Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Sistema Integrado de Gestão Hospitalar

Hugo Alexandre Pereira Afonso
Mateus Valente Frias Silva
Rodrigo de Almeida Martins
Tiago Filipe Ferreira São Bento

Relatório do Projeto prático de Sistemas Operativos

Curso Engenharia Informática

Janeiro de 2026

Politécnico de Viseu
Escola Superior de Tecnologia e Gestão de Viseu

departamento de informática

Sistema Integrado de Gestão Hospitalar

Hugo Alexandre Pereira Afonso (30032)
Mateus Valente Frias Silva (29989)
Rodrigo de Almeida Martins (30773)
Tiago Filipe Ferreira São Bento (31489)

Relatório do Projeto prático de Sistemas Operativos

Curso de Licenciatura em Engenharia Informática

Ano letivo: 2º

Trabalho efetuado sob a orientação de

António Godinho

Pedro Martins

Janeiro de 2026

Agradecimentos

Enquanto grupo, gostaríamos de manifestar o nosso sincero agradecimento aos professores António Godinho e Pedro Martins pelo apoio, orientação e tempo que nos dedicaram ao longo deste trabalho.

Resumo

O presente projeto consiste na arquitetura e implementação de um Sistema Integrado de Gestão Hospitalar, desenvolvido em linguagem C para ambiente Linux. O sistema foi concebido como uma simulação de alta concorrência, destinada a orquestrar recursos críticos (salas cirúrgicas, equipas médicas, *stock* farmacêutico e capacidade laboratorial) em tempo real. A solução afasta-se de uma abordagem monolítica, adotando uma arquitetura de multiprocessamento distribuído, onde a sincronização e a eficiência na comunicação IPC são os pilares fundamentais.

A espinha dorsal da aplicação reside num processo central ("Gestor"), que atua como *bootstrapper* e monitor do sistema. Através da primitiva fork(), este processo estabelece uma hierarquia ao criar quatro subsistemas filhos, autónomos, mas interligados: o Centro de Triagem, responsável pela admissão e priorização de pacientes; os Blocos Operatórios, que gerem a ocupação de recursos físicos e humanos; a Farmácia Central, que controla o inventário e dispensa de medicamentos; e os Laboratórios, encarregues do processamento assíncrono de análises clínicas. Esta separação garante o isolamento de memória e modularidade, permitindo que uma falha num módulo não comprometa fatalmente a integridade total do sistema.

A comunicação entre estas entidades não é trivial, recorrendo a múltiplos mecanismos de IPC para satisfazer diferentes requisitos de fluxo de dados. A injeção de comandos externos (pacientes, consultas e gestão) é realizada via um FIFO designado *input_pipe*, permitindo uma interação dinâmica e não bloqueante com o exterior. Internamente, o fluxo de trabalho baseia-se em MQs, utilizadas para o despacho transacional e assíncrono de pedidos entre departamentos. Paralelamente, a "verdade única" do estado do sistema reside em segmentos de SHM. Estruturas de dados complexas — como o vetor de *stock* de medicamentos, o estado das salas de cirurgia e as métricas globais de desempenho — são mapeadas neste espaço de endereçamento comum, permitindo leitura e escrita ultrarrápidas por todos os processos.

Dada a natureza concorrente do acesso a esta memória partilhada, a integridade dos dados é assegurada por uma rigorosa estratégia de sincronização. O programa emprega Semáforos POSIX para controlo de acesso a recursos limitados e Mutexes para proteger secções críticas de código, prevenindo eficazmente RCs.

A nível intra-processo, a solução adota um modelo *Multithreaded* (utilizando a biblioteca pthread). Cada subsistema possui capacidade de processamento em paralelo, gerindo múltiplas tarefas sem bloquear o fluxo de execução principal. Para tal, implementou-se uma gestão eficiente de estados de espera recorrendo a variáveis de condição, eliminando busy waiting e otimizando a utilização do processador.

Por fim, o sistema integra um módulo de *Logging* centralizado para registo total das operações, possuindo ainda um signal handler para lidar com os sinais SIGINT, SIGUSR1, SIGUSR2 E SIGCHLD, garantindo principalmente um encerramento gracioso com limpeza correta de recursos IPC e a capacidade de extração de estatísticas operacionais em tempo de execução.

Índice

Índice de Diagramas

Índice de Excertos de Código

Lista de abreviaturas

API – *Application Programming Interface* (Interface de Programação de Aplicações)

BO – Bloco Operatório

CPU – *Central Processing Unit* (Unidade de Processamento Central)

FIFO – *First-In, First-Out* (neste contexto, usado como sinónimo de Named Pipe)

GC – Gestor Central

I/O – *Input/Output* (Entrada/Saída)

IPC – *Inter-Process Communication* (Comunicação Inter-Processos)

MQ – *Message Queue* (Fila de Mensagens)

POSIX – *Portable Operating System Interface*

RC – *Race Condition* (Condição de Corrida)

SHM – *Shared Memory* (Memória Partilhada)

SIGINT – *Signal Interrupt* (Sinal enviado tipicamente por Ctrl+C)

SIGUSR – *Signal User Defined*

SIGCHLD – *Signal Child* (Sinal enviado quando filho termina)

SO – Sistema Operativo

UT – *Time Unit* (Unidade de Tempo simulada no sistema)

Glossário

Bootstrapper: Designa o componente ou fase inicial de execução responsável pelo arranque e configuração do ambiente do sistema. Neste projeto, refere-se ao processo pai (main.c).

Busy Waiting (Espera ativa): Técnica de programação (a evitar) onde um processo verifica repetidamente uma condição de *loop*, consumindo ciclos de CPU inutilmente. Neste projeto, foi substituída pelo uso eficiente de variáveis de condição.

Deadlock: Situação crítica em sistemas concorrentes onde dois ou mais processos ficam bloqueados indefinidamente, aguardando recursos que estão retidos uns pelos outros.

Graceful Shutdown (Encerramento Gracioso): Procedimento controlado de encerramento do sistema, assegurando que todos os recursos IPC (memória, filas e semáforos) são libertados e os ficheiros fechados corretamente antes da terminação dos processos.

Mutex (Mutual exclusion): Primitiva de sincronização utilizada para proteger secções críticas do código, garantindo que apenas uma thread acede a um recurso partilhado num dado momento.

Named Pipe (FIFO): Mecanismo de IPC que funciona como um ficheiro especial no sistema de ficheiros, permitindo a comunicação unidirecional entre processos não relacionados (usado aqui para a entrada de comandos no gestor central).

PID (Process Identifier): Identificador numérico único atribuído pelo SO a cada processo em execução, usado para o distinguir e gerir.

Race Condition (Condição de Corrida): Anomalia que ocorre quando o comportamento do software depende da sequência ou temporização incontrolável de eventos (threads), levando a dados corrompidos se não houver sincronização adequada.

Scheduler (Escalonador): Componente lógico responsável por decidir qual a próxima tarefa a ser executada. No projeto, refere-se à lógica interna que prioriza pacientes nas filas de triagem e cirurgia.

Semáforo: Primitiva de sincronização que mantém um contador interno, utilizada para controlar o acesso a um número limitado de instâncias de um recurso.

Shared Memory (Memória Partilhada): O método mais rápido de IPC, onde uma região de memória é mapeada no espaço de endereçamento de múltiplos processo, permitindo a partilha instantânea de dados.

Starvation: Situação em que um processo ou thread fica indefinidamente à espera de recursos ou tempo de CPU porque outros processos com maior prioridade continuam a ser atendidos, impedindo o seu progresso.

Thread-Safe: Qualidade de um código que pode ser executado simultaneamente por múltiplas threads sem causar erro ou corrupção de dados.

Throughput: Quantidade de trabalho ou operações concluídas por um sistema por unidade de tempo, indicando a sua capacidade de processamento.

Wrapper: Função auxiliar criada para capsular uma chamada de sistema de forma a adicionar tratamento de erros automático e simplificando o código principal.

1 Introdução

O presente relatório documenta o ciclo de desenvolvimento, as decisões arquiteturais e a implementação técnica do Trabalho Prático da Unidade Curricular de Sistemas Operativos. Este projeto visa a consolidação dos conhecimentos teóricos sobre gestão de processos, gestão de memória e programação concorrente, aplicando-os num cenário de simulação realista e complexo: um Sistema Integrado de Gestão Hospitalar. Nas secções seguintes, são detalhados o contexto do problema, os objetivos a atingir com esta implementação e a visão macroscópica da solução proposta.

1.1 Enquadramento e Motivação

O desenvolvimento de soluções que operem em ambientes de alta concorrência e com recursos partilhados é um dos desafios mais complexos e fundamentais em Engenharia Informática e, sobretudo, em Sistemas Operativos (SO). Um hospital é, por definição, um sistema onde a procura (pacientes de emergência, consultas agendadas, cirurgias e análises clínicas) é frequentemente imprevisível e os recursos (salas de cirurgia, equipas médias, *stock* farmacêutico) são finitos e escassos.

A simulação deste cenário obriga à transição dos conceitos teóricos para a aplicação prática exigindo uma arquitetura que garanta não apenas a funcionalidade, mas também a robustez, a integridade e a prevenção de situações críticas como deadlocks ou a inanição de processos (*Starvation*).

1.2 Objetivos

O objetivo principal deste trabalho é demonstrar o domínio sobre as primitivas de sistemas UNIX/Linux. O sistema foi desenhado para cumprir rigorosamente os seguintes objetivos técnicos:

- Arquitetura Multiprocesso: Implementação de um sistema onde componentes distintos (Triagem, Blocos Operatórios, Farmácia e Laboratórios) operam em espaços de memória isolados.
- Comunicação Inter-Processos (IPC): Utilização integrada de múltiplos mecanismos – *Message Queues* (MQ) para troca de mensagens assíncronas,

Shared Memory (SHM) para partilha de estado global, e *Named Pipes* (FIFOs) para injeção de comandos.

- Sincronização e Concorrência: Garantia de consistência dos dados em ambiente multithreaded através do uso criterioso de mutexes, semáforos e variáveis de condição.
- Gestão de Recursos: Implementação de algoritmos de escalonamento que priorizem casos urgentes e a alocação de recursos limitados sem conflitos.

1.3 Visão geral da solução

A solução desenvolvida rejeita uma abordagem monolítica em favor de uma estrutura modular iniciada por um processo gestor (*Bootstrapper*). Este processo é responsável pela criação da hierarquia de processos filhos e pela inicialização de todo o ecossistema de IPC.

Cada componente do sistema possui a sua própria lógica de negócio e gere as suas próprias threads, permitindo o processamento paralelo de múltiplos pacientes e pedidos. O sistema simula o ciclo de vida completo de um paciente, desde a admissão e monitorização de sinais vitais (estabilidade), até à realização de intervenções cirúrgicas complexas que requerem a sincronização precisa de múltiplos recursos. Os processos são independentes, mas estão conectados entre si, uns mais que outros, visando responder sempre aos pedidos efetuados por algum dos processos, ou até mesmo o gestor central.

1.4 Estrutura do relatório

O presente relatório encontra-se organizado de forma a refletir o processo de conceção, implementação e análise do sistema desenvolvido. Após a introdução e contextualização do problema, é apresentada a arquitetura global da solução, com especial enfoque no papel do Gestor Central e na decomposição do sistema em subprocessos independentes. Seguidamente, são descritos em detalhe os mecanismos de comunicação inter-processos e de sincronização adotados, bem como as estratégias utilizadas para garantir a consistência dos dados e a robustez em ambientes concorrentes. O relatório aprofunda depois a lógica de negócio e os algoritmos de escalonamento implementados em cada subsistema funcional, evidenciando as decisões técnicas tomadas. Por fim, são analisados os resultados obtidos, discutidas as limitações da solução e apresentadas considerações críticas sobre o trabalho realizado.

2 Arquitetura do Sistema

Neste capítulo é detalhada a solução apresentada para o problema, explorando o papel central do processo gestor na orquestração dos recursos e a anatomia técnica dos subsistemas especializados, evidenciando as estratégias transversais de gestão de ciclo de vida e concorrência adotadas.

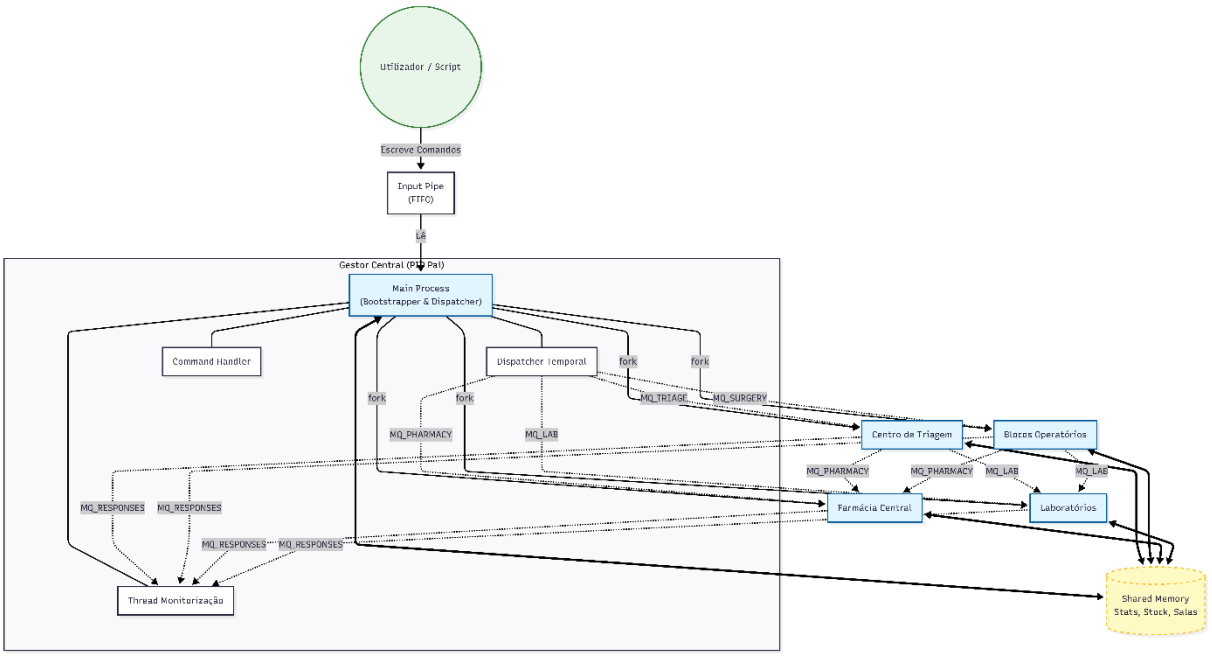


Diagrama 1 – Arquitetura do Sistema

O Diagrama 1 reflete de forma objetiva e geral toda a arquitetura do sistema e como os processos comunicam entre si. Os tópicos seguintes vão aprofundar ainda mais o planeamento existente por detrás de cada processo observado.

2.1 O papel do Gestor central

O processo principal (*main.c*), designado por Gestor Central (GC), atua como o *bootstrapper* e supervisor do sistema. Este processo não executa, portanto, lógica clínica, sendo a sua responsabilidade garantir que o ambiente operacional existe, é estável e coordenar as operações (enviar comandos, receber informações/notificações, receber sinais e tratar do encerramento). A nível técnico, a sua execução segue um fluxo linear e determinístico:

1. Carregamento de Configurações: Lê e valida os parâmetros do ficheiro *config.cfg* (via *config.c*), definindo os limites do sistema.

2. Instanciação de Recursos: Antes de qualquer bifurcação, cria todos os mecanismos IPC (MQ, SHM, semáforos e pipes). Isto garante que todos os filhos já possuem uma cópia destes mesmos mecanismos pronta a ser utilizada, evitando repetição de código.
3. Criação dos Processos: Através da primitiva *fork()*, criamos 4 processos especializados e imediatamente chamamos, em cada processo filho correspondente, a função *main* (ex: *triage_main()*, *surgery_main()*, etc). É ainda guardado o PID de cada processo para monitorização futura.
4. *Loop* de Monitorização de Comandos: Após a inicialização, entra num *loop* infinito onde monitoriza o *Named Pipe* (FIFO), *input_pipe*. O GC lê comandos brutos, valida-os sintaticamente (através de *command_handler.c*) e injeta-os na MQ apropriada, delegando a execução de cada tarefa aos filhos.
5. Relógio Global: Para cumprir requisitos relativos à simulação do tempo, o Gestor possui um relógio interno (em Unidades de Tempo (UT)) que verifica e atualiza o tempo total decorrido em SHM, garantindo a uniformidade entre todos os módulos.
6. *Event Scheduler*: Cada pedido é recebido com um valor *init*, o valor que indica quanto tempo, desde a receção do pedido, deve passar até este ser enviado para o subsistema correspondente. Para satisfazer essa necessidade, o GC consulta o *dispatcher* de eventos, módulo existente em *dispatcher.c* para tomar conhecimento de quando precisa de "acordar" para redirecionar o próximo evento.

No que toca a *threads*, o GC possui a sua *thread* principal, adquirida aquando da execução do programa e uma segunda thread (*t_notification_monitor*) que lê respostas da MQ das

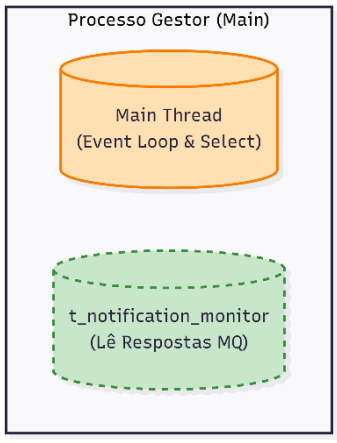


Diagrama 2 – Threads no processo principal(gestor)

respostas (notificações de pedidos enviados, com possibilidade para escalar no futuro). Isto pode ser observado pelo Diagrama 2.

2.2 Arquitetura dos Subprocessos

O sistema implementa um modelo descentralizado onde cada departamento funcional opera como um processo independente. Esta arquitetura garante o isolamento de memória: uma falha crítica num componente não corrompe as estruturas de dados dos restantes, preservando a integridade global do sistema.

2.2.1 Aspetos Transversais e Gestão Comum

Apesar das suas responsabilidades distintas, todos os subprocessos partilham uma base técnica comum para garantir a coerência do sistema, implementada maioritariamente em *manager_utils.c*.

Nesse ficheiro está implementado o tratamento dos sinais e o ciclo de vida dos processos, onde todos os processos (filhos) registam *handlers* específicos (*setup_child_signals*) para a interceção de sinais de terminação (SIGINT, SIGTERM), o que permite uma paragem controlada do programa, possibilitando a oportunidade de terminar qualquer operação em execução. É de notar que os sinais SIGUSR1, SIGUSR2 e SIGCHLD são ignorados, para evitar uma resposta dupla do sistema (apenas o GC deve reagir a estes sinais).

Para além disto, o ficheiro aborda ainda a limpeza dos recursos com *child_cleanup* em cada módulo, assegurando, por exemplo, o *detach* dos segmentos de memória partilhada e a libertação de memória dinâmica local, prevenindo *memory leaks*.

Outro dos aspetos fundamentais dos subprocessos é o seu modelo *multithread*: adota-se um padrão de *Worker-Threads*. Uma *thread* gestora principal consome pedidos da MQ, enquanto *threads* trabalhadoras são lançadas ou acordadas para o processamento pesado, maximizando o paralelismo.

2.2.2 Responsabilidade e Implementação Específica

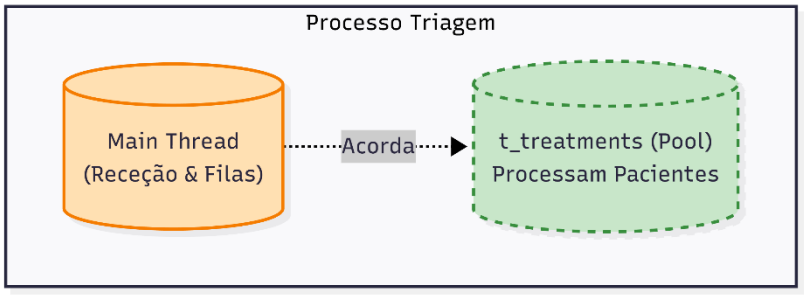


Diagrama 3 - Threads no processo da triagem

O centro de triagem (*triage.c*) gere a admissão de pacientes implementando um modelo de filas de prioridade. Utiliza *threads* dedicadas para a gestão de *mutexes* que protegem a integridade das filas enquanto múltiplas *threads* de tratamento (*t_treatments*) processam admissões simultaneamente, avaliando a estabilidade e encaminhando pedidos para outros setores, como se pode ver pelo Diagrama 3.

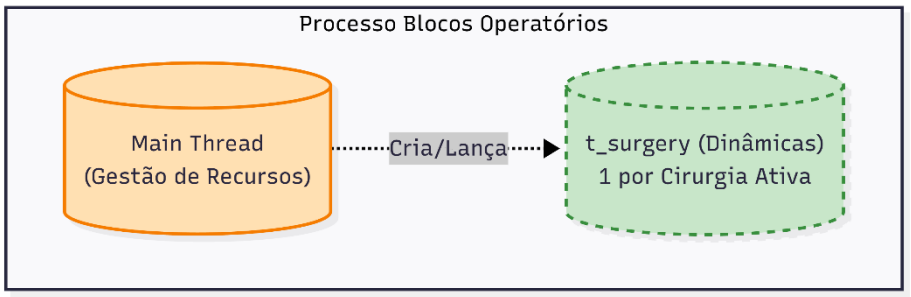


Diagrama 4 – Threads no processo da cirurgia

O módulo de blocos operatórios (*surgery.c*) representa o consumidor crítico de recursos, exigindo a sincronização complexa de múltiplos pré-requisitos. Devido à lógica de negócio pretendida, não podemos avançar com uma cirurgia se não tivermos todas as dependências (sala, equipa, análises e medicamentos) satisfeitas. A exclusão mútua e a gestão de recursos finitos são asseguradas por semáforos POSIX (*sem_bo** e *sem_medical_teams*), garantindo que o número de cirurgias ativas nunca excede a capacidade física máxima. Mais uma vez, possuímos um sistema com *threads* trabalhadoras, como observado no Diagrama 4.

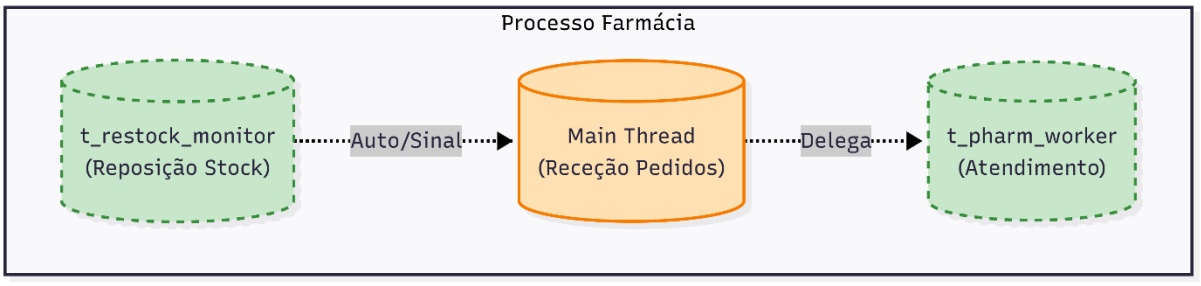


Diagrama 5 - Threads no processo da farmácia

A farmácia neste cenário atua como um guardião do *stock* de medicamentos, lidando com acessos de alta frequência e latência baixa – implementa uma proteção granular de inventário em SHM através de *mutexes*. Destaca-se a introdução de uma *thread* autónoma de monitorização (*t_restock_monitor*), visível no Diagrama 5, que utiliza variáveis de condição para desencadear a reposição automática de *stock* sem bloquear as *threads* de atendimento urgente.

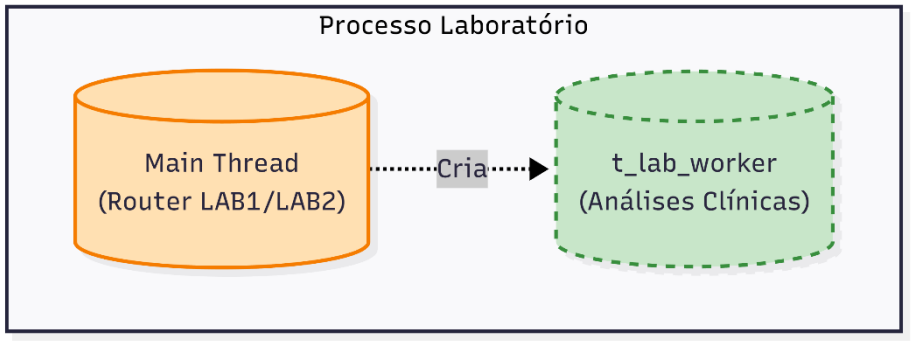


Diagrama 6 - Threads no processo do laboratório

O laboratório é um dos pontos mais críticos do sistema, uma vez que recebe pedidos de 3 fontes diferentes (GC, triagem e cirurgia) e apenas possui 2 laboratórios, ou seja, poucos recursos para muitos pedidos. Este processo simula também o processamento assíncrono de tarefas com duração variável, adotando um modelo de *thread pool* dinâmica – os pedidos são encaminhados para o laboratório específico, onde *threads* trabalhadoras (Diagrama 6) competem por *slots* de processamento limitados por semáforos (*sem_lab**). O resultado é devolvido assincronamente, libertando os processos solicitantes de esperas ativas.

Apesar de parecerem todos bastante semelhantes, a lógica de cada um urge-nos para o uso de estruturas de dados diferentes e inclusivamente até ao uso de recursos IPC de forma pouco habitual. Visto também estarmos a trabalhar com *threads*, foram criadas *wrappers*(em *safe_threads.c*) para todas as funções usadas e da biblioteca <pthread.h>, visando evitar possíveis erros.

3 Mecanismos de Comunicação e Sincronização (IPC)

A complexidade de um sistema destas dimensões reside, fundamentalmente, na forma como os seus componentes trocam dados e coordenam ações. Para responder aos requisitos de desempenho e isolamento, o projeto implementa uma arquitetura híbrida de IPC baseada em primitivas System V (MQ) e POSIX (Semáforos e SHM).

Esta secção detalha a implementação destes mecanismos, com destaque para a camada de abstração (*wrappers*) desenvolvida para garantir a estabilidade do sistema face a interrupções de sinais e erros do sistema. É ainda de notar que todas as estruturas IPC são criadas antes do *fork()* visando serem transversais a todos os processos.

3.1 Filas de Mensagens (MQ)

As MQ constituem a espinha dorsal da comunicação assíncrona entre o gestor central e os subsistemas operacionais, e foi precisamente por permitirem o despacho assíncrono de dados estruturados entre processo desacoplados, garantindo que os pedidos permanecem num *buffer* seguro até que os subsistemas estejam livres para os processar, que as escolhemos.

3.1.1 Modelo de Mensagens e Tipos

Para evitar a rigidez de estruturas monolíticas, adotou-se um modelo de herança de dados em C. Todas as mensagens partilham um cabeçalho comum (*msg_header_t*, definido em *mq.h*), permitindo que funções genéricas processem os dados (como *timestamps* e IDs de paciente) sem conhecerem o conteúdo específico da mensagem.

O sistema utiliza estruturas especializadas para diferentes contextos:

- *msg_new_surgery_t*: Transporta detalhes clínicos (tipo de cirurgia, urgência) e dados logísticos (medicamentos, testes/exames);
- *msg_pharmacy_request_t*: Contém vetores de medicamentos e quantidades, otimizados para pedidos em lote;

- *msg_response_t*: Utilizada para o *feedback loop* (via *MQ_RESPONSES*), permitindo que os subsistemas notifiquem o GC.

3.1.2 Abstração e *Wrappers*

O uso direto de *msgsnd* e *msgrcv* é propenso a falhas silenciosas, especialmente em ambientes com muitos sinais. Para mitigar este risco, foram implementados *wrappers* robustos em *mq.c*, como *send_generic_message()* ou *received_generic_message()*, que foram muito úteis na maioria dos módulos. Para além de informar quando uma mensagem não foi enviada ou de lidar com sinais através de *EINTR*, revelaram-se muito úteis para efeitos de *debugging*, como será explicado em **SECÇÃO**.

3.2 Memória Partilhada (SHM)

Enquanto as MQs transportam eventos, a SHM armazena o estado persistente do sistema. A sua utilização justifica-se pela necessidade de acesso instantâneo (leitura/escrita na ordem dos nanossegundos) a dados globais por múltiplos processos, sem o *overhead* da cópia de mensagens.

3.2.1 Organização das Estruturas

A SHM não é um bloco amorfo; está estruturada hierarquicamente através da *struct* raiz *hospital_shm_t* (em *shm.c*), que agrega ponteiros para segmentos funcionais de cada módulo existente, de forma a uniformizar e facilitar o acesso a esta estrutura.

```
// Hospital SHM memory structure
typedef struct {
    global_statistics_shm_t *shm_stats;
    surgery_block_shm_t *shm_surg;
    pharmacy_shm_t *shm_pharm;
    lab_queue_shm_t *shm_lab;
    critical_log_shm_t *shm_critical_logger;
} hospital_shm_t;
```

Excerto Código 1 - Wrapper global da SHM

Como pode ser observado no Excerto Código 1, este *wrapper* inclui toda a SHM necessária, incluindo um *critical_log_shm_t* que, apesar de não ser utilizado para ler dados

(pois não é explícito no enunciado), foi utilizado para manter um registo claro de todos os eventos de nível crítico do sistema, permitindo a sua expansão no futuro.

Para além disto, é fundamental destacar que cada estrutura de SHM é estritamente controlada por *mutexes* (*p_thread_mutex*), garantindo que uma leitura de estatísticas nunca ocorre a meio de uma atualização de valores.

3.3 *Pipes*

Enquanto as MQ foram utilizadas para a troca de mensagens estruturadas entre processos, os mecanismos de *pipe* foram adotados para gerir fluxos de dados brutos (streams) e eventos de sinalização, tirando partido da sua integração nativa com chamadas de sistemas bloqueantes como o *select()*.

3.3.1 *Named Pipe* (FIFO) – Interface de Entrada

Para a injeção de comandos externos no sistema, optou-se pela criação de um FIFO designado *input_pipe*, em detrimento de *sockets* ou MQ, uma vez que esta abordagem maximiza a interoperabilidade, permitindo o envio de instruções através de ferramentas padrão da *Shell* (como *echo* ou *cat* para *scripts*), dispensando o desenvolvimento de um cliente dedicado, e facilita a integração no ciclo de eventos principal, pois o FIFO é tratado como um descritor de ficheiro convencional monitorizável pela função *select()*.

3.3.2 *Anonymous Pipe* – O "*Self-Pipe Trick*"

Um dos desafios críticos na programação de sistemas UNIX é o tratamento seguro de sinais quando o processo se encontra bloqueado numa chamada de sistema (como o *sem_wait* ou *select*). A execução de lógica complexa (como *printf* ou manipulação de memória) dentro de um *signal handler* é perigosa e não recomendada (não é *async-signal-safe*).

Para resolver este problema, implementou-se o padrão arquitetutal *self-pipe trick*, que consiste em criar um *pipe* anónimo com *pipe(signal_pipe)*, acrescentando depois o descritor de leitura ao conjunto de monitorização do select() no *loop* principal. Desta forma, quando um sinal é intercetado, o *handler* limita-se a escrever um único *byte* na extremidade de escrita deste *pipe*. Esta operação é atómica e segura. A escrita acorda imediatamente o *select()* no processo principal (pois há dados para ler no *pipe*). O fluxo de execução sai do estado e processa o sinal

de forma síncrona e segura no contexto da *thread* principal, evitando *race conditions* (RC) e corrupção de dados.

3.4 Semáforos

A sincronização de recursos finitos e indivisíveis é gerida através de Semáforos Nomeados POSIX. Ao contrário dos *mutexes* (usados para exclusão mútua simples), os semáforos implementam a lógica de contagem de recursos.

Neste modelo, os semáforos são inicializados estrategicamente conforme a natureza do recurso: semáforos binários (inicializados a 1) representam a ocupação de espaços físicos únicos, como as salas de cirurgia (ex: */sem_surgery_bo1*). Esta arquitetura permite que, na ausência de recursos partilhados disponíveis, a thread solicitante transite para um estado de bloqueio eficiente ao nível do *kernel*, libertando ciclos de CPU para outras tarefas até que o recurso seja libertado.

3.4.1 Segurança e robustez

Uma das decisões técnicas mais relevantes para a estabilidade do sistema foi o encapsulamento das primitivas de sincronização na função *sem_wait_safe* (implementada em *sem.c*). O uso direto da chamada de sistema *sem_wait()* acarreta riscos de bloqueio indefinido ou falhas prematuras devido a interrupções por sinais.

Para mitigar estes problemas, o *wrapper* desenvolvido substitui o bloqueio passivo por um ciclo de verificação inteligente utilizando *sem_timedwait*. Esta abordagem permite que a *thread* acorde periodicamente (em intervalos de 100ms) para verificar a *flag* global de *shutdown*, garantindo que nenhum processo permanece "preso" num semáforo aquando do encerramento do sistema. Adicionalmente, a função confere resiliência à aplicação ao filtrar e tratar automaticamente erros do tipo *EINTR*, assegurando a continuidade operacional mesmo quando o processo é sujeito a sinais intensivos de monitorização e estatística.

4 Sincronização e Gestão de Concorrência

Num sistema onde coabitam múltiplos processos e *threads*, a consistência dos dados não é garantida pelo SO, mas sim pela arquitetura de *software*. Sem uma estratégia de sincronização rigorosa, o acesso múltiplo à SHM resultaria inevitavelmente em RCs, corrupção de estruturas de dados e comportamento imprevisível.

Este capítulo detalha as políticas de exclusão mútua e coordenação de tarefas implementadas para garantir a atomicidade das operações e a prevenção de bloqueios, utilizando a biblioteca *pthread* (e os respetivos *wrappers*) e os mecanismos POSIX descritos anteriormente.

4.1 Proteção de Regiões Críticas (*Mutexes*)

A principal estratégia de defesa da integridade dos dados reside na identificação e proteção de regiões críticas – segmentos de código onde recursos partilhados são acedidos ou modificados.

Para tal, optou-se pelo uso de *mutexes*, existindo um para cada subestrutura da SHM como já mencionado. Um bom exemplo desta necessidade reside no módulo da farmácia, na gestão do *stock* – o acesso ao vetor de medicamentos é protegido pelo *shm_pharm->global_mutex* de forma a garantir que, caso duas *threads* tentem decrementar (por exemplo) o *stock* do mesmo medicamento simultaneamente, as operações ocorrem sequencialmente, evitando inconsistências no inventário.

4.2 Eliminação da Espera Ativa (Variáveis de Condição)

Um dos requisitos de eficiência do projeto era a minimização do consumo de CPU. A técnica de *polling* (verificar repetidamente num ciclo *while* se uma condição é verdadeira) foi estritamente rejeitada em favor do uso de variáveis de condição, evitando assim espera ativa.

Esta abordagem permite implementar um modelo de coordenação passiva, de forma que, quando uma *thread* trabalhadora (ex: na triagem) não tem tarefas, ela fica num ciclo infinito a

consumir ciclos de processador. Em vez disso, invoca safe_*pthread_cond_wait*, libertando o *mutex* associado e entrando num estado de sono profundo gerido pelo *kernel*. Assim que o GC ou outra *thread* coloca um novo pedido na fila, envia um sinal (*safe_pthread_cond_signal*), acordando apenas a *thread* necessária para processar o trabalho.

Um exemplo prático desta implementação encontra-se na monitorização da farmácia (*t_restock_monitor*), que permanece suspensa até que o nível de *stock* atinja um limiar crítico, momento em que é "acordada" para iniciar a reposição.

4.3 Prevenção de Deadlocks

O risco de *deadlock* é elevado em sistemas como *surgery.c*, onde uma operação necessita de adquirir múltiplos recursos distintos (sala e equipas médicas) para prosseguir. Se a aquisição não for ordenada, dois processos poderiam bloquear-se mutuamente

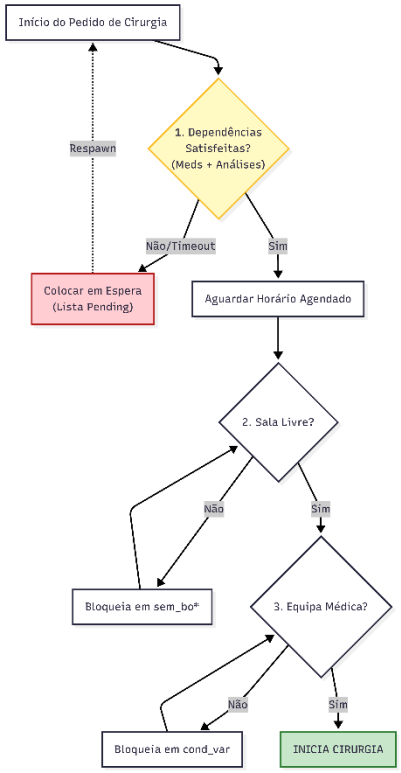


Diagrama 7 - Lógica para evitar deadlocks na cirurgia

indefinidamente.

O Diagrama 7 ilustra o fluxo de execução de uma cirurgia, evidenciando a estratégia hierárquica de aquisição de recursos implementada para maximizar a eficiência. Antes de bloquear qualquer recurso físico, o sistema valida se os pré-requisitos logísticos estão satisfeitos. Caso contrário, o pedido é colocado em espera (*Pending List*), libertando a *thread*

para evitar bloqueios inúteis. Além disso, apenas com as dependências garantidas e no horário agendado, o processo avança para a ocupação dos recursos físicos, utilizando semáforos para o controlo das salas e variáveis de condição para a gestão do pool de equipas médicas (*cond_var*), iniciando a cirurgia apenas quando todos os vetores estão assegurados.

A estratégia de prevenção adotada assenta em dois pilares:

1. Ordenação de Recursos: A aquisição de recursos segue uma hierarquia estrita. A verificação de disponibilidade de sala e equipa é feita de forma atómica ou numa sequência pré-determinada que impede a posse parcial de recursos (*Circular Wait).*
2. *Timeouts* e *Failsafes*: A implementação dos *wrappers* de sincronização (como *sem_wait_safe* e *safe_pthread_cond_timedwait*) introduz um fator temporal na espera. Nenhuma *thread* espera infinitamente por um recurso; se o tempo limite for excedido ou se o sistema entrar em *shutdown*, a espera é abortada, permitindo ao sistema recuperar o controlo e libertar quaisquer recursos já retidos.

4.4 Prevenção de *Starvation* (Inanição)

Para garantir que tarefas de baixa prioridade não sejam perpetuamente adiadas por tarefas de alta prioridade, o sistema implementa políticas de gestão de filas justas. No módulo da triagem, por exemplo, embora exista distinção entre urgências e consultas normais, a estrutura de dados (listas ligadas geridas por *mutexes*) e o algoritmo de escalonamento garantem que os pedidos são processados. Adicionalmente, o uso de filas FIFO para pedidos com a mesma prioridade assegura a equidade temporal no atendimento.

4.5 Segurança no Código

Dada a natureza concorrente da aplicação, todas as funções auxiliares foram desenhadas para serem *thread-safe* e reentrantes. Assim sendo, observou-se um cuidado adicional para evitar o uso de variáveis estáticas ou globais não protegidas dentro de funções e para que todos os *wrappers* tratassem os erros devidamente, garantindo que falhas na criação de *threads* ou na inicialização de *mutexes* são detetadas e registadas imediatamente, evitando comportamentos indefinidos em tempo de execução.

5 Algoritmos e Lógica de Negócio

Enquanto os capítulos anteriores se focaram na estrutura, este capítulo detalha dinâmica comportamental do sistema. Descrevem-se aqui os algoritmos que governam a tomada de decisão nos componentes críticos, justificando as opções tomadas para maximizar o fluxo de pacientes e a eficiência dos recursos.

5.1 Triagem

O algoritmo de gestão da triagem assenta num modelo de priorização dinâmica que gere duas filas de espera distintas protegidas por exclusão mútua. A fila de urgência implementa uma lógica de ordenação multinível onde a estabilidade crítica se sobrepõe ao nível de triagem e ao tempo de chegada, enquanto a fila de consultas obedece estritamente à ordem cronológica dos agendamentos. Para garantir a segurança clínica, um monitor de sinais vitais executa clinicamente um algoritmo de envelhecimento que decrementa a estabilidade dos pacientes e promovendo automaticamente para a fila de urgência quaisquer casos agendados que atinjam um limiar crítico definido, prevenindo a mortalidade por espera administrativa.

No que concerne ao escalonamento de tarefas e à prevenção de *starvation* das consultas perante fluxos intensos de urgências, foi implementada uma estratégia algorítmica de reserva de recursos. Esta lógica atribui comportamentos distintos às *threads* trabalhadoras com base no seu identificador único, onde a maioria dos recursos computacionais prioriza estritamente a fila da urgência, mas uma fração dedicada inverte essa prioridade para processar primariamente as consultas agendadas. Adicionalmente, a gestão de dependências externas (farmácia e laboratório) utiliza um modelo não bloqueante que transita pacientes para uma lista de espera pendente, libertando imediatamente as *threads* de processamento para novos atendimentos até à receção assíncrona das respostas necessárias.

A lógica apresentada acima foi ainda resumida num diagrama (ver Anexo A)

5.2 Cirurgia

O algoritmo do bloco operatório materializa a gestão de recursos críticos finitos através de uma máquina de estados complexa e descentralizada. A lógica assenta na coordenação estrita

de pré-requisitos onde nenhuma cirurgia avança sem a satisfação cumulativa de dependências logísticas (análises e medicamentos) e físicas (salas e equipas médicas).

Para maximizar a eficiência, o módulo implementa um padrão de *threads* trabalhadoras com gestão de suspensão. Cada nova cirurgia é atribuída a uma *thread* dedicada que, em vez de bloquear ativamente recursos enquanto aguarda por respostas externas, verifica a disponibilidade das dependências logísticas. Caso estas não estejam satisfeitas, a *thread* transita o estado da cirurgia para uma lista de pedidos pendentes (*pending_surgery_t*) e termina a sua execução, libertando recursos do SO. Um *dispatcher* central (*thread* principal) atua como reator a eventos, processando respostas assíncronas do laboratório ou farmácia e, apenas quando todas as pré-condições de uma cirurgia são validadas, reativa o processo.

A gestão dos recursos físicos utiliza primitivas de sincronização distintas para modelar a sua natureza. As salas de operação (BO1, BO2, BO3) são controladas por semáforos binários nomeados, garantindo a exclusão mútua no acesso ao espaço físico. Já as equipas médicas, sendo um recurso fungível (pool de N equipas), são geridas através de um contador protegido por um *mutex* e uma variável de condição global. Esta arquitetura impede o *deadlock* circular ao impor uma ordem hierárquica na aquisição de recursos: uma cirurgia só solicita uma sala após garantir que é a sua hora agendada e só solicita uma equipa médica após ter garantido a posse da sala, libertando ambos os recursos imediatamente após a conclusão do procedimento e higienização.

Para consolidar a lógica da cirurgia, criou-se um diagrama sobre a mesma, disponível no Anexo B.

5.3 Farmácia

O funcionamento da farmácia partilha o padrão arquitetural *dispatcher-worker* utilizado nos restantes módulos, onde uma *thread* principal consome pedidos de uma MQ e entrega o processamento a *threads* detalhadas (*detached*). Contudo, a lógica de negócio distingue-se pela implementação de um modelo transacional de acesso aos dados, desenhado para garantir a integridade do *stock* num ambiente de elevada concorrência (até 200 pedidos simultâneos).

O algoritmo de dispensa de medicamentos adota uma estratégia de reserva em duas fases para mitigar RCs. Quando um pedido é processado, a *thread* trabalhadora não se limita a verificar a existência de *stock* (*current_stockI*); ela valida a disponibilidade líquida (stock

menos reservas) e, atomicamente, incrementa um contador de *reserved*. Esta abordagem permite libertar o semáforo de acesso exclusivo durante a simulação da preparação (o tempo de espera), maximizando o paralelismo do sistema, pois outras *threads* podem consultar o *stock* enquanto uma preparação decorre. Apenas na fase final é que o semáforo é readquirido para converter a reserva em decremento efetivo do *stock* físico.

Adicionalmente, o módulo integra um mecanismo reativo de reposição automática de *stock*. Durante o ato de dispensa, o algoritmo verifica se o nível do medicamento desceu abaixo do *threshold*. Se tal ocorrer, é desencadeada uma reposição imediata e automática, garantindo a continuidade operacional sem necessidade de intervenção externa. O sistema suporta ainda o reabastecimento manual através de mensagens prioritárias (*MSG_RESTOCK*), assegurando que a gestão de inventário responde tanto a gatilhos algorítmicos como a comandos administrativos.

Mais uma vez, encontra-se presente no Anexo C um diagrama para ajudar na compreensão e análise do processo da farmácia.

5.4 Laboratório

O laboratório operacionaliza um modelo clássico de produtor-consumidor suportado por um pool de *threads* de dimensão fixa, desenhado para maximizar o débito de processamento de análises clínicas num ambiente de recursos limitados. A *thread* principal assume o papel de *dispatcher*, monitorizando continuamente a fila de mensagens externa para intercetar pedidos – com distinção implícita de prioridade – e encaminhando-os para uma fila de trabalhos interna protegida por mecanismos de exclusão mútua. Esta arquitetura promove um desacoplamento eficaz entre a receção do pedido e a sua execução, permitindo que o sistema amorteça picos de afluência provenientes das 3 fontes de pedidos sem bloquear os processos solicitantes, garantindo assim uma elevada disponibilidade do serviço.

No que concerne à execução técnica, o algoritmo implementa uma gestão rigorosa de alocação de recursos finitos, onde cada trabalhador compete pelo acesso a *slots* de equipamento específicos, controlados por semáforos. A lógica de negócio distingue inteligentemente entre testes simples, que requerem a posse atómica de um único recurso, e procedimentos complexos (como o teste de PREOP), que impõem uma sequencialidade restrita de execução, obrigando a *thread* a adquirir e libertar recursos em fases distintas e ordenadas. Após a simulação temporal do procedimento, o ciclo encerra-se com a persistência dos resultados em ficheiro e o envio

assíncrono de notificações de confirmação para a entidade de origem, restabelecendo o fluxo clínico do paciente.

5.5 Gestor Central (GC)

O GC implementa o núcleo algorítmico do sistema, funcionando como um motor de simulação de eventos discretos. A sua lógica baseia-se num ciclo de eventos multiplexado (*select*), que substitui a execução sequencial por um modelo reativo. Este ciclo monitoriza simultaneamente a entrada de comandos via terminal e FIFOs, bem como um canal dedicado à interceção segura de sinais (*Self-pipe trick*), garantindo que o sistema responde instantaneamente a interrupções externas sem bloquear o fluxo principal.

A gestão temporal é assegurada por um algoritmo de escalonamento dinâmico. O gestor calcula o tempo exato até ao próximo evento agendado ou batimento de relógio (*tick*) e suspende a execução apenas pelo período necessário. Esta abordagem não bloqueante permite o avanço determinístico do relógio lógico do hospital, sincronizando o envio de tarefas para os subsistemas no momento preciso de simulação.

Para o tratamento de respostas assíncronas, o módulo utiliza uma *thread* auxiliar como já mencionado, que é responsável por consumir o *feedback* dos processos filhos, libertando a *thread* principal para a gestão crítica do ciclo de vida e para a propagação coordenada de sinais de encerramento (*Poison Pills*), assegurando a consistência global do sistema.

No Anexo E podemos ver uma representação visual deste *workflow*.

6 Implementação

Neste capítulo será abordada a implementação das opções arquiteturais descritas anteriormente, detalhando as estruturas de dados fundamentais e os mecanismos de segurança implementados. Serão analisados alguns dos excertos de código mais críticos, apesar de não estarem presentes devido às suas dimensões, mas foram descritos de forma concisa.

6.1 Triagem

A triagem foi implementada no ficheiro *triage.*c. A implementação central de escalonamento reside na função *treatment_worker*, onde se materializa a estratégia de reserva de recursos para mitigar a *starvation*. Ao iniciar, cada *thread* avalia o seu identificador único: as *threads* generalistas (Ids 0 e 1) executam o bloco *else*, impondo a verificação primária e bloqueante da *emergency_queue*, enquanto a *thread* especialista (ID 2) entra no bloco *if (is_appointment_specialist)*, invertendo a prioridade para garantir o escoamento da *appointment_queue*. Esta bifurcação lógica assegura que matematicamente, 33% da capacidade de processamento está concentrada para atender consultas, independentemente da carga da urgência, com a benesse de contribuir para atender urgências caso não existam consultas.

Para garantir a eficiência de recursos, a função rejeita a espera ativa (*polling*) em favor de um modelo de coordenação passiva com proteção de concorrência. Como visível no início do ciclo *while*, as *threads* permanecem suspensas na variável de condição *patient_ready_cond* e só acordam quando o contador atómico *pending_patients_count* sinaliza trabalho efetivo. A aquisição dos *mutexes* das filas (*emergency_queue.mutex* ou *appointment_queue.mutex*) é feita de forma granular e apenas no momento exato da extração do nó da lista, minimizando a secção crítica e permitindo que o monitor de sinais vitais insira e promova pacientes concorrentemente sem causar contenção excessiva.

Por fim, a gestão de dependências bloqueantes é resolvida no final do fluxo da mesma função. Caso a triagem determine a necessidade de exames ou medicação, o *worker* não bloqueia à espera de resposta; em vez disso, gera um *operation_id*, envia pedidos assíncronos via MQ e move a estrutura do paciente para uma lista de suspensão (*add_to_pending*). Esta abordagem liberta imediatamente a *thread* de tratamento para retomar o ciclo e processar o próximo paciente da fila, delegando a espera das respostas para a *thread* dedicada *response_dispatcher*.

6.2 Cirurgia

A materialização da máquina de estados descentralizada ocorre na função *surgery_worker* (ficheiro *surgery.c*), que orquestra o ciclo de vida da intervenção. A implementação traduz a lógica de negócio numa sequência linear de etapas de validação: inicialmente, são disparados os pedidos assíncronos sem bloquear a execução. O ponto crítico reside na função *wait_for_dependencies*; se as dependências não estiverem satisfeitas após o *timeout* inicial, a função retorna um código de suspensão. O fluxo desvia então para *put_on_hold*, onde o estado da cirurgia é persistido na memória (*add_to_pending*) e a *thread* termina a sua execução, libertando recursos do sistema operativo enquanto aguarda eventos externos.

A prevenção de *deadlocks* é implementada através da rigidez sequencial na aquisição de recursos. O código impõe que a reserva do espaço físico (*acquire_room*), que utiliza semáforos nomeados) ocorra estritamente antes da requisição de recursos humanos (*acquire_medical_team*). Esta hierarquia de bloqueio, combinada com verificações constantes da *flag* de *shutdown* após cada aquisição, elimina a possibilidade de espera circular e garante que o sistema pode ser encerrado graciosamente mesmo durante a fase de alta contenção de recursos.

O mecanismo de reativação é gerido pelo *dispatcher* (*loop* principal). Ao receber mensagens de *MSG_LAB_RESULTS_READY* ou *MSG_PHARM_READY*, este verifica a lista de pendentes e, caso as pré-condições sejam atingidas, invoca *spawn_surgery_from_pending*. Esta função instancia uma nova *thread* que executa uma variante do *worker* (*surgery_worker_resumed*), desenhada para "saltar" as etapas de validação logística já superadas e iniciar diretamente a fase de espera temporal e aquisição de recursos físicos, completando assim o processo iniciado pela *thread* original.

6.3 Farmácia

A implementação da lógica principal já mencionada sobre a farmácia está presente no ficheiro *pharmacy.c* e é observável principalmente na estrutura da função *pharmacy_worker_thread*, que orquestra a transação. O código impõe uma sequência rigorosa de bloqueios: inicialmente, a *thread* adquire o semáforo de acesso global (*acquire_pharmacy_access*) para executar a fase de validação e reserva

(*check_stock_availability* e *reserve_stock*). O detalhe técnico mais relevante ocorre imediatamente após esta fase: a *thread* liberta explicitamente o semáforo (*release_pharmacy_access*) antes de entrar na simulação temporal (*wait_time_units*). Como já explicado antes, esta decisão é fundamental para melhorar a performance. Apenas após a preparação o semáforo é readquirido para a fase de *commit* (*dispense_medications*), onde a reserva é convertida em decremento de *stock*.

A integridade dos dados ao nível do item é assegurada dentro das funções auxiliares, nomeadamente em *dispense_medications*. Em vez de um bloqueio único para todo o armazém, o código itera sobre o vetor de medicamentos e adquire um *mutex* específico para cada ID (*shm_pharm->medications[med_id].mutex*). É dentro deste contexto atómico que reside a lógica de reposição *inline*: após o decremento do *stock*, uma instrução condicional verifica imediatamente se o valor resultante é inferior ao *threshold*. Se a condição se verificar, o reabastecimento é efetuado na mesma operação atómica, prevenindo que qualquer outra *thread* leia um estado de rutura de *stock* intermédio e garantindo a consistência contínua do inventário.

6.4 Laboratório

A implementação do laboratório concretiza o padrão produtor-consumidor, no ficheiro *lab.c*, através de uma estrutura de dados dedicada, a *job_queue*, que serve de *buffer* intermédio entre a receção de pedidos e a sua execução. O acesso a esta estrutura é estritamente controlado por um *mutex* (*job_queue.mutex*), garantindo a integridade dos dados durante as operações de inserção (*job_queue_push*) pelo *dispatcher* e remoção (*job_queue_pop*) pelos trabalhadores. A coordenção entre estas entidades é assegurada por uma variável de condição (*job_queue.cond*): quando um trabalhador tenta retirar um trabalho e a fila está vazia, ele entra num estado de bloqueio eficiente, sendo acordado apenas quando o produtor insere um novo item ou quando é emitido um sinal de encerramento global.

A gestão de recursos finitos é implementada na função *execute_normal_test* através de semáforos. Antes de iniciar qualquer análise, a *thread* trabalhadora deve adquirir um *token* do semáforo correspondente ao tipo do laboratório (*sem_lab1* ou *sem_lab2*). Se todos os *slots* estiverem ocupados, a *thread* bloqueia ao nível do SO, garantindo que o número de análises simultâneas nunca excede a capacidade física configurada. Adicionalmente, para procedimentos complexos como o teste de PREOP (*execute_preop_test*), o código impõe uma

aquisição sequencial de recursos, como visível no Anexo D, prevenindo *deadlocks* ao garantir que uma *thread* liberta o primeiro recurso antes de solicitar o segundo.

6.5 Gestor Central (GC)

A lógica do GC está encapsulada num *event loop* reativo, implementado através do *select()*. Este mecanismo permite monitorizar simultaneamente descritores de ficheiros distintos (como o *pipe* de comandos e o *self-pipe* de sinais) e integrar a gestão temporal sem recorrer a *polling* ineficiente. O aspeto crítico da implementação é o cálculo dinâmico do *timeout* para o *select()*: em cada iteração, o sistema avalia o tempo restante até ao próximo evento agendado ou batimento de relógio. Se ocorrer uma interrupção externa (como um sinal ou comando), o *kernel* acorda o processo imediatamente; caso contrário, o *timeout* expira precisamente no momento necessário para avançar o relógio de simulação e despachar novas tarefas para os subsistemas, garantindo assim uma sincronização temporal determinística e eficiente.

7 Testes e Validação

A garantia de qualidade de um sistema tão complexo, crítico (caso fosse implementado) e concorrente como este exige uma metodologia de validação rigorosa, capaz de simular cenários de alta entropia que excedem a capacidade de teste manual. Este capítulo detalha a infraestrutura de testes desenvolvida, os cenários de carga submetidos e a análise quantitativa do comportamento do sistema.

7.1 Estratégia de Testes

A estratégia de validação adotou uma abordagem híbrida e incremental, evoluindo de testes unitários funcionais para testes de integração sistémica e, finalmente, testes de stress.

Para suportar esta estratégia, foi desenvolvido um módulo para a automação de testes (localizada em *tests/*), composta por duas camadas distintas:

1. Geradores (em Python): Scripts localizados em *tests/generators/* (ex: *gen_stress_triage.py*, *gen_stress_global.py*) responsáveis por criar volumes massivos de comandos sistematicamente válidos, introduzindo aleatoriedade controlada nos tempos de chegada e prioridades para simular picos de afluência realistas.
2. Orquestradores de Execução (*Shell*): Scripts em *tests/runners/* que automatizam o ciclo de vida do teste: compilam o projeto, lançam o sistema, injetam a carga gerada via *input_pipe*, monitorizam a execução e recolhem os *logs* e relatórios estatísticos finais.

7.2 Casos de Teste

O sistema foi submetido a quatro cenários de validação críticos, desenhados para isolar e verificar componentes específicos da arquitetura:

- CT1 – Saturação da Triagem (*run_triage_test..sh*): Injeção de 500 pacientes num intervalo curto, misturando urgências e consultas agendadas.
    - Objetivo: Validar se o algoritmo de reserva de recursos previne eficazmente a *starvation* das consultas perante um fluxo contínuo de emergências e se a ordenação multinível da fila de urgência é respeitada.

- CT2 – Concorrência na Farmácia (*run_pharmacy_restock_test.sh*): Pedidos simultâneos de múltiplos medicamentos que excedem o *stock* inicial.
    - Objetivo: Verificar a eficácia dos *mutexes* e do mecanismo de a*uto-restock*, garantindo que não existem RCs ou corrupção do inventário partilhado.
- CT3 – Complexidade Cirúrgica (*run_surgery_test.sh*): Agendamento de cirurgias com conflitos de recursos (mesmas salas e equipas limitadas).
    - Objetivo: Confirmar a ausência de *deadlocks* na aquisição hierárquica de recursos (sala e equipa) e a correta suspensão/retoma de cirurgias com dependências pendentes.
- CT4 – Stress Global (*run_global_test.sh*): Simulação de longa duração envolvendo todos os módulos simultaneamente.
    - Objetivo: Testar a estabilidade dos mecanismos IPC e a robustez do GC na orquestração de eventos díspares.

7.3 Resultados

Os resultados obtidos demonstraram a resiliência da solução arquitetural proposta, com alguns resultados interessantes. No cenário CT1, observou-se que, mesmo com a fila de urgência permanentemente ocupada, a *thread* especialista (ID 2) conseguiu processar consistentemente os agendamentos, mantendo o *throughput* de consultas dentro dos parâmetros esperados. O CT2 confirmou que o mecanismo de bloqueio impediu a corrupção do inventário mesmo sob disputa simultânea de recursos, validando a inexistência de RCs e a precisão do *stock* final. No CT3, o sistema geriu corretamente a escassez de salas operatórias; as cirurgias excedentes aguardaram na lista de pendentes sem bloquear o processamento de respostas do laboratório, confirmando a eficácia do modelo assíncrono. Em todos os cenários testados, o sistema encerrou graciosamente após a receção do sinal SIGINT, sem deixar processos *zombie* ou recursos órfãos, validando a lógica de propagação de *Poison Pills*.

O CT4 foi o teste mais crítico, onde o sistema foi levado aos seus limites. Os resultados comprovaram que o sistema conseguiu manter a coerência e trabalhar sem erros em todos os cenários e subsistemas, no entanto, demonstrou também um *bottleneck* na parte do laboratório – os pedidos eram muitos para os recursos que estavam disponíveis (já que é o módulo mais solicitado), e então, conclui-se que se trata de insuficiência de recursos e não de uma ineficiência causada por *software*.

7.4 Análise de Performance

A análise do consumo de recursos evidenciou a eficiência das primitivas de sincronização escolhidas. A utilização de CPU manteve-se residual (< 5%) durante períodos de inatividade, fruto da utilização exclusiva de bloqueios passivos (*sem_wait*, *pthread_cond_wait*, *select*) em detrimento de esperas ativas.

O mecanismo de multiplexagem de I/O no GC permitiu processar milhares de eventos com latência negligenciável, enquanto a granularidade dos *mutexes* na SHM permitiu um elevado grau de paralelismo, evitando que operações de leitura bloqueassem escritas em recursos não relacionados.

7.5 Verificações com Valgrind

A integridade da gestão de memória e concorrência foi auditada utilizando a ferramenta *Valgrind*, com 3 focos principais:

1. *Memcheck*: A execução confirmou a ausência de *memory leaks* definitivos. As funções de limpeza (*child_cleanup* e *manager_cleanup*) libertaram corretamente toda a memória dinâmica alocada e fecharam todos os descritores de ficheiros e *handles* de IPC no momento do encerramento. De forma resumida, nos testes efetuados não foram encontrados erros e não houve bytes perdidos.
2. *Helgrind*: A análise foi fundamental para refinar a proteção das secções críticas e levou à alteração de blocos e ao acréscimo de novos *mutexes*, como para a verificação das *queues* na triagem. No final, todas as variáveis estavam protegidas por *mutexes*, não existindo acessos concorrentes não sincronizados.
3. *DRD*: Por utilizar um algoritmo de deteção distinto do *Helgrind*, esta análise permitiu filtrar potenciais falsos positivos e reforçar a garantia de que o acesso à SHM e a sincronização das *threads* ocorrem sem conflitos de dados ou violações da API POSIX.

Nos documentos enviados, podem ser encontrados exemplos de alguns dos *logs* resultantes destas verificações, comprovando-se o que foi mencionado acima. Podem, no entanto, existir erros suprimidos pelo *valgrind*, mas que são desprezáveis para o bom funcionamento do programa.

8 Estatísticas e Análise

A monitorização do sistema foi assegurada pelo módulo de estatísticas (*stats.c*), que manteve uma imagem em tempo real do estado operacional através de SHM. A análise apresentada baseia-se num *snapshot* final (*stats_snapshot.txt*) recolhido após uma execução contínua do teste de stress global de 1633 segundos (aproximadamente 27 minutos), complementado pela análise dos *logs* de eventos.

8.1 Métricas de *Throughput*

O *throughput* global do sistema representa a capacidade de processamento de pacientes por unidade de tempo, validando a eficiência da arquitetura concorrente.

Considerando o tempo total de execução e o volume de pacientes admitidos e processados na triagem (64 urgências e 46 consultas), calcula-se o débito de admissão:

$$Throughput_{Admissão} = \frac{64 + 46}{1633} \approx 0{,}067\ pacientes/segundo$$

No que toca aos setores especializados, observam-se os seguintes débitos:

- Cirurgia: Com 59 cirurgias concluídas, o sistema manteve um ritmo de 2,16 cirurgias/minuto. Dado que as cirurgias são processos longos e bloqueantes (ocupam salas), este valor indica uma elevada taxa de ocupação dos blocos operatórios.
- Laboratório: O laboratório processou um total de 495 testes (204 + 291), resultando num débito elevado de 18,18 testes/minuto. Este valor confirma que o modelo de *thread* pool foi capaz de absorver picos de pedidos provenientes simultaneamente da triagem e dos blocos operatórios.

8.2 Análise de Latência e Tempos de Espera

A análise de tempos de espera é crucial para validar a eficácia das políticas de escalonamento, sobretudo na triagem, onde existe risco de inanição de pacientes não urgentes. Os dados do *snapshot* final evidenciam um comportamento positivo do algoritmo de reserva de recursos: o tempo médio de espera das consultas agendadas (106,33 UT) foi significativamente

inferior ao das urgências (165,77 UT), com uma diferença aproximada de 59,44 UT. Este resultado valida a decisão de dedicar uma *thread* especialista exclusivamente às consultas, garantindo o cumprimento dos horários mesmo num cenário com maior volume de urgências.

No laboratório, o tempo médio de respostas foi de 15,91 UT, um valor muito reduzido face à duração dos testes, indicando que a pool de 5 *threads* foi suficiente para processar pedidos quase em tempo real, minimizando bloqueios nas cirurgias e na triagem.

A análise da utilização de recursos permitiu identificar gargalos relevantes. Observou-se uma assimetria nos blocos operatórios, com o BO3 próximo da saturação (60,06%) e o BO2 subutilizado (20,58%), refletindo a distribuição desigual dos tipos de cirurgia. O principal gargalo do sistema encontra-se no laboratório 2, cuja taxa de utilização atingiu 95,84%, contrastando com os 48,67% do laboratório 1, devido à concentração de testes complexos e à sua participação nos exames PREOP. Por fim, a farmácia demonstrou elevada eficiência, processando 193 pedidos sem ruturas de *stock*, o que confirma a correta calibração dos limiares de reabastecimento automático.

9 Análise Crítica

Este capítulo encerra o documento com uma reflexão sobre as virtudes e limitações da arquitetura implementada, discutindo as dificuldades técnicas superadas e propondo roteiros para a evolução futura do sistema hospitalar simulado.

9.1 Dificuldades Encontradas

Durante o desenvolvimento deste projeto foram encontradas diversas dificuldades, principalmente resultante de erros/*bugs*. Foi bastante complicado definir uma arquitetura exata, pelo que esta sofreu diversas alterações ao longo do desenvolvimento, por novas descobertas relativamente ao enunciado do projeto que, pela sua dimensão, foi difícil de dissecar e compreender na totalidade.

Relativamente a erros técnicos, encontrámos alguns bastante interessantes e que nos permitiram perceber bem como tudo se interligou neste projeto. Por exemplo, antes da implementação do bater de relógio do GC, algumas funções de simulação de tempo bloqueavam infinitamente, e foi bastante intuitivo aprender a localizar a origem do erro. Ademais, enfrentaram-se muitas dificuldades a coordenar o *shutdown* do programa. Como foram utilizadas *poison pills*, ocorreram por duas vezes erros em que, por existirem RC, um processo enviava uma *poison pill* que era "roubada" por outro processo, levando a que algum deles nunca terminasse. Este *bug* foi muito maçador, mas interessante de corrigir uma vez que nos capacita com outra visão de programas concorrentes como este e os riscos que corremos, sem muitas vezes nos apercebermos. Por fim, ainda sobre *poison pills* e MQs, enfrentámos um *deadlock* ao correr o teste de stress da farmácia. De forma resumida, este erro deveu-se ao facto da farmácia enviar as notificações com o *mtype* errado, levando a que a *thread* das notificações no GC não as lesse, ficando assim presas na MQ (pois existe um limite). Ao fazer isto, quando tentávamos enviar a *poison pill* para terminar a *thread* de notificações, esta nunca era recebida pois não conseguia ser enviada (MQ cheia).

9.2 Limitações

Apesar da robustez demonstrada, a arquitetura atual apresenta limitações decorrentes da sua conceção centralizada em recursos locais. A dependência exclusiva de mecanismos de IPC

System V e POSIX restringe a execução a uma única máquina de mecanismos e escalabilidade horizontal. Adicionalmente, a ausência de persistência em disco torna o sistema volátil, podendo uma falha grave resultar na perda do estado das filas e estatísticas, o que é inaceitável num contexto hospitalar. Por fim, a configuração estática dos pools de *threads* limita a adaptação a picos de carga, criando gargalos que poderiam ser evitados com escalonamento dinâmico. Apesar disto, não poderiam crescer muito mais (principalmente o laboratório devido a exigências do enunciado).

9.3 Considerações/Reflexões finais

De forma geral, consideramos que este trabalho nos proporcionou uma visão muito valiosa sobre SO e como são construídos sistemas concorrentes, as suas potenciais falhas e os cuidados a ter ao fazê-lo. Melhorámos o nosso conhecimento da linguagem C e também o uso de comandos UNIX.

Tentámos superar-nos ao fazer este trabalho e entregar algo completo, funcional e eficiente. De modos gerais, cremos que conseguimos alcançar esses objetivos.

10 Conclusão

O presente projeto permitiu a implementação bem-sucedida de um Sistema Integrado de Gestão Hospitalar, materializando os conceitos fundamentais de SO num cenário de simulação complexo e realista. O objetivo primário de desenvolver uma solução robusta em ambiente UNIX/Linux, capaz de gerir recursos partilhados sob condições de elevada concorrência, foi plenamente atingido.

A arquitetura modular e multiprocesso adotada demonstrou ser uma escolha acertada garantindo o isolamento de memória e a estabilidade operacional entre os subsistemas críticos. A estratégia de comunicação híbrida – combinando a assincronia das MQ para o despacho de tarefas com a eficiência da SHM para a gestão do estado global – provou ser capaz de suportar um elevado débito de processamento sem comprometer a latência do sistema.

Do ponto de vista algorítmico, destaca-se a eficácia das estratégias de sincronização desenhadas para resolver problemas clássicos de concorrência. A implementação de uma hierarquia restrita na aquisição de recursos nos blocos operatórios eliminou a ocorrência de *deadlocks*, enquanto a lógica de reserva de recursos na triagem mitigou eficazmente a *starvation* das consultas agendadas, garantindo a equidade no atendimento. A validação destes mecanismos através de testes de stress automatizados e análise dinâmica de memória e RC (com o *valgrind*) reforçou a confiança na ausência de RC e *memory leaks*.

Como perspetiva de evolução futura, o sistema beneficiaria da implementação de mecanismos de elasticidade dinâmica, permitindo o ajuste automático do número de *threads* trabalhadoras em tempo de execução consoante a carga instantânea do sistema (escalonamento vertical), em vez de depender de limites estáticos. Seria igualmente pertinente a implementação de persistência de dados em disco e a capacidade de recarregar configurações sem reiniciar o processo (*hot-reloading*), permitindo ao sistema recuperar o seu estado operacional após falhas críticas e adaptar-se a novos parâmetros sem interrupção de serviço.

Em suma, o sistema desenvolvido não só cumpre todos os requisitos funcionais e não-funcionais propostos, como demonstra uma gestão eficiente e segura dos recursos computacionais, validando a aplicação prática dos paradigmas de programação concorrente e de sistemas em tempo real.

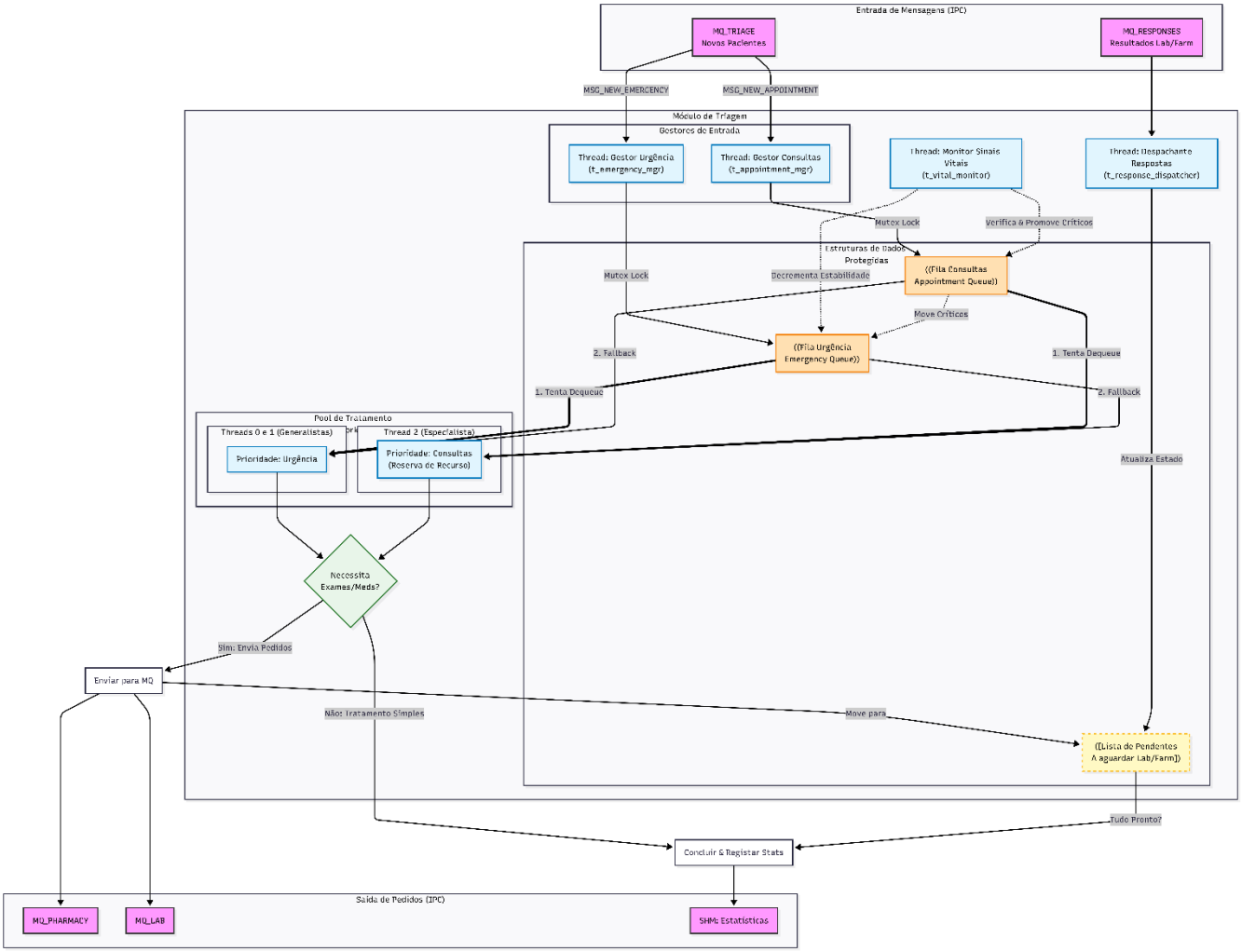
Referências Bibliográficas

Engineers), I. (. (2017). *IEEE Std 1003.1™ — Portable Operating System Interface (POSIX)*. New York, EUA: IEEE.

project, T. L.-p. (s.d. (atualizado continuamente)). *Linux Programmer's Manual (man pages)*. Linux Foundation / Comunidade Open-source.

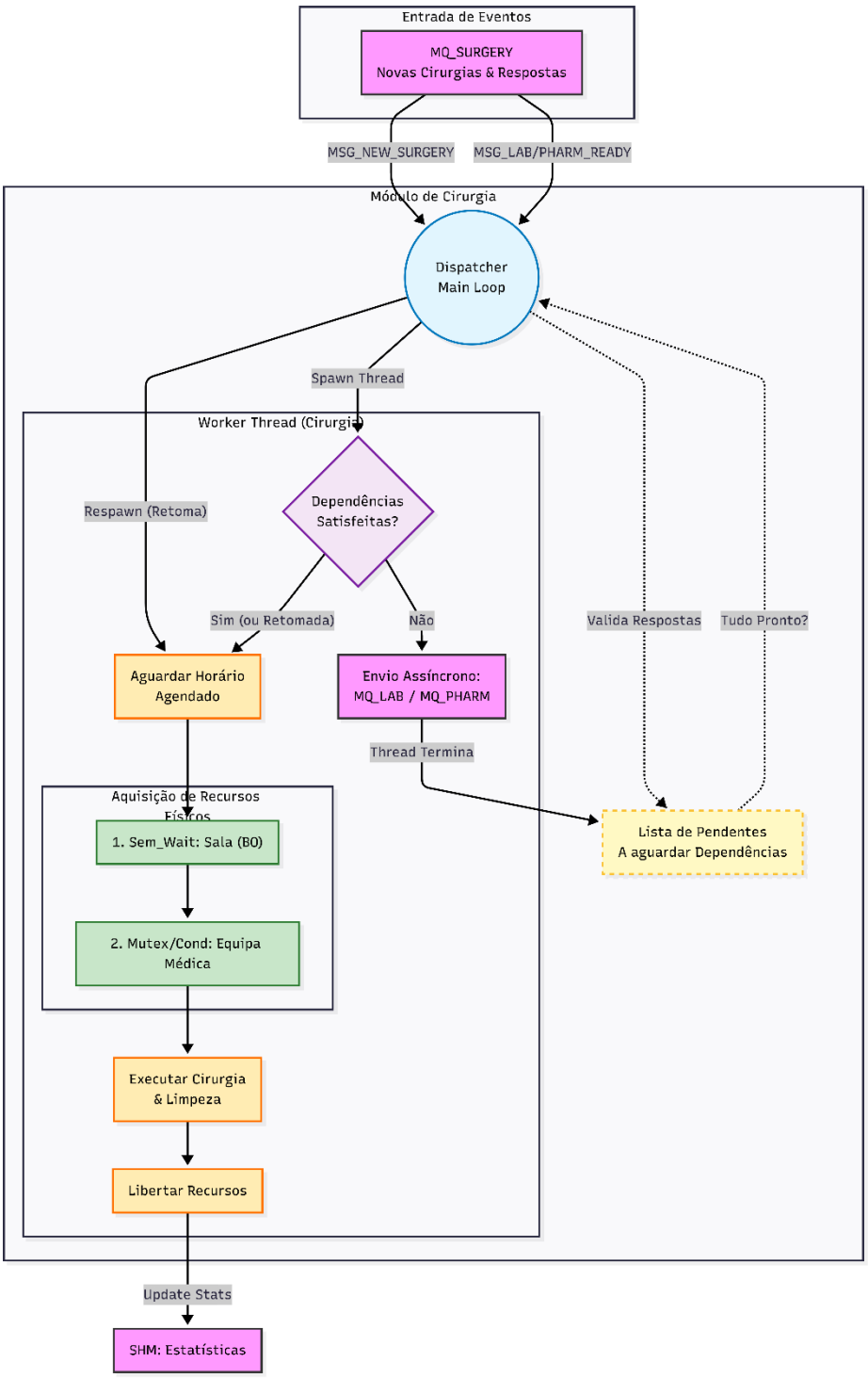
Anexo A

Nota: O diagrama encontra-se em formato *.png* mas estará disponibilizada uma versão *.svg* no conteúdo enviado, na pasta docs/diagrams/worflows/triage.



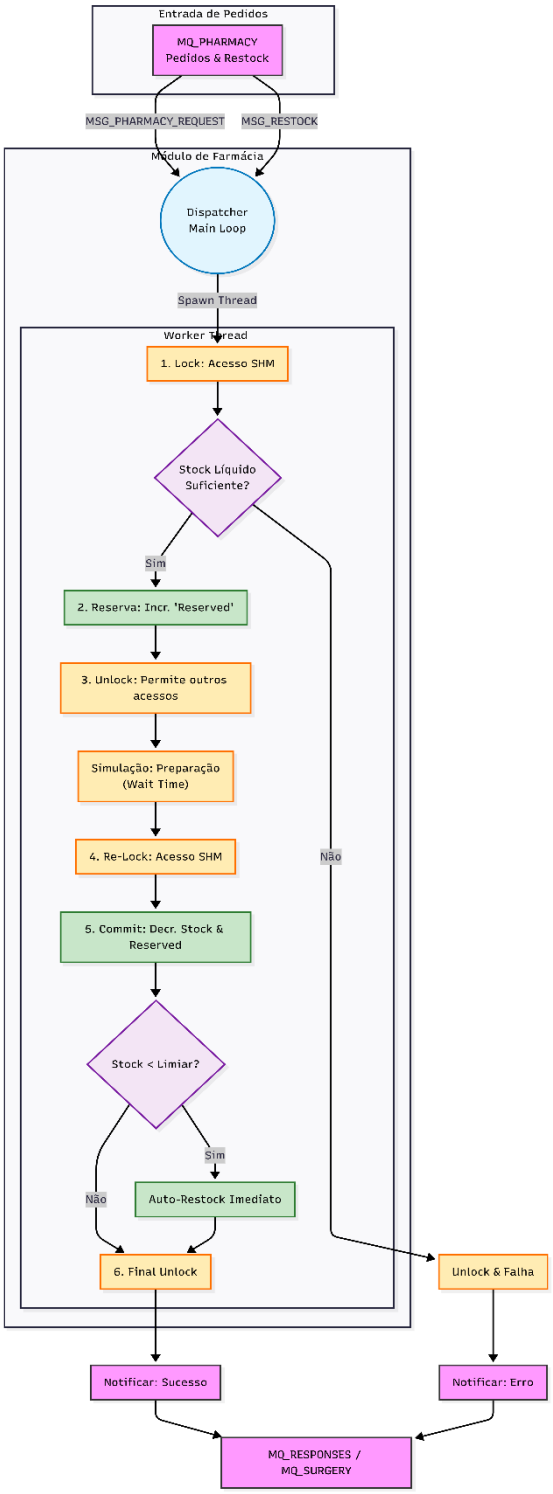
Anexo B

Nota: O diagrama encontra-se em formato *.png* mas estará disponibilizada uma versão *.svg* no conteúdo enviado, na pasta docs/diagrams/worflows/surgery.



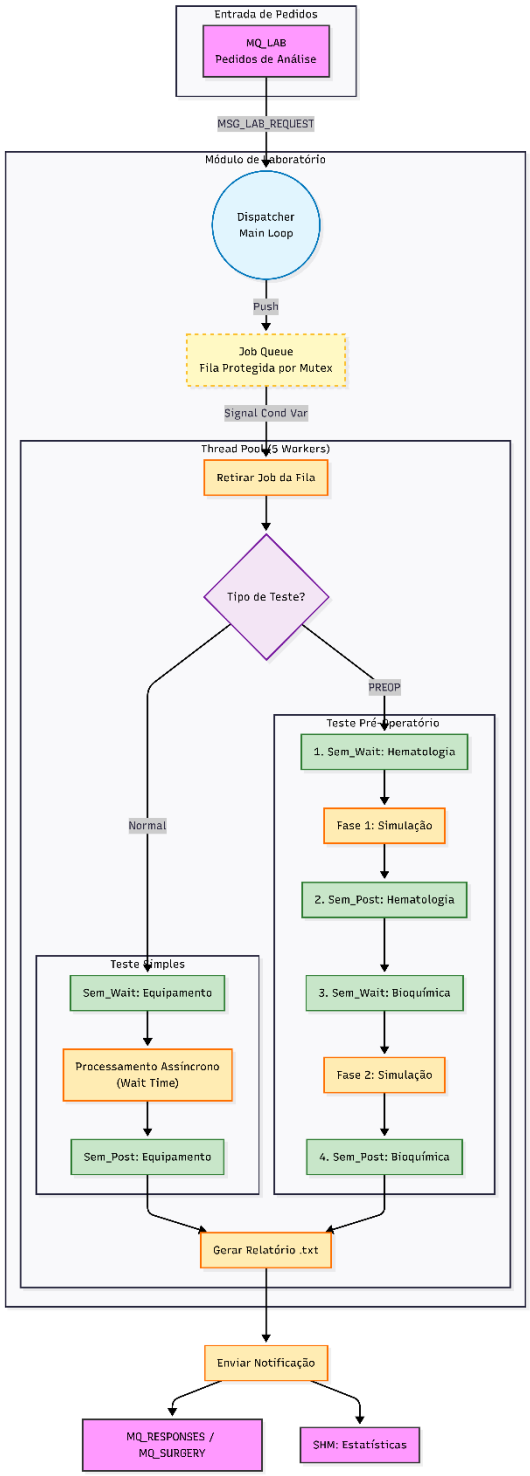
Anexo C

Nota: O diagrama encontra-se em formato *.png* mas estará disponibilizada uma versão *.svg* no conteúdo enviado, na pasta docs/diagrams/worflows/pharmacy.



Anexo D

Nota: O diagrama encontra-se em formato *.png* mas estará disponibilizada uma versão *.svg* no conteúdo enviado, na pasta docs/diagrams/worflows/laboratory.



Anexo E

Nota: O diagrama encontra-se em formato *.png* mas estará disponibilizada uma versão *.svg* no conteúdo enviado, na pasta docs/diagrams/worflows/manager.