CSCE 313-200
Introduction to Computer Systems
Spring 2024

Deadlocks II

Dmitri Loguinov
Texas A&M University

March 18, 2024

Chapter 6: Roadmap

6.1 Principles

6.6 Dining philosophers

6.2 Prevention

6.3 Avoidance

6.4 Detection

6.5 Integrated strategies

6.7 Unix

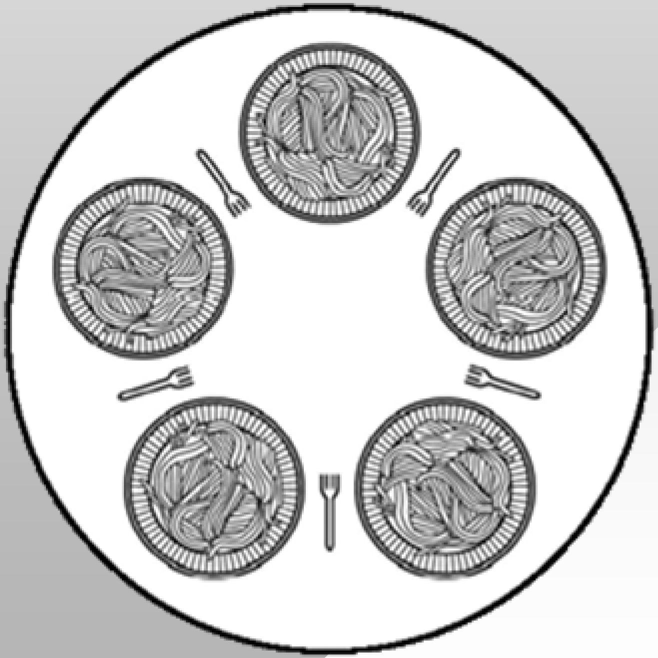
6.8 Linux

6.9 Solaris

6.10 Windows

Dining Philosophers

- Yet another famous synchronization problem
  - Proposed by Dijkstra in 1965
- N philosophers are sitting at a round table with N forks between them
  - Usually N = 5 and the food is spaghetti, but this is not essential
- Each thinks for a random period of time until becoming hungry, then attempts to eat
  - Food requires usage of both adjacent forks

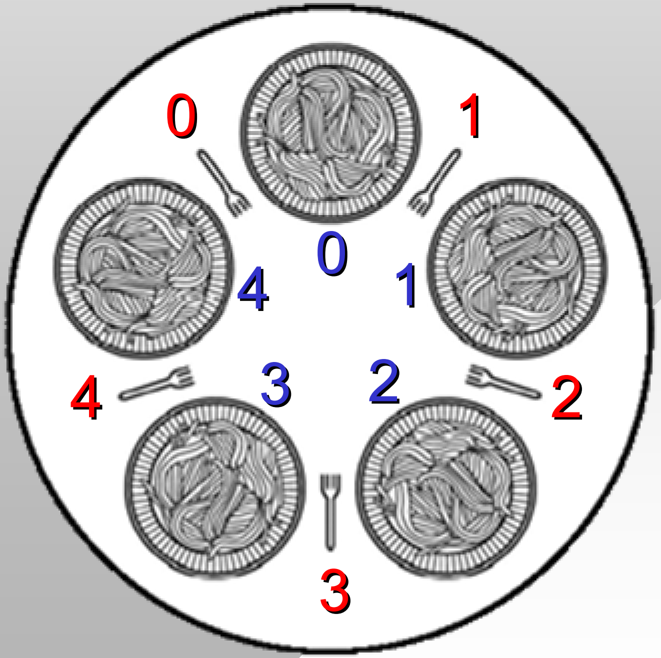


Dining Philosophers

```
Philosopher (int i) {
    while (true) {
       Think ();
       GrabForks (i);
       Eat ();
       DropForks(i);
    }
}
```

- Operation of a philosopher (each is a separate thread $0 \leq i \leq N\text{-}1$)
- Forks are labeled 0 to N-1 as well

```
Mutex mutexFork[N];  // one for each fork

DropForks (int i) {
    mutexFork[i].Unlock();
    mutexFork[(i+1)%N].Unlock();
}
```



- Basic approach DPH v1.0:

```
Mutex mutexFork[N];  // one for each fork

GrabForks (int i) {
    mutexFork[i].Lock();  // right fork
    mutexFork[(i+1)%N].Lock(); // left fork
}
```

- When all are hungry, deadlock is possible

Chapter 6: Roadmap

- 6.1 Principles
- 6.6 Dining philosophers
- 6.2 Prevention
- 6.3 Avoidance
- 6.4 Detection
- 6.5 Integrated strategies
- 6.7 Unix
- 6.8 Linux
- 6.9 Solaris
- 6.10 Windows

Prevention

- In deadlock prevention, the algorithm is modified by programmer to make one of the 4 conditions leading to deadlock impossible
- Condition #1: mutual exclusion
  - Typically cannot be safely eliminated (e.g., cars cannot drive on top of each other thru intersection)
- Condition #2: hold and wait
  - Can be overcome with WaitAll, DPH v1.1

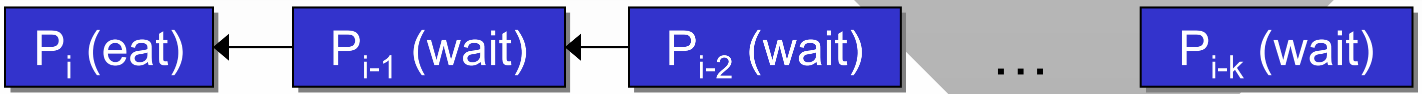
WaitAll is either super slow (Windows) or absent (Unix)

```
Mutex mutexFork[N];  // one mutex for each fork

GrabForks (int i) {
    WaitAll (mutexFork[i], mutexFork[(i+1)%N]); // both forks
}
```

  - Besides speed, main drawback is that all needed mutexes must be known ahead of time and acquired in bulk

Prevention

- Condition #4: circular wait
  - Design algorithm such that a circular deadlock cannot occur
- Notice that presence of 3 or fewer cars (4 or fewer philosophers) cannot cause a cyclic wait graph
  - Use a semaphore to control how many at the table
- Q: how many can eat concurrently?
  - If only $\lfloor N/2 \rfloor$, why allow all N to grab forks?
- How many should be allowed to use forks?
  - To achieve max concurrency, N-1, but …
  - Algorithm is prone to persistent chains of waits:



Prevention

- Suppose T > 0 is the eat+think delay in seconds
  - Max theoretical rate of algorithm is N / 2 * 1 / T
  - If T = 0, then mutex locking/unlocking is the bottleneck

```
CRITICAL_SECTION cs[N];  // one mutex for each fork
HANDLE sema = CreateSemaphore (..., N-1, N-1, ...);

GrabForks (int i) {
    WaitForSingleObject (sema, INFINITE);
    EnterCriticalSection (&cs[i]);
    EnterCriticalSection (&cs[(i+1)%N]);
}
```

DPH v1.2

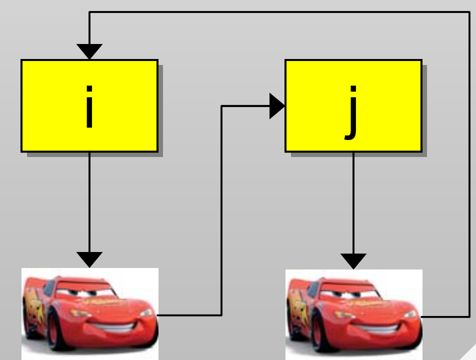
T=0
450K/sec N = 5

T=100ms
10/sec N = 500

- Elegant semaphore solution, but slow
  - T=0: kernel-mode semaphore kills performance
  - T=100ms: prone to sequential chains of waits, in which case performance may deteriorate to 1/T = 10 per second
  - Improves if think delays are random (1700/sec), or max semaphore = N/2 (1900/sec)

Prevention

- Another way to prevent circular wait is to request resources in the same order from all threads
- If thread holds resource i and wants j, then $j > i$
  - If all other threads comply with this rule, a loop back to i in the resource graph is impossible
- DPH v1.3




```
CRITICAL_SECTION cs[N];   // one mutex for each fork

GrabForks (int i) {
    if (i != N-1) {  // not the last guy
        EnterCriticalSection (&cs[i]);
        EnterCriticalSection (&cs[i+1]);
    }
    else {
        // special case, a leftie
        EnterCriticalSection (&cs[0]);
        EnterCriticalSection (&cs[N-1]);
    }
}
```

T=0
2M/sec N = 5

T=100ms
254/sec N = 500

Prevention

- Condition #3: no preemption of held mutexes
  - Let waiter (OS) forcefully remove forks and reassign them
- More realistic version:
  - If unable to make progress, threads can voluntarily release held mutexes, randomly sleep, and start again
- Similar to PC 3.4, which was the fastest in prior tests

```
CRITICAL_SECTION cs[N];  // one mutex for each fork

GrabForks (int i) {
    EnterCriticalSection (&cs[i]);
    do {
       if (TryEnterCriticalSection ( &cs[ (i+1)%N ] ) != 0)
          break;
       // unable to acquire
       LeaveCriticalSection (&cs[i]);
       Sleep (rand()*DELAY);
       EnterCriticalSection (&cs[i]);
    } while (true);
}
```

DPH v1.4

T=0
1.9M/sec
N = 5

T=100ms
2400/sec
N = 500

Debug Session

- Q: Find problems with this program:

```
class X {
    char  *buf;
    int   size;
    X() { buf = new char [100]; size = 100; }
    ~X() { delete buf; }
};
```

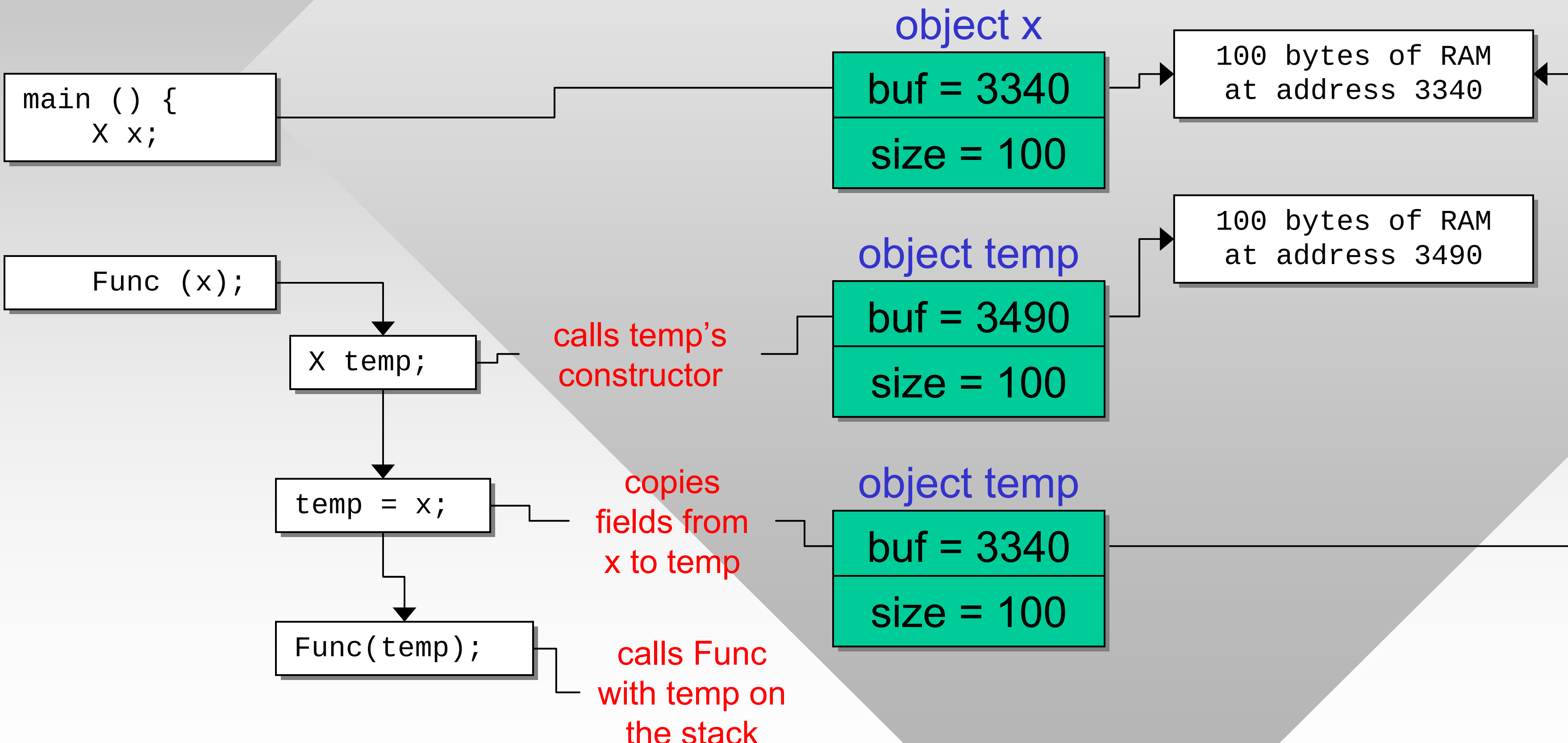
```
main () {
    X x;

    Func (x);
}
```

```
void Func (X x)
{
    return;
}
```

- A: Deletion of invalid block and a memory leak
  - Thrown when main() exits
- Reason is that a copy of x is created to pass to Func
  - This copy gets deleted when Func() returns
  - Which in turn triggers destructor ~X() and deletion of buf
- Finally, when main quits, it calls ~X() again
  - Which attempts to delete buf a second time

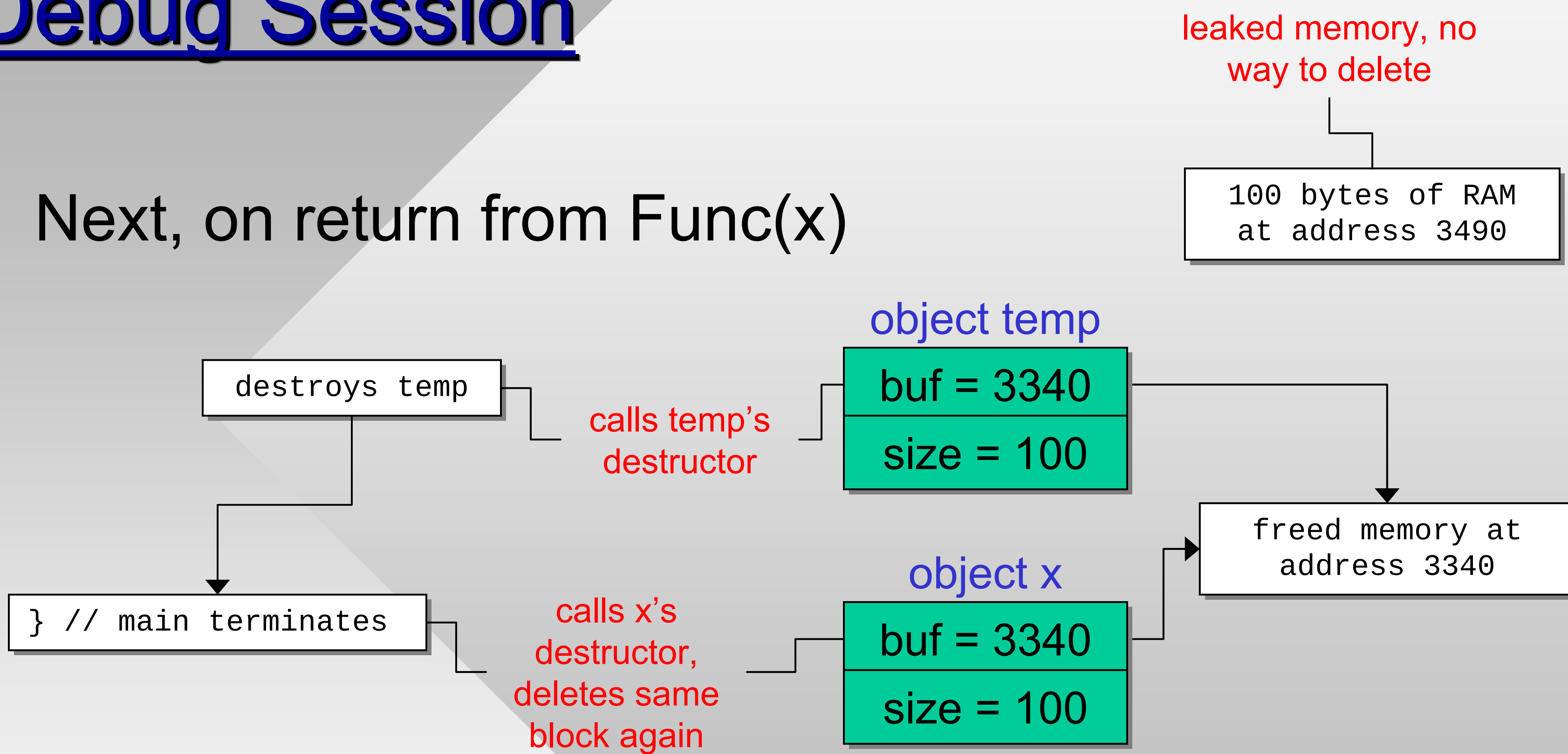
Debug Session

- A walk-thru of what happens:



Debug Session

- Next, on return from Func(x)



- Lesson: pass pointers to classes whenever feasible
  - Saves a lot of headache with copying stuff over, also faster
- If a call-by-value is needed, use copy constructors
  - See http://en.wikipedia.org/wiki/Copy_constructor