Fundamentos de Sistemas de Operação MIEI 2016/2017

1º Teste - 2 de Novembro de 2016– Duração 2h – versão A

Nome: Nº

Avisos: *Sem consulta; a interpretação do enunciado é da responsabilidade do estudante; se necessário explicite na resposta a sua interpretação.*

Algumas chamadas ao sistema UNIX/Linux

int fork ( )

int execvp (char *executable_file, char * args[ ])

int wait (int *status)

int exit (int status)

int pipe (int fd[2])

int dup (int chan)

Algumas funções da biblioteca de Pthreads

Gestão de processos

int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg)

int pthread_join (pthread_t thread, void **retval)

int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *attr) ou

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock (pthread_mutex_t *mutex)

int pthread_mutex_unlock (pthread_mutex_t *mutex)

biblioteca de sockets UDP

int UDP_Open (int port)

int UDP_FillSockAddr (struct sockaddr_in *addr, char* hostName, int port)

int UDP_Read (int sd, struct sockaddr_in *addr, char* buffer, int n)

int UDP_Write (int sd, struct sockaddr_in *addr, char* buffer, int n)

Questão 1 – 2,0 valores

Porque é que um sistema operativo que suporta múltiplos processos só pode cumprir a sua especificação se o CPU tiver dois modos de operação? Diga as diferenças entre o modo utilizador e o modo sistema.

O SO assegura os recursos para as máquinas virtuais de todos os processos; para garantir uma distribuição correta destes recursos, o SO reserva para si próprio a atribuição de recursos. Isto obriga a que apenas o SO possa atribuir recursos e portanto tem de impedir que os processos o possam fazer. O que caracteriza o modo supervisor é a possibilidade de executar instruções privilegiadas, e a atribuição de recursos (CPU, RAM, periféricos) só pode ser feita executado instruções privilegiadas.

Questão 2 – 2,0 valores

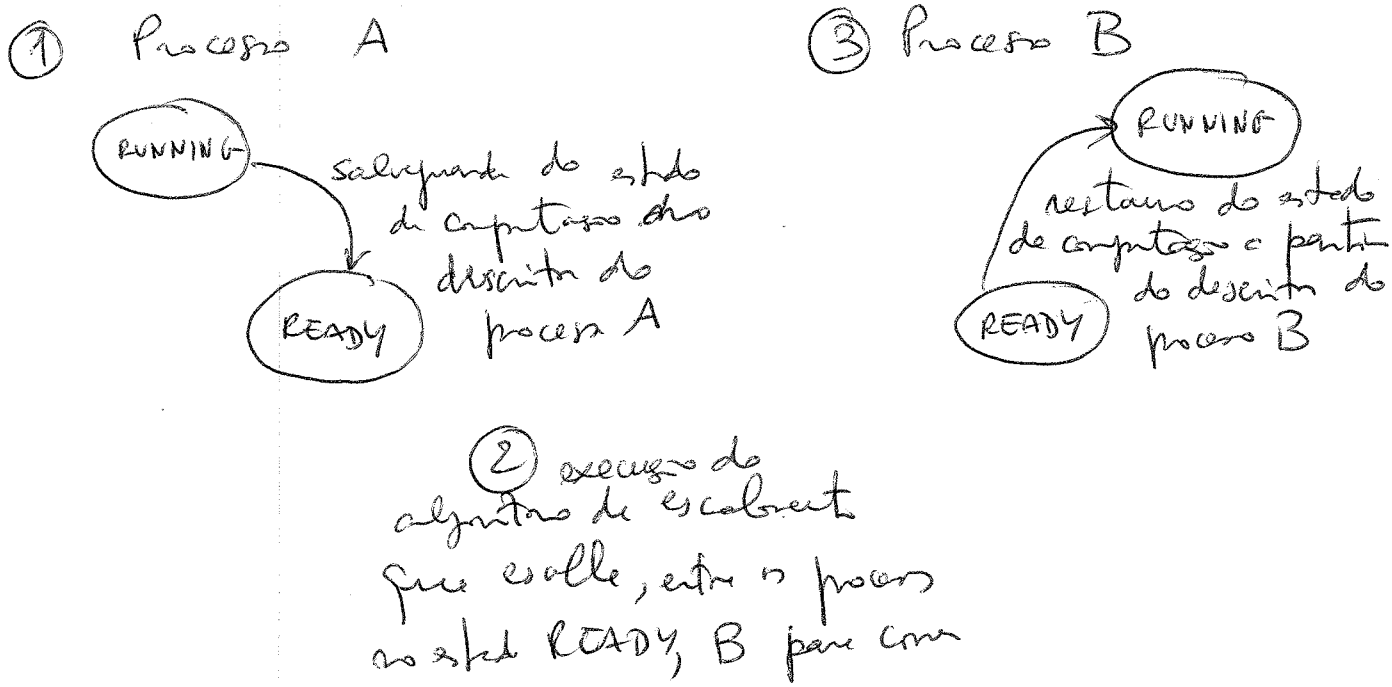
Diga quais são as principais funções de um sistema operativo que suporta múltiplos processos.

Assegura para todos os processos
- um CPU virtual que permite ao processo executar as instruções do programa que lhe foi atribuído
- uma RAM onde instalam o código, dados, heap e pilha do processo
- um conjunto de canais virtuais que permite a interação com o exterior do processo.

O SO assegura que todos os núcleos virtuais executam sem interferirem entre si, isto é, de forma independente

Questão 3 – 2,0 valores

Considere a chamada ao sistema *sched_yield( )* em que, como é sabido, o processo A que invoca a chamada liberta o CPU para que possa ser usado por outros processos; suponha que em consequência desta ação do processo A, é escolhido para ocupar o CPU o processo B. Descreva em detalhe as ações efectuadas pelo sistema operativo, nomeadamente que transições de estados ocorrem para os processos A e B e que ações é que o sistema operativo efetua nos descritores dos processos A e B.



Questão 4 – 3,0 valores

Como é sabido, quando um utilizador escreve na linha de comando do *shell* a sequência

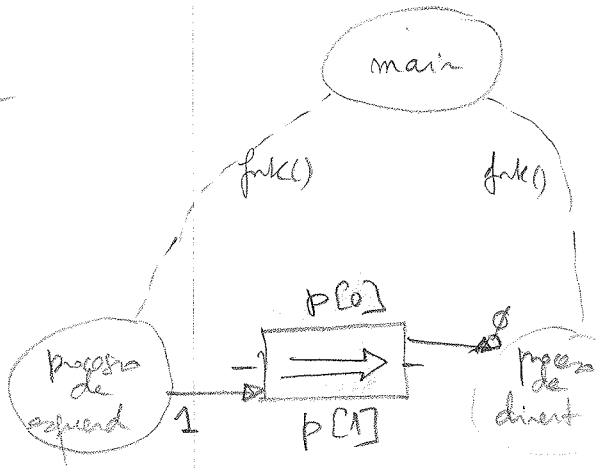
`nome_de_ficheiro_executável1 | nome_de_ficheiro_executável2`

o shell

- Cria um processo *p1* para executar `nome_de_ficheiro_executável1`
- Cria um processo *p2* para executar `nome_de_ficheiro_executável2`
- Redirige o canal *stdout* de *p1* para um *pipe* criado previamente e o canal *stdin* de p2 para o mesmo *pipe*

Suponha que o fragmento de código seguinte foi retirado do código da *shell*. Complete os troços em falta

```
...
char *args[MAXARGS];
int df[2], p1, p2;
// suponha que foi chamada a função makeargv e que args contém apontadores para as
// três componentes da linha e que as 3 strings estão devidamente terminadas.
pipe(fd);
p1 = fork();
if (p1 > 0) { // sem teste de erro
      p2 = fork() ;
      if (p2 > 0) { // sem teste de erro // main
            close (fd[0]); close (fd[1]);
            wait (NULL);
            wait (NULL);
      }
      else {  // processo da direita
            close(0);
            dup (fd[0]);
            args[0] = "nome_do_ficheiro_executavel2";
            args[1] = NULL;
            execvp("nome_do_ficheiro_executavel2", args);
      }
} else {  // processo da esquerda
      close (1);
      dup(fd[1]);
      args[0] = "nome_do_ficheiro_executavel1";
      args[1] = NULL;
      execvp("nome_do_ficheiro_executavel3", args);
}
```



Questão 5 – 3,0 valores

Complete o seguinte programa que pretende utilizar 2 *threads* para calcular o histograma de uma imagem em tons de cinzento. A imagem é representada por uma matriz com SIZE*SIZE posições em que cada posição pode tomar um valor entre 0 e 255; supõe-se que SIZE é par. Recorde-se que o histograma determina quantas ocorrências de cada valor existem na matriz.

```
#include <stdio.h>
#include <pthread.h>
#define SIZE 1024
#define NO_GREY_LEVELS 256
unsigned char image[SIZE][SIZE];
unsigned int histogram[NO_GREY_LEVELS];
// espaço para eventuais declararações em falta
pthread_mutex_t ex;
void *worker(void * arg) {  // o processo que recebe 0 analisa as linhas pares
    int i, j;               // o processo que recebe 1 analisa as linhas impares
    int parity = (int) arg;
    for (i=parity; i< SIZE; i = i+2)
        for(j = 0; j< SIZE; j++) {
            pthread_mutex_lock(&ex); histogram[image[i,j]]++; pthread_mutex_unlock(&ex);
        }
}
int main(int argc, char *argv[]){
    int i; pthread_id t1, t2;
    // espaço para eventuais inicializações
    for(i=0; i<NO_GREY_LEVELS;i++) histogram[i] = 0;
    pthread_create(&t1, NULL, worker , 0 );
    pthread_create(&t2, NULL, worker , 1 );
    pthread_join( t1 , NULL); pthread_join( t2 , NULL);
    for(i = 0; i < NO_GREY_LEVELS; i++)  printf(" %d\n", histogram[i]);
    return 0;
}
```

a) Complete o código apresentado.

b) Quantas operações de sincronização entre os threads são executadas no seu código? Seria possível que fossem menos? Se sim, diga como faria.

Em b) cada thread faz (SIZE/2)*(SIZE) pthread_mutex_lock e pthread_mutex_unlock.
Uma forma de diminuir o número de invocações de lock/unlock era:
- declarar uma versão local do vector histograma em cada thread
- deixar o thread-main somar os dois elementos após fazer os dois join

Questão 6 – 3,0 valores

Considere que se pretende construir um cliente e um servidor sequencial em que

- o cliente recebe do terminal uma linha e envia-a, usando um datagrama UDP, para o servidor
- o servidor
  - conta quantas palavras existem na linha, chamando a função *makeargv( )* usada nas aulas práticas
  - devolve esse número, representando-o como uma sequência de caracteres
- o cliente mostra no ecrã esse número

Complete o código do servidor e do cliente abaixo indicados

Cliente

```
#include "udp_comm.h"
#define BUFSIZE 1024
#define SERVER "localhost"
#define SERVERPORT 1234
#define CLIENTPORT 1235
char request[BUFSIZE];
char reply[BUFSIZE];
struct sockaddr_in serv;
int sock;

int main(int argc, char *argv[]){
      sock = UDP_Open(CLIENTPORT);
      // teste de erro omitido
      fgets(request, BUFSIZE, stdin);

      //Preparar o endereço de destino
      UDP_FillSockAddr(&srv,
            argv[1], SERVERPORT);

      //Enviar a mensagem
      UDP_Write(sock, request,
        strlen(request)+1);

      //Receber a resposta
      UDP_Read(sock, reply,
            BUFSIZE);

      //imprimir o seu conteudo.
      printf("%s\n", reply);
      return 0;
}
```

Servidor

```
#include "udp_comm.h"
#define MAXW  32
#define BUFSIZE 1024
#define SERVERPORT 1234
char request [BUFSIZE];
char reply[BUFSIZE];
struct sockaddr_in cli;
int sock;
char *args[MAXW+1];

int main(int argc, char *argv[]){
      sock = UDP_Open(SERVERPORT);
      // teste de erro omitido
      while(1){
      //Receber o pedido
      UDP_Read(sock, request, BUFSIZE);

      // Preparar a resposta
      int nw = makeargv(request, args);
      // retorna o numero de palavras e
      // preenche args
      //Enviar a resposta
      memset(reply, 0, BUFSIZE);
      int n = sprintf(reply, "%d\n", nw);
      UDP_Write(sock, reply, n+1);

      }

}
```

Questão 7 – 2,0 valores

Nos sistemas operativos que suportam múltiplos processos e que são usados nos computadores pessoais e em servidores, é muito comum que o escalonamento do(s) CPU(s) seja baseado numa variante do algoritmo Multiple Level Feedback Queue (MLFQ). Explique as razões para esta preferência.

Os algoritmos de escalonamento para computadores pessoais e servidores procuram assegurar tempo de resposta curto, justiça e dar prioridade aos processos que façam operações de entrada/saída.
O algoritmo MLFQ assegura
- justiça, porque faz Round Robin entre processos de mesma prioridade
- dá prioridade aos processos que fazem muitas operações de E/S mantendo-os em prioridade alta
- evita a "starvation" de processos com baixa prioridade, recalculando esta periodicamente, aumentando a prioridade dos processos que recentemente tiveram pouco CPU.

Questão 8 – 2,0 valores

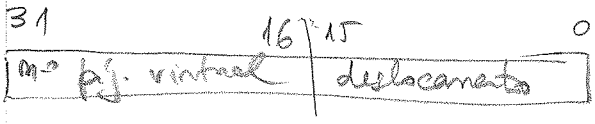
Explique a diferença entre endereços virtuais e endereços físicos e diga porque é que, num sistema operativo que suporta múltiplos processos, é necessário que existam estes dois tipos de endereços.

No processo de compilação/ligação não é conhecido o endereço onde é carregado o programa; assume-se um end. de carregamento inicial Ø. Como os endereços onde o programa é carregado só são conhecidos no momento de carregamento é preciso distinguir entre
- endereços virtuais que são os emitidos pelo CPU e que são independentes da localização do programa na RAM
- endereços físicos que são os que são usados para carregamento programa num dado acesso

A MMU (Memory Management UNIT) faz a transformação de endereços virtuais em endereços físicos

Questão 9 – 1,0 valor

Considere um CPU com um endereço virtual com 32 bits e em que a memória central (RAM) é gerida usando páginas com 64 Kbytes (2^{16}). Qual o número de entradas da tabela de páginas de um processo? Justifique.



16 bit para o nº pág. virtual → 2^{16} entradas