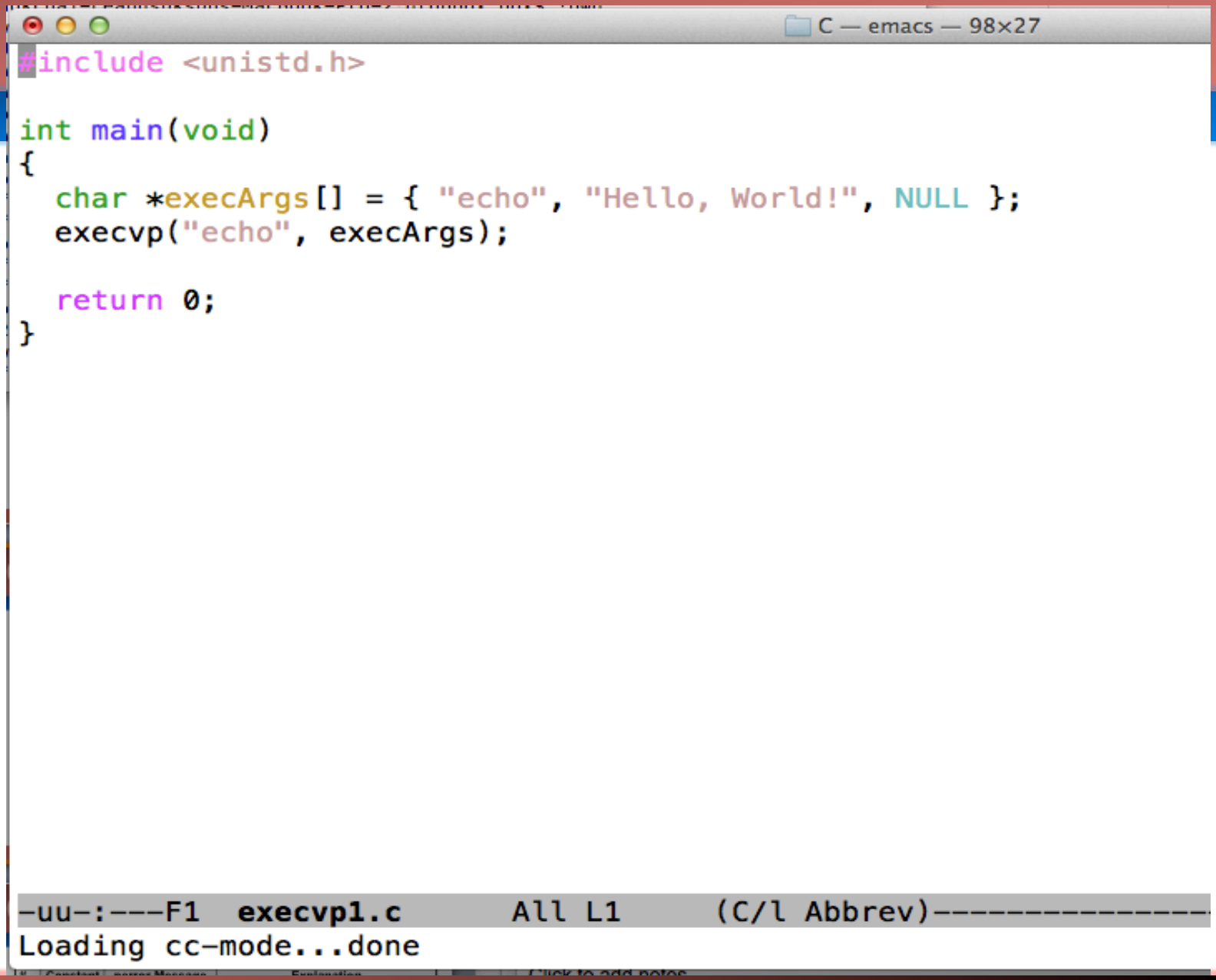
Table 3.5. Summary of the execvp System Call.

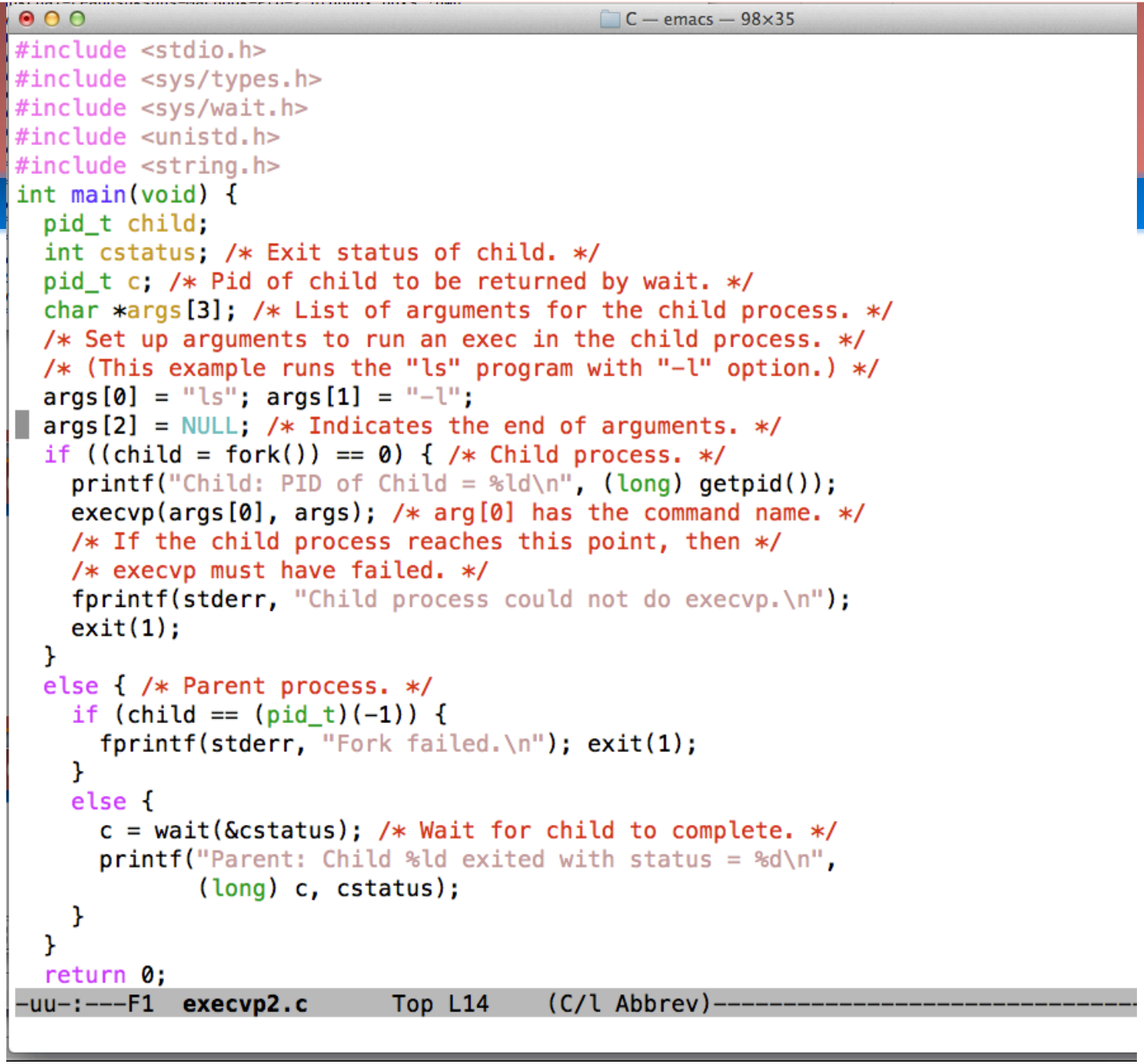
execvp

- As with the execlp call, the initial argument to execvp is a pointer to the file that contains the program code to be executed.
- Unlike execlp, there is only one additional argument that execvp requires.
- The second argument specifies that a reference to an array of pointers to character strings should be passed.

execvp

- If execvp fails, it returns a value of −1 and sets the value in errno to indicate the source of the error.


C — emacs — 98×27
#include <unistd.h>
int main(void)
{
char *execArgs[] = { "echo", "Hello, World!", NULL };
execvp("echo", execArgs);
return 0;
}
-uu-:---F1 execvp1.c All L1 (C/l Abbrev)----------
Loading cc-mode...done


C — emacs — 98×35
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <string.h>
int main(void) {
pid_t child;
int cstatus; /* Exit status of child. */
pid_t c; /* Pid of child to be returned by wait. */
char *args[3]; /* List of arguments for the child process. */
/* Set up arguments to run an exec in the child process. */
/* (This example runs the "ls" program with "-l" option.) */
args[0] = "ls"; args[1] = "-l";
args[2] = NULL; /* Indicates the end of arguments. */
if ((child = fork()) == 0) { /* Child process. */
printf("Child: PID of Child = %ld\n", (long) getpid());
execvp(args[0], args); /* arg[0] has the command name. */
/* If the child process reaches this point, then */
/* execvp must have failed. */
fprintf(stderr, "Child process could not do execvp.\n");
exit(1);
}
else { /* Parent process. */
if (child == (pid_t)(-1)) {
fprintf(stderr, "Fork failed.\n"); exit(1);
}
else {
c = wait(&cstatus); /* Wait for child to complete. */
printf("Parent: Child %ld exited with status = %d\n",
(long) c, cstatus);
}
}
return 0;
-uu-:---F1 execvp2.c Top L14 (C/l Abbrev)

terminating a Process

- A process normally terminates in one of three ways.
- In order of preference, these are
  - It issues (at any point in its code) a call to either exit or _exit.
  - It issues a return in the function main.
  - It falls off the end of the function main ending implicitly.
- Programmers routinely make use of the library function exit to terminate programs.
- This function does not return a value.

exit(): Ending a Process

Include File(s)	<stdlib.h>	Manual Section	3
Summary	void exit(int status);		
Return	Success	Failure	Sets errno
	Does not return	No return	Yes

Ending a Process

- The exit function accepts a single parameter, an integer status value that will be returned to the parent process.
- By convention, a 0 value is returned if the program has terminated normally; other wise, a nonzero value is returned.

Ending a Process

Include File(s)	<stdlib.h>	Manual Section	3
Summary	int atexit(void (*function)(void));		
Return	Success	Failure	Sets errno
	0	-1	No

Table 3.7. Summary of the atexit Library Function.

Ending a Process

- The atexit function is relatively new.
- The definition of atexit indicates that functions to be called (when the process terminates normally) are registered by passing the atexit function the address of the function.
- The registered functions should not have any parameters.
- If atexit is successful in registering the function, atexit returns a 0; otherwise, it returns a −1 but will not set errno. (Table 3.7)

Show: See the example of atexit

Ending a Process

- Programmers may call _exit directly if they wish to circumvent the invocation of atexit registered functions and the flushing of I/O buffers.

  (Table 3.8)

Ending a Process

Include File(s)	<stdlib.h>	Manual Section	3
Summary	void _exit(int status);		
Return	Success	Failure	Sets errno
	Does not return	Does not return	

Table 3.8. Summary of the _exit System Call.

Ending a Process

- The _exit system call, like its relative, exit, does not return.
- This call also accepts an integer status value, which will be made available to the parent process.

Ending a Process

- When terminating a process, the system performs a number of housekeeping operations:
  - All open file descriptors are closed.
  - The parent of the process is notified (via a SIGCHLD signal) that the process is terminating.
  - Status information is returned to the parent process (if it is waiting for it). If the parent process is not waiting, the system stores
  - the status information until a wait by the parent process is affected.
  - All child processes of the terminating process have their parent process ID (PPID) set to 1—they are inherited by init.

Ending a Process

- If the process was a group leader, process group members will be sent SIGHUP/ SIGCONT signals.
- Shared memory segments and semaphore references are readjusted.
- If the process was running accounting, the accounting record is written out to the accounting file.

Waiting on Processes

- More often than not, a parent process needs to synchronize its actions by waiting until a child process has either stopped or terminated its actions.
- The wait system call allows the parent process to suspend its activity until one of these actions has occurred.

(Table 3.9).

Waiting on Processes

Include File(s)	<sys/types.h> <sys/wait.h>	Manual Section	2
Summary	pid_t wait(int *status);		
Return	Success	Failure	Sets errno
	Child process ID or 0	-1	Yes

Table 3.9. Summary of the wait System Call.

Waiting on Processes

- The wait system call accepts a single argument, which is a pointer to an integer, and returns a value defined as type pid_t.

Waiting on Processes

- While the wait system call is helpful, it does have some limitations:
  - It will always return the status of the first child process that terminates or stops. Thus, if the status information returned by wait is not from the child process we want, the information may need to be stored on a temporary basis for possible future reference and additional calls to wait made.
  - Another limitation of wait is that it will always block if status information is not available.

Waiting on Processes

- Another system call, waitpid, which is more flexible (and thus more complex), addresses these shortcomings.
- In most invocations, the waitpid call will block the calling process until one of the specified child processes changes state.

(Table 3.11.)

Waiting on Processes

Include File(s)	<sys/types.h> <sys/wait.h>	Manual Section	2
Summary	pid_t waitpid(pid_t pid, int *status, int options);		
Return	Success	Failure	Sets errno
	Child PID or 0	-1	Yes

Table 3.11. Summary of the waitpid System Call.

Waiting on Processes

- The first argument of the waitpid system call, pid, is used to stipulate the set of child process identification numbers that should be waited for. (Table 3.12).

Waiting on Processes

pid Value	Wait for
<-1	Any child process whose process group ID equals the absolute value of pid.
-1	Any child process—in a manner similar to wait.
0	Any child process whose process group ID equals the caller's process group ID.
>0	The child process with this process ID.

. Table 3.12. Interpretation of pid Values by waitpid

Waiting on Processes

- The second argument, *status, as with the wait call, references an integer status location where the status information of the child process will be stored if the waitpid call is successful.
- The third argument, options, may be 0 (don't care), or it can be formed by a bitwise OR of one or more of the flags listed in.
- (Table 3.13).

Waiting on Processes

FLAG Value	Specifies
WNOHANG	Return immediately if no child has exited—do not block if the status cannot be obtained; return a value of 0, not the PID.
WUNTRACED	Return immediately if child is blocked.

. Table 3.13. Flag Values for waitpid

Waiting on Processes

- If the value given for pid is -1 and the option flag is set to 0, the waitpid and wait system call act in a similar fashion.
- If waitpid fails, it returns a value of –1 and sets errno to indicate the source of the error.

  (Table 3.14).

Waiting on Processes

#	Constant	perror Message	Explanation
4	EINTR	Interrupted system call	Signal was caught during the system call.
10	ECHILD	No child process	Process specified by pid does not exist, or child process has set action of SIGCHILD to be SIG_IGN (ignore signal).
22	EINVAL	Invalid argument	Invalid value for options.
85	ERESTART	Interrupted system call should be restarted	WNOHANG not specified, and unblocked signal or SIGCHILD was caught.

. Table 3.14. waitpid Error Messages

Waiting on Processes

- On some occasions, the information returned from wait or waitpid may be insufficient.
- Additional information on resource usage by a child process may be sought.
- There are two BSD compatibility library functions, wait3 and wait4, that can be used to provide this information.

(Table 3.15).

Waiting on Processes

Include File(s)	`#define _USE_BSD`
	`#include <sys/types.h>`
	`#include <sys/resource.h>`
	`#include <sys/wait.h>`
		Manual Section	3

Summary	pid_t wait3(int *status, int options, struct rusage *rusage); pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);		
Return	Success	Failure	Sets errno
	Child PID or 0	-1	Yes

Table 3.15. Summary of the wait3/wait4 Library Functions.

Waiting on Processes

- The wait3 and wait4 functions parallel the wait and waitpid functions respectively.
- The wait3 function waits for the first child process to terminate or stop.
- The wait4 function waits for the specified PID (pid).
- In addition, should the pid value passed to the wait4 function be set to 0, wait4 will wait on the first child process in a manner similar to wait3.

Waiting on Processes

- Both functions accept option flags to indicate whether or not they should block and/or report on stopped child processes.

(Table 3.16.)

Waiting on Processes

FLAG Value	Specifies
WNOHANG	Return immediately if no child has exited—do not block if the status cannot be obtained; return a value of 0, not the PID.
WUNTRACED	Return immediately if child is blocked.

. Table 3.16. Option Flag Values for wait3/wait4

Waiting on Processes

- The status macros can be used with the status information returned by wait3 and wait4.

(Table 3.17.)

Waiting on Processes

#	Constant	perror Message	Explanation
4	EINTR	Interrupted system call	Signal was caught during the system call.
10	ECHILD	No child process	Process specified by pid does not exist, or child process has set action of SIGCHILD to be SIG_IGN (ignore signal).
22	EINVAL	Invalid argument	Invalid value for options.
85	ERESTART	Interrupted system call should be restarted	WNOHANG not specified, and unblocked signal or SIGCHILD was caught.

. Table 3.17. wait3/wait4 Error Messages

Waiting on Processes

- Both functions accept option flags to indicate whether or not they should block and/or report on stopped child processes.

(Table 3.16.)

Useful functions in stdlib

system()

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

- passes the command to the host environment to be executed and returns after the command has been completed.

- Similar to execxx

- Show example

system()

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

- passes the command to the host environment to be executed and returns after the command has been completed.

- Similar to execxx

- Show example

getenv()

char *getenv(const char *name)

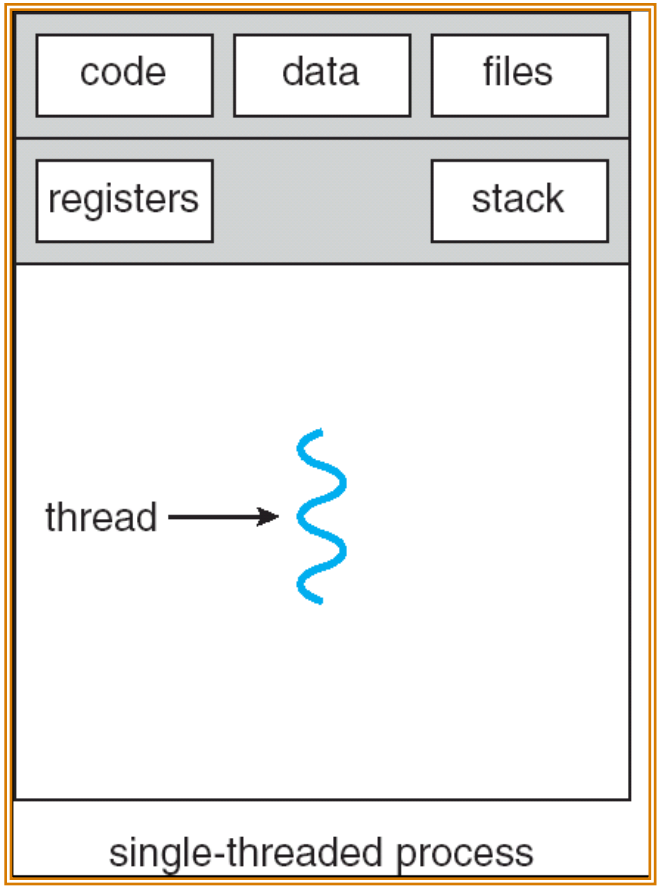
Obtain the value of the system environment variable .

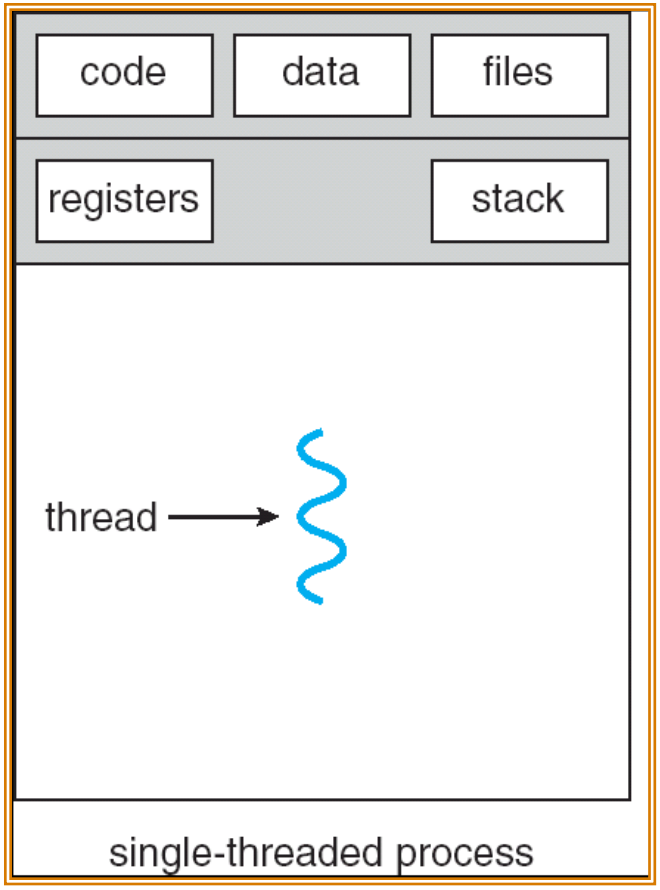
atoi()

int atoi(const char *str)

- converts the string argument **str** to an integer
- If no valid conversion could be performed, it returns zero.

A Process & Processes





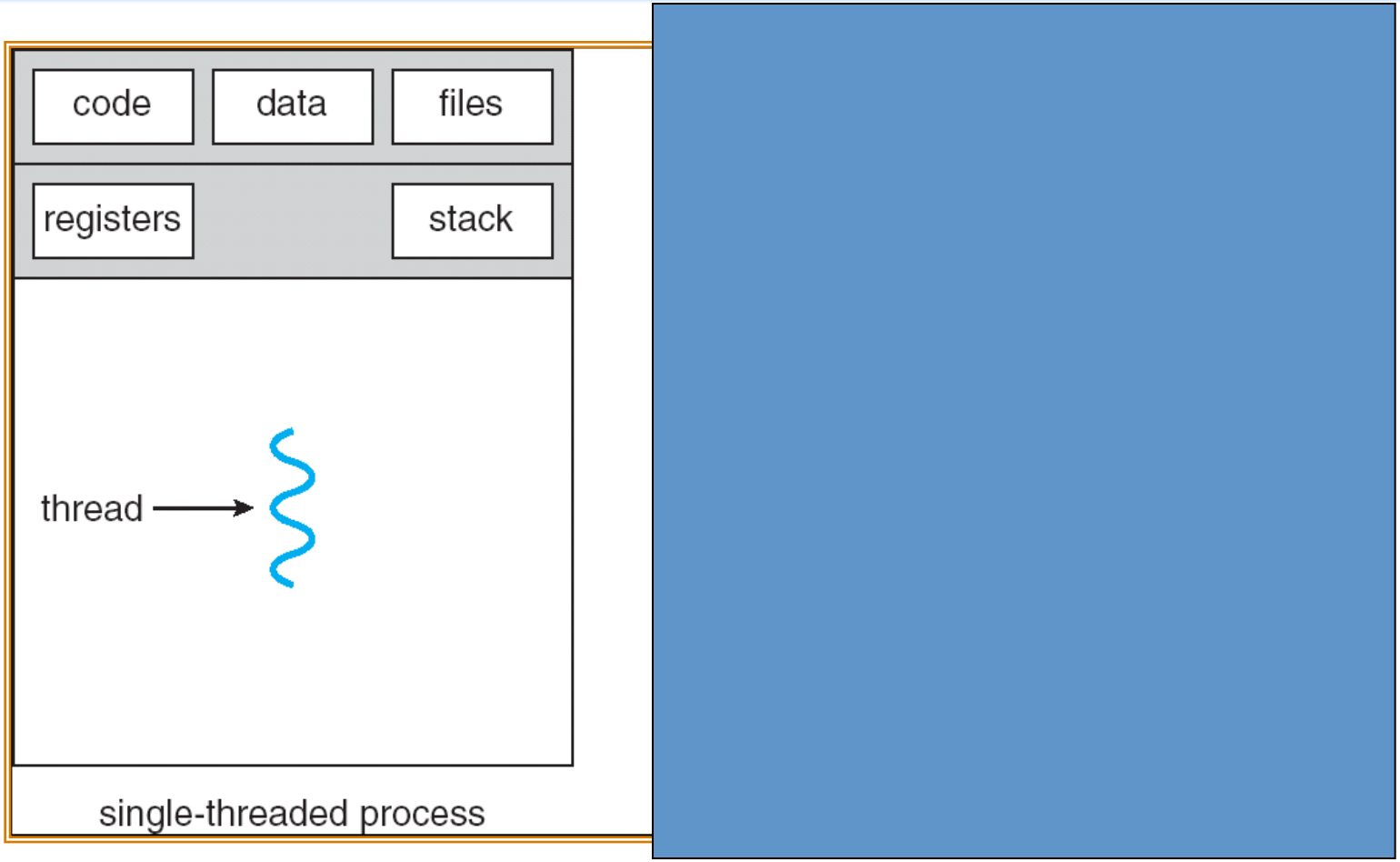
Threads

- A Light weight process where multi threads share code, data and files

Process vs Thread

- A process is a program in execution
- Thread is a light weight process.
- In a situation, when we'd like to run multiple copies of a single application or multiple subtasks, multithread provides advantages over multiple processes

Single and Multithreaded Processes



Benefits of MT

- Responsiveness
- Resource Sharing
- Utilization of MP & Multicore Architectures

Types

- User-level thread

- Kernel-Level Thread

User Threads

- Thread management done by user-level threads library

- Three primary thread libraries:
  - **POSIX Pthreads**
  - Win32 threads
  - Java threads

Kernel Threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

How user-level threads are mapped to kernel ones.

- Many-to-One
- One-to-One
- Many-to-Many

Pthreads

- A user level thread
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

pthread

- All threads within a process share the same address space.
- Threads in the same process share:
  - Process instructions
  - Most data
  - open files
  - signals and signal handlers
  - current working directory
  - User and group id

pthread

- Each thread has a unique:
  - Thread ID
  - set of registers, stack pointer
  - stack for local variables, return addresses
  - signal mask
  - priority
  - Return value: errno
- pthread functions return "0" if ok.
- A thread does not maintain a list of created threads, nor does it know the thread that created it.

Basics

- Thread operations
  - thread creation
  - termination,
  - synchronization (joins, blocking), scheduling,
  - data management
  - process interaction.

Thread creation

- Create an independent thread

```
#include <pthread.h>
```

```
int pthread_create(pthread_t * thread, pthread_attr_t
  * attr, void * (*start_routine)(void *), void * arg);
```

Hello world pthread

pt — emacs — 83×27

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *print_message_function( void *ptr );

main()
{
  pthread_t thread1, thread2;
  char *message1 = "Hello from Thread 1";
  char *message2 = "Hello from Thread 2";
  int  iret1, iret2;

 /* Create independant threads each of which will execute function */
  iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1\
);
  iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2\
);

  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);

  printf("Thread 1 returns: %d\n",iret1);
  printf("Thread 2 returns: %d\n",iret2);
  exit(0);
}
-uu-:**-F1  pt1.c          Top L13    (C/l Abbrev)----------------------------------
```

Simple synchronization

- Normally, a master waits till the created thread finishes before the main thread continues

int pthread_join(pthread_t *th*, **void** ***thread_return*);

– suspends the execution of the calling thread until the thread identified by *th* terminates

`See example pt1.c in ~box/directory`

- Compile:
  - gcc -lpthread pt1.c
    or
  - g++ -lpthread pt1.c

```
[box@oscar box]$ gcc -lpthread pt1.c
[box@oscar box]$ ./a.out
Thread 1
Thread 2
Thread 1 returns: 0
Thread 2 returns: 0
```

- [

show

- gcc file.c -lpthread

File & I/O Management

fopen()

- ***NAME***
- fopen - open a stream
- ***SYNOPSIS***
- #include <stdio.h>

  FILE *fopen(const char *restrict *pathname*, const char *restrict *mode*);
- ***DESCRIPTION***
- The *fopen*() function shall open the file whose pathname is the string pointed to by *pathname*, and associates a stream with it.
- The *mode* argument points to a string. If the string is one of the following, the file shall be opened in the indicated mode. Otherwise, the behavior is undefined.

mode

- *r* or *rb* - Open file for reading.
- *w* or *wb* - Truncate to zero length or create file for writing.
- *a* or *ab* - Append; open or create file for writing at end-of-file.
- *r*+ or *rb*+ or *r*+*b* - Open file for update (reading and writing).
- *w*+ or *wb*+ or *w*+*b* - Truncate to zero length or create file for update.
- *a*+ or *ab*+ or *a*+*b* - Append; open or create file for update, writing at end-of-file.
-

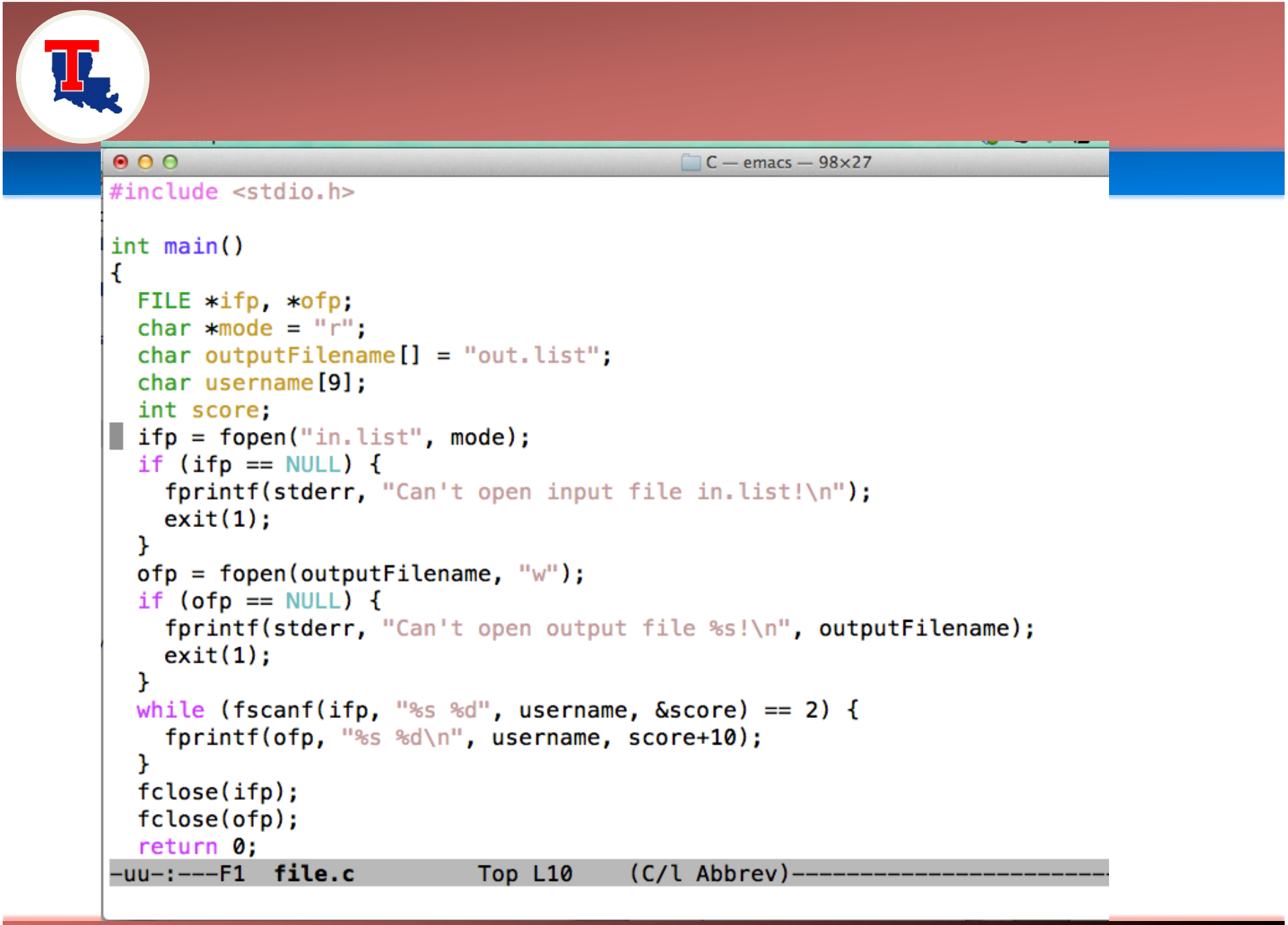
fscanf, scanf, sscanf

- ***NAME***
- fscanf, scanf, sscanf - convert formatted input
- ***SYNOPSIS***
- #include <stdio.h>

  int fscanf(FILE *restrict *stream*, const char *restrict *format*, ...);
  int scanf(const char *restrict *format*, ...);
  int sscanf(const char *restrict *s*, const char *restrict *format*, ...);

fprintf, printf, , sprintf

- ***NAME***
- fprintf, printf, , sprintf - print formatted output
- ***SYNOPSIS***
- #include <stdio.h>
  int fprintf(FILE *restrict *stream*, const char *restrict *format*, ...);
  int printf(const char *restrict *format*, ...);
  int sprintf(char *restrict *s*, const char *restrict *format*, ...);


C — emacs — 98×27
#include <stdio.h>
int main()
{
FILE *ifp, *ofp;
char *mode = "r";
char outputFilename[] = "out.list";
char username[9];
int score;
ifp = fopen("in.list", mode);
if (ifp == NULL) {
fprintf(stderr, "Can't open input file in.list!\n");
exit(1);
}
ofp = fopen(outputFilename, "w");
if (ofp == NULL) {
fprintf(stderr, "Can't open output file %s!\n", outputFilename);
exit(1);
}
while (fscanf(ifp, "%s %d", username, &score) == 2) {
fprintf(ofp, "%s %d\n", username, score+10);
}
fclose(ifp);
fclose(ofp);
return 0;
-uu-:---F1 file.c Top L10 (C/l Abbrev)

freopen

- ***NAME***
- freopen()
- Description
- The C library function **FILE *freopen(const char *filename, const char *mode, FILE *stream)**associates a new **filename** with the given open stream and same time closing the old file in stream.
-

Sample code

C — emacs — 98×27

```c
#include <stdio.h>

int main ()
{
  FILE *fp;

  printf("This text is redirected to stdout\n");

  fp = freopen("file.txt", "w+", stdout);

  printf("This text is redirected to file.txt\n");

  fclose(fp);

  return(0);
}
```

-uuu:---F1 **freopen.c**   All L16   (C/l Abbrev)----------

getcwd()

- ***NAME***
- getcwd - get the pathname of the current working directory
- ***SYNOPSIS***
- #include <unistd.h>

  char *getcwd(char **buf*, size_t *size*);
- ***DESCRIPTION***
- The *getcwd*() function shall place an absolute pathname of the current working directory in the array pointed to by *buf*, and return *buf*. The pathname shall contain no components that are dot or dot-dot, or are symbolic links.

chdir

- ***NAME***
- chdir - change working directory
- ***SYNOPSIS***
- #include <unistd.h>

  int chdir(const char *_path_);
- ***DESCRIPTION***
- The *chdir*() function shall cause the directory named by the pathname pointed to by the *path* argument to become the current working directory; that is, the starting point for path searches for pathnames not beginning with '/'.

Sample code

C — emacs — 98×25

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

main()
{
  char cwd[1024];

  if (chdir("/tmp") != 0)
    perror("chdir() error()");
  else
    {
      if (getcwd(cwd, sizeof(cwd)) == NULL)
        perror("getcwd() error");
      else
        printf("current working directory is: %s\n", cwd);
    }
}
```

```
-uuu:---F1  getcwd.c       All L17    (C/l Abbrev)----------------------
```

Original slides were excerpted from

1. **Inter-process Communications in Linux: The Nooks & Crannies, by John Shapley Gray**
2. System calls by B.Ramamurthy
3. http://www.csie.ncu.edu.tw/~hsufh/COURSES/SUMMER2013/linuxLecture_system_call.ppt