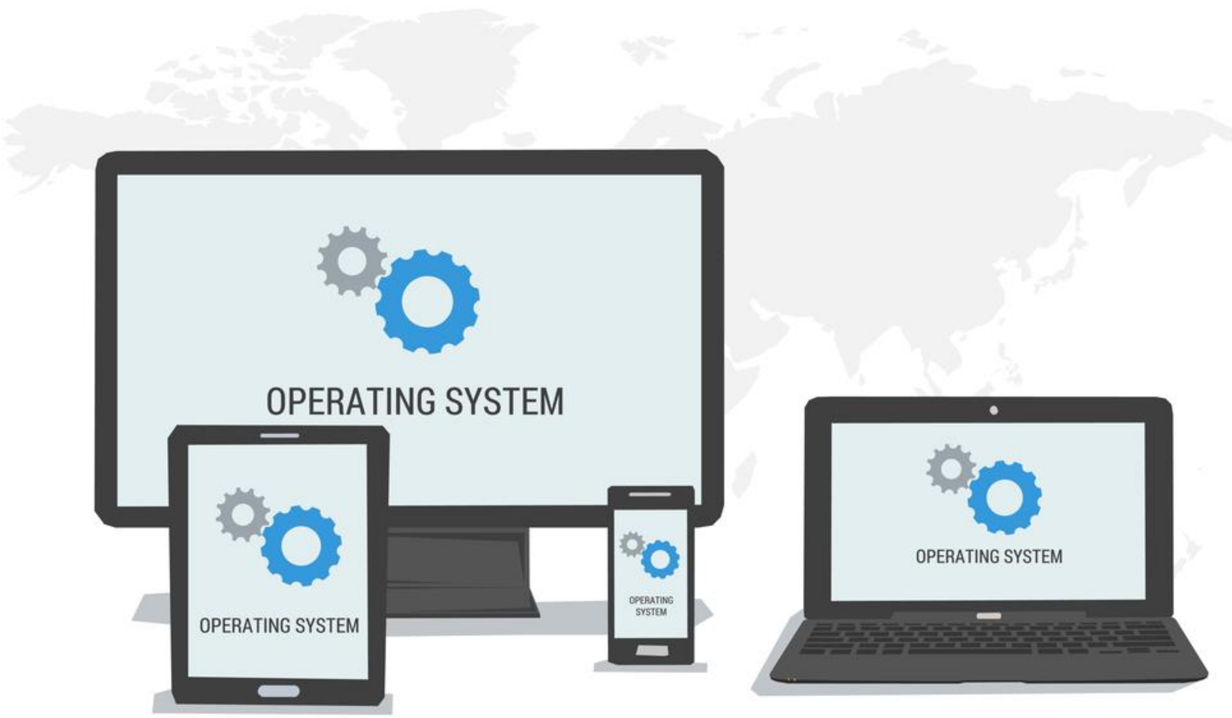

Process Synchronization& Semaphore

By : Mohamed Gamal Maklad

❖ Processes can be classified into:

- **Independent** Process: the execution of **one process has no impact on the execution of the others.**
- **Cooperative** Process: One **process's execution has an effect on the execution of other processes.**

❖ Processes can execute **concurrently or in parallel.**

❖ **The CPU scheduler is used to rapidly switches** between processes to provide concurrent execution

❖ **multi-core** processor **allows parallel** execution of different processes

❖ Concurrent access **to shared data may result** in **data inconsistency**

❖ **Process Synchronization** is the task of coordinating the execution of processes in such a way that **no two processes can have access to the same shared data and resources.**

❖ Race Condition:

- Several **processes access and manipulate the same data** concurrently may lead to incorrect shared data.
- The outcome of the **execution depends on the particular order** in which the access takes place
- **processes are "racing" to access/change the data**

❖ **Producer-Consumer problem:**

- job of the **Producer is to generate the data**, put it into the buffer
- job of the **Consumer is to consume(Delete) the data** from the buffer
- producers and consumers **share the same memory** buffer that **is of fixed-size (bounded buffer)**

❖ **To solve the Producer-Consumer problem:**

- We can do so by having an integer **counter that keeps track of the number of full buffers**
- counter is set to 0 It is **incremented by the producer** after it produces a new buffer and is **decremented by the consumer** after it consumes a buffer.

```
while (true) {
       /* produce an item in next produced */

       while (counter == BUFFER_SIZE) ;
              /* do nothing */
       buffer[in] = next_produced;
       in = (in + 1) % BUFFER_SIZE;
       counter++;
}
```

هو هنا بيتأكد الأول ان bufferلسه فاضي عن طريه
انه يشوف هل count==buffer لو الاتنين قد بعض
كده biffer مليان فمش هيعمل حاجه طب لو الشرط
متحققش هيضيف العنصر في المكان اللي عليه الدور
وبعد كده بيحرك in في المكان اللي بعده وبعدين يزود
count بواحد لأننا اضفنا عنصر

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

هنا نفس الحكايه بس الفرق ان هنا بيتأكد هل الcounter فاضي ولأ لا فو فاضي
=0

فمش هيعمل حاجه طب لوفي عناصر هيحذف ب اندكسout وبعد كده يتنقل ع العنصر اللي عليه الدور ويقلل count بواحد

- ❖ **<u>counter++ could be implemented as</u>**
  - ➢ `register1 = counter`
  - ➢ `register1 = register1 + 1`
  - ➢ `counter = register`
- ❖ **<u>counter—— could be implemented as</u>**
  - ➢ register2 = counter
  - ➢ register2 = register2 - 1
  - ➢ counter = register2
- ❖ **<u>Critical Section Problem</u>**:
  - ➢ Each **process has critical section segment** of code
  - ➢ Process may be changing common variables, updating table, writing file, etc
  - ➢ **<u>When one process in critical section, no other may be in its critical section</u>**
  - ➢ **Each process must ask permission** to **<u>enter critical</u>** section in **entry section**, may follow critical section with **exit section**, then **remainder section**

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

- ❖ **<u>Solution to Critical-Section Problem</u>**:
  - ➢ **<u>Mutual Exclusion</u>** - If process Pi is executing in its critical section, then **no other processes can be executing in their critical sections**

    لو في عمليه بتتنفذ في critical sections مفيش عمليه تانيه تتنفذ معاها
  - ➢ **<u>Progress</u>** - If **no process is executing** in its critical section and **there exist some processes that wish to enter** their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

لو مفيش عمليه بتتنفذ في critical sections وفي عمليات منتظره مفروض ده يسمح ان عمليه واحد تانيه تخش critical sections

  - ➢ **<u>Bounded Waiting-</u>** A bound must exist on the number **of times that other processes are allowed to enter their critical sections** after a process has made a request to enter its critical section and before that request is granted

    ده بيحط limit لكل عملية ان ليها عدد مرات لدخول critical section

❖ Peterson 's Solution:

- **classic software-based** solution to the critical-section
- It **may not work on modern hardware**
- Peterson's solution is **limited to two processes to be synchronized.**
- We have two shared variables:
  - Boolean **flag**[i]: indicates **if a process is interested in entering** the critical section
  - int **turn** :indicates **whose turn is to enter the critical** section

The structure of a process P_i

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

The structure of a process P_j

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

❖ Synchronization Hardware:

- Many systems provide special hardware instructions for critical section code
- All solutions below **based on idea of locking (**Protecting critical regions via locks)
- Uniprocessors could disable interrupts:
  - Currently running code **would execute without preemption**
  - Generally **too inefficient** on multiprocessor systems
- **Modern machines** provide special **atomic hardware** instructions:
  - Atomic = non-interruptible
  - Either **test** memory word and **set** value
  - Or **swap contents** of two memory words

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

❖ test and set Instruction:

- Test and Set is an **atomic hardware instruction** that can be used to solve the synchronization problem. In Test_and_Set, we **have a shared lock variable which can take one of the two values**, **O or 1**

- The Test_and_Set takes a shared variable called **target** that represent the **lock status and return its original value** and then set the variable **to true which means the lock is enabled**

- **Solution using test_and_set()**:
  - Shared boolean **variable lock, initialized to FALSE**
  - A process checks the lock before proceeding to the critical section. **If it's locked, it keeps waiting** until it's unlocked; **if it isn't, it takes the lock and run the critical section**

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */
   /* critical section */
   lock = false;
   /* remainder section */
} while (true);
```

طول ما lock ب true مفيش أي process هتخش
تنفذ الا لما ال processes اللي في critical
sectionتخلص و لما تخلص قيمة lock بتتحول
لfalse

- **compare_and_swap Instruction**:
  - CompareAndSwap hardware instruction is **also an atomic instruction**.
  - It operates **on three variables provided in its parameter**
  - The **lock variable** is set to **new value** only if the lock value is equal to compare variable (called **expected** variable).
  - CompareAndSwap always **returns the original** value of the lock variable.

```
int compare and swap(int *value, int expected, int new_value)
{
   int temp = *value;
   if (*value == expected)
      *value = new_value;
   return temp;
}
```

- **Solution using compare_and_swap:**

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
      /* critical section */
   lock = 0;
      /* remainder section */
} while (true);
```

- **Bounded-waiting Mutual Exclusion with test and set** :

ده بنستخدمه علشان كل عمليه متستناش وقت كتير علشان تتنفذ فكل عمليه هيكون ليها key و waiting فده هيحل مشكلة تحقيق Bounded-waiting

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
```

هو هنا عرف array اسمها waiting و اده هيكون رقم كل عمليه تمام فخلاه ب true و key= True

وبعدها نزل عمل while وخلي الشرط بتاعها ان waiting و key الاتنين يكونوا ب trueف دلوقتي هما فعلا ب true فلما يخش علي اللي مفروض ينفذه هيلاقي Key =test and set (lock)

وبما ان دي اول عمليه فهيكون lock ب false فيقوم خارج من اللوب علشان كده key قيمتها بقت ب false هيطلع يغير waiting برضه للعمليه دي و يخليها ب false وينفذ جزي critical section

بعدها بيشوف ايه العمليه اللي عليها الدوره اللي هو j

وبعدين بيعمل check ان العمليه j مش بتساوي العمليه i و ان waiting [j] قيمته ب false

يعني مش مستنيه تتنفذ ف كده يقوم داخل ينفذ اللوب اللي هو بياخد العمليه اللي بعدها لحد ملاقي عمليه في حالة waiting

وبعدين يتأكد هل I==j لو مش بيساوي بعض كده هيخلي waiting[j]=false فهيسمح ليها انها تخشcritical section

طب نفرض ان مثلا شرط while اتكسر ب انi==j

ده معناه ان هو قعد يدور ورجع تاني لنفس العنصر او العمليه فهيقوم مخلي lock=false بمعني انه هيستني عمليه تيجي تخش و تتنفذ

❖ <u>**Mutex Locks**</u>:

- **Previous solutions are complicated** and generally inaccessible to application programmers. OS designers build software tools to solve critical section problem
- Simplest is **mutex lock, boolean variable** indicating if lock is available or not
- Protect a critical section by **First acquire ( ) a lock** Then **release ( ) the lock**.
  NOTE➔ Calls to acquire ( ) and release ( ) must be atomic
- this solution requires **busy waiting** So This lock therefore called a <u>**spinlock**</u>

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

release() {
    available = true;
}
```

هنا ركز هو عامل not availableفي الشرط فلما تكون availableب True معني كده ان الشرط مش هيتحقق فيطلع بره اللوب وياخد processتانية

- <u>**The busy loop**</u> **used to block processes during the acquire phase is one disadvantage** with the software implementation shown here (as well as the hardware solutions previously presented).
- The <u>**types of locks that used busy loop are referred to as spinlocks,**</u> because the **CPU just sits and spins while blocking the process.**
- **Spinlocks is not ideal for single-CPU as it is wasteful of CPU cycles** that other process might be able to use productively.
- **multiprocessor machines can employ spinlocks** instead of context switches because they may take considerable time

Semaphore

- A semaphore S is **an integer variable** that can only be modified via two **atomic** operations: **wait** ( ) and **signal** ( ) .

```
wait (S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
signal (S) {
    S++;
}
```

- Can **only be accessed via two** indivisible (atomic) operations.
- **semaphore Implementation**:
  - The main issue with semaphores is the busy loop in the wait() which eats CPU cycles without accomplishing anything useful
  - to overcome the need for busy waiting ➔ When a **process is waiting for a semaphore** to become **available**, one option is to **block it and swap it out of the CPU**
  - each semaphore to **have a list of blocked processes** that are **awaiting its availability**, so that one of the **processes can be woken up and switched back** in.
  - **woken process may be switched back into the CPU** promptly or it will be **held in the ready queue** depending on the CPU-scheduling algorithm.
- **Semaphore Implementation with no Busy waiting**:

```
typedef struct{
    int value;
    struct process *list;
} semaphore;
```

  - each semaphore **there is an associated waiting queue**
  - **There are Two operation:**
    - **block** place the process invoking the operation on the appropriate **waiting queue**
    - **wakeup** **remove one of processes in the waiting** queue and **place it in the ready queue**

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```

الموضوع ده كله بيتعمل علشان يقضي علي

Busy loop that cause Spain lock

احنا نطلع العمليه اللي cpu عمال يعمل عليها لوب علي الفاضي ونحط العمليه دي في waiting queue وبعدين يشوف عمليه تانيه ينفذها

Wait

اول حاجه هيعمل ان يقلل قيمه semaphore ب واحد وبعدين يعمل check هلي قيمة semaphore اقل من 0 لو فعلا كده هيضيف العمليه في list وبعدين يعملblock

Signal

هيزود قيمة ال semaphore ب واحد

وبعدين يعمل check هلي القيمه اقل من 0 لو اه هيشيله من waiting list ويعمل wakeup للعمليه

- Deadlock and Starvation :
  - **Deadlock** **two or more processes are waiting** indefinitely for an event that can be **caused by only one of the waiting processes**

    بمعني ان في عمليتين الاتنين كل واحده فيهم مستنيه التانيه تخلص و الاتنين بيفضلوا ف حالة **wait**
  - **Starvation indefinite** blocking A process may **never be removed from the semaphore queue** in which it is suspended
  - **Priority Inversion** Scheduling problem **when lower-priority process holds a lock needed by higher-priority process**

By : Mohamed Gamal Maklad