

Sistemas de Programação II
INF 02828
(Engenharia de Computação / Elétrica)

Sistemas Operacionais
INF 02780
(Ciência da Computação)

1º Trabalho de Programação

Período: 2008/1

Profª Patrícia Dockhorn Costa

Email: pdcosta@inf.ufes.br

Data de Entrega: 27/05/2008

Permitidos Grupos de 2-3 pessoas

Data de entrega inadiável. Para cada dia de atraso, será tirado um ponto da nota.

Material a entregar

- Impresso:

- Relatório contendo as respostas das questões.
- Listagem de todos os arquivos .c e .txt criados

- Por email:

- Relatório (em formato PDF)
- Todos os arquivos .c e .txt criados

A. Process Chain [2,0]

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

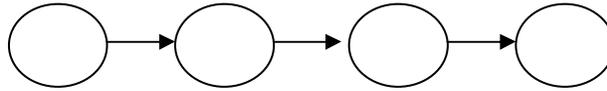
int main (int argc, char *argv[]) {
    pid_t childpid = 0;
    int i, n;

    if (argc != 2){ /* check for valid number of command-line arguments */
        fprintf(stderr, "Usage: %s processes\n", argv[0]);
        return 1;
    }
    n = atoi(argv[1]);
    for (i = 1; i < n; i++)
        if (childpid = fork())
            break;

    fprintf(stderr, "i:%d process ID:%ld parent ID:%ld child ID:%ld\n",
            i, (long)getpid(), (long)getppid(), (long)childpid);
    return 0;
}
```

1. Execute o programa acima (crie “progA.c”) e observe (anote) os resultados para diferentes números de processos (2, 3, 5, 10 processos).

2. Anote a saída de uma execução passando como argumento o valor 4 e preencha o desenho da figura com os IDs dos processos para.



3. Faça diferentes tentativas de execução para descobrir qual o número máximo de processos que é possível criar.
4. Altere o programa (crie “progA4.c”) adicionando uma linha “`sleep(10);`” imediatamente antes do último `fprintf`. Qual o número máximo de processos gerados neste caso?
5. Altere o programa (crie “progA5.c”) colocando o último `fprintf` dentro de um loop que deve ser executado `k` vezes. Coloque uma linha “`sleep(m);`” logo após o `fprintf` dentro deste loop. Receba os valores de `k` e `m` como parâmetros a partir da linha de comando (juntamente com `n`). Execute o programa várias vezes para diferentes valores de `n`, `k` e `m`. Observe, anote e explique os resultados.
6. Altere o programa (crie “progA6.c”) colocando uma chamada `wait` antes do último `fprintf`. Como isso altera a saída do programa?
7. Modifique o programa (crie “progA7.c”) substituindo o último `fprintf` por 4 `fprintf`, um para cada um dos inteiros escritos na saída (apenas o último `fprintf` deverá escrever o caractere de quebra de linha). O que acontece quando este programa é executado? É possível determinar qual processo gerou cada parte da saída? Execute o programa várias vezes e verifique se há alguma diferença na saída. Explique os resultados encontrados.
8. Modifique o programa (crie “progA8.c”) substituindo o último `fprintf` por um loop que lê `nchars` caracteres da entrada padrão, e os armazena em um array chamado `mybuf`. Os valores de `n` e `nchars` devem ser passados como argumentos na linha de comandos. Após o loop, adicione um caractere ‘\0’ na entrada `nchars` do array (para que ele passe a conter uma string). Imprima na saída de erro padrão usando um único `fprintf` o ID do processo seguido de uma vírgula e da string contida em `mybuf`. Execute o programa com diferentes valores de `n` e `nchars`. Observe os resultados. Pressione a tecla `Enter` várias vezes e continue digitando até que todos os processos tenham finalizado. Explique o comportamento do programa criado.

B. Process Fans [2,0]

Desenvolver um programa `runsim.c` que permite a criação de processos *batch*. Esse programa recebe um argumento na linha de comando especificando o número máximo de processos simultâneos. Siga o roteiro a seguir para implementar `runsim`.

1. Escreva um programa chamado `runsim` que recebe um argumento da linha de comando.
2. Faça a checagem do argumento (com impressão da mensagem de “*usage*” em caso de omissão do argumento).
3. Defina a variável `pr_limit` que é inicializada com o valor recebido da linha de comando. Esta variável especifica o número máximo de processos filhos (ativos) ao mesmo tempo.

4. Defina a variável `pr_count`, inicializada com 0. Essa variável indica o número de processos filhos ativos.
5. Execute o loop a seguir até que o “end-of-file” seja lido da entrada padrão.
 - a. Se `pr_count` for igual a `pr_limit`, espere o término de um filho e decremente `pr_count`.
 - b. Leia uma linha da entrada padrão (`fgets`) de no máximo `MAX_CANON` caracteres e execute um programa correspondendo à linha de comando lida criando-se um processo filho (veja função `makeargv()` no ANEXO no final do trabalho).
 - c. Incremente `pr_count` para contabilizar o número de processos filhos ativos.
 - d. Verifique se algum filho já terminou, sem bloquear caso nenhum filho tenha terminado. Decrementa `pr_count` para cada filho finalizado.
6. Após a leitura do “end-of-file” da entrada padrão, aguarde que todos os demais processos filhos terminem, e finalmente finalize o programa.

Escreva um programa chamado `testsim` que recebe dois argumentos da linha de comando: o *sleep time* e o *repeat factor*. O fator de repetição é o número de vezes que `testsim` executa um loop. Neste loop, `testsim` dorme durante o *sleep time* especificado e então imprime uma mensagem com o ID do seu processo na saída de erro padrão.

Utilize `runsim` para executar várias cópias de `testsim`. Para isso, crie um arquivo de teste chamado `testing.data` que contenha os comandos a serem executados. Por exemplo:

```
testsim 5 10
testsim 8 10
testsim 4 10
testsim 13 6
testsim 1 2
```

Execute o programa através do comando a seguir:

```
runsim 2 < testing.data
```

C. Divisão de tarefas [2,0]

Considere os seguinte dados:

- o um vetor de mil inteiros
- o um número inteiro a verificar se ele existe no vetor

Crie um programa “`busca.c`” onde o processo pai deverá criar 4 processos filhos e cada um deles deverá retornar 1 ou 0 no caso de encontrar, ou não, o valor em questão (este valor deverá ser passado como argumento na linha de comando) no seu setor do vetor. O processo pai processará informação enviada pelos processos filhos e apresentará o resultado final.

PS1: Preencha o vetor com valores aleatórios entre 0 e 2000 (veja `rand()` e `srand()`).

PS2: Cada filho deverá percorrer setores diferentes do vetor.

D. Escrevendo um Programa Shell [4,0]

Suponha que você trabalhe para uma empresa que está desenvolvendo o Super Ultra Sistema Operacional (SUSO). Um dos objetivos do SUSO é fornecer diferentes shells adaptados para diferentes usuários. Você está encarregado de desenvolver o *simple shell* (**spsh**), que será programado dentro de “**spsh.c**”.

O chefe do SUSO quer que o seu **spsh** seja implementado da seguinte forma

1. Use a variante “*execvp*” da SVC *exec*.
2. O prompt do **spsh** deve ser “spsh.” concatenado com o path do diretório corrente (use *getenv*(“*PWD*”). Por exemplo: “spsh./home/john/so/>”

Como o chefe do SUSO estava de bom humor, ele não solicitou que o **spsh** fosse capaz de tratar “Pipes” (ex: *ps -aux | grep john*) nem redirecionamento de entrada/saída (ex: *wc -l < infile*). No entanto ele exigiu que o shell fosse capaz de executar comandos em background (onde o último parâmetro deve ser ‘&’).

Nota: Nesta parte do trabalho, apenas pede-se a listagem do código MUITO BEM comentada!

Bibliografia

W. Richard Stevens, *Advanced Programming in the UNIX Environment, 1st Edition* (Cap. 8)

Kay A. Robbins, Steven Robbins, *UNIX Systems Programming: Communication, Concurrency and Threads, 2nd Edition* (Cap 1-3).

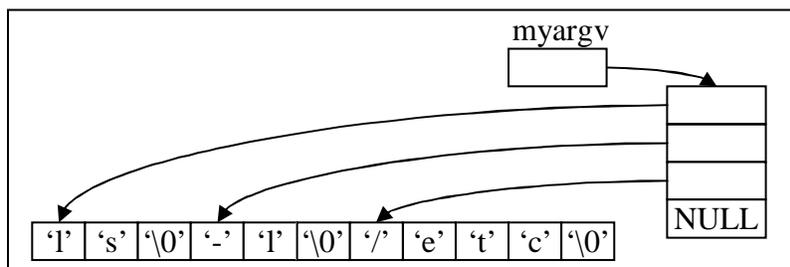
----- ANEXO -----

Função *makeargv* que permite construir o array *argv*[]*, apontado pelo argumento ****argvp*, a partir da string apontada por **s*, usando os delimitadores definidos por **delimiters*. A função retorna o número de tokens encontrados na string.

Se por exemplo fazemos:

```
char comand[] = "ls    -l    /etc";
char delim[] = " \t";
char **myargv;
if (makeargv(comand, delim, &myargv) == -1)
    perror("Child failed to construct argument array");
```

em caso de sucesso, após a chamada, teremos:



```

#include <errno.h>
#include <stdlib.h>
#include <string.h>

int makeargv(const char *s, const char *delimiters, char ***argvp) {
    int error;
    int i;
    int numtokens;
    const char *snew;
    char *t;

    if ((s == NULL) || (delimiters == NULL) || (argvp == NULL)) {
        errno = EINVAL;
        return -1;
    }
    *argvp = NULL;
    snew = s + strspn(s, delimiters); /* snew is real start of string */
    if ((t = malloc(strlen(snew) + 1)) == NULL)
        return -1;
    strcpy(t, snew);
    numtokens = 0;
    if (strtok(t, delimiters) != NULL) /* count the number of tokens in s */
        for (numtokens = 1; strtok(NULL, delimiters) != NULL; numtokens++) ;

    /* create argument array for ptrs to the tokens */
    if ((*argvp = malloc((numtokens + 1)*sizeof(char *))) == NULL) {
        error = errno;
        free(t);
        errno = error;
        return -1;
    }

    /* insert pointers to tokens into the argument array */
    if (numtokens == 0)
        free(t);
    else {
        strcpy(t, snew);
        **argvp = strtok(t, delimiters);
        for (i = 1; i < numtokens; i++)
            *((*argvp) + i) = strtok(NULL, delimiters);
    }
    *((*argvp) + numtokens) = NULL; /* put in final NULL pointer */
    return numtokens;
}

```