

# Implementação de Processos

---

Fontes:  
Silberschatz cap 3  
Tanenbaum cap 2

- Criação de processos
- Processos: chamadas de sistema
- Manipulação de processos
- Terminação de processos

# Criação de processos

---

- Sistemas antigos
  - ♦ apenas o SO podia criar novos processos
- Sistemas atuais
  - ♦ usuários podem criar e destruir processos dinamicamente
    - SO fornece chamadas para manipulação e gerência de processos

# Quando criar um processo? (1)

---

- Em sistemas de uso geral, há, basicamente, 4 meios pelos quais os processos são criados:
  1. inicialização do sistema
  2. execução de uma chamada de sistema p/ criação de processo, realizada por um processo em execução
  3. requisição do usuário p/ criar um novo processo
  4. execução de uma tarefa (job) em lote

# Quando criar um processo? (2)

---

## 1. inicialização do sistema

- Quando o SO é carregado, criam-se vários processos

Processos em primeiro plano (*foreground*)

- interação com os usuários

Processos em segundo plano (*background*)

- não estão associados a um usuário específico
- possuem função definida
- Ex.: programa em background designado a aceitar e-mails (daemons)

# Quando criar um processo? (3)

---

2. execução de uma chamada de sistema p/ criação de processo, realizada por um processo em execução
- Útil quando a tarefa a ser executada baseia-se em diversos processos, que interagem de forma independente
    - Ex.: grande quantidade de dados trazida via rede para ser processada – um processo executa o download e outro processo remove os dados e os processa
    - Sistema multiprocessador: cada processo pode executar em um processador diferente – aumenta desempenho

# Quando criar um processo? (4)

---

## 3. requisição do usuário p/ criar um novo processo

- Em sistemas interativos, usuários iniciam um programa digitando um comando ou clicando em um ícone

Ação inicia um novo processo e executa nele o programa selecionado

# Quando criar um processo? (5)

---

## 4. execução de uma tarefa (job) em lote

- Em sistemas em lote, usuários podem submeter jobs  
Quando SO tiver recursos, irá criar um processo e executar nele o próximo job da fila de entrada

### Nos 4 casos:

- todo novo processo é criado por um processo existente, via syscall
  - função do SO: construir estruturas de dados e alocar espaço de endereçamento
-

# Criação de processo

---

- Tanto no Unix/Linux, como no Windows, o novo processo possui um espaço de endereçamento diferente do processo pai (criador)
  - Não há compartilhamento de memória
  - Esta decisão de projeto mantém a consistência do sistema como um todo
- Um processo filho pode compartilhar com o pai recursos do tipo arquivos abertos,...



# Etapas da criação de um processo

---

- atribuição de identificador único ao processo  
*pid* único
  - alocação de estruturas de dados associadas ao processo  
adição de nova linha à tabela de processos
  - alocação de espaço para a imagem em memória  
código + dados + pilha + bloco de controle
  - inicialização do PCB  
PCB recebe as informações básicas  
PCB é colocado na fila de prontos
  - atualização das listas do SO p/ manter consistência
-

# Processos no Unix

---

- Processo
  - programa em execução
  - ocupa espaço no gerenciador de processos
- Como identificar os processos ?
  - Process identification – PID
  - SO garante que enquanto o processo estiver em execução seu PID será único
  - Parent Process Identification - PPID

# Identificação de processos

---

**pid\_t getpid(void)**

Retorna o PID do processo

**pid\_t getppid(void)**

Retorna o PID do criador do processo (PAI)

# Exemplo de teste do ID dos processos

```
3  #include <stdio.h>
4  #include <unistd.h>
5  #include <stdlib.h>
6
7  int main() {
8      pid_t idProcesso;
9      idProcesso = getpid();
10
11      printf("\nPID = %d\n", idProcesso);
12      printf("PPID = %d\n\n", getppid());
13      getchar();
14
15      exit(0);
16  }
17
18
```

```
pitthan@pitthan: ~
43273 pts/0    00:00:00 bash
43627 ?        00:00:00 kate
43698 pts/0    00:00:00 pid-ppid
43706 ?        00:00:02 gnome-terminal-
43714 pts/1    00:00:00 bash
43790 ?        00:00:00 oosplash
43825 ?        00:00:55 soffice.bin
44113 ?        00:00:00 tracker-store
44144 pts/1    00:00:00 ps
pitthan@pitthan:~$
```

.inha 1 de 21, Coluna 1 INSERIR pt\_BR tabulações emuladas: UTF-8 C

```
pitthan@pitthan:~$ ./pid-ppid
```

```
PID = 43698
PPID = 43273
```

# Criação de processo – ações

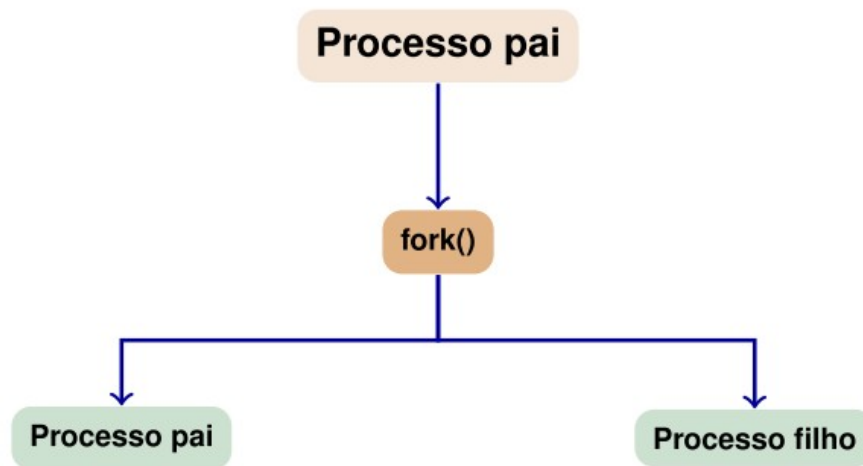
---

- Atribui PID
- Aloca espaço p/ processo
- Inicializa PCB
- Prepara ligações apropriadas
  - Ex.: coloca na lista encadeada que implementa a fila de escalonamento
- Cria/expandi outras estruturas de dados
  - Ex.: mantém um arquivo de contabilidade

# Criação de processo – Unix

---

- Chamada de sistema *fork()*
  - cria um processo filho que herda:
    - cópia idêntica de variáveis e memória do pai
    - cópia idêntica de todos os registradores



# Chamada de sistema *fork()*

---

- *fork()* é invocada uma vez, no processo pai, mas retorna 2 vezes, uma no pai e outra no filho
- processo filho é uma cópia do processo pai
  - áreas do processo pai são duplicadas (código, dados, pilha, memória dinâmica)
- processo filho (assim como o pai) continua a executar as instruções seguintes à chamada *fork()*
- em geral, não se sabe quem continua a executar imediatamente após uma chamada *fork()* (se é o pai ou o filho) - depende do algoritmo de escalonamento

# Sintaxe da chamada de sistema *fork()*

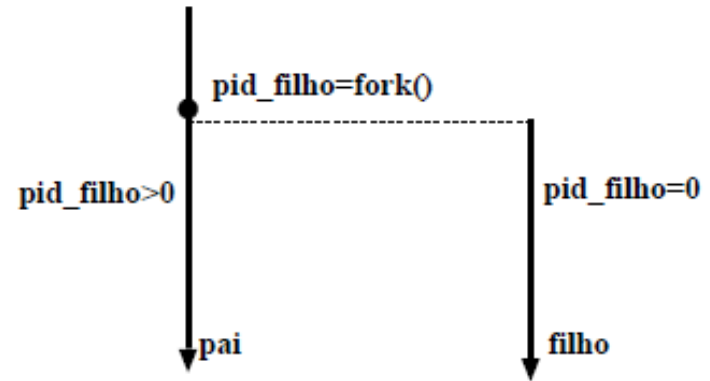
---

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

```
pid_t fork(void);
```

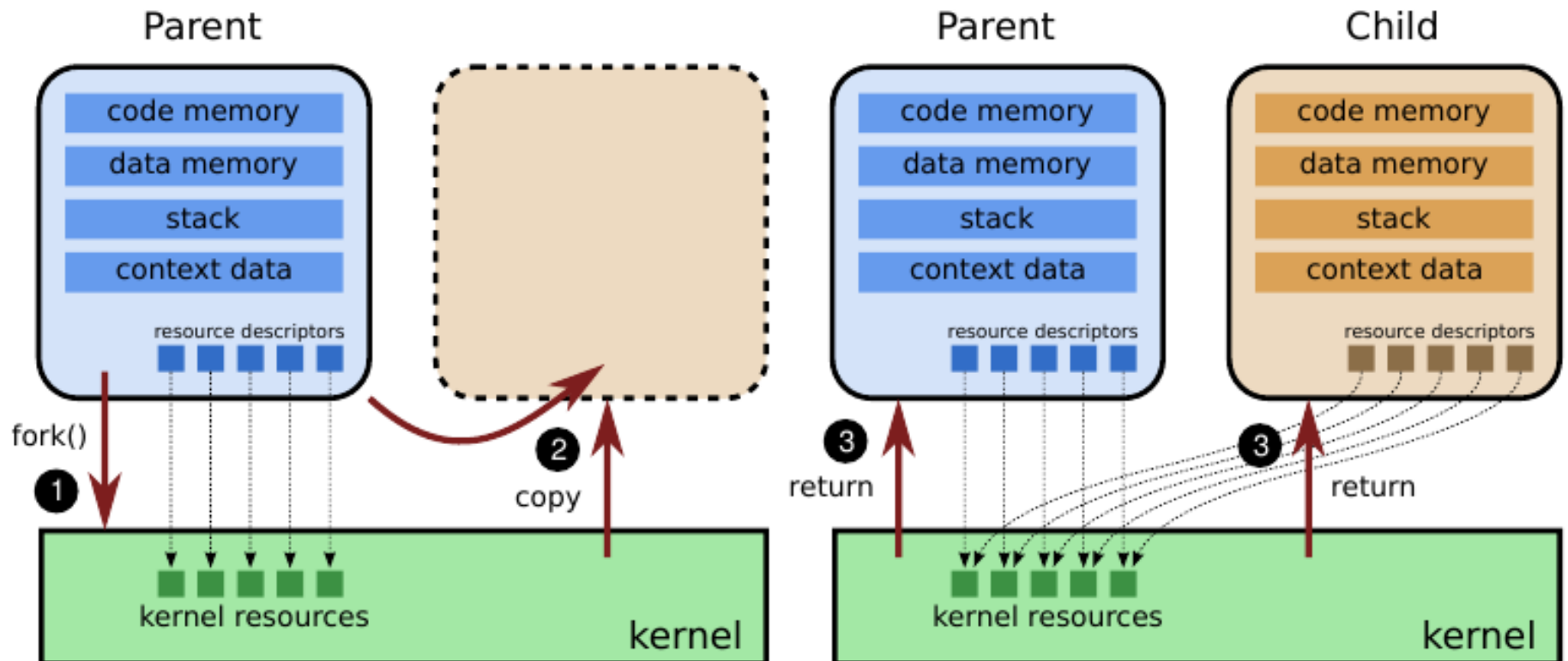
retorna:

- 0, para o processo filho
- PID do filho, para o processo pai
- -1, se houve erro e o serviço não foi executado





# *fork* (1)



# Ex.: chamada fork

Sugestão: pesquisar sobre os comandos ps, pstree e htop

```
5 int main(){
6     pid_t idProcesso;
7
8     idProcesso = fork();
9
10    if (idProcesso < 0){ // erro no fork
11        fprintf(stderr, "fork falhou\n");
12        exit(-1);
13    }
14    else if (idProcesso == 0) // filho
15        printf("sou o FILHO, id = %d, meu pai eh %d\n\n", getpid(), getppid());
16    else // pai
17        printf("sou o PAI, id = %d, meu pai eh %d\n\n", getpid(), getppid());
18
19    getchar();
20    exit(0);
21 }
```

Linha 24 de 26, Coluna 1

```
pitthan@pitthan:~$ gcc -o fork fork.c
pitthan@pitthan:~$ ./fork
sou o PAI, id = 15007, meu pai eh 11956
sou o FILHO, id = 15008, meu pai eh 15007
```

```
pitthan@pitthan: ~
pitthan@pitthan:~$ ps -la
F S  UID      PID     PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S   1000    1107    1105  3  80   0  -  217335 ep_pol tty2      00:16:13 Xorg
0 S   1000    1191    1105  0  80   0  -  47127 poll_s tty2      00:00:00 gnome-sess
0 S   1000   15007   11956  0  80   0  -    622 wait_w pts/0      00:00:00 fork
1 S   1000   15008   15007  0  80   0  -    622 n_tty_ pts/0      00:00:00 fork
4 R   1000   15013  14717  0  80   0  -    2877 -        pts/1      00:00:00 ps
pitthan@pitthan:~$
```

# Processo órfão

---

- quando um processo pai morre antes de seu filho...
  - ♦ processo filho é “adotado” pelo processo inicial do sistema (`init` ou `systemd`)
  - ♦ kernel garante que todos os filhos do processo terminado viram órfãos e são adotados pelo processo inicial
    - ♦ ppid passa a ser pid do `init/systemd`

# Ex.: processo órfão

```
5 ▼ int main(){
6   >> int pid = fork(); // duplica, pai e filho continuam daqui
7   >>
8   ▼ >> if (pid != 0) {>>> // processo pai
9   >>     printf("Sou processo com PID = %d, meu pai eh %d, criei filho com PID = %d\n", getpid(), getppid(), pid);
10  >>   }
11  ▼ >> else {>> >> // processo filho
12  >>     sleep(10);>> >> // garante que o pai termina antes
13  >>     printf("Sou filho com PID = %d, meu pai eh %d\n", getpid(), getppid());
14  >>   }
15  >>   printf("Processo com PID = %d terminou\n", getpid()); // ambos executam
16  >>   exit(0);
17  }
```

Linha 19 de 22, Coluna 1

INSERIR pt\_BR ▾ Tabulações emuladas: 4 ▾

```
pitthan@pitthan:~$ gcc -o orfao orfao.c
pitthan@pitthan:~$ ./orfao
Sou processo com PID = 17808, meu pai eh 11956, criei filho com PID = 17809
Processo com PID = 17808 terminou
pitthan@pitthan:~$ Sou filho com PID = 17809, meu pai eh 1077
Processo com PID = 17809 terminou

pitthan@pitthan:~$ ps -q 1077
  PID TTY          TIME CMD
 1077 ?           00:00:02 systemd
pitthan@pitthan:~$
```

# Processo “zombie”

---

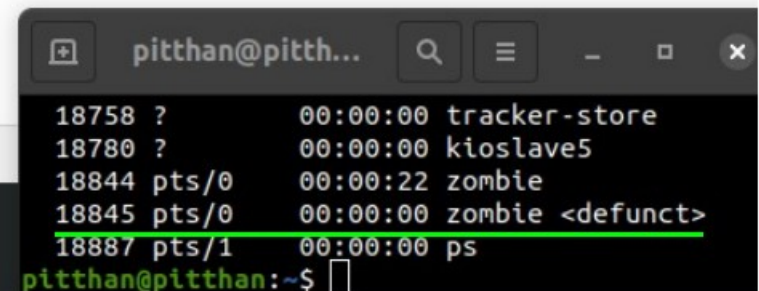
- um processo pode “se terminar” quando seu pai não está a sua espera
  - o processo filho torna-se um processo “zombie”
    - identificação “defunct” ou “zombie” ao lado do nome do processo
  - segmentos de instruções e dados o sistema são automaticamente suprimidos com sua morte
    - processo continua ocupando espaço na tabela de processos do kernel
  - quando seu fim é esperado, ele desaparece da tabela de processos

# Ex.: processo “zombie”

```
5 ▼ int main() {
6     int pid ;
7
8     printf("Sou o processo pai, PID = %d, e vou criar um filho\n",getpid());
9     pid = fork();
10
11 ▼     if(pid == -1) {
12         fprintf(stderr, "fork falhou\n");
13         exit(-1);
14     }
15 ▼     else if(pid == 0) { // filho
16         printf("Sou o filho, PID = %d, vou dormir...\n",getpid());
17         sleep(20);
18         printf("Sou %d e acordei! Vou terminar... oops, virei um 'zombie!!!\n", getpid());
19         exit(0);
20     }
21 ▼     else { // pai
22         printf("Sou %d e vou entrar em loop infinito\n", getpid());
23         for(;;);
24     }
25     exit(0);
26 }
```

Linha 28 de 28, Coluna 1

```
pitthan@pitthan:~$ ./zombie
Sou o processo pai, PID = 18844, e vou criar um filho
Sou 18844 e vou entrar em loop infinito
Sou o filho, PID = 18845, vou dormir...
Sou 18845 e acordei! Vou terminar... oops, virei um 'zombie!!!
```



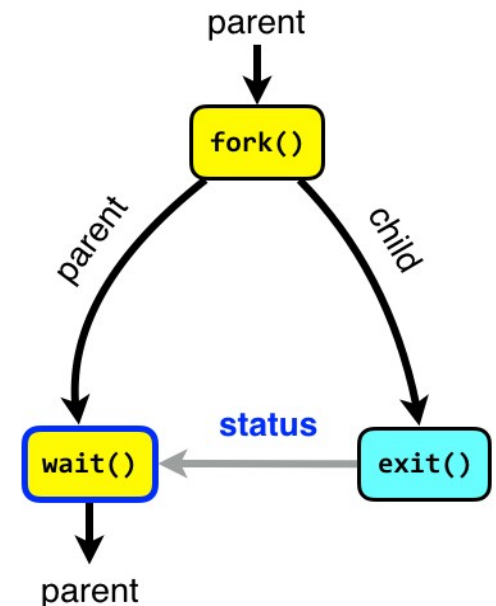
18758	?	00:00:00	tracker-store
18780	?	00:00:00	kioslave5
18844	pts/0	00:00:22	zombie
18845	pts/0	00:00:00	zombie <defunct>
18887	pts/1	00:00:00	ps

```
pitthan@pitthan:~$
```

# Unix – Chamada de sistema *wait()*

---

- Processo pai pode **esperar o término** de um processo filho através da função `wait`
  - retorna o status de retorno de qualquer processo filho que termine
  - um processo que invoque `wait` pode:
    - bloquear – se nenhum dos seus filhos tiver terminado
    - retornar o código de terminação de um filho – caso o filho já tenha terminado
    - retornar um erro – se não tiver filhos





# Ex.: chamada wait

```
6  ▼ int main(){
7    >> pid_t idProcesso;
8    >> int estado, cont = 0;
9
10   >> idProcesso = fork();
11   ▼ >> if (idProcesso < 0) {
12       >> fprintf(stderr, "fork falhou\n");
13       >> exit(-1);
14   }
15   ▼ >> else if (idProcesso != 0){ >> // pai
16       >> printf("Sou processo pai, pid = %d\n", getpid());
17       >> wait(&estado);
18       >> printf("Sou pai (pid = %d), esperei o filho %d\n", getpid(), idProcesso);
19   }
20   ▼ >> else if (idProcesso == 0){ >> // filho
21   ▼ >> while (cont < 3){
22       >> printf("Sou filho (%d), meu pai eh %d\n", getpid(), getppid());
23       >> sleep(2);
24       >> cont++;
25   }
26   >> exit(1);
27   >> }
28   >> exit(0);
29 }
```

Linha 31 de 34, Coluna 1

```
pitthan@pitthan:~$ ./wait
Sou processo pai, pid = 19592
Sou filho (19593), meu pai eh 19592
Sou filho (19593), meu pai eh 19592
Sou filho (19593), meu pai eh 19592
Sou pai (pid = 19592), esperei o filho 19593
pitthan@pitthan:~$
```



# Execução de processos – Unix

---

- Chamada de sistema **exec\***
  - após criado (com fork), o processo filho executa uma outra chamada de sistema (**exec\***) para **mudar sua imagem de memória** e executar um novo programa
    - **imagem de memória**: conteúdo do espaço de endereçamento
  - após criado (com fork), o processo filho executa uma outra chamada de sistema (**exec\***) para **mudar sua imagem de memória** e executar um novo programa
    - **execl(), execl(), execlp(), execv(), execve() e execvp()**

# Exemplo de exec

```
6 ▼ int main(){
7   >> pid_t idProcesso;
8   >> idProcesso = fork();
9   ▼ >> if(idProcesso == -1){
10  >>   >> perror("Fork falhou");
11  >>   >> return 1;
12  >> }
13 ▼ >> if(idProcesso == 0){ >> >> // filho
14  >>   >> printf("Sou o filho (%d), vou trocar de imagem\n\n", getpid());
15  >>   >> execl("/bin/ls", "ls", NULL, NULL);
16  >> }
17  >> printf("Sou o pai (%d), estou esperando meu filho (%d) terminar\n", getpid(), idProcesso);
18 ▼ >> if (idProcesso != wait(NULL)) {
19  >>   >> perror("Pai falhou ao esperar, devido a sinal ou erro");
20  >>   >> return 1;
21  >> }
22  >> printf("Sou o pai (%d), estou terminando\n", getpid());
23  >> return 0;
24 }
25
```

Linha 4 de 27, Coluna 22      INSERIR    pt\_BR    Tabulações emuladas: 4    UTF-8    C

```
pitthan@pitthan:~/Documentos/Documentos/backup-desk-jan2020/ensino/material/sisop-c
od/progs/processos/elc1016-2021-2$ ./exec
Sou o pai (22396), estou esperando meu filho (22397) terminar
Sou o filho (22397), vou trocar de imagem

exec  exec.c  fork2.c  fork.c  orfao.c  pid-ppid.c  wait.c  zombie.c
Sou o pai (22396), estou terminando
pitthan@pitthan:~/Documentos/Documentos/backup-desk-jan2020/ensino/material/sisop-c
od/progs/processos/elc1016-2021-2$
```

# Terminação de Processos no Unix (1)

---

Término normal (voluntário):

a tarefa a ser executada é finalizada

**exit()**

Término com erro (voluntário):

o processo em execução não pode ser finalizado

Ex.: gcc exemplo.c, onde o arquivo exemplo.c não existe

# Terminação de Processos no Unix (2)

---

## Término com erro fatal (involuntário)




- ♦ Erro causado por um bug no programa
  - Exemplos: divisão por 0, acesso à posição de memória inexistente, acesso a posição de memória não pertencente ao processo, execução de uma instrução ilegal, ...


## Término causado por algum outro processo (involuntário)

`kill()`

# Simulador fork, wait, exec

**Simulador de Fork, Wait e Exec**

Copy-on-write: ☐ English:  Ajuda  Sobre 

**Processo Pai** 

```
main() {
  open(file1);
  int const aux = 1;
  int i = 0;
  if (fork() != 0) {
    printf("Eu sou o pai");
    i++;
    wait(NULL);
  } else {
    execv("./filho", NULL);
  }
  return 0;
}
```

**exemplo**

**Visualização dos processos**

**Lista de Processos:**

Processo
Pai

**Tabela de Página Pai**

Página	Endereço	Modo
text	10	rw
data	11	rw

**RAM**

Página	Endereço
text-pai	10
data-pai	11

**Descritores de Arquivo:**

Processo	Arquivo
----------	---------