JOÃO RIBEIRO ANDREOTTI

TESTES DO SISTEMA OPERACIONAL JAMESOS

Trabalho apresentado como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Profª Maísa Soares dos Santos Lopes

VITÓRIA DA CONQUISTA

2024

RESUMO

O desenvolvimento de software cada vez mais necessita da implementação de testes automatizados. O mesmo se aplica para testes de sistemas operacionais. O SO JamesOS foi implementado de uma maneira incremental e modular, porém os testes automatizados não foram desenvolvidos. Por conta disso, este trabalho propõe a implementação de um *framework* de testes de caixa branca, utilizando técnicas de introspecção de máquina virtual para testes cada módulo do JamesOS. O *framework* foi desenvolvido utilizando a linguagem de programação *python*, utilizando a API da ferramenta *Gnu Debugger*. O depurador é utilizado em conjunto com o emulador de sistemas QEMU. Dessa forma, sendo possível controlar o fluxo de execução e o estado so sistema operacional em execução.

Palavras-chaves: Sistemas Operacionais; Teste de Software; Máquina Virtual; Introspecção; Caixa Branca; Grafo de fluxo de controle.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE ABREVIATURAS E DE SIGLAS

API	*Application Programming Interface*
ATF	*Automated Test Framework*
BIOS	*Basic Input/Output System*
CPU	*Central Processing Unit*
DMA	*Direct Memory Access*
DWARF	*Debugging With Arbitrary Record Formats*
ELF	*Executable and Linkable Format*
GDB	*GNU Debugger*
GDT	*Global Descriptor Table*
GFC	Grafo de Fluxo de Controle
IPC	*Interprocess Communication*
JTAG	*Joint Test Action Group*
KVM	*Kernel-based Virtual Machine*
MMU	*Memory Management Unit*
PIT	*Programmable Interval Time*
QMP	*QEMU Monitor Protocol*
ROM	*Read Only Memory*
RSA	*Rivest–Shamir–Adleman*
SD	*Secure Digital*
SO	Sistema Operacional
USB	*Universal Serial Bus*
VGA	*Video Graphics Array*
kSLOC	*1000(K) Source Lines of Code*

SUMÁRIO

1 INTRODUÇÃO

O sistema operacional JamesOS foi criado com o intuito de auxiliar no estudo de Sistemas Operacionais (SOs), seus algoritmos e mecanismos (Correia, 2020). Dessa maneira, utilizando um processador x86, o SO implementou o módulo de inicialização, e outras funcionalidades como *drivers* de vídeo, teclado, e também mecanismos de multitarefas.

O JamesOS não contém nenhum tipo de teste de código, podendo levar a ter diversos erros no sistema operacional. Como o projeto foi desenvolvido com o objetivo de auxiliar no estudo do tópico de SO, e cada vez mais se faz necessário implementar testes nesse tipo de sistema, surge a necessidade da implementação de testes para o JamesOS.

Os testes de *software* são um fator fundamental no desenvolvimento de *software*. Já que, como é exemplificado em Myers et al. (2011) metade do custo de desenvolvimento está associado aos testes.

Em Biggs et al. (2018) é demonstrado que a densidade de erros pode variar de 0.5/mil linhas de código (kSLOC) no melhor caso, até 6/kSLOC no pior caso. Também foi estimado que o *kernel* do Linux na versão 4.15 teria 13.000 *bugs*. Portanto, é extremamente necessário que os códigos sejam validados de uma maneira eficiente, afim de minimizar os erros no núcleo de um sistema operacional, já que tende a ser a parte mais crítica de um sistema.

Atualmente existem diversas maneiras para testar sistemas operacionais. Pode ser utilizado testes baseados em provas matemáticas, como o *seL4* (seL4, 2024). Também pode ser através de testes de módulos em um projeto separado, como no Linux (Larson, 2002). Uma outra possibilidade é através de um *framework* de testes baseados em emulador de processadores, como no HelenOS que utilizou o QEMU (Sucha, 2013).

Os testes comentados utilizam a técnica de testes de caixa branca. Essa técnica de testes tem como base o código fonte do programa (Spillner et al., 2014). Um dos objetivos principais é executar o máximo de linhas de código possível para atingir uma melhor cobertura. Por conta disso, quanto maior a cobertura de testes, maior o número de linhas de código testadas e validadas. Por fim, como testar um sistema operacional utilizando a técnica de testes de caixa branca?

1.1 OBJETIVOS

1.1.1 Geral

Desenvolver um *framework* de testes para o sistema operacional JamesOS. Esse *framework* será utilizado para criar testes de caixa branca, para validar os módulos do sistema operacional.

1.1.2 Específicos

Os objetivos específicos são:

- Analisar ferramentas de testes para sistemas operacionais
- Implementar os testes do JamesOS
- Validar os testes implementados

1.2 METODOLOGIA

Sistemas da informação são implementados em uma organização com o objetivo de melhorar a eficiência e efetividade da organização (Hevner et al., 2004). Para isso é produzido um conhecimento, o qual irá auxiliar no desenvolvidos desses sistemas.

A produção deste conhecimento envolve dois paradigmas, o *design thinking* e *design science*. O primeiro paradigma está associado à ciência tradicional, sendo assim, sua aplicação tem como objetivo explicar ou prever fenômenos organizacionais e humanos. Com isso, utilizando a análise, o projeto, a implementação, o gerenciamento e, por fim, o uso dos sistemas da informação. Já o *design science*, está relacionado à engenharia e a ciência do artificial, sendo um paradigma de solução de problemas. Dessa maneira, aplicando teorias base no desenvolvimento de sistemas da informação.

O *design science* cria e avalia artefatos da tecnologia da informação, os quais tem a intenção de solucionar problemas organizacionais.

Para utilizar o *design science* como um paradigma de pesquisa de sistemas da informação, o *design* deverá ser tratato tanto como um processo quanto como um produto ou artefato. Dessa maneira, o *design-science research* DSR é um paradigma para resolução de problemas, o qual utiliza duas perspectivas diferentes, processos de *design* e *design* de artefatos.

O DSR produz dois processos de *design* e quatro artefatos de *design*. Os dois processos identificados são o de desenvolvimento e o de avaliação. Já os artefatos identificados são os constructos, modelos, métodos e instanciações.

O processo de *design* é uma sequência de atividades as quais produzem um artefato de *design*, utilizando os processos de desenvolvimento e de avaliação. Dessa forma, esse é um processo cíclico entre as atividade de desenvolvimento e de avaliação até que seja desenvolvido um artefato final. Logo, a avaliação do artefato, produz informações para melhor entendimento do problema, ajudando a melhorar a qualidade do artefato e do processo de *design*.

O *design* de artefato tem como objetivo de solucionar problemas não resolvidos. Dessa forma, produzindo os artefatos de constructos, modelos, métodos e instanciações. Os constructos definem a linguagem utilizada para desenvolvimento e comunicação da solução. Os modelos utilizando os constructos para representar situação de mundo real, logo auxiliando no entendimento da solução, já que normalmente estabelece uma conexão entre o problema e a solução. Os métodos definem os processos, podendo ser um método formal, como uma prova matemática, ou descrições informais sobre melhores práticas para buscar a solução desejada. Por fim, as instanciação demonstram como a união dos constructos, modelos e métodos podem ser utilizados para o desenvolvimento de um sistema funcional.

1.2.1 Diretrizes

O DSR tem como fundamento principal que o conhecimento e o entendimento do problema de *design* são adquiridos no desenvolvimento e aplicação do artefato. Por conta disso, foram derivados sete diretrizes, o *design* como artefato, a relevância do problema, a avaliação do *design*, a contribuição da pesquisa, o rigor da pesquisa, o design como processo de busca e a comunicação da pesquisa. Essas diretrizes são explicadas no QUADRO 1.

QUADRO 1 – DIRETRIZES DO DSR

Guia	Descrição
Design como artefato	O DSR precisa produzir um artefato no formato de um modelo, método ou instanciação.
Relevância do problema	O objetivo do DSR é desenvolver tecnologias como solução para problemas relevantes e importantes.
Avaliação do *design*	A utilidade, qualidade e eficiência do artefato gerado deve ser demonstrada rigorosamente através de métodos bem definidos.
Contribuição da pesquisa	Um DSR efetivo deve prover contribuições claras e verificáveis na área de artefato de *design*, fundamentos de *design* e metodologias de *design*.
Rigor da pesquisa	O DSR depende da aplicação de métodos rigorosos para avaliação e construção do *design* do artefato.
Design como processo de busca	A busca para um artefato efetivo requer a utilização de meios disponíveis para atingir um objetivo.
Comunicação da pesquisa	O DSR precisa ser apresentado de maneira efetiva para audiências de tecnologia e também de gerenciamento.

FONTE: Hevner et al. (2004)

2 REFERENCIAL TEÓRICO

2.1 SISTEMAS OPERACIONAIS

Os Sistemas Operacionais (SOs) podem ser compreendidos sob duas perspectivas principais, ambas fundamentadas em suas funcionalidades essenciais. A primeira abordagem os define como uma máquina estendida, enquanto a segunda os caracteriza como gerenciadores de recursos (Tanenbaum; Bos, 2022).

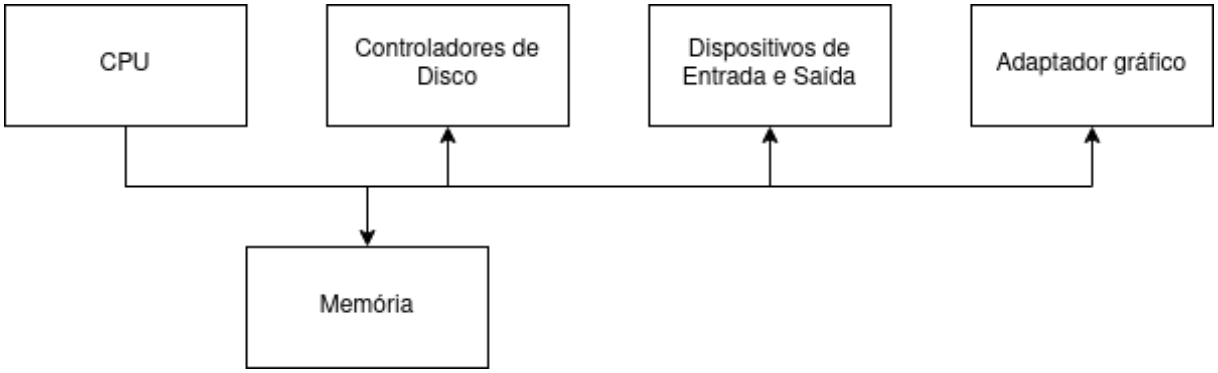
A concepção de um Sistema Operacional como uma máquina estendida refere-se à implementação de abstrações sobre o hardware. Essas abstrações são concretizadas por meio de *drivers* e interfaces, permitindo que operações complexas sejam simplificadas. Por exemplo, para imprimir um texto, o usuário pode simplesmente invocar uma função da interface de impressão, sem a necessidade de se preocupar com os detalhes técnicos do controlador da impressora. Além disso, uma única interface pode ser utilizada para abstrair diferentes controladores, possibilitando a interação com diversos dispositivos de forma uniforme.

Por outro lado, a perspectiva de um Sistema Operacional como gerenciador de recursos enfatiza a técnica de multiplexação, que aloca tempo e espaço para cada recurso disponível. Isso permite o compartilhamento eficiente de processadores, memória, disco e outros dispositivos. Em um SO multitarefa, por exemplo, o processador e a memória são utilizados simultaneamente por diferentes processos, garantindo que a execução de um não interfira no desempenho do outro. A multiplexação temporal é aplicada ao processador, enquanto a multiplexação espacial é utilizada na gestão da memória.

Um sistema computacional moderno e de propósito geral é composto por uma ou mais CPUs conectadas a controladores de disco, memória, controladores USB e adaptadores gráficos (Silberschatz et al., 2018). Essa arquitetura pode ser visualizada na FIGURA 1.

As estruturas dos sistemas operacionais podem ser classificadas em várias categorias: monolíticas, em camadas, *microkernel*, modulares e híbridas. Essas classificações foram desenvolvidas para abordar os desafios inerentes à complexidade dos sistemas operacionais contemporâneos. Assim, cada estrutura oferece um modelo que orienta a interação entre os diferentes componentes do sistema—hardware, sistema operacional e aplicações de usuário.

FIGURA 1 – SISTEMA COMPUTACIONAL MODERNO



FONTE: Silberschatz et al. (2018)

2.1.1 Monolíticas

Nesta estrutura, o sistema operacional é implementado como um único bloco de código que executa em modo privilegiado. Todos os serviços do sistema, como gerenciamento de processos, memória e dispositivos, são integrados em um único núcleo. Essa abordagem proporciona alta eficiência, mas pode dificultar a manutenção e a atualização, pois qualquer modificação no núcleo pode afetar o sistema como um todo.

2.1.2 Em Camadas

Esta estrutura organiza o sistema operacional em diferentes camadas, onde cada camada oferece serviços para a camada superior e utiliza os serviços da camada inferior. Essa abordagem facilita a compreensão e a manutenção do sistema, pois permite que as alterações em uma camada não impactem diretamente as demais, além de possibilitar a reutilização de componentes.

2.1.3 Microkernel

O *microkernel* minimiza as funcionalidades do *kernel* do sistema operacional, delegando a maior parte dos serviços para processos de espaço de usuário. Apenas as funções essenciais, como gerenciamento de processos e comunicação entre processos, são mantidas no *kernel*. Isso resulta em maior modularidade e facilidade de manutenção, mas pode acarretar uma sobrecarga devido à comunicação entre processos.

2.1.4 Modular

A estrutura modular permite que o sistema operacional seja composto por diversos módulos que podem ser carregados e descarregados conforme necessário. Isso proporciona flexibilidade e facilita a personalização do sistema, permitindo que os

usuários escolham os módulos que desejam incluir, sem comprometer a funcionalidade principal do sistema.

2.1.5 Híbridas

As estruturas híbridas combinam características de sistemas monolíticos e *microkernel*, aproveitando os pontos fortes de ambas as abordagens. O *kernel* principal pode incluir algumas funcionalidades adicionais, mas mantém a modularidade e a capacidade de expansão, permitindo um bom equilíbrio entre desempenho e flexibilidade.

2.2 PROCESSADORES

O processo de inicialização de um sistema operacional é dependente da arquitetura utilizada. Dessa forma, para processadores *x86*, é utilizado um programa auxiliar chamado BIOS. Esse programa, além de implementar *drivers* básicos para alguns dispositivos de entrada e saída como VGA e controladores de disco, ele carrega o primeiro código de inicialização do sistema na memória. Já os processadores de arquitetura *arm*, não tem uma BIOS padronizada e não necessariamente precisa ter uma BIOS. A inicialização começa com o processador lendo o código inicial de um dispositivo ROM/FLASH, o qual pode ser o único código executado, mas também pode ser utilizada para carregar outras partes do sistema operacional de uma memória externa como um cartão Secure Digital (SD) (OSDev, 2024b).

O código de inicialização de processadores *x86* é procurado pela BIOS em cada disco acessível por uma assinatura no byte 510 e 511 do disco. Essa assinatura é constituída pelos bytes 0x55 e 0xaa respectivamente. Após a identificação do disco de inicialização, o próximo passo depende do tipo de disco. Por exemplo se o disco é um disquete, como é utilizado pelos JamesOS, os primeiros 512 bytes do disco são carregados na memória, sendo a primeira parte da inicialização de um sistema operacional. Já para disco rígidos o processo é um pouco diferente (OSDev, 2024c).

Após o primeiro estágio de inicialização da BIOS, o código definido pelo usuário é executado. Como apenas 512 bytes do sistema operacional foram carregados na memória, qualquer programa maior do que esse tamanho precisa ser ser carregado nesse estágio. Porém, essa segunda leitura de disco é feita pelo usuário e não pela BIOS. Logo, uma das funções da inicialização de um sistema operacional é carregar a outra parte do código.

Por questões de compatibilidade com processadores de arquiteturas de 16 bits, o processador *x86*, que é uma arquitetura de 32 bits, inicializa em um modo chamado modo real (*Real Mode*). Então, uma outra função do estágio de inicialização inicial é executar os passos para transicionar para 32 bits ou modo protegido (*Protected Mode*).

Para tal fim, as interrupção do processador precisam ser desligadas e uma estrutura binária chamada *Global Descriptor Table* (GDT) é carregada através de uma instrução especial lgdt. Por fim, a conclusão da transição se da pelo bit menos significativo do registrador CR0, precisando ser modificado para 1 e utilizando uma instrução *jmp* para uma *label* que contém código 32 bits em modo protegido.

A partir desse momento, o sistema operacional pode começar a inicialização das suas funções principais, como inicializar o vetor de interrupção, o *Programmable Interval Timer* (PIT) e o teclado. Além disso, os processadores *x86* contém um dispositivo gerenciador de memória chamado *Memory Management Unit* (MMU). Esse dispositivo também é utilizado em conjunto com a GDT, porém, uma outra funcionalidade chamada paginação é amplamente utilizada em sistema operacionais modernos para fornecer uma memória virtual.

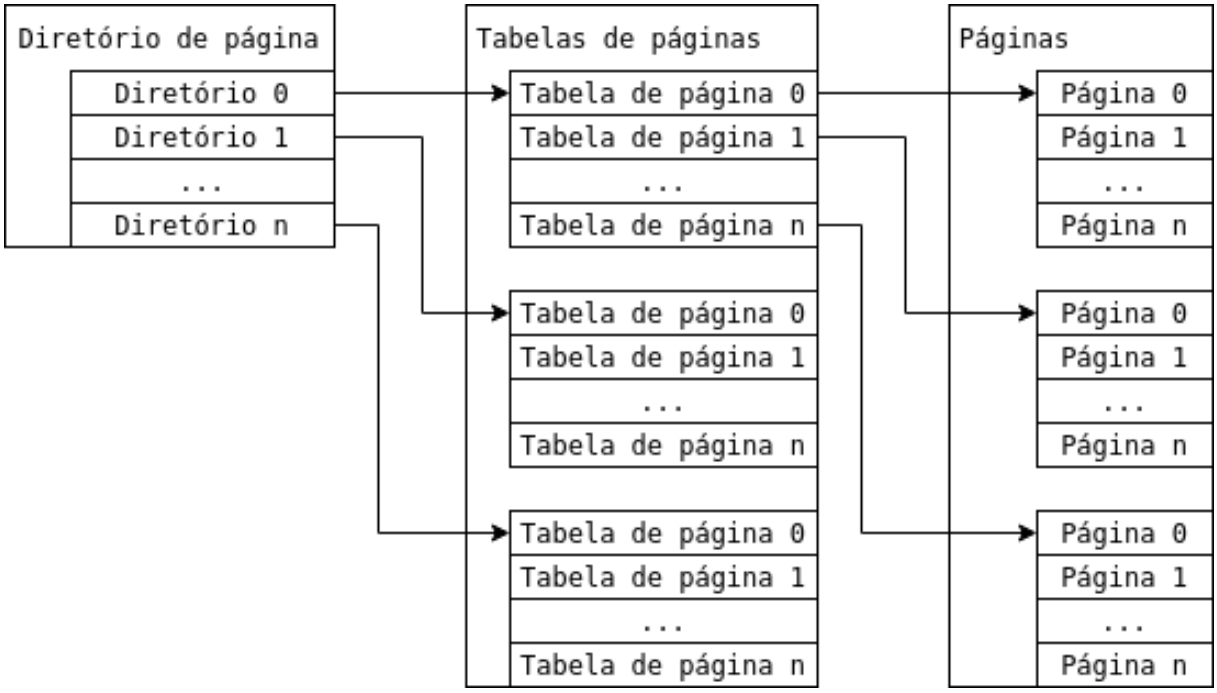
A paginação é um sistema complexo, capaz de criar uma memória virtual para o sistema operacional inteiro. De uma maneira simples, cada parte da memória é mapeada em uma porção de memória chamada página e pode ter um tamanho de 4096 bytes. Cada página é mapeada para uma região da memória física. Por conta dessa estrutura, um sistema computacional com 256MB de memória utilizando arquitetura *x86*, pode ter até 4GiB de memória virtual. Uma outra funcionalidade é que diversas páginas, que são estruturas virtuais, podem ter o mesmo endereço virtual, porém, com um endereço físico diferente. Uma visualização dessas estruturas podem ser vistas na FIGURA 2 (OSDev, 2024j).

2.3 VIRTUALIZAÇÃO

Virtualização é uma tecnologia que permite a abstração do hardware de um único computador em diferentes ambientes de execução, portanto criando a ilusão que cada ambiente está executando em seu próprio computador (Silberschatz et al., 2018). Dessa forma, temos a virtualização da memória sendo utilizada em sistemas operacionais modernos, operando com auxílio de hardware, através da MMU para prover memória virtual (OSDev, 2024i). Essa memória virtual pode ser de um processo de usuário, mas também pode ser mapeada para ser utilizada pelo *kernel* do sistema operacional (OSDev, 2024j).

Além da virtualização de hardware, existe a virtualização de software. Como por exemplo sistemas de virtualização *sandbox*, sendo um dos exemplos mais comuns o sistema operacional *Android*. O *Android* utiliza o método virtualização de aplicação completa, implicando que cada aplicação tem o seu próprio ID *Linux*. Por fim, utilizando esse ID para prover essa *sandbox* em nível de *kernel* (Android, 2024).

FIGURA 2 – ESTRURUA DO SISTEMA DE PAGINAÇÃO



FONTE: OSDev (2024j)

2.3.1 Máquinas virtuais

Avançando nos conceitos de virtualização, chegamos em máquinas virtuais. O sistema operacional *Virtual Machine Facility/370* (VM/370) trouxe a ideia de máquinas virtuais para prover um sistema de vários usuários para o *IBM System/370* de uma maneira aparentemente separada e independente (Creasy, 1981).

As máquinas virtuais são a principal utilidade da virtualização de interfaces, podendo ser classificadas em três categorias, maquinas virtuais de sistema, de sistemas operacional e de processo (Maziero, 2019), como pode ser visto na FIGURA 3.

FIGURA 3 – CLASSIFICAÇÃO MÁQUINAS VIRTUAIS

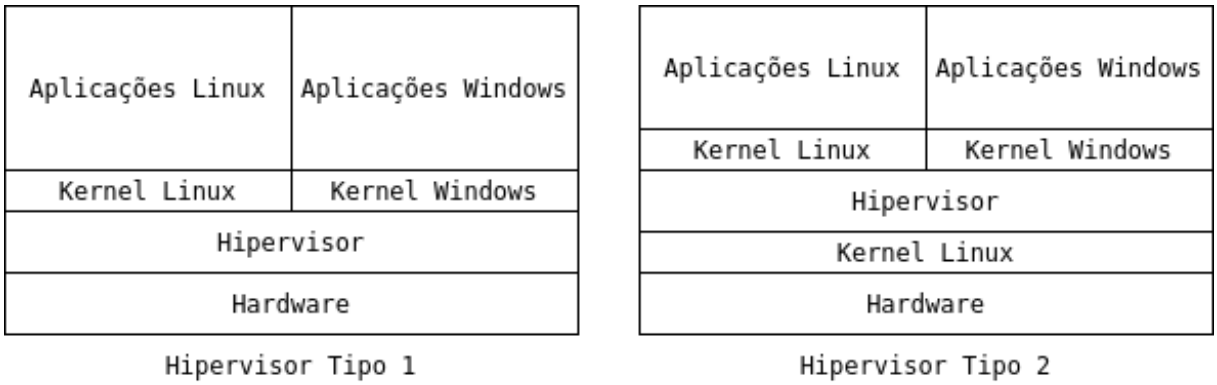
FONTE: Maziero (2019)

2.3.2 Hipervisores

Uma máquina virtual de sistema, ou de hardware, provê uma interface de hardware completa para um ou mais sistemas operacionais convidados, com suas respectivas aplicações, que executam de forma isolada e independente (Maziero, 2019). O provedor dessa interface é chamado de hipervisor, que podem ser classificados em hipervisores nativos ou de tipo 1 e hipervisores convidados ou de tipo 2.

Os hipervisores de tipo 1 tecnicamente funcionam como um sistema operacional, executando no modo do processador mais privilegiado. Tem a função de suportes múltiplas cópias do hardware existente, chamado de máquinas virtuais. Em contrapartida, os hipervisores do tipo 2, são programas os quais dependem do sistema operacional para a alocação e escalonamento de recursos, exatamente como um programa de usuário. Uma visualização da diferença entre os tipos de hipervisores foi representada pela FIGURA 4.

FIGURA 4 – TIPOS DE HIPERVISORES



FONTE: Maziero (2019)

Um dos principais hipervisores de tipo 1 é o XEN (Xen, 2024), tendo como característica um sistema operacional chamado de *Dom0*, o qual tem permissões especiais, podendo ter acesso direto ao hardware, dessa forma, sendo capaz de implementar *drivers* de hardware. Enquanto todos os sistemas operacionais convidados são chamados de *DomU*, interagindo com o *Dom0* para utilização dos *drivers* através de interfaces providas pelo XEN. O KVM (Kernel-based Virtual Machine) (KVM, 2024) é um módulo do Linux que transforma o sistema operacional nativo em um hipervisor de tipo 1 (RedHat, 2024). Como o KVM faz parte do Linux, ele também tem componentes a níveis de *kernel* como um gerenciador de memória, escalonador de processos, *drivers* de dispositivos e mecanismos de segurança.

Já para hipervisores de tipo 2, um exemplo principal para o universo dos

sistemas Linux é o QEMU. O QEMU é um emulador de máquina e virtualizador (QEMU, 2024b) com suporte para diversas arquiteturas como *x86* (processadores Intel 80386 compatíveis), *arm*, *Sparc* e *PowerPC* (Bellard, 2005). Além de prover emulação de dispositivos entrada e saída como teclado, mouse e controladores de disco. Para uma maior eficiência e flexibilidade, o QEMU também pode ser integrado com o KVM e com o XEN.

2.4 DEPURADORES

Um depurador ou *debugger* é um programa com privilégios especiais sob o programa sendo depurado, capaz de iniciar, pausar, inspecionar e modificar o estado do programa. Um dos principais usos de um depurador é para encontrar error no programa analisado (Yuan, 2024). Os depuradores podem ser criados para uma linguagem específica, por exemplo o *Visual Studio Debugger*, o qual suporta diversas linguagens diferentes como o c++, C#, Visual Basic e JavaScript (Microsoft, 2024). Mas também, pode ter um propósito mais geral, como o GDB (GNU Debugger), que além de dar suporte para diversas linguagens de programação como C, C++ e Rust, também é possível depurar arquivos ELF (Executable and Linkable Format). Dessa maneira, provendo uma flexibilidade enorme, já que basicamente o que é necessário para analisar um programa são as instruções de máquina que ele executa (GNU, 2024a).

O DWARF Committee (2024), é um formato para adicionar informações de depuração em um binário. Uma das suas característica é de que não tem uma associação direta com nenhum compilador (Group, 2017). Dessa maneira, criando uma representação assertiva do código fonte, a qual poderá ser interpretada por depuradores. Além de que, tem o objetivo de ser compatível com diferentes linguagens de programação. Com isso, para facilitar a análise de um programa através do GDB, a ferramenta implementou um processador desses símbolos.

O depurador GDB também oferece a opção de depuração remota, através de um GDBServer. O GDBServer é um programa que permite que o GDB acesse um programa depurado através da rede (man7.org, 2024) Para isso, o GDBServer deve executar o programa alvo e o GDB cliente irá se conectar. Um detalhe importante é que o GDB cliente não tem acesso aos símbolos do programa alvo, a não ser que o cliente tenha uma cópia do programa em execução com os símbolos, os quais podem ser carregados explicitamente no GDB.

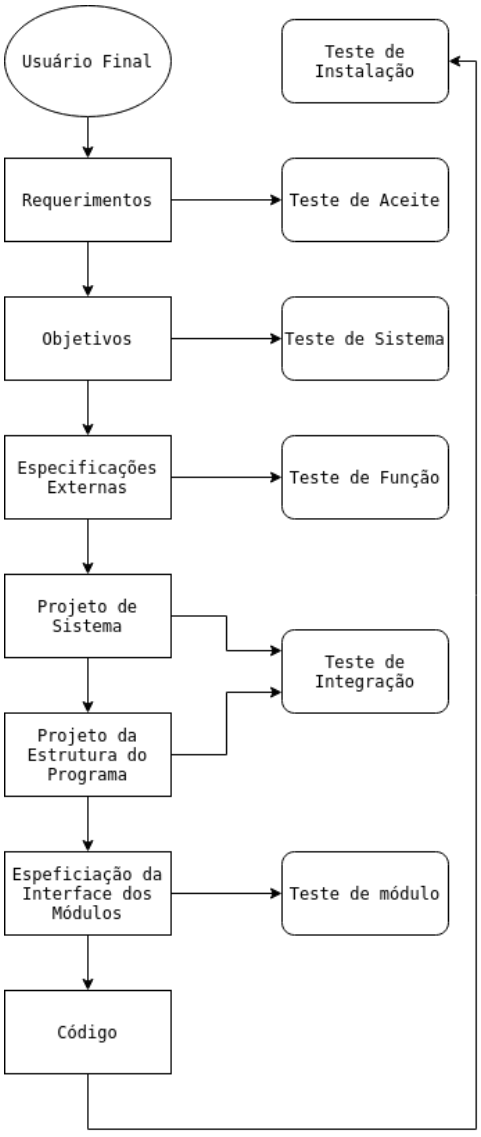
O QEMU implementa um GDBServer (QEMU, 2024c), permitindo que códigos que são executados dentro do QEMU possam ser depurados através do GDB. Logo, é possível depurar um código de um sistema operacional desde a BIOS.

2.5 TESTE DE SOFTWARE

A qualidade de software refere-se ao grau em que um software atende a requisitos e expectativas específicas. Ela abrange aspectos como funcionalidade, usabilidade, eficiência, manutenibilidade e segurança. Para garantir a qualidade, é essencial adotar boas práticas durante o desenvolvimento, como revisões de código e testes de software. Um software de alta qualidade reduz custos de manutenção e facilita a evolução e adaptação do sistema ao longo do tempo (Myers et al., 2011).

O teste de software é diretamente derivado de cada fase de desenvolvimento de software. Como demonstrado na FIGURA 5, cada momento do desenvolvimento implica em um tipo de teste.

FIGURA 5 – RELAÇÃO DO DESENVOLVIMENTO DE SOFTWARE COM OS SEUS RESPECTIVOS TESTES



FONTE: Myers et al. (2011)

Testes de software podem ser divididos em duas estratégias principais, caixa-preta e caixa-branca. A primeira é baseada em dados, implicando que o comportamento interno do programa pode ser ignorado e o foco é apenas nos dados que são utilizados para a entrada e nos dados de saída. Com isso, os testes necessariamente são baseados apenas nas especificações, já que o código é desconhecido ou ignorado. Em contrapartida, os testes de caixa-branca são feitos utilizando apenas o código do programa.

Cada estratégia é utilizada para se beneficiar as suas vantagens, por exemplo, em um teste de caixa-branca o número de caminhos diferentes que um determinado programa pode ser executado é infinito, então testar todos os caminhos é inviável. Da mesma maneira que a entrada de um teste caixa-preta pode ser infinita, como em um compilador.

As metodologias primárias de teste humanas são inspeção de código, passo a passo e usabilidade. Sendo as duas primeiras orientadas a código (MYERS, 2011). Já a de usabilidade, que também pode ser chamada de teste de usuário, tem o objetivo de avaliar a perspectiva do usuário final em um cenário real em relação ao *design*, a interface que está interagindo, e também, caso alguma especificação esteja faltando.

Ambas as metodologias orientadas a código envolvem um grupo de pessoas visualmente inspecionando o código com o objetivo de encontrar erros, mas não de solucioná-los. Um das vantagens dessas metologias é que o erro pode ser precisamente encontrado em uma linha de código específica, em contrapartida aos teste de caixa-preta, os quais apenas identificam um erro a partir de um resultado inesperado, já que o teste não foi baseado no código.

2.5.1 Testes de Caixa-Branca

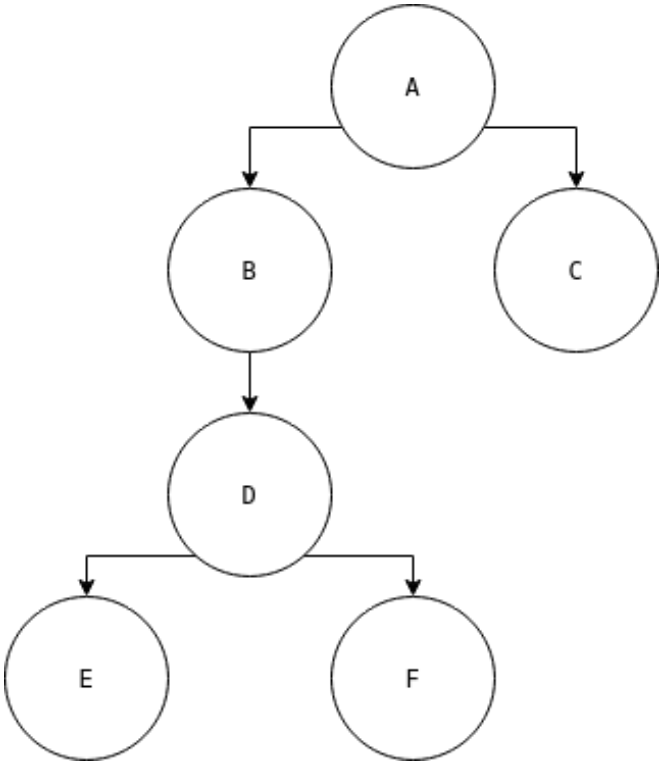
A medida que o tamanho do software vai aumentando, cada vez fica mais difícil de emplacar apenas testes manuais. Além disso, as metodologias humanas citadas anteriormente são mais eficientes para encontrar erros de alto nível, como por exemplo erros relacionados á análise de requisitos. Então, para auxiliar no processo de teste são utilizados testes de módulo ou de unidade. O objetivo dessa metodologia é comparar uma unidade de código à uma um especificação funcional ou de interface, porém, com a finalidade de contradizer ou encontrar um erro.

Para utilizar o teste de unidade, duas diretrizes principais devem ser seguidas, desenvolver casos de teste efetivos e a maneira em que os módulos são integrados no programa sendo testado. Sendo assim, dependendo da integração, os casos de testes podem variar, mas também as ferramentas de teste utilizadas.

Considerando que um programa é constituído de várias unidades e elas são

dependentes em uma maneira hierárquica, o programa pode ser organizado em uma árvore (grafo), como demonstrado na FIGURA 6. Cada nó pode ser testado independente, utilizando *stubs* e *drivers* (funções de teste), utilizando uma abordagem não-incremental. Por outro lado, uma abordagem incremental também pode ser aplicada, testando todos os módulos partindo das folhas até a raiz, ou partindo da raiz até as folhas. Essas abordagens são chamadas de *bottom-up* ou *top-down*, respectivamente. A diferença é que não são utilizados *stubs* na abordagem *bottom-up*, apenas *drivers*. Utilizando apenas *drivers*, os módulos já testados são utilizados no lugar de *stubs*.

FIGURA 6 – ÁRVORE DE MÓDULOS DE UM PROGRAMA



FONTE: Myers et al. (2011)

A abordagem incremental também engloba testes de integração, já que cada unidade pode ser considerada uma folha e todos os outros nós eventualmente irão depender dos seus descendentes. Dessa maneira, integrando com eles.

A execução dos testes também implica uma série de outras situação que devem ser analisadas. Já que, um caso de teste que o módulo foi executado corretamente, não garante que foi um teste eficiente. Da mesma maneira, um caso de teste que gerou um resultado inesperado pode ter ocorrido pelo resultado esperado estar incorreto. Então, a definição do caso de teste e da expectativa é tão importante quanto a execução do código testado.

2.5.2 Testes de alta ordem

Testes de alta ordem são classificados em testes de função, sistema, aceitação e instalação. Cada um desses testes tem objetivos diferentes. O teste de função tem como objetivo encontrar discrepâncias entre as especificações externas do programa, com o programa propriamente dito (Myers et al., 2011). O teste de sistema tem o foco em analisar o desenvolvimento das especificações externas, dessa forma testes são baseados na documentação do usuário, a qual foi utilizada para desenvolver as especificações externas. O testes de aceitação é feito pelo usuário final, com isso, sendo avaliado pelo solicitador do programa. Por fim, o teste de instalação não está relacionado com o processo de desenvolvimento, o objetivo é testar o ambiente o qual o programa irá ser executado, sendo assim, teste de dependências de bibliotecas externas, configuração de hardware e de rede são testados nesse momento.

A utilização dos testes de sistema podem ser extremamente úteis para o teste de sistemas operacionais, já que abrangem uma ampla quantidade de subdivisões. Essas categorias podem ser visualizadas no QUADRO 2.

O teste de volume pode testar o sistema de arquivos de um sistema operacional, utilizando arquivos grandes o suficientes para atingir o limite estabelecido. O teste de estresse pode ser utilizado para testar o número máximo que aplicações em que um sistema suporta. Testes de segurança podem ser aplicados para testar o mecanismo de proteção de memória. O teste de configuração pode ser utilizado para validar se um hardware específico está disponível, assim como a utilização do teste de recuperação pode ser executado para determinar o comportamento do sistema operacional quando ocorre uma falha de hardware.

2.5.3 Evolução de software

A adição de novas funcionalidades ao programa existente normalmente são a maior causa dos bugs (Myers et al., 2011). Para diminuir esse tipo de problemas durante o desenvolvimento, os testes de regressão podem ser utilizados. Eles garantem que novas alterações de código não afetem a funcionalidade existente, causando um erro. Para isso, os casos de testes que foram validados anteriormente são executados novamente, garantindo, pelo menos que o código continua funcionando como esperado.

2.6 TRABALHOS RELACIONADOS

Para testar um programa criado para ser executado em um sistema operacional moderno como Linux, *frameworks* de testes são desenvolvidos para executar funções do programa alvo. A partir disso, um programa de teste é desenvolvido para criar as entradas de uma função, depois executar a função, e por fim, validar o retorno. Por

QUADRO 2 – TESTES DE SISTEMA.

Categoria	Descrição
Facilidade	Garantir que a funcionalidade especificada nos objetivos foi implementada.
Volume	Submeter o programa em altos volumes de dados para serem processados.
Estresse	Submeter o programa em altas cargas de processamento.
Usabilidade	Determidar o quão bem o usuário consegue interagir com o programa.
Segurança	Submeter o programa em medidas seguras.
Desempenho	Determinar se o programa condiz com os requerimentos de vazão.
Armazenamento	Garantir que o programa gerencie o armazenamento de maneira adequada.
Configuração	Validar se o programa funciona normalmente com as configuração recomendadas.
Compatibilidade/Conversão	Determinar a compatibilidade do programa com versões anteriores.
Instalação	Garantir que os métodos de instalação funcionem em todas as plataformas suportadas.
Confiabilidade	Determinar se o programa condiz com as especificações de confiabilidade como tempo de atividade.
Recuperação	Testar se o sistema de recuperação funciona como projetado.
Facilidade de manutenção	Determinar se a aplicação fornece dados utilizados para técnico suporte.
Documentação	Validar a precisão da documentação.
Procedimento	Determinar a precisão de procedimentos especiais de uso ou de manutenção do programa.

FONTE: Myers et al. (2011)

exemplo, para a linguagem Java existe o JUnit (JUnit, 2024), para o c# existe o xUnit (xUnit.net, 2024) e para a linguagem C o Unity (ThrowTheSwitch, 2024).

Para o teste de módulos iniciais de um sistema operacional, como o de inicialização, os *frameworks* citados anteriormente não são adequados, já que ainda não tem um ambiente de execução de programas para executar um código de teste utilizando o mesmo ambiente do sistema operacional. Por conta disso, outras abordagens para depuração e teste precisam ser utilizadas.

2.6.1 Testes para sistemas *Unix*

2.6.1.1 Instrumentação do QEMU

Uma abordagem possível é por instrumentação do QEMU para gerar dados necessários para a depuração. Como foi feito por (Mihajlović et al., 2014), uma instrumentação dinâmica do emulado para gerar rastreio de instruções de processos executados no sistema operacional Linux.

2.6.1.2 Anita

Anita é uma ferramenta para testes automatizados do sistema operacional NetBSD (NetBSD, 2024). A ferramenta já auxiliou na identificação de um grande número de *bugs* no NetBSD, além de ter identificados *bugs* no QEMU e em outros emuladores. Com a ferramenta é possível baixar, instalar e inicializar o sistema operacional dentro de uma máquina virtual com apenas um comando.

O SYSINST é um programa que pode ser utilizado para instalar ou atualizar um sistema NetBSD. Com isso, o foco principal da ferramenta Anita é testar o procedimento de instalação do SYSINST, detectando erros na instalação ou inicialização do sistema operacional. Porém, também é possível utilizar a ferramenta para testar o NetBSD como um todo, executando um conjunto de testes ATF (Automated Test Framework).

2.6.1.3 openQA

O openQA é uma ferramenta automatizada para testes de sistemas operacionais, e também o motor de testes automatizados para o openSUSE (openQA, 2024). Utiliza máquina virtual para a execução dos testes, validando o saída através do console serial e da tela. Também fornece funções para comunicação com o sistema operacional através de dispositivos de entrada e saída, como teclado e mouse. Além de ter funções genéricas para validar o desligamento, pausar e continuar a execução e executar uma extração da memória.

A visualização sobre os testes executados auxilia para a criação de métricas e um histórico da evolução dos testes sobre o sistema. Por isso, o openQA também tem uma aplicação web, que além de mostrar uma visualização dos testes executados, também pode ser utilizada para controlar a execução dos testes. A arquitetura do sistema completo pode ser vista na FIGURA 7.

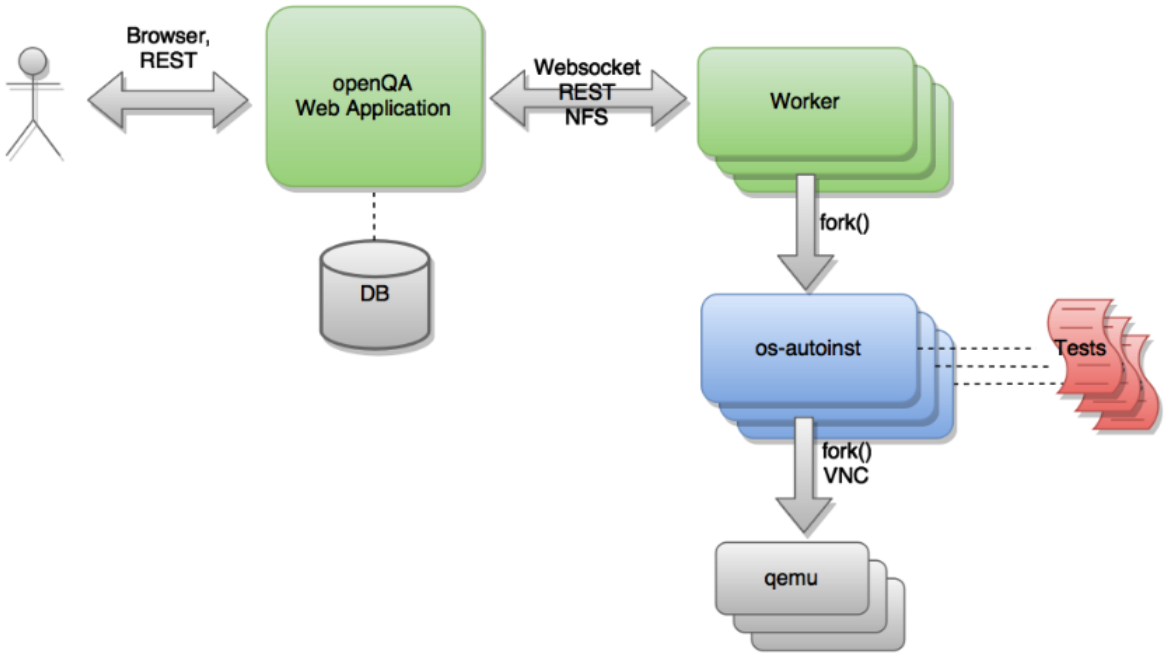
Como exemplo do sistema web openQA (openSUSE, 2024), a FIGURA 8 é uma captura de tela do servidor público do openSUSE, onde é possível visualizar o número de testes executados, além dos que falharam e foram pulados. Por fim, cada teste pode ser analisado individualmente.

2.6.2 Testes para outros sistemas operacionais

2.6.2.1 Genode

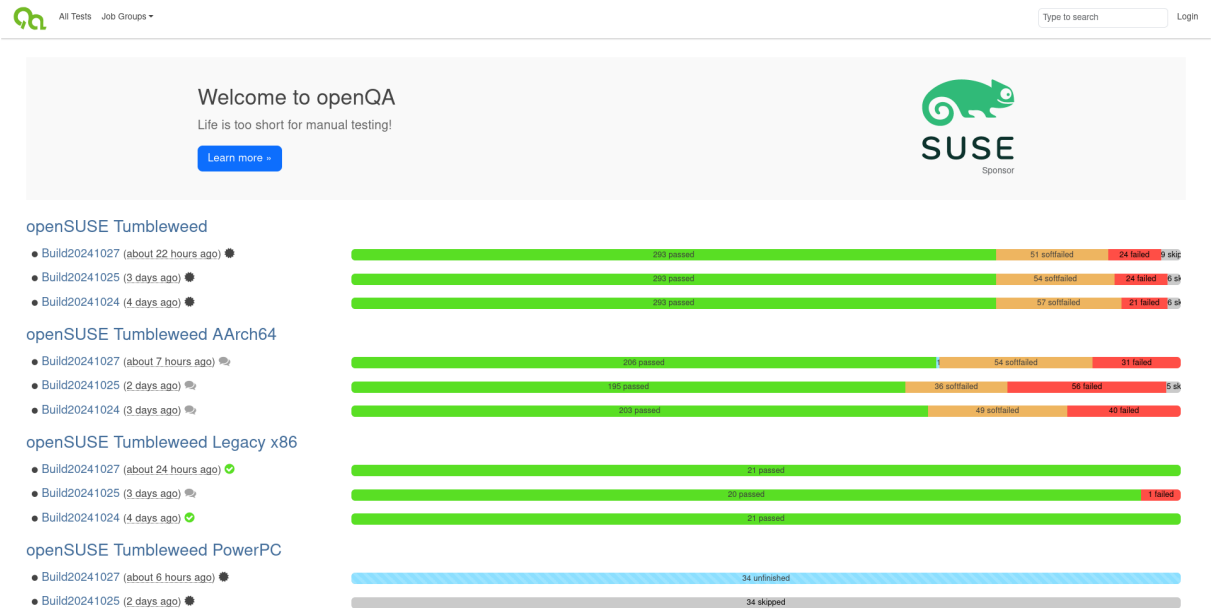
O *framework* de sistema operacional Genode é a implementação da arquitetura Genode (GenodeOS, 2024). É um kit de ferramentas para a implementação de um

FIGURA 7 – ARQUITETURA OPENQA



FONTE: openQA (2024)

FIGURA 8 – TESTES DO SISTEMA OPERACIONAL OPENSUSE



FONTE: openSUSE (2024)

sistema operacional seguro. Pode ser utilizado para sistemas operacionais de sistemas embarcados com 4MB de memória, para sistemas operacionais de propósito geral.

A arquitetura é baseada em uma estrutura recursiva, cada programa é exe-

cutado em uma *sandbox*, e as permissões de acesso para recursos são concedidas conforme a necessidade para atingir o propósito. A estrutura recursiva permite que subprogramas sejam criados, logo sub-*sandboxes* também, porém, as sub-*sandboxes* apenas podem ter as permissões do programa pai ou menos. O *framework* provê mecanismos de IPC de uma maneira restrita, diminuindo a superfície de ataque em funções críticas do sistema. O *framework* tem princípios de *microkernel* em conjunto com Unix.

A portabilidade do Genode entre *kernels* e hardwares é uma das suas principais características. Porém, como cada *kernel* e hardware requer diferentes considerações em relação à inicialização, integração e configuração, um grande conhecimento sobre o sistema executado é necessário. Para simplificar a validação dos testes do sistema operacional o Genode tem a ferramenta *autopilot*. Utilizando a ferramenta é possível executar um conjunto casos de teste em diversas arquiteturas diferentes.

2.6.2.2 seL4

O seL4 é um sistema operacional *microkernel* de propósito geral (seL4, 2024). Foi desenvolvido baseado na família L4 de sistemas operacionais, dessa forma, provendo mecanismos de segurança sem impactar no alto desempenho, a qual é característica necessária para o mundo real. Tem o objetivo de verificar formalmente o código do *kernel*, sendo possível garantir uma ausência de *bugs*.

A prova formal é validada por um provador de teorema, basicamente comparando um modelo formal esperado, com o modelo gerado pelo compilador.

2.6.2.3 Helen OS

O HelenOS é um sistema operacional *microkernel* e multi servidor de propósito geral, com suporte para multi-plataforma. As plataformas suportadas são *amd64*, *arm32*, *ia32*, *ia64*, *mips32*, *ppc32*, *sparc64*. Também tem a funcionalidade de multiprocessamento. Como o HelenOS tem um espaço de usuário multi servidor, diferentes subsistemas e *drivers* são separados em suas respectivas *tasks* e comunicam entre si utilizando um sistema de IPC (Interprocess Communication) implementado pelo *kernel*.

Os testes de software no HelenOS, inicialmente, eram limitados em quantidade, não existia um suporte para execução isolada dos testes e também não era possível gerar um relatório externo a máquina virtual. Os testes existentes englobavam tanto a nível de *kernel* quanto espaço de usuário. Por fim, para solucionar as limitações citadas anteriormente, um *framework* de testes foi desenvolvido por (Sucha, 2013).

2.6.2.4 Teste de recuperação

Um teste muito importante em sistemas operacionais é o de recuperação, já que uma falha no sistema operacional pode necessitar uma falência total, apenas resolvendo após uma reinicialização. O dano causado por esse tipo de problema pode variar, por exemplo se ocorreu em um computador de um usuário padrão, o custo pode ser irrelevante, apesar da frustração causada. Porém, se o mesmo ocorrer em um servidor de banco de dados, por exemplo, algumas transações podem ser perdias e ainda causar indisponibilidade. Em situações mais extremas, uma falha em um sistema operacional de um avião pode causa danos irreparáveis a tripulação. O (Marinescu; Candea, 2011) demonstrou como uma injeção de falha é utilizada para testar a recuperação de um código de uma maneira eficiente, através de um motor de injeção de falha a nível de biblioteca.

2.6.3 Testes de sistemas embarcados

2.6.3.1 ESPRESSIF

A Spressif é um empresa multinacional, que provê soluções para sistemas embarcados. Para auxiliar o desenvolvimento, a Espressif mantêm um *fork* do emulador QEMU com suporte ao ESP32 e algumas variações. Além também de fornecer um *framework* de desenvolvimento chamado ESP-IDF (Systems, 2024). Para a parte de teste, um *plugin* da ferramenta *pytest*, chamado *pytest-embedded* também é fornecido. Dessa maneira, tornando possível a execução de testes através de *scripts* em *python*. O ambiente de teste, nesse caso, é vasto o suficiente para os testes serem executado em um hardware real, utilizando o JTAG. Mas também, é possível a execução dos testes em um ambiente virtual utilizando o próprio QEMU. Como o *fork* do QEMU também implementa suporte para periféricos como eFuses, XTS-AES e RSA, o ambiente de teste se torna ainda mais completo.

2.6.4 Testes de máquinas virtuais

As máquinas virtuais são praticamente essenciais para testes do baixo nível de sistemas operacionais, ou para sistemas embarcados. Por conta disso, gerando uma necessidade desse tipo de software ser o mais preciso possível com as especificações, e também contendo poucos erros. Já que, uma parte do desenvolvimento de um programa para um arquitetura específica possa ser modificado por utilizar a máquina virtual como base. Para resolver esse tipo de problema uma metodologia de *fuzzing* específica de protocolo e análise diferencial foi proposta, além de um protótipo, KEmuFuzzer, desenvolvido (Martignoni et al., 2010). As máquinas virtuais testadas foram BOCHS, QEMU, VirtualBox e VMware.

3 TESTES DO JAMESOS

3.1 APLICAÇÃO DO DSR NO DESENVOLVIMENTO DO TRABALHO

3.1.1 *Design* como artefato

O autor desenvolveu um conjunto de casos de testes de módulo, com seus respectivos *drivers* ou programas de teste, para validar cada módulo do sistema operacional. Os testes foram executados em conjunto com as ferramentas GDB e o QEMU.

3.1.2 Relevância do problema

A relevância do problema se da pela aplicação de técnicas de teste de software em estágios iniciais de um sistema operacional. Dessa forma, é necessário utilizar estratégias específicas para testar esse tipo de sistema.

3.1.3 Avaliação do *Design*

Para a avaliação do *design* não foi utilizado nenhuma método formal. Porém, todos os testes foram validados baseado no Grafo de Fluxo de Controle (GFC) e seu caminho esperado.

3.1.4 Contribuição da pesquisa

A pesquisa ofereceu uma contribuição para a área de teste de software em baixo nível e de sistemas operacionais. Com isso, gerando um *framework* para teste do sistema operacional JamesOS.

3.1.5 Rigor da pesquisa

O rigor da pesquisa foi definido pelo método de teste utilizado. O método foi criar um teste para cada caminho do GFC das funções testadas.

3.1.6 *Design* como processo de busca

O processo de busca foi constituído da análise das ferramentas QEMU e GDB, para validar como os testes seriam feito. E também, algumas funções requerem técnicas específicas para serem testadas, como por exemplo introspecção de memória e de registradores.

3.1.7 Comunicação da pesquisa

A comunicação da pesquisa será através deste documento.

3.2 SISTEMA OPERACIONAL JAMESOS

Sistemas operacionais são programas complexos e considerados partes críticas de um sistema computacional, já que os erros não tratados pelo SO normalmente geram uma instabilidade no estado do sistema, podendo levar à falhas graves. Além disso, grande parte do núcleo de um sistema operacional é altamente dependente de todos os seus módulos. Logo, a mesma suposição pode ser aplicada ao JamesOS.

O JamesOS é um sistema operacional experimental, incremental e modular (Correia, 2020). Cada módulo do JamesOS pode ser classificado como uma funcionalidade, totalizando onze módulos. Esses módulos podem ser visualizados no QUADRO 3. Cada módulo será explicado com mais detalhes no momento dos seus respectivos testes.

QUADRO 3 – MÓDULOS DO JAMESOS

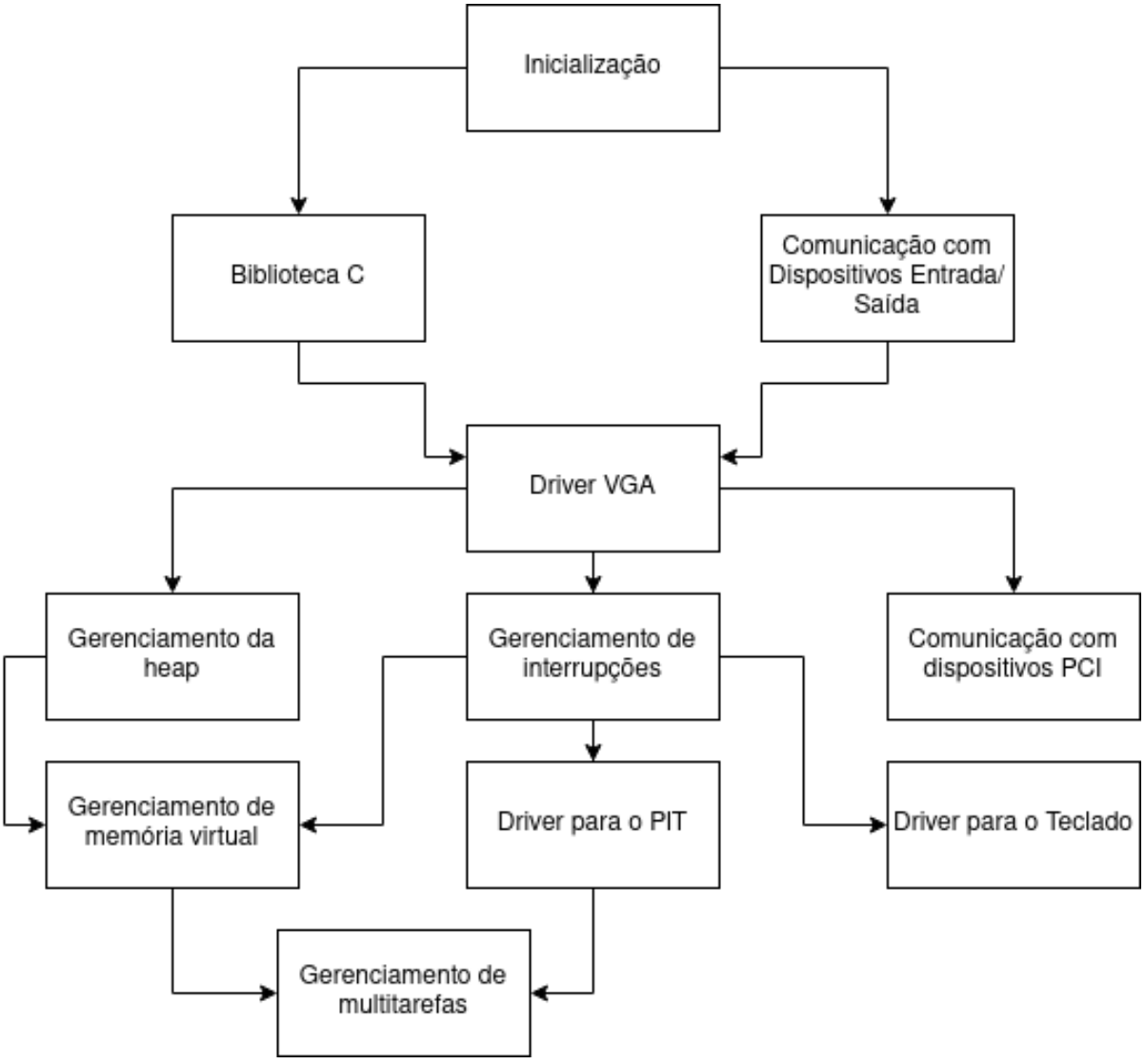
Módulo	Descrição
Módulo 0	Implementação de um mecanismo de inicialização do Sistema Operacional.
Módulo 1	Implementação de uma biblioteca C.
Módulo 2	Implementação de um mecanismo de comunicação com dispositivos de entrada/saída.
Módulo 3	Implementação de um *driver* para vídeo.
Módulo 4	Implementação de um mecanismo de tratamento de interrupções.
Módulo 5	Implementação de um *driver* para o PIT.
Módulo 6	Implementação de um *driver* para o teclado.
Módulo 7	Implementação de um mecanismo de gerenciamento para uma área de pilha.
Módulo 8	Implementação de um mecanismo de gerenciamento de memória virtual utilizando paginação.
Módulo 9	Implementação de um mecanismo de acesso à dispositivos PCI.
Módulo 10	Implementação de um mecanismo de gerenciamento de multitarefas.

FONTE: Correia (2020)

Para o sistema operacional JamesOS, o grafo de dependências é representado pela figura FIGURA 9. Com base nas dependências dos módulos do JamesOS podemos perceber que todo o sistema depende da inicialização, então é um módulo que necessariamente precisa estar funcionando. Já um erro de código na Biblioteca C poderia causar um mal funcionamento do sistema como um todo, mesmo após a inicialização, pois, no caso de um problema de acesso de memória indevido, toda a memória do sistema operacional poderá ser comprometido.

Os módulos que são folha do grafo de dependência também representam partes críticas do sistema operacional. Por exemplo, um mal funcionamento no escalonador do gerenciamento de multitarefas pode causar um comportamento onde apenas uma tarefa é executada, impedindo que o sistema funcione normalmente.

FIGURA 9 – DEPENDÊNCIA DOS MÓDULOS DO JAMESOS.



FONTE: Correia (2020)

Para poder dimensionar a quantidade de erros existentes do JamesOS, é proposto a implementação de testes de módulo, para cada módulo do sistema operacional existente. Dessa forma, será possível criar uma visualização do número de erros por módulo, e também, identificar em qual momento o erro ocorre utilizando uma avaliação manual posteriormente.

3.2.1 Detalhes técnicos

A estrutura do JamesSO é composta pelo sistema de inicialização e pelo núcleo do sistema operacional. Essa separação é feita por conta da maneira que o código é compilado.

O código da inicialização é em linguagem *assembly*, e apesar do código estar separado em diversos arquivos, é tratado como apenas um módulo. Por conta disso, no momento da compilação todos os arquivos serão concatenados e é gerado um arquivo *boot_sector.bin*. O formato desse arquivo é *raw* ou cru. Dessa forma, contendo apenas as instruções *assembly* em binário. A compilação desse módulo é feita pelo programa *nasm* (NASM, 2024) da seguinte maneira *nasm boot/boot_sector.asm -f bin -o boot/boot_sector.bin*. O arquivo *boot_sector.asm* é o principal e inclui todos os outros.

O código do núcleo do sistema operacional é compilado de uma maneira diferente. Mesmo que o objetivo também seja gerar um arquivo binário em formato cru, os passos mudam. Já que, além de código em *assembly*, grande parte do JamesOS é por códigos na linguagem de programa C. Um outro detalhe é que o código precisa ser compilado com o endereço base 0x20000. Para atingir esse requisitos, os códigos são compilados inicialmente para o formato *Executable Link Format* (ELF). Os arquivos em C foram compilados da seguinte maneira *i686-elf-gcc -ffreestanding -c -I include -nostdlib -lgcc -w exemplo.c -o exemplo.o* e os arquivos em *assembly nasm exemplo.asm -f elf -o exemplo.o*.

Utilizando essa estratégia, os arquivos com a extensão *.o* são arquivos ELF independentes. Porém, ainda precisam ser passados por um terceiro programa, o *linker* do pacote de ferramentas *GNU Binutils* (GNU, 2024c). Essa ferramenta será executada da seguinte maneira ‘*i686-elf-ld -z muldefs -o kcore/kernel_main.bin -Ttext 0x20000 –oformat binary *.o*‘, sendo **.o*, todos os arquivos gerados. Logo, gerando o segundo binário em format cru *kernel_main.bin*.

Por fim, a execução do JamesOS feita através de uma imagem composta pela concatenação dos binários de inicialização e pelo núcleo do sistema operacional. O arquivo final é o *os-image.bin*.

O QEMU irá tratar o *os-image.bin* como uma imagem de um disquete, já que o JamesOS foi desenvolvido dessa maneira, e irá iniciar normalmente. O QEMU é executado com os seguintes parâmetros *qemu-system-i386 -drive format=raw,file=os-image.bin,index=0,if=floppy -serial stdio*. Como o JamesOS atualmente tem suporte para placa de rede, os parâmetros que envolvem a inclusão do dispositivo foram omitidos, já que os testes não englobam a pilha de protocolos de redes de computadores.

O processo de compilação e execução do JamesOS é automatizado através da ferramenta *make* (GNU, 2024b). O qual é uma ferramenta que controla a geração

de arquivos executáveis e outros arquivos de um programa, utilizando arquivos fonte.

3.3 FRAMEWORK DE TESTE

O *framework* de teste foi desenvolvido utilizando a ferramenta GDB como depurador e irá se conectar com o QEMU através do protocolo *Remote Serial Protocol* (RSP) (GNU, 2024e). A escolha dessa arquitetura foi baseada na facilidade de aplicar a técnica de *Virtual Machine Introspection* (VMI) Payne (2011) através do GDB. Já que a ferramenta disponibiliza uma API na linguagem de programação *python* para criação de *scripts* (GNU, 2024d).

Uma outra funcionalidade do QEMU é o *QEMU Monitor Interface* (QMP) (QEMU, 2024a). O qual possibilita controlar o sistema emulado. E também, oferece a funcionalidade de simular interrupções de dispositivos de entrada e saída, como por exemplo, o teclado. Dessa forma, é possível enviar comandos para apertar e soltar teclas, facilitando nos testes de dispositivo de teclado.

Um dos requisitos do *framework* de testes é a utilização dos símbolos DWARF. Já que, os testes acessam variáveis e funções através do nome. Porém, as duas imagens intermediárias e a imagem final estão em um formato *raw*. Por conta disso, a imagem *kernel_main.bin* não é suficiente para os testes, já que não contém nenhum símbolo indicando nomes de funções e variáveis, por exemplo.

A adição desses símbolos em um arquivo *raw* não é possível. E também, um dos objetivos desse conjunto de testes é que o código do sistema operacional não seja alterado. Então, para gerar um arquivo contendo os símbolos para ser processado pelo depurador, o processo de compilação sofreu uma alteração.

A primeira alteração foi a adição da *flag -g gdb3* no comando do *i686-elf-gcc*. Com isso, os arquivos ELF intermediários citados anteriormente terão os símbolos DWARF. Porém, essa alteração não é suficiente, já que, o processo atual do *linker* tem como saída um arquivo *raw*, fazendo com que os símbolos sejam descartados. A segunda alteração foi feita para resolver esse problema. Para isso, no momento em que o *linker* é executado pelo *make*, uma execução adicional foi necessária. Assim, a flag `-oformat binary` foi alterada para `-oformat elf`, tendo como saída o arquivo *kernel_main.bin.elf*. Por fim, duas imagens do núcleo do sistema operacional foram geradas, uma no formato *raw* e a segunda no formato ELF.

A execução do QEMU também foi modificada, já que os servidores RSP e QMP precisam ser iniciados. Para isso, as *flags* `-s` e `-qmp tcp:0.0.0.0:4444,server=on,wait=off` são adicionadas no comando de execução, iniciando os dois servidores, respectivamente. O primeiro servidor será acessado pelo GDB e o segundo será acessado através dos *scripts* de teste diretamente. Por fim, um terceiro parâmetro adicional, `-S`, é

para o QEMU não executar a CPU no momento da inicialização. A execução começará após um comando de continuação ser enviado ao QEMU.

3.3.1 Estrutura do projeto

A estrutura de diretórios do projeto é a seguinte:

```
tests
├── james_test.py
├── james_utils.py
├── boot
│   └── test_bootloader.py
├── cpu
│   └── test_isr.py
├── drivers
│   ├── test_keyboard.py
│   └── test_vga.py
├── libc
│   ├── test_bitmap.py
│   ├── test_kheap.py
│   ├── test_mem.py
│   ├── test_minmax.py
│   ├── test_paging.py
│   └── test_string.py
└── multitasking
    └── test_multitaking.py
```

O arquivo *james_test.py* é o programa *python* que irá iniciar todo o processo de teste. O programa irá executar cada arquivo de teste de uma maneira iterativa. Dessa forma, para cada arquivo que o nome tenha a seguinte expressão regular */test_.*py/g*, o sistema operacional será compilado e o QEMU é executado. A partir desse momento, o programa GDB é executado e o arquivo de teste é importado para o escopo do depurador.

O código do *james_test.py* é representado pela FIGURA 10. A classe *JamesTest* é responsável por implementar funções auxiliares para a inicialização dos testes. Logo, tem a funções para iniciar e terminar o processo do QEMU, assim como a função de inicialização do processo do GDB.

O programa vai buscar por todos os arquivos de teste, e para cada arquivo vai instanciar um objeto da classe *JamesTest*, iniciar o QEMU e iniciar o GDB com o arquivo de teste como parâmetro. Após a execução dos testes, o GDB irá finalizar. Nesse ponto o resultado dos testes é coletado pelo *framework*. Cada teste executado irá escrever

no arquivo de saída do *linux*, o *stdout* (linux.die.net, 2024b). Então, baseado em um padrão estabelecido para indicar caso o teste foi positivo ou negativo, os resultados são armazenados em um dicionário em memória. Por fim, por questões de visualização, os resultados são mostrados na tela de uma maneira organizada por arquivo de teste e separados pelos resultados.

A biblioteca *james_utils.py* é composta pela classe JamesUtils e pode ser visualizado na FIGURA 11. O contexto de execução dessas funções é do GDB, por conta disso, esse código só poderá ser executado pelo próprio depurador, não sendo possível executar independente.

Como pode ser visualizado no código do *james_test.py*, o depurador é executado com o parâmetro *–command={test_file}*, o que faz o arquivo de teste ser executado automaticamente pela ferramenta.

Todo arquivo de teste deverá respeitar o padrão descrito na FIGURA 12. Dessa forma, a função *GdbInit* sempre é executada. Essa função tem a responsabilidade de configurar o GDB para que seja executado da maneira esperada. Com isso, configurando a arquitetura para *i386*, se conectando com o servidor iniciado pelo QEMU, e iniciar o sistema operacional para que execute a parte da inicialização e pare de executar na primeira função do núcleo do SO. Além de configurar o *log* da ferramenta e desabilitar qualquer interação manual. Por fim, após os testes, é mandatório executar a função de saída do depurador, já que o *framework* apenas continua a execução quando o processo do GDB é finalizado.

Diversos testes foram omitidos no texto, já que muitos testes apenas aplicam as técnicas demonstradas, sem ter nenhum tipo de problema. Com isso, gerando repetições de testes desnecessárias. Por fim, o código do *framework* foi adicionado na plataforma *github* e todos os grafos de fluxo de controle estão disponíveis no apêndice.

3.3.2 Portas de entrada e saída

As portas de entrada e saída são referentes à um endereço específico no barramento dos processadores x86 (OSDev, 2024f). Esse barramento é utilizado para comunicação com dispositivos externos e é uma alternativa para o *Direct Memory Access* (DMA).

O JamesOS implementa um módulo de portas de entrada e saída. Esse módulo é utilizado para a configuração de *drivers* de dispositivos, como o *Programmable*

FIGURA 10 – ARQUIVO JAMES_TEST.PY.

```
class JamesTest():
    def __init__(self, makefile = 'Makefile'):
        self.makefile = makefile

    def QemuStart(self):
        cwd = os.getcwd()
        os.chdir('../')

        clean = subprocess.run(['/usr/bin/make', 'clean'],\
                stdout = subprocess.PIPE,\
                stdin = subprocess.PIPE,\
                stderr = subprocess.PIPE)
        clean.check_returncode()

        build = subprocess.run(['/usr/bin/make', 'build'],\
                stdout = subprocess.PIPE,\
                stdin = subprocess.PIPE,\
                stderr = subprocess.PIPE)
        build.check_returncode()

        self.qemu = subprocess.Popen(['/usr/bin/make', 'run'],\
                stdout = subprocess.PIPE,\
                stdin = subprocess.PIPE,\
                stderr = subprocess.PIPE)

        os.chdir(cwd)

    def QemuStop(self):
        os.kill(self.qemu.pid, signal.SIGTERM)

    def GdbStart(self, test_file):
        self.gdb = subprocess.Popen(\
                ['/usr/bin/gdb', f'--command={test_file}', '--quiet'],\
                stdout = subprocess.PIPE,\
                stdin = subprocess.PIPE,\
                stderr = subprocess.PIPE)


test_results = {}
for test_file in glob.glob('*/test_*.py'):
    test = JamesTest()
    test.QemuStart()

    test.GdbStart(test_file)
    test.gdb.wait()

    raw_output, _ = test.gdb.communicate()
    lines = raw_output.split(b'\n')
    successes = [line for line in lines if line.startswith(b'\x1b[92m')]
    fails = [line for line in lines if line.startswith(b'\x1b[31m')]

    test_results[test_file] = {
            'successes': successes,
            'fails': fails
            }

    test.QemuStop()

for test_file, results in test_results.items():
    print(f'<><><> {test_file} <><><>')
    for result, messages in results.items():
        print(f'    {result} ({len(messages)})')
        for message in messages:
            print(f'    {message.decode()}')
```

FONTE: Próprio Autor

FIGURA 11 – ARQUIVO JAMES_UTILS.PY.

```python
class JamesUtils():
    def TestMemoryRegion(self, expected, start_memory_address):
        memory_array = gdb.inferiors()[0].read_memory(\
                start_memory_address, len(expected)).tobytes()
        for i in range(len(expected)):
            if expected[i] != memory_array[i]:
                return False
        return True

    def TestDiskLoadOnMemory(self, function, file, offset):
        address = gdb.parse_and_eval(f'{function}')
        length = os.path.getsize(f'{file}') - int(offset)

        os_file = open(file, 'rb')
        os_bytes = os_file.read()[0x200:]

        return self.TestMemoryRegion(os_bytes, address)

    def GdbInit(self, symbol_file = '../kcore/kernel_main.bin.elf'):
        gdb.execute('set arch i386')
        gdb.execute(f'file {symbol_file}')
        gdb.execute('target remote localhost:1234')
        gdb.execute('break entry')
        gdb.execute('set logging file gdb_log.txt')
        gdb.execute('set logging overwrite on')
        gdb.execute('set logging enabled on')
        gdb.execute('set confirm off')
        gdb.execute('set pagination off')
        gdb.execute('continue')

    def GdbStop(self):
        gdb.execute('quit')

    async def QmpCommand(self, command, args = {}):
        while True:
            try:
                qmp = qemu.qmp.QMPClient('jamesOS')
                await qmp.connect(('127.0.0.1', 4444))
                res = await qmp.execute(command, args)
                await qmp.disconnect()
                return res
            except qemu.qmp.protocol.ConnectError:
                continue
            except Exception as e:
                print(e)
                break
```

FONTE: Próprio Autor

Interrupt Controller (PIC) OSDev (2024a) e o *Programmable Interval Timer* (PIT) OSDev (2024k). As funções podem ser utilizadas tanto para escrita quanto para leitura de dados.

Esse módulo não foi testado, pois não foi possível desenvolver uma maneira eficiente de teste. Então, para todos o contexto de teste, foi considerado que o módulo funciona como o esperado.

FIGURA 12 – PADRÃO DE TESTE.

```python
import gdb
import os
import sys
sys.path.append(os.path.dirname('..'))
import james_utils

utils = james_utils.JamesUtils()
utils.GdbInit()


# INÍCIO DOS TESTES

# FIM DOS TESTES


utils.GdbStop()
```

FONTE: Próprio Autor

3.3.3 Técnicas de teste

Os testes do sistema operacional foram feitos utilizando um conjunto de técnicas de introspecção de máquina virtual. Essa introspeção é para a análise dos valores dos registradores, como também para validar valores de variáveis, tanto globais quanto locais. Porém, também foi utilizado a validação do conteúdo da memória de uma maneira direta, utilizando apenas o endereço alvo. Uma outra técnica aplicada foi a de injeção de código, a qual consiste em alterar a memória do sistema operacional em tempo de execução, inserindo os *bytes* das instruções e alterando o fluxo de execução para a posição de memória modificada.

3.3.4 Teste da inicialização do sistema

A inicialização do sistema operacional JamesOS tem dois objetivos. O primeiro é carregar o código do sistema operacional para a memória e o segundo é mudar o sistema operacional para o modo protegido ou de 32 bits.

Para validar a inicialização do sistema operacional, uma comparação é feita com o conteúdo do endereço de memória da função *_start*, já que é a primeira função do SO, com o conteúdo da imagem original. Como o código começa a partir do *byte* 512, por conta dos 512 primeiros serem do código de inicialização, um terceiro parâmetro precisa ser passado para a função. Esse último parâmetro irá ignorar os *bytes* iniciais.

Já para a validação, se a transição do modo real para o modo protegido foi executada, basta validar se o bit menos significativo do registrador cr0 é 1.

O código de inicialização é executado automaticamente pelo *framework* de testes em toda execução. Essa abordagem foi escolhida, pois o código da inicialização pode ser tratado como apenas uma função.

A maneira como os testes foram feitos podem ser vistos na FIGURA 13. O código do padrão do arquivo de teste foi omitido da imagem.

FIGURA 13 – TESTE DA INICIALIZAÇÃO.

```
## BOOTLOADER
assert(utils.TestDiskLoadOnMemory('_start', '../os-image.bin', 512))
print(f'\033[92m[+] BOOTLOADER sistema operactional carregado corretamente.\033[0m')

cr0 = gdb.parse_and_eval('$cr0')
assert((cr0 & 0x1) == 0x1)
print(f'\033[92m[+] BOOTLOADER sistema operactional executando no modo protegido.\033[0m')
```

FONTE: Próprio Autor

3.3.5 Testes das interrupções

O módulo de interrupções do JamesOS é composto por três submódulos, o *Interrupt Service Routines* (ISR) OSDev (2024h), O *Interrupt Descriptor Table* (IDT) OSDev (2024g) e o tratador de interrupções.

O módulo ISR tem a respo sabilidade de configurar o *Programmable Interrupt Controller* (PIC) para tratar as exceções da CPU (OSDev, 2024e). Já o IDT tem a responsabilidade de disponibilizar funções auxiliares para configuração das interrupções estabelecidas pelo núcleo do JamesOS, como por exemplo o *driver* de teclado. Por fim, o tratador de interrupções é uma interface entre o primeiro momento que as interrupções são geradas e as funções do ISR e do IDT. Tem como responsabilidade salvar e restaurar o estado dos registradores, antes e depois da execução das funções que tratam a interrupção.

A primeira parte dos testes desse módulo foi para a validar a configuração dos funções de interface. Dessa forma, executando a função *isr_install* e validando se as estruturas do ISR e do IDT foram preenchidas corretamente. Os testes foram feitos gerando interrupções de software, utilizando injeção de código. Para isso, executando a instrução *int #NUM* e validando a chamada das funções que tratam interrupções, tanto para o ISR quanto para o IDT.

3.3.6 Teste da biblioteca C

A biblioteca C compõe uma parte muito importante do sistema operacional. O objetivo dessa implementação é prover funções para manipulação de *strings*, gerenciamento de memória, paginação e também funções auxiliares como a manipulação de estruturas como o *bitmap* e comparações de inteiros utilizando o *minmax*.

A implementação de uma biblioteca C pode ser baseada em especificações como a ISO/IEC 9899 e INCITS/ISO/IEC 9899-2011 (OSDev, 2024d). Essa implementação requer uma alta quantidade de tempo. Por conta disso, é viável utilizar implementações públicas como a *musl* (musl.libc.org). Porém, normalmente é necessário criar uma camada de portabilidade entre a implementação utilizada e o sistema operacional. Para o sistema JamesOS nenhum padrão em particular foi implementado, já que não é um fator necessário, além de diminuir a complexidade do código.

A biblioteca C do JamesOS é composta pelos módulos: *kheap*, *bitmap*, *mem*, *strings*, *paging* e *minmax*.

Os módulos *paging* e o *kheap* são responsáveis por operações em relação à memória do SO. A paginação é feita pelo *paging*, logo é responsável pelo gerenciamento de memória virtual. Já o *kheap* é o gerenciador de memória *heap* do JamesOS, o qual tem a função de providenciar funções para alocar e desalocar memória na região da *heap*. A *heap* é a memória alocada de maneira dinâmica no momento da execução do SO (Silberschatz et al., 2018).

Os módulos *strings* e *mem* são responsáveis por fazer operações em arranjos ou *arrays*. O primeiro contém funções relacionadas à operações em *strings*, por exemplo converter um número inteiro para *string*, inverter as posições e também calcular o tamanho. Já o *mem* é para operações genéricas de cópia, comparação e movimentação de uma posição de memória em específico.

Já o módulo *bitmap* tem o objetivo de providenciar operações em uma implementação da estrutura *bitmap*, que também é conhecida como *bit array*. Essa estrutura é responsável por criar um arranjo de *bits*. Dessa forma, simulando cada posição do *array* como se fosse um *bit*.

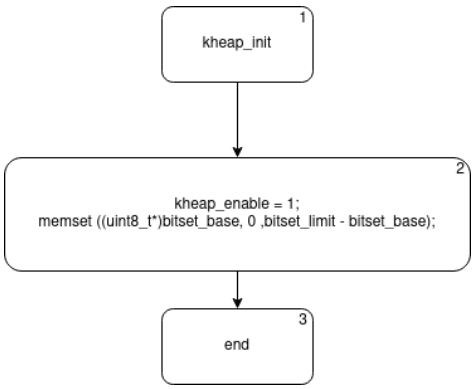
Por fim, o *minmax* tem a responsabilidade comparar inteiros. Logo, implementa funções para comprar dois inteiros e retornar qual é o maior ou o menor.

3.3.6.1 KHeap

A inicialização desse módulo tem o GFC que pode ser visualizado na FIGURA 49. Então, para testar essa função, basta executar, já que todos os caminhos irão ser percorridos obrigatoriamente. Por fim, para validar a execução da função, a variável *kheap_enabled* precisa ser 1 e o *bitset* precisa estar completamente zerado. O *bitset* é uma estrutura *bitmap* e para os testes do *kheap* foi assumido que o funcionamento da estrutura está correto.

A parte da alocação é mais complexa e existem diversos caminhos que podem ser seguidos e que podem ser visualizados na FIGURA 52, mas eventualmente todos

FIGURA 14 – GFC DA FUNÇÃO KHEAP_INIT.



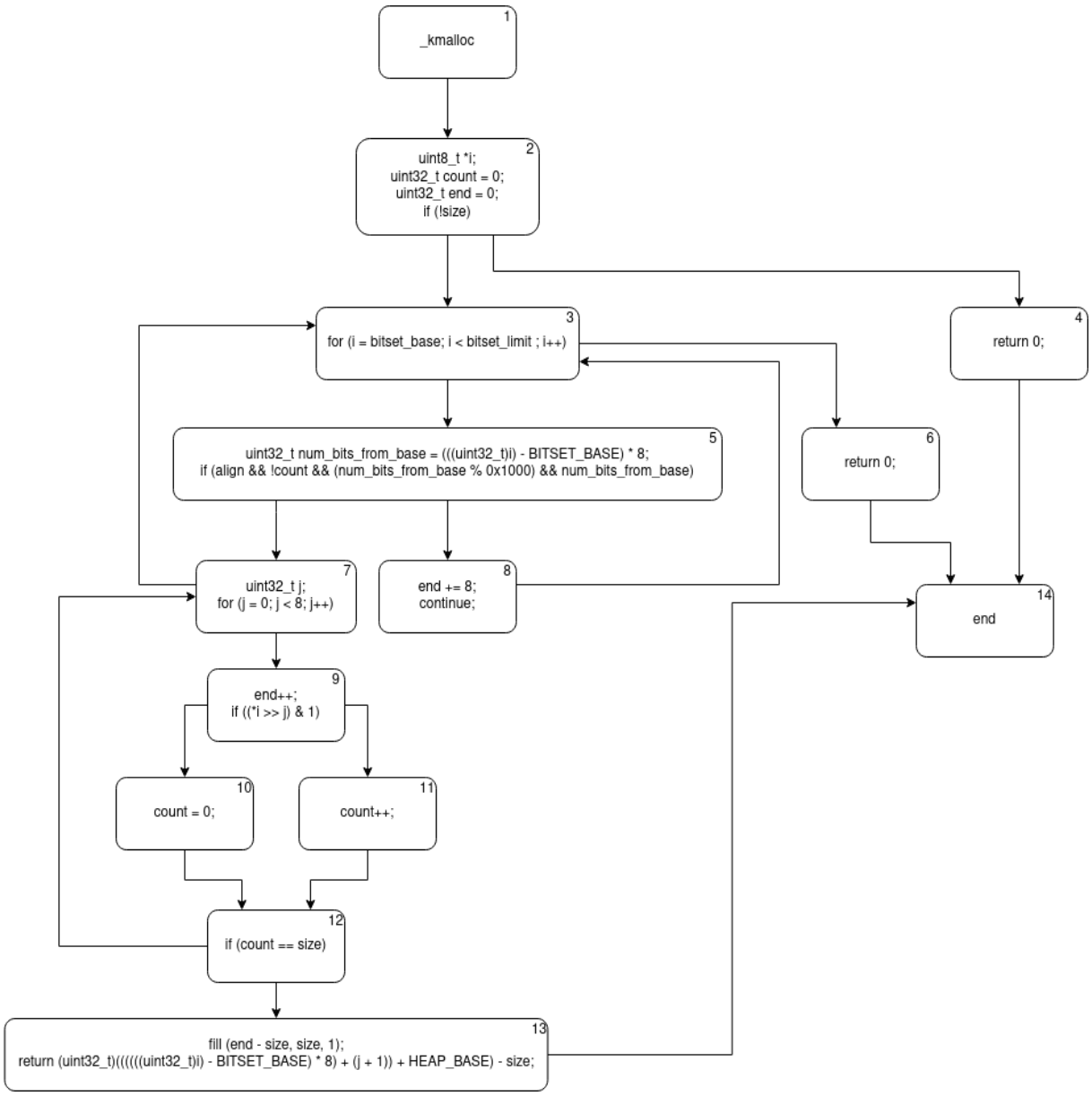
FONTE: Próprio Autor

os nós serão executados, mesmo que a mesma função seja executada mais de uma vez. A primeira execução foram com os parâmetros *_kmalloc(0, 0)*, a segunda *_kmalloc(1, 0)* e a última *_kmalloc(1, 1)*. Os caminhos no GFC foram [1, 2, 4, 14], [1, 2, 3, 5, 7, 9, 11, 12, 13, 14] e [1, 2, 3, 5, 7, 9, 10, 12, 9, 11, 12, 13, 14] respectivamente. Porém no terceiro caso, o nó 8 deveria ter executado e não foi, já que existe um erro no código, e que foi identificado através dos testes. Para verificar o resultado da função, o endereço de retorno é validado.

A execução e validação da função *_kheap* pode ser visualizada pela FIGURA 16. A primeira linha de código equivale a execução do comando *call _kmalloc(0, 0)* dentro do GDB. Esse comando tem a função de executar a função *_kmalloc* com dois inteiros como parâmetros, para isso o GDB lida com a *stack* de maneira automática.

Para a função kfree, apenas uma execução é o suficiente para passar por todos os caminhos. E para a validação, foi utilizada a técnica de introspeção de memória para validar o *bitset*.

FIGURA 15 – GFC DA FUNÇÃO _KMALLOC.



FONTE: Próprio Autor

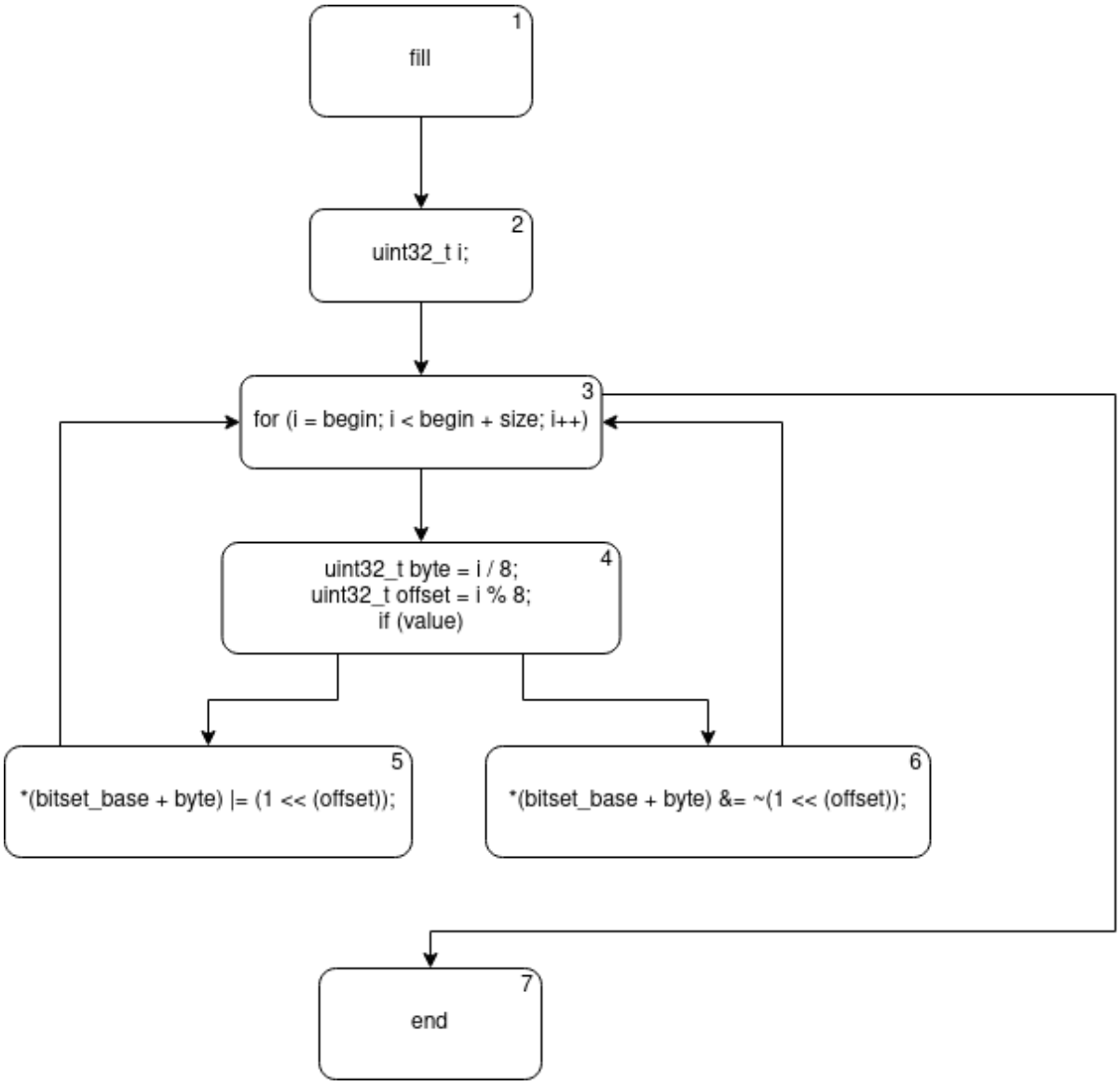
FIGURA 16 – TESTE DA FUNÇÃO _KMALLOC.

```
1 kmalloc = gdb.parse_and_eval('_kmalloc(0, 0)')
2 assert(kmalloc == 0x0)
3 print('\033[92m[+] KHEAP kmalloc(0, 0) executado corretamente.\033[0m')
```

FONTE: Próprio Autor

Por fim, a função *fill* do *KHeap* foi testada. Essa função precisou de várias execuções para que todos os nós do GFC fossem executados. As execução foram *fill(0, 12, 1)* e *fill(4, 4, 0)*. E os caminhos percorridos foram [1, 2, 3, 4, 5, 3, 7], [1, 2, 3, 4, 6, 3, 7].

FIGURA 17 – GFC DA FUNÇÃO FILL.



FONTE: Próprio Autor

Os resultados dos testes do módulo do *kheap* podem ser visualizados na FIGURA 18

FIGURA 18 – RESULTADOS DOS TESTES DO *KHEAP*.

```
<><><> libc/test_kheap.py <><><>
    successes (7)
    [+] KHEAP iniciado corretamente.
    [+] KHEAP fill(0, 12,1) executado corretamente.
    [+] KHEAP fill(4, 4, 0) executado corretamente.
    [+] KHEAP fill(0, 0, 0) executado corretamente.
    [+] KHEAP kmalloc(0, 0) executado corretamente.
    [+] KHEAP kmalloc(1, 0) executado corretamente.
    [+] KHEAP kfree(1175552, 1) executado corretamente.
    fails (1)
    [-] KHEAP kmalloc(1, 1) executado com erro.
```

FONTE: Próprio Autor

3.3.6.2 Paging

O módulo *paging*, como o *kheap*, também tem uma função de inicialização. Porém, a paginação depende de um outro módulo, o de interrupções. E também, por problemas relacionados à depuração, o comando *call* do GDB não poderá ser utilizado. Já que, nesse caso, a paginação não é ativada pelo processador.

O código do inicialização está na FIGURA 19. O módulo *kheap* é utilizado para alocar memória na *heap* de 8 posições. Nesse caso, o GDB irá escrever instruções de máquina na memória, para poder executar instruções arbitrárias através da técnica de injeção de código. Essas instruções serão para ativar as interrupção do sistema através da instrução *sti* e executar a função *paging_init* manualmente. O segundo requisito é iniciar o módulo de interrupções da CPU, através da função *isr_install*.

A sequência de execução será chamar a função *isr_install*, ativar a as interrupções e chamar *paging_init*. Por fim, a validação da execução é executada normalmente.

A mesma técnica de injeção de código também foi utilizada para o teste de falha de página, a qual é uma exceção do processador. Essa exceção é acionada quando uma parte da memória virtual não mapeada é acessada em algum momento do código. Quando isso ocorre, uma função que foi definida no momento da inicialização é chamada. A FIGURA 20 exemplifica o teste. As instruções injetadas equivalem às linhas de código em C *int *x = 0xf01000* e **x = 8*.

3.3.6.3 Bitmap

Como demonstrado anteriormente, o módulo do *bitmap* implementa uma estrutura a qual simula um *array* onde cada posição corresponde a um bit. Para isso, é

FIGURA 19 – TESTE DA INICIALIZAÇÃO DA PAGINAÇÃO.

```
gdb.execute('call kheap_init()')
call_paging_init = gdb.parse_and_eval('kmalloc_u(8)')
call_bytes = b'\xfb' # sti
call_bytes += b'\xb8\x38\x0b\x02\x00' # mov eax, 0x00020b38
call_bytes += b'\xff\xd0' # call eax
gdb.inferiors()[0].write_memory(call_paging_init, call_bytes)

gdb.execute('call isr_install()')
gdb.execute(f'set $pc = {call_paging_init}')
gdb.execute('si')

gdb.execute(f'break *{call_paging_init + len(call_bytes)}')
gdb.execute('continue')
page_enabled = gdb.parse_and_eval('page_enabled')
tables_physical = gdb.parse_and_eval('&kernel_directory->tables_physical')
cr3 = gdb.parse_and_eval('$cr3')
assert(page_enabled == 0x1)
assert(cr3 == tables_physical)
print(f'\033[92m[+] PAGING paging_init() executado corretamente.\033[0m')
```

FONTE: Próprio Autor

FIGURA 20 – TESTE DA EXCEÇÃO FALHA DE PÁGINA.

```
fault_bytes = b'\xfb\xc7\x45\xf4\x00\x10\xf0\x00\x8b\x45\xf4\xc7\x00\x08\x00\x00\x00'
run_fault = gdb.parse_and_eval(f'kmalloc_u({len(fault_bytes)})')
gdb.inferiors()[0].write_memory(run_fault, fault_bytes)
gdb.execute(f'set $pc = {run_fault}')
gdb.execute('si')
gdb.execute('b *0x00020f08') #end page_fault_handler
print(f'\033[92m[+] PAGING page_fault_handler() chamado corretamente.\033[0m')
```

FONTE: Próprio Autor

necessário ter operações básicas de escrita e leitura.

Além das operações citadas, a implementação contém outras funções auxiliares. Como por exemplo a função *bitmap_find_sequence*, que tem como responsabilidade encontrar uma sequência contígua de um valor especificado, seja ele 1 ou 0. O teste dessa função pode ser visualizado na FIGURA 21.

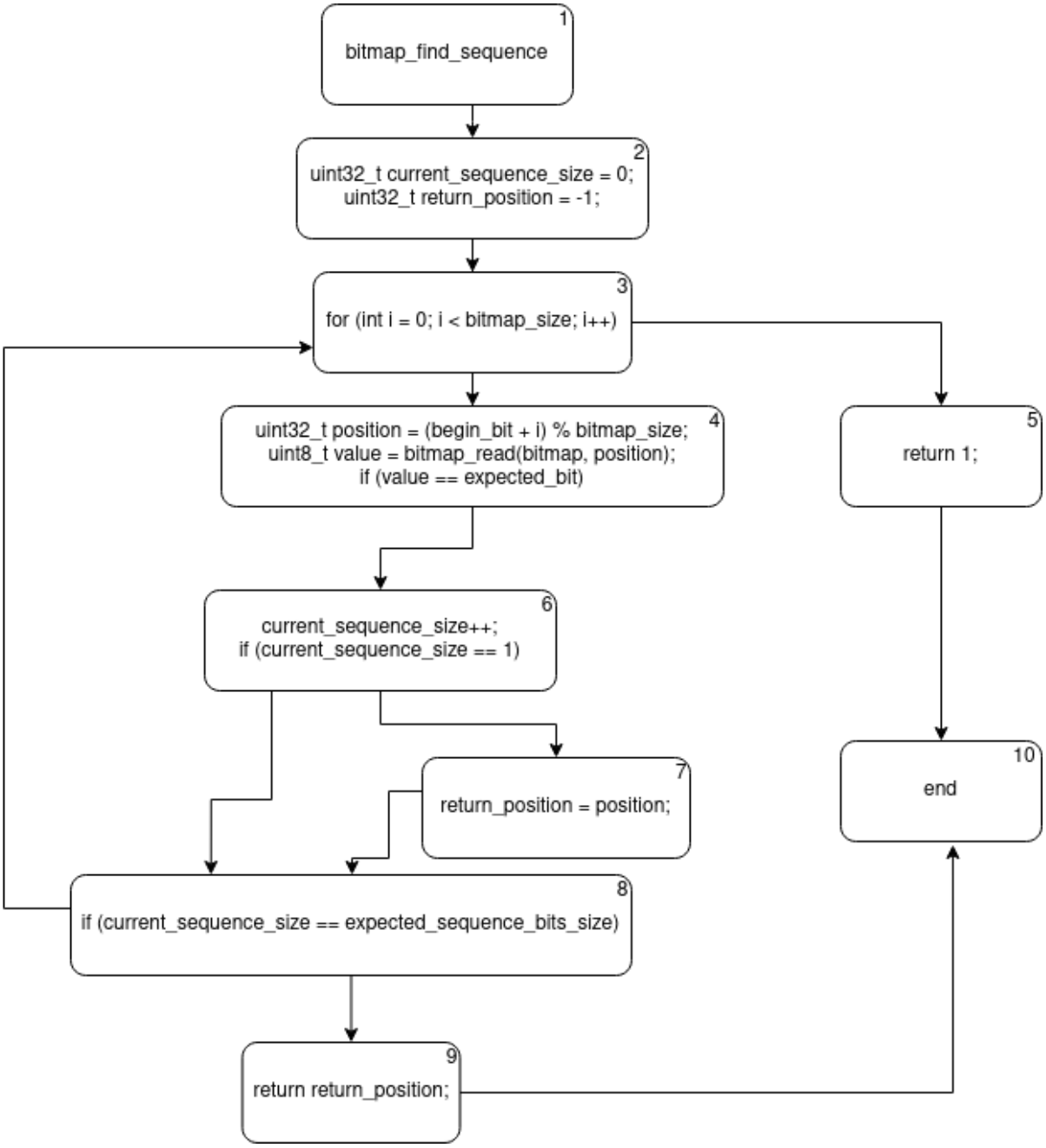
O teste da função retornou um erro. Analisando o GFC da função e comparando com o caminho esperado, é observado que o caminho está correto. A FIGURA 22 representa o GFC, e o caminho percorrido pelo teste foi [1, 2, 3, 4, 6, 7, 8, 3, 5, 10].

FIGURA 21 – TESTE DA FUNÇÃO BITMAP_FIND_SEQUENCE.

```
start_position = gdb.parse_and_eval(f'bitmap_find_sequence({bitmap}, 40, 8, 5, 1)')
try:
    assert(start_position == -1)
    print(f'\033[92m[+] BITMAP bitmap_find_sequence({bitmap}, 40, 8, 5, 1) executado corretamente.\033[0m')
except:
    print(f'\033[31m[-] BITMAP bitmap_find_sequence({bitmap}, 40, 8, 5, 1) executado com erro.\033[0m')
```

FONTE: Próprio Autor

FIGURA 22 – GFC DA FUNÇÃO BITMAP_FIND_SEQUENCE.



FONTE: Próprio Autor

3.3.7 Teses dos drivers

Os *drivers* de dispositivos testados foram do VGA e de teclado.

3.3.7.1 VGA

O módulo do VGA é responsável por criar funções para a escrita na tela, utilizando a técnica de Direct Memory Access (DMA). Com isso, os dados que são mostrados na tela são escrito na memória diretamente, começando no endereço 0xb8000 até 0xb8fa0. Essa porção da memória pode ser tratada como um *array* de 2 bytes por posição. O primeiro byte representa o caractere em ASCII e o segundo tanto a cor do caractere quanto a cor de fundo.

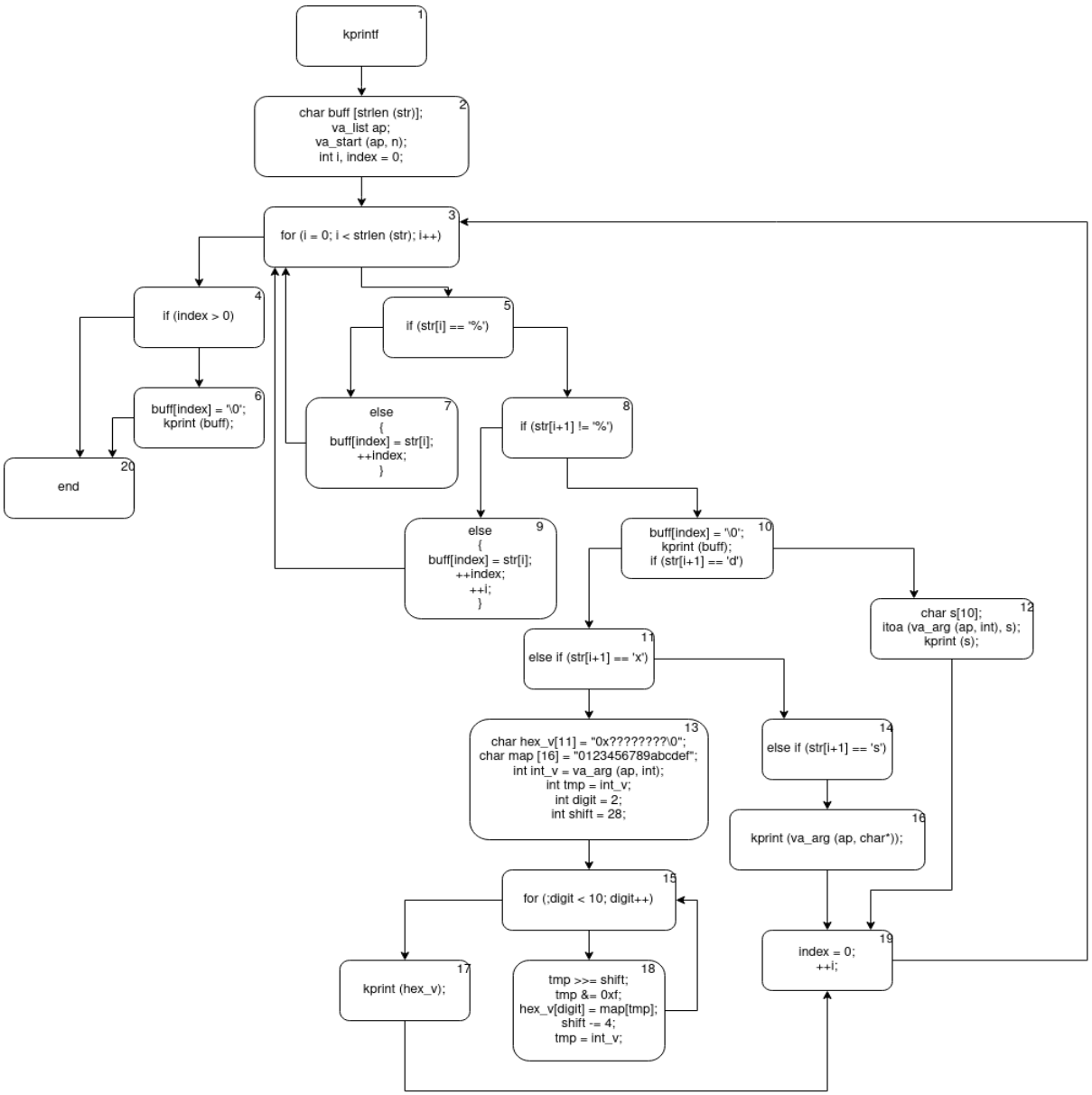
Uma função principal do módulo do VGA é o *kprintf*. Essa função tem o objetivo de mostrar na tela uma *string* com o caracteres de formatação, sendo eles %s, %d e %x. O primeiro é para mostrar uma *string*, o segundo um inteiro em decimal e o terceiro um inteiro em hexadecimal. Dessa maneira, auxiliando no processo de testes manuais e depuração. O GFC dessa função está representado na FIGURA 23. Apesar de ser um grafo com múltiplos caminhos, com apenas uma chamada todos os caminhos podem ser satisfeitos utilizando uma chamada, passando como parâmetro uma *string* com todos os modificadores. Já para a validação do resultado da função, basta validar a região de memória que foi escrita.

O módulo também implementa funções auxiliares para poder imprimir na tela utilizando qualquer posição indicada. A função que implementou essa funcionalidade é a *kprint_at* e tem o GFC representado pela FIGURA 24. Alguns erros relacionados ao VGA podem ser identificados visualmente, e o erro causado por essa função é um desses. Nesse caso, quando a função *kprint_at* chega no fim do *array*, o comportamento esperado é simular a rolagem da tela. Dessa forma, todas as linhas serão movidas para a de cima e a primeira descartada. Porém, quando o teste foi executado, foi observado que o primeiro caractere é ignorado. O caminho percorrido pelo teste foi [1, 2, 4, 5, 4, 5, 6, 7, 8], e também é o caminho esperado.

3.3.7.2 Teclado

O módulo do teclado tem a responsabilidade de tratar as interrupções acionadas pelo teclado. Para isso, irá registrar uma função no vetor de interrupções da CPU, da mesma forma que foi feita nó módulo de paginação. No JamesOS, o tratamento das interrupções é apenas utilizar as portas de entrada e saída do processador para recuperar o código da interrupção. O código pode representar uma tecla pressionada

FIGURA 23 – GFC DA FUNÇÃO KPRINTF

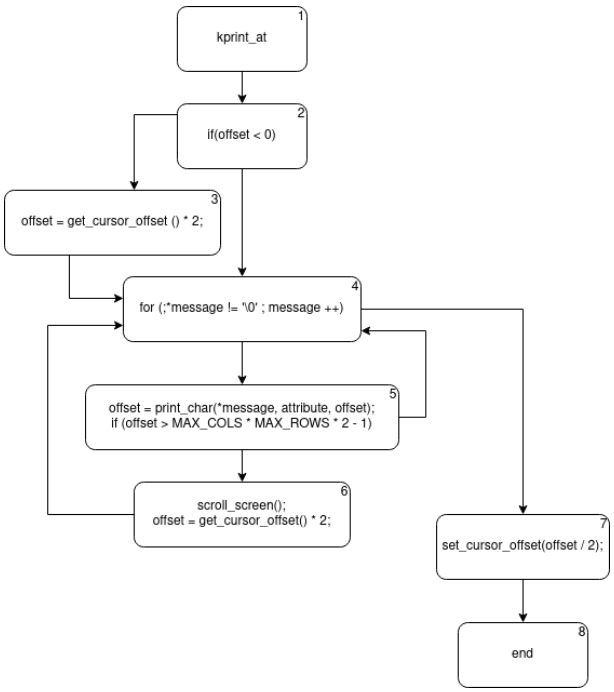


FONTE: Próprio Autor

ou uma tecla liberada. Por exemplo, caso o código da tecla for 0x1e, significa que a tecla ’a’ minúsculo foi pressionada, e no caso de ser 0x9e, ou 0x1e + 0x80, significa que a tecla ’a’ minúsculo foi liberada.

O código de teste pode ser visto na FIGURA 25. A geração das interrupções foram feitas utilizando a interface QMP, a qual foi descrita anteriormente. Para isso, basta se conectar com o QMP e utilizar o comando *send-key* especificando o código da tecla. O tempo em que a tecla ficará pressionada é de 100 milissegundos por padrão. Porém, um requisito para que as interrupções sejam processadas pela CPU, o sistema não pode estar pausado em nenhum ponto de interrupção ou *breakpoint*. Por conta disso, foi necessário direcionar a execução do programa para uma instrução *jmp $*,

FIGURA 24 – GFC DA FUNÇÃO KPRINT_AT



FONTE: Próprio Autor

o que representa um loop infinito. A técnica de injeção de código não foi necessária nesse caso, já que a instrução já existe no código.

FIGURA 25 – CÓDIGO DA FUNÇÃO DE TESTE DO TECLADO

```
gdb.execute('set $pc = 0x20005')
gdb.execute('break *0x2108a')

def test_key(key, code, expected_press, expected_release):
    cont = threading.Thread(target = gdb.execute, args = ('continue',))
    cont.start()
    asyncio.run(utils.QmpCommand('send-key', {
            'keys': [
                {'data': f'{key}', 'type': 'qcode'}
            ]
        }))
    cont.join()

    expected_press = b''.join([b.encode()+b'\x0f' for b in expected_press])
    assert(utils.TestMemoryRegion(expected_press, vga_address))
    print(f'\033[92m[+] KEYBOARD keyboard_callback executado corretamente (press {key}).\033[0m')

    gdb.execute('continue')
    expected_release = b''.join([b.encode()+b'\x0f' for b in expected_release])
    assert(utils.TestMemoryRegion(expected_release, vga_address + (80 * 2)))
    print(f'\033[92m[+] KEYBOARD keyboard_callback executado corretamente (release {key}).\033[0m')

test_key('a', 0x1e, 'scan_code: 30 (a)', 'scan_code: 158 (a released)')
gdb.execute('call clear_screen()')
```

FONTE: Próprio Autor

Com o sistema preparado para receber interrupções, o teste pode ser realizado normalmente. Logo, após enviar o comando para pressionar e liberar uma tecla utilizando o QMP, basta verificar se o que foi escrito na tela está correto.

3.3.8 Testes de multitarefas

O módulo de multitarefa implementa o pseudo paralelismo (Tanenbaum; Bos, 2022) no JamesOS. Dessa maneira, gerenciar a criação, término e execução das tarefas. Cada tarefa tem um contexto próprio, o qual é definido pelo conjunto de registradores, diretório de página e o estado atual. Também contém informações extras como o nome e um identificador único.

Da mesma forma como os outros testes, cada função do módulo foi testada separadamente. Porém, os fluxos que envolvem trocas de contexto, ou seja, uma nova tarefa é executada, a execução das funções são interrompidas antes do fim. Por exemplo, a função *block_task* executa uma outra função chamada *scheduler*, a qual faz a troca de contexto citada. Dessa forma, uma nova tarefa é executada, impedindo a finalização da tarefa inicial.

Um outro problema existente está relacionado à uma limitação do GDB. Quando o comando *call* é executado e um *breakpoint* é atingido, a API retorna uma exceção e o GDB encerra a função, alterando para o estado da pilha. Com isso, não é possível continua a função do estado de onde parou.

Para contornar esse problema, a estratégia é exemplificada pela FIGURA 26. O objetivo é colocar um *breakpoint* no momento mais avançado possível, tratar a exceção no código de teste, e por fim, validar tanto o estado da tarefa em execução, quanto o nome.

FIGURA 26 – CÓDIGO DA FUNÇÃO DO AGENDADOR

```
1 gdb.execute('break *0x00022fa7')
2 try:
3     gdb.execute(f'call scheduler()')
4 except gdb.error as e:
5     pass
6 next_task = gdb.parse_and_eval('next')
7 assert(next_task['state'] == gdb.parse_and_eval('RUNNING'))
8 assert(utils.TestMemoryRegion(b'CHUAZNEGUER', next_task['pname'].address))
9 print(f'\033[92m[+] MULTITASKING scheduler() executado corretamente.\033[0m')
```

FONTE: Próprio Autor

3.4 CONSIDERAÇÕES FINAIS

Os testes feitos no JamesOS identificaram 3 *bugs*, como foi demonstrado. Um no módulo do VGA, outro no *kheap* e um terceiro no *bitmap*. Utilizando a ferramenta *cloc* linux.die.net (2024a) para conter as linhas de código, somando os arquivos de cabeçalho e de código, tanto para a linguagem C quanto para o *assembly*, totalizam 2434 linhas. O número de *bugs* por linhas de código tem a relação de aproximadamente 0.0012, logo a densidade de *bugs* encontrada foi de 1.23 a cada mil linhas de código (kSLOC).

Os testes utilizaram a técnica de avaliar todos os caminhos em um GFC. Apesar de ser uma técnica que não leva em consideração a definição e o uso das variáveis, e também, não utilizar casos de teste com valores limite para uma maior eficiência, ainda assim foi possível demonstrar erros de código.

Os erros encontrados nos testes precisam ainda ser analisados manualmente para a correção. Porém, o código de teste indica a função com erro, o que acelera o processo de análise.

O *framework* de testes desenvolvido conseguiu demonstrar como as técnicas de teste podem ser aplicadas em momentos iniciais de um sistema operacional. Com isso, foi possível testar o código desde o processo de inicialização. E também, demonstrou como as funções podem ser testadas de uma maneira isolada e em tempo de execução.

Diferente dos testes realizados no HelenOS, cada módulo foi testados em uma nova instância do QEMU Sucha (2013). Essa abordagem é considerada não performática, porém, consegue fornecer um isolamento maior entre cada teste realizado. Por exemplo, o módulo de teste do módulo de multitarefa requer a inicialização do módulo de interrupções, porém o do *kheap* não. Caso ambos os testes fossem executados na mesma instância, o isolamento dos módulos seria quebrado.

A utilização da ferramenta GDB em conjunto com o QEMU para a depuração demonstra alguns impedimentos para alguns momentos do teste. Como demonstrado anteriormente, a paginação da CPU não é ativada caso o código executado esteja no contexto do comando *call* do depurador. Dessa forma, criando a necessidade de utilizar a injeção de código dinâmico na memória do SO.

4 CONCLUSÃO

Este trabalho apresentou a implementação do *framework* de testes para o sistema operacional JamesOS. Com isso, foi implementado testes de caixa branca para cada módulo do SO. E também, cada conjunto de teste foi baseado no GFC de cada função do módulo do SO. Por fim, foi demonstrado como testar um sistema operacional utilizando a ténica de teste de caixa branca.

A execução dos testes identificou três erros de código no total. Com isso, a densidade de erros identificados foi de 1.23/kSLOC. Dessa forma, é possível perceber que o resultado ficou entre o melhor caso da densidade no sistema operacional Linux, a qual é de 0.5/kSLOC e o pior de 6/kSLOC (Biggs et al., 2018). Por fim, para identificar o problema em código, uma análise manual posterior é necessária.

Foi possível observar uma limitação nos testes do módulo de multitarefa, já que nem sempre foi possível completar o fluxo do GFC por limitações do ambiente utilizado.

Uma outra limitação está relacionada à possibilidade de testes de todos os módulos. Comparando os módulos testados com a árvore de dependências da FIGURA 9, podemos perceber que apenas o "Driver para o PIT", "Comunicação com dispositivos PCI"e "Comunicação com dispositivos entrada e saída"não foram testados.

A técnica de introspecção de máquina virtual foi utilizada em toda execução e validação dos testes. Com isso, foi possível validar o estado dos registrados da CPU, verificar e alterar o conteúdo da memória do sistema. Dessa maneira, possibilitando a injeção de instruções de máquina dinamicamente durante a execução do SO.

4.1 TRABALHOS FUTUROS

O trabalho apresentou dificuldades e limitações encontradas no desenvolvimentos dos testes. E também, utilizou apenas a técnica de todos os caminhos do GFC. Por conta disso, pode-se entender futuras melhorias para o *framework* de testes, sendo elas:

- Implementar chamada das funções testadas de maneira independente da ferramenta;
- Adicionar a técnica de teste de definição e uso;
- Aumentar a cobertura de testes para os módulos não testados;

REFERÊNCIAS

ANDROID. **App Sandbox**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://source.android.com/docs/security/app-sandbox. Citado 1 vez na página 17.

BELLARD, F. QEMU, a fast and portable dynamic translator. In: PROCEEDINGS of the Annual Conference on USENIX Annual Technical Conference. Anaheim, CA: USENIX Association, 2005. (ATEC '05), p. 41. Citado 1 vez na página 20.

BIGGS, S.; LEE, D.; HEISER, G. The Jury Is In: Monolithic OS Design Is Flawed: Microkernel-based Designs Improve Security. In: PROCEEDINGS of the 9th Asia-Pacific Workshop on Systems. Jeju Island, Republic of Korea: Association for Computing Machinery, 2018. (APSys '18). ISBN 9781450360067. DOI: 10.1145/3265723.3265733. Disponível em: https://doi.org/10.1145/3265723.3265733. Citado 2 vezes nas páginas 10, 53.

COMMITTEE, D. D. I. F. **DWARF Debugging Information Format**. [S.l.: s.n.], 2024. Accessed: 2024-12-01. Disponível em: https://dwarfstd.org/. Citado 1 vez na página 20.

CORREIA, J. P. Implementação de um Sistema Operacional Experimental Incremental e Modular. **UESB**, 2020. Citado 2 vezes nas páginas 10, 31, 32.

CREASY, R. J. The origin of the VM/370 time-sharing system. **IBM J. Res. Dev.**, IBM Corp., USA, v. 25, n. 5, p. 483–490, set. 1981. ISSN 0018-8646. DOI: 10.1147/rd.255.0483. Disponível em: https://doi.org/10.1147/rd.255.0483. Citado 1 vez na página 18.

GENODEOS. **Genode Operating System Framework - Introduction**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://genode.org/documentation/genode-foundations/24.05/introduction/Operating-system_framework.html. Citado 1 vez na página 26.

GNU. **GDB: The GNU Project Debugger**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://www.sourceware.org/gdb/. Citado 1 vez na página 20.

GNU. **GNU Make**. [S.l.: s.n.], 2024. Accessed: 2024-12-01. Disponível em: https://www.gnu.org/software/make/manual/make.html. Citado 1 vez na página 33.

GNU. **LD**. [S.l.: s.n.], 2024. Accessed: 2024-12-01. Disponível em: https://sourceware.org/binutils/docs/ld/. Citado 1 vez na página 33.

GNU. **Python API (Debugging with GDB)**. [S.l.: s.n.], 2024. Accessed: 2024-12-01. Disponível em: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Python-API.html. Citado 1 vez na página 34.

GNU. **Remote Protocol (Debugging with GDB)**. [S.l.: s.n.], 2024. Accessed: 2024-12-01. Disponível em: https://sourceware.org/gdb/current/onlinedocs/gdb.html/Remote-Protocol.html. Citado 1 vez na página 34.

GROUP, D. D. I. **DWARF Debugging Information Format, Version 5**. [S.l.], 2017. Accessed: 2024-12-02. Disponível em: http://dwarfstd.org/doc/dwarf5.pdf. Citado 1 vez na página 20.

HEVNER, A. R.; MARCH, S. T.; PARK, J.; RAM, S. Design Science in Information Systems Research. **MIS Quarterly**, Management Information Systems Research Center, University of Minnesota, v. 28, n. 1, p. 75–105, 2004. ISSN 02767783. Disponível em: http://www.jstor.org/stable/25148625. Acesso em: 28 nov. 2024. Citado 1 vezes nas páginas 11, 13.

JUNIT. **JUnit 5**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://junit.org/junit5/. Citado 1 vez na página 25.

KVM. **KVM - Kernel-based Virtual Machine**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://linux-kvm.org/page/Main_Page. Citado 1 vez na página 19.

LARSON, P. Testing linux with the linux test project. In: OTTAWA Linux Symposium. [S.l.: s.n.], 2002. v. 265. Citado 1 vez na página 10.

LINUX.DIE.NET. **cloc(1) - Linux man page**. [S.l.: s.n.], 2024. Accessed: 2024-12-04. Disponível em: https://linux.die.net/man/1/cloc. Citado 1 vez na página 52.

LINUX.DIE.NET. **stdout(3) - Linux man page**. [S.l.: s.n.], 2024. Accessed: 2024-12-01. Disponível em: https://linux.die.net/man/3/stdout. Citado 1 vez na página 36.

MAN7.ORG. **gdbserver - Linux man page**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://man7.org/linux/man-pages/man1/gdbserver.1.html. Citado 1 vez na página 20.

MARINESCU, P. D.; CANDEA, G. Efficient Testing of Recovery Code Using Fault Injection. **ACM Trans. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 29, n. 4, dez. 2011. ISSN 0734-2071. DOI: 10.1145/2063509.2063511. Disponível em: https://doi.org/10.1145/2063509.2063511. Citado 1 vez na página 29.

MARTIGNONI, L.; PALEARI, R.; FRESI ROGLIA, G.; BRUSCHI, D. Testing system virtual machines. In: PROCEEDINGS of the 19th International Symposium on Software Testing and Analysis. Trento, Italy: Association for Computing Machinery, 2010. (ISSTA '10), p. 171–182. ISBN 9781605588230. DOI: 10.1145/1831708.1831730. Disponível em: https://doi.org/10.1145/1831708.1831730. Citado 1 vez na página 29.

MAZIERO, C. A. **Sistemas operacionais: conceitos e mecanismos**. [S.l.]: DINF - UFPR, 2019. ISBN 9788573353402. Citado 2 vezes nas páginas 18, 19.

MICROSOFT. **Debugger in Visual Studio**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://learn.microsoft.com/en-us/visualstudio/debugger/?view=vs-2022. Citado 1 vez na página 20.

MIHAJLOVIĆ, B.; ŽILIĆ, Ž.; GROSS, W. J. Dynamically Instrumenting the QEMU Emulator for Linux Process Trace Generation with the GDB Debugger. **ACM Trans. Embed. Comput. Syst.**, Association for Computing Machinery, New York, NY, USA, v. 13, 5s, dez. 2014. ISSN 1539-9087. DOI: 10.1145/2678022. Disponível em: https://doi.org/10.1145/2678022. Citado 1 vez na página 25.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of software testing**. 3rd. [S.l.]: Wiley, 2011. ISBN 9781118031964. Citado 4 vezes nas páginas 10, 21, 23–25.

NASM. **NASM**. [S.l.: s.n.], 2024. Accessed: 2024-12-20. Disponível em: https://www.nasm.us/. Citado 1 vez na página 33.

NETBSD. **Anita - NetBSD Virtualization**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: http://www.gson.org/netbsd/anita/. Citado 1 vez na página 26.

OPENQA. **openQA - Test Automation for Software**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://open.qa/. Citado 1 vezes nas páginas 26, 27.

OPENSUSE. **openQA - openSUSE Test Automation**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://openqa.opensuse.org/. Citado 1 vezes nas páginas 26, 27.

OSDEV. **8259 PIC**. [S.l.: s.n.], 2024. Accessed: 2024-12-03. Disponível em: https://wiki.osdev.org/8259_PIC. Citado 1 vez na página 38.

OSDEV. **ARM Targeting Multiple Devices**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://wiki.osdev.org/ARM_TargetingMultipleDevices. Citado 1 vez na página 16.

OSDEV. **Boot Sequence**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: http://wiki.osdev.org/Boot_Sequence. Citado 1 vez na página 16.

OSDEV. **C Library**. [S.l.: s.n.], 2024. Accessed: 2024-12-02. Disponível em: https://wiki.osdev.org/C_Library. Citado 1 vez na página 41.

OSDEV. **Exceptions**. [S.l.: s.n.], 2024. Accessed: 2024-12-03. Disponível em: http://wiki.osdev.org/Exceptions. Citado 1 vez na página 40.

OSDEV. **I/O Ports**. [S.l.: s.n.], 2024. Accessed: 2024-12-03. Disponível em: https://wiki.osdev.org/I/O_Ports. Citado 1 vez na página 36.

OSDEV. **Interrupt Descriptor Table**. [S.l.: s.n.], 2024. Accessed: 2024-12-03. Disponível em: https://wiki.osdev.org/Interrupt_Descriptor_Table. Citado 1 vez na página 40.

OSDEV. **Interrupt Service Routines**. [S.l.: s.n.], 2024. Accessed: 2024-12-03. Disponível em: https://wiki.osdev.org/Interrupt_Service_Routines. Citado 1 vez na página 40.

OSDEV. **Memory Management Unit**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://wiki.osdev.org/Memory_Management_Unit. Citado 1 vez na página 17.

OSDEV. **Paging**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://wiki.osdev.org/Paging. Citado 2 vezes nas páginas 17, 18.

OSDEV. **Programmable Interval Timer**. [S.l.: s.n.], 2024. Accessed: 2024-12-03. Disponível em: https://wiki.osdev.org/Programmable_Interval_Timer. Citado 1 vez na página 38.

PAYNE, B. D. Virtual Machine Introspection. In: **Encyclopedia of Cryptography and Security**. Edição: Henk C. A. van Tilborg e Sushil Jajodia. Boston, MA: Springer US, 2011. P. 1360–1362. ISBN 978-1-4419-5906-5. DOI: 10.1007/978-1-4419-5906-5_647. Disponível em: https://doi.org/10.1007/978-1-4419-5906-5_647. Citado 1 vez na página 34.

QEMU. **Documentation/QMP - QEMU**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://wiki.qemu.org/Documentation/QMP. Citado 1 vez na página 34.

QEMU. **QEMU Documentation - About**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://www.qemu.org/docs/master/about/index.html. Citado 1 vez na página 20.

QEMU. **QEMU GDB Integration**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://qemu-project.gitlab.io/qemu/system/gdb.html. Citado 1 vez na página 20.

REDHAT. **What is KVM (Kernel-based Virtual Machine)?** [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://www.redhat.com/en/topics/virtualization/what-is-KVM. Citado 1 vez na página 19.

SEL4. **seL4: A High-Assurance, High-Performance Microkernel**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://sel4.systems/About/seL4-whitepaper.pdf. Citado 2 vezes nas páginas 10, 28.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 10th. [S.l.]: Wiley, 2018. ISBN 9781119320913. Citado 3 vezes nas páginas 14, 15, 17, 41.

SPILLNER, A.; LINZ, T.; SCHAEFER, H. **Software Testing Foundations: A Study Guide for the Certified Tester Exam**. [S.l.]: Rocky Nook, 2014. ISBN 978-1937538422. Citado 1 vez na página 10.

SUCHA, M. **Testing Framework for HelenOS**. 2013. Diploma Thesis – FMFI.KI - Department of Computer Science, Comenius University, Bratislava, Slovakia. Supervisor: RNDr. Jaroslav Janáček, PhD.; Guarantor: prof. RNDr. Branislav Rovan,

PhD.; Assigned: 21.10.2011; Approved: 02.11.2011. Disponível em: https://www.helenos.org/doc/theses/ms-mthesis.pdf. Citado 3 vezes nas páginas 10, 28, 52.

SYSTEMS, E. **ESP-IDF Programming Guide**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html. Citado 1 vez na página 29.

TANENBAUM, A. S.; BOS, H. **Modern Operating Systems**. 5th. [S.l.]: Pearson, 2022. ISBN 9780137618880. Citado 2 vezes nas páginas 14, 51.

THROWTHESWITCH. **Unity Test Framework**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://github.com/ThrowTheSwitch/Unity. Citado 1 vez na página 25.

XEN. **Xen Project Software Overview**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://wiki.xenproject.org/wiki/Xen_Project_Software_Overview. Citado 1 vez na página 19.

XUNIT.NET. **xUnit - A .NET Testing Framework**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://xunit.net/. Citado 1 vez na página 25.

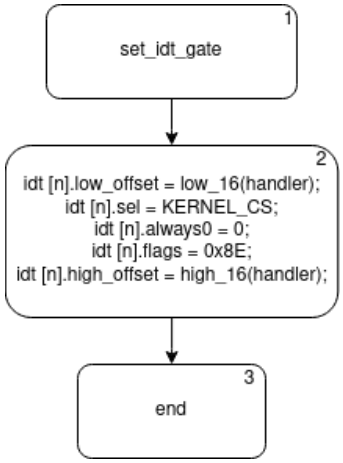
YUAN, W. **Introduction to Debugging**. [S.l.: s.n.], 2024. Accessed: 2024-11-06. Disponível em: https://www.eecg.utoronto.ca/~yuan/teaching/ece244/tutorials/Intro%20to%20Debugging.pdf. Citado 1 vez na página 20.

APÊNDICE 1 – GRAFO DE FLUXO DE CONTROLE DOS TESTES EXECUTADOS

MÓDULO DE INTERRUPÇÕES

Função *set_idt_gate* FIGURA 27

FIGURA 27 – GFC - *SET_IDT_GATE*



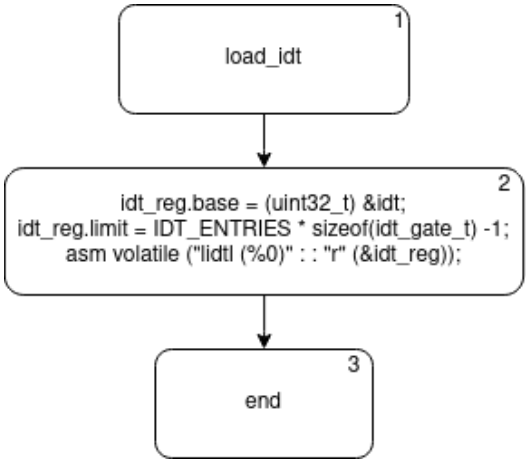
FONTE: Próprio Autor

Execução *set_idt_gate (32, (uint32_t)irq0)*

Caminho executado: [1, 2, 3]

Função *load_idt* FIGURA 28

FIGURA 28 – GFC - LOAD_IDT



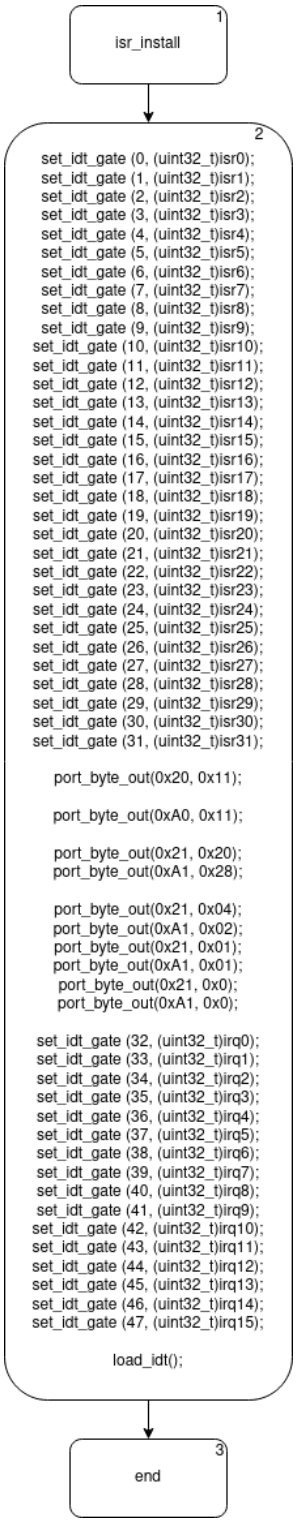
FONTE: Próprio Autor

Execução *load_idt ()*

Caminho executado: [1, 2, 3]

Função *isr_install* FIGURA 29

FIGURA 29 – GFC - ISR_INSTALL



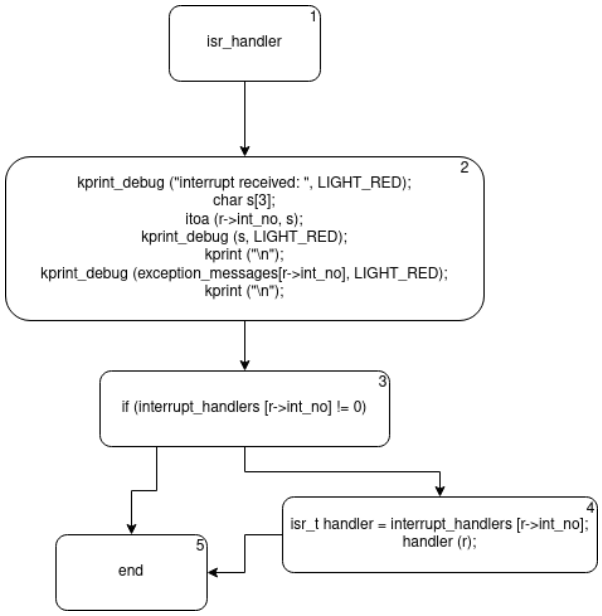
FONTE: Próprio Autor

Execução *isr_install ()*

Caminho executado: [1, 2, 3]

Função *isr_handler* FIGURA 30

FIGURA 30 – GFC - ISR_HANDLER



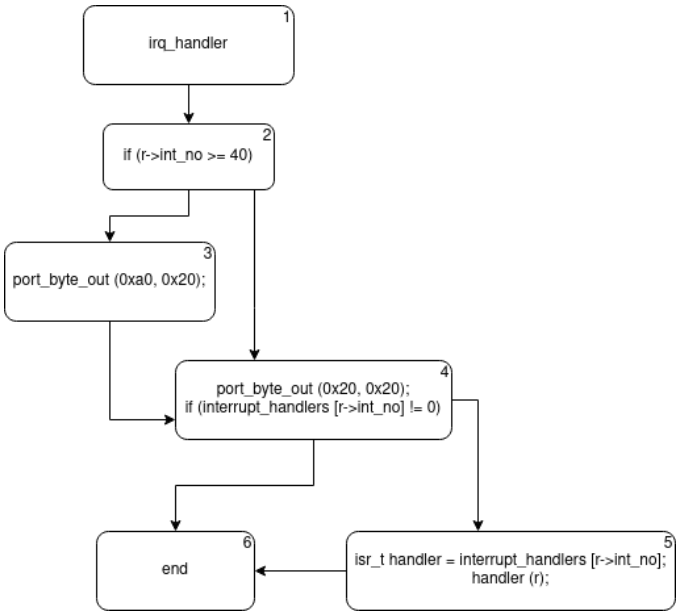
FONTE: Próprio Autor

Execução *isr_handler ()*

Caminho executado: [1, 2, 3, 4, 5]

Função *irq_handler* FIGURA 31

FIGURA 31 – GFC - IRQ_HANDLER



FONTE: Próprio Autor

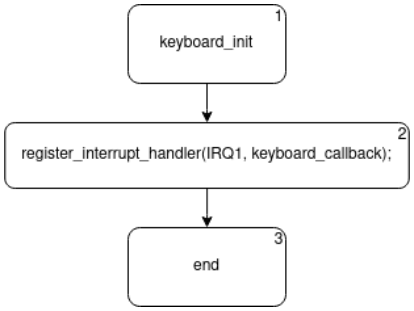
Execução *irq_handler ()*

Caminho executado: [1, 2, 3, 4, 5, 6]

MÓDULO DO *DRIVER* DE TECLADO

Função *keyboard_init* FIGURA 32

FIGURA 32 – GFC - *KEYBOARD_INIT*



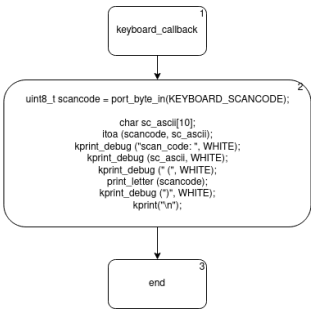
FONTE: Próprio Autor

Execução *keyboard_init ()*

Caminho executado: [1, 2, 3]

Função *keyboard_callback* FIGURA 33

FIGURA 33 – GFC - *KEYBOARD_CALLBACK*



FONTE: Próprio Autor

Execução *keyboard_callback ()*

Caminho executado: [1, 2, 3]

Função *print_letter* FIGURA 34

Execução *print_letter ()*

Caminho executado: [1, 2, 3, 11]

Execução *print_letter ()*

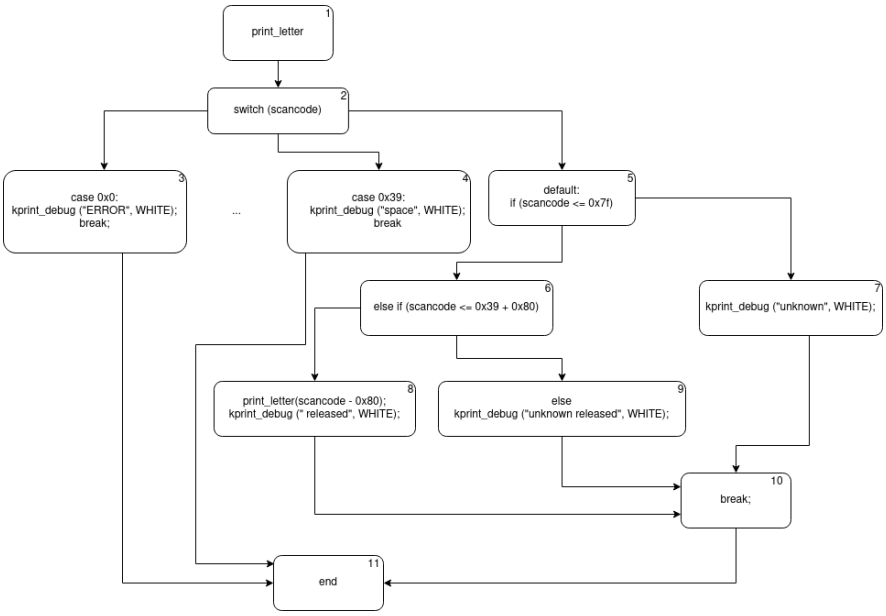
Caminho executado: [1, 2, 3, 4, 11]

Execução *print_letter ()*

Caminho executado: [1, 2, 5, 6, 8, 10, 11]

Execução *print_letter ()*

FIGURA 34 – GFC - *PRINT_LETTER*



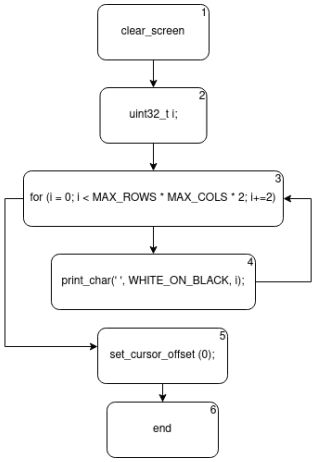
FONTE: Próprio Autor

Caminho executado: [1, 2, 5, 6, 9, 10, 11]

MÓDULO DO *DRIVER* DO VGA

Função *vga_clear_screen* FIGURA 35

FIGURA 35 – GFC - *VGA_CLEAR_SCREEN*



FONTE: Próprio Autor

Execução *vga_clear_screen ()*

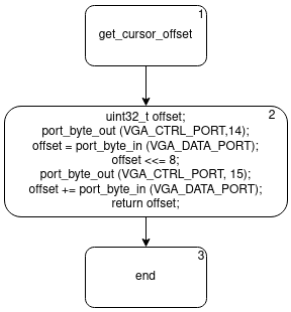
Caminho executado: [1, 2, 3, 4, 3, 5, 6]

Função *vga_get_cursor_offset* FIGURA 36

Execução *vga_get_cursor_offset ()*

Caminho executado: [1, 2, 3]

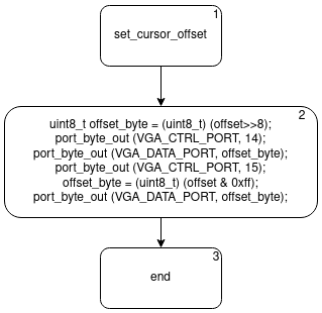
FIGURA 36 – GFC - *VGA_GET_CURSOR_OFFSET*



FONTE: Próprio Autor

Função *vga_set_cursor_offset* FIGURA 37

FIGURA 37 – GFC - *VGA_SET_CURSOR_OFFSET*



FONTE: Próprio Autor

Execução *vga_set_cursor_offset (0)*

Caminho executado: [1, 2, 3]

Função *vga_kprint_at* FIGURA 38

Execução *kprint_at("message_00", 0)*

Caminho executado: [1, 2, 4, 5, 4, 7, 8]

Execução *kprint_at("message_01", 4000)*

Caminho executado: [1, 2, 4, 5, 6, 4, 7, 8]

Função *vga_kprint_debug* FIGURA 39

Execução *kprint_debug("message_05", 0x4)*

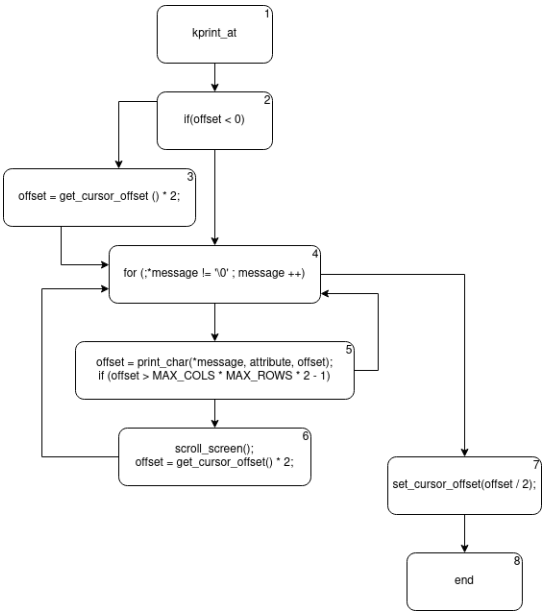
Caminho executado: [1, 2, 3]

Função *vga_kprintf* FIGURA 40

Execução *kprintf("%%modifiers%%: %x, %d, %s", 3, 12345678, 12345678, "teste")*

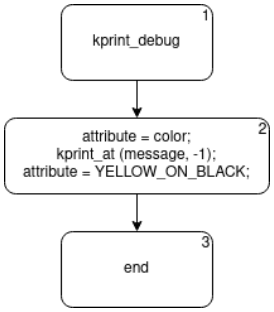
Caminho executado: [1, 2, 3, 5, 8, 9, 3, 5, 7, 3, 8, 10, 11, 13, 15, 18, 15, 17, 19, 3, 5, 8, 10, 12, 19, 3, 5, 8, 10, 11, 14, 16, 19, 3, 4, 6, 20]

FIGURA 38 – GFC - *VGA_KPRINT_AT*



FONTE: Próprio Autor

FIGURA 39 – GFC - *VGA_KPRINT_DEBUG*



FONTE: Próprio Autor

Função *vga_kprint* FIGURA 41

Execução *kprint("message_04")*

Caminho executado: [1, 2, 3]

Função *vga_print_char* FIGURA 42

Execução *print_char(0x61, 0x0f, 0x0)*

Caminho executado: [1, 2, 3, 5]

Execução *print_char(0x61, 0x0f, 0x0)*

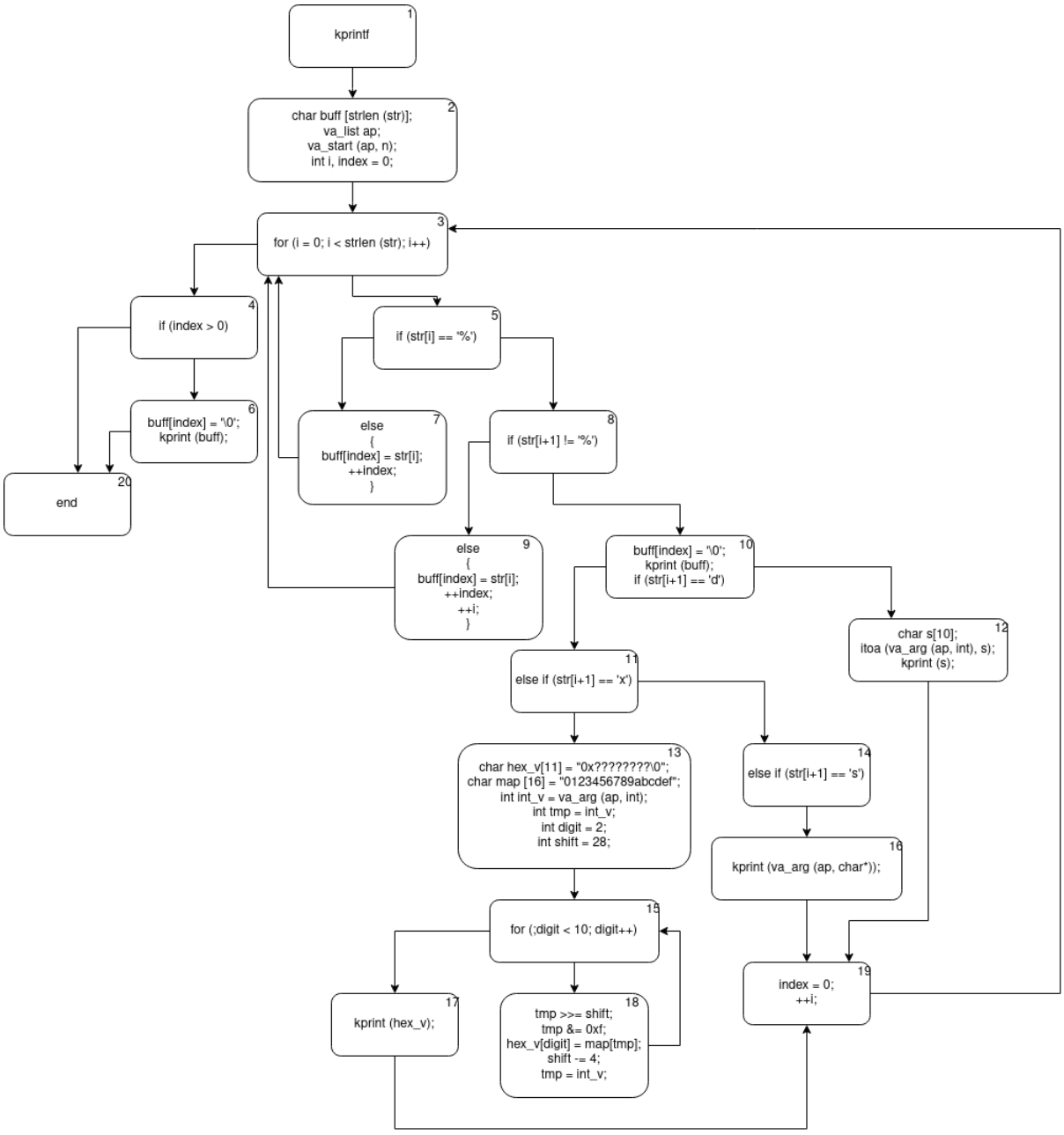
Caminho executado: [1, 2, 3, 4, 5]

Função *vga_scroll_screen* FIGURA 43

Execução *scroll_screen()*

Caminho executado: [1, 2, 3, 5, 3, 4, 6]

FIGURA 40 – GFC - *VGA_KPRINT_KPRINTF*



FONTE: Próprio Autor

MÓDULO DA LIBC

Função *bitmap_bitmap_clear* FIGURA 44

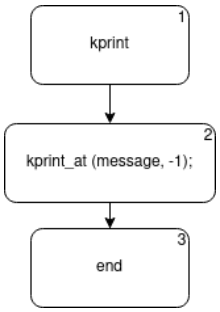
Execução *bitmap_bitmap_clear (0x11f000, 4)*

Caminho executado: [1, 2, 3]

Função *bitmap_bitmap_set* FIGURA 45

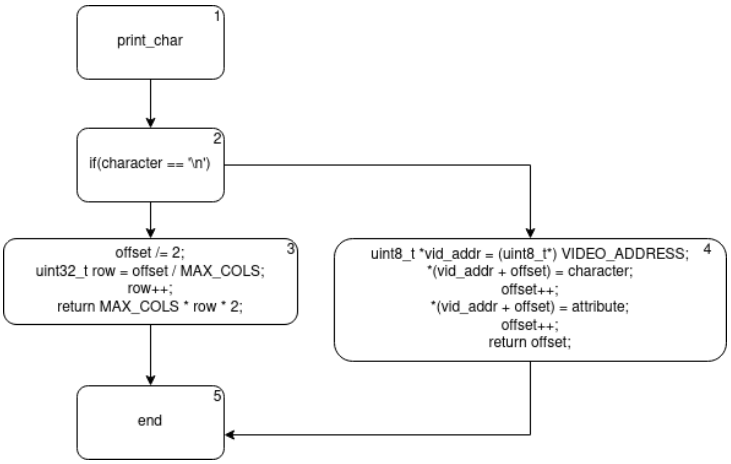
Execução *bitmap_bitmap_set (0x11f000, 4)*

FIGURA 41 – GFC - *VGA_KPRINT_KPRINT*



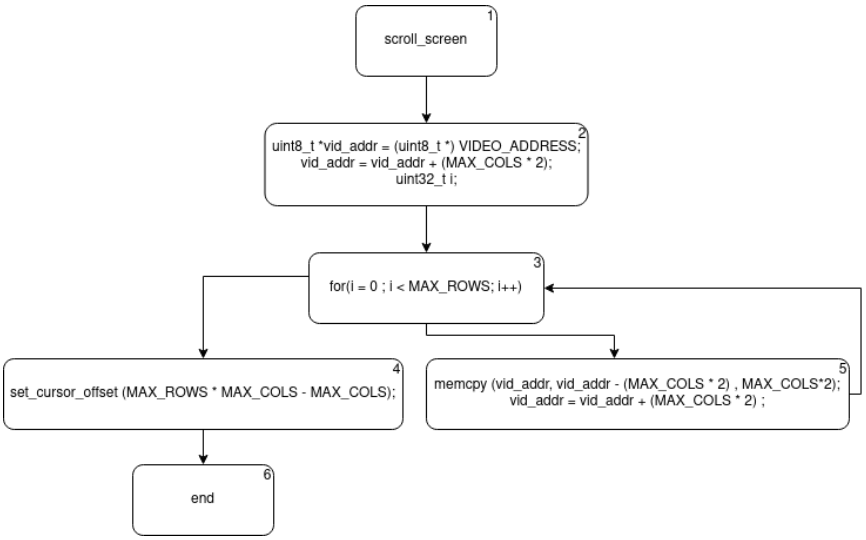
FONTE: Próprio Autor

FIGURA 42 – GFC - *VGA_PRINT_CHAR*



FONTE: Próprio Autor

FIGURA 43 – GFC - *VGA_SCROLL_SCREEN*



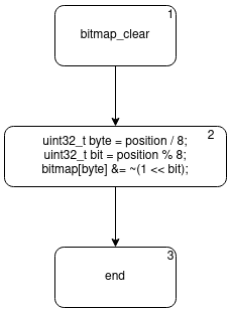
FONTE: Próprio Autor

Caminho executado: [1, 2, 3]

Função *bitmap_bitmap_read* FIGURA 46

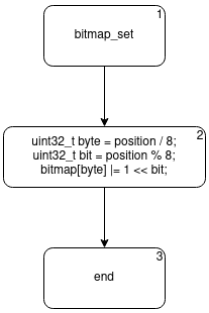
Execução *bitmap_bitmap_read (0x11f000, 4)*

FIGURA 44 – GFC - *BITMAP_BITMAP_CLEAR*



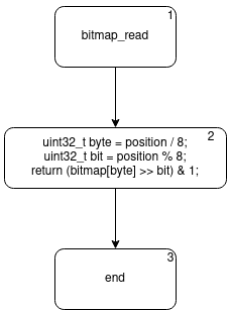
FONTE: Próprio Autor

FIGURA 45 – GFC - *BITMAP_BITMAP_SET*



FONTE: Próprio Autor

FIGURA 46 – GFC - *BITMAP_BITMAP_READ*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3]

Função *bitmap_fill* FIGURA 47

Execução *bitmap_fill (0x11f000, 40, 4, 12, 1)*

Caminho executado: [1, 2, 3, 4, 2, 6]

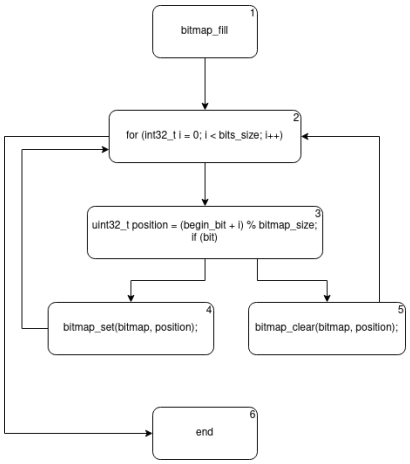
Execução *bitmap_fill (0x11f000, 40, 4, 12, 0)*

Caminho executado: [1, 2, 3, 5, 2, 6]

Função *bitmap_find_sequence* FIGURA 48

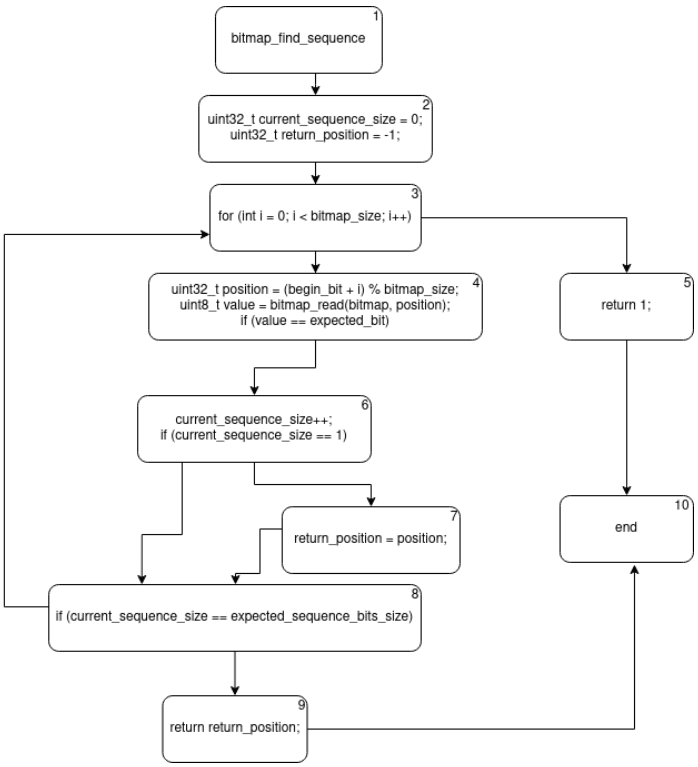
Execução *bitmap_find_sequence (0x11f000, 40, 8, 4, 1)*

FIGURA 47 – GFC - *BITMAP_FILL*



FONTE: Próprio Autor

FIGURA 48 – GFC - *BITMAP_FIND_SEQUENCE*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 4, 6, 7, 8, 9, 10]

Execução *bitmap_find_sequence (0x11f000, 40, 8, 5, 1)*

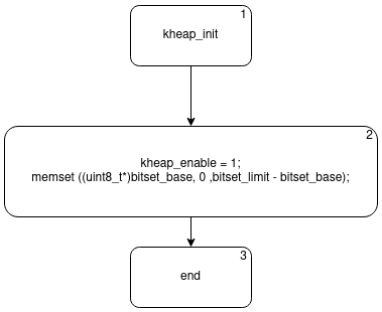
Caminho executado: [1, 2, 3, 6, 8, 3, 5, 10]

Função *kheap_init* FIGURA 49

Execução *kheap_init()*

Caminho executado: [1, 2, 3]

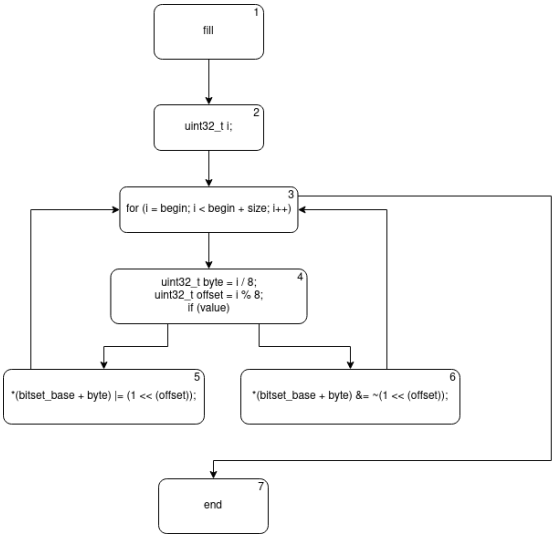
FIGURA 49 – GFC - *KHEAP_INIT*



FONTE: Próprio Autor

Função *kheap_fill* FIGURA 50

FIGURA 50 – GFC - *KHEAP_FILL*



FONTE: Próprio Autor

Execução *fill(0, 12, 1)*

Caminho executado: [1, 2, 3, 4, 5, 3, 7]

Execução *fill(0, 12, 0)*

Caminho executado: [1, 2, 3, 4, 6, 3, 7]

Função *kheap_kfree* FIGURA 51

Execução *kfree(0x11f000, 1)*

Caminho executado: [1, 2, 3]

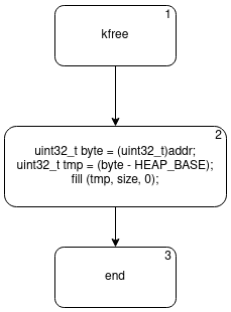
Função *kheap__kmalloc* FIGURA 52

Execução *__kmalloc(0, 0)*

Caminho executado: [1, 2, 4, 14]

Execução *__kmalloc(1, 0)*

FIGURA 51 – GFC - *KHEAP_KFREE*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 5, 7, 9,11, 12, 13, 14]

Execução *__kmalloc(1, 1)*

Caminho executado: [1, 2, 3, 5, 7, 9, 10, 12, 9, 11, 12, 13, 14]

Função *mem_memcmp* FIGURA 53

Execução *memcmp(0x11f000, 0x11f005, 5)*

Caminho executado: [1, 2, 3, 5, 7, 3, 4, 9]

Execução *memcmp(0x11f000, 0xa9, 1)*

Caminho executado: [1, 2, 3, 6, 9]

Execução *memcmp(0x11f000, 0xab, 1)*

Caminho executado: [1, 2, 3, 5, 7, 8, 9]

Função *mem_memcpy* FIGURA 54

Execução *memcpy(0x11f000, 0x11f005, 5)*

Caminho executado: [1, 2, 3, 4, 3, 5]

Função *mem_memmov* FIGURA 55

Execução *memmov(0x11f000, 0x11f005, 5)*

Caminho executado: [1, 2, 4, 5, 7, 9, 7, 11]

Execução *memmov(0x11f005, 0x11f000, 5)*

Caminho executado: [1, 2, 4, 6, 8, 10, 8, 11]

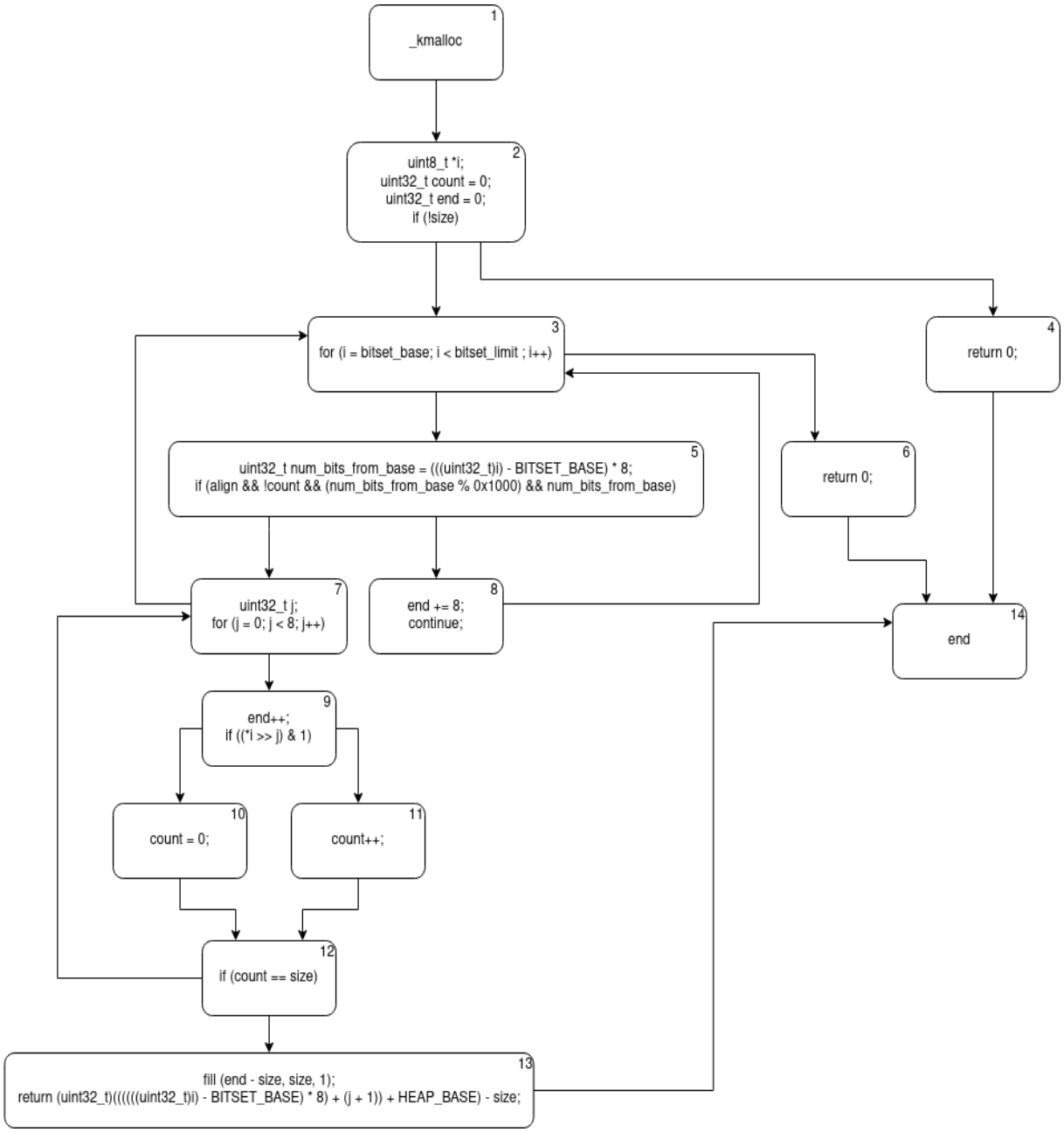
Execução *memmov(0x11f000, 0x11f000, 5)*

Caminho executado: [1, 2, 3, 11]

Função *mem_memset* FIGURA 56

Execução *memset(0x11f000, 0xaa, 5)*

FIGURA 52 – GFC - *KHEAP__KMALLOC*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 2, 4]

Função *minmax_max* FIGURA 57

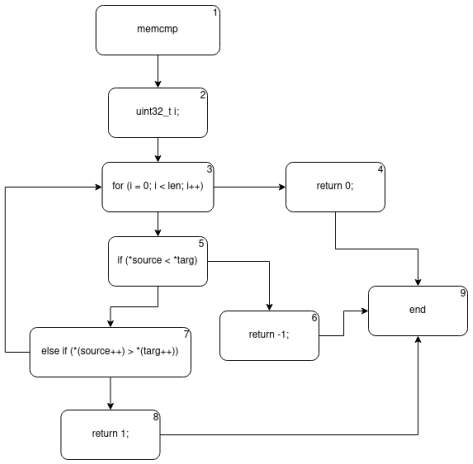
Execução *max(1, 2)*

Caminho executado: [1, 2, 3, 5]

Execução *max(2, 1)*

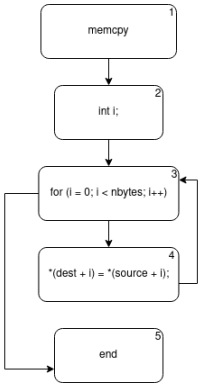
Caminho executado: [1, 2, 4, 5]

FIGURA 53 – GFC - *MEM_MEMCMP*



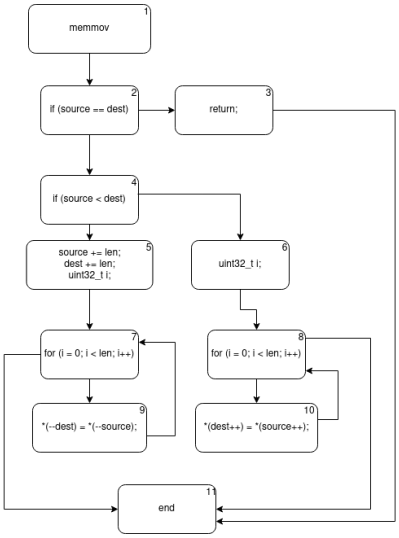
FONTE: Próprio Autor

FIGURA 54 – GFC - *MEM_MEMCPY*



FONTE: Próprio Autor

FIGURA 55 – GFC - *MEM_MEMMOV*

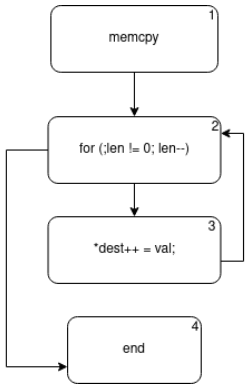


FONTE: Próprio Autor

Função *minmax_min* FIGURA 58

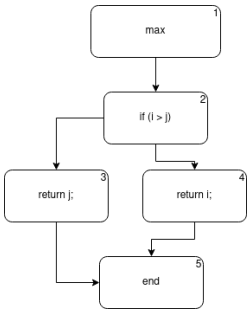
Execução *min(1, 2)*

FIGURA 56 – GFC - *MEM_MEMSET*



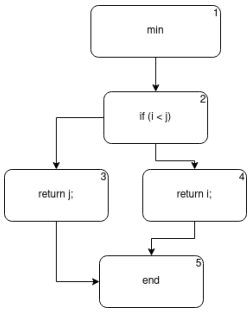
FONTE: Próprio Autor

FIGURA 57 – GFC - *MINMAX_MAX*



FONTE: Próprio Autor

FIGURA 58 – GFC - *MINMAX_MIN*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 5]

Execução *min(2, 1)*

Caminho executado: [1, 2, 4, 5]

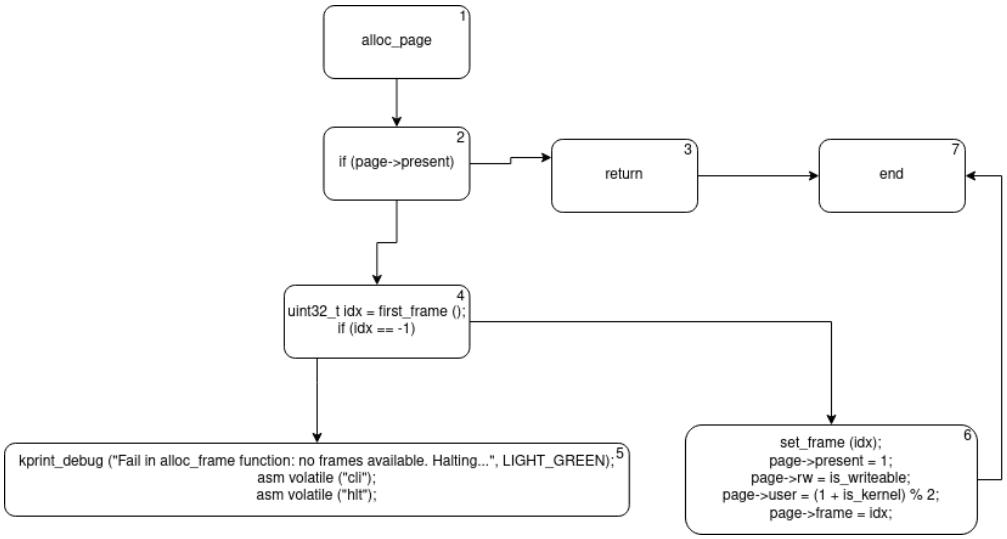
Função *paging_alloc_page* FIGURA 59

Execução *alloc_page(0x1000000, 0x0, 0x0)*

Caminho executado: [1, 2, 4, 6, 7]

Execução *alloc_page(0x1001000, 0x0, 0x0)*

FIGURA 59 – GFC - *PAGING_ALLOC_PAGE*

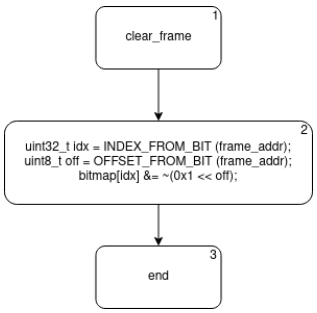


FONTE: Próprio Autor

Caminho executado: [1, 2, 4, 5]

Função *paging_clear_frame* FIGURA 60

FIGURA 60 – GFC - *PAGING_CLEAR_FRAME*



FONTE: Próprio Autor

Execução *clear_frame(3841)*

Caminho executado: [1, 2, 3]

Função *paging_init* FIGURA 61

Execução *paging_init()*

Caminho executado: [1, 2, 3, 4, 3, 5, 6]

Função *paging_free_page* FIGURA 62

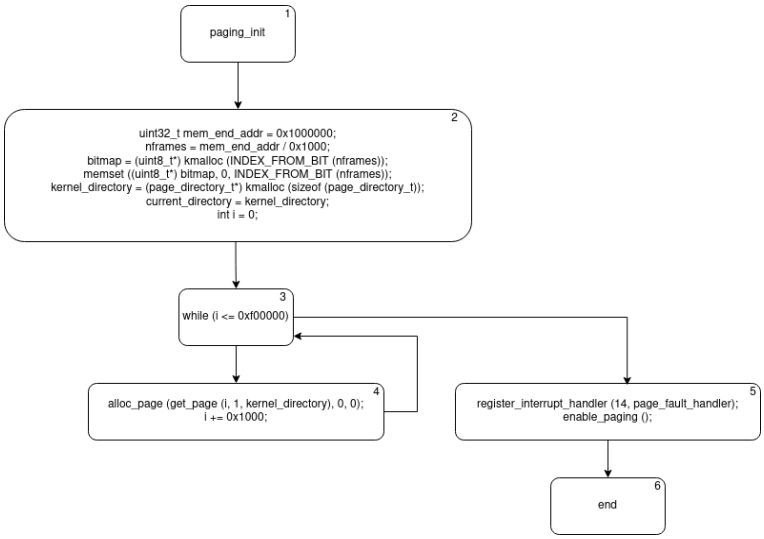
Execução *free_page(0x122400)*

Caminho executado: [1, 2, 3, 5]

Função *paging_get_page* FIGURA 63

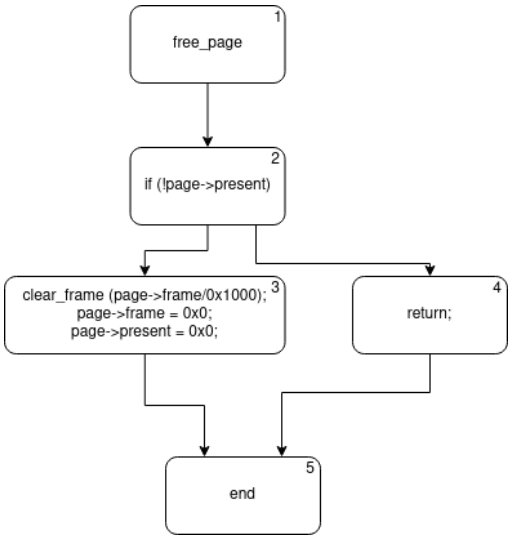
Execução *get_page(0x0, 0x0, 0x120000)*

FIGURA 61 – GFC - *PAGING_INIT*



FONTE: Próprio Autor

FIGURA 62 – GFC - *PAGING_FREE_PAGE*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 7]

Execução *get_page(0x1000000, 0x1, 0x120000)*

Caminho executado: [1, 2, 4, 6, 7]

Execução *get_page(0x1400000, 0x0, 0x120000)*

Caminho executado: [1, 2, 5, 7]

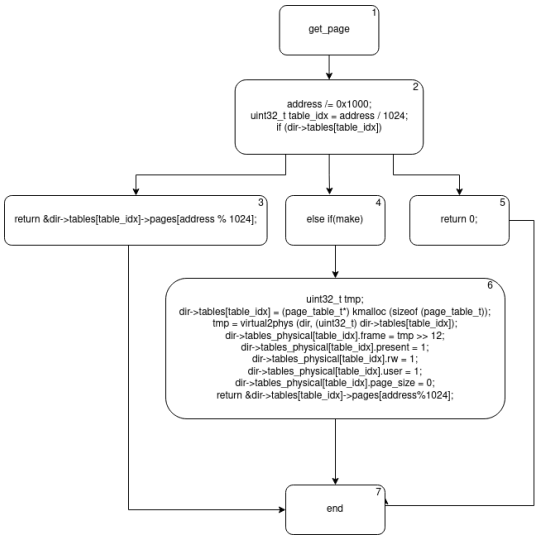
Função *paging_first_frame* FIGURA 64

Execução *first_frame()*

Caminho executado: [1, 2, 3, 5, 6, 7, 3, 5, 6, 7, 8, 9]

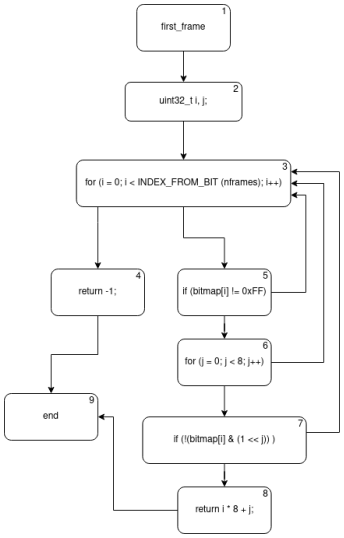
Execução *first_frame()*

FIGURA 63 – GFC - *PAGING_GET_PAGE*



FONTE: Próprio Autor

FIGURA 64 – GFC - *PAGING_FIRST_FRAME*



FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 5, 6, 7, 3, 4, 9]

Função *paging_page_fault_handler* FIGURA 65

Execução *page_fault_handler()*

Caminho executado: [1, 2, 3, 6, 8, 9]

Função *paging_set_frame* FIGURA 66

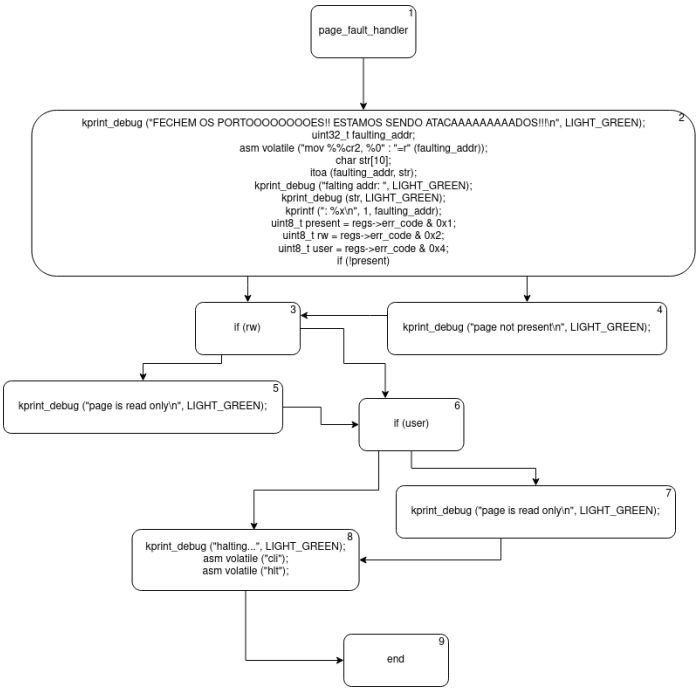
Execução *set_frame(3841)*

Caminho executado: [1, 2, 3]

Função *paging_virtual2phys* FIGURA 67

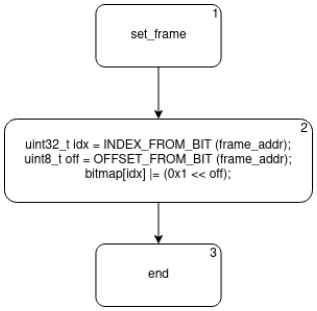
Execução *virtual2phys(0x120000, 0x1234)*

FIGURA 65 – GFC - *PAGING_PAGE_FAULT_HANDLER*



FONTE: Próprio Autor

FIGURA 66 – GFC - *PAGING_SET_FRAME*



FONTE: Próprio Autor

Caminho executado: [1, 2, 4, 9]

Execução *virtual2phys(0x120000, 0x1234)*

Caminho executado: [1, 2, 3, 5, 7, 9]

Execução *virtual2phys(0x120000, 0xffffffff)*

Caminho executado: [1, 2, 3, 6, 9]

Execução *virtual2phys(0x120000, 0x1000000)*

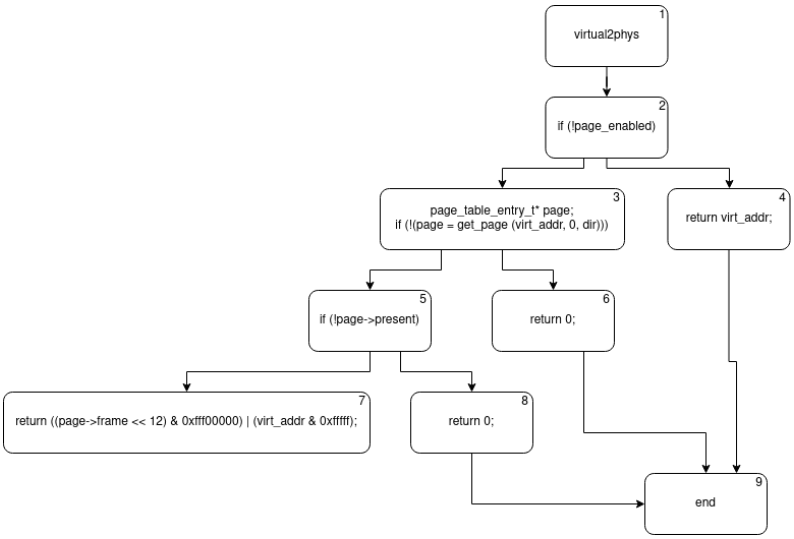
Caminho executado: [1, 2, 3, 5, 8, 9]

Função *strings_itoa* FIGURA 68

Execução *itoa(12345678, 0x11f000)*

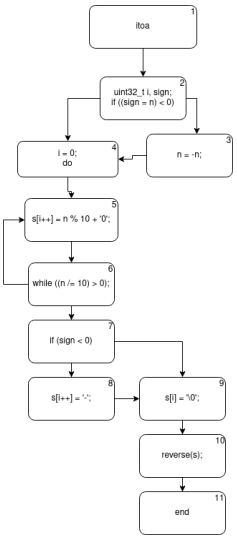
Caminho executado: [1, 2, 4, 5, 6, 5, 6, 7, 9, 10, 11]

FIGURA 67 – GFC - *PAGING_VIRTUAL2PHYS*



FONTE: Próprio Autor

FIGURA 68 – GFC - *STRINGS_ITOA*



FONTE: Próprio Autor

Função *strings_reverse* FIGURA 69

Execução *reverse(0x11f000)*

Caminho executado: [1, 2, 3, 4, 3, 5]

Função *strings_strlen* FIGURA 70

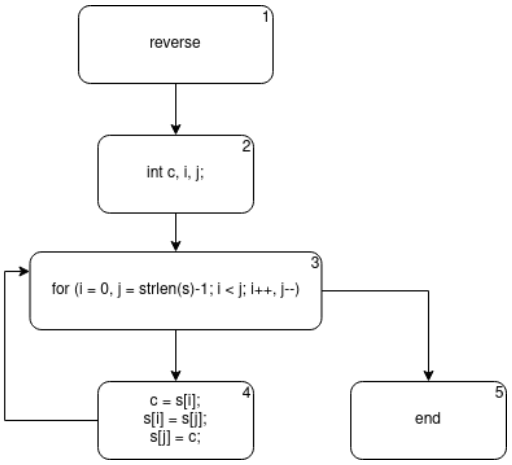
Execução *strlen(0x11f000)*

Caminho executado: [1, 2, 3, 4, 3, 5, 6]

MÓDULO DO MULTI TAREFA

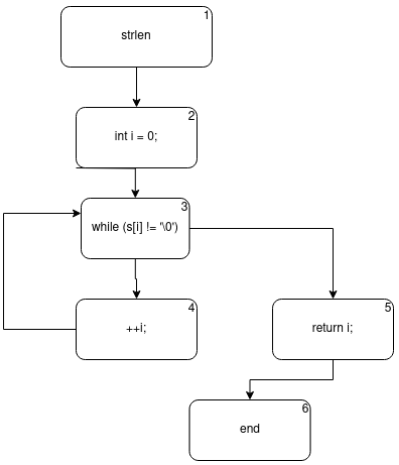
Função *multitasking_block_task* FIGURA 71

FIGURA 69 – GFC - *STRINGS_REVERSE*



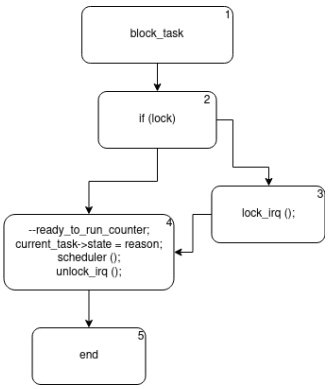
FONTE: Próprio Autor

FIGURA 70 – GFC - *STRINGS_STRLEN*



FONTE: Próprio Autor

FIGURA 71 – GFC - *MULTITASKING_BLOCK_TASK*



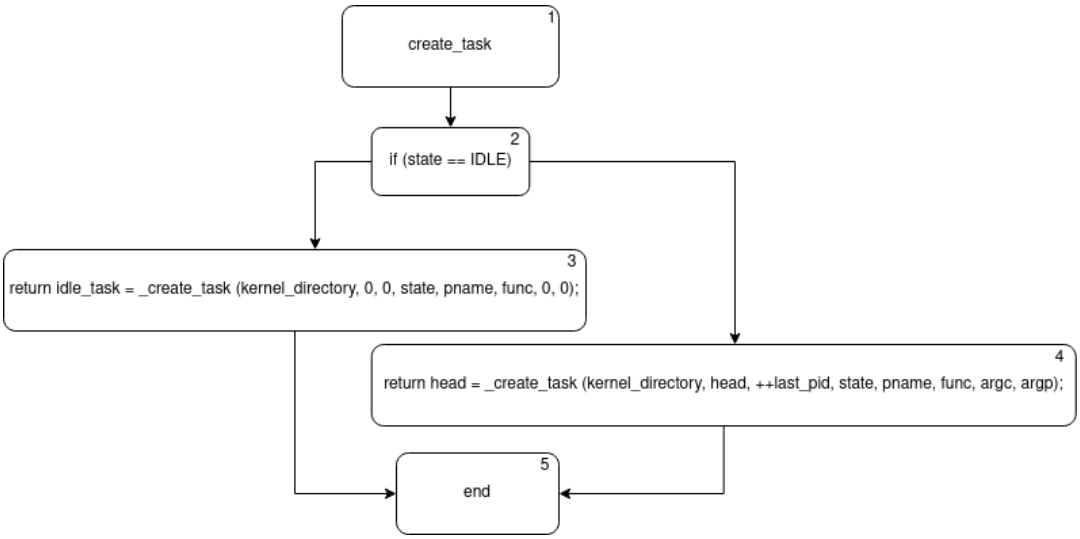
FONTE: Próprio Autor

Execução *block_task(0, 1)*

Caminho executado: [1, 2, 3, 4]

Função *multitasking_create_task* FIGURA 72

FIGURA 72 – GFC - *MULTITASKING_CREATE_TASK*



FONTE: Próprio Autor

Execução *create_task(0, 0x1234, 0x11f003, 0, 1, 0x1234)*

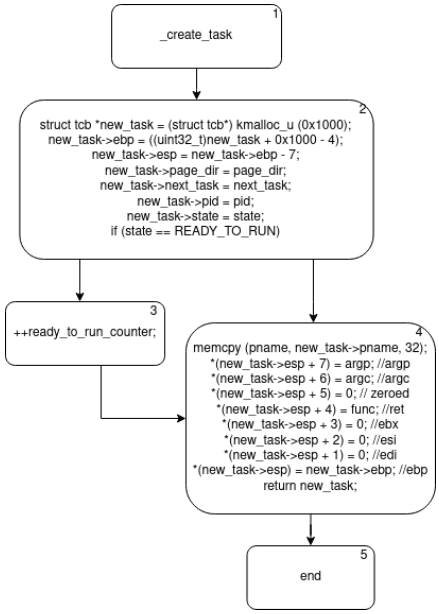
Caminho executado: [1, 2, 3, 5]

Execução *create_task(0, 0x1234, 0x11f003, 4, 1, 0x1234)*

Caminho executado: [1, 2, 3, 4, 5]

Função *multitasking__create_task* FIGURA 73

FIGURA 73 – GFC - *MULTITASKING__CREATE_TASK*



FONTE: Próprio Autor

Execução *_create_task(0x120000, 0, 0, 0, 0x11f003, 0x1234, 0, 0)*

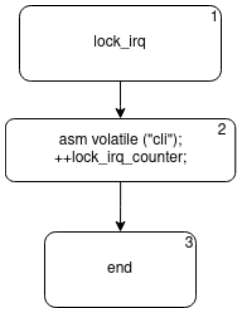
Caminho executado: [1, 2, 4, 5]

Execução *_create_task(0x120000, 0, 2, 2, 0x11f003, 0x1234, 0, 0)*

Caminho executado: [1, 2, 3, 4, 5]

Função *multitasking_lock_irq* FIGURA 74

FIGURA 74 – GFC - *MULTITASKING_LOCK_IRQ*



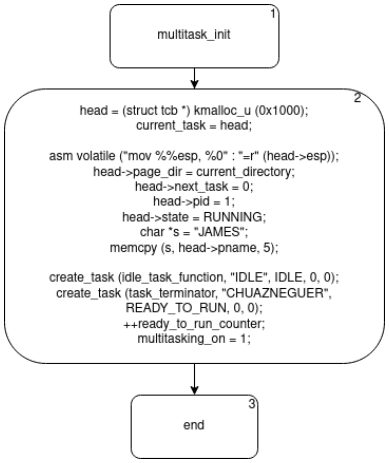
FONTE: Próprio Autor

Execução *lock_irq()*

Caminho executado: [1, 2, 3]

Função *multitasking_init* FIGURA 75

FIGURA 75 – GFC - *MULTITASKING_INIT*



FONTE: Próprio Autor

Execução *multitasking_init()*

Caminho executado: [1, 2, 3]

Função *multitasking_print_task* FIGURA 76

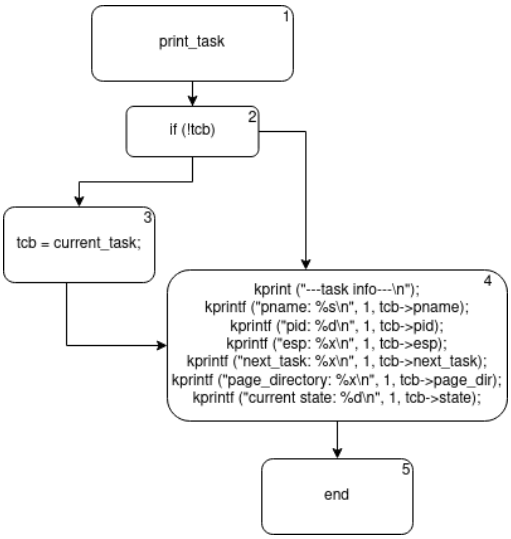
Execução *print_task(0)*

Caminho executado: [1, 2, 3, 4, 5]

Função *multitasking_scheduler* FIGURA 77

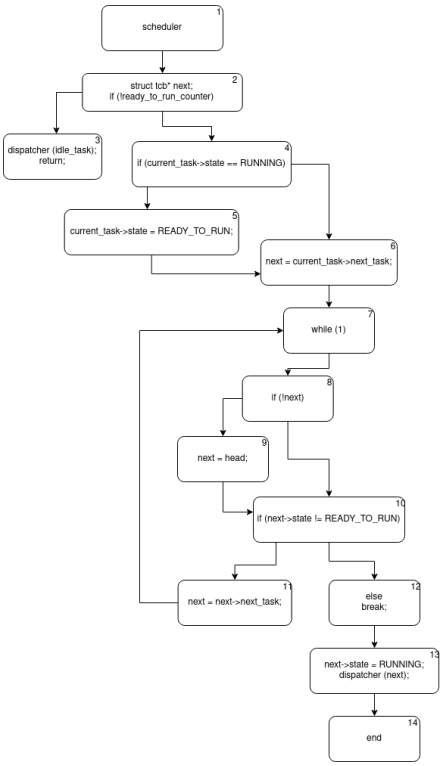
Execução *scheduler()*

FIGURA 76 – GFC - *MULTITASKING_PRINT_TASK*



FONTE: Próprio Autor

FIGURA 77 – GFC - *MULTITASKING_SCHEDULER*



FONTE: Próprio Autor

Caminho executado: [1, 2, 4, 5, 6, 7, 8, 9, 10, 11, 7, 8, 10, 12, 13]

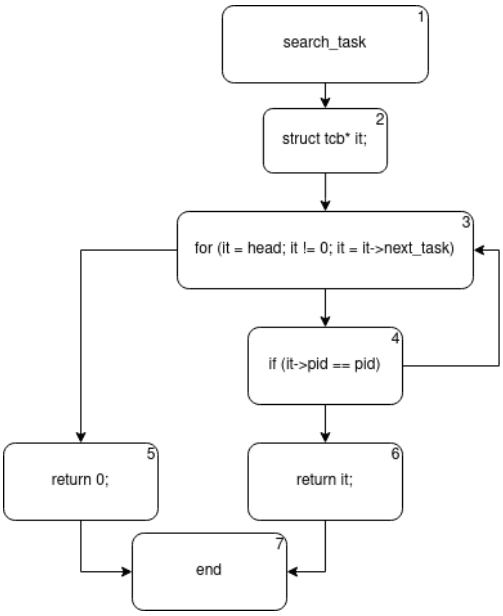
Execução *scheduler()*

Caminho executado: [1, 2, 3]

Função *multitasking_search_task* FIGURA 78

Execução *search_task(5)*

FIGURA 78 – GFC - *MULTITASKING_SEARCH_TASK*



FONTE: Próprio Autor

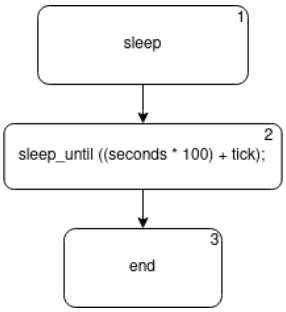
Caminho executado: [1, 2, 3, 4, 6, 7]

Execução *search_task(1234)*

Caminho executado: [1, 2, 3, 4, 3, 5, 7]

Função *multitasking_sleep* FIGURA 79

FIGURA 79 – GFC - *MULTITASKING_SLEEP*



FONTE: Próprio Autor

Execução *sleep(5)*

Caminho executado: [1, 2]

Função *multitasking_sleep_until* FIGURA 80

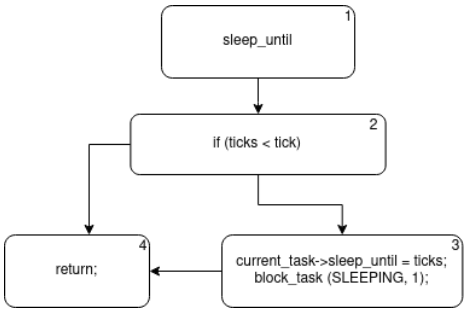
Execução *sleep_until(500)*

Caminho executado: [1, 2, 3]

Função *multitasking_task_termination* FIGURA 81

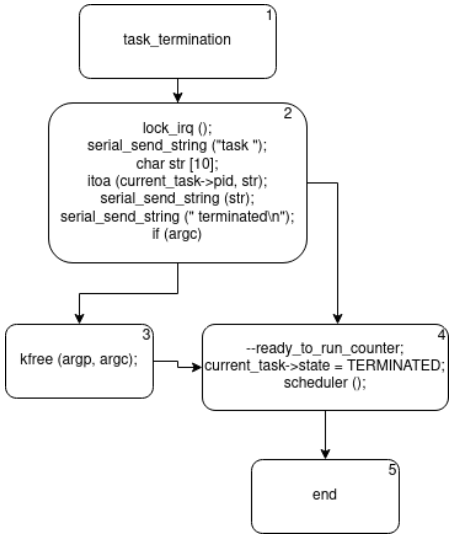
Execução *task_termination(0x12b000)*

FIGURA 80 – GFC - *MULTITASKING_SLEEP_UNTIL*



FONTE: Próprio Autor

FIGURA 81 – GFC - *MULTITASKING_TASK_TERMINATION*

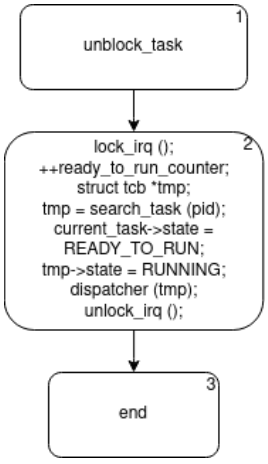


FONTE: Próprio Autor

Caminho executado: [1, 2, 3, 4, 5]

Função *multitasking_unblock_task* FIGURA 82

FIGURA 82 – GFC - *MULTITASKING_UNBLOCK_TASK*



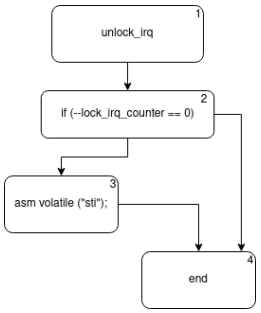
FONTE: Próprio Autor

Execução *unblock_task(1)*

Caminho executado: [1, 2, 3, 4, 5]

Função *multitasking_unlock_irq* FIGURA 83

FIGURA 83 – GFC - *MULTITASKING_UNLOCK_IRQ*



FONTE: Próprio Autor

Execução *unblock_task()*

Caminho executado: [1, 2, 3, 4]