

Week 9

Inter-process Communication

School of Information Technology and Electrical Engineering
The University of Queensland

Process Synchronization

- Concurrent access to shared data may result in data inconsistency
 - Remember, concurrent means interleaved threads of control
- Example of problem

```

int inUse; /* shared var, 1 if resource in use */
Process One
if(!inUse) {
    inUse=1;
    /* use resource */
    ...
}
Process Two
if(!inUse) {
    inUse=1;
    /* use resource */
    ...
}
    
```

Variable read interleaved between read and set (write)

Time

2

Critical Section

- A **critical section** of a process is a segment of code that shouldn't be interleaved with another process' critical section.
- Same for processes sharing data as for threads

3

Semaphores

- Similar to mutex, but more general
- Special integer variable
 - after initialisation, accessed only through two **atomic** operations
 - atomic = indivisible
 - No interleaving will happen
- Operations on semaphore S
 - wait(S) {
 - while(S <= 0) {
 - /* do nothing */
 - S--;
 - signal(S) {
 - S++;

Semaphores (cont.)

- wait(S) also known as P(S)
 - based on Dutch word Proberen (to test)
- signal(S) also known as V(S)
 - based on Dutch word Verhogen (to increment)
- Semaphore value can never be negative
- Need hardware/OS support to ensure that operations are indivisible

5

How to Use Semaphores

- Associate a semaphore S, initially 1, with each shared variable (or set of shared variables)
- Surround corresponding critical section with wait(S) and signal(S) operations:


```

wait(S)
...critical region...
signal(S)
            
```
- This is a **binary semaphore** - always 0 or 1
- Semaphore ensures **mutually exclusive** access to critical region
 - Binary semaphores used for mutual exclusion often called **mutexes**

6

COMP2303
COMP7306

Semaphores for Shared Resources

- If n resources available, initialise semaphore to n
 - allows up to n users
- Generalization of mutex

7

COMP2303
COMP7306

Busy waiting

- We wrote:


```
wait(S) {
    while(S <= 0) {
        ; /* do nothing */
    }
    S--;
}
```
- but OS doesn't actually busy wait
 - Process shifted to waiting queue
 - Process shifted to ready queue when semaphore available
 - If more than one process waiting on a particular semaphore, need to choose process appropriately to prevent **starvation** (i.e. one process waiting indefinitely)

8

COMP2303
COMP7306

Deadlock

- Some processes may wait forever, e.g.

Process One wait(S1); wait(S2); ... signal(S1); signal(S2);	Process Two wait(S2); wait(S1); ... signal(S2); signal(S1);
---	---

← May get stuck here
- Need deadlock avoidance strategies
 - Beyond scope of this course

9

COMP2303
COMP7306

Semaphore APIs

- System V Semaphore API
 - Very complicated to use
 - semget(), semctl(), semop()
- POSIX Semaphore API
 - sem_open()
 - sem_close()
 - sem_unlink()
 - sem_wait() /* wait() or P() */
 - sem_post() /* signal() or V() */
 - Other functions also, e.g.
 - sem_getvalue() – return value of semaphore
 - sem_trywait() – don't block if semaphore is 0
 - sem_timedwait() – wait, but only for a while
- No semaphore programming in this course

10

COMP2303
COMP7306

Shared memory

- Always the case with threads (except for the stack)
 - all allocated memory is shared
- Processes
 - fork() copies memory – memory is not shared
 - There are APIs to share memory between processes, two common APIs:
 - POSIX API
 - shm_open(), shm_unlink() along with mmap()
 - System V API
 - shmget(), shmat(), shmdt(): get, attach, detach
- More details on Thursday...

11

COMP2303
COMP7306

Shared memory APIs

- Two main options:
 - System V
 - POSIX
- We've used POSIX APIs previously, so for familiarity, we'll look at this one
- Both are similar in terms of functionality

12

COMP2303
COMP7306

POSIX shared memory

- Overall outline:
 - Create a segment of memory to share
 - specify it's size
 - map it into the process' address space
 - use it however we choose...
- Shared memory objects are persistent (at least until reboot)

13

COMP2303
COMP7306

Creating: `shm_open`

- First, we need to create a segment of memory to share:
 - `int shm_open(const char *name, int flags, mode_t perms);`
 - name must start with /
 - Use `O_CREAT` flag (as with regular files) if needed
 - perms is the permissions, as for files
 - Return value is a file descriptor

14

COMP2303
COMP7306

Setting size: `ftruncate`

- To set the size of the segment:
 - `int ftruncate(int fildes, off_t length);`
 - Use the descriptor obtained and the length desired
- So how do we access our newly created chunk like any other chunk of memory?
 - need to map it into our process' address space

15

COMP2303
COMP7306

Mapping memory: `mmap`

- We can then map the segment into our process' address space:
 - `void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);`
 - addr: desired address (or NULL)
 - len: length of the segment
 - prot: protection (read-only etc)
 - flags: various flags
 - fildes: the file descriptor we're mapping
 - off: offset within the segment (or file)
 - Return value is a pointer to the memory
 - `mmap` can be used with regular files as well

16

COMP2303
COMP7306

`mmap`: options

- `addr` can be specified explicitly (not usually a reason to do this though)
- `off` and `len` can be used to map only part of the object
- `prot`:
 - `PROT_READ` – data can be read
 - `PROT_WRITE` – data can be written
 - `PROT_EXEC` – data can be executed
 - `PROT_NONE` – data cannot be accessed (?)

17

COMP2303
COMP7306

`mmap`: options

- `flags`:
 - `MAP_SHARED` – changes are immediately visible
 - `MAP_PRIVATE` – changes are private (doesn't make sense for shared memory)
 - `MAP_FIXED` – map to the exact address specified (fail if it's in use)
 - and others...

18

COMP2303
COMP7306

Unmapping: `munmap`

- When we're finished with the segment, we unmap it from our address space:
 - `int munmap(void *addr, size_t len);`
 - `addr`: pointer to the address (usually the same as was returned from `mmap`)
 - `len`: the length of the segment (usually the value specified to `mmap`)

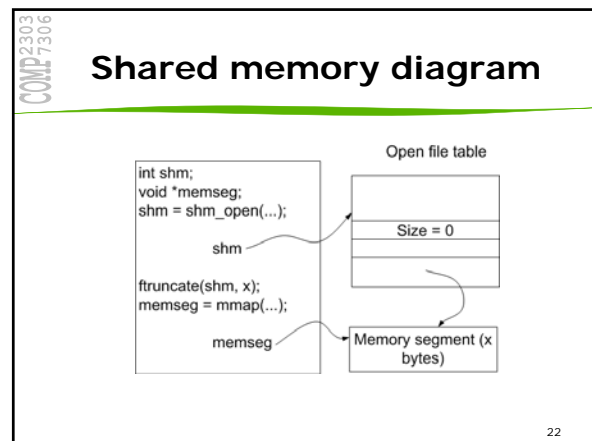
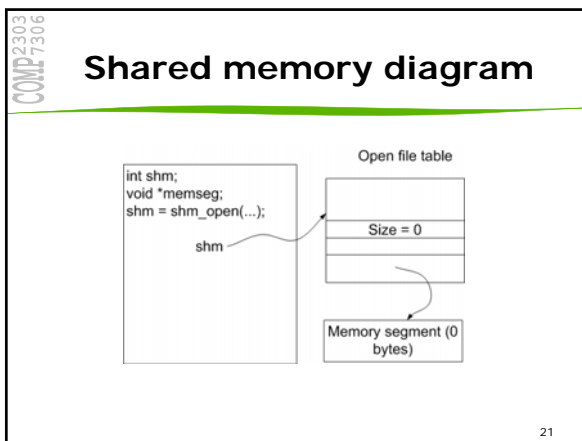
19

COMP2303
COMP7306

Removing segments: `shm_unlink`

- Segments remain mapped (at least until the machine is rebooted)
- Need to explicitly unlink (as with files) to remove:
 - `int shm_unlink(const char *name);`
 - Simply the name of the shared memory object

20



COMP2303
COMP7306

An example: `shm1.c` and `shm2.c`

- Two arbitrary processes can access the memory
- Compare with `pipe()`
- `shm1`:
 - Creates segment
 - Sets its size
 - Maps it
 - Writes some data
 - Waits for id to be changed
 - Removes the segment
- `shm2`:
 - Opens segment
 - Maps it
 - Waits until data has been written
 - Reads the data
 - Sets id to 0

23

```

/* shm1.c */
#include <sys/mman.h> /* For the shm and mmap functions */
#include <unistd.h> /* for ftruncate */
#include <fcntl.h> /* For O_CREAT etc */
#include <stdlib.h>
struct data {
    int id;
    int num;
};
int main(void) {
    int shm, status;
    struct data *mydata;

    shm = shm_open("/shmseg", O_CREAT | O_RDWR, 0777);
    status = ftruncate(shm, sizeof(struct data));
    mydata = (struct data *)mmap(NULL, sizeof(struct data),
        PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);
    mydata->num = 1234;
    mydata->id = 1;
    while (mydata->id == 1);
    shm_unlink("/shmseg");
    exit(0);
}
    
```

24

```
/* shm2 - includes and struct data as for shm1 */
int main(void) {
    int shm, status;
    struct data *mydata;

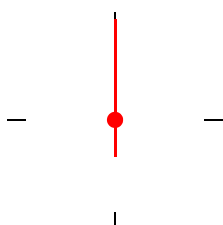
    shm = shm_open("/shmseg", O_RDWR, 0777);
    mydata = (struct data *)mmap(NULL, sizeof(struct
        data), PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);

    while (mydata->id != 1);
    printf("Data has been written: %d\n", mydata->num);
    mydata->id = 0;
    exit(0);
}
```

25

Short Break

- Stand up and stretch



26

Synchronization

- This example uses primitive synchronization (waiting for values to change etc)
- For real programs, will usually need real synchronization methods (like semaphores)

27

Shared memory - summary

- How to use shared memory:
 - Create/open – shm_open
 - (Set size – ftruncate)
 - Map to process – mmap
 - Use!
 - (Unmap – munmap)
 - (Delete – shm_unlink)
- The only functions specific to shared memory are shm_open and shm_unlink

28

Shared memory

- Why use shared memory?
 - High bandwidth communication (compare with pipes)
 - Arbitrary processes can connect to shared memory object (compare with threads)

29

IPC summary

- IPC mechanisms covered:
 - Signals
 - Pipes
 - Shared memory
- Which one should you use?
 - It depends!

30

COMP2303
COMP7306

Other IPC mechanisms

- File locking
 - See `lockf()`
- Named pipes (FIFOs)
 - See `mkfifo()`
- Message queues
 - System V – `msgget()`, `msgctl()`, `msgsnd()`, `msgrcv()`
 - POSIX – `mq_open()`, `mq_close()`, `mq_send()`, `mq_receive()`

↑ Beyond scope of course ↑

- Sockets
 - Covered in Week 11

31

COMP2303
COMP7306

Processes v Threads revisited

- How do you implement your application?
- Processes
- Threads

32

COMP2303
COMP7306

Review - functions

- `pthread_create` v `fork`

33

COMP2303
COMP7306

Review - functions

- `pthread_join` v `wait/waitpid`
- zombies...

34

COMP2303
COMP7306

Review - functions

- `exec`
- `pthread_detach`

35

COMP2303
COMP7306

Review - functions

- `exit` v `pthread_exit`

36

COMP2303
COMP7306

Review - functions

- getpid v pthread_self

37

COMP2303
COMP7306

Review - functions

- kill v pthread_cancel

38

COMP2303
COMP7306

Review

- Threads memory v POSIX shared memory

39

COMP2303
COMP7306

Review

- Mutexes
- Condition variables
- Semaphores

COMP2303
COMP7306

Review

- pipes
- signals