SYNCHRONIZATION - Introduction

- Several processes / threads access or manipulate the same data concurrently. If output is different – order of execution – the condition is called as **Race Condition**
- True Multiprocessing / Multithreading requires simultaneous access as long as data inconsistent scenarios are avoided.
- File locking with read and write permissions are primitive solutions to restricted access to data!
- Transaction Processing Concepts – issues such as Lost Update, Temporary Update, Dirty Read, Incorrect Summary ,etc.

- Transactions there can be equated as Processes or Threads in the OS context.
- Attributes there equate to program variables , local or global variables, file pointers, etc.
- Concurrency control management techniques such as locking, 2phase locking were used to address concurrency related issues.
- In OS context, Synchronization addresses concurrent execution of processes in a safe manner – achieved using Semaphores support in Linux / Windows Operating Systems

- Semaphores help in allowing synchronized access to shared data / program variables in a SAFE and CONSISTENT manner.
- Recall the Producer Consumer discussion we had as part of Inter Process Communication for data exchange.
- Examples of Compiler – Assembler – Loader etc.
- Data to be shared amongst the producer and consumer processes needs to be synchronized
- Eg. Data such as buffer array, indices of in and out, etc. when being manipulated by one process should happen in a safe fashion
- Next slide recalls the earlier code .We shall edit this definition of producer and consumer to include a counter variable to keep track of no of items produced or consumed thus far.

```
while (true)
{
// item not produced
while ((in+1) % BS = = out)
; // do nothing as the Buffer is Full
buffer [in] = next-produced-item;
in  = ( in + 1 ) % BS ;
}
```

```
while (true)
{
// item not consumed
while (in = = out)
; //do nothing as Buffer is Empty
next-consumed-item = buffer[out];
out  =  (out + 1 ) % BS ;
}
```

buffer - circular array with two pointers / access indices

in - next free position in the buffer array ;

out - first full position in the buffer array ;

in == out implies EMPTY; (in+1) % BS = = out implies FULL buffer states

Producer Process:

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

Consumer Process:

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

Producer Consumer - counter tracking

- Both the producer and the consumer are adjusting the value of the variable counter, which can lead to a condition known as a ***race condition***.
- In this condition a piece of code may or may not work correctly, depending on which of two simultaneous processes executes first,
- and more importantly if one of the processes gets interrupted such that the other process runs between important steps of the first process.
- Assume one such order of execution as follows:

Producer:

$$\begin{aligned}&register_1 = \texttt{counter}\\&register_1 = register_1 + 1\\&\texttt{counter} = register_1\end{aligned}$$

Consumer:

$$\begin{aligned}&register_2 = \texttt{counter}\\&register_2 = register_2 - 1\\&\texttt{counter} = register_2\end{aligned}$$

Interleaving:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

- Producer executing "counter++" at the same time the Consumer is executing "counter--".
- If one process gets part way through making the update and then the other process intervenes; counter value may be in INCORRECT or UNSAFE state
- Counter – all are individual instructions how can they be interrupted!
- A++ internally involves 3 operations at the hardware level
- 1) Fetch counter from memory into a register,
- (2) increment or decrement the register, and
- (3) Store the new value of counter back to memory.

- What would be the resulting value of counter if the order of statements T4 and T5 were reversed?
- Race conditions are also very difficult to reproduce.
- race conditions are ***notoriously difficult*** to identify and debug, because by their very nature they only occur on rare occasions
- when the timing is just exactly right. (or wrong! :-
- **solution is to only allow one process at a time to manipulate the value "counter“**
- P-C is a special case of such problems that are in OS circles referred as the **CRITICAL SECTION PROBLEM (CSP)**

- **Sections of the code that are accessing or manipulating shared data are referred as the CRITICAL SECTION**
- If there are number of cooperating processes, (group) each has a critical section of code, with the following conditions and terminologies:
- Only one process in the group can be allowed to execute in their critical section at any one time. If one process is already executing their critical section and another process wishes to do so, **then the second process must be made to wait until the first process has completed their critical section work.**

- The code preceding the critical section, and which controls access to the critical section, is termed the entry section. It acts like a **carefully controlled locking door**.
- code following the critical section is termed the exit section. It **generally releases the lock** on someone else's door, or at least lets the world know that they are no longer in their critical section.
- rest of the code not included in either the critical section or the entry or exit sections is termed the remainder section.

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (TRUE);
```

Any solution to the CSP must satisfy three important properties

- **Mutual Exclusion (MUTEX)**
- **Progress**
- **Bounded Wait (ing)**

More details in next lecture!!!

CRITICAL SECTION PROBLEM

- CSP Solution needs to satisfy three important properties of MUTEX, PROGRESS and BOUNDED WAIT
- Meaning of each property follows:
- **MUTEX** - Only one process at a time can be executing in their critical section.
- **Progress** - If no process is currently executing in their critical section, and one or more processes want to execute their critical section, then **only the processes not in their remainder sections** can participate in the decision, and the decision cannot be postponed indefinitely .
- processes cannot be blocked forever waiting to get into their critical section

CRITICAL SECTION PROBLEM

- **Bounded Waiting** - There exists a limit as to how many other processes can get into their critical sections after a process requests entry into their critical section and before that request is granted. (I.e. a process requesting entry into their critical section will get a turn eventually, and there is a limit as to how many other processes get to go first.)
- Kernel processes can also be subject to race conditions, which can be especially problematic when updating commonly shared kernel data structures such as open file tables or virtual memory management.