Nome do estudante: _____ **Código:** _____

1. [0.8] Explique a relação entre "multiprogramação" e "DMA" (*Direct Memory Access*).

2.
a) [1.2] Num sistema de computação executam dois *threads*, **T1** e **T2**, do mesmo processo. **T1** produz dados que devem ser processados por **T2**. Os dados produzidos são colocados numa variável partilhada, **pdata**, do tipo **PData**. As funções **void put(PData *x)** e **PData get()** permitem, respetivamente, escrever e ler a referida variável. Considerando que dispõe das funções **init(sem,value)**, **wait(sem)** e **signal(sem)** que operam sobre semáforos, indique como implementaria a parte do código de **T1** e **T2** que garante a sincronização destes *threads*: pretende-se que todos os dados produzidos sejam processados e que os mesmos dados não sejam processados mais do que uma vez. Indique também onde deve ser guardada a variável partilhada e como e onde deve ser feita a inicialização do(s) semáforo(s).

b) [1.2]
b1) Comente a seguinte afirmação: uma secção crítica é um pedaço de código que tem de ser executado ininterruptamente.
b2) Explique como é que, na prática, se garante que certo código é executado numa secção crítica.

b1)

b2)

c) [0.8] Em geral, os sistemas operativos ignoram os *deadlocks* nos processos dos utilizadores. Qual a principal medida que os programadores podem usar para que os *deadlocks* não ocorram nos seus processos? Dê um exemplo.

3. [0.8] Em geral, os algoritmos de escalonamento do processador penalizam certo(s) tipo(s) de processos. Que tipo(s)? Como o(s) carateriza?

4.

a) [1.0] Explique, de forma simples mas clara, como é possível executar um processo que, globalmente, ocupa mais memória do que a memória física disponível. Diga o que é necessário para que isso seja possível.

b) [1.0] Explique a importância de saber como são guardados em memória certos tipos de dados estruturados, disponíveis em muitas linguagens de programação, para escrever código eficiente que execute num sistema operativo com gestão de memória virtual. Dê um exemplo, se achar conveniente.

5. [1.2] Em Linux, explique como é estabelecida a relação entre o nome de um ficheiro e os blocos de dados que o compõem. Refira as estruturas de dados do sistema operativo que são utilizadas para esse fim.

Nome do estudante: _____ **Código:** _____

Nota: nas questões seguintes apenas é necessário fazer tratamento de erros nos casos indicados explicitamente

6.

O programa **find_and_copy** percorre o diretório **dir_to_search** e seus subdiretórios à procura de ficheiros cujo nome (completo ou parcial) seja **filename**, copiando esses ficheiros para o diretório **destination_dir**, sendo invocado na linha de comando da seguinte forma: **$ find_and_copy <dir_to_search> <filename> <destination_dir>** .

O programa usa a função **process_dir(dirname),** cujo código parcial se apresenta abaixo, para pesquisar recursivamente na árvore de ficheiros (subdiretórios e ficheiros regulares) da pasta **dirname** os ficheiros a copiar.

```
// variáveis globais: filename, destination_dir
_____ // A PREENCHER na alínea a)
//-------------------------------------------------------------------------------------------------------
int process_dir(char *dirname)
{
  DIR *dir;
  struct dirent *entry;
  struct stat statbuf;
  if (!(dir = opendir(dirname))) return 1;
  while ((entry = readdir(dir)) != NULL) {
    char path[1024];
    // --- BLOCO A ---
    ...
    if (...) { // se 'entry' for um diretório
      if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0)
        continue;
      ... // cria um processo que invoca process_dir()
    }
    // --- FIM DO BLOCO A ---
    // --- BLOCO B ---
    else if (...) { // se 'entry' for um ficheiro regular
      ...
      if (strstr(entry->d_name, filename) != NULL) { // se o nome do ficheiro contiver filename
        ... // cria um processo que invoca o utilitário 'cp'
      }
    }
    // --- FIM DO BLOCO B ---
  }
  return 0;
}
```

a) [1.0] Escreva o código da função **main()** do programa **find_and_copy**, a qual deve: 1) verificar se o número de argumentos do programa é correto, terminando o programa caso não seja; 2) atualizar o valor das variáveis globais do programa: **destination_dir** e **filename** (faça a declaração destas variáveis no espaço reservado no código acima); 3) informar o sistema operativo que se pretende ignorar o sinal habitualmente gerado quando se tecla **CTRL-C**; 4) invocar a função **process_dir**, apresentada acima.

b) [1.5] Implemente o código do **BLOCO A** da função **process_dir**, o qual, sempre que é encontrado um novo subdiretório, cria um processo-filho que faz uma chamada a **process_dir(dirname)**. Explique, numa frase, o que aconteceria se a instrução **if (strcmp(…) || strcmp(…))** tivesse sido omitida.

---

c) [1.5] Implemente o código do **BLOCO B** da função **process_dir**, o qual, quando encontrar um ficheiro com o nome igual ou parcialmente igual a **filename**, cria um processo-filho que trata da cópia desse ficheiro para o diretório **destination_dir**. Esta cópia deve ser efetuada com uma chamada a uma das funções **exec** e ao utilitário **cp**.

Nome do estudante: _____ **Código:** _____

7.

a) [1.5] O programa **compress** obtém da entrada padrão uma sequência de *bytes*, apresentando na saída padrão uma correspondente sequência comprimida. O programa **myprog** recebe como argumento da linha de comando o nome de um ficheiro e escreve na saída padrão o resultado da compressão do ficheiro, que é feita utilizando **compress**.

Escreva o código de **myprog** e faça um esquema dos canais de comunicação do tipo "pipe" que utilizar.

Nota: use a função **full_copy()**, apresentada ao lado.

```
void full_copy(int from, int to) {
  char buffer[BUF_SIZE]; // BUF_SIZE < 1000
  size_t n;
  while ((n = read(from, buffer, BUF_SIZE)) > 0)
    write(to, buffer, n);
  close(from);
  close(to);
}
```

---

b) [1.5] O programa **compressF** efetua o mesmo tipo de compressão que **compress**, mas funciona com FIFOs, recebendo blocos de *bytes* por um FIFO denominado **requests** e retornando correspondentes blocos de *bytes* comprimidos por um FIFO denomina-

```
struct Request {
  pid_t pid;
  size_t nbytes;
  char data[1000];
};
```

```
struct Reply {
  size_t nbytes;
  char data[1000];
};
```

do **ans<pid>**, onde **pid** é o identificador do processo que contacta **compressF**. Os blocos de entrada e saída têm o formato indicado, respetivamente, pelas *struct*'s **Request** e **Reply** (ver página anterior).

Elabore um esquema dos canais de comunicação quando são executadas 2 instâncias de **myprogF** e escreva o código do programa **myprogF** que, de forma semelhante a **myprog** da alínea a), mas agora utilizando **compressF**, faça a compressão de um ficheiro cujo nome obtém da linha de comando. **Notas**: **1)** Considere que o conteúdo do ficheiro a comprimir não excede 1000 *bytes*; **2)** use a função **createRequest()**, apresentada ao lado, em que o parâmetro **filename** representa o nome do ficheiro a comprimir.

```
void createRequest(const char *filename,
                   struct Request *req)  {
  int file = open(filename, O_RDONLY);
  char buffer[BUF_SIZE]; // BUF_SIZE < 1000
  size_t n;
  req->pid = getpid(); req->nbytes = 0;
  while ((n = read(file, buffer, BUF_SIZE)) > 0) {
    memcpy(&req->data[req->nbytes], buffer, n);
    req->nbytes += n;
  }
  close(file);
}
```

---

c) [1.0] Se o campo **data** das *struct*'s **Request** e **Reply** passasse a ter um tamanho de 100000 bytes, que problema(s) pode(m) surgir na comunicação entre os processos **myprogF** e **compressF**? Justifique.

Nome do estudante: _____ **Código:** _____

8.

Considere a seguinte situação da vida real que será simulada em computador, com *threads* e primitivas de sincronização da norma POSIX. A notação e as variáveis a utilizar na simulação vão aparecendo no texto à medida do que for necessário.

Uma ave (**bird**) tem a seu cargo a alimentação de **NB** avezinhas (**baby**), que não conseguem sair do ninho. Quando estão com fome e já não há bocados de comida por perto (**food_bits** é **0**), as avezinhas piam, de forma a alertar a ave que, por isso, vai buscar mais comida. A comida chega sempre em lotes de **F** bocados e as avezinhas comem bocado a bocado.

a) [1.0] Complete o excerto de código ao lado, por forma a se pôr em execução todos os *threads* que representam os personagens da vida real. As variáveis mostradas são auto-explicativas e sabe-se que os *threads* **baby()** recebem o seu número identificador (**0**, **1**, **2**, …) como parâmetro.

```
// global:
#define NB 10
int food_bits = 0;
-------------------------------------------
// in main():
pthread_t tid_bird, tid_baby[NB];
...
pthread_create(&tid_bird, NULL, bird, NULL);
for (...) {
  ...
  pthread_create(..., ..., baby, ...);
}
```

b) [1.0] De tudo o que já foi dito e tendo como orientação o código ao lado, da função-*thread* **bird()**, escreva o código da função-*thread* **baby()**, nas seguintes condições:

- cada avezinha, identificada pelo valor que recebe como parâmetro, localmente guardado em **int id**, vai anotando os bocados que come numa variável local, **int n_bits**;
- as avezinhas estão sempre a comer (ou a tentar fazê-lo) e, quando ficam sem comida, o seu alerta é uma mensagem:
  fprintf(stderr, "I am baby %d, I have already eaten %d bits of food and I am still hungry!", id, n_bits);
- não são usadas quaisquer primitivas de sincronização;
- toda a atividade continua eternamente.

```
// global:
int finish = 0;
-------------------------
void *bird(void *arg) {
  while (finish == 0) {
    if (food_bits == 0) {
      get_food();
      food_bits = F;
    }
  }
  return NULL;
}
```

c) [1.0] Altere o código da função-*thread* **bird()** apresentada em **b)** para que nela, mediante o uso das primitivas de sincronização que achar convenientes,

- a situação de competição (*race condition*) associada a **food_bits** seja tratada;
- deixe de haver espera ativa (*busy-waiting*).

Clarifique a eventual necessidade de criar mais variáveis globais; se a houver, defina-as.

Explique por palavras o que terá de suceder da parte da função-*thread* **baby()** para que o código de **bird()**, que escreveu, funcione corretamente.

d) [1.0] Mais tarde ou mais cedo, cada uma das avezinhas tem de abandonar o ninho (**finish** passará a **1**) . Escreva:

- o código da parte final da função-*thread* **baby()** em que a informação do número de bocados de comida que foram consumidos seja passada ao *thread* principal, **main()**, por via do retorno de **baby()**;
- o código de **main()** em que se espera a finalização da atividade das avezinhas e se imprime para cada uma a mensagem **fprintf(stdout, "Baby number %d has eaten %d bits of food", i, total_bits)**, onde **i** e **total_bits** são inteiros locais de **main()**.

FIM