SY2-DATARACES: 231-250

231 [T / F] A variable in a thread's stack cannot be modified by the peer threads.

232 [T / F] A static C variable is stored in the stack of the thread that calls the function. But there will only be one instance of the variable when the function is called recursively.

233 [T / F] Any global variables or memory lines pointed by any global pointer are essentially memory lines that are "shared" (whose accesses need to be regulated with locks to prevent race conditions).

234 [T / F] When you declare "int yourArray [10]," the instruction "yourArray[1]=777" is the same as "*(yourArray+1) = 777". This is because "yourArray" is a pointer value to the base of your array and adding the pointer by "+1" will cause the value to be incremented by 4 bytes because it is an "int*" pointer.

235 [T / F] A cute C operation such as (*e.g.*, i++) is always atomic (cannot be interrupted/interleaved in the middle).

236 [T / F] Basic machine instructions like add, sub, inc, etc. are atomic (*i.e.*, cannot be interrupted/interleaved in the middle).

237 [T / F] Race condition implies that the outcome can vary from the expectation, hence is hard to debug the root cause.

238 [T / F] Critical section is a property that states: at most one thread runs in mutual exclusion.

239 [T / F] A critical section has instructions that read/write shared data, hence must be protected with locks.

240 [T / F] Adding too many locks or simply wrapping critical sections with locks without reorganizing your code can cause performance problems (a.k.a. the "lock step" performance).

241 [T / F] When using threads and locks, it's a good practice to create embarassingly parallel tasks that do not need to synchronize too often. (See the "Parallelizing a job" slides).

242 [T / F] Only put variables you are protecting in the critical section. It is a good rule of thumb to make critical sections as small as you can as long as there are no synchronization issues.

243 [T / F] Big locks (like a single big bank lock) reduce deadlock probability and deliver better performance than small locks.

244 [T / F] Thread1: {lock(x); lock(y); do something ... }; Thread2: {lock(y); lock(x); do something ... }; This code will always run into a deadlock.

245 [T / F] If your process hits a deadlock/race condition, you can restart the process and will reproduce the bug in one run.

246 [T / F] If you hit a deadlock, the only way to make progress is to restart your job/app/computer, because it's impossible to undo the lock (unacquire the lock), unless you have a special OS/hardware support such as Hardware/Software Transactional Memory.

247 [T / F] Deadlock cannot happen if you use an odd number of locks.

248 [T / F] If you have CPU-intensive computations, you will get the benefit of *parallelism* (make your

program run faster) by running multiple threads on a single CPU core. (If you have a lot of laundries, you will get the benefit of parallelism by breaking them to multiple baskets but only with one washer).

249 [T / F] If you have a mix of CPU-intensive and IO-intensive (network/disk) jobs, you will get the benefit of *concurrency* by running multiple threads even on a single CPU core.

250 [T / F] — 250