Process Control

Process Identifiers

- process id
  - every process has a unique process ID
  - often used as a piece of their identifiers, to guarantee uniqueness
  - #include <unistd.h>
  - pid_t getpid(void);
    - returns the process ID of the current process
  - pid_t getppid(void);
    - returns the process ID of the parent of the current process

Process Identifiers

- pid_t getuid(void);
  - returns the real user ID of the current process
- pid_t geteuid(void);
  - returns the effective user ID of the current process
- gid_t getgid(void);
  - returns the real group ID of the current process
- gid_t getegid(void);
  - returns the effective group ID of the current process

Init Processes

- init
  - pid 1
  - invoked by the kernel at the end of the bootstrap procedure
  - /sbin/init
  - is responsible for bringing up the system after the kernel has been bootstrapped
    - reads /etc/rc* brings the system to a certain state
  - init is a normal user process with superuser privileges

fork( )

- synopsis
  - #include <sys/types.h>
  - #include <unistd.h>
  - pid_t fork(void);
- description
  - the only way a new process is created is when an existing process calls the fork( ) system call
  - the new process created by fork is called the child process

fork( )

- fork( ) is called once but returns twice
  - parent : pid of child process
  - child : 0
- both the child and parent continue executing with the instruction that follows the call to fork( )
- the child is a copy of parent
  - child gets a copy of the parent's data space, heap, and stack
  - often the parent and child share the text segment

fork( )

- copy-on-write
  - regions shared by the parent and child have their protection changed by the kernel read-only
  - if either process tries to modify these regions, the kernel then makes a copy of that piece of memory only
  - the only penalty incurred by fork
    - the time and memory required to duplicate the parent's page tables
    - the time to create a unique task structure for the child

fork( )

- the child process inherit the following from the parent
  - real-uid(gid), effective-uid(gid), supplementary gid
  - process group id, session id, controlling terminal
  - set-user(group)-id flag
  - current working directory, root directory, umask
  - signal mask and dispositions
  - close-on-exec flags
  - environment
  - resource limits

fork( )

- differences between the parent and child
  - the return value from fork
  - PID and PPID
  - child's resource utilizations are set to 0
  - pending alarms are cleared for the child
  - the set of pending signals for the child is set to the empty set
- we never know if the child starts executing before the parent or vice versa
  - depends on the scheduling algorithm used by the kernel

fork( )

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
main()
{
    pid_t pid;
    printf("Hello, my pid is %d\n", getpid());
    if ((pid = fork()) == 0) {                  /* child process */
        printf("child: pid = %d, ppid = %d\n", getpid(), getppid());
    }    else {                                 /* parent process */
        printf("parent: I created child with pid=%d\n",pid);
    }
/* Following line is executed by both parent and child */
    printf("Bye, my pid is %d\n", getpid());
}
```

fork( )

- example2

```
#include  <sys/types.h>
#include  <unistd.h>
#include  <stdio.h>
int glob = 6;
char        buf[] = "a write to stdout\n";
int main(void)
{
     int      var;
     pid_t  pid;
     var = 88;
     if  (write(STDOUT_FILENO, buf, sizeof(buf)-1) !=sizeof(buf)-1) {
               perror(“write error”);
               exit(1);
     }
     printf("before fork\n");     /* we don't flush stdout */
     if ( (pid = fork()) < 0) {
               perror(“fork error”);
               exit(1);
     }
     else if (pid == 0) {              /* child */
               glob++;       /* modify variables */
               var++;
     }
     else    sleep(2);                  /* parent */
     printf("pid = %d, glob = %d, var = %d\n",
                         getpid(), glob, var);
     exit(0);
}
```

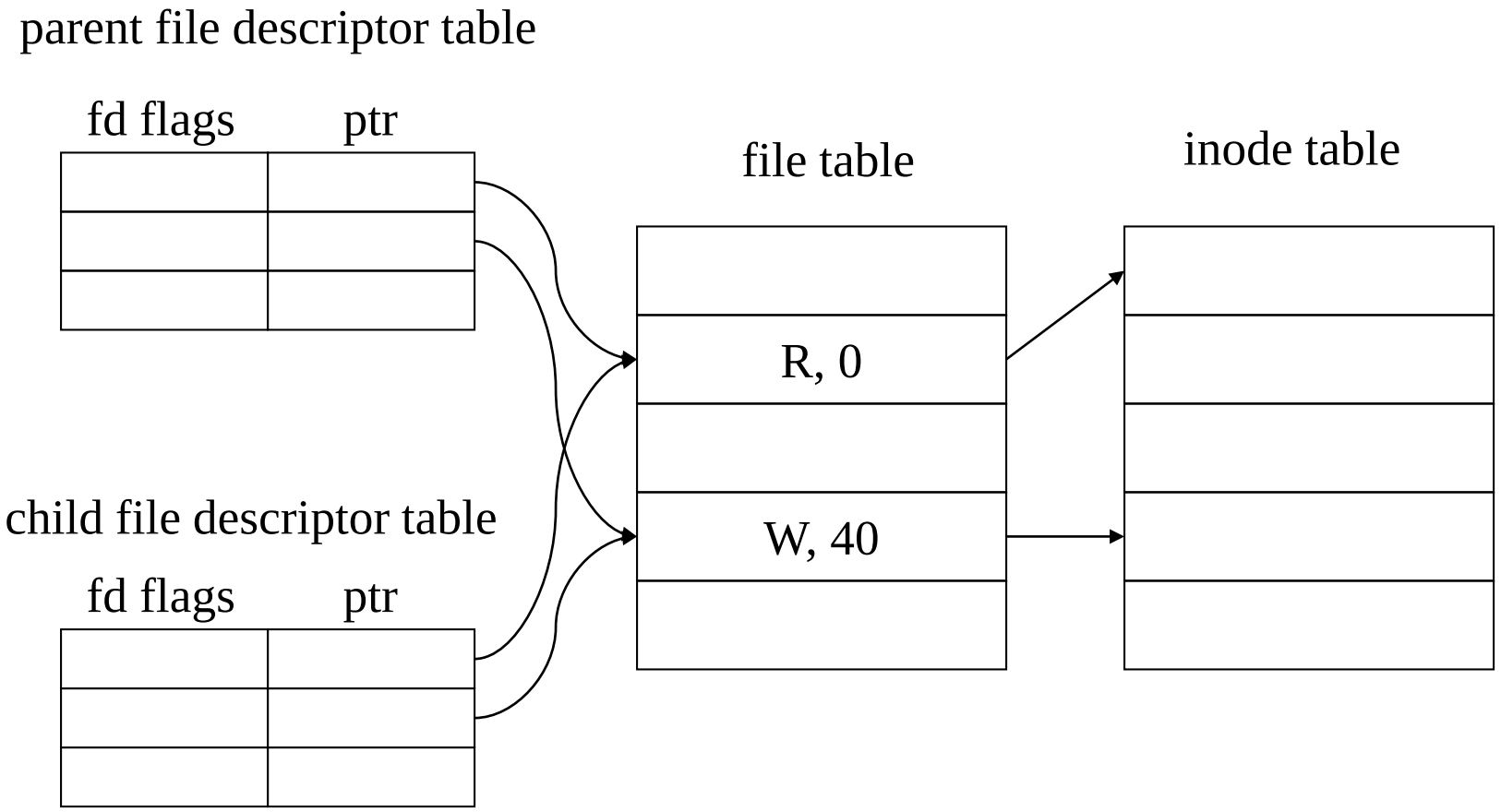
fork( )

- file sharing
  - when we redirect the standard output of the parent, the child's standard output is also redirected
  - all descriptors that are open in the parent are duplicated in the child
- cases for handling the descriptors after a fork
  - the parent waits for the child to complete
    - the parent does not need to do anything with its descriptors
    - when the child terminates, any of the shared descriptors that the child read from or wrote to will have their file offsets updated accordingly

fork( )

- the parent and child each go their own way
  - after the fork, the parent closes the descriptors that it doesn't need and the child does the same thing
  - this way neither interferes with the other's open descriptors
  - this scenario is often the case with network servers

fork( )


parent file descriptor table
fd flags
ptr
file table
inode table
R, 0
W, 40
child file descriptor table
fd flags
ptr

fork( )

- two main reasons for fork to fail
  - if there are already too many processes in the system
  - if the total number of processes for this real user ID exceeds the system's limit
- two uses for fork
  - a process wants to duplicate itself
    - parent and child can each execute different sections of code at the same time
    - common for network server
  - a process wants to execute a different program
    - common for shells
    - the child does an exec( ) right after it returns from the fork( )

vfork( )

- synopsis
  - #include <unistd.h>
  - #include <sys/types.h>
  - pid_t vfork(void);
- description
  - same calling sequence and same return values as fork( )
  - semantic of vfork( ) differs from fork( )
    - intended to create a new process when the purpose of the new process is to exec( ) a new program

vfork( )

- vfork( ) creates the new process just like fork( ), without fully copying the address space of the parent into the child
- the child just calls exec( ) or exit( ) right after the vfork( )
- while the child is running, until it calls either exec( ) or exit( ), the child runs in the address space of the parent
  - provides an efficiency gain
- vfork( ) guarantees that the child runs first, until the child calls exec( ) or exit( )

vfork( )

- example

```
#include<sys/types.h>
#include<unistd.h>
#include<stdio.h>
int         g = 6;
int main(void)
{
    int      var;
    pid_t   pid;
    var = 88;
    printf("before vfork\n");
    /* we don't flush stdio */
    if ( (pid = vfork()) < 0) {
            perror("vfork error");
            exit(1);
    }
    else if (pid == 0) {        /* child */
        g++;                /* modify parent's variables */
        var++;
        _exit(0);         /* child terminates */
    }
    /* parent */
    printf("pid = %d, g = %d, var = %d\n",
            getpid(), g, var);
    exit(0);
}
```

Process Termination

- termination status
  - terminating process notify its parent how it terminated
  - normal termination
    - pass an exit status as argument to exit( ) or _exit( )
    - return value from main( )
    - exit status is converted into a termination status by the kernel when _exit( ) is finally called

Process Termination

- abnormal termination
  - the kernel generates a termination status to indicate the reason for the abnormal termination
- the parent of the terminated process can obtain the termination status from either wait( ) or waitpid( )
- init process becomes the parent process of any process whose parent terminates (inherited by init)

Process Termination

- zombie state
  - the kernel has to keep a certain amount of information consists of PID, the termination status of the process, and the amount of CPU time taken by the process
  - the process that has terminated, but whose parent has not yet waited for it, is called a zombie
  - init is written so that whenever one of its children terminates, init calls one of the wait functions to fetch the termination status
  - init prevents the system from being clogged by zombies

wait( ) and waitpid( )

- synopsis
  - #include <sys/types.h>
  - #include <sys/wait.h>

  - pid_t wait(int *status)
  - pid_t waitpid(pid_t pid, int *status, int options);

- description
  - wait for process termination

wait( ) and waitpid( )

- pid_t wait(int *status)
  - suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function
  - if a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately
  - if the caller blocks and has multiple children, wait returns when one terminates

wait( ) and waitpid( )

- we can always tell which child terminated because the process ID is returned by the function
- Any system resources used by the child are freed
- status
  - if not NULL, wait store status information(termination status) in the location pointed to by status
- macros
  - WIFEXITED(status)
    - is non-zero if the child exited normally.

wait( ) and waitpid( )

- WEXITSTATUS(status)
  - evaluates to the least significant eight bits of the return code of the child which terminated, which may have been set as the argument to a call to exit() or as the argument for a return
  - This macro can only be evaluated if WIFEXITED returned non-zero
- WIFSIGNALED(status)
  - returns true if the child process exited because of a signal which was not caught

wait( ) and waitpid( )

- WTERMSIG(status)
  - returns the number of the signal that caused the child process to terminate
  - This macro can only be evaluated if WIFSIGNALED returned non-zero.

- pid_t waitpid(pid_t pid, int *status, int options);
  - waitpid doesn't wait for the first child to terminate - it has a number of options that control which process it waits for
  - pid < -1
    - wait for any child process whose process group ID is equal to the absolute value of pid

wait( ) and waitpid( )

- pid == -1
  - wait for any child process
  - same behaviors which wait( ) exhibits
- pid == 0
  - wait for any child process whose process group ID is equal to that of the calling process
- pid > 0
  - wait for the child whose process ID is equal to the value of pid

wait( ) and waitpid( )

- option
  - an OR of zero or more of the following constants
  - WNOHANG
    - return immediately if no child has exited
  - WUNTRACED
    - also return for children which are stopped, and whose status has not been reported

wait( ) and waitpid( )

- macros
  - WIFSTOPPED(status)
    - returns true if the child process which caused the return is currently stopped
    - this is only possible if the call was done using WUNTRACED.
  - WSTOPSIG(status)
    - returns the number of the signal which caused the child to stop
    - this macro can only be evaluated if WIFSTOPPED returned non-zero

wait( ) and waitpid( )

- return value
  - The process ID of the child which exited
  - -1 on error
  - zero if WNOHANG was used and no child was available

wait( ) and waitpid( )

- example1

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
main()
{
        pid_t pid;
        int status;
        if ((pid = fork()) == 0) {          /* child process */
                printf("I am a child\n");
                exit(123);
        }
        /* parent process */
        pid = wait(&status);
        printf("parent: child(pid = %d) is terminated with status (%d)\n", pid,
                WEXITSTATUS(status));
}
```

wait( ) and waitpid( )

- example2

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/types.h>
main()
{
    pid_t pid;
    int status;
    if ((pid = fork()) == 0) {     /* child */
        printf("I am a child with pid = %d\n", getpid());
        sleep(60);
        printf("child terminates\n");
        exit(0);
    }
    else {                         /* parent */
        while (1) {
            waitpid(pid, &status, WUNTRACED);
            if (WIFSTOPPED(status)) {
                printf("child stopped, signal(%d)\n", WSTOPSIG(status));
                continue;
            }
```

wait( ) and waitpid( )

```
else if (WIFEXITED(status))
        printf("normal termination with status(%d)\n", WEXITSTATUS(status));
 else if (WIFSIGNALED(status))
         printf("abnormal termination, signal(%d)\n", WTERMSIG(status));
 exit(0);
                }
     }
}
```

```
$ a.out &
$ kill -STOP <child pid>
$ kill -CONT <child pid>
$ a.out &
$ kill -KILL <child pid>
```

wait( ) and waitpid( )

- example3

```
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdio.h>
void err_sys(char *s)
{
    perror(s);
    exit(1);
}
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);
        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");
exit(0);
}
```

exec Functions

- synopsis
  - #include <unistd.h>
  - extern char **environ;
  - int execl(const char *path, const char *arg, ...);
  - int execv(const char *path, char *const argv[]);
  - int execle(const char *path, const char *arg , ..., char * const envp[]);
  - int execve (const char *path, char *const argv[], char *const envp[]);

exec Functions

- int execlp(const char *file, const char *arg, ...);
- int execvp(const char *file, char *const argv[]);

exec Functions

- description
  - execute program
    - when a process calls one of the exec functions, that process is completely replaced by the new program
    - the new program starts executing at its main( ) function
    - exec merely replaces the current process (text, data, stack) with a brand new program from disk

exec Functions

- execve is a system call
  - execl, execv, execle, execlp, execvp are front-ends for the function execve( )
- pathname(filename)
  - the initial argument for these functions is the pathname(filename) of a file which is to be executed.
  - execl, execv, execle, execve take a pathname argument

exec Functions

- execlp, execvp take a filename argument
  - if filename contains a slash, it is taken as a pathname
  - otherwise, the executable file is searched for in the directories specified by the PATH environment variable
  - if PATH isn't specified, the default path ``:/bin:/usr/bin'' is used
  - if permission is denied for a file (the attempted execve returned EACCES), these functions will continue searching the rest of the search path

exec Functions

- argument list
  - the const char *arg and subsequent ellipses in the execl, execlp, and execle functions can be thought of as arg0, arg1, ..., argn
  - together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program
  - the first argument, by convention, should point to the file name associated with the file being executed

exec Functions

- the list of arguments must be terminated by a NULL pointer

- example
  - execl("/bin/ls", "ls", "-al", 0);

exec Functions

- argument vector
  - execv, execve and execvp provide an array of pointers to null-terminated strings that represent the argument list available to the new program
  - the first argument, by convention, should point to the file name associated with the file being executed
  - the array of pointers must be terminated by a NULL pointer
- example
  - char *argv[3] = {“ls”, “-al”, 0};
  - execv(“/bin/ls”, argv);

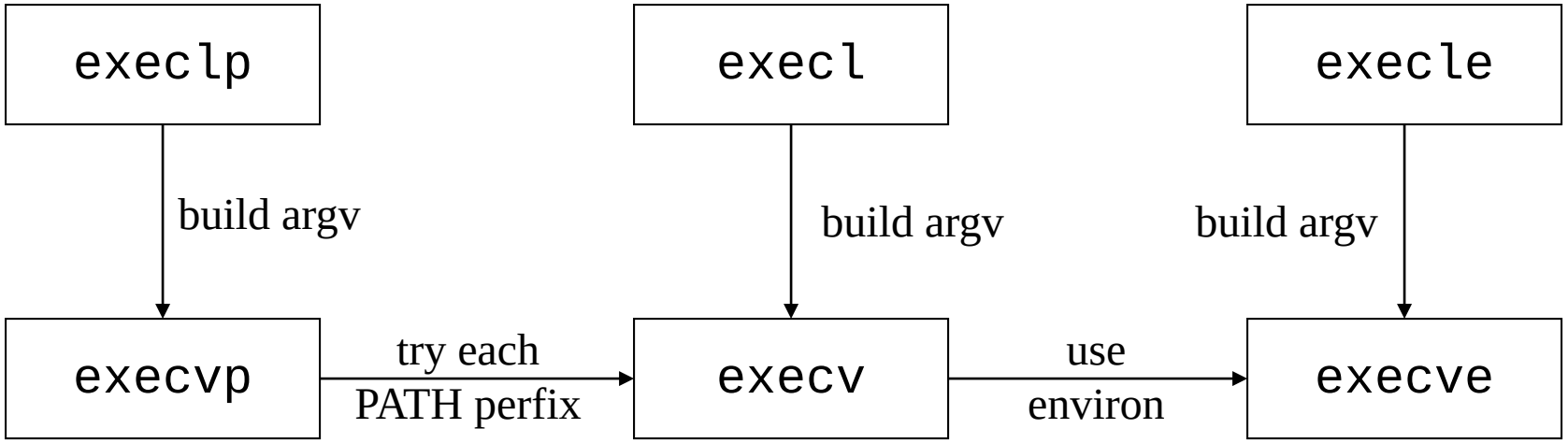
exec Functions

- environment variable
  - execve and execle specifies the environment of the executed process by following the NULL pointer that terminates the list of arguments in the parameter list or the pointer to the argv array with an additional parameter
  - this additional parameter is an array of pointers to null-terminated strings and must be terminated by a NULL pointer
  - the other functions take the environment for the new process image from the external variable environ in the current process

exec Functions

- example
  - char *env[2] = {“TERM=vt100”, 0};
  - execle(“/bin/ls”, “ls”, 0, env);

exec Functions



exec Functions

- example

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

      if ( (pid = fork()) < 0)
            printf("fork error");
      else if (pid == 0) {                  /* child */
            printf("Child : %d\n", getpid());
            if (execl("/usr/bin/ls", "ls", "-aCF", 0) < 0)
                  printf("execl error");
      }

      printf("Parent : %d\n", getpid());
      if (waitpid(pid, NULL, 0) < 0)  /* parent */
            printf("waitpid error");
      exit(0);
}
```

exec Functions

- properties that the new program inherits
  - pid, ppid, real-user(group)-id, supplementary gids
  - process group id, session id, controlling terminal
  - time left until alarm clock
  - current working directory, root directory, umask
  - process signal mask
  - pending signals
  - resource limits
- any signals set to be caught by the calling process are reset to their default behavior

Change uid and gid

- setuid( ) and setgid( )
  - #include <unistd.h>
  - int setuid(uid_t uid)
    - sets the effective user ID of the current process
    - superuser process sets all three user ID to uid
    - nonsuperuser process can set effective user ID to uid, only when uid equals real user ID or the saved set-user-ID
    - in any other cases, setuid returns error
  - int setgid(gid_t gid)

Change uid and gid

ID	exec		setuid(uid)	
	suid bit off	suid bit on	supersuer	other users
real UID	unchanged	unchanged	uid	unchanged
effective UID	unchanged	set from user ID of program file	uid	uid
save set-UID	copied from euid	copied from euid	uid	unchanged

Change uid and gid

- setreuid( ) and seteuid( )
  - #include <unistd.h>
  - int setreuid(uid_t ruid, uid_t euid);
    - sets real and effective user ID's of the current process
    - Un-privileged users may change the real user ID to the effective user ID and vice-versa
    - it is also possible to set the effective user ID from the saved user ID

Change uid and gid

- Supplying a value of -1 for either the real or effective user ID forces the system to leave that ID unchanged
- If the real user ID is changed or the effective user ID is set to a value not equal to the previous real user ID, the saved user ID will be set to the new effective user ID

- int seteuid(uid_t euid);
  - seteuid(euid) is functionally equivalent to setreuid(-1, euid)

Change uid and gid

- setuid-root program wishing to temporarily drop root privileges, assume the identity of a non-root user, and then regain root privileges afterwards cannot use setuid
- can accomplish this with seteuid

- int setregid(gid_t rgid, gid_t egid);
- int setegid(gid_t egid);

Change uid and gid

- int setfsuid(uid_t fsuid);
  - sets the user ID that the Linux kernel uses to check for all accesses to the file system
  - setfsuid will only succeed
    - if the caller is the superuser
    - if fsuid matches either the real user ID, effective user ID, saved set-user-ID, or the current value of fsuid
- int setfsgid(uid_t fsgid);

System Function

- synopsis
  - #include <stdlib.h>
  - int system (const char * string);
- description
  - executes a command specified in string by calling /bin/sh -c string, and returns after the command has been completed
  - during execution of the command, SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored

System Function

- return value
  - The value returned is 127 if the execve() call for /bin/sh fails
  - -1 if there was another error
  - return code of the command otherwise
  - If the value of string is NULL, system() returns nonzero if the shell is available, and zero if not
  - system() does not affect the wait status of any other children (waitpid)

User Identification

- synopsis
  - #include <unistd.h>
  - char *getlogin(void);
- description
  - returns a pointer to a string containing the name of the user logged in on the controlling terminal of the process.
  - returns a null pointer if this information cannot be determined.
  - The string is statically allocated

User Identification

- used when a single user has multiple login names
  - it is possible to use different login shell for each login name with multiple login names
  - can't determine login name with getpwuid(getuid())