Tejas Abhay Kumthekar

MS CS Fall 2015

Advanced OS (P536) Assignment Baby bird

INDEX

A. Creating babybird xinu shell command -

babybird command takes in 3 arguments -
1. Number of baby birds (-b or --babies)
2. Number of worms fetchable by the parent bird (-f or --fetch)
3. Number of worms eatable by each baby bird (-e or --eat)

B. Input Handling with Validations and Graceful Degradation -

1. help

Validation Check -

Check if there are 2 arguments and second argument is either -h or --help

```
if (nargs == 2 && (strncmp(args[1], "--help", 7) == 0) || (strncmp(args[1], "-h",
3) == 0))
```

Displays help like below -

```
printf("Usage: %s\n\n", args[0]);
printf("Description:\n");
printf("\tThe Babybird Problem\n");
printf("\tbabybird -b/--babies <num babies> -f/--fetch <num worms fetch> -e/--eat <num worms eat>\n");
```

2. Argument Count Validations

a. Number of arguments should always be odd (only case in which number of arguments is even is during --help or -h, and we have handled help command already)

```
if(nargs%2 == 0){
		//command name + set of parameters is always odd
		fprintf(stderr, "Please enter correct arguments\n");
		return 1;
}
```

b. Number of arguments should not be greater than 7

```
if (nargs > 7) {
		fprintf(stderr, "%s: too many arguments\n", args[0]);
		fprintf(stderr, "Try '%s --help' for more information\n",
			args[0]);
		return 1;
}
```

3. Further validations

a. Number of baby worms

```
if(strncmp(args[i], "--babies", 9) == 0 || strncmp(args[i], "-b", 3) == 0){
	command = 1;
	if(atoi(args[i+1]) < 1){
		printf("At least 1 baby required\n");
	}
	else{
		babies = atoi(args[i+1]);
		printf("Number of babies = %d\n", babies);
	}
}
```

b. Number of to-fetch worms

```
if(strncmp(args[i], "--fetch", 8) == 0 || strncmp(args[i], "-f", 3) == 0){
	command = 2;
	if(atoi(args[i+1]) < 1){
		printf("At least 1 worm required to be fetched\n");
	}
	else{
		wormsFetch = atoi(args[i+1]);
		printf("Number of worms to fetch = %d\n", wormsFetch);
	}
}
```

c. Number of to-eat worms

```
if(strncmp(args[i], "--eat", 6) == 0 || strncmp(args[i], "-e", 3) == 0){
	command = 3;
	if(atoi(args[i+1]) < 1){
		printf("At least 1 worm required to eat\n");
	}
	else{
		wormsEat = atoi(args[i+1]);
		printf("Number of worms to eat = %d\n", wormsEat);
	}
}
```

4. Graceful Degradation to Defaults

We declared default values for babies, wormsFetch and wormsEat which will be used just in case we fail to get parameters. So, basically, we have a fault tolerant system in place for such scenarios.

C. Implementation

Creating the baby processes-

We create baby processes as specified in the input. We pass in the loop id as the baby process' id which we can use as an unique identifier for every baby process. We also need to know and track every baby process' consumed worms. So we also maintain a dedicated member for that.

Creating the parent process-

We pass in number of worms to be fetched in this process.

Maintaining worm pool-

We maintain a global integer variable wormsPool that stands for the number of worms available in the pool. This is used by baby processes as well as parent process. Baby process decrements variable in the worm pool as soon as it eats a worm (It also decrements its wormsEat member). Parent process, if asked to fetch worms, adds worms to the wormsPool.

Condition Variables and core logic-

Baby process continues to eat worm (one at a time and non-consecutively). If it finds that there are no worms in the wormsPool, it notifies parent process by signaling through the condition variable.

Parent process waits till it gets a signal through the condition variable. Once it gets that signal, it just fills up the wormsPool with worms.

D. Related Questions:

```java
class AOS_Test1 implements Lock {
  private int turn;
  private boolean busy = false;
  public void lock() {
    int me = ThreadID.get();
    do {
      do {
        turn = me;
      } while (busy);
        busy = true;
    } while (turn != me);
  }
  public void unlock() {
    busy = false;
  }
}
```

1. What is starvation? Is this protocol starvation-free?

Starvation is indefinite postponement of a process because it requires some resource before it can run, but that resource is never allocated to this process.
[Ref. https://www.cs.auckland.ac.nz/~alan/courses/os/book/6.Mana.13.starvation.pdf]

This protocol is starvation-free.

2. What is deadlock? Is this protocol deadlock-free?

Let’s assume 2 threads T1 and T2 and 2 resources R1 and R2. T1 is holding R1 while T2 is holding R2. If T1 requires R2 to finish its execution (and then leave a lock on R1) and if T2 requires R1 to finish its execution (and then to leave a lock on R2), then we say that there’s a deadlock.

This protocol is NOT deadlock-free, as there’s no synchronization for the shared resource.