Sistemas Operacionais I

Laboratório 07

Gerência de Processos

Agenda

- Processos em C
- Argumentos de Linha de Comando
- Leiaute da Memória de um Programa C
- Identificadores de Processos
- Criação de Processos
- Funções, fork, exit, wait e exec
- Exercícios

Processo em C

- Um processo em C começa executando pela função main()
- Antes do main(), uma função especial de inicialização é chamada
  - Obtém parâmetros do kernel e realiza configurações
- Programa pode terminar normal ou de forma anormal

Processos em C

- Terminação Normal
  - Retorno de main
  - Chamada a exit
  - Chamada a _exit ou _Exit
  - Retorno da última thread da rotina de inicialização
  - Chamada a pthread_exit da última thread

Processo em C

- Terminação Anormal
  - Chamada a abort
  - Recepção de sinal
  - Resposta da última thread a uma requisição de cancelamento
- exit(0) é o mesmo que return(0) na função main e representa uma terminação normal

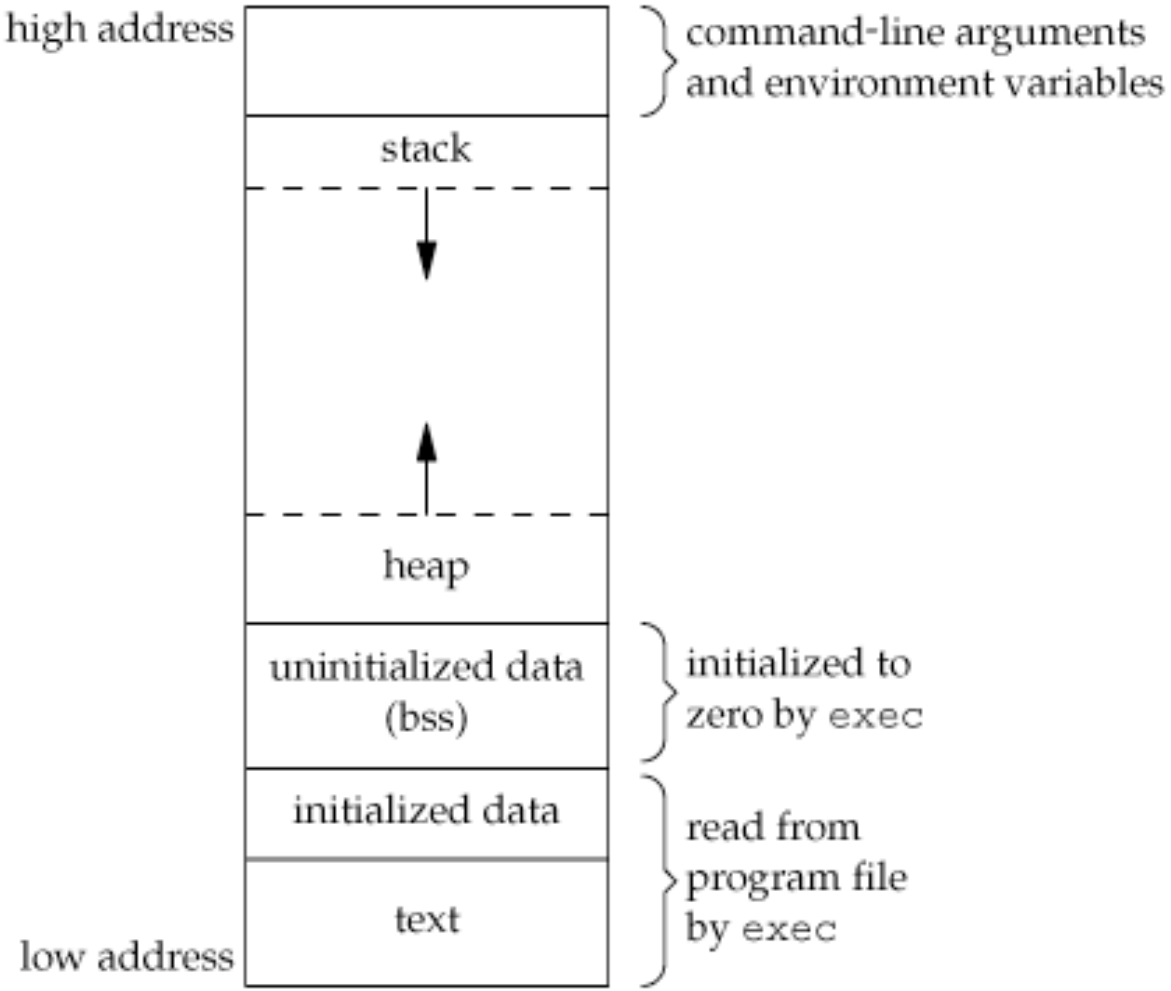
Argumentos de Linha de Comando

- Exemplo que ecoa todos os comandos de linha de argumento:

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
      int i;
      for (i = 0; i < argc; i++)
      printf("argv[%d]: %s\n", i, argv[i]);
      exit(0);
}
```

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

Organização da Memória de um Programa em C



Organização da Memória de um Programa em C

- Segmento de texto: instruções de máquina que o programa executa. Usualmente compartilhado e somente-leitura.
- Segmento de dados inicializados: variáveis globais inicializadas
- Segmento de dados não inicializad / bss (block started by symbol): variáveis globais não inicializadas

Organização da Memória de um Programa em C

- Pilha (stack): armazena variáveis automáticas e informações salvas na chamada de funções.
- Heap: armazenamento de variáveis alocadas dinamicamente.
- Somente são salvos no disco (formato executável) o segmento de texto e os dados de inicialização.

Organização da Memória de um Programa em C

- O comando size no Unix informa o tamanha dos segmentos de texto, dados e bss em bytes
- Exemplo:

```
$ size /usr/bin/cc /bin/sh
text data bss dec hex filename
79606 1536 916 82058 1408a /usr/bin/cc
619234 21120 18260 658614 a0cb6 /bin/sh
```

Identificadores de Processos

- Todo processo tem um único identificador: *Process ID ou PID*
  - número inteiro não negativo único
  - IDs são reutilizáveis: quando o processo termina, seu PID é candidato a reuso
  - Atraso no reuso de PID é imposto por vários Unix

Identificadores de Processos

- PID 0: normalmente é o escalonador (*swapper)*
- PID 1: processo *init* chamado pelo kernel ao final do bootstrap
  - No Linux é o único processo visível após a inicialização
  - Nunca morre, processo normal do usuário

Identificadores de Processos

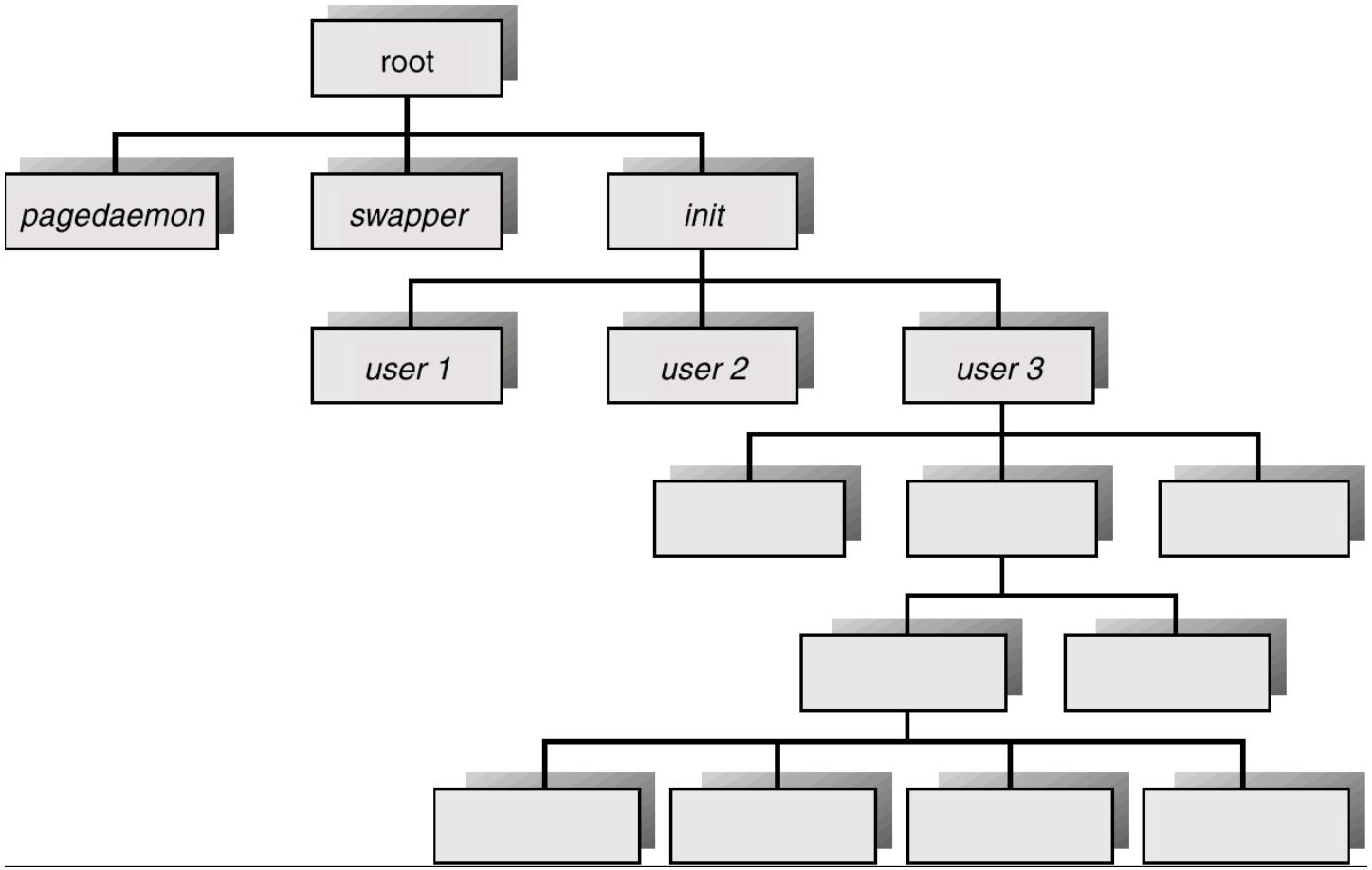
```
#include <unistd.h>
pid_t getpid(void);
/* Retorna o PID do processo chamador */
pid_t getppid(void);
/* Retorna o PID do pai (PPID) do processo chamador*/
uid_t getuid(void);
/* Retorna o real user ID do processo chamador*/
uid_t geteuid(void);
/* Retorna o effective user ID do processo chamador*/
gid_t getgid(void);
/* Retorna o real group ID do processo chamador*/
gid_t getegid(void);
/* Retorna o effective group ID do processo chamador*/
```

Criação de Processos

- Processo → durante sua execução pode criar novos processos por meio de chamadas ao sistema do tipo "create process"
  - Processo que criou → chamado de processo pai
  - Processo criado (novo processo) → chamado de processo filho
  - Cada novo processo pode criar outros processos, formando uma árvore de processos

Criação de Processos

Árvore de processos típica de um sistema UNIX



Criação de Processos

- Quando um processo P_p cria um novo processo P_f, em relação à execução de P_p e P_f:
  - P_p continua a executar concorrentemente com seus processos filhos; ou
  - P_p espera até que alguns ou todos os seus processos filhos tenham terminado

Criação de Processos

- Em relação ao espaços de endereçamento de :
  - P_f é uma duplicação de ; ou
  - P_f tem um programa diferente carregado nele
- Exemplo:
  - **fork()** → system call que cria um novo processo
  - **execve()** → system call que substitui a imagem (espaço de endereçamento de memória) de um processo por outro

Criação de Processos

- Principais chamadas de sistemas:
  - fork(): duplica um processo
  - exit(): termina um processo
  - wait(): espera pelo término de um filho
  - exec(): substitui código, dados e pilhas de um processo

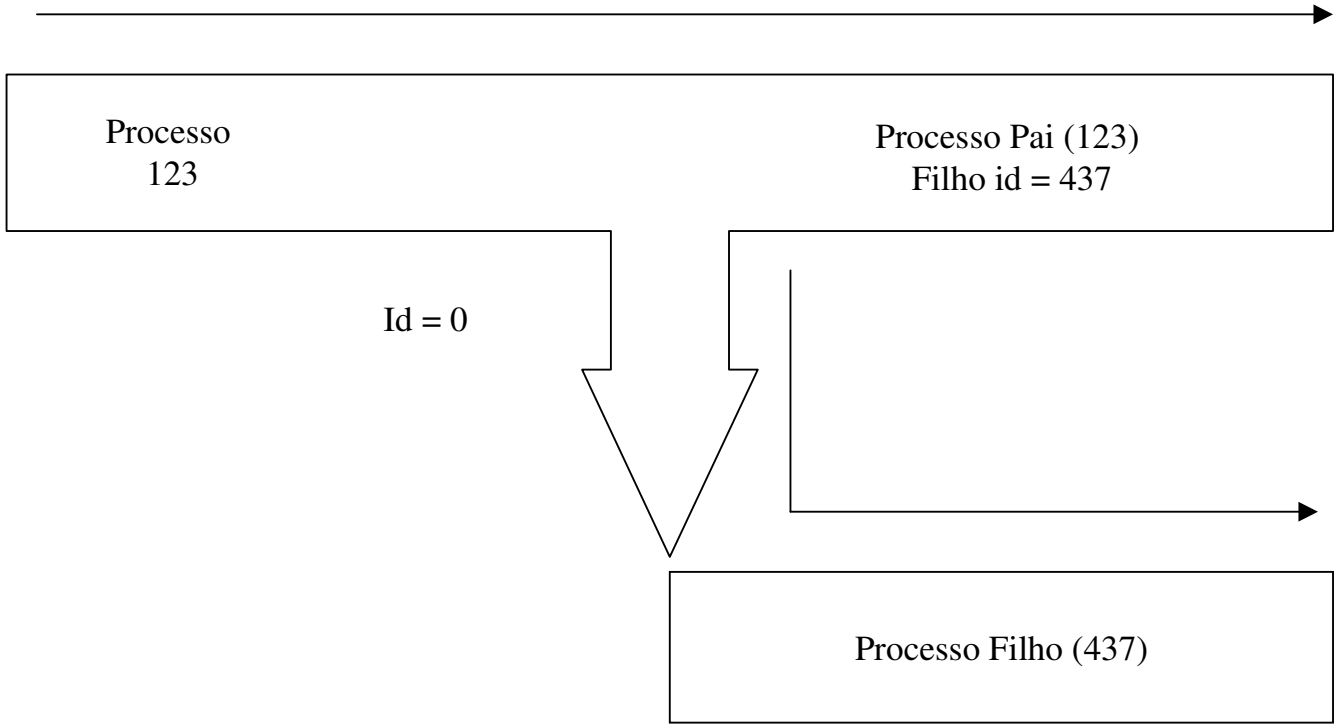
Criação de Processo
Exemplo – fork()

- Chamada de sistema *fork*:
  - Origina uma hierarquia de processos (tree)
  - Duas possibilidades em termos de execução:
    - Processo pai executa concorrentemente ao filho
    - Processo pai espera até o(s) filho(s) terminarem, usa chamada de sistema *wait*
  - Duas possibilidades em termos de espaço de endereçamento:
    - Processo filho é uma duplicação do processo pai
    - Processo filho tem um programa carregado nele utilizando a chamada execve

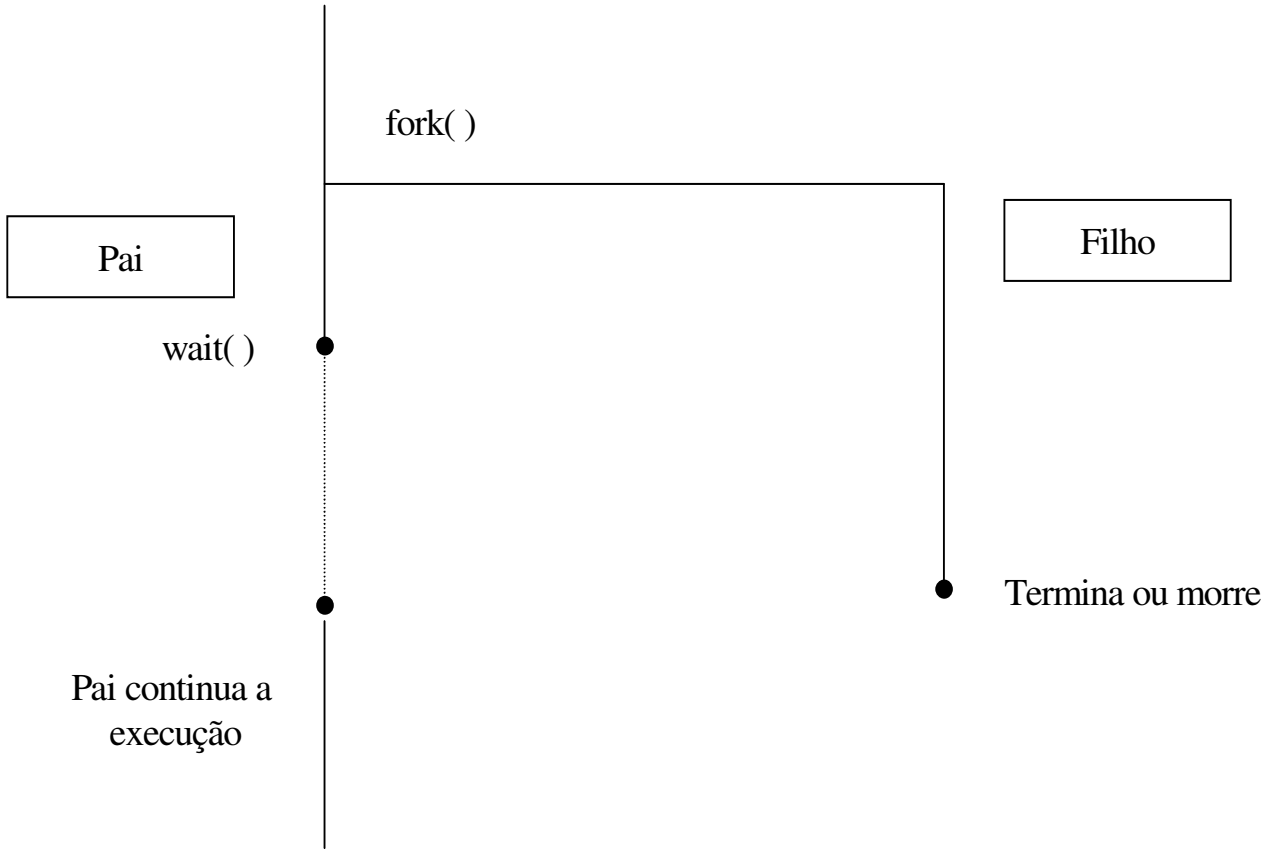
Criação de Processo

```
int pid;

pid = fork();

if (pid == 0) {
        /* código do processo filho */
        exit(0);
}
/*código do processo pai */
wait(...);
```

Criação de Processo



Criação de Processo



Função wait()

- Espera por um processo: pid_t wait (int *status)
  - o processo é suspenso até que um de seus filhos termina
  - a função retorna o pid do filho que termina, pode ficar bloqueado até filho terminar
  - se não tem filhos, retorna imediatamente -1
  - wait prioriza processos zumbis, se existem

Função wait()

```
#include <stdio.h>
main ()
{
    int pid, status, childPid;
    printf ("Eu sou o pai e meu PID e' %d\n", getpid ());
    pid = fork ();
    if (pid != 0)
    {
        printf ("Eu sou o pai com PID %d e PPID %d\n“, getpid (), getppid ());
        childPid = wait (&status); /* Espera filho terminar. */
        printf ("Um filho com PID %d terminou com codigo de saida %d\n“, childPid, status >> 8);
    }
    else
    {
        printf ("Eu sou o filho com PID %d e PPID %d\n“,getpid (), getppid ());
        exit (13);
    }
    printf ("PID %d termina\n", getpid () );
}
```

Função wait()

- Saída:

```
$ ./meuwait
Eu sou o pai e meu PID e' 3830
Eu sou o pai com PID 3830 e PPID 3558
Eu sou o filho com PID 3831 e PPID 3830
Um filho com PID 3831 terminou com codigo de saida 13
PID 3830 termina
```

Funções exec()

- Faz outro programa executar
  - Substitui o processo que chama exec por um novo programa e começa a executar o novo programa
  - PID não muda
  - exec substitui segmentos de texto, dados, heap e pilha por novo programa lido do disco; caso não encontre retorna -1

Funções exec()

- Existem 6 diferentes exec:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,…
/* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0,…
/* (char *)0, char *const envp[] */ );
int execve(const char *pathname, char *const argv[],
char *const envp []);
int execlp(const char *filename, const char *arg0,…
/* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

Funções exec()

```
#include <stdio.h>
#include <unistd.h>
int main()
{
    int pid;
    pid = fork(); /* cria outro processo */
    if (pid < 0) { /* ocorrência de erro*/
        printf("Criação Falhou");
        exit(-1);
    }
    else if (pid == 0) { /* processo filho*/
    execlp("/bin/ls", "ls", NULL);
    }
        else { /* processo pai */
        wait (NULL); /* pai irá esperar o filho completar execução */
        printf ("Filho Completou Execução");
        exit(0);
    }
}
```

Operações com Processos

- Terminação
  - Executa última instrução ou chamada *exit*
  - Pode enviar dados para o pai via wait;
  - Um processo pode causar a terminação de outro (via system call abort). Normalmente somente o pai a utiliza
  - Razões para o processo filho ser terminado:
    - Processo filho excedeu recursos alocados
    - Tarefa solicitada ao processo filho não é mais necessária

Terminação de Processo

- Um processo termina quando executa seu último comando e faz chamada ao sistema do tipo "exit", pedindo para o SO destruí-lo.
- Todos os recursos do processo (memória, arquivos, buffers, …) são desalocados pelo SO
- Processo pode causa a terminação de outro, por meio de uma chamada ao sistema do tipo "terminate process", passando o ID do processo a ser terminado (normalmente somente o processo pai pode terminar os seus processos filhos)

Terminação de Processo

- Alguns SOs não permitem que um processo filho exista, se seu pai já terminou.
- Quando um processo termina, todos os seus filhos também são terminados → chamado de terminação em cascata

Exercícios

1. Crie um programa que cria um processo filho. O filho apresenta na tela a mensagem “Sou filho!”, seu PID e o PPID. Quando ele terminar de executar, o pai apresenta na tela a mensagem “Sou pai!”, o seu PID e PPID.
2. Faça um programa que crie cinco processos filhos. Cada um dorme um tempo diferente e imprime seu PID na tela. O pai aguarda os filhos terminar e imprime “FIM!”.

Outras Chamadas...

- int setuid (uid_t id): configura o ID do usuário do processo chamador
- int seteuid (uid_t id): configura o ID efetivo do usuário do processo chamador
- int setgid (gid_t id): configura o ID do grupo do processo chamador
- int setegid (gid_t id): configura o ID efetivo do grupo do processo chamador