Lecture 12: Embedded Operating Systems

Michael O'Boyle
Embedded Software

Overview

- General requirements
  - Configuration
  - Device Drivers
- Real time operating systems
  - Timing
  - Scheduling
  - Performance
- Examples
- Conclusion

Reuse and Configurability

Knowledge from previous designs to be made available in the form of **intellectual property** (IP, for SW & HW)

- Operating systems
- Middleware

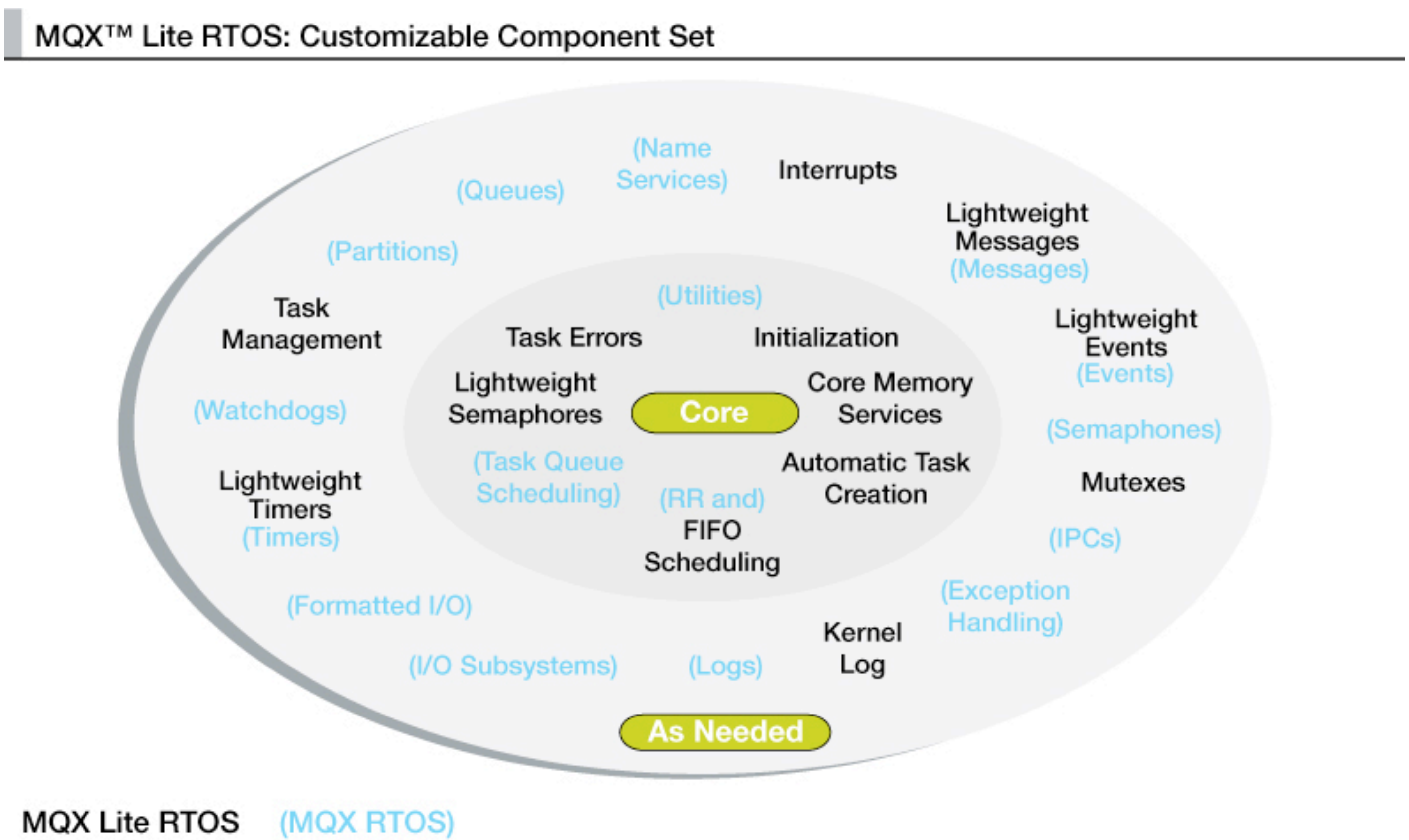
Configurability
No single OS will fit all needs, no overhead for unused functions tolerated ☞ configurability needed.

- Simplest form: remove unused functions (by linker ?).
- Conditional compilation (using #if and #ifdef commands).
- Dynamic data might be replaced by static data.
- Advanced compile-time evaluation useful

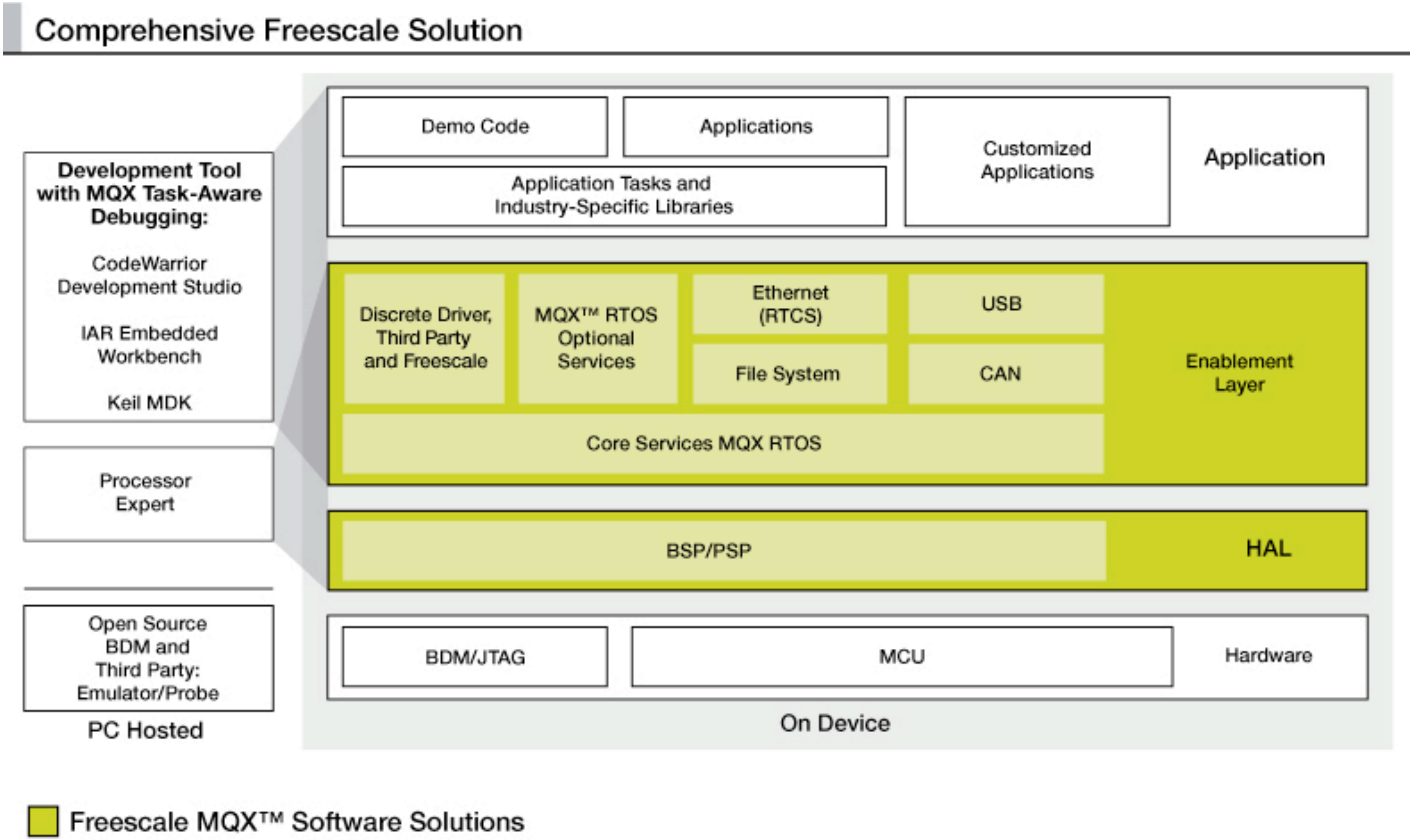
Verification a potential problem -:

- Each derived OS must be tested thoroughly; open source RTOS from Red Hat 100 to 200 configs

MQX -configurable



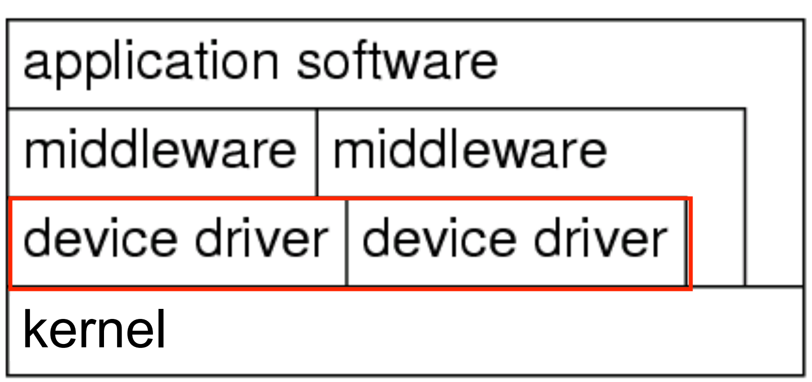
Sits directly on hardware



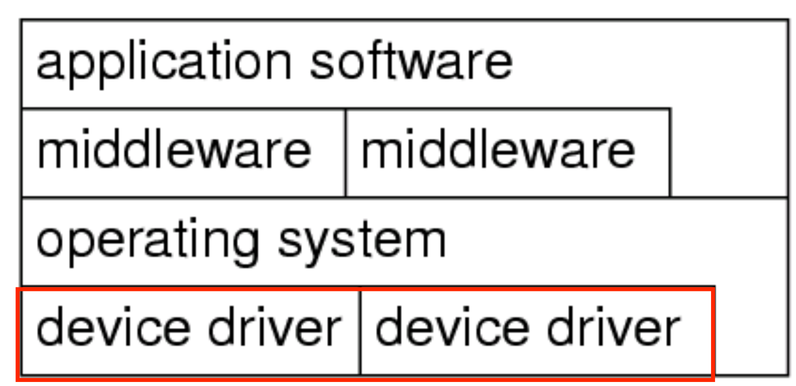
Disc and Network handled by tasks not OS

- **Effectively no device that needs to be supported by all variants of the OS**, except maybe the system timer.
- Many ES without disc, a keyboard, a screen or a mouse.
- Disc & network handled by tasks instead of integrated drivers. Discs & networks can be handled by tasks.
- Otherwise impossible number to support : too expensive

Embedded OS



Standard OS



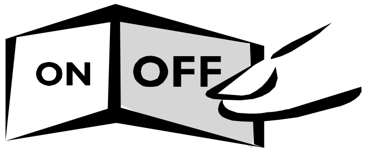
Protection

Protection mechanisms not always necessary:
ES typically designed for a single purpose,
untested programs rarely loaded, SW considered reliable.

Privileged I/O instructions not necessary and
tasks can do their own I/O.

Example: Let **`switch`** be the address of some switch
Simply use

```
load register,switch
```



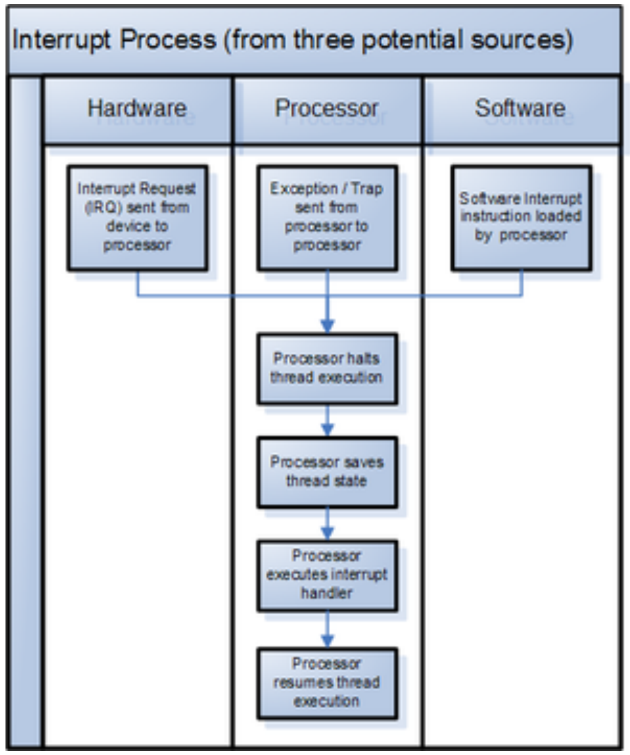
instead of OS call.

However, protection mechanisms may be needed for safety
and security reasons.

Interrupts

Interrupts can be employed by any process
For standard OS: serious source of unreliability.
However

- embedded programs can be considered to be tested,
- since protection is not necessary and
- since efficient control over a variety of devices is required



It is possible to let interrupts directly start or stop tasks

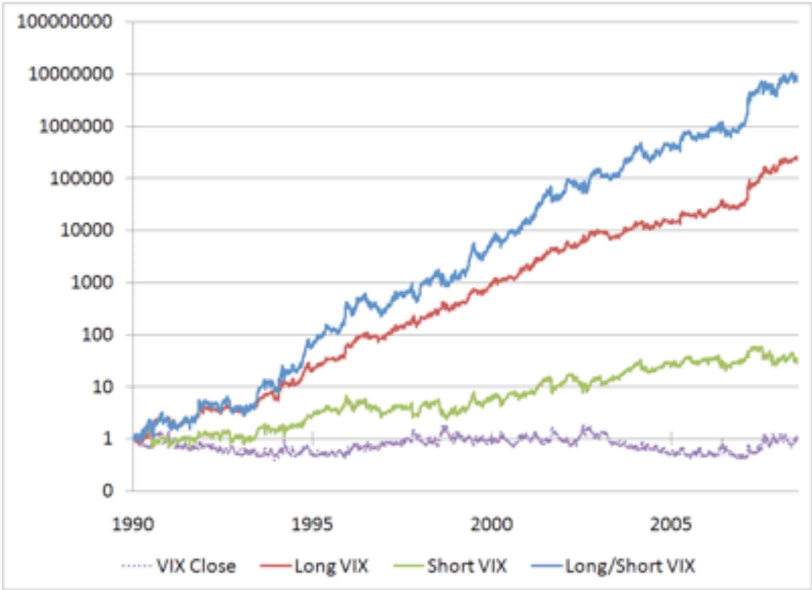
- (by storing the task's start address in the interrupt table).
- More efficient than going through OS services.
- Reduced composability: if a task is connected to an interrupt, it may be difficult to add another task which also needs to be started by an event.

Real time requirements

Def.: *(A) real-time operating system is an operating system that supports the construction of real-time systems.*

The following are the three key requirements

1. **The timing behaviour of the OS must be predictable.**
   All services of the OS: Upper bound on the execution time!
   RTOSs must be timing-predictable:
   - short times during which interrupts are disabled,
   - (for hard disks:) contiguous files to avoid unpredictable head movements.



Timing

2. OS should manage the timing and scheduling

- OS possibly has to be aware of task deadlines;
  (unless scheduling is done off-line).
- Frequently, the OS should provide precise time services with high resolution.

Time plays a central role in “real-time” systems.

Actual time is described by real numbers.

Two discrete standards are used in real-time equipment:

- **International atomic time TAI**
  (french: *temps atomic internationale*)
  Free of any artificial artifacts.
- **Universal Time Coordinated (UTC)**
  UTC is defined by astronomical standards

UTC and TAI identical on Jan. 1st, 1958.

35 seconds had to be added since then.

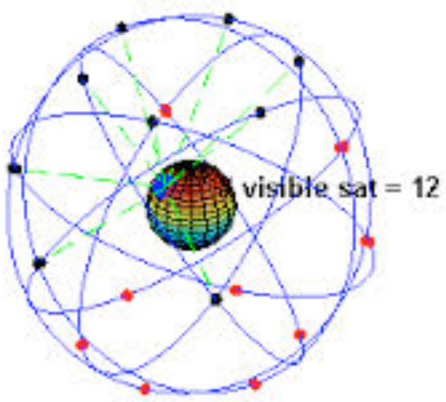
Not without problems: New Year may start twice per night.

Synchronisation: Internal vs External

- Synchronization with one master clock
  - Typically used in startup-phases
  - Distributed synchronization:
  - Collect information from neighbours
    1. Compute correction value
    2. Set correction value

    Precision of step 1 depends on how information is collected:

    - Application level: ~500 µs to 5 ms
    - Operation system kernel: 10 µs to 100 µs
    - Communication hardware: < 10 µs

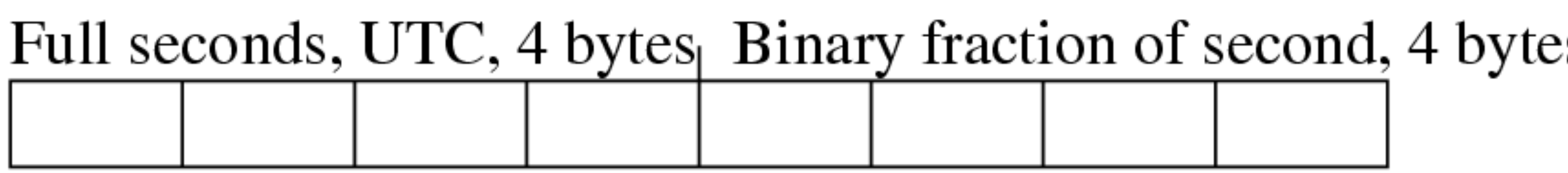


External synchronization guarantees consistency with actual physical time.

- Trend is to use GPS for ext. synchronization
- GPS offers TAI and UTC time information.
- Resolution is about 100 ns.

External Timing

- Problematic from the perspective of fault tolerance:
- Erroneous values are copied to all stations.
- Consequence: Accepting only small changes to local time.
- Many time formats too restricted;
  e.g.: Network Time protocol NTP protocol includes only years up to 2036



Range up the years 2036; 136 year wrap around cycle

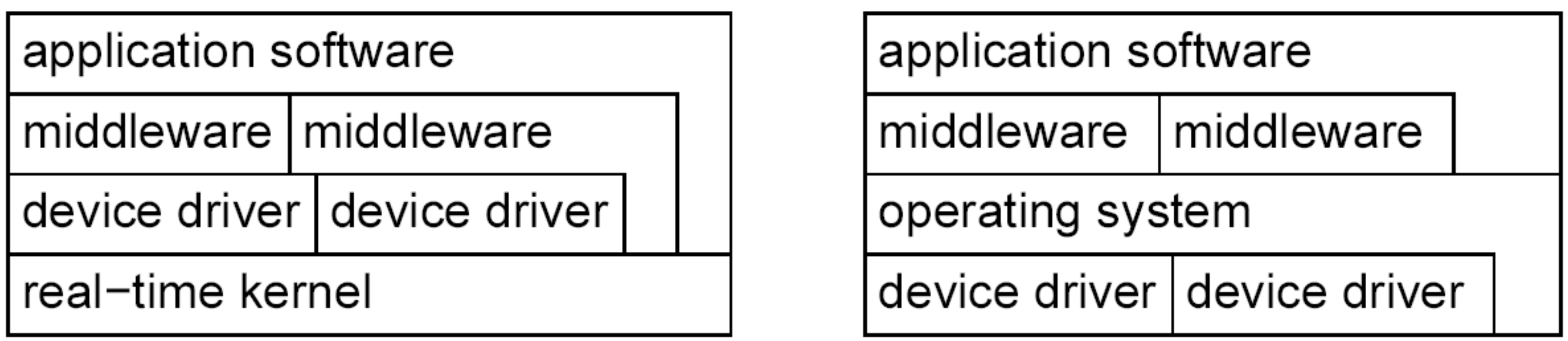
For time services and global synchronization of clocks synchronization see Kopetz, 1997.

OS must be fast: RTOS kernels

3. **The OS must be fast**
   Practically important.

Distinction between

- real-time kernels and modified kernels of standard OSes.



Distinction between

- general RTOSs and RTOSs for specific domains,
- standard APIs (e.g. POSIX RT-Extension of Unix, ITRON, OSEK) or proprietary APIs.

Functionality

Includes

- processor management,
- memory management,
- and timer management;
- task management (resume, wait etc),
- inter-task communication and synchronization.

3 Classes

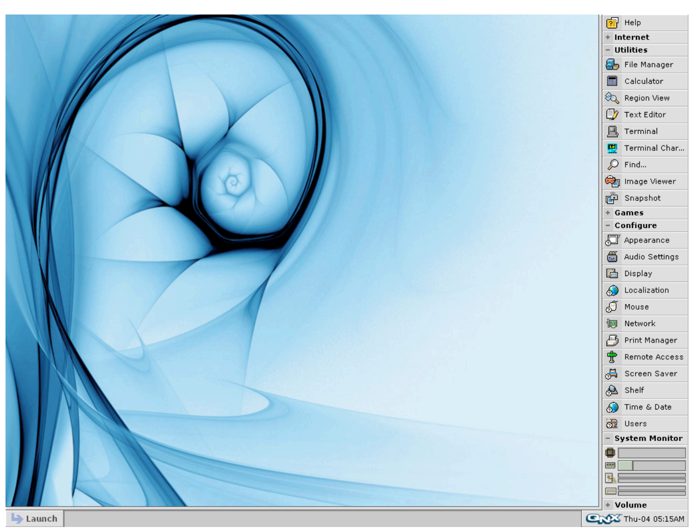
- Fast proprietary kernels
- RT extensions
- Research approaches

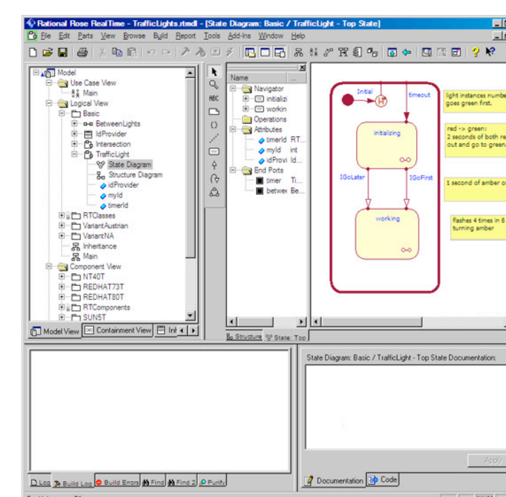
Classes of RTOSes: 1. Fast proprietary kernels

For complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect

[R. Gupta, UCI/UCSD]

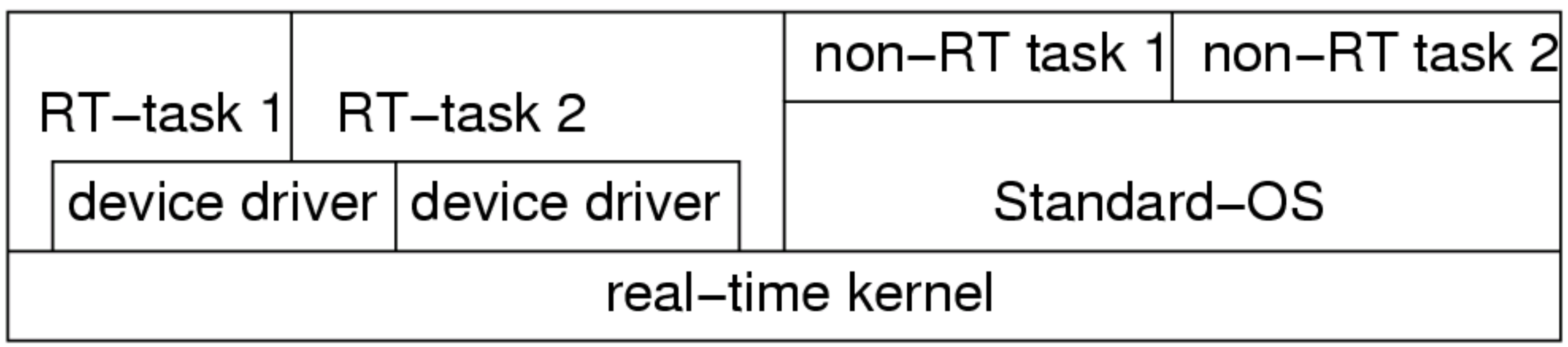
Examples include
QNX, PDOS, VCOS, VTRX32, VxWORKS.





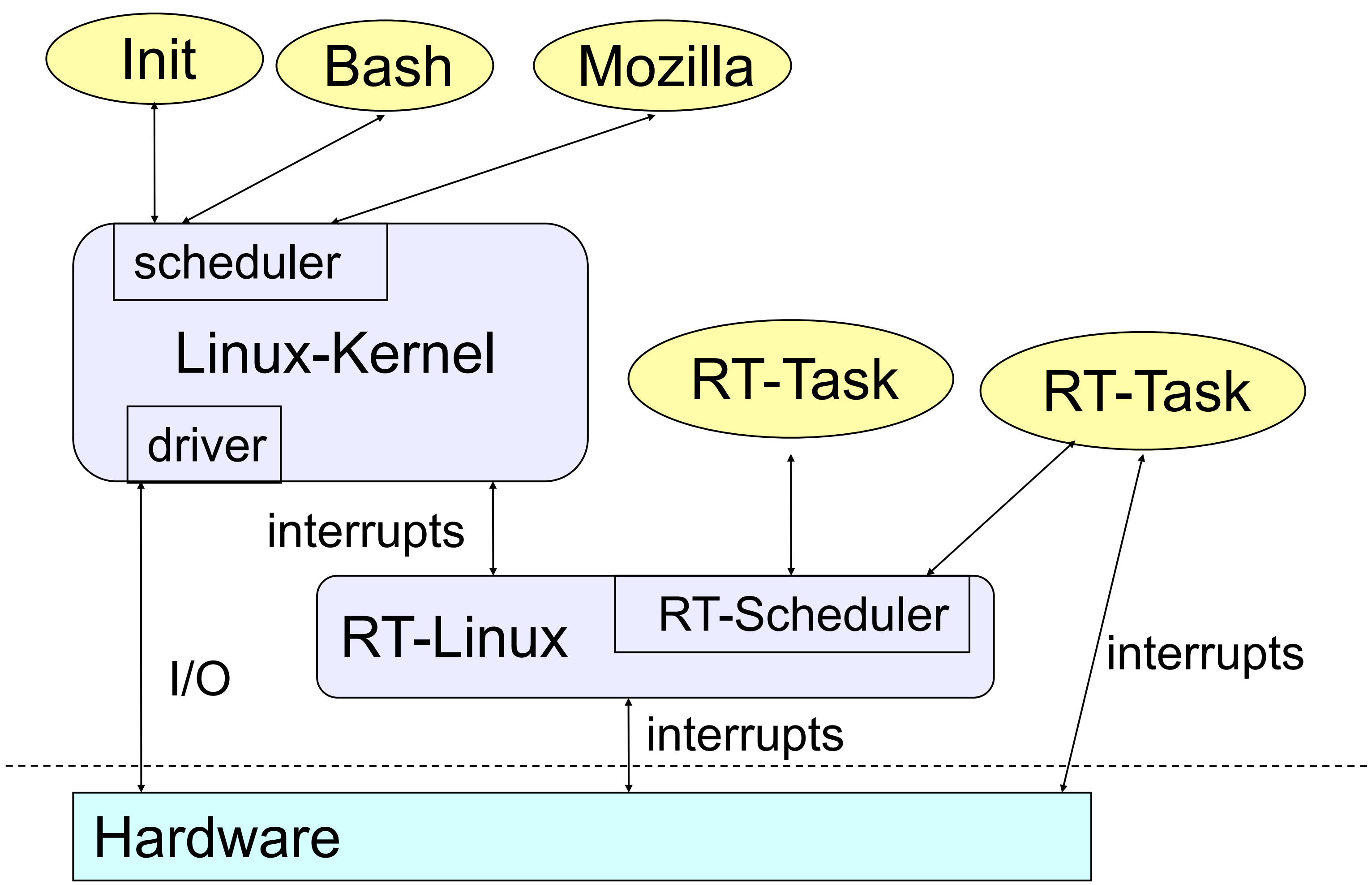
Classes of RTOSs:
2. RT extensions to standard OSs

Attempt to exploit comfortable main stream OS.
RT-kernel running all RT-tasks.
Standard-OS executed as one task.



+ Crash of standard-OS does not affect RT-tasks;
- RT-tasks cannot use Standard-OS services;
  less comfortable than expected

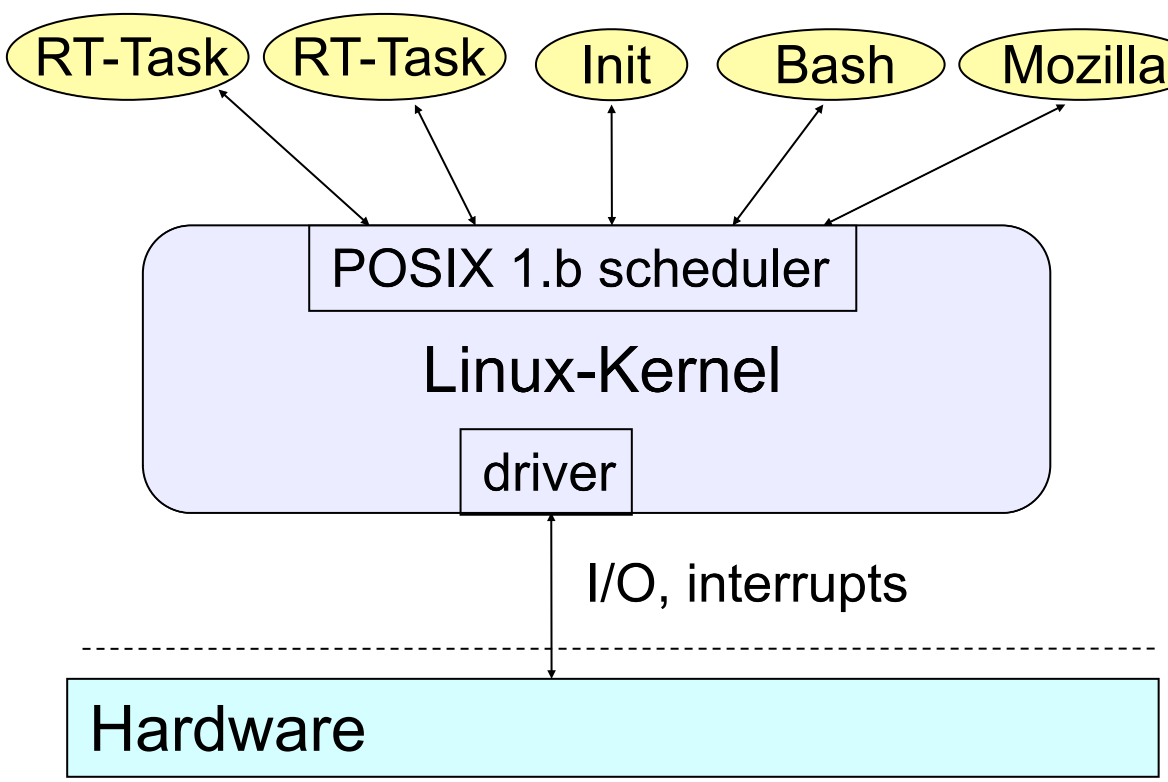
RT-Linux



RT-tasks
cannot use standard OS calls.
Commercially available from
fsmlabs (www.fsmlabs.com)

Posix

Standard scheduler can be replaced by POSIX scheduler implementing priorities for RT tasks



Special RT-calls and standard OS calls available.
Easy programming, no guarantee for meeting deadline

POSIX 1.b scheduler: real time ext

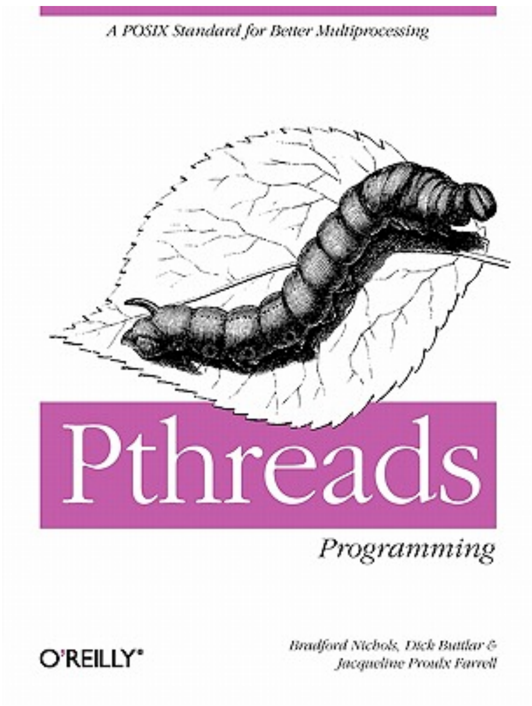
Real time Unix, supports POSIX and pthreads

POSIX provides three mechanisms for concurrency:

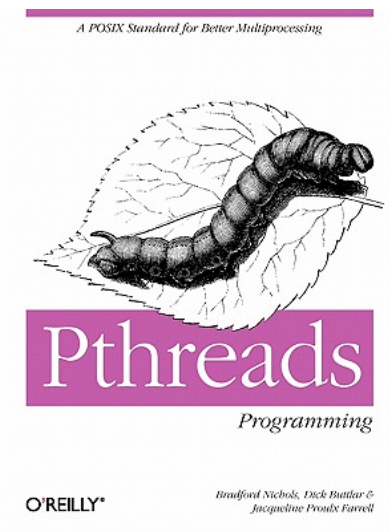
- Traditional Unix `fork` mechanism & associated `wait` call -
  - copy of entire process created & executed concurrently with parent (slow)
- `spawn` system call - equivalent to a combined `fork` & `join`
- Each process can also contain several threads of execution
  - pthreads which share a single address space (similar to Ada tasks & Java threads)

Pthreads: Small data structure - state, priority, etc.

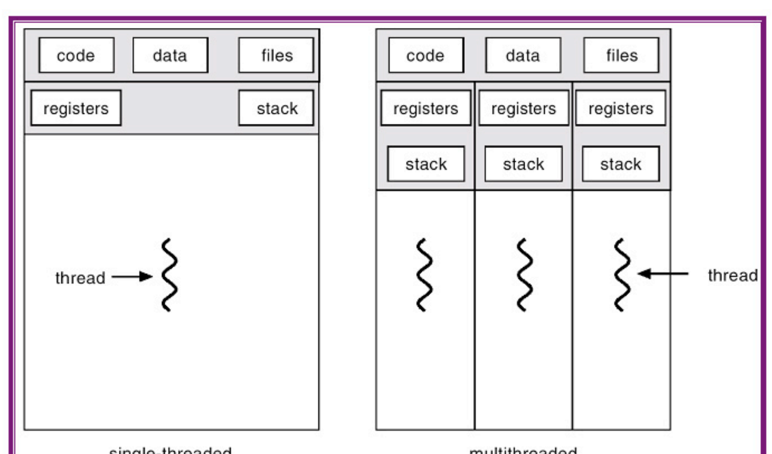
- Major goal: facilitate scheduling and resource allocation
- Priorities allocated at creation. Dynamically changed.
- Scheduling Premptive. FIFO or Round robin. Hook for users



Pthreads attributes



- All threads in POSIX have attributes (e.g. stack size)
- To manipulate these attributes, necessary to define an attribute object
  - (of type `pthread attr t`) - then call functions to get/set attributes
- A thread can be created once the correct attribute object has been established
- A thread becomes ready for execution as soon as it is created by `pthread create`
- A thread can terminate normally, or by calling `pthread exit`,
  - or by receiving a signal sent to it, or aborted by use of `pthread cancel`,
  - or wait for another to terminate (via `pthread join`)



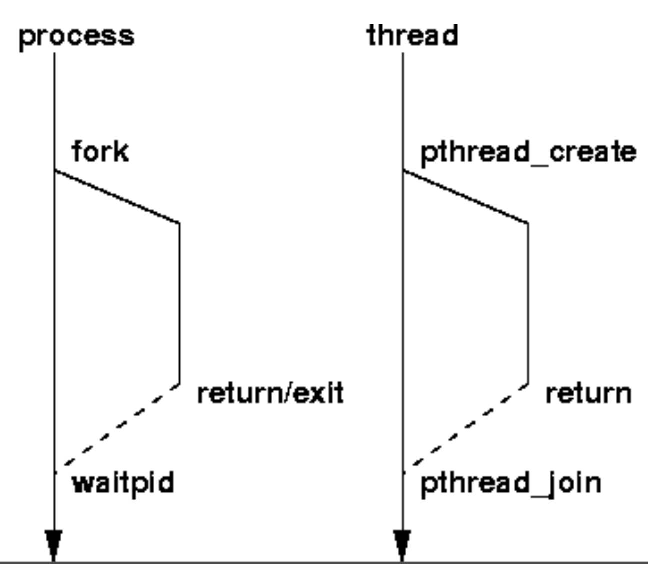
Pthread Interface

To clean up after a thread's execution & reclaiming storage (i.e. detaching) can be done either by:

- calling `pthread join` & waiting until thread terminates, or
- by setting the detached attribute of thread at creation time, or
- dynamically calling `pthread detach`

POSIX interface similar to compiler interface to runtime systems

- Low-level design allows explicit control
- However, higher-level language abstractions provided by Ada & Java removes
    - possibility of errors in using interface (unlike C)



Synchronisation, condition variables and signals

Pthreads synchronisation: mutex, semaphores and condition variables

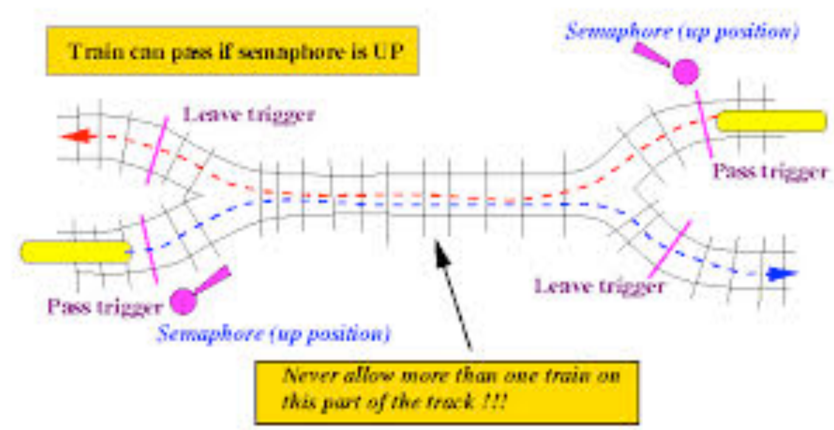
- Mutex -binary semaphore around critical sections.
- Priority inheritance and priority ceiling protocols supported

Condition variables: associated with mutex:wait,signal, broadcast

- P: pthread cond wait(not busy, mxlk) – block until not busy set, release
- Q: pthread cond signal(not busy, mxlk) – wake up Q add to mutex queue

Signals: a software interrupt with data and priority

- Q: siqqueue (P, fault, info) – send fault signal to P with info
- P: sigtimedwait(s,info, dt) – wait for signal set S, timeout after dt



Clocks in POSIX & C

- ANSI C has standard library for interfacing to calendar time
- Defines basic time type `time t` and routines for manipulating objects of type time
  - e.g. `clock gettime`
- POSIX allows many clocks to be supported by an implementation - each with its own identifier
  - of type `clockid t`
- IEEE standard requires at least one clock to be supported
  - `CLOCK REALTIME` - minimum resolution 20ms
- Function `clock getres` allows resolution of clock to be determined

Evaluation

According to Gupta, trying to use a version of a standard OS:

not the correct approach because too many basic and inappropriate underlying assumptions still exist such as optimizing for the average case (rather than the worst case), ... ignoring most if not all semantic information, and independent CPU scheduling and resource allocation.

Dependences between tasks not frequent for most applications of std. OSs & therefore frequently ignored.

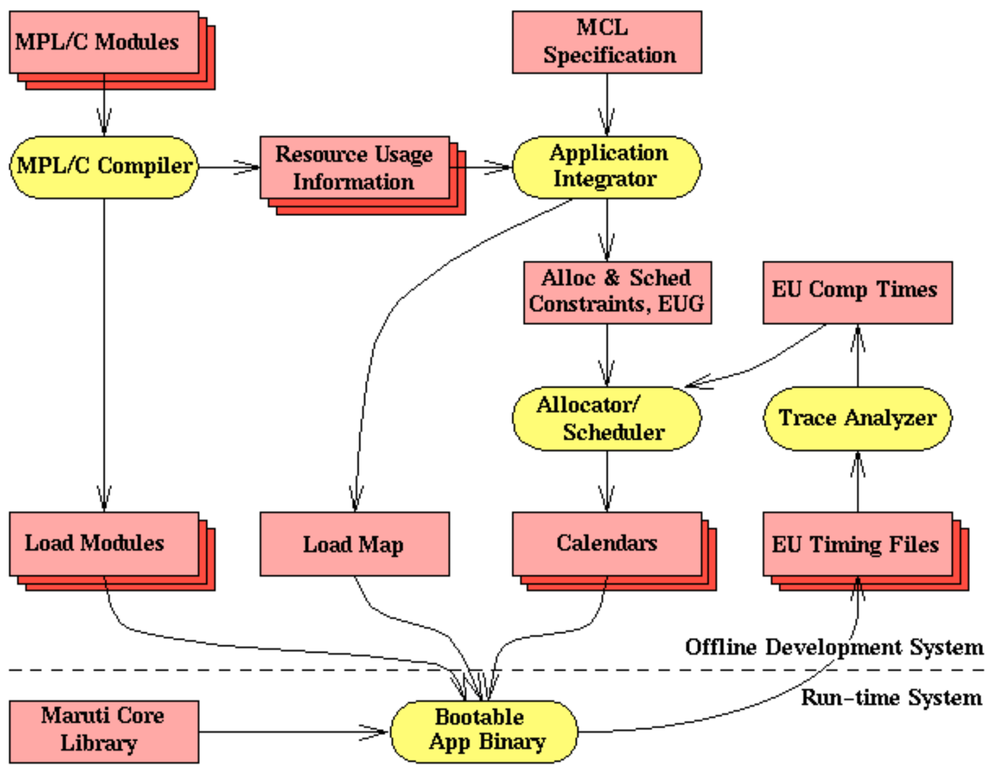
Situation different for ES since dependences between tasks are quite common.

Classes of RTOSs:
3. Research trying to avoid limitations

Research systems trying to avoid limitations.
Include MARS, Spring, MARUTI, Arts, Hartos, DARK, and Melody

Research issues

- low overhead memory protection,
- temporal protection of computing resources
- RTOSes for on-chip multiprocessors
- support for continuous media
- quality of service (QoS) control.



Summary

- General requirements
  - Configuration
  - Device Drivers
- Real time operating systems
  - Timing
  - Scheduling
  - Performance
- Examples
- Conclusion