Operating Systems: Synchronization Tools and examples – Part I

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

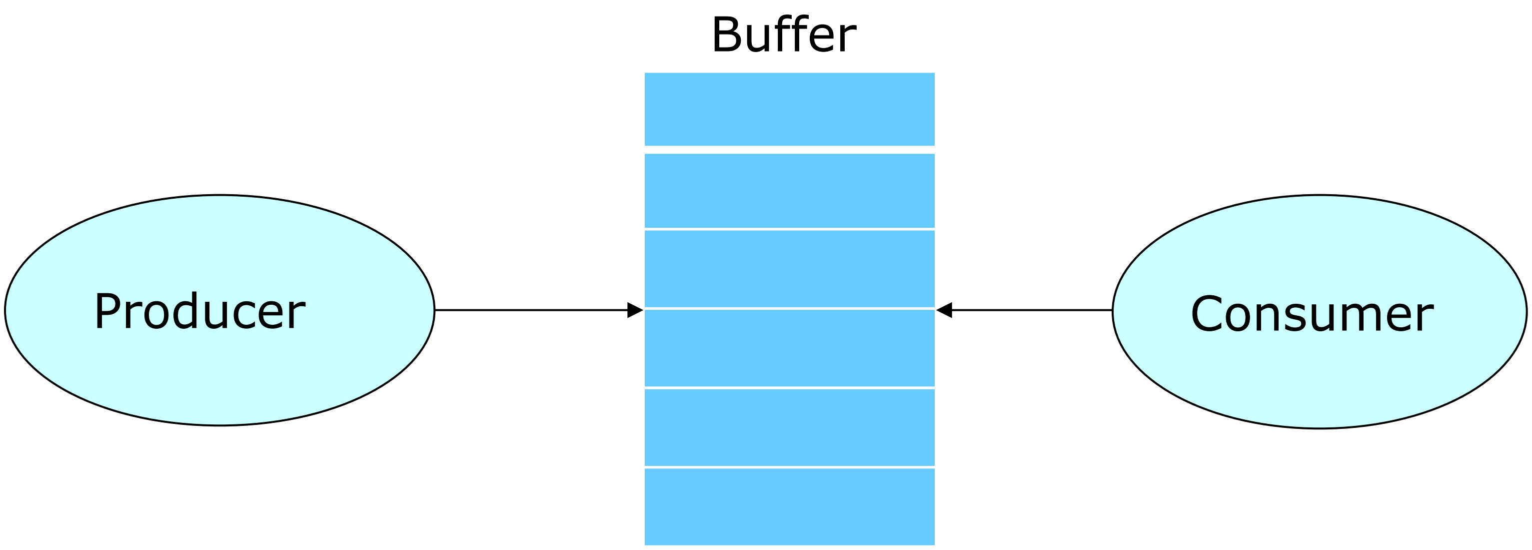
Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapters 6 & 7)

Background

- Processes (and threads) can execute concurrently.
- Concurrent access to shared data may result in data inconsistency.
- Data inconsistency is created by race conditions.
  - Race condition results when several threads try to access and modify the same data concurrently.

Producer Consumer Problem

- A producer produces information and places in the buffer (bounded buffer).
- Consumer consumes information from the buffer.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced, and the producer must wait if the buffer is full.



Producer Consumer Problem

How to keep track of the no.of full buffers?

- Maintain an integer **`counter`**, initially **`counter`** = 0.
- `counter` is incremented (`counter++`) by the producer after it produces a new buffer item.
- `counter` is decremented (`counter--`) by the consumer after it consumes a buffer.

Race Condition

`counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

`counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

Consider this execution interleaving with "counter = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}

S1: producer execute `register1 = register1 + 1` {register1 = 6}

S2: consumer execute `register2 = counter` {register2 = 5}

S3: consumer execute `register2 = register2 - 1` {register2 = 4}

S4: producer execute `counter = register1` {counter = 6 }

S5: consumer execute `counter = register2` {counter = 4}

Critical Section Problem

- Consider system of ***n* parallel** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **Critical Section (CS)** segment of code
  - Process may be changing shared variables, updating table, writing file, etc.

Goal

- When one process is in its critical section, no other process may be in its critical section
- Each process gets a turn (in bounded time) to execute its critical section.

Critical Section

- General structure of process P_i

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

- Each parallel process must ask permission to enter its critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**.

Three solution requirement to CS problem

1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If some processes wish to enter their CS, and no process is executing in its critical section, then
    1. only those processes not executing in their remainder section can participate in deciding which will enter its CS next.
    2. This selection cannot be postponed indefinitely

Solution requirement to CS Problem

3. Bounded Waiting - After a process has made a request to enter its CS, and before that request is granted

- A bound must exist on the number of times other processes are allowed to enter their critical sections.

Assumptions:

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the ***n*** processes

Peterson's Solution

- Good algorithmic description of solving the problem
- Two concurrent process solution (**P_0** and **P_1**)
- Unfortunately, this solution not guaranteed to work on modern hardware, due to vagaries of load and store operations.
- The two processes share two variables:
  - **`int turn;`**
  - **`Boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the CS
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = `*`true`*** implies that process **P_i** is ready!
  - Values of **`flag`** initialized to **`False`**.

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);
```

Entry Section

Exit Section

- Processes can execute this code arbitrarily many times.
- If `i=0` then `j=1` and visa versa.
- While loop in the entry section functions like a filter/trap.
  - The critical section code is executed only after the process exits the while loop.

Correctness of Peterson's Solution

- All three CS requirements are met.
- **Mutual exclusion** is preserved
  - If both $\mathbf{P_0}$ and $\mathbf{P_1}$ want to enter their CS this will result in both

    `flag[0] = T` and `flag[1] = T`
  - But `turn = 0` or `turn = 1.`
  - Since turn can be either 0 or 1 at any given time only one process can enter and thus execute in its CS.

Correctness of Peterson's Solution

- **Progress** is satisfied:
  - At any given time if both P_0 and P_1 want to enter the CS the decision which process enters its CS happens in finite time.
    - If both P_0 and P_1 want to enter this will result in both

      `flag[0] = T` and `flag[1] = T`
    - But `turn = 0` or `turn = 1`. The value of turn depends on the order of execution of the statement `turn = j;` in both processes P_0 and P_1
    - The value of **`turn`** will determine which process gets to enter its CS first.
    - This decision is made in finite time.
  - Additionally, if process P_0 (or P_1) is executing in its remainder CS, it does not affect the other process from entering its CS.
  - Therefore, progress is satisfied.

Correctness of Peterson's Solution

- **Bounded waiting** is satisfied
- Consider, P_0 and P_1 both want to enter their CS.
  - From Progress we know that one will enter its CS (lets say P_0 gets its turn)
  - When P_0 is in CS then `flag[0]=T` and `flag[1] = T` and `turn = 0`
  - When P_0 exits its CS then `flag[0]=F`, `flag[1] = T` and `turn = 0`
  - Thus, P_1 is the process that gets to enter its CS next.
  - Therefore, bounded waiting is satisfied, and a process will enter its critical section after at most **one entry** by *the other process* (bounded waiting).

Some pointers

- If it is possible for ALL processes to be stuck in the entry section, then
    - Progress requirement (condition 2) is violated.
- If a process cannot enter its CS arbitrarily many time even when other processes do not wish to enter their CS, then
    - Progress requirement (condition 1) is violated.
  - If strict order exists in which processes access their CS, then
    - Progress requirement (condition 1) is violated.
- If it is possible for a process to enter its CS infinitely many times without giving other processes that wish to enter their CS a turn, then
  - Bounded waiting requirement is violated.

Other Synchronization solutions

- Peterson's Algorithm drawbacks:
  - works for only two processes and
  - is not guaranteed to work on modern hardware.
- Systems with single processors – could **disable interrupts**
  - Currently running code would execute without preemption
  - Generally, too inefficient on a multiprocessor systems
    - Disabling interrupts on a multiprocessor can be time consuming and system efficiency is decreased.
- All solutions hereafter based on idea of **locking**
  - Protecting critical regions via locks

Critical-Section Handling in OS

- **Preemption**: The operating system removes the process running on the CPU after it has executed for the specified period of time and brings another program in to run on the CPU.
- Two approaches taken by the kernel to handle CS:
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Critical-Section Handling in OS

Why would you prefer a Preemptive kernel?

- As it is more responsive,
- Less risk for a kernel-mode process to use the processor for arbitrarily long period before relinquishing the processor to other waiting processes.

Solution to Critical-section Problem - For Multiple Processes: Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Mutex Locks

- OS designers build software tools to solve critical section problem
  - The simplest of tools is **mutex lock**
- Protect a critical section by
  - first acquiring the lock using **`acquire()`** and
  - releasing the lock using the **`release()`** operation.
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be atomic
  - Usually implemented via hardware atomic instructions

acquire() and release() operations

- Shared boolean `available`.
- `available = True` -> lock is available
- `available = False` -> lock is unavailable.

Acquire Operation:

```
acquire() {
    while (!available);
        /* busy wait */
    available = false;
}
```

Release Operation:

```
release() {

    available = true;

}
```

Solution using Mutex lock

```
do {
     acquire(); /*acquire lock*/
          critical section
     release(); /*release lock*/
          remainder section
 } while (true);
```

Issues with Hardware Instructions and Mutex Locks

- The issue with mutex locks is ***busy waiting***
  - Busy waiting consumes CPU cycles without doing any useful work.
  - This type of lock is known as a ***spinlock***, because the lock just sits there and spins while it waits.
  - Although busy waiting wastes CPU cycles, it saves time by not invoking context switches
  - Useful in multi-processing systems when the wait time is expected to be “short” - One thread spins on one processor while another completes their critical section on another processor.

Pthreads Synchronization

- Pthreads API is available for programmers at the user level and is not part of any particular kernel.
  - Provides mutex locks, condition variables, and read–write locks for thread synchronization.

`pthread_mutex_t` data type for mutex locks.

- mutex initialized with **`pthread_mutex_init()`** function.

```
pthread_mutex_lock();
pthread_mutex_unlock();
pthread_mutex_destroy();
```

Pthread Mutex Example

```
/*Declaring mutex*/

pthread_mutex_t mutex;

/*Initialize mutex returns 0

if mutex initialized with no errors.*/

if (pthread_mutex_init(&mutex, NULL) !=0){

     printf("Error in initializing mutex \n");

}

/*Acquire mutex lock*/

pthread_mutex_lock(&mutex);

/*Critical Section*/

/*Release mutex locks*/

pthread_mutex_unlock(&mutex);
```

`Null` indicates default mutex attributes passed.

Question

Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock with waiting, where the waiting processes sleep while waiting for the lock to become available:

a. The lock is to be held for a short duration.

b. The lock is to be held for a long duration

c. A thread may be put to sleep while holding the lock