Syncronização de Processos

Marcelo Johann

Na aula anterior…

Threads

Plano da aula

1. **IPC**
2. **Indeterminismo**
3. **Race Condition**
4. **Seção Crítica**
5. **Mutexes e Semáforos**
6. **Resolvendo race conditions**
7. **Produtor e Consumidor**
8. **Como se faz com processos?**

IPC - Comunicação entre processos

- como passar informações entre processos?
- como fazer sincronismo por dependência de dados?
- como evitar interferências indesejáveis?

Compartilhamento

- Para comunicarem-se, processos compartilham:
  memória
  - ou com segmentos de dados compartilhados
  - ou como threads
  Arquivos, pipes, mensagens, etc…

Threads, ou Processos leves

São linhas de execução dentro de um mesmo processo

Cada processo

- um único segmento de código
- um único segmento de dados
- um único descritor de processo no kernel

Cada threads

- uma pilha própria
- PC,SP e registradores próprios

Exemplo: primeira.c

Indeterminismo na Concorrência

A ordem na qual as instruções são executadas não é determinística em um programa concorrente, mesmo em uma máquina monoprocessada.

Condições de Corrida

- "O resultado da computação depende da ordem em que as instruções são executadas"
- A ordem não é determinística com T.S.
- Condições de corrida devem ser evitadas!

Exemplo: race0.c

Seções Críticas

- "Parte do código que acessa dados compartilhados e os deixa em estados intermediários inconsistentes"
- Garantir exclusão mútua: somente um processo pode executar dentro da seção (ou região) crítica ao mesmo tempo

Mutexes e Semáforos

Lock() P()

Unlock() V()

Resolvendo Condições de Corrida

Com mutexes

Exemplo: race3mutex.c

Com semáforos

Exemplo: race2sema.c

Produtor/Consumidor com Buffer

Produtor:
Cria itens e deposita em um buffer

Consumidor
Retira itens do buffer e os consome

Buffer:
Array circular com tamanho limitado
Precisa sincronizar acesso, cheio, vazio

Pode haver ou não mais de um Prod. ou Cons.

Produtor e Consumidor em Buffer Limitado

```
#define N 100
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
void producer(void)
 {
 while (true)
  {
  produce_item(&item);
  down(&empty);
  down(&mutex);
  enter_item(item);
  up(&mutex);
  up(&full);
  }
 }
```

```
void consumer(void)
 {
 while (true)
  {
  down(&full);
  down(&mutex);
  remove_item(&item);
  up(&mutex);
  up(&empty);
  consume_item(item);
  }
 }
```

Como se faz com processos?

Semáforos POSIX
Jantar dos Filósofos e Deadlock
Código Thread-safe

Exemplos de códigos com semáforos e sincronização

Semáforos POSIX

- Named semaphores
- Kernel-persistent
- Need open/close, and init

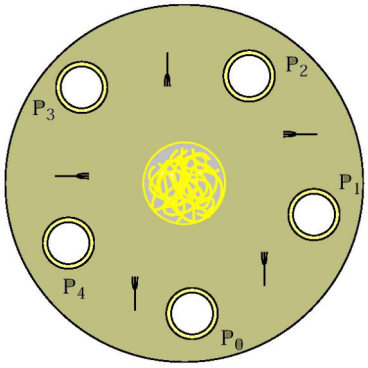
- testes

Como se faz com processos?

- Memória compartilhada

- Semáforos POSIX…

Problema dos filósofos




```
#define N 5
#define LEFT(i) (i+N-1)%N
#define RIGHT(i) (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];
sema_t mutex; // = 1
sema_t Sem[N]; // = 0

void philosopher(int i) {
  while(TRUE){
    think();
    take_forks(i);
    eat();
    put_forks(i);
    }
  }
void take_forks(int i) {
  sema_wait(&mutex);
  state[i] = HUNGRY;
  test(i);
  sema_post(&mutex);
  sema_wait(&Sem[i]);
  }
```

```
void put_forks(int i)
  {
  sema_wait(&mutex);
  state[i] = THINKING;
  test(LEFT);
  test(RIGHT);
  sema_post(&mutex);
  }

void test(int i)
  {
  if ( state[i] == HUNGRY &&
    state[LEFT(i)]!=EATING &&
    state[RIGHT(i)]!=EATING )
    {
    state[i] = EATING;
    sema_post(&Sem[i]);
    }
  }
```


Código Thread-Safe

- Distinção entre threads e funções
- Programas multi-thread
- Funções e bibliotecas reentrantes

You need to be thread-safe!

- Kernel reentrante