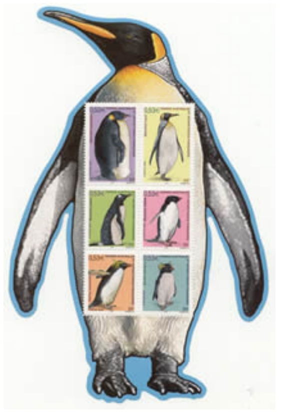
Systems Calls - Procs

Alain NINANE - UCL

15 mars 2016



Process - A definition (I)

- **A process is a program being executed**
  - program: a static object
    - An executable file on the file system
    - A file with the mode ‘x’ (executable) bit set
    - Executable program file
      - machine readable code (a compiled program)
      - human readable code (a program for an interpreter)
        - the interpreter is itself a compiled program
  - process: a dynamic object
    - transient in memory during execution

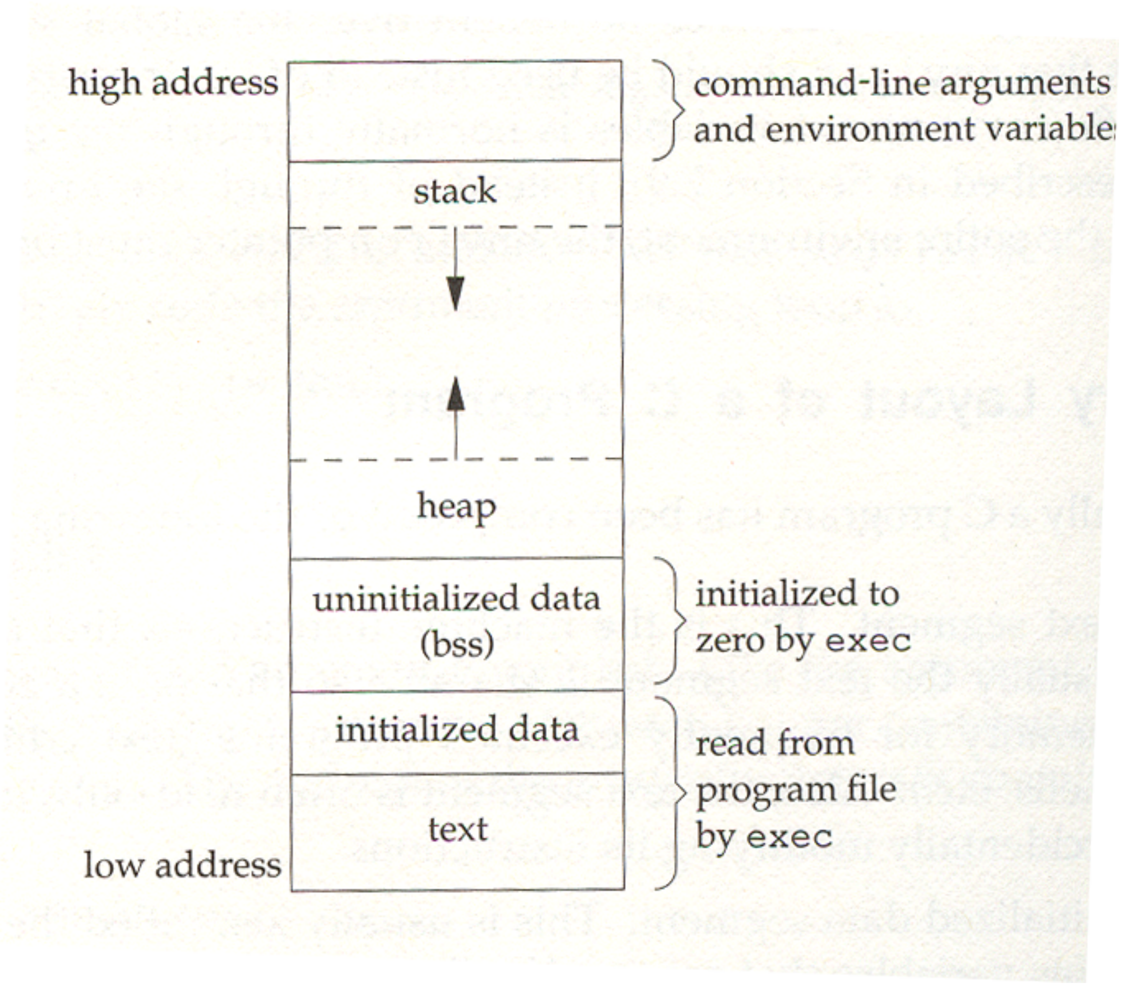
Process/Program - Examples

- Compiled program
  - Needs compilation: cc -o test_01 test_01.c
  - Demonstration
    - ./test_01 &
      - pstree -G -p | more -c
      - ps axjf | grep 500
- Interpreted program
  - Needs change mode: chmod +x test_02.sh test_03.pl
  - shell
    - ./test_02.sh &
      - ps axjf | grep 500   # 500 is my own uid on the test system !!
  - perl
    - ./test_03.pl &
      - ps axjf | grep 500
  - Note: magic characters (/usr/share/file/magic)

Process - A definition (II)

- A process is a program being executed
- **A process contains in memory data**
  - **Text**: the instructions to be executed
    - size a.out
    - nm a.out
    - objdump –d a.out
  - **Global data**: initialized or not
  - **Local data**: on the stack
  - **Heap:** free memory for malloc()
  - **<u>Execution environment</u>**:
    - Program: counter, status, ...
    - System: file table, working dir., control. terminal, user, ...
    - **<u>Environment variables</u>**: system & user defined values
      - Environment variables is a set of pair variables/values that can be accessed by the process **<u>and its childrens</u>**

Process - In Memory Data


high address
command-line arguments and environment variable
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Environment - variables (I)

- A process holds environment variables
  - “System V family shell”
    - printenv, export
  - “Berkeley family shell”
    - printenv
- Settings users’s defined environment variables
  - “System V family shell”
    - export MYENVDATA=“/tmp”
  - “Berkeley family shell”
    - setenv MYENVDATA /tmp

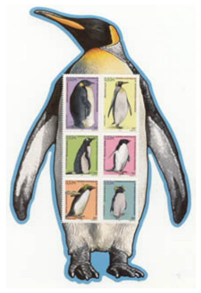
Environment - variables (II)

- Environment is a set of pairs
  - name=value
  - Example:
    - PATH=.:/usr/bin:/bin
    - USER=nina
    - HOME=/disks/home/n/nina
    - TERM=vt100
  - Access from a user's program
    - **extern char **environ**; /* or better !!!! */
    - char *getenv(const char *name);

Environment - PATH explained

- The PATH variable
  - Directory search list for executables programs
  - Example:
    - /usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/home/nina/bin
      - a standard search path
    - .:/home/nina/bin:/usr/local/bin:/bin/....
      - a path with priority to owner & local directories
    - .:/usr/ucb/bin:/usr/bin:....
      - a specialized path on a Sun Solaris system (System V) with priority to compatibility Berkeley commands
      - example:
        - BSD: ps aux
        - System V: ps -eaf

strace demonstration

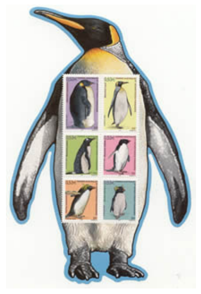


- strace sh -c ls 2>&1 | more -cf
  - Illustrates
    - system calls:
      - execve(), mmap(), open(), read(), fstat(), lseek()
    - shared libraries
      - /etc/ld.so.cache
      - /lib64/libdl.so.2
    - observation:
      - mmap NOT unmapped by close
    - search path
      - through stat() system call

Process - A definition (III)

- A process is a program being executed
- A process holds in memory data
- **A process**:
  - **is identified by a number**, unique in the UNIX system
    - the **pid** : process identifier
      - pids are positive 16 bits integer
      - maximum of ~32.000 processes
  - **executes on behalf of a user**
    - the **uid** : identifies the user executing the process

Process - Starting a process



- **Two** point of view
  - at the command level:
    - by means of a command interpreter (e.g. shell)
  - from within a program:
    - by means of system calls
- **One** point of view
  - The **ONLY** way to create a new process is when an **existing** process calls the **fork()** syscall.

fork - Process creation syscall

- #include <sys/types.h>
- #include <unistd.h>
- pid_t fork(void)
  - **The existing process is duplicated**
  - Processes have:
    - same copies of in-memory data, *with some exceptions ...*
  - Returns:
    - the original (**parent**) process gets child's pid
    - the forked (**child**) process gets 0
  - Errors:
    - returns -1 (set errno)

fork - The simplest example

```
#include <sys/types.h>
#include <unistd.h>
main()
{   pid_t ret = fork();
        if( ret > 0 )
        {       /* ret > 0 -> parent code */
                myPid = getpid();  /* ret != myPid !!! */
        }
        else  if( ret == 0 )
        {       /* ret == 0 -> child code */
                myPid = getpid();
        }
        else
        {       /* ret < 0 -> parent in error */
        }
}
```

fork - shared/unshared data

- Both processes **share**
  - same code
  - same values in variables (except ret value)
  - same open files, file pointers and file descriptors
  - same current working directory
  - same controlling terminal/window
- Both processes **do not share**
  - execution time counters (cleared for the child)
  - semaphore values
  - file locks
  - pending alarm or signal (cleared for the child)

fork - special cases

- Experience:
  - ./test_04_[init|zombie] &
  - ps -U nina lx
- **the parent terminates before the child**
  - the child is attached to process “init” (PPID = 1)
- **the child terminates before the parent**
  - the child becomes a zombie (Z status)
  - a terminated child still exists until the parent:
    - terminates
    - check the child exit() value e.g. with wait() syscall
  - a zombie process doesn’t consumes *any* ressources
    - except entries in the process table

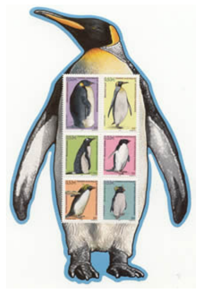
fork - synchronization

- A parent process must sometimes wait until one of its children terminates:
  - synchronization
  - **system calls wait() and exit()** [or **_exit()]**
- General principe (demo test_05)
  - if(fork())
    - /* parent process */
    - wait()
  - else
    - /* child process */
    - exit(value)
  - father continues ...

wait - synchronization

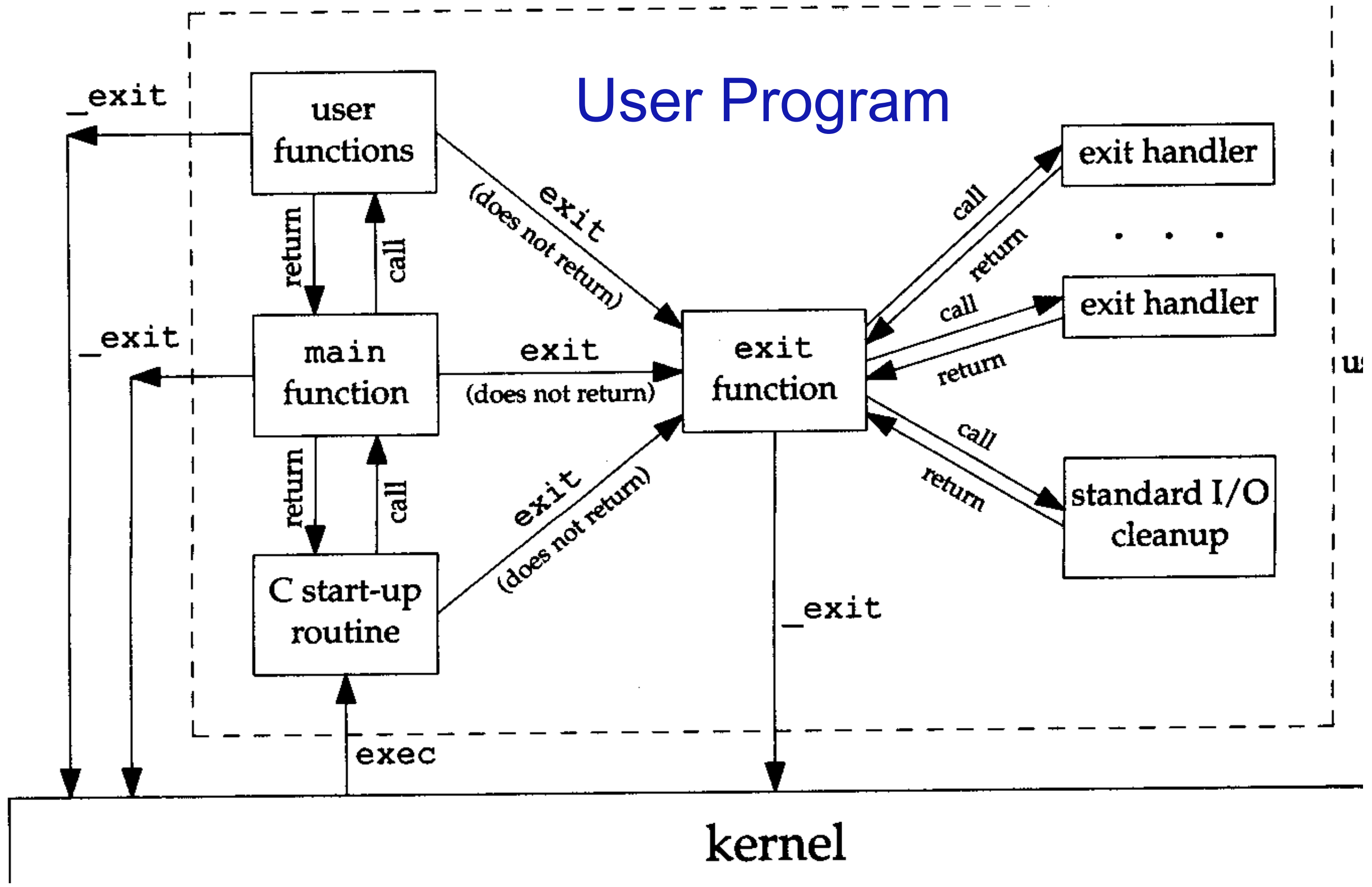
- #include <sys/types.h>
- #include <sys/wait.h>
  - pid_t wait(int *status);
  - pid_t waitpid(pid_t pid, int *status, int options);
- Wait for a child to terminates
  - suspend execution of the “current” process until one of the childrens terminates
  - **returns pid of waited child** or -1
  - can wait for a specific child
  - options & status: read the manual
    - **<u>can</u>** contains exit_value
- A process can wait for several processes
  - Implies several wait() or waitpid() calls !

exit - exit() vs _exit()



- _exit(status) is the “real” system call
  - terminates the current process immediately
    - opened fd’s are closed
    - children are “moved” to another parent (init - pid 1)
    - parent’s process receive a SIGCHILD signal
    - status is returned to the parent process
- exit(status) is a function
  - executes terminations functions
    - such as flushing streams
    - call void (*functions)(void) added to the process with **atexit()**
  - call _exit(status)
- demonstration: **test_06** and **test_06_exit**

exit - execution flow



exit - programming technique

- exit() should be used in place of _exit()
  - calls exit handlers
  - flush stdio streams
- Convention ...
  - program terminates normally: exit(0)
  - program terminates badly:exit(something)
    - something is between 1 and 255 (8 bits)
    - -1 --> 255
  - exit value can be checked by parent process
- Demo: check return value of test_06[_exit]
  - ./test_06[_exit]
  - echo $?

fork - starting processes

- So far so good ...
- fork() starts a new process which is ...
  - just a copy of the parent process ...
- so we just have
  - a process tree (arborescence) of clones
- so we need something more
  - exec() family of functions

exec - executes a program

- executes - not start !!
- process is already started by fork
- **exec()** will overlay the current process with a new one taken from a new image file
- the process continue at the main() function of the image file
- since the calling image is lost, **exec()** should never returns

exec - the simplest example

```
#include <stdio.h>
main()
{   if( fork() == 0 )
    {       execl("/bin/ls","ls","-l","/etc",(char*)NULL);
             /* SHOULD NEVER GETS HERE */
             exit(1);
    }
    /* father process waits */
    wait(&status);
    exit(status);
}
```

exec - a family of function

- **exec()** is a family of functions and 1 system call

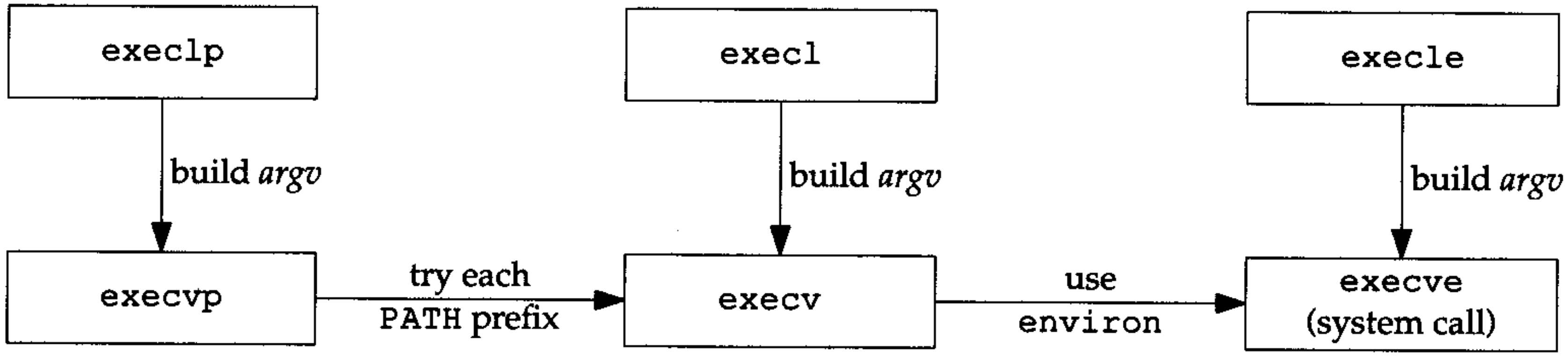
```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *) 0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *) 0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *) 0 */ );

int execvp(const char *filename, char *const argv[]);
```

All six return: –1 on error, no return on success

exec - a family of functions

- functions differs by:
  - enabling or not the search PATH
    - execl, execv: pathname must be exact
    - execlp, execvp: search pathname in PATH
  - passing or not the environment
    - execle, execve: environment variables passed as an array
    - extern char **environ;
  - passing arguments “inline” or through array
    - execl, execlp: args as inline
    - execv, execvp: args as array
- 1 system call: **execve()**

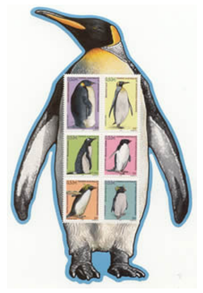
exec - 6 functions pictures



exec - i/o redirection

- Example - i/o redirection in child process */

```
main()
{   if ( fork() == 0 )
     {     close(1);
           open(“myoutput”, O_CREAT|O_RDWR,0644);
           execl(“/bin/date”,”date”,(char*)NULL);
     }
     wait();
     …
     exit(0);
}
```



exec - pass commands to shell

- Example - pass command to a new shell */

```
main()
{   if ( fork() == 0 )
     {          execl("/bin/sh","sh",
                      "-c","ls /etc | wc > test_08.out",(char*)NULL);
      }
      wait();
      exit(0);
}
```

- Can be replaced by the system() function
  - system("ls /etc | wc > test_08.out");