Lecture 3:

Synchronization: Semaphores & Locks
Assignment 1

- Remember: start early!

Procrastinator?
No.
I save all of my work until the last minute because then I’ll be older, therefore wiser.
This Week

- Critical Section Problem

- Synchronization Primitives
  - Semaphores
    - Basic but flexible, Easy to understand
    - Can be hard to program with
  - Synchronization Hardware
  - Spinlocks
    - Very primitive, minimal semantics
Brief preview of 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

- Processes (and threads) interact in a multiprogrammed system
  - To share resources (such as shared data)
  - To coordinate their execution
- 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 of Synchronization

• Two main uses:

  1. Enforce single use of a shared resource
     • Called the critical section problem
     • E.g. using a lock to ensure only one thread can print output to console at a time
        • T1: printf("Hello");  T2: printf("Goodbye");
        ➢ Result should be “HelloGoodbye” OR “GoodbyeHello”, but never “HeGooldbloye” or some other mixture

  2. Control order of thread execution
     • E.g. parent waits for child to finish
        • 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:

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

```c
Deposit(acct, amt) {
    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 co-account holder deposits $100
Example Continued

• We can represent this situation by creating separate threads for each action, which may run at the bank’s central server:

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

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

• What’s wrong with this implementation?
  • Think about potential schedules for these two threads
Interleaved Schedules

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

**Schedule A**

\[
\begin{align*}
\text{balance} &= \text{get\_balance}(\text{acct}) ; \\
\text{balance} &= \text{balance} - \text{amt} ; \\
\text{balance} &= \text{get\_balance}(\text{acct}) ; \\
\text{balance} &= \text{balance} + \text{amt} ; \\
\text{put\_balance}(\text{acct}, \text{balance}) ;
\end{align*}
\]

**Schedule B**

\[
\begin{align*}
\text{balance} &= \text{get\_balance}(\text{acct}) ; \\
\text{balance} &= \text{balance} - \text{amt} ; \\
\text{balance} &= \text{get\_balance}(\text{acct}) ; \\
\text{balance} &= \text{balance} + \text{amt} ; \\
\text{put\_balance}(\text{acct}, \text{balance}) ;
\end{align*}
\]

- 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
    • Aka 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 behavior
  ➔ We need *synchronization*
Caution!

• Bank account problem can occur even with a simple shared variable, even on a uniprocessor:
  • $T_1$ and $T_2$ share variable $X$
  • $T_1$ increments $X$ ($X := X + 1$)
  • $T_2$ decrements $X$ ($X := X - 1$)
  • But at the machine level, we have:

    $T_1$: LOAD X
    INCR
    STORE X

    $T_2$: LOAD X
    DECR
    STORE X

• => Same problem of interleaving can occur!
Aside: What program data is shared?

That is, by threads in the same address space…

- Local variables are not shared (*private*)
  - Each thread has its own stack
  - Local vars are allocated on this private stack
  - Never pass/share/store a pointer to a local variable on another thread’s stack!

- Global variables and static objects are *shared*
  - Stored in the static data segment, accessible by any thread

- Dynamic objects and other heap objs are *shared*
  - Allocated from heap with malloc/free (or new/delete, etc.)
Mutual Exclusion

Given:

- A set of $n$ threads, $T_0, T_1, \ldots, T_{n-1}$
- A set of resources shared between threads
- A segment of code which accesses the shared 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
The Critical Section Problem

- Design a protocol that threads can use to cooperate
  - Each thread must request permission to enter its CS, in its entry section
  - CS may be followed by an exit section
  - Remaining code is the remainder section

- Each thread is executing at non-zero speed
  - no assumptions about relative speed
Critical Section Requirements

1) Mutual Exclusion
   - If one thread is in the CS, then no other is

2) Progress
   - Only threads not in the “remainder” section can influence the choice of which thread enters next, and choice cannot be postponed indefinitely
   - In other words: If no thread is in the CS, and some threads want to enter CS, they should be able to, unrestricted by threads in the “remainder”

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
Some Assumptions & 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 $\text{turn}$ initialized to 0 (or 1)

- If $\text{turn}=i$, thread $T_i$ is allowed into its CS

```c
My_work(id_t id) { /* id_t 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 $\text{turn}=0$, $T_1$ may not enter, even if $T_0$ is in the remainder section
2-Thread Solutions: Take 2

• 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 ..
A Closer Look at 2\textsuperscript{nd} 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 \textit{while} statement, finds \textit{flag} set to false
  • Each thread sets own \textit{flag} to \textit{true} and enters CS
Example Execution Sequence

- Thread0 (id = 0)

  \[
  \text{while } (\text{flag}[1])
  \]
  
  /*false*/

  \[
  \text{flag}[\text{id}] = \text{true};
  \]

  /* in crit. sect. */

- Thread1 (id = 1)

  \[
  \text{while } (\text{flag}[0])
  \]
  
  /*false*/

  \[
  \text{flag}[\text{id}] = \text{true};
  \]

  /* in crit. sect. */

  \[
  \text{flag}[\text{id}] = \text{false};
  \]

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

```c
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 */
    ... /* id can be 0 or 1 */
    flag[id] = true;
    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-Thread Solutions

- 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?
How multithreaded programming feels like ...

Multithreaded programming

Theory

Actual

Source: 9gag.com