

CPSC 457

Dining Philosophers, Locks, Mutexes

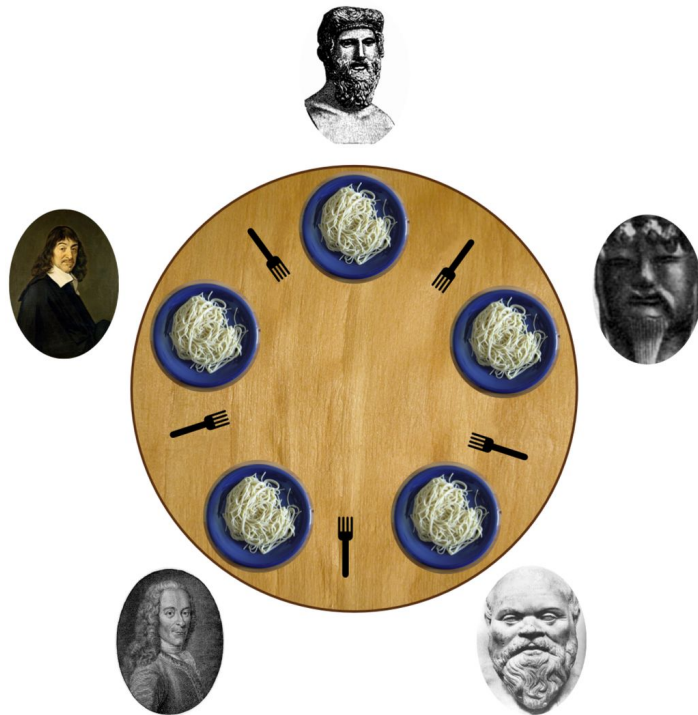
Contains slides from Mea Wang, Andrew Tanenbaum and Herbert Bos

Outline

- dining philosophers
- locks
- mutexes

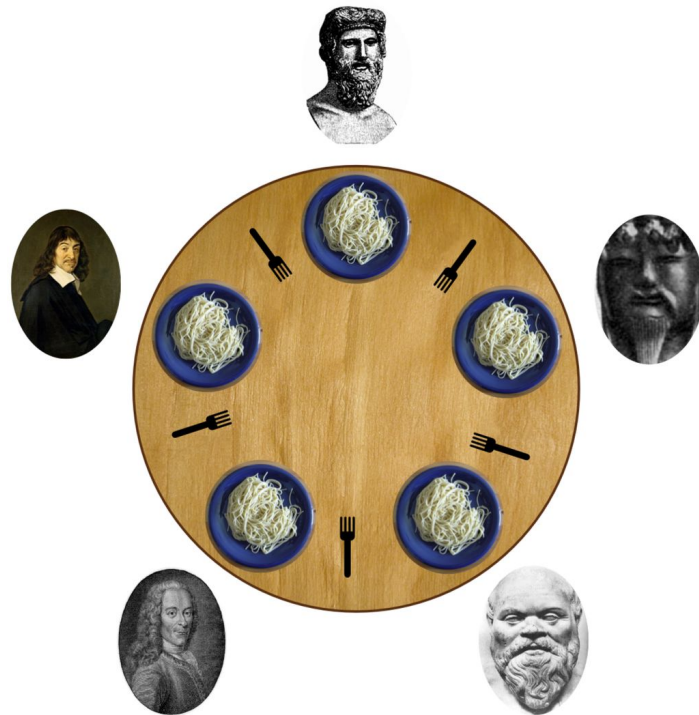
Dining philosophers problem

- 5 philosophers sitting around a table
- philosophers want to do 2 things, forever:
 - eat, and then
 - think
- 5 bowls of food, one for each philosopher
 - noodles, rice, ...
- 5 forks placed between bowls
 - or chopsticks



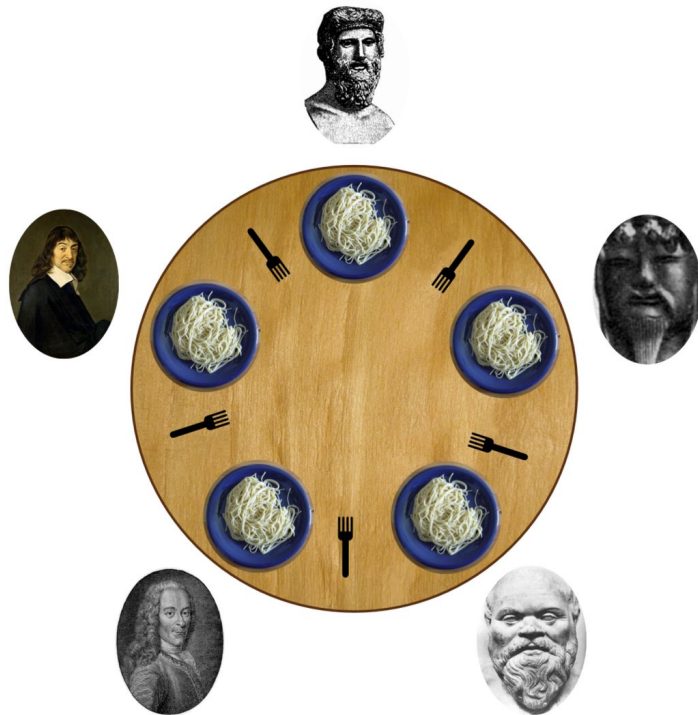
Dining philosophers problem

- before eating, a philosopher must first grab two forks, immediately to the left & right
- philosopher then eats for a short time
- when done eating, the philosopher puts down the forks in their original positions
- philosopher then thinks for a short time



Dining philosophers problem

- software scenario:
5 processes/threads, each needs exclusive access to two resources (eg. files), to proceed
- how to allocate resources so that all process/threads get to execute?
- what is the *best* algorithm for threads/processes to follow?
 - how do we define '*best*'?
 - depends on the the objective...



Dining philosophers problem

- assuming each philosopher eats & thinks for the exact same amount of time, optimal schedule:

repeat:

```
1 & 3 eat  
2 & 4 eat  
3 & 5 eat  
4 & 1 eat  
5 & 2 eat
```

- is there a simple way to code this?
- remember - each philosopher represents an independent thread or process
- and what if some philosophers think/eat more than others?



Attempt 1

- each philosopher follows these steps (algorithm):

repeat forever:

- grab left fork
- grab right fork
- eat
- put forks back
- think

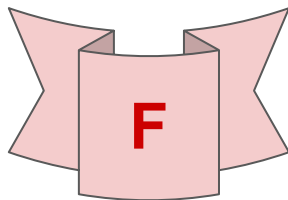
- would this work?

Attempt 1

- each philosopher follows these steps (algorithm):

repeat forever:

```
grab left fork  
grab right fork  
eat  
put forks back  
think
```



- would this work?
- this would lead to a **deadlock**:
 - assuming all philosophers are reasonably synchronized
 - each philosopher will end up grabbing the left fork
 - each philosopher will be 'stuck' trying grab the right fork
 - nobody gets to eat at all

Attempt 2

- each philosopher follows these steps (algorithm):

repeat forever:

repeat:

try to grab left fork

try to grab right fork

if both forks grabbed then break

else put any grabbed forks back and take a short nap

eat

put forks back

think

- would this work?

Attempt 2

- each philosopher follows these steps (algorithm):

repeat forever:

repeat:

try to grab left fork

try to grab right fork

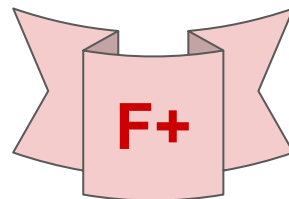
if both forks grabbed then break

else put any grabbed forks back and take a short nap

eat

put forks back

think



- would this work?
- they would reach a **livelock**
 - assuming they are all synchronized
 - all philosophers will indefinitely switch between napping and attempting to eat
 - nobody will eat – form of **starvation**

Attempt 3

- same as before, but there is one pink hat

repeat forever:

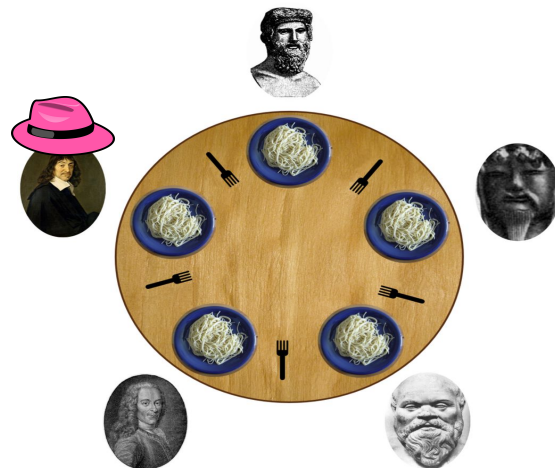
wait for hat

grab forks, eat, put forks back

give hat to someone else

think

- would this work?



Attempt 3

- same as before, but there is one pink hat

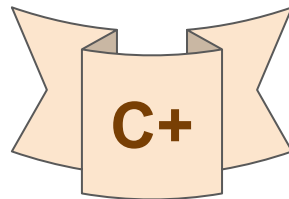
repeat forever:

wait for hat

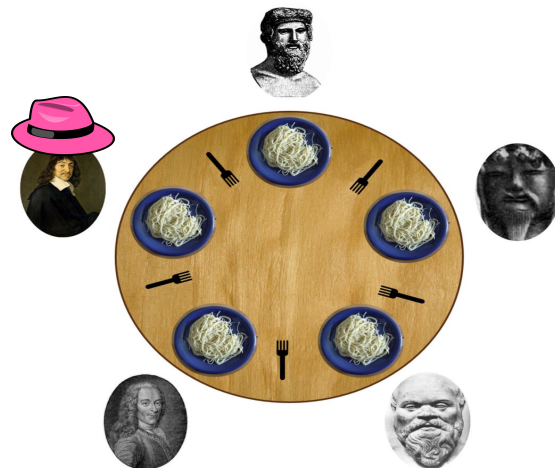
grab forks, eat, put forks back

give hat to someone else

think



- would this work? yes it would, but...
 - only one philosopher is eating at any given time
 - but with 5 forks, 2 philosophers **could** be eating at the same time
 - **non-optimal** use of resources, resulting in reduced parallelism
 - called arbitrator solution



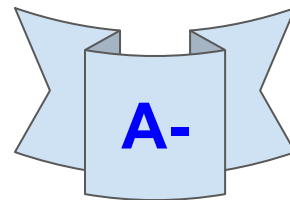
Attempt 4

```
repeat forever:
  repeat:
    try to grab left fork
    try to grab right fork
    if both forks grabbed then break out of loop
    else put any grabbed forks back and take a short RANDOM nap
  eat
  put forks back
  think
```

- would this work?

Attempt 4

```
repeat forever:
  repeat:
    try to grab left fork
    try to grab right fork
    if both forks grabbed then break out of loop
    else put any grabbed forks back and take a short RANDOM nap
  eat
  put forks back
  think
```



- **random timeout** mechanism for preventing deadlocks
- likely to work, often used in real world, for example in networking
- but... still a small chance for starvation:
 - if sleep is the same for all philosophers, no one gets to eat
 - in some cases some philosophers might never get to eat, or some philosophers will get to eat less often than others (**fairness problem**)

Attempt 5

- label the forks with numbers: 1, 2, 3, 4, 5
- each philosopher:
 - picks up the fork with the smallest number first, then the larger number second
 - if unable to pick up both forks, put a claimed fork down and take a short nap
- this is a **resource hierarchy** solution - by establishing a **partial order** on resources
- starvation is still possible
- also not always practical for large/dynamic number of resources
- further attempts left as a homework:
 - two hats, even/odd philosophers, pick left/right forks randomly, hungry vs more hungry queues, ...

Naive algorithm implementation

- even trying to implement a naive solution presents problems
- consider algorithm for philosopher 'i':

```
// global variable representing utencil state
// 0 = unavailable, 1 = available
int forks[5];

while (true)
{
    sleep (s); // think for s seconds
    while (!forks[i] || !forks[i+1]); // i+1 modulo 5 arithmetic
    forks[i] = false;
    forks[i+1] = false;
    sleep (m); // eat for m seconds
    forks[i] = true;
    forks[i+1] = true;
}
```


Naive algorithm implementation

```

while (true)
{
  sleep (s);
  1 while (!forks[i] || !forks[i+1]);
  3 forks[i] = false;
  forks[i+1] = false;
  sleep (m);
  forks[i] = true;
  forks[i+1] = true;
}

```

```

while (true)
{
  sleep (s);
  2 while (!forks[i] || !forks[i+1]);
  forks[i] = false;
  forks[i+1] = false;
  sleep (m);
  forks[i] = true;
  forks[i+1] = true;
}

```

- depending on the execution order (eg. multi-core machines, or timing of context switches)
 - both philosophers could start eating at the same time
 - i.e. both processes could enter the critical region

Algorithm with critical sections

- the shared resource is the global variable `forks[]`
- let's identify critical sections:

```
while (true)
{
    sleep (s);

    while (!forks[i] || !forks[i+1]);
    forks[i] = false;
    forks[i+1] = false;

    sleep (m);

    forks[i] = true;
    forks[i+1] = true;
}
```

now we need a
mechanism to guarantee
mutual exclusion

Mutex (aka Lock)

- **mutex** is a synchronization mechanism used for ensuring **exclusive access** to a resource in concurrent programs
- think of it as a special boolean type that can represent a lock
 - TRUE → locked
 - FALSE → unlocked
- we can set it to True or False, just like a regular boolean variable
- but, if the lock is already locked, and some thread tries to also lock it then the calling thread will be automatically suspended...
until whoever locked the lock unlocks it



Mutex (aka Lock)

pseudocode:

```
mutex lck;
```

```
lck = TRUE; // will block if already TRUE
```

```
//
```

```
// critical section
```

```
//
```

```
lck = FALSE; // may unblock some other thread
```



Mutex (aka Lock)

- mutex is often implemented as an object with two possible states: **locked** and **unlocked**
- two operations: **lock()** and **unlock()**
- if multiple threads call **lock()** simultaneously, only one will proceed, the others will block
- only the thread that locks the mutex can unlock it
- a waiting queue is used to keep track of all threads waiting on the mutex to be unlocked
- once the mutex is unlocked, one of the blocked threads will be unlocked...
 - note: which one thread gets unlocked is usually not predictable
- can be implemented in software via busy waiting,
but usually supported by hardware + OS
- portable libraries often try to use H/W mutex, but are able to fall back to software

Mutexes in pthreads

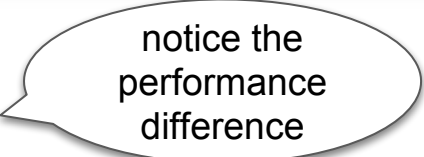
API	Description
<code>pthread_mutex_init()</code>	create a mutex
<code>pthread_mutex_destroy()</code>	destroy a mutex
<code>pthread_mutex_lock()</code>	lock a mutex, block if already locked
<code>pthread_mutex_trylock()</code>	lock a mutex, or fail (non-blocking version)
<code>pthread_mutex_unlock()</code>	unlock a mutex

Counter with mutex

```
#include <pthread.h>
pthread_mutex_t count_mutex; // initialized in main() with pthread_mutex_init()
int counter;                 // initialized in main() with counter = 0

void incr() {
    pthread_mutex_lock(&count_mutex); // acquire the lock
    int x = counter;
    x = x + 1;
    counter = x;
    pthread_mutex_unlock(&count_mutex); // release the lock
}
```

- recall counter example with race condition <https://repl.it/M0oi/0>
- same example fixed with a mutex: <https://repl.it/M0sG/1>



notice the
performance
difference

Dining philosopher with mutex

```
pthread_mutex_t mutex; // a lock, initialized elsewhere

while (true) {
    sleep (s); // think
    pthread_mutex_lock(&mutex);
    while (!forks[i] || !forks[i+1]);
    forks[i] = false;
    forks[i+1] = false;
    pthread_mutex_unlock(&mutex);
    sleep (m); // eating
    pthread_mutex_lock(&mutex);
    forks[i] = true;
    forks[i+1] = true;
    pthread_mutex_unlock(&mutex);
}
```

Would this work?

Dining philosopher with mutex

```
pthread_mutex_t mutex; // a lock, initialized elsewhere
```

```
while (true) {  
    sleep (s); // think  
    pthread_mutex_lock(&mutex);  
    while (!forks[i] || !forks[i+1]);  
    forks[i] = false;  
    forks[i+1] = false;  
    pthread_mutex_unlock(&mutex);  
  
    sleep (m); // eating  
    pthread_mutex_lock(&mutex);  
    forks[i] = true;  
    forks[i+1] = true;  
    pthread_mutex_unlock(&mutex);  
}
```

The diagram illustrates the execution flow of the code. A red line with a red pentagon labeled '1' indicates the start of the first iteration. It follows the code lines: `sleep (s);`, `pthread_mutex_lock(&mutex);`, `while (!forks[i] || !forks[i+1]);`, `forks[i] = false;`, `forks[i+1] = false;`, and `pthread_mutex_unlock(&mutex);`. A red arrow points from the end of this sequence to a second red pentagon labeled '3'. A blue line with a blue pentagon labeled '2' indicates the start of the second iteration. It follows the code lines: `sleep (s);`, `pthread_mutex_lock(&mutex);`, and `while (!forks[i] || !forks[i+1]);`. A blue arrow points from the end of this sequence back to the blue pentagon '2', forming a loop. This represents a deadlock state where both philosophers are waiting for forks that are held by each other.

Deadlock !!!

Summary

- **critical section** - part of the program where a shared resource is accessed & may cause trouble
- **mutual exclusion** - ensuring only one process accesses a resource at a time, eg. only one process can enter critical section at any given time
- **mutex/lock** - mechanism to achieve mutual exclusion, two states + queue
- **deadlock** - a state where each process/thread is waiting on another to release a lock → no progress is made
- **livelock** - states of the processes change, but none are progressing
- **starvation** - one process does not get to run at all
- **unfairness** - not all processes get equal opportunity to progress

- concurrent programming is hard

Summary

- thread issues
 - fork(), cancellation, signals, thread pool
- race condition
 - critical section, mutual exclusion
- dining philosophers problem
- mutex
- deadlock, livelock, starvation, fairness

Reference: 2.2.3, 2.2.7, 2.3.1 - 2.3.3 (Modern Operating Systems)
4.3 - 4.4, 6.1 - 6.2, 6.6 (Operating System Concepts)

Review

- Name/explain two general approaches for cancelling a thread.
- Are signals handled per thread or per process?
- Define:
 - race condition, critical region, mutual exclusion
 - deadlock, livelock, starvation
- Race condition is not a problem among processes, only among threads.

True or False?

- A mutex has only two states: locked and unlocked.

True or False?

- more tutorials on dining philosophers:

<http://cs.mtu.edu/~shene/NSF-3/e-Book/MUTEX/TM-example-philos-1.html>

<http://web.eecs.utk.edu/~mbeck/classes/cs560/560/notes/Dphil/lecture.html>

Questions?