Instructor Notes

Waldspurger, C.A. and Weihl, W.E.
Lottery Scheduling: Flexible Proportional-Share Resource Management
Proceedings of the First OSDI, Monterey CA, November 1994, pp. 1-11.

1. What is the motivation for needing a new scheduler? What are the goals of a lottery scheduler?

- Existing priority-based schedulers give poor control over relative computation rates
  - instead optimize performance of interactive vs. cpu-bound jobs (approx SJF w/ multi-level feedback Q)
  - difficult to adjust priorities to get fairness or fixed rates

Goals?
- give proportional-share resource management
- modularity (hierarchical)

2. What are the advantages of using lottery tickets to represent resource rights?

- abstract (no machine details)
- relative (fraction of resources
  → more if lightly loaded)
- uniform (can be used for diff
  resources - cpu, mem, disk)

3. How is a scheduling decision made? What is the expectation for which process will be scheduled? Does lottery scheduling need to do anything special to guard against starvation?

- Hold a lottery - winner gets scheduled

- probabilistically fair

p, prob of winning : $\frac{t}{T}$

Starvation?

- MLFQ - mechanism to prevent starvation (cludgy)

- if have ticket, eventually win, so nothing extra needed

4. How does one **implement** a lottery? (See Figure 1.) What are ways to optimize the search for the winner?

- Fast random number generator

  0 ... n-1 (n is # active tickets

  → jobs on ready queue)

- List clients, traverse, calc. ticket sum

  O(c) → # active clients

Opt? Sort by tickets

Put in tree O(lg c)

5. **Ticket currencies** enable mutually trusting clients to redistribute tickets in a modular fashion. In Figure 3, how many base tickets does thread2 have? Thread3? Thread4? If thread1 became active, how many base tickets would each thread have?

$$t2: \frac{200}{500} \cdot \frac{200}{200} \cdot 1000 = 400$$

$$t3: \frac{300}{500} \cdot \frac{200}{200} \cdot 1000 = 600$$

$$t4: \frac{100}{100} \cdot \frac{100}{100} \cdot 2000 = 2000$$

T1 active → 300 alice issued tix

no change to t4 base or relative

$$t1: \frac{100}{100} \cdot \frac{100}{300} \cdot 1000 = 333$$

$$t2: \frac{200}{500} \cdot \frac{200}{300} \cdot 1000 = 2667$$

$$t3: \frac{300}{500} \cdot \frac{200}{300} \cdot 1000 = 400$$

1000 tickets

6. **Ticket inflation** involves a client escalating its resource rights by creating more lottery tickets. When is this useful?

-Only within trusting/cooperative clients (e.g. within a currency)

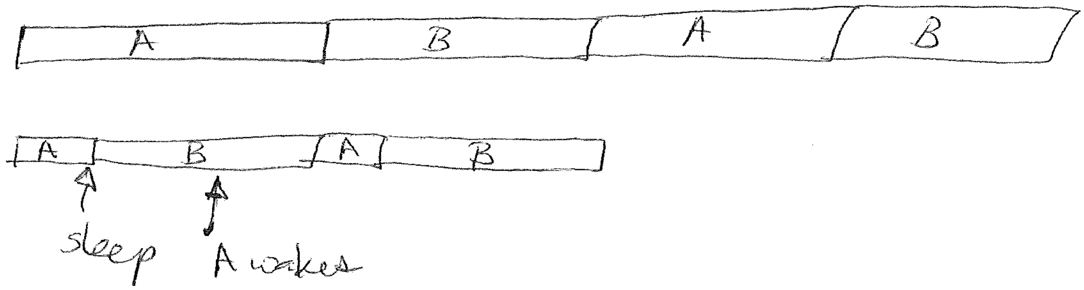
7. **Ticket transfers** are the explicit transfers of tickets from one client to another. Can you think of two examples where ticket transfers could be useful?

- Another process doing work on your behalf (server - RPC)

- Waiting for another process (holding a lock)

8. **Compensation tickets** are used to temporarily inflate tickets by 1/f when a process only uses a fraction f of its quantum. When does this not work as desired?

each has 50 tix



if A uses $\frac{1}{10}$ of quantum, give it $10 \cdot 50$ tix for next lottery

- more likely to win next lottery, but doesn't preempt B...

- Problem: Can't win lotteries not active for!

9. What do Figures 4 and 5 show? Is randomness a good quality for a scheduler?

- Probabilistically fair w/ variance

- Make deterministic scheduler (stride)

- No variance, but not conceptually intriguing!

10. What do Figures 6 and 7 and 9 show?

-Works well, can dynamically adjust tickets

7: Give tix to server on ~~you~~ client behalf

-9: Currencies insulate loads

11. What problem does Figure 8 reveal?

3 : 2 : 1 → 1.9 : 1.5 : 1

Other components don't use proportional share (RR in display)

12. Conclusions?

- Revived interest in scheduling
- Great match for hierarchies; extensible systems
- Good match for proportional shares in shared services
- Simple, cute w/ randomness
  (but why not deterministic version?)

- Doesn't handle I/O well
  (lots harder because of state - disk head)

Banga, G., Druschel, P,. Mogul, J.
Resource Containers: A New Facility for Resource Management in Server Systems
Proceedings of the OSDI-III), New Orleans, LA, February, 1999, 45-58.

1. What is the motivation for resource containers? What are the problems with existing approaches? What is the idea behind a resource container?

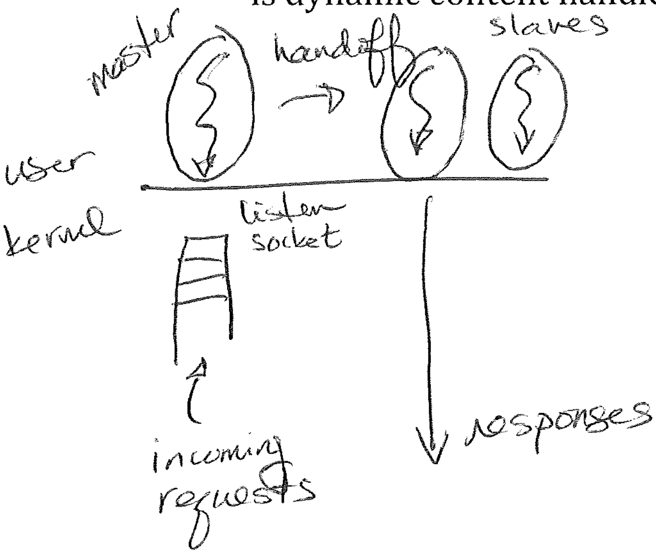
• Resource management in servers important
  + want to exert explicit control over consumption policies (QoS)
  + handle denial-of-service attacks

Problem?
- Protection domain + resource principal (logical task)
  coincide in "process"

- time spent in kernel not charged to processes correctly

New abstraction:
R.C. - associate all resource activity related to particular task

2. Background. How is a process (or thread)-per-connection HTTP server structured? How is an event-driven server structured? How is dynamic content handled?



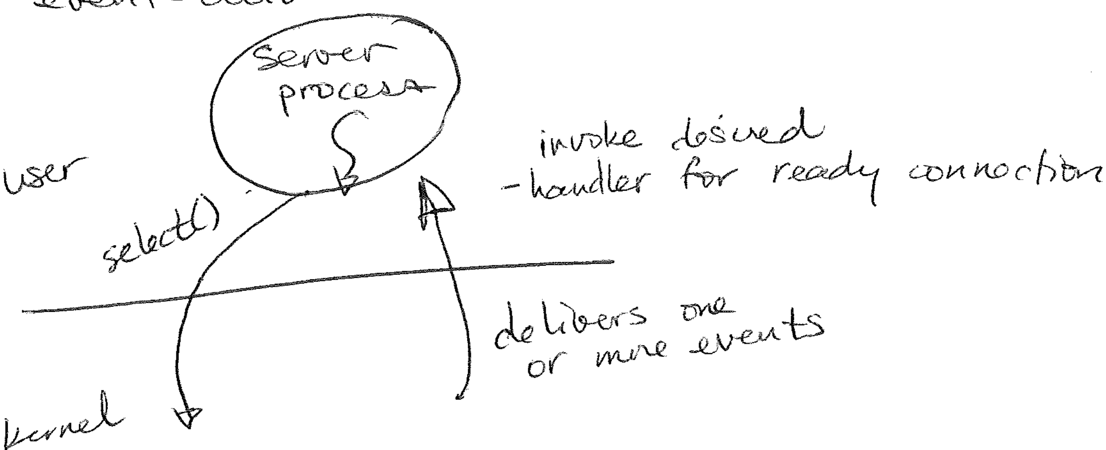
① fork new slave for each new connection

② pool of pre-forked processes (more efficient)

- high c.s. costs
- IPC costs

---

event-driven



- app decides which handler to run (most control)

Dynamic:

- cgi program - fork into separate process for protection (fault isolation)

3. What are the problems (again) with these approaches?

- Different threads handle different work over time (diff. clients, diff pri)

- No kernel accounting - network handling isn't charged to later responsible thread

- Can't associate cgi w/ activity

4. What is a resource container? Is the idea to associate resource usage with a thread or a container?

- Abstract entity that logically contains all resources used to achieve task
  - scheduling parameters, resource limits
  - usage $\longleftrightarrow$ R.C.

Thread binds itself to different R.C over time depending on task it is working on

5. With resource containers, the **binding** between threads and resource containers is dynamic. Can the binding for a thread change over time to different resource containers? Can multiple threads be bound to a single container at the same time?

Yes

Yes

6. Problem: What if a thread switches rapidly between between containers (should it be descheduled when it is associated with a lower priority container???) What is their compromise? *

No!
- too costly to recompute thread priority
+ context switch

Compromise: Scheduler binding
→ Combine all r.c. over which thread is currently multiplexed

→ Performed automatically by system
(look @ active set, remove old ones)

→ Average priority/usage over all

Implication:
- Not 100% accurate
- Low + hi priority tasks shouldn't be handled by same thread
- Does not apply to event-based systems

7. When applying resource containers to a web server... How should resource containers be used with multiple threads? With dynamic content? With events? How does network activity interact with resource containers?

- assign thread to desired R.C. for each new connection depending on policy

- Dynamic:
  assign cgi process to R.C.

- EVENTS:
  - no scheduling by OS
  - Res. Containers used for accounting + Information
  - App uses to determine order to handle events

- Networking
  - Specify ASAP socket belongs to desired R.C.

8. Why do all of their experiments use requests to the same 1KB file?

- Hit in buffer cache
- Don't have I/O incorporated

9. In Figure 11, why can't the system without resource containers give preference to a high-priority client?

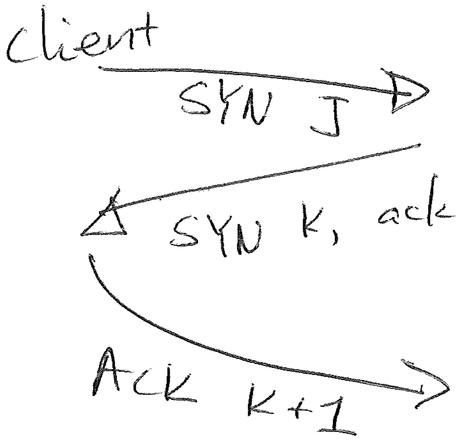
– 1 hi pri, increase # lo pri

w/o R.C.

– @ app level give pri to #1, but lots of work in kernel

10. What is a SYN attack? How can resource containers be used to protect against one?

setup TCP connection w/ 3-way handshake



create entry

must keep state
for time-out (75 sec)

Unmodified: queue fills, can't respond to
any clients

R.C.

- Notify app when SYN dropped +
- Isolate bad client to low priority (0)
  listen() socket

- Still some processing needed, but little

11. Conclusions?

- Good idea to separate accounting from protection
- Not incorporated w/ other resources yet
  - CPU always easiest!