DM510: Deadlocks

Lars Rohwedder

Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

Today’s lecture

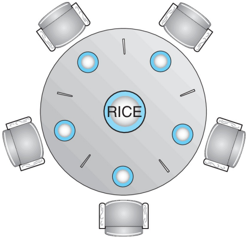
- Chapter 8 of course book

The Problem

Example: dining philosophers

- 5 philosophers alternatingly think and eat
- To eat they need to pick up their left and right chopstick (one at a time)
- Chopsticks (implemented as mutexes) are shared with the neighbors

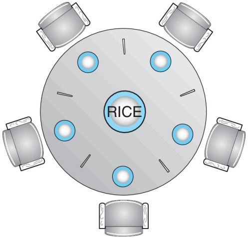
```
/* Algorithm for philosopher i */
while (true) {
        aquire(&chopstick[i]);
        aquire(&chopstick[(i+1)%5]);
        eat();
        release(&chopstick[i]);
        release(&chopstick[(i+1)%5]);
        think();
}
```



Example: dining philosophers

- 5 philosophers alternatingly think and eat
- To eat they need to pick up their left and right chopstick (one at a time)
- Chopsticks (implemented as mutexes) are shared with the neighbors

```
/* Algorithm for philosopher i */
while (true) {
        aquire(&chopstick[i]);
        aquire(&chopstick[(i+1)%5]);
        eat();
        release(&chopstick[i]);
        release(&chopstick[(i+1)%5]);
        think();
}
```



Deadlock

If each philosopher grabs the left chopstick before their neighbor grabs the right one, then they are stuck in a deadlock!

Formal model

- Deadlocks can occur with mutexes (last lecture), files, limited resources, etc.
- Since the problem is the same, we consider it in the following abstract model

Resources

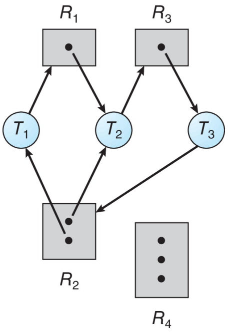
- $R_1, R_2, \ldots, R_m$: resources with one or more **instances**
- **Mutual exclusion:** only one thread can hold the same instance at a time

Threads

- $T_1, T_2, \ldots, T_n$: threads of the system

Edges

- **request edge:** from thread to resource
- **assignment edge:** from resource instance to thread

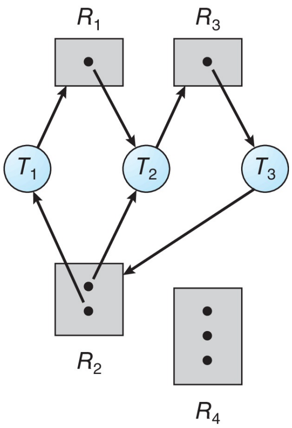


Deadlock characterization

Conditions for deadlock

- **Mutual exclusion** resource instances are held by one thread at a time
- **Hold and wait:** thread holding one resource instance waits for other resources
- **No preemption:** a resource can only be released voluntarily
- **Circular wait:** Threads $T_1, \ldots, T_n$ such that T_i waits for a resource that T_{i+1} (or T_1 if $i = n$) holds for each $i = 1, 2, \ldots, n$.

- Conditions necessary (no cycle $\Rightarrow$ no deadlock)

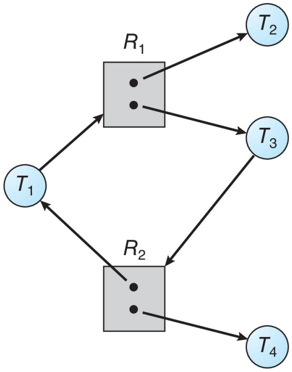


Cycle with deadlock

Deadlock characterization

Conditions for deadlock

- **Mutual exclusion** resource instances are held by one thread at a time
- **Hold and wait:** thread holding one resource instance waits for other resources
- **No preemption:** a resource can only be released voluntarily
- **Circular wait:** Threads $T_1, \ldots, T_n$ such that T_i waits for a resource that T_{i+1} (or T_1 if $i = n$) holds for each $i = 1, 2, \ldots, n$.



Cycle without deadlock

- Conditions necessary (no cycle $\Rightarrow$ no deadlock)
- But not sufficient (cycle $\Rightarrow$ maybe deadlock)

Handling deadlocks

- Ensure that system **never** enters deadlock state by **deadlock prevention** or **deadlock avoidance**
- Allow system to enter deadlock state and recover
- Ignore that there can be deadlocks

Deadlock Prevention

Deadlock prevention

It is enough to remove one of the four necessary conditions for a deadlock

Deadlock prevention

It is enough to remove one of the four necessary conditions for a deadlock

Mutual exclusion

- Make resource sharable if possible, e.g., read-only files

Deadlock prevention

It is enough to remove one of the four necessary conditions for a deadlock

Mutual exclusion

- Make resource sharable if possible, e.g., read-only files

Hold and wait

- Make threads request all resources before execution
- Make threads request resources only when none are allocated
- **Disadvantages of the above:** lower resource utilization, possible starvation

Deadlock prevention

It is enough to remove one of the four necessary conditions for a deadlock

Mutual exclusion

- Make resource sharable if possible, e.g., read-only files

Hold and wait

- Make threads request all resources before execution
- Make threads request resources only when none are allocated
- **Disadvantages of the above:** lower resource utilization, possible starvation

No preemption

- Thread waiting for some resource releases the ones it is currently holding
- Thread is woken up once the new resource and the previously released ones are allocated to it

Deadlock prevention

It is enough to remove one of the four necessary conditions for a deadlock

Mutual exclusion

- Make resource sharable if possible, e.g., read-only files

Hold and wait

- Make threads request all resources before execution
- Make threads request resources only when none are allocated
- **Disadvantages of the above:** lower resource utilization, possible starvation

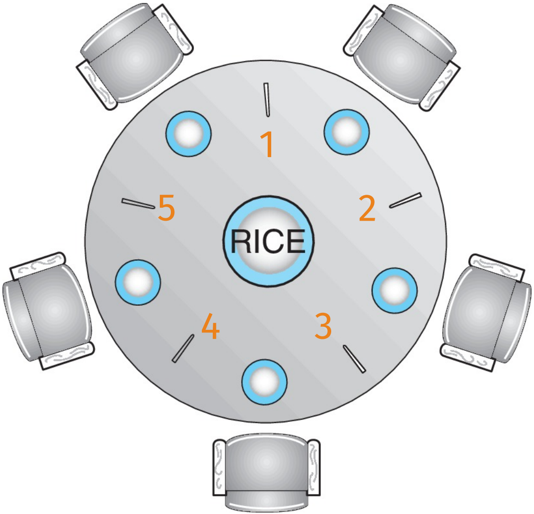
No preemption

- Thread waiting for some resource releases the ones it is currently holding
- Thread is woken up once the new resource and the previously released ones are allocated to it

Circular wait

- Define total order on resources and require threads to request resources in that order

Example: total order



Solution for dining philosophers

```
/* philosopher i=1,2,3,4 */
while (true) {
        aquire(&chopstick[i]);
        aquire(&chopstick[i+1]);
        eat();
        release(&chopstick[i]);
        release(&chopstick[i+1]);
        think();
}
/* for philosopher 5 */
while (true) {
        aquire(&chopstick[1]);
        aquire(&chopstick[5]);
        eat();
        release(&chopstick[1]);
        release(&chopstick[5]);
        think();
}
```

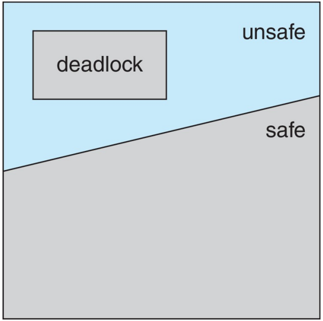
Deadlock Avoidance

Approach

- **Assumption:** we know the resources (and maximum no. instances) a thread can request and no new threads arrive
- Algorithm grants requested resources to threads in such a way that unsafe state (= potential deadlock) never reached

Safe state

- Threads can be reordered as $(T_1, T_2, \ldots, T_n)$ such that for each thread T_i the resource that $T_1, T_2, \ldots, T_{i-1}$ hold suffice to satisfy T_i's additional resource need
- No deadlock because T_1 does not wait; once T_1 is done, T_2 does not wait; once T_1, T_2 are done, T_3 does not wait; etc.



Detecting (un-)safe states

Input

- **avail** $\in \mathbb{Z}^{m}_{\geq 0}$: no. instances of resource not allocated
- **max** $\in \mathbb{Z}^{n\times m}_{\geq 0}$: maximum no. instances of resource a thread might request
- **alloc** $\in \mathbb{Z}^{n\times m}_{\geq 0}$: no. instances of resource allocated to thread

Safe state detection

1. initialize work $\in \mathbb{Z}^{m}_{\geq 0}$ with
$$\text{work}[j] = \text{avail}[j]$$
2. initialize need $\in \mathbb{Z}^{n\times m}_{\geq 0}$ with
$$\text{need}[i,j] = \max[i,j] - \text{alloc}[i,j]$$
3. initialize finish $\in \mathbb{Z}^{n}_{\geq 0}$ with
$$\text{finish}[i] = \text{false}$$
4. while true
   - 4.1 let $i \in \{1, 2, \ldots, n\}$ with
     $\text{finish}[i] = \text{false}$ and
     $\text{need}[i,j] \leq \text{work}[j]\ \forall j$
   - 4.2 if no such i exists: break
   - 4.3 $\text{finish}[i] \leftarrow \text{true}$
   - 4.4 for $j \in \{1, 2, \ldots, m\}$:
     $\text{work}[j] \leftarrow \text{work}[j] + \text{alloc}[i,j]$
5. **Result:** if $\text{finish}[i] = \text{false}$ for some thread T_i then state is unsafe, otherwise safe

Banker's algorithm for deadlock avoidance

- Banker's algorithm tests if request by a thread T_i can be granted
- If T_i has to wait, try again after resources have been released

Input

- **avail** $\in \mathbb{Z}^{m}_{\geq 0}$: no. instances of resource not allocated
- **max** $\in \mathbb{Z}^{n \times m}_{\geq 0}$: maximum no. instances of resource a thread might request
- **alloc** $\in \mathbb{Z}^{n \times m}_{\geq 0}$: no. instances of resource allocated to thread
- Thread T_i and request **req** $\in \mathbb{Z}^{m}_{\geq 0}$ for additional resources

Banker's algorithm

1. if $\mathrm{alloc}[i,j] + \mathrm{req}[j] > \max[i,j]$ for some j: raise runtime error
2. if $\mathrm{req}[j] > \mathrm{avail}[j]$ for some j: make T_i wait
3. store current state in S
4. for each resource R_j:
   /* *give requested resources to* T_i */
   $$\mathrm{avail}[j] \leftarrow \mathrm{avail}[j] - \mathrm{req}[j]$$
   $$\mathrm{alloc}[i,j] \leftarrow \mathrm{alloc}[i,j] + \mathrm{req}[j]$$
5. if unsafe state detected: restore state S and make T_i wait

Deadlock Recovery

Deadlock detection

Input

- **avail** $\in \mathbb{Z}^m_{\geq 0}$: no. instances of resource not allocated
- **alloc** $\in \mathbb{Z}^{n\times m}_{\geq 0}$: no. instances of resource allocated to a thread
- **req** $\in \mathbb{Z}^{n\times m}_{\geq 0}$: no. additional instances of resource requested by a thread

- Periodically check for deadlock
- Algorithm (right) requires $O(mn^2)$ operations

Algorithm

1. let work $\in \mathbb{Z}^m_{\geq 0}$ with
$$\text{work}[j] = \text{avail}[j]$$
2. let finish $\in \mathbb{Z}^n_{\geq 0}$ with
$$\text{finish}[i] = \begin{cases} \text{true} & \text{if alloc}[i,j] = 0 \ \forall j \\ \text{false} & \text{otherwise.} \end{cases}$$
3. while true
   - 3.1 let $i \in \{1, 2, \ldots, n\}$ with
   $$\text{finish}[i] = \text{false and}$$
   $$\text{req}[i,j] \leq \text{work}[j] \ \forall j$$
   - 3.2 if no such i exists: break
   - 3.3 $\text{finish}[i] \leftarrow \text{true}$
   - 3.4 for $j \in \{1, 2, \ldots, m\}$:
   $\text{work}[j] \leftarrow \text{work}[j] + \text{alloc}[i,j]$
4. **Result:** every thread i with $\text{finish}[i] = \text{false}$ is in deadlock

Recovering: process termination

Terminate threads (processes) to remove deadlock. Variants:

- Abort all threads (processes) in a deadlock
- Abort one thread (process) at a time until deadlock is removed. In which order? Examples:
    - By priority
    - By how long the thread has been computing or how much longer it needs
    - By resource usage
    - By resource requirements to complete
    - By number of threads that need to be terminated

Recovering: resource preemption

- pick a victim thread
- roll back thread to safe state and restart from there
- can lead to **starvation** if same thread is picked over and over again, to avoid: include number of times picked in victim selection