

UNIX System Programming

Processes

❖ Objectives

- look at how to program UNIX processes
- `fork()` and `exec()`

1. What is a Process?
2. `fork()`
3. Example: `talkto.c`
4. `exec()`
5. `wait()`
6. Process Data
7. Special Exit Cases
8. Process IDs

Overview

1. What is a Process?
2. `fork()`
3. Example: `talkto.c`
4. `exec()`
5. `wait()`
6. Process Data
7. File Descriptors across Processes
8. Special Exit Cases
9. IO Redirection
10. User/Group ID, real and effective

1. What is a Process?

❖ A process is an executing program.

❖ A process:

```
$ cat file1 file2 &
```

❖ Two processes:

```
$ ls | wc -l
```

❖ Each user can run many processes at once (e.g. using `&`)

A More Precise Definition

❖ A process is the *context* (the information/data) maintained for an executing program.

What makes up a Process?

- ❖ program code
- ❖ data variables
- ❖ open files (file descriptors)
- ❖ an Environment (environment variables; credentials for security)

Some of the Context Information

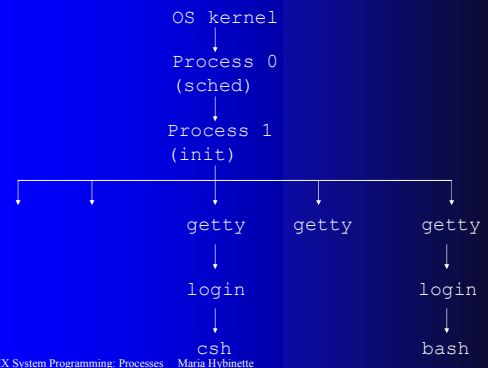
- Process ID (`pid`) unique integer
- Parent process ID (`ppid`)
- Real User ID ID of user/process which started this process
- Effective User ID ID of user who wrote the process' program
- Current directory
- File descriptor table
- Environment `VAR=VALUE` pairs

- Pointer to program code
- Pointer to data Memory for global vars
- Pointer to stack Memory for local vars
- Pointer to heap Malloc'd memory
- Execution priority
- Signal information

Important System Processes

- ❖ `init` – Mother of all processes. `init` is started at boot time and is responsible for starting other processes.
 - `init` uses file `inittab` & directories: `/etc/rc?.d`
- ❖ `getty` – login process that manages login sessions.

Unix Start Up Processes Diagram



Pid and Parentage

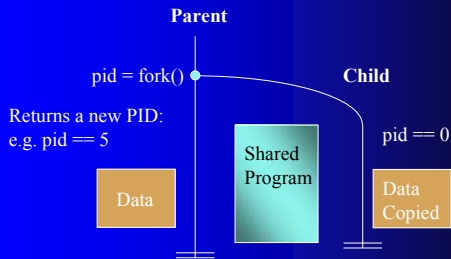
- ❖ A process ID or *pid* is a positive integer that uniquely identifies a running process, and is stored in a variable of type `pid_t`.
- ❖ You can get the process `pid` or parent's `pid`

```
#include <sys/types>
main()
{
    pid_t pid, ppid;
    printf( "My PID is:%d\n", (pid = getpid()) );
    printf( "Par PID is:%d\n", (ppid = getppid()) );
}
```

2. fork()

- ❖ `#include <sys/types.h>`
`#include <unistd.h>`
`pid_t fork(void);`
- ❖ Creates a child process by making a copy of the parent process.
- ❖ Both the child *and* the parent continue running.

fork() as a diagram



Process IDs (pids revisited)

- ❖ `pid = fork();`
- ❖ In the child: `pid == 0;`
In the parent: `pid ==` the process ID of the child.
- ❖ A program can use this `pid` difference to do different things in the parent and child.

fork() Example (parchld.c)

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

int main()
{
    pid_t pid;        /* could be int */
    int i;
    pid = fork();
    if (pid > 0)
    {
        /* parent */
        for( i=0; i < 1000; i++ )
            printf("\t\t\tPARENT %d\n", i);
    }
}
```

```
else
{
    /* child */
    for( i=0; i < 1000; i++ )
        printf( "CHILD %d\n", i );
}
return 0;
}
```

Possible Output

```
CHILD 0
CHILD 1
CHILD 2

PARENT 0
PARENT 1
PARENT 2
PARENT 3

CHILD 3
CHILD 4

PARENT 4

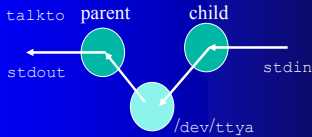
:
```

Things to Note

- ❖ `i` is copied between parent and child.
- ❖ The switching between the parent and child depends on many factors:
 - machine load, system process scheduling
- ❖ I/O buffering effects amount of output shown.
- ❖ Output interleaving is *nondeterministic*
 - cannot determine output by looking at code

3. Example: talkto.c

- ❖ A simple communications program that copies chars from `stdin` to a specified port, and from that port to `stdout`.
- ❖ Use port at `/dev/ttya`



Code for talkto.c

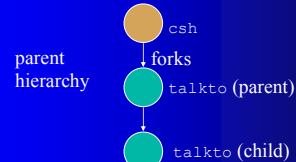
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

int main()
{ int fd, count;
  char buffer[BUFSIZ];
  if (fd = open( "/dev/tty", O_RDWR ) < 0)
  { fprintf(stderr, "Cannot open port\n");
    exit(1);
  }
  :
```

```
if (fork() > 0)
{ /* parent */
 /* copy port input to stdout */
 while(1)
 {
  count = read( fd, buffer, BUFSIZ );
  write( 1, buffer, count );
 }
}
else
{ /* child: copy stdin to port */
 while(1)
 {
  count = read( 0, buffer, BUFSIZ );
  write( fd, buffer, count );
 } /* else */
}
return 0;
} /* main */
```

ps Output

```
$ ps -l
UID  PID  PPID  CMD
500  4712  4711  ksh
500  4983  4712  talkto
500  4984  4983  talkto
500  4992  4712  ps
```



4. exec()

- ❖ Family of functions for replacing process's program with the one inside the `exec()` call. e.g.

```
#include <unistd.h>
int execlp(char *file, char *arg0,
           char *arg1, ..., (char *)0);

execlp("sort", "sort", "-n",
       "foobar", (char *)0);
```

Same as "sort -n foobar"

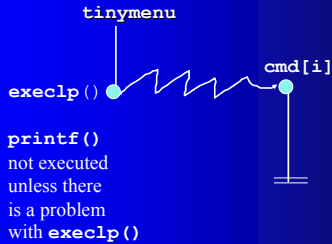
tinymenu.c

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

void main()
{ char *cmd[] = {"who", "ls", "date"};
  int i;
  printf("\0=who 1=ls 2=date : ");
  scanf("%d", &i);

  execlp( cmd[i], cmd[i], (char *)0 );
  printf( "execlp failed\n" );
}
```

Execution

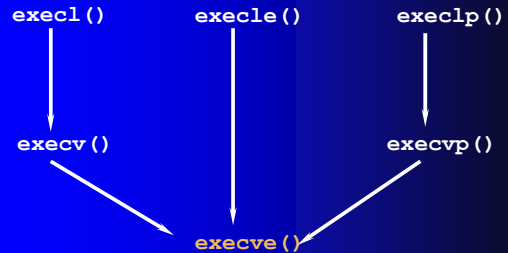


exec(..) Family

- There are 6 versions of the exec function, and they all do about the same thing: they replace the current program with the text of the new program. Main difference is how parameters are passed.

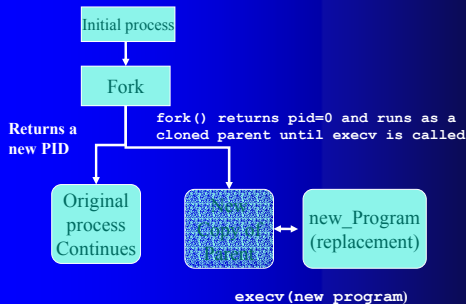
```
int execl( const char *path, const char *arg, ... );
int execlp( const char *file, const char *arg, ... );
int execl( const char *path, const char *arg
    , ..., char *const envp[] );
int execv( const char *path, char *const argv[] );
int execvp( const char *file, char *const argv[] );
int execve( const char *filename, char *const argv [],
    char *const envp[] ); // system call
```

exec(..) Family



fork() and execv()

- execv(new_program, argv[])



5. wait()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
```

- Suspends calling process until child has finished. Returns the process ID of the terminated child if ok, -1 on error.

- statloc can be (int *)0 or a variable which will be bound to status info. about the child.

wait() Actions

- ❖ A process that calls `wait()` can:
 - *suspend* (block) if all of its children are still running, or
 - *return* immediately with the *termination* status of **a** child, or
 - *return* immediately with an *error* if there are no child processes.

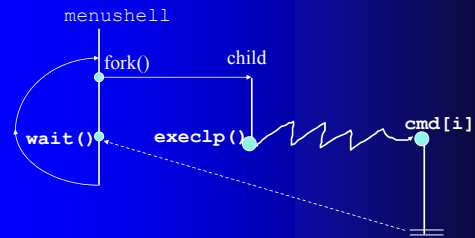
menushell.c

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

void main()
{
    char *cmd[] = {"who", "ls", "date"};
    int i;
    while( 1 )
    {
        printf( "0=who 1=ls 2=date : " );
        scanf( "%d", &i );
        ;
    }
}
```

```
if(fork() == 0)
{ /* child */
    execlp( cmd[i], cmd[i], (char *)0 );
    printf( "execlp failed\n" );
    exit(1);
}
else
{ /* parent */
    wait( (int *)0 );
    printf( "child finished\n" );
}
} /* while */
} /* main */
```

Execution



Macros for wait (1)

- ❖ `WIFEXITED(status)`
 - Returns true if the child exited normally.
- ❖ `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.

Macros for wait (2)

- ❖ `WIFSIGNALED(status)`
 - Returns true if the child process exited *because of a signal* which was not caught.
- ❖ `WTERMSIG(status)`
 - Returns *the signal number* that caused the child process to terminate.
 - This macro can only be evaluated if `WIFSIGNALED` returned non-zero.

waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid( pid_t pid, int *status, int opts )
```

- ❖ `waitpid` - can wait for a particular child
- ❖ `pid < -1`
 - Wait for any child process whose process group ID is equal to the absolute value of `pid`.
- ❖ `pid == -1`
 - Wait for any child process.
 - Same behavior 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`.
- *options*
 - ❖ Zero or more of the following constants can be ORed.
 - `WNOHANG`
 - Return immediately if no child has exited.
 - `WUNTRACED`
 - Also return for children which are stopped, and whose status has not been reported (because of signal).
- Return value
 - ❖ The process ID of the child which exited.
 - ❖ -1 on error; 0 if `WNOHANG` was used and no child was available.

Macros for waitpid

- ❖ `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 signal number* which caused the child to stop.
 - This macro can only be evaluated if `WIFSTOPPED` returned non-zero.

Example: waitpid

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

int main(void)
{
    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;
        }
        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);
    } /* while */
} /* parent */
} /* main */
```

6. Process Data

- ❖ Since a child process is a **copy** of the parent, it has copies of the parent's data.
- ❖ A change to a variable in the child will *not* change that variable in the parent.

Example (globex.c)

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

int globvar = 6;
char buf[] = "stdout write\n";

int main(void)
{
    int w = 88;
    pid_t pid;
    :
```

```
write( 1, buf, sizeof(buf)-1 );
printf( "Before fork()\n" );

if( (pid = fork()) == 0 )
{ /* child */
    globvar++;
    w++;
}
else if( pid > 0 ) /* parent */
    sleep(2);
else
    perror( "fork error" );

printf( "pid = %d, globvar = %d, w = %d\n",
        getpid(), globvar, w );
return 0;
} /* end main */
```

Output

```
❖ $ globex
stdout write /* write not buffered */
```

```
Before fork()
pid = 430, globvar = 7, w = 89
/*child chg*/
pid = 429, globvar = 6, w = 88
/* parent no chg */
```

```
❖ $ globex > temp.out
$ cat temp.out
stdout write
Before fork()
pid = 430, globvar = 7, w = 89
Before fork() /* fully buffered */
pid = 429, globvar = 6, w = 88
```

7. Process File Descriptors

- ❖ A child and parent have copies of the file descriptors, but the R-W pointer is maintained by the system:
 - the R-W pointer is shared

- ❖ This means that a `read()` or `write()` in one process will affect the other process since the R-W pointer is changed.

Example: File used across processes (shfile.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <fcntl.h>
void printpos(char *msg, int fd);
void fatal(char *msg);

int main(void)
{ int fd; /* file descriptor */
  pid_t pid;
  char buf[10]; /* for file data */
  :
```

```
if ((fd=open("data-file", O_RDONLY)) < 0)
    perror("open");

read(fd, buf, 10); /* move R-W ptr */
printpos( "Before fork", fd );

if( (pid = fork()) == 0 )
{ /* child */
    printpos( "Child before read", fd );
    read( fd, buf, 10 );
    printpos( " Child after read", fd );
}
:
```

```

else if( pid > 0 )
{
    /* parent */
    wait((int *)0);
    printpos( "Parent after wait", fd );
}
else
    perror( "fork" );
}

```

```

void printpos( char *msg, int fd )
/* Print position in file */
{
    long int pos;

    if( (pos = lseek( fd, 0L, SEEK_CUR ) < 0L )
        perror( "lseek" );

    printf( "%s: %ld\n", msg, pos );
}

```

Output

```
$ shfile
```

```

Before fork: 10
Child before read: 10
Child after read: 20
Parent after wait: 20

```

what's happened?

8. 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 until the parent does a `wait()`

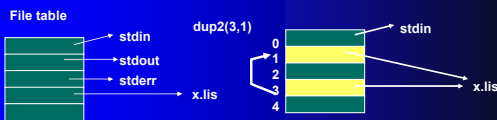
- ❖ 2) A parent exits when 1 or more children are still running
 - children are adopted by the system's initialization process (`/etc/init`)
 - ◆ it can then monitor/kill them

9. I/O redirection

- ❖ The trick: you can change where the standard I/O streams are going/coming from after the fork but before the `exec`

Redirection of standard output

- ❖ Example implement shell: `ls > x.lis`
- ❖ program:
 - Open a new file `x.lis`
 - Redirect standard output to `x.lis` using `dup` command
 - ♦ everything sent to standard output ends in `x.lis`
 - execute `ls` in the process
- ❖ `dup2(int fin, int fout)` - copies `fin` to `fout` in the file table



Example - implement `ls > x.lis`

```
#include <unistd.h>
int main ()
{
    int fileId;
    fileId = creat( "x.lis", 0640 );
    if( fileId < 0 )
    {
        printf( stderr, "error creating x.lis\n" );
        exit (1);
    }
    dup2( fileId, stdout ); /* copy fileId to stdout */
    close( fileId );
    execl( "/bin/ls", "ls", 0 );
}
```

10. User and Group ID

- ❖ Group ID
 - Real, effective
- ❖ User ID
 - Real user ID
 - ♦ Identifies the user who is responsible for the running process.
 - Effective user ID
 - ♦ Used to assign ownership of newly created files, to check file access permissions, and to check permission to send signals to processes.
 - ♦ To change `euid`: executes a `setuid-program` that has the `set-uid` bit set or invokes the `setuid()` system call.
 - ♦ The `setuid(uid)` system call: if `euid` is not superuser, `uid` must be the real `uid` or the saved `uid` (the kernel also resets `euid` to `uid`).
 - Real and effective `uid`: `inherit` (`fork`), `maintain` (`exec`).

Read IDs

- ❖ `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

Change UID and GID (1)

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

- ❖ Sets the effective user ID of the current process.
 - Superuser process resets the real effective user IDs to `uid`.
 - Non-superuser process can set effective user ID to `uid`, only when `uid` equals real user ID or the saved set-user ID (set by executing a `setuid-program` in `exec`).
 - In any other cases, `setuid` returns error.

Change UID and GID (2)

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

Change UID and GID (3)

```
#include <unistd.h>
#include <sys/types.h>
int seteuid( uid_t ruid, uid_t euid )
```

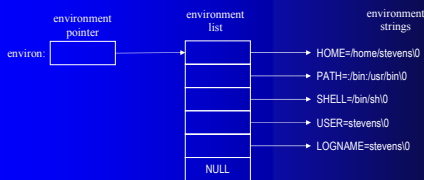
- ❖ Sets real and effective user ID's of the current process.
- ❖ Un-privileged users may change the real user ID to the effective user ID and vice-versa.
- ❖ It is also possible to set the effective user ID from the saved user ID.
- ❖ Supplying a value of -1 for either the real or effective user ID 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.

Change UID and GID (4)

- int seteuid(uid_t euid);
 - ❖ seteuid(euid) is functionally equivalent to seteuid(-1, euid).
 - ❖ 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, because setuid issued by the superuser changes all three IDs. One can accomplish this with seteuid.
- int setegid(gid_t rgid, gid_t egid);
- int setegid(gid_t egid);

11. Environment

```
❖ extern char **environ;
int main( int argc, char *argv[], char *envp[] )
```



Example: environ

```
#include <stdio.h>
void main( int argc, char *argv[], char *envp[] )
{
    int i;
    extern char **environ;

    printf( "from argument envp\n" );

    for( i = 0; envp[i]; i++ )
        puts( envp[i] );

    printf("\nFrom global variable environ\n");

    for( i = 0; environ[i]; i++ )
        puts( environ[i] );
}
```

getenv

- ```
❖ #include <stdlib.h>
char *getenv(const char *name);
```
- Searches the environment list for a string that matches the string pointed to by *name*.
  - Returns a pointer to the value in the environment, or NULL if there is no match.

## putenv

- ```
❖ #include <stdlib.h>
int putenv(const char *string);
```
- Adds or changes the value of environment variables.
 - The argument *string* is of the form name=value.
 - If name does not already exist in the environment, then string is added to the environment.
 - If name does exist, then the value of name in the environment is changed to value.
 - Returns zero on success, or -1 if an error occurs.

Example : getenv, putenv

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

void main(void)
{
    printf("Home directory is %s\n", getenv("HOME"));
    putenv("HOME=/");
    printf("New home directory is %s\n", getenv("HOME"));
}
```