Introduction to Real-Time Operating Systems

SoSe 2018

There will be small text (example code) on the slides, so move up close!
Linux-based **Multiprocessor** Real-Time Operating System

*Developed at MPI-SWS, actively maintained since 2006.*
Agenda

1 Overview
- What is a real-time OS and why should you use one?
- What makes a real-time OS different from a general-purpose OS?
- different classes of real-time operating systems

2 Static RTOSs
- theory to practice: periodic & sporadic tasks in OSEK

3 UNIX-like RTOSs
- theory to practice: periodic & sporadic tasks in POSIX

Ask questions!
What is a real-time OS (RTOS)?
What is a real-time OS (RTOS)?

A resource management and abstraction layer that simplifies the implementation of predictable and well-structured real-time applications.
What is a real-time OS (RTOS)?

A resource management and abstraction layer that simplifies the implementation of predictable and well-structured real-time applications.

**predictable**

- worst-case delays can be determined statically
  - under reasonable assumptions
- system services not subject to inherent unpredictability
What is a real-time OS (RTOS)?

A resource management and abstraction layer that simplifies the implementation of predictable and well-structured real-time applications.

**predictable**

- worst-case delays can be determined statically
  - under reasonable assumptions
- system services not subject to inherent unpredictability

**resource management & abstraction**

- multiplexing of limited hardware resources
- idealized & unified interfaces
<table>
<thead>
<tr>
<th>primary goal</th>
<th>Real-Time OS</th>
<th>General-Purpose OS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>full control over everything</td>
<td>one size fits all / ease of development / ease of deployment / ease of use</td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>--------------------------</td>
<td>------------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td><strong>primary goal</strong></td>
<td><strong>full control</strong> over everything</td>
<td>one size fits all / ease of development / ease of use</td>
</tr>
<tr>
<td><strong>system configuration / driver selection</strong></td>
<td>static / tailored at <strong>compile time</strong></td>
<td>dynamic / discovered at <strong>runtime</strong></td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------------</td>
<td>--------------------</td>
</tr>
<tr>
<td><strong>primary goal</strong></td>
<td>full control over everything</td>
<td>one size fits all / <strong>ease of development</strong> / easy of deployment / ease of use</td>
</tr>
<tr>
<td><strong>system configuration / driver selection</strong></td>
<td>static / tailored at <strong>compile time</strong></td>
<td>dynamic / discovered at <strong>runtime</strong></td>
</tr>
<tr>
<td><strong>performance criterion</strong></td>
<td><strong>predictable</strong> worst case</td>
<td><strong>mean</strong> throughput and latency</td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------</td>
<td>---------------------------------------------</td>
</tr>
<tr>
<td><strong>primary goal</strong></td>
<td>full control over everything</td>
<td>one size fits all / ease of development / easy of deployment / ease of use</td>
</tr>
<tr>
<td><strong>system configuration / driver selection</strong></td>
<td>static / tailored at compile time</td>
<td>dynamic / discovered at runtime</td>
</tr>
<tr>
<td><strong>performance criterion</strong></td>
<td>predictable worst case</td>
<td>mean throughput and latency</td>
</tr>
<tr>
<td><strong>resource management</strong></td>
<td>exclusive reservation / provision on (or near) worst-case needs</td>
<td>overcommit / stochastic multiplexing</td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>--------------------------------------------------</td>
<td>------------------------------------------------</td>
</tr>
<tr>
<td><strong>primary goal</strong></td>
<td>full control over everything</td>
<td>one size fits all / ease of development / ease of use</td>
</tr>
<tr>
<td><strong>system configuration / driver selection</strong></td>
<td>static / tailored at compile time</td>
<td>dynamic / discovered at runtime</td>
</tr>
<tr>
<td><strong>performance criterion</strong></td>
<td>predictable worst case</td>
<td>mean throughput and latency</td>
</tr>
<tr>
<td><strong>resource management</strong></td>
<td>exclusive reservation / provision on (or near) worst-case needs</td>
<td>overcommit / stochastic multiplexing</td>
</tr>
<tr>
<td><strong>workloads</strong></td>
<td>fully known, mostly static, under control of engineer, often simple</td>
<td>unknown, varied, dynamic, often highly complex</td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>primary goal</td>
<td><strong>full control</strong> over everything</td>
<td>one size fits all / <strong>ease of development</strong> / easy of deployment / ease of use</td>
</tr>
<tr>
<td>system configuration</td>
<td>static / tailored at <strong>compile time</strong></td>
<td>dynamic / discovered at <strong>runtime</strong></td>
</tr>
<tr>
<td>/ driver selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>performance criterion</td>
<td><strong>predictable</strong> worst case</td>
<td><strong>mean</strong> throughput and latency</td>
</tr>
<tr>
<td>resource management</td>
<td><strong>exclusive</strong> reservation / provision on (or near) <strong>worst-case needs</strong></td>
<td><strong>overcommit</strong> / stochastic multiplexing</td>
</tr>
<tr>
<td>workloads</td>
<td><strong>fully known</strong>, mostly static, under control of engineer, often <strong>simple</strong></td>
<td>unknown, varied, dynamic, often <strong>highly complex</strong></td>
</tr>
<tr>
<td>process churn</td>
<td>usually low / nonexistent</td>
<td>can be <strong>extremely high</strong> (e.g., make)</td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------</td>
<td>---------------------------------------------------------</td>
</tr>
<tr>
<td>primary goal</td>
<td>full control over everything</td>
<td>one size fits all / ease of development / ease of use</td>
</tr>
<tr>
<td>system configuration</td>
<td>static / tailored at compile time</td>
<td>dynamic / discovered at runtime</td>
</tr>
<tr>
<td>/ driver selection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>performance criterion</td>
<td>predictable worst case</td>
<td>mean throughput and latency</td>
</tr>
<tr>
<td>resource management</td>
<td>exclusive reservation / provision on (or near)</td>
<td>overcommit / stochastic multiplexing</td>
</tr>
<tr>
<td></td>
<td>worst-case needs</td>
<td></td>
</tr>
<tr>
<td>workloads</td>
<td>fully known, mostly static, under control of</td>
<td>unknown, varied, dynamic, often highly complex</td>
</tr>
<tr>
<td></td>
<td>engineer, often simple</td>
<td></td>
</tr>
<tr>
<td>process churn</td>
<td>usually low / nonexistent</td>
<td>can be extremely high (e.g., make)</td>
</tr>
<tr>
<td>performance</td>
<td>well-understood and explicitly specified →</td>
<td>unknown, implicit, or vague “faster is better” →</td>
</tr>
<tr>
<td>expectations</td>
<td>precise policy</td>
<td>fairness &amp; heuristics</td>
</tr>
<tr>
<td></td>
<td>Real-Time OS</td>
<td>General-Purpose OS</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------</td>
<td>--------------------------------------------------------</td>
</tr>
<tr>
<td>primary goal</td>
<td><strong>full control</strong> over everything</td>
<td>one size fits all / ease of development / ease of use</td>
</tr>
<tr>
<td>system configuration / driver selection</td>
<td>static / tailored at <strong>compile time</strong></td>
<td>dynamic / discovered at <strong>runtime</strong></td>
</tr>
<tr>
<td>performance criterion</td>
<td><strong>predictable</strong> worst case</td>
<td><strong>mean</strong> throughput and latency</td>
</tr>
<tr>
<td>resource management</td>
<td>exclusive reservation / provision on (or near) <strong>worst-case needs</strong></td>
<td><strong>overcommit</strong> / stochastic multiplexing</td>
</tr>
<tr>
<td>workloads</td>
<td><strong>fully known</strong>, mostly static, under control of engineer, often <strong>simple</strong></td>
<td>unknown, varied, dynamic, often <strong>highly complex</strong></td>
</tr>
<tr>
<td>process churn</td>
<td>usually low / nonexistent</td>
<td>can be <strong>extremely high</strong> (e.g., make)</td>
</tr>
<tr>
<td>performance expectations</td>
<td>well-understood and <strong>explicitly specified</strong> ➞ <strong>precise policy</strong></td>
<td>unknown, <strong>implicit</strong>, or vague “faster is better” ➞ <strong>fairness &amp; heuristics</strong></td>
</tr>
<tr>
<td>failure handling</td>
<td><strong>fail safe</strong>: fail early and quickly reboot to known-clean state</td>
<td><strong>best effort</strong>: go on as long as possible (e.g., avoid dataloss)</td>
</tr>
</tbody>
</table>
Why use an RTOS?

- real-time application(s)
- RTOS Abstraction & Management Layer
- hardware platform (e.g., microcontroller or SoC)
Why use an RTOS?

To simplify development
- developer productivity
- focus on application, not low-level system management
Why use an RTOS?

**To simplify development**
- developer productivity
- focus on application, not low-level system management

**To obtain a well-structured system**
- use proper abstractions
- encourage modularity & isolation
- separation of concerns
- maintainability
- certification concerns

---

real-time application(s)

RTOS Abstraction & Management Layer

hardware platform (e.g., microcontroller or SoC)
Why use an RTOS?

To simplify development
- developer productivity
- focus on application, not low-level system management

To obtain a well-structured system
- use proper abstractions
- encourage modularity & isolation
- separation of concerns
- maintainability
- certification concerns

To retain flexibility
- portability & integration
- choose among several vendors
- switch / upgrade hardware platforms
Essential RTOS Facilities

real-time application(s)

res. mgt. API  ...  task abstraction

timers  scheduler  synch.

drivers & “board support package” (BSP)

hardware platform
(e.g., microcontroller or SoC)
Essential RTOS Facilities

Task abstraction & scheduler
- notion of sequential execution
  - process / thread / task
- enables concurrency
- enables system composition / integration
Essential RTOS Facilities

Task abstraction & scheduler
- notion of sequential execution
  - process / thread / task
- enables concurrency
- enables system composition / integration

Synchronization & Communication
- coordinate access to resources
- modularity & separation of concerns
Essential RTOS Facilities

Task abstraction & scheduler
- notion of sequential execution
  - process / thread / task
- enables concurrency
- enables system composition / integration

Synchronization & Communication
- coordinate access to resources
- modularity & separation of concerns

Timers
- trigger action in future
- required for periodic tasks
Essential RTOS Facilities

Task abstraction & scheduler
- notion of sequential execution
  ‣ process / thread / task
- enables concurrency
- enables system composition / integration

Synchronization & Communication
- coordinate access to resources
- modularity & separation of concerns

Timers
- trigger action in future
- required for periodic tasks

Events / Interrupts
- react to environment
- required for sporadic tasks

real-time application(s)

res. mgt. API

... task abstraction

scheduler

drivers & “board support package” (BSP)

hardware platform
(e.g., microcontroller or SoC)
Partial List of RTOSs on Wikipedia

AMOS, AMX RTOS, ARTOS (Locamation), ARTOS (Robotu), Atomthreads, AVIX, BeRTOS, BRTOS, CapROS, ChibiOS/RT, ChorusOS, ChronOS, CMX RTOS, CoActionOS, cocoOS, Concurrent CP/M, Concurrent DOS, Contiki, CooCox CoOS, COS, Data General RDOS, Deos, DioneOS, DNIX, DrRtos, DSOS, DSP/BIOS, DSPnano RTOS, DuinOS, eCos, eCosPro, Embkernel, embOS, Embox, **ERIKA Enterprise**, EROS, EUROs, Femto OS, FlexOS, FreeOSEK, **FreeRTOS**, FunkOS, Fusion RTOS, GEC DOS, HeartOS, Helium, HP-1000/RTE, Hybridthreads, IBM 4680 OS, IBM 4690 OS, INTEGRITY, IntervalZero RTX, INtime, ioRTOS, iRTOS, ISIX, ITRON, µTRON, KolibriOS, Lepton, LithOS, LynxOS, Mark3, MaRTe OS, MAX II, IV, Menuet 64, MenuetOS, MERT, Micron µC/OS-II, Micrion µC/OS-III, Microsoft Invisible Computing (MMLite), Milos, mipOS, mLithOS, MontaVista Linux, MP/M, MQX, Multiuser DOS, Nano-RK, Neutrino, Nokia OS, Nucleus OS, Nut/OS, NuttX, On Time RTOS-32, OpenEPOS, OPENRTOS, OS-9, OS20, OS21, OS4000, OSA, OSE, OSEK, Partikle, PaulOS, Phar Lap ETS, Phoenix-RTOS, picoOS, PICOS18, PikeOS, POK, Portos, PowerTV, Prex, Protothreads, pSOS, Q-Kernel-Free, Q-Kernel-Pro, QNX, QP, ReaGOS, Real-time Linux (CONFIG_RT_PREEMPT), REAL/32, RedHawk Linux, REX OS, RIOT, RMX, RSX-11, RT-11, rt-kernel, RT-Thread, RTAI, **RTEMS**, RTLinux, RTXC Quadros, RX-UX832, RX116, RX616, SafeRTOS, Salvo, SCIOPTA, scmRTOS, SDPOS, SHarK, silRTOS, SimpleAVROS, SINTRAN III, Sirius RTOS, SMX RTOS, SOOS Project, Symbian OS, SYS/BIOS, T-Kernel, Talon DSP RTOS, TargetOS, THEOS, ThreadX, TNKernel, Trampoline Operating System (OSEK and AUTOSAR), Transaction Processing Facility, TRON project, **TUD:OS**, u-velOSity, uKOS, Unison RTOS, UNIX-RTR, uOS, uSmartx, velOSity, VRTX, VxWorks, Windows CE, Xenomai, xPC Target, XRTOS, Y@SOS, µnOS, µTasker … **this list is incomplete!**

(*bold* = better-known open-source RTOSs recommended for self-study)
Why are there so many RTOSs?
Why are there so many RTOSs?

Good reason: because of inherent design tradeoffs
- “real-time computing” comes with diverse requirements
- no single RTOS design fits all needs
Why are there so many RTOSs?

Good reason: because of inherent design tradeoffs
- “real-time computing” comes with diverse requirements
- no single RTOS design fits all needs

Pragmatic reason: because of hardware diversity
- many different embedded hardware platforms in use
- 8-, 16-, 32-, 64-bit architectures
- single- vs multicore
- ARM, PowerPC, MIPS, TriCore, SuperH, …
- no single RTOS supports all platforms
Why are there so many RTOSs?

Good reason: because of inherent design tradeoffs
- “real-time computing” comes with diverse requirements
- no single RTOS design fits all needs

Pragmatic reason: because of hardware diversity
- many different embedded hardware platforms in use
- 8-, 16-, 32-, 64-bit architectures
- single- vs multicore
- ARM, PowerPC, MIPS, TriCore, SuperH, …
- no single RTOS supports all platforms

Some not so good reasons…
- Licensing and royalty issues…
- Not invented here…
- We don’t need an OS…
  ‣ …until you suddenly have a homegrown one.
Spectrum of RTOS Design Goals
Spectrum of RTOS Design Goals

Abstraction

- portability, developer productivity, correctness
Spectrum of RTOS Design Goals

Abstraction
- portability, developer productivity, correctness

Isolation
- fault containment (safety)
- security
- ease of reasoning, debugging & certification
Spectrum of RTOS Design Goals

Abstraction
- portability, developer productivity, correctness

Isolation
- fault containment (safety)
- security
- ease of reasoning, debugging & certification

Predictability
- predictable scheduling policy
- bounded scheduling latency
Spectrum of RTOS Design Goals

**Abstraction**
- portability, developer productivity, correctness

**Isolation**
- fault containment (safety)
- security
- ease of reasoning, debugging & certification

**Predictability**
- predictable scheduling policy
- bounded scheduling latency

**Minimal resource requirements**
- ROM, RAM, MMU, CPU
Spectrum of RTOS Design Goals

**Abstraction**
- portability, developer productivity, correctness

**Isolation**
- fault containment (safety)
- security
- ease of reasoning, debugging & certification

**Predictability**
- predictable scheduling policy
- bounded scheduling latency

**Minimal resource requirements**
- ROM, RAM, MMU, CPU

**Simplicity**
- reliability, testing, certification
Spectrum of RTOS Design Goals

**Abstraction**
- portability, developer productivity, correctness

**Isolation**
- fault containment (safety)
- security
- ease of reasoning, debugging & certification

**Predictability**
- predictable scheduling *policy*
- bounded scheduling *latency*

**Minimal resource requirements**
- ROM, RAM, MMU, CPU

**Simplicity**
- reliability, testing, certification

**higher abstraction & better isolation** = increased RTOS complexity

**reduced complexity** = more predictable, simpler, low requirements

**conflicting goals!**
Abstraction & Portability

```c
struct timespec delay, remainder;

/* sleep for 100ms */
delay.tv_sec = 0;
delay.tv_nsec = 1000000000L;
nanosleep(&delay, &remainder);
```
Abstraction & Portability

```
struct timespec delay, remainder;

/* sleep for 100ms */
delay.tv_sec   = 0;
delay.tv_nsec  = 10000000000L;
nanosleep(&delay, &remainder);
```

“Simply wait for 100ms”

- Which clock sources are available?
- Which of these can generate an interrupt to wake up the task?
- Which is most accurate?
- Can (cycle) counters overflow?
- Does the device need to be set into a specific one-shot mode?
- Which interrupt is reserved for this timer?
- Which time units is the device using? Conversion?
- Is the conversion rate dependent on low-power modes?
- Is there a minimum separation between events?
- Is there a later-revision chipset with slightly changed registers?
Example

Linux’s HPET driver

HPET
high-precision event timer
(ubiquitous on x86)

NMI
non-maskable interrupt
(e.g., watchdog timer)

SMI
service-mode interrupt
(e.g., x86 firmware)

static int hpet_next_event(unsigned long delta,  
 struct clock_event_device *evt, int timer)  
{
    u32 cnt;
    s32 res;

    cnt = hpet_readl(HPET_COUNTER);
    cnt += (u32) delta;
    hpet_writel(cnt, HPET_Tn_CMP(timer));

    /*
     * HPETs are a complete disaster. The compare register is
     * based on a equal comparison and neither provides a less
     * than or equal functionality (which would require to take
     * the wraparound into account) nor a simple count down event
     * mode. Further the write to the comparator register is
     * delayed internally up to two HPET clock cycles in certain
     * chipsets (ATI, ICH9,10). Some newer AMD chipsets have even
     * longer delays. We worked around that by reading back the
     * compare register, but that required another workaround for
     * ICH9,10 chips where the first readout after write can
     * return the old stale value. We already had a minimum
     * programming delta of 5us enforced, but a NMI or SMI hitting
     * between the counter readout and the comparator write can
     * move us behind that point easily. Now instead of reading
     * the compare register back several times, we make the ETIME
     * decision based on the following: Return ETIME if the
     * counter value after the write is less than HPET_MIN_CYCLES
     * away from the event or if the counter is already ahead of
     * the event. The minimum programming delta for the generic
     * clockevents code is set to 1.5 * HPET_MIN_CYCLES.
     */
    res = (s32)(cnt - hpet_readl(HPET_COUNTER));

    return res < HPET_MIN_CYCLES ? -ETIME : 0;
}

Linux: x86/kernel/hpet.c
Isolation & Predictability
Isolation & Predictability

Examples: lack of isolation
- memory **corruption** (dangling pointer,…)
- resource exhaustion (out of memory, file descriptors, task IDs, etc.)
- **WCET overrun**, **interrupt storm** (starvation,…)
- cache pollution
- **unbounded operations** in the kernel (e.g., “**wake up all tasks blocked on a lock**”, where “all” may be controlled by attacker)
- **security considerations** (timing channels, etc.)
Isolation & Predictability

Examples: lack of isolation
- memory corruption (dangling pointer, …)
- resource exhaustion (out of memory, file descriptors, task IDs, etc.)
- WCET overrun, interrupt storm (starvation, …)
- cache pollution
- unbounded operations in the kernel (e.g., “wake up all tasks blocked on a lock”, where “all” may be controlled by attacker)
- security considerations (timing channels, etc.)

Examples: sources of unpredictability
- the OS trying to “optimize” resource use by itself, fairness
  ‣ swapping memory / reallocating memory pages / copy-on-write
  ‣ load-balancing tasks or interrupts
  ‣ power management (frequency scaling, CPU sleep states)
  ‣ batch processing of OS requests (e.g., softirqs in Linux)
- interrupt multiplexing
- timer coalescing (delay timers to reduce #wakeups)
Simplicity & Resource Utilization
Simplicity & Resource Utilization

Simplicity

➡ Does one engineer understand the entire RTOS?
  ‣ Is it well documented?
  ‣ Is it testable?

➡ Always reduce the number of “moving parts”.
  ‣ clever OS code = buggy
  ‣ simple OS code = also buggy, but easier to test and diagnose
Simplicity & Resource Utilization

Simplicity

→ Does one engineer understand the entire RTOS?
  ‣ Is it well documented?
  ‣ Is it testable?

→ Always reduce the number of “moving parts”.
  ‣ clever OS code = buggy
  ‣ simple OS code = also buggy, but easier to test and diagnose

Predictability

→ Is the system a black box? Are there unknown special cases?
→ Is all interaction of components understood?
→ Can the designer anticipate (real-time) performance before testing the system?
Simplicity & Resource Utilization

Simplicity
- Does one engineer understand the entire RTOS?
  ‣ Is it well documented?
  ‣ Is it testable?
- Always reduce the number of “moving parts”.
  ‣ clever OS code = buggy
  ‣ simple OS code = also buggy, but easier to test and diagnose

Predictability
- Is the system a black box? Are there unknown special cases?
- Is all interaction of components understood?
- Can the designer anticipate (real-time) performance before testing the system?

Resource Utilization: “do more with less”
- “Real-time” is trivial if there are no resource constraints
  ‣ simply use a dedicated, fast CPU for each task
- Real systems have Size, Weight, and Power (SWaP) and cost constraints.
Partial List of RTOSs on Wikipedia

AMOS, AMX RTOS, ARTOS (Locamation), ARTOS (Robotu), Atomthreads, AVIX, BeRTOS, BRTOS, CapROS, ChibiOS/RT, ChorusOS, ChronOS, CMX RTOS, CoActionOS, cocoOS, Concurrent CP/M, Concurrent DOS, Contiki, CooCox CoOS, COS, Data General RDOS, Deos, DioneOS, DNIX, DrRtos, DSOS, DSP/BIOS, DSPnano RTOS, DuinOS, eCos, eCosPro, Embkernel, embOS, Embox, ERIKA Enterprise, EROS, EUROs, Femto OS, FlexOS, FreeOSEK, FreeRTOS, FunkOS, Fusion RTOS, GEC DOS, HeartOS, Helium, HP-1000/RTE, Hybridthreads, IBM 4680 OS, IBM 4690 OS, INTEGRITY, IntervalZero RTX, INtime, ioRTOS, iRTOS, ISIX, ITRON, µITRON, KolibriOS, Lepton, LithOS, LynxOS, Mark3, MaRTE OS, MAX II, IV, Menuet 64, MenuetOS, MERT, Micrium µC/OS-II, Micrium µC/OS-III, Microsoft Invisible Computing (MMLite), Milos, mipOS, mLithOS, MontaVista Linux, MP/M, MQX, Multiuser DOS, Nano-RK, Neutrino, Nokia OS, Nucleus OS, Nut/OS, NuttX, On Time RTOS-32, OpenEPOS, OPENRTOS, OS-9, OS20, OS21, OS4000, OSA, OSE, OSEK, Partikle, PaulOS, Phar Lap ETS, Phoenix-RTOS, picoOS, PICOS18, PikeOS, POK, Portos, PowerTV, Prex, Protothreads, pSOS, Q-Kernel-Free, Q-Kernel-Pro, QNX, QP, ReaGOS, Real-time Linux (CONFIG_RT_PREEMPT), REAL/32, RedHawk Linux, REX OS, RIOT, RMX, RSX-11, RT-11, rt-kernel, RT-Thread, RTAI, RTEMS, RTLinux, RTXC Quadros, RX-UX832, RX116, RX616, SafeRTOS, Salvo, SCIOPTA, scmRTOS, SDPOS, SHaRK, silRTOS, SimpleAVROS, SINTRAN III, Sirius RTOS, SMX RTOS, SOOS Project, Symbian OS, SYS/BIOS, T-Kernel, Talon DSP RTOS, TargetOS, THEOS, ThreadX, TNKernel, Trampoline Operating System (OSEK and AUTOSAR), Transaction Processing Facility, TRON project, TUD:OS, u-velOSity, uKOS, Unison RTOS, UNIX-RTR, uOS, uSmartx, velOSity, VRTX, VxWorks, Windows CE, Xenomai, xPC Target, XRTOS, Y@SOS, µnOS, µTasker … this list is incomplete!
(bold = better-known open-source RTOSs recommended for self-study)
RTOS Classes

A way to structure the RTOS landscape.

Given so much variety, the categories are necessarily coarse-grained and subject to exceptions.
RTOS Classes

QNX, VxWorks, INTEGRITY, LynxOS, Linux variants…

“thread library” / “executive” / library RTOS

Static RTOS

UNIX-like RTOS

Separation Kernel

OSEK, AUTOSAR, RTEMS, FreeRTOS…

PikeOS, INTEGRITY Multivisor & 178B, VxWorks MILS, L4 variants…
RTOS Classes

- Static RTOS
  - most **average-complexity** embedded systems
  - used for primarily resource-constrained systems (e.g., automotive)

- UNIX-like RTOS
  - avionics & high-integrity systems, legacy systems integration
  - used if flexibility is required or development costs dominate product costs

- Separation Kernel
  - used if flexibility is required or development costs dominate product costs
RTOS Classes — Isolation

- "thread library" / "executive" / library RTOS
- Static RTOS
- UNIX-like RTOS
- Separation Kernel

Increasing isolation:
- no or only limited isolation possible; trust all tasks / components
- isolation by default; only trust the kernel by default
RTOS Classes — Abstraction

Decreasing complexity & abstraction

- **“thread library” / “executive” / library RTOS**
  - hundreds to thousands of lines of code (LOC)
- **Static RTOS**
- **Separation Kernel**
  - microkernels (1k - 20k LOC)
- **UNIX-like RTOS**
  - full OS (≥100k LOC)
RTOS Classes — Resource Reqs.

- Static RTOS
  - "thread library" / "executive" / library RTOS
  - few hundred bytes to few KBs of RAM, ROM;
    8-bit, 16-bit, 32-bit microcontrollers

- Separation Kernel
  - several MB of RAM, ROM;
    32-bit CPU with MMU

- UNIX-like RTOS
  - resource requirements
Static RTOSs
(& mostly static RTOSs)

OSEK, AUTOSAR, RTEMS, FreeRTOS…
Costs of Dynamic Resource Management
Costs of Dynamic Resource Management

Dynamic resource management
- memory: `malloc()`, page allocator, I/O buffers
- tasks: `create_task()`, `fork()`, etc.
- files: `fopen()`, `fread()`, etc.
- synchronization: locks, barriers, condition variables, etc.
- communication: pipes, sockets, etc.
Costs of Dynamic Resource Management

Dynamic resource management
- memory: `malloc()`, page allocator, I/O buffers
- tasks: `create_task()`, `fork()`, etc.
- files: `fopen()`, `fread()`, etc.
- synchronization: locks, barriers, condition variables, etc.
- communication: pipes, sockets, etc.

Why?
- flexibility and portability, efficiency, developer convenience
Costs of Dynamic Resource Management

Dynamic resource management
- memory: `malloc()`, page allocator, I/O buffers
- tasks: `create_task()`, `fork()`, etc.
- files: `fopen()`, `fread()`, etc.
- synchronization: locks, barriers, condition variables, etc.
- communication: pipes, sockets, etc.

Why?
- flexibility and portability, efficiency, developer convenience

Why not?
- runtime overheads (runtime checks)
- implementation overheads (code size)
- accounting overheads (need to keep track of all resources)
- difficult to test and analyze (error conditions)
Static RTOSs
Static RTOSs

Avoid complexity by avoiding dynamic resource management

- all system resources described in configuration at compile time
  - all tasks, all interrupt handlers
  - all memory, memory segments, static variables
  - all locks, and for each task all accessed locks
  - any other kind of resource
- no malloc(), no heap
- no task creation or deletion, etc.
Static RTOSs

Avoid **complexity** by avoiding **dynamic** resource management

- all system resources described in configuration **at compile time**
  - all tasks, all interrupt handlers
  - all memory, memory segments, static variables
  - all locks, and for each task all accessed locks
  - any other kind of resource

- **no** `malloc()`, no heap
- no task creation or deletion, etc.

Typically compiled into single binary image

- RTOS code and application code form one “program”
- RTOS = source and object files + sophisticated build system
- RTOS vendor provides concrete, **platform-specific implementation** of **portable API** of common building blocks (tasks, timers, locks,…).
Static RTOS Workflow & Structure

- **system config**
  - tasks
  - resources
  - ...

- **app source files**
  - *.c, *.h

- **OS source files**
  - *.c, *.h, *.o

- **config scripts**

- **build system**
  - Makefile

- **_start:**
  - ...

- **_main:**
  - ...

- **_schedule:**
  - ...

- **_task1:**
  - ...

- **binary image for µcontroller**

- **compile & link**

- **flash onto MCU**

**developed by user**

**provided by OS vendor**

**tool vendor**

---

*Image: Introduction to Real-Time Operating Systems by MPI-SWS Brandenburg*
Advantages & Limitations
Advantages & Limitations

Low overheads
- tasks execute on “bare metal”
- system calls are just function calls
Advantages & Limitations

Low overheads
- tasks execute on “bare metal”
- system calls are just function calls

Simplicity & predictability
- no unknown tasks, etc.
- amenable to **static analysis**
- full control over hardware
Advantages & Limitations

Low overheads
- tasks execute on “bare metal”
- system calls are just function calls

Simplicity & predictability
- no unknown tasks, etc.
- amenable to static analysis
- full control over hardware

Well-suited for optimizations
- optimizer & linker can remove unused parts of OS
- function inlining can specialize OS services to actual invoked parameters
- no abstraction or indirection costs
Advantages & Limitations

Low overheads
- tasks execute on “bare metal”
- system calls are just function calls

Simplicity & predictability
- no unknown tasks, etc.
- amenable to static analysis
- full control over hardware

Well-suited for optimizations
- optimizer & linker can remove unused parts of OS
- function inlining can specialize OS services to actual invoked parameters
- no abstraction or indirection costs

Low memory requirements
- hardly any memory used for OS structures (no page tables, etc.)
Advantages & Limitations

Low overheads
- tasks execute on “bare metal”
- system calls are just function calls

Simplicity & predictability
- no unknown tasks, etc.
- amenable to static analysis
- full control over hardware

Well-suited for optimizations
- optimizer & linker can remove unused parts of OS
- function inlining can specialize OS services to actual invoked parameters
- no abstraction or indirection costs

Low memory requirements
- hardly any memory used for OS structures (no page tables, etc.)

Limitations
- must statically declare max. #tasks, #locks, #timers, …
- cannot easily reuse memory
  - but can implement own malloc()
- no isolation whatsoever
Advantages & Limitations

**Low overheads**
- tasks execute on “bare metal”
- system calls are just function calls

**Simplicity & predictability**
- no unknown tasks, etc.
- amenable to static analysis
- full control over hardware

**Well-suited for optimizations**
- optimizer & linker can remove unused parts of OS
- function inlining can specialize OS services to actual invoked parameters
- no abstraction or indirection costs

**Low memory requirements**
- hardly any memory used for OS structures (no page tables, etc.)

**Limitations**
- must statically declare max. #tasks, #locks, #timers, ...
- cannot easily reuse memory
  - but can implement own malloc()
- no isolation whatsoever

**Overall**
- better than using no RTOS at all
- best if only limited resources available and workload is (mostly) static & fully trusted
Example: OSEK/VDX & AUTOSAR
Example: OSEK/VDX & AUTOSAR

Typical static RTOS evolution
- starts as application on bare metal
- OS functionality added over time
- finally, reusable OS functionality extracted from app
- Problem: many competing non-standard APIs
  ‣ similar functionality, similar semantics, but not portable
Example: OSEK/VDX & AUTOSAR

Typical static RTOS evolution
- starts as application on bare metal
- OS functionality added over time
- finally, **reusable OS functionality extracted** from app
- **Problem**: many competing non-standard APIs
  - similar functionality, similar semantics, but **not portable**

Solution: standardize APIs
- **OSEK**: *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*
  - 1993, initiated by BMW, Bosch, DaimlerChrysler, Opel, Siemens, VW
- **VDX**: *Vehicle Distributed eXecutive*
  - 1994, PSA and Renault joined OSEK, bringing VDX
- **AUTOSAR**: *AUTomotive Open Systems Architecture*
  - 2003, incorporates and extends OSEK/VDX
Tasks in OSEK

Tasks
- basic tasks: run to completion (no suspensions)
- extended tasks: may block and wait for async. events

Interrupts
- ISR1 — does not invoke OS services
- ISR2 — does invoke OS services

Scheduling
- preemptive fixed-priority scheduling
- all relevant parameters given in system configuration

API
- entire core OS API spec is less than 90 pages
OSEK Periodic Task Example

How to implement a “Liu & Layland”-like periodic task?
OSEK Periodic Task Example

C source code

```c
TASK(MyPeriodicTask)
{
    [...] /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}
```

configuration

```c
/* Definition of task in config */
TASK MyPeriodicTask
{
    PRIORITY = 1;
    STACKSIZE = 512; /* Stack size */
    [...] 
};
...
/* Definition of My_PeriodicTask timing */
ALARM cyclic_my_periodic_task
{
    [...] 
    ACTION = ACTIVATETASK
    {
        TASK = MyPeriodicTask;
    }; 
    AUTOSTART = TRUE 
    {
        ALARMTIME = 1;
        CYCLETIME = 500; /* Executed every 500msec */
        [...] 
    }; 
};
...
```
**OSEK Periodic Task Example**

```c
TASK(MyPeriodicTask)
{
    [... ] /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}
```

---

/* Definition of task in config */

```c
TASK MyPeriodicTask
{
    PRIORITY = 1;
    STACKSIZE = 512; /* Stack size */
    [...]
};
...
/* Definition of My_PeriodicTask timing */

ALARM cyclic_my_periodic_task
{
    [...]
    ACTION = ACTIVATETASK
    {
        TASK = MyPeriodicTask;
    };
    AUTOSTART = TRUE
    {
        ALARMTIME = 1;
        CYCLETIME = 500; /* Executed every 500msec */
        [...]
    };
};
...
```

---

C source code

configuration

task parameter & resources

adapted from http://lejos-osek.sourceforge.net/
OSEK Periodic Task Example

C source code

```c
TASK(MyPeriodicTask) {
    [...]
    /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}
```

Configuration

```c
/* Definition of task in config */
TASK MyPeriodicTask {
    PRIORITY = 1;
    STACKSIZE = 512; /* Stack size */
    [...]
},
...
/* Definition of My_PeriodicTask timing */
ALARM cyclic_my_periodic_task {
    [...]
    ACTION = ACTIVATETASK
    {
        TASK = MyPeriodicTask;
    },
    AUTOSTART = TRUE
    {
        ALARMTIME = 1;
        CYCLETIME = 500; /* Executed every 500msec */
        [...]
    },
};
...
```

An "alarm" is a resource and must be statically declared.

Task parameter & resources
OSEK Periodic Task Example

```c
TASK(MyPeriodicTask)
{
    [...] /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}
```

C source code

```c
/* Definition of task in config */
TASK MyPeriodicTask
{
    PRIORITY = 1;
    STACKSIZE = 512; /* Stack size */
    [...] 
}
...

/* Definition of My_PeriodicTask timing */
ALARM cyclic_my_periodic_task
{
    [...] 
    ACTION = ACTIVATETASK
    {
        TASK = MyPeriodicTask;
    }
    AUTOSTART = TRUE 
    {
        ALARMTIME = 1;
        CYCLETIME = 500; /* Executed every 500msec */
        [...] 
    }
}
...
```

configuration

- task parameter & resources
- an “alarm” is a resource and must be statically declared
- periodic activation

adapted from http://lejos-osek.sourceforge.net/
OSEK Periodic Task Example

```c
TASK(MyPeriodicTask)
{
    [...]
    /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}
```

**C source code**

**WCET can be measured / statically analyzed.**

```c
/* Definition of task in config */
TASK MyPeriodicTask
{
    PRIORITY = 1;
    STACKSIZE = 512; /* Stack size */
    [...]
}
```

**configuration**

**Period given explicitly.**

```c
/* Definition of My_PeriodicTask timing */
ALARM cyclic_my_periodic_task
{
    [...]
    ACTION = ACTIVATETASK
    {
        TASK = MyPeriodicTask;
    }
    AUTOSTART = TRUE
    {
        ALARMTIME = 1;
        CYCLETIME = 500; /* Executed every 500msec */
    }
    [...]
}
```

**But what about deadline?**

**Deadline unknown to system!**

(can manually implement watchdog…)

adapted from http://lejos-osek.sourceforge.net/
OSEK Sporadic Task Example

How to implement a *sporadic* (= event-triggered) task?

**Event-driven activation mechanisms in OSEK**
- `ActivateTask()`
  - very simple, used in upcoming example
- Event mechanism (= condition variables, semaphores)
  - `SetEvent()`, `WaitEvent()`, `ClearEvent()`
- Alarm-callback routines
  - like an interrupt handler called when user-defined counters exceed predefined thresholds
OSEK Sporadic Task Example

```c
TASK(MySporadicTask)
{
    [...] /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}

ISR(MyInterruptHandler)
{
    [...] /* app-specific actions: read HW state register, etc. */
    ActivateTask(MySporadicTask); /* trigger job release */
}
```

```c
/* Definition of task in config */
TASK MySporadicTask
{
    PRIORITY = 1;
    STACKSIZE = 512;
    AUTOSTART = FALSE;
    ACTIVATION = 2;
    [...] 
};
...
ISR MyInterruptHandler
{
    PRIORITY = 1;
    CATEGORY = 2;
    STACKSIZE = 256;
    [...] 
};
```
OSEK Sporadic Task Example

C source code

```c
TASK(MySporadicTask)
{
    [...] /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}

ISR(MyInterruptHandler)
{
    [...] /* app-specific actions: read HW state register, etc. */
    ActivateTask(MySporadicTask); /* trigger job release */
}
```

/* Definition of task in config */
TASK MySporadicTask
{
    PRIORITY = 1;
    STACKSIZE = 512;
    **AUTOSTART = FALSE;**
    ACTIVATION = 2;
    [...]  
};
...
ISR MyInterruptHandler
{
    PRIORITY = 1;
    CATEGORY = 2;
    STACKSIZE = 256;
    [...]  
};
```

configuration

- task initially inactive
- up to two activations are buffered
### OSEK Sporadic Task Example

```c
TASK(MySporadicTask)
{
    [...] /* app-specific actions carried out by one job */
    TerminateTask(); /* indicate job completion */
}

ISR(MyInterruptHandler)
{
    [...] /* app-specific actions: read HW state register, etc. */
    ActivateTask(MySporadicTask); /* trigger job release */
}
```

/* Definition of task in config */
```c
TASK MySporadicTask
{
    PRIORITY = 1;
    STACKSIZE = 512;
    AUTOSTART = FALSE;
    ACTIVATION = 2;
    [...] 
}
```

...  
```c
ISR MyInterruptHandler
{
    PRIORITY = 1;
    CATEGORY = 2;
    STACKSIZE = 256;
    [...] 
}
```

- **C source code**
  - `TerminateTask();` for indicating job completion
  - `ActivateTask(MySporadicTask);` for triggering job release

- **Configuration**
  - `AUTOSTART = FALSE;` means task is initially inactive
  - Up to two activations are buffered
  - ISR priority space separate (always higher than tasks)
  - Only category-two ISRs may call `ActivateTask()`
Resource Sharing in OSEK

How to implement a critical section?

Mutual Exclusion in OSEK

- Implements the OSEK [Priority] Ceiling Protocol (PCP)
  - Task raises own priority to ceiling priority upon resource acquisition.
  - Hence no task accessing same resource can be scheduled.
  - This is equivalent to the Stack Resource Policy (SRP)...

- Ceilings determined automatically from configuration.
Resource Sharing in OSEK

```c
TASK(MyTask)
{
    /* actions carried out by one job */
    GetResource(resource1);
    [...] /* critical section related to resource1, priority raised */
    ReleaseResource(resource1);
    TerminateTask(); /* indicate job completion */
}
```

```c
/* Definition of MyTask*/
TASK MyTask
{
    RESOURCE = resource1;
    [...]  
};
...
/* Definition of resource */
RESOURCE resource1
{
    RESOURCEPROPERTY = STANDARD;
};
...  
```

adapted from http://lejos-osek.sourceforge.net/
POSIX and POSIX-like RTOSs

QNX, VxWorks, INTEGRITY, LynxOS, Linux variants…
POSIX Real-Time Profiles

What does POSIX-compatible mean?
POSIX Real-Time Profiles

What does POSIX-compatible mean?

PSE51 — “minimal”
- single “process”, no MMU required
- no disk, no file system
- similar to OSEK in breadth
POSIX Real-Time Profiles

What does POSIX-compatible mean?

PSE51 — “minimal”
- single “process”, no MMU required
- no disk, no file system
- similar to OSEK in breadth

PSE52 — “controller”
- single process, no MMU required
- simplified file system
- message queues & tracing support
POSIX Real-Time Profiles

What does POSIX-compatible mean?

PSE51 — “minimal”
- single “process”, no MMU required
- no disk, no file system
- similar to OSEK in breadth

PSE52 — “controller”
- single process, no MMU required
- simplified file system
- message queues & tracing support

PSE53 — “dedicated”
- multiple processes, MMU needed
- single user, no shell
- networking, full asynchronous I/O stack
POSIX Real-Time Profiles

What does POSIX-compatible mean?

PSE51 — “minimal”
- single “process”, no MMU required
- no disk, no file system
- similar to OSEK in breadth

PSE52 — “controller”
- single process, no MMU required
- simplified file system
- message queues & tracing support

PSE53 — “dedicated”
- multiple processes, MMU needed
- single user, no shell
- networking, full asynchronous I/O stack

PSE54 — “multi-purpose”
- almost full UNIX
- shell, multi user, security
- full file system

What does POSIX-compatible mean?
POSIX Real-Time Profiles

What does POSIX-compatible mean?

PSE51 — “minimal”
- single “process”, no MMU required
- no disk, no file system
- similar to OSEK in breadth

PSE52 — “controller”
- single process, no MMU required
- simplified file system
- message queues & tracing support

PSE53 — “dedicated”
- multiple processes, MMU needed
- single user, no shell
- networking, full asynchronous I/O stack

PSE54 — “multi-purpose”
- almost full UNIX
- shell, multi user, security
- full file system

Most convenient, just like any other UNIX/POSIX development.
Real-time Linux, QNX, …
Real-Time Profiles

What does POSIX-compatible mean?

PSE51 — “minimal”
- single “process”, no MMU required
- no disk, no file system
- similar to OSEK in breadth

PSE52 — “controller”
- single process, no MMU required
- simplified file system
- message queues & tracing support

PSE53 — “dedicated”
- multiple processes, MMU needed
- single user, no shell
- networking, full asynchronous I/O stack

PSE54 — “multi-purpose”
- almost full UNIX
- shell, multi user, security
- full file system
POSIX Scheduling
POSIX Scheduling

POSIX knows three scheduling policies
- implementations may add additional non-standard policies
POSIX Scheduling

POSIX knows three scheduling policies
- implementations may add additional non-standard policies

SCHED_FIFO
- preemptive fixed-priority scheduling
- FIFO among ready tasks of equal priority
- this matches classic response-time analysis
POSIX Scheduling

POSIX knows three scheduling policies
- implementations may add additional non-standard policies

SCHED_FIFO
- preemptive fixed-priority scheduling
- FIFO among ready tasks of equal priority
- this matches classic response-time analysis

SCHED_RR
- preemptive fixed-priority scheduling
- round-robin among ready tasks of equal priority
- not particularly useful from analysis POV
POSIX Scheduling

POSIX knows three scheduling policies
» implementations may add additional non-standard policies

SCHED_FIFO
» preemptive fixed-priority scheduling
» FIFO among ready tasks of equal priority
» this matches classic response-time analysis

SCHED_RR
» preemptive fixed-priority scheduling
» round-robin among ready tasks of equal priority
» not particularly useful from analysis POV

SCHED_OTHER
» unknown policy timesharing for non-real-time tasks
» in Linux, this is CFS (“fair” scheduling)
POSIX Scheduling

POSIX knows three scheduling policies
- implementations may add additional non-standard policies

**SCHED_FIFO**
- preemptive fixed-priority scheduling
- FIFO among ready tasks of equal priority
- this matches classic **response-time analysis**

**SCHED_RR**
- preemptive fixed-priority scheduling
- round-robin among ready tasks of equal priority
- not particularly useful from analysis POV

**SCHED_OTHER**
- unknown policy timesharing for **non-real-time tasks**
- in Linux, this is CFS ("fair" scheduling)

---

Use this to implement rate-monotonic / deadline-monotonic scheduling.

**Lowest priority:**
- `sched_get_priority_min()`
  [Linux: 1]

**Highest priority:**
- `sched_get_priority_max()`
  [Linux: 99]

POSIX mandates at least 32 priorities.
**POSIX Periodic Task Example**

*How to implement a “Liu & Layland”-like periodic task?*

**POSIX: lower-level than OSEK**
- no ready-made task & periodic alarm abstractions
- need to realize periodic activation with sleep call

**Powerful, but difficult API**
- many options and alternatives, not minimal API
- almost anything is possible
- but it’s easy to inadvertently create **timing bugs**
  ‣ or other issues such as race conditions (avoid signals!)
POSIX Periodic Task, Attempt #1

```c
#define PERIOD_IN_MILLIS 100
#define PERIOD_IN_NANOS (PERIOD_IN_MILLIS * 1000000)

void job(void)
{
    [...] /* periodic activity */
}

void periodic_task(void)
{
    struct timespec delay, remainder;
    int err;

    while (1) /* forever release jobs */
    {
        /* run job */
        job();

        /* sleep till next job release */
        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

Is this correct?
POSIX Periodic Task, Attempt #1

```c
#define PERIOD_IN_MILLIS 100
#define PERIOD_IN_NANOS (PERIOD_IN_MILLIS * 1000000)

void job(void)
{
    /* periodic activity */
}

void periodic_task(void)
{
    struct timespec delay, remainder;
    int err;

    while (1) /* forever release jobs */
    {
        /* run job */
        job();

        /* sleep till next job release */
        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

Is this correct?
POSIX Periodic Task, Attempt #1

```c
#define PERIOD_IN_MILLIS 100
#define PERIOD_IN_NANOS (PERIOD_IN_MILLIS * 1000000)

void job(void)
{
    [...] /* periodic activity */
}

void periodic_task(void)
{
    struct timespec delay, remainder;
    int err;

    while (1) /* forever release jobs */
    {
        /* run job */
        job();

        /* sleep till next job release */
        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

Is this correct?
POSIX Periodic Task, Attempt #1

C source code

```c
#define PERIOD_IN_MILLIS 100
#define PERIOD_IN_NANOS (PERIOD_IN_MILLIS * 1000000)

void job(void)
{
    [...] /* periodic activity */
}

void periodic_task(void)
{
    struct timespec delay, remainder;
    int err;

    while (1) /* forever release jobs */
    {
        /* run job */
        job();

        /* sleep till next job release */
        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

No. Effectively wrong period!

Invokes job every \((\text{response time} + \text{PERIOD})\) milliseconds because actual response time is ignored.

Is this correct?
POSIX Periodic Task, Attempt #2

What if we *measure the response time* and adjust the *sleep duration*?
```c
void periodic_task(void)
{
    struct timespec delay, remainder;
    struct timespec release_time, response_time;
    int err;

    while (1) /* forever release jobs */
    {
        /* record release time */
        err = clock_gettime(CLOCK_MONOTONIC, &release_time);
        assert(err == 0);

        job(); /* run job */

        /* record finish time */
        err = clock_gettime(CLOCK_MONOTONIC, &response_time);
        assert(err == 0);

        /* compute delta */
        timespec_subtract(&response_time, &release_time);

        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        /* adjust sleep to account for response time */
        timespec_subtract(&delay, &response_time);

        /* sleep till next job release */
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

*Is this correct?*
POSIX Periodic Task, Attempt #2

```c
void periodic_task(void)
{
    struct timespec delay, remainder;
    struct timespec release_time, response_time;
    int err;

    while (1) /* forever release jobs */
    {
        /* record release time */
        err = clock_gettime(CLOCK_MONOTONIC, &release_time);
        assert(err == 0);

        job(); /* run job */

        /* record finish time */
        err = clock_gettime(CLOCK_MONOTONIC, &response_time);
        assert(err == 0);

        /* compute delta */
        timespec_subtract(&response_time, &release_time);

        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        /* adjust sleep to account for response time */
        timespec_subtract(&delay, &response_time);

        /* sleep till next job release */
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

Is this correct?
void periodic_task(void)
{
    struct timespec delay, remainder;
    struct timespec release_time, response_time;
    int err;

    while (1) /* forever release jobs */
    {
        /* record release time */
        err = clock_gettime(CLOCK_MONOTONIC, &release_time);
        assert(err == 0);
        job(); /* run job */

        /* record finish time */
        err = clock_gettime(CLOCK_MONOTONIC, &response_time);
        assert(err == 0);

        /* compute delta */
        timespec_subtract(&response_time, &release_time);

        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        /* adjust sleep to account for response time */
        timespec_subtract(&delay, &response_time);

        /* sleep till next job release */
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}

Is this correct?
POSIX Periodic Task, Attempt #2

```c
void periodic_task(void)
{
    struct timespec delay, remainder;
    struct timespec release_time, response_time;
    int err;

    while (1) /* forever release jobs */
    {
        /* record release time */
        err = clock_gettime(CLOCK_MONOTONIC, &release_time);
        assert(err == 0);
        job(); /* run job */

        /* record finish time */
        err = clock_gettime(CLOCK_MONOTONIC, &response_time);
        assert(err == 0);

        /* compute delta */
        timespec_subtract(&response_time, &release_time);

        delay.tv_sec = 0;
        delay.tv_nsec = PERIOD_IN_NANOS;
        /* adjust sleep to account for response time */
        timespec_subtract(&delay, &response_time);

        /* sleep till next job release */
        do {
            err = nanosleep(&delay, &remainder);
            delay = remainder;
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

No. Ignores drift due to potential preemptions!

Process could be preempted just before the first measurement, or any time after the second measurement, which the adjustment would not account for.

Is this correct?
POSIX Periodic Task, Attempt #3

We need to express the desired activation time in *absolute terms*!
Introduction to Real-Time Operating Systems

POSIX Periodic Task, Attempt #3

```c
void periodic_task(void)
{
    struct timespec period, release_time;
    int err;

    period.tv_sec = 0;
    period.tv_nsec = PERIOD_IN_NANOS;

    /* record first release time */
    err = clock_gettime(CLOCK_MONOTONIC, &release_time);
    assert(err == 0);

    while (1) /* forever release jobs */
    {
        job(); /* run job */

        /* compute next release time */
        timespec_add(&release_time, &period);

        do {
            /* sleep to exact absolute release time */
            err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                                  &release_time, NULL);
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```
```c
void periodic_task(void)
{
    struct timespec period, release_time;
    int err;

    period.tv_sec = 0;
    period.tv_nsec = PERIOD_IN_NANOS;

    /* record first release time */
    err = clock_gettime(CLOCK_MONOTONIC, &release_time);
    assert(err == 0);

    while (1) /* forever release jobs */
    {
        job(); /* run job */

        /* compute next release time */
        timespec_add(&release_time, &period);

        do {
            /* sleep to exact absolute release time */
            err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                                  &release_time, NULL);
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```
void periodic_task(void)
{
    struct timespec period, release_time;
    int err;

    period.tv_sec = 0;
    period.tv_nsec = PERIOD_IN_NANOS;

    /* record first release time */
    err = clock_gettime(CLOCK_MONOTONIC, &release_time);
    assert(err == 0);

    while (1) /* forever release jobs */
    {
        job(); /* run job */

        /* compute next release time */
        timespec_add(&release_time, &period);
        do {
            /* sleep to exact absolute release time */
            err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                                  &release_time, NULL);
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
POSIX Periodic Task, Attempt #3

```c
void periodic_task(void)
{
    struct timespec period, release_time;
    int err;

    period.tv_sec = 0;
    period.tv_nsec = PERIOD_IN_NANOS;

    /* record first release time */
    err = clock_gettime(CLOCK_MONOTONIC, &release_time);
    assert(err == 0);

    while (1) /* forever release jobs */
    {
        job(); /* run job */

        /* compute next release time */
        timespec_add(&release_time, &period);

        do {
            /* sleep to exact absolute release time */
            err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                                  &release_time, NULL);
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```
POSIX Periodic Task

```c
void periodic_task(void)
{
    struct timespec period, release_time;
    int err;
    period.tv_sec = 0;
    period.tv_nsec = PERIOD_IN_NANOS;

    /* record first release time */
    err = clock_gettime(CLOCK_MONOTONIC, &release_time);
    assert(err == 0);

    while (1) /* forever release jobs */
    {
        job(); /* run job */
        /* compute next release time */
        timespec_add(&release_time, &period);
        do {
            /* sleep to exact absolute release time */
            err = clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                                   &release_time, NULL);
            /* if we were interrupted by a signal, keep sleeping */
        } while (err != 0 && errno == EINTR);
    }
}
```

This pattern works, but also exemplifies the **challenge** with POSIX: must find just **the right interface/version/parameters** and deal with **error codes**.
POSIX Sporadic Task Example

How to implement a sporadic (\textit{=} event-triggered) task?
POSIX Sporadic Task Example

How to implement a sporadic (= event-triggered) task?

How are events communicated in POSIX?
POSIX Sporadic Task Example

How to implement a sporadic (= event-triggered) task?

How are events communicated in POSIX?

Via file descriptors:
- datagram socket (UDP, CAN, …)
- stream socket (pipe, TCP w/o Nagle’s algorithm)
- device files (GPIOs, serial ports, various buses…)
POSIX Sporadic Task Example

How to implement a sporadic (= event-triggered) task?

How are events communicated in POSIX?

Via file descriptors:
- datagram socket (UDP, CAN, …)
- stream socket (pipe, TCP w/o Nagle’s algorithm)
- device files (GPIOs, serial ports, various buses…)

→ wait for events with select()
**POSIX Sporadic Task Example**

```c
void job(int event_source_fd)
{
    [...] /* consume input from event_source_fd */
}

void sporadic_task(int event_source_fd)
{
    int err;
    fd_set wait_for_fds;

    while (1) /* forever release jobs */
    {
        FD_ZERO(&wait_for_fds);
        FD_SET(event_source_fd, &wait_for_fds);

        /* wait until some data becomes available on the file descriptor */
        err = select(event_source_fd + 1, &wait_for_fds, NULL, NULL, NULL);
        assert(err >= 0 || errno == EINTR);

        if (err == 1 && FD_ISSET(event_source_fd, &wait_for_fds))
            job(event_source_fd); /* run job */
    }
}
```
POSIX Sporadic Task Example

```c
void job(int event_source_fd)
{
    [...] /* consume input from event_source_fd */
}

void sporadic_task(int event_source_fd)
{
    int err;
    fd_set wait_for_fds;
    while (1) /* forever release jobs */
    {
        FD_ZERO(&wait_for_fds);
        FD_SET(event_source_fd, &wait_for_fds);
        /* wait until some data becomes available on the file descriptor */
        err = select(event_source_fd + 1, &wait_for_fds, NULL, NULL, NULL);
        assert(err >= 0 || errno == EINTR);
        if (err == 1 && FD_ISSET(event_source_fd, &wait_for_fds))
            job(event_source_fd); /* run job */
    }
}
```
POSIX Sporadic Task Example

```c
void job(int event_source_fd)
{
  [...] /* consume input from event_source_fd */
}

void sporadic_task(int event_source_fd)
{
  int err;
  fd_set wait_for_fds;

  while (1) /* forever release jobs */
  {
    FD_ZERO(&wait_for_fds);
    FD_SET(event_source_fd, &wait_for_fds);

    /* wait until some data becomes available on the file descriptor */
    err = select(event_source_fd + 1, &wait_for_fds, NULL, NULL, NULL);
    assert(err >= 0 || errno == EINTR);

    if (err == 1 && FD_ISSET(event_source_fd, &wait_for_fds))
      job(event_source_fd); /* run job */
  }
}
```

tell kernel we want to wait until something happens to event_source_fd

wait for event
POSIX Sporadic Task Example

```c
void job(int event_source_fd)
{
    [...] /* consume input from event_source_fd */
}

void sporadic_task(int event_source_fd)
{
    int err;
    fd_set wait_for_fds;
    while (1) /* forever release jobs */
    {
        FD_ZERO(&wait_for_fds);
        FD_SET(event_source_fd, &wait_for_fds);
        /* wait until some data becomes available on the file descriptor */
        err = select(event_source_fd + 1, &wait_for_fds, NULL, NULL, NULL);
        assert(err >= 0 || errno == EINTR);
        if (err == 1 && FD_ISSET(event_source_fd, &wait_for_fds))
            job(event_source_fd); /* run job */
    }
}
```

tell kernel we want to wait until something happens to event_source_fd
wait for event
invoke job if data (= event info) is present
POSIX Sporadic Task Example

```c
void job(int event_source_fd)
{
  [...] /* consume input from event_source_fd */
}

void sporadic_task(int event_source_fd)
{
  int err;
  fd_set wait_for_fds;

  while (1) /* forever release jobs */
  {
    FD_ZERO(&wait_for_fds);
    FD_SET(event_source_fd, &wait_for_fds);

    /* wait until some data becomes available on the file descriptor */
    err = select(event_source_fd + 1, &wait_for_fds, NULL, NULL, NULL);
    assert(err >= 0 || errno == EINTR);

    if (err == 1 && FD_ISSET(event_source_fd, &wait_for_fds))
      job(event_source_fd); /* run job */
  }
}
```

- job must consume (= read) all data related to one invocation (e.g., one packet)
- tell kernel we want to wait until something happens to event_source_fd
- wait for event
- invoke job if data (= event info) is present
Resource Sharing in POSIX
Resource Sharing in POSIX

POSIX pthreads supports two uniprocessor real-time locking protocols

→ Note: by default, none is in use!
Resource Sharing in POSIX

POSIX pthreads supports two uniprocessor real-time locking protocols

- Note: by default, none is in use!

PTHREAD_PRIO_INHERIT
- classic (basic) priority inheritance protocol
- from a schedulability POV, not as good as ceiling-based protocols
- provides no (or very limited) benefit on multiprocessors if tasks have restricted processor affinities
Resource Sharing in POSIX

POSIX pthreads supports two uniprocessor real-time locking protocols

- Note: by default, none is in use!

**PTHREAD_PRIO_INHERIT**

- classic (basic) *priority inheritance* protocol
- from a schedulability POV, not as good as ceiling-based protocols
- provides no (or very limited) benefit on *multiprocessors* if tasks have restricted processor affinities

**PTHREAD_PRIO_PROTECT**

- equivalent to the OSEK **PCP** variant
- works by raising thread priority to ceiling priority upon lock acquisition
- provides no (or very limited) benefit on *multiprocessors*!
Resource Sharing in POSIX

POSIX pthreads supports two uniprocessor real-time locking protocols

- Note: by default, none is in use!

PTHREAD_PRIO_INHERIT
- classic (basic) priority inheritance protocol
- from a schedulability POV, not as good as ceiling-based protocols
- provides no (or very limited) benefit on multiprocessors if tasks have restricted processor

PTHREAD_PRIO_PROTECT
- equivalent to the OSEK PCP variant
- works by raising thread priority to ceiling priority upon lock acquisition
- provides no (or very limited) benefit on multiprocessors!

Must configure priority ceiling manually.
pthread_mutex_setprioceiling()
Resource Sharing in POSIX

POSIX pthreads supports two uniprocessor real-time locking protocols

- Note: by default, none is in use!

PTHREAD_PRIO_INHERIT
- classic (basic) priority inheritance protocol
- from a schedulability POV, not as good as ceiling-based protocols
- provides no (or very limited) benefit on multiprocessors if tasks have restricted processor affinities

On Linux, priority inheritance has much lower average-case overheads.

- This is a Linux implementation artifact ("Futex"), PCP can be made much faster.

PTHREAD_PRIO_PROTECT
- equivalent to the OSEK PCP variant
- works by raising thread priority to ceiling priority upon lock acquisition
- provides no (or very limited) benefit on multiprocessors!
Summary
Summary

Do use an RTOS (rather than running on “bare metal”)
- lower development costs, more flexibility, better maintainability, better reliability
Summary

Do use an RTOS (rather than running on “bare metal”)
- lower development costs, more flexibility, better maintainability, better reliability

RTOSes come in many shapes and sizes
- resource footprint vs. isolation vs. analyzability
- pick the right one depending on application needs
Summary

Do use an **RTOS** (rather than running on “bare metal”) → lower development costs, more flexibility, better maintainability, better reliability

RTOSes come in many shapes and sizes → **resource footprint** vs. **isolation** vs. **analyzability**
→ pick the right one depending on application needs

Structure applications based on **well-understood, analyzable concepts** from the real-time literature
→ examples: periodic and sporadic tasks in OSEK and POSIX
→ typical RTOS provides many more APIs than what is needed → must carefully reason about semantics
Björn Brandenburg
Real-Time Systems Group @ MPI-SWS
https://www.mpi-sws.org/~bbb/

Linux Testbed for Multiprocessor Scheduling in Real-Time Systems

www.litmus-rt.org

Linux-based **Multiprocessor RTOS** developed at MPI-SWS.

If you have strong coding or analytical skills, contact me for thesis opportunities and student research assistantships (“HiWi” jobs).