1 Modeling an Operating System Kernel

Egon Börger, Università di Pisa, Dipartimento di Informatica
Iain Craig, University of Northhampton, Faculty of Applied Sciences

Zusammenfassung. We define a high-level model of an operating system (OS) kernel which can be refined to concrete systems in various ways, reflecting alternative design decisions. We aim at an exposition practitioners and lecturers can use effectively to communicate (document and teach) design ideas for operating system functionality at a conceptual level. The operational and rigorous nature of our definition provides a basis for the practitioner to validate and verify precisely stated system properties of interest, thus helping to make OS code reliable. As a by-product we introduce a novel combination of parallel and interruptable sequential Abstract State Machine steps.

1.1 Introduction

We show how to develop from scratch an easily understandable, accurate high-level view of an OS kernel. The model we propose is algorithmic in nature and can be understood without prior knowledge of formal methods. It can be used by lecturers for teaching the principles of OS design and by practitioners for experiments in OS design (they can realise alternative design decisions by refining the abstract model). In addition, the mathematically accurate character of our operational model permits validation (by simulation) and verification (by proof) of the behavioral properties of such kernels.

We base our work on the recent book [Cra07a], a study of formal models of operating systems kernels which comes with a companion book on the refinement of such models [Cra07b]. The two books use Z [Spi92], and sometimes Object-Z [Smi00], as well as CCS [Mil89] as their description languages. In order to make our model understandable by readers without knowledge of formal specification languages, we use pseudo-code descriptions written as Abstract State Machines (ASMs). We introduce a novel combination of parallel and interruptable sequential ASM steps. ASMs guarantee that our descriptions have a mathematically precise meaning (given by the semantics of ASMs [BS03]); this accurately and directly supports the intuitive operational understanding of the pseudo-code.

Using ASMs also implies that the model can be seamlessly refined to code, a process that can be difficult when starting from purely axiomatic Z specifications [Hal97]. Refinements of ASMs to code can be performed either by programming (e.g., [Mea97, BBD+96]) or compiling to executable code (e.g., [BPS00]).

Our OS kernel model mainly follows the swapping kernel in [Cra07a, Ch.4], which captures the essentials of the MINIX Kernel [Tan87]. As a by-product, the ASM model defined here and the Z model in [Cra07a, Ch.4] can be used in a concrete comparison of the two specification methods.

The kernel is organized as a layered architecture. At the bottom is the hardware. It is linked, via interrupt service routines (ISRs) to a scheduler. It also connects to layers of (in priority order) communicating device, system and user processes. It uses a clock to implement alarms and a storage-management scheme including a swapping mechanism for storing active processes on disk. The link between the hardware and the software model is a collection of hardware-controlled locations monitored by the software model. The kernel model is a collection, inter alia, of interacting components such as the clock process (and associated driver and ISR), the swapper, the scheduler. Due to space limitations, we focus on the clock interrupt but the method is general and can also be used for other kinds of interrupt.

1.2 Clock Interrupt Service Routine

An OS Kernel consists of various interacting components. Central to the kernel is the clock process. It interacts with priority-based scheduling of device, system and user processes that pre-empts user-defined processes. It controls the timing of the swapping of active user processes between main store and a disk; it also implements alarms that wake sleeping processes when their sleep period has expired. The main program of this component is the clock interrupt service routine CLOCKISR. It is triggered by hardware clock ticks, here formalized by a monitored predicate *HwClockTick*. Ticks are assumed to occur at regular intervals, whose length is expressed abstractly by the constant function, *ticklength*.[1]

Upon a *HwClockTick*, one execution round of the clock interrupt service routine is triggered. It consists of three successive steps of CLOCKISR:

- DESCHEDULE the current process *currp*, changing its *status* from *running* to *ready* and saving its current state. In case *currp* is a user process that had *RunTooLong*, it is moved from the head to the tail of the queue of user pro-

[1] Some hardware clock ticks can be missed, due to locking. Thus the system time *now* reflects the relative time resulting from the perceived hardware clock ticks.

cesses that are ready to execute (thus implementing a round robin scheme).

- DRIVETIMEDFEATURES. This decomposes into:
  - Update the current system time *now*;
  - Wake up the CLOCKDRIVER component which performs further timing updates related to process suspension and storage management. Then trigger the SWAPPER, a storage management process using a time criterion to swap user processes between main store and disk;
  - Wake up the DEZOMBIFIER which is related to swapping.
- RESCHEDULE: schedule the next *currp* (which may be the interrupted process DESCHEDULEd in the first step of CLOCKISR). It is selected from the processes ready to execute and its state is restored.

We assume the *HwClockTick* event to be preemptive (read: the monitored predicate becomes false once the triggered rule has fired).

CLOCKISR = **if** *HwClockTick* **then**
DESCHEDULE **step** [2] DRIVETIMEDFEATURES **step** RESCHEDULE

We will describe the concepts involved in the description of CLOCKISR (*Process*es of various kinds, the current process *currp* ($\in$ *Process*), scheduler and time and storage management) in the following sections.

1.2.1 Defining the Submachines of CLOCKISR

The dynamic set *Process* of processes is divided into three disjoint subsets representing three distinct kinds of processes with different scheduling level *schedlev*: user, system and device processes. There is an additional special process: *idleProcess*.

$$Process = DeviceProcess \cup SystemProcess \cup UserProcess \cup \{idleProcess\}$$
$$RealProcess = Process \setminus \{idleProcess\}$$

The currently executing process *currp* can be thought of as the only one with $status(currp) = running$ (read: instruction pointer *ip* pointing into its code); it is selected by the scheduler from a queue, *readyq*, of processes whose *status* is *ready*. Each process has a current state which is saved when the process is DESCHEDULEd and restored when it is RESCHEDULEd. The two submachines SAVESTATE and RESTORESTATE are detailed in Sect. 1.6.

[2] **step** denotes an interruptable sequential composition of ASMs, defined in Sect. 1.8, to be distinguished from the atomic sequential composition denoted by **seq** and defined in [BS03, Ch.4.1]

When the current process is DESCHEDULEd, besides saving its state, its *status* is changed from *running* to *ready*. If *currp* is a user process, its time quantum is updated. This means that a location, $timeQuant(currp)$, is decremented, followed by a check whether *currp* has consumed the assigned time quantum and must therefore be removed from the processor. If *currp* has *RunTooLong*, it is removed from the head of the queue $readyq(schedlev(currp))$[3] of all *ready* processes of its scheduler level that can be be chosen by the scheduler for execution. This is where it was when the scheduler selected it as *currp* (but did not remove it from this queue). When *currp* is returned to the queue, it is placed at the end. [4]

Locking Mechanism. Since the synchronous parallelism of simultaneously executing all applicable rules of an ASM, M, implies the atomicity of each single M-step, at this level of abstraction, we need no other locking techniques. If, in further refinement steps which map the simultaneous one-step execution of different submachines to a sequence of single machine steps, a locking mechanism is required, we indicate this at the top level by a pair of brackets $(M)_{Lck}$. Formally this stands for the execution of M to be preceded by an execution of an appropriate LOCK and to be followed by an UNLOCK machine, using the atomic **seq**uential composition of ASMs (see [BS03, Ch.4.1]):

$$(M)_{Lck} = \text{Lock } \textbf{seq } M \textbf{ seq } \text{UnLock}$$

$$\text{DeSchedule} = \text{SaveState } \textbf{seq} \left(\begin{array}{l} status(currp) := ready \\ \text{HandleProcessQuantum}(currp) \end{array} \right)_{Lck}$$

where

HANDLEPROCESSQUANTUM(p) =
  if $p \in UserProcess$ **then**
    $timeQuant(p) := timeQuant(p) - 1$
    if $RunTooLong(p)$ **then**
      REMOVEHEAD($readyq(schedlev(p))$) **seq**
        ENQUEUE($p, readyq(schedlev(p))$) // insertion at the end

[3] The removal is needed because SCHEDULENEXT selects a ready process to become the new *currp* but does not remove that process from the ready queue. The definition of *OnInterrupt* in [Cra07a, p.177] uses MAKEREADY(p), but does not include the removal from the head of the queue.

[4] This deviates from the definition of *UpdateProcessQuantum* in [Cra07a, p.133] and from its use in the clock driver run process [Cra07a, p.184]. If *UpdateProcessQuantum* is called only by the clock driver run process (which is signaled by the *ServiceISR* of the *CLOCKISR*), the process which was *currp* when the interrupt occurred and should be subject to HANDLEPROCESSQUANTUM has already been descheduled by the *ServiceISR* of the *CLOCKISR* [Cra07a, p.177] and is not current any more so that *UpdateProcessQuantum* does not apply to it, but to the clock driver process (for which it would have no effect because the clock driver run process is not a user process).

$$RunTooLong(p) = (timeQuant(p) - 1 \leq minUserTimeQuant)$$

Similarly, RESCHEDULE involves saving the value of *currp* in a location *prevp*, letting the scheduler determine the new value for *currp* and to RESTORESTATE of the selected process. When no process is ready, the *idleProcess* is scheduled for execution. Otherwise, a *ready* process is selected by the scheduler and is made the new *currp* (with *status running*). We treat the scheduler level of *currp* as a derived location defined by $currplev = schedlev(currp)$.

We recall that, by selecting a new element in SCHEDULENEXT making it the *currp*process, this element is not removed from the *readyq*. The removal may be done when the process is DESCHEDULEd, as explained above.

RESCHEDULE = SCHEDULENEXT **seq** RESTORESTATE(*currp*)[5]
where SCHEDULENEXT =
  $prevp := currp$ // record *currp* as previous current process
  let $p = select_{LowLevelScheduler}(readyq)$
    if $p =$ **undef** // nothing to select since *readyq* is empty
    then $currp := idleProcess$ **else**
      $currp := p$
      $status(p) := running$

DRIVETIMEDFEATURES updates the system time *now* by adding *ticklength* to it and triggers driver processes CLOCKDRIVER and DEZOMBIFIER, in that order. CLOCKDRIVER updates the various time counters related to process suspension and swapping, readies (or "alarms") the processes that are to be resumed and wakes up the SWAPPER process. DEZOMBIFIER kills all zombie processes which by (the updated value of) *now* remain without children processes. Both processes are detailed below. The triggering macros WAKE(*ClockDriver*) and WAKE(*DeZombifier*) are defined in terms of semaphore SIGNALing, which we define in Sect. 1.7, together with the corresponding semaphore mechanism to PUTTOSLEEP(*device*), which is defined in terms of the semaphore WAIT operation. At this point it suffices to be aware that, for a device process *p*, when the device semaphore is signaled, MAKEREADY(*p*) is called, whereas WAIT calls MAKEUNREADY(*p*).

$$\text{MAKEREADY}(p) = \begin{pmatrix} \text{ENQUEUE}(p, readyq(schedlev(p))) \\ status(p) := ready \end{pmatrix}_{Lck}$$

MAKEUNREADY(*r*) =
  DELETE(*r*, *readyq*(*schedlev*(*r*)))
  if $r = head(readyq)$ **then** RESCHEDULE

[5] The use of **seq** could be avoided here by including RESTORESTATE(*p*) into SCHEDULENEXT.

We assume a unique semaphore *device_sema* for each *device*.

$$
\begin{array}{l}
\text{DRIVETIMEDFEATURES} = \\
\quad now := now + ticklength \\
\quad \text{WAKE}(clockDriver)\ \ \mathbf{seq}\ \ \text{WAKE}(deZombifier)
\end{array}
$$

1.3 The CLOCKDRIVER Component

The role of the CLOCKDRIVER routine is twofold, based on the new value of system time *now* which was previously updated during execution of CLOCKISR.

- UPDATESTORAGETIMES deals with the time a process, p, has been main-store ($residencyTime(p)$) or swap-disk resident ($swappedOutTime(p)$). As a consequence WAKE($swapper$) calls SWAPPER for the updated time values.
- RESUMEALARMEDPROCESSES in case there are suspended real processes whose waiting time has elapsed by *now*, i.e. processes, p, with a defined sleeping time $alarmTime(p)$ which no longer exceeds the system time, *now*. RESUMEALARMEDPROCESSES cancels these $alarmTime(p)$ (by making them undefined) and calls MAKEREADY(p) so that p's execution can continue.

CLOCKDRIVER is assumed to be initialized as sleeping by executing operation *clockDriver_sema*.WAIT, which is also performed each time the clock driver has finished one of its runs and is PUTTOSLEEP.

$$
\text{CLOCKDRIVER} = \begin{array}{l}
\left(\begin{array}{l}
\text{UPDATESTORAGETIMES} \\
\text{WAKE}(swapper) \\
\text{RESUMEALARMEDPROCESSES}
\end{array}\right)_{Lck} \\
\text{PUTTOSLEEP}(clockDriver)
\end{array}
$$

On every clock tick, UPDATESTORAGETIMES increments the time that each process has been main-store or swap-disk resident.[6] It is assumed that a process that is not marked as swapped out is resident in main store.

$$
\begin{array}{l}
\text{UPDATESTORAGETIMES} = \mathbf{forall}\ p \in RealProcess \\
\quad \mathbf{if}\ status(p) = swappedout \\
\quad\quad \mathbf{then}\ swappedOutTime(p) := swappedOutTime(p) + 1 \\
\quad\quad \mathbf{else}\ residencyTime(p) := residencyTime(p) + 1
\end{array}
$$

[6] For the reasons explained in Sect. 1.2.1 we have transfered the submachine HANDLEPROCESSQUANTUM [Cra07a, p.184] from CLOCKDRIVER to DESCHEDULE.

RESUMEALARMEDPROCESSES =
  let $alarmed = \{p \in RealProcess \mid alarmTime(p) \leq now\}$
  forall $p \in alarmed$
    $alarmTime(p) :=$ **undef**
    MAKEREADY(p)

1.4 The DEZOMBIFIER Component

The DEZOMBIFIER process counts as a driver process. Its execution is triggered using the *deZombifier_sema*phore. It is assumed that it is initialized as sleeping using *deZombifier_sema*.WAIT. It handles the dynamic set, *zombies*, of so-called zombie processes, i.e. processes, p, with $status(p) = zombie$, which have almost terminated but could not release their storage due to their sharing code with their children processes, some of which up to *now* have not yet terminated. It is necessary, in an atomic action, to delete from *zombies* all those elements which remain without child processes, canceling these 'dead' zombies as children of their parent processes (if any). We consider *children* as derived from the *parent* function by $children(p) = \{q \mid parent(q) = p\}$.

DEZOMBIFIER = (KILLALLZOMBIES)$_{Lck}$
  PUTTOSLEEP(*deZombifier*)
where KILLALLZOMBIES =
  let $deadzs = \{z \in zombies \mid children(z) = \emptyset\}$
    $zombies := zombies \setminus deadzs$
    forall $z \in deadzs$ $parent(z) :=$ **undef**

1.5 The SWAPPER Component

SWAPPER swaps user processes between main store and disk memory, so that more processes can be in the system than main store alone could otherwise support. A process is swapped from disk when its value of *swappedOutTime* is the maximum of the *swappedOutTime*s of all processes currently on disk. SWAPPER counts as a driver process. It is assumed that it is initialized in a sleeping state (by execution of *swapper_sema*.WAIT). The *swapper_sema*phore is used to restart the swapper using the WAKE(*swapper*) operation. SWAPPER is suspended again after one round of DISKSWAPs.

SWAPPER = DISKSWAP **step** PUTTOSLEEP(*swapper*)

1.5.1 The DISKSWAP Component

DISKSWAP uses a function, *nextProcessToSwapIn*, to determine the next process, p, that is to be swapped. If p exists, it is the process with maximum value of *swappedOutTime*. DISKSWAP then computes the memory size of the process, $memSize(p)$, and checks whether the system *CanAllocateInStore* the requested memory space (s). If yes, the machine executes SWAPPROCESSINTOSTORE(p, s); otherwise, it determines the $swapOutCandidate(s)$ with the requested memory size, s. If such a *cand*idate exists, SWAPPROCESSOUT$(cand, mem(cand))$ frees the main store region, $mem(cand)$, occupied by *cand*. The machine will then use the freed memory region and perform SWAPPROCESSIN(p, s).

DISKSWAP $=$ **if** $nextProcessToSwapIn \neq$ **undef then**
  let $p = nextProcessToSwapIn$
  let $s = memSize(p)$ // determine needed process memory size
    if $CanAllocateInStore(s)$ // enough free space in main store?
    then START(SWAPPROCESSINTOSTORE(p, s))[7]
    else let $cand = swapOutCandidate(s)$
      if $cand \neq$ **undef then**
        START(SWAPPROCESSOUT$(cand, mem(cand))$
          step SWAPPROCESSINTOSTORE(p, s))
  where [8]
    $nextProcessToSwapIn =$
      $\iota p\ (swappedOutTime(p) =$
        $max\{swappedOutTime(q) \mid status(q) = swappedout\})$
    $swapOutCandidate(s) = \iota p \in UserProcess$ **with**
      $status(p) = ready$
      $memSize(p) \geq s$
      $residencyTime = max\{residencyTime(q) \mid q \in UserProcess$ **and**
        $status(p) \neq swappedout\}$

This definition implies that no swap takes place if there are no swapped-out processes (and thus there are no processes that can be swapped in) or if the next process to be swapped out (the one with the greatest main-store residency time) does not make the requested main-store space available.

---

[7] START is defined in Sect. 1.8.

[8] Hilbert's ι operator denotes the unique element with the indicated property, if there is one; otherwise its result is **undef**ined.

1.5.2 Storage Management Background

The storage management used by DISKSWAP works on an abstract notion of main *mem*ory: a sequence of *P*rimary *S*torage *U*nits (e.g. bytes or words), i.e. $mem : PSU^*$. It is assumed that each process, p, occupies a contiguous subsequence (called a *memory region*) of *mem*, starting at address $memStart(p)$, of size $memSize(p)$ and denoted $mem(p) = (memStart(p), memSize(p))$. Thus the storage area of p is

$$[mem(memStart(p)), \ldots, mem(memStart(p) + memSize(p) - 1)].$$

The entire main store is considered to be partitioned into a) memory regions occupied by user processes and b) free memory regions. The former are denoted by a set, *usermem*, of *RegionDescr*iptions, $(start, size)$, with start address *start* and length *size*. The latter are denoted by the set $holes \subseteq RegionDescr$.

These concepts allow us to define what, for a given memory region of size s, $CanAllocateInStore(s)$ means: namely that there is a memory hole $h \in holes$ of that size. The computation of this predicate must be protected by a pair of LOCK and UNLOCK machines.

$$CanAllocateInStore(s) = \textbf{forsome } h \in holes\ s \leq size(h)$$

SWAPPROCESSINTOSTORE(p, s) will first ALLOCATEFROMHOLE(s) a hole, h, of sufficient size and use it as the main-store region in which to REQUESTSWAPIN of p, starting at $start(h)$.[9] The request is put into the *swapReqBuff*er of the swap disk driver process, SWAPDISKDRIVER (defined below), which will SIGNAL the *swapDiskDriver_donesema*phore when the requested disk-to-main-store transfer operation has been completed. Then SWAPPROCESSINTOSTORE(p, s) can update the attributes of the newly swapped-in process (its base address, in the relocation register, *memStart*,[10] *status*, *residencyTime*, *swappedOutTime*[11]) and then insert it into the scheduler's queue using MAKEREADY(p).

SWAPPROCESSINTOSTORE(p, s) also invokes READYDESCENDANTS if there are process descendants sharing code the process owns. In fact, when p is swapped out, all its descendants are suspended and placed in the set $blockswaiting(p)$ (see below). Once the parent is swapped in again, all of its children become ready to execute since the code they share has been reloaded into main store (it is supposed to be part of the parent's memory region).

---

[9] We succinctly describe this sequence with the ASM construct **let** $y = M(a)$ **in** N, defined in [BS03, p.172].

[10] There is no need to update $memSize(p)$ since it is known to be s, the size of $p = nextProcessToSwapIn$.

[11] Resetting $swappedOutTime(p)$ to 0 prevents it from being considered when looking for future *swapOutCandidate*s.

```
SwapProcessIntoStore(p, s) =
  let h = AllocateFromHole(s) in memStart(p) := start(h)
  step RequestSwapIn(p, memStart(p))
  step swapDiskDriver_donesema.Wait
  step
    Delete(p, SwappedOut)
    UpdateRelocationReg(p) // update process base address
    residencyTime(p) := 0
    swappedOutTime(p) := 0
    MakeReady(p)
  step if children(p) ≠ ∅ and IsCodeOwner(p) then
    ReadyDescendants(p)
where
  ReadyDescendants(p) = forall q ∈ blockswaiting(p)
    Delete(q, blockswaiting)
    MakeReady(q)
```

RequestSwapIn uses a *swapDiskMsg_sema*phore to ensure exclusive access to the *swapReqBuff*er. After writing the request to the buffer, the machine wakes up the SwapDiskDriver (defined below) to handle the request.

```
RequestSwapIn(p, loadpt) =
  swapDiskMsg_sema.Wait
  step swapReqBuff := SWAPIN(< p, loadpt >)
  step
    swapDiskMsg_sema.Signal
    Wake(swapDiskDriver)
```

AllocateFromHole chooses a hole, *h*, of the requested size. When the machine is called, there is at least one such hole and typically the first appropriate hole in *mem* is chosen. It places the region $(start(h), s)$ into *usermem*; it is also assigned to the output location **result**. It deletes *h* from the set of *holes* and inserts the new hole $(start(h)+s, size(h)-s)$ provided that $size(h)-s>0$.

```
AllocateFromHole(s) =
  ( choose h ∈ holes with s ≤ size(h)
    Insert((start(h), s), usermem)
    result := (start(h), s)
    Delete(h, holes)
    if size(h) − s > 0 then Insert((start(h) + s, size(h) − s), holes) )_Lck
```

SWAPPROCESSOUT(p, st, sz) will SWAPPROCESSOUTOFSTORE(p, st, sz); if p has children it must also BLOCKDESCENDANTS(p). The reason for blocking the children is that they share the code of their parent. Therefore, when a parent process is swapped out, one has to MAKEUNREADY its children (and transitively their children, etc.) since their code is no longer in store but on disk. All descendants of the process are put into a set *blockswaiting*(p), their *status* is updated to *waiting*.[12] The descendants form the transitive closure $child^+$ of the *child* function.

SWAPPROCESSOUT$(p, st, sz) =$
  SWAPPROCESSOUTOFSTORE(p, st, sz)
  step if $children(p) \neq \emptyset$ **and** *IsCodeOwner*(p) **then**
    BLOCKDESCENDANTS(p)
where
  BLOCKDESCENDANTS$(p) =$ **forall** $q \in child^+(p)$
    INSERT$(q, blockswaiting(p))$
    $status(q) := waiting$
    MAKEUNREADY(q)

SWAPPROCESSOUT(p, st, sz) and SWAPPROCESSINTOSTORE(p, sz) are inverse operations. The former performs REQUESTSWAPOUT of p's memory region (starting at *st* upto $st + sz$) by storing the request in the *swapReqBuffer* of the SWAPDISKDRIVER. It inserts p into *SwappedOut* and updates its attributes (its *status*, *swappedOutTime* and *residencyTime*[13]). It uses FREEMAINSTORE(st, sz) to delete the memory region to be swapped out from *usermem* (of size *sz*, starting at *st*) and to insert it into *holes*, merging any adjacent holes that result. Finally it must MAKEUNREADY(p).

SWAPPROCESSOUTOFSTORE$(p, st, sz) =$
  REQUESTSWAPOUT$(p, st, st + sz)$
  step *swapDiskDriver_donesema*.WAIT[14]
  step
    INSERT$(p, SwappedOut)$
    $status(p) := swappedOut$

[12] This leaves the case open that a child may be waiting for a device request completion and not in the *readyq* when its parent is swapped out, so that really it cannot immediately be stopped.

[13] Setting the *residencyTime* to 0 prevents it from being considered when looking for a future *nextProcessToSwapIn*.

[14] In op.cit., p.188, this protection does not appear as part of *swapProcessOut*. We include it to guarantee that before the attributes of the process to be swapped out are updated, the process has actually been swapped out, so that no interference is possible with the subsequent *swapReqBuff* value of SWAPPROCESSINTOSTORE.

$residencyTime(p) := 0$
$swappedOutTime(p) := 0$
FREEMAINSTORE(st, sz)
MAKEUNREADY(p)

The submachines of SWAPPROCESSOUTOFSTORE are defined as follows:

FREEMAINSTORE$(region) =$
(FREEMAINSTOREBLOCK$(region)$ **seq** MERGEADJACENTHOLES)
where
FREEMAINSTOREBLOCK$(region) =$ DELETE$(region, usermem)$
INSERT$(region, holes)$
MERGEADJACENTHOLES = **forall** $h_1, h_2 \in holes$[15]
if $start(h_1) + size(h_1) = start(h_2)$ **then**
DELETE$(h_1, holes)$
DELETE$(h_2, holes)$
INSERT$((start(h_1), size(h_1) + size(h_2)), holes)$

REQUESTSWAPOUT(p, s, e) uses a *swapDiskMsg_sema*phore to ensure exclusive access to *swapReqBuff*, like REQUESTSWAPIN. It writes the request to swap out the main store region of p (the region between the start and end values s, e) and wakes up SWAPDISKDRIVER to handle the request.

REQUESTSWAPOUT$(p, s, e) =$
swapDiskMsg_sema.WAIT
step $swapReqBuff := SWAPOUT(< p, s, e >)$
step
swapDiskMsg_sema.SIGNAL
WAKE(*swapDiskDriver*)

SWAPDISKDRIVER is assumed initially to wait on *swapDiskDriver_sema*. When the semaphore is signaled, it READs the *swapReqBuff*er, which holds the code for data transfer operations the swapper requests the disk to perform. It performs HANDLEREQUEST (if the operation is not the empty *NullSwap*). It signals its *donesema*phore before it suspends on *swapDiskDriver_sema*. HANDLEREQUEST performs the requested data transfer (between main store *mem* and disk memory *dmem*), process deletion or creation (with a given process image) on the disk. The *swapReqBuff*er is cleared when it is READ.

[15] A similar but slightly more complex machine is needed when three or more consecutive holes $h_1, h_2, h_3, \ldots$ may occur which are pairwise (h_1, h_2), (h_2, h_3), ... adjacent.

$$
\begin{array}{l}
\text{SWAPDISKDRIVER} = \\
\quad \textbf{let } rq = \text{READ}(swapReqBuff) \\
\quad\quad \textbf{if } rq \neq NullSwap \textbf{ then} \\
\quad\quad\quad \text{HANDLEREQUEST}(rq) \\
\quad\quad\quad swapDiskDriver_donesema.\text{SIGNAL} \\
\quad\quad \text{PUTTOSLEEP}(swapDiskDriver) \\
\textbf{where} \\
\quad \text{HANDLEREQUEST}(rq) = \textbf{case } rq \textbf{ of} \\
\quad\quad \begin{array}{lll}
SwapOut(p, start, end) & \rightarrow & dmem(p) := [mem(start), \ldots, mem(end)] \\
SwapIn(p, ldpt) & \rightarrow & \textbf{forall } ldpt \leq i < ldpt + memSize(dmem(p)) \\
 & & mem(i) := dmem(p)(i) \\
DelProc(p) & \rightarrow & dmem(p) := \textbf{undef} \\
NewProc(p, img) & \rightarrow & dmem(p) := img
\end{array}
\end{array}
$$

The swap request buffer READ and WRITE operations are defined using a semaphore *swapDiskMsg_sema* which provides the necessary synchronization between the swap disk process and the swapper process. [16]

$$
\text{WRITE}(rq) = \begin{array}{l}
swapDiskMsg_sema.\text{WAIT} \\
\textbf{step } swapReqBuff := rq \\
\textbf{step } swapDiskMsg_sema.\text{SIGNAL}
\end{array}
$$

$$
\begin{array}{l}
\text{READ} = \\
\quad swapDiskMsg_sema.\text{WAIT} \\
\quad \textbf{step} \\
\quad\quad \textbf{result} := swapReqBuff \\
\quad\quad swapReqBuff := Nullswap \\
\quad \textbf{step } swapDiskMsg_sema.\text{SIGNAL}
\end{array}
$$

1.6 Scheduling and State Handling

A standard specialization of SCHEDULENEXT comes as a data refinement of the $select_{LowLevelScheduler}$ function to select $head(readyq)$. Selecting a process should respect the priorities of the three kinds of processes and apply the FIFO principle within each kind. To this end, device processes, p, are declared to have highest priority (lowest $schedlev(p) = 1$) and user processes the lowest priority (so

[16] In [Cra07a, p.160] a more complex scheme is used for READing, where, in order to guarantee mutual exclusion, another semaphore *swapDiskBuff_mutex* is used to protect the access to *swapDiskMsg_sema*.

highest $schedlev(p) = 3$). Each subqueue is managed as a FIFO queue (a Round-Robin scheduler), refining for these queues both ENQUEUE = INSERTATEND and DEQUEUE = REMOVEHEAD. Thus *readyq* is a derived location: the concatenation of the three $readyq(i)$ for device, system and user processes $(i = 1,2,3)$. They are concatenated in priority order, so that when selecting the head of *readyq*, the scheduler always chooses a ready process with highest priority.

$$readyq = readyq(1).ready(2).readyq(3)$$

$$select_{LowLevelScheduler}(readyq) = \begin{cases} head(readyq) & \textbf{if } readyq \neq [] \\ \textbf{undef} & \textbf{else} \end{cases}$$

1.6.0.1 Defining SAVESTATE and RESTORESTATE.

SAVESTATE copies the current processor (hardware) frame, composed of the ‘state’ of *currp* consisting of the instruction pointer *ip*, a set *regs* of registers, the *stack*, the status word *statwd*, which are implicitly parameterized by a processor argument *hw*, to the process *currp* (read: its description in the process table).[17]

SAVESTATE = **if** *currp* ≠ *idleProcess* **then**
  $ip(currp) := ip$
  $regs(currp) := regs$
  $stack(currp) := stack$
  $statwd(currp) := statwd$

RESTORESTATE is the inverse operation. It installs the new current processor frame from the one stored in the process description, whereas the *idleProcess* has no stack and has empty registers and a cleared status word.

RESTORESTATE =
  if *currp* ≠ *idleProcess* **then**
    $ip := ip(currp)$
    $regs := regs(currp)$
    $stack := stack(currp)$
    $statwd := statwd(currp)$
  else
    $ip := idleProcessStartPoint$
    $regs := nullRegs$
    $stack := nullStack$
    $statwd := clearStatWord$

[17] We suppress here the *timeQuant* location because we use only its process description version $timeQuant(p)$. See rule HANDLEPROCESSQUANTUM in Sect. 1.2.1.

1.7 Semaphores

Semphores are described in detail in operating systems texts (e.g., [TAN]). They are composed of a counter, *semacount*, and a queue of processes waiting to enter the critical section, *waiters*. The semaphone counter is initialized to the value *allowed*, the number of processes simultaneously permitted in the critical section. The increment and decrement operations performed by SIGNAL and WAIT must be atomic, hence the use of LOCK and UNLOCK pairs.

To access the critical section, WAIT must be executed and SIGNAL is executed to leave it. WAIT subtracts 1 from *semacount*; SIGNAL adds 1 to it. As long as *semacount* (initialized to $allowed > 0$) remains non-negative, nothing else is done by WAIT and the *currp*rocess can enter the critical section. If *semacount* is negative, at least *allowed* processes are currently in the critical section (and have not yet left it). Therefore, if $semacount < 0$, the *currp*rocess must be added to the set of *waiters*, processes waiting on the semaphore. It is unreadied and its state is saved; its status becomes *waiting*.

$$\text{WAIT} = \begin{pmatrix} \textbf{let}\ newcount = semacount - 1 \\ \quad semacount := newcount \\ \quad \textbf{if}\ newcount < 0\ \textbf{then} \\ \quad\quad \text{ENQUEUE}(currp, waiters)\ \text{// insert at the end} \\ \quad\quad status(currp) := waiting \\ \quad\quad \text{SAVESTATE}(currp) \\ \quad\quad \text{MAKEUNREADY}(currp) \end{pmatrix}_{Lck}$$

The SIGNAL operation adds one to *semacount*. If, after this addition, *semacount* is still not positive, *waiters* contains processes. The one which first entered *waiters* is removed and leaves the critical section; it is made ready. Otherwise only the addition of 1 is performed.

$$\text{SIGNAL} = \begin{pmatrix} \textbf{let}\ newcount = semacount + 1 \\ \quad semacount := newcount \\ \quad \textbf{if}\ newcount \leq 0\ \textbf{then} \\ \quad\quad \textbf{let}\ cand = head(waiters) \\ \quad\quad\quad \text{MAKEREADY}(cand) \\ \quad\quad\quad \text{DELETE}(cand, waiters) \end{pmatrix}_{Lck}$$

We define WAKE and PUTTOSLEEP for the semaphore associated with each *device* with $allowed = 1$ as follows. The typical assumption is that the device is initialized by a WAIT use (omitting the critical section).

$\text{WAKE}(device) = device_sema.\text{SIGNAL}$
$\text{PUTTOSLEEP}(device) = device_sema.\text{WAIT}$

1.8 Appendix. The step Mechanism for ASMs

In contrast to the **seq** mechanism for sequential substeps of atomic turbo ASM steps, as defined in [BS03, Ch.4.1], the **step** mechanism we define here provides a form of non-atomic, interruptable sequential ASM execution that can be smoothly integrated into the basic synchronous parallelism of standard ASMs. The definition uses the concept of control states (also called internal states) known from Finite State Machines (FSMs). A step is considered to consist of the execution of an atomic machine in passing from a source to a target control state. An interruption is allowed to happen in each control state (or more generally in each control state belonging to a specified subclass of control states); when the interrupted machine is readied again, it continues in the control state in which it had been interrupted.[18]

The definition of what one might call *stepped* ASMs starts from control state ASMs as defined in [BS03, p.44], namely ASMs whose rules are all of the form shown in Fig. 1.1.

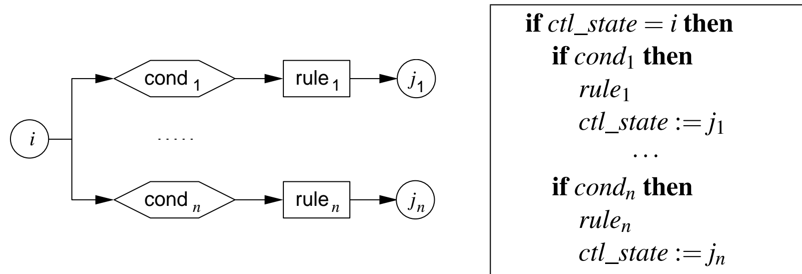

Abbildung 1.1: Control state ASMs: flowchart and code

A stepped ASM is defined as a control state ASM, with rules as in Fig. 1.1, such that all submachines $rule_j$ ($j \in \{j_1, \ldots, j_n\}$) are $step-free$, i.e. contain no *ctl_state* update. [19] The additional step-freeness condition guarantees the atomicity of what is considered here as 'step' in passing from a source control state i to a target

[18] This definition is inverse to the one Lamport has adopted for the $+CAL$ language [Lam08], where atomicity is grafted upon the basic sequential execution paradigm. See also [AB09].

[19] For example turbo ASMs are step-free.

control states j; in other words the execution of any $rule_j$ in a given state terminates in one 'step' if the update set of $rule_j$ is defined in this state (along the standard definition of the semantics of ASM rules, see [BS03, Table 2.2]), otherwise the step is not defined.[20]

Notation. We denote for a stepped ASM M the start ctl_state value by $start(M)$; where needed we denote the end ctl_state value by $end(M)$.

$$\text{START} = (ctl_state := start(M))$$

The following notation hides the control states underlying a stepped ASM.

$$M_1 \textbf{ step} \ldots \textbf{step } M_n = \textbf{case } ctl_state \textbf{ of}$$
$$start(M_i) \textbf{ with } i < n : \begin{cases} M_i \\ ctl_state := start(M_{i+1}) \end{cases}$$
$$start(M_n) : \begin{cases} M_n \\ ctl_state := end(M_n) \end{cases}$$

where usually every M_i is step-free. However, we also use the following notational short form (flattening out the steps in submachine definitions):

M **step** $N =$
  M_1 **step** $\ldots$ **step** M_m **step** N_1 **step** $\ldots$ **step** N_n
where
  $M = M_1$ **step** $\ldots$ **step** M_m
  $N = N_1$ **step** $\ldots$ **step** N_n

The above definition of stepped ASMs also covers the use of interruptable structured iteration constructs. For example, if M is a stepped ASM with unique end control-state $end(M)$, the following machine is also a stepped ASM with unique start and end control states, say *start* respectively *end*. It can be depicted graphically by the usual FSM-like flowchart.

while $Cond$ **do** $M =$
  if $ctl_state = start$ **then**
    if $Cond$ **then** $\text{START}(M)$
      else $ctl_state := end$
  if $ctl_state = end(M)$ **then** $ctl_state := start$

Acknowledgement. We thank Donato Ferrante and Andrea Vandin for critical remarks on an early draft of this paper.

[20]For example for an ASM with an iterative submachine, in some state the update set may not be defined.

Literaturverzeichnis

[AB09] ALTENHOFEN, M. und E. BÖRGER: *Concurrent Abstract State Machines and +CAL Programs*. In: CORRADINI, A. und U. MONTANARI (Herausgeber): *Proc. WADT'08*, LNCS. Springer, 2009.

[BBD+96] BEIERLE, C., E. BÖRGER, I. DURDANOVIĆ, U. GLÄSSER und E. RICCOBENE: *Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code*. In: *LNCS*, Band 1165, Seiten 62–78. Springer, 1996.

[BPS00] BÖRGER, E., P. PÄPPINGHAUS und J. SCHMID: *Report on a Practical Application of ASMs in Software Design*. In: *LNCS*, Band 1912, Seiten 361–366. Springer, 2000.

[BS03] BÖRGER, E. und R. F. STÄRK: *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[Cra07a] CRAIG, I.D.: *Formal Models of Operating System Kernels*. Springer, 2007.

[Cra07b] CRAIG, I.D.: *Formal Refinement for Operating System Kernels*. Springer, 2007.

[Hal97] HALL, J. A.: *Taking Z seriously*. In: *ZUM'97*, Band 1212 der Reihe *LNCS*, Seiten 89–91. Springer, 1997.

[Lam08] LAMPORT, L.: *The +CAL Algorithm Language*. URL http://research.microsoft.com/users/lamport/tla/pluscal.html, 2008.

[Mea97] MEARELLI, L.: *Refining an ASM Specification of the Production Cell to C++ Code*. J. Universal Computer Science, 3(5):666–688, 1997.

[Mil89] MILNER, R.: *Communication and Concurrency*. Prentice-Hall, 1989.

[Smi00] SMITH, G.: *The Object-Z Specification Language*. Kluwer, 2000.

[Spi92] SPIVEY, J. M.: *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.

[Tan87] TANNENBAUM, A.: *Modern Operating Systems: Design and Implementation*. Prentice-Hall, 1987.