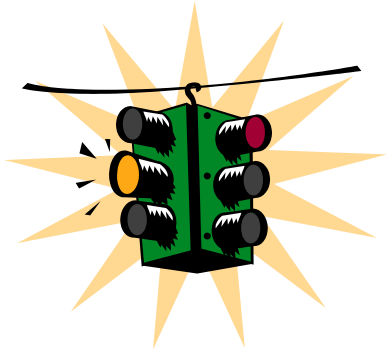
Module 5 - Process Synchronization
(or threads, or children or tasks)

Chapter 6 (Silberchatz)



Problems with concurrency = parallelism

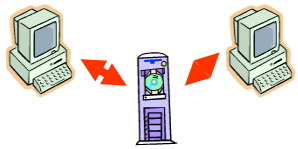
- **Concurrent threads must sometimes share data (files or common memory) and resources**
  - We are therefore talking about *cooperative tasks*
- **If access is not controlled, the result of program execution may depend on interleaving order of the execution of the instructions (*non-determinism*).**
- **A program may give different and sometimes undesirable results from time to time**

An example

- Two threads execute this same procedure and share the same database
- They can be interrupted anywhere
- The result of concurrent execution of P1 and P2 depends on the order of their *interlacing*

```
Mr. X asks for a
plane
reservation

Database says
seat A is
available

Seat A is
assigned to X
and marked
occupied
```

Overview of a possible execution



P1

P2

Interruption or delay

Mr. Leblanc requests a plane reservation

Mr. Guy requests a plane reservation

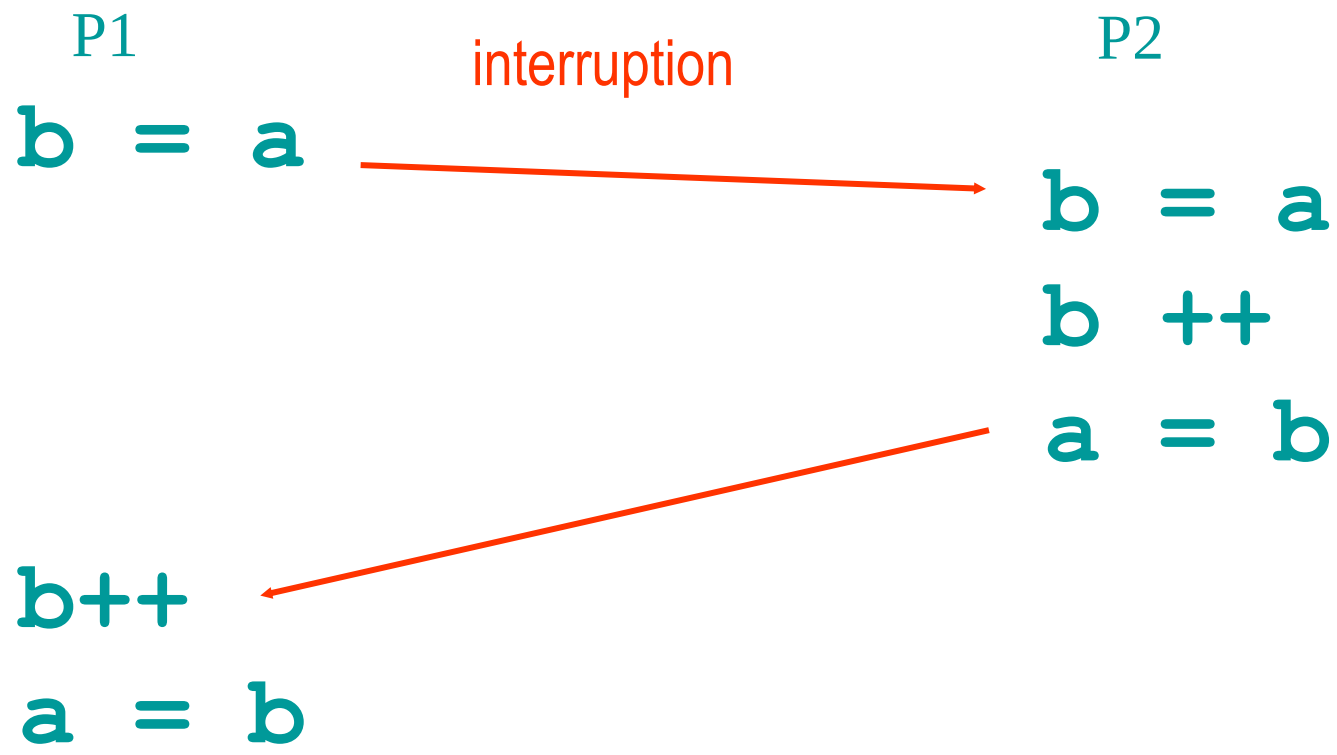
Database says 30A seat is available

Database says 30A seat is available

Seat 30A is assigned to Leblanc and marked occupied

Seat 30A is assigned to Guy and marked occupied

Two operations in parallel on a shared var a (b is private at each process)



Suppose a is 0 at the start

P1 is working on old value of a so the end result will be a = 1.

Will be a = 2 if the two tasks are executed one after the other

If a was saved when P1 is interrupted, it could not be shared with P2 (there would be two a while we want only one)

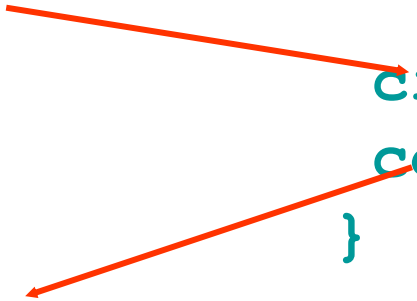
3rd example

Thread P1

```
static char a;

void echo ()
{
 cin >> a;


 cost << a;
}
```

Thread P2

```
static char a;

void echo ()
{

 cin >> a;
 cost << a;
 }
```



If the var a is shared, the first a is deleted
If it is private, the display order is reversed

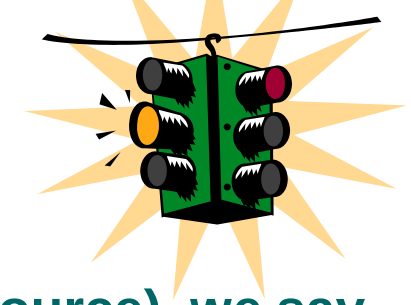
Other examples

- **Threads that work simultaneously on a matrix, eg. one to update it, the other to extract statistics from it**
- **Problem that affects the program of *bounded buffer*, v. textbook**
- **When several threads are executing in parallel, we cannot make any assumptions about the execution speed of the threads, nor their interleaving.**
  - May be different each time the program is run

Critical Section

- **Part of a program whose execution must not *intertwine* with other programs**
- **Once a task enters it, it must be allowed to complete this section without allowing other tasks to play on the same data.**

The problem of the critical section



- When a thread handles a shared data (or resource), we say that it is in a critical section (CS) (associated with this data)
- The problem of the critical section is to find an algorithm of *mutual exclusion* of threads in running their CSs so that the result of their actions *do not depend* of *interleaving order* their execution (with one or more processors)
- The execution of critical sections must be mutually exclusive: at any moment, **a single** thread can execute an CS for a given var (even when there are multiple processors)
- This can be achieved by placing special instructions in the entry and exit sections
- For simplicity, henceforth we assume that there is only one CS in a program.

Program structure

- Each thread must therefore request permission before entering a critical section (CS)
- The section of code that performs this query is the entry section
- The critical section is normally followed by a exit section
- The code that remains is the remaining section (RS): non-critical

```
repeat
 entry section
 critical section
 exit section
 remaining section
forever
```

Application

Mr. X asks for a plane reservation

Entry section

Critical section

Database says seat A is available

Seat A is assigned to X and marked occupied

Exit section

Criteria necessary for valid solutions

- **Mutual exclusion:**
  - At any time, at most one thread can be in a critical section (CS) for a given variable
- **Progress:**
  - no deadlock (Chap 7)
  - if a thread requests to enter a critical section at a time when no other thread requests it, it should be able to enter it
  - No interference:
    - If a thread stops in its remaining section, this should not affect other threads
  - But we assume that a thread which enters a critical section will leave it.
- **Bounded waiting:**
  - no thread eternally prevented from reaching its CS (no starvation)

Types of solutions

- Software solutions
  - algorithms who's correctness does not rely on any other assumptions
  - Peterson's algorithm
- Hardware solutions
  - rely on some special machine instructions
  - testAndSet, xchng
- OS provided solutions
  - higher level primitives (implemented usually using the hw solutions) provided for convenience by the OS/library/language
  - semaphores, monitors

- **All solutions are based on atomic access to main memory: memory at a specific address can only be affected by one instruction at a time, an thus one thread/process at a time.**
- **More generally, all solutions are based on the existence of atomic instructions, that operate as basic CSs.**

Atomicity = indivisibility

Software solutions

(not practical, but interesting to understand the pb)

- **We first consider 2 threads**
  - Algorithms 1 and 2 are not valid
    - Show the difficulty of the problem
  - Algorithm 3 is valid (Peterson's algorithm)

- **Notation**
  - Let's start with 2 threads: T0 and T1
  - When we discuss the task Ti, Tj will always denote the other task (i! = j)

Algorithm 1: threads give each other the turn

- The shared variable turn is initialized to 0 or 1
- The CS of Ti is performed iff turn = i
- Ti is busy waiting if Tj is in CS.
- Works for mutual exclusion!
- No starvation (only 1 thread in turn according to turn).
- But the criterion of progress is not satisfied because the execution of the CSs must strictly alternate

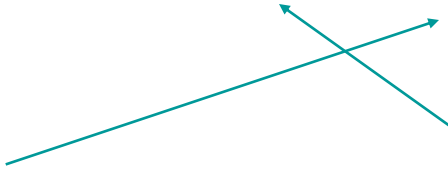
```
Thread Ti:
repeat
  while(turn!= i) {};
  CS
  turn = j;
  RS
forever
```

Do nothing

Ex 1: T0 has a long RS and T1 has a short RS. If turn == 0, T0 enters its CS and then its RS (turn == 1). T1 enters its CS and then its RS (turn == 0), and tries to enter its CS: refused! it must wait for T0 to give it the turn.

initialization of turn to 0 or 1

```
Thread T0:                    Thread T1:
repeat                        repeat
 while (turn!= 0){};           while(turn!=1){};
 CS                            CS
 turn = 1;                     turn = 0;
 RS                            RS
forever                       forever
```



Algorithm 1 overview

Ex 2: Generalization to n threads: each time, before a thread can enter its critical section, it must wait until all the others have had this chance!

Algorithm 2 or the excess of courtesy ...

- A Boolean variable per Thread: flag [0] and flag [1]
- Ti signals that he wants to execute his CS by: flag [i] = true
- But he does not enter if the other is also interested!
- Mutual exclusion ok
- Progress ok
- Absence of famine not satisfied:
- Consider the sequence:
  - T0: flag [0] = true
  - T1: flag [1] = true
    - Each thread will wait indefinitely to execute its CS: we have a *starvation*

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

do nothing


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

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

Algorithm 2 overview

T0: flag [0] = true
T1: flag [1] = true
deadlock!

Algorithm 3

So, neither Algorithm 1, nor Algorithm 2 work. What do we do?

Use the ideas from both of them!

- Use the `flag i` to indicate willingness to enter CS
- But use `turn` to let the other task enter the CS

```
Task T0:
while(true)
{
   flag[0] = true;
     // T0 wants in
   turn = 1;
    // T0 gives a chance to T1
   while
    (flag[1]==true&&turn==1){}
      Critical Section
   flag[0] = false;
    // T0 wants out
      Remainder Section
}
```

```
Task T1:
while(true)
{
   flag[1] = true;
     // T1 wants in
   turn = 0;
    // T1 gives a chance to T0
   while
    (flag[0]==true&&turn==0){}
      Critical section
   flag[1] = false;
    // T1 wants out
      Remainder Section
}
```

Algorithm 3

Lets see if it works …

- **Does it ensure mutual exclusion?**
  - Yes, a task enters CS only if it is its turn when both want in.
- **Does it satisfy bounded waiting?**
  - Yes, when a task is in its RS, the other task can enter its CS (flag of other task is false).
- **Ok, that was the easy part, lets go the crucial question:**
- **Does it satisfy the progress requirement?**
  - A task will enter CS if and only if it is its turn OR the other task does not want to enter CS
  - OK, so if the other task does not want to enter, I can go in, great
  - But that was the easy part. What if both want to enter?

Algorithm 3

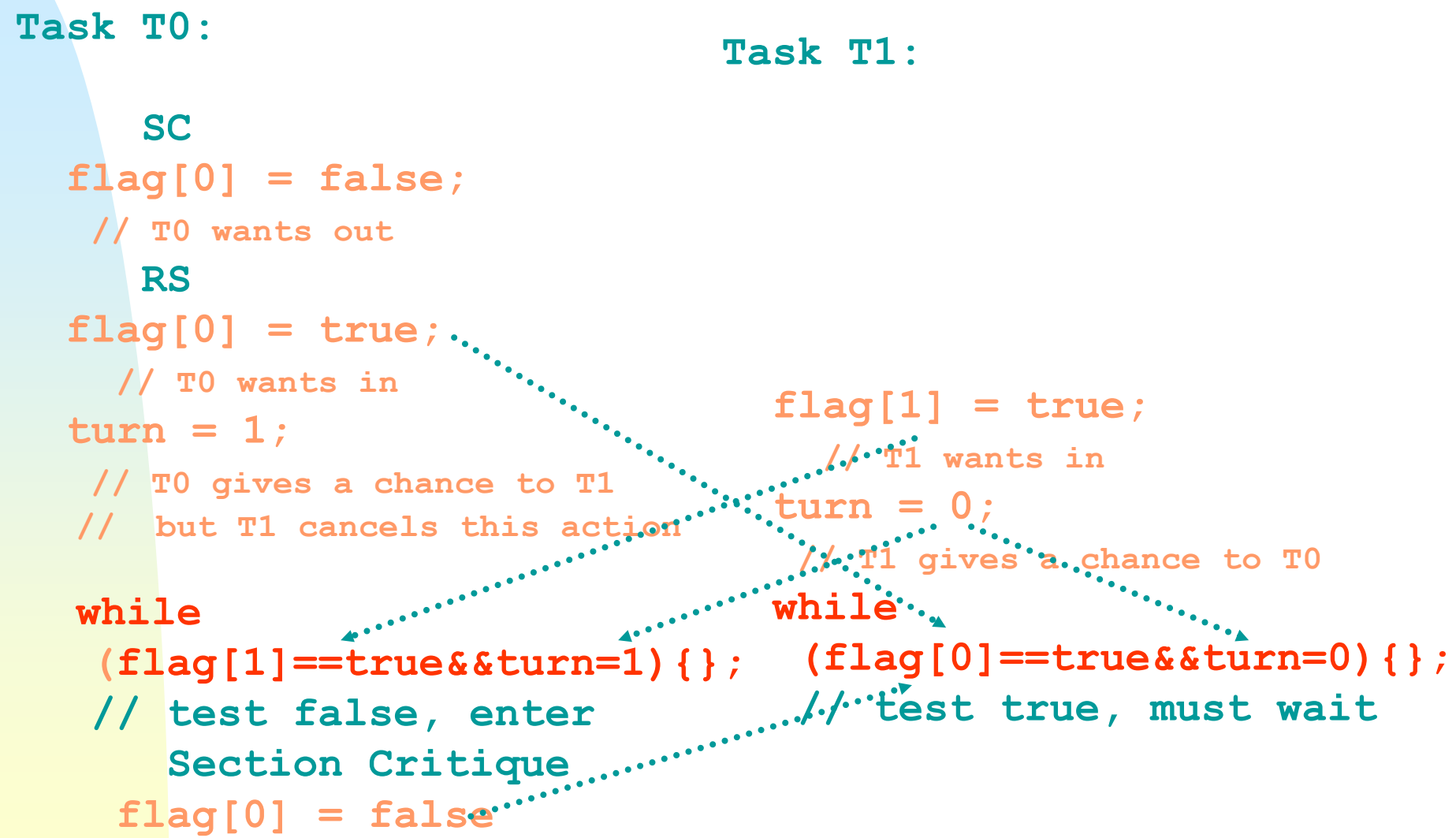
Discussing progress requirement for Algorithm 3…

What happens if both tasks want to enter?

- Both will set their respective flags to true
- Both will set `turn` to the opposite value
  - Wait! Only one of them (the second one, for example T1) will really succeed in this, as `turn` can take only one value
- This means T0 will enter the CS!
- And after T0 exits it, it will set its flag to false and T1 will enter CS.

- **Nonsense! There must be a way they block each other. Perhaps they start staggered…**

Both tasks want to enter!



Algorithm 3

```
Task Ti:
while(true)
{
   flag[i]=true;  // I want in
   turn=j;        // but I let the other in
   while (flag[j]==true && turn==j){} ;
      Critical section
   flag[i]=false; // I no longer want in
      Remainder section
}
```

- **Let's construct a situation where both tasks are blocked…**
- **So, lets assume T0 is blocked.**
- **The only possible place is in the while loop, therefore flag[1] is true and turn = 1.**
- **However, if turn=1, then T1 cannot be blocked.**
- **It might actually work!**
- It indeed works, the algorithm is called Peterson's algorithm

A few additional ideas

- **A solution to the Critical Section Problem (CSP) is robust relative to tasks failing in the RS.**
  - But, if a task fails in the CS, the other task shall be blocked.
- **The Peterson algorithm can be generalised to more than 2 tasks**
  - But, there exists other more elegant algorithms – the banker's algorithm
- **For tasks with shared variable to work properly, all threads involved must use the same synchronization algorithm**
  - A common protocol

Critique of software solutions

- **Difficult to program! And to understand!**
  - The solutions that we will see from now on are all based on the existence of specialized instructions, which facilitate the work.

- **Threads that require entry into their CS are busy waiting; thus consuming CPU time**
  - For long critical sections, it would be better to block the threads/processes that must wait ...

Hardware solutions: disabling interrupts

- On a uniprocessor: mutual exclusion is preserved but the efficiency deteriorates: when in CS it is impossible to interlace execution with other threads in an RS
- Loss of interruptions
- On a multiprocessor: mutual exclusion is not preserved
- A solution that is generally not acceptable

```
Process Pi:
repeat
  disable interrupt
  critical section
  enable interrupt
  remaining section
forever
```

Hardware solutions: specialized machine instructions

- **Normal: While one thread or process is accessing a memory address, no other can access the same address at the same time**
- **Extension: machine instructions executing many actions (e.g. read and write) on the same memory slot atomically (indivisible)**
- **An atomic instruction can only be executed by one thread at a time (even in the presence of several processors)**

The test-and-set statement

- A C ++ version of test-and-set:

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


Atomic Instruction!

- An algorithm using testset for Mutual Exclusion:
- Shared variable b is initialized to 0
- It is the 1st Pi which sets b to 1 which enters CS

```
Pi task:
 while testset (b) == false {};
 CS // enter when true
 b = 0;
 RS
```

The test-and-set (cont.) Statement

- Mutual exclusion is ensured: if Ti enters CS, the other Tj is busy waiting
- Problem: still uses busy waiting
- Can easily provide mutual exclusion but requires more complex algorithms to satisfy other requirements of the critical section problem
- When Ti leaves CS, the selection of Tj which will enter CS is arbitrary: no limit on waiting: Possibility of starvation

'Exchange' instruction

- Some CPUs (eg x86 family) offer an xchg (a, b) instruction which interchanges the contents of a and b *atomically*.
- But xchg (a, b) suffers from the same shortcomings as test-and-set

Using xchg for mutual exclusion (Stallings)

- **The shared variable b is initialized to 0**
- **Each Ti owns a *local variable k***
- **The Ti that can enter its CS is the one that finds b=0**
- **The Ti excludes all others by assigning 1 to b**
  - When the CS is occupied, both k and b will have the value 1 in tasks trying to enter its CS.
  - But k is 0 in the task that has entered its CS.

usage:

```
Task Ti:
while
{
   k = 1
   while k!=0 xchg(k,b);
   Critical Section
   xchg(k,b);
   Remainder Section
}
```

Solutions based on instructions provided by the OS (system calls)

- **The solutions seen so far are difficult to program and lead to bad code.**
- **We would also like it to be easier to avoid common mistakes, like deadlocks, starvation, etc.**
  - Need for higher level instruction
- **The methods that we will see from now on use powerful instructions, which are implemented by calls to the OS (system calls)**

Semaphores

- **A semaphore S is an integer which, except for Initialization, is accessible only by these 2 operations which are atomic and mutually exclusive:**
  - wait (S)
  - signal (S)
- **It is shared between all the processes that are interested in the same critical section**
- **The semaphores will be presented in two stages:**
  - semaphores which are busy waiting
  - semaphores that use waiting queues
- **A distinction is also made between counter and binary semaphores, but the latter are less powerful (see book).**

Spinlocks Unix: Semaphores busy waiting

(busy waiting)

- The easiest way to set up semaphores.
- Useful for situations where the wait is short, or there are a lot of CPUs
- S is an integer initialized to a positive value, so that a first thread can enter the CS
- When S> 0, up to n threads can enter
- When S <= 0, you have to wait for S + 1 signals (other threads) to enter

```
wait (S):
while S <= 0 {};
 S--;
```

Wait if no. threads that can enter = 0 or negative

```
signal (S):
 S ++;
```

Increase the number of threads that can enter by 1

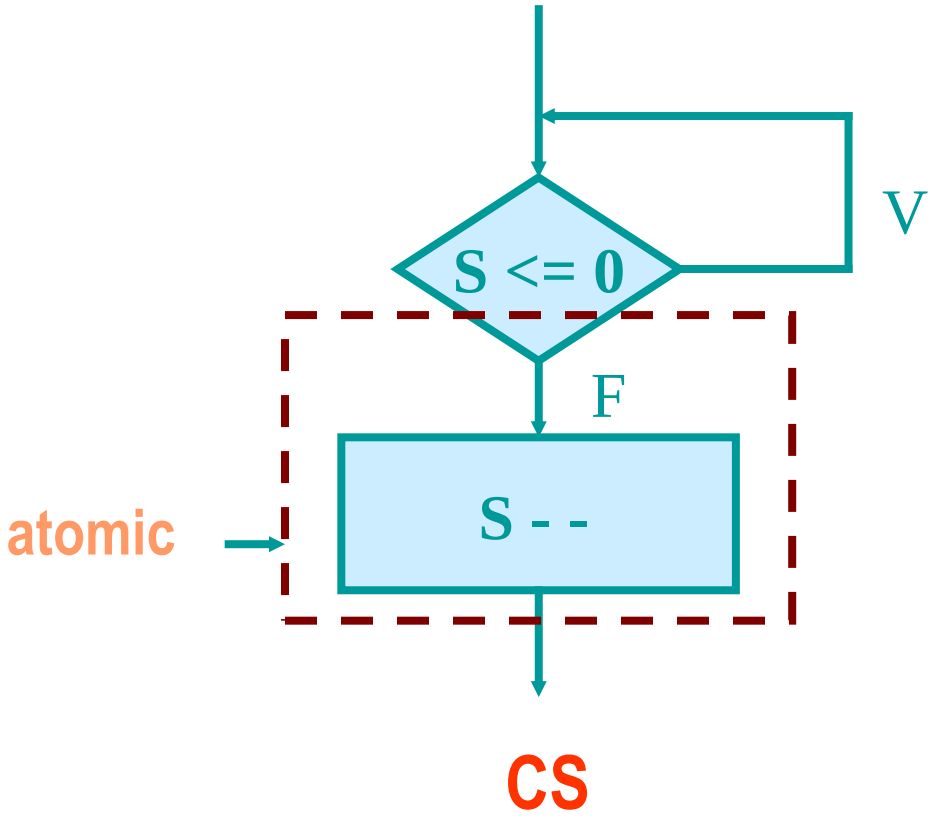
Atomicity

Wait: The test-decrement sequence is atomic, but not the loop!

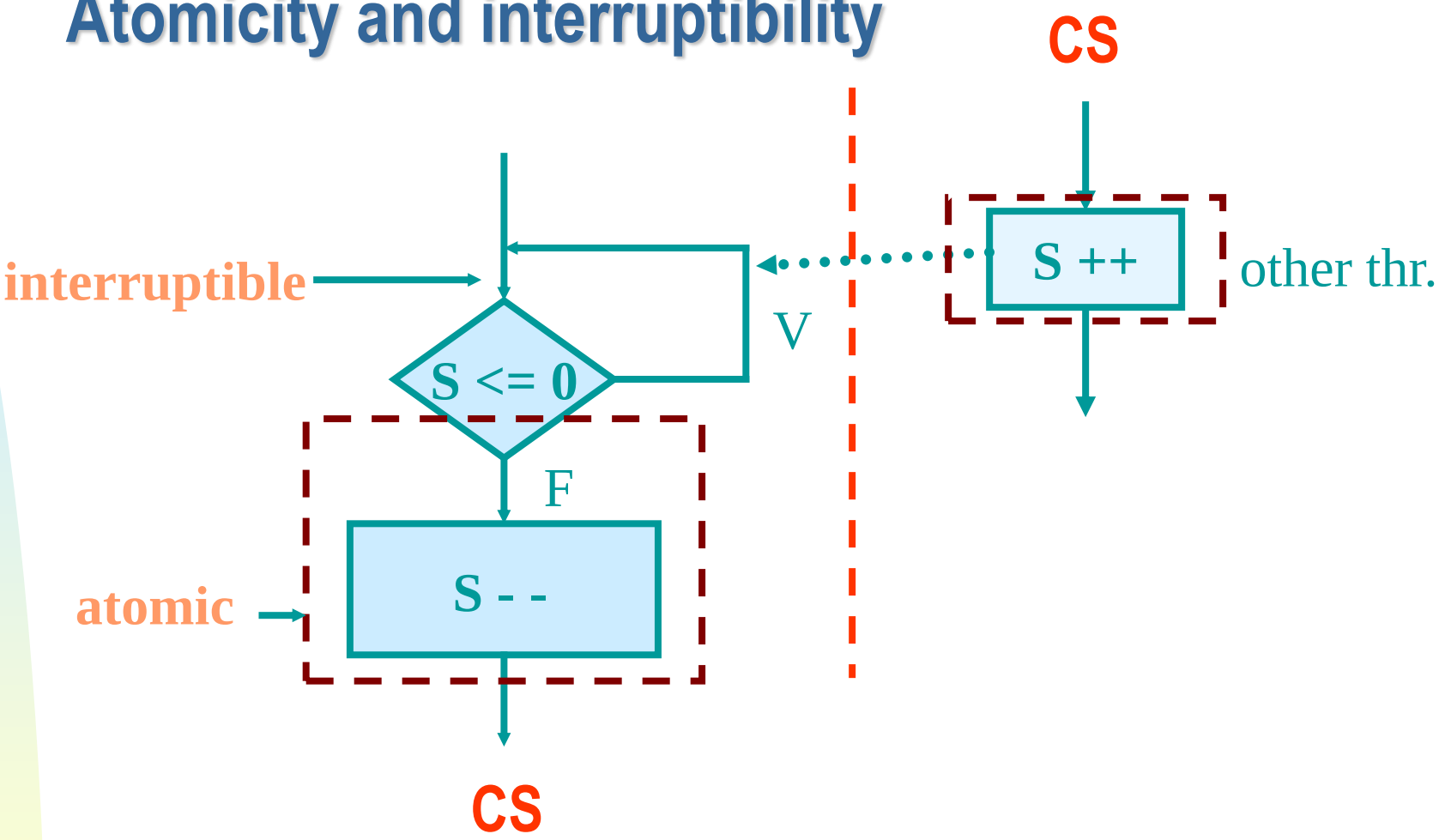
Signal is atomic.

Reminder: atomic sections cannot be executed simultaneously by different threads

(this can be achieved using one of the previous mechanisms)



Atomicity and interruptibility



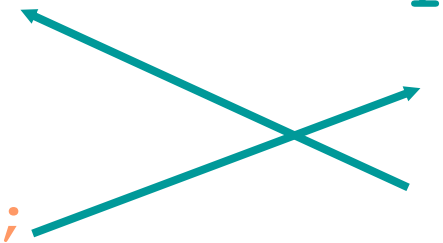
The loop is not atomic to allow another thread to interrupt the wait coming out of the CS

Use of semaphores for critical sections

- For n threads
- Initialize S to 1
- Then only 1 thread can be in its CS
- To allow k threads to execute CS, initialize S to k

```
Thread Ti:
repeat
 wait (S);
 CS
 signal (S);
 RS
forever
```

```
Initializes S to >= 1

Thread T1:                    Thread T2:
  repeat                      repeat
 wait (S);                     wait (S);
    CS                          CS
signal (S);                   signal (S);
    RS                          RS
  forever                     forever
```



Semaphores: global view

Can be easily generalized to more. threads

Using semaphores for thread synchronization

- **We have 2 threads: T1 and T2**
- **S1 statement in T1 must be executed before S2 statement in T2**
- **Define a semaphore S**
- **Initialize S to 0**
- **Correct synchronization when T1 contains:**
  - S1;
  - signal (S);
- **and that T2 contains:**
  - wait (S);
  - S2;

Deadlock and starvation with semaphores

- **Starvation: a thread can never manage to execute because it never tests the semaphore at the right time**
- **Deadlock: Suppose S and Q initialized to 1**

T0

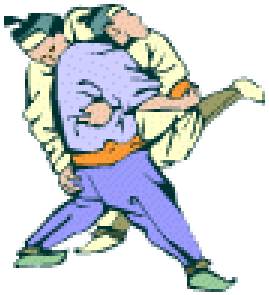
wait (S)

wait (Q)

T1

wait (Q)

wait (S)



Semaphores: observations

```
wait (S):
while S <= 0 {};
 S--;
```

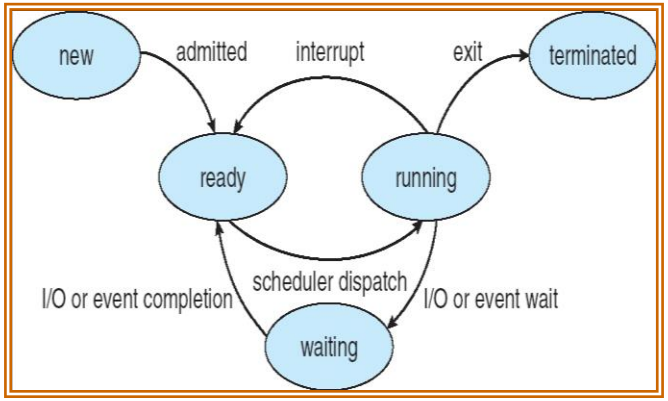
- **When S> = 0:**
  - The number of threads that can run wait (S) without becoming blocked = S
    - S threads can enter the CS
    - note power compared to mechanisms already seen
    - in solutions where S can be >1 it will be necessary to have a 2nd sem. to let them enter one at a time (excl. mutual)
- **When S becomes> 1, the thread that enters the CS first is the first to test S (random choice)**
  - this will no longer be true in the next solution
- **When S <0: the number of threads waiting on S is = | S | - Not applicable for semaphores busy waiting**

How to avoid busy waiting and random choice in semaphores

- When a thread has to wait for a semaphore to become greater than 0, it is put into a queue of threads that wait on the same semaphore.
- The queues can be (FIFO), with priorities, etc. The OS controls the order in which threads enter their CS.
- *wait* and *signal* are calls to the OS like calls to I / O operations.
- There is a queue for each semaphore just as there is a queue for each I/O unit.

Semaphores without busy waiting

- **A semaphore S becomes a data structure:**
  - A value
  - A waiting list L
- **A thread having to wait for a semaphore S, is blocked and added the waiting line S.L semaphore (see blocked state = waiting chap 3).**



- **signal (S) removes (according to a fair policy, ex: FIFO) a thread of S.L and places it on the list of ready threads.**

Implementation

(the boxes represent non-interruptible sequences)

```
wait(S):        S. value -;
                if S. value <0
                { // CS busy
                   add this thread to S.L; block // thread put in wait state
                }

signal(S):      S. value ++;
                if S. value ≤ 0 {{ // threads are waiting
                   remove a process P from SL;
                   wakeup(P)    // chosen thread becomes ready
                }
```

S.value must be initialized to a non-negative value (application dependent, see examples)

Wait and signal themselves contain CSs!

- The operations *wait* and *signal* must be executed atomically (only one thr. at a time)
- In a system with only 1 CPU, this can be achieved by disabling interrupts when a thread is performing these operations.
- Normally, we have to use one of the mechanisms seen before (special instructions, Peterson's algorithm, etc.)
- The busy wait in this case will not be too expensive because *wait* and *signal* are brief

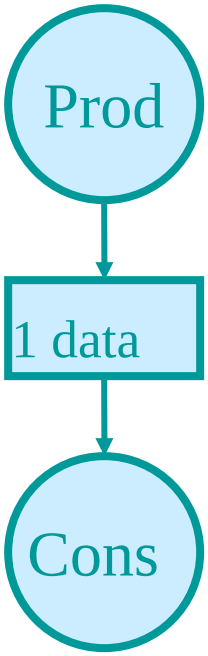
Classic synchronization problems

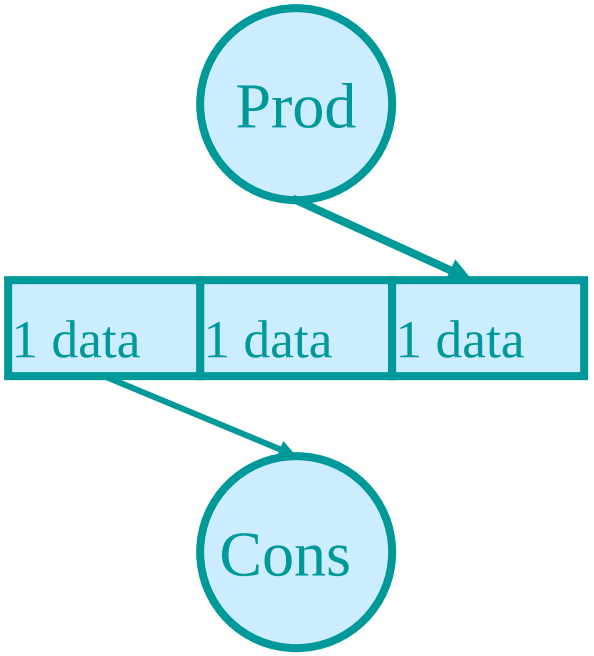
- **Bounded buffer (producer-consumer)**
- **Writers - Readers**
- **The philosophers eating**

The producer - consumer pb

- **A classic problem in the study of communicating threads**
  - a thread *producer* produces data (e.g. records in a file) for a thread *consumer*

Communication buffers

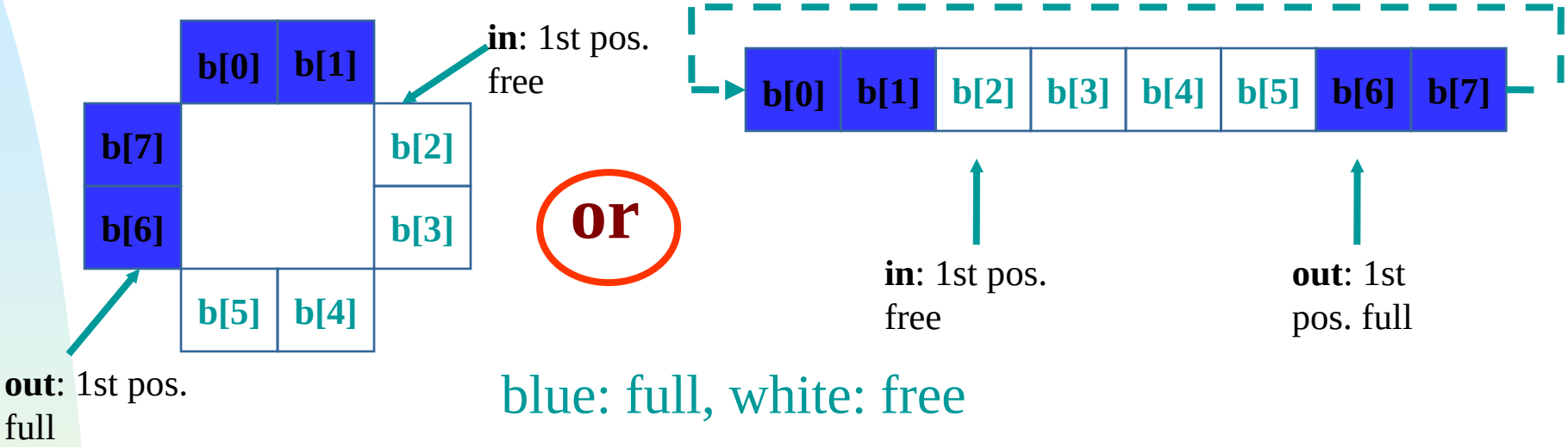




If the buffer is of length 1, the producer and consumer must necessarily go at the same speed

Longer length buffers allow some independence. Eg. on the right the consumer was slower

The bounded buffer (bounded buffer)
a fundamental data structure in OS



blue: full, white: free

The bounded buffer is in the memory shared between consumer and user

Pb of sync between threads for the bounded buffer

- Since the prod and the consumer are independent threads, problems could occur allowing concurrent access to the buffer
- Semaphores can solve this problem

Semaphores: reminder.

- **Let S be a semaphore on a CS**
  - it is associated with a queue
  - S positive: S threads can enter CS
  - S zero: no thread can enter, no thread waiting
  - S negative: | S | thread in queue
- **Wait (S): S - -**
  - if after S> = 0, thread can enter CS
  - if S <0, thread is queued
- **Signal (S): S ++**
  - if after S <= 0, there were threads waiting, and a thread is awakened
- **Indivisibility = atomicity of these ops**

Solution with semaphores

- **A semaphore S for mutual exclusion on access to the buffer**
  - The following semaphores do not do Mutual Exclusion
- **A semaphore NOT to synchronize producer and consumer on the number of consumable items in the buffer**
- **A semaphore E to synchronize producer and consumer on the number of free slots**

P / C solution: bounded circular buffer of dimension k

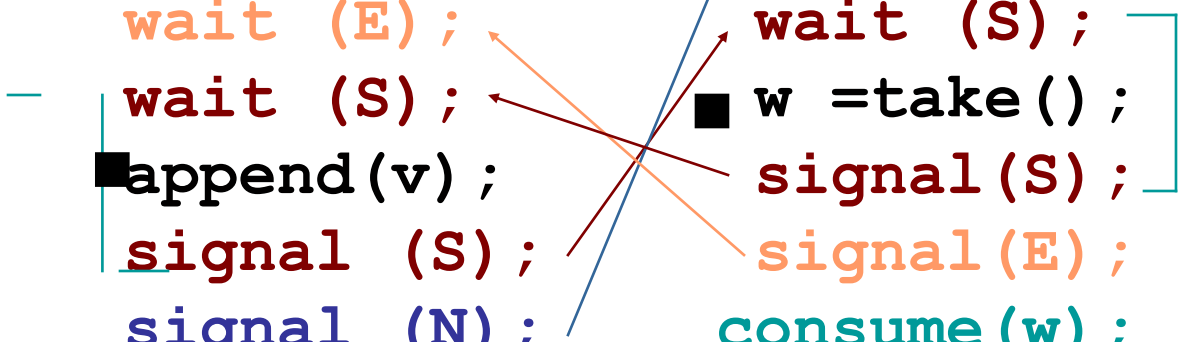
```
Initialization: S.count = 1; // mut. excl.
                N.count = 0; // full slots
                E.count = k; // empty slots
```

```
append (v):
 b[in] = v;
 In++ mod k;
```

```
take ():
 w = b[out];
 Out++ mod k;
 return w;
```

```
Producer:
repeat
 produce v;
 wait (E);
 wait (S);
 append(v);
 signal (S);
 signal (N);
forever
```

```
Consumer:
repeat
 wait (N);
 wait (S);
 w =take();
 signal(S);
 signal(E);
consume(w);
forever
```



- Critical sections

Important points to study

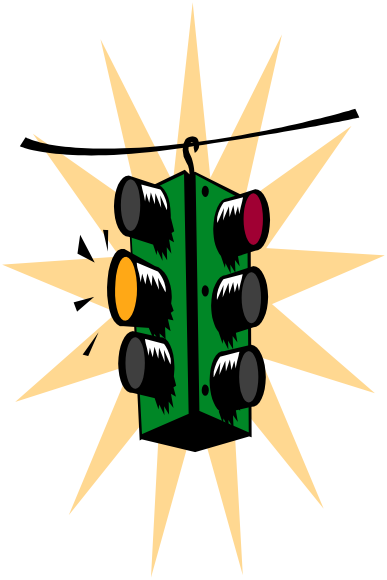
- **possible problems by interchanging the instructions on the semaphores**
  - or by changing their initialization
- **Generalization in case of more. prods and cons**

Important concepts in this part of Chap 6

- The problem of the critical section
- Interlacing and atomicity
- Starvation and deadlock issues
- Software solutions
- Hardware instructions
- Busy or queued semaphores
- How the different solutions work
- The example of the bounded buffer

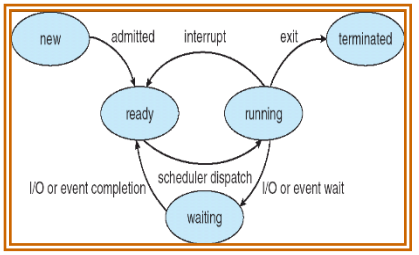
Some examples

- Classic synchronization problems
- Readers - Writers
- The philosophers eating
- Monitors



Semaphores: reminder

(the boxes represent non-interruptible sequences)



***wait*(*S*):**

```
S. value -;
if S. value <0 // CS busy
{
   add this thread to S.L; block // thread put in wait state
}
```

***signal*(*S*):**

```
S. value ++;
if S. value ≤ 0 {{ // threads are waiting
   remove a process P from SL;
   wakeup(P)    // chosen thread becomes ready
}
```

S.value must be initialized to a non-negative value (application dependent, see examples)

Semaphores: reminder.

- **Let S be a semaphore on a CS**
  - it is associated with a queue
  - S positive: S thread can enter CS
  - S zero: no thread can enter, no thread waiting
  - S negative: | S | thread in queue
- **Wait (S): S - -**
  - if after S> = 0, thread can enter CS
  - if S <0, thread is queued
- **Signal (S): S ++**
  - if after S <= 0, there were threads waiting, and a thread is transferred to a ready queue
  - Indivisibility = atomicity of wait and signal

Problem of readers - writers

- **Multiple threads can access a database**
  - To read or write there
- **Writers should be in sync with each other and with readers**
  - a thread must be prevented from reading while writing
  - two Writers must be prevented from writing simultaneously
- **Readers can access it simultaneously**

A solution (does not exclude starvation)

- **Variable readcount: number of threads reading the database**
- **Semaphore mutex: protects the CS where readcount is updated**
- **Semaphore wrt: mutual exclusion between Writers and readers**
- **Writers should wait on wrt**
  - for each other
  - and also the end of all readings
- **Readers should**
  - wait on wrt when there are writers who write
  - block writers on wrt when there are readers who read
  - restart writers when no one is reading

Input and Writers

Input: two semaphores and one variable

```
mutex, wrt: semaphore (init. 1);
readcount: integer (init. 0);
```

Writer

```
wait (wrt);
. . .
// write
. . .
signal (wrt);
```

The readers

```
wait (mutex);
   readcount++;
   if readcount == 1 then wait (wrt);
signal (mutex);

// CS: read

wait (mutex);
   readcount -;
   if readcount == 0 then signal (wrt);
signal (mutex):
```

The first reader in a group might have to wait on wrt, they should also block writers. When he enters, the following will be able to enter freely

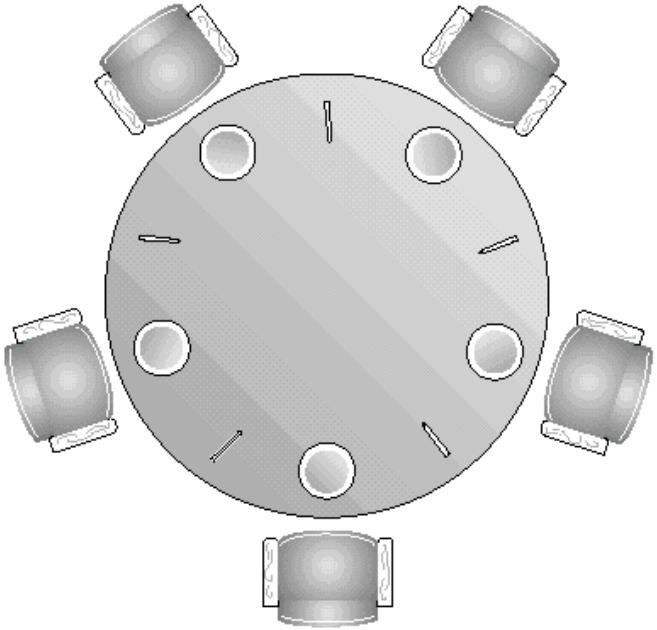
Last outgoing reader must allow writers access

Observations

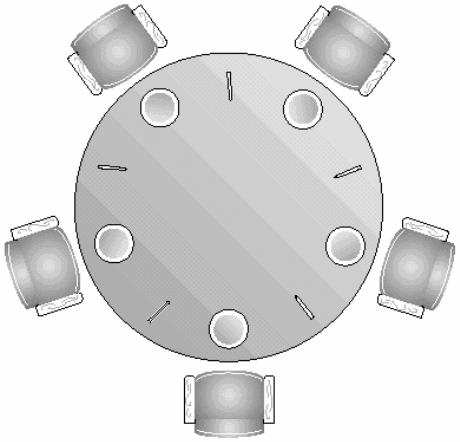
- The 1st reader that enters the CS blocks the writers (wait (wrt)), the last restarts them (signal (wrt))
- If 1 Writer is in the CS, 1 reader waits on wrt, the others on mutex
- a signal (wrt) can run a reader or a writer

The Dining Philosophers Problem

- 5 philosophers think, eat, think, eat, think ….
- In the center a bowl of rice.
- Only 5 chopsticks available.
- Require 2 chopsticks for eating.
- A classical synchronization. problem
- Illustrates the difficulty of allocating resources among process without deadlock and starvation



The Dining Philosophers Problem



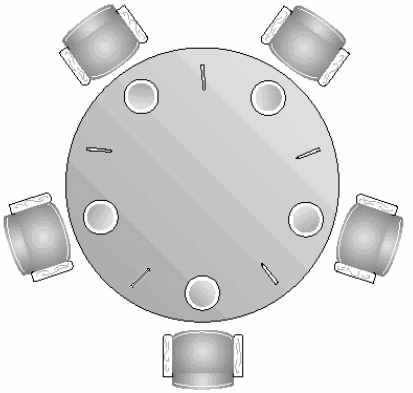
- **Each philosopher is a process**
- **One semaphore per chopstick:**
  - So that two philosophers cannot grab the same chopstick simultaneously
  - semaphore chopst[5];
  - Initialization: chopst[i].value=1 for i = 0 to 4

```
Process Pi:
while(true)
{
 think();
 wait(chopst[i]);
 wait(chopst[(i+1)%5]);
 eat();
 signal(chopst[i+1)%5]);
 signal(shopst[i]);
}
```

Does it work?

- **What happens if each philosopher starts by picking his left chopstick?**
- Deadlock! Poor philosophers, they should know better…

The Dining Philosophers Problem



- A solution: allow only 4 philosophers at a time to try to eat
- Then 1 philosopher can always eat when the other 3 are holding 1 chopstick
- Hence, we can use another semaphore T that would limit at 4 the number of philosophers trying to take a chopstick
- How to initialize T.count?
  - T.value = 4

```
Process Pi:
while(true)
{
 think();
 wait(T);
 wait(chopst[i]);
 wait(chopst[(i+1)%5]);
 eat();
 signal(chopst[(i+1)%5]);
 signal(chopst[i]);
 signal(T);
}
```

Advantage of semaphores
(compared to previous solutions)

- Only one shared variable per critical section
- only two operations: wait, signal
- more localized control (only with the previous ones)
- easy extension in case of more. threads
- possibility of bringing in more. threads both in a critical section
- management of queues by the OS: starvation avoided if the OS is fair (eg FIFO queues)

Problem with semaphores: programming difficulty

- **wait and signal are scattered among several threads, but they must match**
  - See. bounded buffer program
- **Use must be correct in all threads**
- **A single "bad" thread can cause an entire collection of threads to fail (e.g. forgot to signal)**
- **Consider the case of a thread that has waits and signals in loops and tests ...**

Monitors: another solution

- **Constructions (in high-level language) which provide functionality equivalent to semaphores but easier to control**
- **Available in:**
  - Concurrent Pascal, Modula-3 ...
    - *synchronized method* in Java (simplified monitors)

Monitor

- **Is a module containing:**
  - one or more procedures
  - an initialization sequence
  - local variables
- **Characteristics:**
  - local variables accessible only by using a monitor procedure
  - a thread enters the monitor by invoking one of its procedures
  - *only one thread can execute in the monitor* at any moment (but more threads can be waiting in the monitor)

Monitor

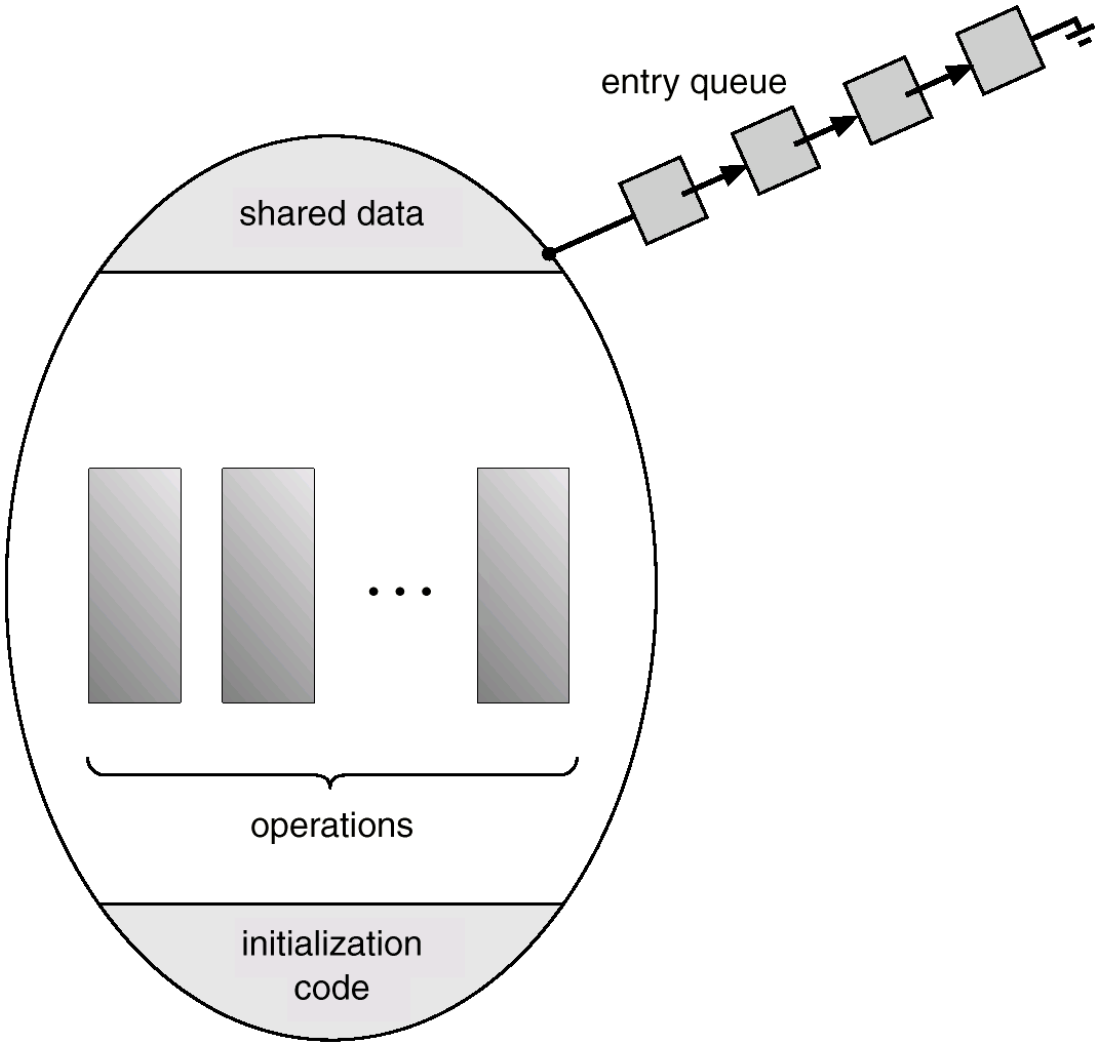
- **It alone ensures mutual exclusion: no need to explicitly program it**
- **We ensure the protection of shared data by placing it in the monitor**
  - The monitor locks the shared data when a thread enters it
- **Thread synchronization is performed by using conditional variables which represent conditions after which a thread might wait before executing in the monitor**

General structure of the monitor (Java style)

```
monitor monitor-name
{// vars declarations
   public entry p1 (...) {method code p1}
   public entry p2 (...) {method code p2}
. . .
}
```

The only way to manipulate the internal vars of the monitor is to call one of the input methods

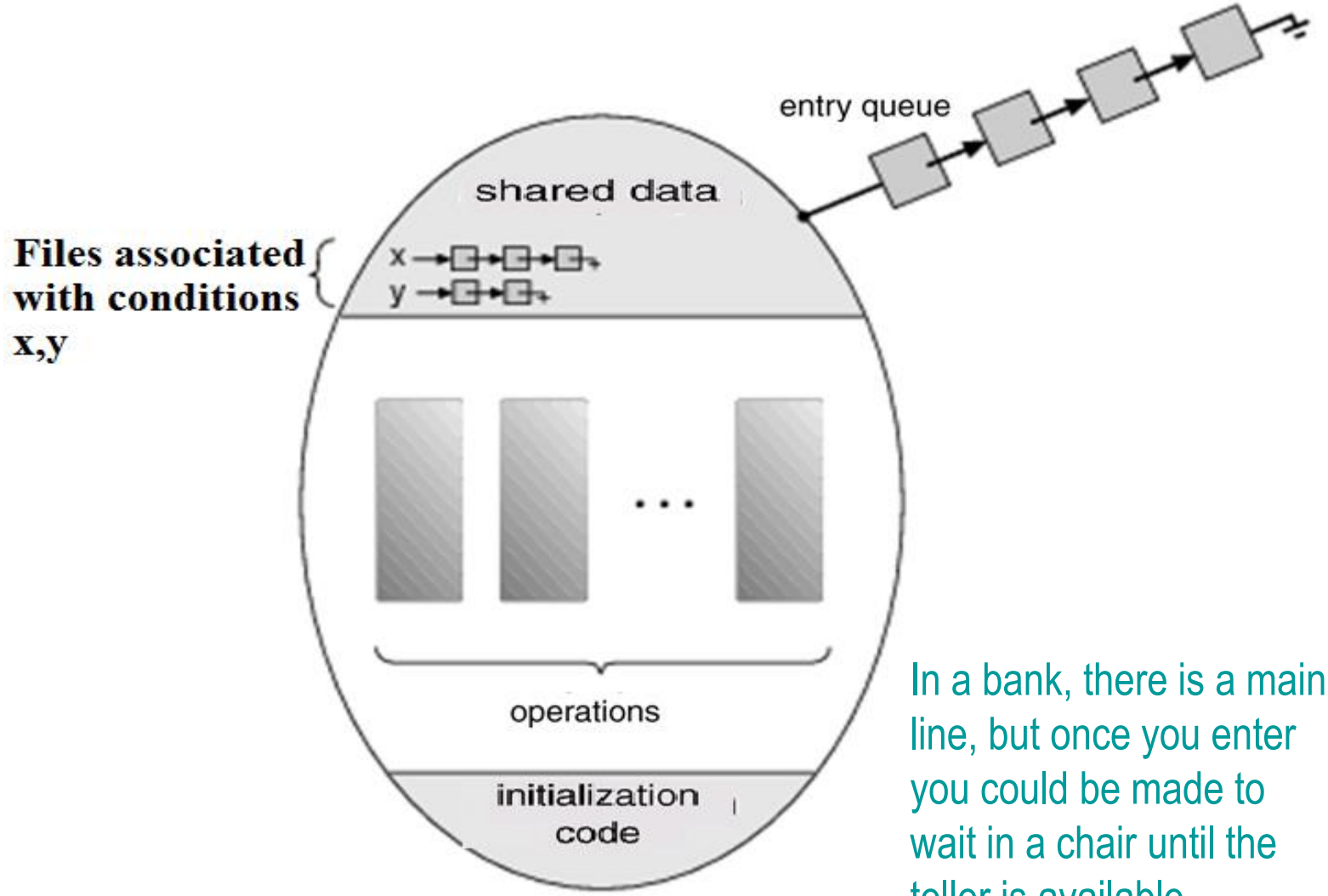
Monitor: Schematic view simplified Java style



Conditional variables (do not exist in Java)

- **are local to the monitor (accessible only within the monitor)**
- **can be accessed and changed only by two functions:**
  - x.wait() blocks execution of the calling process/thread on condition (variable) a
    - the process/thread can resume execution only if another process/thread executes x.signal()
  - x.signal() resume execution of some process/thread blocked on condition (variable) x.
    - If several such process exists: choose any one
    - If no such process exists: do nothing

Monitor with Conditional Variables



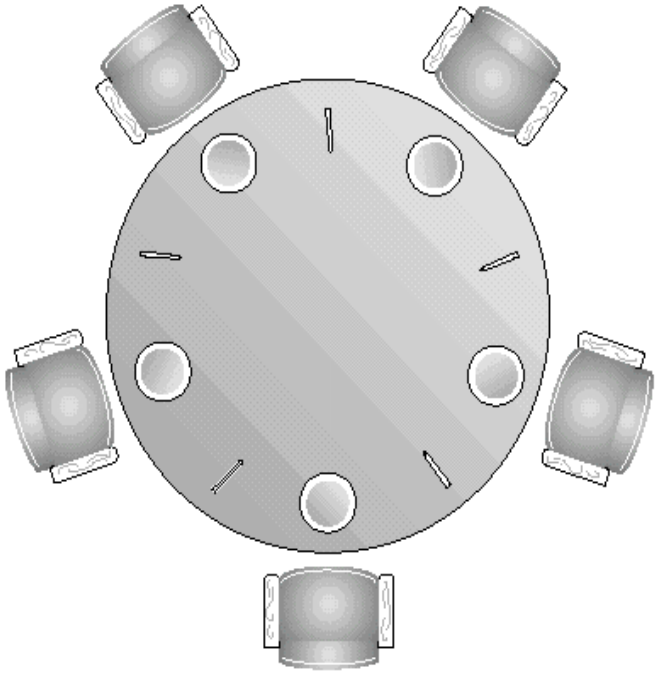
In a bank, there is a main line, but once you enter you could be made to wait in a chair until the teller is available

A Problem with signal()

- **Say a thread/process P executes x.signal() and frees a thread/process Q,**
  - Now, 2 threads/processes that want to execute, P and Q – not allowed
- **Two possible solutions:**
  - Signal and Wait
    - P waits until Q leaves the monitor (e.g. in a special queue – *urgent* – see Stallings).
  - Signal and Continue
    - P continues its execution and Q waits until P leaves the monitor

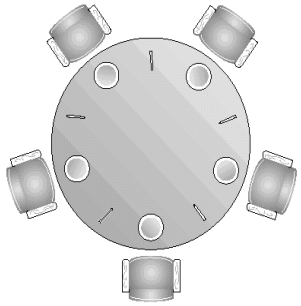
Back to the Dining Philosophers Problem

- 5 philosophers think, eat, think, eat, think ….
- In the center a bowl of rice.
- Only 5 chopsticks available.
- Require 2 chopsticks for eating.
- A classical synchronization. problem



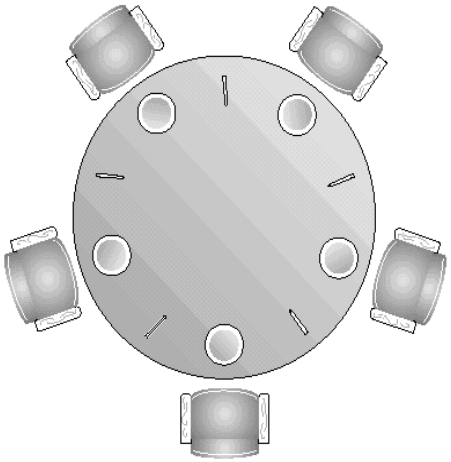
Dinning Philosophers: A Monitor

- **Define the monitor « dp » (dining philosophers)**
- **Shared variables: each philos. has its own state that can be: (thinking, hungry, eating)**
  - philosopher i can only have state[i] = eating if its neighbors are not eating
- **Conditional variables: each philosopher has a condition self**
  - the philosopher i can wait on self [i] if it wants to eat but cannot obtain 2 chopsticks
- **Four functions**
  - pickup(i) – philo. i wants to pick up chopsticks
  - putdown(i) – philo. i puts down his/her chopsticks
  - test(i) – test the state of philo. i
  - Initialization_code() - initialization

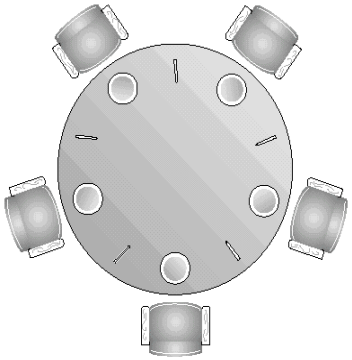


Philosopher i executes the loop:

- **while(true)**
- **{**
- ***dp.pickup(i)***
- **eat**
- ***dp.putdown(i)***
- **}**



A philosopher eats



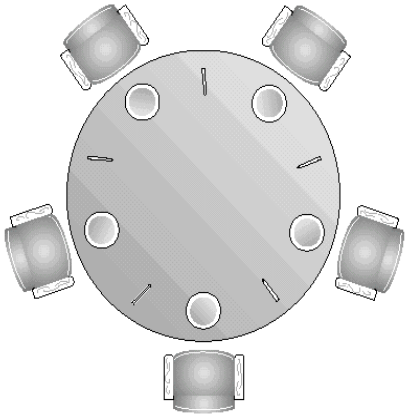
```
void test(int i)
{
     if ( (state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) )
     {
          state[i] = EATING;
          self[i].signal();
     }
}
```

A philosopher eats if his/her neighbors do not eat and he/she is hungry.

When the philosopher starts eating, a signal allows the philosopher to leave the semaphore.

No change in state occurs if tests fail.

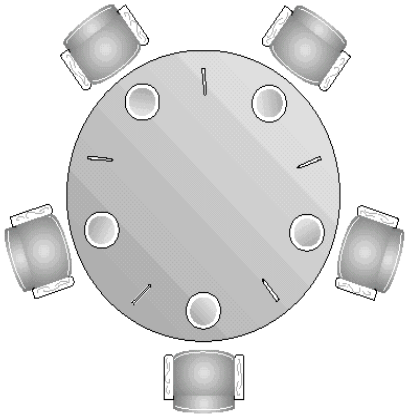
Pickup Chopsticks



```
void pickUp(int i)
{
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
}
```

Phil. tries to start eating by doing a test. If comes out of test not eating, must wait – blocs until the execution of self[i] signal()

Put down Chopsticks



```
void putDown(int i)
{
        state[i] = THINKING;
            // test the two neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
```

Once eating is done, a philosopher tries to get his/her neighbors eating using test.

Relationship between monitors and other mechanisms

- **The monitors are implemented using semaphores or other mechanisms already seen**
- **It is also possible to set up semaphores using monitors!**
  - See text

The problem of CS in practice ...

- **Real systems make available several mechanisms which can be used to achieve the most efficient solution in different situations**

Important concepts in Chapter 6

- **Critical sections: why**
- **Difficulty of the synch problem on CS**
  - Good and bad solutions
- **Atomic access to memory**
- **Pure software solutions**
- **Hardware solution: test-and-set**
- **Solutions by system calls:**
  - Semaphores, monitors, operation
- **Typical problems: bounded buffer, reader-writers, philosophers**


Thank You!
Ευχαριστώ
ขอบคุณ
Vielen Dank
Merci
Teşekkürler
DMnvwd
شكراً
متشكرم
Gracias
Grazie
Bedankt
Dankie
THANK YOU
Köszönettel
Obrigado!
شکریا
Díky
謝謝
GADDA GUEY
WAD MAHAD
SAN TAHAY
감사합니다
Urakoze