Semaphore Implementation

ECE595
Jan 27

Y. Charlie Hu

Synchronization Primitives provided by OS (language/compiler)

- Lock
  - Alone is not powerful enough
- Semaphore (incl. binary semaphore)
  - binary semaphore alone not enough
- Lock and condition variable
- Monitor (hide lock, still use condition variables)

Semaphore

```
wait(S) {
   while (S<=0);
   S--;
}
```

```
signal(S) {
   S++;
}
```

- Semantics:
  - S "counts" the number of "resources"
  - wait(S) signifies "consuming" one if there is any, otherwise wait!
  - signal(S) signifies "producing" one

Semaphore implementation

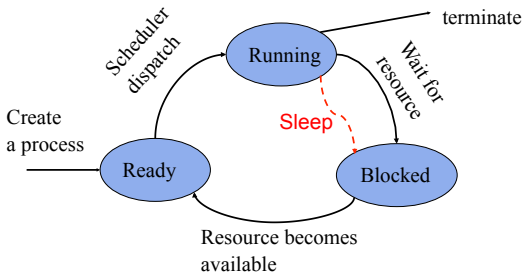
```
wait(S) {
   while (S<=0);
   S--;
}
```

```
signal(S) {
   S++;
}
```

- Can they be implemented in the user space?
  - An intuitive argument?
- No existing hardware implements them directly
  - Scheduling/queuing cannot be easily done in HW

➔ Semaphore must be done in OS, typically with low-level synchronization support from hardware

[lec 2] Process State Transition



Uniprocessor solution

```
typedef struct {
  int count;
  queue q;
} semaphore;

void wait(semaphore s)
{
  if (s->count > 0) {
    s->count --;
    return;
  }
  add(s->q, current_process);
  sleep();
}
```

```
void signal(semaphore s)
{
  if (isEmpty(s->q)) {
    s->count ++;
  } else {
    process = removeFirst(s->q);
    wakeup(process);
    /* put process on Ready Q */
  }
}
```

- Wake up a sleeping process == no op to counter

Challenge

- Need to make primitives atomic!
- What MutEx support do we have from HW on uniprocessor?
  - On uniprocessor, reads and writes are atomic

Uniprocessor solution

```
typedef struct {
  int count;
  queue q;
} semaphore;

void wait(semaphore s)
{
  if (s->count > 0) {
    s->count --;
    return;
  }
  add(s->q, current_process);
  /* imply sleep(); */
}
```

```
void signal(semaphore s)
{
  if (isEmpty(s->q)) {
    s->count ++;
  } else {
    process = removeFirst(s->q);
    wakeup(process);
    /* put process on ReadyQ */
  }
}
```

Uniprocessor solution

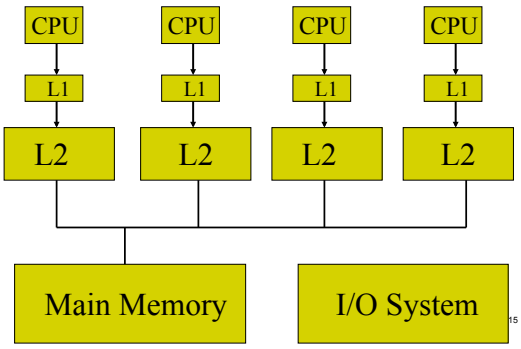
- What can cause the few lines to be not atomic?
- What causes context switches?
- Recall -- only way the OS dispatcher regains control is via interrupts (incl. HW interrupt like timer, SW interrupt like syscalls)
  - E.g. typing -> keyboard interrupt -> handler -> kernel -> user process

Uniprocessor solution: disable interrupts!

```
void  wait(semaphore s)
{
  disable interrupts;
  if (s->count > 0) {
     s->count --;
     enable interrupts;
     return;
  }
  add(s->q, current_process);
  enable interrupts;
  /* implying sleep();*/
}
```

```
void  signal(semaphore s)
{
   disable interrupts;
   if (isEmpty(s->q)) {
     s->count ++;
   } else {
     process = removeFirst(s->q);
     wakeup(process);
     /* put process on Ready Q */
   }
   enable interrupts;
}
```


Generic shared-memory multiprocessor



What about multiprocessors?

- True concurrency – simultaneity!
  - Cache coherence in HW (fairy complicated)
  - For simplicity: a read sees most recent write
- Is turning off interrupt tlocally enough?
- Turning off interrupt on all other processors?
- Use atomic read/write and busy waiting, as in “too much milk”?

[Ice3] Yet Another Possible Solution?

process A

```
leave noteA
while (noteB)
  do nothing;
if (noMilk)
  buy milk;
remove noteA
```

process B

```
leave noteB
if (noNoteA) {
  if (noMilk) {
    buy milk
  }
}
remove noteB
```

- Safe to buy
- If the other buys, I won't
- 2 things we dislike this solution?

A Multiprocessor solution?

```
void  wait(semaphore s)
{
  disable interrupts;
  while (s->count == 0);

  s->count --;
  enable interrupts;
}
```

```
void  signal(semaphore s)
{
  disable interrupts;
  s->count ++;
  enable interrupts;
}
```

- Does this work?

What about multiprocessors? (cont)

- Cannot just turn off interrupts
  - Turn off interrupt on local processor?
  - Turn off all other processors?
- Use atomic read/write and busy waiting, as in "too much milk"?
  - → Concurrent read can happen
- Need more help from HW!
- What is the right, minimal HW support?
  - Hot research topic for a long time

[Lec2] A Possible Solution?

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

```
if ( noMilk ) {
  if (noNote) {
    leave note;
    buy milk;
    remove note;
  }
}
```

- process can get context switched after checking milk and note, but before leaving note
- *Why does it work for human?*

Read-modify-write on CISC

- Most CISC machines provide atomic *read-modify-write* instruction
  - read existing value
  - store back a new value
  - Example: *test-and-set* by IBM and others

```
X = TAS(&lock, 1);

read lock value V;
if 0 set lock to 1 else noop;
return V;
```


Using test-and-set for mutual exclusion (too-much milk)

- Implement a critical section on multiprocessor
  - Prevents 2 processes doing 0-to-1 transition simultaneously

```
global int lock = 0;
…
??????
…
critical section
…
??????
```


Use TAS to implement semaphores on multiprocessor

- For each semaphore, keep an extra integer (lock)

```
typedef struct {
  int lock; /* initially 0 */
  int count;
  queue q; /* queue of procs waiting on this semaphore */
} semaphore;
```


Use TAS to implement semaphores on multiprocessor?

```
void  wait(semaphore s)
{
  disable interrupts;
  ???
  if (s->count > 0) {
     s->count --;
     ???
     enable interrupts;
     return;
  }
  add(s->q, current_process);
  ???
  /* implying sleep(); */
  enable interrupts;
}
```

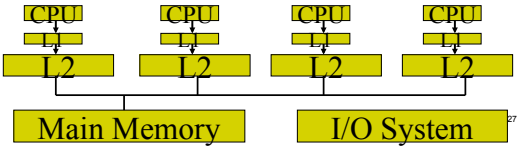
```
void  signal(semaphore s)
{
   disable interrupts;
   ???
   if (isEmpty(s->q)) {
      s->count ++;
   } else {
      thread = removeFirst(s->q);
      wakeup(process);
      /* put process on Ready Q */
   }
   ???
   enable interrupts;
}
```

Really Deep Thinking

- Why is this busy waiting not a concern?

Weaker mechanism on RISC [MIPS R4000 series]

- *Load-linked* instruction: LDL Rx,y
  - loads Rx with a word from mem addr y
  - holds y in per-processor lock register
- Store operation to addr y (by any processor) resets all other processors' lock registers if containing addr y
- *Store-conditionally* instruction: STC Rx, y
  - stores a word iff y matches the processor's lock register
  - indicates success (1) or failure (0)



Weaker mechanism on RISC [MIPS R4000 series]

- *Load-linked* instruction: LDL Rx,y
  - loads Rx with a word from mem addr y
  - holds y in per-processor lock register
- Store operation to addr y (by any processor) resets all other processors' lock register if containing addr y
- *Store-conditionally* instruction: STC Rx, y
  - stores a word iff y matches the processor's lock register
  - indicates success (1) or failure (0)

```
int lock=0;
…
while (ldl(&lock) == 1) || stc((&lock, 1) == 0);

…
critical section
…
lock = 0;
```


Use ldl/stc to implement semaphores on multiprocessor

```
void wait(semaphore s)
{
  disable interrupts;
  ???
  if (s->count > 0) {
    s->count --;
    ???
    enable interrupts;
    return;
  }
  add(s->q, current_process);
  ???
  /* implying sleep(); */
  enable interrupts;
}
```

```
void signal(semaphore s)
{
  disable interrupts;
  ???
  if (isEmpty(s->q)) {
    s->count ++;
  } else {
    thread = removeFirst(s->q);
    wakeup(process);
    /* put process on Ready Q */
  }
  ???
  enable interrupts;
}
```

Implementing Locks and Condition Variables

- Use the same mechanisms for
  - achieving atomicity and
  - avoiding busying waiting

  as in semaphore implementation

Reading assignment

- Chapter 6