Resource Access Control

Radek Pelánek

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

EVROPSKÁ UNIE esf MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY OP Vzdělávání pro konkurenceschopnost UNIVERSITAS MASARYKIANA BRUNENSIS

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

Resources: notions

resource something needed to advance execution of a task; e.g. printer, file, database lock, ...

shared resource resource used by several tasks

mutually exclusive resource shared resource that can be used by only one task at a time

critical section piece of code executed under mutual exclusion constraint

Problems

1. how do we assure mutually exclusive access?
   (see also Operating systems, Parallel Algorithms)
   - mutual exclusion algorithms
   - semaphores
2. how to do scheduling with resources?
   - priority inversion problem
   - priority inheritance/ceiling protocol

Mutual Exclusion and This Course

- mutual exclusion – recurring theme
- today: overview of protocols
- later:
  - programming: exercises (implementation of protocols)
  - verification: basic example for explanation, exercises

Alice, Bob, and Pets

- Alice has a cat
- Bob has a dog
- they share a yard, cat and dog should not be in yard at the same time
- Alice and Bob cannot see the whole yard, but they see each others window
- device a "visual protocol" to ensure the "mutual exclusion" (using e.g. flags in windows)

Alice, Bob, and their First Attempt

Alice:

1. If there is no flag in Bob's window:
   - raise flag
   - unleash cat
2. When cat comes back, lower flag.

Bob:

1. If there is no flag in Alice's window:
   - raise flag
   - unleash dog
2. When dog comes back, lower flag.

Alice, Bob, and Flag Protocol

Alice:

1. Raise flag.
2. When Bob's flag is lowered, unleash cat.
3. When cat comes back, lower flag.

Bob:

1. Raise flag.
2. While Alice's flag is raised:
   - Lower flag.
   - Wait until Alice's flag is lowered.
   - Raise flag.
3. Unleash dog.
4. When dog comes back, lower flag.

Basic Setting

Several processes of the following type:

```
while (true) {
    <noncritical section>;
    <entry section>;
    <critical section>;
    <exit section>;
}
```

Requirements

1. mutual exclusion: only one process at a time in the CS
2. absence of deadlock: in every situation some process can make progress
3. absence of starvation (liveness): if a process wants to access CS, it will eventually be able to do so
4. a process that halts in its noncritical section must do so without interference with other processes

Assumptions

- each process spends only finite time in a critical section
- no assumptions about relative speed of processes
- process interleaving can happen at any point $\rightarrow$ protocol must work for any possible interleaving

Test-and-Set Instruction

- `testset(i)`: atomic instruction (hardware support): `if i = 0 then { i := 1; return true; } else return false;`
- shared variable busy (initial value 0)

```
while (true) {
   <noncritical section>;
   while not testset(busy) do {};
   <critical section>;
   busy := 0;
}
```

Importance of Atomicity

- what happens if the `testset` instruction is not atomic?
- which requirement is violated?
- find the execution which violates mutual exclusion

Software Realization

- now we discuss several software realizations of mutual exclusion
- at first we consider just 2 processes
- we start with wrong attempts – used to illustrate concepts

The First Attempt

shared variable `turn` (initial value 0)

```
Process 0:
while (true) {
    <noncritical section>;
    while turn != 0 do { };
    <critical section>;
    turn := 1;
}
```

```
Process 1:
while (true) {
    <noncritical section>;
    while turn != 1 do { };
    <critical section>;
    turn := 0;
}
```

The First Attempt: Discussion

- mutual exclusion: OK
- absence of deadlock: OK
- strict alternation of processes $\Rightarrow$ starvation
  if one process does not want to access CS or one process wants to access CS much more often than the other one, the protocol does not work (well)

The Second Attempt

shared variables `flag[0]`, `flag[1]` (initialised to `false`) – meaning *I'm in CS*

```
Process 0:
while (true) {
   <noncritical section>;
   while flag[1] do { };
   flag[0] := true;
   <critical section>;
   flag[0] := false;
}
```

```
Process 1:
while (true) {
   <noncritical section>;
   while flag[0] do { };
   flag[1] := true;
   <critical section>;
   flag[1] := false;
}
```

The Second Attempt: Discussion

same as non-atomic `testset`

- absence of starvation: OK
- absence of deadlock: OK
- mutual exclusion not satisfied

The Third Attempt

shared variables `flag[0]`, `flag[1]` (initialed to `false`) – meaning *I want to access CS*

```
Process 0:
while (true) {
   <noncritical section>;
   flag[0] := true;
   while flag[1] do { };
   <critical section>;
   flag[0] := false;
}
```

```
Process 1:
while (true) {
   <noncritical section>;
   flag[1] := true;
   while flag[0] do { };
   <critical section>;
   flag[1] := false;
}
```

The Third Attempt: Discussion

- absence of starvation: OK
- mutual exclusion: OK
- deadlock can occur

Peterson's Algorithm

- `flag[0]`, `flag[1]` (initialed to `false`) – meaning *I want to access CS*
- `turn` (initialized to 0) – used to resolve conflicts

```
Process 0:
while (true) {
   <noncritical section>;
   flag[0] := true;
   turn := 1;
   while flag[1] and
         turn = 1 do { };
   <critical section>;
   flag[0] := false;
}
```

```
Process 1:
while (true) {
   <noncritical section>;
   flag[1] := true;
   turn := 0;
   while flag[0] and
         turn = 0 do { };
   <critical section>;
   flag[1] := false;
}
```

Peterson's Algorithm: Discussion

- mutual exclusion: OK
- absence of starvation: OK
- absence of deadlock: OK

Can be extended for more than 2 processes (non-trivial).

Lamport's Bakery Algorithm

- protocol which works for n processes
- simulation of a "ticket system" at post office (bakery)
- process wants to access CS $\Rightarrow$ it is assigned the "next" ticket
- process with the lowest ticket is allowed to access CS
- non-atomicity of ticket assignment – requires special checking

Lamport's Bakery Algorithm

- number[i] – current ticket number
- choosing[i] – *I'm choosing my ticket number*

```
Process i:
while (true) {
   <noncritical section>;
   choosing[i] := 1;
   number[i] := 1 + max(number[0], ..., number[N-1]);
   choosing[i] := 0;
   for j:=0 to N-1 {
        while (choosing[j]) do {}
        while (number[j] != 0 and
                 (number[j], j) < (number[i], i)) do {}
   }
   <critical section>;
   number[i] := 0;
}
```

Fischer's Protocol

- real-time protocol – correctness depends on timing assumptions
- simple, just 1 shared variable, arbitrary number of processes
- assumption: known upper bound D on reading/writing variable in shared memory
- each process has it's own timer (for delaying)

Fischer's Protocol

- id – shared variable, initialized -1
- each process has it's own timer (for delaying)
- for correctness it is necessary that $K > D$

```
Process i:
while (true) {
   <noncritical section>;
   while id != -1 do {}
   id := i;
   delay K;
   if (id = i) {
      <critical section>;
      id := -1;
   }
}
```

Fischer's Protocol: Exercise

1. suppose $K < D$: find a run which violates mutual exclusion
2. suppose $K > D$: prove the correctness (advanced)

Alur and Taubenfeld's protocol

- Fischer's protocol: process delays even if it is the only trying to access CS
- Alur and Taubenfeld's protocol eliminates this waiting
- same assumptions as Fischer's protocol (particularly known D)
- `x`, `y` – shared int variables, `z` – shared boolean variable

Alur and Taubenfeld's protocol

```
Process i:
while (true) {
   <noncritical section>;
   start: x:=i;
   while (y != 0) do {}
   y := i;
   if (x != i) { delay 2*D;
                 if (y != i) goto start;
                 while (! z) do {}; }
   else {z := true; }
   <critical section>;
   z := false;
   if (y == i) y:= 0;
}
```

Semaphores

- operating system support for resource access control
- semaphore: initialized to non-negative value (typically 1)
- atomic operations `wait`, `signal`
  - decreasing/increasing value
  - blocking

Semaphores

- wait:
  - decrements the semaphore value
  - value becomes negative ⇒ the caller becomes blocked
- signal:
  - increments the semaphore value
  - value not positive ⇒ one process blocked by the semaphore is unblocked (usually in FIFO order)

How can we use semaphores for mutual exclusion?

Mutual Exclusion with Semaphores

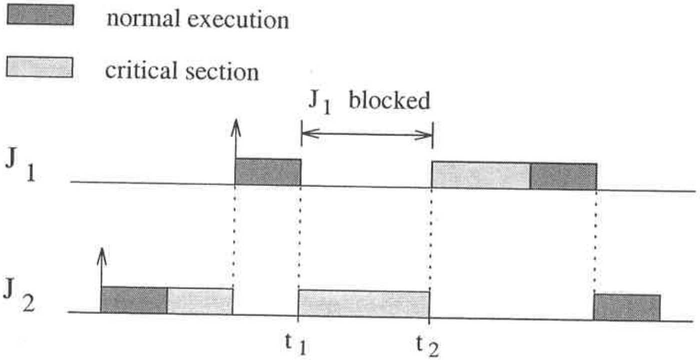
semaphore S (initialized to value 1)

```
while (true) {
    <noncritical section>;
    wait(S);
    <critical section>;
    signal(S);
}
```

Scheduling: The Problem

- we assume
  - fixed priorities of tasks – set by user or by some scheduling algorithm
  - correct algorithm for controlling access to critical section
- preemption
- resources
  - access to resources is only in critical sections
  - critical sections guarded by semaphores

Example of Blocking



Blocking on Critical Section

previous example:

- necessary blocking
- needed for ensuring mutually exclusive access
- bounded waiting, typically very short

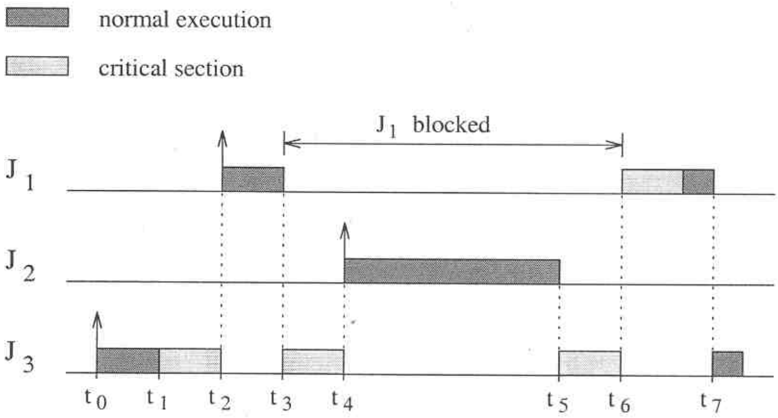
Priority Inversion Problem?

- suppose straightforward scheduling:
  - the ready process with highest priority is running
  - exception: waiting for access to critical section
- can the following happen?
  - process J_1 has higher priority than process J_2
  - J_1 is waiting
  - J_2 is running noncritical code

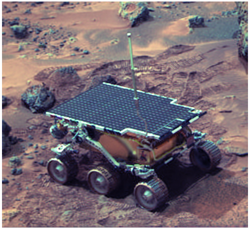
Priority Inversion Problem!

- tree jobs: J_1, J_2, J_3, priorities $p_1 > p_2 > p_3$
- J_1, J_3 share a resource R
- sample run:
  - J_3 acquires access to R
  - preempted by J_1; later J_1 wants access to R, thus J_1 is blocked, control returns to J_3
  - preempted by J_2
- a lower priority task (J_2) is running although a higher priority task (J_1) is blocked even through these two tasks do not have a conflict on a resource $\Rightarrow$ priority inversion
- priority inversion is potentially unbounded

Illustration



Mars Pathfinder



- unmanned spacecraft, landed on Mars in 1997
- frequent deadlocks ⇒ resets, loss of time

Mars Pathfinder

- information bus – shared resource
- tasks:
    - meteorological data gathering task – infrequent, low priority thread
    - communications task – medium priority
    - bus management task – frequent, high priority thread
- priority inversion ⇒ bus management task late ⇒ system watchdog assumes fatal error ⇒ system reset
- no data loss, but remainder of that day activities were not accomplished until the next day

Solution?

- how would you solve the problem?
- devise a scheduling protocol such that “priority inversion problem” does not occur

Solutions

- simple solution: non-preemptive critical sections
  disadvantage: a higher priority task can be unnecessary blocked by an “irrelevant” critical section of a lower priority task
- we will consider two more sophisticated solutions:
  - priority inheritance protocol
  - priority ceiling protocol

Priority Inheritance Protocol

The idea

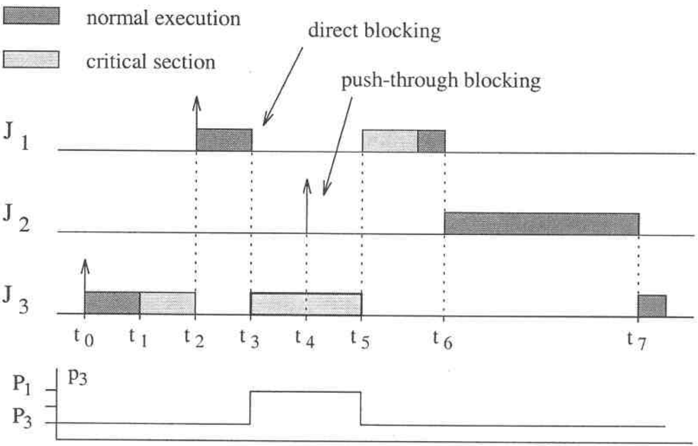
When a task blocks one or more higher-priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

realization non-trivial: (not just) nested critical sections

Protocol Definition

- basic scheduling based on priorities (FIFO)
- if job J_i tries to acquire a resource which is already used by a lower priority job J_k then:
  - J_i is blocked
  - J_k resumes and temporarily inherits priority of J_i
- when a job J_k releases a resource, then:
  - the highest priority job blocked on that resource (if there is any) is awakened
  - J_k assumes the highest priority of jobs still blocked by J_k, if there are none then J_k assumes its normal priority
- priority inheritance is transitive

Example: Blocking



Two Kinds of Blocking

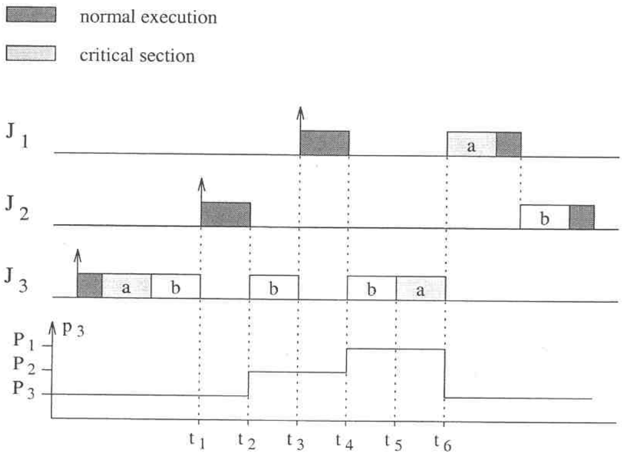
direct blocking

- high-priority job tries to acquire a resource already held by a lower-priority job
- necessary to ensure the consistency of shared resources

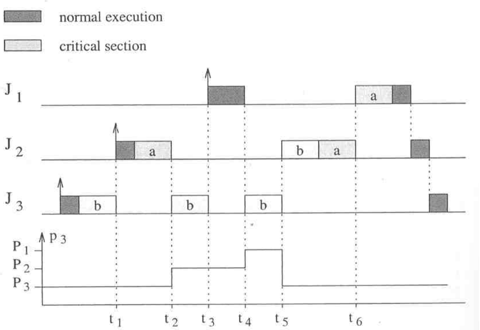
push-through blocking

- medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks
- necessary to avoid unbounded priority inversion

Example 2: Nested Critical Sections



Example 3: Transitive Inheritance



transitive inheritance can occur only in the presence of nested critical sections

Key Property

Priority Inheritance Protocol ensures bounded waiting for resources.

Implementation Considerations

- each tasks: nominal priority, active priority
- semaphore:
  - additional field: `holder` (identification of a process that has the lock)
  - `pi_wait()` – if locked then: store to the queue, inherit priority
  - `pi_signal()` – update active priority, if queue not empty then awaken the highest priority task in queue (update `holder`)

Priority Inheritance Protocol

Recapitulation:

The idea

When a task blocks one or more higher-priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

Example

	τ_1	τ_2	τ_3
C_i	$2(=1+1)$	$2(=2+0)$	$4(=0+4)$
T_i	6	8	12

- $C(=X+Y)$ means: X time units before critical section, Y time units in critical section
- priorities $\tau_1 > \tau_2 > \tau_3$, scheduling according to RM
- construct schedules:
  - without any protocol
  - with priority inheritance protocol

Deadlock

- another problem with (multiple) resources: possible deadlocks
- nested critical sections
- exercise: find the exact scenario
- priority inheritance protocol does not adress this issue
- priority ceiling protocol prevents both:
  - deadlocks
  - priority inversion

Priority Ceiling Protocol

The idea

The protocol does not allow a job to enter a critical section if there are locked semaphores that could block it.
Once a job enters its first critical section, it can never be blocked by lower-priority jobs until completion of the critical section.

Protocol Definition – Semaphores

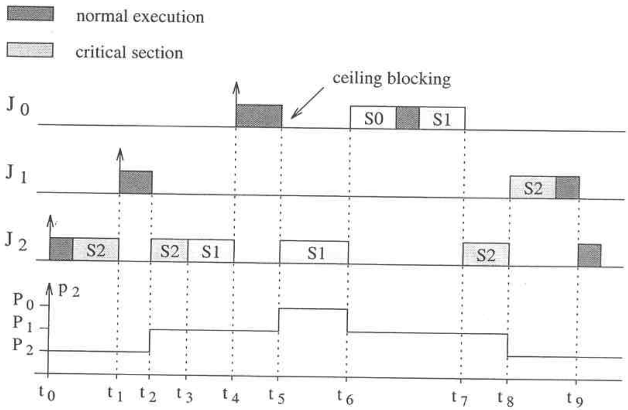
- access to critical sections controlled by semaphores
- each semaphore S_k is assigned a priority ceiling $C(S_k) =$ priority of the highest priority job that can lock it
- static value, can be computed off-line

Protocol Definition – Jobs

- let J_i be a ready job with the highest priority
- let $S*$ be the semaphore with the highest priority ceiling among all the semaphores locked by jobs other then J_i
- to enter any critical section, J_i must have priority higher then $C(S*)$; otherwise J_i becomes blocked
- when a job J_i is blocked, it transmits its priority to a job that holds the resource – details are similar to priority inheritance

Example

semaphore	S0	S1	S2
priority ceiling	$p(J_0)$	$p(J_0)$	$p(J_1)$



Properties

Priority Ceiling Protocol:

- ensures bounded blocking time of high priority jobs
- prevents deadlocks due to circular blocking of resources

Summary

1. how do we assure mutually exclusive access?
   - mutual exclusion, absence of deadlock, absence of starvation, ...
   - protocols: peterson, fischer, bakery, ...
   - semaphores
2. how to do scheduling with resources?
   - priority inversion problem
   - priority inheritance protocol
   - priority ceiling protocol