Tribhuvan University

Institute of Science and Technology

Central Department of Computer Science and Information Technology

Advanced Operating System-Peterson's Algorithm

Submitted by:
Rejina Dahal
Roll No: 15

Submitted to:
Dr. Binod Adhikari
CDCSIT

Date: March 25, 2025

Introduction to Peterson's Algorithm

Peterson's Algorithm is a classic solution to the critical section problem in process synchronization. It ensures mutual exclusion meaning only one process can access the critical section at a time and avoids race conditions. The algorithm uses two shared variables to manage the turn-taking mechanism between two processes ensuring that both processes follow a fair order of execution. It's simple and effective for solving synchronization issues in two-process scenarios [1][2].

Algorithm:

Algorithm for Process P_i:

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    // critical section

    flag[i] = false;

    // remainder section
} while (true);
```

Algorithm for Process P_j:

```
do {
    flag[j] = true;
    turn = i;
    while (flag[i] && turn == i);
    // critical section

    flag[j] = false;

    // remainder section
} while (true);
```

This algorithm ensures that only one process can enter the critical section at any given time while the other process waits its turn.It uses two simple variables one to indicate whose turn it is to access the critical section and another to show if a process is ready to enter [3][4].

Peterson's algorithm uses two key variables:

- **flag[i]**: Indicates if process i wants to enter the critical section.
- **turn**: Indicates whose turn it is to enter the critical section.

The algorithm operates as follows:

1. A process sets its $flag[i] = true$ and sets $turn = j$ (allowing the other process a chance to enter first).
2. The process waits while $flag[j] == true$ and $turn == j$ (indicating the other process is also interested in entering).
3. Once the other process either resets $flag[j]$ or it is not its turn, the process enters the critical section.
4. After execution, the process resets $flag[i] = false$ to allow the other process access.

This mechanism ensures orderly access to the critical section.

Mutual Exclusion

Mutual Exclusion is a property of process synchronization that states that “no two processes can exist in the critical section at any given point of time” [5]. Any process synchronization technique being used must satisfy the property of mutual exclusion, without which it would not be possible to get rid of a race condition. The requirement of mutual exclusion is that when process Pi is accessing a shared resource R1, another process should not be able to access resource R1 until process Pi has finished its operation with resource R1.

Proof:

- Consider two processes, P_i and P_j.
- If both processes want to enter the critical section, at least one must set $turn$ to the other process.
- A process can enter the critical section only if:
  - The other process has set $flag[j] = false$, or
  - $turn \neq i$, meaning it is its turn to enter.
- Since these conditions ensure that both processes cannot enter at the same time, mutual exclusion is guaranteed.

Conclusion: The algorithm guarantees mutual exclusion as one process must wait if the other is inside the critical section.

Indefinite postponement (Starvation)

Indefinite postponement, or starvation, occurs when a process is continuously denied access to a critical section.

Proof:

- The $turn$ variable ensures fairness by granting access alternately between processes P_i and P_j.
- Even if both processes P_i, P_j set their respective flag to true, only the process whose turn it (turn = j for P_j turn = i for P_j)will eventually proceeds.
- The other process must wait, but once the executing process completes and resets $flag[i] = false$ the waiting process can proceed.
- The turn variable ensures that neither process can indefinitely delay the other.
- Once a process exits its critical section, it must allow the waiting process to proceed.
- This guarantees that every process eventually gets a chance to execute.

Conclusion: The algorithm prevents indefinite postponement by guaranteeing that each process gets a fair opportunity to enter.

Progress (Deadlock Freedom)

Progress ensures that if no process is in the critical section, at least one can enter, preventing deadlock.Each process can only be blocked at the while if the other process wants to use the critical section ,And it is the other process's turn to use the critical section. If both of those conditions are true, then the other process will be allowed to enter the critical section, and upon exiting the critical section, will set to false, releasing process i. The shared variable turn assures that only one process at a time can be blocked, and the flag variable allows one process to release the other when exiting their critical section.

Proof:

- Deadlock occurs when two or more processes are waiting indefinitely for each other to release resources.
- In Peterson's algorithm, if no process is inside the critical section, then at least one of the two processes must be able to enter.
- The algorithm enforces progress because:
  - If P_i is waiting and P_j is also waiting, one of them must have its turn set, allowing it to proceed.
  - The `turn` variable ensures that once a process exits, the other gets a chance to execute.
  - Since each process resets `flag[i] = false` after execution, there is no indefinite blocking.

Conclusion: Peterson's algorithm prevents deadlock by ensuring that at least one process will always proceed.

References

[1] G. L. Peterson, "Myths about the mutual exclusion problem," *Inf. Process. Lett.*, vol. 12, no. 3, pp. 115–116, 1981.

[2] A. S. Tanenbaum, *Modern Operating Systems*, 4th ed. Pearson, 2014.

[3] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.

[4] "Mutual Exclusion in Synchronization," GeeksforGeeks, Mar. 20, 2020. https://www.geeksforgeeks.org/mutual-exclusion-in-synchronization/

[5] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, no. 9, p. 569, 1965.