Synchronization

Sanzheng Qiao

Department of Computing and Software

December, 2012

Introduction

Cooperating processes:

The state of one process is shared by another.

- Behavior is nondeterministic: depends on relative execution sequence and cannot be predicted *a priori*.
- Behavior may be irreproducible.

Eg. One process writes “ABC” to the terminal, another wites“CBA”.

Introduction

Why permit processes to cooperate?

Shared resources; want to do things fast (parallel computing).

Basic assumption: the order of some operations is irrelevant.

Examples

- $A = 1$; $B = 2$; same as $B = 2$; $A = 1$.
- $A = B + 1$; $B = 2B$; cannot be re-ordered.
- $A = 1$ and $A = 2$ are in parallel, *race condition*.

If the exact order must be imposed, then there is no point in having multiple processes. Just put everything in one process.

Critical section

The section of program in which a shared variable is accessed.
Example. A joint bank account. Shared variable: bal
Deposit

```
Deposit:                    Withdraw:
  input    dep                input    withd
  load     dep                load     withd
  load     bal                load     bal
  add      bal, dep           sub      bal, withd
  store    bal                store    bal
```

Critical section

An execution sequence.

```
     Deposit                 Withdraw

   load    dep
   load    bal
   add     bal, dep
                          load    withd
                          load    bal
                          sub     bal, withd
                          store   bal
   store bal
```

Requirements for a solution

Requirement 1 (mutual exclusion):

If p_i is executing in its critical section, then no other process can execute in its critical section.

Requirements for a solution

Requirement 1 (mutual exclusion):

If p_i is executing in its critical section, then no other process can execute in its critical section.

A solution:

```
Algorithm A (symmetric for B)
    common variable:
    $TURN: (B, A)$;
    **repeat**
        **while** $TURN \neq A$ **do** skip;
            ⟨critical section⟩
        $TURN = B$;
            ⟨remainder section⟩
    **until** false.
```

Requirements for a solution

Prove mutual exclusion:
When A (B) remains in its critical section, $TURN = A(B)$

When A is entering its critical section ($TURN = A$), B is not in its critical section;

While A remains in its critical section ($TURN = A$), B cannot enter its critical section;

Requirements for a solution

Prove mutual exclusion:
When A (B) remains in its critical section, $TURN = A(B)$

When A is entering its critical section ($TURN = A$), B is not in its critical section;

While A remains in its critical section ($TURN = A$), B cannot enter its critical section;

Problem: strict alternation.

Requirements for a solution

Requirement 2 (progress):

If no process is executing in its critical section and there exist some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision as to who will enter the critical section and the selection cannot be postponed indefinitely.

Requirements for a solution

A solution (progress):

```
New Algorithm A
    common variable:
    $AFlag, BFlag$:
        $(ON, OFF)$;
    **repeat**
1       **while** $BFlag = ON$ **do** skip;
2       $AFlag = ON$;
        ⟨critical section⟩
3       $AFlag = OFF$;
        ⟨remainder section⟩
    **until** false.
```

Symmetric for B.

Requirements for a solution

Prove progress:

If A is in its remainder section (A doesn't want to enter its critical section), B can always enter its critical section. This is simple since we have only two processes. The definition is general.

Requirements for a solution

Prove progress:

If A is in its remainder section (A doesn’t want to enter its critical section), B can always enter its critical section.
This is simple since we have only two processes. The definition is general.

Problem: no guarantee of mutual exclusion. Consider the sequence:
A1, A2, A3, A1, B1, B2, A2, ...

Requirements for a solution

Another solution:

```
New New Algorithm A
    common variable:
    $BFlag, AFlag$:
        $(ON, OFF)$;
    **repeat**
1       $AFlag = ON$;
        **while** $BFlag = ON$ **do** skip;
        $\langle$critical section$\rangle$
        $AFlag = OFF$;
        $\langle$remainder section$\rangle$
    **until** false.
```

Prove: Mutual exclusion; Progress.

Requirements for a solution

Another solution:

```
New New Algorithm A
    common variable:
    $BFlag, AFlag$:
        $(ON, OFF)$;
    **repeat**
1       $AFlag = ON$;
        **while** $BFlag = ON$ **do** skip;
        ⟨critical section⟩
        $AFlag = OFF$;
        ⟨remainder section⟩
    **until** false.
```

Prove: Mutual exclusion; Progress.

Problem: possible deadlock. Consider sequence: A1, B1, ...

A correct solution

```
Correct Algorithm A
        common variable:
        $BFlag, AFlag$:
              $(ON, OFF)$;
        $TURN$: $(B, A)$;
        **repeat**
              $AFlag = ON$;
              $TURN = B$;
              **while** $BFlag = ON$
                    and $TURN = B$ **do** skip;
              ⟨critical section⟩
              $AFlag = OFF$;
              ⟨remainder section⟩
        **until** false.
```

A correct solution

We can prove:

Mutual exclusion;
Progress;
No deadlock (*TURN* is either *A* or *B*).

A correct solution

We can prove:

Mutual exclusion;
Progress;
No deadlock (*TURN* is either *A* or *B*).

Shortcoming: possible starvation.

Requirements for a solution

A desirable property (bounded waiting):

There must exist a bound 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 the request is granted.

Requirements for a solution

A desirable property (bounded waiting):

There must exist a bound 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 the request is granted.

Problems with the solution:

Hard to extend to *n* processes;
Busy wait: a process repeatedly uses the cpu time to check conditions.

Requirements for a solution

A mutual exclusion mechanism:

Mutual exclusion: Only one process in its critical section at a time.

Progress: Allow vacation outside critical section.

No deadlock: If several requests at once, must allow one process to proceed.

Desirable properties:

Fair: Bounded waiting.
Efficient: no busy waiting.
Simple: easy to use.

Atomic operations

An *atomic operation* either happens in its entirety without interruption, or not at all. Cannot be interrupted in the middle.

Eg. suppose printf is atomic, what is output of: printf(“ABC”); printf(“CBA”);?

Atomic operations

An *atomic operation* either happens in its entirety without interruption, or not at all. Cannot be interrupted in the middle.

Eg. suppose printf is atomic, what is output of: printf(“ABC”); printf(“CBA”);?

References and assignments are atomic in almost all systems. $A = B$ will always get a good value for B and set a good value for A (not arrays, records).

In uniprocessor systems, anything between interrupts is atomic.

Building atomic operations

If you don’t have any atomic operation, you can’t make one. Fortunately, the hardware guys give us atomic ops.

If you have an atomic op, you can use it to generate higher-level constructs and make concurrent processing work correctly. This is the approach we’ll take in this class.

Lock and condition

A high-level mechanism (built upon a lower-level one).

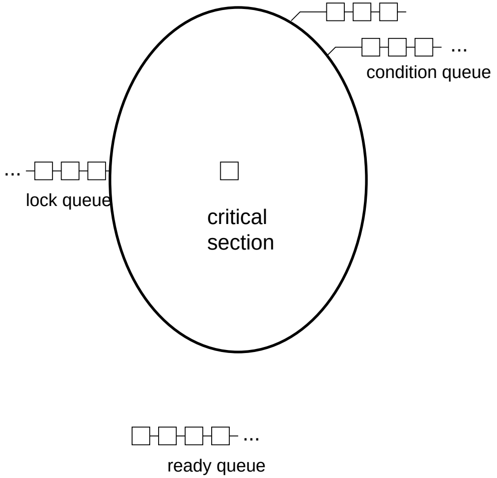
Lock: A synchronization variable that takes two values (BUSY, FREE).

Acquire: An atomic operation that waits until the lock is FREE, then sets the lock BUSY.

Release An atomic operation that wakes up a thread waiting in Acquire if necessary, then sets the lock to FREE.

Condition variable (*always* associated with a lock).
Wait: Releases the lock then waits on the condition. When signaled, re-acquires the lock.
Signal: If there are any waiting on the condition, wakes one up.

Lock and condition



Lock and condition

Usage of lock and condition.

- Always acquire the lock before manipulating shared data.
- Always release the lock after manipulating shared data.
- Do not lock again if the lock is held by the current process.
- Do not unlock if the lock is not held by the current process.

Semaphores

A synchronization variable that takes non-negative integer values.
(Edsger Dijkstra, mid 1960s)

Atomic operations:

- P(): waits for semaphore to become positive, then decrements it by 1
  (“proberen” in Dutch).
- V(): increments semaphore by 1
  (“verhogen” in Dutch).

Semaphores

Solving critical section problem using semaphore

```
semaphore->P()
  <critical section>
semaphore->V()
  <remainder section>
```

Note: initialization of semaphore.

Show mutual exclusion, progress, no starvation (?).

Semaphores

Solving critical section problem using semaphore

```
semaphore->P()
  <critical section>
semaphore->V()
  <remainder section>
```

Note: initialization of semaphore.

Show mutual exclusion, progress, no starvation (?).

Semaphores are elegant. They do a lot more than just mutual exclusion.

Semaphores

Semaphores are not provided by hardware.

Attractive properties of semaphore:

- machine independent
- simple
- work with many processes
- can have many different critical sections with different semaphores
- can acquire many resources simultaneously (multiple P's)
- can permit multiple processes into critical sections, if that is desirable

Using semaphores

Semaphore can be used in two different ways:

- Mutual exclusion: to ensure that only one process is accessing shared information at a time. If there are separate groups of data that can be accessed independently, there may be separate semaphores, one for each group of data. These semaphores are always binary.
- Scheduling: to permit processes to wait for certain things to happen. If there are different groups of processes waiting for different things, there will usually be a different semaphore for each group of processes. These semaphores are not necessary binary.

Using semaphores

Suppose that Pa runs A and Pb runs B and we want to schedule them so that A runs first.

Initialize the semaphore value to 0.

```
In Pa:
       A;
       V();


In Pb:
       P();
       B;
```

Semaphores

A solution for busing waiting:

A waiting queue. (So a semaphore has a value and a queue.)

```
$P()$:
    if $value \leq 0$
        add the calling thread to the waiting queue;
        the calling thread goes to sleep;
    else
        decrement the value by 1;

$V()$:
    if the waiting queue is not empty
        move a thread from the waiting queue to ready queue;
    else
        increment the value by 1;
```

Implementing lock using semaphore

Private:

```
char* name;
Thread* holder;
Semaphore* lockSem;


Lock::Lock(char* debugName)
{
    name = debugName;
    holder = NULL;
    lockSem =
       new Semaphore("Lock Sem", 1);
}
```

Implementing lock using semaphore

```
Lock::Acquire()
{
    ASSERT(holder != currentThread);

    lockSem->P();
    holder = currentThread;
}

Lock::Release()
{
    ASSERT(holder == currentThread);

    holder = NULL;
    lockSem->V();
}
```

Implementing condition using semaphore

```
Condition::Condition(char* debugName)
{
    name = debugName;
    condQueue = new List<Semaphore *>;
}
```

Implementing condition using semaphore

```
Condition::Wait(Lock* conditionLock)
{
    Semaphore *waiter;

    ASSERT(conditionLock->
           isHeldByCurrentThread());

    waiter = new Semaphore("condition", 0);
    condQueue->Append(waiter);
    conditionLock->Release();
    waiter->P();
    conditionLock->Acquire();
    delete waiter;
}
```

Implementing condition using semaphore

```
Condition::Signal(Lock* conditionLock)
{
    Semaphore *waiter;

    ASSERT(conditionLock->
           isHeldByCurrentThread());

    if (!condQueue->IsEmpty()) {
        waiter = condQueue->RemoveFront();
        waiter->V();
    }
}
```

Implementing condition using semaphore

```
Condition::Broadcast(Lock* conditionLock)
{
    while (!condQueue->IsEmpty()) {
        Signal(conditionLock);
    }
}
```

Example: Producer and consumer problem

Initialization: empty.value = bufferSize; full.value=0.

Producer:

```
            produce an item;
            empty→P();
            mutex→P();
            ENQ;
            mutex→V();
            full→V();
```

Consumer:

```
            full→P();
            mutex→P();
            DEQ;
            mutex→V();
            empty→V();
            consume the item;
```

Producer and consumer problem

Study the following:

- Two different ways of using semaphores (mutual exclusion and scheduling).
- Why does producer empty→P()
  but full→V()?
- Why is the order of P's important?
- Is the order of V's important?

Semaphore Implementation

- No existing hardware implementations of P() and V() directly. Thus semaphore must be built up in software using some lower-level synchronization primitive provided by hardware.
- Uniprocessor solution: disable interrupts (remember how dispatcher regains control).

Eg. Nachos implementation.

A hardware synchronization primitive

An Atomic Function: *test-and-set*()

```
**function** $test\text{-}and\text{-}set(target)$
**begin**
        $test\text{-}and\text{-}set := target;$
        $target := true$
**end.**
```

Solving critical section problem using *test-and-set*()

```
$lock$ := false;
repeat
        while $test\text{-}and\text{-}set(lock)$ do skip;
                ⟨critical section⟩
        $lock$ := false;
                ⟨remainder section⟩
until false;
```

Hardware implementation

Machine instruction: $\mathrm{TSTSET}\ \ \mathit{Reg}, \mathit{Var}, \#$

Solving Critical Section Problem Using TSTSET

```
LOOP:    TSTSET     7, lock, 1
         JMPONE     7, LOOP
         ⟨critical section⟩
         MOVE       lock, 0
         ⟨remainder section⟩
```

Problem: busy waiting

Semaphores in a multiprocessor system

Can't just turn off interrupts to get low-level mutual exclusion. Have to be busy-waiting at some level.

Data structure:

```
typedef struct {
    int v;
    list q;
    int t;
} SEMAPHORE;
```

- Disable interrupts to ensure atomicity on this processor;
- Spinlock (busy waiting) to ensure mutually exclusive access to the semaphore value (over all processors);
- Spinlock is useful when locks are expected to be held for short time.

Semaphores in a multiprocessor system

```
$P(s)$
$SEMAPHORE *s;$
{
    disable interrupts;
    **while** ($test\text{-}and\text{-}set(s \rightarrow t) \neq 0$);
    **if** ($s \rightarrow v > 0$) {
        $s \rightarrow v = s \rightarrow v - 1;$
        $s \rightarrow t = 0;$
    } **else** {
        add process to $s \rightarrow q$;
        $s \rightarrow t = 0;$
        go to sleep;
    }
    restore interrupts;
}
```

Semaphores in a multiprocessor system

```
$V(s)$
$SEMAPHORE *s;$
{
    disable interrupts;
    **while** ($test$-$and$-$set(s \rightarrow t) \neq 0$);
    **if** ($s \rightarrow q$ empty) {
        $s \rightarrow v += 1;$
    } **else** {
        remove a process from $s \rightarrow q$;
        wake it up;
    }
    $s \rightarrow t = 0;$
    restore interrupts;
}
```

Language Constructs

General form:

```
class *name*
        variable declarations
entry $P1(\cdots)$
        begin
           ⋮
        end
        $\cdots$
begin
        initialization
end
```

Monitor

A high-level abstraction.
Monitors combine:
Shared data.
Operations on the data.
Synchronization, scheduling.

Existing implementations of monitors are embedded in programming languages.

There is a lock (or binary semaphore) associated with each monitor, mutual exclusion is implicit: P on entry to any procedure, V on exit.

Monitor

Condition variables: things to wait on (scheduling).

Operations on a condition variable:
Wait(): release monitor lock; put process to sleep; when the process wakes up, re-acquire monitor lock immediately.
Signal(): wake up exactly one process suspended by Wait(). If no process is suspended, Signal() has no effect.
Broadcast(): wake up all processes waiting on the condition variable. If no process is waiting, do nothing.

Note:

- There are several different variations on the wait/signal mechanism, in terms of who (signaler, or awakened, or else) gets the monitor lock after a signal.
- Compare wait/signal with P/V.
- Semaphores use a single structure for both mutual exclusion and scheduling, monitors use separate structures.
- Complex synchronization code is separated from other code and put in monitor.

Example: Java

Every object is associated with a lock.
A method can be declared as synchronized.

```
public synchronized void
ENQ(Object item) {
   ...
}
```

Two Java methods: `wait()` and `notify()`
Every object is also associated with a `wait set` containing the threads waiting on the condition.
Java allows only one condition variable.

An implementation of monitor

```
entry (procedure):
                    P(*mutex*)
                    {procedure}
                    **if** *next-count* $> 0$
                      V(*next*)
                    **else**
                      V(*mutex*)
```

mutex: monitor lock (binary semaphore)
next: semaphore for the urgent queue
next-count: number of processes on the urgent queue

An implementation of monitor

condition variable: x;

wait(x)(Hoare style):

```
    $x.count = x.count + 1$
    **if** $next\text{-}count > 0$
        V($next$)
    **else**
        V($mutex$)
    P($x.sem$);
    $x.count = x.count - 1$
```

$x.count$: number of processes waiting on x.
$x.sem$: semaphore (scheduling) associated with x.

An implementation of monitor

signal(x)(Hoare style):

```
    if $x.count > 0$
        $next\text{-}count = next\text{-}count + 1$
        V($x.sem$);
        P($next$);
        $next\text{-}count = next\text{-}count - 1$
```

Example: Bounded buffer problem

```
MONITOR consumer-producer
        $BufferSize = n$;
        circular array $buffer[BufferSize]$;
        index $head \leftarrow 1$, $tail \leftarrow 1$;
        int $full \leftarrow 0$;
        condition $vacant, avail$;
```

Boundid buffer problem

```
**entry** $ENQ(item, buffer)$;
    **if** $full = BufferSize$
        $vacant.wait$;
    $buffer(tail) = item$;
    $tail = (tail + 1) \bmod BufferSize$;
    $full = full + 1$;
    $avail.singal$;
```

Boundid buffer problem

```
**entry** $DEQ(item, buffer)$;
        **if** $full = 0$
           $avail.wait$;
        $item = buffer(head)$;
        $head = (head + 1) \bmod BufferSize$;
        $full = full - 1$;
        $vacant.singal$;
```

producer: $ENQ(item, buffer)$
consumer: $DEQ(item, buffer)$

Example: Readers and writers problem

Consider a shared file. Many readers can access the file simultaneously. If a writer is updating the file, no others can access the file.

Conditions:

(i) A reader shouldn't be permitted to start if there is a writer waiting for the currently active reader to finish. (avoid indefinite postponement of writers)
(ii) All readers waiting at the end of writer execution should be given priority over the next writer. (avoid indefinite postponement of readers)

Readers and writers problem

```
MONITOR readers-writers
        int $act\text{-}rd \leftarrow 0$;
        boolean $act\text{-}wt \leftarrow$ false;
        condition $wait\text{-}rd$, $wait\text{-}wt$;
```

If there is an active writer or a waiting writer, the reader waits. Once activated, it signals other waiting readers to become active.

```
entry start-read
    if $act\text{-}wt$ or not $EMPTY(wait\text{-}wt)$
      $wait\text{-}rd.wait$;
    $act\text{-}rd = act\text{-}rd + 1$;
    $wait\text{-}rd.signal$;
```

Readers and writers problem

If finishing reader finds that it is the last active reader, it signals a waiting writer.

```
**entry** end-read
        $act\text{-}rd = act\text{-}rd - 1;$
        **if** not $EMPTY(wait\text{-}wt)$
           **if** $act\text{-}rd = 0$
              $wait\text{-}wt.signal;$
        **else**
           $wait\text{-}rd.signal;$
```

Readers and writers problem

If there are active readers or if there is an active writer, the new writer waits.

```
entry start-write
    if $(act\text{-}rd \neq 0$ or $act\text{-}wt)$
        $wait\text{-}wt.wait$;
    $act\text{-}wt =$true;
```

Readers and writers problem

If there are readers waiting, the finishing writer signals a reader. Otherwise, it signals another writer.

```
entry end-write
    $act\text{-}wt$ = false;
    if not $EMPTY(wait\text{-}rd)$
        $wait\text{-}rd.signal$;
    else
        $wait\text{-}wt.signal$;
```

Readers and writers problem

```
reader:
        **repeat**
                start-read;
                <reading>
                end-read;
        **until** false
writer:
        **repeat**
                start-write;
                <writing>
                end-write;
        **until** false
```

Deadlocks

Resource classification:

preemptible (resources that can be taken away from a process before that process has finished using it), for example, CPU. nonpreemptible, for example, disks and printers.

Resource allocation:

- Static: A process must acquire all the resources it might possibly use before that process is scheduled (predict the future, impractical).
- Dynamic: A process acquires a resource when the resource is needed.

Starvation and deadlock

Starvation:

A state when a process is continuously denied access to a resource. The process may be able to access to the resource later, but not sure when (eg. bounded waiting is not satisfied).

Deadlock:

A state when every process in a set is waiting for an event that can only be caused by another process in the set. Since all are waiting, none can cause the even to happen (eg. two processes one does P(x); P(y); the other does the reverse).

Necessary Conditions

- Mutual exclusion: resources cannot be shared.
- Hold and wait: processes don't ask for resources all at once. So a process can ask for resources while holding some resources.
- No preemption: onece given, a resource cannot be taken away until the process finishes using it.
- Circular wait: there is circularity in the resource allocation graph.

Resource Allocation Graph

A directed graph:

$G = (V, E)$
$V = \{p_i\} \cup \{r_i\}$
p_i: process
r_i: resource type (may have several identical units)
$E = \{(p_i, r_j)\} \cup \{(r_i, p_j)\}$
(p_i, r_j): request edge
(r_i, p_j): assignment edge

A resource graph is reduced by p_i which is neither blocked nor isolated by removing all edges to and from p_i.

Deadlock prevention

Organize the system so that it is impossible for deadlock ever to occur.

Solution: Breaking any one of the four necessary conditions

1. Mutual exclusion: Don't allow exclusive access. This is probably not reasonable for many applications.

2. Hold and wait: Don't allow waiting.

- Request all before execution.
- Release all before requesting.

This probably causes starvation.

Deadlock prevention

3. No preemption: Allow preemption.

- A process holding resources and requests more but has to wait, all resource held by the process are preempted.
- A process requests a resource but the resource is held by another process which is waiting for a resource. The resource is preempted from the another process.

4. Circular wait: A unique number is assigned to each resource type.

- request resources in strictly increasing order.
- before requesting r_j, release all r_i with higher numbers.

All the solutions are expensive and/or require predicting the future, not practical.

Deadlock detection

Determine when the system is deadlocked and then take drastic action (termination of processes).

A special case:
Each resource type has exactly one unit. Then deadlock in system if and only if cycle exits in the graph.

Algorithm

1. Delete all sink nodes (with only incoming edges);
2. Reduce the graph;
3. Check cycles.

In this special case, the graph can be simplified to a wait-for graph.

General case: Multiple instances

Algorithm

1. Find an unfinished process whose request can be satisfied by available resources;
2. Pretend to finish the process and return the resources held by the process to available;
3. Repeat 1 and 2 until no such processes can be found;
4. If there are unfinished processes, the system is in deadlock state.

Deadlock Avoidance

Determine when the system can be potentially deadlocked and then take action to avoid the possible deadlock.

Safe state:
A state is called safe, if it is possible for the system to satisfy all possible future requests in some order without deadlocks.

Banker’s Algorithm:
Pretend to satisfy the request and check if the system ends in safe state.

Example: A safe state

Total resources: [2 1 2].
Matrix *max*:

$$\begin{array}{cccc} & r_1 & r_2 & r_3 \\ p_1 & 2 & 0 & 0 \\ p_2 & 1 & 1 & 1 \\ p_3 & 0 & 1 & 1 \\ p_4 & 0 & 0 & 2 \end{array}$$

Example: A safe state

Matrices *alloc* and *need*:

$$\begin{array}{cccc} & r_1 & r_2 & r_3 \\ p_1 & 1 & 0 & 0 \\ p_2 & 1 & 0 & 0 \\ p_3 & 0 & 1 & 0 \\ p_4 & 0 & 0 & 1 \end{array} \qquad \begin{array}{cccc} & r_1 & r_2 & r_3 \\ p_1 & 1 & 0 & 0 \\ p_2 & 0 & 1 & 1 \\ p_3 & 0 & 0 & 1 \\ p_4 & 0 & 0 & 1 \end{array}$$

Availabe resources: $[0\ \ 0\ \ 1]$.

Example: An unsafe state

Total resources: [2 1 2].

Matrix *max*:

$$\begin{array}{cccc} & r_1 & r_2 & r_3 \\ p_1 & 2 & 0 & 0 \\ p_2 & 2 & 1 & 0 \\ p_3 & 0 & 1 & 1 \\ p_4 & 0 & 0 & 2 \end{array}$$

Example: An unsafe state

Matrices *alloc* and *need*:

$$\begin{array}{cccc} & r_1 & r_2 & r_3 \\ p_1 & 1 & 0 & 0 \\ p_2 & 1 & 0 & 0 \\ p_3 & 0 & 1 & 0 \\ p_4 & 0 & 0 & 1 \end{array} \qquad \begin{array}{cccc} & r_1 & r_2 & r_3 \\ p_1 & 1 & 0 & 0 \\ p_2 & 1 & 1 & 0 \\ p_3 & 0 & 0 & 1 \\ p_4 & 0 & 0 & 1 \end{array}$$

Availabe resources: [0 0 1].

The weaknesses of the algorithm:

- requires fixed number of resources to allocate;
- requires the population of users remain fixed;
- allows the banker grant all requests within a finite time;
- requires customers repay all loans within a finite time;
- requires users state their maximum needs in advance.

Conclusion

Deadlock is one area where there is a strong theory, but it is almost completely ignored in practice (ostrich approach).

Reason:

Solutions are expensive, and/or assume impractical conditions, and/or require predicting the future.