8-UNIX **exec & fork** System Calls

exec Commands

- exec() = a generic name
- **execve**(*pathname, argv, envp*) (LPI p.514)

  loads a new program and environment

  into the process's memory
- **execvp**(*filename, argv[]*)
- execlp*(filename, arg, …)*
- execl(*filename, arg, …*)
- execv(*filename, argv[]*)
- execle*(pathname, arg, …)* (LPI p.567)

Note: None of the above returns on success; all return -1 on error.

The naming convention: exec*

- 'l' indicates a list arrangement (a series of null terminated arguments)
- 'v' indicate the array or vector arrangement (like the argv structure).
- 'e' indicates the programmer will construct (in the array/vector format) and pass their own environment variable list
- 'p' indicates the current PATH string should be used when the system searches for executable files.

NOTE:

In the four system calls where the PATH string is not used (execl, execv, execle, and execve) the path to the program to be executed must be fully specified.

exec System Call Functionality

Library Call Name	Argument List	Pass Current Environment Variables	Search PATH automatic?
execl	list	yes	no
execv	array	yes	no
execle	list	no	no
execve	array	no	no
execlp	list	yes	yes
execvp	array	yes	yes

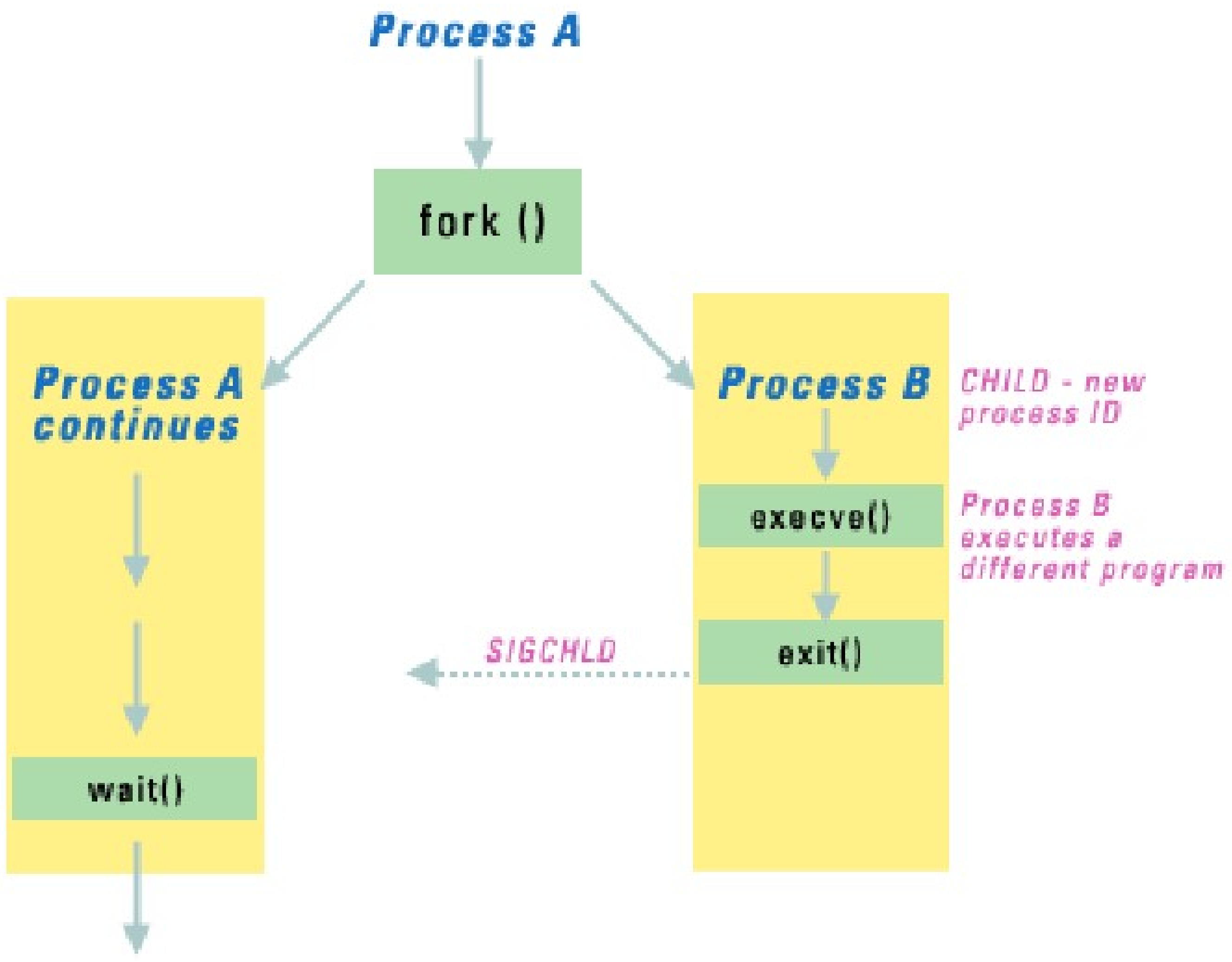
http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#exec

Things to remember about exec*:

- This system call simply replaces the current process with a new program -- the pid does not change
- The exec() is issued by the calling process and what is exec'ed is referred to as the new program -- not the new process since no new process is created
- It is important to realize that control is not passed back to the calling process unless an error occurred with the exec() call
- In the case of an error, the exec() returns a value back to the calling process
- If no error occurs, the calling process is lost

http://www.cs.uregina.ca/Links/class-info/330/Fork/fork.html#exec

fork()
as a diagram – what we ‘ve known



What does *fork()* return?

- On success, the PID of the child process is returned in the parent’s thread of execution, and a 0 is returned in the child’s thread of execution

- On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

fork() – Output

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  pid_t pid;  /* could be int */
  printf("Hello\n");
  pid = fork();
  printf("Goodbye\n");
}
```

fork1.c

Program's output:

Which will it do?

```
A) Hello
   Goodbye

B) Hello
   Goodbye
   Goodbye

C) Hello
   Goodbye
   Goodbye
   Goodbye
   ........
```

fork() – Who prints what ?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
  pid_t pid;        /* could be int */
  printf("Hello\n");
  pid = fork();
  if (pid == 0)
    printf("Child: Goodbye\n");
  else
    printf("Parent: Goodbye\n");
}
```

fork2.c

Program's Output:
Which will it do?

A)
```
Hello
Parent: Goodbye
Child: Goodbye
```

B)
```
Hello
Parent: Goodbye
```

C)
```
Hello
Child: Goodbye
```

fork() – parent waits for child ?

```
…………..
 printf("Hello\n");
 pid = fork();
 if (pid == 0) {
   printf("Child: Goodbye\n");
   sleep(2); // sleep 2 seconds
 }
 else {
   if (wait(&status) == -1)
       perror("Shell Program error");
   else
       printf("Child returned: %d\n",
             WEXITSTATUS(status));
}
```

Program's Output:

```
Hello
Child: Goodbye
Child returned: 0
```

What does the parent process do?

What does the child process do?

fork3.c

Execute a command via argument passing from argv

```
/* Program to execute a command using an argument from argv */
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[])
{
   if (argc != 2) {
      printf("Usage: input a command name with no option i.e. ls \n");
      return 0;
   }
   printf("About to run: %s \n", argv[1]);
   char *cmd[] = {argv[1], 0 }; /* make sure place a 0 at the end of
                                          array of pointer cmd */
   execvp(cmd[0], cmd);
   return 0;
}
```

argv.c

The output of argv.c

```
[bielr@sp1 ClassExamples]> argv ls
About to run: ls
argv                fork1               globvar.out   strspn_example
argv.c              fork1.c             infloop       strspn_example.c
count_chars         fork2               infloop.c     Text1.dat
count_chars.c       fork2.c             ouch          thread
count_words         fork3               ouch.c        thread.c
count_words.c       fork3.c             pause         tlpi_hdr.h
data-file           getenv_putenv       pause.c       waitpid
environ             getenv_putenv.c     shfile        waitpid.c
environ.c           globex              shfile.c      word_count.c
error_functions.h   globex.c            shfile.out    word_counts.c
[bielr@sp1 ClassExamples]>
```

Review I/O Redirection

First, review a program from C7 named *sincos*

Example Trigonometric Functions

```
// prints tables showing the values of cos,sin
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
void tabulate(double (*f)(double), double first, double last, double incr);
int main(void) {
        double final, increment, initial;
        printf (“Enter initial value: “);
        scanf (“%lf”, &initial);
        printf (“Enter final value: “);
        scanf (%lf”, &final);
        printf (“Enter increment : “);
        scanf (%lf”, &increment);
        Printf(“\n    x   cos(x) \n”
            “  ----------  -----------\n”);
        tabulate(cos, initial,final,increment);
        Printf(“\n     x    sin (x) \n”
        “  ----------  -----------\n”);
        tabulate(sin, initial,final,increment);
        return (EXIT_SUCCESS);
}
```

The **main** function in little print. Bigger print used in following slides.

Example Trigonometric Functions (1 of 4)

```
// prints tables showing the values of cos, sin

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

void tabulate(double (*f)(double), double first,
                   double last, double incr);

int main(void)
```

Example Trigonometric Functions (2 of 4)

```
int main(void)
{
        double final, increment, initial;
          // Enter the data at the keyboard
        printf (“Enter initial value: “);
        scanf (“%lf”, &initial);
        printf (“Enter final value: “);
        scanf (%lf”, &final);
        printf (“Enter increment : “);
        scanf (%lf”, &increment);
```

Example Trigonometric Functions (3 of 4)

```
      // Print the headers and call tabulate
    printf("\n    x   cos(x) \n"
           "  ----------  -----------\n");
    tabulate(cos, initial,final,increment);

    printf("\n     x    sin (x) \n"
    "  ----------  -----------\n");
    tabulate(sin, initial,final,increment);
    return (EXIT_SUCCESS);
}
```

Trigonometric Functions (4 of 4)

```
// when passed a pointer f, the function prints a table
// showing the value of f

void tabulate(double (*f) (double), double first,
                    double last, double incr)
{
    double x;
    int i, num_intervals;
    num_intervals = ceil ( (last -first) /incr );
    for (i=0; i<=num_intervals; i++) {
      x= first +i * incr;
      printf(“%10.5f %10.5f\n”, x , (*f) (x));
     }
}
```

Output of the Example

```
Enter initial value: 0
Enter final value: .5
Enter increment: .1

    X           cos(x)
 ----------    -----------
 0.00000       1.00000
 0.10000       0.99500
 0.20000       0.98007
 0.30000       0.95534
 0.40000       0.92106
 0.50000       0.87758

    X           sin(x)
 ----------    -----------
 0.00000       0.00000
 0.10000       0.09983
 0.20000       0.19867
 0.30000       0.29552
 0.40000       0.38942
 0.50000       0.47943
```

Two redirection examples:

1) ls > out.txt

2) sincos < input.txt

In input.txt:

```
5.0    - initial value
10.0 – final  value
1.0  - increment  value
```

Redirection of standard output (redir.c)

Example: implement shell to do: **ls > x.lis**

What happens **inside**:

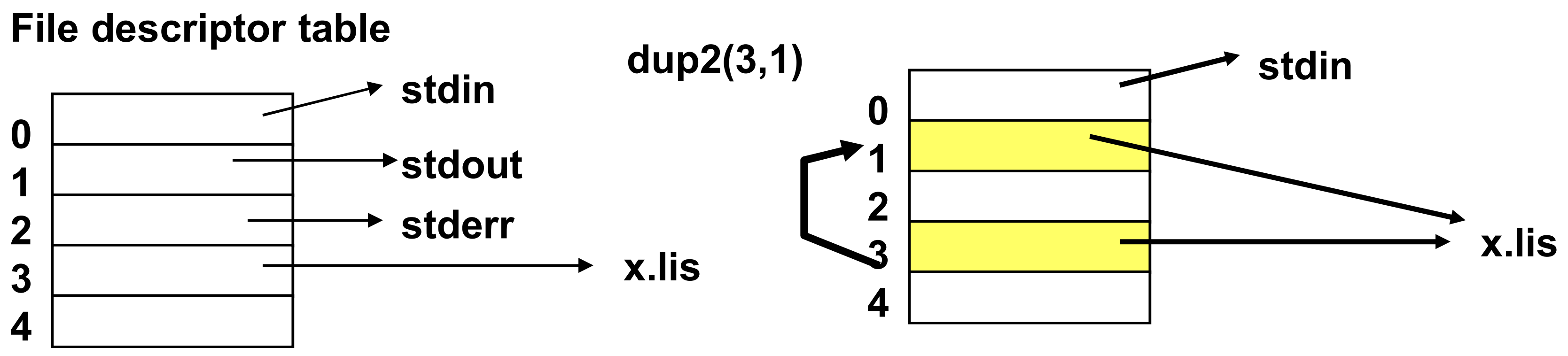
(1) Open a new file *x.lis*

(2) Redirect standard output to *x.lis* using **dup2** command

Everything sent to standard output ends in *x.lis*

(3) execute ls in the process

dup2(int fin, int fout) - copies fin to **fout** in the file table



Note about the next program

It uses the CREAT call…to create a file that does not exist.

It is becoming antiquated.

```
fileID = creat( "x.lis",0640 );    // 0640 is the mode
```

To do the call with OPEN, we would type:

```
fileID = open ( "x.lis", O_WRONLY | O_CREAT | O_TRUNC,
                        S_IRUSR | S_IWUSR | S_IRGRP);
```

0640 = 0400 (User_read) + 0200 (User-write) + 040 (Group-read)

The mode for file permission bits is explained in a chart on

LPI page 295.

Constants for file permission bits

Constant	Octal value	Permission bit
S_ISUID	04000	Set-user-ID
S_ISGID	02000	Set-group-ID
S_ISVTX	01000	Sticky
S_IRUSR	0400	User-read
S_IWUSR	0200	User-write
S_IXUSR	0100	User-execute
S_IRGRP	040	Group-read
S_IWGRP	020	Group-write
S_IXGRP	010	Group-execute
S_IROTH	04	Other-read
S_IWOTH	02	Other-write
S_IXOTH	01	Other-execute

Code Example - implement ls > x.lis

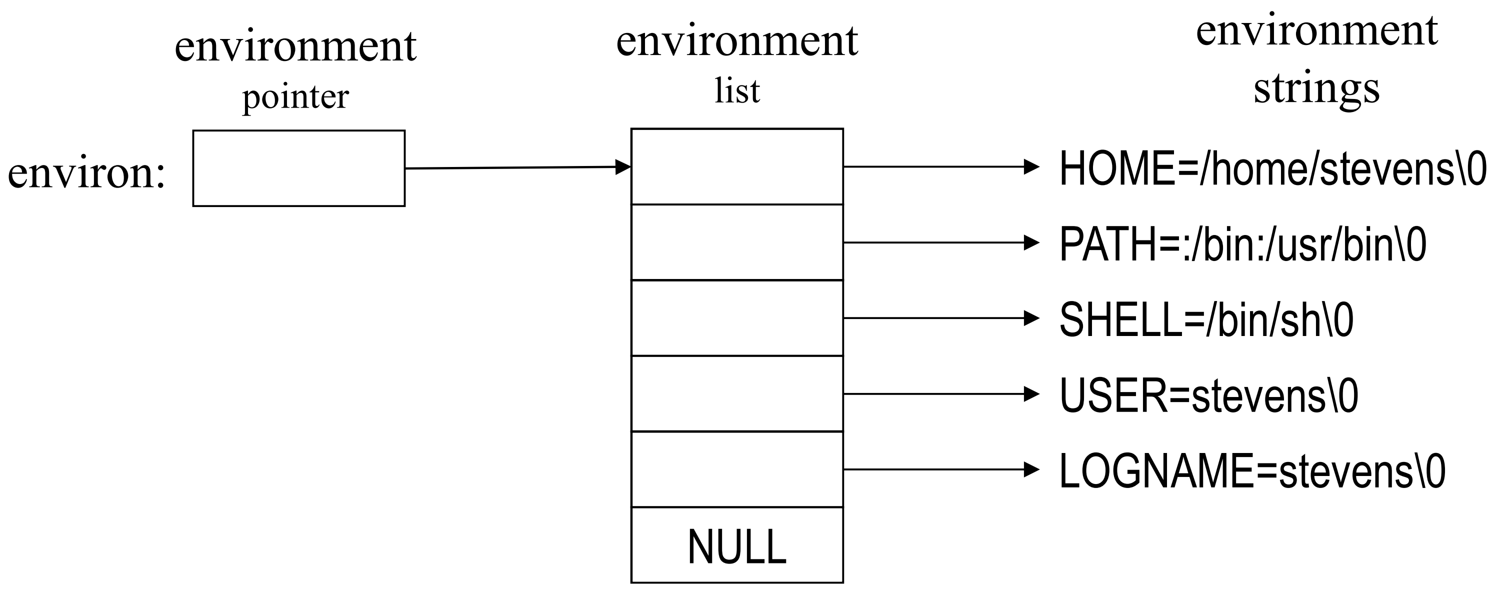
```
/* fileID,   code file = redir.c   */
#include <unistd.h>
int main (void)
{
     int fileID;
     fileID = creat( "x.lis",0640 );    // 0640 is the mode
     if( fileID < 0 )  {
              printf( "error creating x.lis\n " );
              exit (1);
     }
     dup2( fileID, stdout ); /* copy fileID to stdout */
    close( fileID );
    execl( "/bin/ls", "ls", 0 );   return EXIT_SUCCESS;
}
```

Environment

Environment

```
extern char **environ;
```

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



Example: environ

```
#include <stdio.h>
void main( int argc, char *argv[], char *envp[] )
{
   int i;
   extern char **environ;
   printf( “from argument envp\n” );
   for( i = 0; envp[i]; i++ )
        puts( envp[i] );
   printf(“\nFrom global variable environ\n”);
   for( i = 0; environ[i]; i++ )
        puts(environ[i]);
}
```

Sample output

```
MANPATH=/usr/man:/usr/local/man:/usr/share/man:/usr/local/share:man

rvm_bin_path=/usr/local/rvm/bin

HOSTNAME=athena.ecs.csus.edu

GEM_HOME=/usr/local/rvm/gems/ruby-2.0.0-p0

TERM=xterm

SHELL=/bin/bash

HISTSIZE=1000

SSH_CLIENT=130.86.204.182 56016 22

MAIL=/var/spool/mail/nguyendh

PATH=/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/bin:/etc:/usr/local/etc:/usr/bin/X
  11:/opt/bin:/usr/X11R6/bin:/usr/pkg/fm/bin:/usr/pkg/mv/bin:/usr/pkg/pts/bin:/usr
  /pkg/sml/bin:/usr/pkg/syn/bin:/usr/pkg/synmc/bin:/usr/pkg/vcs/bin:/usr/pkg/virsi
  mrm/bin:.

HOME=…..
```

getenv

```
#include <stdlib.h>
    char *getenv(const char *name);
```

Searches the environment list for a string that matches the string pointed to by *name*.

Returns a pointer to the value in the environment, or NULL if there is no match.

For example:

```
myhome = getenv(“HOME”));
```

putenv

- **#include <stdlib.h>**

  int putenv(const char ****string*);**
  - Adds or changes the value of environment variables.
  - The argument *string* is of the form name=value.
  - If name does not already exist in the environment, then string is added to the environment.
  - If name does exist, then the value of name in the environment is changed to value.
  - Returns zero on success, or -1 if an error occurs.
  - For example:
    - putenv("HOME=/");

setenv (1 of 2)

```
#include <stdlib.h>
int setenv(const char *name, const char
          *value, int overwrite);
```

The setenv() function adds the variable name to the environment with the value value, if name does not already exist.

If name does exist in the environment, then

its value is changed to value if overwrite is nonzero

if overwrite is zero, then the value of name is not changed (and setenv() returns a success status).

setenv (2 of 2)

This function makes copies of the strings pointed to by name and value (by contrast with

putenv(3)).

Example: setenv("PWD", tempbuf, **1**);

Example : getenv, putenv (Example 1)

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

void main(void)
{
    printf(“Home directory is %s\n”, getenv(“HOME”));

    putenv(“HOME=/”);

    printf(“New home directory is %s\n”, getenv(“HOME”));
}
```

Example : getenv, setenv (Example 2)

```
//  User types cd command
#ifndef PATH_MAX
#define PATH_MAX 255
#endif
 ………………..
 temp = getenv("HOME");
  chdir(temp); /* CHDIR(2) */
 /* update PWD environment variable with the new directory */
 getcwd(tempbuf,PATH_MAX);
 setenv("PWD", tempbuf,1);
```

User and Group ID

User and Group ID (1 of 2)

Each process has three Ids

(1) Real user ID (RUID)

used to determine which user started the process

Same as the user ID of parent (unless changed)

(2) Effective user ID (EUID)

- Used to assign ownership of newly created files
- to check file access permissions
- to check permission to send signals to processes.

Two ways to change: from set user ID bit on the file being executed, or system call

(3) Saved user ID (SUID)

So previous EUID can be restored

User and Group ID (2 of 2)

Notes:

Real and effective uid: inherit (fork), maintain (exec).

Normally, effective user and effective group user ids have the same value as real user ids except when we change it (in two ways)

Real **group** ID, effective **group** ID, used similarly .

Type of Users in Linux System (optional)

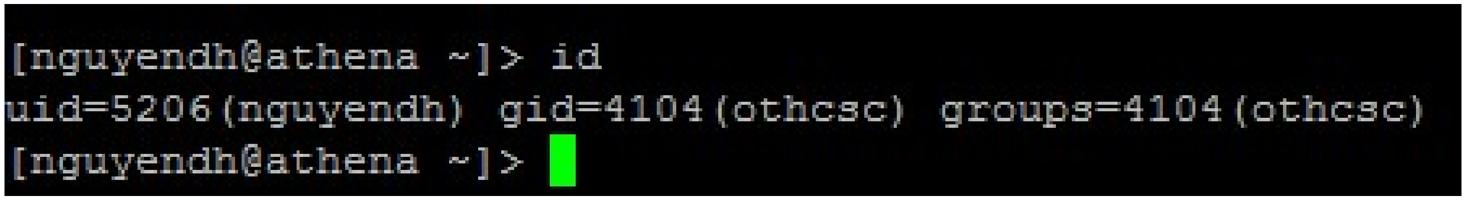
* Super User (or Root) – uid 0

* System User – uid 1-499

* Regular User – uid: 500 < 6000

* Network User – uid > 6000

* Pseudo User – Replica of Super User

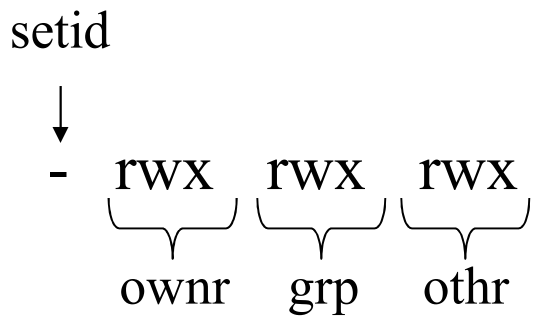


```
[nguyendh@athena ~]> id
uid=5206(nguyendh) gid=4104(othcsc) groups=4104(othcsc)
[nguyendh@athena ~]>
```

User changes password ? i.e use passwd

Recall: Unix file security

- Each file has owner and group
- Permissions set by owner
  - Read, write, execute
  - Owner, group, other
  - Represented by vector of four octal values
- Only owner, root can change permissions
  - This privilege cannot be delegated or shared



Recall: Special note on File type and permissions

File type and permissions

The *st_mode* field is a bit mask serving the dual purpose of identifying the file type and specifying the file permissions. The bits of this field are laid out as shown in Figure 15-1.

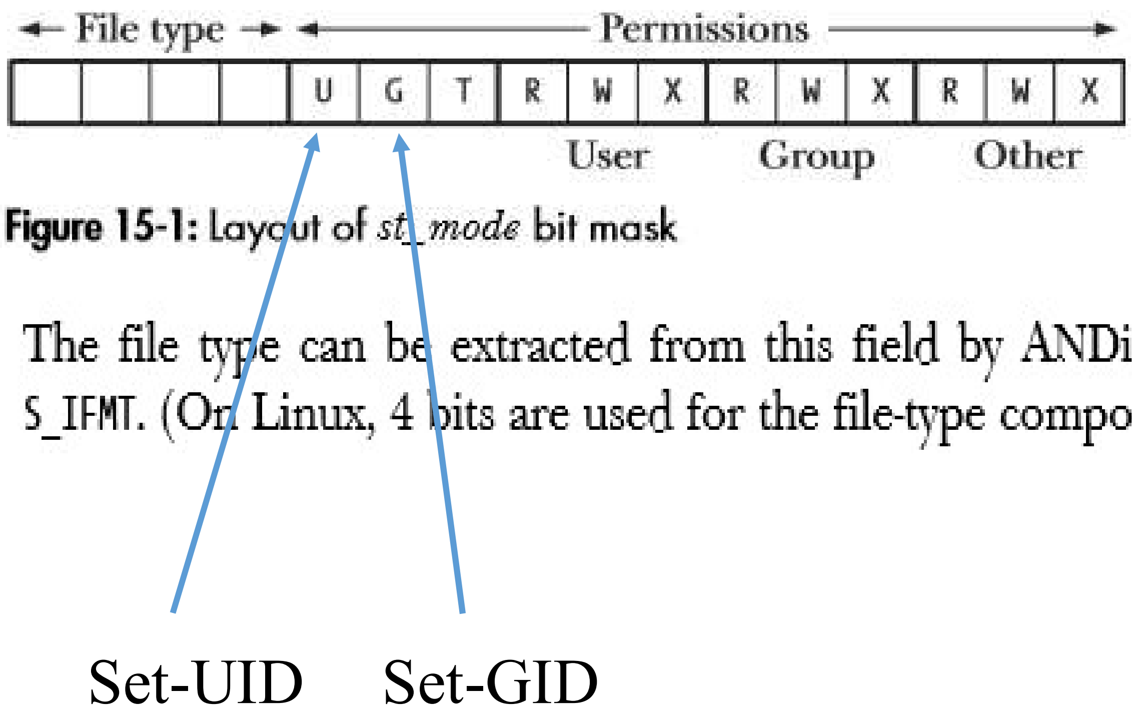


Figure 15-1: Layout of *st_mode* bit mask

The file type can be extracted from this field by ANDing (&) with the constant S_IFMT. (On Linux, 4 bits are used for the file-type component of the *st_mode* field.

LPI (Page 281)

Side Note.

After the Set-UID and Set-GID bits on the previous slide, there is a bit labeled “T”;

That stands for Sticky Bit.

A **sticky bit** is a permission **bit** that is set on a directory that allows only the owner of the file within that directory or the root user to delete or rename the file. No other user has the needed privileges to delete the file created by some other user.

First Option: Set User ID Bit

Owner sets it with chmod

u – owner; g – group; o – all other user

```
chmod u+x executable_file

chmod u+s executable_file

    -rws------ executable_file
```

Process Operations and IDs

- Root
  - ID=0 for superuser root; can access any file
- Fork and Exec
  - Inherit three IDs

ID	Set-user-ID bit off	Set-user-ID bit on
RUID	Unchanged	Unchanged
EUID	Unchanged	Set from user ID of program file
SUID	Copied from EUID	Copied from EUID

Process Operations and IDs (1 of 2)

When a set-user-ID program is run (i.e. loaded into a process’s memory by an *exec()*), the kernel sets the effective user ID of the process to be the same as the user ID of the executable file. Running a set-group-ID program has an analogous effect for the effective group ID of the process. Changing the effective user or group ID in this way gives a process (in other words, the user executing the program) privileges it would not normally have. For example, if an executable file is owned by *root* (superuser) and has the set-user-ID permission bit enabled, then the process gains superuser privileges when that program is run.

(LPI – P169)

Process Operations and IDs – Example (2 OF 2)

```
$ su
Password:
# chown root check_password             Make this program owned by root.
#chmod u+s check_password               With the set-user-ID bit enabled
# ls –l check_password
-rwsr-xr-x    1 root    users   18150 Oct 28 10:49 check_password
# exit
$ whoami                                This is an unprivileged login.
Mtk
$  ./check_password                     But we can now access the shadow
Username:  avr                          password file using this program
Password:
Successfully authenticated: UID=1001
```

LPI page 169-70

Second Option: setuid(*uid*) system calls

ID	**Super user**	**Unprivileged user**
RUID	Set to *uid*	Unchanged
EUID	Set to *uid*	Set to *uid*
SUID	Set to *uid*	unchanged

System call: Read IDs

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

pid_t getuid(void);

Returns the real user ID of the current process

pid_t geteuid(void);

Returns the effective user ID of the current process

gid_t getgid(void);

Returns the real group ID of the current process

gid_t getegid(void);

Returns the effective group ID of the current process

System call: Change UID and GID

```
#include <unistd.h>
int setuid( uid_t uid )
int setgid( gid_t gid )
```

Sets the effective user ID of the current process.

- Superuser process resets the real effective user IDs to *uid*.

- Non-superuser process can set effective user ID to *uid*, only when *uid* equals real user ID or the saved set-user ID (set by executing a setuid-program in exec).

- In any other cases, setuid returns error.

Difference between:
Real User ID
Effective User ID
Saved User ID

http://stackoverflow.com/questions/32455684/difference-between-real-user-id-effective-user-id-and-saved-user-id

or

http://tinyurl.com/z3eakft

The contents of the link are pasted into the next slides.

Page 1 of 2.

The distinction between a real and an effective user id is made because you may have the need to temporarily take another user's identity (most of the time, that would be root, but it could be any user). If you only had one user id, then there would be no way of changing back to your original user id afterwards (other than taking your word for granted, and in case you are root, using root's privileges to change to any user).

So, the real user id is who you really are (the one who owns the process), and the effective user id is what the operating system looks at to make a decision whether or not you are allowed to do something (most of the time, there are some exceptions).

When you log in, the login shell sets both the real and effective user id to the same value (your real user id) as supplied by the password file.

Page 2 of 2.

Now, it also happens that you execute a setuid program, and besides running as another user (e.g. root) the setuid program is also supposed to do something on your behalf. How does this work? After executing the setuid program, it will have your real id (since you're the process owner) and the effective user id of the file owner (for example root) since it is setuid.

The program does whatever magic it needs to do with superuser privileges and then wants to do something on your behalf. That means, attempting to do something that you shouldn't be able to do should fail. How does it do that? Well, obviously by changing its effective user id to the real user id!

Now that setuid program has no way of switching back since all the kernel knows is your id and... your id. Bang, you're dead.

This is what the saved set-user id is for.

Some Applications of fork

Definitions

- **getty** , short for "get tty", is a Unix program running on a host computer that manages physical or virtual terminals (TTYs). When it detects a connection, it prompts for a username and runs the 'login' program to authenticate the user.

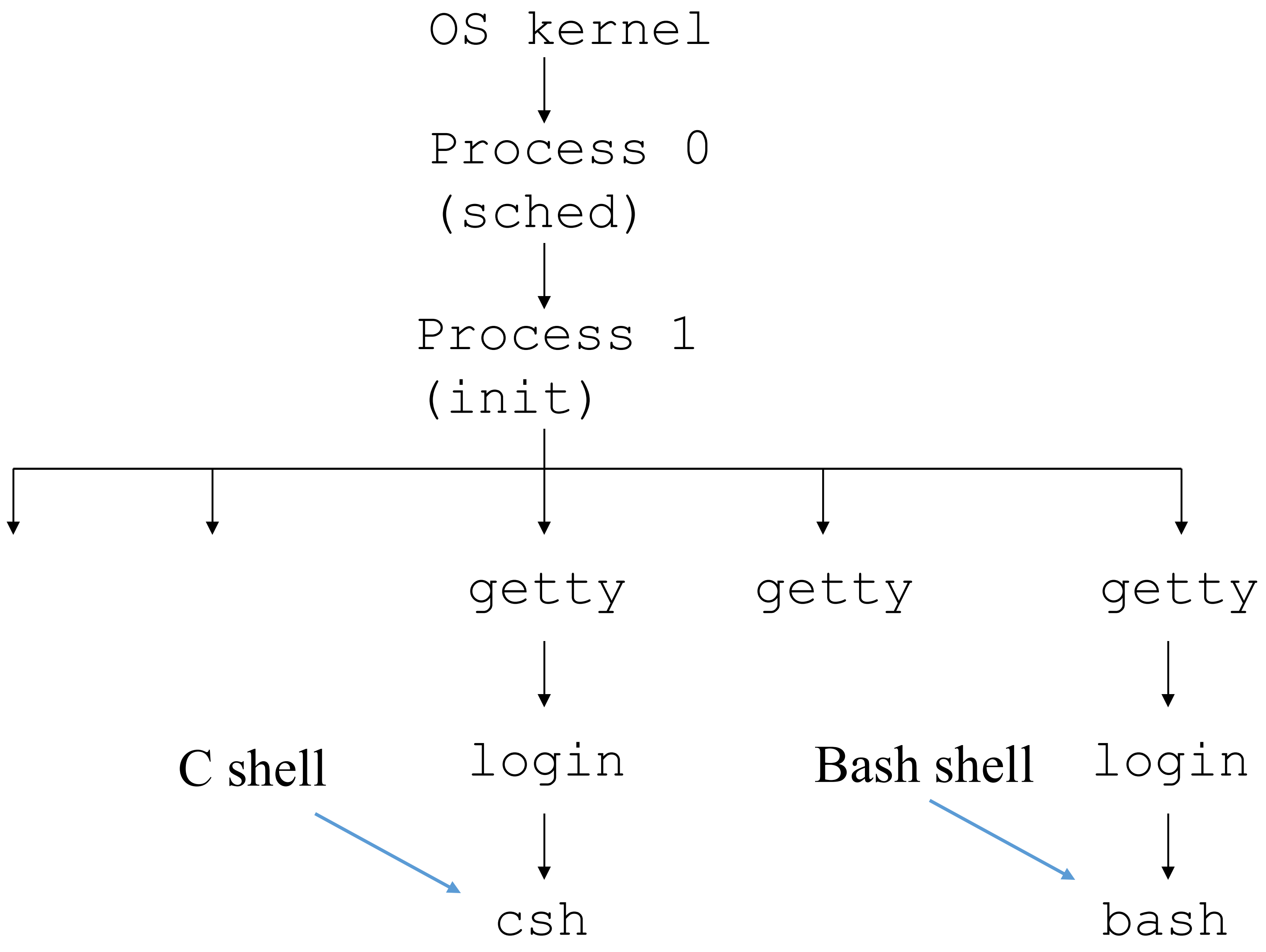
Definition of **init**

- **init** (short for *initialization*) is the first process started during booting of the computer system.
- **init** is a daemon or background process that continues running until the system is shut down.
- It is the direct or indirect ancestor of all other processes and automatically adopts all orphaned processes.
- **init** is started by the kernel using a hard-coded filename; a kernel panic will occur if the kernel is unable to start it.
- **init** is typically assigned process identifier 1.

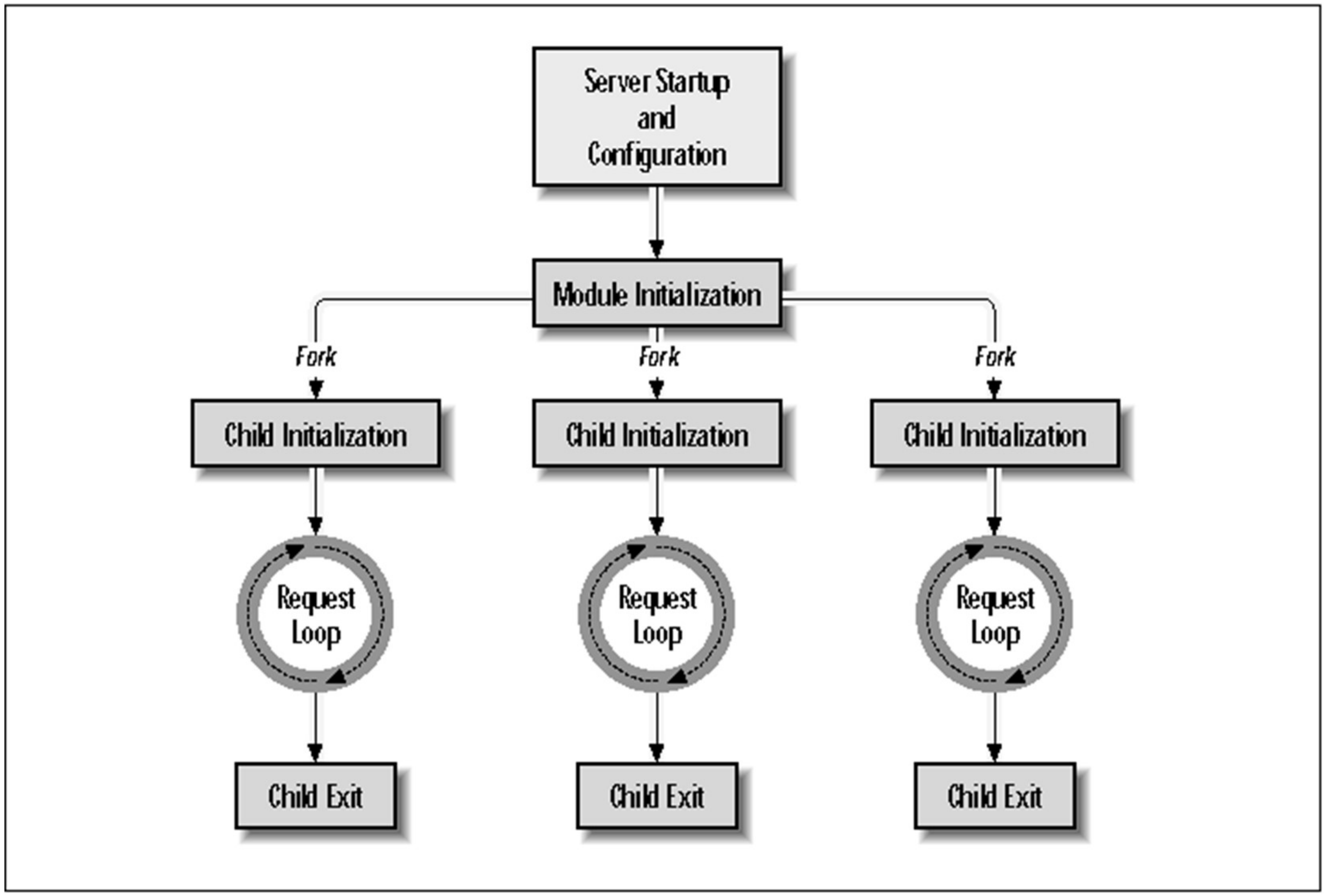
Definitions

- **login** - is used when signing onto a system. If no argument is given, **login** prompts for the username

Unix Start Up Processes Diagram



Apache Web Server



Source: http://docstore.mik.ua/orelly/apache_mod/24.htm

8-UNIX
exec & fork System Calls

The End