UNIVERSIDADE D COIMBRA FACULDADE DE CIÊNCIAS E TECNOLOGIA

Sistemas Operativos
Relatório do Trabalho Prático
Aeroporto

Miguel Rabuge 2018293728
Pedro Rodrigues 2018283166

1. Introdução

Os sistemas operativos disponibilizam aos seus utilizadores um conjunto de funcionalidades que lhes permitem interagir mais facilmente com a máquina, construindo assim uma abstração dos recursos da mesma e facilitando assim a tarefa da sua programação.
Na implementação do trabalho prático proposto, tivemos a oportunidade de explorar diversos mecanismos de controlo comunicação e sincronização utilizados em sistemas operativos modernos (como o Linux). Todos estes mecanismos apareceram associados às funcionalidades de multiprocessamento e *multithreading* que necessitamos de utilizar para cumprir o objetivo de simular, através de um programa escrito em linguagem C, o funcionamento de um aeroporto. Para o implementar, necessitámos também de elaborar um algoritmo de escalonamento de voos que nos permitisse gerir o seu correto fluxo e o cumprimento dos horários previstos para estes.

2. Implementação

Na implementação do trabalho, utilizámos diversas funcionalidades básicas e complexas disponibilizadas pelas livrarias standard da linguagem C, nomeadamente:

- mecanismos de criação de *threads*, utilizando a implementação do POSIX disponibilizadas através da sua API, a livraria <pthread.h>; mecanismos de criação de processos utilizando a *system call fork*()
- os sistemas de comunicação inter-processo (IPC - *Inter Process Communication*) do Unix System V, destacando a "*Shared memory*" e as "*Message Queues*" disponibilizadas através das *APIs,* as livrarias <sys/shm.h>, <sys/ipc.h>, <sys/msg.h>
- semáforos (*named*), *mutexes* e outros mecanismos de sincronização do *posix*, como variáveis de condição, utilizando a *API* do *posix* presente nos ficheiros de headers <semaphore.h>, <pthread.h>
- mecanismos de comunicação através de *FIFO Named Pipes*, através do uso de funcionalidades das livrarias <unistd.h>, <sys/stat.h>, <sys/types.h>, entre outras
- mecanismos de controlo e de tratamento de sinais assíncronos que afetam o fluxo de execução do programa.

Também utilizámos outras funcionalidades disponibilizadas através da linguagem e de outras *APIs* que nos permitiram trabalhar o I/O, com destaque para a escrita de informação para um ficheiro de log e para a leitura de configurações através do ficheiro configs.txt. Todo o tratamento de erros, *parsing* de comandos e do fluxo do correto do programa foi implementado por forma a proteger e a prevenir resultados inesperados do programa, que resultem num terminar do programa com má limpeza de recursos alocados por este em memória ou algum dano eventual no sistema operativo.

1) Compilação / Execução do programa

Para compilar os ficheiros de código do programa, foi construído um *makefile*. Para o utilizar, basta apenas escrever **make** . Caso seja necessário remover os ficheiros objeto criados após a execução do comando, basta correr o comando **make clean**.
Desenvolvemos um pequeno *shell script* que permite a inserção de dados no *named pipe* "*input_pipe*", criado aquando a execução do programa. Este *shell script* recebe um ficheiro de comandos contendo um comando por linha e carrega estes comandos no *pipe*. A sintaxe da sua utilização é a seguinte: **./load_commands.sh <ficheiro de comandos>[.txt]**

2) Notas acerca da implementação

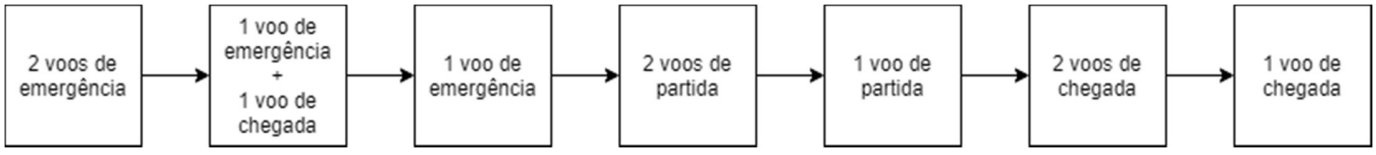
- A *flag* usada na criação do *named pipe* "*input_pipe*" que escolhemos utilizar foi a *flag O_RDWR,* porque, como sabemos, a utilização da *flag O_NONBLOCK* pode levar a situações de espera ativa, que pretendemos evitar, e a utilização da *flag O_RDONLY* levava a uma leitura incorreta dos comandos do *pipe*, visto que o comportamento era bloqueante até o primeiro comando do *pipe* ser lido; após isso, tinha um comportamento não bloqueante, colocando-nos no mesmo caso da utilização do *O_NONBLOCK* e levando-nos a uma situação de espera ativa. Utilizando a *flag O_RDWR,* o *pipe* tem o comportamento desejado, sendo sempre bloqueante após a leitura de um comando inserido pelo utilizador.

- No projeto, ficou bem claro que se existirem *threads* no sistema a correr é necessário que aquando a receção de um *signal* do tipo *SIGINT* o programa termine, mas que espere que as *threads* de voo que estão a correr terminem a sua execução, sendo a sua memória limpa apenas no fim de estas terminarem.
  Também é referido no enunciado que o limite de voos presente no ficheiro das configs apenas diz respeito aos voos ativos no sistema, e não ao total de voos que o programa pode suportar durante a execução. Para conseguir resolver o problema de conseguir eficientemente gerir a limpeza dos recursos após o terminus das *threads*, utilizamos a função *pthread_detach()*. Numa situação normal, temos de guardar sempre os identificadores das *threads* para poder garantir que estas eram joined no fim. O problema reside no facto de que o *pthread_join()* das *threads* teria de ser feito no fim do programa ou garantido por uma outra *thread* que iria verificar se alguma das *threads* de voo já porventura teria terminado, tentando dar *join* da mesma. Esta solução é pouco eficiente, visto que teríamos de percorrer o *array* de identificadores várias vezes ao longo do programa e estaríamos a bloquear o processo de limpeza quando nesse *array* existisse uma *thread* que ainda não tivesse terminado e que demorasse mais tempo que às a seguir a ela.
  A nossa solução parte por deixar as *threads* de voo *detached* e quando ocorrer o terminar normal de cada uma, a memória é limpa e o id dessa *thread* no *array* é posto de novo a 0 (*STATE_FREE*) no *array*. À medida que cada uma vai terminando, envia um sinal a uma variável de condição; caso o utilizador tenha feito o *SIGINT* irá receber esses sinais condicionais e ficar em espera condicional até todos os voos ativos terem terminado, garantindo assim que todas as *threads* terminam e não ficando o programa em espera ativa até todas o fazerem. Temos, portanto, uma variável de condição associada ao número de voos ativos no programa e o *array* de identificadores das *threads* como tínhamos antes, se quiséssemos seguir a outra forma de atacar o problema.

- O Algoritmo de escalonamento que utilizamos para garantir o correto funcionamento do aeroporto permitindo que os voos decorram segundo as condições impostas no enunciado do projeto e das configurações do ficheiro configs.txt é baseado no seguinte:

  1. Existem 3 filas de voos - uma fila de voos de partida, uma fila de voos de chegada, e uma terceira fila de emergência. Todas estas a filas são implementadas recorrendo a listas ligadas, garantido que a memória é economizada de acordo com as necessidades do aeroporto e mantendo a rapidez no acesso a cada um dos seus elementos.
     As filas de partidas são mantidas ordenadas pelo seu tempo de *takeoff* e as filas que representam voos de chegada são mantidas em ordem de acordo com o seu ETA (*Estimated Time of Arrival*) e Fuel; a fila de emergência contém voos de chegada e é mantida pelo ETA e Fuel de cada voo.
  2. O algoritmo de escalonamento começa então por decrementar o Fuel de todos os voos de chegada (e emergência) existentes enquanto verifica se existe algum voo de chegada na fila de voos de chegada que apresente um nível de combustível crítico definido no enunciado do trabalho como sendo (4 + ETA + tempo de aterragem) e se algum voo de emergência ficou com o Fuel a zero.
  3. Caso sejam encontrados voos com esta restrição eles são retirados da fila de voos de chegada e colocados na fila de voos de emergência sendo estes sempre os primeiros a levantar voo quando a pista ficar disponível para tal.
  4. Quando a pista fica livre para aterrarem voos o algoritmo verifica se existe algum voo prioritário na fila de emergência e tenta retirar os dois primeiros voos desta. Caso retire os dois primeiros não há mais pistas para receber aviões e mais partidas ou chegadas terão de esperar que a pista fique livre. Caso seja apenas retirado um voo nesse estado de emergência então o algoritmo encarrega-se de ir ao topo da fila de voos de chegada e retirar um voo dessa fila. Caso não exista nenhum neste estado então de emergência os voos são retirados da fila de voos de partida e são colocados a voar. Na eventualidade de não haverem voos de emergência nem voos de partida, então o algoritmo procura na queue dos voos de chegada.

  Abaixo segue uma ilustração do escalonamento, por ordem de prioridade do algoritmo, da esquerda para a direita.



3. Conclusão e apreciações finais

Ao longo do trabalho, fomos encontrando algumas dificuldades, que tentámos solucionar da melhor forma para garantir que o simulador teria o melhor desempenho e ao mesmo tempo evitar situações de gasto de memória e/ou espera ativa. Em termos de horas despendidas no trabalho, uma estimativa que será bastante por baixo ronda as 70 horas para o Miguel Rabuge e cerca de 65 para o Pedro Rodrigues em termos de trabalho efetivo e retirando toda a parte de elaboração do relatório e *refractor* e de comentário do código, que nos levou bastante tempo. Deixamos em anexo um esquemático do trabalho representativo do funcionamento do programa das estruturas implementadas e de suas interações.