

Sistemas Operativos 2021 / 2022

Licenciatura em Engenharia Informática

Lab. 06 – Sincronização de processos Unix – Memória Partilhada

Nesta aula pretende-se que os alunos fiquem com uma noção prática da sincronização de processos em Linux recorrendo à utilização de memória partilhada.

Ex. 1 – Utilização do *mmap*

O seguinte programa exemplifica a utilização de memória partilhada entre processos Unix recorrendo à chamada de sistema *mmap*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <sys/mman.h>
6
7 int main()
8 {
9     // Create shared memory map
10    int size = 64;
11    int protection = PROT_READ | PROT_WRITE;
12    int visibility = MAP_ANONYMOUS | MAP_SHARED;
13    void *shmem = mmap(NULL, size, protection, visibility, 0, 0);
14
15    int cpid = fork();
16    if (cpid == 0) {
17        printf("Child process!\n");
18        char *msg = "Hello from child process..\n";
19        strcpy(shmem, msg);
20        exit(0);
21    }
22    else {
23        printf("Parent process!\n");
24        sleep(1);
25        printf("shmem=%s", (char*) shmem);
26    }
27
28    return (EXIT_SUCCESS);
29 }
```

Coloque o código num ficheiro de nome *ex1.c*, compile com o *gcc* e verifique o *output* do programa:

```
$ gcc -o ex1 ex1.c
$ ./ex1
```

```
Parent process!
Child process!
shmem=Hello from child process..
```

De uma forma geral, o programa inicia criando a variável *shmem* que irá alocar uma área de memória que é configurada de modo a ser partilhada entre processos. Após o *fork*, o processo filho copia uma *string* para a área de memória partilhada, enquanto o processo pai aguarda um instante e lê a *string* da memória partilhada.

- Veja as entradas de manual da função *mmap* tomando especial atenção aos argumentos *size*, *prot*, *flags* e *fd*. Verifique como os argumentos são utilizados no exemplo acima e justifique a sua utilização.
- Modifique o exemplo anterior de modo a que o processo pai espere pela finalização do processo filho. Deverá remover a função *sleep* e usar as funções *wait* e *exit* usadas nos laboratórios anteriores.
- Altere o código anterior de modo a que seja o processo pai a enviar a mensagem "Hello from parent process.." para o processo filho listar.
- Usando o código da alínea c, coloque um *sleep(1)* antes do *strcpy(shmem, msg)* (no processo pai) de modo a simular um atraso na utilização da memória partilhada. Compile e execute o código, e verifique que o processo filho não escreve a mensagem do processo pai pois termina antes. Resolva o problema usando semáforos, garantindo que, no processo pai, adquira o semáforo antes do *sleep* (não se esqueça de compilar usando *-pthread*).

Ex. 2 – Filas de tarefas distribuídas

Uma fila de tarefas distribuídas é um sistema composto por um processo responsável pela criação de tarefas, e por um conjunto de processos filhos chamados *workers*, responsáveis pela execução das tarefas. O objectivo é distribuir as tarefas por múltiplos processos de modo a aliviar a carga do processo principal.

Neste exercício iremos simular uma fila de tarefas distribuídas em que os *workers* calculam o tamanho das *strings* geradas pelo processo principal.

- Compile e execute o exemplo em anexo. Este exemplo implementa um sistema que cria vários processos filhos que ficam em ciclo infinito à espera de trabalho. No fim, todos os processos filhos (*workers*) são terminados usando a função *kill*.
- Modifique o exemplo de modo a criar uma área de memória partilhada. O processo pai deverá utilizar esta área de memória para colocar uma *string* e os *workers* deverão imprimir

o seu *id* e o tamanho da *string*. Modifique novamente o seu código de modo a que o processo pai vá colocando várias *strings* na memória partilhada. Sugestão: use um ciclo *for*.

- c) O output da alínea anterior mostra que alguns *workers* iniciam o trabalho antes do processo pai colocar a *string* na memória partilhada. Modifique o código de modo a que os *workers* não iniciem o processo sem que haja um trabalho disponível, e de modo a que um trabalho seja executado por um único *worker*.

Sugestão: utilize dois semáforos “*job_ready*” e “*job_done*” para controlar os acessos à memória partilhada. O processo pai deverá colocar uma *string* na memória partilhada, avisar os *workers* que existe uma tarefa pronta e esperar a sua conclusão. Os *workers*, por sua vez, deverão esperar por uma tarefa disponível e avisar o processo pai quando a tarefa estiver terminada.

Não esquecer de compilar usando *-pthread*.

ex2.c

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <signal.h>
5
6 int main()
7 {
8     // Workers
9     int num_workers = 5;
10    int pids[num_workers];
11
12    // Fork worker processes
13    for (int i=0; i<num_workers; i++) {
14        pids[i] = fork();
15        if (pids[i] == 0) {
16            printf("Worker process #%d!\n", i);
17            while (1) {
18                // Do work
19            }
20            exit(0);
21        }
22    }
23
24    // Parent process
25    printf("Parent process!\n");
26
27    // Kill worker processes
28    for (int i=0; i<num_workers; i++) {
29        printf("Killing %d\n", pids[i]);
30        kill(pids[i], SIGKILL);
31    }
32
33    return (EXIT_SUCCESS);
34 }
```

(fim de enunciado)