ET0023 Operating Systems

6. Inter-process Communication

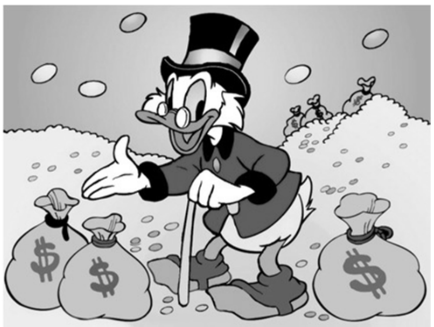
Synchronization

- How can I synchronize the execution of multiple threads of the same process?
  - Problem Definition
    - Race condition
    - Critical-selection problem
  - Solutions
    - Spinlocks
    - Semaphores
  - Usage

Problem Context

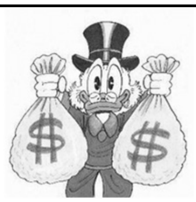
- Multiple threads of the same process have:
  - Private registers and stack memory (the context switching mechanism saves and restores registers when switching from thread to thread)
  - Shared access to the remainder of the process “state”
- Preemptive CPU Scheduling:
  - The execution of a thread is interrupted unexpectedly.
- Multiple cores executing multiple threads of the same process.

Share Counting



Uncle Scrooge is a Disney character that loved money

- Uncle Scrooge wants to count his $1 coins.
- Initially, he uses one thread that increases a variable coin_counter for every $1 coin
- To accelerate the counting, he uses 2 threads but maintains the same variable coin_counter, but shared between the threads.



Share Counting

coin_counter = 0

Thread A

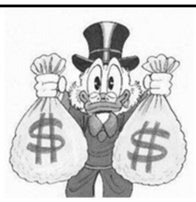
```
while (mc_A_has_coins)
      ++coins_counter ;
```

Thread B

```
while (mc_B_has_coins)
      ++coins_counter ;
```

print coins_counter

- What might go wrong?



Share Counting

Thread A

```
r1 = coins_counter

r1 = r1 + 1

coins_counter = r1
```

Thread B

```
r2 = coins_counter

r2 = r2 + 1

coins_counter = r2
```

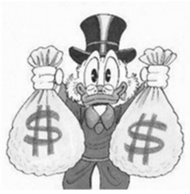
- If coins_counter = 20, what are the possible values after the execution of one A/B loop?

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
- Depends on how threads scheduled
  - Thread A starts
  - Thread A needs to "race" to finish it, because
  - Thread B looks at shared area and changes it
  - Thread 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?
  - Race conditions are intermittent
    - Timing dependent = small changes can hide bug



Share Counting

coin_counter = 0

Thread A

while (my_mc_has_coins)

Enter critical-selection
coins_counter ++
Exit critical-selection

Thread B

while (my_mc_has_coins)

Enter critical-selection
coins_counter ++
Exit critical-selection

print coins_counter

Critical Selection Problem

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

Enter critical-section

Critical section commands

Exit critical-section

Remainder section

Thread A
Enter section
r1 = coins_counter
r1 = r1 + 1
coins_counter = r1
Exit section

Thread B
Enter section
r2 = coins_counter
r2 = r2 + 1
coins_counter = r2
Exit section

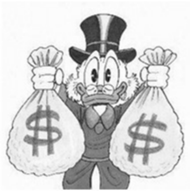
Solution

- LOCK
- A process must acquire a LOCK to enter a critical section.
- Can be implemented using hardware or software

Test-And-Set

- CPU **Hardware** instruction
- Test and modify the content of one word automatically
- Spinlock

```
bool testAndSet(bool *target) {
        bool rv = *target;
        *target = TRUE;
return rv;
}
```



Share Counting

coin_counter = 0

lock = false

Thread A

```
while (my_mc_has_coins)
   while (testAndSet(&lock)) {
           coins_counter ++;
           lock = false;
   }
}
```

Thread B

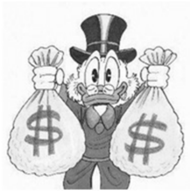
```
while (my_mc_has_coins)
   while (testAndSet(&lock)) {
           coins_counter ++;
           lock = false;
   }
}
```

print coins_counter

Swap

- CPU **Hardware** instruction
- Swap the contents of two **words Automatically**
- Spinlock

```
void swap (bool *a, bool *b) {
       bool temp = *a;
      *a = *b;
      *b = temp;
}
```



Share Counting

coin_counter = 0

lock = false

```
Thread A
 while (my_mc_has_coins)
       keyA = true
       while (keyA == true)
            swap (&lock, &keyA);
       coins_counter ++;
       lock = false;
}
```

```
Thread B
 while (my_mc_has_coins)
       keyB = true
       while (keyB == true)
            swap (&lock, &keyB);
       coins_counter ++;
       lock = false;
}
```

print coins_counter

Problem

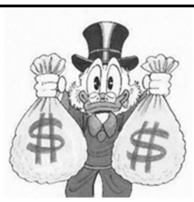
- TestAndSwap and Swap
  - Do not satisfy the bounded-waiting requirement.
  - Are complicated for application programmers to use
- What is the alternative?

Semaphores

- Integer values
- Atomic operations
  - wait()
    - It decrements the value of semaphore variable by 1.
    - If the value becomes negative the process executing wait() is blocked, i.e., added to the semaphore's queue.
  - signal()
    - It increments the value of semaphore variable by 1.
    - After the increment if the value is negative, it transfers a blocked process from the semaphore's queue to the ready queue.

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

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



Share Counting

```
coin_counter = 0
```

```
S = 1
```

Thread A

```
while (my_mc_has_coins)
        wait(S);
        coins_counter ++;
        signal(S);
}
```

Thread B

```
while (my_mc_has_coins)
        wait(S);
        coins_counter ++;
        signal(S);
}
```

```
print coins_counter
```

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
  - E.g. Only one process can access a resource
- Counting semaphore
  - Ranges between 0 and N
  - E.g. N resources and M processes that share the resources
- Synchronization
  - Ranges between 0 and 1
  - E.g. Process A should do task A_t after B having done task B_t

- QUESTIONS