Sistemas Distribuídos
Aula 2

Aula passada

- Logísitica
- Regras do jogo
- Definição e características
- Exemplos
- Objetivos

Aula de hoje

- Processos
- IPC
- Características
- Ex. sinais, *pipes*, *sockets*

Processos

- O que é um *processo*?
  - no contexto de sistemas operacionais

É um programa em execução!

- É a abstração de *execução* do SO
  - unidade de execução, de escalonamento, de endereçamento, de estado de contexto
  - programa se torna um processo (aka. *job, task*)
- Uma das principais tarefas do SO: gerenciar processos

Representando Processos

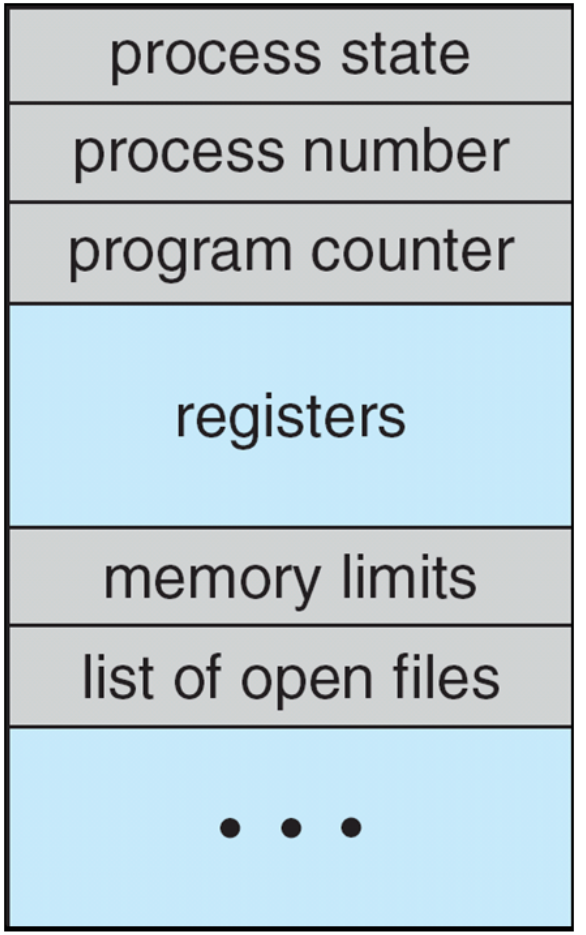
- O que é necessário para o SO representar um processo?

Todo o estado de um programa em execução!

- Espaço de endereçamento (memória do processo)
- Código a ser executado (programa)
- Dados das variáveis do programa (estática e dinâmicas)
- Pilha de execução (chamada de funções)
- Contador de programa (PC)
- Registradores importantes (como o PC)
- Recursos do SO sendo utilizados: descritores de arquivos, sockets, outros dispositivos de I/O, etc
- muitas outras coisas...

Process Control Block (PCB)

- Estrutura de dados mantida pelo SO para cada processo
  - *todo* estado do processo
  - inclusive valor dos registradores (da CPU) quando não executando
- Armazenada em *kernel space*
  - memória usada pelo SO
- Grande quantidade de dados
  - atualizada a “cada instante”



Processos são pesados para o SO!

Troca de Contexto

- O que é isto? Para que serve?

Troca do processo que está em execução!

- CPU (ou *core*) pode executar apenas um processo de cada vez
- SO coloca e remove processos em execução
  - ao remover, salva "*todo*" o estado da CPU
  - ao colocar, restaura "*todo*" o estado CPU
- Durante execução, PCB também é atualizado
- Troca de contexto: 100 a 1000 vezes por segundo!

Realmente incrível!

Estado do Processo

- Indica o que o processo está *fazendo*
- Quais são os estados de um processo?

- ***Running***: executando instruções na CPU
- ***Ready***: aguardando para ser escalonado na CPU
- ***Waiting***: aguardando algum evento para continuar execução (ex. usuário teclar alguma coisa)

- Em que estado um processo qualquer passa a maior parte do tempo?
- Quantos processos podem estar no estado *Running*?
- Quantos processos um SO gerencia normalmente?

Processos se Comunicando

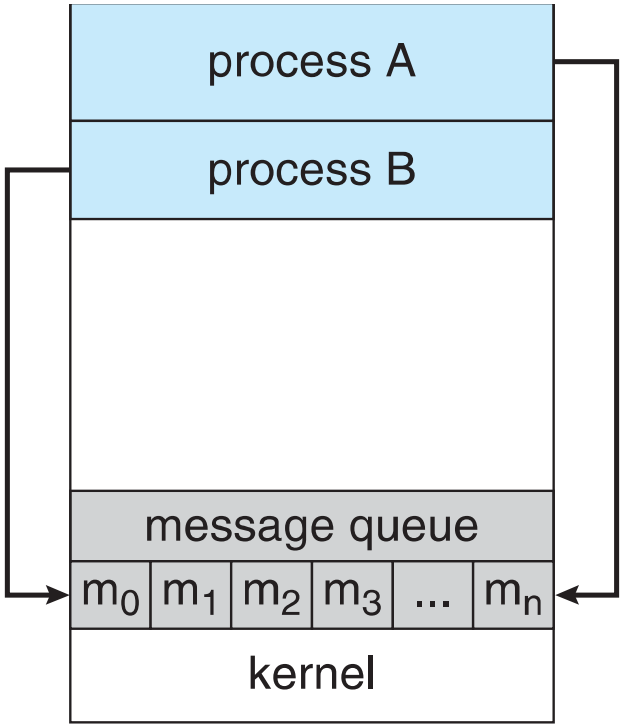
- Processos muitas vezes precisam *interagir*
  - razões para isto?
- SO precisa oferece mecanismos para isto

Interprocess Communication (IPC)

- Dois modelos de comunicação
  - memória compartilhada
  - troca de mensagens
- Para cada modelo, diferentes técnicas e implementações

Modelos de IPC

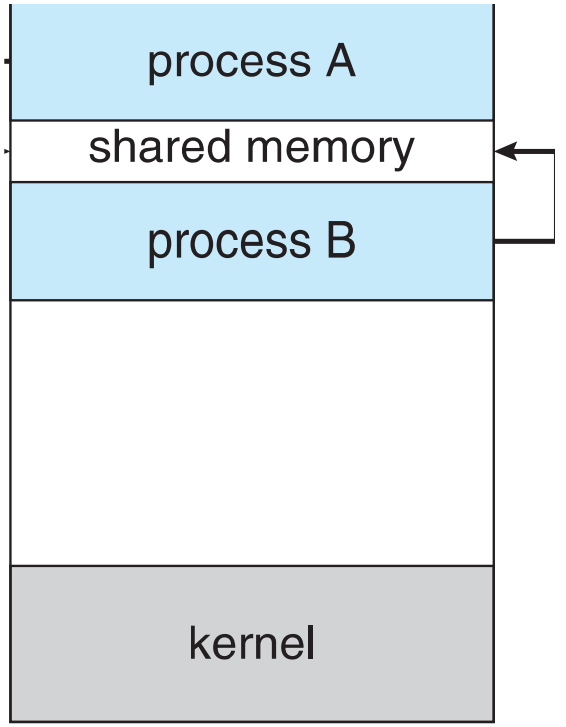
- Troca de mensagens



(a)

- canal de comunicação explícito através de chamadas *send/receive*
- fila de mensagens em *kernel space*

- Memória compartilhada



(b)

- região de memória mapeada em dois ou mais processos
- memória fora do *kernel space*

Memória Compartilhada

- Uma única chamada ao sistema para definir e mapear memória (a ser compartilhada)
- SO não faz mediação de leitura/escrita
- Processos precisam coordenar
  - o que acontece quando os dois escrevem?

Fonte de grandes problemas!

- Coordenação através de ***sincronização***
- Veremos nas próximas aulas

Troca de Mensagens

- Processos trocam mensagens através de chamadas ao sistema (*system call*): *send/receive*
  - “enlace” de comunicação
  - não há memória compartilhada
  - chamada ao sistema para cada mensagem
- Fila de mensagens em *kernel space*
- Características do enlace/troca de mensagens
  - direto x indireto
  - síncrono x assíncrono
  - bufferização

Direto x Indireto

Direto

- Processo origem/destino precisa ser identificado
  - send(P, msg) : envia mensagem ao processo P
  - receive(Q, msg) : recebe mensagem do processo Q
- Enlace criado “automaticamente”
- Enlaces unidirecionais
- Ex. sinais

Indireto

- Mensagens enviadas, recebidas através de uma “caixa postal” (CP)
  - send(A, msg) : envia mensagem para CP A
  - receive(A, msg) : recebe mensagem da CP A
- Processos precisam criar e conectar a CP
- CP são bidirecionais
- Processos pode ter múltiplas CPs
- Ex. pipes, sockets

Síncrono x assíncrono

Síncrono

- Comunicação é bloqueante (*blocking*)
  - send(A, msg) : transmissor aguarda até o receptor receber
  - receive(A, msg) : receptor aguarda até mensagem chegar
- Aguardar = bloqueado: processo em estado *Waiting*
- Forte sincronia entre enviar e receber

Assíncrono

- Comunicação é não-bloqueante (*non-blocking*)
  - send(A, msg) : transmissor envia e continua execução
  - receive(A, msg) : recebe mensagem caso exista (erro caso contrário), e continua execução
- Processos não aguardam = não ficam bloqueados

Exemplo: Produtor-Consumidor

- Dois processos
  - produtor: gera informação
  - consumidor: processa informação
- Produtor envia informação ao consumidor

Produtor

```
...
message produced;
while (true) {
   produced = generate_item();
   send(A, produced);
}
```

Consumidor

```
...
message consumed;
while (true) {
   receive(A, consumed);
   process(consumed);

}
```

- O que acontece com cada caso send/receive bloqueante e não bloqueante?

Bufferização

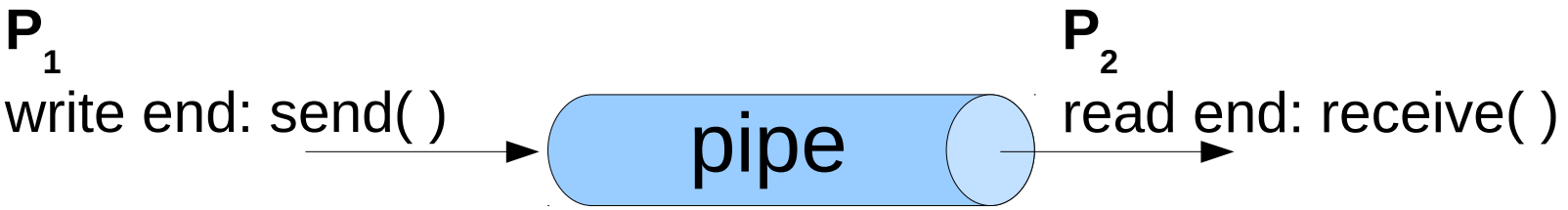
- Mensagens transmitidas podem ser armazenadas pelo “enlace” ou “canal”
  - tanto do lado do transmissor, quanto do lado do receptor (quando processos em SO diferentes)
- Armazenamento feito pelo SO, em *kernel space* (memória do SO)
- Tamanho do buffer é limitado, mas pode ser controlado pelo processo
- Permite transmissão assíncrona
  - “transmite” para o SO; processo contiuna; SO entrega mensagem

Sinais (*signals*)

- Forma primitiva de IPC, usada principalmente pelo SO para se comunicar com processos
- Processo envia mensagem (sinal) diretamente ao outro, de forma assíncrona
- Processo que recebe tem sua execução interrompida para processar a mensagem (sinal)
  - *signal handler* : função que processa a chegada da mensagem (sinal)
- Mensagem possui apenas um número inteiro, com diferentes significados
- Ex. comando *kill* no unix envia sinais

Pipes

- Canal de comunicação (unidirecional) entre processos
  - *anonymous pipes*, conduíte
- Pipe tem dois lados
  - *write end*: processo apenas escreve (send)
  - *read end*: processo apenas le (receive)



- send( ): assíncrono quando há espaço no pipe, síncrono caso contrário (aguarda até ter espaço)
- Ex. “|” no unix cria um pipe entre o *stdout* do primeiro processo e o *stdin* do segundo
- *named pipe* oferece maior funcionalidade (múltiplos processos no mesmo pipe)

Sockets

- Mecanismo mais comum e poderoso para comunicação entre processos
  - base de todos os outros mecanismos, como RPC
- Permite comunicação entre processos de SO diferentes
  - incluindo computadores interligados por rede
  - processo identificado pelo endereço IP e *porta* (número inteiro qualquer, acima de 1024)
- Dois tipos de enlace (sempre bi-direcional)
  - orientado a conexão (TCP): confiável, etc
  - sem conexão (UDP): não-confiável, etc
- API em todas as linguagens de programação