Processos

MCTA026-13 - Sistemas Operacionais

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q1

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de **Sistemas Operacionais na UFABC**.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Este material foi baseado nas ilustrações e texto preparados por Silberschatz, Galvin e Gagne [SGG] e Tanenbaum e Bos [TB] detentores do copyright.
- Algumas figuras foram obtidas em: `http://pngimg.com`

Introdução

Roteiro

UFABC

- Conceito de processo
- Escalonamento de processos
- Operações com processos
- Comunicação inter-processo (IPC)
- Exemplos de sistemas de IPC
- Comunicação em sistemas Cliente/Servidor

Objetivos

UFABC

- Apresentar a noção de um processo - um programa em execução - que forma a base para todo o mecanismo de computação
- Descrever as várias características de um processo, incluindo escalonamento; criação e finalização; e comunicação
- Explorar o uso de comunicação inter-processo utilizando memória compartilhada e passagem de mensagens
- Descrever a comunicação de sistemas cliente-servidor.

Processos

Conceito de processo

UFABC

- Um processo (*process*) é um programa em execução.
- Um programa é uma entidade passiva, código compilado que tipicamente é armazenado no disco em um arquivo executável
- Um processo é uma entidade ativa

Pergunta

Quando um programa se torna um processo?

Conceito de processo

UFABC

- A execução de um programa pode ser iniciada via
  - Clicks do mouse
  - Comando executado em uma CLI
  - Iniciado a partir de outro processo[1]
  - etc...
- Um programa, quando em execução, pode ser representado por múltiplos processos
  - Múltiplos usuários executando o mesmo programa
  - Um mesmo usuário usando várias instâncias de um mesmo programa
  - Por segurança. Por exemplo, Firefox, Chrome, ...

[1] A princípio todos os processos (exceto o SO propriamente dito) são iniciados assim. Por que?

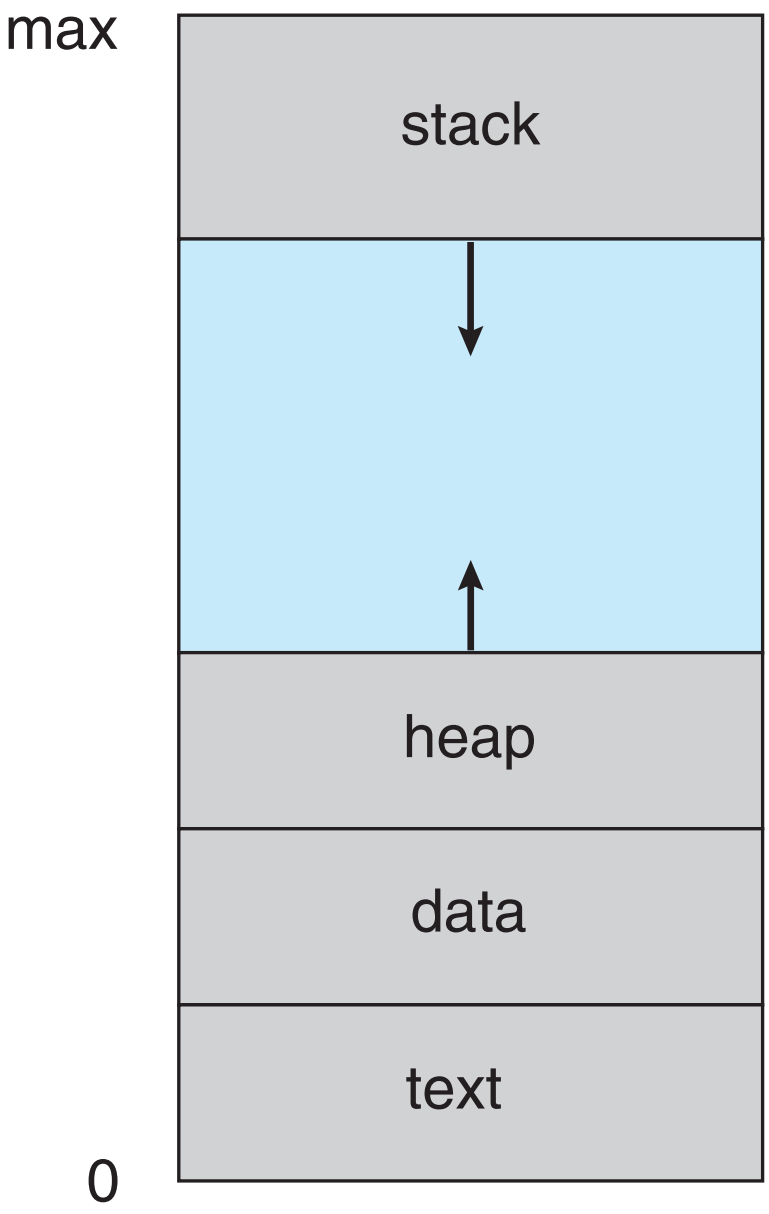
Estrutura de um processo

UFABC

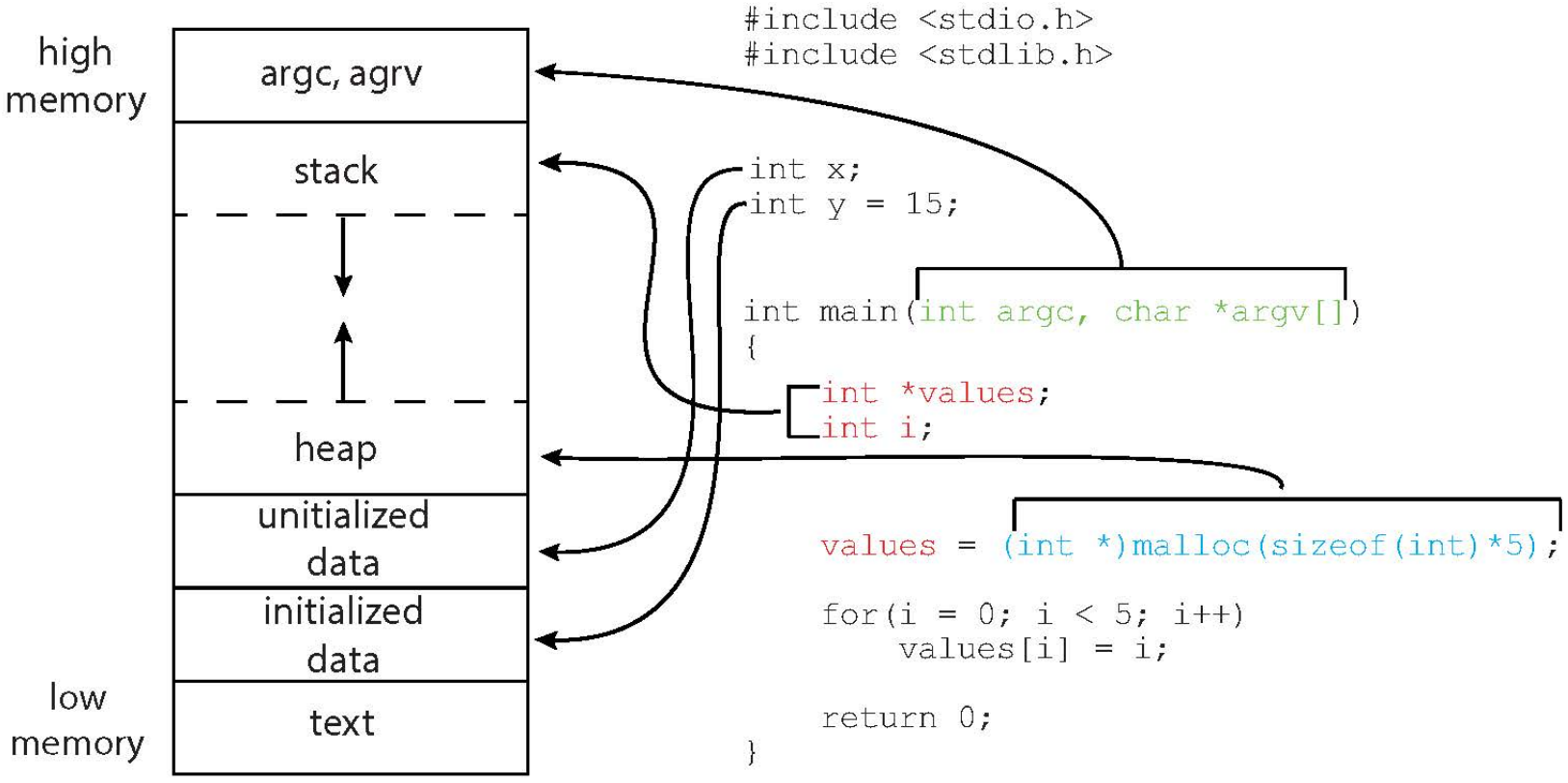
- Um processo é mais do que apenas o código binário:
  - A parte de um processo que contém o código é conhecida como seção de texto (*text section*)
  - O processo também armazena informações sobre suas atividades atuais
    - Program Counter
    - Também armazena o conteúdo de outros registradores do processador (*processor register*)
  - A Pilha de execução (*execution stack*) contém dados temporários como parâmetros de funções, variáveis locais, endereços de retorno
  - A seção de dados (*data section*) contém as variáveis globais
  - Também pode incluir o monte[2] (*heap*) que é a memória alocada dinamicamente durante a execução do processo.

[2] Apesar desta tradução aparecer em diversos livros em português, na prática utiliza-se mais frequentemente o termo em inglês *heap*

Estrutura de um processo em memória



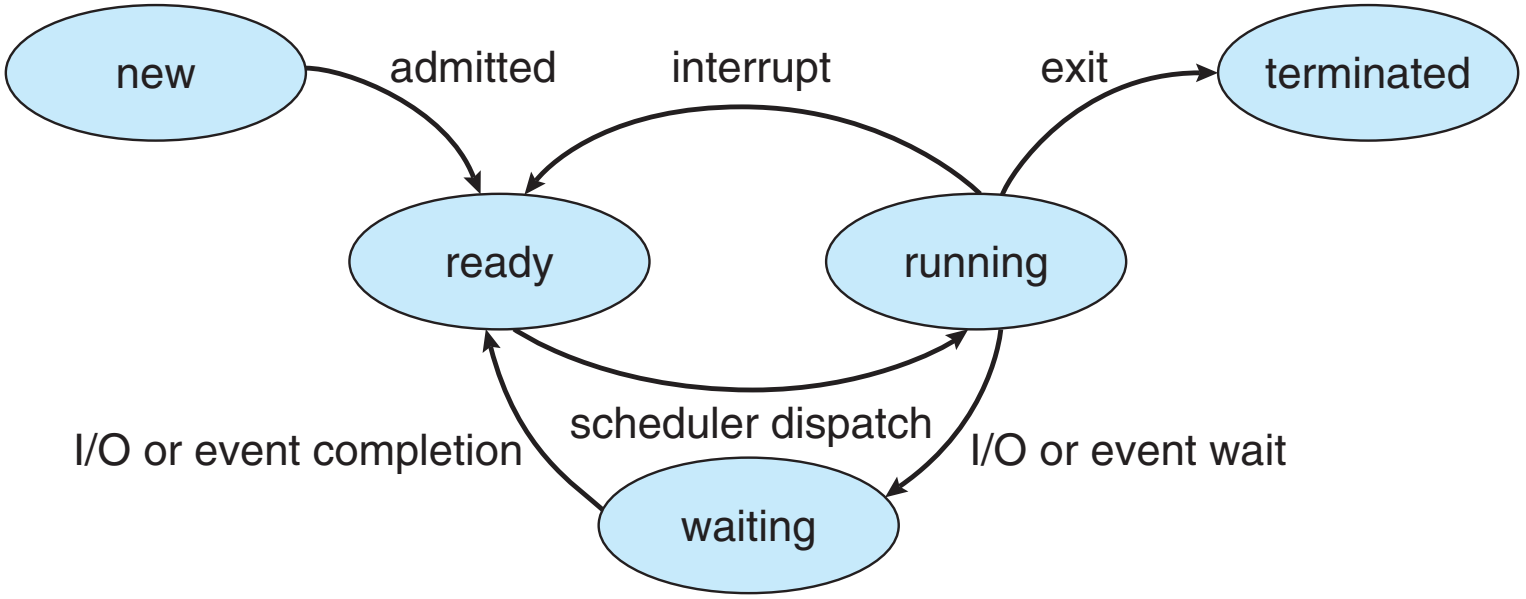
Estrutura de um processo em memória



Estados de um processo

UFABC

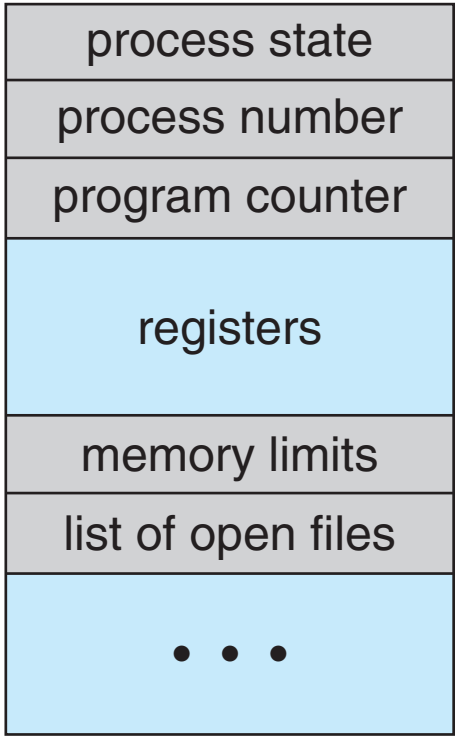
- Conforme um processo executa ele troca de estados
  - novo → o processo está sendo criado
  - executando → instruções estão sendo executadas
  - esperando → o processo está aguardando algum evento
  - pronto → o processo está aguardando ser atribuído a um processador
  - finalizado → o processo finalizou a sua execução



Bloco de controle de processos (PCB)

UFABC

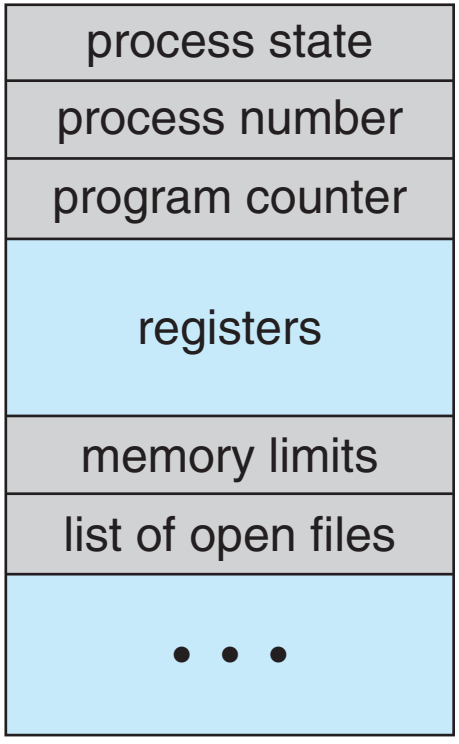
- As informações associadas a cada um dos processos são armazenadas em uma estrutura chamada bloco de controle de processos (*process control block* ou task control block)
  - Estado do processo - executando, novo, ...
  - Program counter - localização da próxima instrução a ser executada
  - Registradores de CPU - conteúdo de todos os registradores relacionados à execução do processo
  - Prioridades de escalonamento na CPU, ponteiros para as filas de escalonamento
  - ...



Bloco de controle de processos (PCB)

UFABC

- ...
  - Informações de gerenciamento de memória - controla a memória alocada por um processo
  - Informações de contabilidade - quantidade de CPU utilizada, tempo passado desde a inicialização, limites de tempo, ...
  - Informações do status de operações de E/S - Dispositivos de E/S alocados ao processo, lista de arquivos abertos, ...



Threads

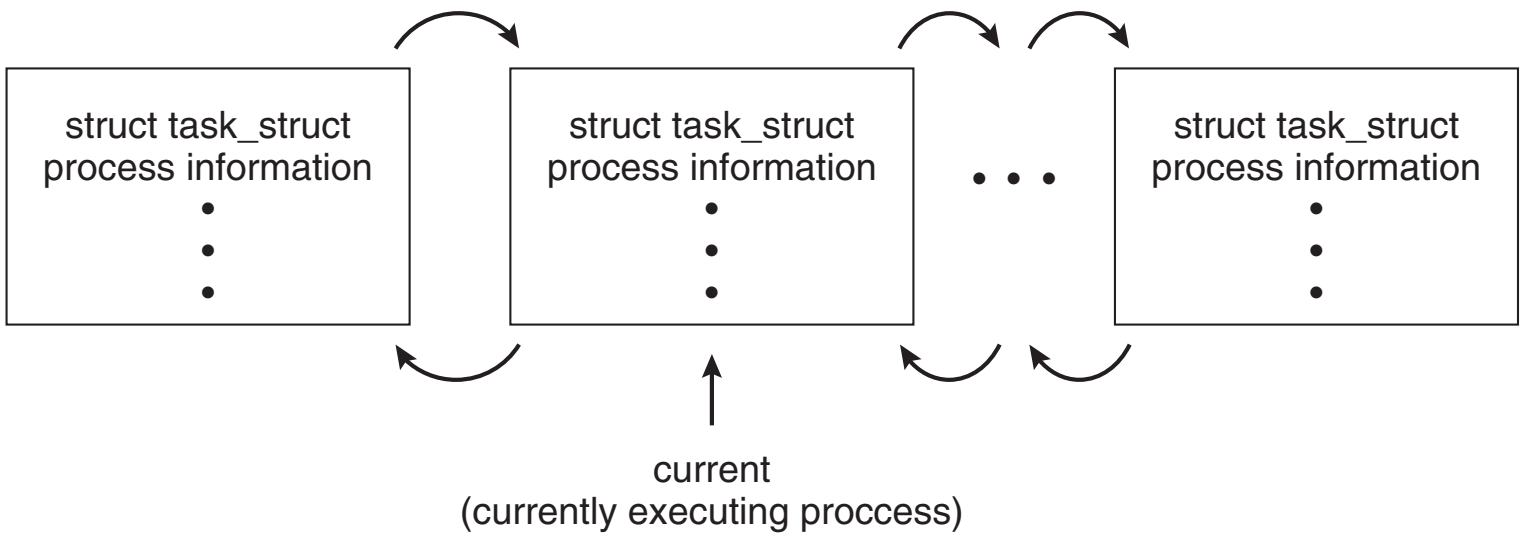
- Até agora só falamos de processos que têm uma única linha de execução
- Considere agora se nós multiplicássemos alguns dados do processo como, por exemplo, os program counters no PCB
  - Assim, múltiplas localizações no binário poderiam ser executadas ao mesmo tempo
  - Múltiplas linhas de execução $\rightarrow$ threads
  - Vamos falar de threads nas aulas seguintes

Representação de processos no Linux

UFABC

No Linux, cada processo é representado por uma struct em C chamada `task_struct`

```
1  pid t_pid; /* process identifier */
2  long state; /* state of the process */
3  unsigned int time_slice /* scheduling information */
4  struct task_struct *parent; /* this process's parent */
5  struct list_head children; /* this process's children */
6  struct files_struct *files; /* list of open files */
7  struct mm_struct *mm; /* address space of process */
```



Escalonamento de processos

Escalonamento de processos

UFABC

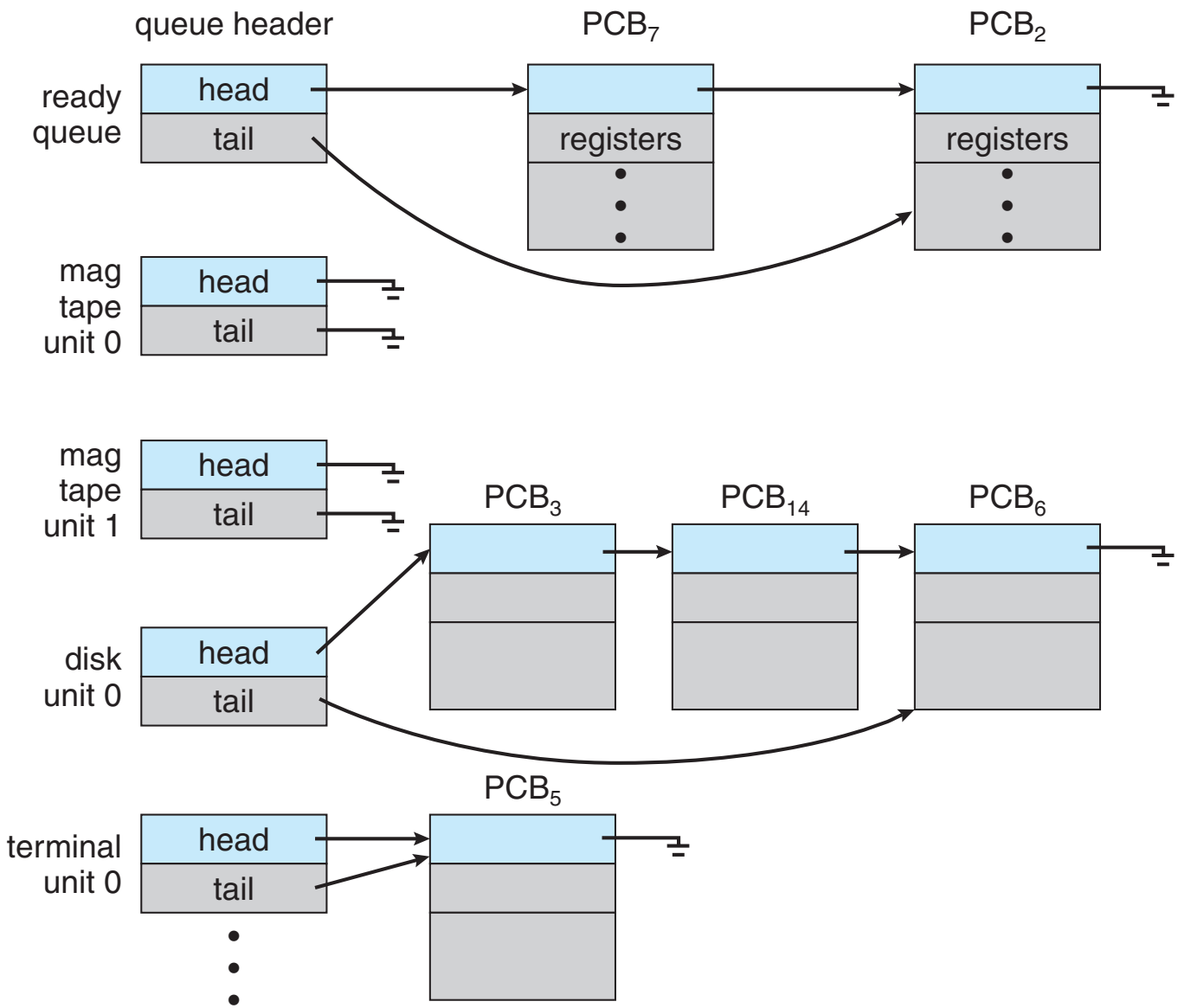
- Tem como meta maximizar o uso da CPU
  - Para isto utiliza troca rápida entre processos para que a CPU permaneça sempre ocupada
- O processo ”abre mão”da CPU em duas situações
  - Requisição de E/S
  - Após *N* unidades de tempo terem decorrido desde que ele foi colocado para executar
- Quando um processo abre mão da CPU ele é colocado em uma fila que contém os processos prontos a serem executados
- O escalonador de processos (*process scheduler*) seleciona dentre os processos prontos a serem executados qual será o próximo a obter o uso da CPU

Filas de escalonamento

UFABC

- O SO mantém filas de escalonamento (*scheduling queues*) de processos
  - Fila de trabalho (*job queue*) mantém todos os processos do sistema
  - Fila de prontos (*ready queue*) mantém uma lista com todos os processos prontos a serem executados
  - Filas de dispositivos (*device queues*) mantém os processos aguardando pelo uso de um dispositivo de E/S
- Os processos migram entre as filas durante a sua execução

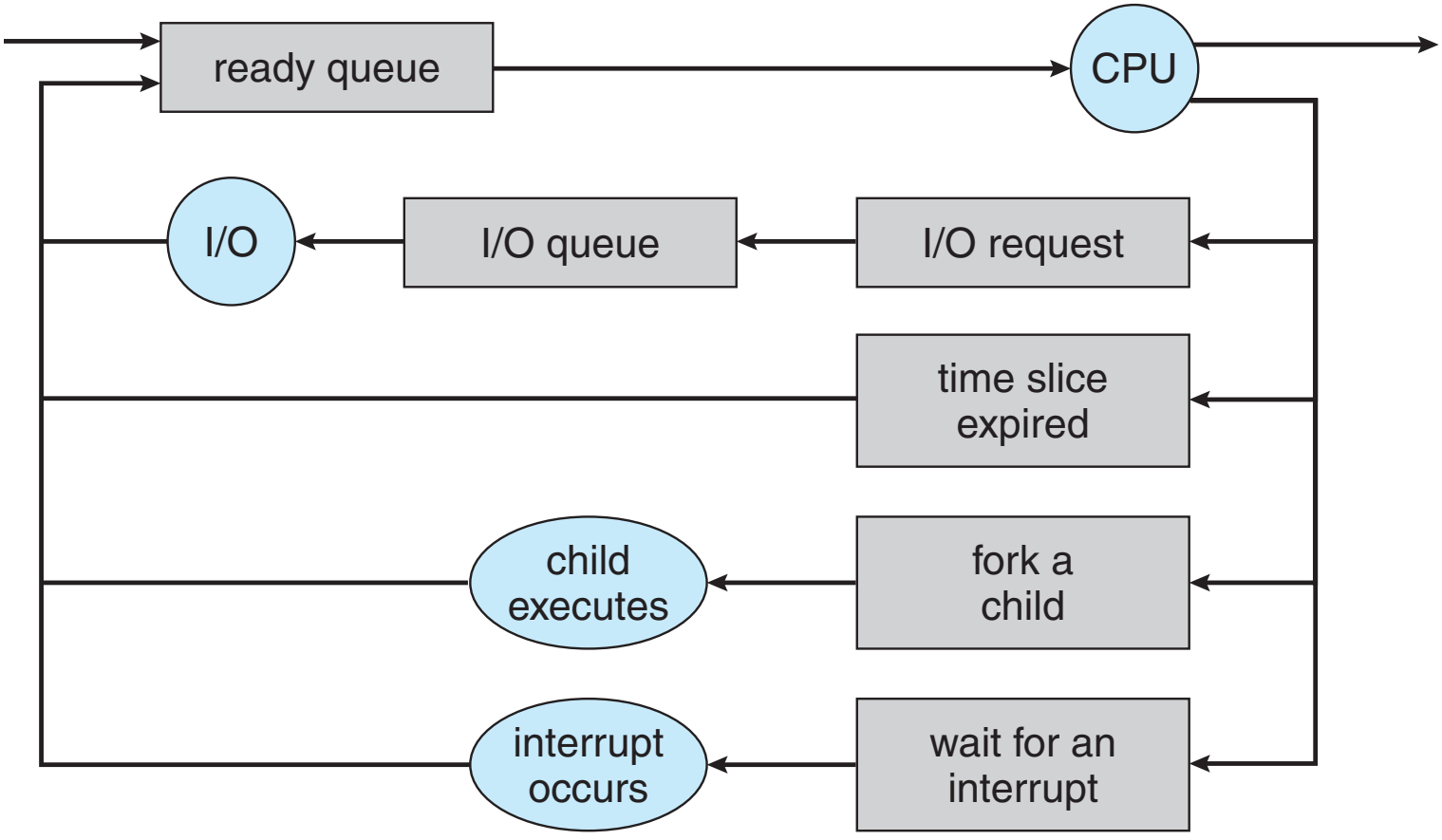
Fila de prontos e diversos dispositivos de E/S



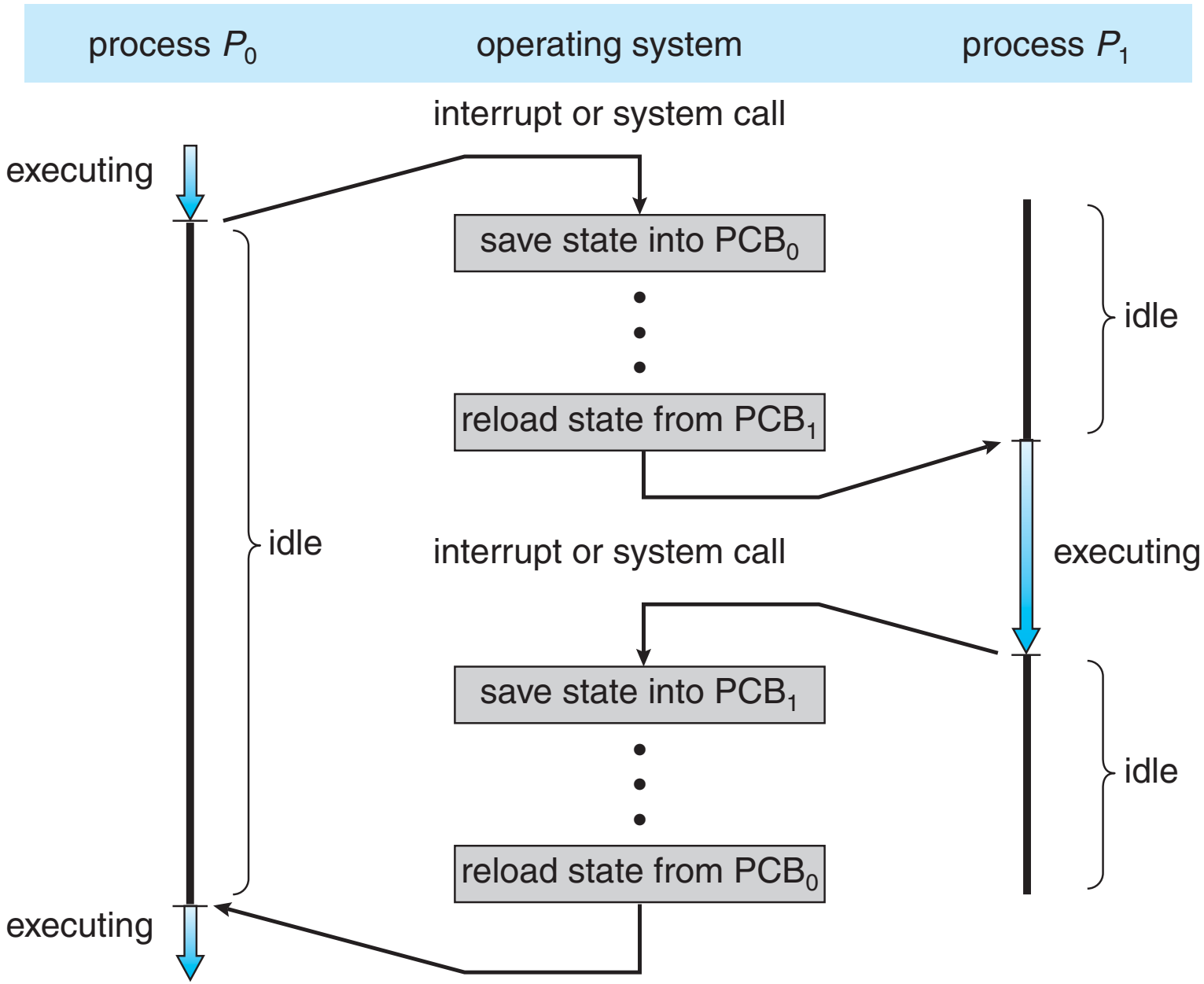
Representando o escalonamento de processos

UFABC

Um diagrama de enfileiramento (*queuing diagram*) representa filas, recursos, fluxos, ...



Troca de processos na CPU



Escalonadores

UFABC

- O escalonador de CPU (*CPU scheduler*) seleciona qual processo deve ser executado em seguida e aloca a CPU
  - Às vezes é único escalonador no sistema
  - É um escalonador de curto prazo, é invocado frequentemente (várias vezes por segundo) → precisa ser rápido
- O escalonador de tarefas (*job scheduler*) seleciona quais processos devem ser trazidos para a fila de prontos
  - O escalonador de tarefas é chamado com uma baixa frequência (segundos, minutos, ...) → não precisa ser tão rápido

Escalonadores e tipos de processos

UFABC

- Processos podem ser descritos como
  - Limitados por E/S (*I/O bound*) - Gasta mais tempo aguardando por operações de E/S do que computação. Uso alto de CPU em vários intervalos curtos de tempo (*short CPU bursts*)
  - Limitados pela CPU (*CPU bound*) - Gasta a maior parte do tempo utilizando a CPU para fazer computação e pouco tempo esperando por operações de E/S. Uso de CPU alto por intervalos longos de tempo (*long CPU bursts*)
- O escalonador de tarefas tenta sempre encontrar um bom *process mix*

Multitasking em sistemas móveis

UFABC

- Alguns sistemas móveis (como as primeiras versões do iOS) permitiam apenas um processo executando por vez
- Isso mudou no iOS 4
  - Apenas um processo em primeiro plano (*foreground*) - controlado pela interface do usuário
  - Múltiplos processos em segundo plano (*background*) - na memória e em execução mas não na tela, e com alguns limites
    - Tarefas curtas e simples para receber notificações
    - Tarefas longas e específicas como tocar música em background
- Android permite execução em segundo plano com menos limitações
  - Processos em segundo plano utilizam serviços (*services*)
  - Serviço pode permanecer ativo ainda que o processo seja suspenso
  - Serviço sem interface com o usuário e com pouca memória

Troca de contexto

UFABC

- Quando a CPU troca o processo executando na CPU, o SO precisa salvar o estado do processo sendo retirado e carregar o estado salvo do processo novo através de uma troca de contexto (*context switch*)
- O contexto (*context*) de um processo é armazenado/representado pelo PCB
- O tempo de troca de contexto é considerado como sendo overhead puro → o sistema não faz trabalho útil durante a troca
  - Quanto mais complexo o SO e o PCB, maior o tempo de troca
- O tempo na prática depende do suporte oferecido pelo hardware
  - Algumas CPUs fornecem diversos conjuntos de registradores por CPU o que permite que diversos contextos estejam carregamos em um dado momento

Operações com processos

Operações com processos

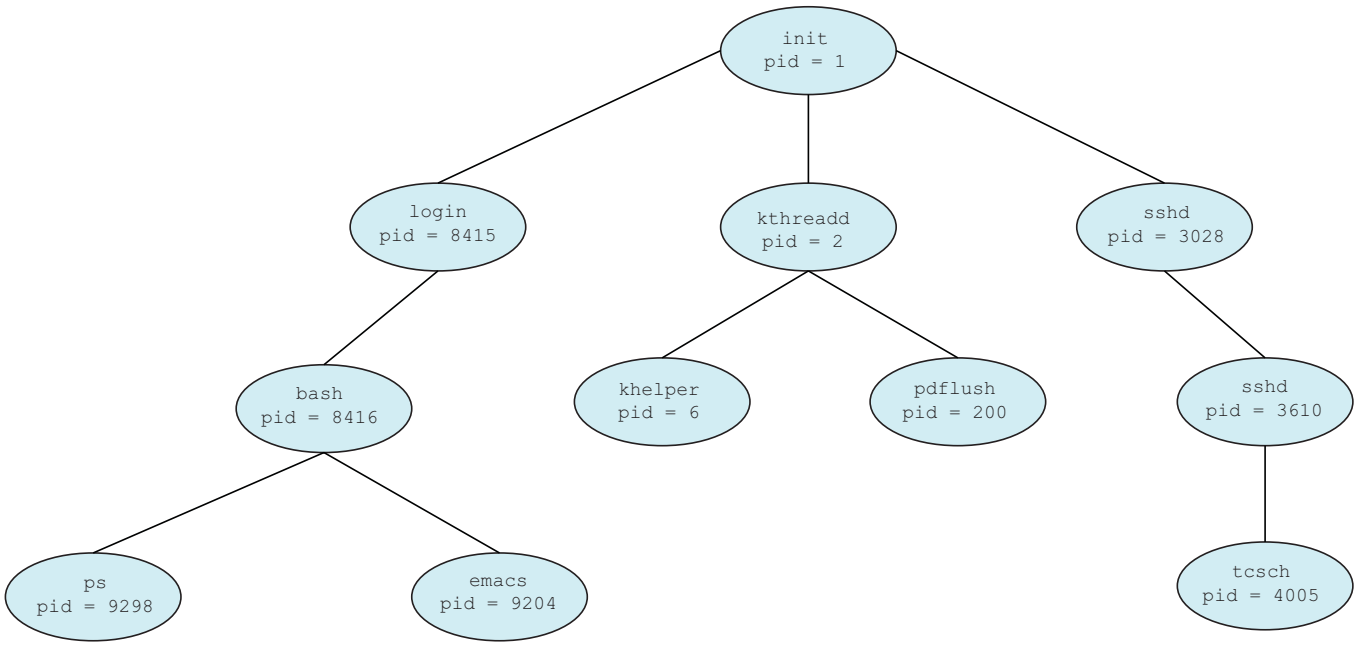
UFABC

- O SO precisa prover os mecanismos para
  - Criação de processos
  - Finalização de processos
  - ... (detalharemos a seguir)

Criação de processos

UFABC

- Um processo pode criar outros processos
- Um processo pai (*parent process*) cria processos filhos (*children processes*) que por sua vez podem criar outros filhos
  - Cria-se uma árvore de processos
- É comum identificarmos cada processo por um número, o seu identificador (*process ID*, PID)

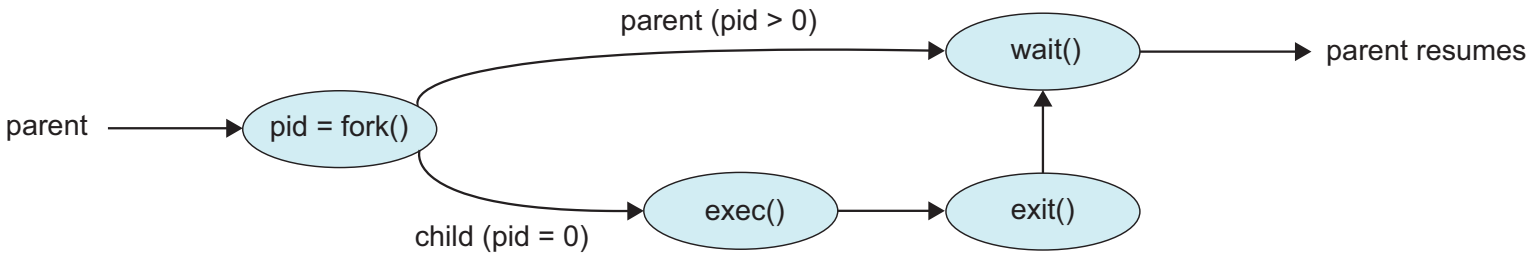


UFABC

- Opções de compartilhamento de recursos entre pais e filhos
  - Pais e filhos compartilham todos os recursos
  - Filhos têm acesso apenas a uma parte dos recursos do pai
  - Pais e filhos não compartilham nada
- Opções de execução
  - Pais e filhos executam concorrentemente
  - Pai aguarda até que os filhos terminem

UFABC

- Espaço de endereçamento
  - Filho recebe um espaço de endereçamento que é uma duplicata do pai
  - Filho carrega um programa no espaço de endereçamento
- Exemplos no UNIX
  - `fork()` cria um novo processo
  - `exec()` usada após um `fork()` para substituir o espaço de memória por um novo programa



Programa em C para criar um processo separado no UNIX

UFABC

```c
int main() {
    pit_t pid;
    /*fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, “Fork Failed”);
        return 1;
    }
    else if (pid == 0) { /*child process */
        execlp(“/bin/ls”,”ls”,NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf(“Child Complete”);
    }
    return 0;
}
```

Fork em C

UFABC

- O programa cria um novo processo no UNIX
- Após o `fork()` há dois processos distintos executando o mesmo programa
  - A única diferença é o valor de `pid` que é 0 no processo filho e o PID do filho no pai
- O processo filho herda: direitos, arquivos abertos, atributos de escalonamento, ...
- O processo filho ”sobrescreve”o seu espaço de endereçamento com a chamada à `execp` (uma versão do `exec`) executando o comando `ls`
- O processo pai aguarda a finalização do filho (`wait()`)
- Quando o filho termina a execução, o pai retoma a sua execução e finaliza chamando `exit()`

Criando outro processo em C - Windows

UFABC

```
1  int main(VOID) {
2      STARTUPINFO si;
3      PROCESS_INFORMATION pi;
4
5      /* allocate memory */
6      ZeroMemory(&si, sizeof(si));
7      si.cb = sizeof(si);
8      ZeroMemory(&pi, sizeof(pi));
```

Criando outro processo em C - Windows

UFABC

```
1  ...
2    /* create child process */
3    if (!CreateProcess(NULL, /* use command line */
4            “C:\\WINDOWS\\system32\\mspaint.exe”,/*
             ↪  command */
5            NULL, /* don’t inherit process handle */
6            NULL, /* don’t inherit threat handle */
7            FALSE, /* disable handle inheritance */
8            0, /* no creation flags */
9            NULL, /* use parent’s environment block */
10           NULL, /* use parent’s existing directory */
11           &si,
12           &pi))
13       {
14   ...
```

Criando outro processo em C - Windows

UFABC

```
    ...
        {
            fprintf{stderr, “Create Process Failed”};
            return -1;
        }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf(“Child Complete”);

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Finalização do processo

UFABC

- Um processo termina a execução quando ele executa a sua última instrução e pede sua finalização pela syscall `exit()`
  - O processo pode devolver um status (tipicamente um número) que pode ser obtido pelo pai pela syscall `wait()`
  - O SO desaloca os recursos que haviam sido alocados para o processo
- Um pai pode finalizar a execução do filho utilizando a syscall `abort()`
  - O filho pode ter excedido o uso de recursos
  - A tarefa que havia sido atribuída ao filho não é mais necessária
  - O pai está finalizando sua execução e o SO não permite que um filho continue a execução sem o pai

Finalização do processo

UFABC

- Alguns SOs não permitem que um filho sobreviva à morte do pai. Assim, se um processo finaliza sua execução todos os filhos finalizam também
  - Finalização em cascata
- O processo pai pode esperar a finalização do filho utilizando a syscall `wait()`. Ela devolve o PID do processo terminado e o status

```
1  pid = wait(&status);
```

- Se o processo pai não invocou `wait()` o processo filho vira um zumbi (*zombie*)
- Se o processo pai for terminado antes que pudesse invocar a função `wait()` o processo se torna um órfão (*orphan*)

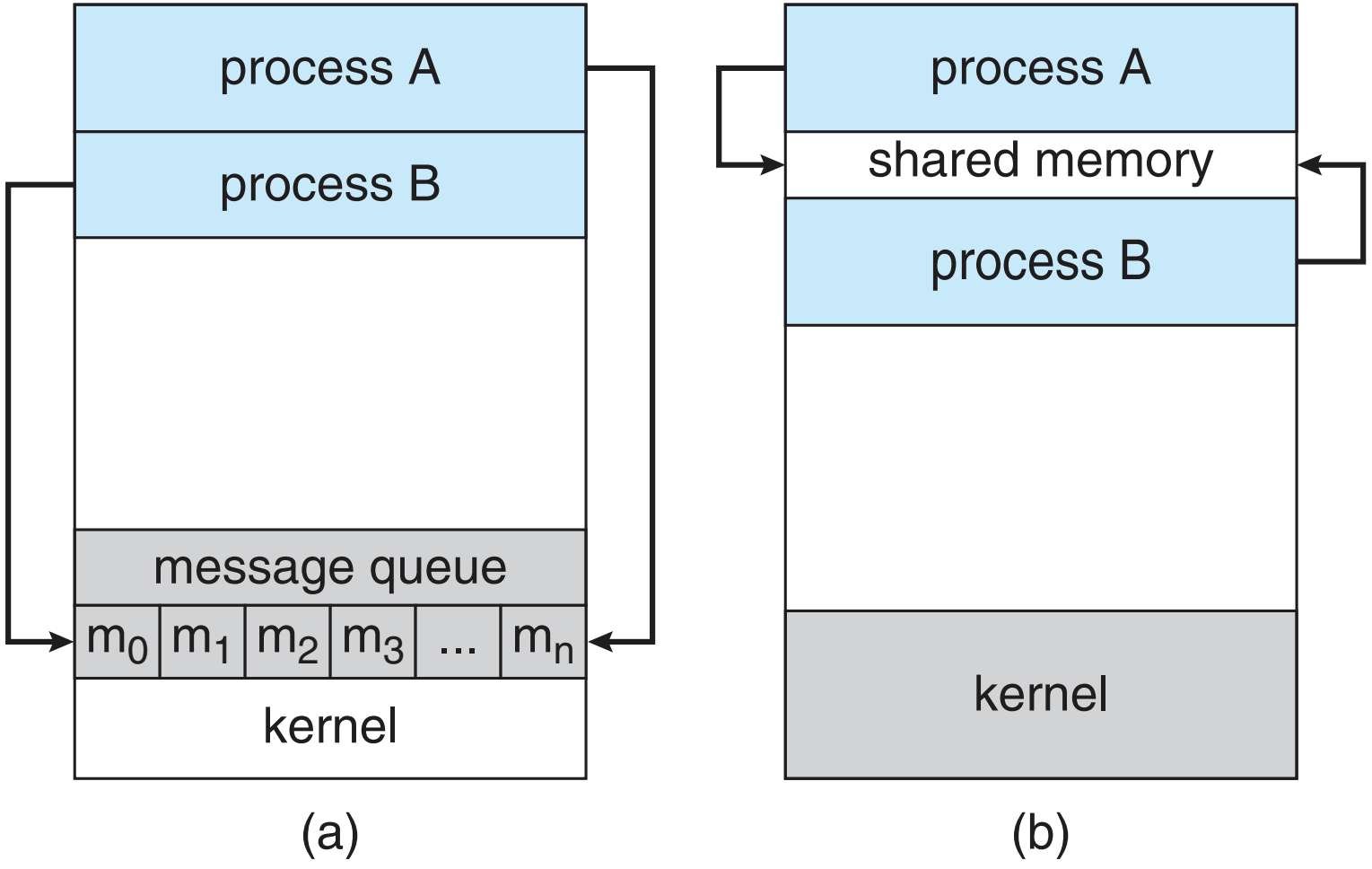
Comunicação inter-processo

Comunicação inter-processo

UFABC

- Processos em uma máquina podem ser independentes ou cooperativos
  - Processos cooperativos podem afetar (ou serem afetados por) outros processos
    - Compartilhamento de dados é uma das maneiras
  - Processos independentes não afetam outros processos
- Razões para utilizar processos independentes
  - Compartilhamento de informações e recursos
  - Aumento de desempenho computacional (paralelismo)
  - Modularidade
  - Conveniência
- Para que dois processos possam cooperar, eles precisam de um mecanismo de comunicação inter-processo (*interprocess communication*, IPC)

- Há dois modelos de IPC
  - Troca de mensagens (*message passing*)
  - Memória compartilhada (*shared memory*)



Memória compartilhada

UFABC

- Uma área de memória que é compartilhada (!) entre os processos que desejam se comunicar
- A comunicação está sob controle dos usuários e não do SO
- O problema mais sério é como prover um sistema de controle de comunicações que permita aos usuários sincronizar as suas ações quando acessam a região compartilhada de memória

UFABC

- Processos cooperativos que acessam dados compartilhados precisam sincronizar o acesso para garantir a consistência dos dados
- Manter a consistência requer mecanismos para garantir a execução ordenada dos processos cooperativos
- Ilustração do problema - O problema do produtor consumidor
  - O processo produtor produz informação que é consumida por um processo consumidor
  - A informação (produto) do produtor é passada para o consumidor por um buffer
  - Dois tipos de buffer costumam ser utilizados
    - buffer ilimitado (*unbounded buffer*) - sem limite no tamanho (limitado apenas pela memória da máquina)
    - buffer limitado (*bounded buffer*) - assume um tamanho fixo de buffer

Solução 1 - Bounded Buffer

UFABC

- Dados compartilhados

```
1  #define BUFFER_SIZE 10
2  typedef struct {
3      ...
4  } item;
5
6  item buffer[BUFFER_SIZE];
7  int in = 0;
8  int out = 0;
```

- A solução apresentada nos próximos slides é correta, mas apenas 9 dos 10 espaços do buffer podem ser utilizados

Solução 1 - Bounded Buffer - Produtor

UFABC

```
1  item next_produced;
2  while (true) {
3      /* produce an item in next produced */
4      while (((in + 1) % BUFFER_SIZE) == out)
5          ; /* do nothing */
6      buffer[in] = next_produced;
7      in = (in + 1) % BUFFER_SIZE;
8   }
```

Solução 1 - Bounded Buffer - Consumidor

UFABC

```
item next_consumed;
while (true) {
    while (in == out)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
 }
```

Sistemas baseados em troca de mensagens

UFABC

- Mecanismo para processos comunicarem e sincronizarem suas ações
  - Sem utilizar memória compartilhada
- O mecanismo oferece duas operações
  - `send(message)`
  - `receive(message)`
- O tamanho da mensagem pode ser fixo ou variável

UFABC

- Se os processos P e Q desejam se comunicar, eles:
  - Estabelecem um link de comunicação
  - Trocam mensagens via `send` / `receive`
- Problemas de implementação
  - Como links são estabelecidos?
  - Um link pode ligar mais de um par de processos?
  - Quantos links podem existir entre cada par de processos?
  - Qual a capacidade de um link?
  - O tamanho da mensagem é fixo ou variável?
  - O link é uni ou bidirecional?

Implementação de um link de comunicação

UFABC

- Físico
  - Memória compartilhada
  - Barramento
  - Rede
- Lógico
  - Direto ou indireto
  - Síncrono ou assíncrono
  - Buffering automático ou explícito

Comunicação direta

UFABC

- Processos precisam dar um nome uns aos outros de maneira explícita
  - `send(P, message)` - Envia mensagem ao processo `P`
  - `receive(Q, message)` - Recebe mensagem do processo `Q`
- Propriedades do link de comunicação
  - Link estabelecido automaticamente
  - Um link é associado a um par de processos que desejam se comunicar
  - Entre cada par há exatamente um link
  - O link pode ser unidirecional, mas tipicamente é bidirecional

Comunicação indireta

UFABC

- Mensagens são entregues e enviadas para caixas de correio (*mailbox*) também conhecidas como portas (*ports*)
  - Cada mailbox tem um identificador único
  - Processo podem se comunicar apenas se eles compartilham uma caixa de correio
- Operações
  - Criar uma nova caixa de correio (porta)
  - Enviar e receber mensagens pela caixa de correio
  - Apagar uma caixa de correio
- As primitivas são definidas como
  - `send (A, message)` - envia mensagem à caixa de correio A
  - `receive (A, message)` - recebe mensagem da caixa de correio A

Comunicação indireta

UFABC

- Propriedades do link de comunicação
  - Link é estabelecido apenas se os processos compartilham uma caixa de correio
  - Um link pode ser associado com muitos processos
  - Cada par de processos pode compartilhar vários links de comunicação
  - Link pode ser uni ou bidirecional

Problemas com a Comunicação indireta

UFABC

- Compartilhamento de caixas de correio
  - P_1, P_2, P_3 compartilham a caixa A
  - P_1 envia. P_2 e P_3 recebem
  - Qual dos processos recebe a mensagem?
- Soluções
  - Permitir que um link seja associado a no máximo 2 processos
  - Permitir que apenas um processo por vez execute a operação `receive`
  - Permitir ao sistema que escolha arbitrariamente o recebedor. Remetente é notificado sobre qual processo recebeu a mensagem

Mecanismos bloqueantes e não-bloqueantes

UFABC

- A troca de mensagens pode ser bloqueante ou não
- Bloqueante é considerado síncrono (*synchronous*)
  - **Blocking send** - o remetente fica bloqueado até que a mensagem tenha sido recebida
  - **Blocking receive** - o recebedor fica bloqueado até que uma mensagem esteja disponível
- Não-bloqueante é considerado assíncrono (*asynchronous*)
  - **Non-blocking send** - o remetente envia a mensagem e prossegue a sua execução
  - **Non-blocking receive** - o recebedor recebe
    - Uma mensagem ou
    - Nulo (*null*)
  - Várias combinações são possíveis, mas quando tanto o `send` quanto o `receive` são bloqueantes temos um rendezvous

Buffering

UFABC

- Seja a comunicação direta ou indireta, as mensagens trocadas por processos que se comunicam são armazenadas em uma fila temporária
- Tais filas podem ser implementadas de três maneiras
  1. Capacidade zero - Nenhuma mensagem é armazenada no link. O remetente deve esperar pelo recebedor (rendezvous)
  2. Capacidade limitada (*bounded capacity*) - Número finito de mensagens que podem ser armazenadas. O remetente precisa aguardar caso o link esteja completamente cheio.
  3. Capacidade ilimitada (*unbounded capacity*) - buffer com capacidade *infinita*. Remetente nunca espera.

Produtor-Consumidor com rendezvous

UFABC

- A sincronização de Produtor-consumidor é trivial quando se utiliza rendezvous
- Produtor

```
1  message next_produced;
2  while (true) {
3      /* produce an item in next produced */
4      send(next_produced);
5   }
```

- Consumidor

```
1  message next_consumed;
2  while (true) {
3      receive(next_consumed);
4
5      /* consume the item in next consumed */
6   }
```

Exemplos de IPC

Exemplos de IPC

UFABC

- POSIX API para memória compartilhada
- Mach OS, que utiliza troca de mensagens
- Windows IPC, que usa memória compartilhada como um mecanismo para fornecer certos tipos de troca de mensagem
- Pipes, um dos mais antigos métodos de IPC em sistemas UNIX

POSIX - Produtor

UFABC

```c
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main() {
    /* the size (in bytes) of shared memory object */
    const int SIZE 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* strings written to shared memory */
    const char *message_0 = "Hello";
    const char *message_1 = "World!";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;
```

UFABC

```
1       ...
2       /* create the shared memory object */
3       shm fd = shm open(name, O_CREAT | O_RDRW, 0666);
4       /* configure the size of the shared memory object */
5       ftruncate(shm_fd, SIZE);
6       /* memory map the shared memory object */
7       ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd,
        ↪  0);
8       /* write to the shared memory object */
9       sprintf(ptr,"%s",message_0);
10      ptr += strlen(message_0);
11      sprintf(ptr,"%s",message_1);
12      ptr += strlen(message_1);
13      return 0;
14  }
```

POSIX - Consumidor

UFABC

```c
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main() {
    /* the size (in bytes) of shared memory object */
    const int SIZE 4096;
    /* name of the shared memory object */
    const char *name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory obect */
    void *ptr;
    ...
```

UFABC

```
    ...
    /* open the shared memory object */
    shm_fd = shm_open(name, O RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd,
    ↪  0);
    /* read from the shared memory object */
    printf("%s",(char *)ptr);
    /* remove the shared memory object */
    shm unlink(name);
    return 0;
}
```

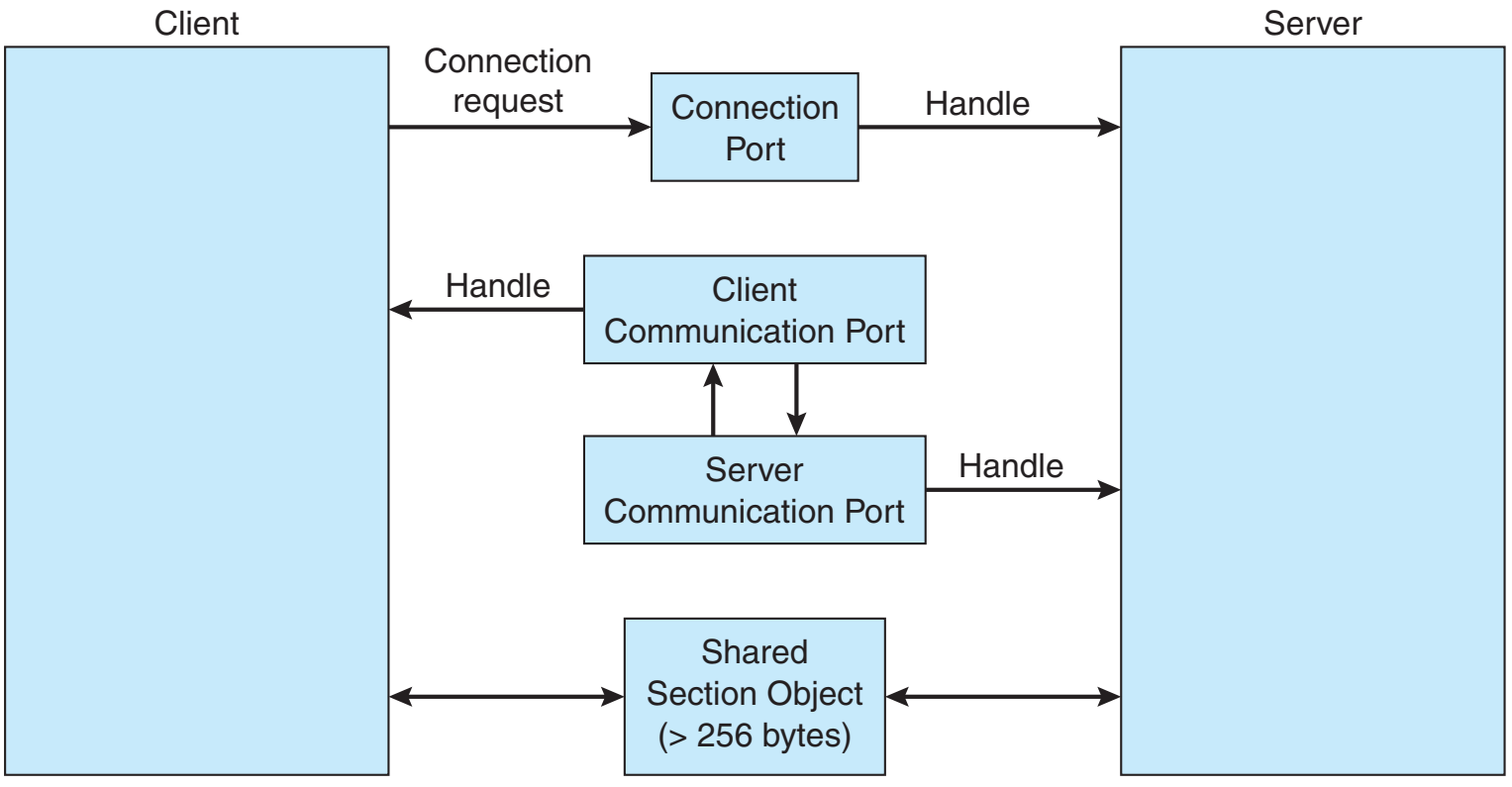
UFABC

- No Mach, a comunicação é baseada em troca de mensagens
- Até chamadas de sistema são mensagens
- Cada tarefa recebe duas caixas de correio quando criada: kernel e notify
- Apenas 3 chamadas de sistema são necessárias para transferência de mensagens
  - `msg_send()`, `msg_receive()`, `msg_rpc()`
- Caixas de mensagem necessárias para comunicação são criadas pela função `port_allocate()`
- Envio e recebimento são flexíveis, por exemplo, há 4 opções se a caixa de correio estiver cheia
  - Aguardar
  - Aguardar até no máximo *n* milissegundos
  - Retorna imediatamente
  - Faz um cache temporário da mensagem

UFABC

- A troca de mensagens é feita via um mecanismo chamado *advanced local procedure call* (LPC)
- Só funciona com processos no mesmo sistema
- Utiliza portas, como caixas de correio, para estabelecer e manter canais de comunicação
- Comunicação funciona da seguinte maneira
  - O cliente abre um handle (uma referência abstrata a um recurso) ao objeto *porta de comunicação* do subsistema
  - O cliente envia uma requisição de conexão
  - O servidor cria uma porta de comunicação privada e devolve o handle ao cliente
  - O servidor e o cliente usam o handle recebido para enviar mensagens ou callbacks e para receber respostas às requisições

Local Procedure Calls - Windows



Pipes

UFABC

- Agem como um conduíte que permite a dois processos se comunicarem
- Problemas
  - Comunicação é uni ou bidirecional?
  - Se for bi-direcional é half ou full-duplex?
  - É preciso haver um relacionamento pai-filho entre os processos que querem se comunicar?
  - Os pipes podem ser utilizados através de uma rede?
- Pipes comuns - não podem ser acessados de fora do processo que os criou. Tipicamente um processo pai cria um pipe e o utiliza para comunicar com o filho
- Pipes nomeados - Podem ser acessados por qualquer processo (independentemente da sua relação hierárquica)

Pipes comuns

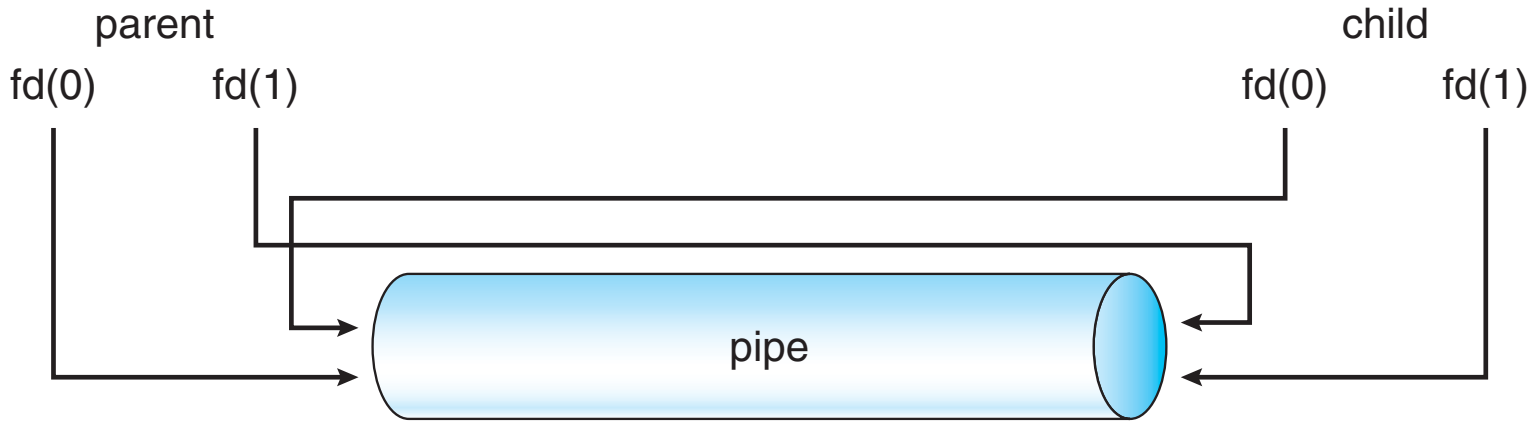
UFABC

- Permitem comunicações no estilo produtor-consumidor
- Produtor escreve em um extremo do pipe (write-end)
- Consumidor escreve no outro extremo do pipe (read-end)
- Pipes comuns são unidirecionais
- Se uma comunicação bidirecional for necessária, é preciso utilizar dois pipes
- Necessitam relacionamento pai-filho para comunicação

Pipes comuns

UFABC

- No UNIX, pipes podem ser criados utilizando a função `pipe(int fd[])`
- A função cria um pipe acessível via `fd[]`
  - `fd[0]` é o extremo de leitura do pipe
  - `fd[1]` é o extremo de escrita do pipe
- O UNIX trata pipes como um tipo especial de arquivo. Assim, pipes podem ser lidos e escritos através das chamadas `read()` e `write()` comuns a arquivos



Pipes comuns

UFABC

- O Windows chama esses pipes de ”pipes anônimos”
- Veja a última aula prática sobre como criar pipes
- Veja também o livro texto [SGG] para outros exemplos de código

Pipes nomeados

UFABC

- Pipes nomeados (*named pipes*) são mais poderosos que pipes comuns
- A comunicação é bidirecional
- Não é necessário uma relação pai-filho
- Vários processos podem usar o mesmo pipe para se comunicar
- Disponível tanto no Windows quanto nos UNIX

Comunicação em sistemas cliente-servidor

UFABC

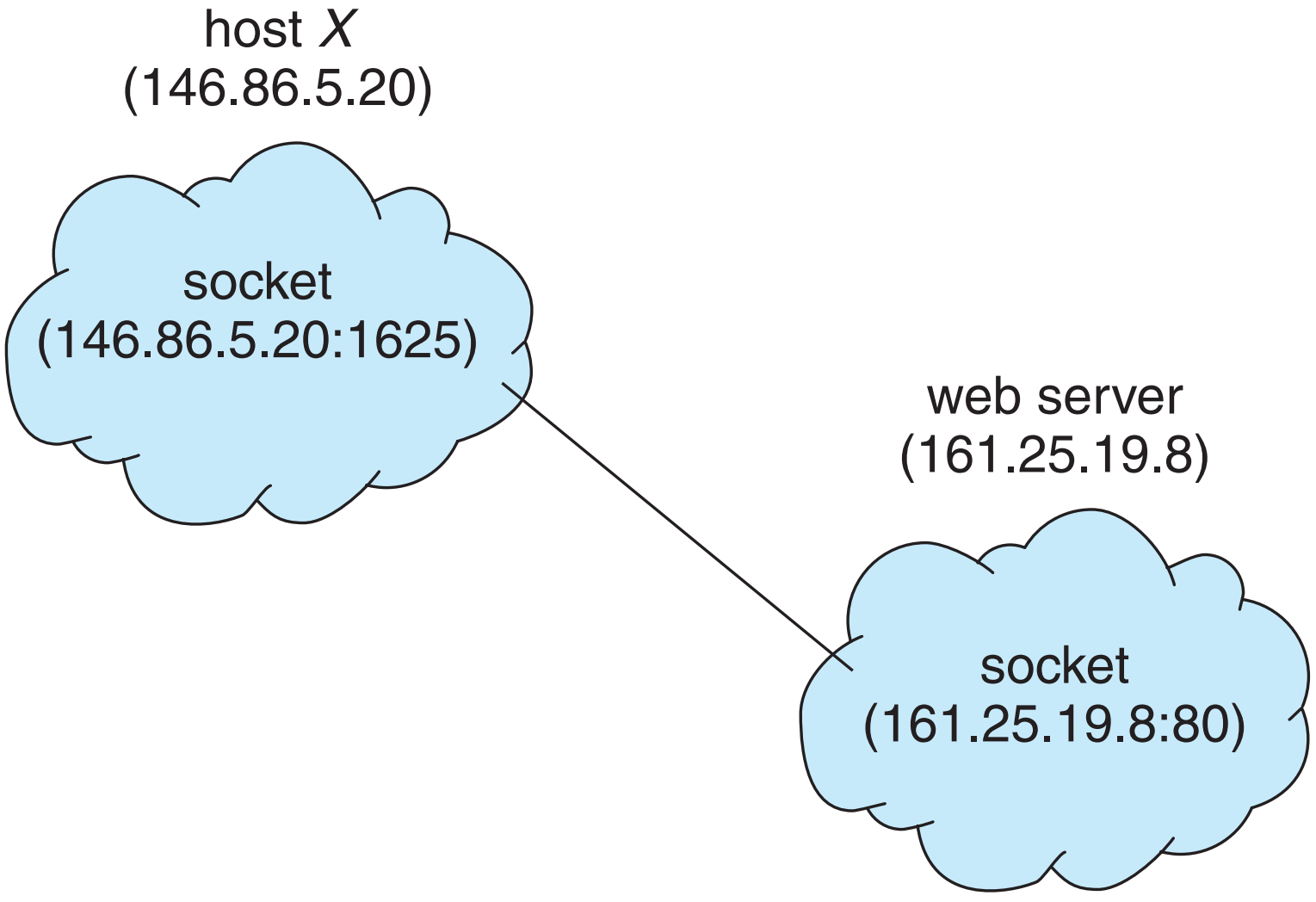
- Soquetes (*sockets*)
- Chamadas de procedimento remoto (*remote procedure calls*, RPC)
- Chamadas de método remoto (*remote method call*)

Soquetes

UFABC

- Um soquete (*socket*) é definido como um *endpoint* para comunicação
- Os pacotes trazem endereço e porta para diferenciar o processo destinatário na máquina destino
- O soquete **208.80.154.224:80** se refere à porta **80** no host **208.80.154.224**
- A comunicação envolve sempre um par de soquetes
- Portas com valores abaixo de 1024 são bem conhecidas e servem requisições de serviços padrão
- O endereço 127.0.0.1 é especial e se refere à própria máquina onde está executando
  - Chamado de endereço de loopback

UFABC


host X
(146.86.5.20)
socket
(146.86.5.20:1625)
web server
(161.25.19.8)
socket
(161.25.19.8:80)

Exemplo de uso de soquetes

UFABC

- Há três tipos de soquetes
  - Orientado a conexão - TCP
  - Sem conexão - UDP
  - Multicast - dado pode ser enviado a múltiplos destinatários

Exemplo de uso de soquetes - Java - Servidor de data

UFABC

```java
import java.net.*;
import java.io.*;
public class DateServer {
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);
      /* now listen for connections */
      while (true) {
        Socket client = sock.accept();
        PrintWriter pout = new
            PrintWriter(client.getOutputStream(), true);
        /* write the Date to the socket */
        pout.println(new java.util.Date().toString());
        /* close the socket and resume */
        /* listening for connections */
        client.close();
      }
```

Exemplo de uso de soquetes - Java - Servidor de data

UFABC

```
1       ...
2       } catch (IOException ioe) {
3         System.err.println(ioe);
4       }
5     }
6   }
```

Remote Procedure Calls

UFABC

- Chamadas remotas de procedimento (RPC) abstraem a chamada de procedimentos através de uma rede
  - Utiliza portas para diferenciar os serviços
- Stubs - proxy do lado do cliente para o procedimento real do lado do servidor
- O stub localiza o servidor e empacota (*marshalls*) os parâmetros
  - O empacotamento deve levar em consideração que o dado será transmitido pela rede
- O stub do lado do cliente recebe a mensagem, desempacota (*unmarshall*) os parâmetros, faz a chamada
- No Windows, o stub é gerado a partir de uma linguagem de especificação de interfaces (MIDL, Microsoft Interface Definition Language)

UFABC

- Os dados devem ser representados de uma maneira agnóstica à arquitetura
- Considere a representação de inteiros de 32-bits
  - Little-endian - Armazena o byte mais significativo antes
  - Big-endian - Armazena o byte menos significativo antes
  - Big-endian é o formato mais utilizado para redes. É inclusive chamado de network byte order
- Comunicação remota pode falhar muito mais facilmente
  - Mensagens podem ser entregues **exatamente uma vez** (em contraposição a **no máximo uma vez**)
- SOs normalmente fornecem um serviço (matchmaker) para auxiliar a conexão cliente e servidor

Execução do RPC