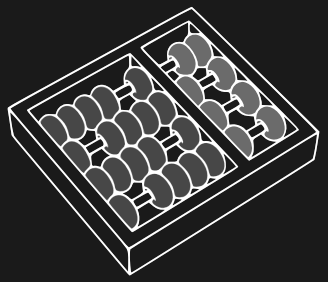


UNICAMP

Processos

Prof. Carlos A. Astudillo

castudillo@ic.unicamp.br

Sistemas Operacionais (MC504A)

2º Semestre 2023

Adaptado de Dave Eckhard

Objetivos da aula

- Processo como uma pseudo máquina
- Ciclo de vida de um processo
- Estados de um processo de kernel

Processos

Analogia entre do Processo e o Cozinheiro

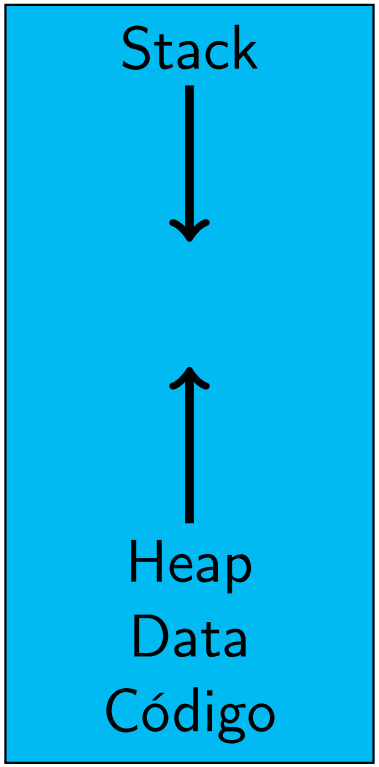
Imagine um cientista da computação com dotes culinários fazendo um bolo:

- Receita = programa
- Cozinheiro = Processador (CPU)
- Ingredientes = dados de entrada

Processo a atividade desempenhada pelo cozinheiro em ler a receita, buscar os ingredientes e assar o bolo.

A computadora

O processo



Registradores

stdin

stdout

timer

Ciclo de vida de um processo

- Ciclo de vida
  - Nascimento
    - Detalhes
  - Trabalho
  - Morte

Nascimento

- De onde vêm os processos?
  - Não da cegonha
- Evento que fazem com que um processo seja criado:
  - Inicialização do sistema.
  - Execução de uma chamada de sistema de criação de processo por um processo em execução.
  - Solicitação de um usuário para criar um novo processo.
  - Início de uma tarefa em lote.

Criação de processos

- O que precisamos?
  - Conteúdo da memória
    - Texto, dados, stack
  - Conteúdos dos registradores do CPU (n deles)
  - Portas de entrada e saída (I/O)
    - Descritores de arquivos, e.g., `stdin`, `stdout`, `stderr`
  - O oculto
    - estado do timer, diretório atual, `umask`

Intimidante?

- Como especificamos todas essas coisas?
- Escolhamos um processo que gostamos
  - Clonemo-lo!

`fork()`

- `fork` é a chamada do sistema original de Unix para a criação de processos
- Memória
  - A copiamos toda
  - Para depois: truques de VM podem fazer a copia mais barata
- Registradores
  - Os copiamos todos
    - Exceto, o pai aprende o ID do processo do filho, e o filho obtém `0`

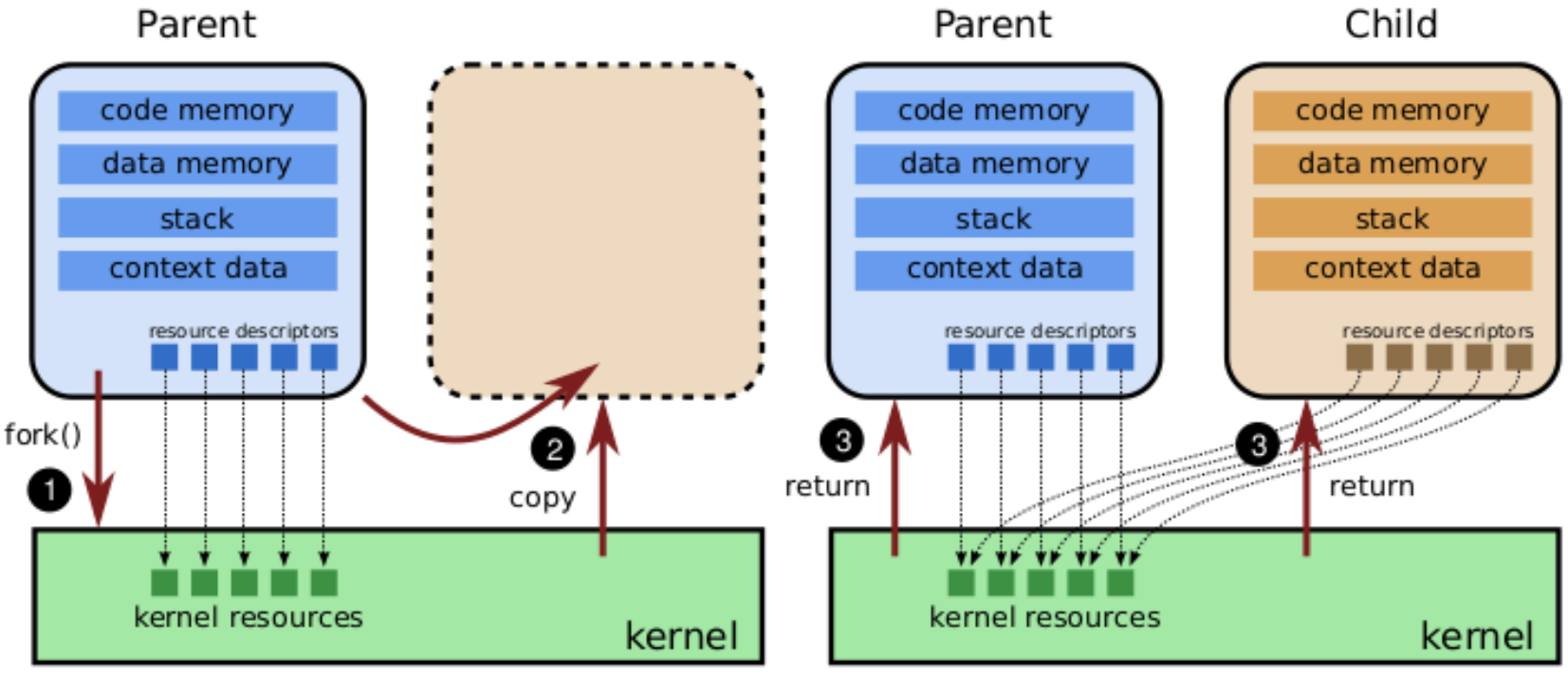
`fork()`

- Descritores de arquivos
  - Os copiamos todos
  - Não copiamos o arquivo
  - Copiamos as referências ao estado dos arquivos abertos
- O oculto
  - Se faz o que é obvio
- Resultado
  - Original, é o processo pai
  - O filho é um processo totalmente especificado, apesar de ter `0` nos parâmetros do `fork`

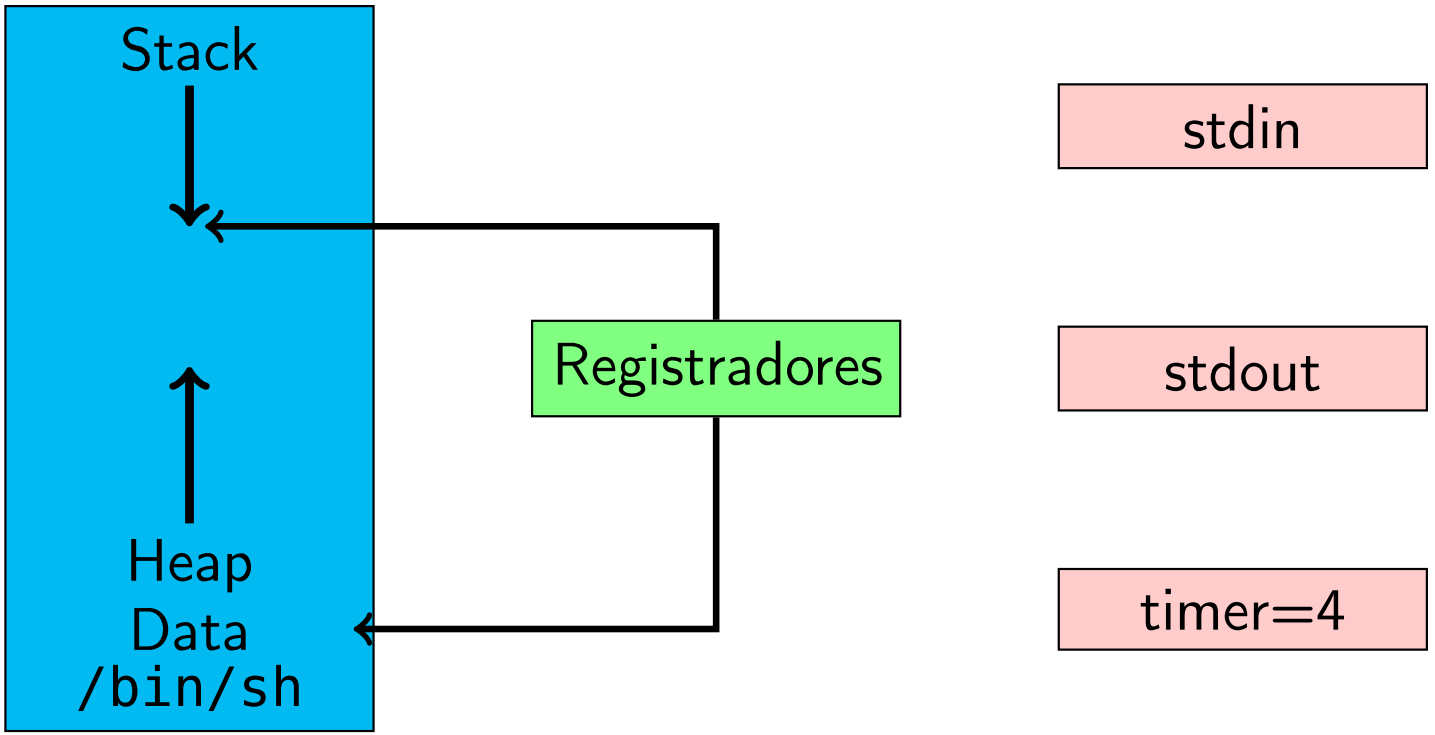
E agora?

- Qual é o ponto de ter dois processos idênticos?
- Fazemos cirurgia plástica ao processo
  - Implantamos nova memória: novo texto de programa
  - Implantamos novos registradores: os antigos não apontam corretamente à nova memória
  - Mantemos a maioria dos descritores de arquivos
    - Bom para cooperar e delegar
  - O oculto
    - Fazemos o que é obvio

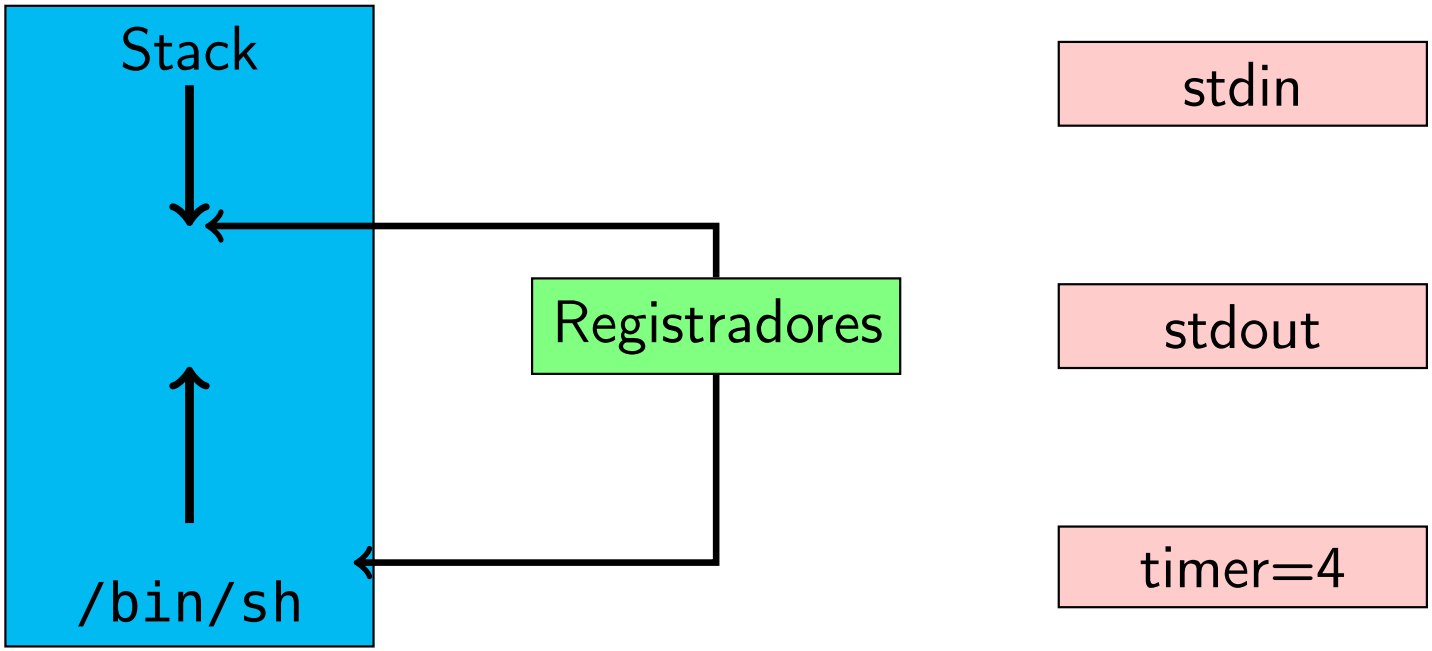
Execução da chamada de sistema fork()



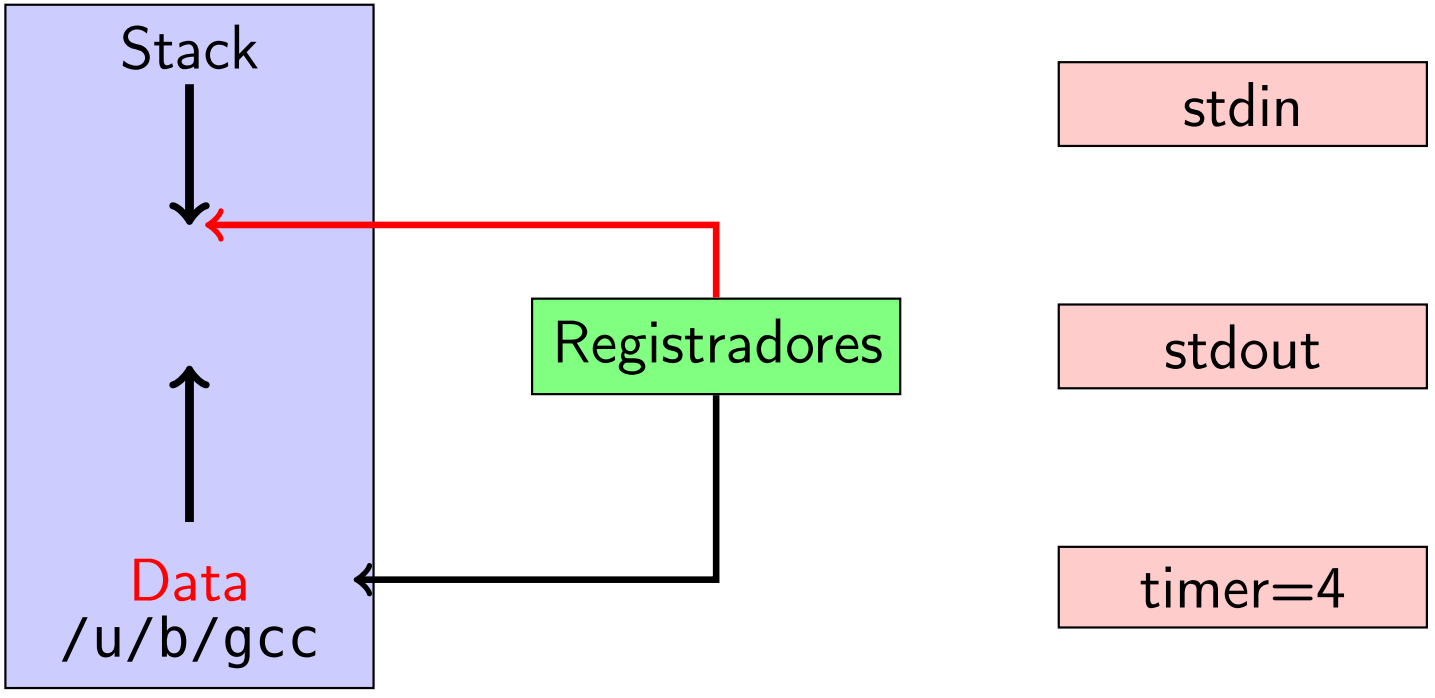
O processo original



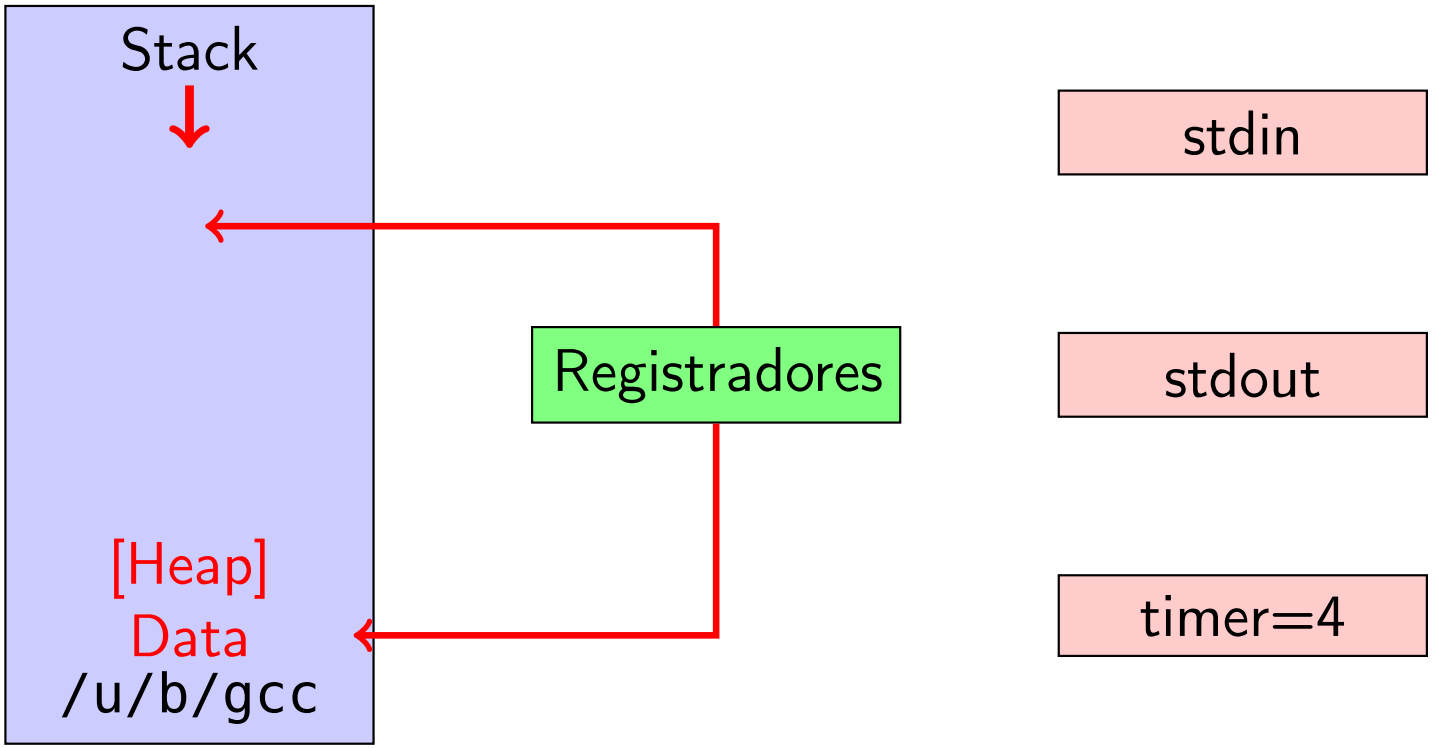
Eliminamos heap e os dados



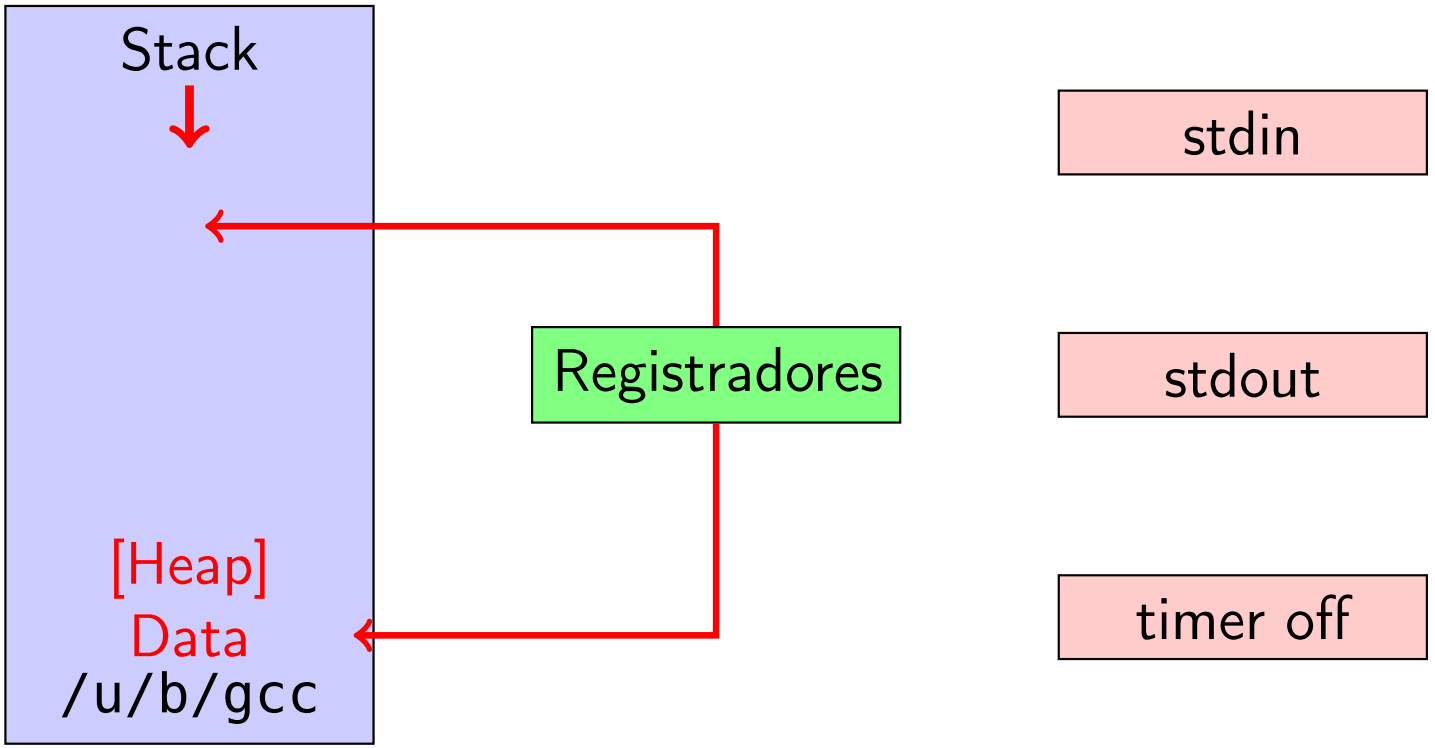
Carregamos o novo código e dados do arquivo



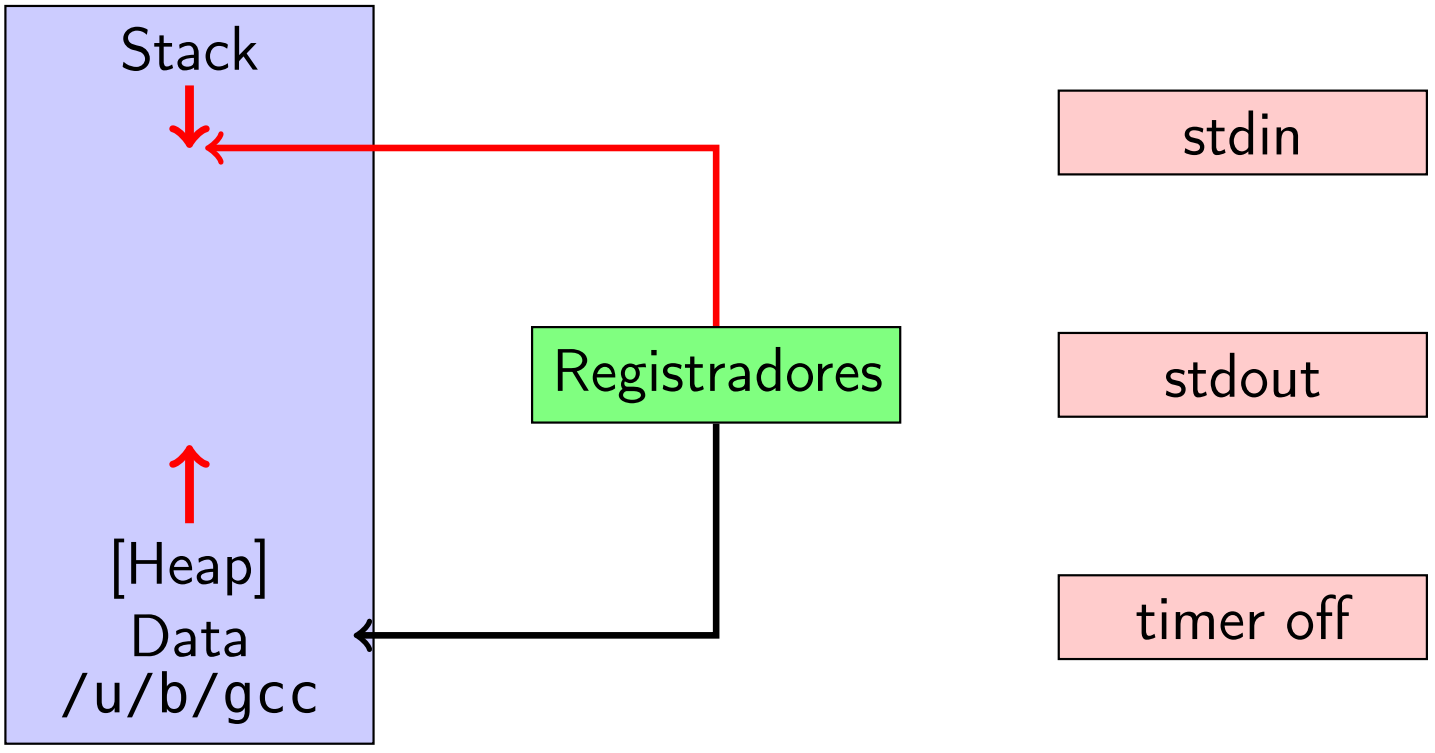
Reset ao stack e à heap



Corrigimos as coisas



Inicializamos os registradores



Formas de executar este procedimento

- execve (*execute program*)

```
int execve(
  char *path,
  char *argv[],
  char *envp[])
```

- spawn()
  - Precisamos especificar todas as características do novo processo (complicado)
  - O bom, não temos que copiar coisas que vamos a eliminar pronto
- fork(), rfork(), clone()[1]
  - Constrói o novo processo através do velho
  - Especifica que coisas se copiam e quais se compartilham

[1] Revisar: http://stackoverflow.com/q/4856255/424986

fork()

Exemplo

```c
int child_pid = fork();
if(child_pid == 0){ // se executa no processo filho apenas
  printf("Sou o processo filho %d\n",getpid());
  return 0;
} else { // se executa no processo pai apenas
  printf("Sou o processo pai de %d\n",child_pid);
  return 0;
}
```

Saídas possíveis

```
Sou o processo pai de 456
Sou o processo filho 456
```

```
Sou o processo filho 456
Sou o processo pai de 456
```

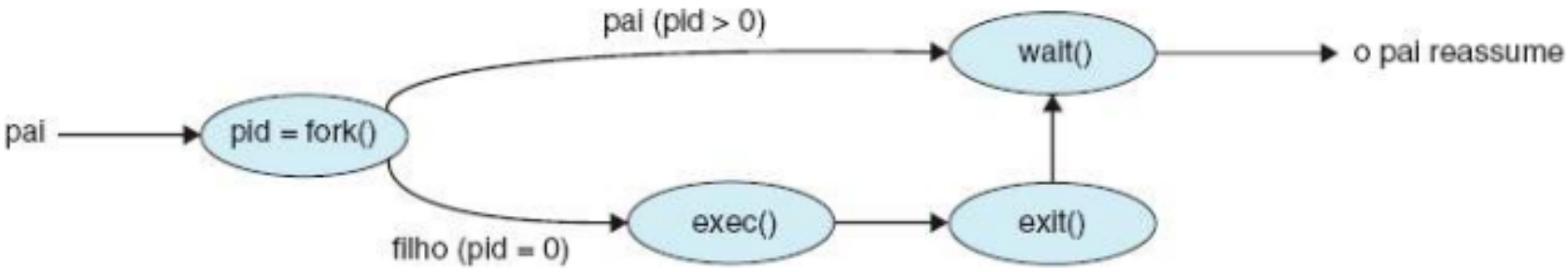
fork()

Outro exemplo

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
pid_t pid;
/* cria um processo-filho */
pid = fork();
if (pid < 0) { /* um erro ocorreu */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* processo-filho */
execlp("/bin/ls","ls",NULL);
}
else { /* processo-pai */
/* o pai esperará que o filho seja concluído */
wait(NULL);
printf("Child Complete");}
return 0;}
```

Estados de um processo

Criação de processo com o uso da chamada de sistema `fork()`



A magia

- O setup do kernel
  - Excluir os dados antigos em memória
  - Excluir o stack antigo em memória
  - Carregar o arquivo executável
- O kernel constrói o stack para o novo processo
  - Transfere `argv[]` e `envp[]` ao topo do stack do novo processo
  - Constrói o frame do stack para `__main()` (`main` do `main`)
  - Estabelece os registradores
    - Ponteiro ao stack (ao topo deste)
    - Contador do programa (ao inicio de `__main()`)

Estados de um processo

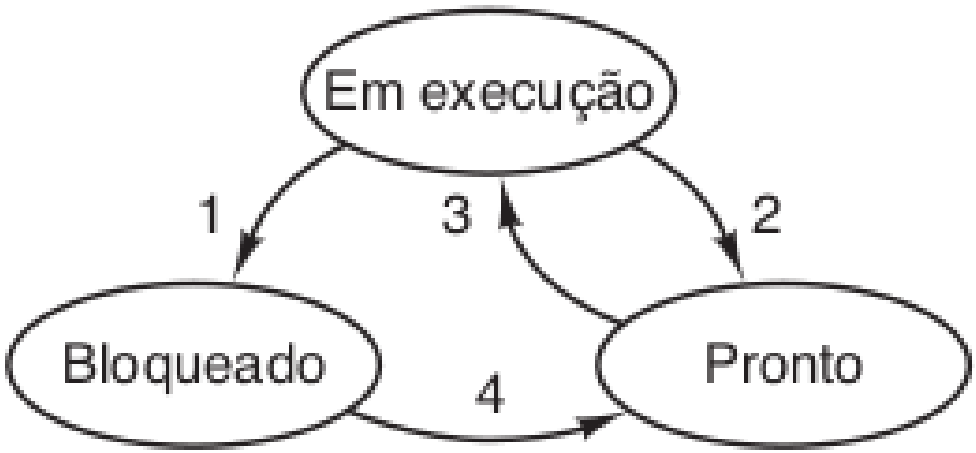
- Em execução
  - Em modo de usuário ou de kernel
- Bloqueado
  - Esperando algum evento
    - Finalização de I/O, saída de outro processo, mensagens, etc.
    - Dormindo (suspendido) por um período de tempo definido
  - Scheduler: *não te executes*

  Q: Modo de usuário? de kernel? ambos? nenhum?
- Pronto/Executável

  Q: Modo de usuário? de kernel? ambos? nenhum?

Estados de um processo

Transição entre estados



1. O processo é bloqueado aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

Outros estados

- Forking
  - Obsoleto, usado alguma vez para um tratamento especial
- Zumbi
  - O processo chamou a `exit()`, o pai não há percebido

Término de processos

- Voluntaria
  - Finalização normal da execução, logoff de usuários
  - Kernel

    ```
    void exit(int reason);
    ```

  - Chamando do sistema do exit no processo

    ```
    /* sair com status 1 */
    exit(1);
    ```

  - Atráves do um return no *main()*
- Excepção de hardware
  - `SIGSEGV`: não há memória aqui para ti (obrigado por participar)
  - Segmentation fault
- Excepção de software
  - `SIGXCPU`: tem usado muito CPU

Chamada do sistema `kill(pid, sig);`

- Entregar `sig` ao processo `pid`
  - Os valores negativos de `pid` têm comportamentos interessantes
- Equivalente ao teclado `^C`
  - `kill(getpid(), SIGINT);`
- Iniciar o parar logging
  - `kill(daemon_pid, SIGUSR1);`
  - `$ kill -USR1 33`
  - `$ kill -USR2 33`
  - É um uso de `kill` que não mata

Limpeza do processo

Todos os recursos do processo — incluindo memória física e virtual, arquivos abertos e buffers de I/O — são desalocados pelo sistema operacional.

- Liberação de recursos
  - Arquivos abertos: `close()` cada um
    - TCP: 2 minutos o mais
    - Disco offline: para sempre (ninguém pode passar)
  - Liberar memória
- Auditoria
  - Levar registrador do uso dos recursos num arquivo mágico
- Terminou?

Zumbis

- O estado do processo converteu ao código de saída
- Espera enquanto o pai chama a `wait()`
  - O código de saída é copiado à memória do pai
  - PCB (process control block) é borrado do kernel

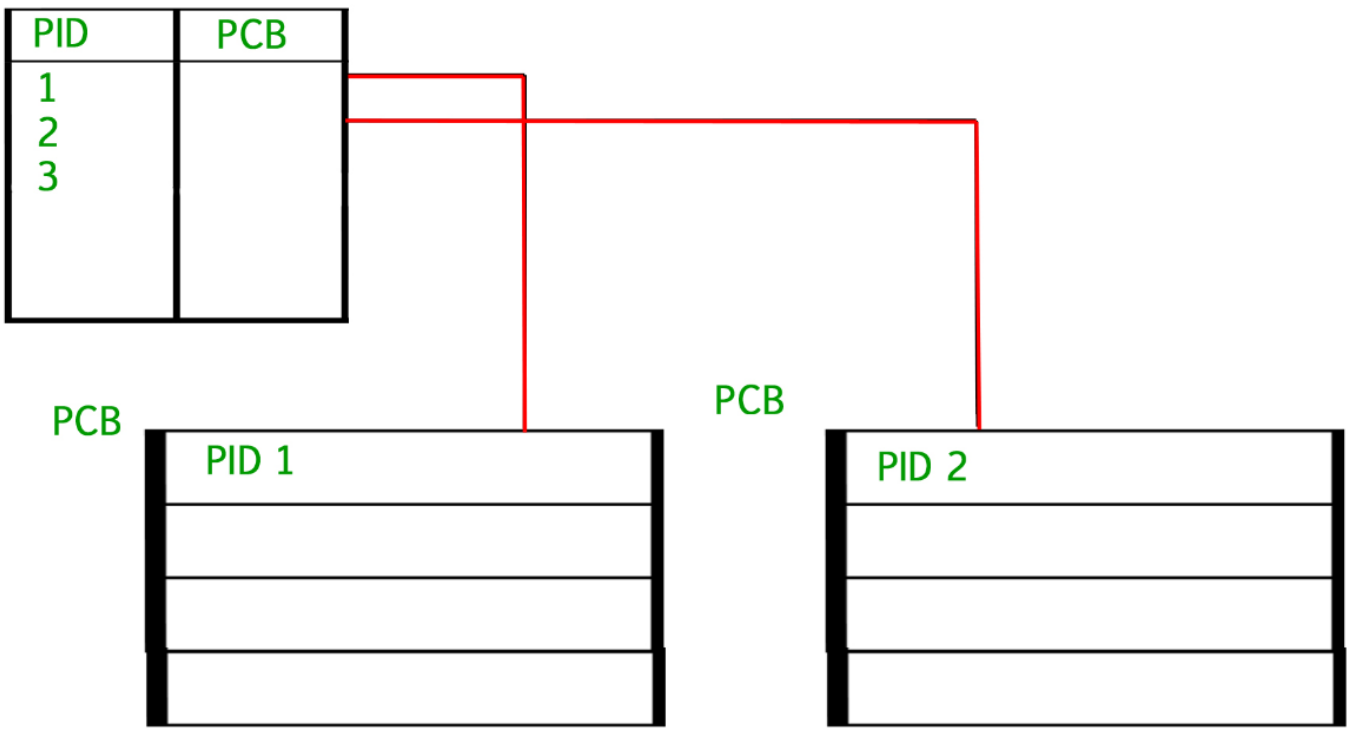
Estado do processo do kernel

Bloco de Controle de Processos (PCB, *process control block*)

- O temido PCB[2]
  - Printer circuit board —não
  - Polychlorinated biphenyl —também não
- Process control block
  - Qualquer coisa sem um endereço de memória visível para o usuário
  - Informação para a administração do kernel
  - Estado do scheduler
  - Mais coisas

[2] http://lmgtfy.com/?q=pcb

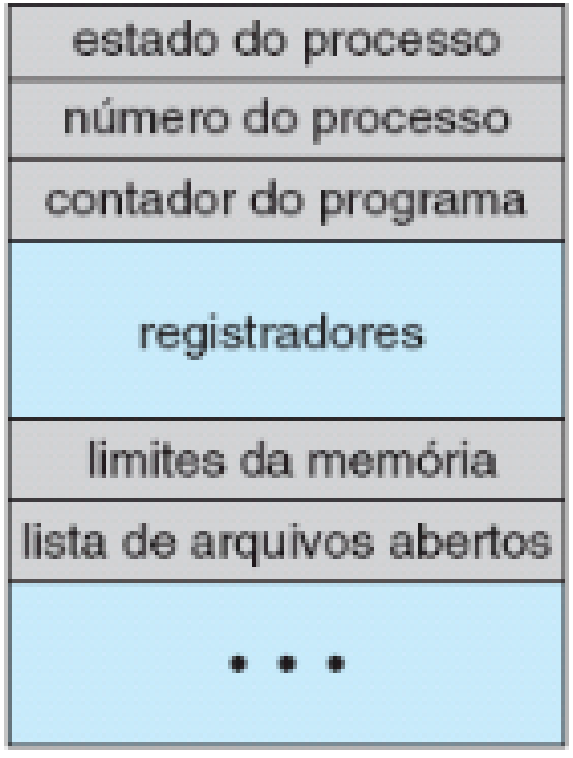
Tabela de Processos e PCB



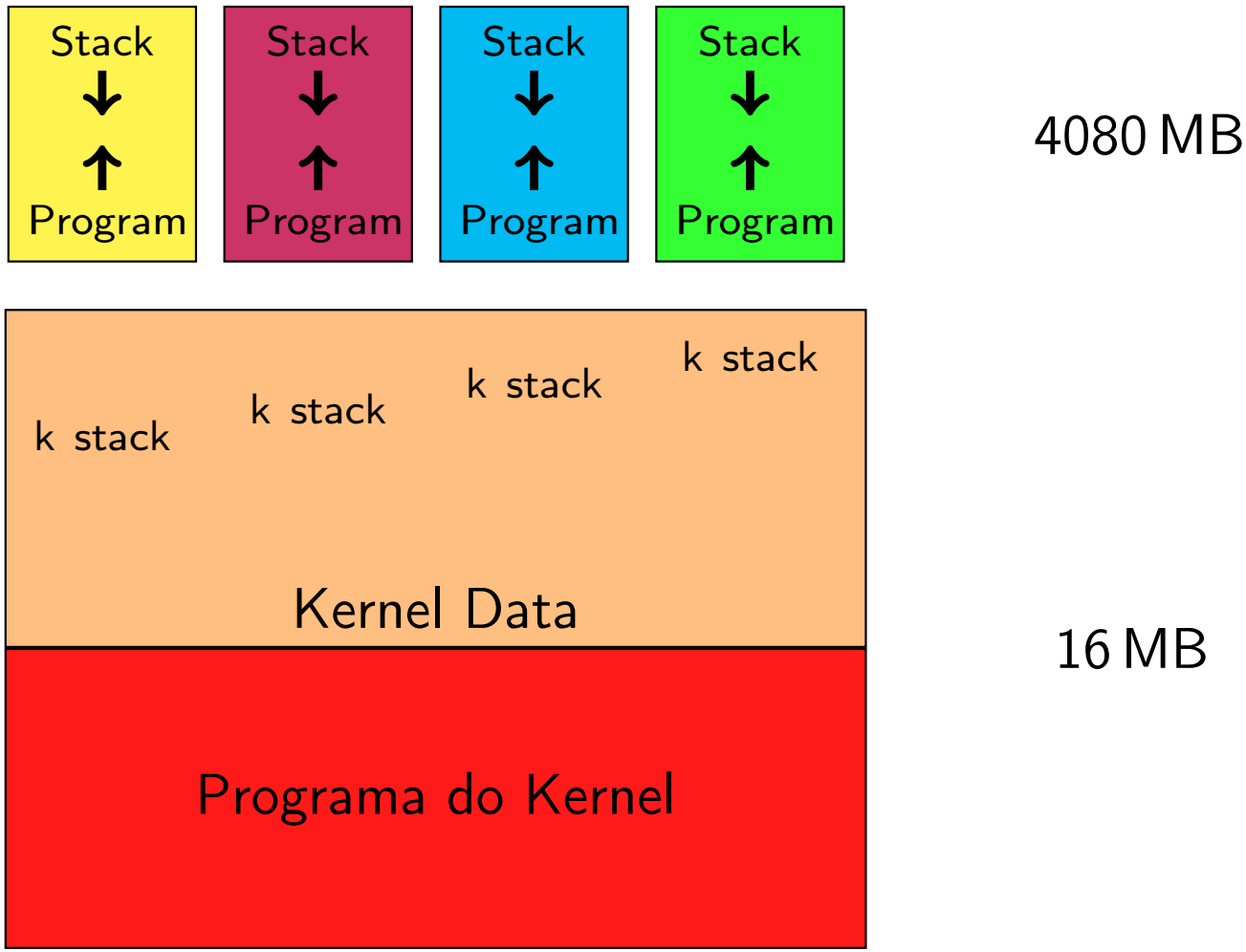
Process table and process control block

Exemplo do conteúdo de PCB

- Estado do processo
- Número do processo, e número do processo pai
- Ponteiro à área de armazenamento de registradores do CPU
- Valor do timer de conta regressiva
- Informação do segmento de memória
  - Lista de segmentos da memória de usuário
  - Referencia do stack do kernel
- Informação do scheduler
  - Posição na lista enlaçada, prioridade, canal onde dorme

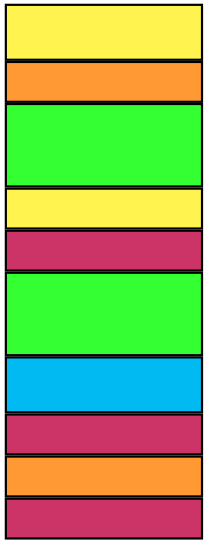


Plano de memória virtual



Plano de memória física

Memória de usuário



240 MB

Memória de kernel

16 MB

Implementação

- `getpid()`
- `fork()`
- `exec()`
- `wait()`
- `exit()`
- A ler os manuais!!

Implementação

wait()

- Aguarda pela morte do filho
- Bloqueia o processo
- Retorna PID do filho morto
- Status indica causa da morte

Exemplo de implementação de um shell UNIX

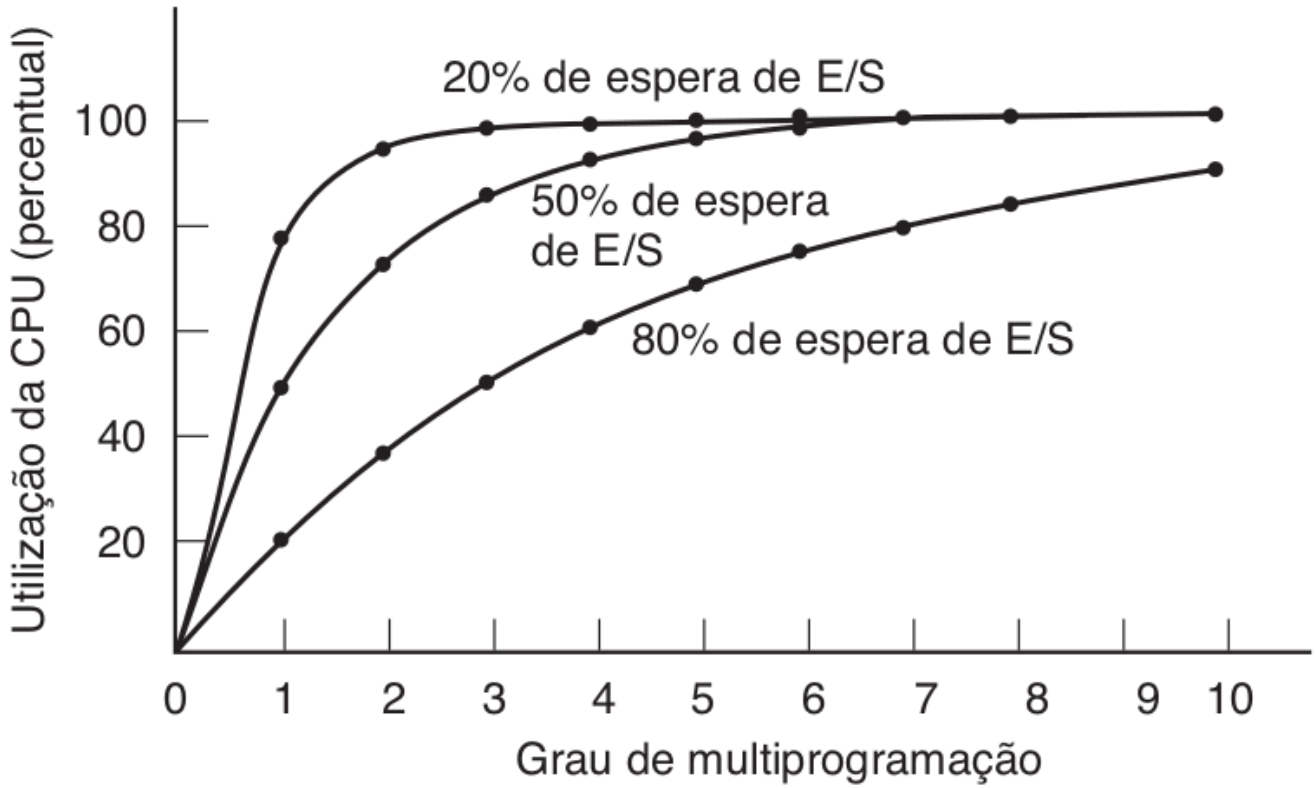
```
main() {
    char *prog = NULL;
    char **args = NULL;

    // Read the input a line at a time, and parse each line into the program
    // name and its arguments. End loop if we've reached the end of the input
    while (readAndParseCmdLine(&prog, &args)) {

        // Create a child process to run the command.
        int child_pid = fork();

        if (child_pid == 0) {
            // I'm the child process.
            // Run program with the parent's input and output.
            exec(prog, args);
            // NOT REACHED
        } else {
            // I'm the parent; wait for the child to complete.
            wait(child_pid);
            return 0;
        }
    }
}
```

Multiprogramação



Utilização da CPU $= 1 - P^n$, onde P é a fração de tempo que um processo se encontra esperando que os dispositivos de E/S conclua uma requisição e n o grau de multiprogramação.

Cooperação entre Processos

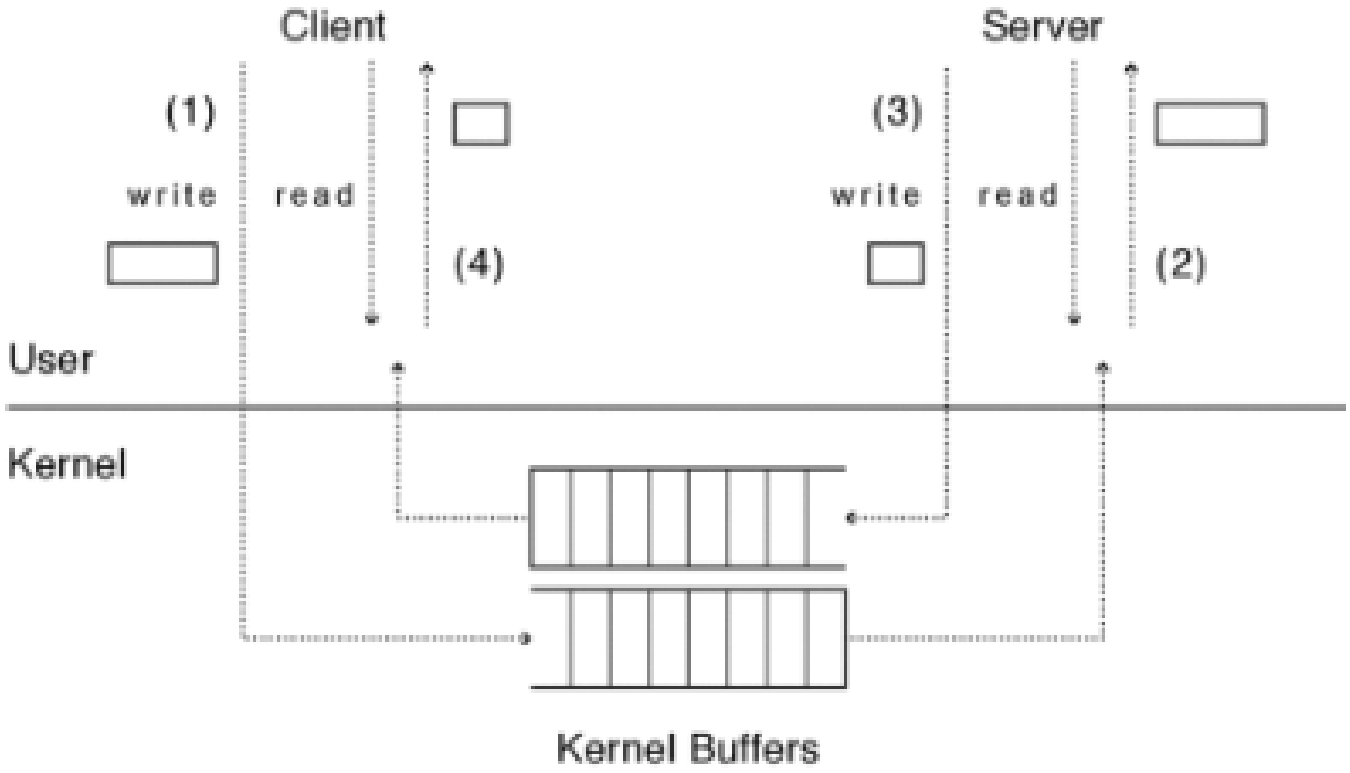
- Compartilhamento de informações: accesso concorrente a arquivos, por exemplo.
- Aumento da velocidade de processamento: subtarefas executadas em paralelo
- Modularidade: funcões do sistema em processos separados
- Conveniência: mesmo um mesmo usuário pode estar editando, imprimindo e compilando simulataneamente.

Comunicação entre Processos

- Produtor-Consumidor
- Cliente-Servidor
- Sistema de Arquivos

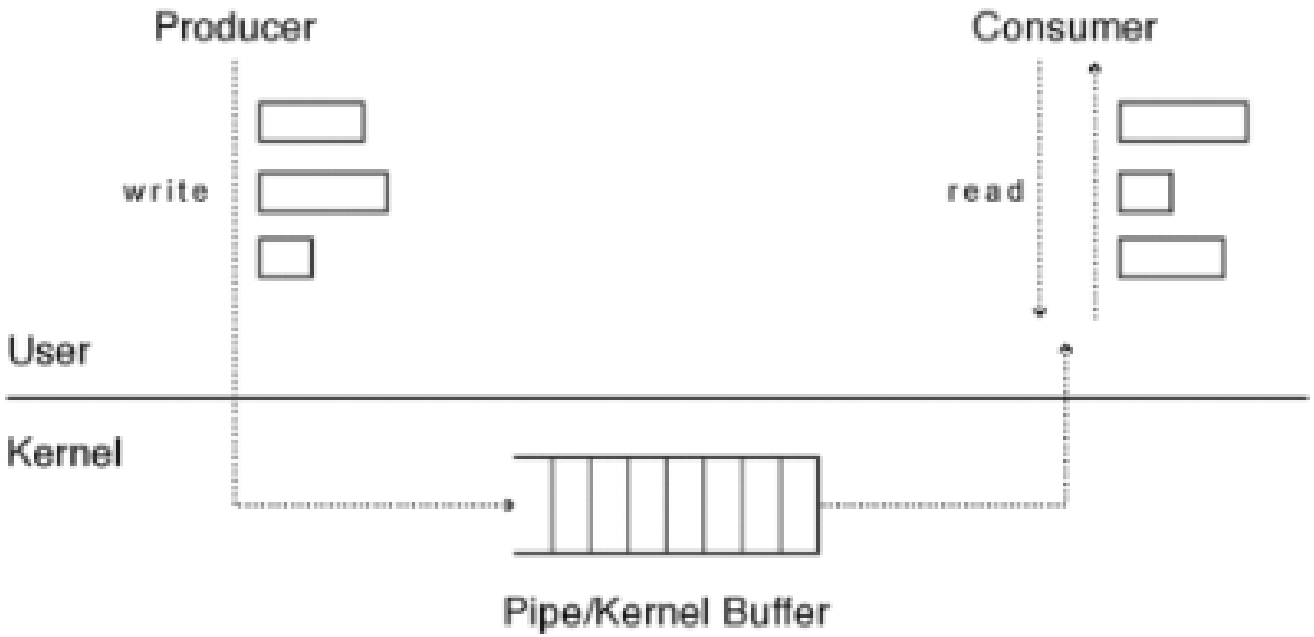
Comunicação entre Processos

Produtor-Consumidor



Comunicação entre Processos

Cliente-Servidor



Pontos chave

- Partes de um processo
  - Físico: páginas de memória, registradores, dispositivos I/O
  - Virtual: regiões de memória, registradores, portos I/O
- Nascimento, trabalho, e morte de um processo
- Escola sobre processos
- *Big picture* do processos
  - Dois vistas: virtual e física
- Comunicação entre processos