Sincronização

Wedson Almeida Filho, 09/10/2023

Revisão de processos e threads

- Processo
  - Instância de um programa rodando
  - Cópias do mesmo programa rodando são independentes
    - Endereço 0x100 no processo 20 é diferente do endereço 0x100 no processo 21
  - Processos podem escolher compartilhar parte da memória
- Thread
  - Linha de execução dentro de um processo
  - Compartilha memória, signais, arquivos, etc. com outras threads no mesmo processo
- Windows: processo é um container de threads
- Linux: processo é o grupo de "tasks" que têm o mesmo "líder"

Execução de threads

- Em sistemas operacionais modernos, a execução é concorrente
  - Um escalonador preemptivo periodicamente:
    - Interrompe a thread rodando
    - Salva o estado da thread
    - Restaura o estado da próxima thread
    - Deixa a próxima thread rodar pelo seu quantum
- A execução pode ser também *paralela*
  - Se o dispositivo tiver mais de uma CPU lógica
  - Mais de uma thread ao mesmo tempo pode rodar

Riscos do paralelismo ou concorrência preemptiva

- Instruções em duas threads podem ser intercaladas arbitrariamente
- Programas que parecem ser corretos nem sempre funcionam em todas as intercalações
- Exemplo:

```
bool saque(struct conta *c, float valor)
{
    float saldo = c->saldo;
    if (valor > saldo) {
        return false;
    }

    c->saldo = saldo - valor;

    return true;
}
```

Threads podem ser interrompidas em cada instrução

- Um exemplo mais simples:

```
void *thread(void *arg)
{
    int i;
    for (i = 0; i < 1000000; i++) {
        counter++;
    }
    return NULL;
}
```

Exemplo anterior em assembly do x86-64

```
12c0:    f3 0f 1e fa           endbr64
12c4:    ba 40 42 0f 00        mov    $0xf4240,%edx
12c9:    0f 1f 80 00 00 00 00  nopl   0x0(%rax)
12d0:    48 8b 05 59 2d 00 00  mov    0x2d59(%rip),%rax  # 4030 <counter>
12d7:    48 83 c0 01           add    $0x1,%rax
12db:    48 89 05 4e 2d 00 00  mov    %rax,0x2d4e(%rip)  # 4030 <counter>
12e2:    83 ea 01              sub    $0x1,%edx
12e5:    75 e9                 jne    12d0 <thread+0x10>
12e7:    31 c0                 xor    %eax,%eax
12e9:    c3                    ret
```

Corrida de dados

- Condição de existência:
  - Pelo menos uma thread escrevendo em uma posição de memória
  - Pelo menos uma thread lendo ou escrevendo a mesma posição de memória
- Consequência
  - Dados podem ser inconsistentes
  - Resultados indesejados
    - Valores “rasgados” (*torn*)
    - Valores desatualizados
    - Computação incorreta
    - Vulnerabilidade de segurança

Exemplo de corrida de dados em segurança

- TOCTOU - Time Of Check to Time Of Use
  - Usuário faz uma chamada de sistema
  - Kernel lê um dado e faz a verificação
  - Usuário modifica dados após verificação
  - Kernel executa operação no novo dado que não foi verificado

- Solução para esses casos é o kernel "capturar" os dados
  - Caso especial porque a relação é de adversários

Seção crítica

- Cada thread pode ter uma ou mais seções críticas
  - Regiões de código com corrida de dados em potencial
- Uma vez identificadas as seções críticas, há 3 requisitos
  - Exclusão mútua: no máximo uma thread executa uma seção crítica por vez.
  - Progresso: quando não há contenção e múltiplas threads querem entrar em seções críticas, elas decidem qual entrará. A decisão não pode ser adiada indefinidamente.
  - Espera limitada: existe um limite para a quantidade de vezes que uma thread pode ser rejeitada quando deseja entrar em uma região crítica.

Algoritmo de Peterson (copiado do livro de Tanenbaum)

https://pastebin.com/LAGtDbQ1

```
#define FALSE 0
#define TRUE  1
#define N     2                       /* number of processes */

int turn;                             /* whose turn is it? */
int interested[N];                    /* all values initially 0 (FALSE) */

void enter_region(int process)        /* process is 0 or 1 */
{
     int other;                       /* number of the other process */
     other = 1 - process;             /* the opposite of process */
     interested[process] = TRUE;      /* show that you are interested */
     turn = process;                  /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)        /* process: who is leaving */
{
     interested[process] = FALSE;     /* indicate departure from critical region */
}
```

Prática 1: Testando o algoritmo de Peterson

- Adaptar exemplo do pthread_mutex para usar enter_region/leave_region
- Executar exemplo e verificar resultado final da contagem

- Atualizando thread para usar argumento:

```
void *thread(void *arg)
{
        int proc = (size_t)arg;
        size_t i = 1000000;
        while (i--) {
                enter_region(proc);
                valor++;
                leave_region(proc);
        }
        return NULL;
}
```

Prática 2: Compilar com otimizações

- Compilar com -O3
- Qual é o resultado?

- Observar no gdb o estado do programa

Verificar no gdb o estado do programa

```
Dump of assembler code for function thread:
   0x0000000000001250 <+0>:  endbr64
   0x0000000000001254 <+4>:  mov    %rdi,%r9
   0x0000000000001257 <+7>:  mov    0x2e0a(%rip),%r8        # 0x4068 <valor>
   0x000000000000125e <+14>: mov    $0x1,%edi
   0x0000000000001263 <+19>: lea    0x2e06(%rip),%rax        # 0x4070 <interested>
   0x000000000000126a <+26>: sub    %r9d,%edi
   0x000000000000126d <+29>: movslq %r9d,%rcx
   0x0000000000001270 <+32>: lea    0x1(%r8),%rsi
   0x0000000000001274 <+36>: movslq %edi,%rdi
   0x0000000000001277 <+39>: add    $0xf4241,%r8
   0x000000000000127e <+46>: xchg   %ax,%ax
   0x0000000000001280 <+48>: movl   $0x1,(%rax,%rcx,4)
   0x0000000000001287 <+55>: cmpl   $0x1,(%rax,%rdi,4)
   0x000000000000128b <+59>: jne    0x1290 <thread+64>
   0x000000000000128d <+61>: jmp    0x128d <thread+61>
   0x000000000000128f <+63>: nop
   0x0000000000001290 <+64>: lea    0x1(%rsi),%rdx
   0x0000000000001294 <+68>: movl   $0x0,(%rax,%rcx,4)
   0x000000000000129b <+75>: cmp    %r8,%rdx
   0x000000000000129e <+78>: je     0x12a5 <thread+85>
   0x00000000000012a0 <+80>: mov    %rdx,%rsi
   0x00000000000012a3 <+83>: jmp    0x1280 <thread+48>
   0x00000000000012a5 <+85>: mov    %r9d,0x2dcc(%rip)        # 0x4078 <turn>
   0x00000000000012ac <+92>: xor    %eax,%eax
   0x00000000000012ae <+94>: mov    %rsi,0x2db3(%rip)        # 0x4068 <valor>
   0x00000000000012b5 <+101>:     ret
```

Prática 3: Adicionar `volatile` às variáveis de Peterson

- Adicionar volatile
- Executar e verificar resultado

- Declarações:

```
volatile int turn;                  /* whose turn is it? */
volatile int interested[N];         /* all values initially 1 (FALSE)
*/
```

Prática 4: Compilar com otimizações

- Compilar com -O3
- Qual é o resultado?

- Observar o código gerado

Código gerado

```
Dump of assembler code for function thread:
   0x0000000000001250 <+0>:  endbr64
   0x0000000000001254 <+4>:  mov    $0x1,%esi
   0x0000000000001259 <+9>:  movslq %edi,%r8
   0x000000000000125c <+12>: mov    0x2e05(%rip),%r9        # 0x4068 <valor>
   0x0000000000001263 <+19>: lea    0x2e06(%rip),%rdx        # 0x4070 <interested>
   0x000000000000126a <+26>: sub    %edi,%esi
   0x000000000000126c <+28>: mov    %r8,%rcx
   0x000000000000126f <+31>: mov    $0xf4240,%edi
   0x0000000000001274 <+36>: movslq %esi,%rsi
   0x0000000000001277 <+39>: nopw   0x0(%rax,%rax,1)
   0x0000000000001280 <+48>: movl   $0x1,(%rdx,%r8,4)
   0x0000000000001288 <+56>: mov    %ecx,0x2dea(%rip)        # 0x4078 <turn>
   0x000000000000128e <+62>: jmp    0x1298 <thread+72>
   0x0000000000001290 <+64>: mov    (%rdx,%rsi,4),%eax
   0x0000000000001293 <+67>: cmp    $0x1,%eax
   0x0000000000001296 <+70>: jne    0x12a2 <thread+82>
   0x0000000000001298 <+72>: mov    0x2dda(%rip),%eax        # 0x4078 <turn>
   0x000000000000129e <+78>: cmp    %eax,%ecx
   0x00000000000012a0 <+80>: je     0x1290 <thread+64>
   0x00000000000012a2 <+82>: movl   $0x0,(%rdx,%r8,4)
   0x00000000000012aa <+90>: sub    $0x1,%rdi
   0x00000000000012ae <+94>: jne    0x1280 <thread+48>
   0x00000000000012b0 <+96>: lea    0xf4240(%r9),%rax
   0x00000000000012b7 <+103>:     mov    %rax,0x2daa(%rip)        # 0x4068 <valor>
   0x00000000000012be <+110>:     xor    %eax,%eax
   0x00000000000012c0 <+112>:     ret
```

Prática 5: desabilitar reordenação do compilador

- Atualizar função para travar

```
void enter_region(int process)       /* process is 0 or 1 */
{
   int other;                        /* number of the other process */
   other = 1 - process;              /* the opposite of process */
   interested[process] = TRUE;       /* show that you are interested */
   turn = process;                   /* set flag */
   asm volatile ("" ::: "memory");
   while (turn == process && interested[other] == TRUE) /* null statement */ ;
}
```

Código gerado

```
Dump of assembler code for function thread:
   0x0000000000001250 <+0>:     endbr64
   0x0000000000001254 <+4>:     mov    $0x1,%esi
   0x0000000000001259 <+9>:     movslq %edi,%r8
   0x000000000000125c <+12>:    lea    0x2dbd(%rip),%rdx        # 0x4020 <interested>
   0x0000000000001263 <+19>:    sub    %edi,%esi
   0x0000000000001265 <+21>:    mov    %r8,%rcx
   0x0000000000001268 <+24>:    mov    $0xf4240,%edi
   0x000000000000126d <+29>:    movslq %esi,%rsi
   0x0000000000001270 <+32>:    movl   $0x1,(%rdx,%r8,4)
   0x0000000000001278 <+40>:    mov    %ecx,0x2daa(%rip)        # 0x4028 <turn>
   0x000000000000127e <+46>:    jmp    0x1288 <thread+56>
   0x0000000000001280 <+48>:    mov    (%rdx,%rsi,4),%eax
   0x0000000000001283 <+51>:    cmp    $0x1,%eax
   0x0000000000001286 <+54>:    jne    0x1292 <thread+66>
   0x0000000000001288 <+56>:    mov    0x2d9a(%rip),%eax        # 0x4028 <turn>
   0x000000000000128e <+62>:    cmp    %eax,%ecx
   0x0000000000001290 <+64>:    je     0x1280 <thread+48>
   0x0000000000001292 <+66>:    addq   $0x1,0x2d7e(%rip)        # 0x4018 <valor>
   0x000000000000129a <+74>:    movl   $0x0,(%rdx,%r8,4)
   0x00000000000012a2 <+82>:    sub    $0x1,%rdi
   0x00000000000012a6 <+86>:    jne    0x1270 <thread+32>
   0x00000000000012a8 <+88>:    xor    %eax,%eax
   0x00000000000012aa <+90>:    ret
```

Prática 6: desabilitar reordenação

- Atualizar função para travar

```
void enter_region(int process)        /* process is 0 or 1 */
{
   int other;                         /* number of the other process */
   other = 1 - process;               /* the opposite of process */
   interested[process] = TRUE;        /* show that you are interested */
   turn = process;                    /* set flag */
   atomic_thread_fence(memory_order_seq_cst);
   while (turn == process && interested[other] == TRUE) /* null statement */ ;
}
```

Código gerado

```
Dump of assembler code for function thread:
   0x0000000000001250 <+0>:     endbr64
   0x0000000000001254 <+4>:     mov    $0x1,%esi
   0x0000000000001259 <+9>:     movslq %edi,%r8
   0x000000000000125c <+12>:    lea    0x2dbd(%rip),%rdx        # 0x4020 <interested>
   0x0000000000001263 <+19>:    sub    %edi,%esi
   0x0000000000001265 <+21>:    mov    %r8,%rcx
   0x0000000000001268 <+24>:    mov    $0xf4240,%edi
   0x000000000000126d <+29>:    movslq %esi,%rsi
   0x0000000000001270 <+32>:    movl   $0x1,(%rdx,%r8,4)
   0x0000000000001278 <+40>:    mov    %ecx,0x2daa(%rip)        # 0x4028 <turn>
   0x000000000000127e <+46>:    lock orq $0x0,(%rsp)
   0x0000000000001284 <+52>:    jmp    0x1298 <thread+72>
   0x0000000000001286 <+54>:    cs nopw 0x0(%rax,%rax,1)
   0x0000000000001290 <+64>:    mov    (%rdx,%rsi,4),%eax
   0x0000000000001293 <+67>:    cmp    $0x1,%eax
   0x0000000000001296 <+70>:    jne    0x12a2 <thread+82>
   0x0000000000001298 <+72>:    mov    0x2d8a(%rip),%eax        # 0x4028 <turn>
   0x000000000000129e <+78>:    cmp    %eax,%ecx
   0x00000000000012a0 <+80>:    je     0x1290 <thread+64>
   0x00000000000012a2 <+82>:    addq   $0x1,0x2d6e(%rip)        # 0x4018 <valor>
   0x00000000000012aa <+90>:    movl   $0x0,(%rdx,%r8,4)
   0x00000000000012b2 <+98>:    sub    $0x1,%rdi
   0x00000000000012b6 <+102>:   jne    0x1270 <thread+32>
   0x00000000000012b8 <+104>:   xor    %eax,%eax
   0x00000000000012ba <+106>:   ret
```

Prática 7: rodar em máquina com ordenação fraca

- Por exemplo, aarch64 (arm64)
- Nestes casos, CPUs reordenam mais agressivamente

- Aumentar número de incrementos para 200.000.000

Código gerado para aarch64

```
Dump of assembler code for function thread:
   0x0000000100003e70 <+0>:     sxtw    x8, w0
   0x0000000100003e74 <+4>:     mov     w9, #0x1                        // #1
   0x0000000100003e78 <+8>:     sub     w10, w9, w0
   0x0000000100003e7c <+12>:    sxtw    x10, w10
   0x0000000100003e80 <+16>:    mov     w11, #0xe100                    // #57600
   0x0000000100003e84 <+20>:    movk    w11, #0x5f5, lsl #16
   0x0000000100003e88 <+24>:    adrp    x12, 0x100008000 <valor>
   0x0000000100003e8c <+28>:    add     x12, x12, #0x8
   0x0000000100003e90 <+32>:    adrp    x13, 0x100008000 <valor>
   0x0000000100003e94 <+36>:    add     x13, x13, #0x10
   0x0000000100003e98 <+40>:    adrp    x14, 0x100008000 <valor>
   0x0000000100003e9c <+44>:    b       0x100003eb4 <thread+68>
   0x0000000100003ea0 <+48>:    ldr     x15, [x14]
   0x0000000100003ea4 <+52>:    add     x15, x15, #0x1
   0x0000000100003ea8 <+56>:    str     x15, [x14]
   0x0000000100003eac <+60>:    str     wzr, [x12, x8, lsl #2]
   0x0000000100003eb0 <+64>:    cbz     x11, 0x100003ee0 <thread+112>
   0x0000000100003eb4 <+68>:    sub     x11, x11, #0x1
   0x0000000100003eb8 <+72>:    str     w9, [x12, x8, lsl #2]
   0x0000000100003ebc <+76>:    str     w0, [x13]
   0x0000000100003ec0 <+80>:    dmb     ish
   0x0000000100003ec4 <+84>:    ldr     w15, [x13]
   0x0000000100003ec8 <+88>:    cmp     w15, w0
   0x0000000100003ecc <+92>:    b.ne    0x100003ea0 <thread+48>  // b.any
   0x0000000100003ed0 <+96>:    ldr     w15, [x12, x10, lsl #2]
   0x0000000100003ed4 <+100>:   cmp     w15, #0x1
   0x0000000100003ed8 <+104>:   b.eq    0x100003ec4 <thread+84>  // b.none
   0x0000000100003edc <+108>:   b       0x100003ea0 <thread+48>
   0x0000000100003ee0 <+112>:   mov     x0, #0x0                        // #0
   0x0000000100003ee4 <+116>:   ret
```

Prática 8

- Adicionar mais barreiras

```
void enter_region(int process)       /* process is 0 or 1 */
{
   int other;                        /* number of the other process */
   other = 1 - process;              /* the opposite of process */
   interested[process] = TRUE;       /* show that you are interested */
   atomic_thread_fence(memory_order_seq_cst);
   turn = process;                   /* set flag */
   atomic_thread_fence(memory_order_seq_cst);
   while (turn == process && interested[other] == TRUE); /* null statement */
   atomic_thread_fence(memory_order_seq_cst);
}
```

Mais alguma oportunidade de reordenação?

- Em máquinas com ordenação fraca

Tipos de ordenação

- Relaxada (relaxed)
- Aquisição (acquire)
- Liberação (release)
- Consistência sequencial (sequential consistency)

Instruções atômicas

- Operações
  - Adição, Ou, E, etc.
- Atribuição
- Leitura
- Troca
  - Leitura e escrita
- Comparação e troca (compare and swap, CAS)
  - Também conhecida como compare and exchange, cmpxchg
  - Três argumentos: endereço, valor esperado, valor desejado
  - Atomicamente: lê valor atual e escreve desejado somente se atual for igual ao esperado
  - `atomic_compare_exchange_strong` em C11

Prática: modificar prática anterior para usar atômicos

```
uint64_t valor = 0;
atomic_bool trava = false;

void enter_region(void)
{
	bool v;
	do {
		v = false;
	} while (!atomic_compare_exchange_strong(&trava, &v, true));
}

void leave_region(void)
{
	atomic_store(&trava, false);
}
```

Problema prático

- O que acontece se a CPU for interrompida enquanto a trava está travada?

Solução

- Desabilitar interrupções enquanto esse tipo de trava estiver travada

Qual é o problema de desempenho?

- Quando há contenção
  - Laço "ocupado" (busy-loop) consome CPU
  - Mas não faz nada útil
  - Pode prevenir que outras threads rodem

Solução: pôr a thread para "dormir" enquanto espera

- Envolver o escalonador
- Assim, a thread esperando não consome CPU
- Quando uma thread sai da seção crítica
  - Libera a trava
  - Acorda uma thread que esteja esperando

- Resolve parcialmente o problema de interrupção também

Problema: como chamar o escalonador em modo usuário?

- Diferentemente do kernel, não há uma função direta

Solução

- Chamadas de sistema

- Exemplo do Linux: `futex`
- Exemplo do Windows: `Mutex`

Futexes

- Chamada de sistema futex
- No nosso caso, só 2 variantes são relevantes
  - FUTEX_WAIT: espera enquanto o valor no endereço *P* seja *x*
  - FUTEX_WAKE: acorda até *n* threads que estejam dormindo no endereço *P*

Prática: usar futexes

```
uint64_t valor = 0;
_Atomic uint32_t trava = 0;

void enter_region(void)
{
	uint32_t v;
	do {
		syscall(SYS_futex, &trava, FUTEX_WAIT, 1);
		v = atomic_exchange(&trava, 1);
	} while (v);
}

void leave_region(void)
{
	atomic_store(&trava, 0);
	syscall(SYS_futex, &trava, FUTEX_WAKE, 1);
}
```

Qual é o novo problema de desempenho?

- Transições de modo usuário para modo kernel são relativamente caras
- Essas transições teriam que acontecer sempre
  - Similar aos Mutexes do Windows

Solução: kernel envolvido só quando há contenção

- "Benaphores" no BeOS
  - Usa semáforo para envolver o kernel
- pthread_mutex no Linux
  - Usa um futex para envolver o kernel
- CRITICAL_SECTION no Windows
  - Usa um keyed-event para envolver o kernel
- Usam instruções atômicas
  - Para entrar e sair da seção crítica quando não há contenção
  - Fazem uma chamada de sistema quando há

Benoit Schillings

História

- Até o Windows XP, `EnterCriticalSection` podia falhar
  - Em caso de contenção, era necessário alocar um evento no kernel
- Mas o tipo de retorno da função é `void`
- Como indicar erro?
- Uma exceção estruturada (structured exception)

- Consequência: entrar numa seção crítica podia gerar uma exceção

- Solução: eventos chaveados (keyed events)

Prática: implementar `Mutex` usando futex

- Evitar envolver o kernel nos casos sem contenção
- Medir o desempenho do resultado

Solução

```
uint64_t valor = 0;
_Atomic uint32_t trava = 0;

void enter_region(void)
{
	uint32_t v = 0;
	if (atomic_compare_exchange_strong(&trava, &v, 1)) {
		return;
	}

	do {
		if (v == 2 || atomic_compare_exchange_strong(&trava, &v, 2)) {
			syscall(SYS_futex, &trava, FUTEX_WAIT, 2);
		}
		v = 0;
	} while (!atomic_compare_exchange_strong(&trava, &v, 2));
}

void leave_region(void)
{
	uint32_t v = atomic_fetch_sub(&trava, 1);
	if (v != 1) {
		atomic_store(&trava, 0);
		syscall(SYS_futex, &trava, FUTEX_WAKE, 1);
	}
}
```

Prática: relaxar o tipo de barreiras

- Usar barreiras mais permissivas
- Em C11:
  - memory_order_relaxed
  - memory_order_acquire
  - memory_order_release

- Medir o desempenho em máquinas x86-64 e arm64

Solução

```
uint64_t valor = 0;
_Atomic uint32_t trava = 0;

void enter_region(void)
{
	uint32_t v = 0;
	if (atomic_compare_exchange_strong_explicit(&trava, &v, 1,
                                                memory_order_acquire, memory_order_relaxed)) {
		return;
	}

	do {
		if (v == 2 || atomic_compare_exchange_weak_explicit(&trava, &v, 2,
                                                    memory_order_relaxed, memory_order_relaxed)) {
			syscall(SYS_futex, &trava, FUTEX_WAIT, 2);
		}
		v = 0;
	} while (!atomic_compare_exchange_weak_explicit(&trava, &v, 2,
                                                memory_order_acquire, memory_order_relaxed));
}

void leave_region(void)
{
	uint32_t v = atomic_fetch_sub_explicit(&trava, 1, memory_order_release);
	if (v != 1) {
		atomic_store_explicit(&trava, 0, memory_order_relaxed);
		syscall(SYS_futex, &trava, FUTEX_WAKE, 1);
	}
}
```

Resultado da execução

Em x86-64:

```
$ time ./mutex
Valor: 20000000

real    0m0.876s
user    0m1.116s
sys     0m0.577s
$ time ./mutex
Valor: 20000000

real    0m0.866s
user    0m1.103s
sys     0m0.585s
$ time ./mutex
Valor: 20000000

real    0m0.896s
user    0m1.115s
sys     0m0.599s
$ time ./mutex2
Valor: 20000000

real    0m0.877s
user    0m1.116s
sys     0m0.567s
$ time ./mutex2
Valor: 20000000

real    0m0.877s
user    0m1.100s
sys     0m0.557s
$ time ./mutex2
Valor: 20000000

real    0m0.848s
user    0m1.041s
```

Em arm64:

```
$ time ./mutex
Valor: 20000000

real    0m0.820s
user    0m1.347s
sys     0m0.189s
$ time ./mutex
Valor: 20000000

real    0m0.849s
user    0m1.399s
sys     0m0.213s
$ time ./mutex
Valor: 20000000

real    0m0.910s
user    0m1.578s
sys     0m0.158s
$ time ./mutex2
Valor: 20000000

real    0m0.524s
user    0m0.646s
sys     0m0.361s
$ time ./mutex2
Valor: 20000000

real    0m0.525s
user    0m0.687s
sys     0m0.326s
$ time ./mutex2
Valor: 20000000

real    0m0.499s
user    0m0.605s
```

API para Mutexes

```
struct mutex;

void inicializar(struct mutex *m);
void travar(struct mutex *m);
void destravar(struct mutex *m);
```

Problema dos produtores e consumidores

- Problema clássico de sincronização
- Componentes
  - Uma fila em memória compartilhada
  - Um conjunto de *produtores*
    - Produzem elementos e os colocam na fila
  - Um conjunto de *consumidores*
    - Retiram elementos da fila e os consomem
- Desafios
  - Corretude
  - Desempenho
  - Uso eficiente de recursos

Exemplo de produtores e consumidores

- A placa de rede e protocolos de rede
  - A placa de rede enfileira pacotes que chegam
    - Protocolos lidam com o mesmo
  - Protocolos enfileiram pacotes a serem enviados
    - A placa de rede os escreve
- Pipe
  - Como na syscall pipe
  - Fila de bytes, um produtor e um consumidor (que podem ser usados paralelamente)
- Logs
  - Vários componentes contribuem com linhas para os logs
  - As linhas são enfileiradas em memória
  - Um agente consome as linhas e as escreve para o disco

Uma implementação ingênua

```
#define TAMANHO 10
volatile int dados[TAMANHO];
volatile size_t inserir = 0;
volatile size_t remover = 0;

void *produtor(void *arg)
{
        int v;
        for (v = 1;; v++) {
                while (((inserir + 1) % TAMANHO) == remover);
                printf("Produzindo %d\n", v);
                dados[inserir] = v;
                inserir = (inserir + 1) % TAMANHO;
                usleep(500000);
        }

        return NULL;
}
```

Uma implementação ingênua (cont)

```
void *consumidor(void *arg)
{
	for (;;) {
		while (inserir == remover);
		printf("%zu: Consumindo %d\n", (size_t)arg, dados[remover]);
		remover = (remover + 1) % TAMANHO;
	}

	return NULL;
}
```

Prática: rodar um produtor e um consumidor

- Usar o código de criação de threads de outros exemplos
- Criar uma thread como produtor
- Criar uma thread como consumidor

- Rodar esse exemplo:
  - Funciona? Ou seja, os valores gerados são consumidos corretamente?
  - Qual é a utilização de CPU?

Prática: rodar um produtor e dois consumidores

- Adicionar uma thread consumidora ao código da prática anterior
- Qual é o resultado?

Prática: corrigir o problema de corretude com um mutex

- Modificar o exemplo anterior para usar um mutex
- É possível resolver o problema de corretude?
- Como está o desempenho?

Solução (produtor)

```
void *produtor(void *arg)
{
        int v;
        for (v = 1;; v++) {
                travar(&mutex_fila);
                while (((inserir + 1) % TAMANHO) == remover) {
                        destravar(&mutex_fila);
                        travar(&mutex_fila);
                }
                printf("Produzindo %d\n", v);
                dados[inserir] = v;
                inserir = (inserir + 1) % TAMANHO;
                destravar(&mutex_fila);

                usleep(500000);
        }
        return NULL;
}
```

Solução (consumidor)

```
void *consumidor(void *arg)
{
	for (;;) {
		travar(&mutex_fila);
		while (inserir == remover) {
			destravar(&mutex_fila);
			travar(&mutex_fila);
		}
		printf("%zu: Consumindo %d\n", (size_t)arg, dados[remover]);
		remover = (remover + 1) % TAMANHO;
		destravar(&mutex_fila);
	}
	return NULL;
}
```

Prática: reduzir o consumo de CPU

- Como fazer evitar os ciclos ocupados?

Semáforo

- Primitiva de sincronização
- Contador sincronizado
  - Não permite que o valor fique negativo
  - Trava a execução
- Possui duas operações (além de inicialização):
  - Incrementar
  - Decrementar

API para semáforos

```
struct semaforo;

void sem_inicializar(struct semaforo *s);
void sem_incrementar(struct semaforo *s);
void sem_decrementar(struct semaforo *s);
```

Prática: implementar um semáforo

- Atualizar exemplo para usar semáforo
- Começar a implementação usando o Mutex

Solução: tipos

```
struct esperando {
	struct mutex m;
	struct esperando *prox;
};

struct semaforo {
	struct mutex trava;
	size_t valor;
	struct esperando *cabeca;
	struct esperando *cauda;
};

void sem_inicializar(struct semaforo *s)
{
	inicializar(&s->trava);
	s->valor = 0;
	s->cabeca = NULL;
	s->cauda = NULL;
}
```

Solução: inicialização e incremento

```
void sem_incrementar(struct semaforo *s)
{
	struct esperando *esp;

	travar(&s->trava);
	esp = s->cabeca;
	if (esp != NULL) {
		s->cabeca = esp->prox;
		if (!s->cabeca) {
			s->cauda = NULL;
		}
	}

	s->valor++;
	destravar(&s->trava);

	if (esp != NULL) {
		destravar(&esp->m);
	}
}
```

Solução: decremento

```
void sem_decrementar(struct semaforo *s)
{
	struct esperando esp;

	for (;;) {
		travar(&s->trava);
		if (s->valor > 0) {
			s->valor--;
			destravar(&s->trava);
			return;
		}

		inicializar(&esp.m);
		travar(&esp.m);
		esp.prox = NULL;
		if (s->cauda) {
			s->cauda->prox = &esp;
		} else {
			s->cabeca = &esp;
		}
		s->cauda = &esp;

		destravar(&s->trava);

		travar(&esp.m);
	}
}
```

Prática: usar semáforo para reduzir CPU

- Atualizar exemplo para usar semáforo
- Parar acordar consumidores

Solução: produtor

```
void *produtor(void *arg)
{
	int v;
	for (v = 1;; v++) {
		travar(&mutex_fila);
		while (((inserir + 1) % TAMANHO) == remover) {
			destravar(&mutex_fila);
			travar(&mutex_fila);
		}
		printf("Produzindo %d\n", v);
		dados[inserir] = v;
		inserir = (inserir + 1) % TAMANHO;
		destravar(&mutex_fila);

		sem_incrementar(&sem_fila);
		usleep(500000);
	}

	return NULL;
}
```

Solução: consumidor

```
void *consumidor(void *arg)
{
	for (;;) {
		sem_decrementar(&sem_fila);
		travar(&mutex_fila);
		printf("%zu: Consumindo %d\n", (size_t)arg, dados[remover]);
		remover = (remover + 1) % TAMANHO;
		destravar(&mutex_fila);
	}

	return NULL;
}
```

Variáveis de condição (condition variables)

- Permitem que threads esperem
  - Até que certas condições sejam satisfeitas
  - Condições podem ser protegidas por mutex
- Outras threads precisam notificar quando condições puder ser satisfeita
- Duas operações
  - Esperar
  - Notificar

- Na prática, destrava mutex e dorme atomicamente

API para variáveis de condição

```
struct condvar;

void cv_inicializar(struct condvar *c);
void cv_acordar_um(struct condvar *c);
void cv_acordar_todos(struct condvar *c);
void cv_esperar(struct condvar *c, struct mutex *m);
```

Prática: implementar variáveis de condição com mutex

- Implementar a API anterior
- Usando a implementação existente de mutexes

Solução: tipos e inicialização

```
struct esperando {
	struct mutex m;
	struct esperando *prox;
};

struct condvar {
	struct mutex trava;
	struct esperando *cabeca;
	struct esperando *cauda;
};

void cv_inicializar(struct condvar *c)
{
	inicializar(&c->trava);
	c->cabeca = NULL;
	c->cauda = NULL;
}
```

Solução: espera

```
void cv_esperar(struct condvar *c, struct mutex *m)
{
        struct esperando esp;
        inicializar(&esp.m);
        travar(&esp.m);
        esp.prox = NULL;

        travar(&c->trava);
        if (c->cauda) {
                c->cauda->prox = &esp;
        } else {
                c->cabeca = &esp;
        }
        c->cauda = &esp;
        destravar(&c->trava);
        destravar(m);

        travar(&esp.m);
        travar(m);
}
```

Solução: acordar todos

```
void cv_acordar_todos(struct condvar *c)
{
	struct esperando *esp;

	travar(&c->trava);
	esp = c->cabeca;
	c->cabeca = NULL;
	c->cauda = NULL;
	destravar(&c->trava);

	while (esp != NULL) {
		struct esperando *prox = esp->prox;
		destravar(&esp->m);
		esp = prox;
	}
}
```

Solução: acordar um

```
void cv_acordar_um(struct condvar *c)
{
	struct esperando *esp;

	travar(&c->trava);
	esp = c->cabeca;
	if (esp != NULL) {
		c->cabeca = esp->prox;
		if (!c->cabeca) {
			c->cauda = NULL;
		}
	}
	destravar(&c->trava);

	if (esp != NULL) {
		destravar(&esp->m);
	}
}
```

Prática: usar variáveis de condição para reduzir CPU

- Atualizar exemplo anterior
- Utilizar variável de condição em vez de semáforo
- Tanto para produtor quanto consumidor

Solução: produtor

```
void *produtor(void *arg)
{
	int v;
	for (v = 1;; v++) {
		travar(&mutex_fila);
		while (((inserir + 1) % TAMANHO) == remover) {
			cv_esperar(&cv_fila, &mutex_fila);
		}
		printf("Produzindo %d\n", v);
		dados[inserir] = v;
		inserir = (inserir + 1) % TAMANHO;
		destravar(&mutex_fila);

		cv_acordar_todos(&cv_fila);
		usleep(500000);
	}

	return NULL;
}
```

Solução: consumidor

```
void *consumidor(void *arg)
{
	for (;;) {
		travar(&mutex_fila);
		while (inserir == remover) {
			cv_esperar(&cv_fila, &mutex_fila);
		}
		printf("%zu: Consumindo %d\n", (size_t)arg, dados[remover]);
		remover = (remover + 1) % TAMANHO;
		destravar(&mutex_fila);

		cv_acordar_todos(&cv_fila);
	}

	return NULL;
}
```