EA878 - Laboratório de Micro e Mini: Software
Atividade 7

1.Tema

Gerenciamento de processos em um sistema multiprogramado.

2.Objetivos

Familiarização com chamadas ao sistema (*system calls*) no ambiente UNIX. Implementação de processos num sistema multiprogramado. Introdução às chamadas ao sistema para manipulação e sincronização de processos.

3.Chamadas ao Sistema

O sistema operacional UNIX foi o pioneiro na adoção de muitos conceitos utilizados no projeto de sistemas operacionais multiprogramados. O núcleo do UNIX é um programa relativamente pequeno, residente na memória e habilitado a manipular os dados pertencentes ao núcleo. Seus códigos são escritos em C (a maior parte) e em linguagem de máquina. Ele provê serviços de:

- **gerenciamento de processos**: inclui o gerenciamento de memória onde os processos são alocados e a intercomunicação entre os processos.
- **gerenciamento de arquivos**: inclui o gerenciamento de entrada/saída que estabelece a conexão entre os processos e os sistemas de arquivos.
- **tratamento de erros** (**perror**): é definida uma variável global **errno** para indicar a causa de um erro.

Quando o computador executa uma instrução do núcleo, a CPU trabalha em um modo especial chamado de **modo núcleo**, no qual ela tem acesso a todos os recursos do sistema. Mas quando o computador executa uma instrução de outros programas que não pertencem ao núcleo, a CPU trabalha em um modo chamado **modo usuário**, no qual ela tem algumas restrições de acesso que são dependentes de sua arquitetura. Isso é feito para garantir uma segurança maior aos processos e permitir um controle mais eficiente pelo núcleo. A comutação do modo usuário para o modo núcleo ocorre sempre quando há solicitação de serviços do núcleo. É importante observar que mudar de contexto (do modo usuário para o núcleo e de volta ao modo usuário) é uma operação custosa e existem técnicas como *threads* (*light weight processes)* que procuram minimizar este custo.

Um serviço do núcleo é sempre solicitado através de uma ou mais **chamadas ao sistema** (*system calls*). Assim como o interpretador de comandos é a interface entre o usuário e o sistema operacional, as chamadas ao sistema constituem a interface entre programas aplicativos e o sistema operacional. Sob o ponto de vista de códigos em C, não existe nenhuma diferença entre as chamadas ao sistema e as chamadas a programas convencionais. A cada chamada ao sistema há um identificador interno associado. Através deste identificador, o núcleo consegue ter acesso ao endereço das instruções correspondentes na **tabela de vetores de chamadas ao sistema**. Um desvio é então feito a este endereço. A última instrução de uma chamada ao sistema é sempre o retorno ao código do usuário (operando no modo usuário).

A maioria das chamadas ao sistema retorna valores do tipo inteiro para indicar a condição de término das chamadas (bem sucedido ou com erro). No arquivo </usr/include/sys/errno.h> encontram-se todos os códigos de erro do sistema. Algumas chamadas ao sistema não podem sofrer preempção por executarem tarefas críticas. Além disso, para evitar que a CPU fique ociosa aguardando, por exemplo, o término de uma operação de entrada/saída de um processo, o núcleo bloqueia este processo e só quando a operação termina, um sinal de interrupção é enviado para reativá-lo. Daí, ele volta a competir pelo uso da CPU.

Os tipos de dados e/ou os valores dos argumentos utilizados pelas chamadas ao sistema são definidos em arquivos separados, que se encontram normalmente no diretório </usr/include>. Portanto, dependendo da chamada é necessário adicionar aos programas linhas de comando como:

```
#include <fcntl.h>          /* qualificadores de arquivos */
#include <signal.h>         /* codigos de sinais de interrupcao */
#include <sys/errno.h>      /* codigos de erro */
#include <sys/types.h>      /* definicao de tipos de dados
                               e macros utilizados pelo sistema */
#include <sys/msg.h>        /* tipos de dados utilizados pela chamadas ao sistema
```

```
                                relacionadas com a passagem de mensagem */
#include <sys/sem.h>          /* tipos de dados utilizados pelas chamadas ao
                                sistema relacionadas com a definicao de semaforos */
```

Nesta atividade serão enfatizadas operações como bifurcação, execução, escalonamento, suspensão, extinção, sincronização e tratamento de sinais de processos no sistema UNIX.

4.Criação de Processos

As únicas entidades ativas no sistema UNIX são os processos, os quais competem por recursos oferecidos pelo sistema operacional (acesso a discos, periféricos e, principalmente, CPU) e interagem com outros processos. Todo processo possui duas importantes propriedades derivadas de sua natureza seqüencial:

- a velocidade com que um processo é executado não interfere no resultado de sua execução;
- se um processo for executado mais de uma vez com os mesmos dados, ele passará precisamente pela mesma seqüência de instruções e fornecerá o mesmo resultado.

Um processo é um "ambiente" de execução (ou uma abstração de um programa em execução) constituído por três segmentos:

- **segmento de instruções** (*text segment*): contém os códigos executáveis de um programa, de tamanho fixo e que podem ser compartilhados por mais de um processo já que não são modificados durante a execução;
- **segmento de dados do usuário** (*user-data segment*): abrange os dados do programa (com ou sem valor inicial), sendo que seu tamanho pode variar dinamicamente (para suportar a alocação dinâmica de espaço em memória, o sistema operacional provê chamadas ao sistema para modificar o tamanho do segmento de dados alocado a cada processo) e não podem ser compartilhados entre os processos;
- **segmento de dados ou pilha do sistema** (*system-data segment* ou *stack segment*): contém as variáveis do ambiente em que o processo é executado e os argumentos de chamada do mesmo; seu tamanho é ajustado dinamicamente pelo núcleo e há um segmento privativo para cada processo.

Todo processo possui uma entrada na **tabela de processos** (residente no espaço de memória reservado ao núcleo). Cada entrada contém as seguintes informações necessárias para o núcleo gerenciá-lo durante a sua existência:

- parâmetros de prioridade para o escalonador poder decidir sempre o próximo processo a ser executado,
- referências aos endereços dos seus três segmentos via **tabela de regiões de processo**,
- formas de tratamento de sinais gerados ou captados por ele, e
- o seu número de identificação (**pid**), a identificação do usuário (**uid**) responsável por sua execução, o seu estado, entre outros dados relacionados à identificação do processo.

Só quando um processo é extinto, sua entrada na tabela de processos é liberada para que ela possa ser reutilizada por outros processos a serem executados posteriormente.

A **tabela de regiões de processo** apontada pela tabela de processos contém as entradas da **tabela de regiões**. Esta **tabela de regiões** é que contém efetivamente os endereços físicos dos segmentos de processo, ou seja, o acesso às regiões é feito de forma indireta para facilitar compartilhamento quando necessário.

O núcleo mantém ainda em memória uma outra estrutura de dados necessária para quando um processo está fisicamente na memória e em estado ativo ou em execução (veja definição destes estados mais abaixo). Essa estrutura, considerada uma extensão da tabela de regiões, é denominada **estrutura de usuário** (*user area*). Ela inclui informações como registradores, estado corrente da chamada ao sistema, descritores dos arquivos utilizados, tempo de CPU consumido, entre outras.

4.1.Árvore de processos

O sistema operacional UNIX é um sistema multiusuário e multiprogramado, no qual múltiplos processos independentes podem coexistir (inclusive vários processos de um mesmo usuário). Normalmente, quando é dado *boot* no sistema, o primeiro processo criado é **swapper** (*pid*=0), que gerencia o escalonamento de processos. Imediatamente, dois outros processos são criados:

1. o processo **init** (*pid*=1) que lê o arquivo </etc/ttys> para saber quais são os terminais ligados ao sistema e prover informações de cada um deles, além de criar para cada terminal o processo **getty**, que fica aguardando o identificador de um usuário inscrito no arquivo </etc/passwd>. Quando uma identificação válida é detectada, o processo **init** é bloqueado e é criado o processo **login**, que por sua vez aciona um interpretador de comandos, por exemplo **sh**, para interpretar e executar os comandos entrados pelo usuário. Quando o processo-filho é extinto, **init** é desbloqueado e aciona novamente **getty;**
2. o processo **pagedaemon** (*pid*=2) que verifica periodicamente a quantidade de *frames* livres na memória. Se o número for muito pequeno (abaixo de um determinado nível preestabelecido), ele procura automaticamente limpar a memória para liberar mais *frames*.

Para criar um novo processo, o UNIX gera uma cópia exata (**bifurcação**) do processo original (chamada ao sistema: `fork`). Denominamos o processo original de **processo-pai** e o novo processo, **processo-filho**. Após a bifurcação, eles executam o mesmo código concorrentemente, mas cada um com a sua própria área de dados. Para substituir o processo copiado (processo-filho) por um novo processo, é preciso sobrescrever os códigos, os argumentos e as variáveis do ambiente copiadas do processo-pai com as informações do novo processo. Isso é feito através da chamada ao sistema denominada `exec`. O identificador (*pid*) do processo-filho e o do seu processo-pai podem ser obtidos respectivamente através de chamadas ao sistema como `getpid` e `getppid`.

Todo processo tem um único pai, mas pode ter vários filhos, determinando assim uma **árvore de processos**. Portanto, exceto pelo primeiro processo (processo 0, na raiz da hierarquia) qualquer outro processo é criado através da chamada `fork`.

O sistema UNIX distingue seis estados de um processo:

- **Em execução**: o processo tem a posse da CPU;
- **Ativo**: o processo não está sendo executado, mas está pronto e competindo pela posse da CPU;
- **Bloqueado**: o processo não precisa ocupar a CPU, pois está aguardando pela liberação de outros recursos ou por dados solicitados a algum periférico;
- **Suspenso**: o processo é "congelado" por um sinal e não mais compete pela CPU;
- **Ocioso**: o processo acaba de ser criado pelo `fork` e não foi ativado ainda;
- **Terminal**: o processo está por ser extinto, mas aguarda que o processo-pai seja notificado disso através de uma chamada ao sistema denominada `wait`. (Obs: os filhos de um processo que for extinto, são *adotados* pelo processo **init**.

5.Concorrência e sincronização

Os processos em UNIX freqüentemente compartilham outros recursos além da CPU. Geralmente, o recurso só pode atender a uma única requisição por vez, e deve atendê-la do princípio ao fim sem interrupção, de modo a evitar inconsistências. Em situações como essas, a parte do código que manipula o recurso compartilhado é denominada *região crítica*. O código que implementa uma região crítica deve obedecer a uma série de restrições:

- dois processos não podem estar simultaneamente executando regiões críticas referentes a um mesmo recurso compartilhado (garantia de mútua exclusão);
- a garantia de mútua exclusão deve ser independente da velocidade relativa dos processos e do número de CPUs;
- nenhum processo executando fora de regiões críticas pode bloquear outro processo;
- nenhum processo deve esperar um tempo arbitrariamente longo para poder executar uma região crítica.

Os algoritmos que procuram implementar as restrições acima são classificados de acordo com o modo com que esperam pela autorização de entrada numa região crítica: espera ocupada (usando a CPU durante a espera) ou bloqueada (não competindo pela CPU). Todo algoritmo de mútua exclusão possui duas funções delimitadoras. A primeira função é chamada quando o processo deseja iniciar a execução de uma região crítica. Quando esta função retorna, o processo está apto a executar a região crítica. Ao final da execução da região crítica, o processo chama a segunda função, que então libera o recurso compartilhado para outro processo.

Para permitir que regiões críticas associadas a recursos compartilhados distintos possam ser executadas ao mesmo tempo, a cada recurso compartilhado é associado um identificador, e as duas

funções que compõem o algoritmo de garantia de mútua exclusão possuem este identificador como parâmetro.

Os mecanismos de mútua exclusão com espera bloqueada são preferidos, por não ocuparem a CPU durante o período de espera. Um dos métodos mais simples de implementação de espera bloqueada é a utilização do par de chamadas ao sistema `sleep` e `wakeup` (que não fazem parte da biblioteca padrão da linguagem C). A chamada `sleep` é responsável por mudar o estado de um processo em execução para bloqueado. O processo bloqueado volta a tornar-se ativo quando um outro processo o desbloqueia através da chamada `wakeup`. Infelizmente, deixar ao programador a responsabilidade de execução destas chamadas pode levar a situações inusitadas como, por exemplo, ao estado de *deadlock*, em que todos os processos encontram-se bloqueados.

A utilização de variáveis do tipo semáforo faz com que as operações de bloqueio e reativação de processos, denominada **sincronização de processos**, não seja realizada diretamente pelo usuário. Estas variáveis contam o número de vezes que a operação `wakeup` foi realizada. É necessário definir as operações `DOWN` e `UP`. A operação `DOWN` verifica se o valor do semáforo é maior que 0. Em caso afirmativo, decrementa este valor e o processo continua. Se o valor é 0, o processo é bloqueado. A operação `UP` incrementa o valor do semáforo quando o recurso for liberado. Se um ou mais processos estiverem bloqueados sob aquele semáforo, um deles é escolhido pelo sistema para completar a operação `DOWN` que o fez bloquear (emitindo-lhe um `wakeup`).

No sistema UNIX as informações de semáforos estão no núcleo para que elas sejam compartilhadas entre processos distintos. Além disso, para garantir a atomicidade de certos grupos de operações sobre os semáforos, estas operações são sempre realizadas no modo núcleo. As facilidades oferecidas pelo sistema para manipular semáforos em nível de chamadas ao sistema são descritas a seguir.

- Um semáforo é criado (ou acessado, caso ele já exista) através da chamada ao sistema **semget**. Cada semáforo tem um vetor de valores não-negativos associado a ele. O núcleo utiliza as seguintes estruturas para descrevê-lo:

```
#include <sys/types.h>
#include <sys/ipc.h>
struct semid_ds {
     struct ipc_perm    sem_perm;      /* permissoes de operacoes            */
     struct sem        *sem_base;      /* endereco do vetor de semaforos     */
     ushort            sem_nsems;      /* numero de semaforos no vetor       */
     time_t            sem_otime;      /* tempo da ultima operacao sobre ele */
     time_t            sem_ctime;      /* tempo da ultima modificacao        */
};
```

onde cada elemento do vetor apontado por <sem_base> contém as seguintes informações:

```
struct sem {
     ushort     semval;      /* valor do semaforo                          */
     short      sempid;      /* pid do ultimo processo que operou semval   */
     ushort     semncnt;     /* qtde de processos que aguardam que semval
                                seja incrementado                          */
     ushort     semzcnt;     /* qtde de processos que aguardam que semval=0 */
};
```

- Para manipular o conjunto de valores de um semáforo é provida uma chamada ao sistema denominada **semop**. A seqüência indivisível de operações a ser executada sobre o seu conjunto de valores é passada por um vetor onde cada elemento especifica uma operação:

```
struct sembuf {
      ushort    sem_num;    /* indice do valor no vetor de valores que deve ser
                               manipulado                                    */
      short     sem_op;     /* tipo de operacao: liberacao do recurso (sem_op>0),
                               alocacao do recurso (sem_op<0) e leitura (sem_op=0) */
      short     sem_flag;   /* opcoes: IPC_NOWAIT e SEM_UNDO)                */
};
```

- Para manipular ou acessar individualmente um valor de um semáforo é disponível a chamada ao sistema **semctl** que suporta as seguintes operações:

GETVAL	ler o valor da posição especificada no vetor de valores
SETVAL	escrever na posição especificada no vetor o valor dado
GETPID	ler o último processo que manipulou o valor especificado

GETNCNT	ler o número de processos que aguardam o incremento de semval
GETZCNT	ler o número de processos que aguardam semval=0
GETALL	ler os valores de um semáforo
SETALL	escrever em todas as posições do vetor de valores
IPC_RMID	remover o semáforo
IPC_SET	alterar as permissões

Os semáforos tornam simples a proteção de recursos compartilhados, mas não garantem que não haja *deadlocks*. Além disso, sua estratégia é susceptível a:

- condições de corrida (*race conditions*), já que **semget** e **semctl** são duas chamadas distintas e
- acúmulo de semáforos "inúteis" no núcleo, se eles não forem explicitamente removidos através da chamada **semctl**.

O conceito de **monitores** representa uma solução de alto nível para o problema de *deadlock*, já que os processos podem ter acesso aos procedimentos do monitor, mas não a suas estruturas internas. Com isso, o problema de *deadlock* pode ser evitado em nível de compilação. Infelizmente, são raras as linguagens de programação que os incoporam. Recentemente, as linguagens orientadas a objeto, como Java, têm oferecido suporte para implementação de monitores.

6.Escalonamento de processos

A implementação de paralelismo aparente em um sistema multiprogramado não é uma tarefa simples. Conceitualmente, cada processo tem uma CPU virtual própria. A tarefa de passar periodicamente a posse efetiva da CPU real de processo a processo é função do sistema operacional. A parte do sistema operacional que toma esta decisão é chamada escalonador e os mecanismos de tomada de decisão são denominados algoritmos de escalonamento.

Desde já, deve ficar claro que a velocidade de execução de um processo é função da quantidade de processos competindo pela CPU. Sendo assim, os processos em UNIX não devem ser programados com base em considerações intrínsecas de tempo.

Os critérios, possivelmente conflitantes, que devem ser observados por um algoritmo de escalonamento são:

- **progresso**: garantir que cada processo tenha acesso à CPU e possa avançar;
- **eficiência**: manter a CPU ocupada o máximo possível;
- **tempo de resposta**: minimizar o tempo de resposta na execução de processos, principalmente os interativos (editores, planilhas, aplicações de multimídia, etc);
- **tempo de espera**: minimizar o tempo de espera de serviços não-interativos (compilação, impressão, etc);
- **vazão**: maximizar o número de processos executados em um intervalo fixo de tempo.

Para suportar melhor os processos interativos e concorrentes, o sistema UNIX distingue dois níveis para o algoritmo de escalonamento. O primeiro nível escolhe, dentre os que estão na memória e prontos para execução, o próximo candidato a ocupar a CPU e o segundo nível é responsável pelo movimento de processos entre a memória real e a memória virtual em disco (**swapping**), de modo a garantir que todos os processos tenham chances de execução.

A estratégia de escalonamento adotada pelo UNIX para o primeiro nível é uma combinação de políticas de *Round Robin* e de prioridades. O tempo da CPU (número de quanta) alocado a cada processo depende do seu grau de importância. Na maioria dos sistemas, um quantum corresponde a 1/50 seg., sendo que seu valor é crítico: se for muito pequeno, diminui a eficiência da CPU, pois passa a haver uma sobrecarga no chaveamento de recursos entre os processos; se for muito grande, provoca uma degradação na resposta a processos interativos.

A fórmula básica usada pelo UNIX para estimar a prioridade de cada processo em modo usuário é:

```
prioridade = (K1/tempo de CPU usado recentemente) + (K2/nível)
```

onde `K1` e `K2` são calculados com base nas características de cada sistema e `nível` é um valor atribuído pelo usuário através do comando `nice`. Note que a equação reflete o que se espera do sistema: a prioridade de um processo diminui se ele ocupa muito a CPU e/ou se o usuário assim determinar.

O algoritmo de escalonamento apresenta os seguintes passos:

- em cada segundo, o escalonador calcula as prioridades de cada processo ativo e os reorganiza em diferentes filas de prioridade;
- em cada quantum, o escalonador seleciona um processo ativo na fila de maior prioridade e aloca a ele a CPU (política de prioridades);
- quando extinguir o seu tempo e o processo não chegar ao final das suas instruções, ele é colocado no final da fila (política de Round Robin);
- se o processo é bloqueado durante a execução, o escalonador seleciona imediatamente um outro processo e aloca a este novo processo à CPU;
- se o processo retorna de uma chamada ao sistema durante o seu tempo da CPU e existe um processo de prioridade maior pronto para executar, então o processo de prioridade menor é trocado pelo de prioridade maior (preempção);
- em cada interrupção de relógio de sistema, o contador de interrupções é incrementado. A cada quatro interrupções, o escalonador recalcula a prioridade do processo em execução. Esta política evita que um mesmo processo tome a CPU por um tempo muito longo.

7.Interrupções de Processos

Os eventos que interrompem o fluxo normal de um programa são conhecidos como **interrupções** e esses eventos são enviados ao processo através de **sinais** (*signals*). No arquivo <usr/include/sys/signal.h> encontram-se todos os tipos de sinais reconhecíveis pelo núcleo do sistema. Com exceção do sinal SIGKILL, os sinais podem ser ignorados ou gerar um desvio a uma função de tratamento *default* ou a uma especificada pelo usuário.

As informações que suportam o tratamento de sinais em cada processo são armazenadas na tabela de processos:

- **bitmap de sinais pendentes**: é um vetor de bits na tabela de processos. Cada bit setado corresponde a um sinal pendente a ser processado (observe que o número de sinais pendentes de um mesmo tipo não é considerado);
- **vetor de tratamento de sinais**: é um vetor na tabela de regiões que descreve a forma de tratamento de cada sinal pendente (ignorar=1; ação *default*=0 e ação especificada pelo usuário=endereço da função). Na chamada ao comando `exec`, todos os sinais ignorados pelo processo-pai permanecem ignorados no processo-filho e os outros recebem o atributo 0 (ação *default*) no processo-filho.

O sistema UNIX dispõe de uma chamada ao sistema denominada `kill`, responsável por enviar um sinal a um processo ou mais processos de um mesmo grupo, ou seja, setar o bit correspondente no *bitmap* de sinais pendentes. Dispõe também de uma chamada denominada `signal` para captar um sinal e especificar a forma do seu tratamento (SIG_IGN→ignora o sinal especificado; SIG_DFL→usa a função de gerenciamento *default* e <nome da função>→usa a função definida pelo usuário). Quando um processo bloqueado recebe um sinal, ele é reativado para processar este sinal.

O núcleo do sistema UNIX só verifica os sinais pendentes no *bitmap* quando ocorre a comutação do modo núcleo para o modo usuário ou então quando o processo é bloqueado ou reativado. Esta estratégia de implementação não é apropriada para suportar aplicações em tempo real.

O algoritmo implementado no sistema UNIX para tratamento de um sinal segue os seguintes passos:

- salva o contador de programa e o apontador de pilha (*stack pointer*) do processo;
- anexa uma "nova área de pilha" à pilha do processo e atualiza o apontador de pilha;
- atribui ao contador de programa o endereço da função gerenciadora do sinal, como se tivesse ocorrido um desvio programado;
- quando o processo retorna ao modo usuário, a função gerenciadora do sinal é executada;
- retorna ao endereço da instrução imediatamente antes do desvio.

8.Atividades

(a) (1,0) Bifurcação e adoção de um processo: execute o programa abaixo e explique em detalhes o seu comportamento, documentando no relatório os resultados da execução.

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

```
main()
{
int pid;
    printf("pid(processo)=%d, pid(processo-pai)=%d\n",getpid(),getppid());
    pid = fork();
    if (pid == 0)
    {
        printf("\nProcesso-filho:\n");
        printf("pid(processo-filho)=%d, pid(processo-pai)=%d\n",getpid(),getppid());
        sleep(6);
        printf("...apos aproximadamente 6 segundos...\n");
        printf("pid(processo-filho)=%d, pid(processo-pai)=%d\n",getpid(),getppid());
    }
    else
    {
        printf("\nProcesso-pai:\n");
        printf("pid(processo-filho)=%d\n",pid);
        sleep(1);
        printf("...apos aproximadamente 1 segundo...\n");
    }
    printf ("Processo %d terminou!\n\n",getpid());
}
```

(b) (1,0) Estado terminal de um processo: execute o programa em *background* (usando o operador &) e verifique o estado do processo-filho (use o comando `ps ux`) enquanto o processo-pai estiver executando. Documente os testes e explique o funcionamento do programa em seu relatório.

```
#include <stdio.h>

main()
{
int pid;
    printf ("pid(processo)=%d, pid(processo-pai)=%d\n",getpid(),getppid());
    pid = fork();
    if (pid == 0)
    {
        printf("\nProcesso-filho:\n");
        printf("pid(processo-filho)=%d, pid(processo-pai)=%d\n",getpid(),getppid());
    }
    else
    {
        printf("\nProcesso-pai:\n");
        printf("pid(processo-filho)=%d\n",pid);
        sleep(10);
    }
    printf("Processo %d terminou!\n",getpid());
}
```

(c) (1,0) Sequenciamento da execução do processo-pai e processos-filho: explique o funcionamento deste programa e apresente o motivo pelo qual a mensagem "Processo xxxx terminou!" não apareceu para o primeiro processo-filho. Observe que, com o uso de *wait*, foi evitado que os processos-filho permanecessem no estado terminal. Por quê (execute em *background* e use o comando `ps ux`)?

```
#include <stdio.h>

main()
{
int pid1, pid2, filho1, filho2;
int estado1, estado2;
    printf("pid(processo)=%d, pid(processo-pai)=%d\n",getpid(),getppid());
    pid1 = fork();
    if (pid1 == 0)
    {
        printf("Primeiro processo-filho:\n");
        printf("pid(1o.processo-filho)=%d, pid(processo-
                        pai)=%d\n",getpid(),getppid());
        sleep(10);
        exit(20);
    }
    else
    {
```

```
        pid2 = fork();
        if (pid2 == 0)
        {
            printf("Segundo processo-filho:\n");
            printf("pid(2o. proc-filho)=%d, pid(processo-
                        pai)=%d\n",getpid(),getppid());
        }
        else
        {
            filho1 = wait(&estado1);
            filho2 = wait(&estado2);
            printf ("Processo-pai:\n");
            printf ("pid(1o. processo-filho)=%d\n",pid1);
            printf ("pid(2o. processo-filho)=%d\n",pid2);
            printf("O processo %d terminou com estado=%d\n",filho1,estado1>>8);
            printf("O processo %d terminou com estado=%d\n",filho2,estado2>>8);
        }
    }
    printf ("Processo %d terminou!\n",getpid());
    exit(30);
}
```

(d) (1,0) Carregamento do segmento de instruções de um processo: explique o funcionamento deste programa e teste-o, observando que o mesmo aceita argumentos. Execute o programa mais de uma vez com os mesmos argumentos e explique o comportamento apresentado pelo tempo de execução do processo-filho. Execute o programa com e sem a linha 15 e explique a diferença de comportamento.

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/time.h>

main(int argc,char *argv[])
{
int pid;
int estado;
struct timeval tv1, tv2;
double delta;
    pid = fork();
    if (pid == 0)
    {
        printf("pid(processo-filho)=%d\n",getpid());
        execlp("cal","cal",argv[1],argv[2],NULL);     /* linha 15 */
    }
    else
    {
        gettimeofday(&tv1,NULL);
        wait(&estado);
        gettimeofday(&tv2,NULL);
        printf ("pid(processo-pai)=%d\n",getpid());
        delta = ((double)(tv2.tv_sec) + (double)(tv2.tv_usec)/1e6) -
                ((double)(tv1.tv_sec) + (double)(tv1.tv_usec)/1e6);
        printf("O tempo de execucao do processo %d e' %lf,", pid, delta);
        printf(" terminando com estado=%d\n", estado>>8);
    }
    printf ("Processo %d terminou!\n", getpid());
    exit(30);
}
```

(e) (1,0) Especificação de formas de tratamento de um sinal: faça vários testes com o programa abaixo e justifique o comportamento apresentado pelo mesmo, documentando no seu relatório.

```
#include <stdio.h>
#include <sys/signal.h>

void ger_nova();

main()
{
void (*ger_antiga)();

    /* ctrl-c == sinal SIGINT */
    printf ("Primeiro trecho de tratamento de ctrl-c \n");
```

```
    signal(SIGINT, SIG_IGN);
    sleep(5);
    printf("\n");
    printf ("Segundo trecho de tratamento de ctrl-c \n");
    signal (SIGINT, ger_nova);
    sleep(5);
    printf("\n");
    printf ("Terceiro trecho de tratamento de ctrl-c \n");
    ger_antiga = (void *) signal(SIGINT, SIG_DFL);
    sleep(5);
    printf("\n");
    printf ("Quarto trecho de tratamento de ctrl-c \n");
    signal(SIGINT, ger_antiga);
    sleep(5);
    printf ("\nTchau!\n");
}

void ger_nova()
{
    printf ("O sinal SIGINT foi captado. Continue a execucao!\n");
}
```

(f) (1,0) Envio de um sinal (suspensão e reativação de processos): execute o código abaixo e explique o seu funcionamento. O que acontece quando eliminamos a linha com a chamada de sistema *wait3*? Qual é a diferença básica entre *wait* e *wait3*? Justifique suas respostas.

```
#include <stdio.h>
#include <sys/wait.h>
#include <sys/signal.h>

main(){
int pid1, pid2,i;
long j;
void ger_sinal();
       signal(SIGCHLD,ger_sinal);
       pid1 = fork();
       if (pid1 == 0){
               while (1){
                  printf("- Processo %d esta' ativo/em execucao\n",getpid());
                  sleep(1);
                  printf("1");
                  fflush(0);
                  for(j=0;j<1000000;j++);
               }
       }
       pid2 = fork();
       if (pid2 == 0){
               while (1){
                  printf("o Processo %d esta' ativo/em execuca \n",getpid());
                  sleep(1);
                  printf("2");
                  fflush(0);
                  for(j=0;j<1000000;j++);
               }
       }
       sleep(5);
       printf("\n%d:vou parar o 1o.",getpid());
       fflush(0);
       kill (pid1,SIGSTOP);
       sleep(5);
       printf("\n%d:vou parar o 2o.",getpid());
       fflush(0);
       kill (pid2,SIGSTOP);
       sleep(5);
       printf("\n%d:vou continuar o 1o.",getpid());
       fflush(0);
       kill (pid1,SIGCONT);
       sleep(5);
       printf("\n%d:vou matar o 2o.",getpid());
       fflush(0);
       kill (pid2,SIGCONT);
       sleep(5);
       printf("\n%d:vou matar o 1o.",getpid());
       fflush(0);
       kill (pid1,SIGINT);
       sleep(5);
       printf("\n%d:vou matar o 2o.",getpid());
       fflush(0);
       kill (pid2,SIGINT);
```

```
        /* Sinais são assíncronos,  logo, não se sabe quando vão "chegar". Se o programa
acabar logo após essas chamadas kill(pidX,SIGINT), os sinais vão chegar tarde demais (o
programa já terá finalizado) e não serão capturados!
        */

        for(i = 0; i < 5; i++){
          printf("\n%d: esperando sinal \"chegar\"...",getpid());
          fflush(0);
          for(j=0;j<1000000;j++);
          sleep(1);
        }
}

void ger_sinal(){

  int pid;
  int estado;

 /* O loop abaixo é um detalhe importante. Um sinal SIGCHLD (em sistemas BSD) pode estar
"representando" múltiplos filhos e então o sinal SIGCHLD significa que um ou mais filhos
tiveram seu estado alterado.
  */

  do{
    pid = wait3(&estado,WNOHANG,NULL);
    printf("%d: Um sinal de mudanca do estado do processo-filho %d foi captado!\n",
           getpid(),pid);
    if (pid > 0){
       printf("O processo-filho %d foi extinto! Estado=%d\n", pid,estado>>8);
    }
  } while( pid > 0 && printf("em loop!  "));

  signal(SIGCHLD,ger_sinal);
}
```

Atenção:

- quando o processo pai recebe e vai atender à chamada SIGCHLD causada pelo término de um processo filho, pode ocorrer que outros processos-filho também terminem e seus sinais SIGCHLD não sejam devidamente captados pelo processo-pai. Em outras palavras, um sinal SIGCHLD pode estar representando o término de um ou mais filhos. Portanto, é recomendável que o processo-pai execute a chamada wait3( ) tantas vezes quantas forem necessárias para garantir que ele irá tomar ciência do término de todos os processos-filho que já tenham terminado. Uma forma de fazer isso é colocando wait3( ) em um loop para que sua execução seja repetida enquanto ela estiver retornando um valor de pid válido. Se não houver processos-filho terminados, a chamada wait3( ) irá retornar -1 ou 0 (devido ao WNOHANG) e o programa deve seguir então seu curso normal.
- poderão surgir dúvidas sobre funcionamento do programa acima, mas uma leitura mais atenta do manual online da função *signal()* e algumas experimentações irão facilitar o entendimento do que está ocorrendo. Por exemplo, um trecho do manual diz que " If signal() or sigset() is used to set SIGCHLD's disposition to a signal handler, SIGCHLD will not be sent when the calling process's children are stopped or continued.". Para entender melhor o que está ocorrendo, é recomendável fazer algumas modificações no programa e experimentar diversas situações:
  - em lugar da chamada *sleep(),* experimente colocar um loop após a chamada *kill()* de forma que haja tempo de receber um sinal do processo-filho que acaba de terminar; veja o exemplo abaixo:

    ```
    .                                   .                                   .
    kill                                                        (pid2,SIGINT);
    for(i          =          0;       i          <          3;          i++){
            printf("Processo   pai   esta'   esperando   um   sinal   chegar...   \n");
            sleep(1);
    }
    ```

  - experimente trabalhar apenas com o processo-pai e um filho para simplificar as mensagens apresentadas e facilitar o entendimento.

9. Atividades do projeto de servidor WEB

- Nesta etapa a atividade relacionada ao desenvolvimento do servidor WEB consiste na introdução de concorrência no servidor. Cada requisição recebida por ele será tratada por um processo independente.
- Imediatamente após a chamda *accept*, o servidor deverá testar se já há N processos-filho em execução.

- Caso afirmativo, ele deverá enviar uma mensagem de indisponibilidade ao cliente (sugerindo que este tente mais tarde), fechar o *socket* e retornar ao *accept*.
- Caso ainda não tenha atingido o limite de N processos-filho o servidor deverá executar a chamada *fork*.
- No processo-pai, o servidor deverá incrementar o contador de processos-filho, fechar o *socket* retornado por *accept* e retornar ao *accept*.
- No processo-filho, o servidor deverá processar a requisição HTTP e terminar com uma chamada *exit*.
- A cada término de processo-filho, o servidor deverá decrementar o contador de processos-filho. Esta operação deve ser feita em uma função que captura o sinal de término do processo-filho (logo o contador deverá ser uma variável global) por meio da chamada *wait3***.** Consulte o manual da chamada para saber como verificar se a mudança de estado se refere ao término do processo-filho.
- Faça diversos testes com esta nova versão do servidor e apresente-os em seu relatório.
- Apresente o código-fonte do seu servidor, documentando com detalhes as partes novas que foram introduzidas nesta atividade.
- Esta atividade vale 4,0 pts.

Observações:

- **Sobre interrupções e chamadas de sistema**

  Em algumas versões do UNIX o envio de um sinal a um processo bloqueado em uma chamada de sistema faz com que a chamada de sistema seja interrompida antes do gerenciador de sinal ser invocado. Neste caso a chamada retorna erro (-1) com código de erro (errno) EINTR. Por exemplo, um processo bloqueado em uma chamada *accept* que recebe um sinal de término de processo-filho pode apresentar este comportamento. Para evitar que este isso comprometa a lógica do programa que manipula sinais é comum "proteger" as chamadas de sistema da seguinte forma:

```
...
do {
   sock = accept(s, (struct sockaddr *)&client, &namelen));
   } while(sock == -1 && errno == EINTR);   /* interrupcao da chamada -
                                               reinicie */
if(sock == -1) {   /* erro nao causado por interrupcao */
  perror("accept()");
  exit(0);
}
...
```

- **Sobre fechamento de conexões no HTTP 1.1**

  O protocolo HTTP versão 1.1 sempre mantém uma conexão TCP/IP ativa mesmo após atendida a requisição que a originou. Isto permite que uma mesma conexão atenda múltiplas requisições, eliminando assim o *overhead* de abertura e fechamento de conexões. Caso o cliente nao deseje manter a conexão ativa, a mensagem HTTP de requisição contém o parâmetro *Connection* com o valor *Close*. Entretanto, nao é recomendado que o servidor HTTP 1.1 permita que conexões permaneçam ativas indefinidamende, pois conexões são recursos limitados de comunicação. Neste caso, o servidor pode (e deve) encerrar uma conexão caso a mesma fique inativa por um período de tempo (5 segundos, por exemplo). Veja o inicio do Cap. 8 da RFC2616.

  No caso desta atividade, ao processar uma requisição o processo deve inspecionar o parâmetro *Connection*. Caso seu valor seja *Close*, o processo deve atender a requisição e encerrar a conexão (chamada de sistema *close*). Caso contrário, o processo deve atender a requisição e mantê-la aberta por um intervalo de tempo (vamos ignorar o campo *Keep-alive,* se ele existir na requisição). O problema que surge é: como e onde controlar este tempo?

  O usual é controlar este tempo na chamada *read*. Como *read* é bloqueante e não possui um parâmetro de *timeout*, a única forma de evitar o bloqueio é controlar o *timeout* através de outra chamada de sistema, no caso a chamada **select**. A idéia é simples: ao invés de bloquear um processo indefinidamente em uma chamada, bloqueia-se o mesmo por um período desejado usando uma chamada *select*. Esta chamada permite ao programador passar até três lista de descritores de arquivo (por exemplo, sockets) e um parâmetro de *timeout*. As listas são descritores monitorados para fins de leitura, escrita e condições

especiais. Funções utilitárias (iniciando por FD_) permitem manipular estas listas. A chamada retorna:

- o número de descritores fornecidos à chamada que estejam "prontos" para leitura ou escrita, ou apresentem condições especiais;
- 0, em caso de *timeout*;
- -1, em caso de erro.

O servidor deve utilizar a chamada *select* para evitar que o cliente mantenha uma conexão aberta indefinidamente sem o envio de requisições. Neste caso, o socket poderá ser manipulado pelo processo-filho da seguinte forma:

```
while(1) {
  /* invoca select com timeout de x segundos */
  flag = select(...);
  if(flag <  0) continue; /* erro no select - continue */
  if(flag == 0)   /* timeout - encerra conexão e termina */
    {
    close(s);
    exit(0);
    }
  /* socket com dados - leitura nao bloqueara' */
  mbytes = read(s, ...);
  ...
}
```

Veja exemplo de uso da chamada *select* no programa teste_select.c abaixo. Consulte a página de manual referente à chamada *select* para detalhes sobre sua utilização.

```
# Programa teste_select.c
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
  int n, c;
  fd_set fds;
  struct timeval timeout;
  int fd;

  fd = 0;              /* stdin */
  FD_ZERO(&fds);       /* zera a lista de descritores */
  FD_SET(fd, &fds);    /* adiciona descritor (stdin) aa lista */
  timeout.tv_sec = 5;  /* 5 segundos */
  timeout.tv_usec = 0; /* 0 microsegundos */

  /* leia o teclado, aguardando no maximo 5 segundos */
  n = select(1, &fds, (fd_set *)0, (fd_set *)0, &timeout);
  if(n > 0 && FD_ISSET(fd, &fds))   /* FD_ISSET testa se descritor
                                       especifico esta' pronto */
    {
      c = getchar();  /* nao ira bloquear */
      printf("Caractere %c teclado.\n", c);
    }
  else if(n == 0)    /* timeout */
        {
        printf("Nada foi teclado em 5s.\n");
        }
       else perror("select()");
  exit(1);
  }
```

Referências Bibliográficas

1. Cardozo, E., M.F. Magalhães, L.F. Faina e I.L.M. Ricarte, *Sistemas Operacionais*. Notas de Aula do Curso EA-877, DCA/FEEC/UNICAMP, 1996.
2. Tanenbaum, A.S. *Modern Operating Systems*. Prentice-Hall International Editions, 1992.
3. Rochkind, M.J. *Advanced UNIX Programming*. Prentice-Hall, Englewood Cliffs, 1985.
4. Glass, G. *UNIX for Programmers and Users - A Complete Guide*. Prentice-Hall International Editions, 1993.
5. Stevens, W.R. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, 1990.
6. Christian, K. Sistema Operacional UNIX. Editora Campus, Rio de Janeiro, 1985
7. -, Manual de Referência do UNIX.