EFFECTIVENESS OF OPERATING SYSTEM PROTOTYPING FROM A TEMPLATE: APPLICATION TO MINIX

Myla Archer, James Bock, Deborah Frincke, and Karl Levitt

Division of Computer Science, University of California, Davis

1. Introduction.

In a previous paper [AFL90], we presented a tool for the rapid prototyping of operating systems based on an executable template operating system specification. By extending the template specification, a designer can generate a rapid prototype of a specific operating system. The template specification consists of sorts (object classes) arranged in a hierarchy of three kinds, where the design decisions made in the template for each kind are in some sense more general than those made for its successor kind. We used the FASE (Final Algebra Specification and Execution) system language [KJA83] as the language for the specification.

It is our belief that the SRM template is general enough to be used as a basis for rapid prototyping nearly any non-distributed operating system at the system call level. Naturally, it is important to test this belief, and to show that the speed of prototype development and the execution speed of a rapid prototype is great enough for the development of a prototype to be useful for testing the properties of a design. Our recent work is aimed at answering these questions, demonstrating the usefulness of having a rapid prototype of an operating system, and evaluating the template methodology in general.

Specifically, we report on the following:

- Our experience in using the template specification to produce a rapid prototype of a real operating system — MINIX [Tanen87].
- Our approach towards enhancing the MINIX rapid prototype (and hence the MINIX design) to produce a multilevel secure MINIX rapid prototype. Other enhancements are under consideration.
- A first step towards a comparison of our FASE template specification with such a template written in other executable specification languages. Currently, we are studying the OBJ and EPROS (the executable subset of VDM) languages.

2. Motivation.

As with other large systems susceptible to errors in the early phases of design, it is useful to evaluate important aspects of an operating system design through rapid prototyping. This clearly holds for systems under development; we also believe it can be true for existing systems that are likely to undergo modification.

Rapid prototyping has not previously been applied to operating systems, despite what seems to us clear reasons to do so. Although there are surely many examples of implementation errors that have plagued operating systems, poor design decisions exist.

Particularly important to evaluate for an operating system is the usefulness of its interface (the system calls) for utilities, such as the shell. The implementation of an actual operating system is complex; however, through the judicious application of abstraction, a rapid prototype of an operating system as a specification that captures the functional behavior of the system calls is not difficult to produce. As we

0-8186-3040-X/92 $3.00 © 1992 IEEE

discuss below, the rapid prototype of an operating system can be significantly smaller and simpler than the final implementation. There is a need for an incremental development methodology, through which those parts of the operating system that bear most on performance can be carried through implementation and evaluation, deferring the implementation of the less performance-critical parts.

The similarity among operating system functional behaviors has led us [AFL90] to the template operating system specification which we call the SRM (Secure Resource Manager). The template specification defines a collection of abstract resources that are accessed by requests from processes. A scheduler coordinates the requests. An abstract interpreter translates the requests by users for operating system services, possibly replacing a given request by finer-grained requests. A request can be denied if it would be in violation of the security policy. The completion of the template entails providing instances for the abstract resources and a more detailed description of the effect of operations.

Our goals in carrying out the template completion for a real operating system are to evaluate the methodology with respect to its appropriateness and ease of use, the feasibility of using it on non-toy systems, and the usefulness of the resultant prototype as an interface for programming and evaluating operating system utilities.

There are other reasons why one might wish to prototype a real operating system. One can experiment with adjustments and enhancements to the existing system more easily in the prototype than in the system itself. A properly done rapid prototype makes it relatively easy to separate concerns regarding properties of the system prototyped, thus facilitating experimentation with enhancements with regard to some particular concern, such as security. In addition, debugging a prototype can be much simpler than debugging its corresponding implementation: in particular, one can more easily set up problem situations in the prototype that can be hard to reproduce in the implementation.

As indicated in the introduction, our template is written in the language of the FASE executable specification system. While we have found the FASE system to be satisfactory for our purposes, we are undertaking a comparison of FASE with other executable specification systems both as a basis for rapid prototyping of large software systems and with regard to feasibility and usefulness of template specifications.

3. A brief review of the SRM template and methodology.

We begin our review of the template and its methodology by briefly recalling some details concerning FASE. FASE specifications are operational in style, and thus executable. At the same time, they free a specifier (or prototyper) from concern for the details of data representation, greatly simplifying the details of operator definitions. FASE also supports incremental implementation: a specification can at any time be replaced by a concrete implementation, while preserving the executability of the prototype. Since FASE is integrated with lisp, the development process in the FASE system can be easily enhanced by writing appropriate lisp programs to exercise specifications.

Central to understanding a FASE specification are the concepts of *distinguishing set* of operations and *tuple* representation of elements of the sort being defined (the TOI, or type of interest) in the specification. The elements of the tuple are in one-to-one correspondence with the distinguishing set operations, which serve to distinguish one TOI element from another. Holding a particular TOI element fixed while a distinguishing set operation is applied to it (and possible other arguments) allows one to determine the constant (or function) that is the corresponding tuple entry for that element. The cross product of the sorts of these constants (or arities of these functions) then can be taken as the domain of abstract representations of the TOI elements, or for short, the representation of the TOI. Explicit tuple representations of elements of a given sort can only be used in the specification where that sort is the TOI.

The SRM template sorts and their representations are given in Figure 1.

Figure 1 also indicates the "kind" of each sort in the template. These are defined as follows: Sorts of kind 1 are completely defined, in the sense that the only operators that may involve tuples in their definitions are those already defined in the template (these operations are known as the "constructors"). Additional, derived operators may be added for convenience. Sorts of kind 2 have a fixed distinguishing set of operations, and hence a fixed tuple representation in every elaboration of the template. Sorts of kind 3 have some fixed part, perhaps only their name, but usually, fixed operator declarations (usually, not fixed operator *definitions*). This hierarchy of kinds has implications both for the kinds of changes one makes in them when specifying a system starting from the template, and for the nature of the general properties one can prove about these data types from the template specification alone.

The hierarchy is also the guide to the most efficient way to instantiate the template for a given system design: namely, first define the elements of the kind 2 specifications needed for a specific application (in particular, the SRMops, the Interp, and the SecPol), and use the needs of these elements to determine the representations of the elements in the kind 3 specifications (such as Objects, Processes, and Args). We refer to this procedure as the template methodology.

More details concerning the template, its methodology, and the FASE system can be found in [AFL90] and [KJA83].

SORT	KIND	REPRESENTATION
SRM	1	SRMopSet × State × Scheduler × Interp × SecPol
SRMopSet	1	SRMop → Bool
State	1	ObjectSet × ProcessSet × RequestList × History
Scheduler	2	(State → Request) × (State → RequestList)
Interp	2	Request → Request
SecPol	2	(State × Request → Bool) × (State × Request → Request)
SRMop	2	Symbol × (State × ArgList → State)
ObjectSet	1	Object → Bool
ObjectPred	2	Object → Bool
Object	3	ObjectId × ···
ProcessSet	1	Process → Bool
ProcessPred	2	Process → Bool
Process	3	ProcessId × ···
RequestList	1	Request × RequestList
Request	1	SRMop × ArgList × ProcessId
History	3	···
ArgList	1	Arg × ArgList
Arg	3	ObjectId × ProcessId × ··· [really ObjectId + ProcessId + ···]
ObjectId	3	···
ProcessId	3	···

Figure 1. Template sorts and their representations.

4. Results of the MINIX prototype exercise.

As indicated in the introduction, we have been developing a prototype of MINIX [Tanen87], an operating system with essentially the same interface (system calls) as UNIX System V. In this section, we discuss what we have learned in the process about the usefulness of our SRM template and its associated methodology, and about the benefits of rapid prototyping an operating system. The MINIX prototype specification itself can be found in [ABFL91].

While having a rapid prototype of MINIX yields the benefits with respect to testing, debugging, and improving the MINIX design that we have mentioned above, the principal benefit we have obtained so far from developing a MINIX prototype from the FASE template has been to subject the methodology of [AFL90] to a more rigorous testing. In particular, we hoped to determine whether our template was in fact an appropriate foundation for a real operating system prototype, whether starting from a template did in fact significantly reduce a prototyper's effort, and whether the resulting prototype was in fact efficient enough for testing purposes. Our results so far confirm that the answer to all these questions is yes.

We have found the template methodology we outlined in [AFL90] and reviewed in the previous section to be appropriate for constructing the MINIX prototype. We began with expanding the specifications of kind 2. For the initial subset of MINIX that we prototyped, the Interp (instruction interpreter) and SecPol (access policy) components of the MINIX SRM could be the trivial ones. While it was clear from the informal description of MINIX in [Tanen87] that there should be (at least) two kinds of objects: files and directories, most of the information needed to determine the appropriate representations of (kind 3) Objects, Processes, and Args came from the definitions of the operators of sort SRMop (a kind 2 data type). We added a small number of kind 4 (special purpose) specifications to handle the details of these representations: Objectkind (to indicate the kind of an Object), SymbolList (for the content of a file Object and the buffer content of a Process), Pathname (to support ObjectIds and represent the content of a link Object), PathnameList (for the content of a directory Object), and Cursortab (to keep track of file cursor positions for a Process). The only changes we made to kind 1 (fixed) specifications were the (permissible) addition of derived operators for convenience: for example, we added an operation *initMINIX* to SRM.

To sum up, for the MINIX example, we have found our template and its associated methodology to be both sufficient and appropriate. We are also further convinced that it reduces the effort needed to produce an operating system prototype. While the effort involved in constructing the MINIX prototype was not trivial, we had only to make the decisions we have outlined above. We estimate that the time taken to elaborate the kind 3 specifications (in particular, Object and Process) is about one man-week each, while each SRMop took about half a day per operation.

The number of lines of code for some of our SRMops (e.g., LSEEK) is not much less than the number of lines of C code for the actual MINIX version; in other cases, because we have abstracted away more implementation details (as in the case of READ and WRITE), our code is significantly shorter. For all the operations, we have found that our code for system operations is generally much easier both to write and understand, since one can forget low-level details of data representation. As an example, we give the definitions related to READ in Appendix A, which also lists the operations we have so far defined. The definitions related to the READ system call in [Tanen87], by contrast, take a few pages, and are considerably more opaque.

Our template also has proved, if used correctly, to be an appropriate framework on which to do incremental development of a prototype. Incrementing our prototype to include more features of MINIX required only localized changes. For example, the initial subset of MINIX prototyped only the file

system, and assumes a single process, the system Process. The addition of user processes required adding some new SRMops and replacing the trivial Interp component by a more complex one (which still has the same behavior with respect to the system Process). For example, the original READ SRMop, which can only be executed by the system Process, must be supplemented by an operation uREAD, which can be executed by a user Process; an appropriate Interp is defined which interprets a uREAD request by a user Process as a READ request by the system Process that has the user ProcessId as an extra argument. When access permissions are modelled, the trivial SecPol will similarly have to be replaced by a more complex one.

It must be noted, however, that to ensure that only localized changes are necessary, one must take the complete top-level system description into account — what all the operations and their arguments will be. Otherwise, additional components may be needed in the tuple representations of kind 3 data types when a new SRMop is added. This would require modification to all definitions involving tuples in those kind 3 data types. This inconvenience could in some cases be mitigated by an appropriate interface that locates tuples in a specification and prompts a specifier for the entries for missing components.

With regard to the efficiency of the MINIX prototype we have built from our template, we have not done any explicit timings. However, we produced a simple driver that updates and displays the MINIX state. We have observed a delay of perhaps one or two seconds in producing the display of updated states when all data type specifications are compiled (as opposed to interpreted), and have found our prototype to be a quite reasonable interactive testing tool.

The FASE system provides for mixing of (executably) specified data types with implemented data types. This feature makes it possible to further speed up a rapid prototype, once certain implementation decisions have been made.

To illustrate the convenience with which we can exercise and visualize our prototype by taking advantage of user-defined grammars and the ability to exercise FASE specifications using lisp programs, we show an abbreviated session using our driver in Appendix B.

In constructing the MINIX prototype, we consistently attempted to derive the MINIX design in as abstract a form as possible. We took our design information from the (relatively) informal descriptions and our understanding of the code in [Tanen87], and when doubt still existed, from testing. Thus we cannot guarantee that our prototype specification behaves identically to the corresponding subset of MINIX. Since a major part of our goal was to test the template methodology on a reasonably nontrivial system, this is not of too great concern. However, we believe the exercise of attempting to define as abstractly as possible the design of an operating system such as MINIX can itself be valuable. For example, determining the correct abstract definition of an ObjectId — when linked directories are allowed, it appears to be a reduced finite automaton on the alphabet of all Symbols, that represents all possible Pathnames for an Object — gives certain insights into the expected observable behavior related to links. Studying the observable changes when the last link to an Object is removed leads to the question of what happens to a Process that had that Object open for reading or writing at the time: this is not immediately clear from the code in [Tanen87]. In our prototype, the question arises naturally, because the entries in the Cursortab of the Process connected with the former ObjectId must be updated. The answer to this question has certain implications for the ultimate implementation of memory manangement. If it is answered wrongly, there can be problems with enforcing the usual file protection policy. Thus, attempting to define the system abstractly can also lead one to considering aspects of the design that may have been overlooked in the implementation.

5. Retrofitting multilevel security to the MINIX rapid prototype.

As pointed out in section 2, one of the uses of a rapid prototype of an existing system is to provide a basis for testing experimental modifications or enhancements to the system. One of our particular interests is retrofitting data security to existing operating systems. We note that the SecPol component of our template provides the natural place to do this. We have developed what we believe is a general design for supporting multilevel securty in MINIX.

There is interest in operating systems that satisfy some security policy (see, e.g., [Gligor87]). Multilevel security (see, e.g., [FLR77]), wherein information does not flow from a user at security level l to a user whose security level l_1 is not at least that of l, is one particular such policy. As we will show below, our MINIX rapid prototype can be easily retrofitted to be multilevel secure by associating appropriate "security levels" with the objects and users, and by returning errors upon an access to any object that is outside the policy. In fact, we can define a "security template" general enough that specific instances can correspond to the standard MINIX data protection scheme as well as a much more stringent multilevel protection policy.

Our security template involves adding the concept of a User to the SRM template: Objects and Processes will all be owned by a User. A User (kind 1) will be a name (Symbol, primitive) coupled with an element of some partially ordered set (Poset, kind 3) describing the multi-level hierarchy. A "security level" for an Object will be an access control list (ACList, kind 1); that for a Process will be the Poset component of its owner. Access control lists list Users together with information on whether they have read, write, or permission-change permission to an Object. An element of the SecPol can be defined for each SRMop that ensures that no Request calling that SRMop can either violate the access control list of an Object, or change it so that it is inconsistent with the multi-level hierarchy. While the exact form of these definitions depends on the individual SRMops, they can all be based on a generic formulation of the restriction they must enforce that can be included in the template.

The standard MINIX data protection scheme can be fit into this template by using a one-element Poset, since MINIX has no multi-level security: any user can share contents of his objects at will with any other user. Superusers are distinguished by means of being especially known to the SecPol, which will, for example, make sure that no Request that would deny them any type of access to any Object will be granted. It is clear that the usual concept of superuser is inconsistent with any nontrivial multilevel security policy.

Thus, we can define an appropriate SecPol for any multi-level security policy. However, it must be noted that perfect enforcement of a security policy is nearly impossible. For example, there is usually information flow in violation of the security policy through the error returns mentioned above. A user could receive an error upon attempting to write to an object that no longer exists. A one-bit channel is thus present between two users not permitted by the security policy to communicate with each other, information being transmitted by a user who deletes an object if he wants to send a 1 and does not delete the object if he wants to send a 0; this channel is called a *covert channel.*

Potentially, a rapid prototype can be helpful in the discovery (and, to some extent, mitigation via design for bandwidth reduction) of such covert channels. These channels are present in the rapid prototype and can be discovered and their bandwidth determined (experimentally) through testing the rapid prototype. Additional covert channels can be introduced in the implementation, and discovered by testing incremental implementations of the prototype.

6. Templates in other executable specification systems.

We have so far found the FASE system to fill our rapid prototyping needs well. However, we wish to examine in detail whether the template notion combined with other specification systems might have advantages. Accordingly, we are also developing an operating system template in VDM [HI88], and also testing the template methodology (as applied to user interface managers) in OBJ3 [GM82, FFAL91].

It is clear that the template methodology can be used in some form in conjunction with either VDM or OBJ3. In any specification system, the obvious advantage of starting from a template is that the overall organization of the design is already thought out. In the FASE system, one is relieved of certain other decisions for a particular data types in the design depending on the kind of that data type: 1, 2, or 3. The distinction between kind 1 and kind 2 appears to be less important for VDM or OBJ3; however, being of at worst kind 2 assures that the data representation is fixed, which is very useful in VDM. The analog in OBJ3 appears to be that there is a "theory" that need not be redefined.

It appears that FASE has advantages over OBJ3 with respect to execution speed. We have not yet been able to compare EPROS (the executable subset of VDM) with respect to speed, but intend to do so.

Specification systems can also be compared on the basis of ease of proof of properties of specified data types; this is something we hope to do for the three systems in the future.

7. Future work.

As we have indicated already, we intend to complete our comparison of executable specification methods, both with respect to suitability for rapid prototyping of operating systems and other software systems such as an interface manager [FFAL91], and with respect to the details of a template specification methodology.

When studying properties of a system design, it is useful to be able to prove certain assertions about the system. For example, one may wish to prove that the system can never be in an undesirable state, by some definition (such as never having two distinct Objects which pass the "eqObject" test). It is of interest to be able to prove such an assertion from certain assertions about the SRMops, the Interp, and the SecPol. We will experiment with proving such assertions for specifications in the different specification languages, using the proof methods appropriate to each language.

We intend to complete of our multilevel secure prototype of MINIX, and experiment with using the prototype to study covert information channels. We also intend to experiment with other enhancements to an operating system that can be realized through a rapid prototype. Among those to be considered include fault tolerance, recovery and converting a single-host operating system to a distributed system.

When enhancements or modifications to an existing system have been designed through its prototype, the question remains how to translate these changes to the implementation. We will study this question with MINIX as our example.

References.

[ABFL91] M. Archer, J. Bock, D. A. Frincke, and K. Levitt, *Elaboration of the SRM operating system template into a MINIX rapid prototype*, Technical Report, Division of Computer Science, University of California, Davis (1991).

[AFL90] M. Archer, D. A. Frincke, and K. Levitt, *A template for rapid prototyping of operating systems*, International Workshop on Rapid System Prototyping, June 4-7, 1990.

[FFAL91] D. A. Frincke, G. Fisher, M. Archer, and K. Levitt, *A new application of template methodology: rapid prototyping of user interface management systems*, Second International Workshop on Rapid System Prototyping (Research Triangle Park, NC, June 11-13, 1991).

[FLR77] R. J. Fiertag, K. Levitt, and L. Robinson, *Proving multilevel security of a system design*, Proceedings of the Symposium on Operating System Principles (1977), 57-95.

[GM82] J. A. Goguen and J. Meseguer, *Rapid prototyping in the OBJ executable specification language*, ACM SIGSOFT Software Engineering Notes, 7(5) (December, 1982), 75-84.

[Gligor87] V. D. Gligor et al., *Design and implementation of secure xenix*, IEEE Transactions on Software Engineering **SE-13**, No. 2 (February, 1987), 208-221.

[HI88] S. Hekmatpour and D. Ince, *Software Prototyping, Formal Methods and VDM*, (Addison-Wesley, 1988).

[KJA83] S. Kamin, S. Jefferson, and M. Archer, *The role of executable specifications: the FASE system*, Proc. IEEE Symposium on Application and Assessment of Automated Tools for Software Development (November, 1983), 105-114.

[Tanen87] Andrew S. Tanenbaum, Operating Systems: Design and Implementation, Prentice-Hall (1987).

Appendix A: Definitions relevant to READ in MINIX prototype.

SRMop

```
nameSRMop : SRMop -> Symbol
apSRMop : SRMop State ArgList -> State
eqSRMop : SRMop SRMop -> Bool
NOop : -> SRMop
LOGIN : -> SRMop
uLOGIN : -> SRMop
CR : -> SRMop
uCR : -> SRMop
OPEN : -> SRMop
uOPEN : -> SRMop
CLOSE : -> SRMop
uCLOSE : -> SRMop
WRITE : -> SRMop
uWRITE : -> SRMop
READ : -> SRMop
uREAD : -> SRMop
LSEEK : -> SRMop
uLSEEK : -> SRMop
LINK : -> SRMop
UNLINK : -> SRMop
FORK : -> SRMop
uFORK : -> SRMop
KILL : -> SRMop
WAIT : -> SRMop
```

```
DISTINGUISHING SET nameSRMop apSRMop;

...
READ => ['READSRMop,    /* READ(fd,len,pid) from uREAD(fd,len) */
          <S,A> |->
            let pid be getpidArg(3, A) in
            let proc be findinProcessSet(isidProcessPred(pid), procsofState(S)) in
            let fd be getsymArg (1, A) in
            let curtab be ctofProcess(proc) in
            let place be lookupCursortab(fd, curtab) in
            if place = errInt then errState else
              let objid be getidCursortab(curtab, fd) in
              let O be objsofState(S) in
              let obj be findinObjectSet(isidObjectPred(objid),O) in
              let len be getintArg (2, A) in
              let end be lengthSymbolList(filecontObject(obj)) in
              if place+len > end then errState else
                let newcurtab be updateposCursortab(curtab, fd, place+len) in
                let buf be extractSymbolList(filecontObject(obj),place+1,len) in
                let newp1 be updatectProcess(proc, newcurtab) in
                let newp2 be updatebufProcess(newp1,buf) in
                updateprocState(proc, newp2, S)];

uREAD => ['uREADSRMop, <S,A> |-> S];
...
```

```
Interp

getreqInterp : State Request Interp -> Request
trivInterp : -> Interp
NOopInterp : -> Interp
MINIXInterp : -> Interp
 ...
uREADInterp : -> Interp
READInterp : -> Interp
 ...
DISTINGUISHING SET getreqInterp;

trivInterp => [<S,r> |-> r];

NOopInterp => [<S,r> |-> r];

 ...

uREADInterp => [<S,r> |-> mkRequest( READ,
                                     let pid be procidofRequest(r) in
                                     let A be argsofRequest(r) in
                                     appendArgList(pidtoArg(pid), A), sysProcessId)];

READInterp => [<S,r> |-> r];

 ...
```

Appendix B: An abbreviated MINIX prototype demo.

```
-> (runsafeMINIX)
Objects:
  Directory (/ ) contains:
System process:
  Buffer contains: [ ]
Requests Queued:

MINIX>  !system: uLOGIN(jim) in MINIX!
Objects:
  Directory (/ ) contains:
System process:
  Buffer contains: [ ]
Requests Queued:
  uLOGIN (jim) <System process>

MINIX>  !system: uLOGIN(karl) in MINIX!
Objects:
  Directory (/ ) contains:
System process:
  Buffer contains: [ ]
Requests Queued:
  uLOGIN (jim) <System process>
  uLOGIN (karl) <System process>

MINIX>  !step step MINIX!
Objects:
  Directory (/ ) contains:
System process:
  Buffer contains: [ ]
User process 0:
  Status: active
  Parent: System process
  Working Dir: /
  User: jim
  Buffer contains: [ ]
  CurTab contains: {}
User process 1:
  Status: active
  Parent: System process
  Working Dir: /
  User: karl
  Buffer contains: [ ]
  CurTab contains: {}
Requests Queued:

MINIX> ...

MINIX>  !0: uFORK() in MINIX!
Objects:
  Directory (/ ) contains:
System process:
  Buffer contains: [ ]
```

```
User process 0:
   Status: active
   Parent: System process
   Working Dir: /
   User: jim
   Buffer contains: [ ]
   CurTab contains: {}
 User process 1:
   Status: active
   Parent: System process
   Working Dir: /
   User: karl
   Buffer contains: [ ]
   CurTab contains: {}
 Requests Queued:
    uCR (fd1, oho, (Objectkind: file)) <User process 0>
    uOPEN (fd2, /oho, rd) <User process 1>
    uWRITE (fd1, [ hello world ]) <User process 0>
    uREAD (fd2, 1) <User process 1>
    uFORK () <User process 0>

 MINIX> !step step step step step MINIX!
 Objects:
    Directory (/ ) contains: oho
    File (/oho ) contains: [ hello world ]
 System process:
    Buffer contains: [ ]
 User process 0:
    Status: active
    Parent: System process
    Working Dir: /
    User: jim
    Buffer contains: [ ]
    CurTab contains: {fd1  wr  (/oho ) -> 2}
  User process 1:
    Status: active
    Parent: System process
    Working Dir: /
    User: karl
    Buffer contains: [ hello ]
    CurTab contains: {fd2  rd  (/oho ) -> 1}
   User process 2:
    Status: active
    Parent: User process 0
    Working Dir: /
    User: jim
    Buffer contains: [ ]
    CurTab contains: {fd1  wr  (/oho ) -> 2}
   Requests Queued:
```