FAQ

- Mutex vs Semaphore
  - Mutex is binary, semaphore is more general
- Hardware (ISA) implementation of atomic instructions
  - needed to acquiring a “lock” (a variable)
  - Examples: test_and_set(&lock), Swap(&lock, &key)
- Atomic instruction in x86?
  - LOCK instruction prefix, which applies to an instruction which does a read-modify-write on memory (INC, XCHG, CMPXCHG etc) makes read-modify-write atomic
- In RISK processors? Instruction-pairs
  - LL (Load Linked Word), SC (Store Conditional Word) in MIPS
  - LDREX, STREX in ARM
- What is both processes are being nice? turn = j;
  - Store is presumed to be atomic in Peterson’s solution
Process Synchronization

EW Dijkstra *Go To Statement Considered Harmful*
Semaphores by 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 $V()$ based on Dutch words
- Definition of the **wait() operation**
  ```
  wait(S) {
      while (S <= 0) // busy wait
          S--;
  }
  ```
- Definition of the **signal() operation**
  ```
  signal(S) {
      S++;
  }
  ```

Waits until another process makes $S=1$

Binary semaphore: When $s$ is 0 or 1, it is a mutex lock
**Wait(S) and Signal (S)**

- **Process 0**
  - Wait(S)
  - Critical section
  - Signal (S)

- **Semaphore S**
  - S = 1
  - S = 0
  - Locked by Process 1

- **Process 1**
  - Wait (S)
  - Busy waiting
  - Gets lock, S -
  - Critical section
  - Signal (S)

- S = 1
  - Locked by Process 1
acquire() and release()

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Semaphore S</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Critical section</td>
<td>0</td>
<td>wait ( ), busy waiting</td>
</tr>
<tr>
<td>Signal ( ) S++</td>
<td>1</td>
<td>Waiting, finished</td>
</tr>
<tr>
<td>..</td>
<td>0</td>
<td>S- -</td>
</tr>
<tr>
<td>Wait( )</td>
<td>0</td>
<td>Critical section</td>
</tr>
<tr>
<td>Wait( )</td>
<td>0</td>
<td>Critical section</td>
</tr>
<tr>
<td>Waiting finished</td>
<td>1</td>
<td>Signal ( ) S++</td>
</tr>
<tr>
<td>S--</td>
<td>0</td>
<td>..</td>
</tr>
<tr>
<td>Critical section</td>
<td>0</td>
<td>..</td>
</tr>
</tbody>
</table>
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

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

  $P_2$:
  
  ```
  wait(synch);
  S_2;
  ```

• Can implement a counting semaphore $S$ as a binary semaphore
The counting semaphore

• Controls access to a finite set of resources
• Initialized to the number of resources
• Usage:
  – Wait (S): to use a resource
  – Signal (S): to release a resource
• When all resources are being used: $S == 0$
  – Block until $S > 0$ to use the resource
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 now 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 good solution

• Alternative: block and wakeup (next slide)
Semaphore Implementation 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 (Cont.)

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);
    }
}

typedef struct{
    int value;
    struct process *list;
} semaphore;

If value < 0 
abs(value) is the number of waiting processes
**Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Let $s$ and $q$ be two semaphores initialized to 1

\[ P_0 \]
\[
\text{wait}(S); \\
\text{wait}(Q); \\
\ldots \\
\text{signal}(S); \\
\text{signal}(Q);
\]

\[ P_1 \]
\[
\text{wait}(Q); \\
\text{wait}(S); \\
\ldots \\
\text{signal}(Q); \\
\text{signal}(S);
\]

- **P0 executes wait(s), P1 executes wait(Q)**
  - P0 must wait till P1 executes signal(Q)
  - P1 must wait till P0 executes signal(S)  Deadlock!
Priority Inversion

• **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

• Solved via **priority-inheritance protocol**
  – Process accessing resource needed by higher priority process
    Inherits higher priority till it finishes resource use
  – Once done, process reverts to lower priority
Classical Problems of Synchronization

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

• Monitors
Bounded-Buffer Problem

- $n$ buffers, each can hold one item
- Binary semaphore (mutex)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - empty: Number of empty slots available
    - Initialized to $n$
  - full: Number of filled slots available $n$
    - Initialized to 0
Bounded-Buffer: Note

• Producer and consumer must be ready before they attempt to enter critical section

• Producer readiness?
  – When a slot is available to add produced item
    • wait(empty)
      – empty is initialized to n

• Consumer readiness?
  – When a producer has added new item to the buffer
    • wait(full)
      – full initialized to 0
The structure of the producer process

do {
  
  ...  
  /* produce an item in next_produced */
  ...
  wait(empty);                          wait till slot available
  wait(mutex);                           Allow producer OR consumer to (re)enter critical section
  ...  
  /* add next produced to the buffer */
  ...
  signal(mutex);                          Allow producer OR consumer to (re)enter critical section
  signal(full);                           signal consumer that a slot is available
} while (true);
The structure of the consumer process

Do {
    wait(full);  wait till slot available for consumption
    wait(mutex); Only producer OR consumer can be in critical section
    ...  
    /* remove an item from buffer to next_consumed */  
    ...  
    signal(mutex);  Allow producer OR consumer to (re)enter critical section
    signal(empty);  signal producer that a slot is available to add
    ...  
    /* 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; they do not perform any updates
  – Writers – can both read and write

• Problem – allow multiple readers to read at the same time
  – Only one single 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 (mutual exclusion for writer)
  – Semaphore `mutex` initialized to 1 (mutual exclusion for read_count)
  – Integer `read_count` initialized to 0 (how many readers?)
The structure of a writer process

```c
    do {
        wait(rw_mutex);
        ...
        /* writing is performed */
        ...
        signal(rw_mutex);
    } while (true);
```

When: writer in critical section and if n readers waiting:
- 1 reader is queued on rw_mutex
- (n-1) readers queued on mutex
Readers-Writers Problem (Cont.)

- The structure of a reader process
  
  ```
  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);
  ```

  **mutex for mutual exclusion to readcount**

  **When:**
  - writer in critical section
  - and if n readers waiting
    - 1 is queued on rw_mutex
    - (n-1) queued on mutex
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
Philosophers spend their lives alternating thinking and eating

Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  – Need both to eat,
  – then release both when done

Each chopstick is a semaphore
  – Grab by executing wait ( )
  – Release by executing signal ( )

Shared data
  • Bowl of rice (data set)
  • Semaphore chopstick [5] initialized to 1
Dining-Philosophers Problem

Plato, Confucius, Socrates, Voltaire and Descartes
• 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?
  – If all of them pick up the the left chopstick first - Deadlock
• Deadlock handling
  – Allow at most 4 philosophers to be sitting simultaneously at the table (with the same 5 forks).
  – 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.
FAQ

• Classes that follow CS370
  – CS455 Distributed Systems  Spring
  – CS457 Networks  Fall
  – CS470 Computer Architecture  Spring
  – CS475 Parallel Programming  Fall
  – CS435: Introduction to Big Data  Spring
FAQ

• Why not use a Boolean variable instead of Mutex?

• If there are more than one processes waiting, will the semaphore value be negative?
  – Negative: number of processes/threads waiting
  – 0: no waiting threads
  – Positive: no waiting threads, a wait operation would not put in queue the invoking thread. Often +1

• How to keep a philosopher from starving?
  – There exist solutions that will avoid a deadlock. However they may allow starvation, unless solution is further refined.

• Why not give each philosopher 2 chopsticks?
  – Nice and elegant solution. Widely used in Chinese restaurants. But takes all the fun away from the problem.
FAQ

• Producer-consumer with bounded buffer
  – Should the production and consumption rates be a perfect match?
  – Can the producer add more than 1 item at a time?

• Monitors: what are they and how to implement them.
  – Details coming up.
Classical Problems of Synchronization

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

• Monitors
Problems with Semaphores

• Incorrect use of semaphore operations:
  – Omitting of wait (mutex)
    • Violation of mutual exclusion
  – or signal (mutex)
    • Deadlock!
Monitors

- Monitor: 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
- Only one process may be active within the monitor at a time
  - Automatically provide mutual exclusion
- Originally proposed for Concurrent Pascal 1975
- Directly supported by Java but not C
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {......}

    Initialization code (...) { ... }
}
}
Schematic view of a Monitor

Only one process/thread in the Monitor

Provides an easy way to achieve mutual exclusion

But ... we also need a way for processes to **block** when they cannot proceed
Condition Variables

The \textit{condition} construct

- condition \texttt{x, y};

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

Compare with semaphore
Difference between the signal() in semaphores and monitors

- **Condition variables in Monitors:** Not persistent
  - If a signal is performed and no waiting threads?
    - Signal is simply ignored
  - During subsequent wait operations
    - Thread blocks

- **Semaphores**
  - Signal increments semaphore value even if there are no waiting threads
    - Future wait operations would immediately succeed!
Monitor with Condition Variables

- Shared data
  - Queues associated with x, y conditions
  - Operations
  - Initialization code

Entry queue
If process P invokes \texttt{x.signal()}, and process Q is suspended in \texttt{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 *Concurrent Pascal* (’75) compromise
  - P executing signal immediately leaves the monitor, Q is resumed.
- Implemented in other languages including C#, Java.
Monitor Solution to Dining Philosophers: Deadlock-free

enum {THINKING, HUNGRY, EATING} state[5];

• state[i] = EATING only if
  – state[(i+4)%5] != EATING && state[(i+1)%5] != EATING

• condition self[5]
  – Delay self when **HUNGRY but unable** to get chopsticks

**Sequence of actions**

• Before eating, must invoke pickup()
  – May result in suspension of philosopher process
  – After completion of operation, philosopher may eat

    think
    DiningPhilosophers.pickup(i);
    eat
    DiningPhilosophers.putdown(i);
    think
enum {THINKING, HUNGRY, EATING} state[5];
monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING} state [5];
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i); //on next slide
        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);
    }
}

Suspend self if unable to acquire chopstick
Check to see if person on left or right can use the chopstick
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;
}
• Philosopher $i$ can starve if eating periods of philosophers on left and right overlap
• Possible solution
  – Introduce new state: STARVING
  – Chopsticks can be picked up if no neighbor is starving
    • Effectively wait for neighbor’s neighbor to stop eating
    • REDUCES concurrency!