Real Time and Embedded Systems

by
Dr. Lesley Shannon
Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc351

Simon Fraser University

Slide Set: 3
Date: September 20, 2011

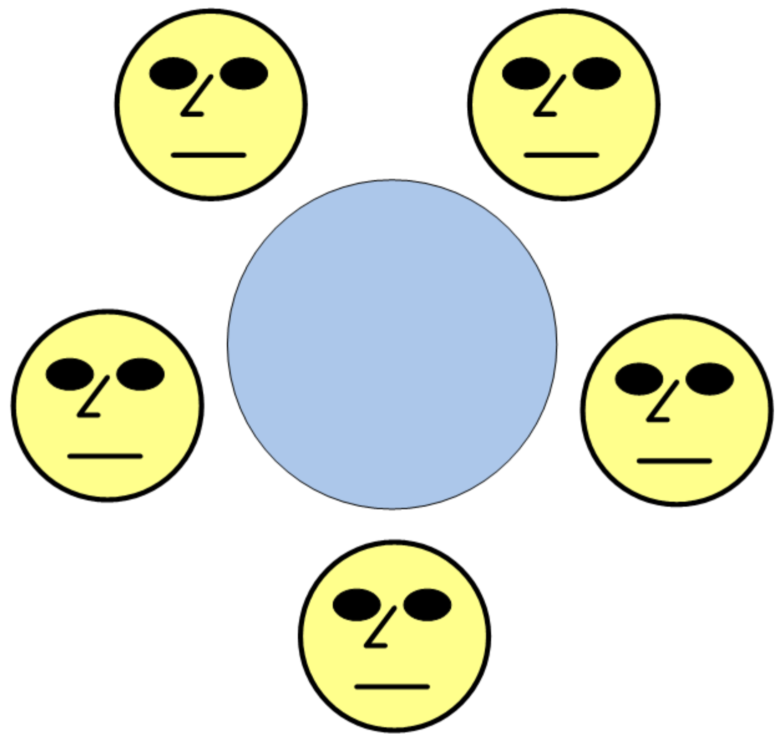
Slide Set Overview

- Synchronization
- Mutexes
- Semaphores
- Classic IPC and Synchronization Problems

Synchronization

The Dining Philosopher's Problem

- Five philosophers sit around a circular table.

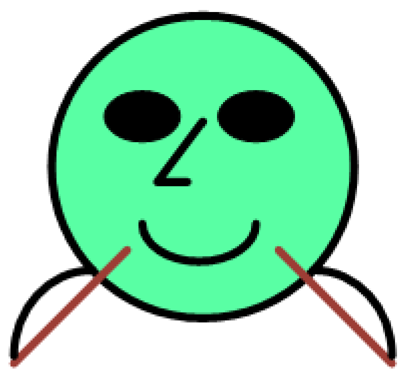


The Dining Philosopher's Problem

- Each philosopher spends his life alternatively:

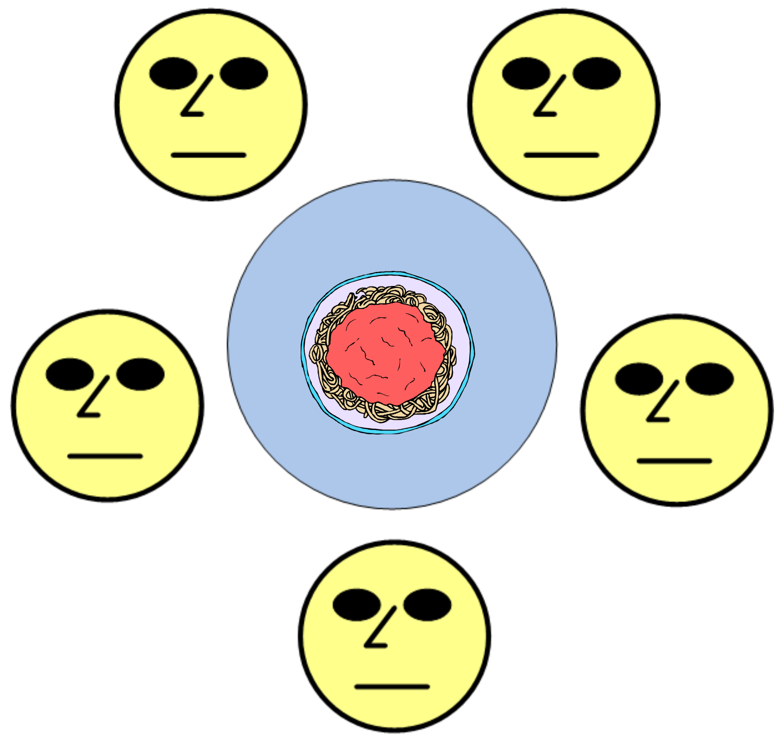
thinking and…

eating.



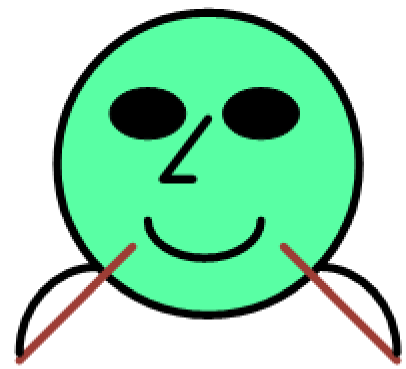
The Dining Philosopher's Problem

- In the centre of the table is a large plate of noodles.



The Dining Philosopher's Problem

- A philosopher needs two chopsticks to eat a helping of noodles.

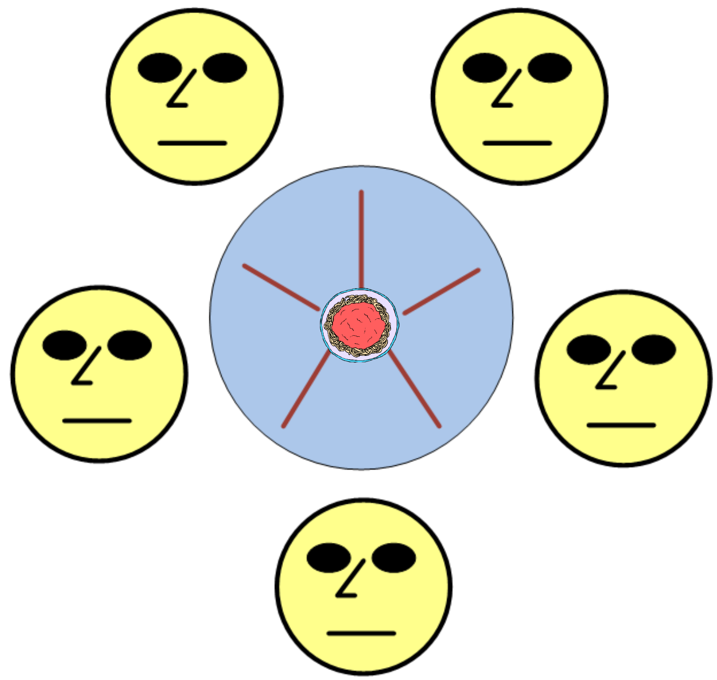


The Dining Philosopher's Problem

- Unfortunately, philosophy is not as well paid as computing

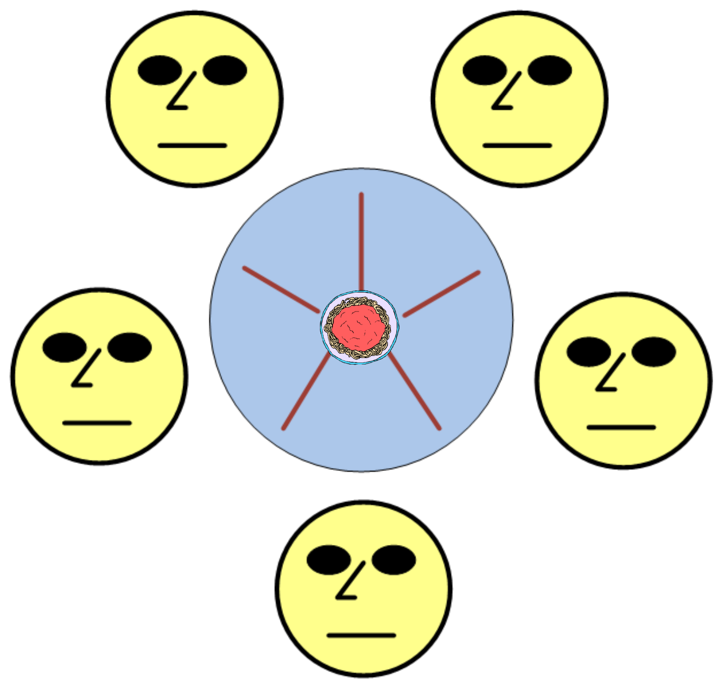
The Dining Philosopher's Problem

- The philosophers can only afford five chopsticks. One chopstick is placed between each pair of philosophers.



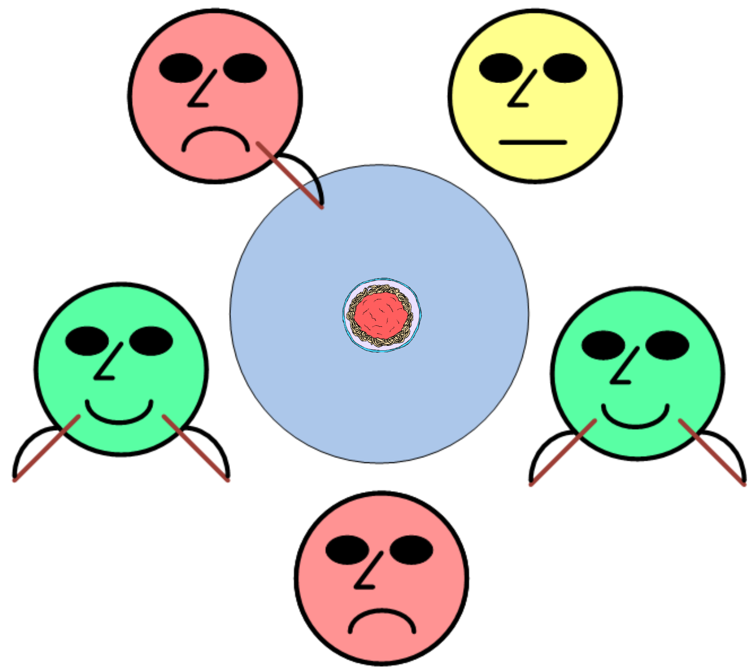
The Dining Philosopher's Problem

- The philosophers agree that each will only use the chopstick to his immediate right and left



The Dining Philosopher's Problem

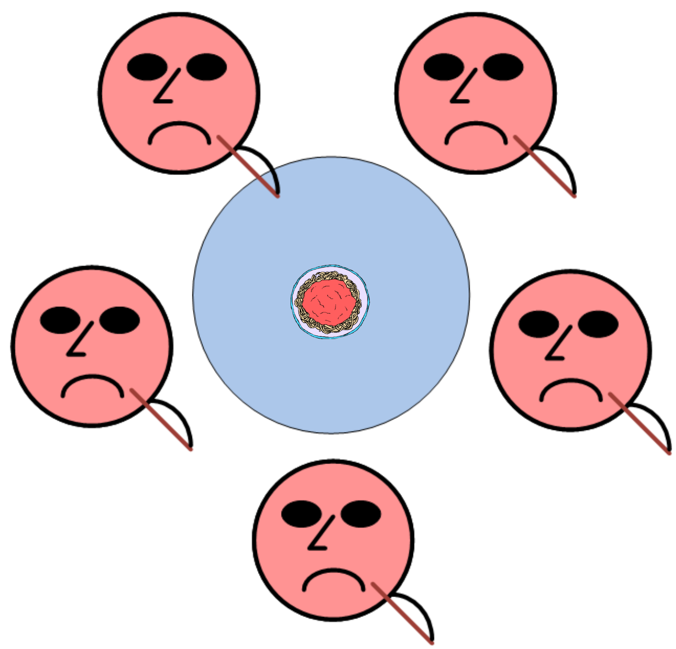
- Philosophers are depicted in yellow when they are thinking, red when hungry and green when eating.



Why do we need synchronization?

The Dining Philosopher's Problem

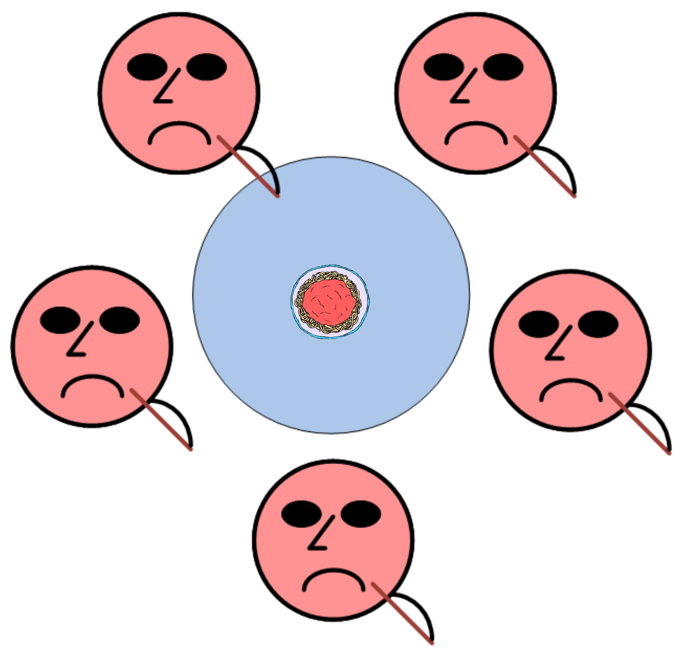
- What if all of the philosophers decide to eat at the same time?
- And what if they are prepared to wait to eat?



The Dining Philosopher's Problem

- What if all of the philosophers decide to eat at the same time?
- And what if they are prepared to wait to eat?

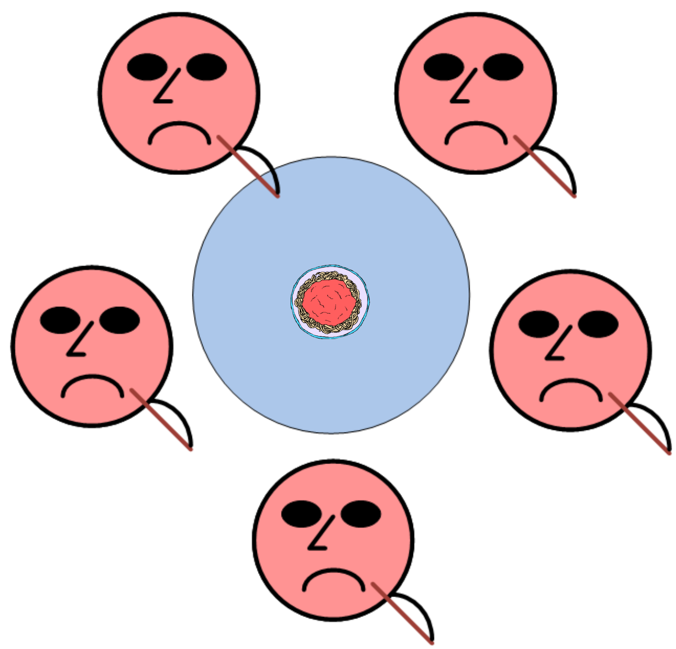
- This could lead to *deadlock*!!!



The Dining Philosopher's Problem

- What if all of the philosophers decide to eat at the same time *and* what if they are only allowed to wait for a fixed time?

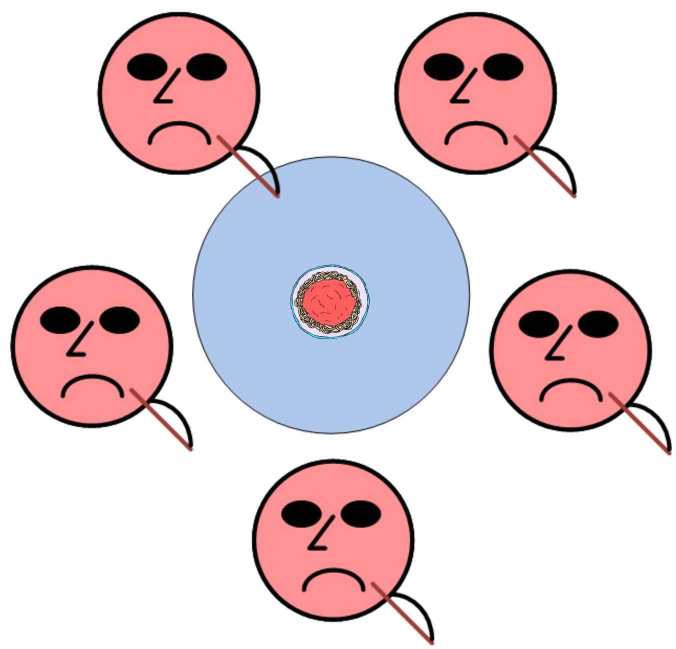
- This would lead to *livelock*!!!



Either way the poor philosophers suffer from ***starvation***!!

The Dining Philosopher's Problem

- How do we solve this problem?



Now let's think about this in terms of programming

What if we have these Cooperating Processes?

```
proc_0 ()
{
    while (TRUE)
    {
        <compute_section>;
                sum += ….;
        <critical_section>;
    }
}
```

```
proc_1()
{
    while(TRUE)
    {
        <compute_section>;
                sum += ….;
        <criticial_section>;
    }
}
```

```
<shared global declarations and initial processing>
fork(proc_0, 0);    //fork(process_call, args);
fork(proc_1, 0);
```

Assumptions about this code

- Writing and reading a memory cell common to the two processes is an indivisible operation
  - Any attempt by the two processes to simultaneously execute read or write operations will result in some unknown serial ordering of the two operation
  - The operations will not happen at the same time

- The processes are not assumed to have any priority,
  - Neither one or the other would take precedence in the case of simultaneous attempts to enter a critical section

Assumptions about this code

- The relative speeds of the processes are unknown
  - One cannot rely on speed differentials (or equivalence) in arriving at the solution (i.e. predict what will happen)

- These processes are assumed to be sequential and cyclic

We need protection for the critical regions of code

Acceptable solutions to the critical section problem are required to meet the following constraints

- Only one process at a time should be allowed to be in its critical section (***mutual exclusion***)
  - i.e. ***mutex***

- Given a critical section is free
  - If a set of processes indicates a need to enter into the critical section, then only those processes competing for the critical section participate in the selection of the process to enter the critical section

Acceptable solutions to the critical section problem are required to meet the following constraints

- Once a process attempts to enter its critical section, it cannot be postponed indefinitely
  - Even if no other process is in its critical section
- After a process requests entry into its critical section, only a bounded number of other processes may be allowed to enter their related critical sections before the original process enters its critical section

Now a little history…

Dijkstra’s Semaphore Primitives

Edsger Dijkstra invented the ***semaphore***

- A primitive to accomplish process synchronization [Dijkstra, 1968]
- Introduced the idea of “cooperating sequential processes”
- Illustrated why synchronization is difficult with conventional machine instructions

Dijkstra’s Semaphore Primitives

In the original paper:

- the “P” operation was short for the Dutch *proberen* “to test”
  - int sem_wait (sem_t *sem) in Linux
- the “V” operation was short for *verhogen* “to increment”
  - int sem_post (sem_t *sem) in Linux
- Let’s look at this in more detail

Using Semaphores

- A semaphore, s, is a non-negative integer variable (i.e a Whole Number) tested or changed by only one of two ***indivisible*** (***atomic***) routines:
  - P(s)/ sem_wait(s)
    - [while(s==0) {wait}; s= s-1;] ´
  - V(s)/sem_post(s)
    - [s = s+1;]

- The square braces surrounding the statements indicate that the operations are ***indivisible/atomic***

Using Semaphores

- Easy case:
  - V(s)/sem_post(s)
    - [s = s+1;]

- The operation [s = s+1;] cannot be interrupted until it has completed

Using Semaphores

- Harder case:
  - P(s)/sem_wait(s)
    - [while (s==0) {wait}; s = s-1;]
- If s >0:
  - s is tested and decremented as an indivisible operation
- If s =0:
  - the process executing the sem_wait() command can be interrupted when it executes the ***wait*** in the ***while*** loop
  - The indivisible operation only applies to the test and resulting control flow

How do we use a semaphore as a "mutex" to protect the critical section?

```
proc_0 () {
    while (TRUE) {
        <compute_section>;
        <critical_section>;
    }
}
```

```
proc_1() {
    while(TRUE) {
        <compute_section>;
        <criticial_section>;
    }
}
```

```
semaphore mutex =1;
fork(proc_0, 0); // Or now a days pthread_create(proc_0,…)
fork(proc_1, 0); //  Ditto
```

//Note sem_init's pshared value changes depending on whether it is to be shared between processos or threads

Using Mutexes

- Recall, a semaphore, s, is a non-negative integer variable tested or changed by only one of two ***indivisible*** (***atomic***) routines:
  - P(s)/ sem_wait(s)
    - [while(s==0) {wait}; s= s-1;]
  - V(s)/sem_post(s)
    - [s = s+1;]

- Conceptually, a mutex is a semaphore with a count of one, however, it may have different properties
  - e.g. Priority inheritance

How do we use “mutex” to protect the critical section?

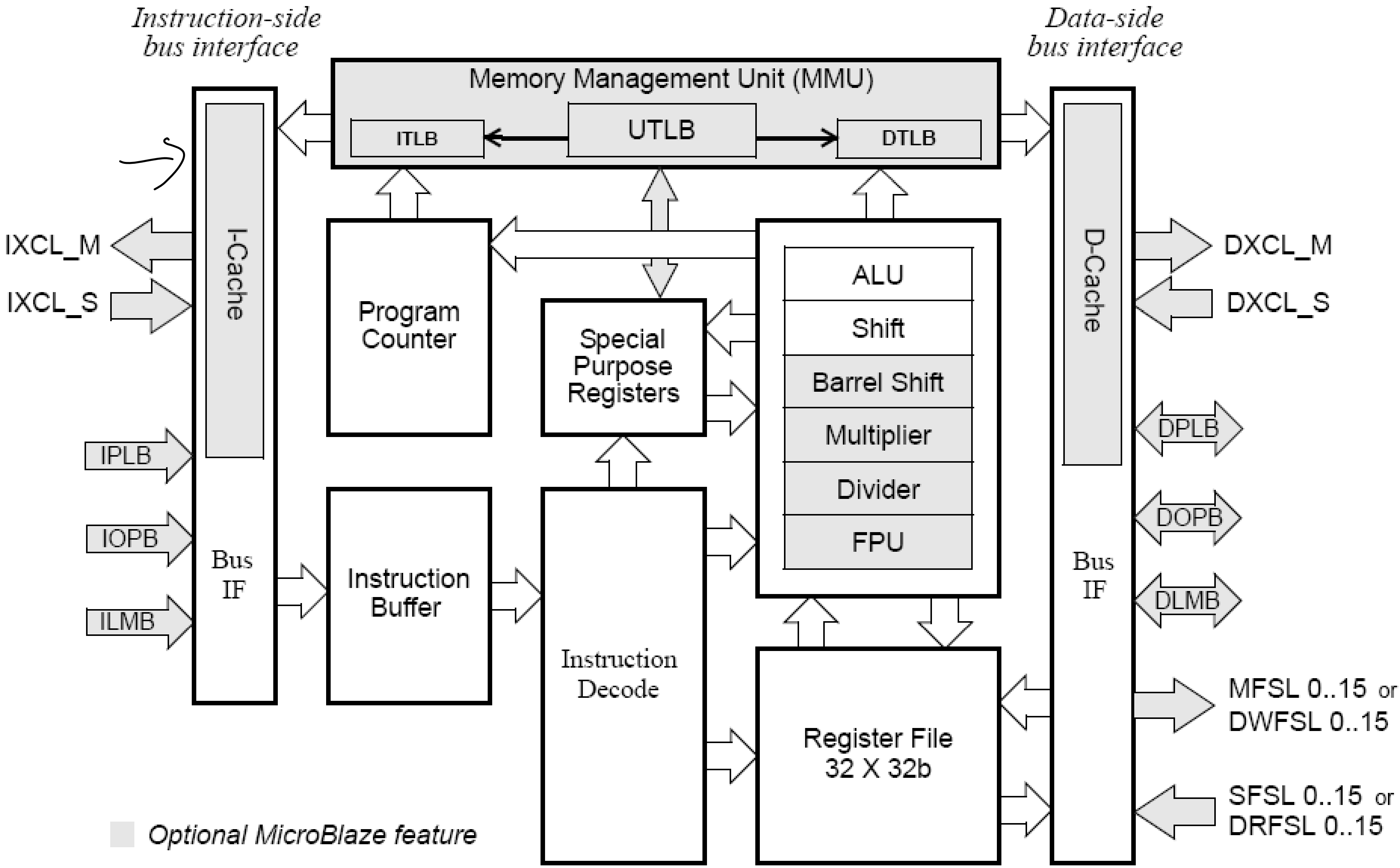
```
thread_0 () {
    while (TRUE) {
            <compute_section>;
            access(CS_resource);
    }
}

thread_1() {
    while(TRUE) {
            <compute_section>;
    access(CS_resource);
    }
}
//Initialization:
ResourceType *CS_resource; //Critical Section resource
mutex mutex =1; //In Linux pthread_mutex_init(pthread_mutex_t *mutex)
Create_thread(thread_0, 0);
Create_thread(thread_1, 0);

//Check out pthread_mutex_lock; pthread_mutex_unlock
```

Xilinx's MicroBlaze

Xilinx's MicroBlaze

- Harvard Architecture
  - Separate Instruction and Datapath
  - Used in:
    - DSP processor architectures
      - E.g. Blackfin from Analog Devices
    - Microcontrollers
      - E.g. PIC from Microchip Technology
- Resource Usage (Assuming lightweight)
  - ~1000 LUTs
  - ~800 Flipflops
- Max Frequency: ~250 MHz
- Xilinx also has PicoBlaze (look it up)

MicroBlaze Processor



MicroBlaze and Mutual Exclusion

- Has No “atomic” instructions
  - Uses “Load Word Exclusive” and “Store Word Exclusive”
  - Check out the “MicroBlaze Processor Reference Guide” available online for free:

http://www.xilinx.com/support/documentation/sw_manuals/xilinx13_2/mb_ref_guide.pdf

Synchronizing IPC

Classic IPC and Synchronization Problems

- The Producer-Consumer Problem
- The Readers-Writers Problem
- The Sleeping Barber Problem

Producer-Consumer Problem [Dijkstra, 1968]

- Picture a system with a Producer process and a Consumer process with N buffer resources
  - Bounds the memory resources
  - Keeps the processes synchronized

- The two processes communicate by:
  - Having the producer
    - Obtain an empty buffer from a pool of empty buffers,
    - Fill the buffer with information
    - Place the full buffer in a pool of full buffers

  - Having the consumer
    - Obtain a full buffer from the pool of full buffers
    - Copy the information out of the buffer
    - Place them empty buffer back in the empty buffer pool

Producer Code

Consumer Code

Readers-Writers Problem [Courtois, et al, 1971]

- Suppose a resource is to be shared among a community of processes of two distinct types:
  - Reader
    - A reader process can share the resource with any other reader process but not with a writer process
  - Writer
    - A writer process requires exclusive access to the resource whenever it acquires access to any resource
- Similar to sharing a file among processes
  - Anyone can read the file, but when writing to the file, only one (writing) process has accesss

Writer Code

Reader Code

The Sleeping Barber Problem [Dijkstra, 1968?]

- Based upon a “barber shop” with:
  - one barber,
  - one barber chair, and a
  - number of chairs for waiting customers.
- When:
  - There are no customers,
    - The barber takes a nap in his chair.
  - A customer arrives,
    - If all chairs are occupied, the new customer leaves
    - Else If the barber is busy cutting hair, the customer sits down
    - Else this is the first customer, so the barber wakes up

The Sleeping Barber Problem [Dijkstra, 1968?]

- Readers-Writers or Producer-Consumer problem?

- Queueing Theory

- The Rendezvous Problem

- Possible Problems

The Barber

The Customers

Questions?

- Our discussion about semaphores has been in terms of separate processes. What about threads? Could semaphores still be used?

- What state is a thread in when it is waiting for access to a critical section (ie waiting on a semaphore)?

Questions?

- What does the term “atomic operation” mean?

- Semaphores provide controlled access to critical sections, but not necessarily mutual exclusion. Can a semaphore be used to provide mutual exclusion (ie act like a mutex)?

Questions?

- Does Linux provide any mutex functions (not just semaphores)?

- Other than mutual exclusion (as opposed to just controlled access), are mutexes and semaphores the same?

Questions?

- Dijkstra posed potential software solutions to the critical section problem and then explained why they failed [Dijkstra, 1968]. One of these examples will be on your midterm and/or final.