Principper for Samtidighed og Styresystemer

Concurrency Problems

René Rydhof Hansen

29 APR 2019

Learning Goals: Last time

After last lecture you

- ... can define what a race condition is
- ... can explain how mutual exclusion can be used to avoid race conditions
- ... can explain strategies for achieving and implementing mutual exclusion
- ... can define mutex and semaphore and explain how they work and where they are useful
- ... can explain how to synchronise two (or more) threads and why it may be necessary

Learning Goals

After today's lecture, you

- ... can define and explain the concept of deadlock
- ... can define and explain Coffman's conditions for deadlock
- ... can explain and use deadlock prevention strategies:
    - prevention
    - avoidance
    - detect-and-recover
- ... can define and explain the following concepts
    - livelock
    - priority inversion
- ... can use the Dining Philosophers example to explain concurrency issues

Concurrency Problems

Concurrency... what's the problem?

Different problem categories

- Race Conditions
  - Atomicity violations
  - Order violations
- Deadlock

Real code, real bugs

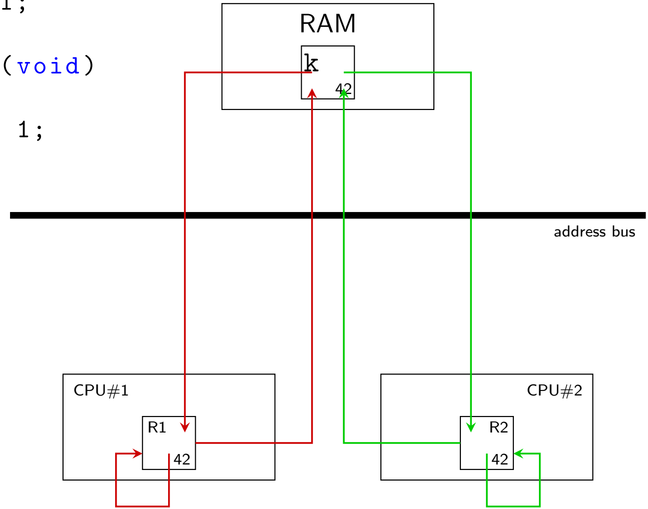
source: [OSTEP]

App	Non-deadlock	Deadlock
MySQL	14	9
Apache	13	4
Mozilla	41	16
OpenOffice	6	2
Total	74	31

Shared Memory Communication: Atomicity Violation

```c
int k = 41;

void inck(void)
{
  k = k + 1;
}
```



Shared Memory Communication: Atomicity Violation

The Fix

```
int k = 41;

void inck(void)
{
  lock_mutex();
  k = k + 1;
  unlock_mutex();
}
```

Synchronisation: Order Violation

The Problem

```
float T;
```

```
void *thread0(void *tid)
{
  T = read_sensor();

}
```

```
void *thread1(void *tid)
{

  massive_computation(T);
  output();
}
```

Synchronisation: Order Violation

The Fix

```
float T;
sem_t ready;
sem_init(&ready,0,0);

void *thread0(void *tid)
{
  T = read_sensor();
  sem_post(&ready);
}
```

```
void *thread1(void *tid)
{
  sem_wait(&ready)();
  massive_computation(T);
  output();
}
```

Deadlock: Basic Concepts

Deadlock

Definition (Deadlock)

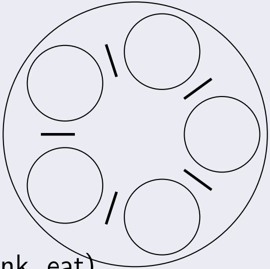
A collection of threads P are in a deadlock state if every thread in P is waiting for an event that can only be generated by another thread in P.

Deadlock

Definition (Deadlock)

A collection of threads P are in a deadlock state if every thread in P is waiting for an event that can only be generated by another thread in P.

Dining Philosophers [Dijkstra]



- Five philosophers (think, eat)
- Must have (both) chopsticks before eating
- Must take left chopstick first

Deadlock

Definition (Deadlock)

A collection of threads P are in a deadlock state if every thread in P is waiting for an event that can only be generated by another thread in P.

Example (Bounded buffer details)

```
void *PROD() {
 while(true) {
  sem_wait(&mutex);
  sem_wait(&free);
  buffer[next_free] = data;
  next_free =
    (next_free + 1) % n;
  sem_post(&used);
  sem_post(&mutex);
 }
}
```

```
void *CONSUMER() {
 while(true) {
  sem_wait(&mutex);
  sem_wait(&used);
  data = buffer[next_used];
  next_used =
    (next_used + 1) % n;
  sem_post(&free);
  sem_post(&mutex);
 }
}
```

Deadlock

Definition (Deadlock)

A collection of threads P are in a deadlock state if every thread in P is waiting for an event that can only be generated by another thread in P.

Example (Bounded buffer ... with a problem! details)

```
void *PROD() {
 while(true) {
  sem_wait(&mutex);
  sem_wait(&free);
  buffer[next_free] = data;
  next_free =
     (next_free + 1) % n;
  sem_post(&used);
  sem_post(&mutex);
 }
}
```

```
void *CONSUMER() {
 while(true) {
  sem_wait(&mutex);
  sem_wait(&used);
  data = buffer[next_used];
  next_used =
     (next_used + 1) % n;
  sem_post(&free);
  sem_post(&mutex);
 }
}
```

Resource Allocation Graph

Definition (Resource Allocation Graph)

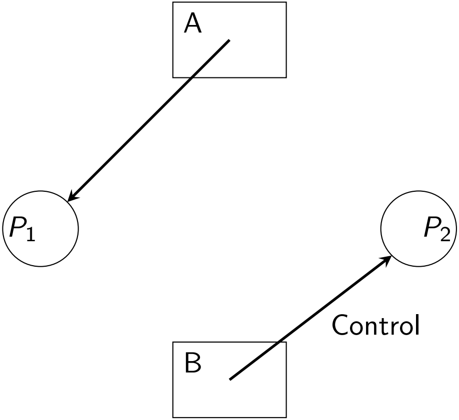
- Directed graph (V, E) with threads P and resources R as nodes: $V = P \cup R$
- An egde $(p, r) \in E$ if the thread $p \in P$ requests access to the resource $r \in R$
- An edge $(r, p) \in E$ if the thread $p \in P$ has access to resource $r \in R$

Deadlock

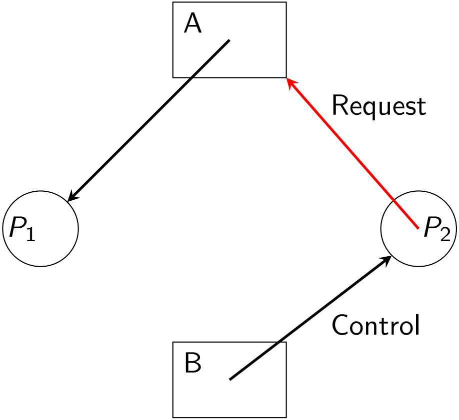
The threads P are in a deadlock state if and only if there is a cycle in the resource allocation graph[a]

[a] Provided there is only one instance of each resource

Resource Allocation Graph

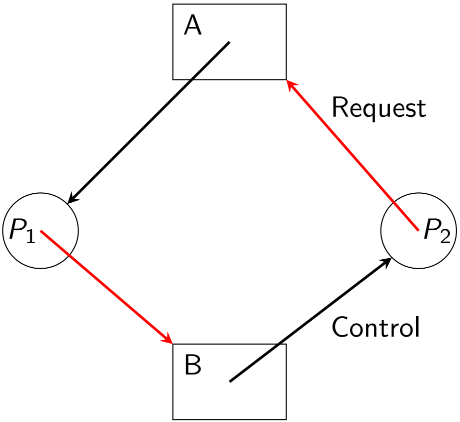


Resource Allocation Graph



Control achieved through requests

Resource Allocation Graph



Control achieved through requests

Necessary conditions for deadlock

Coffman's Conditions (1971)

- Mutual Exclusion
  - Resources cannot be shared
- No preemption
  - Resources cannot be taken away from a thread
- Hold-and-wait
  - Threads may always try to take more resources
- Circular Wait
  - There is a circular dependency of resources

Necessary conditions for deadlock

Coffman's Conditions (1971)

- Mutual Exclusion
  - Resources cannot be shared
- No preemption
  - Resources cannot be taken away from a thread
- Hold-and-wait
  - Threads may always try to take more resources
- Circular Wait
  - There is a circular dependency of resources

Consequence

Deadlock can be prevented by breaking one or more of the conditions

Solution Strategies

Solution Strategies

1. Prevention (forebyggelse)

- Design that makes deadlock impossible (requires proof, e.g., through model-chekcing)
- Invalidate one or more of the conditions necessary for deadlock

2. Avoidance (undgåelse)

- Limited and controlled “lending” of resources
- Block threads with potentially dangerous allocation requests

3. Detection and Recovery (opdag og genopret)

- “Let’s see how bad it gets”

Solution: Deadlock prevention

Break one of Coffman's conditions

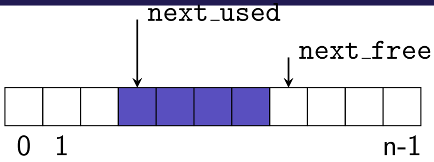
- Avoid: Mutual exclusion
  - Make resources shareable
  - Critical regions are always un-shareable resources
- Allow: Preemption
  - Allow pre-emption of resources (forcibly removing resources)
  - Only possible (safe) if state of resources can be re-created
- Avoid: Hold-and-wait
  - Require all resources to be allocated at once
  - Waste of resources
  - Potentially a long wait
- Avoid: Circular wait
  - Resources are allocated in a specific order
  - All resources must be known beforehand

Example: Bounded buffer and producer/consumer

◂ return

```
sem_t used, free;
int buffer[n],
  next_used = 0,
  next_free = 0;

int main() {
 sem_init(&used,0,0);
 sem_init(&free,0,n);
 sem_init(&mutex,0,1);
}
```

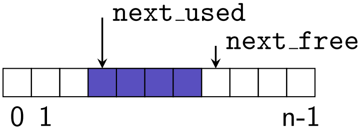



```
void *PROD() {
  while(true) {
    sem_wait(&free);
    sem_wait(&mutex);
    buffer[next_free] = data;
    next_free = (next_free + 1) % n;
    sem_post(&mutex);
    sem_post(&used);
  }
}
```

```
void *CONSUMER() {
  while(true) {
    sem_wait(&used);
    sem_wait(&mutex);
    data = buffer[next_used];
    next_used = (next_used + 1) % n;
    sem_post(&mutex);
    sem_post(&free);
  }
}
```

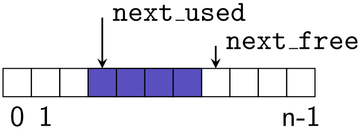
Example: Producer/Consumer

- Mutual exclusion
- Preemption
- Allocate all resources at once
- Predefined allocation order



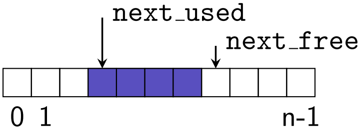
Example: Producer/Consumer

- Mutual exclusion
  - Cells can only be used by one thread at a time
  - Critical region must be protected
- Preemption
- Allocate all resources at once
- Predefined allocation order



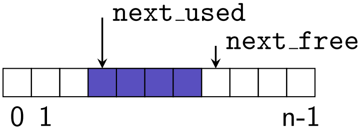
Example: Producer/Consumer

- Mutual exclusion
  - Cells can only be used by one thread at a time
  - Critical region must be protected
- Preemption
  - Variables in a critical region may end up in an inconsistent state if involuntarily “kicked out” of critical region
- Allocate all resources at once
- Predefined allocation order



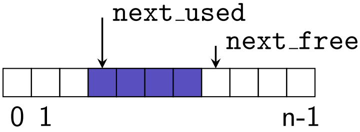
Example: Producer/Consumer

- Mutual exclusion
  - Cells can only be used by one thread at a time
  - Critical region must be protected
- Preemption
  - Variables in a critical region may end up in an inconsistent state if involuntarily "kicked out" of critical region
- Allocate all resources at once
  - Execute `sem_wait()` on both semaphores in an operation
- Predefined allocation order



Example: Producer/Consumer

- Mutual exclusion
  - Cells can only be used by one thread at a time
  - Critical region must be protected
- Preemption
  - Variables in a critical region may end up in an inconsistent state if involuntarily "kicked out" of critical region
- Allocate all resources at once
  - Execute `sem_wait()` on both semaphores in an operation
- Predefined allocation order
  - Exactly the chosen strategy for this example



Solution: Deadlock avoidance

Definition (Safe state)

A state in which there is at least one allocation sequence that allow all threads to finish without deadlock.

Definition (resource terminology)

Total amount of resources and total amount of available resources:

$$\vec{R} = (R_1, R_2, \ldots, R_m) \qquad \vec{V} = (V_1, V_2, \ldots, V_m)$$

Required resources (claims) and allocated resources:

$$\mathbf{C} = \begin{bmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{11} & C_{12} & \cdots & C_{1m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{bmatrix} \qquad \mathbf{A} = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{11} & A_{12} & \cdots & A_{1m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{bmatrix}$$

where C_{ij} (A_{ij}) requirement (current allocation) for process i resource j; with $R_j = V_j + \sum_{i=1}^{n} A_{ij}$ and $A_{ij} \leq C_{ij} \leq R_{ij}$ for all i, j

Deadlock avoidance

Process Initiation Denial

With $n-1$ processes running, only allow creation of process P_n if

$$\forall j: \quad R_j \geq C_{nj} + \sum_{i=1}^{n-1} C_{ij}$$

Pro's and Con's

- Effective (it works)
- Not efficient (too pessimistic)

Precondition

The maximum resource usage of all threads must be known in advance

Safe state

Definition

A system is in a safe state if there exists an ordering $P_1, P_2, \ldots, P_n$ of the system's threads such that

$$\forall i : \forall j : \quad C_{ij} - \sum_{i'=1}^{i} A_{i'j} \;\leq\; V_j$$

In words: a thread P_i can finish with the currently free resources and any resource(s) held by threads $P_{i'}$ for $i' \leq i$.

- In a safe state a system can avoid deadlock by letting processes run to termination starting with P_1
- Will an unsafe state always lead to deadlock?

Safe state

Definition

A system is in a safe state if there exists an ordering $P_1, P_2, \ldots, P_n$ of the system's threads such that

$$\forall i : \forall j : \quad C_{ij} - \sum_{i'=1}^{i} A_{i'j} \ \leq \ V_j$$

In words: a thread P_i can finish with the currently free resources and any resource(s) held by threads $P_{i'}$ for $i' \leq i$.

- In a safe state a system can avoid deadlock by letting processes run to termination starting with P_1
- Will an unsafe state always lead to deadlock? No... depends on scheduling etc.

Banker's Algorithm [Dijkstra 1965]

Definition

```
is_safe(C,A,V) {
  Q = current set of processes
  free = V
  while (Q != ∅)
    remove some P[i] from Q such that
      for all j: C[i,j] - A[i,j] ≤ free[j]
        if no such p exists return FALSE
        else free[j] = free[j] + A[i,j]
  return TRUE
}
```

- Simulates allocation (of a single resource type) and checks if system is still in a safe state
- What if `is_safe` returns `FALSE`? Unsafe state

Banker's Algorithm: Properties and Limitations

- The maximum resource usage of all threads must be known in advance
  - Rarely realistic
  - Partial solution: use program analysis to approximate
- Only for static number of processes: rarely realistic
- Useful (maybe?) for specialised systems (RT, embedded)
- Presumes that processes with all necessary resources allocated will return resources (in finite time)

Example

	A	**B**	**C**
A_{0j} (C_{0j})	2 (3)	1 (2)	0 (1)
A_{1j} (C_{1j})	1 (2)	1 (1)	1 (2)
V_j (R_j)	1 (4)	0 (2)	2 (3)

P0: 1A + 1C? Not ok

P1: 1A + 1C? Ok

Solution: Detection and recovery

Detection and Recovery

- Assume deadlocks are rare
- Wait for deadlock to happen
- Re-establish normal (non-deadlocked) state

Preconditions for Detection and Recovery

- Deadlock state must be detectable (e.g., resource allocation graph)
- It must be possible to define and re-establish normal state

Definition (Ostrich algorithm (Strudsealgoritmen))

Bury head in sand and hope the problem goes away by itself, i.e., leave it to the user

Re-establishing normal state

Abort one of the deadlocked processes

- Acceptable if the process(es) can be restarted
- LaTeX can be restarted, a network connection cannot always be "re-started" or re-established
- Which process should be terminated?

Check points

- Enables system to repeat aborted operations
- ... or to return to consistent state from before operation was aborted
- Example: transactional model for DB's
- Recovery points
- Problematic for processes with side-effects/external communication

Livelock, Starvation, and Priorities

Other "interesting" Concurrency Phenomena

- Livelock
  - Deadlock-like state with no progress although with no cyclic wait
  - Example: ethernet (exponential backoff protocol)
- Starvation
  - Situation where a thread never acquires a resource because other processes always get it first
  - Typical problem for systems with priorities
- Priorities
  - What can have priorities: processes, threads, resources, ...
  - Often useful/necessary with priorities
  - Example: high priority for threads interacting with users (better response time)
  - Problem: starvation, priority inversion

Priority Inversion

Priority Inversion

- When a low-priority thread blocks a high(er)-priority thread
- Happens when a higher-priority threads needs a resource held by a lower-priority thread

Example

Process L with low priority uses the printer; process H with high priority needs the printer.

- Result: H waits for L to finish. Problem?
- Alternative: Other processes with medium priority are scheduled instead of L which thus never finishes; de facto starvation of both H and L.

Example (Mars Pathfinder)

Low priority meteorological thread could not unlock mutex protecting communication bus needed for high priority thread.

Priority Inversion: Solutions

- Interrupt disabling
  - Let interrupt masking be the only way to achive mutex
  - In reality only two priorities: preemptible and non-preemptible
- Priority inheritance
  - A thread T inherits the priority of any higher priority threads blocked on resources held by T
  - Dynamic calculation of ceiling and priority
- Priority ceiling
  - Resources are assigned a priority ceiling corresponding to the highest priority thread that accesses the resource
  - Threads using the resource are temporarily assigned the ceiling priority
  - Requires code-/resource-analysis to determine ceiling
- Immediate Ceiling Priority Protocol (ICPP)
  - Assign ceiling priorities to resources
  - Temporarily raise the priority of a thread to the priority of accessed resources
  - Threads can only request threads with higher priority

Properties of priority ceiling protocols

On a single processor

- A high-priority process is blocked at most once duing its execution by lower-priority processes
- Deadlocks are prevented
- Transitive blocking is prevented
- Mutual exclusive access to resources is ensured by the protocol itself

Comparing OCPP versus ICPP

- Worst-case behaviour is identical (from a scheduling viewpoint)
- ICPP is easier to implement than the original (OCPP) as blocking relationships need not be monitored
- ICPP blocks prior to first execution: fewer context switches
- ICPP requires more priority movements as this happens with all resource usage
- OCPP changes priority only if an actual block has occurred

Alternative: Non-Blocking Data Structures

List insert (blocking)

```
void insert(int value) {
  node_t *n = ...
  n -> value = value;
  mutex_lock();
  n -> next = head;
  head = n;
  mutex_unlock();
}
```

List insert (non-blocking)

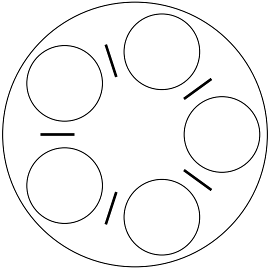
```
void insert(int value) {
  node_t *n = ...
  n -> value = value;
  do {
    n -> next = head;
  } while(CompareAndSwap(&head,n->next,n) == 0);
}
```

Alternative: non-blocking data structures

- Solution at language-level
- Build non-blocking into data structure
- Example: unbounded non-blocking queue; uses `compare-and-swap` instruction directly

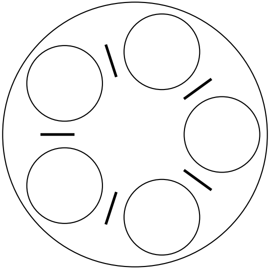
Dining Philosophers

Dining Philosophers [Dijkstra]



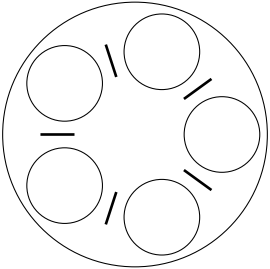
- If each philosopher picks up left chostick, the result is deadlock
  - deny hold-and-wait: put chopstick down if the other is unavailable
  - this results in livelock: pick up, put down, pick up, put down
  - fastest philosopher will eventually break the livelock

Dining Philosophers [Dijkstra]



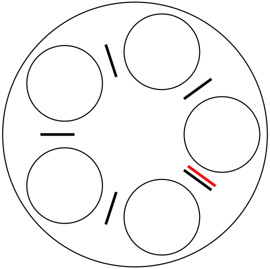
- Allow pre-emption: each philosopher grabs right-hand neighbour's chopstick
  - livelock again: get right chopstick, lose left chopstick, ...
  - strongest philosopher will eventually break the livelock

Dining Philosophers [Dijkstra]



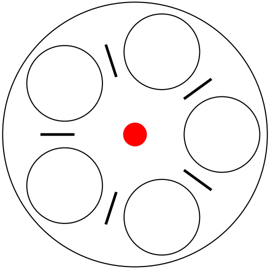
- Allow sharing
  - how to share chopsticks?

Dining Philosophers [Dijkstra]



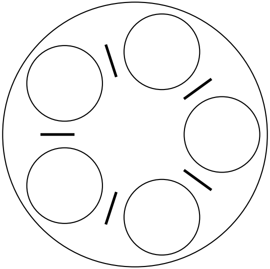
- Avoid circularities
  - add one extra chopstick so both neighbouring philosophers have one chopstick that isn't shared

Dining Philosophers [Dijkstra]



- Add a mutex to arbitrate resource acquisition
  - only the philosopher with the (one and only) soy sauce bottle can pick up chopsticks

Dining Philosophers [Dijkstra]



- Require both chopsticks are available before picking up either: Two philosophers can starve one seated between them

Summary

Concurrency — Important Concepts

- Race condition
  - When the result of a computation depends on the relative speed of the individual threads
  - In other words: the result depends on the actual interleaving of the threads
  - Hard to debug
- Critical region (critical section)
  - Program fragment vulnerable to race conditions
  - Danish: "kritisk region"

> Critical regions must be executed under mutual exclusion

- Mutual exclusion (mutex)
  - When only one thread (among many) can access a given resource or execute specific part of the program-text
  - Danish: "gensidig udelukkelse"
- Atomic
  - Event, or sequence of events, that happen(s) uninterruptedly

Concurrency Summary

- The relative speed of threads is unpredictable
- Identify shared resources
- Identify critical regions
- Ensure mutual exclusion in critical regions
- Do not use scheduling for mutual exclusion
- Do not assume specific ordering on thread wake up
- Avoid busy wait
- Check if libraries are thread-safe before using them
- Check for: race conditions, deadlocks, livelocks, starvation, priority inversion, ...