Concurrent Threads (1)

Recommended Reading:

Bacon, J.:	Concurrent Systems (9, 10, 11)
Silberschatz, A.:	Operating System Concepts (4, 6, 7)
Stallings, W.:	Operating Systems (5, 6)
Tanenbaum, A.:	Modern Operating Systems (2)
Wettstein, H.:	Systemarchitektur (7, 8, 9, 10)

Roadmap for the next lectures

- Problems with Concurrency
- Interrelationship Patterns
- Well known Concurrency Problems
- Signaling and Synchronization
- Mutual Exclusion
- Communication
- Deadlocks
- Transactions

Potential problems with Concurrency

Concurrent threads often need to share resources and/or data (maintained either in shared memory or files)

- If there is no controlled access to shared data, threads may get inconsistent data
- The overall result of an application performed by concurrent threads may depend on the execution sequence of these threads, i.e. a race condition
- Concurrent threads may even induce conflicts due o competition around exclusive resources

Simple Control Problem

Assumption: Given a shared resource (suppose a display)

```
{Thread 1}
boolean c1:=true
while c1 do


 for i=1 step 1 until MAXINT
    display("i ")

 {do something else}
 begin … c1:=false … end
od
```

Expected Behavior: Until c1= false thread 1 displays sequences of **1 2 3 4 … MAXINT** onto the screen

Simple Control Problem

What may happen now?

Assumption: Given a shared resource (suppose a display)

```
{Thread 1}
boolean c1:=true
while c1 do


 for i=1 step 1 until MAXINT
    display("i ")

 {do something else}
 begin … c1:=false … end
od
```

```
{Thread 2}
boolean c2:=true
while c2 do


 for i=1 step 1 until MAXINT
   display("-i ")

 {do something else}
 begin … c2:=false … end
od
```

Output Mix:

1 2 3 4 5 6 7 8 -1 -2 -3 -4 -5 -6 -7 -8 9 10 ...

due to various dispatching activities (e.g. end of time slice)

Problem: Serialization of Critical Sections

Assumption: Given a shared resource (suppose a display)

```
{Thread 1}
boolean c1:=true
while c1 do

 for i=1 step 1 until MAXINT
    display("i ")

 {do something else}
 begin … c1:=false … end
od
```

```
{Thread 2}
boolean c2:=true
while c2 do

 for i=1 step 1 until MAXINT
   display("-i ")

 {do something}
 begin … c2:=false … end
od
```

Red Boxes = Critical Program Sections because both threads access the shared display

Consequence: We have to offer something enabling serialization of critical sections

Another Concurrency Problem

```
integer a,b :=1;  // shared data for both threads
```

```
{Thread 1}

while true do

  a = a + 1;
  b = b + 1;

 {do something else}
od
```

```
{Thread 2}

while true do

  b = b + 2;
  a = a + 2;

 {do something else}
od
```

Both Threads read (and write to) the shared global data a, b
=> data inconsistency, a !=b after some time due to dispatching!

Conclusion: Resource and data sharing may lead to similar problems!

Interrelationship Patterns with Concurrent Threads

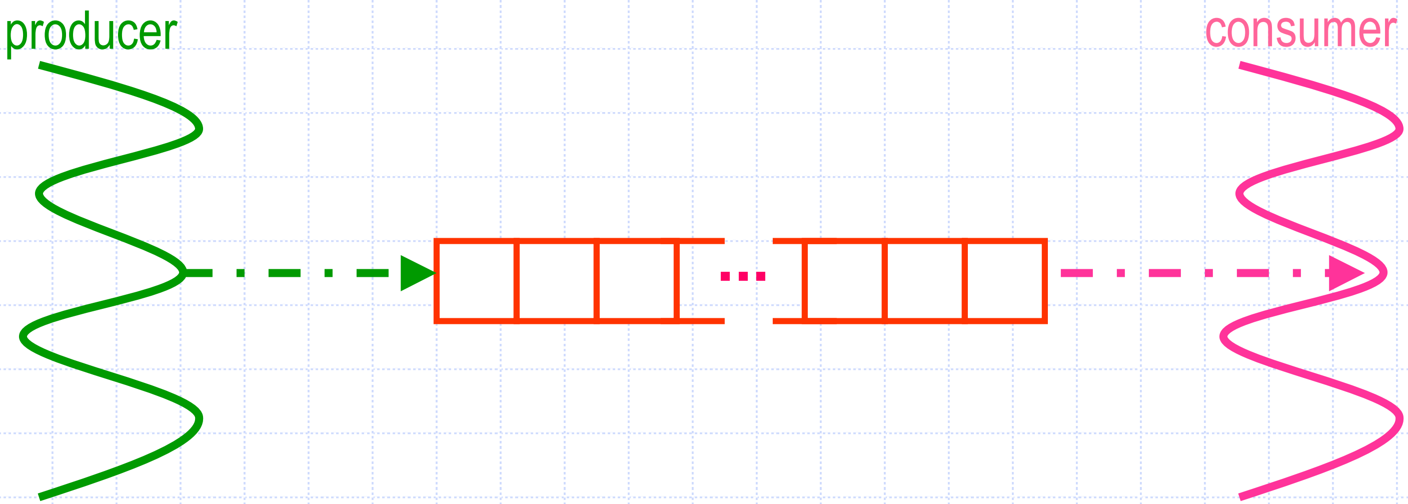
Degree of Awareness	Relationship	Mutual Influence	Control Problems
Unaware of each other	Competition	• Results are independent • Timing may be affected	• Starvation
Indirectly aware of each other (shared object)	Cooperation by Sharing	• Results may be dependent • Timing may be affected	• Mutual Exclusion • Deadlock • Starvation • Data Coherence
Directly aware of each other (communication)	Cooperation by communication	• Results may be dependent • Timing may be affected	• Deadlock • Starvation

Some Classical Concurrency Problems

- Producer/Consumer Problem
- Barbershop Problem
- Reader/Writer Problem
- Cigarette Smoker Problem
- Monkey Rock Problem
- Dining Philosophers

Remark: Discuss them **all** very carefully and find out different solutions, i.e. solutions based upon different synchronization mechanisms (see later)!

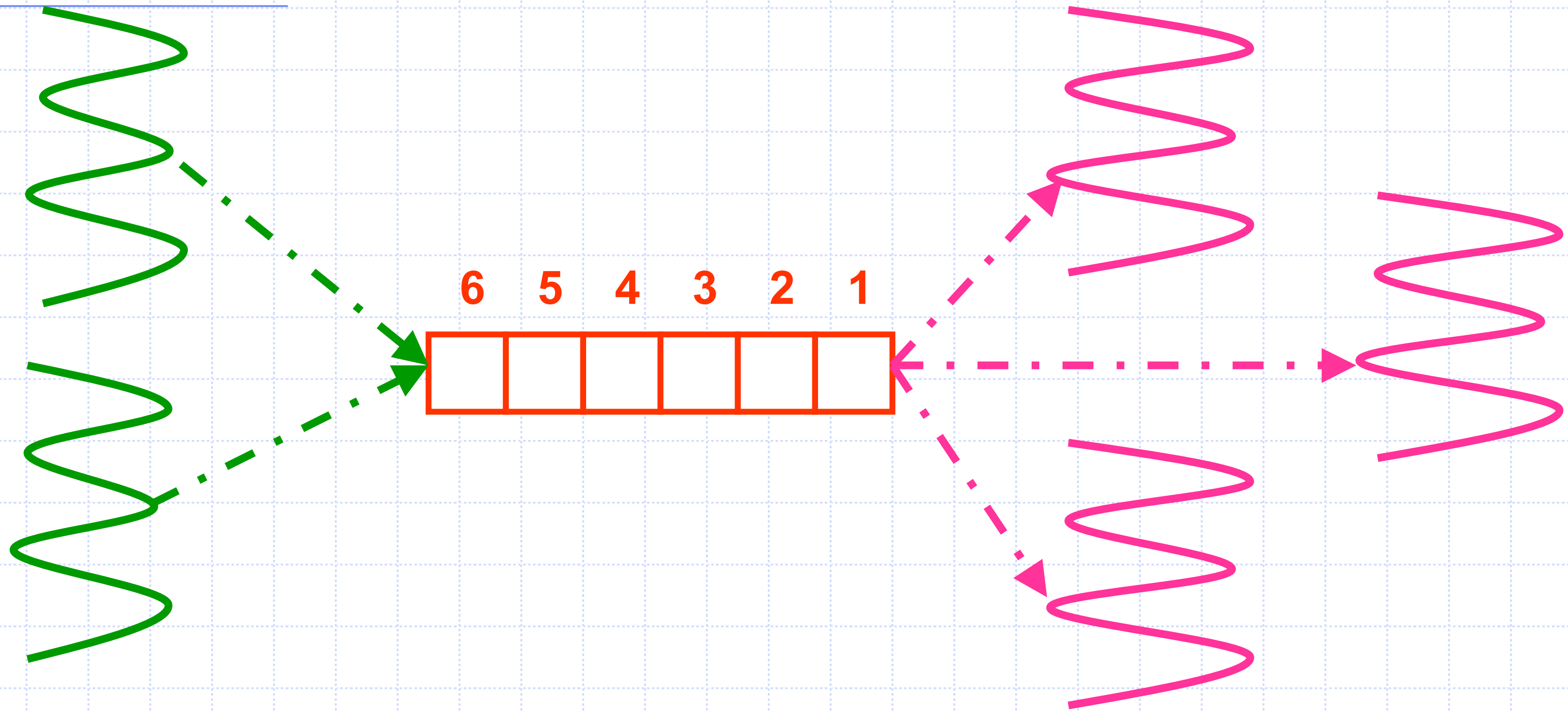
Producer/Consumer Problem with unbounded buffers



Do we really have a concurrency problem with an unbounded buffer, if there are only 1 producer and 1 consumer?

Of course, a consumer cannot consume from an empty buffer.

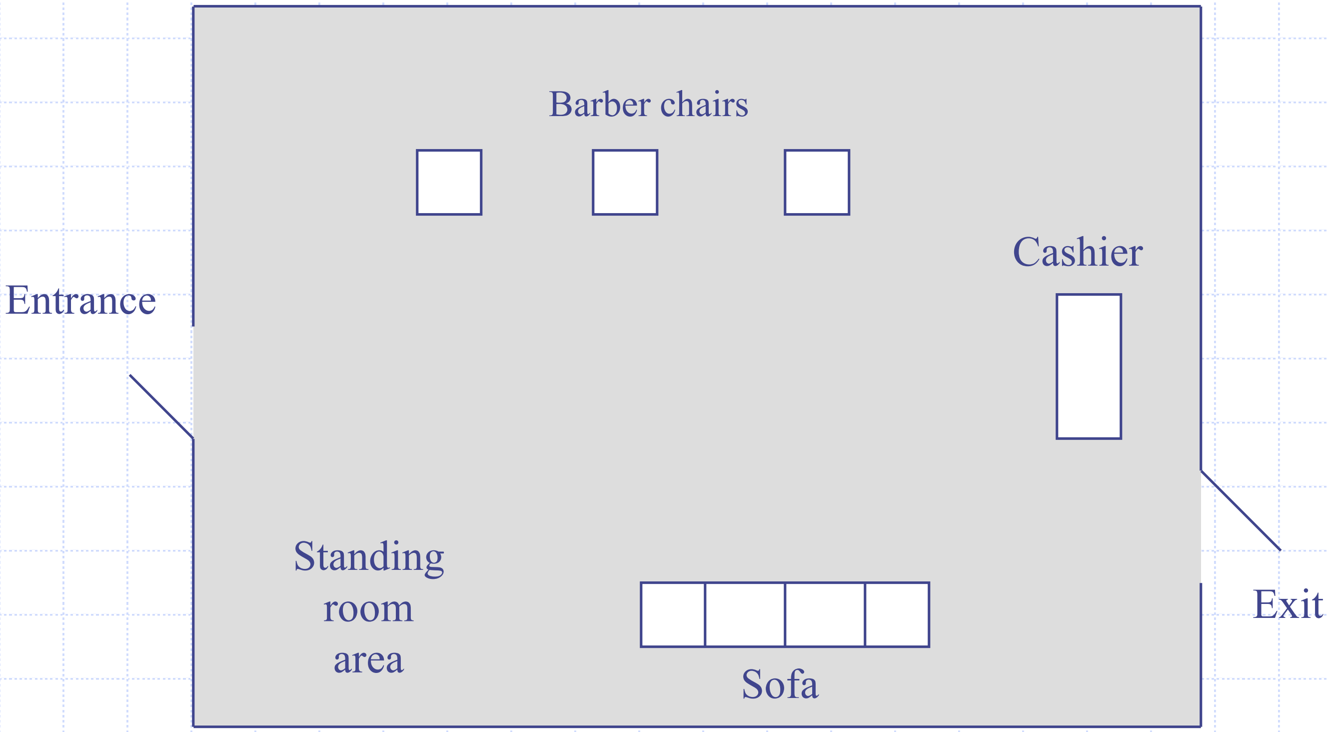
Producer/Consumer Problem with bounded buffers



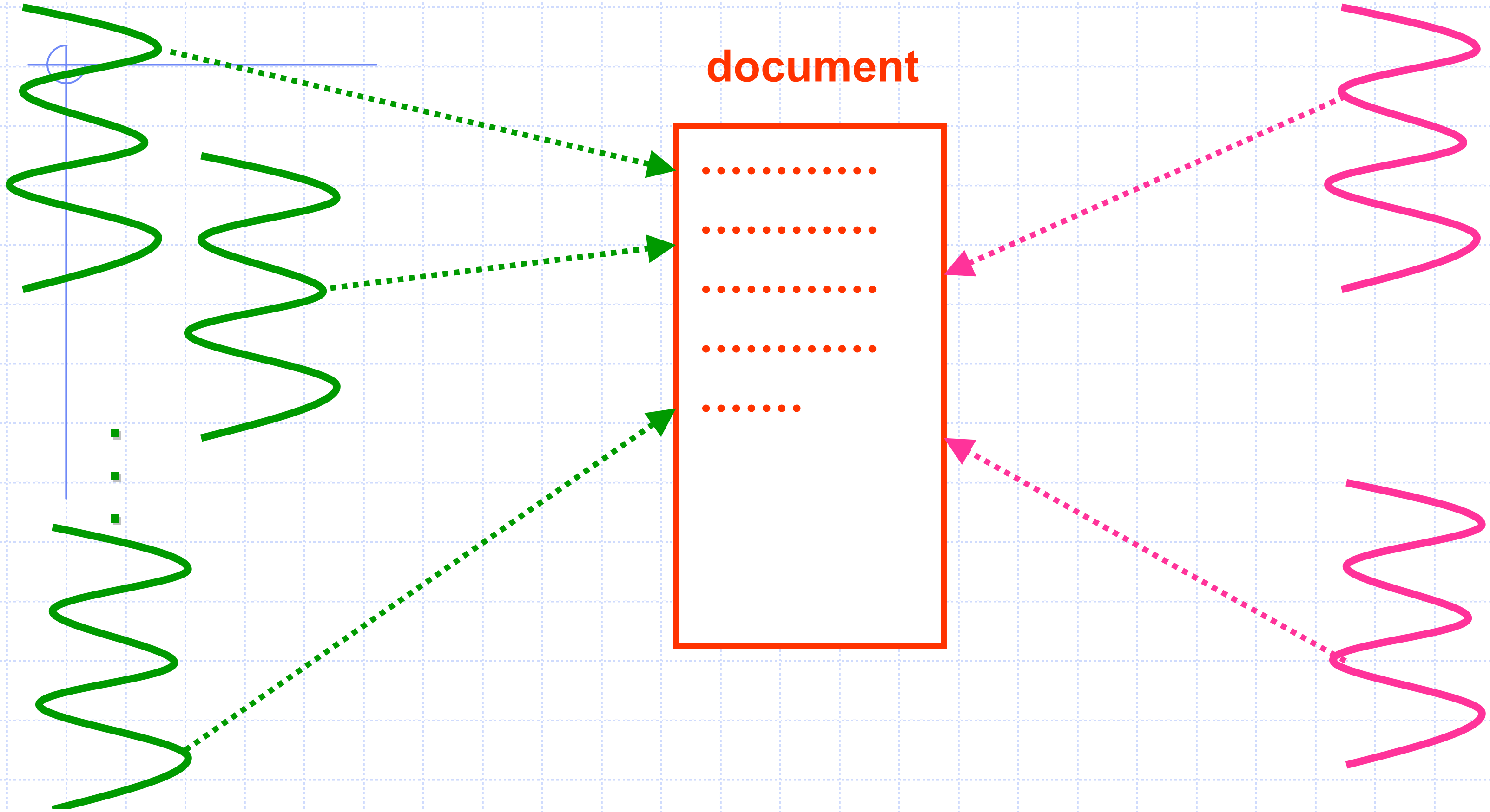
Additional problems with a bounded buffer?

Additional problems with p>1 producer and/or respectively c>1 consumer?

Barber Shop Problem



Reader/Writer Problem



Which Problems may occur?

Data Consistence

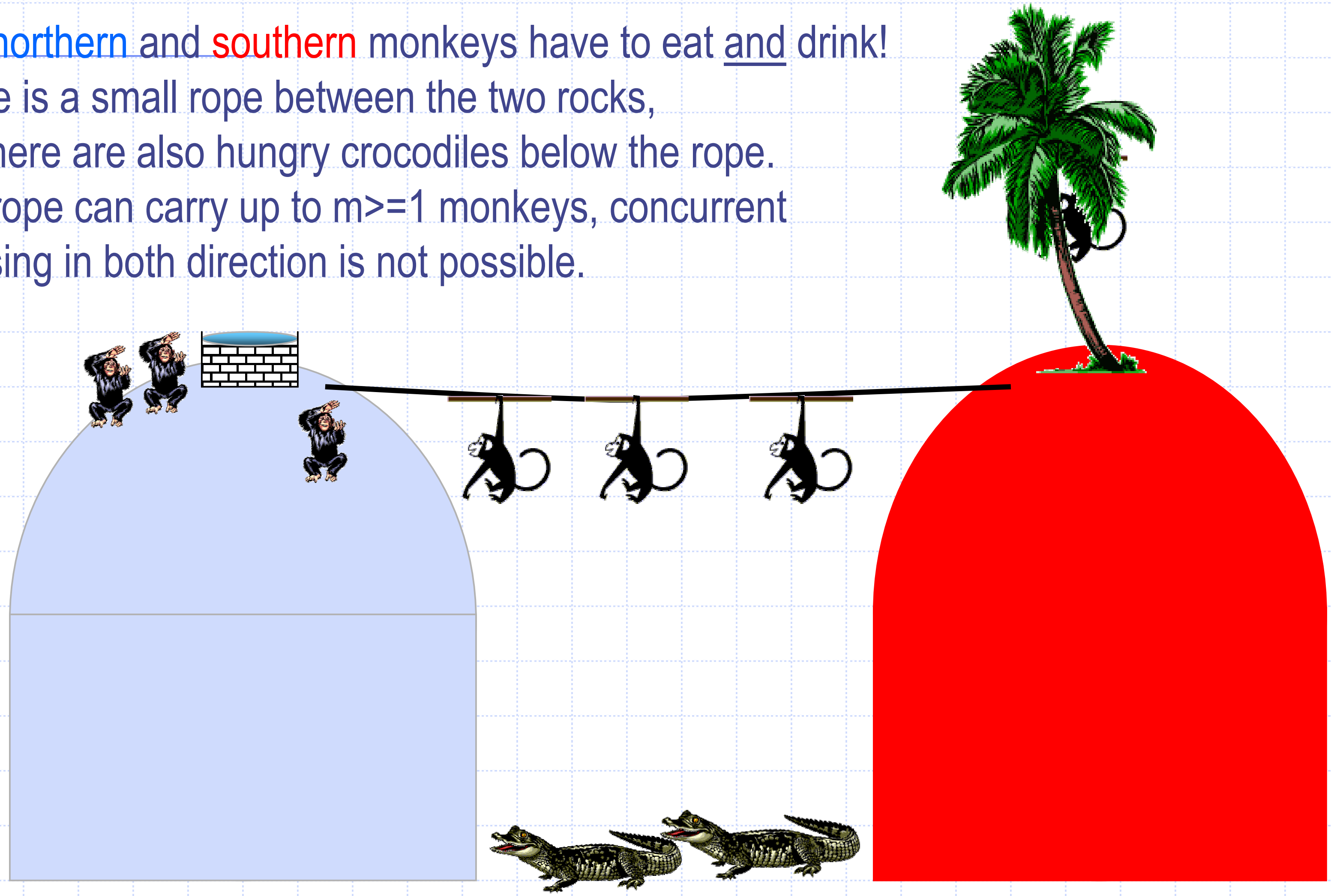
Patil's Smoker Problem

Given: A vendor store for smoking ingredients
3 chain smokers and the sleeping vendor
For smoking a smoker needs: tobacco, paper and matches
Smoker A has his own tobacco
Smoker B has his own paper
Smoker C has his own matches

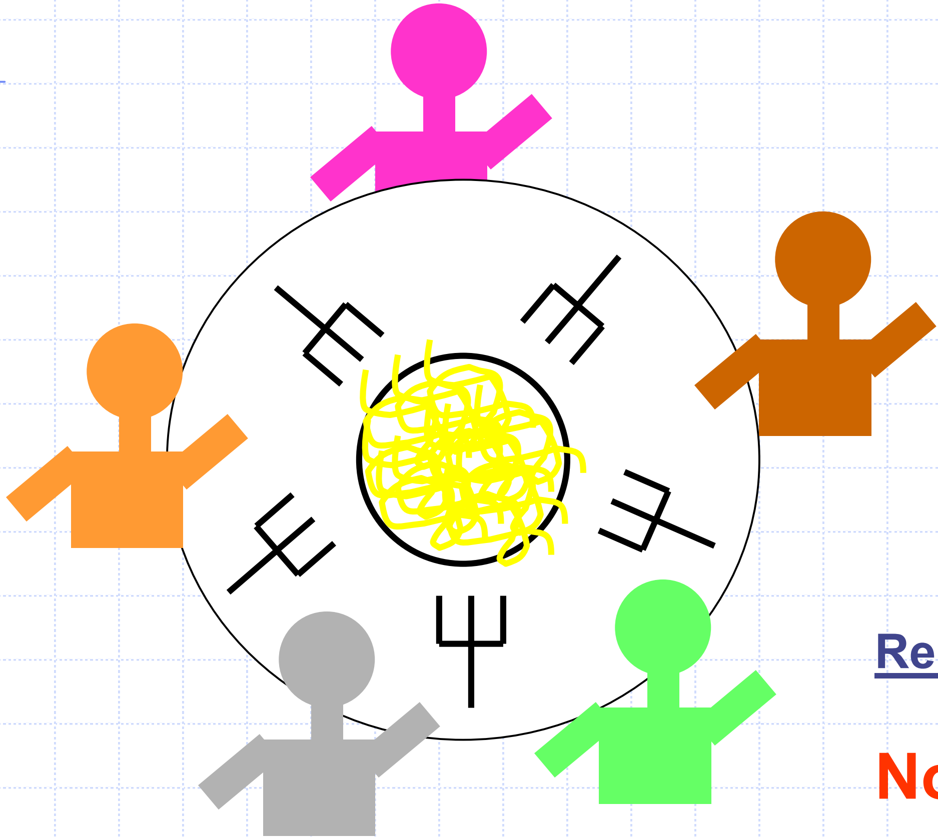
Each time the sleeping vendor is awakened he puts two smoking ingredients on his table allowing one of the smokers to continue his unhealthy pleasure. After smoking, the smoker wakes up the vendor again who then puts another pair of smoking ingredients (at random) thus unblocking another smoker.

Monkey Rock Problem

The northern and southern monkeys have to eat and drink!
There is a small rope between the two rocks,
but there are also hungry crocodiles below the rope.
The rope can carry up to m>=1 monkeys, concurrent
crossing in both direction is not possible.



Dijkstra's famous Dining Philosopher Problem



Life of a philosopher:

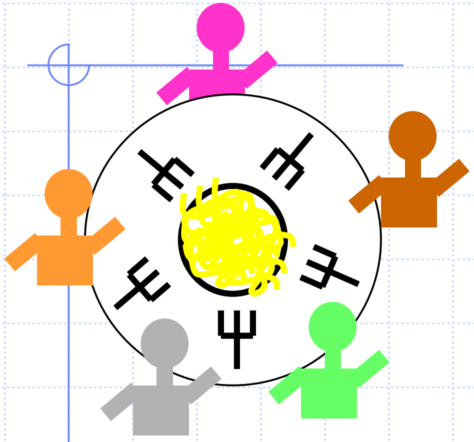
```
repeat forever
begin
 thinking
 getting hungry
 getting 2 forks
 eating
end
```

Requirements:

No Deadlock!

No Starvation!

Dijkstra's Dining Philosophers - solution I



Life of a philosopher:

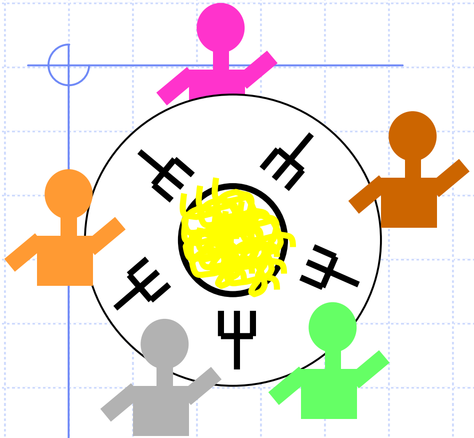
```
do
 think
 get hungry
 acquire 2 forks
 eat
 release forks
od
```

```
think:  do
          if neighbor requests fork
             then give it to him
          fi
        until hungry od.

hungry: do
          if fork missing
             then request it from neighbor
          fi
          if neighbor requests fork
             then give it to him
          fi
        until have both forks od.
```

deadlock free ?

starvation

Dijkstra's Dining Philosophers - solution Ia



Life of a philosopher:

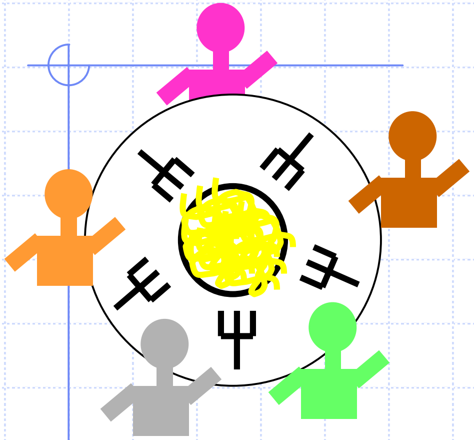
```
do
  think
  get hungry
  acquire 2 forks
  eat
  release forks
od
```

```
think:   do
           if neighbor requests fork
              then give it to him
           fi
         until hungry od.

hungry:  do
           if fork missing
              then request it from neighbor
              if got dirty fork
                 then clean it
              fi
           fi
           if neighbor requests dirty fork
              then give it to him
           fi

         until have both forks od.
```

no starvation ?

deadlock free ?

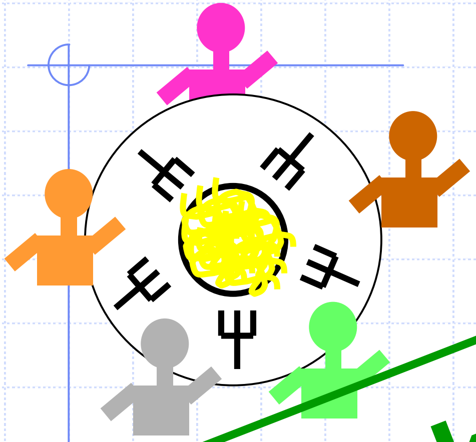
Dijkstra’s Dining Philosophers - solution Ia



conclusion from solution I:

- need communication operations for communicating sequential processes, e.g.
  - send request
  - receive request

Dijkstra's Dining Philosophers - solution II



Life of a philosopher:

```
do
  think
  get hungry
  acquire 2 forks
  eat
  release forks
od
```

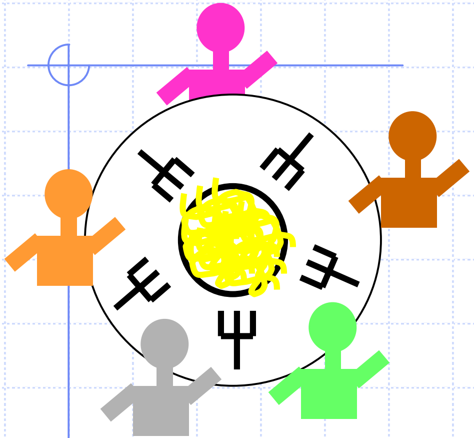
atomic operation

```
Enter critical section
  get left fork
  get right fork
leave critical section
```

deadlock free

starvation

Dijkstra's Dining Philosophers - solution II



conclusion from solution II:

- need synchronization operations on shared data objects, e.g.

?

Signaling and Synchronization

Long History of Signaling Techniques

• **Drums**	(Bush Men)
• **Smoke Signals**	(Red Indians)
• **Signal Fires**	(e.g. in War Times)
• **Signal Flags**	(e.g. Marine)
• **Signal Whistles**	(e.g. Sports)
• **Signal Colors**	(e.g. Traffic Lights, “Mimikri”)
• **Signal Perfumes**	(e.g. Mammals)
• **…**	

Potential meanings of a Signal

- "Pay Attention" (see a siren)
- "Stop" (see road signs)
- "Go Ahead" (see whistle of a station officer)
- "Interrupt" (see whistle of an arbiter in a soccer game)
- ...

=> You must know the exact meaning of each signal
otherwise your reaction on a signal may be wrong

Implementing a Signal

- **Flag** **1 = Signal set, 0 = Signal reset**

  - Continuation (signaled thread may continue)
  - Stop (signaled thread has to wait)
  - Abort (signaled thread has to be aborted)

  …

  see “signal vector” in Unix or Linux

- **Counter** Any value may have a different meaning or just reflects the number of pending signals

Problem: Try to find out when a flag is sufficient and when you’ll need a counter!

Synchronizing a Precedence Relation

```
{Thread 1}
.
.
.
{section a1
   … }

{ section a2
   … }
.
.
.
```

```
{Thread 2}
.
.
.
{ section b1
   … }

{ section b2
   … }
.
.
.
```

Problem: *How to achieve that a1 <* b2 (a1 precedes b2), i.e. section b2 cannot be executed before section a1 has completed?*

Implementing a Precedence Relation

```
1:1_signal s;
```

```
{Thread 1}
.
.
.
{section a1
  … }
Signal(s)
{ section a2
  … }
.
.
.
```

```
{Thread 2}
.
.
.
{ section b1
  … }
Wait(s)
{ section b2
  … }
.
.
.
```

Problem: *How to implement a 1:1_signal_object?*

Principal Types of Solutions

Software solutions

algorithms neither relying on special hardware nor OS features

Hardware solutions

relying on some special machine instructions

Operating System solutions

providing kernel functions to system and application programmers

Remark: Most systems offer just a subset of the above types of solutions.

Software Solutions

Flag *s* as common variable of both threads

What happens if 'signal' is invoked prior to 'wait' ?

What may happen if 'wait' is invoked prior to 'signal' ?

Discuss this "proposal" carefully!

Does it works on any system?

Is it effective and/or efficient?

Implemention of a 1:1 signal in software, version I1

```
module 1:1_signal
  export Signal, Wait;
  import yield;
  type signal = record
        S: signal := reset
  end

  procedure Signal(SO:signal)
   begin
   SO.S := set;
   yield();  /* anonymous yield */
   end

  procedure Wait (SO:signal)
   begin
   while SO.S = reset do
         yield()
   od
   SO.S :=reset
   end
end module
```

Requires cooperative scheduling

No signal loss ?

Efficient ?

Scalable ?

Implemention of a 1:1 signal, version II

```
module 1:1_signal
 export Signal, Wait
 import block, unblock
 type signal = record
     W: thread := nil
 end

 procedure Signal(SO:signal)
  begin
  unblock (SO.W);
  end

 procedure Wait (SO:signal)
  begin
  SO.W := myself ;
  block (myself) ;
  SO.W := nil
  end
end module
```

No signal loss?

Efficient !

Scalable !

Implementing a 1:1 signal, version III

```
module 1:1_signal
 export Signal, Wait
 import UnblockThread, BlockThread
 type signal = record
     S: signal := reset
     W: waiting thread := nil
 end

 procedure Signal(SO:signal)
  begin
  SO.S := set;
  if SO.W ≠ nil then                      {A thread is waiting?}
         UnblockThread(SO.W)                      {unblock it}
  end

 procedure Wait (SO:signal)
  begin
  while SO.S = reset do
        SO.W := myself
        BlockThread(myself)
  od
  SO.S :=reset
  end
end module
```

No signal loss !

Efficient !

Scalable !

Race condition !

Open Problem

Conclusion:

Operations wait and signal should be atomic!
(We'll postpone how to achieve this property.)

Synchronization of a Mutual Precedence Relation

```
{Thread 1}                {Thread 2}
.                         .
.                         .
.                         .
{section a1               {section b1
   … }                       … }


{section a2               { section b2
   … }                       … }
.                         .
.                         .
.                         .
```

Problem: *How to achieve a1 <* b2 and b1 <* a2 ?*

Solution of a Mathematician

Idea: Tranform the new problem
to an already solved one
and take the known solution

```
1:1_signal s1,s2;

{Thread 1}
.
.
.
{section a1
  … }
Signal(s1)
Wait(s2)
{section a2
  … }
.
.
.

{Thread 2}
.
.
.
{section b1
  … }
Signal(s2)
Wait(s1)
{ section b2
  … }
.
.
.
```

Remark: Discuss Pros and Cons!

Mutual Precedence Relations via signal and wait

Pros:

No extra Mechanism for a related Problem

Cons:

Complicated for n > 2 Threads

Low Performance due to many Kernel Calls

Let´s invent a new mechanism with a better behavior!

Synchronize Operation for Mutual Precedence Relations

```
sync s
```

{Thread 1}	{Thread 2}
.	.
.	.
.	.
{section a1	{section b1
… }	… }
Synchronize(s)	Synchronize(s)
{section a2	{section b2
… }	… }
.	.
.	.
.	.

Problem: *How to implement a synchronization module for two threads ?*

Synchronize Operation for Mutual Precedence Relations

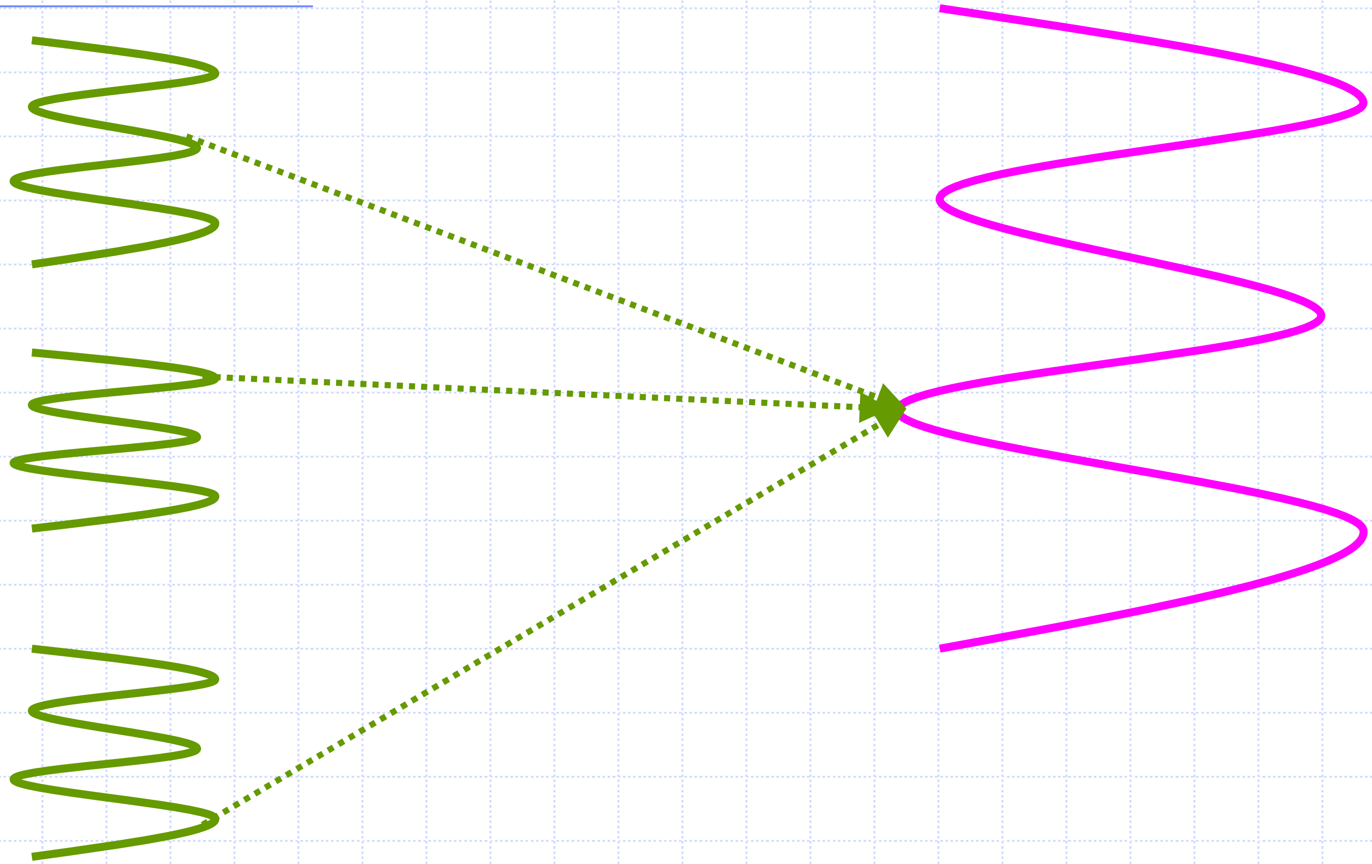
```
module synchronization
 export synchronize
 import UnblockThread, BlockThread
 type sync = record
        S: signal := reset
        W: waiting thread := empty
 end

 procedure Synchronize(SY:sync)
  begin
  if SY.S = reset
  then begin                            {I am first}
              SY.S := set
              SY.W := myself
              BlockThread(myself)    {and wait for my partner}
        end
  else begin                            {I am second and}
              UnblockThread(SY.W)    {release my partner and}
              SY.S := reset          {do a reset for future reuse}
        end
  end
end module
```

Exercise: Generalize this module for n>2 threads!

Application of a Multiple Synchronize Operation

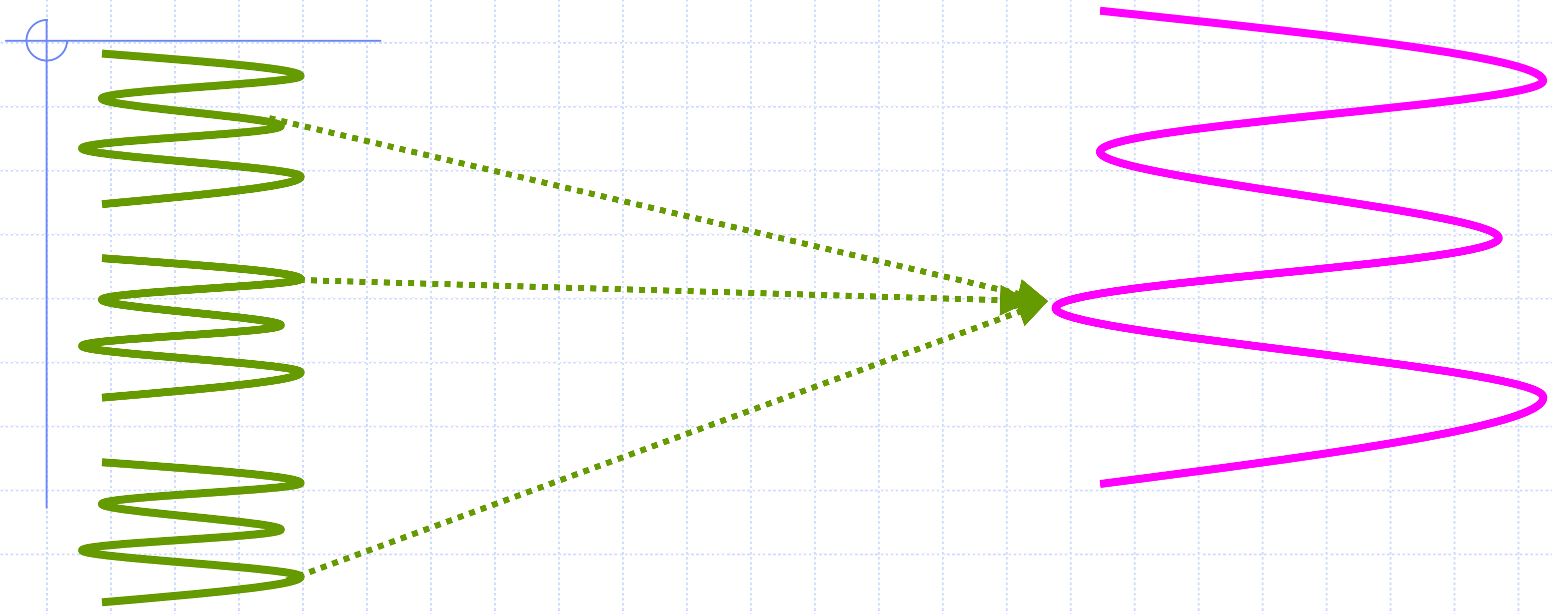
```
. {some numerical problem solved via difference equations}
.
.
while true do
  begin
  for all i,j
     begin
     temp[i,j] := old[i-1,j] + old[i+1,j]
     end
  n_synchronize(S)
  for all i,j
     begin
     old[i,j] := temp[i,j]
     end
  n_synchronize(S)
  end
.
.
.
```

More Patterns on Signaling: Many to One



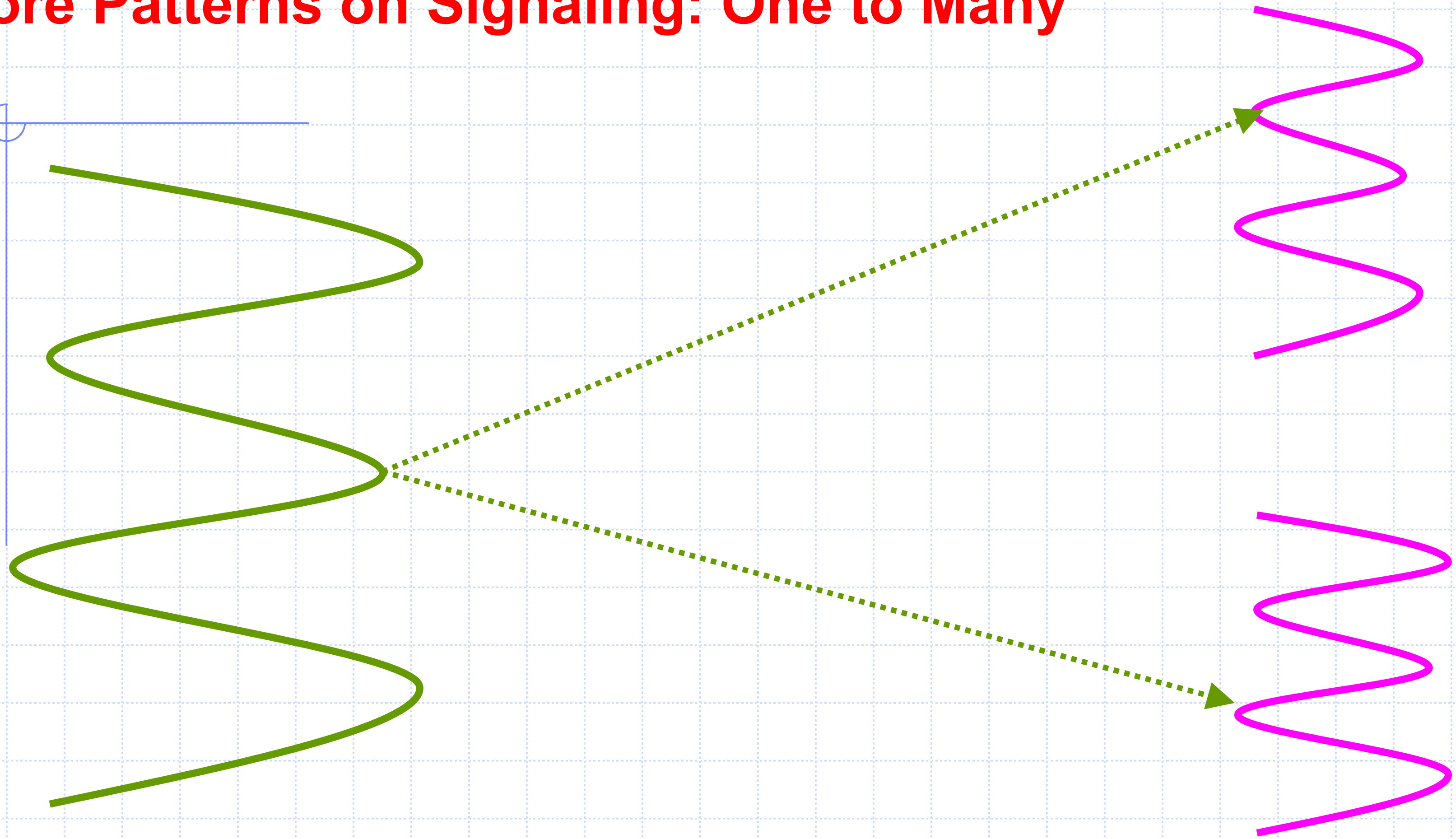
The pink thread may continue iff the 3 other threads passed certain code sections.

Example for a Many to One Signaling Relation



These 4 threads form a team working together on some application problem.
3 production threads calculate intermediate results (buffering in shared memory).
Any of the productions threads signals when he finished another calculating step.
The display thread, averaging these 3 intermediate results, displays the average.

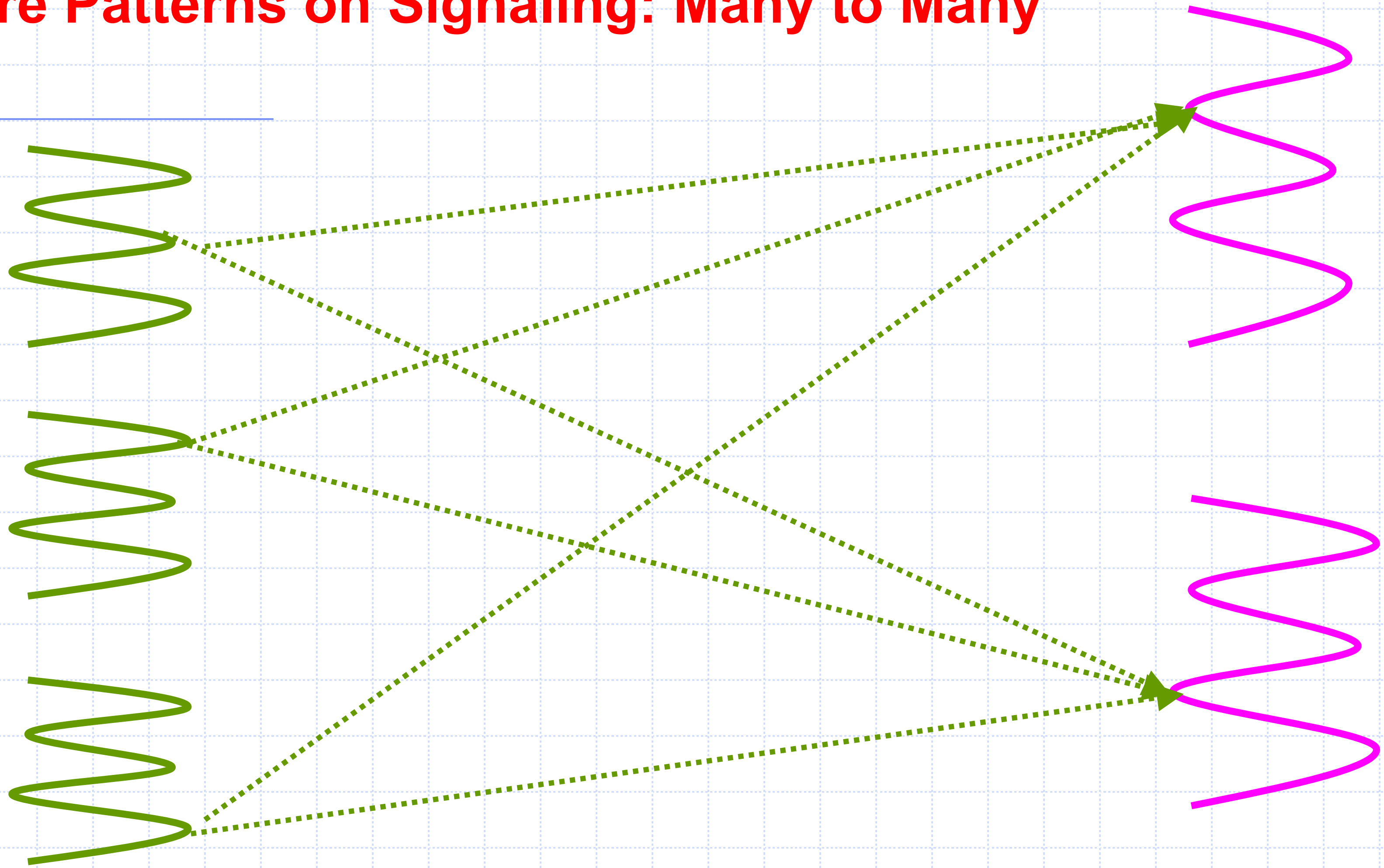
More Patterns on Signaling: One to Many



The 2 pink threads may continue iff the green thread passed a certain code section

Think over an application for this pattern!

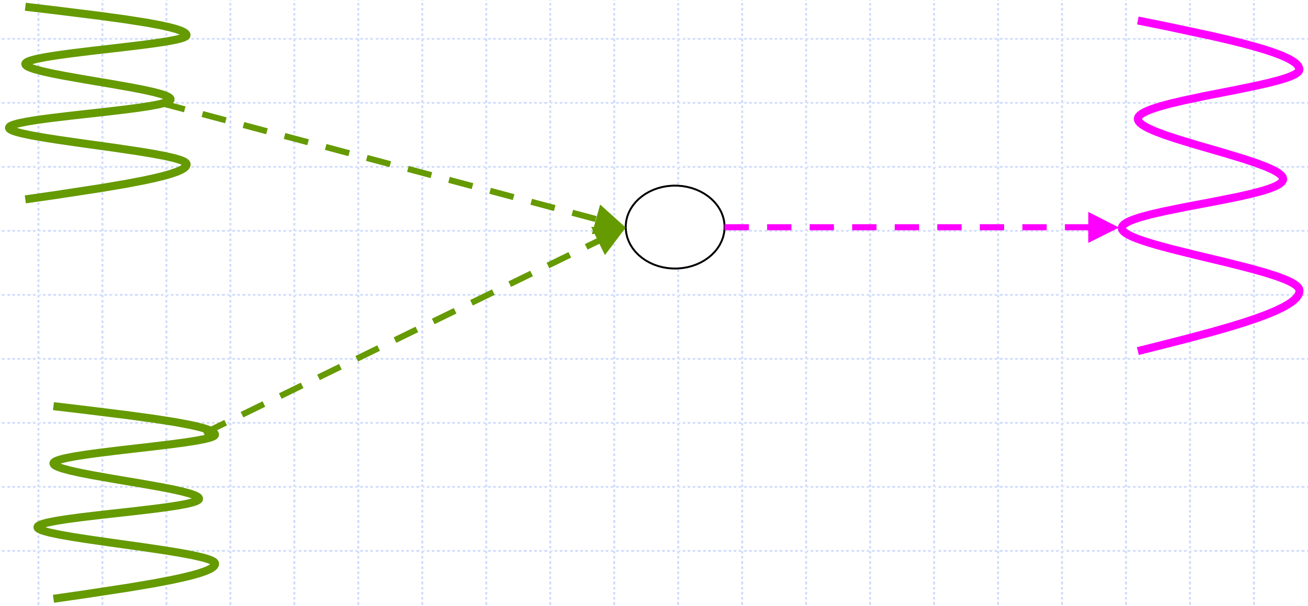
More Patterns on Signaling: Many to Many



2 pink threads may continue iff 3 thread passed certain code sections

Think over an application for this pattern!

Further Patterns of Signaling



It's sufficient for the pink thread that one of the 2 signaling green threads has delivered a signal.

Additional Problem: How to buffer signals?

Buffering Signals

- Each incoming signal is buffered until a potential waiting thread consumes this signal

  Pro: Reaction on each signal

  Con: A deficient signaling source may flood the system

- An incoming signal may overwrite a previous one (e.g. binary semaphore)

  Pro: Reaction only on the newest signal

  Con: Danger of lost signals

Dijkstras (Counting) Semaphores

A semaphore S is an integer variable that, apart from initialization, can only be accessed by 2 atomic and mutually-exclusive operations:

P(S) P ~ Passeren (some dutch signaling language)

V(S) V ~ Verlaaten (see above)

To avoid busy waiting: when a thread cannot "passeren" inside of P(S) it will be put into a blocked queue waiting for an event to be done inside of V(S) by some other thread (hopefully, otherwise it would starve).

Dijkstras (Counting) **Semaphores**

Semantic of a Counting Semaphore for signaling purpose:

1. A **positive** value of the counter indicates how many signals are pending

2. A **negative** value of the counter indicates how many threads are waiting for a signal,
   i.e. are queued within the semaphore object

3. If the **counter = 0** => no thread is waiting for a signal

Dijsktra's: Counting Semaphores (historical 1. solution)

```
module semaphore
 export P,V
 import BlockThread, UnblockThread
 type semaphore = record
      Count: integer := 0                {no signal is pending}
      QWT: list of Threads = empty {queue of waiting threads for sema}
      end

 P(S:semaphore):
        S.Count := S.Count - 1
        if S.Count < 0 then
          insert (S.QWT, myself)          {1 more waiting thread}
          BlockThread(myself)
        fi .

 V(S:semaphore)
        S.Count := S.Count + 1            {1 additional signal}
        if S.Count <= 0 then
         unblockThread(delete first(S.QWT))
        fi .
end
```

Dijsktra's: Counting Semaphores

Remark: Semaphores can also solve the next class of coordination problems:

Critical Sections

The Critical Section Problem (1)

When a thread executes code that manipulates shared data (or resource), we say that the thread is in it's critical section (CS) (for that shared data or resource)

The execution of critical sections must be mutually exclusive: at any time, only one thread is allowed to execute its critical section (even with multiple CPUs)

Then each thread must request the permission to enter it's critical section (CS)

The Critical Section Problem (2)

- The section of code implementing this request is called the entry section
- The critical section (CS) should be closed by a leave section
- The remaining code (outside of CS) is the remainder section (RS)

What do we really need to solve critical section problems?

We have to establish and offer a serialization protocol so that the results of the threads will not depend on the order in which their execution has been interleaved.

Framework for the Analysis of the Solutions

Each thread executes at nonzero speed but no assumption on the relative speed of n threads
General structure of a thread:

Multiple CPUs may be present but memory hardware prevents simultaneous access to the same memory location.
No ordering assumptions for interleaved executions

```
{A general pattern of all threads participating
 in the critical section problem}
  repeat
    enter section
     CS {critical section}
    exit section
     RS {remainder section}
  forever
```

3 Requirements for a Valid Solution

Mutual Exclusion:

At any time, **at most one** thread can be in its critical section (CS)

Progress

If no thread is executing in its CS while some threads wishes to enter, only threads that are not in their RS can participate in the decision: which thread will be the next. This selection cannot be postponed indefinitely

Bounded Waiting

After a thread has made a request to enter it's CS, there is a bound on the number of times that the other threads are allowed to enter their CS otherwise the thread will suffer from starvation

Different Types of Solutions

- Software solutions
  - algorithms neither relying on special hardware nor OS features
- Hardware solutions
  - relying on some special machine instructions
- Operation System solutions
  - provide some kernel functions to the programmers

Software solutions

- First consider only 2 threads
  - Algorithm 1 and 2 will be uncomfortable respectively incorrect
  - Algorithm 3 is correct (Peterson's algorithm)
- Then consider a solution for n threads
  - Bakery algorithm
- Notation
  - We start with 2 threads: T_0 and T_1
  - When presenting thread T_i, T_j always denotes the other thread (i != j)

Algorithm 1

The shared variable turn is initialized (0 or 1) before executing any T_i

T_i's critical section is executed iff turn = i

T_i is busy waiting if T_j is in CS $\Rightarrow$ mutual exclusion is satisfied

Progress requirement is not satisfied since it requires strict alternation of CSs

```
thread Ti:
repeat

while(turn!=i){};
      CSi
   turn:=j;
      RSi
forever
```

Analysis: Suppose T_0 has a large RS_0, whereas T_1 has a small RS_1.
If turn=0, T_0 may enter CS_0, leaves it (turn=1), then executes its long RS_0.

Meanwhile T_1 was also in CS_1, leaves it (turn=0), then executes its RS_1
Then T_1 tries in vain to enter CS_1! T_1 must wait until T_0 leaves its long RS_0

Additional Requirement: The length of an RS shouldn't have any influence on the execution sequence of the participating threads!

Algorithm 2

Keep 1 Boolean variable for each thread: flag[0] and flag[1]
T_i signals that it is ready to enter it's CS by setting: flag[i]:=true
Mutual Exclusion is satisfied but not the progress requirement

```
thread Ti:
repeat
   flag[i]:=true;
   while(flag[j]){};
      CS
   flag[i]:=false;
      RS
forever
```

Analysis: Suppose the following execution sequence holds:
T_0: flag[0]:=true
T_1: flag[1]:=true

What will happen?

Both threads will wait forever, none of them can enter its CS (“***DEADLOCK***”)

Algorithm 3 (Peterson's solution)

Initialization: flag[0]:=flag[1]:=false, and turn:= 0 (or 1)

Willingness to enter CS specified by flag[i]:=true

If both threads attempt to enter their CS simultaneously, only one turn value will last

Exit section: specifies that T_i is unwilling to enter CS

```
thread Ti:
repeat
  flag[i]:=true;
  turn:=j;
  do {} while
  (flag[j]and turn=j);
     CS
  flag[i]:=false;
     RS
forever
```

"Proof" of correctness for Algorithm 3

- To prove that mutual exclusion is preserved:
  - T_0 and T_1 are both in their CS only if flag[0] = flag[1] = true and only if turn = i for each T_i (which is impossible)

- To prove that the progress and bounded waiting requirements are satisfied:
  - T_i cannot enter CS only if stuck in '*while()..*' with condition '*flag[j]* ' and '*turn = j* '.
  - If T_j is not ready to enter CS then ' *! flag[j]* ' and T_i can enter its CS
  - If T_j has set *'flag[j]'* and is in its *'while()..'*, then either *turn=i* or *turn=j*
  - If *turn=i*, then T_i enters CS. If *turn=j* then T_j enters CS, but it will reset *flag[j]* on exit: allowing T_i to enter CS
  - but if T_j has time to set *flag[j]*, it must also set *turn=i*
  - since T_i does not change value of *turn* while stuck in '*while()..*', T_i will enter CS after at most one CS entry by T_j (bounded waiting)

What about Faulty Threads?

If all 3 main criteria (mutual exclusion, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failures within the remainder section (RS) of a thread

Failures within RS are like infinitely long RS, i.e. they should not affect the other thread(s)

However, no valid solution can ever provide any robustness, if a thread fails within its critical section (CS). *Why?*

A Thread failing within its critical section may never perform **`exit_section`**, i.e. neither a signal-operation nor a V(S) nor something comparable thus affecting the other thread(s) severely!!

Bakery Algorithm for n threads

- Before entering their CS, each T_i receives a number. The holder of the smallest number enters its CS (as it is at least in English bakeries, ...)

  If T_i and T_j receive the same number:

  if i < j then T_i is served first, else T_j is served first

  T_i resets its number to 0 in its exit section

  Notation:

  (a,b) < (c,d) if a < c or if a = c and b < d

  max(a_0,...a_k) is a number b such that: b >= a_i for i=0,..k

  Shared data:

  choosing: array[0..n-1] of boolean; initialized to false

  number: array[0..n-1] of integer; initialized to 0

Correctness relies on the following fact:

If T_i is in CS and T_k has already chosen its number[k] != 0,

then (number[i],i) < (number[k],k)

Bakery Algorithm

```
thread Pi:
repeat
  choosing[i]:=true;
  number[i]:=max(number[0]..number[n-1])+1;
  choosing[i]:=false;
  for j:=0 to n-1 do {
    while (choosing[j]) {};
    while (number[j]!=0
        and (number[j],j)<(number[i],i)){};
  }
  CS
  number[i]:=0;
  RS
forever
```

Drawbacks of Software Solutions

Threads requesting to enter their critical sections are busy waiting (consuming processor time needlessly)

- If the critical sections are long enough, it would be more efficient to block threads.

Hardware solutions: *interrupt disabling*

Single processor: mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other threads being in their RS

- Multi processor: mutual exclusion is not preserved at all

- Delay of interrupt handling may affect the whole System

- Application programmers may abuse this facility

```
thread Ti:
repeat
  disable interrupts
   critical section
  enable interrupts
   remainder section
forever
```

<u>Summary:</u> In general not an acceptable solution!

Hardware solutions: *special instructions*

(1) some machines offer instructions that perform ***read-modify-write*** operations atomically (indivisible, on the same memory location):

```
inc     [mem]
xchg    [mem],reg
bts     [mem]          {bit test and set}
```

- (2) some machines offer conditional LD/ST instructions instead:
  - LDL [mem] processor becomes sensitive for memory address *mem*
  - STC [mem] fails if another processor executed STC on the same address in the meantime
- instructions like (1) execute mutually exclusive on multiple CPUs
- like (2) permit to emulate mutually-exclusive instructions

The test-and-set instruction

- An algorithm that uses testset for mutual exclusion:
- Shared variable b is initialized to 0
- Only the first T_i -having set b- can enter its CS

```
boolean testset(int& i)
{ if (i=0) {
     i:=1;
     return true;
  } else
     return false;
}
```

```
thread Ti:
repeat
  repeat{}
  until testset(b);
     CS
  b:=0;
     RS
forever
```

Analysis of the test-and-set solution (1)

Mutual exclusion is preserved: if T_i enters its CS, the other T_j perform busy waiting. Hence there is an **efficiency problem**

When T_i exits its CS, the selection of the T_j who will enter CS is arbitrary: no bounded waiting. Hence **starvation** is possible!

Processors (ex: Pentium) often provide an atomic xchg(a,b) or *compare-and-swap* instruction that swaps the content of a and b.
(But xchg(a,b) suffers from the same drawbacks as test-and-set)

*However, what's the biggest problem with one of these "**spin lock**" solutions?*

Analysis of the test-and-set solution (2)

Repeated test-and-set-instructions may monopolize the system bus affecting each other system activity (if related to that CS or if not)

This is a severe danger for another sort of "**deadlock**" on a Single Processor System

Solutions with OS/PL Support

- **Dijkstras** (Counting) **Semaphores**
- **Dijsktras** (Binary) **Semaphores**
- **Monitors**
- **Transactions**

Dijkstras (Counting) **Semaphores**

Semantic of a Counting Semaphore for **mutual exclusion** of critical sections:

1. A **positive** value of the counter indicates how many threads still may enter the critical section
2. A **negative** value of the counter indicates how many threads are waiting in front of the critical section, i.e. are queued within the semaphore object
3. If the **counter = 0** => no thread is waiting respectively maximally allowed threads are in the critical section

Still the open problem:

How can we get "atomic semaphore-operations" ?

Atomic Semaphore Operations

Problem:
P() and V() each consisting of multiple machine instructions have to be atomic!

Solution:
Need "another" sort of critical sections, hopefully with shorter execution times, to establish atomic semaphore operations!!

"very short" enter_section

P(S)

"very short" leave_section

Revisiting Dijkstras (Counting) Semaphores

The critical sections required for implementing P(S) and V(S) are very short: typically only 10 instructions.

Solutions (to establish these short critical sections around P(S) or V(S)):

- single processor: disable interrupts during those operations
- multi processor: use previous software or hardware schemes

Atomic Counting Semaphores (Single Processor)

```
P(sema S)
 begin
 DisableInterupt
   s.count--
   if s.count < 0then
     BlockThread(S)
   fi
 EnableInterrupt
 end
V(sema S)
 begin
 DisableInterrupt
   s.count++
   if s.count <= 0 then
     UnblockThread(S)
   fi
 EnableInterrupt
end
```

Atomic Counting Semaphores (Multi Processor)

```
P(S)
begin
 while (!TAS(S.flag)){}; busy waiting, flag is sema data
   S.Count:= S.Count-1
   if S.Count < 0 then
     BlockThread(S) and S.flag := 0
   else S.flag :=0
   fi
end
V(S)
begin
 while (!TAS(S.flag)){}; busy waiting
   S.Count:= S.Count+1
   if S.Count <= 0 then
     UnblockThread(S)
   fi
 S.flag :=0
end
```

Next Problem Solving Exercise