Proof of the example scheduler

Below we provide a proof outline for the example scheduler. We establish the judgements for schedule and create in the low-level proof system required by the proof rules of our logic and verify fork in the high-level proof system. Note that despite assertions in the proof being long, all the steps in it are purely mechanical. In fact, the data structure manipulations involved are of the kind that can be handled by automatic tools based on separation logic.1

We abbreviate FRK_FRAME to \( f \). In the proof of fork, \( F \) is the local state of the parent, \( \Sigma \) the contents of its stack and \( P \) the precondition of the newly created process (excluding a copy of the parent’s stack also passed to the child process). In the proof of load_balance, the assertion \( Q \) describes the local state of the schedule function calling it:

\[
\begin{align*}
\text{Process}([l; \text{if } 1; \text{ss, ss, sp; sp – m – 1, gr} : [\text{gr}]] & ) \\
\text{old_process = current[cpu]} & \\
\{ \text{cpu, old_process } = \text{ process}[\text{cpu}] & \\
\text{old_process.kernel_stack = ss &} \\
\text{cpu = k \land 0 \leq sp – ss – m – s – 1 \leq StackBound &} \\
\text{current[k]} & \rightarrow \text{d + d.prev} \\
\text{old_process.prev} & \rightarrow \_ * \\
\text{old_process.next} & \rightarrow \_ * \\
\text{old_process.saved_sp} & \rightarrow \_ * \\
\text{sp – s – m – 1)} & \rightarrow \_ * \\
\text{Process}([l; \text{if } 1; \text{ss, ss, sp; sp – m – 1, gr} : [\text{gr}]] & ) \\
\text{old_process = current[cpu]} & \\
\{ \text{cpu, old_process } = \text{ process}[\text{cpu}] & \\
\text{old_process.kernel_stack = ss &} \\
\text{cpu = k \land 0 \leq sp – ss – m – s – 1 \leq StackBound &} \\
\text{current[k]} & \rightarrow \text{d + d.prev} \\
\text{old_process.prev} & \rightarrow \_ * \\
\text{old_process.next} & \rightarrow \_ * \\
\text{old_process.saved_sp} & \rightarrow \_ * \\
\text{sp – s – m – 1)} & \rightarrow \_ * \\
\text{Process}([l; \text{if } 1; \text{ss, ss, sp; sp – m – 1, gr} : [\text{gr}]] & ) \\
\text{old_process = current[cpu]} & \\
\{ \text{cpu, old_process } = \text{ process}[\text{cpu}] & \\
\text{old_process.kernel_stack = ss &} \\
\text{cpu = k \land 0 \leq sp – ss – m – s – 1 \leq StackBound &} \\
\text{current[k]} & \rightarrow \text{d + d.prev} \\
\text{old_process.prev} & \rightarrow \_ * \\
\text{old_process.next} & \rightarrow \_ * \\
\text{old_process.saved_sp} & \rightarrow \_ * \\
\text{sp – s – m – 1)} & \rightarrow \_ * \\
\text{Process}([l; \text{if } 1; \text{ss, ss, sp; sp – m – 1, gr} : [\text{gr}]] & ) \\
\text{old_process = current[cpu]} & \\
\{ \text{cpu, old_process } = \text{ process}[\text{cpu}] & \\
\text{old_process.kernel_stack = ss &} \\
\text{cpu = k \land 0 \leq sp – ss – m – s – 1 \leq StackBound &} \\
\text{current[k]} & \rightarrow \text{d + d.prev} \\
\text{old_process.prev} & \rightarrow \_ * \\
\text{old_process.next} & \rightarrow \_ * \\
\text{old_process.saved_sp} & \rightarrow \_ * \\
\text{sp – s – m – 1)} & \rightarrow \_ * \\
\end{align*}
\]

void load_balance(int cpu) {
    {cpu < NCPUS ∧ Q ∗ sp... (as + StackSize - 1) → }
}
\[ Q[ap + 2 \cdot \text{sizeof}(\text{int}) - \text{sizeof}(\text{Process})/sp] * \]
\[ sp.(as + \text{StackSize} - 1) \rightarrow \]

// We deallocate local variables here
\[ \{ \text{cpu} = Q * (sp.(as + \text{StackSize} - 1) \rightarrow \} \}

\]

_reparam void create(Process *new_process) {
// Here we move the parameter from gr1 into
// the new_process local variable
\{new_process = gr1, \gamma([if]) = 1 \∧
SchedState[ap - \text{sizeof}(\text{int}) - \text{sizeof}(\text{Process})/sp] *
\text{desc}(\text{new_process}, \gamma) * \text{Process}(\gamma)\}

int cpu;
\{new_process, cpu \mapsto \gamma, \gamma([if]) = 1 \∧
SchedState[ap - \text{sizeof}(\text{int}) - \text{sizeof}(\text{Process})/sp] *
\text{new_process}.prev \mapsto *, \text{new_process}.next \mapsto _, * \}

desc(\text{new_process}, \gamma) * \text{Process}(\gamma)\}

lock(\text{runqueue}_\text{lock}[\text{cpu}]);
\}
// We deallocate local variables here
\{\text{SchedState}[ap - \text{sizeof}(\text{int}) - \text{sizeof}(\text{Process})/sp] *
\text{desc}(\text{new_process}, \gamma) * \text{Process}(\gamma)\}

int fork() {
\{0 ≤ sp - ss ≤ \text{StackBound} - f \∧
ss.(ap - 1) \mapsto * \text{Oblig} * sp.(as + \text{StackSize} - 1) \rightarrow *, F \rightarrow P \}

\}

new_process = alloc(sizeof(Process));
\{new_process \mapsto 0 ≤ sp - ss ≤ \text{StackBound} \∧
ss.(ap - 1) \mapsto * \text{Oblig} * sp.(as + \text{StackSize} - 1) \rightarrow *, F \rightarrow P \}

new_process = alloc(sizeof(Process));
\{new_process \mapsto 0 ≤ sp - ss ≤ \text{StackBound} \∧
ss.(ap - 1) \mapsto * \text{Oblig} * sp.(as + \text{StackSize} - 1) \rightarrow *, F \rightarrow P \}
new_process.timeslice → _* new_process.saved_sp → _* new_process.kernel_stack.. (new_process.kernel_stack + sp − ss − f − 1) → Σ0g * (new_process.kernel_stack + sp − ss − f).. (new_process.kernel_stack + StackSize − 1) → _*}

new_process->saved_sp = new_process->kernel_stack+ _sp−ss+FORK_FRAME+SCHED_FRAME;
{
new_process ⊩ 0 ≤ sp−ss ≤ StackBound ∧ ss..(sp−f−1) → Σ0g * sp.(ss+StackSize−1) → _* F*P*
new_process.prev → _* new_process.next → _*
new_process.timeslice → _*
new_process.saved_sp → new_process.kernel_stack + sp − ss − f − s *
new_process.kernel_stack..
(new_process.kernel_stack + sp − ss − f − 1) → Σ0g * (new_process.kernel_stack + sp − ss − f), (new_process.kernel_stack + StackSize − 1) → _*}

{new_process ⊩ 0 ≤ sp−ss ≤ StackBound ∧ ss..(sp−f−1) → Σ0g * sp.(ss+StackSize−1) → _* F*P*
∃γ.γ(iip) = 1 ∧ γ(⃗gr) = ⃗g ∧ γ(ss) = new_process.kernel_stack ∧ γ(sp) = new_process.kernel_stack + sp − ss − f − m − 1 ∧ γ(if) = 1 ∧ desc(new_process, γ) ∧ γ(sp)−1) → Σ0}

// We assume P satisfies the premis of the Create rule _icall create(new_process);
{
new_process ⊩ 0 ≤ sp−ss ≤ StackBound ∧ ss..(sp−f−1) → Σ0g * sp.(ss+StackSize−1) → _* F*
}

// We deallocate local variables here
{0 ≤ sp−ss−f ≤ StackBound ∧ ss..(sp−1) → Σ0g * sp.(ss+StackSize−1) → _* F}
return 1;
}