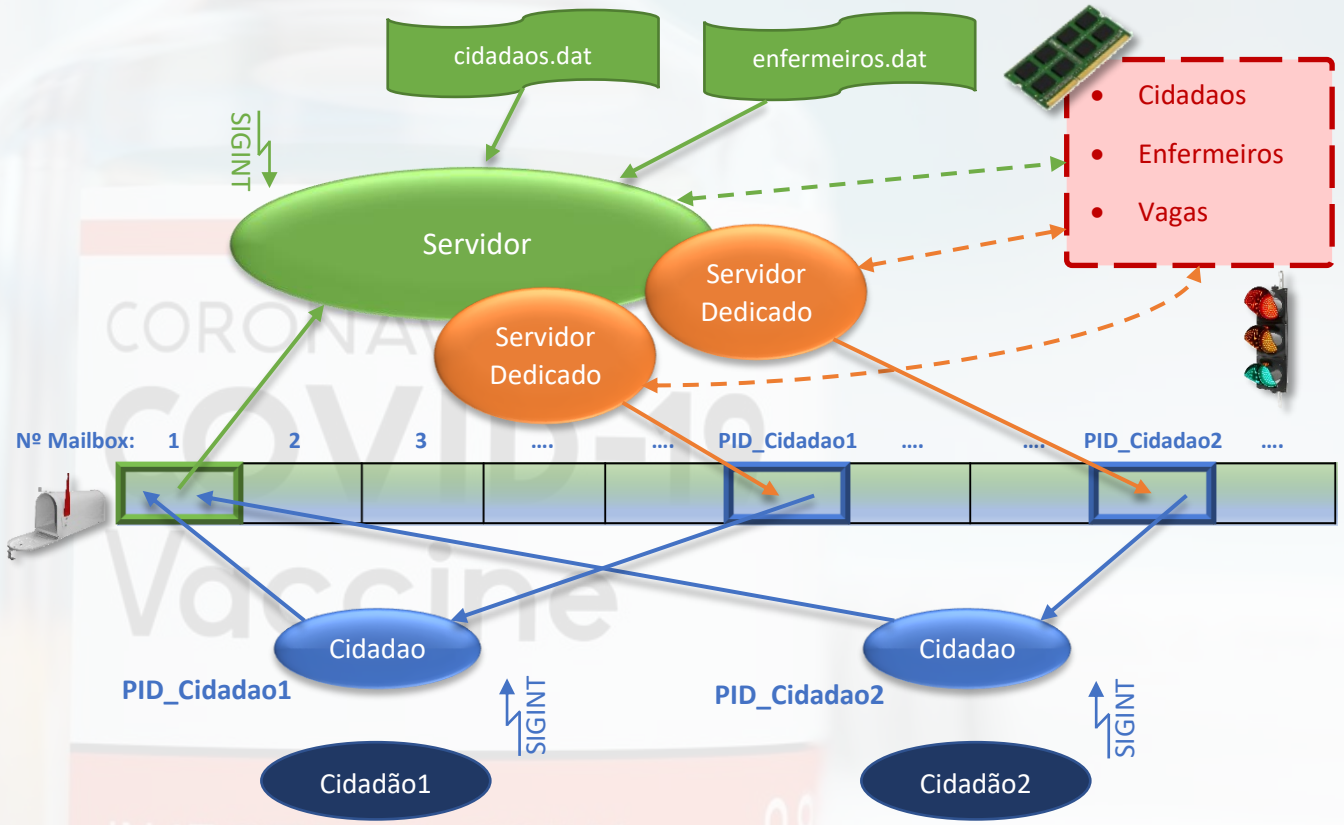
iscte TECNOLOGIAS E ARQUITETURA Departamento de Ciências e Tecnologias da Informação	Sistemas Operativos Ano Letivo 2020/2021 2º Semestre versão 2

Projeto *Covid-IUL (Parte 3)*

Nesta parte do trabalho, será implementado um modelo simplificado da Administração de vacinas dos cidadãos do sistema Covid-IUL, baseado em comunicação por IPC entre processos, utilizando a linguagem de programação C. Considere o seguinte diagrama, que apresenta uma visão geral da arquitetura pretendida:



Ideia Geral: Pretende-se, nesta fase, simular a realização da vacinação no sistema Covid-IUL. Assim, teremos dois módulos a realizar – **Cidadão** e **Servidor**.

Para distinguir os destinatários das mensagens, vamos utilizar o campo **tipo** (Address) da **mensagem**:

- Se **tipo = 1**, a mensagem é destinada ao servidor
- Se **tipo = PID_Cidadao** (obrigatoriamente maior que 1), a mensagem é destinada ao processo cujo *process ID* é igual a PID_Cidadao.

Entrega, relatório e autoavaliação

O trabalho de SO será realizado **individualmente**, logo sem recurso a grupos.

A entrega da Parte 3 do trabalho será realizada através da criação de **um** ficheiro ZIP cujo nome é o nº do aluno, e.g., **"a<nºaluno>-parte-3.zip"** (**ATENÇÃO:** não serão aceites ficheiros RAR, 7Z ou outro formato) onde estarão todos os ficheiros criados. Estes serão apenas os ficheiros de código, ou seja, na terceira parte, apenas os ficheiros de código (**.c** e **.h**). Cada um dos módulos será desenvolvido com base nos ficheiros fornecidos, e que estão na diretoria do Tigre **"/home/so/trabalho-2020-2021/parte-3"**, e deverá incluir nos comentários iniciais um "relatório" indicando a descrição do módulo e explicação do mesmo (poderá ser muito reduzida se o código tiver comentários bem descritivos). Naturalmente, deverão copiar todos estes ficheiros para a vossa área.

Para criarem o ficheiro ZIP, deverão usar, no Tigre, o comando **$ zip a<nº aluno>-parte-3.zip <ficheiros>**, por exemplo:

```
$ zip a123456-parte-3.zip *.c *.h
```

A entrega desta parte do trabalho deverá ser feita por via eletrónica, através do e-learning:

- e-learning da UC Sistemas Operativos;
- Selecionam a opção sub-menu "Conteúdo/Content";
- Selecionem o link "Trabalho Prático 2020/2021 Parte 3";
- Dentro do formulário "Visualizar Exercício de carregamento: Trabalho Prático 2020/2021 Parte 3", selecionem "Anexar Arquivo" e anexem o vosso ficheiro .zip. Podem submeter o vosso trabalho as vezes que desejarem, **apenas a última submissão será contabilizada**. Certifiquem-se que a submissão foi concluída, e que esta última versão tem todas as alterações que desejam entregar dado que os docentes apenas considerarão esta última submissão;
- Avisamos que a hora de entrega final acontece sempre poucos **minutos antes da meia-noite**, pelo que se urge a que os alunos não esperem pela hora final para entregarem e o façam, idealmente um dia antes, ou no pior caso, pelo menos uma hora antes. Não serão consideradas válidas as entregas realizadas por e-mail. Poderão testar a entrega nos dias anteriores para perceberem se têm algum problema com a entrega, sendo que, novamente, apenas a última submissão conta.

Política em caso de fraude

O trabalho entregue deve corresponder ao esforço individual de cada aluno. São consideradas fraudes as seguintes situações: Trabalho parcialmente copiado, facilitar a cópia através da partilha de ficheiros, ou utilizar material alheio sem referir a sua fonte.

Em caso de deteção de algum tipo de fraude, os trabalhos em questão não serão avaliados, sendo enviados à Comissão Pedagógica ou ao Conselho Pedagógico, consoante a gravidade da situação, que decidirão a sanção a aplicar aos alunos envolvidos. Serão utilizadas as ferramentas *Moss* e *SafeAssign* para deteção automática de cópias. Recorda-se ainda que o Anexo I do Código de Conduta Académica, publicado a 25 de janeiro de 2016 em Diário da República, 2ª Série, nº 16, indica no seu ponto 2 que quando um trabalho ou outro elemento de avaliação apresentar um nível de coincidência elevado com outros trabalhos (percentagem de coincidência com outras fontes reportada no relatório que o referido software produz), cabe ao docente da UC, orientador ou a qualquer elemento do júri, após a análise qualitativa desse relatório, e em caso de se confirmar a suspeita de plágio, desencadear o respetivo procedimento disciplinar, de acordo com o Regulamento Disciplinar de Discentes do ISCTE - Instituto Universitário de Lisboa, aprovado pela deliberação nº 2246/2010, de 6 de dezembro.

O ponto 2.1 desse mesmo anexo indica ainda que no âmbito do Regulamento Disciplinar de Discentes do ISCTE-IUL, são definidas as sanções disciplinares aplicáveis e os seus efeitos, podendo estas variar entre a advertência e a interdição da frequência de atividades escolares no ISCTE-IUL até cinco anos.

Parte III – Comunicação usando IPC

Data de entrega: 15 de maio de 2021

Nesta parte do trabalho, os ficheiros a utilizar (os nomes devem ser exatamente estes, usando só minúsculas) são:

- "**cidadaos.dat**" e "**enfermeiros.dat**": Ficheiros binários com a informação sobre os Cidadãos e Enfermeiros, fornecido pelo sistema (copiar para a vossa diretoria a partir da diretoria existente no servidor Tigre **/home/so/trabalho-2020-2021/parte-3/**). **Os alunos deverão assumir que estes ficheiros podem vir a ter outros conteúdos ou outros tamanhos.** A estrutura dos ficheiros é, respetivamente, **Cidadao** e **Enfermeiro**.

As estruturas de dados necessárias para elaborar os módulos desta parte são **Cidadao**, **Enfermeiro** e **Vaga**:

(ficheiros disponibilizados aos alunos no Tigre acima)

`cidadao.c`: Módulo processo Cidadão

`servidor.c`: Módulo processo Servidor

`common.h`: Definições comuns a Cidadão e Servidor

`utils.h`: Macros utilitárias

`cidadaos.dat`: Ficheiro binário com base de dados de cidadãos, pode ser lido usando o Shell script **show-cidadaos.sh**

`enfermeiros.dat`: Ficheiro binário com lista de enfermeiros, pode ser lido usando o Shell script **show-enfermeiros.sh**

```c
typedef struct {
    int num_utente;
    char nome[100];
    int idade;
    char localidade[100];
    char nr_telemovel[10];
    int estado_vacinacao;
    int PID_cidadao;
} Cidadao;

typedef struct {
    int ced_profissional;
    char nome[100];
    char CS_enfermeiro[100];
    int nr_vacinas_dadas;
    int disponibilidade;
} Enfermeiro;

typedef struct {
    int index_enfermeiro;
    int index_cidadao;
    int PID_filho;
} Vaga;
```

```c
typedef struct {
    long tipo;
    struct {
        TipoPedido pedido;
        int num_utente;
        char nome[100];
        int PID_cidadao;
    } dados;
} MsgCliente;

typedef struct {
    long tipo;
    struct {
        StatusServidor status;
        Cidadao cidadao;
    } dados;
} MsgServidor;

typedef struct {
    Cidadao cidadaos[MAX_CIDADAOS];
    int num_cidadaos;
    Enfermeiro enfermeiros[MAX_ENFERMEIROS];
    int num_enfermeiros;
    Vaga vagas[MAX_VAGAS];
} Database;
```

Os alunos deverão, em vez de printf, utilizar sempre as macros "sucesso" e "erro" (definidas em utils.h; a sintaxe destas macros é igual à do printf) para escrever TODAS as mensagens, respetivamente, de sucesso e erro resultantes dos vários passos da aplicação.

cidadao.c

O módulo **Cidadão** simula, na prática, a chegada do cidadão ao centro de saúde da sua localidade para iniciar o processo de vacinação, seguindo as regras do plano de Vacinação.

Assim, definem-se as seguintes tarefas a desenvolver:

C1) Implemente a função `init_ipc()`, que tenta abrir uma fila de mensagens IPC que tem a KEY **IPC_KEY** definida em **common.h** (alterar esta KEY para ter o valor do nº do aluno, como indicado nas aulas). Deve assumir que a fila de mensagens já foi criada. Se tal não aconteceu, dá erro e termina com *exit status* **1**. Esta função, em caso de sucesso, preenche a variável global **msg_id**;

Outputs esperados (os itens entre <> deverão ser substituídos pelos valores correspondentes):

```
@@Error@@: init_ipc) Fila de Mensagens com a Key definida não existe ou não pode ser aberta
@@Success@@: C1) Fila de Mensagens com a Key <IPC_KEY> aberta com o ID <msg_id>
```

C2) Implemente a função `cria_mensagem()`, que:

C2.1) Pede ao **Cidadão** (utilizador) os seus dados, nomeadamente o número de utente e nome, **obrigatoriamente nessa ordem**, preenchendo os dados na variável global **mensagem**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C2.1) Dados Cidadão: <num_utente>, <nome>
```

C2.2) Preenche os campos PID_cidadao da variável global **mensagem** com o PID deste **processo Cidadão**, tipo da mensagem com o **tipo 1**, e **pedido = PEDIDO**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C2.2) PID Cidadão: <PID_Cidadao>
```

C3) Implemente a função `envia_mensagem_servidor()` que envia um pedido de consulta de vacinação para o processo **Servidor**, enviando a **mensagem** para a fila de mensagens; em caso de erro no envio, termina com erro e *exit status* **1**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível enviar mensagem para o servidor
@@Success@@: Mensagem para o servidor enviada
```

C4) Implemente a função `espera_resposta_servidor()`, que espera a resposta do processo **Servidor** (na fila de mensagens com o **tipo = PID_Cidadao**) e preenche a mensagem enviada pelo processo **Servidor** na variável global **resposta**; em caso de erro, termina com erro e *exit status* **1**.

Outputs esperados (os itens entre <> deverão ser substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível ler a resposta do servidor
@@Success@@: Servidor enviou resposta
```

C5) O comportamento do **processo Cidadão** agora irá depender da **resposta** enviada pelo processo **Servidor** no campo **status**. Implemente a função `trata_resposta_servidor()` que faz:

C5.1) Se o estado for **DESCONHECIDO**, imprime uma mensagem de erro, e termina com *exit status* **1**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: C5.1) Não existe registo do utente <num_utente>, <nome>
```

C5.2) Se o estado for **VACINADO**, imprime uma mensagem de sucesso, e termina com *exit status* **0**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C5.2) O utente <num_utente>, <nome> foi vacinado
```

C5.3) Se o estado for **EMCURSO**, imprime uma mensagem de sucesso, e termina com *exit status* **0**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C5.3) A vacinação do utente <num_utente>, <nome> já está em curso
```

C5.4) Se o estado for **AGUARDAR**, imprime uma mensagem de sucesso, aguarda (sem espera ativa!) um tempo **TEMPO_ESPERA** segundos, e depois retorna ao ponto **C3**, enviando novo pedido;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C5.4) Utente <num_utente>, <nome>, por favor aguarde...
```

C5.5) Se o estado for **OK**, imprime uma mensagem de sucesso, e depois vai para o ponto **C6**.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C5.5) Utente <num_utente>, <nome>, vai agora ser vacinado
```

C6) Inicia o processo de vacinação. Implemente a função `vacina()`, que:

C6.1) Chama a função `print_info(cidadao)` com a informação recebida na **resposta** do processo **Servidor**, que irá imprimir a informação completa sobre o cidadão que vai ser vacinado;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C6.1) Dados completos sobre o cidadão a ser vacinado
```

C6.2) Chama novamente a função `espera_resposta_servidor()`, que espera uma nova resposta do processo **Servidor** (na fila de mensagens com o **tipo = PID_Cidadao**) e preenche a mensagem enviada pelo processo **Servidor** na variável global **resposta**; em caso de erro, afixa uma mensagem de erro e termina o processo com *exit status* **1**.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível ler a resposta do servidor
@@Success@@: Servidor enviou resposta
```

C6.3) O comportamento do **processo Cidadão** agora irá depender da **resposta** enviada pelo processo **Servidor** no campo **status**:

C6.3.1) Se o estado for **TERMINADA**, imprime uma mensagem, e termina com *exit status* **0**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: C6.3.1) Utente <num_utente>, <nome> vacinado com sucesso
```

C6.3.2) Se o estado for **CANCEL**, imprime uma mensagem de erro, e termina com *exit status* **1**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: C6.3.2) O servidor cancelou a vacinação em curso
```

C7) O sinal **SIGINT** foi armado para que, quando o utilizador interromper o **processo Cidadão** com **<CTRL+C>**, chame a função `cancela_pedido()`. Implemente esta função tal que:

C7.1) Escreva no ecrã uma mensagem;

Outputs esperados (os itens entre <> deverão ser substituídos pelos valores correspondentes):

```
@@Success@@: C7.1) O cidadão cancelou a vacinação no processo <PID_Cidadao>
```

C7.2) Altera a variável global **mensagem**, tornando **pedido = CANCELAMENTO**. Chama a função `envia_mensagem_servidor()`, que envia a **mensagem** para a fila de mensagens; em caso de erro no envio, afixa uma mensagem de erro e termina com *exit status* **1**;

Outputs esperados (os itens entre <> deverão ser substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível enviar mensagem para o servidor
@@Success@@: Mensagem para o servidor enviada
```

C7.3) Chama novamente a função `espera_resposta_servidor()`, que espera e preenche a variável global **resposta** com a nova resposta do processo **Servidor** (na fila de mensagens com o **tipo = PID_Cidadao**);

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível ler a resposta do servidor
@@Success@@: Servidor enviou resposta
```

C7.4) O comportamento do **processo Cidadão** agora irá depender da **resposta** enviada pelo processo **Servidor**, no campo **status**:

C7.4.1) Se o estado for **CANCEL**, imprime mensagem de sucesso, e termina com *exit status* **0**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: 7.4.1) Servidor confirmou cancelamento
```

C7.4.2) Se o estado for **TERMINADA**, imprime mensagem sucesso, termina com *exit status* **0**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: 7.4.2) A vacinação já tinha sido concluída
```

servidor.c

O processo **Servidor** é responsável pela atribuição de um enfermeiro para administrar as vacinas aos cidadãos que chegam aos Centros de Saúde. Para efeitos de segurança e evitar o contágio, apenas poderão ser dadas no máximo NUM_VAGAS (valor definido em **common.h**) vacinas em simultâneo.

O processo **Servidor** é responsável pelas seguintes tarefas:

S1) Implemente a função `init_ipc()`, que tenta criar:

- uma fila de mensagens IPC;
- um array de semáforos IPC de dimensão 1;
- uma memória partilhada IPC de dimensão suficiente para conter um elemento Database.

Todos estes elementos têm em comum serem criados com a KEY **IPC_KEY** definida em **common.h** (alterar esta KEY para ter o valor do nº do aluno, como indicado nas aulas), e com **permissões 0600**. Se qualquer um destes elementos IPC já existia anteriormente, dá erro e termina com *exit status* **1**. Esta função, em caso de sucesso, preenche as variáveis globais respetivas **msg_id**, **sem_id**, e **shm_id**;

O semáforo em questão será usado com o padrão "Mutex", pelo que será iniciado com o valor 1;

Outputs esperados (os itens entre <> deverão ser substituídos pelos valores correspondentes)**:**

```
@@Error@@: init_ipc) Fila de Mensagens com a Key definida já existe ou não pode ser criada
@@Error@@: init_ipc) Semáforo com a Key definida já existe ou não pode ser criado
@@Error@@: init_ipc) Semáforo com a Key definida não pode ser iniciado com o valor 1
@@Error@@: init_ipc) Memória Partilhada com a Key definida já existe ou não pode ser criada
@@Success@@: S1) Criados elementos IPC com a Key <IPC_KEY>: MSG <msg_id>, SEM <sem_id>, SHM <shm_id>
```

S2) Chama a função `init_database()`, que inicia a base de dados:

- Associa a variável global **db** com o espaço de Memória Partilhada alocado para **shm_id**; se não o conseguir, dá erro e termina com *exit status* **1**;
- Lê o ficheiro **FILE_CIDADAOS** e armazena o seu conteúdo na base de dados usando a função `read_binary()`, assim preenchendo os campos db->**cidadaos** e db->**num_cidadaos**. Se não o conseguir, dá erro e termina com *exit status* **1**;
- Lê o ficheiro **FILE_ENFERMEIROS** e armazena o seu conteúdo na base de dados usando a função `read_binary()`, assim preenchendo os campos db->**enfermeiros** e db->**num_enfermeiros**. Se não o conseguir, dá erro e termina com *exit status* **1**;
- Inicia o array db->**vagas**, colocando todos os campos de todos os elementos com o valor -1.

Outputs esperados (os itens entre <> deverão ser substituídos pelos valores correspondentes)**:**

```
@@Error@@: init_database) Erro a ligar a Memória Dinâmica ao projeto
@@Error@@: read_binary) Erro na leitura do ficheiro FILE_CIDADAOS
@@Error@@: read_binary) Erro na leitura do ficheiro FILE_ENFERMEIROS
@@Success@@: S2) Base de dados carregada com <num_cidadaos> cidadãos e <num_enfermeiros> enfermeiros
```

S3) Implemente a função `espera_mensagem_cidadao()`, que espera uma **mensagem** (na fila de mensagens com o **tipo = 1**) e preenche a mensagem enviada pelo **processo Cidadão** na variável global **mensagem**; em caso de erro, termina com erro e *exit status* **1**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes)**:**

```
@@Error@@: Não é possível ler a mensagem do Cidadao
@@Success@@: Cidadão enviou mensagem
```

S4) O comportamento do processo **Servidor** agora irá depender da **mensagem** enviada pelo **processo Cidadão** no campo **pedido**. Implemente a função `trata_mensagem_cidadao()`, que:

S4.1) Se o pedido for **PEDIDO**, imprime uma mensagem e avança para o passo **S5**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes)**:**

```
@@Success@@: S4.1) Novo pedido de vacinação de <PID_cidadao> para <num_utente>
```

S4.2) Se o estado for **CANCELAMENTO**, imprime uma mensagem, e avança para o passo **S10**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes)**:**

```
@@Success@@: S4.2) Cancelamento de vacinação de <PID_cidadao> para <num_utente>
```

S5) Implemente a função `cria_pedido()`, que processa um pedido de vacinação e envia uma **resposta** ao **processo Cidadão**. Para tal, essa função faz vários checks, atualizando o campo **status** da **resposta**:

S5.1) Procura o **num_utente** e **nome** na base de dados (BD) de Cidadãos:

- Se o **num_utente** e **nome** do Cidadão for encontrado na BD Cidadãos, os dados do cidadão deverão ser copiados da BD Cidadãos para o campo **cidadao** da **resposta**;
- Se o utilizador (Cidadão) não for encontrado na BD Cidadãos ➔ **status = DESCONHECIDO**;
- Se o Cidadão na BD Cidadãos tiver `estado_vacinacao >= 2` ➔ **status = VACINADO**;
- Se o Cidadão na BD Cidadãos tiver `PID_cidadao > 0` ➔ **status = EMCURSO**.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes)**:**

```
@@Error@@: S5.1) Cidadão <num_utente>, <nome> não foi encontrado na BD Cidadãos
@@Success@@: S5.1) Cidadão <num_utente>, <nome> encontrado,
                   estado_vacinacao=<estado_vacinacao>, status=<status>
```

S5.2) Caso o Cidadão esteja em condições de ser vacinado (i.e., se status **não for** DESCONHECIDO, VACINADO nem EMCURSO), procura o enfermeiro correspondente na BD Enfermeiros:

- Se não houver centro de saúde, ou não houver nenhum enfermeiro no centro de saúde correspondente ➔ **status = NAOHAENFERMEIRO**;
- Se há enfermeiro, mas este não tiver disponibilidade ➔ **status = AGUARDAR**.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes)**:**

```
@@Error@@: S5.2) Enfermeiro do CS <localidade> não encontrado na BD Enfermeiros
@@Success@@: S5.2) Enfermeiro do CS <localidade> encontrado,
                   disponibilidade=<disponibilidade>, status=<status>
```

S5.3) Caso o enfermeiro esteja disponível, procura uma vaga para vacinação na BD Vagas. Para tal, chama a função `reserva_vaga(index_cidadao, index_enfermeiro)` usando os índices do Cidadão e do Enfermeiro nas respetivas BDs:

- Se essa função tiver encontrado e reservado um index **vaga_ativa** ➔ **status = OK**;
- Se essa função não conseguiu encontrar uma vaga livre ➔ **status = AGUARDAR**.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: S5.3) Não foi encontrada nenhuma vaga livre para vacinação
@@Success@@: S5.3) Foi reservada a vaga <vaga_ativa> para vacinação, status=<status>
```

S5.4) Se no final de todos os checks, **status for OK**, chama a função `vacina()`, caso contrário, chama a função `envia_resposta_cidadao()`, que envia a resposta ao Cidadão;

S6) Implemente a função `vacina()`, que processa a vacinação.

S6.1) Cria um processo filho através da função fork();

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: S6.1) Não foi possível criar um novo processo
@@Success@@: S6.1) Criado um processo filho com PID_filho=<PID_filho>
```

S6.2) O processo filho chama a função `servidor_dedicado()`;

S6.3) O processo pai regista o process ID do processo filho no campo PID_filho na BD de Vagas com o índice da variável global **vaga_ativa**;

S7) Implemente a função `servidor_dedicado()`, que define o comportamento do servidor dedicado:

S7.1) Arma o sinal SIGTERM. Implemente também a função de tratamento deste sinal, `termina_servidor_dedicado()`, que envia nova resposta para o Cidadao, chamando a função `envia_resposta_cidadao()` contendo os dados do Cidadao preenchidos em **S5.1** e o campo **status=CANCEL**, para indicar que a consulta foi cancelada;

S7.2) Envia a resposta para o Cidadao, chamando a função `envia_resposta_cidadao()`. Implemente também esta função, que envia a mensagem **resposta** para o cidadao, contendo os dados do Cidadao preenchidos em **S5.1** e o campo **status = OK**;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível enviar resposta para o cidadão
@@Success@@: Resposta para o cidadão enviada
```

S7.3) Coloca a disponibilidade do enfermeiro afeto à **vaga_ativa** com o valor 0 (Indisponível);

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: S7.3) Enfermeiro associado à vaga <vaga_ativa> indisponível
```

S7.4) Imprime uma mensagem e aguarda (em espera passiva!) **TEMPO_CONSULTA** segundos;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: S7.4) Vacina em curso para o cidadão <num_utente>, <nome>, e com o
                   enfermeiro <ced_profissional>, <nome> na vaga <vaga_ativa>
```

S7.5) Envia nova resposta para o Cidadao, chamando a função `envia_resposta_cidadao()` contendo os dados do Cidadao preenchidos em **S5.1** e o campo **status=TERMINADA**, para indicar que a consulta terminou com sucesso;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: Não é possível enviar resposta para o cidadão
@@Success@@: Resposta para o cidadão enviada
```

S7.6) Atualiza os dados do cidadão (**estado_vacinacao**) na BD de Cidadãos, e do enfermeiro (incrementa **nr_vacinas_dadas** e coloca **disponibilidade=1**) na BD de Enfermeiros;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: Cidadão atualizado na BD para estado_vacinacao=<estado_vacinacao>,
             Enfermeiro atualizado na BD para nr_vacinas_dadas=<nr_vacinas_dadas> e
             disponibilidade=<disponibilidade>
```

S7.7) Liberta a vaga **vaga_ativa** da BD de Vagas, invocando a função `liberta_vaga(vaga_ativa)`;

S7.8) Termina o processo Servidor Dedicado (filho) com *exit status* **0**.

S8) Implemente a função `reserva_vaga(index_cidadao, index_enfermeiro)`, que tentará reservar uma vaga livre na BD de Vagas. Para tal:

S8.1) Procura uma vaga livre (`index_cidadao < 0`) na BD de Vagas. Se encontrar uma entrada livre:

S8.1.1) Atualiza o valor da variável global **vaga_ativa** com o índice da vaga encontrada;

S8.1.2) Atualiza a entrada de Vagas **vaga_ativa** com o índice do cidadão e do enfermeiro

S8.1.3) Retorna o valor do índice de vagas **vaga_ativa** ou **-1** se não encontrou nenhuma vaga

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: S8.1.3) Não foi encontrada nenhuma vaga livre
@@Success@@: S8.1.3) Foi reservada a vaga livre com o index <vaga_ativa>
```

S9) Implemente a função `liberta_vaga(index_vaga)`, que liberta a vaga da BD de Vagas, colocando o campo **index_cidadao** dessa entrada da BD de Vagas com o valor **-1**.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: S9) A vaga com o index <index_vaga> foi libertada
```

S10) Implemente a função `cancela_pedido()`, que processa o cancelamento de um pedido de vacinação e envia uma resposta ao **processo Cidadão**. Para este efeito, a função:

S10.1) Procura na BD de Vagas a vaga correspondente ao Cidadao em questão (procura por **`index_cidadao`**). Se encontrar a entrada correspondente, obtém o PID_filho do Servidor Dedicado correspondente;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Error@@: S10.1) Não foi encontrada nenhuma sessão de vacinação com o cidadão
                  <num_utente>, <nome>
@@Success@@: S10.1) Foi encontrada a sessão do cidadão <num_utente>, <nome>
                    na sala com o index <index_vaga>
```

S10.2) Envia um sinal **SIGTERM** ao processo **Servidor Dedicado** (filho) que está a tratar da vacinação;

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: S10.2) Enviado sinal SIGTERM ao Servidor Dedicado com
                    PID=<PID_filho>
```

S11) Implemente a função `termina_servidor()`, que irá tratar do fecho do servidor, e que:

S11.1) Envia um sinal **SIGTERM** a todos os processos **Servidor Dedicado** (filhos) ativos;

S11.2) Grava o ficheiro **FILE_ENFERMEIROS,** usando a função `save_binary()`;

S11.3) Grava o ficheiro **FILE_CIDADAOS,** usando a função `save_binary()`;

S11.4) Remove do sistema (IPC Remove) os semáforos, a Memória Partilhada e a Fila de Mensagens.

Outputs esperados (itens entre <> substituídos pelos valores correspondentes):

```
@@Success@@: S11.4) Servidor Terminado
```

NOTA Importante:

Apesar de não estar explicitamente indicado no enunciado, os alunos já conhecem os problemas relativos a exclusão mútua no acesso à Memória Partilhada, pelo que deverão tomar as devidas precauções para que não haja conflitos no acesso à mesma entre o Servidor e os Servidores Dedicados. Relembra-se que a garantia de exclusão deverá ocupar o menor número de tempo possível, para permitir a maior concorrência possível entre os processos a aceder às zonas críticas.

Boa sorte!!!