Universidade Federal de Minas Gerais **Sistemas Operacionais**

Instituto de Ciências Exatas **1° Semestre de 2017**

Departamento de Ciência da Computação **Prof. Flávio Figueiredo**

Alunos: Caíque Bruno Fortunato **Matrícula:** 2013062731

Pâmela Carvalho da Silva 2013073474

Documentação do TP1

Shell + PS + Sinais + Módulos

Introdução

Este trabalho prático tem como principal propósito fixar o conteúdo sobre Processos, aprendido na disciplina de Sistemas Operacionais. Em particular, o foco está no entendimento de conceitos de Pipes, Estruturas de Processos do Kernel e Sinais na prática.

Consequentemente, a principal contribuição do trabalho é entender o funcionamento e tratamento de processos nos Sistemas Operacionais modernos e atuais, mais especificamente no Linux, onde foi realizada a implementação.

Como aprendido na disciplina, um processo possui três etapas: Criação (fork e exec), Execução e Término, e vai ser através deste conceito que a primeira parte do trabalho será desenvolvida. Em seguida, será criada uma PS-Tree que exibirá na tela todos os processos em execução em forma de árvore: do processo pai aos seus respectivos filhos. Já na terceira parte do trabalho será implementado um TOP que exibirá os vinte primeiros processos em execução (informando PID, Nome, Estado e o Usuário que o criou). Na quarta e última etapa será permitido que o comando TOP envie sinais e, por fim, na última parte será criado um módulo PS-tree utilizando estruturas de processos no kernel do Linux.

Desenvolvimento

Todo o trabalho foi implementado em linguagem C em duas máquinas: Uma com sistema operacional Linux na versão 4.4.0-72-generic com processador Intel® Core™ i5 5° geração possuindo 8GB de memória RAM e 1TB de memória de disco. Já a outra máquina possui sistema operacional Linux na versão 4.4.0-72-generic com processador Intel® Core™ i7 5° geração possuindo 8GB de memória RAM e 500GB de memória de disco.

Parte 1: Desenvolvendo um Shell Básico

O principal objetivo da primeira parte do programa é desenvolver um Shell Básico atentando para três principais funcionalidades:

- Modificar a função runcmd;
- Implementar comandos com redirecionamento de entrada e saída;
- Implementar pipes para sequenciamento de comandos;

O esqueleto do shell foi fornecido na especificação, bastando apenas modificar algumas partes do código para atender as especificações listadas acima. Para isso, esta seção será dividida em três etapas, especificando as alterações feitas em cada para implementar as funcionalidades necessárias para o funcionamento do shell.

Modificar a função runcmd

Para implementação foi utilizado o execvp com os seguintes parâmetros: *execvp(ecmd->argv[0], ecmd->argv);* ou seja, conforme especificado pelo site recomendado na especificação: *int execvp(const char *file, char *const argv[]);* Além disso, também foi criado um tratamento de erro utilizando a função perror: *perror(ecmd->argv[0]);*

A escolha da função *execpv* foi pela maior facilidade da implementação e pelos parâmetros. que já existem no código. Além disso, a função funcionou sem maiores problemas.

Comandos com redirecionamento de E/S

Para implementação foi utilizado funções de arquivo como open e close, que possuem especificações que podem ser encontradas no Linux Die.

Primeiro foi liberado o arquivo fd que estava aberto, fechando-o com a função close: *close(rcmd->fd)*; Com o arquivo liberado, a função open foi utilizada para abrir o arquivo no modo de read/write *open(rcmd->file, rcmd->mode);* Por fim, com o arquivo aberto, foi utilizada a função *runcmd(rcmd->cmd);* para executar a instrução.

Pipes para sequenciamento de comandos

Talvez, a implementação mais complicada foi o sequenciamento de comandos, devido a utilização dos comandos fork. A especificação será detalhada abaixo através de etapas:

- A condicional *if (pipe(p) ≤ 0)* verifica se é possível criar o pipe sabendo que 0 é a saída esperada e qualquer valor abaixo de 0 indica erro.
- É criado um Fork() utilizando a função já implementada *fork1()* que executa a instrução *runcmd()* para “esquerda” e outro *fork1()* que executa a instrução *runcmd()* para a “direita”. Para a criação são criadas duas condicionais: *if (fork1() == 0)* e if *(fork1() == 0)* onde cada uma vai tratar um dos “lados” do Fork.
- Para cada lado é executado as instruções da imagem abaixo:

```
if (fork1() == 0)
{
  close(1);
  dup(p[1]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd->left);
}

// Exec. do comando a direita utilizando o fork implementado
if (fork1() == 0)
{
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd->right);
}
```

- Por fim, para indicar que o arquivo nao esta sendo mais utilizado para concluir o gerenciamento de arquivos é utilizado os comandos: *close(p[0])* e *close(p[1])* e a função *wait(NULL);*

Parte 2: Lendo o /proc/ para fazer um PS-Tree

Nesta etapa foi desenvolvido um comando chamado *myps* que deverá ser executado utilizando o programa gerado pela parte 1, ou seja, através do executável myshell, gerado através da compilação do mesmo.

A função do *myps* é simular uma PS-Tree, ou seja, um comando Unix que mostra os processos ativos em forma de árvore, sendo uma alternativa mais visual para o comando ps.

Como aprendido na disciplina, o sistema Unix não permite florestas, todo mundo é filho do *init*, se um processo ficar órfão o *init* adota o mesmo. Tal conceito foi importante e crucial para o desenvolvimento do código que gera a árvore de processos ativos.

Estratégia

Para desenvolver uma PS-Tree foi adotada a seguinte estratégia que será especificada abaixo. Maiores informações sobre os arquivos e diretórios

podem ser encontradas no tópico *Demais informações*, presentes nessa seção.

1. Começar com o processo pai (PID: 1), exibindo o nome do mesmo.
2. Através do processo anterior lido, no caso o processo com PID1, acessar o arquivo: *"/proc/PID/task/PID/children"* e ler quais são os filhos do mesmo.
3. Verificar se o processo possui alguma pilha de thread no diretório: *"/proc/PID/task*, exibindo o nome do mesmo com uma tabulação a mais.
4. Chamar a função de maneira recursiva.

Implementação

Para a implementação foram criadas as seguintes funções:

1. **numProc(int proc, int itensTask[])** Esta função possui como principal objetivo preencher um vetor (itensTask[]) com os *PID*s dos processos que estão dentro da pasta *Task*, retornando a quantidade de itens dentro do vetor. De maneira informal, a função funciona como um comando *ls*, que lista os itens dentro da pasta *Task* e manipula-os conforme especificado.

2. **preenche_vetor(int proc, int processos[])** Esta função possui a principal responsabilidade de entrar no diretório (que será especificado abaixo) e preencher um vetor chamado *processos[]* com os filhos do processo passado como parâmetro. Por exemplo: se a função recebe o id do processo 1 ele entra no diretório: *"/proc/PID/task/PID/children"* e preenche o vetor de *processos[]* com os filhos do mesmo.

3. **imprimeNomeProcesso(int proc)** De maneira intuitiva, esta função recebe um PID e retorna o nome do processo, que está encontrado dentro do arquivo *stat*, presente no diretório: *"/proc/PID/stat".* O nome

do processo é o segundo item presente dentro do stat. (o primeiro é o PID).

4. **imprime_Pstree(int i, int ntabs, int lidos[])** Por fim, a principal função do código constrói a árvore de processos de maneira recursiva, utilizando todas as funções listadas acima. A ideia é: dado um vetor de processos[], *itensTask[]* (conforme já especificados), a função irá calcular a quantidade de tabs necessários (variável *ntabs*) de acordo com o parâmetro que é passado pela função recursiva e exibir o nome do processo. Por exemplo, se um processo tem id 1 e possui os filhos 2 e 3, a função irá:

    a. Exibir o 1 (processo pai) e incrementar o número de tabs
    b. Exibir o processo 2 e verificar se possui filhos, se tiver, incrementa o número de tabs e exibe o filho, se não (que é o caso), volta para a chamada anterior.
        i. Entra na pasta *Task* e verifica se o processo possui alguma thread executando, se tiver, incrementa o número de tabs e exibe os nomes dos processos que estão na pasta. (Exemplo básico: Navegador de internet possivelmente terá itens na pasta Task)
    c. Exibe o processo 3 e realiza os passos do item anterior.

Demais informações:

Observação: O */proc* é um diretório de arquivos especiais do linux que lista os processos em execução. Este arquivo embora possa ser lido como um arquivo normal, na verdade não é um arquivo em disco e sim um file handle para um arquivo especial que lista informações do kernel

- **/proc/[pid]/task/[tid]/children:** É um arquivo contendo uma lista separada por espaços de tarefas que são filhas do processo onde cada filho é representada por um *tid*.

- **/proc/[pid]/task/[tid]/:** É um diretório que contém uma pilha de um thread (onde o <tid> é um ID de thread). Cada processo tem no mínimo 1 thread, se tiver mais o código exibe, conforme especificação. (Motivo pela criação da função *numProc(int proc, int itensTask[]).*

- **/proc/[pid]/task/:** É um arquivo que contém uma lista contendo informações sobre um processo. Para este trabalho nos interessa apenas as primeiras informações presentes no arquivo:

    - (1) *PID:* O ID do processo
    - (2) *Comm:* O nome do arquivo do executável, entre parênteses.
    - (3) *State:* Um dos seguintes caracteres, indicando o processo:

R	Running
S	Sleeping in an interruptible wait
D	Waiting in uninterruptible disk sleep
Z	Zombie
T	Stopped (on a signal) or trace stopped
t	Tracing stop (Linux 2.6.33 onward)
W	Paging (only before Linux 2.6.0)
X	Dead (from Linux 2.6.0 onward)
x	Dead (Linux 2.6.33 to 3.13 only)
K	Wakekill (Linux 2.6.33 to 3.13 only)
W	Waking (Linux 2.6.33 to 3.13 only)
P	Parked (Linux 3.9 to 3.13 only)

Resultados

Neste tópico serão listados alguns resultados provenientes da execução do comando *myps* exibindo a árvore de processos ativos.

Parte 3: Lendo o /proc/ para fazer um PS-Tree

Nesta etapa foi desenvolvido um comando chamado *htop* que deverá ser executado utilizando o programa gerado pela parte 1, ou seja, através do executável myshell, gerado através da compilação do mesmo.

A função do *htop* é simular um comando TOP, ou seja, um comando Unix que mostra os processos em execução no sistema em forma de lista, oferecendo um conjunto de estatísticas sobre o estado geral do sistema.

Conforme especificação, foram limitados somente os 20 primeiros processos para apareceram no comando, sendo que serão exibidas as seguintes informações:

PID	User	PROCNAME	Estado
1272	flavio	yes	S
1272	root	init	S

Estratégia

Para desenvolver um TOP foi adotada a seguinte estratégia: lê o arquivo stat correspondente a cada processo, identificado o PID, Nome do Processo e o Estado. Além disso, a utilização de uma função que identifique o autor do processo.

Implementação

Para a implementação foram criadas as seguintes funções (comentadas somente as mais relevantes):

- **imprimeCabecalho( ):** Imprime o cabeçalho do TOP
- **imprimeAutorProc(int idProc):** Recebe o id do processo e:
  - Cria uma struct do tipo stat e uma variável sb
  - Chama a função stat: stat(file, &sb);
  - Cria outra struct: struct passwd *pw = getpwuid(sb.st_uid);
  - Recupera o nome do autor do processo (pw->pw_name)
- **leProcessos(int cont):** Entra no diretório *"/proc/%PID/"* e procura pelos 20 primeiros processos existentes. Para cada processo, o algoritmo acessa o arquivo *"/proc/%PID/stat"* exibindo: PID, Nome do processo e Estado (o arquivo stat foi especificado na seção anterior). Além disso, a função *imprimeAutorProc(int idProc)* é chamada para exibir o autor do processo. A função é atualizada a cada segundo.
- **leProcessosMandaSinais(int cont):** Exibe o TOP de maneira semelhante com o método anterior, mas sem atualizar a cada segundo, dando chance do usuário matar o processo.

Resultados

Neste tópico serão listados alguns resultados provenientes da execução do comando *topzera:*

Parte 4: Sinais

Implementada no código da parte 3 como a função *enviar_sinal (int pid, int sinal)*. A mesma utiliza uma função da biblioteca <signal.h> para enviar sinal para o processo: kill(pid,NOME_DO_SINAL);.

A função recebe o PID (ID do processo) e um sinal. Conforme descrito na documentação implementamos o sinal 1 - SIGHUP. No entanto, implementamos também outros sinais famosos como: 2 - SIGINT, 15 - SIGTERM, 9 - SIGKILL e 20 - SIGTSTP.

Parte 5: Módulo Linux que funciona similar a uma PS-Tree

A quinta e última parte deste trabalho consiste em criar um módulo Linux que funciona similar a uma PS-Tree. Conforme orientado na especificação, foi utilizado o código de exemplo do myps e os macros indicados.

Para melhor funcionamento, novamente optamos por uma função recursiva que implementa a árvore de processos:

```c
void
pstree(struct task_struct *init_task)
{
    struct task_struct *next_task;
    struct list_head *list;
    list_for_each(list, &init_task->children) {
        next_task = list_entry(list, struct task_struct, sibling);
        printk(KERN_INFO " Name: %s Pid: [%d] State: [%ld]\n", next_task->comm, next_task->pid, next_task->state);
        pstree(next_task);
    }
}
```

A função utiliza um ponteiro para percorrer os filhos do init_task e outro para armazenar a cabeça da lista. Através do list_for_each os filhos são percorridos e a função é chamada recursivamente de modo a percorrer a árvore.

Para cada item é exibido: o Nome, PID e Estado, de modo que o resultado pode ser visualizado abaixo:

```
[14900.911590] Loading module ps
[14900.911597]  Name: systemd Pid: [1] State: [1]
[14900.911601]  Name: systemd-journal Pid: [225] State: [1]
[14900.911604]  Name: systemd-udevd Pid: [253] State: [0]
[14900.911607]  Name: systemd-timesyn Pid: [751] State: [1]
[14900.911610]  Name: cron Pid: [774] State: [1]
[14900.911613]  Name: systemd-logind Pid: [779] State: [1]
[14900.911616]  Name: avahi-daemon Pid: [781] State: [1]
[14900.911619]  Name: avahi-daemon Pid: [944] State: [1]
[14900.911622]  Name: snapd Pid: [795] State: [1]
[14900.911624]  Name: whoopsie Pid: [800] State: [1]
[14900.911627]  Name: thermald Pid: [803] State: [1]
[14900.911630]  Name: dbus-daemon Pid: [805] State: [1]
[14900.911633]  Name: NetworkManager Pid: [828] State: [1]
[14900.911636]  Name: dnsmasq Pid: [1280] State: [1]
[14900.911639]  Name: dhclient Pid: [10929] State: [1]
[14900.911642]  Name: acpid Pid: [829] State: [1]
[14900.911644]  Name: rsyslogd Pid: [831] State: [1]
[14900.911647]  Name: accounts-daemon Pid: [837] State: [1]
[14900.911650]  Name: ModemManager Pid: [838] State: [1]
[14900.911653]  Name: irqbalance Pid: [943] State: [1]
[14900.911655]  Name: lightdm Pid: [971] State: [1]
[14900.911658]  Name: Xorg Pid: [988] State: [1]
[14900.911661]  Name: lightdm Pid: [1139] State: [1]
[14900.911664]  Name: upstart Pid: [1455] State: [1]
[14900.911667]  Name: upstart-udev-br Pid: [1541] State: [1]
[14900.911670]  Name: dbus-daemon Pid: [1542] State: [1]
[14900.911673]  Name: window-stack-br Pid: [1554] State: [1]
[14900.911675]  Name: upstart-dbus-br Pid: [1573] State: [1]
[14900.911678]  Name: upstart-dbus-br Pid: [1575] State: [1]
[14900.911680]  Name: upstart-file-br Pid: [1584] State: [1]
[14900.911683]  Name: ibus-daemon Pid: [1589] State: [1]
[14900.911686]  Name: ibus-dconf Pid: [1625] State: [1]
[14900.911689]  Name: ibus-ui-gtk3 Pid: [1626] State: [1]
[14900.911691]  Name: ibus-engine-sim Pid: [1647] State: [1]
[14900.911694]  Name: gpg-agent Pid: [1607] State: [1]
[14900.911697]  Name: gvfsd Pid: [1611] State: [1]
[14900.911700]  Name: gvfsd-fuse Pid: [1616] State: [1]
[14900.911702]  Name: ibus-x11 Pid: [1630] State: [1]
[14900.911705]  Name: bamfdaemon Pid: [1646] State: [1]
[14900.911708]  Name: hud-service Pid: [1663] State: [1]
[14900.911710]  Name: unity-settings- Pid: [1665] State: [1]
[14900.911713]  Name: syndaemon Pid: [1717] State: [1]
```

Conclusões finais

Com a finalização do trabalho foi possível alcançar diferentes objetivos propostos, entre eles um maior entendimento da teoria e, principalmente, prática de Processos em Sistemas Operacionais modernos.

Antes da implementação do TP o conhecimento da dupla sobre Sistemas Operacionais era bem limitado, então o conhecimento adquirido foi grande e ajudou

muito para entender melhor os conceitos. Contudo, consequentemente, devido ao pouco conhecimento anterior houveram muitas dificuldades como: entender conceitos como *fork* e *exec* na primeira parte do TP, a estrutura do diretório *Proc* e também a quinta parte: o módulo Linux que funciona de maneira similar a uma PS-Tree.

Por fim, a dupla considera que os principais pontos pretendidos pelo trabalho foram alcançados e que muito conhecimento foi adquirido.

Bibliografia

1. Linux Die: https://linux.die.net/
2. TLDP: http://tldp.org/LDP/lpg/node11.html
3. Pipes: http://www2.cs.uregina.ca/~hamilton/courses/330/notes/unix/pipes/pipes.html
4. Jair Junior - Chamadas de sistema: http://www.jairjunior.eng.br/artigos/familia-exec-de-chamadas-de-sistema/