CS-446/646

Synchronization Pitfalls & Exercises

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

Synchronization Errors

Deadlock

Situation where 2 or more processes are never able to proceed because each is waiting on the other to complete something (that can only be completed by the latter)

- Key concept: "*Circular Waiting*"

Race Condition

Timing dependent error involving *Shared State*

- *Data Race*: Concurrent accesses to a shared variable and at least one access is a *Write*
- *Atomicity* Bugs: Code does not enforce the *Atomicity* required for a group of *Memory* accesses
- *Order* Bugs: Code does not enforce the *Order* required for a group of *Memory* accesses

Concurrent coding difficult because:

- Too many schedules (exponential to program size), hard to reason about
- Correct *Concurrent* code does not compose → Can't divide-and-conquer
  - Synchronization crosses abstraction boundaries
  - Local correctness may not yield global correctness

Synchronization Errors

Example 1: Good + Bad → Bad

```
void deposit() { // properly synchronized
  lock();
  ++ balance;
  unlock();
}
```

```
void withdraw() { // no synchronization
  -- balance;
}
```

Remember: Atomic operations

➢ Result: *Race*

Synchronization Errors

Example 2: Good + Good → Bad

```
void deposit(account_t* acnt) {
  lock(acnt->guard);
  ++ acnt->balance;
  unlock(acnt->guard);
}

int balance(account_t* acnt) {
  int b;
  lock(acnt->guard);
  b = acnt->balance;
  unlock(acnt->guard);
  return b;
}
```

```
void withdraw(account_t* acnt) {
  lock(acnt->guard);
  -- acnt->balance;
  unlock(acnt->guard);
}

int sum(account_t* a1, account_t* a2) {
  return balance(a1) + balance(a2)
}

void transfer(account_t* a1, account_t* a2) {
  withdraw(a1);
  deposit(a2);
}
```

- Compose with single-account operations to perform operations on two accounts
  - Separate **deposit**, **withdraw**, **balance**, are *Synchronized*
- Result: *Race* in **sum** and **transfer** (catastrophic or not)

Synchronization Errors

Example 3: Good + Good → *Deadlock*

```
int sum(account *a1, account *a2) {
  int s;
  lock(a1->guard);
  lock(a2->guard);
  s = a1->balance;
  s += a2->balance;
  unlock(a2->guard);
  unlock(a1->guard);
  return s
}
```

Thread 1	Thread 2
`sum(A, B);`	`sum(B, A);`

```
lock(A->guard);
lock(B->guard);
lock(A->guard);   Block
lock(B->guard);   Block
```

- 2nd Attempt: Use *Locks* in **sum**
  - Single **sum** call, *Race-Free*
- Two concurrent *Thread* **sum** calls → *Deadlock*

Note: Can be prevented by a separate **sum** *Lock*, or by *Static Ordering of Resources* (more on that later)

Synchronization Errors

Example 4-a: Monitors also do not compose

```
monitor M1 {
  cond_t cv;
  foo() {
    …
    // releases monitor lock
    wait(cv);
  }

  bar() {
    …
    signal(cv);
  }
};
```

```
monitor M2 {
  f1() {
    …
    M1.foo();
  }

  f2() {
    …
    M1.bar();
  }
};
```

- Caution: Holding *Locks* (in this case *Monitor Lock*) across abstraction boundaries
  - **`foo`** and **`bar`** are **internally** using *Condition Variables* of another *Monitor*
  - The *Monitor M2*'s Procedure cannot know runtime semantics of *M1* to ensure it adheres by the fundamental *Monitor* operations

Synchronization Errors

Example 4-b: Deadlock crossing Abstraction Boundaries

Example when composing operations of *Mutex* and a *Condition Variable*:

Mutex Part / *Cond Var* Part

Thread 1

```
lock(a);
  lock(b);
  while(!ready)
    wait (cv, b);
  unlock(b);        } foo();
unlock(a);
```

Thread 2

```
lock(a);
  lock(b);
  ready = true;
  signal (cv);
  unlock(b);        } bar();
unlock(a);
```

Thread 1 reaches :
`wait (cv, b);` **b** Unlocked, but **a** Locked;

Thread 2 starts, but will never past its **`lock(a)`** to proceed to **`signal()`** *Thread 1*

- *Deadlock*

- The *Cond Var* Parts could be inside functions **`foo`** and **`bar`** that are unknown to us implementation-wise
- *Caution:* Dangerous to hold *Locks* (generally) across Abstraction Boundaries

Synchronization Errors

***Deadlock* Conditions** (all need to hold)

- *Mutual Exclusion*
  - At least one *Resource* must be held exclusively (in a non-sharable mode)
- *Hold and Wait*
  - There must be one process holding one *Resource* and waiting for another *Resource*
- *No Preemption*
  - *Resources* are *Non-Preemptable* (*Critical Sections* cannot be aborted externally)
    - vs *Preemptable* (can be taken away from a process without hurting its execution)
- *Circular Wait*
  - There must exist a set of processes [P_1, P_2,…,P_n] such that P_1 is waiting for P_2, P_2 for P_3, …, and P_n for P_1

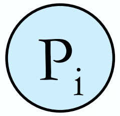
Two approaches to dealing with a *Deadlock*

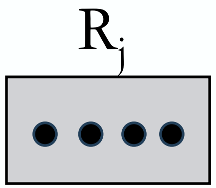
- Proactive: *Prevention*
- Reactive: *Detection & Correction*

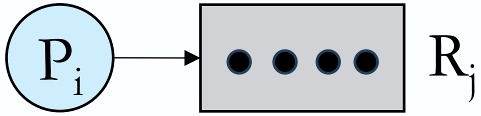
Synchronization Errors

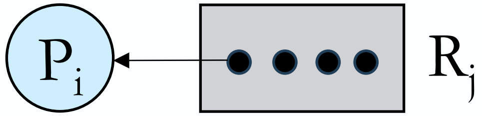
Deadlock Prevention **by Elimination of** ***Circular Waiting***

View system as a *Resource Allocation Graph*

- *Processes(/Threads)* and *Resources(/Locks)* are Nodes
- *Resource Assignments* are Edges: e.g. *Lock* → *Thread*
- *Resource Requests* are Edges: e.g. *Thread* → *Lock*
- *Process* 

- *Resource* with 4 instances 

- P_i requesting Rj 

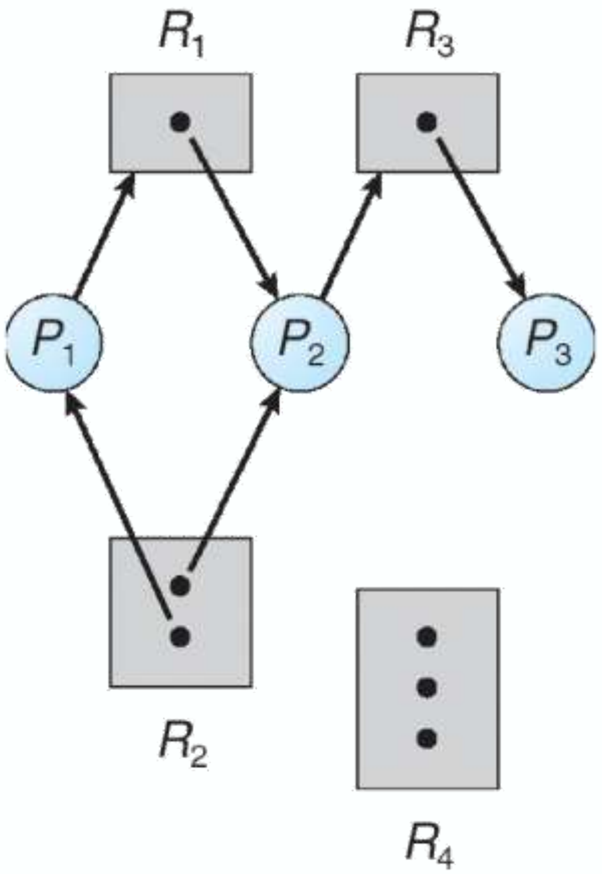
- P_i holding 1 instance of R_j 

Note: Edge removed on `unlock()`

Note: Request Edge Converted to *Assignment Edge* on return of `lock()`

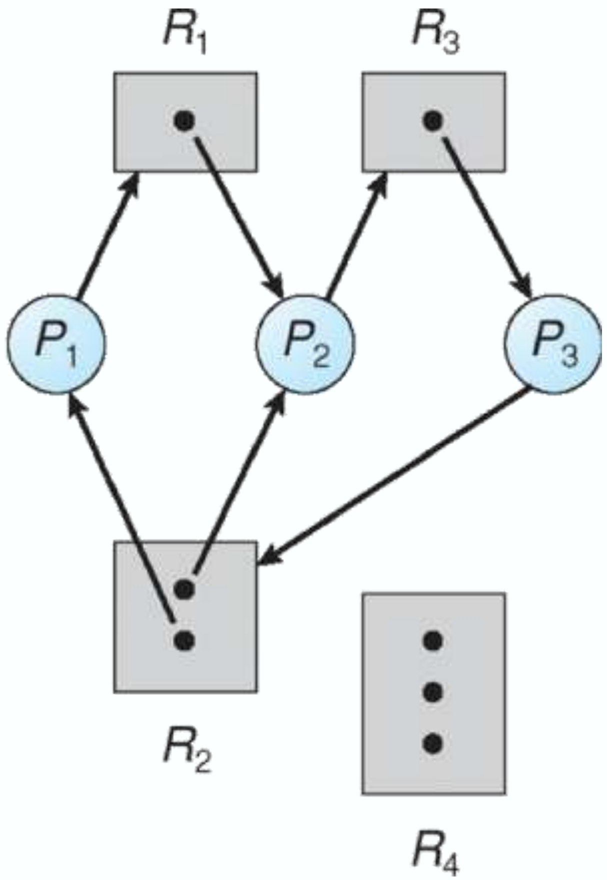
Deadlock Prevention **by Elimination of** ***Circular Waiting***

Example:
Resource Allocation Graph



Deadlock Prevention **by Elimination of** ***Circular Waiting***

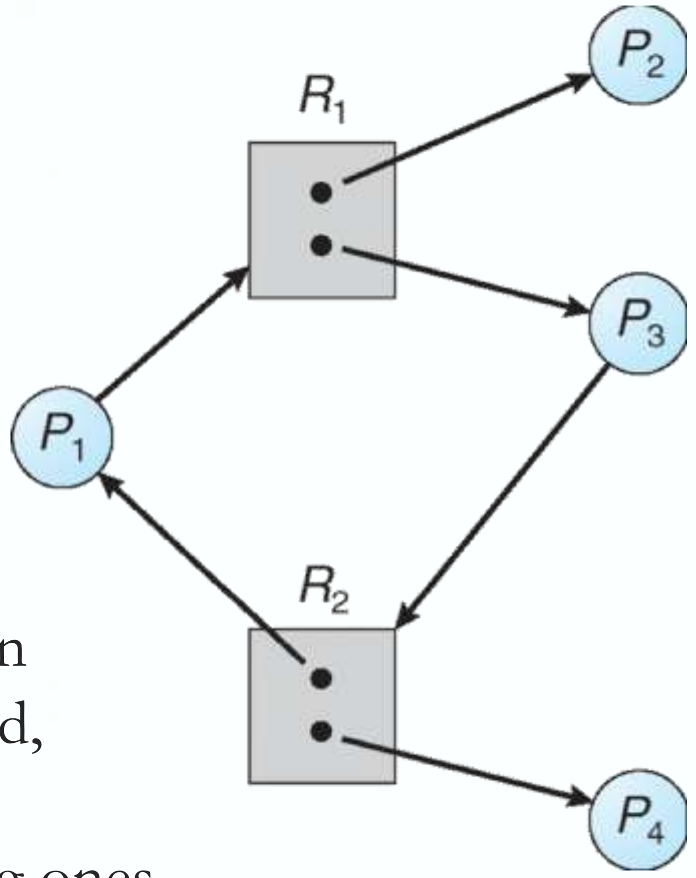
Example:
Resource Allocation Graph
with a **definite**
Deadlock



Note:
Resources that participate in *Circular Wait* are exhausted, and entirely allocated to the waiting processes.

Deadlock Prevention **by Elimination of** ***Circular Waiting***

Example:
Resource Allocation Graph
and a **possible**
Deadlock



Note:
Resources that participate in
Circular Wait are exhausted,
but allocated to different
processes than the waiting ones.

Deadlock Prevention **by Elimination of** ***Circular Waiting***

Resource Allocation Graph Cycles and *Deadlock*

- If Graph has no Cycles → no *Deadlock*
- If Graph contains a Cycle
  - **Definite** *Deadlock* if only a **single unit** per *Resource* – use *Waits-For Graph* (WFG)
    - WFG: Variant of *Resource Allocation Graph* with only *Processes* as Nodes
  - Otherwise, **possible** *Deadlock*

Prevent *Deadlock* with *Static Ordering of Resources*

- Statically number *Resources,* e.g. R_0 is *Mutex* **`m0`**, R_1 is Mutex **`m1`**, etc.
- Require (by-application-design) *Process* to request *Resources* in strict numerical order
  - To have *Deadlock*, a *Process* must be holding R_j and requesting R_i, where $i<j$

Note: E.g. to avoid multi-*Mutex Deadlocks*, we can make sure that there is a static global order for all *Mutexes*, and when *Locks* are taken they are always taken in that order.

Remember: *Example 3* where *Lock* objects **`a->guard`** & **`b->guard`** are flipped in the order they are **`lock()`**ed by the 2 different *Threads*.

Synchronization Errors

Dealing with a *Deadlock*

- *Ignore* it (until it goes away) – "Ostrich" approach
- *Deadlock Prevention* – Make it impossible for a *Deadlock* to happen
- *Deadlock Avoidance* – Control allocation of *Resources*
  - Provide information in advance about what *Resources* will be needed by *Processes* to guarantee that *Deadlock* will not happen
  - System only grants *Resource* Requests if it is guaranteed that the *Process* can obtain all *Resources* it is going to need in every future Requests
  - Effectively avoids Circular-Waits (Wait Dependencies), but it is impractical (and hard) to have to determine in advance all *Resources* that will be needed
- *Deadlock Detection* & *Recovery* – Look for a Cycle in dependencies; "break" the Cycle

Banker's Algorithm

Classic *Deadlock Avoidance* approach for *Resources* with **multiple units**

1. Assign a Credit Limit to each Customer (*Process*)

- For every *Process*, we must establish its required Credit Limit (max number of *Resources* expected to be Requested) in advance – impractical

2. Reject any Request that leads to an *Unsafe State*

- *Unsafe State*: One where a sudden Request by any Customer up to their full Max Credit Limit could lead to a *Deadlock*
- Use a recursive reduction procedure to discover *Unsafe States* and skip them (allow R_j allocation only for *Safe* P_i Requests, and iterate to find sequence of only *Safe* P_i Requests)

3. In practice: System must keep *Resource* usage well below capacity to maintain a surplus

- Should rarely have to be invoked due to low *Resource* utilization

Process	Allocated			Max			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3			
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2	3	3	2
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

Deadlock Detection & Recovery

Deadlock Detection

- *Detection*
  - Traverse the *Resource Allocation Graph* looking for Cycles
  - If a Cycle is found, *Preempt Resource* (force a *Process* that holds it to release it)
- Expensive
  - Many *Processes* and *Resources* to traverse
    - Cycle Detection algorithm (e.g. *Depth-First Search*) has to be ran and parse every Node of the Graph (can have multiple connected components)
- Algorithm invoked depending on
  - How frequent or likely *Deadlock* is
  - How many *Processes* are likely to be affected when it occurs

Deadlock Detection & Recovery

Deadlock Recovery

After a *Deadlock* has been detected, two main options:

- i) Abort *Processes*
  - Abort all *Deadlocked* Processes
    - Processes need to be started over
  - Abort one Process at a time until Cycle is eliminated
    - System needs to rerun *Deadlock Detection* after each abort
- ii) *Preempt Resources* (force their release)
  - Need to: a) Select Process and *Resource* to *Preempt;* b) Suspend selected Process until *Resource* becomes available again; c) Allocate released *Resource* to a Requesting Process
- Other methods:
  - *Priority Inversion* (more on that later in *Scheduling* Lecture); can lead to *Starvation*
  - *Rollback* to previous state (used in Database systems)

Synchronization Errors

Race Detection

Data Race Detection

Will only focus on *Data Race Detection*
- Techniques also exist to detect *Atomicity* and *Order Race* bugs

- Approach 1:
  - *Happens-Before*
- Approach 2:
  - *Lockset* (*Eraser* Algorithm)

Synchronization Errors

Race Detection

Happens-Before relationship & proper *Synchronization*

Definition: Event A "*Happens-Before*" Event B if:

- When both in the same *Thread*
  - B follows A
- When A in *Thread 1*, and B in *Thread 2*, exists a *Synchronization Event* C such that
  - A happens in *Thread 1*
  - C is after A in *Thread 1* and before B in *Thread 2*
  - B happens in *Thread 2*
- To detect *Data Race*, have to monitor **all** data accesses & *Synch* operations, watch for:
  - Access of shared location *v* happens in *Thread* ***T1***
  - Access of shared location *v* happens in *Thread* ***T2***
  - No *Synchronization* operation happens between the accesses
  - One of the accesses is a *Write*

Race Detection

Happens-Before for *Data Race Detection*

Problems:

- Expensive
  - Requires per-*Thread:*
    - List of all accesses to shared data
    - List of all *Synchronization* operations
- High False-Negative rate
  - *Happens-Before* looks out for *Data Races* that will take place during *Runtime*
    - i.e. moments when different *Threads* **actually** access shared data w/o *Synchronization*
  - Depends on *Scheduler*-controlled interleaving of events to elicit **actual** *Data Races*

Synchronization Errors

Race Detection

Eraser

Idea: Check *Invariants*

- Violations of *Invariants* → Likely *Data Races*

What is the tracked *Invariant?*

- The very assumptions about *Locking* and the discipline about protecting shared access
  - We assume that any accesses to shared variables are governed by *Locks*
  - Every access is protected by at least one *Lock*
    - Any **unprotected access** is an error by this discipline

Problem: How to find out which *Lock* protects a shared variable *Dynamically* (during *Runtime*)?

- Relationship between *Locks* and shared variables is not explicitly declared
  - Otherwise could e.g. perform *Static Code Analysis*

Race Detection

Eraser

Lockset Algorithm : *Dynamically* inferring *Lock* relationships

- Idea: Governing *Lock* has to be at least one of the ones held at the time of access

- 1. **C(*v*)**: A (*Lock*)set of candidate *Locks* for protecting shared location ***v***
- 2. Initialize **C(*v*)** to the set of all *Locks*
- 3. Upon access to location ***v*** by thread ***T***, refine **C(*v*)**
  - $C(v) = C(v) \cap$ `locks_held(`T`)`
- 4. **`if`** **C(*v*) = ∅**, report error

Synchronization Errors

Race Detection

Eraser

Problems (too strict)

- Initialization
  - When shared data first created and initialized, no *Locks* normally held
- Read-shared data
  - Shared data only written at initialization and then only *Read*-from (safe)
- Read-Write *Locks*
  - *Remember:* Allow a single Writer and multiple Readers
  - *Read-Write Locks* can be held in either *Write* mode or *Read* mode
    - `read_lock(r_w_m); read(v); read_unlock(r_w_m);`
    - `write_lock(r_w_m); write(v); write_unlock(r_w_m);`
  - A `write(v)` with the *Read-Write Lock* held in *Read* mode → Error

Race Detection

Eraser – Problem Mitigation

- Initialization
  - Do not refine **C(*v*)** until a different *Thread* than creator *Thread* accesses data
    - Only one *Thread* that creates shared data, *Locking* unnecessary at this phase
- Read-shared data
  - Keep refining **C(*v*)** but don't report error until ***v*** has its first *Write*-to happen

    Catches case that **C(*v*)=∅** for shared *Read* operations, and at some point there is a *Write*
- *Reader-Writer Locks*
  - Track *Locks* held only when performing *Write* separately from usual *Lock*-tracking
    - On each `read(v)` by T: $C(v) = C(v) \cap$ `locks_held(T);` `if` $C(v) = \emptyset \rightarrow$ error
    - On each `write(v)` by T: $C(v) = C(v) \cap$ `write_locks_held(T);` `if` $C(v) = \emptyset \rightarrow$ error

more strict refinement rule for *Write*-access

Synchronization Errors

Race Detection

Eraser – Implementation

- Binary (*Runtime*) tool

  Pros:

  - Does not require source code

  Cons:

  - Loses source code semantics
  - Can track *Memory* accesses at Word-level granularity

- How to monitor *Memory* access to implement tracking for the *Lockset* Algorithm?
  - Keep a *Shadow Word* for each *Memory Word* in the Program's Data Section and on the Heap
  - Each *Shadow Word* stores a *Lockset* index
  - A Table maps *Lockset* index to a set of *Locks*
  - Assumption: Not excessively many distinct *Locksets*

Synchronization Errors

Race Detection

Eraser – Overview

- Successes
  - Can help detect bugs in mature software
  - Still suffers from limitations;
    - Major: *Benign Races* (intended *Races*)
- Drawbacks
  - Slow: Monitoring each *Memory* access is costly
    - Can incur 10-30x slowdowns
  - Improvement:
    - Code *Static Analysis*
    - Smart instrumentation (e.g. sampling)
- *Lockset* Algorithm is influential & used by many tools
  - e.g. *Helgrind* (a *Race Detection* tool in *Valgrind*)

Example *Benign Race 1*:
"Double checks" for *Lazy Initialization*

```
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

Compiler reordering

Even this is possible

Idea: Faster if `init_flag` is `true` most of the time (i.e. data initialized already)
But: Still wrong! Compiler/Hardware may reorder lines / loads, etc.

Example *Benign Race 2*:
Statistical counter

```
++ nrequests;
```

Synchronization Exercises

Reader(s)-Writer(s)

Remember: Allow *exclusively* either a single Writer or multiple Readers

```
// number of readers
int readcount = 0;
// mutual exclusion to readcount
mutex_t mutex;
// exclusive writer or reader (binary sem)
semaphore_t w_or_r(1);


void writer() {
  wait(&w_or_r); // lock out readers

  // write()

  post(&w_or_r); // up for grabs
}
```

```
void reader() {
  lock(&mutex); // lock readcount
  readcount += 1; // have one more reader
  if (readcount == 1) // NOT(!): >= 1
    wait(&w_or_r); // synch w/ writers
  unlock(&mutex); // unlock readcount

  // read()

  lock(&mutex); // lock readcount
  readcount -= 1; // have one less reader
  if (readcount == 0)
    post(&w_or_r); // up for grabs
  unlock(&mutex); // unlock readcount
}
```

Reader(s)-Writer(s)

`w_or_r` provides Mutual Exclusion between Readers and Writers

- Writer `wait`/`post`, Reader `wait`/`post` when `readcount` goes from `0` to `1` or from `1` to `0`

If a Writer is writing, where will Readers be waiting?

Once a Writer exits, all Readers can fall through

- Which Reader gets to go first?
- Is it guaranteed that all Readers will fall through?

If Readers and Writers are waiting, and a Writer exits, who goes first?

- "Reader's / Writer's Priority" algorithm – Implementation requires additional variables: ActiveReaders, ActiveWriters, WaitingReaders, WaitingWriters + separate OKtoRead, OKtoWrite *Sems* -or- *CVs*

Why do Readers use `mutex`? Why don't Writers use `mutex`?

What if the `unlock` is above `if (readcount == 1)` ?

- *Data Race:* Similar principle to "*Time-Of-Check-To-Time-Of-Use*" *(TOCTOU)* bugs

Bounded-Buffer

Remember: Ring Buffer with Empty / Full limitations

```
// mutex to shared buffer internal properties (e.g. head, tail)
mutex_t mutex;
// count of empty slots
semaphore_t empty(N);
// count of filled-up slots
semaphore_t filled(0);
```

```
void producer() {
 while (1) {
  // produce_new_resource()
  wait(&empty); // wait for 1 empty slot
  lock(&mutex); // lock buffer list
  // insert_resource_to_buffer()
  unlock(&mutex); // unlock buffer list
  post(&filled); // notify +1 filled slot
 }
}
```

```
void consumer() {
 while (1) {
  wait(&filled); // wait for 1 filled slot
  lock(&mutex); // lock buffer list
  // extract_resource_from_buffer()
  unlock(&mutex); // unlock buffer list
  post(&empty); // notify +1 empty slot
  // consume_resource()
 }
}
```

Synchronization Exercises

Bounded-Buffer

Why is **`mutex`** required?

Where are the *Critical Sections*?

What conditions lead to a *Deadlock*?

- **`N`** = 0
- **`empty`** = 0 and **`filled`** = 0 , and no *Thread* has yet entered their *Critical Section*
  - Also, **`empty`** = 0 and **`filled`** = 0 can lead to single-Producer single-Consumer "ping-pong"

What happens if operations on **`mutex`** and **`filled`**/**`empty`** are switched around?

- i.e. a Consumer doing:

```
while (1) {
  lock(&mutex);
  wait(&filled);
```

If we *Block* here (due to **`filled`**=0 slots available), the **`mutex`** is also *Locked*, and will remain, because no Producer will be able to proceed into its *Critical Section* to produce & **`post()`** (increment) **`filled`**.

- e.g. sequences such as:
  - (from empty) – **`1`** Consumer, then anyone
  - (from empty) – **`N+1`** Producers, then **`1`** Consumer
- The pattern of **`post`**/**`wait`** on **`filled`**/**`empty`** is a common construct often called an *Interlock*

H_2O Problem

Form water out of two Hydrogen *Threads* and one Oxygen *Thread* (H2O)

- Two procedures: **`HArrives()`** and **`OArrives()`**
- A water molecule forms when two (2) **`H`** *Threads* are present and one (1) **`O`** *Thread*
- Otherwise, the "atoms" must wait
- Once all three are present, one of the *Threads* calls **`MakeWater()`** and only then, all three depart

Key variables:

- **`int numH`** – Keeps track of number of H *Threads* waiting
- **`int numO`** – Keeps track of number of O *Threads* waiting
- **`mutex_t mutex`** – Control access to **`numH`** and **`numO`**
- **`List<thread_status *> waitingH`** – **`H`** *Threads* waiting queue
- **`List<thread_status *> waitingO`** – **`O`** *Threads* waiting queue
-

```
struct thread_status { // to keep in queue allowing to suspend/wakeup threads
  bool ready;          // condition predicate
  condition_t cv;      // condition variable (controls thread suspension)
};
```

H_2O Problem

```
int numH = 0; // (global) number of H threads waiting
int numO = 0; // (global) number of O threads waiting
mutex_t mutex; // mutual exclusion
List<thread_status *> waitingH; // H threads waiting queue
List<thread_status *> waitingO; // O threads waiting queue
```

```
void HArrives() {
  lock(&mutex);
  numH++;
  if (numH == 2 && numO >= 1) {
    h = waitingH.pop();
    o = waitingO.pop();
    h->ready = true;
    o->ready = true;
    cond_signal(&h->cv);
    cond_signal(&o->cv);
    numH -= 2;
    numO -= 1;
    // make_water()
  }
  else {
    h = new thread_status;
    waitingH.push(h);
    while (!h->ready)
      cond_wait(&h->cv, &mutex); // releases mutex
    delete h;
  }
  unlock(&mutex);
}
```

H_2O Problem

```
int numH = 0; // (global) number of H threads waiting
int numO = 0; // (global) number of O threads waiting
mutex_t mutex; // mutual exclusion
List<thread_status *> waitingH; // H threads waiting queue
List<thread_status *> waitingO; // O threads waiting queue
```

```
void OArrives() {
  lock(&mutex);
  numO++;
  if (numH >= 2) {
    h1 = waitingH.pop();
    h2 = waitingH.pop();
    h1->ready = true;
    h2->ready = true;
    cond_signal(&h1->cv);
    cond_signal(&h2->cv);
    numH -= 2;
    numO -= 1;
    // make_water()
  }
  else {
    o = new thread_status;
    waitingO.push(o);
    while (!o->ready)
      cond_wait(&o->cv, &mutex); // releases mutex
    delete o;
  }
  unlock(&mutex);
}
```

CS-446/646

Time for Questions !