Dispatcher

+ The dispatcher IS NOT the same as scheduler

+ Dispatcher module give control of the CPU to the processes selected by the scheduler: this involves

  - switching context (you want to reduce the context switching as few as possible)
  - switching to user mode
  - jumping to the proper location in the user program to restart that program

+ While the schedule decide which is the next process to run, the dispatcher is the act of executing the process and starting a process for execution

+ Example of processes in which the dispatcher switching will switch the context from one to another: Instruction pointers, program pointer, which instruction is being executed

+ Dispatch Latency: time it takes for the dispatcher to stop one process and start another running

+ switching from one process to another is the dispatcher job

+ http://lxr.linux.no → kernel/sched.c lines: 5453 (only switching when previous is different than next. Goes to 5470)

Scheduling Criteria

+ CPU utilization: Keep the CPU as busy as possible

+ Throughput: # of processes that complete their execution per time unit.

+ waiting time: Amount of time a process has been waiting in the ready queue ← the amount of time a process doesn't spend executing on CPU (assuming it doesn't perform blocking I/O)
  wait time = Completion time − (arrival time + burst time)
  → Note: waiting time is misleading, it is the time the cpu is ready to use but it waiting to execute. Reduce the waiting time for each process

+ turnaround time: Amount of time to execute a particular process, from submission to extermination
  → Why should turnaround time be more than the actual time of the process in the CPU?
  R: Because the adding time waiting for using the CPU

+ Response Time - amount of time it takes from when a requests was submitted until the first response is produce
  → (time to get the first start, the first response)

22 Optimization Criteria

(May have combination of these)

+ Max CPU utilization
+ Max throughput
+ Min turnaround time
+ Min waiting time
+ Min response time

Note: Real time systems, is when the deadline is most important

Scheduling Algorithm Goals (important) ← Policies that need to be enforced by the system

+ Fairness: No process should wait forever

+ All systems
  - ↳ Fairness - giving each process a fair share of the CPU (based on their importance)
  - ↳ Policy enforcement - seeing that stated policy is carried out
  - ↳ CPU utilization - keep the CPU busy all the time

Old mindset would be in order to get them.

+ Batch systems:
  - ↳ Throughput: maximize jobs per hour
  - ↳ turnaround time: minimize time between submission and termination
  - ↳ CPU utilization: keep the CPU busy all the time.

+ Interactive Systems:
  - ↳ Response time: Respond to requests quickly
  - ↳ Proportionality: Meet users expectations

+ Real-time systems
  - ↳ Meeting deadlines - avoid loosing data
  - Predictability - avoid quality degradation in multimedia systems.

[redacted] self study

First-Come, First-Served (FCFS) scheduling

FCFS p. 152-153 1
FCFS algorithm 378

+ is the simplest non preemptive

+ Processes are assigned the CPU in the order they request it

+ When first job enter the system from outside, it is started immediately and allowed to run without interruptions. Other jobs that come in are put on queue onto the end of the queue

+ Strength: easy to understand and equally easy to program

+ Weakness: ~~many Hotbombs can reduce its~~ [illegible]

+ Suppose that the processes arrive in order P1, P2, P3

Process	Burst Time
P1	24
P2	3
P3	3

Gantt chart

- Waiting time: P1 = 0
  P2 = 24
  P3 = 27
- Average waiting time: $\frac{(0+24+27)}{3} = 17$

+ Note: The context switching is missing in these charges

+ P1 starts immediately
  P2 have to wait until P1 is complete
  P3 have to wait until P2 is complete

? How long the process will take?
? How the process works?

24 FCFS scheduling (cont)

+ Suppose that the process arrive in the order: P2, P3 P1

Gantt chart:

16 − (0 + 7) = 9

↳ waiting time: P1 = 6, P2 = 0, P3 = 3

↳ Average waiting time: $\frac{(6+0+3)}{3} = 3$

↳ Better than previous case

↳ Convoy effect short process behind long process

↳ Shortest job first pass

Shortest-Job-First (SJF) Scheduling

+ This non preemptive batch algorithm assume the run time are known in advance (as in FCFS)

+ When several equally important jobs are sitting in the input queue waiting to be started, the scheduler picks the shortest job first.

+ Consider the length of its next CPU burst for each process.

+ Schedule the process having the shortest next CPU burst

+ One CPU given to the process it cannot be preempted until completes it CPU burst

+ SJF algorithm is optimal
  ↳ Gives a schedule with least average waiting time among all possible scheduling algorithm

Shortest-Job-First (SJR) Scheduling (cont)

↳ Shortest Remaining time Next (SRTF)

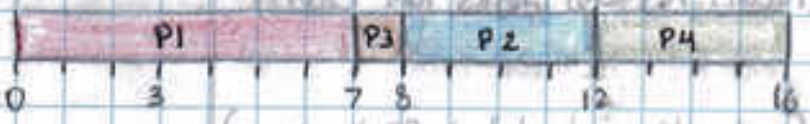
+This is a preemptive SJR: If a new process arrives with CPU burst length less than remaining time of currently executing process, Preempt the current process.

Examples:

Process	Arrival time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

+ wait time = completion time − (arrival time + burst time)

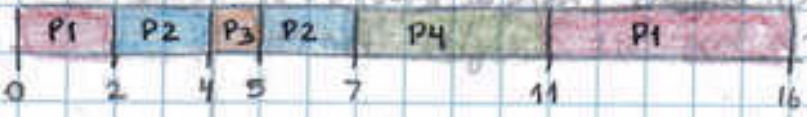
+ NON-PREEmptive Shortest-Job-First (SJR)



↳ P1 was the first to arrive, P3 is the shortest so go next, P2 arrived before P4

↳ Average waiting time: (0+6+3+7)/4 = 4 ← ASK

+ PREEMPTIVE shortest-job-First (SJR)/ shortest Remain time Next



↳ Average time = $\frac{(9+1+0+2)}{4} = 3$

+using exponential averaging, we can predict the time of each process

26 Determining the Length of Next CPU Burst

+ Not easy. Can only guess the length of next CPU burst

+ Can be done by using the length of previous CPU bursts, using exponential averaging

1. t_n = actual length of n^{th} CPU burst
2. τ_{n+1} = predicted value for the next CPU burst
3. α, $0 \leq \alpha \leq 1$
4. Define: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

+ Examples

↳ $\alpha = 0$
↦ $\tau_{n+1} = \tau_n$ (recent history does not count)

↳ $\alpha = 1$
↦ $\tau_{n+1} = \alpha t_n$ (only the actual last CPU burst counts)

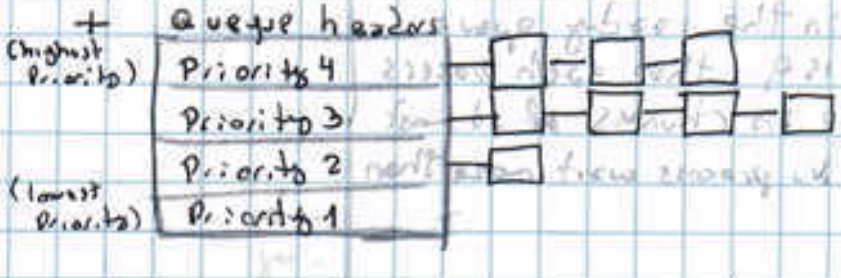
Formula expanded: $\tau_{n+1} = \alpha t_n + (1-\alpha)\alpha t_{n-1} + \cdots + (1-\alpha)^j \alpha t_{n-j} + \cdots + (1-\alpha)^{n+1} \tau_0$

+ Since both α and $(1-\alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor

Priority Scheduling

+ Each process have a priority number (integer).
  ~~This number~~
+ the CPU is allocated to the process with the highest priority (smallest integer = highest priority)

+ Queue headers



+ Shortest-Job-First (SJF) is a priority scheduling algorithm where priority is the predicted next CPU burst time
  ↳ Preemptive
  ↳ Non-Preemtive

+ (Problem) starvation = low priority process may never execute (never gets to use the processor)
  ↳ (solution) Aging = As time progresses increase the priority of a lower priority process that is not receiving CPU time

+

20. Round Robin (RR)

+ Each process get a small unit of CPU time (time quantum) (usually between 10 to 100 ms)

+ After this time has elapsed, the process is preempted and added to the end of the ready queue

+ If there are n processes in the ready queue and the time quantum is q, then each process get $\frac{1}{n}$ of the CPU time in chunks of at most q time units at ones. No process wait more than (n-1) q time units.

+ Performance: q large ⇒ FIFO
  q small ⇒ too much context switching

+ The first output would come faster in Round Robin because some process will run earlier

+ ~~two second time~~

Example: Suppose we have 10 processes sitting in depo. we could give the same time then were start again and fix the time to every process the performance is given the time it give to each process

+ the only problem is if the number of ~~processes~~ is very large it will take a long time

Example (Time quantum = 20)

Process	Burst time
P1	53
P2	17
P3	68
P4	24

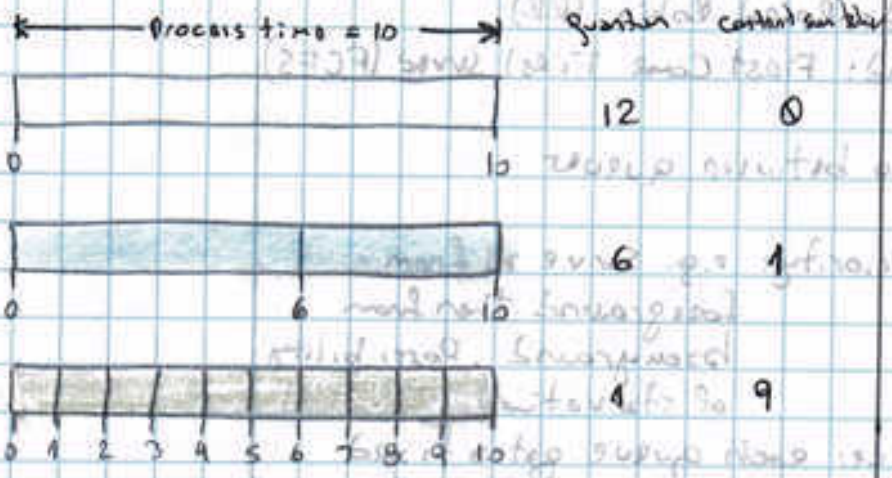
+ typically, higher average turnaround than SJF, but better response time

Time Quantum and Context Switch time

+ The smaller the time quantum, more the number of context switches.

+ Larger the time quantum, larger the response time

+ scheduling algorithm needs to find a balance between both of them



+ Context switching is not for free

+ Every time we have to clean the TLB from scratch which is very expensive, consumed the instruction time a lot.

+ systems without tlb are 5 time slower

TLB: Translation Lookaside Buffer

TLB: 195-197, 197-198, 895

30 Multilevel Queue

+ Ready queue is partitioned into separate queues:

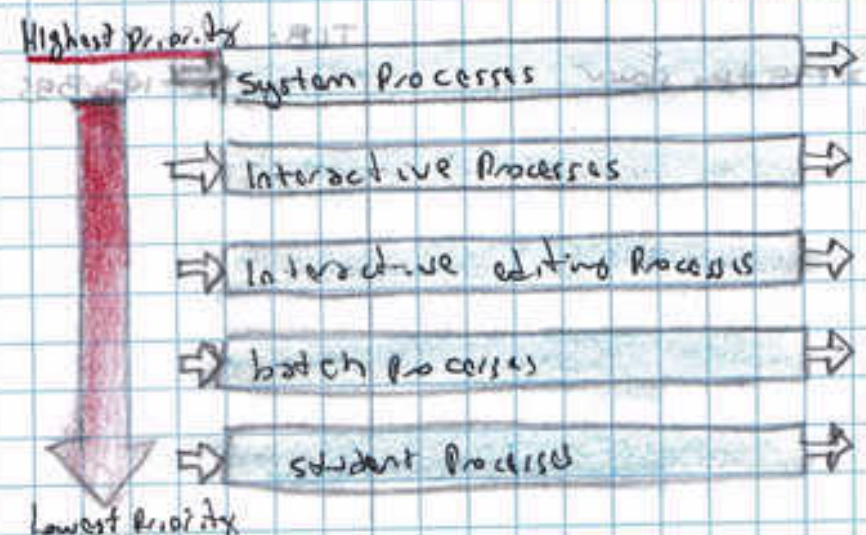
1. foreground (interactive)
2. background (batch)

+ Each queue has its own scheduling algorithm

1. foreground: Round Robin (RR)
2. background: First Come First Served (FCFS)

+ scheduling done between queues

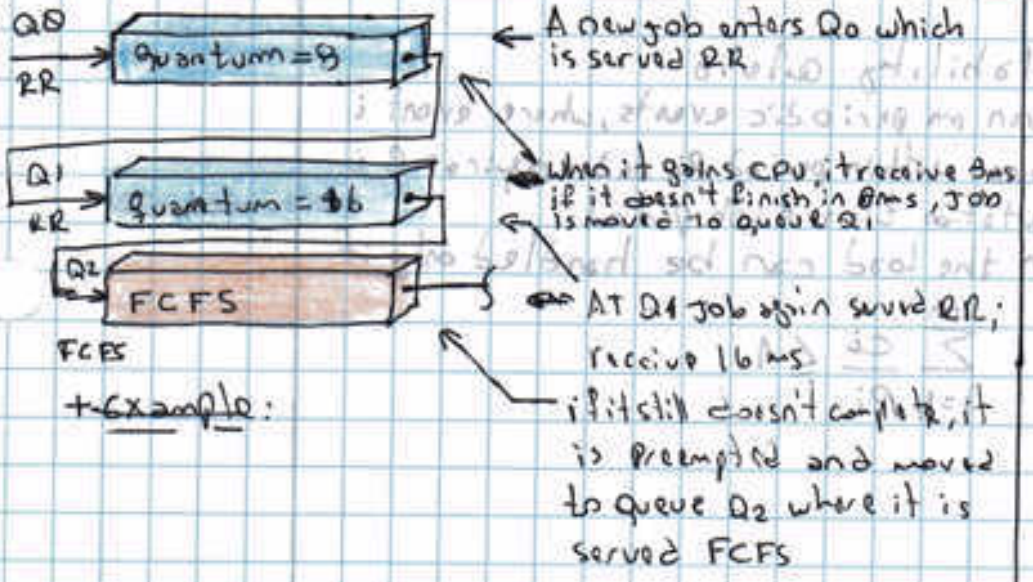
1. Fixed Priority: e.g. serve all from foreground then from background. Possibility of starvation
2. Time slice: each queue get a fixed share of cpu time which it can schedule amongst its processes. i.e.
   a. 80% to foreground in Round Robin
   b. 20% to background in FCFS



Multilevel Feedback Queue (MFQ)

+ A process can move between the various queues

+ MFQ scheduler defined by the following parameters:
  - → Number of queues
  - → Scheduling algorithms for each queue
  - → Method used to determine when to update or demote a process.



+ Example:

32 Real-time Scheduling

+ When each task needs to be completed before a given deadline

+ Hard real-time systems
  - Required to complete a critical task before its deadline
  - For example, in a flight control system, or nuclear reactor

+ Soft real-time systems
  - Meeting deadlines desirable, but not essential
  - For example, video or audio

+ Schedulability criteria
  - Given m periodic events, where event i occurs within period P_i and requires C_i computation time each period
  - Then the load can be handled only if

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

Scheduling Code: kernel/sched.c

Entry points into Kernel:
- arch/x86/entry_32.S
-or-
- arch/x86/entry_64.S

) invokes schedule()

General flow of control during scheduling

- Timer interrupt fires calling scheduler_tick()
-or-
- Some process blocks (or gives up CPU)

When Entry points invokes schedule() function?

- Figures out the next process to schedule and swap the prev. and next process via context_switch() which in turn:
  - Switches memory state using switch_mm()
  - Switches register plus stack state using switch_to()

SJF is optimal w.r.t average wait time

SJF: Shortest-Job-First

? w.r.t:

- Meaning that no other algorithm can archive a lower wait time than SJF

24 SJF is Optimal w.r.t average wait time (continue)

Proof:

1. Assume that there was an algorithm X that gave better average wait time than SJF for a set of N processes.
2. Since X != SJF, it means that there must be at least two processes P1 and P2 in the schedule generated by X such that:
   a. P1 executes before P2 does and...
   b. The CPU execution time of P1 is longer than P2 and...
   c. The average waiting time of the schedule is smaller than given by SJF
3. But, if you swap the position P1 and P2 in the schedule generated by X, the the average wait time goes down.
4. So keep swapping all such process pair that satisfy conditions (a) and (b). Each swap will reduce the average wait time
5. Finally you will end up with a schedule generated by SJF, whose wait time cannot be reduced any further. Hence SJF is Optimal

Fair Scheduling

Fairness:

- Fairness != equal CPU share for all processes
- Say you have N. processes
  1. Each process Pi is assigned a weight wi
  2. The CPU time will be divided among processes in proportion to their weights
  3. If some process does not use its assigned CPU time
     - The "spare" CPU time is divided among the remaining ready processes according to the ratio of their weights

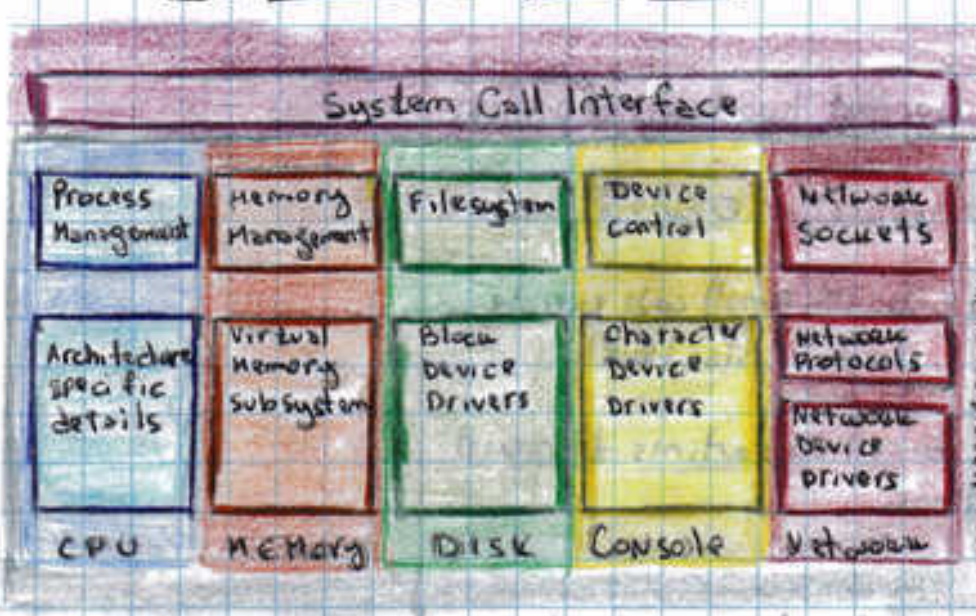
Work-conserving versus Non-work-conserving

- Work-conserving scheduler
  - CPU will not remain idle if there are processes in the ready queue.
- Non-work-conserving scheduler
  - Under some conditions, scheduler may decide to "waste" CPU time even though there may be processes sitting in the wait queue
    Ex: Sometimes real-time process cannot be started before a given time (release time).

Extra Notes

Intro to Systemcalls and Kernel Modules

Organization of Linux Kernel



- They can work as modules or kernel modules
- Components can be load in lazy mode, when need them
- With only modules required for booting, The booting process can be done faster, then other modules are load when need them.

Privilege modes: 0 - higher
3 - lower
1 - } virtualization
2 - }

System Calls

- System calls are like gates. They perform the transfer of control from user level to system (kernel) level in controlled matter.

- OSs typically support two levels of privilege:
  1. User Mode: Application execute at this level
  2. Supervisor Mode: OS(kernel) code execute at this level.

- Applications need to call OS routines to request privileged operations

- System calls
  1. Transfer control from lower privilege level (user mode) to higher privilege level (supervisor mode)
     ex: open, read, write, close, wait, exec, fork, kill.

- Kernel can tightly control entry points for the application into OS.
  - Application cannot randomly jump into any part of the OS code.

- Every system call have their own identity

- When trying to get access to a file, it can get more complicated because the CPU being used may be needed.

what are system calls?

which are the levels of privilege that typically an OS support

what does each system call have?

what happend when trying to get access to a file?