PART 4

PROCESS SYNCHRONIZATION - SOFTWARE

Dr. Soha S. Zaghloul

LAYOUT

- Producer-consumer Problem
- Race Condition
- Critical Section Problem
- Requirements to the Solution of the CS Problem
- Peterson's Solution

Introduction

Simultaneous access to shared data may result in data inconsistency.

Therefore, the execution of cooperating processes that share a logical address space must be organized (synchronized) in a way that ensures data integrity (or data consistency).

This problem is known as race condition:

A situation where several processes access and manipulate the same data simultaneously and the outcome of the execution depends on the particular order in which the access takes place.

PRODUCER-CONSUMER PROBLEM (1) – REVISITED

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
}
```

- Buffer is full → producer waits
- An item is produced
- Update the position of the *in* pointer

```
item next_consumed;
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        /* consume the item in next consumed */
}
```

- Buffer is empty → consumer waits
- An item is consumed
- Update the position of the *out* pointer

Producer-Consumer Problem (2)

Remember that both code segments of the previous slide are running simultaneously.

They are also sharing a variable; namely, counter.

The updated version of the previous slide is as follows:

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    counter--;
        /* consume the item in next consumed */
}
```

RACE CONDITION – SCENARIO (I)

Although the previous code segments run correctly when run sequentially, they might not run correctly if they are executed simultaneously because the order of execution is unpredictable.

Assume the current value of the variable counter = 5.

Consider the following scenario:

P

```
while (true) {
        /* produce an item in next produced */      1
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;                                   3
}
```

- At the start, counter= 5
- Segment (1) of the producer is executed
- Segment (2), the consumer, is assigned to the CPU before counter is incremented →

- counter = 4
- Segment (3) of the producer executes → counter = 5;
- In fact, there are 6 items in the buffer

C

```
while (true) {                                       2
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    counter--;
        /* consume the item in next consumed */
}
```

- counter= 5
- After the consumer executes, counter= 4
- In fact, there are 5 items in the buffer

RACE CONDITION – SCENARIO (2)

Now consider the following sequence:

At the start, the variable counter = 5.

P

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;        2
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

- counter= 5
- After the producer executes, counter= 6
- In fact, there are 5 items in the buffer

C

```
while (true) {
        while (counter == 0)                    1
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    counter--;                                  3
        /* consume the item in next consumed */
}
```

- Initially, counter= 5
- Segment (1) of the consumer executes
- Segment (2), the producer, is assigned to the CPU before counter is decremented →

- counter= 6
- Segment (3) of the consumer executes → counter = 5
- In fact, there are 4 items in the buffer.

RACE CONDITION – SCENARIO (3)

Now, consider the following sequence:

Assume the variable counter = 5.

P 1

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

- Initially, counter= 5
- Segment (1), the whole producer code, is executed → counter= 6 .
- In fact, there are 6 items in the buffer

C 2

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    counter--;
        /* consume the item in next consumed */
}
```

- counter= 6
- After Segment (2), the consumer, executes → counter= 5
- In fact, there are 5 items in the buffer

Race Condition – Scenario (4)

Now, consider the following sequence:

Assume the variable counter = 5.

P

2

```
while (true) {
        /* produce an item in next produced */
        while (counter == BUFFER_SIZE) ;
                /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

- counter= 4
- Segment 2, the producer code executes ➔ counter= 5
- In fact, there are 5 items in the buffer

C

1

```
while (true) {
        while (counter == 0)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
    counter--;
        /* consume the item in next consumed */
}
```

- Initially, counter= 5
- Segment (1), the whole consumer, executes ➔ counter= 4
- In fact, there are 4 items in the buffer

Race Condition – Low-Level Statements

The high-level language statements counter++ & counter-- are implemented in machine language on three steps each as follows:

counter++	counter--
R1 = counter R1 = R1 + 1 counter = R1	R2 = counter R2 = R2 – 1 counter = R2

Where R1 and R2 are the local register on the CPU.

The simultaneous execution of both statements is equivalent to a sequential execution in which the low-level statements are interleaved in some arbitrary order.

However, the order of high-level statements is preserved.

Example of the interleaving sequential low-level statements for the execution of counter++ and counter–- simultaneously is shown below. Assume that the initial value of counter is 5.

S0: producer executes	R1 = **counter**	{R1 = 5}
S1: producer executes	R1 = **R1 + 1**	{R1 = 6}
S2: consumer executes	R2 = **counter**	{R2 = 5}
S3: consumer executes	R2 = **R2 – 1**	{R2 = 4}
S4: producer executes	**counter = R1**	{counter = 6 }
S5: consumer executes	**counter = R2**	{counter = 4}

In fact, there are five items in the buffer (not 4) by the end of execution of S5.

In another run, we might have S5 executing before S4. Thus the incorrect value would be 6 instead of 5.

CRITICAL SECTION PROBLEM (1) - OVERVIEW

For two cooperative processes *Pi* and *Pj*, sharing a common variable *shared*, the code may be divided into two parts:

- The part of the code of *Pi* (or *Pj*), in which the common variable *shared* is used. This is called the critical section (*C*).
- The rest of the code of *Pi* and *Pj* that does not use any common variable. This is called the remainder section (R).

In order to avoid the race condition problem, *Pi* and *Pj* should be synchronized.

Synchronization is performed by prohibiting the simultaneous execution of the critical sections *Ci* and *Cj*, corresponding to the processes *Pi* and *Pj*.

Therefore, for a cooperative process *Pi*, before entering its own critical section *Ci*, it should ensure that no other cooperative process is using the common variable *shared*.

In addition, a process *Pi* should announce its exit from *Ci*; ie. *Pi* does not use any more the common variable *shared* so that it can be used by the other process *Pj*.

In other words, two announcements are necessary for each cooperative process *Pi* when using a common variable:

- 1) *Pi* makes an announcement before entering its critical section *Ci*.
  - This prohibits any other cooperative process *Pj* to enter *Cj*.
- 2) *Pi* makes an announcement as soon as it exits from its critical section *Ci*.
  - This allows any other cooperative process *Pj* to enter *Cj*.

In other words, *Pi* captures the common variable *shared* in the first announcement, and releases it in the second announcement.

Critical Section Problem (2) – Protocol

For a set of cooperative processes, the critical section is defined to be the segment of code in which the process may be changing shared variables, updating a table, writing a file, etc…

The rule to avoid the race condition is that when one process *Pi* is executing in its critical section *Ci*, no other cooperative process *Pj* is allowed to execute in its own critical section *Cj*.

In other words, no two cooperative processes are allowed to execute in their critical sections at the same time.

The critical section problem aims to design a protocol for the cooperative processes so that they fulfill the above rule.

The protocol of the critical section problem consists of four main parts:

1) The entry section

This is the code segment in which a cooperative process *Pi* requests permission to enter its critical section *Ci*.

2) The critical section

This is the code segment in which a process *Pi* uses shared variables.

3) The exit section

This is the code segment in which a cooperative process *Pi* releases the shared variable(s). This is done when *Pi* exits *Ci*.

4) The remainder section

This is the code segment in which a process *Pi* does not use any shared variables.

Critical Section Problem (3) – Pseudo-code

P_i

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Requirements to the Solution of the CS Problem

Many solutions have been proposed to the critical section (CS) problem.

An efficient solution to the CS problem should satisfy the following three requirements:

- Mutual exclusion
- Progress
- Bounded waiting

Requirements to the Solution of the CS Problem – Mutual Exclusion

An efficient solution should ensure mutual exclusion between cooperative processes.

If a process *Pi* is executing in its critical section *Ci*, then no other processes *Pj* can be executing in their corresponding critical section *Cj* at the same time.

In other words, one critical section is executing at a time.

Note that we may have the following scenario:

- A process *P1* shares a variable *shared1-2* with a process *P2*
- The process *P1* shares another variable *shared1-3* with a process *P3*
- In such case there are 4 critical sections *(P1, P2)*, *(P1, P3)*, *(P2, P1)*, and *(P3, P1)*
- The solution of the mutual exclusion problem does not allow *(P1, P2)* and *(P2, P1)* to execute simultaneously.
- Also, *(P1, P3)* and *(P3, P1)* are not allowed to execute simultaneously.
- However, *(P1, P2)* and *(P3, P1)* for example may execute simultaneously since they are working on different shared variables; namely, *shared1-2* and *shared1-3* respectively.

Requirements to the Solution of the CS Problem – Progress

An efficient solution should also ensure the progress of all cooperative processes.

If two (or more) cooperative processes *P1* and *P2* require permission to enter their respective critical sections sharing the same variable *shared* at a time *T1*, then two conditions should be satisfied to ensure the progress requirement:

- Only *P1* and *P2*, are given the right to decide which process (*P1* or *P2*) is given the permission first to enter its critical section. Other processes sharing the same variable but running in their Remainder Section are not involved in this decision.
- The negotiation decision should be made as soon as possible.

Consider the following example for three cooperative processes *P1*, *P2* and *P3* sharing the same variable:

- At some point of time *T0*, there is no process executing in its CS.
- At time *T1* , *P1* and *P2* wish to enter their CS.
- Only *P1* and *P2* negotiate to decide which of them enters the CS.
- Since *P3* is executing in its Remainder Section, it is not involved in the decision.
- The decision should be taken as soon as possible.
- Assume that the decision is given to the favor of *P1*, then *P2* waits.

REQUIREMENTS TO THE SOLUTION OF THE CS PROBLEM – BOUNDED WAITING

An efficient solution should also ensure bounded waiting for any cooperative process

By bounded waiting, it is meant that there is a limit (bound) in the number of times in which a process makes a request to enter its CS before it is given permission to enter its CS.

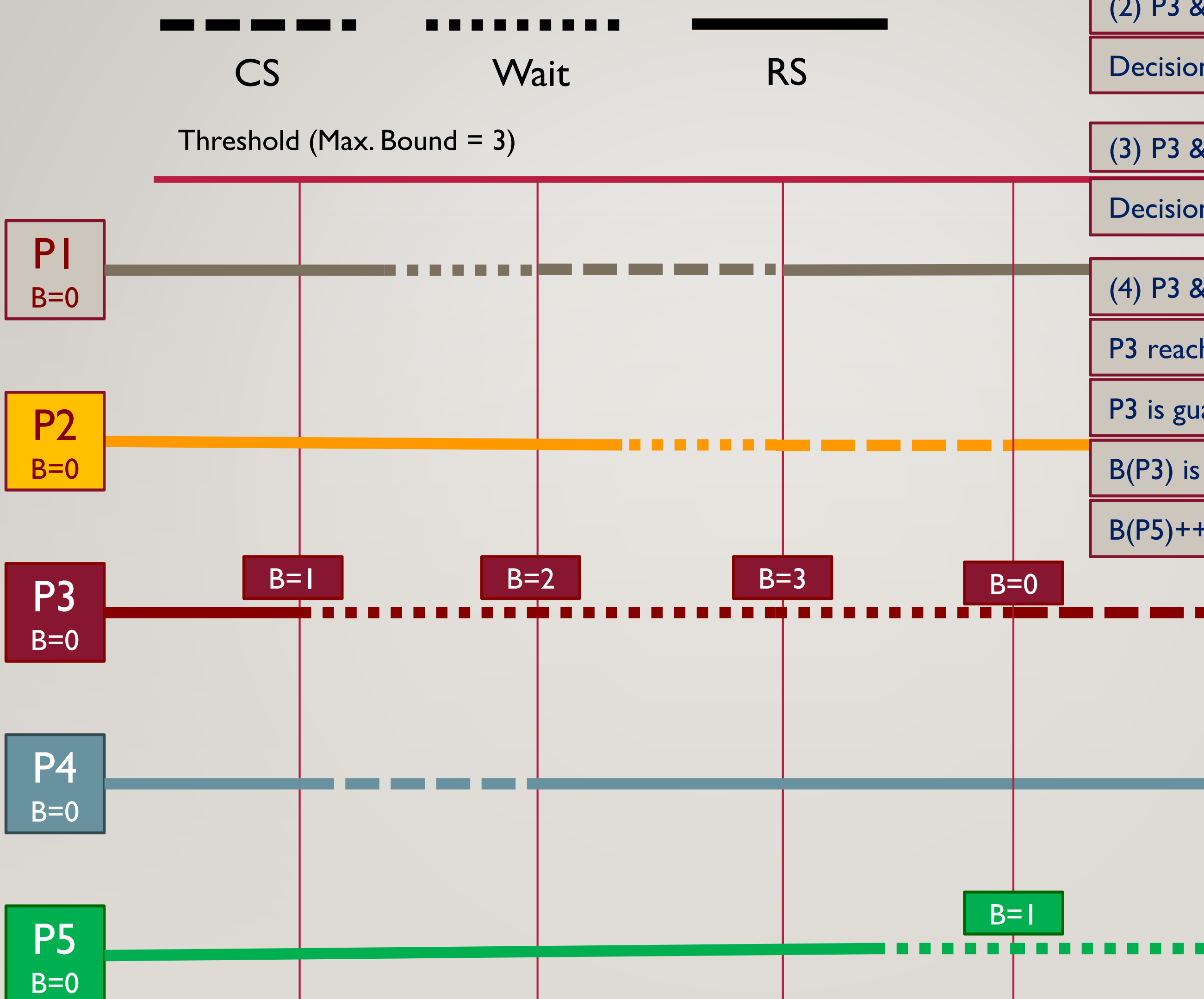
Consider the following example for five cooperative processes *P1*, *P2*, *P3*, *P4* and *P5* sharing the same variable. Assume that the *bound = 3 times*.

- Initially, *bound = 0*, for all processes.
- *P3* & *P4* wish to enter their CS. They negotiate, and the decision is in favor of *P4*
- *P4* enters its CS, and *bound* is incremented *(= 1)* for *P3*.
- Later, *P3* & *P1* wish to enter their CS. The decision is in favor of *P1*.
- *P1* enters its CS and the bound of *P3* is incremented (*bound=2*)
- Later, *P3* and *P2* wish to enter their CS. The decision is in favor of *P2*.
- *P2* enters its CS, and the bound of *P3* is incremented (*bound=3*).
- Later, *P3* and *P5* wish to enter their CS: since the bound of *P3* reached its maximum allowed limit, then *P3* enters its CS without negotiation.

In other words, no process should wait infinitely to enter its CS.

The figure in the next slide illustrates this example.

BOUNDED WAITING – EXAMPLE



(1) P3 & P4 compete for the CS

Decision in favor of P4, B(P3)++

(2) P3 & P1 compete for the CS

Decision in favor of P1, B(P3)++

(3) P3 & P2 compete for the CS

Decision in favor of P2, B(P3)++

(4) P3 & P5 compete for the CS

P3 reached threshold (maximum bound=3)

P3 is guaranteed to enter its CS

B(P3) is reset to 0

B(P5)++

Peterson's Solution

This is a classic software-based solution to the CS problem.

Peterson's solution is restricted to two cooperative processes only.

With two cooperative processes, *P0* and *P1*, Peterson's solution is designed so that they alternate between the execution of their Critical and Remainder Sections.

Peterson's solution requires the two processes to share two data items; namely:

- int turn;
- boolean flag[2];

PETERSON'S SOLUTION – THE TURN VARIABLE

The variable *turn* indicates whose turn it is to enter the CS: *P0* or *P1*.

Therefore, *turn* equals either to 0 or 1, since we deal with two processes only.

When *turn* = *0*, it is the turn of *P0* to enter its CS.
When *turn* = *1*, it is the turn of *P1* to enter its CS.

The next turn may also be expressed as *1 – i*, where *i* is the number of the current process.

It might happen that both processes try to enter their CS at the same time.

- In this case, both processes update the value of *turn* at the same time
- However, only one of these values will last and the other is overwritten
- We cannot predict which process will enter its CS first.

PETERSON'S SOLUTION – THE FLAG ARRAY

The *flag* array indicates if a process is ready to enter its critical section.

If *flag[0] = true*, then *P0* wishes to enter its CS.
If *flag[1] = true*, then *P1* wishes to enter its CS.

If both processes wish to enter their CS, then the variable *turn* decides which process will enter the CS.

Peterson's Solution – The code

The following figure depicts the code segments for two cooperative processes, *P0* and *P1*, that are running simultaneously.

Po	P1

`do {`
`flag[0] = true;`
`turn = 1;`
`while (flag[1] && turn = = 1);`
`critical section`
`flag[0] = false;`
`remainder section`
`} while (true);`
	`do {`
	`flag[1] = true;`
	`turn = 0;`
	`while (flag[0] && turn = = 0);`
	`critical section`
	`flag[1] = false;`
	`remainder section`
	`} while (true);`

Does Peterson's solution fulfill the requirements of the solution to the CS?

- The while loop ensures mutual exclusion.
- Progress is satisfied since turns are assigned to *P0* and *P1* alternatively.
- For the same reason, no process waits infinitely ➔ bounded waiting is satisfied.

Since the three requirements are fulfilled, then Peterson's solution is considered an efficient to the CS problem..

Although Peterson's solution is designed for two cooperative processes; the algorithm can be updated to fit three or more cooperative processes.