CSci 4061
Introduction to Operating Systems

IPC: Message Passing, Shared Memory

Chap 15.1,15.3-15.4

IPC Thusfar

- Files
- Pipes
- Limitations?

Message-Passing

- Unix uses a mailbox-like mechanism
  - Message-queue
  - Sender puts messages into queue
  - Receiver pulls them out

uniquely identifies queue (int)

access permissions on queue

```
#include <sys/msg.h>
int msgget (key_t key,
            int permflags);
```

returns queue_id used for send/receive

Message-Passing (cont'd)

- Message queue is typically larger than pipe buffer
- Unrelated processes can share queue
- Persistent: may outlive the process that created it!
- Meant for discrete messages vs. a near-infinite data stream

- But, still works only on same machine

Send / Receive

```
int msgsnd (int qid,
            const void *message,
            size_t size, int flags)

int msgrcv (int qid,
            void *message,
            size_t size,
            long msg_type, int flags)
```

Only 1 useful flag

Send/Receive (cont'd)

- Both `msgrcv` and `msgsnd` return an error if queue no longer exists

- Message data type

```
typedef char data_t [SOMESIZE];

struct mymsg_t {
     long mtype;  // used for tag selection
     data_t data;
                  // just bunch of contig. bytes
}
```

Example

Sender.c

```
mymsg_t m1 = {15, "hello"},
mymsg_t m2 = {20, "goodbye"};
int mid;
key_t key = 100;
```

u,g,o can read/write into queue

0666

```
mid = msget (key, 0777 | IPC_CREAT);
msgsnd (mid, (void *)&m1, sizeof (data_t), 0);
msgsnd (mid, (void *)&m2, sizeof (data_t), 0);
```

`msgsnd` **will block if queue is full, otherwise:**

```
msgsnd (mid, (void *)&m1, sizeof (data_t),
                                IPC_NOWAIT);
```

Returns -1 if cannot send (and `errno` **=** `ENOMSG`**)**

Example (cont'd)

<u>Receiver.c</u>

```
data_t msg;
int mid;
key_t key = 100;
mid = msget (key, 0666 | IPC_CREAT);
// read msgs with tag 15 and 20
// will block if such messages are not there
msgrcv (mid, (void *)&msg, sizeof (data_t), 20, 0);
msgrcv (mid, (void *)&msg, sizeof (data_t), 15, 0);
```

non-blocking:

```
res = msgrcv (mid, (void *)&msg, sizeof (data_t),
              30, IPC_NOWAIT);
```

Returns -1 if not on queue (and `errno` = `ENOMSG`)

Send/Receive (cont'd)

- If `msg_type` = 0 then return oldest message
  - `msgrcv (mid, (void *)&msg, 0, 0);`
- If `msg_type` < 0 then return message with smallest tag up to X, where X = abs(tag)
  - `msgrcv (mid, (void *)&msg, -99, 0);`
  - Return `msg` with smallest tag <= 99
  - Implements priorities!
  - Or direct messages e.g.:

    jim,sally have tags 33, 55

Pass Arbitrary Data/Messages

- Easy

```
struct mymsg_t {
    long mtype;
    int x;
    int y;
    …
}
```

Restriction:

no pointers

Remove queue

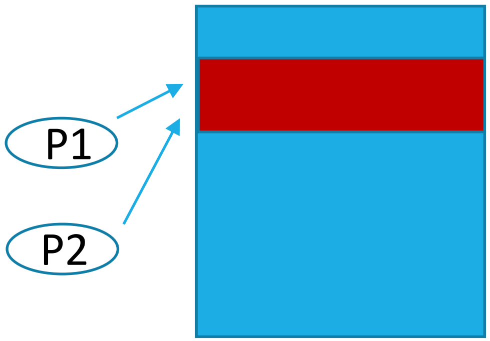
```
msgctl (int qid, IPC_RMID, 0);
```

Shared-Memory in Unix

- Shared-memory allows two or more processes to share a segment of physical memory
  - IPC => read/write shared memory locations
  - E.g. P1:write x, P2: read x

- Why is this the most efficient form of IPC?

- Why must it be used carefully?



- Which one (IPC methods) to use?
  - Personal preference

Shared memory (cont'd)

- In Unix, shared memory requires these steps

1. Create shared-memory segment

Mem size

Unique key

```
#include <sys/shm.h>
int shmget (key_t key, size_t size,
                    int permflags);
```

permissions same as in message queues (execute not used)

Returns segment id (shmid) for subsequent calls

As with message queues, can outlive the creating process!

Shared memory (cont’d)

2. Each process must attach to the segment (extends their VAS)

```
void *shmat (int shmid,
                const void *daddr,
                int shmflags);
```

0: R/W

NULL

Returns start address of segment: error (void*)-1
Can be different in different processes (virtual addresses)

[picture]

Shared memory (cont'd)

3. Detach from shared-memory segment

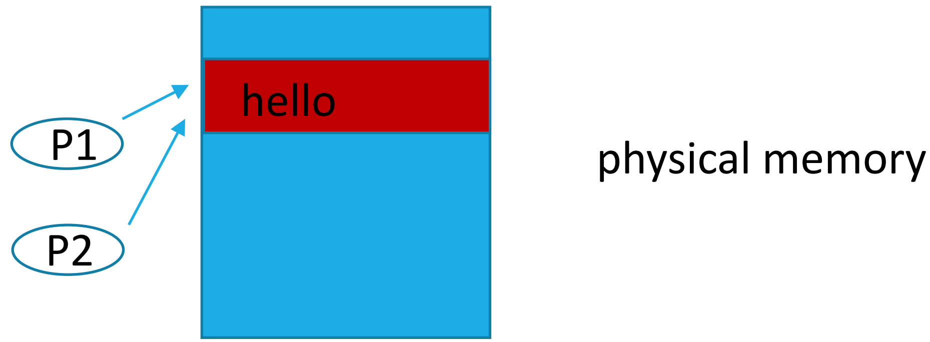
```
int shmdt (void *arg);
```

// `arg` is return ptr from `shmat`

4. Remove shared-memory segment for good

```
shmctl (shmid, IPC_RMID, 0);
```

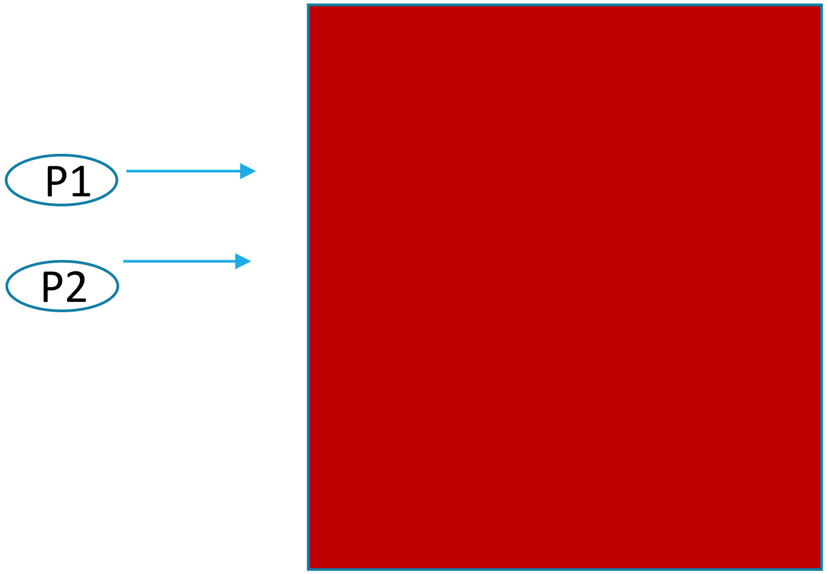
SM Repeat



- Create shared-memory segment `shmget`
  - Do this once -> return handle afterwards
- Each process must attach to the segment (extends their VAS) `shmat`
  - Using the handle
- Use the returned memory address: read/write to share or communicate

Example

- Put a shared buffer in shared memory region
- <picture>



Example

- Shared buffer

```
#define MaxItems 1024
struct buffer_t {
      int next_slot_to_store;
      int next_slot_to_retrieve;
      item_t items [MaxItems];
      int num_items;
}
item_t remove_item (buffer_t *b);
void produce_stuff (buffer_t *b,
                     item_t new_item);
```

Example: Program that uses buffer

```
void main () {
      int BUFFER_KEY = 100;
      buffer_t *b;
      item_t item;
      shmid = shmget (BUFFER_KEY, sizeof (buffer),
                              0666 | IPC_CREAT);
      b = (buffer_t *) shmat (shmid, 0, 0);
      b->next_slot_to_store = 0;
      b->next_slot_to_retrieve = 0;
      // initialize item to store
      produce_stuff (b, item);
      …
      item = remove_item (b);
      …
      shmdt ((void*) b); }  // process can't use b
```

Example (cont'd)

```
void produce_stuff
(buffer_t *b, item_t new_item) {
      if (b->num_items == MaxItems)
            return ERROR;  // later, we'll block
      b->items [b->next_slot_to_store] = new_item;
      b->next_slot_to_store++;
      b->next_slot_to_store %= MaxItems;
      b->num_items++;
      return;
}
```

Example (cont'd)

```
item_t remove_item (buffer_t *b) {
      item_t item;
      if (b->num_items == 0)
            return ERROR; // later, we'll block
      item = b->items
            [b->next_slot_to_retrieve];
      b->next_slot_to_retrieve++;
      b->next_slot_to_retrieve %= MaxItems;

      b->num_items--;
      return item;
}
```

Multiple Processes

- For shared-memory to make sense, need multiple processes
- Multiple processes doing:

```
produce_stuff (b, item);
item = remove_item (b);
```

Assume shared memory segment is created and buffer is initialized

```
void main () { // producer
    int BUFFER_KEY = 100;
    buffer_t *b;
    item_t item;
    shmid = shmget (BUFFER_KEY,
      sizeof (buffer), 0666);
    b = (buffer_t *) shmat (shmid, 0, 0);
    while (1) {
          // initialize item to store
          produce_stuff (b, item);
          …
    }
}
```

```
void main () { // producer
    int BUFFER_KEY = 100;
    buffer_t *b;
    item_t item;
    shmid = shmget (BUFFER_KEY,
      sizeof (buffer), 0666);
    b = (buffer_t *) shmat (shmid, 0, 0);
    while (1) {
          // get item
          item = remove_item (b);
          …
    }
}
```

What may happen?