Operating Systems, Concurrency and Time

Interaction with the Operating System

Johan Lukkien
Questions

• How does an OS serve applications?
  – how does it work, roughly?
  – how does an implementation of, e.g., a semaphore look?
  – how are scheduling decisions taken?

• What are performance considerations?

• What are methods to improve control over real-time behavior?
OS Kernel

Provided service (system call API) to applications (running processes)

Required service from platform (processor, other hardware)

Process management
• `fork()`, `exec()`, `exit()`, `wait()`

Thread management
• `pthread_create()`, `pthread_join()`, ...

Synchronization
• `semaphores`, `condition variables`, ...

Communication
• `pipes`, `sockets`, ....
Provided service (system call API) to applications (running processes)

OS Kernel

Required service from hardware platform (processor, other hardware)

- Kernel/user mode
- trap & interrupt mechanism
- Memory management, protection
- Fast exclusion mechanism
OS API: interactions between user code and OS

- Interaction with the OS is named a *system call*

- Example: execute C statement: \( \text{status} = \text{read}(\text{fd}, \text{buffer}, \text{nbytes}) \)
  - read \( \text{nbytes} \) bytes from \( \text{fd} \), storing it in \( \text{buffer} \)
  - part of a program running on top of the OS (a user program)

- memory: kernel space/user space
  - kernel space only accessible with processor in kernel mode

- parameters: either via registers or via memory
  - 1-3: pushing parameters
  - 4: call library function
  - 5: put code for \text{read} in reg.
  - 6: trap: switch mode and call particular handler
  - 7: handler calls read function handler
  - 8: handler performs read actions (a.o. store data at address). Here *suspension of the calling process* may occur if data needs to come from an io device.
  - 9-11: control back to caller
What makes real-time OS’s different?

- **The API**
  - real-time programs need specific services
    - dealing with time and scheduling, manipulation of sensors & actuators
  - easy extensions for user-specific functionality

- **The footprint** – memory use (for embedded use)
  - small
  - can tweak the footprint to the used functionality – no cost for unused functionality
    - e.g. leave out file-system in embedded application

- **The predictability**
  - known performance, and also high performance in response
    - latency and jitter of API calls and interrupt responses known and bounded
  - protocols for dealing with blocking (system calls, synchronization)
  - sufficient priorities for threads and queues

- **The Architecture**
  - targeted towards predictability and timeliness (low latencies)
  - and a small footprint for embedded purposes
Predictable computation time

- Make sure that the task can finish within $D$
  - Worst case execution times $C$ (WCET) should be calculable ($C \leq D$)
  - The load on the processor resource is not more than $C$ per period $T$

- The predictability of this is endangered by
  - anomalies of the hardware and system software
    - use of DMA that locks the bus
    - cache behaviour
    - interrupts
    - memory management
  - constraints
    - finding required resources locked by other tasks
    - precedence constraints
  - varying resource availability
    - competition of applications
    - physical causes (e.g. interference in wireless channels)
  - absence of transparency in high level languages
    - dynamic variables, garbage collection (memory management)
    - repetition, recursion
Context and System Calls
How does the kernel get control?

Asynchronous interrupt: breaking control flow of application

Synchronous interrupt: part of control flow of application
Mode and context

• A *mode switch* (between kernel and user mode)
  – is needed for any event that requires kernel activity
  – changes the processor mode and loads a new program counter
  – must usually be followed by saving some additional processor state
  – is cheap

• A *context switch* (changing the memory and processor registers)
  – changes the memory management settings
  – changes which user code is executing
  – is needed, besides a mode switch, *for only some events*, e.g. if a process switch is needed
    • remember: a process defines a memory space

• Except for the handler, most of the overhead of a mode switch is performed in hardware
  – hence, we would like to restrict the majority of the OS event handling to just a mode switch
Process State

• A process is represented by a *Process Control Block* (PCB) “inside” the kernel (1 thread for simplicity)
  
  – all info needed to run it
  – identified by an ID or a pointer to this block
    • dependent on representation details, e.g. table, linked list
    • we use: `self` to refer to the current process PCB
  – *state*: ready, waiting, ....
    • + perhaps the queue where the process is on
  – *scheduling queues*
    • queues where this process is part of
  – “...” includes
    • quota limits
    • accounting information
      – e.g. times in user, kernel mode
    • user identification
    • permissions
Typical memory layout for a process

- In user mode, the (threads of the) process can access just its private data and stack

- In kernel mode, the (thread of the) process can access kernel functions and data
  - it uses a new stack for that purpose, stack pointer loaded as part of the trap
  - the kernel runs, in fact, on behalf of the process
    - “the thread (process) runs in kernel mode”
  - this space is shared by all processes

- No other processes are visible
  - hence, in order to deliver data to another process, this data must be copied in between
  - the kernel cannot deliver data to just any process; it first must change the memory settings such that this process becomes visible

- Many system calls are just a trap
Context switch

process $P_0$  
executing

<table>
<thead>
<tr>
<th>interrupt or system call</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{save state into PCB}_0 ]</td>
</tr>
<tr>
<td>[ \cdots ]</td>
</tr>
<tr>
<td>[ \text{reload state from PCB}_1 ]</td>
</tr>
</tbody>
</table>

idle

executing

<table>
<thead>
<tr>
<th>interrupt or system call</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{save state into PCB}_1 ]</td>
</tr>
<tr>
<td>[ \cdots ]</td>
</tr>
<tr>
<td>[ \text{reload state from PCB}_0 ]</td>
</tr>
</tbody>
</table>

idle

executing
Blocking events, queues

- ready queue
- CPU
- I/O
- I/O queue
- I/O request
- time slice expired
- child executes
- fork a child
- interrupt occurs
- wait for an interrupt
Outline of system call handling

**Trap Handler:** *(continue here after mode switch)*
- Check Validity of parameter;
- Call SystemCalls [parameter] (arguments of call);
  - \{ SystemCalls is a table of functions; \}
- Execute return path; \{ check on pending signals, remaining kernel tasks, the need to reschedule \}

**Call implementation:**
- non-blocking: perform the kernel action and return here
- blocking: enqueue Self and call scheduler; return here only after re-schedule

Example: `read(fd, buffer, cnt)`
Example: implementing general P/V

\( P(s): \)

1. EnterCS; \{ inhibit interrupts; and use \( s->lock \) (spinlock) for this semaphore \}
2. \textbf{while} \( s->val=0 \) \textbf{do}
   1. Store self in \( s->queue; \)
   2. Call Scheduler; \{ stop current (putting it in blocked queue!), LeaveCS, start other; \}
   3. EnterCS; \{ resume critical section \}
   4. \textbf{od;}
3. \( s->val := s->val-1; \)
4. LeaveCS \{ Release \( s->lock \) and enable interrupts \}

\( V(s): \)

1. EnterCS; \{ inhibit interrupts; and use \( s->lock \) (spinlock) for this semaphore \}
2. \( s->val := s->val+1; \)
3. \textbf{if} not empty(\( s->queue \)) \textbf{then}
   1. Release one from \( s->queue; \)
4. \textbf{fi}
5. LeaveCS; \{ Release \( s->lock \) and enable interrupt \}

- the semaphore is here a shared data structure (in the kernel)
  - containing lock, value and queue, in this case
  - created by API calls like the ones presented before
  - \( s \) is a descriptor, a pointer to this data structure
- unfairness possible (how?)
- \( P(s) \) has only the waiting part of the call; \( V(s) \) has only the releasing part
- manipulating the process records in the Scheduler and the Release needs protection
Device drivers an interrupts
How are fast responses realized?

• Responses: interrupt, system call

• The essential issue is to control (predict, limit) the duration of any activity of the kernel
  – the longest code path in the kernel
  – the longest non-interruptable path in the kernel

• RTOS: relevant times are part of the specification

• Regular OS: optimized for average case

• Typical approach: *micro kernel*, small code base for essential services, everything else outside

Johan J. Lukkien, j.j.lukkien@tue.nl TU/e Informatica, System Architecture and Networking
Driver elements and control flow

**CONTROLLER**
- may block the caller
- program the controller according to the request
- (or pass on to intermediate layer)

**DRIVER**
- synchronous handling:
  - may block the caller

**asynchronous handling:**
- may unblock previously blocked callers
- *top half*: executed upon interrupt (ISR)
- *bottom half*: executed when OS gets to it

**DEVICE MANAGEMENT** (manage API)
- read
- write .....
Driver elements

• Two parts in a driver:
  – a *synchronous part*: called via user process-kernel sequence
    • results possibly in waiting / suspension of caller
      – but can be done entirely in the system call; if so, the synchronous function simply returns after preparing the device
    • wakeup of such a waiter is through the interrupt routine (the asynchronous part) accessing the data structure recording the waiters
  – an *asynchronous part*: called upon events from the device
    • typically, the *interrupt service routine* (ISR)
    • devided into a *top half* (executed upon handling the interrupt)
      – the most critical actions
    • and a *bottom half* (executed when the OS finds time for this)
      – actually delivering the result to the calling process
      – called: DPC in Windows
  – top half and OS operate as concurrent programs
Responding to interrupts

• State after interrupt occurrence
  – kernel mode, interrupted state saved, all interrupts disabled

• Task of interrupt handling (not necessarily this order)
  1. save additional state not saved by the hardware
  2. re-enable interrupts, except the one under consideration and possibly lower priority ones
  3. take away volatile state from the interrupting device as soon as possible
  4. re-enable the interrupt(s)
  5. process the obtained information
  6. wake up waiting processes and select a new one for continuation
  7. return from interrupt

• Mostly too long for a single critical section
  – 1. through 4. can be done quickly (“top half” in Linux)
  – 5. can be done in a separate task (“bottom half”, DPC)
  – 6. can be done in the return path (from interrupt, from system call)
When to service pending “bottom halves”? 

- **Design options**
  - upon returning from interrupt
  - upon returning from a system call
  - upon calling the scheduler
  - by a separate process, with highest priority
    - a “kernel event handler”, *keventd* in Linux 2.4
    - (replaced by ‘worker threads’ in 2.6)
  - Windows: DPC are queued in three priorities; queue is emptied in response to a software interrupt at DPC/DISPATCH level

- **Possible to have these all as classes, based on urgency**
  - Linux 2.4, Windows

- **Fits naturally the design of a microkernel**
  - driver is a separate, user-level process
  - “top-half” delivers a message to the driver
  - Minix 3
Linux and real-time
Jitter upon periodic task release

• Jitter has several contributions
  – inaccuracy in generating the timer event
  – variations in response to this event
  – variations in actually starting the user handler

• Notice: need to avoid adding up jitter over time (drift)
  – schedule on absolute times, not relative
Latencies and jitters

interrupt (or: event) occurs

OS handler starts

job (task instance) starts

jitter

interrupt handling latency

minimum value for begin time (hence, response time)

must use this as the worst case event latency (activation latency)
(often added to worst-case computation time of the task)
The example of Linux and real-time

- Originally, *Linux kernel not preemptable*
  - kernel acts as a shared resource
  - system calls may last unpredictably long
  - hence, calls of one process may delay call response of another process

- *Interrupt masking (disabling)* during interrupt servicing and parts of kernel operation
  - Interrupt servicing in two parts: bottom and top
  - Top half with interrupts disabled, handles the urgent work
  - Interrupts are disabled too in larger kernel fragments

- Resulting in *blocking and (priority inversion)*
  - on system calls, on interrupt servicing

- *Time granularity*
  - event latency, system call latency and switching time (process / thread /task switching) determine in fact the timescale the system can operate
    - 10+ ms on non-preemptable Linux, 1 ms on preemptable kernel on 750MHz Pentium

- *Virtual memory*
  - adding unpredictability to the first two bullets as well
Approaches for adaptation to real-time

• Make the kernel preemptable (since 2.5.4)
• Patches for low latency

• Let Linux run as a separate (low priority, background) activity under a separate, new, real-time kernel
  – generally referred to as: ‘real-time Linux’
  – RTAI (simple kernel patch), RTLinux (significant rewrite), eCOS, all kind of derivatives

• Add the POSIX 1003.1d real-time extensions
  – also change interrupt handling architecture
  – TimeSys Linux, significant rewrite and addition

• Virtualize the hardware to serve multiple OS
  – ADEOS (Adaptive Domain Environment for OS), XEN,
Measurements on Linux’ induced activation jitter under competing load (2002)

- **Standard**: regular Linux kernel, before 2.5.4
- **Low-latency**: patch for low latency
- **Preemptable**: patch (rewrite) for preemptable kernel (from version 2.5.4)
- **Preemptable-lock-breaking**: in addition, the ability to break long-lasting blocking
Addressing the interrupt latency

- **Interrupt latency:**
  - time lapse from interrupt to start of service routine
  - time lapse from interrupt to start of task instance

- In order to obtain lower latency:
  - increase handling speed (pertains to service routines)
  - decrease inhibition periods (pertains to kernel code & service routines)
  - shorten the time until the interrupt reaches the task (transfer control to task much faster)

<table>
<thead>
<tr>
<th>Application</th>
<th>Latency/Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard O/S</td>
<td>Non Real Time</td>
</tr>
<tr>
<td>Standard Linux</td>
<td>Soft Real Time</td>
</tr>
<tr>
<td>IEEE 1003.1d Linux</td>
<td>Hard Real Time</td>
</tr>
<tr>
<td>Real-Time Linux</td>
<td>Hard Real Time</td>
</tr>
<tr>
<td>RTOS Kernels</td>
<td>Hard Real Time</td>
</tr>
<tr>
<td></td>
<td>100 micro to 100 milliseconds</td>
</tr>
<tr>
<td></td>
<td>1 millisecond</td>
</tr>
<tr>
<td></td>
<td>10 to 10 microseconds</td>
</tr>
<tr>
<td></td>
<td>1 to 10 microseconds</td>
</tr>
<tr>
<td></td>
<td>1 to 10 microseconds</td>
</tr>
</tbody>
</table>

Table from “Introduction to Linux for Real-time Control”, NIST publication (measurement: 2002/2003)
Preemptable kernel with POSIX 1003.1d

- The kernel is capable of
  - preempting lower priority activities, interrupt handling
  - limiting (bounding?) periods of interrupt inhibition

- ‘Bottom halves’ of the interrupt service routines are scheduled, similar to other tasks

- The real-time support within the kernel serves the real-time user processes

- Example: TimeSys Linux

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

7-Dec-16
Adding a micro kernel

• The micro kernel intercepts all (most) interrupts

• Real-time tasks
  – run entirely within the kernel (in kernel mode)
  – are scheduled by the micro kernel

• Allows high-resolution timing

• Communication with user processes is through FIFOs (unstructured byte sequences following first-in-first-out policy)

• Example: RTLinux, RTAI
Adding just interrupt dispatching (nano kernel)

- The nano kernel intercepts all interrupts
- The interrupts are dispatched in order of importance
- Several OS’s can run concurrently
- Possibility: run a micro kernel *concurrently* with Linux
- Example: ADEOS
  - not typically Linux-dependent
  - used for RTAI, because of patent problems

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

7-Dec-16
Linux and real-time, summarizing

• The distance event-task (= event handler) is made small
  – by placing real-time tasks within the kernel
  – real-time tasks take priority over kernel activities

• As a result
  – a special API is constructed for task manipulation, communication, synchronization and timers
  – parts of the application that do not need real-time guarantees can be moved to regular user programs
    • general rule: communication between real-time tasks and non-real-time parts (even system calls!) must be non-blocking. FIFOs or non-blocking synchronization realize this.

• Comparing to ‘regular’ RTOS approaches
  – a regular RTOS associates tasks with user-level threads and processes
  – .... making it necessary to let the entire kernel be predictable
Scheduling
Scheduling (allocation)

- Resource scheduling (allocation): assignment of resources to tasks

- Schedule $S$ is a function that maps a time and a resource to a task: $S(t,r) = P$ means that task $P$ is assigned resource $r$ at time $t$

- What a task is, is context dependent
  - e.g. a process, a thread, the reading of a disk block, handling an interrupt
  - .... a thread of execution, an activity

- The processor resource is a special case
  - we usually say that the task is running

- The interesting part of a schedule is when there is a change
  - the decision procedures for change are interesting
  - also when decisions are possible (system is in decision mode)
    - when is this mode reached for e.g. the processor resource or a memory segment?
      » processor: e.g. process into ready queue; end of time slice; process yielding (blocked)
      » memory: e.g. memory management call; replacement policy by memory subsystem; process termination
Example schedules

- $P_0$, $P_1$, $P_2$: tasks
- $t_0$, $t_1$,....: scheduling points
  - system is in decision mode
  - scheduling decisions are taken
  - scheduling points may differ per resource (not shown here)

- processor resource: $PROC$
  - $S([t_0..t_1), \text{PROC}) = P_0$
  - $S([t_1..t_2), \text{PROC}) = P_1$, etc.

- a memory page: $m_{23}$
  - $S([t_0..t_4), m_{23}) = P_0$, etc.

- another memory page: $m_{56}$
  - $S([t_3..t_4), m_{56}) = P_2$, etc.
Scheduling policies and mechanisms

- The scheduling policy represents the strategy for allocating a resource to a task while in decision mode
  - policy: informal algorithm, tells the decision based on scheduling criteria....
    - task attributes (deadline, response time, ...)
    - the current state, represented by sets of ready processes as well as available and required resources
  - ....or on a lookup
    - pre-computed table

- Scheduling (allocation) mechanisms are different per resource type
  - processor:
    - implicit (i.e., not visible in the program text) management at scheduling points by the OS
      - through saving and restoring context
  - passive resource:
    - explicit locking and unlocking through e.g. semaphores (with FIFO policy) or implicit upon function entry and exit [monitors, e.g. Java]
    - implicit management through the OS, e.g. for memory
Resources and preemption

• A task using a resource usually generates associated state
  – registers etc. in a processor
  – variables inside an object
  – ... what about a cache? is there state?
  – ... are there state-less resources?

• Upon preemption (due to priority or time slice expiration)
  – save the state
  – or destroy the state – roll-back
    • destroys the effort (work) in obtaining it
    • is a network interface card preemptable? or a printer?
  – associated penalty: context-switch time (of that resource)

• Preemption not always (directly) possible
  – hold the resource until preemption is possible (or operation finished)
  – leads to cooperative scheduling
  – penalty for waiters: incurred blocking time (perhaps even deadlock) and priority inversion
Policy: first (last) come first serve

- Resource assigned to tasks in (reverse) order of request arrival
  - resources held until task releases them
  - FCFS results in *non-preemptable* resource allocation
    * only resource scheduling at task synchronization points
    * example: printer spooling

- Applicable to all resource types
  - but typically for state-holding, non-preemptable resources
  - ... because for those, preemption means a loss of work
Policy: time sliced

• Time sliced: resource is given to a task for a certain amount of time (the *quantum*)
  – needs preemptable resources
    • e.g. processor
  – use in combination with other policies
    • round robin = time sliced + fcfs (with partial served sent back to queue)
    • typically for a processor

• Other policies
  – Real-time policies
  – Shortest job next
    • Printer spooling
    • Minimizes the metric: *average waiting time*
Real-time policies

• Fixed Priority: assign priority once and for all
  – Non-preemptive (FPNS)
  – Deferred preemptive: limited preemption points indicating subtasks (FPDS)
  – Preemptive (FPPS)

• Dynamic Priority: determine or vary priority during execution

• Priority assignment
  – *rate monotonic*: assign priorities in order of increasing task periods
    • optimal among all fixed priority assignments for FPPS
  – *deadline monotonic*: assign priorities in reverse order of task deadlines ($D$)
    • optimal when deadlines are shorter than periods
  – *earliest deadline first*: dynamically assign priorities to jobs in reverse order of job deadline
    • optimal
    • admits 100% utilization
Concerns for real-time policies

• Compute the response times
  – ‘sufficient’ or exact schedulability test
  – online acceptance test of a new task
  – critical instant, per task: situation leading to maximum response time
  – how to deal with blocking

• Effect of overrun
  – are all tasks penalized or just the overrunning one
    • EDF: all tasks
    • RMS: just the overrunning task and lower priorities

• Utilization
  – difference between average and worst case
Process scheduling: decision mode

• Process executes a blocking kernel call
  – e.g. a disk read of a block currently not in memory
  – the implementation of the call will store the process state at the reason of the blocking

• A timer interrupt occurs
  – the timer service routine may cause switching processes and use the information saved upon interrupt for this purpose
    • store context of interrupted process

• The process yields the processor
  – i.e. a ‘yield’ call
  – or after a call that changes a process state (e.g. fork(), exit() or communication), rescheduling may be needed
Exercises

• **O.1** Write a small program that measures the execution time of a system call of your choice repeatedly.
  – do this with and without a competing load (i.e., another program that also runs).
  – show the result as a cumulative distribution function

• **O.2** The same exercise as above but now for a timer that fires with a given frequency. Determine the difference between the intended time and the actual time the invoked function receives control.
Exercises

• O.3 Given is a set of tasks. First, ignore the period.
  – Determine schedules and average response times (i.e. the time that elapses from the arrival of a task until the completion) resulting from the shortest job next policy (with preemption) and from the first-in-first-out policy.
  – How much memory is required in total in these cases?
  – How do these numbers compare to round robin with quantum 1?

Second, examine the period
  – Draw a schedule for RMS and EDF, with and without preemption. If deadlines are equal to periods, are all deadlines met? What is the minimum deadline for all tasks?
Exercises

  - How does Windows assign priorities?
    - how many, which classes etc.
  - What is the difference between IRQ levels and priorities?
  - How does Windows manage the dispatcher database in case of multiple processors? What is the motivation for this?
  - How does windows establish quick response to the UI?
  - In which cases is the priority ‘boosted’ and why?
  - How can you deal in Windows with NUMA properties of the machine?
Exercises

• **O.5** Look at your own code project.
  – Which resources do you need and use for which there is competition with other processes on the same computer?
  – Do you influence the scheduling of your system in any way? Would such influence be useful?
Priority Assignment

- **Process:** base priority
  - typically, center of class

- **Threads:**
  - derive priority from process base
  - relative w.r.t. base
  - can be temporarily boosted by OS (only dynamic range)

- **API to manipulate these**
Priorities and interrupt levels

- **APC**: asynchronous procedure call
  - delivered to a thread
  - request to execute a function, in context
    - e.g. I/O completion

- **DPC**: deferred procedure call
  - bottom halves of interrupt routines

- Passive level is reached only when there are no pending DPC / APC
Thread states

- deferred: used to minimize locking time
- standby: thread per processor
- (I think ready \(\rightarrow\) running should be ready \(\rightarrow\) standby)
• Exclusion on multiprocessors:
  – hold spinlock at IRQL_SYNCH_LEVEL
<table>
<thead>
<tr>
<th>T</th>
<th>L1</th>
<th>L2</th>
<th>U1</th>
<th>U2</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
<td>30</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>10</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>T</th>
<th>A, B, C, D</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T</th>
<th>100+200+300+400 = 1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>300+400 = 700</td>
</tr>
<tr>
<td>T</td>
<td>100+200+300 = 600</td>
</tr>
</tbody>
</table>

Race condition interference