Slide 1

Concurrency Control – Part II

COMP3231 Operating Systems

2004/S2

Slide 2

SEMAPHORES

- ➜ Dijkstra (1965) introduced the concept of a semaphore in his study of cooperating sequential processes
- ➜ Semaphores are variables that are used to signal the status of shared resources to processes

How does that work?

- ➜ If a resource is not available, the corresponding semaphore blocks any process waiting for the resource
- ➜ Blocked processes are put into a process queue maintained by the semaphore (avoids busy waiting!)
- ➜ When a process releases a resource, it signals this by means of the semaphore
- ➜ Signalling resumes a blocked process if there is any
- ➜ Wait and signal operations cannot be interrupted
- ➜ Complex coordination can be specified by multiple semaphores

Slide 3

How are semaphores implemented?

- ➜ A semaphore is a variable `s` consisting of
  - an integer value `count` and
  - a process queue `queue`
- ➜ Initially, `count` is set to a nonnegative value and `queue` is empty
- ➜ There are two operations that a process `current` can apply:

`wait(s)`: Decrement `count`; if `count` becomes negative, put `current` into `queue`

`signal(s)`: Increment `count`; if `count` is not positive, unblock a process from `queue`

Slide 4

```
typedef struct {
  int     count;
  queue_t queue;
} semaphore;

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

Slide 5

There are various flavours of semaphores:

- ➜ Counting semaphores versus binary semaphores:
  - In a counting semaphore, `count` can take arbitrary integer values
  - In a binary semaphore, `count` can only be 0 or 1
  - Counting semaphores can be implemented in terms of binary semaphores (how?)
- ➜ Strong semaphores versus weak semaphores:
  - In a strong semaphore, `queue` adheres to the FIFO policy
  - In a weak semaphore, any process may be taken from `queue`
  - Strong semaphores can be implemented in terms of weak semaphores (how?)

Slide 6

Mutual Exclusion

Implementation of mutual exclusion with semaphores:

```
semaphore s;
 s.count = 1;
 s.queue = empty_queue ();

void proc (int i) {
  for (;;) {
    wait (s);
    <critical section>
    signal (s);
    <remainder>
  }
}
void main () {
  parbegin (proc (1), proc (2), ..., proc (n));}
```

Slide 7

Mutex:

- ➜ A semaphore that allows only one process in a critical section is often called a mutex
- ➜ There exist various flavours, such as, read-write mutexes and read-write-update mutexes
- ➜ Given exchange or test-and-set are available, easy to implement in user-level:
  ① test-and-set lock
  ② if succesful, return
  ③ if not, yield current thread, repeat

Slide 8

Semaphores in OS/161

- ➜ defined in `src/kern/thread/synch.c` and `src/kern/include/synch.h`
- ➜ operations are called:
  - `P` (proberen: try), instead of `wait`
  - `V` (verhogen: increase), instead of `signal`
- ➜ definition of data type `semaphore`

```
 struct semaphore {
         char *        name;
         volatile int count;
 };
struct semaphore* sem_create (const char *name, int initial_count);
void              P          (struct semaphore *);
void              V          (struct semaphore *);
void              sem_destroy(struct semaphore *);
```

- ➜ where is the queue??

Slide 9

Mutexes in OS/161

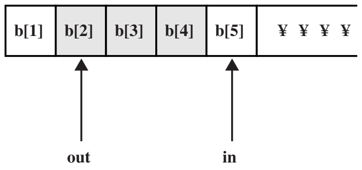
```
struct lock {
        char * name;
        struct thread  *volatile holder;
};

struct lock *lock_create    (const char *name);
void         lock_acquire   (struct lock *);
void         lock_release   (struct lock *);
int          lock_do_i_hold (struct lock *);
void         lock_destroy   (struct lock *);
```

Slide 10

Producer/Consumer Problem

- ➜ One or more producers are generating data and placing these in a buffer
- ➜ A single consumer is taking items out of the buffer one at time
- ➜ Only one producer or consumer may access the buffer at any one time



Slide 11

```
int    in, out;
elem_t b[];

producer:
  for (;;) {
    <produce item v>
    b[in] = v;
    in++;
  }

consumer:
for (;;) {
  while (in <= out)
    /* do nothing */;
  w = b[out];
  out++;
  <consume item w>
}
```

Slide 12

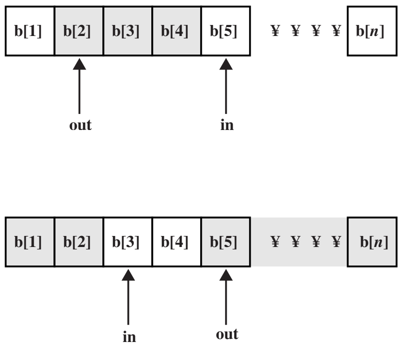
```
semaphore n = init_sem (0);  /* number of items in buffer  */
semaphore s = init_sem (1);  /* access to critical section */

void producer () {
  for (;;) {
    v = produce ();
    wait (s);
    append (v);
    signal (s); signal (n);
}  }

void consumer () {
  for (;;) {
    wait (n); wait (s);
    w = take ();
    signal (s);
    consume (w);
}  }
```

Slide 13

Producer with Circular Buffer



Slide 14

```
int    in, out;
elem_t b[];
```

Producer:

```
for (;;) {
  <produce item v>
  while ((in + 1) % n == out)
    /* do nothing */;
  b[in] = v;
  in = (in + 1) % n;
}
```

Consumer:

```
for (;;) {
  while (in == out)
    /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  <consume item w>
}
```

Assignment:

- ➜ How can this be implemented using semaphores or locks?
- ➜ How can this be implemented if copying to and from buffer takes a substantial amount of time?

Slide 15

Monitors

- ➜ A monitor is a software module implementing mutual exclusion
- ➜ Monitors are easier to program than semaphores
- ➜ Natively supported by a number of programming languages: Concurrent Pascal, Modula-(23) & Java
- ➜ Chief characteristics:
  - Local data variables are accessible only by the monitor (not externally)
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time
- ➜ Main problem: provides less control; coarse grain

Slide 16

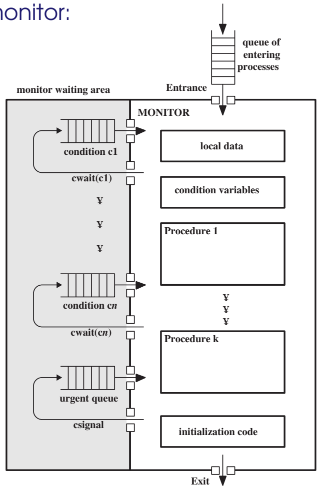
Synchronisation in a monitor:

`cwait (c)`: Suspend `current` on condition `c` (opens monitor to other processes)

`csignal (c)`: Resume execution of a processes suspended on condition `c` (ignored if no such process)

Slide 17

Structure of a monitor:



Slide 18

Producer/consumer using a monitor:

```
char        buffer[N];
int         nextin = 0, nextout = 0, count = 0;
condition_t not_full, not_empty;
```

```
void append (char c) {
  if (N == count)
    cwait (not_full);
  buffer[nextin] = c;
  nextin = (nextin + 1) % N;
  count++;
  csignal (not_empty);
}
```

```
void take (char c) {
  if (0 == count)
    cwait (not_empty);
  x = buffer[nextout];
  nextout = (nextout + 1) % N;
  count--;
  csignal (not_full);
}
```


Slide 19

IPC MECHANISMS

Several different mechanisms can be used for communication

① Shared memory: can be used to exchange information, but synchronisation issues remain
- threads: run in common memory space
- `mmap()` system call

② File system
- normal files
- pipes (FIFOs)
- sockets

③ Message passing
- more abstract communication mechanism

Slide 20

A CLOSER LOOK AT `mmap` AND PIPES

Memory mapped files:

➜ Processes can share files to communicate

➜ `mmap` maps a file into memory

Type of `mmap`:

```
void  *mmap (void      *addr,   /* dst address for map    */
             size_t    len,     /* length of data to map */
             int       prot,    /* protection             */
             int       flags,   /* misc flags             */
             int       fildes,  /* file descriptor        */
             off_t     offset)  /* offset into file       */
```

By using `mmap` in combination with `fork`, processes can simulate "shared memory".

Slide 21

Example: simplified, no error checking!

```
int main () {
  char   *data, *fname = "foo";
  int    fd;
  struct stat sbuf;
  fd = open (fname, O_RDONLY);
  stat (fname, &sbuf);

  data = mmap ((caddr_t) 0,      let the system choose the dest.
               sbuf.st_size,    will be rounded to pagesize
               PROT_READ,       mem. area will be read only
               MAP_SHARED,      shared: changes visible to all proc
               fd,              file desc. of previously opened file
               0);              offset

  printf ("Test mmap:...%c", data[0]);
}
```

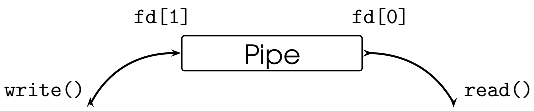
Slide 22

Pipes

What is a file descriptor?

- ➜ `fopen`, `fwrite`, etc are implemented in terms of systems calls like `open`, `write`
- ➜ `fopen` returns a `FILE` pointer
- ➜ `open` returns a file descriptor (`int`)
- ➜ `stdin`, `stdout`, `stderr` are file descriptors

`pipe()` returns a pair of file descriptors:



Slide 23

Example: (no error checking)

```
int  fd[2];
char buf[20];

pipe (fd);

write (fd[1], "test1", 6);
write (fd[1], "test2", 6);
read  (fd[0], buf, 6);
printf ("read %s from pipe \n", buf);
```

Usage:

- ➜ pipes can be used in combination with `fork`
- ➜ named pipes: `mknod`

```
mknod ("testFIFO", S_FIFO | 0644, 0);
```

Slide 24

Sockets

What is a socket?

- ➜ two-way communication pipe
- ➜ can be used to communicate in a wide variety of domains (e.g., internet)

Communication between Processes:

Sockets are also a file in the Unix file system, but offer a different interface

- ➜ `socket()`, `bind()`, `receive()` instead of `open()`, `read()` and `write()`
- ➜ typically used in client/server style programs

Slide 25

Server:

① create socket:

```
s = socket(AF_UNIX, SOCK_STREAM)
```

② bind socket to local address:

```
bind (s, <socket name>)
```

③ listen for incoming connections:

```
listen (s, <max size of incomming connection queue>)
```

④ main server loop:

① accept connection

```
s2 = accept (s, &<remote socket name>)
```

② receive/send

```
recv (s2, &<request>)
send (s2, <answer>)
```

③ close

```
close (s2)
```

Slide 26

Client:

① create socket:

```
s = socket(AF_UNIX, SOCK_STREAM)
```

② connect:

```
connect (s, <socket name>)
```

③ send/receive:

```
send (s, <request>)
recv (s, &<answer>)
```

④ close:

```
close (s)
```


Slide 27

MESSAGE-PASSING IPC

Primitives:

- ➜ Sending a message: `send(dest, msg)`
- ➜ Receiving a message: `receive(source, &msg)`

Different message passing styles:

- ➜ synchronisation: blocking (synchronous) vs. non-blocking (asynchronous)
  - blocking send, blocking receive
  - non-blocking send, blocking receive
  - non-blocking send, non-blocking receive
- ➜ addressing: direct vs. indirect
  - identifier of destination process
  - message to shared data structure (mailbox, port), one-to-one, one-to-many, many-to-many
- ➜ message format: depends on objectives, single computer vs. distributed system, fixed vs. variable-length messages

Slide 28

IPC Implementation Issues:

- ➜ Security & Safety
  - messages may be lost
  - authentication
- ➜ How are links established?
  - automatically
  - have to be set up explicitely
- ➜ What is the capacity of a link?
- ➜ Is the message format fixed or variable?
- ➜ Is a link uni-directional or bi-directional?

Slide 29

IPC: Direct Communication

Processes must name each other explicitly

- ➜ `send(pid, &msg)`
- ➜ `receive(pid, &msg)` — sometimes id of sender cannot be anticipated

Properties of communication link :

- ➜ links established automatically
- ➜ link associated with pair of processes
- ➜ exactly one link between each pair
- ➜ link may be uni- or bi-directional

Slide 30

Indirect communication

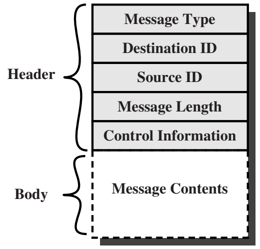
- ➜ Messages go via mailboxes (aka. ports)
  - each port has unique ID
  - communication requires sharing of a port
- ➜ Properties of communication link:
  - links established if processes share a port
  - link may be associated with many processes
  - each pair may share many links
  - link (port) may be uni- or bi-directional
- ➜ Operations: create, delete, send, receive

Slide 31

General Message Format

The format of a message depends on

- ➜ objectives of message facility
- ➜ local or distributed
- ➜ security and safety requirements



Slide 32

Message Buffering:

- ➜ Associate message buffer with link:
  - Zero capacity 0 messages
    - sender blocks until receive (rendezvous)
  - Bounded capacity: finite # messages
    - if full sender blocks or fails
  - Unbounded capacity: infinite # messages
    - sender never blocks

IPC Exception Conditions:

- ➜ Partner process terminated
- ➜ Partner uncommunicative (protocol failure)
- ➜ Message buffer overflow
- ➜ Message lost
- ➜ Message scrambled