Lecture 7:
CVs & Scheduling

CSE 120: Principles of Operating Systems
Alex C. Snoeren

HW 2 Due 10/17
Monitors

- A monitor is a programming language construct that controls access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Why is this an advantage?
- A monitor is a module that encapsulates
  - Shared data structures
  - Procedures that operate on the shared data structures
  - Synchronization between concurrent threads that invoke the procedures
- A monitor protects its data from unstructured access
- It guarantees that threads accessing its data through its procedures interact only in legitimate ways
Monitor Semantics

- A monitor guarantees mutual exclusion
  - Only one thread can execute any monitor procedure at any time (the thread is “in the monitor”)
  - If a second thread invokes a monitor procedure when a first thread is already executing one, it blocks
    » So the monitor has to have a wait queue…
  - If a thread within a monitor blocks, another one can enter

- What are the implications in terms of parallelism in monitor?
Account Example

Monitor `account` {
    double balance;
    double withdraw(amount) {
        balance = balance – amount;
        return balance;
    }
}

 Threads block waiting to get into monitor
withdraw(amount)
balance = balance – amount;
withdraw(amount)
return balance (and exit)
withdraw(amount)
balance = balance – amount
return balance;
withdraw(amount)
balance = balance – amount
return balance;

When first thread exits, another can enter. Which one is undefined.

- Hey, that was easy
- But what if a thread wants to wait inside the monitor?
  » Such as “mutex(empty)” by reader in bounded buffer?
Condition Variables

- Condition variables provide a mechanism to wait for events (a “rendezvous point”)
  - Resource available, no more writers, etc.
- Condition variables support three operations:
  - **Wait** – release monitor lock, wait for C/V to be signaled
    » So condition variables have wait queues, too
  - **Signal** – wakeup one waiting thread
  - **Broadcast** – wakeup all waiting threads
- **Note:** Condition variables are not boolean objects
  - “if (condition_variable) then” … does not make sense
  - “if (num_resources == 0) then wait(resources_available)” does
  - An example will make this more clear
Monitor Bounded Buffer

Monitor `bounded_buffer` {
    Resource buffer[N];
    // Variables for indexing buffer
    Condition not_full, not_empty;

    void put_resource (Resource R) {
        while (buffer array is full)
            wait(not_full);
        Add R to buffer array;
        signal(not_empty);
    }

    Resource get_resource() {
        while (buffer array is empty)
            wait(not_empty);
        Get resource R from buffer array;
        signal(not_full);
        return R;
    }
} // end monitor

◆ What happens if no threads are waiting when signal is called?
Monitor Queues

Monitor `bounded_buffer` {

Condition `not_full`;
…`other variables`…
Condition `not_empty`;

void `put_resource` () {
  …`wait(not_full)`…
  …`signal(not_empty)`…
}

Resource `get_resource` () {
  …
}
}

Waiting to enter
Waiting on condition variables
Executing inside the monitor
Condition Vars != Semaphores

- Condition variables != semaphores
  - Although their operations have the same names, they have entirely different semantics (such is life, worse yet to come)
  - However, they each can be used to implement the other

- Access to the monitor is controlled by a lock
  - `wait()` blocks the calling thread, and gives up the lock
    - To call `wait`, the thread has to be in the monitor (hence has lock)
    - `Semaphore::wait` just blocks the thread on the queue
  - `signal()` causes a waiting thread to wake up
    - If there is no waiting thread, the signal is lost
    - `Semaphore::signal` increases the semaphore count, allowing future entry even if no thread is waiting
    - Condition variables have no history
There are two flavors of monitors that differ in the scheduling semantics of signal()

- **Hoare** monitors (original)
  - signal() immediately switches from the caller to a waiting thread
  - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
  - Signaler must restore monitor invariants before signaling

- **Mesa** monitors (Mesa, Java)
  - signal() places a waiter on the ready queue, but signaler continues inside monitor
  - Condition is not necessarily true when waiter runs again
    - Returning from wait() is only a hint that something changed
    - Must recheck conditional case
Hoare vs. Mesa Monitors

- **Hoare**
  ```
  if (empty)
    wait(condition);
  ```

- **Mesa**
  ```
  while (empty)
    wait(condition);
  ```

- **Tradeoffs**
  - Mesa monitors easier to use, more efficient
    - Fewer context switches, easy to support broadcast
  - Hoare monitors leave less to chance
    - Easier to reason about the program
Condition Vars & Locks

- Condition variables are also used without monitors in conjunction with blocking locks
  - This is what you are implementing in Project 1
- A monitor is “just like” a module whose state includes a condition variable and a lock
  - Difference is syntactic; with monitors, compiler adds the code
- It is “just as if” each procedure in the module calls acquire() on entry and release() on exit
  - But can be done anywhere in procedure, at finer granularity
- With condition variables, the module methods may wait and signal on independent conditions
Using Cond Vars & Locks

- Alternation of two threads (ping-pong)
- Each executes the following:

```c
Lock lock;
Condition cond;

void ping_pong () {
    acquire(lock);
    while (1) {
        printf("ping or pong\n");
        signal(cond, lock);
        wait(cond, lock);
    }
    release(lock);
}
```

- Must acquire lock before you can wait (similar to needing interrupts disabled to call Sleep in Nachos)
- Wait atomically releases lock and blocks until signal()
- Wait atomically acquires lock before it returns
Monitors and Java

- A lock and condition variable are in every Java object
  - No explicit classes for locks or condition variables
- Every object is/has a monitor
  - At most one thread can be inside an object’s monitor
  - A thread enters an object’s monitor by
    - Executing a method declared “synchronized”
      - Can mix synchronized/unsynchronized methods in same class
    - Executing the body of a “synchronized” statement
      - Supports finer-grained locking than an entire procedure
      - Identical to the Modula-2 “LOCK (m) DO” construct
- Every object can be treated as a condition variable
  - Object::notify() has similar semantics as Condition::signal()
Synchronization Summary

- **Semaphores**
  - `wait()/signal()` implement blocking mutual exclusion
  - Also used as atomic counters (counting semaphores)
  - Can be inconvenient to use

- **Monitors**
  - Synchronizes execution within procedures that manipulate encapsulated data shared among procedures
  - Only one thread can execute within a monitor at a time
  - Relies upon high-level language support

- **Condition variables**
  - Used by threads as a synchronization point to wait for events
  - Inside monitors, or outside with locks
In discussing process management and synchronization, we talked about context switching among processes/threads on the ready queue. But we have glossed over the details of exactly which thread is chosen from the ready queue. Making this decision is called scheduling. In this lecture, we’ll look at:

- The goals of scheduling
- Starvation
- Various well-known scheduling algorithms
- Standard Unix scheduling algorithm
Multiprogramming

- In a multiprogramming system, we try to increase CPU utilization and job throughput by overlapping I/O and CPU activities
  - Doing this requires a combination of mechanisms and policy
- We have covered the mechanisms
  - Context switching, how and when it happens
  - Process queues and process states
- Now we’ll look at the policies
  - Which process (thread) to run, for how long, etc.
- We’ll refer to schedulable entities as jobs (standard usage) – could be processes, threads, people, etc.
Scheduling Horizons

- Scheduling works at two levels in an operating system
  - To determine the multiprogramming level – the number of jobs loaded into primary memory
    » Moving jobs to/from memory is often called swapping
  - To decide what job to run next to guarantee “good service”
    » Good service could be one of many different criteria
- These decisions are known as long-term and short-term scheduling decisions, respectively
  - **Long-term** scheduling happens relatively infrequently
    » Significant overhead in swapping a process out to disk
  - **Short-term** scheduling happens relatively frequently
    » Want to minimize the overhead of scheduling
Scheduling Goals

- Scheduling algorithms can have many different goals:
  - CPU utilization
  - Job throughput (# jobs/unit time)
  - Turnaround time ($T_{\text{finish}} - T_{\text{start}}$)
  - Waiting time ($\text{Avg}(T_{\text{wait}})$: avg time spent on wait queues)
  - Response time ($\text{Avg}(T_{\text{ready}})$: avg time spent on ready queue)

- Batch systems
  - Strive for job throughput, turnaround time (supercomputers)

- Interactive systems
  - Strive to minimize response time for interactive jobs (PC)
Starvation

- **Starvation** occurs when a job cannot make progress because some other job has the resource it requires
  - We’ve seen locks, Monitors, Semaphores, etc.
  - The same thing can happen with the CPU!

- Starvation can be a side effect of synchronization
  - Constant supply of readers always blocks out writers
  - Well-written critical sections should ensure bounded waiting

- Starvation usually a side effect of the sched. algorithm
  - A high priority process always prevents a low priority process from running on the CPU
  - One thread always beats another when acquiring a lock
Scheduling

- The **scheduler** (aka dispatcher) is the module that manipulates the queues, moving jobs to and fro.
- The **scheduling algorithm** determines which jobs are chosen to run next and what queues they wait on.
- In general, the scheduler runs:
  - When a job switches states (running, waiting, etc.)
  - When an interrupt occurs
  - When a job is created or terminated
- We’ll discuss scheduling algorithms in two contexts:
  - A **preemptive** scheduler can interrupt a running job
  - A **non-preemptive** scheduler waits for running job to block.
FCFS/FIFO Algorithms

- First-come first-served (FCFS), first-in first-out (FIFO)
  - Jobs are scheduled in order of arrival to ready queue
  - “Real-world” scheduling of people in lines (e.g., supermarket)
  - Typically non-preemptive (no context switching at market)
  - Jobs treated equally, no starvation

- Problem
  - Average waiting time can be large if small jobs wait behind long ones (high turnaround time)
  - You have a basket, but you’re stuck behind someone with a cart
Shortest Job First (SJF)

- **Shortest Job First (SJF)**
  - Choose the job with the smallest expected CPU burst
    - Person with smallest number of items to buy
  - Provably optimal minimum average waiting time

- **Problem**
  - Impossible to know size of CPU burst
    - Like choosing person in line without looking inside basket/cart
  - How can you make a reasonable guess?
  - Can potentially starve

- **Flavors**
  - Can be either preemptive or non-preemptive
  - Preemptive SJF is called shortest remaining time first (SRTF)
Round Robin (RR)

- Round Robin
  - Excellent for timesharing
  - Ready queue is treated as a circular queue (FIFO)
  - Each job is given a time slice called a quantum
  - A job executes for the duration of the quantum, or until it blocks or is interrupted
  - No starvation
  - Can be preemptive or non-preemptive

- Problem
  - Context switches are frequent and need to be very fast
Project 1: Synchronization in Nachos

CSE 120: Principles of Operating Systems
Alex C. Snoeren
Locks & CVs

- **Lock issues**
  - A thread cannot Acquire a lock it already holds
  - A thread cannot Release a lock it does not hold
  - A lock cannot be deleted if a thread is holding it

- **Condition Variable issues**
  - A thread can only call Wait and Signal if it holds the mutex
  - Wait must Release the mutex before the thread sleeps
  - Wait must Acquire the mutex after the thread wakes up
  - A condition variable cannot be deleted if a thread is waiting on it
Mailboxes

- Senders and receivers need to be synchronized
  - One sender and one receiver need to rendezvous

- Issues
  - Block all other senders while waiting for receiver in Send
  - Block all other receivers while waiting for sender in Receive
  - When a condition variable is signaled...
    - The waiting thread is placed on the ready list
    - **But it has not necessarily re-acquired the lock**
    - It only reacquires the lock when it runs again
    - If another thread runs before it does, that thread can acquire the lock before the waiter does
    - Let’s look at an example
while (1) {
    mutex->Acquire();
    printf("ping\n");
    cond->Signal(mutex);
    mutex->Release();
}

while (1) {
    mutex->Acquire();
    cond->Wait(mutex);
    printf("pong\n");
    mutex->Release();
}

Signal places waiter on ready list, and then continues

BUT – the waiter now competes with the signaler to re-acquire the mutex

Output COULD be: ping…ping…ping
Interlocking with Wait/Signal

```c
Mutex *mutex;
Condition *cond;

void ping_pong () {
    mutex->Acquire();
    while (1) {
        printf("ping or pong\n");
        cond->Signal(mutex);
        cond->Wait(mutex);
    }
    mutex->Release();
}
```

Waiting after signaling **interlocks** the two threads.

The thread that signals then does a wait, and cannot proceed until the other thread wakes up from its wait and follows with a signal.
Thread::Join

- **Issues**
  - A thread can only be Joined if specified during creation
  - A thread can only be Joined after it has forked
  - Only one thread can call Join on another
  - A thread cannot call Join on itself
  - A thread should be able to call Join on a thread that has already terminated
    - This is the tricky part
    - Should delay deleting thread object if it is to be joined
      - If it is not going to be Joined, then don’t change how it is deleted
    - Where is it deleted now? Look for use of threadToBeDestroyed
    - Where should joined threads be deleted?
    - Need to delete synch primitives used by Join as well
Thread::setPriority(int)

Issues

- Priorities have the entire range of an “int”
  - Both negative and positive
- If one thread has a priority value that is greater than another, that thread has a higher priority (simple integer comparisons)
- List implementation in list.cc has sorting capabilities
- Only adjust priority of thread when it is placed on ready list
- When transferring priority from a high thread to a low thread, the transfer is only temporary
  - When the low thread releases the lock, its priority reverts
Mating Whales

Issues

- This is a synchronization problem like Bounded-Buffer and Readers/Writers
- You do not need to implement anything inside of Nachos
  - But you will use the synchronization primitives you implemented
  - You can use any synch primitives you want
- You will implement Male, Female, and Matchmaker as functions in threadtest.cc (or equivalent), and create and fork threads to execute these functions in ThreadTest:
  
  ```
  T1->Fork(Male, 0); // could fork many males
  T2->Fork(Female, 0); // could fork many females
  T3->Fork(Matchmaker, 0); // could fork many matchmakers
  ```

- There is no API -- we will compile, run, and visually examine your code for correctness
- Comments will help (both you and us)
Tips

- Use DEBUG macro to trace the interaction of the synchronization primitives and thread context switches
  - Run "nachos –d s –d t" to enable synch and thread debugs

- Good advice available on the Web:
  - Nachos Road Map → Experience With Nachos Assignments → Synchronization