

# GCD for 3 (1/3)

- Consider the following concurrent program aimed to compute the greatest common divisor of three natural numbers: X, Y and Z, i.e. it should terminate in a state P:  $\{\text{gcd}(a,b,c) = \text{gcd}(X,Y,Z) \text{ and } (a=b=c)\}$ . Assume that the underlying hardware provides the normal LOAD and STORE machine instructions so that the program's each individual reference (read or write) to an integer can be considered atomic.

- 

```
int a = X, b = Y, c = Z; cont = true;
co  while (cont) if (a > b) a = a - b;    #process p1
//  while (cont) if (b > c) b = b - c;    #process p2
//  while (cont) if (c > a) c = c - a;    #process p3
//  while (cont) if ((a == b) and (b == c)) cont = false;
oc
```

- (a) Does the program meet the requirements of the At-Most-Once Property? Explain.
- (b) Give a proof in the form of a global invariant.
- (c) Does the program terminate always? Explain.
- (d) Assuming the program terminates, does it always satisfy the predicate P? Explain with a proof outline and an invariant. If your answer to is “NO”, explain with a trace.
- (e) Change the program if needed so that P holds always at the termination.

# GCD for 3 (2/3)

Answers:

- (a) No, because the assignment in each process, like  $a = a - b;$  in  $p1$  refers to variable  $b$  changed by another process  $p2$ , and also variable  $a$  read by another process  $p3$ .
- (b) This is a concurrent variant of the classic Euclidian gcd-algorithm. Its invariant  $I : \{\text{gcd}(a,b,c) = \text{gcd}(X,Y,Z) \text{ and } a>0, b>0, c>0\}$  holds trivially for the “sequential” case, where the body of each while loop, like  $\text{if } (a > b) \ a = a - b;$  in  $p1$  is executed as one atomic action. We may rewrite this fine as grained version using a local variable  $lb$  and breaking the assignment to two without changing it, so that all three parts of the body are now at-most-once and the whole satisfies the proof outline:

$$\begin{aligned} & \{I\} \\ & \text{if } (a > b) \ {I \text{ and } a>b\} \\ & \quad lb = b; \quad \{I \text{ and } a>lb \geq b \text{ and } (\text{gcd}(a,b,c)=\text{gcd}(a,b,c,lb))\} \\ & \quad a=a-lb; \quad \{I\} \end{aligned}$$

So  $I$  is still folds for this fine grained concurrent version.

- (c) Yes, if  $X, Y,$  and  $Z$  are natural numbers. In each step they are decreased until  $a = b = c$  holds. At the latest this will be happen when  $a = b = c = 1$  is reached, ie. if  $X, Y, Z$  are respective primes, ie.  $\text{gcd}(X, Y, Z) = 1$ .

# GCD for 3 (3/3)

(d) No. If the conditions  $(a == b)$  and  $(b == c)$  in  $p4$  are evaluated so that  $b$  is read twice to evaluate the two conditions the following trace scenario may occur:

$p4$ :  $\{a=6, b=6, c=3\}$  while (cont) if  $((a == b) \dots \{\text{locally in } p4: a=b=6\}$

$p2$ :  $\{a=6, b=6, c=3\}$  while (cont) if  $(b > c)$   $b = b - c;$  )  $\{\text{globally: } b=3\}$

$p4$ :  $\{a=6, b=3, c=3\} \dots (b == c)$  cont = false;  $\{\text{locally in } p4: b=c=3\}$

$p1, p2, p3, p4$ :  $\{\text{cont=false}\}$  program terminates  $\{a=6, b=3, c=3\}$ .

(e) If  $b$  is read only once to a local variable  $b1$  in  $p4$ :

```
while (cont) {  
    b1 = b;  
    if  $((a == b1)$  and  $(b1 == c)$ ) cont = false; #process  $p4$   
}
```

the interference above could not happen.

Proof: If  $p4$  assigns  $\text{cont} = \text{false}$ , there must be one common value  $V$  which has been read by  $p4$  at some point of time from  $a$  and  $c$  and  $b (=b1)$ . Suppose - without losing generality - that  $a$  was the first one to reach  $\{a=V\}$ , implying that both  $\{b \geq V=a\}$  and  $\{c \geq V=a\}$  must hold.

Clearly  $a$  can not be changed by  $p1$ . Supposing that  $b$  reaches next the state  $\{b=V\}$ , it must stay and like that, until  $\{c=V\}$ . So the if the program terminates,  $\{a=b=c=V\}$  holds. And because  $I$  holds,  $V = \text{gcd}(X, Y, Z)$ .

NOTE: You may try to write this without using  $p4$  and  $\text{cont}$ , but it would hard to scale it up to a larger problem, like 10 values. This is the sc. *distributed termination problem*.

# CS with Swap(x,y) (1/3)

- Suppose your machine has the following machine instruction:

```
Swap(var1, var2) :
```

```
< tmp = var1; var1 = var2; var2 =tmp;>
```

where tmp is a internal register of the processor.

- (a) Using Swap develop a spin-lock solution to the critical section problem for n processes in a multiprocessor. Do not worry about the eventual entry property. Describe clearly how your solution works and why it's correct. (b) Modify your answer to (a) so that it will also perform well in a multiprocessor system with system caches.

# CS with Swap(x,y) (2/3)

(a) Solution with a global invariant

$I: \{ \text{lock} + \text{Sum}(\text{for all } i: 0 < i \leq n: l_i) = 1 \}$ .

So in CS for  $p_i$   $l_i = 1$  holds, and it implies together with  $I$  that  $\text{lock} = 0$ , ie. no other process  $j$  can get in to its CS.

```
global int lock = 1;
local int li = 0; {I}                                #local for pi
while true {I} {
    while (li == 0) {I} swap(lock, li); {I}
    CS; {I}                                           # li = 1
    swap (lock, li); {I}                               # li = 0
    NCS; {I}
}
```

# CS with Swap(x,y) (3/3)

(b) This solution reduces the number of *write accesses* to the shared variable `lock` dramatically by trying to get it using `swap` only, when `lock` has been 1 just before it.

```
global int lock =1;
local int li = 0; {I}
while true {I} {
  while (li == 0) {
    while (lock = 0) wait(random(t));
                                     # positive waiting time
                                     # can used to reduce
                                     # the number reads

    swap (lock, li);
  }; {I}
  CS; {I}                            # li = 1
  swap (lock, li); {I}                # li = 0
  NCS; {I}
}
```

# Semaphore with INC and DEC (1/3)

- Consider the following proposal for a busy-wait and weakly fair implementation for semaphores for a multiprocessor with INC and DEC machine instructions. INC(x) atomically adds 1, and DEC(x) atomically subtracts 1 from integer variable x. Both INC(x) and DEC(x) return the new value of x.

```
/* P(sem) : */
while (DEC(s) < 0) {
    INC(s); #undo decrement
}

/* V(sem) : */
INC (s)
```

- (a) What is correct in it ?
- (b) What is not correct with the solution?
- (c) What is bad with it performance wise?
- (d) How would you correct it and improve its performance?
- (e) What would be the requirement for a strongly fair implementation?

# Semaphore with INC and DEC (2/3)

- (a) Safety:
  - denote with  $\#V(\text{sem})$ ,  $\#P(\text{sem})$  the number of *passed*  $V(\text{sem})$  and  $P(\text{sem})$  operations
  - $\text{sem} = \text{initial}(\text{sem}) + \#V(\text{sem}) - \#P(\text{sem})$
  - semaphore invariant:  $\text{sem} \geq 0$  is preserved by the implementation, so it's safe
- (b) Livelock:
  - $s = \text{sem} - \text{number of processes between: DEC}(s); \dots \text{INC}(s)$  in the while loop.
  - the following may hold for arbitrary long: (  $\text{sem} > 0$  and  $s < 0$  )
    - (construct a trace demonstrating this!)
  - so several processes wait spinning in the busy loop of  $P(\text{sem})$  while  $\text{sem} > 0$ .
  - this is a livelock, which is caused an obnoxious - but still possible!- scheduling
- (c) The spin loop causes memory read-write contention on  $s$ .

# Semaphore with INC and DEC (3/3)

- (d) To avoid problems of b) and c):
  - enter the try: "DEC(s) < 0" only when s>0 was true before it .
  - introduce extra "while (s≤0) skip;" before it.

```
/* P(s): */                               /* V(s): */
while (s≤0) skip;                           INC (s);
while (DEC(s) < 0) {
    INC(s); #undo decrement
    while (s≤0) skip;
}
```

- No livelock:
  - assume several processes are spinning in: while (s≤0) skip;
  - After a V(s), a bunch of these processes may get out from the spin, the first one them to execute the DEC(s) passes the P(s), and others will continue spinning after doing the corrective action INC(s).
  - So the system makes progress irrespective of the fairness of the scheduling
  - Note: the progress of an individual process is not guaranteed, i.e. the solution is only weakly fair.
  - Note: Memory contention happens only on read s.
- Strong fairness: A process waiting in a P(s) can't be overtaken indefinitely by others. This is NOT a strongly fair implementation.

# Philosophers with Gatekeeper (1/3)

A naïve solution to dining philosophers:

```
sem fork[5]= { 1, 1 , 1 , 1, 1};  
process Philosopher [i = 0 to 4] {  
  while (true) {  
    P(fork[i]); P(fork[(i+1)%5]);  
    eat;  
    V(fork[i]); V(fork[(i+1)%5]);  
    think;  
  }  
}
```

- (a) What is wrong with this? Demonstrate your claim with a trace.
- (b) Fix the solution based on the idea that the number of philosophers allowed to eat at the same time is limited. Use one or more additional semaphores for this.
- (c) Formulate carefully an argumentation that your solution satisfies the following correctness requirements: 1. Safety: When eating, the philosopher should have exclusive access to the two forks on each side. 2. No deadlock can occur, 3. No philosopher would starve, i.e. the solution is fair even if the semaphores are not strongly fair, like e.g. FIFO.

# Philosophers with Gatekeeper (2/3)

(a) Deadlock:

- All the philosophers start with  $P(\text{fork}[i])$ ;
- Circular wait: Everybody has one fork and waits for the other

(b) An extra semaphore  $d$  (doorman) to allow at most 4 philosophers to eat at the same time.

```
sem fork[5]= { 1, 1 , 1 , 1, 1};  
sem d = 4;  
process Philosopher [i = 0 to 4] {  
while (true) {  
    P(d) ;  
    P(fork[i]); P(fork[(i+1)%5]);  
    eat;  
    V(fork[i]); V(fork[(i+1)%5]);  
    V(d) ;  
    think;  
}}
```

- Safety:

$P(\text{fork}[i]) ; \dots \text{eat} ; \dots V(\text{fork}[i]) ;$  forms a critical section for  $\text{fork}[i]$

- No deadlock:

There is at least one process (= philosopher) for which also the other P-operation:  
 $P(\text{fork}[(i+1)\%5])$  will succeed.

# Philosophers with Gatekeeper (3/3)

- No starvation:
  - At most one philosopher is waiting on each semaphore P-operation at any time.
  - Assume the following “not so weak” fairness of the semaphore implementation:
    - V-operation on a semaphore will always free one of the possible processes waiting in the P-operation before any new process starting P-operation only after the V.
    - => no philosopher will get stuck in any of the P-operations, and thus would not starve.
- Notes
  - Any other solution satisfying the initial requirement and assuming a less fair semaphore implementation is also acceptable.
  - The philosophers should not wait for others except for the forks.
  - Try to solve the problem with tuplespace primitives (eg. Linda) without loops.

# One lane bridge (1/5)

- Cars coming from north and south have to cross a river along a very long and narrow one-lane bridge. Cars driving to the same direction may be on the bridge at the same time, while cars heading to opposite direction have to wait. Implement a monitor solution to the problem, where the cars are processes calling the `cross_from_North()` and `cross_from_South()` of the monitor `one_lane_bridge`. Solution must be symmetric. Specify the monitor invariant.
- a) Use the signal wait (SW) semantics and/or signal and continue (SC) semantic. Do not worry about fairness.
- b) Modify your solution to ensure fairness
- c) Modify your solutions for a) and b) for Java.

# One lane bridge (2/5)

a) Simple generic monitor. Does not guarantee fairness. Works for both SC and SW semantics  
Invariant: { { ns=0 or ns=0 } and (non (!empty(ns\_c) => sn > 0 ) and (!empty(sn\_c) => ns > 0)}

```
monitor One_lane_bridge {
    int ns =0;           //north-south cars on the bridge
    int sn =0;           //south-north cars on the bridge
    cond ns_c;           //condition to enter from north
    cond sn_c;           //condition to enter from south
    private procedure startNorth() {
        while (sn > 0) wait(ns_c);
        ns++
    }
    private procedure endSouth() {
        ns--;
        if (ns == 0) signal_all(sn_c); //signals possible waiting sn cars
    }
    public cross_from_North() { // this is needed to provide a simpler API
        startNorth();
        // north-south crossing operation is embedded here
        endSouth();
    }
}

... // the south-north direction is symmetric
```

# One lane bridge (3/5)

C) Java solution without fairness

```
class One_lane_bridge {
    private int ns =0          //north-south cars on the bridge
    private int sn =0          //south-north cars on the bridge
    private int wns =0         //waiting north-south cars
    private int wsn =0         //waiting south-north cars
    private char p= " "       //

    private synchronized void startNorth() {
        while (sn > 0){
            try {wait()};
            catch (interrupter execution ex) {return;}
        };
        ns++;
    }
    private synchronized void endSouth() {
        ns--;
        if (ns == 0)
            notifyAll(); //signals possible waiting sn cars
    }
    public void cross_from_North() {
        startNorth();
        // north-south crossing operation is embedded here
        endSouth();
    }
}
```

# One lane bridge (4/5)

b) Simple general monitor with fairness. Works for SW and SC semantics with FIFO queue. Idea: Stop the traffic if there are waiters in the opposite direction. Once the bridge becomes empty let all waiters in the opposite direction get in.

COMMENT: Wait if there is traffic in my direction and waiters in the opposite direction. Waiters are released from here when  $(sn == 0)$  and  $(ns == 0)$ . Because the first one will put  $(ns > 0)$  so the condition should not be tested again in a `while` loop so that all the cars waiting in the queue at the time of the `signal_all(ns_c)` will be able proceed. New cars entering the monitor may get stuck if `!empty(sn_c)` holds.

```
monitor One_lane_bridge {
    int ns =0;           //north-south cars on the bridge
    int sn =0;           //south-north cars on the brigde
    cond ns_c;           //condition to enter from north
    cond sn_c;           //condition to enter from south

    private procedure startNorth() {
        if (sn > 0) or (!empty(sn_c) and (ns>0)) wait; // READ COMMENT!!
        ns++;
    }
    private procedure endSouth() {
        ns--;
        if (ns == 0) signal_all(sn_c)); //signals possible waiting sn cars
    }
    public cross_from_North() {
        startNorth();
        // north-south crossing operation is embedded here
        endSouth();
    }
    ... // the south-north direction is symmetric
}
```

# One lane bridge (5/5)

C) Java solution with fairness

```
class One_lane_bridge {
    private int ns =0 //north-south cars on the bridge
    private int sn =0 //south-north cars on the brigde
    private int wns =0 //waiting north-south cars
    private int wsn =0 //waiting south-north cars
    private char p= " "

    private synchronized void startNorth() {
        while ((sn>0) or ((wsn>0) and (ns>0) and !(p=="n"))) {
            wns++;
            try {wait();
                catch (interrupter execution ex) {return;}
            wns--;
        };
        if (wns == 0) p = " ";
        ns++;
    }

    private synchronized void endSouth() {
        ns--;
        if (ns == 0) {
            p = "s";
            notifyAll(); //signals possible waiting sn cars
        }
    }

    public void cross_from_North() {
        startNorth();
        // north-south crossing operation is embedded here
        endSouth();
    }
}
```

# Sleeping barber (1/6)

- A barber has a small shop -where only one person can move at a time- with one barber chair and a waiting bench. When no customer is in the shop, the barber waits sleeping. When customer enters and finds the barber sleeping, he awakens the barber, sits in the chair and falls to sleep waiting for the barber to finish the haircut. If barber is already busy, the customer waits sleeping on the bench. After giving the haircut the barber opens the door for the customer to leave, awakens him and waits for the customer exit. After it he starts with a new customer, or if there is no one waiting he falls to sleep waiting for a new one to arrive. Solve the synchronization problem with a monitor `Barber_Shop` called by customers with procedure `get_haircut()`, and by the barber with procedure `get_next_customer()` to start with the next customer and procedure `finished_cut()` to finish with the current one.
- Solve the problem in general and a) and for signal-and-continue (SC) and b) for signal-and-wait (SW) semantics. c) Modify the solution for several barbers each having his own chair in the shop. d) To avoid the single monitor to become a bottleneck, keep the binding between the customer and barber in it, but delegate the rest of their mutual interaction to barber specific monitor `Barber[i]`, so that the call interfaces become:

```
customer_process[i] ..
    barber = Barber_shop.get_free_barber();
    Barber[barber].get_haircut(i);
    ..
barber_process[i] ..
    Barber_shop.get_next_customer(i);
    Barber[i].start.cut();
    Barber[i].finished_cut():
    ..
```

## Sleeping barber (2/6)

The basic solution for both SW and SC: Invariant:  $\{0 \leq \text{barber} + \text{chair} + \text{haircut} + \text{open} + \text{custexit} \leq 1\}$

1. Each wait is inside a while loop to check that the wait condition still holds (to work for SC semantics)
2. For each wait condition we define a state variable, and set it (=1) before the signal and reset it (=0) after the wait, so that no signal is lost and the corresponding wait would not cause a deadlock (SW-semantics):

```
monitor Barber_Shop {
    int barber=0, chair=0, open=0; haircut=0; custexit=0;
    cond barber_available;          # signaled when barber=1
    cond chair_occupied;           # signaled when chair=1
    cond door_open;                # signaled when open=1
    cond customer_left;           # signaled when custexit=1
    procedure get_haircut() {
        while (barber == 0) wait(barber_available); # barber=1
        barber= 0;
        chair= 1; signal(chair_occupied);
        while (open == 0) wait(door_open);          # open=1
        open=0;
        custexit =1; signal(customer_left)
    }
    procedure get_next_customer() {
        barber= 1; signal(barber_available);
        while (chair == 0) wait(chair_occupied); # chair=1
        chair=0;
        haircut =1;                                # hair cut starts
    }
    procedure finished_cut() {                # hair cut ends
        haircut= 0;
        open = 1; signal(door_open);
        while (custexit==0) wait (customer_left) # custexit=1
        custexit=0;
    }
}}
```

## Sleeping barber (3/6)

### a) Minimal solution with SC semantics

No while loops, i.e. no retesting for conditions: `chair_occupied`, `door_open`, `customer_left` is needed, because there is exactly one process waiting for them: One barber and one customer after (\*\*). So each signal on them will transfer control to the other process in the wait, but this happens only after the signaller itself has proceeded to its next possible wait inside the same monitor procedure, so each wait will start on them before the corresponding signal is sent.

```
monitor Barber_Shop{
    ...
    procedure get_haircut() {
        while (barber == 0) wait(barber_available);
        barber= 0; # (**)
        signal(chair_occupied); #1 sequence
        wait(door_open); #4
        signal(customer_left) #5
    }
    procedure get_next_customer() {
        barber= 1;
        signal(barber_available);
        wait(chair_occupied) #2
    }
    procedure finished_cut() {
        signal(door_open); #3
        wait (customer_left) #6
    }
}
```

## Sleeping barber (4/6)

### b) Minimal solution with SW-semantics

In case customer enters before the barber and stops at the `wait(barber_available)` the corresponding signal by the barber `signal(barber_available)` will cause the monitor to give the turn to the customer so that it will do the `signal(chair_occupied)` before the barber does the corresponding `wait(chair_occupied)` so we need the `chair` variable, not to lose this, and cause barber to wait in `wait` (=deadlock). Same holds for condition `customer_left`, but not for `door_open`, because the `signal(door_open)` is done in a different monitor procedure `finished_cut()`. So `wait(door_open)` will be entered by the customer before `signal(door_open)` is done by the barber.

```
monitor Barber_Shop {
    ...
    procedure get_haircut() {
        if (barber == 0) wait(barber_available);
        barber = 0;
        chair = 1; signal(chair_occupied);
        wait(door_open);
        custexit = 1; signal(customer_left)
    }
    procedure get_next_customer() {
        barber = 1; signal(barber_available);
        if (chair == 0) wait(chair_occupied);
        chair = chair - 1;
    }

    procedure finished_cut() {
        signal(door_open);
        if (custexit == 0) wait (customer_left)
        custexit = 0;
    }
}
```

## Sleeping barber (5/6)

c) SC monitor for several barbers. We need to identify the barbers so that the customer knows which barber is serving him. An entering barber puts his ID in a queue data structure: `put (free_barbers, BarberID)`; and starts to wait on his own: `wait (chair_occupied[BarberID])` for the customer, who will get the ID by: `BarberID = get (free_barbers)`; and do the signalling: `signal (chair_occupied[BarberID])`.

```
monitor Barber_Shop                                     # for n barbers
    int queue free_barbers;                             # int queue of process id's for free
    barbers
    cond barber_available;                              # signaled when barber>0
    cond chair_occupied[n];                             # signaled when chair[]>0
    cond door_open[n] ;                                # signaled when open[]>0
    cond customer_left[n];                             # signaled when open[]==0

    procedure get_haircut {
        int BarberID;
        while empty (free_barbers) wait (barber_available);
        BarberID = get (free_barbers);                  # binding the barber to the
        customer
        signal (chair_occupied[BarberID]);
        wait (door_open[BarberID]);
        signal (customer_left[BarberID])
    }

    procedure get_next_customer (int BarberID) {
        put (free_barbers, BarberID);
        signal (barber_available);
        wait (chair_occupied[BarberID]);
    }

    procedure finished_cut (int BarberID) {
        signal (door_open[BarberID]);
        wait (customer_left [BarberID])
    }
}
```

## Sleeping barber (6/6)

### d) 1. Monitor to make 1-to-1 bindings between babers and customers

```
monitor Barber_Shop {
    int queue free_barbers;           # int queue of process id's for free barbers
    cond barber_available;           # signaled when barber>0
    procedure get_free_barber() returns int { #customer wants a free barber
        int BarberID;
        while empty(free_barbers) wait(barber_available);
        BarberID= get(free_barbers);    # binding done here
        return BarberID;
    }
    procedure get_next_customer(int BarberID){ #barber wants a customer
        put (free_barbers,BarberID);
        signal(barber_available);
    }
}
```

### 2. A monitor to synchronize the baber and customer for haircut

```
monitor Barber[n] {
    int chair =0;
    cond chair_occupied;             # signaled when chair >0
    cond door_open;                  # signaled when open >0
    cond customer_left;              # signaled when open ==0
    procedure get_haircut{           # customer sits in the chair of the barber
        chair = chair + 1;           # assigned to him and starts to wait
        signal(chair_occupied);
        wait(door_open);
        signal(customer_left)
    }
    procedure start_cut(){           #barber waits for the customer to sit in oder
        while (chair== 0) wait(chair_occupied); # to start the haircut
        chair = chair-1;
    }
    procedure finished_cut(){        # barbers is done and wakeups the customer
        signal(door_open);
        wait(customer_left)
    }
}
```