Processes @ POSIX/UNIX/Linux

POSIX 1003.1

„**Process** – an address space with one or more threads executing within that address space, and the required system resources for those threads”.

Note: Many of the system resources defined by IEEE Std 1003.1-2001 are shared among all of the threads within a process. These include the process ID, the parent process ID, process group ID, session membership, real, effective, and saved set-user-ID, real, effective, and saved set-group-ID, supplementary group IDs, current working directory, root directory, file mode creation mask, and file descriptors.

Unix processes

- A process is a program in execution
- Processes are identified by their process identifier (PID), an integer
- Each process performs its operation on behalf of a user, characterized with a unique user id (UID) and a group id (GID). UID and GID determine access rights to system resources
- Linux defines **personality identifiers** that can modify semantics of some system calls (to make the calls compatible with variants of UNIX)

Example. Displaying processes of a user lopalski (UID=1253)

$ **ps -lu lopalski**

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
8	O	1253	14457	14379	0	50	20	?	142		pts/24	0:00	ps
8	S	1253	14379	14364	0	50	20	?	423	?	pts/24	0:00	zsh

Unix processes – cont.

- All processes are descendants of an initial system process running with PID=1, UID=0, GID=1 (**init** or its equivalent, notably **systemd**).

Example. Gnu/Linux/Debian – system processes

```
UID    PID  PPID C STIME TTY          TIME CMD
root     1    0    1 16:40 ?        00:00:01 init [2]
root     2    1    0 16:40 ?        00:00:00 [ksoftirqd/0]
root     3    1    2 16:40 ?        00:00:02 [events/0]
root     4    3    0 16:40 ?        00:00:00 [khelper]
root     5    3    0 16:40 ?        00:00:00 [kacpid]
root    29    3    0 16:40 ?        00:00:00 [kblockd/0]
root    39    3    0 16:40 ?        00:00:00 [pdflush]
root    40    3    0 16:40 ?        00:00:00 [pdflush]
root    41    1    0 16:40 ?        00:00:00 [kswapd0]
root    42    3    0 16:40 ?        00:00:00 [aio/0]
.....
daemon 1419   1    0 16:40 ?        00:00:00 /sbin/portmap
root   1753   1    0 16:40 ?        00:00:00 /sbin/syslogd
root   1756   1    0 16:40 ?        00:00:00 /sbin/klogd
.....
root   1797   1    0 16:40 ?        00:00:00 /usr/sbin/inetd
lp     1801   1    0 16:40 ?        00:00:00 /usr/sbin/lpd -s
root   1808   1    0 16:40 ?        00:00:00 /usr/sbin/sshd

root   1857   1    0 16:40 tty2     00:00:00 /sbin/getty 38400 tty2
.....
```

```
PC_lobook:~# pstree -A
init-+-atd
     |-bash---pstree
     |-cron
     |-dhclient
     |-events/0-+-aio/0
     |          |-kacpid
     |          |-kblockd/0
     |          |-khelper
     |          `-2*[pdflush]
     |-exim4
     |-5*[getty]
     |-inetd
     |-khubd
     |-kjournald
     |-klogd
     |-kseriod
     |-ksoftirqd/0
     |-kswapd0
     |-lpd
     |-portmap
     |-rpc.statd
     |-sshd
     `-syslogd
```

Unix processes – cont.

The number, names, PIDs, UIDs/GIDs of system processes differ between UNIX-alike systems, but the **init/systemd** process always exists, has the same PID, UID, GID and mission

Example. SunOS 5.9 – selected system processes

```
    UID   PID  PPID  C    STIME TTY      TIME CMD
    root    0    0   0  04:52:16 ?       0:01   sched
    root    1    0   0  04:52:16 ?       0:03   /etc/init -
    root    2    0   0  04:52:16 ?       0:00   pageout
    root    3    0   1  04:52:16 ?      24:56   fsflush
...
   root  282    1    0  04:52:46 ?       0:01  /usr/sbin/rpcbind

    root  305    1    0  04:52:47 ?       0:00  /usr/sbin/inetd -s
    root  372    1    0  04:52:49 ?       0:55  /usr/lib/autofs/automountd
    root  367    1    0  04:52:48 ?       0:00  /usr/lib/nfs/lockd
daemon  369    1    0  04:52:48 ?       0:00  /usr/lib/nfs/statd
   root  400    1    0  04:52:49 ?       0:00  /usr/sbin/cron
...
    root 1183  1135 0 15:59:16 ?       0:00  dtgreet -display :17
...
```

Run-time process environment

- **Environment variables** are process-accessible strings of the form:

```
name=value
```

Common environment variables::

PATH - a list of paths to programs/scripts which are directly accessible to system shell as external
HOME - home directory of a logged in user
PWD - current directory of a shell
PS1, PS2 – first and second level shell prompts
TERM - terminal type connected to the shell
SHELL - current shell
LOGNAME –logged in user name
RANDOM- random number
EDITOR - current (default) user editor
PPID - identifier of the parent process

- Environment variables are inherited by a child process created with **`fork()`** function call. For **`execle()`**, **`execve()`** calls the set of environment variables can be defined anew; for other functions of exec group – the environment variables are not changed.

Run-time proces environment

- Shell maintains a list of variables. Typical command that creates/assigns value to the variable is

```
variable_name=variable_value
```

The command

```
export   variable_name
```

makes the named variable available to newly created children as their **environment variable**. Values of the variables can be retrieved by a process:

- A C language program can create/modify an environment variables using

```
putenv("name=value");
```

- in a C-program, a value of an environment variable can be retrieved using `getenv()` function, e.g.:

```
char *p=getenv("name");
if(p) printf ("name=%s\n",p);
else printf(„variable name is undefined\n");
```

Run-time proces environment

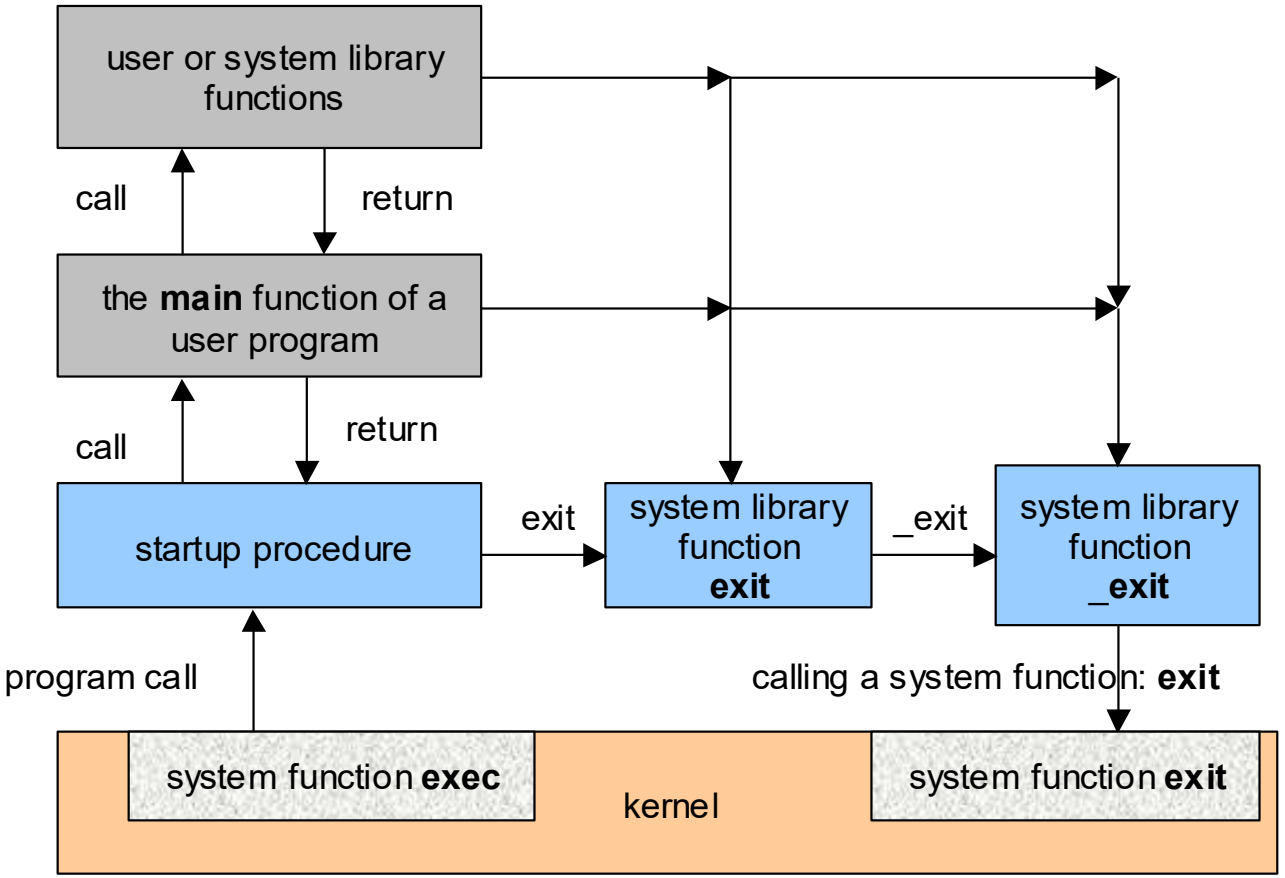
POSIX:

At program start-up, three **streams** are predefined and need not be opened explicitly:

- **standard input** (for reading conventional input),
- **standard output** (for writing conventional output), and
- **standard error** (for writing diagnostic output).

When opened, the standard error stream is **not fully buffered**; the standard input and standard output streams are **fully buffered** if and only **if the stream** can be determined **not to refer to an interactive device.**

Process: from creation till termination



From process creation (system function **exec**) till process termination (system function **exit**)

Process life-cycle

- Process control system calls :
  - **fork** creates a new process (a logical copy of the parent process)
  - **exec** is used after a fork to replace one of the two processes' virtual memory space with a new program
  - **_exit** function terminates process, storing termination status
  - A parent may **`wait`** for a child process to terminate; **`wait`** provides PID and status of a terminated child so that the parent can tell which child terminated and why.

```
pid_t pid1, pid2;
pid1 = fork(); /* duplicating the current process */
if (pid1 < 0) { perror("Fork failed"); exit(1);
} else if (pid1 == 0) { /* child process  */
                execl("/bin/ls", "ls", „-l", NULL); /* Exit code of this process depends here on the code of /bin/ls */
                perror(”execlp”); /* this line gets executed ONLY when execl() failed */ exit(2);
} else { /* parent process */
                if((pid2=wait (&status))>0){/* waiting for a child */
                                fprintf (stderr, "Potomek: PID=%d, status=%d\n”,(int)pid2,status);
                } else {        perror(”wait”); exit(3); }
                                return(0); /* same as exit(0) */
                }
}
```

Process life-cycle

- Child process resulting from `fork()` call inherits from its parent:
  - Content of the address space (except for a value returned by **fork**() call itself.
  - File descriptors
  - Environment variables
  - Mapping of files into memory (`mmap()`)
  - Signal handling
  - Priority and scheduling policy
- Child process does not inherit
  - State of process timers
  - Awaiting signals
  - Asynchronous operations
  - Threads, other than that which made the `fork()` call

Process life-cycle

- A process resulting from **exec** system call inherits the caller:
  - PID, PPID,PGID, RUID, RGID, state of alarms
  - Current and home directories, **umask**, **ulimit**
  - File descriptors, except these which were marked as close-on-exec
  - Environment variables – if not modified call through **`execle()`**, **`execve()`** library functions
  - Control terminal
  - Resource limits
- Signal mask and waiting signals, Note, that default signals with default disposition (SIG_DFL) and ignored (SIG_IGN) keep the signal handling – except SIGCHLD (if it was ignored – SIG_DFL or SIG_IGN disposition results). Signals, which had their user handler set change their disposition to default.
- Status of the floating point unit is reset to initial
- Counter of CPU use is not reset..

Process life-cycle – cont.

- There are two kinds of termination:
  - Normal: by a return from main( ), when requested with the **exit()** , **_exit()** functions
  - Abnormal: when requested by the **abort()** or when some signals are received.
- Upon termination the process file descriptors, directory streams, conversion descriptors and message catalog descriptors open in the calling process shall be closed.
- If the parent process of the calling process has set its SA_NOCLDWAIT flag or has set the action for the SIGCHLD signal to SIG_IGN
  - the process' status information shall be discarded and the lifetime of the calling process shall end immediately.
  - Otherwise, status information shall be generated, and the terminating process shall be transformed into a **zombie** process. The status information shall be made available to the parent process (via **wait( ), waitid()** or **waitpid()** until the process' lifetime ends. Besides SIGCHLD shall be sent to the parent process.
- A **zombie** process - the remains of a live process after it terminates and before its status information is consumed by its parent process.
- **Orphaned** process – a process whose parent process has exited.
- After parent's lifetime has ended, its child processes are inherited by an implementation-defined system process (**init/system,…**). Note: for UNIX that system "substitute parent" process has PID==1. POSIX does not impose PID==1 requirement.

Process creation/control/termination

`pid_t fork(void)`

creates a logical copy of the current process; returns:

–1 when fails (error code in **`errno`**)

0 when stepping into the child process

>0 (child PID) when returning to the parent process

`pid_t vfork(void)`

creates a logical copy of the current process, which shares address space with the parent process. The parent process is stopped till the child process calls **`exec()`** or **`_exit()`** system functions. The child should not call library **`exit()`** function, since it flushes I/O buffers and closes open descriptors (which are shared with the parent). Using **`vfork()`** for large processes saves system effort on creation a copy of the parent – if the child immediately calls **`exec`** and so discards the allocated resources. **Not a POSIX function**.

Process creation/control/termination – cont.

```
int clone(int (*fn)(void *), void *child_stack,
          int flags, void *arg)
```

creates a new process, that can share a selected subset of parent's context. The new proces executes `fn(arg)` (compare to `fork`() !), and when this function executes return – the child is terminated (unless it calls `exit` function or dies because of a signal). **child_stack** points at a sufficiently large memory object, that can be used for stack of the new process. Bits of `flags` select what is to be shared. Symbolically:

- `CLONE_PARENT` – sharing PPID of the parent process
- `CLONE_FS` – sharing file system root, current directory, umask
- `CLONE_FILES` – sharing the file descriptor table
- `CLONE_SIGHAND` – sharing signal table
- `CLONE_VM` – sharing virtual memory

Additionally:

- `CLONE_VFORK` - stops the parent process, until the child terminates

`Clone` returns the thread id (TID) of the child process, or –1 (errno)

Note: `clone` is **not a POSIX function.**

Process creation/control/termination – cont.

`pid_t wait(int *pstatus)`

is waiting for a child process to terminate, returning its PID. The function returns –1 on error (eg. when no child, terminated nor active, exists). On success, when status!=NULL => status=**pstatus* stores information on reason of process termination. The information can be portably retrieved with the following macros:

`WIFEXITED``(status)`	1 when the child terminated with **exit**(), and 0 otherwise
`WEXITSTATUS``(status)`	argument of **exit**(), when **`WIFEXITED(status`**)
`WIFSIGNALED``(status)`	1, when child terminated due to signal delivery, 0 - otherwise
`WTERMSIG``(status)`	nr of the lethal signal (when **`WIFSIGNALED(status)`**)
`WIFSTOPPED``(status)`	1, when the child stopped, 0 - otherwise
`WSTOPSIG(status)`	nr of the stopping signal (when **`WIFSTOPPED(status)`**)

Process creation/control/termination – cont.

`pid_t waitpid(pid_t pid, int *pstatus, int options)`

is waiting for a sub-process specified with *pid*:

- **`pid==-1`** - any child process
- **`pid< -1`** – any child process, which is a member of the group nr **`|pid|`**
- **`pid==0`** – any child process, from the same group as the caller
- **`pid>0`** - the child process with **`PID==pid`**

Parameter *options* is a bit OR of 0, 1 or 2 constants:

`WNOHANG` – **`waitpid`** returns immediately, when no specified child process was found

`WUNTRACED` - **`waitpid`** returns also, when a child process was stopped..

`waitpid` returns PID of the child process (if found) or –1 on failure (the exact reason is specified with the global **`errno`** variable).

Process creation/control/termination – cont.

`void _exit(int status)`

a library function that cause immediate termination of a process by a call to a system function **`exit`**. Note: open files are closed without prior flushing associated buffers. Child processes are adopted by process **`init`**, and the parent process receives signal **`SIGCHLD`** (unless it declared to ignore it).

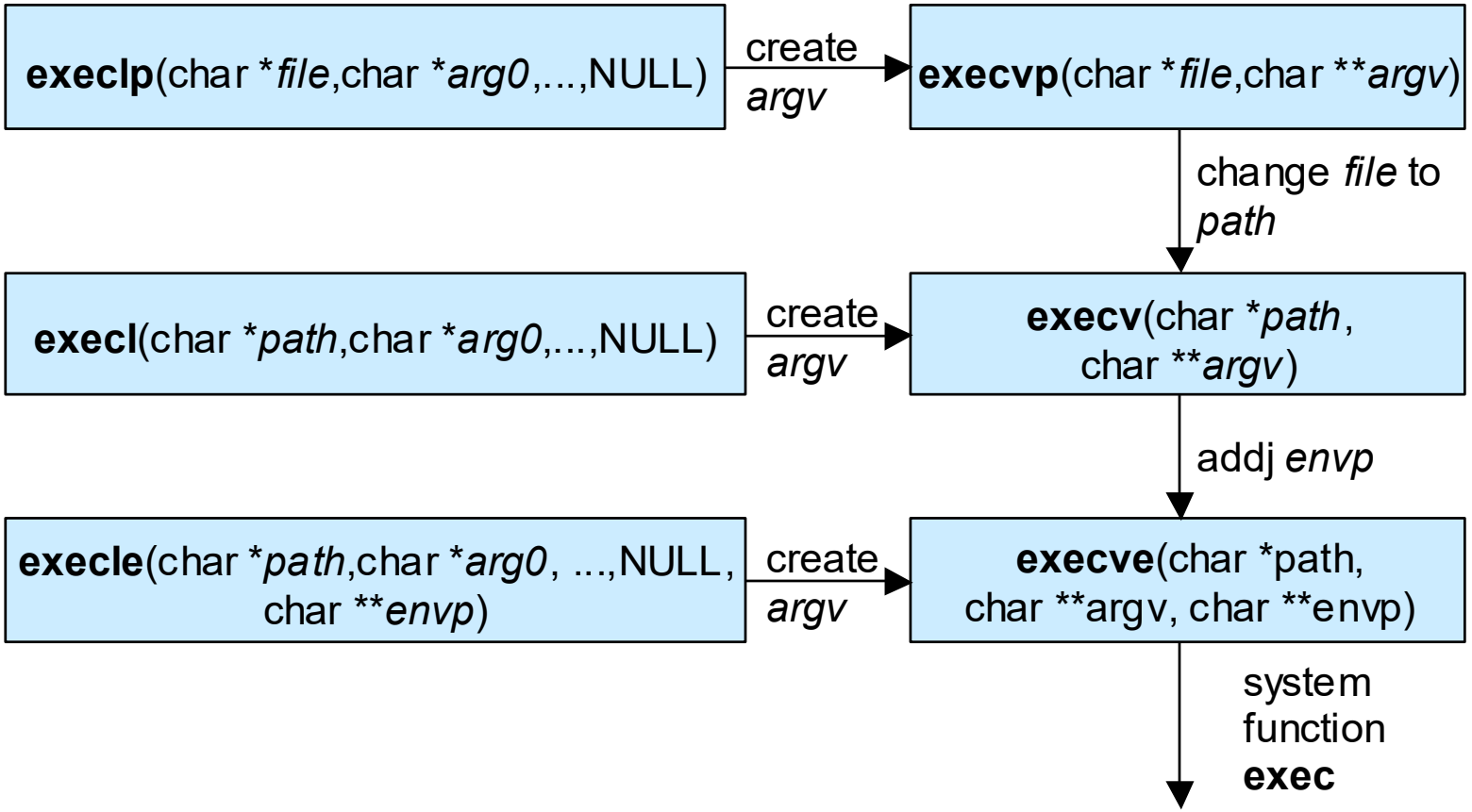
`void exit(int status)`

a library function that calls **`_exit`** after calling user functions previously registered with **`atexit`**() (or **`on_exit`**()), after flushing all I/O user buffers and closing temporary files (created with **`tmpfile`**()). If the terminated process is a leader of a terminal session, then each foreground process of this terminal session group is sent `SIGHUP` signal. Furthermore the terminal session loses its controlling terminal.

The value of **status** may be **0, EXIT_SUCCESS, EXIT_FAILURE**, or any other value, though only the least significant 8 bits (that is, **status & 0377**) shall be available to a waiting parent process.

Process creation/control/termination – cont.

Standard library functions that call the system function: **exec**



Process UID/GID

- **`fork()`** system function copies (*real, effective, saved*) UID and GID of the parent to the child process
- A process which operates with effective UID==0 can change with **`setuid()`** its real UID and with **`setgid()`** its real GID to any registered in the system
- **`exec`** function usually preserves (real, effective, saved) UID and GID of the calling process. However, if a **`setuid`** is set in the i-node of the executable program, then the **effective** and **saved UIDs** of the newly started process become equal to the UID of the file owner; the real user identifier is untouched. Similarly the **setgid** bit changes the effective and saved GIDs of the program. The new program can switch the effective UID/GID between **saved** set-user-ID and **real** UID values.
- **`setuid/setgid`** bits give „ordinary users” the same access to system objects as those available to owners of the executable programs (that have set these bits).
- Use of **`setuid`** i **`setgid`** bits is **discouraged**, because the mechanisms can degrade system security.

Process groups

- **Process group** – a collection of processes that permits the signaling of related processes. Each process is a member of a process group that is identified by a process group ID (leader PID). A newly created process joins the process group of its creator, but later can become leader of a new group.
- **Session** – a collection of process groups established for **job control** purposes (process suspend/resume, fg/bg, control of terminal use). Each process group is a member of a session. A newly created process joins the session of its creator. A process can alter its session membership.
- **Controlling Terminal** – a terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the **foreground process group** associated with the terminal.
- Processes in the foreground job of a controlling terminal have unrestricted access to that terminal; processes of the **background process group** do not
- **Job** - a set of processes, comprising a shell pipeline, and any processes descended from it, that are all in the same process group.

Login shell – traditional Unix view

- Process **`init`** creates a child process **`getty`** (or alike), which determines initial parameters of a terminal connection and starts waiting for a user login name.
- **login** proces reads user password, calculates its hash and compares it to the value stored in file **`/etc/shadow`** (or in another system location). Upon match the process sets new real UID and GID – according to **`/etc/passwd`**
- Finally **`exec`** function is called, starting run of a login shell – as defined in **`/etc/passwd`** file