Chapter 8.
Process Control

System Programming

http://www.cs.ccu.edu.tw/~pahsiung/courses/sp

熊博安

國立中正大學資訊工程學系

pahsiung@cs.ccu.edu.tw

(05)2720411 ext. 33119

Class: EA-104

Office: EA-512

Introduction

- Creation of new processes
- Executing programs
- Process termination
- IDs: real, effective, saved
- system()
- Process accounting

Process Identifiers

- Process ID = a nonnegative integer

PID	Process
0	swapper (scheduler)
1	init (/sbin/init)
2	pagedaemon (virtual memory paging)
3, 4, …	other processes

Identifier functions

- #include <unistd.h>
- pid_t getpid(void); return PID
- pid_t getppid(void); return parent PID
- uid_t getuid(void); return real UID
- uid_t geteuid(void); return effective UID
- gid_t getgid(void); return real GID
- gid_t getegid(void); return effective GID

fork Function

- for() is the ONLY way to create a process in Unix kernel by user

```
#include <unistd.h>
pid_t fork(void);
```

- Returns: 0 in child, child PID in parent, -1 on error

Parent / Child Processes

- Parent and child continue executing instructions following the fork() call
- Child gets a copy of parent's data space, heap, and stack
- Often, read-only text segment is shared
- Often, fork() is followed by exec()
- Waste of space and time for setting up child's program space!!!

Copy-On-Write (COW)

- Memory regions are read-only and shared by parent and child
- If either process wants to write, kernel makes a copy of that memory only for that process.
- Saves space and time!

Variations of fork

- All 4 platforms support vfork(2)
- Linux 2.4.22
  - clone(2) system call
    - Can control what to share between parent and child processes
- FreeBSD 5.2.1
  - rfork(2) system call: similar to clone()
- Solaris 9
  - POSIX Thread: a new process with only the calling thread
  - Solaris Thread: a new process with all threads from the process of calling thread

Figure 8.1: fork()

```
#include “apue.h"

int      glob = 6;        /* external variable in initialized data */
char     buf[] = "a write to stdout\n";

int main(void) {
    int   var;            /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    if (write(STDOUT_FILENO, buf, sizeof(buf)-1) != sizeof(buf)-1)
          err_sys("write error");
    printf("before fork\n");       /* we don't flush stdout */

    if ( (pid = fork()) < 0)        err_sys("fork error");
    else if (pid == 0) {           /* child */
     glob++; var++;                /* modify variables */
    } else
     sleep(2);                     /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

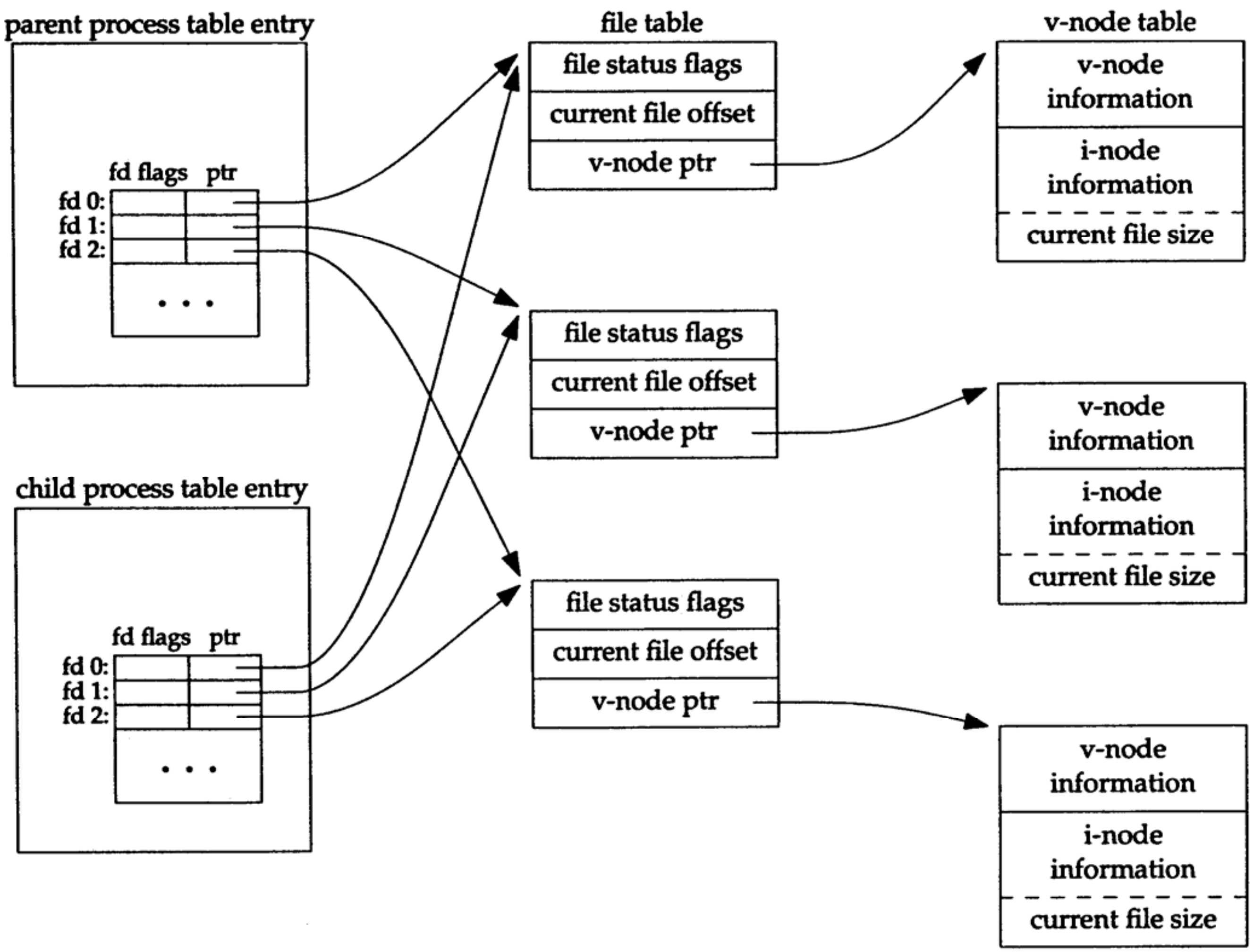
Figure 8.1: results

- **$ ./a.out**
- a write to stdout
- before fork
- pid = 430, glob = 7, var = 89
- pid = 429, glob = 6, var = 88
- **$ ./a.out > temp.out**
- **$ cat temp.out**
- a write to stdout
- before fork
- pid = 432, glob = 7, var = 89
- before fork
- pid = 431, glob = 6, var = 88

File Sharing

- Parent and child share the same file descriptors
- As if dup() had been called on all open file descriptors
- Parent and child share the same file offset, o/w overwrite
- Intermixed output from parent and child

File Sharing between Parent and Child



Handle descriptors after fork

- Parent waits for child to complete
  - File offsets updated by child
- Parent and child go their own way
  - Parent closes unrequired file descriptors
  - Child closes unrequired file descriptors

Fork

- fork fails if:
  - too many processes in system
  - total #processes exceeded for a user
- 2 uses for fork
  - Duplicate of itself for executing different code sections, e.g. network servers
  - Execute a different program, e.g. shells

vfork Function

- Creates a new process only to ‘exec’ a new program
- No copy of parent’s address space for child (not needed!)
- Before exec, child runs in “address space of parent”
- Efficient in paged virtual memory
- Child runs first
- Parent waits until child ‘exec’ or ‘exit’

Figure 8.3: vfork()

```
#include "apue.h"

int     glob = 6;        /* external variable in initialized data */

int main(void) {
    int          var;    /* automatic variable on the stack */
    pid_t pid;

    var = 88;
    printf("before vfork\n");      /* we don't flush stdio */

    if ( (pid = vfork()) < 0)
         err_sys("vfork error");
    else if (pid == 0) {           /* child */
         glob++;                   /* modify parent's variables */
         var++;
         _exit(0);                 /* child terminates */
    }
    /* parent */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

Figure 8.3: results

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

- (increments by child appear in parent address space)
- Instead of _exit() → exit(), results in:

```
$ ./a.out
before vfork
```

← stdio closed by child

exit Functions

- A process can terminate in 8 ways:
- Normal Termination (5 ways)
  - return from main
  - exit
  - _exit or _Exit
  - return from last thread in a process
  - calling pthread_exit from last thread in a process
- Abnormal Termination (3 ways)
  - calling abort (SIGABRT signal) e.g. ÷0
  - process receives signals
  - last thread cancelled (deferred cancellation)

exit Function

- exit status: argument of exit, _exit, _Exit
- termination status:
  - normal: exit status
  - abnormal: kernel indicates reason
- Child returns termination status to parent
  - What if parent terminates before child?

exit Function

- Parent PID (of orphaned child) = 1 (init)
- Suppose child terminates first
- If child disappeared, parent would not be able to check child's termination status
- Zombie: minimal info of dead child process (pid, termination status, CPU time)
- init's inherited child do not become zombies (wait() fetches status)

Child Termination

- Child terminates →
  Kernel sends SIGCHLD signal to parent
- Default action for SIGCHLD signal: ignore it
- Signal handlers can be defined by users
- (Chapter 10)

wait(), waitpid()

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

pid_t wait(int *statloc);

pid_t waitpid( pid_t pid, int *statloc,
               int options);
```

block wait for any one child to terminate

depends on options

place for storing termination status
NULL→no need!

- Return: PID if OK, 0, -1 on error

Termination Status Macros

Macro	Description
`WIFEXITED` (*status*)	True if status was returned for a child that terminated normally. In this case we can execute `WEXITSTATUS` (*status*) to fetch the low-order 8 bits of the argument that the child passed to `exit` or `_exit`.
`WIFSIGNALED` (*status*)	True if status was returned for a child that terminated abnormally (by receipt of a signal that it didn't catch). In this case we can execute `WTERMSIG` (*status*) to fetch the signal number that caused the termination. Additionally, SVR4 and 4.3+BSD (but not POSIX.1) define the macro `WCOREDUMP` (*status*) that returns true if a core file of the terminated process was generated.
`WIFSTOPPED` (*status*)	True if status was returned for a child that is currently stopped. In this case we can execute `WSTOPSIG` (*status*) to fetch the signal number that caused the child to stop.
WIFCONTINUED(status)	True if status was returned for a child that has been continued after a job control stop (XSI extension to POSIX.1; waitpid only)

Figure 8.5: print exit status

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

void pr_exit(int status)
{
   if (WIFEXITED(status))
        printf("normal termination, exit status = %d\n",
                        WEXITSTATUS(status));
   else if (WIFSIGNALED(status))
        printf("abnormal termination, signal number = %d%s\n",
                        WTERMSIG(status),
#ifdef   WCOREDUMP
        WCOREDUMP(status) ? " (core file generated)" : "");
#else
        "");
#endif
   else if (WIFSTOPPED(status))
        printf("child stopped, signal number = %d\n",
                        WSTOPSIG(status));
}
```

Figure 8.6: demo exit status

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

int main(void)
{
   pid_t      pid;
   int        status;

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0)          /* child */
        exit(7);
  if (wait(&status) != pid)  /* wait for child */
        err_sys("wait error");
   pr_exit(status);            /* and print its status */
```

Program 8.4 (II Part)

```
if ( (pid = fork()) < 0)
    err_sys("fork error");
else if (pid == 0)     /* child */
    abort();           /* generates SIGABRT */

if (wait(&status) != pid) /* wait for child */
    err_sys("wait error");
pr_exit(status);  /* and print its status */
```

Program 8.4 (III part)

```
  if ( (pid = fork()) < 0)
      err_sys("fork error");
  else if (pid == 0)                /* child */
      status /= 0;  /* divide by 0 generates SIGFPE */

  if (wait(&status) != pid)         /* wait for child */
      err_sys("wait error");
  pr_exit(status);          /* and print its status */

  exit(0);
}
```

Figure 8.6: results

- **$ ./a.out**
- normal termination, exit status = 7
- abnormal termination, signal number = 6 (core file generated)

SIGABRT

- abnormal termination, signal number = 8 (core file generated)

SIGFPE

pid argument of waitpid()

same as wait()

PID	Interpretation
== -1	Wait for any child process
> 0	Wait for child with PID
== 0	Wait for child with GID==calling process GID
< -1	Wait for child with |PID|

Avoiding zombie processes

- A process forks a child
- It does not wait for the child to complete
- It does not want child to become zombie
- How to do this?
- Answer: fork twice! (Figure 8.8)

Figure 8.8: Avoid Zombie

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

int
main(void)
{
   pid_t      pid;

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {            /* first child */
        if ( (pid = fork()) < 0)
              err_sys("fork error");
        else if (pid > 0) /* parent from second fork */
              exit(0);   /*  == first child        */
```

fork a 2nd time

Program 8.5: Avoid Zombie

so that init becomes the new parent

```
        /* We're the second child; our parent becomes init as
        soon as our real parent calls exit() in the statement
        above. Here's where we'd continue executing, knowing
        that when we're done, init will reap our status. */

        sleep(2);
        printf("second child, parent pid = %d\n", getppid());
        exit(0);
    }

    if (waitpid(pid, NULL, 0) != pid)/* wait for first child */
        err_sys("waitpid error");

    /* We're the parent (the original process); we continue
    executing, knowing that we're not the parent of the second
    child. */

    exit(0);
}
```

Figure 8.8: results

- **$ ./a.out**
- $ second child, parent pid = 1

wait3 and wait4

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options,
            struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options,
            struct rusage *rusage);
```

resource information for terminated processes

- Return: PID if OK, 0 or -1 on error

wait arguments support

Function	*pid*	*options*	*rusage*	POSIX.1	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
wait				•	•	•	•	•
waitid	•	•		XSI				•
waitpid	•	•		•	•	•	•	•
wait3		•	•		•	•	•	•
wait4	•	•	•		•	•	•	•

Figure 8.11 Arguments supported by `wait` functions on various systems

Race Conditions

- Multiple processes share some data
- Outcome depends on the order of their execution (i.e. RACE)
- After fork(), we cannot predict if the parent or the child runs first!
- The order of execution depends on:
  - system load
  - kernel's scheduling algorithm

Race Conditions

- Race condition problems are hard to detect because they work “most of the time”!
- For parent to wait for child
  - call wait, waitpid, wait3, wait4
- For child to wait for parent
  - `while (getppid() != 1) sleep(1);`

polling! wastes CPU time!

use signals or other IPC methods

Race Conditions

- After fork
  - parent and child both need to do something on its own
  - e.g. parent: write a record in a log file
  - e.g. child: creates a log file
- Parent and child need to:
  - **TELL** each other when its initial set of operations are done, and
  - **WAIT** for each other to complete

TELL and WAIT

```
#include "apue.h"
TELL_WAIT(); /* setup for TELL_XXX and WAIT_XXX */
if ( (pid = fork()) < 0 )
   err_sys("fork error");
else if (pid==0) {   /* child */
   /* child does whatever is necessary */
   TELL_PARENT(getppid()); /* tell parent we're done */
   WAIT_PARENT(); /* & wait for parent */
   /* and child continues on its way */
   exit(0);
}
/* parent does whatever is necessary */
TELL_CHILD(pid); /*tell child we're done */
WAIT_CHILD(); /* wait for child */
/* and parent continues on its way */
exit(0);
```

child

parent

Figure 8.12: Race Condition

```
#include        “apue.h"

static void charatatime(char *);

int
main(void)
{
   pid_t       pid;

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {
        charatatime("output from child\n");
   } else {
        charatatime("output from parent\n");
   }
   exit(0);
}
```

Figure 8.12 (continued)

```
static void
charatatime(char *str)
{
  char    *ptr;
  int     c;

  setbuf(stdout, NULL); /* set unbuffered */
  for (ptr = str; c = *ptr++; )
     putc(c, stdout);
}
```

Figure 8.12: results

- **$ ./a.out**
- output from child
- output from parent
- **$ ./a.out**
- oouuttppuutt ffrroomm pcahrielndt
- **$ ./a.out**
- ooutput from parent
- utput from child

Figure 8.13: No race condition

```
#include "apue.h"

static void charatatime(char *);

int
main(void)
{
    pid_t pid;

    TELL_WAIT();

    if ( (pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0) {
        WAIT_PARENT();          /* parent goes first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);
    }
    exit(0);
}
```

Figure 8.13 (continued)

```
static void
charatatime(char *str)
{
  char    *ptr;
  int     c;

  setbuf(stdout, NULL); /* set unbuffered */
  for (ptr = str; c = *ptr++; )
      putc(c, stdout);
}
```

exec Functions

```
#include <unistd.h>
int execl(const char *pathname,
       const char *arg0, … /* (char *)0 */);
int execv(const char *pathname,
       char *const argv[]);
int execle(const char *pathname, const char *arg0, … /*
   (char *)0, char *const envp[] */);
int execve(const char *pathname,
       char *const argv[], char *const envp[]);
int execlp(const char *filename,
       const char *arg0, … /* (char *)0 */);
int execvp(const char *filename,
       char *const argv[]);
```

All return -1 on error, no return on success.

Differences among exec functions

- filename (execlp, execvp: uses PATH) v/s pathname (others: does not use PATH)
- list (l) v/s vector (v)
  - list of arguments (execl, execle, execlp)
  - array of pointers to arguments (execv, execve, execvp)
- pointer to an array of pointers to environment strings (execle, execve) v/s environ (others)

Differences among exec functions

Function	*pathname*	*filename*	Arg list	*argv*[]	environ	*envp*[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

Inheritance by child from parent

- PID, Parent PID
- Real UID, Real GID
- Supplementary GIDs
- Process GID
- Session ID
- Controlling Terminal
- Time Left Until Alarm Clock
- Current Working Directory

Inheritance by child from parent

- Root Directory
- File Mode Creation Mask
- File Locks
- Process Signal Mask
- Pending Signals
- Resource Limits
- tms_utime, tms_stime, tms_cutime, tms_cstime values

Changes on exec()

- FD_CLOEXEC flag for files
  - set ➔ close file descriptors on exec()
  - unset (default) ➔ open across exec()
- FD_CLOEXEC flag for directories
  - closed across exec() (POSIX.1)
- UIDs and GIDs
  - real ➔ same across exec()
  - effective ➔ may change (set-[U,G]ID)

Relationship of 6 exec functions

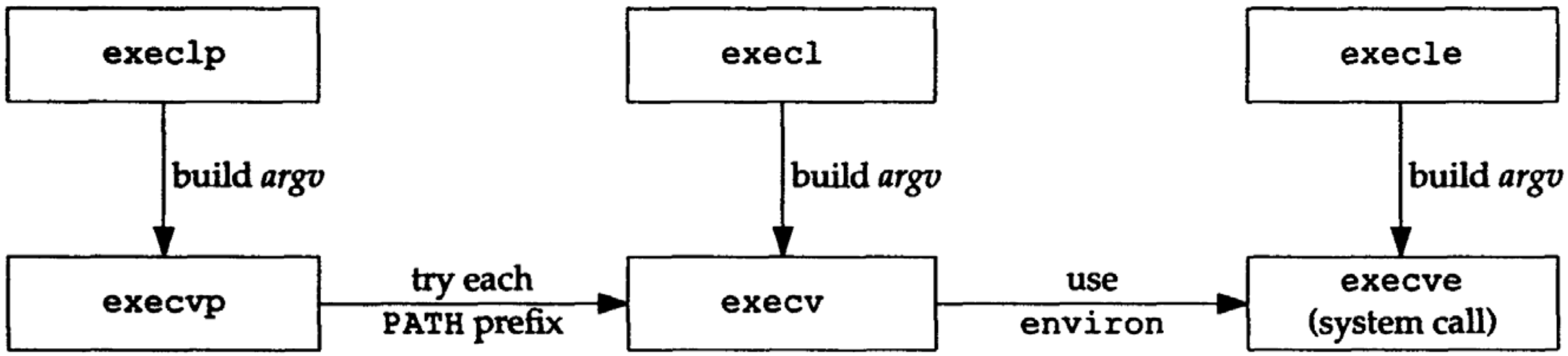


Figure 8.16: exec functions

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

char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };

int main(void) {
   pid_t pid;
   if ( (pid = fork()) < 0) err_sys("fork error");
   else if (pid == 0) {   /* specify pathname, specify environment */
        if (execle("/home/sar/bin/echoall",
        "echoall", "myarg1", "MY ARG2", (char *) 0, env_init) < 0)
        err_sys("execle error");
   }
   if (waitpid(pid, NULL, 0) < 0)
        err_sys("wait error");

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {   /* specify filename, inherit environment */
        if (execlp("echoall", "echoall", "only 1 arg", (char *) 0) < 0)
                err_sys("execlp error");  }
   exit(0); }
```

Figure 8.17: echoall

```
#include       “apue.h"

int
main(int argc, char *argv[])
{
   int          i;
   char         **ptr;
   extern char **environ;

   for (i = 0; i < argc; i++) /* echo all command-line args */
        printf("argv[%d]: %s\n", i, argv[i]);

   for (ptr = environ; *ptr != 0; ptr++)/* and all env strings */
        printf("%s\n", *ptr);

   exit(0);
}
```

Figure 8.16: results

```
$ a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
argv[0]: echoall
$ argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
…
HOME=/home/sar
```

47 more lines that aren't shown

Changing UIDs and GIDs

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

- **Return: 0 if OK, -1 on error**

Changing UIDs and GIDs

- Rules:
- Superuser → real, effective, saved set-UID := uid
- real, saved set-UID = uid → effective := uid
- errno := EPERM; return -1;

Changing 3 UIDs

ID	exec		setuid(*uid*)	
	set-user-ID bit off	set-user-ID bit on	superuser	unprivileged user
real user ID	unchanged	unchanged	set to *uid*	unchanged
effective user ID	unchanged	set from user ID of program file	set to *uid*	set to *uid*
saved set-user ID	copied from effective user ID	copied from effective user ID	set to *uid*	unchanged

What is “saved set-UID”?

- Example: tip (Berkeley) or cu (System V)
- tip: Program to remote connection through modem (exclusive access through lock files)

1. owned by uucp, set-UID bit is SET, on exec:
   - Real UID = our own UID
   - Effective UID = **uucp**
   - Saved set-UID = **uucp**
2. tip accesses lock files owned by uucp (allowed because e-UID == uucp)

What is “saved set-UID”?

3. tip executes `setuid(getuid())`, only e-UID is changed:
   - Real UID = our own UID (unchanged!)
   - Effective UID = our own UID
   - Saved set-UID = **uucp** (unchanged!)

   tip runs with our own UID as effective UID, can access only our own normally accessed files. No additional permissions.

What is “saved set-UID”?

4. tip executes setuid(uucpuid), uucpuid was saved by calling geteuid() when tip started. This call is allowed because the saved set-UID == uucpuid! Thus, we have:
   - Real UID = our own UID (unchanged!)
   - Effective UID = **uucp**
   - Saved set-UID = **uucp** (unchanged!)
5. tip can now release lock files (because its effective UID == uucp!)

setreuid(), setregid()

```
#include <unistd.h>
int setreuid(uid_t ruid, uid_t euid);
int setregid(gid_t rgid, gid_t egid);
```

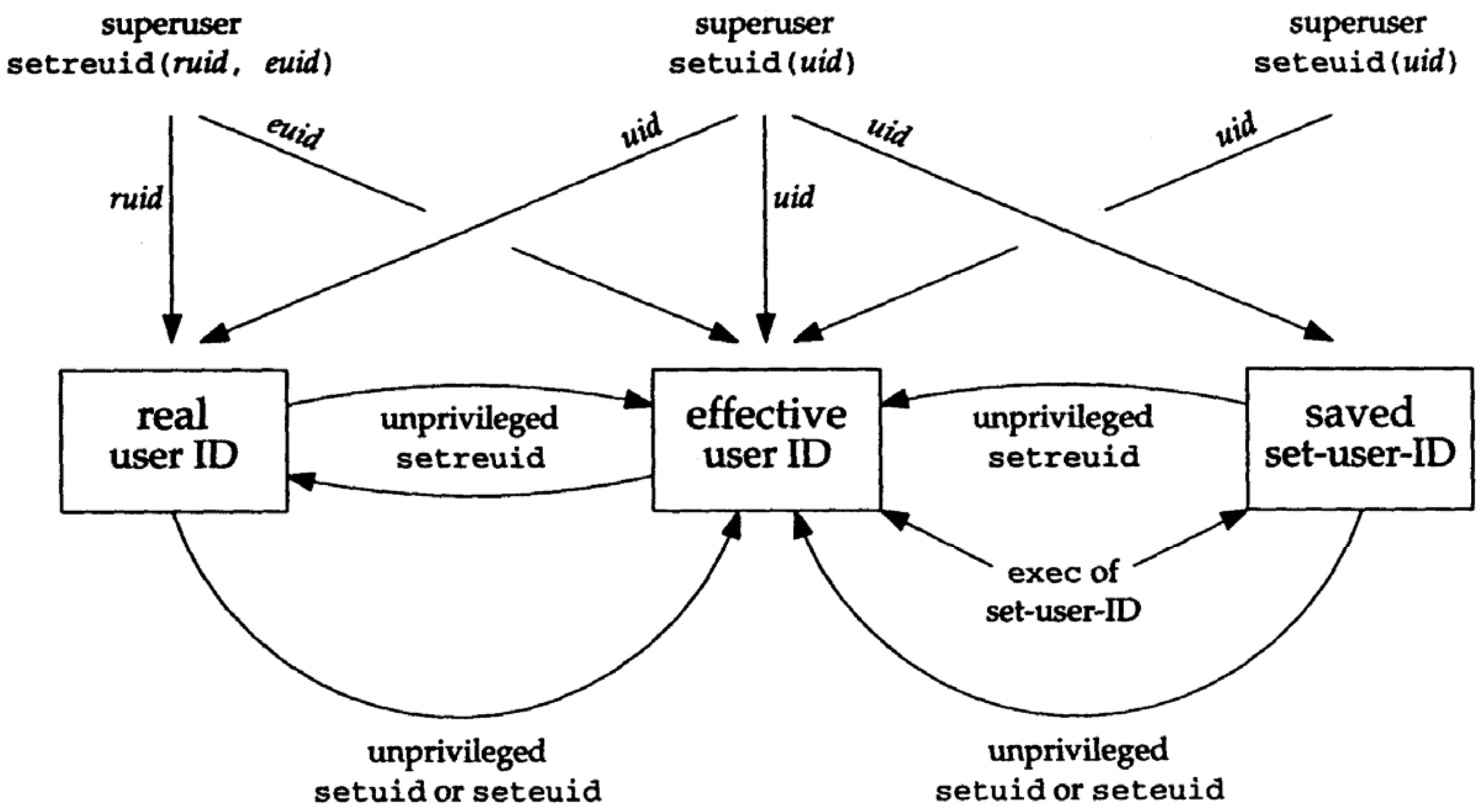
- Return: 0 if OK, -1 on error
- For swapping real ID ←→ effective ID
- XSI extensions in the Single UNIX Specification (BSD supports it currently)

seteuid(), setegid()

```
#include <unistd.h>
int seteuid(uid_t uid);
int setegid(gid_t gid);
```

- Return: 0 if OK, -1 on error
- Only effective UID or GID is changed
- Unpriviledged user:
  - EUID:=uid (== real UID or saved set-UID)
- Priviledged user:
  - EUID:=uid (different from setuid: sets all 3 UIDs)

Summary of set ID functions



Interpreter Files

- Interpreter (script) files that begin with:
- #! pathname [optional-argument]
- E.g.: #! /bin/sh
- E.g.: #! /bin/csh
- E.g.: #! /bin/awk –f
- Allows users an easier and efficient way to execute some commands using scripts
- Limit on 1st line of interpreter files
  - FreeBSD 5.2.1: 128 bytes,
  - Mac OS X 10.3: 512 bytes,
  - Linux 2.4.22: 127 bytes,
  - Solaris 9: 1023 bytes

Figure 8.20: exec an interpreter file

```
#include        <sys/wait.h>
#include        “apue.h"
int main(void) {
   pid_t        pid;

   if ( (pid = fork()) < 0)
        err_sys("fork error");
   else if (pid == 0) {                /* child */
        if (execl("/home/sar/bin/testinterp",
        "testinterp", "myarg1", "MY ARG2", (char *) 0) < 0)
                err_sys("execl error");
   }

   if (waitpid(pid, NULL, 0) < 0)      /* parent */
        err_sys("waitpid error");
   exit(0);
}
```

Figure 8.20: results

```
$ cat /home/sar/bin/testinterp
#! /home/sar/bin/echoarg foo
$ ./a.out
argv[0]: /home/sar/bin/echoarg
argv[1]: foo
argv[2]: /home/sar/bin/testinterp
argv[3]: myarg1
argv[4]: MY ARG2
```

Figure 8.21: awk interpreter file (/usr/local/bin/awkexample)

```
#!/bin/awk -f

BEGIN {
   for (i = 0; i < ARGC; i++)
        printf "ARGV[%d] = %s\n", i, ARGV[i]
   exit
}
```

Figure 8.21: results

```
$ awkexample file1 FILENAME2 f3
ARGV[0]: /bin/awk
ARGV[1]: file1
ARGV[2]: FILENAME2
ARGV[3]: f3
```

- Actually:
  /bin/awk –f /usr/local/bin/awkexample file1 FILENAME2 f3

system Function

```
#include <stdlib.h>
int system ( const char *cmdstring );
```

- Return values:
- -1 with errno set if `fork` or `waitpid` fails
- 127 as if shell exit(127) if `exec` fails
- Termination status of shell if all 3 succeed

Figure 8.22: system

```
#include        <sys/wait.h>
#include        <errno.h>
#include        <unistd.h>

int system(const char *cmdstring)
                /* version without signal handling */
{
   pid_t        pid;
   int          status;

   if (cmdstring == NULL)
        return(1);  /* always a command processor with Unix */

   if ( (pid = fork()) < 0) {
        status = -1;   /* probably out of processes */
   }
```

Program 8.12: system

no need of breaking up cmdstring

to prevent child flushing buffer

```
    } else if (pid == 0) {              /* child */
        execl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
        _exit(127);             /* execl error */

    } else {                            /* parent */
        while (waitpid(pid, &status, 0) < 0)
          if (errno != EINTR) {
            status = -1; /* error other than EINTR */
            break;
          }
    }
    return(status);
}
```

Figure 8.23: calling system

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

int main(void) {
   int          status;

   if ( (status = system("date")) < 0)
        err_sys("system() error");
   pr_exit(status);

   if ( (status = system("nosuchcommand")) < 0)
        err_sys("system() error");
   pr_exit(status);

   if ( (status = system("who; exit 44")) < 0)
        err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```

Figure 8.23: results

```
$ ./a.out
Sun Mar 21 18:41:32 EST 2004
normal termination, exit status = 0 for date
sh: nosuchcommand: not found
normal termination, exit status = 127 for nosuchcommand
sar     :0          Mar 18 19:45
sar     pts/0       Mar 18 19:45 (:0)
sar     pts/1       Mar 18 19:45 (:0)
sar     pts/2       Mar 18 19:45 (:0)
normal termination, exit status = 44 for exit
```

Set User-ID Programs

- What happens if we call system from a set-user-ID program?
- A security hole!
- Should never be done!

Compile into program tsys

Figure 8.24: system from cmd

```
#include      "apue.h"

int
main(int argc, char *argv[])
{
   int        status;

   if (argc < 2)
        err_quit("command-line argument required");

   if ( (status = system(argv[1])) < 0)
        err_sys("system() error");
   pr_exit(status);

   exit(0);
}
```

Compile into program printuids

Figure 8.25: print UIDs

```
#include “apue.h"

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

Figures 8.24, 8.25: results

$ **tsys printuids**
real uid = 224, effective uid = 224
normal termination, exit status = 0

> make tsys set-user-ID

$ **su**
Password:
\# **chown root tsys**
\# **chmod u+s tsys**
\# **ls –l tsys**
-rwsrwxr-x 1 root 16361 Mar 16 16:59 tsys
\# **exit**

$ **tsys printuids**
real uid = 224, effective uid = 0 oops, this is a security hole
normal termination, exit status = 0

User Identification

```
#include <unistd.h>
char *getlogin(void);
```

- Returns: pointer to string giving login name if OK, NULL on error
- Used when a user has more than one login name
- User with getpwnam() to get the user information from passwd file.

Process Times

```
#include <sys/times.h>
clock_t times(struct tms *buf);
```

- Returns: elapsed wall clock time in clock ticks if OK, -1 on error

```
struct tms {
 clock_t tms_utime;  /* user CPU time */
 clock_t tms_stime;  /* system CPU time */
 clock_t tms_cutime; /* user time, term'd children */
 clock_t tms_cstime; /* system time, term'd children*/
};
```

Figure 8.30 (main())

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

static void  pr_times(clock_t, struct tms *, struct tms *);
static void  do_cmd(char *);

int main(int argc, char *argv[]) {
   int          i;

   setbuf(stdout, NULL);
   for (i = 1; i < argc; i++)
        do_cmd(argv[i]);    /*once each command-line arg */
   exit(0);
}
```

Figure 8.30 (do_cmd())

```
static void
do_cmd(char *cmd)         /*execute and time the "cmd" */
{
   struct tms    tmsstart, tmsend;
   clock_t       start, end;
   int           status;

   fprintf(stderr, "\ncommand: %s\n", cmd);

   if ( (start = times(&tmsstart)) == -1)
                                                  /* starting values */
        err_sys("times error");
   if ( (status = system(cmd)) < 0)         /* execute command */
        err_sys("system() error");

   if ( (end = times(&tmsend)) == -1)       /* ending values */
        err_sys("times error");

   pr_times(end-start, &tmsstart, &tmsend);
   pr_exit(status);
}
```

Figure 8.30 (pr_times())

```
static void pr_times(clock_t real, struct tms *tmsstart, struct tms
*tmsend)
{
  static long           clktck = 0;

  if (clktck == 0)      /* fetch clock ticks per second first time */
        if ( (clktck = sysconf(_SC_CLK_TCK)) < 0)
                err_sys("sysconf error");
  fprintf(stderr, "  real:  %7.2f\n", real / (double) clktck);
  fprintf(stderr, "  user:  %7.2f\n",
        (tmsend->tms_utime - tmsstart->tms_utime) / (double) clktck);
  fprintf(stderr, "  sys:   %7.2f\n",
        (tmsend->tms_stime - tmsstart->tms_stime) / (double) clktck);
  fprintf(stderr, "  child user:  %7.2f\n",
        (tmsend->tms_cutime - tmsstart->tms_cutime) / (double) clktck);
  fprintf(stderr, "  child sys:   %7.2f\n",
        (tmsend->tms_cstime - tmsstart->tms_cstime) / (double) clktck);
}
```

Figure 8.30: results

$ **./a.out "sleep 5" "date"**
command: sleep 5
real: 5.02
user: 0.00
sys: 0.00
child user: 0.01
child sys: 0.00
normal termination, exit status = 0

command: date
Mon Mar 22 00:43:58 EST 2004
real: 0.01
user: 0.00
sys: 0.00
child user: 0.01
child sys: 0.00
normal termination, exit status = 0