Outline

• Announcements
  – No class on Thursday, February 18th
    • I will try and reschedule this
  – Programming Project 1
    • due on March 02, 1999

• Language support for synchronization
  – conditional critical regions
  – monitors
  – path expressions

Silberschatz/Galvin: Sections 6.6 - 6.8

Review: Limitations of Semaphores

• No abstraction and modularity
  – a process that uses a semaphore has to know which other processes use the semaphore, and how these processes use the semaphore
  – a process cannot be written in isolation

• Very easy to write incorrect code
  – changing the order of P and V
    • can violate mutual exclusion requirements
    • can cause deadlock
  – similar problems with omission

• Extremely difficult to verify programs for correctness

⇒ Need for still higher-level synchronization abstractions!

Language Support

• Helps simplify expression of synchronization
  – more convenient
  – more secure
  – less buggy

• We shall examine three fundamental constructs
  – conditional critical regions
  – monitors
  – path expressions

• These constructs can be found in several concurrent languages
  – Communicating Sequential Processes (CSP): critical regions
  – Concurrent Pascal: monitors
  – object-oriented languages: Modula-2, Concurrent C, Java
  – Ada83, Ada95
Conditional Critical Regions

- A high-level language declaration
  - informally, it can be used to specify that while a statement $S$ is being executed, no more than one process can access a distinguished variable $v$
  - notation
    \[
    \text{var } v : \text{shared } t; \\
    \text{region } v \text{ when } B \text{ do } S; \\
    \]
  - $v$ is shared and of type $t$
    - can only be accessed within a region statement
  - $B$ is a Boolean expression
  - $S$ is a statement
    - can be a compound statement

Conditional Critical Regions: Benefits

**Bounded-buffer producer/consumer**

```plaintext
var buffer : shared record
  pool: array [0..n-1] of item;   count, in, out: integer;
end;

Producer:
region buffer when count < n  do begin
  pool[in] := nextp;
  in := in + 1 mod n;
  count := count + 1;
end;

Consumer:
region buffer when count > 0  do begin
  nextc := pool[out];
  out := out + 1 mod n;
  count := count - 1;
end;
```

- Guards against simple errors associated with semaphores
  - e.g., changing the order of P and V operations, or forgetting to put one of them
- Division of responsibility
  - the developer does not have to program the semaphore or alternate synchronization explicitly
  - the compiler "automatically" plugs in the synchronization code using predefined libraries
  - once done carefully, reduces likelihood of mistakes in designing the delicate synchronization code

Conditional Critical Regions: Implementation

```plaintext
var mutex: semaphore;
var delay: semaphore;
var count: integer;

P( mutex );
while not B do begin
  try-and-enter;
  S;
leave-critical-region;

V( mutex );
P( delay );
if ( not B ) then V( delay );
else
  V( mutex );
  P( delay );
else count--;

fcount++;
if ( fcount > 0 ) V( second );
else V( mutex );
P( first );
fcount--;
account++;
if ( account > 0 ) V( first );
else V( second );
P( second );
account--;

if ( count > 0 ) then V( delay );
else V( mutex );
```

Monitors

- A collection of
  - private data
  - public procedures
    - only one procedure can be in the monitor at one time
    - each procedure may have
      - local variables
      - formal parameters
    - condition variables
      - queues of processes
      - wait: block on a condition variable
      - signal: unblock a waiting process
      - no-op if no process is waiting
- Who executes after a signal?
Waiting in the Monitor

- Note that the semantics of executing a `wait` in the monitor is that several processes can be waiting “inside” the monitor at any given time but only one is executing
  - wait queues are internal to the monitor
  - there can be multiple wait queues

- Who executes after a signal operation? (say P signals Q)
  - signalee Q continues (advocated by Hoare)
    - logically natural since the condition that enabled Q might no longer be true when Q eventually executes
      - P needs to wait for Q to exit the monitor
  - signaller P continues
    - Q is enabled but gets its turn only after P either leaves or executes a `wait`
    - require that the `signal` be the last statement in the procedure
      - advocated by Brinch Hansen (Concurrent Pascal)
      - easy to implement but less powerful than the other two semantics

Use of Monitors: Dining Philosophers

- Goal: Solve DP without deadlocks
  - Informally:
    - algorithm for Philosopher I
      
        ```pascal
        dp.pickup(i);
        eat;
        dp.putdown(i);
        ```
    - use array to describe state
      
        ```pascal
        var state: array [0..4] of (thinking, hungry, eating);
        ```
    - use array of condition variables to block on when required resources are unavailable
      
        ```pascal
        var self: array [0..4] of condition;
        ```
    - `pickup`:
      - changes state to hungry
      - checks if neighbors are eating
      - if not, grabs forks, and changes state to eating
      - otherwise, waits on self(i)
    - `putdown`:
      - checks both neighbors
      - if either is hungry and can proceed, releases him/her
  - What is missing?
    - philosophers cannot deadlock but can starve
    - for example, we can construct timing relationships such that a waiting philosopher will be stuck in the “self” queue forever
    - monitors have to be enhanced with a fair scheduling policy to avoid starvation
      - both at the level of accessing the monitor
      - as well as to regulate “waking-up” those that are waiting inside
    - you will explore monitor scheduling issues in Project 1

Dining Philosophers using Monitors - 2

```pascal
type dining_philosophers = monitor
var state: array [0..4] of (thinking, hungry, eating);
var self: array [0..4] of condition;

procedure entry pickup(i: 0..4);
    state[i] := hungry;
    test(i);
    if (state[i] != eating )
        self[i].wait;

procedure entry putdown(i: 0..4);
    state[i] := thinking;
    test (ln(i));
    test (rn(i));

procedure test(i: 0..4);
    if (state[ln(i)] != eating and
        state[i] == hungry and
        state(rn(i)) != eating)
        state[i] := eating;
        self[i].signal;
```

Dining Philosophers using Monitors - 3

- What is missing?
  - philosophers cannot deadlock but can starve
    - for example, we can construct timing relationships such that a waiting philosopher will be stuck in the “self” queue forever
  - monitors have to be enhanced with a fair scheduling policy to avoid starvation
    - both at the level of accessing the monitor
    - as well as to regulate “waking-up” those that are waiting inside
  - you will explore monitor scheduling issues in Project 1
Monitors: Other Issues

- **Expressibility**: Are monitors more/less powerful than semaphores or conditional critical regions?
  - these three constructs are equivalent
  - the same kinds of synchronization problems can be expressed in each
  - the other two can be implemented using any one of the constructs
  - e.g., critical regions and monitors using semaphores
    - we talked about how critical regions can be implemented
    - in Project 1, you will build a semaphore-based implementation of monitors

- **Do monitors have any limitations?**
  - absence of concurrency within a monitor
    - workarounds introduce all the problems of semaphores
    - monitor procedures will need to be invoked before and after
    - possibility of improper access, deadlock, etc.

Path Expressions

- Originally proposed by Campbell and Habermann
  - A higher-level construct than so far discussed
    - the construct restricts the set of admissible execution histories of the operations on a shared resource so that no incorrect state is ever reached
      - it indicates the order in which operations on the shared resource can be interleaved
  
  path S end;
  - where S denotes possible execution histories
  - S: an expression whose variables are operations and whose operators are
    - sequencing (;): path A; B; end means A must be performed first followed by B
    - selection (+): path A+B end means that only one of A or B can be executed at a given time, but the order of execution does not matter
    - concurrency ({}): denotes that any number of instances of the operations within the {} can be active at any time

Readers-Writers using Path Expressions

- **Reader’s priority**:
  
  path {read} + write end
  - either several read operations or a single write operation
  - due to [read], if a reader is reading, subsequent readers gain immediate access
  - actually: weak-reader’s priority because when a writer exits, and both a reader and a writer are waiting, any one can get the next access

- **Writer’s priority**:
  
  path start-read + { start-write ; write } end
  path { start-read ; read } + write end
  - when multiple path expressions are provided, all of them must be satisfied
  - second expression: no reader can execute start-read when a writer is performing a write operation
  - first expression: a writer can execute a start-write when another writer is doing the write operation, or a reader is doing the read operation

Synchronization in real OSes

- **Unix**: Single CPU OS
  - implement critical sections using **interrupt elevation**
    - disallow interrupts that can modify the same data
  - another possibility: interrupts never “force” a context switch
    - they just set flags, or wake up processes
  - primitives
    - sleep (address, priority);
    - wake_up (address); -- wakes up all processes sleeping on address
  - typical code
    
    ```
    while (locked) sleep (bufaddr);
    locked = false; wake_up (bufaddr);
    ```
Synchronization in real OSes - 2

- Solaris 2: multi-CPU OS
  - for brief accesses only
    - adaptive mutexes
    - starts off as a standard spinlock semaphore
      - if lock is held by running thread, continues to spin
        » valid only on a multi-CPU system
      - otherwise blocks
  - for long-held locks
    - condition variables
      - wait and signal
    - reader-writer locks
      - for frequent mostly read-only accesses

Synchronization and Communication

- Synchronization primitives
  - assuming shared memory
    - locks
    - semaphores
    - monitors
  - Synchronization can also be constructed using message-passing IPC
    - primitives combine data transfer and synchronization
      - a receive blocks for a message: equivalent to a wait
      - a send enables a process blocking on a receive to make progress: equivalent to a signal
    - you will explore this relationship in more detail in Project 1

Next Lecture

- Process Deadlocks
  - system model
  - deadlock characterization
  - methods for handling deadlocks
  - deadlock prevention

Readings

- Silberschatz/Galvin: Sections 7.1 - 7.4

- Reminder: No class on Thursday!