Parallel and Concurrent Programming CS3S666

Shared Memory

Last Time...

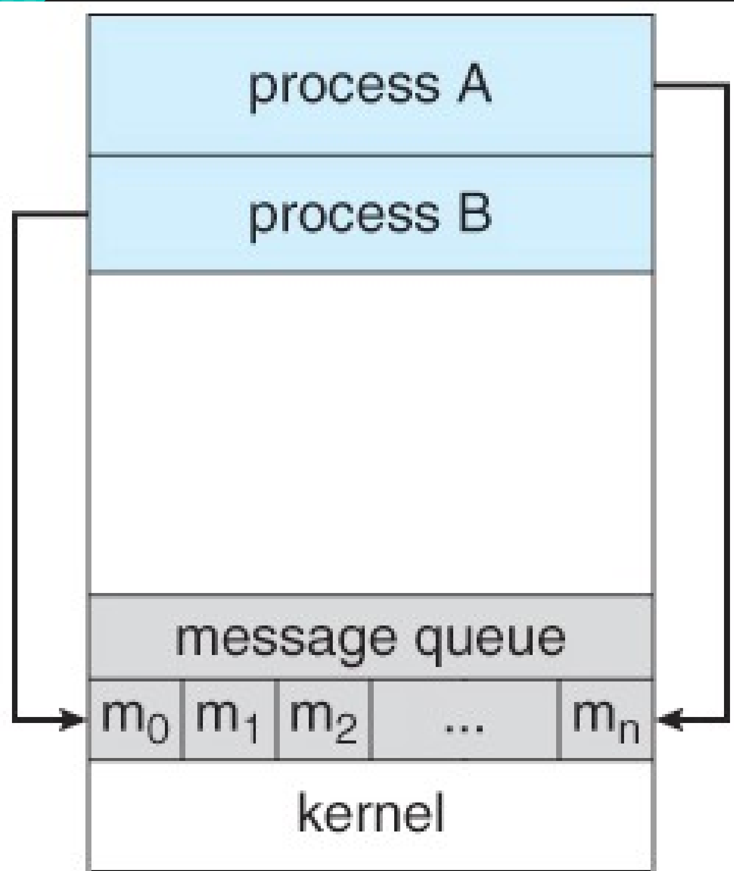
- Used named Pipes to communicate between unrelated processes.

- This week:
  - A closer look at Inter Process Communication.
  - Introduction to shared memory.

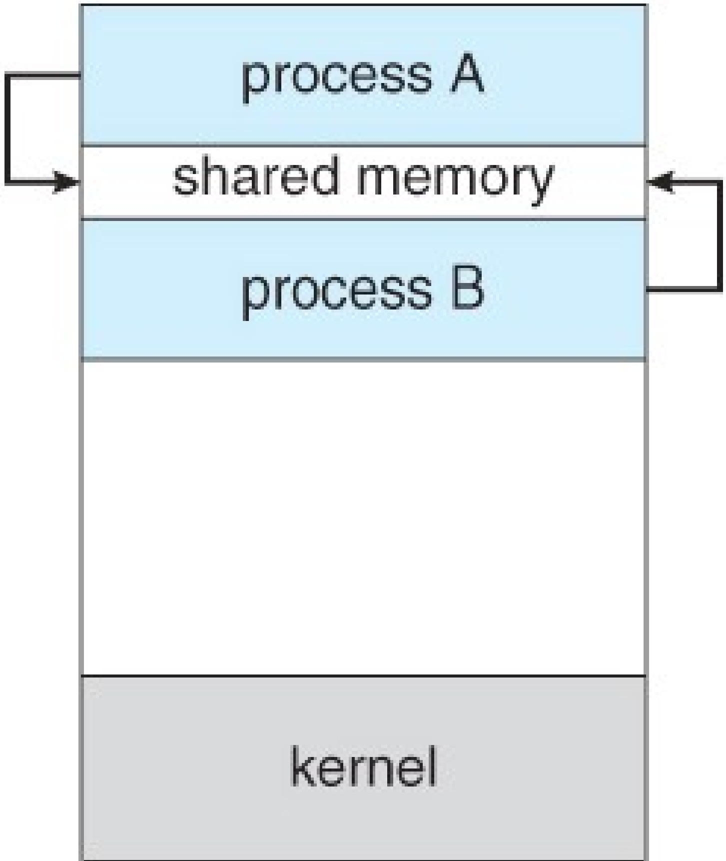
Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



(a) Message Passing



(b) Shared Memory

Cooperating Processes

- **Independent** process cannot affect or be affected by the execution of another process
- **Cooperating** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Shared Memory

- Shared Memory is memory that multiple processes can access provided they have the key.

- All following examples have following headers which will be omitted from all examples to sav

```
1  #include<cstdlib>
2  #include<iostream>
3  #include<unistd.h>
4  #include <sys/wait.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>
```

Creating Memory

```
11  int main(
12  {
13     key_t key;
14     int shmid;
15
16     shmid = shmget(key, 1024, 0644 | IPC_CREAT);
17
18     if (shmid == -1)
19     {
20        cout << "Unable to get shared space." << end
21        exit(1);
22     }
23
24     return(0);
25  }
```

Key so multiple processes can access memory.

Memory Size of segment in bytes.

Create segment if doesn't exist.

Segment access permissions.

Creating a Key

```
11  int main(void)
12  {
13      key_t key;
14      int shmid;
15  
16      if ((key = ftok("SharedMemory.cpp", 'R')) == -1)
17      {
18          cout << "Unable to create a key" << endl;
19      }
20  
21      shmid = shmget(key, 1024, 0644 | IPC_CREAT);
22  
23      if (shmid == -1)
24      {
25          cout << "Unable to get shared space." << endl;
26          exit(1);
27      }
28  
29      return(0);
30  }
```

A file accessible by all processes.

Id value.

Accessing Shared Memory Program 1

```
11  int main(void)
12  {
13    key_t key;
14    int shmid;
15
16    if ((key = ftok(".", 'R')) == -1)
17    {
18      cout << "Unable to create a key" << endl;
19    }
20
21    shmid = shmget(key, 1024, 0644 | IPC_CREAT);
22
23    if (shmid == -1)
24    {
25      cout << "Unable to get shared space." << endl;
26      exit(1);
27    }
28
29    char *data = (char *)shmat(shmid, (void *)0, 0);
30
31    cin >> data;
32
33    return(0);
34  }
```

"." in linux means ?

Function returns void pointer, so cast to type you want.

Can use shared data as if usual variable.

Attach Memory.

Don't worry about these.

Accessing Shared Memory Program 2

```
11  int main(void)
12  {
13    key_t key;
14    int shmid;
15
16    if ((key = ftok(".", 'R')) == -1)
17    {
18      cout << "Unable to create a key" << endl;
19    }
20
21    shmid = shmget(key, 1024, 0644 | IPC_CREAT);
22
23    if (shmid == -1)
24    {
25      cout << "Unable to get shared space." << endl;
26      exit(1);
27    }
28
29    char *data = (char *)shmat(shmid, (void *)0, 0);
30
31    cout << data;
32
33    return(0);
34  }
```

What's the difference ?

Example Run

Program 1

```
simon@simon-VirtualBox:~$ g++ -o SharedMemory SharedMemory.cpp
simon@simon-VirtualBox:~$ ./SharedMemory
HelloWorld
simon@simon-VirtualBox:~$
```

Typed In

Program 2

```
simon@simon-VirtualBox:~$ g++ -o SharedMemory2 SharedMemory2.cpp
simon@simon-VirtualBox:~$ ./SharedMemory2
HelloWorld
simon@simon-VirtualBox:~$
```

Printed out

Compared to previous examples

- Unlike pipes data is not deleted once it’s been accessed.
- Methods don’t block.
- We should also indicate memory not in use:
  - shmdt(data); (detaches data).

Producer-Consumer Problem

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size

Background

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffer elements (circular buffer).
- We can do so by having an integer **count** that keeps track of the number of full buffer elements.
- Initially, count is set to 0. It is incremented by the producer after it produces a new buffer element and is decremented by the consumer after it consumes a buffer element.

Producer (pseudo)

```
while (true)
{

    //produce an item and put in nextProduced
   while (counter == BUFFER_SIZE);// do nothing
   buffer [in] = nextProduced;
   in = (in + 1) % BUFFER_SIZE;
   counter++;
}
```

Consumer (pseudo)

```
 while (true)
{
   while (counter == 0); // do nothing
   nextConsumed =  buffer[out];
   out = (out + 1) % BUFFER_SIZE;
   counter--;
  //consume the item in nextConsumed
}
```

Producer in Code

```
11  int main(void)
12  {
13    const int BUFFER_SIZE = 10;
14
15    key_t key;
16    int shmid;
17
18    if ((key = ftok(".", 'S')) == -1)
19    {
20      cout << "Unable to create a key" << endl;
21    }
22    // Create size of buffer + 1, (+1 for counter).
23    shmid = shmget(key, (BUFFER_SIZE + 1) * sizeof(int), 0644 | IPC_CREAT);
24
25    if (shmid == -1)
26    {
27      cout << "Unable to get shared space." << endl;
28      exit(1);
29    }
30
31    int *data = (int *)shmat(shmid, (void *)0, 0);
32
33    int *counter = data; // First element as counter
34    *counter = 0;
35
36    int *buffer = data + 1; // Second element as data
37    int produced = 0;
38    int pos = 0;
39    while (true)
40    {
41      while (*counter == BUFFER_SIZE); // do nothing
42      buffer[pos] = produced++; // add something to buffer.
43      pos = (pos + 1) % BUFFER_SIZE; // get next position.
44      (*counter)++; // decrement counter.
45      cout << produced - 1 << endl;
46    }
47    shmdt(data);
48    return(0);
49  }
```

Consumer in Code

```
int main(void)
{
  const int BUFFER_SIZE = 10;

  key_t key;
  int shmid;

  if ((key = ftok(".", 'S')) == -1)
  {
    cout << "Unable to create a key" << endl;
  }

  // Create size of buffer + 1, (+1 for counter).
  shmid = shmget(key, (BUFFER_SIZE + 1) * sizeof(int), 0644 | IPC_CREAT);

  if (shmid == -1)
  {
    cout << "Unable to get shared space." << endl;
    exit(1);
  }

  int *data = (int *)shmat(shmid, (void *)0, 0);

  int *counter = data; // First element as counter
  int *buffer = data + 1; // Second element as data

  int produced = 0;
  int pos = 0;

  while (true)
  {
     while (*counter == 0); // do nothing
     produced = buffer[pos]; // get something from buffer.
     pos = (pos + 1) % BUFFER_SIZE; // get next position.
    (*counter)--;// decrement counter.
    cout << produced << endl;
  }
  shmdt(data);
  return(0);
}
```

Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
count = register2
```

Race Condition

- Consider this execution interleaving with “count = 5” initially:

  S0: producer execute register1 = counter {register1 = 5}

  S1: producer execute register1 = register1 + 1 {register1 = 6}

  S2: consumer execute register2 = counter {register2 = 5}

  S3: consumer execute register2 = register2 - 1 {register2 = 4}

  S4: producer execute counter = register1 {count = 6 }

  S5: consumer execute counter = register2 {count = 4}

Next Time

- More on Interprocess Communication.
- Critical Sections & Semaphores.