

CYB 510 OPERATING SYSTEMS

Prepared by Prof. Dr. Lamiaa Fattouh

Lecture 5

جامعة الجميع الذكية

تسعى لتكون الكيان الأول الذي يتبنى مبدأ المجتمع التعليمي
التعاوني ويقدم العلم مقابل العمل لتنمية وبناء المجتمعات.

مزيد من التفاصيل

CYB 510 Process Synchron



جامعة الجميع الذكية
EVERYONE'S SMART UNIVERSITY

Outline

- **Background**
- **The Critical-Section Problem**
- **Peterson's Solution**
- **Hardware Support for Synchronization**
- **Mutex Locks**
- **Semaphores**

Chapter 6



Process Cooperation

- **Concurrent** processes are executing at the **same time**.
- **Cooperating** processes need **interprocess communication (IPC)**

- A **region of memory** that is **shared** by cooperating processes, is **established**.
- Processes can then **exchange information** by **reading** and **writing** data to the shared region

Shared memory



Chapter 6



Process Interaction & Concurrency

➤ Concurrency can be viewed at different levels:

Software
view

- ✓ **multiprogramming** — interaction between **multiple processes** running on **one CPU** (**pseudo-parallelism**)
- ✓ **multithreading** — interaction between **multiple threads** running in **one process**

Hardware
view

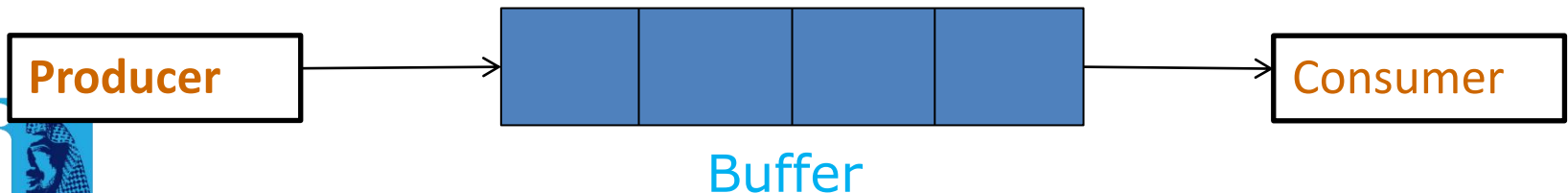
- ✓ **multiprocessors** — interaction between **multiple CPUs** running **multiple processes/threads** (**real parallelism**)
- ✓ **multicomputers** — interaction between **multiple computers** running **distributed processes/threads**

→ the **principles** of **concurrency** are basically **the same in all** of these categories

Remember : producer- consumer

- Concurrent access to shared data may result in **data inconsistency**
- Maintaining **data consistency** requires mechanisms to ensure **the orderly execution of cooperating processes**.
Producer-Consumer is an example of cooperating processes where data consistency is an important issue:
 - Compiler produces assembly code that is consumed by the assembler
 - Web-server provides HTML files that is read by the internet browser

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers. We can do so by having an integer **count** that keeps track of the number of full buffers. Initially, count 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.



Chapter 6

Remember : producer- consumer

```
public class BoundedBuffer implements Buffer
{
    private static final int BUFFER_SIZE = 5;
    private int count; // number of items in the buffer
    private int in; // points to the next free position
    private int out; // points to the next full position
    private Object[] buffer;

    public BoundedBuffer() {
        // buffer is initially empty
        count = 0;
        in = 0;
        out = 0;

        buffer = new Object[BUFFER_SIZE];
    }

    // producers calls this method
    public void insert(Object item) {
        // Figure 3.16
    }

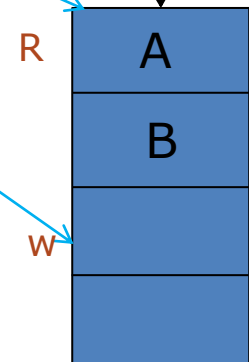
    // consumers calls this method
    public Object remove() {
        // Figure 3.17
    }
}
```

Full
Count = size

Synchronization

Empty
Count = 0

Producer



Consumer

Buffer



```
while (true) {  
    /* produce an item and put in  
    nextProduced */  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++;  
}
```

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

Chapter 6

Bounded-Buffer – Producer

```
while (true) {  
    /* produce an item and put in nextProduced */  
    while (count == BUFFER_SIZE)  
        ; // do nothing  
    buffer [in] = nextProduced;  
    in = (in + 1) % BUFFER_SIZE;  
    count++; //Do further
```

Shared

Bounded-Buffer – Consumer

```
while (true) {  
    while (count == 0)  
        ; // do nothing  
    nextConsumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    count--;  
    /* Do further*/
```

Code correct? Yes but Only if no concurrency. What can happen if P and C run concurrently or multiple P or C run concurrently?

- `count++` could be implemented as

```
register1 = count  
register1 = register1 + 1  
count = register1
```

- `count--` could be implemented as

```
register2 = count  
register2 = register2 - 1  
count = register2
```

- Consider this execution interleaving with “count = 5” initially:

S0: producer execute `register1 = count` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = count` {register2 = 5}
S3: consumer execute `register2 = register2 - 1` {register2 = 4}
S4: producer execute `count = register1` {count = 6}
S5: consumer execute `count = register2` {count = 4}

- **Race Condition** is a **situation** that occurs when two or more **concurrently processes** are **access** (reading or writing) some **shared data** and the final results (**outcome** of the final result) depends on who runs precisely when (**order of scheduling**) . It **leads** to **data inconsistency** and should be **avoided** by guaranteeing that the communicating process are **synchronized** and **coordinating** in some way.

Critical Section

- Let's study the race condition situation more deeply to **guard against** it because Sharing for variable, files, ... is needed everywhere.
- **Avoiding** is possible if we guard the shared region (**critical section (CS)**).
- *Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on.*

Critical Section

- CS similar problem: “**Design a protocol** for a **shared room in an apartment** where no more than one person can be inside the room at any time”? Shared room is the CS which needs permission to **use it alone** and when you finished just give others the chance to use it. **Mutual exclusive Solution**? You inside, others outside. **Buy LOCK or Don't Disturb sign and use the shared room without race condition.**

- The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.
- That is, no two processes are executing in their critical sections at the same time.

- **Critical section problem:** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.
- The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**.
- The remaining code is the **remainder section**.

do {

entry section

critical section

exit section

remainder section

}

while (TRUE);

Chapter 6

Critical Section

```
do {
```

entry section

```
buf [in] = nextP;//produce item  
in = (in + 1) % BUF_SIZE;
```

```
nextC = buf[out];//consume item  
out = (out + 1) % BUF_SIZE
```

critical section

```
Count++
```

```
Count--
```

exit section

remainder section

```
readLine()....
```

```
Println()....
```

```
} while (TRUE);
```

A solution to the critical-section problem must satisfy the following three requirements:

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 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
 3. **Bounded Waiting** - 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
- Assume that each process executes at a nonzero speed
 - No assumption concerning relative speed of the N processes

1. If one process is in critical section test the second process if the second process enter CS then Mutual Exclusion is not valid
2. If the two processes need to enter CS in the same time, if the two process
 - a) enter CS then Mutual Exclusion is not valid
 - b) Looping in the entry section (progress is not valid)

3. If one process is in the remainder section and the second process need to enter its CS if it must wait for the first process the condition of progress is not valid
4. If the two process need to enter critical section if the first enter and leave the second must has the chance to enter the CS if not the bounded waiting is not valid

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)
 do {
 entry section
 critical section
 exit section
 remainder section
 } **while (1);**
- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
 - **int turn;**
initially **turn = 0**
 - **turn = i** $\Rightarrow P_i$ can enter its critical section

Process P_i

```
do {  
    while (turn != i) ;  
    critical section  
    turn = j;  
    remainder section  
} while (1);
```

Process P_j

```
do {  
    while (turn != j) ;  
    critical section  
    turn = i;  
    remainder section  
} while (1);
```

- Satisfies mutual exclusion, but not progress

Chapter 6

Test 1

Processor run one instruction from P1 and one instruction from P2. Initial turn = 1

1- P1 → enter Critical section CS1

2- P2 → LOOP

3- P1 → exist CS1 and turn = 2

4- P2 can enter CS2

Mutual execution is valid

If P1 in remaining section and P2 finish CS2 and set turn = 1

if P2 need to enter CS2 it can not enter

Therefore, **Progress is not valid**

Chapter 6

Algorithm 2

- Shared variables
 - boolean** `flag[2]`;
initially `flag[0] = flag[1] = false`.
 - `flag[i] = true` $\Rightarrow P_i$
ready to enter its critical section
- Process P_i

```
do {  
    flag[i] := true;  
    while (flag[j]) ;  
    critical section  
    flag[i] = false;  
    remainder section  
} while (1);
```

Process P_j

```
do {  
    flag[j] := true;  
    while (flag[i]) ;  
    critical section  
    flag[j] = false;  
    remainder section  
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Chapter 6

Initial flag[1] = false and flag[2]=false

T1: flag[i] := true;

T2: flag[j] := true;

T3: while (flag[j]) ;

T4: while (flag[i]) ;

T5: P1 → LOOP

T6: P2 → LOOP

Progress is not valid

Peterson's Solution

- Two process solution
- Assume that the LOAD and STORE instructions are atomic; that is, cannot be interrupted.
- The two processes share two variables:
 - int **turn**;
 - Boolean **flag[2]**
- The variable **turn** indicates whose turn it is to enter the critical section.
- 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!

Chapter 6

Algorithm 3

Process i
do {

```
flag[i] = TRUE;  
turn = j;  
while ( flag[j] && turn == j);
```

CRITICAL SECTION

```
flag[i] = FALSE;
```

REMAINDER SECTION

} while (TRUE);

Process J
do {

```
flag[j] = TRUE;  
turn = i;  
while ( flag[i] && turn == i);
```

CRITICAL SECTION

```
flag[j] = FALSE;
```

REMAINDER SECTION

} while (TRUE);

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- Notation $\leq$ lexicographical order (ticket #, process id #)
 - $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$
 - $\max(a_0, \dots, a_{n-1})$ is a number, k , such that $k \geq a_i$ for $i = 0, \dots, n - 1$

- Shared data

boolean choosing[n];

int number[n];

Data structures are initialized to **false** and **0** respectively

Chapter 6

Bakery Algorithm

```
do {  
    choosing[i] = true;  
    number[i] = max(number[0], number[1], ..., number [n - 1])+1;  
    choosing[i] = false;  
    for (j = 0; j < n; j++) {  
        while (choosing[j]) ;  
        while ((number[j] != 0) && (number[j],j) < (number[i],i)) ;  
    }  
    critical section  
    number[i] = 0;  
    remainder section  
} while (1);
```

- OS itself consists of many modules (processes) that may run concurrently. What can happen to the shared data structures of the OS kernel? Say Process table, File table, memory allocation table.. etc?
- There are 2 types of Kernel: preemptive and non-preemptive

Chapter 6

Preemptive and Nonpreemptive Kernel

- A **non-preemptive kernel** allows a process to run until it exits kernel mode, block, or voluntarily yields control of CPU.
 - Because there is max one process in kernel mode, **it is free from race conditions (situation to be controlled)** on **kernel** Data structure.
- A **preemptive kernel** preempts a process while it is running in kernel mode
 - Must be carefully designed to **ensure that kernel shared data are free from race conditions** especially on SMP architectures.
 - More suitable for real time programming as it allows preemption
 - May be more responsive (a process can't run for long period)



- Why, then, would anyone favor a preemptive kernel over a non-preemptive one?
- A preemptive kernel is **more suitable for real-time programming**, as it will allow a real-time process to preempt a process currently running in the kernel.
- Furthermore, a preemptive kernel **may be more responsive**, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes.

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems
 - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
 - Atomic = non-interruptable
 - Either test memory word and set value
 - Or swap contents of two memory words

- Definition:

```
boolean TestAndSet (boolean  
*target)  
{  
    boolean rv = *target;  
    *target = TRUE;  
    return rv;  
}
```



Chapter 6

Solution using TestAndSet

- Shared boolean variable lock., initialized to false.
- Solution:

```
do {  
    while ( TestAndSet (&lock ))  
        ; /* do nothing
```

```
    // critical section
```

```
    lock = FALSE;
```

```
    // remainder section
```

```
} while ( TRUE);
```

- Definition:

```
void Swap (boolean *a, boolean *b)
{
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```

Solution using Swap

- Shared Boolean variable lock initialized to FALSE; Each process has a local Boolean variable key.
- Solution:

```
do {  
    key = TRUE;  
    while ( key == TRUE)  
        Swap (&lock, &key );  
  
    // critical section  
  
    lock = FALSE;  
  
    // remainder section  
  
} while ( TRUE);
```

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore S – integer variable
- Two standard operations modify S : `wait()` and `signal()`
 - Originally called $P()$ and $V()$
- Less complicated
- Can only be accessed via two indivisible (atomic) operations

```
– wait (S) {  
    while S <= 0  
        ; // no-op  
    S--;  
}  
  
– signal (S) {  
    S++;  
}
```

Chapter 6

Semaphore as General Synchronization Tool

- **Counting semaphore** – integer value can range over an unrestricted domain. Counting semaphores can be used to control access to a give resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a `wait()` operation on the semaphore (thereby decrementing the count). When a process release a resource, it performs a `signal()` operation (incrementing the count). When the count for the semaphore goes to 0, all resources being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

- Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
 - Also known as **mutex locks**
- Can implement a counting semaphore **S** as a binary semaphore
- Provides mutual exclusion
 - Semaphore **S**; // initialized to 1
 - wait (S);

Critical Section

signal (S);

Chapter 6

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
- Let s and q be two semaphores initialized to 1

P_0

```
wait (S);  
wait (Q);  
.  
.  
.  
signal (S);  
signal (Q);
```

P_1

```
wait (Q);  
wait (S);  
.  
.  
.  
signal (Q);  
signal (S);
```

- Another problem related to deadlocks is **indefinite blocking**, or **starvation**.
- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended. Indefinite blocking may occur if we add and remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.



- Execute B in P_j only after A executed in P_i
- Use semaphore $flag$ initialized to 0
- Code:

P_i	P_j
$\vdots$	$\vdots$
A	$wait(flag)$
$signal(flag)$	B

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- N buffers, each can hold one item
- Semaphore **mutex** provides mutual exclusion for accesses to the buffer pool and initialized to the value 1
- Semaphore **full** initialized to the value 0
- Semaphore **empty** initialized to the value N .
- The empty and full semaphores count the number of empty and full buffers.

- The structure of the producer process

```
do {  
  
    // produce an item  
  
    wait (empty);  
    wait (mutex);  
  
    // add the item to the buffer  
  
    signal (mutex);  
    signal (full);  
} while (true);
```

- The structure of the consumer process

```
do {  
    wait (full);  
    wait (mutex);  
  
    // remove an item from buffer  
  
    signal (mutex);  
    signal (empty);  
  
    // consume the removed item  
  
} while (true);
```

- A data set is shared among a number of concurrent processes
 - Readers – only read the data set; they do **not** perform any updates
 - Writers – can both read and write.
- Problem – allow multiple readers to read at the same time. Only one single writer can access the shared data at the same time.
- No reader will be kept waiting unless a writer has already obtained permission to use the shared object. Writers may starve.
- Shared Data
 - Data set
 - Semaphore **wrt** initialized to 1 and its functions as a mutual-exclusion semaphore for writers. it is also used by the first or last reader that enters or exists the critical section. It is not used by reader who enter or exit while other readers are in their critical sections.
 - Integer **readcount** initialized to 0. Readcount keeps track of how many processes are currently reading the object.
 - Semaphore **mutex** is used to ensure mutual exclusion when readcount is updated. Mutex is initialized to 1.

- The structure of a writer process

```
do {  
    wait (wrt) ;  
  
    // writing is performed  
  
    signal (wrt) ;  
} while (true)
```

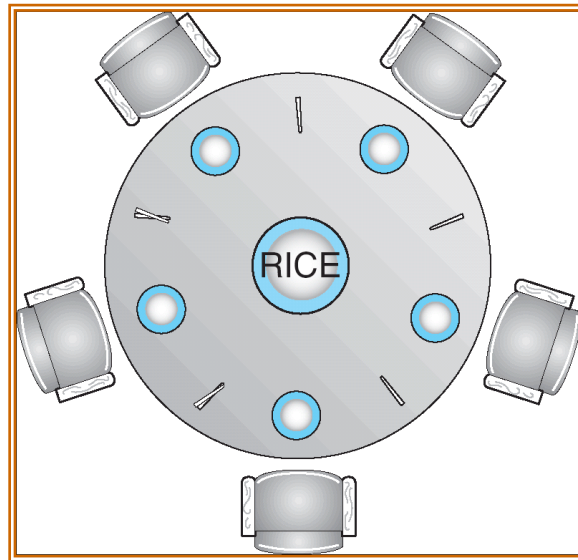
The structure of a reader process

```
do {  
    wait (mutex) ;  
    readcount ++ ;  
    if (readercount == 1) wait (wrt) ;  
    signal (mutex)  
  
    // reading is performed  
  
    wait (mutex) ;  
    readcount -- ;  
    if readcount == 0) signal (wrt) ;  
    signal (mutex) ;  
} while (true)
```

- Note that,
 1. if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex.
 2. When a writer executes signal(wrt), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

Chapter 6

Dining-Philosophers Problem



- Shared data
 - Bowl of rice (data set)
 - Semaphore `chopstick` [5] initialized to 1

- The structure of Philosopher *i*:

```
Do {  
    wait ( chopstick[i] );  
    wait ( chopstick[ (i + 1) % 5] );  
  
    // eat  
  
    signal ( chopstick[i] );  
    signal ( chopstick[ (i + 1) % 5] );  
  
    // think  
  
} while (true) ;
```

Chapter 6

End of Chapter 6