System Architecture

9 Mutual Exclusion

Critical Section & Critical Region
Busy-Waiting versus Blocking
Lock, Semaphore, Monitor

November 24 2008
Winter Term 2008/09
Gerd Liefländer

Agenda

- HW Precondition
- Mutual Exclusion
  - Problem
  - Critical Regions
  - Critical Sections
- Requirements for valid solutions
- Implementation levels
  - User-level approaches
  - HW support
  - Kernel support
- Semaphores
- Monitors

Literature

Bacon, J.:	Operating Systems (9, 10, 11)
Nehmer,J.:	Grundlagen moderner BS (6, 7, 8)
Silberschatz, A.:	Operating System Concepts (4, 6, 7)
Stallings, W.:	Operating Systems (5, 6)
Tanenbaum, A.:	Modern Operating Systems (2)

- Research papers on various locks

Atomic Instructions

- To understand concurrency, we need to know what the *underlying indivisible HW instructions* are

- Atomic instructions run to completion or not at all
  - It is indivisible: it cannot be stopped in the middle and its state cannot be modified by someone else in the middle
  - Fundamental building block: without atomic instructions $\Rightarrow$ we have no way for threads to work together properly

- **`load, store`** of **`words`** are usually atomic

- However, some instructions are **not atomic**
  - VAX and IBM 360 had an instruction to **copy a whole array**

Mutual Exclusion

Mutual Exclusion Problem

Assume at least two concurrent activities

1. Access to a physical or to a logical resource or to shared data has to be done exclusively
2. A program section of an activity, that has to be executed indivisibly and exclusively is called critical section CS[1]
3. We have to establish a specific execution protocol in front and after each CS in order to provide its mutual exclusive execution
4. Activities executing a CS are either threads or processes[2]

[1]Some textbooks require atomic critical sections

[2]In the kernel exception/interrupt handlers also have CSs

Example: Critical Section

```
integer a, b =1;          {shared data}
 {Thread 1}                    {Thread 2}
 while true do                 while true do

   a = a + 1;                   b = b + 2;
   b = b + 1;                   a = a + 2;

 {do something else}           {do something else}
 od                            od
```

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

Critical Regions

- All related CSs in the threads of a multi-threaded application form a critical region
- A CS is related to another one iff both CSs should not run concurrently, e.g. in case they access
  - an exclusive resource
  - the same global data
- Non related CSs can be executed concurrently at will

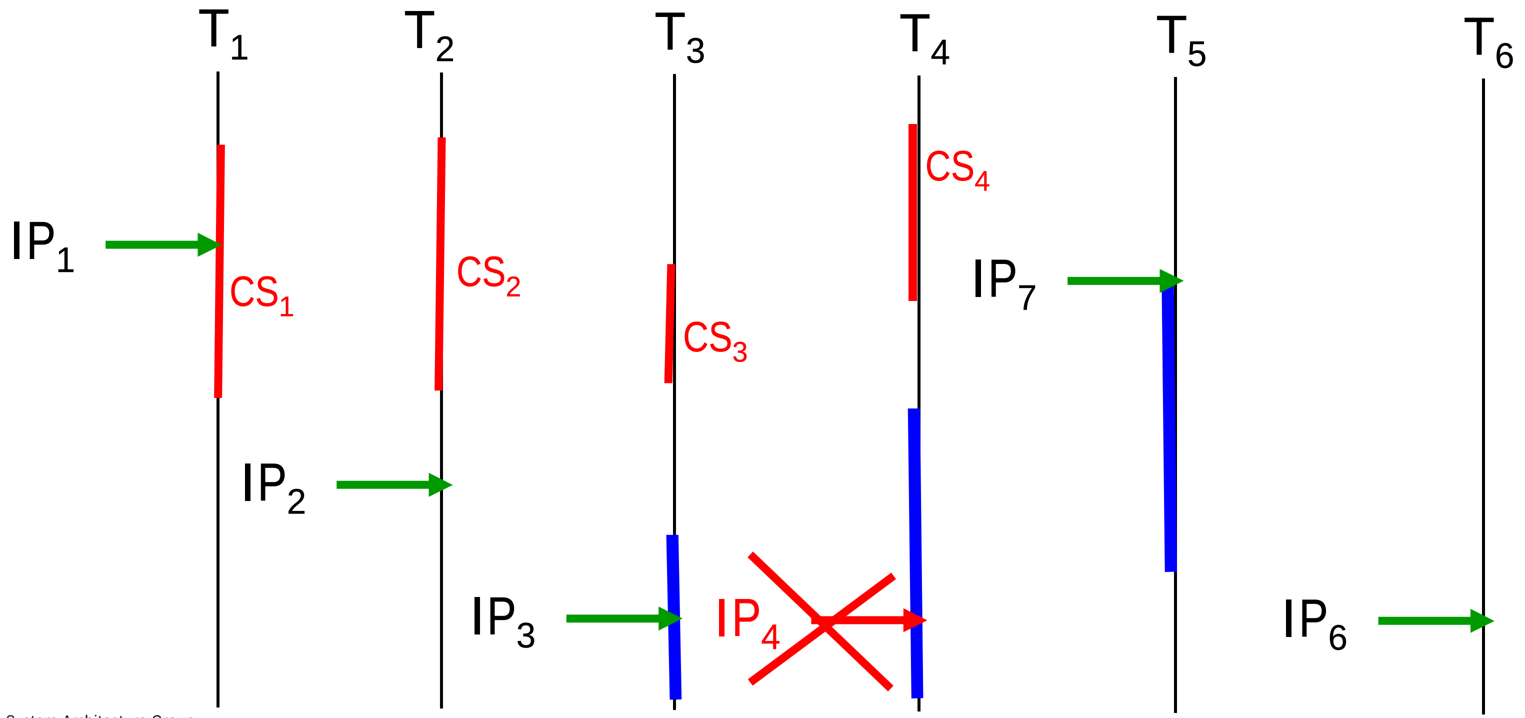
Example: Critical Regions

Suppose: All T_i are KLTs of the same application task
the IP of T_1 is in its “red CS”

All red CSs build up the red critical region CR

All blue CSs build up another CR

__Question:__ What other IP_i are valid at the same time?



Framework of Critical Sections

Sections of code implementing this protocol:

- enter_section
- critical section (CS)
- exit_section

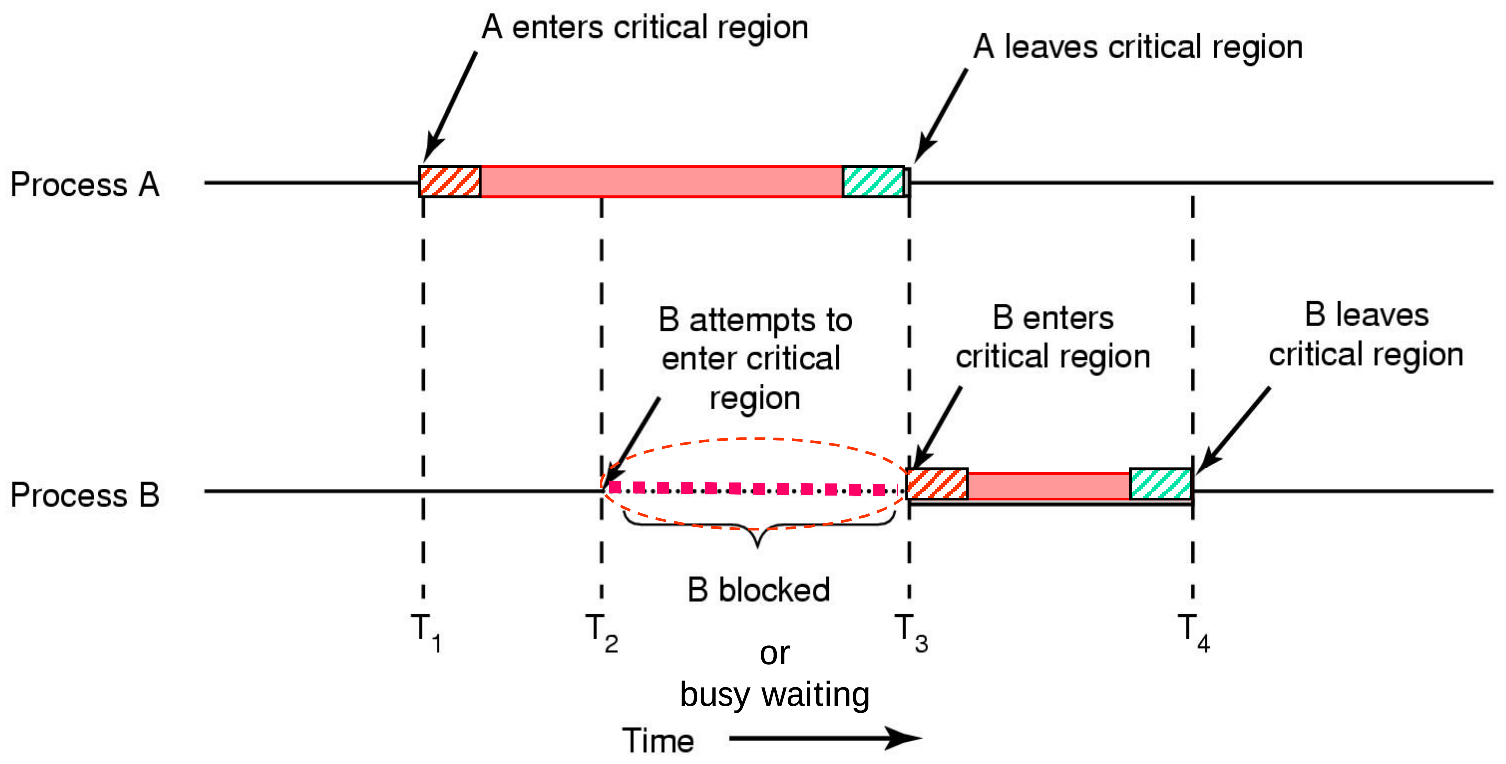
The remaining code (outside of a CS):

- remainder section (RS)

We need a serialization protocol:

Results of involved threads no longer depend on arbitrary interleaving of their execution phases

2 Mutual Exclusion Protocols



- Mutual exclusion of two related critical sections
- The solution space depends on the design of the
  - waiting function in case of a locked CS
  - enter_function ▨ & exit_function ▨

Implementation Levels

Implementation Levels[1]

User-level

- Relies neither on HW instructions nor on specific kernel features

HW-supported

- Relies on special HW instructions

Kernel-supported

- Low-level
- High-level

[1]Compare with signal objects

Applications of Solution Levels

- A multi-threaded application consisting of CSs in its p>1 threads **can** be solved with
  - Coordination-objects provided by a thread library (e.g. user-level monitor) in case of PULTs
  - Kernel-lock in case of KLTs
  - Specific HW instructions (portability problem?)
- An application consisting of CSs in different processes/tasks **must** be solved with
  - Kernel-locks or kernel-monitors or
  - Specific HW instructions (portability problem?)
  - Shared memory concept

Requirements for Valid Solutions

Note:

In many textbooks only three requirements are postulated

Four Necessary CS Requirements

- **Exclusiveness**
  - At most one activity is in the related CS
- **Portability**
  - Make no assumptions about
    - speed
    - number of CPUs
    - scheduling policy
    - ...

  KIT specific
- **Progress**
  - No activity running outside the related CS prevents another activity from entering its related CS
- **Bounded Waiting** (no starvation)
  - Waiting time of an activity in front of a CS must be limited

Analysis: Necessary Requirements

- Exclusiveness
  - If not fulfilled the approach is incorrect
- Portability
  - Some approaches heavily depend on whether we have a
    - single- or a multi-processor
    - time-slice based preemptive scheduling
- Progress
  - Often violated by too simple approaches
- Bounded Waiting
  - Often violated by busy waiting and static priority scheduling

Desirable Properties of CS

- **Performance**[1]
  - Overhead of entering and exiting a CS is small with respect to the execution time of the CS
  - Nevertheless, avoid busy waiting whenever it's possible
- **Adaptability**
  - Depending on the current load
- **Scalability**
  - Does it still work well with t>>1 threads
- **Simplicity**[2]
  - Should be easy to use

[1]One of the major goals in our research group

[2]very hard to decide ("do not use a sledgehammer to kill a fly"), however, that's exactly what we expect from you in the future

User-Level Solutions

Study these approaches
and “solutions” very carefully

There are many published approaches
that do not fulfill all KIT criteria

User-Level Approaches/Solutions

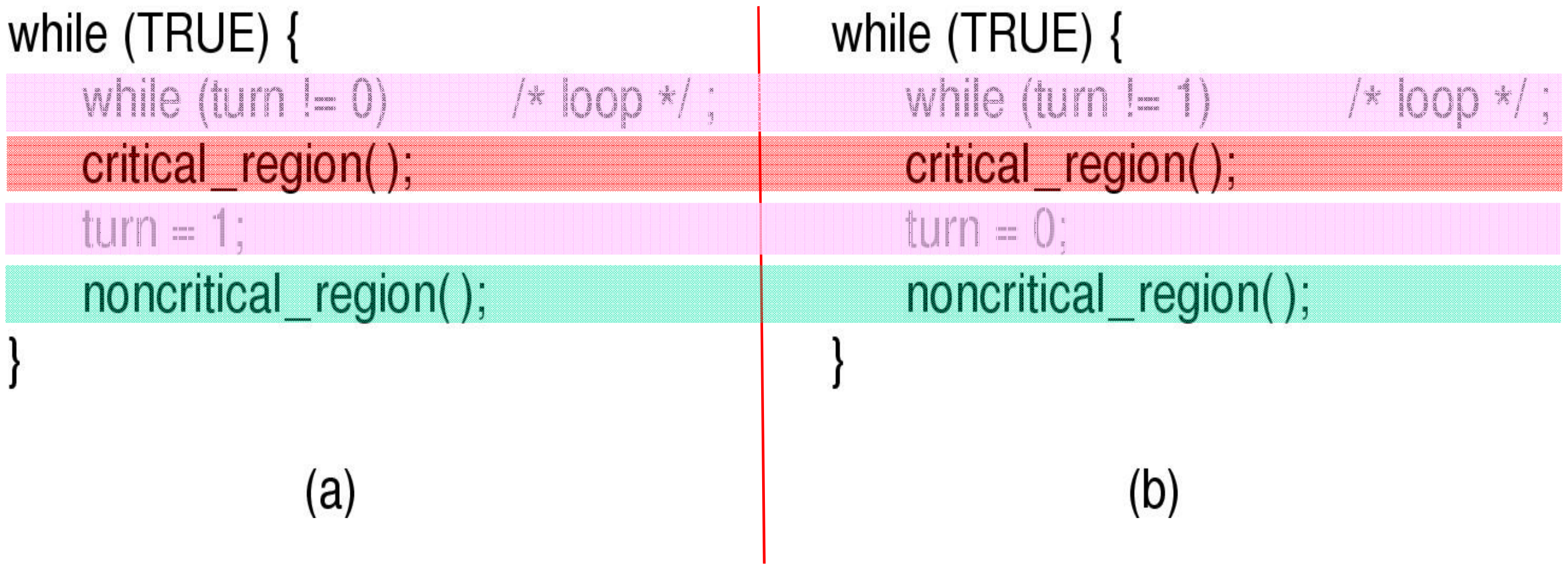
- <u>Simplification:</u>
  We first only discuss problems with either two processes or with two KLTs of the same task or with two PULTs of the same task
- Synchronization is done via global variables
- User-level “solutions” work with busy-waiting
  - Waiting for an event to occur, by polling a condition variable, e.g.

```
while(condition!=true); //just spin
```

  - Busy-waiting consume CPU-time, it is pure overhead, thus in many cases it is inefficient

Approach 1

Use global variable **turn** to denote who can enter CS



```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

Proposed solution to critical section problem
(a) Thread 0. (b) Thread 1

Analysis of Approach 1

Shared variable turn initialized.
T_i's CS executed iff turn = i
T_i is busy waiting if T_j is in CS $\Rightarrow$ mutual exclusion satisfied.
Progress not satisfied since strict alternation of both CSs is presumed

```
turn = 0; /*shared*/
thread Ti:
repeat while(turn≠i){};
        CSi
    turn=j;
        RSi
forever
```

Analysis:

Suppose: long RS_0, short RS_1. If turn=0, T_0 may enter CS_0, leaves it (turn=1), then is executing its very long RS_0. Meanwhile T_1 in CS_1, leaves it (turn=0), executes its short RS1. If T_1 tries to enter CS_1 soon after again, it must wait until T_0 leaves its long RS_0 and its following CS_0

Template for future Analysis

Requirement	Valid	Reason
Mutual Exclusion	yes	Due to turn either thread is in ist CS
No. or speed of CPUs	yes	Threads are alternating in their CS
Scheduling policy	NO	On a single processor and static priority scheme none of the threads might enter their CS, lifelock
Progress	NO	A long RS prohibits, that the other thread can enter twice in a row
Bounded Waiting	yes	In case of a fitting scheduling policy, no if one thread terminates earlier
Performance	NO	Busy waiting induces CPU overhead
Scalability	NO	The disadvantages increase with more threads

Approach 2[1]

- What was the major problem of the first approach?
  - Yes, it is very simple, thus it is robust, but the TID of the PULTs/KLTs have been stored in the synchronization variable, to decide who is allowed to enter and who has to wait

- Next idea:
  - Every PULT/KLT has its own key for its critical section, thus we can achieve, that in case one thread terminates, the other one is still capable to enter its critical section CS
  - No every of the two PULTs/KLTs can compete independently from the other

[1]the Chinese politness protocol

Approach 2

```
enum boolean {false =0, true =1};
boolean flag[2]={false, false};
//indicating: no thread is initially in ist CS
```

```
Thread T0:                        Thread T1:
while(true){                      while(true){
   while(flag[1])//waiting;          while(flag[0]);
    flag[0]=true;                     flag[1]=true;
    CS;                               CS;
    flag[0]=false;                    flag[1]=false;
   RS;                               RS;
}                                 }
```

Analysis of Approach 2

Requirement	Valid	Reason
Mutual Exclusion		
No. or speed of CPUs		
Scheduling policy		
Progress		
Bounded Waiting		
Performance		
Scalability		

Analyze carefully and compare your analysis with the one in your textbook

Approach3

Keep boolean variables for each KLT: `flag[0]` and `flag[1]`

T_i signals that it is ready to enter CS by setting: `flag[i]=true`

Mutual exclusion is satisfied but not the progress requirement

```
flag[2]={false,false};
thread Ti:
repeat
   flag[i]=true;
while(flag[j]){};
       CS
   flag[i]=false;
       RS
forever
```

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

What will happen?

Result: Both threads will wait forever, neither will ever enter its CS $\Rightarrow$ *classical deadlock*

Analysis of Approach 3

Requirement	Valid	Reason
Mutual Exclusion		
No. or speed of CPUs		
Scheduling policy		
Progress		
Bounded Waiting		
Performance		
Scalability		

Analyze carefully and compare your analysis with the one in your textbook

Approach 4

- One problem of approach 3 is, that the PULTs/KLTs set their state related to their CS without bothering the state of the other thread
- If both threads insist of being allowd to enter their CS, the result is a deadlock
- Idea:

  Every thread sets its flag, indicating, that it wants to enter its CS, but it is willing to rest this flag, in order to give the other thread to enter first

Approach 4

```
enum boolean {false=0, true=1};
boolean flag[2]={false, false}; //no one in CS
Thread T0:                          thread T1:
while(true){                        while(true){
    flag[0]=true;                       flag[1]=true;
    while(flag[1]){                     while(flag[0]{
     flag[0]=false;                         flag[1]=false
     //some delay                           //some delay
     flag[0]=true;                          flag[1]=true;
    }                                    }
    CS;                                  CS;
    flag[0]=false;                       flag[1]=false;
   RS;                                   RS;
};                                     };
```

Analysis of Approach 4

Requirement	Valid	Reason
Mutual Exclusion		
No. or speed of CPUs		
Scheduling policy		
Progress		
Bounded Waiting		
Performance		
Scalability		

Analyze carefully and compare your analysis with the one in your textbook

Approach 5: Dekker-Algorithm[1]

```
enum boolean {false=0, true=1};
boolean flag[2]={false, false}; //no one in CS
int turn = 1;  // signals what thread is to be preferred
Thread T0:                 Thread T1:
while(true){               while(true){
   flag[0]=true;                     flag[1]=true;
   while(flag[1]){            while(flag[0]{
     if (turn==1){                if [turn==0){
    flag[0]=false;                flag[1]=false;
    while(turn==1);               while(turn==0);
    flag[0]=true;                 flag[1]=true;
   }}                             }}
   CS;                         CS;
   turn=1;                     turn=0;
   flag[0]=false;              flag[1]=false;
  RS;                          RS;
};                         };
```

[1]Published by Dutch mathematician T. Dekker 1965

Analysis of Dekker Algorithm

Requirement	Valid	Reason
Mutual Exclusion		
No. or speed of CPUs		
Scheduling policy		
Progress		
Bounded Waiting		
Performance		
Scalability		

Analyze carefully and compare your analysis with the one in your textbook

Approach 6: Peterson Algorithm

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

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

If both threads attempt to enter their CS simultaneously, the turn value decides which one will win

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

*Stallings' notation slightly different from Tanenbaum's template

“Proof” of Algorithm 3

- To prove that mutual exclusion is preserved:
  - T0 and T1 are both in their CS only if flag[0] = flag[1] = true
    and only if turn = i for each Ti (which is impossible by definition)
- To prove that the progress and bounded waiting requirements are satisfied:
  - Ti prevented from entering CS only if stuck in ‘*while()..*’ with condition
  - ‘*flag[j]* ’
    and ‘*turn = j* ’.
  - If Tj is not ready to enter CS then *‘ ! flag[j] ’* and Ti can enter its CS
  - If Tj has set *‘flag[j]’* and is in its *‘while()..’*, then either *turn=i* or *turn=j*
  - If *turn=i*, then Ti enters CS. If *turn=j* then Tj enters CS, but it will
    reset *flag[j]* on exit: allowing Ti to enter CS
  - but if Tj has time to set *flag[j]*, it must also set *turn=I*
  - since Ti does not change value of *turn* while stuck in ‘*while()..*’,
    Ti will enter CS after at most one CS entry by Tj (bounded waiting)

Analysis of Peterson’s Algorithm

Requirement	Valid	Reason
Mutual Exclusion		
No. or speed of CPUs		
Scheduling policy		
Progress		
Bounded Waiting		
Performance		
Scalability		

Analyze carefully and compare your analysis with the one in your textbook

Problems with Faulty Threads

If all four necessary criteria (mutual exclusion, progress, portability, bounded waiting) are satisfied, a valid solution will provide robustness against bugs in the RS_i of a KLT. Bugs within RS do not affect the other KLTs.

However, no valid solution can ever provide robustness, if a KLT fails within its critical section

A KLT failing within its CS might never perform exit_section , i.e. no other KLT related to that CS can ever perform enter_section successfully!

Bakery Algorithm for n Threads

Before entering their CS, each T_i receives a number. The holder of the smallest number enters its CS

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 $b < d$
    $\max(a_0,\dots a_k)$ 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
```

Summary: User-Level Approaches

- Activities trying to enter a locked critical section are busy waiting ($\Rightarrow$ wasting processor time)
- If the critical section CS has a long execution phase it is more efficient to block the activity
- On a single processor with static priority scheduling, busy waiting can always lead to starvation, i.e. to a life lock

HW-Support for CS

Interrupt Locking

Test And Set Instruction

How to implement Atomic Operations?

We need additional HW support:

- Disabling interrupts
  - *Why can this prevent a thread switch?*
  - *For all systems?*
- Atomic instructions
  - CPU and bus guarantee entire action will execute atomically
    - Test And Set (TAS instruction)
    - Compare And Swap
    - ...

Interrupt Locking

- It is a quite primitive mechanism, only valid for single processor systems
  - Portability requirement not fulfilled
  - Disabling interrupts for CPU0 does not prevent that on another CPU a conflicting CS is exectuted
  - However, for specific “very short CS” inside the kernel this approach can be a solution for ingle-processors
- Before an activity A_i enters its CS, this A_i disables all interrupts
  - Especially the time-slice interrupt, thus there is no possibility that a thread switch might be induced
  - Structure of mutual-exclusion protocol

```
Activity Ai:
while (true) {
  disable interrupt;
   CS;
  enable interrupt;
   RS;
};
```

Disabling Interrupts at User Level?

Single processor:

Mutual exclusion preserved, but efficiency degraded: while in CS, no interrupt handling anymore

- No *time-slicing* anymore
- *Delay* of interrupt handling may affect the whole system
- Application programmers may abuse → system hangs

```
activity Ai:
while (true) {
  disable interrupt
   CS;
  enable interrupt;
   RS;
};
```

Why?

Multi processors:

Not effective at all

Summary: Approach is unacceptable due to its side effects

Good news: `disable_interrupts` is privileged on CPUs, i.e. it can not be used at application level at all

Interrupt Locking

- Problems with interrupt locking
  - CS must be very short,
    - Interrupts can not be delayed too long, otherwise interrupt signals might be lost
  - Suppose inside the CS, the KLT aborts, then the rest of the system is in a life lock, no interrupt can be handled anymore
  - Mechanism can be used only on single processors
- Summary:
  - Mechanism is not suitable for mutual exclusion and conditional synchronization at application level
  - However, it might be useful inside the kernel

Disabling Interrupts at Kernel Level

- Could help us in implementing atomic Spin Lock operations `Acquire_lock()` & `Release_lock()`

```
struct lock{
  int held = 0;
  }
void Acquire_lock(lock) {
  Disable_Interrupts;
  while (lock->held);
  lock->held=1;
  Enable_Interrupts;
  }
void Release_lock(lock){
  Disable_Interrupts;
  lock->held=0
  Enable_Interrupts;
  }
```

Discuss this approach carefully

- *Do we still have a race condition?*
- *Would you use this approach when you have to solve a critical section problem?*
- *What are the major disadvantages?*

Still Busy Waiting with Side Effects on System

Severe Bug on Previous Slide

```
struct lock{
  int held = 0;
  }
void Acquire_lock(lock) {
  Disable_Interrupts;
  while (lock->held);
  lock->held=1;
  Enable_Interrupts;
  }
void Release_lock(lock){
  Disable_Interrupts;
  lock->held=0
  Enable_Interrupts;
  }
```

Improved Simple Spin Lock?

```
struct lock{
  int held = 0;
  }
void acquire(lock) {
  Disable_Interrupts;
  while (lock->held)    {
       Enable_Interrupts;
                         ——————————————————∃ possibility for a thread switch
       Disable_Interrupts;
   } //systemno longer spinning with pending interrupts
  lock->held=1;
  Enable_Interrupts;
  }
void set_free(lock){
  Disable_Interrupts;
  lock->held=0
  Enable_Interrupts;
  }
```

More HW Support for a Spin-Lock

- If we could test and set the synchronization variable in one atomic instruction, we might have solved the spin-lock problem
- Some processors offer this atomic `testandset` instruction
- Two possible implementations of the TAS instruction:

```
TAS1:
boolean test_and_set( boolean *flag){
  boolean old = *flag;
  *flag = true;
  return old;
}

TAS2:
boolean testset( int i){
  if (i==0){
      i=1; return true
  }
  else return false
}
```

Semantics:
Result = true, the spinlock could be set successfully
Result = false, the spinlock is held by someone else

TAS2 for Mutual Exclusion

```
const int t=100; // number of KLTs
int spinlock;
void T(inti){
 while(true){
   while (!testset(spinlock)); //busy waiting
   CS;
   spinlock = 0;
   RS;
   }
}
void main(){
  spinlock=0;// initially no one is in ist CS
  parbegin (T(1),T(2), …,T(n));
}
```

Spin Lock with TAS1

```
struct lock_SMP{
  int held = false; /* initialization */
  }
void Acquire_SMP(lock_SMP) {
  while (test_and_set(&lock_SMP->held));
  }
void Release_SMP(lock_SMP){
  lock_SMP->held=false
  }
```

Do we still have a race condition at the variable lock_SMP_held?
Is this solution portable and efficient?
What happens when many KLTs try to acquire the same Spin Lock?

Approach with Lock_SMP

```
static lock_SMP Spin = false;

thread Ti {
repeat
  Acquire_SMP(Spin);
     CS
  Release_SMP(Spin);
     RS
forever
}
```

Analysis of TAS1 or TAS2

- Advantages:
  - Number of involved threads is not limited
  - Quite simple approach and easy to understand
  - You can use it also to control many critical regions CRs, as long as you provide a different spinlock variable per CR
- Disadvantages:
  - Busy waiting can always be inefficient
  - A KLT can starve in front of its critical section in case it has only low priority
  - Deadlocks and priority inversion can happen with nested CS

Deeper Analysis of Lock_SMP (1)

- Mutual exclusion is preserved
- However, if one T_i is in its CS, all other T_j –trying to enter their CS- perform busy waiting

$\Rightarrow$ a potential efficiency problem

When T_i exits CS, selection of T_j that will enter its CS is arbitrary $\Rightarrow$ no bounded waiting guaranteed

possible starvation of a T

However, what is the main problem with any of these "spin lock" solutions?

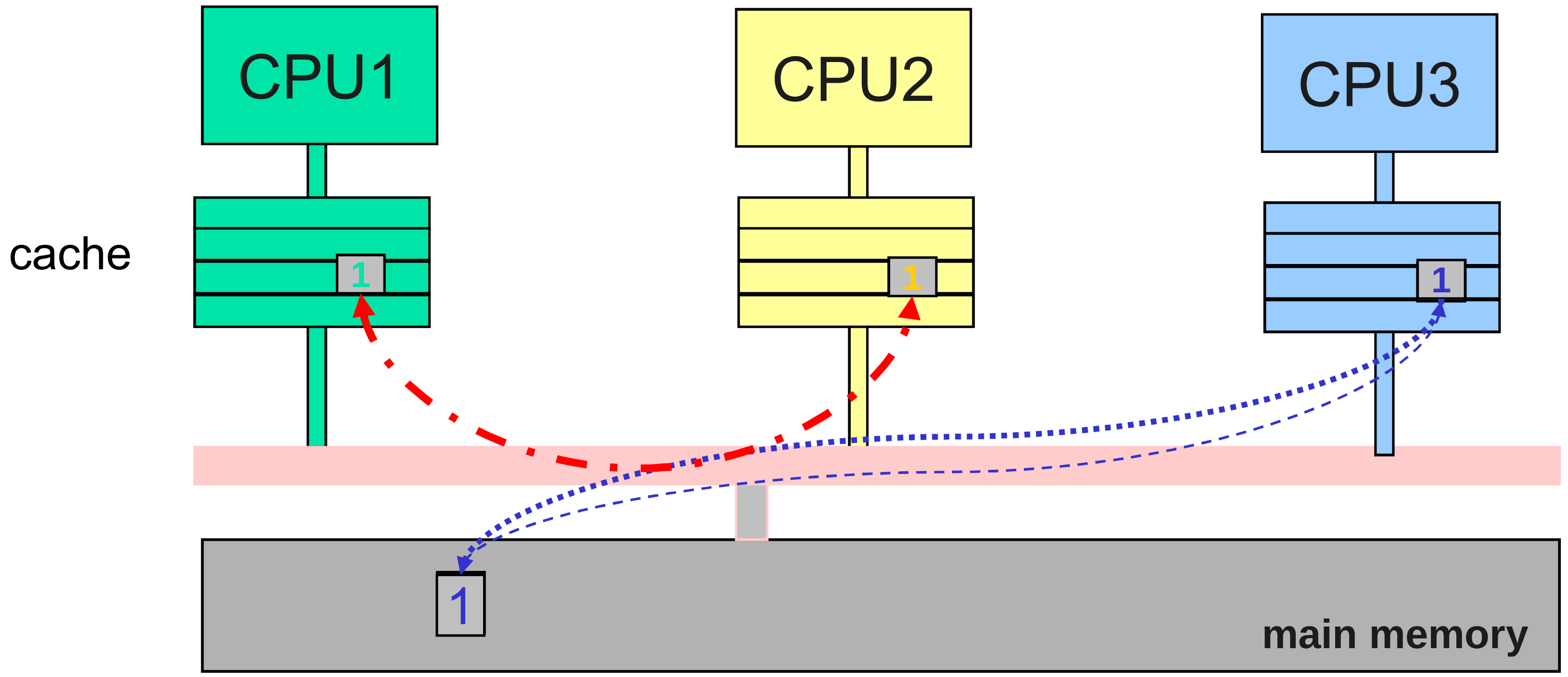
Analysis of LOCK_SMP (2)

Repeated test-and-set-instructions can monopolize the system bus affecting other activities (whether related to that critical section or whether not)

What about consequences concerning cache coherence?

Furthermore there is a severe danger of another sort of starvation on a single processor system (compare with busy waiting at application level)

Spin-Lock Problems in SMPs



<u>Result:</u> "Ping-pong" effect between cache(CPU1) & cache(CPU2) wasting system-bus capacity

<u>Corollary:</u> Design & implement a better spin lock

Other Atomic CPU Instructions

(1) Some machines offer instructions that perform read-modify-write operations atomically (indivisible, 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) allow emulating mutually exclusive instructions

Swap Instruction

```
void swap (boolean *a, boolean *b)
{
   boolean temp = *a;
   *a = *b;
   *b = temp:
          }
```

Can you use this Swap instruction to enable a suitable & portable CS protocol? (see assignments)

Kernel-Level Solutions

Low-Level

High-Level

Mutex at Kernel-Level

- Instead of implementing **`Acquire_lock()`** with busy waiting we can use our Kernel API, i.e. **`BLOCK()`**, **`UNBLOCK()`**
- If lock is currently held by another thread, the thread having called **`Acquire_mutex()`** will be blocked
  - Put the thread to sleep until it can acquire the lock
  - Free the CPU for other KLTs to run
- However, we must also change **`Release_mutex()`**, because in the meantime some thread could have been blocked waiting for the mutex
- Each mutex has an associated wait queue (similar to a semaphore)
- Design and implement a kernel object mutex with atomic **`Acquire_mutex()`** and **`Release_mutex()`** operations at least for a single-processor system

First Approach: A Simple Lock*

- A lock is an object (in main memory) providing the following two operations:
  - `Acquire_lock():` before entering a CS
    - If lock is held, KLT must wait in front of CS
  - `Release_lock():` after leaving a CS
    - Allows another KLT to enter the CS

First Approach: Simple Lock

- After an `Acquire_lock()` there must follow a `Release_lock()`
  - Between `Acquire_lock()` & `Release_lock()`, a KLT is holding the lock (=*current lock holder)*
  - `Acquire()_lock` with blocking waiting only returns when the caller is the current lock holder

1. *What might happen if* ***`Acquire_lock()`*** *and* ***`Release_lock()`*** *calls are not paired?*
2. *What happens when the current lock holder tries to acquire the same lock once more?*

Using the Simple Lock

```
int withdraw(account, amount){
  Acquire_lock(lock1);
  balance = get_balance(account);   }
  balance -= amount;                } CS
  put_balance(account, balance);    }
  Release_lock(lock1);
  return balance;
}
```

Execution with Simple Lock

```
Acquire_lock(lock1);
balance = get_balance(account);
balance -= amount;
```

Thread 1 runs

```
Acquire_lock(lock1);
```

?

Thread 2 runs but must wait on lock

```
put_balance(account, balance);
Release_lock(lock1);
```

Thread 1 runs & completes

```
balance = get_balance(account);
balance -= amount;
put_balance(account, balance);
Release_lock(lock1);
```

Thread 2 resumes and completes

What happens when thread 2 tries to acquire the lock?

1. Approach: Simple Lock

```
struct lock{
    int held = 0;
}
void Acquire_lock(lock){
    while (lock->held);
    lock->held=1;
}
void Release_lock(lock){
    lock->held=0
}
```

bug

Initialization:
lock is free

Spinning

Caller is busy waiting
till lock is released

Most system architects call it a “Spin Lock”

Observation: There is a severe bug *Where?*

Implementing a Spin Lock

Problem:

Internals of both operations have critical sections

- `Acquire_lock()` and `Release_lock()` must be atomic
- The all or nothing principle (see: transactions)

Now, we face a really hard *dilemma*[*]:

We've introduced spin locks to provide a mutual exclusive protocol to solve critical section problems, but our solution contains yet another critical section problem

What to do? Who can help us poor system architects?

Help comes from the processor architect
He helps us to end the recursion

[*] *Baron Münchhausen Syndrome*

Adaptive Lock

- Waiting in front of an adaptive lock is done
  - either via spinning
  - or via blocking
- Criteria for spinning
  - When lock holder is currently running
  - When trying to acquire the lock a timer is started that blocks the caller after n time units
- Criteria for blocking
  - When lock holder is ready or waiting

Recursive Lock

- *How to enhance a simple lock to be able to note that the current lock holder wants to acquire the lock again?*
- *What to do in the release function?*

Counting Semaphore

Counting Semaphore for CS

1. Positive value of counter→ #threads that can enter the “CS” concurrently
   - If mutual exclusion is required, initialize the semaphore counter with 1
2. Negative value of counter → #waiting threads in front of CS, i.e. queued at semaphore object
3. Counter == 0 → no thread is waiting and maximal #threads are currently in CS

<u>Still an open problem:</u>

How to establish atomic semaphore operations?

Implement Counting Semaphores

```
module semaphore {
 export p, v
 import BLOCK, UNBLOCK
 type semaphore = record{
     Count: integer = 1             {CS not yet locked}
     QWT: list of Threads = empty {no waiting threads}
   }
 p(S:semaphore){
     S.Count = S.Count - 1
     if (S.Count < 0){
      insert (S.QWT, myself)        {+ 1 waiting thread}
      BLOCK(myself)
     }}
 v(S:semaphore){
     S.Count = S.Count + 1          {unlock CS }
     if (S.Count ≤ 0) {
      UNBLOCK(take_any_of S.QWT)    {weak semaphore}
     }}
}
```

Atomic Semaphore Operations

Problem:
p() and v() -each consisting of multiple machine instructions- have to be atomic!

Solution:
Use "another" type of critical sections, hopefully with shorter execution times, establishing atomic and exclusive semaphore operations

"very short" enter_section

p(S)

"very short" exit_section

Revisiting Dijkstra’s Semaphores

Short CS implementing atomic, exclusive p(S) and v(S)

Possible solutions for short CS around p(S), v(S):

Single processor:

- Disable interrupts as long as p() or v() *running*
- *Contradiction to our recommendation not to manipulate interrupts at application level?*

Multi processor:

- Use special instructions (e.g. TAS)

Single Processor Solution

```
p(sema S)
 begin
 DisableInterupt
   s.count--
   if (s.count < 0){
     insert_T(s.QWT)
     BLOCK'(S)
   }
   else
 EnableInterrupt
 end
```

```
v(sema S)
 begin
 DisableInterrupt
   s.count++
   if (s.count ≤ 0){
     remove_T(s.QWT)
     UNBLOCK'(S)
   }
 EnableInterrupt
 end
```

What happens, if switching to another thread? Interrupts still disabled?

Multiprocessor Solution

```
P(sema S)
begin
 while (TAS(S.flag)==1){};
    { busy waiting }
    S.Count= S.Count-1
    if (S.Count < 0){
        insert_T(S.QWT)
        BLOCK(S)
        {inkl.S.flag=0)!!!}
    }
    else S.flag =0
end
```

```
V(semaS)
begin
 while (TAS(S.flag)==1){};
    { busy waiting }
    S.Count= S.Count+1
    if S.Count ≤ 0 {
        remove_T(S.QWT)
        UNBLOCK(S)
    }
 S.flag =0
end
```

Weak Counting Semaphores

```
p(S:semaphore)
   S.Count = S.Count - 1;
   if S.Count < 0 {
    insert(S.QWT, myself);  {i.e. somewhere}
    BLOCK(myself);
    }
   fi.

v(S:semaphore)
   S.Count = S.Count + 1;
   if (S.Count ≤ 0) {
    thread = take_any_of S.QWT; {no order}
    UNBLOCK (thread);
    }
   fi
```

Strong[1] Counting Semaphores

Preserves FCFS

```
p(S:semaphore)
   S.Count = S.Count - 1
   if S.Count < 0 {
    append (S.QWT, myself)
    BLOCK(myself)
    }
   fi.

v(S:semaphore)
   S.Count = S.Count + 1         {unlock CS }
   if (S.Count ≤ 0) {
    thread = take_first_of S.QWT;
    UNBLOCK (thread)
}
   fi
```

[1]Strict

Application of Counting Semaphores

Suppose: *n* concurrent threads

Initialize S.Count to 1 ⇒ only 1 thread allowed to enter its CS (i.e. *mutual exclusion*)

Initialize S.Count to k>1 ⇒ k>1 threads allowed to enter their CS

When to use this semantics?

```
thread Ti:
repeat
    p(S);
     CS
    v(S);
     RS
forever
```

Producer/Consumer

A semaphore S to perform mutual exclusion on the buffer:
Only one thread at a time should access the buffer

A semaphore N to synchronize producer and consumer on the number N (= in - out) of items in the buffer:
An item can be consumed only after it has been created

Producer is free to add an item into the buffer at any time, but it has to do P(S) before appending and V(S) afterwards to prevent concurrent accesses by the consumer

It also performs V(N) after each append to increment N
Consumer must first do P(N) to see if there is an item to consume, then it uses P(S) and V(S) while accessing the buffer

Producer/Consumer (∞ Buffer)

```
Initialization:    Auxiliary functions:
  S.count:=1;      append(v){                 take(){
  N.count:=0;      b[in]:=v;                  w:=b[out];
  in:=out:=0;      in++;}                     out++;
                                              return w;}

          Producer:      Consumer:
          repeat         repeat
            produce v;     p(N);
            p(S);          p(S);
            append(v);     w:=take();
            v(S);          v(S);
            v(N);          consume(w);
          forever        forever
```

*Q: Semaphore Solutions**

- *Why do we need mutual exclusion at the buffer?*
- *Why does the producer* `v(N)` *?*
- *Why is the order of the* `p()` *in the consumer important?*
- *Is order of the* `v()` *in the producer important?*
- *Is this solution extensible to p>1 producers and/or c>1 consumers?*

*Be prepared for similar questions in assignments and exams

Summary on Semaphores

Semaphores provide a primitive coordination tool

- for enforcing mutual exclusion **and/or**
- for synchronizing threads

p(S) and v(S) are scattered among several threads. Hence, it's difficult to understand all their effects

Usage must be correct in all threads

One buggy (malicious) thread can crash an entire application or sub system

Recommendation

Avoid using Semaphores*

personal recommendation,
Jochen Liedtke

What to use instead of?

∃ better synchronizations tools?

“Software” Monitors

*<u>Remark:</u> Java language offers monitors
You should be familiar with them

Monitor (1)

- High-level “language construct”
- ~ semantics of binary semaphore, but easier to control
- Offered in some programming languages
  - Concurrent Pascal
  - Modula-3
  - Java
  - …
- Can be implemented using semaphores or other synchronization mechanisms

Monitor (2)

A software module[*] consisting of:

- one or more interface procedures
- an initialization sequence
- local data variables

Characteristics:

- local variables accessible only inside monitor methods
- thread enters monitor by invoking a monitor method
- only one thread can run inside a monitor at any time, i.e. a monitor can be used to implement mutual exclusion

[*]Java's synchronized classes enable monitor-objects

Monitor (3)

Monitor already ensures mutual exclusion $\Rightarrow$
no need to program this constraint explicitly

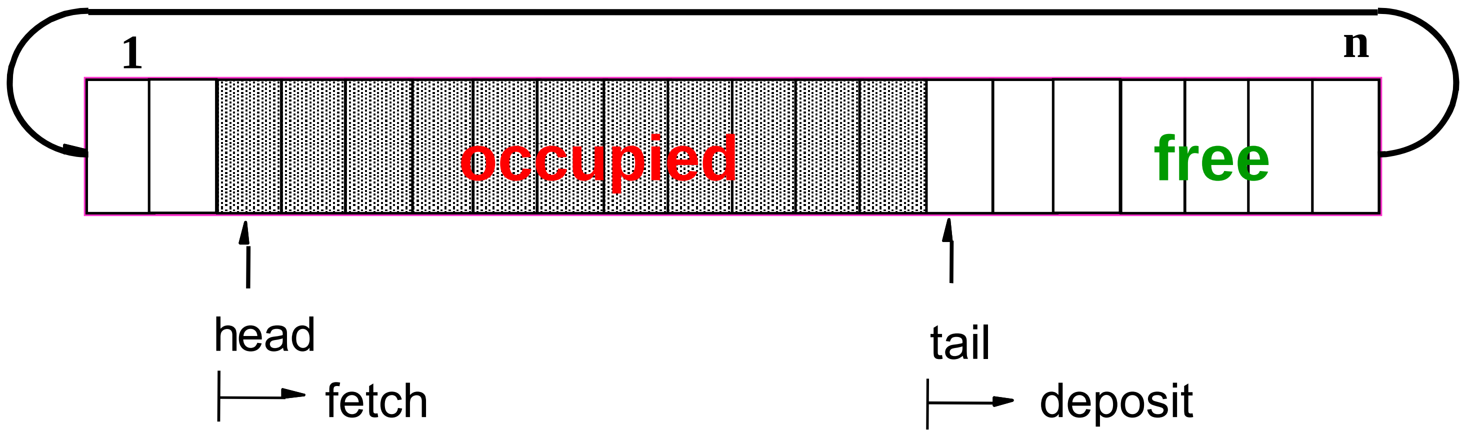
Hence, shared data are protected automatically
by placing them inside a monitor.
Monitor locks its data whenever a thread enters

Additional thread synchronization inside the monitor can be done by the programmer using condition variables

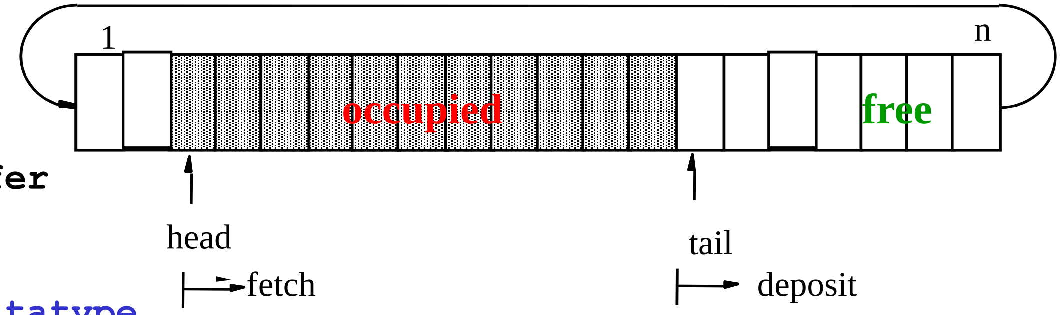
A condition variable represents a certain condition (e.g. an event) that has to be met before a thread may continue to execute one of the monitor procedures

Approach for a Monitor Solution*

Cyclic buffer of N slots with interface operations `fetch()` and `deposit()`

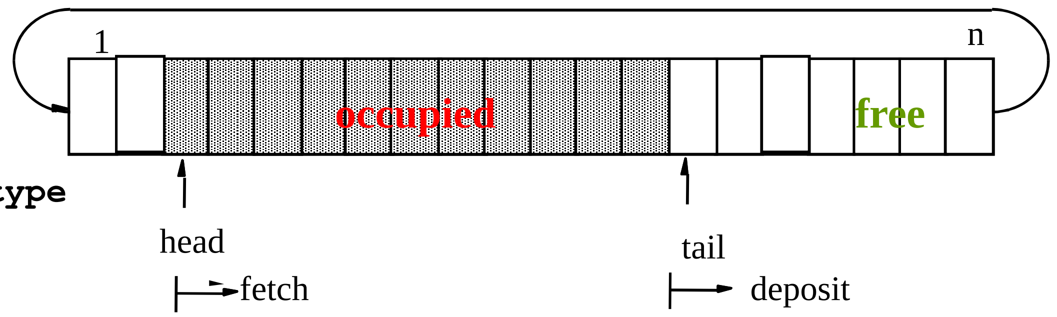


*Detailed example for the development of a solution *"step by step"*



```
monitor module bounded_buffer
 export fetch, deposit;
 buffer_object = record
   array buffer[1..N] of datatype
   head: integer = 1
   tail: integer = 1
   count: integer = 0
 end
 procedure deposit(b:buffer_object, d:datatype)
 begin
   b.buffer[b.tail] = d
   b.tail = b.tail mod N +1
   b.count = b.count + 1                Automatically with mutual exclusion
 end
 procedure fetch(b:buffer_object, result:datatype)
 begin
   result = b.buffer[b.head]
   b.head = b.head mod N +1
   b.count = b.count - 1                Automatically with mutual exclusion
 end
end monitor modul
```

Concurrent deposits or fetches are serialized, but you can still deposit to a full buffer and you can still try to fetch from an empty buffer!
$\Rightarrow$ two additional constraints have to be considered.



```
monitor module bounded_buffer
 export fetch, deposit;
 import BLOCK, UNBLOCK;
 buffer_object = record
   array buffer[1..n] of datatype
   head: integer = 1
   tail: integer = 1
   count: integer = 0
   SWT_D,SWT_F of threads = empty {2 waiting queues due to deposit,fetch}
 end
 procedure deposit(b:buffer_object, d:datatype)
 begin
   while (b.count == n) do BLOCK(b.SWT_D)   Also blocks the monitor
   b.buffer[b.tail] = d
   b.tail = b.tail mod n +1
   b.count = b.count + 1
   if (b.SWT_F ≠ empty) UNBLOCK(b.SWT_F)
 end
 procedure fetch(b:buffer_object, result:datatype)
 begin
   while (b.count == 0) do BLOCK(b.SWT_F)   Also blocks the monitor
   result = b.buffer[b.head]
   b.head = b.head mod n +1
   b.count = b.count - 1
   if (b.SWT_D ≠ empty) UNBLOCK(b.SWT_D)
 end
end monitor modul
```

No longer deposits to a full buffer or fetches from an empty buffer, but ...???

Condition Variables

Local to the monitor (accessible only inside the monitor) can be accessed only by:

`CondWait(cv)`	blocks execution of the calling thread on condition variable **cv**

This blocked thread can resume its execution only if another thread executes CondSignal(cv)

`CondSignal(cv)`	resumes execution of some thread blocked on this condition variable **cv**

If there are several such threads: choose any one

If no such thread exists: void, i.e. nothing to do

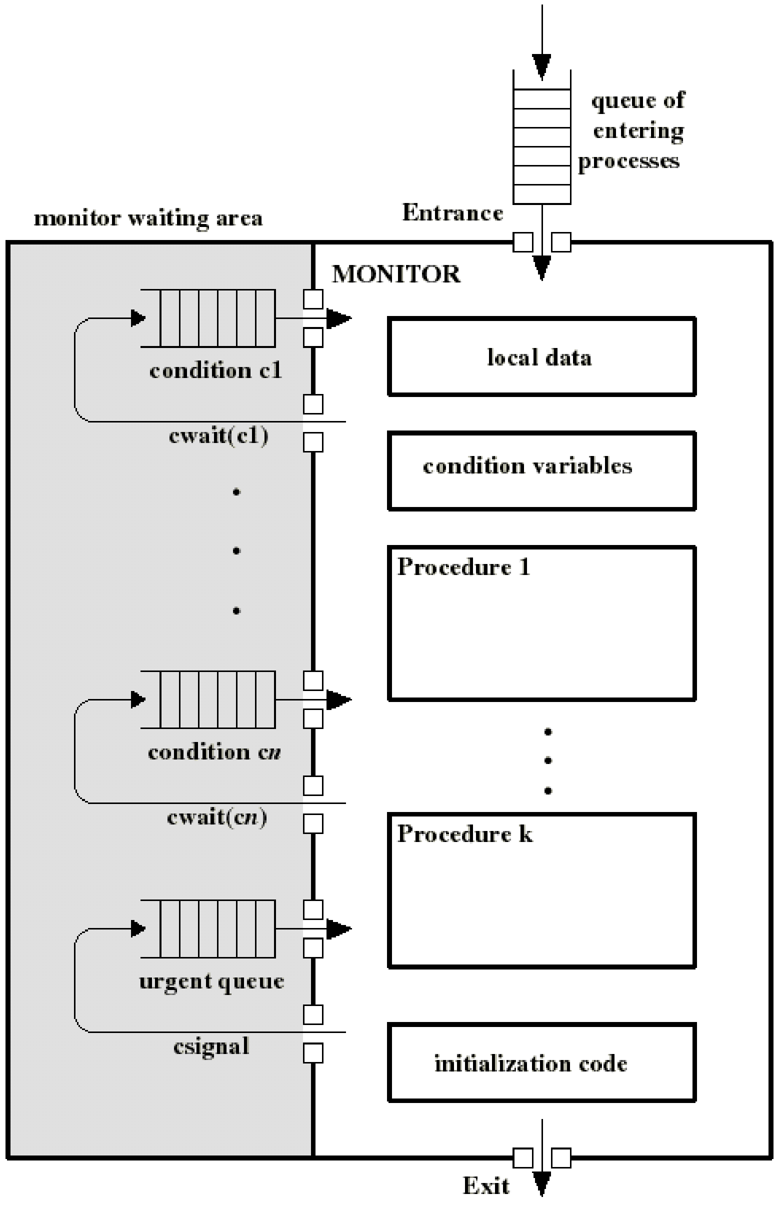
Monitor (4)

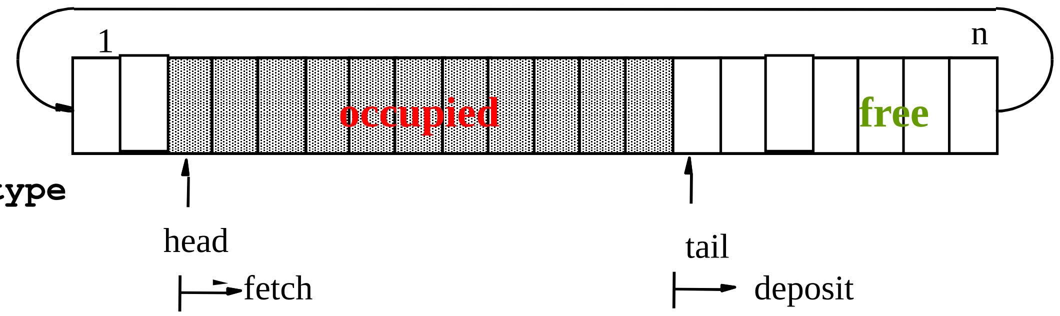
Waiting threads are either in the entrance queue or in a condition queue

A thread puts itself into the condition queue cn by invoking CondWait(cn)

CondSignal(cn) enables one thread, waiting at condition queue cn, to continue

Hence CondSignal(cn) blocks the calling thread and puts it into the urgent queue (unless Condsignal is the last operation of the monitor procedure)





```
monitor module bounded_buffer
 export fetch, deposit
 import CondSignal,CondWait
 buffer_object = record
   array buffer[1..n] of datatype
   head: integer = 1
   tail: integer = 1
   count: integer = 0
   not_full: cond = true
  not_empty: cond = false
 end
 procedure deposit(b:buffer_object, d:datatype)
 begin
   while (b.count == n) do CondWait(b.not_full) {only block thread}
   b.buffer[b.tail] = d
   b.tail = b.tail mod n +1
   b.count = b.count + 1
   CondSignal(b.not_empty)
 end
 procedure fetch(b:buffer_object, result:datatype)
 begin
   while (b.count == 0) do CondWait(b.not_empty) {only block thread}
   result = b.buffer[b.head]
   b.head = b.head mod n +1
   b.count = b.count - 1
   CondSignal(b.not_full)
 end
end monitor modul
```

Summary: Producer/Consumer

Two types of threads:

- Producer(s)
- Consumer(s)

Synchronization is now confined to the monitor

`deposit(...)` and `fetch(...)` are monitor interface methods

If these 2 methods are correct, synchronization will be correct for all participating threads.

```
ProducerI:
repeat
  produce v;
  deposit(v);
forever

ConsumerI:
repeat
  fetch(v);
  consume v;
forever
```

Reader/Writer with Monitor

Using monitors you can also solve reader/writer problems with either reader or writer preference,

Compare your solution with the one using semaphores in one of the text books

Remarks and Open Questions

- Which of the 2 threads T and T' should continue when T executes CondSignal(cv) while T' was waiting due to a previous CondWait(cv)?
- A monitor must stay closed if some externally initiated event occurs, e.g. end of time slice (Otherwise no mutual exclusion anymore)
- However, what to do when a monitor method of monitor M invokes a method of monitor M'?