Concurrent Systems

Nebenläufige Systeme

IX. Deadly Embrace

Wolfgang Schröder-Preikschat

December 18, 2018
Agenda

Preface

Resource Management
  Classification
  Illustrative Example I
  General

Deadlocks
  Fundamentals
  Illustrative Example II
  Counteractive Measures

Summary
discussion on abstract concepts as to the stalemate of interacting processes due to misconstructed or misguided resource allocation

- crosswise request or signalling of a reusable or consumable resource, resp.
- lost release of a produced or beforehand acquired resource

deadly embrace (Ger. tödliche Umarmung, gegenseitiges Sperren) of interacting processes by reason of programming errors

- caused by design faults and to be corrected by design changes
- focal point is to foreground constructive and eclipse analytical measures

exemplification of the classic [1] by means of sample programs

- realising that use cases of functions can uncover critical interdependencies
  - problems that are not obvious when looking at single program statements
- race conditions that are disclosed only with having the big picture in mind

not least, giving an idea of the typical counteractive measures

- prevention, avoidance, or detection and breakup of process deadlocks
- resource allocation graph and, as specialisation of it, wait-for graph
Deadlock

(Ger.) Verklemmung, Blockierung, Stillstand

source: National Geographic
Outline

Preface

Resource Management
  Classification
  Illustrative Example I
  General

Deadlocks
  Fundamentals
  Illustrative Example II
  Counteractive Measures

Summary
for whatever reason, use of operations on any type of resources can cause process or even system deadlocks

**reusable**
- **crosswise request** by different simultaneous processes
- **lost release** of a beforehand acquired resource

**consumable**
- **crosswise signalling** by interacting processes followed by `await`, whereby the signal is not buffered
- **lost release** of a produced resource

abstracting away from hardware, deadlocks are due to software faults
- only simultaneous processes may disclose these faults as error or failure
int send(pid_t pid, char *data, long size) {
    process_t *self = being(ONESELF), *peer = being(pid);

    P(&self->lock);   /* protect oneself: me */
    memcpy(self->outbox.d, data, sizeof(self->outbox.d));

    P(&peer->lock);   /* protect counterpart */
    serve(peer, self); /* message handover */
    V(&peer->lock);   /* unprotect counterpart */
    V(&self->lock);   /* unprotect oneself */

    V(&peer->inbox.gate); /* signal send done */
    P(&self->signal);   /* block on receive */

    return self->merit; /* receiver pid or error code */
}

fictive semaphore-based implementation of a message send operation

susceptible to deadlock in case of preemptive or SMP scheduling, resp.
int send(pid_t pid, char *data, long size) {
  /* ... */
  P(&self -> lock);       /* protect oneself: me */
  P(&peer -> lock);       /* protect counterpart */
  /* ... */
  V(&peer -> lock);       /* unprotect counterpart */
  V(&self -> lock);       /* unprotect oneself */
  /* ... */
}

assuming that a process $P_1$ does $send(P_2)$ and another process $P_2$
does $send(P_1)$, simultaneously

let $A$ be the process descriptor of $P_1$ and let $B$ be the one of $P_2$:
$P_1$: L3 ■ succeeded in completing $P(A)$, locked $A$ and gets preempted
$P_2$: L3 ■ succeeds in completing $P(B)$, locked $B$ and continues
$P_2$: L4 ■ gets blocked in $P(A)$, relinquishes control
$P_1$: L4 ■ resumes and gets blocked in $P(B)$, relinquishes control

$P_1$ and $P_2$ are subject to deadlock because of crosswise requests...
Progress Space

A B

P_1

P_2

deadlock has happened

deadlock is inevitable

assuming that progress of P_1 and P_2 is irreversible

© wosch, thoenig CS (WS 2018/19, LEC 9) Resource Management – Illustrative Example I
assuming that the general semaphore used to signal availability of a consumable resource is replaced by an event variable mechanism:

```c
int send(pid_t pid, char *data, long size) {
    /* ... */
    cause(&peer->event);    /* signal send done */
    await(&self->event);    /* block on receive */
    /* ... */
}
```

furthermore assuming that signalling is non-effective if no process is waiting on it, i.e., is of classical monitor semantics

again, assuming that $P_1$ called $send(P_2)$ and $P_2$ called $send(P_1)$ and that they both overlap in time and space within $send$:

L3 $P_1$ and $P_2$ simultaneously signal each other message handover

L4 both are unable waiting on it at the same moment, so the signal is lost

as a consequence, they will block on a signal that is over¹

things go right if one process waits in the wings to receive the signal

i.e., one process already did $await$ before the other one will do $cause$

¹Of course, an outsider process is able to free $P_1$ or $P_2$ by calling $send$. 
resource management generally pursues the following objectives:
- processing of orders (Ger. Auftragsabwicklung) free of conflict
- correct order management (Ger. Auftragsbearbeitung) in finite time
- balancing and maximise utilisation of resources
- high throughput, short cycle (i.e., throughput) time, high reliability...
- satisfying resource requests free from starvation or deadlock, resp.

To this end, the function is twofold in the following respect:

Accounting:
- of all resources available within the computing system
  - type, class, and number, but also
  - access rights, process allocation, and service life

Control:
- of the processing of the resource requests
  - acceptance and checking (e.g., of the access rights)
  - scheduling of the use of requested resources by processes
  - dispatching of resources to processes

Thereby, revocation and reallocation of resources is means to an end:
- to recapture resources from processes being out of hand or
- to partially or fully virtualise the hardware (e.g. processor or memory)
Methods

- **static, off-line**
  - at load time or at the outset of a particular run-time phase
  - *anticipatory* request for all resources required
  - allocation of resources takes place apart to their use
  - release of all resources occurs upon (phase) exit
  - supports *deterministic* and causes unthrifty processes
  - utilisation and behaviour of the system is *predictable*
  - elongates response time
  - risk of *under-utilisation* due to idle resources

- **dynamic, on-line**
  - at run-time, particularly at arbitrary moments
  - *on-demand* request for each required resource at a time
  - allocation of resources takes place promptly to their use
  - release of each resource occurs when need expires
  - supports *reactive* and causes non-deterministic processes
  - utilisation and behaviour of the system is *hardly predictable*
  - shortens response time
  - risk of *system deadlock* due to awkward resource demands

© wosch, thoenig  CS (WS 2018/19, LEC 9)  Resource Management – General 13
Outline

Preface

Resource Management
  Classification
  Illustrative Example I
  General

Deadlocks
  Fundamentals
  Illustrative Example II
  Counteractive Measures

Summary
Stalemate of Processes

Definition (deadly embrace)

A situation in which the **interacting processes** mutually wait on the occurrence of conditions that can be induced and established only by other processes of this very group.

- concretely, these conditions reveal the state of **resource availability**
  - independently of the kind (reusable, consumable) of resource, interacting processes expect supply by means of corresponding actions of the peers
  - as all these processes wait on each other, no resource becomes available
- according to [1], such deadly embrace of waiting processes can arise:
  1. even though no single process requires more than the total resources that are available in the system and
  2. whether the allocation of resources is the responsibility of the operating system or of the application programs themselves
- waiting can happen in **inactive** (deadlock) or **active** (livelock) mode
Waiting Mode “Inactive”

Definition (dead·lock [5])

1. a standstill resulting from the action of equal and opposed forces; 
   stalemate
2. a tie between opponents in the course of a contest
3. **DEADBOLT**—to bring or come to a deadlock

- strictly speaking, **sleep state** deadly embrace of interacting processes
  - the program counter of a deadlocked process remains constant, for the 
    most part, and waiting means to be:

  - **deep** the process state stays “blocked”, the blocked-on event is defined
    - the process releases its processor in favour of other processes
      - except for the respective—but nevertheless “blocked”—idle process
      - the processor runs in standby mode until a process becomes “ready”

- **benign**, the lesser of two evils (inactive or active stalemate, resp.)
  - in case it cannot be prevented or avoided, it can be detected
  - waiting conditions of stalemate processes can be identified externally
  - differentiation from non-stalemate processes is doubtlessly feasible
Waiting Mode “Active”

Livelock

Definition (livelock)

A deadlock-like situation in which the interacting processes indeed do not block (i.e., relinquish processor control), but they are also unable in making any actual progress in program execution.

denotes the deadly embrace of interacting processes in awake state
- the program counter of a livelocked process keeps changing and waiting means to be either of:
  - busy ■ the process state stays “running”, the process keeps its processor
  - lazy ■ the process state alternates between “running” and “ready”
    ■ the process releases its processor in favour of other processes

■ malign, the larger of two evils (inactive or active stalemate, resp.)
  ■ in case it cannot be prevented or avoided, it also cannot be detected
  ■ waiting conditions of stalemate process cannot be identified externally
  ■ differentiation from non-stalemate processes is hardly or not feasible

Checking whether or not the values of the program counters of apparently stalemate processes stay within a certain values margin—but for how long?
typedef struct account {
    semaphore_t lock;
    double balance;
} account_t;

void transfer(account_t *from, account_t *to, double amount) {
    P(&from->lock); /* acquire source account */
    P(&to->lock); /* acquire target account */
    from->balance -= amount; /* withdraw money */
    to->balance += amount; /* and deposit it */
    V(&to->lock); /* release target account */
    V(&from->lock); /* release source account */
}

assuming that two processes, $P_1$ and $P_2$, perform transfer($A, B$) and transfer($B, A$) simultaneously

- locking sequence: $P_1 : P(A) \rightsquigarrow P_2 : P(B) \rightsquigarrow P_1 : P(B) \rightsquigarrow P_2 : P(A)$
- $P_2$ waits on $A$ occupied by $P_1$ waiting on $B$ occupied by $P_2$: deadlock
- both processes hold and wait for a non-preemptable reusable resource
Approach to the Problem

Indivisible Overall Function

```c
transfer(account_t *from, account_t *to, double amount) {
  static semaphore_t bolt = {1};

  P(&bolt);  /* block transfer operation */
  from->balance -= amount;  /* withdraw money */
  to->balance += amount;   /* and deposit it */
  V(&bolt);  /* clear transfer operation */
}
```

restricts parallelism unnecessarily and is flawed by a race condition

- all transfers are locked out, particularly also those transfers that apply to unrelated pairs of accounts (i.e., reusable resources)
- assuming that, in the background, the source account (from) is subject to a simultaneous process of withdrawal
- a negative balance may be the result, with the following consequence:
  1. either the transfer operation fails due to overdraft or, as supposed here,
  2. interest paid on overdraft (Ger. Überziehungszinsen) is incurred

synchronisation must be all-embracing: per transfer and account

\(^3\)The bank feels happy about this, but not the account holder.
Solution 1

Indivisible Allocation and Overall Function

def transfer(account_t *from, account_t *to, double amount) {
    static semaphore_t bolt = {1};

    P(&bolt);           /* block transfer operation */
    P(&from->lock);     /* acquire source account */
    P(&to->lock);       /* acquire target account */
    V(&bolt);           /* clear transfer operation */

    from->balance -= amount; /* withdraw money */
    to->balance += amount;  /* and deposit it */

    V(&to->lock);         /* release target account */
    V(&from->lock);       /* release source account */
}

classic textbook solution: a measure of **deadlock prevention**
- allocation of source and target account now happens indivisibly
- transfers using the same resource pair, thus, are mutually exclusive
- but the target account **lies waste** for operations not destined for it
  - here: already blocked from line 6, although used not until line 10
transfer(account_t *from, account_t *to, double amount) {
    P(&from->lock);  /* acquire source account */
    from->balance -= amount;  /* withdraw money */
    V(&from->lock);  /* release source account */
    P(&to->lock);  /* acquire target account */
    to->balance += amount;  /* deposit money */
    V(&to->lock);  /* release target account */
}

- a doable solution, however risk is not to see the wood for the trees
- in terms of **deadlock prevention**, one is on the right track—**but**
- as to software structure, one failed to apply Occam’s razor\(^4\)
  - hypothesis that the level of abstraction of the solution is adequate
  - hypothesis that the program is readable and easily adaptable
  - hypothesis that the implementation is efficient

- particularly non-sequential programs must be of a “good” structure

\(^4\)A problem-solving principle, meaning that among competing hypotheses the one with the fewest assumptions should be selected.
Solution III

Procedural Abstraction

```c
1 transfer(account_t *from, account_t *to, double amount) {
2     change(from, -amount); /* withdraw money */
3     change(to, amount); /* and deposit it */
4 }
```

- mutual exclusion using operating system machine level functions:
  - take a sledgehammer to crack the nut...

```c
inline void change(account_t *this, double amount) { 
1     P(&this->lock); /* acquire account */
2     this->balance += amount; /* do operation */
3     V(&this->lock); /* release account */
4 }
```

- mutual exclusion using instruction set architecture level functions:

```c
inline void change(account_t *this, double amount) { 
1     FAA(&this->balance, amount); /* do operation */
2 }
```

```c
#define FAA __sync_fetch_and_add
```

5"
Solution IV

Abstract Data Type

```cpp
monitor Account {
    double balance;
public:
    inline void change(double amount) {
        balance += amount; /* do operation */
    }
};

void transfer(Account& from, account& to, double amount) {
    from.change(-amount); /* withdraw money */
    to.change(amount); /* and deposit it */
}
```

- leave it up to the compiler to do the $P/V$-pair (☉) or the FAA (☉)
- monitor procedure `change` contains neither `wait` nor `signal`, thus monitor `exit` may degenerate to $V(mutex)$ even for Hoare-style [4, p. 551, 1.]
- as `change`, by default, is defined to be indivisible, additional semantics is available to apply the FAA to the otherwise trivial computation
Once the critical section has been identified, to **factor out** is maxim. Although corresponding measures sometimes appear to be superfluous, they increase awareness for the options of improvement. This insight not only holds for the initial design or redevelopment, respectively, but also legacy software.

- in the example shown, the conflict could be eliminated by replacing a sequence of actions by an atomic instruction
  - neither blocking nor non-blocking synchronisation is used in the end
  - but reducing the problem such that downscaling the solution was feasible
- a **constructive approach** has been exercised, which finally opened a path for cross-layer optimisation
  - analytical approaches, if applicable, are without doubt important but they are nevertheless second quality in cases similar to those as were shown
  - here, the problem could be put down to a plain type of critical operation


**Hint (Livelock)**

*All of the following applies also to stillstand in active waiting mode.*

- **necessary conditions** that the interacting processes are subject to:
  1. demand control of the resources required by means of **mutual exclusion**
  2. **hold** a shared resource **and wait** for another one required to proceed
  3. resources held cannot be forcibly removed, are **ineligible for preemption**

- **necessary and sufficient condition** that eventually must be true:
  4. a situation of a **circular wait** has been occurred
     - each process holds one or more resources that are being requested by the next process in the chain

**Hint (Prevention/Avoidance)**

*All of these four conditions must be operative at the same point in time in order to deadlock. Invalidation of only one of these conditions makes the respective process system free of deadlock.*
Deadlock Prevention

**Hint (Primary Prevention)**

*Protect interacting processes from developing a standstill.*

- **indirect methods** that invalidate a necessary condition, only
  1. consider either non-blocking synchronisation or downscaling (p. 22)
  2. acquire all resources at one blow indivisibly (p. 20) or restructure (p. 21)
  3. virtualise selectively so that preemption of real resources becomes eligible, but their virtual analogues still remain ineligible for preemption

- **direct method** that invalidates the necessary and sufficient condition
  4. define a linear and total ordering of resource classes such that resource $R_i$ can be acquired previous to resource $R_j$ only if $i < j$

**Hint (Prophylaxis)**

*As a matter of principle, any rule that “prevents” the occurrence of a deadlock is a **constructive measure** that has to take effect at design and implementation time.*
Deadlock Avoidance

Hint (a priori Knowledge)

**Preliminary information** as to processes and their resource demands.

- Circumvention of deadly embrace by means of **strategical methods**
  - No attempt is made to invalidate any of the necessary conditions
  - But circular wait gets precluded by a **continuous demands analysis**
- Basic approach is to control processes and all their resource requests
  - All processes are subject to continuous checking for an **unsafe state**
  - In case of unsecured resource demand, a **denial of allocation** takes place
  - Effect is to either suspend or refuse serving of requesting processes
- Resource allocation succeeds only with certainty of a **safe state**, i.e.:
  - If there exists a process sequence that satisfies all future resource requests
  - By respecting all present allocations and pending releases (cf. p. 36)

Hint (Avoidance)

In principle, any rule that “avoids” the occurrence of a deadlock is an **analytical measure** that has to take effect at run time.

© wosch, thoenig  CS (WS 2018/19, LEC 9)  Deadlocks—Counteractive Measures 27
Determination of the Unsafe State

- one approach is using a **resource allocation graph** (RAG, cf. p.30)
  - defines a **quantity contract** for process instances regarding demand and current allocation of resources belonging to particular resource classes
  - created at process incarnation time by relying on preliminary information and updated with current data at each resource request
  - ongoing analysis as to the development of **potential cycles** in the graph

- another approach is by application of the **banker’s algorithm** [2]
  - provided that the system knows the quantity of each resource:
    - each process instance could possibly request (**maximum claim**: credit),
    - each process instance is currently holding (**allocated**: debit), and
    - is currently unallocated for all these process instances (**available**: balance)
  - then a **safe state** (cf. p. 36) is given if the request does not exceed:
    - (a) the maximum claim of the requesting process and
    - (b) the currently available stock of resources of the requested class
  - if (a) fails the request is rejected, if (b) fails the process gets suspended

- not only the need for **a priori** data is a big problem, but also scalability
deadlocks are under the **tacit** (Ger. *stillschweigend*) assumption
- no attempt is made to invalidate any of the necessary conditions
- but circular wait is detected by a **sporadic search** for blocked processes
- a **wait-for graph** (cf. p. 31) is created and searched for closed loops
- for that purpose, data derived from a RAG (cf. p. 31) is taken as a basis

detected closed loops are to be broken open in a **downstream step**
- one option is **process termination**, either by selecting a single instance
  or by taking the whole lot on the closed loop
- another option is **resource revocation**, by selecting the apparently most
  effective victim followed by a **rollback** of the owning process instance

as the case may be, the sequence repeats until closed loops vanished

---

**Tightrope Walk between Damage and Cost**

*What breaks in a moment may take years to mend.*\(^a\)

\(^a\)Swedish proverb.
Resource Allocation Graph

- A **directed graph** that interrelates process instances and resources or resource classes, resp.: serves also as basis for a wait-for graph (p. 31)
- Optional (static) preliminary information as to the *requires* relation
  - Resource classes and number of *requires* resources each

- A **dynamic data structure** to be maintained by the operating system

---

6 Mandatory ongoing information to derive a wait-for graph from a RAG.
let $A$ and $B$ be resources of the same resource class (cf. p. 8 and 18):

1. $P_1$ performs $P(A)$, $A$ is unoccupied and gets allocated to $P_1$
2. $P_2$ performs $P(B)$, $B$ is unoccupied and gets allocated to $P_2$
3. $P_1$ performs $P(B)$, $B$ is occupied $\rightarrow P_1$ has to wait for $V(B)$ by $P_2$
4. $P_2$ performs $P(A)$, $A$ is occupied $\rightarrow P_2$ has to wait for $V(A)$ by $P_1$

a closed loop from $P_1$ to $P_2$ via $A$ and $B$, back and forth: deadlock

- response time increase
- throughput decrease
- idle time overexpansion
Outline

Preface

Resource Management
  Classification
  Illustrative Example I
  General

Deadlocks
  Fundamentals
  Illustrative Example II
  Counteractive Measures

Summary
Résumé

- state of **stalemate** of interacting processes because of misguided or misconconstructed resource allocation
  - crosswise request or signalling of a reusable or consumable resource, resp.
  - lost release of a produced or beforehand acquired resource
- a deadly embrace in terms of a **deadlock** (waiting mode “inactive”) rather than livelock (waiting mode “active”)
  - in the face of all logic, the former is benign and the lesser of the two evils
  - in case it cannot be prevented or avoided, it can be detected though
  - differentiation from non-stalemate processes is doubtlessly feasible
- prevention, avoidance, or detection and breakup of process deadlocks as the classic **counteractive measures**

**Hint (Relevancy to Practice)**

*Measures for avoidance or detection of deadlocks are rather irrelevant as to practice. They are hardly realisable, too expensive, and, thus, not applicable. Besides, still dominance of sequential programming makes counteractive measures little necessary ~ ignorance.*
Reference List I

   In: *Computing Surveys* 3 (1971), Jun., Nr. 2, S. 67–78


   In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557

   Simon & Schuster, Inc., 1988
In: Lehrstuhl Informatik 4 (Hrsg.): Concurrent Systems. FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 2
let $P_k$ be a sequential process
let $S$ be an ordered set of those processes
let $b_k$ be the resource claim of a process, $P_k$
let $s(k)$ represent the ordinal number of $P_k \in S$
let $r(t)$ describe the number of resources available at time $t$
let $c_k(t)$ denote the number of resources allocated to $P_k$ at time $t$
then, a state is safe if there is a full sequence $S$ such that:

$$\forall P_k \in S b_k \leq r(t) + \sum_{s(l) \leq s(k)} c_l(t)$$  \hspace{1cm} (1)

**Condition (1)** says that the claim by process $P_k$ must not exceed the sum of the free resources and those resources which will become free “in due time,” when the processes preceding in $S$ have released theirs. [3, p. 375]