[04] SCHEDULING

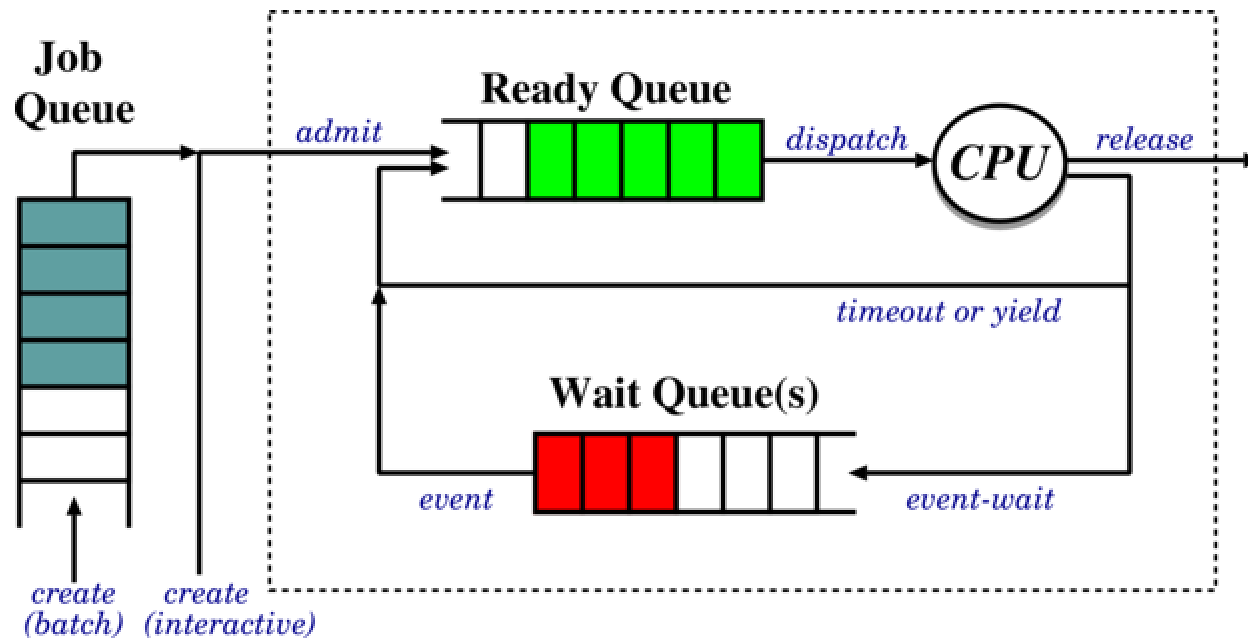
OUTLINE

- Scheduling Concepts
  - Queues
  - Non-preemptive vs Preemptive
  - Idling
- Scheduling Criteria
  - Utilisation
  - Throughput
  - Turnaround, Waiting, Response Times
- Scheduling Algorithms
  - First-Come First-Served
  - Shortest Job First
  - Shortest Response Time First
  - Predicting Burst Length
  - Round Robin
  - Static vs Dynamic Priority

SCHEDULING CONCEPTS

- **Scheduling Concepts**
  - **Queues**
  - **Non-preemptive vs Preemptive**
  - **Idling**
- Scheduling Criteria
- Scheduling Algorithms

QUEUES



- **Job Queue**: batch processes awaiting admission
- **Ready Queue**: processes in main memory, ready and waiting to execute
- **Wait Queue(s)**: set of processes waiting for an IO device (or for other processes)
  - **Job** scheduler selects processes to put onto the ready queue
  - **CPU** scheduler selects process to execute next and allocates CPU

PREEMPTIVE VS NON-PREEMPTIVE

OS needs to select a ready process and allocate it the CPU
When?

- ...a running process blocks (`running → blocked`)
- ...a process terminates (`running → exit`)

If scheduling decision is only taken under these conditions, the scheduler is said to be **non-preemptive**

- ...a timer expires (`running → ready`)
- ...a waiting process unblocks (`blocked → ready`)

Otherwise it is **preemptive**

NON-PREEMPTIVE

- Simple to implement:
  - No timers, process gets the CPU for as long as desired
- Open to *denial-of-service*:
  - Malicious or buggy process can refuse to yield

Typically includes an *explicit* `yield` system call or similar, plus *implicit* yields, e.g., performing IO, waiting

Examples: MS-DOS, Windows 3.11

PREEMPTIVE

- Solves denial-of-service:
  - OS can simply preempt long-running process
- More complex to implement:
  - Timer management, concurrency issues

Examples: Just about everything you can think of :)

IDLING

We will usually assume that there's always something ready to run. But what if there isn't?

This is quite an important question on modern machines where the common case is >50% idle

IDLING

Three options

1. Busy wait in scheduler, e.g., Windows 9x
    - Quick response time
    - Ugly, useless

IDLING

Three options

1. Busy wait in scheduler
2. Halt processor until interrupt arrives, e.g., modern OSs
    - Saves power (and reduces heat!)
    - Increases processor lifetime
    - Might take too long to stop and start

IDLING

Three options

1. Busy wait in scheduler
2. Halt processor until interrupt arrives
3. Invent an idle process, always available to run
    - Gives uniform structure
    - Could run housekeeping
    - Uses some memory
    - Might slow interrupt response

In general there is a trade-off between responsiveness and usefulness. Consider the important resources and the system complexity

SCHEDULING CRITERIA

- Scheduling Concepts
- **Scheduling Criteria**
  - **Utilisation**
  - **Throughput**
  - **Turnaround, Waiting, Response Times**
- Scheduling Algorithms

SCHEDULING CRITERIA

Typically one expects to have more than one option — more than one process is `runnable`

On what basis should the CPU scheduler make its decision?

Many different metrics may be used, exhibiting different trade-offs and leading to different operating regimes

CPU UTILISATION

> *Maximise the fraction of the time the CPU is actively being used*

Keep the (expensive?) machine as busy as possible

But may penalise processes that do a lot of IO as they appear to result in the CPU not being used

THROUGHPUT

> *Maximise the number of that that complete their execution per time unit*

Get useful work completed at the highest rate possible

But may penalise long-running processes as short-run processes will complete sooner and so are preferred

TURNAROUND TIME

> *Minimise the amount of time to execute a particular process*

Ensures every processes complete in shortest time possible

WAITING TIME

> *Minimise the amount of time a process has been waiting in the `ready` queue*

Ensures an interactive system remains as responsive as possible

But may penalise IO heavy processes that spend a long time in the `wait` queue

RESPONSE TIME

Minimise the amount of time it takes from when a request was submitted until the first response is produced

Found in time-sharing systems. Ensures system remains as responsive to clients as possible under load

But may penalise longer running sessions under heavy load

SCHEDULING ALGORITHMS

- Scheduling Concepts
- Scheduling Criteria
- **Scheduling Algorithms**
  - **First-Come First-Served**
  - **Shortest Job First**
  - **Shortest Response Time First**
  - **Predicting Burst Length**
  - **Round Robin**
  - **Static vs Dynamic Priority**

FIRST-COME FIRST-SERVED (FCFS)

Simplest possible scheduling algorithm, depending only on the order in which processes arrive

E.g. given the following demand:

Process	**Burst Time**
P_1	25
P_2	4
P_3	7

EXAMPLE: FCFS

Consider the average waiting time under different arrival orders

$\mathbf{P_1, P_2, P_3}$:

- Waiting time $P_1 = 0, P_2 = 25, P_3 = 29$
- Average waiting time: $\frac{(0+25+29)}{3} = 18$

$\mathbf{P_3, P_2, P_1}$:

- Waiting time $P_1 = 11, P_2 = 7, P_3 = 0$
- Average waiting time: $\frac{(11+7+0)}{3} = 6$

Arriving in reverse order is *three times as good*!

- The first case is poor due to the **convoy effect**: later processes are held up behind a long-running first process
- FCFS is simple but not terribly robust to different arrival processes

SHORTEST JOB FIRST (SJF)

Intuition from FCFS leads us to *shortest job first* (SJF) scheduling

- Associate with each process the length of its next CPU burst
- Use these lengths to schedule the process with the shortest time
- Use, e.g., FCFS to break ties

EXAMPLE: SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Waiting time for $P_1 = 0, P_2 = 6, P_3 = 3, P_4 = 7$. Average waiting time: $\frac{(0+6+3+7)}{4} = 4$

SJF is optimal with respect to average waiting time:

- It minimises average waiting time for a given set of processes
- What might go wrong?

SHORTEST REMAINING-TIME FIRST (SRTF)

Simply a preemptive version of SJF: preempt the running process if a new process arrives with a CPU burst length less than the remaining time of the current executing process

EXAMPLE: SRTF

As before:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

Waiting time for $P_1 = 9, P_2 = 1, P_3 = 0, P_4 = 2$

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

EXAMPLE: SRTF

Surely this is optimal in the face of new runnable processes arriving? Not necessarily — why?

- Context switches are not free: many very short burst length processes may thrash the CPU, preventing useful work being done
- More fundamentally, we can't generally know what the **future** burst length is!

PREDICTING BURST LENGTHS

- For both SJF and SRTF require the next "burst length" for each process means we must estimate it
- Can be done by using the length of previous CPU bursts, using exponential averaging:

  1. t_n = actual length of nth CPU burst.
  2. τ_{n+1} = predicted value for next CPU burst.
  3. For $\alpha, 0 \leq \alpha \leq 1$ define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

PREDICTING BURST LENGTHS

- If we expand the formula we get:

$$\tau_{n+1} = \alpha t_n + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1} \tau_0$$

where τ_0 is some constant

- Choose value of α according to our belief about the system, e.g., if we believe history irrelevant, choose $\alpha \approx 1$ and then get $\tau_{n+1} \approx t_n$
- In general an exponential averaging scheme is a good predictor if the variance is small
- Since both α and $(1-\alpha)$ are less than or equal to one, each successive term has less weight than its predecessor
- NB. Need some consideration of load, else get (counter-intuitively) increased priorities when increased load

ROUND ROBIN

A preemptive scheduling scheme for time-sharing systems.

- Define a small fixed unit of time called a quantum (or time-slice), typically 10 – 100 milliseconds
- Process at the front of the ready queue is allocated the CPU for (up to) one quantum
- When the time has elapsed, the process is preempted and appended to the ready queue

ROUND ROBIN: PROPERTIES

Round robin has some nice properties:

- Fair: given n processes in the ready queue and time quantum q, each process gets $1/n^{th}$ of the CPU
- Live: no process waits more than $(n-1)q$ time units before receiving a CPU allocation
- Typically get higher average turnaround time than SRTF, but better average response time

But tricky to choose the correct size quantum, q:

- q too large becomes FCFS/FIFO
- q too small becomes context switch overhead too high

PRIORITY SCHEDULING

Associate an (integer) priority with each process, e.g.,

Prio	Process type
0	system internal processes
1	interactive processes (staff)
2	interactive processes (students)
3	batch processes

Simplest form might be just system vs user tasks

PRIORITY SCHEDULING

- Then allocate CPU to the highest priority process: "highest priority" typically means smallest integer
  - Get preemptive and non-preemptive variants
  - E.g., SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time

TIE-BREAKING

What do with ties?

- Round robin with time-slicing, allocating quantum to each process in turn
- Problem: biases towards CPU intensive jobs (Why?)
- Solution?
  - Per-process quantum based on usage?
  - Just ignore the problem?

STARVATION

Urban legend about IBM 7074 at MIT: when shut down in 1973, low-priority processes were found which had been submitted in 1967 and had not yet been run...

This is the biggest problem with static priority systems: a low priority process is not guaranteed to run — ever!

DYNAMIC PRIORITY SCHEDULING

Prevent the starvation problem: use same scheduling algorithm, but allow priorities to change over time

- Processes have a (static) base priority and a dynamic effective priority
  - If process starved for k seconds, increment effective priority
  - Once process runs, reset effective priority

EXAMPLE: COMPUTED PRIORITY

First used in Dijkstra's THE

- Timeslots: $\ldots, t, t+1, \ldots$
- In each time slot t, measure the CPU usage of process $j : u^j$
- Priority for process j in slot $t+1$:

$$p^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$$

- E.g., $p^j_{t+1} = \frac{p^j_t}{2} + ku^j_t$
- Penalises CPU bound but supports IO bound

Once considered impractical but now such computation considered acceptable

SUMMARY

- Scheduling Concepts
  - Queues
  - Non-preemptive vs Preemptive
  - Idling
- Scheduling Criteria
  - Utilisation
  - Throughput
  - Turnaround, Waiting, Response Times
- Scheduling Algorithms
  - First-Come First-Served
  - Shortest Job First
  - Shortest Response Time First
  - Predicting Burst Length
  - Round Robin
  - Static vs Dynamic Priority