SYSTEM eCos - Embedded Configurable Operating System

BELONGS TO THE „CYGNUS SOLUTIONS” founded about 1989 initiative connected with an idea of free software („commercial support for the free software”). Recently merged with RedHat.

CYGNUS was also the original developer of CYGWIN, a collection of free software tools to allow various versions of WINDOWS to act similar to a UNIX system (and try to satisfy POSIX norms).

eCos IS DEVELOPED FOR EMBEDDED PROCESSORS - VERY HETEROGENEOUS SET OF PROCESSORS, OFTEN OF LIMITED RESOURCES.

eCos IS DESIGNED AS HIGHLY CONFIGURABLE, WITH HARDWARE LAYER AS ABSTRACTED AS POSSIBLE.

DEVELOPERS HAVE FULL ACCESS TO THE ENTIRE SOFTWARE SOURCE CODE.

„INSTALLABLE” VERSIONS ARE AVAILABLE FOR A WIDE VARIETY OF PRODUCTS

The system is in form of PACKAGES.

Even the KERNEL is a package. IT IS THE KEY PACKAGE, however OPTIONAL! (it is possible to design an application - a single threaded application - which does not use any kernel functionality).
SYSTEM eCos - *Embedded Configurable Operating System*

**KERNEL FUNCTIONALITY:**

- NEW THREAD CREATION (at startup or when the system is running)
- CONTROL OF THREADS AND PRIORITY MANIPULATION
- SCHEDULER POLICY SELECTION
- SYNCHRONIZATION OF THREADS WITH A HELP OF WIDE RANGE OF TOOLS
- HANDLING OF INTERRUPTS AND EXCEPTIONS

The eCos kernel DOES NOT provide memory allocation functions or device drivers.

The memory allocation is handled by a separate package.

Device drivers are separated from the eCos kernel to satisfy the postulate of the maximal hardware abstraction.

One can design a single threaded application which DOES NOT use kernel functionality (kernel package) at all - but still profit from other packages of the eCos. Typically, such polling loop organization of a thread permanently checks all device flags and takes relevant actions to service I/O requirements. Any amount of calculation in each iteration increases the delay between events, eventually leading to an error.

Some advanced packages (e.g. TCP/IP stack) require multi-treading internally and then the kernel package is NOT optional.
SYSTEM eCos - Embedded Configurable Operating System

SCHEDULERS:

Schedulers are needed when system serves multiple threads. eCos kernel may run one of two schedulers: BITMAP scheduler and Multi-Level Queue scheduler (MLQ). Both use simple numerical priorities. The default range of priority levels is 32. That can be changed by the CYGNUM_KERNEL_SCHED_PRIORITIES option.

Level 0 is the highest priority.

BITMAP scheduler allows for one and only one thread per priority level. For the default case it means that the system may be configured with the 32 threads maximum.

BITMAP scheduler keeps track of threads which are currently runnable, identifies the one with the highest priority and similarly cares for those waiting for synchronization (semaphore, mutex or other primitive). Such indexing is simple and fast.

BITMAP scheduler does not support timeslicing so, consequently, the current thread can not be preempted in favour of another thread with the same priority. This makes the BITMAP scheduler totally deterministic.
SYSTEM eCos - *Embedded Configurable Operating System*

SCHEDULERS (cont.):

MLQ scheduler allows to configure the system with multiple threads with the same priority. There is no limit on the number of threads in the system then. Other limitations - e.g. available memory - are, of course, possible.

MLQ scheduler supports TIMESLICING, which is controlled by options, like

- CYGSEM_KERNEL_SCHED_TIMESLICE
- CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS

When active, the scheduler automatically switches from one runnable thread to another one with the same priority after some number of clock ticks.

Timeslicing is operational only when there are two runnable threads with the same priority and no runnable one with the higher priority.

Timeslicing can be disabled and then the thread will not be preempted. It continue running until explicitly yields the control to the processor or is blocked (e.g. waiting for mutex, semaphore ..)
SYSTEM eCos - *Embedded Configurable Operating System*

**SCHEDULERS (MLQ cont.):**

The default behaviour of the MLQ scheduler when threads are blocked is that it uses the *last-in-first-out* queueing. It means that when several threads waits on the semaphore and one post is released then the woken up thread is the last one which called `cyg_semaphore_wait` function. That makes queueing and dequeueing fast but - if waiting threads are of different priorities - then the first woken up thread may not be the one with the highest priority.

The option `CYGIMP_KERNEL_SCHED_SORTED_QUEUE`, when enabled, allows to require the strict priority queueing.

**SYNCHRONIZATION PRIMITIVES:**

- mutexes
- condition variables
- counting semaphores
- mail boxes
- event flags
SYSTEM eCos - *Embedded Configurable Operating System*

SYNCHRONIZATION PRIMITIVES (CONT.):

**MUTEXES** have a distinguished function in eCos: allow safe share of resources by multiple threads.

The mutex is locked by the thread which manipulates the shared resource and then unlocks the mutex again.

The other primitives are used for communication between threads or DSRs with the ISR or DSR with the thread itself.

**CONDITION VARIABLE** can be used by a thread which has locked a mutex but needs to wait for some condition to become true. A thread **WAITS ON** condition variable until another thread or DSR (Deferred Service Routine) use to wake it up. N.B. when thread waits on Condition Variable it releases the mutex (before waiting) and reacquires it before processing. *That is of delicate matter as it is important that no race conditions are introduced. Atomic operations and great care is needed!*

**A COUNTING SEMAPHORE** is used when one needs to communicate that certain event has occurred, or that it has occurred multiple times in succession. The waiting thread is called **the consumer** thread while the thread (or DSR) which posts the event is called **the producer**.
**SYSTEM eCos - Embedded Configurable Operating System**

**SYNCHRONIZATION PRIMITIVES (CONT.):**

The count associated with the semaphore records occurrence of multiple events, which are not lost and allow for succession of multiple wait operations.

MAIL BOXES similarly indicate occurrence of a particular event and at the same time allow to exchange some data item (often it is a pointer to some structure). The mail box has finite capacity and if *producer* generates mail box events faster than they are *consumed* then the mail box is blocked. Such feature usually eliminates mail boxes as a synchronization tools for the DSR - thread communication. They are typically used as a synchronization between threads.

EVENT FLAGS are used to wait for (from the one hand side) or to signal (from the other) various events. For such communication one uses usually a bit masks, in which given bits are associated with particular events. With such simple construct there is no way to deal with multiple events - as in the case of counting semaphores. Similarly, unlike mail boxes, there is no way to pass any additional data. On the other hand one can use event flags for communication between DSR and threads as there is no danger of an overflow and a subsequent blockade.
SYSTEM eCos - *Embedded Configurable Operating System*

**INTERRUPT HANDLING:**

Treads and interrupt handlers interactions.
- continuous polling of a volatile variable
- polling on every clock tick (10 ms)
- use of some synchronization primitive
Problem with possible kernel data corruption.

eCos two-level approach to interrupt handling: ISR and DSR.

**ISR** - fast Interrupt Service Routine - direct response to a „Interrupt Vector”
Efficient service with minimal references to the system leaving the rest of the service to
**DSR** - Deferred Service Routine which completes the interrupt service which may require
much more system calls (via e.g. using condition variables or by posting
a semaphores. It is also DSR which is supposed to contact the thread (e.g. to wake it up)

There are practices of disabling interrupts preventing the ISR from running - rather bad practise and should be used only for a very short periods of time and only for really necessary actions (like manipulating list of free buffers ...)
SYSTEM eCos - *Embedded Configurable Operating System*

**INTERRUPT HANDLING (cont.):**

Somehow similar problem - to the question of interrupts being enabled or disabled - is the problem of the kernel scheduler lock. Scheduler may be locked in response to function like:

```c
  cyg_mutex_lock
cyg_semaphore_post
```

which may be issued by the kernel for time of manipulation with some data structures after which the lock is released. Locking the scheduler may cause pending some DSRs and some chain of synchronization events influencing waking up of threads of various priorities.

... various possible scenarios ...

in which one can illustrate that the two-level approach to interrupt handling realised by eCos allows fo safe solutions (.. e.g. it is safer to leave `cyg_semaphore_post` to a DSR rather than to ISR which disabling interrupts may cause corruption of kernel data structure or even loss of some thread that would never run again).
SYSTEM eCos - *Embedded Configurable Operating System*

**SMP (Symmetric Multiprocessing Systems) SUPPORT**
(available only on some selected architectures and platforms)
- The main startup takes place on only one CPU („primary” CPU)
- All other CPUs are either suspended or “captured” by HAL
- Primary CPU – after some initializations – calls `cyg_start` to enter the application
- Secondary CPUs are initialized ONLY when an application calls `cyg_scheduler_start`
  (this routine scans the list of available secondary CPUs and invokes `HAL_SMP_CPU_START` to start each CPU
  and then calls an internal function `Cyg_Scheduler::start_cpu` to enter the Scheduler for the primary CPU)
- Each secondary CPU starts in the HAL, completes initialization at calls `kernel` at `cyg_kernel_cpu_startup`, claims the scheduler lock and calls `Cyg_Scheduler::start_cpu`, which is common for all CPUs. Now an interrupt object ls installed for inter-CPU interrupts and after that all CPU are equal.
  (Some hardware may still distinguish them delivering interrupts)
HAL Hardware Abstraction Layer

eCOS is written mostly in C++ - HAL: in C and assembler

BASE DEFINITIONS
ARCHITECTURE CHARACTERISATION
INTERRUPT HANDLING
CLOCK AND TIMERS
HAL I/O
CACHE CONTROL
...

BASE: e.g. endianness (bit ordering), label translation, type definitions, …

ARCHITECTURE: register save format, thread context initialisation and switching, stack sizes, address translation, …

INTERRUPT HANDLING: vector numbers, interrupt state control, ISR/VSR management,

CLOCK and TIMERS: clock control, delay routines, clock frequency control, …

HAL I/O: register address, register read, register write …

CACHE CONTROL: cache dimension, global control – synchronize, write, lock, unlock …