PENNSTATE

Concurrency Control via Synchronization

Need for Synchronization

- Activities share resources.
- It is important to coordinate their progress to ensure proper usage.

Examples of coordination

- John and Mary are each printing a different 10 page document on the same printer. You do not want their pages/output interleaving!
  – Exclusion
- John and Mary are sharing a bank account. John deposits $10. Mary should be allowed to withdraw only after this deposit
  - Ordering

PENNSTATE

Resources

- **There are different kinds of resources that are shared:**
  - **Physical (terminal, disk, network, …)**
  - **Logical (files, sockets, memory, …)**
- **For the purposes of this discussion, let us focus on "memory" to be the shared resource**
  - **i.e. threads can all read and write into memory (variables) that are shared.**

PENNSTATE

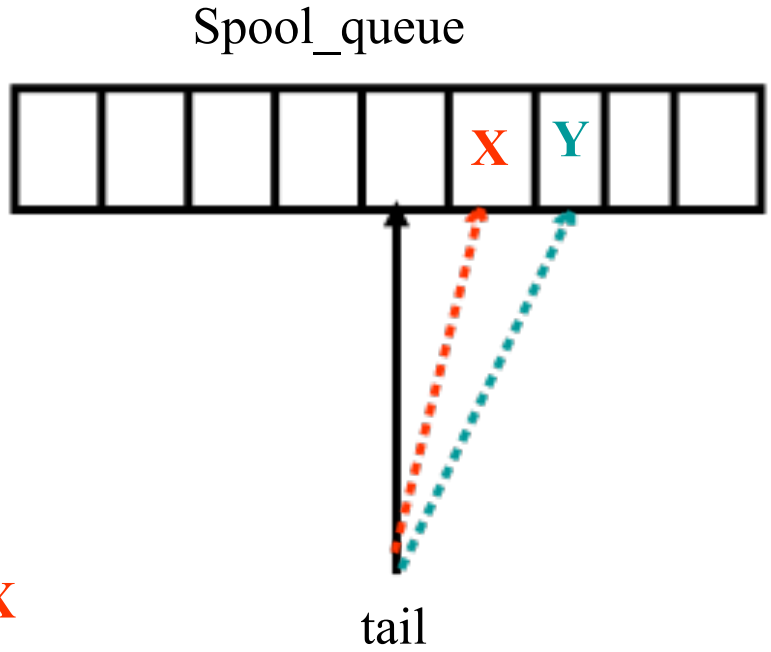
Problems due to sharing

- Consider a shared printer queue, spool_queue[N]
- 2 threads want to enqueue an element each to this queue.
- tail points to the current end of the queue
- Each thread needs to do

tail = tail + 1;
spool_queue[tail] = "element";

Thread will look at tail to know where to put element

What we are trying to do …



Thread 1

tail = tail + 1;
Spool_queue[tail] = X

Thread 2

tail = tail + 1;
Spool_queue[tail] = Y

What is the problem?

- tail = tail + 1 is NOT 1 machine instruction
- It can translate as follows:

  Load tail, R1
  Add R1, 1, R2
  Store R2, tail

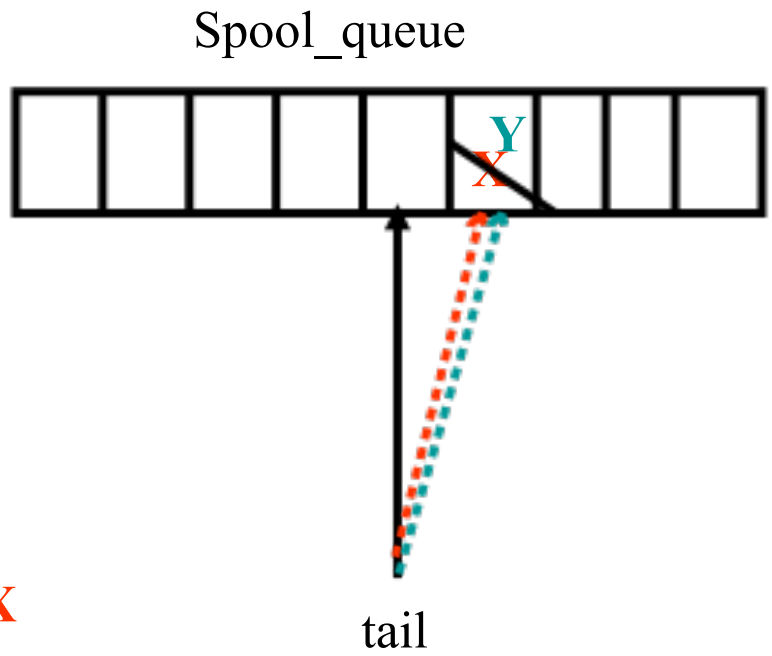
- These 3 machine instructions may NOT be executed atomically.

Atomically

- **If each thread is executing this set of 3 instructions, context switching can happen at any time.**
- **Let us say we get the following resultant sequence of instructions being executed:**

  1: Load tail, R1
  2: Load tail, R1
  1: Add R1, 1, R2
  2: Add R1, 1, R2
  1: Store R2, tail
  2: Store R2, tail

Leading to …



Thread 1

tail = tail + 1;
Spool_queue[tail] = X

Thread 2

tail = tail + 1;
Spool_queue[tail] = Y

PENNSTATE

race condition: result of a sequence of instructions depends on its interleaving with other instructions.

- Situations like this that can lead to erroneous execution depending on who executes when, are called race conditions.

happend in shared memory space like global variable

- Debugging race conditions is NOT easy! since errors can be non-repeatable.

PENNSTATE

Avoiding Race Conditions

- **If we had a way of making those (3) instructions atomic – i.e., while one thread is executing those instructions, another cannot execute the same instructions – then we could have avoided the race condition.**
- **These 3 instructions are said to constitute a critical section.**
- **While one thread is in a critical section, another should NOT be allowed to execute the same critical section – mutual exclusion.**