Universidade Fernando Pessoa
Sistemas Operativos
Trabalho Prático – Partes 1 e 2

Sistemas Operativos

Pedro Sobral
pmsobral@ufp.edu.pt

Bruno Gomes
bagomes@ufp.edu.pt

João Viana
jviana@ufp.edu.pt

Março de 2022

Universidade Fernando Pessoa

Faculdade de Ciências e Tecnologias

Objectivo:

Criar um programa capaz de calcular a ocupação de uma sequência de salas de espera de um serviço de urgência, usando timestamps de entrada e saída nas salas.

1. Definição do problema

Num serviço de urgência são registados os diversos timestamps do percurso do doente. Para o adequado dimensionamento e afetação de recursos i.e. número de médicos, enfermeiros e pessoal auxiliar, torna-se necessário calcular a ocupação de cada uma das zonas do serviço.
Abaixo está uma lista de diferentes momentos de registo de tempos (i.e. Admissão, Inicio triagem, Fim triagem, Inicio médico, Alta médica) e **zonas** do serviço (i.e. **Espera Triagem, Triagem, Sala de Espera**, **Consulta**).

- Timestamp Admissão
    - **Espera Triagem**
- Timestamp Inicio triagem
    - **Triagem**
- Timestamp Fim triagem
    - **Sala de Espera**
- Timestamp Inicio médico
    - **Consulta**
- Timestamp Fim médico i.e. Alta médica

A cada admissão é necessário saber quantas pessoas estão numa dada zona, i.e. têm registo de entrada anterior e saída posterior.
Segue-se (tabela 1) o exemplo em que se descreve o cálculo para a ocupação da **Sala de Espera** (i.e. tempo decorrido entre o fim da triagem e o início médico) para o episódio de urgência 7 (a vermelho). Existe o registo de 6 episódios anteriores, no entanto 2 deles (a

verde) já saíram da zona “sala de espera”, restando apenas 4 ainda nessa zona, sendo essa a ocupação aquando da entrada do episódio 7.

	admissao	inicio_triagem	fim_triagem	inicio_medico	fim_medico
1	1425443342	1425443624	1425443627	1425444728	1425446432
2	1425443410	1425443655	1425443661	1425444677	1425445295
3	1425443419	1425443731	1425443821	1425445300	1425449323
4	1425443602	1425443871	1425443958	1425445198	1425456695
5	1425444113	1425444422	1425444454	1425445921	1425447385
6	1425444148	1425444480	1425444737	1425445275	1425453223
7	1425444514	1425444766	1425444934	1425447787	1425459188
8	1425446050	1425446170	1425446393	1425448000	1425448857
9	1425446878	1425447010	1425447188	1425448140	1425460523
10	1425446974	1425447204	1425447403	1425448786	1425449273
11	1425446994	1425447421	1425447592	1425451032	1425451673
12	1425447046	1425448428	1425448432	1425449260	1425449264

Tabela 1 - Excerto do dataset de input. Tempos em UNIX timestamp.

A mesma abordagem deverá ser seguida para os cálculos de ocupação das restantes zonas / salas (Espera triagem, triagem e consulta).

Para esta primeira fase do projeto, o processo pai deverá carregar para memória todos os timestamps presentes no dataset fornecido.

2. Requisitos

Para a 1º fase de submissão serão considerados os seguintes requisitos:

A. (5%) Receber por parâmetro o número (N) de processos filho a criar, e os nomes dos ficheiros de input / output de dados.
B. (25%) Lançar N processos filho, cada um deles responsável por calcular a ocupação dos serviços/salas partindo de uma dada entrada/data/timestamp inicial. Cada um dos processos filho fica responsável por gravar para ficheiro os valores

de ocupação obtidos. A escrita de novas entradas de ocupação deverá respeitar o seguinte formato:

pid$id,timestamp,sala#ocupação (separador = '\n')

Para o exemplo apresentado:

- id: 7;
- timestamp: 1425444934;
- sala: sala de espera;
- ocupação: 4.

Deverá ser implementada a lógica de balanceamento de carga entre cada um dos processos filho, garantindo que:

a. Não existe *overlapping* entre o trabalho realizado por diferentes processos. Isto é, os valores de ocupação num dado timestamp são calculados por um, e só um, processo filho;
b. A carga computacional exigida a cada um dos processos filho deverá ser tão equilibrada quanto possível.

C. (40%) Esta etapa implica que o programa tire partido da comunicação entre processos com recurso a *pipes*. Ao invés de imprimir diretamente o resultado para ficheiro, cada um dos processos filho deverá retornar ao processo pai (via pipe e utilizando as funções *readn*[2] e *writen*[2]) os níveis de ocupação resultantes. Do ponto de vista do processo pai, duas versões de implementação distintas devem ser consideradas:

a. Encaminhar (sem mecanismos de análise / *parsing*) para ficheiro toda a informação, relativa às taxas de ocupação, recebida via pipe. A escrita para ficheiro deverá manter o protocolo apresentado em B;
b. Lançar M processos filho (M = número de anos no dataset fornecido). O processo pai deverá fazer *parsing* da informação recebida via pipe por ano. Sempre que uma nova trama de ocupação é entregue, o processo pai ficará responsável por encaminhar a mesma ao filho correspondente (M pipes para M filhos). Assim que os N filhos terminem a execução, o processo pai deverá sinalizar os M filhos com um *SIGURS1*[1], ficando estes encarregues de executar (familia de chamadas execl / execv) um programa

fornecido para a geração de gráficos de ocupação. A figura 1 representa o fluxo de dados durante a execução do programa.

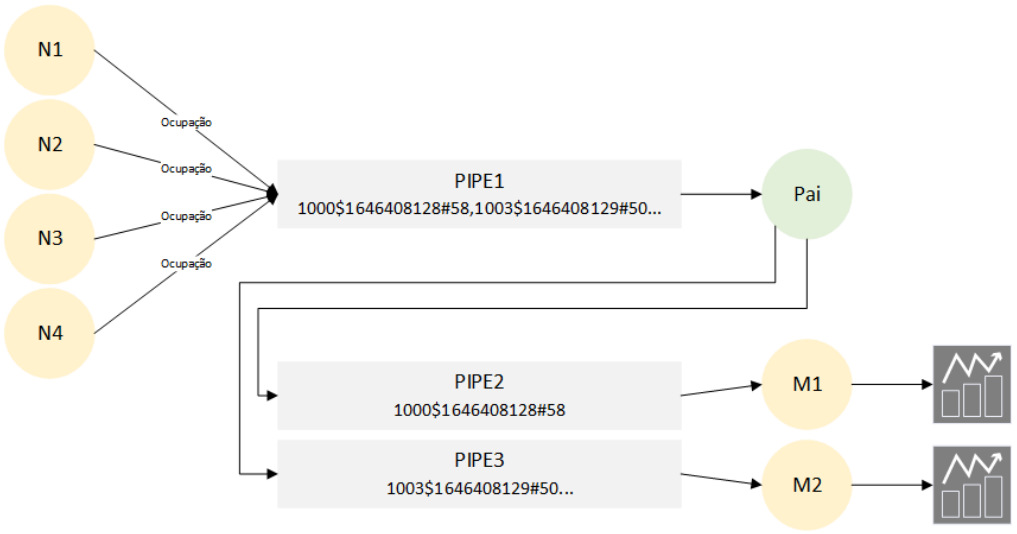

Figura 1 - Fluxo de dados

D. (30%) Esta etapa implica que o programa suporte a comunicação entre processos com recurso a *Unix Domin Sockets.* Cada filho deve estabelecer conexão com o server (pai). O pai deve atender as conexões e, à semelhança do ponto C, armazenar as taxas de ocupação em ficheiro.

Para a 2º fase de submissão serão considerados os seguintes requisitos:

A. (20%) A tarefa principal (main thread) deverá criar N worker threads, cada uma delas responsável por escrever para ficheiro de texto os valores de ocupação obtidos para cada uma das salas em todos os diferentes tempos de referência (**tempos de admissão**). Tirando partido da execução em memória partilhada, as worker threads deverão atualizar uma **variável contadora** de tempos de admissão. Sempre que a ocupação de todas as salas, dado um tempo de referência, é concluída, a worker thread deverá atualizar a variável. Periodicamente (e. g. 1x por segundo) a main thread deverá imprimir para o terminal o total de ocupações já calculado. Por exemplo, para o ID 2, timestamp "1425443410", deverá ser calculada a ocupação das salas "Espera Triagem, Triagem, Sala de Espera, Consulta". Assim que o cálculo esteja terminado a thread deverá incrementar a **variável contadora**.

B. (35%) A main thread deverá lançar P threads produtoras e N consumidoras (Figura 2). Cada uma das P threads produtoras gerará pares sala / ocupação para os timestamp de admissão presentes no dataset. Sempre que calculada a ocupação para uma nova sala, um novo item (produto) deverá ser colocado numa estrutura de dados partilhada com o consumidor. O processo de consumo consiste na remoção do item da estrutura partilhada e escrita do mesmo para ficheiro.

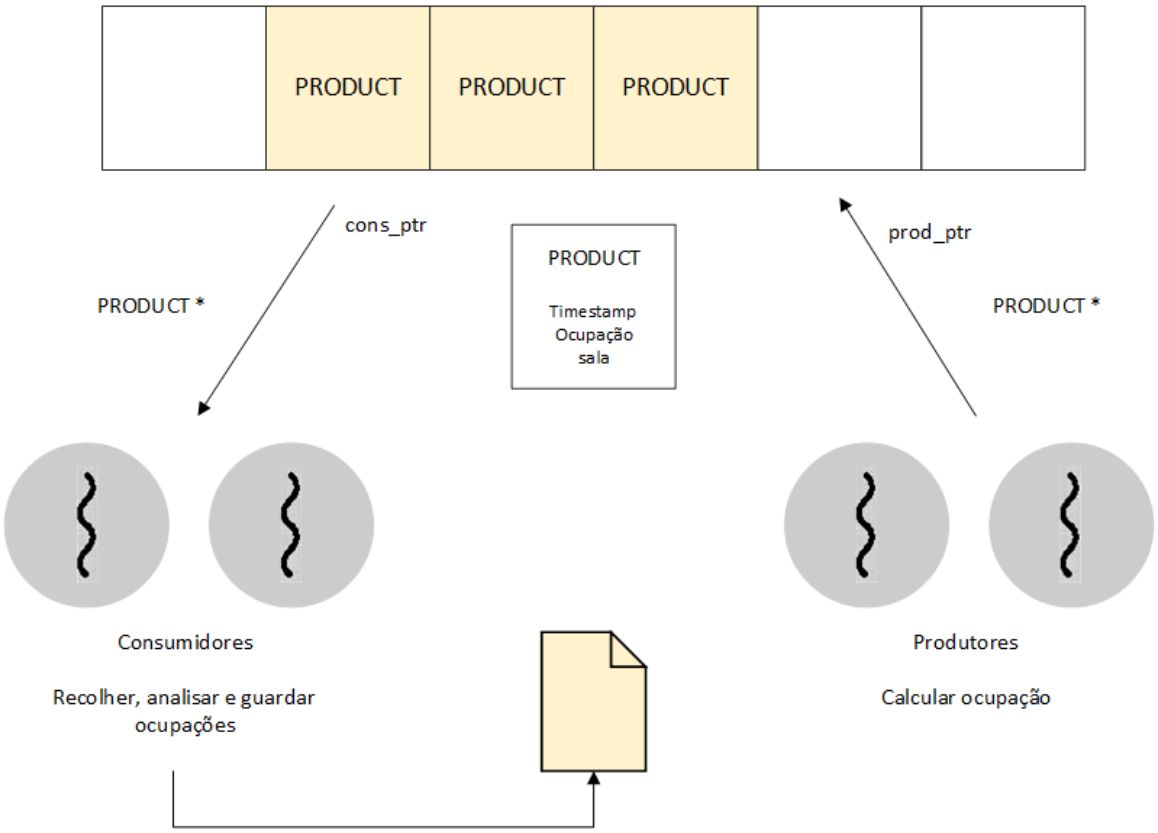

Figura 2 - Produtor-consumidor aplicado ao cálculo concorrente de ocupação das salas

C. (15%) Ao invés de escrever os resultados para um único ficheiro de forma incondicional, as C threads consumidoras ficarão responsáveis por dividir o output por ano, gerando N ficheiros de output (N = número de anos no dataset).

D. (30%) Pretende-se que o requisito B seja adaptado para um paradigma de computação MapReduce (MR) (figura 3). Nesta abordagem simplificada ao MR, Após carregar o ficheiro para memória, M threads Map deverão, para cada entrada no dataset, executar uma função de hash para determinar o thread R correspondente ao tempo de admissão. A troca de informação entre Map e Reduce

threads ocorre via R buffers partilhados com as threads de M (i. e. 1 buffer por Reduce thread). Com base no tempo de admissão recebido, cada uma das R threads fica responsável por determinar qual a ocupação de cada uma das salas no momento de admissão recebido.

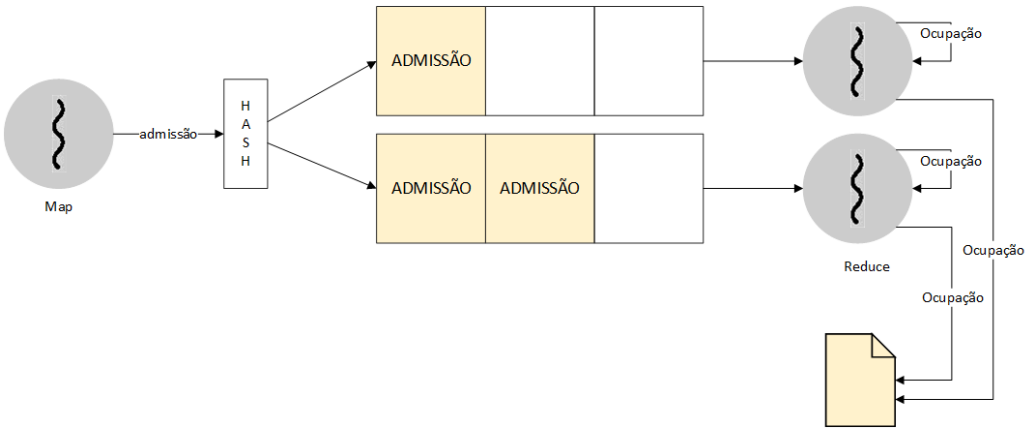

Figura 3 - MapReduce aplicado ao cálculo concorrente de ocupação das salas

3. Notas

Em todo o projeto o aluno deverá utilizar exclusivamente as chamadas ao sistema POSIX nativas (*open, read, write*,...) e não as funções das bibliotecas C (*fopen, fread, fwrite*,...). Em todas as chamadas ao sistema devem ser testadas as eventuais condições de erro recorrendo à função *perror()*.

Este trabalho será realizado individualmente ou em grupos de dois alunos. O trabalho (README e código fonte) tem de ser submetido até à data indicada no sistema de elearning (trabalhos) e será apresentado e defendido de "viva voz" em data a designar pelo docente. Para o cálculo da nota final, uma ponderação de 50% será atribuída a cada uma das fases de submissão.

4. Bibliografia

[1] *Advanced Programming in the UNIX ® Environment* - Signal - 10.14 sigaction Function
[2] *Advanced Programming in the UNIX ® Environment* - Advanced I/O - 14.7 readn and writen Functions
[3] *Advanced Programming in the UNIX ® Environment* - Advanced IPC - 17.2 UNIX Domain Sockets