# vfork Function

- *vfork* has the same calling sequence and same return values as *fork*.
- *vfork* is intended to create a new process when the purpose of the new process is to *exec* a new program.
- *vfork* creates the new process, just like *fork*, without fully copying the address space of the parent into the child, since the child won’t reference that address space - the child just calls *exec* (or *exit*) right after the *vfork*.
- Instead, while the child is running, until it calls either *exec* or *exit*, the child runs in the address space of the parent.
- Another difference between the two functions is that *vfork* guarantees that the child runs first, until the child calls *exec* or *exit*.

# Example of vfork Function

```
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 continues here.
     */
    printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var);
    exit(0);
}
```

## Vfork Example (contd…)

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

- Note that _exit is called in child instead of exit
- _exit does not perform any flushing of standard I/O buffers.
- If call exit instead, the results are indeterminate. Depending on the implementation of the standard I/O library, there might be no difference in the output, or might find that the output from the parent's printf disappear

## Process Termination

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent.
- Because the termination of a child is an asynchronous event, it can happen at any time while the parent is running.
- When SIGCHLD signal is delivered to the parent.
  - The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.
  - The default action for this signal is to be ignored.

# Algorithm for Exit

```
algorithm exit
input : return code for parent process
output : none
{
    ignore all signals;
    if ( process group leader with associated control terminal )
    {
          send hangup signal to all members of process group;
          reset process group for all members to 0;
    }
    close all open files;
    release current directory;
    release current(changed) root, if exists;
    free regions, memory associated with process;
    write accounting record;
    make process state zombie;
    assign parent process ID of all child processes to be init process(1);
          if any children were zombie, send death of child signal to init;
    send death of child signal to parent process;
    context switch;
}
```

# Wait Function

pid_t wait(int **statloc*);

- return: process ID of child if OK, or –1 on error
- argument statloc is a pointer to an integer.
  - If not a null pointer, the termination status of the terminated process is stored in the location pointed to by the argument
  - If don't care about the termination status, simply pass a null pointer as this argument
- Process that calls wait can
  - Block, if all of its children are still running
  - Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
  - Return immediately with an error, if it doesn't have any child processes

# waitpid

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

return: process ID of child if OK, or –1 on error,

0 with WNOHANG option

- The *pid* argument for *waitpid* depends on its value:
  - *pid* = = -1 waits for any child process. In this respect, *waitpid* is equivalent to *wait*.
  - *pid* > 0 waits for the child whose process ID equals *pid*.
  - *pid* = = 0 waits for any child whose process group ID equals that of the calling process.
  - *pid* < -1 waits for any child whose process group ID equals the absolute value of *pid*.

# waitpid (Contd…)

- options for waitpid
  - WCONTINUED: If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned
  - WNOHANG: The waitpid function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
  - WUNTRACED: If the implementation supports job control, the status of any child specified by pid that has stopped and whose status has not been reported since it has stopped, is returned. The WIFSTOPPED macro determines whether the return value corresponds to a stopped child process

# Algorithm for Wait

```
algorithm wait
input : address of variables to store status of exiting process
output : child-pid, child –exit-code{
    if (waiting process has no child process)
        return(error);
    for(;;)
    {   if (waiting process has zombie child)
        {       pick arbitrary zombie child;
                add child CPU usage to parent;
                free child process table entry;
                return(child-pid, child-exit-code);
        }
        sleep at interruptible priority(event child process exits);
    }
}
```

## Macros Related to wait

- Macros to examine the termination status returned by wait and waitpid
- WIFEXITED(status): True if status was returned for a child that terminated normally
  - Can execute WEXITSTATUS (status) to fetch the low-order 8 bits of the argument that the child passed to exit, _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
  - Execute WTERMSIG (status) to fetch the signal number that caused the termination
  - WCOREDUMP (status) that returns true if a core file of the terminated process was generated.

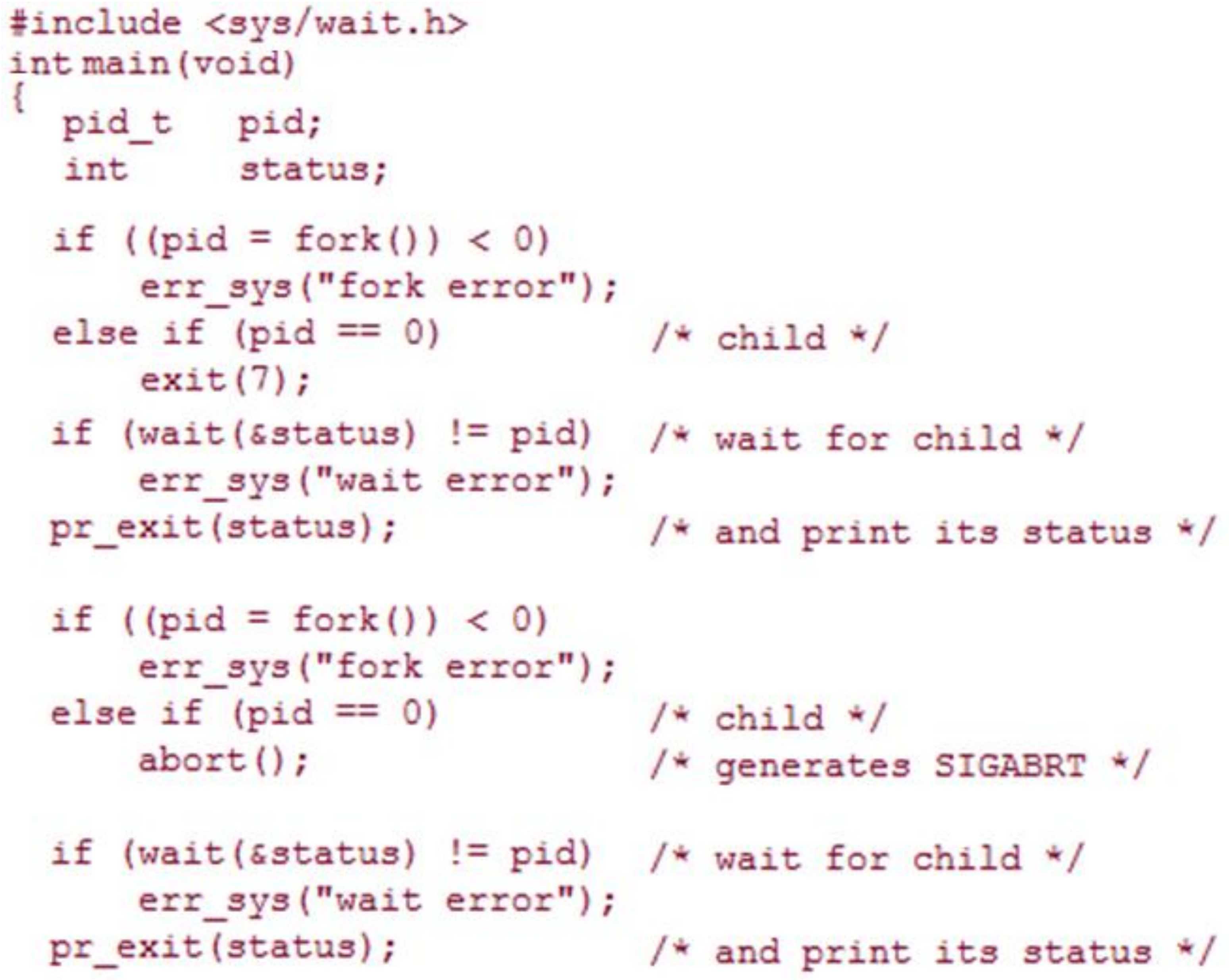
## Macros Related to wait

- WIFSTOPPED (status): True if status was returned for a child that is currently stopped
  - 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

# Print a Description of Exit Status

```c
#include <sys/wait.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));
}
```

# Demonstrate Various Exit Statuses



```
#include <sys/wait.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 */

  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 */
```

# (Contd…)

```
    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);
}
```

```
$ ./a.out
normal termination, exit status = 7
abnormal termination, signal number = 6 (core file generated)
abnormal termination, signal number = 8 (core file generated)
```

Signal number = 6 for SIGABRT and 8 for SIGFPE

## Difference b/w wait and waitpid

- Wait() can block the caller until a child process terminates, whereas waitpid() has an option that prevents it from blocking
- Waitpid() doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.
  - Waitpid lets us wait for one particular process, whereas the wait function returns the status of any terminated child.
  - Waitpid provides a non-blocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.
  - waitpid provides support for job control with the WUNTRACED and WCONTINUED options

# Special Exit Cases

Two special cases:

1) A child exits when its parent is not currently executing `wait()`

- the child becomes a *Zombie*
- `status` data about the child is stored in process table until the parent does a `wait()`

2) A parent exits when one or more children are still running

- Children are called *Orphan*
- children are adopted by init (`/etc/init`)
  - it can then monitor/kill them

# Troubles from Zombie

- Although Zombie processes do not occupy any memory space in the system, it consumes <u>process IDs</u>, which are limited resources too
- CHILD_MAX: max number of simultaneous processes per real user ID
- May not be able to fork new child processes when there are too many zombie processes in the systems
- How to fork a new child process but
  - not asking the parent process to wait for the child AND
  - not generating zombie process?

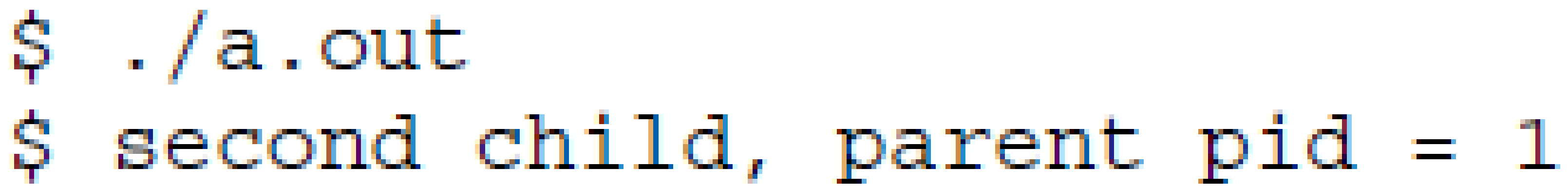
## Avoid Zombie Processes

- Create a process so that the parent forks a child but parent doesn't want to wait for the child to complete and doesn't want the child to become a zombie
  - Trick is to call fork twice

# Avoid Zombie Processes

```
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) exit(0);
             /* the second child; init becomes parent */
             sleep(2);
             printf("second child, parent pid = %d\n", getppid());
             exit(0);
             }
   if (waitpid(pid, NULL, 0) != pid)  err_sys("waitpid error");
   /* the parent (the original process)*/
   exit(0);
}
```

# Avoid zombie processes - Output



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

- Note that the shell prints its prompt when the original process terminates, which is before the second child prints its parent process ID.
- Init runs wait() periodically to reap out zombies

# RACE Conditions

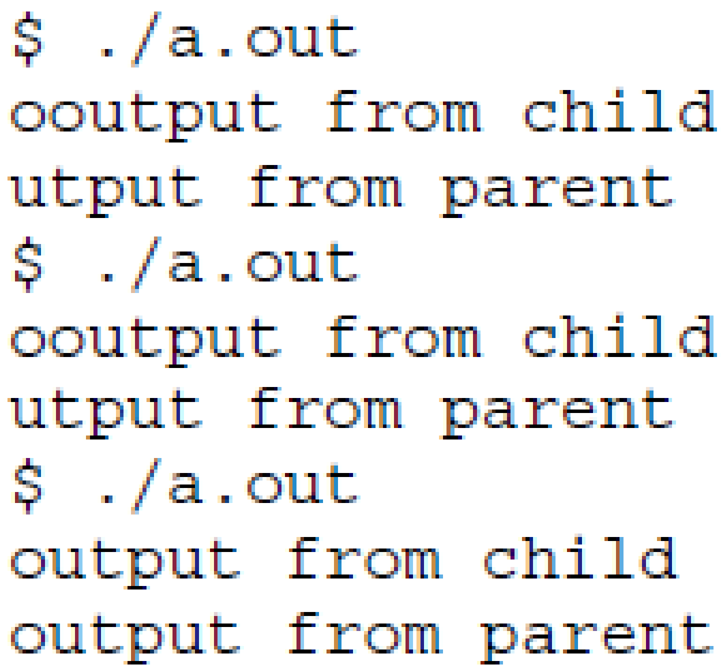
- Race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run
- fork function is a source for race condition depends on whether the parent or child runs first after the fork
- Cannot predict which process 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

```
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);
 }
static void charatatime(char *str)
 { char *ptr; int c;
    setbuf(stdout, NULL); /* set unbuffered */
   for (ptr = str; (c = *ptr++) != 0; ) putc(c, stdout);
}
```

## Output

- Standard output is set unbuffered, so every character output generates a call to write function.
- Kernel switches between the two processes depends on scheduling
- Output is unpredictable



```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

## Avoid Race conditions

- A process that wants to wait for a child to terminate must call one of the wait functions.
- If a process wants to wait for its parent to terminate, a loop of the following form could be used:

```
        while (getppid() != 1)
                sleep(1);
```

- The problem with loop: called polling, wastes CPU time, as the caller is awakened every second to test the condition.

## Avoid Race conditions (cont.)

- To avoid race conditions and to avoid polling, some form of signaling is required between multiple processes.
- Various forms of inter process communication (IPC) can also be used.
- Introduce new functions
  - WAIT_PARENT(), TELL_CHILD(), WAIT_CHILD(), TELL_PARENT()

# Model for Coordination

Requirement for proper coordination is to allow each process to “report” to other processes when it has finished performed a given action needed by the other process.

```
TELL_WAIT();   /* setup TELL_xxx and WAIT_xxx */
if ((pid=fork()) < 0) err_sys(“fork error”);
else if (pid==0) {
   ……… child does necessary action ………..
   TELL_PARENT(getppid());                    /* tell parent we finished*/
   WAIT_PARENT();                    /* and wait for parent*/
   …………..
   exit(0);
}
……… parent does necessary action ………..
TELL_CHILD(pid);  /* tell child we finished*/
WAIT_CHILD();                 /* and wait for child*/
………..
exit(0)
}
```

# Updated program without Race Condition

```
//parent runs first
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);
}
static void charatatime(char *str)
{           char        *ptr;
            int         c;
            setbuf(stdout, NULL);                   /* set unbuffered */
            for (ptr = str; c = *ptr++; )  putc(c, stdout);
}
```

## exec() functions

- fork function creates a new process (child)
- Child may cause another program to be executed by calling one of the exec functions
- When a process calls one of the exec functions, that process is completely replaced by the new program and the new program starts executing at its main function
- Process ID does not change across an exec because a new process is not created
- exec replaces the current process’s text, data, heap and stack segments with a new program from disk

# Algorithm for exec()

```
input : file name, parameter list, environment variables list
output : none
{
    get file inode;
    verify file executable, user has permission to execute;
    read file headers, check that it is a load module;
    copy exec parameters from old address space to system space;
    for(every region attached to process)
            detach all old regions;
    for(every region specified in load module)
    {       allocate new regions;
            attach the regions;
            load region into memory if appropriate;
    }
    copy exec parameters into new user stack region;
    special processing for setuid programs, tracing;
    initialize user register save area for return to user mode;
    release inode of file(input);
}
```

## exec() functions (cont.)

- 6 different exec functions, return: –1 on error, no return on success

```
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 []);
```

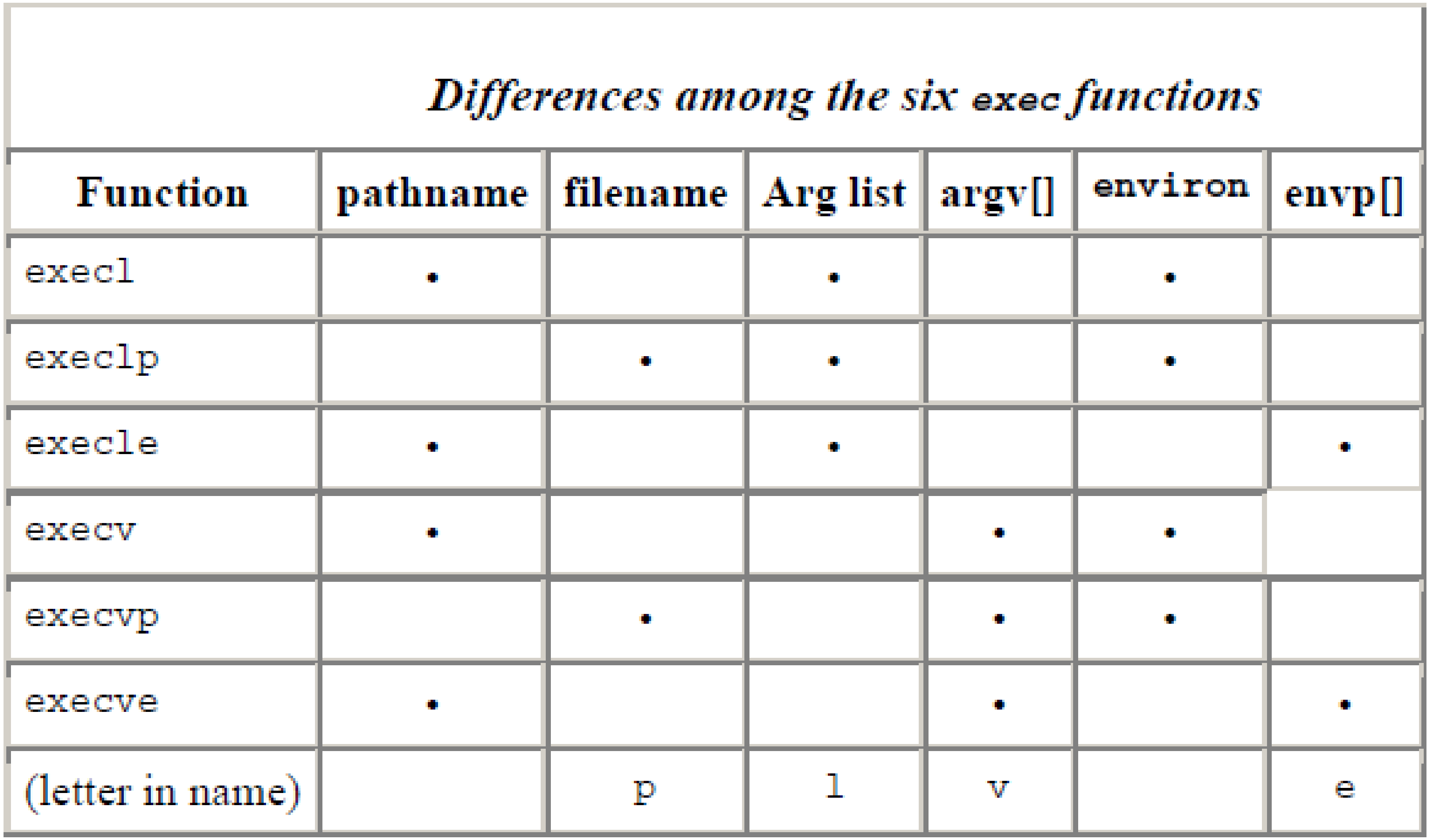
## exec() Functions Differences

- Path name/filename
  - execl, execv, execle, execve take a pathname argument, whereas execlp, execvp take a filename argument
  - If filename contains a slash, it is taken as a pathname
  - Otherwise, the executable filename is searched for in the directories specified by the PATH environment variable
    - PATH=/bin:/usr/bin:/usr/local/bin/:.
- passing of the argument list
  - execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments.
  - execv, execvp, and execve, have to build an array of pointers to the arguments and the address of this array is the argument to these three functions.

## exec() Functions Differences

- Passing of the environment list to the new program.
  - execle and execve functions whose name ends in an e, allow to pass a pointer to an array of pointers to the environment strings
  - Other four functions, use the ***environ*** variable in the calling process to copy the existing environment for the new program

# Differences among the exec() Functions

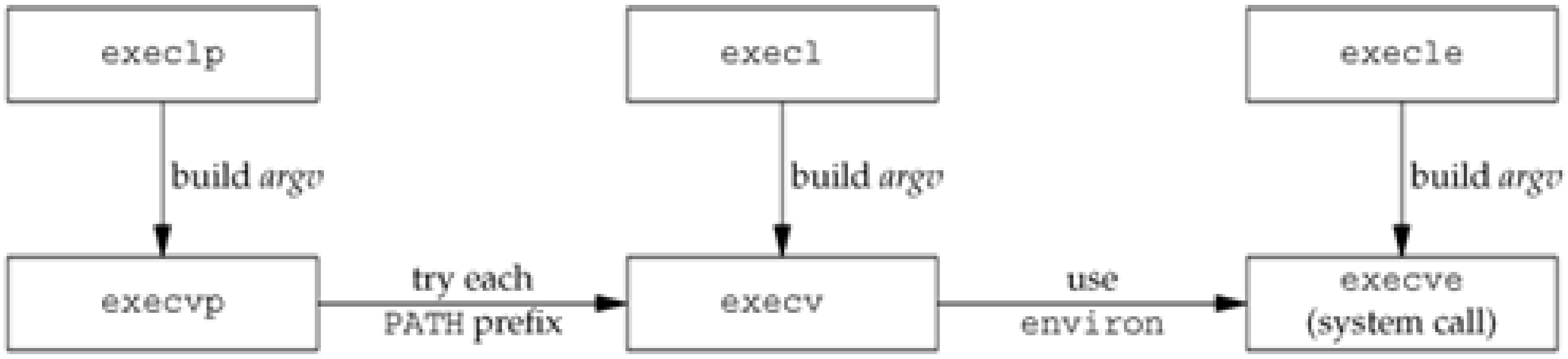


*Differences among the six `exec` functions*

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

# Relationship of the six exec()

- Only one of these six functions, execve, is a system call within the kernel.
- The other five are just library functions that eventually invoke this system call


$$\begin{tikzcd}
\texttt{execlp} \arrow[d, "\text{build } argv"] & \texttt{execl} \arrow[d, "\text{build } argv"] & \texttt{execle} \arrow[d, "\text{build } argv"] \\
\texttt{execvp} \arrow[r, "\text{try each}", "\text{PATH prefix}"'] & \texttt{execv} \arrow[r, "\text{use}", "\texttt{environ}"'] & \texttt{execve} \text{ (system call)}
\end{tikzcd}$$

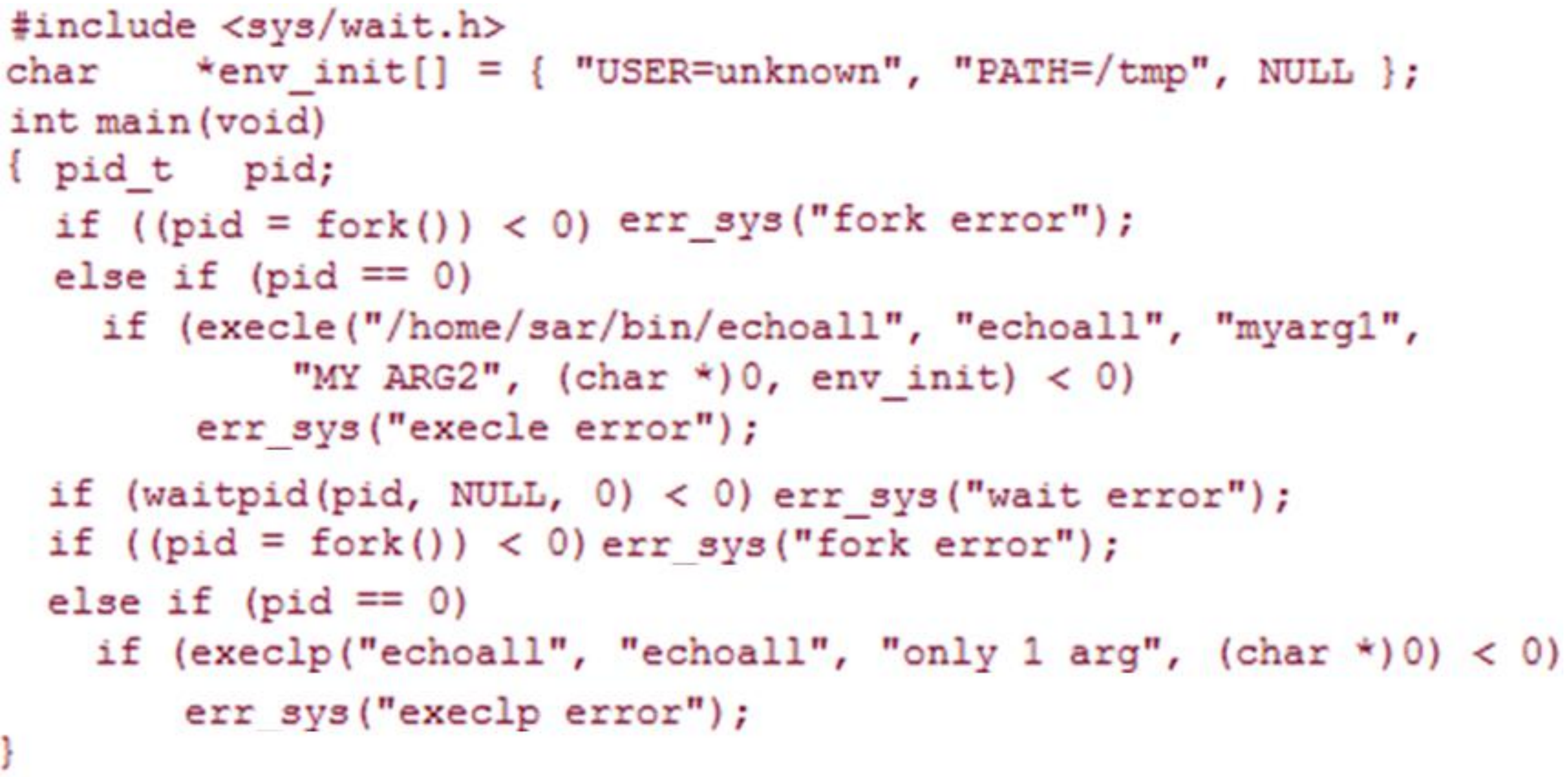
## Properties Inherited from Calling Process

- Process ID and parent process ID
- Real user ID and real group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- Time left until alarm clock
- Current working directory
- Root directory
- File mode creation mask
- File locks
- Process signal mask
- Pending signals
- Resource limits
- Values for tms_utime, tms_stime, tms_cutime, and tms_cstime

# Other Important Points

- Handling of open files depends on the value of the close-on-exec flag for each descriptor.
  - If this flag is set, the descriptor is closed across an exec.
  - Otherwise, the descriptor is left open across the exec.
- Real user ID and the real group ID remain the same across the exec
- Effective IDs can change, depending on the status of the set-user-ID and the set- group-ID bits for the program file that is executed.
  - If the set-user-ID bit is set for the new program, the effective user ID becomes the owner ID of the program file. Otherwise, the effective user ID is not changed

# exec() – Example - 1



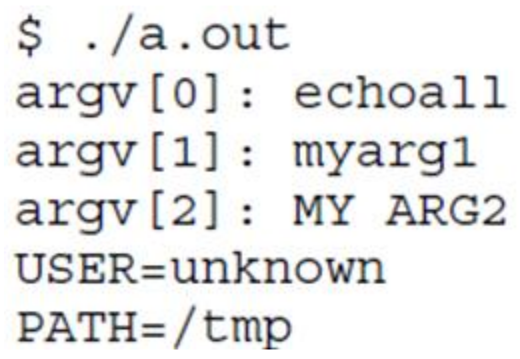
```
#include <sys/wait.h>
char    *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{ pid_t   pid;
  if ((pid = fork()) < 0) err_sys("fork error");
  else if (pid == 0)
    if (execle("/home/sar/bin/echoall", "echoall", "myarg1",
            "MY ARG2", (char *)0, env_init) < 0)
        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)
    if (execlp("echoall", "echoall", "only 1 arg", (char *)0) < 0)
        err_sys("execlp error");
}
```

## exec() – Example -1 (Contd)

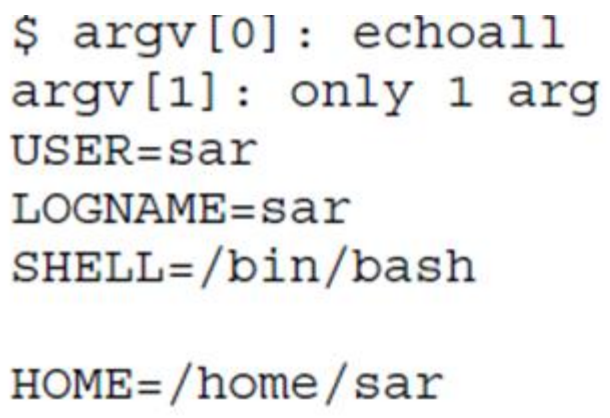
***Echo all Program***

```
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);
}
```

# exec() – Example - 1 - output



```
$ ./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
```

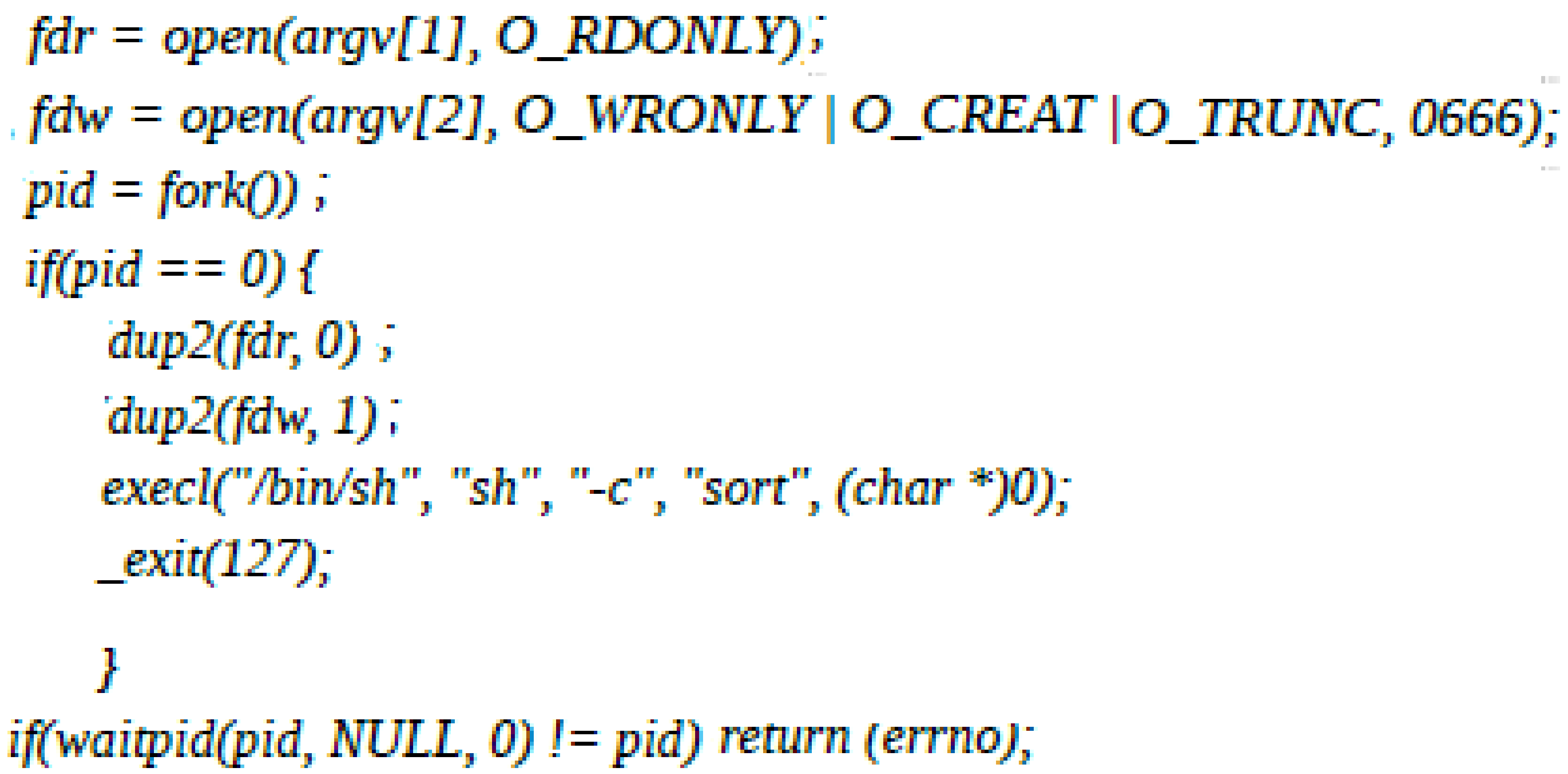
# exec() – Example -2

- This program works like a shell. On the prompt the user enters a command, it should be read and executed. Donot use system API. The program quits, when “logout: is entered.

```
int main() {
   printf("\n$ ");
   while(1) {
      // read cmdstring[i]
        if(strcmp(cmdstring, "logout") == 0)
                 break;
       if((pid = fork()) < 0) return (errno);
       if(pid == 0) {  /*child process*/
               execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
               _exit(127);     /*execl error*/
      if(waitpid(pid, NULL, 0) != pid) return (errno);
      }
}
```

## exec() – Example -3

- Emulate sort <foo >result



```
fdr = open(argv[1], O_RDONLY);
fdw = open(argv[2], O_WRONLY | O_CREAT |O_TRUNC, 0666);
pid = fork()) ;
if(pid == 0) {
     dup2(fdr, 0) ;
     dup2(fdw, 1) ;
     execl("/bin/sh", "sh", "-c", "sort", (char *)0);
     _exit(127);

     }
if(waitpid(pid, NULL, 0) != pid) return (errno);
```

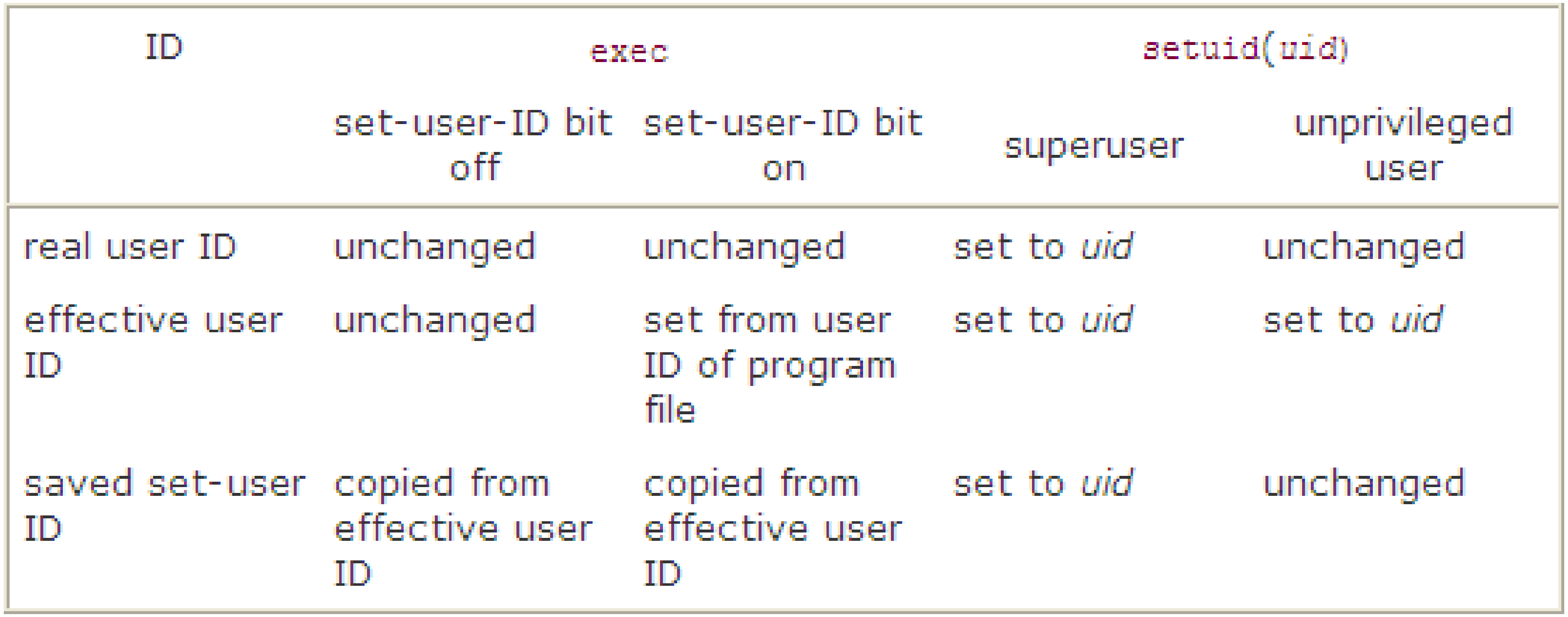
## User/Group ID’s

- Only a superuser process can change the real user ID. Normally, the real user ID is set by the login program when log in and never changes. Because login is a superuser process, it sets all three user IDs when it calls setuid()
- euid is set by exec only if the setuid bit is set for the program file
- saved set-user-ID is copied from the effective user ID by exec
  - If the file's set-user-ID bit is set, this copy is saved after exec stores the effective user ID from the file's owner ID
- Calling setuid() at any time can set the effective user ID to either the real user ID or the saved set-user-ID

## User/Group ID's

- int setuid(uid_t uid);
  - The process == superuser then set real/effective/saved-suid = *uid*
  - Otherwise, euid=*uid if uid == ruid or uid* == saved-suid
  - Or errno=EPERM (_POSIX_SAVED_IDS)
- int setgid(gid_t gid);
  - The same for group as setuid

# Ways to change the 3 user IDs



| 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 |

# User ID - Example

- Example *tip (BSD) to switch between permissions of two users*
  - The setuid bit is on for *tip (owner=uucp).*
    - When we exec it
      - Ruid = ja
      - Euid = uucp
      - Ssuid = uucp
  - *tip* access the required lock file, owned by uucp. Since Euid = uucp, access is allowed
  - *tip* calls *setuid(Ruid).* Since we are not superuser,
      - Ruid = ja
      - Euid = ja
      - Ssuid = uucp
  - Now we can access only the file that we have normal access to. No additional permissions.
  - *tip* calls *setuid(uucp).* This call is allowed as argument of *setuid* is saved set-user-ID
      - Ruid = ja
      - Euid = uucp
      - Ssuid = uucp
  - Now we can operate on locked files.

# User/Group ID’s

int setreuid(uid_t ruid, uid_t euid);

- unprivileged user can swap real user ID and the effective user ID
- Allows a set-user-ID program to swap to the user's normal permissions and swap back again later for setuser-ID operations
- BSD only or BSD-compatibility library

int setregid(uid_t rgid, uid_t egid);

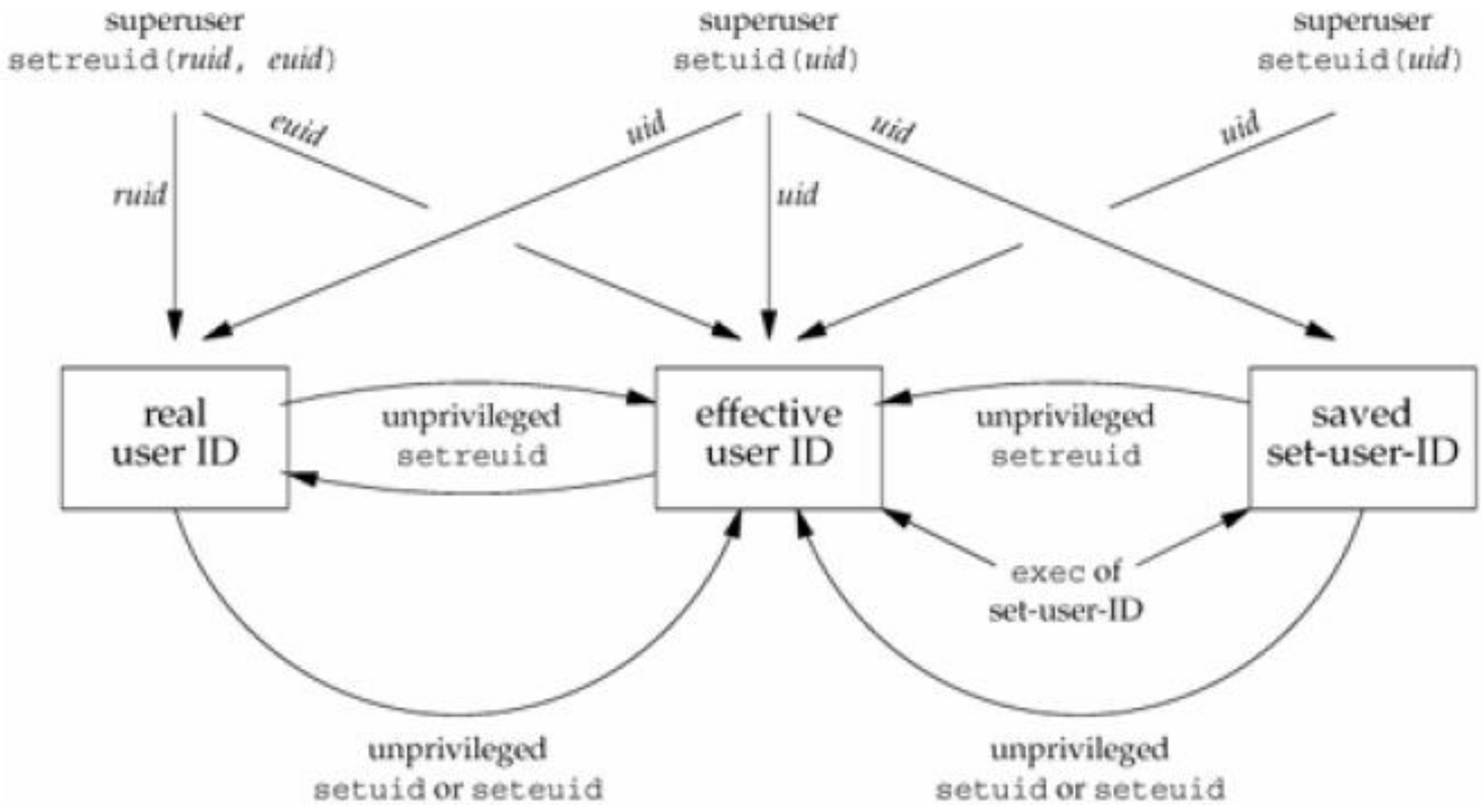
## User/Group ID’s

int seteuid(uid_t uid);

int setegid(uid_t gid);

- Non-superusers can only seteuid=ruid or saved-setuid
- A privileged user only sets euid = uid

# User/Group ID's



## Interpreter File

- Is a file begins with a line of the form:
  #! pathname [optional-argument]
  - E.g., “#! /bin/sh”
- Pathname specified on the first line of the interpreter file is called “Interpreter”
- Implementation:
  - Recognition is done within the kernel
  - Interpreter vs the interpreter files
  - Line-limit of the first line, e.g., 32 chars
- The actual file that gets executed by the kernel is not the interpreter file, but the interpreter.

- cat /home/testinterp

  ```
  #! /home/echoarg foo
  ```

- A program that exec an interpreter file

```
 int main (void)
 {   if ((pid = fork())< 0 )  err_sys("fork error");
     else if (pid == 0) { /*child*/
     if (execl("home/testinterp", "testinterp", "myarg1", "MYARG2", (char *)0)< 0)
                     err_sys("exec error");
     }
     if (waitpid(pid, NULL, 0) < 0) /*parent*/            err_sys("waitpid error");
     exit(0);
 }
```

- Result of execution

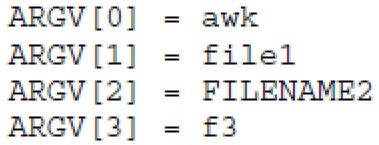
```
$ a.out
argv[0] : /home/echoarg
argv[1] :foo
argv[2] :/home/testinterp
argv[3] :myarg1
argv[4] :MYARG2
```

# Usage of Interpreter file

- "awk –f myfile" lets awk to read an awk program from "myfile".
- awkexample Program

```
#! /bin/awk –f
BEGIN {
   for (i = 0; i < ARGC; i++)
            printf "ARGV[%d] = %s\n", i, ARGV[i]
exit
}
```

- $ awkexample file1 FILE2 f3 (at /usr/local/bin/)



```
ARGV[0] = awk
ARGV[1] = file1
ARGV[2] = FILENAME2
ARGV[3] = f3
```

- When /bin/awk is executed, the command line arguments are /bin/awk –f /usr/local/bin/awkexample file1  FILE2  f3
- awk ignores the first line since the pound sign is awk's comment character

# Interpreter file - Exmple

- We can verify the results

```
$su
Passwd:
# mv /bin/awk      /bin/awk.save
# cp /home/echoarg    /bin/awk
# suspend
[1] + Stopped su
$awkexample file1  FILE2  f3
argv[0]: /bin/awk
argv[1]: -f
argv[2]: /usr/local/bin/awkexample
argv[3]: file1
argv[4]: FILE2
argv[5]: f3
$ fg
su
#mv /bin/awk.save /bin/awk
#exit
```

## Example

- Removing of “-f” from interpreter file

```
$awkexample file1 FILE2 f3
/bin/awk: syntax error at source line 1
```

- This is because the command line arguments in this case are

```
/bin/awk  /usr/local/bin/awkexample file1  FILE2  f3
```

- awk is trying to interpret the string /usr/local/bin/awkexample as an awk program

# Why interpreter files?

- Hide the fact that certain programs are scripts in other languages.
- Interpreter scripts allows to write shell scripts using other than /bin/sh
- Interpreter scripts provide an efficiency gain.  We could still hide that the program is an awk script, by wrapping it in a shell script:

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

  – More work is required with this solution.
  – First, the shell reads the command and tries to execlp the filename. Because the shell  script is an executable file, but isn't a machine executable, an error is returned,
  – Then execlp assumes that the file is a shell script. Then /bin/sh is executed with the pathname of the shell script as its argument. The shell correctly runs our script
  – But to run the awk program, the shell does a fork, exec and wait
  – There is more overhead in replacing an interpreter script with a shell script.

# System()

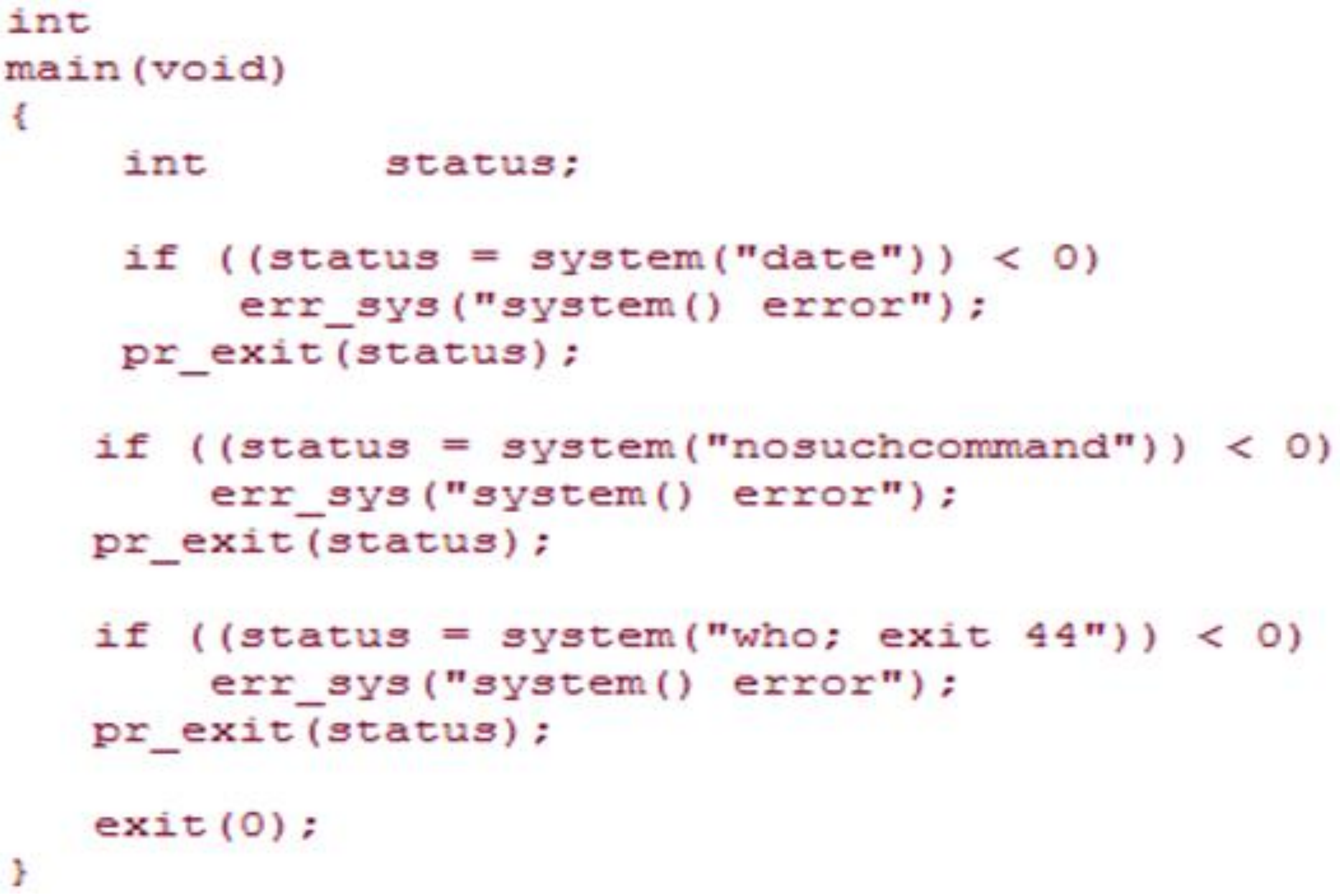
- int system(const char *cmdstring);

  Returns: (see below)
- If cmdstring is a null pointer, system returns nonzero only if a command processor is available
  - Used to determine whether the system() is supported on a given operating system.
  - Under the UNIX , system() is always available.
- Because system is implemented by calling fork, exec, and waitpid, there are three types of return values.
  - If either the fork fails or waitpid returns an error other than EINTR, system returns -1 with errno set to indicate the error.
  - If the exec fails, implying that the shell can't be executed, the return value is as if the shell had executed exit(127)
  - Otherwise, return value is the termination status of the shell
- Eg: system(“date > file”);
- Advantage: system() does all the error handling/ signal handling

## system function without signal handling

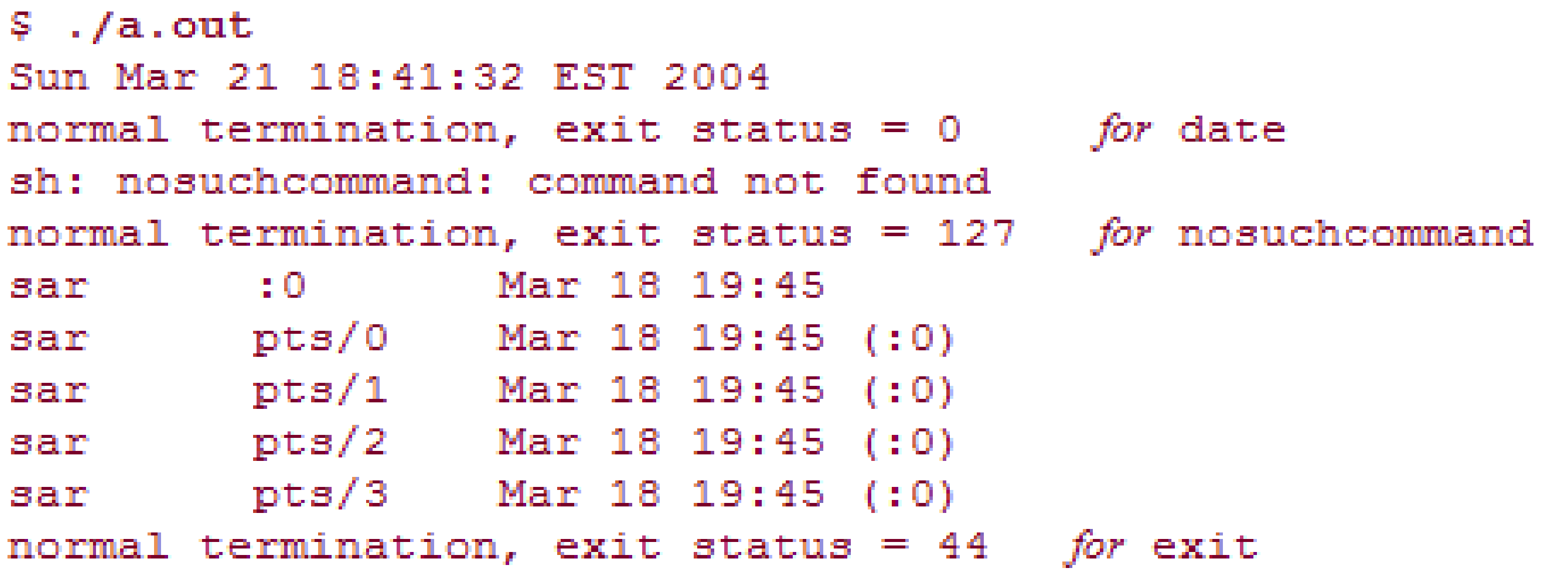
```
int system(const char *cmdstring) /* version without signal handling */
{   pid_t pid; int status;
    if (cmdstring == NULL)   return(1); /* command processor is with UNIX */
    if ((pid = fork()) < 0)   status = -1; /* probably out of processes */
    else if (pid == 0) { /* child */
            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 from waitpid() */
                            break;
                    }
            }
    }
    return(status);
}
```

## Eg: Calling system Function



```
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);
}
```

# Example - Output



```
$ ./a.out
Sun Mar 21 18:41:32 EST 2004
normal termination, exit status = 0       for date
sh: nosuchcommand: command 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)
sar       pts/3     Mar 18 19:45 (:0)
normal termination, exit status = 44     for exit
```

# A security Hole Example

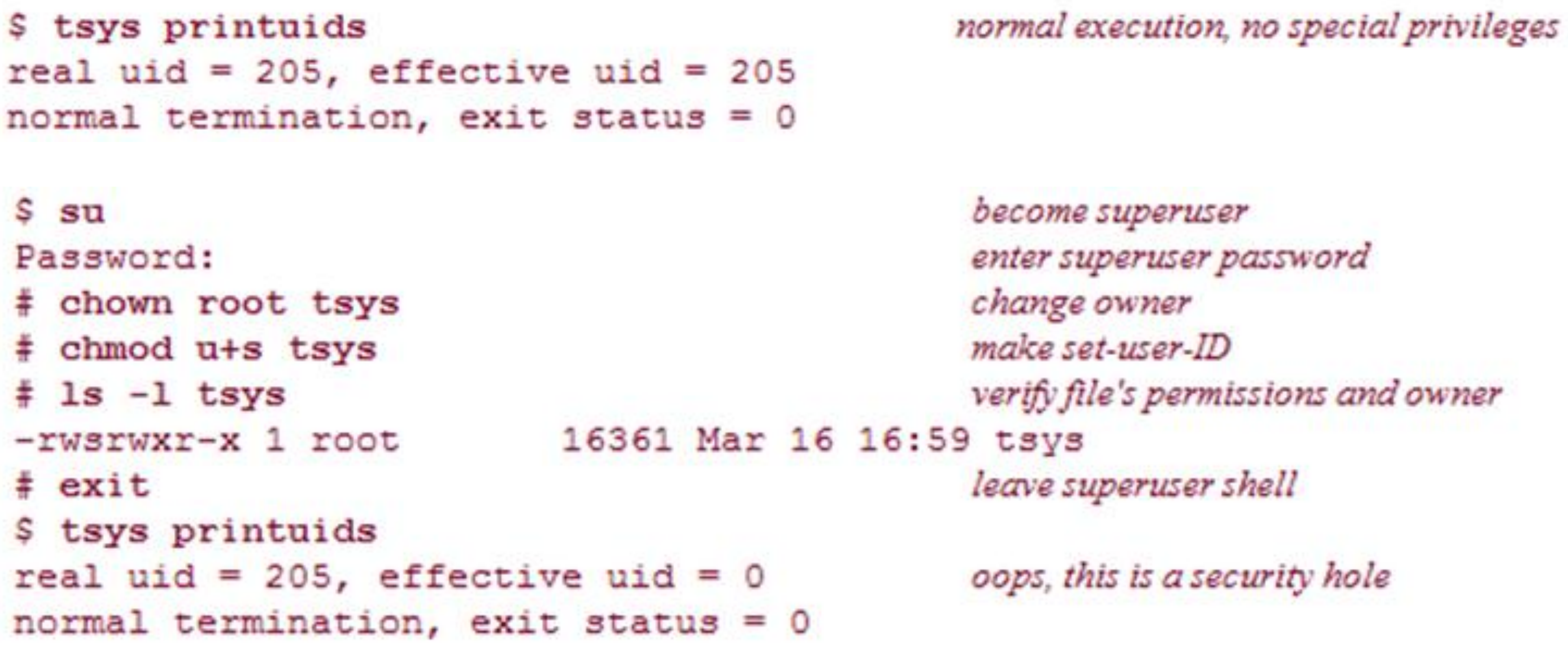
```
int main(int argc, char *argv[])
{ int     status;
  if (argc < 2) err_quit("command-line argument required");
  if ((status = system(argv[1])) < 0) err_sys("system() error");
  pr_exit(status);
  exit(0);
}
```

- We compile this program into executable file *tsys*

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

- We compile this program into executable file *printuids*

# A security Hole Example (Contd)



```
$ tsys printuids                                   normal execution, no special privileges
real uid = 205, effective uid = 205
normal termination, exit status = 0

$ su                                               become superuser
Password:                                          enter superuser password
# chown root tsys                                  change owner
# chmod u+s tsys                                   make set-user-ID
# ls -l tsys                                       verify file's permissions and owner
-rwsrwxr-x 1 root       16361 Mar 16 16:59 tsys
# exit                                             leave superuser shell
$ tsys printuids
real uid = 205, effective uid = 0                  oops, this is a security hole
normal termination, exit status = 0
```

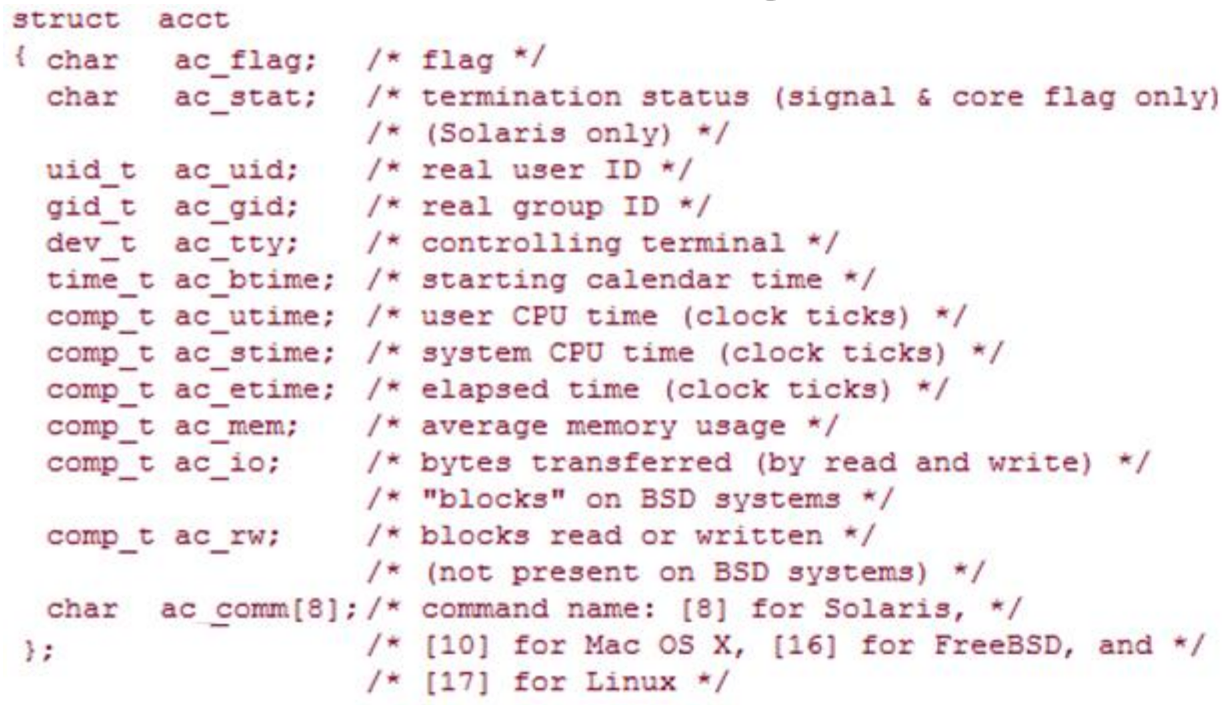
- The superuser permissions given to *tsys* are retained across *fork* and *exec* done by system()
- A set[ug]id program should change its uid's back to the normal after calling fork

```
setuid(getuid());   /* getuid() returns the real uid */
```

# Process Accounting

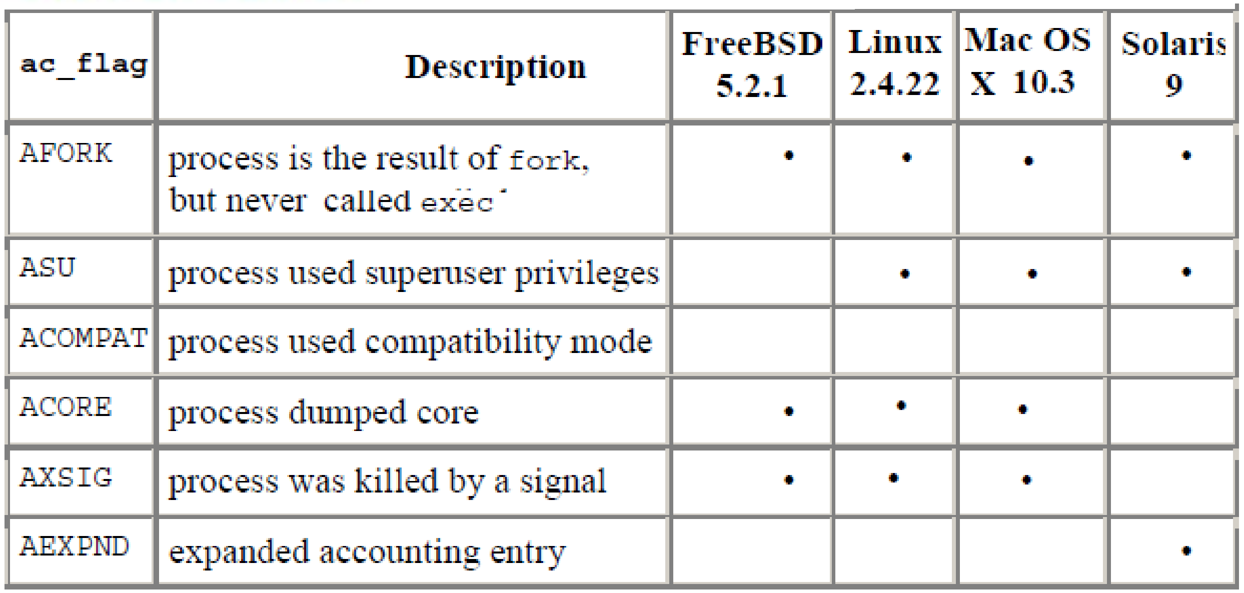
- A superuser executes **accton with a pathname** argument to enable accounting
- Accounting records correspond to processes, not programs
- A new record is initialized by the kernel for the child after a fork, not when a new program is executed
- Records are kept in the process table whenever a new process is created
- Each accounting record is written into the accounting file in the order of the termination order of processes.
  - Accounting file is usually /var/account/acct on FreeBSD and Mac OS X, /var/account/pacct on Linux, and /var/adm/pacct on Solaris.
- Accounting is turned off by executing **accton** without any arguments

# Accounting Record Structure



```
struct  acct
{ char   ac_flag;   /* flag */
  char   ac_stat;   /* termination status (signal & core flag only)
                    /* (Solaris only) */
  uid_t  ac_uid;    /* real user ID */
  gid_t  ac_gid;    /* real group ID */
  dev_t  ac_tty;    /* controlling terminal */
  time_t ac_btime;  /* starting calendar time */
  comp_t ac_utime;  /* user CPU time (clock ticks) */
  comp_t ac_stime;  /* system CPU time (clock ticks) */
  comp_t ac_etime;  /* elapsed time (clock ticks) */
  comp_t ac_mem;    /* average memory usage */
  comp_t ac_io;     /* bytes transferred (by read and write) */
                    /* "blocks" on BSD systems */
  comp_t ac_rw;     /* blocks read or written */
                    /* (not present on BSD systems) */
  char  ac_comm[8]; /* command name: [8] for Solaris, */
 };                 /* [10] for Mac OS X, [16] for FreeBSD, and */
                    /* [17] for Linux */
```

*Values for* `ac_flag`



| ac_flag | Description | FreeBSD 5.2.1 | Linux 2.4.22 | Mac OS X 10.3 | Solaris 9 |
|---|---|---|---|---|---|
| AFORK | process is the result of fork, but never called exec | • | • | • | • |
| ASU | process used superuser privileges | | • | • | • |
| ACOMPAT | process used compatibility mode | | | | |
| ACORE | process dumped core | • | • | • | |
| AXSIG | process was killed by a signal | • | • | • | |
| AEXPND | expanded accounting entry | | | | • |

# Print selected fields from accounting file

```
#include <sys/acct.h>
#define ACCFILE “/var/adm/pacct”
#define FMT "%-*.*s e = %6ld, chars = %7ld, stat = %3u: %c %c %c %c\n“
static unsigned long compt2ulong(comp_t comptime) /* convert comp_t to unsigned long */
{    unsigned long val; int exp;
     val = comptime & 0x1fff; /* 13-bit fraction */
     exp = (comptime >> 13) & 7; /* 3-bit exponent (0-7) */
     while (exp-- > 0)   val *= 8;
     return(val);
}
int main(int argc, char *argv[])
{    struct acct acdata; FILE *fp;
     if ((fp = fopen(argv[1], "r")) == NULL)   err_sys("can't open %s", argv[1]);
     while (fread(&acdata, sizeof(acdata), 1, fp) == 1) {
                 printf(FMT, (int)sizeof(acdata.ac_comm), (int)sizeof(acdata.ac_comm), acdata.ac_comm,
     compt2ulong(acdata.ac_etime), compt2ulong(acdata.ac_io), #ifdef HAS_SA_STAT (unsigned char)
     acdata.ac_stat, #endif acdata.ac_flag & ACORE ? 'D' : ' ', acdata.ac_flag & AXSIG ? 'X' : ' ', acdata.ac_flag
     & AFORK ? 'F' : ' ', acdata.ac_flag & ASU ? 'S' : ' ');
     }
     exit(0);
}
```

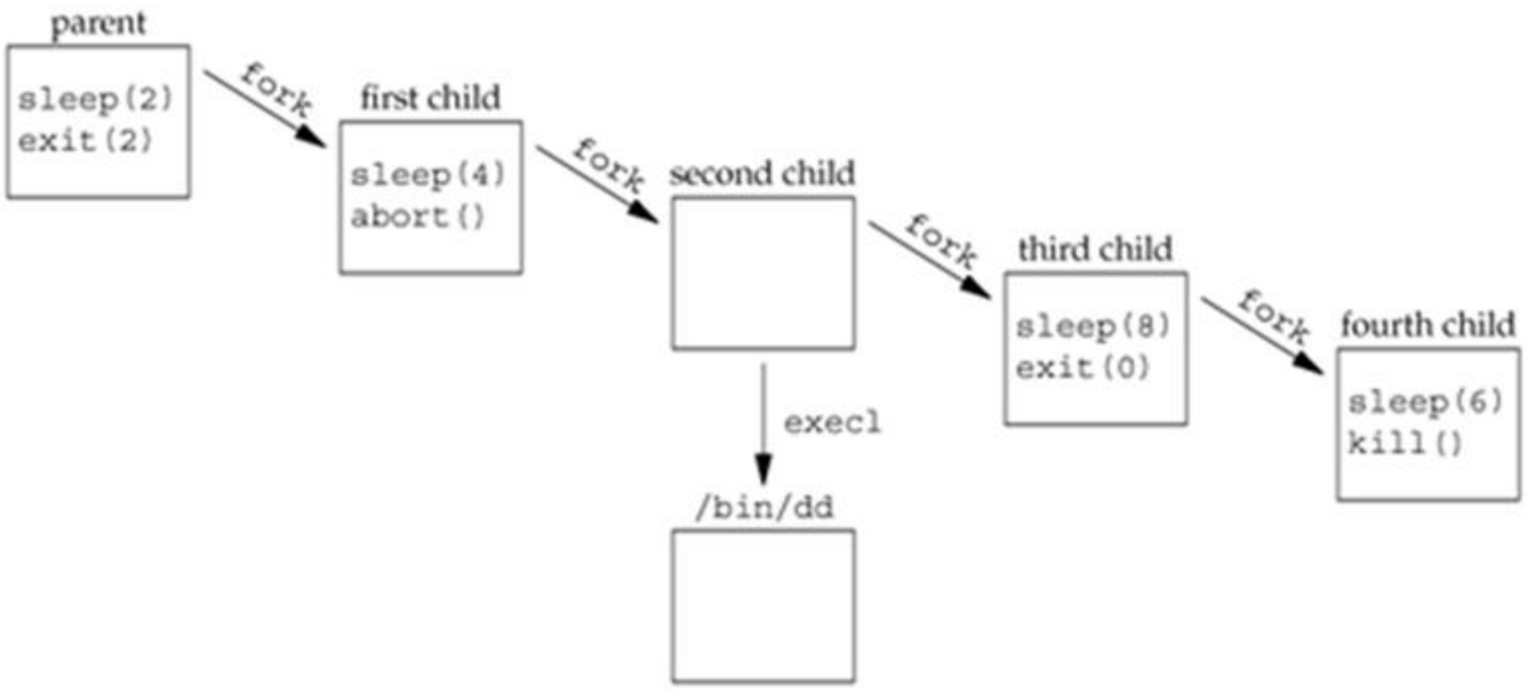
# Process Accounting (Contd)

- To perform the test, do the following:
  - Become superuser and enable accounting, with the accton command
  - when this command terminates, accounting should be on; therefore, the first record in the accounting file should be from this command
  - Exit the superuser shell
  - Run the program
  - Become superuser and turn accounting off.
  - Since accounting is off when this accton command terminates, it should not appear in the accounting file
  - Run the program to print the selected fields from the accounting file

# Process Accounting - Example

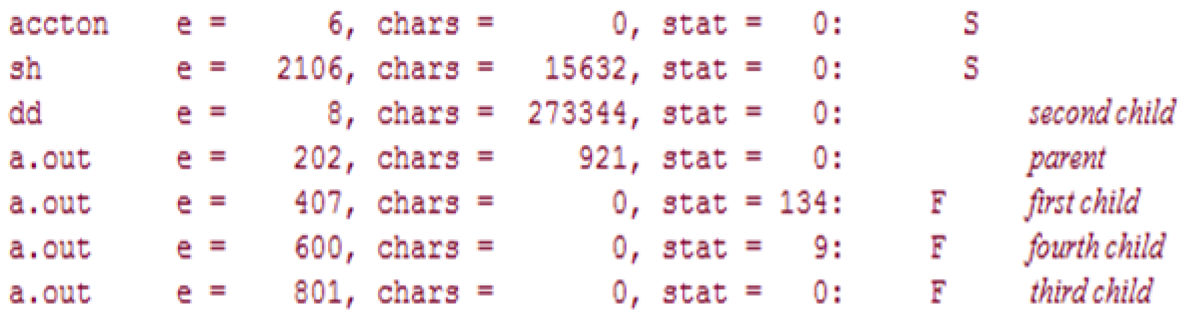
```
int main(void)
{       pid_t pid;
     if ((pid = fork()) < 0)       err_sys("fork error");
     else if (pid != 0) { /* parent */
              sleep(2);  exit(2); /* terminate with exit status 2 */  } /* first child */
     if ((pid = fork()) < 0)  err_sys("fork error"); /* first child */
     else if (pid != 0) {/* first child */
              sleep(4);  abort(); /* terminate with core dump */  }
     if ((pid = fork()) < 0)  err_sys("fork error"); /* second child */
     else if (pid != 0) {/* second child */
              execl("/bin/dd", "dd", "if=/etc/termcap", "of=/dev/null", NULL);
              exit(7); /* shouldn't get here */  }
     if ((pid = fork()) < 0)  err_sys("fork error"); /* third child */
     else if (pid != 0) {/* third child */
              sleep(8);  exit(0); /* normal exit */
     sleep(6); } /* fourth child */
     kill(getpid(), SIGKILL); /* terminate w/signal, no core dump */
     exit(6); /* shouldn't get here */
 }
```

# Process Accounting - Example



```
accton    e =     6, chars =      0, stat =   0:     S
sh        e =  2106, chars =  15632, stat =   0:     S
dd        e =     8, chars = 273344, stat =   0:           second child
a.out     e =   202, chars =    921, stat =   0:           parent
a.out     e =   407, chars =      0, stat = 134:   F       first child
a.out     e =   600, chars =      0, stat =   9:   F       fourth child
a.out     e =   801, chars =      0, stat =   0:   F       third child
```

# Example Output



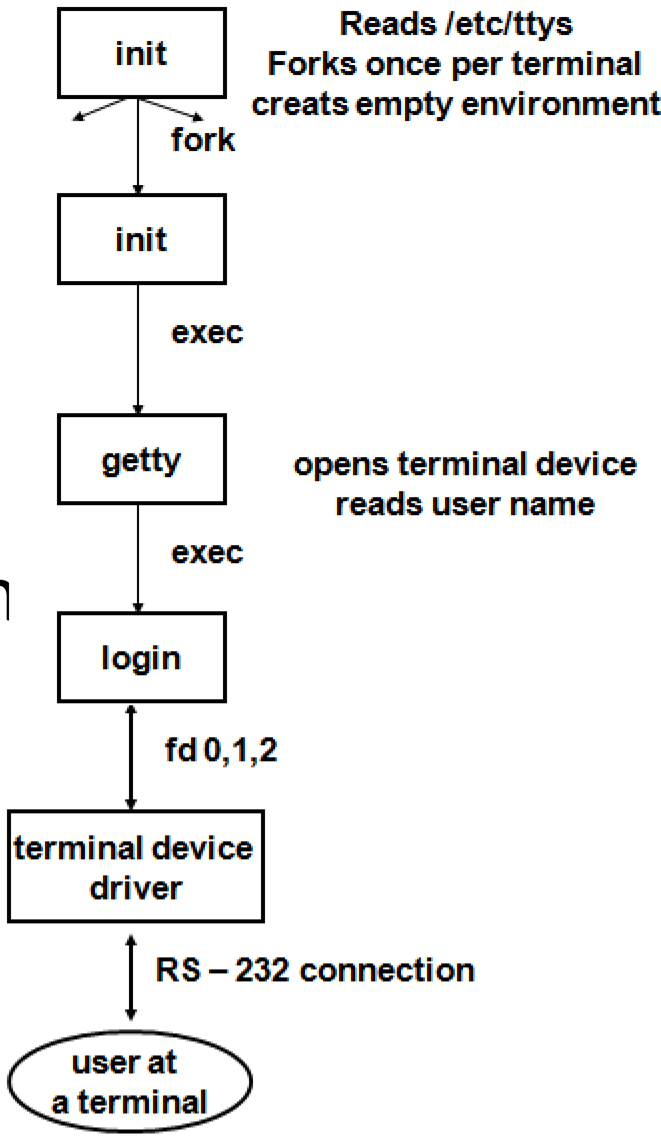
```
accton     e =      6, chars =       0, stat =   0:       S
sh         e =   2106, chars =   15632, stat =   0:       S
dd         e =      8, chars =  273344, stat =   0:            second child
a.out      e =    202, chars =     921, stat =   0:            parent
a.out      e =    407, chars =       0, stat = 134:     F      first child
a.out      e =    600, chars =       0, stat =   9:     F      fourth child
a.out      e =    801, chars =       0, stat =   0:     F      third child
```

- Execution appends six records to the accounting file: one for the superuser shell, one for the test parent, and one for each of the four test children
- A new process is not created by the execl in the second child. There is only a single accounting record for the second child.

# Login Process

- init spawn one getty process per terminal
- Normal Terminal
  - Init → getty → login → shell ↔ terminal driver ↔ user
- When user name is entered, login is execed

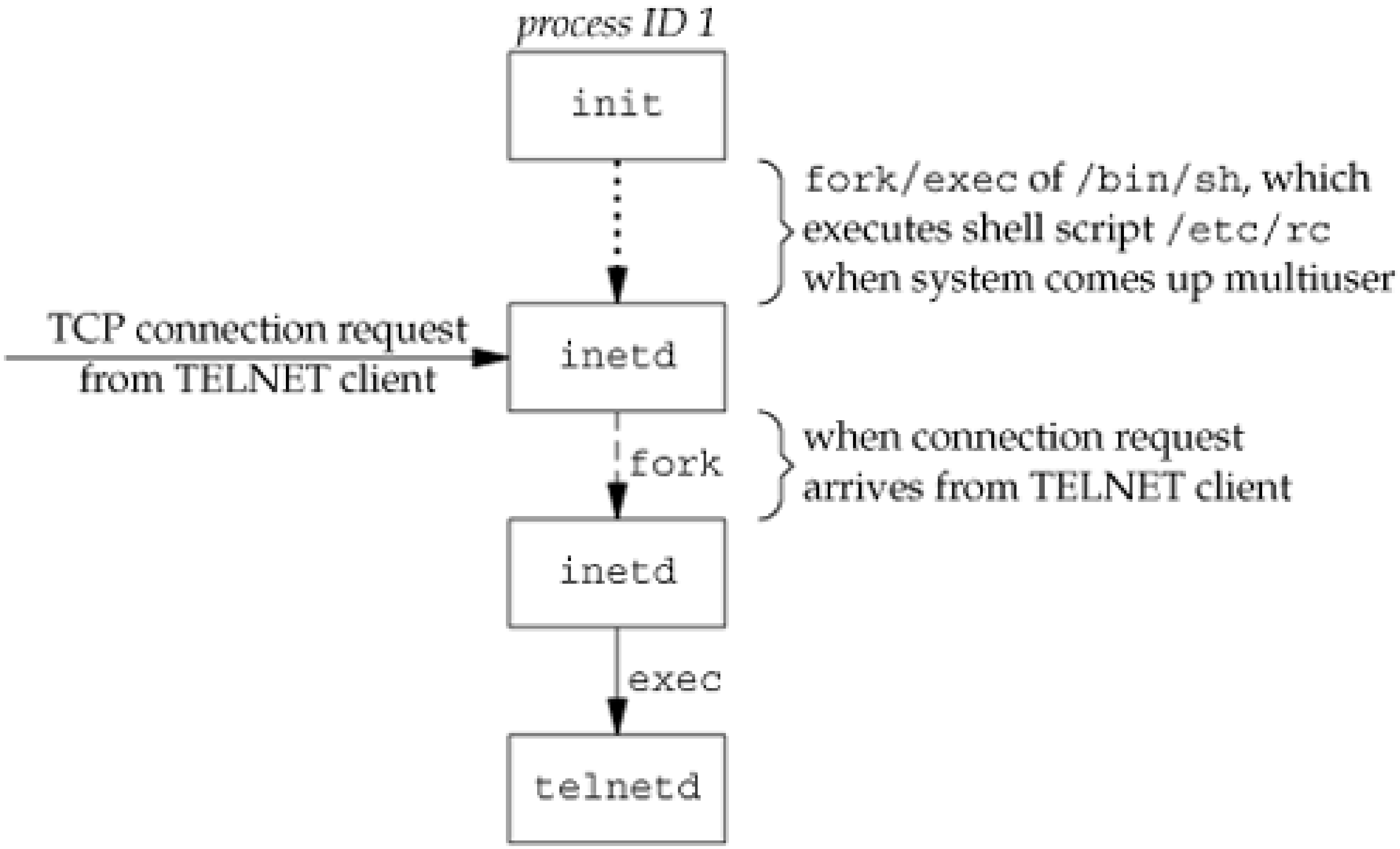
```
execle("/bin/login", "login", "-p", username, (char *)0, envp);
```



## After Success Login

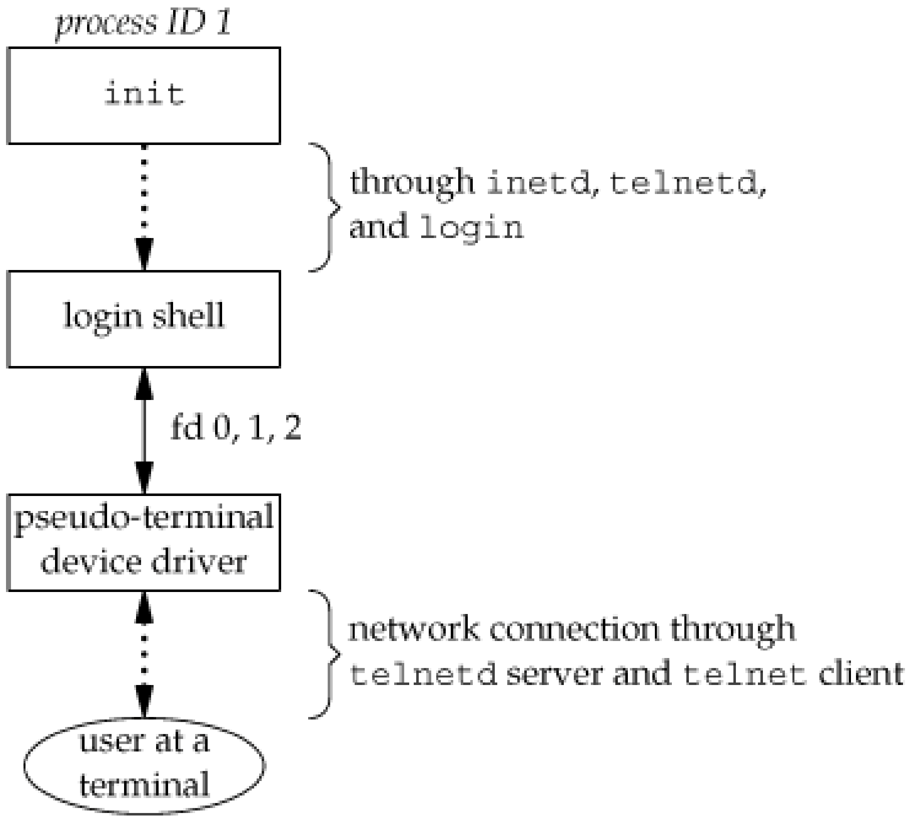
- Change to our home directory (chdir)
- Change the ownership of terminal device (chown) so we own it
- Change the access permissions for terminal device so we have permission to read from and write to it
- Set group IDs by calling setgid and initgroups
- Initialize the environment with all the information that login has: HOME, SHELL, USER and LOGNAME and a default path (PATH)
- Change to user ID (setuid) and invoke our login shell, as in · execl("/bin/sh", "-sh", (char *)0);

# Sequence of processes involved in executing TELNET server


process ID 1
init
fork/exec of /bin/sh, which executes shell script /etc/rc when system comes up multiuser
TCP connection request from TELNET client
inetd
fork
when connection request arrives from TELNET client
inetd
exec
telnetd

# Arrangement of processes after everything is set for a network login

- If we log in correctly, login will
  - Change to home directory (chdir)
  - Change the ownership of terminal device (chown) so we own it
  - Change the access permissions for our terminal device so we have permission to read from and write to it
  - Set group IDs by calling setgid and initgroups
  - Initialize environment : HOME, SHELL, USER and LOGNAME, and a default path (PATH)
  - Change to user ID (setuid) and invoke our login shell, as in execl("/bin/sh", "-sh", (char *)0);


process ID 1
init
through inetd, telnetd, and login
login shell
fd 0, 1, 2
pseudo-terminal device driver
network connection through telnetd server and telnet client
user at a terminal

# Jobs

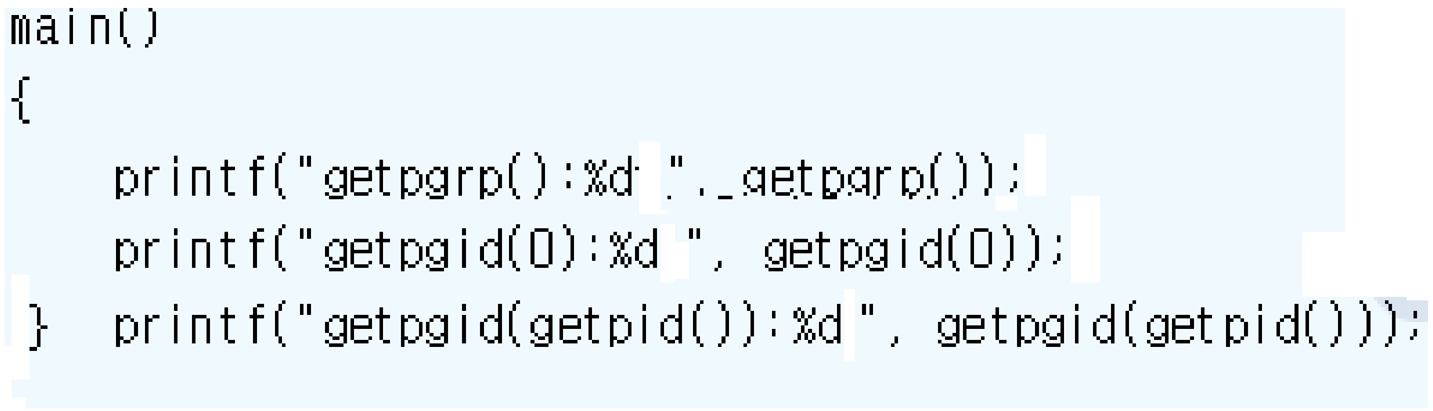
- Multitasking allows run more than one program at a time
- UNIX allows a user to run many processes simultaneously
- A job is a concept used by the shell - any program you interactively start that doesn't detach (ie, not a daemon) is a job
- a job can be in one of three states. It is either in the foreground, in the background, or stopped
- press CtrlZ to suspend a job, can start it back in the foreground (with fg) or in the background (withbg)

# Process Group

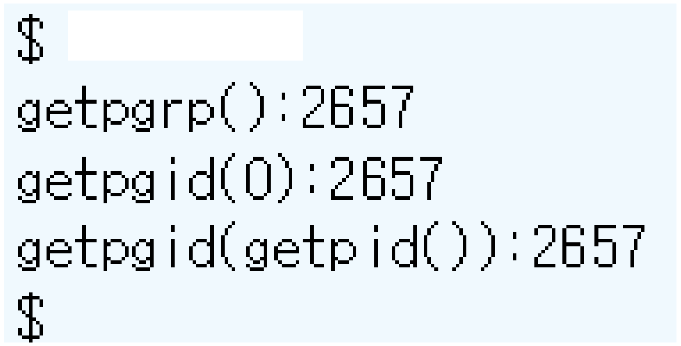
- Is a collection of one or more processes, usually associated with same job that can receive signals from same terminal
- getpgrp returns the process group ID of the calling process:

  pid_t getpgrp(void);
- pid_t getpgid(pid_t pid); takes a pid argument and returns the process group for that process
- Each process group can have a process group leader
  - Leader is identified by its process group ID being  equal to its process ID
- It is possible for a process group leader to create a process group, create processes in the group and then terminate
- Process group still exists, as long as at least one process is in the group, regardless of whether the group leader terminates
  - This is called the process group lifetime—the period of time that begins when the group is created and ends when the last remaining process leaves the group

# Example



```
main()
{
    printf("getpgrp():%d ", getpgrp());
    printf("getpgid(0):%d ", getpgid(0));
}   printf("getpgid(getpid()):%d ", getpgid(getpid()));
```



```
$
getpgrp():2657
getpgid(0):2657
getpgid(getpid()):2657
$
```

## setpgid

- int setpgid(pid_t pid, pid_t pgid);
  - Sets the process group ID to pgid in the process whose process ID equals pid
  - If the two arguments are equal, the process specified by pid becomes a process group leader.
  - If pid is 0, the process ID of the caller is used
  - If pgid is 0, the process ID specified by pid is used as the process group ID
  - A process can set the process group ID of only itself or any of its children
  - It can't change the process group ID of one of its children after that child has called one of the exec functions

## Example

```
int child = fork();
if (child == 0)
{
    setpgid(0, 0); // sure that child is in new pgrp
    printf(“chpgrp=%d pid=%d", getpgrp(), getpid());
}
else
{
    printf(“chpgrp=%d pid=%d", getpgrp(), getpid());
    setpgid(child, child); // child is in new pgrp
}
```

## Process group and waitpid

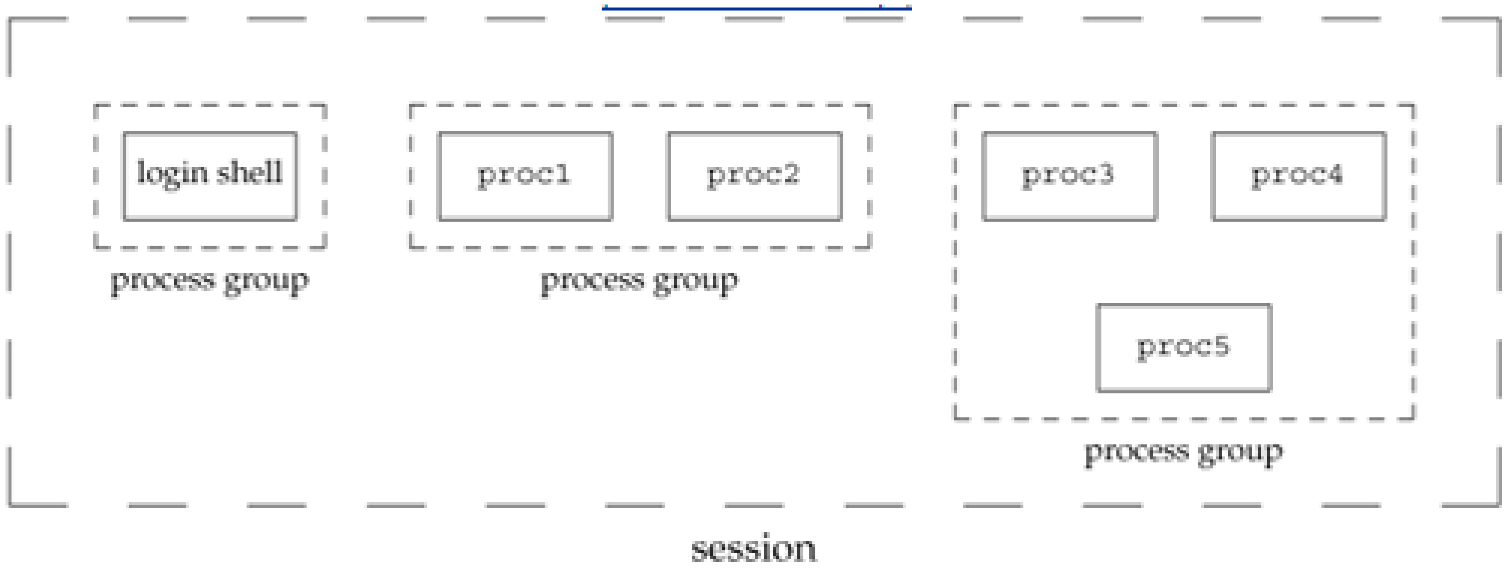
pid_t waitpid(pid_t *pid*, int **statloc*, int *options*);

- *pid* argument for *waitpid* depends on its value:
  - *pid* = = 0 waits for any child whose process group ID equals that of the calling process.
  - *pid* < -1 waits for any child whose process group ID equals the absolute value of *pid*.

## Sessions

- A session is a collection of one or more process groups.
- Arrangement of processes into process groups and sessions generated by executing shell commands

```
proc1 | proc2 &
proc3 | proc4 | proc5
```



## Login shell

- Becomes a control terminal. (A session containing the control terminal)
- Session and group leader
- All processes started from the shell prompt belong to the session

# setsid function

- A process establishes a new session by calling the setsid function.

  pid_t setsid(void); Returns: process group ID if OK, –1 on error
- If the calling process is not a process group leader, this function creates a new session. Three things happen.

1. The process becomes the session leader of this new session. (A session leader is the process that creates a session.) The process is the only process in this new session.
2. The process becomes the process group leader of a new process group. The new process group ID is the process ID of the calling process.
3. The process has no controlling terminal. If the process had a controlling terminal before calling setsid, that association is broken.

## setsid failure

- setsid function returns an error if the caller is already a process group leader
- To ensure this does not happen, the usual practice is to call fork and have the parent terminate and the child continue
- Guaranteed that the child is not a process group leader, because the process group ID of the parent is inherited by the child, but the child gets a new process ID

## getsid function

- pid_t getsid(pid_t pid);

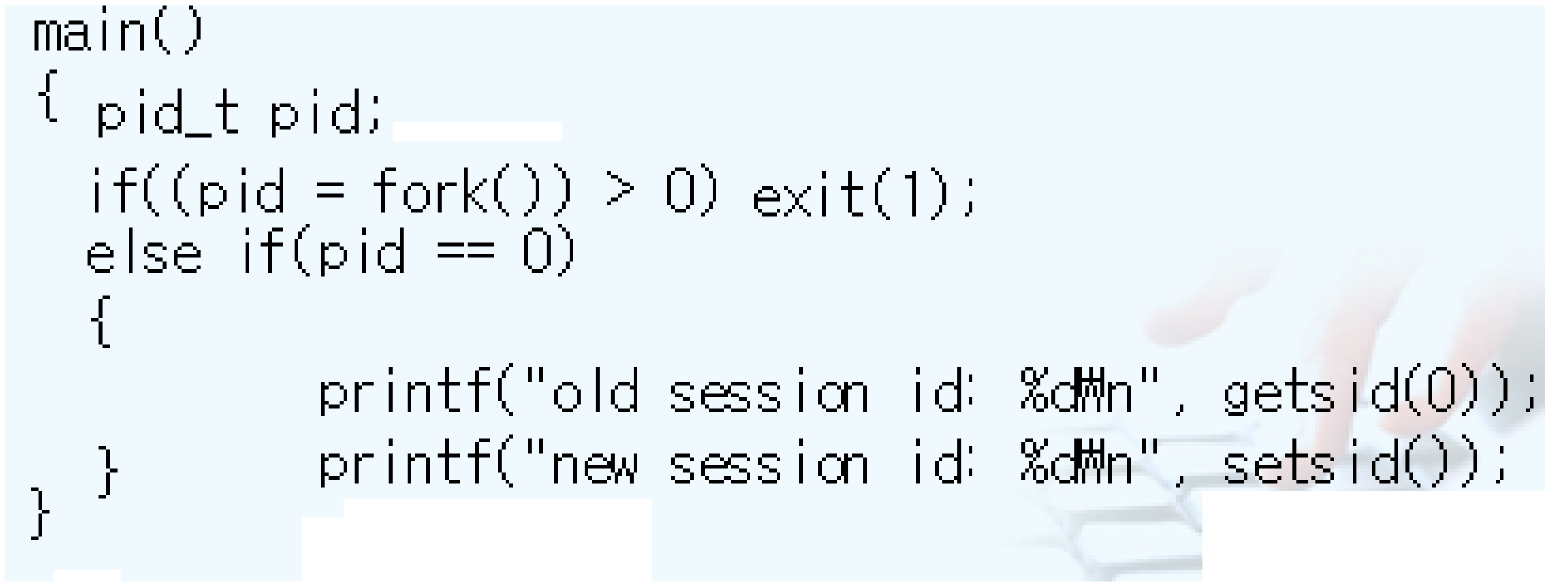
  Returns: session leader's process group ID if OK, –1 on error
- If pid is 0, getsid returns the process group ID of the calling process's session leader

# Example

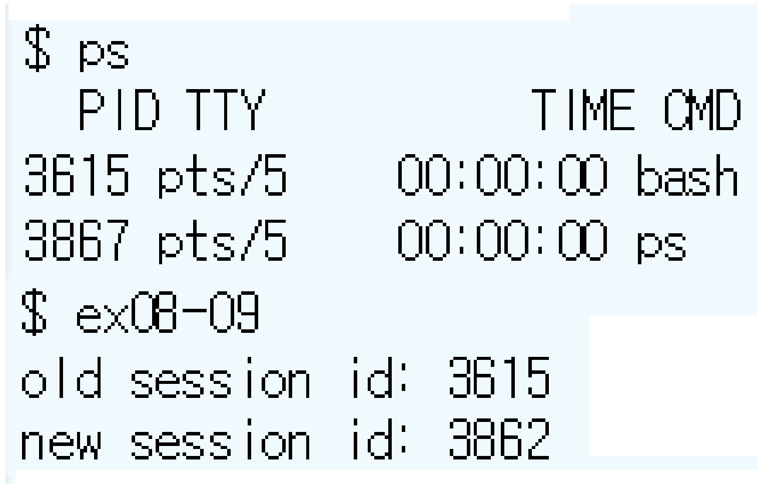
```
main(int argc, char *argv[])
{
    pid_t pid;
    pid = atoi(argv[1]);
    printf("shell process...₩n");
    printf("process id:%d, group id:%d, session id:%d₩n",
            pid, getpgid(pid), getsid(pid));
    printf("current process.. not daemon...₩n");
    printf("process id:%d, group id:%d, session id:%d₩n",
        getpid(), getpgrp(), getsid(0));
}
```

```
$ ps
  PID TTY          TIME CMD
2849 pts/4    00:00:00 bash
3030 pts/4    00:00:00 ps
$ ex08-08 2849 0
shell process...
process id:2849, group id:2849, session id:2849
current process.. not daemon...
process id:3031, group id:3031, session id:2849
$
```

# Example



```
main()
{ pid_t pid;
  if((pid = fork()) > 0) exit(1);
  else if(pid == 0)
  {
          printf("old session id: %d₩n", getsid(0));
  }       printf("new session id: %d₩n", setsid());
}
```

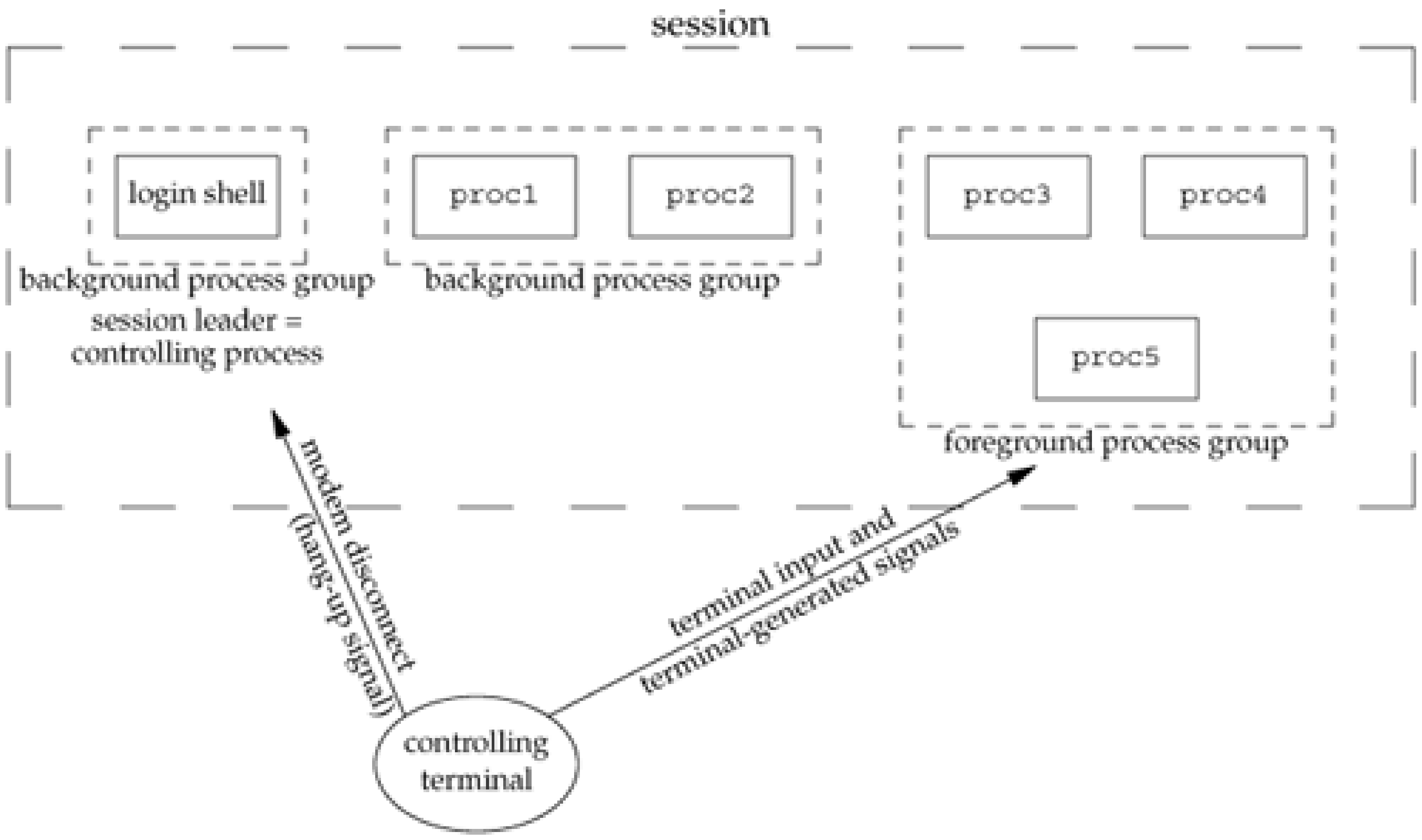


```
$ ps
  PID TTY          TIME CMD
3615 pts/5     00:00:00 bash
3867 pts/5     00:00:00 ps
$ ex08-09
old session id: 3615
new session id: 3862
```

# Controlling Terminal

- A session can have a single controlling terminal. This is usually the terminal device or pseudo-terminal device on which we log in
- It is established automatically when we login
- Session leader that establishes the connection to the controlling terminal is called the controlling process
- Child processes created with fork inherit the controlling terminal from their parent process.
- So all the processes in a session inherit the controlling terminal from the session leader
- Process groups within a session can be divided into a single foreground process group and one or more background process groups
- If a session has a controlling terminal, it has a single foreground process group and all other process groups in the session are background process groups

# Process Groups and Sessions Showing Controlling Terminal



## Controlling Terminal Characteristics

- Whenever we type the terminal's interrupt key (often DELETE or Control-C), this causes the interrupt signal be sent to all processes in the foreground process group
- Whenever we type the terminal's quit key (often Control-backslash), this causes the quit signal to be sent to all processes in the foreground process group
- If a modem (or network) disconnect is detected by the terminal interface, the hang-up signal is sent to the controlling process (the session leader)

## Talk to Controlling Terminal

- There are times when a program wants to talk to the controlling terminal, regardless of whether the standard input or standard output is redirected
- The way a program guarantees that it is talking to the controlling terminal is to open the file /dev/tty.
- This special file is a synonym within the kernel for the controlling terminal
- If the program doesn't have a controlling terminal, the open of this device will fail

## Foreground Process Group Functions

- pid_t tcgetpgrp(int fd); returns the process group ID of the foreground process group associated with the terminal open on fd. –1 on error
- int tcsetpgrp(int fd, pid_t pgrpid); Returns: 0 if OK, –1 on error: If the process has a controlling terminal, the process can call tcsetpgrp to set the foreground process group ID to pgrpid
  - The value of pgrpid must be the process group ID of a process group in the same session, and fd must refer to the controlling terminal of the session
- pid_t tcgetsid(int fd); Returns: session leader's process group ID if OK, –1 on error

## Job Control

- Job control  allows us to start multiple jobs (groups of processes) from a single terminal and to control which jobs can access the terminal and which jobs are to run in the background
- requires three forms of support:
  1. A shell that supports job control
  2. Terminal driver in the kernel must support job control
  3. Kernel must support certain job-control signals
- C shell supported Job control, Bourne shell did not, Korn shell optional

# Job Control Examples

- vi main.c
  starts a job consisting of one process in the foreground.
- The commands

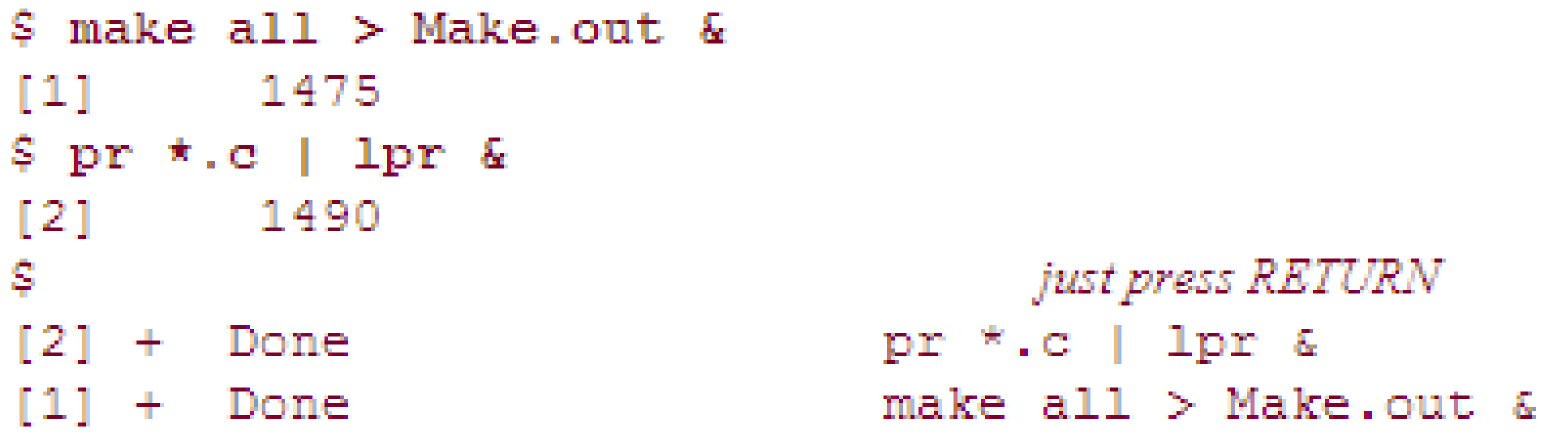
  pr *.c | lpr &

  make all &

  start two jobs in the background. All the processes invoked by these background jobs are in the background

# Job Control Examples

- When we start a background job, the shell assigns it a job identifier and prints one or more of the process IDs.
- When the jobs are done and when we press RETURN, the shell tells us that the jobs are complete.



```
$ make all > Make.out &
[1]     1475
$ pr *.c | lpr &
[2]     1490
$                                    just press RETURN
[2] +  Done                     pr *.c | lpr &
[1] +  Done                     make all > Make.out &
```

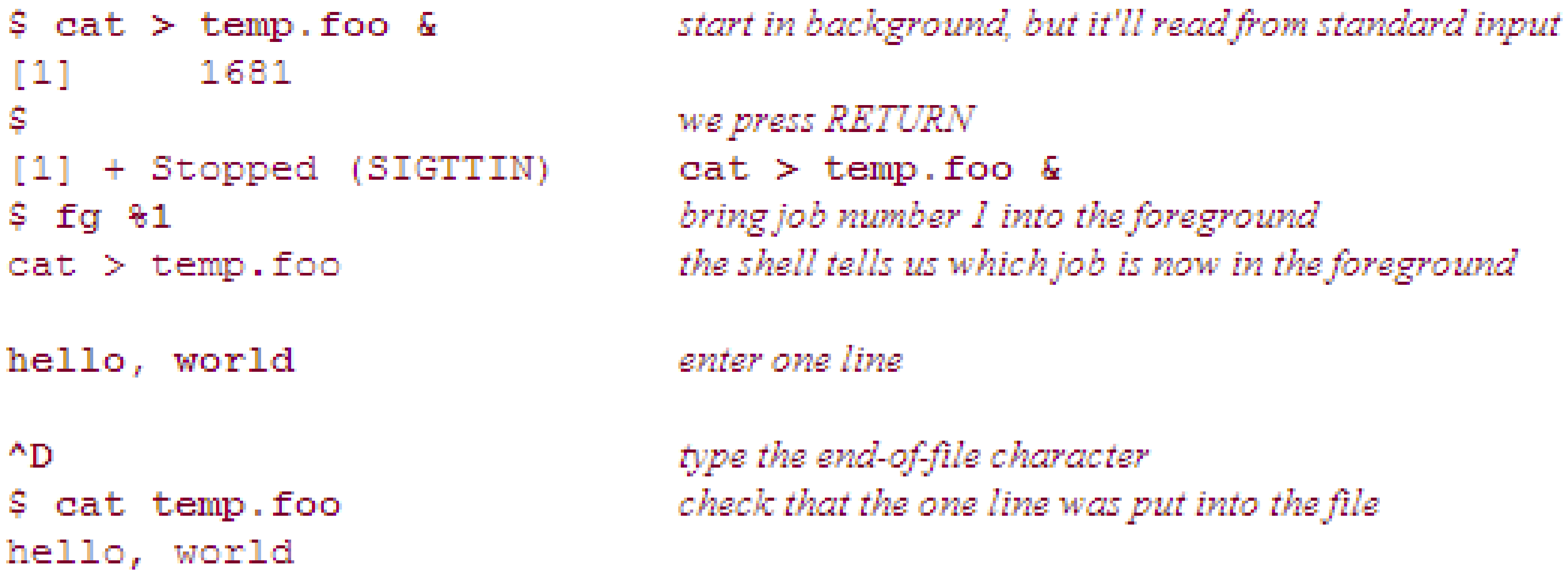
## Signals and Job Control

- Interaction with the terminal driver arises because a special terminal character affects the foreground job:
  - suspend key (typically Control-Z). Entering this character causes the terminal driver to send the SIGTSTP signal to all processes in the foreground process group
  - The jobs in any background process groups aren't affected
- The terminal driver looks for three special characters, which generate signals to the foreground process group
  - Interrupt character (DELETE or Control-C) generates SIGINT
  - Quit character (Control-backslash) generates SIGQUIT
  - Suspend character (Control-Z) generates SIGTSTP

## Terminal Input and Job Control

- Since we can have a foreground job and one or more background jobs, which of these receives the characters that we enter at the terminal?
- Only the foreground job receives terminal input.
- It is not an error for a background job to try to read from the terminal, but the terminal driver detects this and sends a special signal to the background job: SIGTTIN.
- This signal normally stops the background job; by using the shell, we are notified of this and can bring the job into the foreground so that it can read from the terminal

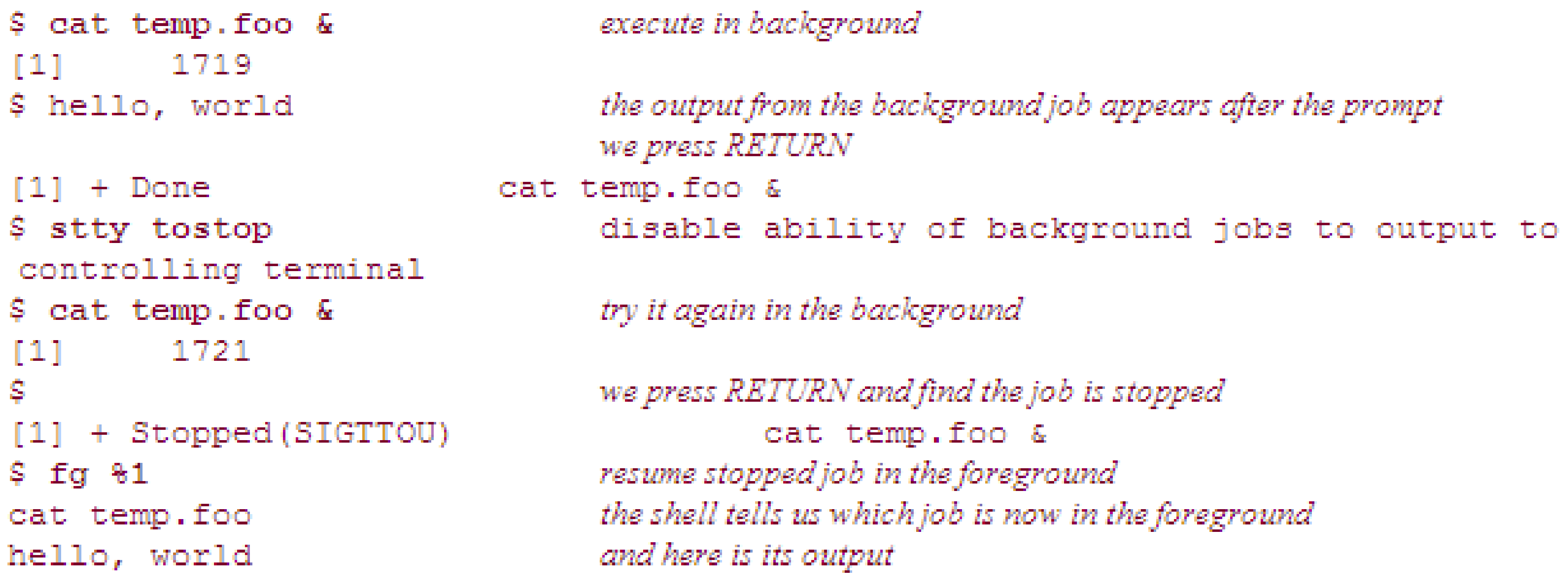
# Background Job reading Terminal



```
$ cat > temp.foo &              start in background, but it'll read from standard input
[1]     1681
$                               we press RETURN
[1] + Stopped (SIGTTIN)         cat > temp.foo &
$ fg %1                         bring job number 1 into the foreground
cat > temp.foo                  the shell tells us which job is now in the foreground

hello, world                    enter one line

^D                              type the end-of-file character
$ cat temp.foo                  check that the one line was put into the file
hello, world
```

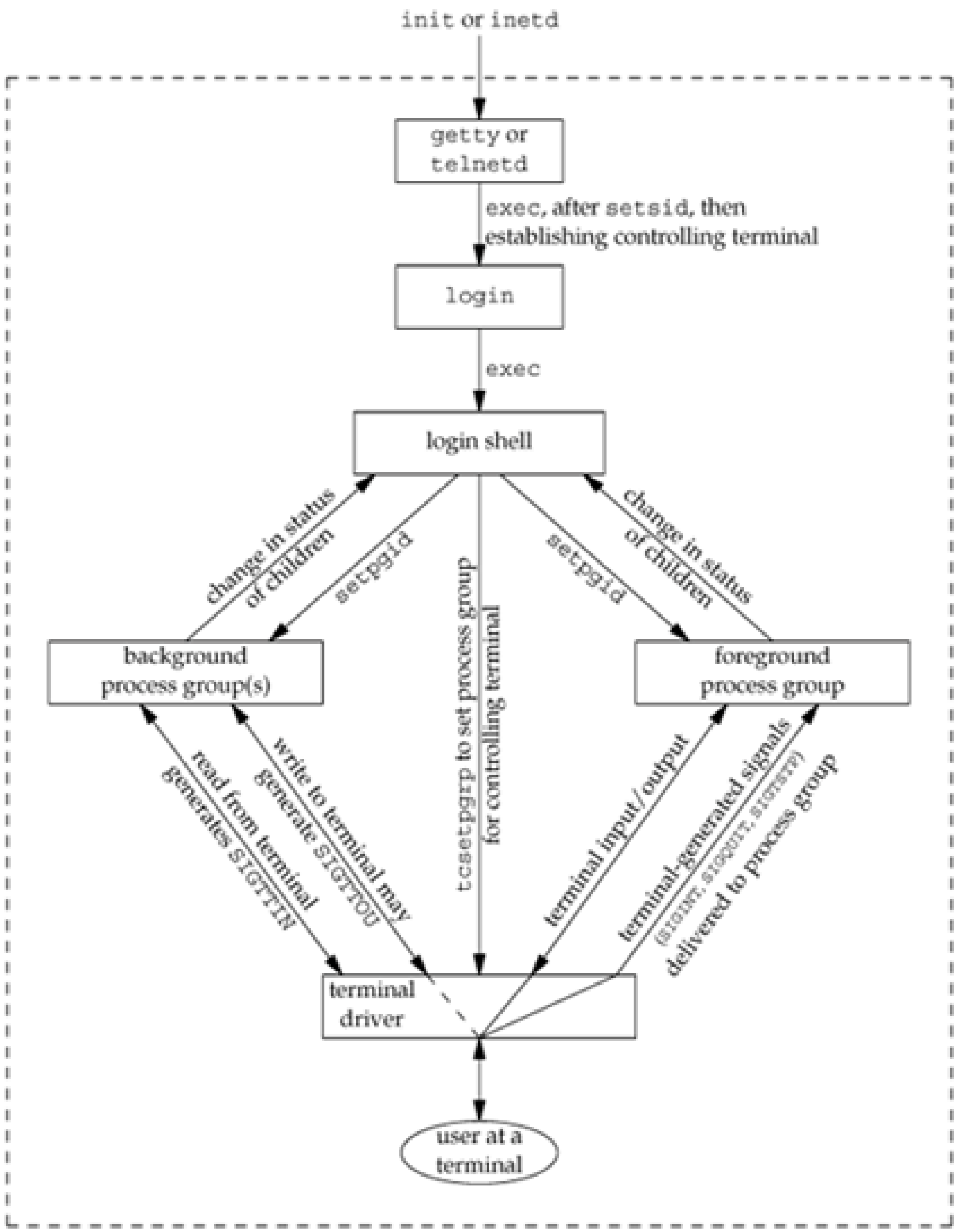
# Job Control Examples

- What happens if a background job outputs to the controlling terminal?
- This is an option that we can allow or disallow.
- Use the stty command to change this option.



```
$ cat temp.foo &                    execute in background
[1]     1719
$ hello, world                      the output from the background job appears after the prompt
                                    we press RETURN
[1] + Done                 cat temp.foo &
$ stty tostop                       disable ability of background jobs to output to
 controlling terminal
$ cat temp.foo &                    try it again in the background
[1]     1721
$                                   we press RETURN and find the job is stopped
[1] + Stopped(SIGTTOU)                      cat temp.foo &
$ fg %1                             resume stopped job in the foreground
cat temp.foo                        the shell tells us which job is now in the foreground
hello, world                        and here is its output
```

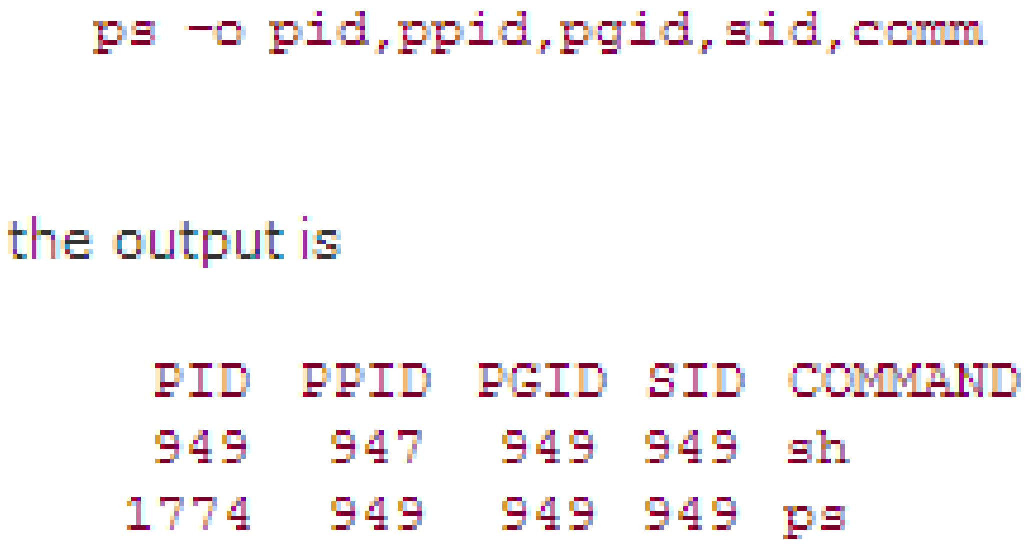
# Summary of Job Control



init or inetd
getty or telnetd
exec, after setsid, then establishing controlling terminal
login
exec
login shell
change in status of children
setpgid
tcsetpgrp to set process group for controlling terminal
setpgid
change in status of children
background process group(s)
foreground process group
read from terminal generates SIGTTIN
write to terminal may generate SIGTTOU
terminal input/output
terminal-generated signals (SIGINT, SIGQUIT, SIGTSTP) delivered to process group
terminal driver
user at a terminal

## Shell Execution of Programs

- With Bourne shell – does not support Job control



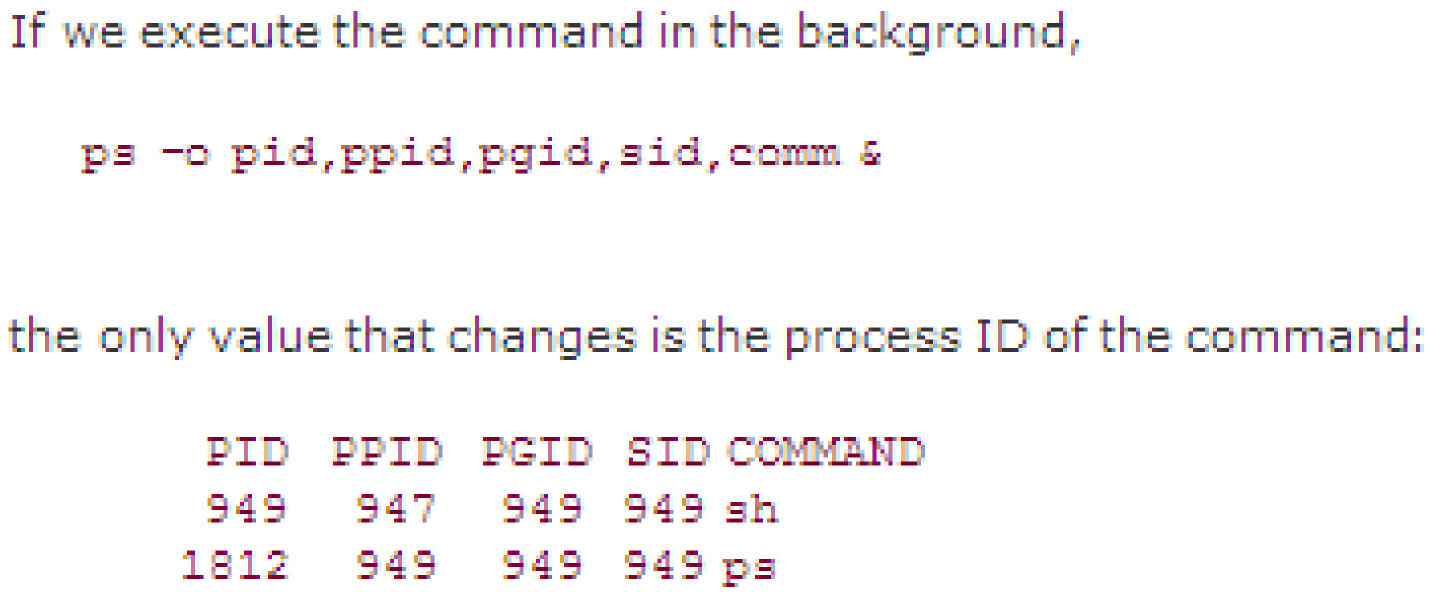
```
ps -o pid,ppid,pgid,sid,comm
```

the output is

```
 PID  PPID  PGID  SID  COMMAND
 949   947   949  949  sh
1774   949   949  949  ps
```

- Shell is parent of ps, both belong to same session and foreground process group

# Bourne shell



If we execute the command in the background,

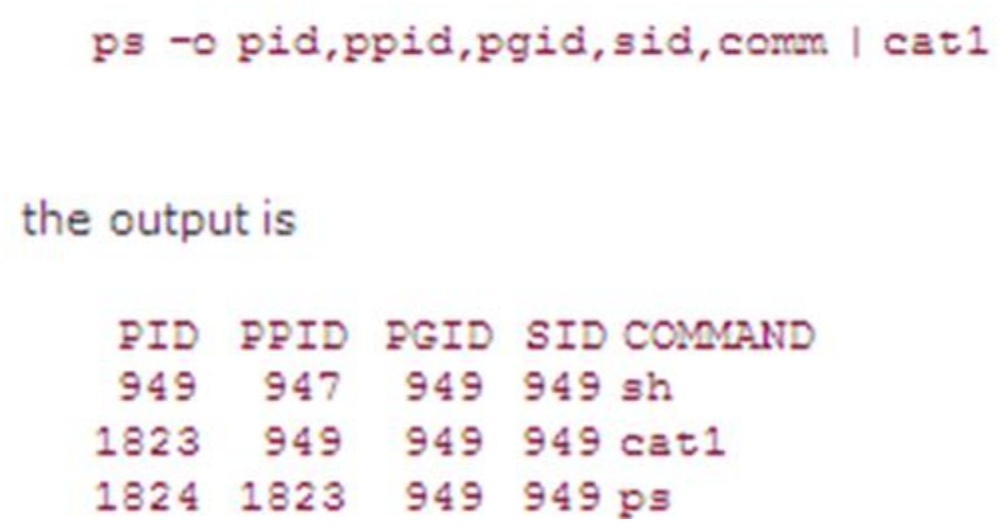
```
ps -o pid,ppid,pgid,sid,comm &
```

the only value that changes is the process ID of the command:

```
  PID  PPID  PGID  SID COMMAND
  949   947   949  949 sh
 1812   949   949  949 ps
```

- No job control, so the background job is not put into its own process group,
- Controlling terminal isn't taken away from the background job.

# Bourne Shell with Pipeline



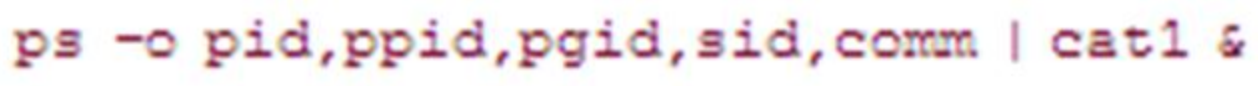
```
ps -o pid,ppid,pgid,sid,comm | cat1
```

the output is

```
 PID  PPID  PGID  SID COMMAND
 949   947   949  949 sh
1823   949   949  949 cat1
1824  1823   949  949 ps
```

- last process in the pipeline (cat1) is the child of the shell and that the first process in the pipeline (ps) is a child of the last process.

## Pipeline at Background



```
ps -o pid,ppid,pgid,sid,comm | cat1 &
```

- only the process IDs change, since shell does not handle job control.

# Bourne Shell

- If a background process tries to read from its controlling terminal: cat > temp.foo &
- Shell automatically redirects the standard input of a background process to /dev/null, if the process doesn't redirect standard input itself
- A read from /dev/null generates an end of file
- Means that background cat process immediately reads an end of file and terminates normally

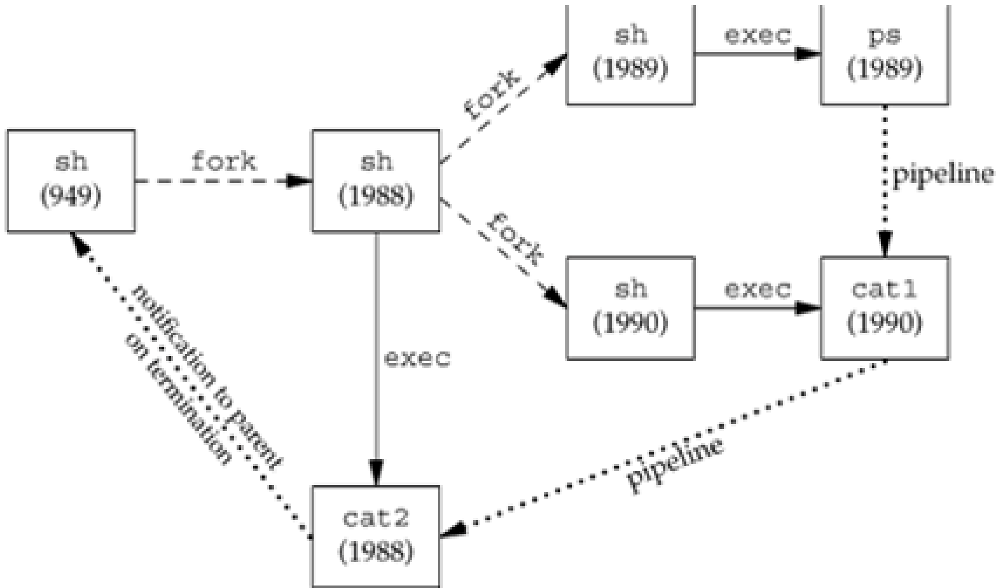
# Multiple Pipelines

```
ps -o pid,ppid,pgid,sid,comm | cat1 | cat2
```

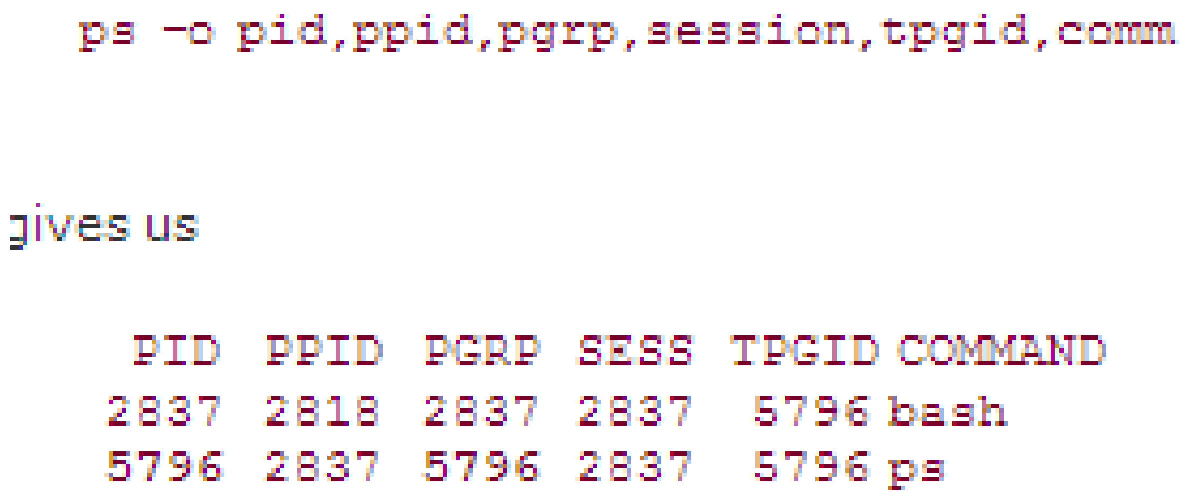
generates the following output

```
 PID  PPID  PGID  SID COMMAND
 949   947   949  949 sh
1988   949   949  949 cat2
1989  1988   949  949 ps
1990  1988   949  949 cat1
```

- Last process in the pipeline is the child of the shell and all previous processes in the pipeline are children of the last process



# Job-control Shell Running on Linux.



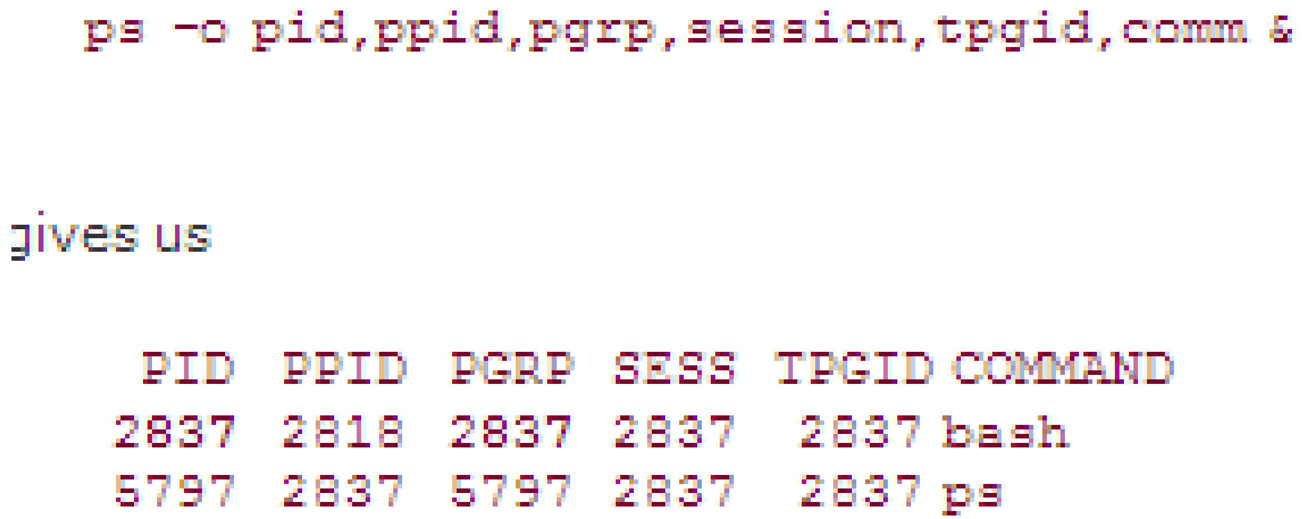
```
ps -o pid,ppid,pgrp,session,tpgid,comm
```

gives us

```
 PID  PPID  PGRP  SESS  TPGID COMMAND
2837  2818  2837  2837   5796 bash
5796  2837  5796  2837   5796 ps
```

- shell places the foreground job (ps) into its own process group (5796).
- ps command is the process group leader and the only process in this process group
- This process group is the foreground process group, since it has the controlling terminal.
- Login shell is a background process group while the ps command executes.

# Background Execution



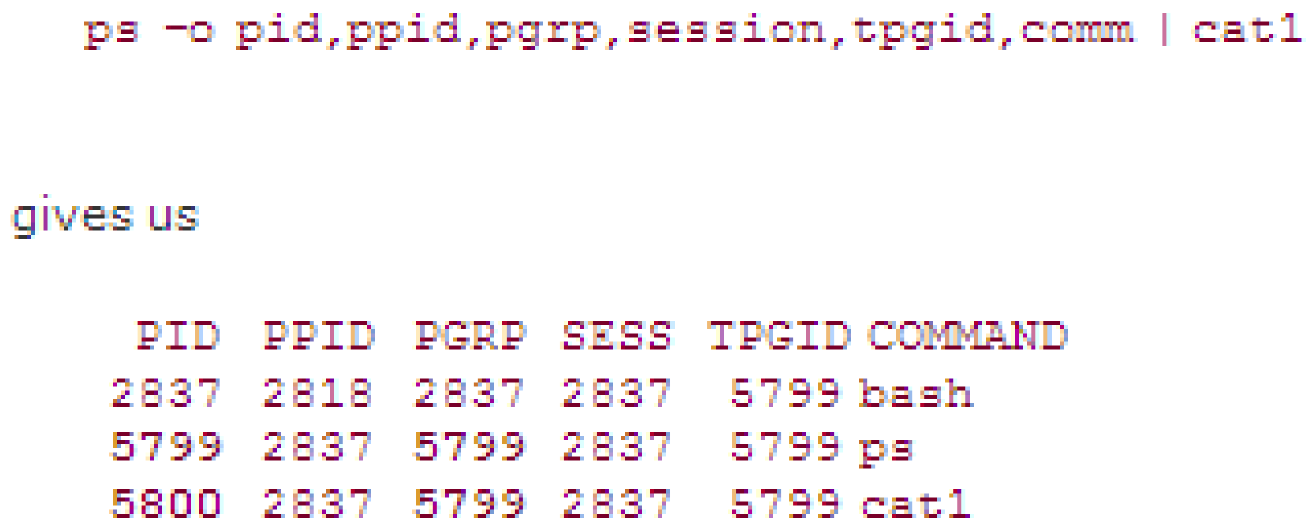
```
ps -o pid,ppid,pgrp,session,tpgid,comm &
```

gives us

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   2837 bash
 5797  2837  5797  2837   2837 ps
```

- ps command is placed into its own process group, but this time the process group (5797) is no longer the foreground process group. It is a background process group.
- The TPGID of 2837 indicates that the foreground process group is our login shell

## Single Pipeline



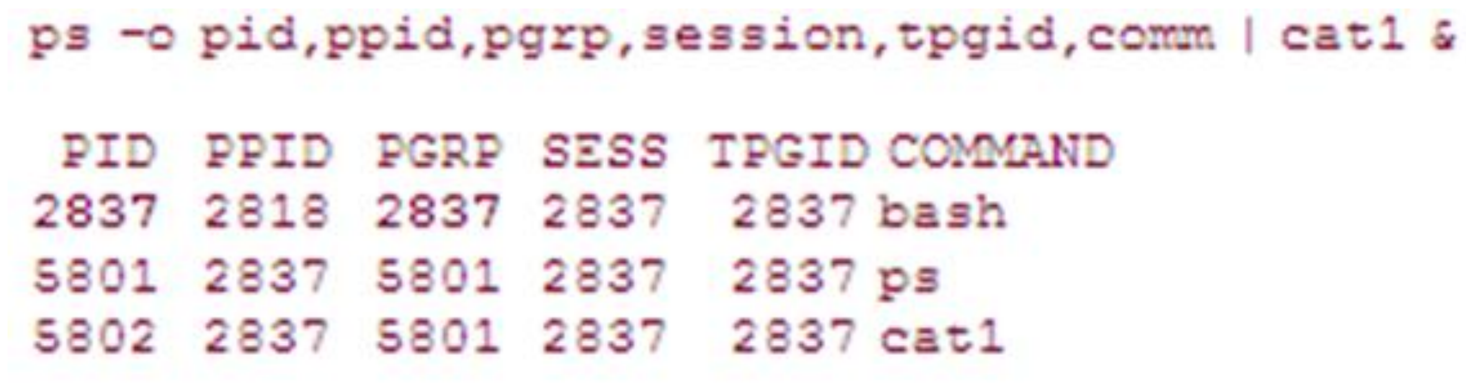
```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1
```

gives us

```
  PID  PPID  PGRP  SESS  TPGID COMMAND
 2837  2818  2837  2837   5799 bash
 5799  2837  5799  2837   5799 ps
 5800  2837  5799  2837   5799 cat1
```

- ps and cat1, are placed into a new process group (5799) and this is the foreground process group
- Bourne shell created the last process in the pipeline first and this final process was the parent of the first process.
- Here, the BASH shell is the parent of both processes

# Pipeline Background



```
ps -o pid,ppid,pgrp,session,tpgid,comm | cat1 &

 PID  PPID  PGRP  SESS  TPGID COMMAND
2837  2818  2837  2837   2837 bash
5801  2837  5801  2837   2837 ps
5802  2837  5801  2837   2837 cat1
```

- Results are similar, but now ps and cat1 are placed in the same background process group

# Orphaned Process Groups

- A process whose parent terminates is called an orphan and is inherited by the init process.
- Normally, the parent of the process group leader is a job control shell (In the same session, but in a different process group)
- If the shell is alive, it can handle the stopping and starting of members in the process group
- When shell dies, there is nobody to continue stopped processes
- SIGHUP  is sent to stopped processes, so they die unless they catch or ignore it and then SIGCONT to continue them

## Orphaned Process Groups

- Normally in a terminal, the session leader (usually shell) controls the controlling terminal
- Session leader sets the controlling terminal to its foreground process group
- If the session leader exits, system revokes further access to controlling terminal and sends a SIGHUP signal to the foreground process group
- By default, this causes the processes to terminate
- Processes can continue running, if a program ignores this signal or establishes a handler for it
- Process groups that continue running even after the session leader has terminated are called orphaned process groups
- As it's controlling process terminated, it cannot access the terminal any more

## Orphaned Process Group (Contd)

- A process can be a member of an orphaned process group even if its original parent process is still alive
- Eg: shell starts a job with single process A. A forks B and the parent shell terminates. Then process B is not an orphan, but member of an orphaned process group

# POSIX.1 and BSD Definition

- Process group is not orphaned as long as a process in the group has a parent in a different process group but in the same session
- Happens when a process forks a child and then dies
  - The child becomes a member of the orphaned group
  - can have problems: child may transform from foreground process to background process automatically
- Terminal signals are not sent to the processes in orphaned groups
- Any stopped process in an orphaned group is automatically sent the SIGHUP and SIGCONT signals when the group is orphaned.

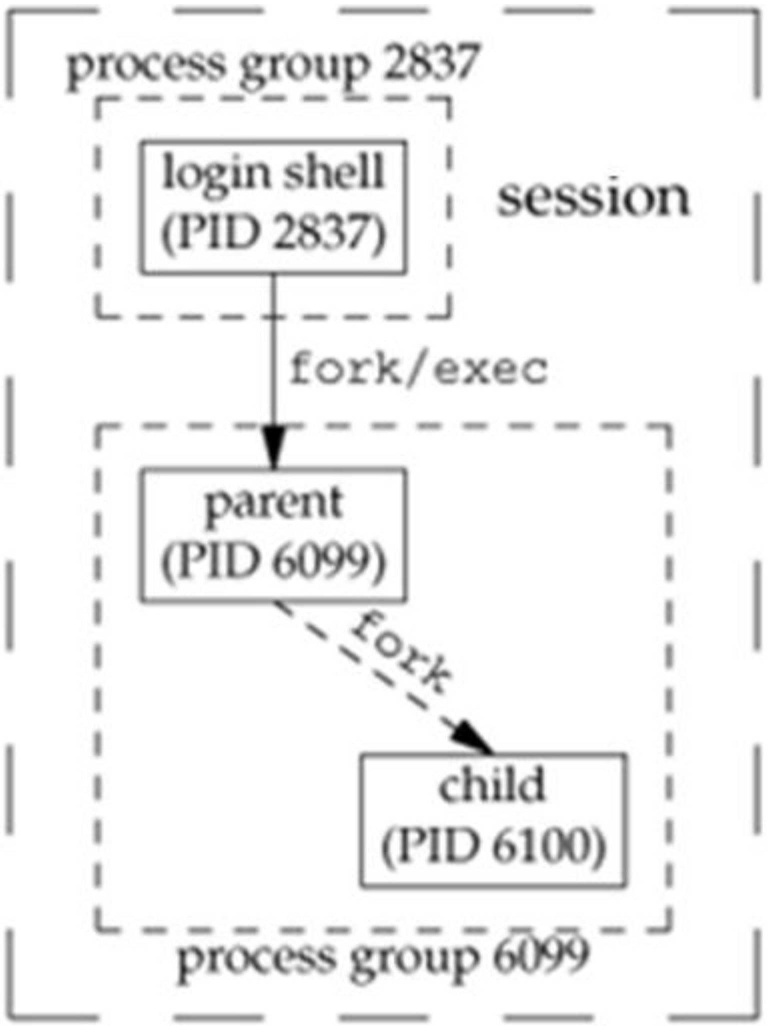
## Creating an Orphaned Process Group

```
static void  sig_hup(int);
static void  pr_ids(char *);
static void    sig_hup (int signo)
 {  printf("SIGHUP received, pid = %d\n", getpid());
    return;
   }
static void  pr_ids (char *name)
{  printf("%s: pid = %d, ppid = %d, pgrp = d\n", name,
   getpid(), getppid(), getpgrp());
    fflush(stdout);
  }
```

# Orphaned Process Group (Cond)

```
int main(void)
{  char   c;
   pid_t  pid;
   pr_ids("parent");
   if ( (pid = fork()) < 0) err_sys("fork error");
   else if (pid > 0) /* parent */
         { sleep(5);        exit(0); }
   else
   {  pr_ids("child");        /* child */
      signal(SIGHUP, sig_hup);       /* establish signal handler */
      kill (getpid(), SIGTSTP);
      pr_ids("child");              /* this prints only if we're continued */
      if (read(0, &c, 1) != -1)
         printf ("read error from control terminal, errno = %d\n", errno);
      exit(0);
    }
}
```

# Orphaned Process Group (Cond)



```
$ ./a.out
parent: pid = 6099, ppid = 2837, pgrp = 6099, tpgrp = 6099
child: pid = 6100, ppid = 6099, pgrp = 6099, tpgrp = 6099
$ SIGHUP received, pid = 6100
child: pid = 6100, ppid = 1, pgrp = 6099, tpgrp = 2837
read error from controlling TTY, errno = 5
```

## Program Details

- Shell (with process group 2837) places the foreground process in its own process group( 6099.
- After the fork, the parent sleeps for 5 seconds to let the child execute before the parent terminates (lightly loaded system)
  - Child establishes a signal handler for the hang-up signal (SIGHUP)
  - Child itself sends stop signal(SIGTSTP) with the kill function.
  - When the parent terminates, the child is orphaned, so the child’s parent process ID becomes 1, the init process ID

# Orphaned Process Group (Cond)

- Child becomes the background process group when the parent terminates, since the parent was executed as a foreground job by the shell
- At this point the child is now a member of an *orphaned process group*
- Since the process group is orphaned when the parent terminates, it is required that every process in the newly orphaned process group that is stopped be sent the hang-up signal (SIGHUP) followed by the continue signal.
- This causes the child to be continued, after processing the hang-up signal

## Daemon process

A daemon is a process that:

- runs in the background
- not associated with any terminal
    - output doesn't end up in another session.
    - terminal generated signals (^C) aren't received.

## Common Daemons

- Web server (httpd)
- Mail server (sendmail)
- SuperServer (inetd)
- System logging (syslogd)
- Print server (lpd)
- router process (routed, gated)

# Coding rules

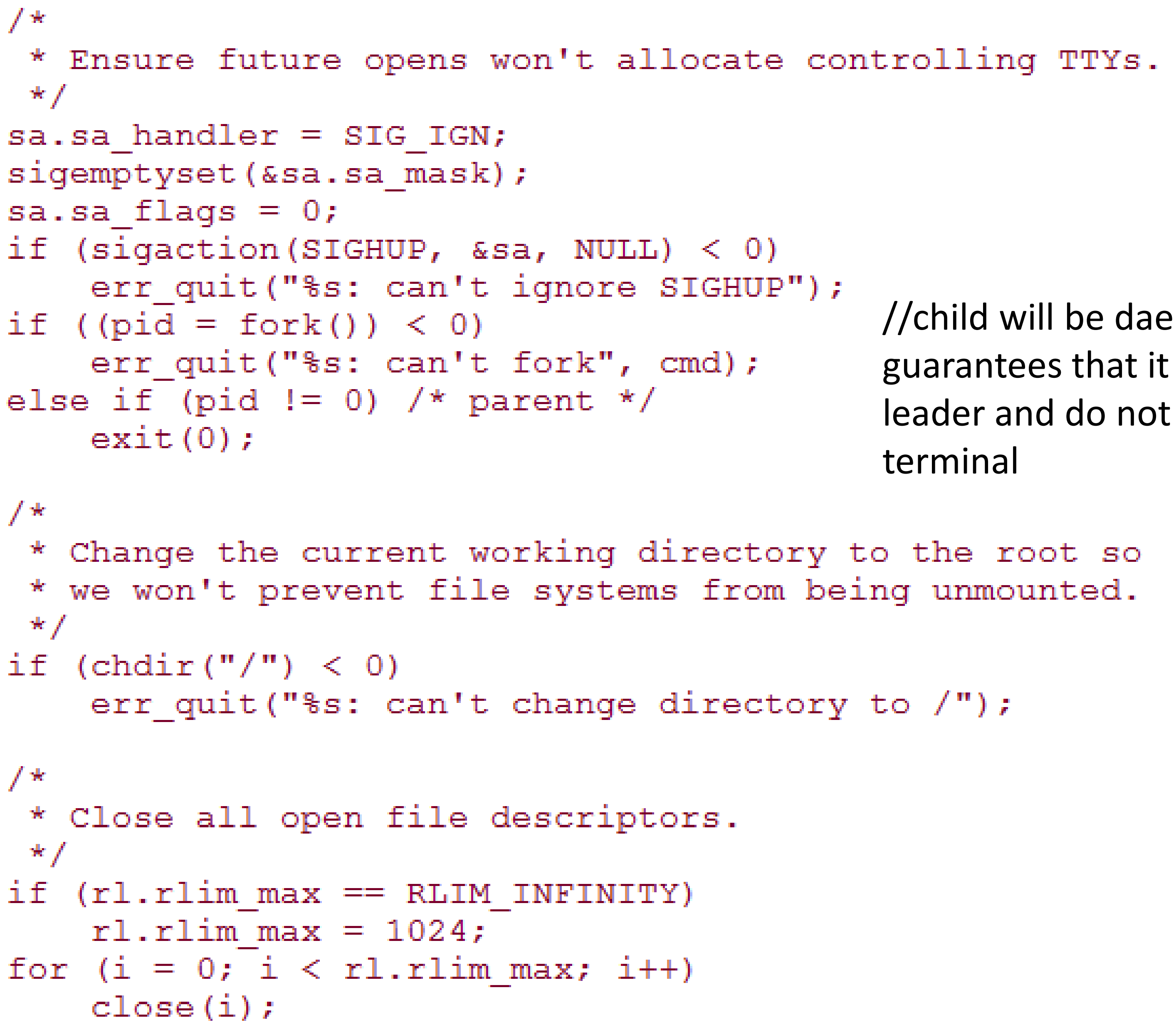
- Call umask to set the file mode creation mask to 0. The file mode creation mask that's inherited could be set to deny certain permissions. If the daemon process is going to create files, it may want to set specific permissions. For example, if it specifically creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts.
- Call fork and have the parent exit. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.
- Call setsid to create a new session. The process (a) becomes a session leader of a new session, (b) becomes the process group leader of a new process group, and (c) has no controlling terminal.
- Under System V based systems, some people recommend calling fork again at this point and having the parent terminate. The second child continues as the daemon. This guarantees that the daemon is not a session leader, which prevents it from acquiring a controlling terminal under the System V rules . Alternatively, to avoid acquiring a controlling terminal, be sure to specify O_NOCTTY whenever opening a terminal device.

# Coding rules (Contd)

- Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.
- Alternatively, some daemons might change the current working directory to some specific location, where they will do all their work. For example, line printer spooling daemons often change to their spool directory.
- Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent. We can use getrlimit function to determine the highest descriptor and close all descriptors up to that value.
- Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed; nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

# Initialize a daemon process

```
void
daemonize(const char *cmd)
{
    int                 i, fd0, fd1, fd2;
    pid_t               pid;
    struct rlimit       rl;
    struct sigaction    sa;
    /*
     * Clear file creation mask.
     */
    umask(0);

    /*
     * Get maximum number of file descriptors.
     */
    if (getrlimit(RLIMIT_NOFILE, &rl) < 0)
        err_quit("%s: can't get file limit", cmd);

    /*
     * Become a session leader to lose controlling TTY.
     */
    if ((pid = fork()) < 0)
        err_quit("%s: can't fork", cmd);
    else if (pid != 0) /* parent */
        exit(0);
    setsid();
```



```
/*
 * Ensure future opens won't allocate controlling TTYs.
 */
sa.sa_handler = SIG_IGN;
sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
if (sigaction(SIGHUP, &sa, NULL) < 0)
    err_quit("%s: can't ignore SIGHUP");
if ((pid = fork()) < 0)
    err_quit("%s: can't fork", cmd);
else if (pid != 0) /* parent */
    exit(0);

/*
 * Change the current working directory to the root so
 * we won't prevent file systems from being unmounted.
 */
if (chdir("/") < 0)
    err_quit("%s: can't change directory to /");

/*
 * Close all open file descriptors.
 */
if (rl.rlim_max == RLIM_INFINITY)
    rl.rlim_max = 1024;
for (i = 0; i < rl.rlim_max; i++)
    close(i);
```

//child will be daemon and guarantees that it is not a session leader and do not have controlling terminal

```
    /*
     * Attach file descriptors 0, 1, and 2 to /dev/null.
     */
    fd0 = open("/dev/null", O_RDWR);
    fd1 = dup(0);
    fd2 = dup(0);

    /*
     * Initialize the log file.
     */
    openlog(cmd, LOG_CONS, LOG_DAEMON);
    if (fd0 != 0 || fd1 != 1 || fd2 != 2) {
        syslog(LOG_ERR, "unexpected file descriptors %d %d %d",
          fd0, fd1, fd2);
        exit(1);
    }
}
```

# Daemon Conventions

- If the daemon uses a lock file, the file is usually stored in /var/run. Note, however, that the daemon might need superuser permissions to create a file here. The name of the file is usually name.pid, where name is the name of the daemon or the service. For example, the name of the cron daemon's lock file is /var/run/crond.pid.
- If the daemon supports configuration options, they are usually stored in /etc. The configuration file is named name.conf, where name is the name of the daemon or the name of the service. For example, the configuration for the syslogd daemon is /etc/syslog.conf.
- Daemons can be started from the command line, but they are usually started from one of the system initialization scripts (/etc/rc* or /etc/init.d/*). If the daemon should be restarted automatically when it exits, we can arrange for init to restart it if we include a respawn enTRy for it in /etc/inittab.
- If a daemon has a configuration file, the daemon reads it when it starts, but usually won't look at it again. If an administrator changes the configuration, the daemon would need to be stopped and restarted to account for the configuration changes. To avoid this, some daemons will catch SIGHUP and reread their configuration files when they receive the signal. Since they aren't associated with terminals and are either session leaders without controlling terminals or members of orphaned process groups, daemons have no reason to expect to receive SIGHUP. Thus, they can safely reuse it.