Deadlocks

INF-2201 Operating Systems Fundamentals – Spring 2015

Bård Fjukstad bard.fjukstad@uit.no b.fjukstad@met.no

Based on a presentations created by
Daniel Stødle, NORUT
And Kai Li and Andy Bavier, Princeton

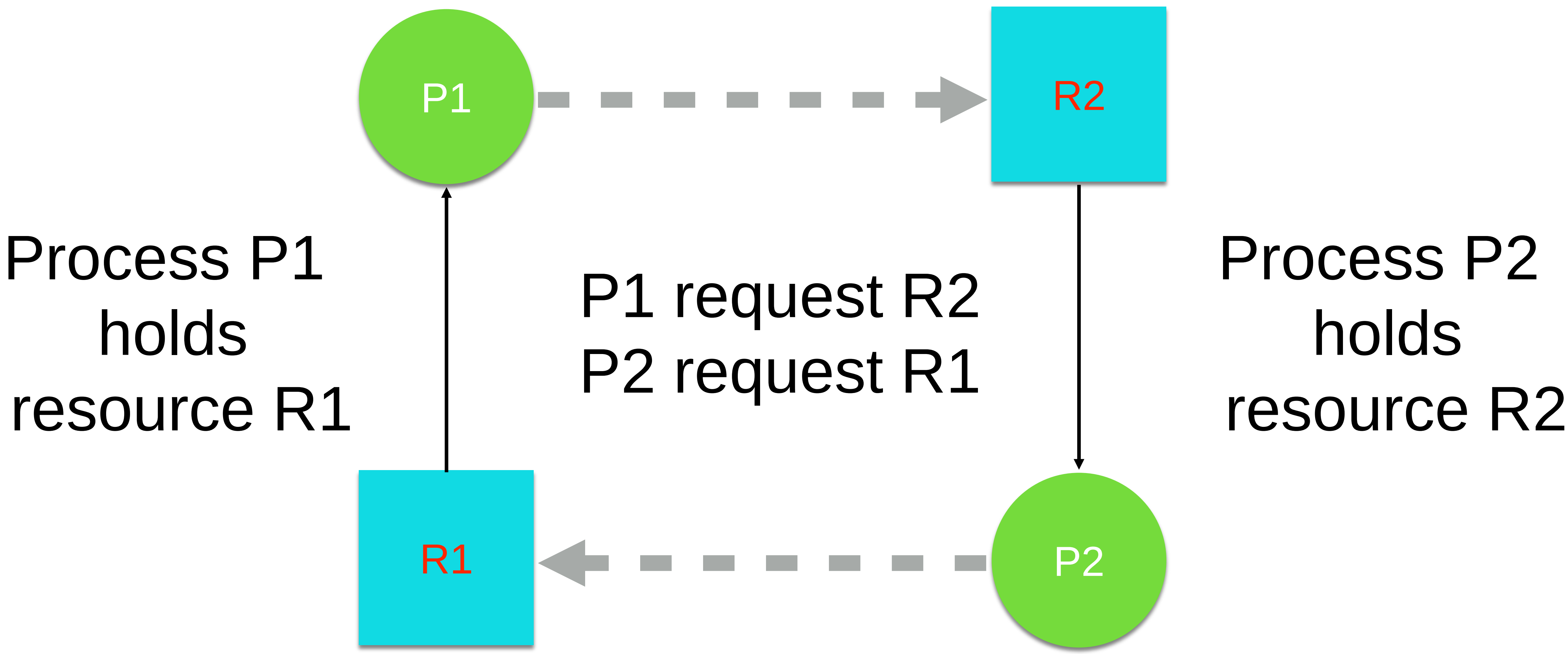
Definition

- “

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

“Modern Operating Systems”, 4rd ed p.439

Example



Resource allocation graph
if cycle ==> Deadlock

Resources

- All operating systems manage resources
  - CPU, memory, disk, external hardware, …
- Processes (or threads) sometimes need exclusive access to resources
  - Devices, files, database tables
  - Shared state (counters, queues, lists, etc)
- Preemptable vs nonpreemptable resources
  - Can the resource safely be taken away from a process?
  - Deadlocks typically involve nonpreemptable resources
  - Request ==> Use ==> Release

Types of deadlock

- Resource deadlocks
  - Two processes block waiting for a resource held by the other
- Communication deadlocks
  - Processes block waiting for replies that might never be forthcoming due to dropped packets
- Livelock
  - Processes are running, but not making progress

Deadlock conditions

- Mutual exclusion
  - Only one process may use a resource at any given time
- Hold and wait
  - Process holds at least one resource and waits for a different resource to become available
- No preemption
  - A resource must be released by the process that acquired it
- Circular wait
  - Every process waits for a resource held by the next process in chain

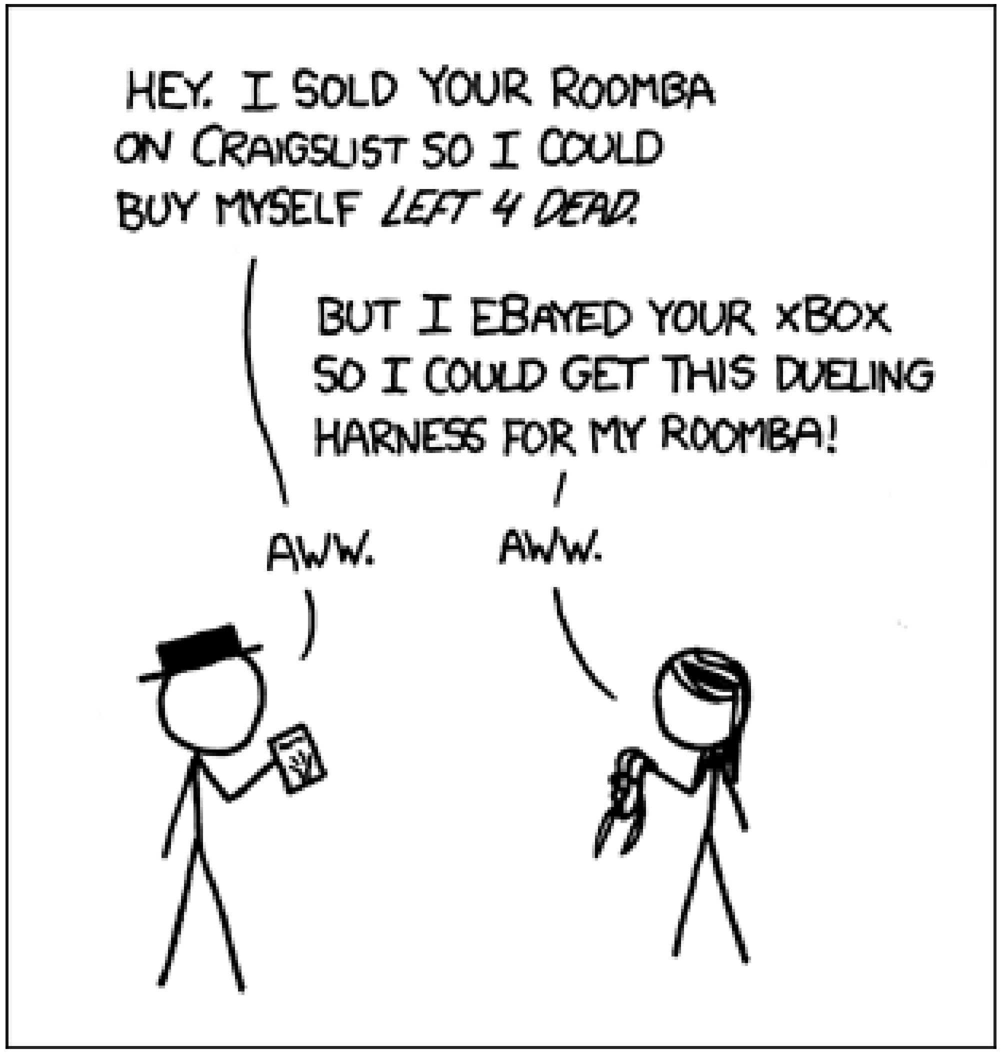


Image credit: Randall Munroe, http://xkcd.com/506/

Deadlocks in practice

```
// gcc -o deadlock-threads deadlock-threads.c -lpthread
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

pthread_mutex_t     mutex1,  mutex2;
int                 thread1_counter     = 0,
                    thread2_counter     = 0,
                    sleep_max_usec      = 1;

void*thread1(void *arg) {
        while (1) {
                pthread_mutex_lock(&mutex1);
                pthread_mutex_lock(&mutex2);
                thread1_counter++;
                printf("Thread 1: %d\n", thread1_counter);
                pthread_mutex_unlock(&mutex2);
                pthread_mutex_unlock(&mutex1);
                usleep((rand()&0x7ffffff) % sleep_max_usec);
        }
        return 0;
}
```

Could OS detect this deadlock?
Deadlock to livelock with spinlocks

```
void*thread2(void *arg) {
        while (1) {
                pthread_mutex_lock(&mutex2);
                pthread_mutex_lock(&mutex1);
                thread2_counter++;
                printf("Thread 2: %d\n", thread2_counter);
                pthread_mutex_unlock(&mutex1);
                pthread_mutex_unlock(&mutex2);
                usleep((rand()&0x7ffffff) % sleep_max_usec);
        }
        return 0;
}


int main(int argc, char *argv[]) {
        pthread_t tid;
        srand(time(0));
        sleep_max_usec = (argc > 1 ? atoi(argv[1]) : 1000);
        pthread_mutex_init(&mutex1, 0);
        pthread_mutex_init(&mutex2, 0);
        pthread_create(&tid, 0, thread1, 0);
        pthread_detach(tid);
        thread2(0);
        return 0;
}
```

Handling deadlock

- Detect and recover
  - More common in database systems, not so much in operating systems
- Avoid deadlocks using “smart” resource allocation
- Prevent by negating one of the deadlock conditions
- Ignore the problem (“Ostrich algorithm”)
  - In practice, this is what most operating systems do
  - Why pay a (potentially severe) performance penalty for an issue that occurs infrequently?

Detection

- Inspect resource allocation graph
  - Look for cycles
  - Depth-first traversal

```
#!/usr/bin/env python
graph = ["R", "A", "B", "C", "S", "D",
                    "T", "E", "F", "U", "V", "W", "G"]
edges = { "R" : [["A", False]], "A" : [["S", False]],
                     "B" : [["T", False]], "C" : [["S", False]],
                     "D" : [["S", False], ["T", False]],
                     "E" : [["V", False]], "F" : [["S", False]],
                     "G" : [["U", False]], "S" : [],
                     "T" : [["E", False]], "U" : [["D", False]],
                     "V" : [["G", False]], "W" : [["F", False]] }

def unmarkEdges():
        for node in edges.keys():
                for i in range(0,len(edges[node])):
                        edges[node][i][1]       = False

def detectDeadlock():
        for node in graph:
                L = [node]
                unmarkEdges()
                while len(L) > 0:
                        currentNode             = L[-1]
                        foundUnmarked           = False
                        print "Current: ", currentNode, "List", L
                        for edge in edges[currentNode]:
                                if not edge[1]:
                                        if edge[0] in L:
                                                L.append(edge[0])
                                                print "Deadlock detected:", L
                                                return True
                                        L.append(edge[0])
                                        edge[1]                 = True
                                        foundUnmarked           = True
                                        break
                        if not foundUnmarked:
                                print "No cycle detected starting at ", L[0], ":", L
                                L = L[:-1]
        return False

if __name__=="__main__":
        detectDeadlock()
```

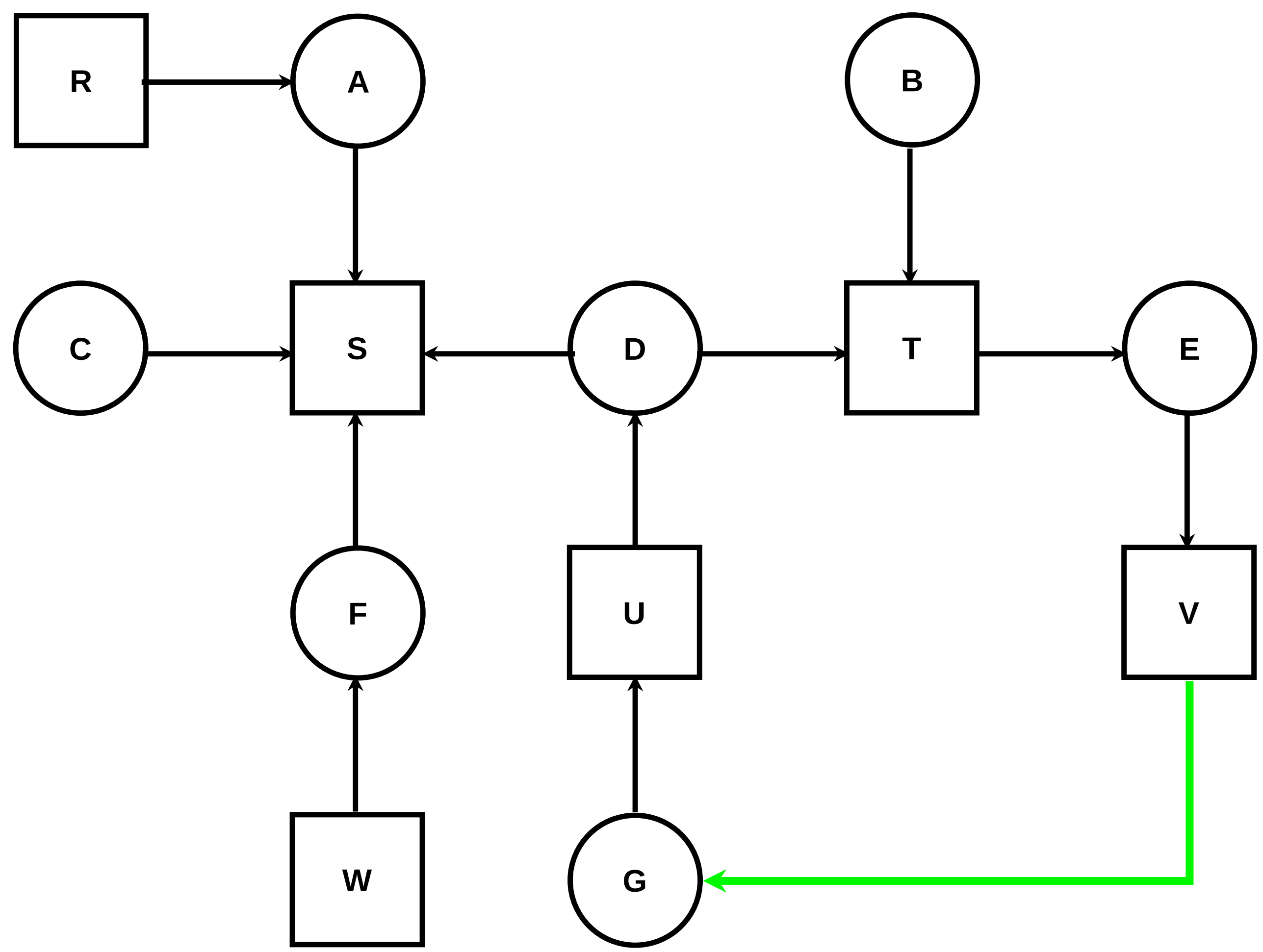


Figure adapted from “Modern Operating Systems”, 4rd ed

Recovery

(hard)

- Preemption
  - Temporarily take a resource away from the process holding it
  - Not usually feasible, depends strongly on characteristics of resource being preempted
- Rollback
  - Processes are checkpointed periodically
  - Deadlock detected? Roll process state back to state before acquisition of a needed resource
- Kill processes
  - Crude example: Linux Out-of-Memory (OOM) killer

Avoidance

- Resources typically not acquired all at once
- Can resource requests be scheduled so as to always avoid deadlock?
  - Yes, if max resource need is known in advance for all processes

Safe and unsafe states

- Assume max number of resources a process will use is known
- Track state of each process' resource usage
- Safe state: A sequence of resource allocations exists that allow all processes to complete
- Unsafe state: No guarantee that all processes will complete without deadlocking

Total: 8

Safe state:

	Has	Max
P_1	2	6
P_2	2	3
P_3	3	5

Free: 1

	Has	Max
P_1	2	6
P_2	3	3
P_3	3	5

Free: 0

	Has	Max
P_1	2	6
P_2	0	0
P_3	3	5

Free: 3

	Has	Max
P_1	2	6
P_2	0	0
P_3	5	5

Free: 1

	Has	Max
P_1	2	6
P_2	0	0
P_3	0	0

Free: 6

Unsafe state:

	Has	Max
P_1	4	6
P_2	1	3
P_3	2	5

Free: 1

?

Either process may require 5 resources, and end up deadlocked

Banker’s algorithm

- Only grant resource requests that lead to a new safe state
- Otherwise, delay requests from processes
- Practical utility of algorithm is questionable
  - Max resource count rarely known in advance
  - Resources may disappear
  - New processes are added to the mix

Prevention

- Four conditions required for deadlock
  - (1) Mutual exclusion, (2) hold and wait, (3) no preemption and (4) circular wait
- Mutual exclusion
  - Deadlocks on a printer can be avoided by spooling output; only the printer daemon talks to the printer
  - Deadlocks still possible if printer daemon waits for complete file and all disk space is consumed
    - This time, we deadlock on the disk

Prevention: Hold and wait

- Request all resources needed upfront
  - What if we don’t know which resources are needed?
  - Inefficient, as resources are tied up until the process exits
- Release held resources, then attempt to acquire all resources at once

Prevention: No preemption

- Virtualized resources (printer daemon)
- Not really possible for database tables
    - Data corruption/loss

Prevention: Circular wait

- Only one resource at a time per process
  - Need a different resource? Release the other resource first
- Global resource numbering
  - Always acquire resources in ascending order
  - Cycles can never occur
  - Is it practical to assign a strict order of all resources governed by an operating system?

Starvation

- Similar to deadlock/livelock
- Scheduling policy affects possibility of starvation
- For instance, shortest job first may result in longer-running jobs never getting a chance to execute/acquire resources
  - FIFO allocation of resources avoids starvation

Halo

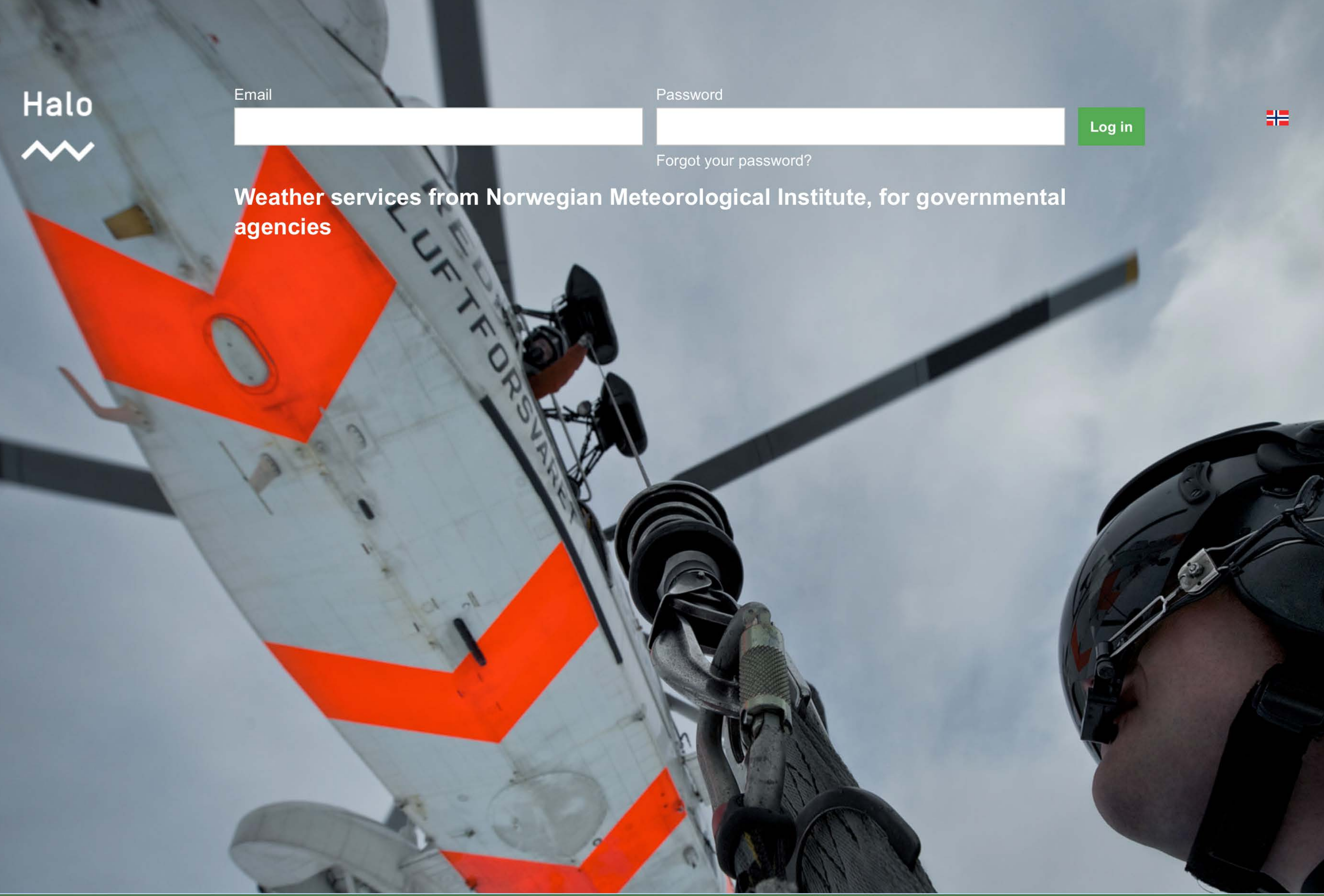
Email

Password

Log in

Forgot your password?

Weather services from Norwegian Meteorological Institute, for governmental agencies



Practical Starvation

- Extreme Weather event «NINA» January 2015
- yr.no hits 6 mill distinctive users and handles this
- halo.met.no is resource for governmental use
  - Server crashes
  - «ulimit» sets maximum number of open files for a process. Set to 10 000. Previously observed around 1 000.
  - Demand requests around 12 000 files.
  - Smart programmer had good error reporting

Debugging deadlocks

- pthread_setname_np(…), pthread_getname_np(…)
  - “np” = not portable
  - Debuggers typically display thread name
  - getname() can be used for printf’s
- dtrace and similar tools

Summary

- Deadlock: Set of processes where each is waiting for resources that can only be released by another in same set.
- Four conditions required for deadlock
  - (1) Mutual exclusion, (2) hold and wait, (3) no preemption and (4) circular wait
- Handling:
  - Reboot, remove device, kill application and restart. Checkpoint the application and kill.

Other presentations

- Princeton – Singh
  - http://www.cs.princeton.edu/courses/archive/fall18/cos318/lectures/9.Deadlock.pdf

Questions?