CSE 120 Principles of Operating Systems

Fall 2021

Lecture 7: Conditional Variables, Concurrency Bugs

Yiying Zhang

Announcements

- Midterm coming up (10/26 2-3:20pm)
  - Remote exam (mostly to be conducted on Canvas)
  - Time is synced, no make-up exams
  - Will cover everything until today (not including CPU scheduling)
  - We will include problems about project in the exam
    - » Everyone should work on the project!
  - Open book (open everything, you can even Google if you have time)
  - Practice midterm exam to be released soon (before next Tue)
  - Because we cannot cover a lot in midterm, we are changing the weight of midterm and final
    - » Midterm 25% and final 35% (if do two projects)
    - » Midterm 20% and final 30% (if do three projects)

Announcements

- Important notice regarding project GitHub repo set up
  - Read your email (sent out ~30min ago)
- Project 0 and homework 1 graded (grades on Canvas)
- Last lecture's slides updated to consistently use thread instead of process
- Work on your project 1!
- Start preparing for the midterm

[lec5] Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a *guard* on the lock itself

```
void acquire (lock) {
    disable interrupts;
    while (test-and-set(lock→guard))  ;
    if (lock→held == 0) {
        lock→held = 1;
        lock→guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock→Q;
    lock→guard = 0;
    go to sleep;
    enable interrupts;
}
```

```
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}
```

What should the woken up thread do next?

```
void release (lock) {
    disable interrupts;
    while (test-and-set(lock→guard))  ;
    lock→held = 0;
    if (lock→Q is not empty)
        move a waiting thread to the ready queue;
    lock→guard = 0;
    enable interrupts;
}
```

[lec5] Implementing Locks with a queue

- If cannot hold lock, give up CPU (move to block queue)
- Use a *guard* on the lock itself

```
void acquire (lock) {
    disable interrupts;
    while (test-and-set(lock→guard))  ;
    if (lock→held == 0) {
        lock→held = 1;
        lock→guard = 0;
        enable interrupts;
        return;
    }
    put current thread on lock→Q;
    lock→guard = 0;
    go to sleep;
    enable interrupts;
}
```

A possible race condition?

```
struct lock {
    int held = 0;
    int guard = 0;
    queue Q;
}
```

What should the woken up thread do next?

```
void release (lock) {
    disable interrupts;
    while (test-and-set(lock→guard))  ;
    if (lock→Q is empty)
        lock→held = 0;
    if (lock→Q is not empty)
        move a waiting thread to the ready queue;
    lock→guard = 0;
    enable interrupts;
}
```

[lec6] Semaphore

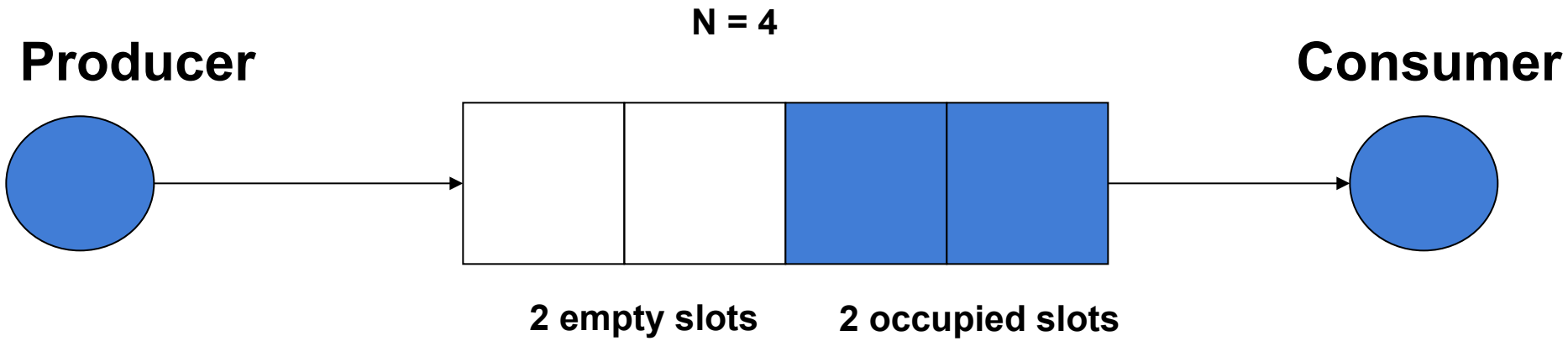
- A (non-negative) integer value and two primitive operations
  - wait(semaphore): an atomic operation that waits for semaphore to become greater than 0, then decrements it by 1
  - signal(semaphore): an atomic operation that increments semaphore by 1

[lec6] Two usages of semaphores

- For mutual exclusion:
  - to ensure that only one process is accessing shared info at a time.
  - Semaphores or binary semaphores?
- For condition synchronization:
  - to permit processes to wait for certain things to happen
  - Semaphores or binary semaphores?

[lec6] Producer & Consumer Problem

- Producer: creates copies of a resource
- Consumer: uses up (destroys) copies of a resource.
- Buffers: fixed size, used to hold resource produced by producer before consumed by consumer



[lec6] Readers-Writers problem

- A data object is shared among multiple processes
- Allow concurrent reads (but no writes)
- Only allow exclusive writes (no other writes or reads)

[lec6] Use TAS to implement semaphores on multiprocessor

```
void  wait(semaphore s)
{
  disable interrupts;
 while (1 == tas(&lock,1));
  if (s->count > 0) {
      s->count --;
      lock = 0;
      enable interrupts;
      return;
  }
  add(s->q, current_threads);
  lock=0;
  sleep(); /* re-dispatch */
  enable interrupts;
}
```

```
void  signal(semaphore s)
{
   disable interrupts;
   while (1 == tas(&lock,1));
   s->count ++;
   if (!isEmpty(s->q)) {
      thread = removeFirst(s->q);
      wakeup(thread);
      /* put thread on Ready Q */
    }
   lock = 0;
    enable interrupts;
}
```

Producer & Consumer – semaphore, counting is tricky

Producer

```
while (1) {

  produce an item;

  wait(EMPTY);

  acq(lock);
  insert(item to pool);
  rel(lock);

  signal(FILLED)
}
```

Consumer

```
While (1) {

  wait(FILLED);

  acq(lock);
  remove(item from pool);
  rel(lock)

  sginal(EMPTY);

  consume the item;
}
```

Init: FILLED = 0; EMPTY = N;

Producer & Consumer -- is there something simpler than semaphore?

Producer

```
while (1) {

    produce an item;


    if (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);


}
```

Consumer

```
While (1) {


 if (pool is Empty {

     wait(NotEMPTY);

  }
 record if pool was full;
 remove(item);

 if (pool was Full)
     signal(NotFULL);


 consume the item;
}
```

Producer & Consumer -- is there something simpler than semaphore?

Producer

```
while (1) {

    produce an item;

    acquire(mutex);
    if (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}
```

Consumer

```
While (1) {

 acquire(mutex);
 if (pool is Empty {

     wait(NotEMPTY);

  }
 record if pool was full;
 remove(item);

 if (pool was Full)
     signal(NotFULL);
 release(mutex);

 consume the item;
}
```

Producer & Consumer -- is there something simpler than semaphore?

Producer

```
while (1) {

    produce an item;

    acquire(mutex);
    if (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}
```

Consumer

```
While (1) {

 acquire(mutex);
 if (pool is Empty) {

      wait(NotEMPTY);

 }
 record if pool was full;
 remove(item);

 if (pool was Full)
     signal(NotFULL);
release(mutex);

consume the item;
}
```

Put me To sleep

If anyone is sleeping, wake it up (no counting)

Producer & Consumer -- is there something simpler than semaphore?

Producer

```
while (1) {

    produce an item;

    acquire(mutex);
    if (pool is Full) {
        release(mutex);
        wait(NotFULL);
        acquire(mutex);
    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}
```

Put me To sleep

If anyone is sleeping, wake it up (no counting)

Consumer

```
While (1) {

    acquire(mutex);
    if (pool is Empty) {
        release(mutex);
        wait(NotEMPTY);
        acquire(mutex);
    }
    record if pool was full;
    remove(item);

    if (pool was Full)
        signal(NotFULL);
    release(mutex);

    consume the item;
}
```

Producer & Consumer -- is there something simpler than semaphore?

Producer

```
while (1) {

    produce an item;

    acquire(mutex);
    if (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}
```

The simplification implies NotFull is tied to mutex

Consumer

```
While (1) {

 acquire(mutex);
 if (pool is Empty) {

     wait(NotEMPTY);

  }
 record if pool was full;
 remove(item);

 if (pool was Full)
     signal(NotFULL);
 release(mutex);

 consume the item;
}
```

Mutual Exclusion provided by OS or language/compiler

- Semaphore
  - Powerful, but kind of low level
  - can we have a high level abstraction?
- Locks and condition variables
  - Lock alone is not flexible enough
  - Need some mechanism to check conditions
- Monitor

Conditional Variables

- An explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition)
  - Also called wait (Java, C++), sleep (Nachos, C#)
- Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition)
  - Wake up one: wake (Nachos, C#), notify (Java), notify_one (C++)
  - Wake up all: wakeAll (Nachos, C#), notifyAll (Java), notify_all (C++)

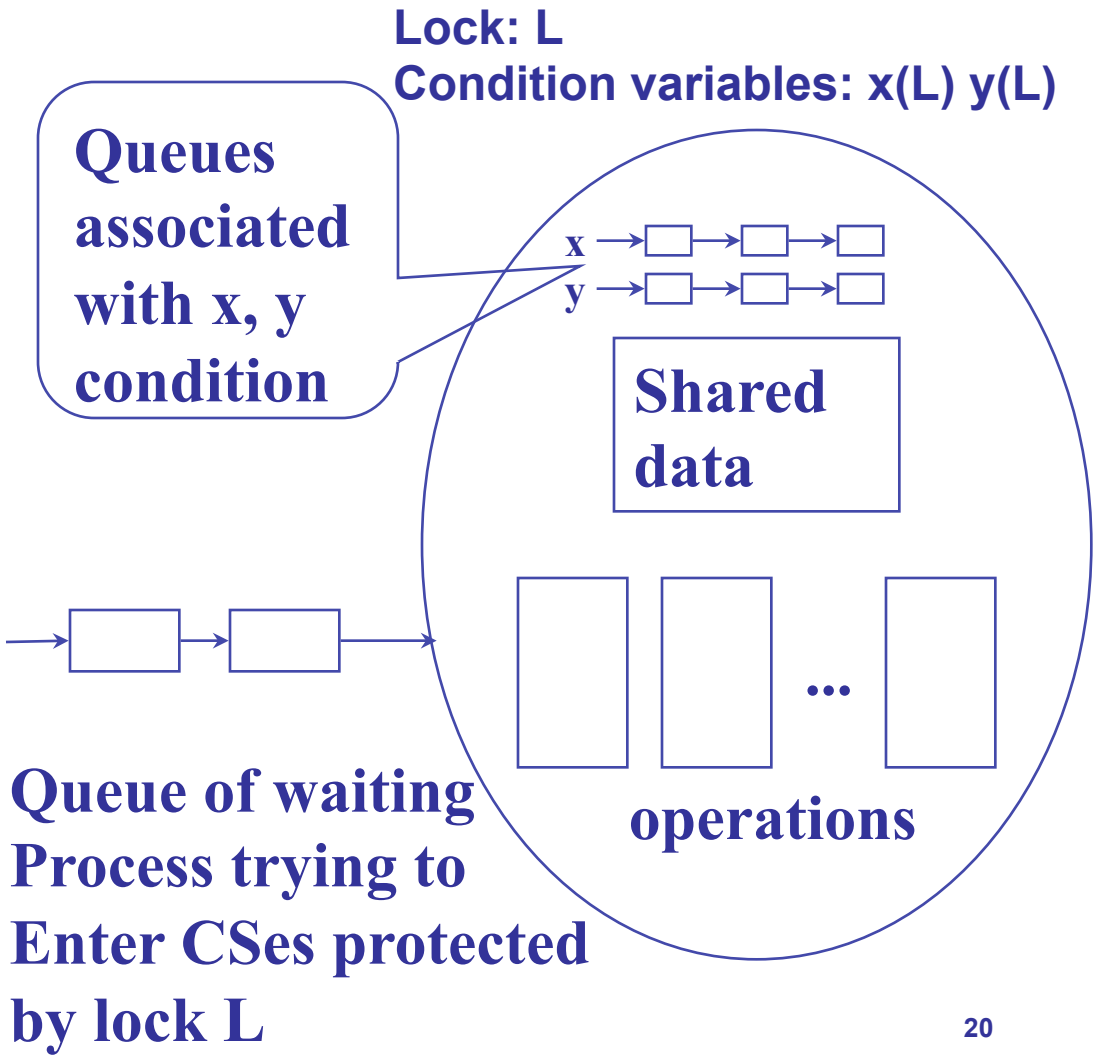
Conditional Variables

- Used in conjunction with locks
- Used inside critical section to wait for certain conditions
- Contrast with Semaphore:
  - Has no counting bundled
  - More intuitive to many people

- Usage
  - On creation, specify which mutex it is associated with

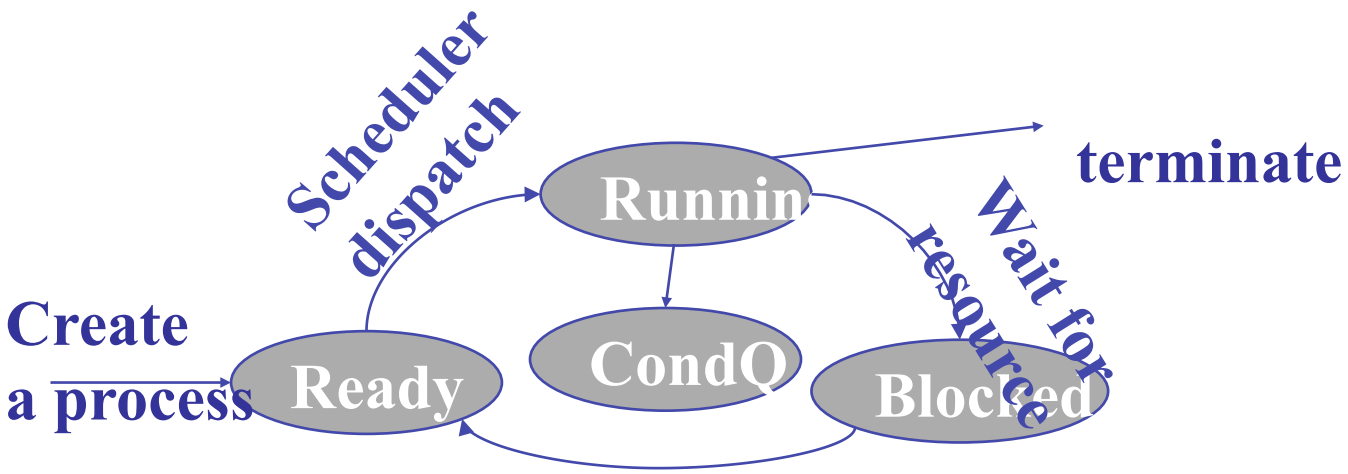
Conditional Variables

- Wait (condition)
  - Block on “condition”
- Signal (condition)
  - Wakeup one or more threads blocked on “condition”
- Conditions are like semaphores but:
  - signal is no-op if none blocked
  - There is no counting!



"Wow, I like condition variables"

- One problem – what happens on wakeup?
  - Only one thing can be inside critical section
  - But wakeup implies both signaler and waiter may be in critical section, who should go on?



Signal Semantics

- signal() places a waiter on the ready queue, but signaler continues inside lock
  - Known as “Mesa” style
  - Easy to implement
  - Another early-time semantics is Hoare style (signaler gives up lock, waiter runs immediately)

- What can happen when the awaken thread gets a chance to run?
  - E.g. pool is full, producer 1 waits; consumer signals it; p1 in ready queue; consumer release(lock); p2 comes along…

Producer & Consumer -- use condition variables – problem?

Producer

```
while (1) {

 produce an item;

    acquire(mutex);
    if  (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
         signal(NotEMPTY);
    release(mutex);
}
```

Consumer

```
While (1) {
 acquire(mutex);
if  (pool is Empty) {

     wait(NotEMPTY);

 }
record if pool was full;
remove(item);

if (pool was Full)
    signal(NotFULL);
some other work;
release(mutex);

consume the item;
}
```

Signal Semantics

- What can happen when the awaken thread gets a chance to run?
  - E.g. pool is full, producer 1 waits; consumer signals it; p1 in ready queue; consumer release(lock); p2 comes along…
- Condition not necessarily true when waiter runs again
  - Returning from wait() is only a hint that something changed
  - Must recheck conditional case

Producer & Consumer – use condition variables – how to fix?

Producer

```
while (1) {

 produce an item;

    acquire(mutex);
    while (pool is Full) {

        wait(NotFULL);

    }
    record if pool was empty;
    insert(item);

    if (pool was empty)
        signal(NotEMPTY);
    release(mutex);
}
```

Consumer

Is this busy waiting?

```
While (1) {

acquire(mutex);
 while (pool is Empty) {

     wait(NotEMPTY);

 }
 record if pool was full;
 remove(item);

 if (pool was Full)
     signal(NotFULL);
 release(mutex);

consume the item;
}
```

Be Careful About Pitfalls: CVs Cannot Be "Tested"

```
acquire(lock);
…
while (CV != true) {
   wait(CV);
}
…
release(lock);
```

```
acquire(lock);
…
while (flag != true) {
   wait(CV);
}
…
release(lock);
```

- Do not use a CV as a predicate
- Need to use a separate flag

Be Careful About Pitfalls: CVs Require Holding Lock

```
acquire(lock);
…
release(lock);
wait(CV);
acquire(lock);
…
release(lock);
```

```
acquire(lock);

…
wait(CV);
…

acquire(lock);
```

- Do not release the lock before using the CV
  - Using a CV requires a thread to hold the lock
- Purpose of a CV is to enable threads to block while in a critical section

Be Careful About Pitfalls: Need Lock When Testing Flag

```
…
if (check-condition) {
  acquire(lock);
  wait(CV);
  release(lock);
}
…
```

```
acquire(lock);
…
if (check-condition) {
   wait(CV);
}
…
release(lock);
```

- Testing a condition needs to be done while holding the lock
- It is a shared variable that can lead to race conditions

Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways
- If curious, read more in backup slides

Synchronization Primitives Summary

- Lock
  - Only achieves mutual exclusion
- Semaphores
  - Has built-in counters, and thus can express more semantics
  - Can be inconvenient to use
- Condition variables
  - Used by threads as a synchronization point to wait for events
  - Used with locks or inside monitors
- Monitors
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
  - Relies upon high-level language support

Concurrency Bugs

Concurrency Bugs

- Concurrency bugs
  - Bugs happened with parallel (concurrent) threads
  - Can happen with both shared memory and message passing
  - 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

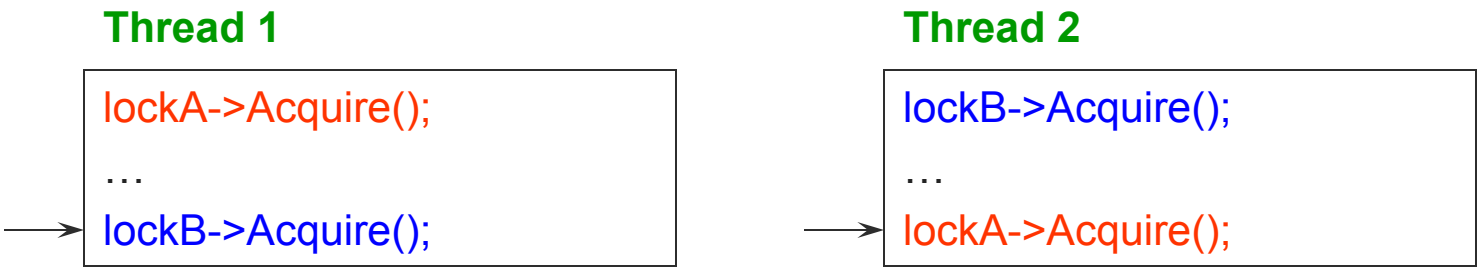
- Philosophers eat/think
- Eating needs two forks
- Pick one fork at a time

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

More in backup slides

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.



Deadlock with Join

Thread A

```
…
B.join();
…
```

Thread B

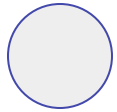
```
…
A.join();
…
```

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, $T_i \rightarrow R_i$, 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

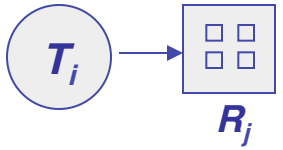
Resource-Allocation Graph (Cont.)

- Thread

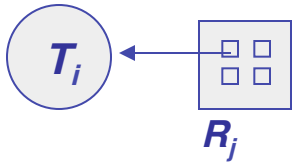


- Resource type with 4 instances

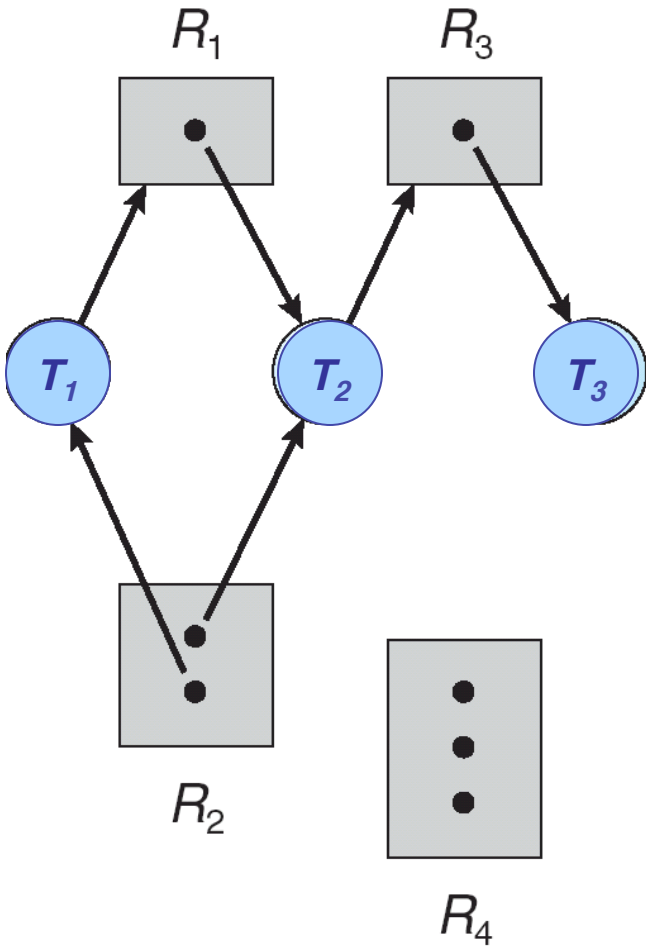
- T_i requests instance of R_j



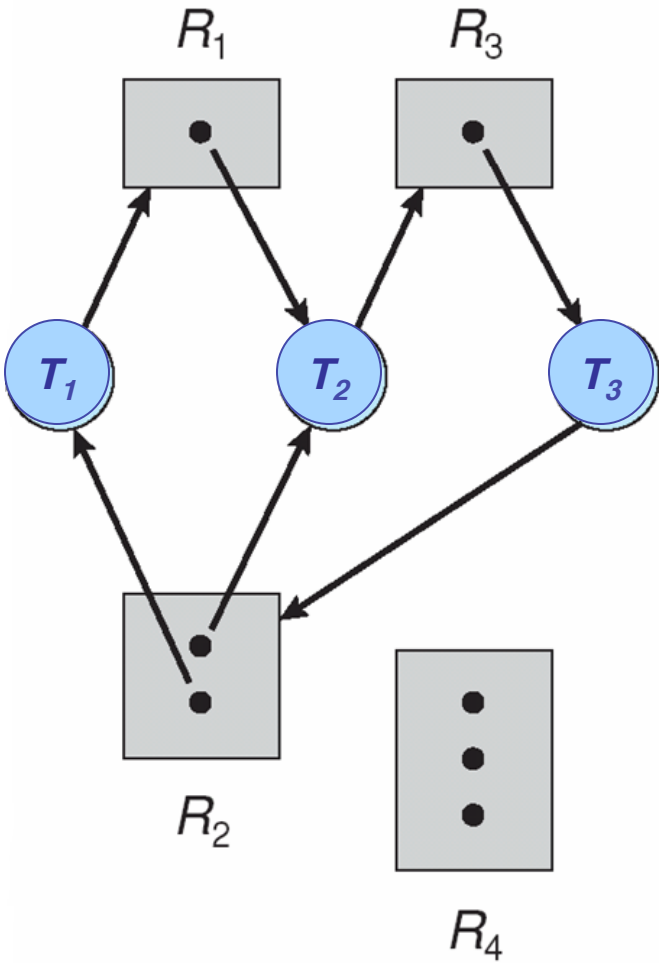
- T_i is holding an instance of R_j



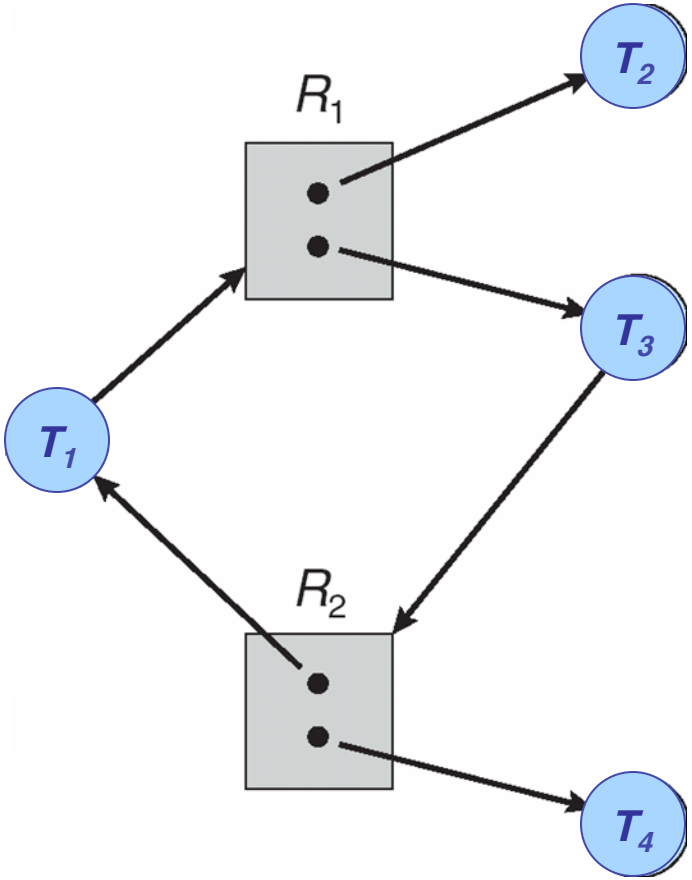
Resource Allocation Graph – is there a deadlock?



Resource Allocation Graph with a cycle – is there a deadlock?

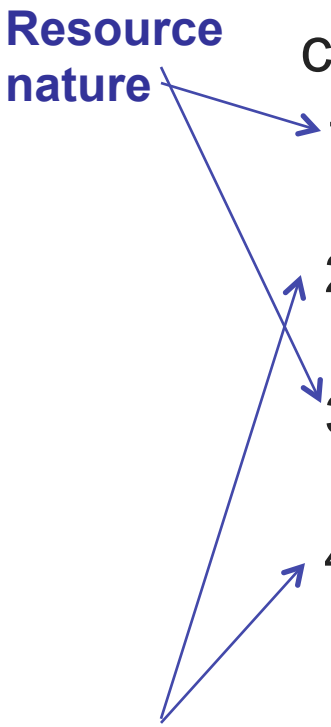


Resource Allocation Graph with a cycle – is there a deadlock?



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, …,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

Critical Section

```
1  void mptctl_simplified(unsigned long arg) {
2    mpt_ioctl_header khdr, __user *uhdr = (void __user *) arg;
3    MPT_ADAPTER *iocp = NULL;
4
5    // first fetch
6    if (copy_from_user(&khdr, uhdr, sizeof(khdr)))
7      return -EFAULT;
8
9    // dependency lookup
10   if (mpt_verify_adapter(khdr.iocnum, &iocp) < 0 || iocp == NULL)
11     return -EFAULT;
12
13   // dependency usage
14   mutex_lock(&iocp->ioctl_cmds.mutex);
15   struct mpt_fw_xfer kfwdl, __user *ufwdl = (void __user *) arg;
16
17   // second fetch
18   if (copy_from_user(&kfwdl, ufwdl, sizeof(struct mpt_fw_xfer)))
19     return -EFAULT;
20
21
22   mptctl_do_fw_download(kfwdl.iocnum, ......);
23   mutex_unlock(&iocp->ioctl_cmds.mutex);
24 }
```

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!

Other Blocking Bugs: Message-Passing Related

```
1    func finishReq(timeout time.Duration) r ob {
2  -   ch := make(chan ob)
3  +   ch := make(chan ob, 1)
4      go func() {
5        result := fn()
6        ch <- result // block
7      }
8      select {
9        case result = <- ch:
10         return result
11       case <- time.After(timeout):
12         return nil
13     }
14   }
```

actual bug in Kubernetes!

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, ...);
        ...
}
```

Not Atomic!

```
Thread 2::
thd->proc_info = NULL;
```

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…

- Read Chapters 7, 8, 32

Backup Slides

Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways

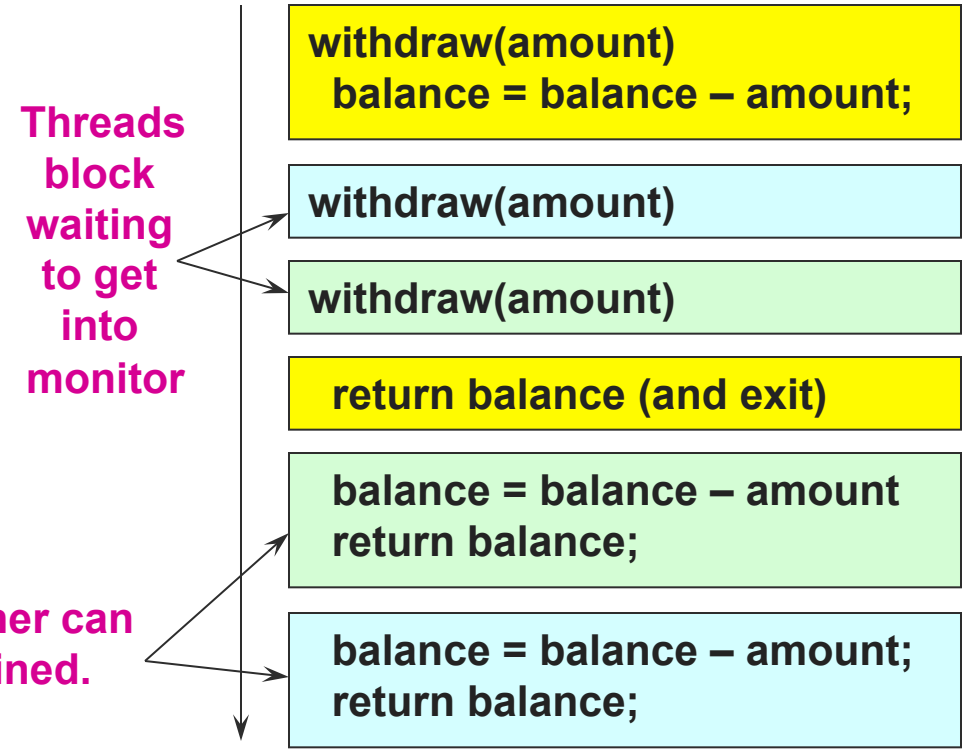
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    - » So the monitor has to have a wait queue…
  - Can have a condition variable inside a monitor

Account Example

```
Monitor account {
  double balance;

  double withdraw(amount) {
    balance = balance – amount;
    return balance;
  }
}
```



- Hey, that was easy!
- But what if a thread wants to wait inside the monitor?
  - » Such as “mutex(empty)” by reader in bounded buffer?

Monitors, Monitor Invariants and Condition Variables

- A monitor invariant is a safety property associated with the monitor, expressed over the monitored variables. It holds whenever a thread enters or exits the monitor.
- A condition variable is associated with a condition needed for a thread to make progress once it is in the monitor.

```
Monitor M {
  ... monitored variables
  Condition c;

  void enterMonitor (...) {
    if (extra property not true) wait(c);        waits outside of the monitor's mutex
    do what you have to do
    if (extra property true) signal(c);          brings in one thread waiting on condition
  }
```

Monitors and Java

- A lock and condition variable are in every Java object
  - Later added explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object's monitor
  - A thread enters an object's monitor by
    - » Executing a method declared synchronized
      - Can mix synchronized/unsynchronized methods in same class
    - » Executing the body of a synchronized statement
      - Supports finer-grained locking than an entire method
      - Identical to the Modula-2 "LOCK (m) DO" construct
  - The compiler generates code to acquire the object's lock at the start of the method and release it just before returning
    - » The lock itself is implicit, programmers do not worry about it

Monitors and Java

- Every object can be treated as a condition variable
  - Half of Object's methods are for synchronization!
- Take a look at the Java Object class:
  - Object.wait(*) is wait (Condition.sleep in Nachos)
  - Object.notify() is signal (Condition.wake)
  - Object.notifyAll() is broadcast (Condition.wakeAll)

Modern Languages

- Modern languages provide some form of locks and condition variables for synchronization and coordination
  - C, C++, C#, Java, Go, Rust, …
  - Most common form of synchronization you will encounter
- Typically locks are explicit
  - Programmers have to use acquire and release explicitly
    - » C++ and Rust have “release on return” language semantics
    - » A half-way monitor implementation…
  - Even Java eventually added separate classes (Lock, Condition) for flexibility

Classic Synchronization Problems

1. Producer-consumer problem (bounded buffer problem)

2. Readers-writers problem

3. Dining philosophers problem

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:

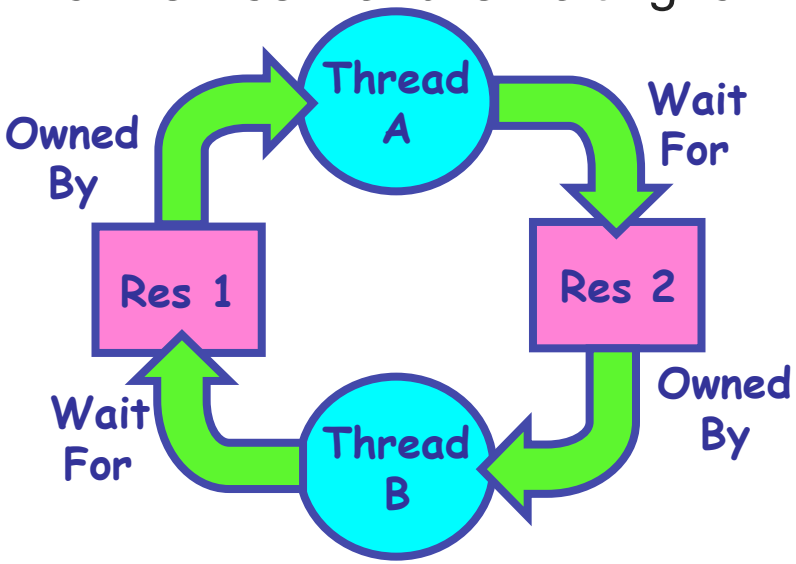
```
int left(int p) { return p; }
int right(int p) { return (p + 1) % 5; } // Assuming 5 philosophers
sem forks[5]; // semaphores for the 5 forks
```

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!

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

A flawed conceptual solution

```
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!

Dijstra's Solution

```
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!