CIS 415
Operating Systems
Synchronization

Prof. Allen D. Malony
Department of Computer and Information Science
Fall 2014
Logistics

- Office hours
  - Friday, Oct. 24 changed to Wednesday, Oct. 22
  - Monday, Oct. 27 changed to Wednesday, Oct. 29

- Help Desk!!!
  - Wednesday, Oct. 22, 11:00-13:00, Room 100
  - TA’s office hour held during Help Desk period

- Project 1 due next Monday, October 27, midnight

- Assignment 1 handed back this week
Outline

- Background
- Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Mutual Exclusion (Mutex) Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
Objectives

- To present the concept of process synchronization.
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data.
- To present both software and hardware solutions of the critical-section problem.
- To examine several classical process-synchronization problems.
- To explore several tools that are used to solve process synchronization problems.
Background

- Early operating systems research was where fundamental problems of concurrency first arose
  - Dijkstra was the 1972 Turing Award winner
- OS must function as a concurrent system
- Concurrent systems must address problems of how concurrent processes work consistently together
Concurrent and Synchronization

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
  - OS itself may very well be designed to have concurrent execution of software as part of its execution
- There are different kinds of resources that are shared between processes:
  - Physical (terminal, disk, network, …)
  - Logical (files, sockets, memory, …)
  - Memory
- Concurrent access to shared resources must be done in a consistent manner
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- This is the role of synchronization
Resources

- There are different kinds of resources that are shared between processes:
  - Physical (terminal, disk, network, …)
  - Logical (files, sockets, memory, …)
- For the purposes of this discussion, let us focus on “memory” to be the shared resource
  - Processes can all read and write into memory (variables) that are shared
Problems Due to Sharing

- Consider a shared printer queue
  - spoolQueue[N]
- 2 processes want to enqueue an element each to this queue
- tail points to the current end of the queue
- Each process needs to do
  
  ```
  tail = tail + 1;
  spoolQueue[tail] = “element”;
  ```
What we are trying to do?

Process 1

tail = tail + 1;
spoolQueue[tail] = X

Process 2

tail = tail + 1;
spoolQueue[tail] = Y
What is the problem?

- \( \text{tail} = \text{tail} + 1 \) is NOT a single machine instruction
  - So, what? Why do we care?

- It can translate as follows to assembly code:
  
  - \( \text{Load tail, R1} \)
  - \( \text{Add R1, 1, R2} \)
  - \( \text{Store R2, tail} \)

- These 3 machine instructions may NOT be executed atomically (can not guarantee it)

- So, what is the problem?
Interleaving

- Each process executes this set of 3 instructions
- Interrupts might happen at any time
  - Thus, a context switch can happen at any time
- Suppose we have the following scenario:

  Process 1
  
  Load tail, R1
  Add R1, 1, R2
  Store R2, tail

  Process 2
  
  Load tail, R1
  Add R1, 1, R2
  Store R2, tail
Leading to ...

Process 1

tail = tail + 1;
spoolQueue[tail] = X

Process 2

tail = tail + 1;
spoolQueue[tail] = Y
Race Conditions

- Situations like this that can lead to erroneous execution are called *race conditions*.
- Definition: a *race condition* in concurrent execution is when the outcome of the execution depends on the particular interleaving of concurrent instructions.
- Debugging race conditions can be fun! Not!
- Race conditions are timing dependent!
  - Errors can be non-repeatable
- “State” can potentially become inconsistent
Ok, how do we avoid race conditions?

- Definition: A set of instructions is *atomic* if it executed as if it was a single instruction
  - While one process is executing those instructions, another process cannot execute the same instructions

- Suppose the 3 assembly instructions we were looking at were atomic

- Does this avoid the race condition?

- Definition: When executing a set of instructions is vulnerable to a race condition, that set of instructions are said to constitute a *critical section*
Real Life Critical Section

Waiting to buy iPad 2 at Beijing Apple Store

Waiting to ... well ... you know
Critical Section Problem (Dijkstra, 1965)

- Consider system of \( n \) processes \( \{P_0, P_1, \ldots, P_{n-1}\} \)
- Each process has critical segment of code
  - Process may be changing common variables, updating table, writing file, …
  - When one process in critical section, no other may be in its critical section (mutual exclusion)
- Critical section problem is to design a protocol between the processes to solve this
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section
Critical Section

- General structure of process $P_i$

```java
    do {
        entry section
        critical section
        exit section
        remainder section
    } while (true);
```
Requirements for Solution

- **Mutual exclusion**
  - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections

- **Progress**
  - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely

- **Bounded waiting**
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the $N$ processes
How to Implement Critical Sections

- Implementing critical sections follows three fundamental approaches

1. Disable Interrupts
   - Effectively stops scheduling other processes

2. Busy-wait/spinlock solutions
   - Pure software solutions
   - Integrated hardware-software solutions

3. Blocking Solutions
Disabling Interrupts

- We already know how to prevent another process from interrupting the current running process
  - Do not allow interrupts

- Advantages:
  - Simple to implement (single instruction)

- Disadvantages:
  - Do not want to give such power to user processes
  - Does not work on a multiprocessor
  - Disables multiprogramming even if another process is NOT interested in critical section
Busy Waiting

- Overall philosophy:
  - Keep checking some state (variables) until they indicate other process(es) are not in critical section

```c
locked = FALSE;

P1 {
  while (locked == TRUE);
  locked = TRUE;

  /***************************************************************************/
  (critical section code)
  /***************************************************************************/

  locked = FALSE;
}

P2 {
  while (locked == TRUE);
  locked = TRUE;

  /***************************************************************************/
  (critical section code)
  /***************************************************************************/

  locked = FALSE;
}
```

- Is there an interleaving where this fails?
Strict Alternation

- Consider this code
- Does it work?
- What are the problems, if any?
  - Is there mutual exclusion?
  - Is there progress?
  - Is there bounded waiting?

```c
turn = 0;

P0 {
    while (turn != 0);
    critical section
    turn = 1;
}

P1 {
    while (turn != 1);
    critical section
    turn = 0;
}
```
Fixing the “progress” requirement

- What about this code?
- Problem:
  - Both can set their flags to true and wait indefinitely for the other
  - They got mutual exclusion, for sure!
- Does NOT meet the progress or bounded waiting requirements

```c
bool flag[2]; // initialized to FALSE

P0 {
    flag[0] = TRUE;
    while (flag[1] == TRUE)
        ;
    /* critical section */
    flag[0] = FALSE;
}

P1 {
    flag[1] = TRUE;
    while (flag[0] == TRUE)
        ;
    /* critical section */
    flag[1] = FALSE;
}
```
Peterson’s Solution

- Consider 2 processes
- Assume that the LOAD and STORE instructions are atomic and cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `boolean flag[2]`
- Variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section
  - `flag[i] = true` implies that process $P_i$ is ready!
Algorithm for Process $P_i$

do { 
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);  
        critical section
    flag[i] = false;
        remainder section
} while (true);
Does Peterson’s Solution work?

- Prove that the 3 CS requirement are met:
  - Mutual exclusion is preserved
    - $P_i$ enters CS only if:
      - either $flag[j] = false$ or $turn = i$
        - if both processes are interested in entering, then 1 condition is false for one and true for the other process
  - Progress requirement is satisfied
    - a process wanting to enter will be able to do so at some point
  - Bounded-waiting requirement is met
    - eventually it will be $P_i$’s turn if $P_i$ wants to enter
Multiple Processes

- The problem with Peterson’s solution is that it ONLY works for 2 process solutions
- How do we extend for multiple processes?
- Is there a way to modify Peterson’s approach?
- Consider the following:

```c
int turn;
int flag[N];  /* all set to FALSE initially */

enterCS(int myid) {
    otherid = (myid+1) % N;
    turn = otherid;
    flag[myid] = TRUE;
    while (turn == otherid && flag[otherid] == TRUE) ;
    /* proceed if turn == myid or flag[otherid] == FALSE */
}
leave_CS(int myid) {
    flag[myid] = FALSE;
}
```

- Does it work?
Bakery Algorithm (Leslie Lamport, 1974)

- We need to enforce a sequence in some manner that everyone will follow and contribute to making progress.

- Think about a bakery ... Hmm

Notation: \((a,b) < (c,d)\) if \(a < c\) or \(a = c\) and \(b < d\)

Every process has a unique id (integer) \(P_i\)

bool choosing[0..n-1]; /* all set to FALSE */
int number[0..n-1]; /

```c
enter_CS(myid) {
    choosing[myid] = TRUE;
    number[myid] =
        max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[myid] = FALSE;
    for (j=0 to n-1) {
        while (choosing[j])
            ;
        while (number[j] != 0)
            && ((number[j], P_j) < (number[myid], myid))
            ;
    }
}

leave_CS(myid) {
    number[myid] = 0;
}
```
Evaluation of the Bakery Algorithm

- Does it work?
- What do you need to prove?
- Show that it meets
  - Mutual exclusion
  - Progress
  - Bounded waiting requirements
- Need to know that maximum # processes
Looking to Hardware

- Complications arose because we had atomicity only at the granularity of a machine instruction
  - What a machine instruction could do is (was) limited

- Can we provide specialized instructions in hardware to provide additional functionality (with an instruction still being atomic)?
  - Looking again to hardware to help solve a OS problem

- Many systems (now) provide hardware support for implementing the critical section code

- All solutions below based on idea of locking
  - Protecting critical regions via locks
Synchronization Hardware

- Uniprocessors – could disable interrupts
  - Currently running code executes without preemption
  - Generally too inefficient on multiprocessor systems
    - operating systems using this not broadly scalable

- Modern CPUs provide atomic hardware instructions (2 general types)
  - Test-and-Set
    - test memory word and set value
  - Compare-and-Swap
    - swap contents of 2 memory words

- How does this help?
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
**test_and_set Instruction**

- **Definition:**
  ```java
  boolean test_and_set (boolean *target) {
      boolean rv = *target;
      *target = TRUE;
      return rv;
  }
  ```

- **Executed atomically**
- **Returns the original value of passed parameter**
- **Set the new value of passed parameter to TRUE**
Solution using test_and_set()

- Shared boolean variable lock, initialized to FALSE

Solution:

```c
    do {
        while (test_and_set(&lock))
          ; /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
    } while (true);
```
**compare_and_swap Instruction**

- **Definition:**

  ```c
  int compare_and_swap(int *value, int expected, int newvalue) {
      int temp = *value;
      if (*value == expected) {
          *value = newvalue;
          return temp;
      }
  }
  ```

- Executes atomically
- Returns the original value of passed parameter `value`
- Set the variable `value` to the value of the passed parameter `newvalue`
  - Only if `value == expected`
- That is, the swap takes place only under this condition
Solution using `compare_and_swap`

- Shared integer `lock` initialized to 0;
- Solution:
  ```c
  do {
    while(compare_and_swap(&lock, 0, 1) != 0)  
      ; /* do nothing */
  /* critical section */
  lock = 0;
  /* remainder section */
  } while (true);
  ```
Bounded-waiting Mutex with test_and_set

do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = test_and_set(&lock);
    waiting[i] = false;
    /* critical section */
    j = (i + 1) % n;
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = false;
    else
        waiting[j] = false;
    /* remainder section */
} while (true);
Spinning vs. Blocking

- In the previous solutions, we are spinning (busy-waiting) for some condition to change
  - This change should be effected by some other process
  - We are “presuming” that this other process will eventually get the CPU
    - some kind of preemptive scheduler
- This can be inefficient because:
  - You are wasting your time quantum spinning
  - Sometimes, your programs may not work!
    - suppose if the OS scheduler is not preemptive
Blocking Approaches

- If instead of busy-waiting, the process relinquishes the CPU at the time when it cannot proceed, the process is said to block
  - It is still wanting to get into the critical section
  - It is put in the blocked queue.
- It is the job of the process changing the condition to wake up a blocked process
  - Moves it from blocked back to ready queue
- Advantage:
  - Do not unnecessarily occupy CPU cycles
- Disadvantages?
Enter_CS(L) {
    Disable Interrupts
    Check if anyone is using L
    If not {
        Set L to being used
    }
    else {
        Move this PCB to Blocked queue for L
        Select another process to run from Ready queue
        Context switch to that process
    }
    Enable Interrupts
}

Exit_CS(L) {
    Disable Interrupts
    Check if blocked queue for L is empty
    if so {
        Set L to free
    }
    else {
        Move PCB from head of Blocked queue of L to Ready queue
    }
    Enable Interrupts
}

NOTE: These are OS system calls!
Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock (mutex for “mutual exclusion”)
- Protect a critical section by first acquire() a lock then release() the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a spinlock
acquire() and release()

```c
acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}
release() {
    available = true;
}
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (true);
```
Synchronization Constructs

- Synchronization requires more than just exclusion
  - If printer queue is full, I need to wait until there is at least 1 empty slot

- Note that `acquire()` / `release()` are not very suitable to implement such synchronization

- We need constructs to enforce orderings
  - A should be done after B
Semaphore (Dijkstra)

- Synchronization tool that provides more sophisticated ways (than mutex locks) for process to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()` (originally called `P()` and `V()` by Dijkstra)
- Definition of the `wait()` operation
  ```
  wait(S) {
    while (S <= 0) ; // busy wait
    S--; 
  }
  ```
- Definition of the `signal()` operation
  ```
  signal(S) { 
    S++; 
  }
  ```
Semaphore Usage

- **Counting semaphore**
  - Integer value can range over an unrestricted domain

- **Binary semaphore**
  - Integer value can range only between 0 and 1
  - Same as a mutex lock

- Can solve various synchronization problems with semaphores

- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$
  - Create a semaphore “synch” initialized to 0
    - $P_1$:
      ```
      S1;
      signal(synch);
      ```
    - $P_2$:
      ```
      wait(synch);
      S2;
      ```

- Can implement a counting semaphore S as a binary semaphore
Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` ...
  - ... on the same semaphore ...
  - ... at the same time

- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section
  - Could have busy waiting in critical section implementation
    - but implementation code is short
    - little busy waiting if critical section rarely occupied

- Note that applications may spend lots of time in critical sections and therefore this is not a particularly good solution
**Semaphores with no Busy waiting**

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - `block` – place the process invoking the operation on the appropriate waiting queue
  - `wakeup` – remove one of processes in the waiting queue and place it in the ready queue

```c
typedef struct {
    int value;
    struct process *list;
} semaphore;
```
Implementation with no Busy waiting

```c
wait(semaphore *S) {
    S->value--;  
    if (S->value < 0) {
        add this process to S->list;   
        block();
    }
}

signal(semaphore *S) {
    S->value++;  
    if (S->value <= 0) {
        remove a process P from S->list;  
        wakeup(P);
    }
}
```

NOTE: These are OS system calls, and there is no atomicity lost during the execution of these routines (interrupts are disabled).
Deadlock and Starvation

- **Definition**: A *deadlock* situation occurs if 2 or more processes are waiting indefinitely for an event that only one of the waiting processes can cause.

- Let S and Q be two semaphores initialized to 1

  
  \[
  \begin{align*}
  &P_0 & P_1 \\
  &wait(S); & wait(Q); \\
  &wait(Q); & wait(S); \\
  &... & ... \\
  &signal(S); & signal(Q); \\
  &signal(Q); & signal(S); \\
  \end{align*}
  \]

- **Starvation (indefinite blocking)**
  - A process may never be removed from the semaphore queue in which it is suspended.

- **Priority inversion**
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process.
  - Solved via priority-inheritance protocol.
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
Bounded-Buffer Problem

- $n$ buffers, each can hold one item
- Semaphore $mutex$ initialized to the value 1
- Semaphore $full$ initialized to the value 0
- Semaphore $empty$ initialized to the value $n$
Bounded Buffer Problem – Producer

- The structure of the producer process

```c
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```
Bounded Buffer Problem – Consumer

- The structure of the consumer process
  
  ```
  do {
      wait(full);
      wait(mutex);
      ...
      /* remove an item from buffer next_consumed */
      ...
      signal(mutex);
      signal(empty);
      ...
      /* consume the item in next_consumed */
      ...
  } while (true);
  ```
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - *Readers* – only read the data set (does not perform any updates)
  - *Writers* – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one writer can access the shared data at the same time
- Several variations of how readers and writers are considered
  - All involve some form of priorities
- Shared data
  - Data set
  - Semaphore *rw_mutex* initialized to 1
  - Semaphore *mutex* initialized to 1
  - Integer *read_count* initialized to 0
Readers-Writers Problem – Writer

The structure of a writer process

```c
    do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
    } while (true);
```
Readers-Writers Problem – Reader

- The structure of a reader process
  ```c
  do {
    wait(mutex);
    read_count++;
    if (read_count == 1) {
      wait(rw_mutex);
      signal(mutex);
    }
    …
    /* reading is performed */
    …
    wait(mutex);
    read_count--;
    if (read_count == 0) {
      signal(rw_mutex)
      signal(mutex)
    }
  } while (true);
  ```
Readers-Writers Problem Variations

- First variation – no reader kept waiting unless writer has permission to use shared object
- Second variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks
Dining Philosophers Problem (Dijkstra)

- Philosophers spend their lives alternating thinking and eating.
- Philosophers do not interact with their neighbors, but occasionally try to pick up 2 chopsticks (one at a time) to eat.
  - Need both to eat, then release both when done.
- In the case of 5 philosophers:
  - Shared data:
    - bowl of rice (data set)
Dining Philosophers Problem Algorithm

- The structure of Philosopher $i$:
  
  ```
  do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );
    //   eat
    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
    //   think
  } while (TRUE);
  ```

- What is the problem with this algorithm?
Preventing Starving Philosophers

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available
    - Picking must be done in a critical section
  - Use an asymmetric solution
    - An odd-numbered philosopher picks up first the left chopstick and then the right chopstick
    - Even-numbered philosopher picks up first the right chopstick and then the left chopstick
Problems with Semaphores

- Incorrect use of semaphore operations:
  - `signal (mutex) .... wait (mutex)`
  - `wait (mutex) ... wait (mutex)`
  - Omitting of `wait (mutex)` or `signal (mutex)` (or both)

- Deadlock and starvation are possible
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Abstract data type
  - Internal variables only accessible by code within the procedure
  - Routines (operations) that operate on the internal variables (shared)
  - External world only sees these operations (not the shared data or how the operations and synchronization are implemented)
- Only one process may be active within the monitor at a time
  - All the processes that are executing monitor code, there can be at most 1 process in ready queue (rest are either blocked or not in monitor!)

```plaintext
monitor monitor-name {
    // shared variable declarations
    procedure P1 (...) { .... }
    procedure Pn (...) {......}
    Initialization code (...) { ... }
}
```
Schematic view of a Monitor
Condition Variables

- condition x, y;

- Two operations are allowed on a condition variable:
  - x.wait() – a process that invokes the operation is suspended until x.signal()
  - x.signal() – resumes one of processes (if any) that invoked x.wait()
    - if no x.wait() on the variable, then it has no effect on the variable

- NOTE: If the signal comes before the wait, the signal gets lost!!! – You need to be careful since signals are not stored unlike semaphores
Monitor with Condition Variables

- Shared data
- Queues associated with x, y conditions
- Operations
- Initialization code
- Entry queue
**Condition Variables Choices**

- If process $P$ invokes $x.signal()$, and process $Q$ is suspended in $x.wait()$, what should happen next?
  - Both $Q$ and $P$ cannot execute in parallel
  - If $Q$ is resumed, then $P$ must wait

- Options include
  - *Signal and wait* – $P$ waits until $Q$ either leaves the monitor or it waits for another condition
  - *Signal and continue* – $Q$ waits until $P$ either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Java, C#, Concurrent Pascal, …
Monitor Solution to Dining Philosophers

```c
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING) state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```
Monitor Solution to Dining Philosophers

```c
void test (int i) {
    if ((state[(i + 4) % 5] != EATING)
        && (state[i] == HUNGRY)
        && state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {  
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```
What does each Philosopher do?

- Each philosopher $i$ invokes the operations `pickup()` and `putdown()` in the following sequence:

  ```
  DiningPhilosophers.pickup(i);
  EAT
  DiningPhilosophers.putdown(i);
  ```

- No deadlock, but starvation is possible
If several processes queued on condition \( x \), and \( x.signal() \) executed, which should be resumed?

- FCFS frequently not adequate
- conditional-wait construct of the form \( x.wait(c) \)
  - Where \( c \) is priority number
  - Process with lowest number (highest priority) is scheduled next
Next Class

- Deadlocks