Número: Nome:

LEIC/LETI – 2017/18 - 1º Teste de Sistemas Operativos

21 de Novembro de 2018

Responda no enunciado, apenas no espaço fornecido. Identifique todas as folhas.

Duração: 1h30m

Grupo I [5 Val]

1. [1 val] Considere o programa *whoami,* que as *man pages* descrevem como: "`Print the user name associated with the current effective user ID.`"
   Suponha que, ao correr comando *ls –l whoami*, se obtinha o seguinte output:

```
-rwxr-xr-x  1 root  root  19168 Sep  9  2014 whoami
```

   e que, logo de seguida, o dono do ficheiro acima ativava o respetivo setUID bit, usando o comando:

```
chmod u+s whoami
```

   Quais serão os outputs obtidos se o utilizador "xpto" executar o comando *whoami*, (i) antes e (ii) depois de se executar o comando chmod mencionado acima?

Antes:
xpto
Depois:
root

2. Considere o seguinte programa:

```
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <stdlib.h>

4 int i=0;

5 int main() {
6   int pid, status;
7   i=1;
8   pid = fork();
9   if (pid == 0) {
10      i=i*2;
11      printf("A:%d\n",i);
12      execl("/usr/bin/whoami","/usr/bin/whoami",NULL);
13      printf("AA:%d\n",++i);
14      exit(i);
15  }
16  else if (pid != -1){
17    sleep(1);
18    i=i*3;
19    printf("B:%d\n",i);
20    wait(&status);
21    if (WIFEXITED(status))
22          printf("BB:%d\n", WEXITSTATUS(status));
23    exit (0);
24  } else {
25    printf ("erro no fork\n");
26    exit (-1);
27  }
28 }
```

a. [1 val] Quantos processos são criados por este programa?

E' criado um processo filho na linha 8. Claramente, também é criado um processo para suportar a execução do programa.

b. [1 val] Supondo que as chamadas às funções *fork* (linha 8) e *execl* (linha 12) são bem sucedidas, e que o *whoami* termina com sucesso (i.e., *exit(0)*), qual será o *output* produzido pelo programa?

A:2
Nome do utilizador//output do whoami
B:3
BB:0

c. [1 val] Supondo que a chamada a *execl* na linha 12 falha, qual será o *output* produzido pelo programa?

A:2
AA:3
B:3
BB:3

d. [1 val] Durante a execução do programa acima, é possível que temporariamente exista um processo *zombie*. Ilustre indicando em que momento surge o processo *zombie* e em que momento ele muda para outro estado.

O processo filho entra no estado *zombie* ao terminar, e fica neste estado até o processo pai conseguir executar a chamada sistema *wait*.

Grupo II [9 Val]

Imagine que o IST decide pôr em prática um novo modelo de funcionamento do *shuttle* entre Alameda e Taguspark.

- Neste novo modelo, existem *N* terminais (com teclado e ecrã) espalhados por cada campus de onde parte do *shuttle*. Cada pessoa interessada em deslocar-se no shuttle deve encontrar um dos terminais e, através do mesmo, pode reservar até 4 lugares que ainda estejam livres.
- Neste novo modelo, o *shuttle* deixa de ter horário fixo. Em vez disso, o *shuttle* fica estacionado até todos os lugares estarem reservados. Quando tal acontece, um terminal (com ecrã apenas) colocado junto ao acesso para o *shuttle* anuncia que o mesmo está pronto a partir dentro de 10 minutos, devendo os passageiros com reservas entrar e ocupar os seus lugares.

Pediu-se a um programador *que não frequentou a disciplina de Sistemas Operativos* para implementar o novo modelo num programa com múltiplas tarefas (*N* tarefas, cada uma a interagir com cada terminal; e uma tarefa que transmite o anúncio final quando o *shuttle* está lotado e pronto a partir). Esse programa encontra-se listado abaixo, com as seguintes simplificações:

i) o tratamento de retorno de erros é omitido no código abaixo e pode também ser omitido nas suas respostas;
ii) só considera a partida de *um shuttle* (omite-se o tratamento dos *shuttles* seguintes);
iii) os pedidos de reserva são sempre de 4 lugares (e não menos que isso).

```
#define D ... //Lotação do shuttle
#define MAXSTR ... //Dim. da string

/* Representa os canais de input e/ou output de um
terminal do sistema */
typedef struct {
  FILE *in;
  FILE *out;
} terminal_t;


/* Variáveis partilhadas */
int mapaAtual[D] = {LIVRE, LIVRE, .., LIVRE};
int numVagas = D;



void *fnAnuncio (void *arg) {
  FILE* outfd = ((terminal_t *) arg)->out;

  while (TRUE) {
    while (numVagas > 0);
    fprintf(outfd, "Shuttle parte em 10min\n");

    /* Prepara próximo shuttle */
    /* ... [código omitido] ... */
  }
}
```

```
void *fnTerminalReserva(void *arg) {
  FILE* infd = ((terminal_t *) arg)->in;
  FILE* outfd = ((terminal_t *) arg)->out;

  char mapaStr[MAXSTR];
  int paraReservar[4];

  while (TRUE) {
    /* Lê conteúdo do mapa de reservas e
      converte-o para a string mapaStr */
    lerMapa(mapaStr);
    fprintf(outfd, "Escolha 4 lugares:\n");
    fprintf(outfd, "%s", mapaStr);

    /* Recebe de inputfd os lugares a reservar e
      preenche-os em paraReservar */
    lerPedidosDeReserva(inputfd, paraReservar);

    int sucesso = TRUE;
    int i;

    //1. Verifica se todos os lugares estão livres
    for (i=0; i<4; i++) {
      if (mapaAtual[paraReservar[i]] == OCUPADO) {
        sucesso = FALSE; break;
      }
    }
    //2. Se sim, reserva todos os lugares
    if (sucesso == TRUE) {
      for (i=0; i<4; i++) {
        mapaAtual[paraReservar[i]] = OCUPADO;
      }
    }

    if (sucesso == TRUE) {
      numVagas -= 4;
      fprintf(outfd, "Lugares reservados!\n");
    }
    else fprintf(outfd, "Tente de novo.\n");
  }
}
```

Número: _____ Nome: _____

1. [2 val] Complete a excerto da função *main* abaixo, que cria as tarefas necessárias.

```
/* === Funções auxiliares === */
/* Abre o canal de input do terminal indicado em argumento */
FILE *abrirTerminalIn(int);
/* Abre o canal de output do terminal indicado em argumento */
FILE *abrirTerminalOut(int);


int main () {
  pthread_t tid[N+1];

  terminal_t *t;

  for (int i=0; i<N; i++) {
    FILE *in = abrirTerminalIn(i);
    FILE *out = abrirTerminalOut(i);

    t = (terminal_t*) malloc(sizeof(terminal_t));

    // Inicializa t e passa-o como argumento de input à nova tarefa [COMPLETAR]

    t->in = in; t->out = out;

    pthread_create(&tid[i], 0, fnTerminalReserva, t);

  }

  FILE *display = abrirTerminalOut(N);
  t = (terminal_t*) malloc(sizeof(terminal_t));

  // Inicializa t e passa-o como argumento de input à nova tarefa [COMPLETAR]

  t->out = display;

  pthread_create(&tid[N], 0, fnAnuncio, t);

  /* ... restante código omitido ... */
}
```

2. [4 val] O excerto de código sombreado a cinzento na função fnTerminalReserva não está corretamente implementado para ser executado concorrentemente. Proponha uma alternativa correta para esse excerto.
   Na sua resposta:
   - Deve permitir o maior paralelismo possível entre tarefas que pretendam reservar conjuntos disjuntos de lugares. Deve também prevenir situações de inter-blocagem.
   - Deve apenas modificar o excerto de linhas da função *fnTerminalReserva* assinalado a cinzento. Caso encontre problemas de concorrência noutras partes do programa, ignore-os nesta alínea.
   - Pode usar pseudo-código, desde que legível.
   - Pode recorrer aos mecanismos de sincronização estudados nas aulas (*mutexes*, trincos leitura-escrita, semáforos, variáveis de condição). Não esquecer de os declarar e inicializar.

Declaração/inicialização de variáveis partilhadas adicionais:

```
mutex_t m[D];
for (int i=0; i<D; i++)
  mutex_init(&m[i]);
```

Nova implementação do excerto de código sombreado na função *fnTerminalReserva:*

```
    sort(paraReservar); //Ordena as entradas no vetor
    for (i=0; i<4; i++) {
      mutex_lock(&m[paraReservar[i]);
      if (&mapaAtual[paraReservar[i]] == OCUPADO) {
        sucesso = FALSE;
        for (int k=0; k<=i; k++)
           mutex_unlock(&m[paraReservar[k]]);
        break;
      }
    }
    if (sucesso == TRUE) {
      for (i=0; i<4; i++) {
        mapaAtual[paraReservar[i]] = OCUPADO;
        mutex_unlock(&m[paraReservar[k]]);
      }
    }
```

Nota: há diferentes soluções corretas para esta questão, pelo que apresentamos apenas uma possibilidade (em pseudo-código).
Uma solução alternativa (entre outras) seria usar a variante "trylock" para adquirir o mutex associado a cada lugar e, em caso de lugar ocupado ou trancado, libertar os mutexes/lugares adquiridos até ao momento.

3. [3 val] A função *fnAnuncio* baseia-se em espera ativa. Recorrendo a semáforos ou variáveis de condição, em combinação com um *mutex*, proponha uma alternativa que elimine essa limitação. Não se esqueça de declarar e inicializar os mecanismos de sincronização que precisar. Apresente **apenas** as partes do programa original que modificaria (nas funções *fnAnuncio* e/ou *fnTerminalReserva*).

Abaixo apresentamos soluções baseadas em semáforo e em variável de condição (ambas igualmente corretas).

Declaração/inicialização de variáveis partilhadas adicionais:

```
mutex_t mVagas;
mutex_init(&numVagas);

sem_t shuttleCheio;
sem_init(&shuttleCheio, 0, 0);
/*...ou...*/
cond_t shuttleCheio;
cond_init (&shuttleCheio);
```

Alterações às funções *fnAnuncio* e/ou *fnTerminalReserva:*

```
void *fnAnuncio (void *arg) {
  FILE* outfd = ((terminal_t *) arg)->out;

  while (TRUE) {
    sem_wait(&shuttleCheio);
/* ... ou ... */
    mutex_lock(&mVagas);
    while (nVagas > 0) cond_wait(&shuttleCheio, &mVagas);
    mutex_unlock(&mVagas);
```

```
    fprintf(outfd, "Shuttle parte em 10min\n");
    /* ... */
  }
}

void *fnTerminalReserva(void *arg) {
  /* ... */
  if (sucesso == TRUE) {
      mutex_lock(&mVagas);
      nVagas -= 4;
      if (nVagas==0)
          sem_notify(&shuttleCheio);
          /* ... ou ... */
          cond_signal(&shuttleCheio);
      mutex_unlock(&mVagas);

      fprintf(outfd, "Lugares reservados!\n");
    }
    /* ... */
}
```

Grupo III [6 Val]

1. [2,5 val] Implemente um programa que executa a seguinte concatenação de comandos Unix:

`cat ./file.txt | grep SO > /tmp/mynamedpipe`

Assuma que /tmp/mynamedpipe é um *named pipe* previamente criado por um processo servidor S, que aguarda mensagens por esse pipe. S já está implementado (não precisa de o implementar). Sugestões: execute cada programa (*cat* e *grep*) em processos distintos, com o *stdout* do 1º processo redirecionado para o *stdin* do 2º processo; redirecione o *stdout* do 2º processo para o *named pipe*. Na sua solução, omita o código de tratamento de erros.

```
#include <unistd.h>
#include <fcntl.h>

int main() {

    int fds[2];
    pipe(fds);

    int pid=fork();
    if (pid==0) {
        close(0);
        dup(fds[0]);
        close(fds[0]);
        close(fds[1]);
        int pipe_fds=open("/tmp/mynamedpipe",O_WRONLY);
        close(1);
        dup(pipe_fds);
        close(pipe_fds);
        execl("/usr/bin/grep","grep","SO",0);
    }
    else {
        close(1);
        dup(fds[1]);
        close(fds[0]);
        close(fds[1]);
        execl("/bin/cat","cat","./file.txt",0);
    }
}
```

2. [0,75 val] Suponha que um processo executa **duas** vezes *write(fd, buf, strlen(buf))* onde:
   - fd é um descritor de ficheiro aberto em modo de escrita e associado a um *named pipe*
   - buf é uma cadeia de caracteres (*string*) cujo valor é "Sistemas Operativos"

e que, a seguir, um outro processo executa *read(fd, rec_buf, 100)*, onde rec_buf é um *byte array* previamente alocado com tamanho 100.
Todas estas chamadas retornaram com sucesso.

Após a chamada a *read*, qual é o valor retornado e qual o conteúdo do rec_buf?

Valor de retorno: 38
Buf=[Sistemas OperativosSistemas Operativos<valores arbitrários>"

3. [0,75 val] Suponha que um processo executa **duas** vezes *sendto(fd, buf, strlen(buf),0, dest_addr, dim)* onde:
   - *fd* é um descritor de ficheiro associado a um *datagram socket* de domínio *Unix*
   - *buf* é uma cadeia de caracteres (*string*) cujo valor é "Sistemas Operativos"
   - *dest_addr* e *dim* são o endereço de um socket datagram e a respetiva dimensão.

   e que, a seguir, um outro processo execute *recvfrom(fd2, rec_buf, 100, 0, NULL, NULL)* onde:
   - fd2 é o descritor de ficheiro associado ao *datagram socket* de domínio Unix com endereço *dest_addr.*
   - rec_buf é um *byte array* previamente alocado de tamanho 100.

   Todas estas chamadas retornaram com sucesso.

Após a chamada a recvfrom, qual é o valor retornado e qual o conteúdo do rec_buf?

Valor de retorno: 19
Buf=[Sistemas Operativos<valores arbitrários>"

4. Considere o seguinte programa:

```
1. #include <stdio.h>
2. #include <unistd.h>
3. #include <signal.h>
4. #include <stdlib.h>

5. int pid;

6. void c (int s) {
7.     char ch;
8.     printf("Certeza?\n");
9.     ch = getchar();
10.    if (ch == 's')
11.         exit(0);
12.    signal (SIGINT, c);
}

int main () {
13.     signal (SIGINT, c);
14.     for (;;) pause();
}
```

a) [1 val] Assumindo que a função signal tem a semântica System V, o que acontece se o utilizador carregar duas vezes, de seguida, nas teclas CTRL+C?

O programa termina: ao ativar a função de tratamento do signal, o SO associa a rotina por omissão ao SIGINT (gerado pelo CTRL+C), o que termina o processo.

b) [1 val] Assumindo agora que a função *signal* tem semântica BSD, o que acontece se o utilizador carregar duas vezes, de seguida, nas teclas CTRL+C?

O segundo CTRL+C é recebido enquanto o processo executa a função de tratamento do signal, provavelmente bloqueado no getchar(). Nesta fase a receção do signal é inibida e o processo não termina. Caso o utilizador insira um caracter diferente do 's', a função de tratamento será chamada novamente.