Anthony Ventresque

anthony.ventresque@ucd.ie

Operating Systems

COMP30640

Process Management III
Synchronisation

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Outline

- Concurrent Execution
- Synchronising Concurrent Processes
- Critical Sections
- Mutual Exclusion
- Semaphores

Take home message:

Concurrent access to shared data may result in data inconsistency. Several mechanisms exist to ensure the orderly execution of cooperating processes

Basic Example of Concurrent Execution

```
int A;
```

Process 1

```
A = 1;
if(A == 1)
    printf("Process 1 wins");
```

Process 2

```
A = 2;
if(A == 2)
    printf("Process 2 wins");
```

- Variable A is shared by the two processes
- When the two processes are run concurrently on one processor, which one "wins"?
  - the outcome of the concurrent execution depends on which assignment takes place first (***race condition***)
- Note: the same situation would apply if we would consider two threads of a single process using a shared variable (instead of two processes)

Atomic Operations

- Can we possibly get some value different from either 1 or 2 in A due to concurrent execution
  - if the two instructions “A=1” and “A=2” happened to be executed concurrently?

- NO: ***References and assignments (i.e., read & write operations) are atomic in all CPUs***
  - An atomic operation cannot be interrupted, in order to avoid illogical outcomes

- This basic atomicity is provided by the hardware
  - However higher-level constructs are not atomic in general
  - Higher-level construct: any sequence of two or more CPU instructions

Higher-level Example of Concurrent Execution

```
int B;
```

Process 1

```
B = 0;
while(B < 10) B++;
printf("Process 1 finished");
```

Process 2

```
B = 0;
while(B > -10) B--;
printf("Process 2 finished");
```

- Variable B is shared
  - Post increments/decrements ("B++", "B– –") are atomic
- Are there race conditions in this example?
  - Will the processes finish?
- Issue: "***while***" sections above do not behave atomically
  - ***Process synchronisation is all about getting high-level constructs to behave atomically***

Motivation: “Too Much Milk”

- Two flatmates sharing a fridge
  - Problem: they would like to have at most one bottle of milk in the fridge at any given time

Time	Person A	Person B
3:00	Look in Fridge. Out of milk	
3:05	Leave for store	
3:10	Arrive at store	Look in Fridge. Out of milk
3:15	Buy milk	Leave for store
3:20	Arrive home, put milk away	Arrive at store
3:25		Buy milk
3:30		Arrive home, put milk away

  - A and B must synchronise (cooperate) to achieve their goal
  - Main issue: behaviour of A and B is not atomic

Important Definitions

- ***Race Condition*** (RC): output of a concurrent program depends on the order of operations between process/ threads
- ***Synchronization***: using atomic operations to ensure cooperation between process/threads
- ***Mutual Exclusion*** (ME): ensuring that only one process/thread at a time holds or modifies a shared resource
- ***Critical Section*** (CS): piece of code that only one process/thread can execute at once
  - Critical section is the result of mutual exclusion
  - Critical section and mutual exclusion are two ways of describing the same thing

Locking and mutual Exclusion

- Achieving ME in a CS always involve some sort of ***locking mechanism***
  - ***Locking***: preventing someone else from doing something with the resource shared with them ("monopolise CS for a while")

- Locking involves three rules:
  - Must lock before entering CS
  - Must unlock when leaving CS
  - Must wait if lock is locked when trying to enter CS

Too Much Milk: Solution 1

- Use a note to avoid buying too much milk:
  - Leave a note before buying (kind of “lock”)
  - Remove note after buying (kind of “unlock”)
  - Don’t buy if note (wait)
- Suppose a computer tries this (remember, only memory read/write are atomic):

```
if (noMilk) {
   if (noNote) {
      leave Note;
      buy milk;
      remove note;
   }
}
```

- Result?
  - Still too much milk but only occasionally!
  - Thread can get context switched after checking milk and note but before buying milk!
  - Makes it really hard to debug

Too Much Milk: Solution 2

- As a workaround, let us change the meaning of the note:
- B buys if there is a note and A buys if there is no note

Process A

```
if(no_note) {
    if(no_milk)
        buy milk;
    put note;
}
```

Process B

```
if(note) {
    if(no_milk)
        buy milk;
    remove note;
}
```

- Does this really work?
  - This attempt creates true ME in the CS
  - But assume that B goes on holidays: A will starve
    - Similar issue if B is very slow
    - The relative speed of the processes is an issue
    - Processes must take turns to be in the CS (i.e. the same flatmate cannot buy milk twice in a row)

Too Much Milk: Solution 3

- In order to try to avoid the last issue, let us use ***two notes*** (***note A*** and ***note B***) and some basic courtesy protocol

Process A	Process B
put note_A;	put note_B;
if(no_note_B) {	if(no_note_A) {
if(no_milk)	if(no_milk)
buy milk;	buy milk;
}	}
remove note_A;	remove note_B;

- Each process can now examine each other's status, but not modify it. Does this solution work?
  - Better than before: there is ME *and* none of the processes starves if the other one goes on holidays
  - But relative speed still an issue: we need to decide who buys when both of them leave notes simultaneously. . .

Too Much Milk: Solution 4

- Here is a possible two-note solution:

Thread A

```
leave note A;
while (note B) {\\X
   do nothing;
}
if (noMilk) {
   buy milk;
}
remove note A;
```

Thread B

```
leave note B;
if (noNote A) {\\Y
   if (noMilk) {
      buy milk;
   }
}
remove note B;
```

- Does this work? Yes. Both can guarantee that:
  - It is safe to buy, or
  - Other will buy, ok to quit
- At X:
  - if no note B, safe for A to buy,
  - otherwise wait to find out what will happen
- At Y:
  - if no note A, safe for B to buy
  - Otherwise, A is either buying or waiting for B to quit

Solution 4 Discussion

- Protects a single "Critical-Section" piece of code for each thread:

```
if (no_milk) {
   buy milk;
}
```

- Solution 4 works. It was actually the first solution ever given to the mutual exclusion problem (Dekker), but it is ***not satisfactory***:
  - Really complex – even for this simple an example
    - Hard to convince yourself that this really works
  - A's code is different from B's – what if lots of threads?
    - Code would have to be slightly different for each thread
  - While A is waiting, it is consuming CPU time
    - This is called "busy-waiting"
- There's a better way
  - Have hardware provide better (higher-level) primitives than atomic load and store
  - Build even higher-level programming abstractions on this new hardware support
  - It is necessary to have standard synchronisation mechanisms in an OS, which can automatically fulfill some minimum requirements

Conditions for True Solution to CS Problem (Dijkstra)

1. ***Mutual exclusion***
   - One process at most inside the CS at any time
2. ***Progress***
   - A process in execution out of a CS cannot prevent other processes from entering it
   - If several processes are attempting to enter a CS simultaneously the decision on which one goes in cannot be indefinitely postponed
   - A process may not remain in its CS indefinitely (neither terminate inside it)
3. ***Bounded waiting*** (no starvation)
   - A process attempting to enter its CS will eventually do so

Notes:

- These are ***necessary and sufficient*** conditions, provided that basic operations are atomic
- No assumptions are made about: number of processes, relative speed of processes, or underlying hardware

Desirable Properties of a ME mechanism

- Simple:
  - Systematic and easy to use (e.g., just bracket the CS): avoid unexpected “catches” in purported solutions
  - Easy to maintain
- Efficient:
  - Do not use up substantial amounts of resources when waiting (e.g., no busy-waiting)
  - The overhead due to entering and leaving the CS has to be small, compared to the work done inside it
  - Scalable: it should work when many agents share the CS

Mechanisms for Implementing ME in a CS

Three basic mechanisms:

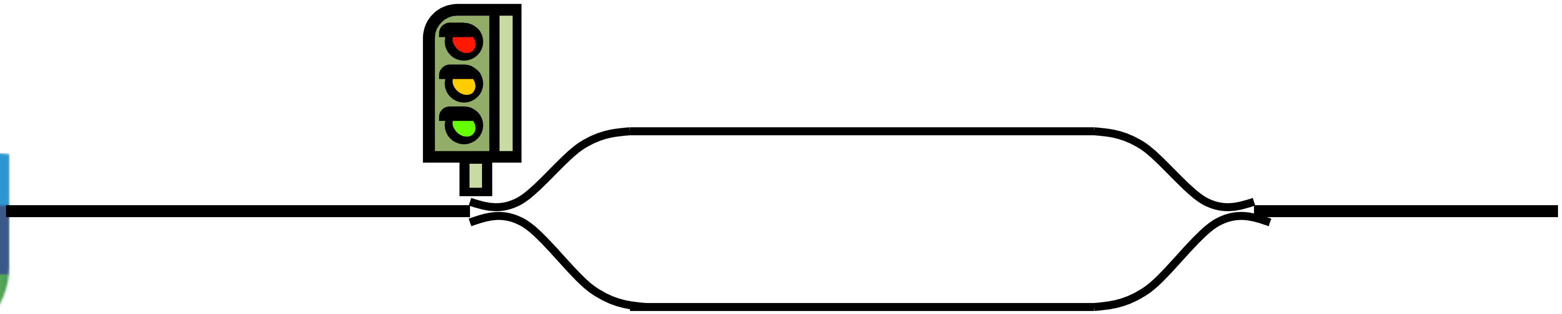
1. Semaphores
   – Simple, but hard to program with (low level)
2. Monitors
   – More abstract, higher level mechanism (language support)
3. Messages
   – Very flexible and simple method of interprocess communication (IPC) & synchronisation

Semaphores

- Semaphores are a kind of generalized lock
  - Main synchronization primitive used in original UNIX
- Definition: a Semaphore has a non-negative integer value and supports the following two operations:
  - ***P()***: an atomic operation that waits for semaphore to become positive, then decrements it by 1
    - Think of this as the wait() operation
  - ***V()***: an atomic operation that increments the semaphore by 1, waking up a waiting P, if any
    - This of this as the signal() operation
- Dijkstra, who first proposed semaphores (1965), was Dutch:
  - ***P****roberen* = to probe
  - ***V****erhogen* = to increment

Semaphores Like Integers Except

- Semaphores are like integers, except
  - No negative values
  - Only operations allowed are P and V – can't read or write value, except to set it initially
  - Operations must be atomic
    - Two P's together can't decrement value below zero
    - Similarly, thread going to sleep in P won't miss wakeup from V – even if they both happen at same time
- Semaphore from railway analogy
  - Here is a semaphore initialized to 2 for resource control:



Semaphores

- Definition: a semaphore is a protected integer variable S with an associated queue of waiting processes, upon which only two atomic operations P() and V() may be performed
- It is represented using an abstract data type which includes:
  - protected integer variable S (i.e., protected counter)
  - queue of waiting processes: processes in the queue are blocked
  - atomic operations:

P(S)

```
if(S > 0)
    S--;
else
    enqueue_calling_proc();
```

V(S)

```
if(queue_not_empty)
    resume_enqueued_proc();
else
    S++;
```

Semaphores and Mutual Exclusion

- A CS may be protected by a semaphore → we may implement ME by means of semaphores; example:
  - Initialise S = 1
  - To enter the CS, execute ***P*** on its semaphore
  - When leaving the CS, execute ***V*** on its semaphore
  - Therefore, two or more processes sharing a CS and this semaphore achieve ME by executing

```
P(S);
  CS
V(S);
```

Semaphores and Mutual Exclusion

- Some remarks:
  - The initial value of S is typically non-negative ($S \geq 0$)
  - If we initialise $S > 1$ more than one process at a time can get into the CS (and thus there is no ME); this is used in a particular type of semaphores called counting semaphores
- Types of semaphores, depending on possible values of S:
  - $S \in \{0, 1\} \rightarrow$ ***binary semaphore*** ($\approx$ mutex)
  - $S \in \{0, 1, 2, 3 \ldots\} \rightarrow$ ***general (or counting) semaphore***

Example (Milk and Semaphores)

- Use of semaphores to solve the “milk problem”

```
semaphore S(1,NULL);
```

processes A & B (in a closed loop)

```
P(S);
  if(no_milk)
      buy milk;
V(S);
```

- Simple, symmetric and efficient solution (compare with the four attempts at obtaining an *ad-hoc* solution)

Conclusion

- Important concept: Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives

- Showed how to protect a critical section with only atomic load and store ⇒ pretty complex!

- Showed several constructions of Locks
  - Must be very careful not to waste/tie up machine resources
    - Shouldn't disable interrupts for long
    - Shouldn't spin wait for long
  - Key idea: Separate lock variable, use hardware mechanisms to protect modifications of that variable

- Semaphores: Like integers with restricted interface
  - Two operations:
    - P(): Wait if zero; decrement when becomes non-zero
    - V(): Increment and wake a sleeping task (if exists)
    - Can initialize value to any non-negative value
  - Use separate semaphore for each constraint