1. Consider a system with three chain smoker processes and one agent (of death) process. Each smoker continuously rolls a cigarette and then smokes it. But to roll and smoke a cigarette, the smoker needs three ingredients: tobacco, paper, and matches. One of the smoker processes has paper, another has tobacco, and the third has matches. The agent has an infinite supply of all three materials. The agent places two of the ingredients on the table. The chain smoker who has the remaining ingredient then makes and smokes a cigarette, signalling the agent upon completion. The agent then puts out another two of the three ingredients, and the cycle repeats (until all three processes die an unnatural death from lung cancer). Write pseudocode to synchronize the agent and the smokers.

2. Consider a set of concurrent processes $P_1, P_2, \ldots, P_7$ whose execution must obey the following precedence constraints.

- $P_1$ and $P_2$ cannot execute until $P_5$ has finished.
- $P_4$ cannot execute before $P_1$ and $P_3$ have finished.
- $P_3$ cannot execute until $P_4$ or $P_6$ have finished.
- $P_7$ can execute at any time unless $P_1$ has finished, in which case, it never gets to execute.

Program the execution of the task force using semaphores. Each process should appear as the original application-related code of the process with optional prologue and epilogue of semaphore-based operations. For example, represent each process as

```c
process P1()
{
    <prologue>

    body_of_P1();

    <epilogue>
}
```

The term “executing a process” may be interpreted as letting execution of that process proceed past the prologue of semaphore operations; “finishing a process” would refer to the process leaving its application-related code and executing its epilogue.

Show all global semaphore declarations and their initialization, and the complete prologues and epilogues of all seven processes. You may assume that each process is “straight-line code;” i.e., they don’t loop. You may find it useful to draw a picture illustrating the dependencies.
3. Consider a system of \( n \) similar processes, \( \{ P_0, P_1, \ldots, P_{n-1} \} \). Process \( P_i \) appears as:

```c
process P_i()
{
    initial phase_i();
    BARRIER;
    final phase_i();
}
```

The \texttt{BARRIER} construct blocks a process until all of the processes in the system are at the barrier. Then all of the processes are allowed to proceed. Barrier synchronization is especially useful for certain types of parallel computations which operate in phases (iterative numerical computations are a prime example). In the above system, all of the processes must complete their initial phases, and then all may begin on their final phases. No process can begin its final phase until all have completed their initial phase.

Implement \texttt{BARRIER} using semaphores as the synchronization mechanism. Show all semaphores and related data structures; state their initial values; and show the way that \texttt{BARRIER} is implemented for generic process \( P_i \). No extra processes may be created to support your implementation.

4. Problem 5.22 on pages 262–3. This problem involves the use of semaphores to coordinate a Santa process (or thread), nine (identical) reindeer processes, and any number of elf processes. Fun!