Número: Nome:

LEIC/LETI – 2019/20 – 1º Teste de Sistemas Operativos

23 de novembro de 2019, Duração: 1h30m

- **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, 5v]

Considere a seguinte implementação simplificada de uma *shell* que executa comandos em *background*.

```
int main() {
  char **command;
  int n = 0;
  while (TRUE){
    //Função auxiliar que lê o próximo comando + respetivos argumentos
    //e os retorna em vetor de strings
    command = read_command();
    if (strcmp(command[0],"exit")==0)
      terminateShell(n);
    if (fork () == 0)
      execv (command[0], command);
    n ++;
    //Função auxiliar que liberta o vetor de strings
    free_command(command);
}
```

1. [0,9v] Ao receber o comando "exit" a *shell* chama a função *terminateShell,* que espera pela terminação de todos os *n* comandos que foram lançados em *background* e logo de seguida termina o processo da *shell*. Implemente esta função.

```
void terminateShell(int n) {
  for (int i=0; i<n; i++)
    wait(NULL);
  exit(EXIT_SUCCESS);
}
```

2. [2v] Assuma que existe um novo comando "fg", que coloca a *shell* à espera até que o anterior comando lançado (em *background*) tenha terminado. Só depois disso é que a *shell* volta a receber e a lançar novos comandos. Se chamado *k* vezes consecutivas, a *shell* deve esperar pelos últimos *k* comandos lançados.
   Apresente abaixo uma extensão ao programa inicial que implemente este novo comando.
   Na sua solução:
   - Pode assumir que, ao longo da execução da *shell* são lançados, no máximo, MAX comandos.
   - Recorra à função sistema *waitpid*, que é uma variante da função *wait* definida da seguinte forma (excerto das *man pages*):

> *pid_t waitpid(pid_t pid, int *status, int options);*
> *The waitpid() system call suspends execution of the calling process until a child specified by pid argument has changed state.*

Nota: No argumento options, passe simplesmente 0 (ou seja, nenhuma opção avançada).

```
int main() {
  char **command;
  int n = 0;

  pid_t pid[MAX];

  while (TRUE){

    command = read_command();
    if (strcmp(command[0],"exit")==0)
      terminateShell(n);

    if (strcmp(command[0],"fg")==0) {
      if (n > 0) {
        waitpid(pid[n-1], NULL, 0);
        n--;
      }
    }
    else {
      pid[n]=fork();
      if (pid[n] == 0)
        execv (command[0], command);
      n ++;
    }
    free_command(command);
  }
}
```

3. [1,2v] Numa dada diretoria existem os seguintes executáveis:

```
-rwx------ 1 bruno alunos       20 set 13 18:16  f1
-rwxr-x--- 1 paulo funcionarios 24 oct 13 10:06  f2
-rwxr-x--- 1 ana   alunos       01 set 13 19:00  f3
-r-x--x--x 1 joana docentes     03 set 13 15:20  f4
```

Considere que o utilizador *bruno*, pertencente ao grupo *alunos*, executa a *shell* nesta diretoria e tenta executar os ficheiros listados acima. Para cada caso, indique se a chamada tem sucesso (riscar o que não interessa).

a) Ficheiro f1: [sucesso] ~~[erro]~~
b) Ficheiro f2: ~~[sucesso]~~ [erro]
c) Ficheiro f3: [sucesso] ~~[erro]~~
d) Ficheiro f4: [sucesso] ~~[erro]~~

4. [0,9v] Um programador inexperiente decidiu implementar uma alternativa à *shell* apresentada no início deste grupo que corre cada comando numa nova tarefa (em vez de um novo processo filho).

```
void *executaComando(char **args) {
  execv (args[0], args);
}

int main() {
[...]
  command = read_command();
  pthread_create(0, 0, executaComando, command);
[...]
}
```

Ao testar o programa acima com uma sequência de vários comandos, a *shell* já não se comporta como esperado. Descreva sucintamente o novo comportamento (errado) do programa acima.
Na sua resposta, tenha em conta o seguinte excerto das *man pages* das funções *exec**:

> *Executes the program pointed to by filename. [...] All threads other than the calling thread are destroyed during an exec*().*

Resposta:

Esta *shell* apenas executa o primeiro comando inserido (assumindo que não é 'exit' nem 'fg'), não aceitando mais nenhum comando. O processo da *shell* termina quando o programa lançado terminar.

Grupo II [Comunicação entre processos, 6v]

Considere que um servidor pretende disponibilizar aos clientes a possibilidade de ser invocado quer através de um *socket* em modo *Datagram* ou em modo *Stream*.
Considere que, numa rotina de inicialização, o programa servidor já criou os dois *sockets* (cujos descritores estão atribuídos nas variáveis *dgrmfd* e *strmfd,* respetivamente) e atribuiu-lhes nomes. No caso do *socket stream,* também já efectuou a chamada a *listen*.

1) Considerando o *socket datagram (dgrmfd).*
   a) [0,6v] Para o servidor poder receber o próximo pedido recebido pelo *socket*, em teoria é possível usar as funções *read* ou *recvfrom*. No entanto, a grande maioria recorre à *recvfrom.* Porquê?
      i. Tem melhor desempenho.
      ii. É a única que permite saber quem foi o emissor do pedido.
      iii. Recebe menos argumentos.
      iv. É a única que não é bloqueante.

      ii

2) Considerando agora o *socket stream (strmfd).*
   a) [0,6v] Suponha que pretende que três clientes enviam comandos utilizando ligações com esse *socket*. Escolha a mais adequada:
      i. Os clientes enviam as mensagens utilizando *write* para o *socket* inicial do servidor (*strmfd*).
      ii. Os clientes utilizam *sendto* para o *socket* inicial do servidor (*strmfd*).
      iii. Os clientes têm de efetuar inicialmente uma ligação usando a função *connect*, assim criando um novo *socket* no cliente que fica ligado ao servidor.
      iv. Os clientes têm de efetuar inicialmente uma ligação usando a função *connect*, assim criando um novo *socket* no servidor que fica ligado ao do cliente.

      iv
   b) [0,8v] Entre as características seguintes, assinale as que comuns e as que são distintas entre *sockets stream* e a *named pipes*.
      i. Em ambos, a função *read* bloqueia quando não há dados para receber.
      ii. Ambos são canais bidirecionais.
      iii. Quando usados no contexto de uma máquina apenas, ambos usam nomes de ficheiros como nomes.
      iv. Ambos podem ser usados também entre processos em máquinas distintas, desde que ligadas em rede.

Comuns: **i, iii** Distintas: **ii, iv**

3) Embora as perguntas anteriores se tenham focado em cada *socket* isoladamente, o servidor pretende ser capaz de ser invocado a partir de ambas as vias. Para tal, o programa do servidor tem o seguinte bloco de código (intencionalmente incompleto), que define a máscara do *select* e efetuar a chamada a esta *system call*, dentro de um ciclo infinito de execução.

```
int newfd = -1;
char buffer[DIM];
fd_set testmask, mask;

FD_ZERO(&testmask);
FD_SET(strmfd,&testmask);
FD_SET(dgrmfd,&testmask);
for(;;) {
   mask = testmask;
   s= select(MAXSOCKS,&mask,0,0,0);
   if (FD_ISSET(dgrmfd, &mask)) {
     /* to be completed (a) */
   }
   if (FD_ISSET(strmfd, &mask)) {
     newfd = accept(strmfd,(struct sockaddr*)&clientaddr, &clientlen);
     /* to be completed (b) */
   }
   /* to be completed (c) */
}
```

a. [1v] Se for indicado que há atividade no *socket Datagram*, o programa deve colocar os dados disponíveis na variável *buffer*. Proponha o código para substituir o comentário *"to be completed (a)"*.

```
recvfrom(dgrmfd, buffer, DIM, 0, (struct sockaddr*)&clientaddr, &clientlen);
```

Suponha que, na sequência do *select*, o servidor recebeu um pedido de ligação e que a aceitou. Considere que se pretendia que o processo servidor também atendesse esse novo cliente **sem criação de novo processo ou tarefa**.

b. [1,5v] Pretende-se que, na próxima iteração do ciclo, a chamada a *select* também retorne caso cheguem dados a partir dessa nova ligação. Apresente as instruções que colocaria no local onde está o comentário *"to be completed (b)"* para cumprir este requisito.

```
FD_SET(newfd, &testmask);
```

c. [1,5v] Complementando a alínea anterior, pretende-se que, se na próxima iteração do ciclo houver dados para ler enviados pela nova ligação, estes sejam lidos para a variável *buffer.* Apresente o código que implementa este requisito, que deverá substituir o comentário *"to be completed (c)".*

```
if (FD_ISSET(newfd, &mask))
  read(newfd, buffer, DIM);
```

Grupo III [Sincronização em memória partilhada, 9v]

Considere o programa seguinte:

```
1.  #include <pthread.h>
2.  #define Nthreads 5
3.
4.  int numTerminatedThreads = 0;
5.  int sleepTime[] = {6,5,4,5,3};
6.  pthread_t tid[Nthreads];
7.
8.  static void *threadFunc(void *arg)
9.  {
10.     int idx = (int) arg;
11.     sleep(sleepTime[idx]);       /* Simula a execução de algum trabalho */
12.     numTerminatedThreads ++;
13.     printf("Thread %d terminating (#Terminated=%d)\n", idx, numTerminatedThreads);
14.     return NULL;
15. }
16.
17. int main()
18. {
19.     int idx;
20.
21.     /* Create all threads */
22.     for (idx = 0; idx < Nthreads; idx++)
23.         pthread_create(&tid[idx], NULL, threadFunc, idx));
24.
25.     for (idx = 0; idx < Nthreads; idx++) {
26.         pthread_join(tid[idx], NULL);
27.         printf("Reaped thread %d\n", idx);
28.         processReapedThread(idx); /* Função muito demorada */
29.     }
30.     exit(EXIT_SUCCESS);
31. }
```

Executando o programa acima, obteve-se o seguinte *output*:

```
1.  Thread 4 terminating (#Terminated=1)
2.  Thread 2 terminating (#Terminated=2)
3.  Thread 3 terminating (#Terminated=3)
4.  Thread 1 terminating (#Terminated=4)
5.  Thread 0 terminating (#Terminated=5)
6.  Reaped thread 0
7.  Reaped thread 1
8.  Reaped thread 2
9.  Reaped thread 3
10. Reaped thread 4
```

1) Explique o output do programa
   a) [0,7v] Porque é que as linhas 1 e 2 aparecem na ordem acima?

Porque, embora a tarefa 2 tenha sido criada sensivelmente antes da tarefa 4, a tarefa 2 bloqueou-se durante um período mais longo que a tarefa 4 (4 segundos vs. 3 segundos, resp.).

   b) [0,7v] As linhas 3 e 4 aparecem por uma ordem diferente da ordem pela qual as tarefas *(threads)* respetivas foram criadas. Acha esta inversão estranha ou é justificável? Justifique.

É justificável. Embora a tarefa 1 seja criada sensivelmente antes da tarefa 3, ambas se bloqueiam por um período igual (5 segundos). Caso o escalonamento do SO decida dar execução mais cedo à tarefa 3, a ordem de terminação pode ser diferente da ordem de criação.

c) [0,7v] Porque é que cada linha "Reaped thread ..." não aparece imediatamente após a linha impressa pela tarefa em causa?

Porque essa linha ("Reaped thread ...") só é impressa depois das tarefas com *idx* anterior à tarefa em causa terem terminado.

2) [1,5v] Ao contrário do que acontece no output acima, é possível que duas tarefas distintas imprimam o mesmo valor em "`#Terminated=...`"? Se sim, proponha uma alteração ao programa para que este *bug* deixe de se verificar (apresente apenas as linhas modificadas). Se não, justifique.

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

static void *threadFunc(void *arg)
{
  int idx = (int) arg;
  int numTerminatedAux;
  sleep(sleepTime[idx]);      /* Simula a execução de algum trabalho */
  pthread_mutex_lock(&m);
  numTerminatedAux = (++numTerminatedThreads);
  pthread_mutex_unlock(&m);
  printf("Thread %d terminating (#Terminated=%d)\n", idx, numTerminatedAux);
  return NULL;
}
```

3) Sobre a criação das tarefas (linhas 20-21):

a) [0,8v] Explique o significado dos parâmetros da função `pthread_create` (linha 23). Na sua resposta, junto a cada parâmetro indique se é um parâmetro de entrada ("in"), saída ("out") ou entrada/saída ("in-out")?

Arg. 1: **Identificador da nova tarefa criada (out)**
Arg. 2: **Atributos avançados da nova tarefa (in)**
Arg. 3: **Função que a nova tarefa executará (in)**
Arg. 4: **Argumento a passar à nova tarefa (in)**

b) [0,7v] Suponha que, algum tempo depois de todas as tarefas terem sido criadas, a primeira dessas tarefas (idx=0) pretende aceder à variável *tid* e lá consultar os identificadores das tarefas que foram criadas **após** a criação desta tarefa.
A tarefa tem acesso à variável *tid* e ao seu conteúdo mais recente? Justifique.

Sim, pois as tarefas do mesmo processo partilham as variáveis globais.

c) [0,7v] Compare esta situação com a que existiria se, em vez de se criarem tarefas, se criassem processos filho. A resposta à alínea acima (relativa aos processos filho) seria idêntica? Justifique.

Não. Embora o espaço de endereçamento do processo filho comece como uma cópia do do pai, ambos são distintos e isolados. Ou seja, a variável *tid* no processo pai não é partilhada com a variável *tid* no processo filho, logo alterações feitas pelo pai à sua *tid* após a criação do filho não são refletidas na *tid* do filho.

4) [3,2v] Após descobrir que cada tarefa nova terminou, a tarefa inicial chama a função *processReapedThread*, que é muito demorada. No entanto, o programa implementado é muito ineficiente pois só executa *processReapedThread* para uma dada tarefa *k* após a tarefa *k-1* ter terminado e sido processada (mesmo que *k* termine muito antes de *k*-1),
Proponha uma alternativa ao programa acima que implemente a seguinte otimização: as chamadas às funções *pthread_join* e *processReapedThread* para a tarefa *k* devem ocorrer **imediatamente** após essa tarefa terminar (mesmo que as tarefas criadas antes de *k* não tenham ainda terminado).

Sugestões:
- Mantenha um vetor em que cada entrada indica qual o estado de cada tarefa: após ser criada está *alive*; quando termina passa a *terminated;* quando a tarefa inicial executa *pthread_join* sobre a tarefa previamente terminada, esta passa a *joined.* Este vetor já se encontra declarado e inicialmente preenchido abaixo.
- Recorra a uma variável de condição para implementar a espera (da tarefa que corre a função *main*) até que a próxima tarefa termine. Essa variável (chamada *cond*) já se encontra declarada e inicializada abaixo.

```
#include <pthread.h>
#define Nthreads 5

int numTerminatedThreads = 0;
int sleepTime[] = {6,5,4,5,3};
thread_t tid[Nthreads];

enum tstate {                   /* Thread states */
    TS_ALIVE,                   /* Thread is alive */
    TS_TERMINATED,              /* Thread terminated, not yet joined */
    TS_JOINED                   /* Thread terminated, and joined */
};
enum tstate state[Nthreads];

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

/* Declare e inicialize aqui variáveis adicionais que precise */

pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
int numJoinedThreads = 0;
```

```
static void *threadFunc(void *arg) {

  /* ... */

  pthread_mutex_lock(&m);
  state[idx] = TERMINATED;
  numTerminatedThreads ++;
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&m);

  printf(...);
  return NULL;

}
```

```
int main() {
  int idx;

  for (idx = 0; idx < Nthreads; idx++) {
    state[idx] = TS_ALIVE;
    pthread_create(&tid[idx], NULL, threadFunc, idx));
  }

  pthread_mutex_lock(&m);
  while (numJoinedThreads < Nthreads) {
    while (numTerminatedThreads == 0)
      pthread_cond_wait(&cond, &m);
    for (idx = 0; idx < Nthreads; idx++) {
      if (state[idx] = TERMINATED) {
        pthread_join(tid[idx], NULL);
        processReapedThread(idx);
        state[idx] = JOINED;
        numJoinedThreads ++;
        numTerminatedThreads --;
      }
    }
  }
  pthread_mutex_unlock(&m);

  exit(EXIT_SUCCESS);
}
```