Scheduling

2.4

Scheduling

The process of determining how CPU resources will be allocated.

- Which process next?
- How long should it have the CPU?

Is it important?

- Processor rich environment for load → not so much
- Processor poor … → critical

The best scheduler…

depends on what you are looking for.

- These are always important:
  - fairness – "fair" share for each process
  - policy enforcement
  - balance – avoid unused resources
- Process type:
  - Batch – non-interactive processes
    - maximize throughput
    - minimize turnaround
    - maximize CPU usage
  - Frequently referred to as a "job" (from the days of punch card computing)

- Interactive systems
  - response time
  - proportional to expectations
- Real time systems
  - predictability
  - meeting deadlines

How do I know if my scheduler is any good?

(metrics)

- CPU Utilization

  Percent of time scheduled
  light: 40% - heavy 90%

- Throughput - # completed jobs per unit time
- Turnaround – Elapsed time between submission & completion of batch jobs
- Wait – Amount of time spent in ready queue
- Response – Time between input and *start* of output
- Proportionality – Based on user expectations (e.g. Likert scale)

- Predictability – Robust behavior for many environments

We cannot do it all....
Most general-purpose OSs optimize wait and response time

Dispatcher

THE NEXT FEW DEFINITIONS ARE THINGS YOU NEED TO KNOW…

Responsible for context switches between processes

- Interrupt occurs and dispatch service routine invoked
- Save memory management unit (MMU) data
- Save process state (e.g. registers)
- Flush and clear cache* and processor pipeline
- Load new MMU data
- Load new process state
- Return from interrupt into new process

* OS/CPU dependent

Process behavior

- CPU burst – The amount of time a process can execute before it needs I/O
- CPU burst times depend on the type of program executing, but typically have a distribution:
- I/O burst – Time spent waiting for I/O

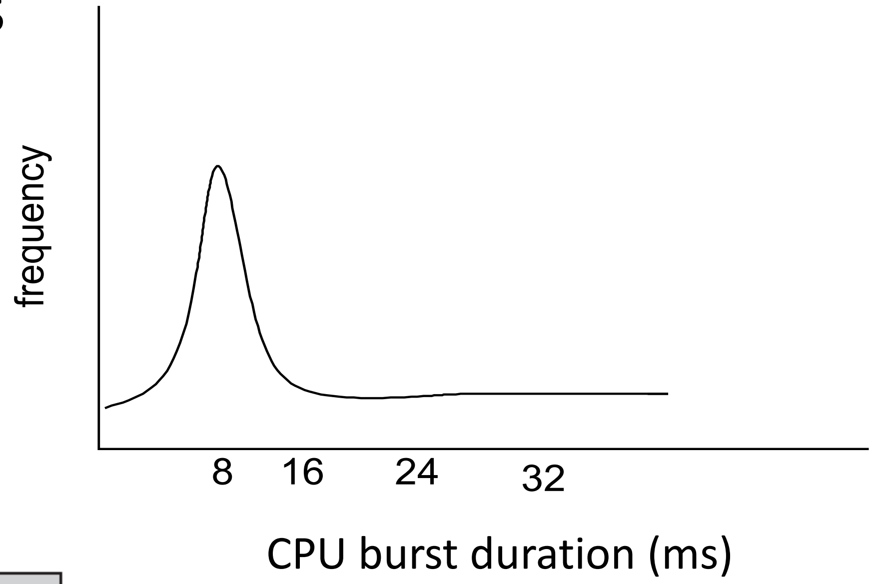
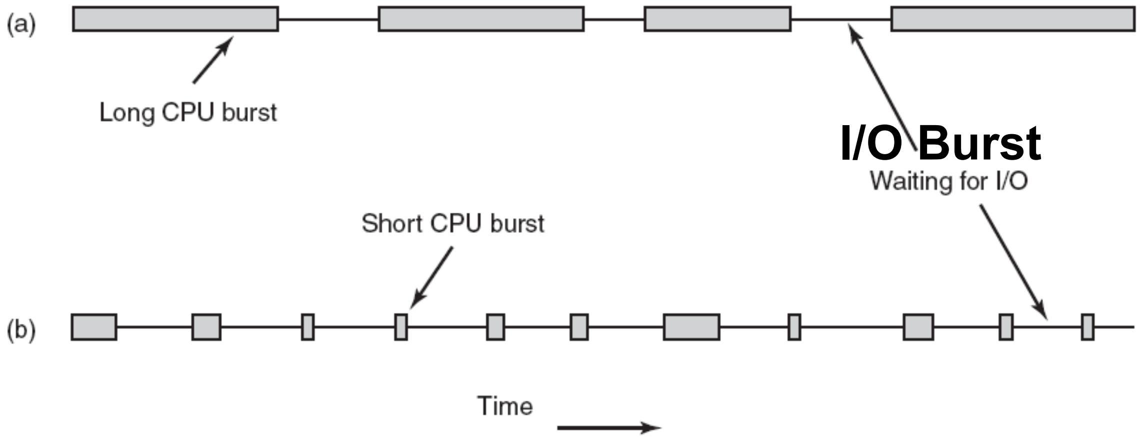
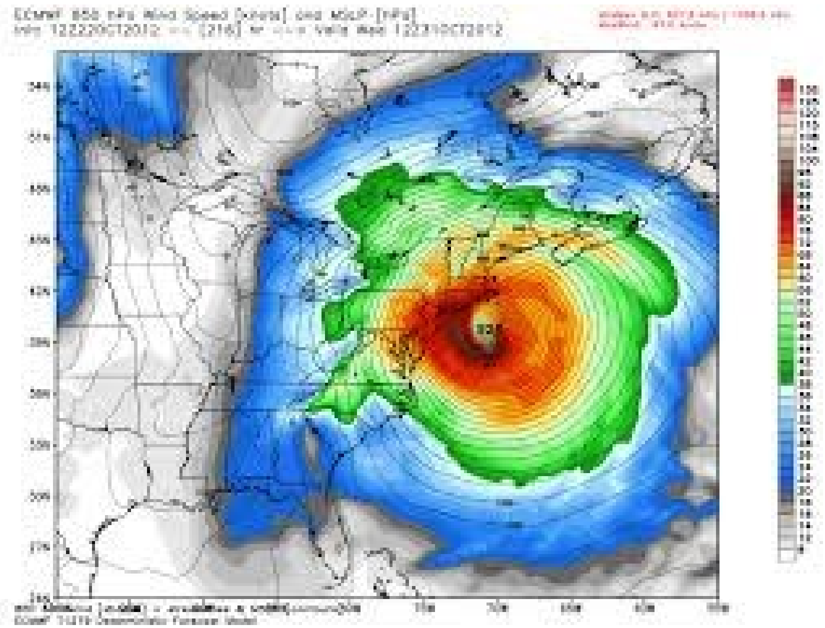


Fig. 2-39



Process behavior



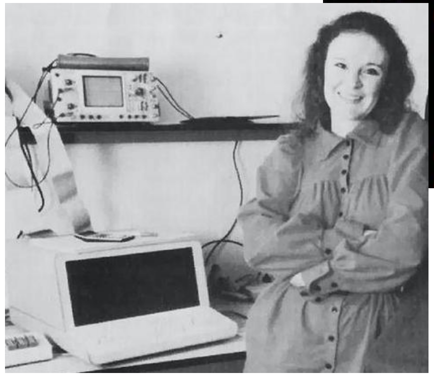
Washington Post

a) CPU bound

Spends more time computing than I/O

b) I/O bound

Spends more time handling input/output



Dona Bailey
One of the principal authors of Centipede

Preemption

- Preemption is when a CPU is pulled from running *before* its CPU burst is complete.

- Why would we want to do this?
  - Policy dictates that the process has met its time quantum, an amount of time allowed for the process to execute.

  - An event occurs and a new process is scheduled

© Sesame Workshop

Scheduling algorithms

- Classified as preemptive or non-preemptive.
- Try to avoid starvation – stuck in ready queue without being scheduled

© Sesame Workshop

Types of schedulers

- Admissions scheduler – Determines when a process can start.
  - Not commonly used on consumer operating systems.
  - Never used for interactive processes.
- Memory scheduler – Suspends/reloads processes when performance degrades due to poor performance metrics
  - Not usually used with virtual memory
  - Not usually used with interactive processes.
- Short-term scheduler – Determines which process is assigned CPU resources next.

Batch short-term schedulers

- First come first served (FCFS)
  - Non-preemptive
  - Ready processes are stored in an FCFS ready queue
  - Processes get CPU until CPU burst expires (in a multiprocessing system)

  - One job with a high CPU burst can greatly impact the average turn around time.

Batch short-term schedulers

- Shortest job first (SJF)
  - Non-preemptive
  - Ready queue is organized by known average run times for process to complete
  - Book example shows running to completion instead of scheduling CPU bursts (e.g. not a multiprocessing system)
  - One job with a high CPU burst can greatly impact the average turn around time.

Example

P_A 9 min

P_B 4 min

P_C 6 min

- Assumptions: No multiprocessing, all jobs started at same time
- Turnaround time:

first come, first serve			
P	start	complete	turnaround
P_A	0	9	9
P_B	9	13	13
P_C	13	19	19

Shortest job first			
P	start	complete	turnaround
P_B	0	4	4
P_C	4	10	10
P_A	10	19	19

- Mean turnaround?

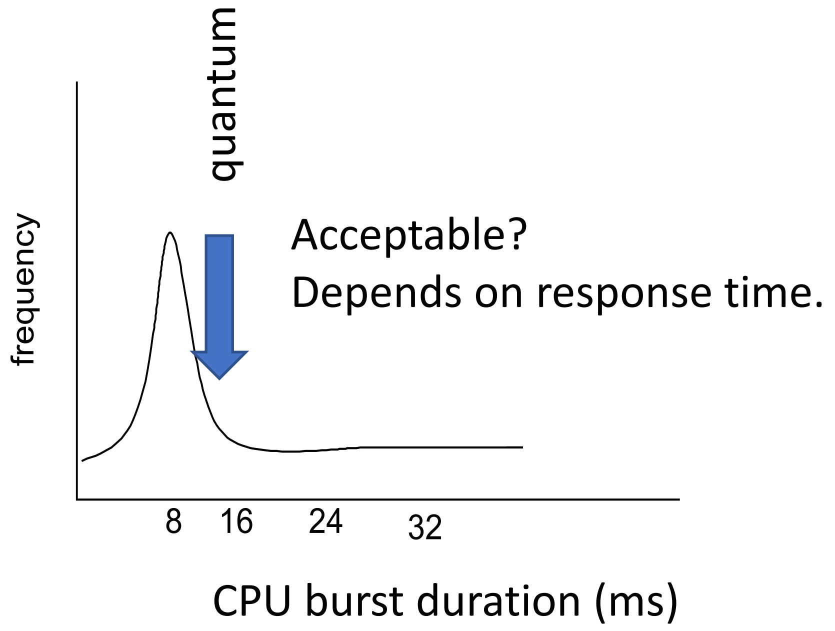
$$FCFS: \frac{9+13+19}{3} \approx 13.7 \; SJF: \frac{4+10+19}{3} = 11$$

Interactive short-term schedulers

- Round robin
  - Preemptive scheduler, each process given quantum units of time
  - Process executes until:
    - CPU burst terminates, or
    - a quantum timer expires

  What happens to the process in each case?

  - How long should quantum be?
    - Too long: Poor response time
    - Too short: Waste time in context switches



Interactive short-term schedulers

- Priority scheduler
  - Each process assigned a priority level (number)
  - Processes in ready queue scheduled by priority
  - Ties broken by policy rule, e.g. FCFS
  - Preemptive: Arrival of a higher priority process can displace an executing process

  - Assignment of priority is a policy decision. Sample policies:
    - Sales team processes > developer team ☹
    - Set priority as a function of the duration of the last CPU burst (example of dynamic priority)

Interactive short-term schedulers

- Priority scheduler
  - Processes subject to starvation
  - Aging: Method for preventing starvation

```
while (true) {
   delay for a time;
   increase the priority of each P in ready queue
}
```

© Sesame Workshop

Interactive short-term schedulers

- Shortest process next
  - Special case of priority scheduling
  - Priority inversely proportional the length of the *next* CPU burst

The next shortest process is…

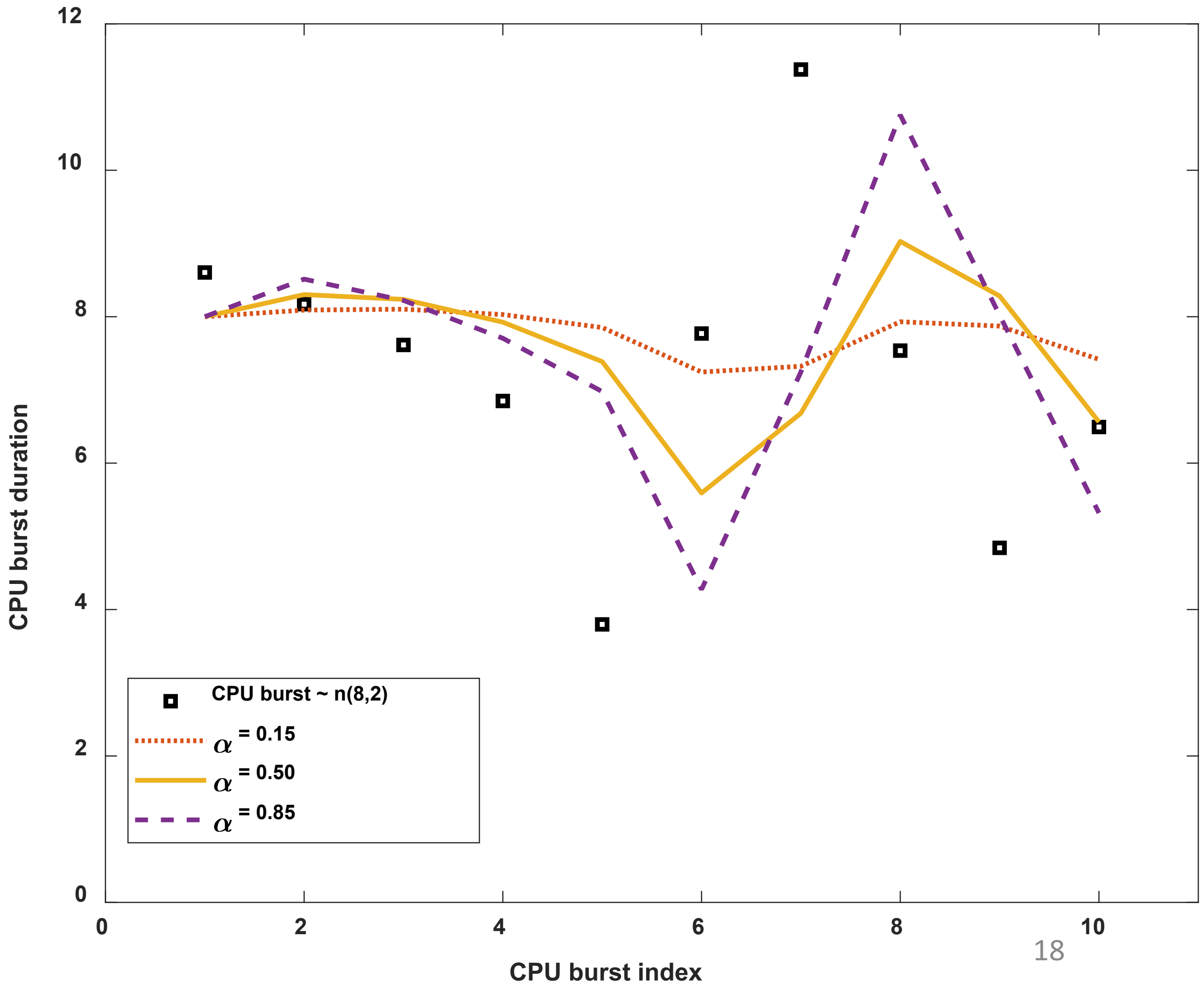
Johnny Carson as Carnac

Interactive short-term schedulers

- Shortest process next
  - Predicting the future with *exponential averaging*

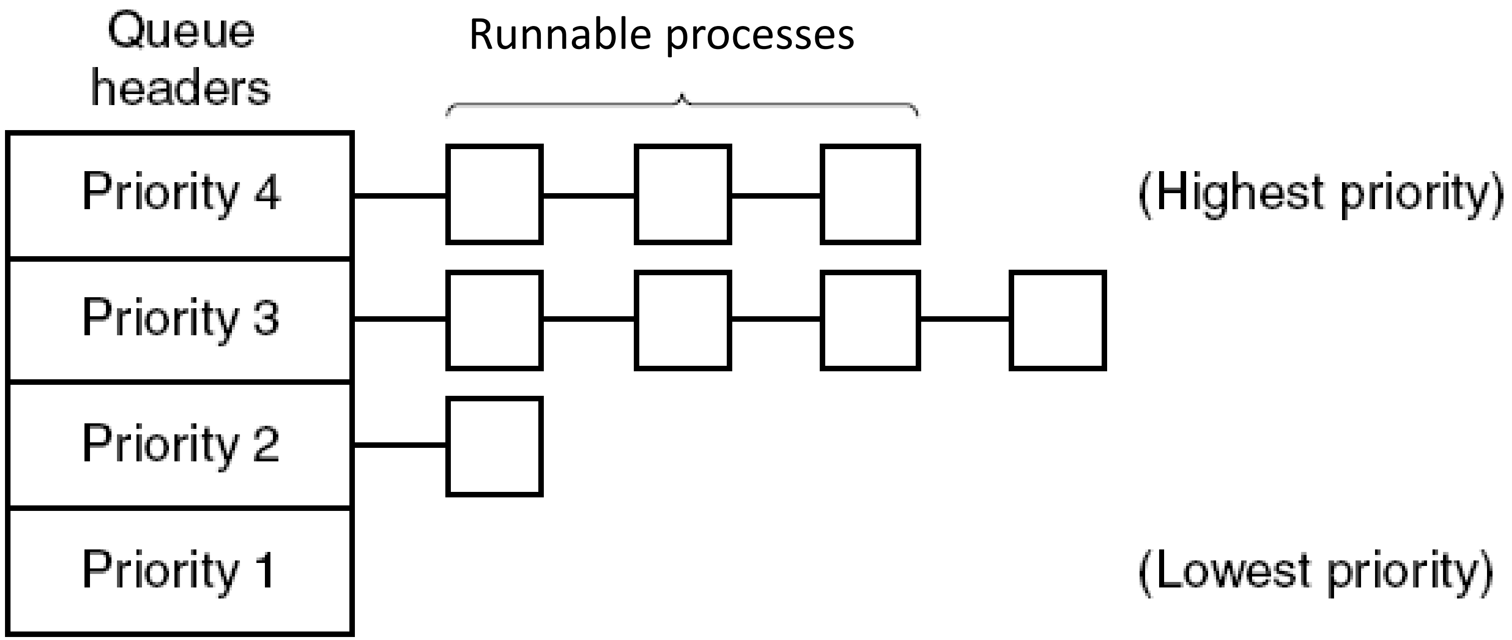
$$p(n) = \begin{cases} \mu_{CPU\ burst} & n = 1 \\ \alpha \cdot b(n-1) + (1-\alpha) \cdot p(n-1) & n > 1 \end{cases}$$

  - $p(n)$ – prediction of nth CPU burst
  - $\mu_{CPU\ burst}$ - average CPU burst
  - $b(n)$ – nth CPU burst
  - Weight $\alpha \in (0,1)$



Interactive short-term schedulers

- Multiple queues
  - Each queue has a possibly different scheduling policy
  - Queue scheduling
    - Only when higher priority queues empty, or
    - Time share the queues.
    - Common to allow longer scheduling for low priority queues.



also known as multi-level scheduling

Interactive short-term schedulers

- Guaranteed scheduling (preemptive)
  - Processes will receive $1/n^{th}$ of the CPU
  - Must track history of process and schedule appropriately
- Fair share scheduling (preemptive)
  - Similar to guaranteed scheduling
  - Switches fairness from process-centric to user-centric
  - Each user gets 1/n’th of CPU that is shared amongst their processes
- Lottery scheduling
  - Stochastic scheduling algorithm – Non-deterministic
  - Each process given f(N) lottery tickets
  - Winner gets scheduled
  - Processes can share tickets

Real-time systems

Ansys.com

- Real-time systems are for processes that expect high responsiveness
- Two types:
  - Hard: Processes request CPU time from OS. If granted, we guarantee that the demand will be met
  - Soft: OS tries “really hard”
- When might you use each type of system?

Real-time systems

- Most real-time scheduling is in response to external events
  - aperiodic – We do not know when they will happen
  - periodic – Occur regularly, e.g. 10 cycles/s (10 Hz)
- Soft-real time scheduling
  - Real-time processes assigned high priorities
  - Frequently given separate queue
- We will not discuss hard real-time scheduling in detail, but we will discuss committing periodic events

Hard-real time periodic events

- Commitment issues
  - Given event *x* that occurs every P_x ms (period) and requires C_x ms (cost) of CPU time, can we schedule this?

- Depends on:
  - Overhead for operating system and any other processes that cannot be preempted.
  - What other periodic tasks have we already committed?

Hard-real time periodic events

Automobile with 3 real-time systems:

Task	Frequency (Hz)	Time (ms)
Antilock brake system (ABS)	60	4
Collision detections	10	15
Night vision display	24	1

Hard-real time periodic events

- Convert frequency to duration / event

  e.g. $60\ Hz = 60\frac{cycles}{s} \rightarrow \frac{1\ s}{60\ cycles} \approx .1667\ s/cycle$ P_{ABS} = 16.67 ms

Task	P_{Task} (ms)	C_{Task} (ms)
Antilock brake system (ABS)	16.67	4
Collision detections	100.00	15
Night vision display	41.67	1

- In this system, 8% of the time is spent on system tasks.

Hard real-time periodic events

- Is this system schedulable? $$\sum_{i=1}^{N} \frac{C_i}{P_i} + Overhead \leq 1$$

Task	P_{Task} (ms)	C_{Task} (ms)
Antilock brake system (ABS)	16.67	4
Collision detections	100.00	15
Night vision display	41.67	1

Overhead = 0.08

$$\underbrace{\frac{4}{16.67} + \frac{15}{100} + \frac{1}{41.67}}_{ABS+collision+night\ vision} + .08 \leq 1$$

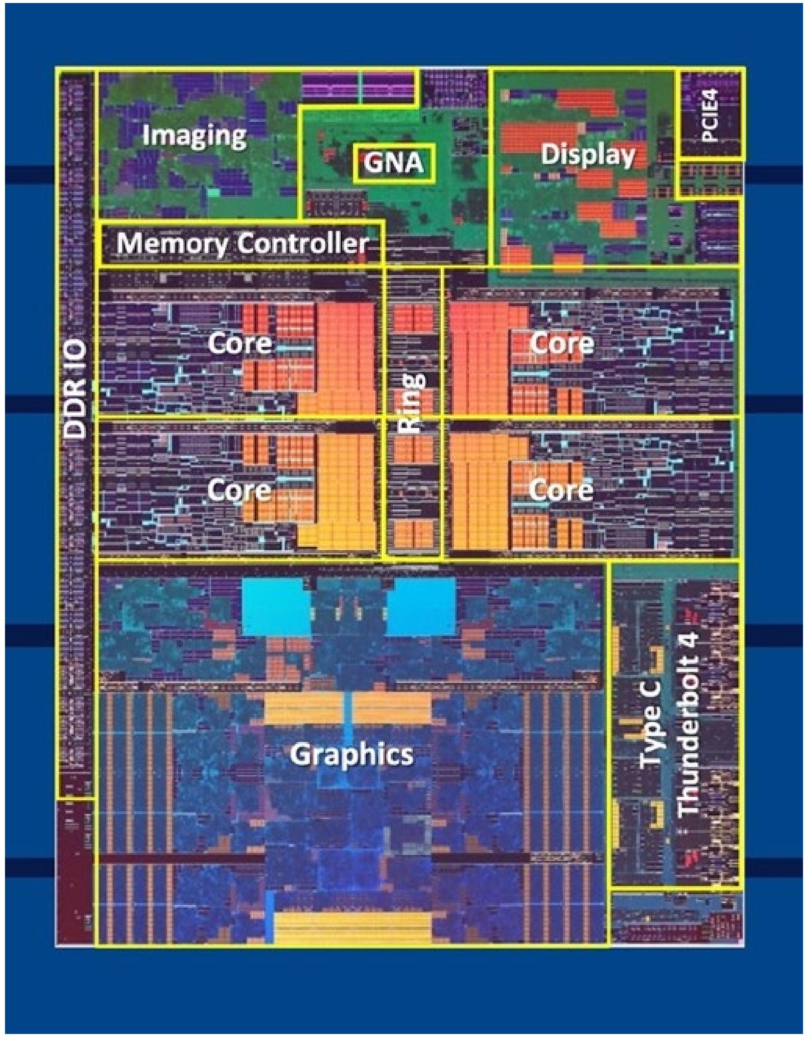
=.494 Schedulable with room to spare!

Multiprocesor scheduling

- Simplest solution
  - Only one OS thread/process is responsible for selecting in a global queue Requires a critical section:
    - only one process can access queue at a time
    - *tranquilo* (rest easy), we study this later…
  - ✓ Load balancing easy
  - ✓All the CPUs are timeshared…
  - X Contention for the ready queue

Multiprocesor scheduling

- Smart scheduling – If user process in critical section, let it run until it exits
- Affinity scheduling – Try to schedule processes on same CPU (and hope that the cache is still valid)
- Two-level scheduling
  - Assign thread to CPU with lowest load
  - Each CPU has its own scheduler



Intel Willow Cove architecture (ca. 2021)

Large-scale multiprocessor scheduling

- On large systems (e.g. global climate models with thousands of CPUs), different strategies are needed
  - Goals: Increase time that communicating processes/threads are running in parallel
  - Space sharing
    - Threads are assigned to individual cores
    - Wastes time when in I/O burst as no other thread to run
  - Time and space sharing: gang scheduling
    - Quantum divides CPUs by time
    - Related threads, or “gangs,” are assigned to individual cores
    - When a gang-member blocks, we do not start the next thread

I Am a Fugitive From a Chain Gang 1932 © David Meeker