Today

• How can I synchronize the execution of multiple threads of the same process?
• Example
• Race condition
• Critical-Section Problem
• Spinlocks
• Semaphors
• Usage
Problem Context

- Multiple threads of the same process have:
  - Private registers and stack memory (the context switching mechanism saves and restores registers when switching from thread to thread)
  - Shared access to the remainder of the process “state”

- Preemptive CPU Scheduling:
  - The execution of a thread is interrupted unexpectedly.

- Multiple cores executing multiple threads of the same process.
Share Counting

- Mr Skroutz wants to count his $1-bills.
- Initially, he uses one thread that increases a variable `bills_counter` for every $1-bill.
- Then he thought to accelerate the counting by using two threads and keeping the variable `bills_counter` shared.
Share Counting

\[
bills\_counter = 0
\]

- **Thread A**
  
  ```python
  while (machine_A_has_bills)
  
  bills_counter++
  
  
  print bills_counter
  ```

- **Thread B**

  ```python
  while (machine_B_has_bills)
  
  bills_counter++
  ```

- What it might go wrong?
Share Counting

- Thread A
  \[ r_1 = \text{bills\_counter} \]
  \[ r_1 = r_1 + 1 \]
  \[ \text{bills\_counter} = r_1 \]

- Thread B
  \[ r_2 = \text{bills\_counter} \]
  \[ r_2 = r_2 + 1 \]
  \[ \text{bills\_counter} = r_2 \]

If \( \text{bills\_counter} = 42 \) what are its possible values after the execution of one A/B loop?
Shared counters

- One possible result: everything works!
- Another possible result: lost update!
  - Difficult to debug
- Called a “race condition”
Race conditions

- Def: a *timing dependent error involving shared state*
  - Whether it happens depends on how threads scheduled
  - In effect, once thread A starts doing something, it needs to “race” to finish it because if thread B looks at the shared memory region before A is done, A’s change will be lost.

- Hard to detect:
  - All possible schedules have to be safe
    - Number of possible schedule permutations is huge
    - Some bad schedules? Some that will work sometimes?
  - They are intermittent
    - Timing dependent = small changes can hide bug
Critical-Section Problem

\[ bills\_counter = 0 \]

- **Thread A**
  - while (my\_machine\_has\_bills)
    - enter critical section
    - \( bills\_counter++ \)
    - exit critical section

- **Thread B**
  - while (my\_machine\_has\_bills)
    - enter critical section
    - \( bills\_counter++ \)
    - exit critical section

\textit{print} \( bills\_counter \)
Critical-Section Problem

- The solution should satisfy:
  - Mutual exclusion
  - Progress
  - Bounded waiting

- Let's find some possible solutions.

- **enter section**
  - **critical section**
  - **exit section**
  - **remainder section**
Share Counting

- Thread A

  enter section

  \[ r1 = \text{bills\_counter} \]

  \[ r1 = r1 + 1 \]

  \[ \text{bills\_counter} = r1 \]

  exit section

- Thread B

  enter section

  \[ r2 = \text{bills\_counter} \]

  \[ r2 = r2 + 1 \]

  \[ \text{bills\_counter} = r2 \]

  exit section

Propose solution.
Does it satisfy mutual exclusion, progress and bounded waiting?
Solution

- LOCK
- A process must acquire a lock to enter a critical section.
- Hardware or Software-based implementation
TestAndSet

- **Hardware** instruction.
- **Test and modify** the content of one word atomically.
- **Spinlock**

```c
boolean TestAndSet(boolean *target){
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```
Share Counting

\[ \text{bills\_counter} = 0 \]
\[ \text{lock} = \text{FALSE} \]

- **Thread A**

  ```
  while (machine\_A\_has\_bills) {
    while (TestAndSet(&lock)) {
    
    }
    bills\_counter++
    lock = FALSE;
  }
  ```

- **Thread B**

  ```
  while (machine\_B\_has\_bills) {
    while (TestAndSet(&lock)) {
    
    }
    bills\_counter++
    lock = FALSE;
  }
  ```

  \textbf{print} \text{bills\_counter}

\begin{verbatim}
boolean TestAndSet(boolean *target) {
  boolean rv = *target;
  *target = TRUE;
  return rv;
}
\end{verbatim}
Swap

- **Hardware** instruction.
- Swap the contents of two *words* atomically.
- **Spinlock**

```c
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
Share Counting

\[ bills\_counter = 0 \]
\[ lock = FALSE \]

### Thread A

```c
while (machine_A_has_bills){
    keyA = TRUE;
    while (keyA == TRUE)
        Swap(&lock, &keyA);
    bills_counter++
    lock = FALSE;
}
```

### Thread B

```c
while (machine_B_has_bills){
    keyB = TRUE;
    while (keyB == TRUE)
        Swap(&lock, &keyB);
    bills_counter++
    lock = FALSE;
}
```

\textbf{print} \: bills\_counter

```c
void Swap (boolean *a, boolean *b){
    boolean temp = *a;
    *a = *b;
    *b = temp;
}
```
But...

- TestAndSet and Swap do not satisfy the bounded-waiting requirement.
- Why?
- What is the solution?

- TestAndSet and Swap are complicated for application programmers to use.
- What is the alternative?
Semaphores

- Integer value
- Atomic operations
  - wait
  - signal

```c
wait(S) {
    while S <= 0
        ;
    S--;
}

signal(S) {
    S++;
}
```
Share Counting

\[ \text{bills}_\text{counter} = 0 \]
\[ S = 1 \]

- Thread A
  
  ```
  while (machine_A_has_bills){
    wait(S);
    bills_counter++
    signal(S);
  }
  ```

- Thread B
  
  ```
  while (machine_B_has_bills){
    wait(S);
    bills_counter++
    signal(S);
  }
  ```

print \text{bills}_\text{counter}

```c
wait(S) {
  while S <= 0 
  ;
  S--;
}
```
Spinlock

- This implementation of Semaphors, TestAndSet and Swap require **busy waiting**.
- The waiting processes should **loop continuously** in the entry code.
- Valuable **CPU** cycles are **wasted**.

**Solution:**
- **Block** the waiting process.
- **Signal** blocked process when the semaphore is “available”.
Semaphores

typedef struct {
    int value;
    struct process *list;
} semaphore;

wait (semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal (semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
Usage

- Binary semaphore (mutex)
  - Ranges between 0 and 1.
  - Ex. Only one process can access a resource.
- Counting semaphore
  - Ranges, usually, between 0 and N.
  - Ex. N resources and M processes that share the resources.
- Synchronization
  - Ranges between 0 and 1.
  - Ex. Process A should do task At after process B having done task Bt.
Today

• How can I synchronize the execution of multiple threads of the same process?
• Example
• Race condition
• Critical-Section Problem
• Spinlocks
• Semaphors
• Usage