Lecture 7: CVs & Scheduling

CSE 120: Principles of Operating Systems
Alex C. Snoeren

HW 2 Due 4/24 at 12:30
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
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
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
  - I/O 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
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
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
Ada-style Rendezvous

- 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 speak
  - Block all other receivers while waiting for sender in listen
  - 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
CSE 120 – Lecture 7: CVs and Scheduling

Synchronizing w/ Wait/Signal

```c
while (1) {
    mutex->Acquire();
    printf("ping\n");
    cond->Signal(mutex);
    mutex->Release();
}
```

Signal places waiter on ready list, and then continues

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

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

Output COULD be:

```
ping...ping...ping
```
Mutex *mutex;
Condition *cond;

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

Issues

- 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
  » Where is it deleted now? Look for use of threadToBeDestroyed
  » Recall Java is garbage collected
  » 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)
- 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
Hawaiian Boats

- 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 Child and Adult as thread classes, and create and fork threads to execute these functions
  - There is no API – you are expected to call AutoGrader functions when appropriate events occur
  - Comments will help (both you and us)