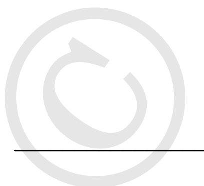
Student notes

Linux kernel internals

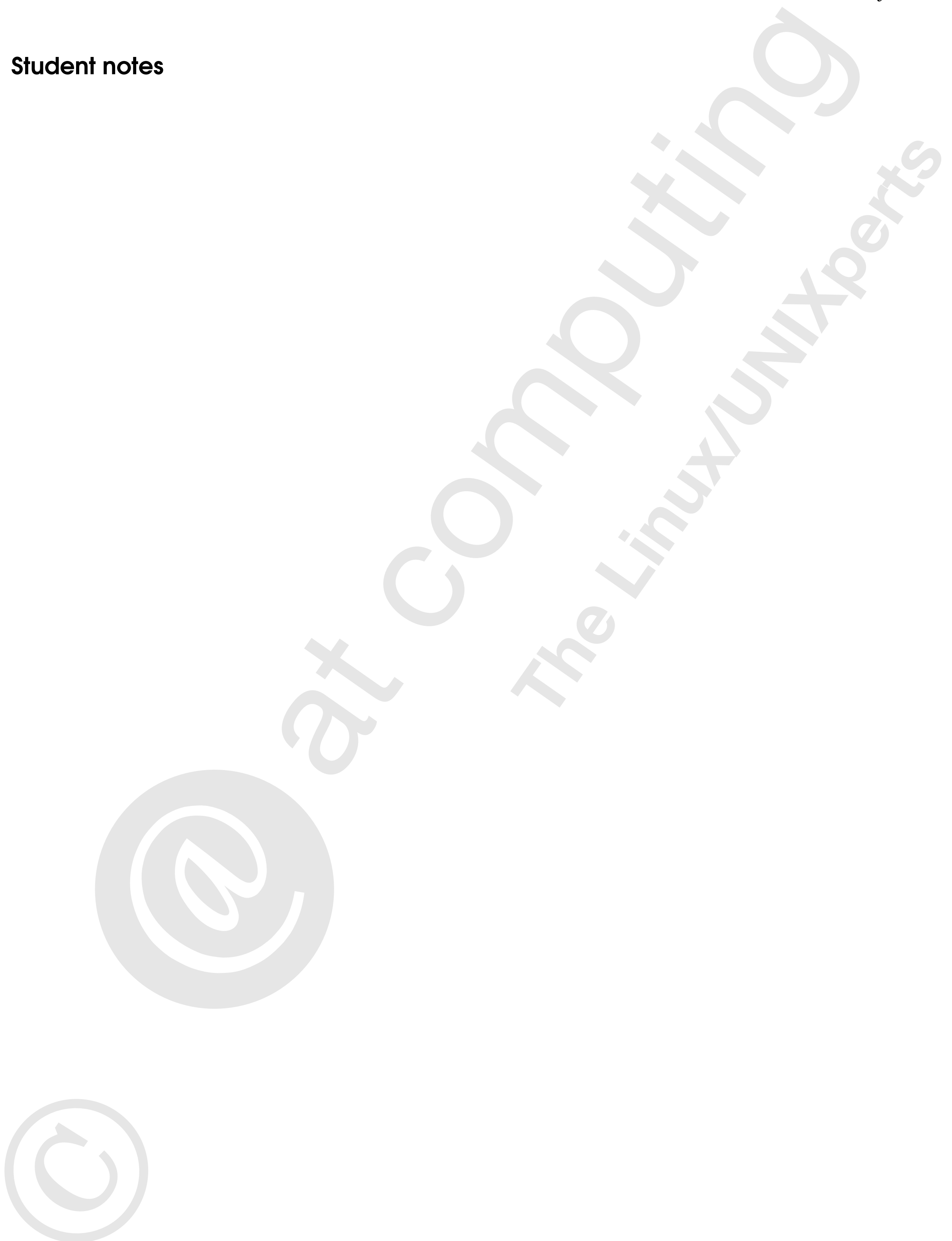
07. Process subsystem

Nijmegen, The Netherlands



Copyright © AT Computing 2002, 2005
Version: 2a

Student notes



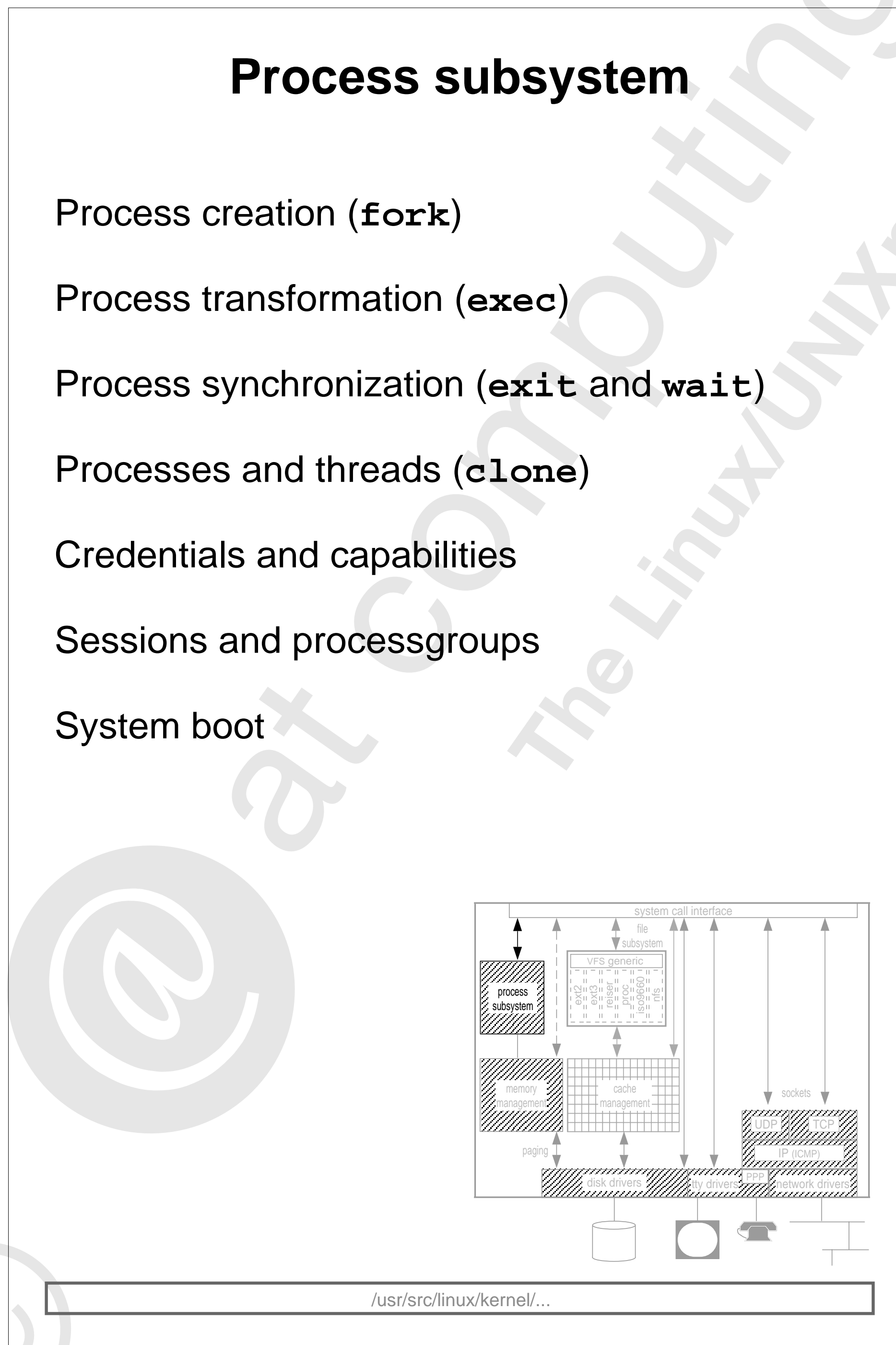


Figure 1.

Student notes

This slide shows the use of the four most important system calls for process management:

`fork` System call which duplicates the current process (from that moment on the *parent*) to create a new child-process. One process enters the system call and two processes return from this system call.

`exec` System call which enables a process to continue with another program (i.e. executable image). The current text, data and stack are removed; new text and data are loaded from the program file while a new stack is initialized.

`exit` System call to finish the entire process. As argument for this system call an exit-code has to be specified with convention: 0 means okay, 1-255 means something went wrong during execution.

`wait` System call used by a parent process to await a state-change by one of its children. State-changes reacted upon:

- Child stopped (i.e. temporarily suspended).
  The parent is notified via the return value of the `wait` that the child has entered the stopped state.
  Example: If a process is started by the shell, the shell enters the system call `wait`. If the user suspends the program by entering `^Z` (the tty-driver generates the stop-signal), the shell leaves the `wait` and gives a message that the process has stopped. By entering the command `fg`, the shell sends a continuation-signal to the stopped process and enters system call `wait` again.

  By the way: A similar scheme is used by a debugger (e.g. `gdb`). The child-process (program to be debugged) is forked by the debugger, and waits in stopped state. When a breakpoint is set in the code of the child, the debugger (parent) can pass control to the child-process and issue a system call `wait`. Once the child process hits the breakpoint, it enters the stopped state and the debugger is woken up from the `wait` again. For this scheme the system call `ptrace` is used.

- Child finished.
  Child has issued system call `exit` (own initiative) or received a deadly signal (from the outside world).
  Return values which are passed to the parent after issueing a `wait` are the PID of the finished child and the exit code; if the lowest byte of the exit code larger than zero, it reflects the deadly signal by which the process is finished. If the least significant byte is zero, the byte before the least significant byte contains the exit-code specified by the child itself.

© AT Computing The Linux/UNIX experts

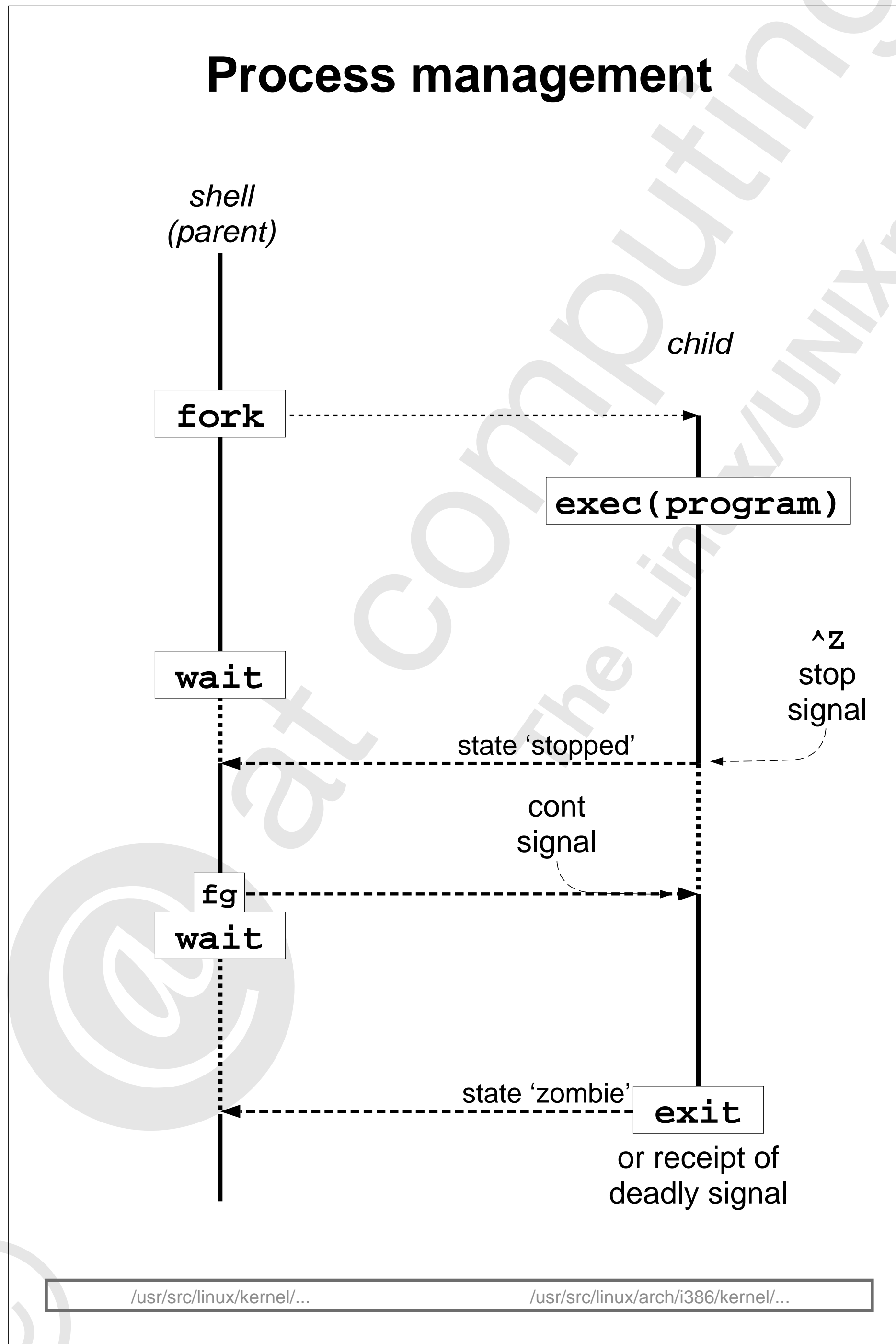


Figure 2.

Student notes

The slide shows that the system call `fork` copies the `task_struct` of the parent to the new-born child, including all related satellite struct's:

`signal_struct`	containing the shared signal information (shared pending signals) and other info like resource limits, session-id, processgroup-id, controlling tty, etc,
`sighand_struct`	containing the information how to handle signals (default, ignore or catch in subroutine),
`fs_struct`	containing the reference to the current directory, the root directory and the umask,
`mm_struct`	containing the description of the process' virtual memory area's, and
`files_struct`	containing the filedescriptor list (list of open files).

In this way the child inherits the parent's environment, such as open files, current directory, signal treatment definitions, etc.

Notice that the reference-counters of the struct `file` (for open files) and struct `dentry` (for current directory and root directory) will be incremented due to the new references.

© @ at computing The Linux/UNIXperts

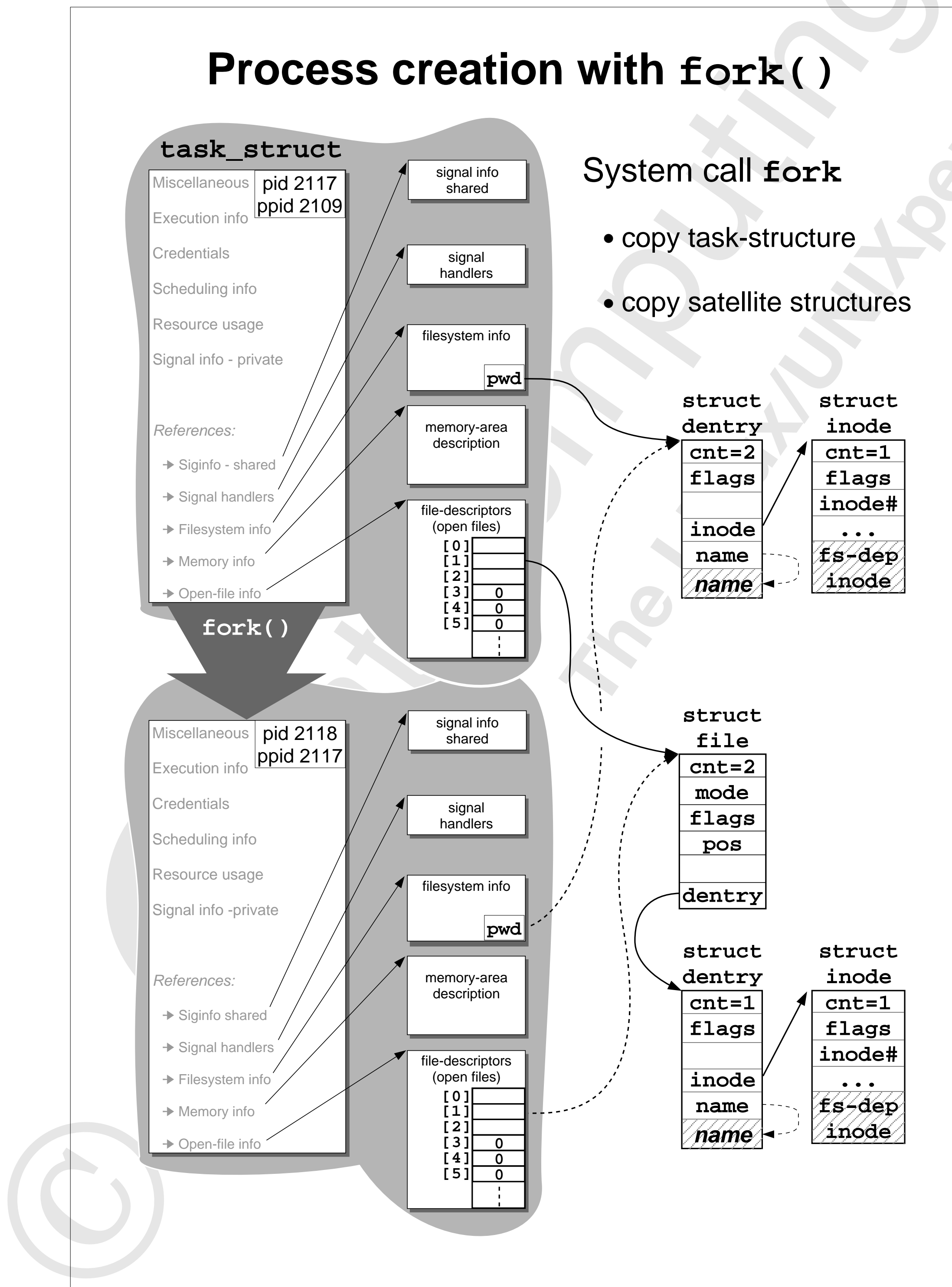


Figure 3.

Student notes

The kernel-subroutine do_fork in .../kernel/fork.c is used for the system call fork as well as for the system call clone (difference in step 8):

1. Copy parent task_struct to new child task_struct.
2. Check number of processes for this user (struct user_struct): if larger than rlim_cur (default: max_threads ÷ 2, also for rlim_max), the fork fails.
3. Increment number of processes for this user and increment refcount user.
4. Check if current number of processes in system (variable nr_threads) exceeds maximum allowed (variable max_threads). Variable max_threads is calculated as number of physical memory pages ÷ 8, so 12,5% of the memory may be filled with process administration (task_struct and satellite-structures consume approximately 1 page).
   Variable max_threads can be viewed and modified via /proc/sys/kernel/threads-max.
5. Increment refcount in inherited exec_domain (see explanation later on).
6. Determine PID for child process by incrementing variable last_pid. If this value exceeds the maximum allowed PID (default 32767 — view and modify via /proc/sys/kernel/pid_max), it is reset to 300 (margin not to clash with PID's of running daemons). Before the new PID is used, the kernel checks if the PID-value is still in use from a previous cycle (not only as PID, but also as PGID or SID).
7. Clear pending signals for child (should not be inherited from parent).
   Set process timers and statistical counters for child to zero.
   Set start-time of child to variable jiffies (clock-ticks since boot).
8. Copy the satellite structures for the child process.

   ■ The system call clone uses this algorithm as well. However in that case the satellite structures are shared with the parent (so not copied) on request with the flags CLONE_..... □

9. Copy thread (i.e. cpu-registers) and fill the child's eip-register with address ret_from_fork. The current algorithm continues in the parent's context; as soon as the child is scheduled later on, it proceeds at address ret_from_fork. This piece of code can be found in .../arch/i386/kernel/process.c.
10. Give half of the parent's current time-slice to the child.
11. Increment variables nr_threads and total_forks (process creations since boot).
12. Set state of child to running, add it to the runqueue and force a process-switch from the parent (that executed this algorithm) to the child. The child is now able to exec a new program, before memory-pages are copied for the child via the copy-on-write algorithm.
13. In parent's context: Return child-PID.

At address ret_from_fork (continuation-point for child) no special actions are issued (finish system call and return to user mode).

© AT Computing The Linux/UNIX Xperts

Process creation

System call **fork**:

- copy parent **task_struct** to child **task_struct**
- fail if #processes for this user exceeded (if not superuser)
- increment #processes for this user
- fail if #processes in system exceeds maximum
  nr_threads >= max_threads atkins> var
- determine new pid for child (**last_pid**) atkins> var
- clear pending signals for child
- reset statistical counters for child to zero
- copy **files_struct**
  copy **fs_struct**
  copy **signal_struct**
  copy **sighand_struct**
  copy **mm_struct**
- copy cpu-registers from parent to child;
  fill **eip** of child with other continuation address
- increment **nr_threads** and **total_forks** atkins> var
- set child's state to 'running', add to runqueue and
  force process-switch to child

/usr/src/linux/kernel/fork.c /usr/src/linux/kernel/pid.c /usr/src/linux/arch/i386/kernel/process.c

© at computing the Linux/unixperts

Figure 4.

Student notes

The dynamic linker/loader (`ld.so`, `ld-linux.so`) itself is a shared object which is loaded during the system call `exec` in the address space of the program to be activated. The kernel takes care that the loader-code is activated before the program's start-subroutine is activated.

The dynamic loader loads (via system call `mmap`) the other shared objects which are required by this program before passing control to the start-subroutine of the program (activities of the dynamic loader can be seen at the beginning of a system call trace with command `strace`). The file names of the shared objects (e.g. `libc.so.`*n* where *n* is the version number) will be searched in the following directories (in this order):

1. Directory name(s) found in `DT_RPATH` section of program file.
2. Directory name(s) found in shell environment `LD_LIBRARY_PATH` (except for setuid/setgid programs).
3. Directory name(s) found in `DT_RUNPATH` section of program file.
4. Directory name(s) found in `/etc/ld.so.cache`.
   This file can be generated with the program `ldconfig`. This program searches for shared objects in a number of directories and assembles the non-ASCII file `/etc/ld.so.cache`. The directories searched by `ldconfig` are `/lib`, `/usr/lib`, the directories found in the configuration-file `/etc/ld.so.conf` and directory names specified as command-arguments for `ldconfig`.
5. Default directories `/lib` and `/usr/lib`, unless compiler-option "-z nodeflib" had been used.

The dynamic loader can be controlled by certain environment-variables, a.o.:

- `LD_BIND_NOW` filled with any string: resolve all symbols at startup instead of with first reference.
- `LD_WARN` filled with any string: warn about unresolved symbols.
- `LD_PRELOAD` filled with list of (space-separated) shared object names: load these objects before other objects to overrule the particular subroutines; instead of this environment variable, such list of (space-separated) shared object names can also be defined in the file `/etc/ld.so.preload`.

See also man-pages of `ld.so`.

Some performance-measurements:

- The C-program "`main(){}`" is compiled statically (`cc -static`) and dynamically (just `cc`). Starting this program 10,000 times needs (20.7 - 1.4) 19.3 seconds for the static version and (39.5 - 1.4) 38.1 seconds for the dynamic version (the subtracted 1.4 seconds is the time needed for an *empty* shell-loop of 10,000 times).
- The C-program "`main(){return; printf(); scanf(); ...}`" is only compiled dynamically. Starting this program 10,000 times with the environment variable `LD_BIND_NOW` requires 25% more time than without this variable.

© AT Computing The Linux/UNIX experts

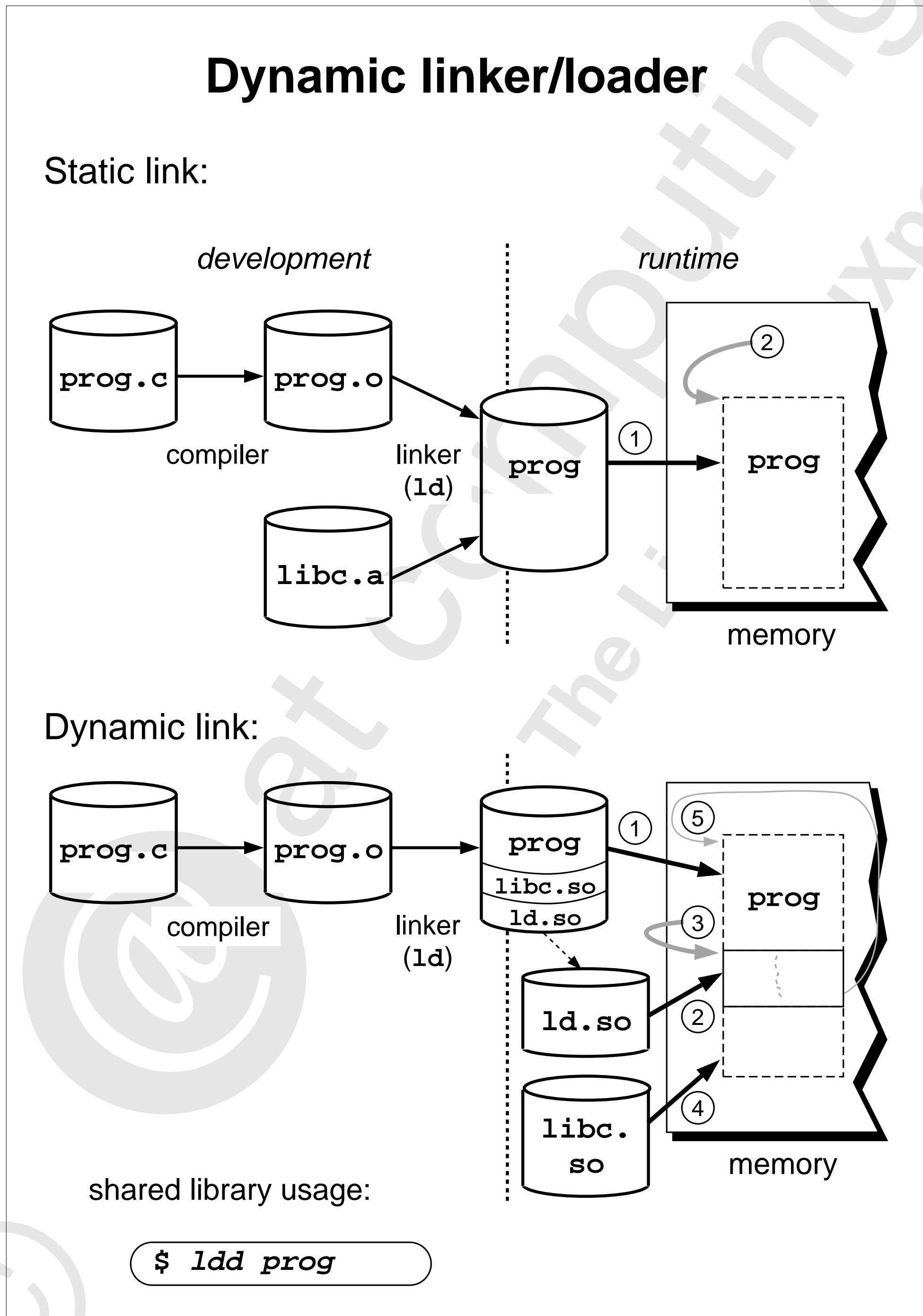


Figure 5.

© @ at Computing The LinuXperts

Student notes

The kernel-subroutine do_execve in .../fs/exec.c is used for the system call exec; the system call arguments program-file name, argv[] and env[] are the calling-arguments for this subroutine:

1. Open the program file.
2. Count the number of entries in argv[] and envp[].
3. Check the mode of the program file: is it executable, etc.
   If suid-bit set, fill euid of process with uid of file.
   If sgid-bit set, fill egid of process with gid of file.
4. Read first 128 bytes of program file and store in temporary buffer.
5. Allocate memory, and copy the argument strings and environment strings from user space to kernel space. Once the current user space is removed and created again, these strings are copied to the new stack (to be passed as argument for the main subroutine).
   The maximum amount of kernel memory reserved for this purpose is 32 pages (defined value MAX_ARG_PAGES), i.e. 128 Kbytes for Intel-32bits. When this is not sufficient, the exec will fail and the shell reports “arg list too long” (try: “cat /usr/*/*/*”).
6. This is the end of the generic part of exec.
   Linux can handle several types of program file formats (e.g. ELF, A.OUT, etc). For every file format, a set of subroutines is required to handle that type of program file (loadable module or statically linked in kernel); this set of subroutines is registered in an entry of the formats-list.

   At this point the subroutine load_binary of every entry in the formats-list is called (passing the temporary buffer with the first 128 bytes from the executable file — see step 4) until one of them recognizes the format.
   E.g. the load_binary of the ELF-set tries to recognize the ELF magic number in the temporary buffer. If not recognized, it returns a failure and the next load_binary (e.g. the one of A.OUT) is called to verify if the magic number is recognized.
   If no load_binary subroutine has recognized this program file, the system call exec fails.

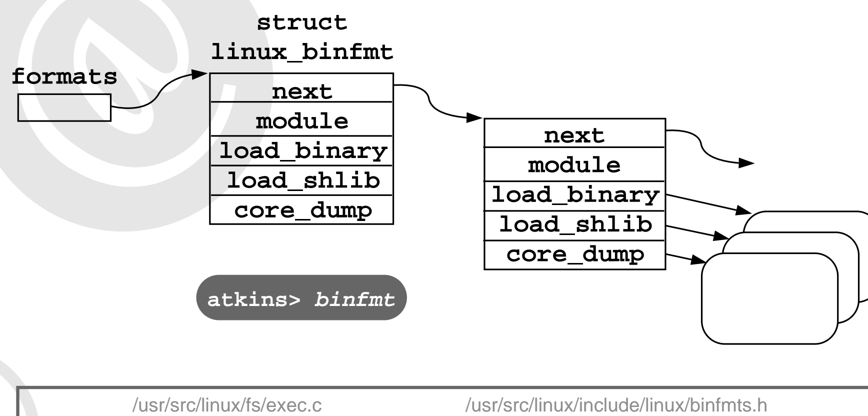
   The load_binary subroutine that recognizes the format of the program file, takes care of further handling of the system call exec. On the next slide an example of the load_binary subroutine for ELF can be found.

at computing the Linux/UNIXperts

Process transformation

System call **exec** (generic part):

- open program file
- determine size of arguments and environment variables in current program image
- check mode of program file
  - if **suid**-bit set, fill process' **euid** with **uid** of file
  - if **sgid**-bit set, fill process' **egid** with **gid** of file
- read 128 bytes from program file into buffer
- allocate kernel buffer (maximum 32 memory pages) copy arguments and environment variables to this buffer
- call function **load_binary** of all format-handlers to recognize type of program file (ELF, A.OUT, script, ...) and issue further treatment



/usr/src/linux/fs/exec.c /usr/src/linux/include/linux/binfmts.h

© at computing The Linux/UNIX experts

Figure 6.

Student notes

The `load_binary` subroutine for `ELF`-files (subroutine `load_elf_binary` in the file `.../fs/binfmt_elf.c`):

1. Check magic number to verify it is `ELF`-format.
2. Read `ELF`-headers from program file.
3. Check if a section named "`.interp`" is present, containing the name of an "interpreter" (=runtime loader). This section is present for dynamically linked programs. If present:
   - read loader path-name (e.g. `/lib/ld-linux.so.2` can be a symlink to `/lib/ld-2.2.2.so`),
   - open the loader,
   - read 128 bytes in memory buffer,
   - check magic number of loader.
4. Call to generic subroutine `flush_old_exec` (in `.../fs/exec.c`):
   Remove other threads (if any) from threadgroup; current thread continues as threadgroup leader.
   Cleanup `task_struct` by removing stuff related to old program file (command-name, cpu-registers, reset signal-catcher addresses to "default", close files with option "close-on-exec").
   Set `rss` in `task_struct` to zero.
5. Remove current virtual memory (text, data, stack of old program).
6. Setup of virtual memory for new program file.
7. Copy saved argument and environment strings to new stack.
8. If loader present (see step 3), load it in new address space and fill `eip`-register with entry-point address of loader.
   If loader not present, fill `eip`-register with entry-point address of program file itself.
   Continue at next instruction (`eip`-register) .....

Another example is the `load_binary` subroutine for shell-scripts (subroutine `load_script` in the file `.../fs/binfmt_script.c`):

1. Check if the first two bytes of the file contain the magic number "`#!`" (hash-bang).
   If that is the case, the rest of the line should contain the path-name of a program file (e.g. `/bin/bash`), optionally followed by one argument.
2. Remove superfluous spaces, tabs and line-feed characters from the first line.
3. Repeat steps 1 till 4 of the subroutine `do_exec` (previous notes page) for the program file found in the first line of the script.
4. Repeat step 6 of the subroutine `do_exec` (previous notes page) for the new program file.

© AT Computing – The Linux/UNIXperts

Process transformation

System call **exec** (ELF specific **load_binary**):

- check magic number to verify ELF format
- read section headers from program file
- check existance of section **.interp** (name of loader)
  - read pathname of dynamic loader
  - open loader file
  - read 128 bytes to verify magic number of loader
- reset signal catcher addresses to ‘default’
- close files with option ‘close-on-exec’
- remove virtual memory of old program
- setup virtual memory for new program
- copy arguments and environment from kernel buffer to new stack
- if dynamic loader is present,
  load and activate dynamic loader at its entry-point
  else
  activate program itself at its entry-point

/usr/src/linux/fs/binfmt_elf.c

© at computing
The Linux/UNIXperts

Figure 7.

Student notes

Kernel-subroutine do_exit in .../kernel/exit.c handles system call exit:

1. Check if current process has PID 0 or 1 → panic.
2. Set process flag EXITING and stop process timers.
3. Write accounting record (if process accounting active).
4. Remove current virtual memory (text, data, stack, ...)
5. Remove satellite structures, except the ones related to signals.
   Close all files from fd_array.
   Close current directory and root directory.
6. If process is session leader, disconnect relation with controlling tty.
7. Store exit-code (8 bits left-shifted) in task_struct.
8. If process has child-processes, search a new parent for those children (i.e. process-id 1 is set as PPID for child-processes).
9. If process is process group leader (meaning the process group will now be "orphaned"), send signal combination SIGHUP/SIGCONT to stopped processes.
10. Generate signal SIGCHLD to notify parent (possibly waking up parent).
11. Remove all threadgroup-members (if any).
12. Set process state to ZOMBIE.
13. Call the scheduler (this process is dead now).

Kernel-subroutine sys_wait4 in .../kernel/exit.c handles system call wait:

1. Check all child processes of current process (via separate chain):
   a. Child in state STOPPED?
      - Resume parent (finish system call wait) indicating the presence of a stopped child.
   b. Child in state ZOMBIE?
      - Register cpu-usage of child and child's children in task_struct of parent (can be requested with system call times).
      - Decrement number of processes for this user.
        Decrement number of processes in the system.
        Remove satellite structures related to signals.
      - Remove task_struct of child.
      - Resume parent (finish system call wait) indicating the presence of a exited child.
2. No stopped or zombie child!
   If option WNOHANG specified for system call wait, resume parent.
   If signal is pending, resume parent.
3. Parent continues waiting (just calls the scheduler).

© AT Computing The Linux/UNIXperts

Process synchronization

System call **exit** (also for deadly signal):

- write accounting record (if mechanism activated)
- remove virtual memory
- remove satellite structures (incl. close all files), except shared signal info and signal handlers
- store exit-code in **task_struct**
- assign all child processes to pid 1 (**init**)
- send signal SIGCHLD to parent
- set process state 'zombie' and call scheduler....

System call **wait**:

- if any child process in state 'zombie'
  - register cpu-usage of child and its descendants
  - decrement #processes for this user
    decrement #processes in system (**nr_threads**)
    remove remaining satellite structures
    remove **task_struct**
  - resume parent (finish system call **wait**)
- if parent ignores SIGCHLD, actions automatically done

/usr/src/linux/kernel/exit.c

© at computing The Linux/UNIXperts

Figure 8.

Student notes

The idea behind multithreading is that several execution-threads can be active in parallel (e.g. simultaneously on more processors) using a shared environment (resources). That means that the active part is multiplied, all using the same passive part. In general, a thread is defined as an independent stream of instructions that can be scheduled by the operating system.

In the context of traditional UNIX flavors, a thread exists within a process and shares the process' resources with other threads (related to that process).

In the context of Linux, a thread has been implemented as a new "process" (`task_structure`) that is able to share parts of the "process"-administration (i.e. the satellite structures) with other threads (implemented as "processes").
All related threads are bundled within a *thread-group*, that has one leader (the original main thread), and zero or more members.

The system call `__clone` can be used to make a new thread, sharing particular satellite-structures with other threads. This system call uses the same algorithm as the system call `fork`, but has the possibility to define a selection of satellite structures to be shared.

Signal handling

In a multithreaded environment, certain signals are sent to the thread-group. These signals are registered in the *shared list* of pending signals. Preferably the kernel activates the main thread (i.e. threadgroup leader) to handle such signal. If that is not possible (e.g. because the main thread has blocked the signal), another thread from the thread-group is chosen to handle the shared signal.
Signals sent by the command `kill` (using system call `kill`) are always meant for the thread-*group*.

Signals can also be sent to a *specific* thread. These signals are registered in the *private list* of pending signals and can only be handled by that specific thread.

A fatal signal will destroy the entire thread-group.

Pthreads

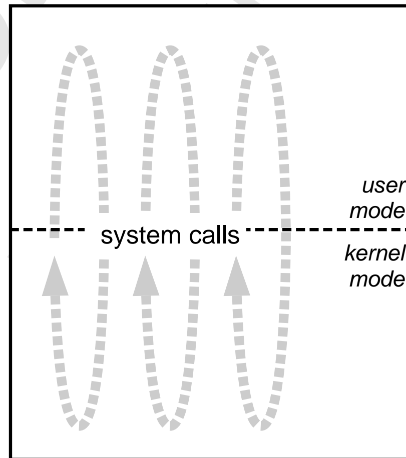
Portable multithreaded applications can be developed using the *Pthreads*-library (a POSIX-compliant API). The subroutine `pthread_create` uses the system call `clone` (it specifies the flags `CLONE_THREAD`, `CLONE_SIGHAND`, `CLONE_FS`, `CLONE_VM`, and `CLONE_FILES` — see next notes page).

© AT Computing – The Linux/UNIX experts

Multithreading

Program with several execution threads (active parts) sharing same environment

- threads share
  - pid, ppid, ...
  - static data area
  - set of open files
  - filesystem info (e.g. current directory)
  - signal definition
  - ......
- but have their private
  - (saved) cpu-registers
  - stack
  - state
  - .....



Threads provide ability to perform several tasks simultaneously (on several CPU's) using same resources

Thread creation with system call **`clone`**

- new task-structure which (optionally) shares satellite structures

Figure 9.

© at computing x/UNIXperts

Student notes

The system call __clone expects the following arguments:

1. Address of start subroutine
   The new thread will proceed at this point. The C library subroutine __clone issues the system call clone which copies the task_struct; after returning from this system call, the C library subroutine __clone calls the specified start subroutine (i.e. the kernel is not aware of that).

2. Address of new stack
   Before the system call clone is issued, the calling process should allocate stack space (e.g. via malloc). Since stacks grow from high to low addresses, the *end address* of the allocated space should be passed.

3. Flags
   In the flags field a signal number is specified, to be sent to the "parent" as soon as the new clone dies (e.g. signal SIGCHLD).
   Furthermore a number of flags can be OR-ed to request sharing of satellite structures:

   CLONE_PARENT
   : The parent-id of the new thread is copied from the parent-id of the current thread (so the new thread fakes it is a descendent of the same parent).

   CLONE_THREAD
   : Share the shared signal information (resource limits, session, processgroup, controlling tty, pending signals). Furthermore the TGID (threadgroup-id) of the new thread is copied from the current thread (new thread becomes member of threadgroup of calling thread) and a reference is made to the task_struct of the leader of the threadgroup.
   Notice that a new thread created via the system call clone gets a new unique PID. According to the definition that threads should be able to share the *same* PID, the system call getpid always returns the TGID, not the PID! The system call getppid always returns the TGID of the parent of the threadgroup-leader.

   CLONE_SIGHAND
   : Share the structure describing the signal handlers.

   CLONE_FS
   : Share the structure describing the filesystem info.

   CLONE_VM
   : Share the structure describing the virtual memory area's.

   CLONE_FILES
   : Share the structure describing the open files.

4. Argument for start subroutine
   One argument (void *) can be passed for the start subroutine called in the new thread.

The "parent" gets the PID of the new clone as return-value (or -1 in case of failure).

© AT Computing The Linux/UNIXperts

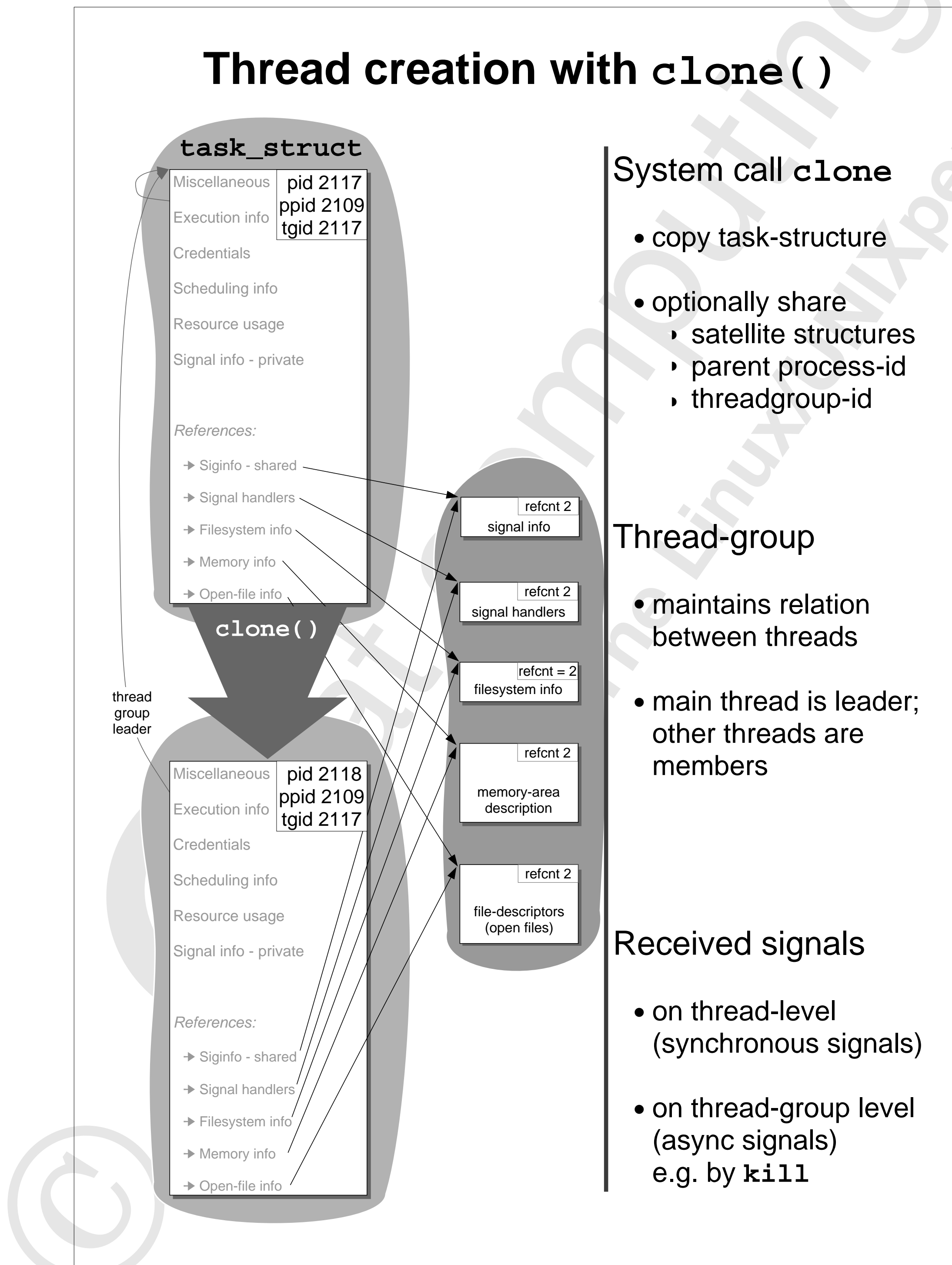


Figure 10.

Student notes

The remaining part of this chapter covers the subject of capabilities, sessions and process groups, and system boot.

© at computing The Linux/UNIXperts