Justin Ciocoi

Oct. 29, 2023

CSCI 375 Textbook Notes

Chapter 3: Processes

3.1: Process Concept

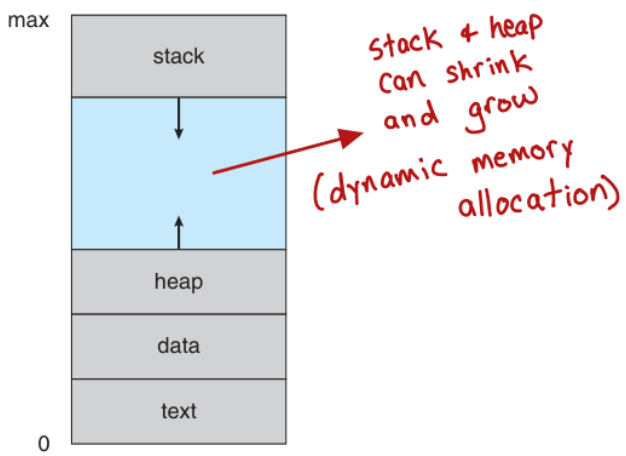
- **3.1.1: The Process**
  - Informally, a process is a program in execution
  - The status of a process's current activity is represented by the value of the *program counter* and the contents of the processor's registers
  - The memory layout of a process is typically divided into sections as shown below
  - 

    Figure 3.1 Layout of a process in memory.

  - The *text section* contains the executable code
  - The *data section* contains global variables
  - The *heap section* which is memory that is dynamically allocated during a program's run time
  - The *stack section* which is temporary data storage when invoking functions, such as function parameters, return addresses, and local variables

- Although the stack and heap sections of a process grow *towards* each other, the operating system must ensure that they do not overlap

- **3.1.2: Process State**
  - As a process executes, it changes its *state*
  - A process's state can be any one of the following
    - **New**, when the process is being created
    - **Running**, when instructions are being executed
    - **Waiting**, when the process is waiting for some event, such as I/O completion or signal reception, to occur
    - **Ready**, when a process is waiting to be assigned to a processor
    - **Terminated**, when a process has finished execution
  - These names are arbitrary and vary across operating systems, but the concepts presented here remain consistent

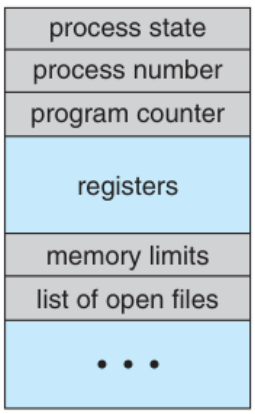
- **3.1.3: Process Control Block**
  - Each process is represented in the operating system by a *process control block (PCB)*, which is illustrated below
  - 

Figure 3.3 Process control block (PCB).

  - In brief, the PCB serves as the repository for all of the data needed to start or restart a process, along with some accounting data

- **3.1.4: Threads**

    - So far, the process model that has been described has implied that a process is a program that performs a single *thread* of execution
    - In most modern operating systems, the process concept has extended to allow a process to have multiple threads of execution and thus perform multiple tasks at a time

3.2: Process Scheduling

- The objective of multiprogramming is to have some process running on the CPU at all times such that the system is maximizing CPU utilization
- In order to meet this objective, the *process scheduler* selects a ready process for program execution on a core
- In general, most processes can be described as either *I/O bound* or *CPU bound*
    - I/O bound processes spend more time doing I/O than computations
    - CPU bound processes spend more time doing computations than I/O
- **3.2.1: Scheduling Queues**
    - As processes enter the system, they are put into a *ready queue*, where they are ready and waiting to execute on a CPU's core
    - Similarly, processes that in are in a waiting state are placed in a *waiting queue*
- **3.2.3: Context Switch**
    - As mentioned in chapter 1, interrupts case the operating system to change a CPU core from its current task and to run a kernel routine
    - When an interrupt occurs, the system must save the *context* of the currently running process so that it can restore that process when the interrupting process is done
    - Generically, we say that a system performs a *state save* of the current state of the CPU core, and then a *state restore* to resume operations
    - This operation is known as a *context switch*, and the duration of this operation is pure overhead since no useful work is done during a context switch
    - The duration of a context switch is highly dependent on hardware support

3.3: Operations on Processes

- **3.3.1: Process Creation**

- During the course of a process's execution, a process may create several new processes
  - Here, the creating process is called the *parent process* and the created process are called *child processes*
- Each of these processes may also create new processes, thus forming a *tree* or processes
- Most operating systems identify processes according to a unique *process identifier (PID)*, which is typically an integer number
- When a parent process creates a child process, that child process will need resources to accomplish its task
  - A child process might be able to obtain these resources directly from the operating system, or it might be constrained to a subset of the resources from the parent process
  - The parent process may partition its resources among child processes, or it may be able to share resources among several child processes
- In addition to providing resources to a child process, a parent process will also pass along initialization data, or input, to the child process
- When a process creates a new process, two possibilities for execution exist
  - The parent continues to execute concurrently with its children
  - The parent waits until some or all of its children have terminated

- **3.3.2: Process Termination**
  - A process will terminate when it finishes executing its final statement and asks the operating system to delete it by using the `exit()` system call
  - All of the resources of this process will be deallocated and reclaimed by the operating system
  - A parent process might terminate the execution of one of its child processes for a variety of reasons, such as
    - The child process has exceeded the usage of its resources which it has been allocated
    - The task assigned to the child process is no longer required
    - The parent is exiting, and the operating system does not allow a child process to continue if its parent terminates

    - When an operating system does not allow a child process to exist after its parent process has terminated, all child processes will also be terminated, which is known as *cascading termination*
    - A *zombie* process is one that has had its resources deallocated by the operating system, but its parent process has not yet called `wait()`
        - All processes transition into this state as they terminate, but generally they only spend a brief period of time as zombie processes
    - An *orphan* process is one that has had its resources deallocated by the operating system, and has had its parent process terminate before calling `wait()`

3.4: Inter-process Communication

- Processes which are executing concurrently in an operating system might be *independent processes*, which do not share data with any other processes, or *cooperating processes*, which can effect or be effected by other processes executing in the system
- There are a variety of different reasons why providing an environment that allows for process cooperation is advantageous
    - **Information sharing**, since multiple applications might be interested in the same piece of data, we should provide an environment to allow *concurrent* access to this data
    - **Computation speedup**, since we can increase the speed of computation by breaking a down into several smaller sub-tasks and executing them concurrently
        - This can only be achieved on a system with multiple computing cores
    - **Modularity**, which allows for more flexibility by dividing system functions into separate processes or threads
- Fundamentally, there are two models when it comes to the implementation of inter-process communication
    - *Shared memory*
    - *Message passing*
-

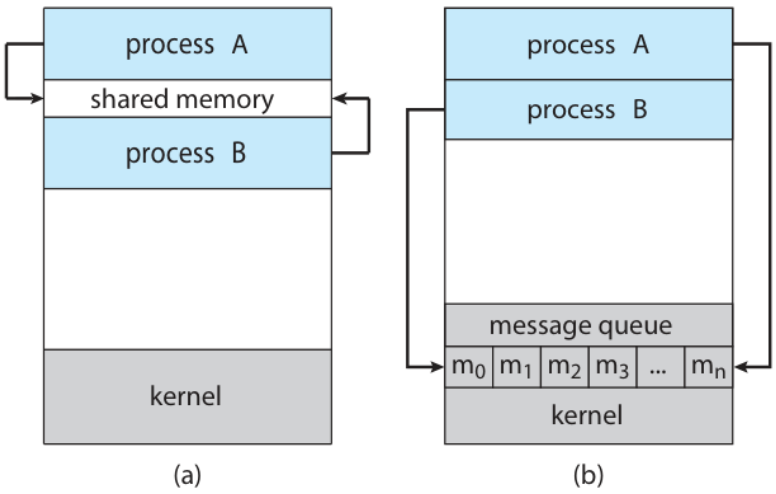


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

- Both of these models are common in operating systems, and many systems implement both
- Message passing is useful for exchanging smaller amounts of data, since no conflicts need to be avoided, and it is also easier to implement
- Despite the need to synchronize concurrent access to shared memory, using this approach is generally faster than message passing, since message passing is generally implemented using system calls and requires more time-consuming kernel interventions

3.5: IPC in Shared-Memory Systems

- Using shared memory as the medium for inter-process communication requires communicating processes to establish a region of shared memory
- Normally, the operating system tries to prevent one process from accessing another process's memory, but shared memory requires that two or more processes agree to lift this restriction
- Here, the processes are also responsible for ensuring that they are not writing to the same location simultaneously
- In order to illustrate the concept of cooperating processes in a fundamental sense, we will consider the **producer-consumer problem**
- A *producer* process produces information that is consumed by a *consumer* process
- **Producer**

```
item next_produced;

while(true)
{
    //produce an item in next_produced

    while(((in+1)%BUFFER_SIZE)==out)
        ; //do nothing

    buffer[in] = next_produced;
    in = (in+1) % BUFFER_SIZE;
}
```

- **Consumer**

```
item next_consumed;

while(true)
{
    while(in==out)
        ; // do nothing

    next_consumed = buffer[out];
    out = (out+1) % BUFFER_SIZE;

    //consume the item in next_consumed
}
```

- One solution to this problem uses shared memory, where a *buffer* of items exist that can be filled by the producer, and emptied by the consumer
- **Buffer**

```
#define BUFFER_SIZE 10

typedef struct{
    .....
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- This *buffer* will reside in a region of memory that is shared by the producer and consumer processes

- A producer can produce one item *while* the consumer is consuming another item
- Two types of buffers can be used
    - The *unbounded* buffer places no practical limit on the size of the buffer
    - The *bounded* buffer assumes a fixed buffer size

3.6: IPC in Message-Passing Systems

- Another way to achieve the effects of inter-process communication described in the previous section is to implement *message passing*
- This allows processes a mechanism to communicate and synchronize their actions without sharing the same address space\
- A message passing *facility* or the mechanism by which it is achieved, must provide at least two operations
    - `send(message)`
    - `receive(message)`
- **3.6.1: Naming**
    - Processes which want to communicate with each other must have a way to refer to each other
    - They can communicate either directly or indirectly
    - Under *direct communication*, each process that wishes to communicate with another must explicitly name the recipient or sender of the communication
    - In this scheme, the `send()` and `receive()` primitives are defined as
        - `send(P, message)` - Send a message to process P
        - `receive(Q, message)` - Receive a message from process Q
    - A link will be established automatically between every pair of processes that want to communicate
        - A link is associated with exactly two processes, and between each pair of processes, there exists exactly one link
    - This is a *symmetric* scheme, whereas an *asymmetric* scheme exists where only the sender names the recipient, and the receive primitive is defined as follows
        - `receive(id, message)` - Receive a message from any process where the variable `id` is set to the name of the process with which communication has taken place
    - In an *indirect* communication scheme, messages are sent to, and received from *mailboxes*, or *ports*

- For example, POSIX message queues use an integer value to identify a mailbox
- A process can communicate with another process via a number of different mailboxes, but two processes may only communicate if they have a shared mailbox
- Here, the `send()` and `receive()` primitives are defined as follows
  - `send(A, message)` - Send a message to mailbox A
  - `receive(A, message)` - Receive a message from mailbox A
- In this system, a link is established between two processes only if both have a shared mailbox
- A link may be associated with more than two processes, and between communicating processes, a number of links might exist, with each link corresponding to one mailbox

- **3.6.2: Synchronization**
  - Message passing may be implemented in either a *synchronous* or *asynchronous* manner
    - *Synchronous send*, where the sending process is blocked until the message is received by the receiving process or mailbox
    - *Asynchronous send*, where the sending process sends the message and resumes operations
    - *Synchronous receive*, where the receiver blocks until a message is available
    - *Asynchronous receive*, where the receiver retrieves either a valid message or a null
  - Different combinations of these `send()` and `receive()` primitives can be used
  - **Message-Passing Producer**
    - ```
      message next_produced;

      while(true)
      {
          //produce an item in next_produced

          send(next_produced);
      }
      ```
  - **Message-Passing Consumer**
      ```

```
message next_consumed;

while(true)
{
    receive(next_consumed);

    //consume item in next_consumed
}
```

- 3.6.3: Buffering
  - Messages that are being exchanged by communicating processes reside in a temporary queue, which is typically implemented in one of three ways
    - **Zero capacity**, where the queue has a maximum length of zero, thus not allowing any messages in the queue, so the sender must block until the recipient receives this message
    - **Bounded capacity**, where the queue has a finite length n
    - **Unbounded capacity**, where the queue's length is potentially infinite

3.7: Examples of IPC Systems

- 3.7.1: POSIX Shared Memory
  - Several different IPC mechanisms are available for use in POSIX systems, including shared memory and message passing
  - In shared memory, a process must first create a shared-memory object using the `shm_open()` system call
    - `fd = shm_open(name, 0_CREAT | 0_RDWR, 0666);`
  - The first parameter here specifies the name of the shared-memory object
  - The subsequent parameters specify that the shared-memory object has yet to be created (`0_CREAT`) and that the object is open for reading and writing (`0_RDWR`)
  - The last parameter specifies the file-access permissions of the shared-memory object
  - A successful call to `shm_open()` returns an integer file descriptor for the shared-memory object

- Once the object is established, the `ftruncate()` function is used to configure the size of the object in bytes using the call
  - `ftruncate(fd, 4096)`
- This call sets the size of the object to 4096 bytes
- Finally, the `mmap()` function established a memory-mapped file containing the shared-memory object, as well as a pointer to this file used for accessing the shared-memory object

- **POSIX Shared-Memory Producer**

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main()
{
    //size, in bytes, of shared-memory object
    const int SIZE = 4096;
    //name of shared-memory object
    const char *name = "OS";
    //strings written to shared memory
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    //shared-memory file descriptors
    int fd;
    //pointer to shared-memory object
    char *ptr;

    //create shared memory object
    fd = shm_open(name, O_CREAT || O_RDWR, 0666);

    //configure sie of shared-memory object
    ftruncate(fd, SIZE);

    //memory map the shared-memory object
    ptr = (char*)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    //write to shared-memory object
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);

    return 0;
}
```

- POSIX Shared-Memory Consumer

- ```c
  #include <stdio.h>
  #include <stdlib.h>
  #include <fcntl.h>
  #include <sys/shm.h>
  #include <sys/stat.h>
  #include <sys/mman.h>

  int main()
  {
      //the size, in bytes, of the shared memory object
      const int SIZE = 4096;

      //name of the shared-memory object
      const char *name = "OS";

      //shared memory file descriptor
      int fd;

      //pointer to shared-memory object
      char *ptr;

      //open the shared-memory object
      fd = shm_open(name, O_RDONLY, 0666);

      //memory map the shared-memory object
      ptr = (char *);
      mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

      //read from the shared-memory object
      printf("%s", (char *)ptr);

      //remove shared-memory object
      shm_unlink(name);

      return 0;
  }
  ```
  ```