Lecture 3: Intro to Concurrent Processing using Semaphores

- Semaphores;
- The Producer-Consumer problem;
- The Dining Philosophers problem;
- The Readers-Writers Problem:
  - Readers' Preference
  - Passing the Baton
  - Ballhausen's Solution

Semaphores

- Dekker's algorithm solves the mutual exclusion problem on a shared memory machine with no support from the hardware or software.
- Semaphores are a higher level concept than atomic instructions.
- They are atomic actions & usually implemented at OS level
- A semaphore `S` is a non-negative integer variable that has exactly two operations defined for it:

  `P(S)` If `S > 0` then `S = S-1`, otherwise suspend the process.

  `V(S)` If there are processes suspended on this semaphore wake one of them, else `S = S + 1`.
- An important point is that V(S), as it is currently defined, does not specify which of the suspended processes to wake.

Semaphores (cont'd): Semaphore Invariants

- The following invariants are true for semaphores:

$$S \geq 0$$

$$S = S_0 + \#V - \#P$$

where S_0 is the initial value of semaphore S.

Semaphores (cont'd): Mutual Exclusion

- With semaphores, guaranteeing mutual exclusion for *N* processes is trivial:

```
# a semaphore to guarantee mutual exclusion among n processes
sem mutex := 1
const N := 20

process p(i := 1 to N)
   do true ->
       Non_critical_Section
       P(mutex) # grab mutex semaphore
       Critical_Section
       V(mutex) # release mutex semaphore
   od
end
```

Semaphores (cont'd): Proof for Mutual Exclusion

- Theorem: Mutual exclusion is satisfied.
- Proof: Let $\#CS$ be the number of processes in their CS.
- We need to prove that $\#CS + mutex = 1$ is an invariant.

Eqn(1): $\#CS = \#P - \#V$ (from the program structure)

Eqn(2): $mutex = 1 - \#P + \#V$ (semaphore invariant)

Eqn(3): $mutex = 1 - \#CS$ (from (1) and (2))

$\Rightarrow mutex + \#CS = 1$ (from (2) and (3))

QED

Semaphores (cont'd): Proof for Deadlock Avoidance

- Theorem: The program cannot deadlock
- *Proof:* This would require all processes to be suspended in their `P(mutex)` operations.
- Then $mutex = 0$ and $\#CS = 0$ since no process is in its CS.
- The critical section invariant just proven is :

$$mutex + \#CS = 1$$

$\Rightarrow 0 + 0 = 1$ which is impossible.

Types of Semaphores

- Defined above is a general semaphore. A *binary semaphore* is a semaphore that can only take the values 0 and 1.
- Choice of which suspended process to wake gives the following definitions:
  - *Blocked-set semaphore* — Awakens any one of the suspended processes.
  - *Blocked-queue semaphore* — Suspended processes are kept in FIFO & are awakened in order of suspension. This is the type implemented in SR.
  - *Busy-wait semaphore* — The value of the semaphore is tested in a busy wait loop, with the test being atomic. There may be interleavings between loop cycles.

Types of Semaphores: Proofs

- Theorem With busy-wait semaphores, starvation is possible.
- *Proof:* Consider the following execution sequence for 2 processes.

1. P1 executes `P(mutex)` and enters its critical section.
2. P2 executes `P(mutex)`, finds `mutex=0` and loops.
3. P1 finishes CS, executes `V(mutex)`, loops back and executes `P(mutex)` and enters its CS.
4. P2 tests `P(mutex)`, finds `mutex=0`, and loops.

Types of Semaphores: Proofs (cont'd)

1. Theorem With blocked-queue semaphores, starvation is impossible.

- *Proof:*
  - If P1 is blocked on `mutex` there will be at most N-2 processes ahead of P1 in the queue.
  - Therefore after N-2 `V(mutex)` P1 will enter its critical section.

2. Theorem With blocked-set semaphores, starvation is possible for $N \geq 3$.

- Proof:
  - For 3 processes it is possible to construct an execution sequence such that there are always 2 processes blocked on a semaphore.
  - `V(mutex)` is required to only wake one of them, so it could always ignore one and leave that process starved.

The Producer-Consumer Problem

This type of problem has two types of processes:

Producers	processes that, due to some internal activity, produce data to be sent to consumers.
Consumers	processes that on receipt of a data element consume data in some internal computation.

- Could join processes synchronously, such that data is only transmitted when producer is ready to send it & consumer is ready to receive it.
- More flexible to connect producers/consumers by a buffer (ie a *queue*)
- For an infinite buffer then the following invariants hold for the buffer:

$$\#elements \geq 0$$

$$\#elements = 0 + in\_pointer - out\_pointer$$

- These invariants are exactly the same as the semaphore invariants with a semaphore called *elements* and an initial value 0.

The Producer-Consumer Problem (cont'd)

```
var buffer [?]:int
var in_pointer:int := 0,
out_pointer:int := 0
sem elements := 0

process producer
   do true ->
        buffer[in_pointer]:=produce()
        in_pointer:=in_pointer+1
        V(elements)
   od
end
```

```
process consumer
var i:int
   do true ->
        P(elements)
        i:=buffer[out_pointer]
        out_pointer:=out_pointer+1
        consume(i)
   od
end
```

- Can be modified for real bounded circular buffers using another semaphore to count empty places in the buffer.

CA463D Lecture Notes (Martin Crane 2013) 11

The Producer-Consumer Problem (cont'd)

```
const N := 100
var buffer [N]:int
var in_pointer:int := 0, out_pointer:int
:= 0
sem elements := 0
sem spaces := N

process producer
var i:int
   do true ->
      i := produce ()
      P (spaces)
      buffer [in_pointer] := i
      in_pointer:=(in_pointer+1) mod N
      V (elements)
   od
end
```

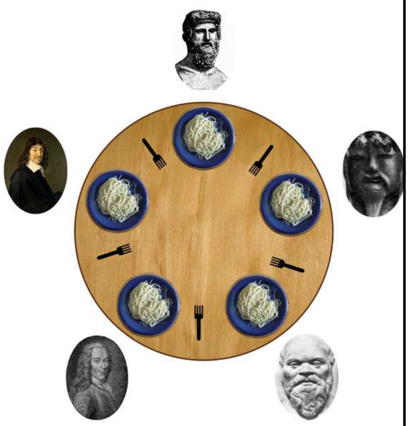
```
process consumer
var i:int
   do true ->
      P (elements)
      i := buffer [out_pointer]
      out_pointer:=(out_pointer+1)mod N
      V (spaces)
      consume (i)
   od
end
```

- As an exercise prove the following:
  (i) No deadlock, (ii) No starvation &
  (iii) No data removal/appending from an empty/full buffer resp.

CA463D Lecture Notes (Martin Crane 2013) 12

The Dining Philosophers Problem

- An institution hires five philosophers to solve a difficult problem.
- Each philosopher only engages in two activities - *thinking* & *eating*.
- Meals are taken in the diningroom which has a table set with five plates & five forks (or five bowls and five chopsticks).
- In the centre of the table is a bowl of spaghetti that is endlessly replenished.
- The philosophers, not being very dextrous, require two forks to eat;
- Philosopher may only pick up the forks immediately to his left right.



Dining Philosophers (cont'd)

- For this system to operate correctly it is required that:

1. A philosopher eats only if he has two forks.
2. No two philosophers can hold the same fork simultaneously.
3. There can be no deadlock.
4. There can be no individual starvation.
5. There must be efficient behaviour under the absence of contention.

- This problem is a generalisation of multiple processes accessing a set of shared resources;
  - e.g. a network of computers accessing a bank of printers.

Dining Philosophers: First Attempted Solution

- Model each fork as a semaphore.
- Then each philosopher must wait (execute a P operation) on both the left and right forks before eating.

```
sem fork [5] := ([5] 1)
# fork is array of semaphores all initialised to have value 1
process philosopher (i := 0 to 4)
    do true ->
            Think ( )
            P(fork [i])                 #grab fork[i]
            P(fork [(i+1) mod 5]        #grab rh fork
            Eat ( )
            V(fork [i])                 #release fork[i]
            V(fork [(i+1) mod 5]        #and rh fork
    od
end
```


Dining Philosphers: Solution #1

- This is called a symmetric solution since each task is identical.
- Symmetric solutions have advantages, e.g. for load-balancing.
- Can prove no fork is ever held by two philosophers since `Eat()` is the CS of each fork. If $\#P_i$ is the number of philosophers holding fork *i* then we have $\boldsymbol{Fork(i) + \#P_i = 1}$

  (ie either philosopher is holding the fork or sem is 1)
- Since a semaphore is non-negative then $\boldsymbol{\#P_i \leq 1}$.
- However, system can deadlock (i.e none can eat) when all philosophers pick up their left forks together;
  - i.e. all processes execute `P(fork[i])` before `P(fork[(i+1)mod 5]`
- Two solutions:
  - Make one philosopher take a right fork first (asymmetric solution);
  - Only allow four philosophers into the room at any one time.

Dining Philosophers: Symmetric Solution

- This solution solves the deadlock problem.
- It is also symmetric (i.e. all processes execute the same piece of code).

```
sem Room := 4
sem fork [5] := ([5] 1)
Process philosopher (i := 0 to 4)
    do true ->
            Think ( ) # thinking not a CS!
            P (Room)
            P(fork [i])
            P(fork [(i+1) mod 5]
            Eat ( )     # eating is the CS
            V(fork [i])
            V(fork [(i+1) mod 5]
            V (Room)
    od
end
```


Dining Philosophers: Symmetric Solution (cont'd) Proof of No Starvation

Theorem Individual starvation cannot occur.

- *Proof:*
  - For a process to starve it must be forever blocked on one of the three semaphores, `Room`, `fork [i]` or `fork [(i+1) mod 5]`.
  - **a)** `Room` semaphore
  - If the semaphore is a blocked-queue semaphore then process `i` is blocked only if `Room` is 0 indefinitely.
  - Requires other 4 philosophers to be blocked on their left forks, since if one of them can get two forks he will finish, put down the forks and signal Room (by `V(Room)`).
  - So this case will follow from the `fork[i]` case.

Dining Philosophers: Symmetric Solution (cont'd) Proof of No Starvation

b) `fork[i]` semaphore

– If philosopher `i` is blocked on his left fork, then philosopher `i-1` must be holding his right fork.

– Therefore he is eating or signalling he is finished with his left fork,

– So will eventually release his right fork (ie philosopher `i`'s left fork).

c) `fork[i+1] mod 5` semaphore

– If philosopher `i` is blocked on his right fork, this means that philosopher (`i+1`) has taken his left fork and never released it.

– Since eating and signalling cannot block, philosopher (`i+1`) must be waiting for his right fork,

– and so must all the others by induction: $i+j, 0 \leq i \leq 4$.

– But with `Room` semaphore invariant only 4 can be in the room,

– So philosopher i cannot be blocked on his right fork.

The Readers-Writers Problem

- Two kinds of processes, readers and writers, share a DB.
- Readers execute transactions that examine the DB, writers execute transactions that examine and update the DB.
- Given that the database is initially consistent, then to ensure that it remains consistent, a writer process must have exclusive access.
- Any number of readers may concurrently examine the DB.
- Obviously, for a writer process, updating the DB is a CS that cannot be interleaved with any other process.

The Readers-Writers Problem (cont'd)

```
const M:int := 20, N:int := 5
var nr:int :=0
sem mutexR := 1
sem rw := 1

process reader (i:= 1 to M)
    do true ->
        P (mutexR)
        nr := nr + 1
        if nr = 1 -> P (rw) fi
        V (mutexR)
        Read_Database ( )
        P (mutexR)
        nr := nr - 1
        if nr = 0 -> V (rw) fi
        V (mutexR)
    od
end

process writer(i:=1 to N)
    do true ->
        P (rw)
        Update_Database ( )
        V (rw)
    od
end
```

- Called the *readers' preference* solution since if some reader is accessing the DB and a reader and a writer arrive at their entry protocols then the readers will always have preference over the writer process.

The Readers-Writers Problem (cont'd)

- The Readers Preference Solution is not a fair one as it always gives readers precedence over writers
- So a continual stream of readers will block any writer process from updating the database.
- To make it fair need to use a *split binary semaphore*, i.e. several semaphores with the property that sum is 0 or 1.
- We also need to count the number of suspended reader processes and suspended writer processes.
- This technique is called *passing the baton*.

Readers/Writers: Passing the Baton

```
const M:int := 20, N:int := 5
sem e:=1,r:=0,w:=0 #0≤(e+r+w)≤ 1

var nr:int :=0, nw:int := 0
var sr:int:=0, sw:int:=0 # no. of
            #suspended readers & writers

process reader (i:= 1 to M)
    do true ->
        P (e)
        if nw > 0 ->
                sr:= sr + 1; V(e); P(r)
        fi
        nr := nr + 1
        if sr > 0 ->
                sr := sr - 1; V (r)
        [] sr = 0 -> V(e)
        fi
        Read_Database ( )
        P (e)
        nr := nr - 1
        if nr = 0 and sw > 0 ->
                sw := sw - 1; V (w)
        [] nr >0 or sw = 0 -> V(e)
        [] sr >0 and nw = 0 -> V(r)
        fi
    od
end

process writer (i:= 1 to N)
    do true ->
        P (e)
        if nr > 0 or nw > 0 ->
                sw:= sw + 1; V(e); P(w)
        fi
        nw := nw + 1
        V (e)
        Update_Database ( )
        P (e)
        nw := nw - 1
        if sr >0 -> sr:= sr-1;V(r)

        [] sw >0 -> sw:= sw-1;V(w)

        [] sr =0 and sw =0 -> V(e)
        fi
    od
end
```


Readers/Writers: Passing the Baton (cont'd)

- Called 'Passing the Baton' because of way signalling takes place (when a process is executing within a CS, it holds the 'baton').
- When that process gets to an exit point from that CS, it 'passes the baton' to some other process.
- If (more than)one process is waiting for a condition that is now true, 'baton is passed' to one such process, randomly.
- If none is waiting, baton is passed to next one trying to enter the CS for the first time, i.e. trying **`P(e)`**.

Passing the Baton (cont'd): Scenarios…

- Suppose a writer is in first….
  - Any readers executing `P(e)` will be suspended in a FIFO queue (`sr:=sr+1`)
  - The writer will finish, execute `P(e)`, decrement `nw` and eventually signal a suspended (or maybe a new) reader who can then increment `nr`, awake the suspended reader..
  - Note that the `if` in SR is non-deterministic (any of the else-if arms (`[]`) which apply can be executed non-deterministically)
- Suppose a reader is first to grab the entry semaphore….
  - More readers can be let in as there are no `sr`'s (`[] sr=0->V(e)`)
  - A writer can come in but is immediately suspended pending the signal from the last reader to exit after reading the database
  - Note: the `if` at the end of process reader is also non-deterministic

Readers-Writers: Ballhausen's Solution

- The idea behind this solution is one of efficiency: one reader takes up the same space as all readers reading together.
- A semaphore `access` is used for readers gaining entry to the DB, with a value initially equalling the total number of readers.
- Every time a reader accesses the DB, the value of `access` is decremented and when one leaves, it is incremented.
- When a writer wants to enter the DB it will occupy all space step by step by waiting for all old readers to leave and blocking entry to new ones.
- The writer uses a semaphore **`mutex`** to prevent deadlock between two writers trying to occupy half of the available space each.

Readers-Writers: Ballhausen's Solution (cont'd)

```
sem mutex = 1
sem access = m

process reader (i = 1 to m)
    do true ->
        P(access)

        # ... reading ...

        V(access)
        # other operations
    od
end

process writer (j = 1 to n)
    do true ->
        P(mutex)
        fa k = 1 to m ->
                P(access)
        af

        #... writing ...

        fa k = 1 to m ->
                V(access)
        af
        # other operations
        V(mutex)
    od
end
```