**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;
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
    - saving thread context ⇒ stopping current execution & saving all data needed to execute later.
  - Process:
    - s/w processor in whose context one or more threads may be executed.
    - executing a thread means executing a series of instructions in it’s context.

Context Switching:

- Processor context:
  - minimal collection of values stored in processor registers to run some instructions, e.g., stack pointer, addressing registers, program counter.
- Thread context:
  - minimal collection of values stored in registers & memory, to run some instructions, i.e., processor context, state.
- Process context:
  - minimal collection of values stored in registers & memory, used to execute 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 lots of external events & threads block on a per-event basis ⇒ if kernel can’t tell them apart, 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 due to 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 that threads do not interfere with each other
  - This brings us on to the topic of *Concurrent Correctness*
A Model of Concurrent Programming

- Can define a concurrent program as the interleaving of sets of sequential atomic instructions.
  - i.e. a set of interacting sequential processes, execute at the same time, on the same or different processors.
  - processes are said to be interleaved, i.e. at any given time each processor is executing one of the instructions of the sequential processes.
  - relative rate at which instructions 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 includes instructions like INC then this program will be correct no matter which instruction is executed first.
- If all arithmetic must be performed in registers then the following interleaving does not produce the desired results.
- This dependency on unforeseen circumstances is known as a Race Condition
- A concurrent program must be correct under all possible interleavings.
Correctness: A More Formal Definition

- If $P(\bar{a})$ is a property of input (pre condition), and $Q(\bar{a}, \bar{b})$ is a property of input & output (post condition), then correctness is defined as:

- Partial correctness:
  
  \[
  P(\bar{a}) \land \text{Terminates}(\text{Prog}(\bar{a}, \bar{b})) \Rightarrow Q(\bar{a}, \bar{b})
  \]

- Total correctness:
  
  \[
  P(\bar{a}) \Rightarrow [\text{Terminates}(\text{Prog}(\bar{a}, \bar{b})) \land Q(\bar{a}, \bar{b})]
  \]

- Totally correct programs terminate. A totally correct specification of the incrementing tasks is:
  
  \[
  \forall a \in \mathbb{N} \Rightarrow [\text{Terminates}(\text{INC}(a, a)) \land a=a+1]
  \]

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 ways to specify fairness.

– Weak Fairness
  If a process continuously makes a request, eventually it will be granted.

– Strong Fairness
  If a process makes a request infinitely often, eventually it will be granted.

– Linear waiting
  If a process makes a request, it will be granted before any other process is granted the request more than once.

– FIFO
  If a process makes a request, it will be granted before any other process makes a later request.

Mutual Exclusion

• As seen, a concurrent program must be correct in all allowable interleavings.
• So there must be some sections of the different processes which cannot be allowed to be interleaved.
• These are called critical sections.
• We will attempt to solve the mutual exclusion problem using software first before more sophisticated 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

```c
/* Copyright © 2006 M. Ben-Ari. */
void q() {
    while (1) {
        cout << "q non-critical section \n";
        while ((!(turn == 2)));
        cout << "q critical section \n";
        turn = 1;
    }
}

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

```c
/* Copyright © 2006 M. Ben-Ari. */
void q() {
    while (1) {
        cout << "q non-critical section \n";
        while (!(wantp == 0));
        wantq = 1;
        cout << "q critical section \n";
        wantq = 0;
    }
}

void p() {
    while (1) {
        cout << "p non-critical section\n";
        while (!(wantq == 0));
        wantp = 1;
        cout << "p critical section\n";
        wantp = 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 checks wantq and finds wantq== 0.
  - q sets wantq= 1.
  - q enters its critical section.
  QED
Software Solutions to Mutual Exclusion Problem # 3

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:

\[
\begin{align*}
\text{wantp} &= 1 \equiv \text{at}(c_1) \lor \text{at}(d_1) \lor \text{at}(e_1) \\
\text{wantq} &= 1 \equiv \text{at}(c_2) \lor \text{at}(d_2) \lor \text{at}(e_2) \\
\neg(\text{at}(d_1) \land \text{at}(d_2)) &= \text{Eqn}(1) \\
\text{Eqn}(2) \equiv &
\end{align*}
\]

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

Software Solutions # 3 (cont’d)

• Eqn (1) is initially true:
  - Only the $b_1 \to c_1$ and $e_1 \to a_1$ transitions can affect its truth.
  - But each of these transitions also changes the value of $\text{wantp}$.

• A similar proof is true for Eqn (2).

• Eqn 3 is initially true, and
  - can only be negated by a $c_2 \to d_2$ transition while $\text{at}(d_1)$ is true.
  - But by Eqn (1), $\text{at}(d_1) \Rightarrow \text{wantp}=1$, so $c_2 \to d_2$ cannot occur since this requires $\text{wantp}=0$. 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 $\text{wantp}$ | q assigns 1 to $\text{wantq}$ |
  | p tests $\text{wantq}$ & remains in its do loop | q tests $\text{wantp}$ & 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;
    }
    cout << "p critical section\n";
    wantp = 0;
  }
}

void q()
{
  while (1) {
    cout << "q non-critical section\n";
    wantq = 1;
    while (wantp == 1) {
      wantq = 0;
    }
    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**.
     This is a form of deadlock. In **deadlock** there is no possible interleaving which allows processes to enter their CS. In **livelock**, some interleavings succeed, but there are sequences which do not succeed.
Proof of Failure of Attempt 4:
1. By Starvation
   - $p$ sets $want_p$ to 1.
   - $p$ completes a full cycle:
     - Checks $want_q$ Enters CS
     - Resets $want_p$ Does non-CS
     - Sets $want_p$ to 1
   - $q$ sets $want_q$ to 1
   - $q$ checks $want_p$, sees $want_p=1$ & resets $want_q$ to 0
   - $q$ sets $want_q$ to 1 and back
2. By Livelock
   - $p$ sets $want_p$ to 1.
   - $p$ tests $want_q$, remains in its do loop
   - $p$ resets $want_p$ to 0 to relinquish attempt to enter CS
   - $p$ sets $want_p$ to 1
   - $q$ sets $want_q$ to 1
   - $q$ tests $want_p$, remains in its do loop
   - $q$ resets $want_q$ to 0 to relinquish attempt to enter CS
   - $q$ sets $want_q$ to 1 and back
   - 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);
            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);
            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.
• For $N$ process mutual exclusion problem, there are many algorithms; all 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 some 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.
Mutual Exclusion for $N$ Processes:
The Bakery Algorithm (cont’d)

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

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])))
                    if (j<0) j = NODES; j++;
                for (j=0; j<NODES; j++)
                    number[j]=0;
                for (j=0; j<NODES; j++)
                    choosing[j]=0;
                cout << "process " << i << " CS\n";
                num[i]=0;
                p(0); p(1); p(2); // 3 processes here
            }
    }
}

main() {
    int j;
    for (j=0; j< NODES; j++)
        number[j]=0;
    for (j=0; j< NODES; j++)
        choosing[j]=0;
    cout << "processes here\n";
    cobegin {
        p(0); p(1); p(2); // 3 processes here
    }
}

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:
  \[
  S \geq 0 \quad (1) \\
  S = S_0 + \#V - \#P \quad (2)
  \]
  where \( S_0 \) is initial value of Semaphore \( S \)

Semaphores for Mutual Exclusion

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

```c
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:**
- Requires all processes to be suspended on their \( P(\text{mutex}) \) operations.
- Then \( mutex = 0 \) 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: Awakens any one of the suspended processes.
  - Blocked-queue semaphore: Suspended processes are kept in FIFO & are awakened in order of suspension.
  - Busy-wait semaphore: Semaphore value 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. 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≥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.

---

Ye Classicale Problemes of Synchronization

1. Ye Probleme of Ye Producers & Consumers
2. Ye Probleme of Ye Readers & Writers
3. Ye Probleme of Ye Dining Philosophers
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, the following invariants hold for the buffer:
  
  $$\#\text{elements} \geq 0$$

  $$\#\text{elements} = 0 + \text{in\_pointer} - \text{out\_pointer}$$

- 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)

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

- This demonstrates the case of a real, bounded circular buffer used 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

---

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 dining room 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.

```plaintext
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 (); // grab fork[i]
    P(fork (i)); // grab left fork
    P(fork ((i+1) mod 5)); // grab right fork
    Eat ();
    V(fork (i)); // release fork[i]
    V(fork ((i+1) mod 5)); // release right fork
  }
}
```
Dining Philosophers: 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 2 philosophers hold same fork as `Eat()` is fork's CS.
  - If \( \#P_i \) is number of philosophers holding fork \( i \) then \( Fork(i) + \#P_i = 1 \)
    (ie either philosopher is holding the fork or sem is 1)
- Since a semaphore is non-negative then \( \#P_i \leq 1 \).
- 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 #2: Symmetric Solution

```c
/* 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.

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 ≤ i ≤ 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/update the DB.
- Given that the DB is initially consistent, then to ensure that it remains so, 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)

```c
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 as if some reader accesses DB and reader + writer arrive at their entry protocols then readers always have preference over writers.
Readers-Writers: Ballhausen’s Solution

- 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.
- Ballhausen’s solution aims to tackle this:
  - The idea behind the 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 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.

```c
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);
            }
        }
        // other operations
    }
}

int main ( ) {
    cobegin
        reader (1); reader (2); reader (3);
        writer (1); writer (2);
    }
```
Monitors

- Main issue to semaphores is they’re low level programming construct
  - If one coder forgets to do \texttt{V()} operation on a semaphore after a CS, then the whole system can deadlock.
- What is required is a higher level construct that groups the responsibility for correctness into a few modules.

- \textit{Monitors} are such a construct. These are an extension of the monolithic monitor found in OS for allocating memory etc.
  - They\textit{ encapsulate} a set of procedures, and the data they operate on, into single modules (\textit{monitors})
  - They guarantee that only one process can execute a procedure in the monitor at any given time (mutual exclusion).
  - Of course different processes can execute procedures from different monitors at the same time.

Monitors (cont’d): Condition Variables

- Synchronisation is achieved by using \textit{condition variables}, data structures that have 3 operations defined for them:
  - \texttt{wait(C)} The process that called the monitor containing this operation is suspended in a FIFO queue associated with C. Mutual exclusion on the monitor is released.
  - \texttt{signal(C)} If the queue associated with C is non-empty, wake the process at the head of the queue.
  - \texttt{non-empty(C)} Returns true if the queue associated with C is non-empty.

- Note the difference between the \texttt{P} in semaphores and \texttt{wait(C)} in monitors: latter always delays until \texttt{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
  – Thus awakens another process suspended in the monitor,
  – So aren’t there 2 processes in same monitor at same time?
    – Yes.

• To solve this, several signalling mechanisms can be implemented, the simplest is signal & continue mechanism.
  – Under these rules the process in the monitor that signals a condition variable is allowed to continue to completion,
  – 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

```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(OK2Rd);  }

  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.

```c
/* 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 \texttt{c\_count}
  – one semaphore, \texttt{s}, to ensure mutual exclusion
  – \& another, \texttt{c\_semaphore}, to act as the condition variable.

* wait translates as:

  ```c
  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:

  ```c
  if (c\_count > 0)
      V(c\_semaphore) // only signal if waiting processes
  else
      V(s) // admit another process
  ```
Dining Philosophers Using Monitors

```c
monitor (fork_mon)
/* Assumes: wait( ), signal( )* /
/* and condition variables */
int fork := ((5) 2);
condition (ok2eat, 5)
/* array of condition variables */
if ( fork((i+1)mod 5) == 2 )
signal(0k2eat((i+1)mod 5));
// rh phil can eat
if ( fork ((i-1) mod 5) == 2 )
signal(0k2eat((i-1)mod 5));
// lh phil can eat
void (take_fork (i)) {
  if ( fork(i) != 2 )
    waitc (ok2eat(i));
  fork((i-1) mod 5) :=
    fork((i+1) mod 5) :=
    fork((i-1) mod 5)-1;
    fork((i+1) mod 5)-1;
    fork((i-1) mod 5)+1;
    fork((i+1) mod 5)+1;
    void philo (int i) {
      fork((i+1) mod 5) :=
        fork((i-1) mod 5)+1;
        fork((i+1) mod 5)+1;
    }
    fork((i-1) mod 5) :=
    fork((i+1) mod 5) :=
    fork((i-1) mod 5)+1;
    fork((i+1) mod 5)+1;
    void main() {
      cobegin {
        philo(1); philo(2); philo(3); philo(4); philo(5); }
    }
}
```

Lecture 2: Concurrent Correctness Support

Dining Philosophers: Proof of No Deadlock

**Theorem**

Solution Doesn’t Deadlock

**Proof:**

- Let \( #E \) = number of eating philosophers, \( \Rightarrow \) have taken both forks.
- Then following invariants are true from the program:

\[
\nonempty(\text{ok2eat}[i]) \Rightarrow \text{fork}[i] < 2 \quad \text{eqn (1)}
\]

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

- Deadlock implies \( #E = 0 \) and all philosophers are enqueued on \( \text{ok2eat} \) and none are eating:
  - If they are all enqueued then (1) implies \( \sum \text{fork}[i] \leq 10 \)
  - If no philosopher is eating, then (2) implies \( \sum \text{fork}[i] \leq 5. \)
- Contradiction implies that the 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 interact.

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

```c
void (get_haircut()) { 
  do 
  while (barber==0) 
    barber := barber - 1; 
  chair := chair + 1; 
  signalc (chair_occupied); 
  do 
  while (open==0) 
    open := open - 1; 
  signalc (customer_left); 
} // called by customer
```

```c
void (get_next_customer( )) { 
  barber := barber +1; 
  signalc(barber_available); 
  do 
  while (chair==0 ) 
    chair := chair -1; 
} // called by barber
```

```c
void (finished_cut( )) { 
  open := open +1; 
  signalc (door_open); 
  do 
  while (open==0) 
    open := open +1; 
} // 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);
    }
}

• 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