Especificação Abstrata do Núcleo de um Sistema Operacional e sua formalização na linguagem Z

Luciano Barreto, Aline Andrade, Adolfo Duran, Caíque Lima, Ademilson Lima

[1]Departamento de Ciência da Computação
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros, s/n - Campus Universitário de Ondina
40.110-170 Salvador - BA - Brazil

{lportoba,aline,adolfo}@ufba.br,{cailemos,adesanlim}@dcc.ufba.br

Abstract. *One of the mini challenges in software verification related to the Grand Challenge defined by Tony Hoare concerns the formal specification and verification of an operating system kernel. This paper proposes a simple and correct (yet preliminary) specification of an OS kernel in Z, which simplifies the understanding and verification of operating system components.*

Resumo. *A especificação formal e verificação do núcleo de um sistema operacional constitui um dos mini desafios propostos pela comunidade de métodos formais no contexto dos Grandes Desafios em verificação de software (Grand Challenges) proposto por Tony Hoare. Este artigo apresenta uma proposta de kernel simples e correto (ainda que preliminar) especificada em Z, que simplifica a compreensão e verificação de componentes do sistema operacional.*

1. Introdução

A verificação formal e segura do comportamento do funcionamento de um Sistema Operacional (SO) é um desafio que perdura tanto no contexto industrial quanto no âmbito da comunidade acadêmica [Bevier 1989, Walker et al. 1980]. A complexidade inerente ao projeto e desenvolvimento de um SO, aliada aos sobrevalorizados requisitos de alto desempenho e reduzido *time-to-market* para lançamento de novas versões tornam essa missão ainda mais desafiadora. O objeto proposto neste artigo se insere no contexto do grande desafio (*grand challenge*) em verificação de software proposto por Hoare [Hoare 2003], voltado para o emprego de técnicas de verificação formal na resolução de problemas reais e de larga escala. Mais especificamente, neste contexto, nosso objetivo considera a formalização de famílias de sistemas operacionais que forneçam garantias sobre sua correção às aplicações e aos usuários.

A despeito da relevância do tema, permanece incipiente o número de trabalhos que versam sobre a construção de sistemas operacionais verificáveis e corretos. O livro escrito por Ian Craig [Craig 2007] propõe a formalização de um núcleo (kernel) simplificado de um sistema operacional com o intuito de garantir certas propriedades na concepção do sistema. Apesar da relevância desse trabalho, as especificações propostas são extensas e revelam aspectos menos genéricos (*i.e.*, muito especializados) acerca dos subsistemas do núcleo do SO em questão. Em particular, as políticas referentes à gestão de processos carecem de maior nível de abstração, sobrecarregando excessivamente a especificação do sistema operacional proposto (atualmente em torno de 32 páginas na linguagem de

especificação Z[Woodcock and Davies 1996]). Além disso, a especificação possui alguns erros e inconsistências, alguns destes descobertos pelos autores desse trabalho. Portanto, a necessidade da especificação formal, correta e concisa do kernel de um sistema operacional, permanece commo uma atividade relevante de pesquisa na comunidade científica.

Tais demandas motivaram a realização desse trabalho, o qual consiste na definição e na verificação da especificação de um kernel com o intuito de simplificar e abstrair ao máximo a funcionalidade dos componentes do sistema operacional. Nossa abordagem é voltada à definição das políticas e não dos mecanismos (*i.e.* questões de implementação e de baixo nível) referentes ao kernel. A apresentação e avaliação desse kernel é o principal objeto desse artigo.

O restante deste artigo está estruturado da seguinte maneira. A seção 2 apresenta as decisões de projeto e modelo do kernel abragendo aspectos relativos ao gerenciamento de processos, troca de mensagens e sincronização de processos. A seção 3 efetua uma avaliação preliminar da nossa abordagem, ao passo que a seção 4 apresenta os trabalhos correlatos. Por fim, a seção 5 apresenta as considerações finais e discute possíveis trabalhos futuros.

2. Lucix: um kernel simples e verificável

O objetivo desse artigo consiste na especificação abstrata das funcionalidades essenciais de um kernel de sistema operacional. O princípio fundamental de construção do kernel Lucix focaliza na especificação de políticas relativas aos componentes do sistema operacional (*e.g.*, escalonamento de processos, troca de mensagens, sincronização) sem atrelar à especificação desses componentes; em detrimento dos mecanismos (*i.e.*, bibliotecas e estruturas de dados) utilizados na implementação dessa políticas. Nosso objetivo consiste em construir um kernel funcional, o mais enxuto e legível quanto possível.

Nas próximas seções apresentaremos as idéias centrais da especificação do kernel aliadas às suas correspondentes especificações em Z, as quais foram modeladas e verificadas através da ferramenta Z/EVES[Saaltink 1997].

2.1. Visão Geral

A especificação de Lucix contém as funcionalidades essenciais de um microkernel na visão de Jochan Liedtke[Liedtke 1995]: gerenciamento básico de processos (criação, destruição, escalonamento); um mecanismo simples de comunicação entre processos, baseado no paradigma síncrono de troca de mensagens, implementado através de primitivas `send` e `receive`; e um mecanismo básico de sincronização de processos, baseados em semáforos binários *mutex*, implementado através de primitivas `up` e `down`. Aspectos relativos à concepção de *device drivers*, tratadores de interrupção e sistemas de arquivos, embora importantes, não serão tratados nesse artigo. Nesse âmbito, estamos atualmente trabalhando na especificação formal em Z de um sistema de arquivos verificável baseado no padrão POSIX.

A base da especificação do kernel consiste na subdivisão dos processos em conjuntos distintos, caracterizados pelo status de execução do processo. Assim, podemos verificar o comportamento coerente de um processo durante o seu ciclo de vida através da análise da validade das transições entre esses conjuntos/estados. Tal advento nos permite realizar diversas verificações simples, porém importantes no modelo do kernel, as

quais seriam arduamente verificáveis diretamente na análise manual do código fonte da implementação. Além disso, esta abordagem torna a especificação concisa e clara (por estar fundamentada em conceitos básicos de teoria de conjuntos) e, em nosso entendimento, facilita sobremaneira a compreensão do funcionamento do kernel (atividades relacionadas às chamadas de sistema, escalonamento de processos etc.).

2.2. Especificação do kernel em Z

O cerne das especificações em Z são os chamados *schemas* (doravante utilizaremos o termo esquema, em português), os quais estruturam e encapsulam objetos. A estrutura básica de um esquema contém uma declaração de variáveis e um predicado que restringe os valores das variáveis declaradas. Os predicados são essenciais na definição de pré e pós-condições associadas a uma operação, por exemplo. Um esquema possui escopo global podendo ser utilizado na definição de outros esquemas ou estruturas da linguagem.

Tal qual outros SOs, em Lucix nos referimos a noção de *processo* como entidade básica de execução. Cada processo é caracterizado por uma lista de atributos (descritos pelo esquema `PROCESS`): um identificador único (`pid`), um identificador do seu criador ou processo pai (`ppid`) e um nível de importância (`prio`) utilizado para definir a ordem de execução dos processos no processador (supomos aqui um modelo de escalonamento no qual os processos tem uma prioridade fixa).

$$PID == \mathbb{N}$$

PROCESS

$$pid : PID$$
$$ppid : PID$$
$$prio : PID$$

O kernel inicia sua operação através da execução do processo `Init`, o qual efetua a criação dos demais processos do sistema, produzindo, assim, uma arborescência dos mesmos. Este processo está definido no esquema `PROCESSInit`, descrito a seguir. O processo `Init` possui os atributos `pid` e `ppid` com o valor zero e valor de prioridade 0, ou seja, é o processo menos importante do sistema. Cabe ressaltar que as aspas simples num esquema Z (como em `pid´`) representam a ocorrência da modificação do estado desse atributo.

PROCESSInit

$$PROCESS'$$

$$pid' = 0$$
$$ppid' = 0$$
$$prio' = 0$$

Cabe ao processo `Init` a criação dos demais processos do sistema operacional como, por exemplo, o interpretador de comandos *shell*) ou *scripts* de inicialização dos demais serviços. Após a inicialização dos serviços do sistema operacional, supõe-se que

o código do processo `Init` seja um laço infinito sem qualquer funcionalidade adicional. Como consequência, o processo `Init` será executado toda vez que não houver processos prontos para executar. Nesse modelo, portanto, o processo `Init` assume as funções do processo que executa quando o sistema estiver ocioso, também chamado de *idle process* em alguns sistemas operacionais. Dessa forma, adicionar procedimentos de gerenciamento de energia quando da ociosidade do processador se resume a dotar o código do processo `Init` de tal funcionalidade.

2.3. Gerenciamento de processos

Em Lucix, os processos do kernel são organizados em conjuntos disjuntos cujos membros possuem a mesma característica relativa ao seu estado/status de execução. A Figura 1 apresenta o diagrama de estados finitos que descreve o modelo de conjuntos de processos utilizados em nossa abordagem. Os círculos representam conjuntos de processos que têm em comum o mesmo estado de execução e as setas indicam as transições válidas entre os conjuntos.

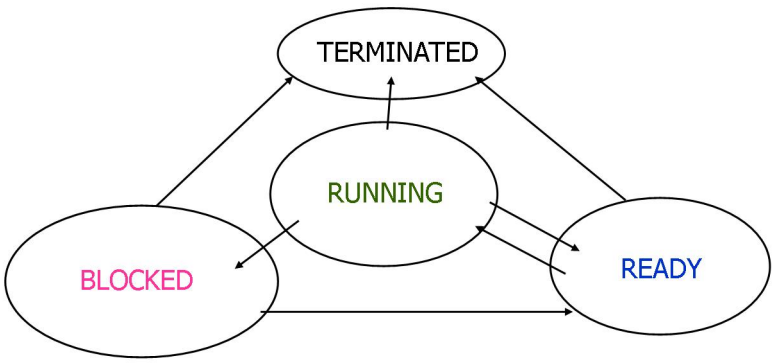

Figure 1. Conjuntos de processos em Lucix

O conjunto `ready` contém os processos prontos para executar, o conjunto `blocked` armazena os processos que estão bloqueados à espera de algum evento, o conjunto `running` (unitário em máquinas uniprocessadas) abriga os processos em execução e o conjunto `terminated` denota os processos finalizados. A união de todos esses conjuntos é representada pelo conjunto `allprocess`. As transições de estado entre conjuntos são disparadas por meio de eventos específicos do sistema operacional considerado. Em essência, tais eventos são provocados por interrupções de hardware (*e.g.*, final de operação de E/S, alarme disparado pelo temporizador/relógio do sistema e chamadas de sistema ou *system calls* (*e.g.*, criação e término de processo, aquisição e liberação de semáforo, leitura e escrita em arquivos). Alguns exemplos de eventos são: bloqueio e desbloqueio de processo devido a operações de entrada/saída, preempção do processo corrente por um processo mais importante ou por estouro do seu quantum de tempo de execução etc.

Ainda que os eventos sejam específicos ao sistema operacional considerado, é importante garantir que as ações realizadas pelo tratamento de tais eventos sejam coerentes em relação ao funcionamento do kernel. Por exemplo, o bloqueio de um processo causado por uma chamada à função `down()` para obtenção de um semáforo deve, obrigatoriamente, transferir este processo do conjunto `running` para o conjunto `blocked`.

A especificação desse modelo de sistema em Z está encapsulada no esquema `OSPROCESSES` a seguir. Cada processo é associado a um identificador de processo único. Para tanto, definimos um contador `lastpid` que guarda o último valor de identificador usado e é incrementado quando da criação de um novo processo. Assim, garantimos a unicidade dos identificadores de processo no sistema operacional. Os conjuntos `ready`, `blocked`, `terminated`, `running`, e `allprocesses` são definidos como subconjuntos finitos de `PROCESS`. Há ainda os conjuntos `sending`, `receiving` e `blocked_others` que são subconjuntos de `blocked` e que contém, respectivamente, os processos que aguardam para enviar mensagens, os que aguardam para receber mensagens e os demais processos bloqueados à espera de outros eventos (a barra invertida representa a operação de diferença entre conjuntos). Em seguida, consideramos que todos esses conjuntos são disjuntos dois a dois, representados pela intersecção vazia entre tais pares de conjuntos. Por fim, especificamos o conjunto `allprocesses` como a união de todos os demais conjuntos e definimos que dois processos são iguais se, e somente se, possuem o mesmo `pid`.

OSPROCESSES

$$
\begin{array}{l}
lastpid : PID \\
ready : \mathbb{F}\, PROCESS \\
blocked : \mathbb{F}\, PROCESS \\
sending : \mathbb{F}\, PROCESS \\
receiving : \mathbb{F}\, PROCESS \\
other\_blocked : \mathbb{F}\, PROCESS \\
terminated : \mathbb{F}\, PROCESS \\
running : \mathbb{F}\, PROCESS \\
allprocesses : \mathbb{F}\, PROCESS \\
\hline
sending \subseteq blocked \\
receiving \subseteq blocked \\
sending \cap receiving = \emptyset \\
blocked\_others = blocked \setminus sending \setminus receiving \\
ready \cap blocked = \emptyset \\
ready \cap terminated = \emptyset \\
ready \cap running = \emptyset \\
blocked \cap terminated = \emptyset \\
blocked \cap running = \emptyset \\
terminated \cap running = \emptyset \\
allprocesses = ready \cup blocked \cup terminated \cup running \\
\forall q, r : allprocesses \bullet q = r \Leftrightarrow q.pid = r.pid
\end{array}
$$

Como propriedade fundamental, definimos que os conjuntos `ready`, `blocked`, `terminated` e `running` são disjuntos. Ou seja, nenhum processo pode pertencer a mais de um conjunto simultaneamente. Ademais, as ações que envolvem a entrada e retirada de processos dos conjuntos devem satisfazer a condição de que um processo sempre deve pertencer a apenas um dado conjunto por vez. Em outros termos, as ações, garantidamente, sempre deixam um processo em um dos conjuntos.

O esquema `OSPROCESSESInit` caracteriza o estado inicial do sistema opera-

cional. Inicialmente, os conjuntos `ready`, `blocked` e `terminated` estão vazios e o processo `Init` é posto em execução, ou seja, no conjunto `running`. Por fim, a última linha do esquema denota que o conjunto `allprocesses` contém apenas o processo `Init`, visto que os demais conjuntos estão vazios.

OSPROCESSESInit

$$
\begin{array}{|l}
\hline
OSPROCESSES' \\
\hline
ready' = \emptyset \\
blocked' = \emptyset \\
terminated' = \emptyset \\
lastpid' = 0 \\
\exists PROCESS' \bullet PROCESSInit \land running' = \{\theta PROCESS'\} \\
allprocesses' = running' \\
\hline
\end{array}
$$

2.3.1. Criação e destruição de processos

A fim de gerenciar os processos, definimos modelos que definem o comportamento das chamadas de sistema através dos esquemas `PCREATE` e `PEND`, os quais criam e finalizam processos, respectivamente. Neste artigo, apresentamos esquemas simplificados, os quais abstraem as questões referentes à inicialização de atributos específicos do processo tais como: informações de contexto referentes aos espaços de endereçamento do processo, arquivos abertos e em uso, semáforos e outras variáveis de lock em uso etc. Por óbvio, estes esquemas podem ser facilmente especializados em razão das funcionalidades específicas desejadas pelo projetista do kernel.

O esquema `PCREATE`, descrito a seguir, ilustra as principais ações executadas na criação de um novo processo. Em primeiro lugar, a variável `lastpid` é incrementada e são inicializados dois atributos do novo processo: `pid` (a partir do valor de `lastpid`) e `ppid` (do processo pai, o qual invocou a chamada de sistema). Por fim, o novo processo é colocado no conjunto de processos prontos (`ready`). Vale ressaltar que a modelagem deste tipo de esquema visa abstrair as estruturas de dados e mecanismos de implementação empregados na concepção de uma chamada de sistema tal qual `fork` em sistemas Unix.

PCREATE

$$
\begin{array}{|l}
\hline
\Delta OSPROCESSES \\
pproc? : PROCESS \\
proc! : PROCESS \\
\hline
lastpid' = lastpid + 1 \\
proc!.pid = lastpid' \\
proc!.ppid = pproc?.pid \\
ready' = ready \cup \{proc!\} \\
\hline
\end{array}
$$

O esquema PEND, por sua vez, denota uma chamada de sistema que implementa o término voluntário de um processo em execução, tal qual a chamada de sistema exit em sistemas Unix. De forma abstrata, desconsiderando especificidades do kernel, o processo chamador finaliza sua execução através de sua remoção do conjunto running para terminated.

PEND

$\Delta OSPROCESSES$
$proc! : PROCESS$

$running' = running \setminus \{proc!\}$
$terminated' = terminated \cup \{proc!\}$

2.4. Comunicação entre Processos

Esta seção descreve a modelagem de comunicação entre processos de forma síncrona em Lucix. Na abordagem adotada, o processo emissor não precisa ser bloqueado se houver outro processo esperando por sua mensagem; ou seja, o emissor realiza a entrega da mensagem e prossegue seu fluxo de execução. Extensões para implementação de primitivas assíncronas e outras variantes de comunicação são igualmente possíveis e requerem pouco esforço de implementação.

O esquema MSG a seguir define os campos de uma mensagem. O atributo source denota o processo emissor da mensagem, target simboliza o processo receptor, header representa o cabeçalho da mensagem e, por fim, o campo data possui a sequência de dados (bytes) transferida entre os processos.

$BYTE == 0..255$

MSG

$source : PID$
$target : PID$
$header : \mathrm{seq}\, BYTE$
$data : \mathrm{seq}\, BYTE$

A definição do serviço de entrega de mensagens síncronas (esquema MSGSyncService) é representado pela disjunção de dois esquemas MSGSyncSend e MSGSyncSendBlk. A diferença entre eles é que, no primeiro caso, o emissor da mensagem permanece em execução ao passo que, no segundo caso, o processo emissor é bloqueado. Portanto, para que uma mensagem seja enviada, é necessário que o processo emissor esteja em execução (conjunto running). Ao final, o processo emissor permanecerá em execucão (caso o receptor esteja aguardando por sua mensagem) ou bloqueado (caso o receptor ainda não esteja aguardando pela mensagem).

$MSGSyncService \mathrel{\widehat=} MSGSyncSend \lor MSGSyncSendBlk$

O esquema `MSGSyncSend` representa a situação na qual o processo receptor está aguardando a mensagem, ou seja, pertence ao conjunto `receiving`. Neste caso, o emissor continua sua execução normalmente e o processo receptor é desbloqueado devido ao recebimento da mensagem.

MSGSyncSend

$$
\begin{array}{l}
\Delta OSPROCESSES \\
src? : PID \\
trg? : PID \\
header? : \mathrm{seq}\, BYTE \\
data? : \mathrm{seq}\, BYTE \\
msgSend! : MSG \\
\hline
\exists s, t : PROCESS \mid s \in running \wedge t \in receiving \bullet s.pid = src? \wedge t.pid = trg? \\
\wedge\ msgSend! = \theta MSG[source := src?, target := trg?, header := header?, data := data?] \\
\wedge\ blocked' = blocked \setminus \{t\} \wedge ready' = ready \cup \{t\}
\end{array}
$$

Por sua vez, o esquema `MSGSyncSendBlk` representa a situação na qual o processo receptor não está aguardando a mensagem (*i.e.* não pertence ao conjunto `receiving`). Neste caso, o emissor é bloqueado por meio da sua remoção do conjunto `running` seguida de posterior inclusão no conjunto `sending`.

MSGSyncSendBlk

$$
\begin{array}{l}
\Delta OSPROCESSES \\
src? : PID \\
trg? : PID \\
header? : \mathrm{seq}\, BYTE \\
data? : \mathrm{seq}\, BYTE \\
msgSend! : MSG \\
\hline
\exists s, t : PROCESS \mid s \in running \wedge t \notin receiving \bullet s.pid = src? \wedge t.pid = trg? \\
\wedge\ msgSend! = \theta MSG[source := src?, target := trg?, header := header?, data := data?] \\
\wedge\ running' = running \setminus \{s\} \wedge sending' = sending \cup \{s\}
\end{array}
$$

O esquema `MSGReceiveService` representa o serviço de recebimento síncrono de mensagens por um processo. Este esquema é construído através da disjunção dos esquemas `MSGSyncReceive` e `MSGSyncReceiveBlk`, descritos a seguir.

$$MSGReceiveService \mathrel{\widehat=} MSGSyncReceive \vee MSGSyncReceiveBlk$$

O esquema `MSGSyncReceive` define a situação na qual o processo em execução recebe, de forma síncrona, uma mensagem já enviada por outro processo (representado anteriormente pelo esquema `MSGSyncSendBlk`). Neste caso, o processo emissor encontra-se no estado `sending`. Portanto, cabe basicamente ao esquema `MSGSyncReceive` remanejar o processo emissor do conjunto `blocked` para o conjunto `ready`.

MSGSyncReceive

$\Delta OSPROCESSES$
$msg : MSG$

$\exists s, t : PROCESS \mid s \in sending \land t \in running \bullet s.pid = msg.source$
$\land\ t.pid = msg.target \land blocked' = blocked \setminus \{s\} \land ready' = ready \cup \{s\}$

Por outro lado, o esquema `MSGSyncReceiveBlk` representa o cenário no qual o processo emissor da mensagem não está apto a enviá-la, ou seja, ainda não foi colocado no conjunto `sending`. Nesta situação, o processo receptor sai de execução (do conjunto `running`) e, em seguida, é posto em espera (no conjunto `receiving`).

MSGSyncReceiveBlk

$\Delta OSPROCESSES$
$msg : MSG$

$\exists s, t : PROCESS \mid s \notin sending \land t \in running \bullet s.pid = msg.source$
$\land\ t.pid = msg.target \land running' = running \setminus \{t\} \land receiving' = receiving \cup \{t\}$

3. Avaliação preliminar

Z é uma linguagem de especificação formal baseada na notação de conjunto e lógica matemática, que permite a verificação da correção da especificação através de verificadores automáticos de teoremas. Através da sua notação de esquemas, é possível a criação de um modelo abstrato do kernel que poderá ser refinado para modelos de implementação específicos. Os esquemas agregam à Z capacidades de modularização, encapsulamento, reuso, entre outras facilidades. Portanto, a modelagem de um sistema operacional em Z dirige o centro de atenção do projetista nas questões essenciais (políticas) do kernel, dispensando a escolha de mecanismos particulares para a implementação de tais políticas. A separação entre políticas e mecanismos consiste em uma importante meta no projeto de sistemas operacionais corretos e reutilizáveis.

A ferramenta Z-Eves é um provador automático de teoremas que permite a realização de provas de consistência, verificação de pré-condições e provas específicas do domínio da aplicação em especificações escritas em Z. Neste trabalho utilizamos esta ferramenta para verificar a especificação abstrata do kernel proposto, garantindo a correção da mesma frente a estas propriedades. Efetuar provas de teoremas no Z/EVES, no entanto, requer expertise e, por vezes, a definição de axiomas adicionais para otimizar o processo de prova, o que reduz consideravelmente a concisão das especificações. Mas esta característica é inerente aos provadores automáticos de teoremas, que podem ser considerados em realidade como checadores de provas, pois dependem fortemente da intervenção humana para guiar o processo de verificação.

A especificação Z de Lucix atualmente consiste de 13 esquemas que utilizam em torno de 80 linhas de especificação, considerando as declarações e predicados. Esta especificação define o funcionamento básico do kernel em torno da gestão de processos, envio de mensagens síncronas e assíncronas e sincronização de processos através de

semáforos (estes dois itens não descritos neste artigo). Do ponto de vista qualitativo, a despeito da necessidade da fluência em Z para entendimento das especificações, o código Z de Lucix é conciso e fácil de ler. O nosso modelo é mais simples e tem um maior nível de abstração do que o proposto por Craig, podendo ser mais facilmente estendido para outras especificações que especializam políticas e mecanismos de uma determinada família de kernel do sistema operacional.

4. Trabalhos Correlatos

O livro de Ian Craig [Craig 2007] constitui o trabalho mais próximo em relação a nossa abordagem. Neste livro, o autor apresenta alguns tipos de kernels e suas especificações em Z. Embora inovador nesse aspecto, as especificações são pouco legíveis e contém erros. Além disso, algumas especificações carecem de maior nível de abstração no ponto em que misturam inadequadamente política e mecanismo. Por exemplo, o escalonador é baseado em prioridades fixas implementados como uma lista de processos ordenada, o que requer a especificação e verificação das funções de gestão dessa lista, prejudicando fatalmente a legibilidade e abstração.

Recentemente, grande atenção tem sido devida ao seL4[Klein et al. 2009b, Klein et al. 2009a] por ter sido considerado o primeiro kernel de sistema operacional completamente provado formalmente. As especificações foram traduzidas para Haskell e analisadas pelo provador de Isabelle/HOL. Os autores atestam que todas as funcionalidades importantes foram verificadas em software até o nível de implementação em C.

Outra corrente relevante, e bem sucedida, em termos de verificação formal e busca de erros em sistemas operacionais tem se empenhado na aplicação de técnicas de verificação de modelos (*model checking*) em sistemas de arquivos[Yang et al. 2006] e alocação de memória[Gallardo et al. 2009], por exemplo.

5. Considerações finais

Este artigo apresentou os fundamentos da modelagem de um kernel simples e verificável para construção de um sistema operacional ou famílias de sistemas operacionais. A especificação é concisa (atualmente 80 linhas entre declarações e predicados em Z), o que a torna legível e evita a utilização de mecanismos de baixo nível que compliquem seu entendimento e verificação.

Ainda em consonância com os desafios em verificação de software, estamos especificando o sistema de arquivos baseado no padrão POSIX com o intuito de integrá-lo ao kernel definido nesse artigo. Como trabalhos futuros, é importante especificar e verificar outros componentes importantes do kernel a exemplo do mecanismo de interrupções de hardware e *device drivers*, não contemplados nessa proposta.

Agradecimentos

Este projeto foi parcialmente financiado pelo CNPq através de recursos do Edital Universal MCT/CNPq 15/2007 (Projeto 484179).

References

Bevier, W. R. (1989). Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396.

Craig, I. D. (2007). *Formal Refinement for Operating Systems Kernels*. Springer.

Gallardo, M. D. M., Merino, P., and Sanán, D. (2009). Model checking dynamic memory allocation in operating systems. *Automated Reasoning*, 42(2-4):229–264.

Hoare, C. A. R. (2003). The verifying compiler software grand challenge for computer research. *Journal of ACM*, 50(1):63–69.

Klein, G., Derrin, P., and Elphinstone, K. (2009a). Experience report: sel4: formally verifying a high-performance microkernel. In *Proceedings of the 14th ACM SIGPLAN international Conference on Functional Programming (ICFP)*, pages 91–96, New York, NY, USA. ACM.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. (2009b). sel4: formal verification of an os kernel. In *Proceedings of the 22nd Symposium on Operating Systems Principles (SOSP)*, pages 207–220, New York, NY, USA. ACM.

Liedtke, J. (1995). On micro-kernel construction. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)*, pages 237–250, New York, NY, USA. ACM.

Saaltink, M. (1997). The z/eves system. In *Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation (ZUM)*, pages 72–85, London, UK. Springer-Verlag.

Walker, B. J., Kemmerer, R. A., and Popek, G. J. (1980). Specification and verification of the ucla unix security kernel. *Commun. ACM*, 23(2):118–131.

Woodcock, J. and Davies, J. (1996). *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Yang, J., Twohey, P., Engler, D., and Musuvathi, M. (2006). Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423.