Sistemas Operacionais

Trabalho – Fundamentos UNIX

Prof. Daniel Sundfeld
daniel.sundfeld@unb.br

Agenda

- O sistema Unix
- Criação de Processos
- Carga de arquivo executável
- Sinais
- Término de processos
- Processos Zumbis

UNIX

- Final da década de 60: o sistema operacional MULTICS (MULTiplexed Information andComputing Service foi projetado para ser um sistema operacional padrão, pelos melhores pesquisadores da época (MIT, GE, Bell Labs). Sucesso acadêmico, mas um fracasso comercial

UNIX

- UNICS: sistema projetado na Bell Labs (abandonado)
- Reescrito em uma linguagem de alto nível (B)
- Ritchie cria a linguagem C e reescreve o UNIX
- Bell Labs doa o código às universidades
- Novas versões surgiram

UNIX

- Sistema em linguagem de alto nível foi amplamente aceito pela comunidade e facilitou a portabilidade do sistema para múltiplas arquiteturas
- Em 1976 é publicado o primeiro UNIX do meio universitário, UNIX versão 6
- AT&T lança versões comerciais UNIX system 3 em 1982

UNIX

- UNIX system V é uma versão melhorada do system III e vira um sucesso
- O UNIX BSD é desenvolvido em Berkeley a partir da versão 6 do UNIX: pilha TCP/IP é implementada e permite comunicação em rede

UNIX

- O sistema Linux foi desenvolvido em 1991 por Linus Torvalds
- Código do Linux compartilhado no grupo de discussão do sistema do Tanenbaum Minix
- Projeto ativo e amplamente utilizado até hoje, focado exclusivamente no desenvolvimento do Kernel

UNIX

- O Linux precisa de software para ser útil a um computador, utiliza o código do projeto GNU
- Ao se utilizar e customizar as versões de diversos códigos, é criada uma distribuição Linux
- GNU/Linux != Linux

UNIX

- Hoje, UNIX é um termo utilizado para uma grande família de Sistemas Operacionais
- Família BSD: FreeBSD, OpenBS
- Família Solaris: SunOS
- Família Mac OS X
- Esses sistemas possuem estruturas de diretórios e comandos de terminal similares

UNIX

- Comando para listar processos e verificar seu estado: ps

```
[user@station ~]$ ps
    PID TTY          TIME CMD
   9385 pts/0    00:00:00 bash
  10042 pts/0    00:00:00 ps
```

- Imprimir tudo:

```
ps -ax -o pid,ppid,command
```

UNIX

- Cada processo possui um identificador único chamado de “process id”: pid
- Todo processo possui um processo pai, e então contém o “parent process id”, ppid
- Em C (ou C++), pode usar o cabeçalho <unistd.h> para ter acesso à API do sistema operacional POSIX e usar uma série de funções para manipulação de processos

UNIX

- Obter o de pid e ppid

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

int main()
{
    pid_t meupid = getpid();
    printf("Pid atual do processo: %d\n", meupid);

    pid_t pidpai = getppid();
    printf("Pid do pai do processo: %d\n", pidpai);
    return 0;
}
```

Criação de Processos

- Os processos podem ser criados tipicamente em um terminal, onde são digitados os comandos
- Dentro de um código-fonte, é possível executar o comando system
- Essa chamada irá criar um subprocesso de shell (/bin/sh) que irá efetuar a execução do comando

Criação de Processos

- Um programa em C que lista todos os arquivos da pasta atual

```c
#include <stdlib.h>

int main()
{
    return system("ls -la");
}
```

Criação de Processos

- Esse código não é portátil e funcionará exclusivamente em sistemas UNIX
- No Windows, é comum a prática de utilizar o system("pause") para parar a execução do código C ao final dele, quando se utiliza uma IDE que invoca o terminal Windows (que fecha rapidamente)

Criação de Processos

- Outras linguagens (scripts Bash) são mais apropriadas quando se deseja executar comandos (ou diversos comando) em um terminal, sem que seja necessário criar um binário executável para isso
- Para se criar novos processo em C, recomenda-se criar processos utiliando o `fork()`

Criação de Processos

- A se chamar, um novo processo é criado e quase todo o conteúdo da tabela de processos é copiado para o novo processo
- Informações como pid, ppid são únicos
- Mas o processo filho continua a execução do mesmo ponto que o processo pai
- Além disso, o retorno da função é diferente para os processos e pode ser usado para diferenciá-los

Criação de Processos

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

int main()
{
    pid_t id_filho;

    printf("Processo inicial possui pid: %d\n", getpid());
    id_filho = fork();
    if (id_filho == 0)
        printf("Ola, sou o processo filho de pid: %d!\n", getpid());
    else
        printf("Ola, sou o processo pai e criei o filho: %d!\n", id_filho);
    return 0;
}
```

```
Processo inicial possui pid: 16370
Ola, sou o processo pai e criei o filho: 16371!
Ola, sou o processo filho de pid: 16371!
```

Carga de Arquivo Executável

- Desta forma, os processos são criados por clonagem e executarão o mesmo programa
- Como seria possível executar programas diferentes?
- Alguns sistemas permitem criar novo processo através de uma nova imagem de programa
- No UNIX, deve-se realizar uma nova chamada: exec

Carga de Arquivo Executável

- Existe um conjunto de funções exec, todas com o objetivo de alterar o programa em execução do processo atual
- A função retorna apenas em caso de falha: em caso de sucesso a imagem em execução é alterada e o programa é executado a partir do início

Carga de Arquivo Executável

- As famílias de função exec:
- P: (execvp, execlp ...) procuram executar o novo programa de diferentes diretórios do PATH, sem a letra é necessário utilizar full path do binário
- V: (execv, ...) são utilizadas para criar programas com argumentos
- E: também recebem uma lista de varíaveis de ambiente

Carga de Arquivo Executável

- Normalmente, a função exec é usada em conjunto com o fork, para não se alterar um programa que já está em execução a muito tempo

Carga de Arquivo Executável

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

int main()
{
    char *arg[] = { "ls", "-la", NULL };
    pid_t id_filho = fork();
    if (id_filho != 0)
        printf("Ola, sou o processo pai e criei o filho: %d!\n", id_filho);
    else
    {
        execvp("ls", arg);
        printf("Erro!\n");
        return 1;
    }
    return 0;
}
```

Sinais

- Sinais são um mecanismo para comunicar e manipular processos no UNIX
- A linguagem C padrão provê a biblioteca signal.h
- Quando um sinal é recebido pelo processo, a função atual é interrompida e o sinal é tratado por uma função especial chamada de “handler”
- No entanto, o sinal também pode ser ignorado

Sinais

- Um dos usos mais comuns de sinais é para terminar o processo em execução
- Para isso, são usados os sinais SIGTERM e SIGKILL
- Também são definidos sinais de usuário, para que o programador possa definir um uso para eles: SIGUSR1 e SIGUSR2

Sinais

- Os sinais também são usados para o sistema operacional informar erros, SIGSEGV (falha de segmentação), SIGFPE (exeção de ponto flutuante), etc
- A função signal é usada para alterar o comportamento
- Também podemos usar o sigaction, veja o manual signal 2, especialmente a parte “Portability”

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

int contador = 0;

void handler(int signal_number)
{
   contador++;
}

int main ()
{
    struct sigaction sa = {};
    sa.sa_handler = &handler;
    sigaction (SIGUSR1, &sa, NULL);
    printf("Pressione E para terminar a execucao:\n");
    while (getchar() != 'E')
        ;
    printf ("SIGUSR1 recebido %d vezes\n", contador);
    return 0;
}
```

Envio de Sinais

- Se testar esse código, verá que ele irá imprimir que nenhum sinal foi recebido (provavelmente)
- Mas como enviar um sinal para o processo?
- Pode ser usado o comando ou a função kill

Envio de Sinais

- Quando se executa o código anterior:
  - `./programa_de_sinais`
- Usamos o programa ps para determinar o pid do processo (por exemplo, 2240)
- E enviamos o sinal com o comando kill:
  - `[user@station ~]$ kill -SIGUSR1 2240`

Envio de Sinais

- Desta forma, o código anterior irá imprimir quantas vezes, o sinal foi recebido:
-
```
[user@station ~]$ ./programa_de_sinais
Pressione E para terminar a execucao:
E
SIGUSR1 recebido 1 vezes
```

Envio de Sinais

- A função “kill” também pode ser utilizada para enviar um sinal para o processo em um código fonte C:
- `int kill(pid_t pid, int sig);`
- Por exemplo:
- `kill(pid_filho, SIGUSR1);`

Lista de Sinais

Signal	(x86/ARM)	SIGUSR1	10
SIGHUP	1	SIGSEGV	11
SIGINT	2	SIGUSR2	12
SIGQUIT	3	SIGPIPE	13
SIGILL	4	SIGALRM	14
SIGTRAP	5	SIGTERM	15
SIGABRT	6	SIGSTKFLT	16
SIGIOT	6	SIGCHLD	17
SIGBUS	7	SIGCONT	18
SIGFPE	8	SIGSTOP	19
SIGKILL	9	SIGTSTP	20
		(...)	

Término de Processos

- Por padrão, o comando kill envia o signal SIGTERM, cujo padrão é terminar a execução de um processo
- Dois sinais podem ser usados para terminar:
  - SIGTERM
  - SIGKILL
- O sinal SIGTERM pode ser ignorado, mas o SIGKILL não pode

Término de Processos

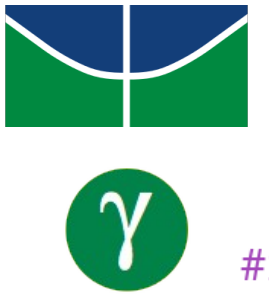
- Além disso, um processo pode terminar por vontade própria de duas formas:
  - Retornando da função main
  - Executando a função exit
- Normalmente:
  - `exit(0)` é utilizado para indicar sucesso
  - `exit(1)` é usado para indicar um erro

Término de Processos

- A função wait pode receber um argumento do tipo *int, para indicar o status da saída
- As macros WIFEXITED e WEXITSTATUS podem ser usadas para ler o inteiro e testar se o processo terminou e o valor de retorno

Aguardando o Término

- Quando o processo pai evoca o processo filho, ele pode aguardar o fim do processo utilizando a função `wait()`
- Se o filho não terminou, o processo pai irá bloquear aguardando o término do processo
- Para aguardar a chegada de um sinal, também podemos utilizar a função `pause()`



Aguardando o Término

```c
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
    pid_t id_filho;
    int status;

    printf("Processo inicial possui pid: %d\n", getpid());
    id_filho = fork();
    if (id_filho == 0)
    {
        printf("Ola, sou o processo filho de pid: %d!\n", getpid());
        exit(0);
    }

    wait(NULL);
    printf("Ola, sou o processo pai e o filho terminou!\n");
    return 0;
}
```

Processos Zumbis

- Quando um processo cria outro processo, assume-se que ele irá receber as informações de término do processo pela função wait
- O que acontece se o processo não estiver em wait?
- O processo filho fica em um estado zumbi, indicando que deseja terminar, mas ainda não foi aguardado pelo processo pai

Processos Zumbis

- Só quando a função wait for chamada pelo pai, o resultado da execução é passado para o pai, o processo filho é deletado e a função wait retorna imediatamente

Processos Zumbis

```c
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

int main ()
{
    pid_t filho;

    filho = fork();
    if (filho > 0)
        sleep(6);
    else
        exit(0);
    wait(NULL);
    return 0;
}
```

Processos Zumbis

- Exercício: ler no livro como terminar os processos zumbis com tratamento de sinal e de modo assíncrono

Referências

- Capítulo 3 – MITCHELL, Mark; OLDHAM, Jeffrey; SAMUEL, Alex. Advanced linux programming. Berkeley, CA: New riders, 2001.