Outline for Today

- Objectives:
  - The critical section problem.
  - To learn how to reason about the correctness of concurrent programs.
  - Synchronization mechanisms
- Administrative details:
  - Groups again…check to make sure I’ve got them right
  - Picture makeup

The Critical Section Problem

```c
while (1)
{
  ...other stuff...

  critical section
  exit_region();

}
```

Proposed Algorithm for 2 Process Mutual Exclusion

```c
Boolean flag[2];
proc (int i) {
  while (TRUE){
    compute;
    flag[i] = TRUE ;
    while(flag[(i+1) mod 2]) ;
    critical section;
    flag[i] = FALSE;
  }
}
```

flag[0] = flag[1]= FALSE;
fork (proc, 1, 0);
fork (proc, 1,1);

Is it correct?

Assume they go lockstep.
Both set their own flag to TRUE. Both busywait forever on the other’s flag -> deadlock.

The Trouble with Concurrency in Threads...

```c
while(i<10) {x=x+1; i++;}
while(j<10) {x=x+1; j++;}
```

What is the value of x when both threads leave this while loop?
Proposed Algorithm for 2 Process Mutual Exclusion

- enter_region:
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) (no_op);

- exit_region:
  needin [me] = false;

Is it correct?

Interleaving of Execution of 2 Threads (blue and green)

```plaintext
enter_region:
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) (no_op);

Critical Section

exit_region:
  needin [me] = false;
```

```plaintext
enter_region:
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) (no_op);

Critical Section

exit_region:
  needin [me] = false;

Oooops!
```

Greedy Version (turn = me)

```plaintext
needin [blue] = true;
needin [green] = true;
turn = green;
turn = blue;
while (needin [green] && turn == green)
  Critical Section
    while (needin [blue] && turn == blue)(no_op);
    while (needin [blue] && turn == blue)(no_op);
  needin [blue] = false;
  while (needin [blue] && turn == blue)
    Critical Section
  needin [green] = false;
```

```plaintext
needin [blue] = true;
needin [green] = true;
turn = blue;
while (needin [green] && turn == green)
  Critical Section
    turn = green;
    while (needin [blue] && turn == blue)
      Critical Section
        Oooops!
```
Synchronization

- We illustrated the dangers of race conditions when multiple threads execute instructions that interfere with each other when interleaved.
- Goal in solving the critical section problem is to build synchronization so that the sequence of instructions that can cause a race condition are executed *as if* they were indivisible (just appearances)
- “Other stuff” can be interleaved with critical section code as well as the enter_region and exit_region protocols, but it is deemed OK.

Peterson’s Algorithm for 2 Process Mutual Exclusion

- enter_region:
  
  needin [me] = true;
  turn = you;
  while (needin [you] && turn == you) {no_op};

- exit_region:
  
  needin [me] = false;

What about more than 2 processes?

Can we extend 2-process algorithm to work with n processes?

Can we extend 2-process algorithm to work with n processes?

Idea: Tournament
Details: Bookkeeping (left to the reader)
Lamport’s Bakery Algorithm

- **enter_region:**
  
  choosing[me] = true;
  number[me] = max(number[0:n-1]) + 1;
  choosing[me] = false;
  for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
      or ((number[j] == number[me]) and (j < me)))) {skip}
  }

- **exit_region:**
  number[me] = 0;

Interleaving / Execution Sequence with Bakery Algorithm

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing= True</td>
<td>Choosing= True</td>
<td>Choosing= False</td>
<td>Choosing= False</td>
</tr>
<tr>
<td>Number [0]= 0</td>
<td>Number [1]= 1</td>
<td>Number [3]= 0</td>
<td>Number [2]= 0</td>
</tr>
</tbody>
</table>

for (j=0; n-1; j++) {
  while (choosing[j] != 0) {skip}
  while((number[j] != 0 ) and ((number[j] < number[me])
    or ((number[j] == number[me]) and (j < me)))) {skip}
}
for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}

Thread 0
Choosing= False
Number [0]= 2

Thread 1
Choosing= False
Number [1]= 1

Thread 2
Choosing= False
Number [2]= 0

Thread 3
Choosing= False
Number [3]= 1

for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}

Thread 0
Choosing= False
Number [0]= 2

Thread 1
Choosing= False
Number [1]= 1

Thread 2
Choosing= False
Number [2]= 0

Thread 3
Choosing= False
Number [3]= 1

for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}

Thread 0
Choosing= False
Number [0]= 2

Thread 1
Choosing= False
Number [1]= 1

Thread 2
Choosing= True
Number [2]= 3

Thread 3
Choosing= False
Number [3]= 1

for (j=0; n-1; j++) {
    while (choosing[j] != 0) {skip}
    while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) {skip}
}

Thread 0
Choosing= False
Number [0]= 2

Thread 1
Choosing= False
Number [1]= 0

Thread 2
Choosing= True
Number [2]= 3

Thread 3
Choosing= False
Number [3]= 1
for (j=0; n-1; j++) {
    while (choosing[j] != 0) { skip }
    while((number[j] != 0 ) and ((number[j] < number[me])
        or ((number[j] == number[me]) and (j < me)))) { skip }
}

Hardware Assistance

- Most modern architectures provide some support for building synchronization: atomic read-modify-write instructions.
- Example: test-and-set (loc, reg) [ sets bit to 1 in the new value of loc; returns old value of loc in reg ];
- Other examples: compare-and-swap, fetch-and-op

Busywaiting with Test-and-Set

- Declare a shared memory location to represent a busyflag on the critical section we are trying to protect.
- enter_region (or acquiring the "lock"):
  waitloop: tsl busyflag, R0 // R0 = busyflag; busyflag = 1
  bnz R0, waitloop // was it already set?
- exit_region (or releasing the "lock"):
  busyflag = 0

Pros and Cons of Busywaiting

- Key characteristic - the “waiting” process is actively executing instructions in the CPU and using memory cycles.
- Appropriate when:
  - High likelihood of finding the critical section unoccupied (don’t take context switch just to find that out) or estimated wait time is very short
- Disadvantages:
  - Wastes resources (CPU, memory, bus bandwidth)
Blocking Synchronization

- OS implementation involving changing the state of the "waiting" process from running to blocked.
- Need some synchronization abstraction known to OS - provided by system calls:
  - mutex locks with operations acquire and release
  - semaphores with operations P and V (down, up)
  - condition variables with wait and signal

Template for Implementing Blocking Synchronization

- Associated with the lock is a memory location (busy) and a queue for waiting threads/processes.
- Acquire syscall:
  while (busy) {enqueue caller on lock's queue} /*upon waking to nonbusy lock*/ busy = true;
- Release syscall:
  busy = false; /* wakup */ move any waiting threads to Ready queue

Pros and Cons of Blocking

- Waiting processes/threads don’t consume resources
- Appropriate: when the cost of a system call is justified by expected waiting time
  - High likelihood of contention for lock
  - Long critical sections
- Disadvantage: OS involvement -> overhead

Semaphores

- Well-known synchronization abstraction
- Defined as a non-negative integer with two atomic operations
  - P(s) - [wait until s > 0; s--]
  - V(s) - [s++]
- The atomicity and the waiting can be implemented by either busywaiting or blocking solutions.
Semaphore Usage

- Binary semaphores can provide mutual exclusion (solution of critical section problem)
- Counting semaphores can represent a resource with multiple instances (e.g. solving producer/consumer problem)
- Signaling events (persistent events that stay relevant even if nobody listening right now)

The Critical Section Problem

while (1)
{
  ... other stuff ...
  critical section
  P(mutex)
  ... other stuff ...
  V(mutex)
}

Semaphore: mutex initially 1

Condition Variables

- We build the monitor abstraction out of a lock (for the mutual exclusion) and a set of associated condition variables.
- Wait on condition: releases lock held by caller, caller goes to sleep on condition’s queue.
  When awakened, it must reacquire lock.
- Signal condition: wakes up one waiting thread.
- Broadcast: wakes up all threads waiting on this condition.

Monitor Abstraction

- Encapsulates shared data and operations with mutual exclusive use of the object (an associated lock).
- Associated Condition Variables with operations of Wait and Signal.
EnQ: \( \text{acquire (lock);} \)
if \( \text{head} = \text{null} \)
  \( \text{head} = \text{item}; \)
  \( \text{signal (lock, notEmpty);} \)
else \( \text{tail} \rightarrow \text{next} = \text{item}; \)
  \( \text{tail} = \text{item}; \)
  \( \text{release(lock);} \)

deQ: \( \text{acquire (lock);} \)
if \( \text{head} = \text{null} \)
  \( \text{wait (lock, notEmpty);} \)
else \( \text{tail} \rightarrow \text{next} = \text{item}; \)
  \( \text{tail} = \text{item}; \)
  \( \text{release(lock);} \)

EnQ: \( \text{acquire (lock);} \)
if \( \text{head} = \text{null} \)
  \( \text{head} = \text{item}; \)
  \( \text{signal (lock, notEmpty);} \)
else \( \text{tail} \rightarrow \text{next} = \text{item}; \)
  \( \text{tail} = \text{item}; \)
  \( \text{release(lock);} \)

deQ: \( \text{acquire (lock);} \)
if \( \text{head} = \text{null} \)
  \( \text{wait (lock, notEmpty);} \)
else \( \text{tail} \rightarrow \text{next} = \text{item}; \)
  \( \text{tail} = \text{item}; \)
  \( \text{release(lock);} \)
Monitor Abstraction

\textbf{EnQ}: \texttt{(acquire (lock);} \texttt{if (head == null)} \texttt{)\{head = item; signal (lock, notEmpty);} \texttt{else tail->next = item; tail = item; release(lock);\}}

\textbf{deQ}: \texttt{(acquire (lock);} \texttt{if (head == null)} \texttt{)\{wait (lock, notEmpty);} \texttt{item = head; if (tail == head) tail = null; head=item->next; release(lock);\}}

Nachos-style Synchronization

\texttt{synch.h, cc}

\begin{itemize}
  \item Semaphores
    \texttt{Semaphore::P}
    \texttt{Semaphore::V}
  \item Locks and condition variables
    \texttt{Lock::Acquire}
    \texttt{Lock::Release}
    \texttt{Condition::Wait (conditionLock)}
    \texttt{Condition::Signal (conditionLock)}
    \texttt{Condition::Broadcast (conditionLock)}
\end{itemize}

Tweedledum and Tweedledee

\begin{itemize}
  \item Separate threads executing their respective procedures. The code below is intended to cause them to forever take turns exchanging insults through the shared variable X in strict alternation.
  \item The \texttt{Sleep()} and \texttt{Wakeup()} routines operate as follows:
    \begin{itemize}
      \item Sleep blocks the calling thread,
      \item Wakeup unblocks a specific thread if that thread is blocked, otherwise its behavior is unpredictable
    \end{itemize}
\end{itemize}
a) The code shown above exhibits a well-known synchronization flaw. Outline a scenario in which this code would fail, and the outcome of that scenario.

void Tweedledum()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee);
        Sleep();
    }
}

void Tweedledee()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee);
        Sleep();
    }
}

If dee goes first to sleep, the wakeup is lost (since dum isn’t sleeping yet). Both sleep forever.

b) Show how to fix the problem by replacing the Sleep and Wakeup calls with semaphore P (down) and V (up) operations.

void Tweedledum()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledee);
        P(dum);
        Sleep();
    }
}

void Tweedledee()
{
    while(1) {
        Sleep();
        x = Quarrel(x);
        Wakeup(Tweedledum);
        V(dee);
    }
}

semaphore dee = 0;
semaphore dum = 0;

Classic Problems

There are a number of “classic” problems that represent a class of synchronization situations:

- Critical Section problem
- Producer/Consumer problem
- Reader/Writer problem
- 5 Dining Philosophers

The Critical Section Problem

while (1)
{
    other stuff...
}
critical section

DONE
Producer / Consumer

Producer:
while(whatever)
{
    locally generate item
    fill empty buffer with item
}

Consumer:
while(whatever)
{
    get item from full buffer
    use item
}

Semaphores: emptybuf initially N; fullbuf initially 0;

5 Dining Philosophers

5 Dining Philosophers

while (food available)
{
    pick up 2 adj. forks;
    eat;
    put down forks;
    think awhile;
}

Template for Philosopher

while (food available)
{
    "pick up forks"/
    "put down forks"/
    think awhile;
}
### Naive Solution

while (food available)
{
    /*pick up forks*/
    P(fork[left(me)]);
    P(fork[right(me)]);

    eat;

    /*put down forks*/
    V(fork[left(me)]);
    V(fork[right(me)]);

    think awhile;
}

Does this work?

### Simplest Example of Deadlock

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Interleaving</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(R1)</td>
<td>P(R1)</td>
<td>P(R2)</td>
</tr>
<tr>
<td>P(R2)</td>
<td>P(R2)</td>
<td>P(R1)</td>
</tr>
<tr>
<td>V(R1)</td>
<td>P(R1) waits</td>
<td>V(R2)</td>
</tr>
<tr>
<td>V(R2)</td>
<td>P(R2) waits</td>
<td>V(R1)</td>
</tr>
</tbody>
</table>

R1 and R2 initially 1 (binary semaphore)

### Conditions for Deadlock

- Mutually exclusive use of resources
  - Binary semaphores R1 and R2
- Circular waiting
  - Thread 0 waits for Thread 1 to V(R2) and Thread 1 waits for Thread 0 to V(R1)
- Hold and wait
  - Holding either R1 or R2 while waiting on other
- No pre-emption
  - Neither R1 nor R2 are removed from their respective holding Threads.

### Philosophy 101
(or why 5DP is interesting)

- How to eat with your Fellows without causing *Deadlock.*
  - Circular arguments (the circular wait condition)
  - Not giving up on firmly held things (no preemption)
  - Infinite patience with Half-baked schemes (hold some & wait for more)
- Why *Starvation* exists and what we can do about it.
Dealing with Deadlock

It can be prevented by breaking one of the prerequisite conditions:

- Mutually exclusive use of resources
  - Example: Allowing shared access to read-only files (readers/writers problem)
- Circular waiting
  - Example: Define an ordering on resources and acquire them in order
- Hold and wait
- No pre-emption

Circular Wait Condition

```c
while (food available)
{
    if (me == 0) {P(fork[left(me)]); P(fork[right(me)])};
    else {P(fork[right(me)]); P(fork[left(me)])};
    eat;
    V(fork[left(me)]); V(fork[right(me)]);
    think awhile;
}
```

Hold and Wait Condition

```c
while (food available)
{
    while (forks[me] != 2)
    {
        blocking[me] = true; V(mutex); P(sleepy[me]); P(mutex);
    }
    forks[leftneighbor(me)]--; forks[rightneighbor(me)]--;
    V(mutex);
    eat;
    P(mutex); forks[leftneighbor(me)]++; forks[rightneighbor(me)]++;
    if (blocking[leftneighbor(me)]) {blocking[leftneighbor(me)] = false; V(deepy[leftneighbor(me)])};
    if (blocking[rightneighbor(me)]) {blocking[rightneighbor(me)] = false; V(deepy[rightneighbor(me)])};
    think awhile;
}
```

Starvation

The difference between deadlock and starvation is subtle:

- Once a set of processes are deadlocked, there is no future execution sequence that can get them out of it.
- In starvation, there does exist some execution sequence that is favorable to the starving process although there is no guarantee it will ever occur.
- Rollback and Retry solutions are prone to starvation.
- Continuous arrival of higher priority processes is another common starvation situation.
Boolean eating [5];
Lock forkMutex;
Condition forksAvail;

void PickupForks (int i) {
    forkMutex.Acquire( );
    while ( eating[(i-1)%5] || eating[(i+1)%5] )
    forksAvail.Wait(&forkMutex);
    eating[i] = true;
    forkMutex.Release( );
}

void PutdownForks (int i) {
    forkMutex.Acquire( );
    eating[i] = false;
    forksAvail.Broadcast(&forkMutex);
    forkMutex.Release( );
}

while (food available) {
    forkMutex.Acquire( );
    while (forks [me] != 2) {
        blocking[me]=true;
        forkMutex.Release( );
        sleep( );
        forkMutex.Acquire( );
    }
    forks [leftneighbor(me)]--; forks [rightneighbor(me)]--;
    forkMutex.Release( );
    eat;
    forkMutex.Acquire( );
    forks[leftneighbor(me)] ++; forks [rightneighbor(me)]++;
    if (blocking[leftneighbor(me)] || blocking[rightneighbor(me)])
        wakeup ();
    forkMutex.Release( );
    think awhile;
}

Readers/Writers Problem
Synchronizing access to a file or data record in a database such that any number of threads requesting read-only access are allowed but only one thread requesting write access is allowed, excluding all readers.

Template for Readers/Writers

Reader() {while (true)}
{      /*request r access*/
    read
    /*release r access*/
}

Writer() {while (true)}
{      /*request w access*/
    write
    /*release w access*/
}
**Template for Readers/Writers**

**Reader()**

```c
{while (true) {
    fd = open(foo, 0);
    read
    close(fd);
}
```

**Writer()**

```c
{while (true) {
    fd = open(foo, 1);
    write
    close(fd);
}
```

---

**R/W - Monitor Style**

```c
Boolean busy = false;
int numReaders = 0;
Lock filesMutex;
Condition OKtoWrite, OKtoRead;

void startRead () {
    filesMutex.Acquire( );
    while ( busy )
        OKtoRead.Wait(&filesMutex);
    numReaders++;
    filesMutex.Release( );}

void endRead () {
    filesMutex.Acquire( );
    numReaders--;
    if (numReaders == 0)
        OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );}

void startWrite() {
    filesMutex.Acquire( );
    while (busy || numReaders != 0)
        OKtoWrite.Wait(&filesMutex);
    busy = true;
    filesMutex.Release( );}

void endWrite() {
    filesMutex.Acquire( );
    busy = false;
    OKtoRead.Broadcast(&filesMutex);
    OKtoWrite.Signal(&filesMutex);
    filesMutex.Release( );}
```