FAQ

• Why are critical sections important?
  – Correctness, data corruption

• Can’t critical sections cause starvation?
  – Not if they satisfy..

• What happens if only one program is running with its critical section, and it is nice. Will it get stuck?

• Two processes do not share any resources, do they need critical sections?

• Bounded buffer: Problem only arises if the producer is faster?

• Are critical sections for two interacting processes the same length?
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
  
  **P1:**
  
  ```
  S_1;
  signal(synch);
  ```

  **P2:**
  
  ```
  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)
Implementation with no Busy waiting (Cont.)

```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);
    }
}
```

If value < 0, abs(value) is the number of waiting processes

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

Colorado State University
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
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
The condition construct

- 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

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 `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 *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

class DiningPhilosophers {
    public void think()
    public void eat()
    public void pickup(int i)
    public void putdown(int i)
}

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 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
test() to see if philosopher I can eat

```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;
}
```

Eat only if HUNGRY and Person on Left AND Right are not eating

Signal a process that was suspended while trying to eat
Possibility of starvation

• 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!
Monitor Implementation Mutual Exclusion

For each monitor

- Semaphore mutex initialized to 1
- Process must execute
  - `wait(mutex)`: Before entering the monitor
  - `signal(mutex)`: Before leaving the monitor
Monitor Implementation Using Semaphores

• Variables

```
semaphore mutex;     // (initially = 1) allows only one process to be active
semaphore next;     // (initially = 0) causes signaler to sleep
int next_count = 0;  // num of sleepers since they signalled
```

• Each procedure $F$ will be replaced the compiler by

```
wait(mutex);
...

    body of F;
    ...

if (next_count > 0)
    signal(next)
else
    signal(mutex);
```

• Mutual exclusion within a monitor is ensured
Monitor Implementation – Condition Variables

- For each condition variable \( x \), we have:

  ```
  semaphore x_sem; // (initially = 0) causes caller of wait to sleep
  int x_count = 0; // number of sleepers on condition
  ```

- The operations `x.wait` and `x.signal` can be implemented as:

<table>
<thead>
<tr>
<th>The operation x.wait can be implemented as:</th>
<th>The operation x.signal can be implemented as:</th>
</tr>
</thead>
</table>
  | `x_count++;
  if (next_count > 0)
  signal(next);
  else
  signal(mutex);
  wait(x_sem);
  x_count--;` | `if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
  }` |
Resuming Processes within a Monitor

• If several processes queued on condition \( x \), and \( x.\text{signal}() \) is executed, which should be resumed?
• FCFS frequently not adequate
• **conditional-wait** construct of the form \( x.\text{wait}(c) \)
  – Where \( c \) is **priority number**
  – Process with lowest number (highest priority) is scheduled next
Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

  ```
  R.acquire(t);
  ...
  access the resource;
  ...
  
  R.release;
  ```

- Where R is an instance of type ResourceAllocator
A Monitor to Allocate Single Resource

```java
monitor ResourceAllocator
{
    boolean busy;
    condition x;

    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }

    void release() {
        busy = FALSE;
        x.signal();
    }

    initialization code() {
        busy = FALSE;
    }
}
```

- **Sleep, Time used to prioritize waiting processes**
- **Wakes up one of the processes**
Java Synchronization

• For simple synchronization Java provides the synchronized keyword
  – synchronizing methods
    public synchronized void increment() { c++; }
  – synchronizing blocks
    synchronized(this) {
      lastName = name;
      nameCount++;
    }

• wait() and notify() allows a thread to wait for an event. A call to notify. all() allows all threads that are on wait() with the same lock to be released

• For more sophisticated locking mechanisms, starting from Java 5, the package java.concurrent.locks provides additional locking
Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads
Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses condition variables
- Uses readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile
Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)
Linux Synchronization

• Linux:
  – Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  – Version 2.6 and later, fully preemptive

• Linux provides:
  – Semaphores
  – atomic integers
  – spinlocks
  – reader-writer versions of both

• On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption
Pthreads Synchronization

• Pthreads API is OS-independent
• It provides:
  – mutex locks
  – condition variable
• Non-portable extensions include:
  – read-write locks
  – spinlocks
Alternative Approaches

• Transactional Memory

• OpenMP

• Functional Programming Languages
A memory transaction is a sequence of read-write operations to memory that are performed atomically.

```c
void update()
{
    /* read/write memory */
}
```
• OpenMP is a set of compiler directives and API that support parallel programming.

```c
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.