Resource Allocation, Dining Philosophers

Maruth Goyal

UT Austin

Spring 2021

Table of Contents

Resource Allocation

- Last time, we looked at Mutual Exclusion.
- There was one "resource" (the critical region), and no two processors were allowed to simultaneously use the resource (enter the region).

Resource Allocation

- Last time, we looked at Mutual Exclusion.
- There was one "resource" (the critical region), and no two processors were allowed to simultaneously use the resource (enter the region).
- Today, however, we will consider a generalization of this problem.
- **Resource Allocation**:
  - Consider the existence of many different resources.

Resource Allocation

- Last time, we looked at Mutual Exclusion.
- There was one "resource" (the critical region), and no two processors were allowed to simultaneously use the resource (enter the region).
- Today, however, we will consider a generalization of this problem.
- **Resource Allocation**:
  - Consider the existence of many different resources.
  - Users may require different combinations of these resources to perform their tasks.

Resource Allocation

- Last time, we looked at Mutual Exclusion.
- There was one "resource" (the critical region), and no two processors were allowed to simultaneously use the resource (enter the region).
- Today, however, we will consider a generalization of this problem.
- **Resource Allocation**:
  - Consider the existence of many different resources.
  - Users may require different combinations of these resources to perform their tasks.
  - The restrictions on which users can simultaneously use resources are (potentially) weaker than mutual exclusion.

Resource Allocation

- Last time, we looked at Mutual Exclusion.
- There was one "resource" (the critical region), and no two processors were allowed to simultaneously use the resource (enter the region).
- Today, however, we will consider a generalization of this problem.
- **Resource Allocation**:
  - Consider the existence of many different resources.
  - Users may require different combinations of these resources to perform their tasks.
  - The restrictions on which users can simultaneously use resources are (potentially) weaker than mutual exclusion.
  - eg: Two database transactions trying to perform I/O on disjoint disk pages should be able to proceed unhindered, unless there is a dependency of some sort between the pages (for instance, indexing information).

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$
- Some different ways to specify which users can proceed simultaneously:

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$
- Some different ways to specify which users can proceed simultaneously:
- **Expicit Resource Specification**:
  - For each user U_i, specify a set $R_i \subseteq R$ indicating which resources are required by the user.

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$
- Some different ways to specify which users can proceed simultaneously:
- **Expicit Resource Specification**:
  - For each user U_i, specify a set $R_i \subseteq R$ indicating which resources are required by the user.
  - *Exlcusion policy:* Any two users U_i, U_j such that $R_i \cap R_j \neq \emptyset$ may not proceed simultaneously.

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$
- Some different ways to specify which users can proceed simultaneously:
- **Expicit Resource Specification**:
  - For each user U_i, specify a set $R_i \subseteq R$ indicating which resources are required by the user.
  - *Exlcusion policy:* Any two users U_i, U_j such that $R_i \cap R_j \neq \emptyset$ may not proceed simultaneously.
- **General Resource Specification**:
  - Define a set E of subsets of $U = \{U_i\}$, closed under superset.

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$
- Some different ways to specify which users can proceed simultaneously:
- **Expicit Resource Specification**:
  - For each user U_i, specify a set $R_i \subseteq R$ indicating which resources are required by the user.
  - *Exlcusion policy:* Any two users U_i, U_j such that $R_i \cap R_j \neq \emptyset$ may not proceed simultaneously.
- **General Resource Specification**:
  - Define a set E of subsets of $U = \{U_i\}$, closed under superset.
  - Each set in E defines a set of users such that it is invalid for them to proceed simultaneously.

Resource Allocation

- More formally, we consider a set of resources $R = \{r_1, \ldots, r_n\}$
- Some different ways to specify which users can proceed simultaneously:
- **Expicit Resource Specification**:
  - For each user U_i, specify a set $R_i \subseteq R$ indicating which resources are required by the user.
  - *Exlcusion policy:* Any two users U_i, U_j such that $R_i \cap R_j \neq \emptyset$ may not proceed simultaneously.
- **General Resource Specification**:
  - Define a set E of subsets of $U = \{U_i\}$, closed under superset.
  - Each set in E defines a set of users such that it is invalid for them to proceed simultaneously.
- Not equivalent methods of specification:
  $R = \{r_1\}$, $U = \{u_1, u_2, u_3\}$, $E = \{\{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$

Resource Allocation

- The computational model will be similar to last time.

Resource Allocation

- The computational model will be similar to last time.
- For each resource r_i we will have an assosciated **read-modify-write** variable in shared memory accessible by all users which use that resource.

Resource Allocation

- The computational model will be similar to last time.
- For each resource r_i we will have an assosciated **read-modify-write** variable in shared memory accessible by all users which use that resource.
- Each user U_i has a corresponding agent A_i which it corresponds with using the same API as last time in order to gain access to resources.

Resource Allocation

- The complete specification is mostly similar to mutual exclusion:

Definition (Solving Resource Allocation)

A shared memory system A solves the resource allocation problem for a collection of users if:

1. (**Well-formedness**): In any execution, and for any i, the subsequence describing the interaction between U_i and A is well-formed for i (i.e., follows cyclic pattern of API).

Resource Allocation

- The complete specification is mostly similar to mutual exclusion:

Definition (Solving Resource Allocation)

A shared memory system A solves the resource allocation problem for a collection of users if:

1. (**Well-formedness**): In any execution, and for any i, the subsequence describing the interaction between U_i and A is well-formed for i (i.e., follows cyclic pattern of API).
2. (**Exclusion**): There is no reachable system state (that is, a combination of an automaton state for A and states for all the U_i) in which the set of users in the critical section is in E.

Resource Allocation

- The complete specification is mostly similar to mutual exclusion:

Definition (Solving Resource Allocation)

A shared memory system A solves the resource allocation problem for a collection of users if:

1. (**Well-formedness**): In any execution, and for any i, the subsequence describing the interaction between U_i and A is well-formed for i (i.e., follows cyclic pattern of API).
2. (**Exclusion**): There is no reachable system state (that is, a combination of an automaton state for A and states for all the U_i) in which the set of users in the critical section is in E.
3. (**Progress**): At any point in *fair execution*:
    1. If at least one user is in T and no user is in C, then at some later point *some* user enters C
    2. If at least one user is in E, then at some later point some user enters R.

Dining Philosophers Problem

- To help understand resource allocation in general, it is often helpful to look at a simpler special case, famously known as the **Dining Philosophers problem**.

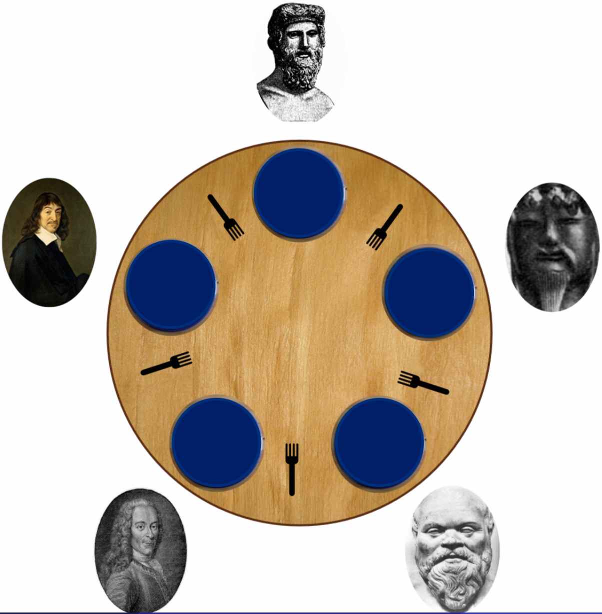
Dining Philosophers Problem

- To help understand resource allocation in general, it is often helpful to look at a simpler special case, famously known as the **Dining Philosophers problem**.

Definition (Dining Philosophers Problem [Dijkstra, 1971])

- N Philosophers are sitting around a round table having dinner.
- Because they are Philosophers they are normally in a THINKING state.
- Occasionally, they may snap out of their thoughts and decide to enter an EATING state. However, to do so they must pick up the two forks adjacent to them (who knows why, philosophers are a mystery).
- Here's the catch: there are N forks. i.e., each philosopher has one fork on either side of them.
- What protocol should the philosophers follow so that someone in the EATING state can pick up both forks, without a fight starting between philosophers.

Dining Philosophers



Dining Philosophers

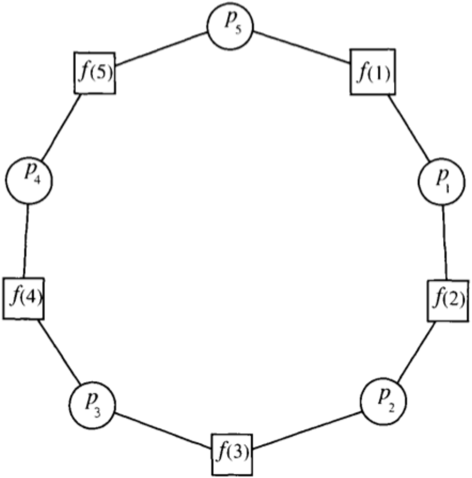


Figure 11.1: Dining Philosophers problem ($n = 5$).

Table of Contents

Solving Dining Philosophers

- First off, we will start by ruling out a whole class of solutions.
- In particular, any "symmetric" algorithm cannot solve dining philosophers.
  - Eg: if everyone tries picks up their left fork first, this will result in deadlock.

Solving Dining Philosophers

- First off, we will start by ruling out a whole class of solutions.
- In particular, any "symmetric" algorithm cannot solve dining philosophers.
  - Eg: if everyone tries picks up their left fork first, this will result in deadlock.
- The argument for why is identical to the one used for leader election.
- Thus we shall restrict our attention to algorithms which utilize further structure in the problem.

Solving Dining Philosophers

- Since everyone picking the same (relative) fork can result in deadlock, we need to somehow break symmetry.
- Somewhat like leader election, we will again utilize the UID of the philosopher.

Solving Dining Philosophers

- Since everyone picking the same (relative) fork can result in deadlock, we need to somehow break symmetry.
- Somewhat like leader election, we will again utilize the UID of the philosopher.
- In particular, we will make it such that the parity of the UID determines the behavior.

Solving Dining Philosophers

Dining Philosophers Solution

1. Philosophers with odd UID will try to get their right fork first, while even UIDs will try to get their left fork first.

Solving Dining Philosophers

Dining Philosophers Solution

1. Philosophers with odd UID will try to get their right fork first, while even UIDs will try to get their left fork first.
2. The **read-modify-write** variables will be used to represent the underlying data of a synchronized queue of length 2. (CAS loop)

Solving Dining Philosophers

Dining Philosophers Solution

1. Philosophers with odd UID will try to get their right fork first, while even UIDs will try to get their left fork first.
2. The **read-modify-write** variables will be used to represent the underlying data of a synchronized queue of length 2. (CAS loop)
3. To acquire a fork, a philosopher inserts their UID into the queue.

Solving Dining Philosophers

Dining Philosophers Solution

1. Philosophers with odd UID will try to get their right fork first, while even UIDs will try to get their left fork first.
2. The **read-modify-write** variables will be used to represent the underlying data of a synchronized queue of length 2. (CAS loop)
3. To acquire a fork, a philosopher inserts their UID into the queue.
4. If they're the first in queue, the fork is their's and they may proceed into the critical section.

Solving Dining Philosophers

Dining Philosophers Solution

1. Philosophers with odd UID will try to get their right fork first, while even UIDs will try to get their left fork first.
2. The **read-modify-write** variables will be used to represent the underlying data of a synchronized queue of length 2. (CAS loop)
3. To acquire a fork, a philosopher inserts their UID into the queue.
4. If they're the first in queue, the fork is their's and they may proceed into the critical section.
5. When they exit, they pop their entries off the respective queues.

Solving Dining Philosophers

Dining Philosophers Solution

1. Philosophers with odd UID will try to get their right fork first, while even UIDs will try to get their left fork first.
2. The **read-modify-write** variables will be used to represent the underlying data of a synchronized queue of length 2. (CAS loop)
3. To acquire a fork, a philosopher inserts their UID into the queue.
4. If they're the first in queue, the fork is their's and they may proceed into the critical section.
5. When they exit, they pop their entries off the respective queues.
6. The other philosopher (if applicable) in the queue may either (1) keep polling the queue to see if it is now first, or (2) be sent an interrupt or something.

Solving Dining Philosophers

Transitions of i:

try_i
Effect:
$pc := test\text{-}right$

$test\text{-}right_i$
Precondition:
$pc = test\text{-}right$
Effect:
if i is not on $f(i).queue$ then
add i to $f(i).queue$
if i is first on $f(i).queue$ then
$pc := test\text{-}left$

$test\text{-}left_i$
Precondition:
$pc = test\text{-}left$
Effect:
if i is not on $f(i+1).queue$ then
add i to $f(i+1).queue$
if i is first on $f(i+1).queue$ then
$pc := leave\text{-}try$

$crit_i$
Precondition:
$pc = leave\text{-}try$
Effect:
$pc := crit$

$exit_i$
Effect:
$pc := reset\text{-}right$

$reset\text{-}right_i$
Precondition:
$pc = reset\text{-}right$
Effect:
remove i from $f(i).queue$
$pc := reset\text{-}left$

$reset\text{-}left_i$
Precondition:
$pc = reset\text{-}left$
Effect:
remove i from $f(i+1).queue$
$pc := leave\text{-}exit$

rem_i
Precondition:
$pc = leave\text{-}exit$
Effect:
$pc := rem$

Table of Contents

Dining Philosophers Analysis

Theorem

The aforementioned algorithm solves the dining philosophers problem.

Dining Philosophers Analysis

Theorem

The aforementioned algorithm solves the dining philosophers problem.

- The well-formedness follows immedeately from the structure of the code.

Dining Philosophers Analysis

Theorem

The aforementioned algorithm solves the dining philosophers problem.

- The well-formedness follows immedeately from the structure of the code.

Theorem

The aforementioned algorithm guarantees exclusion as required by the Dining Philosophers problem.

Dining Philosophers Analysis

Theorem

The aforementioned algorithm solves the dining philosophers problem.

- The well-formedness follows immedeately from the structure of the code.

Theorem

The aforementioned algorithm guarantees exclusion as required by the Dining Philosophers problem.

Proof.

1. Suppose for contradiction two philosophers end up using a fork at the same time.
2. In order for this to happen, both of them must have been first in the synchronized queue at the same time. $\perp$

$\square$

Dining Philosophers Analysis

- In order to prove progress, we will actually show something stronger by bounding the maximum waiting time between when a philosopher begins trying to acquire the forks, and when they get them. We will assume number of processors n is even.

Dining Philosophers Analysis

- In order to prove progress, we will actually show something stronger by bounding the maximum waiting time between when a philosopher begins trying to acquire the forks, and when they get them. We will assume number of processors n is even.

Theorem

In the aforementioned algorithm, the time from when a process i enters TRY *until it enters* CRITICAL *is at most $3c + 18\ell$ where c is an upper bound on the amount of time spent by any process in the critical region, and ℓ an upper bound on the time it takes a processor to take a single step.*

Dining Philosophers Analysis

Theorem

In the aforementioned algorithm, the time from when a process i enters $\textsc{Try}$ *until it enters* $\textsc{Critical}$ *is at most $3c + 18\ell$.*

Proof.

- Suppose a process is trying to get its first fork.
  - Suppose it gets it immedeately, and then spends at most S time getting its second fork and enterring the critical region, then the toal time is $\ell + S$.

$\square$

Dining Philosophers Analysis

Theorem

In the aforementioned algorithm, the time from when a process i enters Try *until it enters* Critical *is at most $3c + 18\ell$.*

Proof.

- Suppose a process is trying to get its first fork.
  - Suppose it gets it immedeately, and then spends at most S time getting its second fork and enterring the critical region, then the toal time is $\ell + S$.
  - Suppose another process is using the fork. Then, since number of processors is even, it must also be its first fork. Thus, it will take at most $S + c + \ell$ time for the process to release the fork, and an additional $\ell + S$ for the first process to enter the region. Total $c + 2\ell + 2S$.

□

Dining Philosophers Analysis

Proof Cont'd.

- It remains to bound S.
  - Suppose it immedeately gets the second fork.
    - Then, time taken is ℓ to test the second fork, ℓ to acquire, and ℓ to go into critical section.
    - Total 3ℓ

Dining Philosophers Analysis

Proof Cont'd.

- It remains to bound S.
  - Suppose it immedeately gets the second fork.
    - Then, time taken is ℓ to test the second fork, ℓ to acquire, and ℓ to go into critical section.
    - Total 3ℓ
  - Suppose another process has second fork.
    - Again, because of even number of processors, must be that processor's second fork too.
    - The time taken by that processor is at most $2\ell + c + 2\ell$

□

Dining Philosophers Analysis

Proof Cont'd.

- It remains to bound S.
  - Suppose it immedeately gets the second fork.
    - Then, time taken is ℓ to test the second fork, ℓ to acquire, and ℓ to go into critical section.
    - Total 3ℓ
  - Suppose another process has second fork.
    - Again, because of even number of processors, must be that processor's second fork too.
    - The time taken by that processor is at most $2\ell + c + 2\ell$
    - i.e., time to acquire fork and enter critical region, exit critical region, and pop off queue.

□

Dining Philosophers Analysis

Proof Cont'd.

- It remains to bound S.
  - Suppose it immedeatELY gets the second fork.
    - Then, time taken is ℓ to test the second fork, ℓ to acquire, and ℓ to go into critical section.
    - Total 3ℓ
  - Suppose another process has second fork.
    - Again, because of even number of processors, must be that processor's second fork too.
    - The time taken by that processor is at most $2\ell + c + 2\ell$
    - i.e., time to acquire fork and enter critical region, exit critical region, and pop off queue.
    - After that, original processor needs 2ℓ steps to acquire and enter critical region.
    - Total $c + 8\ell$.

Dining Philosophers Analysis

Proof Cont'd.

- It remains to bound S.
  - Suppose it immedeately gets the second fork.
    - Then, time taken is ℓ to test the second fork, ℓ to acquire, and ℓ to go into critical section.
    - Total 3ℓ
  - Suppose another process has second fork.
    - Again, because of even number of processors, must be that processor's second fork too.
    - The time taken by that processor is at most $2\ell + c + 2\ell$
    - i.e., time to acquire fork and enter critical region, exit critical region, and pop off queue.
    - After that, original processor needs 2ℓ steps to acquire and enter critical region.
    - Total $c + 8\ell$.
- Taking maximums and subtituting, we get total time T to critical region satisfies $T \leq 3c + 18\ell$.

□

Table of Contents

Generalization

- This algorithm for the Dining Philosopher's problem can be generalized to work for any explicit resource specification.
- However, we require we are given some **total order** on the resources.

Generalization

- This algorithm for the Dining Philosopher's problem can be generalized to work for any explicit resource specification.
- However, we require we are given some **total order** on the resources.
- The algorithm then proceeds by having processes attempting to acquire resources in the order specificed by the total order from least to greatest.
- Deadlock-freedom is guaranteed since there are no waiting cycles by virtue of the order.

Generalization

- This algorithm for the Dining Philosopher's problem can be generalized to work for any explicit resource specification.
- However, we require we are given some **total order** on the resources.
- The algorithm then proceeds by having processes attempting to acquire resources in the order specificed by the total order from least to greatest.
- Deadlock-freedom is guaranteed since there are no waiting cycles by virtue of the order.
- The choice of order can have a big impact on performance.
- Naïve total order can lead to a chain of $n - 1$ processes waiting on the next one.

Coloring Algorithm

- To reduce the length of the chain, we consider a more clever order. [Lynch, 1981]

Coloring Algorithm

- To reduce the length of the chain, we consider a more clever order. [Lynch, 1981]
- Consider the graph with resources as nodes, with edges iff there exist two processes which require both resources.
- We consider any coloring χ of this graph, and totally order the colors arbitrarily.

Coloring Algorithm

- To reduce the length of the chain, we consider a more clever order. [Lynch, 1981]
- Consider the graph with resources as nodes, with edges iff there exist two processes which require both resources.
- We consider any coloring χ of this graph, and totally order the colors arbitrarily.
- Then, we consider the partial order induced on resources by the order on colors.

Coloring Algorithm

- To reduce the length of the chain, we consider a more clever order. [Lynch, 1981]
- Consider the graph with resources as nodes, with edges iff there exist two processes which require both resources.
- We consider any coloring χ of this graph, and totally order the colors arbitrarily.
- Then, we consider the partial order induced on resources by the order on colors.
- Now, every processor has a total order on the set of resources it needs and proceeds as usual.

Coloring Algorithm

- To reduce the length of the chain, we consider a more clever order. [Lynch, 1981]
- Consider the graph with resources as nodes, with edges iff there exist two processes which require both resources.
- We consider any coloring χ of this graph, and totally order the colors arbitrarily.
- Then, we consider the partial order induced on resources by the order on colors.
- Now, every processor has a total order on the set of resources it needs and proceeds as usual.

Theorem

Using the above algorithm, if k colors are used in χ, and at most m processors require any single resource, then the time between a process going from T to C is at most $O(m^k c + km^k \ell)$.

References I

Dijkstra, E. W. (1971).
Hierarchical ordering of sequential processes.
In *The origin of concurrent programming*, pages 198–227. Springer.

Lynch, N. A. (1981).
Upper bounds for static resource allocation in a distributed system.
J. Comput. Syst. Sci., 23(2):254–278.

Questions?

Thank You!