Inter-Processor Communication Design

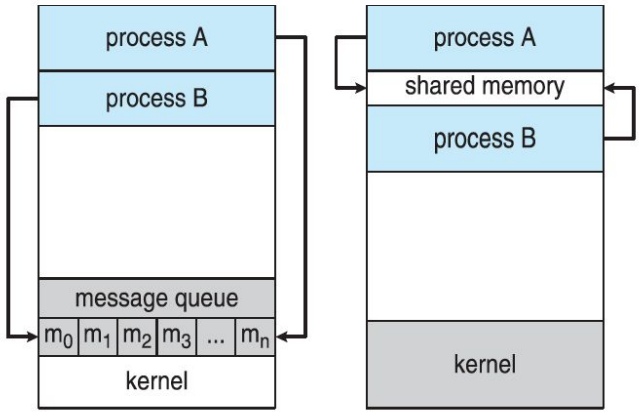
IPC Concepts

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.



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

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.

Interprocess Communication – Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - `send`(*message*)
  - `receive`(*message*)
- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

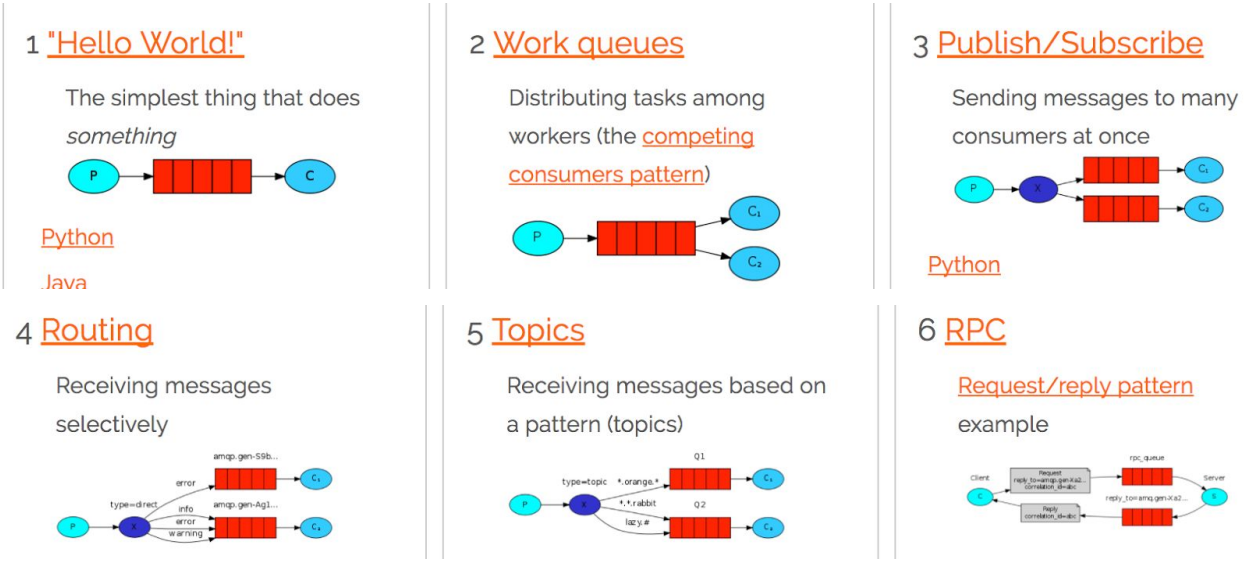
- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**

Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Message Passing Models:

From a popular message queue tool: RabbitMQ
https://www.rabbitmq.com/getstarted.html



IPC in Pi-OS: Topic-based message passing queue

The pi-OS provide the ability for applications to communicate. The kernel will keep a queue of topics, each topic may have several subscribers. Each task can subscribe some topics. Each task has a message queue to store all the messages of booked topics. When a sender send a message of topic_A, the kernel is responsible for passing this message to all the subscribers.

Core data structures:

```
64  #define TOPIC_QUEUE_LEN 8
65  static topic_struct * topic_struct_list[TOPIC_QUEUE_LEN];
```

```
13  typedef struct Tmessage_struct{
14      char topic[32];
15      char sender_name[32];
16      int points;  // Receiver can get points from this messsage
17      int msg_id;
18      char msg[64];
19  }message_struct;
20
21  #define MAX_SUBSCRIBER_NUM 10
22  typedef struct Ttopic_struct{
23      char topic[32];
24      int subscriber_number;
25      struct Ttask_struct* owner;
26      struct Ttask_struct* subscriber_list[MAX_SUBSCRIBER_NUM];
27  }topic_struct;
```

```
56  #define TASK_MSG_QUEUE_LEN 3
57
58  typedef struct Ttask_struct{
59      int state;
60      int prio;
61      int policy;
62      struct Ttask_struct* parent;
63      struct list_head tasks;
64      int pid;
65      void (*RunFunc)();
66      void (*taks_entry)(struct Ttask_struct * pTask);
67      unsigned    WakeTicks;
68      TTaskRegisters     Regs;
69      u8          *pStack;
70      unsigned        StackSize;
71      int active_child_count;
72      int isWaiting;
73      message_struct msgQueue[TASK_MSG_QUEUE_LEN];
74  }task_struct;
```

Important Functions

A task can create topic, subscribe a topic, send message to a topic group, and receive message.
Create topic: kernel will create a new topic struct, and put it in the topic_struct_list. This topic struct will be initialized to have zero subscribers.
Subscribe topic: kernel will put requesting task in the subscriber_list of a certain topic.
Send(topic, message*): kernel will copy this message to all the subscribers in this topic subscriber list. So this copy will be put in each subscribing task message queue. If queue full, oldest message will be replaced.
Receive(topic, message*): Task want to receive message from its message queue. Kernel will choose the oldest message in the queue, copy it. And remove this message in the message queue.

Mission:

Implement ipcSend(void * msg_send) & ipcReceive(void * msg_receive)

```
267 //Copy this message to all the subscribers' message queue;
268 //If msg queue full, replace the oldest one
269 int CKernel::ipcSend(void * msg_send){
270     //write_log("now in CKernel::ipcSend ");
271     message_struct *pMsg = (message_struct *) msg_send;
272     //TODO
273
274     return 0;
275 }
276
277 //Non-blocking receive; read message from message queue to msg_receive,
278 //read the oldest one and clear that space in message queue
279 int CKernel::ipcReceive(void * msg_receive){
280     //write_log("now in CKernel::ipcReceive ");
281     message_struct *pMsg = (message_struct *) msg_receive;
282     task_struct * pTask = getScheduler()->pCurrentTask;
283     //TODO
284
285     return 0;
286 }
```

The codes you need to add is about 50 lines, so it's mainly understanding work.

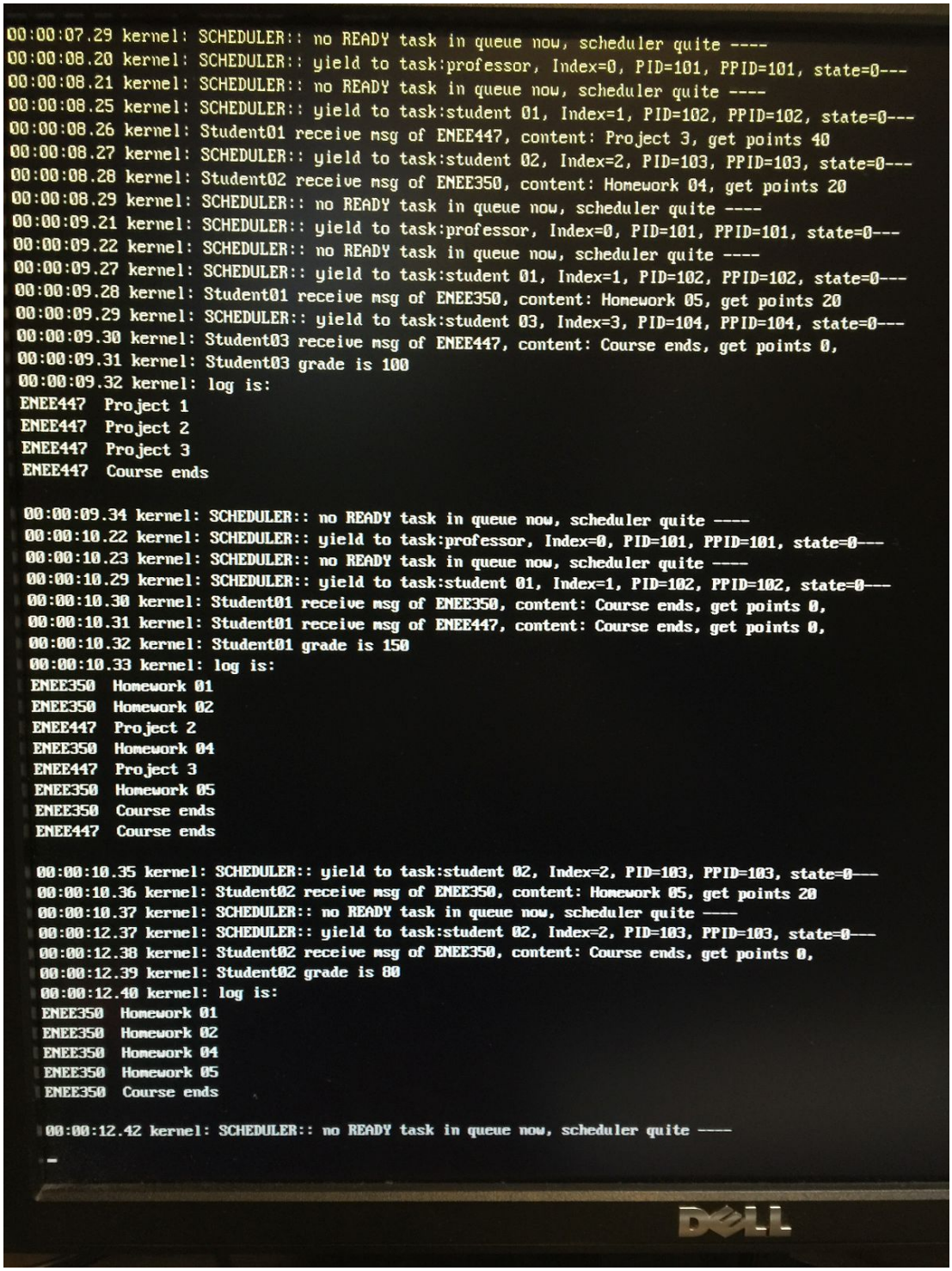
Tasks

There are four user tasks in project 6, professor task, student01 task, student02 task, student03 task, and their behavior is defined:

```
87  void task_professor_run()
88  {
89      message_struct msg;
90      printf("PROFESSOR:  Now we are in task_professor_run  ************");
91
92      ipc_create(TOPIC_01);
93      ipc_create(TOPIC_02);
94
95      sleep(2);
96      printf("PROFESSOR:  lecturing");
97      memset(&msg, 0, sizeof(msg));
98      strcpy(msg.topic, TOPIC_01);
99      strcpy(msg.msg, HOMEWORK_01);
100     msg.points = 20;
101     ipc_send(&msg);
102     sleep(1);
103     memset(&msg, 0, sizeof(msg));
104     strcpy(msg.topic, TOPIC_02);
105     strcpy(msg.msg, PROJECT_01);
106     msg.points = 30;
107     ipc_send(&msg);
108     sleep(1);
109     printf("PROFESSOR:  lecturing");
110     memset(&msg, 0, sizeof(msg));
111     strcpy(msg.topic, TOPIC_01);
112     strcpy(msg.msg, HOMEWORK_02);
113     msg.points = 20;
114     ipc_send(&msg);
```

```
180 void task_student01_run()
181 {
182     printf("STUDENT_01:  Now we are in task_student01_run  ************");
183     sleep(1);
184     ipc_subscribe(TOPIC_01);
185     ipc_subscribe(TOPIC_02);
186     sleep(2);
187     message_struct msg;
188     char log[200];
189     memset(log, 0 , 200);
190     void * pFile = open("stu01.log", 1);
191     seek(pFile, 0);
192     int recv_flag = 1;
193     int course_to_wait = 2;
194     int grade = 0;
195     while(course_to_wait){
196         memset(&msg, 0, sizeof(message_struct));
197         recv_flag = ipc_receive(&msg);
198         if(1==recv_flag){
199             printf("Student01 receive no msg, sleep ");
200             sleep(2);
201             continue;
202         }
203         if(0==msg.points){
204             printf("Student01 receive msg of %s, content: %s, get points %d, msg=%d", msg.topic, msg.msg,
205             course_to_wait --;
206             write(pFile, msg.topic, strlen(msg.topic));
207             write(pFile, "  ", 2);
208             write(pFile, msg.msg, strlen(msg.msg));
209             write(pFile, "\n", 1);
210             continue;
211         }
212         printf("Student01 receive msg of %s, content: %s, get points %d", msg.topic, msg.msg, msg.points);
213         write(pFile, msg.topic, strlen(msg.topic));
```

Sample output:



Goal:

Understanding IPC and message queue
Implementat send & receive