Exercises 4

Task 4.1. Draw the Deadlock

Let us formalize the rice-eating problem, shown in the lecture 3, with the following program:

MEMORY	`chopsticks = 2`
Eater 1	`TakeStick(); TakeStick(); Eat(); ReleaseStick(); ReleaseStick();`
Eater 2	`TakeStick(); TakeStick(); Eat(); ReleaseStick(); ReleaseStick();`

If an `Eater` process is unable to get a chopstick, it waits for the next stick available. Note: there is no loop in the algorithm; the eaters eat once, and then the program terminates.

Draw a part of the state space diagram for this program that contains both starting and deadlock states. Highlight the deadlock state.

Task 4.2. Counting Sorter

Write a Promela program that performs a "counting sorting" of the array `A[N]`, which contains only numbers $1 \cdots P$. This is done as follows:

1. Create the global index variable `i = 0`, and the arrays `A[N]` and `R[N]`.
2. The main process with `_pid` 0 fills the array `A[N]` with values $1 \cdots P$ (you can hardcode some numbers) and wait until all other processes finish their job.
3. Each of P worker processes counts how many times its `_pid` appears in the array and stores the result in a local variable `p_count`.
4. Each worker process performs `p_count` times an assignment `R[i++] = _pid`. This operation must be performed in order: a process with higher `_pid` must wait until all processes with lower `_pid` values finish their work.
5. The main process prints the content of `R`.

Task 4.3. Competitive Coin Tossing

Use Promela to simulate a coin tossing game between two competing processes. Each process selects one of two random alternatives — "heads" or "tails", and stores its choice in a global variable. The third process tosses the coin (i.e., selects another random value) and prints the outcome. If one and only one process have guessed the outcome, it is declared a winner. If neither or both players are correct, the game is draw.

Hint: use `if` with two true conditions to perform a random choice.

Task 4.4. Raw Power Max

Write a Promela program that finds the maximum value in the given array as follows:

1. The main process fills the array `A[N]` with some values (you can just hardcode some numbers) and waits until all other processes finish their job.
2. Each of *N* worker processes checks whether the value of `A[_pid]` is higher than the currently identified global maximum value, and updates the maximum if necessary.
3. The main process prints the found maximum.