Fundamentos de Sistemas de Operação
1º Teste, 22 de Outubro de 2022

NOME DO ESTUDANTE: _____ Nº: _____

A duração do teste é 1h45 (incluindo a tolerância). Nas questões de escolha múltipla, as respostas erradas têm uma cotação negativa correspondente a 20% da classificação da questão. Por exemplo, se errar a resposta a uma questão cotada para 1.0 valor, terá uma cotação de –0,2 valores nessa questão. A classificação total das questões de escolha múltipla pode, portanto, ser negativa.

As questões de escolha múltipla devem ser respondidas na folha própria para o efeito.
Para anular uma resposta coloque uma cruz por cima e pinte a nova resposta (✗ ○ ○ ● ○).
Para reativar uma resposta previamente anulada faça um círculo à volta da resposta a reativar (⊗ ○ ○ ✗ ○).

As questões de desenvolvimento devem ser respondidas no próprio enunciado.

As última folha (páginas 13 e 14) pode ser destacada e usada para rascunho.

Fundamentos de Sistemas de Operação
1º Teste, 22 de Outubro de 2022

QUESTÕES DE ESCOLHA MÚLTIPLA — VERSÃO A

1) Considere a realização de duas funções **`myRCbegin()`** e a sua "dual" ou "complementar", **`myRCend()`**, que se querem baseadas em mutexes e que se destinam, respectivamente, a "marcar" o início e o fim regiões críticas; a "caixa" no meio da figura mostra a inicialização.

```
// Declaração e inicialização do mutex
pthread_mutex_t mtx= inicial;
```

```
void myRCbegin(pthread_mutex_t *m) {
  func1(m);
}
```

```
void myRCend(pthread_mutex_t *m) {
  func2(m);
}
```

Qual das alíneas abaixo representa a informação em falta para uma implementação correcta?

a) *func1* é **`pthread_mutex_lock`**, *func2* é **`pthread_mutex_unlock`**, *inicial* é **`0`**
b) *func1* é **`pthread_mutex_lock`**, *func2* é **`pthread_mutex_unlock`**, *inicial* é **`1`**
c) *func1* é **`pthread_mutex_lock`**, *func2* é **`pthread_mutex_unlock`**, *inicial* é **`PTHREAD_MUTEX_INITIALIZER`**
d) *func1* é **`sem_post`**, *func2* é **`sem_wait`**, *inicial* é **`0`**
e) *func1* é **`sem_wait`**, *func2* é **`sem_post`**, *inicial* é **`1`**

2) Num sistema de operação, muitas das técnicas e opções utilizadas no seu "desenho" (*design*) ou implementação são motivadas pela necessidade de acelerar a execução de aplicações e diminuir a latência de acesso aos dados. Assim, é importante conhecer as ordens de grandeza (valores típicos) de tempos de: execução de instruções sobre registos do CPU (tCPUins), acesso a RAM (tRAM), acesso a um bloco (não *cached*) de disco num HDD (tHDD) e num SSD (tSSD), tradução de endereço virtual/físico num TLB aquando de um hit (tHIT) ou miss (tMISS), etc., pois só assim se compreende o SO. Considerando as seguintes unidades **`ns`** (nano-segundos) **`µs`** (micro-segundos) e **`ms`** (mili-segundos), qual das opções abaixo representa valores credíveis?

a) tCPUins: dezenas de ns; tRAM: dezenas de ns; tHDD: dezenas de ms; tSSD: µs; tHIT: dezenas de ns; tMISS: dezenas de µs
b) tCPUins: ns; tRAM: dezenas de ns; tHDD: ms; tSSD: dezenas/centenas de µs; tHIT: ns; tMISS: ns
c) tCPUins: ns; tRAM: ms; tHDD: ms; tSSD: dezenas/centenas de µs; tHIT: ns; tMISS: dezenas de µs
d) tCPUins: ns; tRAM: dezenas de ns; tHDD: ms; tSSD: ms; tHIT: ns; tMISS: dezenas de µs
e) tCPUins: ns; tRAM: dezenas de ns; tHDD: ms; tSSD: dezenas/centenas de µs; tHIT: ns; tMISS: dezenas de µs

3) O objetivo **fundamental** de um sistema de operação é:

a) isolar entre si os utilizadores e os seus processos
b) suportar a execução de aplicações
c) permitir aos utilizadores aceder com facilidade ao hardware
d) permitir aos programadores aceder com facilidade ao hardware
e) permitir aos administradores de sistema gerir com facilidade o hardware

4) Uma região crítica (RC) é uma zona de código:

a) executada por dois (ou mais) fluxos concorrentes de instruções que estão localizadas entre um **`pthread_mutex_unlock`** , que marca o início da RC, e um **`pthread_mutex_lock`**, que marca o fim da RC

b) na qual dois (ou mais) fluxos concorrentes de instruções alteram o estado de dados/recursos partilhados potencialmente produzindo um resultado (e.g., um "valor") final que depende da ordem pela qual foram escalonadas ("executadas") as instruções desses fluxos

c) executada por dois (ou mais) fluxos concorrentes de instruções que estão localizadas entre um **`pthread_mutex_lock`** , que marca o início da RC, e um **`pthread_mutex_unlock`**, que marca o fim da RC

d) executada por dois (ou mais) fluxos concorrentes de instruções que estão localizadas entre um **`sem_wait`**, que marca o início da RC, e um **`sem_post`**, que marca o fim da RC

e) executada por dois (ou mais) fluxos concorrentes de instruções que estão localizadas entre um **`sem_post`**, que marca o início da RC, e um **`sem_wait`**, que marca o fim da RC

5) Considere a realização de duas funções **`depositar()`** e a sua "dual" ou "complementar", **`levantar()`**, que se querem executadas por duas (ou mais) *threads* distintas e que operam sobre o saldo de uma **mesma** conta bancária (i.e., o apontador aponta a mesma variável). Será que a execução concorrente das duas operações (funções) quando, e.g., um utilizador executa um levantamento no multibanco "ao mesmo tempo" que um depósito do seu salário é efectuado pela sua entidade empregadora pode conduzir a um saldo incorrecto?

```
void depositar(int val, int *saldo) {
   (*saldo)+= val;
}
```

```
void levantar(int val, int *saldo) {
   (*saldo)-= val;
}
```

a) Não, porque nos processadores modernos a execução de uma instrução C **+=** é atómica

b) Não se o computador no qual se executam as operações só tiver um CPU (só com um *core*); sim, se o CPU for *multicore*, ou se houver vários CPUs

c) Não, mas só se o(s) processador(es) nos quais se executam as operações forem Intel ou AMD, pois nesses a execução de instruções C **+=** é atómica

d) Sim, porque a execução das instruções-máquina de uma das funções pode ser suspensa para passar à execução das instruções-máquina da outra e, quando a primeira for retomada, já o valor de **`*saldo`** não corresponder ao valor guardado aquando da suspensão

e) Sim, porque **`saldo`** não é um **`int`**, mas sim um pointer e a manipulação de valores via apontadores é atómica

6) A característica **fundamental** de um sistema de operação que suporta multiprogramação é permitir:

a) controlar de forma concorrente todo o hardware

b) executar de forma concorrente vários processos

c) executar de forma concorrente várias *threads*

d) controlar de forma concorrente vários periféricos

e) controlar de forma concorrente vários CPUs

7) O espaço de endereçamento de um processo é:

a) o conjunto de todas as posições de memória real

b) o conjunto das posições de memória virtual que potencialmente podem ser acedidas pelo processo durante a sua execução

c) o conjunto de posições de memória virtual que constituem as zonas de código, *stack* e *heap*

d) o conjunto das posições de memória física que podem ser acedidas pelo processo durante a sua execução

e) o conjunto de todas as posições de memória física, do menor ao maior endereço físico

8) Considere o fragmento de programa abaixo. Qual das alíneas descreve o comportamento do programa?

```
void fill(char letra, char b[]) { for(int i=0; i<10; i++) b[i]= letra; }

void *fun(void *a){
  char bif[20]; memset(bif, 0, 20);
  int fd=open("f",O_RDONLY);                                    //(1)
  int e=read(fd, bif, 10);                                      //(2)
  return NULL;
}

int main(...) {
  char buf[10];
  int fd=open("f",O_WRONLY|O_CREAT|O_TRUNC, 0666);
    for(int i=0; i<1000; i++) {
      fill('A'+(i%26), buf); write(fd, buf, 10);
    }
  close(fd);

  pthread_t t1, t2;
  pthread_create(&t1, NULL, fun, NULL);
  pthread_create(&t2, NULL, fun, NULL);
  pthread_join(t2, NULL);  pthread_join(t1, NULL);              //(3)
  ...
}
```

a) Em (1), abre-se o ficheiro numa *thread*, mas quando a outra tenta fazer o mesmo, o programa termina com *segmentation fault*

b) Em (1), abre-se o ficheiro numa *thread*, mas quando a outra faz o mesmo, o **`fd`** da primeira é "esmagado" pelo valor obtido na última, pelo que o ficheiro só está aberto uma vez; em (2) uma das *threads* lê 10 caracteres **`'A'`** e a outra lê 10 caracteres **`'B'`** ficando o *buffer* **`bif`** com 10 caracteres **`'A'`** seguidos de 10 caracteres **`'B'`**; em (3) a ordem está trocada, o "join" de **`t1`** tem de ser feito antes de **`t2`**, senão o programa fica bloqueado (*deadlocked*)

c) Em (1), cada *thread* abre o ficheiro uma vez; em (2) ambas as *threads* leem 10 caracteres **`'A'`** ficando cada *buffer* **`bif`** com 10 caracteres **`'A'`** seguidos de 10 caracteres "nulos"; em (3) cada "join" espera pelo fim da correspondente *thread*

d) Em (1), cada *thread* abre o ficheiro uma vez; em (2) ambas as *threads* leem 10 caracteres **`'A'`** ficando um dos *buffers* **`bif`** com 10 caracteres **`'A'`** seguidos de 10 caracteres "nulos" e o outro *buffer* **`bif`** com 10 caracteres seguidos de 10 caracteres "nulos"; em (3) cada "join" espera pelo fim da correspondente *thread*

e) Em (1), cada *thread* abre o ficheiro uma vez; em (2) ambas as *threads* leem 10 caracteres **`'A'`** ficando um dos *buffers* **`bif`** com 10 caracteres **`'A'`** seguidos de 10 caracteres "nulos" e o outro *buffer* **`bif`** com 10 caracteres **`'B'`** seguidos de 10 caracteres "nulos"; em (3) a ordem está trocada, o "join" de **`t1`** tem de ser feito antes de **`t2`**, senão o programa fica bloqueado (*deadlocked*)

9) Quando se usa uma instrução similar a **`fd=open("f", O_RDWR)`** para abrir um ficheiro para leitura e escrita, os dados "movimentados" com instruções **`read()`** e **`write()`** estão sujeitos a *caching* efectuado pelo SO,

a) o que aumenta a velocidade das operações de escrita, que deixam de ser bloqueantes, mas pode causar perda de dados no caso de falhas de energia

b) o que aumenta a velocidade das operações de leitura, que deixam de ser bloqueantes

c) o que aumenta a velocidade das operações de leitura, que deixam de ser síncronas, e escrita, que deixam de ser bloqueantes mas pode causar perda de dados no caso de situações que causem *crashes* do SO, ou em casos de falhas de energia

d) o que aumenta a velocidade das operações de leitura e escrita, que deixam de ser síncronas, mas pode causar perda de dados no caso de situações que causem *crashes* do SO, ou em casos de falhas de energia

e) o que aumenta a velocidade das operações de leitura e escrita, mas pode causar perda de dados no caso de situações que causem *crashes* do SO, ou em casos de falhas de energia

10) Considere o fragmento de programa abaixo. Qual das alíneas descreve o comportamento do programa?

```
void fill(char letra, char b[]) { for(int i=0; i<10; i++) b[i]= letra; }

int main(...) {
  char buf[10];
  int fd=open("f",O_WRONLY|O_CREAT|O_TRUNC, 0666);
    for(int i=0; i<1000; i++) {
      fill('A'+(i%26), buf); write(fd, buf, 10);
    }
  close(fd);
  char bif[20]; int e; memset(bif, 0, 20);
  fd=open("f",O_RDONLY);
  e=read(fd, bif, 20);                                    //(1)
  if (fork()) { e=read(fd, bif, 10); wait(NULL); }        //(2)
  else        { e=read(fd, bif, 10); exit(0); }           //(3)
  lseek(fd, 10000, SEEK_SET);
  e=read(fd, bif, 20);                                    //(4)
...
}
```

a) Em (1) **e==20** e **bif** tem 10 A seguidos de 10 B; em (2) **e==10** e **bif** tem 10 C seguidos de 10 B ou então 10 D seguidos de 10 B; em (3) **e==10** e **bif** tem 10 C seguidos de 10 B ou então 10 D seguidos de 10 B; em (4) **e==0** e **bif** tem o mesmo conteúdo que tinha anteriormente

b) Em (1) **e==-1** e **bif** tem conteúdo indefinido; em (2) **e==10** e **bif** tem 10 A seguidos de 10 caracteres indefinidos ou então 10 B seguidos de 10 caracteres indefinidos; em (3) **e==10** e **bif** tem 10 A seguidos de 10 caracteres indefinidos ou então 10 B seguidos de 10 caracteres indefinidos; em (4) **e==-1** e **bif** tem conteúdo indefinido

c) Em (1) há um *segmentation fault* e o programa termina

d) O programa termina com erro (**errno** indica esse erro) na instrução "**fd=open("f",O_RDONLY);**" que está antes de (1), pois está-se a abrir uma 2ª vez o mesmo ficheiro, agora em "leitura", o que é incompatível com a anterior abertura em "escrita"

e) Nenhuma das opções anteriores descreve o comportamento do programa

11) Durante a vida de um processo, este passa por diferentes estados; os mais relevantes são:

a) Em espera, ou Adormecido (*Waiting* ou *Sleeping*); Em execução (*Running*); Substituído (*Preempted*)

b) Pronto (*Ready*); Em espera, ou Adormecido (*Waiting* ou *Sleeping*); Escalonado (*Scheduled*)

c) Pronto (*Ready*); Em execução (*Running*); Fim de Fatia de Tempo (*End of Timeslice*)

d) Pronto (*Ready*); Em espera, ou Adormecido (*Waiting* ou *Sleeping*); Em execução (*Running*)

e) Pronto (*Ready*); Em espera, ou Adormecido (*Waiting* ou *Sleeping*); Despachado (*Dispatched*)

12) As estratégias fundamentais usadas pelos SOs modernos para promover ("implementar") a execução concorrente de vários processos consistem em executar um outro processo quando o processo que está em execução

a) desencadeia uma operação de I/O

b) esgota a sua fatia-de-tempo (*timeslice*)

c) desencadeia uma operação bloqueante ou esgota a sua fatia-de-tempo

d) faz uma chamada de uma função de sistema

e) precisa de mais memória

13) Considere o fragmento de programa abaixo. Qual das alíneas descreve o resultado de uma execução standard do programa (assim: **`$ ./programa`**)?

```
int main(...) {
  close(0); close(1);
  int p[2]; pipe(p);

  if (fork()) {
    close(0);                              //(1)
    write(1, "ola\n". 4);                  //(2)
    wait(NULL);
  } else {
    close(1);                              //(3)
    char buf[5]; memset(buf, 0, 5);
    read(0, buf, 4);                       //(4)
    write(2, buf, 4);                      //(5)
  }
...
}
```

a) No ecrã aparece **`ola`** e o cursor muda de linha
b) Em (1) há um erro pois o canal 0 já estava fechado, mas a execução continua; em (2) há um erro porque o canal 1 está fechado, e o programa fica bloqueado no **`wait()`**
c) Em (2) há um erro pois o canal 1 está fechado, e o programa fica bloqueado no **`wait()`**
d) Em (3) há um erro pois o canal 1 já estava fechado, mas a execução continua; em (4) há um erro porque o canal 0 está fechado; no ecrã aparece "lixo" por causa de (5) escrever um **`buf`** que tem zeros
e) O programa aborta com "*segmentation fault*"

14) O espaço de endereçamento de um processo típico é constituído por várias regiões; as mais relevantes são

a) *stack*, *heap*, *buffers*, código e dados
b) *stack*, *heap*, *buffers*, *cache*, código e dados
c) *stack*, *heap*, *cache*, código e dados
d) *stack*, *heap*, *mapped I/O*, código e dados
e) *stack*, *heap*, código e dados

15) Para que um processo em execução possa ser retirado do processador (a.k.a. CPU) para, no seu lugar, ser colocado um outro, o SO tem de guardar alguma informação sobre o estado do processo "que vai sair" e repor a correspondente informação de estado do processo "que vai entrar". O módulo, ou "entidade" que, no SO, desempenha essa tarefa designa-se

a) *device driver*
b) CPU *driver*
c) *dispatcher* ou escalonador de curto prazo (*short-term scheduler*)
d) CPU *handler*
e) TLB (*Translate Look-aside Buffer*)

16) Para que um processo em execução possa ser retirado do processador (a.k.a. CPU) para, no seu lugar, ser colocado um outro, o SO tem de guardar alguma informação sobre o estado do processo "que vai sair" e repor a correspondente informação de estado do processo "que vai entrar". De entre os vários itens de informação, o(s) mais importante(s) é/são

a) os registos do CPU
b) as páginas de memória e os registos do CPU
c) o estado de todos os periféricos e os registos do CPU
d) o mapa do TLB (*Translate Look-aside Buffer*) e os registos do CPU
e) os registos de todos os CPUs (ou *cores*, no caso de um CPU *multicore*)

17) O que, fundamentalmente, distingue um processo de um thread é:

a) Um processo tem as seguintes zonas: código, dados, stack e heap; os threads não têm stack, usam o do processo
b) Os threads dispõem de mecanismos para exclusão mútua; os processos, não
c) Um processo pode lançar threads, mas um thread não pode lançar processos
d) Os processos, por omissão, não partilham o espaço de endereçamento entre si; os threads, partilham
e) Nenhuma das opções anteriores é verdadeira

18) Qual é o número de processos **novos/criados** (excluindo, portanto, o processo inicial que os criou) quando um processo executa duas instruções **`fork()`** em sequência, i.e.,

a) 1
b) 2
c) 3
d) 4
e) Erro: não é permitido executar *forks* seguidos sem um *if* em cada um

```
…
fork();
fork();
…
```

19) Qual o número de **novas** (excluindo, portanto a main thread inicial) *threads* **activas** no ponto de observação **`<Ponto P>`** quando um processo executa a sequência de instruções na "caixa" i.e.,

a) 0
b) 1
c) 2
d) Programa errado; a ordem dos joins está trocada
e) Programa errado, a função f não pode ser lançada 2 vezes

```
pthread_create(&t,NULL,f,NULL);
pthread_create(&q,NULL,f,NULL);
pthread_join(t, NULL);
pthread_join(q, NULL);
<Ponto P>
```

20) Quais as razões **fundamentais** para a existência de duas APIs (por ex. para acesso a ficheiros), sendo uma a da biblioteca da linguagem C (abrev. **libC**), e outra a do núcleo (*kernel*) do SO (abrev. ***syscall***)?

a) A **libC** oferece uma API simples, e portabilidade da linguagem para diferentes SOs; a ***syscall*** oferece os mesmos serviços da **libC**, mas de forma mais eficiente, embora a API seja mais difícil de usar
b) A **libC** oferece uma API simples, com mais funcionalidades que a ***syscall*** (ex., **`printf`**), e portabilidade da linguagem para diferentes SOs; a ***syscall*** apenas interessa a programadores do *kernel*
c) A **libC** oferece uma API simples, com mais funcionalidades que a ***syscall*** (ex., **`printf`**), e portabilidade da linguagem para diferentes SOs; a ***syscall*** apenas interessa para tirar partido do *hardware* (ex., vários CPUs)
d) A **libC** garante a portabilidade da linguagem para diferentes SOs; a ***syscall*** oferece todos os serviços do SO, alguns dos quais podem não estar disponíveis na **libC**
e) Nenhuma das anteriores

Fundamentos de Sistemas de Operação
1º Teste, 22 de Outubro de 2022

QUESTÕES DE DESENVOLVIMENTO — VERSÃO A

D1) Complete o programa abaixo, uma versão Produtor/Consumidor com semáforos, que deve funcionar correctamente para 1 *thread* produtor e 1 *thread* consumidor a executarem-se concorrentemente.

```
#define N       10
char bufCir[N];

sem_t slotsCheios, slotsVazios; // contadores de slots vazios e cheios no vector (buffer circular)
int c=0, p= 0; // índices que indicam as próximas posições a consumir e produzir

void putInBufCir(char chr) { bufCir[p]= chr; p= (p+1)%___; } // manter índice actualizado
void getFromBufCir(char *chr) { *chr= bufCir[c]; c= (c+1)%___; } // idem...

void produz(char c) {
    _________(&slotsVazios); //avançar se possível
      putInBufCir(c);
    _________(&slotsCheios); //actualizar a contagem
}

void *produtor(void * args) {
  for (int i= 0; i < 24; i++) produz('A'+i);
  return NULL;
}

void consome(char *chr) {
    _________(&slotsCheios); //avançar se possível
      getFromBufCir(chr);
    _________(&slotsVazios); //actualizar a contagem
}

void *consumidor(void * args) {
  char chr;
  for (int i= 0; i < 24; i++)
    { consome(&chr); printf("%c ",chr); fflush(stdout); }
  return NULL;
}

int main(int argc, char *argv[]) {
  pthread_t p1, p2;

  _____________________________; _____________________________; //inicializar os semáforos

  pthread_create(&___, NULL, ____________, NULL); //lançar um thread
  pthread_create(&___, NULL, ____________, NULL); //lançar o outro thread

  _____________________________;  _____________________________; //esperar o fim dos threads

  return 0;
}
```

D2) Pretende-se implementar num programa um par de processos produtor/consumidor. O processo pai recorre à função **`produtor()`** para produzir uma sequência de caracteres; o processo filho usa a função **`consumidor()`** para "receber" os caracteres produzidos e afixa-os no ecrã. Os dois processos comunicam através de um *pipe*.

Complete o programa abaixo de forma a obter a funcionalidade acima descrita, tendo em atenção que o programa deve somente terminar quando o consumidor tiver afixado no ecrã tudo o que foi produzido.

```
// Assuma todos os includes necessários

void produtor(int fd) {
  char c;
  for (int i= 0; i < 24; i++) { c='A'+i; write(fd, &c, 1); }
}

void consumidor(int fd) {
  char c;
  while ( read(fd, &c, 1) > 0 ) { printf("%c", c); }
}

int main(int argc, char *argv[]) {
// Declare as variáveis de que necessita
// Crie o pipe e o(s) processo(s) necessário(s) e execute num deles
// a função produtor() e no outro a função consumidor().
// Garanta que termina o(s) processo(s) após a comunicação terminar
```

```
}
```

D3) Implemente a função **`run_program`** que executa um programa, cujo nome é fornecido através do parâmetro **`prog`**; contudo, quando o **`prog`** for executado o seu *standard error* (**`stderr`**, *file descriptor* nº 2) não deverá estar "associado", como é habitual, ao ecrã, mas será redireccionado para o ficheiro cujo nome é indicado em **`error_file`**. O programa a executar não tem argumentos e deve ser executado no contexto de um novo processo, criado para o efeito. Além disso, a função deve esperar pela conclusão da execução de **`prog`**, retornando o valor 0 se a execução foi desencadeada com sucesso e -1 no caso contrário.

```
// Assuma todos os includes necessários
int run_program (char* prog, char* error_file)
{




}
```

Protótipos de funções, declarações de tipos, e outras informações úteis
(nota: simplificadas de acordo com a forma de utilização em FSO)

```
int fork()
int wait(int *status)
int exit(int status)

int execlp(char *program, char *arg0, char *arg1, ...)
int execvp(char *program, char *const argv[])


pthread_mutex_t mut
pthread_mutex_t mut=PTHREAD_MUTEX_INITIALIZER
pthread_mutex_lock(pthread_mutex_t *mut)
pthread_mutex_unlock(pthread_mutex_t *mut)
pthread_create(pthread_t *thr, NULL, void *(*func)(void *), void *arg);
pthread_join(pthread_t *thr, void **ret)


sem_t var
sem_init(sem_t *var, 0, value)
sem_post(sem_t *var)
sem_wait(sem_t *var)


int open(char *path, int flags, [int mode])
int close(int fd)
int pipe(int pipefd[2])
int dup(int oldfd)
int read(int fd, void *buf, int nbytes)
int write(int fd, void *buf, int nbytes)
```