Anthony Ventresque

anthony.ventresque@ucd.ie

Operating Systems

COMP30640

Process Management IV
Synchronisation (2)

School of Computer Science, UCD

Scoil na Ríomheolaíochta, UCD

Announcements

- ***In-class test***:
  - 25th or 26th October(room not known yet)
  - 50 minutes
  - 20%

Outline

- Understand the Producer Consumer synchronisation problem
- Understand and be able to use higher-level synchronisation techniques:
  - monitors
  - messages

Take home message:

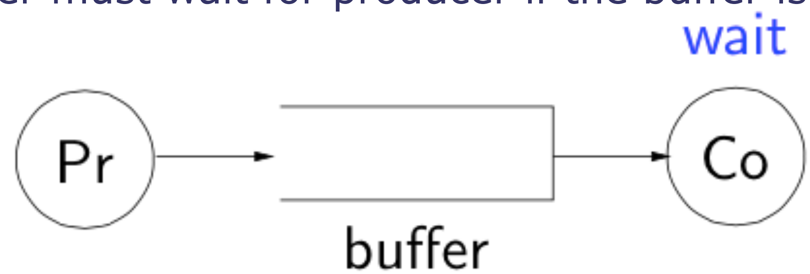
Semaphores can lead to deadlock and monitors can solve this.

Producer/Consumer Problem

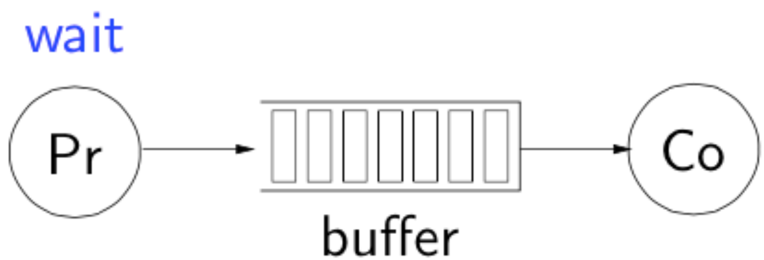
- A common synchronisation problem
- Consider a ***producer process*** supplying a resource (also called message) to a ***consumer process***
  - ***Producer***: creates instances of a resource
  - ***Consumer***: uses up (destroys) instances of a resource
- Producer and consumer share a ***buffer*** into which resources are placed by producer and from which resources are removed by consumer
  - Buffer has finite size (fixed number of slots)
- Example: use of a printer
  - Printing process: producer
  - Spooler: buffer
  - Printer: consumer

Producer/Consumer Problem Constraints

- Consumer must wait for producer if the buffer is empty



- Producer must wait for consumer if the buffer is full



- ***Synchronisation:*** keeping producer and consumer in step
  - Buffer constitutes a CS of the Producer/Consumer problem
  - Usually need mutually exclusive access to the buffer
- Note: here we have assumed just one producer and one consumer; in general there can be more than one producer and more than one consumer sharing a buffer

Producer/Consumer Problem: Naive Solution Attempt with Semaphores

```
semaphore S(1,NULL);
```

Producer

```
while(true) {
    msg=produce_message();
    P(S);
    if(buffer_full)
        wait();
    put_message(msg);
    V(S);
}
```

Consumer

```
while(true) {
    P(S);
    if(buffer_empty)
        wait();
    msg=get_message();
    V(S);
    consume_message(msg);
}
```

- Put message() and get message(): put a message in/ get a message from the buffer, respectively
- If buffer is full or empty “solution” above does not work

Producer/Consumer Algorithm with Semaphores

- Three semaphores are needed for a true solution:

```
int N=buffer_size;             (number of slots in the buffer)
semaphore S(1,NULL);           (anybody accessing the buffer?)
semaphore full_s(0,NULL);      (are there full slots?)
semaphore empty_s(N,NULL);     (are there empty slots?)
```

Producer

```
while(true) {
    msg=produce_message();
    P(empty_s);
    P(S);
    put_message(msg);
    V(S);
    V(full_s);
}
```

Consumer

```
while(true) {
    P(full_s);
    P(S);
    msg=get_message();
    V(S);
    V(empty_s);
    consume_message(msg);
}
```

- full_s and empty_s are ***counting semaphores***

Producer/Consumer Algorithm with Semaphores (Optimisation)

- It is also possible to allow simultaneous access to the buffer to one producer and one consumer: four semaphores are required

```
int N=buffer_size;
semaphore Sp(1,NULL);
semaphore Sc(1,NULL);
semaphore full_s(0,NULL);
semaphore empty_s(N,NULL);
```

(a producer accessing the buffer?)
(a consumer accessing the buffer?)

Producer

```
while(true) {
    msg=produce_message();
    P(empty_s);
    P(Sp);
    put_message(msg);
    V(Sp);
    V(full_s);
}
```

Consumer

```
while(true) {
    P(full_s);
    P(Sc);
    msg=get_message();
    V(Sc);
    V(empty_s);
    consume_message(msg);
}
```

Monitors

- Semaphores are nice but they have drawbacks:
  - Joint use of more than one semaphore can lead to ***deadlocks***, if the order of Ps is not correctly set
  - → difficult to program with semaphores
- Monitors are higher-level sync primitives (i.e., tied to a higher-level programming language) proposed by Hoare (1974)
  - A ***monitor*** is a collection of procedures, variables and data grouped together in a special kind of structure
  - Goal is to avoid the “catches” that can arise when protecting critical sections (CS) with semaphores

Monitor Structure

- Example: Monitor in pseudo-code

```
monitor example {
    int i;              (example internal data)
    condition c;        (example condition variable)

    void p() {          (example monitor procedure)
        ...
    }
}
```

- Rules:
  - Processes may call monitor procedures whenever they wish
  - Processes can't access internal monitor data (variables, etc) using external procedures
  - At most, only one process at any time can be active inside a monitor
    - This means ME inside the monitor, and it is guaranteed by the compiler

Blocking in Monitors

- ME is implemented relying on internal ***condition*** variables, which have two special operations associated:

1. ***wait***(*condition*): executed when the monitor discovers that a process cannot proceed
    - The monitor blocks a process calling ***wait***() and makes it wait on ***condition*** (blocked, hence out of the monitor)
    - Another process can be then allowed into the monitor
    - A process that invokes ***wait***() is ***always blocked*** (note: this is <u>unlike invoking P in semaphores</u>)
2. ***signal***(*condition*): executed to wake up a process waiting on condition
    - After executing ***signal***() the calling process <u>must exit the monitor immediately</u> (in order to guarantee ME)

Producer/Consumer Algorithm with Monitors

```
monitor pr_co {
   int count;
   condition full_s,empty_s;

   void put(msg) {
      if(count==N) wait(empty_s);
      put_message(msg);
      count++;
      if(count==1) signal(full_s);
   }

   msg get() {
      if(count==0) wait(full_s);
      msg=get_message();
      count--;
      if(count==N-1) signal(empty_s);
   }
}
```

Producer

```
while(true) {
   msg=produce_message();
   pr_co.put(msg);
}
```

Consumer

```
while(true) {
   msg=pr_co.get();
   consume_message(msg);
}
```

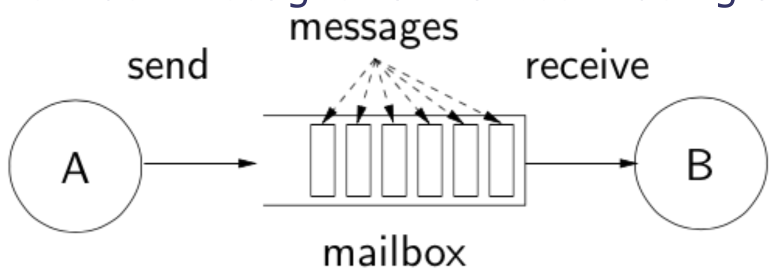
- Simplest possible consumer and producer code
- Monitor takes care of any issues, not producer or consumer

Message System

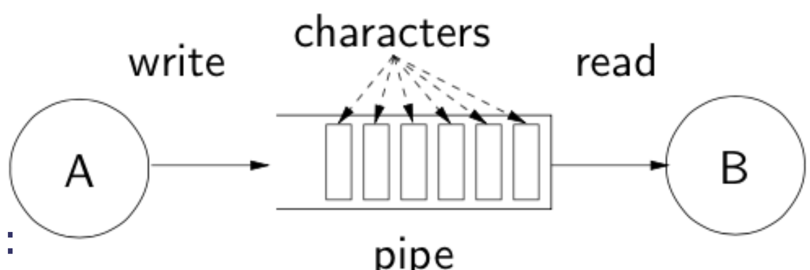
- Synchronisation with both semaphores and monitors relies on ***shared memory***
- The message system is a mechanism for inter-process communication (IPC), that can also be relied upon for synchronisation ***without having to share memory***
- Elements in the message system
  - ***Message*:** Information that can be exchanged between two (or more) processes or threads
  - ***Mailbox*:** A place where messages are stored between the time they are sent and the time they are received
    - It usually resides in kernel space

Message System Scheme

- Process A sends messages to Process B using a mailbox:



- Example: a **pipe** can be implemented with messages



- Remarks:
  - One process or another owns the data; never two at the same time (as it happens with data in shared memory)
  - Duplex communication (streams) would require two mailboxes
  - The message system is an example of a Producer /Consumer setting

Message System: Operations & Addressing

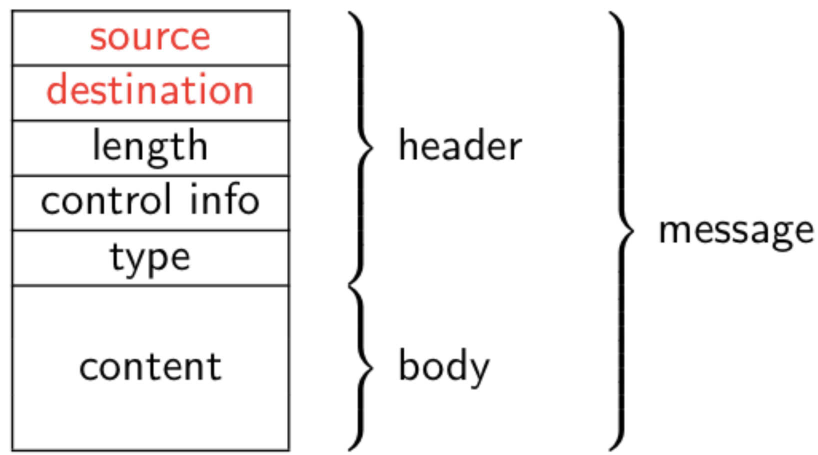
- Two basic operations (aka primitives):
  - ***send***(mailbox, message): place message in mailbox
  - ***receive***(mailbox, message): remove message from mailbox

- Addressing
  - Message identifies source and destination processes (for instance through their PIDs)
  - May be multiple destinations, or destination may be unspecified (broadcast)

- In practice we also need the system calls ***create***(mailbox), ***delete***(mailbox)

Message Format

- Depends on the objectives of the message facility
  - Fixed-length: minimise processing and storage overhead
  - Variable-length: more flexible approach for Oss



- No constraints with respect to content (it can be empty)

Why Use Messages?

- Many kinds of applications fit into the model of processing a sequential flow of information (e.g., Unix filters)

- Communicating processes sometimes need to be separate:
  - They do not trust each other (e.g., OS vs. user)
  - The programs were written at different times by different programmers who knew nothing about each other
  - They run on different processors (locally or on a network)
  - Less error-prone without shared memory

- Apart from sharing memory semaphores and monitors have other issues:
  - Semaphores are too low level (system calls very specific to OS)
  - Monitors need same structures/programming language for all processes

Mutual Exclusion with Messages (Example)

- Example: ME of **L processes** P(int I), sharing a resource (CS)

parent process

```
create_mailbox(mbox);
send(mbox,NULL);
parallel {
    P(1);
    P(2);
    ...
    P(L);
}
```

process P(int I)

```
while(true) {
    receive(mbox,msg);
        CS
    send(mbox,NULL);
}
```

- An empty message (NULL) in the mailbox is the token required to access the CS

Producer/Consumer Algorithm with Messages

- Not only do we need to ensure ME but also to keep a count:
  - Remember: Producer/Consumer buffer has finite size N
- Two mailboxes are needed
  - One mailbox for consumers: data produced and consumed
  - One mailbox for producers: accountancy purposes
- Producer and Consumer synchronise using these mailboxes as follows:
  - Producers generate data and send it to the consumer's mailbox when the producer's mailbox indicates there is at least one free slot in the buffer
  - Consumers take messages from the consumer's mailbox when available, and then send ***empty*** messages to the producer's mailbox (to signal new free slots)

Producer/Consumer Algorithm with Messages (II)

Parent Process

```
create_mailbox(cons_mbox);
create_mailbox(prod_mbox);
for(I=0;I<N;I++)
    send(prod_mbox,NULL);
parallel {
    producers();
    consumers();
}
```

producer()

```
while(true) {
    msg=produce_message();
    receive(prod_mbox,token);
    send(cons_mbox,msg);
}
```

consumer()

```
while(true) {
    receive(cons_mbox,msg);
    consume_message(msg);
    send(prod_mbox,NULL);
}
```

Conclusion

- Explored the resource Producer Consumer synchronisation problem and approaches to solve it
- Semaphores have drawbacks that can lead to deadlocks, if the order of Ps is not correctly set.
- Monitors are higher-level synchronisation primitives that avoid the “catches” that can arise when protecting critical sections (CS) with semaphores
- Messages are a mechanism for inter-process communication (IPC), that can also be relied upon for synchronisation without having to share memory