Operating Systems

Lecture 2 - Process and Thread

Adrien Krähenbühl

Master of Computer Science
PUF - Hồ Chí Minh

2016/2017

Processes

Exceptions & interruptions

System calls

Threads

Threads and processes

Presentation

Definition

A process is:

Definition

A process is:

- ✓ A program in execution

Definition

A process is:

- ✓ A program in execution
- ✓ An instance of a program running on a computer

Definition

A process is:

- ✓ A program in execution
- ✓ An instance of a program running on a computer
- ✓ An entity able to be assigned and executed on a processor

Definition

A process is:

- ✓ A program in execution
- ✓ An instance of a program running on a computer
- ✓ An entity able to be assigned and executed on a processor
- ✓ Activity unit characterized by the execution of an instruction set, a current state and a set of system resources

Objectives of a process

- ✓ All applications seems progress simultaneously
  - Physical processors execute it alternatively
  - Each application thinks to be alone on the computer
- ✓ Share available resources between multiple applications
- ✓ Use processor and devices efficiently

Characterization

Initialization
- ✓ Program code
- ✓ Set of input data

Execution
- ✓ Control block with a PID (Process IDentifier)

Control block

The control block of a process consists in:

- ✓ one **unique** PID
- ✓ a memory address (location of user data, program, stack, ...)
- ✓ a Program Counter (PC)
- ✓ a scheduling state (Waiting, Running, Blocked, Terminated, ...) with a priority
- ✓ the state of I/Os
- ✓ a privilege level
- ✓ statistics

Genealogy

- ✓ A process has a **unique** parent and can have sons
- ✓ The parent is notified of the death of its sons (and can wait until this moment)
- ✓ Orphan case: they need a parent (in Linux, Init will do this)

Process execution

A lot of architectures

- ✓ x86/x86_64
- ✓ IA64
- ✓ Alpha
- ✓ PowerPC
- ✓ Sparc
- ✓ ARM
- ✓ MIPS

What is the most sold?

Execution in processors

- ✓ One instruction per cycle
  - Cycle duration fixed, depends on the frequency
- ✓ Regular interruptions by clock
  - About 1 time/second in Linux
- ✓ Several execution modes with different privileges

Processor state and context

User registers
- ✓ Data
- ✓ Addresses of segment, stack,
- ...

Status and control counters
- ✓ Program Counter (PC)
- ✓ Arithmetic status

Execution modes

Several privilege modes:

- ✓ Set up physically by the processor
- ✓ Access to control registers
- ✓ Low-level instructions for I/Os
- ✓ Memory management
  - Example: access to CR3 for the page table on x86 processors
- ✓ Access to some memory areas
  - Restricted by structures pointed some registers
- ✓ Reduced instruction set depending on the privilege

Execution modes

The kernel can do everything:

- ✓ Protection Ring 0 on x86
- ✓ Except in case of hardware virtualization

The user is very limited:

- ✓ Protection Ring 3 on x86
- ✓ No access to the kernel memory
  - No visible in the page table (impossible to change without privilege)
- ✓ No low-level access to hardware
  - Except if prior authorization (ioperm, iopl, ...)

Processes

Exceptions & interruptions

System calls

Threads

Threads and processes

Presentation

- ✓ Unexpected events that temporarily suspend execution
- ✓ The processor jumps automatically and immediately to a handler:
  - Function whose address was defined by OS at boot
- ✓ Routing
  - Suspend execution of the current program
  - Resume the initial code at the end of the handler

Exceptions

- ✓ Interruption **by the processor** in case of error:
  - Arithmetical (division by 0)
  - Bad memory access (segfault, . . .)
- ✓ The processor doesn't know how to continue:
  1. It asks to the system to help it
  2. The system routes to the **handler** that will correct the error
  3. Resuming of the execution at the same location
     - the faulty instruction is re-executed
- ✓ The application does not realize anything
- ✓ Exception only possible when the processor is working
  - Requires an execution context (process, kernel thread, . . .)
  - In user or kernel space
- ✓ If the handler has successfully repaired the error:
  - Resuming of the execution (ex: a page fault)

  else
  - Task is killed (ex: segmentation fault)

Interruptions

- ✓ IRQ = Interruption Request
- ✓ Interruption = message of a **peripheral device**
  - If old device, electrical signal on a dedicated processor pin
  - if recent device, writing at a specific memory address
- ✓ **Asynchronous event**: the processor could continue to run by ignoring it during some time
  - but OS and/or applications would have problems: not completed I/O, network card saturated, ...
- ✓ Interruption **possible at any time**
  - Even if the processor does nothing
  - It is sufficient that only one device is initialized
- ✓ Temporary routing in the device driver
  - Ex: Request handling at reception of network packet
- ✓ No direct influence on the task that was running
  - Application does not realize anything
  - ...but can wake up tasks that were waiting on an I/O ending

Exceptions & interruptions

Very similar treatments:

1. Temporary use of the **privileged mode**
   - Only one core, only one suspended task
2. Execution of the **specific handler**:
   - Depends on exception or interruption . . . and parameters
3. **Resume** the initial execution

Processes

Exceptions & interruptions

System calls

Threads

Threads and processes

Special exception: the system calls

Definition

Exception forced by application to perform a privileged operation.

A system call:

- ✓ **switches the execution mode** of processor on demand
  - Switch to a more privileged mode by a special instruction: dedicated exception (int80, ...), Callgate, dedicated instruction (syscenter, syscall, ...)
- ✓ happens only from the user space, at an application request
- ✓ is performed in a process context, in privileged mode
- ✓ is called by applications in user mode: mainly by functions from `libc`
- ✓ is an obligation for all applications: impossible to cheat
  $\rightarrow$ check of parameters in privileged mode

How it works?

A system call is performed by the handler of the special instruction:

1. Handler address defined by the kernel at boot
   - Identical to handlers of interruptions/exceptions
   - The kernel jump directly at this address at the system call time
2. Temporary execution in privileged mode
   - the process is not put to sleep, its execution is routed
3. Call of a sub-handler in the table of system calls
   - table non-editable from the user space.

The call

Convention on parameters passing:

- ✓ Number in `%eax` on Linux/x86
- ✓ Arguments: `%ebx`, `%ecx`, `%edx`, `%esi` and `%edi`
- ✓ And in the stack if necessary (`mmap`)

Call sequence:

1. Save parameters
2. Save user context
3. Put parameters in registers
4. Call of handler → jump to `syscall_table[number]`

The resuming

1. Restore user registers
2. Convention on the return value of the system call: in `%eax` on Linux/x86
3. Possibilities to pass more parameters or return values
   - By pointers in args: the kernel can read/write in user space
4. Concrete return in user space by special instruction (`iret`, `sysexit`, ...)

Concretely in Linux

- ✓ Assembly code hidden in the `glibc`
  - call of normal functions
- ✓ 382 system calls in Linux 4.9/x86
  - A lot of are deprecated (51 unimplemented on x86_64)
  - The `glibc` adapts to system calls available on the computer

Look at the code of the `glibc` and the linux kernel !

Virtual system calls

- ✓ System calls are expensive
  - 100ns on modern x86, sometimes more than 1µs
- ✓ Sometimes stupid: the return of `getpid` not very variable.
- ✓ To optimize some calls in user space:
  - mapping of simple kernel codes and data (`gettimetoday`, `getcpu`)
  - remember some values (`getpîd`, `getppid`, ...)
- ✓ Risks of security? (virtual call for `setuid`)

Execution context

- ✓ Process in user mode
- ✓ Process in kernel mode during a system call
- ✓ Process in kernel mode during a handling of an interruption/exception
  - Temporary privileged mode
  - Process temporary suspended → resumes the control at the handling ending
  - Restricted execution due to a very limited context (deactivated interruptions)

Add one?

- ✓ Dangerous for security reasons
  - Ex: if I add a system call “I become root”
- ✓ Static table
  - Hard to modified
  - Often target of hackers (modification or replacement)
- ✓ Numbers fixed by binary compatibility
- ✓ How to do commands on demand? (in lecture on I/Os)

Processes

Exceptions & interruptions

System calls

Threads

Threads and processes

Create a process on Windows

- ✓ `CreateProcess()` create a process from a command line
  - New address space (and resources)
  - New execution queue in this space
- ✓ If we want modified attributes:
  - we have to pass a lot of arguments
  - or use `CreateProcessAsUser()`
  - Not flexible/extensible

Create a process on Linux

- ✓ Create a new process **then** execution of a command
  - `fork()` + `exec()` in place of `CreateProcess()`
  - We can configure a lot of things between the two commands
- ✓ `fork()` to duplicate
  - Duplication in user space (possibility to share resources)
  - New execution queue in this space
  - Child identical to its parent, except the PID
- ✓ `exec()` to transform:
  - a new program is executed by the process

Launching a task

The kernel creates a user context:

- ✓ pointing to the function to execute
  - `main()` for a process
  - the required function for a thread
- ✓ with a new stack

The older context is:

- ✓ destroyed in case of `exec()`
- ✓ kept in case of new task
- ✓ (does not exist in case of Init)

After creation, switch into user space.

Thread creation

We add an execution queue in the current address space

- ✓ `pthread_create()`

End of a task

- ✓ The end of a task takes place in the kernel
  - use of a system call or an exception
- ✓ The execution queue is destroyed
  - It frees up the used resources (the last user deletes it)
  - Identical for sharing between threads or processes

End of a thread

- ✓ `pthread_exit()` is called, implicitly or explicitly
  - resources specific to the thread are destroyed
- ✓ The last thread make the system call allowing to destroy the process
  - The shared resources are destroyed

End of a process

- ✓ A task ending sends a return code to its parent
- ✓ The parent wait the termination of one (`waitpid()`) or any (`wait()`) child
- ✓ The parent gets back the return code of its child
- ✓ A child is a zombie task while its parent ignore it

Process life and scheduling states

During a "normal" execution

Regular alternation between Ready and Running states according to the scheduler decisions

During a blocking system call

- ✓ Sleeping in a waiting queue (waiting on an event)
- ✓ Switch to Ready at the event reception (device interruption, action of another process, . . .)

Run queue

- ✓ Processes ready to be executed are stored in a special queue: a run queue.
  - it can be unique and global
  - we can have one run queue by core
- ✓ Only for the Ready processes!
- ✓ The not ready processes are in other special queues
- ✓ This avoid the search of the only ready process in a big queue!

Waiting queues

Processes not ready are in special queues:

- ✓ waiting queue of a driver if I/O waiting
- ✓ waiting queue dedicated to logical event
  - Ex: waiting on data in a tube
  - Ex: `pthread_join()`
- ✓ not in a queue only if they are waked up by PID
  - Ex: suspended by Ctrl+Z and waked up by SIGCONT
  - The place where signals are emitted for each PID is a pseudo waiting queue

Active waiting

The active waiting of an event:

- ✓ is an infinite loop (until the event occurs)
- ✓ monopolizes the processor until the event
- ✓ has a very good reactivity
- ✓ wastes processor cycles (acceptable in µs)

Semi-active waiting

The semi-active waiting is an active waiting by handing over the processor by using `yield()` at each loop.

- ✓ unpredictable reactivity
- ✓ can waste processor time
- ✓ can be used when a passive waiting is totally impossible

Passive waiting

- ✓ Wake up mechanism during an event
  - the task is store in a waiting queue
  - no longer in the run queue, ignored by the scheduler
- ✓ The one who triggers the event move the task from the waiting list to the run queue
- ✓ Putting into sleep and awakened is expensive
  - context switch
  - delay if it has not the priority (cannot preempt immediately)
- ✓ Does it worth to sleep?
  - No if the waiting delay is very short
  - Active waiting more reactive... but expensive

Need to find the right compromise.

Processes

Exceptions & interruptions

System calls

Threads

Threads and processes

Interest of threads

- ✓ Resource sharing between tasks
  - Trivial share memory, no complex initialization
- ✓ Better use of processors
  - One program on many processors
  - Some criticisms since the advent of multicore processors
- ✓ Recovering of I/Os
  - Execution of a thread while another is blocked by waiting for an event

Threads in Linux

- ✓ Intern to a process
- ✓ The process became the container
  - At least one thread
  - Shared resources

Shared resources... or not

- ✓ Distinct execution contexts: stack, registers, ...
- ✓ Possible to share:
  - Address space (in memory)
  - Open files
  - Signals
  - Process identifiers (sub-identifiers per thread)
- ✓ Sharing choice at creation time
  - see manpage of `clone()`

Kernel level threads vs. User level threads

Kernel level

- ✓ Threads supported by operating system
- ✓ OS handles scheduling, creation, synchronization

User level

- ✓ Library with code for creation, termination, scheduling
- ✓ Kernel sees one execution context: one process
- ✓ May or may not be preemptiv

User level threads

Advantages

- ✓ Low cost: user level operations that do not require switching to the kernel
- ✓ Scheduling algorithms can be replaced easily & custom to app
- ✓ Greater portability

Disadvantages

- ✓ If a thread is blocked, all threads for the process are blocked
  - Every system call needs an asynchronous counterpart
- ✓ Cannot take advantage of multiprocessing

You can have both

User level thread library on top of multiple kernel threads.

- **1:1** kernel threads only (1 user thread = 1 kernel thread)
- **N:1** user threads only (N user threads on 1 kernel thread/process)
- **N:M** hybrid threading (N user threads on M kernel threads)

Model 1:1

- ✓ One kernel thread for one user thread
- ✓ As expensive as a process
- ✓ Multi-thread processes advantageous: a normal time slice for each thread
- ✓ Ex: NPTL (Native Posix Thread Library)
  - Creation: 6µs
  - Context switch: 0.5µs

Threads in user space

- ✓ One kernel thread launch several user threads
- ✓ Reduced scheduling costs
  - We stay in user space
- ✓ Multiple threads ignored by the system
  - Only one time slice for all the process
  - if one thread is blocked, all threads are blocked
- ✓ Ex: Marcel
  - Creation: 0.21µs
  - Context switch: 0.22µs

Model M:N

- ✓ Several user threads on several kernel threads
- ✓ If one thread blocks a task, other tasks can run
- ✓ Add kernel threads on the fly:
  - Scheduler Activations: collaboration between kernel and user schedulers to add a kernel task if required
- ✓ Ex: Thr (Solaris), NGPT (Next Generation Posix Threading), Marcel

Context switch in user space

- ✓ How to save/restore the context without break the builder context
- ✓ `{set}{get}{make}{swap}context`
- ✓ `{set}{long}jump`

Behavior of multi-treaded programs

- ✓ During a `fork()`, duplicate all or only one thread?
- ✓ In case of fail?
- ✓ Only one thread: does that make any sens?