Concurrency and Synchronization

CS 447
Monday 3:30-5:00
Tuesday 2:00-3:30

Prof. R K Joshi
CSE, IIT Bombay

Interleaving in a multiprogamming environment

- A scenario
- Observe the interleaving of execution

Consider the following processes

```
P0: v=read(counter);
    v=v+1;
    write(counter,v);

P1: v=read(counter);
    v=v+1;
    write(counter,v);
    ...
Pn: v=read(counter);
    v=v+1;
    write(counter,v);
```

counter is shared among all processes.

Each process updates a counter
at entry and eventually exits

Counter must indicate how many
Of these processes executed in the
system

An execution trace

Time slice	Statement executed	P0::v	P1::v	counter
0	P0::read	0	--	0
1	P1::read	0	0	0
2	P1::=	0	1	0
3	P1::write	0	1	1
4	P0::=	1	1	1
5	P0::write	1	1	1

Observation

- Value of counter is incorrect
  - Concurrent execution through interleaving
  - No control over access
  - Mutual exclusion requirement

Critical Section

- CS = Code that accesses a shared resource
  - This section of code is a critical section
  - Critical section needs to be protected from violation of mutual exclusion
  - i.e. CS needs to execute mutually exclusively over other CSs
  - Critical sections operate on one or more of the same shared resource

A Critical Section Protocol

P0	P1

while (true) {
  protocol *enter CS*
 	
    Critical section
    code
 	
  protocol *exit CS*
}
	while (true) {
	  protocol *enter CS*
	
	    Critical section
	    code
	
	  protocol *exit CS*
	}

What constitutes the enter and exit protocols?

- Interrupt based CS
- Signaling and messaging
- Shared variables + atomic R/W operations
- Semaphores
- Monitors
- Spin locks

What properties must your critical section protocols guarantee

- Correctness of mutual exclusion
- Progressiveness
- Freedom from deadlocks
- Freedom from livelocks
- Freedom from starvation

Shared variables and atomic R/W based solutions

- Processes use shared variables
- They may use local varibles
- They perform local computations
- They decide locally based on the shared state (variables) on whether their entry code is successful
- After the CS, they execute CS exit code

Taking Turns

Shared variable Turn=0

P0	P1

while (true) {
  while (Turn!=0);
 	
    Critical section
      code
 	
  Turn=0;
}
	while (true) {
	  while (Turn!=1)
	
	    Critical section
	      code
	
	  Turn=1;
	}

Taking Turns

Shared variable Turn=0

P0	P1

while (true) {
while (Turn!=0);
 	
Critical section
code
 	
~~Turn=0;~~ Turn=1;
}
	while (true) {
	while (Turn!=1)
	
	Critical section
	code
	
	~~Turn=1;~~ Turn=0;
	}

Observations and Problems faced

- Mutual exclusion requirement is guaranteed
- The solution violates the *progressiveness* property
- If a process is not interested in CS, there is no progress
- Progressiveness – Uninterested processes must not hold interested process from entering CS

An improvisation

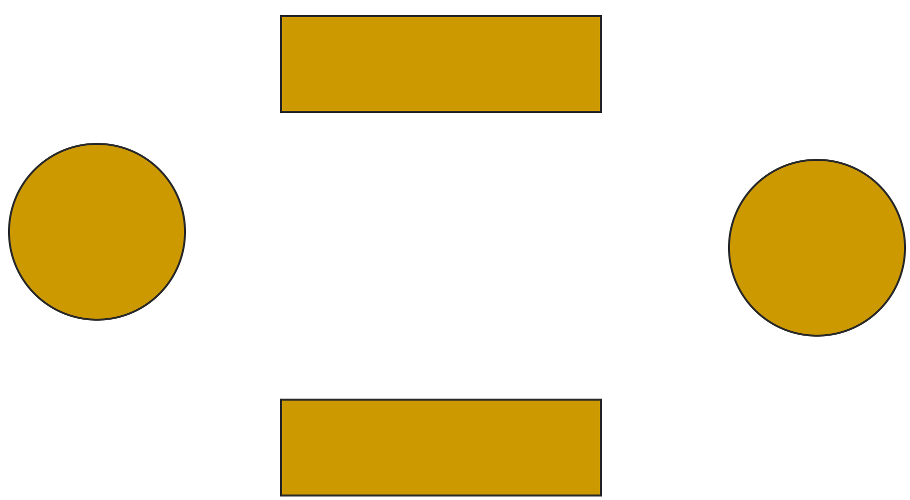
Shared Willing[0]=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  while (Willing[1]);
 	
    Critical section
      code
 	
  Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  while (Willing[0]);
	
	    Critical section
	      code
	
	  Willing[1]=0;
	}

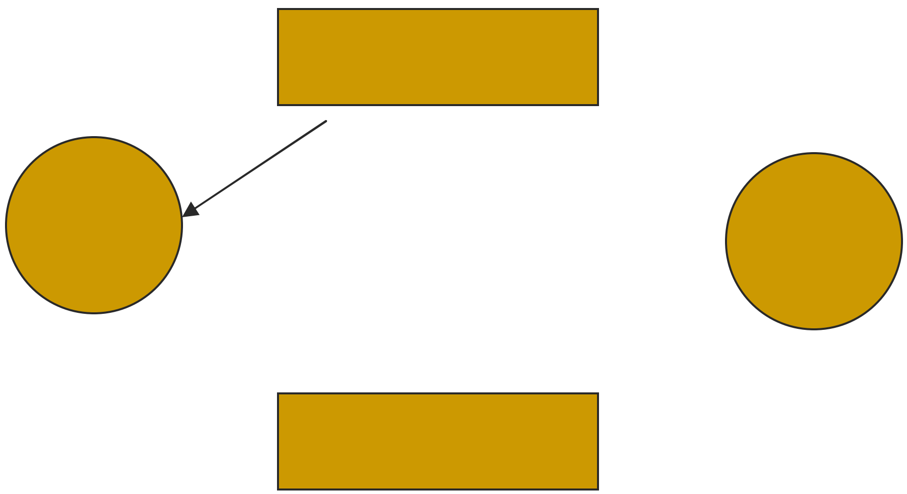
Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



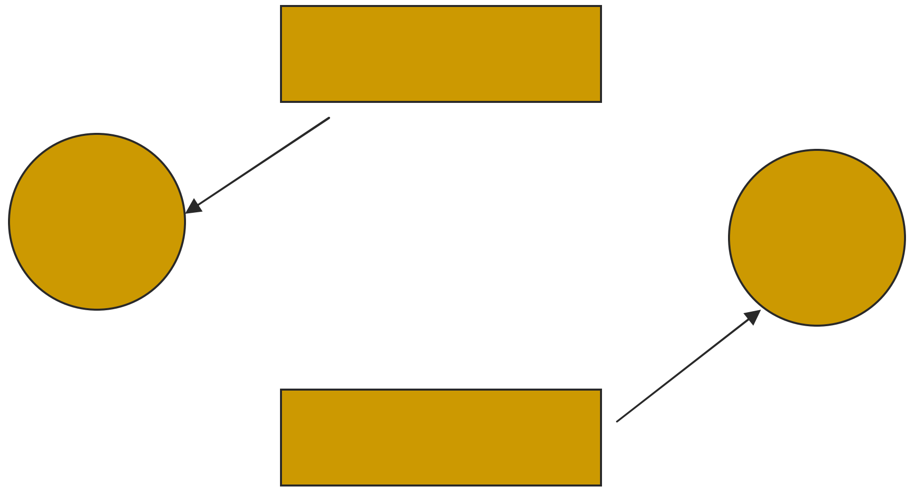
Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



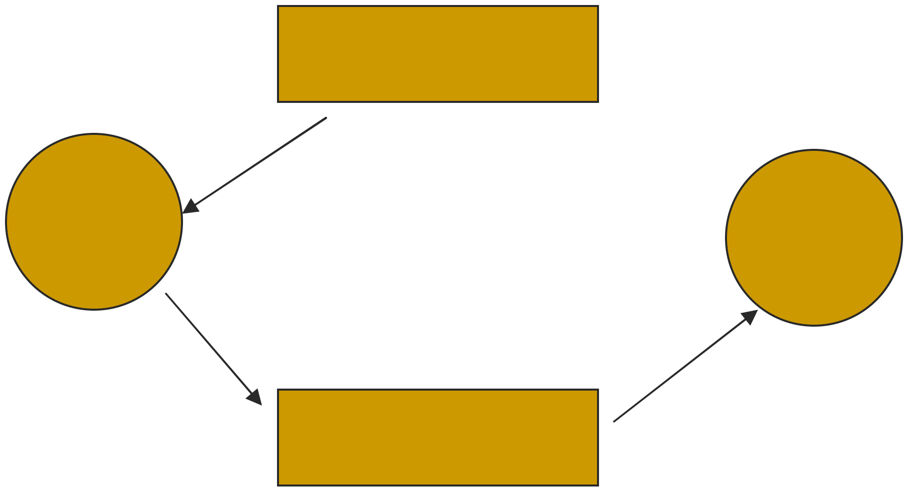
Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



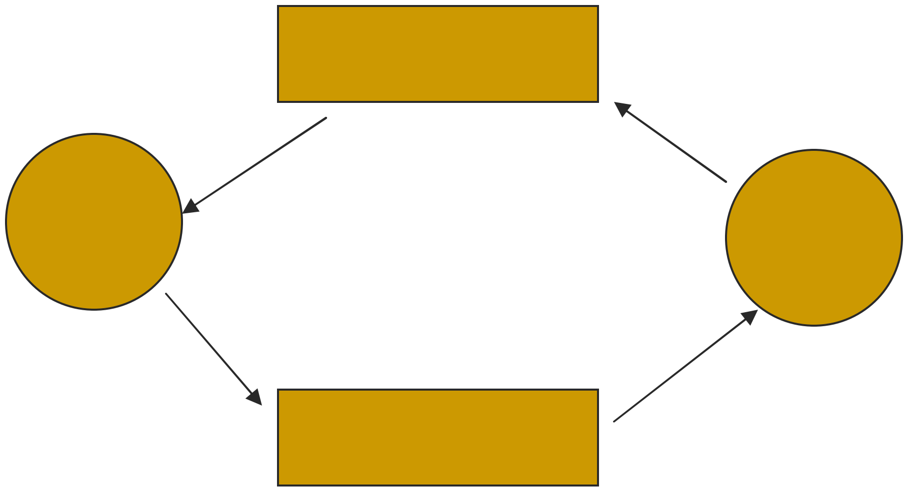
Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



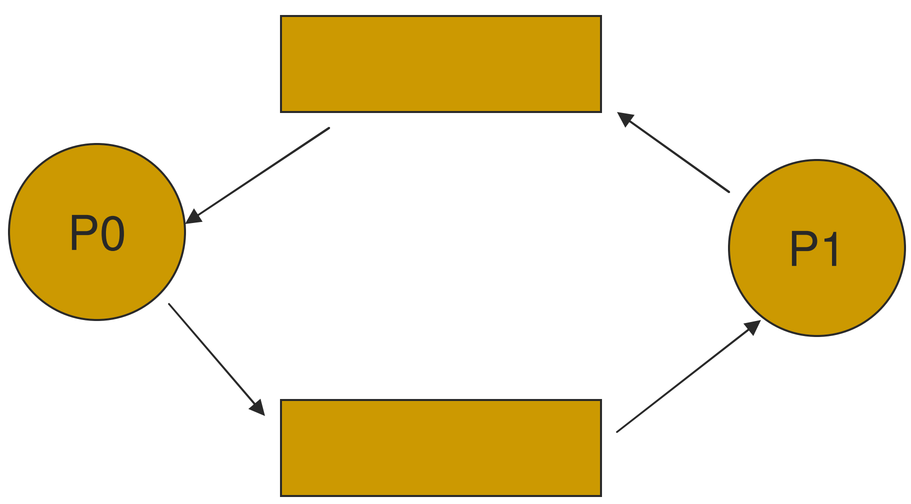
Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



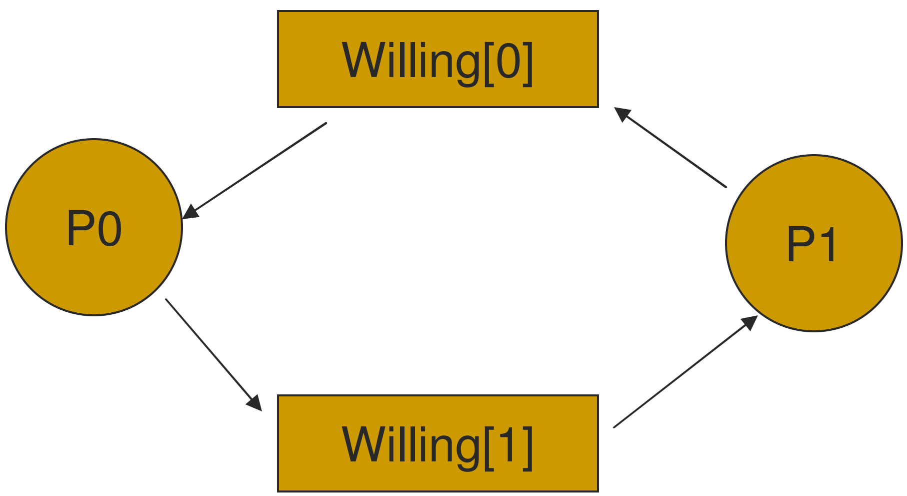
Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



Observations and Problems faced

- Correctness of ME guaranteed
- It is progressive
- But possibility of a deadlock



An Attempt

(erroneous: firstly, the entry code does not allow entry to CS)

Shared Willing[0]=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  do
    Willing[0]=1;
    if !(willing[0] AND willing[1])
        while (Willing[1]);
  while (!willing[0]);
    Critical section
        code
 	
  Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  do if !(willing[0] AND willing[1])
	        while (Willing[0]);
	 while (!willing[1);
	
	    Critical section
	        code
	
	  Willing[1]=0;
	}

Towards removing deadlock possibility

Shared Willing[0]=0;
Shared Willing[1]=0;

<table>
<tr><th>P0</th><th>P1</th></tr>
<tr><td><pre>
while (true) {
  Willing[0]=1;
  while (Willing[1])
    Do Something!;

    Critical section
        code

  Willing[0]=0;
}
</pre></td><td><pre>
while (true) {
  Willing[1]=1;
  while (Willing[0])
   Do Something!;

    Critical section
        code

   Willing[1]=0;
}
</pre></td></tr>
</table>

Improvisation

Shared Willing[0]=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  while (willing [1]) {
    Willing[0]=0;
    sleep for some time
    Willing [0]=1;
  }
        Critical section code
 	
  Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  while (willing [0]) {
	    Willing[1]=0;
	    sleep for some time
	    Willing [1]=1;
	  }
	        Critical section code
	
	  Willing[1]=0;
	}

Observations and problems faced

- Solves the deadlock problem
- But a livelock may occur
- Processes may get locked forever in a cycle of release-wait-hold

How to remove the livelock possibility?

- Why make both processes retract?
- Let only one process retract if it senses that livelock is possible

Solution that is deadlockfree, progressive, but livelock-prone

Shared Willing[0]=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  while (willing [1]) {
    Willing[0]=0;
    sleep for some time
    Willing [0]=1;
  }
Critical section code
 	
  Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  while (willing [0]) {
	    Willing[1]=0;
	    sleep for some time
	    Willing [1]=0;
	  }
	Critical section code
	
	  Willing[1]=0;
	}

Possibility of Livelock is removed

Shared Willing[0]=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  while (willing [1]) {
    Willing[0]=0;
    sleep for some time
    Willing [0]=1;
  }
        Critical section code
 	
  Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  while (willing [0]);
	
	        Critical section code
	
	  Willing[1]=0;
	}

Possibility of Livelock is removed

Shared Willing[0]=0;
Shared Willing[1]=0;

<table>
<tr><th>P0</th><th>P1</th></tr>
<tr><td><pre>
while (true) {
  Willing[0]=1;
  while (willing [1]) {
    Willing[0]=0;
    sleep for some time
    if (!Willing[1]) Willing [0]=1;
  }
        Critical section code

   Willing[0]=0;
}
</pre></td><td><pre>
while (true) {
  Willing[1]=1;
  while (willing [0]);

        Critical section code

   Willing[1]=0;
}
</pre></td></tr>
</table>

It’s unfair to P0!

- Why should it be P0 to withdraw all thetime!

  And

- P0 may have to withdraw its willingness forever in a specific interleaving sequence → Starvation

Dekker's Algorithm (1965)

- Starvation free
- Livelock free
- Deadlock free
- Progressive

Attempt to remove starvation

Shared Willing[0]=0;   Shared turn=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  if (turn=0)
    while (willing [1]) {
     Willing[0]=0;
     while (Willing[1]);
     Willing[0]=1;
  }
Else while (Willing[1]);
        Critical section code
 	
   Willing[0]=0;
turn=1;
}
	while (true) {
	  Willing[1]=1;
	  if (turn=1)
	    while (willing [0]) {
	     Willing[1]=0;
	     while (Willing[0]);
	     Willing[1]=1;
	  }
	Else while (Willing[0]);
	        Critical section code
	
	   Willing[1]=0;
	turn=0;
	}

Attempt to remove starvation - II

Shared Willing[0]=0;   Shared turn=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  if (turn=0)
    while (willing [1]) {
     Willing[0]=0;
     while (Willing[1]);
     Willing[0]=1;
  }
Else while (Willing[1]);
            Critical section code
 	
   Willing[0]=0;
turn=0;
}
	while (true) {
	  Willing[1]=1;
	  if (turn=1)
	    while (willing [0]) {
	     Willing[1]=0;
	     while (Willing[0]);
	     Willing[1]=1;
	  }
	Else while (Willing[0]);
	            Critical section code
	
	   Willing[1]=0;
	turn=1;
	}

Attempt to remove starvation - III

Shared Willing[0]=0; Shared turn=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
while (willing [1])
    if (turn=0) {
    Willing[0]=0;
    while (Willing[1]);
    Willing[0]=1;
   }
   else while (Willing[1]);
   Critical section code
 	
  Willing[0]=0;
turn=1;
}
	while (true) {
	  Willing[1]=1;
	while (willing [0])
	    if (turn=1) {
	    Willing[1]=0;
	    while (Willing[0]);
	    Willing[1]=1;
	   }
	   else while (Willing[0]);
	      Critical section code
	
	  Willing[1]=0;
	turn=0;
	}

Possibility of Livelock is removed

Shared Willing[0]=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  while (willing [1]) {
    Willing[0]=0;
    while (Willing[1]);
    Willing[0]=1;
  }
Critical section code
 	
  Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  while (willing [0]);
	
	Critical section code
	
	  Willing[1]=0;
	}

Dekker's Algorithm

Shared Willing[0]=0;   Shared arbitrator=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  while (willing [1]) {
    if (arbitrator=1) Willing[0]=0;
    while (arbitrator=1);
    Willing [0]=1;
  }
      Critical section code
 	
   Willing[0]=0; arbitrator=1;
}
	while (true) {
	  Willing[1]=1;
	  while (willing [0]) {
	    if (arbitrator=0) Willing[1]=1;
	    while (arbitrator=0);
	    Willing [1]=1;
	  }
	      Critical section code
	
	   Willing[1]=0; arbitrator=0;
	}

Hammer in place of a screw driver?

- Can we design something simpler?
- After all we need freedom from
  - Non-progressive blockages!
  - Deadlocks!
  - Livelocks!
  - Starvation!

Peterson’s Algorithm (around 1986)

Shared Willing[0]=0; Shared arbitrator=0;
Shared Willing[1]=0;

P0	P1

while (true) {
  Willing[0]=1;
  arbitrator=1;
  while (willing [1]) && (arbitrator=1);
 	
    Critical section code
 	
   Willing[0]=0;
}
	while (true) {
	  Willing[1]=1;
	  arbitrator=0;
	  while (willing [0]) AND (arbitrator=0);
	
	    Critical section code
	
	   Willing[1]=0;
	}

N process solution

- Lamport’s Bakery Algorithm
  - A process picks up a token number
  - They all go with their critical sections according to the order defined by the token numbers
  - …..

Towards Lamport's Bakery Algorithm

Shared what's shared?

Pi
Pickup a sequence number for itself; Wait for some condition; CS Reset to old state

Towards Lamport's Bakery Algorithm

Shared current;

Pi
Myseqno = current + 1 Wait for some condition; CS Reset to old state

Incorrect!

Towards Lamport's Bakery Algorithm

Shared current=0;
Shared willing [0..N-1];

Pi
Willing [i] = 1 while (current!=i); CS Willing[i]=0 Current= min (I, willing[i]=1 over i=0..N-1)

non progressive!

Towards Lamport's Bakery Algorithm

Shared seqno[0..N-1] = MAX
Shared current = 0

Pi

```
seqno [i] = current
Current =  current + 1
For j=0; j<I; j++
   While (seqno [j] < = seqno [i]) ;
For (j=i+1; j<N; j++)
    while (seqno[j] < seqno [i]);
CS

Seqno [i] = MAX
```

Incorrect!

Towards Lamport's Bakery Algorithm

Shared seqno [0..N-1] =0
Shared scanning [0..N-1]=0

Pi
Myseqno = max (seqno [0..N-1]) + 1 For (j=0; j<N; j++) while (seqno [j]!=0) AND ((seqno [j] < seqno[i]) OR ((j<i) AND (seqno[i]=seqno[j])) ; CS Reset to old state

Lamport's Bakery Algorithm

Shared sequenceNo[0..N-1]=0;
Shared choosing [0..N-1] =0

Pi

choosing[i]=true;
sequenceNo [i] = max (sequenceNo[0]...sequenceNo[N-1])+1;
choosing[i]=false;
For j=0 to n-1
    while (choosing[j]);
    while (sequenceNo[j] !=0) AND ((sequenceNo [i] > sequenceNo[j]) OR (sequenceNo[i]=sequenceNo[j] AND i>j));
CS
sequenceNo[i]=0;

What are the drawbacks of the algorithmic solutions?

- i.e. solutions with shared variables and atomic read and write?
  - Scalability: No of processes is to be known statically
  - Busy wait
  - Responsibility of implementation is with user

- Pointers to OS-supported solution?