HW 3 Solutions

1. The uses of threading

Sometimes. Let's take the single-CPU case first. The multithreaded code is not going to give substantially better performance since only one thread can execute at any given time. Moreover, the overhead of synchronization may slow the program down. On a multi-CPU machine, it also depends. If the threads are spending most of their time doing synchronization, then the multithreaded program may, again, have worse turn-around time. If, however, synchronization is rare and the threads can be run on different hardware cores (i.e., the threads are working simultaneously and in parallel), then a multithreaded process can indeed turn around a job more quickly than a single-threaded one (because parallelism is being exploited).

2. Threads vs. processes

A C

3. Race conditions

Semantics of `add(a,b)` is that when the function returns, the "new" a should be the sum of the "old" a and b. If we have initially, `Point a = {1,1}; Point b = {1,1};` Thread 1 calls `add(&a,&b)`, Thread 2 calls `add(&b,&a)`; **Event Sequence Memory** Thread 1 executes first statement of add; `a = {2,1}; b = {1,1};` Thread 2 executes first statement; `a = {2,1}; b = {3,1};` Thread 2 executes second statement; `a = {2,1}; b = {3,2};` Thread 1 executes second statement; `a = {2,3}; b = {3,2};`

So in the end, `a = {2,3}; b = {3,2};` The semantics of the add function is broken. This is a race condition.

4 Synchronization: warmup

```
int i = 0;
scond_t cond;
smutex_t mutex;

void foo(void *)
{
  smutex_lock(&mutex);
  printf("I am foo!!!\n");
  i = 1;
  scond_signal(&cond, &mutex);
  smutex_unlock(&mutex);
}

void boo(void *)
{
  smutex_lock(&mutex);
  while (!i) {
```

```
    scond_wait(&cond, &mutex);
  }
  printf("I am boo!!!!\n");
  smutex_unlock(&mutex);
}

int main(int argc, char**argv)
{
  smutex_init(&mutex);

  scond_init(&cond);

  thread* t1 = create_thread(foo);
  thread* t2 = create_thread(boo);

  join_thread(t1);
  join_thread(t2);

  smutex_destroy(&mutex);

  scond_destroy(&cond);

  exit(0);
}
```

5 More practice with synchronization

```
class Table {
  public:
    void matchSmokerUseTable();
    void paperSmokerUseTable();
    void tobaccoSmokerUseTable();
    void agentUseTable();

    Table();
    int getPaper();
    int getTobacco();
    int getMatch();
    ~Table();

  private:
    scond_t tableEmptyCond; // contract: when table is empty, signal this
    scond_t tableFullCond; // when table is full, signal this
    smutex_t tableMutex;

    int paper, tobacco, match; // shared state, encapsulated
    // table is small, so can at most have two items at once
```

```
    bool tableIsFull();
    bool tableIsEmpty();
}
```

```
Table::
Table() {
  paper = 0;
  tobacco = 0;
  match = 0;
  scond_init(&tableFullCond);
  scond_init(&tableEmptyCond);
  smutex_init(&tableMutex);
}

Table::
~Table() {
  scond_destroy(&tableEmptyCond);
  scond_destroy(&tableFullCond);
  smutex_destroy(&tableMutex);
}

// warm up example
int
Table::getPaper() {
  smutex_lock(&tableMutex);
  int r = paper;
  smutex_unlock(&tableMutex);
  return r;
}
// get tobacco and get match is similar

// private method, why we don't need mutex here?
bool
Table::tableIsFull() {
  return paper + tobacco + match >= 2;
}

bool
Table::tableIsEmpty() {
  return !tableIsFull();
}

void
Table::agentUseTable() {
  smutex_lock(&tableMutex);
```

```
  while (tableIsFull()) {
    scond_wait(&tableEmptyCond, &tableMutex);
  }

  chooseIngredients(&paper, &tobacco, &match);

  // why broadcast instead of signal?
  scond_broadcast(&tableFullCond, &tableMutex);
  smutex_unlock(&tableMutex);
}

void
Table::paperSmokerUseTable() {
  smutex_lock(&tableMutex);

  while(tableIsEmpty() || (match < 1 || tobacco < 1)) {
    scond_wait(&tableFullCond, &tableMutex);
  }
  match--;
  tobacco--;

  scond_signal(&tableEmptyCond, &tableMutex);
  smutex_unlock(&tableMutex);
}
```