Midterm Exam 1 Review
Computer System Structure

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources
    - CPU, memory, I/O devices
  - Operating system
    - Controls and coordinates use of hardware among various applications and users
  - Application programs – define the ways in which the system resources are used to solve the computing problems of the users
    - Word processors, compilers, web browsers, database systems, video games
  - Users
    - People, machines, other computers
Four Components of a Computer System

- User 1
- User 2
- User 3
- ... User n

- Compiler
- Assembler
- Text Editor
- ... Database System

System and Application Programs

Operating System

Computer Hardware
Operating System Definition

- OS is a **resource allocator**
  - Manages all resources
  - Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  - Controls execution of programs to prevent errors and improper use of the computer
Computer System Organization

- Computer-system operation
  - One or more CPUs, device controllers connect through common bus providing access to shared memory
  - Concurrent execution of CPUs and devices competing for memory cycles. A memory controller synchronizes access to the memory.
Common Functions of Interrupts

- Interrupt transfers control to the interrupt service routine generally, through the interrupt vector, which contains the addresses of all the service routines.
- Interrupt architecture must save the address of the interrupted instruction.
- A trap or exception is a software-generated interrupt caused either by an error or a user request.
- An operating system is interrupt driven.
Interrupt Timeline

- CPU
  - User process executing
  - I/O interrupt processing

- I/O device
  - Idle
  - Transferring

- I/O request
- Transfer done
- I/O request
- Transfer done
Storage Structure

- Main memory – only large storage media that the CPU can access directly
  - Random access
  - Typically volatile
- Secondary storage – extension of main memory that provides large nonvolatile storage capacity
- Hard disks – rigid metal or glass platters covered with magnetic recording material
  - Disk surface is logically divided into tracks, which are subdivided into sectors
  - The disk controller determines the logical interaction between the device and the computer
- Solid-state disks – faster than hard disks, nonvolatile
  - Various technologies
  - Becoming more popular
Storage Hierarchy

- Storage systems organized in hierarchy
  - Speed
  - Cost
  - Volatility
- Caching – copying information into faster storage system; main memory can be viewed as a cache for secondary storage
- Device Driver for each device controller to manage I/O
  - Provides uniform interface between controller and kernel
Storage-Device Hierarchy

- registers
- cache
- main memory
- solid-state disk
- hard disk
- optical disk
- magnetic tapes
Caching

- Important principle, performed at many levels in a computer (in hardware, operating system, software)
- Information in use copied from slower to faster storage temporarily
- Faster storage (cache) checked first to determine if information is there
  - If it is, information used directly from the cache (fast)
  - If not, data copied to cache and used there
- Cache smaller than storage being cached
  - Cache management important design problem
  - Cache size and replacement policy
Direct Memory Access Structure

- Used for high-speed I/O devices able to transmit information at close to memory speeds
- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention
- Only one interrupt is generated per block, rather than the one interrupt per byte
A von Neumann architecture
Most systems use a single general-purpose processor

- Most systems have special-purpose processors as well, like device controllers

**Multiprocessors** systems growing in use and importance

- Also known as parallel systems, tightly-coupled systems

Advantages include:

1. Increased throughput
2. Economy of scale
3. Increased reliability – graceful degradation or fault tolerance

Two types:

1. **Asymmetric Multiprocessing** – each processor is assigned a specific task: boss and worker processors.
2. **Symmetric Multiprocessing** – each processor performs all tasks
Symmetric Multiprocessing Architecture
A Dual-Core Design

- Multi-chip and **multicore**
- Multicore system is more efficient than multi-chip system
  - On-chip communication is faster than between-chip communication.
Clustered Systems

- Like multiprocessor systems, but multiple systems working together
  - Usually sharing storage via a storage-area network (SAN)
  - Provides a high-availability service which survives failures
    - Asymmetric clustering has one machine in hot-standby mode
    - Symmetric clustering has multiple nodes running applications, monitoring each other
  - Some clusters are for high-performance computing (HPC)
    - Applications must be written to use parallelization
Operating System Structure

- **Multiprogramming (Batch system)** needed for efficiency
  - Single user cannot keep CPU and I/O devices busy at all times
  - Multiprogramming organizes jobs (code and data) so CPU always has one to execute
  - A subset of total jobs in system is kept in memory
  - One job selected and run via **job scheduling**
  - When it has to wait (for I/O for example), OS switches to another job

- **Timesharing (multitasking)** is logical extension in which CPU switches jobs so frequently that users can interact with each job while it is running, creating **interactive** computing
  - **Response time** should be < 1 second
  - Each user has at least one program executing in memory ⇒ **process**
  - If several jobs ready to run at the same time ⇒ **CPU scheduling**
  - If processes don’t fit in memory, **swapping** moves them in and out to run
  - **Virtual memory** allows execution of processes not completely in memory
Dual-mode operation allows OS to protect itself and other system components

- **User mode** and **kernel mode**
- **Mode bit** provided by hardware (e.g., CS register in CPU)
  - Provides ability to distinguish when system is running user code or kernel code
  - Some instructions designated as privileged, only executable in kernel mode
  - System call changes mode to kernel, return from call resets it to user

![Diagram showing user process executing, calls system call, return from system call, kernel with mode bit 0 for trap and mode bit 1 for execute system call, dual mode operation concept.]
Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
  - **User interface** - Almost all operating systems have a user interface (UI).
    - Varies between *Command-Line (CLI)*, *Graphics User Interface (GUI)*, *Batch*
  - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
  - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
One set of operating-system services provides functions that are helpful to the user (Cont.):

- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)

- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing
  - Debugging facilities can greatly enhance the user’s and programmer’s abilities to efficiently use the system
Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.

- **Accounting** - To keep track of which users use how much and what kinds of computer resources

- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
A View of Operating System Services

user and other system programs

<table>
<thead>
<tr>
<th>GUI</th>
<th>batch</th>
<th>command line</th>
</tr>
</thead>
<tbody>
<tr>
<td>user interfaces</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

system calls

<table>
<thead>
<tr>
<th>program execution</th>
<th>I/O operations</th>
<th>file systems</th>
<th>communication</th>
<th>resource allocation</th>
<th>accounting</th>
</tr>
</thead>
<tbody>
<tr>
<td>error detection</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>services</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>protection and security</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

operating system

hardware
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level Application Programming Interface (API) rather than direct system call use
- Three most common APIs are Windows API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
Example of System Calls

- System call sequence to copy the contents of one file to another file

Example System Call Sequence
- Acquire input file name
- Write prompt to screen
- Accept input
- Acquire output file name
- Write prompt to screen
- Accept input
- Open the input file
  - if file doesn't exist, abort
- Create output file
  - if file exists, abort
- Loop
  - Read from input file
  - Write to output file
  - Until read fails
- Close output file
- Write completion message to screen
- Terminate normally
System Call Implementation

- Typically, a number associated with each system call
  - **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
  - Just needs to obey API and understand what OS will do as a result call
  - Most details of OS interface hidden from programmer by API
    - Managed by run-time support library (set of functions built into libraries included with compiler)
Design and Implementation of OS not “solvable”, but some approaches have proven successful.

Internal structure of different Operating Systems can vary widely.

Start the design by defining goals and specifications.

Highest level: affected by choice of hardware, type of system.

The requirements can be divided into User and System goals.
  - User goals – operating system should be convenient to use, easy to learn, reliable, safe, and fast.
  - System goals – operating system should be easy to design, implement, and maintain, as well as flexible, reliable, error-free, and efficient.
Important principle to separate

**Policy**: What will be done?

**Mechanism**: How to do it?

Mechanisms determine how to do something, policies decide what will be done

The separation of policy from mechanism is a very important principle, it allows maximum flexibility if policy decisions are to be changed later (example – timer)

Specifying and designing an OS is highly creative task of software engineering
General-purpose OS is very large program

Various ways to structure ones
- Simple structure – MS-DOS
- More complex -- UNIX
- Layered – an abstraction
- Microkernel – Mach
Traditional UNIX System Structure

Beyond simple but not fully layered

- (the users)
  - shells and commands
  - compilers and interpreters
  - system libraries

- system-call interface to the kernel
  - signals terminal handling
  - character I/O system
  - terminal drivers
  - file system
  - swapping block I/O system
  - disk and tape drivers
  - CPU scheduling
  - page replacement
  - demand paging
  - virtual memory

- kernel interface to the hardware
  - terminal controllers
  - terminals
  - device controllers
  - disks and tapes
  - memory controllers
  - physical memory
Microkernel System Structure

- Application Program
- File System
- Device Driver

- Interprocess Communication
- memory management
- CPU scheduling

- microkernel
- hardware

User mode
Kernel mode
Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms job and process almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- Multiple parts
  - The program code, also called text section
  - Current activity including program counter, processor registers
  - Stack containing temporary data
    - Function parameters, return addresses, local variables
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
Process Concept (Cont.)

- Program is **passive** entity stored on disk (**executable file**), process is **active**
  - Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, etc
- One program can be several processes
  - Consider multiple users executing the same program
Process State

- As a process executes, it changes **state**
  - **new**: The process is being created
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **ready**: The process is waiting to be assigned to a processor
  - **terminated**: The process has finished execution
Diagram of Process State

1.36 Silberschatz, Galvin and Gagne ©2013
Process Control Block (PCB)

Information associated with each process
(also called **task control block**)

- Process state – running, waiting, etc
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information – priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files
CPU Switch From Process to Process

process $P_0$  operating system  process $P_1$

executing  interrupt or system call  idle

interrupt or system call

save state into PCB$_0$

reload state from PCB$_1$

save state into PCB$_1$

executing

reload state from PCB$_0$
Process Scheduling

- Maximize CPU use, quickly switch processes onto CPU for time sharing
- **Process scheduler** selects among available processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues
Ready Queue And Various I/O Device Queues
Queueing diagram represents queues, resources, flows
Schedulers

- **Short-term scheduler** (or **CPU scheduler**) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) ⇒ (must be fast)
- **Long-term scheduler** (or **job scheduler**) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) ⇒ (may be slow)
  - The long-term scheduler controls the **degree of multiprogramming**
- Processes can be described as either:
  - **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
  - **CPU-bound process** – spends more time doing computations; few very long CPU bursts
- Long-term scheduler strives for good **process mix**
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
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB → the longer the context switch
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU → multiple contexts loaded at once
Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next
**Process Creation**

- **Parent** process create **children** processes, which, in turn create other processes, forming a **tree** of processes.
- Generally, process identified and managed via a **process identifier (pid)**.
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent’s resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate
Process Creation (Cont.)

- **Address space**
  - Child duplicate of parent (has the same program as the parent)
  - Child has a program loaded into it

- **UNIX examples**
  - `fork()` system call creates new process. The new process consists of a copy of the address space of the original process.
  - `exec()` system call used after a `fork()` to replace the process’ memory space with a new program

```
fork()                                 exit()

parent → wait  → resumes

move itself off the ready queue until the termination of the child

child → exec()
```
The only difference is that the value of pid for the child process is zero, while that for the parent is the actual pid of the child process.
Process Termination

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)  
  - Process’ resources are deallocated by operating system
- Parent may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
Process Termination

- Some operating systems do not allow a child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - **cascading termination.** All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the `wait()` system call. The call returns status information and the pid of the terminated process
  
  ```
  pid = wait(&status);
  ```
- If no parent waiting (did not invoke `wait()`) process is a **zombie**
  - Once the parent calls `wait()`, the process identifier of the zombie process and its entry in the process table are released.
- If parent terminated without invoking `wait`, process is an **orphan**
  - Assigning the init process as the new parent, periodically invokes `wait()`
Processes within a system may be independent or cooperating.

Cooperating processes can affect or be affected by other processes, including sharing data.

Reasons for cooperating processes:
- Information sharing (shared files)
- Computation speedup (parallel subtasks)
- Modularity (system function divided into separate processes)
- Convenience

Cooperating processes need interprocess communication (IPC).

Two models of IPC:
- Shared memory
- Message passing
Communications Models

(a) Message passing. (b) shared memory.
An area of memory shared among the processes that wish to communicate

- Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

- 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.

- Synchronization is discussed in great details in Chapter 5.
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)`
- Particularly useful in a distributed environment
- The `message` size is either fixed or variable
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?
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
Race Condition

- `counter++` could be implemented as

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

- `counter--` could be implemented as

  ```plaintext
  register2 = counter
  register2 = register2 - 1
  counter = register2
  ```

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

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0:</td>
<td>producer execute <code>register1 = counter</code></td>
<td><code>register1 = 5</code></td>
</tr>
<tr>
<td>S1:</td>
<td>producer execute <code>register1 = register1 + 1</code></td>
<td><code>register1 = 6</code></td>
</tr>
<tr>
<td>S2:</td>
<td>consumer execute <code>register2 = counter</code></td>
<td><code>register2 = 5</code></td>
</tr>
<tr>
<td>S3:</td>
<td>consumer execute <code>register2 = register2 - 1</code></td>
<td><code>register2 = 4</code></td>
</tr>
<tr>
<td>S4:</td>
<td>producer execute <code>counter = register1</code></td>
<td><code>counter = 6</code></td>
</tr>
<tr>
<td>S5:</td>
<td>consumer execute <code>counter = register2</code></td>
<td><code>counter = 4</code></td>
</tr>
</tbody>
</table>
Critical Section Problem

- Consider system of \( n \) processes \( \{p_0, p_1, \ldots, p_{n-1}\} \)
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this problem
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**
Critical Section

- General structure of process $P_i$

```c
do {
    entry section
    critical section
    exit section
    remainder section
} while (true);
```
1. **Mutual Exclusion** - If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted:
   - Assume that each process executes at a nonzero speed.
   - No assumption concerning relative speed of the $n$ processes.
Peterson’s Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the load and store machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - int turn;
  - Boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!
Algorithm for Process \( P_i \)

do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
    critical section
    flag[i] = false;
    remainder section
} while (true);
Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
    - Either test memory word and set value
    - Or swap contents of two memory words
test_and_set Instruction

Definition:

```c
boolean test_and_set (boolean *target) {
    boolean rv = *target;
    *target = TRUE;
    return rv;
}
```

1. Executed atomically
2. Returns the original value of passed parameter
3. Set the new value of passed parameter to “TRUE”.
Solution using test_and_set()

- Shared Boolean variable lock, initialized to FALSE
- Solution:

  ```c
  do {
    while (test_and_set(&lock))
      ; /* do nothing */
    /* critical section */
    lock = false;
    /* remainder section */
  } while (true);
  ```
compare_and_swap Instruction

Definition:

```c
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;

    if (*value == expected)
        *value = new_value;
    return temp;
}
```

1. Executed atomically
2. Returns the original value of passed parameter “value”
3. Set the variable “value” the value of the passed parameter “new_value” but only if “value” ==“expected”. That is, the swap takes place only under this condition.
Solution using compare_and_swap

- Shared integer “lock” initialized to 0;
- Solution:
  ```
  do {
    while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
  /* critical section */
  lock = 0;
  /* remainder section */
  } while (true);
  ```
Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first acquire() a lock then release() the lock
  - Boolean variable indicating if lock is available or not
- Calls to acquire() and release() must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires busy waiting
  - This lock therefore called a spinlock
acquire() and release()

- **acquire()** {
  - while (!available)
    - ; /* busy wait */
  - available = false;
}

- **release()** {
  - available = true;
}

- do {
  - acquire lock
  - critical section
  - release lock
  - remainder section
} while (true);
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore $S$ – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()` and `signal()`
    - Originally called `P()` and `V()`
  - Definition of the `wait()` operation
    ```
    wait(S) {
        while (S <= 0)  
            ; // busy wait  
        S--;  
    }
    ```
  - Definition of the `signal()` operation
    ```
    signal(S) {  
        S++;  
    }
    ```
Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a *mutex lock*
- Can solve various synchronization problems
- Consider $P_1$ and $P_2$ that require $S_1$ to happen before $S_2$
  
  Create a semaphore “*synch*” initialized to 0
  
  **P1:**
  
  ```
  S_1;
  signal(synch);
  ```

  **P2:**
  
  ```
  wait(synch);
  S_2;
  ```

- Can implement a counting semaphore $S$ as a binary semaphore
Semaphore Implementation

- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time.
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section.
  - Could now have busy waiting in critical section implementation:
    - But implementation code is short.
    - Little busy waiting if critical section rarely occupied.
Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
  - value (of type integer)
  - pointer to next record in the list
- **typedef struct**{
  int value;
  struct process *list;
} semaphore;
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue
Implementation with no Busy waiting (Cont.)

```c
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}
```

```c
signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```
Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let $S$ and $Q$ be two semaphores initialized to 1

  \[
  P_0 \\
  \quad \text{wait}(S); \\
  \quad \text{wait}(Q); \\
  \quad \ldots \\
  \quad \text{signal}(S); \\
  \quad \text{signal}(Q);
  \]

  \[
  P_1 \\
  \quad \text{wait}(Q); \\
  \quad \text{wait}(S); \\
  \quad \ldots \\
  \quad \text{signal}(Q); \\
  \quad \text{signal}(S);
  \]

- **Starvation** – indefinite blocking
  - A process may never be removed from the semaphore queue in which it is suspended

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via priority-inheritance protocol
Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
Bounded-Buffer Problem

- \( n \) buffers, each can hold one item
- Semaphore `mutex` initialized to the value 1
- Semaphore `full` initialized to the value 0
- Semaphore `empty` initialized to the value \( n \)
The structure of the producer process

```c

do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```
Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```c
Do {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
    ...
} while (true);
```
Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do not perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore `rw_mutex` initialized to 1
  - Semaphore `mutex` initialized to 1
  - Integer `read_count` initialized to 0
The structure of a writer process

\[
\text{do } \{ \text{wait(rw_mutex);} \\
\quad \ldots \\
\quad \text{/* writing is performed */} \\
\quad \ldots \\
\quad \text{signal(rw_mutex);} \\
\} \text{ while (true);} 
\]
The structure of a reader process

```c
    do {
        wait(mutex);
        read_count++;  
        if (read_count == 1)
            wait(rw_mutex);
        signal(mutex);
        ...
        /* reading is performed */
        ...
        wait(mutex);
        read_count--;  
        if (read_count == 0)
            signal(rw_mutex);
        signal(rw_mutex);
        signal(mutex);
    } while (true);
```
Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1
Dining-Philosophers Problem Algorithm

- The structure of Philosopher $i$:

  ```c
  do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

    // eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

    // think

  } while (TRUE);
  ```

- What is the problem with this algorithm?
Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

```c
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { .... }

    procedure Pn (...) {......}

    Initialization code (...) { ... }
}
```
Condition Variables

- **condition** \( x, y \);
- Two operations are allowed on a condition variable:
  - \( x.wait() \) – a process that invokes the operation is suspended until \( x.signal() \)
  - \( x.signal() \) – resumes one of processes (if any) that invoked \( x.wait() \)
    - If no \( x.wait() \) on the variable, then it has no effect on the variable
Monitor with Condition Variables

- Shared data
  - Queues associated with $x$, $y$ conditions
  - Operations
  - Initialization code

- Entry queue
Monitor Solution to Dining Philosophers

monitor DiningPhilosophers
{
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test(((i + 4) % 5));
        test(((i + 1) % 5));
    }
}
Solution to Dining Philosophers (Cont.)

```c
void test (int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
```
Each philosopher $i$ invokes the operations `pickup()` and `putdown()` in the following sequence:

```
DiningPhilosophers.pickup(i);
EAT
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible