Non-Preemptive Scheduling

- Consider in the context of CPU scheduling
- Scheduled process runs until it yields CPU
- Works well for simple systems
  - Small numbers of processes
  - With natural producer consumer relationships
- Good for maximizing throughput
- Depends on each process to voluntarily yield
  - A piggy process can starve others
  - A buggy process can lock up the entire system

When Should a Process Yield?

- When it knows it's not going to make progress
  - E.g., while waiting for I/O
  - Better to let someone else make progress than sit in a pointless wait loop
- After it has had its "fair share" of time
  - Which is hard to define
  - Since it may depend on the state of everything else in the system
- Can't expect application programmers to do sophisticated things to decide

Non-Preemptive Scheduling Algorithms

- First come first served
- Shortest job next
- Real time schedulers

First Come First Served

- The simplest of all scheduling algorithms
- Run first process on ready queue
  - Until it completes or yields
- Then run next process on queue
  - Until it completes or yields
- Highly variable delays
  - Depends on process implementations
- All processes will eventually be served

First Come First Served Example

Dispatch Order		0, 1, 2, 3, 4			
Process	Duration		Start Time		End Time
0	350		0		350
1	125		350		475
2	475		475		950
3	250		950		1200
4	75		1200		1275
Total	1275				
Average wait			595		

Note: Average is worse than total/5 because four other processes had to wait for the slow-poke who ran first.

When Would First Come First Served Work Well?

- FCFS scheduling is very simple
- It may deliver very poor response time
- Thus it makes the most sense:
  1. In batch systems, where response time is not important
  2. In embedded (e.g. telephone or set-top box) systems where computations are brief and/or exist in natural producer/consumer relationships

Shortest Job First

- Find the shortest task on ready queue
  - Run it until it completes or yields
- Find the next shortest task on ready queue
  - Run it until it completes or yields
- Yields minimum average queuing delay
  - This can be very good for interactive response time
  - But it penalizes longer jobs

Shortest Job First Example

Dispatch Order			4,1,3,0,2		
Process	Duration		Start Time		End Time
4	75		0		75
1	125		75		200
3	250		200		450
0	350		450		800
2	475		800		1275
Total	1275				
Average wait			305		

Note: Even though total time remained unchanged, reordering the processes significantly reduced the average wait time.

Is Shortest Job First Practical?

- How can we know how long a job is going to run?
  - Processes predict for themselves?
  - The system predicts for them?
- How fair is SJF scheduling?
  - The smaller jobs will always be run first
  - New small jobs cut in line, ahead of older longer jobs
  - Will the long jobs ever run?
    - Only if short jobs stop arriving ... which could be never
- This is called *starvation*
  - It is caused by discriminatory scheduling

What If the Prediction is Wrong?

- Regardless of who made it
- In non-preemptive system, we have little choice:
  - Continue running the process until it yields
- If prediction is wrong, the purpose of Shortest-Job-First scheduling is defeated
  - Response time suffers as a result
- Few computer systems attempt to use Shortest-Job-First scheduling
  - But grocery stores and banks do use it
    - 10-item-or-less registers
    - Simple deposit & check cashing windows

Real Time Schedulers

- For certain systems, some things must happen at particular times
  - E.g., industrial control systems
  - If you don't rivet the widget before the conveyer belt moves, you have a worthless widget
- These systems must schedule on the basis of real-time deadlines
- Can be either *hard* or *soft*

Hard Real Time Schedulers

- The system absolutely must meet its deadlines
- By definition, system fails if a deadline is not met
  - E.g., controlling a nuclear power plant . . .
- How can we ensure no missed deadlines?
- Typically by very, very careful analysis
  - Make sure no possible schedule causes a deadline to be missed
  - By working it out ahead of time
  - Then scheduler rigorously follows deadlines

Ensuring Hard Deadlines

- Must have deep understanding of the code used in each job
  - You know exactly how long it will take
- Vital to avoid non-deterministic timings
  - Even if the non-deterministic mechanism usually speeds things up
  - You're screwed if it ever slows them down
- Typically means you do things like turn off interrupts
- And scheduler is non-preemptive

How Does a Hard Real Time System Schedule?

- There is usually a very carefully pre-defined schedule
- No actual decisions made at run time
- It's all been worked out ahead of time
- Not necessarily using any particular algorithm
- The designers may have just tinkered around to make everything "fit"

Soft Real Time Schedulers

- Highly desirable to meet your deadlines
- But some (or any) of them can occasionally be missed
- Goal of scheduler is to avoid missing deadlines
  - With the understanding that you might
- May have different classes of deadlines
  - Some "harder" than others
- Need not require quite as much analysis

What If You Don't Meet a Deadline?

- Depends on the particular type of system
- Might just drop the job whose deadline you missed
- Might allow system to fall behind
- Might drop some other job in the future
- At any rate, it will be well defined in each particular system

What Algorithms Do You Use For Soft Real Time?

- Most common is Earliest Deadline First
- Each job has a deadline associated with it
  - Based on a common clock
- Keep the job queue sorted by those deadlines
- Whenever one job completes, pick the first one off the queue
- Perhaps prune the queue to remove jobs whose deadlines were missed
- Minimizes total lateness