Recall: Internal OS File Description

* Internal data structure everything about the file (what file descriptor refers to)
  + Where it resides
  + Its status
  + How to access it
* Pointer: struct file \*file (file description structure)
  + Everything accessed with file descriptor has one of these
  + This resides in kernel memory (it’s memory addresses mean nothing to user)
  + Different from FILE\* (buffered user memory buffering file 🡺 user has access to)
* Struct file\_operations \*f\_op
  + Describe how this particular device implements its operations
    - For disks: points to file operations
    - For pipes: points to pipe operations
    - For sockets: points to socket operations
  + Layer of indirection allows everything to look a file from user level
* File\_operations: Why everything can look a file
  + Seek, read, write, mmap, flush, etc 🡺 Operations/handler functions for file operations 🡺 something must implement to be able to look like a file
  + Associated with particular hardware device
  + Registers/unregisters itself with the kernel
  + Handler functions for each of the file operations

Device Driver: Driver specific code in the kernel that interacts directly with the device hardware

* Supports a standard, internal interface up (close to file operations)
* Same kernel I/O system can interact easily with different device drivers
* Special device-specific configuration supported with ioctl() system call
* Device drivers gives ability to interact identically with variety of devices (plug in USB key vs spinning Disk Drive)
* Device drivers typically divided into two pieces:
  + Top half: accessed in call path from system calls
    - A diagram of a life cycle

      Description automatically generated with low confidenceImplements a set of standard, cross-device calls like open(), close(), read(), write(), ioctl(), strategy()
    - This is the kernel interface to the device driver
    - Top half will start I/O to device, may put thread to sleep until finished (if I/O is slow)
  + Bottom half: run as interrupt routine
    - Gets input or transfers next block of output
    - May wake sleeping threads if I/O now complete

Device drivers run in the kernel stack of process which issued syscall 🡺 then kernel thread goes to sleep until I/O finishes 🡺 Device starts executing

It is the kernel thread which goes to sleep during I/O operations (great under one-to-one model as other user threads can run since their kernel threads don’t go to sleep)

A picture containing text, font, line, white

Description automatically generated

Goal for Today:

* Discussion of Scheduling:
  + Which thread should run on the CPU next?
  + How frequently do I loop?
  + Which thread do I cut off?
* Scheduling goals, policies
* Look at number of different schedulers

A diagram of a child execution process

Description automatically generated with low confidenceQuestions: How is the OS to decide which of several tasks to take off a queue?

* Scheduling: deciding which threads are given access to resources from moment to moment
  + Often we think in terms of CPU, but should also think about resources like network BW or disk access

Scheduling: All about queues

* CPU scheduling big area of research in early 70s
* Many implicit assumptions for CPU scheduling
  + One program per user
  + One thread per program
  + Programs are independent
* Clearly these are unrealistic, but they simplify the problem so it can be solved
  + For instance: is “fair” about fairness among users or programs?
    - If I run one compilation job and you run five, do you get five times much CPU on many OSs
  + The high level goal: Dole out CPU time to optimize some desired parameters of system
* Assumption: CPU bursts
  + Execution model: programs alternate between bursts of CPU and bursts of I/O
    - Programs typically use CPU for some period of time, then does I/O, then uses CPU again
    - Each scheduling decision is about which job to give to the CPU for use by its next CPU burst
    - With timeslicing, thread my be forced to give up CPU before finishing current CPU burst

Scheduling Policy Goals/Criteria

* Minimize Response Time
  + Minimize elapsed time to do an operation (or job)
  + Response time is what the user sees:
    - Time to echo a keystroke to the editor
    - Time to compile program
    - Real-time Tasks: Must meet deadlines imposed by World (hitting brake in a car)
* Maximize Throughout
  + Maximize operations (or jobs) per second
  + Throughput is related to response time, but not identical
    - Minimizing response time leads to more context switching
  + Two parts to maximize throughput
    - Minimize overhead (for example, context switching)
    - Efficient use of resources (CPU, disk, memory, etc) 🡺 give chance for cache to build (context switching destroys cache)
* Fairness
  + Share CPU among users in some equitable way
  + Fairness is not minimizing average response time
    - Better average response time by making system less fair

**FCFS Scheduling**

* First-Come, First-Served (FCFS)
  + Also “First In, First Out” (FIFO) or “Run until done”
    - In early systems, FCFS meant one program
    - Now, means keep CPU until thread blocks
  + Suppose processes arrive in order, P1 (24), P2 (3), P3 (3)
    - Waiting time for P1 = 0, P2 = 24, P3 = 27
    - Average waiting time: (0 + 24 + 27)/ 3 = 17
    - Average completion time: (24 + 27 + 30)/3 = 27
  + Alternatively, suppose they arrive P2, P3, P1
    - Waiting for P1 = 6, P2 = 0, P3 = 3
    - Average waiting time: (6 + 0+ 3)/3 = 3
    - Average completion time: (3+6+30)/3 = 13
  + Convoy effect: Short process stuck behind long process
  + With FCFS non-preemptive scheduling, convoys of small tasks tend to build up when a large one is running
  + FIFO Pros and Cons:
    - Simple (+)
    - Short jobs get stuck behind long ones (-)

A picture containing screenshot, text, line, font

Description automatically generated

**Round Robin Scheduling**

* FCFS Scheme: Potentially bad for short jobs!
  + Depends on submit order
  + If you are first in line at supermarket with milk, you don’t care who is behind you, on the other hand…
* Round Robing Scheme: Preemption!
  + Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds
  + After quantum expires, the process is preemoted and added to the end of the ready queue
  + n processes in ready queue and time quantum q
    - Each process gets 1/n of the CPU time
    - In chunks of at most q time units
    - No process wait more than (n-1)q time units
  + Performance
    - q large 🡺 FCFS
    - q small 🡺 Interleaved (really small 🡺 hyperthreading)?
    - q must be large with respect to context switch, otherwise overhead is too high but small enough to get responsiveness
  + P1 (53), P2 (8), P3 (68), P4 (24), q = 20

A picture containing text, font, screenshot, line

Description automatically generated

* + RR Pros and Cons:
    - Better for short jobs, Fair (+)
    - Context-switching time adds up for long jobs (-)

Decrease Response Time

* T1: Burst Length 10
* T2: Burt Length 1
* Q = 10
  + Average Response Time: (10 + 11) / 2 = 10.5
* Q = 5
  + Average Response Time: (6 + 11) / 2 = 8.5
* Why not make Q tiny?
  + Overhead from context switching
* Smaller slices can also decrease completion/response time

Graphical user interface, application

Description automatically generated

How to implement RR in the Kernel?

* FIFO Queue, as in FCFS
* But preempt job after quantum expires, and send it to the back of the queue
  + How? Timer Interrupt!
  + And, of course, careful synchronization
* How do you choose time slice?
  + If too big?
    - Response time suffers
  + Too small?
    - Throughput suffers
  + Actual choices of timeslice:
    - Initially, UNIX timeslice one second:
      * Worked ok, when UNIX was used by one or two people
      * What if three compilations going on? 3 seconds to echo each keystroke!
    - Need to balance short-job performance and long job throughput:
      * Typical time slice today is between 10ms - 100ms
      * Typical context switching overhead is 0.1ms – 1ms
      * Roughly 1% overhead

Comparisons between FCFS and Round Robin

* Table

  Description automatically generatedAssuming zero-cost context-switching, is RR always better than FCFS?
* Also cache state must be shared between all jobs with RR but can be devoted to each job with FIFO
* Total time for RR longer even for zero-cost switch

Graphical user interface, application, table, Excel

Description automatically generatedRound-Robin allows use to get midway between worst and best behavior of FCFS

How do we get responsiveness without disturbing the long processes?

All this analysis relies on knowing CPU burst after the fact. How do we make this kind of analysis in advance?

Handling differences in Importance: Strict Priority Scheduling/**Multi-Level Feedback Scheduling**

* Use Priority Queue instead of Queue
* Execution Plan:
  + Always execute highest-priority runnable jobs to completion
  + Each queue can be processed in RR with some time quantum

Diagram

Description automatically generated

* Problems:
  + Starvation:
    - Lower priority jobs don’t get to run because higher priority jobs
  + Deadlock: Priority inversion:
    - Happens when low priority task has lock needed by higher priority task
    - Usually involves third, intermediate priority task preventing high-priority task from running
  + How to fix problems?
    - Dynamic priorities – adjust base-level priority up or down based on heuristics about interactivity, locking, burst behavior, etc…

Scheduling Fairness

What about fairness?

* Strict fixed-priority scheduling between queues is unfair (run highest, next, etc)
  + Long running jobs may never get CPU
* Must give long-running jobs a fraction of the CPU even when there are shorter jobs to run
* Tradeoff: fairness gained by hurting avg response time

How to implement fairness?

* Could give each queue some fraction of the CPU
  + What if one long-running job and 100 short running ones?
    - Sometimes express lanes in supermarket get so long, you can get better service by going into one of the other lines
  + Could increase priority of jobs that don’t get service
    - Done in some variants of UNIX
    - This is an ad-hoc solution – what rate should you increase priorities?
    - Ad as systems gets overloaded, no job gets CPU time, so everyone increases in priority 🡪 Interactive jobs suffer

What if we knew the Future?

* Could we always mirror best FCFS?
* Short Job First (SJF):
  + Run whatever job has least amount of computation to do
* Shorter Remaining Time First (SRTF):
  + Preemptive version of SJF: if job arrives and has shorter time to completion than remaining time on the job 🡺 immediately preempt CPU
* These can be applied to whole program or current CPU burst
  + Idea is to get short jobs out of system’
  + Big effect on short jobs, only small effect on long ones
  + Result is better average response time
* SJF/SRTF are the best you can do at minimizing average response time
  + Provably optical
  + Since SRTF is always at least as good as SJF, focus on SRTF
* SRTF vs FCFS
  + IF all jobs are the same length?
    - SRTF becomes same as FCFS
  + What if jobs have varying length?
    - SRTF: short jobs not stuck behind long ones

Diagram

Description automatically generatedChart

Description automatically generated with medium confidence

SRTF Discussion:

* Starvation
  + SRTF can lead to starvation if many small jobs
  + Large jobs can never get to run
* Somehow need to predict future
  + How can we do this?
  + Sime systems ask the user
    - When you submit a job, have to say how long it will take
    - To stop cheating, system kills job if it takes too long
  + But: hard to predict job’s runtime even for non-malicious users
* Bottom line, can’t really know long job will take
  + However use SRTF as a yardstick for measuring other policies
  + Optimal, so can’t do any better
* SRTF Pros & Cons
  + Optimal (average response time) (+)
  + Hard to predict future (-)
  + Unfair (-)

Predicting the Length of the next CPU Burst

* Adaptive: Changing polict based on past behavior
  + CPU Scheduling, in virtual memory, in file systems
  + Works because programs have predictable behavior
    - If program was I/O bound in the past, likely in future
    - If computer behavior were reandom, wouldn’t help
  + Example: SRTF with estimated burst length
    - Use and estimator function on previous bursts:
      * Let tn-1, tn-2, tn-3, etc be previous CPU burst lengths. Estimate next burst tn = f(tn-1, tn-2, tn-3, …)
      * Function f could be many different time series estimation schemes (Kalman filters, etc)

**Lottery Scheduling**:

* Give each job some number of lottery tickers
* On each time slice, randomly pick a winning ticket
* One average, CPU time is proportional to number of tickets given to each job
* How to assign tickts?
  + To approximate SRTF, short running jobs get more, long running jobs get fewer
  + To avoid starvation, every job gets at least one ticket (everyone makes progress)

Table

Description automatically generated

How to evaluate a Scheduling algorithm?

* Deterministic modeling
  + Takes a predetermined workload and compute the performance of each algorithm for that workload
* Queueing models
  + Mathematical approach for handling stochastic workloads
* Implementation/Simulation:
  + Build system which allows actual algorithms to be run against actual data
  + Most flexible/general