CENTRO UNIVERSITÁRIO DE ANÁPOLIS – UniEVANGÉLICA
BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO

UTILIZAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO RUST NO DESENVOLVIMENTO DE SISTEMAS OPERACIONAIS

PHILIPE ALEXANDRE FIAIA COSTA

Anápolis - GO
2019

PHILIPE ALEXANDRE FIAIA COSTA

UTILIZAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO RUST NO DESENVOLVIMENTO DE SISTEMAS OPERACIONAIS

Trabalho de Conclusão de Curso II apresentado como requisito parcial para a conclusão da disciplina de Trabalho de Conclusão de Curso II do curso de Bacharelado em Engenharia de Computação do Centro Universitário de Anápolis - UniEVANGÉLICA.

Orientador: Prof. Alexandre Moraes Tannus

Anápolis - GO

2019

PHILIPE ALEXANDRE FIAIA COSTA

UTILIZAÇÃO DA LINGUAGEM DE PROGRAMAÇÃO RUST NO DESENVOLVIMENTO DE SISTEMAS OPERACIONAIS

Trabalho de Conclusão de Curso II apresentado como requisito parcial para a obtenção de grau do curso de Bacharelado em Engenharia de Computação do Centro Universitário de Anápolis – UniEVANGÉLICA.

Aprovado(a) pela banca examinadora em 4 de dezembro de 2019, composta por:

Alexandre Moraes Tannus
Alexandre Moraes Tannus
Presidente da Banca

Aline Dayany de Lemos
Aline Dayany de Lemos
Prof(a). Convidado(a)

Millys Fabrielle Araujo Carvalhaes
Prof(a). Convidado(a)

“A coisa mais misericordiosa do mundo, creio eu, é a incapacidade da mente humana de correlacionar tudo o que sabe.”

(H. P. Lovecraft)

Resumo

Este trabalho apresenta a proposta da utilização da linguagem de programação Rust no desenvolvimento de sistemas operacionais, que são normalmente desenvolvidos utilizando a linguagem de programação C. Por ser uma linguagem relativamente antiga (1972), diversos pontos negativos foram subvertidos por novas linguagens, sendo Rust uma delas.

Para analisar o uso de Rust, foram comparados um sistema operacional escrito em C (Minix) e um sistema operacional escrito em Rust (Redox), tendo os estudos de caso de DEITEL, DEITEL e CHOFFNES (2005) como pontos a serem analisados, além da execução do algoritmo de Matmul e Base64 para a medição de desempenho. Com esta publicação, espera-se identificar a viabilidade da utilização de Rust no desenvolvimento de sistemas operacionais.

Palavras-chave: Linguagem de Programação Rust; Linguagem de Programação C; Sistemas Operacionais; Análise de Sistemas Operacionais; Análise de desempenho.

Abstract

This scientific work presents the propose of utilization of the programming language Rust in operating systems development, that are ordinarily developed using the C programming language. Despite being a relatively old language (1972), several negative points were subverted by newest languages, Rust is one of this languages.

To analyze the use of Rust, were compared an operating system wroted in C (Minix) and an operating system wroted in Rust (Redox), by having the case studies of DEITEL, DEITEL and CHOFFNES (2005) as points to be analyzed, besides the execution of Matmul and Base64 algorithm to measure the performance. With this publication, is expected to identify the viability of utilization of Rust on operating systems development.

Keywords: Rust Programming Language; C Programming Language; Operating Systems; Operating Systems Analysis; Performance Analysis.

Lista de Abreviaturas e Siglas

API	*Application Programming Interface*
Ext	*Extended File System*
EHCI	*Extensible Host Controller Interface*
FAT	*File Allocation Table*
Grub	*GRand Unifield Bootloader*
HPFS	*High Performance File System*
HD	*Hard Disk*
ISO	*International Organization for Standardization*
LILO	*LInux LOader*
Kb	*Kilobytes*
MS-DOS	*Microsoft Disk Operating System*
NCP	*NetWare Control Protocol*
NTFS	*New Technology File System*
POSIX	*Portable Operating System Interface*
PROCFS	*Proc FIle System*
RAM	*Random Access Memory*
ReiserFS	*Reiser File System*
SMB	*Server Message Block*
SysV	*System Five*
UDF	*Universal Disc Format*
UFS	*Unix File System*
URL	*Uniform Resource Locator*
USB	*Universal Serial Bus*
XFCE	*XForms Common Environment*

Lista de Tabelas

Lista de Ilustrações

Sumário

1. Introdução

Sistemas computacionais são constituídos por diferentes componentes de *hardware*. O controle e otimização de tais componentes é uma tarefa extremamente difícil no desenvolvimento de sistemas. O trabalho de um sistema operacional é justamente tal gerenciamento, além de oferecer aos desenvolvedores uma interface mais simples de trabalhar, pois é a camada mais baixa de interação do *software* com o *hardware* (TANENBAUM, 2003, OLIVEIRA; CARISSIMI; TOSCANI, 2010). Os sistemas operacionais são em grande parte desenvolvidos utilizando a linguagem de programação C. Tal linguagem foi criada em 1972 objetivando a não utilização de linguagens de baixo nível, como Assembly no desenvolvimento do sistema operacional Unix (DAMAS, 2016). Com o passar dos anos, outras linguagens de programação foram criadas, contornando problemas que C possui. Entretanto, C continua no mercado de desenvolvimento de sistemas operacionais, e seus problemas, por sua vez, continuam prejudicando os desenvolvedores de sistemas.

Se não houvessem sistemas operacionais, o trabalho de um desenvolvedor de *software* seria significativamente árduo, pois o mesmo deveria ter conhecimentos de cada *hardware* para o qual estivesse desenvolvendo a aplicação.

Em função dos fatos históricos citados anteriormente, sistemas operacionais têm como sua principal linguagem de programação, a linguagem C. Esta linguagem possui problemas, como difícil localização de erros e gerenciamento de memória, o que linguagens mais atuais já corrigiram ou amenizaram.

Utilizar uma linguagem moderna (criada em 2009), como Rust, poderia a princípio resolver alguns dos problemas de C, já que tem como alguns de seus princípios solucionar a pouca atenção em segurança, o pouco suporte à concorrência, falta de *affordance* (potencial de algo ser utilizado como foi projetado) e controle de recursos limitado, proporcionando funções nativas que ajudam no controle dos problemas citados, além de um compilador robusto que pode prever e impedir erros de execução (como por exemplo, ponteiros mal referenciados).

Promover a ampla utilização de Rust no desenvolvimento de sistemas operacionais pode ocasionar, de acordo com NILSSON e ADOLFSSON (2017), uma mudança no modo em que sistemas operacionais são desenvolvidos, beneficiando os possíveis utilizadores da linguagem.

Também é válido ressaltar a importância para o meio acadêmico, pois tal contexto pode ascender a maior produção de estudos e pesquisas no quesito produção de sistemas

operacionais. Sendo assim, é um processo que pode iniciar na academia e estender-se para a indústria de desenvolvimento de sistemas, beneficiando empresas, desenvolvedores e até estudantes.

Caso um sistema operacional fosse desenvolvido utilizando Rust, seria possível diminuir ou até mesmo resolver os problemas causados pelo uso da linguagem C? Qual a viabilidade no desenvolvimento de sistemas operacionais utilizando Rust?

2. Objetivos

2.1. Objetivo Geral

Comparar sistemas operacionais de propósito geral desenvolvidos em C e Rust.

2.2. Objetivos Específicos

- Analisar os principais requisitos funcionais e não funcionais de um sistema operacional.

- Verificar métricas de desempenho das linguagens que compõem os sistemas operacionais Redox e Minix.

3. Referencial Teórico

Sistemas computacionais são criados com o principal objetivo de facilitar a resolução de problemas dos usuários, e para atingir tal objetivo, os usuários utilizam de aplicações (programas), que por sua vez necessitam da utilização de *hardware* (componentes físicos) para fazer cálculos, armazenar dados e etc. A utilização de *hardware* por cada aplicação implica em alguns problemas, como: escrita de códigos específicos para cada componente físico, diferentes aplicações deveriam possuir o mesmo código (ou códigos semelhantes) para as mesmas operações de acesso/controle de *hardware*, cada *hardware* só possuiria uma aplicação executada por vez (SILBERSCHATZ; GALVIN; GAGNE, 2015).

Desenvolver um sistema operacional é uma tarefa complexa, onde o programador deve se atentar aos mínimos detalhes, já que o *hardware* não é fácil de se utilizar/programar (SILBERSCHATZ; GALVIN; GAGNE, 2015).

3.1. Desenvolvimento de *software* e linguagem de programação

O desenvolvimento de um *software* é feito por meio da programação, que pode ser entendida como um conjunto de instruções dadas a um *hardware*, que serão seguidas para a resolução de problemas, execução de processos, etc. O meio de comunicação entre o computador e o desenvolvedor de *software* (ou programador) é a linguagem de programação (VAREJÃO, 2004).

Linguagens de programação são chamadas assim, pois, se assemelham às linguagens naturais no quesito comunicação de ideias, mas se diferem no modo como o faz. A comunicação é feita entre programadores e computadores, e tem um domínio de expressão reduzido quando comparado às linguagens naturais, o que facilita a comunicação de ideias computacionais. Assim como as linguagens naturais, também existem diversos tipos de linguagens de programação, e cada uma cumpre um paradigma específico, ou seja, padrões de resolução de problemas, que geralmente são relacionados aos gêneros dos sistemas (como sistemas desktop, móveis, embarcados, distribuídos, web e etc.) (TUCKER; NOONAN, 2010).

Atualmente, os sistemas operacionais mais utilizados são o Windows, Linux e OS X, que foram desenvolvidos tendo como sua principal linguagem de programação a linguagem C.

3.1.1. Linguagem C

Segundo DAMAS (2016), a linguagem de programação C foi desenvolvida pela *Bell Telephone Laboratories* em 1972, para ser utilizada no desenvolvimento do sistema

operacional Unix, visando evitar a utilização de linguagens de baixo nível, como Assembly (linguagem em que o Unix começou a ser desenvolvido). Os sistemas operacionais mais utilizados atualmente tiveram fortes influências pelo Unix, e em razão disto, também foram desenvolvidos utilizando a linguagem C.

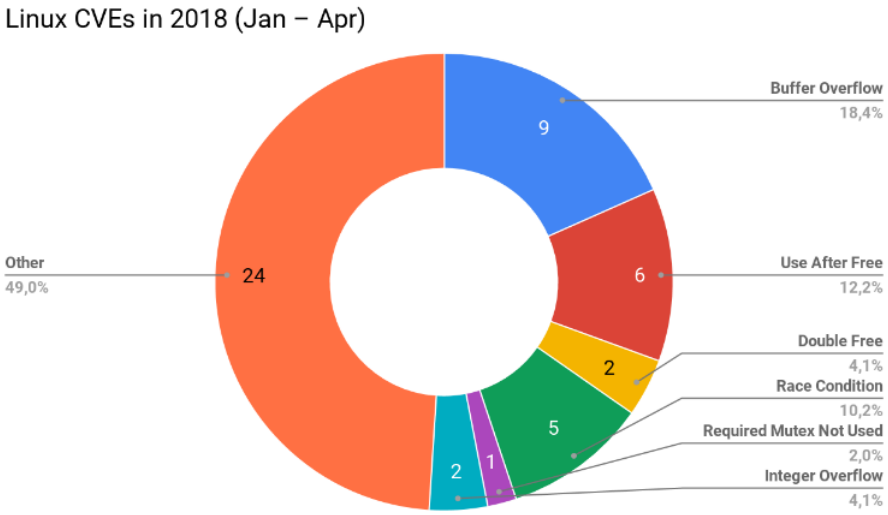
Apesar da linguagem C ser amplamente utilizada no desenvolvimento de sistemas operacionais, existem problemas que podem ser causados pelo uso indevido da linguagem. Em C, deve-se lidar explicitamente com gerenciamento de memória, o que frequentemente causam sistemas defeituosos, já que diversos programadores têm dificuldades em entender. Encontrar *bugs* (erros) na fase de desenvolvimento do *software* torna-se uma tarefa complicada quando a linguagem de programação C é utilizada, já que seu compilador mais utilizado, o GCC, não previne erros de execução, e oferece poucas ferramentas de *debug* de código. Quando *bugs* não são detectados antes da implantação do *software*, problemas como segurança e privacidade se tornam críticos, além do alto custo de correção dos erros (NILSSON; ADOLFSSON, 2017).

3.1.2. Linguagem Rust

Linguagens de programação modernas resolveram vários problemas que acontecem em C. Uma dessas linguagens, é Rust, que de acordo com a documentação, tem como objetivos oferecer uma alternativa para outras linguagens que não possuem nível satisfatório de abstração e eficiência. Particularmente em relação à pouca atenção em segurança, pouco suporte à concorrência, falta de *affordance* e controle de recursos limitado (RUST-FAQ, 2018?).

Para isto, Rust conta com um compilador que exerce a função de *gatekeeper*, recusando compilar códigos com *bugs* "enganadores", como ponteiros mal referenciados, duplo *free* de memória, e até mesmo problemas de concorrência, além de garantir estabilidade por meio de adição de recursos e refatoração (KLABNIK; NICHOLS, 2018). Tal compilador também checa cada variável e endereço de memória referenciável, o que pode parecer um empecilho para uma escrita de um código eficaz e expressivo, porém, acontece o contrário, já que escrever um código "perigoso" é praticamente impossível, pois quase todas as operações causariam problemas de segurança de memória, o que seria barrado pelo compilador (KASHITSYN, 2018).

Figura 1 - Vulnerabilidades do *kernel* Linux



Fonte: CVE Details (2018)

A Figura 1 mostra causas de erros no *kernel* Linux durante os meses de janeiro e abril no ano de 2018, e pode-se notar, que vários *bugs* do sistema operacional Linux são causados por concorrência e segurança de memória, o que seria fundamentalmente impossível de acontecer em um subconjunto seguro de Rust, ou seja, a utilização de Rust, poderia prevenir diversos *bugs* do Linux em sua fase de desenvolvimento (KASHITSYN, 2018).

Atualmente, já existem sistemas operacionais que utilizam de Rust em seu desenvolvimento. O que possui desenvolvimento mais avançado é o Redox. Sendo um sistema operacional de propósito geral, utilizado em *desktops*, Redox é escrito em puro Rust buscando trazer as inovações da linguagem para um sistema operacional (REDOX, 2018?). Outro sistema que pode-se tomar como caso de sucesso é o Tock, que é um sistema operacional destinado à dispositivos embarcados (dispositivos geralmente utilizados para a Internet das Coisas) (TOCK, 2018).

3.2. Desenvolvimento de sistemas operacionais

Para analisar se Rust é uma linguagem viável para o desenvolvimento de sistemas operacionais, alguns conceitos de sistemas operacionais devem ser entendidos, como por exemplo, a arquitetura de um *kernel*. De acordo com MACHADO e MAIA (2011), algumas das principais funções de um núcleo de um sistema operacional são:

- Criação e eliminação de processos e *threads*;
- Sincronização e comunicação entre processos e *threads*;
- Gerenciamento de memória;

- Sistemas de arquivos.

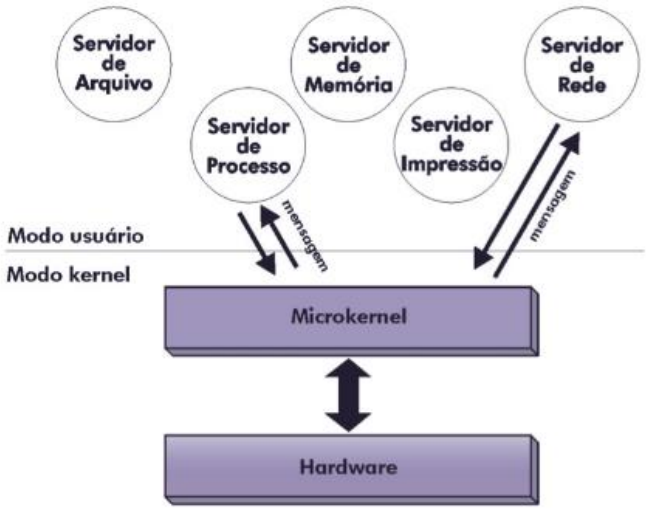
3.2.1. *Kernel*

O *kernel* é o núcleo de um sistema operacional, responsável por executar as chamadas de sistema, tendo como seus principais componentes o gerenciamento do processador, da memória, da entrada e saída, além do sistema de arquivos. Existem algumas maneiras de implementar um *kernel*, dentre elas, as mais utilizadas são a arquitetura *microkernel*, arquitetura monolítica e arquitetura modular (OLIVEIRA; CARISSIMI; TOSCANI, 2010).

3.2.1.1. Arquitetura *microkernel*

Este conceito surgiu no sistema operacional Mach, nos anos 80, na Universidade de Carnegie-Mellon (MACHADO; MAIA, 2013). E de acordo com TANENBAUM e WOODHULL (2008), a arquitetura *microkernel* é uma tendência para os sistemas operacionais modernos, tendo uma ideia de núcleo mínimo (modo *kernel*) e com a maior parte das funções do sistema implementados em processos de usuário (modo usuário), como pode ser visto na Figura 2.

Figura 2 - Arquitetura *microkernel*



Fonte: MACHADO e MAIA (2013)

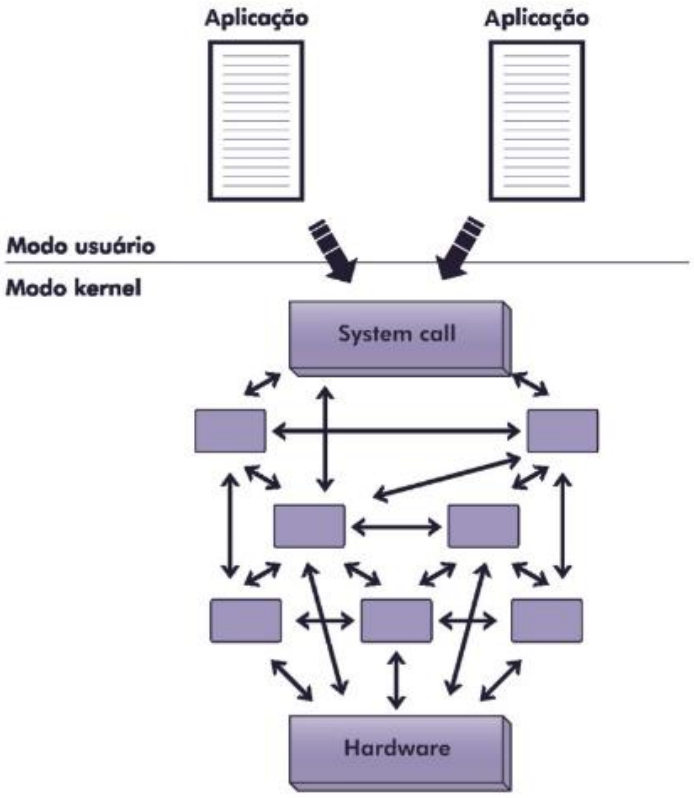
Quando uma aplicação necessite de algum serviço do modo de *kernel*, uma solicitação é feita para o processo responsável. “[...] A aplicação que solicita o serviço é chamada de cliente, enquanto o processo que responde à solicitação é chamado de servidor” (MACHADO; MAIA, 2013).

O *microkernel* possui vantagens como a modularidade, portabilidade de melhor uso de recursos, por utilizar de módulos separados em modo usuário que podem ser adicionados, reiniciados e substituídos em tempo de execução, sem ao menos recompilar o *microkernel* (OLIVEIRA; CARISSIMI; TOSCANI, 2010).

3.2.1.2. Arquitetura monolítica

"A arquitetura monolítica pode ser comparada com uma aplicação formada por vários módulos que são compilados separadamente e depois *linkados*, formando um grande e único programa executável, onde os módulos podem interagir livremente", como pode-se notar na Figura 3 (MACHADO; MAIA, 2013).

Figura 3 - Arquitetura monolítica



Fonte: MACHADO e MAIA (2013)

O projeto MS-DOS e os primeiros sistemas Unix adotaram esta arquitetura por sua simplicidade e bom desempenho (MACHADO; MAIA, 2013).

3.2.1.3. Arquitetura modular

De acordo com OLIVEIRA, CARISSIMI e TOSCANI (2010), a arquitetura modular é uma espécie de arquitetura híbrida, unindo as vantagens da arquitetura *microkernel*

(modularidade, portabilidade e melhor uso de recursos) e da arquitetura monolítica (desempenho). Sua maior diferença com a arquitetura *microkernel* é que os módulos não são inteiramente separados, e podem utilizar da troca de mensagens. Quando um módulo é carregado, ele passa a ser parte do sistema operacional, permitindo melhor comunicação com os módulos do núcleo. Os sistemas operacionais mais utilizados atualmente utilizam desta arquitetura.

3.2.2. Threads

Com a evolução do *hardware*, os *softwares* podem passar a utilizar mais recursos. Pensando nisso, processadores com mais de um núcleo de processamento (cada núcleo é tratado como um processador diferente pelo sistema operacional), são ideais para sistemas multiprogramáveis. Antes da existência de mais de uma unidade central de processamento, os sistemas operacionais eram monoprogramáveis, ou seja, o processador executaria apenas um programa ou processo por vez, permanecendo dedicado para isto. Já nos sistemas multiprogramáveis (tipo de sistema operacional mais utilizado atualmente), muitos programas ou processos podem estar na memória competindo pela utilização do processador (MACHADO; MAIA, 2011).

Atualmente, existem 3 maneiras de desenvolver um sistema multiprogramável de acordo com MACHADO e MAIA (2011), são eles: Processos independentes, onde um processo pode criar outro processo independente, que não possuirá vínculos com seu criador; Subprocessos, onde um processo pode criar um processo filho, que será dependente do pai, e por sua vez pode criar outros processos filhos, de forma hierárquica, em que caso o processo pai deixe de existir, todos os processos filhos subsequentes também deixarão de existir; *Threads*, que são um conceito criado para diminuir o tempo de criação, eliminação e troca de contexto entre processos, onde um processo pode possuir vários *threads*, que são executados de forma independente, mas por estarem no mesmo processo, sua comunicação é mais simples e rápida.

O modelo mais utilizado atualmente é o modelo de *threads*, em função de seus benefícios, como maior capacidade de respostas, compartilhamento de recursos simplificados, economia de recursos, escalabilidade em arquiteturas de multiprocessos. Em um ambiente *multithread*, programas não são associados a processos, mas sim a *threads*, onde cada programa sempre terá ao menos um *thread*. O *thread* atua como uma sub-rotina de um programa, sendo executado de forma assíncrona e concorrente (SILBERSCHATZ; GALVIN; GAGNE, 2015).

Porém, de acordo com SILBERSCHATZ, GALVIN e GAGNE (2015) ao programar em threads, alguns desafios surgem, sendo alguns deles a identificação de tarefas, onde deve-se analisar a aplicação afim de ser separada em tarefas, de forma concorrente, idealmente suportando execução independente e paralela; o equilíbrio, pois com as tarefas identificadas, a programação deve ser feita de modo em que cada tarefa desenvolva o mesmo esforço e mesmo valor que as demais, verificando se a utilização de núcleos/processadores dedicados realmente é necessária; a divisão de dados, pois além de tarefas separadas, os dados também devem ser separados em núcleos/processadores diferentes; a dependência de dados dado que caso um *thread* necessite de dados de outro *thread*, estes devem ser sincronizados de forma a acomodar tal dependência; teste e depuração, uma vez que dividir um programa em tarefas e distribuí-las em *threads* pode dificultar a depuração e o teste, já que diversos caminhos diferentes podem ser tomados de forma simultânea.

3.2.2.1. Concorrência

A concorrência acontece quando processos ou *threads* de execuções paralelas acessam o mesmo recurso ao mesmo tempo, isso gera uma inconsistência no dado gravado. Partes de uma aplicação onde existe compartilhamento de recursos são chamadas de seção crítica. Para evitar esses problemas, existem mecanismos de controle chamados de condições de corrida (MACHADO; MAIA, 2013).

3.2.2.1.1. Mutex

Mutex é a abreviação de *mutual exclusion* (exclusão mútua). É a solução mais simples utilizada para resolver o problema de concorrência. Uma variável do tipo mutex pode ter apenas os valores livre e ocupado, tendo somente as operações de *lock*, que é utilizada pelos processos para solicitar acesso a uma seção crítica, e *unlock*, que é utilizada para informar que o processo não deseja mais utilizar uma seção crítica (OLIVEIRA; CARISSIMI; TOSCANI, 2010).

3.2.2.1.2. Sincronização condicional

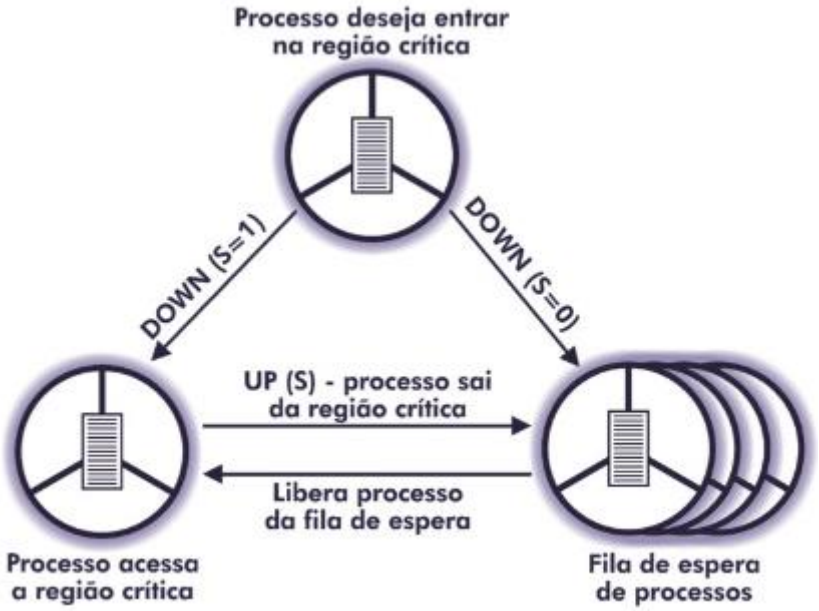
De acordo com MACHADO e MAIA (2013), a sincronização condicional é quando se faz necessário a sincronização de processos para acessar um determinado recurso. O recurso pode não estar disponível em função de uma condição específica, caso o recurso não esteja disponível, o processo deve permanecer bloqueado até que a condição seja satisfeita para liberá-lo. Utilizar a sincronização condicional separadamente não resolve problemas como a

espera ocupada, e por isso, a sincronização condicional deve ser implementada juntamente de outros conceitos como por exemplo: mutex, semáforo e3 monitor.

3.2.2.1.3. Semáforo

Mecanismo criado pelo matemático holandês E. W. Dijkstra em 1965. O semáforo é uma variável que é acessada pelas operações atômicas (funções que não podem ser interrompidas no meio de outra operação) P (do holandês *probern*, que significa testar), também chamada de DOWN, e V (do holandês *verhogen*, que significa incrementar), também chamada de UP (SILBERSCHATZ; GALVIN; GAGNE, 2015). Um semáforo é composto por um número inteiro e uma fila de processos. No momento em que um processo executa uma operação P em um semáforo, o valor inteiro do semáforo é decrementado, e se tal valor for negativo, o processo vai para o final da fila, se existir qualquer processo bloqueado na fila (como por exemplo, esperando um recurso), o primeiro da fila é liberado (OLIVEIRA; CARISSIMI; TOSCANI, 2010). Semáforos também podem ser implementados juntamente com a técnica de exclusão mútua.

Figura 4- Utilização de semáforo binário juntamente do mutex



Fonte: MACHADO e MAIA (2013)

A Figura 4 mostra que o semáforo possuirá um valor inteiro, que indica se o recurso solicitado está sendo acessado por um processo concorrente, tal valor pode ser 0 ou 1. Sempre que um processo precisar entrar em uma seção crítica, ele chamará a função DOWN, caso o semáforo retornar valor 1, o valor será decrementado e o processo solicitante poderá entrar na

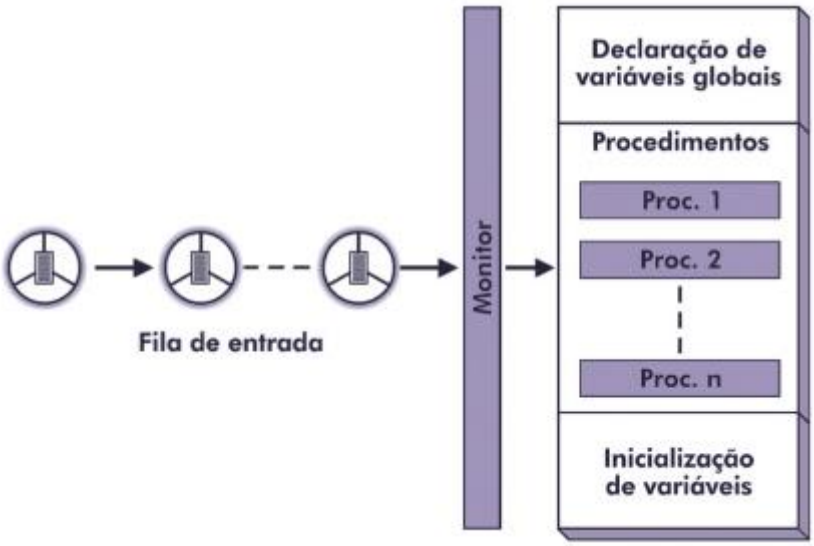
seção crítica, caso o semáforo retornar 0, o processo não poderá entrar na seção crítica e será redirecionado para o fim da lista de espera. Quando o processo sai da seção crítica, a função UP será executada e o valor do semáforo incrementado, e caso exista processos na fila, teste será executado (MACHADO; MAIA, 2013).

MACHADO e MAIA (2013) explicam que semáforos podem ser implementados com sincronização condicional, onde o processo executa a função DOWN no semáforo e permanece em estado de espera até que a operação seja completada. Depois da operação finalizada, a função UP é executada e o semáforo é libera o processo do estado de espera.

3.2.2.1.4. Monitor

Mecanismo criado cientista da computação inglês C. A. R. Hoare. Monitores tornam o desenvolvimento de aplicações concorrentes mais simples, pois são estruturados (diferentemente dos semáforos), tornando-o menos suscetível a *bugs* (OLIVEIRA; CARISSIMI; TOSCANI, 2010).

Figura 5 - Estrutura do monitor



Fonte: MACHADO e MAIA (2013)

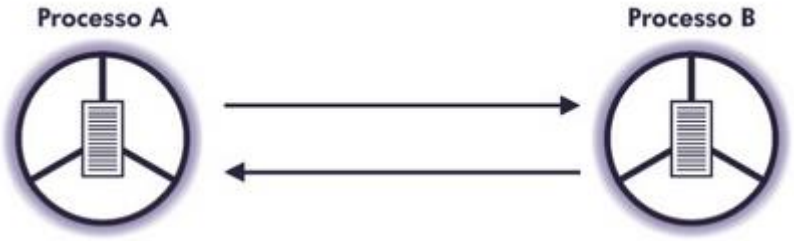
A Figura 5 mostra a estrutura de um monitor, que é formado de procedimentos e variáveis encapsulados em um módulo. A implementação do mutex é automática dentro de um monitor. Quando um processo chama um dos procedimentos, o monitor verifica se existe algum processo executando um procedimento no monitor, caso exista, o processo é direcionado para o fim de uma fila de entrada, caso não exista, o processo executa o procedimento requerido. As variáveis globais são acessíveis apenas no módulo do monitor, e

não são visíveis externamente, e a inicialização de variáveis é executada apenas uma vez, quando o programa que implementa o monitor é inicializado (MACHADO; MAIA, 2013).

3.2.2.1.5. Troca de Mensagens

MACHADO e MAIA (2013) definem a troca de mensagens como "um mecanismo de comunicação e sincronização entre processos". Para que processos se comuniquem, um canal de comunicação deve existir, como pode ser verificado na Figura 6, sendo um *buffer* ou um *link* de rede de computadores. A troca de mensagens é feita por meio de das funções: SEND, que permite o envio da mensagem, e RECEIVE, que permite o recebimento da mensagem, os processos que querem fazer as trocas de mensagens necessitam possuir os dois métodos.

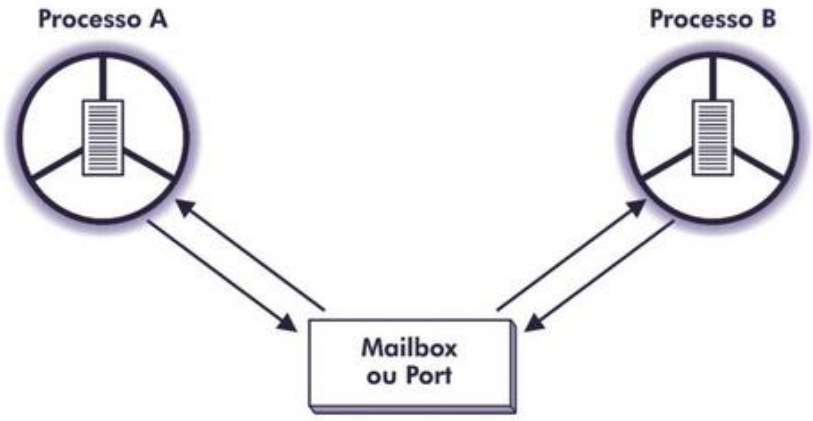
Figura 6 - Comunicação direta



Fonte: MACHADO e MAIA (2013)

De acordo com MACHADO e MAIA (2013), uma das maneiras de implementar a troca de mensagens, é pela comunicação direta, em que um processo deve endereçar explicitamente o nome do processo receptor ou transmissor ao enviar ou receber uma mensagem. Isto faz com que ao mudar a identificação do processo, o código deve ser alterado e recompilado.

Figura 7 - Comunicação indireta



Fonte: MACHADO e MAIA (2013)

Outra maneira de implementar a troca de mensagens é pela comunicação indireta, onde as mensagens são guardadas em um *buffer* chamado *mailbox* ou *port*, o processo transmissor envia a mensagem e o receptor retira a mensagem, como pode ser verificado na imagem 7. Vários processos podem estar associados a *mailbox*, e ao invés de passar o nome do processo, na comunicação indireta, são passados os nomes das *mailboxes* (MACHADO; MAIA, 2013).

3.2.2.2. *Deadlocks*

Em sistemas multiprogramáveis, múltiplos processos podem tentar acessar um recurso ao mesmo tempo, para não haver inconsistência de dados, porém, quando um processo solicita o uso de um recurso que já está sendo usado por outro, o processo entra em estado de espera. Existem casos em que o processo não consegue sair do estado de espera, pois já existem outros processos na fila esperando pelo acesso dos dados, essa situação é chamada de *deadlock* (SILBERSCHATZ; GALVIN; GAGNE, 2015).

De acordo com SILBERSCHATZ, GALVIN e GAGNE (2015) um *deadlock* pode acontecer em razão de 4 motivos, são eles:

- Exclusão mútua, apenas um processo pode acessar o recurso por vez, se já estiver sendo utilizado por outro, o processo deverá ser atrasado até que o recurso seja liberado;

- Retenção e espera, um processo está com a posse de um recurso, mas esperando outros recursos de outro processo;

- Inexistência de preempção, é impossível solicitar a outro processo que libere o recurso, pois o recurso só poderá ser liberado voluntariamente;

- Espera circular, o processo 1 está esperando por um recurso do processo 2 que está esperando um recurso do processo 3 que está esperando um recurso retido pelo processo 1.

Um *deadlock* só acontece se as 4 condições forem atendidas. A maioria dos sistemas operacionais atuais não se preocupam com *deadlocks*, ficando a cargo dos desenvolvedores das aplicações tal manipulação.

3.2.3. Sistemas de arquivo

Segundo TANENBAUM (2003), qualquer aplicação necessita de armazenar e recuperar dados, e enquanto um processo está em execução, pode-se armazenar dados de forma limitada dentro de seu espaço de armazenamento. Em razão de tal limitação, aplicações mais robustas necessitam de maior espaço para armazenamento, recorrendo ao armazenamento em disco. Tais dados são armazenados em unidades chamadas arquivos, e não são afetados pelo ciclo de vida de um processo. A parte do sistema operacional que trata de arquivos e sua forma de

armazenamento é chamada de sistemas de arquivos. Sistemas de arquivos podem ser utilizados em qualquer dispositivo de memória não volátil (memórias que não necessitam de energia para armazenar dados).

3.2.3.1. Partições e ponto de montagem

Ao instalar um sistema operacional, é necessário criar ou modificar partições já existentes, que é a divisão de um disco físico em discos lógicos. Cada disco lógico pode ter sistemas de arquivos diferentes, tais sistemas de arquivos também podem ter diferentes formas de organizar arquivos, diretórios, controlar espaço livre e etc. O ponto de montagem é um diretório raiz, que a partir deste é possível definir outros diretórios e montar outros sistemas de arquivos. O *kernel* Linux suporta diferentes tipos de sistemas de arquivos, como Ext, Ext2, Ext3, ReiserFS, VFAT, PROCFS, SMB, NCP, ISO9660, SysV, HPFS, UFS, etc. O sistema nativo do Linux é o Ext2, o sistema nativo do Windows é o NTFS, mas também suporta FAT, FAT32, HPFS (OLIVEIRA; CARISSIMI; TOSCANI, 2010).

3.2.3.2. Ext2

De acordo com OLIVEIRA; CARISSIMI; TOSCANI (2010), o Ext2 é utilizado desde 1994 na maioria das distribuições Linux por padrão. Com sua estrutura básica sendo o bloco, o Ext2 pode armazenar dados de controle sobre o sistema de arquivos. Ao criar uma partição Ext2, ela deve ser dividida em mais uma partição chamada *boot loader*, reservada para o *boot* da máquina, com as informações do carregador do sistema operacional, no Linux os mais famosos são o Grub (*GRand Unified Bootloader*) e o LILO (LInux *LOader*).

3.2.3.3. NTFS

OLIVEIRA, CARISSIMI e TOSCANI (2010) também dizem que o NTFS é um sistema de arquivos próprio do Windows, projetado para cobrir as necessidades críticas em ambientes corporativos, como segurança, suporte para sistemas POSIX, facilidade de recuperação de dados e tolerância a falhas, suporte a grandes discos e arquivos, facilidade de indexação e fluxos de dados múltiplos.

3.2.4. Gerenciamento de Memória

TANEMBAUM (2008) diz que a maior parte dos computadores utiliza uma hierarquia de memórias, que é composta por uma pouca quantidade de memória cache (de até 12 MB), volátil e de custo alto, uma grande memória principal (de até 16 GB), também conhecida como memória RAM, também volátil e de custo médio, e, uma memória secundária, não

volátil, em disco, com velocidade e custo baixos. O gerenciamento dessas memórias é o que mantém o controle do que está em uso, alocando memória para os processos necessitados e liberando dos processos que foram finalizados, além de trocar os processos para diferentes tipos de memória (citadas acima). Sistemas operacionais podem implementar o gerenciamento de memória em duas maneiras, alternando processos entre memória principal e disco durante a execução do processo, e não alterando.

Existem alguns algoritmos para alocação de memória, seriam eles: *first fit*, o mais rápido dos algoritmos, que percorre a lista de segmentos até encontrar um espaço grande o suficiente; *next fit*, funcionando da mesma maneira que o *first fit*, mas com a diferença de que na próxima vez que pesquisar um espaço de memória, inicia-se de onde parou a pesquisa anteriormente; *best fit*, percorrendo a lista inteira, este algoritmo busca pelo espaço mais adequado; *worst fit*, sendo o mais lento dos algoritmos, busca sempre o maior espaço disponível TANEMBAUM (2008).

4. Análise de sistemas operacionais

Nesta pesquisa, foram avaliados e comparados os sistemas operacionais Redox (escrito em Rust) e Minix (escrito em C). Redox foi escolhido por ser um sistema operacional de propósito geral, que possui o desenvolvimento mais avançado quando comparados a outros sistemas em Rust. Minix foi escolhido por também ser um sistema operacional de propósito geral, e possuir a mesma arquitetura de núcleo que o Redox.

Os pontos que foram avaliados em cada sistema operacional foram escolhidos por conta de estudos de casos de sistemas operacionais retratados por DEITEL, DEITEL e CHOFFNES (2005), sendo desbravados pelas documentações dos sistemas avaliados.

4.1. Arquitetura de Núcleo

Em sua documentação, os criadores de Redox admitem que tiveram uma inspiração na arquitetura de Minix. Utilizando a arquitetura *microkernel*, utiliza o mínimo de abstração no *kernel-space*, tendo como ênfase o *user-space*, diferentemente de *kernels* monolíticos (REDOX, 2018?). Minix utiliza a mesma arquitetura *microkernel*.

Ambos os sistemas operacionais que serão comparados neste trabalho (Minix e Redox) utilizam desta arquitetura, onde com tal divisão, as partes se tornam menores e mais gerenciáveis, além do mais, como cada parte é executada individualmente e em modo usuário, os módulos não possuem acesso direto ao *hardware*, e se apenas um módulo tiver algum problema, os outros módulos não são afetados (TANENBAUM; WOODHULL, 2008).

Apesar de todos os benefícios da arquitetura *microkernel*, na prática, existem vários problemas, como desempenho, que é prejudicado pela mudança no modo de acesso a todo momento, além de operações de entrada/saída, que necessitam de acesso de forma mais direta ao *hardware* (MACHADO; MAIA, 2013).

4.2. Gerenciamento de Processo

Por possuir um *microkernel*, o gerenciamento de processos no Minix se localiza na camada de Processos de Servidor, estando separado do núcleo. Seu gerenciador de processos executam as chamadas relacionadas a iniciar ou interromper a execução de processos, além de chamadas que podem alterar o estado de execução de um determinado processo. Seu gerenciador de processos também é seu gerenciador de memória, dificultando na separação de responsabilidades (TANEMBAUM, 2008).

Diferentemente do Minix, Redox assume o gerenciamento de processos e *threads* como essenciais, e ao invés de possuir uma camada separada mas acoplada no núcleo (como no

Minix), tal gerenciamento é feito diretamente em seu núcleo, juntamente com o gerenciamento de recursos e chamadas de sistema, pelo gerenciamento de processo ser junto ao núcleo e não em módulos, o desempenho é maior, porém, a responsabilidade do núcleo aumenta e a modularidade é perdida (REDOX, 2018?).

4.3. Gerenciamento de Memória

Redox usa de uma biblioteca feita em Rust chamada *ralloc*, porém, sua documentação infelizmente não oferece informações o suficiente sobre seu funcionamento interno, apenas como usá-la.

O Minix, como citado anteriormente, possui um componente de gerenciamento de memória sendo o mesmo de gerenciamento de processos. Tais decisões arquiteturais foram escolhidas para manter o sistema fácil de ser compreendido, além de tornar fácil a portabilidade para outros tipos de *hardware* (TANEMBAUM, 2008), mas por outro lado, aumenta a responsabilidade do módulo de gerenciamento de memória/processos.

4.4. Sistemas de Arquivos

Segundo TANEMBAUM (2008), o sistema de arquivos no Minix é um programa executado em *user-space* para ler e escrever arquivos, que recebe instruções dos processos de usuário. Sua arquitetura usa de árvores com seus respectivos nós para armazenar os arquivos.

Redox não utiliza de sistema de arquivos. O meio de armazenamento é baseado no 9P do sistema operacional Plan 9 da Bell Labs, usando o conceito de *"Everything is a URL"* (tudo é uma URL) ao invés do tradicional *"Everything is a File"* (tudo é um arquivo), onde ao invés de utilizar uma árvore de arquivos, o Redox utiliza de protocolos para distinguir os recursos. Por exemplo, um dispositivo USB não se encaixa logicamente no conceito de arquivo; no Redox, dispositivos USB são reconhecidos como EHCI, baseado em um esquema de protocolos. Arquivos comuns são reconhecidos pelo protocolo chamado *file* (REDOX, 2018?).

Sua documentação afirma que utilizando esta estrutura, aplicações podem utilizar da API do sistema operacional para comunicarem entre si, criando um ecossistema de comunicação primitiva e facilitando a implementação de comunicação das aplicações. Por ser um conceito disponível apenas no Redox, não é possível compará-lo diretamente com o conceito de *"Everything is a File"* e entender seus pontos positivos e negativos apenas mediante a documentação do próprio Redox (REDOX, 2018?).

4.5. Gerenciamento de Entrada/Saída

Redox utiliza *Schemes* para possuir uma interface unificada de Entrada/Saída. *Schemes* funcionam juntamente com as URL's (modo de armazenamento de Rust), que são abertas para acessar um recurso. Tais *Schemes* podem ser definidos tanto no *kernel-space* quando no *user-space* (assim como seus *drivers*), mas preferencialmente são definidos no *user-space* por questões de segurança. Redox ainda não possui detalhes em sua documentação sobre como funcionam os *drivers* (REDOX, 2018?).

Os *drivers* no Minix também são executados em *user-space*, para que não tenham acesso às estruturas de dados do núcleo. Sua maior vantagem é que se por algum motivo, um *driver* tenha defeito, o sistema inteiro não falhará, mas outra vantagem, é que qualquer *driver* pode ser parado, executado, reiniciado ou substituído em tempo de execução, além de que um *driver* não necessário na inicialização do sistema pode ser executado posteriormente (TANEMBAUM, 2008), porém, isso diminui seu desempenho por conta da mudança de acesso (*user-space* e *kernel-space*).

4.6. Comunicação Interprocessos

No Minix, a comunicação interprocessos se dá por chamadas de funções de bibliotecas em C, seriam elas *send* para enviar uma mensagem, *receive* para receber uma mensagem e *sendrec* para enviar uma mensagem e aguardar a resposta. Minix utiliza de algumas técnicas de concorrência já tratadas neste trabalho, como exclusão múltipla, semáforos, mutex e monitores (TANEMBAUM, 2008).

No Redox, a responsabilidade de comunicação interprocessos é em *user-space*, mas sua parte básica é em *kernel-space*, e sua documentação não oferece informações sobre seu funcionamento. Rust oferece diversos recursos (funções e bibliotecas implementadas nativamente, como a biblioteca *thread*) para facilitar a implementação de soluções como mutex, exclusões múltiplas e comunicação interprocessos, a documentação do Redox, que ainda é muito pobre, não fornece informações como as técnicas utilizadas para contornar os problemas, diferentemente do Minix.

4.7. Escalabilidade

Ambos os sistemas, por possuírem arquitetura *microkernel*, são altamente escalonáveis, pois várias camadas podem ser acopladas no núcleo, e por terem tais partes separadas, podem ser substituídas, ou adicionadas até mesmo em tempo de execução, e sem recompilar o núcleo (MACHADO; MAIA, 2013).

4.8. Segurança

De acordo com a documentação do Redox, sistemas com arquitetura *microkernel* são indiscutivelmente mais seguros do que sistemas que adotam a arquitetura monolítica. Por possuir um *design* minimalista e componentes que podem ser substituídos, como serviços e *drivers*, tais componentes não têm acesso ao *kernel-space*, diminuindo drasticamente as chances de alteração de funções do núcleo (REDOX, 2018?).

Em uma arquitetura monolítica, *drivers* e outros componentes de um sistema operacional podem agir livremente, sem nenhuma restrição, já que não são executados em um módulo separado (MACHADO; MAIA, 2013).

5. Análise de linguagens

Também foi medido o desempenho das linguagens (Rust e C) quanto ao uso de memória e tempo de execução. Para a execução dos testes, foram implementados um algoritmo de multiplicação de matrizes e um algoritmo de codificação e decodificação de base64, escritas tanto em C quanto em Rust, sendo medidos por um *script* externo escrito em Python, disponível no apêndice A para não favorecer alguma das linguagens na medição. Os algoritmos testados são *Open Source*, com o código no GitHub (https://github.com/kostya/benchmarks), com apenas algumas alterações para a mudança de tamanho (de matrizes em Matmul e de *strings* em Base64), tais algoritmos foram referenciados em RUST-FAQ (2018?).

Devido ao Redox ser um sistema ainda muito jovem, ainda não existe um interpretador Python, e devido a isso, os testes não puderam ser executados em ambos os sistemas (Redox e Minix), portanto, os testes foram feitos para avaliar apenas as linguagens, pois o desempenho de cada algoritmo influencia diretamente na implementação de um sistema operacional.

A máquina utilizada para os testes possui as seguintes configurações:

- Sistema operacional Arch Linux com interface xfce, com arquitetura x64.

- *Hardware*: Processador Intel core i7 - 4510U, 8 gb de memória RAM, 1 tb de memória em HD.

5.1. Algoritmo Matmul

O algoritmo de multiplicação de matrizes foi escolhido por poder ser utilizado na definição de resolução de tela em um sistema operacional.

De acordo com TOSCANI e VELOSO (2012), o algoritmo de multiplicação de matrizes é um problema que ilustra a estratégia de programação dinâmica, que consiste em multiplicar um número n de matrizes, calculando o produto M: M_1 x M_2 x ... x M_n, onde cada matriz M_i tem b_{i-1} linhas e b_i colunas, para i = 1, ..., n.

Para medir o desempenho das linguagens, o algoritmo de Matmul deste trabalho foi executado com duas matrizes, iniciando-se por matrizes 100 x 100, e aumentando de 100 em 100, até 1000 x 1000. Aplicados em C e em Rust, o algoritmo de Matmul foi executado 5 vezes para cada tamanho de matriz diferente, e tirado a média para geração dos gráficos.

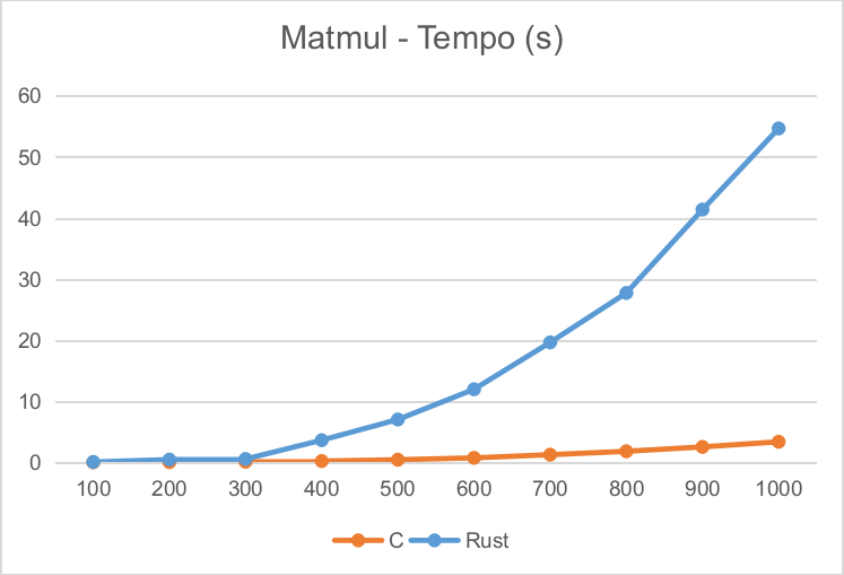
Nos gráficos abaixo, o eixo X representa o tamanho da matriz (ex: 100 x 100). Nos gráficos relacionados ao tempo, o eixo Y representa o tempo em segundos, já nos gráficos relacionados à memória em Kb, o eixo Y representa a memória utilizada. Suas implementações encontram-se nos apêndices B (linguagem C) e C (linguagem Rust).

Tabela 1 - Tempo de execução do algoritmo de Matmul em Rust e C

	Matmul - Tempo (s)	
	Rust	**C**
100	0.060975933	0.005464363
200	0.425793648	0.031774521
300	0.451550388	0.096753359
400	3.413755083	0.217971897
500	6.587900782	0.432188797
600	11.22921777	0.74058671
700	18.39579639	1.256493759
800	25.93174071	1.813234234
900	38.87247143	2.532265711
1000	51.31326752	3.376733208

Fonte: O autor (2019)

Figura 8 - Tempo de execução do algoritmo de Matmul em Rust e C



Fonte: O autor (2019)

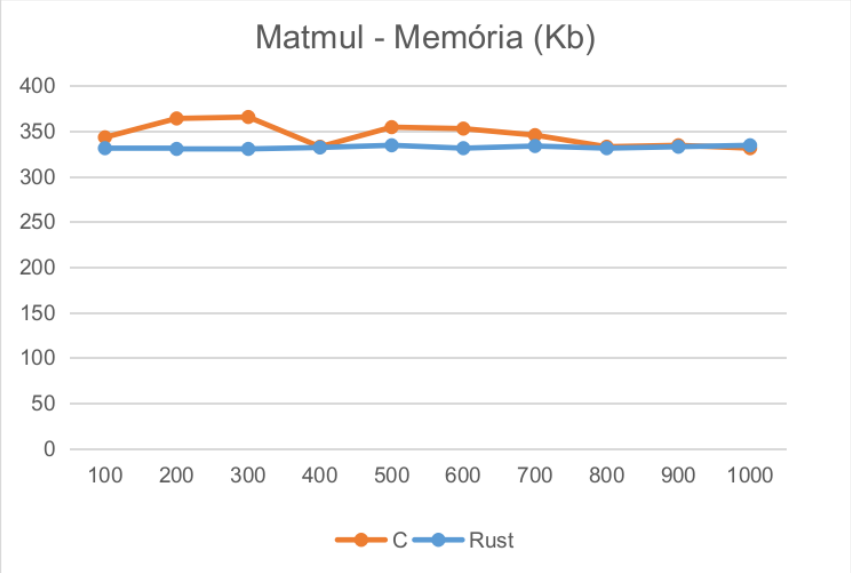
Tabela 2 - Gasto de memória do algoritmo de Matmul em Rust e C

	Matmul - Memória (Kb)	
	Rust	**C**
100	331.2	343.2

200	330.4	364
300	330.4	365.6
400	332	332.8
500	334.4	354.4
600	331.2	352.8
700	333.6	345.6
800	331.2	332.8
900	332.8	334.4
1000	334.4	331.2

Fonte: O autor (2019)

Figura 9 - Gasto de memória do algoritmo de Matmul em Rust e C



Fonte: O autor (2019)

Pode-se perceber pelos resultados da Tabela 2 e Figura 9, que o uso de memória em C varia entre 330 a 334, em Rust, o uso de memória varia entre 331 a 365. A variação de memória é bem pequena, mas pode-se notar um aumento do uso de memória nas matrizes maiores em Rust, porém a variação é bem menor.

Com relação ao tempo, verificando a Tabela 1 e Figura 8, C foi muito superior, variando entre 0.005 a 3.37 segundos, enquanto Rust varia entre 0.06 a 51.37 segundos.

5.2. Algoritmo Base64

O algoritmo de codificação e decodificação em Base64 foi escolhido por poder ser utilizado na parte de segurança em um sistema operacional.

De acordo com SINGH e SUPRIYA (2013), o algoritmo Base64 é geralmente usado para codificar dados binários em texto ASCII. Representado por um alfabeto baseado em 64 caracteres:

Tabela 3 - Correlação valor x caracter

Valor	Caracter
0 ... 25	‘A’ ... ‘Z’
26 ... 51	‘a’ ... ‘z’
52 ... 61	‘0’ ... ‘9’
62	‘+’
63	‘/’
(Pad)	‘=’

Fonte: SINGH e SUPRIYA (2013)

Para medir o desempenho das linguagens, o algoritmo de codificação e decodificação Base64 deste trabalho foi executado com palavras de tamanho 10^5 *bits* até 10^9 *bits*, pois C não suporta 10^{10}. Aplicados em C e em Rust, o algoritmo Base64 também foi executado 5 vezes para cada tamanho de palavra diferente, e tirado a média para geração dos gráficos.

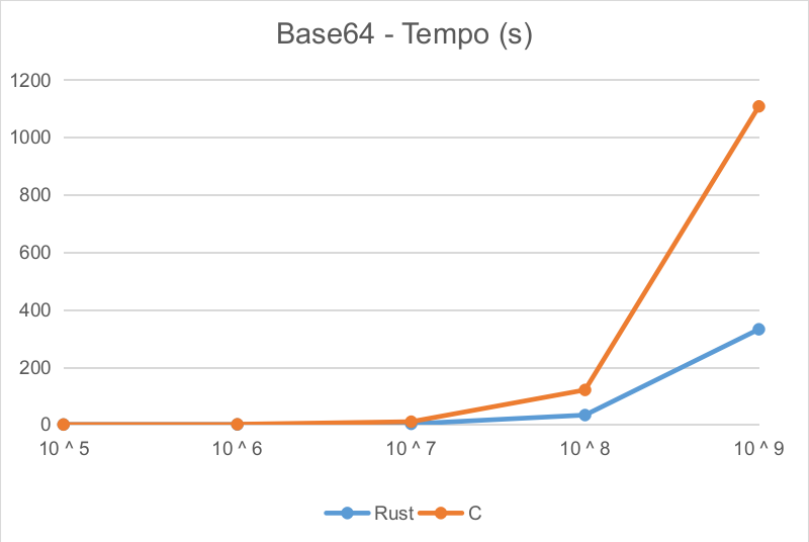
Nos gráficos abaixo, o eixo X representa o tamanho da palavra em *bits*. Nos gráficos relacionados ao tempo, o eixo Y representa o tempo em segundos, já nos gráficos relacionados à memória em Kb, o eixo Y representa a memória utilizada. Suas implementações encontram-se nos apêndices D (linguagem C) e E (linguagem Rust).

Tabela 4 - Tempo de execução do algoritmo de codificação e decodificação Base64 em Rust e C

	Matmul - Tempo (s)	
	Rust	**C**
10^5	0.027418137	0.072684765
10^6	0.238914967	0.690730381
10^7	2.389983463	7.473716021
10^8	32.89690599	87.50068111
10^9	331.5864094	776.0505461

Fonte: O autor (2019)

Figura 10 - Tempo de execução do algoritmo de codificação e decodificação Base64 em Rust e C



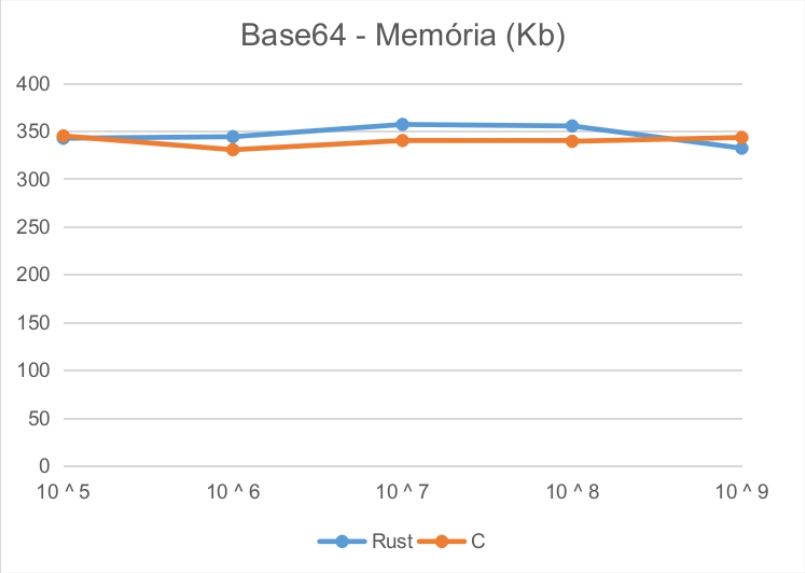
Fonte: O autor (2019)

Tabela 5 - Gasto de memória do algoritmo de codificação e decodificação Base64 em Rust e C

	Matmul - Memória (Kb)	
	Rust	**C**
10^5	342.4	344.8
10^6	344	330.4
10^7	356.8	340
10^8	355.2	339.2
10^9	332	343.2

Fonte: O autor (2019)

Figura 11- Gasto de memória do algoritmo de codificação e decodificação Base64 em Rust e C



Fonte: O autor (2019)

Com os resultados da Tabela 5 e Figura 11, percebe-se que o uso de memória em C varia apenas entre 330 e 343, enquanto em Rust, a quantidade de memória gasta varia entre 332 e 356.

Levando em consideração o tempo, Rust é mais eficiente, como pode ser verificado na Tabela 4 e Figura 10, variando entre 0.02 e 331.58 segundos, enquanto C, varia entre 0.072 e 776.05 segundos, de forma exponencial. Foi utilizado em Rust a biblioteca base64, que pode ter colaborado com a melhora de desempenho.

6. Conclusão

Sistemas operacionais escritos em Rust ainda são muito recentes (2015), devido a própria linguagem ser recente (2010). Sua documentação ainda é muito primitiva, não possuindo diversos pontos essenciais de explicação sobre seu design e arquitetura, dificultando a análise deste trabalho.

Apesar das dificuldades, Redox mostra grande maturidade nos principais requisitos funcionais e não-funcionais de um sistema operacional. Como requisitos funcionais pode-se citar a arquitetura de núcleo *microkernel*, gerenciamento de memória e processos em módulos separados, um sistema de arquivos não convencional baseado em URL's e interface unificada para entrada/saída por meio de *schemes*. Em função da arquitetura *microkernel*, vários requisitos-não funcionais são contemplados, como a alta portabilidade e escalabilidade, já que os módulos do *kernel* podem ser acoplados e desacoplados em tempo de execução e sem recompilação do *kernel*, além da segurança por conta da separação de módulos em *user-space*. Porém, possui menor desempenho por conta da mudança do modo de *kernel* a todo momento.

A similaridade de arquitetura de Redox com o Minix é visível, pois em sua documentação, os criadores de Redox admitem inspiração no Minix. As diferenças são poucas, Minix utiliza o sistema de arquivos convencional e não possui interface unificada de entrada/saída. Minix é muito bem documentado, com um livro escrito por seu criador, contendo os conceitos de um sistema operacional, peculiaridades do Minix e até mesmo partes de seu código.

Em relação às linguagens, C e Rust obtiveram uso de memória RAM muito similares, variando muito pouco a quantidade de memória durante a execução dos programas. Em relação ao tempo de execução, C foi melhor no algoritmo de Matmul, mas teve desempenho inferior no algoritmo de codificação e decodificação Base64, além de não conseguir suportar uma palavra de tamanho 10^{10} *bits*. Rust, por outro lado, teve desempenho inferior no algoritmo de Matmul, mas obteve grande sucesso no algoritmo de codificação e decodificação Base64, variando muito pouco o tempo de execução entre os tamanhos da palavra.

Pode-se concluir que com os resultados acima, Rust consegue confrontar C, possuindo resultados parecidos no contexto de desempenho de linguagem, e provando que um sistema operacional em Rust pode ter os mesmos requisitos de um sistema operacional em C.

7. Referências Bibliográficas

CVE Details. **Linux Kernel Vulnerability Statistics**. Disponível em: <https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33>. Acesso em: 27 out. 2019.

DAMAS, Luís. **Linguagem C**. 10ª ed. Rio de Janeiro: LTC, 2016. 374 p.

DEITEL, Harvey M.; DEITEL, Paul J.; CHOFFNES, David R. **Sistemas Operacionais**. São Paulo: Pearson Prentice Hall, 2005. 760 p.

KLABNIK, Steve; NICHOLS, Carol. **The Rust Programming Language**. San Francisco: [s.n.], 2018. 427 p.

KASHITSYNM, Dmitry. **Why Rust?**. 2018. Disponível em: <https://medium.com/paritytech/why-rust-846fd3320d3f>. Acesso em: 11 ago. 2019.

LEVY, Amit; et al. **The Case for Writing a Kernel in Rust**. APSys, 2017. 7 p.

MACHADO, Francis Berenger; MAIA, Luiz Paulo. **Arquitetura de sistemas operacionais**. 5ª ed. Rio de Janeiro: LTC, 2013. 250 p.

MACHADO, Francis Berenger; MAIA, Luiz Paulo. **Fundamentos de sistemas operacionais**. 1ª ed. Rio de Janeiro: LTC, 2011. 112 p.

NILSSON, Frederick; ADOLFSSON, Niklas. **A Rust-based Runtime for the Internet of Things**. Chalmers University of Technology, 2017. 93 p.

OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. **Sistemas operacionais**. 4ª ed. Porto Alegre: ARTMED, 2010. 374 p.

REDOX. **The Redox Operating System**. Disponível em: <https://doc.redox-os.org/book/>. Acesso em: 16 out. 2018.

RUST. **Frequently Asked Questions**. Disponível em: <https://www.rust-lang.org/en-US/faq.html>. Acesso em: 13 nov. 2018.

SILBERSCHATZ, Abraham; GALVIN, Peter Baer; GAGNE, Greg. **Fundamentos de Sistemas Operacionais**. 9. ed. Rio de Janeiro: LTC, 2015.

SINGH, Gurpreet; SUPRIYA. **Modified Vigenere Encryption Algorithm and its Hybrid Implementation with Base64 and AES**. Computer Science and Engeneering Department of Sri Guru Granth Sahib World University, 2013. 6p.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. 2. ed. São Paulo: Prentice Hall, 2003. 695 p.

TANENBAUM, Andrew S.; WOODHULL, Albert S. **Sistemas operacionais: projeto e implementação**. 3. ed. Porto Alegre: Bookman, 2008. 990 p.

TOCK. **Getting Started**. Disponível em: <https://www.tockos.org/documentation/getting-started>. Acesso em: 16 out. 2018.

TOSCANI, Laira Vieira; VELOSO, Paulo A. S. **Complexidade de Algoritmos**. 3. ed. Porto Alegre: Bookman, 2012. 262 p.

TUCKER, Allen B.; NOONAN, Robert E. **Linguagens de Programação**: Princípios e Paradigmas. 2. ed. Porto Alegre: McGraw-Hill, 2008. 391 p.

VAREJÃO, Flávio Miguel. **Linguagem de programação: conceitos e técnicas**. Rio de Janeiro: Elsevier, 2004. 334 p.

8. Apêndices

APÊNDICE A - SCRIPT EM PYTHON PARA TESTE DOS BINÁRIOS

```
import os
import psutil
import subprocess
import time

processo = psutil.Process(os.getpid())

memoria1 = processo.memory_info()
inicio = time.time()

subprocess.call(['/home/base64-c/base64'])
#Alterar o caminho para cada teste

memoria2 = processo.memory_info()
fim = time.time()

print("Diferença de memória:", (memoria2.rss - memoria1.rss) / 1024)
print("Tempo gasto:", fim - inicio)
```

APÊNDICE B - MATMUL EM C

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

double **mm_init(int n) {
  double **m;
  int i;
  m = (double**)malloc(n * sizeof(void*));
  for (i = 0; i < n; ++i)
    m[i] = calloc(n, sizeof(double));
  return m;
}

void mm_destroy(int n, double **m) {
  int i;
  for (i = 0; i < n; ++i) free(m[i]);
  free(m);
}

double **mm_gen(int n) {
  double **m, tmp = 1. / n / n;
  int i, j;
  m = mm_init(n);
  for (i = 0; i < n; ++i)
    for (j = 0; j < n; ++j)
      m[i][j] = tmp * (i - j) * (i + j);
  return m;
}

double **mm_mul(int n, double *const *a, double *const *b) {
  int i, j, k;
  double **m, **c;
  m = mm_init(n); c = mm_init(n);
  for (i = 0; i < n; ++i) // transpose
    for (j = 0; j < n; ++j)
      c[i][j] = b[j][i];
  for (i = 0; i < n; ++i) {
    double *p = a[i], *q = m[i];
    for (j = 0; j < n; ++j) {
      double t = 0.0, *r = c[j];
      for (k = 0; k < n; ++k)
        t += p[k] * r[k];
      q[j] = t;
    }
  }
  mm_destroy(n, c);
  return m;
}
```

```
int main(int argc, char *argv[]) {
  int n = 4 * pow(10, 2);
  printf("%d\n", n);
  double **a, **b, **m;
  if (argc > 1) n = atoi(argv[1]);
  n = (n/2) * 2;
  a = mm_gen(n); b = mm_gen(n);
  m = mm_mul(n, a, b);
  fprintf(stderr, "%lf\n", m[n/2][n/2]);
  mm_destroy(n, a); mm_destroy(n, b); mm_destroy(n, m);
  return 0;
}
```

APÊNDICE C - MATMUL EM RUST

```
fn new_mat(x: usize, y: usize) -> Vec<Vec<f64>> {
    vec![vec![0f64; y]; x]
}

fn mat_gen(n: usize) -> Vec<Vec<f64>> {
    let mut m = new_mat(n, n);
    let tmp = 1f64 / (n as f64) / (n as f64);

    for i in 0 .. n {
        for j in 0 .. n {
            m[i][j] = tmp * (i as f64 - j as f64) * (i as f64 + j as
f64);
        }
    }
    m
}

#[inline(never)]
fn mat_mul(a: &[Vec<f64>], b: &[Vec<f64>]) -> Vec<Vec<f64>> {
    let m = a.len();
    let n = a[0].len();
    let p = b[0].len();

    let mut b2 = new_mat(n, p);
    for i in 0 .. n {
        for j in 0 .. p {
            b2[j][i] = b[i][j];
        }
    }

    let mut c = new_mat(m, p);

    for (i, ci) in c.iter_mut().enumerate() {
        for (cij, b2j) in ci.iter_mut().zip(&b2) {
            *cij = a[i].iter().zip(b2j).map(|(&x, y)| x * y).sum();
        }
    }

    c
}

fn main() {
      let size = 9 * i32::pow(10, 2);
      println!("{}", size);
    let n = std::env::args()
            .nth(1)
            .unwrap_or(size.to_string().into())
            .parse::<usize>()
```

```
            .unwrap() / 2 * 2;

    let a = mat_gen(n);
    let b = mat_gen(n);
    let c = mat_mul(&a, &b);

    println!("{}", c[n / 2][n / 2]);
}
```

APÊNDICE D - BASE64 EM C

```
#include "stdlib.h"
#include "stdio.h"
#include "time.h"
#include <stdint.h>
#include <math.h>

typedef unsigned int uint;
const char* chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
static char decode_table[256];

long int encode_size(long int size) {
  return (long int)(size * 4 / 3.0) + 6;
}

long int decode_size(long int size) {
  return (long int)(size * 3 / 4.0) + 6;
}

void init_decode_table() {
  for (int i = 0; i < 256; i++) {
    char ch = (char)i;
    char code = -1;
    if (ch >= 'A' && ch <= 'Z') code = ch - 0x41;
    if (ch >= 'a' && ch <= 'z') code = ch - 0x47;
    if (ch >= '0' && ch <= '9') code = ch + 0x04;
    if (ch == '+' || ch == '-') code = 0x3E;
    if (ch == '/' || ch == '_') code = 0x3F;
    decode_table[i] = code;
  }
}

#define next_char(x) char x = decode_table[(unsigned char)*str++];
if (x < 0) return 1;

int decode(long int size, const char* str, long int* out_size,
char** output) {
  *output = (char*) malloc( decode_size(size) );
  char *out = *output;
  while (size > 0 && (str[size - 1] == '\n' || str[size - 1] == '\r'
|| str[size - 1] == '=')) size--;
  const char* ends = str + size - 4;
  while (1) {
    if (str > ends) break;
    while (*str == '\n' || *str == '\r') str++;

    if (str > ends) break;
    next_char(a); next_char(b); next_char(c); next_char(d);
```

```
    *out++ = (char)(a << 2 | b >> 4);
    *out++ = (char)(b << 4 | c >> 2);
    *out++ = (char)(c << 6 | d >> 0);
  }

  int mod = (str - ends) % 4;
  if (mod == 2) {
    next_char(a); next_char(b);
    *out++ = (char)(a << 2 | b >> 4);
  } else if (mod == 3) {
    next_char(a); next_char(b); next_char(c);
    *out++ = (char)(a << 2 | b >> 4);
    *out++ = (char)(b << 4 | c >> 2);
  }

  *out = '\0';
  *out_size = out - *output;
  return 0;
}

void encode(long int size, const char* str, long int* out_size,
char** output) {
  *output = (char*) malloc( encode_size(size) );
  char *out = *output;
  const char* ends = str + (size - size % 3);
  uint n;
  while (str != ends) {
    uint64_t n = __builtin_bswap64(*(uint64_t*)str);
    *out++ = chars[(n >> 26) & 63];
    *out++ = chars[(n >> 20) & 63];
    *out++ = chars[(n >> 14) & 63];
    *out++ = chars[(n >> 8) & 63];
    str += 3;
  }
  long int pd = size % 3;
  if  (pd == 1) {
    n = (uint)*str << 16;
    *out++ = chars[(n >> 18) & 63];
    *out++ = chars[(n >> 12) & 63];
    *out++ = '=';
    *out++ = '=';
  } else if (pd == 2) {
    n = (uint)*str++ << 16;
    n |= (uint)*str << 8;
    *out++ = chars[(n >> 18) & 63];
    *out++ = chars[(n >> 12) & 63];
    *out++ = chars[(n >> 6) & 63];
    *out++ = '=';
  }
```

```
  *out = '\0';
  *out_size = out - *output;
}

int main() {
  init_decode_table();

  const long int STR_SIZE = pow(10, 6);
  const int TRIES = 100;

  char *str2;
  long int str2_size;
  char *str3;
  long int str3_size;

  char *str = (char*) malloc(STR_SIZE + 1);
  for (long int i = 0; i < STR_SIZE; i++) { str[i] = 'a'; }
  str[STR_SIZE] = '\0';

  long int s = 0;
  clock_t t = clock();
  for (int i = 0; i < TRIES; i++) {
    encode(STR_SIZE, str, &str2_size, &str2);
    s += str2_size;
    free(str2);
  }
  printf("encode: %d, %.2f\n", s, (float)(clock() -
t)/CLOCKS_PER_SEC);

  encode(STR_SIZE, str, &str2_size, &str2);

  s = 0;
  t = clock();
  for (int i = 0; i < TRIES; i++) {
    if (decode(str2_size, str2, &str3_size, &str3) != 0) {
      printf("error when decoding");
    }
    s += str3_size;
    free(str3);
  }
  printf("decode: %d, %.2f\n", s, (float)(clock() -
t)/CLOCKS_PER_SEC);
}
```

APÊNDICE E - BASE64 EM RUST

```
extern crate base64;

use base64::{encode, decode};

const TRIES: usize = 100;

fn main() {
  let size = i32::pow(10, 6) as usize;

  let input = vec![b'a'; size];
  let mut output = String::new();

  let mut time_start = precise_time_ns();
  let mut sum = 0;
  for _ in 0..TRIES {
    output = encode(&input);
    sum += output.len();
  }
  println!("encode: {}", sum);

  sum = 0;
  for _ in 0..TRIES {
    sum += decode(&output).unwrap().len();
  }
  println!("decode: {}", sum);
}
```