Avaliação Prática 1
Programação Concorrente 2025.1

Alunos:

- Guilherme Sousa Lopes (535869)
- Lucas Rodrigues Aragão (538390)

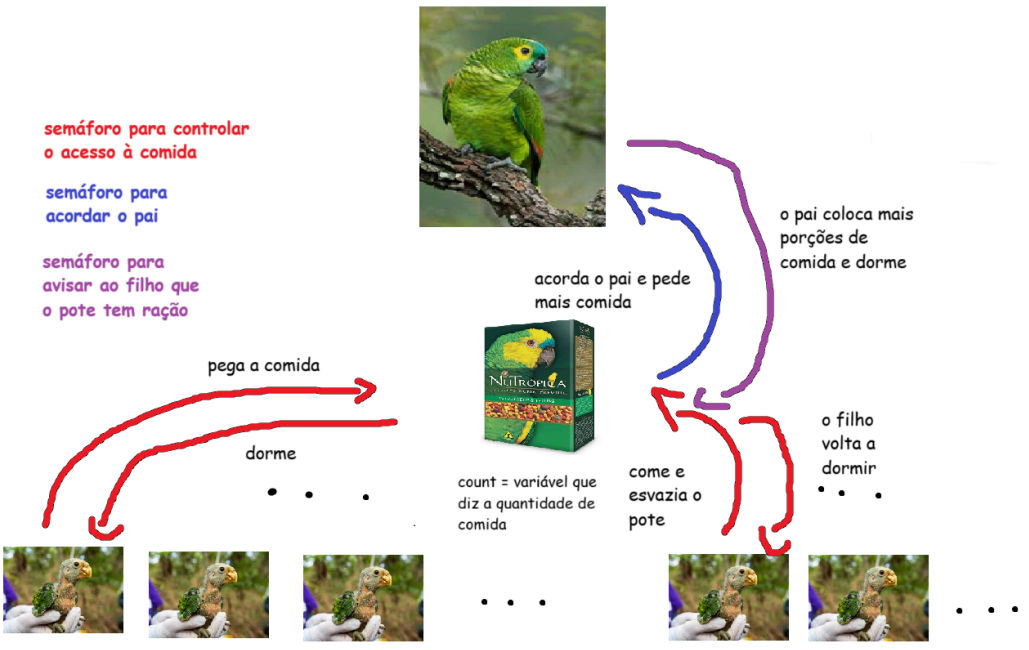
1. Introdução

Neste documento faremos a derivação de solução para o problema dos pássaros famintos, apresentada no exercício 4.35 no livro-texto da disciplina.

> 4.35 *The Hungry Birds.* Given are `n` baby birds and one parent bird. The baby birds eat out of a common dish that initially contains `F` portions of food. Each baby repeatedly eats one portion of food at a time, sleeps for a while, and then comes back to eat. When the dish becomes empty, the baby bird who empties the dish awakens the parent bird. The parent refills the dish with `F` portions, then waits for the dish to become empty again. This pattern repeats forever.
>
> Represent the birds as processes and develop code that simulates their actions. Use semaphores for synchronization.

Observando o problema percebemos alguns pontos importantes de cara.

1. O acesso à comida deve ser controlado para que consigamos controlar a quantidade de comida no pote.
2. O processo filho precisa de alguma maneira comunicar ao pai que a comida acabou.
3. E o processo pai também precisa falar ao filho que o pote está com comida novamente.



2. Solução de semáforos

- Eventos relevantes e contadores associados
    - A quantidade de ração no pote: ***food_count***
        - Toda vez que um filho pega uma ração:
            food_count = food_count - 1
        - Toda vez que o pai coloca ração novamente:
            food_count = F
    - Saber se alguém está acessando o pote de ração:
        food_access[1 … n]
        - Quando alguém está:
            food_access[i] = 1
        - Quando não está:
            food_access[i] = 0
    - Acordar o pai: ***call_parent***
        - Se o pai está acordado, ou seja, colocando comida:
            call_parent = 0
        - Se o pai está dormindo:
            call_parent = 1
    - Após colocar a comida o pai deve avisar ao filho: ***call_children***
        - Se o filho não está aguardando o chamado:
            call_children = 0
        - Se o filho está aguardando o chamado:
            call_children = 1

    - Contador de vezes que o pai foi chamado: ***count_call_parent***
    - Contador de vezes que os filhos pegaram comida: ***count_food_acess***

- Esboçar a solução inicial
    - Para esse esboço apenas colocamos as operações com os contadores e quando elas devem ser executadas.
    - Não existe qualquer controle de concorrência real para acesso de variáveis.
    - A solução inicial é vista a seguir:

esboço de solução

```
int food_count = F, count_food_access = 0, count_call_parent = 1;
int call_parent = 1, call_children = 1;
int food_access[1 ... n];

process children [1 ... n] {
  while (true){
      food_access[i] = 1;
      food_count = food_count - 1;
      if (food_count == 0) {
        call_parent = 0;
        // espera call_children
        call_children = 0;
        }
      }
      count_food_access = count_food_access + 1;
      food_access[i] = 0;
      sleep;
    }
  }
}
process parent {
  while (true){
    if (call_parent == 0) {
      food_count = F;
      call_children = 1;
      call_parent = 1;
      count_call_parent = count_call_parent + 1;
    }
  }
}
```

- Criação de invariantes
    - Somente um pode acessar o pote de comida
        - {FOOD: food_access[1] + … + food_access[n] <= 1}

    - A quantidade de ração que já foi comida não pode ser maior que o número de comida depositada no pote.
        - {RESTOCK: count_food_access <= (count_call_parent) * F}

    - A quantidade de ração no pote sempre deve estar entre 0 e F.
        - {PORTION: 0 <= food_count <= F}

    - Juntar as três invariantes numa geral para o problema.
        - *BIRDS: {FOOD and RESTOCK and PORTION}*

- Cálculo de condições no esboço

```
int food_count = F, count_food_access = 0, count_call_parent = 1,
int call_parent = 1, call_children = 1;
int food_access[1 ... n];

process children [1 ... n] {
  while (true){
      {food_access[0 … n]=0}
      food_access[i] = 1;
      {food_access[0 … i-1]=0 ^ food_access[i]=1 ^ food_access[i+1 … n]=0}

      {food_count > 0}
      food_count = food_count - 1;
      if (food_count == 0) {
        call_parent = 0;
        // espera call_children
        call_children = 0;

        }
      }
      {count_food_access < count_call_parent * F}
      count_food_access = count_food_access + 1;
      {count_food_access < count_call_parent * F}
      food_access[i] = 0;
      {food_count > 0}
      sleep;
    }
```

```
  }
}


process parent {
  while (true){
    if (call_parent == 0) {
      {food_count = 0}
      food_count = F;
      {food_count = F}
      call_children = 1;
      call_parent = 1;
      {count_food_access = count_call_parent * F}
      count_call_parent = count_call_parent + 1;
      {count_food_access <= count_call_parent * F}
    }
  }
}
```

- Fazer a solução de grossa granularidade usando await

```
int food_count = F, count_food_access = 0, count_call_parent = 1,
int call_parent = 1, call_children = 1;
int food_access[1 ... n];

process children [1 ... n] {
  while (true){

      {food_access[0 ... n]=0}

      <await (food_access[0 ... n] = 0) food_access[i] = 1; >

      {food_access[0 ... i-1]=0 ^ food_access[i]=1 ^ food_access[i+1 ... n]=0}
      {food_count > 0}
      food_count = food_count - 1;
      if (food_count == 0) {
        call_parent = 0;
        // espera call_children
        <await (call_children = 1)>
        call_children = 0;
```

```
        }
      }
      {count_food_access < count_call_parent * F}
      count_food_access = count_food_access + 1;
      {count_food_access < count_call_parent * F}
      food_access[i] = 0;
      {food_count > 0}
      sleep;
    }
  }
}


process parent {
  while (true){
    <await (call_parent = 0)>
      {food_count = 0}
      food_count = F;
      {food_count = F}
      call_children = 1;
      call_parent = 1;
      {count_food_access = count_call_parent * F}
      count_call_parent = count_call_parent + 1;
      {count_food_access <= count_call_parent * F}
  }
}
```

- Mudança de variável e retirada dos comandos await
  - **sem_food** = 1 - food_access[1] + … food_access[n]
  - Resultando em
    <await (food_sem > 0) food_sem = food_sem – 1; food_access[i] = 1;>
  - Com isso food_access virou uma variável auxiliar e podemos retirá-la pela regra das variáveis auxiliares. E food_sem é um semáforo para controlar o acesso à comida.
  - **sem _parent** = 1 - call_parent
  - Resultado em
    - sem_parent = 1, na chamada em children.
    - <await sem_parent = 1>, no início de parent.
  - **sem_children =** call_children.

- Podemos retirar os comandos await e substituí-los por comandos de P e V de semáforos resultando na solução final.

```
int food_count = F, time_sleep = 50;
sem sem_parent = 0, sem_children = 0;
sem sem_food = 1;

process children [1 ... n] {
  while (true){
      P(sem_food);
      food_count = food_count - 1;
      if (food_count == 0) {
        V(sem_parent);
        P(sem_children);
        }
      }
      V(sem_food);
      usleep(time_sleep);
    }
  }
}


process parent {
  while (true){
      P(sem_parent);
      food_count = F;
      V(sem_children);
  }
}
```

3. Solução usando monitores

```
monitor HungryBirds{
	int   count_food = F;
	cond children;
	cond parent;
	food_access = true;
	get_food(){
		arrive_time = clock();
		while (!food_access) {
			wait(children, arrive_time);
		}
		food_access = false;
	}

	release_food(){
		count_food--;
		if (count_food == 0){
			signal(parent);
			wait(children, 0) // urgent wait
			}
		food_access = true;
		signal(children);
	}

	restock_food(){
		wait(parent);
		count_food = F;
		signal(children)
	}
}
```

```
/ simulação dos processos
     process children[1 ... n]{
      // chamar a função de pegar comida
      while(true){
            HungryBirds.get_food();
            usleep(50);
      }
     }

     process parent(){
      while(true){
            HungryBirds.restock_food(); // fica sempre esperando o
     sinal para reabastecer
      }
     }
```

- A solução funciona?
    - Podemos aproveitar parte da invariante da solução de semáforos

    - Somente um pode acessar o pote de comida
        - {FOOD: food_access[1] + … + food_access[n] <= 1}

    - A quantidade de ração que já foi comida não pode ser maior que o número de comida depositada no pote.
        - {RESTOCK: count_food_access <= (count_call_parent) * F}

    - A quantidade de ração no pote sempre deve estar entre 0 e F.
        - {PORTION: 0 <= food_count <= F}

    - Juntar as três invariantes numa geral para o problema.
        - *BIRDS: {FOOD and RESTOCK and PORTION}*

- Vamos checar se as invariantes são mantidas no código.
- FOOD: Somente um pássaro acessa o pote de comida por vez.
    - A implementação final não usa o food_access como vetor, mas para fins de explicação utilizaremos aqui.

```
monitor HungryBirds{
	int   count_food = F;
	cond children;
	cond parent;
	food_access[1 ... n] = [0, ... , n];
	{FOOD: sum(food_access) = 0}
	get_food(i){

		arrive_time = clock();
		while (sum(food_access) > 0) {
			{FOOD: sum(food_access) = 1}
			wait(children, arrive_time);
			{FOOD: sum(food_access) = 0}
		}
		{FOOD: sum(food_access) = 0}
		food_access[i] = 1];
		{FOOD: sum(food_access) = 1}
	}

	release_food(i){
		count_food--;
		if (count_food == 0){
			signal(parent);
			wait(children, 0) // urgent wait
			}
		{FOOD: sum(food_access) = 1}
		food_access[i] = 0;
		{FOOD: sum(food_access) = 0}
		signal(children);
	}
```

Veja que, em momento algum, mais de um pássaro está acessando o pote de comida. Isso porque o valor só é aumentado quando

- Não há ninguém usando o pote, o que é feito durante a checagem do while, e por isso ele passa direto.
- Ele recebe o sinal de que pode comer,
  - Que é feito pelo chamado de release food, que tira o acesso da comida de um pássaro e sinaliza para o próximo que ele pode comer.

- RESTOCK: A quantidade de ração que já foi comida não pode ser maior que o número de comida depositada no pote.
    - Para isso, vamos ter que colocar duas variáveis auxiliares para fazer a checagem da condição. Note que count_call_parent já é inicializada com 1, pois “aquela comida tem que ter sido colocada por alguém”

```
monitor HungryBirds{
      int   count_food = F;
      int count_food_access = 0;
      int count_call_parent = 1;
      cond children;
      cond parent;
      food_access[1 ... n] = [0, ... , n];
      get_food(i){
            arrive_time = clock();
            {count_food_access < count_call_parent * F}
            while ((sum(food_access) > 0)) {
                  wait(children, arrive_time);
            }
            {count_food_access < count_call_parent * F}
            count_food_access ++;
            {count_food_access <= count_call_parent * F}
            food_access[i] = 1];
      }
      release_food(i){
            count_food--;
            if (count_food == 0){
                  {count_food_access == count_call_parent * F}
                  signal(parent);
                  {count_food_access < count_call_parent * F}
                  wait(children, 0) // urgent wait
                  }
            food_access[i] = 0;
            signal(children);
      }
      restock_food(){
            wait(parent);
            {count_food_access == count_call_parent * F}
            count_food = F;
            count_call_parent++;
            {count_food_access < count_call_parent * F}
            signal(children)
      }}
```

Note que a propriedade é mantida pois toda vez que os valores se igualam, a comida é reabastecida, logo o valor de $count\ call\ parent\ *\ F$ volta a ser maior que $count\ call\ food$. Tirando essas situações o valor de $call\ food\ access$ sempre é menor.

- PORTION: A quantidade de ração no pote sempre deve estar entre 0 e F.
    - Aqui é só fazer as checagens na variável $count\ food$.

```
monitor HungryBirds{
      int   count_food = F;
      {count_food == F}
      cond children;
      cond parent;
      food_access[1 ... n] = [0, ... , n];
      get_food(i){
            arrive_time = clock();
            {count_food =< F and count_food > 0}
            while ((sum(food_access) > 0)) {
                  wait(children, arrive_time);
            }
            food_access[i] = 1];
      }
      release_food(i){
            {count_food =< F and count_food > 0}
            count_food--;
            {count_food < F and count_food >= 0}
            if (count_food == 0){
                  {count_food == 0}
                  signal(parent);
                  {count_food  == F}
                  wait(children, 0) // urgent wait
                  }
            food_access[i] = 0;
            signal(children);
      }
      restock_food(){
            wait(parent);
            {count_food == 0}
            count_food = F;
            {count_food == F}
            signal(children)
      }}
```

Note que mais, a variável sempre está no intervalo [0, F], os momentos em que isso é ameaçado são:

- Quando o contador chega a 0. Mas isso é resolvido com uma chamada para o pai, que reabastece o pote de comida, atribuindo valor de F ao contador.
- Quando o contador é igual a F. Ocorreria o perigo do contador subir ainda mais, extrapolando o limite. Entretanto isso não acontece, pois o valor do contador só pode subir com chamadas ao pai, do contrário o contador só diminui.

Com isso, as três condições foram mantidas pelo programa, e assim, a solução mantém a invariante do problema.

BIRDS: {FOOD and RESTOCK and PORTION}