Race conditions and Locks

Race conditions and Locks • Multiprogramming and concurrency bring in the problem of race conditions, where multiple processes executing concurrently on shared data may leave the shared data in an undesirable, inconsistent state due to concurrent execution. Note that race conditions can happen even on a single processor system, if processes are context switched out by the scheduler, or are interrupted otherwise, while updating shared data structures.

• Consider a simple example of two processes of a process incrementing a shared variable. Now, if the increments happen in parallel, it is possible that the processes will overwrite each other's result, and the counter will not be incremented twice as expected. That is, a line of code that increments a variable is not atomic, and can be executed concurrently by different threads.

ex. bank applications

• Pieces of code that must be accessed in a mutually exclusive atomic manner by the contending processes are referred to as critical sections. Critical sections must be protected with locks to guarantee the property of mutual exclusion.

The code to update a shared counter is a simple example of a critical section.

Code that adds a new node to a linked list is another example.

A critical section performs a certain operation on a shared data structure, that may temporarily leave the data structure in an inconsistent state in the middle of the operation.

Therefore, in order to maintain consistency and preserve the invariants of shared data structures, critical sections must always execute in a mutually exclusive fashion.

Locks guarantee that only one contending thread resides in the critical section at any time, ensuring that the shared data is not left in an inconsistent state.