Systems Programming

- UNIX System Calls: C functions that provide access to the
- File Management (P.16, 12.1)
- Process Management (P.16, 12.2)
- Error Handling (P.16, 12.3)

* File, system, processes, and error handling.
system call error. (Error 0 is displayed if no error)
display str followed by a description of the last

/* standard C function int stdio.h

void perror(char* str) function perror() describes the system-call error

Global variable errno (holds the numeric code of the last system-call

- This value does not tell much about the cause of the failure.
- System call returns a value of -1 when it fails.
- ex. open a file may fail because file does not exist!
- Most system calls are capable of failing in some way.

Error Handling:
{ /* Error: set error variable
   * Error: Display previous error num (21)
   * Error: Display after successful call
   */
   int fd = open("nonexistent.txt", O_RDONLY | O_CREAT, 0644);
   /* Execute a successful system call */
   { /* Error: set error variable
       * Error: Print
       */
       printf("\%n", errno);
       } (fd != -1) (if (fd != -1) { /* Error occurred */
       return -1; /* An error occurred */
       } /* Open a non-existent file to cause an error */
       int fd;
       } ()
# include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
close(fd); /* close file */

seek(fd, ...); /* seek within file */

write(fd, ...); /* write to file */

read(fd, ...); /* read from file */

fcntl(fd, ...); /* set IO flags if necessary */

if (fd == -1) /* deal with error */

fd = open(filename, ...);

int fd; /* file descriptor 0: std. in, 1: std. out, 2: std. err*/

Typical sequence:

System Calls: open, fcntl, read, write, lseek, unlink, close

FILE MANAGEMENT:
Flags.

System calls open() and fcntl() both allow you to manipulate these

others

or title or not

The inductive title output to title should be appended to end

or not when process execs

The inductive title the title descriptor should be closed

* Changes on read/write/seek

* Title pointer (stores offset within title)

* Each descriptor has its own private set of properties

- Can open title several times (with different descriptors)
id = open (filename, O_RDONLY);
ex. tmpfd = open (tmpname, O_CREAT | O_RDWR, 0600);

- permissions: supplied only when file is created (ex. 0600 octal)
  O_TRUNC: if file exists it is truncated to length 0
  O_NONBLOCK: for named pipes

 _flags: O_EXCL, O_CREAT, O_APPEND
  mode: privilege or flags or flags together with zero or more
  file: absolute or relative path name
- allows you to open an existing file or create a new file for I/O

    if (open (filename, int mode [, int permissions])
      Open file
ex. charsread = read(td, buffer, BUFFER_SIZE);

It charsread == -1 fasterror()
It charsread == 0 break;
It charsread

- returns -1 if unsuccessful
- copy after end of title
- returns number of bytes copied (returns 0 if it attempts to
is then updated accordingly
- the bytes are read from current position (file pointer) which
into buffer
- copies up to count bytes from the title referenced by td
- typedet untyped int size-t;
- typedet int szize-t;
- szize-t read(int rd, void *buf, szize-t count)

Reading from regular title: read()
{
    if (charWritten > charsRead) error();
    charWritten = write(tmpBuf, buff->buf, BUFFERSIZE);
}

ex. If (stdinInput) {

    returns -1 if unsuccessful
    (should check this return value)
    Returns number of bytes copied (should count bytes copied)
    Write
    If 0 -APPEND is set, the pointer is set to end of file before each
    The bytes are written into current position (the pointer)
    When the bytes referenced by the
    Title referenced by the
    The bytes copied up to count bytes from the buffer but into the
    size = write(stdinFd, void *buf, size = count)
    Writing to regular file: write()
charsread = read(fd, buffer, insstart[i+1] - insstart[i]);

exit seek(fd, insstart[i], SEEK-SET);

- returns -1 if unsuccessful
- returns new file position if successful
- seeks fails if moved before start of file

mode = SEEK-END : offset relative to end of file
mode = SEEK-CHR : offset relative to current position of file
mode = SEEK-SET : offset relative to start of file
- mode determines how offset is to be used.
- changes file pointer
- updates long offset

off-t seek (int fd, off-t offset, int mode)

moving file pointer: seek()
see next example.

- If you move past the end of title and write there,

\[ \text{currentOffset = lseek(fd, 0, SEEK_CUR)} \]

- To find out current position use:

unix automatically extends the size of the title and

\( (0) \)

\( \text{ treats intermediate area as NULLS } \)

unix does not allocate disk area for intermediate space!
```c
22 -rw------- 1 raz 0 0
118 -rw------- 1 raz 0
0 -rw-r--r-- 1 raz

% Is - Is 'text
% parse

{ }

choice (fd, "file", ");
write (fd, "title", ");
write (fd, "0"/"", 1)
for (t = 1; t <= 6000; t++)
write (fd, "normal", 6)
fd = open ("normal.txt", O_CREAT | O_RDONLY, 0600);
/* Create a normal file */

choice (fd, "file", ");
write (fd, "title", ");
seek (fd, 6000, SEEK_SET);
write (fd, "parse", 6)
fd = open ("parse.txt", O_CREAT | O_RDONLY, 0600);
/* Create a parse file */

main

/*****************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```
Returns 0 if success; -1 if failure.

Yourself.

are automatically closed; still it is a good idea to close
it.

- Freed; releases resources when a process terminates, all

frees td; releases resources, when a process terminates, all

frees
if (standardInput unlinK(tempName))

ex.

if tempName is the last hard link, temp resources are deallocated.
removes the hard link from the tempName to its title;

int unlinK(const char *tempName)

Delete a title: unlinK()
System calls used: close(), untim()  
Step 5: close Title; remove it if temp. Title; (pass2)

System calls used: Isseek() 
(pass2, processline, reverseLine)  
To std. output; reverse the line if -c option 
Step 4: Read input title again, put backwread, copying each line 

System calls used: read(), write() 
each chunk into temp. Title, (pass2, tracklines)  
each line in an array. If reading from std. input, write 
Step 3: Read from Title in chunks storing starting offsets of

System calls used: open() 
store input; otherwise open Title for input (pass2)  
Step 2: If reading from standard input, create temp. Title to

System calls used: none 
Step 1: Parse command line (parsesArguments, processArguments)

Algorithm: 
if Title is missing, standard input is used 
the -c option reverses each line; 
reverse all lines in Title 
% reverse -c Title
```c
void receiveLine (char* buffer, int size);
void processLine (int i);
void pass ()
void setState (char* buffer, int characters);
void processComponents (char* str);
void parseCommandline (char* argv[])
*/
*/
#define MAX_LINES 100000
#define NAME_SIZE 12
#define BUFFER_SIZE 4096
*/
*/
enum {STIDIN, STSTDOUT, STDERR};
enum {FALSE, TRUE};
*/
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
*/
*/
#include <ctype.h>
*/
*/
#include <errno.h>
*/
*/
#include <fcntl.h>
*/
*/
#include <unistd.h>
*/
*/
#include <sys/types.h>
*/
*/
#include <malloc.h>
*/
*/
#include <errno.h>
*/
*/
#include <limits.h>
*/
*/
#include <assert.h>
*/
*/
#include <ctype.h>
*/
*/
#include <string.h>
*/
*/
#include <time.h>
*/
* /
*/
#include <sys/time.h>
*/
*/
#include <sys/resource.h>
*/
*/
#include <malloc.h>
*/
*/
#include <stdio.h>
*/
*/
#include <string.h>
*/
*/
#include <ctype.h>
*/
*/
#include <unistd.h>
*/
*/
#include <fcntl.h>
*/
*/
#include <errno.h>
*/
*/
#include <limits.h>
*/
*/
#include <assert.h>
*/
*/
#include <ctype.h>
*/
*/
#include <string.h>
*/
*/
#include <unistd.h>
*/
*/
#include <fcntl.h>
*/
*/
#include <errno.h>
*/
*/
#include <limits.h>
*/
*/
#include <assert.h>
*/
*/
#include <ctype.h>
*/
*/
#include <string.h>
*/
*/
#include <unistd.h>
*/
*/
#include <fcntl.h>
*/
*/
#include <errno.h>
*/
*/
#include <limits.h>
*/
*/
#include <assert.h>
*/
*/
```


/*******************************************************************************/

{
    /* done */
    return EXIT_SUCCESS (0)
}

/* parse second pass through input */
parse ();
/* parse first pass through input */
parseCommandLine (argc, argv);
parseCommandLine (argc, argv);

int main (int argc, char *argv[])

/*******************************************************************************/

/* file descriptor of input */
tint tf;
/* current position in input */
tint tifstream = 0;
/* store offsets of each line */
tint tifstream [MAX_LINES];
/* total number of lines in input */
tint tifstream = 0;
/* set to true if read line */
/* set to true if option is used */
char tifstream [NAME_SIZE];
/* set to true if name is NULL */
/* globs */
/* */


```c
{  
  //standardinput = (filename == NULL)  
  {  
    /* User error */  
    else  
    if (filename != NULL)  
      processoptions (argv[i])  
      if (argv[i] == [0][i])  
        for (i = 1; i < argc; i++)  
          if (argv[i]  
            /* Parse command line arguments */  
          }  
```
```c
{  
    exit EXIT_FAILURE;  
}  

/* print (stderr, "usage: reverse -c [filename]""); */
void usage() {
}

/*****************************/

{  
    {  
        break;  
        } /* usageerror */
    default:  
        break;  
    charopt = TRUE;  
    case ".":  
        */ print on command line flag */
        switch (str) {  
        case "":  
            /for (i = 1; str[i] != NULL; i++) 
            the if:  
                /
                /parse options */
        }
}

/*****************************/
/** offset of first line */

{ 

  if (fio == -1) {
    fio = open (filename, O_RDONLY);
  } /* Open named file for reading */

  } /* Open named file for reading */

} /* Open named file for reading */

{ if (tmpfd == -1) (err = open (tempname, O_CREAT | O_WRONLY, 0600)); /* create temporary file to store copy of input */

  if (tmpfd = open (tempname, O_RDONLY, 0600)) {
    sprintf (tempname, "rev.%d", getpid ()); /* Use random name */
    ifd = stdin;
  } /* Use random name */

} /* Use random name */

if (stdin) { /* Read from standard input */

    if (buffer [BUFFER_SIZE] > input [buffer]) {
      printf ("Warning: character overflow!");
      return 1; /* Character overflown */
    } /* Character overflown */

} /* Character overflown */

/* ******************************************/
{ if (standardinput)
   td = tmp;
   /* If reading from standard input, prepare td for pass */
 }
int offset = jcount + i]
   /* Store offset of trailing line, if present */
 {
    if (charwritten != charread) { charwritten = charread; }
    if (standardinptr) { /* Copy line to temporary file */
      if (charwritten != charread) { /* Process line */
        if (charread == 0) break; /* EOF */
        if (charread == 1) { /* Error */
          printf ("%d", charread - buffer-size);
        } /* Fill buffer */
      } /* Read all input */
  } /* Read all input */
```c
{ 
    /* Remove temp title */
    close (fd);
    /* Close input file */

    processLine (t);
    for (i =lineno - 1; i >= 0; i--)
    { /* Scan input title again, displaying lines in reverse order */
        printf ("");
    }
}

void process(void)
{ 
    char [] buffer = "";

    for (i = 0; i < characters; i++)
    { /* Store offsets of each line start in buffer */
        if (buffer [i] == \n)
            lineno = filloffset;
        lineno ++; /* Update current title position */
    } (for)
}

void tracklines (char * buffer, int characters)
{ /* ******************************************** */

    /********** *******************************************/

} /* ******************************************** */
```
/**************************************************************************/

{ 
  /* Write (i, buffer, charStart)*/
  char charStart=i;  
  char charEnd = read (fd, buffer, i);   
  fseek (fd, i, SEEK_SET); /* Find the line and read it */
  char buffer[BUFSIZ];
  int charStart, charEnd;
  /* Read a line and dispay it */
  void processLine (int i) {

}
{  
    exit (1);  
    /* describe error */  
} ()

void factorizer()

=localhost

{  
    /* end */  
    /* increment start index */  
    *start = *tmp;  
    *tmp = buffer[end];  
    end = start;  
    while (start > end) {  
        buffer[start] = buffer[end];  
        buffer[end] = buffer[start];  
        *start++;  
        *end--;  
        tmp = *start;  
        *end = tmp;  
    }  
    if (buffer[end] == \"\" || --end; --end; --end;  
        Leave trailing newline  
        char tmp;  
    }  
    if (start == size - 1)  
        reverse all the characters in the buffer  
    }  
    void reverse(char* buffer, int size)
The [-t delay] option.

The default delay between scans is 10 seconds which can be overridden by
option (-t count) in which case it scans only count times (0 count)
By default, monitor will scan forever unless overridden by [-t count]

DELETED: indicates that the title was deleted since the last scan
NEW: indicates that the title was newly created since the last scan
MODIFIED: indicates that the title was modified since the last scan

Title modification is indicated as follows:

If title is a directory, all of the titles inside the directory
are scanned.

About any of the titles that were modified since the last scan.

Scans all specified titles every delay seconds and displays information

getentent(title) : obtains directory entries
stat() : similar to stat
getstatusinfo(title) : obtains status information about a title

Uses 3 new advanced system calls:
Second example: monitor.c
but was present in previous scan, it is marked deleted
At the end of a scan, if a title is not present in current scan
MODIFIED message
If the title is already in scan table and has been modified, then
- If the title is not in the scan table, it is added
- It calls geteents() to scan directories
- It uses stat() to get information (type and last modification time)
- It uses stat() to get information (type and last modification time)
- Monitor program continuously scans the specified titles/directories

*/
/** * Seconds between loops */
/* Number of times to loop */
/* Count of titles on command line */
/* One per matching title */
/char *kimnames [MAX-FILES]; */
/* One per title on command line */
/* globs */
/** */
*/
*/ Information from start */
*/ To detect changes */
char *kimname[MAX-FILES];
*/
*/ Algorithm + data structure:
*/
S-ISRIO(mode) returns true if title is a pipe
S-ISREG(mode) returns true if title is a regular file
S-ISBLK(mode) returns true if title is a block special device
S-ISOCHR(mode) returns true if title is a character special device
S-ISDIR(mode) returns true if title is a directory

Macros:
stat(time the last status-change time
stat(time the last modification time
stat(time the last access time
stat(size the file size
stat gid the group ID
stat uid the user ID
stat inodes the hard-link count
stat mode the permission flags
stat type the file type number
stat dev the device number

and includes the following fields:
The stat structure is defined in /usr/include/sys/stat.h

stat(title) fill the buffer with information about title name
int stat(char *name, struct stat *buf)

system call:
the decode of time is done using localtime() and strftime()

is called to display title's new status
the title's status entity. If status has changed, update entity()
special title, it calls update entities() which either adds or updates
if the title is a regular title, character special title or block

each of the entries in the directory
process directory function applies monitored() recursively to

if (S_ISDIR(statbuf.st_mode))

exe. result = stat(title name, &statbuf�
default : same as stat() except it takes title descriptor as first
symbolic link instead rather than the title it refers to
insert() : same as stat() except it returns information about the
advance to the next directory entry
processDirectory function skips . and .. and uses seek to

"-1 if error
0 if last directory entry has already been read
returns length of the directory entry if successful

d-name : the filename
d-rlength : length of the directory string
d-offset : the offset of the next directory entry

The directory is stored in /usr/include/sys/dirent.h and contains
The dirent struct is defined in /usr/include/sys/dirent.h
and contains
position and dirent struct pointed to by but with the next entry.
reads the directory file with descriptor fd from the current

int getdirectory(int fd, struct dirent*, int * structures)

Reading Directory Information: getdirectory()
#include <time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <stdio.h>

monitor.c
#define DEFAULT_DELAYTIME 10
#define DEFAULT_LOOP_COUNT 1
#define NOT_FOUND

#define MAX_FILENAME 50
#define MAX_FILES 100

/* globals */

struct status
{ /* Information from sensor */
    int la...
/**************************************************************************


    This file is part of the OpenJDK source tree.

    Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions
    are met:

    1. Redistributions of source code must retain the above copyright
       notice, this list of conditions and the following disclaimer.

    2. Redistributions in binary form must reproduce the above copyright
       notice, this list of conditions and the following disclaimer in
       the documentation and/or other materials provided with the
       distribution.

    3. All advertising materials mentioning features or use of this
       software must display the following acknowledgment:
       "This product includes software developed by Sun Microsystems,
       Inc."." AWARD"

    4. Neither the name of Sun Microsystems, Inc. nor the names of its
       contributors may be used to endorse or promote products
       derived from this software without specific prior written
       permission.

    THIS SOFTWARE IS PROVIDED ''AS IS'' AND ANY EXPRESSED OR IMPLIED
    WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
    OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
    DISCLAIMED.  IN NO EVENT SHALL SUN MICROSYSTEMS, INC. OR ITS
    CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
    CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
    ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

    Original authors:  David S. Hanek, James McCauley, Peter G. McDermid,
                      Charles E. Ralston, and Michael Sullivan

    Modifications:  The original Berkeley portions of this software were
                    developed by David S. Hanek, James McCauley, and
                    Charles E. Ralston.

    Copyright 1983 Sun Microsystems, Inc.

**************************************************************************/

{ if (ttitlecount == 0) { ;
    ttitlecount++;
    processparameters (area); ;
    if (area) { ;
        for (i = 1; i < MAX_PARAMETERS + 1; i++) { ;
            parsecommand (int area, char *argv);
        }
    }
} /* * Parse command line arguments */

{ /* * Return EXIT_SUCCESS */
    /* * Execute main monitor loop */
    /* * Parse command line */
    parsecommand (area, argv);
    main (int argc, char *argv); }
}
{  
  return (number);  
  
  /* There must be a number */  
  {  
    
    int digits();  
    int digits = 0;  
    {  
      int digits = number * [0, 10 + str] + [1];  
      }  
  }  
  
  /* Count the digits in the number */  
  int getNumber(char* str)  
  {  
    int digits = 0;  
    {  
      int digits = number * [0, 10 + str] + [1];  
      }  
      }  
  
  /* Convert a numeric ASCII option to a number */  
  int getNumber(char* str, char* str);  
  
  for (i = 0; i < str; i++)  
    {  
      int i;  
      char* str;  
      }  
  
  /* Parse options */  
  void processOptions(char* str)  
  {  
    }
{ }
{ }

\{ 

\}

\}

\{ 

\}

\{ 

\}

\{ 

\}
updatestat (title, filename); /* Update title
else if (S-IRQ (mode) || S-ISTR (mode) || S-ISBLK (mode))
processdirectory (filename);
/* Process directory */

mode = statbuf.fsmode; /* Mode of file

{ /* Return
  if (result) { cannot start, \n"filename"
  } /* Status was not available
  */
  result = stat (title, filename, &statbuf); /* Obtain file status

  if (result)
    mode = mode;
  status = statbuf;
  } /* Process a single file/directory */
}
void monitorevent (char *filename)
```c
{
    /* Close directory */
    close (dp);

    /* Call recursively */
    montoratle (filename)
    splitr (filename, "/", &name, dep->name);
} /* ... */
/* . . . */
/* . . . */
if (strcmp (dep->name, "") == 0) { skip. and
    _while (dep = readdir (dp)) { /* Read all directory entries */
        /* ... */
        if (dp == NULL) { fatalerror;
            open = opendir (dirname);
        }
        char filename [MAX_FILENAME]
        struct dirent * dep;
        /* ... */
        /* Process all files in the named directory */
        void processdirectory (char * dirname) {
```
{ return NOT-FOUND; }


cmp (statuses[i].filename, filename) == 0 { return (i); }

if (statuses[i].lastCyle &
for (i = 0; i < MAX-FILES; ++)

int i;

/* locate the index of a named filename the status array */
int findEntry (char *filename)

{ /* Update status array */

if (entryIndex == NOT-FOUND)

/* Update Entry (entryIndex, status); */

else

/* Add new entry */

entryIndex = addEntry (filename, status); /* Find existing entry */

*/

int entryIndex;

/* Add a status entry if necessary */

void updateStat (char *filename, struct status *stat); }


```
{
    return (NOT-FOUND);
}

if (states[1].lastCycle == states[1].thisCycle) return (1);

for (i = 0; i < MAX-FILES; i++)
    if (i !=)
        /* Return the next free index in the status array */
        int nextFree ()

    {
        /* Display status information */
        printf("%d", status[1].status = status[1].thisName, status[1].thisName);
        /* Add status information */
        strcpy (states[1].status = states[1].thisName, status[1].thisName);
        /* Add status information */
        if (index == NOT-FOUND) return (NOT-FOUND);
        /* Find the next free entry */
        int nextFree ();
        /* Add a new entry into the status array */
        int addEntry (char* filename, struct states* status)
```
```c
{
  /* EXIT FAILURE */
  exit(EXIT_FAILURE);
}
```
```c
{  
  astitime (struct astitime *)
  printf ("%s\n", startbuf->st-time,
           " DISPLAY a status buffer */
  void printfast (struct startbuf)
}
```
```c
{  
  printfast (states[index].status)
  printf ("%s", startbuf->time,
           " DISPLAY an entry of the status array */
  void printfast (int index)
}
```
```c
{  
  printfast (int index)
  printf ("%s", startbuf->time,
           " CHANGED */
  void printfast (int index)
  if (states[index].status = startbuf->time,
     " DISPLAY information if the title has been modified */
  } /* DISPLAY information if the title has been modified */
  } /* DISPLAY information if the title has been modified */
```
```c
```
62 is the GROUPID for GROUP name (see /etc/group for GROUP IDs)

{  
  if (flag == 1) perror("error changing GROUP ID for test.txt");
    flag = chown("test.txt", -1, 62);
    if (flag)
      int main()
  }

the ID should not change

ownerID and GROUPID respectively (a value of -1 indicates that

These cause the owner and GROUP IDs of filename to be set to

int chown (int fd, uid_t ownerID, gid_t GrouppID)
int chown (const char* filename, uid_t ownerID, gid_t GrouppID)
int chown (const char* filename, gid_t GrouppID)

Miscellaneous File Management System Calls
{ if (flag == -1) {
    flag = chmod("test.txt", 0600);
    if (flag)
        int flag;
}  

int main()
{
    set_uid and set GID flags have values 0400 and 0200 respectively
    (ex. 0600)

    these change the mode of fileName to mode (specified as octal);

    int Fchmod (int fd, int mode)
    int chmod (const char *fileName, int mode)
and a value of -1 otherwise.

They both return the index of the new title descriptor if successful.

In both cases, the original and copied title descriptors share
the same title pointer and access mode.

dup2() chooses newpd if it is currently active and points it to
to the same title which oldpd points to.
dup() finds the smallest title descriptor entry and points it
to dup2 (int oldpd, int newpd).

int dup2 (int oldpd)
{ int dup (int oldpd);
    dup2 (int oldpd);}

duplicate a title descriptor: dup() and dup2()
{  
  write("2", "n", 2);
  dup2(3, 2);
  write("doc", 4);
  printf("%d\n", tds);
  tds = dup(tfd);
  close(0);
  write("up", 3);
  printf("%d\n", tds);
  tds = dup(tfd);
  write("what's", 6);
  printf("%d\n", tfd1);
  tfd1 = open("test.txt", O_RDONLY | O_TRUNC | O_CREAT);
  int tfd1, tfd2, tds;
} main()
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
ex. Fcntl(td, F-SETPF, O-WSYN | O-APFLD);

title-status flags and mode

etc. See P 47 of text

cmd = F-GETPF returns a number corresponding to the current

title-status flags to are

cmd = F-SETPF sets the current title-status flags to are

with descriptor td; arg is an optional argument for cmd.

performs the operation encoded in cmd on the title associated

(int fcntl (int td, int cmd, int arg)

Ftitle descriptor operations:
Responsible for user logins.

Initially, init duplicates (forks) several times and each child process

process; so init is the ancestor of all subsequent processes.
The only way to create a new process is to duplicate an existing

with process ID = 1.

When UNIX starts (boots), there is only one process, called init,'
prompt to accept the next command.
and the parent process presents the user with the shell
the parent process is informed (using a signal)
‘When the child process terminates using exit()’,
– using wait()
– The parent process waits for the child process to terminate
– utility using exec() (differently)
The child shell process replaces its code with that of the
– The shell first duplicates itself using fork()

Consider the way a shell process executes a utility in the foreground:

| It is very common for a parent process to suspend itself until |
| one of its child processes terminates. For example: |
child process is not created.

If fork() fails, it returns a -1 to the parent process and the
and the PID of the child process to the parent.

If fork() succeeds, it returns a value of 0 to the child process

process IDs (pid) and parent
tables. The only difference is in the process IDs (pid) and parent
parent’s code, data, stack, open file descriptors, and signals
a duplicate of the parent process; it inherits a copy of its
causes a process to duplicate. The child process is almost exactly

fork(), getpid(), getppid(), wait(), exit(), exec()

System calls:
They always succeed. (The PID value for process with ID=1 is 1.)

Return the process' ID and parent process ID respectively.

```
pid.t ETcpd(pid) (pid)
pid.t ETcpd(pid) (pid)
```
PID 640 terminates.
I'm the child process with PID 640 and Ppid 1.
PID 639 terminates.
MY child's PID is 640.
I'm the parent process with PID 639 and Ppid 446.
I'm the original process with PID 639 and Ppid 446.

`$`  

```c
#include <stdio.h>

int main()
```

```c
{  
  
  /* Both processes execute this */  
  getpid () ;  
  }  

  if ( pid == 0 )  
  {  
    fork ( ) ;  
  }  
  else  
  {  
    parent terminate with PID %d and Ppid %d.  
  }  
```

```c
/* This is the parent, so I must be the child */  
```
/* Both processes execute this */

if (getpid () != 0) {
    printf ("Child process with PID %d and PPID %d.\n", getpid (), getpid ());
}
else {
    printf ("My child's PID is %d.\n", getpid ());
}
*/

if (pid = fork ()) {
    if (pid = 0) { /* Branch based on return value from fork */
        printf ("In the original process with PID %d and PPID %d.\n",
                 getpid (), getpid ());
        printf ("the child process is automatically adopted by the init process.\n"
                 Orphan Process: If a parent process terminates before its child terminates,"
exit() never returns just always accepts its children's termination code.

adopted by just by setting their PID's to 1.

The kernel makes sure that all of a terminating process' children are

a parent accepts a child's termination code by executing wait()

return code is called a zombie process.

a process which is waiting for its parent process to accept this

8 bits of status are used; so value 0-255.

and waits for its termination code (status) to be accepted. (only lower

when a child process terminates, it sends its parent a SIGCHLD signal

data, and stack; then terminates the process.

closes all of a process' file descriptors, deallocates its code,

void exit(int status)

termination of a process: exit()
```c
#include <stdio.h>

int main() {
    printf("I am exiting with status code 42\n")
    return 42;
}
```
Zombie processes is a problem.

One entry in the system's fixed-size process table. Too many zombie processes do not take up system resources but do use up system resources. The child's process will remain a zombie process.

If the parent process is alive but is unmaintaining to accept the child's termination code (because it never executes wait()),

If the parent process is dead, it will adopt processes and accept its termination code.

A process that cannot leave the system (even if it has exit-ed). Zombie processes:
#include <stdio.h>

main()

{ entry (42)
    /* * Exit with a tiny number */
} else

    { sleep (1000)
        /* * Never terminate, and never execute a wait */
    } while (1)

} /* () Branch based on return value from fork */

if (pid = 0) /* Duplicate */
    /* () pid = fork */
    /* () main

#include <stdio.h>
the zombies.

already zombies’, wait() returns immediately with the status of one of
If a process executes a wait() and one or more of its children are

immediately with a value of 1.
If a process executes a wait() and has no children, wait() returns

and the last bit is set to 1 if the child core dumped.
If the rightmost byte of status is non-zero, the rightmost 7 bits

are equal to the signal number that caused the child to terminate
8 bits of the value returned by the child’s exit() or return() call

and places a status code into status that is encoded as
A successful call to wait() returns the PID of the child process that
causes a process to suspend until one of its child processes terminates.

pid = wait(NULL); /*status)

waitfor a child: wait()
PID 695 terminates
A child with PID 696 terminated with exit code 42
I'm the child process with PID 696 and PID 418
I'm the parent process with PID 695 and PID 418
I'm the parent process and my PID is 695

$ wait

  ! ( ) print "PID %d a terminate", $pid

  \{ /* * */ exit (42); /* EXIT with a sliLy number */
  \{ print ("I'm the child process with PID %d and PID %d",
      $pid, $pid%4\n      )

  \} else

  \{ /* * */
  \{ pid = fork ()
      if pid = 0
          " \{ print ("I'm a child with PID %d terminated with exit code %d\n"
              $pid, $pid%4\n
          )
    else
      " \{ print ("I'm the parent process with PID %d and PID %d",
          $pid, $pid%4\n          ) /* * */
      \} /* * */

# include <stdio.h>
int execvp(const char *path, const char *argv[])
const char *argv, NULL)
  ...,
int execvp(const char *path, const char *argv[])
const char *argv, NULL)
  ...,
int execvp(const char *path, const char *argv[])
const char *argv, NULL)
  ...

The following two use $PATH variable to find the executable:

int execvp(const char *path, const char *argv[])
const char *argv, NULL)
  ...

only the code that the process is executing changes.
When a process executes exec() its PID and Ppid stay the same.

A process may replace its current code, data, and stack with those of another executable by using one of the exec() family of system calls.

Diffrerentially a Process: exec()
A successful exec() never returns.

Starts executing the new code.

Replacing its code, data, and stack with those of the executable and replacing this code, data, and stack with those of the executing and otherwise the calling process.

If executable is not found, -1 is returned otherwise the calling process.

The command line argument for executable (argv[0]: name of executable)

argv[1]: name of executable for execvp/execvp

path: executable (relative or absolute path for execvp/execvp)
298 Jul 22 20:20 zombie.c
4174 Jul 24 12:08 zombie
2058 Jul 22 20:23 vount.c
284 Jul 22 20:22 myclip.c
226 Jul 22 20:22 background.c
187 Jul 22 20:24 arm.c

total 38

I'm processes 710 and I'm about to exec an ls -l

$ myexec

{ max
    printf ("This line should never be executed");
    if (execl ("/bin/ls", "ls", NULL, NULL) == NULL)
        printf ("I'm processes %d and I'm about to exec an ls -l", getpid ());
} main
#include <stdio.h>
chdir() returns -1 if it fails; 0 if it succeeds.

have execute permission from the directory for chdir() to succeed.

sets a process' current working directory to pathname. The process must

int chdir(const char* pathname)

the shell's current working directory.

-e. When a utility is run from a shell, the utility process inherits

A child process inherits the current working directory from its parent;

process.

Every process has a current working directory that is used when

ChangeDirectory(); chdir()
```c
#include <stdio.h>

main()
```

Smaller the priority value, faster the process will run.

Only super user can change to negative priority.
adds delta to the priority (legal values of priority -20 to +19)

(int ncie(int delta))

CHANGE THE PRIORITY USING ncie()
Child process inherits the priority of the parent process and can

CHANGE PRIORITY: ncie()
The get functions always succeed; the set methods will succeed only when executed by super user or if id equals real or effective

td of the process.

```python
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
udt.setEdtd(uid=td)
```

Getting and setting user and group ids:
The orphaned child process is adopted by init.

The orphaned process creates a child process to exec the specified executable and terminates.

The orphaned process creates a child process to exec a program in the background.

Program which uses fork() and exec() to execute a program.

$ background &

{ }

int (fork () == 0) */ CHILD[

} } /* Execute other program */

execvp (argv[1], argv[1]);

fprintf (stderr, "$\n", argv[1]);

#include <stdio.h>