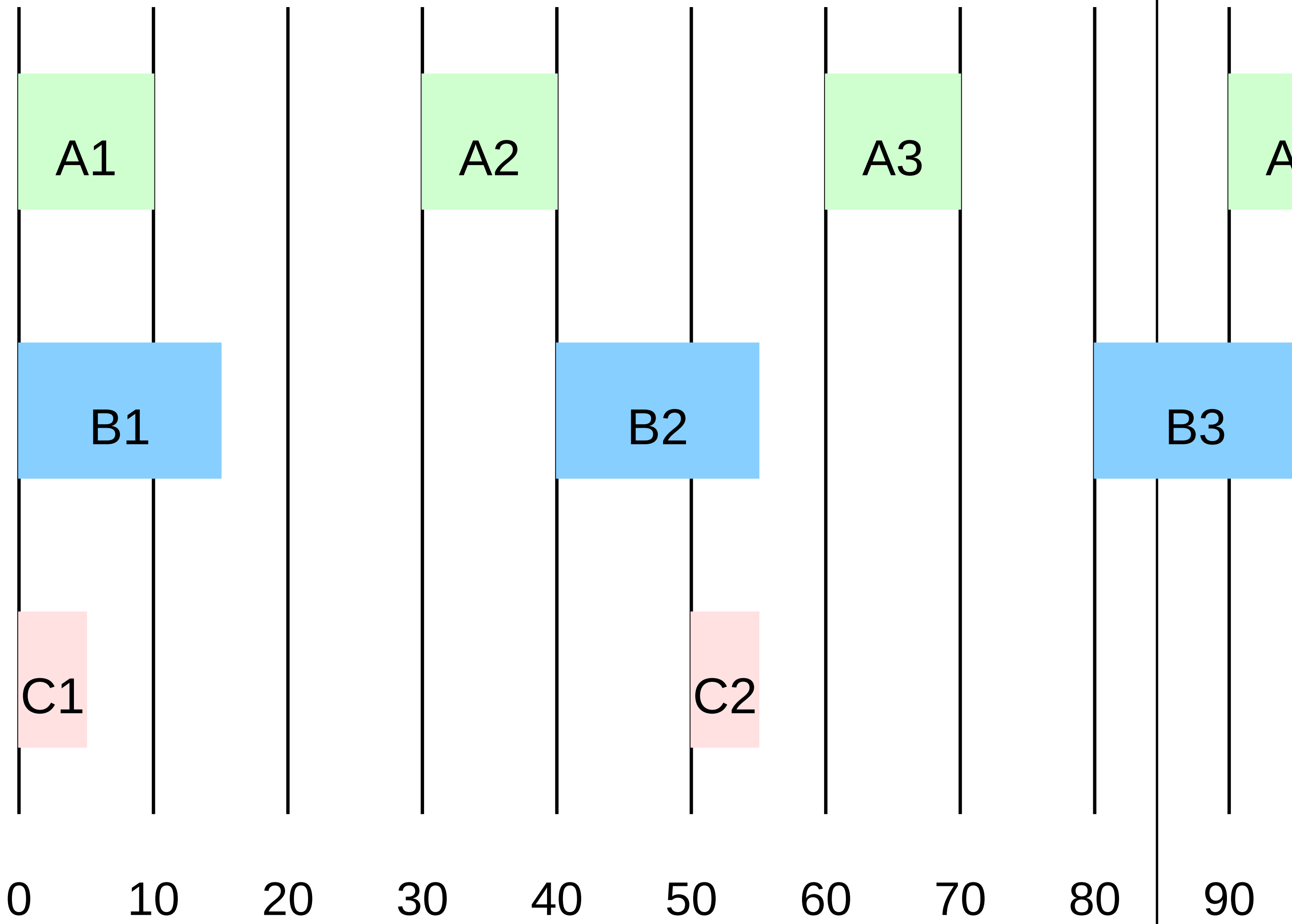
Multiple Real-Time Processes

- A runs every 30 msec; each time it needs 10 msec of CPU time
- B runs 25 times/sec for 15 msec
- C runs 20 times/sec for 5 msec
- For our equation, A uses $10/30$ of the CPU, B uses $15/40$, and C uses $5/50$; that's about 81%

Diagram



$A1$ must finish before $A2$ starts, $B1$ before $B2$,...

3 / 39

Other Issues

- Some real-time systems permit preemption; some do not
- Desirability depends on system type (text's discussion is for multimedia system, which are usually preemptible
- May have aperiodic processes in the mix
- Static or dynamic scheduling

Rate Monotonic Scheduling

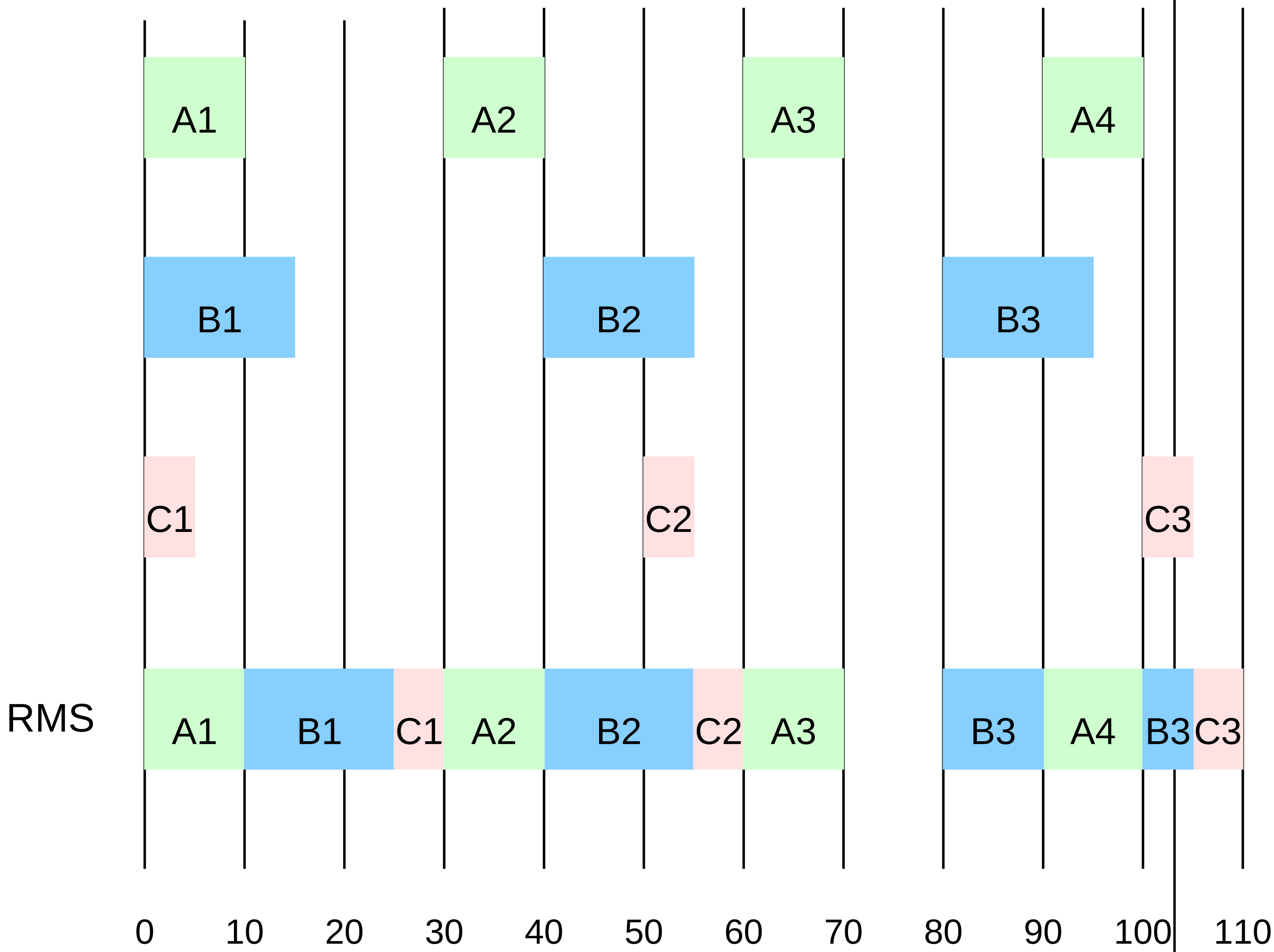
- A static scheduling algorithm by Lieu and Layland (1973)
- Conditions:
  1. Each periodic process completes within its slot
  2. No interprocess dependencies
  3. Each process needs the same amount of CPU each time
  4. Non-periodic processes have no deadlines
  5. Preemption happens instantly with no overhead
- Yes, this is an oversimplified model. . .

Algorithm

- Assign a process priority equal to its frequency:
  $A = 33$, $B = 25$, $C = 20$
- Always run the highest-priority runnable process
- Thus, A can preempt B or C; B can preempt C
- Proved optimal among class of static algorithms

6 / 39

RMS



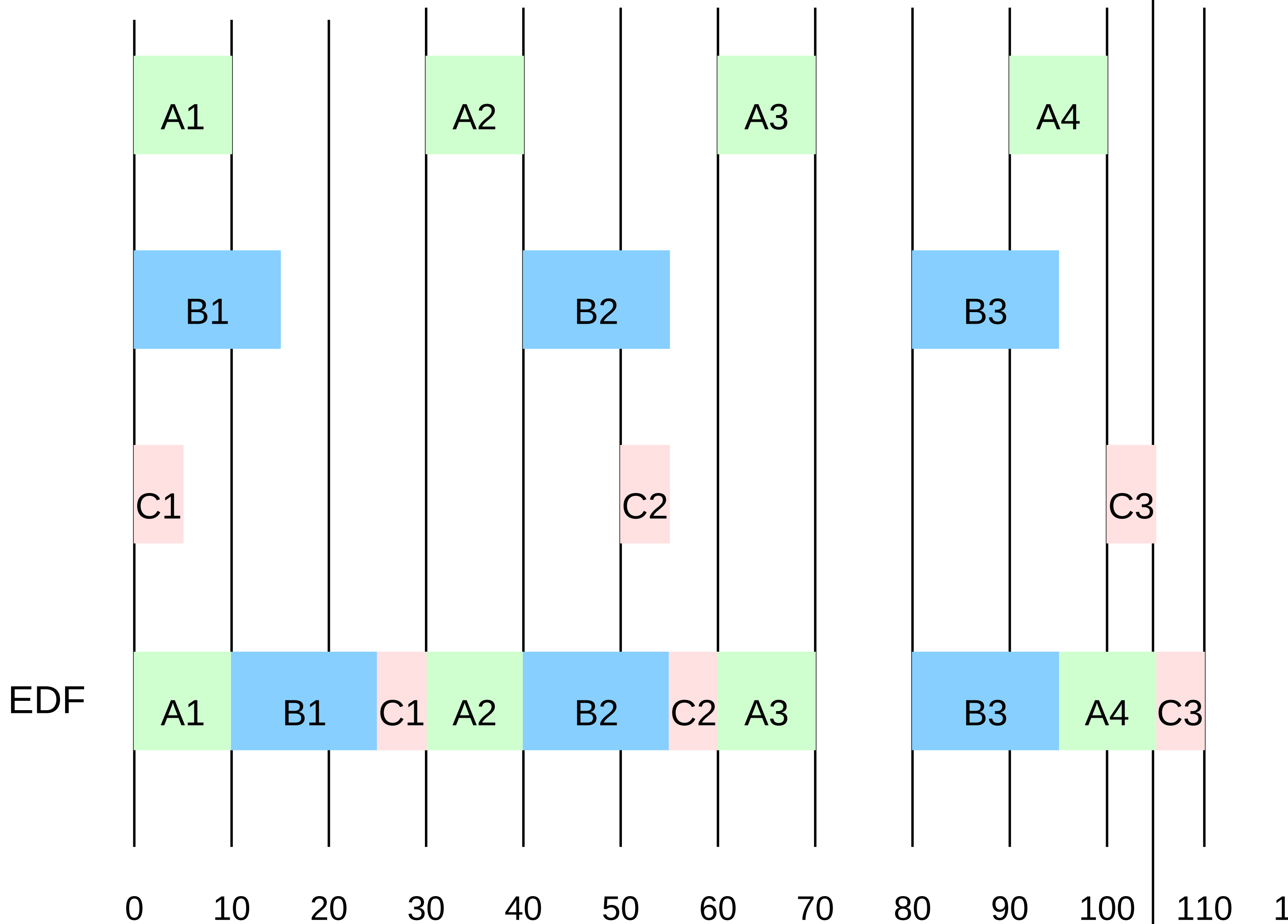
Note that $B3$ is preempted to let $A4$ run

Earliest Deadline First

- More general model
- Supports aperiodic events, non-identical CPU bursts
- Dynamic priority assignment
- When a process starts, it announces its deadline
- Priorities are assigned in order of deadline
- Initially, A goes first, because it has to finish by $T = 30$; B's deadline is $T = 40$ and C's is $T = 50$

8 / 39

EDF

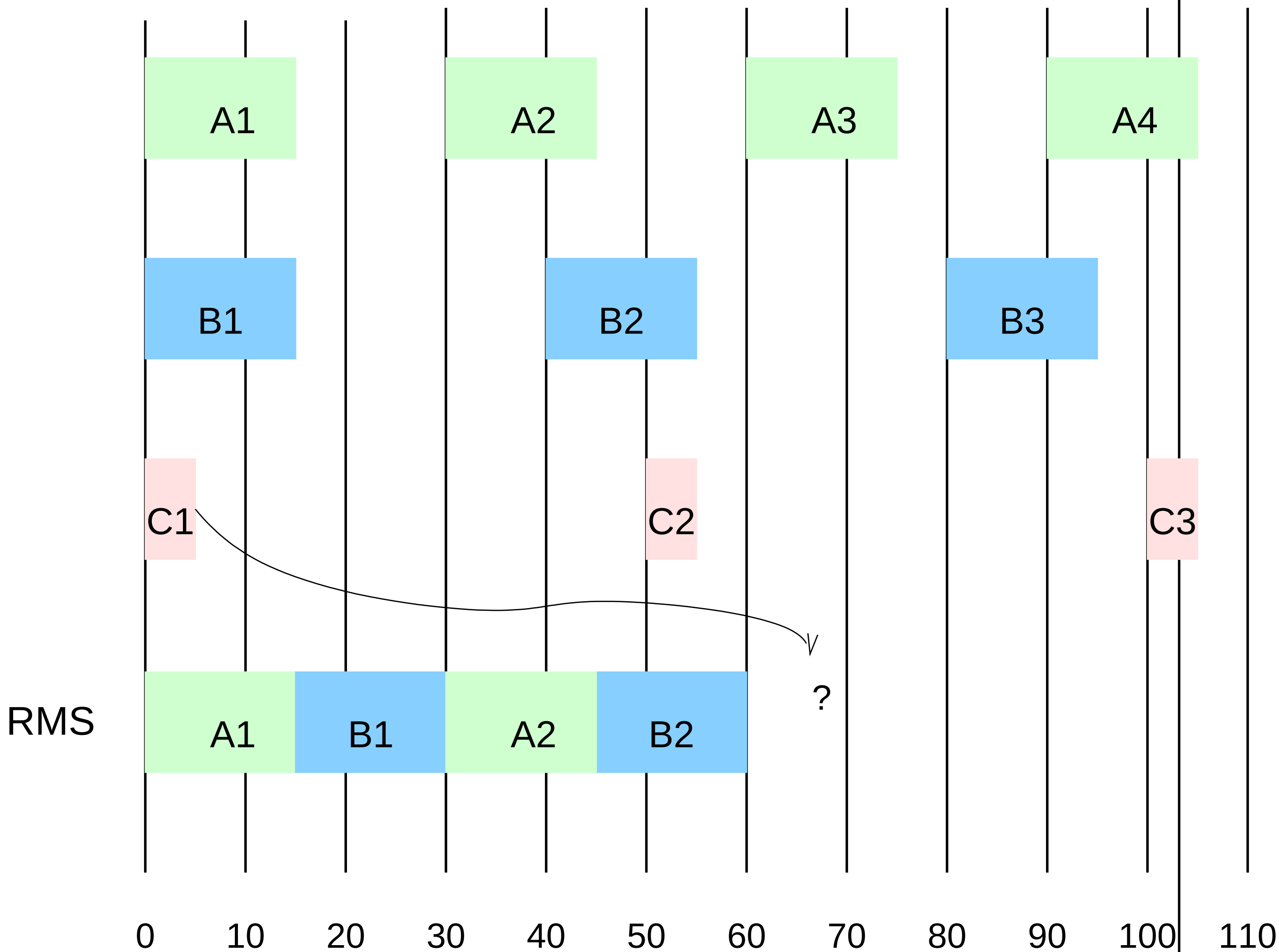


At $T = 80$, it gives B priority

RMS Doesn't Always Work

- Suppose that A needs 15 msec each time
- Our formula says we're ok: $(15/30 + 15/40 + 5/50) = 97.5\%$
- But it fails
- $A1$ runs from $T = 0$ to $T = 15$; $B1$ runs from $T = 15$ to $T = 30$
- At that point, $A2$ is ready, and has a higher priority than $C1$; $B2$ follows it
- There's no time for $C1$ before $C2$ has to start

RMS Failure



$C1$ can't run before $C2$ has to start

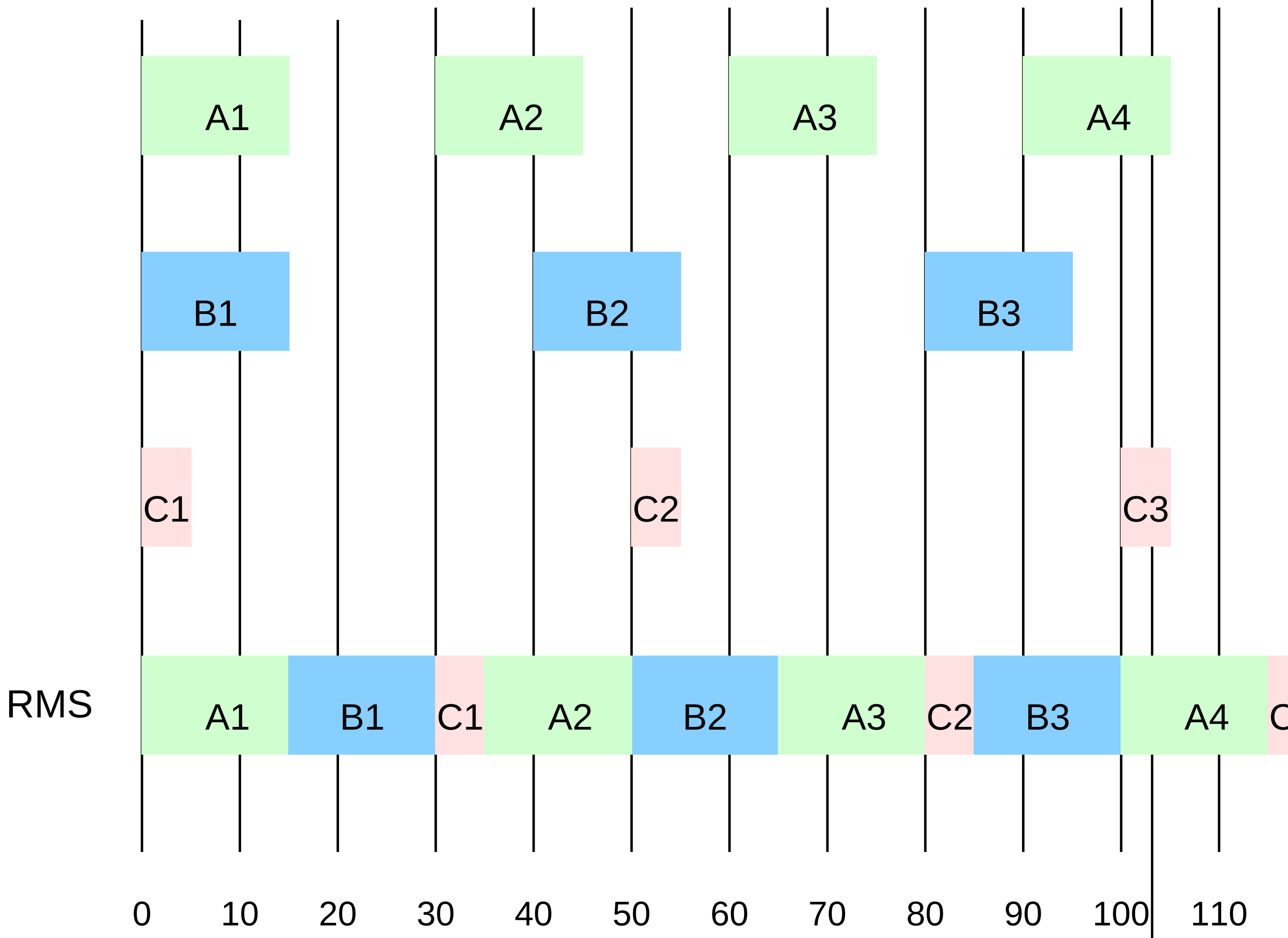
Why Did it Fail?

- RMS is only guaranteed to work if

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

- As $m \to \infty$, utilization approaches $\ln 2 = 0.693$
- For $m = 3$, it can fail (though won't always) at 78%
- Maximum allowed utilization goes down as m increases
- EDF will succeed for this example
- The CPU idle period — 2.5% — will occur every 200 msec

EDF Succeeds



Note that $C1$ runs before $A2$

Gantt Charts

- The diagrams I've been using are called *Gantt Charts*
- Useful tool for modeling process scheduling
- Especially useful with an automated tool

14 / 39

Other Issues

15 / 39

Scheduling Threads

- With user-level threads, there is no interaction with the scheduler
- A kernel-level thread implementation relies on the system's scheduler
- It's often beneficial to schedule several threads from the same process consecutively — avoid changes to the memory map
- Similar logic says keep threads from the same process on a single CPU, if feasible
- Application-level thread scheduler can handle priorities more easily, though kernel-level priorities aren't hard to set

16 / 39

Scheduling and Multiprocessor Systems

- What processes run on which processor?
- Does it matter?
- What are the implications?

17 / 39

Asymmetric Multiprocessing

- All kernel functions are handled by the master CPU
- The only thing the other CPUs do in the kernel is pull processes off the run queue and do context switches
- Simplifies OS design — locking is much simpler
- Common first step in OS conversion for multiprocessor use

18 / 39

Symmetric Multiprocessing

- Each processor can do anything
- Possible to have more than one CPU in the kernel simultaneously
- Need fine-grained locking; single "big lock" is almost the same as asymmetric MP
- Locking is a very important issue for multiprocessors; we'll discuss this more on Wednesday

Processor Affinity

- Sometimes, it's better (or necessary) for a given process to execute on a specific CPU
- Example cited earlier: memory map (and cache)
- I/O issues — sometimes a specific I/O devices is on a local bus

How Do We Measure CPU Time?

- Early clocks were low resolution
- Unix classic: 60 Hz
- Too coarse to measure a fraction of a quantum
- Besides, the clock was an I/O device, hence slow to access

21 / 39

Statistical Time

- At each clock tick, add 1 to the current process' CPU counter
- Actually, two counters, one for user mode and one for kernel mode
- Not accurate over short periods:
  - A process may run for too short a time and not get charged
  - A process may start its quantum right before the timer tick and be charged too much
- Statistically, though, it's good enough

22 / 39

I/O and Memory

- CPU time isn't the only scarce resource
- Especially today, total system performance is limited by I/O bandwidth and memory availability
- Must read programs in from disk
- Must have memory for them
- That may mean paging out another process, which puts more load on the disk
- The scheduler *should* interact with the I/O subsystem and the memory subsystem

23 / 39

Different Schedulers 24 / 39

Scheduler Algorithms

- Modern systems may have many processes running
- At this instant, for example, `cluster` is running 525 processes
- Even on modern CPUs, we don't want scheduling algorithms that iterate over all processes
- Linux uses a O(1) scheduler — scheduling decisions take constant time, regardless of the number of processes

25 / 39

Policy versus Mechanism

- Put some basic mechanism (or mechanisms) in the kernel
- Permit user processes to set parameters that control scheduling
- Simplest example: `nice` command
- Solaris permits much more control: three classes of scheduler, and parameters within that class

Solaris Scheduler

- Scheduler classes: real-time, time-sharing, interactive
- Parameters:

  Real-Time priority, quantum
  Kernel threads (System only)
  Timesharing priority, priority limit
  Interactive priority, priority limit

- Newer versions of Solaris have fair-share scheduling and fixed-priority scheduling

Solaris Priorities

Priority	Quantum	New Priority	After Sleep
0	200	0	50
5	200	0	50
15	160	5	51
		...	
50	40	40	58
55	40	45	58
59	20	49	59

Higher priority processes get shorter quanta. Process get a lower priority when they use up their quantum, and higher priority after blocking for I/O.

Linux Scheduler

- Both real-time and priority
- Real-time scheduling can be round-robin or FCFS
- Dynamic timeslice (quantum) computation
- Kernel preemption possible if no locks are held

Windows XP

- Real-time and priority scheduling
- Priority classes: Real-time, High, Above Normal, Normal, Below Normal, Idle
- Relative priority within class: Time-critical, Highest, Above Normal, Normal, Below Normal, Idle
- Effective priority calculated from this matrix
- Priority lowered on quantum expiration
- Extra priority boost for process associated with current window
- Unix can't easily do that — the window manager knows nothing of processes

30 / 39

Evaluating Scheduler Algorithms

31 / 39

Algorithm Evaluation

- First question: what criteria do you want to optimize for?
- Possibilities include CPU utilization, responsiveness, real-time scheduling, etc.
- Several ways to do the evaluation

32 / 39

Deterministic Modeling

- Start with a specific workload, i.e., of process' CPU demands and arrival times
- Model them with Gantt charts, as we've seen
- Evaluate according to desired metric
- Simple and fast — and only useful for loads that look a lot like what you model

33 / 39

Queueing Theory

- Start with probability distributions of CPU requests, arrival times, etc.
- Common assumption: arrivals are distributed according to a Poisson distribution
- Sample (and simple) result:

  Let n be the average queue length, W the average queue wait time, and λ the mean interarrival time (regardless of distribution). Then

$$n = \lambda \cdot W$$

34 / 39

Simulation

- Build a simulator of the scheduling algorithm
- Feed in simulated inputs and see what happens
- Simulations can be fed by probability distributions or by trace data from real systems
- Such trace data is an excellent way to compare two different simulators

Metacomment

- Trace data is *always* useful
- Instruction traces, network packet traces, CPU load traces, etc.
- Some such datasets become *the* way to evaluate new schemes

Build It and Try It

- Build a real system and see what happens
- But what's the load? Real users?
- If you're lucky, you have trace data to feed in (and a system amenable to such replays)

Limits of Evaluation

- Load changes
- Sometimes, load changes *because* of scheduler changes
- Example: if a process gets the CPU more quickly after a disk I/O request, it may be able to issue the next requiest within the rotational delay of the disk
- Conclusion: we need flexible scheduling algorithms that can be tuned at each site

Summary

- Scheduling is a complex matter
- The criteria and algorithms have changed somewhat, but the problem remains
- Any time you click on something and it doesn't respond immediately, there's a scheduler problem

39 / 39