Concurrency basics

CSE 536 Spring 2024

jedimaestro@asu.edu

Outline

- Review about race conditions and locks
- Deadlocks and starvation
- Semaphores
- Producer consumer
- Dining philosophers
- Mutex’s, monitors and, futex’s

Review: this is a race condition without the lock

- <u>Thread #1</u>

  lock(L)

  x := x + 1

  unlock(L)

  Lock L

  Move x into Register

  Add 1 to Register

  Move Register into x

  Unlock L

- <u>Thread #2</u>

  lock(L)

  x := x + 1

  unlock(L)

  Lock L

  Move x into Register

  Add 1 to Register

  Move Register into x

  Unlock L

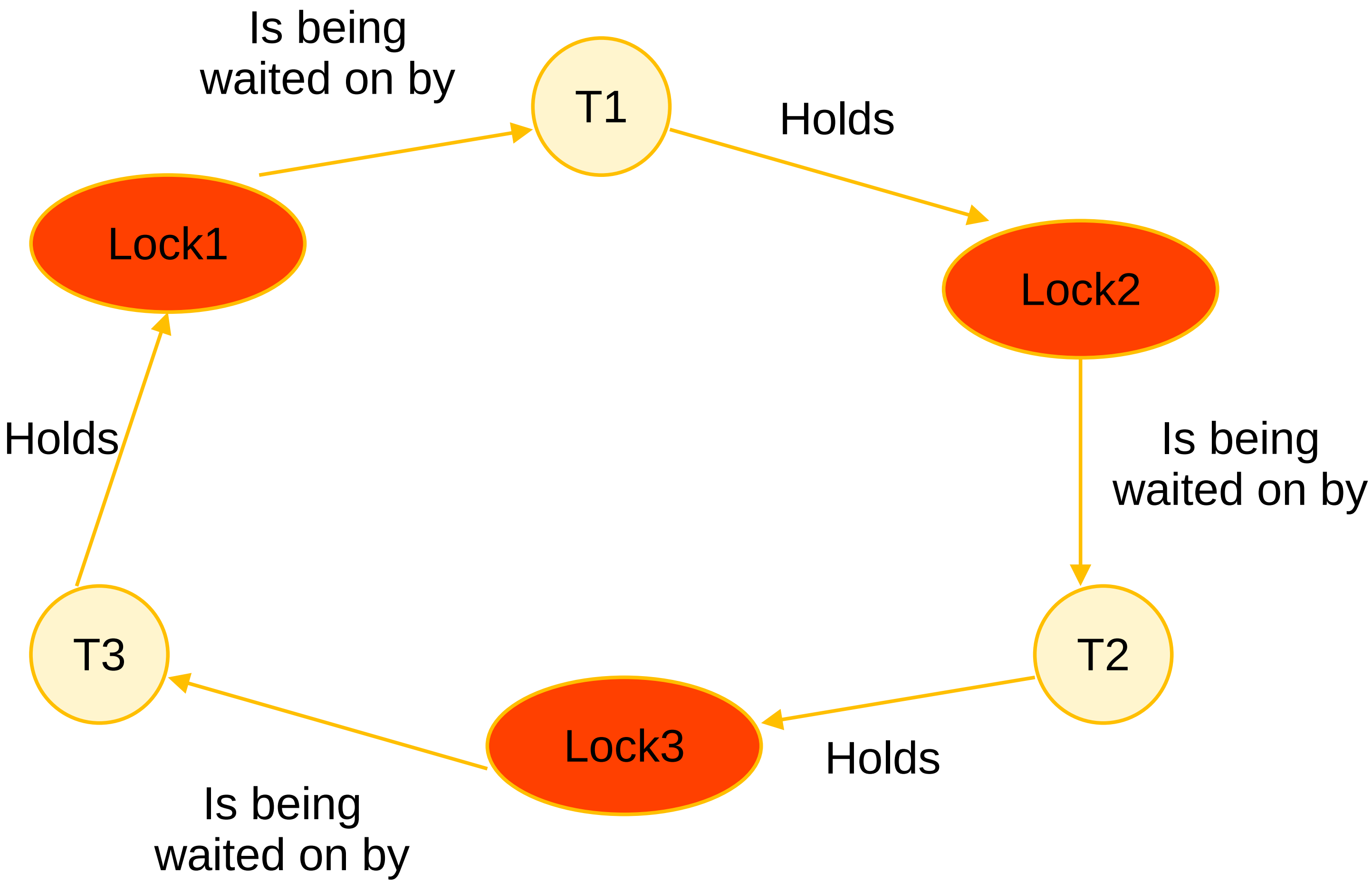
Terminology: the code between the lock and unlock is called the *critical section*.

Source: Patrick Bridges' slides…

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/31-Concurrency-Bugs-Deadlock.pdf

Deadlock conditions

- All four conditions must be met for deadlock to occur, i.e., if you break any of these you have mitigated deadlocks
  - Mutual exclusion (exclusive access to resources)
  - Hold-and-wait (hold resources while obtaining others)
  - No preemption (can't take resources away from threads)
  - Circular wait (circular chain of threads waiting on resources)


Is being
waited on by
T1
Holds
Lock1
Lock2
Holds
Is being
waited on by
T3
T2
Lock3
Holds
Is being
waited on by

Break circular wait

- Programming discipline, no OS support needed
- Always grab locks in the same order
- *E.g.*, always grab Lock1 before Lock2, and always grab Lock3 last

Breaking hold-and-wait

- Grab all locks at the same time, atomically, by defining a global lock, *e.g.*:

```
Lock(GlobalLock);
Lock(Lock1);
Lock(Lock3);
Unlock(GlobalLock);
```

- Not good for parallelism

Breaking no preemption

```
top:
        lock(L1);
        if( tryLock(L2) == -1 ){
                unlock(L1);
                goto top;
        }
```

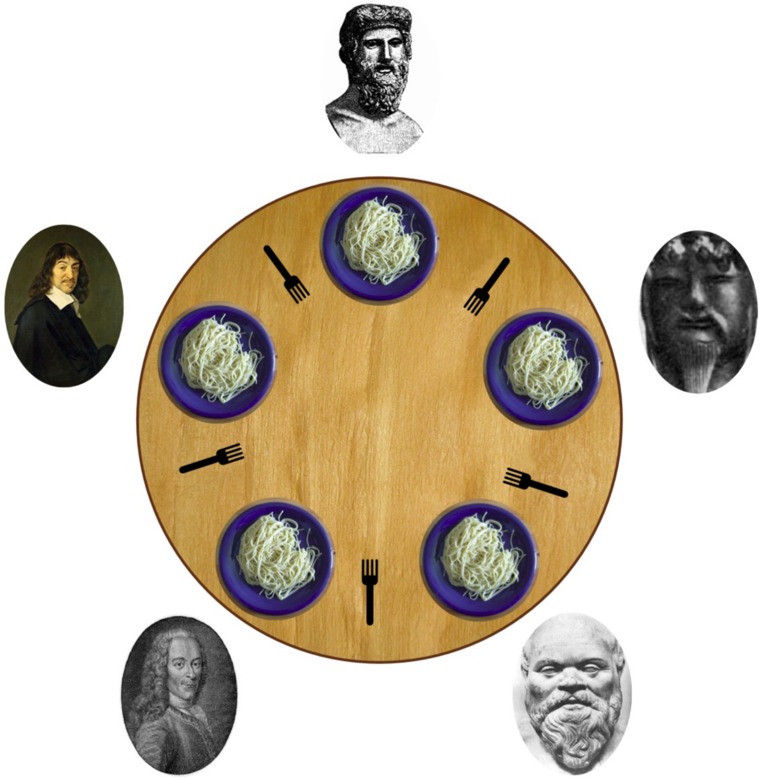
- Problem: live lock
- Solution: random delay

Breaking mutual exclusion

```c
int CompareAndSwap(int *address, int expected, int new){
    if(*address == expected){
        *address = new;
        return 1; // success
    }
    return 0;
}
```

```c
void insert(int value) {
    node_t *n = malloc(sizeof(node_t));
    assert(n != NULL);
    n->value = value;
    do {
        n->next = head;
    } while (CompareAndSwap(&head, n->next, n));
}
```

https://en.wikipedia.org/wiki/Dining_philosophers_problem



Requirements

- No deadlocks
- No starvation
- High degree of parellelism

Semaphores

- Invented by Edsger Dijkstra in 1962 or 1963
- https://en.wikipedia.org/wiki/Semaphore_(programming)

https://en.wikipedia.org/wiki/Electrologica_X8#/media/File:Electrologica_X8.jpg

https://en.wikipedia.org/wiki/Semaphore_(programming)#/media/File:Rail-semaphore-signal-Dave-F.jpg

Semaphore operations

- wait
  - Also known as
    - P
    - proberen
    - prolaag
    - down
    - acquire

- signal
  - Also known as
    - V
    - verhogen
    - vrijgave
    - up
    - release

Things we can do with semaphores

- Locks
  - *a.k.a.* binary semaphores
- Producer-consumer
  - uses binary and counting semaphores
- Dining philosophers solution

Atomic operations

```
function V(semaphore S, integer I):
    [S ← S + I]

function P(semaphore S, integer I):
    repeat:
        [if S ≥ I:
        S ← S − I
        break]
```

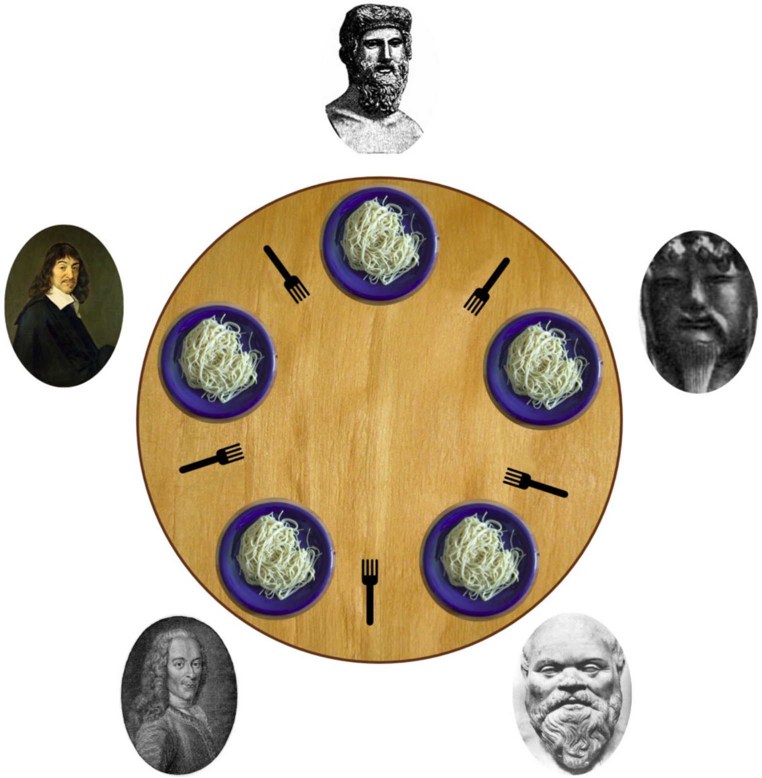
Producer-Consumer Problem

- Producer produces items
- Consumer consumes them
- Can have multiple producers and consumers running in parallel
- Requirements:
  - Concurrency (if there’s work to do and a thread to do it, they should do it...)
  - No race conditions

```
produce:
    P(emptyCount)
    P(useQueue)
    putItemIntoQueue(item)
    V(useQueue)
    V(fullCount)
```

```
consume:
    P(fullCount)
    P(useQueue)
    item ← getItemFromQueue()
    V(useQueue)
    V(emptyCount)
```

https://en.wikipedia.org/wiki/Dining_philosophers_problem



```
void getforks() {
sem_wait(forks[left(p)]);
sem_wait(forks[right(p)]);
}

void putforks() {
sem_post(forks[left(p)]);
sem_post(forks[right(p)]);
}
```

The getforks() and putforks() Routines (Broken Solution)

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/30-Concurrency-Bugs-Non-Deadlock.pdf

```
void getforks() {
sem_wait(forks[left(p)]);
sem_wait(forks[right(p)]);
}

void putforks() {
sem_post(forks[left(p)]);
sem_post(forks[right(p)]);
}
```

Deadlock!

The `getforks()` and `putforks()` Routines (Broken Solution)

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/30-Concurrency-Bugs-Non-Deadlock.pdf

```
void getforks() {
if (p == 4) {
        sem_wait(forks[right(p)]);
        sem_wait(forks[left(p)]);
} else {
        sem_wait(forks[left(p)]);
        sem_wait(forks[right(p)]);
}
}
```

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/30-Concurrency-Bugs-Non-Deadlock.pdf

https://en.wikipedia.org/wiki/Lock_(computer_science)

- “While a binary semaphore may be colloquially referred to as a mutex, a true mutex has a more specific use-case and definition, in that only the task that locked the mutex is supposed to unlock it.”
- Basic problem with sempahores: you have no idea which thread is holding which resource
- “a true mutex has a more specific use-case and definition, in that only the task that locked the mutex is supposed to unlock it”
  - Implies OS support, or some type of runtime environment + memory safety
- If you wrap a mutex in an object-like programming construct you can call it a monitor
  - Ada, C#, Java, Go, Mesa, Python, ...

Problems with semaphores

- Priority inversion (*vs.* OS can do priority inheritance)
- Premature task termination (*vs.* OS can release mutexes)
- Termination deadlock (*vs.* OS can release mutexes)
- Recursion deadlock (*vs.* mutexes can be reentrant)
- Accidental release (*vs.* OS can raise an error)

Back down to hardware-level and OS-level things (slides by Patrick Bridges)…

https://www.cs.unm.edu/~crandall/operatingsystems20/slides/26-Concurrency-Critical-Sections-2.pdf

Need OS support

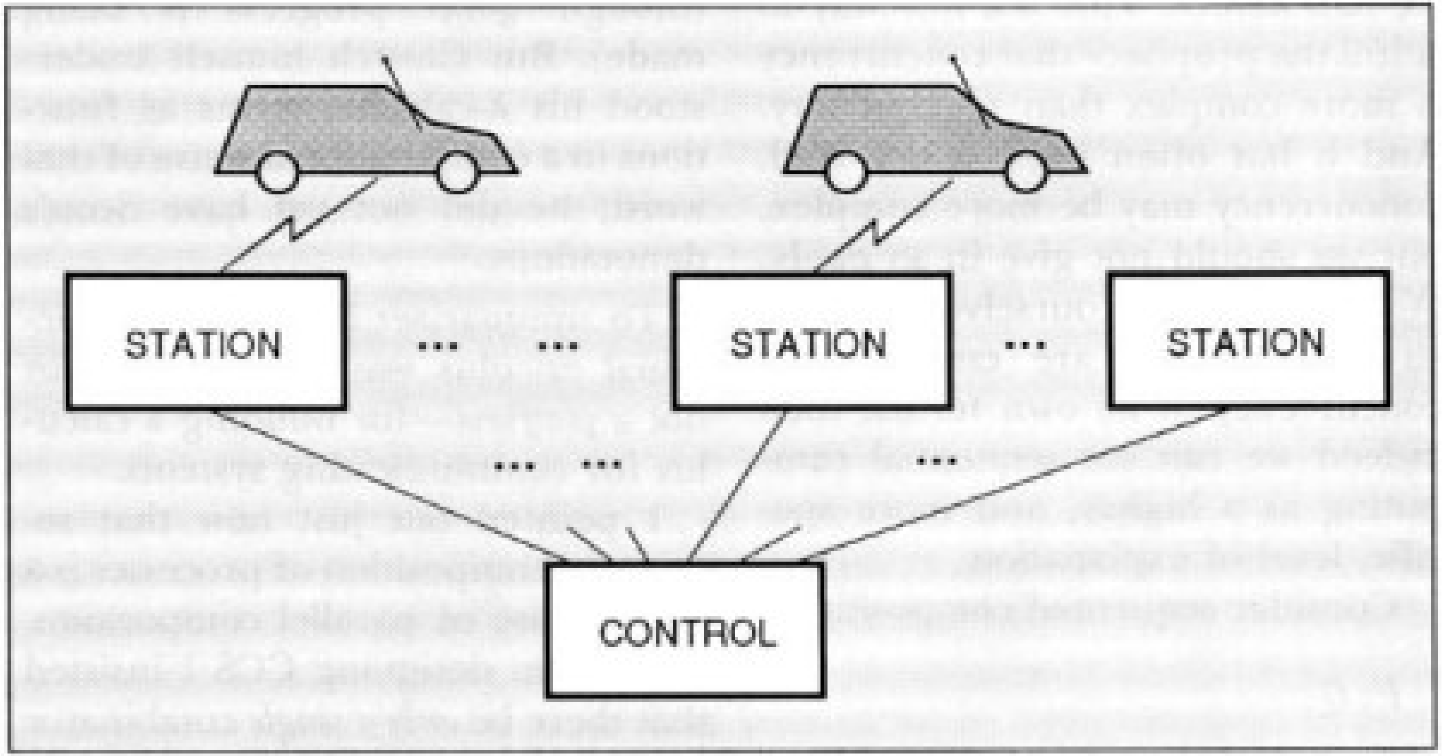
- Spinning to wait for a lock uses up 100% of a CPU when you're scheduled
- Do this instead...

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Linux's futex (similar to `setpark`, `park`, and `unpark` on Solaris)

- `futex_wait(address, expected)`
  - Put the calling thread to sleep
  - If the value at `address` is not equal to `expected`, the call returns immediately.
- `futex_wake(address)`
  - Wake one thread that is waiting on the queue.

Can we use semaphores, mutexes, *etc.* for this?



https://dl.acm.org/doi/pdf/10.1145/151233.151240

Coming up...

- poll(), select(), and epoll()
  - Event-based and asynchronous I/O
- Message passing
- Remote Procedure Calls