System Architecture

8 Coordination

Concurrency Problems
Orthogonal Design Parameter
Unilateral synchronization (Signals)

November 19 2008
Winter Term 2008/09
Gerd Liefländer

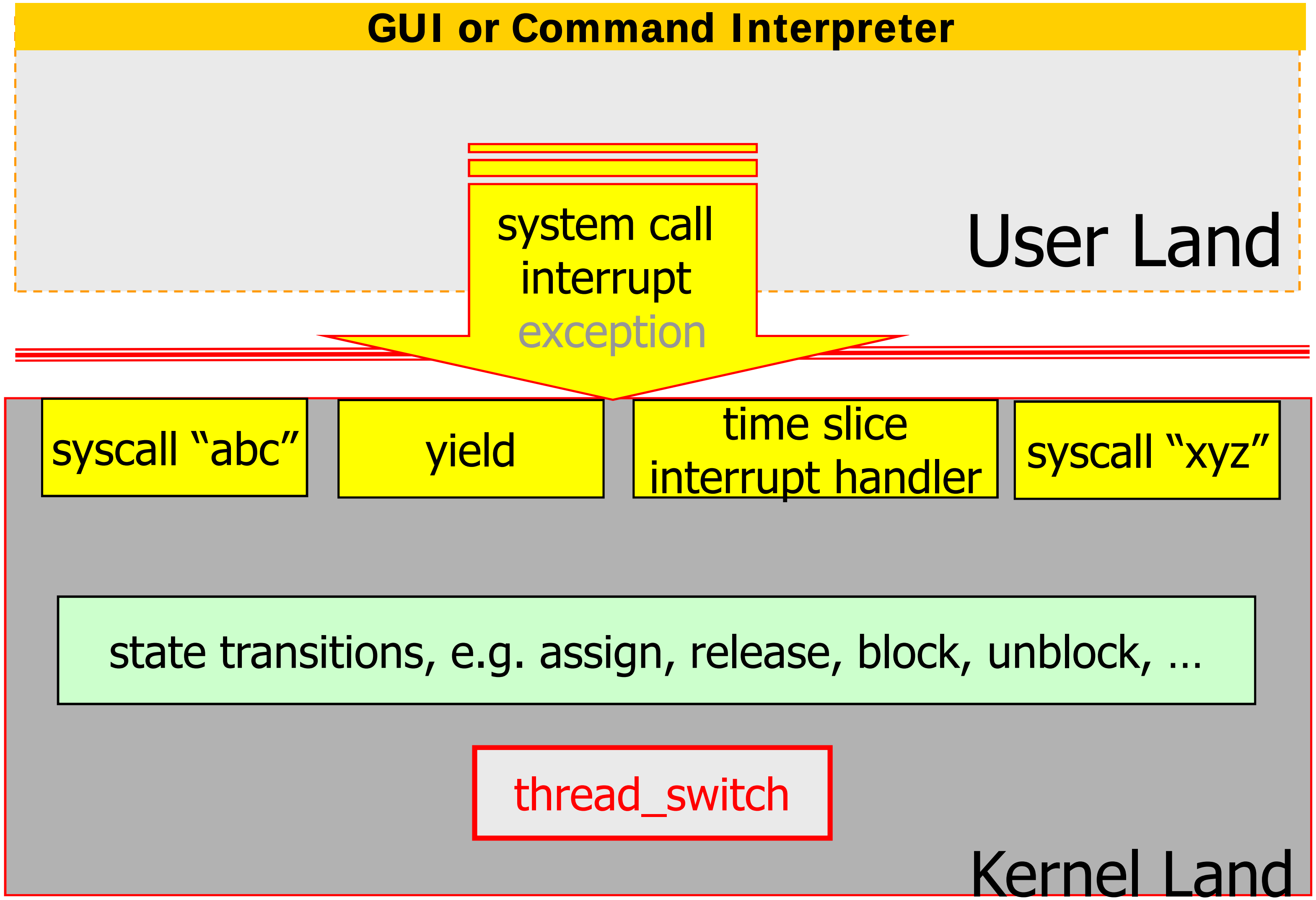
Literature

- 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)
- Additional Research Papers

Agenda

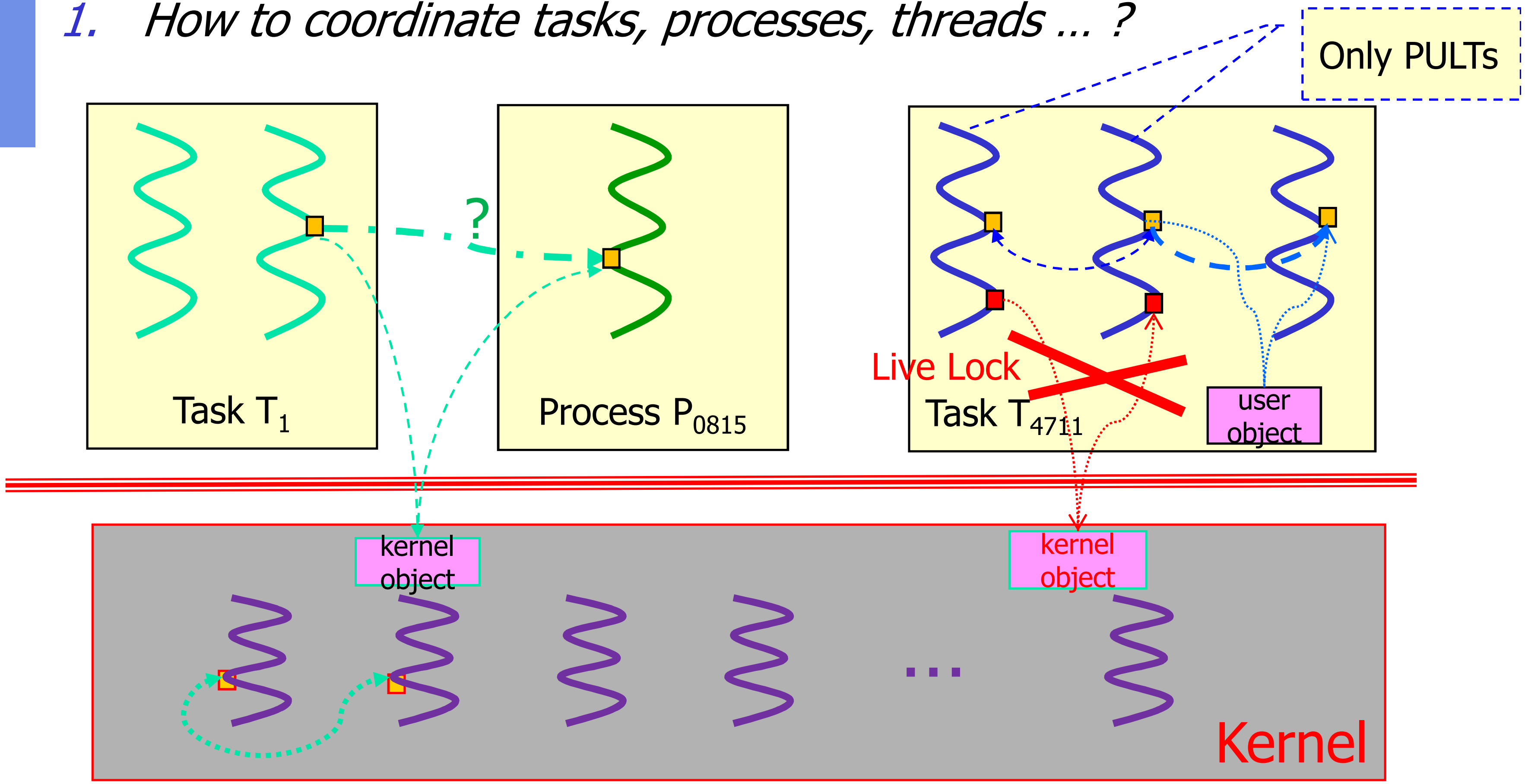
- Review: Discussed Kernel Components
- Introduction & Motivation
  - Example Problems
  - Definitions and Notions
    - Condition Synchronization(Signaling)
    - Mutual Exclusion
- Standard Concurrency Problems
- Design Space of Condition Synchronization
  - Signaling
    - Flags
    - Semaphores
  - Mutual Exclusion

Discussed Kernel Components



Two Levels: Controlling Concurrency

1. *How to coordinate tasks, processes, threads … ?*



2. *How to coordinate kernel activities, i.e. kernel mode threads, exceptions handlers and interrupt handlers?*

Motivation & Introduction

Why do we need coordination?

Whom do we have to coordinate?

Activities of same or of different kinds?

Can we classify coordination problems?

Why Coordination?

Threads access exclusive resources and/or access common data (e.g. objects in a shared memory or records of a shared files)

- If result of applications depends on the execution sequence of the threads → race condition
- Concurrent threads might have access conflicts, e.g. competing for exclusive resources
- If there is no controlled access to common data, threads can produce inconsistent data

Bad Timing

Need for protocols

Too many cooks spoil the broth

Whom to coordinate?

- In many cases, concurrent activities of the same kind have to be coordinated, e.g. because
  - they access common data within a task, e.g. k>1 KLTs share a file
    - to preserve file consistency, we must coordinate file updates
  - they share the same physical/logical resource
- In some cases, a process must be coordinated with all PULTs of another multi-threaded task, because they all compete for the same physical device, e.g. the network-card

Classification of Coordination

- Condition Synchronization
  - Waiting for the occurrence of a condition
  - Goal: determine a specific order of operations

- Mutual Exclusion
  - Prevent concurrent access to exclusive resources
  - Goal: consistency of data

Orthogonal Concepts

- Condition synchronization and mutual exclusion

Mutual Exclusion	Condition Syncronization	Goal
-	-	Independent Activities
-	+	Precedence Relation
+	-	Data Consistency
+	+	Data Consistency & Precedence Relation

Shared Resources?

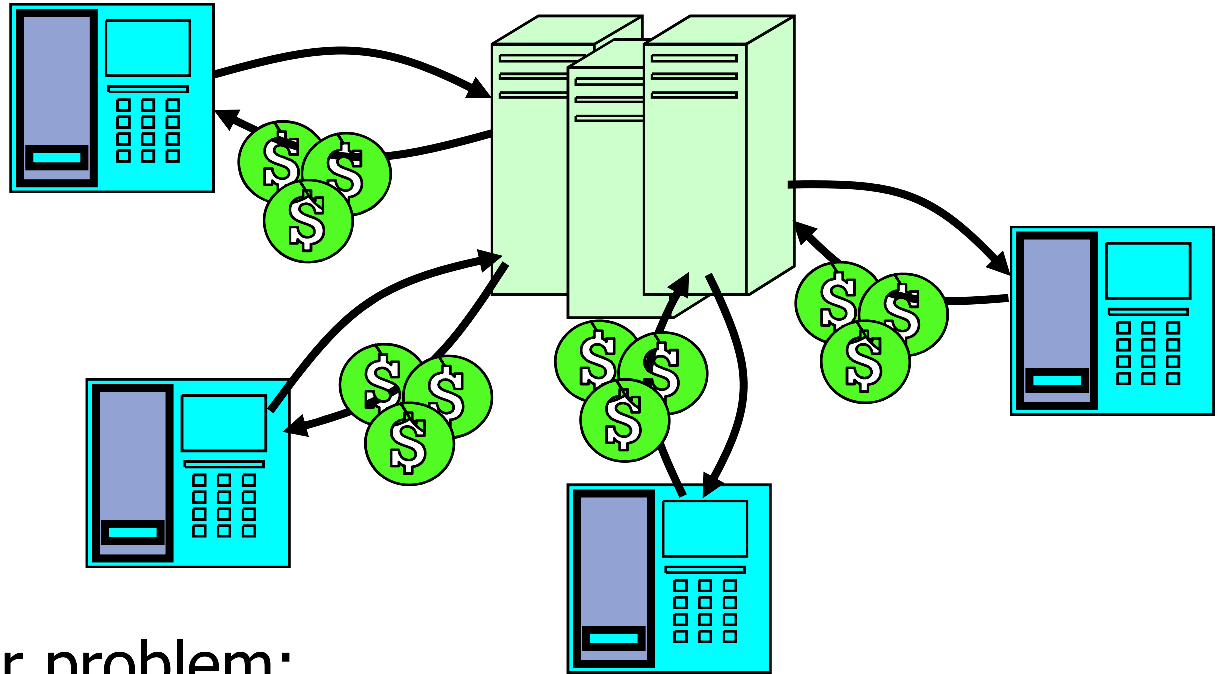
Basic problem:

- At least two concurrent threads access a shared object
- If shared object is modified by at least one thread, access must be serialized

- We'll study concepts & mechanism to control access to shared resources
  - HW-mechanisms: TestAndSet, …, EnableInterrupt
  - Low level mechanisms: Condition variables
  - high level mechanisms: semaphore, monitors
  - User land algorithms: Dekker, Peterson

- Any coordination stuff is complicated & rife with pitfalls
  - There are "solutions" even in OS-textbooks that are invalid
  - Details are important to get a valid solution

ATM Bank Server



- ATM server problem:
  - Service a set of requests
  - Do so without corrupting the underlying database containing your and others' bank accounts
  - Neither hand out too much money nor too few money

Bank Account Example

Design a function to withdraw money from your bank account:

```
int withdraw(account, amount){
  balance = get_balance(account);
  balance -= amount; /*balance is local*/
  put_balance(account, balance);
  return balance;}
```

Now suppose that you and your spouse share a bank account with a current balance of $ 2000.00

- *What might happen if you both go to separate ATMs* and simultaneously withdraw $100 respectively $1000 from your account?*

*ATM = automated teller machine

Bank Account Example (2)

We model these concurrent actions by two threads, for each ATM one withdrawal thread:

```
Thread 1
int withdraw(account, amount){
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;}
```

you

```
Thread 2
int withdraw(account, amount){
    balance = get_balance(account);
    balance -= amount;
    put_balance(account, balance);
    return balance;}
```

spouse

- *Any problems?*
- *What are the possible balance values after each thread has terminated?*

Bank Account Example (3)

Interleaved execution of both threads due to

- preemptive scheduling, i.e.
- there might be a thread_switch after each instruction

Let's study a possible interleaved execution trace of both threads:

```
balance = get_balance(account);
balance -= amount;    /*  100 $ */
```

thread_switch

```
balance = get_balance(account);
balance -= amount;    /* 1000 $ */
put_balance(account, balance);
```

thread_switch

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

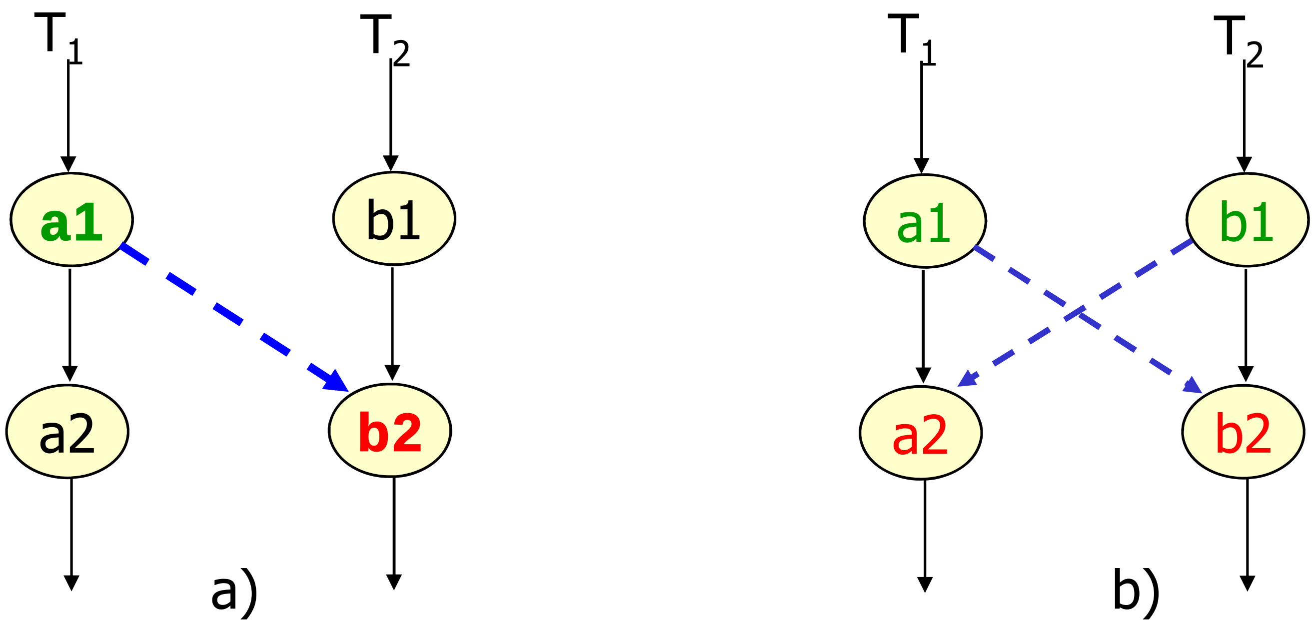
What's the account balance afterwards?

- *Who is happier, your bank or you and/or your spouse?*

Design Parameters

- Type(s) of Involved Activities
  - Only PULTs of the same task
  - Only KLTs of the same task
  - KLTs of different tasks
  - Processes and KLTs
- Number of Involved Activities
  - Only 2 activities
  - a >2 activities
  - a>>2 (we must find highly scalable solutions)
- Uni-/Multilateral Synchronization
- Busy waiting or blocked waiting (or adaptive waiting?)
- Levels of Implementation
  - User level
  - Kernel level
  - HW level

Uni-/Multilateral Synchronization

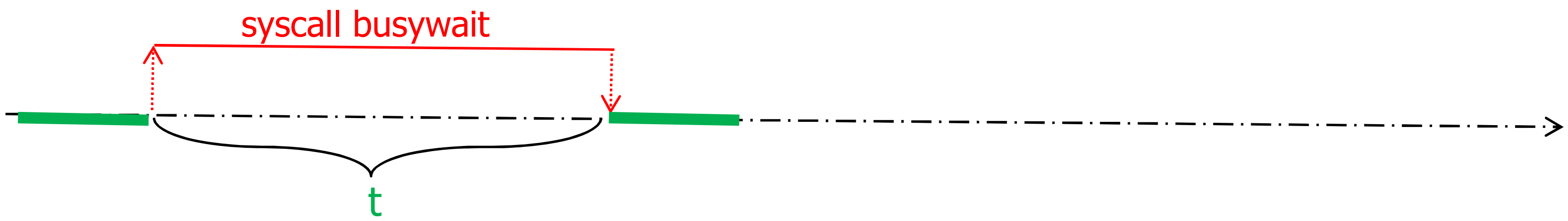


a) Unilateral: section b2 waits until section a1 has completed, sections a1 & a2, i.e. thread T_1 is not affected at all

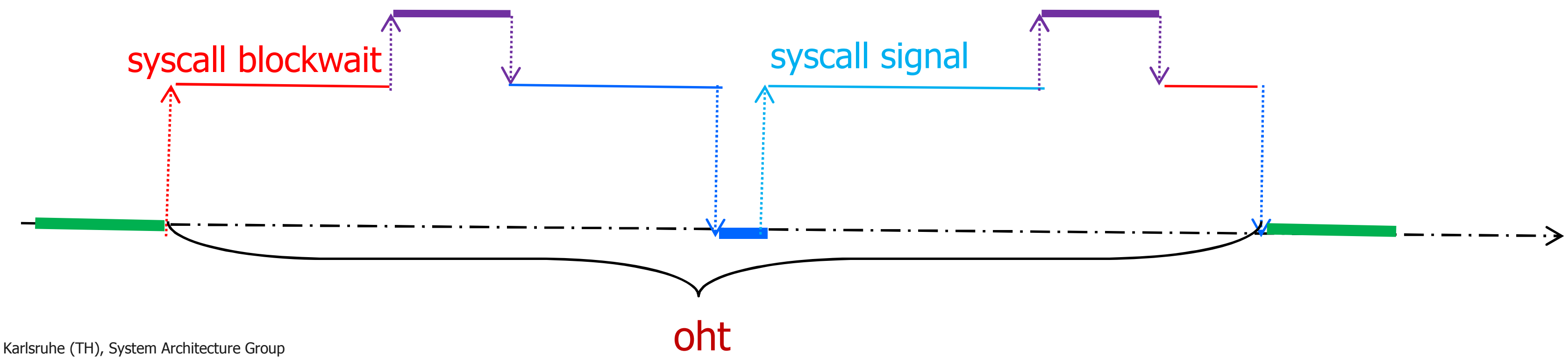
b) Multilateral: sections b2 and a2 wait until section a1 respectively b1 have completed.

Busy/Blocked Waiting

- Busy waiting
  - Whenever you can guarantee that the event will occur within t time units whereby the overhead for blocking (& later deblocking) the activity is oht > t time units



- Blocking waiting in all other cases



Levels of Implementation

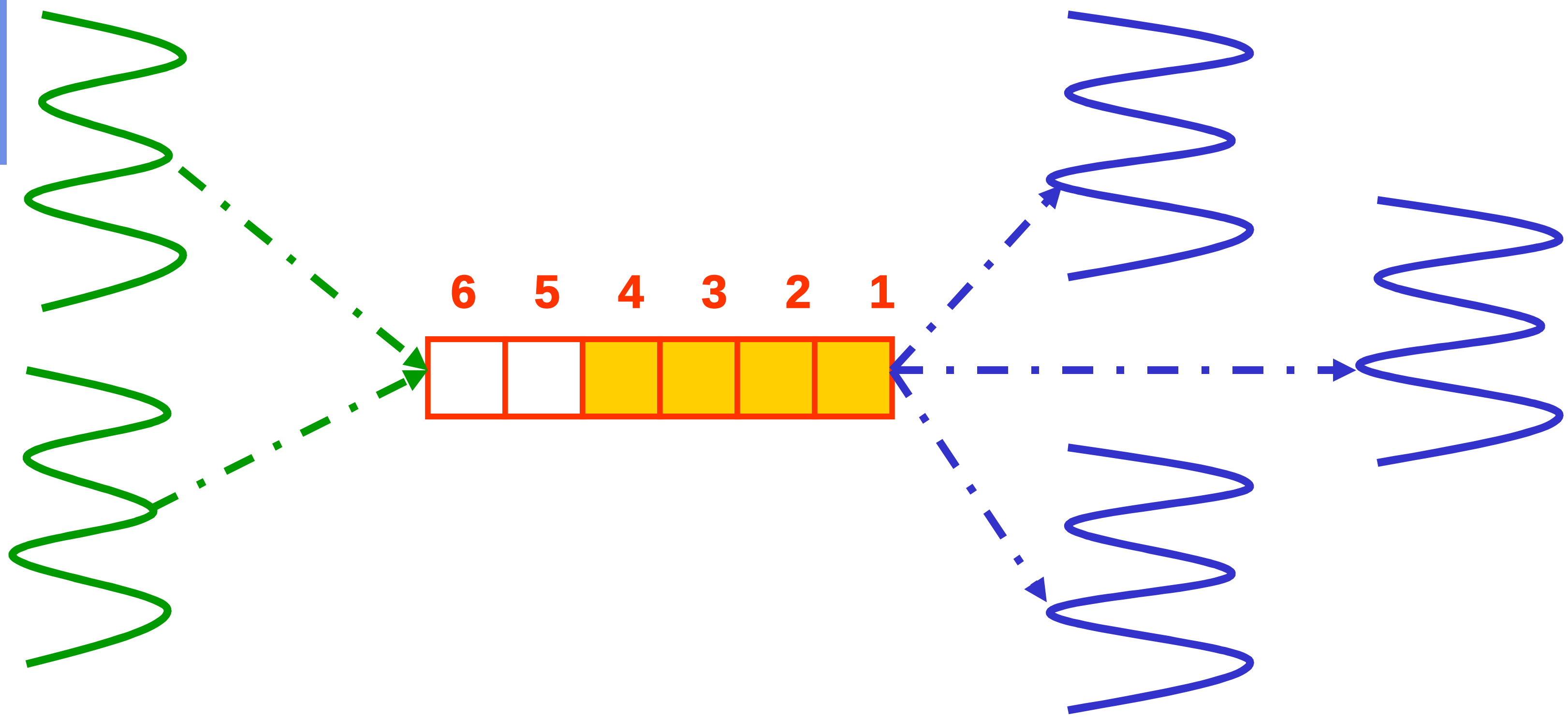
Application Programs	synchronized methods or synchronization algorithms
Kernel API	Various Locks Semaphores Monitors Send/Receive
Hardware	Load/Store Disable Ints Test&Set Comp&Swap …

- We are going to design & implement various synchronization primitives using atomic operations
  - Everything is painful if the only atomic HW instructions are `load` and `store`
  - We must provide primitives, useful at kernel- and at user-level

Concurrency Problems

Standard Problems

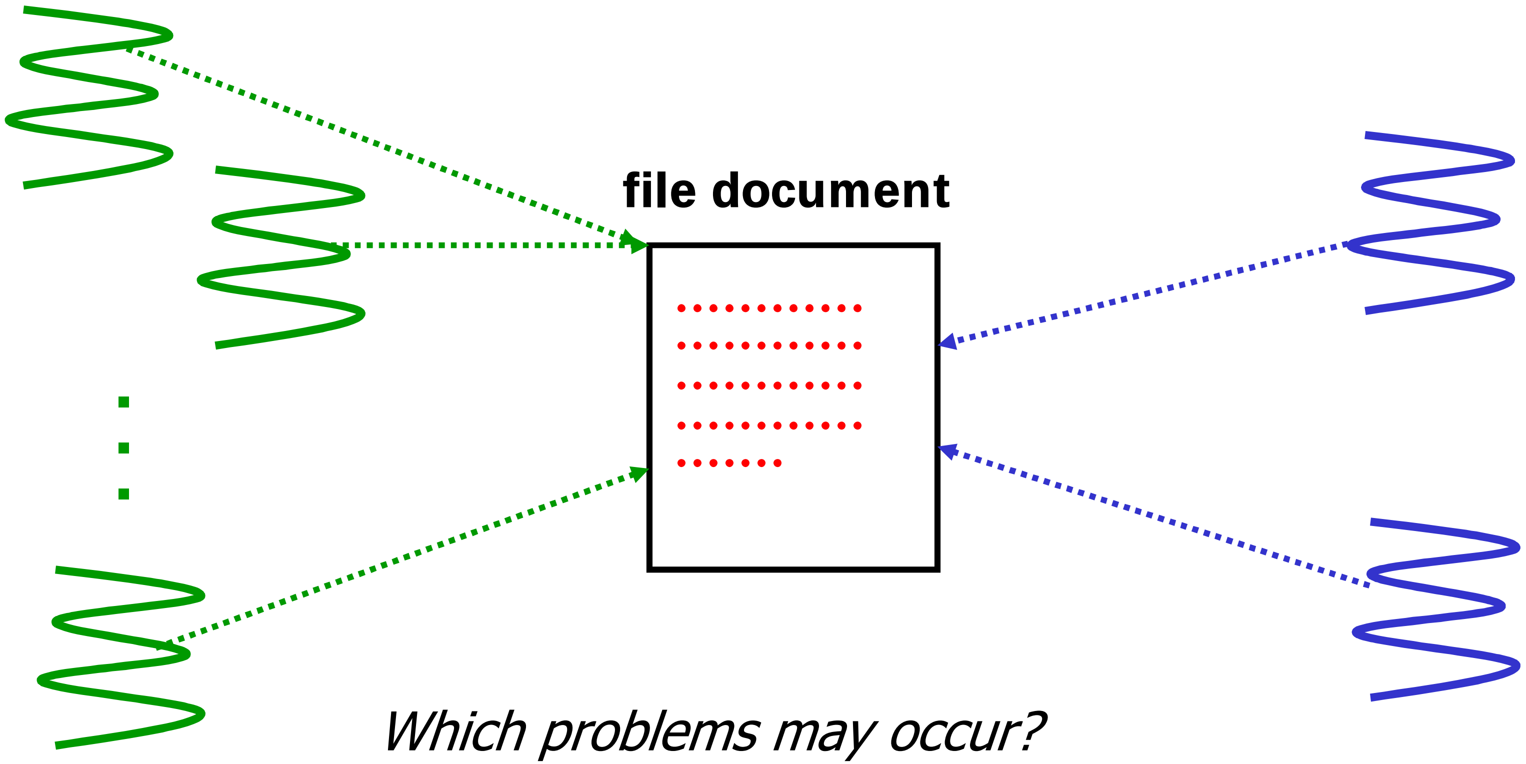
Producer/Consumer (Bounded Buffer)



Problems with bounded buffer?

Additional problems with p>1 producer or c>1 consumer?

Reader/Writer Problem



Which problems may occur?

⇒ Data inconsistency

Dining Philosopher Problem[1]

Life of a philosopher:

```
repeat forever
begin
 thinking
 getting hungry
 getting the two
 neighbored forks
 eating
end
```

?

Requirements:

No Starvation

No Deadlock

[1]Edsgar Wybe Dijkstra

Condition Synchronization or Signaling

HW Signals & Interrupt Handling

Simple Signal Objects

Handling Interrupts

Complex Signal Objects

Semantics of Simple[1] Signaling

- Only one of the involved activities potentially waits, the other is notifying(signaling) its partner activity
- Simple signaling is used at various levels:
  - HW-level
    - Peripheral device sends I/O signal (interrupt)
    - CPU sends inter-processor-signal to another CPU (SMP)
  - SW-level
    - Thread T_i wants to notify another thread when T_i has reached a certain point (=IP value) within its program
    - Process sends an abort signal (kill) to one its family processes

[1]Only 2 activities are involved

Interrupt Handling

- Scheduling interrupt handlers is often prescribed by HW design
  - Sequential versus nested interrupt handling
  - Interrupt priorities
  - Round robin interrupt handling
- Depending on the HW implementation some devices share one single “interrupt line”
- In any case, an interrupt handler has to guarantee to handle all interrupts otherwise events might be lost
- In some extremely **critical kernel code paths** a CPU does not want to be interrupted, i.e. it **disables** (or **masks out**) all or at least some HW interrupts

First Approach: Design a Signal

- Flag (1 = Signal set, 0 = Signal not valid)
  - Signal only reflects that a event had happened
- Vector of flags a bit more comfortable (see Unix)
  - You can distinguish between specific signal events
  - You might order signal vector according to signal priorities
- Counter
  - Each value might have a different meaning or
  - Value reflects the number of pending signals
- Implement **`flag`** or **`counter`** as global variable

see **`http://www.frostbytes.com/~jimf/papers/signals/signals.html`**

First View Analysis

- All `flag` oriented solutions suffer from the fact:
  - If a signal is not accepted in time, it is lost
- Use flags in environments (e.g. inside the kernel) with a deterministic behavior of all involved activities
- Busy waiting for signals to arrive can lead to system starvation

Interrupt Handling

- Timer interrupt (enabling time slice mechanism)
- All other interrupts are handled similarly
  - Interrupt handling can involve a thread_switch

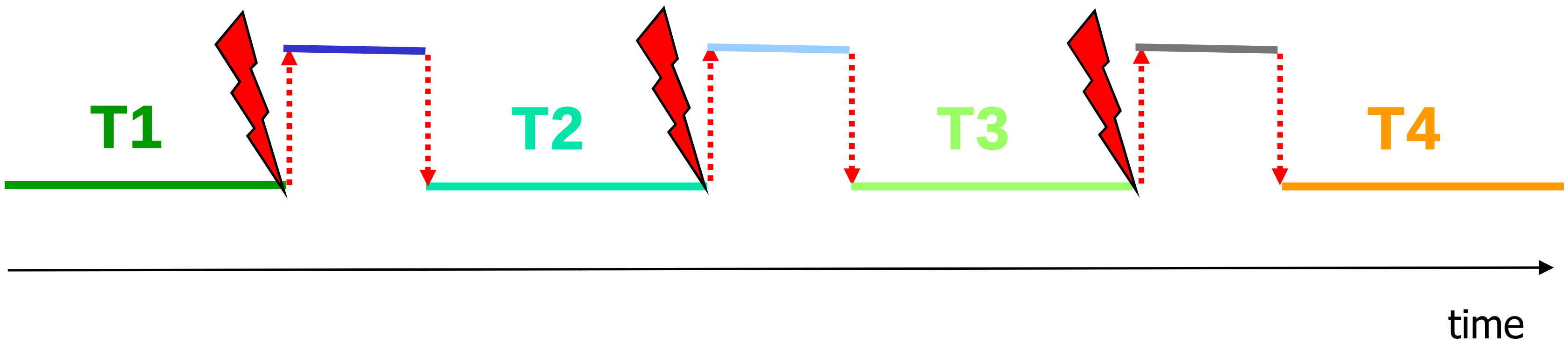
1. HW saves user land context of the interrupted user land activity, in case it has to resume it, i.e.
   1. immediately after handling the current interrupt or
   2. some time later, depending on scheduling

2. HW saves kernel land context of a kernel activity
   1. However, sometimes the current kernel activity is in a “critical section”, i.e. a previous interrupt handler
   2. We need HW support to protect those very

      ”low critical sections”

How to use Flags?

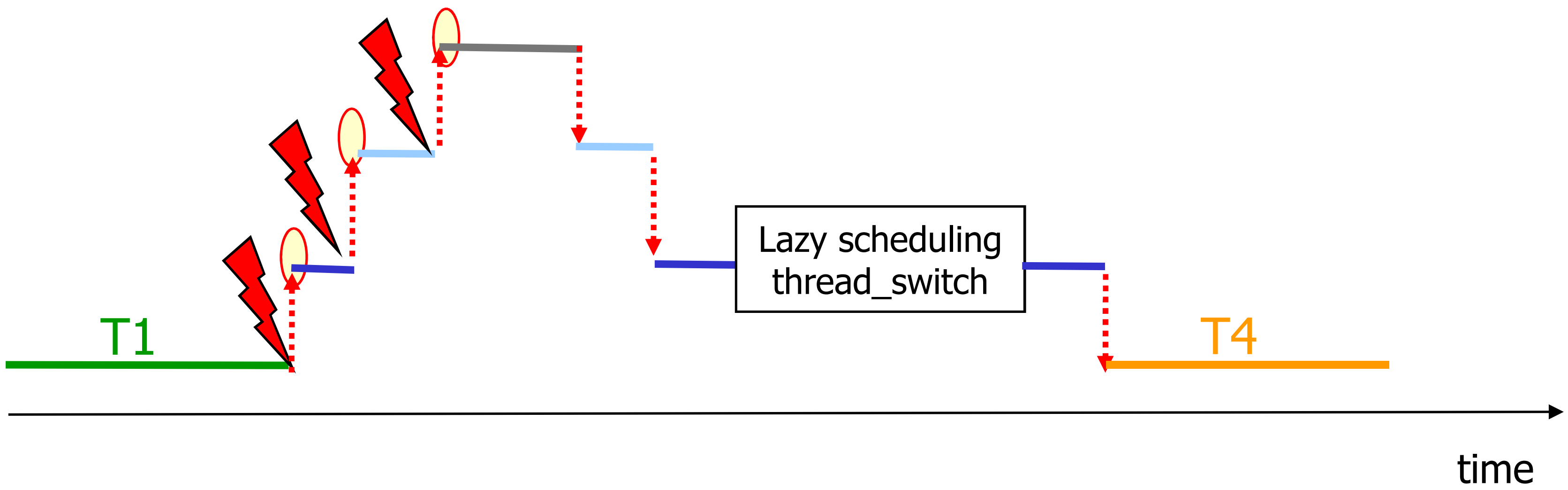
- Flags are used in almost every system
- Assume: Inside every interrupt handler another user level activity has to be assigned for whatever reason, i.e. every preempted activity T_i will be preempted by another one



Suppose the interrupts occur with short arrival times, and you support a nested interrupt schema.

Example: Usage of Flags

- Lazy scheduling and thread switching: wait until you have to return from kernel to user land
- Every interrupted IR-handler with a potential scheduler activity sets the "scheduling flag"
- Benefit: You avoid system overhead (thread_switch or even process_switch) if T2 or T3 belong to different address spaces



Low level critical sections, protected via privileged instructions allowing to disable "all" or at least "some" interrupts

Signal Objects at User Level

Avoiding Signals

Implementing Signal Objects

Producer/Consumer (Bounded Buffer)

Assume: Both, producer and consumer are KLTs
Bounded buffer is a global array

Coordination problems:

- The producer having put another item into the buffer notifies the consumer that there is another item in the buffer (to be taken)
- The consumer having taken an item from the buffer notifies the producer that there is another free slot in the buffer (to be filled)
- The producer has to wait when the buffer is full
- The consumer has to wait when the buffer is empty

Approach 1: Cons./Prod. Problem

```
#define BUFFER_SIZE 10
typedef struct {              /* Buffer element   */
        . . .
} item;
item buffer[BUFFER_SIZE];  /* Contiguous buffer */
int in = 0;                /* Pointer for filling */
int out = 0;               /* Pointer for deleting */

void producer(void){      /* executed as a thread */
 item nextProduced;
 while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing, but busy waiting */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
 }
}
```

Always 1 buffer element is not used

Approach 1: Cons./Prod. Problem

```
#define BUFFER_SIZE 10
Typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
void producer() {
 item nextProduced;
 while (1) {
  while (((in + 1) % BUFFER_SIZE) == out)
    ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
 }
}}
```

```
void consumer(void){
 item nextConsumed;
 while (1) {
   while (in == out)
      ; /* do nothing */
   nextConsumed = buffer[out];
   out = (out + 1) % BUFFER_SIZE;
 }
}}
```

You can only use up to BUFFER_SIZE-1 elements!

Efficient? Scalable?

Scalability?

Producer/consumer approach 1 scalable?

- When it works effectively with
  - $p \geq 1$ producers and/or
  - $c \geq 1$ consumers and/or
  - $0<b<MAXINT$ sized buffers
- Previous solution is not scalable
  - Only valid for $p = 1$ & $c = 1$ & BUFFER_SIZE $\neq 1$

Efficiency?

When is a producer/consumer solution efficient?

- When it works fast and with low overhead
- Previous solution is not efficient because
  - Busy waiting wastes CPU cycles
  - It’s up to the kernel scheduler (iff scheduler will be activated) to prevent inefficient busy waiting

How to improve?

  - Use the kernel API to avoid busy waiting

Kernel API to avoid Busy Waiting

- **`BLOCK()`**
  - ~ Tanenbaum's and Unix's **`sleep()`**
- **`UNBLOCK()`**
  - ~ Tanenbaum's and Unix's **`wakeup()`**
- **`YIELD()`**
  - Anonymous **`yield()`** of KLTs

System Call BLOCK*

```
BLOCK(condition c, myself)
{
  block(CT, c.SWT) {state transition}
  NT = Schedule()
  CT = Thread_Switch(NT)
  assign(CT)       {state transition}

}
```

*Solution for the 3-state model;
analogous functions UNBLOCK or YIELD

2nd Approach Producer/Consumer

```
#define N 100                        /*number of slots in buffer*/
int count = 0;                       /*number of items in buffer*/

void producer(void){
 int item;
 while (1) {                                     /*repeat forever*/
   item =produce_item();                      /*produce next item*/
   if (count==N) BLOCK(myself);                   /*if buffer full*/
   insert_item(item);                         /*put item into buffer*/
   count = count + 1;                          /*increment counter*/
   if (count ==1) DEBLOCK(consumer);          /*was buffer empty?*/
 }}

void consumer(void){
 int item;
 while (1) {
   if (count ==0) BLOCK(myself)                   /*if buffer empty*/
   item = remove_item();                 /*take item out of buffer*/
   count = count - 1;                          /*decrement counter*/
   if (count==N-1) DEBLOCK(producer);          /*was buffer full?*/
   consume_item(item);                       /* consume current item*/
}}
```

Looks great, but ...

A race condition on shared variable count

Summary

- Usage of KLT-operations at the kernel-API can be dangerous
- Better to offer well-defined signal-objects or synchronization-objects with atomic methods
- First Attempts:
  - Signal objects with *busy* waiting
  - Signal objects with *blocked* waiting

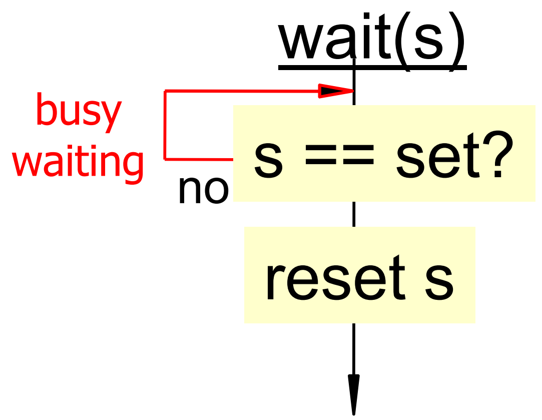
1:1_Signal_Object

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

Thread T1                        Thread T2
.                                .
.                                .
.                                .
{ section a1                     { section b1
  … }                              … }
Signal(T2,s) ······▶             Wait(T1,s)
{ section a2                     { section b2
  … }                              … }
.                                .
.                                .
.                                .
```

1st Approach: Simple Signal Object

`flag s` is a common shared variable of both threads



Hint: Discuss this approach carefully!
Does it work on every system effectively and/or efficiently?
Does it work with any types of activities?

2nd Approach Simple SO

```
module 1:1_signal
 export signal, wait;
 import YIELD;
 type signal = record
     S: signal = reset  /* Initialization important */
 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

Remark: It's similar to the *busy waiting* solution.

3rd Approach Simple SO

```
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        /* necessary or redundancy? */

  end
end module
```

~ sleep and wakeup semantics

Simple implementation of the thread state *blocked*$_i$

4rth Approach Simple SO

```
module 1:1_signal
 export signal, wait
 import UNBLOCK, BLOCK
 type signal = record
     S: signal = reset
     W: waiting thread = nil
 end
 procedure signal(SO:signal)
  begin
  SO.S = set;
  if SO.W ≠ nil then               {thread waiting?}
        UNBLOCK(SO.W)              {unblock it}
  end
 procedure wait(SO:signal)
  begin
  while SO.S = reset do
       SO.W = myself
       BLOCK(myself)
  od
  SO.S =reset
  end
end module
```

Race condition!

Open Problem

Conclusion:

- Signal interface operations `wait()` and `signal()` or `notify()` should be atomic
- *How to achieve this property? (see later)*

Assumption:

Implement the previously mentioned signal objects with the monitor concept at user level or implement them as kernel objects with mutually exclusive interface functions, then some approaches can be valid.

Personal Recommendation:

Avoid any form of busy waiting at user level if you want to produce portable applications[1]

[1]In embedded system you often have proprietary code

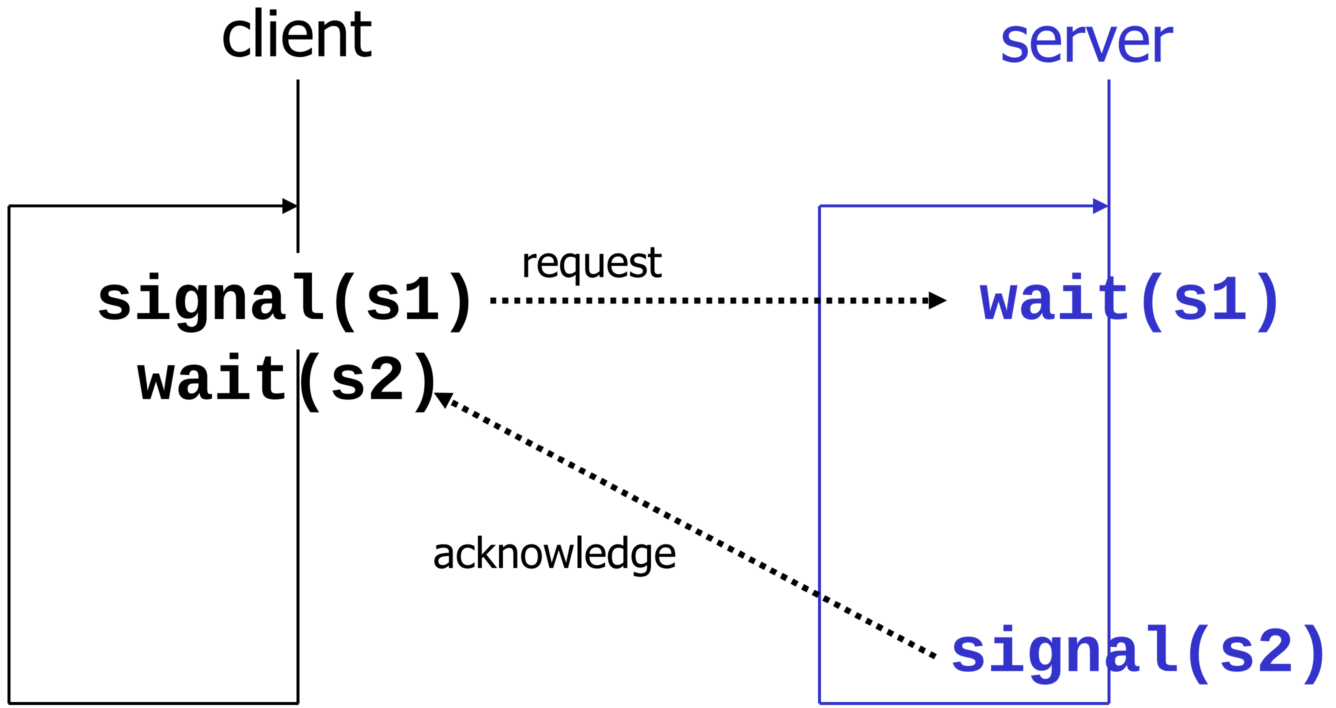
Summary

1. All versions had potential signal losses, i.e. signaling threads may overwrite as-yet unconsumed signals

2. Signal operation = asynchronous, i.e. in case of a non-waiting partner it is not blocked

→ Danger of flooding another thread or even a sub system (e.g. a server)

How to Prevent Flooding?



Remark: **`signal()`** is still asynchronous

Is there a more obvious solution?

Does it help against malicious clients?

Synchronous Signal Object

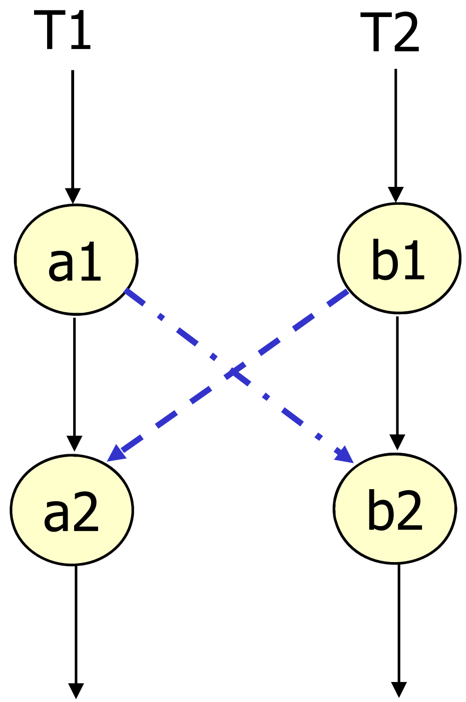
```
module synchronous 1:1_signal
 export signal, wait
 import BLOCK, UNBLOCK
 type signal = record
        S: signal = reset
        SW: waiting thread = NIL    {signaling thread}
        WW: waiting thread = NIL    {waiting thread}
 end
 procedure signal(SO:signal)
  begin
  SO.S = set;
  if SO.W ≠ nil then                {thread waiting}
        UNBLOCK(SO.WW)              {unblock it}
  else  begin SO.SW = myself
              BLOCK(myself) end
  end
 procedure wait (SO:signal)
  begin
  while SO.S == reset do
        SO.WW = myself
        BLOCK(myself)
  od
  SO.S =reset
  UNBLOCK(SO.SW)
  end
end module
```

Evaluate this proposal carefully

Mutual Precedence Relation

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


{section a2
   … }
.
.
.
```




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


{section b2
   … }
.
.
.
```

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

Pure Synchronization

```
sync s;     /* synchronization object */
{Thread 1}                 {Thread 2}
.                          .
.                          .
.                          .
{section a1                {section b1
  … }                        … }
synchronize(s)<---------->synchronize(s)
{section a2                {section b2
  … }                        … }
.                          .
.                          .
.                          .
```

Problem:

How to implement a synchronization object for 2 threads?

Simple Synchronization Object*

```
module synchronization
 export synchronize
 import INBLOCK, BLOCK
 type sync = record
      S: signal = reset
      W: waiting thread = NIL
 end
 procedure Synchronize(SY:sync)
  begin
  if SY.S = reset
  then begin                          {I am first}
          SY.S = set
          SY.W = myself
               BLOCK(myself)  {and wait for my partner}
      end
  else begin                          {I am second and}
          SY.S = reset                {do a reset for future reuse}
          UNBLOCK(SY.W)               {release my partner}
                end
  end
end module
```

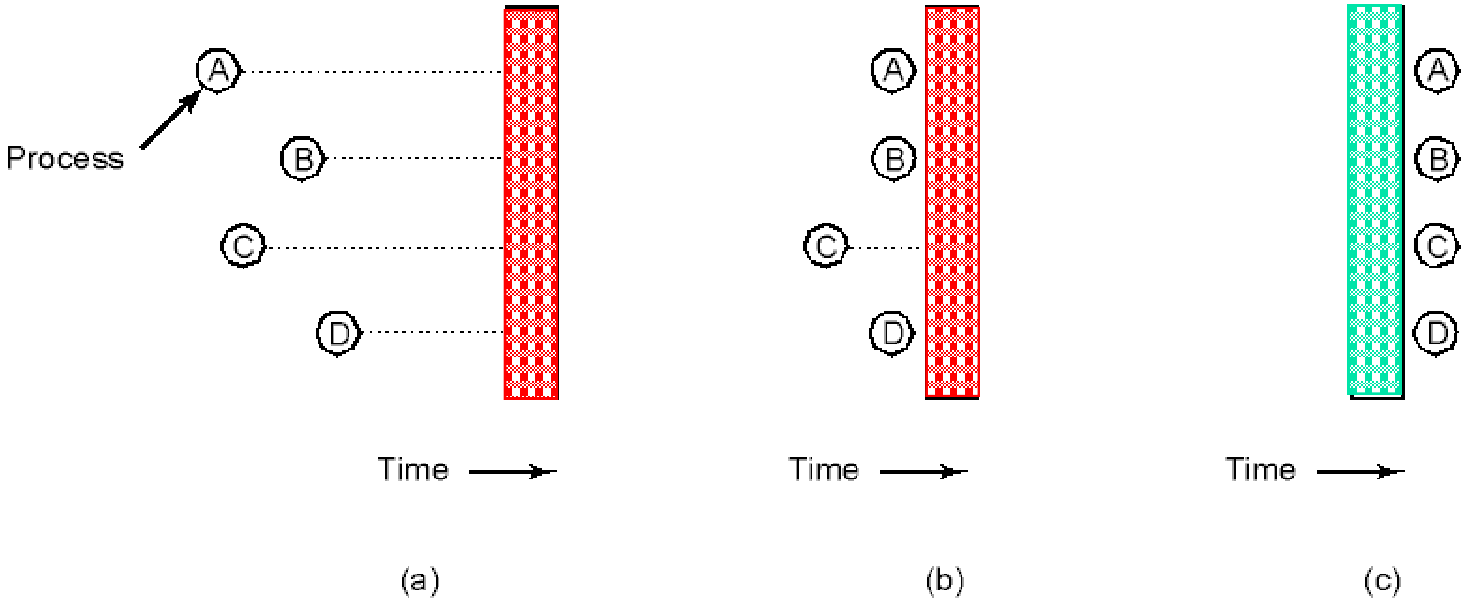
Hint*: Generalize this module for n > 2 threads

Application of N-Way Synchronization[1]

```
…
{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)   {First usage}
  for all i,j
     begin
     old[i,j] = temp[i,j]
     end
  n_synchronize(S)   {Second usage}
  end
…
```

[1] barrier synchronization

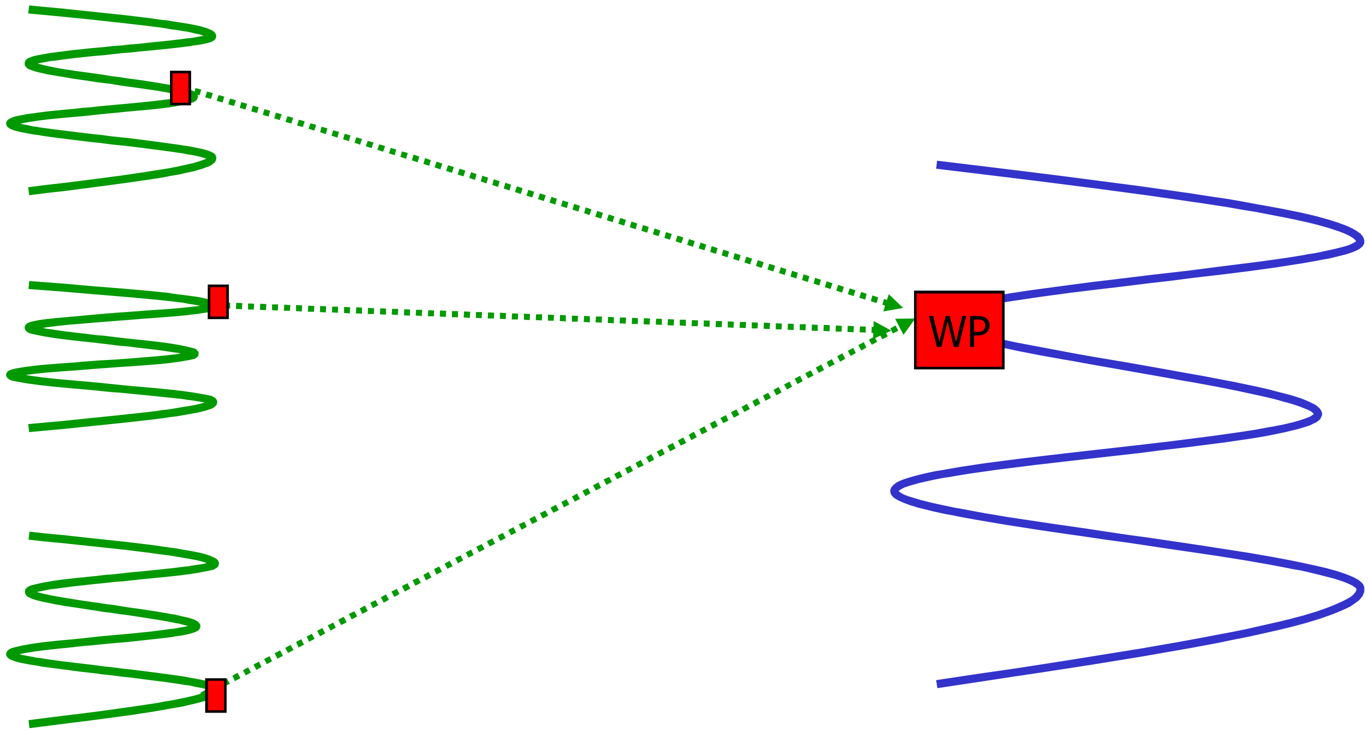
Barrier Synchronization



- Use of a barrier
  - (a) Threads approaching a barrier
  - (b) All Threads but one blocked at barrier
  - (c) Last thread arrives, all can run again

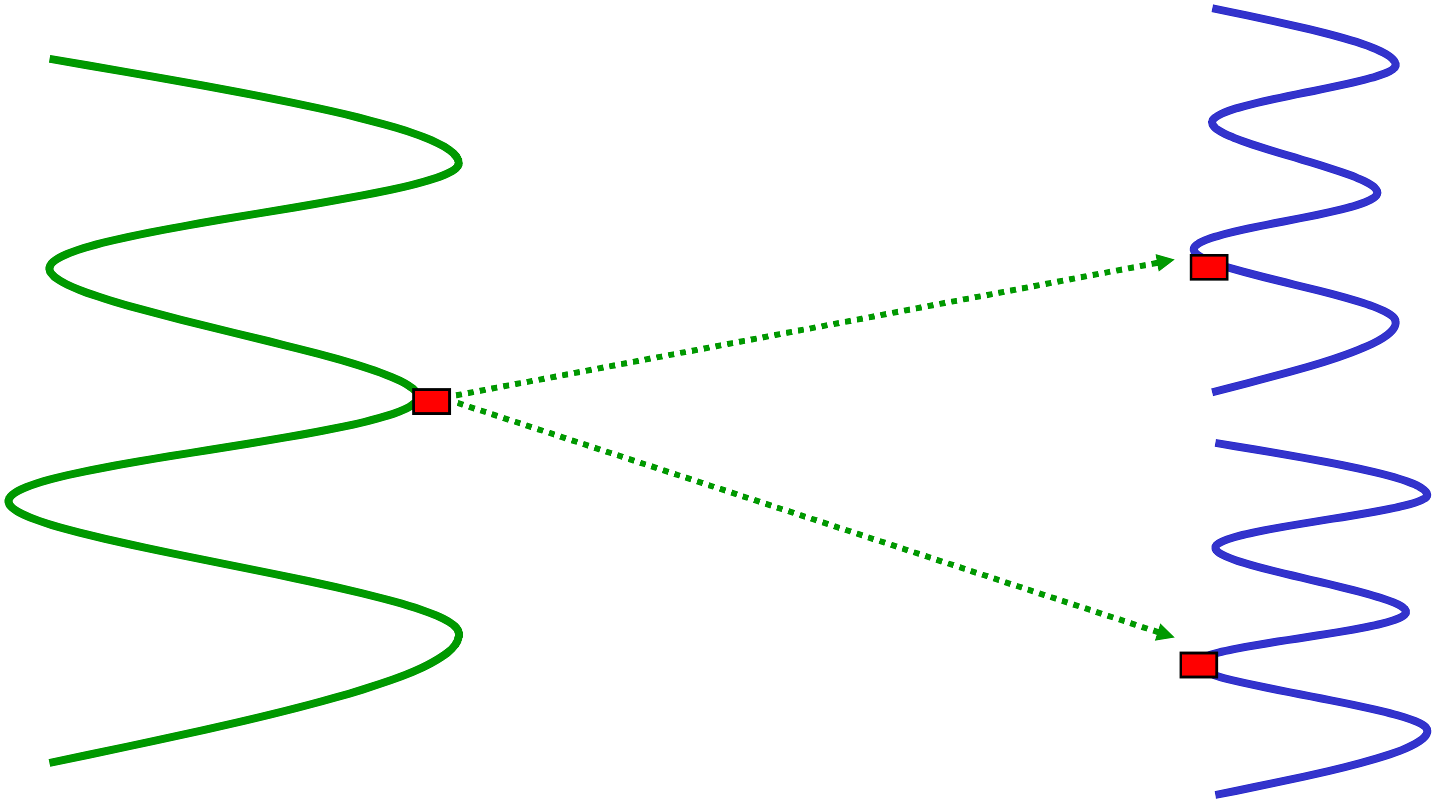
High Level Signal Concepts

Signal Pattern: Many to One (m:1)



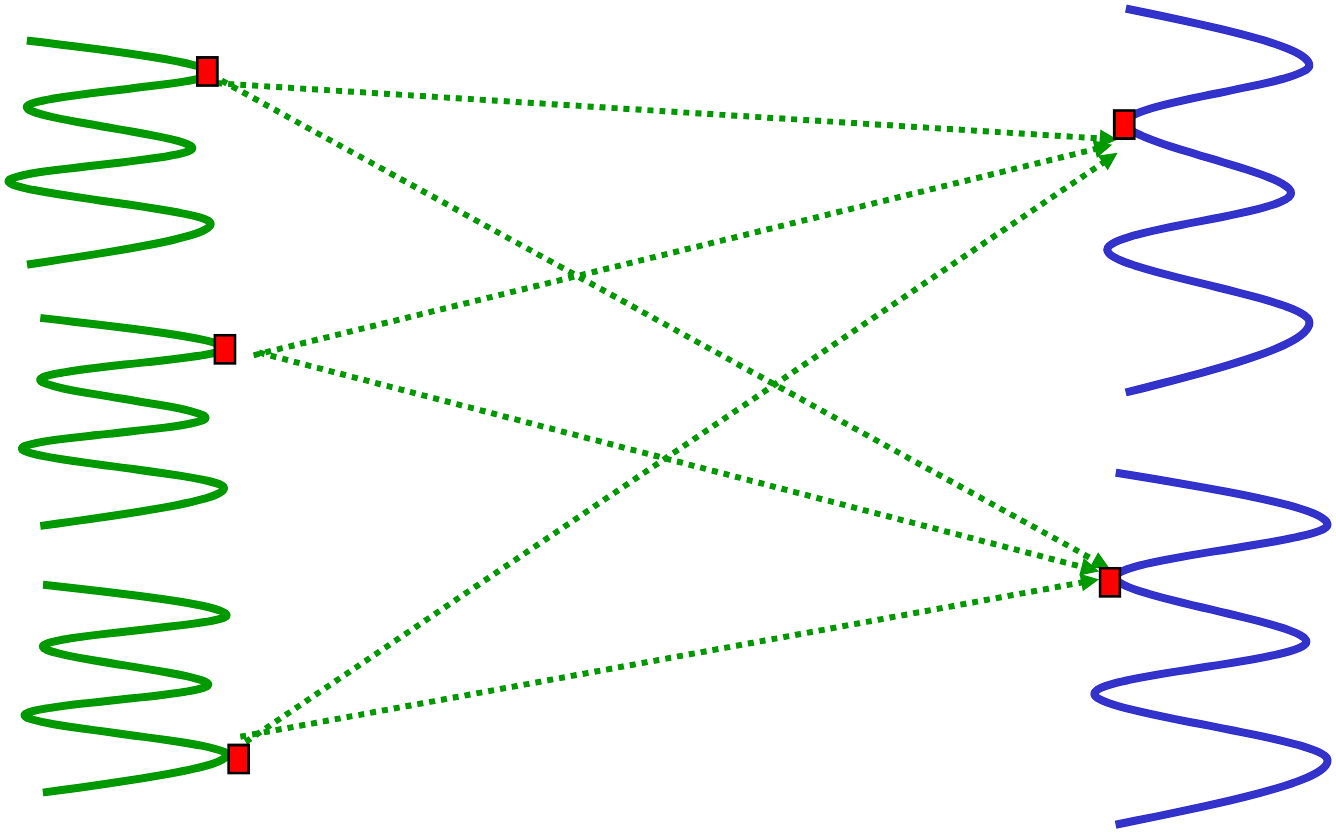
Semantics:
The blue thread can only continue at WP if all the three green threads have reached a specific SP in their code

Pattern: One to Many (1:n)

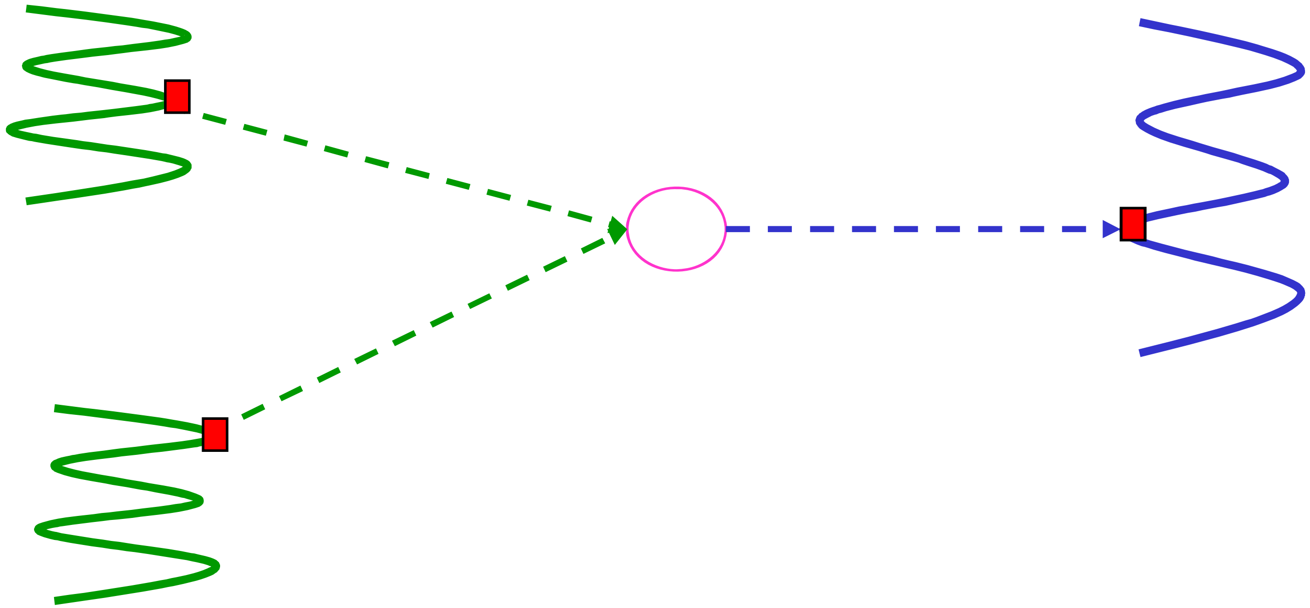


Semantic: The two blue threads can only continue
iff the green thread has passed a code section

Pattern: Many to Many (m:n)



Alternative Semantics for Signals



The blue thread can continue at WP when one of the two green threads has reached its SP

Additional Problem: How to buffer signals?

Buffering Signals

Buffering Signals

- Every incoming signal is buffered until a potential waiting thread consumes this signal
  - Pro: Reaction on each signal
  - Con: Deficient signaling source floods the system
- An incoming signal overwrites a previous one (e.g. a flag or a binary semaphore)
  - Pro: Reaction only on the newest signal
  - Con: Danger of lost signals

Kernel Signal Objects

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) $\Rightarrow$ 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

Semantics of a counting semaphore (for signaling):

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

Remark (Margo Seltzer, Harvard University, Cambridge & Boston, MA, USA): "A semaphore offers a simple and elegant mechanism for mutual exclusion and other things"

Counting Semaphores (1)

```
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}
      BLOCK(myself)
     fi
 v(S:semaphore)
     S.Count = S.Count + 1       {+ 1 pending signal}
     if S.Count <= 0 then
      UNBLOCK(delete first(S.QWT))
     fi
end
```

Examples of Signal Objects

Unix-Signal (Wikipedia)

A signal is a limited form of IPC used in Unix, Unix-like, and other POSIX-compliant operating systems.

Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred.

When a signal is sent to a process, the operating system interrupts the process' normal flow of execution.

Execution can be interrupted during any non-atomic instruction.

If process has previously registered a signal handler, that routine is executed.

Otherwise the default signal handler is executed.

Unix Signals

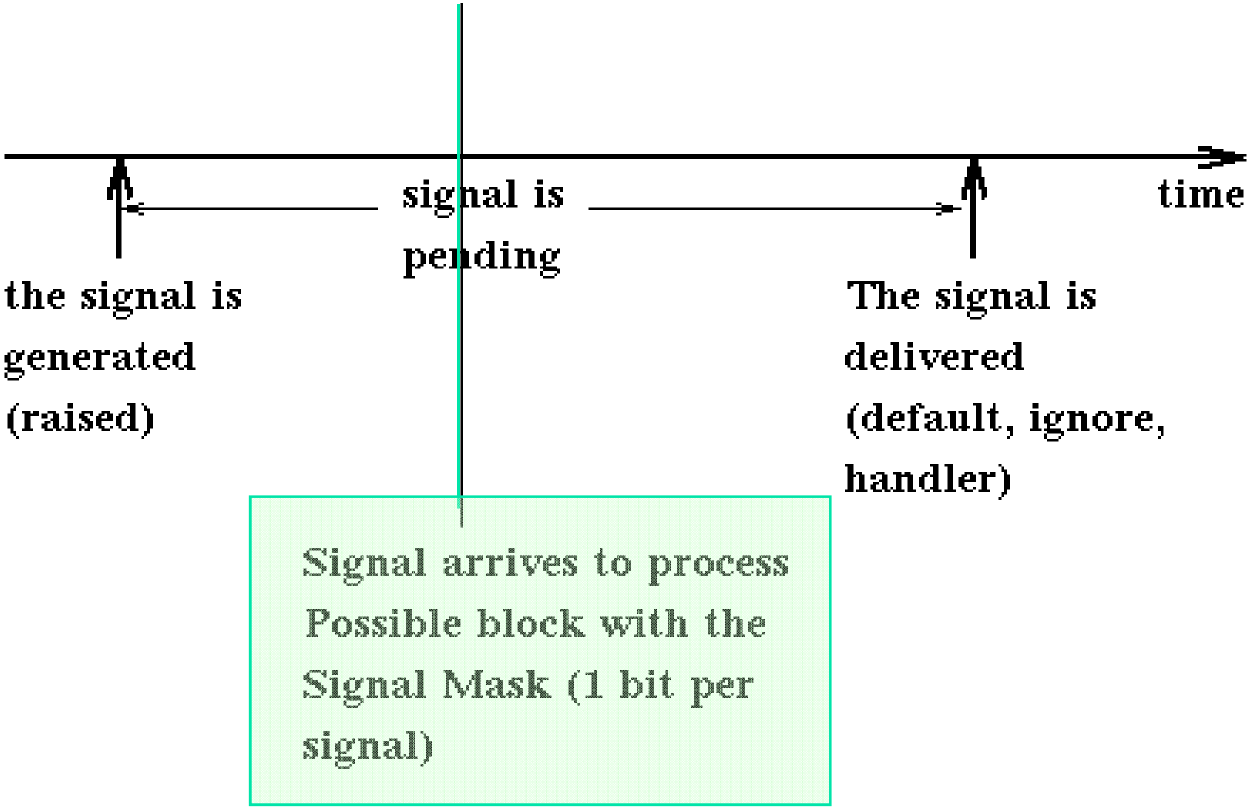
- Besides a terrible notation (e.g. kill = signal) $\exists$ no common semantics nor a widely accepted interface
- They are four different signal versions:
  - System-V unreliable
  - BSD
  - System-V reliable
  - POSIX
- Using Unix signals can lead to severe race conditions
- Programming is cumbersome

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

Unix Signals*

The following diagram describes how a Unix signal is raised, possibly it will be blocked before delivery, and then it is handled



*http://www.xs4all.nl/~evbergen/unix-signals.html

Unix Signals

- A Unix signal can be received
  - synchronously or
  - asynchronously
- Synchronous signals (typically sent to the same process)
  - Exception address violation
  - Exception division by zero
  - …
- Asynchronous signal (typically sent to another process)
  - <CTRL><C> = SIGINT ~ terminate process immediately
  - <CTRL><Z> = SIGTSTP ~ suspend process
  - ***Command*** kill -<signal> <PID>
  - ***System Call*** kill (see following slide)
  - Timer has expired

Using System Call **kill()**

```
#include <unistd.h> /* standard unix functions,
   like getpid() */
#include <sys/types.h> /* various type
   definitions, like pid_t */
#include <signal.h> /* signal name macros, and
   the kill() prototype */
/* first, find my own process ID */
   pid_t my_pid = getpid();
/* now that I got my PID, send myself STOP signal. */
   kill(my_pid, SIGSTOP);
```

Unix Signal Handlers

- Default handler
  - *Running in user or kernel mode?*
- User-defined handlers
  - Implement short signal handlers
  - Be careful when using system calls
  - Some Unix systems require another installing of the same signal handler if it should be used a second time

How to use Unix Signal handlers, see
http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html

Preview

Remark:

Kernel semaphore objects offer primitive, yet robust synchronization methods for processes and KLTs

⇒

However, semaphores also solve another class of coordination problems:

Mutual Exclusion