Programação Concorrente

Odorico Machado Mendizabal

Universidade Federal de Santa Catarina – UFSC
Departamento de Informática e Estatística – INE

Processos – Parte 3

Objetivo da aula

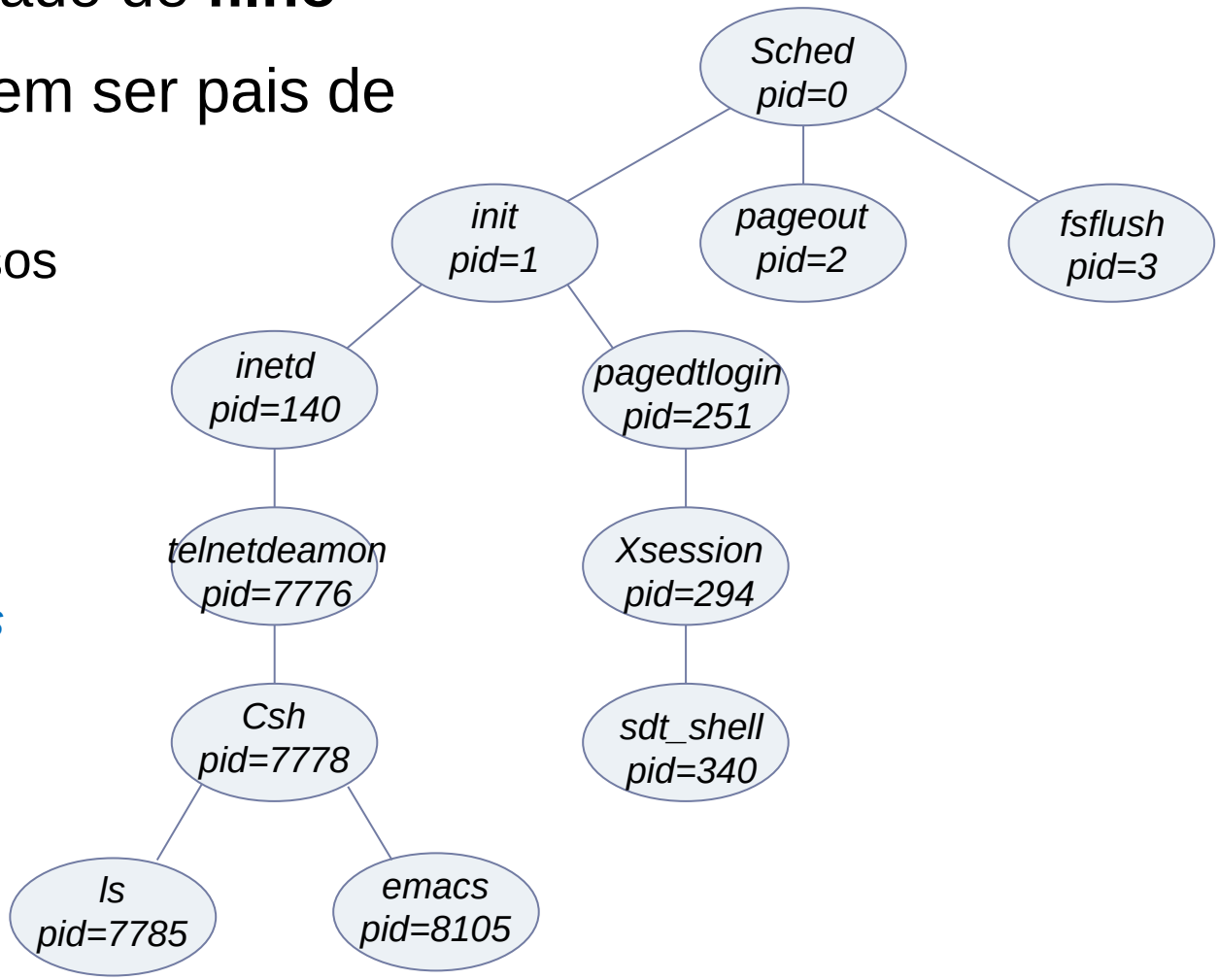
- Identificar os componentes de um processo e ilustrar como eles são representados e gerenciados por um SO
- Descrever o ciclo de vida de um processo e trocas de contexto
- Desenvolvimento de programas usando chamadas ao sistema adequadas
- Descrever e comparar mecanismos de comunicação entre processos

Criação de processos

- Quando o SO inicializa (*boot* do sistema), um processo privilegiado é criado
  - Processo *init* (Linux)
  - *Session Manager Subsystem* – *smss.exe* (Windows)
- Estes processos iniciais iniciam vários outros processos
  - Processos para registro de eventos
    - rsyslogd (Linux)
    - winlogon.exe (Windows)
  - interface gráfica para o usuário e terminal para linha de comando
    - csrss.exe (Windows)
  - Etc..

Criação de processos – Hierarquia de processos

- Processo criador é chamado de **pai**
- Novo processo chamado de **filho**
- Processos filhos podem ser pais de novos processos
  - Hierarquia de processos



Exemplo de árvore de processos em sistema Solaris

Criação de processos

- Quando um processo cria outro processo, ele pode:
  - Executar simultaneamente com seus filhos
  - Espera até que algum ou todos os filhos terminem

- Com relação ao espaço de endereço do novo processo:
  - O filho pode ser uma duplicata do pai (usando **`fork()`**)
  - Ou, ele carrega um novo programa
    - Uso das chamadas ao sistema do tipo **`exec`**:
      - **`execl()`, `execle()`, `execlp()`, `execv()`, `execve()`, `execvp()`**

Criação de processos no Unix

- Processo identificados por um identificador único
  - PID (*Process Identifier*)
- Chamadas ao sistema
  - **`fork()`**
    - Novo processo é uma cópia do espaço de endereçamento do pai
    - Pai e filho seguem a execução com o próximo comando após o `fork()`
    - Valor de retorno:
      - Processo pai recebe o PID do filho
      - Processo filho recebe 0
  - **`exec()`**
    - Carrega um arquivo binário na memória, destruindo a imagem do programa em execução
    - Com `exec()`, o filho pode executar outro programa diferente do pai

Criação de processos no Unix

- Processo identificados por um identificador único
  - PID (*Process Identifier*)
- Chamadas ao sistema
  - **`fork()`**
    - Novo processo é uma cóp
    - Pai e filho seguem a exec `fork()`
    - Valor de retorno:
      - Processo pai recebe o P
      - Processo filho recebe 0
  - **`exec()`**
    - Carrega um arquivo binár programa em execução
    - Com `exec()`, o filho pode executar outro programa diferente do pai

`Fork()` pode falhar, retornando um valor negativo (código de erro):

- O sistema impõe um limite no número de processos em execução

- O sistema impõe um limite no número de processos (`MAXUPRC`) de um mesmo usuário em execução (`<sys/param.h>`)

- Se houver espaço insuficiente para novos processos na área de *swap*

Término de processos

- Chamadas ao sistema:
  - **`exit()`**
    - Retorna um valor *status* de término para o seu pai (normalmente um valor inteiro)
    - Todos os recursos do processo são liberados pelo SO
- Se pai aguarda em uma chamada `wait()`, este receberá o pid do filho que executou `exit()`
  - **`exit()`** e **`wait()`** são primitivas de sincronização

- Processos podem ser terminados involuntariamente
  - O pai pode terminar a execução de um filho
  - Alguns sistemas não permitem que o filho exista se o pai tiver terminado (o SO pode finalizar os processos órfãos)
    - Término em cascata

Criação de processos em sistemas *Unix-like*

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  pid_t pid;
  pid = fork();
  if(pid < 0){
    fprintf(stderr, "Falha na execução do fork");
    return 1;
  }
  else if(pid == 0){
    execlp("/bin/ls","ls",null);
  } else{
    wait(NULL);
    printf("Filho terminou a tarefa");
  }
  return 0;
}
```

Criação de processos em sistemas *Unix-like*

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  pid_t pid;
  pid = fork();        Criação do processo filho
  if(pid < 0){
    fprintf(stderr, "Falha na execução do fork");
    return 1;
  }
  else if(pid == 0){
    execlp("/bin/ls","ls",null);
  } else{
    wait(NULL);
    printf("Filho terminou a tarefa");
  }
  return 0;
}
```

Criação de processos em sistemas *Unix-like*

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  pid_t pid;
  pid = fork();        Criação do processo filho
  if(pid < 0){
    fprintf(stderr, "Falha na execução do fork");
    return 1;
  }
  else if(pid == 0){              É o processo filho
    execlp("/bin/ls","ls",null);
  } else{
    wait(NULL);
    printf("Filho terminou a tarefa");
  }
  return 0;
}
```

Criação de processos em sistemas *Unix-like*

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  pid_t pid;
  pid = fork();
  if(pid < 0){
    fprintf(stderr, "Falha na execução do fork");
    return 1;
  }
  else if(pid == 0){
    execlp("/bin/ls","ls",null);
  } else{
    wait(NULL);
    printf("Filho terminou a tarefa");
  }
  return 0;
}
```

Criação do processo filho

É o processo filho

Executa o comando ls

Criação de processos em sistemas *Unix-like*

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  pid_t pid;
  pid = fork();
  if(pid < 0){
    fprintf(stderr, "Falha na execução do fork");
    return 1;
  }
  else if(pid == 0){
    execlp("/bin/ls","ls",null);
  } else{
    wait(NULL);
    printf("Filho terminou a tarefa");
  }
  return 0;
}
```

Criação do processo filho

É o processo filho

Executa o comando ls

Pai sai da fila de pronto e aguarda o término do filho

Criação de processos em sistemas *Unix-like*

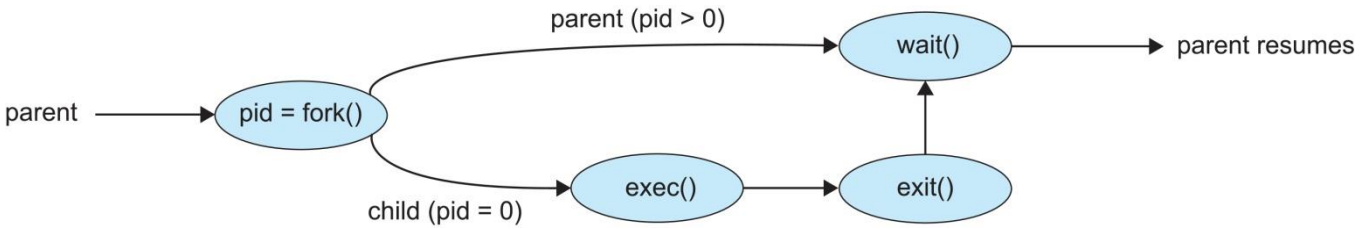
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main(){
  pid_t pid;
  pid = fork();
  if(pid < 0){
    fprintf(stderr, "Falha na execução do fork");
    return 1;
  }
  else if(pid == 0){
    execlp("/bin/ls","ls",null);
  } else{
    wait(NULL);
    printf("Filho terminou a tarefa");
  }
  return 0;
}
```

Criação do processo filho

É o processo filho

Executa o comando ls

Pai sai da fila de pronto e aguarda o término do filho



Outras chamadas à sistema relacionadas a processos

```
pid_t waitpid(pid_t pid, int *status, int options)
```

- Espera por um processo filho particular (pid) terminar a sua execução

```
pid_t getpid()
```

- Retorna o identificador do processo em execução

```
pid_t getppid()
```

- Retorna o identificador do processo pai do processo em execução

...

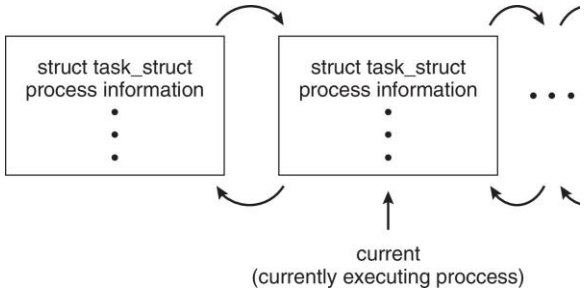
Estudo de caso – Sistema *Unix-like*

Hierarquia de processos

- O primeiro processo a ser criado é o *init* (pid=1)
  - Ele é iniciado no momento do *boot* e termina apenas quando o sistema é desligado
- Cada processo tem apenas um pai, mas um pai pode ter vários filhos
- O relacionamento entre processos é mantido no PCB

Linux `task_struct`

```
pid t_pid;          /* identificador do processo */
long state;
unsigned int time_slice
struct task_struct *parent;/* pai do processo */
struct list_head children; /* filhos do processo */
struct files_struct *files;
struct mm_struct *mm;
```



Estudo de caso – Sistema *Unix-like*

Criação de processos

- Uso da chamada ao sistema **`fork()`**
- O SO executa as seguintes funções
  1) Aloca um espaço na tabela de processos para o novo processo
  2) Atribui um identificador único ao processo filho
  3) Faz uma cópia da imagem do processo pai (exceto regiões de memória compartilhada)
  4) Retorno do **`fork()`**: retorna 0 para o processo filho e o PID do processo filho para o processo pai

Estudo de caso – Sistema *Unix-like*

Criação de processos

- Devido ao alto custo em copiar todo o conjunto de dados do processo pai para o filho, em sistemas Linux, o comando fork implementa uma estratégia de Cópia-na-escrita (CoW – *Copy on Write*)
- CoW: Ao invés de copiar todo o espaço de endereçamento de uma vez, processos pai e filho compartilham a memória e, uma vez que algum deles queira atualizar um dado (uma operação de escrita), então é feita uma duplicata do dado e cada processo mantém a sua cópia. Este processo proporciona que a atualização do espaço de dados do filho seja feita sob-demanda, apenas quando necessária

Estudo de caso – Sistema *Unix-like*

Término de processos

- Se o processo pai termina antes do processo filho?
  - O processo filho se torna órfão
    - Em sistemas *Unix-like*, um processo órfão é imediatamente adotado pelo processo *init*

- Se o processo filho termina antes do processo pai?
  - O processo filho se torna um zumbi (*zombie* ou *defunct process*)
    - Um processo que terminou a execução mais ainda possui uma entrada na tabela de processos
    - Esta entrada ainda é necessária para o pai ler o estado de saída do processo filho

O que é necessário para criar um processo

- Construir um PCB
  - Operação de baixo custo
- Criar tabela de páginas para formar o espaço de endereçamento do processo
  - Operação mais cara
- Copiar dados do processo pai?
  - Chamada fork()
    - Originalmente faz uma cópia completa da memória do pai
      - Operação muito cara
    - Muito menos cara com o uso de "*cópia na escrita*" (*copy on write*)
- Copiar estado de E/S
  - descritores de arquivos, *sockets*, etc.
    - Operação de custo moderado

Referências

Parte destes slides são baseadas em material de aula dos livros:

- *OLIVEIRA, Rômulo Silva de; CARISSIMI, Alexandre da Silva.; TOSCANI, Simão Sirineo. Sistemas operacionais. 4. ed. Porto Alegre: Bookman, 2010. xii, 374p. (Livros didáticos, n.11) ISBN 9788577805211*

- *SILBERSCHATZ, Abraham.; GAGME, Greg; GALVIN, Peter B. Sistemas operacionais com Java. Rio de Janeiro: Elsevier, 2008. 673 p. ISBN 9788535224061*

- *TANENBAUM, Andrew S. Sistemas operacionais modernos. 3. ed. Rio de Janeiro (RJ): Prentice-Hall do Brasil, 2010. xiii, 653p. ISBN 9788576052371*