Chapter 3

Objectives

- introduce the notion of a **process**, a program in execution.
- describe various features of processes, including scheduling, creating, termination, communication
- explore **interprocess communication** using **shared memory** and **message passing**
- describe communication in **client-server** systems

Process

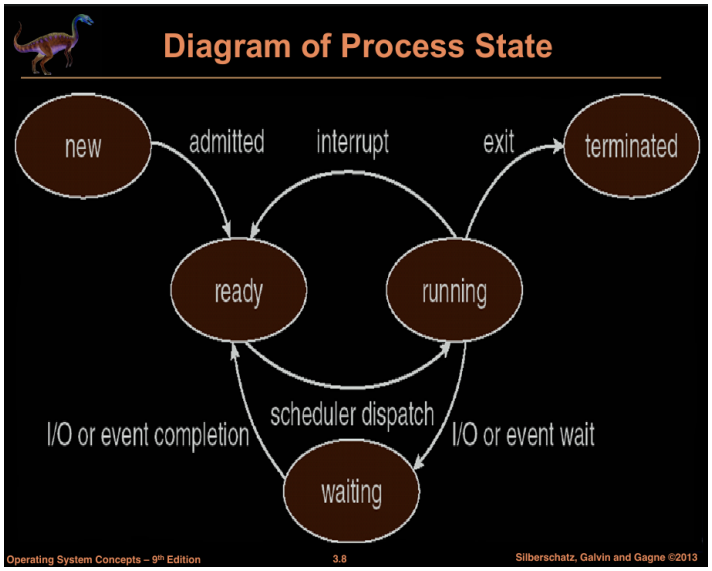
A process is a program in execution

Multiple parts (COMP-2660):

- program code, also called *text section*
- *program counter* (EIP register), processor registers
- *Stack* (runtime) containing temporary data; ie, stack segment, function params, return addresses, local vars
- *Data section* containing global variables
- *Heap* containing memory dynamically allocated during run time

Process State

State	
new	process is being created (or created but not yet admitted)
ready	process is waiting to be assigned to a processor
running	instructions are being executed
waiting	process is waiting for some event to occur
terminated	process has finished execution



Process Control Block

Process represente in OS by a *task control block*

- contains process state: running, waiting, etc
- *program counter*: address of next instruction to be executed; EIP register
- *CPU registers*: contents of all process-centric registers - EAX, ESI, ESP, EFLAGS, etc
- *CPU scheduling* information: memory allocated to the process, EBP, segment registers, page and segment tables, etc
- *Accounting information*: CPU used, clock time elapsed since start, time limits
- *I/O status info*: I/O devices allocates to process, list of open files, etc

Process Scheduling

Process scheduler selects among available processes to be executed on CPU (= CPU scheduler + job scheduler + …)

- **Job Queue**: set of all processes in the system
- **Ready Queue**: set of all processes residing in main memory, ready and waiting to be executed (= linked list of PCBs)
- **Device Qeueues**: set of processes waiting for an I/O device

Schedulers

- **Short-term scheduler** (or CPU scheduler): selects which process should be executed next and allocates the CPU to that process (chapter 6)
  - sometimes only scheduler in system (eg UNIX, MS Windows)
  - short-term scheduler invoked frequently (milliseconds) ⇒ (must be fast)
- **Medium-term scheduler**: added in some OS in order to reduce the degree of multi-programming
  - remove process from memory (ready queue), store on disk, bring back in from disk to continue executionCPU-bound
- **Long-term scheduler** (or job scheduler): selects which process should be brought into the ready queue
  - invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - controls the *degree of multiprogramming*
  - degree = number of processes in memory
  - stable degree: average # of process creations == average # of process departures
  - thus, invoked only when a process leaves the system
- Processes can be either:
  - **I/O bound process**: spends more time doing I/O than computations, many short CPU bursts. *ready* queue is almost always empty if all processes are I/O bound
  - **CPU-bound process**: spends more time doing computations; few very long CPU bursts. *I/O* queue is almost always empty if all processes are CPU-bound

Context Switch

When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**

- **context** of a process represented in the PCB
  - = values of CPU registers, process state, memory management information, etc…
  - **save/restore** contexts to/from PCBs when switching among processes
- context-switch time is *pure* overhead; system does no useful work while switching
  - both switched processes are idle during CPU switch
  - more complex OS and PCB, longer context switch time

Operations on Processes

- processes execute concurrently, are dynamically created/deleted

- OS must provide mechanisms for processes creation, termination, etc

Process Creation

- **parent** process creates **children** processes, which creats other processes, forming a **tree** of processes
- processes managed via **process identifier** (pid)
- Resource sharing options when a process creates a child process. Some design options:
  - parent and children share all resources
  - children share subset of parents resources
  - parent and child share no resources
- Execution options when a process creates a new process
  - parent and children execute concurrently
  - parent waits until some or all of its chilren terminate
- **Address-space options** when a process creates a new process
  - child is duplicate of parent; it has same program and data as its parent
  - child has a *new* program loaded into it

UNIX examples

- `fork()` system-call creates new process; copy of address-space of parent
  - both parent and child execute concurrently after fork
  - child: same program as parent but fork() returns 0 to child, child_pid to parent
- `exec()` sys-call used after a fork to replace process' memory space w/ a new program and start execution

Process Termination

- process asks OS to delete it using `exit()` system call
  - returns status data from child to parent (via `wait()` sys-call)
  - process' resources are deallocated by OS
- parent may terminate execution of (some) children processes using the `abort()` sys-call (parent needs to know PID of child)
  - child has exceeded allocated resources (parent must inspect state of its children)
- parent is exiting, OS does not allow child to continue if its parent terminates
  - **cascading termination**: all children, grandchildren, etc are terminated. initiated by OS
- parent process may wait for termination of a child by using `wait()` sys-call. `wait` returns status information and the pid of the terminated process
  - `pid = wait(&status);`
  - in UNIX, terminate process using `exit(1)` w/ status parameter
  - `wait()` passed as a parameter allowing parent to obtain exit status of child

Zombie: if parent has not yet invoked `wait()` but child process has terminated
Orphan if parent has terminated without invoking `wait()`, child process is alive

Interprocess Communication (IPC)

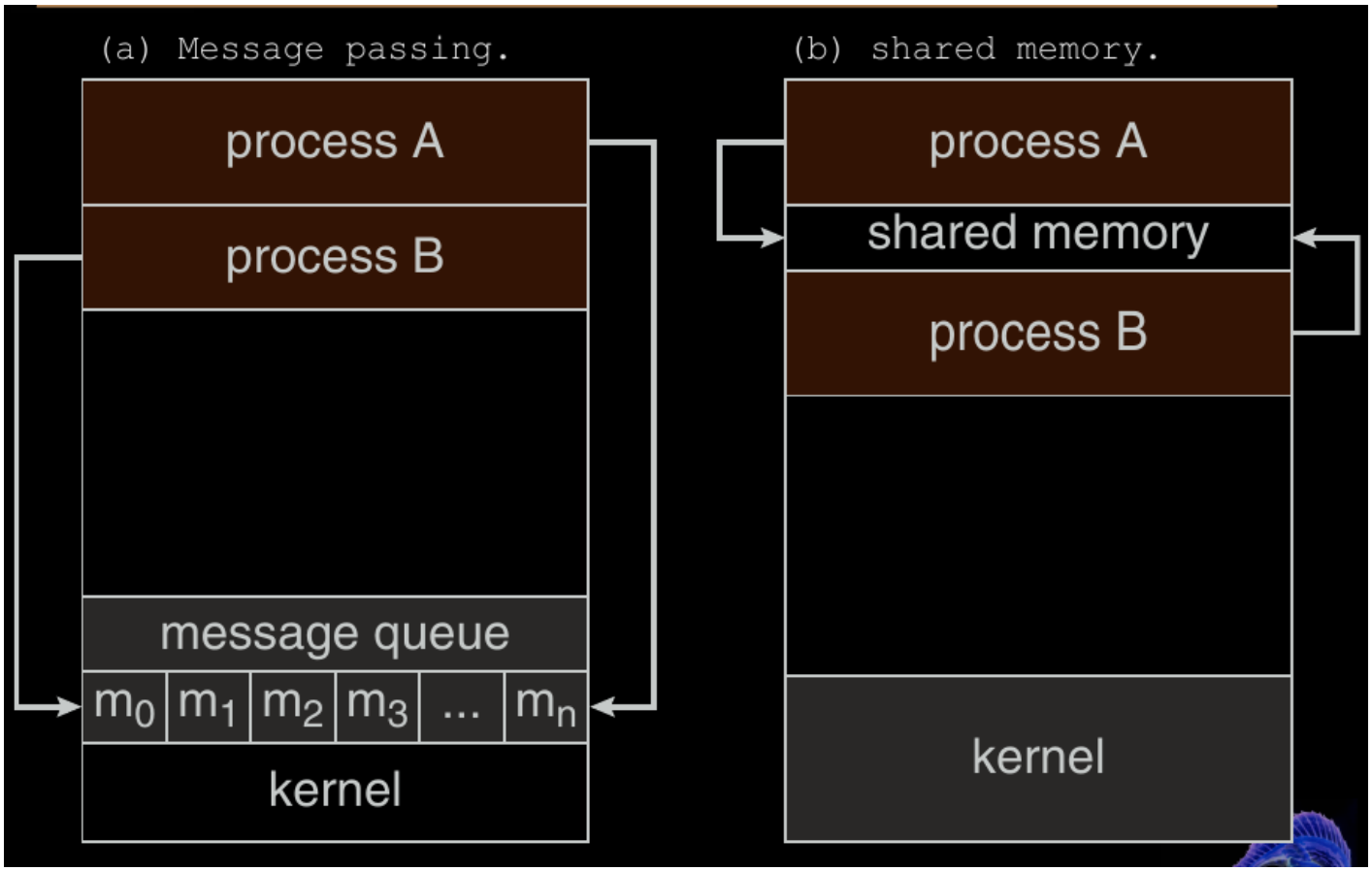
- processes w/in a system may be *independent* or *cooperating*
  - independent process cannot affect/be affected by execution of another process
  - cooperating process can affect or be affected by other processes, including shared data
  - cooperating processes need *interprocess communication* (IPC) mechanism to exchange data and information

Advantages of cooperating processes:

- information sharing: many users sharing the same file
- computation speedup: in multi-core systems
- modularity: recall communication system programs in CH2
- convenience: same user working on many tasks at the same time

Two models of IPC:

- **Shared memory**: faster than MP; uses sys-calls only to establish shared-memory regions
- **Message passing**: useful for small data exchanges, easier to implement in distributed systems. Performs better than SM on multi-core systems; cache coherency issues with SM



IPC - Shared Memory

An area of memory shared among the processes that wish to communicate

- SM resides in the address space of the process that creates the SM
  - any cooperating process must also attach the SM in their address space
- normally, OS prevents a process from accessing another process' memory
  - process cooperating via SM must agree to remove this restriction

Communication is under the control of the users processes, not the OS

- application programmer *explicitly* writes the code for sharing memory
- processes ensure that they not write to the same location simultaneously

Major issues is to provide mechanism that will allow the user processes to sync their actions when they accessed shared memory

- solution to *producer-consumer* problem

Shared-Memory Systems

Producer-Consumer Problem: paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process

Solution: producer and consumer processes **share a buffer** (shared-memory)

- synchronization: consumer should not consume data not yet produced

- two types of buffers:
  - unbounded-buffer: places no practical limit on the size of the buffer
  - bounded-buffer: assumes that there is a fixed buffer size

Shared-Memory: Bounded Buffer

A shared memory solution which can only use `BUFFER_SIZE-1` elements. It is `empty` if `in==out` and full if `((in + 1) % BUFFER_SIZE) == out`

```
#define BUFFER_SIZE 10
typedef struct{
    ...
} item;     // item to be produced/consumed

item buffer[BUFFER_SIZE];   // shared buffer
int in = 0;     // producer produces an item into in
int out = 0;        // consumer consumes an item from out
```

Shared-Memory: Bounded-Buffer - Producer

```
item next_produced; // stores new item to be produced
...
while(true){
    // produce an item in next_produced
    while (((in + 1) % BUFFER_SIZE) == out)
        ;   // do nothing while buffer is full

    buffer[in] = next_produced; // item is produced
    in = (in + 1) % BUFFER_SIZE;    // update in pointer
}
```

Shared-Memory: Bounded Buffer - Consumer

```
item next_consumed; // stores item to be consumed
...
while(true){
    while(in == out)
        ;   // do nothing while buffer is empty

    next_consumed = buffer[out];    // item is consumed
    out = (out + 1) % BUFFER_SIZE;  // update out pointer
    // consume the item in next_consumed
}
```

Message-Passing Systems

Mechanism for processes to communicate and to synchronize their actions without sharing any address space! Useful when communicating processes are in different computers

IPC facility provides two operations: - **communication link** must exist between communicating processes, then - `send(message)` - `receive(message)` - `message` size is either fixed or variable sized

If processes P and Q wish to communicate, they need to:

- establish a **communication link** between them
- exchange messages via `send/receive`

Implementation of Communication Link

- **Physical**: shared memory (not the same as logical SM), hardware bus, network
- **Logical**:
  - direct or indirect communication
  - synchronous or asynchronous communication
  - automatic or explicit buffering

MP: Direct Communication

- `send(P, message)`: send a message to process P
- `receive(Q, message)`: receive a message from process Q

Properties of direct communication link:

- links are established automatically, simply by naming the process
- 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

Direct communication schemes:

- **symmetric**: both sender, receiver must name the other to communicate
- **asymmetric**: only the sender names the recipient
  - `send(P, message)`: send a message to process P
  - `receive(id, message)`: receive a message from any process id

Cons: must explicitly state all process identifiers

MP: Indirect Communication

Messages are sent/received from mailboxes (also referred to as ports), each with a unique id. Processes can communicate if they share a mailbox

Properties of indirect communication link:

- link established only if process share a common mailbox
- link may be associated with many processes
- each pair of processes may share several communication links
- link may be uni/bi-directional

The OS provides operations allowing a process to:

- **CREATE** a new mailbox M (also called port)
  - the **owner** is the process that creates the mailbox M and is part of its address space
  - the owner can only receive messages through this mailbox M
- **send/receive** messages through mailbox
  - a **user** is the process which can only send messages to this mailbox M
- **DESTROY** a mailbox: mailbox disappears when its owner process terminates
  - user process must be notified that the mailbox no longer exists

The OS may also own a mailbox, independent and not attached to any process
Primitives are defined as:
`send(A, message)`: send a message to mailbox A
`receive(A, message)`: receive a message from mailbox A

MP: Synchronization

There are different options for implementing the `send()` and `receive()` primitives. Message passing may be either blocking or non-blocking. if both `send()` and `receive` are blocking, we have a *rendezvous*.

Blocking is considered *synchronous*

- **blocking send**: the sender is blocked until the msg is delivered
- **blocking receive**: the receiver is blocked until a message is available

Non-blocking is considered *asynchronous*

- **non-blocking send**: sender sends the msg and continues its tasks
- **non-blocking receive**: receiver retrieves:
  - valid message, or
  - null message

MP: Buffering

Queue of messages is attached to the communication link. It's implemented in one of three ways:

1. **Zero capacity**: no msgs are queued on a link
   - no buffering; no msg can be waiting in the link
   - sender must wait for receiver (rendezvous) to receive message
2. **Bounded capacity**: queue has a finite length of n messages
   - sender must wait if link is full, if not then
   - msgs are placed on the buffer w/o waiting fore receiver to receive
3. **Unbounded capacity**: infinite length
   - sender never waits