Review for Midterm

Starring Ari and Tyler

Basic OS structure

- OS has two chief goals: arbitrating access to resources, and exposing functionality.
- Often go together: we arbitrate hardware by wrapping in higher-level interface that naturally incorporates protection.
- Examples of arbitration and protection?

Moving bits in and out

- Can either do I/O via memory mapping or special instructions. (feature of hardware, not OS)
- Some mix of interrupts and DMA for data path.
- Traps to indicate software conditions/ system calls.

Processes vs Threads

- What do they have in common?
- How are they different?

Scheduling

- OS has to pick which process/thread gets CPU time, and for how long.
- Lots of scheduling algorithms: most real-world ones are some flavor of round-robin.
- Do some sort of feedback to size CPU bursts to process behavior: want interactive processes to get scheduled more often.

Semaphores

- You’ve all built them.

Monitors

- More modern and high-level than semaphores.
- Defines (lexically or dynamically) a region of code in which only one thread is running.
- Lexically: Java's synchronized(foo) {}
- Dynamically: Monitor_enter, Monitor_leave. (Pthreads, windows, etc)

Condition variables

- Monitors alone not enough.
- Condition variables let a thread atomically release monitor lock and stop.
- Can be woken without breaking monitor guarantee. (Though ambiguous who runs next)
- Hoare vs Mesa semantics

Deadlock

- Conditions:
  - ?

Deadlock

- Conditions:
  - Mutual exclusion: resources aren't shared.
  - Hold+Wait: process holding resource can request more
  - No preemption: resources not taken from process.
  - Circular Dependence

Dealing with deadlock

- Many strategies -- either proactive or reactive
- ?

Dealing with deadlock

- Many strategies -- either proactive or reactive
  - Impose total ordering on resources.
  - Have “resource acquisition” operation check for cycles. (Hard, if resources can’t be neatly enumerated)
  - Banker’s algorithm? (Hard in practice, since don’t know maximum need)

Virtual Memory

- Originally, swapping to share scarce RAM; one process at a time loaded into memory.
- Paging allows more fine-grained allocation.
- These days, VM primarily for process isolation.
- Idea is that processes cannot utter name of someone else's storage, so no possibility of corruption.

Paging

- Paging requires separating "virtual" and "physical" addresses; this way same physical address can be used by different processes and system will catch accesses.
- Mapping from virtual to physical address stored in page tables.
- Some hardware support needed: have to catch accesses, and to make this go acceptably fast, need hardware caches (TLB)

Address translation

- Can't have map for every address -- too expensive.
- Map by ***pages*** instead.
- Suppose we have a virtual address `0xAABB1234`.
- Suppose 4k pages. Then last three nybbles are page offset, rest is page tag.
- So we lookup `0xAABB1`, fetch the page, and then add back the `234`.

Worked example

- Let's do it for two-level paging and 40-bit addresses; 4kb pages, tag sizes of 16 and 12 bits.
- Suppose address is `0xAABB123456`.
- Then offset on page is...

Worked example

- Let's do it for two-level paging and 40-bit addresses; 4kb pages, tag sizes of 16 and 12 bits.
- Suppose address is `0xAABB123456`.
- Then offset on page is `0x456`.
- Index into first table is...

Worked example

- Let's do it for two-level paging and 40-bit addresses; 4kb pages, tag sizes of 16 and 12 bits.
- Suppose address is `0xAABB123456`.
- Then offset on page is `0x456`.
- Index into first table is... `0xAABB`.
- And index into second table is....

Worked example

- Let's do it for two-level paging and 40-bit addresses; 4kb pages, tag sizes of 16 and 12 bits.
- Suppose address is `0xAABB123456`.
- Then offset on page is `0x456`.
- Index into first table is... `0xAABB`.
- And index into second table is `0x123`.

Doing the translation

- Same example of `0xAABB123456`.
- First index into outer table to map `0xAABB`.
- Write that down as YYYY.
- Next lookup page table YYYY. Map `0x123` to ZZZ.
- Final address is `0xYYYYZZZ123456`.

Page tables

- Many different page table layouts.
- Either direct, or inverted.
- Inverted relies on hashing to find match.
- Can afford overhead; “almost all” lookups hit TLB instead.
- Cost on context switch; have to refresh TLB

Page ejection

- If doing VM to share RAM, will sometimes need to eject pages from RAM to make way for others.
- Can get ***thrashing*** if pages ejected too often. (Throughput falls drastically)
- Try to do clever algorithms to do as few read-ins as possible.

Caching....

- Physical RAM now looks like a cache for total memory.
- Significant theory behind caching....
- Can't do perfectly; ideal is to eject page that won't be used again for longest time.
- Use LRU or WS as approx.

Working set

- Idea behind WS: track pages used in last t seconds, or last k faults.
- Keep those working sets if possible; else suspend and swap out a process.
- Hard to really implement.
- Tracking page access is expensive.
- Real systems try to approximate LRU and WS.