Concurrent Programming
A Review?

Why use processes/threads?
• To capture naturally concurrent activities within the structure of the programmed system.
  – Asynchronous events
• To gain speedup by overlapping activities or exploiting parallel hardware.

Power/energy implications?

Threads and Processes

• Decouple the resource allocation aspect from the control aspect
• Thread abstraction - defines a single sequential instruction stream (PC, stack, register values)
• Process - the resource context serving as a “container” for one or more threads (shared address space)
• Kernel threads - unit of scheduling (kernel-supported thread operations → still slow)

Threads and Processes

Address Space

User-Level Threads

• To avoid the performance penalty of kernel-supported threads, implement at user level and manage by a run-time system
  – Contained “within” a single kernel entity (process)
  – Invisible to OS (OS schedules their container, not being aware of the threads themselves or their states). Poor scheduling decisions possible.
• User-level thread operations can be 100x faster than kernel thread operations, but need better integration / cooperation with OS.
The Trouble with Threads...

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

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

Nondeterminism

- What unit of work can be performed without interruption? **Indivisible or atomic** operations.
- **Interleavings** - possible execution sequences of operations drawn from all threads.
- **Race condition** - final results depend on ordering and may not be “correct”.

Reasoning about Interleavings

- On a uniprocessor, the possible execution sequences depend on when context switches can occur:
  - Voluntary context switch - the process or thread explicitly yields the CPU (blocking on a system call it makes, invoking a Yield operation).
  - Interrupts or exceptions occurring - an asynchronous handler activated that disrupts the execution flow.
  - Preemptive scheduling - a timer interrupt may cause an involuntary context switch at any point in the code.
- On multiprocessors, the ordering of operations on shared memory locations is the important factor.

Unprotected Shared Data

```
Thread
for (i=0; i<20; i++){
    key = rand;
    SortedInsert (key);
}
for (i=0; i<20; i++){
    SortedRemove (*key);
    print (key); }
```
Critical Sections

- If a sequence of non-atomic operations must be executed as if it were atomic in order to be correct, then we need to provide a way to constrain the possible interleavings in this **critical section** of our code.
  - Critical sections are code sequences that contribute to "bad" race conditions.
  - Synchronization needed around such critical sections.
- **Mutual Exclusion** - goal is to ensure that critical sections execute atomically w.r.t. related critical sections in other threads or processes.
  - How?

The Critical Section Problem

Each process follows this template:

```
while (1)
{
    ...other stuff...  // processes in here shouldn't stop others
    enter_region( );
    critical section
    exit_region( );
}
```

The problem is to define `enter_region` and `exit_region` to ensure mutual exclusion with some degree of fairness.

Implementation Options for Mutual Exclusion

- Disable Interrupts
- Busywaiting solutions - spinlocks
  - execute a tight loop if critical section is busy
  - benefits from specialized atomic (read-mod-write) instructions
- Blocking synchronization
  - sleep (enqueued on wait queue) while C.S. is busy
Synchronization primitives (abstractions, such as locks) which are provided by a system may be implemented with some combination of these techniques.
Peterson's Alg. 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?

Interleaving of Execution of 2 Threads (blue and green)

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

Critical Section

exit_region:
  needin [me] = false;

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

Critical Section

exit_region:
  needin [me] = false;

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;

Greedy Version (turn = me)

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

needin[me] = true; turn = you;
needin[me] = true; turn = you;
needin[me] = true; turn = you;
needin[me] = true; turn = you;

Idea: Tournament
Details: Bookkeeping (left to the reader)

Hardware Assistance

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

Busywaiting with Test-and-Set

- Declare a shared memory location to represent a \texttt{busyflag} on the critical section we are trying to protect.
- \texttt{enter\_region} (or \textit{acquiring} the “lock”):
  \hspace{1cm} waitloop: tsl busyflag, R0 // R0 = busyflag; busyflag = 1
  \hspace{1cm} bnz R0, waitloop // was it already set?
- \texttt{exit\_region} (or \textit{releasing} the “lock”):
  \hspace{1cm} 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)
  - Looks busy if system is observing behavior

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
  
  \[
  P(s) \cdot \text{[wait until } s > 0; \ s-]\ 
  \]

  \[
  V(s) \cdot [s++]\ 
  \]

- 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)
- Signalling events (persistent events that stay relevant even if nobody listening right now)

**The Critical Section Problem**

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

Semaphore: mutex initially 1

**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.
**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**

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

```
deQ { acquire (lock); 
   if (head == null) 
      wait (lock, notEmpty); 
   item = head; 
   if (tail == head) tail = null; 
   head = item->next; 
   release (lock); }
```
Monitor Abstraction

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

deo: acquire (lock);
if (head == null)
   wait (lock, notEmpty);
   item = head;
if (tail == head) tail = null;
head=item->next;
release(lock);

Pitfalls

✓ Race conditions, failure to implement mutual exclusion within critical sections of code.
✓ Performance Issues (including energy implications)
   ~ Difficulty of detecting idleness with busywaiting synchronization
✓ Deadlock
✓ Starvation
✓ Priority inversion
Mars Pathfinder Example

- In July 1997 Pathfinder’s computer reset itself several times during data collection and transmission from Mars.
  - One of its processes failed to complete by a deadline, triggering the reset.
- Priority Inversion Problem
  - A low priority process held a mutual exclusion semaphore on a shared data structure but was preempted to let higher priority processes run.
  - The high priority process that failed to complete on time was blocked on this semaphore and priority inheritance was not enabled.
  - Meanwhile a bunch of medium priority processes ran, until finally the deadline ran out. The low priority semaphore-holding process never got the chance to run again in that time to the point of releasing the mutex.

5 Dining Philosophers

Template for Philosopher

```c
while (food available)
{
    /*pick up forks*/
    eat;
    /*put down forks*/
    think awhile;
}
```

Naive Solution

```c
while (food available)
{
    /*pick up forks*/
    P(fork[left(me)]);
    P(fork[right(me)]);
    eat;
    V(fork[left(me)]);
    V(fork[right(me)]);
    /*put down forks*/
    think awhile;
}
```
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.

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;
}
```

Ordered resources

Hold and Wait Condition

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