Outline Part 1

- Objectives:
  - To define the process and thread abstractions.
  - To briefly introduce mechanisms for implementing processes (threads).
  - To introduce the critical section problem.
  - To learn how to reason about the correctness of concurrent programs.

- Administrative details:
  - Groups are listed with “class pics”.
  - Pictures – make sure name-to-face mapping is correct.
  - Password protected
    name: cps110  passwd: OntheGo

Concurrence

- Multiple things happening simultaneously
  - logically or physically
- Causes
  - Interrupts
  - Voluntary context switch (system call/trap)
  - Shared memory multiprocessor

HW Support for Atomic Operations

- Could provide direct support in HW
  - Atomic increment
  - Insert node into sorted list??
- Just provide low level primitives to construct atomic sequences
  - called synchronization primitives
    LOCK(counter->lock); // Wait here until unlocked
    counter->value = counter->value + 1;
    UNLOCK(counter->lock);
- test&set (x) instruction: returns previous value of x and sets x to “1”
  LOCK(x) => while (test&set(x));
  UNLOCK(x) => x = 0;
The Basics of Processes

- Processes are the OS-provided abstraction of multiple tasks (including user programs) executing concurrently.
- A Process IS: one instance of a program (which is only a passive set of bits) executing (implying an execution context – register state, memory resources, etc.)
- OS schedules processes to share CPU.

Why Use Processes?

- To capture naturally concurrent activities within the structure of the programmed system.
- To gain speedup by overlapping activities or exploiting parallel hardware.

Separation of Policy and Mechanism (System Design Principle)

- "Why and What" vs. "How"
- Objectives and strategies vs. data structures, hardware and software implementation issues.
- Process abstraction vs. Process machinery
  Can you think of examples?
Process Abstraction

- Unit of scheduling
- One (or more*) sequential threads of control
  - program counter, register values, call stack
- Unit of resource allocation
  - address space (code and data), open files
  - sometimes called *tasks or jobs*
- Operations on processes: fork (clone-style creation), wait (parent on child), exit (self-termination), signal, kill.

  Process-related System Calls in Unix.

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

Thread

Thread

Address Space
An Example

Doc formatting process

Editing thread: Responding to your typing in your doc

Autosave thread: periodically writes your doc file to disk

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.

Process Mechanisms

• PCB data structure in kernel memory represents a process (allocated on process creation, deallocated on termination).
• PCBs reside on various state queues (including a different queue for each “cause” of waiting) reflecting the process’s state.
• As a process executes, the OS moves its PCB from queue to queue (e.g. from the “waiting on I/O” queue to the “ready to run” queue).
Context Switching

- When a process is running, its program counter, register values, stack pointer, etc. are contained in the hardware registers of the CPU. The process has direct control of the CPU hardware for now.
- When a process is not the one currently running, its current register values are saved in a process descriptor data structure (PCB - process control block).
- Context switching involves moving state between CPU and various processes’ PCBs by the OS.

Process State Transitions

![Process State Transitions Diagram]

Interleaved Schedules

![Interleaved Schedules Diagram]
PCBs & Queues

The Trouble with Concurrency in Threads...

What is the value of x when both threads leave this while loop?

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”.
Interleaving

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load x (value 0)</td>
<td>Load x (value 0)</td>
</tr>
<tr>
<td>Incr x (value 1 in reg)</td>
<td>Incr x (value 1 in reg)</td>
</tr>
<tr>
<td>Store x (value 1)</td>
<td>Store x (value 1)</td>
</tr>
<tr>
<td>Store x (value 1) for 2nd iteration</td>
<td>Store x for 2nd iteration (value 1)</td>
</tr>
<tr>
<td>Incr x (value 2 in reg)</td>
<td>Incr x (value 2 in reg)</td>
</tr>
<tr>
<td>Store x for 10th iteration (value 10)</td>
<td>Store x for 10th iteration</td>
</tr>
</tbody>
</table>

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.

The Trouble with Concurrency

- Two threads (T1, T2) in one address space or two processes in the kernel
- One counter

Solution: Atomic Sequence of Instructions

\[ \text{begin atomic} \]
\[ \text{ld} \ (\text{count}) \]
\[ \text{add} \]
\[ \text{switch} \]
\[ \text{ld} \ (\text{count+1}) \]
\[ \text{add} \]
\[ \text{wait} \]
\[ \text{st} \ (\text{count+2}) \]
\[ \text{end atomic} \]

- Atomic Sequence
  - Appears to execute to completion without any intervening operations

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 is 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...
  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
- Busywait 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.

The Trouble with Concurrency in Threads...

```
while(i<10)
    {x = x+1; i++;
Thread0
while(j<10)
    {x = x+1; j++;
Thread1
```

What is the value of x when both threads leave this while loop?

Range of Answers

**Process 0**
- LD x // x currently 0
- Add 1
- ST x // x now 1, stored over 9
- Do 9 more full loops // x = 9
- Do 9 more full loops // leaving x at 10

**Process 1**
- LD x // x currently 0
- Add 1
- ST x // x now 1
- Do 8 more full loops // x = 9
- LD x // x now 1
- Add 1
- ST x // x = 2 stored over 10
The Critical Section Problem

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

critical section
exit_region( );
}

Proposed Algorithm for 2 Process Mutual Exclusion

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?

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)

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

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

Greedy Version (turn = me)
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?

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) ;
    while((number[j] != 0 ) and ((number[j] < number[me])
      or ((number[j] == number[me]) and (j < me)))) ;
  }

• exit_region:
  number[me] = 0;

Explanation of Lamport's Bakery Algorithm

choosing[me] = true;
number[me] = max(number[0:n-1]) + 1;
choosing[me] = false;
/* choosing[i] is false when number[i] is not changing to non-zero */

for (j=0; n-1; j++ ) { 
  while (choosing[j] != 0) ;
  while((number[j] != 0 ) and ((number[j] < number[me])
      or ((number[j] == number[me]) and (j < me)))) ;
  /* While thread i is in this for-loop, number[i] is non-zero: if thread j (j<i) arrives later, while i is examining number[i], and sets choosing[j] false, number[j]=number[i]. Even if thread i has examined j already, i can enter CS(i), and j will NOT enter CS(j) until i leaves CS(i), and sets number[i] to 0. */

Interleaving / Execution Sequence with Bakery Algorithm
<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing= True</td>
<td></td>
</tr>
<tr>
<td>Number [0]= 0</td>
<td></td>
</tr>
<tr>
<td>Choosing= True</td>
<td></td>
</tr>
<tr>
<td>Number [1]= 1</td>
<td></td>
</tr>
</tbody>
</table>

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

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
    bnvz 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;
  /* wakeup */ move any waiting threads to Ready queue
Pros and Cons of Blocking

- Waiting processes/threads don’t consume CPU cycles
- 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
  \( \Delta \) overhead