Operating Systems W5L2 - Race Conditions (ctd) and Deadlocks

Class	Operating Systems
Created	@Sep 30, 2020 12:39 PM
Materials	04 - Deadlocks.pdf
Reviewed	☐
Type	Lecture

We started off by reviewing sleep/wakeup and the producer/consumer problem from last class and went over that code

Operating Systems W5L1 - Race Conditions (ctd)

Solving the Lost Wakeup Problem

▼ What is the Lost Wakeup Problem?

When `consumer()` stops after reading `count == 0`, such as an interrupt occurring, then there's no chance to call `sleep()`. The next lecture will pick up on how we tackle this idea and (hopefully) find solutions for it.

- One solution is to add a *wakeup waiting bit*
  - An extra bit is set upon wakeup, and if the process attempts to sleep later but the bit is set, the "state" is kept awake and the bit is reset
  - Problem is... we may have more than two processes → So how many bits are we meant to use...?
- The best solution for this is **semaphores**

- Stores integer to count number of wakeups "saved for future use" — This integer is called a **semaphore**
    - This "future use" pertains to wakeup calls being saved for (and thus consumed by) processes in the future
- There are two primitives- up and down- key here at play. They are atomic operations.
    - **Up:** Increments stored integer, waking up a sleeping process (if there is one)
    - **Down**: If the integer is 0, puts process to sleep. Otherwise, it decrements and continues
- ▼ Code using semaphores

```
#define N 100                          /* number of slots in the buffer */
typedef int semaphore;                 /* semaphores are a special kind of int */
semaphore mutex = 1;                   /* controls access to critical region */
semaphore empty = N;                   /* counts empty buffer slots */
semaphore full = 0;                    /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                    /* TRUE is the constant 1 */
          item = produce_item( );      /* generate something to put in buffer */
          down(&empty);                /* decrement empty count */
          down(&mutex);                /* enter critical region */
          insert_item(item);           /* put new item in buffer */
          up(&mutex);                  /* leave critical region */
          up(&full);                   /* increment count of full slots */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                    /* infinite loop */
          down(&full);                 /* decrement full count */
          down(&mutex);                /* enter critical region */
          item = remove_item( );       /* take item from buffer */
          up(&mutex);                  /* leave critical region */
          up(&empty);                  /* increment count of empty slots */
          consume_item(item);          /* do something with the item */
     }
}
```

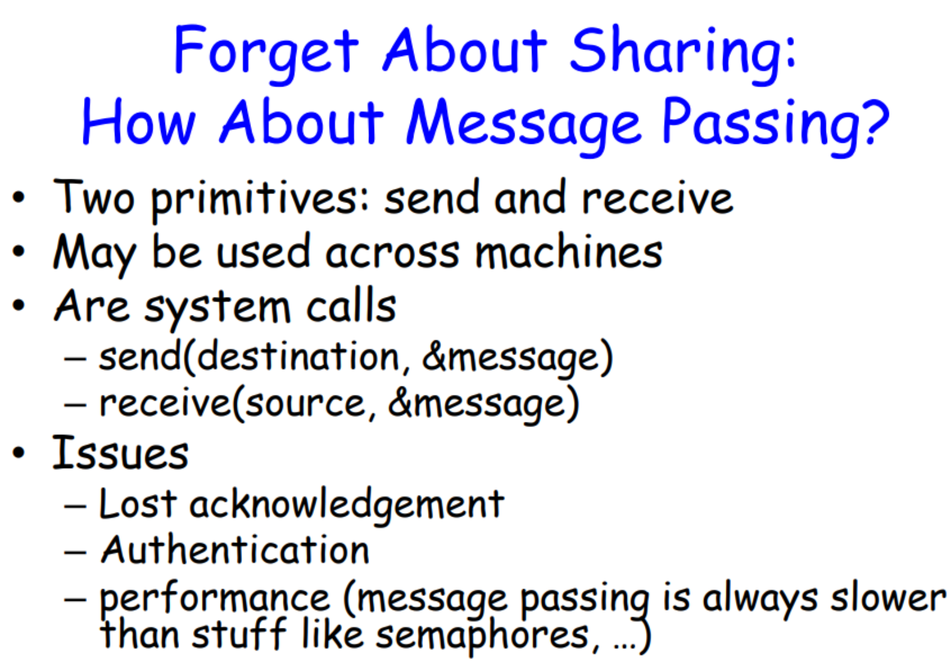
Mutexes

- A **mutex** is a variable that can either be locked or unlocked; in these two possible states

- It's a nickname for locks, in a way, which are used to manage critical sections
- Managed using `TSL` (see previous slides) and `XCHG` registers
- While we did say processes don't share address space, there are two ways they technically can
    1. Some of the shared data structures are stored at kernel level → Accessed via system calls
    2. Most modern operating systmes offer ways for processes to "share" portions of their address space with other processes

▼ Message passing

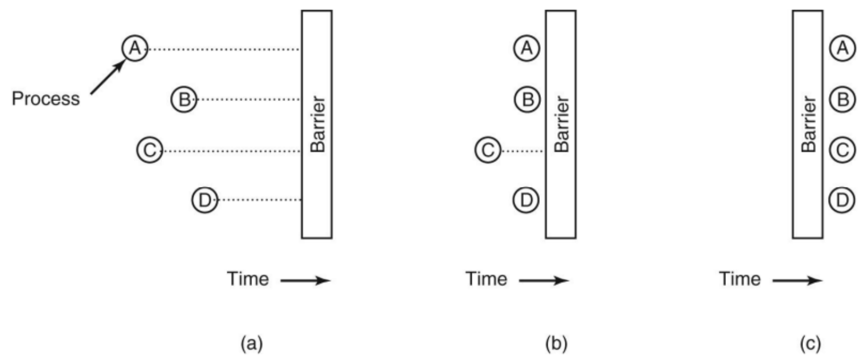
This concept won't be thoroughly discussed in the course, but worth mentioning, according to Zahran.



Deadlocks

- **Barriers** are synchronization mechanisms that are intended for a gorup of porcesses

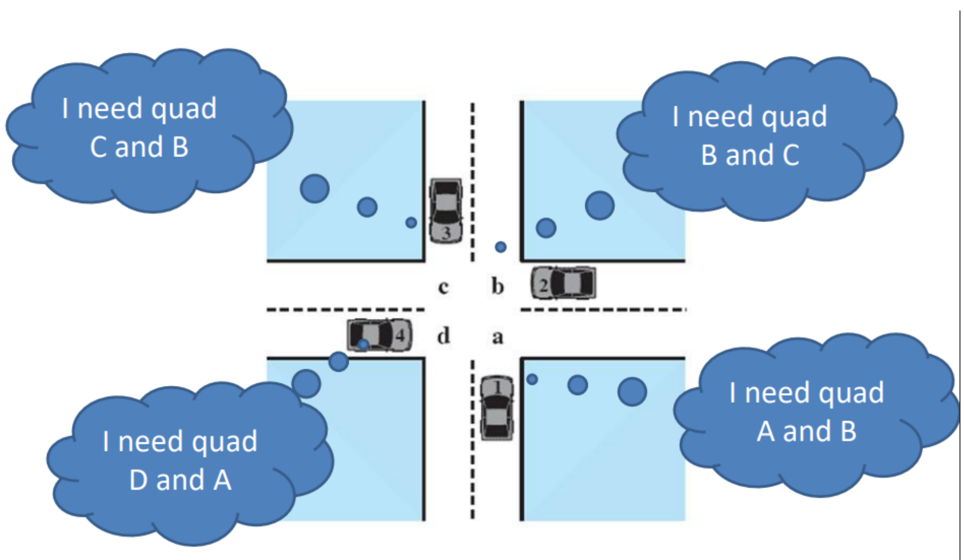
▼ Barriers diagram

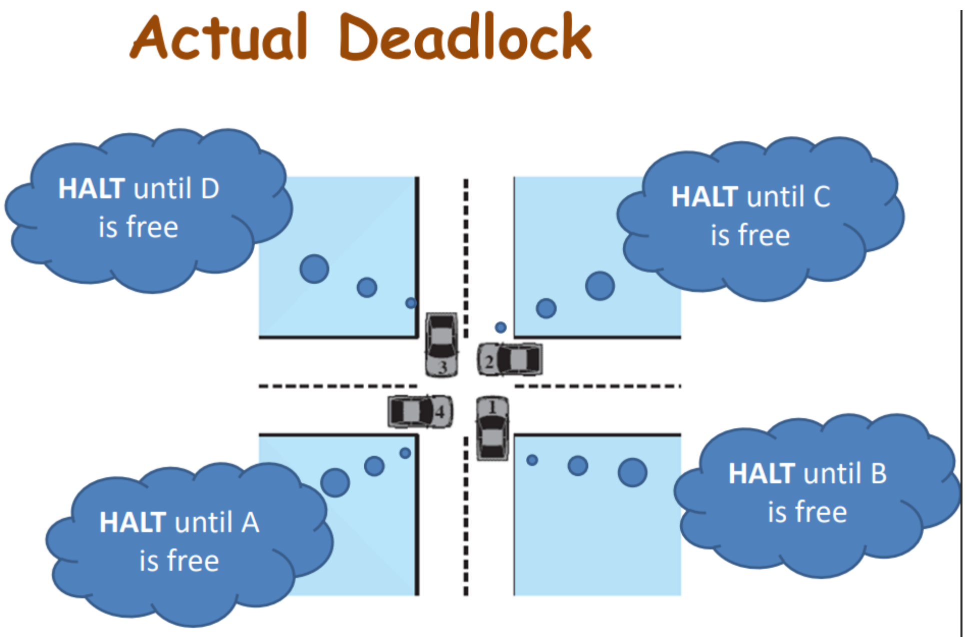


- We want to do everything in our power to prevent *deadlocks*

▼ What is a deadlock? What is a good analogy for it?

A deadlock is when multiple processes need to use multiple resources, but can not reach them because other processes are blocking them off. A good analogy is gridlock (as in traffic).





- Deadlocks occur among *processes* who need to acquire *resources* in order to *progress*.

Resources

- A resource is anything *acquired, used,* then *released*.
- Preemptable vs. non-preemptable resources
  - **Preemptable:** Can be taken away from the process with no ill-effect
  - **Non-preemptable:** Cannot be taken away from the process without causing the computation to fail. In other words (since negation can be confusing), *if you take the resource away from the process, the computation (and thus the process) will fail.*
- Resource categories
  - Reusable
    - Can be safely used by only one process and is *not* depleted by that use
    - This includes processors, I/O channels, main and secondary memory, devices, and data structures (such as files, databases, and semaphores)
  - Consumable

- Can be created (produced) and destroyed (consumed)
- This includes interrupts, signals, messages, and information. Things within I/O buffers

> ❗ We discussed lab1 at the end of class, approximately the last 10 minutes of the lecture recording.