Outline

- Concurrency issues
- Cooperating processes
- Race condition
- Critical section – software solutions
- Hardware based solutions
- Semaphore
- Monitors

Concurrency

- Our motivation
  - overlap computation with I/O
  - maximize the use of system resources
- Approaches
  - **hardware parallelism**: CPU computing, one or more I/O devices are running at the same time
  - **pseudo parallelism**: rapid switching back and forth of the CPU among processes, pretending that those processes run concurrently
  - **real parallelism**: can only be achieved by multiple CPUs

Barriers to Concurrency

- Few languages have direct support for concurrency (Java, Ada, but not C/C++)
- Managing concurrent processes and the resources they require is complicated
  - OS's usually provide only minimal support
  - Many different ways to provide support and none is dominant

Concurrent Processes

- Concurrent processes can:
  - Compete for shared resources
  - Cooperate with each other in sharing the global resources
- OS deals with competing processes
  - carefully allocating resources
  - properly isolating processes from each other
- OS deals with cooperating processes
  - providing mechanisms to share resources

Processes: *Competing*

- Processes that do not work together cannot affect the execution of each other, but they can compete for devices and other resources
- **Example:** Independent processes running on a computer
- **Properties:** Deterministic; reproducible
  - Can stop and restart without side effects
  - Can proceed at arbitrary rate

Processes: *Cooperating*

- Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other
- **Example**: Transaction processes in an airline reservation system
- **Properties**:
  - Share a common object or exchange messages
  - Non-deterministic; May be irreproducible
  - Subject to *race conditions* – coming up!

Why Cooperation?

- Cooperation clearly presents challenges – why do we want it?
- We may want to share resources:
  - Sharing a global task queue
- We may want to do things faster:
  - Read next block while processing current one; divide a job into pieces and execute concurrently
- We may want to solve problems modularly

UNIX example:

```
cat infile | tr ' ' '\012' | tr '[A-Z]' '[a-z]' | sort | uniq -c
```

Problem with C

- Hard to understand or v
executing programs
- Instructions of the prog
arbitrarily
  - For cooperating processes, the order of (some) instructions are irrelevant
  - However, certain instruction combinations must be avoided, for example:

Process A	Process B	concurrent access
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

<u>Original application..</u>

A = 1
B = 2
A = B + 1
B = B * 2

Race Conditions

- When two or more processes are reading or writing some shared data and final result depends on who runs precisely when is a *race condition*
- How to avoid race conditions?
  - Prohibit more than one process from reading and writing shared data at the same time
  - Essentially we need *mutual exclusion*
- Mutual exclusion:
  - When one process is reading or writing a shared data, other processes should be prevented from doing the same
  - Providing mutual exclusion in OS is a major design issue

An Example

A & B are updating a
drawing and B depositing.

Correct balance = $100
($50 withdrawal and $50
deposit)

Initial balance = $100

Process A

R1 <- $100

R2 <- $50

R1 <- $100 - $50

balance <- R1 = $50

store R1, balance

Process B

R3 <- $100

R4 <- $50

R3 <- $100 + $50

balance <- R3 = $150

store R3, balance

Possible values for balance are $50, $150, and
$100!

An Example....

- Suppose we have the following code for the account transactions

```
authenticate user
open account
load R1, balance
load R2, amount (-ve for withdrawal)
add R1, R2
store R1, balance
close account
display account info
```

An Example....

- Suppose we have the following code for the account transactions

Critical section

authenticate user
open account
load R1, balance
load R2, amount (-ve for withdrawal)
add R1, R2
store R1, balance
close account
display account info

Critical Section

- Part of the program that accesses shared data
- If we arrange the program runs such that no two ~~programs~~ are in their critical sections at the same time, race conditions can be avoided

processes

Critical Section

Requirements of a Critical Section

- For multiple programs to cooperate correctly and efficiently:
  - No two processes may be **simultaneously in** their critical sections
  - No assumptions be made about **speeds** or number of CPUs
  - No process running **outside** its critical section may block other processes
  - No process should have to **wait forever** to enter its critical section

Critical Sections

- Critical region execution should be *atomic* i.e., its runs “all or none”
- Should not be interrupted in the middle
- For efficiency, we need to minimize the length of the critical sections
- Most inefficient scenario
  - Whole program is a critical section – no multiprogramming!

Road to a Solution

- Simplest solution:
  - Disable all interrupts
  - This should work in a single processor situation
  - Because interrupts cause out-of-order execution due to interrupt servicing
  - With interrupts disabled, a process will run until it yields the resource voluntarily
- Not practical:
  - OS won't allow a user process to disable all interrupts – OS operation will be hindered too!
  - Does not work on multiprocessors

Road to a Solution

- Idea: To come up with a software solution
- Use “lock” variables to prevent two processes entering the critical section at the same time – all along we are talking about a single critical section
- Use variable `lock`
- If `lock == 0` set `lock =1` and enter_region
- If `lock == 1` wait until lock becomes `0`
- Does not work, Why??

Strict Alternation

- Two processes take turns in entering the critical section
- Global variable turn set either to 0 or 1

Process 0

```
while (TRUE) {
    while (turn !=0);
    critical_section();
    turn = 1;
    non_critical();
}
```

Process 1

```
while (TRUE) {
    while (turn !=1);
    critical_section();
    turn = 0;
    non_critical();
}
```

Strict Alternation

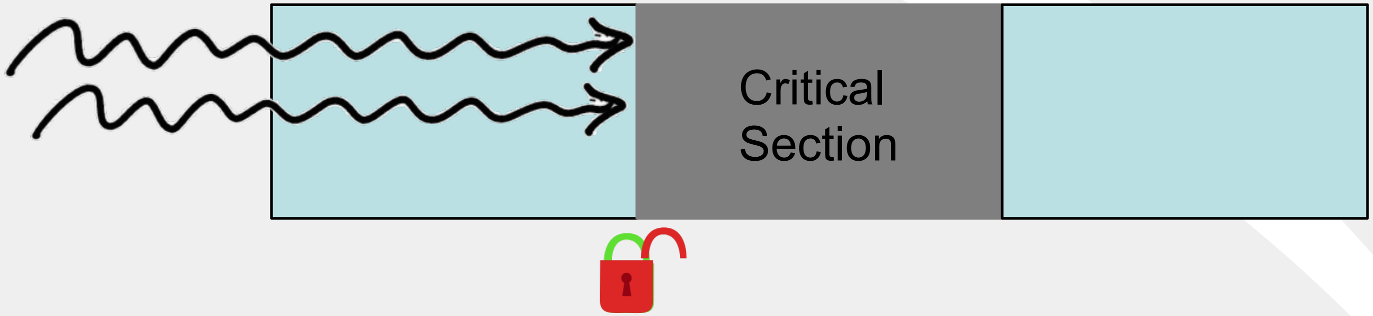
- What is the problem with strict alternation?
- We can have starvation, why??
- What are the other drawbacks?
  - Continuously testing a variable until some value appears is called **busy waiting**
  - Busy waiting wastes CPU time – should be used when the expected wait is short
  - A lock that uses busy waiting is called **spin lock**

Example

Kernel level multi-threads are used for concurrent processing in a data-parallel application. The application has a critical section. If a single thread of execution is used the critical section takes 2 seconds and other parts take 6 seconds. We have a large dataset so we use 2 threads. What is the run time? We have an even larger dataset so we use 8 threads. What is the run time?

Assume single CPU and ideal time sharing (no overhead). All threads starting at the same time.

Locks – An Illutration



Mutual Exclusion: First Attempt

- The simplest mutual exclusion strategy is taking turns as considered earlier

```
/* process 0 */
.
.
while (turn != 0);
/* critical section */
turn = 1;
.
.
```

```
/* process 1 */
.
.
while (turn != 1);
/* critical section */
turn = 0;
.
.
```

Mutual Exclusion: Second Attempt

- First attempt problem – single key shared by the two processes
- Each have their own key to the critical section
- Solution does not work! Why??

```
/* process 0 */
.
.    .
while (flag[1]);
flag[0] = true;
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
while (flag[0]);
flag[1] = true;
/* critical section */
flag[1] = false;
.
.
```

Mutual Exclusion: Third Attempt

- This works, i.e., provides mutual exclusion
- Has **deadlock** – why?

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1]);
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0]);
/* critical section */
flag[1] = false;
.
.
```

Mutual Exclusion: Fourth Attempt

- This works
- Has **livelock** – why?

```
/* process 0 */
.
.
flag[0] = true;
while (flag[1])
{
        flag[0] = false;
        /* random delay */
        flag[0] = true;
}
/* critical section */
flag[0] = false;
.
.
```

```
/* process 1 */
.
.
flag[1] = true;
while (flag[0])
{
        flag[1] = false;
        /* random delay */
        flag[1] = true;
}
/* critical section */
flag[1] = false;
.
.
```

Dekker's Algorithm

```
/* process 0 */
…
flag[0] = true;
while (flag[1])
    {
        if (turn == 1)
        {
            flag[0] = false;
            while (turn == 1);
            flag[0] = true;
        }
    }
    / * critical section */
    turn = 1;
    flag[0] = false;
    …

  …
```

```
/* process 1 */
…
flag[1] = true;
while (flag[0])
    {
        if (turn == 0)
        {
            flag[1] = false;
            while (turn == 0);
            flag[1] = true;
        }
    }
    / * critical section */
    turn = 0;
    flag[1] = false;
    …

  …
```

Peterson's Algorithm

- Dekker's algorithm is complicated
  - hard to prove the correctness
- Peterson's algorithm is much simpler
- Based on the same idea of using the *turn* variable to arbitrate and an array of flags to express interest to enter the critical section

Peterson's Algorithm

```
/* process 0 */
…
flag[0] = true;
turn = 1;
while (flag[1] &&
          turn == 1);
/ * critical section */
flag[0] = false;
/* remainder */
…
```

```
/* process 1 */
…
flag[1] = true;
turn = 0;
while (flag[0] &&
          turn == 0);
/* critical section */
flag[1] = false;
/* remainder */
…
```

Can Hardware Help?

- Dekker's and Peterson's algorithms are pure software solutions
- Can hardware provide any help?
  - make the solution more efficient
  - make it scalable to more processes?
- Yes!
- Current microprocessors have hardware instructions supporting mutual exclusion

Test and Lock

- `TSL RX, LOCK` is a typical CPU instruction providing support for mutual exclusion
  - read the contents of memory location `LOCK` into register `RX` and stores a non-zero value at memory location `LOCK`
  - operation of reading and writing are indivisible - atomic

```
enter_section:
    TSL REGISTER, LOCK
     // copy lock to reg and set lock to 1
    CMP REGISTER, #0
    // was lock zero??
    JNE enter_section
    // if non zero, lock was set
    RET
```

```
leave_region:
    MOVE LOCK, #0
    RET
```

Properties of Machine Instruction Approach

- Advantages:
  - applicable to any number of processes
  - can be used with single processor or multiple processors that **share a single memory**
  - simple and easy to verify
  - can be used to support multiple critical sections, i.e., define a separate variable for each critical section

Properties of Machine Instruction Approach

- Disadvantages:
  - **Busy waiting is employed** – process waiting to get into a critical section consumes CPU time
  - **Starvation is possible** – selection of entering process is arbitrary when multiple processes are contending to enter

Busy Waiting Approaches

- Mutual exclusion schemes discussed so far are based on busy waiting
- Busy waiting not desirable:
- Suppose a computer runs two processes H: high priority and L: low priority
  - scheduler always runs H when it is in ready state
  - at a certain time L is in its critical section and H become runnable
  - H begins busy waiting to enter the critical section
  - L is never scheduled to leave the critical section
  - there is a **deadlock**
  - this situation is sometimes referred to as the **priority inversion problem**

Sleep/Wakeup Approach

- Alternative to busy waiting that is inefficient and deadlock prone is to use a sleep/wakeup approach
- Implemented by the Semaphores

Semaphores

- Fundamental principle:
  - two or more processes can cooperate by sending simple messages
  - a process can be forced to stop at a specific place until it receives a specific message
  - complex coordination can be satisfied by appropriately structuring these messages
  - for messaging a special variable called semaphore ***s*** is used
  - to transmit a message via a semaphore a process executes signal(s)
  - to receive a message via a semaphore a process executes wait(s)

Semaphores

- Operations defined on a semaphore:
  - can be initialized to a nonnegative value – set semaphore
  - wait – decrements the semaphore value – if value becomes negative, process executing wait is blocked
  - signal – increments the semaphore value – if value is not positive, a process blocked by a wait operation is unblocked

Semaphores

- A definition of semaphore primitives…

```
struct semaphore {
        int count;
        queueType queue;
}
```

```
void wait(semaphore s)
{
    s.count--;
    if (s.count < 0)
    {
        place this process in
        s.queue;
        block this  process
    }
}
```

```
void signal(semaphore s)
{
    s.count++;
    if (s.count <= 0)
    {
        remove a process P
        from s.queue;
        place process P on
        ready list;
    }
}
```

Semaphores

- Wait and Signal primitives are assumed to be atomic
  - they cannot be interrupted and treated as an indivisible step
- A queue is used to hold the processes waiting on a semaphore
- How are the processes removed from this queue?
  - FIFO: process blocked longest should be released next – ***strong semaphore***
  - Order not specified – ***weak semaphore***

Type of sema

These are initial values. A semaphore can have negative value
at any time .. many threads or processes waiting on it

- Semaphores are usually
  ways:
  - **binary**—is a semaphore wi
    of 0 or 1
    - Or a value of FALSE o RUE if you prefer
    - Initialized to 1 for mutex applications
  - **counting**—is a semaphore with an integer value ranging between 0 and an arbitrarily large number - initial value might represent the number of units of the critical resources that are available - also known as a **general** semaphore

Producer-Consumer: Semaphores

```
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;


producer() {
    int item;

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

```
// protects the critical section
// counts the empty slots
// counts full buffer slots


consumer() {
    int item;

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

Producer-Consumer: Semaphores

- One binary semaphore **mutex** to ensure only one process is manipulating the buffers at a time
- **Empty** and **Full** counting semaphores, to count the number of empty or full buffer slots respectively
  - Empty initialized to N, so waiting on empty means waiting if there are no empty slots (*buffer full!*)
  - Full initialized to 0, so waiting on full means waiting if there are no full slots (*buffer empty!*)

Primitives Revisited

- The synchronization primitives discussed so far are all of the form:

```
entry protocol
   < access data >
exit protocol
```

- Semaphores give us some abstraction: implementation of the protocol to access shared data is transparent to the user
- More abstraction would be of additional benefit (as it often is!)

Monitors

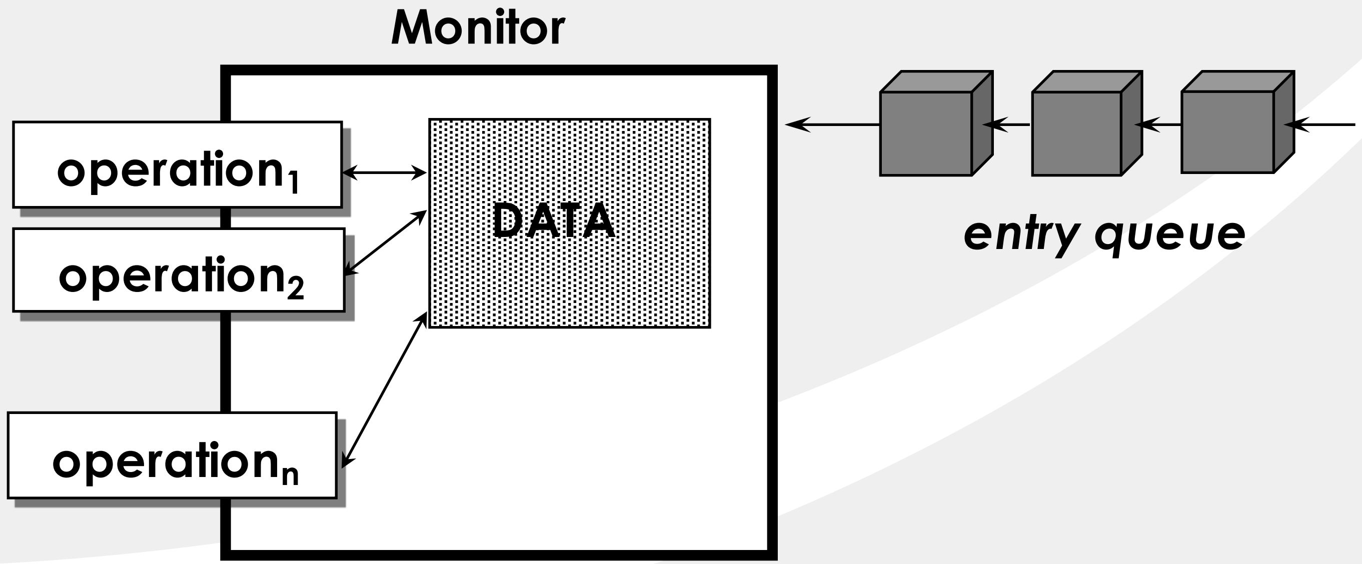
- A monitor is a high-level (programming language) abstraction that combines (and hides) the following:
  - shared data
  - operations on the data
  - synchronization using condition variables

Monitors

- Mutual exclusion --- not the only thing we need for concurrency
- Abstraction provided by monitors allows us to deal with many different issues in concurrency
  - block processes when a resource is busy
- With a monitor data type, we define a set of variables that define the state of an instance of the type, and a set of procedures that define the externally available operations on the type

Monitors

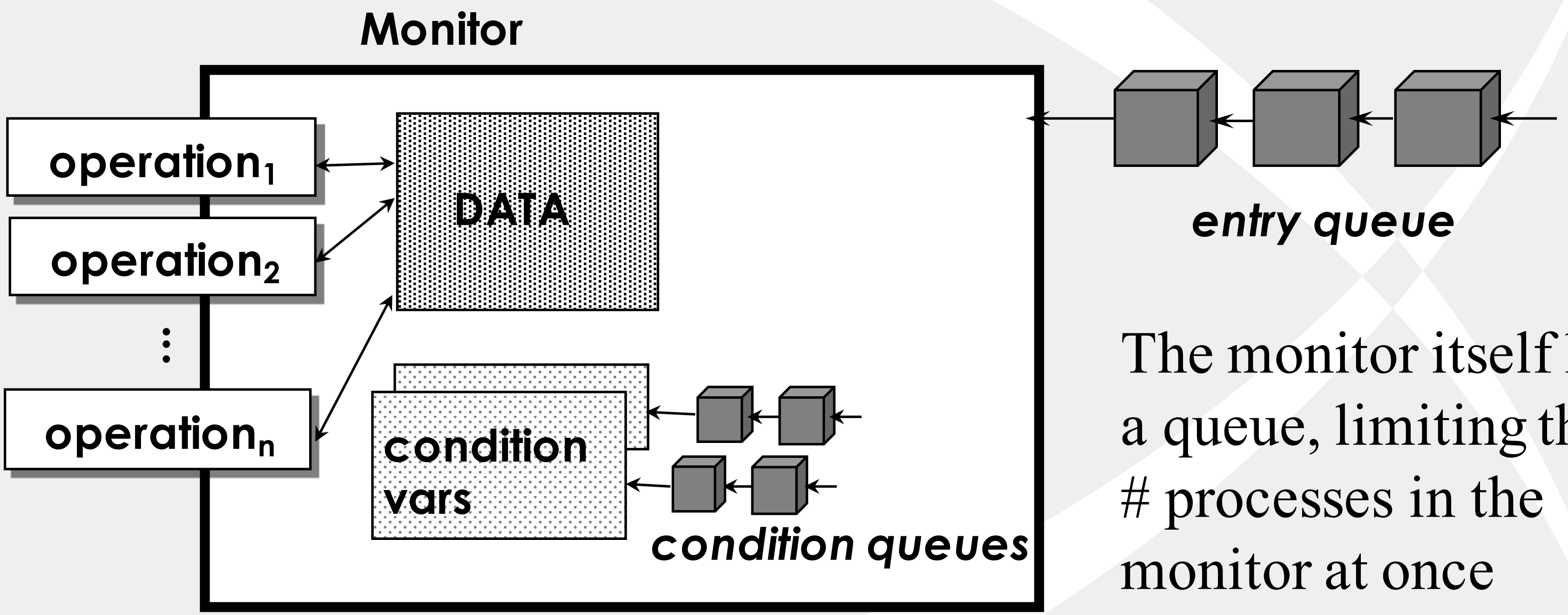
- Monitors are more than just objects
- A monitor ensures that only one process at a time can be active within the monitor – so you don't need to code this explicitly



Monitors

- Just this on its own though isn't enough – need the additional synchronization mechanisms
- Monitors use **condition variables** to provide user-tailored synchronization and manage each with a separate queue for each condition
- The only operations available on these variables are **WAIT** and **SIGNAL**
  - **wait** suspends process that executes it until someone else does a **signal** (different from semaphore wait!)
  - **signal** resumes one suspended process. no effect if nobody's waiting (different from semaphore free/signal)!

Monitor Abstraction



The monitor itself has a queue, limiting the # processes in the monitor at once

How Monitors Work

- Remember only one process can be active in the monitor at once. Some will be waiting to get in, others will be waiting on conditions in the monitor
- If P executes x.signal (signal for condition x), and there's a process Q suspended on x, both can't be active at once. 2 possibilities:
  - P waits on some condition till Q leaves monitor
  - Q waits on some condition till P leaves monitor
- Also compromises: if P executes signal, it leaves monitor automatically, resuming Q

Problems with *synch* Primitives

- *Starvation*: the situation in which some processes are making progress toward completion but some others are locked out of the resource(s)
- *Deadlock*: the situation in which two or more processes are locked out of the resource(s) that are held by each other

Problems with *synch* Primitives

- The most important deficiency of the primitives discussed so far is that they were all designed on the concept of one or more CPUs accessing a ‘‘common’’ memory
  - Hence, these primitives are not applicable to distributed systems. **Solution**? Message passing...

Synch Primitives—Summary