opiframe
Knowledge equals power

Opiframe Oy

Linux Process Model Basics

Content

- Process model basics. More on this later and also in kernel (task_structs and such)
- Daemons
- Syslogger

The process model

opiframe
Knowledge equals power

- Each time a program is started the OS will create a new process and load the image of the program into this process.
- After that the process is ready to be scheduled and it will start.
- Of course things are more complicated but this serves as an approximation.

The process model

- You could argue that process is just an instance of a program.
- There might be multiple instances of the same program each running independently even by different users.
- Still you only have one program file on your hard disk.

The process model

- When there are processes ready to be run in the system, bits of them will be run consecutively by the part of the kernel called scheduler.
- Scheduler is invoked via the hardware clock mechanism with intervals of 10 milliseconds.
- This interval is called time slice or scheduling quantum. It provides basic function for multi-tasking.

The process model

- When the scheduler is run it will check through it's list of current processes.
- It selects the processes to be run based on importance attached to these processes.
- In linux processes are given an integer between -20 and 20 where -20 is given to most important process.

The process model

- After the program instance ends it self or is terminated by the user, the kernel will mark that process as terminated and scheduler will not consider this process for running again.

- The process is not removed from the list of running processes because the parent process might be interested in the results of this process.

The process model

- Process is also a data structure inside the kernel memory area describing various aspects of the program instance.

- Things like open file descriptors, currently allocated memory, shared memory maps, processor state, current working directory and loads of other stuff.

The process model

- There is also information about the owners of the process.
- Each process in linux has:
- A process id (PID)
- A parent process id (PPID)
- An user id (UID)
- An effective user id (EUID)
- and similar ids for groups.

The process model

- When an user starts a process the process will inherit the UID and the GID values from the process that the user started the program from.
- EUID and EGID usually match UID and GID.
- EUID and EGID are checked by the kernel against operation credentials when process tries to do something privileged.

The process model

- Processes also work as a security boundary.
- This is because process can only access memory areas that are assiociated with it
- It cannot access any devices directly (including I/O) and it cannot access files or network resources directly.

The process model

- Once the process wants to do anything listed above it needs to ask the kernel to do something for it.
- Meaning of course that the kernel services the processes.
- The possibilites are limitless but the mechanism for the processes to do this is called system calls.

The process model

opiframe
Knowledge equals power

- In Linux processes are not created out of thin air but rather a system call **fork()** .
- When a process issues a **fork()** the operating system will create a new process ID (pid).
- It will also create a new entry in the kernel process table.

The process model

- The new process will contain the same binary image as the original process.
- We have just created an exact copy of the running process except for,
- The new process has an unique PID and the parent PID is set to the PID of original process.

The process model

- Also things like pending signals (signals come later on) and file locks are not inherited.

- There is a final major difference. The value that the **fork()** returns to the caller:
- In the original process, it will return the PID of the process created
- In the new process, it will return 0.
- This is the ONLY way to distinguish between two processes after **fork()** in the same code.

The process model

- Modern POSIXes handle forking by marking the affected memory areas as ”copy-on-write”.
- It means that the new process will be allocated unique memory ONLY when it will write something into memory.
- When the new process writes something only those pages will be copied that are written on-demand.

The process model

- When a process forks, the new will be the original one's child process.
- The original process is the parent process of the child.
- This is true for all cases of **fork().**

The process model

- If the parent process terminates before the child process does, the kernel will move the child process under the init process (PID 1).
- In every situation, all normal processes have parents.
- A child with parent process terminated is called orphan process. Orphans are claimed by init process.

The process model

- Forking is the basic tool in your system programming arsenal.
- Another part of the new process creation is the a family of system calls called **exec()**.
- The whole process creation can be called fork-exec.

The process model

- **exec()** tells the kernel to load the program on top of the current code memory.
- When the code contained in the binary is loaded on top of the code memory area, execution will be reset to the start of the new binary.
- After some linking, the **main()** of the new binary is executed.

The process model

- All open files and other process state is preserved.
- If you have not closed stdin/stdout, the same ones will be used by the new program.
- The memory heap will not be zeroed or any other memory areas for that matter.

The process model

- The new binary code normally contains sections of data or code that will be loaded on top of the old code, so normally this is not a problem.

- However, if you have sensitive data in dynamically allocated memory, you should always zero it out before execing.

The process model

- The best way to protect yourself is to close all unnecessary files, free all allocated memory and use **memset()** to zero out the heap.

- Also pointing your variables and pointers to NULL is a good idea after forking but before execing.

The process model

opiframe
Knowledge equals power

- The different **exec()** family system calls are:
- **int execl(const char *_path_, const char *_arg_, ...);**
  int execlp(const char *_file_, const char *_arg_, ...);
  int execle(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_[]);

- The initial argument for these functions is the pathname of a file which is to be executed.

The process model

opiframe
Knowledge equals power

- 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 process model

- The first argument, by convention, should point to the filename associated with the file being executed.

- The list of arguments *must* be terminated by a NULL pointer.

The process model

- The **execv**() and **execvp**() functions 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 filename associated with the file being executed.
- The array of pointers *must* be terminated by a NULL pointer.

The process model

- The **execle**() function also 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.

- The functions **execlp**() and **execvp**() will duplicate the actions of the shell in searching for an executable file. In short they use PATH variable.

The process model

opiframe
Knowledge equals power

- These all are wrappers to the actual system call **execve().**
- **int execve(const char **filename*, char *const** *argv*[],
  char *const *envp*[]);
- **execve**() executes the program pointed to by *filename*.

The process model

- *argv* is an array of argument strings passed to the new program. *envp* is an array of strings, conventionally of the form **key=value**, which are passed as environment to the new program.

- Both *argv* and *envp* must be terminated by a null pointer.

The process model

- Things to note about **exec()** system calls:
- **exec()** doesn't change PID.
- pending signals are cleared.
- signal handlers are reset to default action.
- the suid bit is set if it exists for the binary.
- **exec()** does not return to the caller unless an error occured.

- Root privileges and **exec()** can be a potential security hazard.

The process model

- If you want to start a binary without running it on top of the current code you can use **system()**.

- int **system**(const char *command);

- **system**() executes a command specified in *command* by calling **/bin/sh -c** *command*, and returns after the command has been completed.

The process model

- Now that we have both fork() and exec() in our arsenal we are pretty much set for other program executing.
- After all that forking and execing, how does the original process know when the child process has finished.
- This is the territory of a system call called **wait()**.

The process model

- System call **wait()** is actually just a wrapper for **waitpid()**. They are prototyped as:
- **pid_t wait(int ****status***);**
  pid_t waitpid(pid_t ***pid*****, int ****status***, int** ***options*****);**

- These system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.

The process model

- A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

- If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call.

The process model

- In **waitpid()** the first argument can be:
- **< -1** meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.
- **-1** meaning wait for any child process.
- **0** meaning wait for any child process whose process group ID is equal to that of the calling process.
- **> 0** meaning wait for the child whose process ID is equal to the value of *pid*.

The process model

- Once a child process terminates, the kernel will free up all the resources of this child.
- The return code and exit status needs to be communicated to the parent process incase it is interested.
- The kernel cannot know for certain whether this is the case

The process model

- Just to be on a safe side, the kernel will still leave the the process table allocation for the child alive.

- Only when the parent will call **wait()**, the kernel will be free to de-allocate the process table entry.

The process model

- A process that has been terminated, but not been waited on, is called a zombie.
- Since every zombie takes one entry in the kernel process table reaping the children is quite important.
- The scheduling decision time will be affected by the number of processes in the process table

The process model - limits

opiframe
Knowledge equals power

- Older UNIX systems one process could easily allocate all memory and use all available processor time.

- A mechanism was constructed to stop this. This is called **resource limits**.

The process model - limits

- Resources are limited as follows:
- Kernel limits
- Hard limits set by admin
- Startup scripts or shell init scripts with soft limits
- **ulimit** command for users
- All processes starting from a given shell start with those limits.
- Processes may lower the limits themself.
- Processes can increase the limits up to hard limits.

The process model - limits

- The resources that can be limited (an incomplete list):
- Maximum virtual memory
- Maximum CPU time in seconds
- Maximum file size
- Maximum number of file locks
- Maximum number of child processes that can be created
- Maximum size of the memory stack

The process model - limits

- Resources can be handled with system calls
- **#include <sys/resource.h>**
- **int getrlimit(int** *resource*, **struct rlimit ****rlim*);
  int setrlimit(int *resource*, **const struct rlimit ****rlim*);

- **Struct rlimit** is defined in resource.h and has both soft limit and hard limit members.

The process model – Access control

- Access rights are inherited from parent process.
- Process may set a filesystem bitmask that will be applied the process creates new files.
- This mask (umask) will be inherited by child processes.

The process model – Access control

- The access rights mechanism is applied on all file system accesses for a process.
- The same access rights mechanism is also applied across other mechanisms.
- A list of access control bits which defines what can be done is called access control list or ACL.

The process model – Access control

- Remember the maxim that everything is a file that is not a process.
- This means that the access rights mechanisms are the same for everything else also.
- This simplicity makes security easier in linux than in many other systems.

Process capabilities

- Instead of allowing process EUID=0 (root permissions) we find out what are the actions that require privilege and list them.
- We don't give process full access instead we give limited privilege to do specific action.
- We also want to control the inheritance of these capabilities to child processes.

Process capabilities

- These capabilities come as a set.

- Each process has three fields for each capability.
- E-bit or effective capability. This is the bit checked when action is tried.
- P-bit tells the kernel which capability process may opt to turn off.
- I-bit tells which capabilities will be inherited by a child process.

Process capabilities

opiframe
Knowledge equals power

- An example list of possible capabilities:
- **CAP_CHOWN** Allow arbitrary changes to file UIDs and GIDs
- **CAP_NET_ADMIN** Allow various network-related operations
- **CAP_NET_BIND_SERVICE** Allow binding to Internet domain reserved socket ports (port numbers less than 1024).
- **CAP_SYS_CHROOT** Permit calls to **chroot**().
- **CAP_KILL** Bypass permission checks for sending signals

- As default all processes in Linux have all these capabilities turned off except for processes that run as root.

System Calls

- System calls are the mechanism for processes to actually get something done.
- The concept of having a strongly limited ”gateway” into the kernel services has been used extensively in many operating systems.
- This is a common feature in systems with monolithic kernels.

System Calls

- The processor is fetching instructions from some physical memory locations and executing them one at a time.
- Instructions are split into data-transfer, data-modifying and control transfer instructions.
- When the processor will execute data-transferring instructions the CPU will check whether the address is allowed currently.

System Calls

- We run the user processes in a special processor mode also known as processor context.
- This mode context contains limits suited for restricted memory access.
- This means that the CPU may execute instructions that only access some limited subset of memory.

System Calls

- If the process wants to use any real services it issues a request to a kernel service.
- Kernel then waives the memory limitations for a moment.
- After completing the task the CPU restores these limitations.

System Calls – context switching

opiframe
Knowledge equals power

- The way that linux implements the context switches and the user / kernel mode split is through software interrupt gates.
- This is a special form of CPU instruction that will:
- Load a new set of limits instead of current ones
- Remember which program location the program must continue after the interrupt
- Store other state about the CPU that needs to be preserved
- Instruct the CPU to load instruction from some pre-defined location in memory.

System Calls – context switching

- Linux has only one software interrupt gate (known as interrupt number 128(0x80)).
- The gate points to a code known as system call handler.
- It is a piece of code written in assembly in the architecture specific directories of the kernel source.

The process model - session

- Whenever you log into an UNIX system a structure called session is created within the kernel.

- Session holds the information about the controlling tty-device and other state information.

The process model - session

- When you logout the process known as the session leader will be asked to shutdown by the login system.
- The process that creates a session will become session leader automatically.
- The session leader will then stop any processes within its own session and finally exit itself.

The process model - session

- Each process within the same session is connected to same controlling tty-device if they have one.
- Processes cannot move between sessions but they can create new ones.
- You can use **setsid()** system call to do this.
- **pid_t setsid(void);**

The process model – process group

- Within a session processes are grouped together in process groups.
- Each process within one group is related to the group.
- Process groups are used for job control meaning that if you terminate the group leader process all other group processes terminate also.

The process model – process group

- An example of a process group would be a piped command like **ls -la | grep Linux**.

- Here **ls -la** would be the process group leader and if you terminate that by CTRL+C the **grep** process will also terminate.

Daemons

- A daemon is a process that is disconnected from the ordinary input and output flow.

- It also has been severed from its parents and grandparents, terminal and session in order for daemon process to remain active after user has logged out.

Daemons

- In order for a daemon to remain active after user has logged out it must obviously disconnect itself from the login session created by the user at login.

- It also must disconnect itself from the controlling terminal of the login process.

Daemons

- The way to create daemons is:
- **fork()** and then **exit()** parent. This will close the controlling tty-device.
- create a new session (**setsid()**). This will avoid the process being killed by the session leader since it will become one.
- change current working directory to something safe.
- reset the umask. This makes the file creation ACL bits controllable.
- Close all standard file descriptors (**stdin**, **stdout**, **stderr**)

Syslogger

opiframe
Knowledge equals power

- There is a daemon process called **syslogd** that is running in all Linux systems.
- It reads a well known file descriptor for specially formatted data sequences.
- It then adds timestamps and other information and forwards this information to a pre-determined log file.

Syslogger

opiframe
Knowledge equals power

- To use this system logger first you need to open an access to the logging system.

- You want to select the **facility** in which your messages will be logged to (others exist):
- LOG_AUTH, authority log
- LOG_DAEMON, daemon log
- LOG_MAIL, mail log
- LOG_USER, user log

Syslogger

- After having opened this access to syslogd you can send it messages by using the **syslog()** function.
- It accepts a priority number which further helps the system to determine what to do with your logging message.
- Priorities range from LOG_EMERG to LOG_DEBUG. You should use LOG_ERR, LOG_WARNING or LOG_NOTICE.