TabulaROSA: Tabular Operating System Architecture for Massively Parallel Heterogeneous Compute Engines

Citation: Kepner, Jeremy, Ron Brightwell, Alan Edelman et al. in Proceedings of the 2018 IEEE High Performance Extreme Computing Conference (HPEC), 25-27 Sept. 2018, Waltham, MA, USA, © 2018 IEEE.

As Published: https://dx.doi.org/10.1109/HPEC.2018.8547577

Publisher: Institute of Electrical and Electronics Engineers (IEEE)

Persistent URL: https://hdl.handle.net/1721.1/126114

Version: Original manuscript: author's manuscript prior to formal peer review

TabulaROSA: Tabular Operating System Architecture for Massively Parallel Heterogeneous Compute Engines

Jeremy Kepner[1−4], Ron Brightwell[5], Alan Edelman[2,3], Vijay Gadepally[1,2,4], Hayden Jananthan[1,4,6], Michael Jones[1,4], Sam Madden[2], Peter Michaleas[1,4], Hamed Okhravi[4], Kevin Pedretti[5], Albert Reuther[1,4], Thomas Sterling[7], Mike Stonebraker[2]

[1]MIT Lincoln Laboratory Supercomputing Center, [2]MIT Computer Science & AI Laboratory,
[3]MIT Mathematics Department, [4]MIT Lincoln Laboratory Cyber Security Division,
[5]Sandia National Laboratories Center for Computational Research,
[6]Vanderbilt University Mathematics Department,
[7]Indiana University Center for Research in Extreme Scale Technologies

***Abstract*—The rise in computing hardware choices is driving a reevaluation of operating systems. The traditional role of an operating system controlling the execution of its own hardware is evolving toward a model whereby the controlling processor is distinct from the compute engines that are performing most of the computations. In this context, an operating system can be viewed as software that brokers and tracks the resources of the compute engines and is akin to a database management system. To explore the idea of using a database in an operating system role, this work defines key operating system functions in terms of rigorous mathematical semantics (associative array algebra) that are directly translatable into database operations. These operations possess a number of mathematical properties that are ideal for parallel operating systems by guaranteeing correctness over a wide range of parallel operations. The resulting operating system equations provide a mathematical specification for a Tabular Operating System Architecture (TabulaROSA) that can be implemented on any platform. Simulations of forking in TabularROSA are performed using an associative array implementation and compared to Linux on a 32,000+ core supercomputer. Using over 262,000 forkers managing over 68,000,000,000 processes, the simulations show that TabulaROSA has the potential to perform operating system functions on a massively parallel scale. The TabulaROSA simulations show 20x higher performance as compared to Linux while managing 2000x more processes in fully searchable tables.**

I. Introduction

Next generation computing hardware is increasingly purpose built for simulation [1], data analysis [2], and machine learning [3]. The rise in computing hardware choices: general purpose central processing units (CPUs), vector processors, graphics processing units (GPUs), tensor processing units (TPUs), field programmable gate arrays (FPGAs), optical computers, and quantum computers is driving a reevaluation of operating systems. Even within machine learning, there has been a trend toward developing more specialized processors for different stages and types of deep neural networks (e.g., training vs inference, dense vs sparse networks). Such hardware is often massively parallel, distributed, heterogeneous, and non-deterministic, and must satisfy a wide range of security requirements. Current mainstream operating systems (OS) can trace their lineages back 50 years to computers designed for basic office functions running on serial, local, homogeneous, deterministic hardware operating in benign environments (see Figure 1). Increasingly, these traditional operating systems are bystanders at best and impediments at worse to using purpose-built processors. This trend is illustrated by the current GPU programming model whereby the user engages with a conventional OS to acquire the privilege of accessing the GPU to then implement most OS functions (managing memory, processes, and IO) inside their own application code.

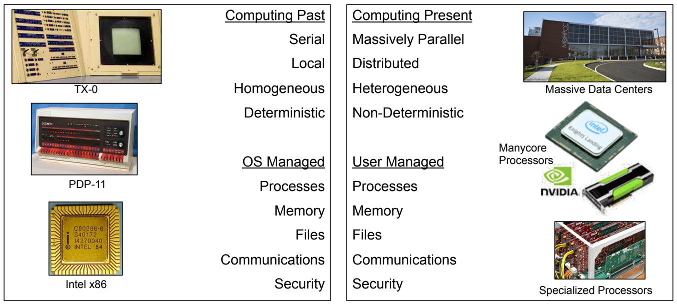

Fig. 1. Current operating systems can trace their lineage back to the first computers and still have many features that are designed for that era. Modern computers are very different and currently require the user to perform most operating system functions.

The role of an operating system controlling the execution of its own hardware is evolving toward a model whereby the controlling processor is distinct from the compute engines that are performing most of the computations [4]–[6]. Traditional operating systems like Linux are built to execute a shared kernel on homogeneous cores [7]. Popcorn Linux [8] and

This material is based upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

K2 [9] run multiple Linux instances on heterogeneous cores. Barrelfish [10] and FOS (Factored Operating System) [11] aim to support many heterogeneous cores over a distributed system. NIX [12], based on Plan 9 [13], relaxes the requirement on executing a kernel on every core by introducing application cores. Helios [14], a derivative from Singularity [15], reduces the requirements one step further by using software isolation instead of address space protection. Thereby, neither a memory management unit nor a privileged mode is required. In order to address the increasing role of accelerators, the M3 operating system goes further and removes all requirements on processor features [16].

In this context, an OS can be viewed as software that brokers and tracks the resources of compute engines. Traditional supercomputing schedules currently fill the role of managing heterogeneous resources but have inherent scalability limitations [17]. In many respects, this new operating system role is akin to the traditional role of a database management system (DBMS) and suggests that databases may be well suited to operating system tasks for future hardware architectures. To explore this hypothesis, this work defines key operating system functions in terms of rigorous mathematical semantics (associative array algebra) that are directly translatable into database operations. Because the mathematics of database table operations are based on a linear system over the union and intersection semiring, these operations possess a number of mathematical properties that are ideal for parallel operating systems by guaranteeing correctness over a wide range of parallel operations. The resulting operating system equations provide a mathematical specification for a Tabular Operating System Architecture (TabulaROSA) that can be implemented on any platform. Simulations of selected TabularROSA functions are performed with an associative array implementation on state-of-the-art, highly parallel processors. The measurements show that TabulaROSA has the potential to perform operating system functions on a massively parallel scale with 20x higher performance.

II. Standard OS and DBMS Operations

TabulaROSA seeks to explore the potential benefits of implementing OS functions in way that leverages the power and mathematical properties of database systems. This exploration begins with a brief description of standard OS and DBMS functions. Many concepts in modern operating systems can trace their roots to the very first time-sharing computers. Unix was first built in 1970 on the Digital Equipment Corporation (DEC) Programmed Data Processor (PDP) [18], [19], which was based on the MIT Lincoln Laboratory Transistorized eXperimental computer zero (TX-0) [20]–[23]. Modern operating systems, such as Linux, are vast, but their core concepts can be reduced to a manageable set of operations, such as those captured in the Xv6 operating system [24]. Xv6 is a teaching operating system developed in 2006 for MIT's operating systems course 6.828: Operating System Engineering. Xv6 draws inspiration from Unix V6 [25] and Lions' Commentary on UNIX, 6th Edition [26]. An elegant aspect of Unix is that its primary interface is the C language, which is also the primary language that programmers use to develop applications in Unix. Xv6 can be summarized by its C language kernel system calls that define its interface to the user programmer (see Figure 2). This subset of system calls is representative of the services that are core to many Unix based operating systems and serves as a point of departure for TabulaROSA. At a deeper level, many of the Xv6 operating system functions can be viewed as adding, updating, and removing records from a series of C data structures that are similar to purpose-built database tables.

fork()	Create process
exit()	Terminate current process
wait()	Wait for a child process to exit
kill(pid)	Terminate process pid
getpid()	Return current process's id
sleep(n)	Sleep for n seconds
exec(filename, *argv)	Load a file and execute it
sbrk(n)	Grow process's memory by n bytes
open(filename, flags)	Open a file; flags indicate read/write
read(fd, buf, n)	Read n byes from an open file into buf
write(fd, buf, n)	Write n bytes to an open file
close(fd)	Release open file fd
dup(fd)	Duplicate fd
pipe(p)	Create a pipe and return fd's in p
chdir(dirname)	Change the current directory
mkdir(dirname)	Create a new directory
mknod(name, major, minor)	Create a device file
fstat(fd)	Return info about an open file
link(f1, f2)	Create another name (f2) for the file f1
unlink(filename)	Remove a file

Fig. 2. (Adapted from [24]). Xv6 operating system kernel functions.

Modern database systems are designed to perform many of the same functions of purpose-built data analysis, machine learning, and simulation hardware. The key tasks of many modern data processing systems can be summarized as follows [27]

- Ingesting data from operational data systems
- Data cleaning
- Transformations
- Schema integration
- Entity consolidation
- Complex analytics
- Exporting unified data to downstream systems

To meet these requirements, database management systems perform many operating system functions (see Figure 3) [28].

An elegant aspect of many DBMS is that their primary data structures are tables that are also the primary data structures programmers use to develop applications on these systems. In many databases, these table operations can be mapped onto well-defined mathematical operations with known mathematical properties. For example, relational (or SQL) databases [29]–[31] are described by relational algebra [32]–[34] that corresponds to the union-intersection semiring $\cup.\cap$ [35]. Triple-store databases (NoSQL) [36]–[39] and analytic databases (NewSQL) [40]–[45] follow similar

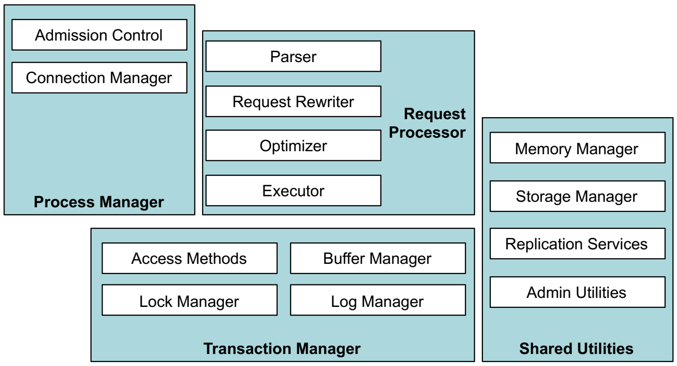

Fig. 3. (Adapted from [28]). Database management systems provide many of the functions found in an operating system. Nearly all functions are managed via manipulations of database tables.

mathematics [46]. The table operations of these databases are further encompassed by associative array algebra, which brings the beneficial properties of matrix mathematics and sparse linear systems theory, such as closure, commutativity, associativity, and distributivity [47].

The aforementioned mathematical properties provide strong correctness guarantees that are independent of scale and particularly helpful when trying to reason about massively parallel systems. Intersection $\cap$ distributing over union $\cup$ is essential to database query planning and parallel query execution over partioned/sharded database tables [48]–[54]. Similarly, matrix multiplication distributing over matrix addition ensures the correctness of massively parallel implementations on the world's largest supercomputers [55] and machine learning systems [56]–[58]. In software engineering, the scalable commutativity rule guarantees the existence of a conflict-free (parallel) implementation [59]–[61].

III. Associative Array Algebra

The full mathematics of associative arrays and the ways they encompass matrix mathematics and relational algebra are described in the aforementioned references [35], [46], [47]. Only the essential mathematical properties of associative arrays necessary for describing TabulaROSA are reviewed here. The essence of associative array algebra is three operations: element-wise addition (database table union), element-wise multiplication (database table intersection), and array multiplication (database table transformation). In brief, an associative array $\mathbf{A}$ is defined as a mapping from sets of keys to values

$$\mathbf{A} : K_1 \times K_2 \to \mathbb{V}$$

where K_1 are the row keys and K_2 are the column keys and can be any sortable set, such as integers, real numbers, and strings. The row keys are equivalent to the sequence ID in a relational database table or the process ID of file ID in an OS data structure. The column keys are equivalent to the column names in a database table and the field names in an OS data structure. $\mathbb{V}$ is a set of values that forms a semiring $(\mathbb{V}, \oplus, \otimes, 0, 1)$ with addition operation $\oplus$, multiplication operation $\otimes$, additive identity/multiplicative annihilator 0, and multiplicative identity 1. The values can take on many forms, such as numbers, strings, and sets. One of the most powerful features of associative arrays is that addition and multiplication can be a wide variety of operations. Some of the common combinations of addition and multiplication operations that have proven valuable are standard arithmetic addition and multiplication $+.\times$, the aforementioned union and intersection $\cup.\cap$, and various tropical algebras that are important in finance [62]–[64] and neural networks [65]: $\max.+$, $\min.+$, $\max.\times$, $\min.\times$, $\max.\min$, and $\min.\max$.

The construction of an associative array is denoted

$$\mathbf{A} = \mathbb{A}(\mathbf{k}_1, \mathbf{k}_2, \mathbf{v})$$

where $\mathbf{k}_1$, $\mathbf{k}_2$, and $\mathbf{v}$ are vectors of the row keys, column keys, and values of the nonzero elements of $\mathbf{A}$. When the values are 1 and there is only one nonzero entry per row or column, this associative array is denoted

$$\mathbb{I}(\mathbf{k}_1, \mathbf{k}_2) = \mathbb{A}(\mathbf{k}_1, \mathbf{k}_2, 1)$$

and when $\mathbb{I}(\mathbf{k}) = \mathbb{I}(\mathbf{k}, \mathbf{k})$, this is array is referred to as the identity.

Given associative arrays $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, element-wise addition is denoted

$$\mathbf{C} = \mathbf{A} \oplus \mathbf{B}$$

or more specifically

$$\mathbf{C}(k_1, k_2) = \mathbf{A}(k_1, k_2) \oplus \mathbf{B}(k_1, k_2)$$

where $k_1 \in K_1$ and $k_2 \in K_2$. Similarly, element-wise multiplication is denoted

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$$

or more specifically

$$\mathbf{C}(k_1, k_2) = \mathbf{A}(k_1, k_2) \otimes \mathbf{B}(k_1, k_2)$$

Array multiplication combines addition and multiplication and is written

$$\mathbf{C} = \mathbf{A}\mathbf{B} = \mathbf{A} {\oplus}.{\otimes} \mathbf{B}$$

or more specifically

$$\mathbf{C}(k_1, k_2) = \bigoplus_k \mathbf{A}(k_1, k) \otimes \mathbf{B}(k, k_2)$$

where k corresponds to the column key of $\mathbf{A}$ and the row key of $\mathbf{B}$. Finally, the array transpose is denoted

$$\mathbf{A}(k_2, k_1) = \mathbf{A}^{\mathsf{T}}(k_1, k_2)$$

The above operations have been found to enable a wide range of database algorithms and matrix mathematics while also preserving several valuable mathematical properties that ensure the correctness of parallel execution. These properties include commutativity

$$\begin{array}{rcl} \mathbf{A} \oplus \mathbf{B} & = & \mathbf{B} \oplus \mathbf{A} \\ \mathbf{A} \otimes \mathbf{B} & = & \mathbf{B} \otimes \mathbf{A} \\ (\mathbf{A}\mathbf{B})^{\mathsf{T}} & = & \mathbf{B}^{\mathsf{T}}\mathbf{A}^{\mathsf{T}} \end{array}$$

associativity

$$
\begin{aligned}
(\mathbf{A} \oplus \mathbf{B}) \oplus \mathbf{C} &= \mathbf{A} \oplus (\mathbf{B} \oplus \mathbf{C}) \\
(\mathbf{A} \otimes \mathbf{B}) \otimes \mathbf{C} &= \mathbf{A} \otimes (\mathbf{B} \otimes \mathbf{C}) \\
(\mathbf{A}\mathbf{B})\mathbf{C} &= \mathbf{A}(\mathbf{B}\mathbf{C})
\end{aligned}
$$

distributivity

$$
\begin{aligned}
\mathbf{A} \otimes (\mathbf{B} \oplus \mathbf{C}) &= (\mathbf{A} \otimes \mathbf{B}) \oplus (\mathbf{A} \otimes \mathbf{C}) \\
\mathbf{A}(\mathbf{B} \oplus \mathbf{C}) &= (\mathbf{A}\mathbf{B}) \oplus (\mathbf{A}\mathbf{C})
\end{aligned}
$$

and the additive and multiplicative identities

$$
\mathbf{A} \oplus \mathbb{0} = \mathbf{A} \qquad \mathbf{A} \otimes \mathbb{1} = \mathbf{A} \qquad \mathbf{A}\mathbb{I} = \mathbf{A}
$$

where $\mathbb{0}$ is an array of all 0, $\mathbb{1}$ is an array of all 1, and $\mathbb{I}$ is an array with 1 along its diagonal. Furthermore, these arrays possess a multiplicative annihilator

$$
\mathbf{A} \otimes \mathbb{0} = \mathbb{0} \qquad \mathbf{A}\mathbb{0} = \mathbb{0}
$$

Most significantly, the properties of associative arrays are determined by the properties of the value set $\mathbb{V}$. In other words, if $\mathbb{V}$ is linear (distributive), then so are the corresponding associative arrays.

IV. TabulaROSA Mathematics

There are many possible ways of describing the Xv6 OS functions in terms of associative arrays. One possible approach begins with the following definitions. $\mathbf{P}$ is the distributed global process associative array, where the rows are the process IDs and the columns are metadata describing each process. In Xv6, there are approximately ten metadata fields attributed to each process. Notional examples of dense and sparse schemas for $\mathbf{P}$ are shown in Figure 4. In this analysis, a hybrid schema is assumed as it naturally provides fast search on any row or column, enables most OS operations to be performed with array multiplication, and allows direct computation on numeric values. $\mathbf{p}$ is a vector containing one or more unique process IDs and is implicitly the output of the getpid() accessor function or all processes associated with the current context. Similarly, $\dot{\mathbf{p}}$ is implicitly the output of allocproc().

Associative array specifications of all the Xv6 functions listed in Figure 2 are provided in Appendix A. Perhaps the most important of these functions is fork(), which is used to create new processes and is described mathematically as follows

```
$\dot{\mathbf{p}}$ = fork()        # Function for creating processes
– – – – – – – – – – – – – – – – – – – – – – – – – – – – –
  $\dot{\mathbf{p}}$  = allocproc()          # Create new process IDs
  $\dot{\mathbf{P}}$  = $\mathbb{I}(\dot{\mathbf{p}},\mathbf{p})\mathbf{P}$          # Create new $\dot{\mathbf{P}}$ from $\mathbf{P}$
  $\dot{\mathbf{P}}$ $\oplus$= $\mathbb{I}(\dot{\mathbf{p}},\mathrm{parent}|\mathbf{p})$   # Add parent identifiers
  $\dot{\mathbf{P}}$ $\oplus$= $\mathbb{I}(\mathbf{p},\mathrm{child}|\dot{\mathbf{p}})$     # Add child identifiers
  $\mathbf{P}$ $\oplus$= $\dot{\mathbf{P}}$       # Add new processes to global table
```

where | implies concatenation with a separator such as |. The above mathematical description of fork() can also be algebraically compressed into the following single equation

$$
\mathbf{P} \oplus= \mathbb{I}(\dot{\mathbf{p}},\mathbf{p})\mathbf{P} \oplus \mathbb{I}(\dot{\mathbf{p}},\mathrm{parent}|\mathbf{p}) \oplus \mathbb{I}(\mathbf{p},\mathrm{child}|\dot{\mathbf{p}})
$$

Additionally, forking the same process into multiple new processes can be done by adding nonzero rows to $\mathbb{I}(\dot{\mathbf{p}})$. Likewise, combining existing processes into a single new process can be done by adding nonzero entries to any row of $\mathbb{I}(\dot{\mathbf{p}},\mathbf{p})$. Thus, the associative array representation of fork() can accommodate a range of fork() operations that normally require distinct implementations. The above equation is independent of scale and describes a method for simultaneously forking many processes at once. The above equation is linear, so many of the operations can be ordered in a variety of ways while preserving correctness. Because dominant computation is the array multiplication $\mathbb{I}(\dot{\mathbf{p}},\mathbf{p})\mathbf{P}$, the performance and scalability of forking in TabulaROSA can be estimated by simulating this operation.

	dir	file	parent	size	state
c321-p123	tmp/	a.txt	c321-p103	1K	sleeping
c321-p124	doc/	b.csv	c321-p104	10K	runnable
c321-p125	img/	c.tsv	c321-p105	100K	running
c321-p126	bin/	d.bin	c321-p106	1M	zombie

	dir|bin/	dir|doc/	dir|img/	dir|tmp/	file|a.txt	file|b.csv	file|c.tsv	file|d.bin	parent|c321-p103	parent|c321-p104	parent|c321-p105	parent|c321-p106	size|100K	size|10K	size|1K	size|1M	state|runnable	state|running	state|sleeping	state|zombie
c321-p123				1	1				1						1				1	
c321-p124		1				1				1				1			1			
c321-p125			1				1				1		1					1		
c321-p126	1							1				1				1				1

Fig. 4. Notional examples of dense (top) and sparse (bottom) schemas for the distributed global process associative array $\mathbf{P}$, where the rows are the process IDs and the columns are metadata describing each process.

V. Simulation Results

The associative array representation of fork() indicates that the core computational driver in a massively parallel implementation would be associative array multiplication $\mathbb{I}(\dot{\mathbf{p}},\mathbf{p})\mathbf{P}$. This operation can be readily simulated with the D4M (Dynamic Distributed Dimensional Data Model) implementation of associative arrays that are available in a variety of programming languages (d4m.mit.edu) [47], [66], [67]. In the simulation, each forker constructs a $2^{18} \times 2^{18}$ sparse associative array $\mathbf{P}$ with approximately 10 nonzero entries per row. The row keys of $\mathbf{P}$ are the globally unique process IDs $\mathbf{p}$. The column keys of $\mathbf{P}$ are strings that are surrogates for process ID metadata fields. The fork() operation is simulated by array multiplication of $\mathbf{P}$ by a $2^{18} \times 2^{18}$ permutation array $\mathbb{I}(\dot{\mathbf{p}},\mathbf{p})$. Each row and column in $\mathbb{I}(\dot{\mathbf{p}},\mathbf{p})$ has one randomly assigned nonzero entry that corresponds to the mapping of the current process IDs in $\mathbf{p}$ to the new process IDs in $\dot{\mathbf{p}}$. The full sparse representation is the most computationally challenging as it represents the case whereby all unique metadata have their own column and are directly searchable. For comparison, the Linux fork() command is also timed, and it is assumed that the maximum number of simultaneous processes on a compute node is the standard value of 2^{16}.

Both the D4M simulation and the Linux fork() comparison were run in parallel on a supercomputer consisting of 648 compute nodes, each with at least 64 Xeon processing cores, for a total of 41,472 processing cores. The results are shown in Figures 5 and 6. In both cases, a single forker was run on 1, 2, 4,. . . , and 512 compute nodes, followed by running 2, 4, . . . , and 512 forkers on each of the 512 compute nodes to achieve a maximum of 262,144 simultaneous forkers. This pleasingly parallel calculation would be expected to scale linearly on a supercomputer.

The number of processes managed in the D4M simulation grows linearly with the number of forkers, while in the Linux fork() comparison the number of processes managed grows linearly with the number of compute nodes. Figure 5 shows the total number of processes managed, which for D4M is 2^{18} times the number of forkers with a largest value of 2^{36} or over 68,000,000,000. For the Linux fork() comparison, the total number of processes managed is 2^{16} times the number of nodes with a largest value of 2^{25}. In both computations, the number of processes managed could be increased. For these calculations, typical values are used. Since associative array operations are readily translatable to databases using disk storage, the capacity of this approach is very large even when there are high update rates [68].

Figure 6 shows the rate at which processes forked in the D4M simulation grows linearly with the number of forkers and peaks at 64 forkers per node, which corresponds to the number of physical cores per node. In the Linux fork() comparison, the rate grows linearly with the number of forkers and peaks at 8 forkers per node. The largest fork rate for D4M in this simulation is approximately 800,000,000 forks per second. The largest fork rate for the Linux fork() comparison is approximately 40,000,000 forks per second.

The purpose of these comparisons is not to highlight any particular attribute of Linux, but merely to set the context for the D4M simulations, which demonstrate the size and speed potential of TabulaROSA.

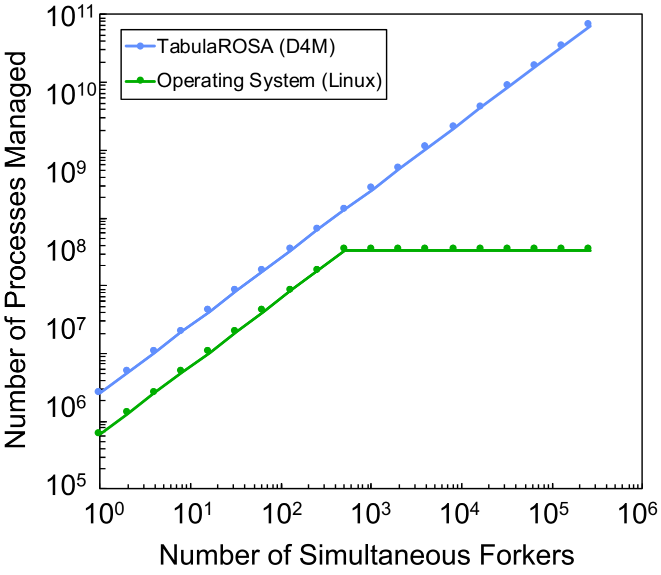

Fig. 5. Number of processes managed vs. total number of forkers for the TabulaROSA D4M simulation and the Linux operating system running on a 32,000+ core system.

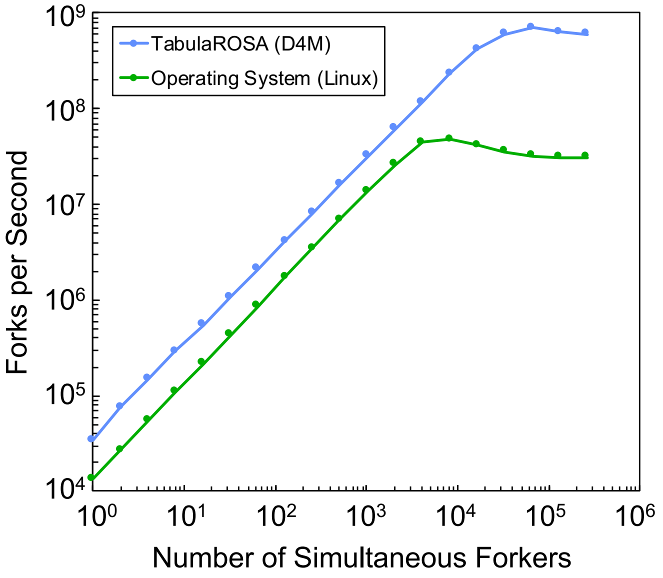

Fig. 6. Fork rate vs. total number of forkers for the TabulaROSA D4M simulation and the Linux operating system running on a 32,000+ core system.

VI. Conclusion

The rise in computing hardware choices: general purpose central processing units, vector processors, graphics processing units, tensor processing units, field programmable gate arrays, optical computers, and quantum computers are driving a reevaluation of operating systems. The traditional role of an operating system controlling the execution on its own hardware is evolving toward a model whereby the controlling processor is completely distinct from the compute engines that are performing most of the computations. In many respects, this new operating system role is akin to the traditional role of a database management system and suggests that databases may be well suited to operating system tasks for future hardware architectures. To explore this hypothesis, this work defines key operating system functions in terms of rigorous mathematical semantics (associative array algebra) that are directly translatable into database operations. Because the mathematics of database table operations are based on a linear system over the union and intersection semiring, these operations possess a number of mathematical properties that are ideal for parallel operating systems by guaranteeing correctness over a wide range of parallel operations. The resulting operating system equations provide a mathematical specification for a Tabular Operating System Architecture (TabulaROSA) that can be implemented on any platform. Simulations of forking in TabularROSA are performed by using an associative array implementation and are compared to Linux on a 32,000+ core supercomputer. Using over 262,000 forkers managing over 68,000,000,000 processes, the simulations show that TabulaROSA has the potential to perform operating system functions on a massively parallel scale. The TabulaROSA simulations show 20x higher performance compared to Linux, while managing 2000x more processes in fully searchable tables.

Acknowledgments

The authors wish to acknowledge the following individuals for their contributions and support: Bob Bond, Paul Burkhardt, Sterling Foster, Charles Leiserson, Dave Martinez, Steve Pritchard, Victor Roytburd, and Michael Wright.

References

[1] T. Sterling, M. Anderson, and M. Brodowicz, *High Performance Computing: Modern Systems and Practices*. Morgan Kaufmann, 2017.

[2] W. S. Song, V. Gleyzer, A. Lomakin, and J. Kepner, "Novel graph processor architecture, prototype system, and results," in *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2016.

[3] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[4] P. Beckman, R. Brightwell, B. de Supinski, M. Gokhale, S. Hofmeyr, S. Krishnamoorthy, M. Lang, B. Maccabe, J. Shalf, and M. Snir, "Exascale operating systems and runtime software report," *US Department of Energy, Technical Report, December*, 2012.

[5] M. Schwarzkopf, "Operating system support for warehouse-scale computing," *PhD. University of Cambridge*, 2015.

[6] P. Laplante and D. Milojicic, "Rethinking operating systems for rebooted computing," in *Rebooting Computing (ICRC), IEEE International Conference on*, pp. 1–8, IEEE, 2016.

[7] L. Torvalds, "Linux: a portable operating system," *Master's thesis, University of Helsinki, dept. of Computing Science*, 1997.

[8] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: bridging the programmability gap in heterogeneous-isa platforms," in *Proceedings of the Tenth European Conference on Computer Systems*, p. 29, ACM, 2015.

[9] F. X. Lin, Z. Wang, and L. Zhong, "K2: a mobile operating system for heterogeneous coherence domains," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 285–300, 2014.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 29–44, ACM, 2009.

[11] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 76–85, 2009.

[12] F. J. Ballesteros, N. Evans, C. Forsyth, G. Guardiola, J. McKie, R. Minnich, and E. Soriano-Salvador, "Nix: A case for a manycore system for cloud computing," *Bell Labs Technical Journal*, vol. 17, no. 2, pp. 41–54, 2012.

[13] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom, "Plan 9 from bell labs," *Computing systems*, vol. 8, no. 2, pp. 221–254, 1995.

[14] E. B. Nightingale, O. Hodson, R. McIlroy, C. Hawblitzel, and G. Hunt, "Helios: heterogeneous multiprocessing with satellite kernels," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 221–234, ACM, 2009.

[15] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi, "Language support for fast and reliable message-based communication in singularity os," in *ACM SIGOPS Operating Systems Review*, vol. 40, pp. 177–190, ACM, 2006.

[16] N. Asmussen, M. Völp, B. Nöthen, H. Härtig, and G. Fettweis, "M3: A hardware/operating-system co-design to tame heterogeneous many-cores," in *ACM SIGPLAN Notices*, vol. 51, pp. 189–203, ACM, 2016.

[17] A. Reuther, C. Byun, W. Arcand, D. Bestor, B. Bergeron, M. Hubbell, M. Jones, P. Michaleas, A. Prout, A. Rosa, and J. Kepner, "Scalable system scheduling for hpc and big data," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 76–92, 2018.

[18] J. B. Dennis, "A multiuser computation facility for education and research," *Communications of the ACM*, vol. 7, no. 9, pp. 521–529, 1964.

[19] G. Bell, R. Cady, H. McFARLAND, B. Delagi, J. O'Laughlin, R. Noonan, and W. Wulf, "A new architecture for mini-computers: The dec pdp-11," in *Proceedings of the May 5-7, 1970, spring joint computer conference*, pp. 657–675, ACM, 1970.

[20] J. Mitchell and K. Olsen, "Tx-0, a transistor computer with a 256 by 256 memory," in *Papers and discussions presented at the December 10-12, 1956, eastern joint computer conference: New developments in computers*, pp. 93–101, ACM, 1956.

[21] W. A. Clark, "The lincoln tx-2 computer development," in *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, pp. 143–145, ACM, 1957.

[22] J. McCarthy, S. Boilen, E. Fredkin, and J. Licklider, "A time-sharing debugging system for a small computer," in *Proceedings of the May 21-23, 1963, spring joint computer conference*, pp. 51–57, ACM, 1963.

[23] T. Myer and I. E. Sutherland, "On the design of display processors," *Communications of the ACM*, vol. 11, no. 6, pp. 410–414, 1968.

[24] R. Cox, M. F. Kaashoek, and R. Morris, "Xv6, a simple unix-like teaching operating system," 2011.

[25] O. Ritchie and K. Thompson, "The unix time-sharing system," *The Bell System Technical Journal*, vol. 57, no. 6, pp. 1905–1929, 1978.

[26] J. Lions, *Lions' Commentary on UNIX 6th Edition*. Peer to Peer Communications, ISBN 1-57398-013-7, 2000.

[27] M. Stonebraker, "The seven tenets of scalable data unification," 2017.

[28] J. M. Hellerstein and M. Stonebraker, *Readings in database systems*. MIT Press, 2005.

[29] M. Stonebraker, G. Held, E. Wong, and P. Kreps, "The design and implementation of ingres," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 3, pp. 189–222, 1976.

[30] C. J. Date and H. Darwen, *A guide to the SQL Standard: a user's guide to the standard relational language SQL*. Addison-Wesley, 1989.

[31] R. Elmasri and S. Navathe, *Fundamentals of database systems*. Addison-Wesley Publishing Company, 2010.

[32] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

[33] D. Maier, *The theory of relational databases*, vol. 11. Computer science press Rockville, 1983.

[34] S. Abiteboul, R. Hull, and V. Vianu, eds., *Foundations of Databases: The Logical Level*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1st ed., 1995.

[35] H. Jananthan, Z. Zhou, V. Gadepally, D. Hutchison, S. Kim, and J. Kepner, "Polystore mathematics of relational algebra," in *Big Data Workshop on Methods to Manage Heterogeneous Big Data and Polystore Databases*, IEEE, 2017.

[36] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[37] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[38] L. George, *HBase: the definitive guide: random access to your planet-size data*. " O'Reilly Media, Inc.", 2011.

[39] A. Cordova, B. Rinaldi, and M. Wall, *Accumulo: Application Development, Table Design, and Best Practices*. " O'Reilly Media, Inc.", 2015.

[40] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik, "C-Store: A column-oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases*, pp. 553–564, VLDB Endowment, 2005.

[41] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1496–1499, 2008.

[42] M. Balazinska, J. Becla, D. Heath, D. Maier, M. Stonebraker, and S. Zdonik, "A demonstration of scidb: A science-oriented dbms," *Cell*, vol. 1, no. a2, 2009.

[43] M. Stonebraker and A. Weisberg, "The voltdb main memory dbms.," *IEEE Data Eng. Bull.*, vol. 36, no. 2, pp. 21–27, 2013.

[44] D. Hutchison, J. Kepner, V. Gadepally, and A. Fuchs, "Graphulo implementation of server-side sparse matrix multiply in the accumulo database," in *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2015.

[45] V. Gadepally, J. Bolewski, D. Hook, D. Hutchison, B. Miller, and J. Kepner, "Graphulo: Linear algebra graph kernels for nosql databases," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*, pp. 822–830, IEEE, 2015.

[46] J. Kepner, V. Gadepally, D. Hutchison, H. Jananthan, T. Mattson, S. Samsi, and A. Reuther, "Associative array model of sql, nosql, and newsql databases," in *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2016.

[47] J. Kepner and H. Jananthan, *Mathematics of Big Data*. MIT Press, 2018.

[48] G. M. Booth, "Distributed information systems," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pp. 789–794, ACM, 1976.

[49] D. E. Shaw, "A relational database machine architecture," in *ACM SIGIR Forum*, vol. 15, pp. 84–95, ACM, 1980.

[50] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[51] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE micro*, vol. 23, no. 2, pp. 22–28, 2003.

[52] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, 2010.

[53] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 61–72, ACM, 2012.

[54] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, *et al.*, "Spanner: Googles globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[55] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, pp. 803–820, 2003.

[56] M. F. Møller, "A scaled conjugate gradient algorithm for fast supervised learning," *Neural networks*, vol. 6, no. 4, pp. 525–533, 1993.

[57] P. J. Werbos, *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*, vol. 1. John Wiley & Sons, 1994.

[58] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[59] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 4, p. 10, 2015.

[60] A. T. Clements, M. F. Kaashoek, E. Kohler, R. T. Morris, and N. Zeldovich, "The scalable commutativity rule: designing scalable software for multicore processors," *Communications of the ACM*, vol. 60, no. 8, pp. 83–90, 2017.

[61] S. S. Bhat, *Designing multicore scalable filesystems with durability and crash consistency*. PhD thesis, Massachusetts Institute of Technology, 2017.

[62] P. Klemperer, "The product-mix auction: A new auction design for differentiated goods," *Journal of the European Economic Association*, vol. 8, no. 2-3, pp. 526–536, 2010.

[63] E. Baldwin and P. Klemperer, "Understanding preferences:'demand types', and the existence of equilibrium with indivisibilities," *SSRN*, 2016.

[64] B. A. Mason, "Tropical algebra, graph theory, & foreign exchange arbitrage," 2016.

[65] J. Kepner, M. Kumar, J. Moreira, P. Pattnaik, M. Serrano, and H. Tufo, "Enabling massive deep neural networks with the GraphBLAS," in *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2017.

[66] J. Kepner, W. Arcand, W. Bergeron, N. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic Distributed Dimensional Data Model (D4M) database and computation system," in *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 5349–5352, IEEE, 2012.

[67] A. Chen, A. Edelman, J. Kepner, V. Gadepally, and D. Hutchison, "Julia implementation of the dynamic distributed dimensional data model," in *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2016.

[68] J. Kepner, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Hubbell, P. Michaleas, J. Mullen, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Achieving 100,000,000 database inserts per second using accumulo and d4m," in *High Performance Extreme Computing Conference (HPEC)*, IEEE, 2014.

Appendix A: TabulaROSA Specification

$\mathbf{P}$ is the distributed global process associative array, where the rows are the process IDs and the columns are metadata describing each process. In Xv6, approximately 10 metadata fields are attributed to each process. Notional examples of dense and sparse schemas for $\mathbf{P}$ are shown in Figure 4. In this analysis, a hybrid schema is assumed as it naturally provides fast search on any row or column and allows most OS operations to be performed with array multiplication while still being able to perform direct computation numeric values. $\mathbf{p}$ is a vector containing one or more unique process IDs and is implicitly the output of the getpid() accessor function or all processes associated with the current context. Similarly, $\dot{\mathbf{p}}$ is implicitly the output of allocproc(). $\mathbf{F}$ is the distributed global files associative array where the rows are the file IDs and the columns are metadata describing each file and arguments for corresponding file operations. $\mathbf{f}$ is a vector containing one or more unique file IDs, and $\mathbf{A_f}$ are the associative arrays corresponding to the contents in file identifiers $\mathbf{f}$.

```
exec($\dot{\mathbf{F}}$)            # Load files and execute them
- - - - - - - - - - - - - - - - - - - - - - - - - -
   $\dot{\mathbf{f}} = \text{open}(\dot{\mathbf{F}})$                # Open files
   $\mathbf{P} \oplus= \mathbb{I}(\mathbf{p}, \dot{\mathbf{f}})\dot{\mathbf{F}}$             # Replace current instructions
```

```
$\mathbf{A}_{\text{buf}}$ = read($\mathbf{A_f}$, row, col)   # Read selected data
- - - - - - - - - - - - - - - - - - - - - - - - - -
   $\dot{\mathbf{f}} = \text{open}(\dot{\mathbf{F}})$                 # Open files
   $\mathbf{A}_{\text{buf}} = \mathbb{I}(\text{row})\mathbf{A_f}\mathbb{I}(\text{col})$   # Select and copy to buffer
```

```
$\mathbf{A_f}$ = write($\mathbf{A}_{\text{buf}}$, row, col)   # Write selected data
- - - - - - - - - - - - - - - - - - - - - - - - - -
   $\dot{\mathbf{f}} = \text{open}(\dot{\mathbf{F}})$                 # Open files
   $\mathbf{A_f} = \mathbb{I}(\text{row})\mathbf{A}_{\text{buf}}\mathbb{I}(\text{col})$   # Select and copy to file
```

```
$\dot{\mathbf{p}}$ = fork()         # Function for creating processes
- - - - - - - - - - - - - - - - - - - - - - - - - -
   $\dot{\mathbf{p}}$   = allocproc()          # Create new process IDs
```

```
  $\dot{\mathbf{P}} \;=\; \mathbb{I}(\dot{\mathbf{p}},\mathbf{p})\mathbf{P}$          # Create new $\dot{\mathbf{P}}$ from $\mathbf{P}$
  $\dot{\mathbf{P}} \oplus= \mathbb{I}(\dot{\mathbf{p}},\mathrm{parent}|\mathbf{p})$     # Add parent identifiers
  $\dot{\mathbf{P}} \oplus= \mathbb{I}(\mathbf{p},\mathrm{child}|\dot{\mathbf{p}})$      # Add child identifiers
  $\mathbf{P} \oplus= \dot{\mathbf{P}}$       # Add new processes to global table
```

```
exit()                # Exit current processes
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus=$ -$(\mathbb{I}(\mathbf{p})\mathbf{P})$          # Remove exiting processes
```

```
wait()                # Wait for child processes to exit
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\dot{\mathbf{P}} = \mathbb{I}(\mathbf{p})\ \mathbf{P}\ \mathbb{I}(\mathrm{child}|*)$    # Get child processes
  while($\mathbf{P}$)
    $\dot{\mathbf{P}} = \mathbb{I}(\mathbf{p})\ \mathbf{P}\ \mathbb{I}(\mathrm{child}|*)$    # Get exiting processes
```

```
kill($\mathbf{p}$)               # Terminate processes
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus=$ -$(\mathbb{I}(\mathbf{p})\mathbf{P})$        # Remove processes to kill
```

```
$\mathbf{p}$ = getpid()        # Return current process IDs
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{p} = \mathrm{row}(\mathbf{P}\ \mathbb{I}(\mathrm{current}|*))$   # Get process IDs
```

```
$\dot{\mathbf{p}}$ = allocproc()     # Return vector of new process IDs
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\dot{\mathbf{p}}$ = rand(), hash(), $\ldots$  # Create new process IDs
```

```
sleep($n$)             # Sleep for $n$ seconds
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus= \mathbb{A}(\mathbf{p},\mathrm{sleep},n)$   # Add $n$ seconds of sleep
```

```
sbrk($n$)              # Grow memory by $n$ bytes
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus= \mathbb{A}(\mathbf{p},\mathrm{memory},n)$   # Add $n$ bytes of memory
```

```
chdir(dir)            # Change current directories
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus= \mathbb{A}(\mathbf{p},\mathrm{cwd},\mathrm{dir})$       # Add new dir
```

```
$\dot{\mathbf{f}}$ = open($\dot{\mathbf{F}}$)       # Open files $\dot{\mathbf{F}}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus= \dot{\mathbf{F}} \oplus \mathbb{A}(\dot{\mathbf{f}},\mathrm{open},1)$   # Mark as open
```

```
close($\dot{\mathbf{f}}$)            # Close file IDs $\dot{\mathbf{f}}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{P} \oplus=$ -$\mathbb{A}(\dot{\mathbf{f}},\mathrm{open},1)$       # Remove open flags
```

```
$\dot{\mathbf{F}}$ = fstat($\dot{\mathbf{f}}$)       # Get metadata on $\dot{\mathbf{f}}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\dot{\mathbf{F}} = \mathbb{I}(\dot{\mathbf{f}})\mathbf{F}$             # Get corresponding rows
```

```
mkdir($\dot{\mathbf{F}}$)          # Make diretories $\dot{\mathbf{F}}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  close(open($\dot{\mathbf{F}}$))          # Open and close to create
```

```
$\dot{\mathbf{F}}$ = dup($\dot{\mathbf{f}}$)         # Duplicate file descriptors
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\dot{\mathbf{f}}\ \ = \ldots$           # Create new file IDs
  $\mathbf{F} \oplus= \mathbb{I}(\dot{\mathbf{f}},\mathbf{f})\mathbf{F}$   # Copy file metadata to new IDs
```

```
link($\dot{\mathbf{F}}$)           # Create links to files
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{F} \oplus= \dot{\mathbf{F}}$          # Copy file metadata to new files
```

```
unlink($\dot{\mathbf{F}}$)        # Unlink files
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  $\mathbf{F} \oplus=$ -$\dot{\mathbf{F}}$         # Remove file metadata from files
```

```
mknod($\dot{\mathbf{F}}$)          # Make nodes to devices $\dot{\mathbf{F}}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  close(open($\dot{\mathbf{F}}$))          # Open and close to create
```

```
pipe($\dot{\mathbf{F}}$)           # Make pipe $\dot{\mathbf{F}}$
- - - - - - - - - - - - - - - - - - - - - - - - - - - -
  close(open($\dot{\mathbf{F}}$))          # Open and close to create
```