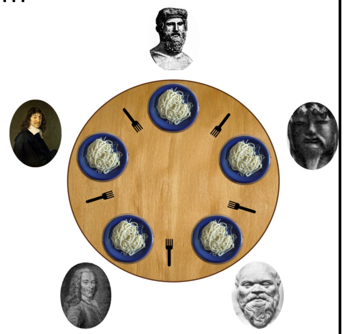
Operating Systems

Sateesh K. Peddoju

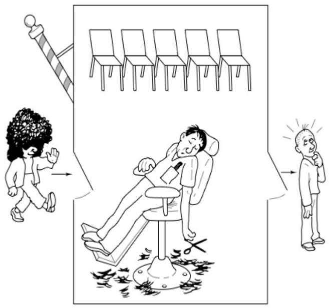
Process Synchronization

- Dining Philosopher's Problem
  - 5 Philosophers
  - 5 chopsticks (forks)
  - Random times eat
  - Must pick
    - One right & One left
  - Issues
    - Deadlock or Starvation
  - Solutions?
    - Only 4 to be hungry
    - Allow – if both sticks are available
    - –



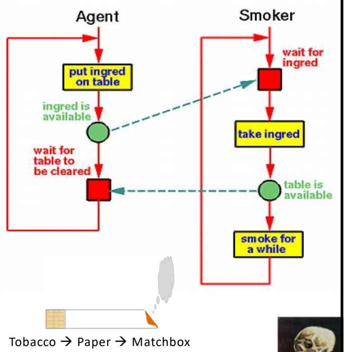
Process Synchronization

- Barber Shop Problem
  - sleeping Barber Problem
  - Single Barber
  - A barber's chair
  - n – chairs for waiting
  - Barber spends his life in shop
  - Sleeps until a customer wakes up.
  - Customer waits, if busy
  - No chairs, leaves & comes back
- Solution
  - Mutual Exclusion



Process Synchronization

- Cigarette Smokers Problem



Cigarette Smoking is injurious to health

Process synchronization

- Producer-Consumer Problem
  - (or Bounded-Buffer Problem)
- Readers-Writers Problem

CLASSICAL PROBLEMS OF SYNCHRONIZATION

PRODUCER – CONSUMER PROBLEM

(BOUNDED-BUFFER)

~~WITHOUT SEMAPHORES~~

Bounded-Buffer ~~without semaphores~~

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
  . . .                    /* int data;   */
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```


Bounded-Buffer ~~without semaphores~~

- **Producer process**

```
item nextProduced;

while (1) {
      while (counter == BUFFER_SIZE)
               ; /* do nothing */
      buffer[in] = nextProduced;
      in = (in + 1) % BUFFER_SIZE;
      counter++;
}
```

- **Consumer process**

```
item nextConsumed;

while (1) {
      while (counter == 0)
               ; /* do nothing */
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;
      counter--;
}
```

Bounded-Buffer ~~without semaphores~~

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

9

9

Bounded-Buffer ~~without semaphores~~

- The statement "**count++**" may be implemented in machine language as:

  register1 = counter
  register1 = register1 + 1
  counter = register1

- The statement "**count--**" may be implemented as:

  register2 = counter
  register2 = register2 – 1
  counter = register2

10

10

Bounded-Buffer ~~without semaphores~~

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

11

11

Bounded-Buffer ~~without semaphores~~

- Assume **counter** is initially 5. One interleaving of statements is:

  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

12

12

Concurrency

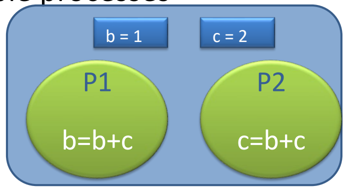
- Benefits
  - Communication among the processes
  - Sharing Resources
  - **Synchronization of multiple processes**
  - Allocation of processors time
- Difficulties
  - Sharing global resources (ex: global variable)
    - Race Condition
  - Management of allocation of resources (optimality?)
  - Programming errors difficult to locate
  - **Inconsistent data**

13

13

Concurrency – Race Condition

- A race condition occurs when multiple processes or threads read and write data items so that the **final result depends on the order of execution of instructions** in the multiple processes



14

14

Concurrency – OS Design Concerns

- Multiprogramming
  - Management of multiple processes in Uni-processor
- Multiprocessing
  - Management of multiple processes in Multi-processors
- Distributed Processing
  - Management of multiple processes executing on multiple, distributed machines

15

15

OS Design Concerns

- Keep track of active processes
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources
- *Result of process must be independent of the speed of execution of other concurrent processes*

16

16

Contexts of Concurrency

- Non-concurrent vs. Concurrent code
- Structured Applications
  - Concurrency is part of programming
  - User Threads
- Operating System Structure
  - Concurrency is part of OS
  - Kernel Threads
- Requirement: Identify concurrent sections of the code.

17

17

Critical Section

- A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.

18

18

Critical Section Problem

- Design a protocol that the processes can use to cooperate.
- Each process must request to enter critical section
- Entry Section
- Exit Section
- Remainder Section

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

19

19

Solution to Critical Section Problem

Mutual Exclusion

1. Only one process at a time is allowed in the critical section for a resource

Progress

2. A process that is in its reminder section must do so without interfering with other processes
3. A process must not be delayed access to a critical section when there is no other process using it
4. It must not be possible for a process to be delayed indefinitely
5. A process remains inside its critical section for a finite time only
6. We assume that each process is executing at non-zero speed. However, we can make no assumption concerning the relative speed of the processes.

Bounded Waiting

7. If another process is waiting to enter the critical section, there exists a bound or limit on the number of times current process to enter its critical section

20

20

Mutual Exclusion

- The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

```
/* PROCESS 1 */
void P1
{
  while (true) {
    /* preceding code /;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */
void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

. . .

```
/* PROCESS n */
void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

21

21

Meeting the Requirements…

- Software Approach:
  - Leave the responsibility with the processes to handle the situation (neither OS nor PL will support)
  - Prone to high processing overhead and bugs
- Hardware Approach:
  - Using special purpose machine instructions (at ISA)
  - Reduces overhead but not the general-purpose solution
- OS and PL Approach:
  - Provide support within OS and PL itself.

22

22

Software Approaches

- Peterson's Solution
  - Restricted to two processes only (P0, P1) (Pi, Pj).
  - int *turn*;

/* whose turn it is to enter critical section */

  - boolean *flag[2]*;

/* which process is ready to enter critical section */

  - Mutual exclusion?
  - Progress ?
  - Bounded Wait ?

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = FALSE;

        remainder section

} while (TRUE);
```

23

23

Software Approaches

- A general Solution - Lock

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (TRUE);
```

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

24

24

Software Approaches

- Disadvantage
  - Prone to high processing overhead and bugs

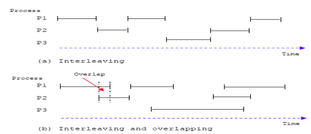
25

25

Hardware Approaches

- Interrupt Disabling
  – uniprocessor
  – Disabling interrupts guarantees mutual exclusion

  – Disadvantages:
    – Processor is limited in its ability to interleave programs
    – Multiprocessing
      - disabling interrupts on one processor will not guarantee mutual exclusion



26

Hardware Approaches

- Special Machine Instructions
  – Performed in a single instruction cycle
  – Not subject to interference from other instructions
  – TestAndSet() and Swap()   (Reading)

27

27

Hardware Approaches

- Advantages

  – Applicable to any number of processes on either a single processor or multiple processors sharing main memory

  – It is simple and therefore easy to verify

  – It can be used to support multiple critical sections

28

28

Hardware Approaches

- Disadvantages
  - Busy-waiting
    - consumes processor time
  - Starvation
    - more than one process is waiting → some process may not be selected → waiting indefinitely.
  - Deadlock
    - If a low priority process (say P1) executing in the critical region and a higher priority process (P2) needs, the higher priority process will obtain the processor.
    - If P2 tries to access a resource which P1 holds, it will be denied due to mutual exclusion.
    - P1 to wait for the critical region, P2 to wait for resource

29

29

OS and PL Approaches

- **Semaphores**
  - Binary Semaphores
    - Mutex
  - Condition Variable
- **Monitor**
  - Event Flags
- **Mailboxes/Messages**
- Spinlocks

30

30

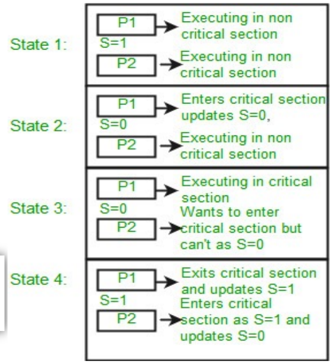
OS and PL Approaches

- Semaphore

```
do {
   waiting(mutex);

      // critical section

   signal(mutex);

      // remainder section
}while (TRUE);
```

```
wait(S) {
    while S <= 0
       ; // no-op
    S--;
}
```

```
signal(S) {
    S++;
}
```



31

31

Semaphore Implementation

- When a process must wait on a semaphore, it is added to the list of processes
- A Signal() operation removes one process from the list of waiting processes and awakens that process.

```
typedef struct {
      int value;
      struct process *list;
} semaphore;
```

32

32

OS and PL Approaches

- Semaphore – Disadvantages
  - Busy waiting

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block();
        }
}

signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
                remove a process P from S->list;
                wakeup(P);
        }
}
```

Solution

OS should have system calls:
block()
wakeup()

May be negative also

If negative → *s* = no. of processes waiting for *s*

```
signal(S) {
    S++;
}
```

```
wait(S) {
    while S <= 0
      ; // no-op
    S--;
}
```

33