Lecture 6
Process Coordination

February 9, 2004
Outline

• Announcements
  – Lab 1 demos today and tomorrow
    • My office: 715 Broadway, Room 704
  – Questions?

• Process Cooperation
  – Shared memory and message passing
  – Critical sections
    • Petersen’s 2-process solution
  – Locks, Semaphores, Condition variables

[Silberschatz/Galvin/Gagne, Sections 7.1 – 7.4]
(Review) Process Cooperation

• Why do processes cooperate?
  – modularity: breaking up a system into several sub-systems
    – e.g.: an interrupt handler and device driver that need to communicate
  – convenience: users might want to have several processes share data
  – speedup: a single program is run as several sub-programs

• How do processes cooperate?
  – communication abstraction: producers and consumers
    • producers produce a piece of information
    • consumers use this information
  – abstraction helps deal with general “phenomena” and simplifies correctness arguments

• Two general classes of process cooperation techniques
  – shared memory
  – message passing
Shared Memory (Procedure-oriented System)

- Processes can directly access data written by other processes
  - examples: POSIX threads, Java, Mesa, small multiprocessors

- A finite-capacity shared buffer

```plaintext
N: integer                        -- buffer size
nextin = nextout = 0 initially;   -- start of buffer
buffer: array of size N

Producer:
Repeat
  -- produce an item in tempin
  while (nextin+1) mod n = nextout do wait-a-bit;
  buffer[nextin] := tempin;
  nextin := (nextin+1) mod n;

Consumer:
Repeat
  while nextin = nextout do wait-a-bit;
  tempout := buffer[nextout];
  nextout := (nextout+1) mod n;
  -- consume the item in tempout
```
Message Passing (Message-oriented System)

- Execution is in separate address spaces
  - communication using message channels
  - examples: UNIX processes, large multiprocessors, etc.

- Components
  - messages and message identifiers
  - message channels and ports
    - channels (pipes) must be bound to ports
    - queues associated with ports
  - message transmission system calls
    - SendMessage[channel, body] returns id
    - AwaitReply[id]
    -RecvMessage[port] returns id
    - SendReply[id, body]

- Many variants: See Section 4.5
  ➤ Focus on shared memory for next few lectures
Bounded Buffers Using Counters

N: integer                        -- buffer size
counter: integer = 0 initially;
nextin = nextout = 0 initially;  -- start of buffer
buffer: array of size N

Producer:
Repeat
  -- produce an item in tempin
  while counter = N do wait-a-bit;
  buffer[nextin] := tempin;
  nextin := (nextin+1) mod n;
  counter := counter+1;

Consumer:
Repeat
  while counter = 0 do wait-a-bit;
  tempout := buffer[nextout];
  nextout := (nextout+1) mod n;
  counter := counter-1;
  -- consume the item in tempout

Producer and Consumer processes are asynchronous! execution of these two statements can be interleaved (e.g., because of interrupts)
Interleaving of Increment/Decrement

- Each of increment and decrement are actually implemented as a series of machine instructions on the underlying processor

  **Producer**
  
  register1 := counter
  
  register1 := register1 + 1
  
  counter := register1

  **Consumer**

  register2 := counter

  register2 := register2 - 1

  counter := register2

- An interleaving
  
  - counter = 5; a producer followed by a consumer

  **Producer**

  register1 := counter

  register1 := register1 + 1

  counter := register1

  **Consumer**

  register2 := counter

  register2 := register2 - 1

  counter := register2

  {register1 = 5}

  {register1 = 6}

  {register2 = 5}

  {register2 = 4}

  {counter = 6}

  {counter = 4}
The Problem

• Increment and decrement are not *atomic* or *uninterruptable*
  – two or more operations are executed *atomically* if the result of their execution is equivalent to that of some serial order of execution
  – operations which are always executed atomically are called *atomic*
    • byte read; byte write;
    • word read; word write

• The code containing these operations creates a *race condition*
  – produces inconsistencies in shared data

• Reasons for non-atomic execution
  – interrupts
  – context-switches
The Solution

• The producer and consumer processes need to synchronize
  – so that they do not access shared variables at the same time

  – this is called mutual exclusion
    • the shared and critical variables can be accessed by only one process at a time
  – access must be serialized even if the processes attempt concurrent access
    • in the previous example: counter increment and decrement operations

• General framework for achieving this: Critical Sections
  – work independent of the particular context or need for synchronization
Critical Sections

- Critical sections: General framework for process synchronization

\[
\text{ENTRY-SECTION} \\
\text{CRITICAL-SECTION-CODE} \\
\text{EXIT-SECTION}
\]

- the ENTRY-SECTION controls access to make sure that no more than one process $P_i$ gets to access the critical section at any given time
  - acts as a guard
- the EXIT-SECTION does bookkeeping to make sure that other processes that are waiting know that $P_i$ has exited

- How can we implement critical sections?
  - turn off interrupts around critical operations
  - build on top of atomic memory load/store operations
  - provide higher-level primitives
Two-Process Solutions: Turn Counters

• Shared integer variable: $\text{turn}$ (initialized to 0)
  – for $i \in \{0, 1\}$: $P_i$ executes:
    
    \begin{verbatim}
    while (turn != i) wait-a-bit;
    CRITICAL SECTION;
    turn := j;
    \end{verbatim}

    – the while loop is the *entry* section
      • process $P_i$ waits till its turn occurs
    – the single instruction $\text{turn} := j$ constitutes the *exit* section
      • informs the other process of its turn

• Mutual exclusion?
  – assume atomic loads and stores

• Drawbacks?
  – if $P_1$ never wants to execute the critical section, $P_0$ cannot reenter;
    • access *must* alternate
Two-Process Solutions: Array of Flags

- Boolean array $\text{flag}$ (initialized to false), $P_1$ executes:
  1: $\text{flag}[i] := \text{true};$
  2: while $\text{flag}[j]$ wait-a-bit;
  3: $\text{flag}[i] := \text{false};$

- Mutual exclusion?

  ![Diagram showing the execution of $P_0$ and $P_1$ and their access to the critical section]

- Is this good enough?
- No: $P_0$ and $P_1$ can be looping on instruction 2 forever
Criteria for Correctness

Three conditions

• Mutual exclusion
• Progress
  – at least one process requesting entry to a critical section will be able to enter it if there is no other process in it
• Bounded waiting
  – no process waits indefinitely to enter the critical section once it has requested entry
Two-Process Solutions: Petersen’s Algorithm

- Combines the previous two ideas
  
  1: \text{flag}[i] := \text{true}
  
  2: \text{turn} := j
  
  3: while (\text{flag}[j] \text{ and } (\text{turn} == j)) \text{ wait-a-bit}

  CRITICAL SECTION

  4: \text{flag}[i] := \text{false}

- Does the algorithm satisfy the three criteria?
Petersen’s Algorithm: Mutual Exclusion

1: flag[i] := true
2: turn := j
3: while (flag[j] and (turn == j)) wait-a-bit
   CRITICAL SECTION
4: flag[i] := false

• Suppose: P_0 is in its critical section, and P_1 is wanting to enter
• This can happen only if either
  – (case 1) P_0 found flag[1] false, or
  – (case 2) P_0 found turn == 0
  – in the first case: P_1 will set turn after P_0 did, and find turn == 0
  – in the second case: P_1 has already set turn = 0
  – in both cases: P_1 will wait till flag[0] == false
Petersen’s Algorithm: Progress and Bounded Waiting

1: flag[i] := true
2: turn := j
3: while (flag[j] and (turn == j)) wait-a-bit
   CRITICAL SECTION
4: flag[i] := false

- To prove progress:
  - if P_1 is not ready to enter the critical section
    - flag[1] will be false \(\Rightarrow\) P_0 can enter

- To prove bounded waiting:
  - let P_0 be in the critical section and P_1 be waiting on instruction 3 above
  - if P_0 exits and goes elsewhere,
    - either P_1 will find flag[0] to be false
    - if not, P_0 will attempt to reenter the critical section, setting turn := 1
    - in either case, P_1 will find the condition for waiting in (3) to be false and will enter the critical section
Can These Solutions be Extended to >2 Processes?

- N-process solutions
  - do exist: Bakery Algorithm (see Section 7.2.2)
  - but reasoning gets even more complicated!

- So, we can implement critical sections using only support for atomic memory loads and stores
- But, there must be an easier way!

- Higher-level synchronization primitives
  - locks (mutexes), semaphores, condition variables
  - rely on more support from hardware
    - disabling of interrupts: only around the primitives
    - atomic read-modify-write operations
Synchronization Primitives (1): Locks (Mutexes)

- **Locks**
  - a single boolean variable $L$
    - in one of two states: AVAILABLE, BUSY
  - accessed via two *atomic* operations
    - **LOCK** (also known as *Acquire*)
      
      ```
      while ( L != AVAILABLE ) wait-a-bit
      L = BUSY;
      ```
    - **UNLOCK** (also known as *Release*)
      
      ```
      L = AVAILABLE;
      
      wake up a waiting process (if any)
      ```
  - process(es) waiting on a LOCK cannot “lock-out” process doing UNLOCK

- **Critical sections using locks**
  
  ```
  LOCK( L )
  CRITICAL SECTION
  UNLOCK( L )
  ```
  
  - Mutual exclusion? Progress? Bounded waiting?
Synchronization Primitives (2): Semaphores

- **Semaphores**
  - a single integer variable $S$
  - accessed via two *atomic* operations
    - **WAIT** (sometimes denoted by $P$)
      ```
      while $S \leq 0$ do wait-a-bit;
      S := S-1;
      ```
    - **SIGNAL** (sometimes denoted by $V$)
      ```
      S := S+1;
      ```
      wake up a *waiting* process (if any)
  - WAITing process(es) cannot “lock out” a SIGNALing process

- **Binary semaphores**
  - $S$ is restricted to take on only the values 0 and 1
  - **WAIT** and **SIGNAL** become similar to **LOCK** and **UNLOCK**
  - are *universal* in that counting semaphores can be built out of them
Uses of Semaphores

• Mutual exclusion (initially $S = 1$)
  
  $\text{P}(S)$
  CRITICAL SECTION
  $\text{V}(S)$

• Sequencing (initially $S = 0$)
  
  $P_1$  
  $P_2$

  Statement 1
  $\text{V}(S)$
  $\text{P}(S)$

  Statement 2

• Detailed examples of its use in Lecture 7
Universality of Binary Semaphores

- Implement operations on a (counting) semaphore \texttt{CountSem}
  - use binary semaphores $S_1 = 1$, $S_2 = 0$
  - integer $C =$ initial value of counting semaphore

\begin{align*}
\text{P}(\text{CountSem}) & & \text{V}(\text{CountSem}) \\
\text{P}(S_1); & & \text{P}(S_1); \\
C := C - 1; & & C := C + 1; \\
\text{if ( } & & \text{if ( } \\
C < 0 \text{ ) then} & & C <= 0 \text{ ) then} \\
\text{begin } & & \text{V}(S_2); \\
V(S_1); & & \text{else } \text{V}(S_1); \\
V(S_1); & & \text{V}(S_1);
\end{align*}

- $S_1$ ensures mutual exclusion for accessing $C$
- $S_2$ is used to block processes when $C < 0$
- is a race condition possible after $\text{V}(S_1)$ but before $\text{P}(S_2)$?
Synchronization Primitives (3): Condition Variables

- Condition variables
  - an *implicit* process queue
  - three operations that *must be performed within a critical section*
    - **WAIT**
      - associate self with the implicit queue
      - suspend self
    - **SIGNAL**
      - wake up exactly one suspended process on queue
        - has no effect if there are no suspended processes
    - **BROADCAST**
      - wake up all suspended processes on queue

- Two types based on what happens to the process doing the SIGNAL
  - Mesa style (Nachos uses Mesa-style condition variables)
    - **SIGNAL-ing process continues in the critical section**
      - resumed process must re-enter (so, is not guaranteed to be the next one)
  - Hoare style
    - **SIGNAL-ing process immediately exits the critical section**
      - resumed process now occupies the critical section
Uses of Condition Variables

- Can be used for constructing
  - critical sections, sequencing, …

- Primary use is for waiting on an event to happen
  - after checking that it has not already happened
    - WHY IS THIS IMPORTANT?

- Example: Three processes that need to cycle among themselves
  <print 0>; <print 1>; <print 2>; <print 0>; <print 1>; …
  - One variable: \( \text{turn} \); three condition variables: \( \text{cv}_0, \text{cv}_1, \text{cv}_2 \)
  - Process \( P_i \) executes (in a critical section)

\[
\text{while} \ ( \text{turn} \neq i) \ 	ext{WAIT}(\text{cv}_i) \\
<\text{do the operation}> \\
\text{turn} := (\text{turn} + 1) \mod 3; \text{SIGNAL}(\text{cv}_{\text{turn}})
\]
Higher-level Synchronization Primitives

- Several additional primitives are possible
  - Built using locks, semaphores, and condition variables

- An example: Event Barriers (see Nachos Lab 3)