Learning Outcomes

- Understand the role of the scheduler, and how its behaviour influences the performance of the system.
- Know the difference between I/O-bound and CPU-bound tasks, and how they relate to 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.

- It is in most realistic scenarios
  - Multitasking/Multi-user System
    - Example
      - Email daemon takes 2 seconds to process an 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

(b) I/O-Bound process
- Spends 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

**Observation**
- 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

**Key Insight**
- 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?
- 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 monopolise 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 users perceived performance
  - Realtime Systems
    - Jobs have deadlines, must schedule such that all jobs (predictably) meet their deadlines.

Goals of Scheduling Algorithms

- All Algorithms
  - Fairness
    - Give each process a fair share of the CPU
  - Policy Enforcement
    - Whatever 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

Interactive Scheduling

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
**Example**

- 5 Process
  - 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

**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 1 ms 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

- 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
Traditional UNIX Scheduler

- Priority = CPU_usage + nice + base
  - CPU_usage = number of clock ticks
  - Nice is a value given to the process by a user to permanently boost or reduce its priority
  - Base is a set of hardwired, negative values used to boost priority of I/O bound system activities
    - Swapper, disk I/O, Character I/O

Multiprocessor Scheduling

- Given X processes (or threads) and Y CPUs,
  - how do we allocate them to the CPUs

A Single Shared Ready Queue

- When a CPU goes idle, it take the highest priority process from the shared ready queue

Single Shared Ready Queue

- Pros
  - Simple
  - Automatic load balancing
- Cons
  - Lock contention on the ready queue can be a major bottleneck
    - Due to frequent scheduling or many CPUs or both
  - Not all CPUs are equal
    - The last CPU a process ran on is likely to have more related entries in the cache.

Affinity Scheduling

- Basic Idea
  - Try hard to run a process on the CPU it ran on last time
- One approach: Multiple Queue Multiprocessor Scheduling

Multiple Queue SMP Scheduling

- Each CPU has its own ready queue
  - Coarse-grained algorithm assigns processes to CPUs
    - Defines their affinity, and roughly balances the load
- The bottom-level fine-grained scheduler:
  - Is the frequently invoked scheduler (e.g. on blocking on I/O, a lock, or exhausting a timeslice)
  - Runs on each CPU and selects from its own ready queue
    - Ensures affinity
  - If nothing is available from the local ready queue, it runs a process from another CPUs ready queue rather than go idle
    - Terned “Work stealing”
Multiple Queue SMP Scheduling

• Pros
  – No lock contention on per-CPU ready queues in the (hopefully) common case
  – Load balancing to avoid idle queues
  – Automatic affinity to a single CPU for more cache friendly behaviour