Scheduling

What is Scheduling?
– On a multi-programmed system
  • We may have more than one Ready process
– On a batch system
  • We may have many jobs waiting to be run
– On a multi-user system
  • We may have many users concurrently using the system
  • The scheduler decides who to run next.
  – The process of choosing is called scheduling.

Is scheduling important?
• It is not in certain scenarios
  – If you have no choice
    • Early systems
      – Usually batching
      – Scheduling algorithm simple
        » Run next on tape or next on punch tape
    – Only one thing to run
      • Simple PCs
        – Only ran a word processor, etc....
      • Simple Embedded Systems
        – TV remote control, washing machine, etc....

Is scheduling important?
• It is in most realistic scenarios
  – Multitasking/Multi-user System
    • Example
      – Email daemon takes 2 seconds to process each email
      – User clicks button on application.
    • Scenario 1
      – Run daemon, then application
        » System appears really sluggish to the user
    • Scenario 2
      – Run application, then daemon
        » Application appears really responsive, small email delay is unnoticed
  • Scheduling decisions can have a dramatic effect on the perceived performance of the system
  – Can also affect correctness of a system with deadlines

Application Behaviour
• Bursts of CPU usage alternate with periods of I/O wait

Application Behaviour
a) CPU-Bound process
  • Spends most of its computing
  • Time to completion largely determined by received CPU time
Application Behaviour

b) I/O-Bound process
- Spend most of its time waiting for I/O to complete
- Small bursts of CPU to process I/O and request next I/O
- Time to completion largely determined by I/O request time

Observations

- Generally, technology is increasing CPU speed much faster than I/O speed
  ⇒ CPU bursts becoming shorter, I/O waiting is relatively constant
  ⇒ Processes are becoming more I/O bound

Observations

- We need a mix of CPU-bound and I/O-bound processes to keep both CPU and I/O systems busy
- Process can go from CPU- to I/O-bound (or vice versa) in different phases of execution

Observations

- Choosing to run an I/O-bound process delays a CPU-bound process by very little
- Choosing to run a CPU-bound process prior to an I/O-bound process delays the next I/O request significantly
  - No overlap of I/O waiting with computation
  - Results in device (disk) not as busy as possible
  ⇒ Generally, favour I/O-bound processes over CPU-bound processes

When is scheduling performed?

- A new process
  - Run the parent or the child?
- A process exits
  - Who runs next?
- A process waits for I/O
  - Who runs next?
- A process blocks on a lock
  - Who runs next? The lock holder?
- An I/O interrupt occurs
  - Who do we resume, the interrupted process or the process that was waiting?
- On a timer interrupt? (See next slide)
- Generally, a scheduling decision is required when a process (or thread) can no longer continue, or when an activity results in more than one ready process.

Preemptive versus Non-preemptive Scheduling

- Non-preemptive
  - Once a thread is in the running state, it continues until it completes, blocks on I/O, or voluntarily yields the CPU
  - A single process can monopolised the entire system
- Preemptive Scheduling
  - Current thread can be interrupted by OS and moved to ready state.
  - Usually after a timer interrupt and process has exceeded its maximum run time
  - Can also be as a result of higher priority process that has become ready (after I/O interrupt).
  - Ensures fairer service as single thread can’t monopolise the system
  - Requires a timer interrupt
Categories of Scheduling Algorithms

- The choice of scheduling algorithm depends on the goals of the application (or the operating system)
  - No one algorithm suits all environments
- We can roughly categorise scheduling algorithms as follows
  - Batch Systems
    - No users directly waiting, can optimise for overall machine performance
  - Interactive Systems
    - Users directly waiting for their results, can optimise for user perceived performance
  - Realtime Systems
    - Jobs have deadlines, must schedule such that all jobs (mostly) meet their deadlines.

Goals of Scheduling Algorithms

- All Algorithms
  - Fairness
    - Give each process a fair share of the CPU
  - Policy Enforcement
    - What ever policy chosen, the scheduler should ensure it is carried out
  - Balance/Efficiency
    - Try to keep all parts of the system busy

Goals of Scheduling Algorithms

- Interactive Algorithms
  - Minimise response time
    - Response time is the time difference between issuing a command and getting the result
    - E.g. selecting a menu, and getting the result of that selection
    - Response time is important to the user’s perception of the performance of the system.
  - Provide Proportionality
    - Proportionality is the user expectation that short jobs will have a short response time, and long jobs can have a long response time.
    - Generally, favour short jobs

Goals of Scheduling Algorithms

- Real-time Algorithms
  - Must meet deadlines
    - Each job/task has a deadline.
    - A missed deadline can result in data loss or catastrophic failure
    - Aircraft control system missed deadline to apply brakes
  - Provide Predictability
    - For some apps, an occasional missed deadline is okay
    - E.g. DVD decoder
    - Predictable behaviour allows smooth DVD decoding with only rare skips

Round Robin Scheduling

- Each process is given a timeslice to run in
- When the timeslice expires, the next process preempts the current process, and runs for its timeslice, and so on
  - The preempted process is placed at the end of the queue
- Implemented with
  - A ready queue
  - A regular timer interrupt

Interactive Scheduling
Example

- 5 Processes
  - Process 1 arrives slightly before process 2, etc...
  - All are immediately runnable
  - Execution times indicated by scale on x-axis

Round Robin Schedule

- Timeslice = 1 unit
- Timeslice = 3 units

Round Robin

- Pros
  - Fair, easy to implement
- Cons
  - Assumes everybody is equal
- Issue: What should the timeslice be?
  - Too short
    - Waste a lot of time switching between processes
    - Example: timeslice of 4ms with 1ms context switch = 20% round robin overhead
  - Too long
    - System is not responsive
    - Example: timeslice of 100ms
      - If 10 people hit “enter” key simultaneously, the last guy to run will only see progress after 1 second.
    - Degenerates into FCFS if timeslice longer than burst length

Priorities

- Each Process (or thread) is associated with a priority
- Provides basic mechanism to influence a scheduler decision:
  - Scheduler will always choose a thread of higher priority over lower priority
- Priorities can be defined internally or externally
  - Internal: e.g. I/O bound or CPU bound
  - External: e.g. based on importance to the user

Example

- 5 Jobs
  - Job number equals priority
  - Priority 1 > priority 5
  - Release and execution times as shown
- Priority-driven preemptively scheduled
**Priorities**

- Usually implemented by multiple priority queues, with round robin on each queue
- **Con**
  - Low priorities can starve
  - Need to adapt priorities periodically
    - Based on ageing or execution history

**Traditional UNIX Scheduler**

- **Two-level scheduler**
  - High-level scheduler schedules processes between memory and disk
  - Low-level scheduler is CPU scheduler
    - Based on a multi-level queue structure with round robin at each level

- **The highest priority (lower number) is scheduled**
- Priorities are re-calculated once per second, and re-inserted in appropriate queue
  - Avoid starvation of low priority threads
  - Penalise CPU-bound threads
Real-time Scheduling

• Correctness of the system may depend not only on the logical result of the computation but also on the time when these results are produced, e.g.
  – Tasks attempt to control events or to react to events that take place in the outside world
  – These external events occur in real time and processing must be able to keep up
  – Processing must happen in a timely fashion, neither too late, nor too early

Real Time System (RTS)

• RTS accepts an activity A and guarantees its requested (timely) behaviour B if and only if
  – RTS finds a schedule
    • that includes all already accepted activities Ai and the new activity A,
    • that guarantees all requested timely behaviour Bi and B, and
  – Otherwise, RT system rejects the new activity A.

Typical Real Time Systems

• Control of laboratory experiments
• Robotics
• (Air) Traffic control
• Controlling Cars / Trains / Planes
• Telecommunications
• Medical support (Remote Surgery, Emergency room)
• Multi-Media

• Remark: Some applications may have only soft-real time requirements, but some have really hard real-time requirements

Hard-Real Time Systems

• Requirements:
  – Must always meet all deadlines (time guarantees)
  – You have to guarantee that in any situation these applications are done in time, otherwise dangerous things may happen

Examples:
1. If the landing of a fly-by-wire jet cannot react to sudden side-winds within some milliseconds, an accident might occur.
2. An airbag system or the ABS has to react within milliseconds
**Soft-Real Time Systems**

Requirements:
Must mostly meet all deadlines, e.g. 99.9% of cases

Examples:
1. Multi-media: 100 frames per day might be dropped (late)
2. Car navigation: 5 late announcements per week are acceptable
3. Washing machine: washing 10 sec over time might occur once in 10 runs, 50 sec once in 100 runs.

**Predictability, not Speed**

- Real time systems are NOT necessarily fast
- Real time systems can be slow, as long as they are predictably so.
  - It does not matter how fast they are, as long as they meet their deadlines.

**Properties of Real-Time Tasks**

- To schedule a real time task, its properties must be known *a priori*
- The most relevant properties are
  - Arrival time (or release time) $a_i$
  - Maximum execution time (service time)
  - Deadline $d_i$

**Categories of Real-Time Tasks**

- Periodic
  - Each task is repeated at a regular interval
  - Max execution time is the same each period
  - Arrival time is usually the start of the period
  - Deadline is usually the end
- Aperiodic (and sporadic)
  - Each task can arrive at any time

**Real-time scheduling approaches**

- Static table-driven scheduling
  - Given a set of tasks and their properties, a schedule (table) is precomputed offline.
  - Used for periodic task set
  - Requires entire schedule to be recomputed if we need to change the task set
- Static priority-driven scheduling
  - Given a set of tasks and their properties, each task is assigned a fixed priority
  - A preemptive priority-driven scheduler used in conjunction with the assigned priorities
  - Used for periodic task sets

**Real-time scheduling approaches**

- Dynamic scheduling
  - Task arrives prior to execution
  - The scheduler determines whether the new task can be admitted
    - Can all other admitted tasks and the new task meet their deadlines?
    - If no, reject the new task
  - Can handle both periodic and aperiodic tasks
Scheduling in Real-Time Systems

- We will only consider periodic systems

Schedulable real-time system

- Given
  - \( m \) periodic events
  - Event \( i \) occurs within period \( P_i \) and requires \( C_i \) seconds

- Then the load can only be handled if

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
\]

Two Typical Real-time Scheduling Algorithms

- Rate Monotonic Scheduling
  - Static Priority priority-driven scheduling
  - Priorities are assigned based on the period of each task
    - The shorter the period, the higher the priority
  - Earliest Deadline First Scheduling
    - The task with the earliest deadline is chosen next

A Scheduling Example

- Three periodic Tasks

Is the Example Schedulable

\[
\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1
\]

\[
\frac{10}{30} + \frac{15}{40} + \frac{5}{50} = 0.808
\]

- YES

Two Schedules: RMS and EDF

Let’s Modify the Example Slightly

- Increase A’s CPU requirement to 15 msec
  - The system is still schedulable

\[
\frac{15}{30} + \frac{15}{40} + \frac{5}{50} = 0.975
\]
RMS and EDF

RMS failed, why?

• It has been proven that RMS is only guaranteed to work if the CPU utilisation is not too high
  – For three tasks, CPU utilisation must be less than 0.780
    • We were lucky with our original example
      \[ \sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{\frac{1}{m}} - 1) \]

EDF

• EDF always works for any schedulable set of tasks, i.e. up to 100% CPU utilisation

• Summary
  – If CPU utilisation is low (usual case, due to safety factor in estimating execution times)
    • Can use RMS which is simple and easy to implement
  – If CPU utilisation is high
    • Must use EDF