Intro to Synchronization

Brief intro to scheduling

We have:

- Multiple threads/processes ready to run
- Some mechanism for switching between them
    - *Context switches*
- Some policy for choosing the next process to run
    - This policy may be pre-emptive
    - Meaning thread/process can't anticipate when it may be forced to yield the CPU
    - By design, it is not easy to detect it has happened (only the timing changes)

Synchronization

- Arbitrary interleaving of thread executions can have unexpected consequences
  - We need a way to restrict the possible interleavings of executions
  - Scheduling is invisible to the application
  - Synchronization is the mechanism that gives us this control

Flavours

1. Enforce single use of a shared resource
   - Called the critical section problem
   - E.g. Two threads printing to the console
     - T1: printf(“Hello”);   T2: printf(“Goodbye”);
     - Result is “HelloGoodbye” OR “GoodbyeHello”, but never “HeGooldbloye” or some other mixture
2. Control order of thread execution
   - E.g. parent waits for child to finish
   - E.g. Ensure menu prints prompt after all output from thread running program

Motivating Example

- Suppose we write functions to handle withdrawals and deposits to bank account:

```
Withdraw(acct, amt) {
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    return balance;
}
```

```
Deposit(account, amount) {
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    return balance;
}
```

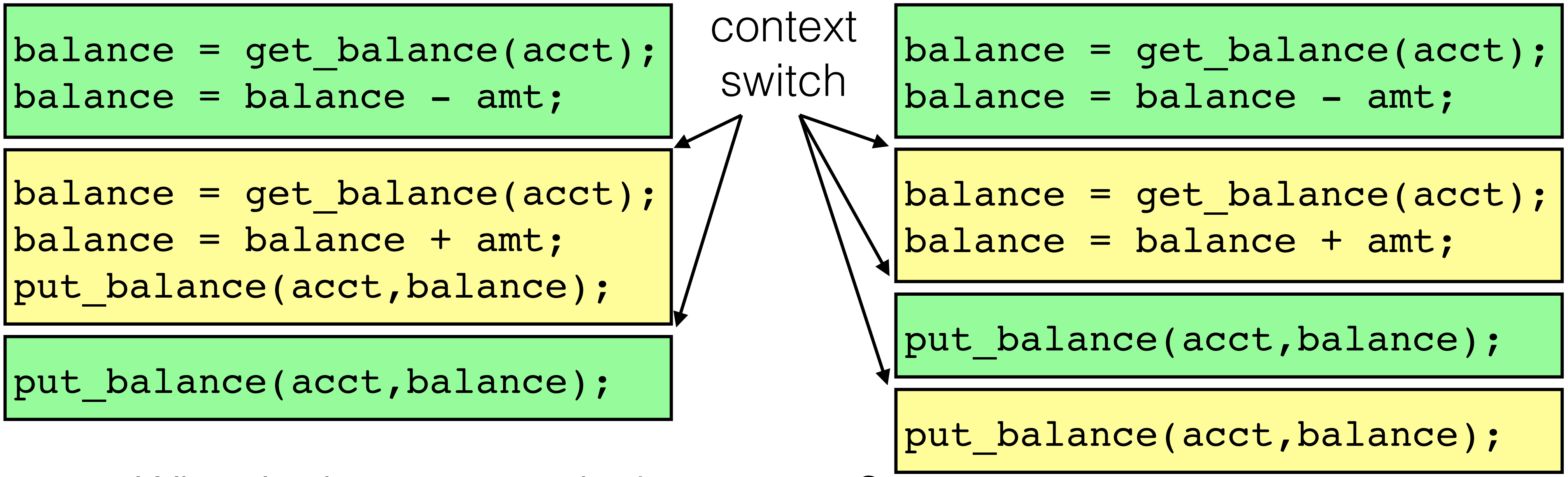
- Now suppose you share this account with someone and the balance is $1000

- You each go to separate ATM machines - you withdraw $100 and your S.O. deposits $100

🟩 Withdraw 🟨 Deposit

Interleaved Schedules

- The problem is that the execution of the two processes can be interleaved



```
balance = get_balance(acct);
balance = balance - amt;
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct,balance);
```

```
put_balance(acct,balance);
```

```
balance = get_balance(acct);
balance = balance - amt;
```

```
balance = get_balance(acct);
balance = balance + amt;
```

```
put_balance(acct,balance);
```

```
put_balance(acct,balance);
```

- What is the account balance now?
    - Is the bank happy with our implementation?
    - Are you?

What went wrong

- Two concurrent threads manipulated a shared resource (the account) without any synchronization
  - Outcome depends on the order in which accesses take place
    - This is called a race condition
- We need to ensure that only one thread at a time can manipulate the shared resource
    - So that we can reason about program behaviour
    - We need synchronization

Caution!

- Race conditions can occur even with a simple shared variable, even on a uniprocessor:
  - T1 and T2 share variable X
  - T1 increments X (X := X + 1)
  - T2 increments X (X : = X + 1)
  - But at the machine level we have:

```
T1:     LOAD X
        INCR
        STORE X
```

```
T2:     LOAD X
        DECR
        STORE X
```

  - Same problem of interleaving can occur

Example code…

```
int count = 0;

void *thread1(void *threadid) {
    printf("Start thread %ld\n", (long int)threadid);
    int i;
    for(i = 0; i < 100000; i++) {
        count ++;
        printf("%ld, %d\n", (long int)threadid, count);
    }
    printf("End thread %ld\n", (long int) threadid);

    pthread_exit(NULL);
}
```

```
load count
count = count + 1
store count
```

Mutual Exclusion

- Given:
  - A set of n threads: T0, T1, … Tn
  - A set of resources shared between threads
  - A segment of code which accesses the share resources, called the critical section (CS)
- We want to ensure that:
  - Only one thread at a time can execute in the critical section
  - All other threads are forced to wait on entry
  - When a thread leaves the CS, another can enter

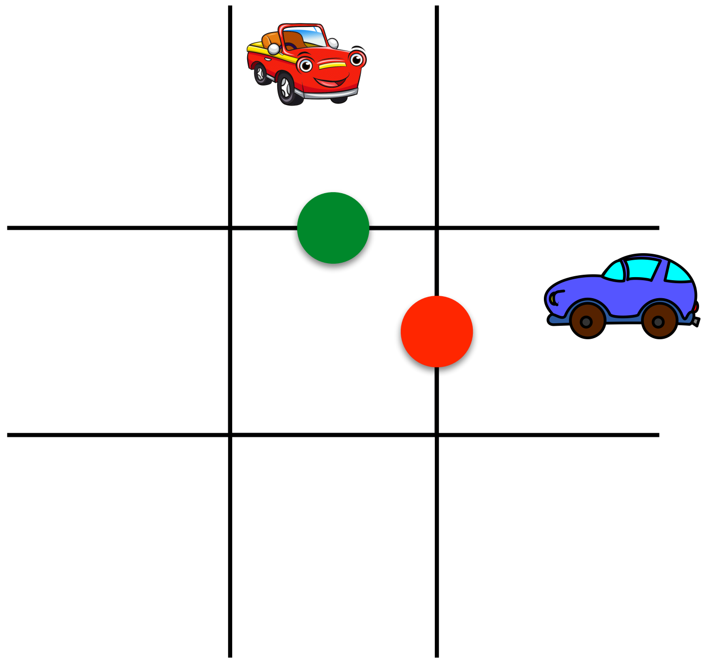
Critical Section Requirements

1. **Mutual Exclusion**
    - If one thread is in the CS, then no other is
2. **Progress**
    - If no thread is in the CS, and some threads want to enter CS, only threads trying to get into the CS section can influence the choice of which thread enters next, and choice cannot be postponed indefinitely
3. **Bounded waiting (no starvation)**
    - If some thread T is waiting on the CS, then there is a limit on the number of times other threads can enter CS before this thread is granted access

- Performance
    - The overhead of entering and exiting the CS is small with respect to the work being done within it

Progress?

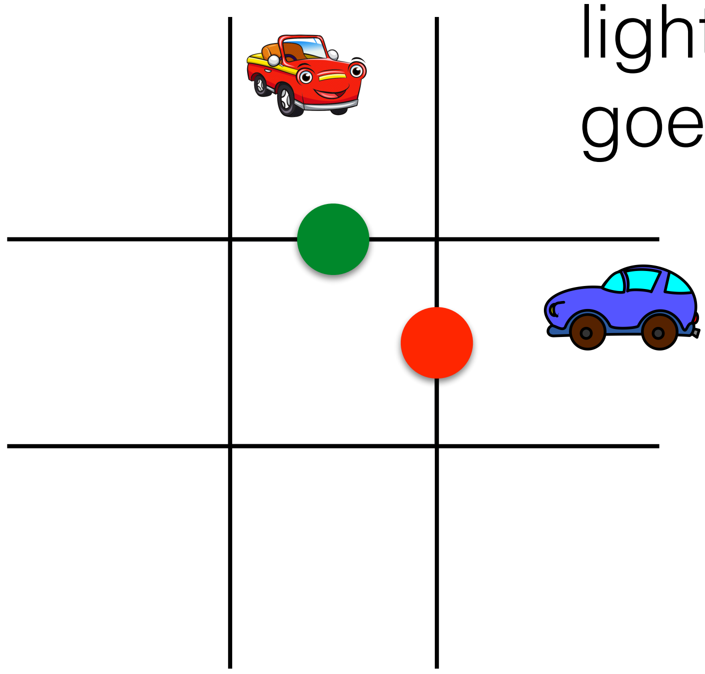
A car may only go through the intersection if the light is green



Progress?

A car may only go through the intersection if the light is green

So this scenario is fine because the red car will go through, light changes, blue car goes through

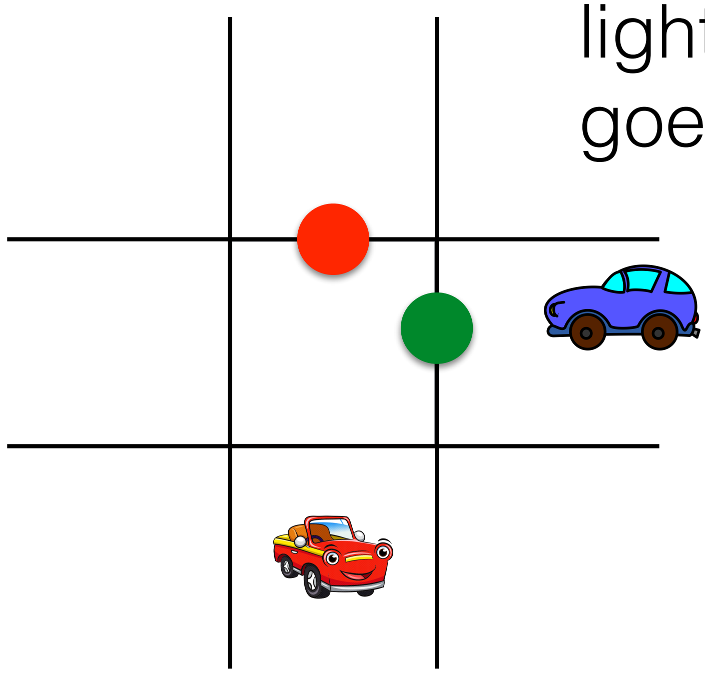


The light changes only after one car has gone through a green light.

Scenario 1

A car may only go through the intersection if the light is green

So this scenario is fine because the red car will go through, light changes, blue car goes through



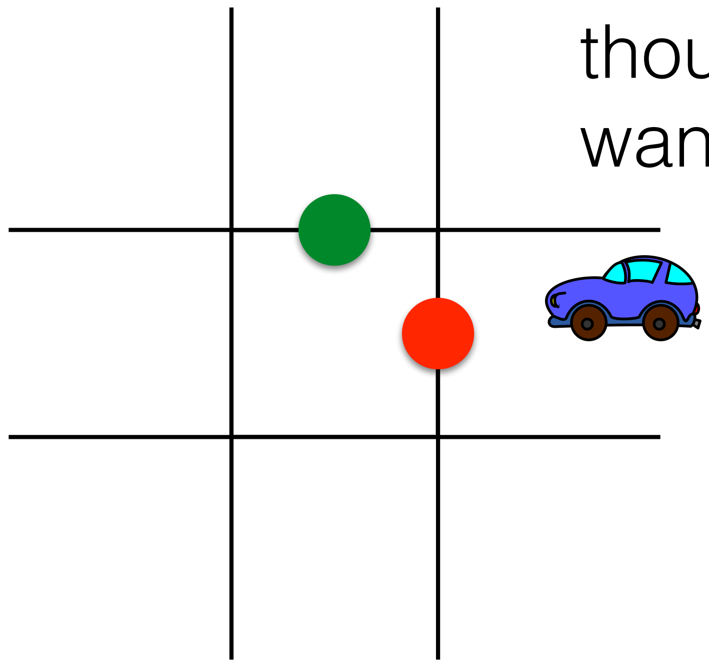
The light changes only after one car has gone through a green light.

Scenario 2

A car may only go through the intersection if the light is green

What if no car goes through to cause the light to change?

The blue car cannot make progress, even though no other car wants to go through.



The light changes only after one car has gone through a green light.

Some Assumption and Notation

- Assume no special hardware instructions (no H/W support)
- Assume no restrictions on the # of processors (for now)
- Assume that basic machine language instructions (LOAD, STORE, etc.) are atomic:
  - If two such instructions are executed concurrently, the result is equivalent to their sequential execution in some unknown order
  - On modern architectures, this assumption may be false
- Let's consider a simple scenario: only 2 threads, numbered T_0 and T_1
  - Use T_i to refer to one thread, T_j for the other (j=1-i) when the exact numbering doesn't matter
- Let's look at one solution… [Exercise]

2-thread solutions: Take 1

- Let the threads share an integer variable `turn` initialized to 0 (or 1)
- If `turn=i` , thread T_i is allowed into its CS

```
My_work(id_t id) {        /* id can be 0 or 1 */
  ...
  while (turn != id) ;    /* entry section */
  /* critical section, access protected resource */
  turn = 1 - id;          /* exit section */
  ...                     /* remainder section */
```

✓ Only one thread at a time can be in its CS

✗ Progress is not satisfied

- Requires strict alternation of threads in their CS:
  if `turn=0`, T_1 may not enter, even if T_0 is in the remainder section

2-thread solutions: Take 1

- First attempt does not have enough info about state of each process. It only remembers which process is allowed to enter its CS
- Replace turn with a shared flag for each thread
    - `boolean flag[2] = {false, false}`
    - Each thread may update its own flag, and read the other thread's flag
    - If `flag[i]` is true, T_i is ready to enter its CS
- Exercise ..

Closer Look at Second Attempt

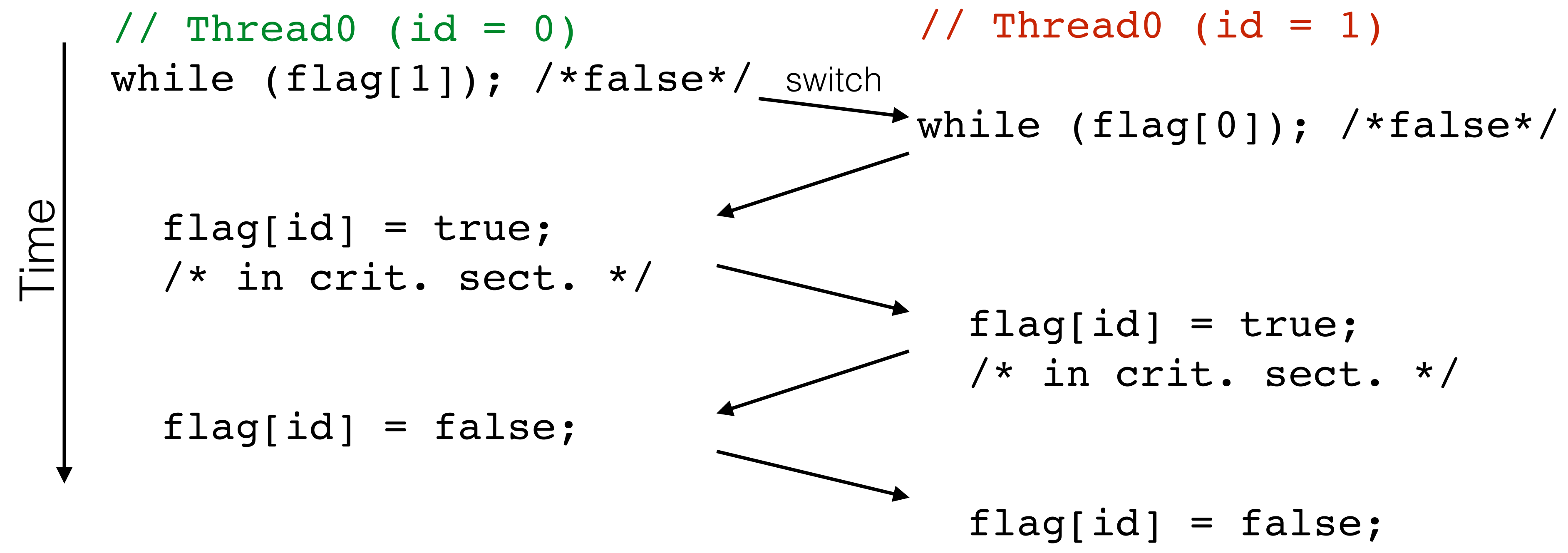
```
My_work(id_t id) {        /* id can be 0 or 1 */
  ...
  while (flag[1-id]) ;    /* entry section */
  flag[id] = true;        /* indicate entering CS */
  /* critical section, access protected resource */
  flag[id] = false;       /* exit section */
  ...                     /* remainder section */
```

- Progress guaranteed?
- Starvation?

❌Mutual exclusion is not guaranteed

- Each thread executes while statement, finds flag set to false
- Each thread sets own flag to true and enters CS

Example Execution Sequence



```
// Thread0 (id = 0)
while (flag[1]); /*false*/

   flag[id] = true;
   /* in crit. sect. */

   flag[id] = false;
```

```
// Thread0 (id = 1)
while (flag[0]); /*false*/

   flag[id] = true;
   /* in crit. sect. */

   flag[id] = false;
```

- Can’t fix this by changing order of testing and setting flag variables (leads to deadlock)

2-Thread Solutions: Take 3

- Combine key ideas of first two attempts for a correct solution
- The threads share the variables turn and flag (where *flag* is an array, as before)
- Basic idea:
    - Set own flag (indicate interest) and set turn to self
    - Spin waiting while turn is self AND other has flag set (is interested)
    - If both threads try to enter their CS at the same time, turn will be set to both 0 and 1 at roughly the same time. Only one of these assignments will last. The final value of turn decides which of the two threads is allowed to enter its CS first.
- This is the basis of Dekker's Algorithm (1965) and Peterson's Algorithm (1981) - Modern OS book, 4th Ed. (A. Tanenbaum) – Fig 2.24, p 125

Peterson's Algorithm

```
int turn;
int flags[2];  /* Shows "interest" in the CS,
                  Are both initially 0, aka false. */

My_work(id_t id) {       /* id can be 0 or 1 */
  ...
  flag[id] = true;       /* entry section */
  turn = id;
  while (turn == id && flag[1-id]) ;
  /* critical section, access protected resource */
  flag[id] = false;      /* exit section */
  ...                    /* remainder section */
}
```

Convince yourself that this works.

Multiple Threads Solution

- Peterson’s Algorithm can be extended to N threads
- Another approach is Lamport’s Bakery Algorithm
  - Upon entering each customer (thread) gets a #
  - The customer with the lowest number is served next
  - No guarantee that 2 threads do not get same #
    - In case of a tie, thread with the lowest id is served first
    - Thread ids are unique and totally ordered
- Mutual exclusion? Progress guaranteed? Starvation?

What multithreaded program feels like…

Source: 9gag.com

Primitive Locks

- With hardware support (details next class) we can create simple locks
    - spin locks are one variation

Using locks

```
Withdraw(acct, amt) {
     acquire(lock);
     balance = get_balance(acct);
     balance = balance - amt;
     put_balance(acct,balance);
     release(lock);
     return balance;
}
```

```
Deposit(account, amount) {
     acquire(lock);
     balance = get_balance(acct);
     balance = balance + amt;
     put_balance(acct,balance);
     release(lock);
     return balance;
}
```

Possible Schedule

```
acquire(lock);
balance = get_balance(acct);
balance = balance - amt;
```

```
acquire(lock);
```

```
put_balance(acct,balance);
release(lock);
```

```
balance = get_balance(acct);
balance = balance + amt;
put_balance(acct,balance);
release(lock);
```