4 Concurrency

Concurrency Problems

Signals & Synchronization

Semaphore

Mutual Exclusion

Critical Section

Monitors

Roadmap for Today

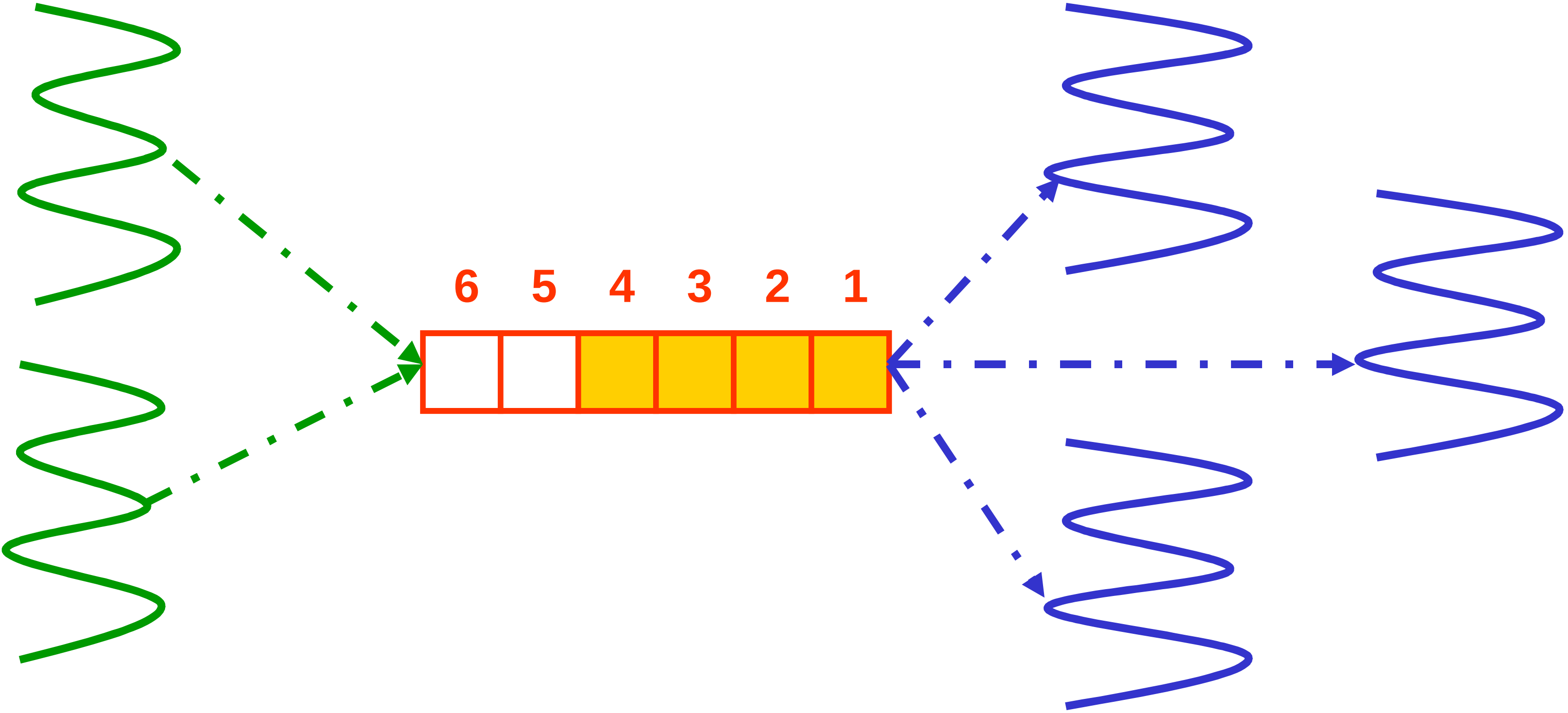
- Concurrency Problems
  - Producer / Consumer and
  - Reader / Writer
- Synchronization Mechanisms
  - Signal
  - Semaphore
  - Monitor

Concurrency Problems

Producer Consumer Problem

Reader Writer Problem

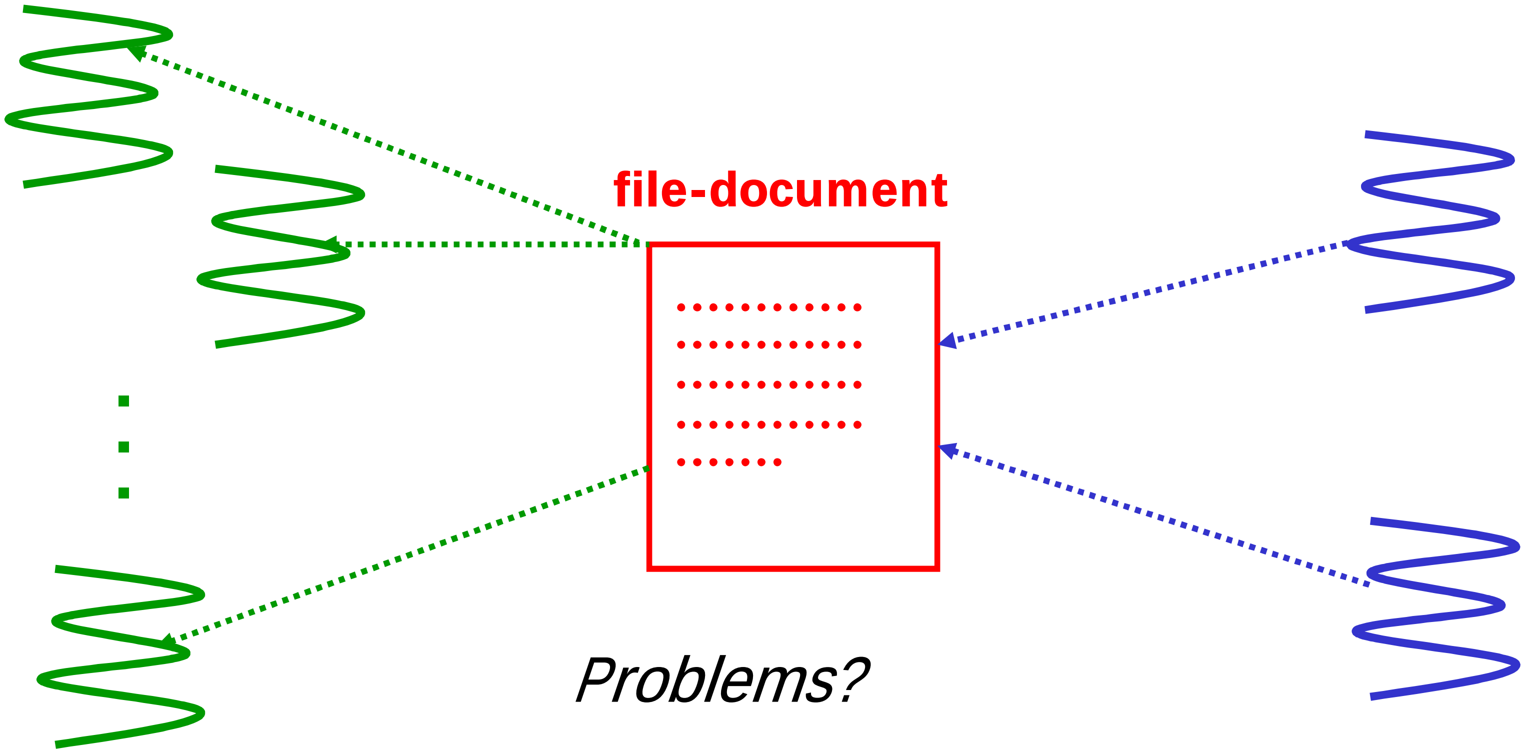
Producer/Consumer & Bounded Buffer



Concurrency problems with bounded buffers?

Problems with p>1 producers or c>1 consumers?

Reader/Writer Problem



Problems?

Possible solutions?

Assignment 2

- All concurrency problems have to be solved using Java monitors or specific self-made semaphores implemented by Java monitors
  - Study how to use those Java monitors (some hints are given in the assignments)
  - We do not accept solutions where one centralized thread is used to do the sequencing job, i.e. somewhere in your code there must be properly positioned assignments with **`wait()`** and **`notify()`**
- Do a nice graphic to visualize your solutions of the experiments

Signal Mechanism

History of Signals

Application of signals

Don’t mix up with Unix signals

Semantics of Signals

- “Pay Attention” (see a siren)
- “Stop” (see road signs)
- “Go Ahead” (officer at a train station)
- “**Interrupt**” or (arbiter in a soccer game)
  “**Resume Playing**”
- ...

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

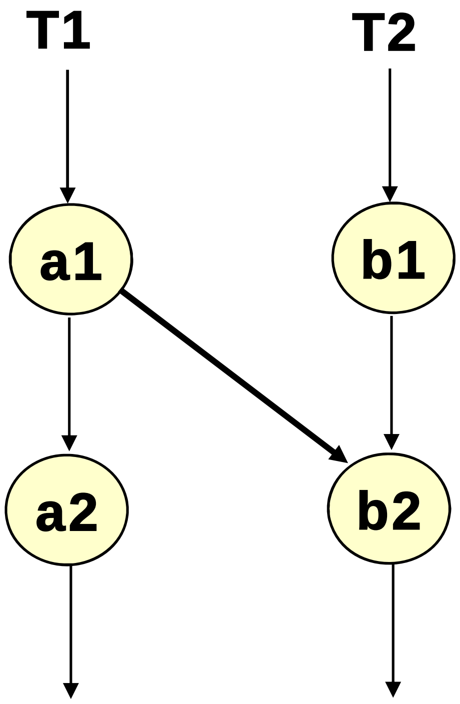
- Counter Any value may have a different meaning or just reflects the number of pending signals

Problem: Try to find out when a flag is sufficient
or when you better use a counter variable!

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 has to wait until section a1 has completed*

Synchronize a Precedence Relation

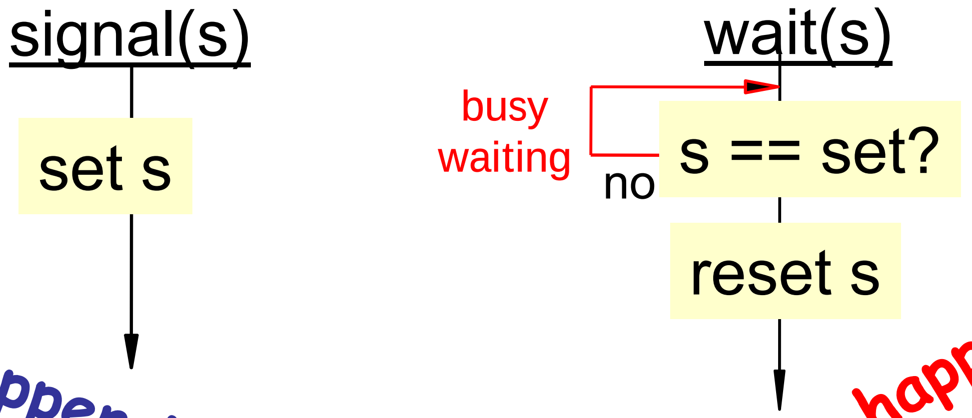
```
1:1_signal s;    /* type 1:1_signal_object */

 {Thread 1}                         {Thread 2}
 .                                  .
 .                                  .
 .                                  .
 { section a1                       { section b1
   … }                                … }
 Signal(s) ·············►           Wait(s)
 { section a2                       { section b2
   … }                                … }
 .                                  .
 .                                  .
 .                                  .
```

<u>Problem:</u> *How to implement a 1:1_signal_object?*

User-Level Signal Object: FLAG

Simple **`flag s`** as a common shared global variable of both threads



signal(s)

set s

wait(s)

busy waiting

no

s == set?

reset s

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

What can happen if 'wait' is invoked prior to 'signal'

Hint: Discuss this approach carefully! *Does it work on every system effectively and/or efficiently?*

Principal Types of Solutions

- Software Solutions (at application level)
  - Algorithms neither rely on special processor hardware nor on special OS features
- Hardware Solutions
  - Rely on some special machine instructions
  - Offering a kind of atomicity
- OS Solutions (offered by kernel)
  - Provide “kernel-interface functions” for application programmers

Remark: Most systems offer only a subset of these solutions.

Semaphores

Dijkstras (Counting) Semaphores

Definition:

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 ~ **P**asseren (from Dutch signaling language
some say **p**roberen ~ decrement)

V(S) V ~ **V**erlaaten (see above,
some say **v**erhogen ~ increment)

Dijkstras (Counting) Semaphores

How to design and implement counting semaphores?

- To avoid *busy waiting*:
- When thread cannot “passeren” inside of P(S) ⇒ put calling thread into a blocked queue waiting for an event
- Occurrence of event will be signaled via V(S) by another thread (hopefully)
    - *What happens if not?*

Dijkstras Semaphores

Semantic of a counting semaphore (for **signaling**):

- A positive value of counter indicates:
  #signals currently pending
- A negative value of the counter indicates:
  #threads waiting for a signal,
  i.e. are queued within the semaphore object
- If counter == 0 ⇒ no thread is waiting
  and no signal is pending

Remark (from Margo Seltzer, Harvard USA):

> “A semaphore offers a simple and elegant
> mechanism for mutual exclusion and other things”

Counting Semaphores (First solution)

```
module semaphore
 export p, v
 import BLOCK, UNBLOCK
 type semaphore = record
     Count: integer = 0              {no signal pending}
     QWT: list of Threads = empty  {no waiting threads}
   end
 p(S:semaphore)
     S.Count = S.Count - 1
     if S.Count < 0 then
      insert (S.QWT, myself)        {+ 1 waiting thread}
      sleep(myself)
     fi
 v(S:semaphore)
     S.Count = S.Count + 1           {+ 1 pending signal}
     if S.Count <= 0 then
      wakeup(delete first(S.QWT))
     fi
end
```

Unix Signals

- Besides a terrible notation (e.g. kill = signal) ∃ no common semantics nor a widely accepted interface
- They are four different versions:
  - System-V unreliable
  - BSD
  - System-V reliable
  - POSIX
- Using Unix signals may lead to severe race conditions
- Programming is quite cumbersome

Unix Signals

SIGNAL	ID	DEFAULT	DESCRIPTION
SIGHUP	1	Termination	Hang up on controlling terminal
SIGINT	2	Termination	Interrupt. Generated when we enter CTRL-C
SIGQUIT	3	Core	Generated when at terminal we enter CTRL-\
SIGILL	4	Core	Generated when we execute an illegal instruction
SIGTRAP	5	Core	Trace trap (not reset when caught)
SIGABRT	6	Core	Generated by the abort function
SIGFPE	8	Core	Floating Point error
SIGKILL	9	Termination	Termination (can't catch, block, ignore)
SIGBUS	10	Core	Generated in case of hardware fault or invalid address
SIGSEGV	11	Core	Generated in case of illegal address
SIGSYS	12	Core	Generated when we use a bad argument in a system service call
SIGPIPE	13	Termination	Generated when writing to a pipe/socket when no reader anymore
SIGALRM	14	Termination	Generated by clock when alarm expires
SIGTERM	15	Termination	Software termination signal
SIGURG	16	Ignore	Urgent condition on IO channel
SIGCHLD	20	Ignore	A child process has terminated or stopped
SIGTTIN	21	Stop	Generated when a background process reads from terminal
SIGTTOUT	22	Stop	Generated when a background process writes to terminal
SIGXCPU	24	Discard	CPU time has expired
SIGUSR1	30	Termination	User defiled signal 1
SIGUSR2	31	Termination	User defined signal 2

Recommended Reading

- Bacon, J.: OS (9, 10, 11)
    - Exhaustive (all POSIX thread functions)
    - Event handling, Path Expressions etc.
- Nehmer, J.: Grundlagen moderner BS (6, 7, 8)
- Silberschatz, A.: OS Concepts (3, 4, 6)
- Stallings, W.: OS (5, 6)
- Tanenbaum, A.: MOS (2)

Mutual Exclusion

Critical Sections

When a thread accesses shared data or an exclusive resource, ⇒ thread executes a critical section (CS)

A thread may have different CSs, even nested ones

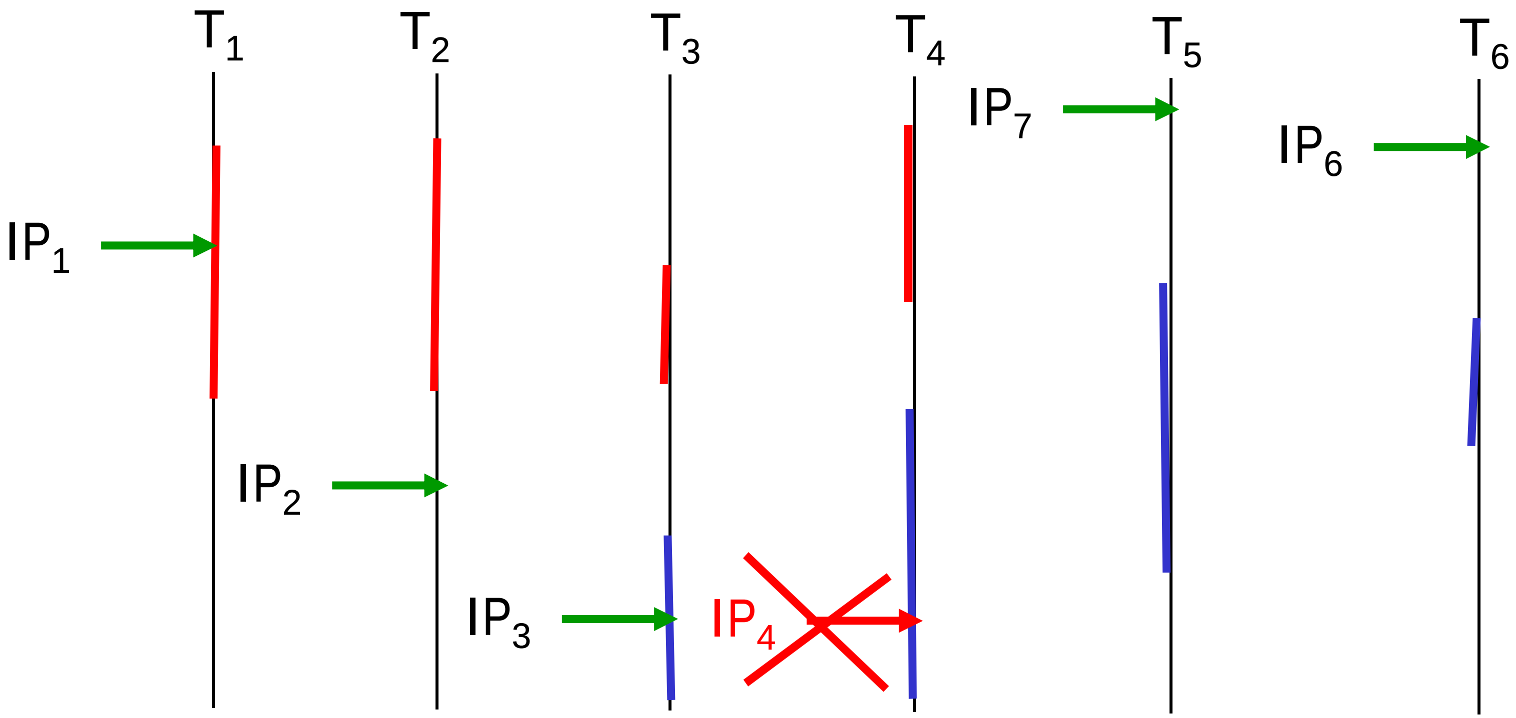
Executing a CS must be mutually exclusive, i.e. at any time, only 1 thread is allowed to execute the *related CS*

⇒ Each thread must request the permission
to enter a critical section (CS),
i.e. it must *obey a certain protocol*

Critical Sections

Suppose: All Ti are KLTs of same Task
(IP of) T_1 is in its "red CS"

Question: *What IP_i are valid at the same time?*



Again Counting Semaphore

Semantic for *"mutual" exclusion* of CSs:

1. *Positive* value of counter→ #threads that can enter their CS
    - If mutual exclusion, **# allowed threads = 1**
2. *Negative* value of counter → #waiting threads in front of CS, i.e. being queued at semaphore object
3. *Counter == 0* → no thread is waiting respectively maximal #threads currently in CS

- Still an open problem:
  How to establish "atomic semaphore-operations"?

Application of Counting Semaphores

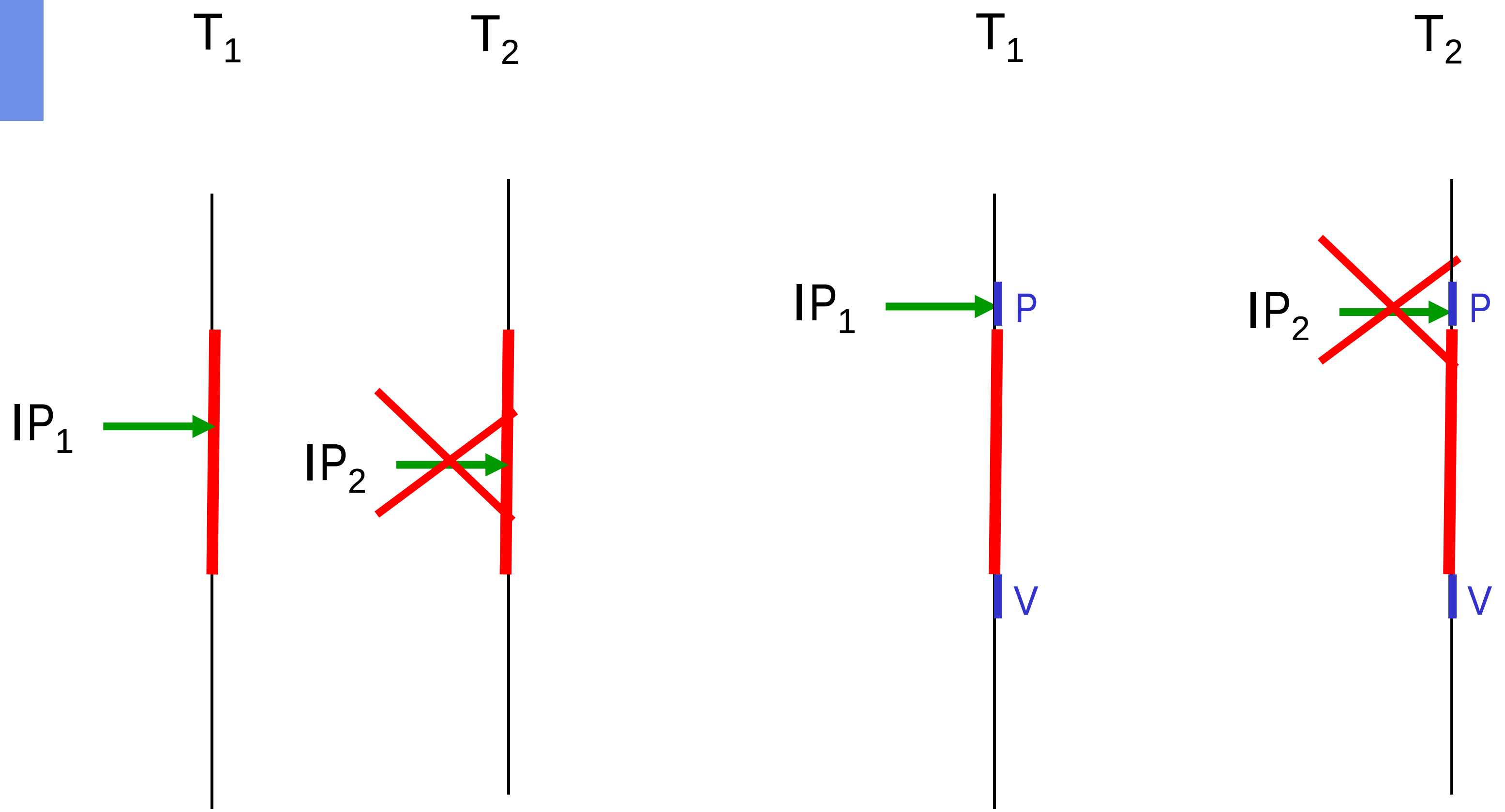
Suppose: *n* concurrent threads

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

Initialize S.Count to k ⇒
k threads allowed to enter their “CS”

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

Why Atomic Semaphore Operation?



We have to implement P() and V() in such a way, that these operations are hopefully shorter critical sections!!!

Atomic Semaphore Operation

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

Monitors

Monitor (1)

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

Monitor (2)

A software module* containing:

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

Characteristics:

- local variables accessible only by monitor's procedures
- thread enters the monitor by invoking an interface procedure
- only one thread can be executed in the monitor at any time, i.e. a monitor may be used for implementing *mutual exclusion*

*Java's synchronized classes enable monitor-objects (already used in Assignment 2)

Monitor (3)

Monitor already ensures *mutual exclusion* ⇒
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

Condition Variables

in Java: **`wait()`**

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 will execute 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

In Java: **`notify()`** or **`notifyAll()`**

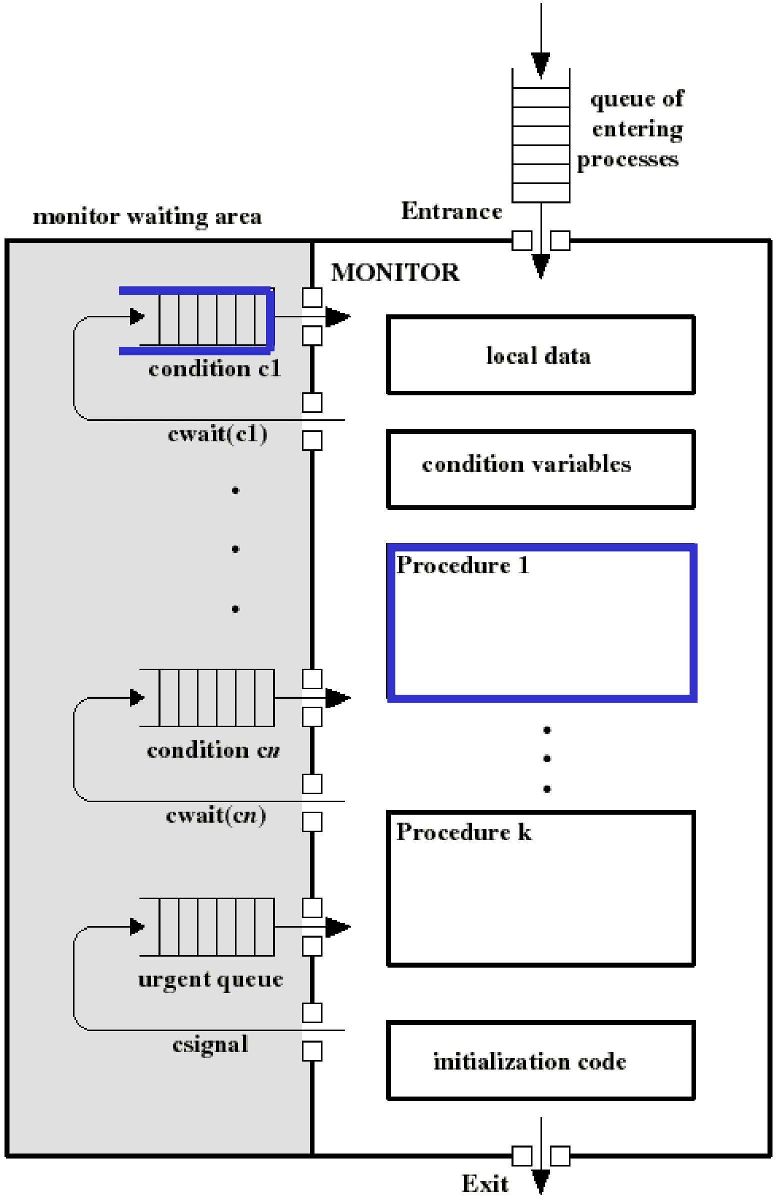
Monitor (4)

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)

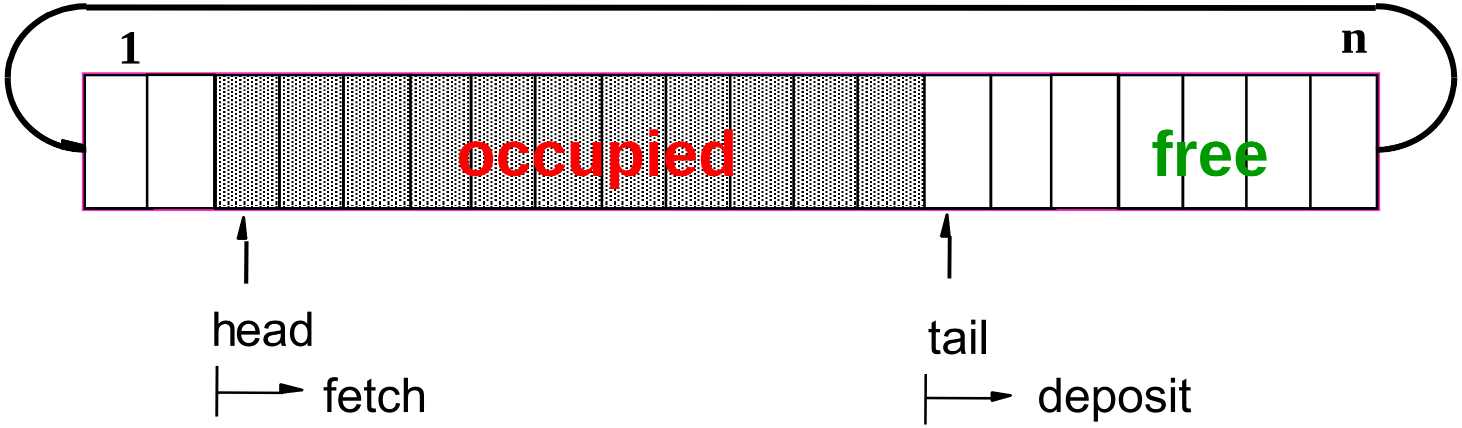
CondSignal(cn) enables one thread, waiting at condition queue cn, to continue

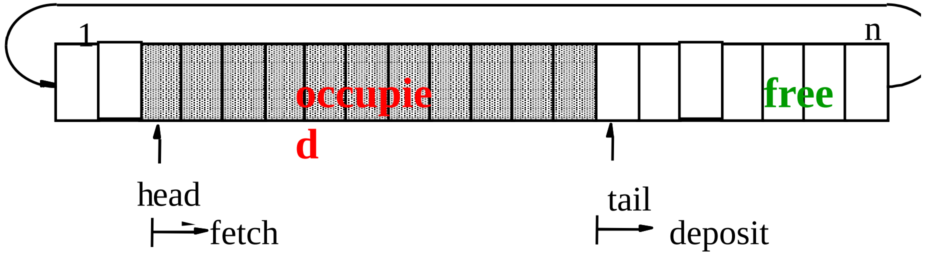
Hence CondSignal(cn) blocks the calling thread and puts it into the urgent queue (unless csignal is the last operation of the monitor procedure)



Example of a Monitor* (without condition variables)

Contiguous array as the cyclic buffer of N slots with interface operations fetch() and deposit()





```
monitor module bounded_buffer
 buffer = record
   array buffer[N] of datatype
   head: integer = 0
   tail: integer = 0
   count: integer = 0
 end
 monitor procedure deposit(b:buffer, d:datatype)
 begin
   b.buffer[b.tail] = d
   b.tail = b.tail mod N
   b.count = b.count + 1
 end
```

Automatically with mutual exclusion

```
 procedure fetch(b:buffer, result:datatype)
 begin
   result = b.buffer[b.head]
   b.head = b.head mod N
   b.count = b.count - 1
 end
end monitor modul
```

Automatically with mutual exclusion

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! ⇒ two additional constraints have to be considered.

Monitor Solution

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
```