Locks, Mutexes, and Dining Philosophers

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

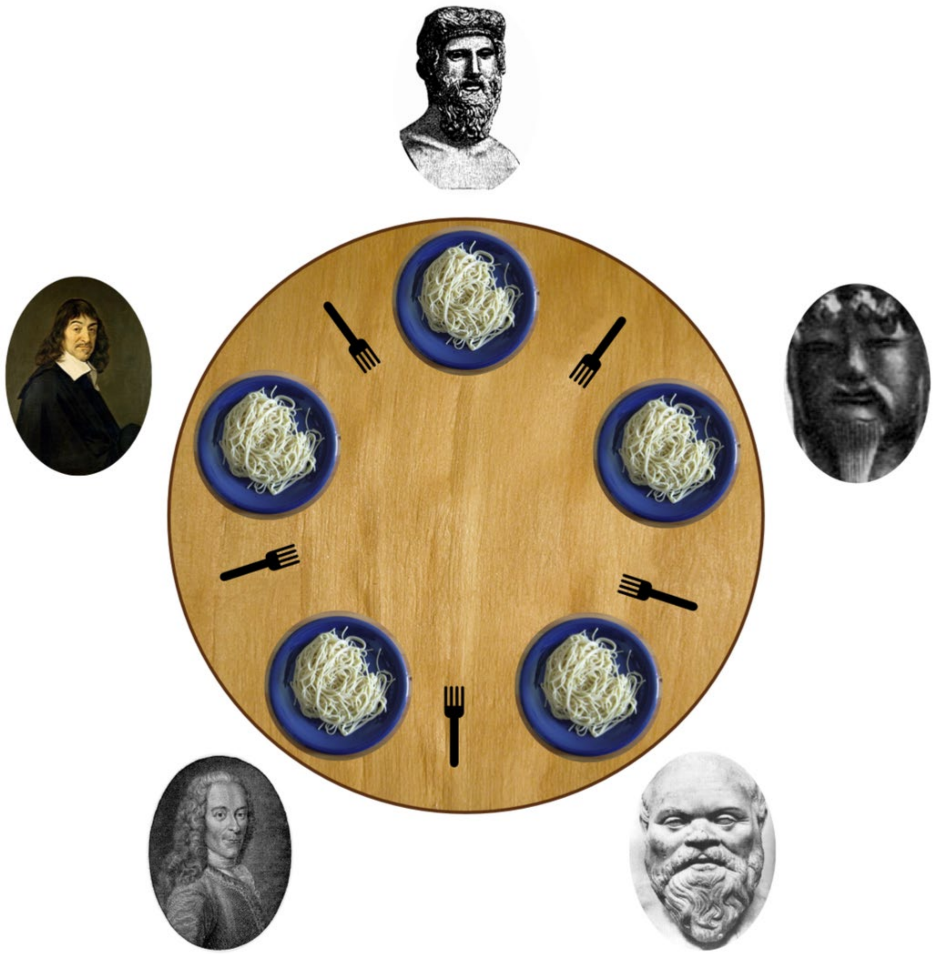
Topics

- dining philosophers
- locks
- mutexes

Dining Philosophers

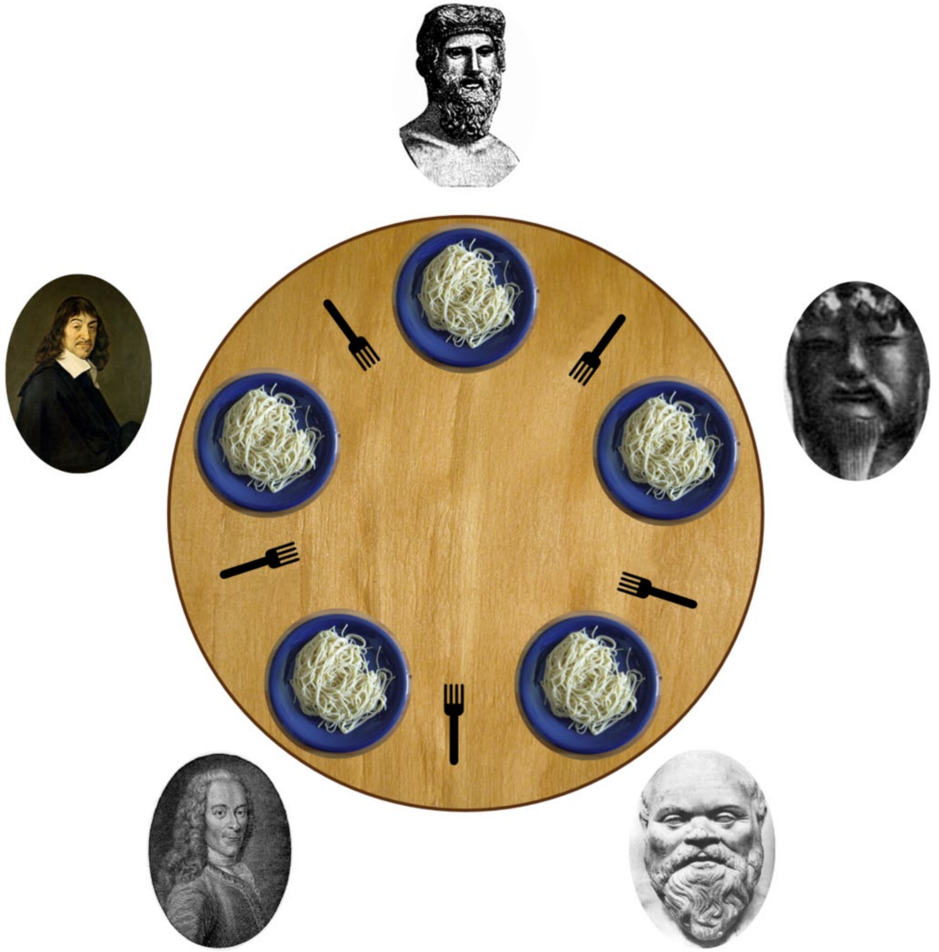
Dining philosophers problem

- 5 philosophers sitting around a table
- 5 bowls of food, one for each philosopher
- 5 forks placed between bowls
- philosophers alternate between eating and thinking
- philosophers don't mind sharing forks



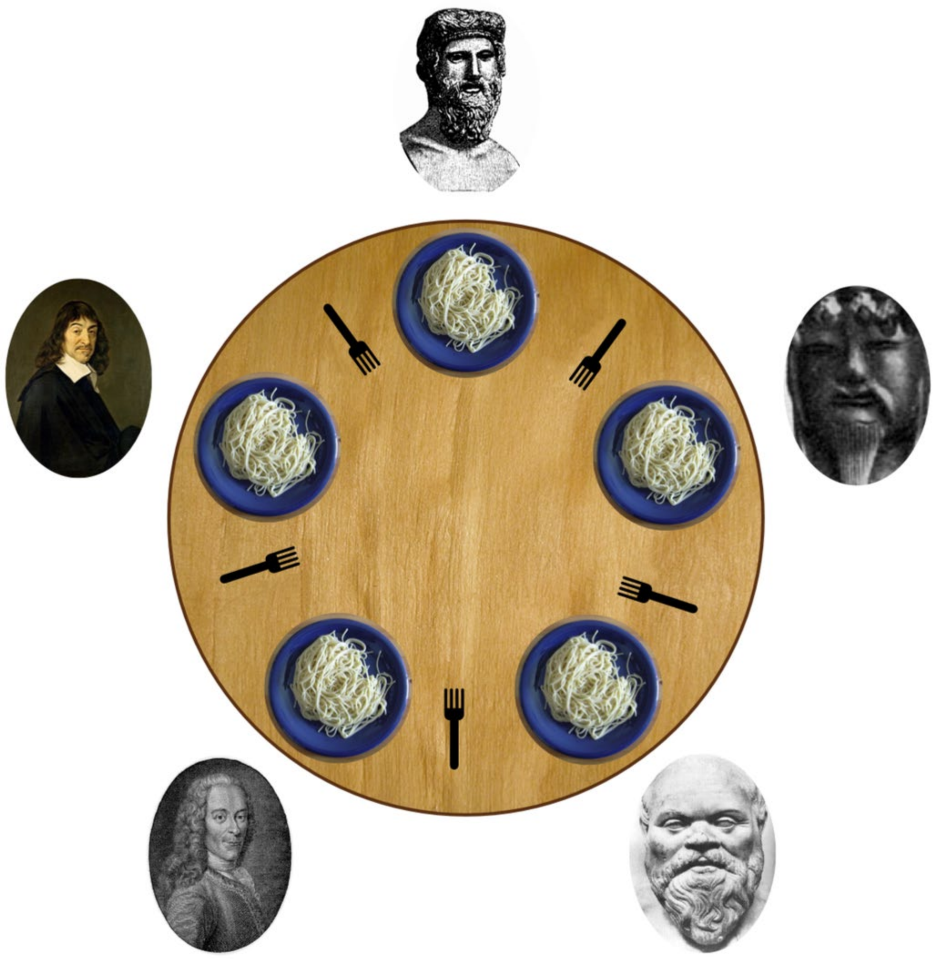
Dining philosophers problem

- before eating, a philosopher must first grab both 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 frequent exclusive access to two resources (e.g. each needs to update 2 files)
- 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...
  - what are we trying to optimize?



UNIVERSITY OF CALGARY

Dining philosophers problem

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

  repeat:

  philosophers 1 & 3 eat

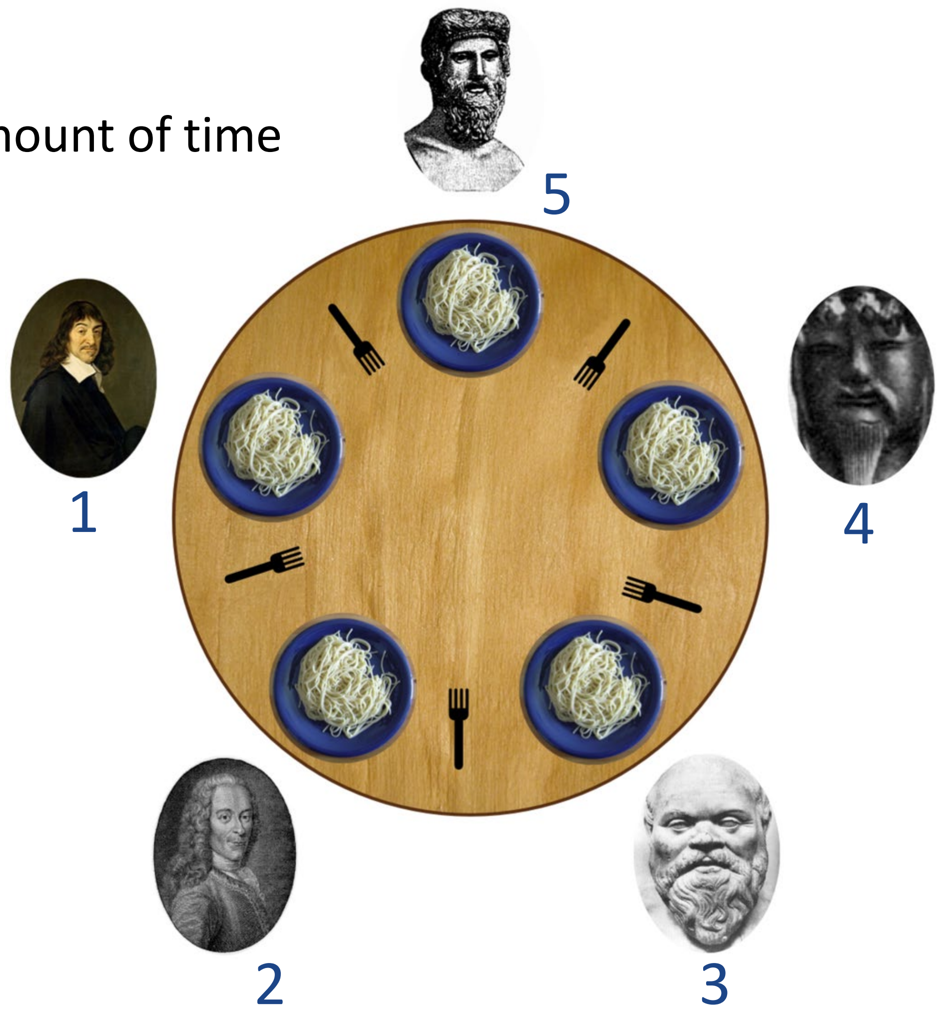
  philosophers 2 & 4 eat

  philosophers 3 & 5 eat

  philosophers 4 & 1 eat

  philosophers 5 & 2 eat

- is there a simple way to code this?
  remember that each philosopher represents an independent thread or process
- not optimal if some philosophers think/eat more than others



Dining Philosophers Attempt to do Something

Attempt 1

- each philosopher follows these steps (algorithm):

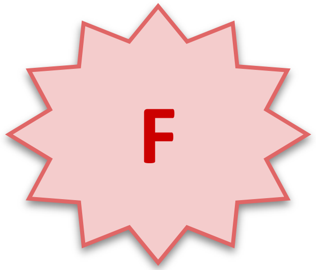
```
repeat forever:
       grab left fork
       grab right fork
       eat
       put forks back
       think
```

- 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?
- no, this could lead to a **deadlock**:
  - assuming all philosophers are reasonably synchronized
  - each philosopher could end up grabbing the left fork
  - then 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
```

F+

- would this work?
- philosophers could reach a **livelock**
  - every philosopher grabs left fork, but fails to grab right fork
  - all philosophers would 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

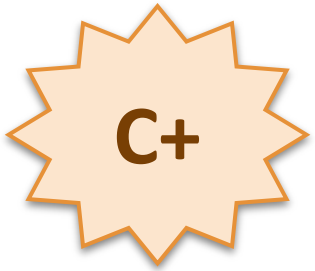
```
repeat forever:
    wait for a hat
    grab forks, eat, put forks back
    give hat to "someone" else
    think
```

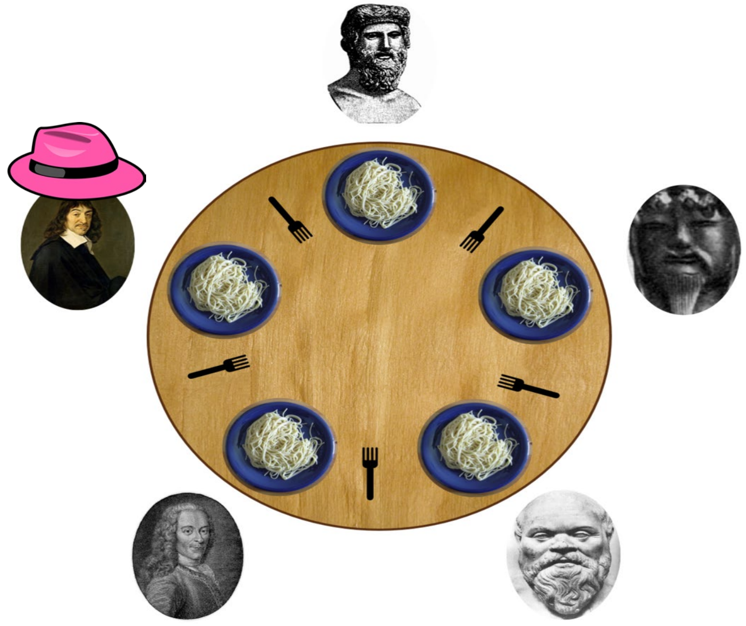
- would this work?

Attempt 3

- same as before, but there is one pink hat

```
repeat forever:
    wait for a 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

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

A-

```
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
```

- the random nap will desynchronize the philosophers and is likely to work over long time
- sometimes used in real world, e.g. in networking (Exponential backoff)
- but…
  - if nap time is the same for neighbors, they do not get to eat (temporary starvation)
  - some philosophers might sleep longer than others, and eat less often (fairness problem)

Attempt 5

B+

- 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

- called a **resource hierarchy** solution – by establishing a partial order on resources

- starvation is still possible, although very unlikely
- reduced parallelism in general cases
  - e.g. already have lock on 2, 3, but now need 1, must first release 2, 3, then re-acquire 1, 2, 3

- it is not always practical for large and/or 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, ...

Algorithms

Naive algorithm implementation

- let's try a naive implementation of a philospher
- consider algorithm #1 for philosopher 'i':

```
// global variable representing fork state
// false = unavailable, true = available
bool 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;
}
```

Critical Sections

Naive algorithm implementation

```
while (true)
{
  sleep (s); // think
  while
(!forks[i]||!forks[i+1]){;}

  forks[i] = false;
  forks[i+1] = false;
  sleep (m); // eat
  forks[i] = true;
  forks[i+1] = true;
}
```

```
while (true)
{
  sleep (s); // think
  while
(!forks[i]||!forks[i+1]){;}

  forks[i] = false;
  forks[i+1] = false;
  sleep (m); // eat
  forks[i] = true;
  forks[i+1] = true;
}
```

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

Algorithm with critical sections

- the shared resource is the global variable `forks[]`
- let's identify critical sections (parts of code that use the shared resource):

```
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;
}
```

critical section 1: `while (!forks[i] || !forks[i+1]);` `forks[i] = false;` `forks[i+1] = false;`

critical section 2: `forks[i] = true;` `forks[i+1] = true;`

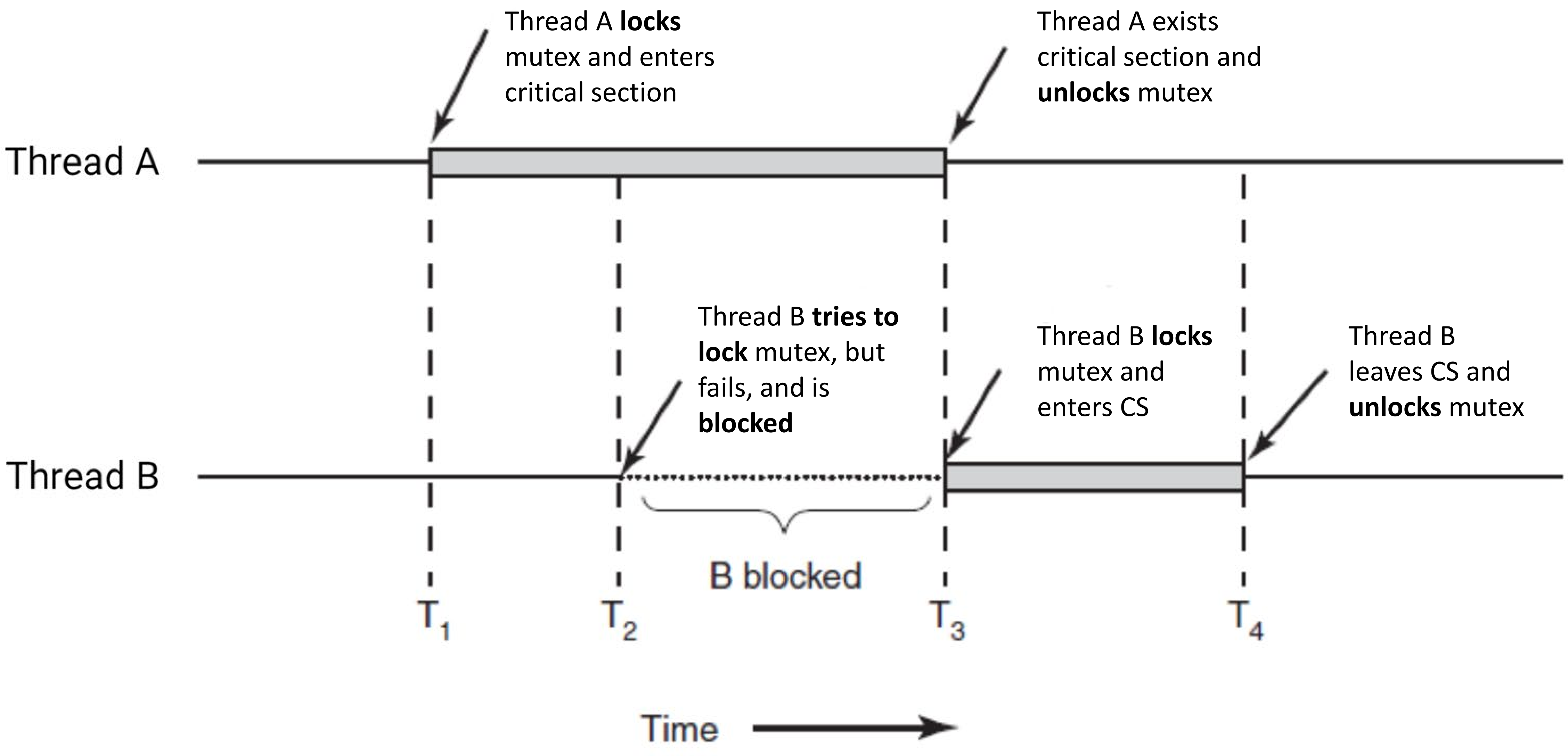
- now we need a mechanism to protect these sections via mutual exclusion

Mutexes

Mutex (aka Lock)

- **mutex** is a synchronization primitive, usually used for ensuring **exclusive access** to a resource in concurrent programs
- mutex has two possible states: **locked** and **unlocked**, and two atomic operations: `lock()` and `unlock()`
- if multiple threads call `lock()` simultaneously, only one will proceed, the rest 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 waiting 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 will try to use H/W mutex, but are able to fall back to software

Using mutexes to protect critical sections



Mutex (aka Lock)

pseudocode

```
// initialize mutex and share across all threads,
// e.g. via global variable
mutex m;

// in each thread
void run()
{
     non-critical_section_code
     // before entering critical section, lock the mutex
     lock(m);
     // now it's safe to access a shared resource
     critical_section_code
     // to exit CS, we unlock the mutex
     unlock(m);
     non-critical_section_code
}
```

Pthreads mutex

API	Description
`pthread_mutex_init()`	initialize a new mutex (unlocked state)
`pthread_mutex_destroy()`	destroy a mutex
`pthread_mutex_lock()`	try to lock a mutex, block if already locked
`pthread_mutex_unlock()`	unlock a mutex
`pthread_mutex_trylock()`	try to lock a mutex, or fail (non-blocking version of lock)

Counter with Mutex

Counter with mutex (pthreads)

```
#include <pthread.h>

pthread_mutex_t count_mutex; // must be initialized with pthread_mutex_init(),
e.g. in main()
int counter;                 // initialized with counter = 0, e.g. in main()

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
}
```

all 3 lines must be in the same critical section !

notice the performance difference

Counter with mutex (C++ mutex)

```
#include <mutex>

std::mutex m;                    // no need to further initialize
int counter = 0;

void incr() {
  m.lock();                      // acquire the lock
    counter ++;
  m.unlock();                    // release the lock
}
```

notice the performance difference

Dining Philosopher with Mutexes

Dining philosopher with mutex

Would this work?

```
pthread_mutex_t mutex;

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);   // eat

    pthread_mutex_lock(&mutex);
     forks[i] = true;
     forks[i+1] = true;
    pthread_mutex_unlock(&mutex);
}
```

Dining philosopher with mutex

```
pthread_mutex_t mutex;

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);   // eat

    pthread_mutex_lock(&mutex);
     forks[i] = true;
     forks[i+1] = true;
    pthread_mutex_unlock(&mutex);
}
```

1 thread 1 claims forks and starts to eat

2 thread 2 attempts to claim forks, but gets stuck in while loop

3 thread 1 finishes eating and attempts to return forks, but gets stuck unable to lock mutex

Deadlock !!!

UNIVERSITY OF CALGARY

Review

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 wheare 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

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

Onward to … condition variables and semaphores

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/