What program data is shared between threads?

- 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

TODAY:

- Synchronization
Motivating Example

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

\[\text{Withdraw(acct, amt)} \{
    \text{balance} = \text{get_balance(acct)};
    \text{balance} = \text{balance} - \text{amt};
    \text{put_balance(acct, balance)};
    \text{return balance};
\}\]

\[\text{Deposit(acct, amount)} \{
    \text{balance} = \text{get_balance(acct)};
    \text{balance} = \text{balance} + \text{amt};
    \text{put_balance(acct, balance)};
    \text{return balance};
\}\]

• Idea: Create separate threads for each action, which may run at the bank’s central server
  • 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

\[\text{balance} = \text{get_balance(acct)};\]
\[\text{balance} = \text{balance} - \text{amt};\]
\[\text{context switch};\]
\[\text{balance} = \text{get_balance(acct)};\]
\[\text{balance} = \text{balance} + \text{amt};\]
\[\text{context switch};\]
\[\text{put_balance(acct, balance)};\]

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

Schedule B

\[\text{balance} = \text{get_balance(acct)};\]
\[\text{balance} = \text{balance} + \text{amt};\]
\[\text{context switch};\]
\[\text{balance} = \text{get_balance(acct)};\]
\[\text{balance} = \text{balance} - \text{amt};\]
\[\text{context switch};\]
\[\text{put_balance(acct, balance)};\]

• 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 behavior
  - We need synchronization

Example continued …

- Could the same problem occur with a simple shared variable:
  - \( T_1 \) and \( T_2 \) share variable \( X \)
  - \( T_1 \) increments \( X \) \((X := X+1)\)
  - \( T_2 \) decrements \( X \) \((X := X-1)\)
  - At the machine level, we have:

```plaintext
T_1: LOAD X INCR STORE X
T_2: LOAD X DECR STORE X
```

- Same problem of interleaving can occur!

Mutual Exclusion

- Given:
  - A set of \( n \) threads, \( T_0, T_1, \ldots, T_n \)
  - 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

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

- 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
Critical Section Requirements (1)

- Design a protocol that threads can use to cooperate

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

Critical Section Requirements (2)

- Design a protocol that threads can use to cooperate

2) Progress
   - If no thread is in the CS, and some threads want to enter CS, only threads not in the "remainder" section can influence the choice of which thread enters next, and choice cannot be postponed indefinitely

Critical Section Requirements (3)

- Design a protocol that threads can use to cooperate

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

Critical Section Requirements (4)

- Design a protocol that threads can use to cooperate

4) Performance
   - The overhead of entering and exiting the CS is small with respect to the work being done within it
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 not in the “remainder” 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
4) 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 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
- If only two threads, we number them $T_0$ and $T_1$
  • Use $T_i$ to refer to one thread, $T_j$ for the other ($j=1-i$)
  • Let’s look at one solution...

2-Thread Solutions: 1st Try

- Let the threads share an integer variable turn initialized to 0 (or 1)
- If $turn=i$, 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 */
    }
    /* end of critical 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 $T_2$ is in the remainder section

2-Thread Solutions: 2nd Try

- 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

```c
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
A Closer Look at 2nd Attempt

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

2-Thread Solutions: 3rd Try

- 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)

```
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 */
}
```

```
Enter_region(id_t id) { /* id can be 0 or 1 */
    flag[id] = true; /* indicate entering CS */
    turn = id;
    while (turn == id && flag[other] == true);
}
```

```
Leave_region(id_t id) { /* id can be 0 or 1 */
    flag[id] = false;
}
```

```
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 */
}
```

```
Enter_region(id_t id) { /* id can be 0 or 1 */
    flag[id] = true; /* indicate entering CS */
    turn = id;
    while (turn == id && flag[other] == true);
}
```

```
Leave_region(id_t id) { /* id can be 0 or 1 */
    flag[id] = false;
}
```

2-Thread Solutions: 3rd Try

- Imagine two threads i and j execute `Enter_region()` at the same time:

<table>
<thead>
<tr>
<th>Thread i</th>
<th>Thread j</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flag[i] = true;</code></td>
<td><code>flag[j] = true;</code></td>
</tr>
<tr>
<td><code>while(turn==i &amp;&amp; flag[j]==true);</code></td>
<td><code>while(turn==j &amp;&amp; flag[i]==true);</code></td>
</tr>
</tbody>
</table>

- Basic idea: if both try to enter at the same time, `turn` will be set to both 0 and 1 at roughly the same time. Only one assignments will last. The final value of `turn` decides who gets to go first.
- This is the basis of Peterson’s Algorithm

Higher-level Abstractions for CS’s

- Locks
  - Very primitive, minimal semantics
- Semaphores
  - Basic, easy to understand, hard to program with
- Monitors
  - High-level, ideally has language support (Java)
Synchronization Hardware

- To build these higher-level abstractions, it is useful to have some help from the hardware
- On a uniprocessor:
  - disable interrupts before entering critical section
  - prevents context switches
  - Doesn’t work on multiprocessor
- Need some special atomic instructions

Atomic Instructions: Test-and-Set

- Test-and-set uses a lock variable
  - Lock == 0 => nobody is using the lock
  - Lock == 1 => lock is in use
  - In order to acquire lock, must change it’s value from 0->1

```java
boolean test_and_set(boolean *lock)
{
    boolean old = *lock;
    *lock = True;
    return old;
}
```

- Hardware executes this atomically!

A Lock Implementation

- There are two operations on locks: acquire() and release()

```java
boolean lock;
void acquire(Boolean *lock) {
    while(test_and_set(lock));
};
void release(Boolean *lock) {
    *lock = false;
};
```

- This is a spinlock
  - Uses busy waiting - thread continually executes while loop in acquire(), consumes CPU cycles

Atomic Instructions: Test-and-Set

- The semantics of test-and-set are:
  - Record the old value of the variable
  - Set the variable to some non-zero value
  - Return the old value

```java
boolean test_and_set(boolean *lock)
{
    boolean old = *lock;
    *lock = True;
    return old;
}
```

- lock is always True on exit from test-and-set
  - Either it was True (locked) already, and nothing changed
  - or it was False (available), but the caller now holds it
- Return value is either True if it was locked already, or False if it was previously available
Using Locks

Function Definitions

<table>
<thead>
<tr>
<th>Withdraw(acct, amt)</th>
<th>Deposit(account, amount)</th>
</tr>
</thead>
<tbody>
<tr>
<td>acquire(lock);</td>
<td>acquire(lock);</td>
</tr>
<tr>
<td>balance = get_balance(acct);</td>
<td>balance = get_balance(acct);</td>
</tr>
<tr>
<td>balance = balance - amt;</td>
<td>balance = balance + amt;</td>
</tr>
<tr>
<td>put_balance(acct, balance);</td>
<td>put_balance(acct, balance);</td>
</tr>
<tr>
<td>release(lock);</td>
<td>release(lock);</td>
</tr>
<tr>
<td>return balance;</td>
<td>return balance;</td>
</tr>
<tr>
<td>acquire(lock);</td>
<td>acquire(lock);</td>
</tr>
<tr>
<td>balance = get_balance(acct);</td>
<td>balance = get_balance(acct);</td>
</tr>
<tr>
<td>balance = balance - amt;</td>
<td>balance = balance + amt;</td>
</tr>
<tr>
<td>put_balance(acct, balance);</td>
<td>put_balance(acct, balance);</td>
</tr>
<tr>
<td>release(lock);</td>
<td>release(lock);</td>
</tr>
<tr>
<td>balance = get_balance(acct);</td>
<td>balance = get_balance(acct);</td>
</tr>
<tr>
<td>balance = balance + amt;</td>
<td>balance = balance + amt;</td>
</tr>
<tr>
<td>put_balance(acct, balance);</td>
<td>put_balance(acct, balance);</td>
</tr>
<tr>
<td>release(lock);</td>
<td>release(lock);</td>
</tr>
<tr>
<td>return balance;</td>
<td>return balance;</td>
</tr>
</tbody>
</table>

Possible schedule

| acquire(lock); | acquire(lock); |
| put_balance(acct, balance); | put_balance(acct, balance); |
| release(lock); | release(lock); |

Semaphores

- Integer variable count with two **atomic** operations
  - Operation **wait** (also called P or decrement)
    - block until count > 0 then decrement variable
      ```
      wait(semaphore *s) {
        while (s->count == 0) ;
        s->count -= 1;
      }
      ```
  - Operation **signal** (also called V or increment)
    - increment count, unblock a waiting thread if any
      ```
      signal(semaphore *s) {
        s->count += 1;
        ... //unblock one waiter
      }
      ```
- A queue of waiting threads

Higher-level Abstractions for CS’s

- **Locks**
  - Very primitive, minimal semantics
  - Operations: acquire(lock), release(lock)
- **Semaphores**
  - Basic, easy to understand, hard to program with
- **Monitors**
  - High-level, ideally has language support (Java)

The readers/writers problem

- An object is shared among several threads
  - Some only read the object, others only write it
  - We can allow multiple concurrent readers
  - But only one writer
  - How can we implement this with semaphores?
Use three variables
- Semaphore \texttt{w\_or\_r} - exclusive writing or reading
  - Think of it as a token that can be held either by the group of readers or by one individual writer.
  - Which thread in the group of readers is in charge of getting and returning the token?
  - "Last to leave the room turns off the light"

\textbf{Multiple readers} or \ldots \textbf{One writer}

\textbf{Shared object}

\textit{The readers/writers problem}

- Use three variables
  - Semaphore \texttt{w\_or\_r} - exclusive writing or reading
  - \texttt{int readcount} - number of threads reading object
  - Needed to detect when a reader is the first or last of a group.
  - Semaphore \texttt{mutex} - control access to \texttt{readcount}

\textbf{Writer’s operation:}

```c
//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
    wait(w_or_r); //lock out others
    Write;
    signal(w_or_r); //up for grabs
}
```

\textbf{Reader’s operation:}

```c
Reader {
    wait(mutex); //lock readcount
    // one more reader
    readcount += 1;
    //Update read_count
    signal(mutex);
    //lock readcount
    readcount -= 1;
    if(readcount == 0)
        signal(w_or_r);
    }
```
Reader’s operation:

Reader {
    wait(mutex); //lock readcount
    // one more reader
    readcount += 1;
    // is this the first reader?
    if(readcount == 1)
        //sync w/ writers
        wait(w_or_r);
        //unlock readcount
        signal(mutex);
    Read;
}

Reader’s operation:

Reader {
    wait(mutex); //lock readcount
    // one more reader
    readcount += 1;
    // is this the first reader?
    if(readcount == 1)
        //sync w/ writers
        wait(w_or_r);
        //unlock readcount
        signal(mutex);
    Read;
    wait(mutex); //lock readcount
    readcount -= 1;
    if(readcount == 0)
        signal(w_or_r);
        signal(mutex);
}

Reader’s and writers operation:

//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
    wait(w_or_r); //lock out others
    Write:
    signal(w_or_r); //up for grabs
}

Reader’s and writers operation:

//number of readers
int readcount = 0;
//mutual exclusion to readcount
Semaphore mutex = 1;
//exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
    wait(w_or_r); //lock out others
    Write:
    signal(w_or_r); //up for grabs
}

Suppose I’m the first reader
arriving while writer is active. What happens?

Suppose I’m the second reader
arriving while writer is active. What happens?
**Reader’s and writers operation:**

```c
// number of readers
int readcount = 0;
// mutual exclusion to readcount
Semaphore mutex = 1;
// exclusive writer or reading
Semaphore w_or_r = 1;

Writer {
    wait[w_or_r]: // lock out others
    Write;
    signal[w_or_r]: // up for grabs
}

Reader {
    wait(mutex): // lock readcount
    // one more reader
    readcount += 1;
    // is this the first reader?
    if(readcount == 1)
        // sync w/ writers
        wait[w_or_r];
    // unlock readcount
    signal(mutex);
    Read;
    wait(mutex);
    // lock readcount
    readcount -= 1;
    if(readcount == 0)
        signal[w_or_r];
    signal(mutex);
}
```

Once the writer exits, which reader gets to go first?

If both readers and writers are waiting, once the writer exits, who goes first?

---

**Notes on Readers/Writers**

- If there is a writer
  - First reader blocks on w_or_r
  - All other readers block on mutex
- Once a writer exits, all readers can proceed
  - Which reader gets to go first?
- The last reader to exit signals a waiting writer
  - If no writer, then readers can continue
- If readers and writers are waiting on w_or_r, and a writer exits, who goes first?
  - Depends on the scheduler

---

**Higher-level Abstractions for CS’s**

- **Locks**
  - Very primitive, minimal semantics
  - Operations: acquire(lock), release(lock)
- **Semaphores**
  - Basic, easy to understand, hard to program with
- **Monitors**
  - High-level, ideally has language support (Java)
Motivation for monitors

- It’s easy to make mistakes with semaphores

```c
Writer {
    wait(w_or_r);
    Write;
    wait(w_or_r);
}
```

```c
Writer {
    signal(w_or_r);
    Write;
    signal(w_or_r);
}
```

Monitor Diagram

- an abstract data type: with restriction that only one process at a time can be active within the monitor
- Local data accessed only by monitor’s procedures
- Process enters monitor by invoking 1 of its procedures
- Other processes that attempt to enter monitor are blocked

Bank example with monitors

Monitor Account {
    int balance;
    void withdraw(int amount){
        balance -= amount;
    }
    void deposit(int amount){
        balance += amount;
    }
    ...
}

Enforcing single access

- A process in the monitor may need to wait for something to happen
- May need to let other process use the monitor
- Provide a special type of variable called a condition
- Operations on a condition variable are:
- wait (suspend the invoking process)
- signal (resume exactly one suspended process)
- if no process is suspended, a signal has no effect
- How does that differ from Semaphore’s wait & signal?
Monitor Diagram

More on Monitors
- If process $P$ executes an $x$ signal operation and $\exists$ a process $Q$ waiting on condition $x$, we have a problem:
  - $P$ is already “in the monitor”, does not need to block
  - $Q$ becomes unblocked by the signal, and wants to resume execution in the monitor
  - but both cannot be simultaneously active in the monitor!

Monitor Semantics for Signal
- Hoare monitors
  - signal() immediately switches from the caller to a waiting thread
  - Need another queue for the signaler, if signaler was not done using the monitor
- Brinch Hansen
  - Signaler must exit monitor immediately
    - i.e. signal() is always the last statement in monitor procedure
- Mesa monitors
  - Signal() places a waiter on the ready queue, but signaler continues inside monitor