Número: Nome:

LEIC/LETI – 2020/21 – 1º Teste de Sistemas Operativos

21 de novembro de 2020, Duração: 1h00m

- **Responda no enunciado, apenas no espaço fornecido. Identifique todas as folhas.**
- **Por omissão, os excertos de código omitem o tratamento de erros; nas suas respostas com código, pode também omitir o tratamento de erros.**
- Nas perguntas de escolha múltipla, **uma resposta errada desconta ½ ou ¼ da cotação consoante haja 2 ou 4 opções de escolha, respetivamente.**

Grupo I [Programação com processos, 7v]

Pretende-se desenvolver um programa que:

i) Recebe os *pathnames* de dois ficheiros executáveis (como argumentos de linha de comandos);

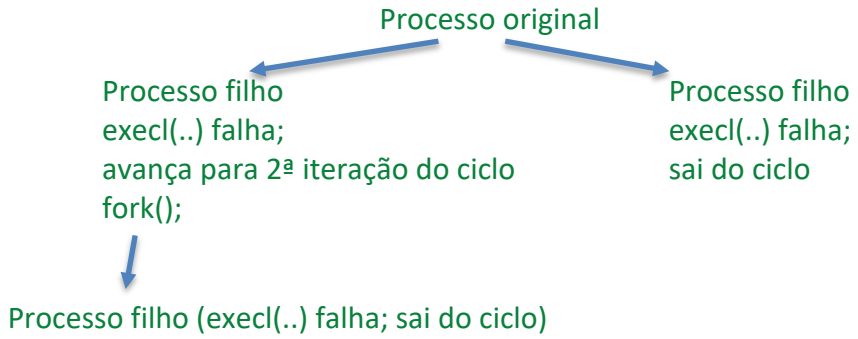
ii) Executa ambos em paralelo (sem argumentos adicionais de linha de comando), cada um num novo processo filho; e

iii) Imprime no ecrã uma mensagem indicando qual deles terminou mais cedo.

1. [3v] **Complete o esqueleto seguinte** de forma a construir o programa desejado:
   - Não se esqueça de declarar as variáveis adicionais que sejam necessárias e preencher os espaços a sombreado.
   - Omita o tratamento de retornos de erro em todas as chamadas às funções sistema (ou seja, assuma que todas as chamadas sistema têm sucesso).
   - Só precisa apresentar o código até ao passo iii) acima (pode omitir o restante código).

```
int main(int argc, char **argv) {
  int p;
  int pid[2];

  if (argc < 2) return 1;



  for (int i=0; i<2; i++) {

    char *path = argv[i+1];

    p = fork();

    if (p == 0)
      execl(path, path, 0);

    else
      pid[i] = p;

  }

  p = wait(NULL);

  if (p==pid[0])
    printf("O primeiro programa terminou mais cedo");
  else
    printf("O segundo programa terminou mais cedo");


}
```

2. [1v] Assuma que, contrariando o pressuposto de que "*todas as chamadas sistema têm sucesso*", eram passados dois *pathnames* errados e, consequentemente, as chamadas à função *execl* falhavam.
Quantos processos filho seriam criados no total? Responda sob a forma de uma árvore de processos (cuja raíz corresponde ao processo original).



Nota: consoante a solução dada na alínea anterior, diferentes respostas foram aceites como corretas nesta alínea.

3. [1v] Assuma que experimentava construir uma alternativa do programa anterior em que cada executável corria numa nova tarefa (em vez de num processo filho). Teria o desfecho esperado? Se sim, indique uma vantagem dessa alternativa. Se não, justifique sucintamente.

Não. Como múltiplas tarefas do processo partilham obrigatoriamente o mesmo programa, não seria possível colocar duas tarefas (do mesmo processo) a correr dois executáveis distintos.
(Sugestão: experimentar construir programa com duas tarefas que chamam execl e conferir o que acontece assim que uma das tarefas chama execl.)

4. O utilizador *Alice*, que pertence ao grupo *alunos*, executou o programa da alínea 1 passando os seguintes *pathnames* como argumentos: ./exe1 ./exe2

Na diretoria corrente (na qual o utilizador *Alice* executou o programa), a listagem usando `ls -l` revela:

```
-rwx--x--- 1 Bob      alunos    20 set 13 18:16  exe1
-rwxr-x--x 1 Charlie  docentes  24 oct 13 10:06  exe2
```

a. O programa terá sucesso ao tentar executar (através de *execl)* ambos os executáveis?

[0,5v] Primeiro *pathname:* Sim **X** Não ☐
[0,5v] Segundo *pathname:* Sim **X** Não ☐

b. [1v] Assumindo que o programa tem sucesso ao tentar executar ambos, qual será o UID efetivo dos processos filho após cada chamada a *execl*?

Primeiro processo filho:	**Alice**
Segundo processo filho:	**Alice**

Grupo II [Tarefas e sincronização em memória partilhada, 13v]

1. Considere o seguinte programa com múltiplas tarefas.

```
void *threadFn(void *arg) {
 int *id = (int*) arg;
 printf("Ola (tid:%d)! ", *id);
 ...
}

int main (void) {
 int tid;
 pthread_t pthreads[N];

 for (tid=1; tid<=N; tid++) {
   pthread_create (&pthreads[N], 0, threadFn, &tid);
 }
 ...
}
```

Executou-se várias vezes o programa acima com N=4. Para cada um dos *outputs* seguintes, indique se ele pode ser por vezes observado.

a. [1v] "`Ola (tid:1)! Ola (tid:3)! Ola (tid:4)! Ola (tid:2)!`"
Sim **X** Não ☐

b. [1v] "`Ola (tid:1)! Ola (tid:3)! Ola (tid:3)! Ola (tid:4)!`"
Sim **X** Não ☐

Notas:

- Nas primeiras especificações da função *pthread_create* (conforme o *standard POSIX*), o 1º argumento era opcional; ou seja, podia ser passado NULL caso o programador não pretendesse obter o objeto *pthread_t* referente à nova tarefa criada.
  Como o 1º argumento da função é irrelevante para o exercício em causa, por simplicidade o enunciado original adotava essa forma (i.e., *pthread_create(NULL, ...)*).
  No entanto, versões recentes do *standard* tornaram o 1º argumento obrigatório. Ou seja, caso o programador (erradamente) passe NULL, as implementações atuais da função causam um *segmentation fault*. Nesse caso, o programa não terá nenhum *output*.
  Tendo isto em conta, considerou-se igualmente correto:
  - Quem não reparou no problema do 1º argumento a NULL e respondeu "sim" na alínea a.
  - Quem, sabendo da obrigação do 1º argumento ser "não NULL", respondeu "não" a ambas as alíneas.
- O enunciado original, por lapso, tinha "`Ola (tid:1)! Ola (tid:2)! Ola (tid:2)! Ola (tid:4)!`" na alínea b. Nesta formulação, a alínea tinha um grau de dificuldade superior ao esperado na disciplina de SO. Por isso decidimos anular esta alínea. Em alternativa, a sua cotação foi transferida para alínea a, que passou a valer 2v

2. Considere o seguinte programa, que oferece diferentes operações sobre um registo que pode conter um valor inteiro. Inicialmente, o registo está vazio.

```
int registo;
int preenchido = 0;

int preencher(int v) {
  if (preenchido)
    return ERRO;
  registo = v;
  preenchido = 1;
  return OK;
}
```

```
int retirar() {
  int v;
  if (!preenchido)
    return ERRO;
  v = registo;
  preenchido = 0;
  return v;
}

int consultar() {
  int v;
  if (!preenchido)
    return ERRO;
  v = registo;
  return v;
}
```

a) Construiu-se um programa paralelo, no qual múltiplas tarefas chamam as funções implementadas acima para aceder ao registo partilhado. Dos seguintes comportamentos inesperados, indique quais se poderão observar:

1) [0,5v] Um valor foi preenchido no registo partilhado mas perdeu-se (i.e., desapareceu sem alguma vez ter sido retornado pela função *retirar*).
Sim **X** Não ☐

2) [0,5v] Um valor preenchido foi retirado em duplicado por tarefas diferentes.
Sim **X** Não ☐

b) [4v] Este programa claramente tem secções críticas que estão por proteger. Escolha entre trincos lógicos (*mutexes*) ou trincos de leitura-escrita, tendo em conta que a operação *consultar* é executada 90% das vezes (sendo as funções *preencher* ou *retirar* chamadas nas restantes 10% das vezes).
Indique aquilo que modificaria para corrigir este lapso.

```
int registo;
int preenchido = 0;

//Declarar aqui o(s) trinco(s)
rwlock_t lock;

int preencher(int v) {
  wrlock(lock);
  if (preenchido) {
    unlock(lock);
    return ERRO;
  }
  registo = v;
  preenchido = 1;
  unlock(lock);
  return OK;
}
```

```
int retirar() {
  int v;
  wrlock(lock);
  if (!preenchido) {
    unlock(lock);
    return ERRO;
  }
  v = registo;
  preenchido = 0;
  unlock(lock);
  return v;
}

int consultar() {
  int v;
  rdlock(lock);
  if (!preenchido) {
    unlock(lock);
    return ERRO;
  }
  v = registo;
  unlock(lock);
  return v;
}
```

c) [4v] Estenda agora o programa **inicial** para que as funções *preencher* e *consultar* passem a ter um comportamento bloqueante. Ou seja, enquanto não houver condições para preencher (registo já preenchido) ou consultar (registo vazio), estas funções devem esperar.
Note:

- **Recorra exclusivamente a mutexes e/ou a variáveis de condição**.
- Por simplicidade, a **função *retirar* deve manter o comportamento não bloqueante** (ou seja, retorna ERRO caso o registo esteja vazio).

```
int registo;
int preenchido = 0;

//Declarar aqui mutexes/vars.
condição/etc.
mutex_t m;
cond_t podePreencher, podeConsultar;

int preencher(int v) {

  lock(m);
  while (preenchido)
    wait(podePreencher, m);

  registo = v;
  preenchido = 1;

  broadcast (podeConsultar);
  unlock(m);

  return OK;
}
```

```
int retirar() {
  int v;

  lock(m);
  if (!preenchido) {
    unlock(m);
    return ERRO;
  }
  v = registo;
  preenchido = 0;
  signal (podePreencher);
  unlock(m);

  return v;
}

int consultar() {
  int v;

  lock(m);
  while (!preenchido)
    wait(podeConsultar, m);

  v = registo;
  unlock(m);

  return v;
}
```

3. Considere as seguintes diferentes implementações concorrentes das funções que controlam o acesso a um parque de estacionamento. Estas implementações podem sofrer de incorreções ou ineficiências importantes. Chamamos a atenção para as porções destacadas no código.

Versão A	Versão B

`int vagas = N;`
`mutex m; cond c;`
 	
`void entrar() {`
`  lock(m);`
`  if (vagas == 0)`
`  wait(c, m);`
`  vagas --;`
`  unlock(m);`
`}`
 	
`void sair() {`
`  lock(m);`
`  vagas ++;`
`  signal(c);`
`  unlock(m);`
`}`
	`int vagas = N; mutex m;`
	
	`void entrar() {`
	`  do {`
	`  lock(m);`
	`  if (vagas>0) break;`
	`  else unlock(m);`
	`  } while (1);`
	`  vagas --;`
	`  unlock(m);`
	`}`
	
	`void sair() {`
	`  lock(m);`
	`  vagas ++;`
	`  unlock(m);`
	`}`

a. [1v] Executou-se cada versão dentro de um programa com muitas tarefas (que chamam as funções *entrar* e *sair*). Usado a ferramenta *time*, uma das versões acima consome um tempo total de CPU **muito superior** à outra. Qual?

1) Versão A
2) Versão B

B

b. [1v] Um requisito desejável é assegurar que o número de carros simultaneamente dentro do parque nunca excede a lotação máxima (N).
Qual das versões **não garante** este invariante?

1) Versão A
2) Versão B
3) Todas garantem o invariante
4) Nenhuma garante o invariante

A