SO: Sincronização

Sistemas Operacionais
2017-1
Flavio Figueiredo (http://flaviovdf.github.io)

Mapa da Disciplina

- Fim da Seção de Processos
  - SOs geralmente são lecionados em três grandes temas
    - Processos
    - Memória
    - Arquivos
- Silberschatz Ed9
  - Sincronização: Capítulo 5
  - Deadlocks: Capítulo 7
- Tanenbaum (Sistemas Operacionais Modernos):
  - Fim do Capítulo 2
  - Parte do Capítulo 6 (Impasses)
- http://pages.cs.wisc.edu/~remzi/OSTEP/
  - Parte “Azul”: 25 até 34
- Depois disto vamos para segunda parte, memória

Lembrando de PThreads

- **int pthread_create(pthread_t *_thread_, const pthread_attr_t *_attr_, void *(*_start_routine_) (void *), void *_arg_);**
  - Cria nova Thread
  - Inicia a execução da Thread
  - Ponteiro para a Thread
  - Atríbutos
  - Ponteiro para função
  - Argumentos da Função

- **int pthread_join(pthread_t _thread_, void **_retval_);**
  - Espera a thread finalizar
  - Copia o valor de retorno para retval

Qual vai ser a saída do código seguinte

Exemplo

https://github.com/flaviovdf/SO-2017-1/blob/master/examples/threads/exemplo1.c

Exemplo

https://github.com/flaviovdf/SO-2017-1/blob/master/examples/threads/exemplo2.c

Exemplo

https://github.com/flaviovdf/SO-2017-1/blob/master/examples/threads/exemplo3.c

Respostas

1. Não Sei

Respostas

1. Não Sei

2. Não Sei

Respostas

1. Não Sei
2. Não Sei
3. Pior ainda, nem sei o sinal

Olhando no Nível de Instruções

`count++` em assembler:

```
a) MOV R1, $counter
b) INC R1
c) MOV $counter, R1
```

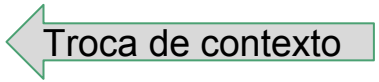
`count--` em assembler:

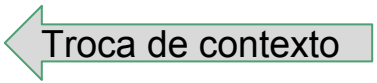
```
x) MOV R2, $counter
y) DEC R2
z) MOV $counter, R2
```

- Cada instrução é independente
- Interrupções podem ocorrer entre quaisquer duas instruções
  - Logo, trocas de contexto também podem ocorrer
- As sequências [a,b,c] e [x,y,z] podem ocorrer intercaladas

Falta de sincronização

```
a) MOV R1, $counter        R1 = 5
b) INC R1                  R1 = 6
x) MOV R2, $counter        R2 = 5
y) DEC R2                  R2 = 4
c) MOV $counter, R1        counter = R1 = 6
z) MOV $counter, R2        counter = R2 = 4
```





Condições de corrida

Dados e estruturas são acessados de forma concorrente

Resultado depende da ordem de execução dos processos

- Resultado é *indeterminado*

Necessidade de sincronização

Problema da seção crítica

Contexto

- Vários processos utilizam uma estrutura de dados compartilhada
- Cada processo tem um segmento de código onde a estrutura é acessada
- Processos executam a uma velocidade não nula (fazem progresso)
- O escalonamento e velocidade de execução são indeterminados

Garantir que apenas um processo acessa a estrutura por vez

Problema da seção crítica

Requisitos da solução:

- Exclusão mútua
  - Apenas um processo na seção crítica por vez
- Progresso garantido
  - Se nenhum processo está na seção crítica, qualquer processo que tente fazê-lo não pode ser detido indefinidamente
  - [Outra Forma de Escrever] Nenhum processo fora de sua região crítica pode bloquear outros
- Espera limitada
  - Se um processo deseja entrar na seção crítica, existe um limite no número de outros processos que entram antes dele

Controle de acesso à seção crítica

Considere dois processos, Pi e Pj

Processos podem compartilhar variáveis
para conseguir o controle de acesso

```
do {
    enter section
    // critical section
    leave section
    // remainder section
} while (1);
```

Como garantir as 3 condições?

1. Pensando em interrupções
2. Usando conceitos conhecidos while/for/if apenas

Para ajudar: Duas Threads i, j

Solução Bazooka

- Desabilitar interrupções
- Parar o SO todo menos o processo que vai utilizar região crítica

Solução Bazooka

- Desabilitar interrupções
- Parar o SO todo menos o processo que vai utilizar região crítica
- Meio extrema, mas ok, funciona.
  - Pelo menos para a exclusão mútua
- Perdemos tudo que aprendemos de escalonamento
  - Além de outros problemas como processos fazendo tarefas de SO
- Vamos pensar em algo melhor

Tentativa 1

```
int turn; // variável de controle, compartilhada, inicializada para i ou j

do {
    // id da thread na variável i
    while(turn != i);
    // critical section.
    turn = j;
    // remainder section
} while (1);
```

Tentativa 2

```
int queue[2] = {0, 0}; // variável de controle, compartilhada

do {
    queue[i] = 1;
    while(queue[j]);
    // critical section
    queue[i] = 0;
    // remainder section
} while (1);
```

Tentativa 3: Solução de Peterson

```
int queue[2] = {0, 0}; // variável de controle, compartilhada
int turn = i;
do {
    queue[i] = 1;
    turn = j;
    while(queue[j] && turn == j);
    // critical section
    queue[i] = 0;
    // remainder section
} while (1);
```

Solução de Peterson

- Funciona em sistemas de um processador apenas
- Hardwares modernos com vários cores não garantem sequência de operações
- Caches criam a oportunidade de discórdia
  - Cada CPU observa um valor diferente
- Instruções atômicas de hardware para resolver isto
  - e.g., TSL
- Mais complicada com n processos
  - Ainda mais se n é dinâmico


SCREW IT
I GIVE UP

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
boolean test_and_set(boolean *target) {
   boolean old = *target;
   *target = 1;
   return old;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
boolean test_and_set(boolean *target) {
   boolean old = *target;
   *target = 1;
   return old;
}
```

```
int lock = 0;
do {
    while(test_and_set(&lock));
    // critical section
    lock = 0;
    // remainder section
} while (1);
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Hardware de sincronização

Hardware provê operação atômica de leitura e escrita

```
void swap(boolean *a, boolean *b) {
   boolean tmp = *a;
   *a = *b;
   *b = tmp;
}
```

```
int lock = 0; // compartilhada
do {
    boolean key = true;
    while(key) swap(&lock, &key);
    // critical section
    lock = 0;
    // remainder section
} while (1);
```


SCREW IT
I GIVE UP

Exclusão mútua justa - cada processo passa a vez para o próximo na fila

```
int lock = 0; int n; //n tem o número de processos
do {
    waiting[i] = 1;
    while(waiting[i] && test_and_set(&lock));
    // critical section
    waiting[i] = 0;
    j = (i + 1) % n;
    while((j != i) && !waiting[j])
        j = (i + 1) % n;
    if (j == i) { lock = 0; }
    else { waiting[j] = 0; }
    // remainder section
 } while (1);
```

Ainda temos problemas?

Ainda temos problemas?

Todos os algoritmos com base em busy wait.

Sistema Operacional Ajuda

- Primitivas de block/wakeup (wait/notify em Java)
- Processo que chama block passa a esperar
- Processo que chama wakeup acorda um outro processo esperando (block)
  - Se existir, se não a vida continua

Semáforos

Primitivas de alto nível oferecidas pelo sistema operacional ou linguagem

Conceitualmente, semáforo é uma variável inteira acessível por duas operações atômicas

Funciona com qualquer número de processos. Conceito com busy wait abaixo

```
wait(int *s) {
   while(*s <= 0);
   (*s)--;
}
// acquire, down, lock
```

```
signal(int *s) {
   (*s)++;
}

// release, up, unlock
```

Implementação de semáforos

```
typedef struct {
    int value;
    struct process *list;
} sem_t;

void wait(sem_t *s) {
   s->value--;
   if(s->value < 0) {
      add(me, s->list);
      block();
   }
}
```

```
void signal(sem *s) {
   s->value++;
   if(s->value <= 0) {
      p = remove(s->list);
      wakeup(p);
   }
}
```


LIES
LIES EVERYWHERE
memegenerator.net

Ainda Precisamos de Busy Wait e/ou Desabilitar Interrupções

- block
- wakeup
- Locks com base em espera ocupada
  - Bem codificadas, poucas instruções

Solução

- Fazer um semáforo binário com spinlock ou desabilitando interrupções
  - bin_wait
  - bin_notify

- Construir outras primitivas com base neste

Semáforo de Tamanho N

```
typedef struct {
    int value;
    binsem_t wait;
    binsem_t mutex;
} sem_t;

void signal(sem *s) {
   bin_wait(&s->mutex);
   s->value++;
   if(s->value <= 0)
      bin_signal(&s->wait);
   bin_signal(&s->mutex);
}
```

```
void wait(sem_t *s) {
   bin_wait(&s->mutex);
   s->value--;
   if(s->value < 0) {
      bin_signal(&s->mutex);
      bin_wait(&s->wait);
   } else {
      bin_signal(&s->mutex);
   }
}
```

Deadlocks

```
// P1              // P2
wait(S);           wait(Q);
wait(Q);           wait(S);
…                  …
signal(S);         signal(Q);
signal(Q);         signal(S);
```

Deadlocks

```
// P1
wait(S);
wait(Q);
…
signal(S);
signal(Q);
```

```
// P2
wait(Q);
wait(S);
…
signal(Q);
signal(S);
```

```
P1: wait(S);
P2: wait(Q);
P1: wait(Q);
P2: wait(S);
// pwned
```

Primitivas

Mutex

- Semanticamente: Semáforo de tamanho 1
- [Geralmente] Apenas o processo/thread que adquire pode liberar
  - Erro em pthreads
  - ReentrantLock em Java lança exception
- lock/unlock
- Não precisa ser necessariamente implementado com semáforos
  - Mutex com espera ocupada: spinlocks

Como garantir que apenas a thread/processo que chamou lock pode chamar unlock?

Mutex

```
typedef struct {
  semaphore_t mutex = 1; //semáforo tamanho 1
  semaphore_t holderGuard = 1;
  int holder = -1;
} mutex_t;

void lock(mutex_t *mutex) {
  wait(&mutex_t->holderGuard);
  while (mutex_t->holder != -1) {
    signal(&mutex_t->holderGuard);
    wait(&mutex_t->mutex);
    wait(&mutex_t->holderGuard);
  }
  mutex_t->holder = myid();
  signal(&mutex_t->holderGuard);
}
```

```
int unlock(mutex_t *mutex) {
  wait(&mutex_t->holderGuard);
  if (mutex_t->holder != myid()) {
    signal(&mutex_t->holderGuard);
    return ERROR;
  }
  else {
    signal(&mutex_t->mutex);
    mutex_t->holder = -1;
    signal(&mutex_t->holderGuard);
  }
  return 0;
}
```

Spurious Wakeup

https://en.wikipedia.org/wiki/Spurious_wakeup

Chamadas Mutex em pthreads

```
//Cria mutex
int pthread_mutex_init(pthread_mutex_t *m, …)

//Trava mutex
int pthread_mutex_lock(pthread_mutex_t *m)

//Tenta travar e retorna imediatamente caso falhe
int pthread_mutex_trylock(pthread_mutex_t *m)

//Desbloqueia mutex
int pthread_mutex_unlock(pthread_mutex_t *m)
```

Sincronização com semáforos de tamanho 0

```
Processo Pi, task
inicializada com 0

// do something
signal(&task);
```

```
Processo Pj depende da
tarefa realizada por Pi

wait(&task);
// do something else
```

Monitores

- Solução mais alto nível
- Exemplo:
  - Synchronized em Java
- Apenas uma thread por vez entra em um método synchronized
  - Imagine um mutex sendo bloqueado no início do método
  - Desbloqueado no fim
- Primitivas de wait e notify
  - wait libera a CPU e entra em uma fila de espera
  - notify avisa para alguém que está esperando

Exemplo de Monitores em Java

```
public class SomeSharedResource {
     public synchronized void doSomething() {
          //Código aqui. Apenas 1 thread por vez na classe
     }
     public synchronized void doSomethingElse() {
          //Código aqui. Apenas 1 thread por vez na classe
     }
}
```

Exemplo de Monitores em Java

```
public class Exemplo2 {
     private final Object monitor = new Object();
     public void doSomething() {
          synchronized(monitor) {
               //Código aqui.
               monitor.wait(); //Libera recursos. Espera.
          }
     }
     public void doSomethingElse() {
          synchronized(object) {
               //Código aqui.
               monitor.notify(); //Libera recursos. Acorda.
          }
     }
}
```

Condições

- Similar a monitores
  - wait
  - notify

- Associados a um mutex

```
int pthread_cond_init(pthread_cond_t *cond, …);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *m);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Como implementar monitores?

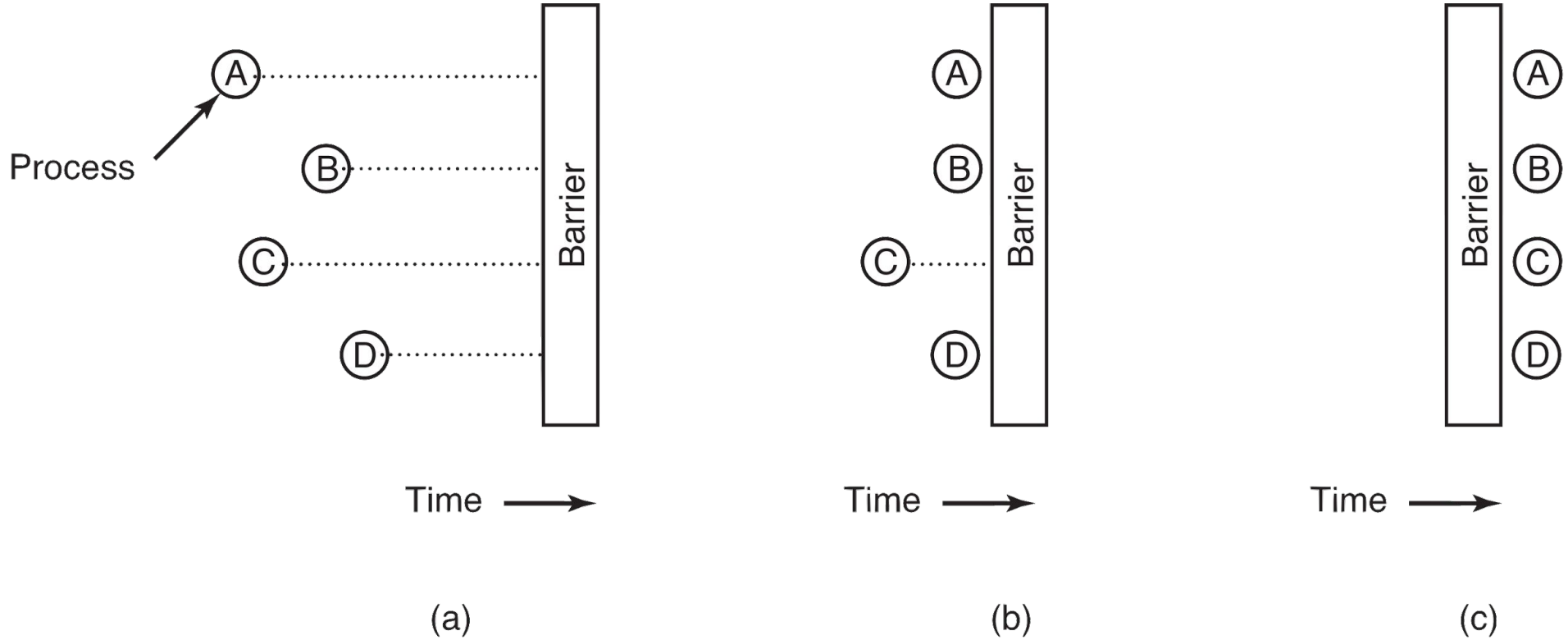
Monitores e Condições

`semaphore next = 0;`
`semaphore mutex = 1;`
`int next_count = 0;`
	`semaphore x_sem = 0;`
	`int x_count = 0;`
	`//x é uma condição`	

decorator de F	*x.wait()*	*x.signal() ou x.notify();*

`wait(mutex);`
`  …`
`  body of F;`
`  …`
`if (next_count > 0)`
`  signal(next);`
`else`
`  signal(mutex);`
	`x_count++;`
	`if (next_count > 0)`
	`  signal(next);`
	`else`
	`  signal(mutex);`
	`wait(x_sem);`
	`x_count--;`
		`if (x_count > 0) {`
		`  next_count++;`
		`  signal(x_sem);`
		`  wait(next);`
		`  next_count--;`
		`}`

Barreiras


Process
A
B
C
D
Barrier
Time
(a)
A
B
C
D
Barrier
Time
(b)
A
B
C
D
Barrier
Time
(c)

Como implementar barreiras?

Notas Finais

- Partindo de 1 primitiva (Semáforo) podemos construir as outras

- O semáforo tem que ser a base?
  - Poderia ser Mutex
    - [Pergunta de Entrevista] Como implementar um semáforo com mutex?
  - Razões históricas
  - Mais simples e flexível

Problemas Clássicos

Produtores Consumidores

- Fila Limitada de tamanho N
- Produtor trava quando a fila tem N elementos
- Consumidor trava quando a fila tem 0 elementos

Produtores Consumidores

```
semaphore mutex = 1
semaphore empty = BUFSZ;
semaphore full = 0;
```

Producer:

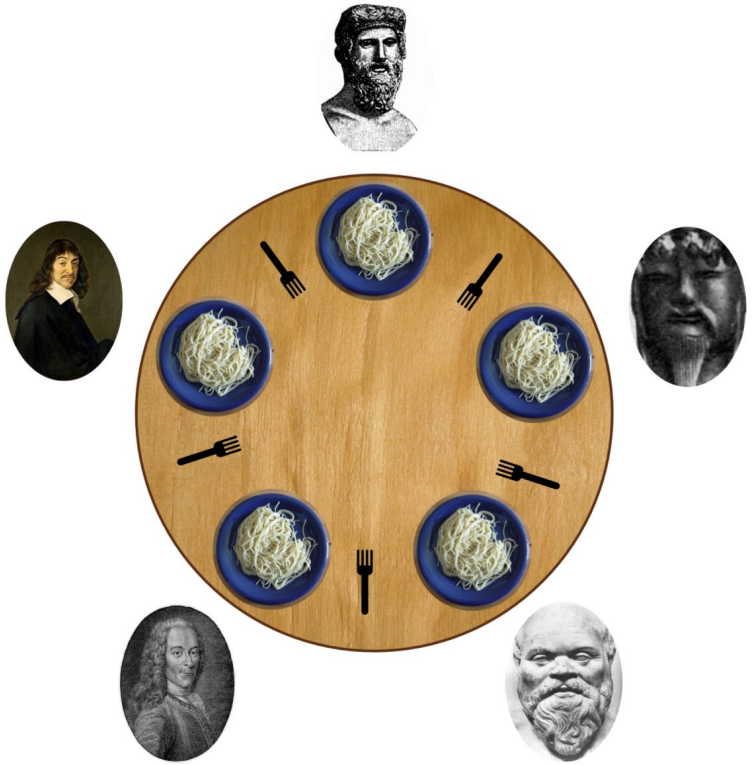
```
do {
  // generate item
  wait(empty);
  wait(mutex);
  // item → buffer
  signal(mutex);
  signal(full);
} while (1);
```

Consumer:

```
do {
  wait(full);
  wait(mutex);
  // item ← buffer
  signal(mutex);
  signal(empty);
  // process item
} while (1);
```

Jantar dos Filósofos

```
do {
     pickUpForks();
     eat();
     dropForks();
     think();
} while (1);
```



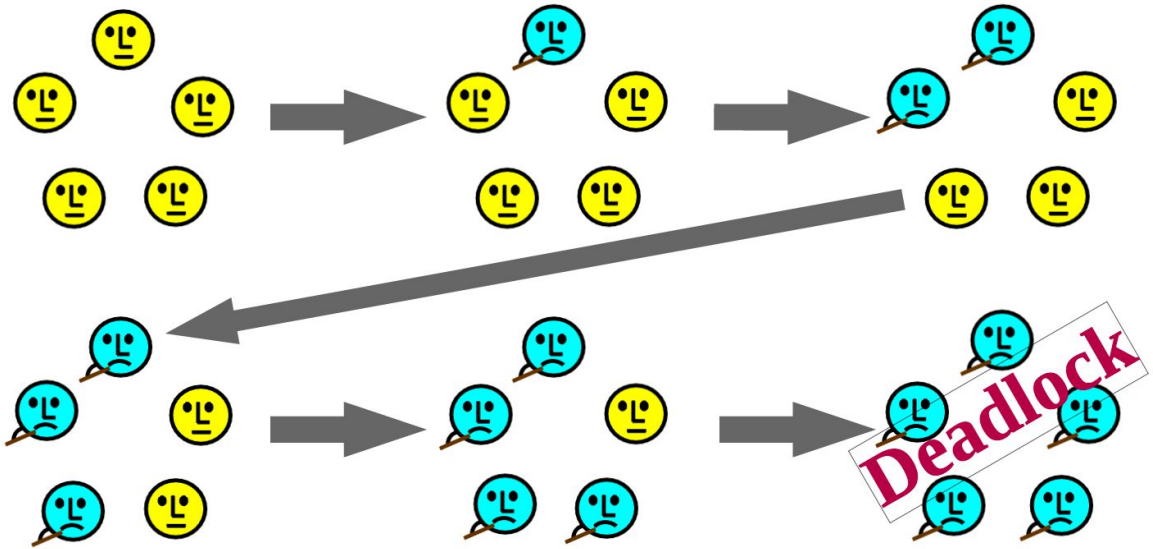
Uma Solução

Philosopher i:

```
do {
   // think
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
   // eat
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
} while (1);
```

Jantar dos Filósofos

- Para resolver precisamos tirar ciclos
  - Causam deadlocks



Jantar dos Filósofos

- Uma Solução
  - Pegar o garfo de menor ID
  - Vai ser o garfo da esquerda para todos menos o último filósofo
  - Sempre alguém vai conseguir comer dessa forma
  - [No fim] Removemos o ciclo de depêndencias dos garfos

Jantar dos Filósofos

- Uma Solução
  - Pegar o garfo de menor ID
  - Vai ser o garfo da esquerda para todos menos o último filósofo
  - Sempre alguém vai conseguir comer dessa forma
  - [No fim] Removemos o ciclo de depêndencias dos garfos

- Outra Solução
  - Garçon
  - Trava todos os garfos
  - Escolhe quem pode comer

Leitores e Escritores

- Imagine um espaço de memória que pode ser lido e escrito
  - um DB por exemplo
- Apenas 1 escritor pode editar a memória
- N-Leitores podem ler
  - Leituras não causam corrupções
- A leitura não pode ser corrompida por um escritor

Leitores Escritores

```
semaphore mutex = 1
semaphore writer = 1;
int rdcnt = 0;
```

Writer:

```
do {
  // other code
  wait(writer);
  // modify data
  signal(writer);
} while (1);
```

Reader:

```
do {
  wait(mutex);
  rdcnt++;
  if(rdcnt == 1) wait(writer);
  signal(mutex);
  // read data
  wait(mutex);
  rdcnt--;
  if(rdcnt == 0) signal(writer);
  signal(mutex);
  // other code
} while (1);
```