**Operating System Home Assignment 1**

**Aditya Parikh**

**ID - 121001**

1. Consider the following C code that calls fork(). If you assume that the child process is always

Scheduled before the parent process, what will be the output?

int main()

{

int i;

for (i = 0; i < 3; i++) {

if (fork() == 0) {

printf ("Child sees i = %d\n", i);

exit (1);

} else {

printf("Parent sees i = %d\n", i);

}

}

}

**A:** **The output of the above code is:**

**Parent sees i = 0**

**Child sees i = 0**

**Parent sees i = 1**

**Child sees i = 1**

**Parent sees i = 2**

**Child sees i = 2**

* **Using vfork we can schedule child always before the parent process.**

**Child sees i = 0**

**Parent sees i = 0**

**Child sees i = 1**

**Parent sees i = 1**

**Child sees i = 2**

**Parent sees i = 2**

1. Consider the following C code that creates and joins with two threads. Assuming that the threads are scheduled completely before the parent process (i.e., have a higher priority), what will be the output from running this program? Be careful! There is a significant trick!

int a = 0;

void \*print\_fn(void \*ptr)

{

int tid = \*(int \*)ptr;

int b = 0;

a++; b++;

printf("id: %d a: %d b: %d\n", tid, a, b);

while (1); // Spin-wait here forever

}

int main()

{

pthread\_t t1, t2;

int tid1 = 1;

int tid2 = 2;

int ret1, ret2;

a++;

printf("Parent says a: %d\n", a);

ret1 = pthread\_create(&t1, NULL, print\_fn, (void \*)&tid1);

ret2 = pthread\_create(&t2, NULL, print\_fn, (void \*)&tid2);

if (ret1 || ret2) {

fprintf(stderr, "ERROR: pthread\_create failed\n");

exit(1);

}

if (pthread\_join(t1, NULL)) {

perror("join of t1");

exit(1);

}

if (pthread\_join(t2, NULL)) {

perror("join of t2");

exit(1);

}

printf("Thread 1 and 2 complete\n");

}

**A:** **Here the threads are never formed because of the infinite while loop.**

**Parent says a: 1**

**id: 2 a: 2 b: 1**

**id: 1 a: 2 b: 1**

**Above is the result with while loop. If we remove the while loop then the threads are formed. Also after removing while loop threads are joined randomly.**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

1. In some multi-threaded applications, m user-level threads are mapped to n kernel-level threads. Why

Can this be a good idea (compared to using only user-level or only kernel-level threads?)

**A: Because user level threads have benefits like faster performance while kernel level threads have benefits of context switching if thread is blocked. So to achieve best application both user level threads and kernel level threads are mapped with each other. Therefore m user level threads are mapped to n kernel level threads.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. For what relative values of m and n is this mapping a possibility (or at all reasonable)? For which relative values is this the best choice?
   1. m >> n
   2. m > n
   3. m (approx.) = n
   4. m < n
   5. m << n

* **A collection of ULT's for all KLT's**

**This situation is ideal on a shared memory system. There is essentially a "pool" of ULT's to which each KLT has access. Ideally, the threading library scheduler would assign ULT's to each KLT upon request as opposed to the KLT's accessing the pool individually. The later could cause race conditions or deadlocks if not implemented with locks or something similar.**

* **A collection of ULT's for all KLT**

**This situation is ideal on a distributed memory system. Each KLT would have a collection of ULT's to run. The drawback is that the user (or the threading library) would have to divide the ULT's between the KLT's. This could result in load imbalance since it is not guaranteed that all ULT's will have the same amount of work to complete and complete roughly the same amount of time. The solution to this is allowing for ULT migration; that is, migrating ULT's between KLT's.**