Operating Systems W5L1 - Race Conditions (ctd)

Class	Operating Systems
Created	@Sep 28, 2020 12:49 PM
Materials	
Reviewed	☐
Type	Lecture

> ! Always differentiate coherence and performance.

Avoiding Race Conditions

▼ What is **mutual exclusion**?

Prohibiting one process from sharing data at the same time as another process

- Achieveing mutual exclusion is a major OS design issue, particular the *appropriate primitive operations*.
  - "Appropriate" here is referencing the ability of an operationg to be both correct and efficient
- **Critical Region:** The part of a process where the shared memory is being accessed

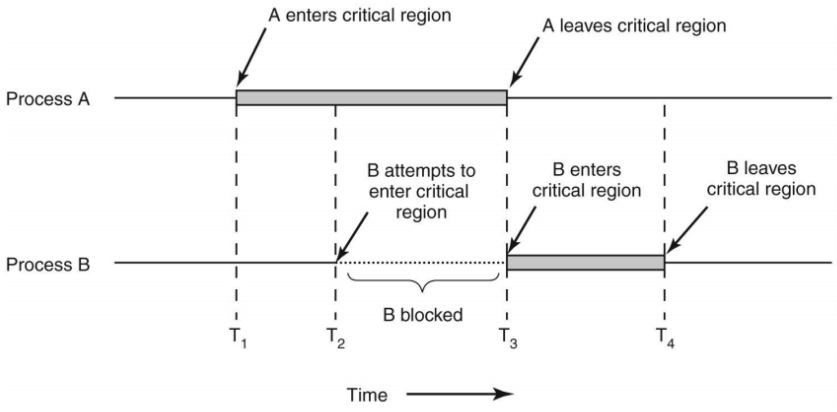
Conditions of Good Solutions

- Lots of research being done on this, and many conditions need to be considered

> ! These concepts are reviewed from the perspective of just two processes to start as a base that we will branch off of later in the course.

1. No two processes simultaneously inside their critical region (this is the **race condition**)
    - Not simultaneously *executing*, but simultaneously *inside* the memory location
2. No assumptions may be made about speeds or number of CPUs
3. No process running outside its critical region may block the use of the same critical region by other processes
4. No process should wait forever to enter its critical region

▼ Critical Region Diagram



- The operating system is the one that schedules and coordinates these things, *not* the programmers

Primitive Solutions

1. Disabling Interrupts

    Have each process disable all interrupts just after entering the critical region and re-enable them just before leaving it.

    ▼ Downfalls

- Unwise to give user processes the power to turn off interrupts
- Affects only one CPU and not other CPUs in the system in case of multicore or multiprocessor systems

2. Lock Variables

   Have a shared (lock) variable that is initially set to 0. When a process wants to enter its critical region, it tests the lock. This lock variable is available to all processes.

   ▼ Downfalls

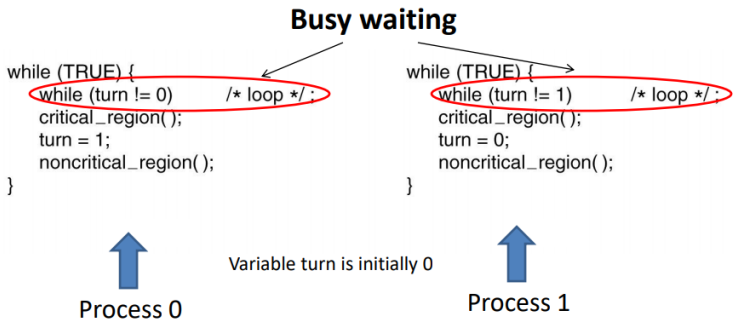
   - Process A reads the lock and finds it 0
   - Before it can set it to 1, process A is stopped and process B starts
   - Process B finds the lock to be 0, so it sets it to 1 and enters the critical region
   - Process B is stopped and process A runs
   - Process A sets the lock to 1 and enters the critical region

   This implies that two processes can be in the same critical region.

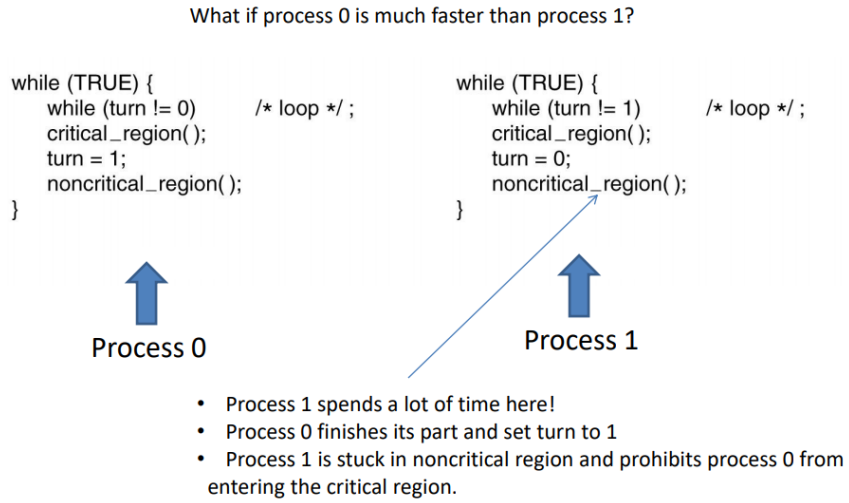
3. Strict Alternation

   Waiting is not a good thing, and there is a lot of waiting going on here — **basic waiting**. This also doesn't violate solutions, and it is correct, but it violates performance efficiency.

   ▼ Diagram



▼ Downfalls



Processes can get sutck, violating condition three and blocking another process.

The above solutions were the "basic" or "primitive" solutions published at the time. Then along came Señor Peterson (Señor added for no particular reason, I have no idea where he's from)

4. Peterson's Solution

▼ Overview Diagram

The entire idea is the ability for programs to enter and exit regions, specifically a critical region. The `enter_region` and `exit_region` functions are system calls. It is functional since it;s executed by the OS / on a kernel level, however it's still a lot of waiting and thus not entirely *efficient*.

process 0	process 1
enter_region(0)	enter_region(1)
Critical Section	Critical Section
leave_region(0);	leave_region(1);

▼ Code Implementation

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

Hardware Solution

- The TSL solution → `TSL(RX, LOCK)` — An atomic operation
- TSL → **T**est and **S**et **L**ock
- Reads contents of memory address `lock` and puts it into register `RX`. Then a non-zero number is put into memory address `lock`.

▼ Assembly Code Implementation(s)

```
enter_region:
        TSL REGISTER,LOCK          | copy lock to register and set lock to 1
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was nonzero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

```
enter_region:
        MOVE REGISTER,#1           | put a 1 in the register
        XCHG REGISTER,LOCK         | swap the contents of the register and lock variable
        CMP REGISTER,#0            | was lock zero?
        JNE enter_region           | if it was non zero, lock was set, so loop
        RET                        | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0               | store a 0 in lock
        RET                        | return to caller
```

Summary on Previous Solutions

▼ What is the common drawback on the "primitive" solutions?

Primarily, the drawback is that they all involve a lot of busy waiting. This wastes time that the CPU could be working as well as the priority inversion problem.

- Mutual exclusion fails if a process "cheats", such as calling `enter_region` and `leave_region` in the cincorrect timing. How is this solved? See below 😉

Sleep and Wakeup

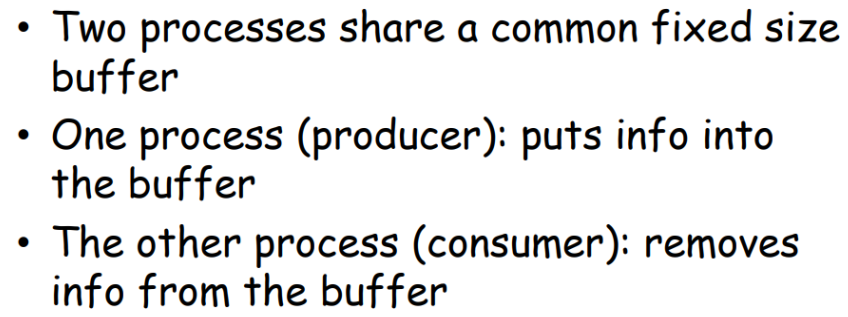
- **Sleep:** Causes the caller to block until another process wakes it up.
- **Wake:** Has one parameter, and that parameter is the process to be woken up.
- Both are IPC primitives — In other words, `sleep` and `wake` functions are system calls
- Implements block (sleep) and thus froming a process from CPU to then in turn not waste CPU time, freeing space up for another process

- This is still implementing *Peterson's Solution*, just in a different way (that "way" being these two syscalls).

Producer Consumer Problem

Specific Example: When editing a video, you need to play video back to see if what you've done looks good (unless you're incredibly daring). When you press play, the software loads the video by adding it to the buffer- producing it. When the video itself plays, it removes the data from the buffer- consuming it. Shoutout to Premiere Pro gang.

▼ Slide Bullets



▼ C Pseudocode Implementation

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item( );               /* generate next item */
          if (count == N) sleep( );             /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep( );             /* if buffer is empty, got to sleep */
          item = remove_item( );                /* take item out of buffer */
          count = count – 1;                    /* decrement count of items in buffer */
          if (count == N – 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

- `count` is protected, and this is given by the fact that we are dealing with multiple processes
- When `count` is incremented in the producer code, it means that if `count == 1` then it was 0 before. If it was 0 before, it informs us that *there was nothing in the buffer.*
- **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.