EA872 - Laboratório de Programação de Software Básico

Atividade 7

Gerenciamento de processos em um sistema multiprogramado

1. Objetivos

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

2. Chamadas de 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 da alocação de processos no processador e na memória e a comunicação entre eles;
- **gerenciamento de arquivos**: inclui o gerenciamento de de arquivos abertos e fechados, mapeamento destes em memória RAM (para agilizar acesso), mapeamento de recursos de entrada/saída em arquivos, gerenciamento de conexão entre os processos e os sistemas de arquivos;
- **gerenciamento de memória**: inclui o gerenciamento de áreas de memória ocupada e livre, controle de acesso a áreas de memória protegidas, associação de áreas de memória com processos; gerenciamento de áreas do disco como expansão da memória real em RAM (memória virtual);
- **gerenciamento de entrada e saída**: inclui o gerenciamento dos circuitos periféricos de entrada/saída (teclado, mouse, monitor, interfaces de rede com e sem fio, portas serias e paralelas) e de sensores (incluindo o relógio de tempo real), estabelecendo a conexão destes com os processos e os arquivos.

Uma outra função importante do núcleo é o **tratamento de erros** com o uso de `perror`, uma função de gerenciamento de erros que define uma variável global chamada `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 todas as instruções de seu repertório e, consequentemente, 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 restrições de acesso a algumas instruções (que dependem da 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 (de um processo para outro e entre os modos núcleo e usuário) é uma operação custosa e existem técnicas como *threads* (*light weight processes)* que procuram minimizar este custo.

Um serviço do núcleo é normalmente executado em modo núcleo e é sempre solicitado através de uma ou mais **chamadas de sistema** (*system calls*). Assim como o interpretador de comandos é a interface entre o usuário e o sistema operacional, as chamadas de 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 de sistema e as chamadas a programas convencionais. Para cada chamada de 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 de sistema**.

Um desvio é então feito a este endereço. A última instrução de uma chamada de sistema é sempre o retorno ao código do usuário (operando no modo usuário).

A maioria das chamadas de sistema retorna valores do tipo inteiro para indicar a condição de término das chamadas (bem sucedido ou com erro). No arquivo </usr/include/asm-generic/errno.h> (e em outros similares em </usr/...> encontram-se todos os códigos de erro do sistema.

Algumas chamadas de sistema não podem sofrer preempção (isto é, não podem ser interrompidas antes de concluírem uma parte do trabalho pré-definida) por executarem tarefas críticas para o sistema como um todo. Se forem interrompidas antes de concluírem o que estão fazendo, corre-se o risco de que outras chamadas acabem destruindo o trabalho que ela estava fazendo.

Por outro lado, para evitar que a CPU fique ociosa aguardando, por exemplo, o término de uma (custosa) 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 e voltar a competir pelo uso da CPU.

Os tipos de dados e/ou os valores dos argumentos utilizados pelas chamadas de sistema são definidos em arquivos separados, que se encontram normalmente no diretório </usr/include>. Portanto, dependendo da chamada é necessário adicionar linhas de comando aos programas como as mostradas a seguir. Observe que pode haver mudanças nos locais onde tais arquivos estão armazenados, dependendo da versão de UNIX ou de Linux utilizado.

```
#include <fcntl.h>              /* qualificadores de arquivos */
#include <signal.h>             /* codigos de sinais de interrupcao */
#include <linux/errno.h>        /* codigos de erro */
#include <linux/types.h>        /* definicao de tipos de dados
                                   e macros utilizados pelo sistema */
#include <linux/msg.h>          /* tipos de dados utilizados pela chamadas de sistema
                                   relacionadas com a passagem de mensagem */
#include <linux/sem.h>          /* tipos de dados utilizados pelas chamadas ao
                                   sistema relacionadas com a definicao de semaforos */
```

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

3. 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 deve possuir duas importantes propriedades derivadas de sua natureza sequencial:

- 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 sequência 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*): é o segmento de memória que contem 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*): é o segmento de memória que contém os dados do programa (com ou sem valor inicial), os quais não podem ser compartilhados entre os processos. Seu tamanho pode variar dinamicamente e, para suportar a alocação dinâmica de espaço em memória, o sistema operacional provê chamadas de sistema para modificar o tamanho do segmento de dados alocado a cada processo;
- **segmento de dados do sistema ou de pilha** (*system-data segment* ou *stack segment*): é o segmento de memória que 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 ele também é 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 gerenciar um processo durante a sua existência:

- parâmetros de prioridade para o escalonador poder decidir qual 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 memória onde reside um processo. Assim, o acesso às regiões de memória do processo é feito de forma indireta para facilitar compartilhamento de segmentos entre processos quando necessário.

O núcleo mantém ainda em memória uma outra estrutura de dados necessária para um processo que esteja 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 de sistema, descritores dos arquivos utilizados, tempo de CPU consumido, entre outras.

3.1.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 (troca) de processos na CPU. Em seguida, dois outros processos são criados:

1. o processo **init** (*pid*=1) que lê o arquivo </etc/ttys> ou </etc/securetty> 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 a autenticação de um usuário inscrito no arquivo </etc/passwd>. Quando uma identificação válida é detectada, o processo **init** é bloqueado e é criado um processo que executa o código de **login**, que por sua vez aciona um interpretador de comandos, por exemplo **sh**, para interpretar e executar os comandos entrados pelo usuário. Quando este último processo é extinto, **init** é desbloqueado e aciona novamente **getty;**
2. o processo **pagedaemon** (ou **khugepaged**) que, entre outras coisas, verifica periodicamente a quantidade de *frames* (quadros) livres na memória. Se o número estiver 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 de 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-cópia (processo-filho) por um novo processo, é preciso sobrescrever os códigos, os argumentos e as variáveis do ambiente copiados do processo-pai com as informações do novo processo. Isso é feito através da chamada de sistema denominada `exec`. O identificador (*pid*) do processo-filho e o do seu processo-pai podem ser obtidos respectivamente através de chamadas de 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 na competição 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 está "congelado" por um sinal e não 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 de sistema denominada `wait` (Obs: os filhos de um processo que é extinto são "adotados" pelo processo **init**).

4. Concorrência e sincronização

Os processos em UNIX frequentemente compartilham outros recursos além da CPU. Geralmente, o recurso só pode atender 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 (autorizado) 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 de 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, denominadas **sincronização de processos**, não sejam realizadas 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` e fazendo-o continuar.

No sistema UNIX as informações de semáforos estão no núcleo para que elas sejam compartilhadas entre processos distintos. Além disso, para garantir a atomicidade (característica de indivisibilidade: operações são todas executadas ou nenhuma é) 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 de sistema são descritas a seguir.

- Um semáforo é criado (ou acessado, caso ele já exista) através da chamada de 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 de sistema denominada **semop**. A sequência indivisível de operações a ser executada sobre o conjunto de valores do semáforo 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 usa-se a chamada de 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 pid do 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 que podem disputar o controle do semáforo e gerar resultados inconsistentes e
- acúmulo de semáforos "inúteis" no núcleo, se 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 incorporam. As linguagens orientadas a objeto, como Java, têm oferecido suporte para implementação de monitores.

5. 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, já que não temos como controlar facilmente o número de processos nesta competição.

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.

O algoritmo de escalonamento de processos tem evoluído constantemente e pode variar bastante de uma versão para outra do kernel (núcleo) do SO. Nas discussões abaixo são apresentadas ideias básicas que norteiam os algoritmos de escalonamento, as quais são implementadas de formas distintas em diferentes versões de kernel.

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 processos 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 no 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 executa 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 CPU a ele (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 CPU a este novo processo;

- se o processo retorna de uma chamada de sistema durante o seu tempo na 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 N (por exemplo, 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.

6. 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/signal.h>** encontram-se todos os tipos de sinais reconhecíveis pelo núcleo do sistema (uma outra lista pode ser vista com **man -s 7 signal**). 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 de 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 do 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.

7. Atividades a serem desenvolvidas no laboratório

As atividades práticas a serem realizadas nesta atividade requerem um conhecimento do gerenciamento de processos feito pelo Unix. Por isso, preparamos um questionário inicial que deve ser respondido e entregue durante o horário do laboratório. Responda no Moodle as questões abaixo usando respostas curtas com suas próprias palavras. Cada questão vale 0,2 pontos.

1. O que é execução em modo núcleo?
2. Qual é a diferença de execução em modo núcleo para execução em modo usuário?
3. Qual é o objetivo de se ter estes dois modos de execução em um SO?
4. O que são chamadas de sistema?
5. Dê um exemplo de situação em que um programa que você está fazendo precisa recorrer a uma chamada de sistema e um exemplo em que uma chamada deste tipo não é necessária.

6. Um processo é criado em Unix por meio de uma de duas chamadas possíveis: fork() e exec(). Qual é a principal diferença entre elas?
7. A memória ocupada por um processo é dividida em partes, que são gerenciadas de formas distintas. Que partes são estas e qual é a principal característica de cada uma?
8. Por que se diz que há uma árvore de processos em Unix?
9. Quando um processo é criado como cópia de outro, como o SO consegue distinguir entre eles?
10. Qual a diferença entre os estados “Ativo” e “Em execução” de um processo?

8. Atividades para o relatório

8.1 Responda às seguintes questões complementares sobre conceitos relativos a gerência de processos. Cada resposta correta vale até 0,2 pts.

1. O que é preciso ocorrer para que um processo em estado terminal libere os recursos que estava ocupando?
2. Por que alguns trechos de programas são chamados de “região crítica”?
3. Em unix, dois recursos compartilhados distintos podem ser acessados simultaneamente por dois processos distintos (um recurso para cada processo)? Por que?
4. Que providências devem ser tomadas por um processo antes e após a execução de um código de região crítica?
5. O que são e para que servem os semáforos em um SO?
6. O que significa fazer uma operação DOWN em um semáforo?
7. Escolha e explique a motivação de um dos objetivos de um escalonador de processos.
8. Como o comando **nice** do Unix afeta a prioridade de execução de um processo?
9. Cite um exemplo de uma situação em que seja necessário enviar um sinal a um processo e justifique sua escolha.
10. O que é preciso fazer em um programa para que, durante sua execução, um sinal enviado a ele não seja ignorado?

8.2 Compile, execute e estude os códigos desta seção (fornecidos em arquivo .zip), de forma a poder responder as questões específicas de cada um.

(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>
#include <stdlib.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>
#include <stdlib.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>
#include <stdlib.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 16 e explique a diferença de comportamento.

```
#include <stdio.h>
#include <stdlib.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 16 */
    }
    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 <stdlib.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 <stdlib.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.

