I/O Automaton Model of Operating System Primitives

Daniel Yates[*] Nancy Lynch[†] Victor Luchangco [‡] Margo Seltzer[§]

May 13, 1999

Abstract

Current research in the field of operating systems has been very systems-oriented and result driven. Little theoretical research has been done in considering the formal ramifications of these systems level decisions, or in mapping out the topology of the standard operating system. Because of this, it is often difficult in operating systems work to get a clear picture of the high-level interactions between different OS services, or to apportion programming efforts across well-defined interfaces. Formal specification of the operating system structure and methodology would provide a framework for clearer study, discussion, and implementation of operating systems.

We present a formal method for modelling an operating system as a distributed system of state machines. Drawing the connection between the various independent services of an operating system and the independent agents of distributed systems, we model each service of the operating system as an asynchronous I/O Automaton. Demonstrating both the instructional and functional value of this modelling technique, we present here two views of the operating system. The first view, the User Level model, provides a simplified abstraction of the operating system. This acts as a operating system interface specification as well as an easy first step for teaching students in the field. The second view, or Kernel Level model, provides an implementation of the User Level abstract specification, and unveils many of the realities which were abstracted away in the User Level model. It provides a framework for research in formalizing operating systems, as well as providing a clear and concise description of many of today's operating system elements. Finally, using the powerful mathematical tools developed for I/O Automata, we make two formal modifications to the Kernel Level model and prove that it in fact does implement the User Level specification. We thereby assert that the two models are, from the perspective of the processes, functionally equivalent.

Keywords: operating system, formal model, I/O Automata

[*]Harvard Department of Computer Science, 469 Lowell House Mail Center, Cambridge, MA 02138
[†]MIT Department of Computer Science, 545 Technology Square NE43-366, Cambridge, MA 02139
[‡]MIT Department of Computer Science, 54 Technology Square NE43-372, Cambridge, MA 2139
[§]Harvard Department of Computer Science, 29 Oxford Street Pierce 110D, Cambridge, MA 02138

1 Introduction

Good design, from the highest level down, is fundamental to the success and efficacy of operating systems. The goal of design is to describe a set of modules that interact with one another in simple, well defined ways. In achieving this goal, a well-designed system enables people to work independently on different modules, with the guarantee that the modules will fit together to accomplish the larger purpose.

We present here a formal model of generic operating system primitives using I/O Automata. Leveraging the mathematical structure of automata, we design a formal infrastructure that focuses thought on system error management and interface design. Through this focus, we arrive at a design that is clear and concise, as well as tailored to managing system unreliability and breakdown. Modelling the operating system at two levels, we provide both an abstract specification of the system and a more detailed implementation of that specification. We then prove, using formal methods developed for I/O Automata, that the implementation does in fact simulate the specification.

1.1 Background: Formal Models and Specifications

Research in formal models and specifications has been going on for decades. Starting in the early 1970's, work was done to show that all computable functions over an abstract type could be defined algebraically using equations of a simple form. Much thought went into these specifications, and many people considered the problems of when such specifications constituted an adequate definition.

From these works many different approaches to formal modelling have sprung up. Some researchers have stayed within the limits of algebraic equations, working on more abstract and streamlined versions of the work done before. One example of such work is Yuri Gurevich's work on the Abstract State Machine (ASM) Project (formerly known as the Evolving Algebras Project), an attempt to bridge the gap between formal methods of computation and practical specification methods [1]. The ASM method focuses on building an algebra for a given algorithm, and using such an algebra to convince oneself of the correctness of the algorithm.

Other researchers, deciding that purely algebraic approaches to specification were unlikely to be practical, proposed a combination of algebraic and operational specifications, termed by John V. Guttag to be "dyadic specifications"[2]. Guttag, along with fellow researchers in his LARCH group, proceeded to create a powerful theorem prover to check and reason about specifications [3]. These specifications have been used for checking proofs, as well as for verifying program correctness.

The effort to create the Spec language, led by Butler Lampson, is of particular interest, as it has been used successfully in formalizing problems in operating systems research. Based on Edsger Dijkstra's *Guarded Commands* and augmented by Lampson with constructs for concurrency and atomicity, Spec is a formal specification language designed for modelling distributed systems and algorithms [4]. Lampson's design of Spec is compelling, as it defines many interesting formal tools such as correspondence proofs based upon module trace comparison. It also focuses on defining abstract data types with provable properties and then running actions on state machines which use these data types. With this methodology, Spec has been used successfully in working on such problems as caching, fault-tolerance, concurrency, naming, and security.

While these efforts were, and have been, very successful at what they set out to do, none of them have been directly applied to design and specification of operating systems. Many of the smaller subsystems that compose operating systems have been modelled extensively, but never the operating system itself. Yet many of the issues plaguing operating system designers are the same issues run into by researchers working on distributed systems. As the utility of formal specification tools has been proven in the distributed systems arena, it seems only reasonable to try a similar approach in the study of operating systems.

The I/O Automata model developed by Lynch and Tuttle [6] is the ideal candidate for such a formal specification of an operating system. Designed with the issues of distributed systems in mind, the I/O Automata is an easily abstractable, relatively compact representation of a state machine. With most actions driven by I/O, the emphasis is on reacting properly to the environment and dealing with outside stimuli. Considering the various components of an operating system in this light leads us to several beneficial results. First, it encourages thinking about interface design between components and about detailed formal specifications of interactions. Second, it enforces a certain level of independence and error management between components, as they should be ready to deal with unexpected inputs. Third, it allows for a clear and concise description of the operating system design without many of the implementation details, making for a good tool for discussing and learning operating system primitives.

1.2 Overview of this thesis

The key results of this thesis are the I/O Automata descriptions of a generic operating system at an abstract specification level (User Level), and a more detailed, implementation level (Kernel Level). The design of these descriptions is loosely based on the structure of the NACHOS instructional operating system [7]. Together with a correspondence proof showing that the Kernel Level simulates the User Level, these two models of the operating system provide a formal framework for designing, working on, and teaching operating system primitives.

The thesis is organized as follows: Section 2 defines the I/O automaton model used to formally specify the operating system at both levels of abstraction, and then goes on to define and justify the mathematical tools used in the correspondence proof. Section 3 introduces some formal definitions and conventions used throughout the thesis, including the definition of many of the data types used in the operating system model. Section 4 describes the User Level model of the operating system, first with an intuitive explanation of the specification, and then through the usage of I/O automata. Section 5 does for the Kernel Level model what section 4 did for the User Level, providing the intuitive and I/O automata model of the Kernel Level implementation of the operating system. Section 6 introduces an abstraction automaton, which is necessary to enable the User Level Process to interact in the Kernel Level model. Section 7 describes a series of strong correspondence proofs between several User and Kernel Level automata, and section 8 provides the full simulation proof between the User and Kernel Level operating systems. Section 9 describes potential for future work in the area and concludes the thesis.

2 Formal Model: Definitions and Fundamental Theorems

I/O Automata models are very general, suitable for describing nearly all kinds of asynchronous concurrent systems. They provide a framework for reasoning about system components that interact with each other and that operate at arbitrary relative speeds.

2.1 Definition of I/O Automata[5]

An I/O automaton's signature is simply a description of its input, output, and internal actions. We assume a universal set of *actions*. A *signature* S is a triple consisting of three disjoint sets of actions: the *input actions*, $in(S)$, the *output actions*, $out(S)$, and the *internal actions*, $int(S)$. We define the *external actions*, $ext(S)$, to be $in(S) \cup out(S)$; the *locally controlled actions* $local(S)$, to be $out(S) \cup int(S)$; and $acts(S)$ to be all the actions of S. The *external signature*, $extsig(S)$, is defined to be the signature $(in(S), out(S), \emptyset)$.

An *I/O automaton* A, consists of five components:

- $sig(A)$, a signature
- $states(A)$, a (not necessarily finite) set of *states*
- $start(A)$, a nonempty subset of $states(A)$ known as the *start states* or *initial states*
- $trans(A)$, a *state-transition relation*, where $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$; this must have the property that for every state s and every input action π, there is a transition $(s, \pi, s') \in trans(A)$

We call an element (s, π, s') of $trans(A)$ a *transition*, or *step* of A. The transition (s, π, s') is called an *input transition, output transition*, or *internal transition* based on whether the action π is an input, output, or internal action. We write $s \xrightarrow{\pi} s'$ as shorthand for $(s, \pi, s') \in trans(A)$. If for a particular state s and action π, A has some transition of the form $s \xrightarrow{\pi} s'$, then we say that π is *enabled* in s. Since every input action is required to be enabled in every state, automata are said to be *input-enabled.* The input-enabling assumption means that the automaton is not able to somehow "block" input actions from occurring. This assumption means, for example, that a process has to be prepared to cope in some way with any possible message value when a message arrives.

There are two major advantages to having the input-enabling property. First, a serious source of errors in the development of system components is the failure to specify what the component does in the face of unexpected inputs. Using a model that requires consideration of arbitrary inputs is helpful in eliminating such errors. Second, use of input-enabling makes the basic theory of the model work out nicely; in particular input-enabling makes it reasonable to use simple notions of external behavior for an automaton, based on sequences of external actions.

In describing I/O Automata, the transition relation is written in a *precondition-effect* style. This style groups together all the transitions that involve each particular type of action into a single piece of code. The code specifies the conditions under which the action is permitted to occur as a predicate on the pre-state s. Then it describes the changes that occur as a result of the action in the form of a simple program that is applied to s to yield s'. The entire piece of code gets executed in a single, atomic transition.

An *execution fragment* of A is either a finite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$, or an infinite sequence $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$, of alternating states and actions of A such that $s_k \xrightarrow{\pi_{k+1}} s_{k+1}$ is a transition of A for every $k \geq 0$. Note that if the sequence is finite, it must end with a state. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by $execs(A)$. A state is said to be *reachable* in A if it is the final state of a finite execution of A.

2.2 Composition

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing individual system components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving π, so do all component automata that have π in their signatures. In order for a countable set of automata to be *compatible*, their signatures $\{S_i\}_{i \in I}$ must be such that $\forall\ i, j \in I, i \neq j$, the following hold:

- $int(S_i) \cap acts(S_j) = \emptyset$
- $out(S_i) \cap out(S_j) = \emptyset$
- No action is contained in infinitely many sets acts(S_i)

Upon composing a set of automata, output actions of the components become output actions of the composition, internal actions of the components become internal actions of the composition, and actions that are inputs to some components but outputs of none become input actions of the composition. Formally, the *composition* $S = \prod_{i \in I} S_i$ of a countable compatible collection of signatures $\{S_i\}_{i \in I}$ is defined to be the signature with

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$
- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

With that defined, it becomes easy to define the composition of a countable compatible collection of automata $\{A_i\}_{i \in I}$. It is the automaton defined as

- $sig(A) = \prod_{i \in I} sig(A_i)$
- $states(A) = \prod_{i \in I} states(A_i)$
- $start(A) = \prod_{i \in I} start(A_i)$
- $trans(A)$ is the set of triples $s \xrightarrow{\pi} s'$ such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $s_i \xrightarrow{\pi} s'_i \in trans(A_i)$; otherwise $s_i = s'_i$

2.3 Hiding

We now define an operation that "hides" output actions of an I/O automaton by reclassifying them as internal actions. This prevents them from being used for further communication and means that they are no longer included in traces.

We first define the hiding operation for signatures: if S is a signature and $\Sigma \subseteq out(S)$, then $hide_{\Sigma}(S)$ is defined to be the new signature S', where $in(S') = in(S)$, $out(S') = out(S) - \Sigma$, and $int(S') = int(S) \cup \Sigma$.

The hiding operation for I/O Automata is now easy to define: if A is an automaton and $\Phi \subseteq out(A)$, then $hide_{\Phi}(A)$ is the automaton A' obtained from A by replacing $sig(A)$ with $sig(A') = hide_{\phi}(sig(A))$.

2.4 Simulations

The simulation proof in Section 7 relies on the tools developed for hierarchical automata. We can show that a lower level automata implements a higher level automata by obtaining a one-way relationship between the two, showing that for any execution of the lower-level automaton there is a "corresponding" execution of the higher-level automaton. One typically does this by defining a simulation relation between states of the two automata:

Let A and B be two I/O Automata with the same external interface; we think of A as the lower-level automaton, and B as the higher-level one. Suppose F is a binary relation over $states(A)$ and $states(B)$, that is, $F \subseteq states(A) \times states(B)$. Then F is a *simulation relation* from A to B, provided that both of the following are true:

1. If $s \in start(A)$, the $F[s] \cap start(B) \neq \emptyset$.
2. If s is a reachable state of A, $u \in F[s]$, where u is a reachable state of B, and $s \xrightarrow{\pi} s' \in trans(A)$, then there is an execution fragment α of B starting with u and ending with some $u' \in F[s']$, such that $trace(\alpha) = trace(\pi)$.

Lemma 1: Let F_1 be a simulation relation from A_1 to B_1, F_2 be a simulation relation from A_2 to B_2, and F be the binary relation from $states(A_1 \times A_2)$ to $states(B_1 \times B_2)$ where $(u_1, u_2) \in F[(s_1, s_2)]$ if and only if $u_1 \in F_1[s_1] \wedge u_2 \in F_2[s_2]$. F is a simulation relation from $A_1 \times A_2$ to $B_1 \times B_2$.

Proof:

We prove this by showing that F adheres to the definition of simulation relation.

1. If $(s_1, s_2) \in start(A_1) \times start(A_2)$, then $\exists (u_1, u_2) \in start(B_1) \times start(B_2)$ such that $(u_1, u_2) \in F[(s_1, s_2)]$. From the definition of composition, we see that the start states of the composition of two automata A and B are $start(A) \times start(B)$. We know from the simulation relation that $s_1 \in start(A_1)$ and $s_2 \in start(A_2)$ implies that $\exists u_1 \in F_1[s_1]$ such that $u_1 \in start(B_1)$ and $\exists u_2 \in F_2[s_2]$ such that $u_2 \in start(B_2)$. Therefore, $\exists (u_1, u_2) \in F[(s_1, s_2)]$ such that $(u_1, u_2) \in start(B_1, B_2)$.
2. If (s_1, s_2) is a reachable state of $A_1 \times A_2$, $(u_1, u_2) \in F[(s_1, s_2)]$ is a reachable state of $B_1 \times B_2$,

and $(s_1, s_2)\xrightarrow{\pi}(s'_1, s'_2) \in \mathit{trans}(A_1 \times A_2)$, then there is an execution fragment α of $B_1 \times B_2$ starting with (u_1, u_2) and ending with some $(u'_1, u'_2) \in F[(s'_1, s'_2)]$ such that $\mathit{trace}(\alpha) = \mathit{trace}(\pi)$.

For $s_1 \in \mathit{states}(A_1)$, $u_1 \in F_1[s_1]$, and $s_1\xrightarrow{\pi}s'_1$, we know from the simulation relation that there is an execution fragment α_1 of B_1 starting with u_1 and ending with some $u'_1 \in F[s'_1]$, such that $\mathit{trace}_{B_1}(\alpha_1) = \mathit{trace}_{A_1}(\pi)$. Similarly, for $s_2 \in \mathit{states}(A_2)$, $u_2 \in F_2[s_2]$, and $s_2\xrightarrow{\pi}s'_2$, we know from the simulation relation that there is an execution fragment α_2 of B_2 starting with u_2 and ending with some $u'_2 \in F[s'_2]$, such that $\mathit{trace}_{B_2}(\alpha_2) = \mathit{trace}_{A_2}(\pi)$. Therefore, (u_1, u_2) transitions to (u'_1, u'_2) by $\alpha_1 \cdot \alpha_2$. As $\mathit{extsig}(A_1) = \mathit{extsig}(B_1)$, and $\mathit{extsig}(A_2) = \mathit{extsig}(B_2)$, the composition A of A_1 and A_2 will have the same external signature as the composition B of B_1 and B_2. Therefore, since $\mathit{trace}_{B_1}(\alpha_1) = \mathit{trace}_{A_1}(\pi)$ and $\mathit{trace}_{B_2}(\alpha_2) = \mathit{trace}_{A_2}(\pi)$, we can conclude that $\mathit{trace}_B(\alpha_1 \cdot \alpha_2) = \mathit{trace}_A(\pi)$

$\square$

2.5 Strong Correspondence

In the consideration of the various automata which we define in this thesis, it will be meaningful for us to draw formal parallels between individual automata in the specification and implementation. For example, it is intuitive that there is a relationship between the Memory Manager at the specification level, and the Virtual Memory Manager at the implementation level; we would like to formalize this relationship. However, because these automata do not have the same external signatures, we cannot prove that one implements the other. Instead, we must define a new notion, that of *strong correspondence*, which we will use as an intermediate step in our simulation proof: Let A and B be two I/O Automata where $\mathit{in}(B) \subseteq \mathit{in}(A), \mathit{out}(B) \subseteq \mathit{out}(A), \mathit{int}(B) \subseteq \mathit{int}(A)$; we think of B as the higher-level automaton, and A as the lower-level one. Suppose F is a binary relation over $\mathit{states}(A)$ and $\mathit{states}(B)$, that is $F \subseteq \mathit{states}(A) \times \mathit{states}(B)$. Then F is a *strong correspondence* provided that both of the following are true:

1. If $s \in \mathit{start}(A)$, then $F[s] \cap \mathit{start}(B) \neq \emptyset$.

2. If $s\xrightarrow{\pi}s' \in \mathit{trans}(A) \wedge u \in F[s] \wedge \pi \in \mathit{acts}(B)$ then $\exists u' \in F[s']$ such that $u\xrightarrow{\pi}u' \in \mathit{trans}(B)$.

3. If $s \xrightarrow{\pi} s' \in trans(A) \wedge u \in F[s] \wedge \pi \notin acts(B)$ then $u \in F[s']$

With this definition we will be able to show a strong correspondence between individual automata in the User Level model with automata in the Kernel Level model. We will then be able to compose these strongly corresponding automata with each other to arrive at a composition of automata in the specification that strongly correspond to a matching composition in the implementation. The need to compose strongly corresponding automata motivates the following theorem.

Let A_1, A_2, B_1, and B_2 be four I/O Automata where $in(B_1) \subseteq in(A_1), out(B_1) \subseteq out(A_1), int(B_1) \subseteq int(A_1)$, and $in(B_2) \subseteq in(A_2), out(B_2) \subseteq out(A_2), int(B_2) \subseteq int(A_2)$; we think of A_1 and A_2 as the lower-level automata, and B_1 and B_2 as the higher-level ones. Suppose F_1 is a strong correspondence over $states(A_1)$ and $states(B_1)$, and F_2 is a strong correspondence over $states(A_2)$ and $states(B_2)$. Let F be a binary relation over $states(A_1) \times states(A_2)$ and $states(B_1) \times states(B_2)$, where $(u_1, u_2) \in F[(s_1, s_2)]$ if and only if $u_1 \in F_1[s_1] \wedge u_2 \in F_2[s_2]$.

Theorem 1: F is a strong correspondence from $states(A_1 \times A_2)$ to $states(B_1 \times B_2)$.

Proof:

We prove this by showing that F adheres to the definition of strong correspondence. The first property in the definition can be proven straightforwardly. The second and third properties combine to form four cases with two automata: $\pi \in acts(B_1) \wedge acts(B_2)$, $\pi \in acts(B_1)$, $\pi \in acts(B_2)$, and $\pi \notin acts(B_1) \vee acts(B_2)$.

1. If $(s_1, s_2) \in start(A_1 \times A_2)$ then $\exists (u_1, u_2) \in F[(s_1, s_2)]$ such that $(u_1, u_2) \in start(B_1 \times B_2)$.

   From the definition of composition, we see that the start states of the composition of two automata A and B are $start(A) \times start(B)$. We know from strong correspondence that $s_1 \in start(A_1)$ and $s_2 \in start(A_2)$ implies that $\exists u_1 \in F_1[s_1]$ such that $u_1 \in start(B_1)$, and $\exists u_2 \in F_2[s_2]$ such that $u_2 \in start(B_2)$. Therefore, $\exists (u_1, u_2) \in F[(s_1, s_2)]$ such that $(u_1, u_2) \in start(B_1 \times B_2)$.

2. If $(s_1, s_2) \xrightarrow{\pi} (s_1', s_2') \in trans(A_1 \times A_2) \wedge (u_1, u_2) \in F[(s_1, s_2)] \wedge \pi \in acts(B_1) \wedge \pi \in acts(B_2)$ then $\exists (u_1', u_2') \in F[(s_1', s_2')]$ such that $(u_1, u_2) \xrightarrow{\pi} (u_1', u_2') \in trans(B_1 \times B_2)$.

   From the definition of strong correspondence we know that $\exists u_1' \in F_1[s_1']$ such that $u_1 \xrightarrow{\pi} u_1'$, and $\exists u_2' \in F_2[s_2']$ such that $u_2 \xrightarrow{\pi} u_2'$. Therefore $(u_1', u_2') \in F[(s_1', s_2')]$ by the definition of strong correspondence. From the definition of composition we know that $(u_1, u_2)\xrightarrow{\pi}(u'_1, u'_2)$, therefore $\exists(u'_1, u'_2) \in F[(s'_1, s'_2)]$ such that $(u_1, u_2)\xrightarrow{\pi}(u'_1, u'_2) \in \mathit{trans}(B_1 \times B_2)$.

3. If $(s_1, s_2)\xrightarrow{\pi}(s'_1, s'_2) \in \mathit{trans}(A_1 \times A_2) \wedge (u_1, u_2) \in F[(s_1, s_2)] \wedge \pi \in \mathit{acts}(B_1) \wedge \pi \notin \mathit{acts}(B_2)$ then $\exists(u'_1, u_2) \in F[(s'_1, s'_2)]$ such that $(u_1, u_2)\xrightarrow{\pi}(u'_1, u_2) \in \mathit{trans}(B_1 \times B_2)$.

   From the definition of strong correspondence we know that $\exists u'_1 \in F_1[s'_1]$ such that $u_1\xrightarrow{\pi}u'_1$, and $u_2 \in F_2[s'_2]$. Therefore $(u'_1, u_2) \in F[(s'_1, s'_2)]$ by the definition of strong correspondence. From the definition of composition we know that $(u_1, u_2)\xrightarrow{\pi}(u'_1, u_2)$, therefore $\exists(u'_1, u_2) \in F[(s'_1, s'_2)]$ such that $(u_1, u_2)\xrightarrow{\pi}(u'_1, u_2) \in \mathit{trans}(B_1 \times B_2)$.

4. If $(s_1, s_2)\xrightarrow{\pi}(s'_1, s'_2) \in \mathit{trans}(A_1 \times A_2) \wedge (u_1, u_2) \in F[(s_1, s_2)] \wedge \pi \notin \mathit{acts}(B_1) \wedge \pi \in \mathit{acts}(B_2)$ then $\exists(u_1, u'_2) \in F[(s'_1, s'_2)]$ such that $(u_1, u_2)\xrightarrow{\pi}(u_1, u'_2) \in \mathit{trans}(B_1 \times B_2)$.

   This argument holds for exactly the same reason as the argument in the previous item.

5. If $(s_1, s_2)\xrightarrow{\pi}(s'_1, s'_2) \in \mathit{trans}(A_1 \times A_1) \wedge (u_1, u_2) \in F[(s_1, s_2)] \wedge \pi \notin \mathit{acts}(B_1) \wedge \pi \notin \mathit{acts}(B_2)$ then $(u_1, u_2) \in F[(s'_1, s'_2)]$.

   From the definition of strong correspondence we know that $\exists u_1 \in F_1[s'_1]$ and $u_2 \in F_2[s'_2]$. Therefore $(u_1, u_2) \in F[(s'_1, s'_2)]$ by the definition of strong correspondence.

$\square$

In addition to being able to compose strongly corresponding automata, we will need to compose automata in the Kernel Level which correspond to nothing in the User Level with strongly corresponding automata in the Kernel Level, and still maintain the strong correspondence. This is necessary as there are automata in the Kernel Level, such as the Scheduler and Interrupt Handler, which do not exist in any form in the User Level model, but must be included in the correspondence so as to prove that the full Kernel Level operating system implements the User Level system.

Let A_1 and B be two automata and let F be a strong correspondence from A_1 to B Let A_2 be an automaton which is compatible with A_1. Define G to be a binary relation from $A_1 \times A_2$ to B, where $u_1 \in G[(s_1, s_2)]$ if and only if $u_1 \in F_1[s_1]$.

Theorem 2: G is a strong correspondence from $A_1 \times A_2$ to B_1.

Proof: Intuitively, the states of A_2 have no impact on the correspondence, and therefore, if F is a strong correspondence between A_1 and B_2, then the correspondence will still hold with the addition of the unaccounted for states of A_2. Formally the result is derived from the definition of strong correspondence.

1. If $(s_1, s_2) \in start(A_1 \times A_2)$ then $\exists u_1 \in G[(s_1, s_2)]$ such that $u_1 \in start(B)$.

   From the definition of composition, we see that the start states of the composition of two automata A and B are $start(A) \times start(B)$. We know from strong correspondence that $s_1 \in start(A_1)$ implies that $\exists u_1 \in F[s_1]$ such that $u_1 \in start(B)$. Therefore, $\exists u_1 \in [(s_1, s_2)]$ such that $u_1 \in start(B_1)$.

2. If $(s_1, s_2) \xrightarrow{\pi} (s_1', s_2') \in trans(A_1 \times A_2) \wedge u_1 \in F[(s_1, s_2)] \wedge \pi \in acts(B_1)$
   then $\exists u_a' \in F[(s_a', b')]$ such that $u_1 \xrightarrow{\pi} u_1' \in trans(B_1)$

   From the definition of strong correspondence we know that $\exists u_1' \in F[s_1']$ such that $u_1 \xrightarrow{\pi} u_1'$. Therefore $u_1' \in F[(s_1', s_2')]$ by the definition of F. Therefore $\exists u_1' \in F[(s_1', s_2')]$ such that $u_1 \xrightarrow{\pi} u_1' \in trans(B_1)$

3. If $(s_1, s_2) \xrightarrow{\pi} (s_1', s_2') \in trans(A_1 \times A_2) \wedge u_1 \in F[(s_1, s_2)] \wedge \pi \notin acts(B_1)$
   then $u_1 \in F[(s_1', s_2')]$

   From the definition of strong correspondence we know that $\exists u_1 \in F[s_1']$. Therefore $u_1 \in F[(s_1', s_2')]$ by the definition of F. Therefore $u_1 \in F[(s_1', s_2')]$.

□

With this machinery we can provide strong correspondences between many of the component automata of the two models, providing interesting results in their own right, as well as simplifying the act of proving the simulation between the two fully composed operating systems.

2.6 From Strong Correspondence to Simulation

As the goal of the strong correspondence relation is to enable a simulation proof, the final steps in the proof require converting a strong correspondence relation to a simulation relation. We do this in two steps:

Theorem 3: If F is a strong correspondence from A to B, and $\Sigma \subseteq out(A)$, then $hide_\Sigma(A)$ strongly corresponds to $hide_\Sigma(B)$.

Proof: This is derived easily from the definition of strong correspondence. Hiding is merely the act of changing output actions to internal actions. In the definition of strong correspondence we see that there is never any consideration of action types, merely the existential evaluation of whether the action is or is not present in the automaton. Therefore, the action of hiding, which consists entirely of change action types, has no impact on strong correspondence.

□

Theorem 4: If F is a strong correspondence from A to B and $in(A) = in(B), out(A) = out(B), int(A) \supseteq int(B)$ then A simulates B.

Proof: We show that the definition of strong correspondence between A and B, augmented with the fact that $in(A) = in(B), out(A) = out(B)$, is strong enough to imply the definition of a simulation relation.

Directly from the first requirement of strong correspondence we know that if $s \in start(A)$, then $\exists u \in F[s] \cap start(B)$.

From strong correspondence we know that if s is a reachable state of A, $\exists u \in F[s]$ which is a reachable state of B. We also know that all actions π in A correspond either to π in B or the empty transition. So there are two cases:

1. $\pi \in trans(B)$. In this case the execution fragment α is s, π, s', and therefore $trace(\alpha) = trace(\pi)$.

2. $\pi \notin trans(B)$. In this case the execution fragment $\alpha = \epsilon$. We know that π in A must be internal, because otherwise $extsig(A) \neq extsig(B)$, and therefore $trace(\alpha) = trace(\pi)$.

So, from the additional condition that $in(A) = in(B)$, and $out(A) = out(B)$, strong correspondence implies simulation relation. □

3 Preliminary Definitions and Conventions

The following notation is used throughout the rest of the thesis in the descriptions of the I/O Automata models.

- Predefined Sets and Variables:

Booleans	the set {TRUE, FALSE}
Bytes	the set of arrays of eight $\{0, 1\}$
ByteArrays	the set of arrays of *Bytes*
Chars	the set of all ASCII characters
Files	the set of all possible file names in the File System
Paths	the set of all possible file paths in the File System
Procs	the set of all processes. All possible processes are in this set, each with an infinite number of copies, so that there is no limitation to any permutation of processes which can be run on the operating system.
Sockets	the set of all possible network connections from the local machine

- AllocateArrays:

  An AllocateArray manages a large array which it allocates, upon request, into separate contiguous subarrays, or blocks. Each block is a pair $(head, n)$ where *head* is the index in the large array of the first element of the subarray, and n is the length of the block. All currently allocated blocks are maintained in an internal set by the AllocateArray. The AllocateArray structure implements the following operations:

$Empty(n)$	Initialize an AllocateArray to manage an array of size n, no blocks allocated.
$FindBlock(p)$	Find the block in the AllocateArray with index p in the large array. If such a block exists, return it, otherwise return FAIL.
$NewBlock(n)$	Find an unused block of size n. Return the index of the head of the block if one exists, else return FAIL.
$RemoveBlock(p)$	Remove the block starting with first index p. $FAIL$ if such a block does not exist.
$AddBlock(p, n)$	Add the block (p, n) to the internal set. $FAIL$ if any of the specified locations are already in a subarray.

- Queues:

  A Queue is a list of elements which adds new elements to the end of the list, and removes elements from the front of the list. In addition to these standard operations the queues used throughout the thesis have a few additional properties which enhance their use. Queues used here have the following operations:

Empty	Initialize a queue to hold no elements.
Enqueue(x)	Add an element x to the end of the queue.
Dequeue	Remove an element from the front of the queue. $FAIL$ if empty.
Head	Return the value of the element at the front of the queue without removing it.
Remove(x)	Remove the first instance of element x from the queue and maintain the queue structure.
Size	Return the number of entries currently stored in the queue.

- Sets:

  Sets have all the typical operations of a set. I will primarily use $\cup, \cap, +, -, \subset, \subseteq$.

- State variables of the form $State_x$:

  For each process x there is a unique instance of the state variable $State_x$.

- Actions of the form Action_x

  For each process x there is a unique action Action_x.

4 User Level Specification

4.1 Description

In many ways, the process of building a proper formal design for a computer system is a microcosm of building the system itself. In building a computer system, the first step taken is making a high level specification detailing system component interfaces and functions. So it is in the formal design process. We present here, in the form of the User Level model, an abstract specification of an operating system detailing the interfaces between the various components of the system, and outlining the required functionality of each of the components.

In specifying an operating system, we consider what it interacts with and the role that it plays in that interaction. One view of an operating system, and perhaps the easiest one to consider initially, is as a resource manager, distributing the hardware resources of the computer system to the processes running on it. This is the perspective that we take in creating the User Level model of the operating system. Our concern is not with the interaction between the operating system and the hardware; rather we are interested in the operating system/process interface, and the interactions that occur across that boundary. In this capacity, we investigate the process' view of the operating system, and how the two interact.

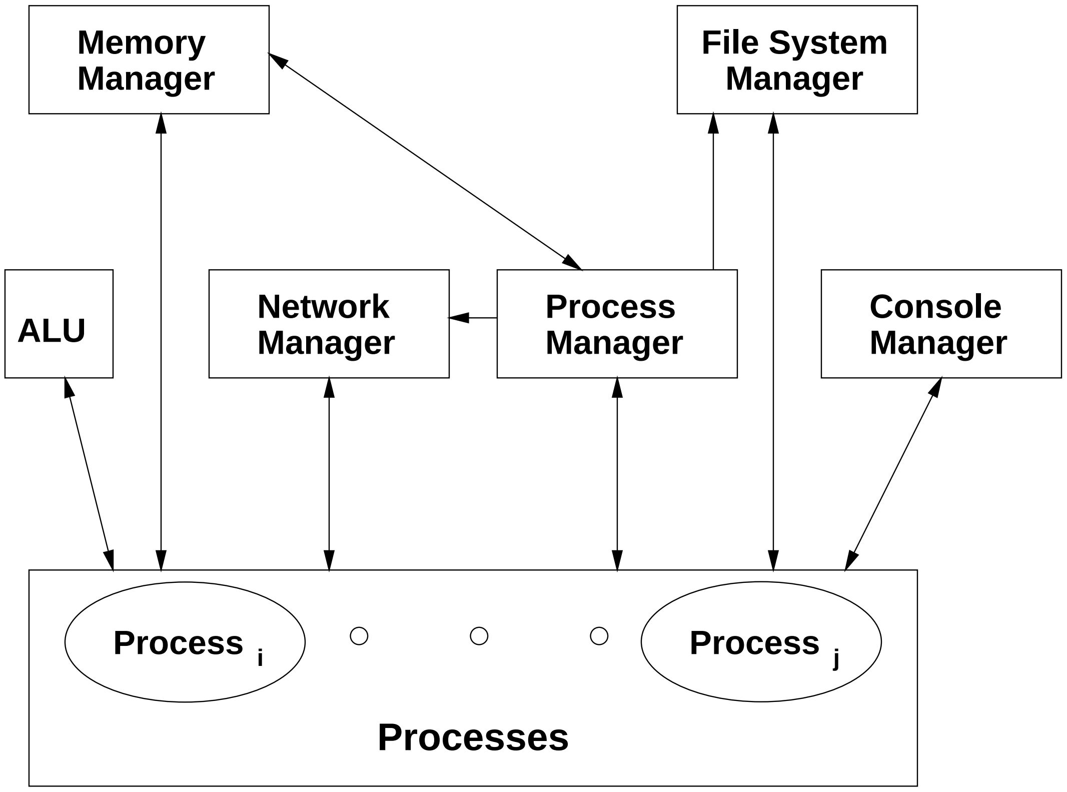

Figure 1: User Level model: conceptual framework

Much of our specification of the operating system at this level is motivated by our description of the processes that interact with it. We model processes here as independent, concurrently running automata, interacting with the operating system through the system call interface. As such, that interaction consists of requests for resources and the responses received for those requests. Processes are not concerned with the means by which operating systems provide resources, merely that resources are provided in a consistent and reliable manner. We define the resources of interest to a process to be: memory, console read/write capabilities, file system access, network access, computational resources, and process life cycle management. Memory resources, managed by the Memory Manager, include such functionality as acquiring and freeing memory, as well as reading from and writing to it. Console resources, governed by the Console Manager, control the simple I/O devices for displaying results to and receiving input from the user. File system and network resources allow for the manipulation of files and handling of network connections. These resources are controlled by the File System and Network Managers respectively. Computing resources supported by the ALU assist processes in performing arithmetic operations and storing the results of these operations in per process registers. Finally, Process life cycle management is handled by the Process Manager, and deals with such process requests as spawning new processes, process exiting, and system halt. As we see in the User Level I/O Automata model, Figure 1, each of these resources is modeled by a separate automaton. By and large, the operating system automata at this level are independent; the only interconnections are due to the Process manager, which notifies the different resource managers when a process has been created or killed, and the ALU, which stores registers in the Memory Manager. Communication between automata in the figure are represented by arrows—each arrow head represents one direction of communication. Because there is no limitations on the number of processes, they are represented by the large block titled "Processes", with an indeterminate number of them contained within. There is in fact a separate communication link between every process and operating system automaton, but for graphical reasons these are represented as one single link for all processes.

Because the User Level model, in trying to maintain the simplicity and streamlined aspects of the system call interface, takes the view of the operating system from the process perspective, it does not model all aspects of real operating systems. The processes are modelled as running independently and concurrently; consequently there is no role for scheduling or resource sharing in this model. Notably, this is not a failing of the model, rather it is simply an interesting point about the need for scheduling. The idea of a multi-process operating system does not demand scheduling—it only requires that the operating system provide multiple processes with multiple resources at what appears to the processes to be the same time. Running all the processes on one processor and dividing processor time with a scheduler is not the only solution to this problem, it is a choice of implementation. Clearly it would be possible to have a separate processor for each process (as there are in some parallel machines), and this would require no scheduling at all. We already see that the specification, in forcing us to state exactly what we require from the operating system, has eliminated certain aspects of the system and focused heavily on others.

While the abstractions which we make in defining the User Level model hide some of the intricacies of real modern operating systems, the resultant operating system specification does well in providing the highest level design of an operating system. This is exactly what we are looking for in this model—an abstract specification not necessarily of what an operating system looks like, but of what functionality it provides. Later, in the Kernel Level model, we will see more aspects of real modern operating systems, including interrupt driven I/O, process scheduling, and virtual memory management. These concepts are critical to the success of a real operating system, but they do not affect the functional specification which we desire in the User Level model.

4.2 Process Definition

The Process definition motivates the rest of the operating system design. Defined to be an independent, concurrently running automaton, the Process represents the view of the operating system environment maintained by a typical program. Ignorant of any sort of resource sharing or limitations, the Process runs under the assumption of a concurrent system with contention free resources.

Processes access all operating system resources through a well-defined system call interface with the operating system. The operating system is implemented so as to support that interface, and all processes must be designed to work within its limits. Through understanding the Process' interaction with the operating system, one understands the reasons for the particular implementation of each of the OS modules at both levels of specificity described in this thesis. In Figure 2 one sees the signature of the Process and the single piece of process state that is necessary for the process to interact properly with the operating system. For the most part, there is no limitation on the actions of processes; this is intended to model real programs which come in all forms. The only action that is fixed is the $\mathsf{Activate}_x$ action, which enforces the concept of killing and creating processes. Even this action, however, is not required to be exactly as stated; it need only to guarantee that it sets the *Active* variable according to the input.

$\mathsf{MallocResult}_x$, $\mathsf{MemReadResult}_x$, and $\mathsf{MemWriteDone}_x$ are all inputs from the Memory Manager, with the complementary output actions of Malloc_x, Free_x, $\mathsf{MemRead}_x$ and $\mathsf{MemWrite}_x$. Input from the Process Manager comes only from the input action $\mathsf{Activate}_x$, which sets the process' *Active* boolean state variable, while the process outputs Halt, Exit_x, KillProc and Spawn. Input from the console manager comes from $\mathsf{ConsoleReadResult}_x$ and $\mathsf{ConsoleWriteResult}_x$, with output going to it in the forms of $\mathsf{ConsoleRead}_x$ and $\mathsf{ConsoleWrite}_x$. The File System Manager receives from the process the outputs FSOpen_x, $\mathsf{FSCreate}_x$, $\mathsf{FSReadAt}_x$, $\mathsf{FSWriteAt}_x$, $\mathsf{FSClose}_x$, and $\mathsf{FSDelete}_x$. Similarly, the process receives as input from the File System Manager, $\mathsf{FSOpenResult}_x$, $\mathsf{FSCreateResult}_x$, $\mathsf{FSReadAtResult}_x$, $\mathsf{FSWriteAtResult}_x$, $\mathsf{FSCloseResult}_x$, and $\mathsf{FSDeleteResult}_x$. The Network Man-

Process

Signature

Input:

- $\mathsf{MallocResult}_x(p)$, where $p \in \mathbf{N} \cup \{FAIL\}$
- $\mathsf{MemReadResult}_x(v)$, $v \in Bytes$
- $\mathsf{MemWriteDone}_x$
- $\mathsf{Activate}_x(b)$, $b \in \{0, 1\}$
- $\mathsf{ConsoleReadResult}_x(y)$, $y \in Chars \cup \{EOF\}$
- $\mathsf{ConsoleWriteResult}_x(y)$, $y \in \{Busy, DONE\}$
- $\mathsf{FSOpenResult}_x(y, path, z)$, $y \in Files, path \in Paths, z \in Files \cup \{FAIL\}$
- $\mathsf{FSCreateResult}_x(y, path, z)$, $y \in Files, path \in Paths, z \in Files \cup \{F\}$
- $\mathsf{FSReadAtResult}_x(y, path, A)$, $y \in Files$, $path \in Paths, A \in ByteArrays \cup \{FAIL\}$
- $\mathsf{FSWriteAtResult}_x(y, path, z), y \in Files$, $path \in Paths, z \in \{DONE, FAIL\}$
- $\mathsf{FSCloseResult}_x(y, path, z), y \in Files$, $path \in Paths, z \in \{DONE, FAIL\}$
- $\mathsf{FSDeleteResult}_x(y, path, z), y \in Files$, $path \in Paths, z \in \{DONE, FAIL\}$
- $\mathsf{NetOpenResult}_x(y, z)$, $y \in Sockets, z \in \{DONE, FAIL\}$
- $\mathsf{NetReadResult}_x(y, A)$, $y \in Sockets$, $A \in ByteArrays \cup \{DONE, FAIL\}$
- $\mathsf{NetWriteResult}_x(y, z)$, $y \in Sockets, z \in \{DONE, FAIL\}$
- $\mathsf{NetCloseResult}_x(y, z)$, $y \in Sockets, z \in \{DONE, FAIL\}$
- $\mathsf{OperationResult}_x(opVal)$, $x \in Procs$, $opVal \in \mathbf{N}$

Output:

- $\mathsf{Malloc}_x(n)$, $n \in \mathbf{N}$, $x \in Procs$
- $\mathsf{Free}_x(p)$, $p \in \mathbf{N}$
- $\mathsf{MemRead}_x(p)$, $p \in \mathbf{N}$
- $\mathsf{MemWrite}_x(p, v)$, $p \in \mathbf{N}$, $v \in Bytes$
- Halt
- Exit_x
- $\mathsf{Spawn}(y)$, $y \in Procs$
- $\mathsf{KillProc}_x$
- $\mathsf{ConsoleRead}_x$
- $\mathsf{ConsoleWrite}_x(c)$, $c \in Chars$
- $\mathsf{FSOpen}_x(y, path)$, $y \in Files, path \in Paths$
- $\mathsf{FSCreate}_x(y, path, n)$, $y \in Files, path \in Paths$, $n \in \mathbf{N}$
- $\mathsf{FSReadAt}_x(y, path, n, m)$, $y \in Files$, $path \in Paths, n, m \in \mathbf{N}$
- $\mathsf{FSWriteAt}_x(y, path, n, m, A)$, $y \in Files$, $path \in Paths, n, m \in \mathbf{N}, A \in ByteArrays$
- $\mathsf{FSClose}_x(y, path)$, $y \in Files, path \in Paths$
- $\mathsf{FSDelete}_x(y, path)$, $y \in Files, path \in Paths$
- $\mathsf{NetOpen}_x(y)$, $y \in Sockets$
- $\mathsf{NetRead}_x(y, n)$, $y \in Sockets, n \in \mathbf{N}$
- $\mathsf{NetWrite}_x(y, A)$, $y \in Sockets, A \in ByteArrays$
- $\mathsf{NetClose}_x(y)$, $y \in Sockets$
- $\mathsf{OperationReq}_x(op)$ $\quad x \in Procs$, $op \in Operations$

State

$Active \in Booleans$, TRUE if the process is activated.

Actions

```
Input Activate_x(b)
    Eff: if $b = 1$
             $Active :=$ TRUE
         else
             $Active :=$ FALSE
```

All internal and output actions in the process must have $Active$ =TRUE as a precondition.

Figure 2: The Process (User Level Specification)

ager provides the process with the inputs NetOpenResult, NetReadResult, NetWriteResult, and NetCloseResult, and receives the outputs NetOpen, NetRead, NetWrite, and NetClose. Finally, the process communicates to the ALU with the output $\mathsf{OperationReq}_x(op)$, and receives the results in the input $\mathsf{OperationResult}(opVal)$.

4.3 Memory Manager

In the User Level specification, the Memory Manager is responsible for the allocation and deallocation of memory space to the processes, as well as reading from and writing to allocated memory. The physical implementation of the memory is abstracted away from the processes, as they can only access memory by passing pointers to the operating system through a series of system calls. Upon creation, each process is allocated a process space from which it may use memory; this space is fixed for the duration of the process. There is no notion of Virtual Memory, as exists in the Kernel Level implementation, so the number of processes is limited by the size of the memory space.

The state of the Memory Manager is primarily kept in $\mathit{MemSpace}_x$, Used_x, and $\mathit{Freelist}_x$ which together comprise the memory state for each $\mathit{Process}_x$. These per process structures keep track of the memory that has been allocated and that can still be allocated to each process, as well as all of the data values that are stored per process.

The paths of requests for Malloc, MemRead and MemWrite and Free go between the Memory Manager and the processes, and are all fairly similar. Each of the first three requests makes some decisions and then enqueues a command in the *Responses* queue, to be picked up by the output actions MallocResult, MemReadResult, and MemWriteDone respectively. The Free request differs from these three actions in that it exhibits no external response. MemRead and MemWrite both use the Used_x data structure to evaluate if a memory request is validly within memory already allocated to the process. Similarly, the Malloc input action uses the $\mathit{Freelist}_x$ to determine if there is any memory available to be allocated. If such free memory exists, it is added to the Used_x structure and removed from the $\mathit{Freelist}_x$.

Requests for CreateProc and KilledProc are handled in a similar fashion to the other input actions, with the difference that communication is between the Memory Manager and the Process Manager. Each input action simply enqueues the command $(CREATEPROC, x)$ in the case of CreateProc, or $(KILLEDPROC, x)$ in the case of KilledProc, in the *Responses* queue. Each of

Memory Manager

Signature

Input:

- $\mathsf{Malloc}_x(n)$, $n \in \mathbf{N}$, $x \in \mathit{Procs}$
- $\mathsf{Free}_x(p)$, $p \in \mathbf{N}$
- $\mathsf{MemRead}_x(p)$, $p \in \mathbf{N}$
- $\mathsf{MemWrite}_x(p, v)$, $p \in \mathbf{N}$, $v \in \mathit{Bytes}$
- $\mathsf{KilledProc}(x)$, $x \in \mathit{Procs}$
- $\mathsf{CreateProc}(x)$, $x \in \mathit{Procs}$

Output:

- $\mathsf{MallocResult}_x(p)$, where $p \in \mathbf{N} \cup \{FAIL\}$
- $\mathsf{MemReadResult}_x(v)$, $v \in \mathit{Bytes}$
- $\mathsf{MemWriteDone}_x$
- $\mathsf{KillProc}(x)$
- $\mathsf{KilledProcResult}(x, v)$, $x \in \mathit{Procs}$, $v \in \{DONE, FAIL\}$
- $\mathsf{CreateProcResult}(x, v)$, $x \in \mathit{Procs}$, $v \in \{DONE, FAIL\}$

State

SPACESIZE, fixed size of process space.

$\mathit{MemSpace}_x$, the memory for process x, an array of size $MAXHEAP + MAXSTACK$. Initially all 0.

Used_x, an AllocateArray holding arrays from $\mathit{MemSpace}_x$, the set of bytes used by process x, initially *Empty*.

$\mathit{Freelist}_x$, an AllocateArray holding arrays from $\mathit{MemSpace}_x$ consisting of bytes that have not yet been allocated in the heap of process x. Empty when process x is not active.

Responses a queue of triples (x, y, z), $x \in \{KILLPROC, KILLPROCRESULT, CREATEPROCRESULT, MALLOC, MEMREAD, MEMWRITE, REGLOADRESULT\}$, $y \in \mathit{Procs}, z \in \{\mathbf{N}, FAIL, DONE, 0\}$. Initially *Empty*.

Actions

```
Input Malloc_x(n)
  Eff: $p := \mathit{Freelist}_x.\mathit{NewBlock}(n)$
       if $p = FAIL$ then
           $\mathit{Responses}.\mathit{Enqueue}$
               $(MALLOC, x, FAIL)$
       else
           $\mathit{Freelist}_x.\mathit{RemoveBlock}(p)$
           $\mathit{Used}_x.\mathit{AddBlock}(p, n)$
           $\mathit{Responses}.\mathit{Enqueue}(MALLOC, x, p)$

Input Free_x(p)
  Eff: if $\mathit{Used}_x.\mathit{FindBlock}(p)$
           $\mathit{Freelist}_x.\mathit{AddBlock}(p, n)$
           $\mathit{Used}_x.\mathit{RemoveBlock}(p)$

Input MemRead_x(p)
  Eff: if $\mathit{Used}_x.\mathit{FindBlock}(p)$
           $\mathit{Responses}.\mathit{Enqueue}(MEMREAD, x, p)$
       else
           $\mathit{Responses}.\mathit{Enqueue}(KILLPROC, x)$

InputMemWrite_x(p, v)
  Eff: if $\mathit{Used}_x.\mathit{FindBlock}(p)$
           $\mathit{Responses}.\mathit{Enqueue}(MEMWRITE, x, p, v)$
       else
           $\mathit{Responses}.\mathit{Enqueue}(KILLPROC, x)$

Input CreateProc(x)
  Eff: $y := DONE$ or $y := FAIL$
           non-deterministically
       if $y = DONE$
           $\mathit{Freelist}_x.\mathit{Size} := \mathit{SPACESIZE}$
           $\mathit{Used}_x := \emptyset$
       $\mathit{Responses}.\mathit{Enqueue}$
           $(CREATEPROC, x, y)$

Input KilledProc(x)
  Eff: $y := DONE$ or $y := FAIL$
           non-deterministically
           $\mathit{Freelist}_x := \emptyset$
       $\mathit{Responses}.\mathit{Enqueue}(KILLEDPROC, x, y)$
```

Figure 3: Memory Manager (User Level Specification)

Memory Manager

Actions (Cont.):

Output KilledProcResult(x, y)
Pre: $Responses.Head =$
$(KILLEDPROC, x, y)$
Eff: $Responses.Dequeue$

Output MemWriteDone$_x$
Pre: $Responses.Head = (MEMWRITE, x, p, v)$
Eff: $MemSpace_x[p] := v$
$Responses.Dequeue$

Output MallocResult$_x(p)$
Pre: $Responses.Head = (MALLOC, x, p)$
Eff: $Responses.Dequeue$

Output MemReadResult$_x(v)$
Pre: $Responses.Head = (MEMREAD, x, p)$
Eff: $v := MemSpace_x[p]$
$Responses.Dequeue$

Output KillProc(x)
Pre: $Responses.Head = (KILLPROC, x)$
Eff: $Responses.Dequeue$

Output CreateProcResult(x, y)
Pre: $Responses.Head =$
$(CREATEPROC, x, y)$
Eff: $Responses.Dequeue$

Figure 4: Memory Manager (User Level Specification) Cont.

their respective output actions CreateProcResult and KilledProcResult decides non-deterministically whether or not the action will succeed, and sends a response to the Process Manager. The non-determinism in the output actions simulates the interaction between the memory module and the file system that occurs in real implementations of operating systems. In creating a process space, an operating system tries to allocate swap space in the file system for the virtual memory of a process, and may find out that there isn't enough disk space to allocate the swap. This results in a failure to create the process that is unrelated to the state of the main memory. Similarly, in the killing of a process, the file system may fail to delete the swap space of a process, resulting in a failure that is outside the control of the Memory Manager. We therefore model this potential for failure with a non-deterministic failure when creating or killing a process.

The output KillProc is output whenever a process requests to read or write memory that it does not own. In this case the Memory Manager determines that the process is attempting an illegal action, and sends the KillProc command to the Process Manager requesting that the process be terminated.

4.4 Process Manager

The User Level specification of the Process Manager is an automaton that controls the life cycles of processes. It is responsible for handling actions that activate and deactivate processes, including a

Process Manager

Signature

Input:
- $\mathsf{Halt}()$
- $\mathsf{Exit}_x()$
- $\mathsf{Spawn}(y)$, $y \in \mathit{Procs}$
- $\mathsf{OnOff}(b)$, $b \in \{0,1\}$
- $\mathsf{KillProc}(x)$, $x \in \mathit{Procs}$
- $\mathsf{KilledProcResult}(x,v)$, $x \in \mathit{Procs}$, $v \in \{DONE, FAIL\}$
- $\mathsf{CreateProcResult}(x,v)$, $x \in \mathit{Procs}$, $v \in \{DONE, FAIL\}$

Output:
- $\mathsf{Activate}_x(b)$, $b \in \{0,1\}$
- $\mathsf{KilledProc}(x)$, $x \in \mathit{Procs}$
- $\mathsf{CreateProc}(x)$, $x \in \mathit{Procs}$

State

Active, $\subseteq \mathit{Procs}$, the set of all active processes, initially $\emptyset$.

Jobs a queue of doubles (x,y), $x \in \{CREATEPROC, KILLEDPROC\}, y \in \mathit{Procs}$. Initially *Empty*.

Responses, a queue of pairs (x,y), $x \in \mathit{Procs}$, $y \in \{0,1\}$. Initially *Empty*.

Shell, a special process which is activated when the process handler is turned on.

On, $\in$ *Booleans*, TRUE if the machine is on. Initially FALSE.

Actions

```
Input Halt_x
    Eff: ∀x ∈ Active
             Jobs.Enqueue(KILLEDPROC, x)

Input Exit_x
    Eff: Jobs.Enqueue(KILLEDPROC, x)

Input OnOff(x)
    Eff: if x = 1 ∧ On = FALSE
             On := TRUE
             Jobs.Enqueue(CREATEPROC, Shell)
         else if x = 0 ∧ On = TRUE
             On := FALSE
             ∀x ∈ Active
                 Jobs.Enqueue(KILLEDPROC, x)

Input KillProc(x)
    Eff: Jobs.Enqueue(KILLEDPROC, x)

Input Spawn(y)
    Eff: Jobs.Enqueue(CREATEPROC, y)

Input KilledProcResult(x, v)
    Eff: if v = DONE
             Responses.Enqueue(x, 0)

Input CreateProcResult(x, v)
    Eff: if v = DONE
             Responses.Enqueue(x, 1)

Output Activate_x(b)
    Pre: Responses.Head = (x, b)
    Eff: if b = 0
             Active := Active − {x}
         else
             Active := Active + {x}
         Jobs.Dequeue

Output KilledProc(x)
    Pre: Jobs.Head = (KILLEDPROC, x)
    Eff: Jobs.Dequeue

Output CreateProc(x)
    Pre: Jobs.Head = (CREATEPROC, x)
    Eff: Jobs.Dequeue
```

Figure 5: Process Manager (User Level Specification)

system OnOff switch, and the Halt, Exit_x, Spawn, and KillProc input actions. The essential control structure of the Process Manager is the set *Active*, which holds all processes that are currently running.

Essentially, all commands to kill a process follow the same path of actions, whether it is Halt or OnOff(0) that kill all processes, or Exit_x or KillProc(x) that kill only one process at a time. An initial input action comes in signalling to the Process Manager that a process x needs to be killed. That action is queued in the *Jobs* queue in the form $(KILLEDPROC, x)$. The output action KilledProc is then output to the Memory Manager, Network Manager and the File System Manager, notifying them that a process was killed. The Memory Manager replies to this action with KilledProcResult, which, as detailed in the Memory Manager section, fails non-deterministically. The Process Manager, having received the result from the Memory Manager, queues a deactivation command in the form $(x, 0)$ in the *Responses* queue, if the result from the KilledProcResult was not $FAIL$. This response is then output through the command $\mathsf{Activate}_x(0)$ to the process, shutting it down.

The path of actions taken to create a process from the OnOff(1) and Spawn(y) commands is very similar to those taken to kill processes. The same process is used to send out the CreateProc output as was used for KilledProc. The same procedure occurs in the Memory Manager, where it non-deterministically outputs a $FAIL$ or $DONE$ for the creation of the process which is received in the CreateProcResult input in the Process Manager. Given a $DONE$ response, this is then sent through the *Responses* queue to the $\mathsf{Activate}_x$ output action.

The *Active* state is only modified in the $\mathsf{Activate}_x$ output, as that action is the single point through which all successful create and kill process actions pass.

It is interesting to note that, in terms of theoretical modelling difficulty, it is very easy to model the concept of an initial bootup *Shell* in the operating sytem. We simply specify a special process to be that shell, and turn it on automatically when the machine is turned on. In practice, one would like such a process to be designed so as to enable user input to request the spawning of other processes. Theoretically, however, such capabilities are not necessary, as the shell is simply the process which is turned on at startup of the computer.

4.5 Console Manager

The Console Manager controls the reading from and writing to the single console of the computer. Essentially, the Console simply keeps the input from the user (not a process, the actual human user) in *Buffer*, and gives it out to the processes that request it, while simultaneously outputting to the screen any data that processes ask it to write to the output console. The only limitation on reading and writing is that no two write commands can be issued at the same time—the console remains busy during a write command, and cannot receive any other write requests until it finishes its current task.

The Console Manager is a relatively simple automaton. A read request comes into the automaton in the form of a $\mathsf{ConsoleRead}_x$ command, which is immediately queued up in the *Jobs* queue in the form $(READ, x)$. This command is then immediately handled by the internal Read command, which enqueues a response in the *Responses* queue of the form $(READRESULT, x, c)$ where c is the first character in *Buffer* if it is not empty, and EOF otherwise. This result is then dispatched to the process through the output ConsoleReadResult.

Write requests occur in almost the same way, with the initial request coming in as $\mathsf{ConsoleWrite}_x(c)$, and being enqueued in the *Jobs* queue as the command $(WRITESCREEN, x, c)$ if the console is not busy writing to the screen already. This is then picked up by the internal action WriteToScreen, which writes the character to the screen, sets the *Busy* variable back to FALSE, and the enqueues a response of $(WRITESCREEN, x, DONE)$ in the *Responses* queue. If the console is busy, then it rejects the write request immediately and enqueues a response of $(WRITERESULT, x, FAIL)$ in the *Responses* queue. In either case, the result is output by the ConsoleWriteResult command, which goes directly to the process which made the write request originally.

4.6 ALU

The ALU at the User Level represents the computational resources provided by the processor. It not only services computational operations, but it stores the results for the processes in sets of per process registers. In keeping with the model of the processes at the User Level, we model the ALU in such a way so as to allow for concurrent access to it from multiple processes. This demands that the ALU be able to serve multiple processes at any given time, switching among their respective

Console Manager

Signature

Input:
- $\mathsf{ConsoleRead}_x$
- $\mathsf{ConsoleWrite}_x(c)$, $c \in \mathit{Chars}$

Output:
- $\mathsf{ConsoleReadResult}_x(y)$, $y \in \mathit{Chars} \cup \{EOF\}$
- $\mathsf{ConsoleWriteResult}_x(y)$, $y \in \{FAIL, DONE\}$

Internal:
- Read
- LoadBuffer
- $\mathsf{WriteToScreen}(c)$

State

Buffer, a queue of *Chars*, the items input from the console and not yet read. Initially *Empty*.

Responses a queue of triples (x, y, z), $x \in \{WRITERESULT, READRESULT\} y \in \mathit{Procs}, z \in \mathit{Chars} \cup \{EOF, \mathit{Busy}, DONE\}$. Initially *Empty*.

Jobs, a queue of pairs $(x, y), x \in \{WRITESCREEN, READ\}, y \in \mathit{Chars}$ to be output to the screen. Initially *Empty*.

Busy, boolean variable. TRUE when the console is writing to the screen. Initially FALSE.

Actions

```
Input ConsoleRead_x
    Eff: $Jobs.Enqueue(READ, x)$

Input ConsoleWrite_x(c)
    Eff: if $Busy$
             $Responses.Enqueue(WRITERESULT,$
                 $x, FAIL)$
         else
             $Busy :=$ TRUE
             $Jobs.Enqueue(WRITESCREEN, x, c)$

Internal Read
    Pre: $Jobs.Head = (READ, x)$
    Eff: if $Buffer$ not empty
             $c := Buffer.Dequeue$
         else
             $c := EOF$
         $Responses.Enqueue(READRESULT, x, c)$

Internal LoadBuffer
    Pre: Input character $c$ from User
    Eff: $Buffer.Enqueue(c)$

Internal WriteToScreen(c)
    Pre: $Jobs.Head = (WRITESCREEN, x, c)$
    Eff: $Jobs.Dequeue$
         $Busy :=$ FALSE
         $Responses.Enqueue(WRITERESULT,$
             $x, DONE)$

Output ConsoleReadResult_x(y)
    Pre: $Responses.Head = (READRESULT, x, y)$
    Eff: $Responses.Dequeue$

Output ConsoleWriteResult_x(y)
    Pre: $Responses.Head = (WRITERESULT, x, y)$
    Eff: $Responses.Dequeue$
```

Figure 6: Console Manager (User Level Specification)

ALU

Signature

Input:
  OperationReq$_x(op)$ $x \in Procs$, $op \in Operations$

Output:
  OperationResult$_x(opVal)$, $x \in Procs$, $opVal \in \mathbf{N}$
  KillProc(x), $x \in Procs$

Internal:
  Operate(x, op), $x\ inProcs$, $op \in Operations$

State

$Registers$, a table of registers for all processes, indexed by process. Initially all $\emptyset$.

$OutstandingOps$, a set of all process-operation pairs for which the operations are still outstanding. Initially $\emptyset$.

$Responses$, a queue of results of the operations, (x, y, z), $x \in \{KILLPROC, OPRESULT\}$, $y \in Procs$, $z \in opVals$. Initially $Empty$.

Actions

```
Input OperationReq$_x(op)$
    Eff: if $\exists y \in Operations$ such that
                $(x, y) \in OutstandingOps$
                $Responses.Enqueue(OPRESULT, x, FAIL)$
         else
                $OutstandingOps := OutstandingOps \cup (x, op)$

Internal Operate$(x, op)$
    Pre: $(x, op) \in OutstandingOps$

    Eff: do operation, store in $Registers[x]$
         if operation raises exception
                $Responses.Enqueue(KILLPROC, x)$
         else
                $Responses.Enqueue$
                     $(OPRESULT, x, opVal)$
         $OutstandingOps := OutstandingOps - (x, op)$

Output OperationResult$_x(opVal)$
    Pre: $Responses.Head =$
                $(OPRESULT, x, opVal)$
    Eff: $Responses.Dequeue$

Output KillProc$(x)$
    Pre: $Responses.Head = (KILLPROC, x)$
    Eff: $Responses.Dequeue$
```

Figure 7: ALU (User Level Specification)

register sets. In order to model this functionality simply, we describe the ALU as being able to directly access all the registers of all the processes. Each process is only allowed to operate on its own registers, and has no knowledge of those belonging to other processes.

ALU requests come in the form of the $\mathsf{OperationReq}_x(op)$ input action, asking the ALU to do operation *op* for process x. The ALU places the operation into the *OutstandingOps* set, in order for it to be executed. The internal action Operate pulls operation requests out of the *OutstandingOps* set, performs the operation, and enqueues the result in the *Responses* queue. If the operation requested resulted in an exception, for example if the process requested to divide by zero, the ALU enqueues a $KILLPROC$ command which is sent to the Process Manager via the output command KillProc. In the usual case of a successful operation, it enqueues an $OPRESULT$ command for the process which requested the operation, resulting in the output action $\mathsf{OperationResult}_x(opVal)$.

4.7 File System Manager

The File System Manager interfaces the processes to the long term data store of the machine. This interface provides the processes with a medium within which they can store data which will persist when the process or computer is turned off. Using a basic tree structure to store the files, each file is stored according to its path and its filename. All files are leaves in the *DirectoryStruct* data structure, and the path of the file dictates the nodes traversed from the root to the file. All file sizes are specified upon creation of the files and do not change throughout the lifetime of the file. The *DirectoryStruct* data structure implements the following five operations:

1. $Add(x, path, dir)$, create a node with name x, add it to the tree in the path specified by *path*. If dir = TRUE make the node a directory, otherwise make it a file. If *path* doesn't exist, or if the last node in the path is a file and not a directory, return $FAIL$.

2. $Remove(x, path)$, delete the node with name x at location *path* from the tree, return $FAIL$ if does not exist.

3. $Find(x, path)$, find the file x at location *path*. Return TRUE if it exists, otherwise FALSE.

4. $Read(x, path, n, m)$, read m bits of data from file x starting at location n. Return FAIL if x is not at least $n + m$ bits long, otherwise return the bit array from n through $n + m$.

File System Manager

Signature

Input:

- $\mathsf{FSOpen}_x(y, path)$, $y \in Files, path \in Paths$
- $\mathsf{FSCreate}_x(y, path, dir, n)$, $y \in Files$, $path \in Paths, n \in \mathbf{N}$
- $\mathsf{FSReadAt}_x(y, path, n, m)$, $y \in Files$, $path \in Paths, n, m \in \mathbf{N}$
- $\mathsf{FSWriteAt}_x(y, path, n, m, A)$, $y \in Files$, $path \in Paths, n, m \in \mathbf{N}, A \in ByteArrays$
- $\mathsf{FSClose}_x(y, path)$, $y \in Files, path \in Paths$
- $\mathsf{FSDelete}_x(y, path)$, $y \in Files, path \in Paths$
- $\mathsf{KilledProc}(x)$, $x \in Procs$

Internal:

Open	Create
ReadAt	WriteAt
Close	Delete

Output:

- $\mathsf{FSOpenResult}_x(y, path, z)$, $y \in Files$, $path \in Paths, z \in Files \cup \{FAIL\}$
- $\mathsf{FSCreateResult}_x(y, path, z)$, $y \in Files$, $path \in Paths, z \in Files \cup \{FAIL\}$
- $\mathsf{FSReadAtResult}_x(y, path, A)$, $y \in Files$, $path \in Paths, A \in ByteArrays \cup \{FAIL\}$
- $\mathsf{FSWriteAtResult}_x(y, path, z)$, $y \in Files$, $path \in Paths, z \in \{DONE, FAIL\}$
- $\mathsf{FSCloseResult}_x(y, path, z)$, $y \in Files$, $path \in Paths, z \in \{DONE, FAIL\}$
- $\mathsf{FSDeleteResult}_x(y, path, z)$, $y \in Files$, $path \in Paths, z \in \{DONE, FAIL\}$

State

$Open_x$, set of files, specificied by name and path, currently held by process x. Initially *Empty*.

DirectoryStruct, a tree structure for storing files by name and path, representing the directory structure of the system. All internal nodes are directories, with files only residing at the leaves of the tree. Each file name is unique from its siblings. Initially *Empty*.

FSCommands, the set $\{OPEN, CLOSE, CREATE, DELETE, READAT, WRITEAT\}$.

FSResults, the set $\{DONE, FAIL\} \cup Bytes \cup ByteArrays$.

Responses a queue of quintuples (v, w, x, y, z), $v \in FSCommands, w \in Procs, x \in Files, y \in Paths, z \in FSResults$. Initially *Empty*.

Jobs a queue of septuples (t, u, v, w, x, y, z), $t \in FSCommands, u \in Procs$, $v \in Paths$, $w \in Files, x \in \mathbf{N}$, $z \in ByteArrays$. Initially *Empty*.

Actions

```
Input FSOpen_x(y, path)
    Eff: $Jobs.Enqueue(OPEN, x, y, path)$

Input FSCreate_x(y, path, dir, n)
    Eff: $Jobs.Enqueue$
             $(CREATE, x, y, path, dir, n)$

Input FSDelete_x(y, path)
    Eff: $Jobs.Enqueue$
             $(DELETE, x, y, path)$

Input FSReadAt_x(y, path, n, m)
    Eff: $Jobs.Enqueue$
             $(READAT, x, y, path, n, m)$

Input FSClose_x(y, path)
    Eff: $Jobs.Enqueue$
             $(CLOSE, x, y, path)$

Input FSWriteAt_x(y, path, n, m, A)
    Eff: $Jobs.Enqueue$
             $(WRITEAT, x, y, path, n, m, A)$

Input KilledProc(x)
    Eff: $Open_x := \emptyset$

Internal Open
    Pre: $Jobs.Head = (OPEN, x, y, path)$
    Eff: if $DirectoryStruct.Find(y, path) \vee$
         $\forall z, (y, path) \cap Open_z = \emptyset$
             $Responses.Enqueue$
                 $(OPEN, x, y, path, DONE)$
             $Open_x := Open_x \cup \{(y, path)\}$
         else
             $Responses.Enqueue$
                 $(OPEN, x, y, path, FAIL)$
         $Jobs.Dequeue$
```

Figure 8: File System Manager (User Level Specification)

File System Manager

Actions (Cont.):

```
Internal Create
    Pre: $Jobs.Head = (CREATE, x, y, path, dir, n)$
    Eff: if $DirectoryStruct.Find(y, path)$
            $Responses.Enqueue$
                $(CREATE, x, y, path, FAIL)$
         else
            $Responses.Enqueue$
                $(CREATE, x, y, path, DONE)$
            $DirectoryStruct.Add(y, path, dir, n)$
         $Jobs.Dequeue$

Internal Delete
    Pre: $Jobs.Head = (DELETE, x, y, path)$
    Eff: if $DirectoryStruct, Find(y, path) \wedge$
         $\forall x, y \notin Open_x$
            $DirectoryStruct.Remove(y, path)$
            $Responses.Enqueue$
                $(DELETE, x, y, path, DONE)$
         else
            $Responses.Enqueue$
                $(DELETE, x, y, path, FAIL)$
         $Jobs.Dequeue$

Internal ReadAt
    Pre: $Jobs.Head = (READAT, x, y, path, n, m)$
    Eff: if $y \in Open_x$
            $A := DirectoryStruct.Read(y, path, n, m)$
            $Responses.Enqueue$
                $(READAT, x, y, path, A)$
         $Jobs.Dequeue$

Internal Close
    Pre: $Jobs.Head = (Close, x, y, path)$
    Eff: if $(y, path) \in Open_x$
            $Open_x := Open_x - \{(y, path)\}$
            $Responses.Enqueue$
                $(CLOSE, x, y, path, DONE)$
         else
            $Responses.Enqueue$
                $(CLOSE, x, y, path, FAIL)$
         $Jobs.Dequeue$

Internal WriteAt
    Pre: $Jobs.Head =$
             $(WRITEAT, x, y, path, n, m, A)$
    Eff: if $(y, path) \in Open_x$
             $result =$
                 $DirectoryStruct.Write(y, path, n, m, A)$
         else
             $result = FAIL$
         $Responses.Enqueue$
             $(WRITEAT, x, y, path, result)$
         $Jobs.Dequeue$

Output FSOpenResult$_x(y, path, z)$
    Pre: $Responses.Head = (OPEN, x, y, path, z)$
    Eff: $Responses.Dequeue$

Output FSCreateResult$_x(y, path, z)$
    Pre: $Responses.Head = (CREATE, x, y, path, z)$
    Eff: $Responses.Dequeue$

Output FSCloseResult$_x(y, path, z)$
    Pre: $Responses.Head = (CLOSE, x, y, path, z)$
    Eff: $Responses.Dequeue$

Output FSDeleteResult$_x(y, path, z)$
    Pre: $Responses.Head = (DELETE, x, y, path, z)$
    Eff: $Responses.Dequeue$

Output FSReadAtResult$_x(y, path, A)$
    Pre: $Responses.Head = (READAT, x, y, path, A)$
    Eff: $Responses.Dequeue$

Output FSWriteAtResult$_x(y, path, z)$
    Pre: $Responses.Head = (WRITEAT, x, y, path, z)$
    Eff: $Responses.Dequeue$
```

Figure 9: File System Manager (User Level Specification) Cont.

5. $Write(x, path, n, m, A)$, write A, a bit array of length m, to file x starting at location n. Return FAIL if x is not at least $n + m$ bits long, otherwise DONE.

All file system actions, with the exception of KillProc, deal with file manipulation and communicate directly with the processes. Much like the automata we have seen earlier, all of the file related input actions of the File System Manager enqueue their input request into the internal *Jobs* queue, passing with it all arguments necessary to process the request. These requests are then served by the internal actions whose names correspond to the input actions. Once served, the response to the action is then enqueued into the *Responses* queue, from which the corresponding output action outputs the result back to the process that originally requested the action.

For each process x, the $Open_x$ set maintains all the files which it currently has opened. When a process is killed, the File System receives the KillProc signal from the Process Handler, signalling it to set $Open_x$ to $\emptyset$.

The issues involved with the Open, Create, Delete, and Close operations deal mainly with checking to see if a given file exists or is open. Open only returns successfully if the file exists and is not open by any process. Create only succeeds if the file does not exist, while Delete succeeds only if the file exists and is not open by any process. Finally, Close succeeds only on existing files which were initially opened by the process that attempts the close action.

The ReadAt and WriteAt commands work only on open, existing files, where the read and write requests fall within the size limits of the file.

4.8 Network Manager

The User Level specification of the Network Manager describes an automaton that provides the interface for processes to communicate over the network. The Network Manager keeps track of open connections held by processes, and allows processes to read and write from the network. It continuously reads in data from each socket into its respective *InBuffer*, and outputs data for each socket from its respective *OutBuffer*. The socket connections are made on a per process basis, so no more than one socket is allowed open at a given time between two processes x and y.

Like the File System Manager, all the external actions of the Network Manager, with the exception of the KillProc input action, go directly between the Network Manager and the currently

Network Manager

Signature

Input:
- $\mathsf{NetOpen}_x(y)$, $y \in \mathit{Sockets}$
- $\mathsf{NetRead}_x(y, n)$, $y \in \mathit{Sockets}$, $n \in \mathbf{N}$
- $\mathsf{NetWrite}_x(y, A)$, $y \in \mathit{Sockets}$, $A \in \mathit{ByteArrays}$
- $\mathsf{NetClose}_x(y)$, $y \in \mathit{Sockets}$
- $\mathsf{KilledProc}(x)$, $x \in \mathit{Procs}$

Output:
- $\mathsf{NetOpenResult}_x(y, z)$, $y \in \mathit{Sockets}$, $z \in \{DONE, FAIL\}$
- $\mathsf{NetReadResult}_x(y, A)$, $y \in \mathit{Sockets}$, $A \in \mathit{ByteArrays} \cup \{DONE, FAIL\}$
- $\mathsf{NetWriteResult}_x(y, z)$, $y \in \mathit{Sockets}$, $z \in \{DONE, FAIL\}$
- $\mathsf{NetCloseResult}_x(y, z)$, $y \in \mathit{Sockets}$, $z \in \{DONE, FAIL\}$

Internal:
- Open
- Read
- Write
- Close
- $\mathsf{NetReadInBuffer}_{x,y}$, $x \in \mathit{Procs}, y \in \mathit{Sockets}$
- $\mathsf{NetWriteOutBuffer}_{x,y}$, $x \in \mathit{Procs}, y \in \mathit{Sockets}$

State

Open_x, the set of sockets currently opened by process x. Initially $\emptyset$.

$\mathit{Network}$, the set of all computers with which connections can be opened.

$\mathit{InBuffer}_{x,y}$ a queue holding incoming data from socket connection y for process x. Initially *Empty*.

$\mathit{OutBuffer}_{x,y}$ a queue holding outgoing data from socket connection y for process x. Initially *Empty*.

$\mathit{NetCommands}$ the set $\{OPEN, CLOSE, READ, WRITE\}$.

$\mathit{NetResults}$ the set $\{DONE, FAIL, \} \cup \mathit{ByteArrays}$.

$\mathit{Responses}$ a queue of triples (x, y, z), $x \in \mathit{NetCommands}, y \in \mathit{Procs}, z \in \mathit{NetResults}$. Initially *Empty*.

Jobs a queue of quadruples (w, x, y, z), $w \in \mathit{NetCommands}, x \in \mathit{Procs}, y \in \mathbf{N}$, $z \in \mathit{ByteArrays}$. Initially *Empty*.

Actions

```
Input NetOpen_x(y)
    Eff: $Jobs.Enqueue(OPEN, x, y)$

Input NetClose_x(y)
    Eff: $Jobs.Enqueue(CLOSE, x, y)$

Input NetRead_x(y, n)
    Eff: $Jobs.Enqueue(READ, x, y, n)$

Input NetWrite_x(y, n, A)
    Eff: $Jobs.Enqueue(WRITE, x, y, n, A)$

Input KilledProc(x)
    Eff: $Open_x := \emptyset$
         $\forall y \in Network$
             $OutBuffer_{y,x} := \emptyset$

Internal Open
    Pre: $Jobs.Head = (OPEN, x, y)$
    Eff: if $y \in Network \wedge y \notin Open_x$
             $Responses.Enqueue(OPEN, x, y, DONE)$
             $Open_x := Open_x \cup \{y\}$
         else
             $Responses.Enqueue(OPEN, x, y, FAIL)$
         $Jobs.Dequeue$

Internal Close
    Pre: $Jobs.Head = (CLOSE, x, y)$
    Eff: if $y \in Open_x$
             $Open_x := Open_x - \{y\}$
             $Responses.Enqueue(CLOSE, x, y, DONE)$
         else
             $Responses.Enqueue(CLOSE, x, y, FAIL)$
         $Jobs.Dequeue$
```

Figure 10: Network Manager (User Level Specification)

Network Manager

Actions (Cont.):

```
Internal Write
    Pre: $Jobs.Head = (WRITE, x, y, n, A)$
    Eff: if $y \in Open_x$
            if $OutBuffer.Size + n \leq MAXBUF$
                $OutBuffer.Enqueue(A)$
                $Responses.Enqueue$
                    $(WRITE, x, y, DONE)$
         else
            $Responses.Enqueue(WRITE, x, y, FAIL)$
         $Jobs.Dequeue$

Internal Read
    Pre: $Jobs.Head = (READ, x, y, n)$
    Eff: if $y \in Open_x$
            if $InBuffer_{x,y}.Empty$ = FALSE
                $A$ := first $n$ bytes in $InBuffer_{x,y}$
                $InBuffer_{x,y}.Dequeue$ $n$ times
            else
                $A := EOF$
            $Responses.Enqueue(READ, x, y, A)$
         else
            $Responses.Enqueue(READ, x, y, FAIL)$
         $Jobs.Dequeue$

Internal NetReadInBuffer$_{x,y}$
    Pre: Incoming data $A$ from connection
             $y$ to process $x$
    Eff: $InBuffer_{x,y}.Enqueue(A)$

Internal NetWriteOutBuffer$_{x,y}$
    Pre: $OutBuffer_{x,y}.Empty$ = FALSE
    Eff: $OutBuffer_{x,y}.Dequeue$

Output NetOpenResult$_x(y, z)$
    Pre: $Responses.Head = (OPEN, x, y, z)$
    Eff: $Responses.Dequeue$

Output NetCloseResult$_x(y, z)$
    Pre: $Responses.Head = (CLOSE, x, y, z)$
    Eff: $Responses.Dequeue$

Output NetReadResult$_x(y, A)$
    Pre: $Responses.Head = (READ, x, y, A)$
    Eff: $Responses.Dequeue$

Output NetWriteResult$_x(y, z)$
    Pre: $Responses.Head = (WRITE, x, y, z)$
    Eff: $Responses.Dequeue$
```

Figure 11: Network Manager (User Level Specification) Cont.

running processes. These actions are first enqueued in the *Jobs* queue, and then acted upon by their respective internal actions, placing the result in the *Responses* queue.

Also like the File System Manager, each of the internal actions is primarily concerned with checking that actions are only performed on sockets which are closed in the case of Open, and open in the cases of Read, Write, and Close. Otherwise, most of the actions are composed of keeping the $Open_x$ data structure up to date, and properly responding to requests.

The two internal actions $\mathsf{NetReadInBuffer}_{x,y}$ and $\mathsf{NetWriteOutBuffer}_{x,y}$ are responsible for reading in data off of the network and writing out data placed into the buffers by local processes wanting to communicate. Because the interface between the operating system and the network is not relevant to the processes nor the rest of the operating system, we consider this to be an internal action as it has no impact on the rest of our model.

5 Kernel Level Specification

Having seen the User Level model, we understand an abstract specification of an operating system. It now becomes interesting to look at how such a system is implemented. The motivation for a more detailed, reality-driven implementation is to build a model that is meaningful for a real operating system, but still adheres to the goals and ideals set out in the original specification. Such a model would provide researchers with a formal structure to the operating system, enabling clearer thought about its various aspects while still providing enough detail to make the model meaningful. At the same time, this model would be designed so as to implement the abstract specification which is the User Level model, and thereby provide the error-proof, simple process interface which is so desirable.

We choose to model a standard uniprocessor system, selecting as our operating system motivation a simplified UNIX-like model that allows concurrent processes to run independently in their own protected memory spaces. Such an implementation introduces the complexities of Virtual Memory systems, ALU register swapping, interrupt driven I/O and process scheduling. This greatly changes the topology of the operating system, as the number of components and the communication between them increases dramatically. Processes are still viewed as autonomous and concurrent, but in this model they are only given access to valuable resources such as the ALU or

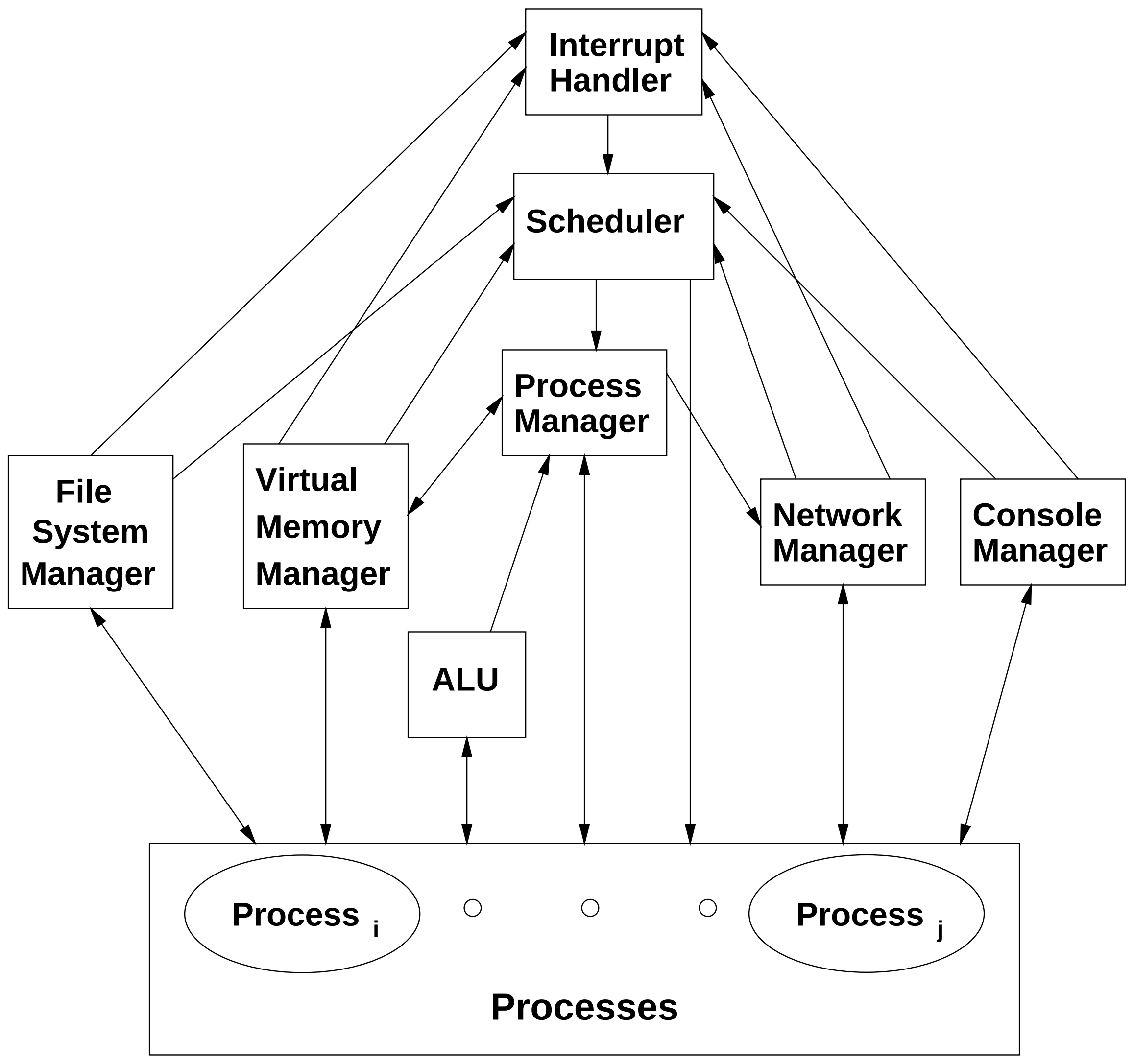

Figure 12: Kernel Level model: Operating System topology

memory as dictated by a scheduling module. No longer is the operating system merely a server of resources to the processes, it owns all of the computational resources, without which a process can do nothing. This model reveals a good deal more about the actual nature of modern operating systems, providing an interesting formal approach to current implementations.

The introduction of the Scheduler and Interrupt Handler each create obvious changes to the operating system topology, as each of them introduces a new automaton into the model. Another change that is equally dramatic and perhaps not as noticeable comes as a result of virtual memory. With virtual memory, the Virtual Memory Manager relies on an internal secondary storage device for expanding its memory capacity, thereby complicating every memory access. This model of the memory management is realistic to real operating system methods, and still adheres to the abstract specification. Similarly, the ALU is modelled more realistically, only holding one set of registers at a time, using the an internal register memory to store all the others.

In the presentation of the User Level model we presented the Process first so as to motivate the definition of the rest of the operating system. In the Kernel Level model we postpone discussing the Process until the end. In keeping with our goal of implementing an operating system that exactly implements the interface of the User Level model, we would like to be able to use the exact same process definition in both models. This requires us, however, to introduce several theoretical notions which enable us to convert the Kernel Level operating system interface into one with which the User Level Process can iteract. These theoretical notions center around the creation of an Abstraction Automaton which acts as a buffer between the User Level Process and the Kernel Level operating system. Because we want to shield the User Level Process from the scheduling which transpires in the operating system, the Abstraction Automaton intercepts the Scheduling commands from the Scheduler to the Processes, and handles all of the scheduling issues itself. We discuss these concepts at the end of the Kernel Level section, and include in the discussion the interactions with the processes.

5.1 Scheduler

The Scheduler does not exist in the User Level Specification, as it is a solution to the implementation issue of limited processor resources. Because only one process can actually run at any given time on a uniprocessor system, the scheduler is responsible for switching between the different processes

Scheduler

Signature

Input:
- InterruptActivate
- Schedule(x, b) $\quad x \in Procs, b \in \{0,1\}$
- Activate(x, b) $\quad x \in Procs, b \in \{0,1\}$

Output:
- $\mathsf{Run}_x(b)$ $\quad x \in Procs, b \in \{0,1\}$

State

Active, the active process. Initially 0.

Ready, a queue of descheduled processes waiting to run. Initially *Empty*.

Sleep, a set of descheduled processes unable to run. Initially, all processes are in *Sleep*.

Responses, a queue of pairs corresponding to scheduling changes. Each pair is (x, y), $x \in Procs$, $y \in \{0,1\}$. Initially *Empty*.

Actions

```
Input InterruptActivate
    Eff: $Ready.Enqueue(Active)$
         $Responses.Enqueue(Active, 0)$
         $Active := Ready.Dequeue$
         $Responses.Enqueue(Active, 1)$

Input Activate$(x, b)$
    Eff: if $b = 0$
             if $Active = x$
                 $Responses.Enqueue(Active, 0)$
                 $Active := Ready.Dequeue$
                 $Responses.Enqueue(Active, 1)$
             else if $x \in Ready$
                 $Ready.Remove(x)$
             else if $x \in Sleep$
                 $Sleep.Remove(x)$
         else if $b = 1$ and $x \notin Active \cup Ready \cup Sleep$
             $Ready.Enqueue(x)$

Input Schedule$(x, b)$
    Eff: if $b = 1 \wedge x \in Sleep$
             $Ready.Enqueue(x)$
             $Sleep := Sleep - \{x\}$
         if $b = 0 \wedge x \notin Sleep$
             $Sleep := Sleep \cup \{x\}$
             if $Active = x$
                 $Responses.Enqueue(x, 0)$
                 $Active := Ready.Dequeue$
                 $Responses.Enqueue(Active, 1)$
             else if $x \in Ready$
                 $Ready.Remove(x)$

Output $\mathsf{Run}_x(b)$
    Pre: $Responses.Head = (x, b)$
    Eff: $Responses.Dequeue$
```

Figure 13: Scheduler (Kernel Level Specification)

such that each process gets a fair share of processor time. All requests to the scheduler come from within the operating system—processes are not allowed to send requests to the scheduler.

The important state of the scheduler consists of the *Active* process and the *Ready* and *Sleep* queues. The *Active* process is the process that is currently scheduled to run. Because we model a single processor machine, there is only one *Active* process. However, this model could easily be generalized to a multi-processor machine by making *Active* the set of all currently running processes. The *Ready* queue is a queue of processes that are ready to run but are currently not scheduled. The *Sleep* queue holds all processes that are waiting on I/O requests to finish. These processes are not ready to be scheduled until the request finishes, at which point they are reawakened and placed back on the *Ready* queue.

The Scheduler simulates a Round Robin scheduling algorithm, and cycles through these processes, putting the just-descheduled process at the back of the queue and scheduling the process at the front of the queue. The decision to model a round-robin scheduling algorithm was arbitrary—any properly fuctioning scheduling algorithm, such as multi-level feedback queues or fair share scheduling, would be fine.

Requests to deschedule processes come through the other manager automata in the operating system. Whenever an I/O operation occurs, whether it be from the Virtual Memory, File System, Console, or Network, the first action taken by each of these modules is to request that the scheduler deactivate the process making the I/O request. In real operating systems, because I/O requests take a relatively long time, the processor idles while a process waits on an I/O response. In order to not waste processor time the waiting process is descheduled until the request is finished, at which point it is reawakened.

Requests to schedule processes come from the Process Manager and the Interrupt Handler. The Process Manager sends the Scheduler Activate output actions, which signals to the scheduler when a process has been created or killed. Killed processes are removed from the data structures in the scheduler, while newly created processes are added to the *Ready* queue. The Interrupt Handler sends the Scheduler a periodic InterruptActivate output which signal the Scheduler to switch equally between processes even without any form of I/O requests. Also, whenever an I/O request is finished, the Interrupt Handler is signalled by the manager handling the request, and it passes this on to the scheduler in the form of a $Schedule(x, 1)$ request. This puts a sleeping process back onto the

Ready queue, enabling it to be rescheduled the next time it is at the head of the queue.

The single output of the Scheduler is the Run command, notifying the process whether it can or cannot run. As we will see later, that action will not go directly to the process but will instead go to the Abstraction Automaton, as we would like to maintain the interface we arrived at with the User Level Process.

5.2 Interrupt Handler

The Interrupt Handler, like the Scheduler, is a module that does not exist in the User Level Specification, but exists as another artifact of limited resources. It is through the Interrupt Handler that I/O devices notify the scheduler that their results are ready, and it is also how the scheduler is signalled periodically to switch processes.

The Interrupt Handler receives only one input action: Interrupt(x). These actions come from the various I/O managers in the operating system, signalling the Interrupt Handler that process x has been served, and that it is ready to be rescheduled. The Interrupt Handler, upon receiving these Interrupt requests, enqueues the process x in the *Responses* queue. From this the output Schedule$(x, 1)$ is sent to the scheduler, and the process is dequeued from the *Responses* queue.

The other output action, InterruptActivate, is not driven as a result of any input actions, but is triggered by an internal clock mechanism that notifies the Scheduler every *TimerTicks* ticks of the clock. The speed at which this clock ticks impacts performance, controlling the rate at which the operating system switches between processes, but it has no effect on the correctness of the system.

5.3 ALU

The Kernel Level ALU provides the same functionality as the User Level ALU, but it does so with more limited resources. Like a real operating system, the ALU can only operate on a single set of registers at a time, and must switch between register sets when receiving operation requests from different processes. We model this action by keeping the current set of registers in the state variable *CurrentRegs*, while internally storing the other registers in an array of memory, *RegMem*. While in most cases in real operating systems, the memory for the registers is stored on the same physical media as the process memory, at this level of abstraction the two have no logical relationship, and

Interrupt Handler

Signature

Input:
  Interrupt(x) $x \in Procs$

Output:
  Schedule($x, 1$) $x \in Procs$
  InterruptActivate

State

Responses, a queue of *Procs*, processes ready to be scheduled. Initially *Empty*.

TimerTicks, number of ticks between periodic activation of the scheduler.

Clock, a monotonically increasing counter, triggered to increment at regularly spaced real-time intervals. Initially 0.

Actions

Input Interrupt(x)
  Eff: $Responses.Enqueue(x)$

Output InterruptActivate
  Pre: $Clock \bmod TimerTicks = 0$
  Eff: None.

Output Schedule($x, 1$)
  Pre: $Responses.Head = x$
  Eff: $Responses.Dequeue$

Figure 14: Interrupt Handler (Kernel Level Specification)

therefore we represent them separately.

The ALU signature is identical to that of the User Level ALU, and the flow of actions occurs in the same way. ALU requests come in the form of the $\mathsf{OperationReq}_x(op)$ input action, asking the ALU to do operation *op* for process x. These requests are placed into the *OutstandingOps* set, from which the Operate action takes its input. When necessary, the ALU switches registers from *RegMem*, preparing the *CurrentRegs* variable for the computation. Once the registers are correctly initiated, the ALU computes the operation and enqueues the result in the *Responses* queue. As in the User Level model, if the operation requested resulted in an exception, the ALU enqueues a $KILLPROC$ command which is sent to the Process Manager. In the usual case of a successful operation, it enqueues an $OPRESULT$ command for the process which requested the operation, resulting in the ouput action $\mathsf{OperationResult}_x(opVal)$.

ALU

Signature

Input:
  $\mathsf{OperationReq}_x(op)$ $x \in \mathit{Procs}$, $op \in \mathit{Operations}$

Internal:
  $\mathsf{Operate}(x, op)$, $x \in \mathit{Procs}$, $op \in \mathit{Operations}$

Output:
  $\mathsf{OperationResult}_x(opVal)$, $x \in \mathit{Procs}$, $opVal \in \mathbf{N}$
  $\mathsf{KillProc}(x)$, $x \in \mathit{Procs}$

State

CurrentRegs, the registers of process currently being serviced. Initially $\emptyset$.

CurrentProc, the current process whose registers are being stored in *CurrentRegs*. Initially 0.

OutstandingOps, a set of process-operation pairs for all outstanding operations. Initially $\emptyset$.

RegMem, an array of registers for all processes, indexed by process. Initially all registers store 0.

Responses, a queue of results of the operations, (x, y, z), $x \in \{KILLPROC, OPRESULT\}$, $y \in \mathit{Procs}$, $z \in opVals$. Initially *Empty*.

Actions

```
Input OperationReq_x(op)
    Eff: if ∃y ∈ Operations such that
              (x, y) ∈ OutstandingOps
              Responses.Enqueue(OPRESULT, x, FAIL)
         else
              OutstandingOps := OutstandingOps ∪ (x, op)

Output OperationResult_x(opVal)
    Pre: Responses.Head =
              (OPRESULT, x, opVal)
    Eff: Responses.Dequeue

Output KillProc(x)
    Pre: Responses.Head = (KILLPROC, x)
    Eff: Responses.Dequeue

Internal Operate(x, op)
    Pre: (x, op) ∈ OutstandingOps

    Eff: if CurrentProc ≠ x
             if CurrentProc ≠ 0
                 Registers.[CurrentProc] :=
                     CurrentRegs
             CurrentProc := x
             CurrentRegs := RegMem[x]
         Conduct operation.
         Place result in CurrentRegs
         if operation raises exception
             Responses.Enqueue(KILLPROC, x)
         else
             Responses.Enqueue
                 (OPRESULT, x, opVal)
         OutstandingOps := OutstandingOps − (x, op)
```

Figure 15: ALU (Kernel Level Specification)

5.4 Virtual Memory Manager

The Virtual Memory Manager is the most complicated module in the Kernel Level Specification. While it implements the same functionality as its User Level analog, the Virtual Memory Manager introduces a whole new level of complexity with swap spaces and page tables. Whether the Process Manager is trying to create a process and needs to allocate a Process Space, or an active process is trying to read from memory, nearly all memory related requests have the potential of requiring page swapping, and swap space access.

The basic motivation for Virtual Memory is twofold: give each running process the illusion that it is the only process on the machine, and enable the capability to run more programs concurrently by using a secondary storage device as a swap space to store program memory that has not been used for a while. The first is solved through the usage of Page Tables, which map the process' virtual memory space to physical addresses. These mappings are done in large blocks of memory, called pages, so as to reduce the size of the page tables. The second goal of enabling more processes is realized by creating a process swap space for each process. The swap space of a process is the same size as its entire memory space, allowing for the process memory to be written to the swap space when the main memory is needed for other operations. Swap space management drives most of the increased complexity in the Virtual Memory Manager.

The internal state of the Virtual Memory Manager is comprised both of data structures which manage individual process spaces and structures which are used for the maintenance of the system memory as a whole. Individual process space state includes $\mathit{PageTable}_x$, $\mathit{Freelist}_x$, and Used_x. $\mathit{PageTable}_x$ is the page table of the process, storing the mapping between virtual memory addresses and the physical locations of those addresses. It is implemented using a PageTable, which is an array indexed by virtual page number, storing at each location the physical page to which the virtual page maps. In addition to storing the physical page, the PageTable stores a $.Valid$ bit for each page, signifying whether the page being indexed has been loaded into main memory. The entire PageTable has a single value $.Size$ which stores the size in pages of the PageTable. Upon creation of a process, the $\mathit{PageTable}$ entries are all marked as invalid, and they are only assigned to physical pages when they are accessed for reads or writes. $\mathit{Freelist}_x$ and Used_x are complements of each other: the first holds the free space available for allocation in the process' heap, while the second contains all the memory that has already been allocated.

Virtual Memory Manager

Signature

Input:

```
    Malloc_x(n), n ∈ N
    Free_x(p), p ∈ N
    MemRead_x(p), p ∈ N
    MemWrite_x(p, v), p ∈ N, v ∈ Bytes
    CreateProc(x), x ∈ Procs
    KilledProc(x), x ∈ Procs
```

Internal:

```
    Swap
```

Output:

```
    MallocResult_x(p), p ∈ N ∪ {FAIL}
    MemReadResult_x(v), v ∈ Bytes
    MemWriteDone_x
    CreateProcResult(x, v), x ∈ Procs,
        v ∈ {DONE, FAIL}
    KilledProcResult(x, v), x ∈ Procs,
        v ∈ {DONE, FAIL}
    KillProc(x), x ∈ Procs
    Interrupt(x), x ∈ Procs
    Schedule(x, 0), x ∈ Procs
```

State

Mem, an array of bytes, indexed by location. Initially all bytes have value 0.

MemPages, a PageArray, the bit array of available pages. All bits are initialized to zero.

$PageTable_x$, a PageTable, the page table for process x. Initially all values are 0.

$Freelist_x$, an AllocateArray, the free list of memory in the heap of process x. Initially *Empty*.

$Used_x$, an AllocateArray, list of memory in the heap that is currently in use by process x. Initially *Empty*.

SwapStore, a directory into which all swap spaces are placed. Initially *Empty*.

PAGESIZE, fixed size of pages in system.

SPACESIZE, fixed size of process space, is a multiple of *PAGESIZE*.

VMCommands, the set $\{MALLOC, READ, WRITE, CREATE, KILL, KILLED, SCHEDULE, INTERRUPT\}$.

VMArguments, the set $\mathbf{N} \cup \{DONE, FAIL\} \cup Bytes \cup ByteArrays$.

Responses, a queue of triples (x, y, z), $x \in VMCommands, y \in Procs, z \in VMArguments$. Initially *Empty*.

Actions

```
Input Malloc_x(n)
    Eff: $Responses.Enqueue(SCHEDULE, x)$
         $p := Freelist_x.NewBlock(n)$
         if $p = FAIL$ then
             $Responses.Enqueue$
                 $(MALLOC, x, FAIL)$
         else
             $Freelist.RemoveBlock(p)$
             $Used_x.AddBlock(p, n)$
             $Responses.Enqueue(MALLOC, x, p)$
         $Responses.Enqueue(INTERRUPT, x)$

Input Free_x(p)
    Eff: $Responses.Enqueue(SCHEDULE, x)$
         if $Used_x.FindBlock(p)$
             $Freelist_x.AddBlock(p, n)$
             $Used_x.RemoveBlock(p)$
         $Responses.Enqueue(INTERRUPT, x)$

Input CreateProc(x)
    Eff: if $SwapStore.Add(x, SPACESIZE) = FAIL$
             $Responses.Enqueue(CREATE, x, FAIL)$
         else
             $PageTable_x.Size := SPACESIZE$
             for $i := 1$ to $PageTable_x.Size$ do
                 $PageTable_x[i].Valid :=$ FALSE
             $Freelist_x.Size := SPACESIZE$
             $Used_x := \emptyset$
             $Responses.Enqueue$
                 $(CREATE, x, DONE)$
```

Figure 16: Virtual Memory Manager (Kernel Level Specification)

Virtual Memory Manager

Actions (cont.)

```
Input MemRead_x(p)
  Eff: $VirtPage := \lfloor p/PAGESIZE \rfloor$
       if $Used_x.FindBlock(p)$
           $Responses.Enqueue(SCHEDULE, x)$
           if $PageTable_x[VirtPage].Valid = \mathrm{FALSE}$
               $Responses.Enqueue$
                   $(SWAP, READ, x, p)$
           else
               $Responses.Enqueue(READ, x, p)$
       else
           $Responses.Enqueue(KILL, x)$

Input KilledProc(x)
  Eff: for $i := 1$ to $PageTable_x.Size$ do
           if $PageTable_x[i].Valid = \mathrm{TRUE}$
               $PhysPage := PageTable_x[i].PhysPage$
               $MemPages.Free(PhysPage)$
       $PageTable_x.Size := 0$
       if $SwapStore.Remove(x) = DONE$
           $Responses.Enqueue(KILLED, x, DONE)$
       else
           $Responses.Enqueue(KILLED, x, FAIL)$

Input MemWrite_x(p, v)
  Eff: $VirtPage := \lfloor p/PAGESIZE \rfloor$
       if $Used_x.FindBlock(p)$
           $Responses.Enqueue(SCHEDULE, x)$
           if $PageTable_x[VirtPage].Valid = \mathrm{FALSE}$
               $Responses.Enqueue$
                   $(SWAP, WRITE, x, p, v)$
           else
               $Responses.Enqueue(WRITE, x, p, v)$
       else
           $Responses.Enqueue(KILL, x)$

Output CreateProcResult(x, y)
  Pre: $Responses.Head = (CREATE, x, y)$
  Eff: $Responses.Dequeue$

Output KilledProcResult(x, y)
  Pre: $Responses.Head = (KILLED, x, y)$
  Eff: $Responses.Dequeue$

Output MemWriteDone_x
  Pre: $Responses.Head = (WRITE, x, p, v)$
  Eff: $VirtPage := \lfloor p/PAGESIZE \rfloor$
       $PhysPage := PageTable_x[VirtPage].PhysPage$
       $Mem[PhysPage + p \bmod PAGESIZE] := v$
       $MemPages.UsedUpdate(PhysPage)$
       $Responses.Enqueue(INTERRUPT, x)$
       $Responses.Dequeue$

Internal Swap
  Pre: $Responses.Head =$
           $(\alpha = (SWAP, Command, x, p, v))$
  Eff: $VirtPage := \lfloor p/PAGESIZE \rfloor$
       $PhysPage := MemPages.NextUnused$
       if $PhysPage = INVALID$
           $PhysPage :=$
               $MemPages.LeastRecentlyUsed$
           $PageTable_x[VirtPage] := PhysPage$
           $A := PAGESIZE$ sized array
               containing $PhysPage$
           $P := MemPages[PhysPage].Process$
           $WritePage :=$
               $MemPages[PhysPage].VirtPage$
           $n := WritePage * PAGESIZE$
           $m := n + PAGESIZE$
           $SwapStore.Write(P, A, n, m)$
       $n := VirtPage * PAGESIZE$
       $m := n + PAGESIZE$
       $Mem[PhysPage] := SwapStore.Read(x, n, m)$
       $Mem[PhysPage].Process := x$
       $Mem[PhysPage].VirtPage := VirtPage$
       $PageTable_x[VirtPage].Valid := \mathrm{TRUE}$
       $PageTable_x[VirtPage].PhysPage := PhysPage$
       $Responses.Enqueue(Command, x, p, v)$
       $\alpha := (Command, x, p, v)$

Output MemReadResult_x(v)
  Pre: $Responses.Head = (READ, x, p)$
  Eff: $VirtPage := \lfloor p/PAGESIZE \rfloor$
       $PhysPage := PageTable_x[VirtPage].PhysPage$
       $v := Mem[PhysPage + p \bmod PAGESIZE]$
       $MemPages.UsedUpdate(PhysPage)$
       $Responses.Enqueue(READ, x, v)$
       $Responses.Enqueue(INTERRUPT, x)$
       $Responses.Dequeue$

Output MallocResult_x(p)
  Pre: $Responses.Head = (MALLOC, x, p)$
  Eff: $Responses.Dequeue$

Output KillProc(x)
  Pre: $Responses.Head = (KILL, x)$
  Eff: $Responses.Dequeue$

Output Interrupt(x)
  Pre: $Responses.Head = (INTERRUPT, x)$
  Eff: $Responses.Dequeue$

Output Schedule(x, 0)
  Pre: $Responses.Head = (SCHEDULE, x)$
  Eff: $Responses.Dequeue$
```

Figure 17: Virtual Memory Manager (Kernel Level Specification) Cont.

System-wide memory management is stored in *Mem*, *MemPages*, and *SwapStore*. *Mem* is the actual memory of the system, and therefore contains all the memory values at all locations. *MemPages* stores information about the state of each of the physical pages in memory. For each page, *MemPages* stores a bit signifying whether it is in use (1) or available (0), as well as the value *.Process*, storing the Process which currently is mapping to that physical page, and *.VirtPage*, the virtual page number in that process' PageTable which points to the physical page. *MemPages* implements the following functions:

- *.NextUnused*, returns the next unused page in memory, if one exists, and marks it as used. Otherwise, it returns $INVALID$.
- *.LeastRecentlyUsed*, returns the page in memory least recently used
- *.UsedUpdate*(*Page*), updates the state used to calculate the *LeastRecentlyUsed* page. The page *Page* is updated as having just been used.
- *.Free*(*Page*), marks *Page* as unused.

SwapStore is a set of swap spaces, each one identified by the name of the process for which it is providing swap space. Functionally, the *SwapStore* structure is very similar to the *DirectoryStruct* in the File System; the main difference is that *SwapStore* lacks any sort of hierarchical directory strucuture, and is simply one large set of files. *SwapStore* implements the following five operations:

1. *Add*(x), create a swap space with name x, if it exists return FAIL, otherwise return DONE.
2. *Remove*(x), delete the swap space with name x, return $FAIL$ if does not exist.
3. *Find*(x), find the swap space with name x. Return TRUE if it exists, otherwise FALSE.
4. *Read*(x, n, m), read m bits of data from swap space x starting at location n. Return FAIL if x is not at least $n + m$ bits long, otherwise return the bit array from n through $n + m$.
5. $Write(x, A, n, m)$, write A, a bit array of length m, to swap space x starting at location n. Return FAIL if x is not at least $n + m$ bits long, otherwise DONE.

The two simplest action paths originate from Malloc and Free requests. These correspond closely with their counterparts in the User Level mode. In the case of Malloc, the state changes are made in the Malloc input action, and the result is enqueued in the *Responses* queue, where it is output by MallocResult. Free, on the other hand, has no corresponding output, and the transaction finishes within the action. Each action additionally deschedules the process which made the request at the beginning of the action, and sends an interrupt to the Interrupt Handler to wake up the process at the end of the transaction.

The input actions of MemRead and MemWrite, although internally more complicated, result in paths of actions similar to Malloc. First the request is checked to see if it is for a valid memory location. If it is not, the command $KILL$ is enqueued in the *Responses* queue, and a request is sent to the Process Manager to kill the process which made the illegal request. If the request is valid, the action Schedule$(x, 0)$ is first output to the scheduler. Then the data which is to be read or written is checked to see if it is currently in the main memory. If the data has been swapped out, the memory is checked for any unused pages. If no pages are unused, the least recently used page is swapped out, and the desired page is read into that location in main memory. If there is an unused page, no pages are swapped out, and the desired page is read into one of the unused pages. Once the data has been loaded into memory, it is read or written to, depending on the action, and the result is enqueued in the *Responses* queue, followed by the enqueuing of an interrupt requeswtin the *Responses* queue, requesting that the process making the request be reawakened. The action is then finished either by MemReadResult for read requests, or by MemWriteDone for write requests.

The remaining two input actions are CreateProc and KilledProc. Each of these is less complicated than MemRead and MemWrite. In the case of CreateProc, the Virtual Memory Manager attempts to create a new swap space, and if it fails, it enqueues a failed result in the *Responses* queue, to be output by CreateProcResult. Otherwise it initializes the process specific data structures for a new process space and enqueues a success result in the *Responses* queue, which is also eventually output by CreateProcResult. KilledProc acts very similarly, attempting to kill a process swap space, and then eliminating any process state it was storing for the now defunct process.

5.5 Console, Network, File System and Process Manager

The differences in these four automata from their User Level analogs are minimal for the Console, File System, and Network Managers, and actually nonexistent for the Process Manager.

The Console, File System and Network Managers at the Kernel Level implementation incorporate the existence of the Scheduler and the Interrupt Handler into their previous forms in the User Level Specification. Essentially the same, each now makes sure to deschedule a process before committing to the I/O operation, and only reschedules it upon completion.

The changes to the Console Manager automaton are minor—there is the addition of the $\mathsf{Schedule}$ and $\mathsf{Interrupt}(x)$ outputs, and code in the other actions to enable those outputs. Each input action begins with the enqueuing of the command $(SCHEDULE, x)$ in the *Jobs* queue, which turns into the $\mathsf{Schedule}(x, 0)$ output. Similarly, each transaction ends with the enqueueing of the command $(INTERRUPT, x)$ into the *Responses* queue, which is output as the action $\mathsf{Interrupt}_x$.

Changes to the File System and Network Manager automata are equally minor and similar. The machinery for scheduling and interrupting processes is integrated into the models, but otherwise the baseline functionality remains the same.

It is interesting to note how few changes are made in these four systems, despite the introduction of relatively large changes in the operating system topology. It becomes apparent at this stage that there is a benefit in clearly modelling the separate system components, as such separation of functionality makes it easy and efficient to port unchanged modules from one model to the next. Because the life cycles of the processes, their interactions with the console, their need for persistent storage, and their usage of the network have not changed, neither do the modules that handle these resources.

6 Abstraction Automaton

In designing a Kernel Level model that implements the User Level specification, we need a system that maintains the same interface with the Process as originally stated in the User Level design. In order to maintain the signature of the User Level Process for the processes in the Kernel Level model, it is necessary to introduce a theoretical interface automaton into the Kernel Level system.

Console Manager

Signature

Input:
- $\mathsf{ConsoleRead}_x$
- $\mathsf{ConsoleWrite}_x(c)$, $c \in Chars$

Internal:
- Read
- $\mathsf{LoadBuffer}$
- $\mathsf{WriteToScreen}(c)$

Output:
- $\mathsf{ConsoleReadResult}_x(y)$, $y \in Chars \cup \{EOF\}$
- $\mathsf{ConsoleWriteResult}_x(y)$, $y \in \{FAIL, DONE\}$
- $\mathsf{Schedule}(x, 0)$, $x \in Procs$
- $\mathsf{Interrupt}(x)$

State

Buffer, a queue of *Chars*, the items input from the console and not yet read. Initially *Empty*.

Responses a queue of triples (x, y, z), $x \in \{READRESULT, WRITERESULT, INTERRUPT\}, y \in Procs, z \in Chars \cup \{EOF, Busy, DONE\}$. Initially *Empty*.

Jobs queue of pairs (x, y), $x \in \{READ, WRITESCREEN, SCHEDULE\}, y \in Procs, z \in Chars\}$. Initially *Empty*.

Busy, boolean variable. TRUE when the console is writing to the screen. Initially FALSE.

Actions

```
Input ConsoleRead_x
    Eff: $Jobs.Enqueue(SCHEDULE, x)$
         $Jobs.Enqueue(READ, x)$

Internal LoadBuffer
    Pre: Input character $c$ from User
    Eff: $Buffer.Enqueue(c)$

Internal WriteToScreen(c)
    Pre: $Jobs.Head = (WRITESCREEN, x, c)$
    Eff: $Jobs.Dequeue$
         $Busy := \text{FALSE}$
         $Responses.Enqueue$
             $(WRITERESULT, x, DONE)$
         $Responses.Enqueue(INTERRUPT, x)$

Internal Read
    Pre: $Jobs.Head = (READ, x)$
    Eff: if $Buffer$ not empty
             $c := Buffer.Dequeue$
         else
             $c := EOF$
         $Jobs.Dequeue$
         $Responses.Enqueue(READRESULT, x, c)$
         $Responses.Enqueue(INTERRUPT, x)$

Input ConsoleWrite_x(c)
    Eff: if $Busy$
             $Responses.Enqueue$
                 $(WRITERESULT, x, FAIL)$
         else
             $Busy := \text{TRUE}$
             $Jobs.Enqueue(SCHEDULE, x)$
             $Jobs.Enqueue(WRITESCREEN, x, c)$

Output ConsoleReadResult_x(y)
    Pre: $Responses.Head = (READRESULT, x, y)$
    Eff: $Responses.Dequeue$

Output ConsoleWriteResult_x(y)
    Pre: $Responses.Head = (WRITERESULT, x, y)$
    Eff: $Responses.Dequeue$

Output Schedule(x, 0)
    Pre: $Jobs.Head = (SCHEDULE, x)$
    Eff: $Jobs.Dequeue$

Output Interrupt(x)
    Pre: $Responses.Head = (INTERRUPT, x)$
    Eff: $Responses.Dequeue$
```

Figure 18: Console Manager (Kernel Level Specification)

Network Manager

Signature

Input:
- $\mathsf{NetOpen}_x(y)$, $x \in Procs$, $y \in Sockets$
- $\mathsf{NetRead}_x(y, n)$, $x \in Procs$, $y \in Sockets, n \in \mathbf{N}$
- $\mathsf{NetWrite}_x(y, n, A)$, $x \in Procs$, $y \in Sockets$, $n \in \mathbf{N}, A \in ByteArrays$
- $\mathsf{NetClose}_x(y)$, $x \in Procs$, $y \in Sockets$
- $\mathsf{KilledProc}(x)$, $x \in Procs$

Internal:
- $\mathsf{NetReadInBuffer}_{x,y}$, $x \in Procs, y \in Bytes$
- $\mathsf{NetWriteOutBuffer}_{x,y}$, $x \in Procs, y \in Bytes$
- Open
- Read
- Write
- Close

Output:
- $\mathsf{NetOpenResult}_x(y, z)$, $x \in Procs$, $y \in Sockets$, $z \in \{DONE, FAIL\}$
- $\mathsf{NetReadResult}_x(A, z)$, $x \in Procs$, $y \in Sockets$, ¿$A \in ByteArrays \cup \{DONE, FAIL\}$
- $\mathsf{NetWriteResult}_x(y, z)$, $x \in Procs$, $y \in Sockets$, $z \in \{DONE, FAIL\}$
- $\mathsf{NetCloseResult}_x(y, z)$, $x \in Procs$, $y \in Sockets$ $z \in \{DONE, FAIL\}$
- $\mathsf{Schedule}(x, 0)$, $x \in Procs$
- $\mathsf{Interrupt}(x)$, $x \in Procs$

State

$Open_x$, the set of sockets currently opened by process x. Initially *Empty*.

Network, the set of all computers with which connections can be opened.

$InBuffer_{x,y}$ a queue holding incoming data from socket connection y for process x. Initially *Empty*.

$OutBuffer_{x,y}$ a queue holding outgoing data from socket connection y for process x. Initially *Empty*.

NetCommands the set $\{OPEN, CLOSE, READ, WRITE, INTERRUPT, SCHEDULE\}$.

NetResults the set $\{DONE, FAIL, \} \cup ByteArrays$.

Jobs a queue of triples (x, y, z), $x \in Procs, y \in NetCommands, z \in NetResults$. Initially *Empty*.

Responses a queue of triples (x, y, z), $x \in Procs, y \in NetCommands, z \in NetResults$. Initially *Empty*.

Actions

```
Input NetOpen_x(y)
    Eff: $Jobs.Enqueue(SCHEDULE, x)$
         $Jobs.Enqueue(OPEN, x, y)$

Input NetRead_x(y, n)
    Eff: $Jobs.Enqueue(SCHEDULE, x)$
         $Jobs.Enqueue(READ, x, y, n)$

Input NetClose_x(y)
    Eff: $Jobs.Enqueue(SCHEDULE, x)$
         $Jobs.Enqueue(CLOSE, x, y)$

Input KilledProc(x)
    Eff: $Open_x := EMPTY$
         $\forall y \in Network$
             $OutBuffer_{y,x} := \emptyset$

Input NetWrite_x(y, n, A)
    Eff: $Jobs.Enqueue(SCHEDULE, x)$
         $Jobs.Enqueue(WRITE, x, y, n, A)$

Internal Open
    Pre: $Jobs.Head = (OPEN, x, y)$
    Eff: if $y \in Network \wedge y \notin Open_x$
             $Responses.Enqueue(OPEN, x, y, DONE)$
             $Open_x := Open_x \cup \{y\}$
         else
             $Responses.Enqueue(OPEN, x, y, FAIL)$
         $Responses.Enqueue(INTERRUPT, x)$
         $Jobs.Dequeue$
```

Figure 19: Network Manager (Kernel Level Specification)

Network Manager

Actions: (cont.)

```
Internal Close
    Pre: $Jobs.Head = (CLOSE, x, y)$
    Eff: if $y \in Open_x$
             $Open_x := Open_x - \{y\}$
             $Responses.Enqueue(CLOSE, x, y, DONE)$
         else
             $Responses.Enqueue(CLOSE, x, y, FAIL)$
         $Responses.Enqueue(INTERRUPT, x)$
         $Jobs.Dequeue$

Internal Read
    Pre: $Jobs.Head = (READ, x, y, n)$
    Eff: if $y \in Open_x$
             if $InBuffer_{x,y}.Empty =$ FALSE
                 $A :=$ first $n$ bytes in buffer
             else
                 $A := EOF$
             $Responses.Enqueue(READ, x, y, A)$
         else
             $Responses.Enqueue(READ, x, y, FAIL)$
         $Responses.Enqueue(INTERRUPT, x)$
         $Jobs.Dequeue$

Internal Write
    Pre: $Jobs.Head = (WRITE, x, y, n, S)$
    Eff: if $y \in Open_x$
             if $OutBuffer.Size + n \leq MAXBUF$
                 $OutBuffer.Enqueue(A)$
                 $Responses.Enqueue$
                     $(WRITE, x, y, DONE)$
         else
             $Responses.Enqueue$
                 $(WRITE, x, y, FAIL)$
         $Responses.Enqueue(INTERRUPT, x)$
         $Jobs.Dequeue$

Internal NetReadInBuffer$_{x,y}$
    Pre: Incoming data $A$ from
             connection $y$ to process $x$

    Eff: $InBuffer.Enqueue(A)$

Internal NetWriteOutBuffer$_{x,y}$
    Pre: $OutBuffer.Empty =$ FALSE

    Eff: $OutBuffer.Dequeue$

Output NetOpenResult$_x(y, z)$
    Pre: $Responses.Head = (OPEN, x, y, z)$
    Eff: $Responses.Dequeue$

Output NetCloseResult$_x(y, z)$
    Pre: $Responses.Head = (CLOSE, x, y, z)$
    Eff: $Responses.Dequeue$

Output NetReadResult$_x(A, z)$
    Pre: $Responses.Head = (READAT, x, A, z)$
    Eff: $Responses.Dequeue$

Output NetWriteResult$_x(y, z)$
    Pre: $Responses.Head = (WRITEAT, x, y, z)$
    Eff: $Responses.Dequeue$

Output Schedule$(x, 0)$
    Pre: $Jobs.Head = (SCHEDULE, x)$

    Eff: $Jobs.Dequeue$

Output Interrupt$(x)$
    Pre: $Responses.Head = (INTERRUPT, x)$

    Eff: $Responses.Dequeue$
```

Figure 20: Network Manager (Kernel Level Specification) Cont.

File System Manager

Signature

Input:

- $\mathsf{FSOpen}_x(y, path), y \in \mathit{Files}, path \in \mathit{Paths}$
- $\mathsf{FSCreate}_x(y, path, n), y \in \mathit{Files}, path \in \mathit{Paths}, n \in \mathbf{N}$
- $\mathsf{FSReadAt}_x(y, path, n, m), y \in \mathit{Files}, path \in \mathit{Paths}, n, m \in \mathbf{N}$
- $\mathsf{FSWriteAt}_x(y, path, n, m, A), y \in \mathit{Files}, path \in \mathit{Paths}, n, m \in \mathbf{N}, A \in \mathit{ByteArrays}$
- $\mathsf{FSClose}_x(y, path), y \in \mathit{Files}, path \in \mathit{Paths}$
- $\mathsf{FSDelete}_x(y, path), y \in \mathit{Files}, path \in \mathit{Paths}$
- $\mathsf{KilledProc}(x),\ x \in \mathit{Procs}$

Internal:

Open	Create
ReadAt	WriteAt
Close	Delete

Output:

- $\mathsf{FSOpenResult}_x(y, path, z), y \in \mathit{Files}, path \in \mathit{Paths}, z \in \{DONE, FAIL\}$
- $\mathsf{FSCreateResult}_x(y, path, z), y \in \mathit{Files}, path \in \mathit{Paths}, z \in \{DONE, FAIL\}$
- $\mathsf{FSReadAtResult}_x(y, path, A), y \in \mathit{Files}, path \in \mathit{Paths}, A \in \mathit{ByteArrays} \cup \{FAIL\}$
- $\mathsf{FSWriteAtResult}_x(y, path, z), y \in \mathit{Files}, path \in \mathit{Paths}, z \in \{DONE, FAIL\}$
- $\mathsf{FSCloseResult}_x(y, path, z), y \in \mathit{Files}, path \in \mathit{Paths}, z \in \{DONE, FAIL\}$
- $\mathsf{FSDeleteResult}_x(y, path, z), y \in \mathit{Files}, path \in \mathit{Paths}, z \in \{DONE, FAIL\}$
- $\mathsf{Schedule}(x, 0),\ x \in \mathit{Procs}$
- $\mathsf{Interrupt}(x),\ x \in \mathit{Procs}$

State

Open_x, a set of files currently opened by process x. Initially *Empty*.

DirectoryStruct, a tree of files, representing the directory structure of the system. All leaves of the tree are files whose paths are specified by the path from the root to the leaf in the tree. Each name is unique from its siblings. Initially *Empty*.

FSCommands, $\{SCHEDULE, OPEN, CREATE, DELETE, CLOSE, READAT, WRITEAT, KILLPROC, INTERRUPT\}$

Jobs a queue of triples (x, y, z), $x \in \mathit{FSCommands}, y \in \mathit{Procs}, z \in \mathit{Files}$. Initially *Empty*.

Responses a queue of sextuples (u, v, w, x, y, z), $u \in \mathit{FSCommands}, v \in \mathit{Procs}, w \in \mathit{Files}, x \in \mathbf{N}, y \in \mathbf{N}, z \in \mathit{ByteArrays}$. Initially *Empty*.

```
Input FSOpen_x(y, path)
    Eff: Jobs.Enqueue(SCHEDULE, x)
         Jobs.Enqueue(OPEN, x, y, path)

Input FSCreate_x(y, path, n)
    Eff: Jobs.Enqueue(SCHEDULE, x)
         Jobs.Enqueue(CREATE, x, y, path, n)

Input FSDelete_x(y, path)
    Eff: Jobs.Enqueue(SCHEDULE, x)
         Jobs.Enqueue(DELETE, x, y, path)

Input FSReadAt_x(y, path, n, m)
    Eff: Jobs.Enqueue
            (SCHEDULE, x)
         Jobs.Enqueue
            (READAT, x, y, path, n, m)

Input FSClose_x(y, path)
    Eff: Jobs.Enqueue
            (SCHEDULE, x)
         Jobs.Enqueue
            (CLOSE, x, y, path)
```

Figure 21: File System Manager (Kernel Level implementation)

File System Manager

Actions: (cont.)

```
Input FSWriteAt_x(y, path, n, m, A)
    Eff: Jobs.Enqueue
             (SCHEDULE, x)
         Jobs.Enqueue
             (WRITEAT, x, y, path, n, m, A)

Input KilledProc(x)
    Eff: Open_x := ∅

Internal Open
    Pre: Jobs.Head = (OPEN, x, y, path)
    Eff: if DirectoryStruct.Find(y, path) ∧
         ∀x, y ∩ Open_x = ∅
             Responses.Enqueue
                 (OPEN, x, y, path, DONE)
             Open_x := Open_x ∪ {(y, path)}
         else
             Responses.Enqueue
                 (OPEN, x, y, path, FAIL)
         Responses.Enqueue(INTERRUPT, x)
         Jobs.Dequeue

Internal Create
    Pre: Jobs.Head = (CREATE, x, y, path, n)
    Eff: if DirectoryStruct.Find(y, path)
             Responses.Enqueue
                 (CREATE, x, y, path, FAIL)
         else
             Responses.Enqueue
                 (CREATE, x, y, path, DONE)
             DirectoryStruct.Add(y, path, n)
         Responses.Enqueue(INTERRUPT, x)
         Jobs.Dequeue

Internal Delete
    Pre: Jobs.Head(DELETE, x, y)
    Eff: if DirectoryStruct.Find(y, path) ∧ ∀x, y ∉ Open_x
             DirectoryStruct.Delete(y, path)
             Responses.Enqueue
                 (DELETE, x, y, path, DONE)
         else
             Responses.Enqueue
                 (DELETE, x, y, path, FAIL)
         Responses.Enqueue(INTERRUPT, x)
         Jobs.Dequeue

Internal ReadAt
    Pre: Jobs.Head = (READAT, x, y, path, n, m)
    Eff: if (y, path) ∈ Open_x
             A := DirectoryStruct.Read(y, path, n, m)
         else
             A := FAIL
         Responses.Enqueue
             (READAT, x, y, path, A)
         Responses.Enqueue(INTERRUPT, x)
         Jobs.Dequeue

Internal Close
    Pre: Jobs.Head = (CLOSE, x, y, path)
    Eff: if (y, path) ∈ Open_x
             Open_x := Open_x − {(y, path)}
             Responses.Enqueue
                 (CLOSE, x, y, path, DONE)
         else
             Responses.Enqueue
                 (CLOSE, x, y, path, FAIL)
         Responses.Enqueue(INTERRUPT, x)
         Jobs.Dequeue

Internal WriteAt
    Pre: Jobs.Head = (WRITEAT, x, y, path, n, m, A)
    Eff: if y ∈ Open_x
             result := DirectoryStruct.Write
                 (y, path, n, m, A)
         else
             result := FAIL
         Responses.Enqueue
             (WRITEAT, x, y, path, result)
         Responses.Enqueue(INTERRUPT, x)
         Jobs.Dequeue

Output FSOpenResult_x(y, path, z)
    Pre: Responses.Head = (OPEN, x, y, path, z)
    Eff: Responses.Dequeue

Output FSCreateResult_x(y, path, z)
    Pre: Responses.Head = (CREATE, x, y, path, z)
    Eff: Responses.Dequeue

Output FSCloseResult_x(y, path, z)
    Pre: Responses.Head = (CLOSE, x, y, path, z)
    Eff: Responses.Dequeue
```

Figure 22: File System (Kernel Level Specification) Cont.

File System Manager

Actions: (cont.)

Output $\mathsf{FSDeleteResult}_x(y, path, z)$
Pre: $Responses.Head =$
$(DELETE, x, y, path, z)$
Eff: $Responses.Dequeue$

Output $\mathsf{FSReadAtResult}_x(y, path, A)$
Pre: $Responses.Head =$
$(READAT, x, y, path, A)$
Eff: $Responses.Dequeue$

Output $\mathsf{FSWriteAtResult}_x(y, path, z)$
Pre: $Responses.Head =$
$(WRITEAT, x, y, path, z)$
Eff: $Responses.Dequeue$

Output $\mathsf{Schedule}(x, b)$
Pre: $Jobs.Head = (SCHEDULE, b)$
Eff: $Jobs.Dequeue$

Output $\mathsf{Interrupt}(x)$
Pre: $Responses.Head = (INTERRUPT, x)$
Eff: $Responses.Dequeue$

Figure 23: File System (Kernel Level Specification) Cont.

As one may already have noticed, the Process signature detailed at the beginning of this thesis does not include any input or output actions for interfacing with the Scheduler. Yet it is apparent that the Scheduler must have some sort of interaction with the Process in order to schedule and deschedule it. This theoretical construct works to resolve that paradox.

We design the Abstraction automaton to be an interface layer between the Kernel Level Process and the Operating System that abstracts away the details of the scheduler. Instead of building such intelligence into the Process, the Abstraction automaton provides, through a series of per process queues, an interface for the Kernel Level Process that is identical to the one for the User Level Process. In this way, the Kernel Level Process does not receive any of the Scheduler's $\mathsf{Run}_x(b)$ commands, and is enabled to act as if it has the operating system's attention at all times. The Abstraction automaton, on the other hand, only passes on operating system and process outputs to and from the single process which is currently running. It receives the $\mathsf{Run}_x(b)$ commands, and activates the queues of the process dependent upon the $\mathsf{Run}_x(b)$ commands it receives.

Aside from being a convenient abstraction for the purposes of maintaining the process interface across the User and Kernel Level models of the operating system, the existence of the Abstraction automaton is grounded in a real system perspective. The computer programmer, in writing process programs, is not burdened with the realities of load sharing that occurs on a real multi-process platform. Instead, he/she writes the program as if it was the only process running on the system,

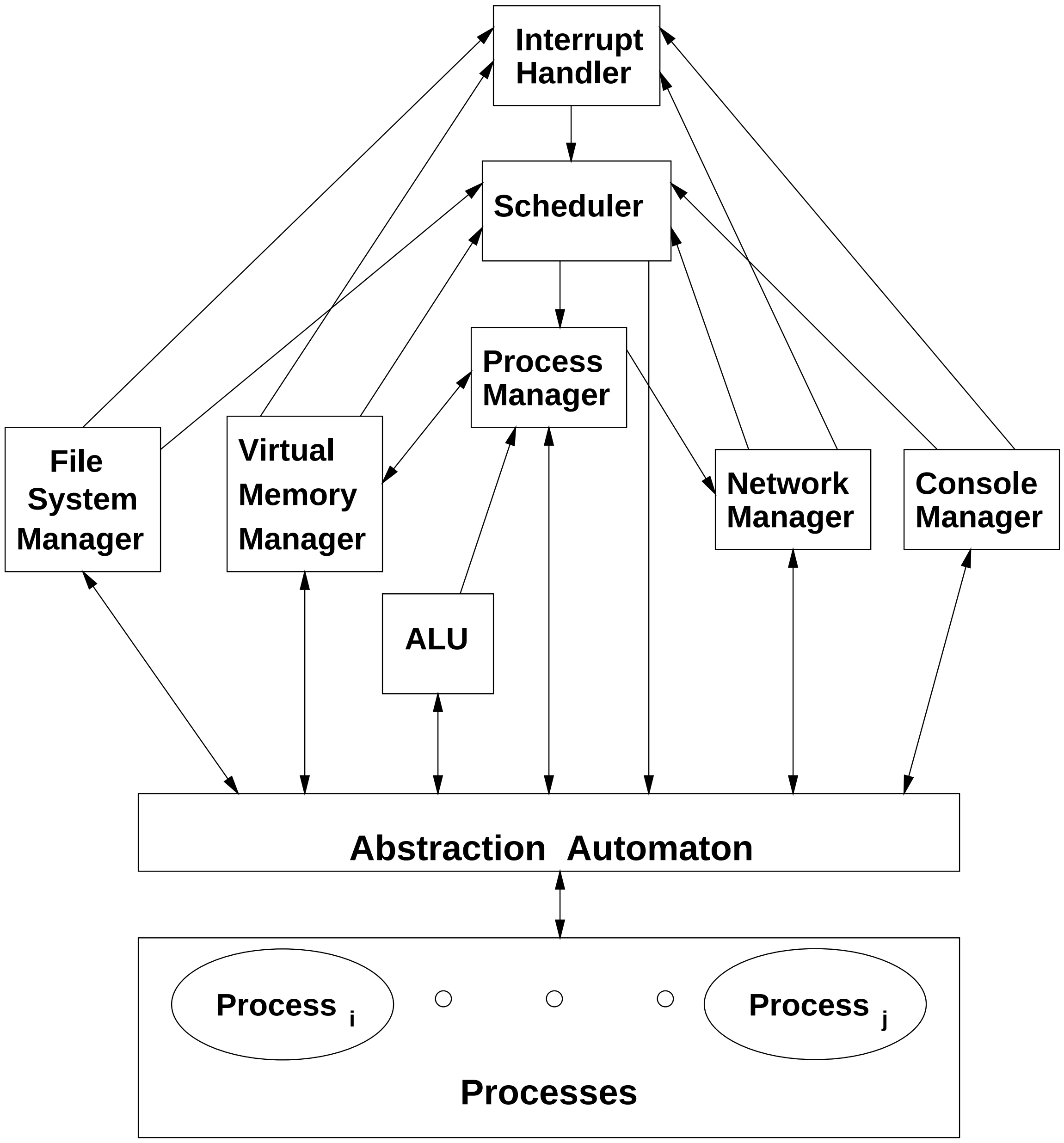

Figure 24: Kernel Level model with Abstraction Automaton

Abstraction Automata

Signature

Input:
  $\mathsf{ProcessAction}_x(i)$, $x \in \mathit{Procs}$, $i \in \mathit{ProcActions}$
  $\mathsf{OSAction}_x(i)$, $x \in \mathit{Procs}$, $i \in \mathit{OSActions}$
  $\mathsf{Run}_x(b)$
Output:
  $\mathsf{ProcessAction}_x(i)$, $x \in \mathit{Procs}$, $i \in \mathit{ProcActions}$
  $\mathsf{OSAction}_x(i)$, $x \in \mathit{Procs}$, $i \in \mathit{OSActions}$

State

$\mathit{ProcOutputs}_x$, the queue of outputs from process x heading towards the operating system. Initially *Empty*.

$\mathit{OSOutputs}_x$, the queue of outputs from the operating system heading towards process x. Initially *Empty*.

Running, the set of processes currently selected to run by the scheduler. Initially *Empty*.

Actions

```
Input ProcessAction_x(i)
    Eff: $\mathit{ProcOutputs}_x.\mathit{Enqueue}(i)$

Input OSAction_x(i)
    Eff: $\mathit{OSOutputs}_x.\mathit{Enqueue}(i)$

Input Run_x(b)
    Eff: if $b = 0 \wedge x \in \mathit{Running}$
             $\mathit{Running} := \mathit{Running} - x$
         else if $b = 1 \wedge x \notin \mathit{Running}$
             $\mathit{Running} := \mathit{Running} + x$

Output OSAction_x(i)
    Pre: $\mathit{Running} = x$
         $\mathit{OSOutputs}_x.\mathit{Head} = i$
    Eff: $\mathit{OSOutputs}_x.\mathit{Dequeue}$

Output ProcessAction_x(i)
    Pre: $\mathit{Running} = x$
         $\mathit{ProcOutputs}_x.\mathit{Head} = i$
    Eff: $\mathit{ProcOutputs}_x.\mathit{Dequeue}$
```

Figure 25: Abstraction Automaton

at a level of abstraction that removes the concept of resource sharing. This is exactly what the Abstraction automaton provides at the Kernel Level. The reason why this type of module is not necessary in a real operating system is because processes in a real operating system are not autonomous state machines running at their own schedule, but are in fact merely streams of code being run on a single machine. Because of this, they do not need to be signalled to be descheduled; instead the scheduler simply switches from running one process to running the next. In choosing to model the Processes as automata, we give them more independence, including control over when they can and cannot run. We therefore introduce the Abstraction automaton in order to remedy the problem.

In order to simplify the readability of the Abstraction automaton, we enumerate the output actions of the Kernel Level Process in the order listed in the Process definition in the User Level, representing the ith output action of Process x, generally as $\mathsf{ProcessAction}_x(i)$. Similarly, we do the same thing for the Kernel Level model of the operating system, ordering the modules alphabetically and then enumerating all of their actions together into one large index of actions. Call these operating system outputs $\mathsf{OSAction}(x, i)$ for the ith operating system output directed towards Process x. We leave out one operating system action from the index, $\mathsf{Run}_x(b)$, which we will deal with separately in the Abstraction Automaton.

With this simplification of the representation of the Abstraction Automaton, it becomes a queue storage house which funnels all outputs of process x to the operating system into the queue $ProcOutputs_x$, and all outputs of the operating system to process x into the queue $OSOutputs_x$. The variable *Running* is set to the current process i enabled by the scheduler, and the only outputs from the Abstraction automaton are from dequeuing the two queues $ProcOutputs_x$ and $OSOutputs_x$.

7 Correspondence Proofs

The power of the User-level definition of the Operating System is that it abstracts away the difficulties of an actual implementation, while still displaying the proper behavior of one. We can therefore use the User-level definition as a specification, and justify the Kernel-level implementation by showing that it implements the User-level specification. In order to show this correspondence between the two levels we use the mathematical tools developed for I/O automata, proving that the Kernel

Level model *simulates* the User Level. This is accomplished by eventually composing together all of the automata in the Kernel Level Model into one automaton, doing the same with the User Level Model, and then proving that the composition at the Kernel Level implements that at the User Level.

Before composing all of the automata at the Kernel Level we first use the concept of strong correspondence which we developed in Section 2, to prove several strong correspondences between the following pairs of Kernel Level and User Level models: Kernel and User Memory Manager, Kernel and User File System Manager, Kernel and User Console Manager, Kernel and User Network Manager, and Kernel and User ALU. We exclude the Process Manager from our proofs, simply because it is identical in all respects on both levels, and therefore trivially corresponds. Once we have proved these strong correspondences, we then compose together at each level all the modules for which we have proven strong correspondences. That is to say, we compose together the Kernel Level Memory, File System, Console, Network, ALU, and Process Managers, and then separately compose together their User Level counterparts.

Composing together at each level all automata for which we have proven strong correspondences implies that we have composed the entire User Level Model, completing the composition at that level. However, the composition of strongly corresponding modules leaves out the Scheduler, Interrupt Handler, and Abstraction automata from the Kernel Level model. Therefore, we must complete the composition of the Kernel Level model by composing the already composed strongly corresponding components with the remaining components. Because of several of the general theorems we proved earlier in this thesis, these steps follow rather directly.

Finally, after having composed all operating system automata in both the User and Kernel Level modules, we can prove a simulation relation between them, showing that the Kernel Level model does in fact implement the User Level specification.

7.1 Memory Manager Simulation

To show that the Virtual Memory Manager $\mathcal{K}$ at the Kernel-level meets the specification of the Memory Manager at the User-level $\mathcal{U}$, we establish a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

Theorem 5.1 The relation F in the figure 26 is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

F is a relation between states in $\mathcal{U}$ and states in $\mathcal{K}$ such that $u \in F[k]$ if and only if:

- $u.MemSpace_x[p] = k.Mem[k.PageTable_x[\lfloor p/PAGESIZE \rfloor].PhysPage * PAGESIZE + (p \bmod PAGESIZE)]$ if $k.PageTable_x[\lfloor /PAGESIZE \rfloor].Valid = TRUE$, otherwise $= SwapStore.Read(x, 1, p)$.
- $u.Used_x = k.Used_x$
- $u.SPACESIZE = k.SPACESIZE$
- $u.Freelist_x = k.Freelist_x$
- $u.Responses$ is a subsequence of $k.Responses$, where for each instance of $(SWAP, READ, x, p)$ or $(READ, x, p)$ in $k.Responses$, there is an instance of $(READ, x, p)$ in $u.Responses$. Similarly, each instance of $(SWAP, WRITE, x, p, v)$ or $(WRITE, x, p, v)$ in $k.Responses$ corresponds to an instance of $(WRITE, x, p, v)$ in $u.Responses$. Additionally, $(SCHEDULE, x)$ and $(INTERRUPT, x)$ in $k.Responses$ correspond to nothing in $u.Responses$. All other elements in $k.Responses$ are present in $u.Responses$.

Figure 26: Memory Manager strong correspondence from implementation to specification

The relationship between the User Level $u.MemSpace_x[p]$ and the Kernel Level $k.Mem$ and $k.SwapStore$ describes the way in which the Virtual Memory Manager at the User Level simulates a flat memory space for each of the processes. If memory location p for process x is currently loaded into memory in $k.Mem$, then the value found in $u.MemSpace_x[p]$, can be found be doing a virtual memory translation to find the location of the value in memory, and looking it up in $k.Mem$. If the memory location is not currently loaded into main memory then it is stored at location p in the process x's swap space.

The relationship between $u.Responses$ and $k.Responses$ is not conceptually complicated—$u.Responses$ is essentially a simplification of $k.Responses$. It is the same queue, except that $u.Responses$ contains no requests for scheduling or interrupts, and because there is no swapping in the user level, read and write requests do not go through a swap state.

Proof: In order to show that F is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$ we first show that for each start state of $\mathcal{K}$, there exists a corresponding start state of $\mathcal{U}$, and that this correspondence is preserved by each step of $\mathcal{K}$.

If k is a start state of $\mathcal{K}$, Mem and $SwapStore_x$ are all 0, $Used_x$, $Freelist_x$, and $Responses$ are empty, and $SPACESIZE$ is set to a predefined value. The start state of U corresponds to this state, since it too has $MemSpace_x$ set to all 0, $Used_x$, $Freelist_x$, and $Responses$ all empty, and $SPACESIZE$ set to the same predefined value.

Additionally, we show that $in(\mathcal{U}) \subseteq in(\mathcal{K}), int(\mathcal{U}) \subseteq int(\mathcal{K}), out(\mathcal{U}) \subseteq out(\mathcal{K})$. This is obvious from inspection of the automata.

To establish that the strong correspondence is preserved by every step of the implementation, suppose that k and u are reachable states of $\mathcal{K}$ and $\mathcal{U}$ respectively such that $u \in F[k]$ and that $k \xrightarrow{\pi} k'$. We show there exists a state $u' \in F[k']$ such that $u \xrightarrow{\pi} u'$ if $\pi \in acts(\mathcal{U})$, otherwise $u = u'$.

$\pi = \mathsf{Malloc}_x(n), \mathsf{Free}_x(p)$,these actions each correspond to their respective actions in the specification. Each action, in both the implementation and the specification, modifies $\mathit{Freelist}_x$ and $Used_x$ in the same fashion. The only difference between the actions in the specification and the implementation is the enqueuing of $(SCHEDULE, x)$ and $(INTERRUPT, x)$ in the Kernel level. This does not affect the relation, so all changes to pertinent state are identical.

$\pi = \mathsf{CreateProc}(x)$, $\mathsf{KilledProc}(x)$, these actions each correspond to their respective actions in the specification. All changes made to $\mathit{Freelist}_x$, Used_x, $k.\mathit{SwapStore}$, $k.\mathit{PageTable}_x$, and $\mathit{Responses}$ maintain the correspondence relation.

$\pi = \mathsf{Swap}$, this item has no corresponding action in the specification, and therefore corresponds to the empty transition. Although it makes many changes to $\mathit{PageTable}_x$, Mem, and $\mathit{SwapStore}$, one can see from inspection that no changes are of the fashion that change the relation F. Therefore, the change in state corresponds to no change in the state in the specification, and relation is preserved.

$\pi = \mathsf{MemRead}_x(p)$, $\mathsf{MemWrite}_x(p, v)$, these input actions each correspond to their respective input actions in the specification. The only differences between the Kernel level and User level descriptions is the presence of $SCHEDULE$ and $INTERRUPT$ commands in the Kernel level, which aren't modelled in the User level, and the option of enqueueing $(SWAP, \mathit{Command}, \dots)$ instead of $(\mathit{Command}, \dots)$ in $k.\mathit{Responses}$. This too corresponds to the User level specification, and therefore the strong correspondence is preserved.

$\pi = \mathsf{Schedule}(x, 0)$, $\mathsf{Interrupt}(x)$, these actions have no corresponding action in the specification, and corresponds to the empty transition. All changes to state in these functions are not modelled in the specification, as the only action is the addition of $SCHEDULE$ and $INTERRUPT$ commands in the $k.\mathit{Responses}$ queue. These changes do not change the relation in the strong correspondence function, and the relation is therefore preserved.

$\pi = \mathsf{MallocResult}_x(p)$, $\mathsf{CreateProcResult}(x, y)$, $\mathsf{KilledProcResult}(x, y)$, $\mathsf{KillProc}(x)$, these actions correspond to the same action in the specification, and are enabled in the same state—the existence of the corresponding command in the *Responses* queue. All state changes are exactly the same, consisting of dequeueing the action from *Responses*, and thereby preserve the strong correspondence.

$\pi = \mathsf{MemReadResult}_x(v)$, $\mathsf{MemWriteDone}_x$, these actions correspond to the same action in the specification. Changes made to $k.Mem$ reflect the changes made to $u.MemSpace_x$, thereby preserving the state correspondence. Each are enabled by the existence of $READ$ and $WRITE$ respectively in the $u.Responses$ and $k.Responses$ queue, which is preserved by F.

7.2 File System Manager Simulation

To show that the File System Manager $\mathcal{K}$ at the Kernel-level meets the specification of the File System Manager at the User-level $\mathcal{U}$, we establish a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

Theorem 5.2 The relation F in figure 27 is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

F is a relation between states in $\mathcal{U}$ and states in $\mathcal{K}$ such that $u \in F[k]$ if and only if:

- $u.Open_x = k.Open_x$
- $u.DirectoryStruct = k.DirectoryStruct$
- $u.Jobs$ is a subsequence of $k.Jobs$, where every command in $k.Jobs$ is in $u.Jobs$ with the exception of commands of the form $(SCHEDULE, x)$.
- $u.Responses$ is a subsequence of $k.Responses$, where every command in $k.Responses$ is in $u.Responses$ with the exception of commands of the form $(INTERRUPT, x)$.

Figure 27: File System Manager strong correspondence from implementation to specification

This mapping demonstrates the great similarity between the User Level and Kernel Level File System Managers, as the only serious difference comes in the absence of requests to the scheduler and interrupt handler in the User Level Model.

Proof: In order to show that F is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$ we first show that for each start state of $\mathcal{K}$, there exists a corresponding start state of $\mathcal{U}$, and that this correspondence is preserved by each step of $\mathcal{K}$.

If k is a start state of $\mathcal{K}$, $Open_x$, *DirectoryStruct*, *Jobs*, and *Responses* are *Empty*. The start state of U corresponds to this state, since it too has $Open_x$, *DirectoryStruct*, *Jobs*, and *Responses* *Empty*.

Additionally, we show that $in(\mathcal{U}) \subseteq in(\mathcal{K}), int(\mathcal{U}) \subseteq int(\mathcal{K}), out(\mathcal{U}) \subseteq out(\mathcal{K})$. This is obvious from inspection of the automata.

To establish that the strong correspondence is preserved by every step of the implementation, suppose that k and u are reachable states of $\mathcal{K}$ and $\mathcal{U}$ respectively such that $u \in F[k]$ and that $k \xrightarrow{\pi} k'$. We show that there exists a state $u' \in F[k']$ such that if $\pi \in acts(\mathcal{U})$, $u \xrightarrow{\pi} u'$, and otherwise $u = u'$.

1. $\pi = \mathsf{FSOpen}_x(y)$, $\mathsf{FSCreate}_x(y,n)$, $\mathsf{FSReadAt}_x(y,n,m)$, $\mathsf{FSWriteAt}_x(y,n,m,A)$, $\mathsf{FSClose}_x(y)$, $\mathsf{FSDelete}_x(y)$, these actions in the implementation correspond to their respective actions in the specification. For each action the triple (π, x, y) is enqueued in the *Jobs* queue. In the implementation, the pair $(SCHEDULE, x)$ is additionally enqueued, but that action is not represented in the specification. The state function is preserved, so the strong correspondence is as well.

   $\pi =$ Open, Create, ReadAt, WriteAt, Close, Delete, these actions in the implementation correspond to their respective actions in the specification. Each is enabled by the presence of (π, x, y) at the head of the $k.$*Jobs* queue. The equality of the *Jobs* queue is maintained through the correspondence function, therefore we know that these actions are enabled at the User level. In each case the triple (π, x, y) is dequeued from the *Jobs* queue, while the triple (π, x, y) is enqueued in the *Responses* queue. Because equality of the *DirectoryStruct* structure is maintained through the state correspondence function, we know that all operations on files will return the same value in both models. Additionally, the $Open_x$ structure is maintained identically, preserving the state correspondence in both automata. In the implementation, the pair $(INTERRUPT, x)$ is additionally enqueued in *Responses*, but that action is not represented in the specification. The state function is preserved, so the strong correspondence is as well.

   $\pi = \mathsf{FSOpenResult}_x(y,z)$, $\mathsf{FSCreateResult}_x(y,z)$, $\mathsf{FSReadAtResult}_x(y,A)$, $\mathsf{FSWriteAtResult}_x(y,z)$, $\mathsf{FSCloseResult}_x(y,z)$, $\mathsf{FSDeleteResult}_x(y,z)$ these actions correspond to their respective actions in the specification. The state change is identical in the implementation and the specification.

   $\pi = \mathsf{Schedule}(x, 0)$, $\mathsf{Interrupt}(x)$ these actions in the implementation have no corresponding action in the specification, and therefore correspond to the empty transition. The state function is preserved, as the deletions to *k.Jobs* and *k.Responses* are not of the form that requires any related deletion to *u.Jobs* or *u.Responses*. So the state function still maps the new implementation state to the same state in the specification, and the transition is the empty transition.

7.3 Console Manager Simulation

To show that the Console Manager at the Kernel-level meets the specification of the Console Manager at the User-level, we establish a strong correspondence from $\mathcal{K}$=Kernel-level implementation to $\mathcal{U}$=User-level specification.

Theorem 5.3 The relation F in figure 28 is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

F is a relation between states in $\mathcal{U}$ and states in $\mathcal{K}$ such that $u \in F[k]$ if and only if:

- $u.Buffer = k.Buffer$
- $u.Busy = k.Busy$
- *u.Responses* is a subsequence of *k.Responses*, where each element in *k.Responses* is in *u.Responses* if and only if it is of the form $(WRITERESULT, y, z)$ or $(READRESULT, y, z)$.
- *u.Jobs* is a subsequence of *k.Jobs*, where each element in *k.Jobs* is in *u.Jobs* if and only if it is of the form $(WRITESCREEN, y)$ or $(READ, x)$.

Figure 28: Console Manager strong correspondence from implementation to specification

Much like the File System Manager, the Kernel Level Console Manager differs from its User Level counterpart only in the addition of requests to the Scheduler and Interrupt Handler for scheduling requests. This difference manifests itself in nearly every action, with the enqueueing of $SCHEDULE$ or $INTERRUPT$ commands in the *k.Responses* and *k.Jobs* queues.

Proof: In order to show that F is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$ we first show that for each start state of $\mathcal{K}$, there exists a corresponding start state of $\mathcal{U}$, and that this correspondence is preserved by each step of $\mathcal{K}$.

If k is a start state of $\mathcal{K}$, the *Buffer*, *Jobs* and *Responses* sets are empty, and *Busy* is FALSE. The start state of U corresponds to this state, since it has *Buffer*, *Jobs* and *Responses* empty, and $Busy$ = FALSE.

Additionally, we show that $in(\mathcal{U}) \subseteq in(\mathcal{K}), int(\mathcal{U}) \subseteq int(\mathcal{K}), out(\mathcal{U}) \subseteq out(\mathcal{K})$. This is obvious from inspection of the automata.

To establish that the strong correspondence is preserved by every step of the implementation, suppose that k and u are reachable states of $\mathcal{K}$ and $\mathcal{U}$ respectively such that $u \in F[k]$ and that $k \xrightarrow{\pi} k'$. We show that there exists a state $u' \in F[k']$ such that if $\pi \in acts(\mathcal{U})$, $u \xrightarrow{\pi} u'$, and otherwise $u = u'$.

1. $\pi =$ $\mathsf{ConsoleRead}_x$, this action in the implementation corresponds to the same action in the specification. In both cases the pair $(READ, x)$ is enqueued in the *Jobs* queue. In the implementation, the pair $(SCHEDULE, x)$ is additionally enqueued, but that action is not represented in the specification. The state function is preserved, as is the simulation.

   $\pi =$ $\mathsf{ConsoleWrite}_x, \mathsf{WriteToScreen}$, these actions simulate their corresponding actions in the specification. The state change in the implementation matches the state change in the specification. *u.Responses* and *k.Responses* are changed in the exact same fashion in both, as is the state variable *Busy* in $\mathsf{ConsoleWrite}_x$. *u.Jobs* maintains the property that it is a subsequence of *k.Jobs*, where every instance of $(WRITESCREEN, y)$ in *k.Jobs* is present in *u.Jobs*. Similarly, *u.Responses* maintains the property that it is a subsequence of *k.Responses* where every instance of $(WRITERESULT, y, z)$ and $(READRESULT, y, z)$ present in *k.Responses* is present in *u.Responses*.

   $\pi =$ $\mathsf{LoadBuffer}$, this simulates the same action in the specification. The state change is identical in the implementation and the specification.

   $\pi =$ $\mathsf{ConsoleReadResult}_x(y), \mathsf{ConsoleWriteResult}_x(y)$, these actions simulate their corresponding actions in the specification. The state change is identical in the implementation and the specification.

   $\pi =$ $\mathsf{Schedule}(x, 0)$, $\mathsf{Interrupt}(x)$, these actions in the implementation simulate their corresponding actions in the specification, and therefore corresponds to the empty transition. All state changes in the implementation are such that they do not affect the strong correspondence relation.

7.4 Network Manager Simulation

To show that the Network Manager at the Kernel-level meets the specification of the Network Manager at the User-level, we establish a strong correspondence from $\mathcal{K}$=Kernel-level implementation to $\mathcal{U}$=User-level specification.

Theorem 5.4 The relation F in figure 29 is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

F is a relation between states in $\mathcal{U}$ and states in $\mathcal{K}$ such that $u \in F[k]$ if and only if:

- $u.Open_x = k.Open_x$
- $u.Network = k.Network$
- $u.InBuffer_{x,y} = k.InBuffer_{x,y}$
- $u.OutBuffer_{x,y} = k.OutBuffer_{x,y}$
- $u.Jobs$ is a subsequence of $k.Jobs$, where every command in $k.Jobs$ is in $u.Jobs$ with the exception of commands of the form $(SCHEDULE, x)$.
- $u.Responses$ is a subsequence of $k.Responses$, where every command in $k.Responses$ is in $u.Responses$ with the exception of commands of the form $(INTERRUPT, x)$.

Figure 29: Network Manager strong correspondence from implementation to specification

Like the Console Manager and File System Manager before, the File System Manager's correspondence is one simply of scheduling commands in the Kernel Level Model and not in the User Level Model.

Proof: In order to show that F is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$ we first show that for each start state of $\mathcal{K}$, there exists a corresponding start state of $\mathcal{U}$, and that this correspondence is preserved by each step of $\mathcal{K}$.

Additionally, we show that $in(\mathcal{U}) \subseteq in(\mathcal{K}), int(\mathcal{U}) \subseteq int(\mathcal{K}), out(\mathcal{U}) \subseteq out(\mathcal{K})$. This is obvious from inspection of the automata.

If k is a start state of $\mathcal{K}$, the $Open_x$, $Network$, $InBuffer_{x,y}$, $OutBuffer_{x,y}$, $Responses$ and $Jobs$ sets are empty. The start state of U corresponds to this state, since it has the $Open_x$, $Network$, $InBuffer_{x,y}$, $OutBuffer_{x,y}$ and $Jobs$ sets empty as well.

To establish that the strong correspondence is preserved by every step of the implementation, suppose that k and u are reachable states of $\mathcal{K}$ and $\mathcal{U}$ respectively such that $u \in F[k]$ and that $k \xrightarrow{\pi} k'$. We show that there exists a state $u' \in F[k']$ such that if $\pi \in acts(\mathcal{U})$, $u \xrightarrow{\pi} u'$, and otherwise $u = u'$.

$\pi = \mathsf{NetOpen}_x(y)$, $\mathsf{NetRead}_x(y, n)$, $\mathsf{NetWrite}_x(y, n, A)$, $\mathsf{NetClose}_x(y)$ these action in the implementation correspond to their respective actions in the specification. In each case the triple (π, x, y) is enqueued in the $Jobs$ queue. In the implementation, the pair $(SCHEDULE, x)$

is additionally enqueued, but that action is not represented in the specification. The state function is preserved, so the strong correspondence is as well.

$\pi =$ Open,Read, Write, Close, these actions in the implementation correspond to their respective actions in the specification. In each case the triple (π, x, y) is dequeued from the *Jobs* queue, and is enqueued in the *Responses* queue. Because the $InBuffer_{x,y}$, $OutBuffer_{x,y}$ and *Network* structures are maintained through the state correspondence function, we know that all identical operations on them will have identical results. Additionally, the $Open_x$ structure is maintained identically, preserving that state correspondence in both automata. In the implementation, the pair $(INTERRUPT, x)$ is additionally enqueued in *Responses*, but that action is not represented in the specification. The state function is preserved, with no external image, so the strong correspondence is as well.

$\pi =$ KilledProc(x), this action in the implementation corresponds to the same action in the specification. Because $u.Network$ and $k.Network$ are maintained identically, y will either be in both or neither of them. Therefore, all changes made to $u.OutBuffer_{x,y}$ and $k.OutBuffer_{x,y}$ will be identical.

$\pi =$ NetOpenResult$_x(y, z)$, NetReadResult$_x(A, z)$, NetWriteResult$_x(y, z)$, NetCloseResult$_x(y, z)$, NetReadInBuffer$_{x,y}$, NetWriteOutBuffer$_{x,y}$, these actions in the implementation correspond to their respective actions in the specification. The state change is identical in the implementation and the specification.

$\pi =$ Schedule$(x, 0)$, Interrupt(x), these actions in the implementation have no corresponding action in the specification, and therefore correspond to the empty transition. The state function is preserved, as the deletions to $k.Jobs$ and $k.Responses$ are not of the form that require any related deletions to $u.Jobs$ or $u.Responses$. So the state function still maps the new implementation state to the same state in the specification, and the transition is the empty transition.

7.5 ALU Simulation

To show that the ALU at the Kernel-level meets the specification of the ALU at the User-level, we establish a strong correspondence from $\mathcal{K}$=Kernel-level implementation to $\mathcal{U}$=User-level specification.

Theorem 5.5 The relation F in figure 30 is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$.

F is a relation between states in $\mathcal{U}$ and states in $\mathcal{K}$ such that $u \in F[k]$ if and only if:

- $u.Responses = k.Responses$
- $u.OutstandingOps = k.OutstandingOps$
- $u.Registers[k.CurrentProc] = k.CurrentRegs$, for all other registers, $u.Registers = k.RegMem$.

Figure 30: ALU strong correspondence from implementation to specification

The main difference between the ALU in the User Level Model and the Kernel Level Model is the existence of a single current set of registers in the Kernel Level. This requires the Kernel Level Model to switch between registers before performing actions for different processes, an action that is automatic and immediate for the User Level Model. Because of this, the correspondence between the two levels demonstrates the relationship of the current registers in the Kernel Level with one set of registers at the User Level, and all other registers at both levels corresponding in a straight forward fashion.

Proof: In order to show that F is a strong correspondence from $\mathcal{K}$ to $\mathcal{U}$ we first show that for each start state of $\mathcal{K}$, there exists a corresponding start state of $\mathcal{U}$, and that this correspondence is preserved by each step of $\mathcal{K}$.

Additionally, we show that $in(\mathcal{U}) \subseteq in(\mathcal{K}), int(\mathcal{U}) \subseteq int(\mathcal{K}), out(\mathcal{U}) \subseteq out(\mathcal{K})$. This is obvious from inspection of the automata.

If k is a start state of $\mathcal{K}$, the *Responses* queue is *Empty* and the *OutstandingOps* set is $\emptyset$, while *Registers* is initially all 0. The start state of U corresponds to this state, since it has the *Responses* queue *Empty* and *OutstandingOps* set $\emptyset$ as well, with $CurrentProc = 0$, and $RegMem$ initially 0.

To establish that the strong correspondence is preserved by every step of the implementation, suppose that k and u are reachable states of $\mathcal{K}$ and $\mathcal{U}$ respectively such that $u \in F[k]$ and that $k \xrightarrow{\pi} k'$. We show that there exists a state $u' \in F[k']$ such that if $\pi \in acts(\mathcal{U})$, $u \xrightarrow{\pi} u'$, and otherwise $u = u'$.

$\pi = \mathsf{OperationReq}_x(op)$, this action in the implementation corresponds to its respective action in the specification. In both cases, the action is triggered by the same external input, and both modify the *OutstandingOps* and *Responses* state variables in the exact same way. The state relation function is preserved, consequently the strong correspondence is preserved as well.

$\pi = \mathsf{Operate}(x, op)$, this action in the implementation corresponds to its respective action in the specification. Both actions have the same precondition, $(x, op) \in \mathit{OutstandingOps}$. As the state relation function preserves equality between $u.\mathit{OutstandingOps}$ and $k.\mathit{OutstandingOps}$, $\mathsf{Operate}(x, op)$ must be enabled in the specification whenever it is executed in the implementation. All operations to $k.\mathit{CurrentProc}$, $k.\mathit{CurrentRegs}$, and $k.\mathit{RegMem}$ preserve the relation with $u.\mathit{Registers}$. The changes to $k.\mathit{Responses}$ and $k.\mathit{OutstandingOps}$ are identical to those made to $u.\mathit{Responses}$ and $u.\mathit{OutstandingOps}$. Therefore the state relation is preserved, as is the strong correspondence.

$\pi = \mathsf{OperationResult}_x(opVal)$, $KillProc(x)$, both of these actions correspond to their respective actions in the specification. Both are enabled by the same state in their respective *Responses* queues, which correspond by the state relation function. The action makes identical changes to the $u.\mathit{Responses}$ queue and the $k.\mathit{Responses}$ queue at either level. As the changes in the implementation are identical to those in the specification, the state relation function is preserved, and so is the strong correspondence.

7.6 From Correspondence to Simulation

Having established strong correspondences for the Memory, File System, Network, Console, Process, and ALU automata, the task remains of proving that the full Kernel Level operating system simulates the User Level specification. We prove this in several steps, following the path of reasoning outlined by the general theorems we established earlier in section 2.

First, we compose the automata in each level for which we have already established strong correspondences with their counterparts at the other level. That is to say, at the Kernel Level we compose into one larger automaton the ALU, Virtual Memory, File System, Network, Console, and Process Managers. Similarly we compose their analogs at the User Level.

Lemma 2: The composition of Kernel Level ALU, Virtual Memory, File System, Network, Console, and Process Managers strongly corresponds to the composition of User Level ALU, Virtual

Memory, File System, Network, Console, and Process Managers.

Proof: From Theorem 1 we know that strong correspondence holds under composition. We have shown in section 7 that each of the Kernel and User Level pairs of ALU, Memory, File System, Network, Console, and Process Managers strongly correspond. From the definition of composition in Section 2.2 it is obvious that these automata are compatible, meaning that we can compose them without any conflicts. The signatures of the composing automata S_i are such that $\forall i, j$, $int(S_i) \cap acts(S_j) = \emptyset$, and $out(S_i) \cap out(S_j) = \emptyset$. We can therefore conclude that the composition of these automata at the Kernel Level strongly corresponds to the composition at the User Level.

□

We first note that the strong correspondence proven in Lemma 2 includes all of the automaton in the User Level Model. Therefore, in order to prove that the Kernel Level Model strongly corresponds to the User Level Model, all that remains to be shown is that the inclusion of the remaining Kernel Level automata does not destroy the strong correspondence. These remaining automata are the Scheduler, Interrupt Handler, and Abstraction Automaton.

Lemma 3: The composition of Kernel Level Scheduler, Interrupt Handler, Abstraction Automaton, ALU, Virtual Memory, File System, Network, Console, and Process Managers strongly corresponds to the composition of User Level ALU, Virtual Memory, File System, Network, Console, and Process Managers.

Proof: The only difference between this correspondence and that from Lemma 2 is the addition of the Scheduler, Interrupt Handler, and Abstraction Automaton to the Kernel Level Composition. From Theorem 2 we know that, in the case of an already existing strong correspondence, composing additional compatible automata in the lower level does not affect the strong correspondence of the composition. By inspection it is obvious that the Scheudler, Interrupt Handler, and Abstraction Automaton are compatible with the other automata in the Kernel Level Model. We can therefore add them into the composition, and maintain the strong correspondence between the Kernel Level and User Level Models.

□

We have now established a strong correspondence between the Kernel Level operating system and the User Level operating system. The final step is the establishing of a simulation.

Theorem 6: The Kernel Level implementation simulates the User Level specification.

Proof: From Theorem 4 we know that if F is a strong correspondence from A to B, and $in(A) = in(B), int(A) \supseteq int(B), out(A) = out(B)$, then A simulates B. Considering the external signatures of the composed Kernel Level $\mathcal{K}$ and User Level $\mathcal{U}$ operating system automata, it becomes apparent that $in(\mathcal{U}) \subseteq in(\mathcal{K})$, and $out(\mathcal{U}) \subseteq out(\mathcal{K})$. All output actions in $\mathcal{K}$ which are not in $\mathcal{U}$ can be hidden using the hiding operator defined in Section 2.3 and proven to maintain the strong correspondence from Theorem 3. The only difficulty lies in eliminating any additional input actions from $\mathcal{K}$. However there are not additional input actions, as we defined the interface of the operating systems to be equal, and input actions are only lost, never created, in composition. So $in(A) = in(B)$, $out(A) = out(B)$, and therefore $\mathcal{K}$ simulates $\mathcal{U}$.

$\square$

8 Conclusions and Future Work

The I/O Automata models of a generic operating system described in this thesis make a first foray into the relatively uncharted territory of formal design of operating system topology. Using a state machine model, we create both an abstract specification and a conceptual implementation of an operating system, and prove that the implementation simulates the specification. Formal models for operating systems will help the field to provide a formal framework for current and future research, and to teach new students.

The formal structure provided by the automata model of the operating system creates a pedagogical tool for instructing students in operating system design. Few, if any, modern texts break down the operating system into its component parts as distinctly as the automata model does. This separation of modules is critical throughout the period of first understanding operating system mechanics. While explaining the operating system at a well abstracted level, the automata model and its formal method of reasoning guides students to focus on interface specification and modularity within the operating system, an invaluable lesson in good system design.

Perhaps more important than its pedagogical repercussions, a formal model of an operating system provides a clean theoretical framework upon which to build new operating system research.

Initially, it is easy to see how one could expand on the relatively simple implementation presented here, adding new components and expanding on the complexity of existing ones. One such example is the File System Manager, as one could easily write replacement modules that implemented a log-structured file system, the UNIX fast file system, or many others. The only requirement for the module is that it meet the abstract specification, allowing it to integrate immediately into the rest of the operating system.

Another interesting example for a module that would be valuable to explore in the framework of an operating system is a caching automaton. Such an automaton could be specified to have a simple interface between it and processes allowing requests and responses for memory values, and an interface to the Memory Manager allowing for memory reads and writes. One could explore various sorts of caching algorithms in the context of the larger operating system, as well as easily modify the cache from servicing the memory to servicing the File System or any other sort of buffered I/O device.

Similarly, work on modules such as schedulers could be easily contextualized within the framework of an operating system. As much research has gone into various scheduling algorithms for different applications, having the capability to easily incorporate new designs into the already existing operating system structure is desirable. In this way scheduling research could be done separately from other operating system research, with the knowledge that adherence to the automaton interface guaranteed the compatibility of any advanced scheduler designs. Scheduling research is completely performance based, and therefore the true power of an operating system framework for scheduling research could only be realized with the addition of some form of performance analysis. The general I/O automaton model can be used to analyze performance, but one must extend the underlying automaton model to that of timed I/O automata. This extension adds in time-passage actions, and therefore enables performance measurements. Such automata permit performance analysis, but it still remains to develop easily usable analysis techniques for this application area.

Future work would not be limited to adding new automata and increasing the complexity of existing ones; if one modified the operating system to handle multiple processors (not a difficult task) one could also explore many of the more esoteric problems involved in parallel processing. Such modifications would be trivial in some automata modules, such as the scheduler, where variables such as *Active* would simply change from one active process to a set of them. In other automata, such as the Memory Manager, more complicated issues would arise, all of which would need to be modelled in that automaton. Significant work has been done in the area of formal models for distributed problems such as memory consistency and inter-processor data routing. It would be fascinating to place these algorithms in the context of a parallel processor operating system model, to see what kind of conclusions can be drawn from the more complete view of the machine which the larger scoped model provides.

Finally, and perhaps most interestingly, it would be fascinating to look at different operating system paradigms, especially those which have come out in recent years. The current work done on the exokernel operating system at MIT, and the VINO project at Harvard both describe operating systems which could exhibit inherent topological and theoretical differences from the operating system presented in this thesis. For such operating systems a new abstract specification could be necessary, and it would be interesting to see how these new paradigms manifest themselves in the formalization.

References

[1] Y. Gurevich. Evolving algebras 1993: Lipari guide. *Specification and Validation Methods*, pages 9–36, 1995.

[2] J. Guttag. Dyadic specification and its impact on reliability. Computer Science TR CSRG-45, University of Toronto, Dec. 1974.

[3] J. Guttag and J. Horning. Larch: Languages and tools for formal specification. *ACM Transactions on Computer Systems*, 1993.

[4] B. Lampson. 6.826 principles of computer systems. Computer Science TR MIT/LCS/RSS 19, MIT LCS, 1992.

[5] N. Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[6] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, Sept. 1989.

[7] T. Narten. A roadmap through nachos. *Duke University*, 1995.

I/O Automaton Model of Generic Operating System Primitives

A Thesis presented

by

Daniel Yates

To

Computer Science

in partial fullfillment of the honors requirements

for the degree of

Bachelor of Arts

Harvard College

Cambridge, Massachusetts

April 5, 1999