CS 134:
Operating Systems
Scheduling

CS34

2013-05-17

CS 134:
Operating Systems
Scheduling

Process Switching

Class Exercise

When can/do we switch processes (or threads)?

Process Switching

Class Exercise

When can/do we switch processes (or threads)?

Process Switching

We could switch processes any time the OS has control, i.e.,

- Interrupt occurs
  - Clock
  - I/O interrupt
  - Page fault
- Trap occurs
  - Trace
  - Protection fault
- System call
  - I/O request
  - Wait for child
  - *etc.*

Process Switching

We could switch processes any time the OS has control, i.e.,
- Interrupt occurs
  - Clock
  - I/O interrupt
  - Page fault
- Trap occurs
  - Trace
  - Protection fault
- System call
  - I/O request
  - Wait for child
  - *etc.*

Process Switch Overheads

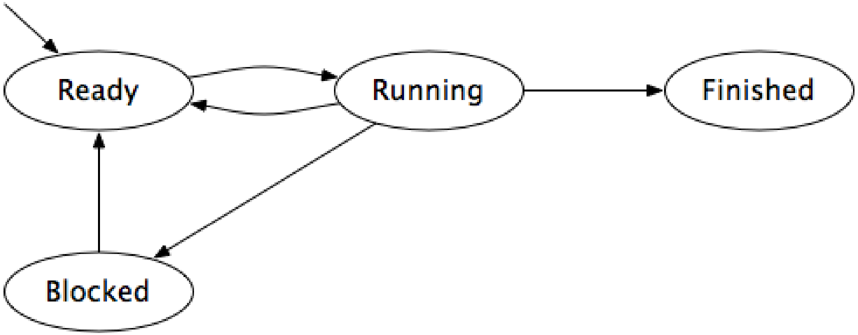
To switch processes, system must

- Save the state of the old process
- Load the saved state for the new process

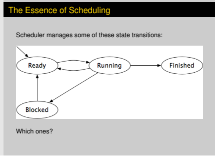
Process Switch Overheads
To switch processes, system must
- Save the state of the old process
- Load the saved state for the new process

The Essence of Scheduling

Scheduler manages some of these state transitions:



Which ones?



Scheduling Goals

Many different scheduling algorithms

- Tradeoffs
- Different goals ⇒ Different choices

What are some possible goals for a scheduler?

- What could we try to optimize?

Scheduling Goals

Many different scheduling algorithms
- Tradeoffs
- Different goals ⇒ Different choices

What are some possible goals for a scheduler?
- What could we try to optimize?

Scheduling Exercise

Consider the following set of running processes

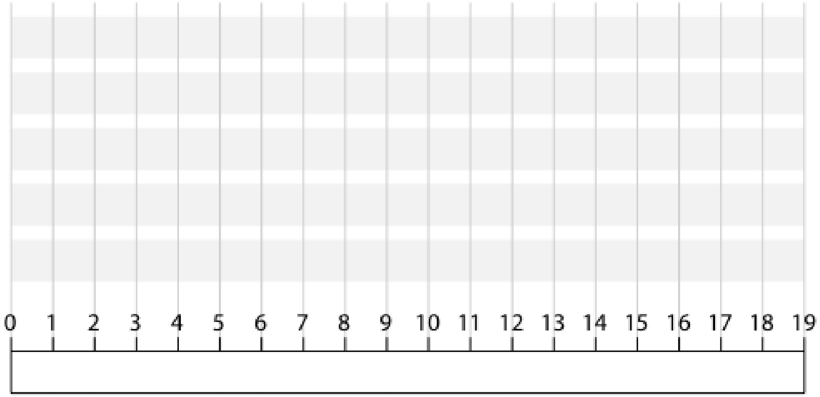
Process	Arrival Time	Burst Time	Priority (*if applicable*)
A	0	10	3
B	0	1	1
C	0	2	3
D	0	1	4
E	0	5	2

Scheduling Exercise

Consider the following set of running processes

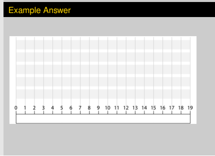
Process	Arrival Time	Burst Time	Priority (*if applicable*)
A	0	10	3
B	0	1	1
C	0	2	3
D	0	1	4
E	0	5	2

Example Answer

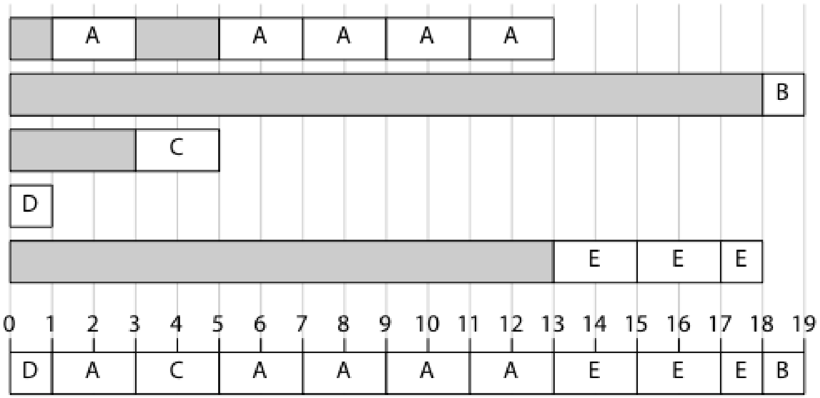


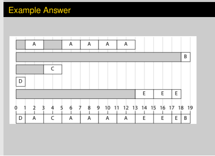
2013-05-17

CS34
└Scheduling
└Example Answer



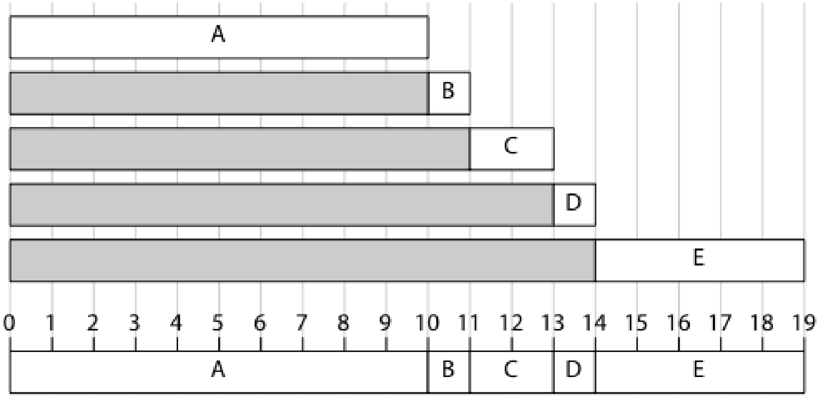
Example Answer

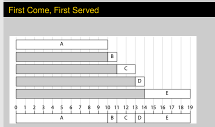




This is a priority-based round-robin scheduler with a 2-second (well, 2-unit) time slice, where higher numbers are higher priorities. All processes arrive at time 0, so D runs first (priority 4). Then A and C alternate; C quickly finishes so A hogs the CPU until it's done. Then E runs exclusively, followed by B.

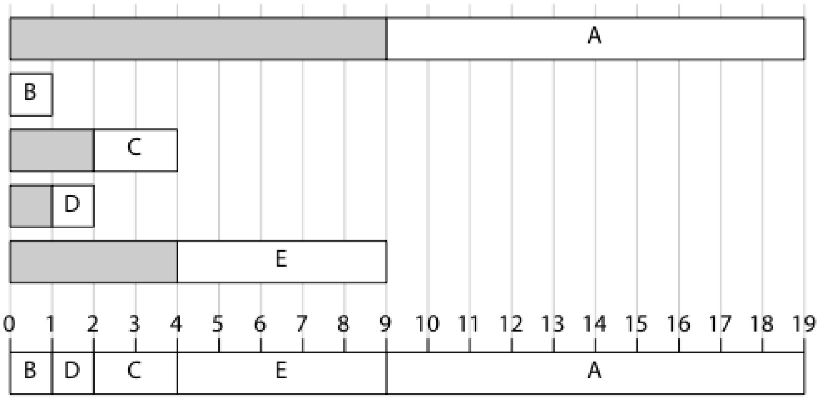
First Come, First Served

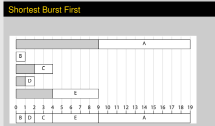




We assume that although the processes all arrive at time 0, they arrive in alphabetical order. Simple.

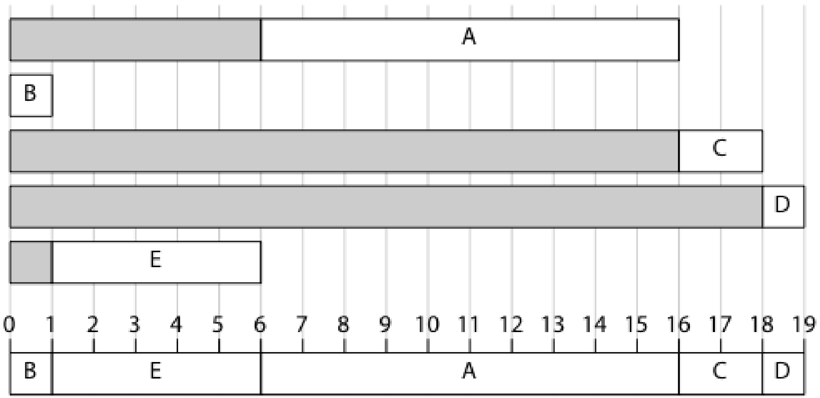
Shortest Burst First





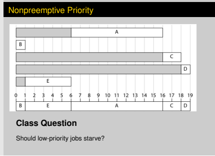
Within each burst leave, it’s FCFS.

Nonpreemptive Priority



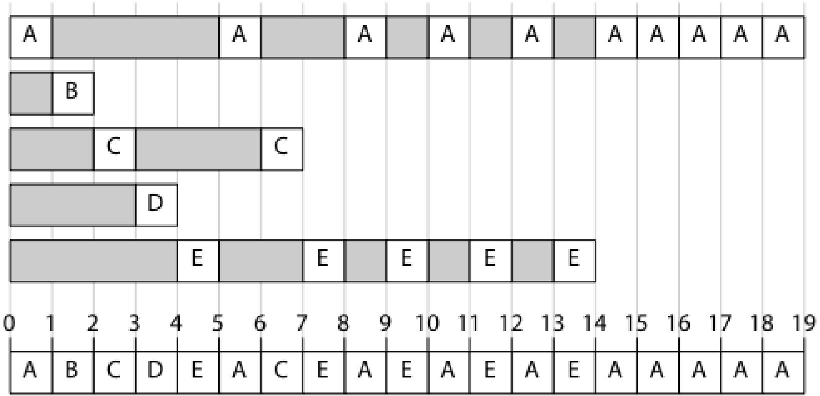
Class Question

Should low-priority jobs starve?



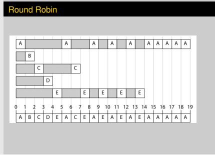
Note that here, **low** numbers mean high priority. Urgh! So we run B, then E, then A, C, D in that order. It's FCFS sorted by priority.

Round Robin

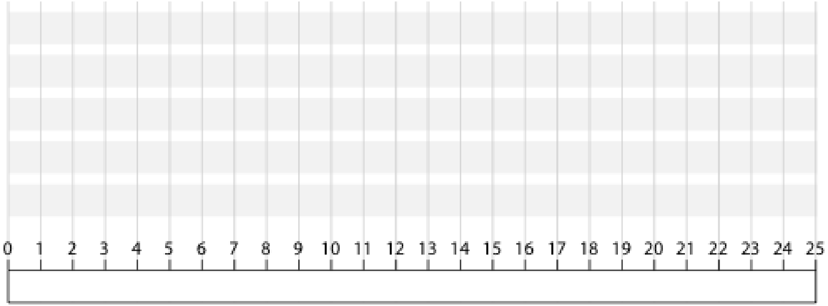


2013-05-17

CS34
└ Scheduling
└ Round Robin

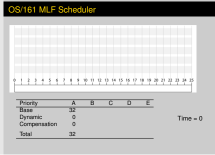


OS/161 MLF Scheduler



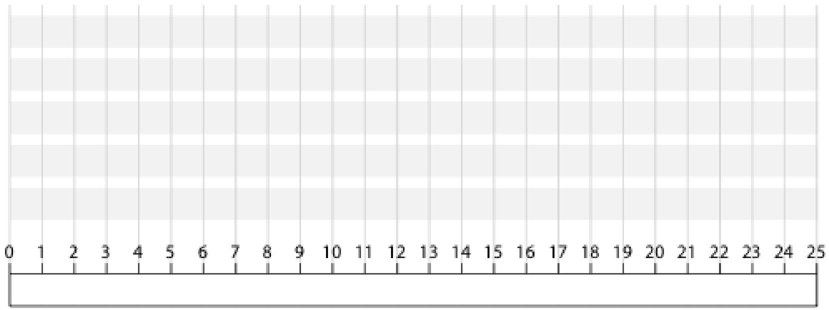
Priority	A	B	C	D	E
Base	32				
Dynamic	0				
Compensation	0				
Total	32				

Time = 0



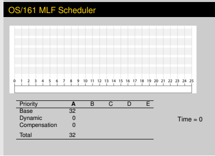
MLF == Multi-Level Feedback. Three parameters are summed: base priority (niceness, -256 to 256); dynamic priority (decremented by delta of 16 when stopped by a clock interrupt, incremented by 16 when blocks or yieds); and compensation priority (set to 0 when scheduled, incremented whenever passed over).

OS/161 MLF Scheduler



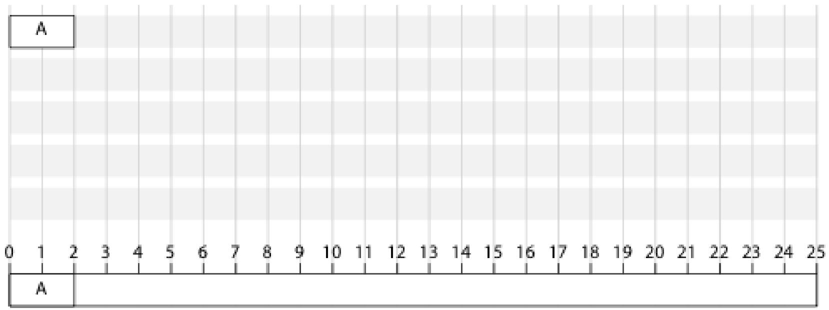
Priority	**A**	B	C	D	E
Base	32				
Dynamic	0				
Compensation	0				
Total	32				

Time = 0



Initially A is the only process, so it is chosen to run (boldface).

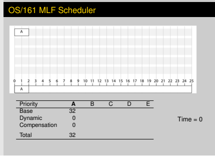
OS/161 MLF Scheduler



Priority	**A**	B	C	D	E
Base	32				
Dynamic	0				
Compensation	0				
Total	32				

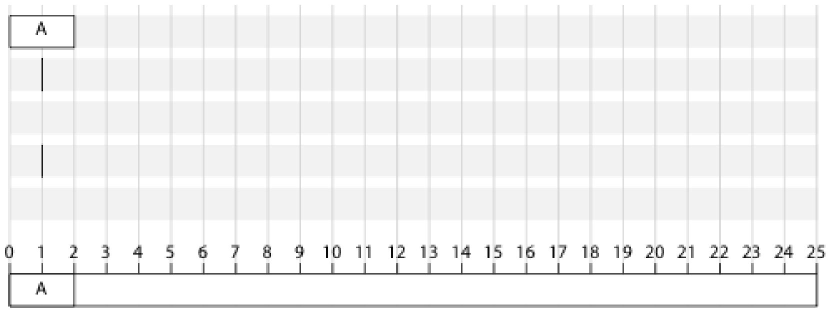
Time = 0

2013-05-17



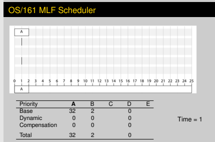
A begins running. It has a 2-unit time slice.

OS/161 MLF Scheduler



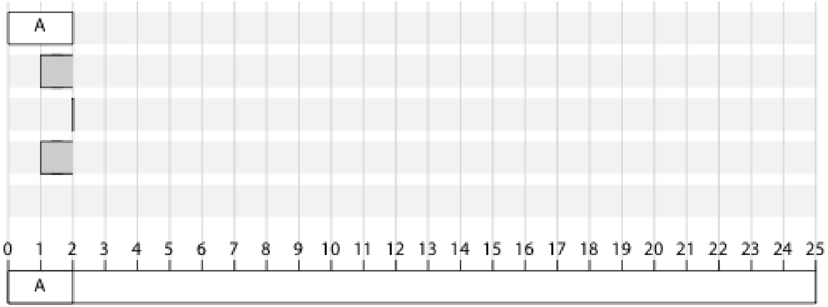
Priority	**A**	B	C	D	E
Base	32	2		0	
Dynamic	0	0		0	
Compensation	0	0		0	
Total	32	2		0	

Time = 1



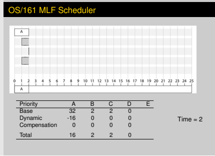
B and D arrive, with different base priorities.

OS/161 MLF Scheduler



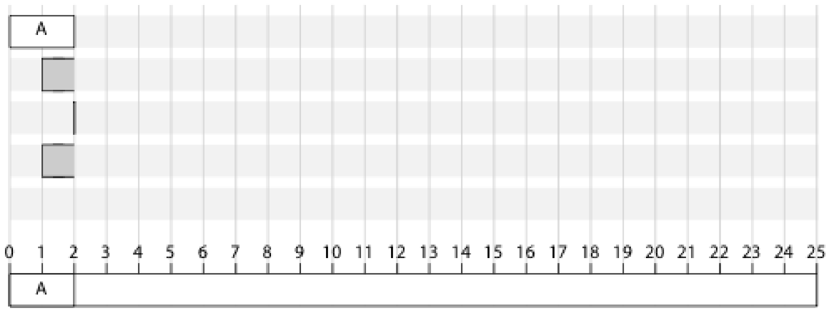
Priority	A	B	C	D	E
Base	32	2	2	0	
Dynamic	-16	0	0	0	
Compensation	0	0	0	0	
Total	16	2	2	0	

Time = 2



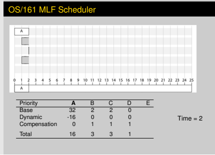
A’s first timie slice expires, so its dynamic priority is reduced. But it’s still highest.

OS/161 MLF Scheduler



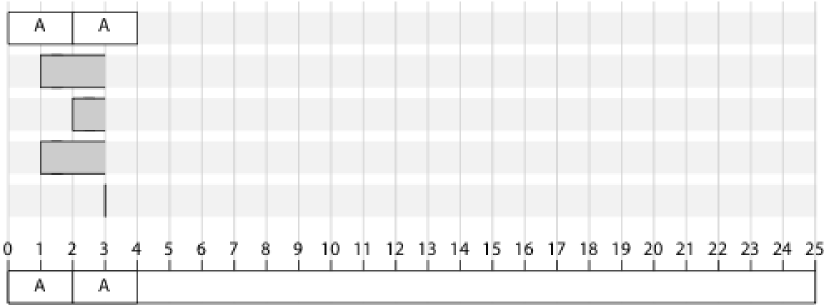
Priority	**A**	B	C	D	E
Base	32	2	2	0	
Dynamic	-16	0	0	0	
Compensation	0	1	1	1	
Total	16	3	3	1	

Time = 2



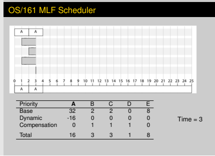
A continues to run. Everybody else gets compensation.

OS/161 MLF Scheduler



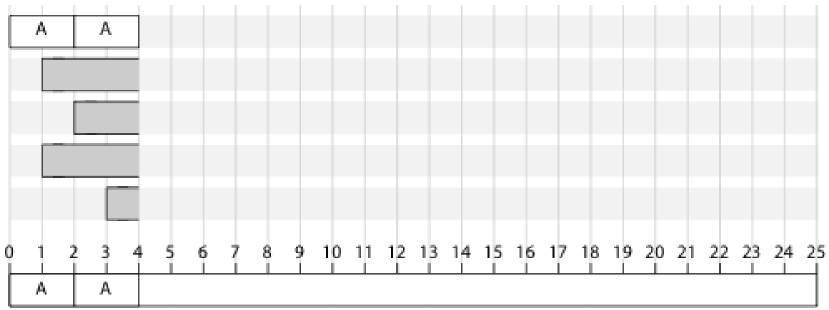
Priority	**A**	B	C	D	E
Base	32	2	2	0	8
Dynamic	-16	0	0	0	0
Compensation	0	1	1	1	0
Total	16	3	3	1	8

Time = 3



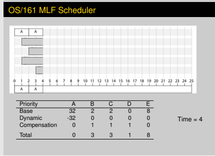
E arrives at time 3. A still has the highest priority. We don't compensate because a time slice didn't end, so we didn't reschedule.

OS/161 MLF Scheduler



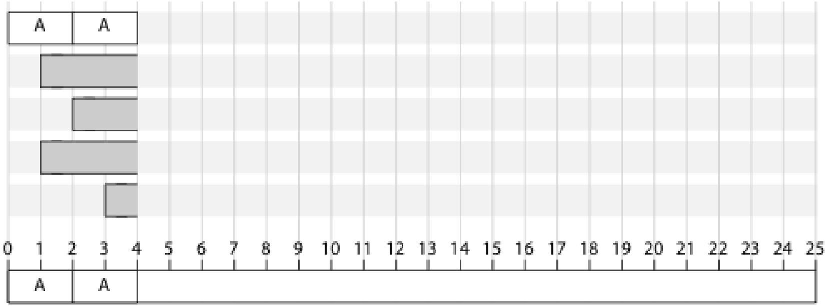
Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	0	0	0	0
Compensation	0	1	1	1	0
Total	0	3	3	1	8

Time = 4



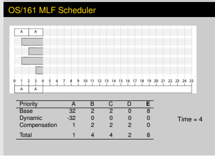
A's time slice ends, so we reduce its dynamic priority. Time to reschedule!

OS/161 MLF Scheduler



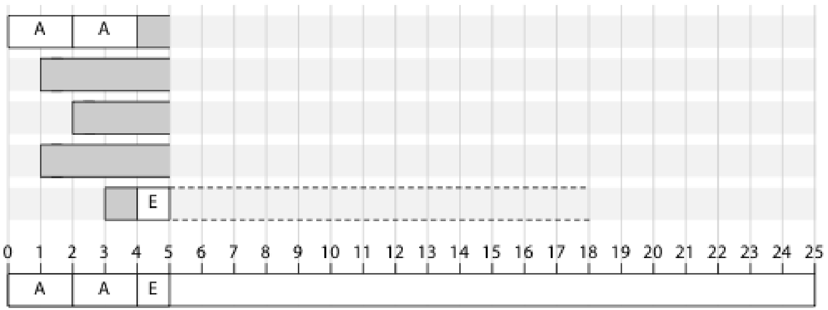
Priority	A	B	C	D	**E**
Base	32	2	2	0	8
Dynamic	-32	0	0	0	0
Compensation	1	2	2	2	0
Total	1	4	4	2	8

Time = 4



E now has the highest priority. Everybody else (including A) gets compensation.

OS/161 MLF Scheduler

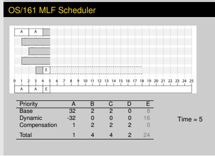


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	0	0	0	16
Compensation	1	2	2	2	0
Total	1	4	4	2	24

Time = 5

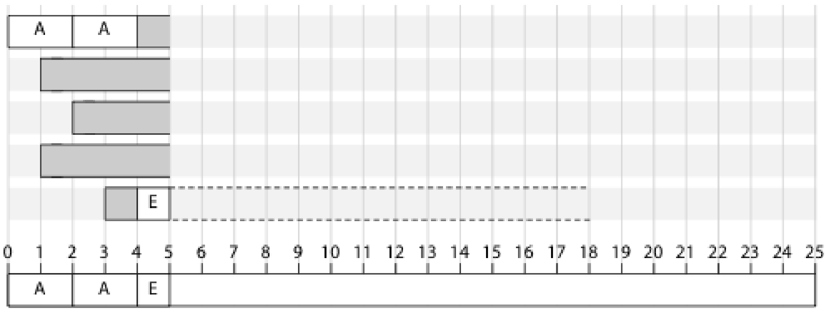
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



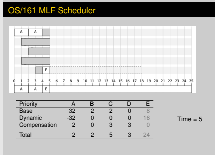
E runs for one time unit and then blocks for I/O. It gets a 16-point dynamic priority boost for doing I/O, but isn't eligible for scheduling because it's blocked.

OS/161 MLF Scheduler



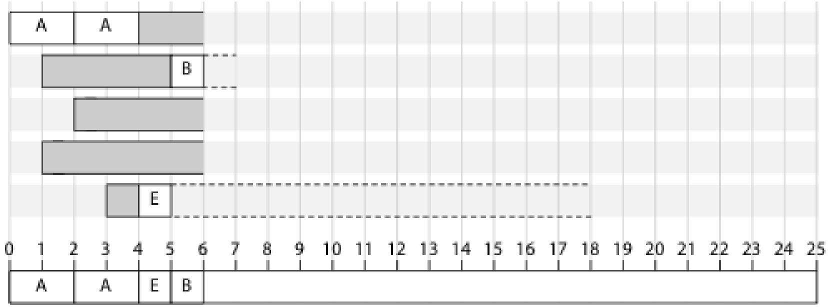
Priority	A	**B**	C	D	E
Base	32	2	2	0	8
Dynamic	-32	0	0	0	16
Compensation	2	0	3	3	0
Total	2	2	5	3	24

Time = 5



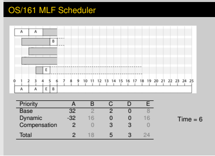
B and C are now tied for the highest total, so we arbitrarily choose B. B's compensation gets set to 0, and everybody else who is passed over has their compensation incremented. Note that E doesn't get compensation because it is blocked for I/O, so it wasn't passed over.

OS/161 MLF Scheduler



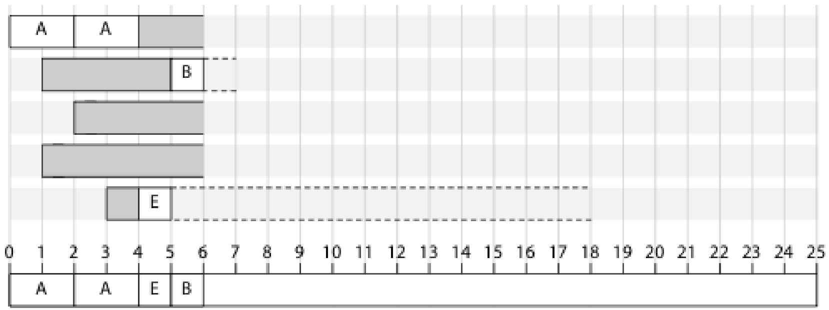
Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	16	0	0	16
Compensation	2	0	3	3	0
Total	2	18	5	3	24

Time = 6



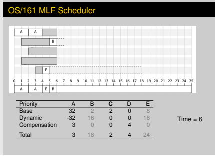
B runs for one unit and blocks for I/O. It gets a dynamic boost of 16. C now has the highest priority.

OS/161 MLF Scheduler



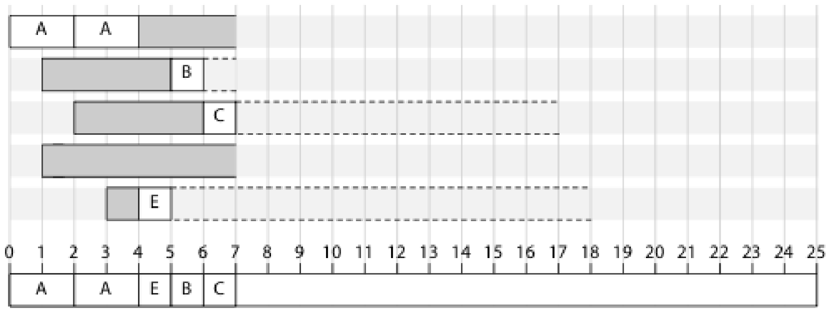
Priority	A	B	**C**	D	E
Base	32	2	2	0	8
Dynamic	-32	16	0	0	16
Compensation	3	0	0	4	0
Total	3	18	2	4	24

Time = 6



C is chosen to run (boldface) and has its compensation set to 0; everybody non-blocked gets a compensation bump.

OS/161 MLF Scheduler

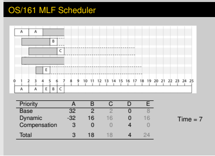


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	16	16	0	16
Compensation	3	0	0	4	0
Total	3	18	18	4	24

Time = 7

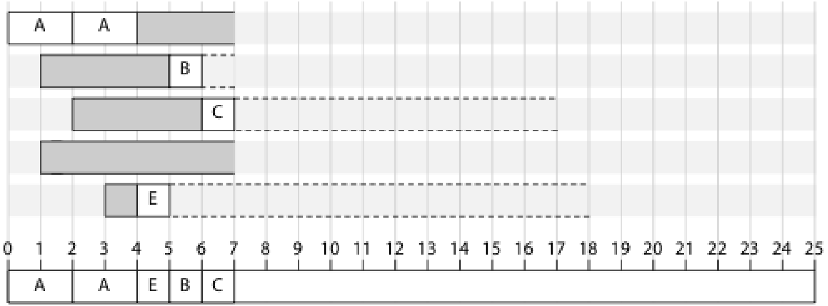
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



B’s I/O is finished, and C blocks for I/O. C gets a dynamic boost of 16. B’s numbers don’t change; they were handled when it blocked.

OS/161 MLF Scheduler

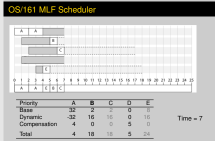


Priority	A	**B**	C	D	E
Base	32	2	2	0	8
Dynamic	-32	16	16	0	16
Compensation	4	0	0	5	0
Total	4	18	18	5	24

Time = 7

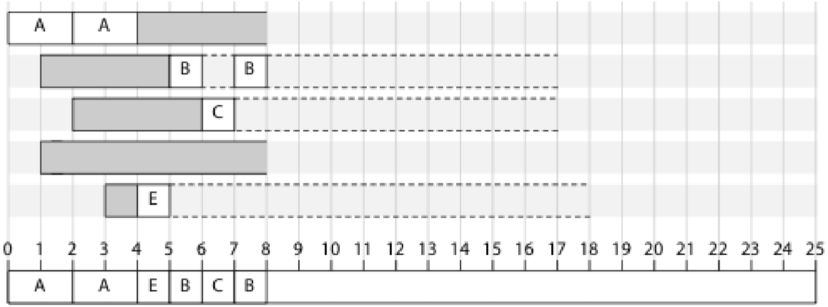
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



B runs again (as a reward for having done I/O). A and C get compensation.

OS/161 MLF Scheduler

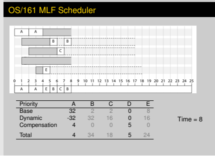


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	32	16	0	16
Compensation	4	0	0	5	0
Total	4	34	18	5	24

Time = 8

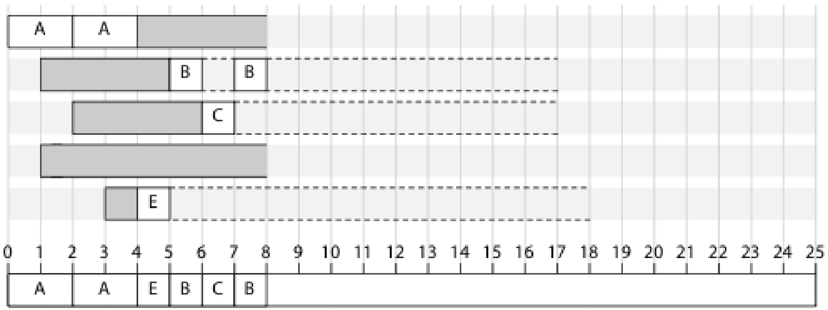
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



B again blocks, getting another dynamic boost. D will run next.

OS/161 MLF Scheduler

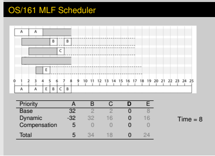


Priority	A	B	C	**D**	E
Base	32	2	2	0	8
Dynamic	-32	32	16	0	16
Compensation	5	0	0	0	0
Total	5	34	18	0	24

Time = 8

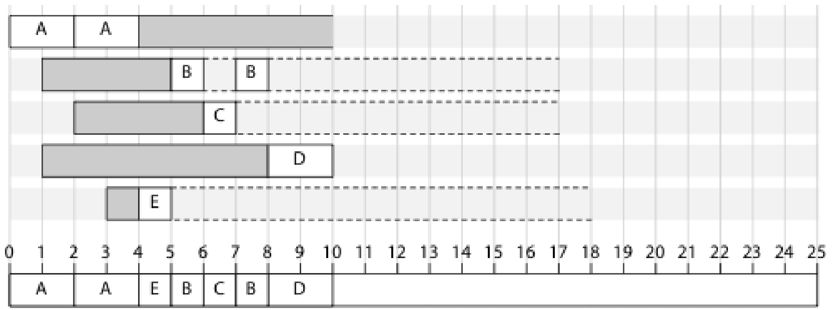
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



D runs, so its compensation gest set to 0. A gets more compensation.

OS/161 MLF Scheduler

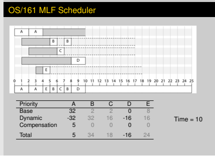


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	32	16	-16	16
Compensation	5	0	0	0	0
Total	5	34	18	-16	24

Time = 10

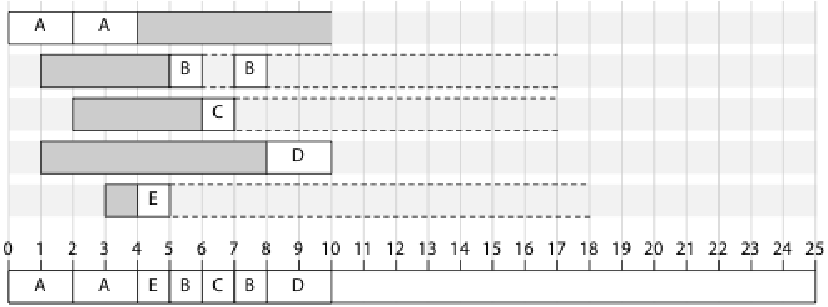
2013-05-17

CS34
└─Scheduling
└─OS/161 MLF Scheduler



We skip to time 10, when D's time slice runs out. It gets a dynamic penalty. A will run next.

OS/161 MLF Scheduler

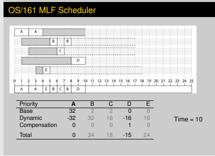


Priority	**A**	B	C	D	E
Base	32	2	2	0	8
Dynamic	-32	32	16	-16	16
Compensation	0	0	0	1	0
Total	0	34	18	-15	24

Time = 10

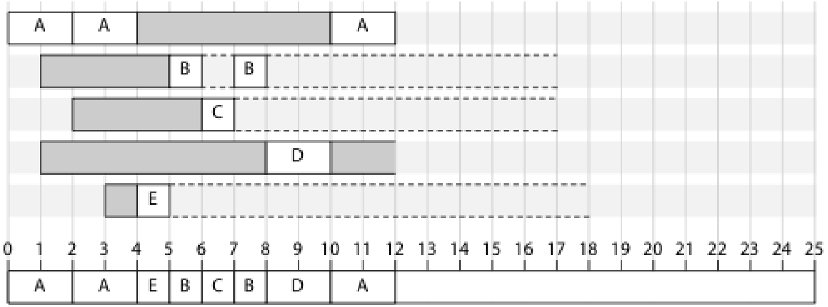
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



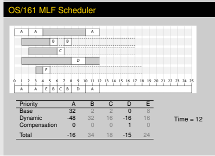
A is chosen to run; its compensation is reset. D gets compensation.

OS/161 MLF Scheduler



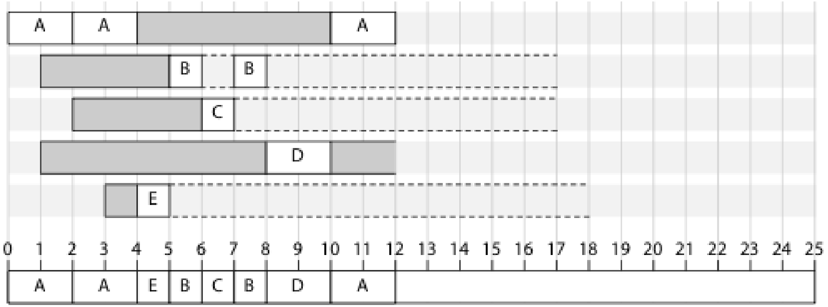
Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-48	32	16	-16	16
Compensation	0	0	0	1	0
Total	-16	34	18	-15	24

Time = 12



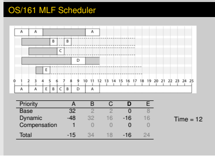
A expires its time slice. It becomes (barely) lower priority than D.

OS/161 MLF Scheduler



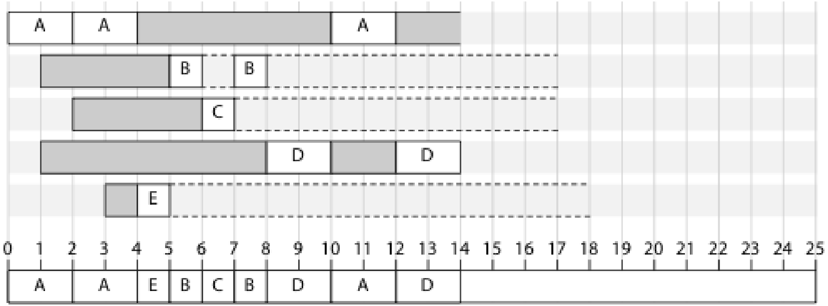
Priority	A	B	C	**D**	E
Base	32	2	2	0	8
Dynamic	-48	32	16	-16	16
Compensation	1	0	0	0	0
Total	-15	34	18	-16	24

Time = 12



D is chosen to run; A gets compensation.

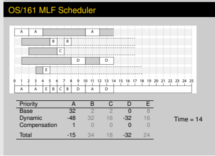
OS/161 MLF Scheduler



Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-48	32	16	-32	16
Compensation	1	0	0	0	0
Total	-15	34	18	-32	24

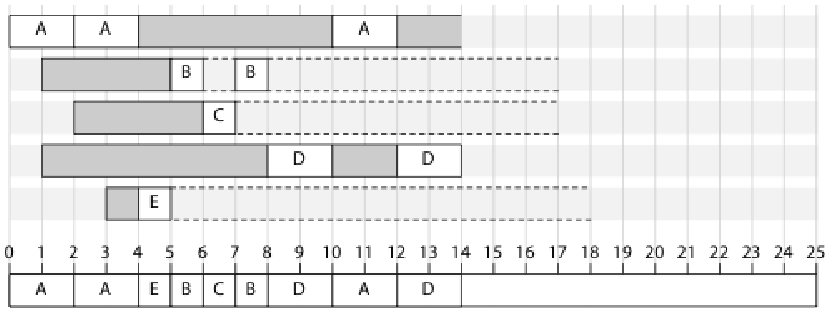
Time = 14

2013-05-17



D uses up its slice. A and D are now round-robining.

OS/161 MLF Scheduler

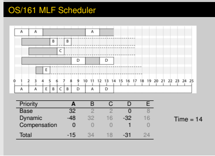


Priority	**A**	B	C	D	E
Base	32	2	2	0	8
Dynamic	-48	32	16	-32	16
Compensation	0	0	0	1	0
Total	-15	34	18	-31	24

Time = 14

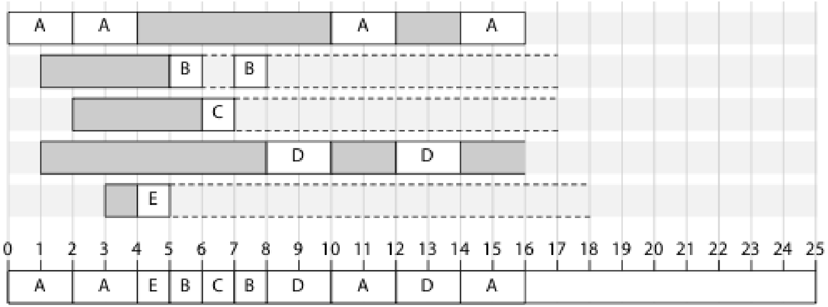
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



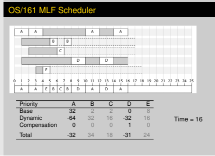
A runs.

OS/161 MLF Scheduler



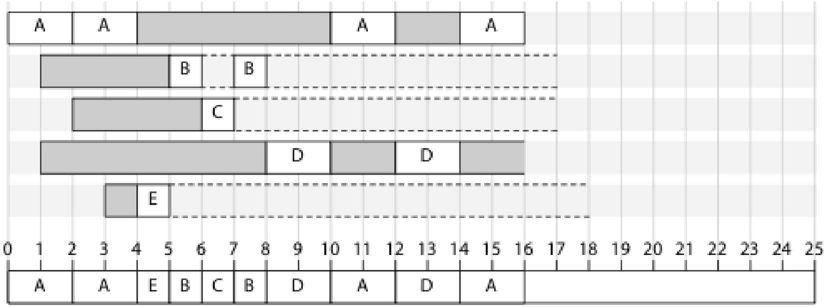
Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-64	32	16	-32	16
Compensation	0	0	0	1	0
Total	-32	34	18	-31	24

Time = 16



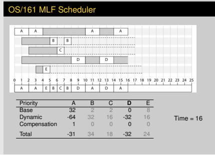
A expires.

OS/161 MLF Scheduler



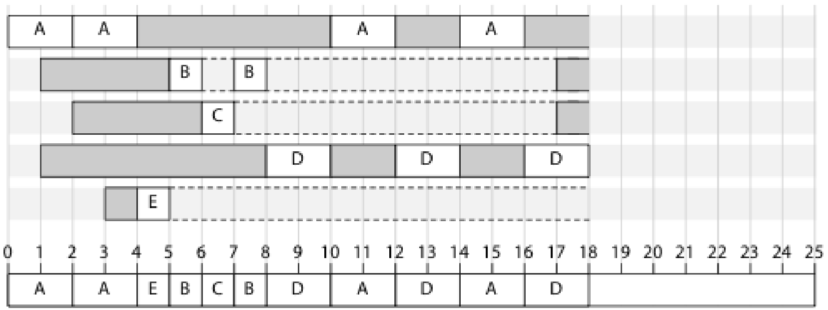
Priority	A	B	C	**D**	E
Base	32	2	2	0	8
Dynamic	-64	32	16	-32	16
Compensation	1	0	0	0	0
Total	-31	34	18	-32	24

Time = 16



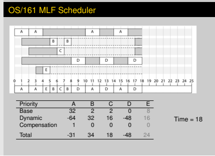
D runs.

OS/161 MLF Scheduler



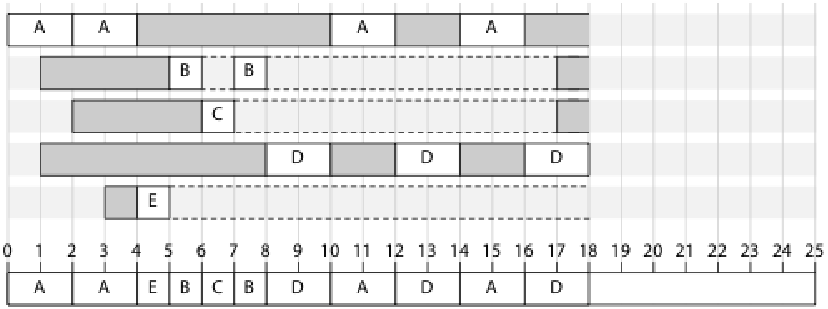
Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-64	32	16	-48	16
Compensation	1	0	0	0	0
Total	-31	34	18	-48	24

Time = 18



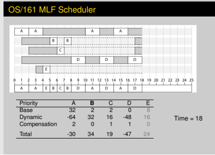
B and C finish I/O at time 17. Properly speaking, they should interrupt D at this point. But either the diagram is wrong, or the MLF scheduler refuses to preempt a running CPU-bound task. (If so, it's not doing well; this would be a good time to discuss deceptive idleness and anticipatory scheduling.)

OS/161 MLF Scheduler



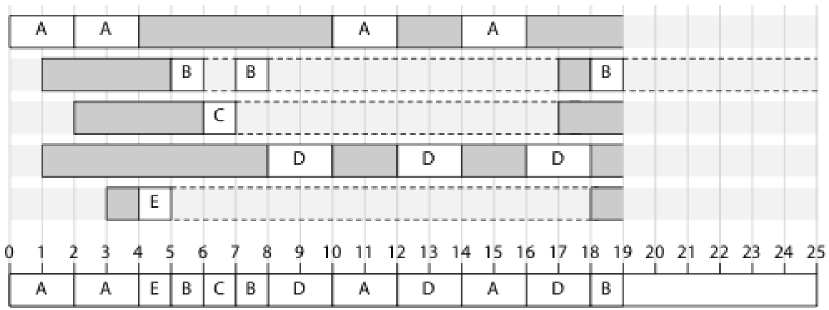
Priority	A	**B**	C	D	E
Base	32	2	2	0	8
Dynamic	-64	32	16	-48	16
Compensation	2	0	1	1	0
Total	-30	34	19	-47	24

Time = 18



B is now the highest priority. Compensations are adjusted, and it runs.

OS/161 MLF Scheduler

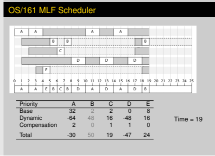


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-64	48	16	-48	16
Compensation	2	0	1	1	0
Total	-30	50	19	-47	24

Time = 19

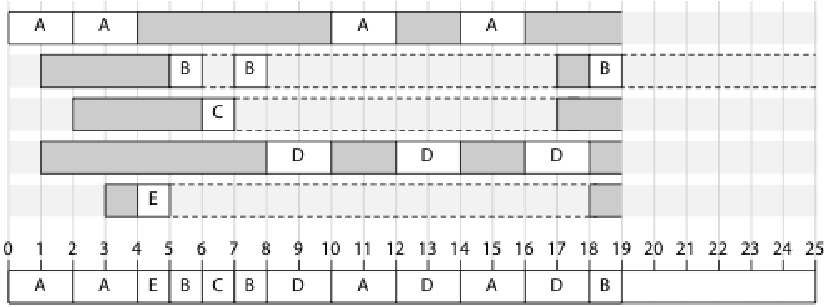
2013-05-17

CS34
└─Scheduling
└─OS/161 MLF Scheduler



B is heavily I/O-bound, so it blocks yet again, getting another dynamic boost.

OS/161 MLF Scheduler

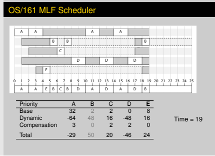


Priority	A	B	C	D	**E**
Base	32	2	2	0	8
Dynamic	-64	48	16	-48	16
Compensation	3	0	2	2	0
Total	-29	50	20	-46	24

Time = 19

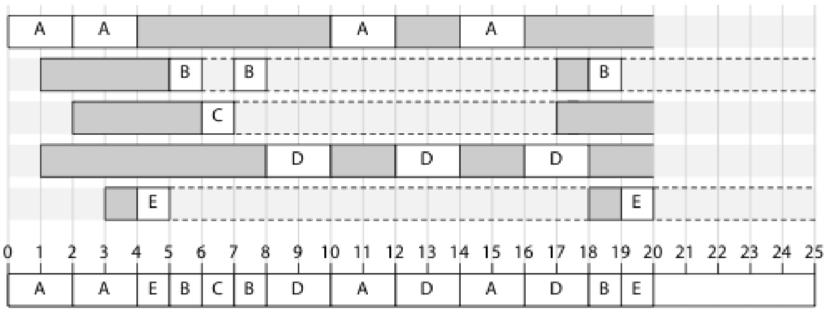
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



E has a higher base priority than C, so it runs. (Eventually, C's compensation would help it out.)

OS/161 MLF Scheduler

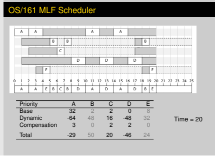


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-64	48	16	-48	32
Compensation	3	0	2	2	0
Total	-29	50	20	-46	24

Time = 20

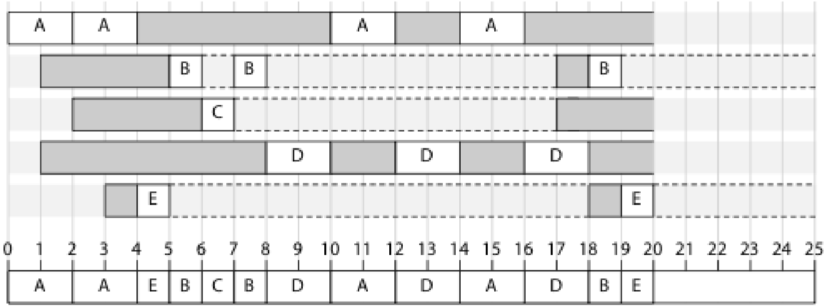
2013-05-17

CS34
└─Scheduling
└─OS/161 MLF Scheduler



E blocks for I/O. C will run now.

OS/161 MLF Scheduler

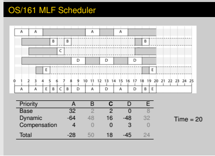


Priority	A	B	**C**	D	E
Base	32	2	2	0	8
Dynamic	-64	48	16	-48	32
Compensation	4	0	0	3	0
Total	-28	50	18	-45	24

Time = 20

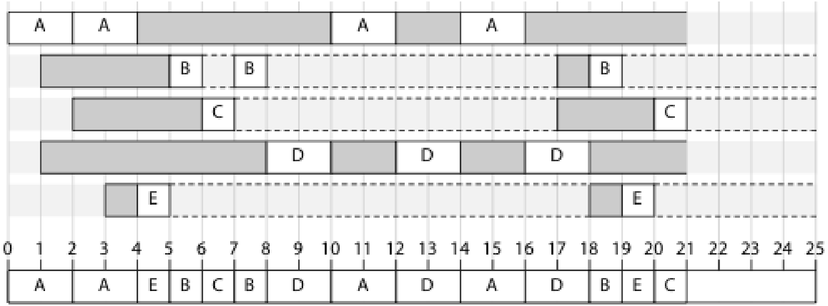
2013-05-17

CS34
└─Scheduling
└─OS/161 MLF Scheduler



C runs.

OS/161 MLF Scheduler

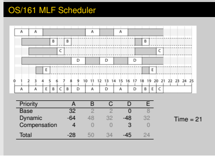


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-64	48	32	-48	32
Compensation	4	0	0	3	0
Total	-28	50	34	-45	24

Time = 21

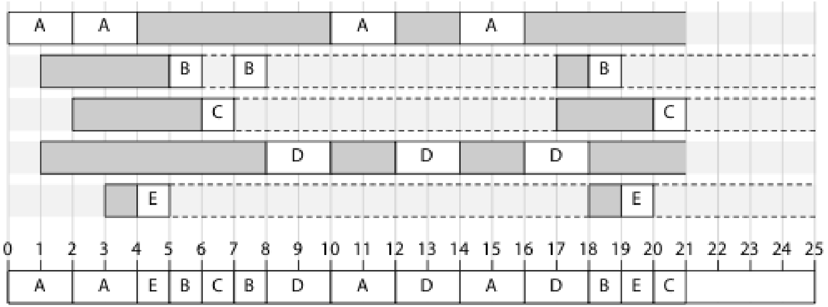
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



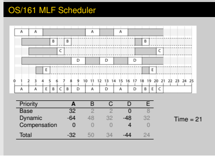
C blocks, getting another 16 dynamic. Now we only have CPU-bound processes left.

OS/161 MLF Scheduler



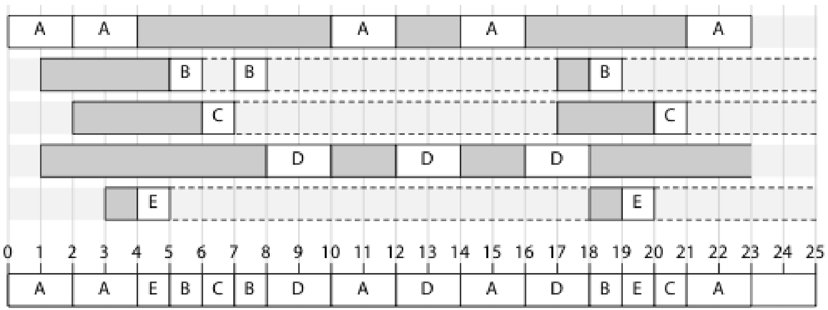
Priority	**A**	B	C	D	E
Base	32	2	2	0	8
Dynamic	-64	48	32	-48	32
Compensation	0	0	0	4	0
Total	-32	50	34	-44	24

Time = 21



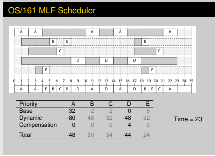
A runs.

OS/161 MLF Scheduler



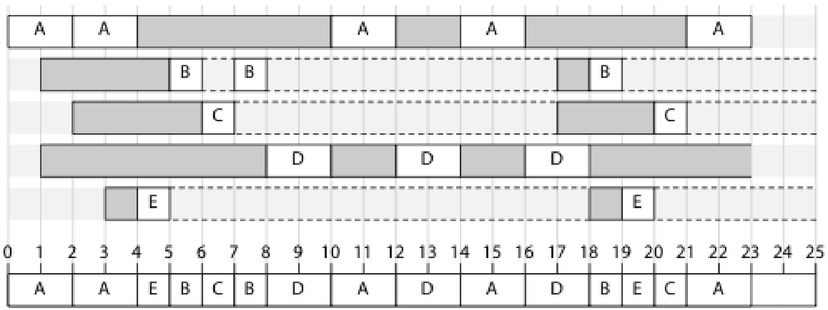
Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-80	48	32	-48	32
Compensation	0	0	0	4	0
Total	-48	50	34	-44	24

Time = 23



A is preempted.

OS/161 MLF Scheduler

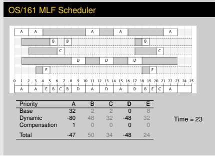


Priority	A	B	C	**D**	E
Base	32	2	2	0	8
Dynamic	-80	48	32	-48	32
Compensation	1	0	0	0	0
Total	-47	50	34	-48	24

Time = 23

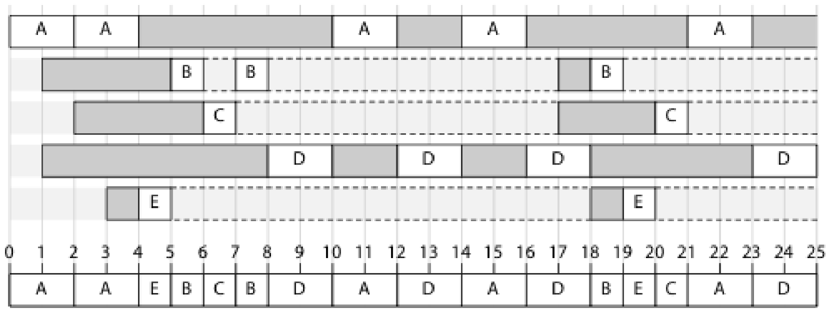
2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



D runs.

OS/161 MLF Scheduler

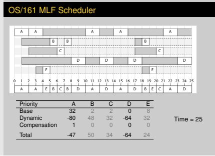


Priority	A	B	C	D	E
Base	32	2	2	0	8
Dynamic	-80	48	32	-64	32
Compensation	1	0	0	0	0
Total	-47	50	34	-64	24

Time = 25

2013-05-17

CS34
└Scheduling
└OS/161 MLF Scheduler



D is preempted. A will run again here.

Priority Inversion

What happens when a low-priority thread holds a lock that a high priority thread wants?

Priority Inversion

What happens when a low-priority thread holds a lock that a high priority thread wants?

Real-Time Scheduling

Consider two applications

- Video playback
- Controlling cancer treatment X-ray

How can we deal with their needs?

Real-Time Scheduling
Consider two applications
Video playback
Controlling cancer treatment X-ray
How can we deal with their needs?

Two approaches:

Hard Real-Time

You do some form of “admission control”:

- A process will say “this is what I need in the future”
- Depending on what it’s already committed to, the scheduler will say yes or no to that process

Soft Real-Time

- You usually give a deadline (“I want this done by this time”)
- If it doesn’t get done, it doesn’t get done (dropping frames during video playback)
  - You might not notice if only a few operations don’t make it

Class Exercise

Should we preempt kernel code, or wait until we hit user code?

Class Exercise
Should we preempt kernel code, or wait until we hit user code?

In hard real-time, definitely yes.
Traditionally, the kernel was never preemptible. A system call went until it decided to yield.
Advantages to non-preemptible kernel: easier to code; easier to make thread-safe; avoids many race conditions and bugs.
Advantages to preemptible kernel: you're worrying about mutithreaded cores anyway; can improve average latency; time spent in kernel could lead to scheduling unfairness; even with fair scheduler that tracks in-kernel time, you could get bad latency; it's possible to have kernel ignore timer interrupts (prevent preemption), while still having other interrupts (like disk access).
But. . . there are other solutions to latency. Instead of preemption, kernel could explicitly yield during hard/slow operations (original Unix kernel did that).
OS 161 preempts the kernel.
Once you have a multiprocessor, you already need locks and stuff in your kernel, so making it preemptible is not as big a deal.