Logistics

• Project 1: Design Doc due Wednesday
  • Design reviews with TAs on Friday/Monday
• HW 1 Due 7/12
• Pintos Overview Session: Monday
  • 11am-1pm, Wozniak Lounge
CPU & I/O Bursts

Support interactive programs: prefer I/O-bound tasks
Recall: Evaluating Schedulers

- **Response Time** (ideally low)
  - What user sees: from keypress to character on screen

- **Throughput** (ideally high)
  - Total operations per second
  - Problem: Overhead (e.g. from context switching)

- **Fairness**
  - Conflicts with best avg. throughput/resp. time
Recall: First-Come First-Served

- Just run processes in order of arrival

- Convoy Effect: Short processes stuck behind long processes
Recall: Round Robin

• Give out small units of CPU time ("time quantum")

• Preempt a thread when its quantum expires

• Cycle through ready threads
Recall: Priority Scheduling

- Something gives jobs (processes) priority
  - User sets it explicitly
  - System manipulates priorities in pursuit of some policy

- Always run the \textbf{ready} thread with \textit{highest priority}
Recall: Shortest Job First & Shortest Remaining Time First

• **Provably Optimal** with respect to \( \text{Response Time} \)

• Key Idea: remove convoy effect
  • Short jobs always stay ahead of long ones
Recall: Multi-Level Feedback Scheduling

• Observe process behavior to approximate SRTF
• Starvation still possible without a workaround
• Basis for real OS schedulers, e.g. Linux O(1)
Linux Completely Fair Scheduler

• Goal: Each process gets an equal share of CPU

• $N$ threads "simultaneously" execute on $1/N^{th}$ of CPU

At any time $t$ we would observe:

```
| T_1 | T_2 | T_3 |
```

CPU Time

$t/N$
Linux Completely Fair Scheduler

- Can't do this with real hardware
  - Still need to give out full CPU in time slices
- Instead: track CPU time given to a thread so far

Scheduling Decision:
- "Repair" illusion of complete fairness
- Choose thread with minimum CPU time
Linux CFS

• Constraint 1: *Target Latency*
  • Period of time over which every process gets service
  • Preserves response time

• Target Latency: 20ms, 4 Processes
  • Each process gets 5ms time slice

• Target Latency: 20 ms, 200 Processes
  • Each process gets 0.1ms time slice
  • Recall Round-Robin: Huge context switching overhead
Linux CFS

• Constraint 2: Minimum Granularity
  • Minimum length of any time slice
  • Protects throughput

• Target Latency 20ms, Minimum Granularity 1ms, 200 processes
  • Each process gets 1ms time slice
Linux CFS

• What if we want to use `nice` to change priority?
• Key Idea: Assign a weight $w_i$ to each process $i$

• Originally: $Q = \text{Target Latency} \times \frac{1}{N}$
• Now: $Q_i = (w_i / \sum_p w_p) \times \text{Target Latency}$
Linux CFS

- Target Latency = 20ms,
- Minimum Granularity = 1ms
- Two CPU-Bound Threads
  - Thread A has weight 1
  - Thread B has weight 4
- Time slice for A? 4 ms
- Time slice for B? 16 ms
Linux CFS

- Track a thread's *virtual* runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly

![Diagram showing physical CPU time with two bars representing threads A and B, with A having a lower runtime value of 4 compared to B's 16.](image-url)
Linux CFS

- Track a thread's *virtual* runtime rather than its true physical runtime
- Higher weight: Virtual runtime increases more slowly
- Lower weight: Virtual runtime increases more quickly

Actually Used for Decisions
Real-Time Scheduling

• Goal: Guaranteed Performance
  • Meet deadlines even if it means being unfair or slow
  • Limit how bad the worst case is

• Hard real-time:
  • Meet all deadlines (if possible)
  • Ideally: determine in advance if this is possible
Real-Time Example

Preemptible tasks with known deadlines \((D)\) and known burst times \((C)\)
What if we try Round-Robin?
Earliest Deadline First

• Priority scheduling with preemption
• Priority proportional to time until deadline
• Example with periodic tasks:

$T_1 = (4,1)$

$T_2 = (5,2)$

$T_3 = (7,2)$
EDF: Feasibility Testing

• Even EDF won't work if you have too many tasks
• For $n$ tasks with computation time $C$ and deadline $D$, a feasible schedule exists if:

$$\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq 1$$
# Choosing the Right Scheduler

<table>
<thead>
<tr>
<th>I Care About:</th>
<th>Then Choose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Throughput</td>
<td>FCFS</td>
</tr>
<tr>
<td>Avg. Response Time</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>I/O Throughput</td>
<td>SRTF Approximation</td>
</tr>
<tr>
<td>Fairness (CPU Time)</td>
<td>Linux CFS</td>
</tr>
<tr>
<td>Fairness – Wait Time to Get</td>
<td>Round Robin</td>
</tr>
<tr>
<td>CPU</td>
<td></td>
</tr>
<tr>
<td>Meeting Deadlines</td>
<td>EDF</td>
</tr>
<tr>
<td>Favoring Important Tasks</td>
<td>Priority</td>
</tr>
</tbody>
</table>
Priority and Locks

At this point, which job does the scheduler choose?

Job 3 (Highest Priority)
Priority and Locks

Job 3 attempts to acquire lock held by Job 1
At this point, which job does the scheduler choose?
Job 2 (Medium Priority)
Priority Inversion
Solution: Priority Donation

Job 3 temporarily grants Job 1 its highest priority to run on its behalf
Priority and Locks

Priority 1
Priority 2
Priority 3

Job 2
Job 3

Job 3 acquires lock, runs again
Break
A Better Lock Implementation

• Interrupt-based solution is ok for single core
• But doesn't work well on multi-core machines

• Solution: Hardware support for atomic operations
Recall: Atomic Operations

• Definition: An operation runs to completion or not at all
• Foundation for synchronization primitives

• Example: Loading or storing a word
More Powerful Atomic Ops

- Atomic load/store not good enough to build a lock

- Instead: Hardware instructions that atomically read a value from (shared) memory and write a new value

- Hardware responsible for making this work in spite of caches
Read/Modify/Write Instructions

• test\&set (&address) {

    result = M[address]; // return result from “address” and
    M[address] = 1; // set value at “address” to 1

    return result;
}

• swap (&address, register) {

    temp = M[address]; // swap register’s value to
    M[address] = register; // value at “address”

    register = temp;
}

• compare\&swap (&address, reg1, reg2) {

    if (reg1 == M[address]) { // If memory still == reg1,
        M[address] = reg2; // then put reg2 => memory
        return success;
    } else { // Otherwise do not change memory

        return failure;
    }
}


Locks with Test & Set

```c
int value = 0; // Free

Acquire() {
    while (test&set(value)); {
        // Do nothing
    }
}

Release() {
    value = 0;
}
```

• **Lock Free:** test&set reads 0, sets value to 1, returns 0 (old value), acquires

• **Lock Busy:** test&set reads 1, sets value to 1, returns 1 (old value), repeats

Remember:
```c
test&set (&address) {
    result = M[address];
    M[address] = 1;
    return result;
}
```
Locks with Test & Set

```java
int value = 0; // Free

Acquire() {
    while (test&set(value)); {
        // Do nothing
    }
}

Release() {
    value = 0;
}
```

- **Busy Waiting**: Consumes CPU time while waiting
  - Keeps other threads from using CPU
  - Maybe even the thread holding the lock
- These are known as spin locks
Memory Traffic

- **test&set** requires communication among CPU cores

- **Why? Caching**
  - Local cache needs to "own" memory address to write to it

- So something like `while(test&set(...) );` generates a lot of communication
Test, Then Test&Set

int value = 0; // Free
Acquire() {
  do {
    do {
      while(value);   // Wait until might be free
    } while(test&set(&value)); // exit if get lock
  }
}

Release() {
  value = 0;
}

• while(value) reads from local cache only
  • Hardware knows that this value is shared with others
  • Wait for cache invalidation from lock holder
• Still busy waiting, but fewer test&set operations
Recall: Using Interrupts

- Idea: Disable interrupts for **mutual exclusion** on accesses to value indicating lock status

```c
Acquire() {
    disable interrupts;
    if (value == BUSY) {
        put thread on wait queue;
        run_new_thread()
        // Enable interrupts?
    } else {
        value = BUSY;
    }
    enable interrupts;
}

Release() {
    disable interrupts;
    if (anyone waiting) {
        take a thread off queue;
    } else {
        Value = FREE;
    }
    enable interrupts;
}
```
Locks with test&set

- Use spin locks rather than interrupts to build the "real" locks everyone will use

```c
int guard = 0;
int value = 0;

Acquire() {
    // Short busy-wait time
    while (test&set(guard));
    if (value == 1) {
        put thread on wait-queue;
        go to sleep()& guard = 0;
    } else {
        value = 1;
        guard = 0;
    }
}

Release() {
    // Short busy-wait time
    while (test&set(guard));
    if anyone on wait queue {
        take thread off wait-queue
        Place on ready queue;
    } else {
        value = 0;
    }
    guard = 0;
}
```
Semaphore

• Another tool for synchronization (generalized lock)

• **Definition**: A non-negative integer value with two possible operations
  - **P()** or **down()** or **wait()**: *atomically* wait for semaphore to become positive, then decrement it by 1
  - **V()** or **up()** or **signal()**: *atomically* increment semaphore (waking up a P() thread)
Semaphores Like Integers But...

- **Cannot read/write value directly**
  - `down()` and `up()` only
  - Except when initializing semaphore

- **Never negative:** -- something waits instead
  - Two down operations can't go below 0, some thread wins, and the other blocks (atomic)
Simple Semaphore Patterns

- **Mutual Exclusion**: (Same as lock)
  - Called a "binary semaphore"
    
    ```
    initial value of semaphore = 1;
    semaphore.P();
    // Critical section goes here
    semaphore.V();
    ```

- **Signaling** other threads, e.g. `ThreadJoin`
  
  ```
  Initial value of semaphore = 0
  ThreadJoin {
    semaphore.P();
  }
  ThreadFinish {
    semaphore.V();
  }
  ```
Intuition for Semaphores

• What do you need to wait for?
  • Example: Critical section to be finished
  • Example: Queue to be non-empty
  • Example: Array to have space for new items

• What can you count that will be 0 when you need to wait?
  • Example: # of threads currently in critical section
  • Example: # of items currently in queue
  • Example: # of free slots in array

• Then: Use semaphore operations to maintain count
Summary

• Linux Completely Fair Scheduler
  • Always pick thread with minimum CPU time so far

• Real-Time Scheduling & Earliest Deadline First
  • Optimal, proven feasibility check

• Priority Inversion: Subtle consequence of strict priority scheduling

• Building better locks: Use hardware-provided atomic operations

• Semaphores: Generalizing lock's BUSY/FREE to an integer counter