Scheduling

- On most systems, there will be many more processes running than there are CPUs. The scheduler determines which job gets to use the CPU at what time.
- PCs are very different from mainframes
  - Many machines are limited by the rate of input rather than the speed of the CPU → IO bound
  - On high-end networked workstations and servers, scheduling is very important.
- Must pick right process to run
  - Make efficient use of CPU, but also minimise process switching, which has a cost in itself:
    - State has to be saved – registers and memory map
      - Specifically, TLB has to be flushed (see Ch 8).
    - New process’ state has to be loaded

Scheduling Categories

- Goals: fairness, policy enforcement, balance
- Depends on the OS and environment (even system/user pt of view)
- Batch systems → throughput, turnaround time, CPU utilisation
  - Interactive → response time, proportionality (users expectations)
  - Real-time → meeting deadlines, predictability
- Batch algorithms
  - first-come-first-served – next on queue is served (simple) (FIFO)
  - Shortest-Job-first – well known times, only good if 0 interarrival time
  - Shortest-remaining-time – for pre-emptive jobs (new short jobs get good service)
Three level Scheduling

- Jobs arrive at system, placed in input queue stored on disk
- **Admission scheduler** decided which to admit to system
  - Job enters system and process created — competes for CPU
  - If memory can’t hold processes – swapped to disk
  - Memory scheduler determines which kept in memory which on disk
    - Remember swapping to and from disk costs (disk IO bw goes down)
    - Degree of multi-programming = no processes wanted in mem
- CPU scheduler decides on which ready process to run

Round-Robin

- Process assigned time interval (**quantum**) (Round-Robin)
  - If process is running at end of quantum then
    - CPU pre-empted and given to another process
  - **Simple algorithm**
    - list of runnable processes,
    - end of quantum process moves to end of list.
  - Length of quantum affects performance, Why?
    - E.g. context switch = 1msec, quantum = 4 msec → CPU spend 20% on admin
    - Quantum = 100 msec = ? %

Round-Robin cont.

- Performance cont.
  - If 10 users hit <return> same time
    - → 10 processes on queue
    - CPU idle therefore starts first one
    - Next don’t get CPU until 100 msec later! Last key = 1sec!!! 1
      - 1 second response time for <return> NO THANKYOU!
    - Good to have quantum > mean CPU burst → pre-emption rarely will happen
      - Why is this good?
        - **Recommendation is around 20-50 msec [Tanenbaum]**

Priority Scheduling

- **all processes are equal but some are more equal than others**
  - RR= all processes are equal
  - Pecking order (deans, profs, SLs, janitors and then students ;-) )
  - Assigned a **priority** and highest runs
  - Even PC (single user) multiple processes e.g. mail daemon vs. video player
  - Prevent greedy processes by reducing their priority at clock ticks
Priority Scheduling cont.

- Can assign costs to priority.
- E.g. my fab research job on a supercomputer gets 100 and I pay for it, whereas student compile gets 10 for free! :-)
  - Nice → user reduces priority to be real nice to others
  - (ask about the dept, has anyone ever ever used it?)

- Dynamic allocation:
  - IO bound job gets higher CPU priority, why?
  - During IO, the IO-Process waits until finished
  - Gets CPU immediately so it can get out of the way
  - Out of the way = out of memory and let CPU bound on for longer in parallel

- Beware of Starvation

Multiple Queues

- Large quantum = poor response time
- Processes divided into priority classes
- Processes in highest class run for 1 quantum
  - Process in next highest run for 2 quanta next 4 etc
  - Whenever a process has used its quantum, it is moved down a class
  - E.g.
    - Process requires 100 quanta
    - Gets 1 quantum then swaps out, then 2 quanta, 4, 8, 16, 32, 64 (37)
    - 7 Swaps in total, how many for RR?

Policy Vs Mechanism

- Large apps like DBMS have many child processes
  - DBMS knows how to schedule better than OS
  - Therefore is OS separates mechanism from policy
  - DBMS can communicate (via parameters) to OS

- Threads
  - User level - use RR and Pri to schedule them
    - won’t have interrupt for long threads hogging time, context switch is lightweight
  - Kernel – context switch is full state save (heavy/slow)

(1) Priority - Pre-emption

- Each process is assigned a priority.
- A process runs until it suspends itself or it is interrupted.
- If the interrupt makes a higher priority process ready the high priority process will be run.
  - i.e. the lower priority process is pre-empted.
(2) Run-to-completion Scheduling

(Natural Break)

- Process runs until it suspends itself.
- Suspending itself could be (in the case of Simple Kernel)
  - P() on a semaphore
  - delay()
- The same process will run after an interrupt has occurred as was running before.

Example: Interrupt Latency with Natural Break.

- Ready-Queue State: P1, P2
- Clock Interrupt at time T puts P3 on readyQ.
- Ready-Queue State: P1, P2, P3
- P1 then runs for 2 milliseconds before suspending itself.
- P2 runs for 3 milliseconds before suspending itself.
- Thus the latency between the clock interrupt occurring and P3 being run on the processor is 5 milliseconds.

Advantages:
- Simple to implement
- Efficient implementation of mutual exclusion

Disadvantages:
- Assumes well-behaved user processes.
- No loops without breaks (delay/P)
- Short time between breaks
- Response time depends on run-times of user programs.

(3) Time Sliced Scheduling

- Each process is run until its time slice expires or until it suspends itself.
- Used in time sharing systems to share processor among multiple users.

Exercise:

What transition in the Simple Kernel must be disallowed if it is to be modified to a Natural Break Scheduling Strategy?

Which Kernel procedures must be modified and how?
(4) Multi-Level Queues - Dynamic Priority Assignment

- Unblocked processes
- Completion
- Pre-emption

In real life: Scheduling in UNIX

- Low-level scheduling algorithm is a dynamic priority scheduling algorithm
  - See Tanenbaum Ch 10.
  - There are many different kinds of Unix; the scheduling I describe here is designed to be consistent and a good representation of Unix scheduling, but may not completely correspond to any one particular instance of Unix.
- Each process is assigned a base priority by the OS.
  - Depends on the type of process (kernel vs. user mode, etc.)
  - Range: -20 <= base <= 20. -20 is the highest base priority!
- Users can assign a lower priority to their own processes
  - Represented by a value "nice": -20 <= nice <= 20. Only root can set negative.
- Initial priority of a process: priority = base + nice (within [-20,20]).
- One ready queue per priority
- Basic scheduling algorithm then selects first process from highest non-empty ready queue (this part is like Simple Kernel).
- Priorities are re-calculated once per second, based on recent CPU utilisation
  - Q: number of time slices a process ran during the last second.
  - Penalty: CPU utilisation = Q + (CPU utilisation_{-1}) / 2
  - Note that this penalty decays exponentially
  - Priorities are re-calculated: priority = CPU utilisation + nice + base
- Example:
  - Process with base priority 0, nice 0. Runs for one full second. What is its priority for the next 5 seconds, assuming it does not run again?
  - The effect is a scheduling system that favours interactive processes.

In real life: Linux Scheduling

- Linux scheduling is different from UNIX. Three classes:
  - "real-time" FIFO
  - "real-time" round-robin
  - Timesharing
- See Tanenbaum Section 10.3; same caveat as UNIX scheduling.
- Each process has a static priority and a niceness, with a fixed relationship priority = 20 - nice. Valid range [1,40]. 40 is highest.
  - Default static priority for user processes is 20 (niceness 0).
  - Adjust the static priority of a process when it starts:
    - /bin/nice -n <niceness> <command>
  - Only root can invoke /bin/nice with a negative niceness.
- In addition to its static priority, each process has a dynamic priority known as its quantum. quantum is initialised to priority.
  - Clock tick every 10msec, 10ms interval known as a "jiffy"
  - quantum of running process is decremented by 1 at every clock tick.
- We define goodness = quantum + priority if quantum > 0, else 0.
- When the scheduler is called, the process with the highest goodness greater than zero is selected to run.
Linux Scheduling Continued

- The scheduler is not run every 10ms – too expensive. Instead, the scheduler runs when the current process becomes ineligible to run (because its goodness becomes 0, or it blocks, etc.)
- Eventually, the goodness of all ready processes becomes zero
  - Question: which processes do not have goodness zero?
  - When that happens, we recalculate the dynamic priority (quantum) of all processes based on quantum = (quantum / 2) + priority
  - The effect is that processes that did not utilise their full quantum start with higher quantum after recalculation.
  - The time period between re-calculations of dynamic priority is called an epoch.
- Example:
  - Two processes, CPU intensive, assume no semaphores. A is started normally, B is started with /bin/nice -n 15 B. Question: What percentage of CPU do A and B receive, respectively?

In real Life: Windows 2000 Scheduling

- See Tanenbaum book.
- Windows 2000 has processes and (within each process), threads.
  - Two API functions to interact with scheduler: SetPriorityClass (process) and SetThreadPriority.
- System has 32 priorities (0 to 31)
  - Combination of priority class and thread priority is mapped to priority number via table below.
  - Each thread has a base priority. Current priority can be higher (see next slide)

<table>
<thead>
<tr>
<th>Win32 process class priorities</th>
<th>Realtime</th>
<th>High</th>
<th>Above Normal</th>
<th>Normal</th>
<th>Below Normal</th>
<th>Idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time critical</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Highest</td>
<td>26</td>
<td>15</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>Above normal</td>
<td>25</td>
<td>14</td>
<td>11</td>
<td>9</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>Normal</td>
<td>24</td>
<td>13</td>
<td>10</td>
<td>8</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>Below normal</td>
<td>23</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>Lowest</td>
<td>22</td>
<td>11</td>
<td>8</td>
<td>6</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Idle</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Windows 2000 Scheduling Continued

- 32-level ready queue (see picture)
  - Scheduler selects first thread from highest non-empty ready queue.
- Scheduling is done on a thread basis, independent of process membership.
- Modifications to this basic system to improve interactive performance:
  - I/O completes and releases waiting thread: current priority boosted depending on device (e.g. 1 for disk, 8 for sound!)
  - Thread waiting on semaphore (etc.) released: boosted by 2 if “active window”, 1 otherwise.
  - To deal with priority inversion
    - These boosts are not forever, decay by 1 on each quantum where thread is active.
- Default quantum is 20ms.
  - Windows that become “active” get more time.