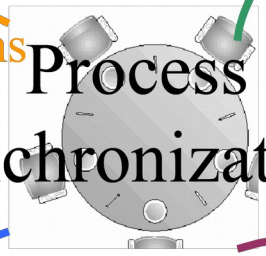
Module 5: Process Synchronization

Reading: Chapter 6

Process Synchronization



What is the problem ?

Solutions

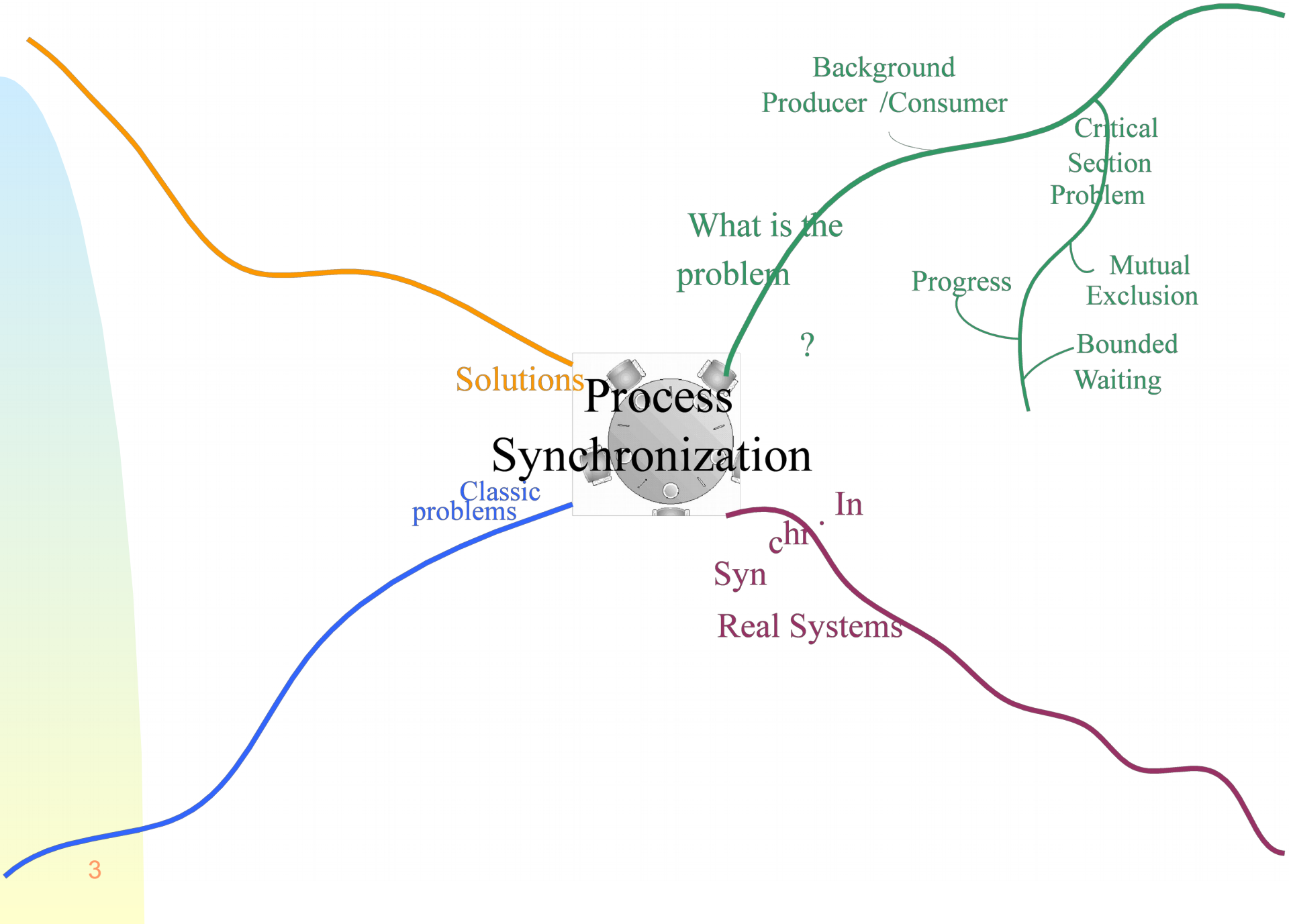
Classic problems

In

chr.

Syn

Real Systems


Process Synchronization
What is the problem ?
Background
Producer /Consumer
Critical Section Problem
Mutual Exclusion
Progress
Bounded Waiting
Solutions
Classic problems
In
Syn
Real Systems

Problems with concurrence = parallelism

- Concurrent threads sometimes share data (files and common memory) and resources
  - These are cooperative tasks
- If access is not controlled, the result of the program execution may depend on the order of interleaving of instruction execution (non-deterministic).
- A program can give different results from one execution to another and sometimes undesirable results.

Is there a problem?

Consider the following code, in which a child is sharing memory with the parent that created it.

What is the output produced by this code?

```
a = 2;
b = 3;
pid = clone();
if (pid < 0)
   // error handling
else if (pid == 0)
{
   // child thread
   b = 4;
   c = a+b;
   printf("%d\n", c);
}
else
{
   // parent thread
   a = 0;
   c = a+b;
   printf("%d\n",c);
}
```

Shared Data Structure Example

OK, that was silly, what about a more real-life example?

Very simplified queue:

```
enqueue(data):
    qData[inPos] = data;
    inPos++;

dequeue():
    data = qData[outPos];
    outPos++;
```

What happens if several processes/threads try to use the queue functions concurrently?

Producer-Consumer Example

The Producer executes

```
while (true)
{
    while (count == BUFFER_SIZE)
                ; // do nothing
    // produce an item and put in
    // nextProduced
    buffer[inIx] = nextProduced;
    inIx = (inIx + 1) % BUFFER_SIZE;
    count++;
}
```

Producer-Consumer Example

The Consumer executes

```
while (true)
{
    while (count == 0)
                ; // do nothing
    nextConsumed =  buffer[outIx];
    outIx = (outIx + 1) % BUFFER_SIZE;
    count--;
    // consume the item in nextConsumed
}
```

So, does it work? Why?

- Lets see…

Producer-Consumer Example

- `count++` could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving:

S0: producer execute `register1 = count` {register1 = 5}
S1: producer execute `register1 = register1+1` {register1 = 6}
S2: consumer execute `register2 = count` {register2 = 5}
S3: consumer execute `register2 = register2-1` {register2 = 4}
S4: producer execute `count = register1` {count = 6 }
S5: consumer execute `count = register2` {count = 4}

Summary of the problem

- **Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources**
- **If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data**
- **The results of actions performed by concurrent processes will then depend on the order in which their execution is interleaved –** race condition
- **Let us abstract the danger of concurrent modification of shared variables into the** critical-section problem.

Critical-Section Problem

- **The piece of code modifying the shared variables where a thread/process needs** exclusive **access to guarantee consistency is called** critical section.
- **The general structure of each thread/process can be seen as follows:**

```
while (true)
{
    entry_section
    critical section (CS)
    exit_section
    remainder section (RS)
}
```

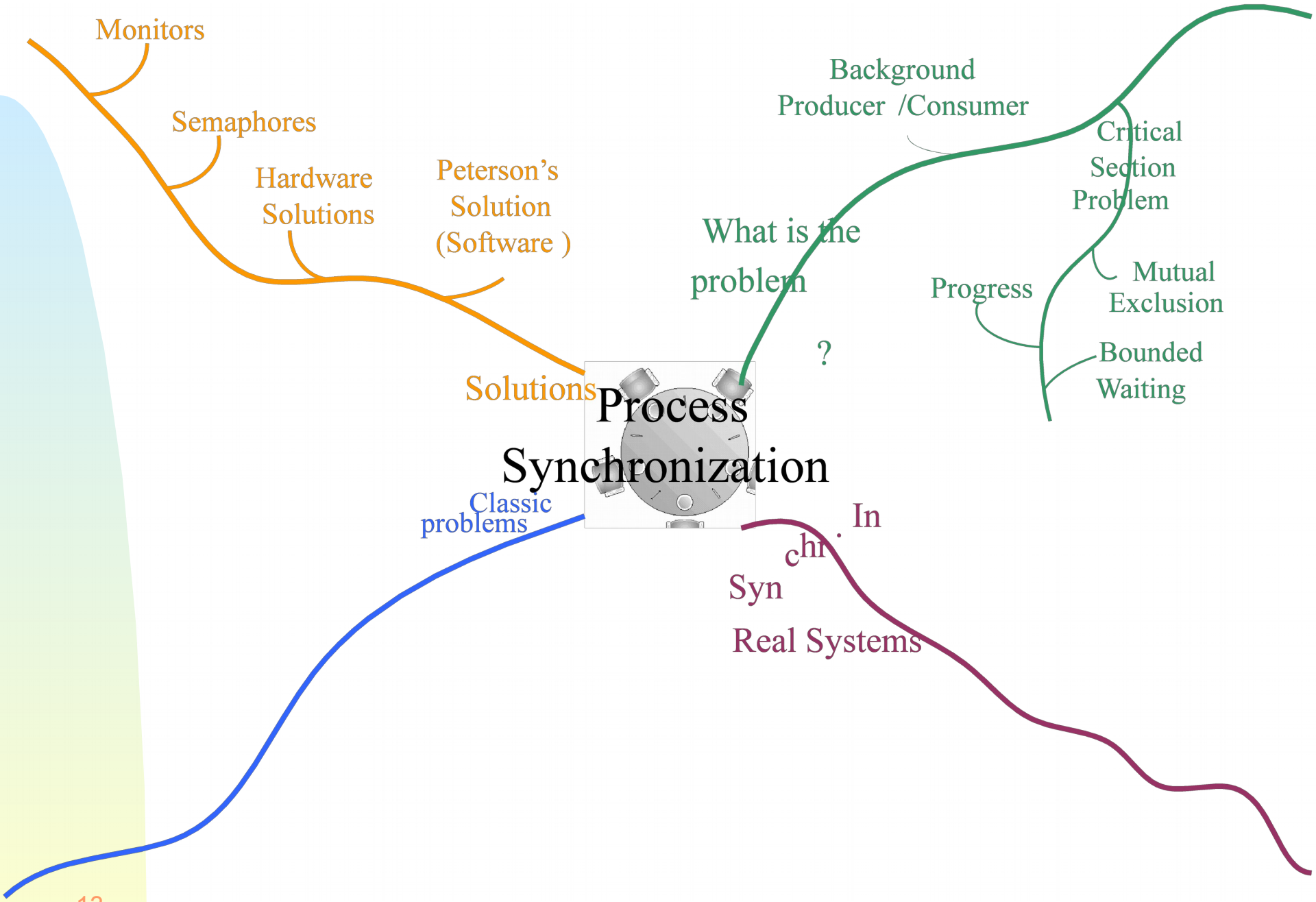
- Critical **(CS) and** remainder **(RS) sections are given,**
- **We want to design entry and exit sections so that the following requirements are satisfied:**

Solution Requirements for CSP

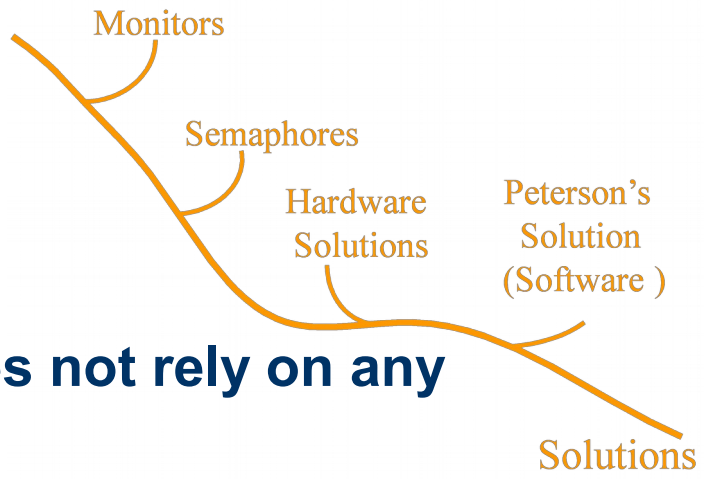
1. Mutual Exclusion - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. Progress - If there exist some processes wishing to enter their CS and no process is in their CS, then one of them will eventually enter its critical section.
   - No deadlock
   - Non interference – if a process terminates in the RS, other processes should still be able to access the CS.
   - Assume that a thread/process always exits its CS.
3. Bounded Waiting - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - No famine.

Assumptions

- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the N processes
- Many CPUs may be present but memory hardware prevents simultaneous access to the same memory location
- No assumption about the order of interleaved execution


Monitors
Semaphores
Hardware Solutions
Peterson's Solution (Software)
Solutions
Background
Producer /Consumer
What is the problem
?
Critical Section Problem
Progress
Mutual Exclusion
Bounded Waiting
Process Synchronization
Classic problems
In
Syn
Real Systems

CSP 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 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.**

Software Solutions

Lets start with a simpler problem – only two tasks

- **Two tasks, T_0 and T_1 (also T_i and T_j, where i!= j)**
- **We want to design the entry and exit sections**

```
while (true)
{
    entry_section
    critical section (CS)
    exit section
    remainder section (RS)
}
```

Algorithm 1

Idea:

- Have a shared variable `turn` indicating whose turn it is now. It is initialized to 0 or 1, does not matter.
- Task Ti may enter the critical section if and only if `turn` = i
- After exiting the critical section, turn is set to the other value, to let the other task gain access to its critical section
- Ti can be in a busy wait if Tj is in its CS.

```
Task T0:
while(true)
{
   while(turn!=0){/*bw*/}
   Critical Section
   turn=1;
   Remainder Section
}
```

```
Task T1:
while(true)
{
   while(turn!=1){/*bw*/}
   Critical Section
   turn=0;
   Remainder Section
}
```

Algorithm 1

Discussion:

- **Does it ensure mutual exclusion?**
  - **Yes, at any moment, `turn` has only one value, and if a task Ti is in its critical section, then `turn` = i**
- **Does it satisfy the bounded waiting requirement?**
  - **Yes, the tasks alternate**
- **Does it satisfy the progress requirement?**
  - **No, because it requires strict alternation of CSs**
  - **If a task requires its CS more often then the other, it cannot get it.**

```
Task Ti:
while(true)
{
   while(turn!=i){};
   Critical Section
   turn=j;
   Remainder Section
}
```

Algorithm 2

OK, Algorithm 1 was not that great, we do not need to alternate.

Idea:

- **Lets introduce shared variables `flag[0]` and `flag[1]` to indicate whether task 0 and task 1 resp. wants to enter the CS**
- **Task i sets to true the `flag[i]` before trying to enter the CS and set to false `flag[i]` after completing the CS.**
- **Ti does not enter the CS while `flag[j]` is true.**
- **If only one task wants to enter the CS several times, it has no problems, as the other flag is always unset**

```
Task T0:
while(true)
{
   flag[0] = true;
   while(flag[1]){/*bw*/}
   Critical Section
   flag[0] = false;
   Remainder Section
}
```

```
Task T1:
while(true)
{
   flag[1] = true;
   while(flag[0]){/*bw*/}
   Critical Section
   flag[1] = false;
   Remainder Section
}
```

Algorithm 2

Discussion:

- **Does it ensure mutual exclusion?**
  - **Yes, a process in CS has its flag set, but it does not enter CS if the flag of the other process is set**
- **Does it satisfy the bounded waiting requirement?**
  - **If the task Ti is scheduled after the Tj has set flag[j] to false but before it set again flag[j] to true, Ti will enter its CS.**
- **Does it satisfy the progress requirement?**
  - **What happens after the following sequence of instructions:**
    - **T0: flag[0] = true;**
    - **T1: flag[1] = true;**
  - No progress, deadlock occurs.


```
Task Ti:
while(true)
{
   flag[i] = true;
   while(flag[j]){/*bw*/ }
   Critical Section
   flag[i] = false;
   Remainder Section
}
```

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.

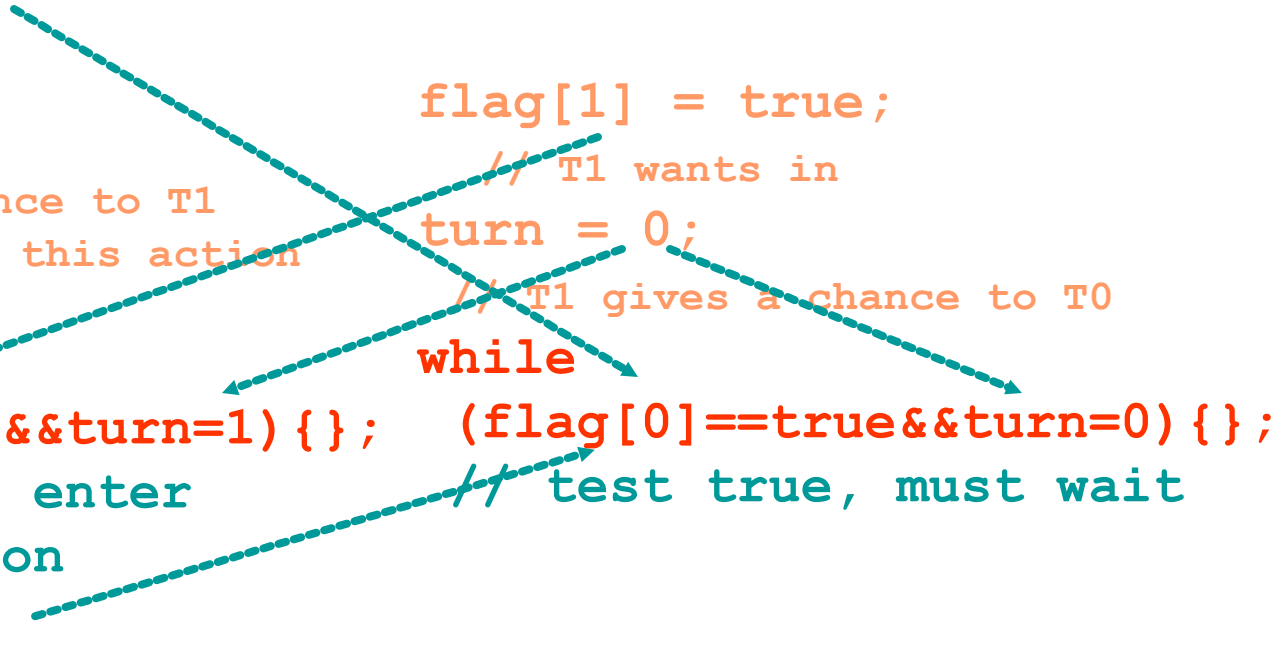
- Nonsense! There must be a way they block each other. Perhaps they start staggered…

Both tasks want to enter!

```
Task T0:                                  Task T1:

    CS
flag[0] = false;
 // T0 wants out
    RS
flag[0] = true;                           flag[1] = true;
   // T0 wants in                           // T1 wants in
turn = 1;                                 turn = 0;
 // T0 gives a chance to T1                // T1 gives a chance to T0
//  but T1 cancels this action

while                                     while
 (flag[1]==true&&turn=1){};                (flag[0]==true&&turn=0){};
 // test false, enter                      // test true, must wait
 Critical Section
flag[0] = false
```



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

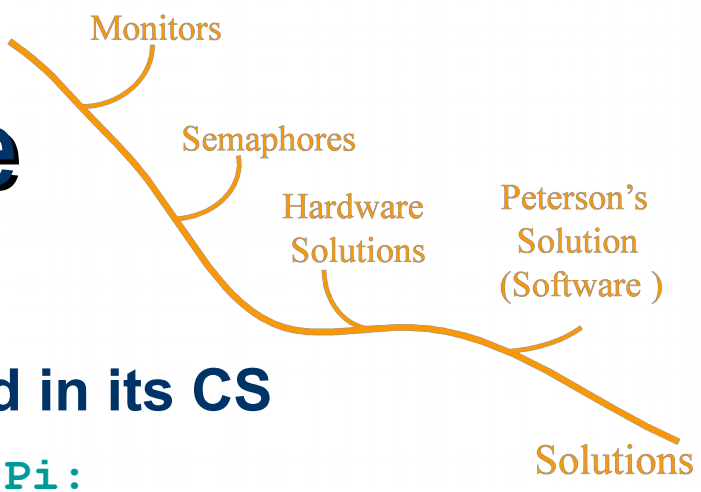
A few additional ideas

- A solution to the 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 baker's algorithm
- For tasks with shared variable to work properly, all threads involved must used the same synchronization algorithm
  - A common protocol

Critique of software solutions

- Difficult to program! And to understand!
  - The subsequent solutions we shall study are based on the existence of specialized instructions, which makes the work easier.

- Threads/processes that desire entry into their CS are busy waiting; which consumes CPU time.
  - For long critical sections, it is preferable to block threads/processes that must wait….



Hardware Solution: disable interrupts

Simple solution:

- **A process would not be preempted in its CS**

```
Process Pi:
while(true)
{
   disable interrupts
   critical section
   enable interrupts
   remainder section
}
```

Discussion:

- **Efficiency deteriorates: when a process is in its CS, it's impossible to interlace the execution of other processes in their RS.**
- **Loss of interruptions**
- **On a multiprocessor system: mutual exclusion is not assured.**
- **A solution that is not generally acceptable.**

Hardware solutions: specialized instructions

- Normal: when a thread or process access an address of memory, no other can access the same memory address at the same time.
- Extension: define machine instructions that perform multiple actions (ex: reading and writing) in the same memory location atomically (indivisible).
- An atomic instruction can only be executed by a single thread/process at a time (even in the presence of multiple processors).

The test-and-set instruction

- **Pseudocode for test-and-set:**

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

Atomic instruction!

- **An algorithm using test-and-set for mutual exclusion:**
  - **Variable b is initialized to 0**
  - **The first Ti to set b to 1 enters its CS.**

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

The test-and-set instruction (cont.)

- Mutual exclusion is assured: if Ti enters the CS, the other Tj are busy waiting.
  - Problem: still using busy waiting.
- Can easily attain mutual exclusion, but needs more complex algorithms to satisfy the other requirements to the CSP.
  - When Ti leaves its CS, the selection of the next Tj is arbitrary: no bounded waiting: starvation is possible.

Using xchg for mutual exclusion (Stallings)

- The 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.
- xchg(a,b) suffers from the same problems as the test-and-set

- Certain CPUs (ex: Pentium) offer the instruction xchg(a,b) that exchanges atomically two variables.

usage:

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

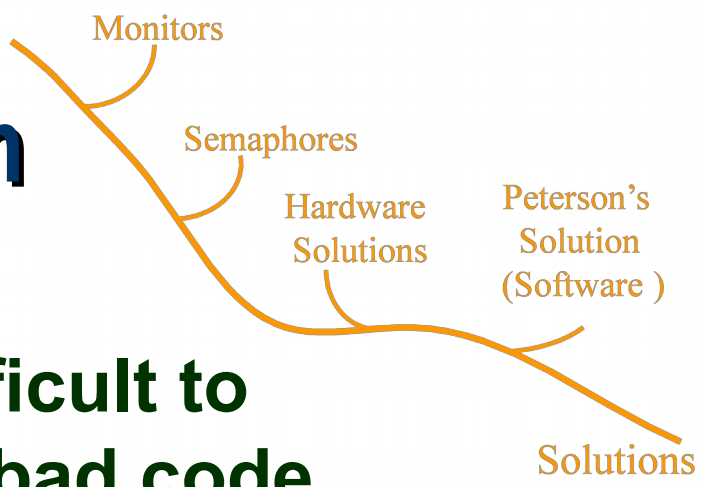
Drawback of software & hardware solutions

- **Processes that are requesting to enter in their critical section are busy waiting (consuming processor time needlessly)**

Idea:

- **If Critical Sections are long, it would be more efficient to block processes that are waiting...**
- **Can the OS help us? It can block processes.**

Solutions based on system calls – the semaphore



- Solutions seen so far are difficult to program and easily leads to bad code.
- Need to find methods to avoid common errors, such as deadlocks, starvation, etc.
  - Need some higher level functions.
- The subsequent methods we shall study use more powerful functions that the OS can provide with system calls.
- Lets first study sempahores.

Semaphores – Version 1 – spinlocks

- The semaphore is a simple integer variable S on which three operations are allowed:
  - Initialization to a positive value (including zero).
  - wait(S)
    - Originally called P() (proberen → to test)
    - Also called acquire()
  - signal(S)
    - Originally called V() (verhogen → to increment)
    - Also called release()
- The first version of semaphores studied use busy wait.

Spinlocks: Busy Wait Semaphores

- Easiest way to implement semaphores.
- Used in situations where waiting is brief or with multi-processors.
- When S=n (n>0), up to n processes will not block when calling wait().
- When S becomes 0, processes block in the call wait() up until signal() is called.
- A call to signal() unblocks a blocked process or increments the semaphore value.

```
wait(S)
{
   while(S<=0);//no-op
   S--;
}
```

The sequence `S<=0, S-` must be atomic.

```
signal(S)
{
    S++;
}
```

The signal call must be atomic.

Using semaphores as a Solution to the CSP.

- Can be applied with n processes.
- Initialize S to 1.
  - Means only 1 process can enter its CS.
- To allow k processes to enter their CS, initialize S to k.

```
do
{
    wait(S);
      /*Critical Section*/
    signal(S);
      /*Remainder Section*/
} while(TRUE);
```

Using semaphores for process synchronization

- We have 2 processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- Can we do that using semaphores?

Thinking….

- P2 has to wait until P1 executes S1 …
- … then P1 needs to signal P2 that it can execute S2
- So, let P2 issue wait(S) before S2 and P1 do signal(S) after S1
- How should we initialize S?
  - S.value = 1 does not work, as P2 can be scheduled first and execute statement S2
  - S.value = 0 works

```
Process P1:
  S1
  signal(S);
```

```
Process P2:
 wait(S);
 S2
```

Semaphores: observations

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

- When S becomes >1, the process that unblocks first is the first to test S (random)
  - Possibility of famine exists.
  - This is not true for Version 2
- Still using busy waiting.
- With version 1 (spinlocks): S is always >= 0.
- It is possible to limit the values of the semaphore to 0 and 1
  - This is the binary semphore and also called the mutex (for mutual exclusion).
  - Easier to implement
- The more general semaphore is called the counting semaphore

Semaphores – Version 2 – no busy wait

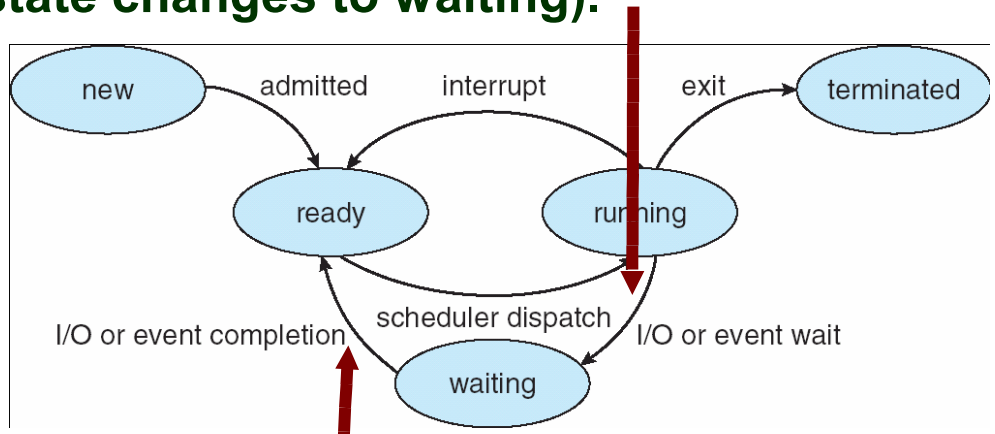
- When a process must wait for a semaphore to become greater than 0, place it in a queue of processes waiting on the same semaphore
- The queues can be FIFO, with priorities, etc. The OS controls the order in which processes are selected from the queue.
- Thus *wait* and *signal* become system calls similar to requests for I/O.
- There exists a queue for each semaphore similar to the queues defined for each I/O device.

Semaphore – Version 2

- The semaphore S now becomes a data structure:

```
typedef struct
{
  int value;
  struct process *list;
} semaphore;
```

- A process waiting on a semaphore S, is blocked and placed in the queue S.list (its state changes to waiting).



- Signal(S) removes (according to the queuing discipline, such as FIFO) a process from S.list and places it on the ready queue (process enters the ready state).

Implementation of wait()

```
wait(semaphore *S)
{
    S->value--;
    if(S->value < 0)
    {
        add the process to S->list;
        block(); /*process goes to state waiting*/
    }
}
```

- This system call must be atomic.
- When value becomes negative, its absolute value represents the number of blocked processes in list.
  - Recall that spinlock values never become negative (note that decrement and test operations are reversed)

Implementation of signal

```
signal(sempahore *S)
 {
     S->value++;
     if(S->value <= 0)
     { /* processes are waiting */
        Remove process P from S->list;
        wakeup(P); /*process becomes ready*/
     }
  }
```

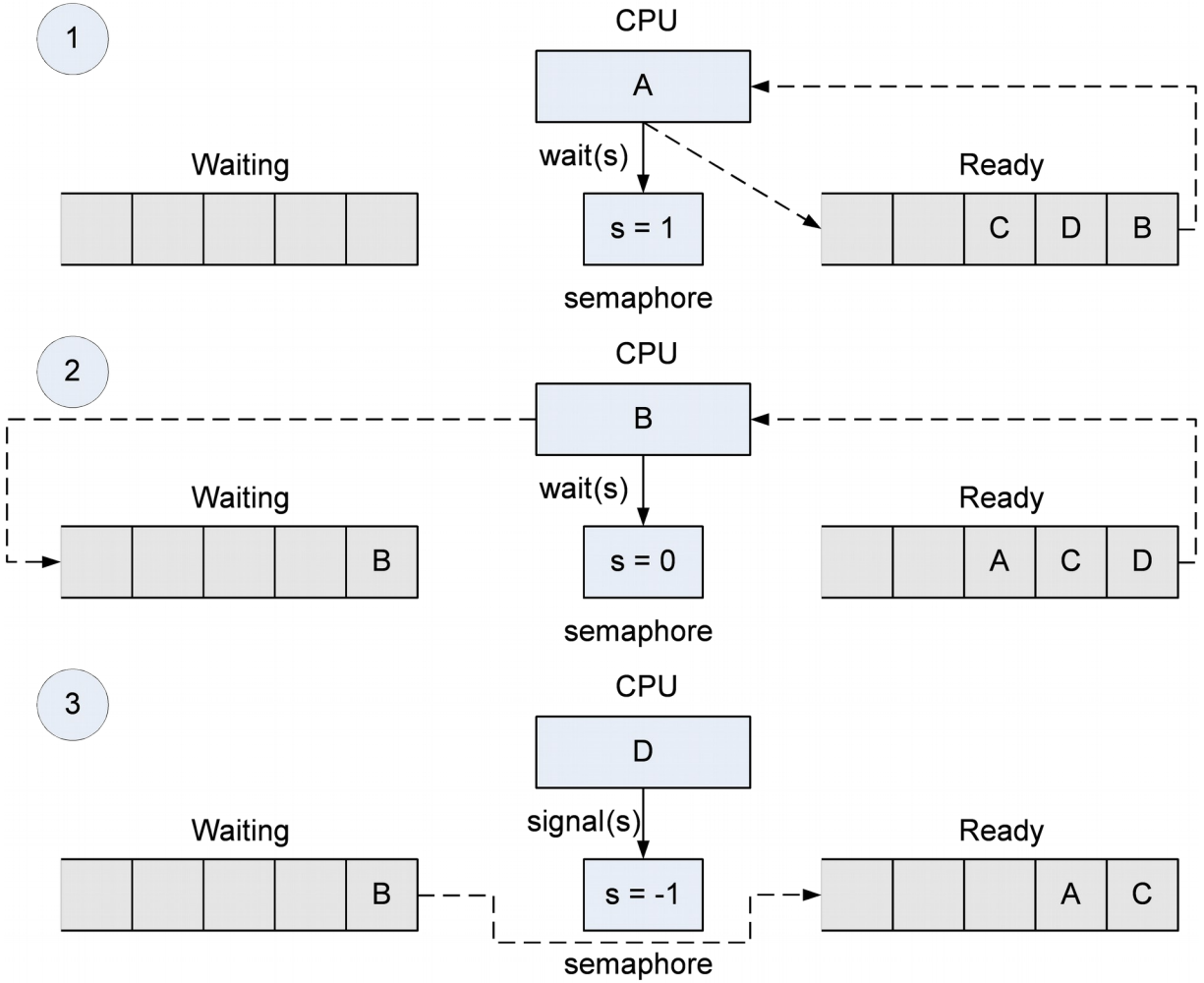
- This system call must be atomic
- The value of the semaphore can be initialised to a positive value (including 0).

Atomic execution of the wait() and signal() calls

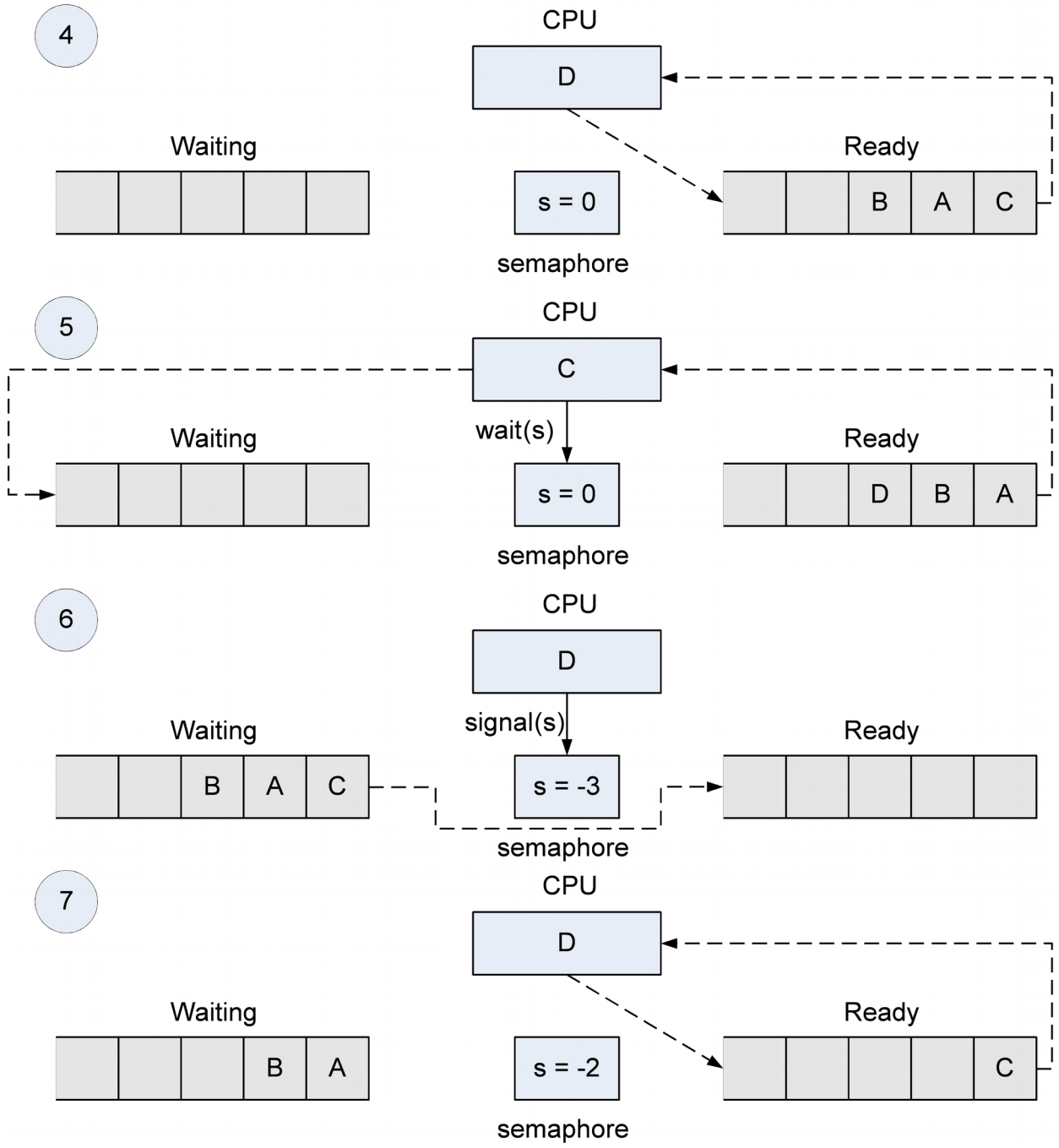
- wait() and signal() are in fact critical sections.
- Can we use semaphores for these critical sections?
- With single processor systems
  - Can disable interrupts
  - Operations are short (about 10 instructions)
- With SMP systems
  - Spinlocks
  - Other software and hardware CSP solutions.
- Result: we have not eliminated busy waiting
  - But the busy waiting has been reduced considerably (to the wait() and signal() calls)
  - The semaphores (version 2), without busy wait, are used within applications that can spend long periods in their critical section (or blocked on a semaphore waiting for a signal) – many minutes or even hours.
- Our synchronization solution is thus efficient.

Example of semaphore operation (Stallings)

- Process D
  - Produces some result and uses signal(s) to indicate the availability of results to other processes
- Processes A, B, C:
  - Consumes the results from D, and uses wait(s) to access them.
- Scenario 1:
  - D has already indicated that a result is ready and thus s=1



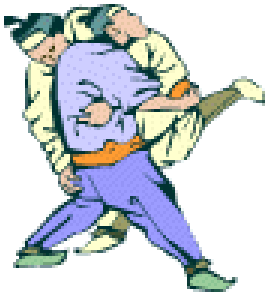
Example of semaphore operation (Stallings)



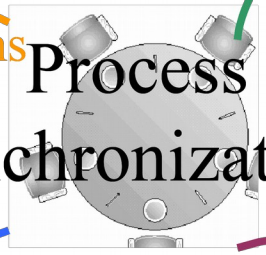
Deadlock and Starvation with Semaphores

- **Starvation: consider what can happen if a LIFO queuing discipline were used in a semaphore.**
- **Deadlock: Given semaphores S and Q both initialized to 1**

P0	P1
wait(S)	
	wait(Q)
wait(Q)	wait(S)
signal(S)	signal(Q)
signal(Q)	signal(S)



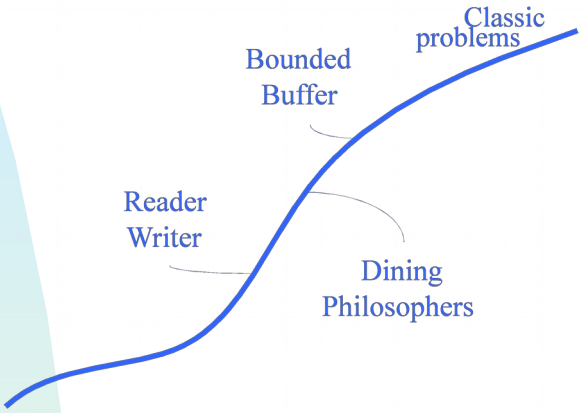
Process Synchronization



We will come back to monitors later

- Monitors
- Semaphores
- Hardware Solutions
- Peterson's Solution (Software)
- Solutions
- What is the problem ?
- Background Producer /Consumer
- Critical Section Problem
- Progress
- Mutual Exclusion
- Bounded Waiting
- Classic problems
- Bounded Buffer
- Reader Writer
- Dining Philosophers
- In
- chr
- Syn
- Real Systems

Classical Problems of Synchronization



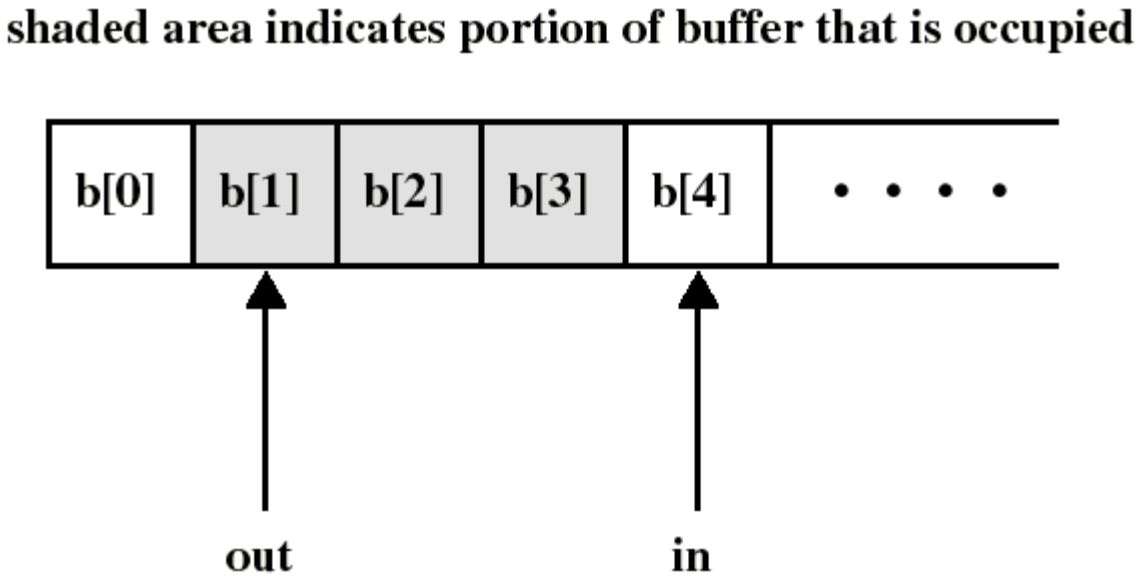
- Why to study them?
- These are abstractions of the typical synchronization problems that occur again and again
  - Demonstrate the typical issues in process synchronization
- Will allow us to practice using the semaphores

The Bounded Buffer problem

- A producer process produces information that is consumed by a consumer process
  - Ex1: a print program produces characters that are consumed by a printer
  - Ex2: an assembler produces object modules that are consumed by a loader
- We need a buffer to hold items that are produced and eventually consumed
- A common paradigm for cooperating processes

Start with an Unbounded buffer

- We assume first an unbounded buffer consisting of a linear array of elements
- `in` points to the next location in the buffer to receive the item to be produced
- `out` points to the next item to be consumed



Unbounded buffer

How to start?

- We need to ensure mutual exclusion when accessing the buffer
  - Have a semaphore `bmutex` initialized to 1
  - Call `wait(bmutex)` before each access of the buffer
  - Call `signal(bmutex)` when finished accessing the buffer
- How to make sure the consumer does not consume something that was not yet produced?
  - Have another semaphore `num`, where `num.value` is the number of produced, but not yet consumed items
  - When producer produces an item, calls `signal(num)`
  - When consumer wants to consume item, calls `wait(num)`
  - How to initialize `num.value`?

Solution for the unbounded buffer

```
Initialization:
  bmutex.value = 1;
  num.value = 0;
  in = out = 0;
```

```
  Critical Sections
append(v)        take()
{                {
  b[in] = v;       w = b[out];
  in++;            out++;
}                  return w;
                 }
```

```
Producer:
while(true)
{
  v = produce();
  wait(bmutex);
  append(v);
  signal(bmutex);
  signal(num);
}
```

```
Consumer:
while(true)
{
  wait(bmutex);
  w = take();
  signal(bmutex);
  consume(w);
  wait(num);
}
```

Does it actually work?

Solution for the unbounded buffer

```
Initialization:
  tmutex.value = 1;
  num.value = 0;
  in = out = 0;
```

```
  Critical Sections
append(v)        take()
{                {
  b[in] = v;       w = b[out];
  in++;            out++;
}                  return w;
                 }
```

```
Producer:
while(true)
{
  v = produce();
  wait(bmutex);
  append(v);
  signal(bmutex);
  signal(num);
}
```

```
Consumer:
while(true)
{
  wait(bmutex);
  wait(num);
  w = take();
  signal(bmutex);
  consume(w);
}
```

Will it work now?

Solution for the unbounded buffer

```
Initialization:
  tmutex.value = 1;
  num.value = 0;
  in = out = 0;
```

```
  Critical Sections
append(v)        take()
{                {
  b[in] = v;       w = b[out];
  in++;            out++;
}                  return w;
                 }
```

```
Producer:
while(true)
{
  v = produce();
  wait(bmutex);
  append(v);
  signal(bmutex);
  signal(num);
}
```

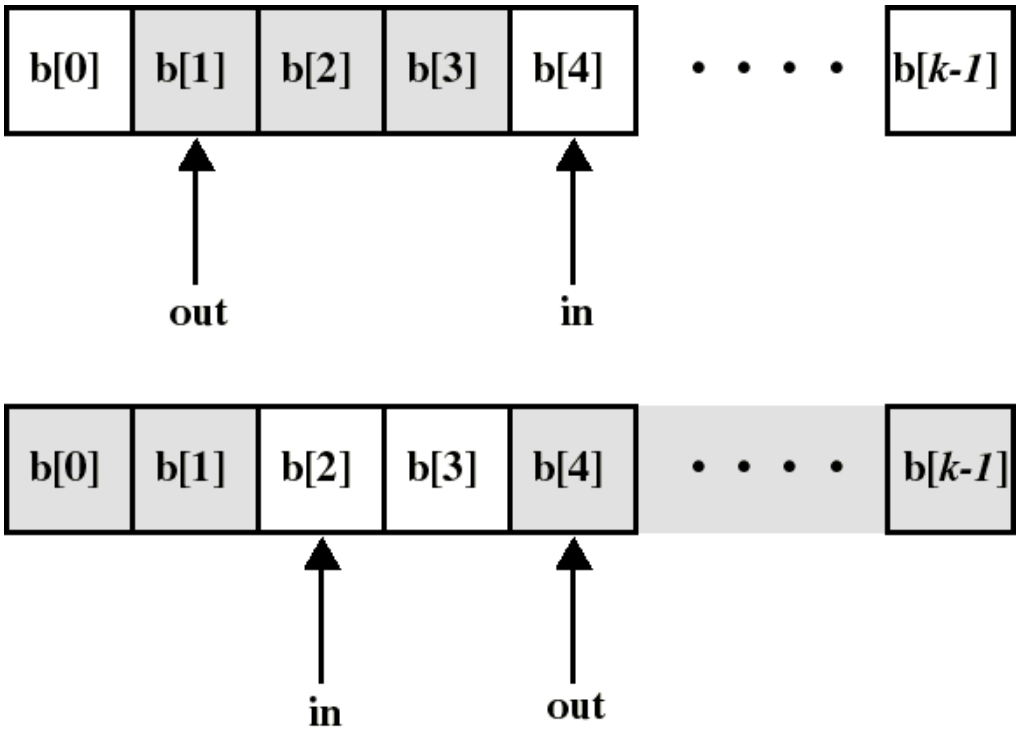
```
Consumer:
while(true)
{
  wait(num);
  wait(bmutex);
  w = take();
  signal(bmutex);
  consume(w);
}
```

Will it work now? Yes!

The Unbounded Buffer

- **Remarks:**
  - **Putting `signal(num)` inside the CS of the producer (instead of outside) has no effect since the consumer must always wait for both semaphores before proceeding**
  - **The consumer must perform `wait(num)` before `wait(bmutex)`, otherwise deadlock occurs if consumer enter CS while the buffer is empty**
- **Using semaphores is a difficult art...**

Bounded buffer of size k



- can consume only when number `num.value` of (consumable) items is at least 1
- can produce only when number `emty.value` of empty spaces is at least 1

Bounded Buffer

- As before:
  - we need a semaphore `bmutex` to have mutual exclusion on buffer access
  - we need a semaphore `num` to synchronize producer and consumer on the number of consumable items
    - don't consume if `num.value` <=0
- In addition:
  - we need a semaphore `emty` to synchronize producer and consumer on the number of empty spaces
    - don't produce if `emty.value` <=0
    - How to initialize `emty.value`?

Solution for the bounded buffer

```
Initialization:
   bmutex.value = 1;
   num.value = 0;
   emty.value = k;
   in = out = 0;
```

```
Producer:
while(true)
{
   v = produce();
   wait(emty);
   wait(bmutex);
   append(v);
   signal(bmutex);
   signal(num);
}
```

Critical Sections

```
append(v)
{
   b[in] = v;
   in=(in+1)%k;
}
```

```
take()
{
   w = b[out];
   out=(out+1)%k;
   return w;
}
```

```
Consumer:
while(true)
{
   wait(num);
   wait(bmutex);
   w = take();
   signal(bmutex);
   signal(emty);
   consume(w);
}
```

Readers-Writers Problem

- There are two types of processes accessing a shared database
  - readers only reads the data, but does not modify them
  - writers that want to modify the data
- In order to maintain consistency, as well as efficiency, the following rules are used
  - Several readers can access the database simultaneously
  - A writer needs to have an exclusive access to the database (has a critical section)
    - No readers or other writers are allowed while a writer is writing
  - What to do if there are several readers in the system and a writer arrives?
    - While there is a reader active, do not have new readers wait (first readers-writers problem).
    - No new reader is admitted if there is a writer waiting (second readers-writers problem).
    - Both might lead to starvation

Readers-Writers Problem

Hmmm, how to start?

- Have a semaphore wrt for allowing only one write process to modify the database
  - The writer process can simply use it, i.e.
    - wait(wrt), followed by CS, and then signal(wrt)
- How to prevent a writer entering when there is a reader?
  - The first reader grabs the semaphore wrt, i.e. executes wait(wrt) to prevent the writer from entering its CS
  - The last reader releases semaphore wrt (i.e. executed signal(wrt)
  - Now the writer can enter
- We need a counter `readers` to know how many readers are in the system, to know when to grab and release the wrt semaphore
  - Only the first one and the last one should do that
- But then we need a second semaphore rmutex to guard access to `readers`

Readers-Writers Problem

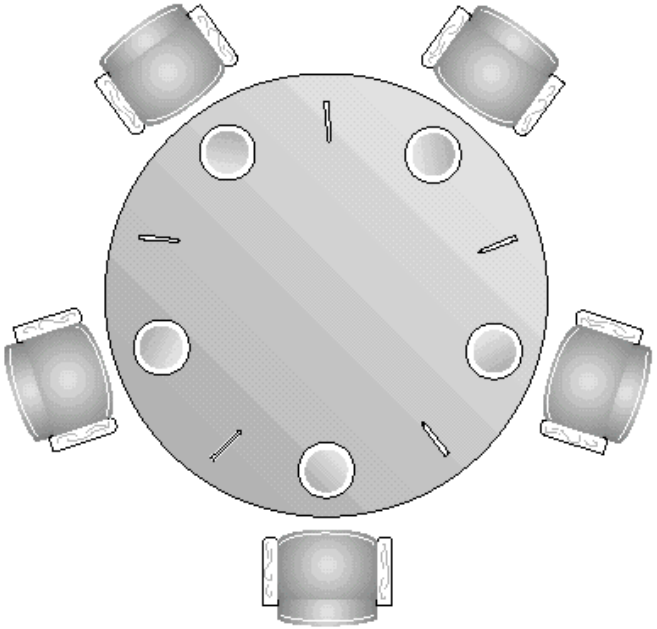
```
Initialization:
wrt.value = 1;
rmutex.value = 1;
readers = 0;
```

```
Writer:
while(true)
{
  wait(wrt);
  writer's CS;
  signal(wrt);
}
```

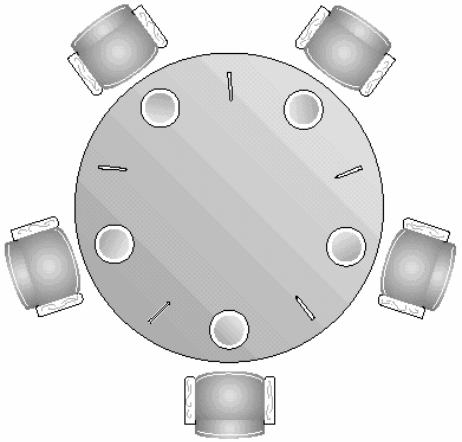
```
Reader:
while
{
  wait(rmutex);
  readers++;
  if (readers == 1)
      wait(wrt);
  signal(rmutex);
  reading database;
  wait(rmutex);
  readers--;
  if (readers == 0)
    signal(wrt);
  signal(rmutex);
}
```

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



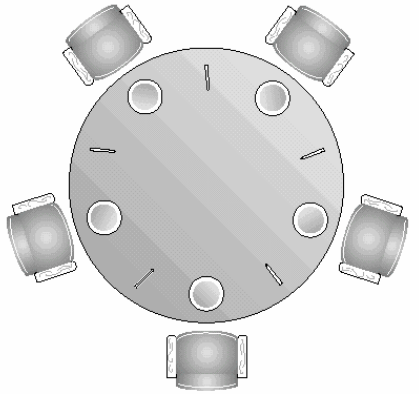
- Each philosopher is a process
- One semaphore per chopstick:
  - So that two philosophers cannot grab the same chopstick simultaneously
  - semaphore chopst[5];
  - Initialization: chopst[i].value=1 for i = 0 to 4

```
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

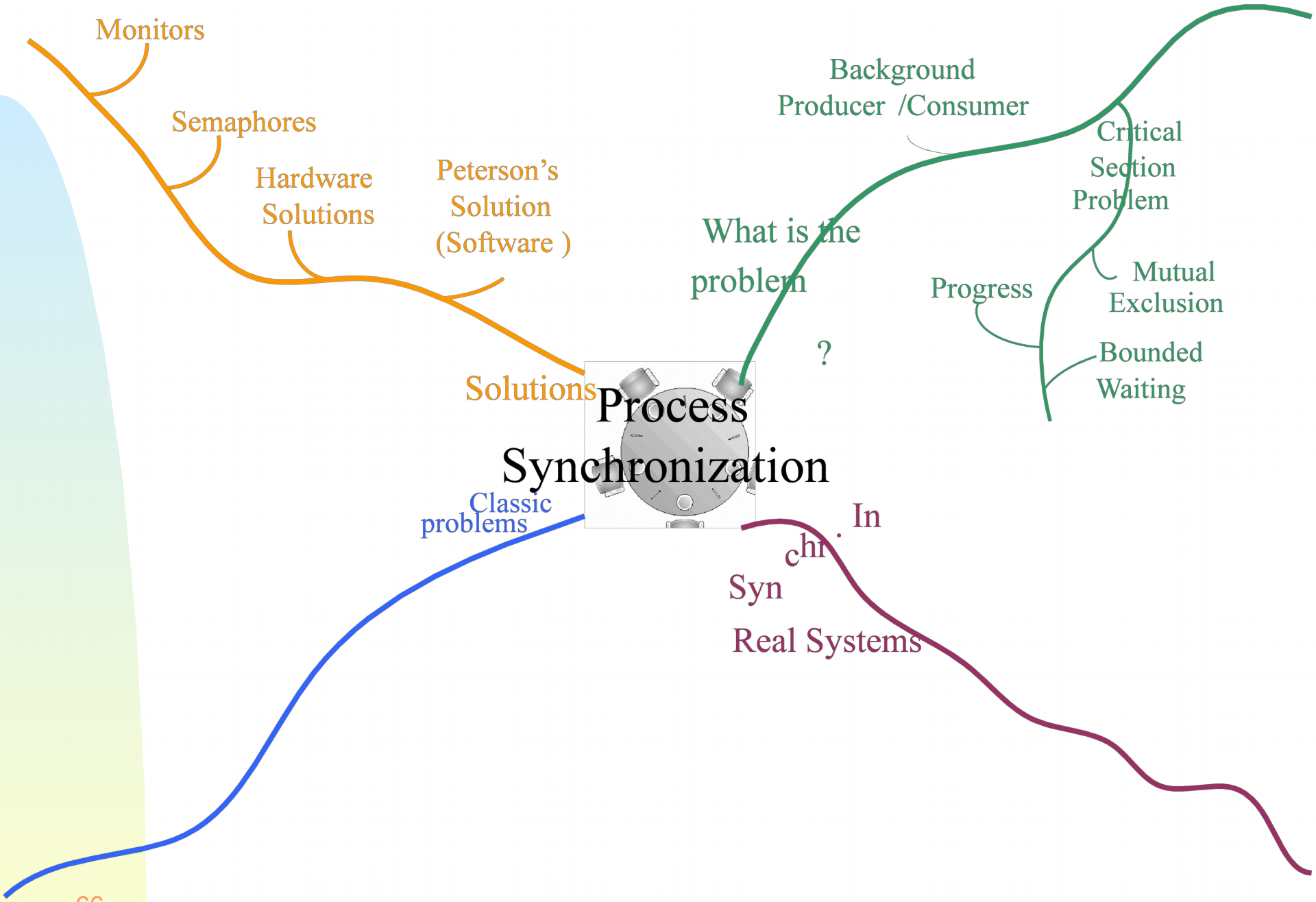


- A solution: allow only 4 philosophers at a time to try to eat
- Then 1 philosopher can always eat when the other 3 are holding 1 chopstick
- Hence, we can use another semaphore T that would limit at 4 the number of philosophers trying to take a chopstick
- How to initialize T.count?
  - T.value = 4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

```
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);
}
```

Advantages of semaphores (relative to other synchr. solutions)

- Single variable (data structure) per critical section.
- Two operations: wait, signal
- Can be applied to more than 2 processes.
- Can have more than a single process enter the CS.
- Can be used for general synchronization.
- Service offered by OS, including blocking and queue management.


Monitors
Semaphores
Hardware Solutions
Peterson's Solution (Software)
Solutions
Background
Producer /Consumer
What is the problem
?
Critical Section Problem
Progress
Mutual Exclusion
Bounded Waiting
Process Synchronization
Classic problems
In
Syn
Real Systems

Problems with semaphores: programming difficulty

- Wait and signal are scattered across different programs and running threads/processes
  - Not always easy to understand the logic
- Usage must be correct in all threads/processes
- One « bad » thread/process can make a collection of threads/processes fail (e.g. forget a signal)

Monitors

- Are high-level language constructs that provide equivalent functionality to that of semaphores but are easier to control
- Found in many concurrent programming languages
  - Concurrent Pascal, Modula-3, C#, Java...
- Can be implemented with semaphores...
  - See section 6.7.3 of the textbook

Monitors

- Is a software module (ADT – abstract data type) containing:
  - one or more procedures
  - an initialization sequence
  - local data variables
- Characteristics:
  - local variables accessible only by monitor's procedures
  - a process enters the monitor by invoking one of it's procedures
  - only *one process can be executing* in the monitor at *any one time* (but a number of threads/processes can be waiting in the monitor).

Monitors

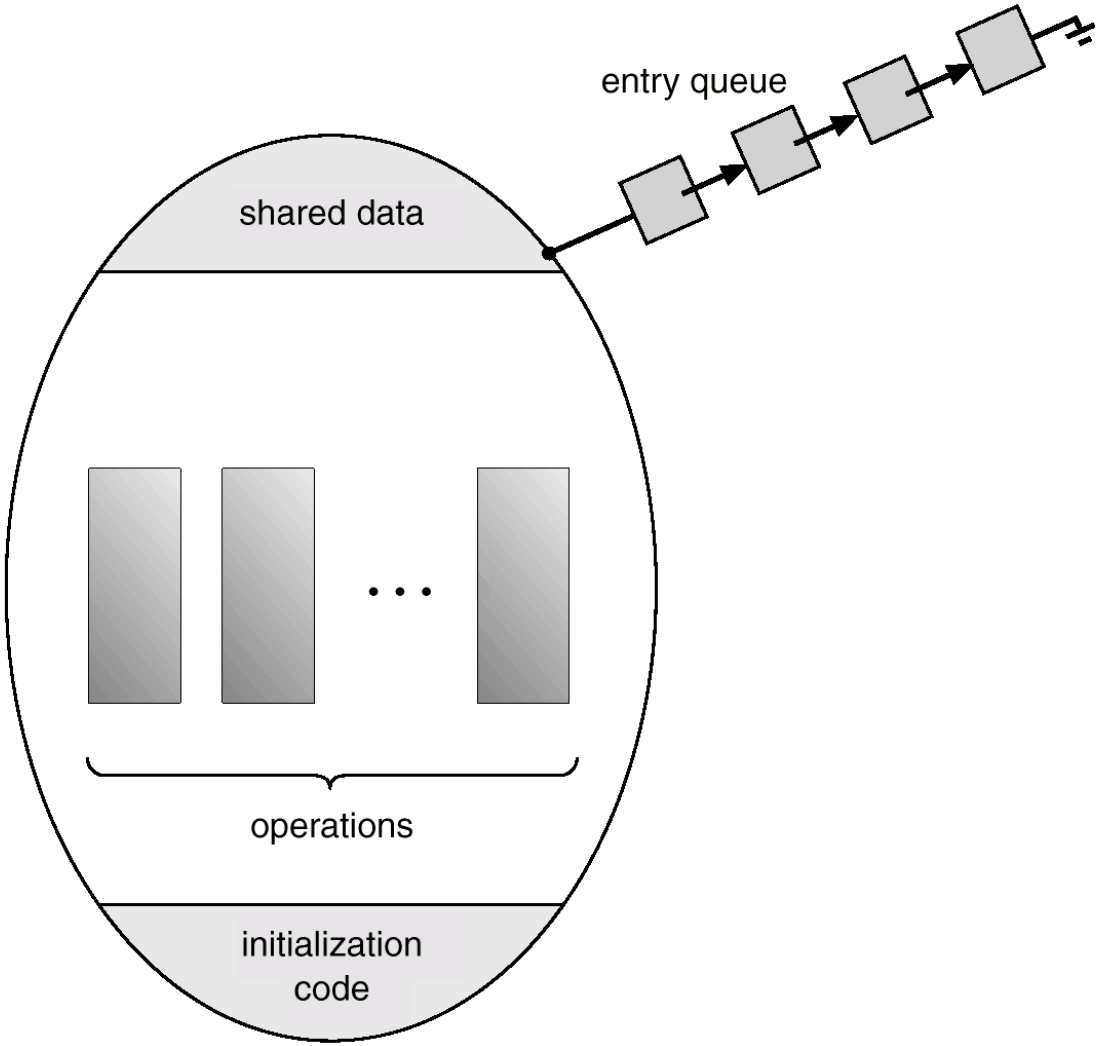
- The monitor ensures mutual exclusion: no need to program this constraint explicitly
- Hence, shared data are protected by placing them in the monitor
  - The monitor locks the shared data on process entry
- Process synchronization is done by the programmer by using condition variables that represent conditions a process may need to wait for before executing in the monitor

General Structure of the Monitor

```
monitor monitor-name
{
  /* shared variable declarations */
      p1(. . .) {p1 code}
      p2(. . .) {p2 code}
      . . .
      initialization(. . .)
            {initialization code}
}
```

The only way to manipulate the shared internal variables in the monitor is by calling one of the procedures/functions/methodes p1, p2, …

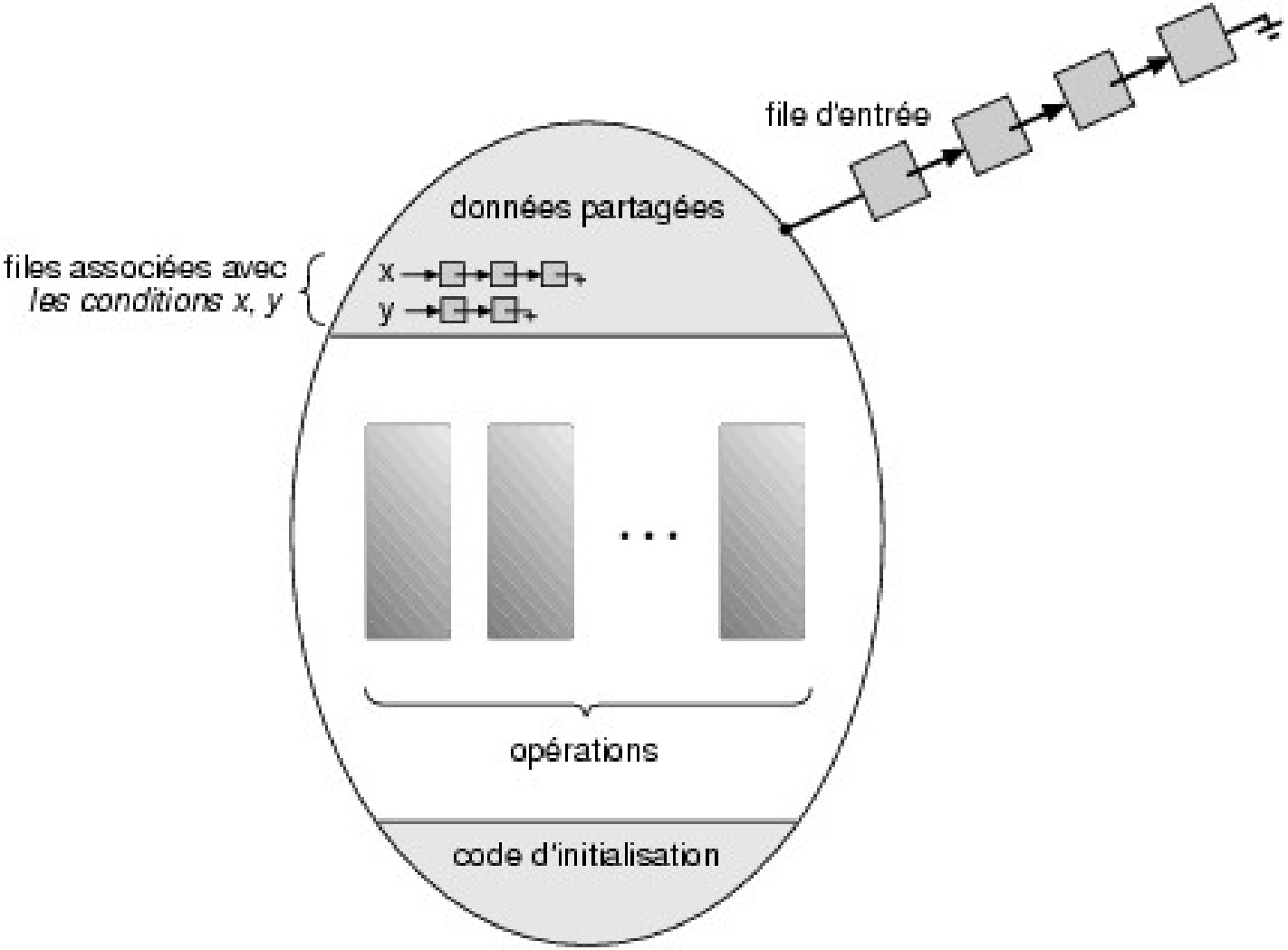
Monitor: schematic view



Condition variables

- are local to the monitor (accessible only within the monitor)
- can be access 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 one based on queuing scheme (e.g. FCFS, priority - see section 6.7.4 of the textbook)
    - If no such process exists: do nothing

Monitor with Conditional Variables

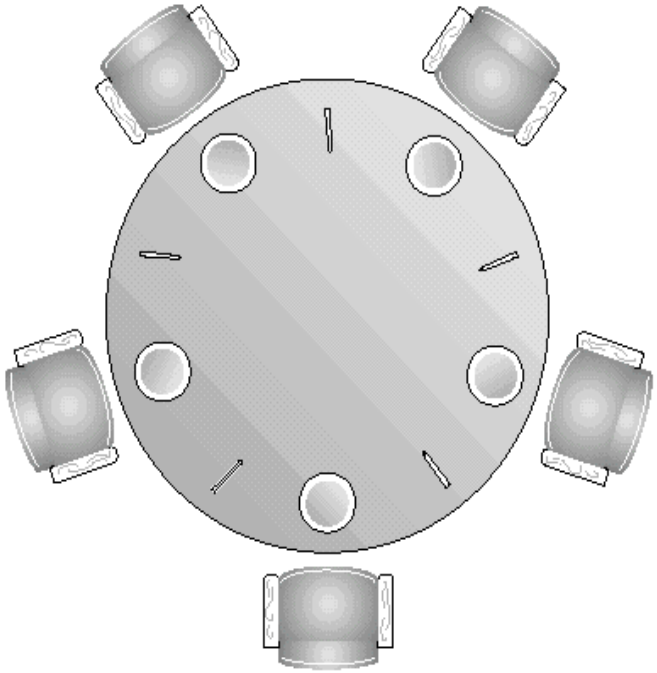


A Problem with signal()

- 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

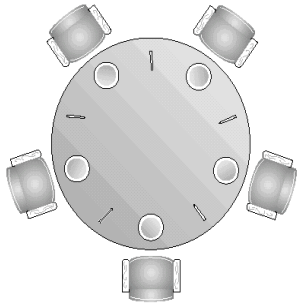
Back to the Dining Philosophers Problem

- 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



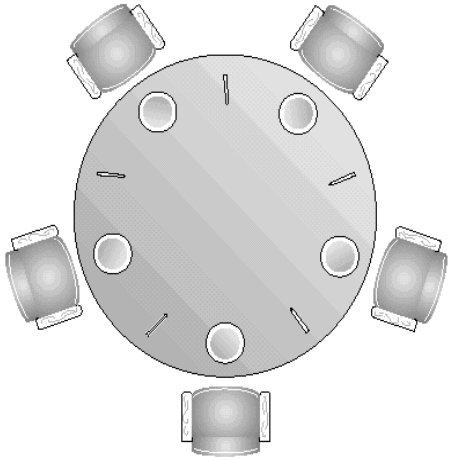
Dinning Philosophers: A Monitor

- 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 iif 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 in the monitor
  - 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

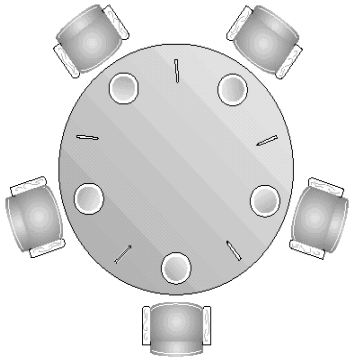


Philosopher i executes the loop:

```
while(true)
{
   think
   dp.pickup(i)
   eat
   dp.putdown(i)
}
```



A philosopher eats



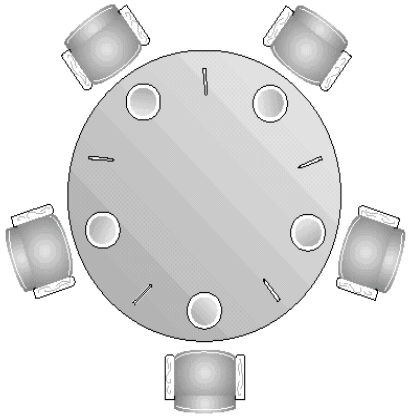
```
void test(int i)
{
    if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) )
    {
        state[i] = EATING;
        self[i].signal();
    }
}
```

A philosopher eats if his/her neighbors do not eat an 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.

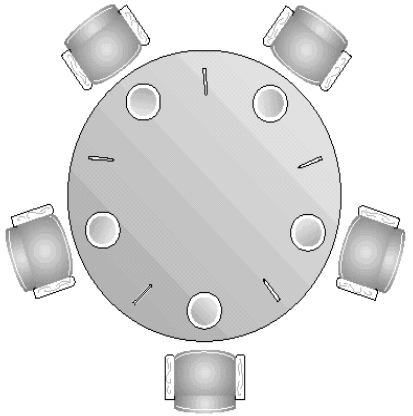
Pickup Chopsticks



```
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



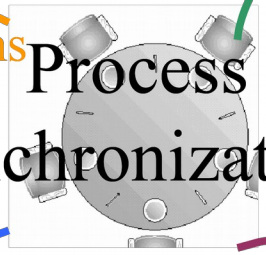
```
void putDown(int i)
{
        state[i] = THINKING;
            // test the two neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
}
```

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

An ingenious solution – note that the chopsticks are implicit.

Worth studying.

Process Synchronization



Monitors

Semaphores

Hardware Solutions

Peterson's Solution (Software)

Solutions

Background Producer /Consumer

What is the problem

?

Critical Section Problem

Mutual Exclusion

Progress

Bounded Waiting

Classic problems

Bounded Buffer

Reader Writer

Dining Philosophers

In

Synhr Real Systems

Java

Windows XP

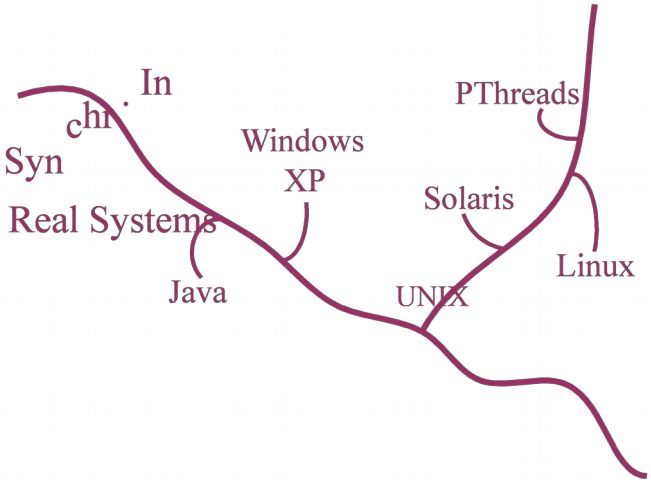
UNIX

Solaris

PThreads

Linux

Java Class Semaphore



- Counting semaphore
- Characteristics :
  - Can choose to use FIFO or not.
  - Semaphore value represents a number of available permits
  - `release()` increments this value – adds permits.
  - `acquire()` blocks if no permit is available.
  - In principle the number of permits is never negative.
    - One exception, it is possible to initialize the number of permits to a negative value.

Semaphore Constructors

```
Semaphore(int permits)
Semaphore(int permits, boolean fair)
```

- **The parameter `permits`: defines the initial number of permits.**
- **The parameter `fair`: If `fair` is `true` the FIFO discipline is used with the semaphore queue.**
- **The first constructor gives a value of `false` to the parameter `fair`, that is, does not use FIFO with the queue.**

The method `acquire()`

- This method plays the same role as the function wait() studied in class.
- Verifies the number of available permits in the semaphore.
  - If a permit is available (semaphore value > 0), the number of semaphores is decremented and the method returns; the thread can continue executing.
  - If no permit is available (semaphore value <= 0), the thread blocks and is placed on a queue.
    - When the permit becomes available, a thread in the queue is removed and allowed to consume the permit.

The method `acquire()`

- This method will throw an exception when a thread is interrupted (with the method `interrupt()`).
  - It throws `InterruptedException`.
  - Use the method in a `try/catch` structure:

```
try{sem.acquire();}
catch (InterruptedException e) { }
```

  - Add `break;` as shown below to break out of a loop.

```
try{sem.acquire();}
catch (InterruptedException e) { break; }
```

The method `acquireUninterruptibly()`

- This method provides the same functionality as `acquire()`,
- But it does not throw an exception with the thread is interrupted.
- Not necessary to use the `try/catch` with this method.
- Use the method with threads that are not to be interrupted.

The method `release()`

- This method plays the same role as the function `signal()`.
  - It increments the number of permits in the semaphore (i.e. its value).
  - If threads blocked on the semaphore, releases a thread that is blocked on the semaphore to consume the permit

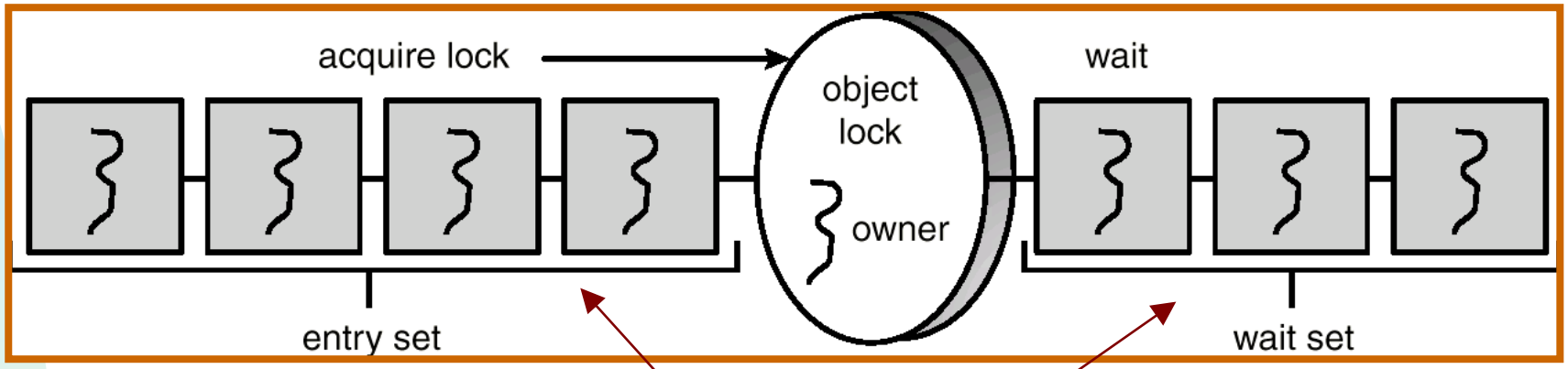
Other methods in Semaphore

- See Java documentation for more details on other methods available in Semaphore.
  - Other versions of `acquire()`: `acquireUninterruptibly()`, `tryAcquire()`, and `release()`
  - `availablePermits()`
  - `drainPermits()` and `reducePermits()`
  - `isFair()`
  - `hasQueuedThreads()`, `getQueueLength()`, `getQueuedThreads()`

A simple monitor in Java

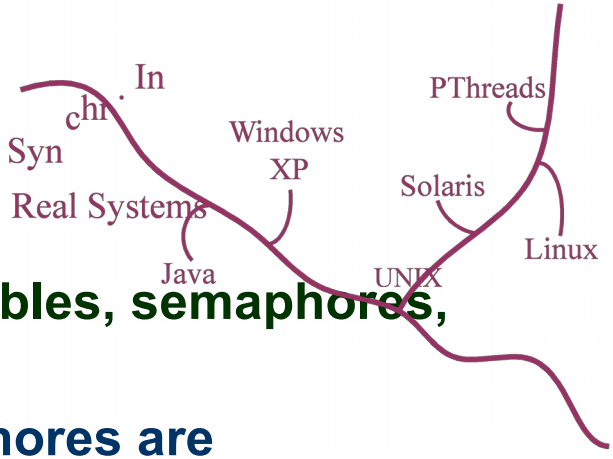
- Objects with synchronized methods in Java are essentially monitors (although simple)
  - A single thread at a time can execute such methods.
- Two queues are defined for each Java object:
  - Entry set
  - Wait set
- An object can only have a single wait set (queue)
  - Important limitation that can complicate things in Java…
- Wait in Java is more or less the same as wait() described for monitors
- Signal is called notify
  - Notify() frees 1 single thread
  - NotifyAll frees all threads
  - Freed threads are not executed: they are placed in the entry set.

Java Entry Sets and Wait Sets



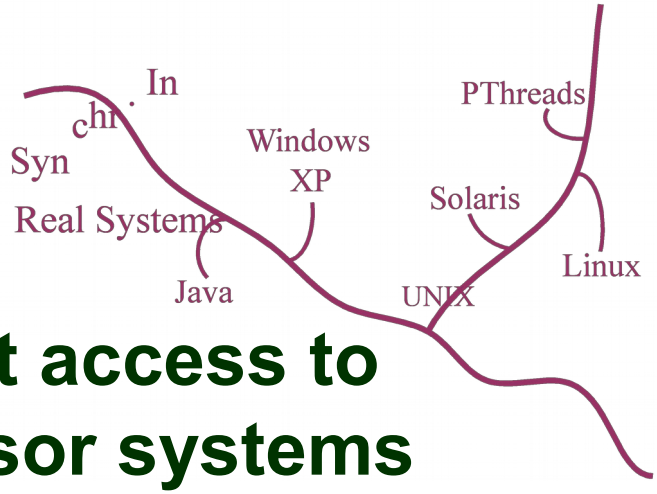

```
public class SimpleClass
{
      …
   public synchronized void SynchrMethod()
     {
           …
           wait();
           …
     }
}
```

Solaris Synchronization



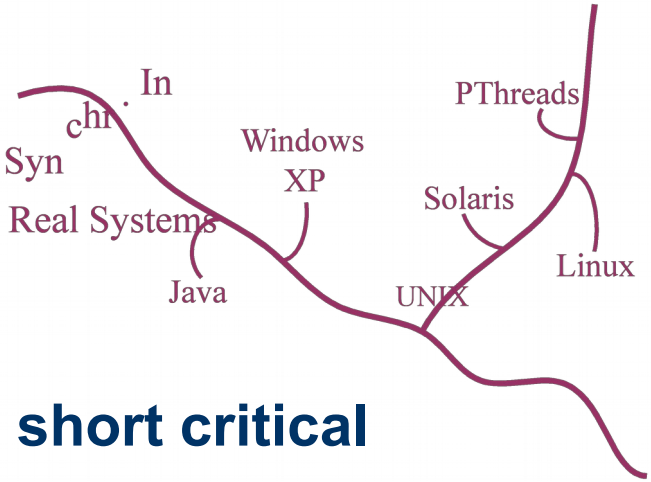
- Implements adaptive mutexes, conditional variables, semaphores, reader-writer locks and turnstiles
  - Conditional variables (monitors) and semaphores are implemented as presented in this module
- Uses adaptive mutexes for efficiency when protecting data from short code segments
  - Starts as a spinlock if thread holding lock is likely to finish
  - Blocks if thread holding lock is not running
- Uses readers-writers locks for accessing data frequently for read only purposes
  - Expensive to implement – used on long sections of code.
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Queues associated to threads rather than synchr. objects
  - When a thread is the first object to block on an object, its turnstile becomes the blocking queue for the object

Windows XP Synchronization



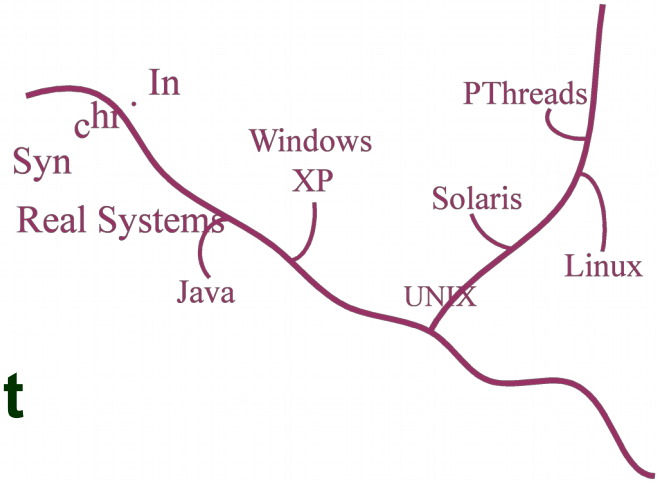
- Uses interrupt masks to protect access to global resources on uniprocessor systems
  - Uses spinlocks on multiprocessor systems
  - Threads holding a spinlock is never preempted.
- Also provides dispatcher objects for synchronization outside the kernel
  - Provides different mechanisms including mutexes, semaphores, events and timers.
  - Event: acts much like a condition variable

Linux Synchronization



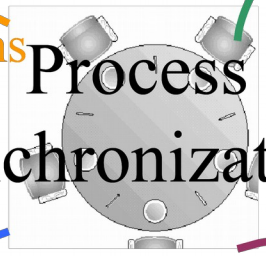
- On single processor systems:
  - disables interrupts to implement short critical sections
- On SMP systems
  - The spinlock is the fundamental locking mechanism.
- Spinlocks/disabling interrupts are mechanisms used for holding lock during short periods.
- Linux also provides semaphores:
  - Also reader-writer version is available.

Pthreads Synchronization



- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
  - read-write locks
- Non-portable extensions include:
  - semaphores
  - spin locks

Process Synchronization



Monitors

Semaphores

Hardware Solutions

Peterson's Solution (Software)

Solutions

Background Producer /Consumer

What is the problem ?

Critical Section Problem

Mutual Exclusion

Progress

Bounded Waiting

Classic problems

Bounded Buffer

Reader Writer

Dining Philosophers

In Synhr Real Systems

Windows XP

PThreads

Solaris

Linux

Java

UNIX