Operating Systems

Lecture 5b

Dr Ronald Grau School of Engineering and Informatics Spring term 2018

Previously

Process Synchronisation

- Inter-Process Communication
- Race conditions
- Communication models
- Critical section
- Software vs. Hardware solutions
- Condition synchronisation

Today

Process Synchronisation

- Critical Sections
- Common application problems:
  - Producer-Consumer
  - Readers-Writers (Lab next week)
- Synchronisation primitives:
  - Mutex, Semaphore, Monitor, Condition Variable, . . .
  - Transactional Memory
  - Message Passing

Recap: Synchronisation

Questions

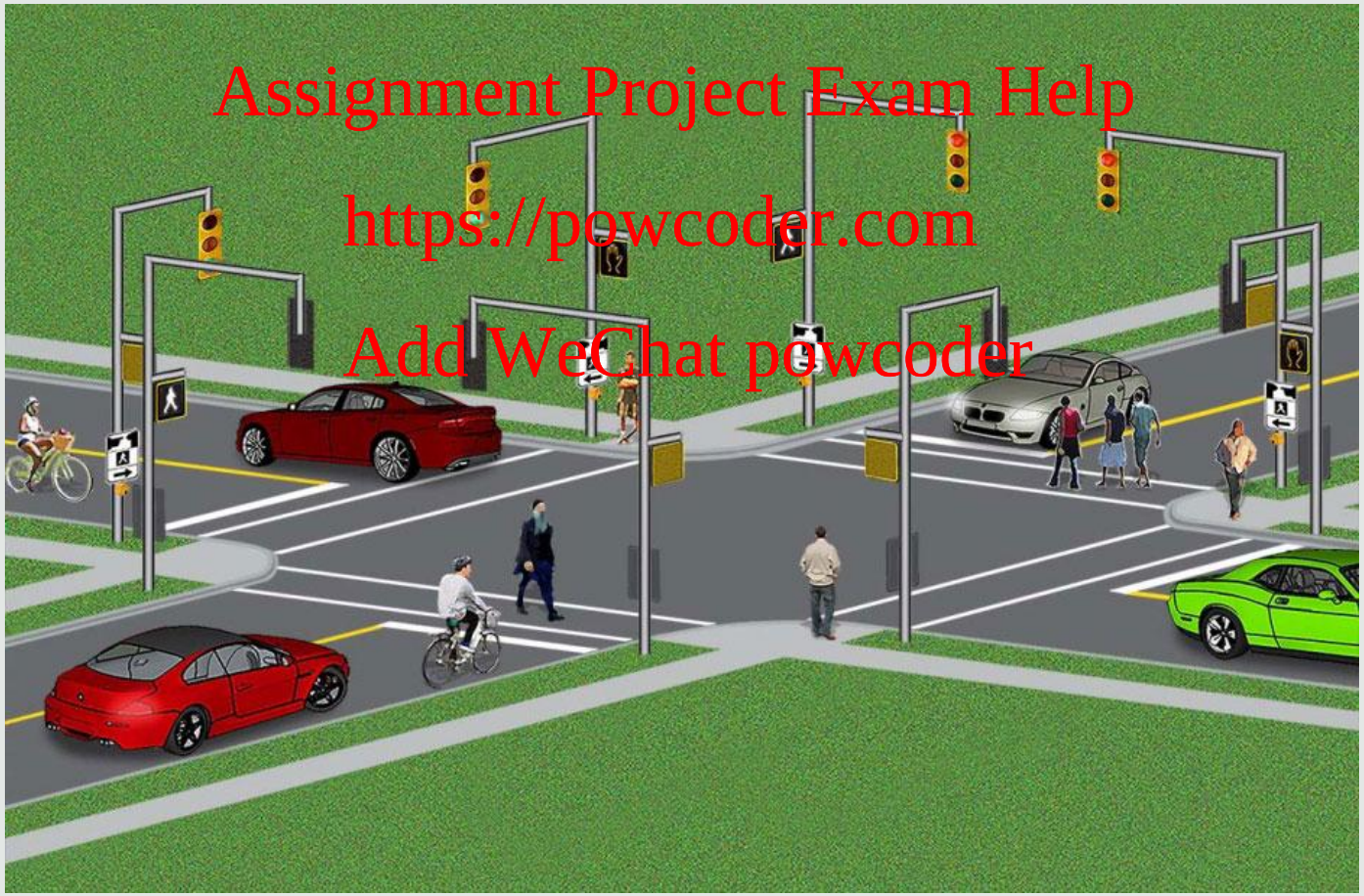
- What are mechanisms with which processes can communicate with each other?
- Which problem can occur when processes communicate via shared memory?
- What is mutual exclusion?
- What is busy waiting?
- What is condition synchronisation?
- What is an example of a hardware-supported synchronisation primitive?

Recap: Critical Section

- Mutual exclusion: If a process is executing in its critical section, then other processes must not execute in their critical section.
- Progress (Prevent deadlock): If no process is executing in its critical section and some processes wish to enter their critical sections, then the selection of which process may enter cannot be postponed indefinitely.
- Bounded waiting (Prevent starvation): If a process wishes to enter the critical section then only a bounded number of processes may enter the critical section before it.

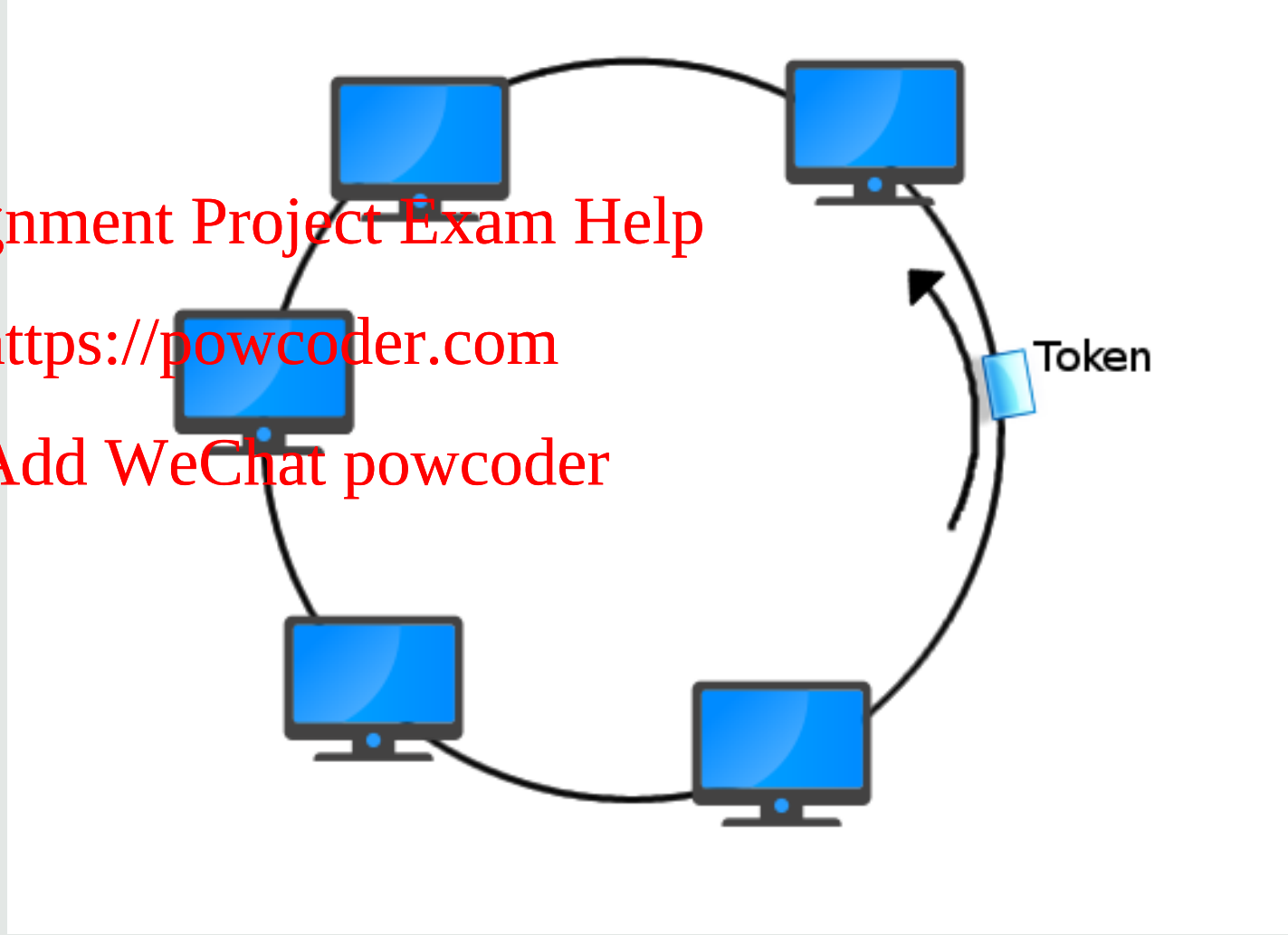
Recap: Critical Section

Where is the critical section?



Recap: Critical Section

Token-controlled synchronisation



Recap: Critical Section solutions

Software

- Assume atomicity of read and write operations

Hardware-supported

- More powerful instructions (e.g. test-and-set)

Higher-level APIs

- Wrap low-level primitives into library:
  Semaphores, monitors, condition variables, etc
- Thread-safe data structures:
  Blocking queues, synchronised maps, etc

Bakery algorithm (Leslie Lamport 1974)

How does it work?

- Customers draw a ticket as they arrive
- The ticket has a number that indicates when it is the customer's turn to be served
- When translated to operating systems, the processes are the customers

Bakery algorithm (Leslie Lamport 1974)

- We have n processes
- Shared variables:
  `boolean[n] choosing;`
  `int[n] number;`
- There is a chance that tickets may have the same number
- if number == 0 for any process, then that process does not request entering the critical section (already in it)
- Instead of keeping busy waiting, the current thread can also yield the CPU to the next thread (Cooperative multithreading)

```
do {
choosing[i] = true;
number[i]=1+max(number[0],...,number[n-1]);
choosing[i]=false;
for(int j=0;j<n;j++) {
    while(choosing[j]) {
          /* do nothing */
    }
    while(number[j]!=0 &&
          (number[j]<number[i] ||
           number[j]==number[i] && j<i)) {
                /* do nothing */
    }
}
//critical section
number[i]=0;
//...
}
while(!done);
```

Synchronisation Primitives

Synchronisation algorithms tend to be tricky to implement

- Do not re-implement them in a user program.
- Use the synchronisation primitives provided by the OS.
- Usually, better performance: blocking instead of busy waiting.

Mutex

- Data structure: boolean m
- Operations:
  - acquire(m): process blocks until m is false, then sets to true
  - release(m): sets m to false

- E.g. PThreads:
  - `int pthread mutex lock(pthread mutex t *mutex);`
  - `int pthread mutex unlock(pthread mutex t *mutex);`
  - `int pthread mutex trylock(pthread mutex t *mutex);`
- Common use of terminology (in books, papers):
  - acquire, lock, wait, P
  - release, unlock, signal, V

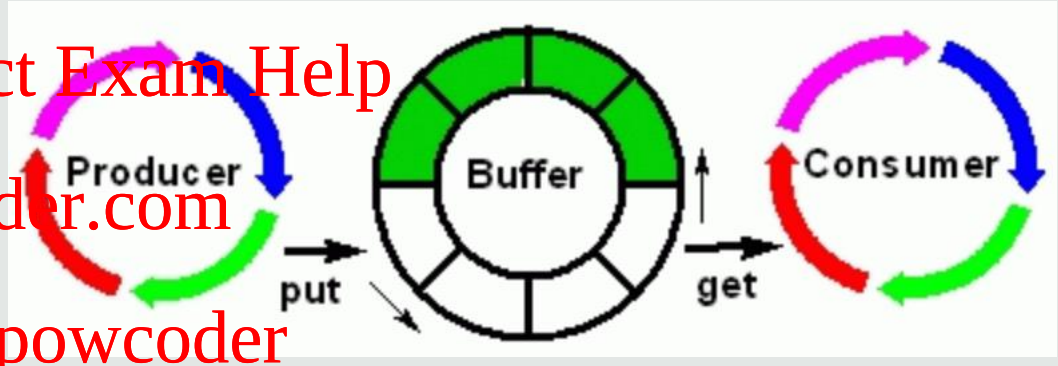
Semaphore (Edsger Dijkstra 1963)

- Generalisation of a mutex: allows n processes to access a resource
- Data structure: int c (initialised to n), queue q
- Operations:
  - acquire(): process blocks if c=0 and is added to q, otherwise decrement c
  - release(): remove a process from q and unblock it if q is non-empty, otherwise increment c
- Application example: license server for software applications
- Java:
  - `Semaphore(int permits, boolean fair)`
  - `void acquire(int permits)`
  - `void release(int permits)`

Producer-Consumer Problem

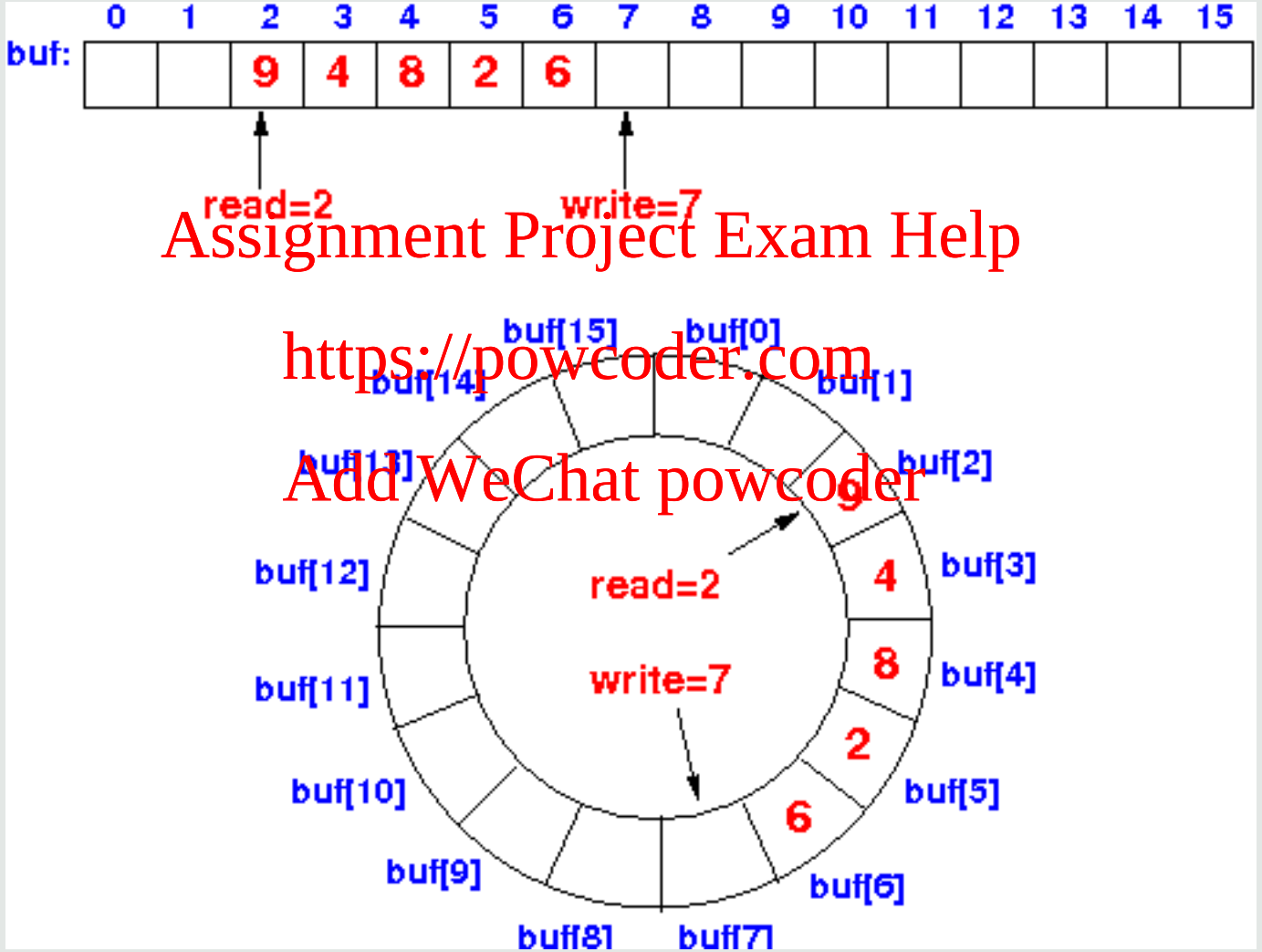
Synchronisation algorithms tend to be tricky to implement

- Producer: generates data
- Consumer: reads data



- Bounded buer (ringbuffer):
- Producer waits for the consumer to read data if buffer full
- Consumer waits until data is available if buffer empty

Producer-Consumer Problem



Producer-Consumer Problem


```
// Ringbuffer
Data[n] b; read = 0; write = 0;

init(m, 1); // put/get mutex
init(a, 0); // data available
init(f, N); // buffer full

void put(Data v) {
      b[write] = v;
      write = (write+1) % N;
}

Data get() {
      v = b[read];
      read = (read+1) % N;
      return v;
}
```



```
// Producer
while (true) {
      v = produce();
      acquire(f);
      acquire(m);
      put(v);
      release(m);
      release(a);
}
// Consumer
while (true) {
      acquire(a);
      acquire(m);
      v = get();
      release(m);
      release(f);
      consume(v);
}
```

Condition Variable, Monitor

- Condition Variable:
  - await(c, v): blocks until c = v
  - signal(c, v): sets c to value v

- Monitor: abstract data type that provides mutual exclusion
  - Encapsulates variables: only accessible from monitor functions
  - Monitor functions only executed by one process at a time

- Java:
  - Monitor functions qualified `synchronized`
  - Boolean condition variable: `wait`, `notify`

More primitives

- Spinlocks:
  - Semaphore implemented by busy-waiting
- Event count:
  - Data structure: int c
  - advance(c): increments c
  - await(c, n): blocks while c < n
  - E.g. for periodic condition synchronisation

More primitives

- Sequencer:
  - Data structure: int s
  - ticket(s): returns s and increments s
- Barrier:
  - barrier operation
  - Processes in a group of processes can only proceed when all of them have called the barrier operation.
  - Memory barrier (fence): hardware-level synchronisation

Message Passing

Data structure: Queue

- Operations:
  - send: blocks until the message has been received by the receiver
  - receive: blocks until a message has been received
- Message Passing Interface (MPI):
- `void MPI Send(void *buff, int count, MPI Datatype type, int dest, int tag, int comm)`
- `void MPI Recv(void *buff, int count, MPI Datatype type, int source, int tag, int comm, MPI Status *status)`

Transactional Memory

Data structure: Undo logs

- Operations:
  - start transaction: starts a transaction
  - commit transaction: returns true if the transaction succeeded otherwise rolls back the changes
- Learning from database concepts
  - Atomicity - Consistency - Isolation - Durability
  - Requires conflict detection and resolution
  - Implementation in Software and Hardware
  - Challenging to implement efficiently

Summary

- Synchronisation objectives
  - Avoid inconsistencies
  - Control the sequencing of operations
- Critical Sections
  - Part of each process that we have to protect
  - Mutual exclusion, freedom of deadlock and freedom of starvation
- Common application problems:
  - Producer-Consumer
  - Readers-Writers (next lab)
- Synchronisation primitives:
  - Mutex, Semaphore, Monitor, Control variable, etc.
- Transactional Memory
- Message Passing

Read

- Tanenbaum & Bos., Modern Operating Systems
  - Chapter 2.5
- Silberschatz et al., Operating System Concepts
  - Chapter 6

Next Lecture

- Introduction
- Operating System Architectures
- Processes
- Threads - Programming
- Process Scheduling - Evaluation
- Process Synchronisation
- **Deadlocks**
- Memory Management
- File Systems
- Input / Output
- Security and Virtualisation