Notes and References on Project 1

Texas Cyber Range (TxCR) Hints

- 1. Do NOT shut down your TxCR VM! You do not have permission to restart it and will have to submit a trouble ticket to have it restarted.
- 2. The email address for a trouble ticket is txcr-support@tamu.edu
- 3. Expected turnaround time for a trouble ticket response is 24 hours
- 4. Project extensions will not be granted if you require helpdesk assistance within 24 hours of the due date.
- 5. I do not have admin rights on the Texas Cyber Range so I cannot resolve issues for.
- 6. The most straightforward way to move files between your TxCR VM and Canvas is via your Google Drive.
- 7. Be careful customizing your VM
  - Particularly installing packages that will not be present on the machine your code is tested on

Project 1

Develop a "correct solution" to the multiple dimensioned, multiple producer/consumer (That is multiple producers > 3 and multiple consumers > 3) problem that send and receive messages directly among themselves. Make sure that each message is received by one and only one consumer and that every producer and consumer is treated in a fair manner (no starvation). No centralized task manager is allowed. That is, you cannot simply queue all requests from each producer and then sequentially assign them to consumers. *No common shared memory*.

Producer:

1: if msg from Consumer, place msg on queue w/timestamp

2: send ack to Consumer (I.e., I have a resource)

Consumer:

1: determines resource need.

2: receives resource and processes it, return to step 1

To demonstrate your solution, you must illustrate it with actual running code (that completely documents your solution), and examples. (I.e., a correct implementation of Presser, Computing Surveys, 7,1, March 1975, pp.21-44).

Use gcc on your Texas Cyber Range VM to implement your solution.

Develop a "correct solution"

- to the multiple dimensioned, multiple producer/consumer problem.
- (That is multiple producers > 3 and multiple consumers > 3) problem that send and receive messages directly among themselves.
- How you demonstrate this is largely up to you.
  - Prompting the user for inputs is OK
  - Developing your own input file is OK
  - In either case – be explicit in demonstrating your solution
- Message Passing is required by the problem statement.
  - Message passing is NOT shared memory
- **Hints:**

1. Make sure that any library class you use does not violate the guidance in the assignment: notably no shared memory and no centralized task manager.

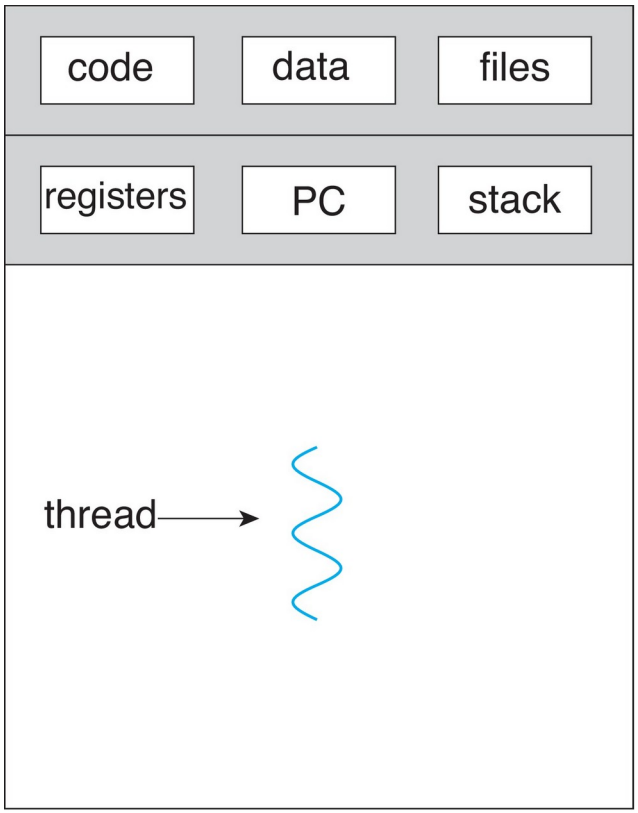
2. The differentiation between shared memory and message passing is illustrated in figure 3.11 in the textbook and on slide 30 in lesson 5 and 6.

3. The project is based on processes NOT threads.

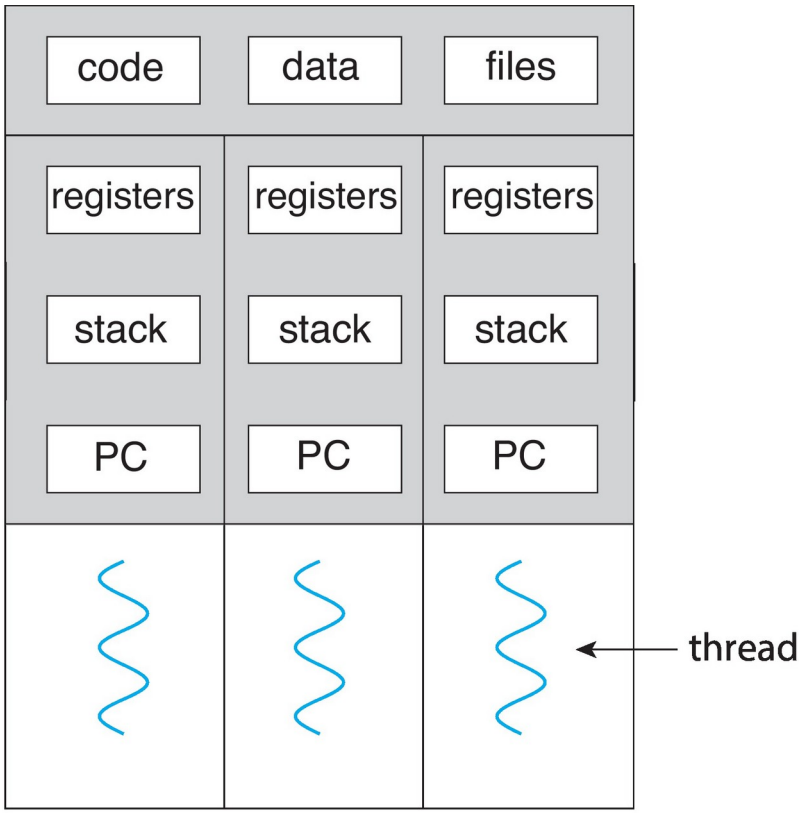
Common Errors (FFI)

1. Not using semaphores
2. Not using message passing
3. Hard coding runs that do NOT show correct a correct solution to the n-dimensioned multiple producer multiple consumer problem
4. Use of a centralized task manager
5. Queueing all requests from each producer and then sequentially assigning them to consumers.
6. Using predefined functions without understanding what the predefined function is actually doing.

Single and Multithreaded Processes



single-threaded process



multithreaded process

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

Interprocess Communication

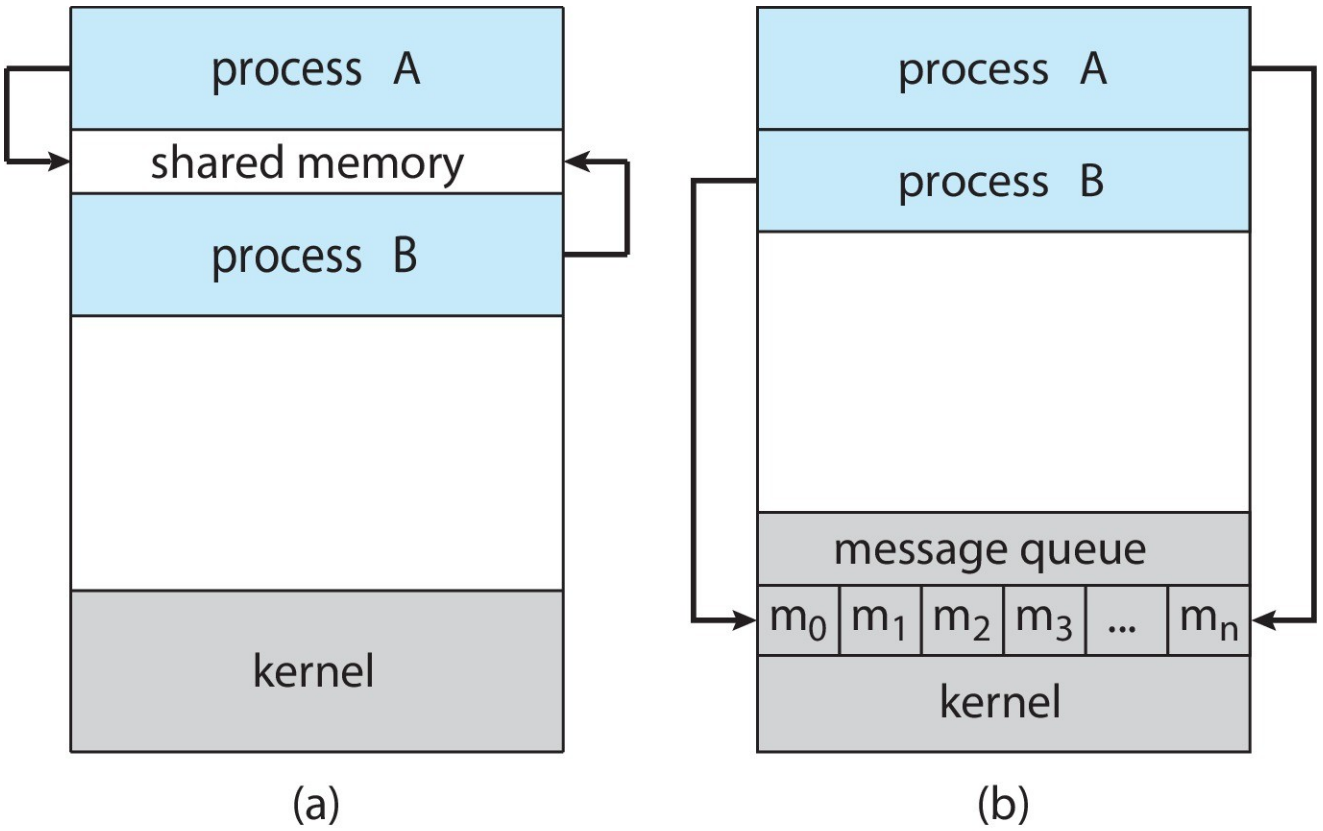
- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes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

IPC – 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.**
  - **Note from Hamilton – this excludes use of some shared memory declarations**
  - **If you choose a multithreaded option – you still must explicitly use message passing and the P and V primitives**
- Major issue is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
  - In the context of project 1 this refers to the queue specified in the problem statement.
- Synchronization is discussed in greater detail in Chapters 6 & 7.

Communications Models

(a) Shared memory. (b) Message passing.



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**
  - **A rendezvous is an example of centralized task management**

Producer-Consumer: Message Passing

- Producer

```
message next_produced;
while (true) {
 /* produce an item in next_produced */

 send(next_produced);
}
```

- Consumer

```
message next_consumed;
while (true) {
  receive(next_consumed)

 /* consume the item in next_consumed */
}
```

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)
     - This is an example of centralized task management
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Priority Scheduling (chapter 5)

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
  - Rumor has it that when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Deadlock versus Starvation in OS

Deadlock:
Deadlock occurs when each process holds a resource and wait for other resource held by any other process. Necessary conditions for deadlock to occur are Mutual Exclusion, Hold and Wait, No Preemption and Circular Wait. In this no process holding one resource and waiting for another get executed. For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1. Hence both process 1 and process 2 are in deadlock.

Starvation:
Starvation is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time. In heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. In starvation resources are continuously utilized by high priority processes. Problem of starvation can be resolved using Aging. In Aging priority of long waiting processes is gradually increased.