Processes, Threads, and Interrupts

Structure of processes and threads, purpose and implementation of interrupts

Contents

- What is a process?
- What are threads?
- Thread states
- Interrupts
- Processes and Lightweight Processes (Threads) in UNIX
  - Process structure
  - Creating and deleting processes
- Processes and Threads in Windows
  - Structure of processes and threads
  - Creating and deleting processes and threads
Programs, Processes, and Threads

Threads

Thread States

Interrupts

Interrupt Priorities

Control Flow: Threads und Interrupts

Processes, Threads, and Interrupts in UNIX

Address Space of a Process in UNIX

Process Creation in UNIX

Structure of a Program, which calls fork

Executing a Different Program

A Highly Simplified Shell

Process Hierarchy in UNIX

Data Structures for a Process Hierarchy

Terminating a Process in UNIX

Process States in UNIX

Lightweight Processes (= Threads)

Interrupts in UNIX (1)

Interrupts in UNIX (2)

Processes, Threads, and Interrupts in Windows

Virtual Address Space in 32-Bit-Windows

Virtual Address Space in 64-Bit-Windows

Processes, Threads, and Jobs

Process Creation (1)

Process Creation (2)

Process Creation (3)

Terminating Threads and Processes

Thread States

Interrupts (1)

Interrupts (2)

Interrupts (3)
Programs, Processes, and Threads

- When using multitasking, a computer executes several programs “simultaneously”. Every single program is executed in the context of a process (sometimes also referred to as a task).

- In today's systems, processes are further subdivided into executable units, called threads.

- The operating system chooses one of the threads for execution (per CPU), and switches forth and back between the different threads (scheduling).

- An important part of a process is its (virtual) address space, which contains, for example,
  - the code of the executed program,
  - the current values of the program's global variables.

Threads

- Most operating systems subdivide a process into threads.
  - A thread is an executable unit within a process.
  - Threads are the units to be scheduled.
  - The threads of a process
    - share a common address space (and hence all execute the same program),
    - each have a separate set of register contents (the thread context or hardware context),
    - each have a separate stack (which also contains the local variables).

- Advantages of threads:
  - On an SMP-system several threads of a process can be active at the same time.
  - Method of parallel programming on SMP-systems („shared-memory“ programming model).
  - Simple use of shared data by the threads of a single process (however: synchronization is necessary!).
  - Simplification of the program structure, for example for a server program, which handles several client requests.
Thread States

- A thread always is in one of three basic states:
  - **Current, or running**: the thread is using the CPU.
  - **Ready, or computable**: the thread could run, if it had the CPU (which currently is assigned to some other thread).
  - **Blocked, or waiting**: the thread waits for the occurrence of some event (e.g. completion of an input/output operation). At the moment it does not need the CPU.

![Thread States Diagram]

Interrupts

- A running thread can be interrupted by an interrupt. A distinction is made between:
  - **External interrupts**: caused by some asynchronous event (e.g. the system clock, or an I/O device).
  - **Exceptions** (internal interrupts): caused by an event that stems from the execution of the thread (e.g. division by zero, access violation, ...).

- When an interrupt occurs,
  - execution is switched to kernel mode,
  - part of the thread context (PC, SP, a few general registers) of the current thread is saved. (This is not a context switch, but a switch from thread activity to interrupt activity.)
  - the interrupt handler (or interrupt service routine) belonging to this particular interrupt is determined using an interrupt vector which was initialized when the system was booted,
  - the PC is loaded with the starting address of the interrupt handler, and the handler is executed,
  - The saved part of the thread context is restored, and execution is switched back to user mode,
  - control is returned to the interrupted thread or (if necessary) the scheduler is invoked.
Interrupt Priorities

- Every interrupt is assigned a specific interrupt priority.

Interrupt priorities must not be confused with thread priorities. Every thread, no matter of which (thread) priority, runs at interrupt priority 0.

- A status register in the CPU contains the interrupt priority the CPU is currently running at.

- An interrupt is only handled immediately, when its interrupt priority is strictly bigger than the current interrupt priority of the CPU.

- Interrupt priorities are also used for synchronizing access to the data structures of the operating system:
  - Every data structure is assigned an interrupt priority.
  - The data structure may only be accessed (by agreement), when the CPU is currently running with exactly this interrupt priority.

Control Flow: Threads und Interrupts

1. Thread is running
2. Interrupt occurs and control is transferred to the operating system
3. Interrupt is identified and dispatched to the appropriate handler for execution
4. Scheduler is called to select a new thread
5. New thread starts running
6. Return to interrupted thread
The address space of a process is divided into at least three areas (regions):
- **Text region:** the code of the program (shareable)
- **Data region:** data (variables) of the program
- **Stack region:** the stack(s)

In kernel mode, separate kernel regions are used.
Process Creation in UNIX

- Processes are created with the system call `fork`.
  - Syntax: `retval = fork();`
  - `fork` creates a child process, which is an exact copy of the parent process (i.e. same program, same PC, same open files etc.).
  - Only difference: the return value \(^{*}\text{retval}^{*}\) of the function call is the PID of the newly created process in the parent process, while it is 0 in the child process.

- Principal steps of `fork`:
  - Determine a unique ID for the child process,
  - Create a logical copy of the parent process,
  - (Details about how memory contents are handled can be found in the chapter „Memory Management“)
  - Increment the usage counters for open files,
  - Return the ID of the child process to the parent process, 0 to the child process.

Structure of a Program, which calls `fork`

```c
retval = fork();  /* retval is used to distinguish parent and child process */
if (retval < 0) {
   /* fork unsuccessful, e.g. because memory or process table are full */
} else if (retval > 0) {
   /* Code for the parent process */
} else {
   /* Code for the child process */
}
```
Executing a Different Program

- The system call `exec`
  - calls some other program, hence loads the address space of the process with the contents of a new program file (executable image).
  - Syntax: `exec (filename, argv, envp)`
    - `argv` Pointer to the arguments for the program
    - `envp` Pointer to environment variables for the program
  - When a program contains the line
    ```
    main (argc, argv)
    ```
    then `argv` is a copy of the `argv`-parameter of the `exec`-call
- Example:
  ```
  main()
  {
    int status;
    if (fork() == 0) /* if child process */
      exec("bin/date","date",0);
    wait(&status) /* Get exit state of child process */
  }
  ```

A Highly Simplified Shell

```c
while (1) {
    /* repeat forever */
    type prompt(); /* display prompt at terminal */
    read_command(command,params); /* read input */
    retval = fork();
    if (retval < 0) {
        printf("fork cannot be executed");
        handle_error; /* e.g. end process */
    }
    if (retval != 0) {
        /* parent process */
        waitpid (-1, &status, 0);
    } else {
        /* child process */
        exec (command, params, 0);
    }
}
```
Process Hierarchy in UNIX

- During the bootstrap, the process P0 is created „from scratch“. P0
  - always runs in kernel mode,
  - creates the process P1 (init-process), which executes the program /etc/init in user mode,
  - creates further kernel mode processes,
  - thereafter becomes the swapper process.

- P1 is the ancestor of all other processes.
  - P1 creates „daemons“, that is processes independent of any user, for particular tasks like, for example,
    - printer spooling,
    - network processes.
  - P1 creates a separate process for each interactive user.
  - P1 is blocked until one of its child processes terminates.

Data Structures for a Process Hierarchy

- In Linux, the process data structures contain the following pointers, which constitute the process hierarchy:
Terminating a Process in UNIX

- Processes are terminated with the system call `exit`.
  - Syntax: `exit (status);`
  - The status value is retained, so that the parent process can request it.

- Important steps when executing `exit`:
  - All child processes continue to run and obtain the init-process (P1) as their parent process, except:
  - When the process is connected with a terminal, all child processes obtain a `hangup`-signal and terminate themselves.
  - Files are closed and resources released.
  - The process state is set to `zombie`.
  - A `death-of-child`-signal is sent to the parent process.

- Termination of a process is requested
  - explicitly by calling `exit`,
  - implicitly through an internal call to `exit` by the kernel, e.g. when the program ends, or when the process obtains a signal (see the chapter „Synchronisation“).

Process States in UNIX

1. User Mode Running
2. Kernel Mode Running
3. Created
4. Ready to run
5. Preempted
6. Zombie
7. Fork
8. Swapped
9. Swapped

Diagram:
- User Mode Running
- Kernel Mode Running
- Created
- Ready to run
- Preempted
- Zombie
- Fork
- Swapped
- Swapped
Lightweight Processes (= Threads)

- In most Unix derivates, threads are called lightweight processes.
- Each process can contain one or more lightweight processes.
  - Some informations in the data structures of the process are specific to a certain lightweight process and must hence be moved to appropriate new data structures.
  - The implementation of some system calls must be rethought and possibly modified. Example: should fork only duplicate the calling lightweight process, or all lightweight processes?
- The units for the scheduling done by the operating system are the lightweight processes.
- Most Unix derivates also have so-called user threads. They differ significantly from "normal" threads (i.e. lightweight processes), and will not be covered here.

Interrupts in UNIX (1)

- Kinds of interrupts:
  - Exceptions
  - Hardware Interrupts
  - Software Interrupts
- Interrupts have a priority (processor execution level), e.g.
  
<table>
<thead>
<tr>
<th>Power Failure</th>
<th>Clock</th>
<th>Network Devices</th>
<th>Disk</th>
<th>Terminals</th>
<th>Software Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  - An interrupt is only handled immediately, when its (interrupt) priority is bigger than the interrupt priority of the currently executing code.
Interrupts in UNIX (2)

Steps when handling an interrupt:
- Switch to kernel mode.
- Save the current register contents (new context layer on the stack).
- Determine the address of the interrupt handler through the interrupt vector.
- Run the interrupt handler.
- Return and restore the previous context.

When a system call is made:
- Same steps as with an interrupt, except that the address of the code is taken from a system call table.

A series of interrupts

Operating Systems 1
Processes, Threads, and Interrupts 21
Chr. Vogt
Virtual Address Space in 32-Bit-Windows

- 32-bit Windows uses a linear 32-bit address space:

- In WNT Enterprise Edition, W2K Advanced Server, and Enterprise Server 2003, the process address space can be extended to 3 GB by specifying a boot switch.

Virtual Address Space in 64-Bit-Windows

- The current implementation of Windows XP / Vista / Windows 7 and Windows Server 2003/2008 on 64-bit processors uses:
  - 8192 GB for the process address space (resp. 7152 GB on IA-64)
  - 512 GB for the system address space

- These current implementation limits can easily be extended in future releases.
Processes, Threads, and Jobs

- All processes in Windows are subdivided into threads.

- A thread
  - shares the virtual address space with all other threads of the same process,
  - shares resources (e.g., access tokens, object handles) with all other threads of the same process,
  - has a separate thread context (contents of the registers),
  - has a separate user mode and a separate kernel mode stack.

- A job object (introduced with Windows 2000) combines one or more processes. All processes of a job can be handled as a single unit, e.g., when
  - establishing CPU time limits,
  - assigning priorities,
  - etc.

Process Creation (1)

- The way of creating a process depends on which subsystem performs the process creation.
Process Creation (2)

- The subsystem obtains a process handle and provides the application with the information specific for this subsystem.

![Diagram of process creation](image)

Process Creation (3)

- A process can be created
  - by calling an appropriate API function (e.g., `CreateProcess` in the Win32 API),
  - by typing the name of an executable file in a command window,
  - by double-clicking an executable file in the explorer.

- During process creation
  - it’s determined what kind of file it is:
    - MS-DOS program
    - MS-DOS batch file
    - Win16 program
    - Win32 program
    - OS/2 program
    - POSIX program
  - the necessary subsystem will be started if necessary (only the Win32 subsystem is always running),
  - the process and the primary thread are created, and the program is started in the primary thread.

- Additional threads can be created by the program with the Win32 API function `CreateThread`. 
Terminating Threads and Processes

- A thread is terminated when its program terminates.
- A process is terminated, when the last thread of the process terminates.
- All types of objects - hence also threads and processes - are automatically deleted by the Windows object manager when
  - there are no more handles for this object, and
  - there are no more references by the operating system to the object.
- The respective thread or process object will hence only be deleted, when no other process has an object handle to it.

Thread States

- Create and initialize thread object → Initialized
- Place in ready queue → Read
- Thread waits on an object handle → Waiting
- Set object to signaled state → Ready
- Select for execution → Running
- Context switch to it and start its execution (dispatching) → Standby
- Preempt (or time quantum ends) → Preempt
- Execution completes → Terminated
Interrupts (1)

- Kinds of interrupts:
  - Hardware interrupts: asynchronous
  - Software interrupts: asynchronous
  - Exceptions: synchronous

**Remark:** System calls are handled very similar to exceptions.

- Interrupts have an **Interrupt Request Level (IRQL)**.

- Interrupts are handled by the trap handler in the kernel.
  - Execution switches to kernel mode, and the processor level is raised to the IRQL of the interrupt.
  - A trap frame is created, saving the status of the interrupted thread.
  - The service routine belonging to this interrupt is called and executed.
  - The previous IRQL and thread status are restored, and execution switches back to user mode.

Interrupts (2)
Interrupts (3)

1. An interrupt occurs.
2. The interrupt dispatcher retrieves the IRQ of the interrupt source and indexes into the IDT.
3. The interrupt dispatcher follows the pointer and calls the correct handling routine.

<table>
<thead>
<tr>
<th>IDT</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
</tr>
<tr>
<td>Power</td>
</tr>
<tr>
<td>IPI</td>
</tr>
<tr>
<td>Clock</td>
</tr>
<tr>
<td>Device n</td>
</tr>
<tr>
<td>Device 1</td>
</tr>
<tr>
<td>Dispatch/DPC</td>
</tr>
<tr>
<td>APC</td>
</tr>
<tr>
<td>Low</td>
</tr>
</tbody>
</table>