

Deadlocks

Deadlock examples

Real-life deadlock



Deadlock examples

Mutually recursive locking

```
void thread1(void)
{
    lock(lock1);
    lock(lock2);

    ... /* computation */ ...

    unlock(lock2);
    unlock(lock1);
}
```

```
void thread2(void)
{
    lock(lock2);
    lock(lock1);

    ... /* computation */ ...

    unlock(lock2);
    unlock(lock1);
}
```

- Will this code always reach completion?

Analysis

- Somewhat similar to race conditions
 - Will probably reach completion most of the time
 - But in rare cases, a certain order of execution will make the code deadlock
- *Could be solved simply by respecting consistent structure*
 - Core design pattern in concurrent programming

Deadlock examples

Dining Philosophers



```
sem_t fork[N];

void philosopher(int i)
{
    sem_t right = fork[i];
    sem_t left  = fork[(i + 1) % N];

    while (1) {
        think();
        sem_down(right);
        sem_down(left);
        eat();
        sem_up(left);
        sem_up(right);
    }
}
```

Analysis

- Deadlock if they're each able take their right fork
 - All waiting for their left fork to proceed

Deadlock

Definition(s)

Starvation

- One or more tasks of a group are indefinitely blocked
 - Not able to make progress
 - These tasks are *starving*
- But, the group as a whole is still making progress

Deadlock

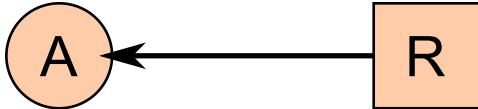
- Group of tasks is deadlocked if each task in the group is waiting for an event that only another task in the group can trigger
 - Usually, the event is the release of a currently held resource
 - None of the tasks can
 - run
 - releases resources
 - or be awakened
- A **deadlock** is the ultimate stage of **starvation**

Deadlock

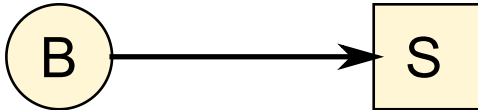
Resource allocation graph

Formalism

- Task A is *holding* resource R

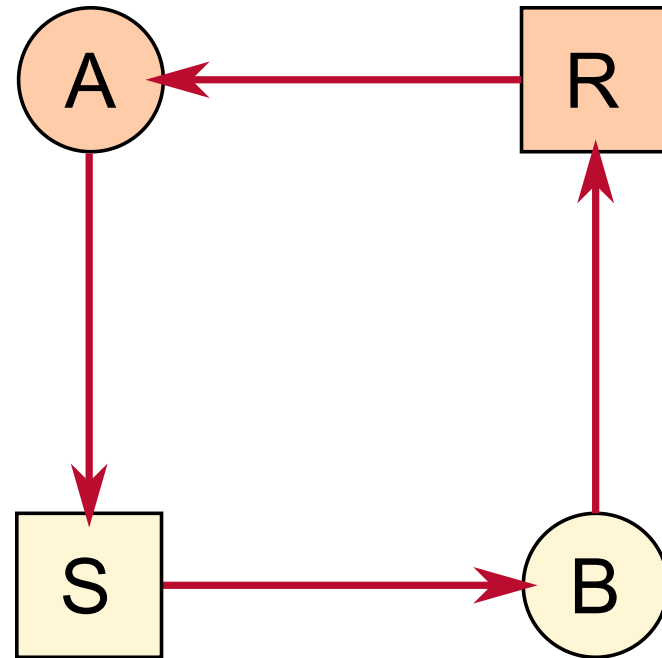


- Task B is *requesting* resource S



Example

- A is requesting S while holding R
- B is requesting R while holding S



- Cycle in the resource allocation graph expresses a **deadlock**!

Deadlock

Four conditions describing a deadlock

1. Mutual exclusion/bounded resources

- A resource is assigned to exactly one task at a time

2. Hold and wait

- Each task is holding a resource while waiting for another resource

3. No preemption

- Resources cannot be preempted

4. Circular wait

- One task is waiting for another in a circular fashion

When a deadlock occurs, it means that **all four** conditions are met



Deadlock strategies

Dealing with deadlocks

- **Ostrich algorithm**

- Just ignore the problem altogether
- Reset when it happens



- **Detection and recovery**

- Let the problem occurs
- Fix it afterwards



- **Dynamic avoidance**

- Careful resource allocation during execution
- Avoid having the problem



- **Prevention**

- Negating one of the four necessary conditions so that, by design, the problem cannot happen



Deadlock strategies

Ostrich algorithm

A.k.a burying one's head in the sand and pretending there is no problem

Idea

- If the OS kernel locks up
 - Reboot!
- If an application hangs
 - Kill the application and restart!
- Etc.



Mitigation

- Make it less painful for users
 - "Autosave" principle
- If an application runs for a while and then hangs
 - Checkpoint the application
 - Change the environment (reboot?)
 - Restart the application from the checkpoint

Deadlock strategies

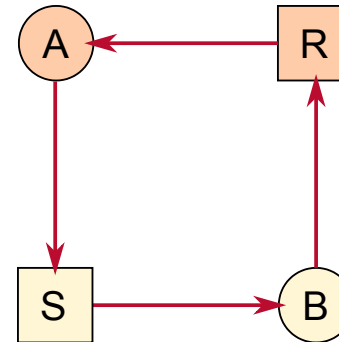
Detection and recovery

Idea

- Let deadlocks happen and then deal with the situation
 - Need to detect deadlock first
 - Once deadlock is detected, recover from it

1. Detection

- Track resource ownerships and requests
- If a cycle can be found within the graph, then there is a deadlock



2. Recovery

- Kill task
 - Choose task that can be rerun from the beginning
- Rollback task
 - Regular checkpoint and restart from it
- Resource preemption
 - Take missing resource from another task
 - Very difficult in practice

Deadlock strategies

Cycle detection algorithm

For each node N in the graph

1. $L := \text{empty list}$, all edges are unmarked
2. Add current node to L
 - If node already in L: **cycle!**
3. Pick an unmark edge, mark it, follow to new current node and go to 2
 - If no outgoing unmarked edges, go to 4
4. Remove current node from list, go back to previous node and go to 3
 - If back to root node, **no cycle!**

ECS 150 - Deadlocks

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021



Recap

Deadlock

- Starvation is when one or more tasks are indefinitely blocked
- Deadlock is the ultimate stage of starvation

```
void thread1(void)
{
    lock(lock1);
    lock(lock2);
    ...
}
void thread2(void)
{
    lock(lock2);
    lock(lock1);
    ...
}
```

- Code subject to deadlock doesn't mean that it ever will

Conditions

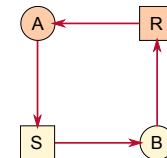
1. Mutual exclusion/bounded resources
2. Hold and wait
3. No preemption
4. Circular wait

Deadlock strategies

- Ostrich algorithm



- Detection and recovery



- Dynamic avoidance
- Prevention

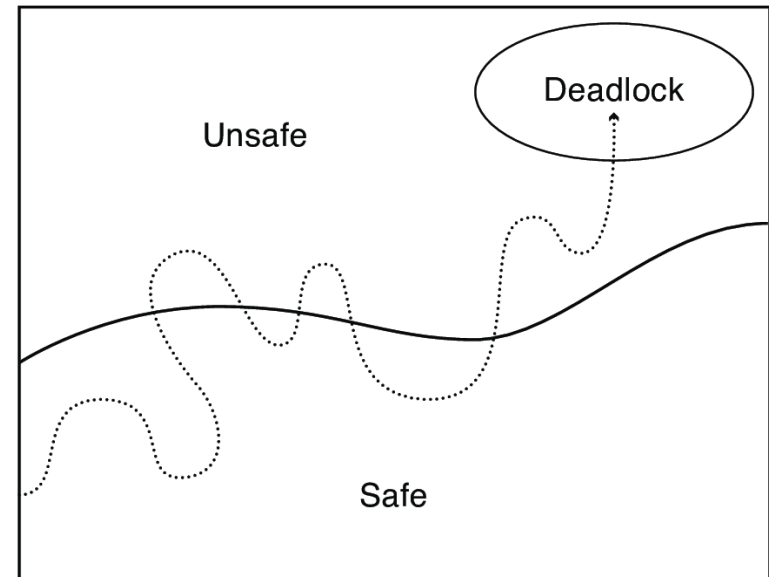
Deadlock strategies

Dynamic avoidance

- When a resource is requested, resource is granted only if it maintains the system in a *safe state*

States

- **Safe state**
 - For any possible sequence of resource requests, there exists at least one safe sequence of processing the requests that eventually succeeds in granting all pending and future requests.
- **Unsafe state**
 - In an unsafe state, there exists at least one sequence of pending and future resource requests that leads to deadlock no matter what processing order is tried.
- **Deadlock**



Deadlock strategies

Banker's algorithm

1. State maximum resource needs in advance
2. Allocate resources dynamically, when resource is requested
 - Wait if request is impossible
 - Wait if request would lead to unsafe state
3. Request can be granted only if there exists a sequential ordering of threads that is deadlock free

Deadlock strategies

Banker's algorithm -- Example

	Maximum Need	Currently allocated
Task A	9	3
Task B	4	2
Task C	7	2
Free resources	10	3

- B requests 2 resources: should grant request or not?
- A requests 1 resources: should grant request or not?

Scalability

For tracking multiple resources:

- Two 2-D matrices
 - Allocated
 - Max needed

Deadlock strategies

Prevention

By design, **invalidate** one of the four conditions:

1. Mutual exclusion/bounded resources
2. Hold and wait
3. No preemption
4. Circular wait



"I'll have an ounce of prevention."

Deadlock strategies

Invalidate "Mutual exclusion/bounded resources"

More resources

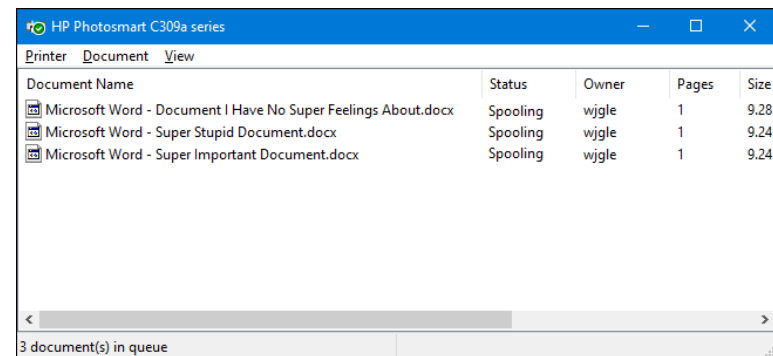
- Increase number of available resources

```
sem_t fork[N + 1];
```

Fix to dining philosophers problem

Virtualization

- Virtualize resources
 - E.g., printer queue

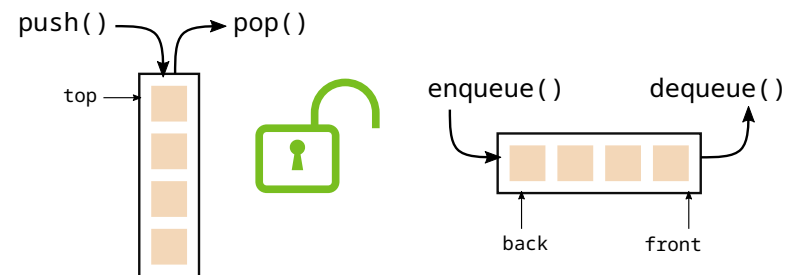


Document Name	Status	Owner	Pages	Size
Microsoft Word - Document I Have No Super Feelings About.docx	Spooling	wjgle	1	9.28
Microsoft Word - Super Stupid Document.docx	Spooling	wjgle	1	9.24
Microsoft Word - Super Important Document.docx	Spooling	wjgle	1	9.24

3 document(s) in queue

Lock-free resources

- Make resources sharable without mutual exclusion requirement
 - Lock-free data structures



Deadlock strategies

Invalidate "Hold and wait"

- Don't hold resources when waiting for another

Two-phase locking

- Phase 1:
 - Try to lock all required resources before proceeding
- Phase 2:
 - If successful, use needed resources and release
 - Otherwise, release all resources and try acquiring them again

Example

```
while (1) {  
    think();  
    sem_down(right);  
    if (!sem_try_down(left)) {  
        sem_up(right);  
        continue;  
    }  
    eat();  
    sem_up(left);  
    sem_up(right);  
}
```

Fix to dining philosophers problem

Issues

- Starvation if task is never able to grab all the resources it needs
- Complicated to manage when the amount of resources grows

Deadlock strategies

Invalidate "No preemption"

- Make *resources* preemptable by runtime system
 - Preempt requesting tasks' resources if all not available
 - Preempt resources of waiting tasks to satisfy request
- Only works when it's easy to save and restore state of resource
 - Memory page via swapping
 - CPU via context switching
- In most cases, *impossible...*

Deadlock strategies

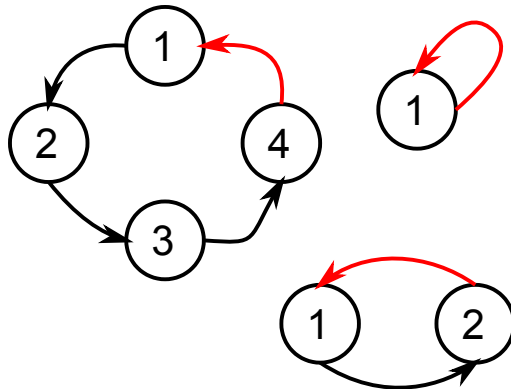
Invalidate "Circular wait"

- Impose an order of requests for all resources
 - Assign a unique ID to each resource
 - All requests must be in ascending order of the IDs

Rationale

Intuition is that a cycle requires

- Edge from high to low node
- Edge to same node



Example

```
sem_t right = fork[i];
sem_t left  = fork[(i + 1) % N];

while (1) {
    think();
    if ((i + 1) % N < i) {
        // Only for last philosopher
        sem_down(left);
        sem_down(right);
    } else {
        // Regular philosophers
        sem_down(right);
        sem_down(left);
    }
    eat();
    sem_up(left);
    sem_up(right);
}
```

Fix to dining philosophers problem

Livelock

Principle

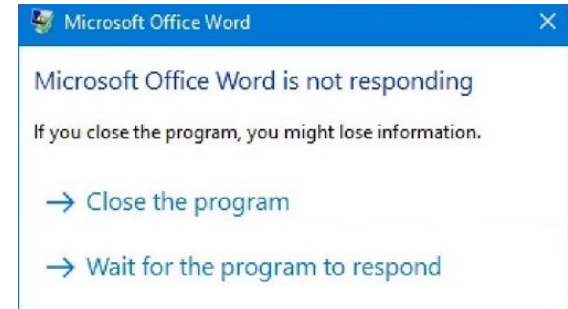
Variant of deadlock where two or more tasks are continuously changing their state in response to the other task(s) without doing any useful work.

- Similar to deadlock: no progress
- But no task is blocked waiting for anything
- Can be a risk with detection and recovery algorithms...

Real-life strategies

Ostrich algorithm

- Most systems implement the Ostrich approach



Dynamic avoidance

- A couple of POSIX functions can fail with `EDEADLK` errno
 - File-locking (POSIX `lockf()`)
 - Relocking a mutex (`pthread_mutex_lock()`)

Detection and recovery/dynamic avoidance

- Some specialized systems have elaborate mechanisms
 - Database, banking

Prevention

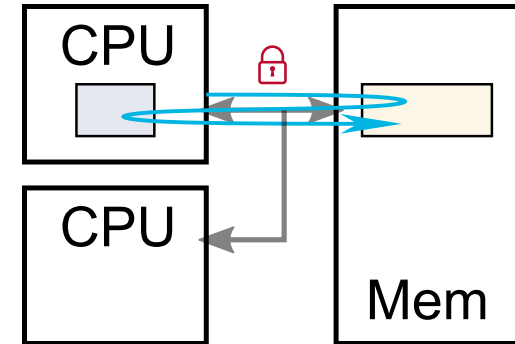
- Avoid deadlocks by not having locks in the first place!
 - Lock-free data structures
 - Transactional memory

Real-life strategies

Lock-free data structures

- Often based on *compare-and-swap*-like atomic instructions

```
// Equivalent of a CAS hardware instruction
// in software
ATOMIC int cas(int *mem, int old, int new)
{
    if (*mem != old)
        return 0;
    *mem = new;
    return 1;
}
```



Common data structures

- Stacks
 - Last In, First Out
- Queues
 - First In, First Out
- Sets
 - Collection of ordered items
- Hash tables
 - Collection of unordered items

Real-life strategies

Lock-free data structures

Stack ("naive" version)

```
void stack_push(stack_t stack, void *data)
{
    struct node *n = malloc(sizeof(struct node));
    n->data = data;

    do {
        n->next = stack->top;
    } while (!cas(&stack->top, n->next, n));
}
```

What about `stack_pop()`?

- See [link](#) for more details

Real-life strategies

Transactional memory

- No locks, just shared data
- Optimistic concurrency
 - Execute critical section speculatively, abort on conflict
- Software or hardware implementations

API

`begin_transaction()`:

- Create checkpoint
- Start tracking read set (remember memory addresses that are read)
- See if anyone else is trying to write to these addresses
- Locally buffer all the writes (invisible to other processors)

`end_transaction()`:

- Check read set: is data still valid? (i.e. no writes to any of them)
 - Yes: commit transaction => perform writes atomically
 - No: abort transaction => restore checkpoint

Real-life strategies

Example

```
void perform_transfer(int from, int to, int amount)
{
    begin_transaction();
    if (accounts[from].balance >= amount) {
        accounts[from].balance -= amount;
        accounts[to].balance += amount;
    }
    end_transaction();
}
```

```
void thread_a(void)
{
    /* Transfer 1a */
    perform_transfer(1, 4, 100);
    ...
    /* Transfer 2a */
    perform_transfer(2, 5, 74);
}
```

```
void thread_b(void)
{
    /* Transfer 1b */
    perform_transfer(3, 2, 3942);
    ...
    /* Transfer 2b */
    perform_transfer(5, 2, 93);
}
```