Semaphore Implementation

ECE469, Jan 26

Yiying Zhang

5. Readers-Writers problem
19. Semaphore implementation
32. Using test-and-set(TAS) for mutual exclusion
39. Implementing Locks + Condition Variables
47. Load-Linked and Store-Conditional

Reading

- Dinosaur Ch 6
- Comet Ch 28
- Lab 2 starts

Big Picture So Far

- Getting synchronization right is hard concurrent process
  - Even some of your esteemed faculty members (e.g., Yiying) can get it wrong.
- How to pick between locks, semaphores, conditional variables, monitors???
- Locks are very simple for many cases.
  - Issues: Maybe not be the most efficient solution
  - For example, can't allow multiple readers but one writer inside a standard lock.
- Condition variables allow threads to sleep while holding a lock
  - Just be sure you understand whether they use Mesa or Hoare semantics!
- Semaphores provide pretty general functionality
  - But also make it really easy to botch things up.
- Monitors are a higher-level semantics for using locks and condition variables.

Semaphore classic examples

1. Producer-consumer problem

2. Readers-writers problem

3. Dining philosophers problem

[lec5] Semaphore classic problem 2: Readers-Writers problem

- A data object is shared among multiple processes
- Allow concurrent reads (but no writes)
- Only allow exclusive writes (no other writes or reads)

[lec5] Readers-Writers problem (Solution 1)

- Constraints:
  - Writers can only proceed if there are no readers/writers
  - Readers can proceed only if there are no writers
  - ➔ use a single semaphore BlockWrite
  - To keep track of how many are reading
  - ➔ use a shared variable
  - Only one process manipulates state variable at once
  - ➔ use semaphore Mutex

- Initialization:
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - semaphore Mutex = 1; // binary semaphore (basic lock)
  - int Readers = 0; // count of readers reading in critical section

Writer

P(BlockWrite); // wait to lock the shared resource for a writer

< Do the Writing >

V(BlockWrite);

Reader

```
P(Mutex);
Readers++;
if (Readers == 1) // first reader acquire write lock
            P(BlockWrite);
V(Mutex);

< Do the Reading >

P(Mutex);
Readers--;
if (Readers == 0) // last (only) reader releases write lock
V(BlockWrite);
V(Mutex);
```

What will happen in different scenarios?

1. The first reader blocks if there is a writer; any other readers who try to enter block on mutex.
2. The last reader to exit signals a waiting writer.
3. When a writer exits, if there is both a reader and writer waiting, which goes next depends on the scheduler.
4. If a writer exits and a reader goes next, then all readers that are waiting will fall through.
5. Does this solution guarantee all threads will make progress?

- Writes can starve
- => Read preference

once reader get lock, and many reader following, writer cannot get the lock

[lec4] What is a good solution

- Only one process inside a critical section
- Processes outside of critical section should not block other processes
- No one waits forever
- No assumption about CPU speeds
- Works for multiprocessors

Readers-Writers problem (Solution 2)

- How do we let reads yield to writes?
  - int Readers = 0, Writers = 0; // count of readers and writers in critical section
  - semaphore BlockWrite = 1; // used to allow ONE writer or MANY readers
  - semaphore BlockRead = 1; // used to block readers
  - semaphore RMutex = 1; // binary semaphore for Readers
  - Semaphore WMutex = 1; // binary semaphore for Writers

Write

```
P(WMutex);
Writers++;
if (Writers == 1) // block readers
        P(BlockRead);
V(Wmutex);

P(BlockWrite); // ensures only one writer
< Do the Writing >
V(BlockWrite);

P(WMutex);
Writers--;
if (Writers == 0) // enable readers
        V(BlockRead);
V(WMutex);
```

Reader

```
P(BlockRead); // at most one reader can go before a pending write
P(RMutex);
Readers++;
if (Readers == 1) // first reader acquire write lock
        P(BlockWrite);
V(RMutex);
V(BlockRead);

< Do the Reading >

P(RMutex);
Readers--;
if (Readers == 0) // last (only) reader releases write lock
        V(BlockWrite);
V(RMutex);
```

- Any problem?

Problem of solution 2

- Reader starvation

- Is there a solution that's fair to both reads and writes?
    - Take-home puzzle problem (not graded)

Reader-writer problem a possible fair solution?

- An idea: use a FIFO queue for all readers and writers

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)

What are the inner workings of these powerful synchronization tools

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

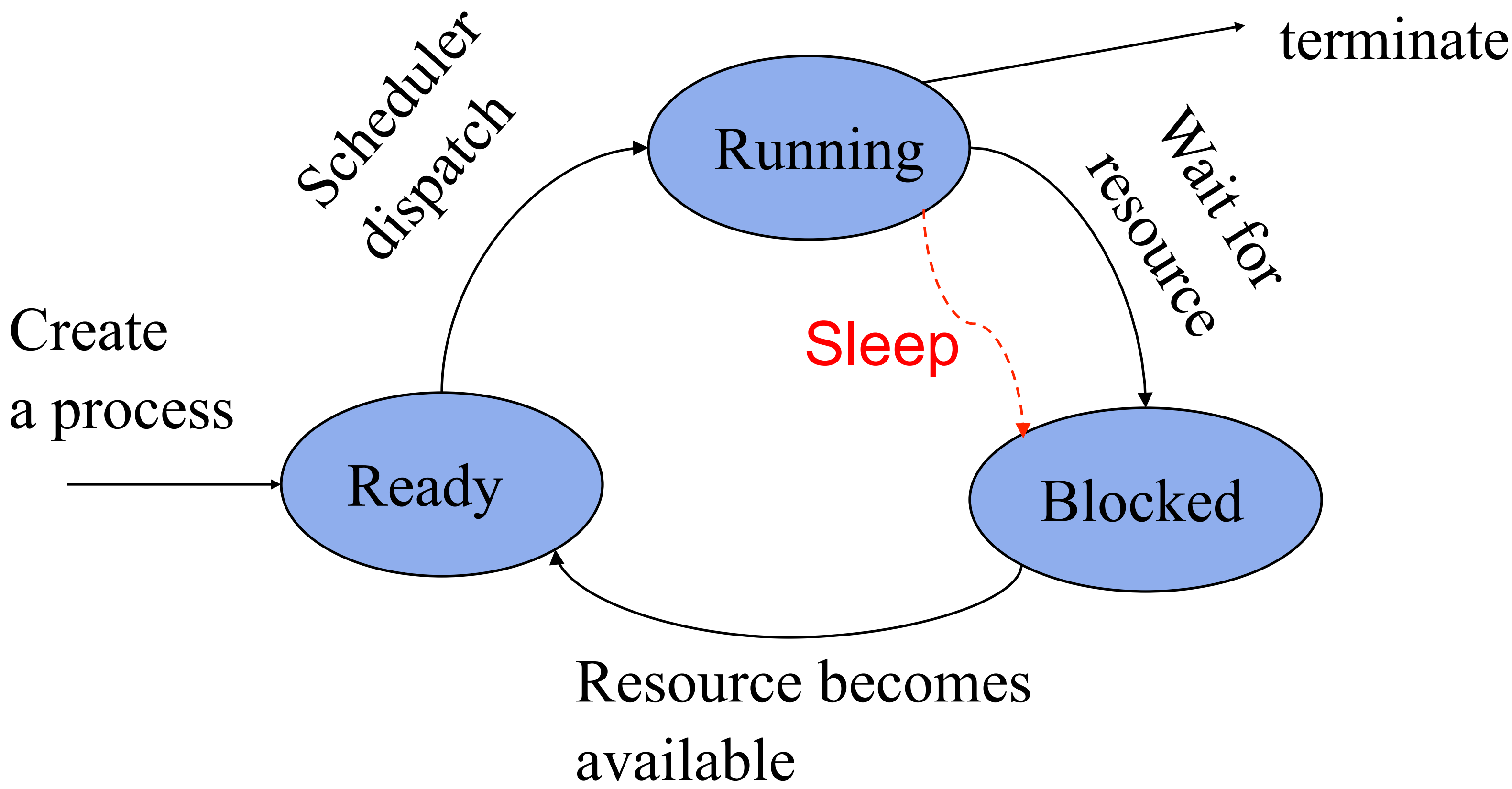
```
wait(S) {
    while (S<=0);
    S--;
}
```

```
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 3] Process State Transition



Uniprocessor solution

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

```
void  wait(semaphore s)
{

   if (s->count > 0) {
      s->count --;
// sleep -> do not need while loop
      return;
   }
   add(s->q, current_process);

   sleep();
}
```

```
void  signal(semaphore s)
{

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

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

- What can cause the few lines to be not atomic?

- What causes context switches?

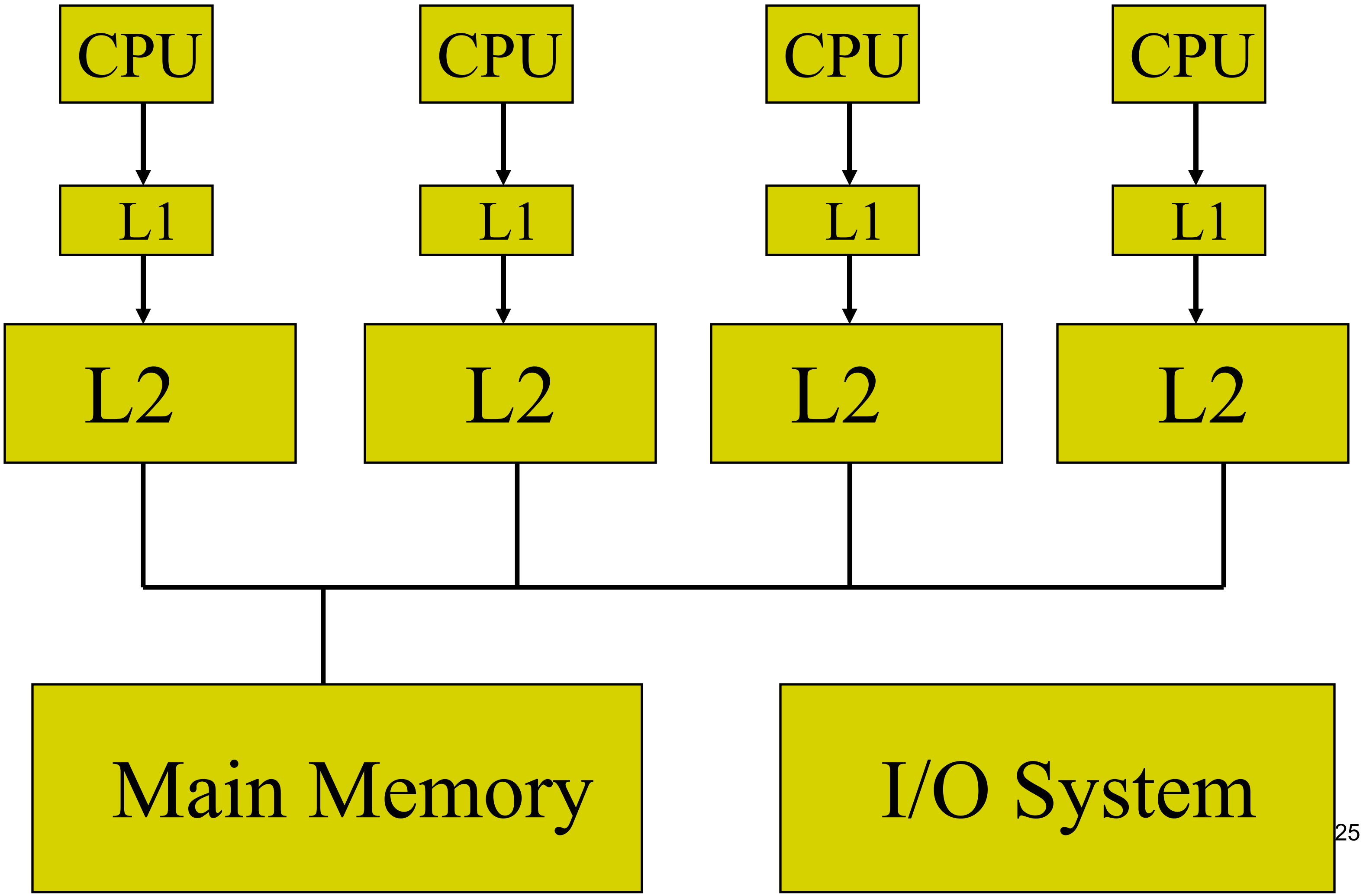
- Recall -- only way the OS dispatcher regains control is via interrupts (incl. explicit requests, i.e. syscalls)
  - E.g. typing -> keyboard interrupt -> handler -> kernel -> user process

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(); // re-dispatch
}
```

```
void  signal(semaphore s)
{
   disable interrupts;
   s->count ++;
   if (!isEmpty(s->q)) {
     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 interrupts enough?
  - Turning off interrupt on a processor does not prevent other processors from accessing shared memory
  - Turning off interrupt on all other processors?
- Use atomic read/write and busy waiting?

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 solution work?

- Did we achieve atomicity of the while loop (busy wait) and the decreasing of s->count?
  no because ++ is more than one instruction
- Busy waiting on local processor is not efficient
- What if the signaler is scheduled on the same core as wait?
- What about concurrent memory access (s->count)?

What about multiprocessors? (cont)

- Cannot just turn off interrupts
  - Turn off interrupt on a processor does not prevent other processors from accessing shared memory
  - Turn off all other processors?
- Use atomic read/write and busy waiting?
  - → Concurrent read can happen
  - → "Busy waiting" is not acceptable

- Need more help from HW!
- What is the right HW support?
  - Hot research topic for a long time

[lec4] 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

```
int TAS(int *old_ptr, int new {
 int old = *old_ptr; // fetch old value at old_ptr
 *old_ptr = new; // store 'new' into old_ptr
 return old; // return the old value
}
```

  - Using TAS to implement lock (mutex)

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
…
??????
```

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

- Implementation with spin lock (busy wait)

```
global int lock = 0;
...
while (TAS(&lock, 1) == 1);
...
critical section
...
lock = 0;
```

Evaluating spin lock implementation with TAS

- Correctness?
  - Yes (assume preemptive scheduler on uniprocessor)
- Performance?
  - No, esp. when critical section is long

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;
 while (1 == tas(&lock,1));
  if (s->count > 0) {
     s->count --;
     ???
     enable interrupts;
     return;
  }
  add(s->q, current_process);
  ???
  sleep(); /* re-dispatch */
  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;
}
```

Use TAS to implement semaphores on multiprocessor?

```
void  wait(semaphore s)
{
   disable interrupts;
  while (1 == tas(&lock,1));
   if (s->count > 0) {
       s->count --;
       lock = 0;
       enable interrupts;
       return;
   }
   add(s->q, current_process);
   lock=0;
   sleep(); /* re-dispatch */
   enable interrupts;
}
```

```
void  signal(semaphore s)
{
    disable interrupts;
    while (1 == tas(&lock,1));
    if (isEmpty(s->q)) {
       s->count ++;
    } else {
       thread = removeFirst(s->q);
       wakeup(process);
       /* put process on Ready Q */
     }
    lock = 0;
     enable interrupts;
}
```

Deep Thinking

- Why is this busy waiting not a concern?
  - What’s our critical section here?

- Can we remove the disable/enable interrupts?
  - With interrupts, when a process that gets the lock (pass the while loop) get context switched out
  - All other wait processes on other cores will be busy waiting

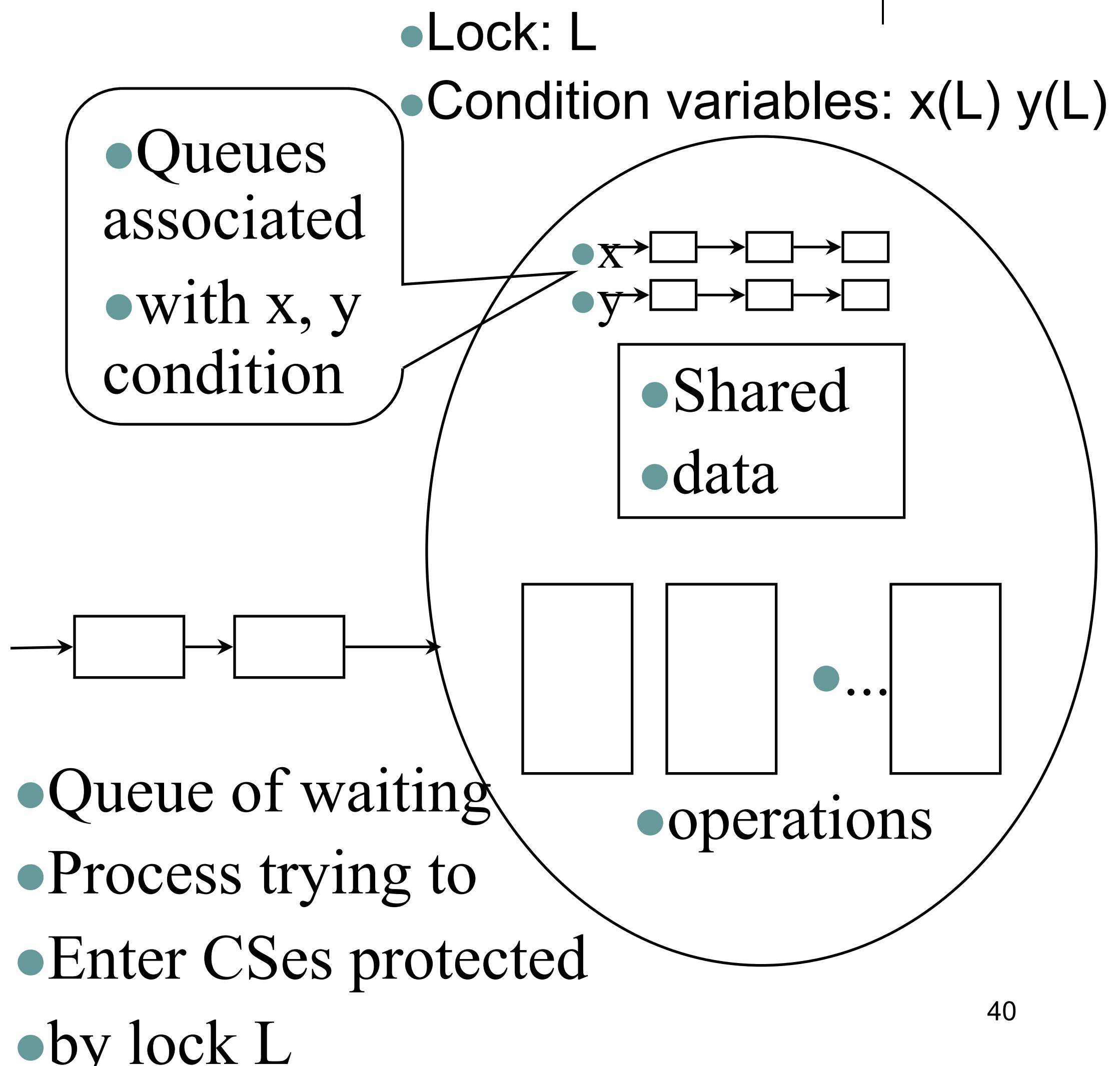
Implementing Locks+Condition Variables

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

  as in semaphore implementation

[lec5] Condition Variables

- Wait (condition)
  - Block on "condition"
- Signal (condition)
  - Wakeup a process blocked on "condition"
- Conditions are like semaphores but:
  - signal is no-op if none blocked
  - There is no counting!



[lec5] Two Options of Signaler

- Relinquishes control to the awaken process; suspend signaler (Hoare-style, early time)
  - Signaler gives up lock, waiter runs immediately
  - Waiter gives back lock and CPU to signaler after critical sec.
  - Complex if the signaler has other work to to
  - In general, easy to prove things about system (e.g. fairness)

- Continues its execution (Mesa-style, modern)
  - Signaler keeps lock and CPU
  - Waiter put on ready queue
  - Easy to implement (e.g., no need to keep track of signaler)
  - But, what can happen when the awaken process gets a chance to run?
    - E.g. producer 1 context switch at line 1, producer 2 wait, consumer 1 signals producer 2

Roadmap on Process Management

- Processes
- Need for process synchronization
- Mutual exclusion & Critical section
- “Too much milk” problem

- Semaphore (binary semaphore)
- Producer-consumer problem
- Condition variable (Monitor)

- *Time to dive inside OS*

Busy waiting does not work – We are going back to where we started (in a way)?

- Uniprocessor (read/write atomic)
- Concurrent execution → needs busy waiting
  - Busy waiting in apps not efficient → Atomic sync primitive from OS: e.g. semaphore
  - OS has to implement semaphore atomically → disable interrupt
- Multiprocessor (read/write concurrent)
  - Concurrent read/write by processes on processors
  - If do not disable interrupt on all processors?
    - Needs to bring back busy-waiting to help
    - Concurrent execution: No busy waiting -> no mutual exclusion -> no atomicity -> No P & V

Semaphores vs. lock/condition variables

- survey:
  - How many feel lock/cond variables are easier?
- My carefully designed lecture was unintentionally sabotaged by …
  - The sequence of using lock, using semaphore, and then using lock/cond variables
- We then dived into the inner workings condition variables
  - Which is not a bad because we needed to understand
  - Along way, many engr ideas optimizing perf came up
  - They were all good ideas, showing we engr can opt perf

Sema vs. lock/condition variables

- But let us not loose sight on the big picture
  - That lock/condition vars are more natural
- Let us decouple
  - How easy it is to program (use)
    - Lock/cond variables fit naturally the programming thought process (evolution from lock())
      - Details of condition left to programmers → more flexible
    - Semaphores are like revolution, clean-slate new design
      - Bundle condition (i.e., resource counting) with sync variable (i.e., semaphore) may not be easy in general
  - How easy it is to understand how ** works internally

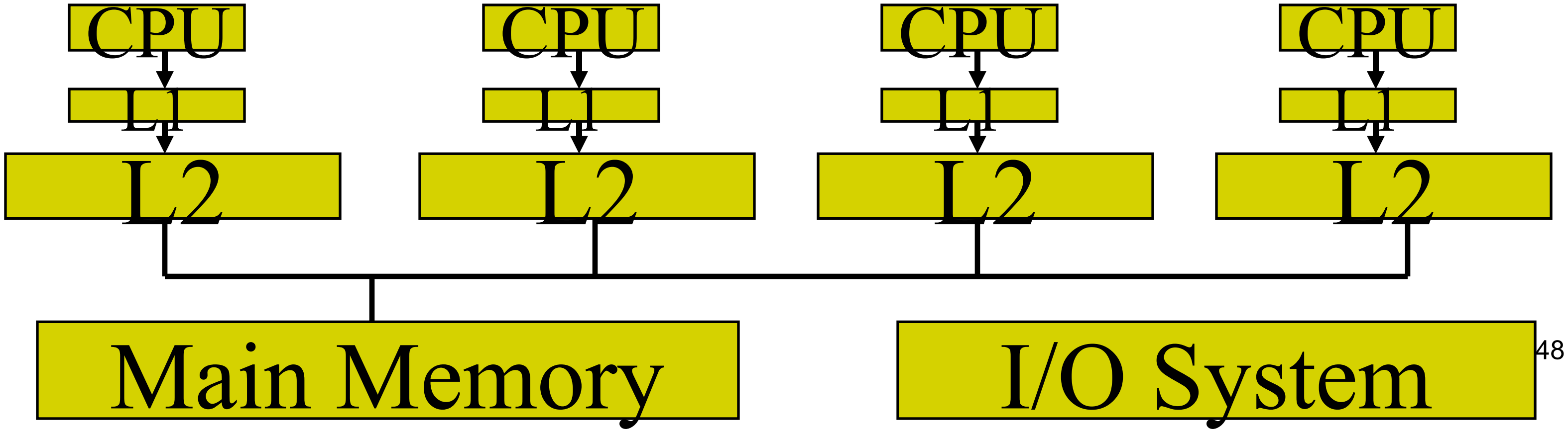
Backup Slides

Load-Linked and Store-Conditional

- Load-linked
  - Operates like a typical load instruction, and simply fetches a value from memory and places it in a register.
- Store-conditional
  - Only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place.

Load-Linked and Store-Conditional 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-conditional* instruction: STC Rx, y
  - stores a word iff y matches the processor's lock register
  - indicates success (1) or failure (0)



Using load-linked and store-conditional to implement locks

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

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

can ldl,

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);
  ???
  sleep(); /* re-dispatch */
  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;
}
```

Use ldl/stc to implement semaphores

```
void  P(semaphore s)
{
   disable interrupts;
   while (ldl(s->lock) == 1 ||
              stc(s->lock, 1) == 0);
   if (s->count > 0) {
      s->count --;
      s->lock = 0;
      enable interrupts;
      return;
   }
   add(s->q, current_process);
   s->lock = 0;
   sleep(); /* re-dispatch */
   enable interrupts;
}
```

```
void  V(semaphore s)
{
   disable interrupts;
   while (ldl(s->lock == 1) ||
              stc(s->lock, 1) == 0);
   if (isEmpty(s->q)) {
      s->count ++;
   } else {
      thread = removeFirst(s->q);
      wakeup(process);
      /* put process on Ready Q */
   }
   s->lock = 0;
   enable interrupts;
}
```