LECTURE 2: SUPPORT FOR CORRECTNESS IN CONCURRENCY

Intro to Concurrent Processing

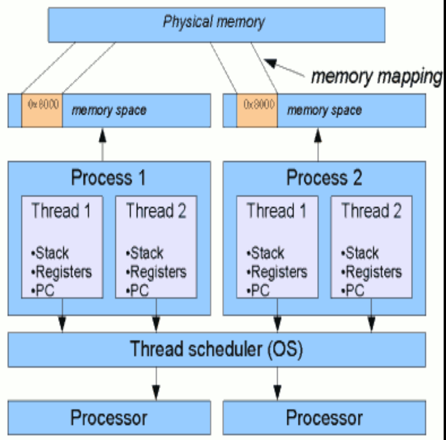
- Recap on Threads and Processes.
- Basic models of correctness in concurrency.
- Software Solutions to Mutual Exclusion.
  - Dekker's Algorithm.
  - Mutual Exclusion for n processes: The Bakery Algorithm.
- Higher level supports for Mutual Exclusion:
  - Semaphores & Monitors
  - Emulating Semaphores with Monitors & Vice Versa
- Solution of Classical Problems of Synchronization:
  - The Readers-Writers Problem
  - The Dining Philosophers problem in SR;
  - The Sleeping Barber Problem;

SECTION 2.0: RECAP & CONCURRENT CORRECTNESS BASICS

Threads/Processes Recap

Introduction to Threads

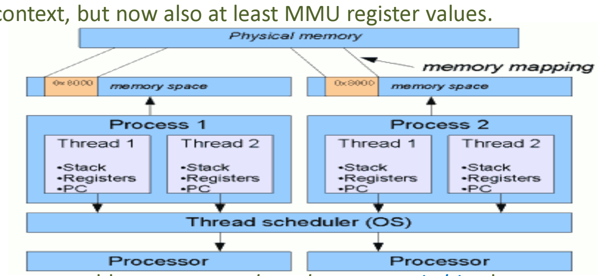
- *Basic idea*: build *virtual* processors in software, on top of *physical* processors:
  - *Processor*:
    - gives set of instructions (with ability to automatically run a series of them).
  - *Thread*:
    - minimal s/w processor in whose context can execute some instructions.
    - save thread context ⇒ stop current run & save all data needed to run later.
  - *Process*:
    - s/w processor in whose context can run one/ more threads .
    - run thread ⇒ run series of instructions in it's context .



Threads/Processes Recap (/2)

Context Switching:

– *Processor context:*
  - minimal value set stored in processor registers to run some instructions, e.g., stack pointer, addressing registers, program counter.
– *Thread context:*
  - minimal value set stored in registers & memory, to run some instructions, i.e., processor context, state.
– *Process context:*
  - minimal value set stored in registers & memory, used to run a thread,
  - i.e., thread context, but now also at least MMU register values.



– *Observations:*
  - threads share same address space ⇒ *thread context switching* happens entirely without OS; *process switching* is generally more expensive OS must get involved.
  - creating & destroying threads is much cheaper than doing so for processes.

Threads/Processes Recap (/3)

Threads and Operating Systems:

– *Main issue:*
  - should OS *kernel* provide threads, or implement them as *user-level* packages?

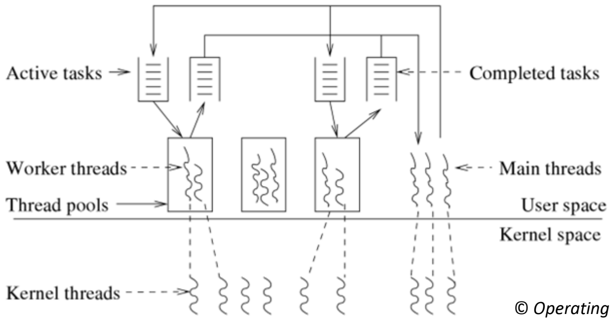
– *User-space solution:*
  - single process handles all operations ⇒implementations can be very efficient.
  - all services provided by kernel are done on behalf of process thread lives in ⇒ if kernel blocks a thread, entire process blocks.
  - use threads for many external events; threads block on a per-event basis ⇒ if kernel can't distinguish them, how can signalling events happen?

– *Kernel solution:*
  - kernel should contain thread package implementation ⇒ all operations (creation, synchronisation) return as system calls
  - operations that block a thread are no longer a problem: kernel schedules another available thread within same process.
  - handling external events is simple: kernel schedules event's thread.
  - *big problem*: efficiency loss as each thread operation needs trap to kernel.

Threads/Processes Recap (/4)

Threads and Operating Systems (cont'd):

– *Conclusion:*
  - Try to mix user-level and kernel-level threads



  - We'll return to *thread pool* abstraction when looking at Java
  - For now, need to ensure threads do not interfere with each other
  - Neatly tees up topic of *Concurrent Correctness*

A Model of Concurrent Programming

- Concurrent code: *interleaving sets of sequential atomic instructions*.
  – i.e. some interacting sequential processes execute simultaneously, on same or different processor(s).
  – processes *interleaved* i.e. at any time each processor runs one of instructions of the sequential processes.
  – relative rate at which steps of each process execute is not important.

- Each sequential process consists of a series of *atomic instructions*.
- *Atomic instruction* is an instruction that once it starts, proceeds to completion without interruption.
- Different processors have different atomic instructions, and this can have a big effect.

A First Attempt to Define Correctness

```
P1:    load reg,   N
P2:    load reg,   N
P1:    add reg,    #1
P2:    add reg,    #1
P1:    store reg,  N
P2:    store reg,  N
```

- If processor has instructions like `INC` this code is correct no matter which instruction is executed first.
- If all math done in registers then results obtained depend on interleaving.
- This dependency on unforeseen circumstances is known as a *Race Condition*
- A concurrent program *must be* correct under all possible *interleaving*s.

Correctness: A More Formal Definition

- *Correctness*:
- If $P(\vec{a})$ is property of input (pre condition), and $Q(\vec{a},\vec{b})$ is a property of input & output (post condition), then correctness is defined as:
  - Partial correctness:
    $$P(\vec{a}) \wedge \text{Terminates}\{Prog(\vec{a},\vec{b})\} \Rightarrow Q(\vec{a},\vec{b})$$
  - Total correctness:
    $$P(\vec{a}) \Rightarrow \left[\text{Terminates}\{Prog(\vec{a},\vec{b})\} \wedge Q(\vec{a},\vec{b})\right]$$
- Totally correct programs terminate. A totally correct specification of the incrementing tasks is:
  $$a \in \mathbb{N} \Rightarrow [\text{Terminates}\{\mathbf{INC}(a,a)\} \wedge a{=}a{+}1]$$

Types of Correctness Properties

There are 2 types of correctness properties:

1. **Safety properties** — These must *always* be true.
   - *Mutual exclusion* — Two processes must not interleave certain sequences of instructions.
   - *Absence of deadlock* — Deadlock is when a non-terminating system cannot respond to any signal.
2. **Liveness properties** — These must *eventually* be true.
   - *Absence of starvation* — Information sent is delivered.
   - *Fairness* — That any contention must be resolved.

Correctness: Fairness

- There are 4 different way to specify *fairness*.
  - *Weak Fairness* — A process continuously requesting eventually has it granted.
  - *Strong Fairness* — If a process makes a request infinitely often, eventually it will be granted.
  - *Linear waiting* — A process requesting, is granted it before another is granted a request > once.
  - *FIFO* — A process making a request is granted it before another one making a later request

SECTION 2.1: MUTUAL EXCLUSION: BASIC SOFTWARE SOLUTIONS

Mutual Exclusion (ME)

- From above, concurrent code must be correct in all allowable interleavings.
- So some (ME)parts of different processes *cannot* be interleaved
- These are called *critical sections*.
- Try solving ME issue with software before advanced solutions

```
// Pseudo Code showing a critical section shared by
// different processes
   while (true)
        // Non_Critical_Section
        // Pre_protocol
        // Critical_Section
        // Post_protocol
   end while
```

Software Solution to Mutual Exclusion Problem # 1

```
/* Copyright © 2006 M. Ben-Ari. */
int turn = 1;

void p()
{
    while (1) {
      cout << "p non-critical section \n";
      while (!( turn == 1 ));
      cout << "p critical section \n";
      turn = 2;
      }
}
```

```
void q()
{
    while (1) {
      cout << "q non-critical section \n";
      while (!(turn == 2));
      cout << "q critical section \n";
      turn = 1;
      }
}
main() {
   cobegin {
     p(); q();
     }
}
```

- This solution satisfies mutual exclusion. ☑
- Cannot deadlock, as both **p,q** would have to loop on **turn** test infinitely and fail.
  - Implies **turn==1** and **turn==2** at the same time.
- No starvation: requires one task to execute its CS infinitely often as other task remains in its pre-protocol.
- Can fail in absence of contention: if p halts in CS, q will always fail in pre-protocol.
- Even if p, q guaranteed not to halt, both are forced to execute at the same rate. This, in general, is not acceptable.

Software Solutions to Mutual Exclusion Problem # 2

```
/* Copyright © 2006 M. Ben-Ari. */

int wantp = 0;
int wantq = 0;

void p()
{
    while (1) {
      cout << "p non-critical section\n";
      while (!(wantq == 0));
      wantp = 1;
      cout << "p critical section\n";
      wantp = 0;
      }
}
```

```
void q()
{
    while (1) {
      cout << "q non-critical section \n";
      while (!(wantp == 0));
      wantq = 1;
      cout << "q critical section \n";
      wantq = 0;
      }
}
main() {
   cobegin {
     p(); q();
     }
}
```

- The first attempt failed because both processes shared the same variable.
- The Second Solution unfortunately violates the mutual exclusion requirement.
- To prove this only need to find one interleaving allowing p & q into their CS at same time.
- Starting from the initial state, we have:

p checks **wantq** and finds **wantq**=0.
p sets **wantp**= 1.
p enters its critical section.

q checks **wantp** and finds **wantp**= 0.
q sets **wantq**= 1.
q enters its critical section.
QED

Software Solutions to Mutual Exclusion Problem # 3

```
/* Copyright © 2006 M. Ben-Ari. */

int wantp = 0;
int wantq = 0;

void p()
{
    while (1) {
  a1  cout << "p non-critical section\n";
  b1  wantp = 1;
  c1  while (!(wantq == 0));
  d1  cout << "p critical section\n";
  e1  wantp = 0;
    }
}

void q()
{
    while (1) {
  a2  cout << "q non-critical section \n";
  b2  wantq = 1;
  c2  while (!(wantp == 0));
  d2  cout << "q critical section\n";
  e2  wantq = 0;
    }
}
main() {
  cobegin {
    p(); q();
  }
}
```

- Problem with #2 is once pre-protocol loop is completed can't stop process from entering CS
- So the pre-protocol loop should be considered as part of the critical section.
- We can prove that the mutual exclusion property is valid. To do this we need to prove that the following equations are *invariants*:

$$\texttt{wantp=1} \equiv at(c_1) \vee at(d_1) \vee at(e_1) \qquad \text{Eqn(1)}$$

$$\texttt{wantq=1} \equiv at(c_2) \vee at(d_2) \vee at(e_2) \qquad \text{Eqn(2)}$$

$$\neg\{at(d_1) \wedge at(d_2)\} \qquad \text{Eqn(3)}$$

(here $at(x) \Rightarrow$ x is the next instruction to be executed in that process.)

Software Solutions # 3 (cont'd)

- Eqn (1) is initially true:
  - Only the $b_1 \rightarrow c_1$ and $e_1 \rightarrow a_1$ transitions can affect its truth.
  - But each of these transitions also changes the value of **wantp**.
- A similar proof is true for Eqn (2).
- Eqn 3 is initially true, and
  - can only be negated by a $c_2 \rightarrow d_2$ transition while $at(d_1)$ is true.
  - But by Eqn (1), $at(d_1) \Rightarrow$**wantp=1**, so $c_2 \rightarrow d_2$ cannot occur since this requires **wantp=0**. Similar proof for process q.
- But there's a problem with deadlock, if the program executes one instruction from each process alternately:

p assigns 1 to **wantp**.	q assigns 1 to **wantq**
p tests **wantq** & remains in its **do** loop	q tests **wantp** & remains in its **do** loop

Result Deadlock!

Software Solutions to Mutual Exclusion Problem # 4

- Problem with third proposed solution was that once a process indicated its intention to enter its CS, it also **insisted** on entering its CS.

- Need some way for a process to relinquish its attempt if it fails to gain immediate access to its CS, and try again.

Software Solutions to Mutual Exclusion Problem # 4

```
/* Copyright © 2006 M. Ben-Ari. */

int wantp = 0;
int wantq = 0;

void p()
{
    while (1) {
      cout << "p non-critical section\n";
      wantp = 1;
      while (wantq == 1) {
         wantp = 0;
         wantp = 1; }
      cout << "p critical section\n";
      wantp = 0;
    }
}
```

```
void q()
{
    while (1) {
      cout << "q non-critical section\n";
      wantq = 1;
      while (wantp == 1) {
         wantq = 0;
         wantq = 1; }
      cout << "q critical section\n";
      wantq = 0;
    }
}
main() {
/* As before */
}
```

- This proposal has two drawbacks:

1. A process can be starved.
   Can find interleavings where a process can never enter its critical section.
2. The program can *livelock* (a form of deadlock).
   In *deadlock* no possible interleaving allows processes into CS.
   In *livelock*, some interleavings succeed, but some sequences don't.

Software Solutions # 4 (cont'd)

Proof of Failure of Attempt 4:

1. By Starvation

p sets **wantp** to 1.

p completes a full cycle:

Checks **wantq** Enters CS
Resets **wantp** Does non-CS
Sets **wantp** to 1

q sets **wantq** to 1
q checks **wantp**, sees **wantp**=1 & resets **wantq** to 0

q sets **wantq** to 1 and back

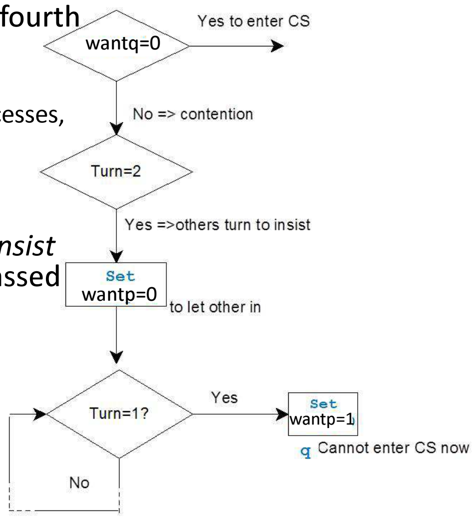
2. By Livelock

p sets **wantp** to 1.
p tests **wantq** , remains in its **do** loop
p resets **wantp** to 0 to relinquish attempt to enter CS
p sets **wantp** to 1

q sets **wantq** to 1
q tests **wantp** , remains in its **do** loop
q resets **wantq** to 0 to relinquish attempt to enter CS
q sets **wantq** to 1
etc

Dekker's Algorithm

- A combination of the first and fourth proposals:
  - First proposal explicitly passed right to enter CSs between the processes,
  - whereas fourth proposal had its own variable to prevent problems in absence of contention.
- In Dekker's algorithm right *to insist* on entering a CS is explicitly passed between processes.



Dekker's Algorithm (cont'd)

```
/* Copyright © 2006 M. Ben-Ari.
*/

int wantp = 0;
int wantq = 0;
int turn = 1;

void p()
{
    while (1) {
      cout << "p non-CS \n";
      wantp = 1;
      while (wantq == 1) {
       wantp = 0;
       while (!(turn == 1));
              wantp = 1; }
      cout << "p CS\n";
      turn = 2;
      wantp = 0;
    }
}
```

```
void q()
{
    while (1) {
      cout << "q non-CS\n";
      wantq = 1;
      while (wantp == 1) {
       wantq = 0;
       while (!(turn == 2));
              wantq = 1; }
      cout << "q CS\n";
      turn = 1;
      wantq = 0;
    }
}
main() {
/* As before */
}
```


Mutual Exclusion for n Processes: The Bakery Algorithm

- Dekker's Algorithm solves mutual exclusion problem for 2 processes.
- Many algorithms solve N process ME problem; all are complicated and relatively slow to other methods.
- *The Bakery Algorithm* is one where processes take a numbered ticket (whose value constantly increases) when it wants to enter its CS.
- The process with the lowest current ticket gets to enter its CS.
- This algorithm is not practical because:
  – ticket numbers will be unbounded if a process is always in its critical section, and
  – even in the absence of contention it is very inefficient as each process must query the other processes for their ticket number.

```
/* Copyright (C) 2006 M. Ben-Ari. */
```

Mutual Exclusion for *N* Processes: The Bakery Algorithm (cont'd)

```
const int NODES = 3;
      int num[NODES];
      int choose[NODES];

  int Max() {
  int Current = 0;
  int i;
    for (i=0; i <NODES; i++)
      if (num[i] > Current) Current = num[i];
    return Current;
  }

    void p(int i) {
    int j;
        while (1)         {
          cout << "proc " << i << " non-CS\n";
          choose[i] = 1;
          num[i]= 1 + Max();
          choose[i] = 0;
          for (j=0; j <NODES; j++)
            if (j != i)         {
              while (!choose[j]);
              while (!
               ((num[j]==0)||(num[i]<num[j])||
               ((num[i]==num[j])&&(i < j))) );
              }
          cout << "process " << i << " CS\n";
          num[i]=0;
          }
    }
```

```
main() {
  int j;
  for (j=0; j <NODES; j++) number[j]=0;
  for (j=0; j <NODES; j++) choose[j]=0;
  cobegin {
    p(0); p(1); p(2); // 3 processes here
  }
}
```


SECTION 2.2: HIGHER LEVEL SUPPORT FOR MUTUAL EXCLUSION: SEMAPHORES & MONITORS

Semaphores

- A more general synchronization mechanism
- Operations: ***P*** (wait) and ***V*** (signal)
- ***P***(S)
  - If semaphore variable S is nonzero, decrements S and returns
  - Else, suspends the process
- ***V***(S)
  - If there are processes blocked for S, restarts exactly one of them
  - Else, increments S by 1
- The following invariants are true for semaphores:

$$S \geq 0 \text{ (1)}$$

$$S = S_0 + \#V - \#P \text{ (2)}$$

where S_0 is initial value of Semaphore S

Semaphores for Mutual Exclusion

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

```
semaphore mutex = 1;

void P (int i) {
while (1) {
       // Non Critical Section Bit
       P(mutex) // grab the mutual exclusion semaphore
       // Do the Critical Section Bit
       V(mutex) //grab the mutual exclusion semaphore
       }
   }

int main ( ) {
       cobegin {
              P(1); P(2);
       }
}
```

Semaphores: Proof of Mutual Exclusion

- Theorem Mutual Exclusion is satisfied
- *Proof:* Let $\#CS$ be the number of processes in their CS
- We need to prove that $mutex + \#CS = 1$ 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: Proof of No Deadlock

Theorem The program cannot deadlock

- *Proof:*
  - Deadlock needs all processes to be suspended on their `P(mutex)` operations.
  - So $mutex = 0$ and $\#CS = 0$ as no process is in its critical section
  - The critical section invariant just proven is

$$mutex + \#CS = 1$$

$$\Rightarrow 0 + 0 = 1$$

which is clearly 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* — Wakes any one suspended process
  - *Blocked-queue semaphore* — Suspended processes are kept in FIFO & woken in order of suspension
  - *Busy-wait semaphore* — semaphore value is tested in a busy-wait loop, with atomic test. Some loop cycles may be interleaved.

Types of Semaphores: Proofs

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

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

Types of Semaphores: Proofs (/2)

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

- *Proof:*
  - If P(1) is blocked on `mutex` there will be at most N-2 processes ahead of P(1) 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 can construct an execution sequence so 2 processes are always blocked on a semaphore.
  - `V(mutex)` only has to wake one, so can always ignore one & let it starve

SECTION 2.3: Ye Classicale Problemes of Synchronization

1. The Producer-Consumer Problem

This type of problem has two types of processes:

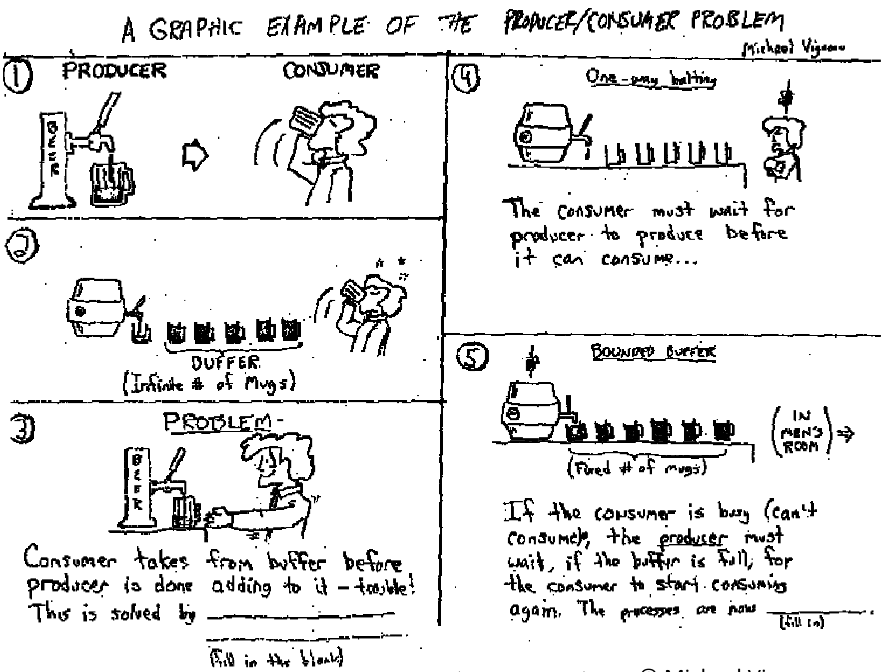
Producers	processes that, from some inner activity, produce data to send to consumers.
Consumers	processes that on receipt of a data element consume data in some internal computation.

- Can join processes synchronously, so data is only sent when producer can send it & consumer receive.
- Better to connect them by a buffer (ie a *queue*)
- For an infinite buffer, the following invariants hold for the buffer:

$$\#elements \geq 0$$

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

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



The Producer-Consumer Problem (cont'd)

```
/* Copyright (C) Wikipedia */
/* Assumes various procedures e.g. P,V */
int in_pointer = 0, out_pointer = 0
semaphore elements = 0; // items produced
semaphore spaces = N;   //spaces left

void producer( int i) {
    while (1) {
        item = produceItem();
        P(spaces);
        putItemIntoBuffer(item);
        in_pointer:=(in_pointer+1) mod N;
        V(elements);
    }
}
```

```
void consumer( int i) {
    while (1) {
        P(elements);
        item = removeItemFromBuffer();
        out_pointer:=(out_pointer+1)mod N
        V(spaces);
        consumeItem(item);
    }
}
int main ( ) {
        cobegin {
producer(1); producer (2); consumer (1);
consumer (2); consumer (3); }
}
```

- Shows the case of a real, bounded circular buffer to count empty places/spaces in the buffer.
- As an exercise prove the following:
  (i) No deadlock, (ii) No starvation &
  (iii) No data removal/appending from an empty/full buffer respectively

2. The Dining Philosophers Problem

- DCU hires 5 philosophers for hard problems
- Philosophers only *think* & *eat*
- Dining table has five plates & five forks*.
- Each plate is endlessly refilled.
- Thinkers aren't dextrous & need 2 forks to eat
- Philosopher may only pick up the forks immediately to his left right.

*or five bowls and five chopsticks

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.

```
semaphore fork [5] := ((5) 1)
/* pseudo-code for attempt one */
/* fork is array of semaphores all initialised to have value 1 */
process philosopher (i := 0 to 4) {
     while (1) {
            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);        // release rh fork
     }
}
```

Dining Philosphers: Solution #1

- Called a *symmetric solution* as each task is identical.
- Symmetric solutions have advantages, e.g. for load-balancing.
- Can prove no 2 philosophers hold same fork as **`Eat()`** is fork's CS.
  - If $\#P_i$ is number of philos with fork *i* then $\boldsymbol{Fork(i) + \#P_i = 1}$

    (ie either philo has the fork or sem is 1)
- Since a semaphore is non-negative then $\boldsymbol{\#P_i \leq 1}$.
- But deadlock possible (i.e none can eat) when all philos pick up their left forks together;
  - i.e. all 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#2: Symmetric Solution

```
/* pseudo-code for room solution to dining philosophers */
/* fork is array of semaphores all initialised to have value 1 */

semaphore Room := 4
semaphore fork (5) := ((5) 1)
process philosopher (i := 0 to 4) {
     while (1) {
            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);
      }
}
```

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

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 three semaphores, **`Room`**, **`fork [i]`** or **`fork [(i+1) mod 5]`**.

  `a) Room` semaphore

  - If semaphore is blocked-queue type then process **`i`** is blocked only if **`Room`** is 0 indefinitely.
  - Needs other 4 philosophers to block on their left forks, as one will finish (if gets 2 forks), put down forks & signal Room (**`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.

3. The Readers-Writers Problem

- Two kinds of processes, readers & writers, share a DB.
- Readers run transactions that examine the DB, writers can examine/update the DB.
- Given initial DB consistency, to ensure that it stays so, writer process must have exclusive access.

- Any number of readers may concurrently access the DB.
- Obviously, for writers, writing is a CS; cannot interleave with any other process.

The Readers-Writers Problem (cont'd)

```
int M:= 20; int N:= 5; int nr:=0;
sem mutexR := 1; sem rw := 1

process reader (i:= 1 to M) {
   while (1)   {
        P (mutexR);
        nr := nr + 1;
        if nr = 1 P (rw); end if
        V (mutexR);
        Read_Database ( );
        P (mutexR);
        nr := nr - 1;
        if nr = 0 V (rw) end if
        V (mutexR);
   }
}
```

```
process writer(i:=1 to N) {
   while (1)
        P (rw);
        Update_Database ( );
        V (rw);
   }
}
```

- Called *readers' preference* solution:
  If a reader accesses DB then reader & writer arrive at their entry protocols then readers always have preference over writers.

Readers-Writers: Ballhausen's Solution

- Readers' Preference isn't fair.
- A continual flow of readers blocks writers from updating the database.
- Ballhausen's solution tackles this:
  - Solution idea: Efficiency: one reader takes up the same space as all readers reading together.
  - A semaphore `access` is used for readers gaining entry to 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.
  - A writer wants to enter DB, occupies 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 available space each.

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

```
sem mutex = 1;
sem access = m;

void reader ( int i ) {
   while (1)
        P(access);
        // ... reading ...
        V(access);
        // other operations
   }
}
```

```
void writer ( int i ) {
   while (1)  {
        P(mutex);
        for k = 1 to m {
                P(access);
        }
        //... writing ...
        for k = 1 to m {
                V(access);
        }
        // other operations
        V(mutex);
   }
}
int main ( ) {
cobegin
  reader (1);reader (2);reader (3);
  writer (1); writer (2);
}
```

Monitors

- Main issue to semaphores: low level coding construct
  - If one coder forgets to do `V()` after CS, the whole system can deadlock.
- Need a higher level construct that groups the responsibility for correctness into a few modules.
- *Monitors* do this. They're an extension of the monolithic monitor used in OS to allocate memory etc.
  - *Encapsulate* procedures & their data into single modules (*monitors*)
  - Ensure only one process execute a monitor procedure at once (=>ME).
  - Of course different processes can execute procedures from different monitors at the same time.

Monitors (cont'd): Condition Variables

- Synchronise using *condition variables*, data structures with 3 commands defined for them:

wait (C)	Process calling the monitor with this command suspends in a FIFO queue associated with *C*. ME on monitor is released.
signal (C)	If the queue associated with *C* is non-empty, wake the process at the head of the queue.
non-empty (C)	Gives true if queue on with *C* is non-empty.

- NB: difference btw `P` in semaphores & `wait(C)` in monitors:
  - latter always delays until `signal(C)` is called,
  - former only if the semaphore variable is zero.

Monitors (cont'd): Signal & Continue

- If a monitor guarantees mutual exclusion:
  - A process uses the *signal* operation
  - So wakes up another process suspended in the monitor,
  - So 2 processes in same monitor at once????
  - Yes.
- To solve: a few signalling constructs: simplest *signal & continue*.
  - With this, process in monitor signalling a condition variable is allowed to run to finish,
  - So the *signal* operation should be at the end of the procedure.
  - Process suspended on condition variable, but now awake, is scheduled for *immediate resumption*,
  - After exit from monitor of process that signalled condition variable.

Readers-Writers Using Monitors in C

```
/* Copyright (C) 2006 M. Ben-Ari */
monitor RW {
  int NR = 0, NW = 0;
  condition OK2Rd, OK2Wr;

  void StartRead() {
    if (NW || !empty(OK2Wr))
       waitc(OK2Rd);
    NR := NR + 1;
    signalc(OK2Rd);  }

  void EndRead() {
    NR := NR - 1;
    if (NR == 0) signalc(OK2Wr); }

  void StartWrite() {
    if (NW || (NR! = 0))
       waitc(OK2Wr);
    NW = 1;
  }

   void EndWrite() {
     NW = 0;
     if (empty(OK2Rd))
        signalc(OK2Wr);
     else signalc(OK2Rd); } }

void Reader(int N) { int i;
  for (i = 1; i < 10; i++) {
    StartRead();
    cout << N << "reading" << '\n';
    EndRead();  } }

void Writer(int N) { int i;
 for (i = 1; i < 10; i++) {
    StartWrite();
    cout << N << "writing" << '\n';
    EndWrite();   } }

void main() {
  cobegin { Reader(1); Reader(2);
Reader(3); Writer(1); Writer (2);}
}
```

File rw_control.c

Emulating Semaphores Using Monitors

- Semaphores/monitors are concurrent programming primitives of equal power: Monitors are just a higher level construct.

```
/* Copyright (C) 2006 M. Ben-Ari. */
monitor monsemaphore {
int semvalue = 1;
condition notbusy;

void monp()   {
        if (semvalue == 0)
                waitc(notbusy);
        else
                semvalue = semvalue - 1;
        }

void monv()   {
        if (empty(notbusy))
                semvalue = semvalue + 1;
        else
                signalc(notbusy);
        }

}
```

```
int n;

void inc(int i)
{
  monp();
  n = n + 1;
  monv();
}

main() {
  cobegin {
  inc(1); inc(2);
  }
  cout << n;
}
```


Emulating Monitors Using Semaphores

- Need to implement *signal and continue* mechanism.
- Do this with
  - a variable **c_count**
  - one semaphore, **s**, to ensure mutual exclusion
  - & another, **c_semaphore**, to act as the condition variable.
- **wait** translates as:

```
c_count := c_count + 1;
V (s);
P (c_semaphore);            // wait always suspends
c_count := c_count - 1;     // 1 less process in monitor
```

- & **signal** as:

```
if ( c_count > 0 )
     V (c_semaphore)         // only signal if waiting processes

else
     V (s)                   // admit another process
```

Dining Philosophers Using Monitors

```
monitor (fork_mon)
/* Assumes: wait( ), signal( )*/
/* and condition variables    */
   int fork:= ((5) 2);
   condition (ok2eat, 5)
/* array of condition variables */

   void (take_fork (i)) {
     if ( fork (i) != 2 )
        waitc (ok2eat(i));

        fork ((i-1) mod 5):=
                fork((i-1) mod 5)-1;
        fork ((i+1) mod 5) :=
                fork((i+1) mod 5)-1;
   }

   void release_fork (i)        {
       fork ((i-1) mod 5):=
                fork((i-1) mod 5)+1;
       fork ((i+1) mod 5) :=
                fork((i+1) mod 5)+1;

    }

        if ( fork((i+1)mod 5) ==2 )
          signalc(ok2eat((i+1)mod 5));
                //rh phil can eat

        if ( fork ((i-1)mod ) == 2 )
          signalc(ok2eat((i-1)mod 5));
                //lh phil can eat

    }
 }

 void philo ( int i )     {
     while (1)   {
         Think ( );
         take_fork (i);
         Eat ( );
         release_fork (i);
     }
 }
 void main( ) {
     cobegin { philo(1); philo(2);
 philo(3); philo(4); philo(5); }
 }
```


Dining Philosophers: Proof of No Deadlock

<u>Theorem</u> Solution Doesn't Deadlock

- *Proof:*
  - Let $\#E =$ number of eating philosophers, $=>$ have taken both forks.
  - Then following invariants are true from the program:

$$Non-empty(\mathtt{ok2eat[i]}) \Rightarrow \mathtt{fork[i]} < 2 \qquad \text{eqn (1)}$$

$$\sum_{\mathtt{i}=1}^{5} fork[\mathtt{i}] = 10 - 2(\#E) \qquad \text{eqn (2)}$$

- Deadlock means $\#E = 0$, all philosophers are queued on `ok2eat` and none can eat:
  - If all enqueued then (1) => $\sum \mathtt{fork[i]} \leq 10$
  - If no philosopher is eating, then (2) => $\sum \mathtt{fork[i]} \leq 5$.
- Contradiction! => solution does not deadlock.
- But individual starvation can occur. How? How to avoid?

Monitors: The Sleeping Barber Problem (cont'd)

- The barber and customers are interacting processes,
- The barber shop is the monitor in which they intereact.

Monitors: The Sleeping Barber Problem

- A small barber shop has two doors, an entrance and an exit.
- Inside, barber spends all his life serving customers, one at a time.

1. When there are none in the shop, he sleeps in his chair.
2. If a customer arrives and finds the barber asleep:
   - he awakens the barber,
   - sits in the customer's chair and sleeps while hair is being cut.
3. If a customer arrives and the barber is busy cutting hair,
   - the customer goes asleep in one of the two waiting chairs.
4. When the barber finishes cutting a customer's hair,
   - he awakens the customer and holds the exit door open for him.
5. If there are waiting customers,
   - he awakens one and waits for the customer to sit in the barber's chair,
   - otherwise he sleeps.

Monitors: The Sleeping Barber Problem (cont'd)

- Use three counters to synchronize the participants:
  - `barber`, `chair` and `open` (all initialised to zero)
- Variables alternate between zero and unity:
  1. `barber==1` the barber is ready to get another customer
  2. `chair==1` customer sitting on chair but no cutting yet
  3. `open==1` exit is open but customer not gone yet,
- The following are the synchronization conditions:
  - Customer waits until barber is available
  - Customer remains in chair until barber opens it
  - Barber waits until customer occupies chair
  - Barber waits until customer leaves

Monitors: Sleeping Barbers (cont'd)

```
monitor (barber_shop)
   int barber:=0; int chair :=0; int open :=0;
   condition (barber_available) ;      // signalled when barber > 0
   condition (chair_occupied) ;        // signalled when chair > 0
   condition (door_open) ;             // signalled when open > 0
   condition (customer_left) ;         // signalled when open = 0
```

```
void (get_haircut()) {
    do
      waitc(barber_available)
    while ( barber==0)

    barber := barber - 1;
    chair := chair + 1;

    signalc (chair_occupied);
    do
      waitc (door_open)
    while (open==0)

    open := open - 1;
    signalc (customer_left);
} // called by customer
```

```
void (get_next_customer( ))  {
     barber := barber +1;
     signalc(barber_available);

     do
       waitc(chair_occupied)
     while ( chair == 0 )

     chair := chair -1;
} // called by barber

void (finished_cut( )) {
     open := open +1;
     signalc (door_open);

     do
       waitc(customer_left)
     while (open==0)
}  // called by barber
}
```

Sleeping Barber Using Monitors (cont'd)

```
void customer ( i ) {
    while (1) {
         get_haircut ( );
         // let it grow
    }
}

void barber ( i ) {
    while (1) {
         get_next_customer ( );
         // cut hair
         finished_cut ( )
    }
}

int main ( ) {
  cobegin {
         barber (1); barber (2);
         customer (1); customer (2);
  }
}
```


Sleeping Barber Using Monitors (cont'd)

- For the Barbershop, the monitor provides an environment for the customers and barber to rendezvous
- There are four synchronisation conditions:
  - Customers must wait for barber to be available to get a haircut
  - Customers have to wait for barber to open door for them
  - Barber needs to wait for customers to arrive
  - Barber needs to wait for customer to leave
- Processes
  - wait on conditions using **`wait()`**s in loops
  - **`signal()`** at points when conditions are true

Summary

- Can define a concurrent program as the interleaving of sets of sequential atomic instructions.
- Ensuring correctness of concurrent programs is tough even for two process systems as need to ensure both *Safety* & *Liveness* properties.
- Semaphores & Monitors facilitate synchronization among processes.
- Monitors are higher level but can emulate either one by other.
- Both have been used to simulate classical synchronization Problems:
  – Producers & Consumers
  – Readers & Writers
  – Dining Philosophers