Lecture 5
Interprocess Cooperation and Synchronization
February 11, 2002

Outline

- Announcements
  - Lab 1 due 6:00pm today, demos scheduled for Wednesday, Thursday
  - Lab 2 out today, due back Feb 25th

- Process Cooperation
  - Why required?
  - Shared memory and message passing

- Synchronization
  - Critical sections
  - Two-process solutions for mutual exclusion
  - Higher-level primitives: locks, semaphores, condition variables

[ Silberschatz/Galvin/Gagne: Sections 4.4-4.5, 7.1-7.2, 7.4]

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
  
  \[
  \begin{align*}
  N & : \text{integer} & \text{buffer size} \\
  \text{nextin} & = \text{nextout} = 1 \text{ initially}; & \text{start of buffer} \\
  \text{buffer} & : \text{array of size N}
  \end{align*}
  \]

  \textbf{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;

  \textbf{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 operations
    - 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

\[ N: \text{integer} \quad \text{-- buffer size} \]
\[ \text{counter: integer = 0 initially;} \quad \text{-- buffer size} \]
\[ \text{nextin = nextout = 1 initially;} \quad \text{-- start of buffer} \]

\text{buffer: array of size N}

\textbf{Producer:}

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

\textbf{Consumer:}

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

---

Interleaving of Increment/Decrement

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

\texttt{Producer}
\begin{align*}
\text{register1} & := \text{counter} \\
\text{register1} & := \text{register1} + 1 \\
\text{counter} & := \text{register1}
\end{align*}

\texttt{Consumer}
\begin{align*}
\text{register2} & := \text{counter} \\
\text{register2} & := \text{register2} - 1 \\
\text{counter} & := \text{register2}
\end{align*}

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

\texttt{Producer}
\begin{align*}
\text{register1} & := \text{counter} \\
\text{register1} & := \text{register1} + 1 \\
\text{counter} & := \text{register1}
\end{align*}

\texttt{Consumer}
\begin{align*}
\text{(register1 = 5)} & \\
\text{(register1 = 6)} & \\
\text{(register2 = 5)} & \\
\text{(register2 = 4)} & \\
\text{(counter = 6)} & \\
\text{(counter = 4)} & \\
\text{counter} & := \text{register2}
\end{align*}

---

The Problem

- Increment and decrement are not \textit{atomic} or \textit{uninterruptable}
  - two or more operations are executed \textit{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 \textit{atomic}
    - byte read; byte write;
    - word read; word write

- The code containing these operations creates a \textit{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

  ENTRY-SECTION
  CRITICAL-SECTION-CODE
  EXIT-SECTION

  - the ENTRY-SECTION controls access to make sure that no more than one process Pi 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 Pi 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: turn (initialized to 0)
  - for i ∈ {0, 1}; Pi executes:
    while (turn != i) wait-a-bit;
    CRITICAL SECTION;
    turn := j;
  - the while loop is the entry section
    - process Pi waits till its turn occurs
  - the single instruction turn := j constitutes the exit section
    - informs the other process of its turn
- Mutual exclusion?
  - assume atomic loads and stores
- Drawbacks?
  - if Pi never wants to execute the critical section, P-i cannot reenter;
    - access must alternate

Two-Process Solutions: Array of Flags

- Boolean array flag (initialized to false), Pi executes:

  1: flag[i] := true;
  2: while flag[j] wait-a-bit;
  3: flag[i] := false;

- Mutual exclusion?

- 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 ($\text{flag}[j]$ and ($\text{turn} == j$)) wait-a-bit
   \hspace{1cm} \text{CRITICAL SECTION}
4: $\text{flag}[i] := \text{false}$

Does the algorithm satisfy the three criteria?

Petersen’s Algorithm: Mutual Exclusion

1: $\text{flag}[i] := \text{true}$
2: $\text{turn} := j$
3: while ($\text{flag}[j]$ and ($\text{turn} == j$)) wait-a-bit
   \hspace{1cm} \text{CRITICAL SECTION}
4: $\text{flag}[i] := \text{false}$

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 $\text{flag}[1]$ false, or
- (case 2) $P_0$ found $\text{turn} == 0$
  - in the first case: $P_1$ will set $\text{turn}$ after $P_0$ did, and find $\text{turn} == 0$
  - in the second case: $P_1$ has already set $\text{turn} == 0$
  - in both cases: $P_1$ will wait till $\text{flag}[0] == \text{false}$

Petersen’s Algorithm: Progress and Bounded Waiting

To prove progress:
- if $P_1$ is not ready to enter the critical section
  - $\text{flag}[1]$ will be false → $P_0$ 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 $\text{flag}[0]$ to be false
    - if not, $P_0$ will attempt to reenter the critical section, setting $\text{turn} := j$
    - 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 6.2.2)
  - 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 <= 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)
  
  P( S )
  CRITICAL SECTION
  V( S )

- Sequencing (initially S = 0)
  
  P1
  Statement 1
  V( S )
  P2
  Statement 2

- Detailed examples of its use in Lecture 6
Universality of Binary Semaphores

- Implement operations on a (counting) semaphore \texttt{CountSem}
  - use binary semaphores S1 = 1, S2 = 0
  - integer C = initial value of counting semaphore

  \begin{verbatim}
  P(CountSem) V(CountSem)
  P(S1); C := C-1; if (C < 0 ) then begin V(S1); P(S2); end else V(S1);
  C := C+1; if (C <= 0) then V(S2);
  \end{verbatim}

  - S1 ensures mutual exclusion for accessing C
  - S2 is used to block processes when C < 0
  - is a race condition possible after V(S1) but before P(S2)?

Synchronization Primitives (3): Condition Variables

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

- Two types based on what happens to the process doing the \texttt{SIGNAL}
  - Mesa style (Nachos uses Mesa-style condition variables)
    - \texttt{SIGNAL}-ing process continues in the critical section
    - resumed process must re-enter (so, is not guaranteed to be the next one)
  - Hoare style
    - \texttt{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
  - \texttt{print 0}; \texttt{print 1}; \texttt{print 2}; \texttt{print 0}; \texttt{print 1}; …
  - One variable: \texttt{turn}; three condition variables: \texttt{cv0, cv1, cv2}
  - Process P_i executes (in a critical section)

  \begin{verbatim}
  if (turn != i) \texttt{WAIT(cv_i)}
  <do the operation>
  turn := (turn + 1) mod 3; \texttt{SIGNAL(cv_{turn})}
  \end{verbatim}

Higher-level Synchronization Primitives

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

- An example: \texttt{Event Barriers} (see Nachos Lab 2)
Next Two Lectures

- Implementing the synchronization primitives
- Classical process synchronization problems
  - Mutual exclusion
  - Sequencing
  - Producer consumer
  - Readers-writers
  - Dining philosophers
- Language support for process synchronization
  - Critical regions
  - Monitors
  - Message passing

Reading
- Silberschatz/Galvin: Chapter 7

Extra Material
(if time permits)

Implementing the Synchronization Primitives

- Need support for atomic operations from the underlying hardware
  - applicable only to a small number of instructions
  - else, can implement critical sections this way

Three choices
- Use n-process mutual-exclusion solutions
  - complicated
- Selectively disable interrupts on uniprocessors
  - so, no unanticipated context switches ⇒ atomic execution
  - solution adopted in Nachos (see Lab 2 for details)
- Rely on special hardware synchronization instructions
- Can implement one primitive in terms of another
  - Nachos Lab 2

Implementation Choices (1): Interrupt Disabling

- Semaphores
  \[
  P(S) \quad \text{DISABLE-INTERRUPTS} \\
  \text{while } S \leq 0 \text{ do } \text{wait-a-bit} < \text{ENABLE-INTERRUPTS; YIELD CPU} \\ 
  S := S-1; \quad \text{ENABLE-INTERRUPTS} \\
  \]

- V(S)
  \[
  \text{DISABLE-INTERRUPTS} \\
  S := S+1; \quad \text{wake up a waiting process} \quad \text{ENABLE-INTERRUPTS} \\
  \]

- Drawback
  - a process spins on this loop till it gets a chance to enter critical section
  - can waste substantial amount of CPU cycles idling
  - Even if wait-a-bit is implemented as
    - give up CPU (i.e. put at the end of ready queue)
    - since there are still context switches
  - not a very useful utilization of valuable cycles
Efficient Semaphores

- Implement P and V differently
  - maintain an explicit wait queue organized as a scheduler structure

```
type semaphore = record
  value: integer;
  L: list of processes;
end;
```

```
P(S):  S.value := S.value - 1;
    if ( S.value < 0 ) then begin
      add process to S.L block;
    end;

V(S):  S.value := S.value + 1;
    if ( S.value <= 0 ) then begin
      remove P from S.L
      wakeup(P);
    end;
```

- still need atomicity: can use previously discussed solutions
  - can have spinning but only for a small period of time (~10 instructions)
  - queue enqueue/dequeue must be fair
    - not required by semantics of semaphores

Implementation Choices (2): Hardware Support

- Rationale: Hardware instructions enable simpler/efficient solutions to common synchronization problems
  - disabling interrupts is a brute-force approach
  - does not work on multiprocessors
    - simultaneous disabling of all interrupts is not feasible

- Two common primitives
  - test-and-set
  - swap

Semantics of Hardware Primitives

- **Test-and-set**
  - given boolean variables X, Y, atomically set X := Y; Y := true

  ```
  function test-and-set(var target:boolean) boolean;
  begin
    test-and-set := target;
    target := true;
  end;
  ```

- **Swap**
  - atomically exchange the values of given variables X and Y

  ```
  temp = X; X = Y; Y = temp;
  ```

  - can emulate test-and-set

  ```
  function test-and-set(var v: boolean): boolean
  var t := true;
  swap (v, t);
  return t;
  ```

Implementing Locks Using Test-and-Set

```
LOCK:
  L : boolean := false
  while test-and-set(lock) wait-a-bit

UNLOCK
  lock := false
```

- Properties of this implementation
  - Mutual exclusion?
    - first process P_i entering critical section sets lock := true
    - test-and-set (from other processes) evaluates to true after this
  - Progress?
    - trivially true
  - Unbounded waiting
    - possible since depending on the timing of evaluating the test-and-set primitive, other processes can enter the critical section first
    -See Section 6.3 for a solution to this problem
Synchronization Primitives 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
    ENTRY: `while (locked) sleep(bufaddr);`
    `locked = true;`
    EXIT: `locked = false; wake_up (bufaddr);`

Synchronization Primitives in Real OSes (contd.)

- 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