Description of Bossa events

The following is a description of the events handled by the Bossa implementation of the Linux 2.4 scheduling policy (found here). In general, Bossa events are organized in a hierarchy, describing first the behavior of the event and then the position in the kernel from which the event is notified. Thus, block.* matches all of the blocking events in the kernel. A complete list of the events is in the file include/linux/bossa_events.h, contained in the current version of the Bossa implementation. A description of the notation used in describing the event types is contained here.

The event types are as follows:

- **process.new** : This event is used when a new process is created, in the file kernel/fork.c following the various Linux-related initializations of the new process. This event is to be provided only by a scheduler that is declared to be a default scheduler (using the default keyword at the beginning of the policy), and that thus may receive newly forked processes.

  The event has both a source process (the parent of the new process) and a target process (the new process itself). The source process is normally in a RUNNING state. If the source process is about to be preempted, it can be in a READY state. If some interrupt has occurred that implies that the source process should be considered to be blocked from the point of view of the policy, then the source process can be in a BLOCKED state. More information about this last possibility is available here.

  If the source process is initially in a RUNNING state, the handler can put it in a READY, eg if the source process gives some of its right to execute to its child, and thus is forced to preempt itself. In the cases where the source process is not mentioned in the output portion of the type, the source process must remain in its initial state, as described here.

  ```
  [tgt in NOWHERE, src in RUNNING] -> { [tgt in READY], [[src, tgt] in READY] }

  [[] = RUNNING, src in READY, tgt in NOWHERE] -> [tgt in READY]

  [[] = RUNNING, src in BLOCKED, tgt in NOWHERE] -> [tgt in READY]
  ```

  The following type additionally describes the situation at boot time, or when the parent of the process is managed by a different scheduler than the one destined for the child (when a hierarchy of schedulers is used).

  ```
  [tgt in NOWHERE] -> [tgt in READY, src in NOWHERE]
  ```

- **process.end** : This event is used when the structure associated with the target process is about to be freed by its parent, in release_task of kernel/exit.c. The process must be in a BLOCKED state when this event is notified.

  ```
  [tgt in BLOCKED] -> [tgt in TERMINATED]
  ```

- **block.*** : This event occurs when a process blocks. Notifications occur throughout the kernel, wherever the state of a process is set to TASK_INTERRUPTIBLE or TASK_UNINTERRUPTIBLE at the point of a call to schedule. The target process is normally in a READY state, but may be in a READY or BLOCKED state as described for the source process in the type of process.new

```
[tgt in RUNNING] -> [tgt in BLOCKED]

[[] = RUNNING, tgt in READY] -> [tgt in BLOCKED]

[[] = RUNNING, tgt in BLOCKED] -> [tgt in BLOCKED]
```

- **unblock.preemptive** : The event occurs when the kernel wakes up the set of processes that have been added to some wait queue. This takes place in the function try_to_wake_up of kernel/sched.c. Because a process on a wait queue has not necessarily actually blocked, the target process can be in any state other than a TERMINATED state.

  In the first rule, the target process is actually blocked and becomes runnable. In the second rule, the running process is additionally indicated to be preempted by moving the running process to a READY state, which causes a new running process to be elected). The third rule allows the target process to remain in a BLOCKED state or move between BLOCKED states, to account for the possibility that it is not actually blocked waiting for the event associated with the wait queue, but has yielded the processor for some other reason (e.g. a yield event). In the remaining cases, the target process is either in a RUNNING or READY state and no state transition is allowed.

  ```
  [tgt in BLOCKED] -> [tgt in READY]

  [p in RUNNING, tgt in BLOCKED] -> [[p,tgt] in READY]

  [tgt in BLOCKED] -> [tgt in BLOCKED]

  [tgt in RUNNING] -> []

  [tgt in READY] -> []
  ```

- **unblock.nonpreemptive** : This event is similar to **unblock.preemptive** , but the waking process is not allowed to preempt the running process. This event occurs at only a few places in the kernel (specifically in pipe.c).

  ```
  [tgt in BLOCKED] -> { [tgt in READY], [tgt in BLOCKED] }

  [tgt in RUNNING] -> []

  [tgt in READY] -> []
  ```

- **unblock.timer.*** : This event occurs when a timer set by a user process times out. The target of the event is the user process associated with the timer. The following timer-related functions are provided by the Bossa run-time system:

  - void start_absolute_timer(timer,time);: set timer for time units (jiffies) in the future.
  - void start_relative_timer(timer,time);: set timer for the absolute time time.
  - void stop_timer(timer);: stop timer.
  - time now(): returns the current time.

  There must be one unblock.timer handler for each global timer variable and timer process/scheduler field. For example, if a policy defines the timer-typed global variable xxx, then the policy must defined the handler unblock.timer.xxx. A process scheduler may not define the generic handler unblock.timer.*.

This handler may only be defined by a virtual scheduler, which uses this handler to forward a timer event associated with a timer defined by a child scheduler.

The event notification occurs in the Bossa run-time system, kernel/bossa.c. The target process can be in any state other than a TERMINATED state.

The following type rules apply to unblock.timer.* and to unblock.timer.*name* , where *name* is the name of a timer declared in the process/scheduler structure of the current scheduler. The first rule means that if the target process is in a RUNNING state, then it can be left where it is or moved to a READY or BLOCKED state. Additionally, processes can change state within the READY state class and within the BLOCKED state class. The second and third rules allow the target process to move between the READY and BLOCKED state classes, and other processes to change state within these state classes. The final rule allows preemption of the running process if the target process ends up in a READY state.

```
[tgt in RUNNING] ->
  { [READY!, BLOCKED!], [tgt in READY!, BLOCKED!], [READY!, tgt in BLOCKED!] }

[tgt in READY] -> { [READY!, BLOCKED!], [READY!, tgt in BLOCKED!] }

[tgt in BLOCKED] -> { [READY!, BLOCKED!], [tgt in READY!, BLOCKED!] }

{ [p in RUNNING, tgt in READY], [p in RUNNING, tgt in BLOCKED] } ->
    [[p,tgt] in READY!, BLOCKED!]
```

The following type rules apply to unblock.timer.*name* , where *name* is the name of a timer declared as a global variable in the current scheduler. These are similar in spirit to the types for the case where the timer is associated with a particular process or scheduler, except that there is no target. In general, processes in exactly one state class are allowed to change to another state class, and other processes are allowed to change state within their state class. If there is a process in the RUNNING state class and processes are moved from the BLOCKED state class to the READY state class, the the process in the RUNNING state class can be preempted.

```
[p in RUNNING] ->
   { [READY!, BLOCKED!], [READY!, p in BLOCKED!], [p in READY!, BLOCKED!] }

[p in READY] -> { [READY!, BLOCKED!], [READY!, p in BLOCKED!] }

[p in BLOCKED] -> { [READY!, BLOCKED!], [p in READY!, BLOCKED!] }

[p0 in RUNNING, p1 in BLOCKED] -> [[p0,p1] in READY!, BLOCKED!]
```

More information about timers is available in the release notes for September 15, 2003

- **system.clocktick** : This event occurs on each clock tick in which some process other that the idle process is currently running. The target of the event is the currently running process. This event is notified from the update_times function in kernel/timer.c which is invoked as part of the bottom half of the kernel timer. In Bossa, the target process is always in a RUNNING state. The target process can be left where it is or moved to a READY or BLOCKED state.

  ```
  [tgt in RUNNING] -> { [], [tgt in READY], [tgt in BLOCKED] }
  ```

- **yield.system.pause.*** : This event occurs at various places in the kernel where SCHED_YIELD is added to the policy of the process and the kernel process election function schedule is then called. The

effect of SCHED_YIELD is that the process remains eligible for election but is temporarily given the lowest possible priority.

The target process is normally in a RUNNING state, in which case it is preempted by placing it in a READY state. As described in the case of process.new, the target process can also be in a READY state or a BLOCKED state. If it is in a READY state then it can change state within this state class. If it is in a BLOCKED state, then no state change is allowed.

```
[tgt in RUNNING] -> [tgt in READY]

[[] = RUNNING, tgt in READY] -> [tgt in READY]

[[] = RUNNING, tgt in BLOCKED] -> []
```

- **yield.system.immediate.*** : This event occurs at various places in the kernel when schedule is called without changing the state of the process. In this case, the intent is that the call to schedule should cause the ready process that currently has the highest priority to be given the CPU. Unfortunately, the event types, or more precisely the Bossa classes, are not fine-grained enough to distinguish between **yield.system.immediate.*** and **yield.system.pause.*** , and the type rules are thus the same.

  ```
  [tgt in RUNNING] -> [tgt in READY]

  [[] = RUNNING, tgt in READY] -> [tgt in READY]

  [[] = RUNNING, tgt in BLOCKED] -> []
  ```

- **yield.user.*** : Notification of this event replaces the Linux function sys_sched_yield defined in kernel/sched.c, which implements the sched_yield system call. The target is the process that would like to yield. This process can be in the RUNNING state, or in a READY or BLOCKED state, if the process is about to be preempted.

  ```
  [tgt in RUNNING] -> [tgt in READY]

  [[] = RUNNING, tgt in READY] -> [tgt in READY]

  [[] = RUNNING, tgt in BLOCKED] -> []
  ```

- **preempt** : This event occurs in a scheduling hierarchy when a parent scheduler decides to move its child scheduler in the RUNNING state to a state in some other class. In this case, the compiler implicitly generates a preempt event notification from the parent to the preempted child scheduler. The handler for this event in the child checker must preempt the running process.

  ```
  [tgt in RUNNING] -> [tgt in READY]
  ```

- **bossa.schedule** : This event occurs when a new ready process should be elected. Notification of the event occurs in the schedule function defined in kernel/sched.c, in the case that the scheduler state indicates that there is some process in a READY state.

  ```
  [[] = RUNNING, p in READY] -> [p in RUNNING, READY!]
  ```