# System Programming

## Module 4

# EGCI 252
# System Programming

Computer Engineering Department
Faculty of Engineering
Mahidol University

# Race Condition

A race condition occurs when multiple processes are trying to do something with shared resource and the final outcome depends on the order in which the processes run. The fork function is a lively breeding ground for race conditions, if any of the logic after the fork either explicitly or implicitly depends on whether the parent or child runs first after the fork. In general, we cannot predict which process runs first. Even if we knew which process would run first, what happens after that process starts running depends on the system load and the kernel's scheduling algorithm.

# Race Condition: Example

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    pid_t pid; char *msg, c; int n;

    printf("fork program starting\n");
    pid = fork();
    switch(pid)
    {
        case -1:  exit(1);
        case 0:  msg = "C"; n = 35; break;
        default:  msg = "P"; n = 35; break;
    }

    setbuf(STDOUT, NULL);    // set unbuffered

    for (; n>0; n--)
    {
        while(c = *msg++)
        {putc(c, STDOUT);  sleep(1);}
    }
    if (pid)
    {
        int stat_val; pid_t child_pid;

        child_pid = wait(&stat_val);
        printf("Child has finished: PID = %d\n",
                child_pid);
        if (WIFEXITED(stat_val)
            printf("Child exited with code %d\n",
                    WEXITSTATUS(stat_val);
        else
            printf("Child terminated abnormally\n");
    }
    exit(0);
}
```

# Avoiding Race Condition: Example

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


int main()
{
    pid_t pid; char *msg, c;
    int n, turn = 0, flagfd;


    flagfd = open(“flag”, O_RDWR | O_CREAT,
     S_IRUSR |  S_IWUSR);
    write(flagfd, &turn, 1);
    printf(“fork program starting\n”);
    pid = fork();
    switch(pid)
    {
          case -1:  exit(1);
          case 0:  msg = “Child!\n”; n = 5; break;
          default:  msg = “Parent!\n”; n = 5; break;
    }


    setbuf(STDOUT, NULL);    // set unbuffered

    for (; n>0; n--)
    {
          if (pid)
          {
               while (turn)
               { sleep(1); read (flagfd, &turn, 1); }
          }
          else
          {
               while (!turn)
               { sleep(1); read (flagfd, &turn, 1); }
          }
          while (c = *msg++) putc(c, STDOUT);
          if (pid) { turn = 1; write (flagfd, &turn, 1);  }
          else { turn = 0; write (flagfd, &turn, 1); }
    }
    return 0;
}
```

# User Identification : User ID

- The user ID from our entry in the password file is a numeric value that identifies us to the system. This user ID is assigned by the system administrator when our login name is assigned, and we cannot change it. The user ID is normally assigned to be unique for every user. We'll see how the kernel uses the user ID to check whether we have the appropriate permissions to perform certain operations.
- We call the user whose user ID is 0 either root or the superuser. The entry in the password file normally has a login name of root, and we refer to the special privileges of this user as superuser privileges. If a process has superuser privileges, most file permission checks are bypassed. Some operating system functions are restricted to the superuser. The superuser has free rein over the system.

# User Identification : Group ID

- Our entry in the password file also specifies our numeric group ID. This too is assigned by the system administrator when our login name is assigned. Typically, the password file contains multiple entries that specify the same group ID. Groups are normally used to collect users together into projects or departments. This allows the sharing of resources, such as files, among members of the same group. There is also a group file that maps group names into numeric group IDs. The group file is usually /etc/group.
- The use of numeric user IDs and numeric group IDs for permissions is historical. With every file on disk, the file system stores both the user ID and the group ID of a file's owner. Storing both of these values requires only four bytes, assuming that each is stored as a two-byte integer. Early UNIX systems used 16-bit integers to represent user and group IDs. Contemporary UNIX systems use 32-bit integers.

# User Identification : Supplementary Group IDs

- In addition to the group ID specified in the password file for a login name, most versions of the UNIX System allow a user to belong to additional groups. This started with 4.2BSD, which allowed a user to belong to up to 16 additional groups. These supplementary group IDs are obtained at login time by reading the file /etc/group and finding the first 16 entries that list the user as a member. As we shall see in the next chapter, POSIX requires that a system support at least eight supplementary groups per process, but most systems support at least 16.

# Set-User-ID and Set-Group-ID

- Every process has six or more IDs associated with it.

| | |
|---|---|
| real user ID<br>real group ID | who we really are.<br>the two owner IDs are properties of the file. |
| effective user ID<br>effective group ID<br>supplementary group IDs | used for file access permission checks.<br>the two effective IDs and the supplementary group IDs are properties of the process. |
| saved set-user-ID<br>saved set-group-ID | saved by exec functions. |

# Set-User-ID and Set-Group-ID (cont.)

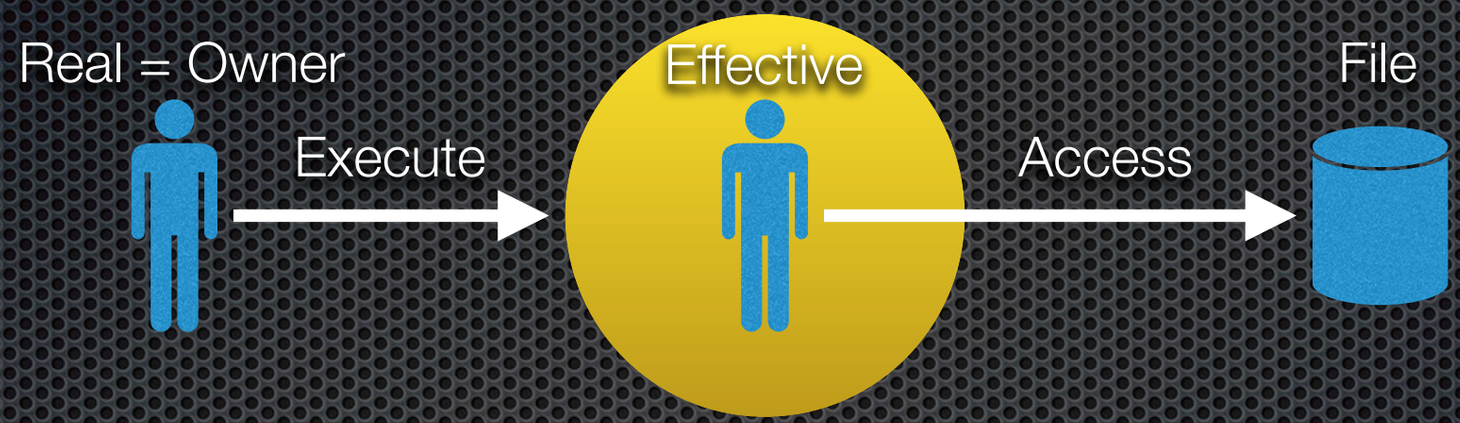
- Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.
- When we execute a program file, the effective user ID of the process is usually the real user ID, and the effective group ID is usually the real group ID. But the capability exists to set a special flag in the file's mode word (st_mode) that says "when this file is executed, set the effective user ID of the process to be the owner of the file (st_uid)." Similarly, another bit can be set in the file's mode word that causes the effective group ID to be the group owner of the file (st_gid). These two bits in the file's mode word are called the set-user-ID bit and the set-group-ID bit.

# Set-User-ID and Set-Group-ID (cont.)

- For example, if the owner of the file is the superuser and if the file's set-user-ID bit is set, then while that program file is running as a process, it has superuser privileges. This happens regardless of the real user ID of the process that executes the file. As an example, the UNIX System program that allows anyone to change his or her password, passwd(1), is a set-user-ID program. This is required so that the program can write the new password to the password file, typically either /etc/passwd or /etc/shadow, files that should be writable only by the superuser. Because a process that is running set-user-ID to some other user usually assumes extra permissions, it must be written carefully.

# Real User ID & Effective User ID

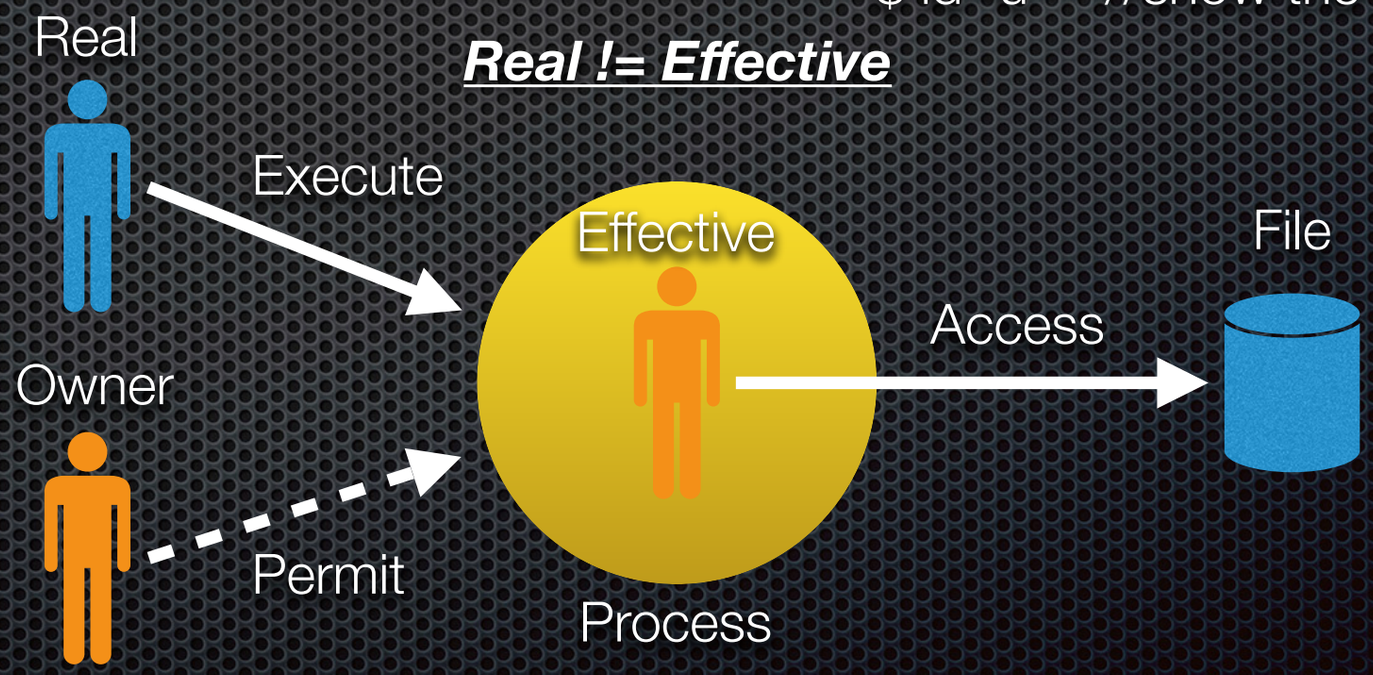
***Real == Effective***




Process

$ id -u  //show the effective user ID

***Real != Effective***



# File Access Permissions

- The st_mode value encodes the access permission bits for the file.
- There are nine permission bits for each file, divided into three categories.

| st_mode mask | Meaning |
|---|---|
| S_IRUSR<br>S_IWUSR<br>S_IXUSR | user-read<br>user-write<br>user-execute |
| S_IRGRP<br>S_IWGRP<br>S_IXGRP | group-read<br>group-write<br>group-execute |
| S_IROTH<br>S_IWOTH<br>S_IXOTH | other-read<br>other-write<br>other-execute |

# File Access Tests

The file access tests performed by the kernel are as follows:

- If the effective user ID of the process is 0 (the superuser), access is allowed. This gives the superuser free rein throughout the entire file system.
- If the effective user ID of the process equals the owner ID of the file (i.e., the process owns the file), access is allowed if the appropriate user access permission bit is set. Otherwise, permission is denied. By appropriate access permission bit, we mean that if the process is opening the file for reading, the user-read bit must be on. If the process is opening the file for writing, the user-write bit must be on. If the process is executing the file, the user-execute bit must be on.

# File Access Tests (cont.)

- If the effective group ID of the process or one of the supplementary group IDs of the process equals the group ID of the file, access is allowed if the appropriate group access permission bit is set. Otherwise, permission is denied.
- If the appropriate other access permission bit is set, access is allowed. Otherwise, permission is denied.

# Inherited Properties of Child Processes

- Real user ID, real group ID,
- **Effective user ID**, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

# Effective User ID Example

```
// euid_exp.c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
   int pid = fork();

   if(pid == 0)
   {
      printf("Child   : RUID = %d, EUID = %d\n", getuid(), geteuid());
      execlp(argv[1], argv[1], argv[2], NULL);
   }
   else
   {
      wait(NULL);
      printf("Parent : RUID = %d, EUID = %d\n", getuid(), geteuid());
   }
}
```

# Effective User ID Example (Cont.)

- In order to make this work, two users are needed.

Owner of this program:

```
$ mkdir /tmp/tmp_dir                  //Make a temporary directory
$ touch /tmp/tmp_dir/demo             //Create a simple file in the directory
$ chmod o-rwx /tmp/tmp_dir            //Disable all permissions for other users
$ chmod u+s euid_exp                  //Turn on the set-user-ID bit
$ cp euid_exp /tmp                    //Copy the program to the tmp directory
```

Another user:

```
$ cd /tmp
$ ls tmp_dir
  ls: tmp_dir: Permission denied
$ ./euid_exp ls /tmp/tmp_dir          //Execute the program to access the tmp_dir
  Child  : RUID = 501, EUID = 502
  demo
  Parent : RUID = 501, EUID = 502
```

# System( ) Function

- int system(const char * cmd)
- system() forks a child process that exec's /bin/sh, which in turn runs the command cmd
- As such, it has the following qualities:
  - it's easy and familiar to use
  - it's inefficient
  - because it uses system variables and executes from a shell, it can be a security risk if the command is setuid or setgid
- example:  system("ls –la /usr/bin");

# System( ) Example

- Using system library function

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

int main()
{
  printf(“Using system function for running ps program.\n”);
  system(“ps -ax”);
  printf(“ps completed\n”);
  return 0;
}
```

# Warning on system function

- Do not use system() from a program with set-UID or set-GID privileges, because inherited privileges and strange values for some environment variables might be used to subvert system integrity. Use the **exec**(3) family of functions instead.
- system() will not, in fact, work properly from programs with set-UID or set-GID privileges on systems on which /bin/sh is bash version 2, since bash 2 drops privileges on startup. (Debian uses a modified bash which does not do this when invoked as sh.)

# System( ) in Set-User-ID Programs

- Execute the command-line argument using system

```
// mysys.c
#include <stdio.h>

int main(int argc, char *argv[])
{
    int status;
    if (argc < 2)
    {
       printf(“\nCommand-line argument required!\n");
       exit(0);
    }
     if ((status = system(argv[1])) < 0)
       perror("system() error");
    return 0;
}
```

```
// showuid.c
#include <stdio.h>

int main(void)
{
   printf(“Real UID = %d,
           Eff UID = %d\n”,
           getuid(), geteuid());
   return 0;
}
```

# System( ) in Set-User-ID Programs (cont.)

- Running both programs as the following:
  (In order to make this work, the system needs to the active root account on the system)

```
$ ./mysys ./showuid                    //normal execution, no special privileges
   real uid = 205, effective uid = 205
$ sudo chown root mysys                //change owner
$ sudo chmod u+s mysys                 //make set-user-ID
$ ls -l mysys                          //verify file's permissions and owner
  -rwsrwxr-x 1 root 16361 Mar 16 16:59 mysys
$ ./ mysys ./showuid
   real uid = 205, effective uid = 0  //oops, this is a security hole
```

An example command that would allow hacker to access the system with the new user account “./mysys ./showuid’; /usr/sbin/useradd ‘hacker”

# End of Module