A Guide to Understanding sched2.tla

This document will break down each line of mathematics in sched2.tla (the TLA+ spec of the Xv6 scheduler), and how it models or approximately models the behavior of the actual scheduler. Each line of math will be explained and the original Xv6 code will be shown where necessary.

Initialization and Type Information

`TypeInfo` below speficies the structure of each variable (`procTable`, `cpus`, `pTableLock`, etc.). For example, `procTable` is essentially a function that maps a process number to the corresponding process's current state and CPU it is being run on. `Init` initializes the system to have the very first process in the process table set to `RUNNING`, and has all the other processes `NONRUNNABLE`. The corresponding CPUs are set. The scheduler isn't running, and nobody is holding on to the `pTableLock`. The `tlb` is a function mapping a CPU number to an address space. The `head` is the last process to have been run (the first process, in our case).

NOTE: indexing is done from `1` in TLA+, just as it is in MATLAB. Thus the first entry in the `procTable` would be `procTable[1]`.

```
 TypeInfo ==
    /\ procTable \in [(1 .. numProcs) ->
                         {RUNNABLE, NOTRUNNABLE, RUNNING} \X (0 .. numCPUs)]
    /\ cpus \in [(1 .. numCPUs) -> (0 .. numProcs)]
    /\ pTableLock \in {0, 1}
    /\ tlb \in [(1 .. numCPUs) -> (0 .. numProcs)]
    /\ scheduling \in (0 .. numCPUs)
    /\ head \in (1 .. numProcs)

Init ==
    /\ procTable = [p \in (1 .. numProcs) |->
                         IF p = 1 THEN <<RUNNING, p>> ELSE <<NOTRUNNABLE, 0>>]
    /\ cpus = [c \in (1 .. numCPUs) |->
                 IF c = 1 THEN 1 ELSE 0]
    /\ pTableLock = 0
    /\ tlb = [c \in (1 .. numCPUs) |-> IF c = 1 THEN 1 ELSE 0]
    /\ scheduling = 0
    /\ head = 1
```

Choosing the next process

For TLA+ convenience, the assumption that there is a `RUNNABLE` process is made. `ChooseProc(p)` is just an operator that is called.

```
 \* The next process to be run.
\* IMPORTANT: This operator is valid ONLY when there exists a RUNNABLE process.
ChooseProc(p) ==
    LET mod1(i) == ((i - 1) % numProcs) + 1 IN
    mod1(
        CHOOSE i \in ((p + 1) .. (p + numProcs)):
            /\ procTable[mod1(i)][1] = RUNNABLE
            /\ \A x \in ((p + 1) .. (p + numProcs)):
                \/ procTable[mod1(x)][1] # RUNNABLE
                \/ i <= x
    )
```

The above TLA+ logic starts at some `p` (an entry in the process table), and then searches the process table (`procTable`) for a free entry, just as the logic in `int scheduler(void)` in proc.c does:

```
for(;;){
    //...
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);
  }
```

Preemption

This clause models what happens when an interrupt occurs and the scheduler is invoked. The Xv6 code from trap.c is invoked when the appropriate timer interrupt occurs. The process is then told to yield the CPU.

```
void
trap(struct trapframe *tf) {
    // ...

    // Force process to give up CPU on clock tick.
    // If interrupts were on while locks held, would need to check nlock.
    if(myproc() && myproc()->state == RUNNING &&
        tf->trapno == T_IRQ0+IRQ_TIMER)
        yield();
}
```

And `yield()` grabs the proc table lock. The state of the current process is changed from `RUNNING` to `RUNNABLE`.

```
// Give up the CPU for one scheduling round.
void
yield(void)
{
  acquire(&ptable.lock);  //DOC: yieldlock
  myproc()->state = RUNNABLE;
  sched();
  release(&ptable.lock);
}
```

And `sched()` performs a few checks, and finally switches to the scheduler's context.

```
void
sched(void)
{
  int intena;
  struct proc *p = myproc();

  if(!holding(&ptable.lock))
    panic("sched ptable.lock");
  if(mycpu()->ncli != 1)
    panic("sched locks");
  if(p->state == RUNNING)
    panic("sched running");
  if(readeflags()&FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

On entering the scheduler (via `swtch`), the page table is updated and the TLB is flushed with `switchkvm();`, which does this:

```
 void
switchkvm(void)
{
  lcr3(V2P(kpgdir));   // switch to the kernel page table
}
```

The below models the steps above. However, whereas the above happens non-atomically (tomically?), they are compressed into a single step in the TLA+ spec. There are a few inconsistencies between the original scheduler and this specification as a result. For example, perhaps before the process yielded, another timer interrupt occurred triggering a second `yield()`. First of all, can this happen? (Is it skipped when we're in the kernel?) If it does, what's the outcome? (Seems to be unfairness to the process; it will be end up running the scheduler again when it becomes its turn to run).

In choosing a degree of atomicity, some tradeoffs are made. This was a tradeoff I made.

Preemption happens when a timer interrupt occurs on a CPU that is currently running a process. Preemption can happen when the proc table lock isn't held (only then will the `acquire(&ptable.lock);` above succeed). The lock is then held (`pTableLock' = 1`). The process table is updated to have the entry change from `RUNNING` to `RUNNABLE`. The TLB is updated, correctly mirroring the original Xv6 code. `head` is updated to let the scheduler know where to start its search from.

```
 \* The transition from RUNNING to RUNNABLE. The scheduler can now be invoked.
Preemption ==
    /\ \E c \in (1 .. numCPUs):
        /\ cpus[c] # 0
        /\ pTableLock = 0
        /\ pTableLock' = 1
        /\ scheduling' = c
        /\ procTable' = [i \in (1 .. numProcs) |->
                            IF i = cpus[c] THEN <<RUNNABLE, 0>> ELSE procTable[i]]
        /\ cpus' = [i \in (1 .. numCPUs) |->
                        IF i = c THEN 0 ELSE cpus[i]]
        /\ tlb' = [i \in (1 .. numCPUs) |->
                        IF i = c THEN 0 ELSE tlb[i]]
        /\ head' = cpus[c]
```

Scheduling

The main scheduling algorithm (round robin) is in proc.c:

```c
 void
scheduler(void)
{
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;

  for(;;){
    // Enable interrupts on this processor.
    sti();

    // Loop over process table looking for process to run.
    acquire(&ptable.lock);
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
      if(p->state != RUNNABLE)
        continue;

      // Switch to chosen process.  It is the process's job
      // to release ptable.lock and then reacquire it
      // before jumping back to us.
      c->proc = p;
      switchuvm(p);
      p->state = RUNNING;

      swtch(&(c->scheduler), p->context);
      switchkvm();

      // Process is done running for now.
      // It should have changed its p->state before coming back.
      c->proc = 0;
    }
    release(&ptable.lock);

  }
}
```

According to the code above, a new process is searched for starting from the current process that was preempted. That idea is reflected in the TLA+ spec below: the search for a new process starts from `head`. Either a process to be run is found, or isn't. Regardless, the scheduler relinquishes the proc table lock. The scheduler can be invoked once again `MagicSchedule`.

```
\* 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.
Schedule ==
    /\ scheduling # 0
    /\
        \/
            /\ \E p \in (1 .. numProcs) : procTable[p][1] = RUNNABLE
            /\ LET newProc == ChooseProc(head) IN
                /\ procTable' = [i \in (1 .. numProcs) |->
                                    IF i = newProc THEN <<RUNNING, scheduling>>
                                    ELSE procTable[i]]
                /\ tlb' = [i \in (1 .. numCPUs) |->
                            IF i = scheduling THEN newProc ELSE tlb[i]]
                /\ cpus' = [i \in (1 .. numCPUs) |->
                            IF i = scheduling THEN newProc ELSE cpus[i]]

        \/
            /\ \A p \in (1 .. numProcs) : procTable[p][1] # RUNNABLE
            /\ UNCHANGED tlb
            /\ UNCHANGED cpus
            /\ UNCHANGED procTable
    /\ scheduling' = 0
    /\ pTableLock' = 0
    /\ UNCHANGED head
```

Sleeping / Terminating

A process can sleep or complete execution. The following code runs when sleep is attempted:

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if(p == 0)
    panic("sleep");

  if(lk == 0)
    panic("sleep without lk");

  // Must acquire ptable.lock in order to
  // change p->state and then call sched.
  // Once we hold ptable.lock, we can be
  // guaranteed that we won't miss any wakeup
  // (wakeup runs with ptable.lock locked),
  // so it's okay to release lk.
  if(lk != &ptable.lock){  //DOC: sleeplock0
    acquire(&ptable.lock);  //DOC: sleeplock1
    release(lk);
  }
  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  // Tidy up.
  p->chan = 0;

  // Reacquire original lock.
  if(lk != &ptable.lock){  //DOC: sleeplock2
    release(&ptable.lock);
    acquire(lk);
  }
}
```

Once again, the proc table lock is acquired and then `sched()` is invoked. This is similar to the first few steps described in the Preemption section previously. Of course, here the state of the process is set to `SLEEPING` instead of `RUNNABLE`.

The termination of a process runs the following code (in proc.c):

```
 void
exit(void)
{
  struct proc *curproc = myproc();
  struct proc *p;
  int fd;

  if(curproc == initproc)
    panic("init exiting");

  // Close all open files.
  for(fd = 0; fd < NOFILE; fd++){
    if(curproc->ofile[fd]){
      fileclose(curproc->ofile[fd]);
      curproc->ofile[fd] = 0;
    }
  }

  begin_op();
  iput(curproc->cwd);
  end_op();
  curproc->cwd = 0;

  acquire(&ptable.lock);

  // Parent might be sleeping in wait().
  wakeup1(curproc->parent);

  // Pass abandoned children to init.
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->parent == curproc){
      p->parent = initproc;
      if(p->state == ZOMBIE)
        wakeup1(initproc);
    }
  }

  // Jump into the scheduler, never to return.
  curproc->state = ZOMBIE;
  sched();
  panic("zombie exit");
}
```

I have chosen not to model the file system operations in this spec, as my focus is the scheduler algorithm / protocol. Note the state change above to `ZOMBIE`.

Both `ZOMBIE` and `SLEEPING` are treated as `NOTRUNNABLE` in the TLA+ spec below. The proc table lock is acquired (it must not be held first), and the scheduler is invoked (see the Scheduling section).

```
\* 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 ==
    /\ \E c \in (1 .. numCPUs):
        /\ cpus[c] # 0
        /\ pTableLock = 0
        /\ pTableLock' = 1
        /\ scheduling' = c
        /\ procTable' = [i \in (1 .. numProcs) |->
                            IF i = cpus[c] THEN <<NOTRUNNABLE, 0>> ELSE procTable[i]]
        /\ cpus' = [i \in (1 .. numCPUs) |->
                        IF i = c THEN 0 ELSE cpus[i]]
        /\ tlb' = [i \in (1 .. numCPUs) |->
                        IF i = c THEN 0 ELSE tlb[i]]
        /\ head' = cpus[c]
```

Spawning New Processes

Several procedures such as spawning new processes (via `fork`) go beyond the scheduler code (therefore, I haven't modeled it). However, not allowing new processes to spawn altogether is not faithful to the working of Xv6 and the goal of this project. This project's goal is to verify the scheduler's correctness. By denying the existence of some scenarios (such as having new processes come up), we are not fully testing the scheduler. The `MagicRunnable` section below takes care of spawning.

```
\* The magic transition from NOTRUNNABLE to RUNNABLE. Choose the first available
\* procTable entry.
MagicRunnable ==
    /\ pTableLock = 0
    /\ \E p \in (1 .. numProcs) :
        /\ procTable[p][1] = NOTRUNNABLE
        /\ \A x \in (1 .. numProcs):
            \/ p <= x
            \/ procTable[x][1] # NOTRUNNABLE
        /\ procTable' = [i \in (1 .. numProcs) |->
                            IF i = p THEN <<RUNNABLE, 0>> ELSE procTable[i]]
    /\ UNCHANGED cpus
    /\ UNCHANGED tlb
    /\ UNCHANGED pTableLock
    /\ UNCHANGED scheduling
    /\ UNCHANGED head
```

Invoking the Scheduler on an Empty CPU

In the case where there are no processes to run, a CPU will run the scheduler and then relinquish the proc table lock. The below models the situation where a CPU was unable to find a process to pick up and run, and is in the transition period between releasing the proc table lock and regaining it to run the scheduler once again. The scheduler is invoked once again, but the proc table lock needs to be acquired first. The TLA+ spec below does exactly that.

```
 \* The scheduler can come alive magically, from the first idle CPU.
MagicSchedule ==
    /\ pTableLock = 0
    /\ \E c \in (1 .. numCPUs):
        /\ cpus[c] = 0
        /\ \A x \in (1 .. numCPUs): (c <= x \/ cpus[x] # 0)
        /\ head' = numProcs
        /\ scheduling' = c
    /\ pTableLock' = 1
    /\ UNCHANGED cpus
    /\ UNCHANGED tlb
    /\ UNCHANGED procTable
```

Putting It Together

Any of the transitions described above (`Preemption`, etc.) could occur, depending on the current state of the system. The `Next` transition combines the above transitions.

```
 \* One of the following actions can happen. The actions lead to processes coming
\* alive, dying, being scheduled, being preempted, etc.
Next ==
    \/ Preemption
    \/ Sleep
    \/ Schedule
    \/ MagicRunnable
    \/ MagicSchedule
```

The Invariants

This ensures that the page table is being updated correctly. A CPU running the scheduler must use the scheduler's page table (which only contains kernel address mappings), and a CPU running a process must use that particular process's page table.

```
 \* Every RUNNING process is using the right TLB.
TLBValid ==
    /\ \A c \in (1 .. numCPUs):
        \/ cpus[c] = 0 /\ tlb[c] = 0
        \/ cpus[c] # 0 /\ cpus[c] = tlb[c] /\ procTable[cpus[c]][1] = RUNNING
```

Once the CPU starts running the scheduler, it must not be associated with a process.

```
 \* The CPU running the scheduler must not have a process associated with it
SchedCPUIsFree ==
    \/ scheduling = 0
    \/ cpus[scheduling] = 0
```

The proc table lock must be held whenever scheduling is going on.

```
 \* If the scheduler is active, the pTableLock must be held.
\* scheduler active implies pTableLock is held.
SchedulerHasLock ==
    /\ (scheduling # 0) => (pTableLock = 1)
```

The invariant `NotSameProc` makes sure that no two CPUs are running the same process.

```
 \* True if there are two CPUs that map to the same process.
SameProc ==
    /\ \E i \in (1 .. numCPUs) :
        cpus[i] # 0 /\ (\E j \in (1 .. numCPUs) : i # j /\ cpus[i] = cpus[j])

\* No two CPUs should be running the same process
NotSameProc ==
    /\ ~ SameProc
```

Conclusion

I hope this guide sufficiently explains the TLA+ spec of the Xv6 scheduler. This guide should have also explained why certain decisions were made with regards to level of abstraction and atomicity. I agree that this spec is in no way a complete proof of the scheduler: I intended to verify the high level working of the scheduler, and that is what this spec has done. Perhaps in the future, I will refine this spec further (and probably end up writing a new one) that doesn't compress steps together into a single atomic step.