A Write Operation in the Kernel

```
queued_len += user_len;
if (queued_len == user_len)
        startio();
```

If there's no I/O in progress start it up.

But an Interrupt Happens in the Middle

```
queued_len += user_len;
                                    queued_len -= io_block_len;
if (queued_len == user_len)
        startio();
```

The **`if`** statement will fail to start the I/O

What's the Problem?

- With interrupts, code execution isn't linear
- Two different flows of execution accessed the same variable at about the same time
- We need a way to prevent that
- As always, multiprocessors makes this much worse

Race Condition

- Undesirable behavior that can occur from inappropriate reliance on ordering of operations

When Can Race Conditions Occur?

- Between threads
- Between processes
- Between main level and interrupt level
- Between multiple processors
- Between the CPU and I/O devices
- Any time two flows of control can access the same storage

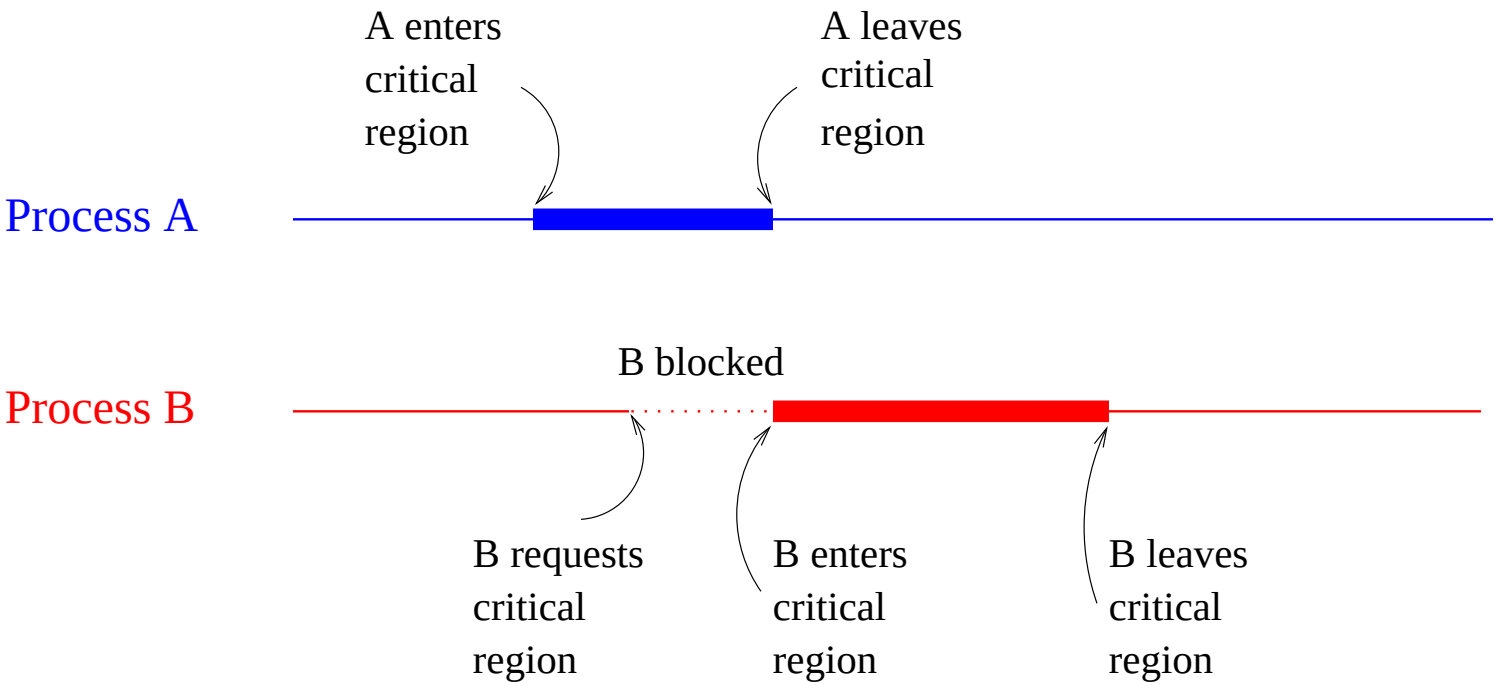
Race Conditions are Bad

- System hangs
- System crashes
- Lost data
- Security problems
- *Unpredictability*

Critical Regions

- The part of a program that uses a shared variable is a *critical region*
- Two (or more) programs can't be in their critical regions at the same time
- One program (or process or thread or device) has to block if it needs access to the critical region while the other is in its critical region

Critical Regions



Desired Properties

1. No two processes can be simultaneously in their critical regions (safety)
2. No assumptions can be made about CPU speeds, number of processors, etc. (generality)
3. Processes not requesting entry to a critical region must not block because another process is using it (efficiency)
4. All processes must eventually get a chance to enter the critical region (fairness)

Entering a Critical Region

- We have to have mechanisms that mediate access to critical regions
- Control flows must — somehow — signal when they're entering and leaving a critical region
- Access to shared data must not occur outside the critical region

Disabling Interrupts

- Available only to the kernel
- Better not be available to user processes!
- Doesn't work well on multiprocessors
- A special-case solution for the kernel only

Lock Variables

- How about this?

```
while (lock != 0)
    ;
/* Start critical region */
lock = 1;
...
/* End critical region */
lock = 0;
```

- Doesn't work — there's a window between testing for zero and setting it to 1

Let's Use a C Feature: ++

- ```
  while(lock++ != 0)
          lock--;
  /* critical region ...*/
  ```
- Is **`lock++`** *atomic*?
- No — the language makes no guarantees about that!
- IBM mainframe sequence:

```
L     R0,lock          Fetch variable into Register 0
LR    R1,R0
A     R1,=F'1'         Increment it
ST    R1,lock          Store new value in 'lock'
C     R0,=F'0'         Compare original value to 0
BZ    CRITREG          If so, go on...
```

- No atomic "increment" instruction!
```

Test and Set Lock

- We need an *atomic* test/set instruction
- Some — but not all — architectures have one
- The instruction reads the old value and writes non-zero into the location
- The memory bus is locked during this instruction — even a second CPU can't intervene

Using “Test and Set Lock”

```
LOCKTST TSL    R0,lock  Copy lock to R0 and set it non-zero
        CMP    R0,#0    Was it zero?
        BNZ    LOCKTST  Branch back if non-zero
```

Instead of a special instruction, the Pentium has a LOCK prefix that can be applied to several instructions.

Strict Alternation

Process 0

```
while (TRUE) {
    while (turn != 0)
        ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

Process 1

```
while (TRUE) {
    while (turn != 1)
        ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

Disadvantages of Strict Alternation

- Suppose that Process 1's non-critical region code takes a long time to execute
- It won't re-enter its critical region
- Until it does, Process 0 can't re-enter its own critical region
- This violates our efficiency principle

Peterson's Algorithm (1981)

- Simple algorithm using only ordinary (i.e., non-locking) instructions
- Described in a two-page paper that includes a (simple) proof
- See the reading list

Peterson's Algorithm

```
int turn, interested[2];
void enter_region(int process)
{
    int other;
    other = 1-process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process && interested[other] == TRUE)
            ;
}
void leave_region(int process)
{
    interested[process] = FALSE;
}
```

Spin Locks

- All three correct solutions — Test and Set Lock, Strict Alternation, Peterson's Algorithm — involve *busy waiting*
- Also known as a *spin lock*
- Acceptable for short waits, especially on multiprocessors or when dealing with interrupt contexts
- For general-purpose use, need a solution that permits sleeping

Spin Locks and Priority Inversion

- Suppose we have two processes with different priorities, H and L
- L never runs if H is runnable
- While H is sleeping, the low priority process L grabs the spin lock
- H wakes up and tries to get the lock
- It spins, waitng for L to free it
- But L can’t get the CPU, so it doesn’t progress
- One example of a *deadlock*

Classic Problem: Produce-Consumer

- Producer wants to add elements to bounded-size buffer
- If buffer is full, producer must sleep
- Consumer wants to remove items from buffer
- If buffer is empty, consumer must sleep

Producer-Consumer

```
#define N 100
int count = 0;
```

Producer

```
while (TRUE) {
    item = produce_item();
    if (count == N) sleep();
    insert_item();
    if (count++ == 0)
        wake(consumer);
}
```

Consumer

```
while (TRUE) {
    if (count == 0) sleep();
    item = remove_item();
    if (count-- == N)
        wake(producer);
    consume_item(item);
}
```

Obvious race conditions

Semaphores

- Invented by Dijkstra in 1965
- Two operations: **down** (sometimes known as **P**) and **up** (also known as **V**)
- **down**: decrements semaphore variable if greater than zero; if 0, process sleeps before doing the decrement
- Note: check, decrement, and sleep are *atomic*
- **up**: increments semaphore; if a process was sleeping on it, wake it and let it do its decrement
- Generalizes well to multiple users of the semaphore

Using Semaphores

```
semaphore mex = 1;

down(&mex);
/* critical region */
up(&mex);
```

Implementing Semaphores

- Typically done in the kernel
- Mask interrupts while manipulating semaphore
- On multiprocessors, use Test and Set Lock protection as well
- Both of these are for only a few microseconds — not a serious problem

Producer-Consumer with Semaphores

```
#define N 100
int count = 0;
semaphore mutex = 1, empty = N, full = 0;
```

Producer

```
while (TRUE) {
    item = produce_item();
    down(&empty);
    down(&mutex);
    insert_item();
    up(&mutex);
    up(&full);
}
```

Consumer

```
while (TRUE) {
    down(&full);
    down(&mutex);
    item = remove_item();
    up(&mutex);
    up(&empty);
    consume_item(item);
}
```

Two Different Uses of Semaphores

- Counting semaphores — atomic way to manipulate a counter with blocking if 0
- In the example, **`empty`** counts how many slots are free and **`full`** is the number of slots that are full
- Used for sleep/wake when buffer is in the wrong state
- Mutual exclusion semaphores — make sure only process is in critical region
- In this case, it protects access to the buffer
- (Probably, that code should be in the **`insert_item()`**/ **`remove_item()`** routines)

Mutexes

- Special case of semaphore: useful when no counting is needed
- If Test and Set Lock instruction is available, easy to implement at user level for thread package
- Try to grab lock; if unsuccessful, let another thread run
- But — if the thread holding the lock is blocked, this thread will burn CPU time until the OS intervenes

The Risks of Semaphores and Mutexes

- Using semaphores and mutexes correctly is difficult
- If you get it wrong, you can deadlock
- Testing is difficult, because it's all timing-dependent
- We need a higher-level construct

Monitors

- Invented by Hoare (1974) and Brinch Hansen (1975)
- A programming language construct
- Java has it; C and C++ do not
- A **`monitor`** is like a **`class`**, but only one thread can be executing in it at a time
- In other words, they're a language implementation of mutexes

Monitors and Application Blocking

- We still need a way to block if, say, the buffer is full
- Two new operations: **`wait`** and **`signal`**
- Subtle semantics about who gets to run within the monitor after a **`signal`** call
  - Hoare: **`signal`**er blocks; **`wait`**er runs
  - Brinch Hansen: **`signal`** can only be done when exiting the monitor
  - Or: **`signal`**er keeps running; **`wait`**er resumes when **`signal`**er exits monitor

The Disadvantages of Monitors

- Monitors are nice, but they're only available in a very few languages
- Monitors, semaphores, and mutexes are find on a single machine, even a multiprocessor
- They don't work well without some shared memory
- What about a distributed system connected via a LAN?

Message Passing

- Model based on network operations:
  `send(dest, &mesg)`;
  `recv(src, &mesg)`;
- Receiver blocks if there's no data available
- Sender blocks if the receiver's buffers are full
- Complex network questions, including acknowledgment, flow control, error detection and correction, message ordering, and authentication — all out of scope for this course

Using Message Passing

- Many different paradigms of how to use it
- Rendezvous — one message at a time; sender and receiver operate in lock-step
- Mailboxes — fixed-size buffers on channel; sender blocks if they're full
- Explicit requests — empty messages from the consumer to the producer

Summary

- IPC can be hard
- Many possible schemes, depending on environment and underlying OS
- Danger of deadlock — topic for next class. . .