Sistemas Operacionais

Prof. Rafael Obelheiro
rafael.obelheiro@udesc.br

Processos e Threads

Sumário

Conceito

- Um **processo** é um **programa em execução**

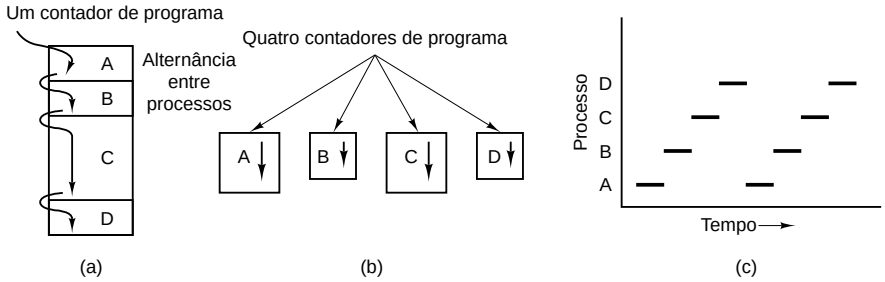
  código + conteúdo das variáveis + ponto de execução

  ⇓

  registradores, contador de programa, pilha

  - cada processo enxerga uma CPU virtual
- **Multiprogramação**: vários processos carregados na memória ao mesmo tempo
  - máquinas monoprocessadas: apenas um processo executa de cada vez
    - ⋆ **pseudoparalelismo**
  - máquinas multiprocessadas: paralelismo real

Multiprogramação de quatro processos



- Processos não devem fazer hipóteses temporais ou sobre a ordem de execução
  - primitivas de sincronização

Criação de processos

Principais eventos que levam à criação de processos:

1. Início do sistema
2. Execução de chamada ao sistema de criação de processos
   - `fork` (Unix), `CreateProcess` (Windows), `SYS$CREPRC` (VAX/VMS)
3. Solicitação do usuário para criar um novo processo
4. Início de um job em lote

Tipos de processos

- Processos interativos: interagem com usuários
  - primeiro plano (*foreground*)
- Processos de segundo plano (*background*): serviços do sistema
  - *daemons*

Exemplo: chamada `fork`

- No Unix, processos são criados através da chamada `fork`
- O processo filho é idêntico ao processo pai:
  - código e dados são copiados
  - diferença está no valor de retorno da função `fork()`
    - ★ no processo pai, a função retorna o identificador (PID) do filho
    - ★ no processo filho, a função retorna 0
  - a chamada `exec` pode ser usada para substituir o processo corrente

```
f = fork();
if (f == 0)  {                    /* processo filho */
   printf("processo filho\n");
   exit(4);                            /* retorna 4 */
} else {                           /* processo pai */
   printf("processo pai\n")
   w = waitpid(f, &rc, 0);      /* espera retorno   */
}                               /* do filho (rc==4) */
```


Término de processos

- Condições para o término de um processo:
  - saída normal (voluntária)
  - saída por erro (voluntária)
    - ★ programa detecta um erro
  - erro fatal (involuntário)
    - ★ programa faz algo ilegal
  - cancelamento por outro processo (involuntário)
- O término de um processo pode causar o término dos processos que ele criou
  - não ocorre nem em Unix nem em Windows

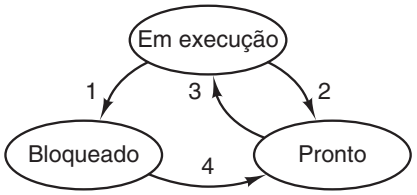
Hierarquias de processos

- Processos “procriam” por várias gerações
  - um processo pai cria processos filhos, que por sua vez também criam seus filhos, *ad nauseam*
- Leva à formação de **hierarquias** de processos
- Chamadas “grupos de processos” no Unix
  - sinalizações de eventos se propagam através do grupo, e cada processo decide o que fazer com o sinal (ignorar, tratar ou “ser morto”)
  - todos os processos Unix descendem de *init*
    - ★ *systemd* em várias distribuições Linux
- Windows não possui hierarquias de processos
  - todos os processos são criados iguais

Estados de um processo

- Um processo pode assumir diversos estados no sistema
  - **em execução**: processo que está usando a CPU
  - **pronto**: processo temporariamente parado enquanto outro processo executa
    - ★ fila de prontos (aptos)
  - **bloqueado**: esperando por um evento externo

Transições de estado de um processo



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

- Processos entram no sistema na fila de prontos
- Transições dependem de interrupções para sinalizar condições
  - término de operações de E/S, passagem do tempo, ...

Implementação de processos

- As informações sobre os processos do sistema são armazenadas na **tabela de processos**
  - cada entrada é chamada de **descritor de processo** ou **bloco de controle de processo**

Gerenciamento de processos	Gerenciamento de memória	Gerenciamento de arquivos
Registradores	Ponteiro para o segmento de código	Diretório-raiz
Contador de programa	Ponteiro para o segmento de dados	Diretório de trabalho
Palavra de estado do programa	Ponteiro para o segmento de pilha	Descritores de arquivos
Ponteiro de pilha		Identificador (ID) do usuário
Estado do processo		Identificador (ID) do grupo
Prioridade		
Parâmetros de escalonamento		
Identificador (ID) do processo		
Processo pai		
Grupo do processo		
Sinais		
Momento em que o processo iniciou		
Tempo usado da CPU		
Tempo de CPU do filho		
Momento do próximo alarme		

O papel das interrupções

- Interrupções são fundamentais para multiprogramação
  - sinalizam eventos no sistema
  - dão oportunidade para que o SO assuma o controle e decida o que fazer
- **Processos não executam sob o controle direto do SO**
  - o SO só assume quando ocorrem interrupções ou chamadas de sistema (implementadas com *traps*)

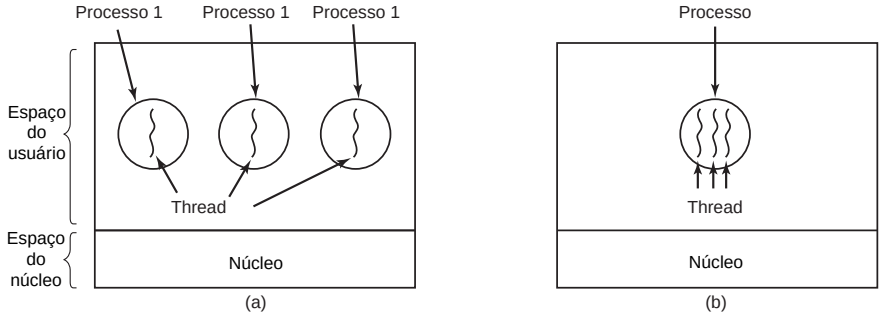
Sumário

O modelo de thread (1/2)

- Processos possuem
  - um espaço de endereçamento
  - uma thread de execução ou fluxo de controle
- Processos agrupam recursos
  - espaço de endereçamento (código+dados), arquivos, processos filhos, alarmes pendentes, . . .
  - esse agrupamento facilita o gerenciamento
- A thread representa o estado atual de execução
  - contador de programa, registradores, pilha
- A unificação é uma conveniência, não um requisito

O modelo de thread (2/2)

- Múltiplas threads em um processo permitem execuções paralelas sobre os mesmos recursos
  - análogo a vários processos em paralelo
- Processos leves ou multithread



(a) 3 processos com uma thread

(b) Um processo com 3 threads

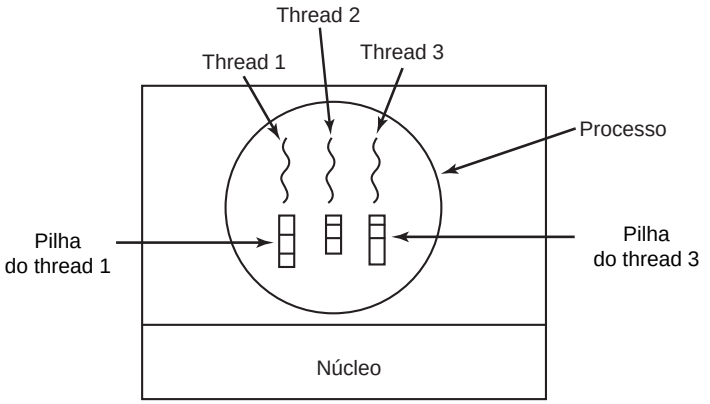
Compartilhamento de recursos (1/2)

- As várias threads de um processo compartilham muitos dos recursos do processo
  - **não existe proteção entre threads**

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

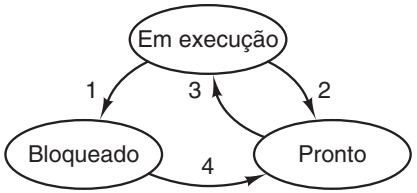
Compartilhamento de recursos (2/2)

- Cada thread precisa da sua própria pilha
  - mantém suas variáveis locais e histórico de execução



Estados de uma thread

- Uma thread pode ter os mesmos estados de um processo
  - em execução, pronto, bloqueado



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

- Dependendo da implementação, o bloqueio de uma das threads de um processo pode bloquear todas as demais

Vantagens de threads

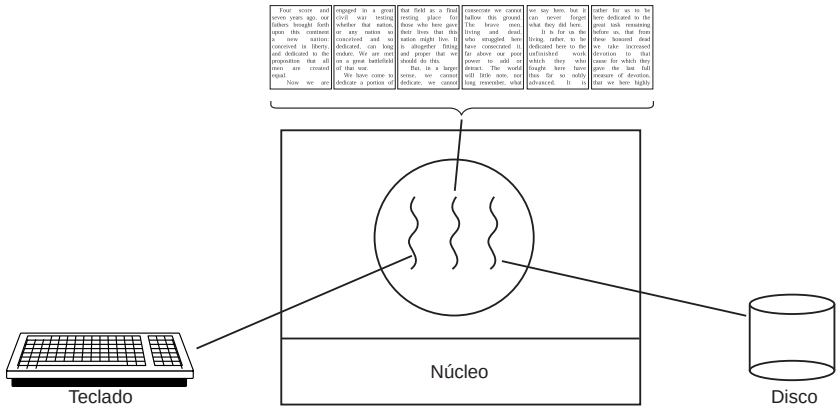
- Possibilitar soluções paralelas para problemas
  - cada thread sequencial se preocupa com uma parte do problema
  - interessante em aplicações dirigidas a eventos
- Desempenho
  - criar e destruir threads é mais rápido
  - o chaveamento de contexto é muito mais rápido
  - permite combinar threads I/O-bound e CPU-bound

Problemas com threads

- Complicações no modelo de programação
  - um processo filho herda todas as threads do processo pai?
  - se herdar, o que acontece quando a thread do pai bloqueia por uma entrada de teclado?
- Complicações pelos recursos compartilhados
  - e se uma thread fecha um arquivo que está sendo usado por outra?
  - e se uma thread começa uma alocação de memória e é substituída por outra?

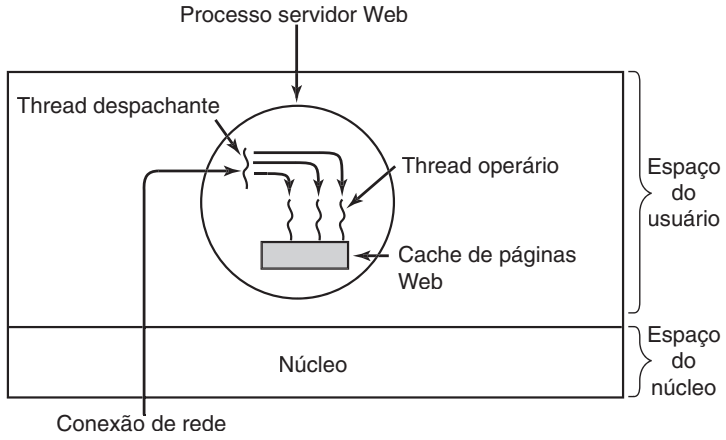
Exemplos de uso de threads (1/3)

- Processador de texto com 3 threads
  - considere a implementação monothread



Exemplos de uso de threads (2/3)

- Servidor web multithreaded



Exemplos de uso de threads (3/3)

- Código simplificado do servidor web

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

(a) despachante

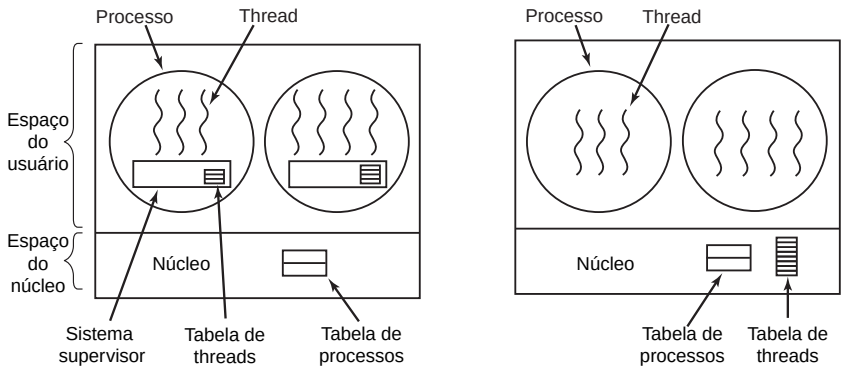
```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if(page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

(b) operário

Implementação de threads

- Existem dois modos principais de se implementar threads
  (a) threads no espaço do usuário (N:1)
  (b) threads no espaço do núcleo (1:1)



- Implementações híbridas também são possíveis

Threads de usuário

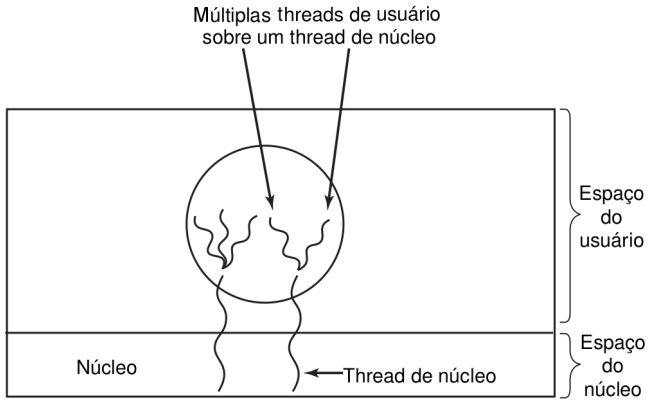
- As threads são implementadas por uma biblioteca, e o núcleo não sabe nada sobre elas
  - N threads são mapeadas em um processo (N:1)
  - núcleo escalona processos, não threads
  - o escalonamento de threads é feito pela biblioteca
- Vantagens
  - permite usar threads em SOs que não têm suporte
  - chaveamento de contexto entre threads não requer chamada de sistema → desempenho
- Desvantagens
  - tratamento de chamadas bloqueantes
  - preempção por tempo é complicada

Threads de núcleo

- O núcleo conhece e escalona as threads
  - não há necessidade de biblioteca
  - modelo 1:1
- Vantagens
  - facilidade para lidar com chamadas bloqueantes
  - preempção entre threads
- Desvantagens
  - operações envolvendo threads têm custo maior
    - ★ exigem chamadas ao núcleo

Threads híbridas

- Combina os dois modelos anteriores



Convertendo código para multithreading

- Problemas em potencial
  - variáveis globais modificadas por várias threads
    - ★ proibir o uso de variáveis globais
    - ★ permitir variáveis globais privativas de cada thread
  - bibliotecas não reentrantes ou não thread-safe: funções que não podem ser executadas por mais de uma thread
    - ★ permitir apenas uma execução por vez
    - ★ mudar para versão não reentrante e thread-safe
      ⇒ ex: trocar `random()` por `random_r()`
  - sinais
    - ★ quem captura? como tratar?
  - gerenciamento da pilha
    - ★ o sistema precisa tratar o overflow de várias pilhas

Sumário

Introdução a Pthreads

- O padrão IEEE POSIX 1003.1c define uma API para programação usando threads
  - POSIX threads ⇒ Pthreads
- Implementações disponíveis para diversas variantes de UNIX e Windows
  - nível de usuário ou nível de núcleo
- Windows: Cygwin, MinGW

Programando com Pthreads (Linux/Cygwin)

- Para usar as funções da biblioteca Pthreads, deve-se incluir o cabeçalho `pthread.h`

```
#include <pthread.h>
```

- Para compilar um programa com Pthreads, deve-se passar a opção `-pthread` para o `gcc`

```
$ gcc -Wall -pthread -o prog prog.c
```

Criando threads

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

- `pthread_t *thread`: identificador (ID) da thread, passado por referência
- `pthread_attr_t *attr`: atributos da thread
  - `NULL` para atributos default
  - manipulados via funções `pthread_attr_`*nnnn*
- `void *(*start_routine)`: ponteiro para a função onde inicia a thread
  - função possui um único parâmetro, `void *`
  - valor de retorno da função também é `void *`
- `void *arg`: argumento para `start_routine`
  - `NULL` se não há argumentos
- Retorna 0 para sucesso, outro valor em caso de erro

Encerrando threads

- A execução da thread encerra quando:
  - ela retorna de `start_routine()`
  - ela invoca `pthread_exit()`
    - ★ permite retornar um código de status
  - ela é cancelada por outra thread com `pthread_cancel()`
  - o processo inteiro encerra com `exit()` ou `exec()`

Um exemplo simples (`simples.c`)

```
#include <pthread.h>
#include <stdio.h>

#define NUM_THREADS     5

void *PrintHello(void *arg) {
   long tid = (long)arg;
   printf("Alo da thread %ld\n",
          tid);
   pthread_exit(NULL);
}
```

```
int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for (t=0; t<NUM_THREADS; t++){
      printf("main: criando thread %ld\n", t);
      rc = pthread_create(&threads[t],
                          NULL,
                          PrintHello,
                          (void *)t);
      if (rc) {
         printf("ERRO - rc=%d\n", rc);
         exit(-1);
      }
   }
   /* Ultima coisa que main() deve fazer */
   pthread_exit(NULL);
}
```


Passando parâmetros para a thread

- A função onde a thread inicia só aceita um parâmetro `void *`
- Parâmetros de outros tipos requerem casting
  - vide exemplo anterior
- A conversão segura para 32 e 64 bits é ponteiro ↔ `long`
- Para passar múltiplos parâmetros, pode ser usada uma `struct`

Esperando a conclusão de uma thread

- Por padrão, uma thread é criada como *joinable*
  - a thread que a criou pode esperar que ela termine e recuperar o status retornado
- `int pthread_join(pthread_t thread, void **value_ptr)`
  - `pthread_t thread`: ID da thread a esperar
  - `void **value_ptr`: endereço da variável onde é armazenado o valor de retorno
    - ★ especificado em `pthread_exit()`
  - retorna 0 para sucesso, outro valor em caso de erro
  - **thread chamadora fica bloqueada**

Obtendo o valor de retorno de uma thread

```
void *funcThread(void *arg) {
   long ret;
   ...
   pthread_exit((void *) ret);
}
int main(int argv, char *argv[]) {
   void *status;
   long ret;

   pthread_create(&thr, NULL, funcThread, NULL);
   ...
   pthread_join(thr, &status);
   ret = (long) status;
   ...
}
```

- A thread coloca o valor de retorno em `pthread_exit()`, e o valor é recuperado com `pthread_join()`

Obtendo e comparando IDs de thread

- Cada thread tem um ID único
  - IDs devem ser considerados objetos opacos → o programa não deve assumir nada sobre sua implementação
- `pthread_t pthread_self(void)`
  - retorna o ID da thread corrente
- `int pthread_equal(pthread_t t1, pthread_t t2)`
  - compara os IDs `t1` e `t2`
  - retorna 0 se `t1` ≠ `t2`, outro valor se `t1` = `t2`

Sumário

Conceitos de comunicação interprocessos

- Processos e threads que estão executando em paralelo podem
  1. se comunicar
  2. acessar dados compartilhados
     - ★ por definição, processos executam em espaços de endereçamento distintos (memória privada) e threads executam no mesmo espaço de endereçamento (memória compartilhada)
- Os mecanismos de **comunicação interprocessos** de um SO são usados para implementar essas funcionalidades e auxiliar no seu gerenciamento
  - mecanismos de comunicação propriamente ditos
    - ★ pipes, filas de mensagens, sockets, memória compartilhada
  - mecanismos de coordenação entre processos e threads
    - ★ como evitar problemas de concorrência
    - ★ como determinar a sequência de execução de processos/threads
- Consideraremos processos, mas valem igualmente para threads

O problema da concorrência

- Considere que uma aplicação bancária tem um código equivalente ao seguinte:

```
void depositar(long *saldo, long valor) {
   (*saldo) += valor;
}
```

- Esse código será compilado para algo como

```
1: r1 <- *saldo       ; carrega *saldo em r1
2: r2 <- valor        ; carrega valor em r2
3: r1 <- r1 + r2
4: *saldo <- r1       ; carrega r1 em *saldo
```

  onde `r1` e `r2` são registradores
- A função `depositar()` executa corretamente quando apenas um processo manipula as contas, mas o que pode acontecer se houver mais de um processo trabalhando sobre os mesmos dados?

Uma execução correta

- O saldo inicial da conta 171 é zero
- Dois depósitos são efetuados quase ao mesmo tempo, um de R$ 50 e outro de R$ 1000

```
/* cta[171] == 0 */
depositar(&cta[171], 50)          depositar(&cta[171], 1000)
1: r1 <- *saldo  ; r1 <- 0
2: r2 <- valor   ; r2 <- 50
3: r1 <- r1 + r2 ; r1 <- 50
4: *saldo <- r1  ; cta[171] <- 50
                                  1: r1 <- *saldo  ; r1 <- 50
                                  2: r2 <- valor   ; r2 <- 1000
                                  3: r1 <- r1 + r2 ; r1 <- 1050
                                  4: *saldo <- r1  ; cta[171] <- 1050
```

- O saldo final da conta é R$ 1050 (correto)

Uma execução problemática

- O saldo inicial da conta 171 é zero
- Dois depósitos são efetuados quase ao mesmo tempo, um de R$ 50 e outro de R$ 1000

```
/* cta[171] == 0 */
depositar(&cta[171], 50)          depositar(&cta[171], 1000)
1: r1 <- *saldo  ; r1 <- 0
2: r2 <- valor   ; r2 <- 50
3: r1 <- r1 + r2 ; r1 <- 50
                                  1: r1 <- *saldo  ; r1 <- 0
                                  2: r2 <- valor   ; r2 <- 1000
                                  3: r1 <- r1 + r2 ; r1 <- 1000
                                  4: *saldo <- r1  ; cta[171] <- 1000
4: *saldo <- r1  ; cta[171] <- 50
```

- O saldo final da conta é R$ 50
  - **o depósito de R$ 1000 foi perdido**

Condições de disputa

- Quando dois ou mais processos manipulam dados compartilhados simultaneamente e o resultado depende da ordem precisa em que os processos são executados
  - **erros dinâmicos**: podem ocorrer ou não, de forma não determinística
- Também chamadas de condições de corrida
- No exemplo, os dados compartilhados são representados pela base de contas (variável `cta`)

Condições de Bernstein

- Em 1966, Bernstein formalizou um conjunto de condições que devem ser respeitadas para evitar condições de disputa
- Notação: para um processo p_i
  - $\mathcal{R}(p_i)$: conjunto de variáveis lidas por p_i
  - $\mathcal{W}(p_i)$: conjunto de variáveis escritas por p_i
- Dois processos p_1 e p_2 podem executar em paralelo sem risco de condição de disputa ($p_1 \| p_2$) se e somente se:

$$p_1 \| p_2 \iff \begin{cases} \mathcal{R}(p_1) \cap \mathcal{W}(p_2) = \varnothing \\ \mathcal{R}(p_2) \cap \mathcal{W}(p_1) = \varnothing \\ \mathcal{W}(p_1) \cap \mathcal{W}(p_2) = \varnothing \end{cases}$$

- **Condições de disputa só existem quando houver escritas concorrentes**

Regiões críticas

- Partes do código em que há acesso a memória compartilhada e que pode levar a condições de disputa
  - também chamadas de seções críticas
  - podem ser identificadas usando as condições de Bernstein
- Na função `depositar()`, a região crítica é a linha

```
(*saldo) += valor;
```

- Um programa pode ter várias seções críticas, relacionadas entre si ou não
  - depende dos dados compartilhados que são manipulados

Exclusão mútua

- É necessário haver **exclusão mútua** entre os processos durante suas regiões críticas
- A ideia básica é introduzir um **protocolo de acesso** para a região crítica
  - também chamado de **guardas**

região crítica $\Longrightarrow$

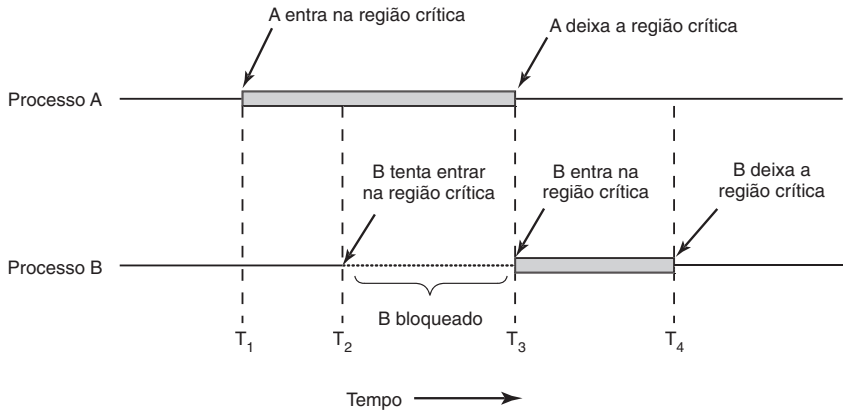
```
enter(RC_i)
  região crítica
leave(RC_i)
```

- Uma solução para o problema de exclusão mútua é um par de algoritmos ou primitivas que implementam essas guardas
  - funções `enter()` e `leave()`

Condições para exclusão mútua

- Quatro condições necessárias para prover exclusão mútua:
  1. Nunca dois processos podem estar simultaneamente em uma região crítica
  2. Nenhuma afirmação sobre velocidades ou número de CPUs
  3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos
  4. Nenhum processo deve esperar eternamente para entrar em sua região crítica

Exclusão mútua em regiões críticas



Exclusão mútua com espera ocupada

- Existem diversas soluções para o problema de exclusão mútua
- Algumas delas se baseiam em espera ocupada (ociosa)
  - o processo fica em loop até conseguir entrar na seção crítica
- Exemplos
  - desabilitação de interrupções
  - variáveis de impedimento (lock)
  - alternância obrigatória
  - solução de Peterson
  - instrução TSL

Desabilitação de interrupções

- Se as interrupções forem desabilitadas o processo não perde a CPU
  - transições de estado ocorrem por interrupções de tempo ou E/S
- Poder demais para processos de usuário
  - podem deixar de habilitar as interrupções (de propósito ou não)
- Não funciona em multiprocessadores
- Muito usada no núcleo do SO para seções críticas curtas
  - exemplo: atualização de listas encadeadas

Variáveis de impedimento (lock)

- Uma variável lógica que indica se a seção crítica está ocupada

```
1: while (lock == 1)
2:     ;     /* loop vazio */
3: lock = 1;
4: /* seção crítica */
5: lock = 0;
6: /* seção não crítica */
```

- Solução sujeita a condições de disputa

Condição de disputa envolvendo lock

```
1: while (lock == 1)
2:     ;     /* loop vazio */
3: lock = 1;
4: /* seção crítica */
5: lock = 0;
6: /* seção não crítica */
```

instante	proc 1	proc 2	`lock`
t_1	1		0
t_2		1	0
t_3		3	$\not{0} \rightarrow 1$
t_4		4	1
t_5	3		$\not{1} \rightarrow 1$
t_6	4		1

- No instante t_6 os dois processos estão executando na região crítica ao mesmo tempo
- O problema é que um processo pode perder a CPU entre o teste do valor de `lock` (linha 1) e a atualização da variável (linha 3)

Alternância obrigatória (1/2)

- Cada processo tem a sua vez de entrar na seção crítica
  - variável `turn`
- Ainda é espera ocupada
  - desperdício de CPU
- Não funciona bem se um dos processos é muito mais lento do que o outro
  - viola a condição 3

Alternância obrigatória (2/2)

```
while (TRUE) {
    while (turn !=0)          /* laço */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

(a) código para o processo 0

```
while (TRUE) {
    while (turn !=1)          /* laço */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

(b) código para o processo 1

Solução de Peterson (1/2)

- Combina variáveis de lock e alternância obrigatória
- Funcionamento
  - antes de usar as variáveis compartilhadas, o processo *i* chama `enter_region(i)`
  - depois que terminou de usar as variáveis compartilhadas, o processo *i* chama `leave_region(i)`

Solução de Peterson (2/2)

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

int turn;                           /* de quem é a vez? */
int interested[N];                  /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);     /* processo é 0 ou 1 */
{
    int other;                      /* número de outro processo */

    other = 1 – process;            /* o oposto do processo */
    interested[process] = TRUE;     /* mostra que você está interessado */
    turn = process;                 /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) //* comando nulo */;
}

void leave_region(int process)      /* processo: quem está saindo */
{
    interested[process] = FALSE;    /* indica a saída da região crítica */
}
```


Instrução TSL (*test and set lock*)

- Instrução de máquina que lê o conteúdo de uma variável e armazena o valor 1 nela
  - operação atômica (indivisível)
- Exige suporte de hardware
- Funciona para vários processadores
  - barramento de memória é travado para evitar acessos simultâneos

Exclusão mútua com TSL

```
enter_region:
    TSL REGISTER,LOCK       | copia lock para o registrador e põe lock em 1
    CMP REGISTER,#0         | lock valia zero?
    JNE enter_region        | se fosse diferente de zero, lock estaria ligado,
                              portanto continue no laço de repetição
    RET | retorna a quem chamou; entrou na região crítica

leave_region:
    MOVE LOCK,#0            | coloque 0 em lock
    RET | retorna a quem chamou
```

Instrução XCHG (*eXCHanGe*)

- Instrução atômica que troca o conteúdo de dois registradores ou um registrador e uma posição de memória
  - disponível na arquitetura x86
- Exclusão mútua com XCHG

```
1: enter_region:
2:    mov  $1, %eax       ! EAX <- 1
3:    xchg %eax, lock     ! troca EAX e lock
4:    cmp  $0, %eax       ! se EAX==0, lock era 0, e RC estava livre
5:    jnz  enter_region   ! RC estava ocupada, fica no loop
6:    ret

7: leave_region:
8:    movl $0, lock
9:    ret
```


Exclusão mútua sem espera ocupada

- Soluções de exclusão mútua baseadas em espera ocupada são indesejáveis
  - um loop vazio ocupa o processador
- Isso evita que outros processos executem
  - incluindo um processo na seção crítica
- Pode causar **inversão de prioridade**
  - processo mais prioritário fica no loop e um menos prioritário não consegue liberar a seção crítica
- Melhor seria se o processo que encontra a seção crítica ocupada pudesse ficar bloqueado até que a seção crítica fosse liberada

Primitivas bloqueantes

- Primitivas bloqueantes bloqueiam o processo chamador até que ele seja sinalizado (tipicamente por outro processo)
  - algumas verificam uma condição e bloqueiam se ela não for satisfeita
  - outras bloqueiam incondicionalmente
    - ★ necessário cuidado para não introduzir condições de disputa
- Exemplos
  - `sleep()` e `wakeup()`
    - ★ `sleep()` é incondicional → difícil evitar condição de disputa
  - semáforos (variantes: mutexes, futexes)
  - monitores
  - barreiras
  - variáveis de condição

Semáforos

- Solução proposta por E. W. Dijkstra nos anos 60
- Um semáforo S é uma variável com dois atributos
  - um contador
  - uma fila de processos bloqueados no semáforo
  - quando negativo, o módulo do contador indica quantos processos estão bloqueados em S
- Semáforo só pode ser manipulado por duas primitivas **atômicas**
  - `down(S)`: decrementa S; se S<0, bloqueia
  - `up(S)`: incrementa S; se S≤0, acorda um processo que está esperando por S

Semáforos

- Solução proposta por E. W. Dijkstra nos anos 60
- Um semáforo S é uma variável com dois atributos
  - um contador
  - uma fila de processos bloqueados no semáforo
  - quando negativo, o módulo do contador indica quantos processos estão bloqueados em S
- Semáforo só pode ser manipulado por duas primitivas **atômicas**
  - `down(S)`: decrementa S; se S<0, bloqueia
  - `up(S)`: incrementa S; se S≤0, acorda um processo que está esperando por S

ATENÇÃO: essa definição das primitivas é ligeiramente diferente da definição do Tanenbaum

Exclusão mútua usando semáforos (1)

```
semaphore s = 1;
...
down(&s);
  /* região crítica */
up(&s);
/* região não crítica */
```

- Semáforos binários
  - inicializados em 1
  - controlam acesso à região crítica
    - ★ RC guardada com `down()` e `up()` sobre o mesmo semáforo
- RCs relacionadas devem estar protegidas pelo mesmo semáforo
  - acesso aos mesmos dados compartilhados

Exclusão mútua usando semáforos (2)

```
1: ...
2: down(&s);
3:  /* região crítica */
4: up(&s);
5: /* região não crítica */
```

tempo	P1	P2	s
t_1	1		1
t_2	2		$\not{1} \rightarrow 0$
t_3		1	0
t_4		2	$\not{0} \rightarrow -1$ (P2 dorme)
t_5	3		−1
t_6	4		$\not{-1} \rightarrow 0$ (acorda P2)
t_7		3	0
t_8		4	$\not{0} \rightarrow 1$

- Em t_2, P1 decrementa s de 1 para 0 e continua (RC livre)
- Em t_4, P2 decrementa s de 0 para −1 e bloqueia (RC ocupada)
- Em t_6, P1 incrementa s de −1 para 0, verifica que s ≤ 0, e acorda P2
- Caso simétrico aconteceria se P2 executasse a linha 2 antes de P1

Sincronização com semáforos (1)

- Semáforos também podem ser usados para **sincronizar** processos
  - garantir uma sequência desejada de execução
  - exemplo: P2 só pode executar uma instrução Y depois que P1 executou a instrução X
- Essa sincronização é implementada com P1 sinalizando uma condição para P2, indicando que X já foi executada
  - se a condição já foi satisfeita, P2 continua; caso contrário, ele espera a sinalização de P1
- Estrutura geral
  - semáforo com valor inicial zero
  - processo que **sinaliza** (P1) usa `up()`
  - processo que **espera** (P2) usa `down()`
- Semáforos contadores

Sincronização com semáforos (2)

- Exemplo: garantir que B3 só execute depois de A2

```
        semaphore s = 0;

A1  ...                                B1  ...
A2  fgets(str, MAX_STR, stdin);        B2  down(&s);
A3  up(&s);                            B3  processa(str);
A4  ...                                B4  ...
```

  - se A executar primeiro, `s` será 1, e B não bloqueia ao chegar em B2
  - se B executar primeiro, `s` será 0, e B bloqueia ao chegar em B2
    - ★ B será desbloqueado quando A executar A3

Implementação de semáforos

- Semáforos são implementados no núcleo do SO
- O semáforo é uma variável inteira
- Dificuldade é garantir atomicidade das operações `down()` e `up()`
- Uso de soluções com espera ocupada
  - desabilitação de interrupções
  - instrução TSL/XCHG

Problema dos produtores-consumidores

- Dois tipos de processos compartilham um buffer de tamanho limitado
  - produtor insere itens no buffer
    - ★ não pode inserir se o buffer estiver cheio
  - consumidor retira itens do buffer
    - ★ não pode retirar se o buffer estiver vazio
  - apenas um processo pode acessar o buffer em um dado momento
    - ★ acessos simultâneos ao buffer estão sujeitos a inconsistências
- Processos executam indefinidamente
- Generalização de diversas situações de IPC que ocorrem na prática
  - servidor web multithread
    - ★ thread despachante produz requisições
    - ★ threads operárias consomem requisições

Produtores-consumidores com semáforos

```
semaphore mutex = 1;  /* exclusão mútua no acesso ao buffer */
semaphore full = 0;   /* conta lugares preenchidos no buffer */
semaphore empty = N;  /* conta lugares vazios no buffer */
```

```
void produtor(void) {
  int item;
  while (TRUE) {
P1:  item = produz_item();
P2:  down(&empty);
P3:  down(&mutex);
P4:  insere_buf(item);
P5:  up(&mutex);
P6:  up(&full);
  }
}
```

```
void consumidor(void) {
  int item;
  while (TRUE) {
C1:  down(&full);
C2:  down(&mutex);
C3:  item = retira_buf();
C4:  up(&mutex);
C5:  up(&empty);
C6:  consome_item(item);
  }
}
```

Semáforos usados na solução

- `mutex`: semáforo binário
  - garante que apenas um processo acesse o buffer de cada vez
    - ★ exclusão mútua entre P-C, P-P e C-C
- `full`, `empty`: semáforos contadores
  - `empty` conta o nº de lugares vazios no buffer
    - ★ quando o buffer estiver cheio, `empty` ≤ 0, e o produtor bloqueia até que um consumidor retire um item
  - `full` conta o nº de lugares preenchidos no buffer
    - ★ quando o buffer estiver vazio, `full` ≤ 0, e o consumidor bloqueia até que um produtor insira um item
- Solução funciona para quaisquer quantidades de produtores e consumidores

Mutex como primitiva

- Em alguns casos é implementada uma versão simplificada de semáforos binários, chamada de mutex
  - apenas exclusão mútua, sem contagem
  - pode ser implementado em espaço de usuário
    - ★ desde que o processador suporte TSL/XCHG

```
1: mutex_lock:
2:    tsl  %eax, mutex    # EAX=mutex, mutex=1
3:    cmp  $0, %eax       # se EAX==0, mutex estava livre...
4:    jz   done           # ... mutex adquirido, pode encerrar
5:    call thread_yield   # escalona outra thread
6:    jmp  mutex_lock     # quando voltar, tenta novamente
7: done:                  # encerra quando thread obteve mutex
8:    ret                 #

9: mutex_unlock:
10:   mov  $0, mutex      # libera mutex
11:   ret
```

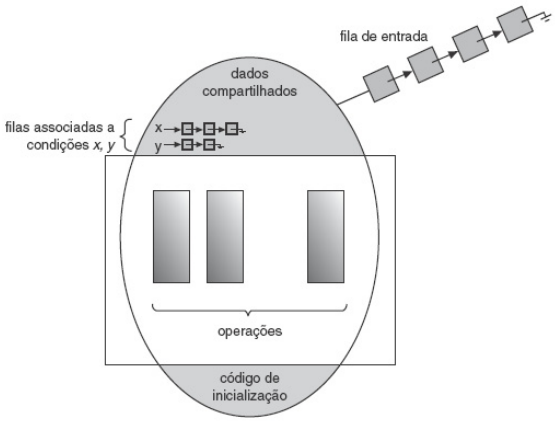

Futexes (*Fast userspace mutexes*)

- Mutexes em espaço de usuário são rápidos quando há pouca contenção
  - espera ocupada quase não ocorre na prática
- Chamadas para o kernel evitam espera ocupada, mas são lentas
  - ineficientes com pouca contenção → overhead
- Futexes tentam combinar os benefícios das duas abordagens
  1. A tentativa de travar um futex ocorre em espaço de usuário
  2. Se o futex já estava travado, ocorre uma chamada para o kernel para colocar o processo em uma fila de bloqueados
  3. Ao liberar o futex, o processo que o detinha verifica se há processos bloqueados; se houver, avisa ao kernel para desbloquear um deles
- Kernel só é envolvido quando ocorrer contenção

Monitores

- Sincronização baseada em uma construção de linguagem de programação
- Criados por Per Brinch Hansen e Tony Hoare na década de 70
- Um monitor encapsula dados privados e procedimentos que os acessam
  - semelhante a uma classe
- Apenas um processo pode estar ativo no monitor em um dado instante
  - exclusão mútua entre processos
- Sincronização é feita usando variáveis de condição
  - duas operações: `wait()` e `signal()`

Estrutura de um monitor



Semântica de `signal()`

- O que acontece quando um processo executa `signal()`?
  - *signal-and-exit*
    - ★ o processo atual deve sair do monitor após o `signal()`
    - ★ o processo sinalizado entra no monitor
  - *signal-and-continue*
    - ★ o processo atual continua executando
    - ★ o processo sinalizado compete pelo monitor no próximo escalonamento

Produtor-consumidor usando monitor

```
monitor ProdCons
  condition notFull, notEmpty;
  integer count := 0;

  procedure enter(item i);
  begin
    if count = N then wait(notFull);
    enter_item;
    count := count + 1;
    if count = 1 then signal(notEmpty);
  end;

  procedure remove;
  begin
    if count = 0 then wait(notEmpty);
    i := remove_item;
    count := count - 1;
    if count = N-1 then signal(notFull);
    return i;
  end;
end monitor;
```

```
procedure producer;
begin
  while true do
  begin
    i := produce_item;
    ProdCons.enter(i);
  end
end;

procedure consumer;
begin
  while true do
  begin
    i := ProdCons.remove;
    consume_item(i);
  end
end;
```


Monitores em Java

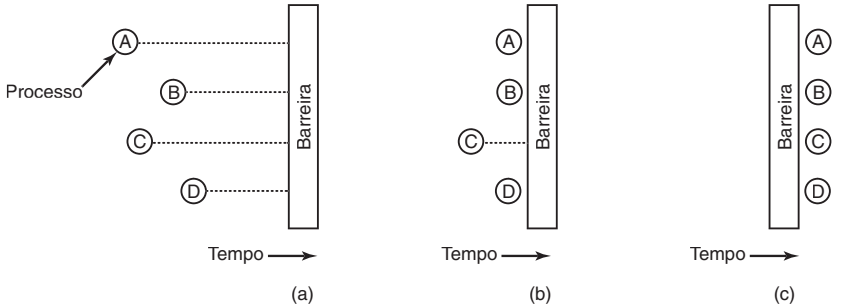
- Java tem suporte parcial a monitores através de métodos **synchronized**
- Exclusão mútua no acesso ao objeto
  - apenas para métodos `synchronized`
  - o que acontece se há métodos não-`synchronized`?
- Uma única variável de condição anônima e implícita
- Operações: `wait()`, `notify()`, `notifyAll()`
- Semântica *signal-and-continue*

Semáforos vs. monitores

- Monitores são mais fáceis de programar, e reduzem a probabilidade de erros
  - ordem de `down()` e `up()`
- Monitores dependem de linguagem de programação, enquanto semáforos são implementados pelo SO
  - podem ser usados com C, Java, BASIC, ASM, ...
- Ambos podem ser usados apenas em sistemas centralizados (com memória compartilhada)
  - sistemas distribuídos usam troca de mensagens

Barreiras

- Mecanismo usado para definir um ponto de sincronização para múltiplos processos/threads



Exemplo de barreira

```
#define NTHREADS 10

void *thread(void *arg) {
   pthread_barrier_wait(&barr);
   ...
}

int main(void) {
   pthread_barrier_init(&barr, NULL, NTHREADS);
   for (i=0; i < NTHREADS; i++) {
      rc = pthread_create(&thr[i], NULL, thread, NULL);
      ...
}
```

- Todas as threads iniciam “juntas”
  - mais precisamente, as 9 primeiras ficam esperando na barreira até que a décima chegue ali

Sumário

Mecanismos de IPC no Linux

- Sistemas Unix dão suporte a diversos mecanismos de IPC
  - pipes, filas de mensagens, memória compartilhada, semáforos
- Consideraremos aqui cinco mecanismos
  - Pthreads
    - ★ mutexes, variáveis de condição e barreiras
  - processos
    - ★ memória compartilhada e semáforos

Mutexes

- Usados para garantir **exclusão mútua** em regiões críticas
  - semelhantes a semáforos binários
- Principais chamadas envolvendo mutexes
  - criação: `pthread_mutex_init()`
  - uso: `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `pthread_mutex_trylock()`
  - destruição: `pthread_mutex_destroy()`

Criando um mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
```

- `pthread_mutex_t *mutex`: endereço do mutex
- `pthread_mutexattr_t *attr`: atributos do mutex
  - `NULL` para atributos default
- Mutex é criado destravado
- Retorna 0 para sucesso, outro valor em caso de erro
- Os dois trechos abaixo criam um mutex `mtx` inicializado com atributos default

```
1. pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

```
2. pthread_mutex_t mtx;
   pthread_mutex_init(&mtx, NULL);
```


Destruindo um mutex

Quando não é mais necessário, um mutex deve ser destruído

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- `pthread_mutex_t *mutex`: endereço do mutex
- Retorna 0 para sucesso, outro valor em caso de erro

Travando e destravando um mutex

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

- Retornam 0 para sucesso, outro valor em caso de erro
- Quando um mutex é destravado com `pthread_mutex_unlock()`, não é possível determinar qual das threads bloqueadas será escalonada
- `pthread_mutex_trylock()` trava o mutex caso esteja livre, ou retorna imediatamente (sem bloquear), devolvendo `EBUSY`
  - evita bloqueio
  - é preciso ter cuidado com condições de disputa

Exclusão mútua em Pthreads usando mutex

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
...
pthread_mutex_lock(&mtx);
  /* região crítica */
pthread_mutex_unlock(&mtx);
/* região não crítica */
```

- O mutex precisa ser compartilhado pelas threads, então é tipicamente uma variável global
- RCs relacionadas devem estar protegidas pelo mesmo mutex
  - acesso aos mesmos dados compartilhados

Variáveis de condição

- Mecanismo de **sincronização** entre threads
- Usadas em conjunto com mutexes
- Principais chamadas envolvendo variáveis de condição
  - criação: `pthread_cond_init()`
  - destruição: `pthread_cond_destroy()`
  - uso: `pthread_cond_wait()`, `pthread_cond_signal()`, `pthread_cond_broadcast()`

Criando uma variável de condição

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t *attr);
```

- `pthread_cond_t *cond`: endereço da variável de condição
- `pthread_condattr_t *attr`: atributos da variável de condição
  - `NULL` para atributos default
- Retorna 0 para sucesso, outro valor em caso de erro
- Os dois trechos abaixo criam uma variável de condição `cond` inicializada com atributos default

```
1. pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

```
2. pthread_cond_t cond;
   pthread_cond_init(&cond, NULL);
```

Destruindo uma variável de condição

Quando não é mais necessária, uma variável de condição deve ser destruída

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- `pthread_cond_t *cond`: endereço da variável de condição
- Retorna 0 para sucesso, outro valor em caso de erro

Esperando por uma variável de condição

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

- Bloqueia a thread até que `cond` seja sinalizada
- **O acesso à variável de condição `cond` deve estar protegido por `mutex`**
  - `mutex` é automaticamente destravado quando a thread bloqueia
  - quando a condição for sinalizada, a thread retoma a execução com `mutex` travado para seu uso
  - é preciso destravar `mutex` ao final da região crítica
- Deve-se verificar se a condição foi efetivamente satisfeita, pois o desbloqueio pode não ter sido causado pela sinalização da variável
- Retorna 0 para sucesso, outro valor em caso de erro

Sinalizando uma variável de condição

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- `pthread_cond_signal()` acorda uma thread bloqueada por uma variável de condição
  - deve ser invocada com `mutex` travado
  - thread sinalizada só retoma execução depois que `mutex` for destravado por quem executou `pthread_cond_signal()`
- `pthread_cond_broadcast()` acorda todas as threads bloqueadas por uma variável de condição
  - valem as mesmas restrições de `pthread_cond_signal()`
- **Se nenhuma thread estiver bloqueada esperando pela condição, a sinalização é perdida**
  - semelhante a `sleep()` e `wakeup()`
- Retornam 0 para sucesso, outro valor em caso de erro

Sincronização com variáveis de condição (1)

- Variáveis de condição precisam estar associadas a um **predicado**
  - uma variável lógica ou outra condição (ex. `nitens == 0`)
- Antes de invocar `pthread_cond_signal()`, pode ser necessário indicar que o predicado foi satisfeito

```
pthread_mutex_lock(&mtx);
predicado = TRUE;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&mtx);
```

- Antes de invocar `pthread_cond_wait()`, é preciso verificar o predicado

```
pthread_mutex_lock(&mtx);
while (!predicado)
  pthread_cond_wait(&cond, &mtx);
...
pthread_mutex_unlock(&mtx);
```

  - o `while` é necessário porque outra thread pode obter o mutex antes e invalidar o predicado

Sincronização com variáveis de condição (2)

- Exemplo: garantir que a thread B só execute `processa()` depois da thread A executar `fgets()`

```
pthread_mutex_t mtxl = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condl = PTHREAD_COND_INITIALIZER;
int str_lida = FALSE;
```

thread A

```
...
pthread_mutex_lock(&mtxl);
fgets(str, MAX_STR, stdin);
str_lida = TRUE;
pthread_cond_signal(&condl);
pthread_mutex_unlock(&mtxl);
...
```

thread B

```
...
pthread_mutex_lock(&mtxl);
while (!str_lida)
   pthread_cond_wait(&condl, mtxl);
processa(str);
pthread_mutex_unlock(&mtxl);
...
```


Barreiras

- Definem um ponto único de **sincronização** para múltiplas threads
- Principais chamadas envolvendo barreiras:
  - criação: `pthread_barrier_init()`
  - destruição: `pthread_barrier_destroy()`
  - uso: `pthread_barrier_wait()`

Criação e destruição de barreira

```
int pthread_barrier_init(pthread_barrier_t *barrier,
    pthread_barrierattr_t *attr, unsigned count);

int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- `pthread_barrier_t *barrier`: endereço da barreira
- `pthread_barrierattr_t *attr`: atributos da barreira
  - `NULL` para atributos default
- `unsigned count`: número de threads que esperam na barreira
  - deve ser maior que 0
- Retornam 0 para sucesso, outro valor em caso de erro

Sincronizando em uma barreira

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- `pthread_barrier_t *barrier`: endereço da barreira
- Espera na barreira `barrier` até que `count` threads o façam
- Quando a última thread chega na barreira, esta é reiniciada com `count`
  - se `count` for menor que o número de threads que invocam `pthread_barrier_wait()`, algumas threads podem ficar bloqueadas indefinidamente

Criação de memória compartilhada (1)

- Processos no Unix possuem seu próprio espaço de endereçamento
  - mesmo um processo criado com `fork()` tem **apenas uma cópia** do espaço de endereçamento do processo pai
- É possível definir regiões de memória compartilhada entre processos

Criação de memória compartilhada (2)

- Criação de memória compartilhada tem 3 etapas
  1. Obter um descritor de arquivo para um objeto
  2. Definir o tamanho do objeto
  3. Mapear o objeto na memória e obter um ponteiro para ele

Exemplo: compartilhando um inteiro

```
int *ptr, rc, fd;

fd = shm_open("/shm", O_RDWR | O_CREAT, S_IRUSR | S_IWUSR); /*1*/
if (fd == -1) exit(1);
rc = ftruncate(fd, sizeof(int));                            /*2*/
if (rc == -1) exit(2);
ptr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
           fd, 0);                                          /*3*/
if (ptr == MAP_FAILED) exit(3);
```

- `ptr` aponta para a área de memória compartilhada

`shm_open()`

```
int shm_open(const char *name, int oflag, mode_t mode);
```

- `char *name`: nome a ser dado para a região de memória
  - deve começar por / e conter caracteres válidos para nomear arquivos
  - se `name=="abc"`, será criado um arquivo `/dev/shm/abc`
- `int oflag`: flags de abertura (combinadas com `|`)
  - `O_RDONLY` (leitura) **ou** `O_RDWR` (leitura e escrita)
  - `O_CREAT`: cria o objeto caso não exista
  - `O_EXCL`: se usada com `O_CREAT` retorna erro caso o objeto exista
  - `O_TRUNC`: trunca o objeto caso já exista
- `mode_t mode`: define as permissões de acesso ao recurso
  - usado apenas quando o recurso é criado (`| O_CREAT`)
  - flags são as mesmas usadas pela chamada `open(2)`
    - `S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR`
    - `S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP`
    - `S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH`
- Retorna um descritor de arquivo ou -1 em caso de erro
- **Tamanho inicial do objeto é zero**

`ftruncate()`

```
int ftruncate(int fd, off_t length);
```

- Usado para definir o tamanho da região de memória compartilhada
  - `shm_open()` cria região com tamanho zero
- `int fd`: descritor de arquivo retornado por `shm_open()`
- `off_t length`: tamanho desejado
  - bytes do conteúdo são zerados
- retorna 0 para sucesso ou -1 em caso de erro

mmap()

```
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- Mapeia `length` bytes do arquivo `fd` na memória, iniciando em `offset`
- `void *start`: endereço inicial preferencial para mapear o arquivo
  - `NULL` indica que não há preferência
  - endereço efetivamente usado é retornado pela função
- `size_t length`: tamanho da área de memória a mapear
  - tipicamente é o tamanho do arquivo
- `int prot`: flags de proteção para a área de memória
  - `PROT_NONE` → memória não pode ser acessada
  - uma combinação de `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`

mmap()

```
void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

- `int flags`: opções de mapeamento
  - para memória compartilhada, deve incluir `MAP_SHARED`
- `int fd`: descritor do arquivo a ser mapeado
  - retornado por `shm_open()`
- `off_t offset`: posição inicial do arquivo, em bytes
  - 0 para início
- `mmap()` retorna o endereço inicial do mapeamento ou `MAP_FAILED` em caso de erro
- O descritor de arquivo pode ser fechado sem afetar o acesso à memória compartilhada

Liberação de memória compartilhada

```
int munmap(void *start, size_t length);
```

- Desfaz o mapeamento de memória de `length` bytes começando em `start`
- Parâmetros devem ser os mesmos usados no `mmap()`

```
int shm_unlink(const char *name);
```

- Remove a área de memória compartilhada chamada `name`
- Objeto só é destruído quando todos os mapeamentos tiverem sido desfeitos (com `munmap()`)

Exemplo: liberando o inteiro compartilhado

```
rc = munmap(ptr, sizeof(int));
if (rc == -1) exit(7);
rc = shm_unlink("/shm");
if (rc == -1) exit(8);
```


Semáforos POSIX

- Existem duas APIs para uso de semáforos em Unix
  - POSIX
  - System V
  - a API POSIX é mais simples, porém menos portável
- Principais chamadas da API POSIX
  - criação: `sem_open()`, `sem_init()`
  - uso: `sem_wait()`, `sem_post()`
  - liberação: `sem_destroy()`, `sem_close()`, `sem_unlink()`
- Semáforos POSIX podem ser usados tanto com processos quanto com threads

Semáforos anônimos e nomeados

- Existem dois tipos de semáforos, **nomeados** (*named*) e **anônimos** (*unnamed*)
- A diferença é que semáforos nomeados têm um arquivo associado
  - permite que processos não relacionados acessem um mesmo semáforo
- Os semáforos anônimos precisam estar em memória compartilhada
  - processos: entram na região de memória compartilhada
    - ★ `shm_open()` + `mmap()`
  - threads: podem ser usadas variáveis globais ou variáveis alocadas dinamicamente no heap
    - ★ `malloc()`

Criação de semáforos nomeados

```
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
                unsigned int value);
```

- `char *name`: nome do semáforo
  - valem as mesmas regras de `shm_open()`
  - se `name=="abc"`, será criado um arquivo `/dev/shm/sem.abc`
- `int oflag`: flags de criação
  - pode conter 0 ou `O_CREAT` e `O_EXCL`
  - se `O_CREAT` for usada, `mode` e `value` têm que estar presentes
- `mode_t mode`: permissões de acesso
  - mesmos valores de `shm_open()`
- `unsigned int value`: valor inicial do semáforo
- A função retorna um ponteiro para o semáforo criado ou `SEM_FAILED` em caso de erro

Criação de semáforos anônimos

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

- `sem_t *sem`: endereço do semáforo (tipicamente passado por referência)
- `int pshared`: indica se o semáforo é compartilhado pelas threads do mesmo processo (= 0) ou por processos distintos (≠ 0)
- `unsigned int value`: valor inicial do semáforo
- Retorna 0 para sucesso ou -1 em caso de erro

Operações sobre semáforos

- `int sem_wait(sem_t *sem);`
  - equivale a `down(&sem)`
- `int sem_post(sem_t *sem);`
  - equivale a `up(&sem)`
- Ambas as funções retornam 0 para sucesso e -1 em caso de erro
  - em caso de erro, o valor do semáforo não é alterado

Liberação de semáforos

- Semáforos anônimos
  - `int sem_destroy(sem_t *sem);`
    - ★ libera os recursos do SO associados ao semáforo
    - ★ se houver algum processo bloqueado no semáforo o comportamento é indefinido
- Semáforos nomeados
  - `int sem_close(sem_t *sem);`
    - ★ desvincula o semáforo do processo
  - `int sem_unlink(const char *name);`
    - ★ libera os recursos do SO associados ao semáforo
    - ★ semáforo só é efetivamente destruído quando não estiver sendo usado por nenhum processo
- Todas as funções retornam 0 para sucesso e -1 em caso de erro

Exclusão mútua com semáforos anônimos e threads

```
sem_t s;
...
sem_init(&s, 0, 1);   /* s = 1 */
...
sem_wait(&s);
  /* região crítica */
sem_post(&s);
/* região não crítica */
```

- Semáforos binários
  - inicializados em 1
  - controlam acesso à região crítica
    - ★ RC guardada com `sem_wait()` e `sem_post()` sobre o mesmo semáforo
- RCs relacionadas devem estar protegidas pelo mesmo semáforo
  - acesso aos mesmos dados compartilhados

Sincronização com semáforos anônimos e threads

- Exemplo: garantir que B3 só execute depois de A2

```
sem_t s;
sem_init(&s, 0, 0);
```

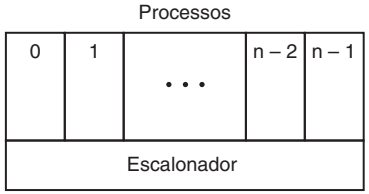
A1	`...`	B1	`...`
A2	`fgets(str, MAX_STR, stdin);`	B2	`sem_wait(&s);`
A3	`sem_post(&s);`	B3	`processa(str);`
A4	`...`	B4	`...`

  - se A executar primeiro, `s` será 1, e B não bloqueia ao chegar em B2
  - se B executar primeiro, `s` será 0, e B bloqueia ao chegar em B2
    - ★ B será desbloqueado quando A executar A3

Sumário

Conceito de escalonamento

- Uma CPU ou núcleo é um recurso indivisível
- Um requisito básico de sistemas multiprogramados é decidir qual processo deve executar a seguir, e por quanto tempo
  - multiplexação no tempo
  - o componente do SO que faz isso é o **escalonador** (*scheduler*)
  - o escalonador implementa um **algoritmo de escalonamento**
- Visão dos processos



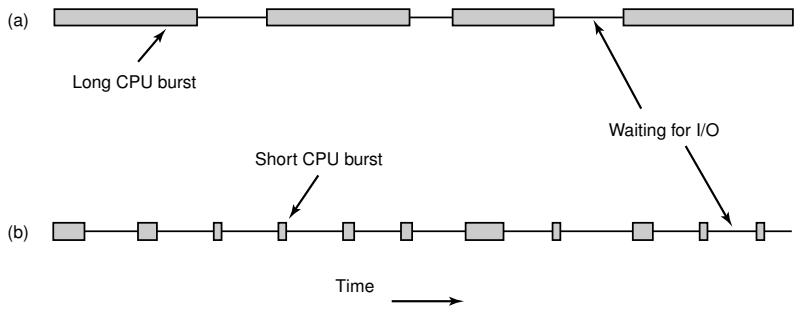
Conceito de escalonamento

- Para trocar o processo em execução é necessário um **chaveamento de contexto**
  1. Contexto do processo atual é salvo na tabela de processos
     - ★ registradores de CPU/memória + estruturas de dados do núcleo
  2. Contexto do novo processo é carregado da tabela de processos
  3. Novo processo inicia sua execução
- Algoritmos se diferenciam pelo trade-off entre quanto tempo cada processo executa e a responsividade do sistema
  - minimizar overhead de troca de contexto × minimizar tempo de espera pela CPU
  - todos visam a usar a CPU de modo eficiente

Comportamento dos processos

- Em geral, processos alternam ciclos de uso de CPU com ciclos de requisição de E/S
  - o processo executa várias instruções de máquina e faz uma chamada de sistema solicitando um serviço do SO
- Existem duas grandes classes de processos
  - orientados a CPU (*CPU-bound*)
  - orientados a E/S (*I/O-bound*)
  - há processos que alternam essas características

Representação do comportamento



(a) um processo orientado a CPU

(b) um processo orientado a E/S

Quando escalonar

Existem diversas situações em que o escalonador é invocado

- na criação de um processo
- no encerramento de um processo
- quando um processo bloqueia
- quando ocorre uma interrupção de E/S
- quando ocorre uma interrupção de relógio
  - escalonamento preemptivo

Escalonamento preemptivo e não preemptivo

- No escalonamento não preemptivo, um processo só pára de executar na CPU se quiser
  - invocação de uma chamada de sistema
  - liberação voluntária da CPU
- No escalonamento preemptivo um processo pode perder a CPU mesmo contra sua vontade
  - preempção por tempo (mais comum)
  - preempção por prioridade
    - ★ chegada de um processo mais prioritário
  - além das possibilidades do não preemptivo

Categorias de algoritmos

- Existem três categorias básicas de algoritmos de escalonamento
- **Lote (batch)**
  - sem usuários interativos
  - ciclos longos são aceitáveis – menos preempções
  - em alguns casos o tempo de execução pode ser estimado
- **Interativo**
  - com usuários interativos
  - tempo de execução é indeterminado
  - ciclos curtos para que todos os processos progridam
- **Tempo real**
  - processos com requisitos temporais específicos

Objetivos do algoritmo de escalonamento

- Quais os critérios podem ser usados para avaliar um algoritmo de escalonamento?
- **Vazão (throughput)**: número de jobs processados por hora
- **Tempo de retorno**: tempo médio do momento que um job é submetido até o momento em que foi terminado
  - mais importante para sistemas em lote
- **Tempo de reação** (*response time*): tempo entre a emissão de um comando e a obtenção do resultado
  - mais importante para sistemas interativos

Algoritmos de escalonamento

- Escalonamento para sistemas em lote
  1. Primeiro a chegar, primeiro a ser servido (FCFS)
  2. Job mais curto primeiro (SJF)
  3. Próximo de menor tempo restante (SRTN)
- Escalonamento para sistemas interativos
  1. Alternância circular (*round-robin*)
  2. Por prioridades
  3. Filas múltiplas
  4. Fração justa

FCFS

- Processos são atendidos por ordem de chegada
  - primeiro a chegar, primeiro a ser servido
  - *first come, first served (FCFS)*
- O processo escalonado usa a CPU por quanto tempo quiser – não preemptivo
  - até encerrar, bloquear ou entregar o controle
- Simples de implementar
- Não diferencia processos orientados a CPU e orientados a E/S
  - pode prejudicar os orientados a E/S

SJF (*Shortest Job First*)

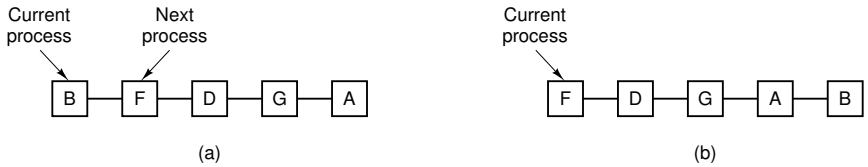
- Os processos mais curtos são atendidos primeiro
  - mais curto = menor tempo de CPU
- Não preemptivo
- Algoritmo com menor tempo médio de retorno
- Premissas
  - todos os jobs estão disponíveis simultaneamente
  - a duração dos ciclos de CPU é conhecida a priori

Próximo de menor tempo restante

- *Shortest remaining time next (SRTN)*
- Variante preemptiva do SJF
- Quando chega um novo processo, seu tempo de CPU é comparado com o tempo restante do processo que está executando
  - se for menor, o processo atual é preemptado e o novo processo escalonado em seu lugar
- Garante bom desempenho para jobs curtos
- Requer tempos conhecidos de CPU

Alternância circular (*round-robin*)

- Cada processo que ganha a CPU executa durante um determinado tempo (o **quantum**)
- Se o processo não liberar a CPU, ao final do quantum ele perde o processador e volta para a fila de prontos
  - algoritmo preemptivo
- Exemplo: B usa todo o seu quantum



Determinando o quantum

- A decisão de projeto mais importante no *round-robin* é o tamanho do quantum
- Quanto menor o quantum, maior o overhead
  - tempo para chaveamento de contexto se aproxima do tempo de execução
- Quanto maior o quantum, pior o tempo de reação
  - ocorrem menos preempções
  - processo demora mais a ser escalonado
  - prejudica processos orientados a E/S
- Na prática, o quantum fica entre 20 e 100 ms

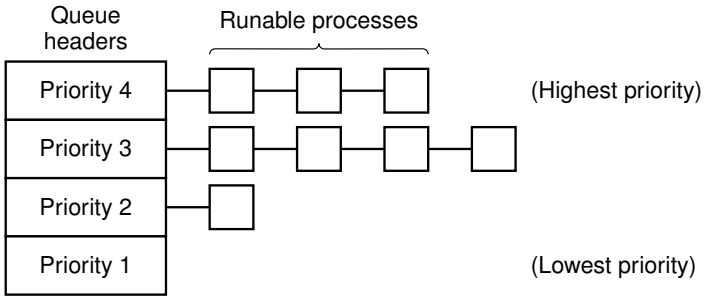
Escalonamento por prioridades (1/3)

- Nem todos os processos têm a mesma prioridade
  - o antivírus não deve prejudicar a exibição de um vídeo, por exemplo
- Escalonamento por prioridades
  - cada processo recebe uma prioridade
  - escala de prioridades pode ser
    - ★ positiva: mais prioritário ⇒ *maior* valor de prioridade
    - ★ negativa: mais prioritário ⇒ *menor* valor de prioridade
  - o processo mais prioritário executa
  - para evitar que processos mais prioritários executem indefinidamente, a prioridade pode ser periodicamente ajustada
  - prioridade preemptiva vs não preemptiva

Escalonamento por prioridades (2/3)

- Prioridades podem ser estáticas ou dinâmicas
  - igual à fração do último quantum usada, p.ex.
- É comum agrupar os processos em classes de prioridades
  - prioridade entre as classes
  - *round-robin* dentro de cada classe

Escalonamento por prioridades (3/3)



Inanição e envelhecimento

- No escalonamento por prioridades, os processos de baixa prioridade podem sofrer **inanição** (*starvation*)
  - nunca serem escalonados devido aos processos de alta prioridade monopolizarem o processador
    - ★ processos de longa duração, que não bloqueiam
    - ★ chegada constante de novos processos de alta prioridade
- A solução é usar um mecanismo de **envelhecimento** (*aging*)
  - aumenta a prioridade dos processos que estão há muito tempo na fila de prontos sem executar
    - ★ prioridades dinâmicas, e não estáticas

Filas múltiplas

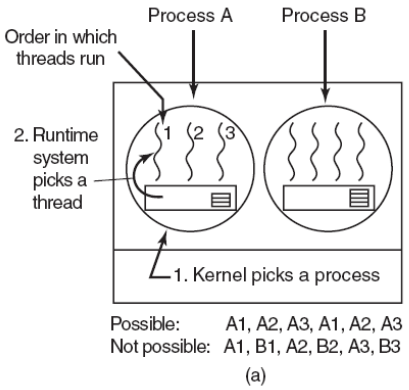
- Variante do escalonamento por prioridades
- Cada classe de prioridade tem um quantum
  - classes mais prioritárias têm quantum menor
  - se o quantum acaba antes que o processo consiga concluir o ciclo de CPU, ele muda de prioridade
- Reduz a quantidade de chaveamentos de contexto para processos orientados a CPU
- Processos interativos têm alta prioridade
  - usuários de processos em lote descobriram que podiam acelerar seus processos usando o terminal

Escalonamento por fração justa

- *Fair share scheduling*
- Os algoritmos anteriores tratam todos os processos de forma igual
  - usuários com muitos processos têm mais tempo de CPU do que usuários com poucos processos
- A ideia do *fair share* é atribuir uma fração da CPU para cada usuário
  - o escalonador escolhe o processo a executar de modo a respeitar essas frações
- Outras possibilidades existem, dependendo da noção de “justiça”

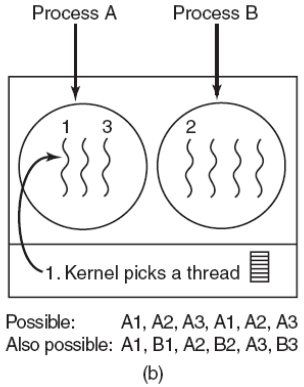
Escalonamento de threads de usuário

- O núcleo escalona um processo, e o escalonador do runtime pode chavear entre as threads desse processo
  - preempção em geral voluntária (*yield*), não por tempo
  - qualquer algoritmo de escalonamento pode ser usado, inclusive um específico para a aplicação



Escalonamento de threads de núcleo

- O escalonador do SO escolhe uma thread de qualquer processo
  - processo pode ou não ser considerado pelo algoritmo
    - ★ como chavear processos é mais caro que chavear threads, escalonador pode dar preferência a outra thread do mesmo processo
  - escalonamento específico para a aplicação é inviável



Sumário

Escalonamento no Linux

- Linux possui threads de kernel
  - escalonamento de threads, não de processos
- Para o escalonamento, diferentes classes de threads são consideradas, em ordem de precedência
  1. Com deadline (`SCHED_DEADLINE`)
  2. Tempo real (`SCHED_FIFO`, `SCHED_RR`)
  3. Tempo compartilhado (`SCHED_OTHER`, `SCHED_BATCH`)
- Cada classe usa um algoritmo de escalonamento diferente

Threads SCHED_DEADLINE

- Classe específica para threads com requisitos de tempo real
  - uma thread deve receber R μs de tempo de execução a cada P μs, e a execução deve ocorrer dentro de D μs a partir do início do período
    - ★ P: período, intervalo entre execuções consecutivas
    - ★ D: deadline (prazo)
    - ★ R: tempo de execução no pior caso (WCET)
- Usa o algoritmo *Earliest Deadline First* (EDF)
  - a thread com o menor deadline executa primeiro
  - um mecanismo de controle de admissão é usado para garantir que todos os deadlines podem ser respeitados
    - ★ se não for possível garantir, `sched_setattr()` falha
  - a fração de tempo que pode ser usada por threads `SCHED_DEADLINE` é configurável (default 95%)
    - ★ `/proc/sys/kernel/sched_rt_runtime_us` / `sched_rt_period_us`

Threads de tempo real

- Apesar do nome, não há deadlines ou garantias temporais
- Cada thread possui uma prioridade estática entre 1 (menos prioritária) e 99 (mais prioritária)
- O escalonador simplesmente escolhe a primeira thread pronta na fila mais prioritária
  - maior número de prioridade
- Cada CPU tem sua própria fila

Threads SCHED_FIFO

- Escalonadas em ordem
- Uma thread executa até
  - bloquear
  - liberar voluntariamente a CPU
    - ★ invocando `sched_yield()`
  - uma thread mais prioritária ficar pronta
    - ★ chegar no sistema ou ser desbloqueada
- A thread suspensa volta para a fila de sua mesma prioridade

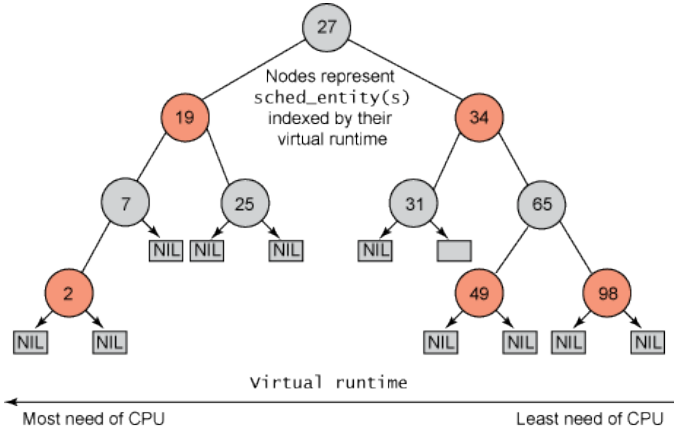
Threads SCHED_RR

- Semelhantes às threads FIFO, mas podem ser preemptadas por tempo
- Quando uma thread perde o processador, o tempo de CPU usado é descontado do quantum
  - quando ela retoma o processador, o valor remanescente é usado
- Quando o quantum zera, ele é restaurado ao valor inicial e a thread colocada no final da fila da sua prioridade
- Threads `SCHED_FIFO` e `SCHED_RR` com a mesma prioridade ficam na mesma fila
  - a única diferença é a preempção por tempo com `SCHED_RR`

Threads SCHED_OTHER e SCHED_BATCH

- *Completely Fair Scheduler* (CFS)
- Threads são ordenadas pelo seu tempo virtual de execução
  - tempo de execução em ns, ponderado pela prioridade (valor de *nice*, $-20 \ldots + 19$)
  - threads SCHED_BATCH são consideradas CPU-bound e penalizadas com tempo virtual de execução maior
- Thread com o menor tempo virtual é a próxima a executar
  - a fatia de tempo é calculada dinamicamente, em função da carga no sistema
  - limites superior e inferior são usados para manter a fatia dentro de uma faixa aceitável
- O tempo virtual das threads é mantido em uma árvore rubro-negra
  - uma árvore (*runqueue*) por CPU
  - buscas, inserções e remoções são $O(\log n)$
  - menor tempo virtual é sempre o elemento mais à esquerda

Completely Fair Scheduler (CFS)



https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Escalonamento em multiprocessadores

- Em sistemas com vários processadores, existem benefícios de se manter uma thread sempre na mesma CPU
  - aproveitamento da cache do processador
  - isso é chamado de **afinidade de processador**
- *Runqueues* por CPU favorecem essa afinidade
- Periodicamente o escalonador balanceia a carga entre as CPUs, levando em consideração desempenho e afinidade

Estruturas adicionais

- Escalonador considera apenas threads escalonáveis
  - conteúdo das *runqueues*
- Threads bloqueadas são colocadas em **waitqueues**
  - cada evento pelo qual uma thread pode estar esperando possui uma *waitqueue*
  - facilita o desbloqueio de threads
- Filas no kernel são gerenciadas com o auxílio de **spinlocks** para garantir exclusão mútua
  - variáveis do tipo `lock`
  - usam instrução tipo TSL em multiprocessadores

Bibliografia


Andrew S. Tanenbaum.
Sistemas Operacionais Modernos, 4ª Edição. Capítulos 2 e 10 (Linux).
Pearson Prentice Hall, 2016.


Carlos A. Maziero.
Sistemas Operacionais: Conceitos e Mecanismos. Capítulos 4–6, 10–12.
Editora da UFPR, 2019.
http://wiki.inf.ufpr.br/maziero/doku.php?id=socm:start