Operating Systems
2010/2011

Blocking and locking
(with figures from Bic & Shaw)

Johan Lukkien
Blocking & locking

- **Blocking:**
  - waiting for a certain condition to become true

- **Starvation:**
  - unpredictable, even infinite blocking times
  - the opposite of fairness
  - can be the result of interference of scheduling and blocking
    - typical, in real-time systems
  - can be the result of cooperation of several processes
    - e.g. the ‘dining philosophers’ example
    - perhaps in combination with a greedy scheduler
  - happens through livelock or other forms of blocking

- **Livelock**
  - repeated ‘trying’ to pass a critical condition without making progress
    - typically, associated with polling
  - results functionally in starvation or deadlock

- **Deadlock**
  - extreme case of starvation: continuation not possible
    - typically, a ‘cul-de-sac’ state of the system
Blocking times and scheduling: priority inversion

- A low priority task obtains a resource; a high priority task waits on it
- A middle priority task pre-empts the low priority task
  - the high priority task now waits on the middle priority task
  - ... and executes effectively at the low priority
  - ... unfairness, hence starvation danger
A solution: priority inheritance protocol

- The priority of the task $P$ using the resource is *dynamically adjusted* to be the maximum of
  - the priority of any other task that is blocked on the allocated resources of $P$
  - (... and its own priority)

- ...middle priority jobs will wait now.
Recall action synchronization: preventing deadlock

• The exercises A4, A7, A8, give the following insights for deadlock prevention

• Let critical sections terminate
  – in principle, no \( P \) operations between \( P(m)...V(m) \)

• Use a fixed order in \( P \)-operations on semaphores
  – \( P(m);P(n);... \) in one process may deadlock with \( P(n);P(m);... \) in another process
  – in fact: satisfy the synchronization conditions in a fixed order
    • this can be easily verified inside an operating system

• Beware of greedy consumers
  – Let \( P(a)^k \) be an indivisible operation when there is a danger of deadlock

*In general: avoid cyclic waiting!*
Deadlock: terminology

- Deadlock is usually associated with access to resources
  
  - consumable resources: resource use takes it away (variable number)
    - typical producer / consumer problems
    - example: characters typed at a keyboard, blocks of data from the network
  
  - reusable resources: resource is shared (fixed number)
    - typical mutual exclusion problems (consumer and producer are same process), readers/writers problems
    - example: processor, memory blocks, .... physical entities
Definitions

• **We call a task (process or thread) **blocked** if:**
  – it is waiting on a blocking synchronization action
  – or has terminated (for technical reasons included)

• **Definition:** a set $D$ of tasks with at least one not terminated is called **deadlocked** if
  – all tasks in $D$ are blocked, and
  – for each non-terminated task $t$ in $D$, any task that might unblock $t$ is also in $D$
Deadlock: conditions

- Conditions for deadlock to be possible
  - mutual exclusion
  - hold and wait
    - “greediness”, several resources of the same type as well as several of different types are reserved incrementally
  - no preemption
  - circular waiting

- These all play a role and can be addressed explicitly in the solution
  - i.e., deadlock is addressed by avoiding these conditions

- However, deadlock is more general than these four
  - it may depend on general synchronization conditions (cf. condition synchronization)
Model for analysis: graphs

• For consumable resources and general condition synchronization, the graph representation is the *wait-for* graph
  – nodes: tasks
    • i.e., the activities, thread/process
  – edges: a *wait-for, or blocked-on* relationship
    • an edge $t_0 \rightarrow t_1$ means that $t_0$ may unblock $t_1$
Wait-for graph

- **p1:**
  - ... $P(m); x := x+y; \text{sigall (c)}; V(m) ...$
- **p2:**
  - ... $P(m); x := a; \text{sigall (c)}; V(m) ...$
- **p3:**
  - ... $P(m); \textbf{while } x<0 \textbf{ do wait (m,c) od}; x := x-1; b := true; \text{signal (d)}; V(m) ...$
- **p4:**
  - ... $P(m); \textbf{while not } b \textbf{ do wait (m, d) od}; ...b := false; V(m) ...$

- The graph captures a possible dynamic situation, a system state
  - possibility of existence of the graph needs evidence
    - e.g., if $a$ is always negative, there is never a state with an arrow $p2 \rightarrow p3$
  - may label the arrows with corresponding conditions
    - i.e., state information about the state when this waiting occurs
    - e.g., if this graph occurs, we have: $x<0$ and $\neg b$

- **Note:**
  - we leave out the dependence on $m$ since we know mutual exclusion does not add to deadlock provided critical sections terminate.
    - Question: what are the critical sections then?
Model for analysis: graphs

• For consumable resources and general condition synchronization, the graph representation is the *wait-for* graph
  – nodes: tasks
    • i.e., the activities, thread/process
  – edges: a *wait-for*, or *blocked-on* relationship
    • an edge \( t_0 \rightarrow t_1 \) means that \( t_0 \) may unblock \( t_1 \)

• With reusable resources (action synchronization), we use the *resource dependency* graph
  – bipartite graph with two classes of nodes: tasks & resources
  – edges of three types, capturing states, and classes of states
    • task has requested and now waits for the resource
    • task has acquired (holds) the resource
    • task may request the resource
  – the graph captures *dynamic* states (a particular one, or all)
  – three events change the state
    • *request* (by a task),
    • *acquire* (response to a request by the system, according to a policy),
    • *release* (by the task)
Resource dependency graph

• Edges
  – $p \rightarrow R : p$ requests $R$
  – $R \rightarrow p : p$ holds $R$

• $p$ blocked: it has a (stable, not directly removable) outgoing arrow
  – removable: the requested resource is free

• Examples
  – $p_1$ holds one of $R_1$
  – $p_2$ holds one of $R_1$ and requests two of $R_2$
Analysis: reduction

- Assume that the graph represents a stable state

- Repeatedly remove a non-blocked task and all its incoming connections
  - this represents possible completion of that task

- Remaining set: deadlocked
  - a knot

- Sufficient condition for deadlock for the greedy allocation policy: existence of a knot in the original graph
  - greedy: direct allocation decision based on availability
  - this greedy allocation could be implemented just with a semaphore (counting the resources)

- Question: is the example reducible?
Reduction in progress...

(a) $p_1 \rightarrow p_2 \rightarrow p_3$

(b) $p_2 \rightarrow p_1 \rightarrow p_3$

(c) $p_3 \rightarrow p_4$
Dealing with deadlock

- Deadlock occurs infrequent because
  - low probability of occurrence
  - considered as a problem by programmers
    • hence, most often solved

- Approaches
  - Ignore
    • timeouts
    • external intervention
  - Prevent (avoidance from programmer side)
    • using the discussed design techniques
  - Avoid (system side)
    • dynamic checks
  - Detect and recover
Prevention

• Make the reduced dependency graphs empty by construction
• Prevent cycles in the wait-for graphs

• Use the mentioned techniques (see before on action synchronization)
  – prevent greediness, use fixed nesting (prevent circular wait), have terminating critical sections
  – not always possible

• Use preemption of the resource when needed
  – not always possible

• “All resources at once” (through condition synchronization)
  – avoid the “wait-and-hold” greediness
  – leads to possible starvation, see Philosophers
Prevention (cnt’d)

• Avoid cyclic dependencies (hence circular wait)
  – extremely boring programs

• Prove correctness, typically by contraposition
  – assume, a deadlock occurs, show a contradiction

  – in principle: examine all possible combinations of blocking states in all tasks
    • examine the corresponding dependency and wait-for graphs and show that
deadlocked ones are not possible or not reachable

• for example, for resource dependency graph: examine the reachable states of the
  Finite State Machine corresponding to the request/acquisition sequences
Example: state diagrams and reachable states

- Example of a system trace of \( p1 \) and \( p2 \)
  - 2 resources, \( R1 \) and \( R2 \)
  - \( p2 \) reserves in opposite order than \( p1 \)

- State diagram according to all possible traces
  - deadlocked state turns out to be reachable
Avoidance – prevent from system side

• Maintain as an invariant that no deadlocked sets can occur
  – upon each blocking action
    • investigate if always an open execution path remains (system remains in a “safe” state)
      – e.g. check semaphore $P$-operation order for each task
    • otherwise, deny the action or postpone completion
  – **Note**: just make sure the reduced dependency is empty

• Postponing works if the blocking actions refer to resource allocations – then we can compute the future states
  – example: bankers algorithm
    • need information about possible system behaviors in terms of resource requirements (maximum numbers)
  – requires separation between request and grant
    • hence, no greediness in the granting!
  – very difficult (impossible?) to extend to general condition synchronization
    • needs information about the evolution and modification of variables
Bankers algorithm: problem description

• Given:
  – set of $N$ tasks
  – set of $R$ resources
    • $c[j]$: number of type $j$ resources
    • $max[i,j]$: maximum number of type $j$ resources task $i$ needs

• Tasks acquire resources incrementally and release those eventually

• Requirement
  – synchronize requests such that always each task can acquire resources until its specified maximum
    • not having this requirement represents a deadlock (potential? or guaranteed?)
(Maximum) claim graph

- The specified maximum is included in the resource dependency graph, as dashed arrows
- This gives the maximum claim graph
Problem analysis

- **Problem:** while giving out resources arbitrarily, a state may be reached such that
  
  - no additional request can be served of some tasks (not enough left), ever
  
  - hence these tasks can never proceed, and will also never give back their reservations
  
  - .... actually, an instance of “wait-and-hold”
Formalization

• Given:
  – set of \( N \) tasks
  – set of \( R \) resources
    • \( c[j] \): number of type \( j \) resources
    • \( max[i,j] \): maximum number of type \( j \) resources task \( i \) needs

• State
  • \( av[j] \): available type \( j \) resources
  • \( alloc[i,j] \): type \( j \) resources in use by task \( i \)
  • \( claim[i,j] \): maximum number type \( j \) resources to be claimed by task \( i \)
    – initially, \( claim = max, av = c, alloc = 0 \)
    – Invariants:
      » \( av[j] = c[j] - (\sum i: alloc[i,j]) \)
      » \( claim[i,j] + alloc[i,j] = max[i,j] \)

• Note: we use vector and matrix addition, vector comparison and assignment with the 0 vector
  – all interpreted componentwise
Formalization (cnt’d)

- A state is called **safe** for a task if this task can be given its maximum number of resources eventually
  - possibly by giving available resources to other tasks first and then waiting until these release them again
  - if that is not needed the state is called **open** for that task
    - \( \text{claim}[i,j] \leq \text{Av}[j], \text{all } j \) (or: \( \text{claim}[i] \leq \text{Av} \))

- A state is called safe in general if it is safe for all tasks

- The initial state must be safe (what does this mean?)

- A new state resulting from granting a request is accepted only if it is safe
  - note: just requesting does not change state safety

- Now the problem reduces to verification of state safety
Example: avoiding deadlock

- Avoiding deadlock
  - (a) to (b): give resource to $p1$ leaves a reducible graph
  - (a) to (c): granting $p2$'s request leaves a non-reducible graph – hence denied or postponed
  - Note: also granting $p3$’s request is safe
State safety upon new request

• Assume the current state is safe

• Consider a new request by task $i$
  – $req[j]$ denotes the number of type $j$ resources requested by task $i$

• The new state is obtained as follows
  – $av, alloc[i], claim[i] := av - req, alloc[i] + req, claim[i] - req$

• It is enough to verify whether this new state is safe for just task $i$
  – if it is, then eventually all its resources will be returned
  – thus reaching an even safer state than the one before the adjustment

• Task $i$ is safe when its claims can be satisfied, either directly or by completing any of the other tasks (this must be verified)

• **Note:** completing the open tasks corresponds to the graph reduction mentioned before
Algorithm

func NextOpen (Av: Vector; Claim, Alloc: Matrix): int
{ returns index of first open and relevant task (one that has claimed resources) }
\[\text{for } i := 0 \text{ to } N-1 \text{ do if } \text{Alloc}[i] <> 0 \text{ and } \text{Claim}[i] <= \text{Av} \text{ then return}(i) \text{ fi od; return (N)}\];

func Safe (Av: Vector; Claim, Alloc: Matrix; req: Vector; i: int): bool
{ returns whether the parameters encode a transition to a new safe state
precondition: Av <= Req (otherwise the transition is impossible anyway) }
\[\text{var } k: \text{int};
\text{Av, Alloc}[i], \text{Claim}[i] := \text{Av}-\text{req}, \text{Alloc}[i]+\text{req}, \text{Claim}[i]-\text{req}; \{ \text{determine new state} \}
\text{k := NextOpen (Av, Claim, Alloc)};
\text{while } k <> N \text{ and not } (\text{Claim}[i] <= \text{Av}) \{ \text{there are open tasks left and } i \text{ itself is not open in current state} \}
\text{do } \text{Av, Alloc}[k], \text{Claim}[k] := \text{Av}+\text{Alloc}[k], 0, \text{Max}[k]; \{ \text{assume open task } k \text{ completes} \}
\text{k := NextOpen (Av, Claim, Alloc)};
\text{od};
\text{return (Claim}[i] <= \text{Av});
\]
Detection

- Incremental
  - upon each blocking action
    - investigate if a deadlock occurs
    - if so, deny the action
      - this may be too late though, roll-back may be needed
    - close to avoidance
      - example: checking of semaphore nesting

- Repeatedly, monitor the system
  - gives overhead
  - needs detection algorithm
Detection (cnt’d)

• Dealing with the error
  – locally, inside the task that tried blocking
  – globally, through a recovery policy

• kill
  – all
  – selectively, based on criteria, like priority, progress made etc.

• roll back to safe state
  – works only if alternatives exist
  – need recording checkpoints, i.e., enough history information to restart

• preempt resources, if possible
  – select victim based on criteria
  – however, one may argue that the deadlock does not exist in this case
Strategy depends on resource type

• Swap space in secondary memory
  – reserve all, or avoidance

• Process resources: files, printers, ....
  – require declaration ahead of time
  – use avoidance

• Main memory
  – prevention through preemption
  – see swap space if preemption is not possible

• Internal, shared resources (i/o ports, shared memory segments)
  – prevention through ordering
## Overview (Stallings)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Resource Allocation Policy</th>
<th>Different Schemes</th>
<th>Major Advantages</th>
<th>Major Disadvantages</th>
</tr>
</thead>
</table>
| Prevention    | Conservative; undercommit resources | Requesting all resources at once | • Works well for processes that perform a single burst of activity  
• No preemption necessary | • Inefficient  
• Delays process initiation  
• Future resource requirements must be known by processes |
|               |                           | Preemption         | • Convenient when applied to resources whose state can be saved and restored easily | • Preempts more often than necessary |
|               |                           | Resource ordering  | • Feasible to enforce via compile-time checks  
• Needs no run-time computation since problem is solved in system design | • Disallows incremental resource requests |
| Avoidance     | Midway between that of detection and prevention | Manipulate to find at least one safe path | • No preemption necessary | • Future resource requirements must be known by OS  
• Processes can be blocked for long periods |
| Detection     | Very liberal; requested resources are granted where possible | Invoke periodically to test for deadlock | • Never delays process initiation  
• Facilitates online handling | • Inherent preemption losses |

Additional starvation