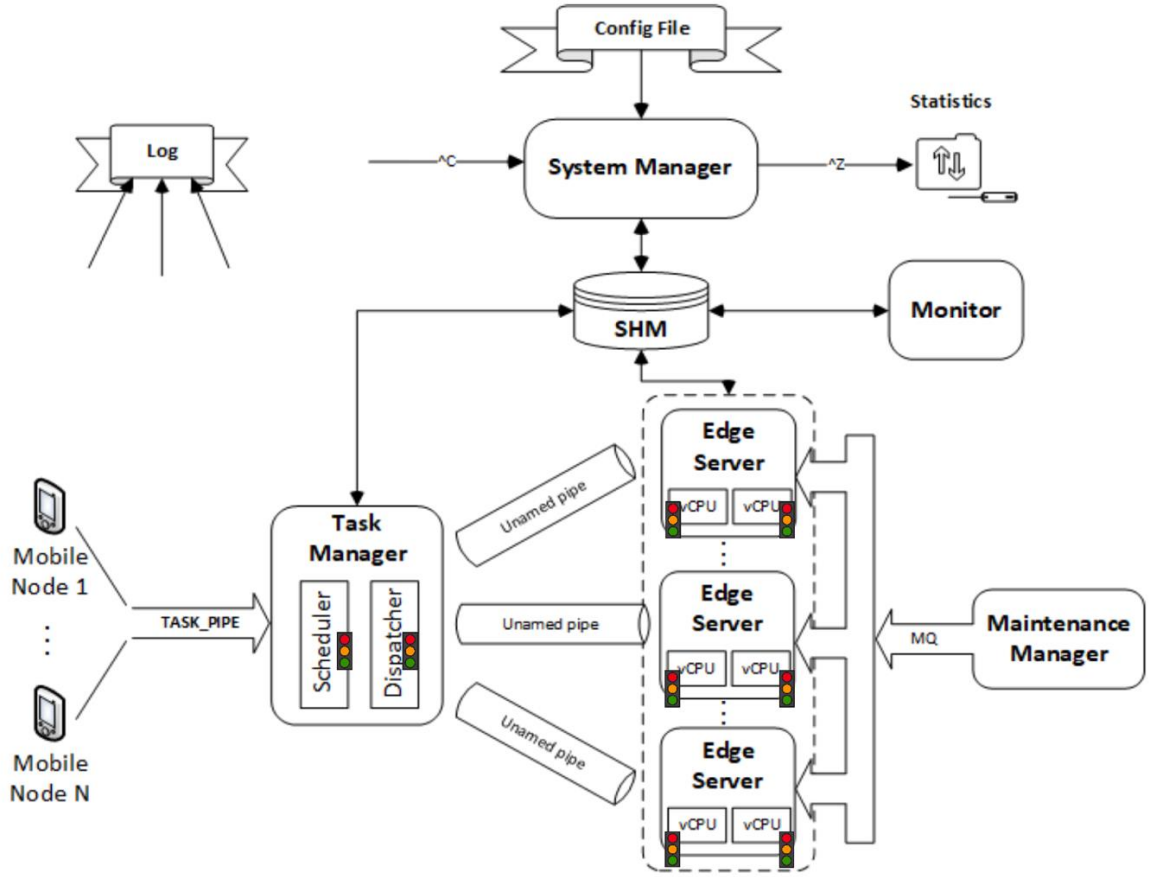
Relatório Projeto SO
Offloading Simulator

Filipe David Amado Mendes
Nº Estudante: 2020218797
Miguel Ângelo Graça Meneses
Nº Estudante: 2020221791
Docente Responsável: Prof. Doutor Vasco Pereira



Introdução

Offloading Simulator é um projeto que tem como objetivo simular a troca de mensagens entre diferentes dispositivos móveis e um computador, que executa os pedidos realizados. Para tal, são implementados mecanismos de sincronização, mecanismos de comunicação entre processos, como pipes, sinais, entre outras funcionalidades.
Este processo foca-se na aprendizagem dos fundamentos de sistemas operativos e como estes tratam dos processos e das mensagens que têm que processar.

Makefile

O ficheiro makefile apenas e usado para a compilação do programa. É um ficheiro simples, visto que apenas existem dois ficheiros a ser compilados, mobile_node.c e offloading_simulator.c.

Header file

Para inicialização das funções, criação das structs utilizadas e inicialização das variáveis globais foi criado um header, declarations.h. A utilização de um header facilita a programação. Permite organizar o código, evitando ter as declarações necessárias no topo do ficheiro .c.

Mobile node

O ficheiro mobile_node.c implementa os dispositivos móveis que enviam mensagens para o computador. Para isso o ficheiro recebe como argumeto o número de pedidos a gerar, o

intervalo entre pedidos em ms, os milhares de instruções de cada pedido e o tempo máximo para execução.
A função main abre o named pipe (TASK_PIPE) para escrita e cria o processo do mobile_node. O processo mobile_node envia os pedidos através do TASK_PIPE até que o número de pedidos acabe, enviando em alguns casos o comando "STATS" ou "EXIT".
Apesar da implementação do mobile_node não ter uma grande dificuldade, vários problemas surgiram à medida que o programa foi ficando completo. A maioria foi resolvido, porém um erro persistiu e não foi possível resolvê-lo. Para a abertura do TASK_PIPE, visto que o mobile_node apenas pode enviar mensagens e não recebê-las, foi usada a flag O_WRONLY. Porém, apesar de já ter sido criado, o programa não reconhece o named pipe e devolve um erro dizendo que não existe um ficheiro TASK_PIPE, mesmo incluindo o path completo. Para resolver este problema foram então usadas as flags O_RDWR | O_NONBLOCK. Este método resolveu o problema da abertura, porém mesmo assim, o programa não faz a escrita no TASK_PIPE, apesar de estar implementado.

Offloading simulator

No ficheiro offloading_simulator.c estão implementados todos os processos necessários, assim como as threads utilizadas.
A função main inicializa o programa. Começa por abrir, ou criar caso não exista, o ficheiro log, cria o TASK_PIPE, para leitura no Task_Manager, e a Message Queue que será utilizada pelo Maintenance_Manager para comunicar com os Edge_Servers. De seguida a função main faz a leitura do ficheiro de configuração, passado como argumento na execução do programa, e faz a criação da memória partilhada. Por último, cria todos os processos necessários para correr o programa.
O processo Task_Manager começa por alocar memória para a Task Queue, por criar os processos dos Edge_Servers e por abrir o TASK_PIPE para leitura. O mesmo problema de abertura do named pipe ocorreu neste ficheiro, então foram utilizadas as mesmas flags para a abertura (O_RDWR | O_NONBLOCK). De seguida cria as threads thread_sheduler e thread_dispatcher, que apenas terminam quando o programa acaba. Por fim, até que o programa acabe, o Task_Manager lê os pedidos recebidos no TASK_PIPE e coloca-os na fila de tarefas, enviando-os através de unnamed pipes para cada Edge_Server.
Cada processo Edge_Server correr continuamente até que o programa acabe. Começa por fazer a leitura das tarefas recebidas no unnamed pipe que lhe esta associado através da shared memory. Caso receba uma mensagem de manutenção, este termina as tarefas que está a fazer e de seguida entra em manutenção por um período de tempo aleatório. Caso não receba nenhuma mensagem, então verifica o estado de performance e envia a próxima tarefa para o vCPU pretendido.
O Monitor tem apenas uma função no programa, verificar o estado da Task Queue e alterar a performance dos Edge_Servers consoante o número de tarefas na fila.
O Maintenance Manager escolhe aleatoriamente um Edge_Server para entrar em manutenção, enviando uma mensagem de aviso para o escolhido, esperando por uma mensagem de confirmação de volta e enviando aprovação para o que o Edge_Server inicie efetivamente a manutenção. Por último espera um intervalo de tempo aleatório e escolhe outro edge_server id para enviar mensagem. Para que nunca estejam todos os Edge_Servers em manutenção em simultâneo, uma variável global e aumentada e diminuída sempre que começa e acaba uma manutenção, respetivamente.
Para os vCPUs foram criadas duas threads: slow_vCPU e fast_vCPU. Estes apenas executam as tarefas, esperando o tempo de execução delas, ou seja, a performance do respetivo vCPU * número de intruções a realizar (em milhões). Para evitar que mais que uma tarefa seja enviada para o vCPU, pthread_mutexes foram implementados como mecanismo de sincronização.

Bloqueando no início da thread e libertando no final, evita que o Edge_Server consiga envias duas ou mais tarefas ao mesmo tempo.
O thread_scheduler é uma thread que verifica continuamente todas as taferas na Task Queue e as ordena consoante a sua prioridade. Além disso, caso a tarefa já não tenha tempo de ser executada e eliminada. Para que não sejam lidas mensagens novas durante o ordenamento foi implementado um mutex, que da lock no início da thread e unlock no final.
O thread_dispatcher é semelhante ao thread_scheduler, porém a sua única função é verificar se as tarefas ainda têm tempo de ser executadas. De forma semelhante ao thread_scheduler, para que não sejam lidas mensagens novas enquanto e feita a verificação das existentes na fila, foi implementado de forma igual um mutex.
Para controlo de sinais existem duas funções. A função sigint termina o programa, eliminando as tarefas que ainda estão a ser executadas e chamando as funções statistics e clean_resources. Por outro lado, a função statistics apenas escreve no ecrã as statisticas do programa.
Por último, a função clean_resources fecha os ficheiros abertos, desaloca a memoria partilhada e a Task Queue e fecha a fila de mensagens e o TASK_PIPE.

Shared memory

Para a shared memory, foi criada uma struct EdgeServer. A struct contém a informação sobre os vCPUs, as threads, a performance, os unnamed pipes, entre outras variáveis utilizadas pelos processos. Para criação da shared memory for criado um array de EdgeServers, cada um contendo a informação relativa ao seu EdgeServer.

Mecanismos de sincronização

Inicialmente, para a sincronização dos processos tinham sido implementados semáforos e para as threads mutexes. Os mutexes das threads dão lock no início da thread e unlock no final, permitindo assim que apenas uma tarefa seja analizada ou realizada de cada vez.
Porém, a implementação de semáforos deu bastantes erros e decidi que seria melhor não os implementar. Depois de vários testes, tanto em MacOS como em Linux, os erros foram sempre os mesmos. O compilador de C dá um erro e diz que a função sem_wait.c não existe na biblioteca. Depois de bastante pesquisa, encontrei várias documentações oficiais do Linux que mostram que a função sem_wait foi descontinuada. Assim preferi apenas implementar mutexes e retirar os semáforos, visto que para a implementação destes a função sem_wait seria essencial.

Divisão de esforço

Infelizmente, neste tópico não posso ser positivo. Este projeto foi bastante trabalhoso e demorado, com muita pesquisa e esforço envolvido. Porém este esforço veio apenas de um lado, visto que o meu colega Miguel não nada.
Visto que eu tenho um Mac M1, tem sido bastante difícil conseguir instalar uma VM para conseguir instalar Linux. Acabei por instalar um programa chamado Parallels, porem é pago e só consegui a versão de teste, que tem um tempo limitado. Para a primeira defesa, mesmo sabendo que eu não conseguia correr o código do projeto no meu computador, o Miguel não instalou a VM aconselhada pelo professor e nem sequer levou o computador para a defesa. Por isso, tive que mostrar o código no meu computador e pedir ao professor que testasse no seu. Além disso, não leu o código e nem tentou perceber o que eu tinha feito.
Para a entrega final, eu fiz o código todo e o relatório. Não consegui reunir uma única vez com o meu colega para discutir ideias ou trabalhar no projeto e ele não mostrou interesse em perguntar o que eu tinha feito ou o que ele poderia fazer.
Apesar de não achar correto falar das pessoas, não acho de todo justo eu ter tido todo o trabalho e o meu colega ser avaliado de forma igual a mim. Falei com o Miguel e expliquei-lhe que não

gostei da atitude dele. Por isso, além de explicar ao professor, gostava também de deixar a situação explicita neste relatório.

Conclusão

Concluindo, este projeto foi bastante complexo. Além do uso de bibliotecas desconhecidas, os conceitos utilizados foram bastante difíceis de compreender. Porém foi bastante interessante e aprendi conceitos importantes para qualquer área da informática.
Assim, apesar de bastante difícil, este projeto foi bastante essencial para adquirir conhecimentos importantes para a vida de um informático.