MODULE *sched2*

There are some number of processes that can be run, say *numProcs*. Each process table entry is marked as NOT-*RUNNABLE* (it doesn't exist or it is asleep), *RUNNABLE*, or *RUNNING*. A process that exists may transition while it is *RUNNING* to either NOT-*RUNNABLE* (it decides to die or sleep), or can be pre-empted and transition to *RUNNABLE*. In both cases, the scheduler will be invoked. A process entry marked as NOT-*RUNNABLE* can transition to *RUNNABLE* at any time. This is the case if the task a process is waiting on completes (*i.e.* the process wakes up), or a new process is spawned, and needs to be scheduled. A *RUNNABLE* process cannot transition to NOT-*RUNNABLE*. Although this could happen via the "kill" command, in this model we will not allow it.

The model will be instantiated as a snapshot in the middle of *Xv*6's operation. In other words, we are not going to model the *Xv*6 startup routine, which includes (keep in mind this is after bootloading) instantiating kernel memory, starting up the other processors and their associated data structures, instantiating user memory, running the first user process, etc. Instead, we will assume that initially, a single process exists that is running on the first *CPU*. Other processes may be spawned (and when they are, we will take care to place them in particular places on the process table, so as to limit the model checking state-space).

According to the transition rules mentioned previously, it should now be possible to have no *RUNNABLE* processes. This would happen if the only running process there was chose to die / sleep. It's also possible for the process table to fill up completely. We don't need to worry about the process table to "overflow", since we only allow (and the *Xv*6 only allows) a fixed maximum number of processes anyway.

The *procTable* lock, the *TLB*, and kernel, user modes are also modeled.

These are the steps for pre-emption:
- A *RUNNING* process goes from user to kernel mode (the timer TRAP causes this).
- The process acquires (or tries to acquire) the ptable lock. If it is unavailable, then the process simply waits. The process's state is still *RUNNING*.
- If the process acquires the ptable lock, it then changes its state from *RUNNING* to *RUNNABLE*. At this point, it is the only process running the scheduler (it has acquired the ptable lock). The *TLB* is changed to reflect the kernel's page table.
- The process, or now scheduler, finds a free process to run. There should be at least one (itself: it should be in the *RUNNABLE* state). The process is chosen, the *TLB* is is refreshed (via loading $\%cr3$), and the ptable lock is released.

To create a new process, the ptable lock is acquired magically, and a process is spawned. The reason I'm going with this approach is to cap the model checking state-space. In real life, a process may spawn another process via "fork" or some other syscall. That call would then have to be modeled, and I'd rather not do that. It shouldn't be too complex; it would involve modifying the ptable and setting up the new process's memory. But my goal isn't to model check the "fork" syscall, it's to model check the scheduler. And so I'll pass.

The process spawned will take on the lowest ptable index entry available. This happens in the actual *Xv*6 scheduler, and greatly benefits me by once again limiting the state-space.

Finally, the steps for dying / sleeping are similar to the pre-emption steps. Once again, the process tries to acquire the ptable lock, and when it finally does, it marks itself off as NOT-*RUNNABLE* instead. It then does everything else from step three onwards described under pre-emption. The only difference is that there may not be a single process to run. And what happens when there isn't? If the other process tried to sleep, it wouldn't be able to, since it would require the ptable lock – DEADLOCK! This is avoided in the way the scheduler algorithm works, described below.

Information about the Scheduler Algorithm (Round Robin)
The scheduler runs the round robin algorithm to pick the next process to run. There isn't a global round robin "head", that points at the next process to run. Instead, whenever the scheduler is run after the death / falling asleep / pre-emption of a process, the search is performed starting at the process immediately after the unlucky process. When there are no processes to run, the scheduler relinquishes the ptable lock briefly, before trying to starting to search again. This lets other processes acquire the ptable lock if they need to, for example, when they want to go to sleep.

EXTENDS *Naturals*, *TLC*

Probably going to run with *numProcs* as 4, *numCPUs* set to 2.

CONSTANTS *numProcs*, *numCPUs*

The various process entry states:

CONSTANTS *RUNNABLE*, *NOTRUNNABLE*, *RUNNING*

procTable: a mapping from process number to process state and the *CPU RUNNING* it.
cpus: a mapping from *CPU* number to the process it is *RUNNING*.
pTableLock: a lock on the process table. When not held, 0.
tlb: a mapping from *CPU* to the page table the *CPU* is using. Kernel's (0) or a process's (*PID*).
scheduling: denotes when the scheduler is active, 0 when inactive. Otherwise contains *CPU ID*.
head: where the *RR* search starts from.

VARIABLES *procTable*, *cpus*, *pTableLock*, *tlb*, *scheduling*, *head*

The classes to which each variable belongs.

$$
\begin{aligned}
&TypeInfo \triangleq\\
&\quad \land\ procTable \in [(1 \,..\, numProcs) \rightarrow \{RUNNABLE, NOTRUNNABLE, RUNNING\} \times (0 \,..\, numCPUs)]\\
&\quad \land\ cpus \in [(1 \,..\, numCPUs) \rightarrow (0 \,..\, numProcs)]\\
&\quad \land\ pTableLock \in \{0, 1\}\\
&\quad \land\ tlb \in [(1 \,..\, numCPUs) \rightarrow (0 \,..\, numProcs)]\\
&\quad \land\ scheduling \in (0 \,..\, numCPUs)\\
&\quad \land\ head \in (1 \,..\, numProcs)
\end{aligned}
$$

Initially, only one process, the very first in the table, is active. All the other processes don't exist. New processes will be magically spawned later. *cpus* is initialized so that only the first CPU is running something: the first process. The tlb is initialized to reflect that, and the ptable lock is not held.

$$
\begin{aligned}
& Init \triangleq \\
& \quad \land procTable = [p \in (1 \,..\, numProcs) \mapsto \\
& \qquad\qquad\qquad \text{IF } p = 1 \text{ THEN } \langle RUNNING, p \rangle \text{ ELSE } \langle NOTRUNNABLE, 0 \rangle] \\
& \quad \land cpus = [c \in (1 \,..\, numCPUs) \mapsto \\
& \qquad\qquad\qquad \text{IF } c = 1 \text{ THEN } 1 \text{ ELSE } 0] \\
& \quad \land pTableLock = 0 \\
& \quad \land tlb = [c \in (1 \,..\, numCPUs) \mapsto \text{IF } c = 1 \text{ THEN } 1 \text{ ELSE } 0] \\
& \quad \land scheduling = 0 \\
& \quad \land head = 1
\end{aligned}
$$

The next process to be run.
IMPORTANT: This operator is valid ONLY when there exists a $RUNNABLE$ process.

$$
\begin{aligned}
& ChooseProc(p) \triangleq \\
& \quad \text{LET } mod1(i) \triangleq ((i - 1)\% numProcs) + 1 \text{IN} \\
& \quad mod1(\\
& \qquad \text{CHOOSE } i \in ((p + 1) \,..\, (p + numProcs)) : \\
& \qquad\quad \land \ procTable[mod1(i)][1] = RUNNABLE \\
& \qquad\quad \land \ \forall x \in ((p + 1) \,..\, (p + numProcs)) : \\
& \qquad\qquad\quad \lor procTable[mod1(x)][1] \neq RUNNABLE \\
& \qquad\qquad\quad \lor i \leq x \\
& \quad)
\end{aligned}
$$

The transition from $RUNNING$ to $RUNNABLE$. The scheduler can now be invoked.

$$
\begin{aligned}
& Preemption \triangleq \\
& \quad \land \exists c \in (1 \,..\, numCPUs) : \\
& \qquad \land cpus[c] \neq 0 \\
& \qquad \land pTableLock = 0 \\
& \qquad \land pTableLock' = 1 \\
& \qquad \land scheduling' = c \\
& \qquad \land procTable' = [i \in (1 \,..\, numProcs) \mapsto \\
& \qquad\qquad\qquad \text{IF } i = cpus[c] \text{ THEN } \langle RUNNABLE, 0 \rangle \text{ ELSE } procTable[i]] \\
& \qquad \land cpus' = [i \in (1 \,..\, numCPUs) \mapsto \\
& \qquad\qquad\qquad \text{IF } i = c \text{ THEN } 0 \text{ ELSE } cpus[i]] \\
& \qquad \land tlb' = [i \in (1 \,..\, numCPUs) \mapsto \\
& \qquad\qquad\qquad \text{IF } i = c \text{ THEN } 0 \text{ ELSE } tlb[i]] \\
& \qquad \land head' = cpus[c]
\end{aligned}
$$

The $pTableLock$ should be held, but we will check only for the "scheduling" condition, as within the scheduler, we don't perform an actual check on the $pTableLock$.

$$
\begin{aligned}
& Schedule \triangleq \\
& \quad \land scheduling \neq 0
\end{aligned}
$$

$\land$

$\quad\lor$

$\quad\quad\land\ \exists\, p \in (1\,..\,numProcs) : procTable[p][1] = RUNNABLE$

$\quad\quad\land\ \textsc{let}\ newProc \triangleq ChooseProc(head)\ \textsc{in}$

$\quad\quad\quad\land\ procTable' = [i \in (1\,..\,numProcs) \mapsto$

$\qquad\qquad\qquad\qquad \textsc{if}\ i = newProc\ \textsc{then}\ \langle RUNNING, scheduling\rangle\ \textsc{else}\ procTable[i]]$

$\quad\quad\quad\land\ tlb' = [i \in (1\,..\,numCPUs) \mapsto$

$\qquad\qquad\qquad \textsc{if}\ i = scheduling\ \textsc{then}\ newProc\ \textsc{else}\ tlb[i]]$

$\quad\quad\quad\land\ cpus' = [i \in (1\,..\,numCPUs) \mapsto$

$\qquad\qquad\qquad \textsc{if}\ i = scheduling\ \textsc{then}\ newProc\ \textsc{else}\ cpus[i]]$

$\quad\lor$

$\quad\quad\land\ \forall\, p \in (1\,..\,numProcs) : procTable[p][1] \neq RUNNABLE$

$\quad\quad\land\ \textsc{unchanged}\ tlb$

$\quad\quad\land\ \textsc{unchanged}\ cpus$

$\quad\quad\land\ \textsc{unchanged}\ procTable$

$\land\ scheduling' = 0$

$\land\ pTableLock' = 0$

$\land\ \textsc{unchanged}\ head$

The transition from *RUNNING* to *NOTRUNNABLE*. This is very similar to *Preemption* above, except for the small change that *RUNNING* goes to *NOTRUNNABLE* instead of *RUNNABLE*. The scheduler is invoked, and it will deal with whether there is a running process.

$Sleep \triangleq$

$\quad\land\ \exists\, c \in (1\,..\,numCPUs) :$

$\quad\quad\land\ cpus[c] \neq 0$

$\quad\quad\land\ pTableLock = 0$

$\quad\quad\land\ pTableLock' = 1$

$\quad\quad\land\ scheduling' = c$

$\quad\quad\land\ procTable' = [i \in (1\,..\,numProcs) \mapsto$

$\qquad\qquad\qquad \textsc{if}\ i = cpus[c]\ \textsc{then}\ \langle NOTRUNNABLE, 0\rangle\ \textsc{else}\ procTable[i]]$

$\quad\quad\land\ cpus' = [i \in (1\,..\,numCPUs) \mapsto$

$\qquad\qquad\qquad \textsc{if}\ i = c\ \textsc{then}\ 0\ \textsc{else}\ cpus[i]]$

$\quad\quad\land\ tlb' = [i \in (1\,..\,numCPUs) \mapsto$

$\qquad\qquad\qquad \textsc{if}\ i = c\ \textsc{then}\ 0\ \textsc{else}\ tlb[i]]$

$\quad\quad\land\ head' = cpus[c]$

The magic transition from *NOTRUNNABLE* to *RUNNABLE*. Choose the first available *procTable* entry.

$MagicRunnable \triangleq$

$\quad\land\ pTableLock = 0$

$\quad\land\ \exists\, p \in (1\,..\,numProcs) :$

$\quad\quad\land\ procTable[p][1] = NOTRUNNABLE$

$\quad\quad\land\ \forall\, x \in (1\,..\,numProcs) :$

$\quad\quad\quad\lor\ p \leq x$

$\lor\ procTable[x][1] \neq NOTRUNNABLE$

$\land\ procTable' = [i \in (1\ ..\ numProcs) \mapsto$
$\quad \text{IF } i = p \text{ THEN } \langle RUNNABLE, 0\rangle \text{ ELSE } procTable[i]]$

$\land$ UNCHANGED $cpus$
$\land$ UNCHANGED tlb
$\land$ UNCHANGED $pTableLock$
$\land$ UNCHANGED $scheduling$
$\land$ UNCHANGED $head$

The scheduler can come alive magically, from the first idle CPU.

$MagicSchedule \triangleq$

$\land\ pTableLock = 0$
$\land\ \exists\, c \in (1\ ..\ numCPUs):$
$\quad \land\ cpus[c] = 0$
$\quad \land\ \forall\, x \in (1\ ..\ numCPUs) : (c \leq x \lor cpus[x] \neq 0)$
$\quad \land\ head' = numProcs$
$\quad \land\ scheduling' = c$
$\land\ pTableLock' = 1$
$\land$ UNCHANGED $cpus$
$\land$ UNCHANGED tlb
$\land$ UNCHANGED $procTable$

One of the following actions can happen. The actions lead to processes coming alive, dying, being scheduled, being preempted, etc.

$Next \triangleq$

$\lor\ Preemption$
$\lor\ Sleep$
$\lor\ Schedule$
$\lor\ MagicRunnable$
$\lor\ MagicSchedule$

Every $RUNNING$ process is using the right TLB.

$TLBValid \triangleq$

$\land\ \forall\, c \in (1\ ..\ numCPUs):$
$\quad \lor\ cpus[c] = 0 \land tlb[c] = 0$
$\quad \lor\ cpus[c] \neq 0 \land cpus[c] = tlb[c] \land procTable[cpus[c]][1] = RUNNING$

The CPU running the scheduler must not have a process associated with it

$SchedCPUIsFree \triangleq$

$\lor\ scheduling = 0$
$\lor\ cpus[scheduling] = 0$

If the scheduler is active, the $pTableLock$ must be held.

scheduler active implies $pTableLock$ is held.

$SchedulerHasLock \triangleq$
$\quad \land (scheduling \neq 0) \Rightarrow (pTableLock = 1)$

True if there are two $CPUs$ that map to the same process.

$SameProc \triangleq$
$\quad \land \exists i \in (1 \,..\, numCPUs) : cpus[i] \neq 0 \land (\exists j \in (1 \,..\, numCPUs) : i \neq j \land cpus[i] = cpus[j])$

No two $CPUs$ should be running the same process

$NotSameProc \triangleq$
$\quad \land \neg SameProc$