ASOP - Arquitecturas e Sistemas Operativos (2009/10)

2ª Parte do Exame (Sistemas Operativos/A)
5 de Julho de 2010

IMPORTANTE: Esta parte do Exame está cotada para 10 valores, tem uma duração de 1 hora e deverá ser realizada individualmente e sem consulta de apontamentos.

IMPORTANTE: Por favor, responda às perguntas 2 e 3 numa folha e às perguntas 4 e 5 noutra folha. A pergunta 1 deverá ser respondida nesta folha, pelo que deverá preencher o seu nome no espaço reservado para o efeito.

Nome:

1- Em cada alínea desta pergunta deverá escolher a afirmação mais completa que é verdadeira. Indique-a no espaço reservado para o efeito nesta folha, a qual deverá ser entregue no final.

A cotação de cada pergunta é a seguinte:
- Cada resposta correcta vale 0,4 valores;
- Cada resposta errada vale -0,1 (note o sinal menos) valores;
- A ausência de resposta vale 0 valores.

1.1- [0,4v/2min] Sobre chamadas ao sistema:

A O conjunto das chamadas ao sistema dum sistema operativo (SO) constituem a interface de programação (API) desse SO.

B Uma chamada ao sistema é invocada usando instruções de chamada de subrotinas tal como `call`, a diferença é as suas instruções fazerem parte do *kernel*.

C O tempo de invocação dum segmento de código é independente dele ser implementado como uma chamada ao sistema ou como uma função.

Resposta:

1.2- [0.4v/2min] Sobre *threads* e o exemplo do servidor *Web* apresentado nas aulas teóricas.

A Em vez de *threads* podia usar-se processos, mas a solução resultante teria muito provavelmente um menor desempenho.

B Em vez de *threads* podia usar-se processos.

C Em vez de *threads* podia usar-se processos, mas a solução resultante seria muito provavelmente mais susceptível a *bugs*.

Resposta:

1.3- [0,4v/2min] Sobre o escalonamento de processos:

A Num algoritmo de escalonamento sem preempção, um processo nunca passa do estado de *running* para o estado de *ready* a menos que o processo invoque uma chamada ao sistema para o efeito.

B A forma mais eficiente de implementar um algoritmo preemptivo, é o SO fazer a comutação de processos quando da invocação de chamadas ao sistema.

C Quando a maioria dos processos são IO-bound é irrelevante se o algoritmo de escalonamento usado é ou não preemptivo.

Resposta:

1.4- [0.4v/2min] Sobre *race conditions*.

A Uma *race condition* nunca pode ocorrer no acesso concorrente a uma variável cujo valor não pode ser alterado.

B Uma *race condition* é um *bug* que se manifesta em todas as execuções.

C O *algoritmo da avestruz*, i.e. ignorar o problema, é uma boa solução para *race conditions*.

Resposta:

1.5- [0,4v/2min] Sobre suporte do *hardware* para exclusão mútua.

A As instruções *read-modify-write* atómicas podem ser usadas quer em sistemas mono-processador quer em sistemas multi-processador.

B Em sistemas mono-processadores, o uso de inibição/permissão de interrupções para resolver *race conditions* permite tanta concorrência quanto instruções *read-modify-write* atómicas.

C A disponibilização de operações de sincronização ao nível do SO, p.ex. *mutexes*, torna desnecessário o suporte em HW para exclusão mútua.

Resposta:

1.6- [0.4v/2min] Sobre monitores e variáveis de condição.

A Sem variáveis de condição, os monitores asseguram apenas exclusão mútua.

B As variáveis de condição POSIX servem apenas para implementar monitores em C.

C Monitores obrigam sempre a *busy-waiting*.

Resposta:

1.7- [0.4v/2min] Sobre *deadlocks* (bloqueio mútuo).

A Um processo não pode participar da cadeia de processos mutuamente bloqueados, se estiver bloqueado à espera dum recurso mas não tiver em sua posse qualquer recurso.

B O bloqueio mútuo (*deadlock*) é um problema que ocorre exclusivamente no acesso a recursos físicos.

C Recursos preemptíveis não podem dar origem a *deadlocks*.

Resposta:

1.8- [0,4v/2min] Sobre sistemas de gestão de memória.

A O tamanho das páginas usadas em sistemas de memória virtual paginada é determinado pelo *hardware*.

B *Swapping* designa a transferência dados entre quaisquer dois níveis da hierarquia de memória.

C A gestão de memória com partições de tamanho fixo pode ser implementada de forma eficiente sem recurso a qualquer mecanismo em *hardware*.

Resposta:

1.9- [0.4v/2min] Sobre memória virtual paginada.

A Num sistema de memória virtual paginada os endereços que circulam no barramento de endereços são endereços físicos.

B Num sistema de memória virtual paginada os endereços no código dum programa guardado na memória são endereços físicos.

C A rapidez de execução dum processo é independente do número de *page faults* que esse processo tem.

Resposta:

1.10- [0.4v/2min] Sobre sistemas de ficheiros em Unix/Linux.

A Os elementos dum directório podem ser de 1 de 2 tipos: *hard links* ou *soft links*.

B Para que uma dado recurso possa ser abstraído como um ficheiro terá que suportar quer acesso aleatório quer acesso sequencial.

C A *buffer-cache* permite acelerar o processo de resolução de nomes.

Resposta:

2- Responda a cada uma das perguntas seguintes.

2.1- [0.5v/3min] Sobre a implementação de *threads*. A frequência de *page-faults* numa aplicação *multi-threaded* poderá ser um factor na escolha do tipo de implementação de *threads* a usar por essa aplicação? Justifique.

2.2- [0.5v/3min] Considere a implementação de memória virtual paginada com uma tabela de 2 níveis. Quantos acessos à memória podem ser necessários para determinar o endereço físico duma página? Justifique.

3- Considere os seguintes segmentos de código duma solução para o problema do *bounded-buffer* com semáforos.

```
typedef struct {
    int in, out;
    sem_t mutex;
    sem_t slots;
    sem_t items;
    void *buf[BUF_SIZE];
} bbuf_t;
1: void enter(bbuf_t *bbp, void *op) {
2:    sem_wait(&(bbp->mutex));
3:    sem_wait(&(bbp->slots));
4:    bbp->buf[bb->in] = op;
5:    bbp->in = (bbp->in+1) % BUF_SIZE;
6:    sem_post(&(bbp->mutex));
7:    sem_post(&(bbp->items));
8: }
A: void *remove(bbuf_t *bbp) {
B:    void *op;
C:    sem_wait(&(bbp->mutex));
D:    sem_wait(&(bbp->items));
E:    op = bbp->buf[bb->out];
F:    bbp->out = (bbp->out+1) % BUF_SIZE;
G:    sem_post(&(bbp->mutex));
H:    sem_post(&(bbp->slots));
I: }
```

3.1- [0.5v/3min] Explique para que serve cada um dos semáforos da estrutura `bbuf_t` e com que valores deverão ser inicializados.

3.2- [0.5v/4min] Explique e mostre através duma sequência de execução de instruções que este código pode dar origem a um problema de sincronização. (Use os "números" indicados no início de cada linha.)

4- [2v] Considere o seguinte programa.

```
 1: #include <stdio.h>
 2: #include <unistd.h>
 3:
 4: int random() {
 5:     return ...; //gera um numero entre 1 e 10
 6: }
 7:
 8: int main() {
 9:
10:     pid_t p;
11:     int a = 0;
12:
13:     p = fork();
14:     if(p == -1) {
15:         printf("Erro no fork()\n");
16:         return -1;
17:     } else if (p == 0) {
18:         a = random();
19:         printf("1: a = %d\n", a);
20:         return a;
21:     } else {
22:         printf("2: a = %d\n", a);
23:     }
24:     printf("3: a = %d\n", a);
25:
26:     return 0;
27: }
```

4.1- [6min] Apresente todas sequências distintas de *strings* que o programa pode imprimir, assumindo que a invocação de `random()` retorna sempre o valor 3?

4.2- [7min] Altere o programa de modo a que na ausência de erros todas as intruções `printf()`, independentemente do processo que as executa, imprimam o valor retornado pela invocação da função `random()` na linha 18. Para tal, considere as seguintes possibilidades:

i) mecanismos de sincronização entre processos;

ii) apontadores;

justificando se o mecanismo em questão não for adequado.

5- [2v] Considere as seguintes funções e declarações.

```
#include <stdio.h>
#include <pthread.h>

#define ITERATIONS 10000
#define NUMBER_OF_THREADS 6

unsigned count = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;

void *thread(void *vargp)
{
    printf("thread n %i has just started\n", (int *)vargp);
    int i;
    for (i = 0; i < ITERATIONS; i++) {
        count++;
    }
    printf("thread n %i has just finished\n", (int *)vargp);

    return NULL;
}

int main( int argc, char *argv[])
{
    pthread_t tid[NUMBER_OF_THREADS];
    int i;

    ...

    printf("count=%d\n", count);
    return 0;
}
```

5.1- [6 min] Complete o código da função `main()`, que deve criar e aguardar a terminação de `NUMBER_OF_THREADS` *threads*, baseados na função `thread()`, passando aos *threads* o número do *thread* criado (este número deverá ser sequencial e estar entre 0 e `NUMBER_OF_THREADS`).

5.2- [8 min] Para cada um dos seguintes objectivos:

i) maximizar a concorrência da execução do programa;

ii) minimizar o tempo de execução (em vez de maximizar concorrência);

proteja as secções críticas de código na função `thread()`, usando o *mutex* `m` já declarado e inicializado. Justifique.

Protótipos

Processos

```
pid_t fork(void);
int execve(const char *filename, char *const argv[], char *const envp[]);
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
void _exit(int status);
WIFEXITED(); // macro
WEXITSTATUS(); // macro
```

Threads

```
int  pthread_create(pthread_t  *  thread, pthread_attr_t * attr,
                    void *(*start_routine)(void *), void * arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t th, void **thread_return);
pthread_t pthread_self(void);
```

Mutexes/Locks

```
pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int  pthread_mutex_init(pthread_mutex_t  *mutex,
                        const pthread_mutex_attr_t *mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Variáveis de Condição

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int    pthread_cond_init(pthread_cond_t    *cond,    pthread_condattr_t
       *cond_attr);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_destroy(pthread_cond_t *cond);
```

Semáforos

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

Sistema de Ficheiros

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
        flags: O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, O_APPEND
        mode: S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP,
              S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
        prot: PROT_NONE, PROT_EXEC, PROT_READ, PROT_WRITE
        flags: MAP_FIXED, MAP_SHARED, MAP_PRIVATE
int munmap(void *start, size_t length);
int msync(void *start, size_t length, int flags);
off_t lseek(int fildes, off_t offset, int whence);
       whence: SEEK_SET, SEEK_CUR, SEEK_END
```