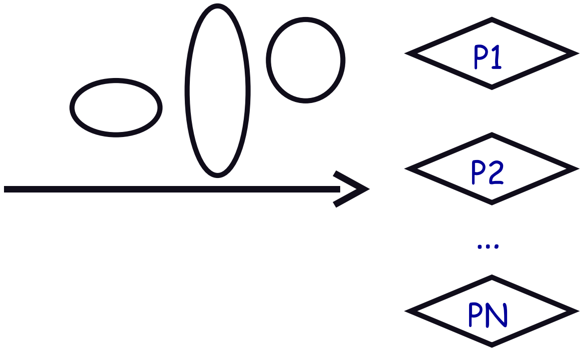
CS350: Operating Systems

Lecture 10: Scheduling

Ali Mashtizadeh

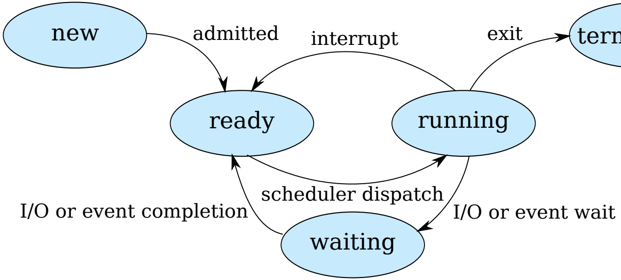
University of Waterloo

CPU Scheduling



- The scheduling problem:
  - Have K jobs ready to run
  - Have $N \geq 1$ CPUs
  - Which jobs to assign to which CPU(s)
- When do we make decision?

CPU Scheduling



- Scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from new/waiting to ready
  4. Exits
- Non-preemptive schedules use 1 & 4 only
- Preemptive schedulers run at all four points

Scheduling criteria

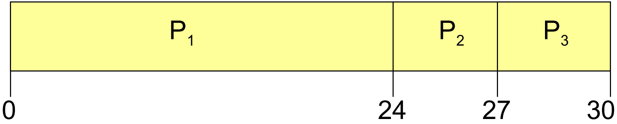
- Why do we care?
  - What goals should we have for a scheduling algorithm?

Scheduling criteria

- Why do we care?
  - What goals should we have for a scheduling algorithm?
- *Throughput* – # of procs that complete per unit time
  - Higher is better
- *Turnaround time* – time for each proc to complete
  - Lower is better
- *Response time* – time from request to first response (e.g., key press to character echo, not launch to exit)
  - Lower is better
- Above criteria are affected by secondary criteria
  - *CPU utilization* – fraction of time CPU doing productive work
  - *Waiting time* – time each proc waits in ready queue

Example: FCFS Scheduling

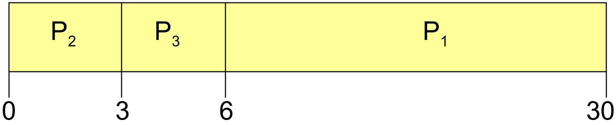
- Run jobs in order that they arrive
  - Called "*First-come first-served*" (FCFS)
  - E.g.., Say P_1 needs 24 sec, while P_2 and P_3 need 3.
  - Say P_2, P_3 arrived immediately after P_1, get:



- Dirt simple to implement—how good is it?
- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround Time: $\mathrm{P}_1 : 24$, $\mathrm{P}_2 : 27$, $\mathrm{P}_3 : 30$
  - Average TT: $(24 + 27 + 30)/3 = 27$
- Can we do better?

FCFS continued

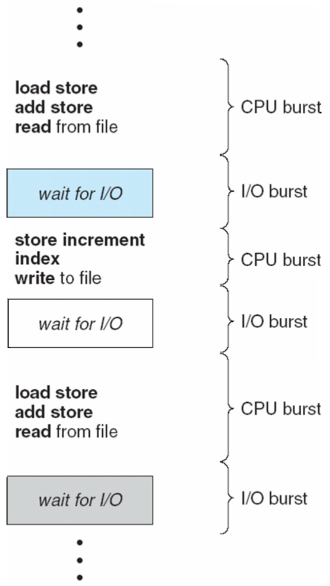
- Suppose we scheduled P_2, P_3, then P_1
  - Would get:



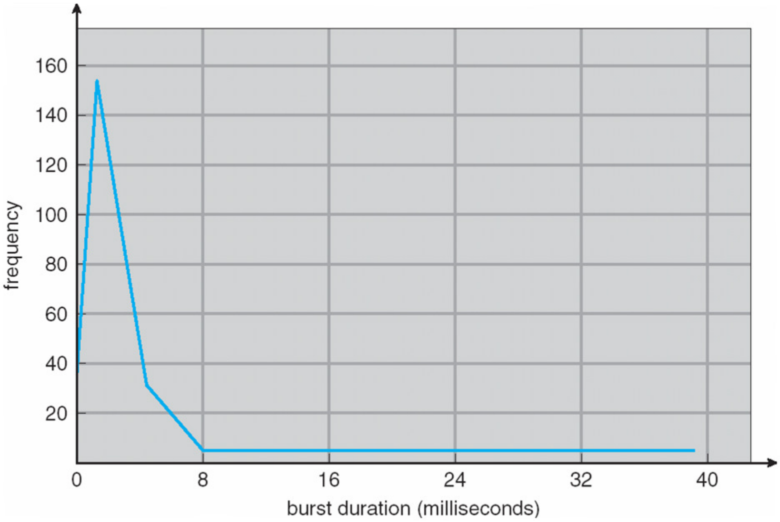
- Throughput: 3 jobs / 30 sec = 0.1 jobs/sec
- Turnaround time: $P_1 : 30$, $P_2 : 3$, $P_3 : 6$
  - Average TT: $(30 + 3 + 6)/3 = 13$ – much less than 27
- Lesson: scheduling algorithm can reduce TT
  - Minimizing waiting time can improve RT and TT
- What about throughput?

Bursts of computation & I/O

- Jobs contain I/O and computation
  - Bursts of computation
  - Then must wait for I/O
- To Maximize throughput
  - Must maximize CPU utilization
  - Also maximize I/O device utilization
- How to do?
  - Overlap I/O & computation from multiple jobs
  - Means *response time* very important for I/O-intensive jobs: I/O device will be idle until job gets small amount of CPU to issue next I/O request



Histogram of CPU-burst times



- **What does this mean for FCFS?**

FCFS Convoy effect

- CPU-bound jobs will hold CPU until exit or I/O (but I/O rare for CPU-bound thread)
  - long periods where no I/O requests issued, and CPU held
  - Result: poor I/O device utilization
- Example: one CPU-bound job, many I/O bound
  - CPU-bound job runs (I/O devices idle)
  - CPU-bound job blocks
  - I/O-bound job(s) run, quickly block on I/O
  - CPU-bound job runs again
  - I/O completes
  - CPU-bound job continues while I/O devices idle
- Simple hack: run process whose I/O completed?
  - What is a potential problem?

SJF Scheduling

- *Shortest-job first* (SJF) attempts to minimize TT
  - Schedule the job whose next CPU burst is the shortest
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)
- What does SJF optimize?

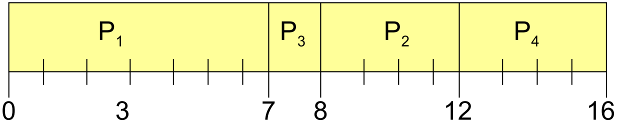
SJF Scheduling

- *Shortest-job first* (SJF) attempts to minimize TT
  - Schedule the job whose next CPU burst is the shortest
- Two schemes:
  - *Non-preemptive* – once CPU given to the process it cannot be preempted until completes its CPU burst
  - *Preemptive* – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt (Known as the *Shortest-Remaining-Time-First* or SRTF)
- What does SJF optimize?
  - Gives minimum average *waiting time* for a given set of processes

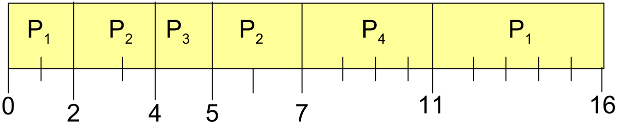
Examples

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- Non-preemptive



- Preemptive



- Drawbacks?

SJF limitations

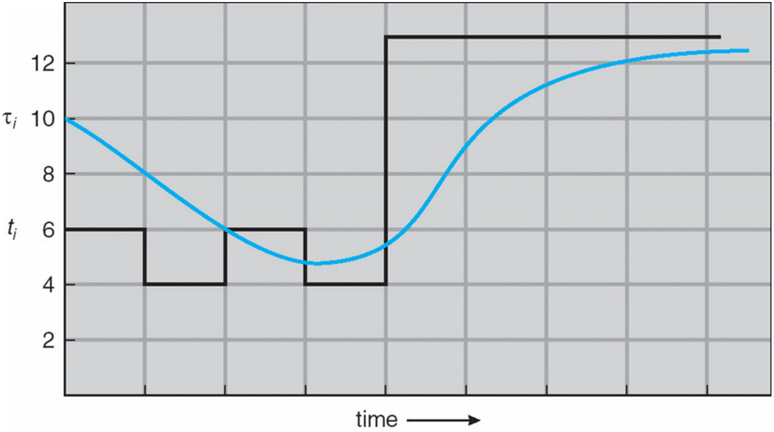
- Doesn't always minimize average turnaround time
  - Only minimizes waiting time, which minimizes response time
  - Example where turnaround time might be suboptimal?

- Can lead to unfairness or starvation
- In practice, can't actually predict the future
- But can estimate CPU burst length based on past
  - Exponentially weighted average a good idea
  - t_n actual length of proc's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $n+1^{st}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

SJF limitations

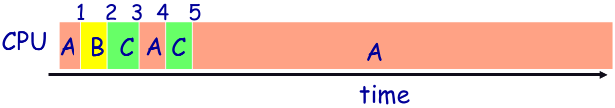
- Doesn't always minimize average turnaround time
  - Only minimizes waiting time, which minimizes response time
  - Example where turnaround time might be suboptimal?
  - Overall longer job has shorter bursts
- Can lead to unfairness or starvation
- In practice, can't actually predict the future
- But can estimate CPU burst length based on past
  - Exponentially weighted average a good idea
  - t_n actual length of proc's n^{th} CPU burst
  - τ_{n+1} estimated length of proc's $n + 1^{st}$
  - Choose parameter α where $0 < \alpha \leq 1$
  - Let $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

Exp. weighted average example



CPU burst (t_i)		6	4	6	4	13	13	13	…
"guess" (τ_i)	10	8	6	6	5	9	11	12	…

Round robin (RR) scheduling



- Solution to fairness and starvation
  - Preempt job after some time slice or *quantum*
  - When preempted, move to back of FIFO queue
  - (Most systems do some flavor of this)
- Advantages:
  - Fair allocation of CPU across jobs
  - Low average waiting time when job lengths vary
  - Good for responsiveness if small number of jobs
- Disadvantages?

RR disadvantages

- Varying sized jobs are good ...what about same-sized jobs?
- Assume 2 jobs of time=100 each:

- Even if context switches were free...
  - What would average completion time be with RR?
  - How does that compare to FCFS?

RR disadvantages

- Varying sized jobs are good ...what about same-sized jobs?
- Assume 2 jobs of time=100 each:

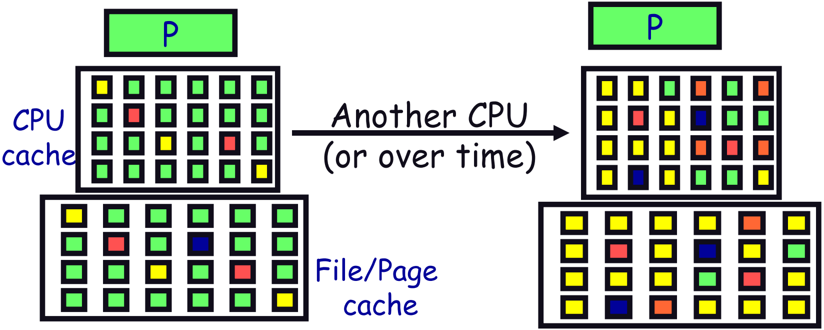
- Even if context switches were free...
  - What would average completion time be with RR? *199.5*
  - How does that compare to FCFS? *150*

Context switch costs

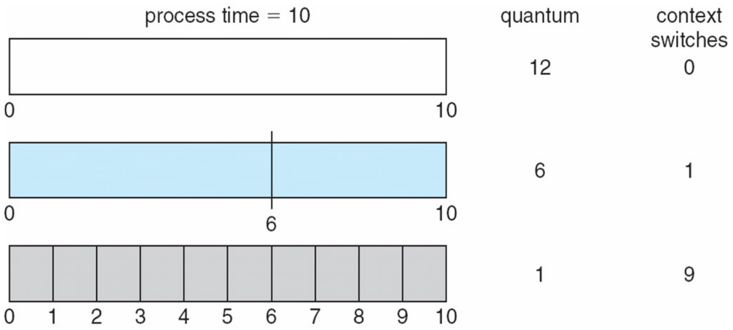
- **What is the cost of a context switch?**

Context switch costs

- What is the cost of a context switch?
- Brute CPU time cost in kernel
  - Save and restore resisters, etc.
  - Switch address spaces (expensive instructions)
- Indirect costs: cache, buffer cache, & TLB misses

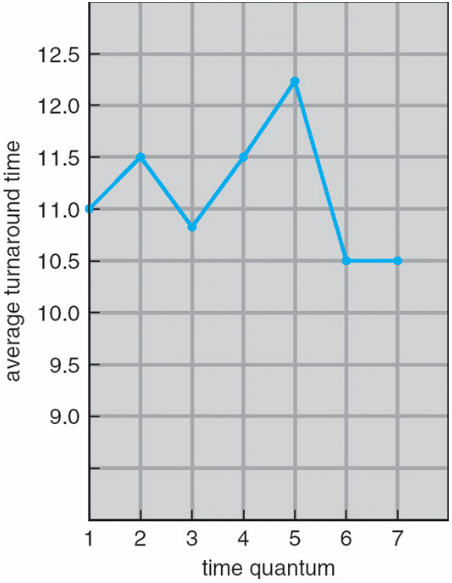


Time quantum



- How to pick quantum?
  - Want much larger than context switch cost
  - Majority of bursts should be less than quantum
  - But not so large system reverts to FCFS
- Typical values: 10–100 msec

Turnaround time vs. quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

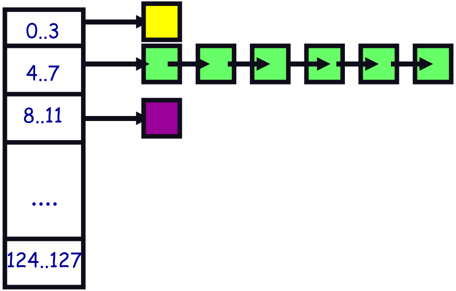
Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is a priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?

Priority scheduling

- Associate a numeric priority with each process
  - E.g., smaller number means higher priority (Unix/BSD)
- Give CPU to the process with highest priority
  - Can be done preemptively or non-preemptively
- Note SJF is a priority scheduling where priority is the predicted next CPU burst time
- Starvation – low priority processes may never execute
- Solution?
  - Aging: increase a process's priority as it waits

Multilevel feeedback queues (BSD)



- Every runnable process on one of 32 run queues
  - Kernel runs process on highest-priority non-empty queue
  - Round-robins among processes on same queue
- Process priorities dynamically computed
  - Processes moved between queues to reflect priority changes
  - If a process gets higher priority than running process, run it
- Idea: Favor interactive jobs that use less CPU

Process priority

- p_nice – user-settable weighting factor
- p_estcpu – per-process estimated CPU usage
  - Incremented whenever timer interrupt found proc. running
  - Decayed every second while process runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \mathsf{load}}{2 \cdot \mathsf{load} + 1}\right) \texttt{p\_estcpu} + \texttt{p\_nice}$$

  - Load is sampled average of length of run queue plus short-term sleep queue over last minute
- Run queue determined by $\texttt{p\_usrpri}/4$

$$\texttt{p\_usrpri} \leftarrow 50 + \left(\frac{\texttt{p\_estcpu}}{4}\right) + 2 \cdot \texttt{p\_nice}$$

(value clipped if over 127)

Sleeping process increases priority

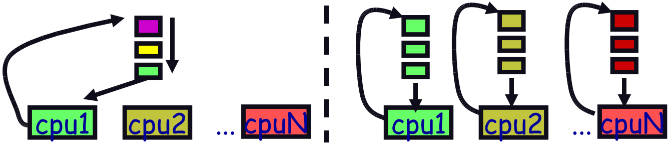
- `p_estcpu` not updated while asleep
  - Instead `p_slptime` keeps count of sleep time
- When process becomes runnable

$$\texttt{p\_estcpu} \leftarrow \left(\frac{2 \cdot \text{load}}{2 \cdot \text{load} + 1}\right)^{\texttt{p\_slptime}} \times \texttt{p\_estcpu}$$

  - Approximates decay ignoring nice and past loads
- Previous description based on *The Design and Implementation of the 4.4BSD Operating System* by McKusick

Multiprocessor scheduling issues

- Must decide on more than which processes to run
  - Must decide on which CPU to run which process
- Moving between CPUs has costs
  - More cache misses, depending on arch more TLB misses too
- *Affinity scheduling*—try to keep threads on same CPU



  - But also prevent load imbalances
  - Do *cost-benefit* analysis when deciding to migrate

Thread dependencies

- Say H at high priority, L at low priority
  - L acquires lock l.
  - Scenario 1: H tries to acquire l, fails, spins. L never gets to run.
  - Scenario 2: H tries to acquire l, fails, blocks. M enters system at medium priority. L never gets to run.
  - Both scenes are examples of *priority inversion*
- Scheduling = deciding who should make progress
  - A thread's importance should increase with the importance of those that depend on it
  - Naïve priority schemes violate this

Priority donation

- **Example 1:** L **low,** M **medium,** H **high priority**
  - L **holds lock** l
  - M **waits on** l**,** L**'s priority raised to** $\mathrm{L}_1 = \max(\mathrm{M}, \mathrm{L}) = 4$
  - **Then** H **waits on** l**,** L**'s priority raised to** $\max(\mathrm{H}, \mathrm{L}_1) = 8$
- **Example 2: Same** $\mathrm{L}, \mathrm{M}, \mathrm{H}$ **as above**
  - L **holds lock** l**,** M **holds lock** l_2
  - M **waits on** l**,** L**'s priority now** $\mathrm{L}_1 = 4$ **(as before)**
  - **Then** H **waits on** l_2**.** M**'s priority goes to** $\mathrm{M}_1 = \max(\mathrm{H}, \mathrm{M}) = 8$**, *and*** L**'s priority raised to** $\max(\mathrm{M}_1, \mathrm{L}_1) = 8$
- **Example 3:** L **(prio 2),** $\mathrm{M}_1, \ldots \mathrm{M}_{1000}$ **(all prio 4)**
  - L **has** l**, and** $\mathrm{M}_1, \ldots, \mathrm{M}_{1000}$ **all block on** l**.** L**'s priority is** $\max(\mathrm{L}, \mathrm{M}_1, \ldots, \mathrm{M}_{1000}) = 4$**.**

Borrowed Virtual Time Scheduler [Duda]

- Many modern schedulers employ notion of *virtual time*
  - Idea: Equalize virtual CPU time consumed by different processes
  - Examples: Linux CFS
- Idea: Run process w. lowest *effective virtual time*
  - A_i – *actual virtual time* consumed by process i
  - *effective virtual time* $E_i = A_i - (\text{warp}_i \; ? \; W_i : 0)$
- Supports real-time applications:
  - Warp factor allows borrowing against future CPU time
  - Allows an application to temporarily violate fairness

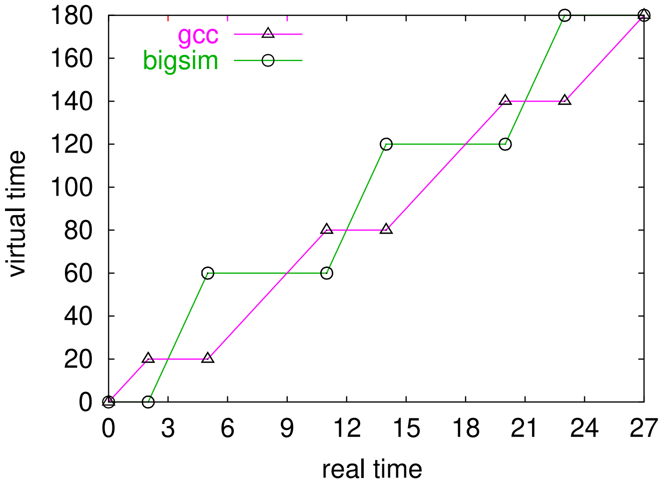
Process weights

- Each process i's faction of CPU determined by weight w_i
  - i should get $w_i / \sum_j w_j$ faction of CPU
  - So w_i is seconds per virtual time tick while i has CPU
- When i consumes t CPU time, track it: A_i += t/w_i
- Example: gcc (weight 2), bigsim (weight 1)
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, ...
  - Lots of context switches, not so good for performance
- Add in context switch allowance, C
  - Only switch from i to j if $E_j \leq E_i - C/w_i$
  - C is wall-clock time ($\gg$ context switch cost), so must divide by w_i
  - Ignore C if j just became runable...why?

Process weights

- Each process i's faction of CPU determined by weight w_i
  - i should get $w_i / \sum_j w_j$ faction of CPU
  - So w_i is seconds per virtual time tick while i has CPU
- When i consumes t CPU time, track it: A_i += t/w_i
- Example: gcc (weight 2), bigsim (weight 1)
  - Assuming no IO, runs: gcc, gcc, bigsim, gcc, gcc, bigsim, ...
  - Lots of context switches, not so good for performance
- Add in context switch allowance, C
  - Only switch from i to j if $E_j \leq E_i - C/w_i$
  - C is wall-clock time ($\gg$ context switch cost), so must divide by w_i
  - Ignore C if j just became runable to avoid affecting response time

BVT example

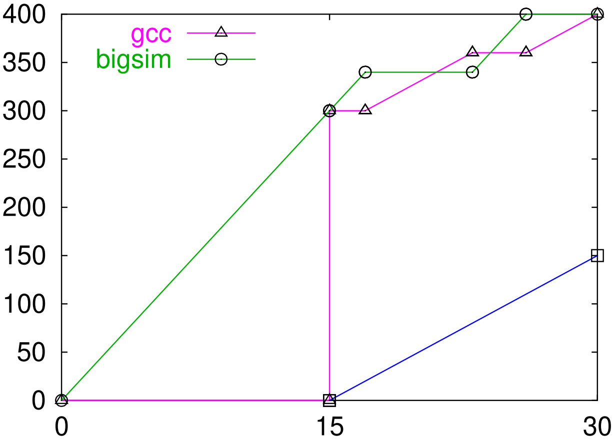


- gcc has weight 2, bigsim weight 1, $\mathrm{C} = 2$, no I/O
  - bigsim consumes virtual time at twice the rate of gcc

Sleep/wakeup

- Must lower priority (increase A_i) after wakeup
  - Otherwise process with very low A_i would starve everyone
- Bound lag with Scheduler Virtual Time (SVT)
  - SVT is minimum A_j for all runnable threads j
  - When waking i from voluntary sleep, set $A_i \leftarrow \max(A_i, SVT)$
- Note voluntary/involuntary sleep distinction
  - E.g., Don't reset A_j to SVT after page fault
  - Faulting thread needs a chance to catch up
  - But do set $A_i \leftarrow \max(A_i, SVT)$ after socket read
- Note: Even with SVT A_i can never decrease
  - After short sleep, might have $A_i > SVT$, so $\max(A_i, SVT) = A_i$
  - i never gets more than its fair share of CPU in long run

gcc wakes up after I/O

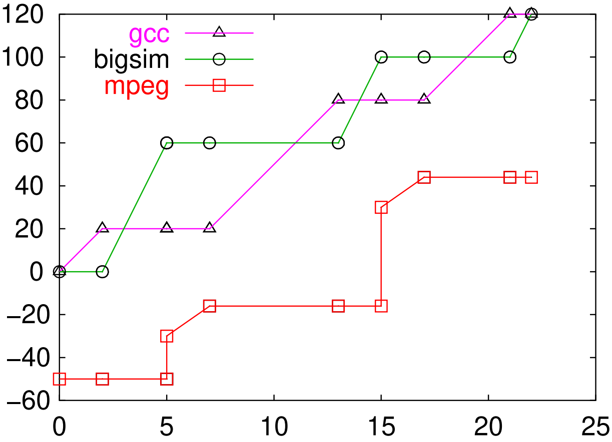


- **gcc's $\mathrm{A_i}$ gets reset to SVT on wakeup**
  - **Otherwise, would be at lower (blue) line and starve bigsim**

Real-time threads

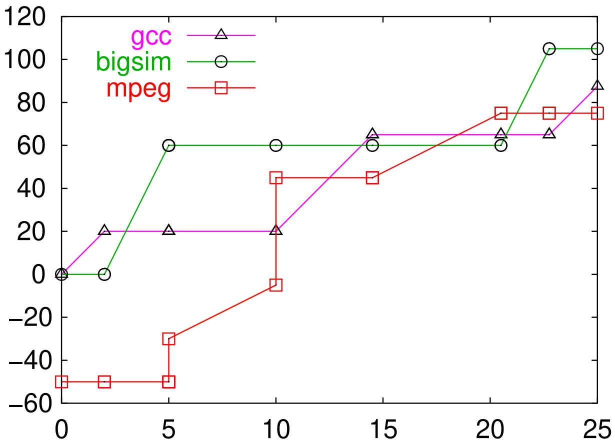
- Also want to support soft real-time threads
  - E.g., mpeg player must run every 10 clock ticks
- Recall $E_i = A_i - (\text{warp}_i \; ? \; W_i : 0)$
  - W_i is *warp factor* – gives thread precedence
  - Just give mpeg player i large W_i factor
  - Will get CPU whenever it is runable
  - But long term CPU share won't exceed $w_i / \sum_j w_j$
- Note W_i only matters when warp_i is **true**
  - Can set warp_i with a syscall, or have it set in signal handler
  - Also gets cleared if i keeps using CPU for L_i time
  - L_i limit gets reset every U_i time
  - $L_i = 0$ means no limit – okay for small W_i value

Running warped



- mpeg player runs with -50 warp value
  - Always gets CPU when needed, never misses a frame

Warped thread hogging CPU



- mpeg goes into tight loop at time 5
- Exceeds $\mathrm{L_i}$ at time 10, so $\mathrm{warp_i} \leftarrow$ false

Lottery Scheduler [Waldspurger]

- Reading assignment a great paper and simple algorithm
- Randomly select a process to run!
- Process priorities are determined by a number of tickets (or shares)