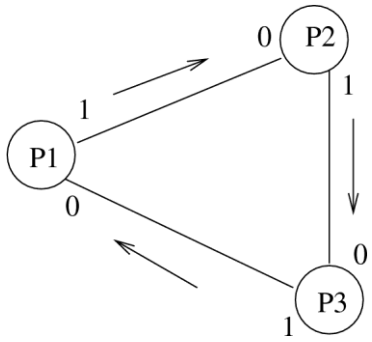
Fundamentos de Sistemas de Operação – Questões-tipo

<u>Questão 1</u>. Considere a seguinte configuração de processos concorrentes, no sistema Unix, onde se indicam os números dos canais standard (0 e 1) de cada processo e as suas interligações, correspondentes aos arcos da figura, realizadas através de *pipes* Unix, com o seguinte significado:

qualquer caracter escrito no canal 1 de P1 possa ser obtido através do canal 0 de P2,

qualquer caracter escrito no canal 1 de P2 possa ser obtido através do canal 0 de P3,

qualquer caracter escrito no canal 1 de P3 possa ser obtido através do canal 0 de P1,



a) Escreva um programa em C, baseado em chamadas ao sistema Unix, executado por um processo p0 inicial que crie os três processos p1, p2 e p3, e que desencadeie a realização das interligações indicadas na figura, através de *pipes*. Após as inicializações que sejam necessárias, os processos p1, p2 e p3 devem executar os programas contidos em ficheiros executáveis, cujos nomes são indicados pelas variáveis, respectivamente, P1exe, P2exe e P3exe.

b) Modifique o programa da alínea **a)**, de tal modo que os processos p1, p2, p3, criados pelo processo p0 inicial, sejam forçados a aguardar (no estado Bloqueado) até que o processo p0 lhes envie uma indicação para iniciarem a execução dos seus programas (P1exe, P2exe ou P3exe) . Utilize *pipes* pare este efeito.

c) Modifique o programa da alínea **a)** de modo a que o processo p0 inicial fique a aguardar a terminação dos processos p1, p2 e p3, e depois termine também.

d) Admita que, <u>durante a execução</u> do programa da alínea **a)**. se pretendia que, através de uma indicação enviada pelo processo p0 a cada um dos processos p1, p2, p3, o sentido das interligações indicadas na figura fosse invertido, de tal modo que o processo p1 passasse a enviar para o processo p3, o processo p3 a enviar para o processo p2 e o processo p2 a enviar para o processo p1.
Sem apresentar o programa, explique que acções seriam necessárias para esse efeito.

<u>Questão 2.</u> Considere um sistema de conversação (*chat*) que permite a troca de mensagens entre 3 processos (P1, P2,P3) que se assume que foram previamente criados. As mensagens (que na realidade são *strings*) são dirigidas a um servidor centralizado (S), também previamente criado, que tem como tarefa fazer a disseminação. Esta disseminação depende do tipo da mensagem: mensagens públicas devem ser entregues a todos os processos, excepto o que enviou; mensagens privadas devem entregues apenas ao destinatário. Cada processo Pi (i:1,2,3) **tem apenas acesso** às estruturas que representam a fila filai que o liga ao servidor. As filas são representadas por regiões em memória partilhada.

a) Pretende-se que os processos clientes P1, P2 e P3 tenham à sua disposição as seguintes operações:

- *void enviaMensagemPrivada(char *mensagem, int proc)* – envia uma mensagem privada para *proc*, onde *proc* é um número de 1 a 3 que identifica univocamente um processo. A função não é bloqueante, isto é, não espera que a mensagem seja entregue. No entanto, deve garantir que a mensagem é colocada na fila relevante.
- *char* recebeMensagem()* - recebe uma mensagem, com semântica bloqueante. Devolve um apontador para a mensagem.

A comunicação entre os clientes e o servidor assenta em regiões de memória partilhada, que se admitem previamente criadas e nas quais estão representadas as seguintes filas, um par de filas dedicado à comunicação entre cada processo e o servidor:

- Fila filaA_i fila em memória partilhada para as mensagens a receber pelo processo cliente Pi.
- Fila filaB_i fila em memória partilhada para as mensagens enviadas pelo processo cliente Pi.

Cada processo (P1,P2,P3) **tem apenas acesso** ao par de filas filaA_i/filaB_i que partilha com o servidor. Admita que cada fila apenas tem espaço limitado para conter um máximo de N mensagens, sendo N uma constante, e que todas as mensagens têm um tamanho fixo. Existem as seguintes funções disponíveis:

- *void P(int sem)* e *void V(int sem),* semáforos com a semântica definida por Dijkstra.
- *int colocaNaFila(Fila f, void* m)* – coloca uma mensagem m na fila f. Retorna 0 caso a mensagem seja colocada com sucesso ou -1 em caso contrário. Esta operação não controla os acessos concorrentes.
- *void* retiraDaFila(Fila f)* – Retira uma mensagem da fila f. Retorna NULL caso a fila esteja vazia. Esta operação não controla os acessos concorrentes.

Indique os semáforos de que necessita para sincronizar a comunicação entre um cliente e o servidor. Para cada semáforo indique o seu propósito e o seu valor inicial.

b) Nas mesmas condições da alínea a), implemente as funções *enviaMensagemPrivada* e recebeMensagem, definidas na alínea a), recorrendo às filas em memória partilhada dadas e a semáforos. Lembre-se que o envio é não bloqueante e que a recepção é bloqueante. Nesta alínea, não pode usar filas de mensagens Unix nem *pipes*.

Questão 3. Considere, no Unix, que um processo inicial P0 criou a seguinte configuração de dois processos filhos, P1 e P2, interligados através de um pipe e, depois disso, o processo P0 terminou.

Ficheiro F1 > P1 ----> Pipe pp1 ---> P2 > Ficheiro F2

em que se têm as seguintes ligações:
canal standard out de P1 mantém-se ligado ao lado de escrita de Pipe pp1
canal standard in de P1 mantém-se aberto para ler de F1
canal standard in de P2 ligado ao lado de leitura de Pipe pp1
canal standard out de P2 aberto para escrever em F2

Admita que os processos **P1** e **P2**, após estabelecidas as ligações indicadas, executam os programas contidos respectivamente nos ficheiros executáveis "**/tmp/fP1**" e "**/tmp/fP2**".

Pretende-se garantir que, em caso de o processo P2 terminar inesperadamente, através de acções equivalentes à chamada ao SO exit(), o processo P1 seja capaz de reagir, reconfigurando o sistema para o seguinte esquema:

Ficheiro F1 > P1 ----> Pipe pp2 ---> P3 > Ficheiro F2

em que P3 é um novo processo (que executa o ficheiro "**/tmp/fP2**") e pp2 é um novo pipe, tal que:
canal standard out de P1 ligado ao lado de escrita de Pipe pp2
canal standard in de P3 ligado ao lado de leitura de Pipe pp2
canal standard in de P1 aberto para ler de F1
canal standard out de P3 aberto para escrever em F2, a partir do ponto em que P2 estava a escrever em F2, quando terminou.

Usando chamadas ao sistema Unix, **apresente apenas o código, em C, que P1 deve executar** para garantir a reconfiguração do sistema nas condições apresentadas.

Questão 4. Explique as diferenças, as vantagens e os inconvenientes dos seguintes modelos de comunicação disponíveis no sistema Unix. Nessa comparação, oonsidere as duas dimensões: (i) o modelo oferecido ao programador; (ii) os mecanismos que suportam a implementação.

a) comunicação por memória partilhada entre processos (processos criados por *fork*)
b) comunicação por filas de mensagens

Questão 5. Considere, num ambiente de multiprogramação Unix, um conjunto de N+1 processos concorrentes, cujos identifcadores (*process id*) são denotados por P0, P1, P2, ... PN. Admita que todos o processos já se encontram criados.

Processo P0 - mestre:

este processo está continuamente a gerar tarefas e a distribuí-las pelos N processos (trabalhadores) P1, ... PN, para estes as executarem;

para que o processo P0 possa distribuir as tarefas pelos trabalhadores, considera-se uma fila (F) de tarefas, a qual é globalmente acessível a todos os processos, de forma concorrente e tem uma capacidade limitada a um certo número de tarefas, dependente da implementação;

cada tarefa posta na fila é descrita por um descritor. Não precisa de se preocupar com a estrutura interna do descritor que descreve cada tarefa, excepto que cada descritor tem um tamanho fixo de *bytes*, definido por uma constante chamada SIZE.;

O processo P0 coloca cada tarefa na fila F, através da seguinte operação:

colocar (tarefa *T); – O argumento T aponta o descritor da tarefa a colocar. No caso de a fila F estar cheia, o processo P0 tem de aguardar, bloqueado, até poder completar esta operação. Esta operação deve implementar toda a sincronização necessária para que o acesso à fila F seja correcto e coerente.

Processos P1, ... PN – trabalhadores:

cada uma das tarefas deve ser executada por um qualquer (mas apenas por um) dos processos P1,..., PN.

cada um dos processos P1, ... PN efectua um ciclo de dois passos, que repete indefinidamente,:

1- aguardar até obter uma tarefa da fila F
2- executar a tarefa obtida (esta acção assume-se já implementada e não nos interessa aqui);

Cada processo P1,,.., PN obtém cada tarefa da fila F, através da seguinte operação:

obter (tarefa *T); – o processo aguarda, bloqueado, até que possa obter (e remover) uma tarefa da fila. O argumento T aponta o descritor da tarefa obtida e, então, o processo prossegue a execução do seu ciclo, conforme foi acima explicado. Esta operação deve implementar todas a sincronização necessária para que o acesso à fila F seja correcto e coerente.

Para cada uma das alíneas seguintes, pede-se que descreva em C (pseudo-código), a implementação da fila global F e das funções *colocar () e obter()*.

a) Nesta alínea, para representar e controlar o acesso à fila global F, apenas pode utilizar filas de mensagens. Admite-se que as filas necessárias já se encontram criadas.

Apresente em C (pseudo-código), a implementação das duas funções *colocar* e *obter*.

Deve usar explicitamente as funções definidas pelas chamadas ao Unix para operar sobre *message queues*.

Indique todas as declarações de variáveis e funções auxiliares de que necessite. Nesta alínea, não pode utilizar memória partilhada nem semáforos.

b) Nesta alínea, para representar e controlar o acesso à fila global F, apenas pode utilizar comunicação por memória partilhada entre processos (shared memory Unix System V) e sincronização por semáforos, segundo as operações definidas por Dijkstra (P e V). Admite-se que as regiões de memória partilhada já se encontram criadas (assim, não precisa de usar as chamadas ao sistema shmget/shmatt, basta declarar as estruturas necessárias em pseudo-código).

Admite-se que a fila F, representada em memória partilhada, tem uma capacidade máxima limitada a N descritores de tarefas.

Admite-se que as operações auxiliares para de acesso à estrutura de dados que representa a fila F em memória, já estão implementadas (mas note que estas duas funções não tratam da sincronização dos acessos concorrentes):

*inserir(tarefa *T)* - insere um elemento - só pode ser invocada se a fila não estiver cheia,

*remover(tarefa *T)* - remover um elemento - só pode ser invocada se a fila não estiver vazia,

Apresente em C (pseudo-código), a implementação das duas funções *colocar* e *obter*

Indique todas as declarações de variáveis que assume partilhadas em memória, os semáforos necessários e os seus valores iniciais.

Questão 6. Considere, no Unix, um grupo de N processos concorrentes, cujos identifcadores são denotados por P1, P2, ... PN. Pretende-se implementar, usando chamadas ao sistema Unix, um mecanismo de difusão de mensagens, baseado na seguinte função:

difundir_mensagem(char * Msg, int Wait)

tal que o processo invocador envia uma mensagem a todos os outros processos. O argumento *Msg* aponta uma cadeia de caracteres correspondente ao conteúdo da mensagem a enviar. O argumento *Wait* indica, caso seja 1, que o processo invocador deve aguardar que todos os processos receptores confirmem a recepção da mensagem; caso o argumento *Wait* seja 0, a operação tem uma semântica assíncrona ou não bloqueante.

Cada processo pode receber mensagens através da invocação da função

receber_mensagem (char *Msg)

em que o argumento aponta a mensagem obtida. Esta operação é bloqueante.

a) Nesta alínea, pretende-se implementar as operações acima indicadas, nas seguintes condições:

-- para a comunicação apenas pode utilizar filas de mensagens e admite-se que as filas necessárias já se encontram criadas. Não pode utilizar memória partilhada ou semáforos.

-- deve indicar o pseudo-código, em C, das duas funções ***difundir_mensagem*** e ***receber_mensagem***, bem como todas as declarações de variáveis relevantes e todas as filas de que necessite,

b) Nesta alínea, preende-se implementar as operações acima indicadas, nas seguintes condições:

-- apenas se pode utilizar comunicação por memória partilhada (shared memory Unix System V) e sincronização por semáforos, segundo a definição de Dijktstra, Admite-se que as regiões de memória partilhada necessárias já se encontram criadas e os processos já se ligaram a elas. Admite-se que os semáforos necessários já se encontram criados.

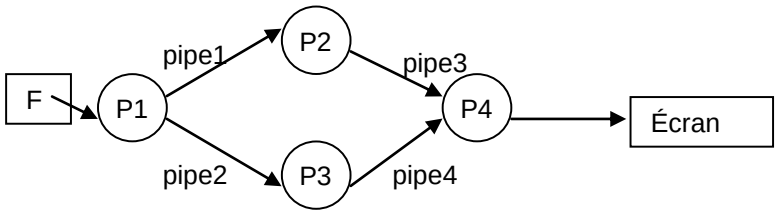
-- deve indicar o pseudo-código, em C, das duas funções ***difundir_mensagem*** e ***receber_mensagem***, bem como todas as declarações de variáveis relevantes, incluindo as variáveis e as filas partilhadas em memória e os semáforos necessários, indicando os seus valores iniciais.

Questão 7.

a) Usando chamadas ao sistema Unix, escreva, em C, a funcao transfer_process() que delega a continuação da execução do programa associado ao processo invocador, num novo processo e destrói o processo original. O novo processo deve ter exactamente o mesmo contexto e ambiente do processo original e deve continuar a execução no mesmo ponto do programa em que o processo original se encontrava.

b) Nos mesmos pressupostos da alínea a), apresente o programa modificado da função transfer_process() com a única alteração seguinte: uma vez criado, o novo processo deve iniciar a sua execução na 1a instrução do programa associado ao processo invocador (em vez de no mesmo ponto do programa em que o processo original se encontrava). Admita que o programa em execução pelo processo original está contido num ficheiro executável de nome "/usr/bin/f".

Questão 8. Para implementar, no Unix, uma aplicação para mostrar vídeos, usando múltiplos processadores, dividem-se as várias fases por diferentes processos, comunicando entre si por *pipes* Unix, segundo o esquema seguinte:



- O processo P1 é responsável por ir lendo, do ficheiro F, sucessivos blocos de 16 Kbytes, distribuindo depois cada bloco, alternadamente, para cada um dos processos P2 e P3.

- Os processos P2 e P3 vão processando cada um dos blocos recebidos e vão enviando os resultados, também em blocos de 16KB, para o processo P4.

- O processo P4 vai lendo, alternadamente, os blocos recebidos de P2 e P3 e vai-os afixando no écran.

a) Escreva, em C, as acções do programa que se encarregam de lançar os processos indicados e da criação dos *pipes* necessários. Admita que cada um dos processos criados, P1, P2, P3 e P4, inicia de imediato a execução de uma das funções seguintes, respectivamente, **proc1(), proc2(), proc3()** e **proc4()** (admita nesta alínea que estas funções estão já definidas e se encarregam de estabelecer as necessárias inicializações das ligações aos *pipes*).

b) Escreva, em C, o código da função **proc1()** executada por **P1**. Esta função deve ler, de um ficheiro cujo nome é indicado na variável **char *F**, sucessivos blocos de 16KB cada, e escrever os blocos lidos, alternadamente, num dos dois pipes **pipe1** e **pipe2**.

c) Admita que os processos **P2** e **P3** executam um mesmo programa, contido no ficheiro executável "**/usr/bin/decod**", responsável por processar os blocos que lhes vão chegando pelos respectivos canais de **standard input,** e por irem colocando os blocos com os resultados, nos respectivos canais de **standard output**. Escreva, em C, o código das funções **proc2()** e **proc3()**, que devem efectuar as redirecções de canais que forem necessárias, antes de iniciarem a execução do programa "**decod**" por cada processo P2 e P3.

Questão 9. Considere o mesmo problema da **Questão 8** mas onde agora a comunicação dos processos P2 e P3 com o processo P4 se realiza **exclusivamente através de uma única fila de mensagens FM**. Reescreva os programas das funções **proc2()** e **proc3()** e apresente o código da função **proc4()**, garantindo a leitura alternada de mensagens provenientes de P2 e P3, usando exclusivamente chamadas ao sistema Unix sobre a fila de mensagens FM. Admita que a fila FM já se encontra criada e que cada mensagem trocada tem exactamente o tamanho de um bloco.

Questão 10. Considere uma zona de memória partilhada por **N** processos concorrentes e com a capacidade máxima de um bloco de 1Kbytes. Um processo **P1** produz um bloco de cada vez que escreve nessa zona. Cada um dos restantes processos **P2, ... PN** deve ler o conteúdo desse bloco. Só após todos os processos (P2, ... PN) terem lido o conteúdo desse bloco, será este considerado lido e a zona de memória será considerada livre.

- O processo **P1** invoca a função **pôrBloco(buf)** que o bloqueia até que a zona de memória esteja livre e, então, copia o conteúdo de **buf** para a zona de memória partilhada.
- Cada um dos processos **P2, ... PN** invoca a função **lerBloco(buf),** que o bloqueia até que exista um novo bloco para ler. O processo obtém, depois, uma cópia do bloco lido para **buf** e aguarda, bloqueando-se até que todos os outros processos (do conjunto **P2-PN)** tenham também lido esse bloco.

Apresente o código C das funções **pôrBloco(buf)** e **lerBloco(buf),** baseando-se exclusivamente em comunicações por memória partilhada e em sincronização por semáforos, segundo a definição de Dijkstra.

Questão 11. Apresente o pseudocódigo das acções efectuadas pelas seguintes operações sobre semáforos, segundo a definição de Dijkstra, em que 'sem' indica um semáforo. Deve descrever todas as estruturas de dados auxiliares ou outras operações básicas de que necessite. Assuma que a indivisibilidade do código que apresentar está garantida:

a) Operação P(sem)
b) Operacão V(sem)

Questão 12. Considere o problema dos leitores e escritores, processos concorrentes com acesso a uma base de dados, através das operações ler(R) e escrever(R) em que R designa um valor lido ou escrito. Admita que estas operações, que processam o acesso aos registos da base de dados, já estão implementadas, mas não controlam as interferências devidas aos acesso concorrentes dos múltiplos clientes leitores e escritores.

a) Baseandose em semáforos, segundo a definição de Dijkstra, e em comunicação por memória partilhada, apresente o pseudocódigo das acções seguintes:

pedir_ler() pedir_escrever()
terminar_ler() terminar_escrever()

que devem ser invocadas respectivamente, pelos leitores e escritores, antes e depois de poderem começar a ler ou a escrever. Note que as operações pedir_ler() e pedir_escrever() devem ser bloqueantes, caso o leitor ou o escritor não possa prosseguir por a base de dados estar correntemente ocupada por outros processos.

b) Explique quais as propriedades da solução que apresentou em a), do ponto de vista da justiça (fairness) no tratamento dos pedidos dos leitores e escritores.

Questão 13. Usando chamadas ao sistema Unix, escreva, em C, o pseudocódigo de uma funcao

*redirect_IO(char *filename_in, char *filename_out)*

que redirige os canais standard de entrada e de saída do processo invocador, repectivamente, para o ficheiro de nome absoluto indicado pelo argumento filename_in e para o ficheiro de nome absoluto indicado pelo filename_out.

Questão 14. Considere que, no sistema Unix, um processo foi criando processos filhos, por diversas invocações da chamada ao SO ***fork()***. Admita que, no momento da terminação do processo pai, é sempre invocada uma função ***new_exit()*** que desencadeia a seguinte sequência de acções:

(1) invoca a função ***terminating()*** indica aos filhos a terminação do pai, baseando-se em comunicação por ***pipes***.

(2) invocação da chamada ao SO ***exit()***

Usando chamadas ao sistema Unix, **apresente**, em C, o pseudocódigo das seguintes funções:

void terminating()

int wait_parent() bloqueia o processo invocador até que o seu processo pai assinale a terminação da sua execução, devolvendo, como valor de retorno, o identificador do processo pai.

Explique todas as hipóteses e estruturas de dados que assumir na sua resolução.

Questão 15.

Considere que, no sistema Unix, se pretende realizar o esquema seguinte de comunicação por mensagens entre três processos concorrentes **P1**, **P2** e **P3**. Admita que as variáveis **P1**, **P2** e **P3** assumem, respectivamente, os valores dos identificadores daqueles processos no sistema Unix (*process identifiers*).
Os processos estão obrigados a comunicarem entre si através do seguinte esquema, em que **fM1**, **fM2**, **fM3** são **filas de mensagens**:

P1 --> fila fM1 -------->
procS -------> fila fM3 ---> P3
P2 --> fila fM2 -------->

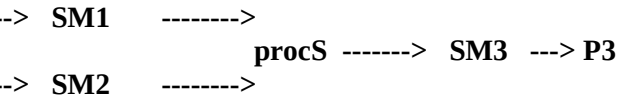
em que: **P1** só tem acesso a **fM1; P2** só tem acesso a **fM2; P3** só tem acesso a **fM3**.
O servidor auxiliar **procS** tem acesso às três filas e tem como função ir transferindo as mensagens para **fM3**, à medida que chegam a **fM1 ou** a **fM2.**
(Note: As acções de **procS** podem ser executadas concorrentemente.)

Pretende-se realizar as duas operações seguintes:
- enviar(M): invocada pelo processo **P1** ou por **P2**, desencadeia o envio de uma mensagem **M**, de modo a que possa ser recebida pelo processo **P3**. O envio é não bloqueante. A mensagem **M** deve basear-se na convenção da comunicação por filas de mensagens.

- receber(M, Pe): invocada por **P3**, recebe uma mensagem, devolvida no argumento **M.** A mensagem recebida depende do valor passado no argumento **Pe**:
-- se **Pe** == **0** – deve receber a mensagem mais antiga que tenha chegado a **fM3**.
-- se **Pe** == **P1** – deve receber a mensagem mais antiga enviada por **P1**
-- se **Pe** == **P2** - deve receber a mensagem mais antiga enviada por **P2**
Esta operação de recepção é bloqueante.

==> Programe em C, usando chamadas ao Unix, as acções das operações **enviar()** e **receber() e as acções do processo procS para transferir as mensagens.** Não necessita de programar a criação das filas nem dos processos. Nesta questão, **não pode** utilizar **pipes, memória partilhada ou semáforos.**

Questão 16.
Admita que se pretende realizar um esquema de comunicação por mensagens entre três processos **P1**, **P2** e **P3, como na Questão 2(***T2***), mas implementado sobre comunicação por memória partilhada**. Admita que, em vez de filas de mensagens, existem as seguintes áreas de memória partilhada (*shared memory Unix IPC System V*), exclusivas para a comunicação entre os processos:



SM1 - exclusiva para comunicação **entre P1 e procS** (onde P1 põe mensagens)
SM2 - exclusiva para comunicação **entre P2 e procS** (onde P2 põe mensagens)
SM3 - exclusiva para comunicação **entre procS e P3** (onde P3 obtém mensagens)

Como na Questão **2**(*T2*), o servidor auxiliar **procS** tem acesso às três áreas e tem como função ir transferindo as mensagens para **SM3**, à medida que são colocadas em **SM1** ou **SM2.** Admita mensagens de um tamanho fixo e que cada fila pode conter um máximo de MAX mensagems.

Admita que existem duas primitivas auxiliares **'inserir(fila,M)'** e **'remover(fila,M)'** para aceder às filas em memória partilhadas, mas que não estão protegidas comtra acessos concorrentes.

Baseando-se em **semáforos**, apresente o pseudo-código das acções do processo **procS** para o encaminhamento das mensagens e o pseudo-código das operações seguintes:

enviar(M): invocada por **P1** ou **P2**, desencadeia o envio de uma mensagem **M**, de modo a que possa ser recebida pelo processo **P3**. O envio é não bloqueante.
receber(M): invocada por **P3**, recebe uma mensagem, em **M**. A recepção é bloqueante.
Suponha que as áreas de memória partilhadas e os processos foram previamente criados. Nesta questão, **não pode** utilizar **filas de mensagens Unix nem pipes.**

Questão 17. Considere que, no sistema Unix, dois processos concorrentes, de identificadores (processid) denotados respectivamente por P1 e P2, deverão poder sincronizarse mutuamente, através da invocação da seguinte função:

aguardar_parceiro();

Esta operação, após ser invocada por cada um dos processos (P1 e P2), só lhes deve permitir continuarem a execução até que ambos os processos a tenham invocado, ou seja, quando um dos processos (P1 ou P2) a invocar, esse processo deve aguardar, no estado bloqueado, até que o outro processo (P2 ou P1) a invoque também. Admita que os processos P1 e P2 são criados por um mesmo processo pai, de identificador (processid) denotado por P0. Admita que o processo P1 executa um programa contido num ficheiro "/usr/user1/f1" e o processo P2 executa um programa contido num ficheiro "/usr/user1/f2". Baseando-se em pipes Unix para realizar a sincronização entre processos, descreva em C (pseudocódigo) uma implementação

da operação *aguardar_parceiro()* e também o programa que o processo P0 deve executar para incializar os processos P1 e P2, bem como quaisquer outras estruturas necessárias.

Nesta questão não pode utilizar memória partilhada, semáforos, nem filas de mensagens, nem sinais Unix.

Questão 18. Pretende-se implementar, com base em chamadas ao sistema Unix, duas operações atómicas *Send* e *Receive*, que dão acesso a uma fila global de mensagens, suportando a comunicação entre processos concorrentes. Nesta questão, considera-se uma implementação baseada em *memória partilhada e semáforos*.

Assumem-se as seguintes condições:

(i) Considera-se uma fila única (F) de mensagens, a qual é globalmente acessível, segundo uma disciplina de acesso FIFO (*first-in-first-out*), por todos os processos, de forma concorrente e tem uma capacidade limitada a um certo número de mensagens, dependente da implementação. Cada mensagem posta na fila é descrita por um descritor contendo o identificador do processo emissor e o texto da mensagem enviada. Admita que cada descritor tem um tamanho fixo de *bytes*.

(ii) Um processo coloca uma mensagem na fila F, através da seguinte operação:

Send (Mensagem *M);

– O argumento M aponta o descritor da mensagem a colocar. A operação é não bloqueante, excepto no caso de a fila F estar cheia, devendo então o processo aguardar, bloqueado, até poder completar esta operação. Esta operação deve garantir toda a sincronização necessária para que o acesso à fila F seja correcto e coerente.

(iii) Um processo obtém uma mensagem da fila F, através da seguinte operação:

Receive (Mensagem *M);

– o processo aguarda, bloqueado, até que possa obter (e remover), segundo a ordem FIFO, uma mensagem da fila F. O argumento M aponta uma estrutura contendo o descritor da mensagem. Esta operação deve garantir todas a sincronização necessária para que o acesso à fila F seja correcto e coerente.

→ Pede-se que descreva em C, a implementação das funções *Send () e Receive()*.

Admita que, para representar e controlar o acesso à fila global F, apenas pode utilizar comunicação por memória partilhada entre processos (shared memory Unix System V) e sincronização por semáforos, segundo as operações definidas por Dijkstra (P e V). Admite-se que as regiões de memória partilhada necessárias já se encontram criadas (assim, não precisa de usar as chamadas ao sistema shmget/shmatt), mas precisa de declarar as estruturas de dados necessárias. Admite-se que a fila F, representada em memória partilhada, tem uma capacidade máxima limitada a N mensagens.

Admite-se que as seguintes operações auxiliares de acesso à estrutura de dados que representa a fila F em memória, já estão implementadas (mas note que estas duas funções não tratam da sincronização dos acessos concorrentes):

*inserir(Mensagem *M)* -- insere uma mensagem - só pode ser invocada se a fila não estiver cheia

*remover(Mensagem *M)* -- remove uma mensagem - só pode ser invocada se a fila não estiver vazia

Apresente em C, a implementação das duas funções *Send* e *Receive.*

Indique todas as declarações de variáveis que assume partilhadas em memória, os semáforos necessários e os seus valores iniciais. Nesta alínea não pode utilizar filas de mensagens, *pipes* ou sinais Unix.