Reference: Operating Systems
by
Harvey M. Deitel, Paul J. Deitel, David R. Choffnes

Consider a scenario in which two independent processes access shared memory location and modify it. An example could be `deposit()` and `withdraw()` processes in banking. Let us say a user A is trying to withdraw money from his account and his friend B is depositing money to the account. Both processes read a shared variable `balance` (that is the amount of money A has in his account).
Pseudocode shall look as follows:

withdraw(w_amt)	deposit(d_amt)
(W1) Read balance (W2) Add w_amt to balance (W3) Update balance	(D1) Read balance (D2) Add w_amt to balance (D3) Update balance

Please note that amount passed to withdraw and deposit can not be a shared variable as amount to withdraw can differ from amount to deposit. This is reflected by use of `w_amt` and `d_amt`.

In real systems, a process can undergo context switch after any instruction. Let us consider following execution instance (given `balance=1000, w_amt=200, d_amt=300` before execution starts)

Execution Instance 1:

W1 executes // `balance=1000`	
W2 executes	
Context switch occurs	
	D1 executes // `balance=1000`
	D2 executes
	D3 executes // `balance` updated to `1300`
	Context switch occurs
W3 executes // `balance` updated to `800`	

Outcome is incorrect as `balance=800` instead of correct value 1100.

Execution Instance 2

	D1 executes // `balance=1000`
	D2 executes
W1 executes // `balance=1000`	
W2 executes	
W3 executes // `balance` updated to `800`	
Context switch occurs	
	D3 executes // `balance` updated to `1300`
	Context switch occurs

Outcome is incorrect as `balance=1300` instead of correct value 1100.

Execution Instance 3:

W1 executes // `balance=1000`	
W2 executes	
W3 executes // `balance` updated to `800`	
	D1 executes // `balance=800`
	D2 executes
	D3 executes // `balance` updated to `1100`
	Context switch occurs

Outcome is correct as `balance=1100` (correct value).

Execution Instance 4:

	D1 executes // `balance=800` D2 executes D3 executes // `balance` updated to `1100` Context switch occurs
W1 executes // `balance=1000` W2 executes W3 executes // `balance` updated to `800`	

Outcome is correct as `balance=1100` (correct value).

What shall be outcome in following instances.

1. W1; D1; D2; W2; D3; W3
2. W1; W2; D1; W3; D2; D3;
3. W1; D1; W2; D2; W3; D3;
4. D1; W1; D2; W2; D3; W3;
5. D1; D2; W1; W3; D3; W3;

In above example, outcome can be 800, 1100 or 1300 depending on order of execution. Whenever outcome depends on the order of execution, it is termed race condition. The origin of race condition lies in sharing of a variable, which is being modified by more than one process that execute independent of each other in an asynchronous manner (no synchronization among processes).

Solution:
Ensure that access to shared resource is in mutual exclusion mode. At a given time, only one process is allowed to access the resource and unless the process is finished modifying (or processing) the resource, other processes are not allowed access.

Critical Section: Set of Instructions (of the process/thread code) that access a shared resource. A process can have multiple critical sections.

Solution to Critical Section Problem should satisfy:

1. Mutual Exclusion: At any time only one process should be in critical section.
2. No strict alteration: There should not be any order imposed on entry of processes to critical section.
3. Deadlock Free: The solution should not incur any deadlock.
4. No Infnite Postponement (or Bounded Wait): No process should be infintely postponed waiting for entry to critical section.

Solution #1

```
Initialization:
      int processNumber = 1;
      startProcesss(); /* Initialize and launch two processes */
```

Process P1:	Process P2:

`while(!done) {`
`  while(processNumber == 2)`
`  ;  //Wait`
`  /* enterMutualExclusion() */`
`  /* Critical section code */`
`  processNumber = 2;`
`  /* exitMutualExclusion() */`
`  /* Remaining Code */`
`}`
	`while(!done) {`
	`  while(processNumber == 1)`
	`  ;  //Wait`
	`  /* enterMutualExclusion() */`
	`  /* Critical section code */`
	`  processNumber = 1;`
	`  /* exitMutualExclusion() */`
	`  /* Remaining Code */`
	`}`

Problems: (1) Strict alteration. What if one of the process does not run?
(2) Busy waiting. CPU cycles are wasted.
(3) Lock synchronization problem: faster process constrained by speed of slower process.

Solution #2

```
Initialization:
    boolean P1_inside = false;
    boolean P2_inside = false;
    startProcesss(); /* initialize and launch both processes */
```

Process P1:	Process P2:

while(!done) {
while(P2_inside)
; // Wait
/* enterMutualExclusion */
P1_inside = true;
/* Critical section code */
P1_inside = false;
/* Remaininig Code */
}
	while(!done) {
	while(P1_inside)
	; // Wait
	/* enterMutualExclusion */
	P2_inside = true;
	/* Critical section code */
	P2_inside = false;
	/* Remaininig Code */
	}

Problems: Incorrect solution as does not even guarantee mutual exclusion problem.P1 checks P2_inside and finding it false exits the inner while loop. But before P1 could set P1_inside=true; context switch takes place. P2 gets the chance and finding P2_inside=false, exits loop and is ready for entry to critical section. So both processes can enter critical section violating mutual exclusion property.

Solution #3

```
Initialization:

    boolean P1_WantsToEnter = false;
    boolean P2_WantsToEnter = false;

    startProcesss(); /* initialize and launch both processes */
```

Process P1:	Process P2:

while(!done) {
P1_WantsToEnter = true;
/* enterMutualExclusion */
while(P2_WantsToEnter)
; // Wait
/* Critical section code */
P1_WantsToEnter = false;
/* Remaining Code */
}
	while(!done) {
	P2_WantsToEnter = true;
	/* enterMutualExclusion() */
	while(P1_WantsToEnter)
	; // Wait
	/* Critical section code */
	P2_WantsToEnter = false;
	/* Remaining Code */
	}

Problem: Deadlock can take place. P1 executes first statement and sets P1_wantsToEnter=true. Context switch happens. And now P2 runs and sets P2_WantsToEnter=true. Now both processes are deadlocked.

Solution #4

Initialization:

```
    boolean P1_WantsToEnter = false;
    boolean P2_WantsToEnter = false;
    startProcesss(); /* initialize and launch both processes */
```

Process P1:	Process P2:

while(!done) {
P1_WantsToEnter = true;
/* enterMutualExclusion() */
/* handle deadlock here */
while(P2_WantsToEnter) {
P1_WantsToEnter = false;
/* random wait */
P1_WantsToEnter = true;
}
/* critical section code */
P1_WantsToEnter = false;
/* Remaining Code */
}
	while(!done) {
	P2_WantsToEnter = true;
	/* enterMutualExclusion() */
	/* handle deadlock here */
	while(P1_WantsToEnter) {
	P2_WantsToEnter = false;
	/* random wait */
	P2_WantsToEnter = true;
	}
	/* critical section code */
	P2_WantsToEnter = false;
	/* Remaining Code */
	}

Problem: This version is able to break out of deadlocks by forcing each looping process to repeatedly set its flag to false for brief periods. Guarantees mutual exclusion, but introduces indefinite postponement.

Dekker's Algorithm (Correct Solution)

Initialization:

```
    int favoredProcess = 1;
    boolean P1_WantsToEnter = false;
    boolean P2_WantsToEnter = false;
    startProcesss(); /* initialize and launch both processes */
```

Process P1:	Process P2:

while(!done) {
P1_WantsToEnter = true;
/* enterMutualExclusion() */
/* handle deadlock here */
while(P2_WantsToEnter) {
if(favoredProcess == 2) {
P1_WantsToEnter = false;
while(favoredProcess == 2)
; /* busy wait */
P1_WantsToEnter = true;
}
}
/* Critical section code */
favoredProcess = 2;
P1_WantsToEnter = false;
/* Remaining Code */
}
	while(!done) {
	P2_WantsToEnter = true;
	/* enterMutualExclusion() */
	/* handle deadlock here */
	while(P1_WantsToEnter) {
	if(favoredProcess == 1) {
	P2_WantsToEnter = false;
	while(favoredProcess == 1)
	; /* busy wait */
	P2_WantsToEnter = true;
	}
	}
	/* Critical section code */
	favoredProcess = 1;
	P2_WantsToEnter = false;
	/* Remaining Code */
	}

Peterson’s Algorithm (Correct Solution)

- Simpler version of Dekker's algorithm v5, and less complicated.

Initialization:

```
int favoredProcess = 1;
boolean P1_WantsToEnter = false;
boolean P2_WantsToEnter = false;

startProcesss(); /* initialize and launch both processes */
```

Process P1:	Process P2:

while(!done) {
P1_WantsToEnter = true;
favoredProcess = 2;
while(P2_WantsToEnter &&
favoredProcess == 2)
; // wait
/* Critical section code */
P1_WantsToEnter = false;
/* Remaining Code */
}
	while(!done) {
	P2_WantsToEnter = true;
	favoredProcess = 1;
	while(P1_WantsToEnter &&
	favoredProcess == 1)
	; // wait
	/* Critical section code */
	P2_WantsToEnter = false;
	/* Remaining Code */
	}

Lamport's Bakery Algorithm (Correct Solution for N-process Mutual Exclusion)

- Suitable for multiprocessor systems and 'N' processes
- Each process assigned a ticket number which is used to determine the process granted access to critical section
- Process may obtain the same ticket number. There is a mechanism to settle ties so that only one process can access the critical section at a time. Ties can occur when a context switch happens during ticket dispensation.
- When a process exits its critical section, it sets its ticket = 0 to indicate that it is no longer executing in its critical section nor it is attempting to enter its critical section.
- Though all processes share the arrays choosing[] and ticket[], process Tx is the only one that can modify values in choosing[x] and ticket[x]. This prevents processes from reading inconsistent data.
- FCFS order until multiple processes possess the same ticket.

System:

```
/* Array to record which processes are taking a ticket */
boolean choosing[n];

/* Value of the ticket for each process initialized to 0 */
int tickets[n];

/* Initialize and launch all processes */
startProcesss();
```

Process Tx:

```
    x = ProcessNumber();
    while(!done) {
        /* Take a ticket */
        choosing[x] = true; /* begin ticket selection process */
        ticket[x] = maxValue(ticket)+1;
        choosing[x] = false; /* end ticket selection process */

        /* Wait until turn of current ticket comes */
        for(i=0; i < n; i++) {
            if(i == x)
                continue;

            /* Busy wait while Ti is choosing a ticket */
            while(choosing[i] != false)
                  ;

            /* Busy wait while current ticket value is lowest */
            while(ticket[i] != 0 && ticket[i] < ticket[x])
                  ;

            /* In case of tie, favor smaller process number */
            if(ticket[i] == ticket[x] && i < x) {
                /* Busy wait until Ti exits critical section*/
                while(ticket[i] != 0);
            }
        }

        /* Critical section code */
        ticket[x] = 0;
        /* exitMutualExclusion() */
    }
```