Slide 1

Administrivia

- (By e-mail.)

Slide 2

Mutual Exclusion — Review/Recap

- Recall problem: Add something to generic code that enforces that only one process at a time can be in a “critical region”. Equivalent to implementing locks.
- Several non-working solutions proposed, then finally one approach (Peterson’s algorithm) that at least guarantees mutual exclusion, but has shortcomings:
- Anything that uses shared variables requires some attention on modern hardware given how writes to RAM actually work.
- Blocking by busy-waiting might not be fair, and isn’t efficient.
- To do better, need help from hardware and from O/S (or other library).

Slide 3

Sidebar: TSL Instruction

- A key problem in concurrent algorithms — "atomicity" (operations guaranteed to execute without interference from another CPU/process). Hardware can provide some help with this.
- E.g., "test and set lock" (TSL) instruction:

  `TSL registerX, lockVar`

  (1) copies `lockVar` to `registerX` and (2) sets `lockVar` to non-zero, *all as one atomic operation*.

  How to make this work is the hardware designers' problem!
- Note that this is very much like textbook's `TestAndSet` instruction. Most current hardware provides similar instruction(s); textbook describes several. Recall `ll` and `sc` from CSCI 2321.

Slide 4

Proposed Solution Using TSL Instruction

- Shared variables:

```
int lock = 0;
```

Pseudocode for each process:

```
while (true) {
    enter_cr();
    do_cr();
    leave_cr();
    do_non_cr();
}
```

Assembly-language routines:

```
enter_cr:
    TSL regX, lock
    compare regX with 0
    if not equal
        jump to enter_cr
    return
leave_cr:
    store 0 in lock
    return
```

- Does it work? Yes . . .

Slide 5

Solution Using TSL Instruction, Continued

- Proposed invariant: "`lock` is 0 exactly when no processes in their critical regions, and nonzero exactly when one process in its critical region." ("Exactly when" here means "if and only if".)
- If this invariant holds, that means first requirement is met. (Does it hold? Next slide.) Others met too — well, except that it might be "unfair" (some process waits forever).
- Is this a better solution? Simpler than Peterson's algorithm, but still involves busy-waiting. (Also depends on hardware features that *might* not be present, but these days almost all hardware has something similar.)

Slide 6

Solution Using TSL Instruction, Continued

- Proposed invariant: "`lock` is 0 exactly when no processes in their critical regions, and nonzero exactly when one process in its critical region." ("Exactly when" here means "if and only if".)
- True initially.
- Could change when a process enters its critical region — but notice that only happens when `lock` is 0.
- Also doesn't change when a process leaves its critical region.
- So okay.

Mutual Exclusion — Recap

- So with help from special instructions such as TSL, we have something that mostly solves the mutual-exclusion problem — “spin lock” (because a process/thread spins if lock not available).
- One problem — inefficient. Could address that with revision to `enter_cr()`:

```
enter_cr:
    TSL registerX, lockVar
    compare registerX with 0
    if equal, jump to ok
    invoke scheduler # thread yields to another thread
    jump to enter_cr
ok:
    return
```

- But fairness still not guaranteed, and this seems pretty low-level, so might be hard to use for more complicated problems.
- So, people have proposed various “synchronization mechanisms” — more-abstract ways of coordinating what processes do. A key point is providing *something* that potentially makes a process wait.

Slide 9

Semaphores

- History — 1965 paper by Dijkstra (possibly earlier work by Iverson, or so says a former faculty member who knows of Iverson through his work on APL/J).
- Idea — define semaphore ADT:
  - "Value" — non-negative integer.
  - Two operations, *both atomic*:
    - up (V) — add one to value.
    - down (P) — block until value is nonzero, then subtract one.
- Ignoring for now how to implement this — is it useful?

Slide 10

Mutual Exclusion Using Semaphores

- Shared variables:

  ```
      semaphore S(1);
  ```

  Pseudocode for each process:

  ```
  while (true) {
      down(S);
      do_cr();
      up(S);
      do_non_cr();
  }
  ```

- Proposed invariant: "S has value 1 exactly when no process in its critical region, 0 exactly when one process in its critical region, and never has values other than 0 or 1."

Mutual Exclusion Using Semaphores, Continued

Slide 11

- Proposed invariant again: "S has value 1 exactly when no process in its critical region, 0 exactly when one process in its critical region, and never has values other than 0 or 1."
- True initially.
- Could change when a process enters its critical region — but this is essentially exactly when a `down(S)` completes, so okay.
- Could change when a process leaves its critical region — but this is essentially exactly when an `up(S)` completes, so okay.

Classical IPC Problems — Review/Recap

Slide 12

- Problems meant to represent many commonly-occurring situations in which processes have to coordinate in some way.
- We've talked about one — mutual exclusion — but there are others. Next . . .

Slide 13

Bounded Buffer Problem

- (Example of slightly more complicated synchronization needs.)
- Idea — we have a buffer of fixed size (e.g., an array), with some processes ("producers") putting things in and others ("consumers") taking things out. Synchronization:
  - Only one process at a time can access buffer.
  - Producers wait if buffer is full.
  - Consumers wait if buffer is empty.
- Example of use: print spooling (producers are jobs that print, consumer is printer — actually could imagine having multiple printers/consumers).

Slide 14

Bounded Buffer Problem, Continued

- Shared variables:

```
buffer B(N); // initially empty, can hold N things
```

Pseudocode for producer:

```
while (true) {
    item = generate();
    put(item, B);
}
```

Pseudocode for consumer:

```
while (true) {
    item = get(B);
    use(item);
}
```

- Synchronization requirements:

1. At most one process at a time accessing buffer.
2. Never try to `get` from an empty buffer or `put` to a full one.
3. Processes only block if they "have to".

Slide 15

Bounded Buffer Problem, Continued

- We already know how to guarantee one-at-a-time access. Can we extend that?
- Three situations where we want a process to wait:
  - Only one get/put at a time.
  - If B is empty, consumers wait.
  - If B is full, producers wait.

Slide 16

Bounded Buffer Problem, Continued

- What about three semaphores?
  - One to guarantee one-at-a-time access.
  - One to make producers wait if B is full — so, it should be zero if B is full — “number of empty slots”?
  - One to make consumers wait if B is empty — so, it should be zero if B is empty — “number of slots in use”?

Slide 17

Bounded Buffer Problem — Solution

- Shared variables:

```
buffer B(N); // empty, capacity N
semaphore mutex(1);
semaphore empty(N);
semaphore full(0);
```

Pseudocode for producer:

```
while (true) {
    item = generate();
    down(empty);
    down(mutex);
    put(item, B);
    up(mutex);
    up(full);
}
```

Pseudocode for consumer:

```
while (true) {
    down(full);
    down(mutex);
    item = get(B);
    up(mutex);
    up(empty);
    use(item);
}
```

Slide 18

Semaphores – Review

- A “synchronization mechanism” — way of controlling interaction among processes in a more abstract way than the first few solutions to the mutual exclusion problem.
- Semaphore as ADT:
  - “Value” — non-negative integer.
  - Two operations, “up” and “down”, *both atomic*.
- Allows for nice solution for mutual exclusion, also ability to solve more complex problems (e.g., bounded buffer).

Slide 19

Implementing Semaphores

- We want to define:
  - Data structure to represent a semaphore.
  - Functions `up` and `down`.
- `up` and `down` should work the way we said, and we'd like to do as little busy-waiting as possible.

Slide 20

Implementing Semaphores, Continued

- Idea — represent semaphore as integer plus queue of waiting processes (represented as, e.g., process IDs).
- Then how should this work . . .

Slide 21

Implementing Semaphores, Continued

- Variables — integer `value`, queue of process IDs `queue`.

```
down() {
    bool zero;
    enter_cr();
    zero = (value == 0);
    if (!zero)
        value -= 1;
    else
        enqueue(current_process, queue);
    leave_cr();
    if (zero)
        block();    // mark current process blocked
}
```

```
up() {
    process p = null;
    enter_cr();
    if (empty(queue))
        value += 1;
    else
        p = dequeue(queue);
    leave_cr();
    if (p != null)
        unblock(p);     // mark p runnable
}
```

- `enter_cr()`, `leave_cr()` as described previously.

Slide 22

Sidebar: Shared Memory and Synchronization

- Solutions that rely on variables shared among processes assume that assigning a value to a variable actually changes its value in memory (RAM), more or less right away. Fine as a first approximation, but reality may be more complicated, because of various tricks used to deal with relative slowness of accessing memory:

  Optimizing compilers may keep variables' values in registers, only reading/writing memory when necessary to preserve semantics.

  Hardware may include cache, logically between CPU and memory, such that memory read/write goes to cache rather than RAM. Different CPUs' caches may not be in synch (though this is something the hardware takes care of in sensible systems?).

Sidebar: Shared Memory and Synchronization, Continued

- So, actual implementations need notion of “memory fence” — point at which all apparent reads/writes have actually been done. Some languages provide standard ways to do this; others (e.g., C!) don’t. C’s `volatile` (“may be changed by something outside this code”) helps some but may not be enough.
- Worth noting, however, that many library functions / constructs include these memory fences as part of their APIs (e.g., Java `synchronized` blocks).

Minute Essay

- Does what I’m saying about using invariants to reason about concurrent algorithms make sense to you?
- Other questions?