Announcements

• Midterm
  ♦ Can bring one A4 paper cheatsheet, double sided

• Next Wed no lecture -> turned into my office hour (CSE 3123), plus the Monday office hour
  ♦ Can’t make either? Send me email to request another time
Concurrency Bugs

• Concurrency bugs
  ♦ Bugs happened with parallel (concurrent) threads
  ♦ Very hard to debug because of the non-deterministic nature of parallel programs

• Blocking bugs
  ♦ Concurrency bugs that cause one or more thread to stuck (cannot make progress)
  ♦ E.g., deadlock

• Non-blocking bugs
  ♦ Concurrency bugs that do not block any thread’s execution but results in undesired behavior
  ♦ E.g., data race
Deadlock

- Synchronization is a live gun – we can easily shoot ourselves in the foot
  - Incorrect use of synchronization can block all processes
  - You have likely been intuitively avoiding this situation already
- More generally, threads that try to acquire multiple resources generate dependencies on those resources
  - Locks, semaphores, monitors, etc., just represent the resources that they protect
- If one thread tries to acquire a resource that a second thread holds, and vice-versa, they can never make progress
- We call this situation deadlock, and we’ll look at:
  - Definition and conditions necessary for deadlock
  - Representation of deadlock conditions
  - Approaches to dealing with deadlock
Deadlock Example
Deadlock Example: Dining Philosophers’ Problem

- Dijkstra 1971
- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time

Subject to deadlock if they all pick up their “right” fork simultaneously!

More in backup slides
Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)

2. Readers-writers problem

3. Dining philosophers problem
Deadlock Definition

- Deadlock is a problem that can arise:
  - When threads compete for access to limited resources
  - When threads are incorrectly synchronized

- Definition:
  - Deadlock exists among a set of threads if every thread is waiting for an event that can be caused only by another thread in the set.

```
Thread 1
lockA->Acquire();
...
lockB->Acquire();

Thread 2
lockB->Acquire();
...
lockA->Acquire();
```
Deadlock with Join

Thread A

...  
B.join();  
...  

Thread B

...  
A.join();  
...
Resource Allocation Graph

- Deadlock can be described using a resource allocation graph (RAG)
- The RAG consists of a set of vertices $T=\{T_1, T_2, \ldots, T_n\}$ of threads and $R=\{R_1, R_2, \ldots, R_m\}$ of resources
  - A directed edge from a thread to a resource, $T_i \rightarrow R_j$, means that $T_i$ has requested $R_j$
  - A directed edge from a resource to a thread, $R_i \rightarrow T_i$, means that $R_j$ has been allocated by $T_i$
  - Each resource has a fixed number of units
- If the graph has no cycles, deadlock cannot exist
- If the graph has a cycle, deadlock may exist
Resource-Allocation Graph (Cont.)

- Thread

- Resource type with 4 instances

- $T_i$ requests instance of $R_j$

- $T_i$ is holding an instance of $R_j$
Resource Allocation Graph – is there a deadlock?
Resource Allocation Graph with a cycle – is there a deadlock?
Resource Allocation Graph with a cycle – is there a deadlock?
Starvation vs Deadlock

- Starvation vs. Deadlock
  - Starvation: thread waits indefinitely
    - Example, low-priority thread waiting for resources constantly in use by high-priority threads
  - Deadlock: circular waiting for resources
    - Thread A owns Res 1 and is waiting for Res 2
      Thread B owns Res 2 and is waiting for Res 1

- Deadlock ⇒ Starvation but not vice versa
  - Starvation can end (but doesn’t have to)
  - Deadlock can’t end without external intervention
Conditions for Deadlock

- Deadlock can exist if and only if the following four conditions hold simultaneously:
  1. **Mutual exclusion** – At least one resource must be held in a non-sharable mode
  2. **Hold and wait** – There must be one thread holding one resource and waiting for another resource
  3. **No preemption** – Resources cannot be preempted (critical sections cannot be aborted externally)
  4. **Circular wait** – There must exist a set of threads \([T_1, T_2, T_3, \ldots, T_n]\) such that \(T_1\) is waiting for \(T_2\), \(T_2\) for \(T_3\), etc.

Eliminating *any* condition eliminates deadlock!
Four Possible Strategies to Deal With Deadlocks

1. Ignore the problem
   ♦ It is user’s fault
   ♦ used by most operating systems, including Linux

2. Detection and recovery (by OS)
   ♦ Fix the problem after occurring

3. Dynamic avoidance (by OS, programmer help)
   ♦ Careful allocation

4. Prevention (by programmer, practically)
   ♦ Negate one of the four conditions
2. Detection and Recovery

• Detection and recovery
  ♦ Allow deadlocks to happen but detect them and recover

• To do this, we need two algorithms
  ♦ One to determine whether a deadlock has occurred
  ♦ Another to recover from the deadlock
2. Deadlock Detection

• Detection
  ♦ Traverse the resource graph looking for cycles
  ♦ If a cycle is found, preempt resource (force a thread to release)

• Expensive
  ♦ Many threads and resources to traverse

• Invoke detection algorithm depending on
  ♦ How often or likely deadlock is
  ♦ How many threads are likely to be affected when it occurs
2. Deadlock Recovery

Once a deadlock is detected, we have two options...

1. Abort threads
   - Abort all deadlocked threads
     » Threads need to start over again
   - Abort one thread at a time until cycle is eliminated
     » System needs to rerun detection after each abort

2. Preempt resources (force their release)
   - Need to select thread and resource to preempt
   - Need to rollback thread to previous state
   - Need to prevent starvation
3. Deadlock Avoidance

- **Avoidance**
  - Provide information in advance about what resources will be needed by threads to guarantee that deadlock will not happen
  - System only grants resource requests if it knows that the thread can obtain all resources it needs in future requests
  - Avoids circularities (wait dependencies)

- **Tough**
  - Hard to determine all resources needed in advance
  - Good theoretical problem, not as practical to use
4. Deadlock Prevention

- Remove any of the four conditions of deadlocks
- Remove mutual exclusion
  - E.g., make resources sharable, not always possible
- Remove hold and wait
  - E.g., try to lock all needed resources at the beginning. If successful, use the resources & release them. Otherwise, release all resources and start over
- Preemption
  - E.g., if a request from a thread holding resources cannot be satisfied, preempt the thread and release all resources
- No circular wait
  - E.g., impose some order of requests for all resources
Deadlock Summary

• Deadlock occurs when threads are waiting on each other and cannot make progress
  ♦ Cycles in Resource Allocation Graph (RAG)

• Deadlock requires four conditions
  ♦ Mutual exclusion, hold and wait, no resource preemption, circular wait

• Four approaches to dealing with deadlock:
  ♦ Ignore it – Living life on the edge
  ♦ Avoidance – Carefully control allocation
  ♦ Detection and Recovery – Look for a cycle, preempt or abort
  ♦ Prevention – Make one of the four conditions impossible
Other Blocking Bugs: Forgetting to Release Lock

```c
void mptctl_simplified(unsigned long arg) {
    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
    MPT_ADAPTER *iocp = NULL;

    // first fetch
    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
        return -EFAULT;

    // dependency lookup
    if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
        return -EFAULT;

    // dependency usage
    mutex_lock(&iocp->ioctl_cmds.mutex);
    struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;

    // second fetch
    if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
        return -EFAULT;

    mptctl_do_fw_download(kfwdl.iocnum, ....);
    mutex_unlock(&iocp->ioctl_cmds.mutex);
}
```

Fig. 1: A dependency lookup double-fetch bug, adapted from __mptctl_ioctl in file drivers/message/fusion/mptctl.c

actual bug in a Linux driver!
Non-Blocking Bugs

• Atomicity-Violation Bugs
  ♦ The desired serializability among multiple memory accesses is violated (i.e. a code region is intended to be atomic, but the atomicity is not enforced during execution).
  ♦ Real example in MySQL

Thread 1::
if (thd->proc_info) {
  ...
  fputs(thd->proc_info, ...);
  ...  
}

Thread 2::
thd->proc_info = NULL;

Not Atomic!
Non-Blocking Bugs

• Order-Violation Bugs
  ♦ The desired order between two (groups of) memory accesses is flipped (i.e., A should always be executed before B, but the order is not enforced during execution)

Thread 1::
void init() {
  ...
  mThread =
  PR_CreateThread(mMain, ...);
  ...
}

Thread 2::
void mMain(...) {
  ...
  mState = mThread->State;
  ...
  }

Next time...

- Midterm review
Dining Philosophers’ Problem

- Dijkstra 1971
- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time
Dining philosophers problem

Abstraction of concurrency-control problems
The need to allocate several resources among several processes while being deadlock-free and starvation-free
Rules of the Game

- The philosophers are very logical
  - They want to settle on a shared policy that all can apply concurrently
  - They are hungry: the policy should let everyone eat (eventually)
  - They are utterly dedicated to the proposition of equality: the policy should be totally fair
Basic Operation of Each Philosopher

while (1) {
    think();
    getforks();
    eat();
    putforks();
}

Helper functions:
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; } // Assuming 5 philosophers
sem forks[5]; // semaphores for the 5 forks
What can go wrong?

• Primarily, we worry about:
  ♦ Starvation: A policy that can leave some philosopher hungry in some situation (even one where the others collaborate)
  ♦ Deadlock: A policy that leaves all the philosophers “stuck”, so that nobody can do anything at all
  ♦ Livelock: A policy that makes them all do something endlessly without ever eating!
A flawed conceptual solution

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

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

Oops! Subject to deadlock if they all pick up their “right” fork simultaneously!
Dijkstra’s Solution

```c
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)]);
    }
}
```
Other Dining Philosophers Solutions

- Allow only 4 philosophers to sit simultaneously
- Asymmetric solution
  - Odd philosopher picks left fork followed by right
  - Even philosopher does vice versa
- Pass a token
- Allow philosopher to pick fork only if both available
Solutions are less interesting than the problem itself!

- In fact the problem statement is why people like to talk about this problem!
- Rather than solving Dining Philosophers, we should use it to understand properties of solutions that work and of solutions that can fail!