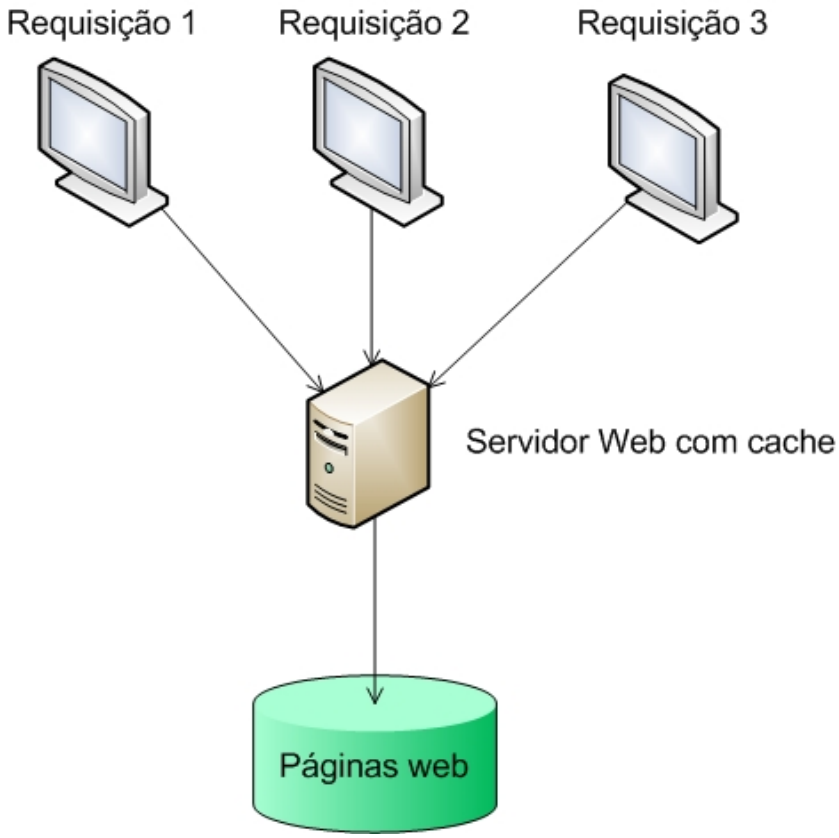
# Processos e threads

## Objetivos

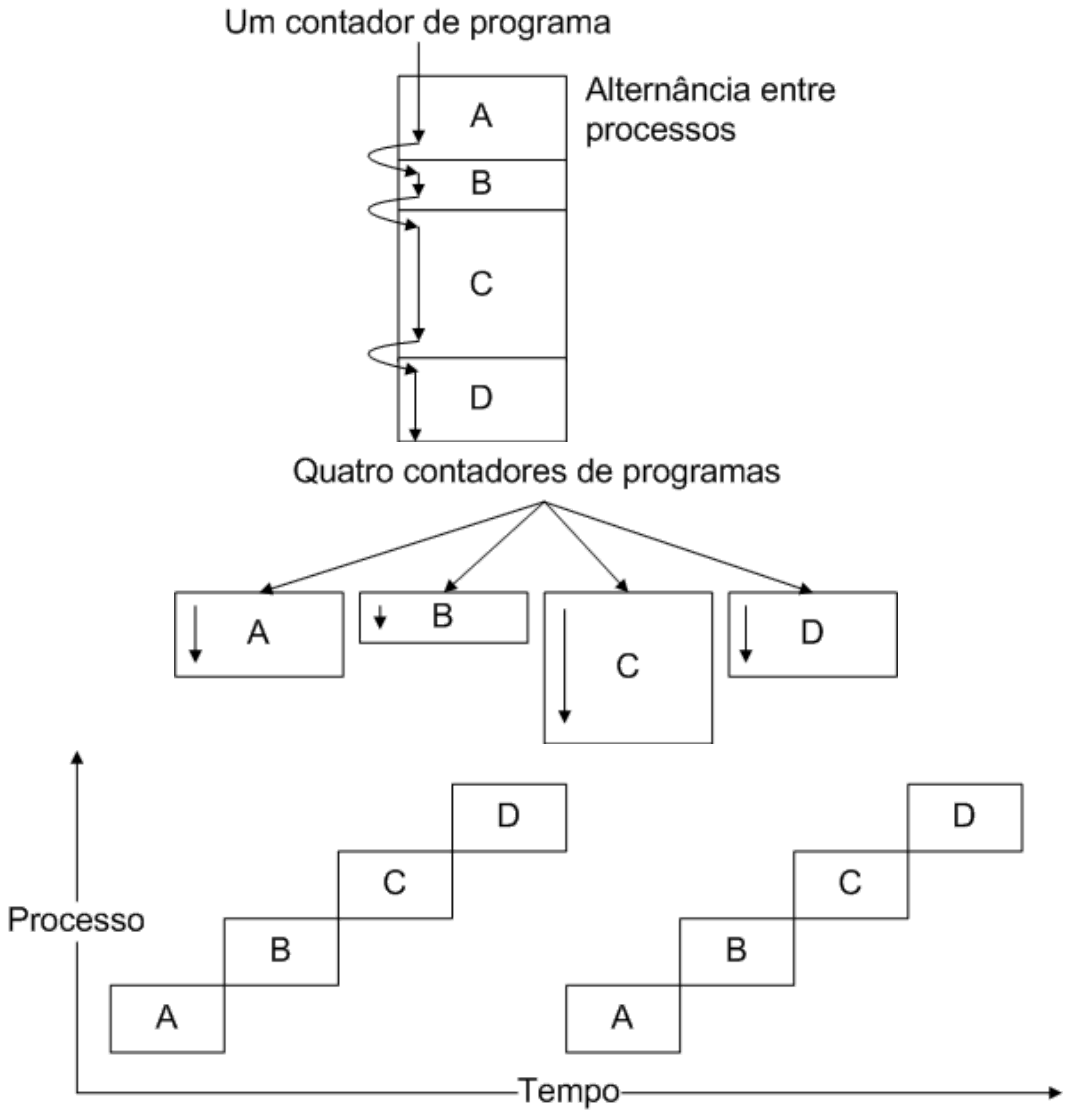
- Aprofundar conhecimentos sobre processos
- Conhecer o conceito de threads
- Compreender as restrições e os algoritmos para comunicação entre processos
- Diferenciar algoritmos de escalonamento de processos

## Processos

- Abstração de programa em execução
- Capacidade dos computadores modernos
  - Executar vários processos ao mesmo tempo
- Exemplos
  - Servidor web
    - Múltiplas requisições de páginas
    - Nem sempre ocorre cache hit
    - Algumas requisições não devem esperar busca em disco




- Chaveamento de programa para programa
  - Dezena ou centena de milisegundos de execução
- Pseudoparalelismo
  - Monoprocessadores
    - Única CPU com memória compartilhada
  - Multiprocessadores
    - Várias CPUs com memória compartilhada
- Processo sequencial ou processo
  - Executado passo a passo para se atingir objetivo
  - Conjunto de valores
    - Contador de programa
    - Registradores
    - Variáveis

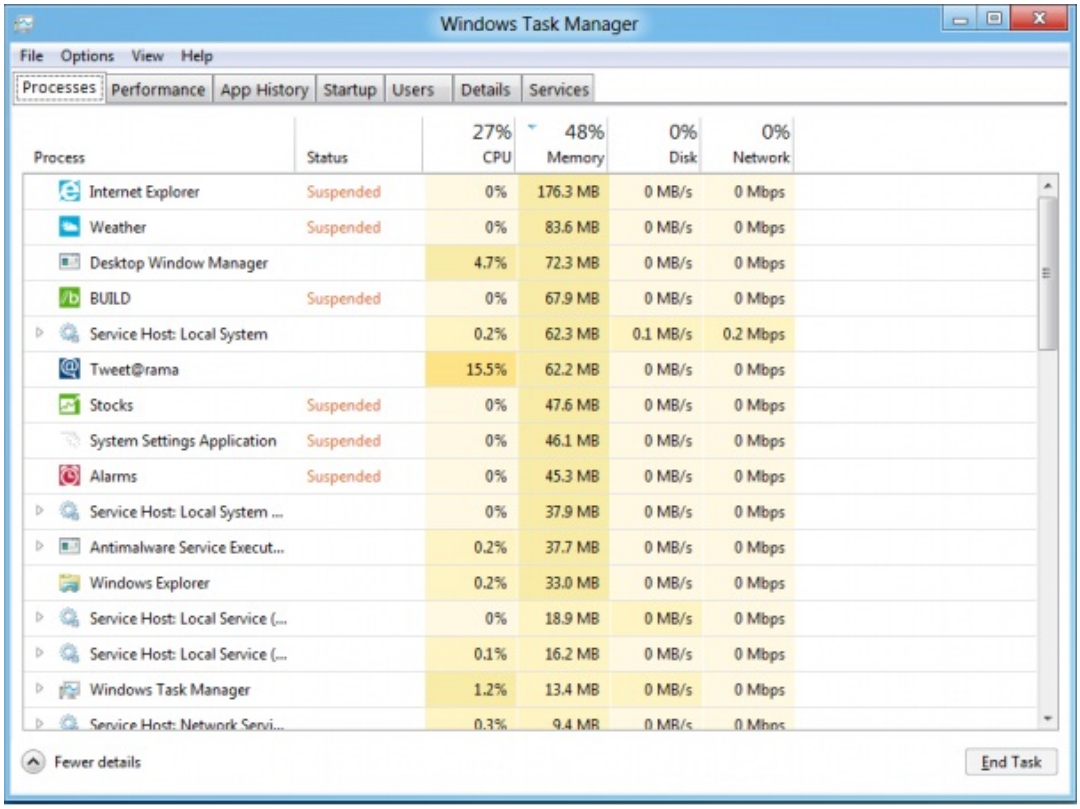



## Criação de processos

- Sistemas específicos como controlador de microondas
  - Único processo iniciado com sistema operacional
- Sistemas de propósito geral
  - Mecanismo para criação e encerramento de processos
- Eventos
  - Início do sistema
  - Execução de chamada de sistema de criação de processo por processo já em execução
  - Requisição de usuário
  - Início de tarefa em lote (batch job)
- Classificação
  - Primeiro plano ou foreground
    - Interagem com usuário
  - Segundo plano, background ou daemons
    - Não estão associados a usuário
    - Exemplo
      - Serviço de impressão
      - Detecção de conexão wireless
- Visualização
    - Programa ps no UNIX

```
root@host [~]# ps
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   1412   480 ?        S    12:38   0:00 init [3]
root     13644  0.0  0.1   2060 1104 ?        S    12:38   0:00 /bin/bash /etc/rc.d/rc 3
root     13996  0.0  0.0   1480   500 ?        S    12:38   0:00 syslogd -m 0
named    14010  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14011  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14012  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14013  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14014  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14015  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14016  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
named    14017  0.0  0.2 19100 2948 ?        S    12:38   0:00 /usr/sbin/named -u named
root     14028  0.0  0.0   1404   360 ?        S    12:38   0:00 /usr/sbin/courierlogger -pid=/var/spool/authd
root     14029  0.0  0.0   1736   548 ?        S    12:38   0:00 /usr/libexec/courier-authlib/authdaemond
root     14036  0.0  0.0   1736   168 ?        S    12:38   0:00 /usr/libexec/courier-authlib/authdaemond
root     14037  0.0  0.0   1736   168 ?        S    12:38   0:00 /usr/libexec/courier-authlib/authdaemond
root     14038  0.0  0.0   1736   168 ?        S    12:38   0:00 /usr/libexec/courier-authlib/authdaemond
root     14039  0.0  0.0   1736   168 ?        S    12:38   0:00 /usr/libexec/courier-authlib/authdaemond
root     14040  0.0  0.0   1736   168 ?        S    12:38   0:00 /usr/libexec/courier-authlib/authdaemond
root     32256  0.0  0.0   3532   984 ?        S    12:39   0:00 /usr/sbin/sshd
root     32265  0.0  0.0   2060 1016 ?        S    12:39   0:00 /bin/sh /usr/bin/mysqld_safe --datadir=/var/l
mysql    32286  0.0  1.4 64668 14884 ?        S    12:39   0:00 /usr/sbin/mysqld --basedir=/ --datadir=/var/l
mysql    32289  0.0  1.4 64668 14884 ?        S    12:39   0:00 /usr/sbin/mysqld --basedir=/ --datadir=/var/l
mysql    32290  0.0  1.4 64668 14884 ?        S    12:39   0:00 /usr/sbin/mysqld --basedir=/ --datadir=/var/l
mysql    32291  0.0  1.4 64668 14884 ?        S    12:39   0:00 /usr/sbin/mysqld --basedir=/ --datadir=/var/l
mysql    32292  0.0  1.4 64668 14884 ?        S    12:39   0:00 /usr/sbin/mysqld --basedir=/ --datadir=/var/l
mysql    32293  0.0  1.4 64668 14884 ?        S    12:39   0:00 /usr/sbin/mysqld --basedir=/ --datadir=/var/l
```

- Gerenciador de tarefas no Windows




- Criação
  - UNIX
    - Esquema permite redirecionar entrada-padrão, saída-padrão e erros-padrão
    - Chamada a fork
    - Seguida de execve
  - Windows
    - Uso do CreateProcess
- Espaço de endereçamento distinto
  - Ambos os casos

## Término de processos

- Processos são encerrados após sua execução
- Eventos
  - Voluntários
    - Saída normal

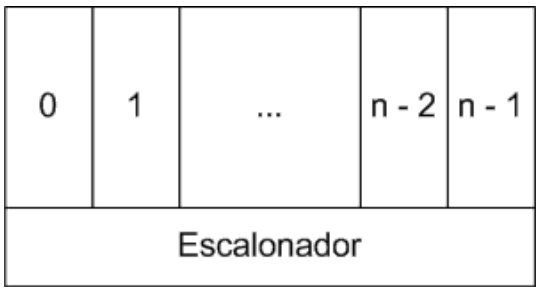
        - Saída por erro
    - Involuntários
        - Erro fatal
        - Cancelamento por outro processo
- Saída normal
    - Após encerramento da atividade proposta
        - Exemplo
            - Compilação de programa
    - UNIX
        - Chamada exit
    - Windows
        - Chamada ExitProcess
- Saída por erro
    - Impossível realizar tarefa
        - Exemplo
            - Arquivo inexistente fornecido para compilação
    - Programas interativos normalmente não saem
        - Apresentam erro em janela
- Erro fatal
    - Erro de programa
        - Exemplo
            - Execução de instrução ilegal
            - Referência a memória inexistente
            - Divisão por zero
    - Programa pode ser sinalizado que houve erro
        - Sistemas operacional deve ser avisado
        - Não provoca encerramento
- Cancelamento por outro processo
    - Processo solicita encerramento de outro
    - UNIX
        - Chamada kill
    - Windows
        - Chamada TerminateProcess

## Hierarquia de processos

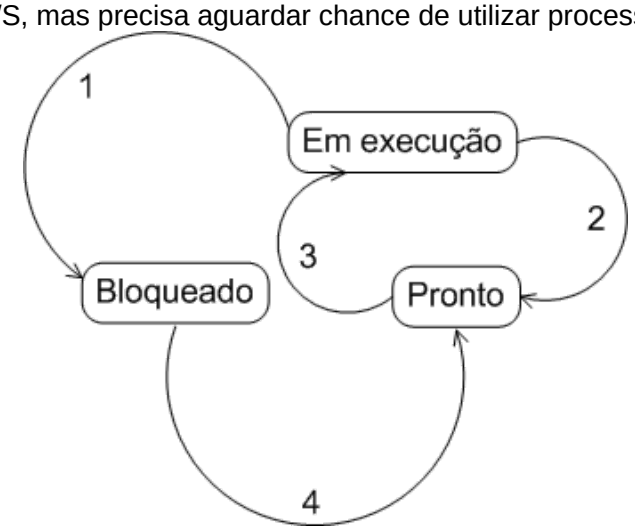
- Ligação entre processos
- UNIX
    - Hierarquia obrigatória
    - Processo init presente na iniciação do sistema
    - Bifurcação para cada terminal
    - Todo processo pertence a cadeia de init
- Windows
    - Utilizar handle
        - Identificador de processo
    - Hierarquia através da passagem de handle

## Estado de processos

- Estados possíveis
    - Em execução
        - Utilizando CPU
    - Pronto
        - Executável aguardando CPU
    - Bloqueado
        - Incapaz de executar aguardando ocorrência de evento
- Escalonador
    - Nível mais baixo do sistema operacional
    - Controla iniciação e bloqueio de processos




- Transições
  1. Processo aguardando E/S
  2. Tempo de utilização do processador encerrado
  3. Processo ganha direito de utilizar processador
  4. Processo encerrou E/S, mas precisa aguardar chance de utilizar processador



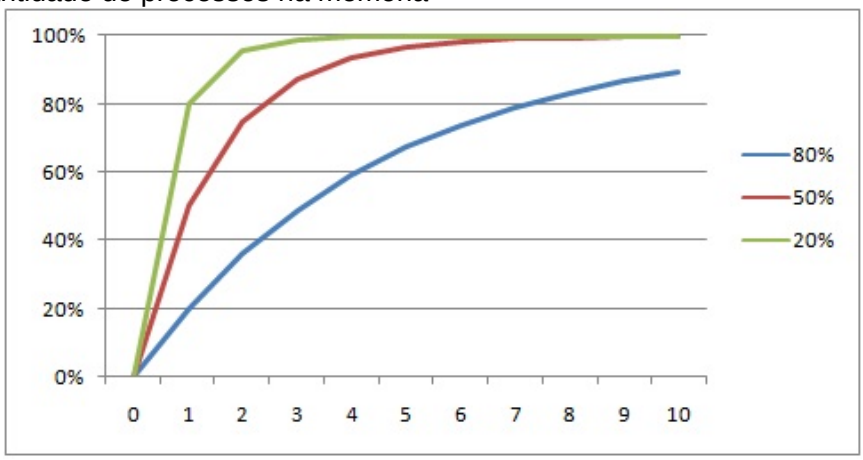

## Implementação de processos

- Tabela de processos
  - Tabela com entrada para cada processo
  - Blocos de controle de processo (Process Control Blocks)
- Informações no bloco
  - Gerenciamento de processo
    - Registros
    - Contador de programa (PC)
    - Palavra de estado do programa (PSW)
    - Ponteiro da pilha (SP)
    - Estado do processo
    - Prioridade
    - Parâmetros de escalonamento
    - ID do processo
    - Processo pai
    - Grupo de processo
    - Sinais
    - Momento que foi iniciado
    - Tempo de CPU usado
    - Tempo de CPU de processo filho
    - Tempo de alarme seguinte
  - Gerenciamento de memória
    - Ponteiro para informações sobre segmento de texto
    - Ponteiro para informações sobre segmento de dados
    - Ponteiro para informações sobre segmento de pilha
  - Gerenciamento de arquivo
    - Diretório raiz
    - Diretório de trabalho
    - Descritores de arquivo
    - ID do usuário
    - ID do grupo
- Arranjo de interrupções
  - Contém endereços de rotinas de serviços de interrupção

- Tratamento de interrupções
    1. Hardware empilha PC
    2. Hardware carrega novo PC a partir do arranjo de interrupções
    3. Procedimento em Assembly salva registradores
    4. Procedimento em Assembly configura nova pilha
    5. Serviço de interrupções em C executa
    6. Escalonador decide qual processo é o próximo a executar
    7. Procedimento em C retorna para código em Assembly
    8. Procedimento Assembly inicia novo processo atual

### Modelando a multiprogramação

- Emprego da CPU
    - Ponto de vista probabilístico
    - Eventos condicionais
- Grau de multiprogramação
    - GM = 1 - $p^n$
    - Em que
        - GM é grau de multiprogramação
        - p é percentual de espera
        - n é quantidade de processos na memória




- Exemplo
    - Condição inicial
        - Computador com 512MB
        - Sistema operacional utiliza 128MB
        - Três programas com 128MB cada
        - 80% do tempo esperando E/S
        - GM = 1 - $0,8^3$ = 49%
    - Adição de mais 512MB
        - n aumenta de 3 para 7
        - GM = 1 - $0,8^7$ = 79%
        - Ganho de 30 pontos percentuais
    - Adição de mais 512MB
        - n aumenta de 7 para 11
        - GM = 1 - $0,8^{11}$ = 91%
        - Ganho de apenas 12 pontos percentuais

# Threads

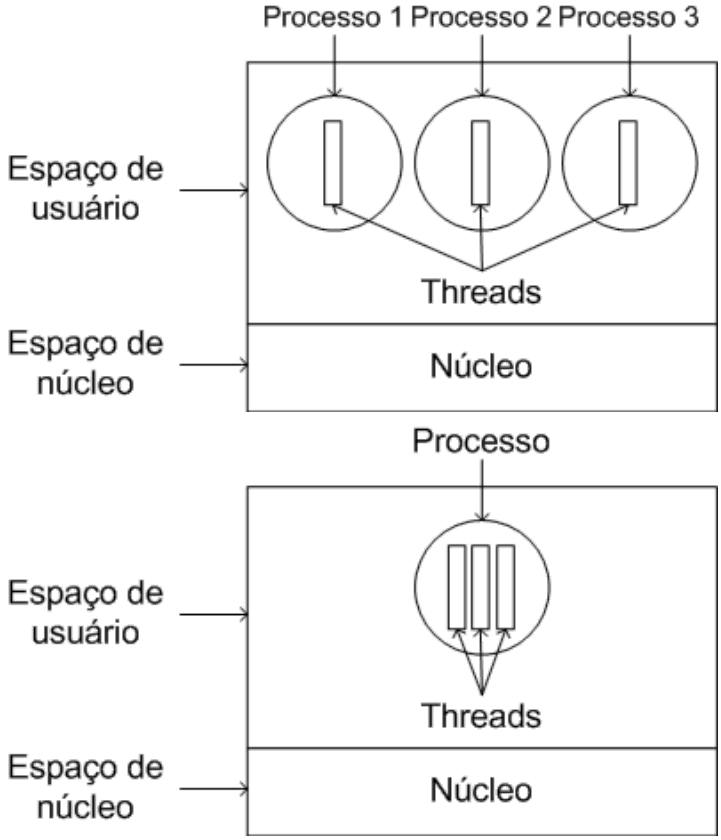
- Cenário comum
    - Um espaço de endereçamento
    - Único thread
- Possibilidade
    - Espaço de endereçamento compartilhado
    - Vários threads
        - Como processos separados

### O uso de threads

- Aplicações com múltiplas atividades em paralelo
  - Necessário compartilhar memória
- Rapidez na criação e destruição
  - 100 vezes mais rápido que processo
- Aceleram aplicação quando há muita E/S
  - Atividades se sobrepõem
- Paralelismo de processos
  - Sistemas com múltiplas CPUs
- Exemplo
  - Editor de texto com três threads
    1. Interage com usuário (interface gráfica)
    2. Reformata documento
    3. Salva periodicamente
  - Necessário compartilhamento de recurso
    - Processos não resolveriam

## O Modelo de thread clássico

- Diferença
  - Processo
    - Agrupamento de recursos
  - Threads ou processos leves
    - Múltiplas execuções em processo (multithread)




- Multithread funciona como multiprogramação
  - Chaveamento rápido
    - Impressão de execução paralela
  - Mesmas transições
    - Em execução, bloqueado e pronto
  - Compartilhamento de variáveis globais
    - Não é necessário proteção
    - Programador é responsável pela cooperação
- Itens de processo e de threads

| **Itens compartilhados por processos** |
|---|
| Espaço de endereçamento |
| Variáveis globais |
| Arquivos abertos |
| Processos filhos |

| |
|---|
| Alarmes pendentes |
| Sinais e manipuladores de sinais |
| Informação de contabilidade |

| **Itens específicos por thread** |
|---|
| Contador de programa |
| Registradores |
| Pilha |
| Estado |

- Execução de threads
  - Normalmente inicia com único thread
  - Chamadas a *thread_create* com indicação de rotina a executar
  - Normalmente, sem hierarquia
  - Thread encerra trabalho com *thread_exit*
  - Para esperar fim de outra, utiliza-se *thread_join*
  - Para desistir da rodada de execução, executa-se *thread_yield*

## Threads POSIX

- Padrão IEEE 1003.1c
- Pacote chamado Pthreads com 60 funções
- Exemplo

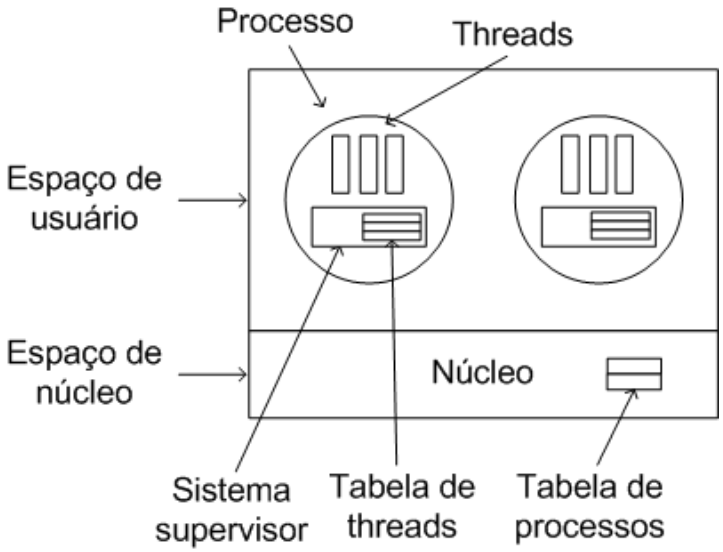
| **Chamada de thread** | **Descrição** |
|---|---|
| pthread_create | Cria um novo thread, retorna identificador do thread |
| pthread_exit | Conclui a chamada de thread |
| pthread_join | Espera que thread específico seja abandonado, recebe identificador do thread como parâmetro |
| pthread_yield | Libera CPU para que outro thread seja executado |
| pthread_attr_init | Cria, inicializa e retorna uma estrutura de atributos do thread com valores predefinidos |
| pthread_attr_destroy | Remover uma estrutura de atributos do thread |

- Código que cria threads

```
#include "pthread.h"
#include "stdio.h"
#include "stdlib.h"

#define QTD_DE_THREADS 10

void *imprime_alo_mundo(void *tid)
{   //Esta funcao imprime o identificador do thread e sai
    printf("Alo mundo. Saudacoes da thread %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{   //O programa principal cria 10 threads e sai
    pthread_t threads[QTD_DE_THREADS];
    int status, i;

    for (i = 0; i < QTD_DE_THREADS; i++)
    {   printf("Metodo principal. Criando thread %d\n", i);
        status = pthread_create(&threads[i], NULL, imprime_alo_mundo, (void *)i);
        if (status != 0)
        {   printf("Oops. pthread_create retornou codigo de erro %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

## Implementação de threads no espaço do usuário

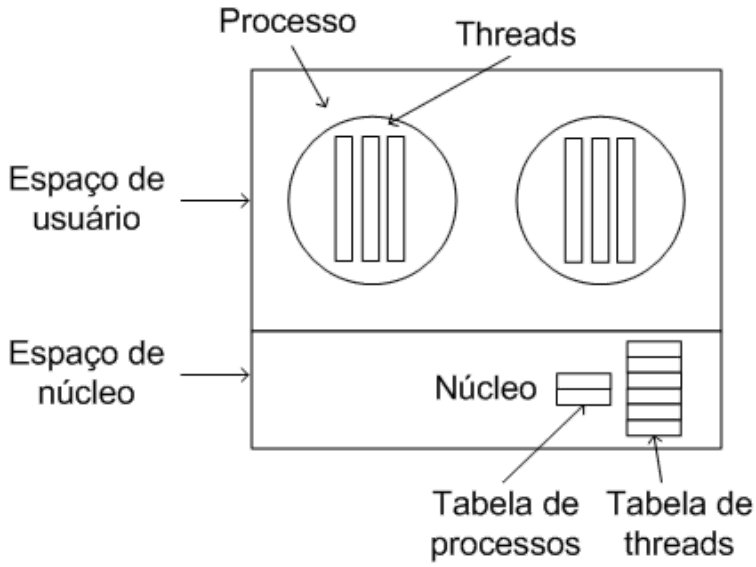
- Pacote de threads dentro do espaço do usuário
  - Sistema de tempo de execução (runtime) gerencia threads
  - Processo possui tabela de threads
    - Análoga a tabela de processos do núcleo




- Vantagens
  - Implementado em sistemas operacionais sem suporte a threads
  - Chaveamento entre threads é mais rápido do que entre processos
    - Não salva contexto do processo
    - Não executa TRAP
  - Permite algoritmos de escalonamento personalizado
- Desvantagens
  - Chamadas com bloqueio
    - Parariam as outras threads
    - Alterar sistema operacional não é possível
    - Solução ineficiente e deselegante
      - Substituir chamada (read) por verificação com chamada (select + read)
      - Verificação executa chamada se não bloquear
  - Bloqueio das threads for falta de página
    - Provocado por única thread
  - Escalonamento voluntário
    - Mecanismo de interrupções forçadas geraria sobrecarga

## Implementação de threads no núcleo

- Criação e destruição de threads via chamadas ao sistema operacional




- Vantagens
  - Não é necessário runtime
  - Não existe tabela de threads nos processos
  - Chamadas bloqueantes via chamada ao sistema operacional
    - Execução de thread do mesmo ou de outro processo
  - Permite reciclar threads
    - Não são destruídas
- Desvantagens
  - Em caso de fork, todas as threads devem ser copiadas?
    - Impossível determinar
  - Ocorrência de sinais
    - Enviados para o processo
      - Qual thread vai tratar?
    - Múltiplas threads interessadas no mesmo sinal

## Implementações híbridas

- Combinação das duas abordagens
  - Processo cria threads de núcleo
  - Threads de núcleo são multiplexadas em threads de usuário

## Ativações de escalonador

- Threads de núcleo são mais lentas
- Ativação de escalonador
  - Simula threads de núcleo em threads de usuário
  - Upcall
    - Núcleo avisa sistema de tempo de execução ocorrência de bloqueio de thread
    - Sistema de tempo de execução aloca outra thread
- Viola estrutura de camadas

## Threads pop-up

- Utilizadas em sistemas distribuídos
- Cada mensagem recebida provoca criação de thread para tratamento de mensagem
- Diferença
  - Com thread normal
    - Thread é criado
    - Bloqueado na ocorrência de `receive`
    - Contexto restaurado quando recebe mensagem
  - Com thread Pop-up
    - Criado rapidamente
    - Não é necessário restaurar registradores e pilha

## Convertendo o código monothread em código multithread

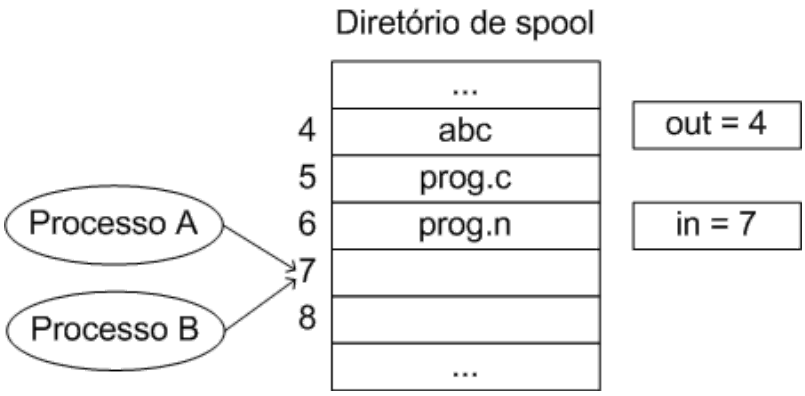
- Conversão não é trivial
- Variáveis locais e parâmetros
  - Não causam grandes problemas
- Variáveis globais
  - Fonte de erros
  - Exemplo
    - Thread 1 executa chamada ao sistema
    - Chamada gera erro que é gravado em errno (variável global do UNIX)
    - Thread 2 executa outra chamada ao sistema
    - Chamada sobre escreve valor em errno
    - Thread 1 lê valor inválido
- Soluções
  - Proibir variáveis globais
    - Impacto em programas existentes
  - Atribuir variáveis globais para cada thread
    - Falta de suporte em linguagens
    - Necessário criar rotinas especiais
      - create_global("bufptr")
      - set_global("bufptr", &buf)
      - bufptr = read_global("bufptr")

# Comunicação entre processos

- Frequentemente processos precisam se comunicar
  - De forma estruturada e sem interrupções
- Mecanismos válidos para processos e threads

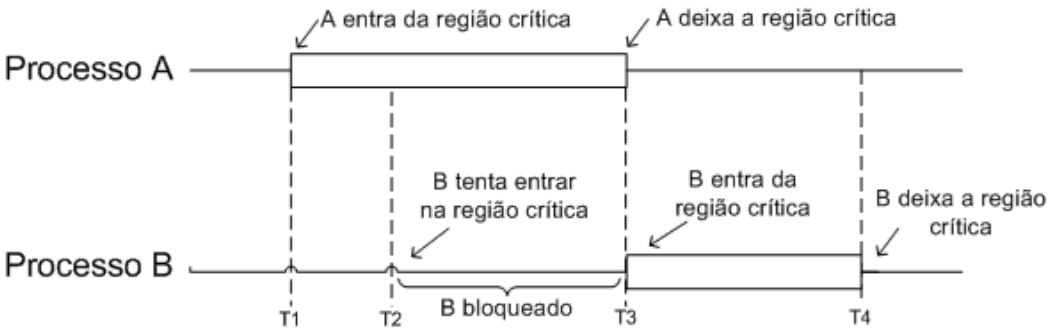
## Condições de corrida

- Condições de corrida ou race conditions
  - Mais de um processo compartilhando recurso
  - Resultado final depende de quem executa e quando
  - Difícil de testar devido a aleatoriedade
- Exemplo de fila de impressão
  - Diretório de spool com nomes de arquivos a imprimir
    - Numerados com 0, 1, 2, ...
  - Deamon de impressão imprime arquivos e remove do spool
    - Variável out aponta próximo a imprimir
    - Variável in indica próxima vaga na fila
  - Situação
    - Processo A lê in e armazena valor 7 na variável local proxima_vaga_livre
    - Sistema operacional retira processo A e coloca processo B para executar
    - Processo B lê in e armazena valor 7 na variável local proxima_vaga_livre
    - Processo B armazena nome do arquivo a ser impresso
    - Processo B atualiza in com valor 8 (proxima_vaga_livre + 1)
    - Sistema operacional retira processo B e coloca processo A para executar
    - Processo A armazena nome do arquivo a ser impresso, sobreescrevendo arquivo do processo B
    - Processo A atualiza in com valor 8 (proxima_vaga_livre + 1)
    - Deamon imprime arquivo do processo A
  - Problema
    - Arquivo do processo B nunca será impresso




## Regiões críticas

- Região crítica (critical region) ou seção crítica (critical section)
  - Parte do programa que faz acesso a memória compartilhada
  - Ocasionam disputas
- Com evitar condições de disputa?
  - Exclusão mútua (mutual exclusion)
    - Impedir acesso simultâneo a memória compartilhada
  - Condições a satisfazer
    1. Dois processo não podem estar na região crítica
    2. Sem considerações relacionadas a velocidade e a quantidade de CPUs
    3. Nenhum processo fora da região crítica pode bloquear outro
    4. Nenhum processo pode esperar eternamente para entrar na região crítica




## Exclusão mútua com espera ociosa

- Alternativas para exclusão mútua
  - Desabilitando interrupções
    - Processo desabilita todas as interrupções
      - Não ocorre mais chaveamento de processos
    - Se processo esquecer de habilitar?
      - Travamento do sistema
    - Se interrupções de única CPU forem desabilitadas?
      - Outras CPUs continuam acessando memória compartilhada
  - Variáveis do tipo trava (lock)
    - Utiliza variável compartilhada
      - Se variável = 0, entra em região crítica e escreve 1
      - Se variável = 1, aguarda variável receber 0
    - Isomórfico ao problema da fila de impressão
  - Chaveamento obrigatório
    - Utiliza teste de variável compartilhada
    - Valor associado a cada processo
    - Baseado em espera ociosa (busy waiting)
      - Não é eficiente
      - Processo 0

```
while (TRUE)
{       while(controle != 0); //laco
```

```
        regiao_critica();
        controle = 1;
        regiao_nao_critica();
}
```

- Processo 1

```
while (TRUE)
{       while(controle != 1); //laco
        regiao_critica();
        controle = 0;
        regiao_nao_critica();
}
```

- Violação da condição 3

| Tempo | Valor de controle | Processo 0 | Processo 1 | Comentário |
|---|---|---|---|---|
| 1 | controle = 0 | while(controle != 0); | | |
| 2 | controle = 0 | regiao_critica(); | | |
| 3 | controle = 0 | | while(controle != 1); | Espera ociosa |
| 4 | controle = 0 | controle = 1; | | |
| 5 | controle = 1 | | while(controle != 1); | Agora consegue entrar na região crítica |
| 6 | controle = 1 | | regiao_critica(); | |
| 7 | controle = 1; | | controle = 0; | |
| 8 | controle = 0; | regiao_nao_critica(); | | |
| 9 | controle = 0; | while(controle != 0); | | Processo 0 executa rapidamente o seu laço |
| 10 | controle = 0; | regiao_critica(); | | |
| 11 | controle = 0; | controle = 1; | | |
| 12 | controle = 1; | regiao_nao_critica(); | | |
| 13 | controle = 1; | while(controle != 0); | | |
| 14 | | | regiao_nao_critica(); | Processo 0 bloqueado por Processo 1 que não está na região crítica |

- Solução de Peterson
  - Utiliza duas rotinas e variáveis compartilhadas
  - Entrada em região crítica
    - Processo chama entrar_regiao_critica passando número como argumento
  - Saída de região crítica
    - Processo executa sair_regiao_critica passando número como argumento

```
#define FALSE 0
#define TRUE 1
#define N 2 //numero de processos
```

```
int controle; //de quem eh a vez?
int interessado[N]; //todos os valores 0 (FALSE)

void entrar_regiao_critica(int processo) //processo eh 0 ou 1
{       int outro = 1 - processo; //numero do outro processo
        interessado[processo] = TRUE; //mostra que processo esta interessado
        controle = processo; //alterar valor de controle
        while (controle == processo && interessado[outro] == TRUE); //aguarda vez
}

void sair_regiao_critica(int processo) //processo que esta saindo
{       interessado[processo] = FALSE; //indica saida de regiao critica
}
```

- Execução sequencial
  - Nenhum processo em região crítica
  - Processo 0 chama `entrar_regiao_critica`
  - Manifesta interesse em `interessado[0] = TRUE`
  - Modifica controle para 1
  - Como `interessado[1] == FALSE`, `entrar_regiao_critica` retorna imediatamente
  - Processo 1 chama `entrar_regiao_critica`
  - Como `interessado[0] == TRUE`, processo 1 espera
  - Processo 0 chama `sair_regiao_critica`, produzindo `interessado[0] = FALSE`
  - Processo 1 está liberado para continuar
- Execução concorrente
  - Dois processos chamam `entrar_regiao_critica`
  - Ambos armazenam valores em `controle`
  - Apenas o último que conta!
  - Se processo 1 foi o último, então `controle == 1`
  - Processo 0 retorna de `entrar_regiao_critica`
  - Processo 1 aguarda processo 0 chamar `sair_regiao_critica`

- A instrução TSL
  - Test and Set Lock
  - TSL RX, LOCK
    - Lê conteúdo na posição LOCK
    - Armazena em RX
    - Armazena valor diferente de zero na posição LOCK
  - Trava com auxílio de hardware
  - Impede acesso ao barramento de memória durante leitura
  - Rotinas em Assembler devem ser chamadas
    - Suporte no conjunto de instruções da CPU
  - Significado
    - Zero é trava aberta
    - Diferente de zero é trava fechada
  - Instrução XCHG é equivalente no conjunto x86

```
entrar_regiao_critica:
    TSL REGISTER, LOCK; copia lock para o registrador e poe lock em 1
    CMP REGISTER, #0; lock valia zero?
    JNE entrar_regiao_critica; se fosse diferente, lock estaria ligado, portanto continue r
    RET; retorna a quem chamou, entrou na regiao critica

sair_regiao_critida:
    MOV LOCK, #0; coloque 0 em lock
    RET;
```

## Dormir e acordar

- Estratégias baseadas em espera ociosa
  - Solução de Peterson
  - Instrução TSL
- Chamadas de sistema
  - Sleep adormece processo que chama

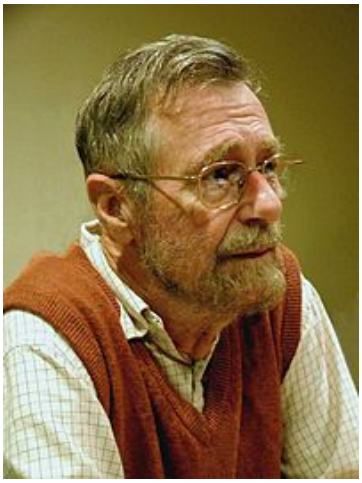
    - Wakeup acorda processo informado
- O problema do produtor-consumidor (ou buffer limitado)
    - Dois processos compartilham buffer
    - Produtor põe itens no buffer
    - Consumidor retira itens do buffer
    - Variáveis
        - Controle armazena quantidade de itens
        - N indica tamanho máximo do buffer
    - Problemas
        - Produtor quer colocar novos itens, mas buffer está cheio
        - Consumidor quer retirar itens, mas buffer está vazio
    - Solução

```
#define N 100 //numero de lugares no buffer
int contador = 0; //numero de itens no buffer

void produtor(void)
{       int item;
        while (TRUE) //repita para sempre
        {       item = produzir_item(); //gera proximo item
                if (contador == N) //se buffer estiver cheio
                {       sleep(); //processo vai dormir
                }
                inserir_item(item); //poe um item no buffer
                contador = contador + 1; //incremente o contador de itens no buffer
                if (contador == 1) //o buffer estava vazio?
                {       wakeup(consumidor); //acorde o consumidor
                }
        }
}

void consumidor(void)
{       int item;
        while (TRUE) //repita para sempre
        {       if (contador == 0) //se o buffer estiver vazio
                {   sleep(); //processo vai dormir
                }
                item = remover_item(); //retira item do buffer
                contador = contador - 1; //decresca de um o contador de itens no buffer
                if (contador == N - 1) //o buffer estava cheio?
                {       wakeup(produtor); //acorde o produtor
                }
                consumir_item(item);
        }
}
```

    - Situação de limitação do algoritmo
        1. Consumidor lê contador = 0
        2. Escalonador executa produtor
        3. Produtor insere item no buffer
        4. Produtor incrementa contador
        5. Produtor acorda consumidor
        6. Consumidor ainda não estava dormindo
        7. Sinal para acordar é perdido

## Semáforos

- Proposto por Dijkstra em 1965



- Semáforo indica quantos processos (threads) podem ter acesso a determinado recurso
- Duas operações atômicas
  - Acesso exclusivo ao semáforo
  - down (ou wait)
    - Se semáforo for igual a zero, bloqueia processo
    - Caso contrário, gasta um sinal de acordar

```
void down(semaforo s)
{   if (s == 0)
    {   bloqueia_processo();
    } else
    {   s--;
    }
}
```

  - up (ou signal)
    - Se semáforo for igual a zero e tem processo esperando, desbloqueia um processo
      - Um processo terá chance de avançar o semáforo
      - Variável `s` permanece em zero
    - Caso contrário, adiciona um sinal de acordar

```
void up(semaforo s)
{   if (s == 0 && existe_processo_bloqueado())
    {   desbloqueia_processo();
    } else
    {   s++;
    }
 }
```

- Aplicações com semáforos
  - Trava
    - Apenas um thread em região crítica
    - Caso específico chamado mutex
  - Contagem e trava
    - Quantidade limitada de threads executando
    - Threads excedentes aguardam vez de executar
  - Notificação
    - Thread aguarda notificação para prosseguir
    - Exemplo

```
semaforo a = 1;
semaforo b = 0;
//thread A
void ping()
{       while (TRUE)
        {       down(a);
                print("ping");
                up(b);
        }
```

```
}
//thread B
void pong()
{       while (TRUE)
        {       down(b);
                print("pong");
                up(a);
        }
}
```

- Resolvendo problema do produtor-consumidor utilizando semáforo
  - Não há perda de sinal
  - Chamadas ao sistema up e down
    - Desabilita interrupções
    - Coloca processos para dormir se necessário
  - Semáforo dura tempo curto
  - Produtor e consumidor pode ter tempo longo
  - Semáforos
    - Mutex
      - Garantir exclusão mútua
    - Cheio e vazio
      - Coloca produtor para dormir, se buffer está cheio
      - Coloca consumidor para dormir, se buffer está vazio

```
#define N 100 //número de lugares no buffer
typedef int semaforo //semáforos são um tipo especial de int
semaforo mutex = 1; //controla o acesso à região crítica
semaforo vazio = N; //conta os lugares vazios no buffer
semaforo cheio = 0; //conta os lugares preenchidos no buffer

void produtor(void)
{   int item;
    while (TRUE) //TRUE é a constante 1
    {   item = produzir_item(); //gera algo para pôr no buffer
        down(&vazio); //decresce o contador vazio
        down(&mutex); //entra na regia crítica
        inserir_item(item); //põe novo item no buffer
        up(&mutex); //sai da região crítica
        up(&cheio); //incrementa o contador de lugares preenchidos
    }
}

void consumidor(void)
{   int item;
    while (TRUE) //laço infinito
    {   down(&cheio); //decresce o contador cheio
        down(&mutex); //entra na região crítica
        item = remove_item();
        up(&mutex); //sai da região crítica
        up(&vazio); //incrementa o contador de lugares vazios
        consume_item(item); //faz algo com o item
    }
}
```

## Mutexes

- Mutual exclusion
- Versão simplificada de semáforo
  - Não utiliza contador
- Só possui dois estados
  - Desimpedido
  - Impedido
- Mutex permite acesso de thread a região crítica
  - Se mutex está desimpedido
    - Chamada prossegue e thread acessa região
  - Senão
    - Thread é bloqueada e aguarda outra thread chamar mutex_unlock
- Rotinas

```
mutex_lock:
    TSL REGISTER, MUTEX; copia mutex para registrador e atribui a ele o valor 1
    CMP REGISTER, #0; o mutex era zero?
    JZE ok; se era zero, o mutex estava desimpedido, portanto retorne
    CALL thread_yield; o mutex está ocupado; escalone outro thread
    JMP mutex_lock; tente novamente mais tarde
ok: RET; retorna a quem chamou; entrou na região crítica

mutex_unlock:
    MOVE MUTEX, #0; coloque 0 em mutex
    RET; retorna para quem chamou
```

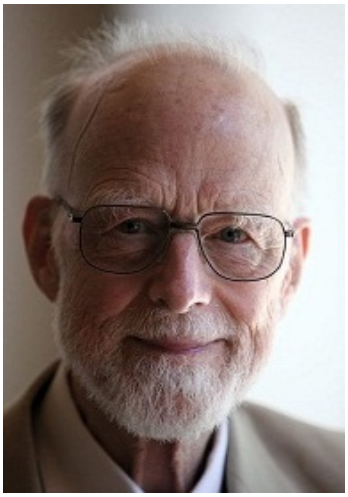
- Diferença em relação ao entrar_regiao_critica
  - Thread chama thread_yield ao invés de ficar constantemente testando
    - Sem espera ocupada
- Mutexes em pthreads

| Chamada de thread | Descrição |
|---|---|
| pthread_mutex_init | Cria um mutex |
| pthread_mutex_destroy | Destrói um mutex existente |
| pthread_mutex_lock | Conquista uma trava ou bloqueio |
| pthread_mutex_trylock | Conquista uma trava ou falha |
| pthread_mutex_unlock | Libera uma trava |

- Diferença entre mutex_lock e mutex_trylock
  - Flexibilidade na decisão de esperar
- Como processos vão ter acesso a variável comum?
  - Espaços de endereçamento disjuntos
  - Duas possibilidades
    - Estruturas de dados (semáforos) no núcleo
      - Acesso via chamadas ao sistema operacional
      - UNIX (sem_init e sem_open) e Windows (CreateSemaphore)
    - Compartilhamento de trechos de espaços de endereçamento
      - UNIX (shmget e shmat) e Windows (CreateFileMapping e OpenFileMapping)

## Monitores

- Proposto por Hoare e Hansen



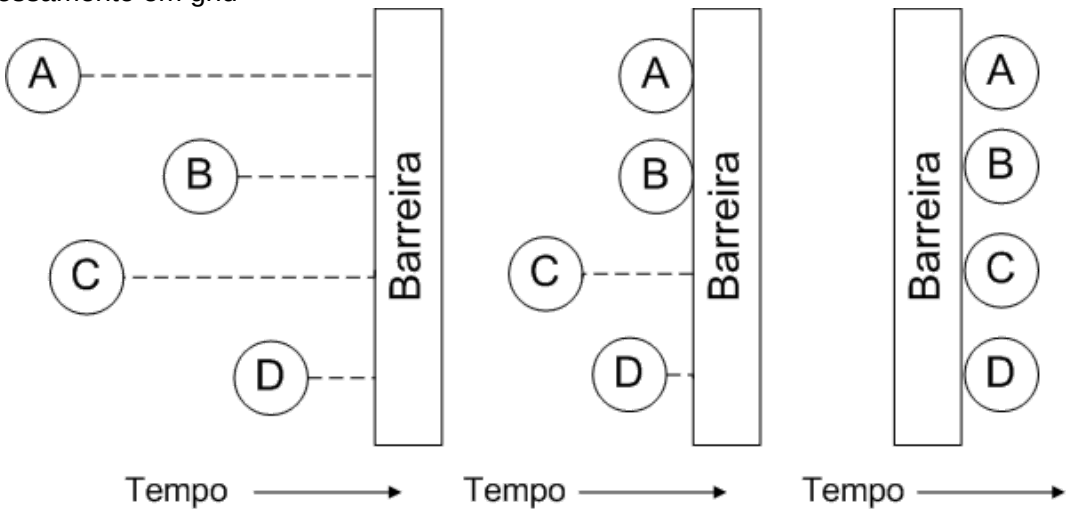

- Objetivo é simplificar o uso de semáforos
- Monitores
  - Composição
    - Iniciação
    - Dados privados
    - Rotinas
    - Fila de entrada
  - Possibilita que único processo esteja ativo em um monitor
- Construção dependente da linguagem de programação
  - Compilador adiciona instruções adicionais no começo
  - Instruções bloqueiam processo se outro já estiver ativo no monitor

- Monitores em Java são definidos com palavra `synchronized`
  - Em cabeçalhos de métodos
  - Em início de blocos

```
public class ContadorSincronizado
{       private int contador = 0;
        private Object trava = new Object();

        public synchronized void incrementar()
        {       contador++;
        }

        public synchronized void decrementar()
        {       contador--;
        }

        public int getValor()
        {       synchronized (trava)
                {       return contador;
                }
        }
}
```

### Barreiras

- Envolve grupos de processos
- Aplicações divididas em fases
  - Processo só avança para próxima quando os outros estiverem prontos
- Chamada de sistema barrier
- Exemplo
  - Processamento em grid




# Escalonamento

- Cenário
  - Computador multiprogramado
  - Processos e threads competem pela CPU
- Qual processo executar?
  - Escalonador decide

### Introdução ao escalonamento

- Comportamento do processo
  - Limitados pela CPU
    - CPU-bound
  - Limitados pela E/S
    - IO-bound
- CPUs mais rápidas

    - Processos tendem a ficar limitados por E/S
    - Processo IO-bound deve ter prioridade
        - Mantém o disco ocupado
- Quando escalonar?
    - Criação de processo
    - Término de processo
    - Processo bloqueia sobre semáforo
    - Ocorrência de interrupções
- Categorias de algoritmos
    - Não preemptivo
        - Para processo quando ocorre E/S ou encerramento do processo
    - Preemptivo
        - Executa processo por tempo fixo
- Inanição (Starvation) em ambiente multitarefa
    - Processos de baixa prioridade não executam devido aos de maiores prioridades
- Ambientes
    - Lote
        - Usuário não exige resposta rápida
        - Longos intervalos de tempo para cada processo
            - Redução de chaveamento
            - Melhor desempenho
        - Exemplo
            - Folha de pagamento
    - Interativo
        - Preempção é essencial
            - Evitar que processo se aposse da CPU
        - Exemplo
            - Programas com interfaces gráficas
    - Tempo real
        - Preempção pode ser desnecessária
            - Processos construidos para executar em períodos curtos
        - Exemplo
            - Visualizadores de vídeo

## Objetivos do algoritmo de escalonamento

- Todos os sistemas
    - Justiça
        - Dar porção justa de CPU para cada processo
    - Aplicação da política
        - Verificar a aplicação da política (prioridade de cada thread definida por cada processo) estabelecida
    - Equilíbrio
        - Manter todas as partes do sistema ocupadas
- Sistemas em lote
    - Vazão
        - Maximizar número de tarefas por horas
    - Tempo de retorno
        - Minimizar tempo entre submissão e término
    - Utilização de CPU
        - Manter CPU ocupada o tempo todo
            - Menor importância em relação as anteriores
- Sistemas interativos
    - Tempo de resposta
        - Responder rapidamente às requisições
    - Proporcionalidade
        - Satisfazer às espectativas dos usuários
- Sistemas de tempo real
    - Cumprimento de prazos
        - Evitar a perda de dados
    - Previsibilidade
        - Evitar degradação de qualidade em sistemas multimídia

## Escalonamento em sistemas em lote

- Primeiro a chegar, primeiro a ser servido
  - First come, first served (FCFS)
  - Fila única
    - Sem preempção
  - Simples de implementar
  - Ociosidade de partes do sistema
- Tarefa mais curta primeiro
  - Shortest Job First (SJF)
  - Requer conhecimento do tempo de execução
    - Tarefas rotineiras
  - Exemplo sem SJF

| Processo | Tempo individual | Tempo de retorno (sem SJF) |
|---|---|---|
| A | 8 minutos | 8 minutos |
| B | 4 minutos | 12 minutos |
| C | 4 minutos | 16 minutos |
| D | 4 minutos | 20 minutos |
| Média | | 14 minutos |

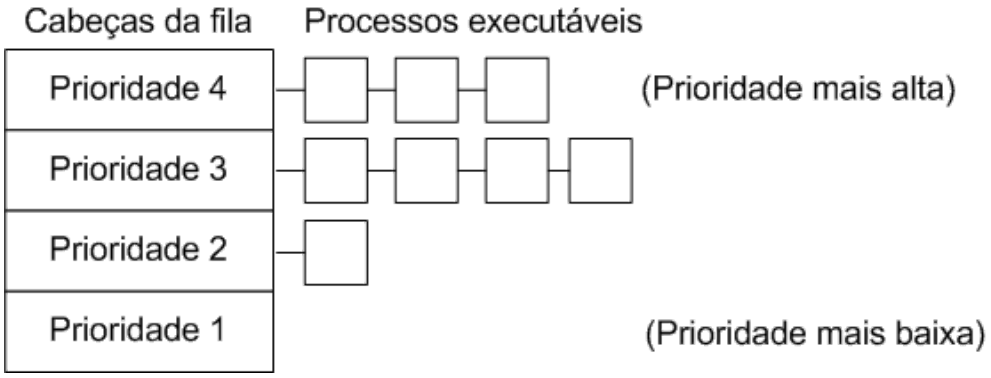
  - Exemplo com SJF

| Processo | Tempo individual | Tempo de retorno (com SJF) |
|---|---|---|
| B | 4 minutos | 4 minutos |
| C | 4 minutos | 8 minutos |
| D | 4 minutos | 12 minutos |
| A | 8 minutos | 20 minutos |
| Média | | 11 minutos |

- Próximo de menor tempo restante
  - Shortest Remaining Time Next (SRTN)
  - Requer conhecimento prévio do tempo
  - Escalonador compara
    - Tempo total de nova tarefa
    - Tempo restante de tarefa atual
  - Substitui processo para reduzir tempo médio de retorno

## Escalonamento em sistemas interativos

- Escalonamento por chaveamento circular (round-robin)
  - Mais antigo, simples e justo
  - Atribui intervalo de tempo (quantum)
  - No final do quantum
    - Se processo estiver executando
      - CPU sofre preempção
      - Próximo processo passa a executar
    - Senão
      - CPU é chaveada para outro processo
  - Tamanho do quantum
    - Curto
      - Provoca desperdício de tempo com chaveamento
    - Longo
      - Tempo médio de resposta pobre para requisições rápidas
- Escalonamento por prioridades
  - Prioridade atribuida a cada processo
    - Daemon que envia email
    - Exibição de um vídeo
  - Processos de alta prioridade não podem monopolizar CPU
    - Estratégias
      - Prioridade reduzida a cada execução de quantum
      - Estabelecer quantum máximo
  - Agrupamento de processos de mesma prioridade
    - Usar escalonamento circular entre eles




- Próximo processo mais curto
  - Esquema semelhante a processamento em lotes
  - Melhor tempo de resposta
  - Envelhecimento (Aging)
    - Estimativa de tempo de execução
    - $T_{estimado} = aT_0 + (1 - a)T_1$
      - Constante de envelhecimento "a"
- Escalonamento garantido
  - Sistema atribui 1/n de tempo da CPU
    - Por usuário em sistemas multiusuários
    - Por processo em sistemas monousuário
  - Calcula tempo atribuido e tempo realmente utilizado
    - Taxa = $t_{atribuido} / t_{utilizado}$
  - Escalonador executa processos com taxa menores
- Escalonamento por loteria
  - Distribuir bilhetes que permitem acesso a recursos
    - Processos mais importantes recebem mais bilhetes
  - Sortear bilhetes
    - Probabilidade de processos menos importantes é menor
- Escalonamento por fração justa (fair-share)
  - Cenário
    - Primeiro usuário lança nove processos
    - Segundo executa apenas um
    - 90% dos recursos destinados ao primeiro usuário
  - Divisão do tempo igualmente entre usuários

## Escalonamento em sistemas de tempo real

- Exemplos
  - Conversão de áudio a partir de CD
  - Monitoramento de pacientes em hospitais
  - Piloto automático de aeronaves
  - Robôs de automação industrial
- Tempo real
  - Crítico
  - Não crítico
- Eventos
  - Periódicos
  - Aperiódicos
- Sistema escalonável
  - $\sum C_i / P_i \leq 1$
  - Em que
    - $C_i$ é o tempo de execução do processo i
    - $P_i$ é o período de repetição do processo i
  - Exemplo
    - Tempos e períodos

| Processo | C(ms) | P(ms) | C/P |
|---|---|---|---|
| 1 | 100 | 50 | 0,50 |
| 2 | 200 | 30 | 0,15 |
| 3 | 500 | 100 | 0,20 |

        - $\sum C_i / P_i \leq 1$?
            - $0,50 + 0,15 + 0,20 \leq 1$
            - $0,85 \leq 1$ (escalonável)

### Política *versus* mecanismo

- Processo pode possuir diversos processos filhos
- Escalonadores não consideram importância dos filhos
- Algoritmo de escalonamento pode ser parametrizado
- Processo pai configura prioridades dos filhos
- Separação da política (processos) do mecanismo (núcleo)

### Escalonamento de threads

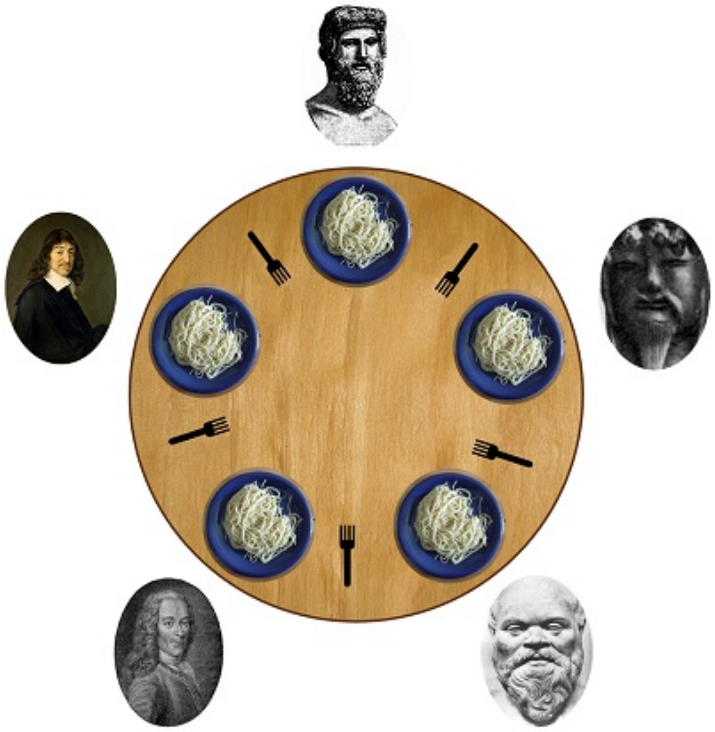
- Dois níveis de paralelismo
    - Processos
    - Threads
- Threads de usuário
    - Núcleo não tem conhecimento das threads
    - Chaveamento entre threads através do runtime
    - Algoritmos comuns
        - Escalonamento circular
        - Escalonamento por prioridade
- Threads de núcleo
    - Núcleo escolhe quem deve executar
    - Chaveamento mais lento nas threads de núcleo
        - Requer chamadas ao sistema
        - Requer trocas de contexto
    - Escalonador dá preferência para threads do processo já em execução

## Problemas clássicos de IPC (Comunicação entre processos)

- Amplamente discutidos
- Testes de métodos de sincronização

### O problema do jantar dos filósofos

- Proposto por Dijkstra em 1965
- Definição
    - Cinco filósofos em mesa circular
    - Cada filósofo com um prato de espaguete
    - Cada filósofo precisa de dois garfos
    - Entre cada prato existe um garfo
    - Filósofo pode comer ou pensar
    - Se filósofo conseguir pegar dois garfos, ele come
    - Caso contrário, devolve garfo e vai pensar



- Objetivo
  - Cada filósofo deve comer e não devem travar
- Possíveis soluções
  - Solução óbvia
  - Solução com devolução de garfo
  - Solução com espera aleatória
  - Solução com semáforo
  - Solução com semáforos
- Solução óbvia

```
#define N 5 //o numero de filosofos
void philosopher(int i) //i: o numero do filosofo, de 0  a 4
{       while (TRUE)
        {       think(); //filosofo esta pensando
                take_fork(i); //pega o garfo esquerdo
                take_fork((i + 1) % N); //pega o garfo direito
                eat(); //come o espaguete
                put_fork(i); //devolve o garfo esquerdo
                put_fork((i + 1) % N); //devolve o garfo direito a mesa
        }
}
```

  - Se todos pegarem os garfos à esquerda, ninguém vai conseguir pegar o garfo da direita
    - Algoritmo começar pelo garfo da direita não resolve problema
  - Impasse (Deadlock)
    - Processo necessita de recurso detido por outros
    - Processo detém recurso necessitado por outros

- Solução com devolução de garfo
  - Pegar garfo direito, se esquerdo não estiver disponível, devolver direito e aguardar
  - Se todos pegarem os garfos à esquerda, todos vão devolver e depois repetir
    - Algoritmo começar pelo garfo da direita não resolve problema
  - Inanição (Starvation) em ambiente concorrente
    - Processo espera indefinidamente por recurso
    - Sem ocorrência de bloqueio
- Solução com espera aleatória
  - Aguardar tempo aleatório, se garfo não estiver disponível
  - Diminuição da probabilidade de travamento
    - Uso no protocolo Ethernet
  - Possibilidade de travamento não pode ser eliminada
- Utilizar semáforo
  - Proteger cinco comandos após `think` com semáforo binário
  - Limitado do ponto de vista prático
    - Único filósofo poderia comer por vez
- Utilizar semáforos

```
#define N 5 //o numero de filosofos
#define LEFT (i+N-1)%N //o número do vizinho a esquerda de i
#define RIGHT (i+1)%N //o número do vizinho a direita de i
#define THINKING 0 //o filósofo esta pensando
#define HUNGRY 1 //o filósofo esta tentanto pegar garfos
#define EATING 2 //o filósofo esta comendo
typedef int semaphore; //semáforos sao tipos especiais de int
int state[N]; //arranjo para controlar o estado de cada um
semaphore mutex = 1; //exclusao mutua para regiões críticas
semaphore s[N]; //um semáforo por filósofo

void philosopher(int i) //i: o número do filósofo, de 0 a N-1
{       while (TRUE) //repeta para sempre
        {       think(); //o filósofo está pensando
                take_forks(i); //pega dois garfos ou bloqueia
                eat(); //hummm! espaguete!
                put_forks(i); //devolve os dois garfos à mesa
        }
}

void take_forks(int i) //i: o número do filósofo, de 0 a N-1
{       down(&mutex); //entra na região crítica
        state[i] = HUNGRY; //registra que filósofo está faminto
        test(i); //tenta pegar dois garfos
        up(&mutex); //saí da região crítica
        down(&s[i]); //bloqueia se os garfos não foram pegos
}

void puts_forks(int i) //i: o número do filósofo, de 0 a N-1
```

```
{       down(&mutex); //entra na região crítica
        state[i] = THINKING; //registra que filósofo está faminto
        test(LEFT); //vê se vizinho da esquerda pode comer agora
        test(RIGHT); //vê se vizinho da direita pode comer agora
        up(&mutex); //saí da região crítica
}

void test(int i) //i: o número do filósofo, de 0 a N-1
{       if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING)
        {       state[i] = EATING;
                up(&s[i]);
        }
}
```

- Estado indica se está pensando, faminto (tentando pegar garfos) ou comendo
- Filósofo só pode comer se vizinhos não estiverem comendo
  - Vizinhos definidos pelas macros LEFT e RIGHT
- Código utiliza array de semáforos
  - Só bloqueia filósofos com garfos ocupados
- Cada filósofo executa a rotina `philosopher`
  - Informando seu número

# Referências bibliográficas


- TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 3 ed. cap. 2 (Processos e threads).