Processes

- Basic concept to build the OS, from old IBM mainframe OS to the most modern Windows
- Used to express the requirements to be met by an OS
  - Interleave the execution of multiple processes, to maximize CPU utilization while providing good response time
  - Allocate resources to processes using a policy while avoiding deadlocks
  - Support interprocess communications and user creation of processes to help structuring applications
- Background
  - Computer platform
    * Collection of hardware resources – CPU, memory, I/O modules, timers, storage devices
  - Computer applications
    * Developed to perform some task
    * Input, processing, output
  - Efficient to write applications for a given CPU
    * Common routines to access computer resources across platforms
    * CPU provides only limited support for multiprogramming; software manages sharing of CPU and other resources by multiple applications concurrently
    * Data and resources for multiple concurrent applications must be protected from other applications
- Process
  - Abstraction of a running program
  - Unit of work in the system
  - Split into two abstractions in modern OS
    * Resource ownership (traditional process view)
    * Stream of instruction execution (thread)
  - Pseudoparallelism, or interleaved instructions
  - A process is traced by listing the sequence of instructions that execute for that process
- Modeling sequential process/task
  - Program during execution
  - Program code
  - Current activity
  - Process stack
    * Function parameters
    * Return addresses
    * Temporary variables
  - Data section
    * Global variables
- Concurrent Processes
  - Multiprogramming
  - Interleaving of traces of different processes characterizes the behavior of the CPU
  - Physical resource sharing
    * Required due to limited hardware resources
- Logical resource sharing
  * Concurrent access to the same resource like files
- Computation speedup
  * Break each task into subtasks
  * Execute each subtask on separate processing element
- Modularity
  * Division of system functions into separate modules
- Convenience
  * Perform a number of tasks in parallel
- Real-time requirements for I/O

- Process Hierarchies
  - Parent-child relationship
  - `fork(2)` call in Unix
  - In older non-multitasking systems such as MS-DOS, parent suspends itself and lets the child execute

- Process states
  - A two-state process model
    * Simplest possible model
    * A process is either executing (running state) or it is idle (not-running state)
    * For a new process, the OS creates a new process control block and brings that process into memory in a not-running state
  - A five-state model
    * Running
    * Ready – Not running, waiting for the CPU
    * Blocked – Wait on an event (other than CPU)
    * Two other states complete the five-state model – New and Exit
      - A process being created can be said to be in state New; it will be in state Ready after it has been created
      - A process being terminated can be said to be in state Exit

- Above model suffices for most of the discussion on process management in operating systems; however, it is limited in the sense that the system screeches to a halt (even in the model) if all the processes are resident in memory and they all are waiting for some event to happen
- Create a new state Suspend to keep track of blocked processes that have been temporarily kicked out of memory to make room for new processes to come in
- The state transition diagram in the revised model is
- Which process to grant the CPU when the current process is swapped out?
  * Preference for a previously suspended process over a new process to avoid increasing the total load on the system
  * Suspended processes are actually blocked at the time of suspension and making them ready will just change their state back to blocked
  * Decide whether the process is blocked on an event (suspended or not) or whether the process has been swapped out (suspended or not)

- The new state transition diagram is

![State Transition Diagram](image)

- Process sleep state
  * A process can put itself to sleep while waiting for an event
    - Instead of constantly polling for input from keyboard, a shell puts itself to sleep
  * Process sleeps on a particular wait channel (WCHAN)
  * When the event associated with WCHAN occurs, every process waiting on that WCHAN is woken up
  * The awakened processes check to see if the signal was meant for them
    - Consider a set of processes waiting for data from the disk
    - Once data becomes available, processes check whether the data is ready for them
  * If the signal is not for the processes, they put themselves to sleep on the same WCHAN

**Process control**

- Modes of execution
  - OS execution vs user process execution
  - OS may prevent execution of some instructions in user mode and allow them to be executed only in privileged mode (also called kernel mode, system mode, or control mode)
    * Read/write a control register, such as PSW
    * Primitive I/O and memory management
  - The two modes protect the OS data structures from interference by user code
  - Kernel mode provides full control of the system that may not be needed for user programs
  - The kernel mode can be entered by setting a bit in the PSW
  - The system can enter privileged mode as a result of a request from user code and returns to user mode after completing the request

- Implementation of processes
  - Process table
    * One entry for each process
Interprocess Communication

- program counter
- stack pointer
- memory allocation
- open files
- accounting and scheduling information

- **Interrupt vector**
  - Contains address of *interrupt service procedure*
    - saves all registers in the process table entry
    - services the interrupt

- **Process creation**
  - Assign a unique process identifier to the new process; add this process to the system process table that contains one entry for each process
  - Allocate space for all elements of process image – space for code, data, and user stack; values can be set by default or based on parameters entered at job creation time
  - Allocation of resources (CPU time, memory, files) – use either of the following policies
    - New process obtains resources directly from the OS
    - New process constrained to share resources from a subset of the parent process
  - Build the data structures that are needed to manage the process, especially process control block
  - When is a process created? – job submission, login, application such as printing
  - Initialization data (input)
  - Process execution
    - Parent continues to execute concurrently with its children
    - Parent waits until all its children have terminated

- **Process switching**
  - Interrupt a running process and assign control to a different process
  - Difference between process switching and mode switching
  - When to switch processes
    - Any time when the OS has control of the system
    - OS can acquire control by
      - Interrupt – asynchronous external event; not dependent on instructions; clock interrupt
      - Trap – Exception handling; associated with current instruction execution
      - Supervisor call – Explicit call to OS

- **Processes in Unix**
  - Identified by a unique integer – *process identifier*
  - Created by the `fork(2)` system call
    - Copy the three segments (instructions, user-data, and system-data) without initialization from a program
    - New process is the copy of the address space of the original process to allow easy communication of the parent process with its child
    - Both processes continue execution at the instruction after the `fork`
    - Return code for the `fork` is
      - zero for the child process
      - process id of the child for the parent process
    - Implementation of `fork(2)` in Unix
- Both parent’s data and code need to be duplicated in the copies assigned to child
- Not very efficient to make copies since most of the time, *fork(2)* may be followed by an *exec* call
- Hardware paging allows kernels to use Copy-On-Write approach to defer page duplication until the last possible moment; that is, when parent or child need to write into the page

- Use *exec(2)* system call after *fork* to replace the child process’s memory space with a new program (binary file)
  * Overlay the image of a program onto the running process
  * Reinitialize a process from a designated program
  * Program changes while the process remains

- *exit(2)* system call
  * Finish executing a process
  * Kernel releases resources owned by the process
  * Sends a SIGCHLD signal to parent

- *wait(2)* system call
  * Wait for child process to stop or terminate
  * Synchronize process execution with the *exit* of a previously *forked* process

- *signal(3)* library function
  * Control process response to extraordinary events
  * The complete family of *signal* functions (see man page; section 7) provides for simplified signal management for application processes

- Daemons or kernel threads
  * Privileged processes in Unix
  * Run in kernel mode in kernel address space
  * Background processes to do useful work on behalf of the user
    * Just sit in the machine, doing one or the other thing
  * Differ from normal processes in the sense that daemons do not have a *stdin* or *stdout*, and sleep most of the time
    * Communication with humans achieved via logs
  * Created during system startup and remain alive until the system is shut down
  * Common daemons are
    * *update* to synchronize the file system with its image in kernel memory
    * *cron* for general purpose task scheduling
    * *lpd* or *lpsched* as a line printer daemon to pick up files scheduled for printing and distributing them to the printers
    * *init* – the boss of it all
    * *swapper* to handle kernel requests to swap pages of memory to/from disk

- Zombies
  * Processes waiting to send a message to parent so that they can die
  * *init* routinely issues *wait(2)* system call whose side effect is to get rid of all orphaned zombies

- Wait queues
  * Represent sleeping processes to be woken up by kernel when a condition becomes true
  * Used for interrupt handling, process synchronization, and timing
  * Disk operation to terminate, a system resource to be released, or a fixed interval of time to elapse
  * A process waiting for a specific event is put into the corresponding wait queue
  * Modified by interrupt handlers and major kernel functions
    * Must be protected from concurrent access
Interprocess Communication

- Synchronization achieved by a spin lock in the wait queue head

- **MS-DOS Processes**
  - Created by a system call to load a specified binary file into memory and execute it
  - Parent is suspended and waits for child to finish execution

- **Process termination**
  - **Normal termination**
    * Process terminates when it executes its last statement
    * Upon termination, the OS deletes the process
    * Process may return data (output) to its parent
  - **Abnormal termination**
    * Process terminates by executing the library function `abort(3C)`
    * All the file streams are closed and other housekeeping performed as defined in the signal handler
  - **Termination by another process**
    * Termination by the system call `kill(2)` with the signal `SIGKILL`
    * Usually terminated only by the parent of the process because
      - child may exceed the usage of its allocated resources
      - task assigned to the child is no longer required
  - **Cascading termination**
    * Upon termination of parent process
    * Initiated by the OS

- **Process removal**
  - A process can query the kernel to get the execution state of its children
  - A process can create a child process to perform a specific task and `wait` to check whether the child has terminated
  - The termination code of child tells the parent process whether the task is completed successfully
  - Because of these design choices, Unix kernel is not allowed to discard data in a PCB right after the process terminates; it has to wait till the parent issues a `wait` that refers to the terminated process
  - **EXIT_ZOMBIE** state: process is technically dead but its descriptor must be saved until the parent has received notification
  - If the parent is dead, the orphan becomes a child of `init` who destroys zombies by issuing a `wait`

**Process states in Linux**

- Described by six flags and are mutually exclusive

- **TASK_RUNNING**

- **TASK_INTERRUPTIBLE**
  - Process is suspended, waiting for a condition such as hardware interrupt, a system resource, or delivery of a signal
  - Changes to **TASK_RUNNING** when that happens

- **TASK_UNINTERRUPTIBLE**
  - Delivering a signal to sleeping process leaves it state unchanged
  - Process opens a device file and corresponding device driver starts to probe for corresponding hardware device
Interprocess Communication

* Device driver cannot be interrupted until the probing is complete, or hardware device can be left in an unpredictable state

- TASK_STOPPED
  - Process execution stopped
  - Result of receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal

- TASK_TRACED
  - Process stopped by a debugger

- EXIT_ZOMBIE
  - Process finished execution but parent has not yet issued a wait system call

- EXIT_DEAD
  - Process being removed after the parent has just issued a wait system call
  - Changing state from EXIT_ZOMBIE to EXIT_DEAD avoids race conditions due to other threads of execution that execute wait()-like calls on the same process

Principles of concurrency

- Management of processes and threads is the central theme in OS design

  Multiprogramming: Management of multiple processes within a uniprocessor system

  Multitasking: Management of multiple processes by interleaving their execution on a uniprocessor system, possibly by scheduling

  Multiprocessing: Management of multiple processes within a multiprocessor

  Distributed processing: Management of multiple processes executing on multiple distributed systems; Clustering

- Concurrency
  - Encompasses a host of design issues, including communication among processes, sharing and competing for resources, synchronization of activities of multiple processes, and allocation of CPU time to processes
  - Concurrency arises with
    * Multiple applications – Processing time shared among a number of active applications
    * Structured applications – A single application effectively programmed as a set of concurrent modules
    * OS structure – OS implemented as a set of processes or threads

- cobegin/coend
  - Also known as parbegin/parend
  - Explicitly specify a set of program segments to be executed concurrently

```plaintext
  cobegin
    p_1;
    p_2;
    ...
    p_n;
  coend;
```

\[(a + b) \times (c + d) - (e/f)\]
cobegin
t_1 = a + b;
t_2 = c + d;
t_3 = e / f;
coend
t_4 = t_1 * t_2;
t_5 = t_4 - t_3;

• fork, join, and quit Primitives
  – More general than cobegin/coend
  – fork x
    * Creates a new process q when executed by process p
    * Starts execution of process q at instruction labeled x
    * Process p executes at the instruction following the fork
  – quit
    * Terminates the process that executes this command
  – join t, y
    * Provides an indivisible instruction
    * Provides the equivalent of test-and-set instruction in a concurrent language
      if ( ! --t ) goto y;

  – Program segment with new primitives
    m = 3;
fork p2;
fork p3;
p1 : t1 = a + b; join m, p4; quit;
p2 : t2 = c + d; join m, p4; quit;
p3 : t3 = e / f; join m, p4; quit;
p4 : t4 = t1 * t2;
t5 = t4 - t3;

• Modern parallel programming language (TBB)
  – Serial loop
    for ( int i = 0; i < 10000; i++ )
a[i] = f(i) + g(i);
  – Parallel loop in Intel TBB (threading building blocks)
    tbb::parallel_for ( 0, 10000, [&](int i) { a[i] = f(i) + g(i); } );
    – parallel_for creates tasks that apply the loop body to each element in range
    – The & in the lambda expression indicates that variable a should be captured by reference

Process Control Subsystem in Unix
  • Significant part of the Unix kernel (along with the file subsystem)
  • Contains three modules
    – Interprocess communication
    – Scheduler
    – Memory management
Interprocess Communication

- Race conditions
  - A race condition occurs when two processes (or threads) access the same variable/resource without doing any synchronization
  - One process is doing a coordinated update of several variables
  - The second process observing one or more of those variables will see inconsistent results
  - Final outcome dependent on the precise timing of two processes
  - Example
    * One process is changing the balance in a bank account while another is simultaneously observing the account balance and the last activity date
    * Now, consider the scenario where the process changing the balance gets interrupted after updating the last activity date but before updating the balance
    * If the other process reads the data at this point, it does not get accurate information (either in the current or past time)

- OS concerns
  - Keeping track of different processes through PCBs
  - Allocating and deallocating various resources for active processes, including CPU time, memory, files, and I/O devices
  -Protecting data and physical resources of each process against unintended or deliberate interference by other processes
  - Functioning of a process and its I/O which proceed at different speeds, relative to the speed of other concurrent processes

Critical Section Problem

- Section of code that modifies some memory/file/table while assuming its exclusive control
- Mutually exclusive execution in time
- Template for each process that involves critical section

```c
    do
    { ... / * Entry section; */
        critical_section(); / * Assumed to be present */
        ... / * Exit section */
        remainder_section(); / * Assumed to be present */
    } while ( 1 );
```

You are to fill in the gaps specified by . . . for entry and exit sections in this template and test the resulting program for compliance with the protocol specified next

- Design of a protocol to be used by the processes to cooperate with following constraints
  - Mutual Exclusion – If process $p_i$ is executing in its critical section, then no other processes can be executing in their critical sections.
  - Progress – If no process is executing in its critical section, the selection of a process that will be allowed to enter its critical section cannot be postponed indefinitely.
- Bounded Waiting – There must exist a bound 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.

- Assumptions
  - No assumption about the hardware instructions
  - No assumption about the number of processors supported
  - Basic machine language instructions executed atomically

- Disabling interrupts
  - Brute-force approach
  - Not proper to give users the power to disable interrupts
    - User may not enable interrupts after being done
    - Multiple CPU configuration
  - In current systems, interrupts must be disabled inside some critical kernel regions
    - Critical regions must be limited because kernel and interrupt handlers should be able to run most of the time to take care of any event

- Lock variables
  - Share a variable that is set when a process is in its critical section

- Strict alternation

  extern int turn;    /* Shared variable between both processes */

  void process ( const int me ) /* me can be 0 or 1 */
  {
      int other = 1 - me;
      do
      {
          while ( turn != me ) /* do nothing */ ;
          critical_section();
          turn = other;
          remainder_section();
      } while ( 1 );
  }

  - Does not satisfy progress requirement
  - Does not keep sufficient information about the state of each process

- Use of a flag

  extern int flag[2];    /* Shared variable; one for each process */

  void process ( const int me ) /* me can be 0 or 1 */
  {
      int other = 1 - me;
      do
      {
          flag[me] = 1;    /* true */
          while ( flag[other] );
          critical_section();
          flag[me] = 0;    /* false */
remainder_section();
} while ( 1 );

– Satisfies the mutual exclusion requirement
– Does not satisfy the progress requirement

\[
\begin{align*}
\text{Time } T_0 & \quad p_0 \text{ sets flag[0] to true} \\
\text{Time } T_1 & \quad p_1 \text{ sets flag[1] to true}
\end{align*}
\]

Processes \( p_0 \) and \( p_1 \) loop forever in their respective while statements
– Critically dependent on the exact timing of two processes
– Switch the order of instructions in entry section
  * No mutual exclusion

• Peterson’s solution
  – Combines the key ideas from the two earlier solutions

\[
\begin{align*}
\text{extern int flag[2];} & \quad / * \text{Shared variables} */ \\
\text{extern int turn;} & \quad / * \text{Shared variable} */ \\
\text{void process ( const int me )} & \quad / * \text{me can be 0 or 1} */ \\
\{ \\
\text{\quad int other = 1 - me;} \\
\text{\quad do} \\
\text{\quad \{} \\
\text{\quad \quad / * Entry section */} \\
\text{\quad flag[me] = true;} & \quad / * \text{Raise my flag} */ \\
\text{\quad turn = other;} & \quad / * \text{Cede turn to other process} */ \\
\text{\quad while ( flag[other] && turn == other ) ;} \\
\text{\quad critical_section();} \\
\text{\quad / * Exit section */} \\
\text{\quad flag[me] = false;} \\
\text{\quad remainder_section();} \\
\text{\quad } \\
\text{\quad } \\
\text{\quad \} \text{ while ( 1 );} \\
\text{\} \\
\}
\end{align*}
\]

• Multiple Process Solution – Solution 4
  – The array \( \text{flag} \) can take one of the three values \((\text{idle, want-in, in-cs})\)

\[
\begin{align*}
\text{enum state \{ idle, want_in, in_cs \};} \\
\text{extern int turn;} \\
\text{extern state flag[n];} & \quad / * \text{Flag corresponding to each process in shared memory} */ \\
\text{process ( const int i )} \\
\{ \\
\text{\quad int j;} & \quad / * \text{Local to each process} */ \\
\text{\quad do} \\
\text{\quad \{} \\
\end{align*}
\]
do
{
    flag[i] = want_in;     // Raise my flag
    j = turn;             // Set local variable
    while ( j != i )
        j = ( flag[j] != idle ) ? turn : ( j + 1 ) % n;
    // Declare intention to enter critical section
    flag[i] = in_cs;
    // Check that no one else is in critical section
    for ( j = 0; j < n; j++ )
        if ( ( j != i ) && ( flag[j] == in_cs ) )
            break;
    } while ( j < n ) || ( turn != i && flag[turn] != idle );
    // Assign turn to self and enter critical section
    turn = i;
    critical_section();
    // Exit section
    j = (turn + 1) % n;
    while (flag[j] == idle)
        j = (j + 1) % n;
    // Assign turn to next waiting process; change own flag to idle
    turn = j;
    flag[i] = idle;
    remainder_section();
} while ( 1 );

- \( p_i \) enters the critical section only if \( \text{flag}[j] \neq \text{in-cs} \) for all \( j \neq i \).
- \( \text{turn} \) can be modified only upon entry to and exit from the critical section. The first contending process enters its critical section.
- Upon exit, the successor process is designated to be the one following the current process.
- Mutual Exclusion
  * \( p_i \) enters the critical section only if \( \text{flag}[j] \neq \text{in-cs} \) for all \( j \neq i \).
  * Only \( p_i \) can set \( \text{flag}[i] = \text{in-cs} \).
  * \( p_i \) inspects \( \text{flag}[j] \) only while \( \text{flag}[i] = \text{in-cs} \).
- Progress
  * \( \text{turn} \) can be modified only upon entry to and exit from the critical section.
  * No process is executing or leaving its critical section \( \Rightarrow \text{turn} \) remains constant.
  * First contending process in the cyclic ordering \( (\text{turn}, \text{turn}+1, \ldots, n-1, 0, \ldots, \text{turn}-1) \) enters its critical section.
- Bounded Wait
  * Upon exit from the critical section, a process must designate its unique successor the first contending process in the cyclic ordering \( \text{turn+1}, \ldots, n-1, 0, \ldots, \text{turn-1}, \text{turn} \).
  * Any process waiting to enter its critical section will do so in at most \( n-1 \) turns.

- Bakery Algorithm
  - Each process has a unique id
  - Process id is assigned in a completely ordered manner

```c
extern bool choosing[n];    /* Shared Boolean array */
extern int number[n];       /* Shared integer array to hold turn number */

void process_i ( const int i ) /* ith Process */ {
  do
    choosing[i] = true;
    number[i] = 1 + max(number[0], ..., number[n-1]);
    choosing[i] = false;
    for ( int j = 0; j < n; j++ )
      { 
        while ( choosing[j] );  // Wait while someone else is choosing
        while ( ( number[j] ) && (number[j],j) < (number[i],i) );
      }
  critical_section();
  number[i] = 0;
  remainder_section();
  while ( 1 );
}
```
- If \( p_i \) is in its critical section and \( p_k \ (k \neq i) \) has already chosen its \( \text{number}[k] \neq 0 \), then \( (\text{number}[i],i) \) < \( (\text{number}[k],k) \).

Synchronization Hardware

- **test_and_set** instruction

```c
int test_and_set (int& target )
{
  int tmp;
  tmp = target;
  target = 1; /* True */
  return ( tmp );
}
```

- Implementing Mutual Exclusion with **test_and_set**

```c
extern bool lock ( false );

do
  while ( test_and_set ( lock ) );
critical_section();
lock = false;
remainder_section();
while ( 1 );

Semaphores

- Commonly used in many applications to communicate such as parking an airplane
- Producer-consumer Problem
  - Shared buffer between producer and consumer
  - Number of items kept in the variable `count`
  - Printer spooler
  - The `|` operator
  - Race conditions
- An integer variable that can only be accessed through two standard atomic operations – wait (P) and signal (V)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semaphore</th>
<th>Dutch</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait</td>
<td>P</td>
<td><code>proberen</code></td>
<td>test</td>
</tr>
<tr>
<td>Signal</td>
<td>V</td>
<td><code>verhogen</code></td>
<td>increment</td>
</tr>
</tbody>
</table>

- The classical definitions for `wait` and `signal` are

\[
\text{wait (S): } \quad \text{while (S <= 0); } \\
\text{S--;}
\]

\[
\text{signal (S): S++;}
\]

- Mutual exclusion implementation with semaphores

```c
do
    wait (mutex);
    critical_section();
    signal (mutex);
    remainder_section();
while ( 1 );
```

- Synchronization of processes with semaphores

```
p1 \quad S_1;
    signal (synch);
```

```
p2 \quad \text{wait (synch); } \\
    S_2;
```

- Implementing Semaphore Operations
  - Binary semaphores using `test_and_set`
    * Check out the instruction definition as previously given
  - Implementation with a `busy-wait`
class bin_semaphore
{
    private:
        bool s;  /* Binary semaphore */

    public:
        bin_semaphore() // Default constructor
            : s ( false )
        {}

        void P() // Wait on semaphore
        {
            while ( test_and_set ( s ) );
        }

        void V () // Signal the semaphore
        {
            s = false;
        }
};

– General semaphore

class semaphore
{
    private:
        bin_semaphore mutex;
        bin_semaphore delay;
        int count;

    public:
        semaphore ( const int num = 1 ) // Constructor
            : count ( num )
        {
            delay.P();
        }

        void P ()
        {
            mutex.P();
            if ( --count < 0 )
            {
                mutex.V();
                delay.P();
            }
            mutex.V();
        }

        void V ()
        {
            mutex.P();
            if ( ++count <= 0 )
                delay.V();
            else
                mutex.V();
        }
    }
Busy-wait Problem – Processes waste CPU cycles while waiting to enter their critical sections
* Modify wait operation into the block operation. The process can block itself rather than busy-waiting.
* Place the process into a wait queue associated with the critical section
* Modify signal operation into the wakeup operation.
* Change the state of the process from wait to ready.

Block-Wakeup Protocol

// Semaphore with block wakeup protocol
class sem_int{
private:
  int value;  // Number of resources
  queue<pid_t> l;  // List of processes
public:
  void sem_int( const int n = 1 );  // Constructor
  { value = n; }
  l = queue<pid_t>( 0 );  // Empty queue

  void P()
  { if ( --value < 0 )
    { pid_t p = getpid();
      l.enqueue( p );  // Enqueue the invoking process
      block( p );
    }
  }

  void V()
  { if ( ++value <= 0 )
    { process p = l.dequeue();
      wakeup( p );
    }
  }
};

Producer-Consumer problem with semaphores

extern semaphore mutex;  // To get exclusive access to buffers
extern semaphore empty ( n );  // Number of available buffers
extern semaphore full ( 0 );  // Initialized to 0

void producer()
{ do
  { }
produce (item);
empty.P(); // empty is semaphore
mutex.P(); // mutex is semaphore
put (item);
mutex.V()
full.V()
} while (1);
}

void consumer()
{
do
{
full.P();
mutex.P();
remove (item);
mutex.V();
empty.V();
consume (item);
} while (1);
}

Problem: What if order of wait is reversed in producer

Thundering herd

• All processes in a wait queue are woken up simultaneously in response to an event
• They race for a resource that can be accessed by only one of them; remaining processes are put back to sleep
• Avoid the problem by waking up only one process

Higher-Level Synchronization Methods

• P and V operations do not permit a segment of code to be designated explicitly as a critical section.
• Two parts of a semaphore operation; should be treated as distinct
  – Block-wakeup of processes
  – Counting of semaphore
• Possibility of a deadlock – Omission or unintentional execution of a V operation.
• Monitors
  – Implementation easiest to view as a class with private and public functions
  – Collection of data [resources] and private functions to manipulate this data
  – A monitor must guarantee the following:
    * Access to the resource is possible only via one of the monitor procedures
    * A process enters the monitor by invoking one of its public procedures
    * Procedures are mutually exclusive in time; only one process at a time can be active within the monitor
  – Additional mechanism for synchronization or communication – the condition construct

```c
condition x;
```
* condition variables are accessed by only two operations – wait and signal
Interprocess Communication

*x.wait()* suspends the process that invokes this operation until another process invokes *x.signal()*

*x.signal()* resumes exactly one suspended process; it has no effect if no process is suspended

- Selection of a process to execute within monitor after *signal*

  *x.signal()* executed by process P allowing the suspended process Q to resume execution
  
  1. P waits until Q leaves the monitor, or waits for another condition
  2. Q waits until P leaves the monitor, or waits for another condition

  Choice 1 advocated by Hoare

- The Dining Philosophers Problem – Solution by Monitors

```cpp
enum state_type { thinking, hungry, eating };

class dining_philosophers {
    private:
        state_type state[5]; // State of five philosophers
        condition self[5]; // Condition object for synchronization

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

    public:
        dining_philosophers() // Constructor
        {
            for ( int i = 0; i < 5; state[i++] = thinking );
        }

        void pickup ( const int i ) // i corresponds to the philosopher
        {
            state[i] = hungry;
            test ( i );
            if ( state[i] != eating )
                self[i].wait();
        }

        void putdown ( const int i ) // i corresponds to the philosopher
        {
            state[i] = thinking;
            test ( ( i + 4 ) % 5 );
            test ( ( i + 1 ) % 5 );
        }
    }

- Philosopher i must invoke the operations pickup and putdown on an instance dp of the dining_philosophers monitor
```
dining_philosophers dp;

dp.pickup(i); // Philosopher i picks up the chopsticks
...
dp.eat(i); // Philosopher i eats (for random amount of time)
...
dp.putdown(i); // Philosopher i puts down the chopsticks

- No two neighbors eating simultaneously – no deadlocks
- Possible for a philosopher to starve to death

• Implementation of a Monitor

  - Execution of procedures must be mutually exclusive
  - A wait must block the current process on the corresponding condition
  - If no process in running in the monitor and some process is waiting, it must be selected. If more than one waiting process, some criterion for selecting one must be deployed.
  - Implementation using semaphores

  * Semaphore mutex corresponding to the monitor initialized to 1
    - Before entry, execute wait(mutex)
    - Upon exit, execute signal(mutex)
  * Semaphore next to suspend the processes unable to enter the monitor initialized to 0
  * Integer variable next_count to count the number of processes waiting to enter the monitor

```
void proc() { ... } // Body of process
...
if ( next_count > 0 )
    next.signal();
else
    mutex.signal();
```

* Semaphore x.sem for condition x, initialized to 0
* Integer variable x.count

```c
class condition {
    int num_waiting_procs; // Processes waiting on this condition
    semaphore sem; // To synchronize the processes
    static int next_count; // Processes waiting to enter monitor
    static semaphore next;
    static semaphore mutex;

public:
    condition() // Default constructor
        : num_waiting_procs ( 0 ), sem ( 0 )
    {}

    void wait()
    {
        num_waiting_procs++; // # of processes waiting on this condition
        if ( next_count > 0 ) // Someone waiting inside monitor?
            next.signal(); // Yes, wake him up
        else
```
mutex.signal(); // No, free mutex so others can enter
sem.wait(); // Start waiting for condition
num_waiting_procs--; // Wait over, decrement variable

void signal()
{
    if ( num_waiting_procs <= 0 ) // Nobody waiting?
        return;
    next_count++; // # of ready processes inside monitor
    sem.signal(); // Send the signal
    next.wait(); // You wait; let signaled process run
    next_count--; // One less process in monitor
}

Message-Based Synchronization Schemes

• Process interaction involves two things: synchronization (mutual exclusion) and communication (information exchange)

• Communication between processes is achieved by:
  – Shared memory (semaphores, CCRs, monitors)
  – Message systems
    * Desirable to prevent sharing, possibly for security reasons or no shared memory availability due to different physical hardware

• Communication by Passing Messages
  – Processes communicate without any need for shared variables
  – Paradigm of choice for distributed systems, shared memory multiprocessors, and uniprocessors
  – Two basic communication primitives
    * send message
    * receive message

    send(P, message) Send a message to process P
    receive(Q, message) Receive a message from process Q

  – Messages passed through a communication link

• Producer/Consumer Problem

void producer ()
{
    while ( 1 )
    {
        produce ( data );
        send ( consumer, data );
    }
}

void consumer ()
{
    while ( 1 )
    {
        receive ( producer, data );
        consume ( data );
    }
}

• Issues to be resolved in message communication
  – Synchronous v/s Asynchronous Communication
Upon `send`, does the sending process continue (asynchronous or nonblocking communication), or does it wait for the message to be accepted by the receiving process (synchronous or blocking communication)?

What happens when a `receive` is issued and there is no message waiting (blocking or nonblocking)?

**Implicit v/s Explicit Naming**

Does the sender specify exactly one receiver (explicit naming) or does it transmit the message to all the other processes (implicit naming)?

```
send (p, message)    # Send a message to process p
send (A, message)    # Send a message to mailbox A
```

Does the receiver accept from a certain sender (explicit naming) or can it accept from any sender (implicit naming)?

```
receive (p, message)    # Receive a message from process p
receive (id, message)   # Receive a message from any process; id is the process id
receive (A, message)    # Receive a message from mailbox A
```

**Ports and Mailboxes**

- Achieve synchronization of asynchronous process by embedding a busy-wait loop, with a non-blocking `receive` to simulate the effect of implicit naming
  - Inefficient solution

- Indirect communication avoids the inefficiency of busy-wait
  - Make the queues holding messages between senders and receivers visible to the processes, in the form of mailboxes
  - Messages are sent to and received from mailboxes
  - Most general communication facility between \( n \) senders and \( m \) receivers
  - Unique identification for each mailbox
  - A process may communicate with another process by a number of different mailboxes
  - Two processes may communicate only if they have a shared mailbox

- Properties of a communication link
  - A link is established between a pair of processes only if they have a shared mailbox
  - A link may be associated with more than two processes
  - Between each pair of communicating processes, there may be a number of different links, each corresponding to one mailbox
  - A link may be either unidirectional or bidirectional

- Ports
  - In a distributed environment, the `receive` referring to same mailbox may reside on different machines
  - Port is a limited form of mailbox associated with only one receiver
  - All messages originating with different processes but addressed to the same port are sent to one central place associated with the receiver

**Remote Procedure Calls**

- High-level concept for process communication, allowing functions to be called without using `send/receive` primitives
  - `send/receive` work like semaphores, taking attention away from the task at hand
  - RPCs allow the called function to be perceived as a service request
Interprocess Communication

- Transfers control to another process, possibly on a different computer, while suspending the calling process
- Called procedure resides in separate address space and no global variables are shared
- Return statement executed by called function returns control to the caller
- Communication strictly by parameters
  
  \[
  \text{send (RP\_guard, parameters);} \\
  \text{receive (RP\_guard, results);} \\
  \]

- The remote procedure guard is implemented by

  \[
  \text{void RP\_guard ( void )} \\
  \{
  \text{do} \\
  \text{receive (caller, parameters);} \\
  \text{...} \\
  \text{send (caller, results);} \\
  \text{while ( 1 );} \\
  \}
  \]

- Static versus dynamic creation of remote procedures

Signals and Interprocess Communication in Unix/Linux

- POSIX standard defines about 20 signals, two of which are user definable
- Process can react to signals in two ways
  1. Ignore the signal
  2. Asynchronously execute a signal handler
- If the process does not specify one of those two alternatives, kernel performs a default action based on signal number as follows:
  - Terminate the process
  - Dump core and terminate the process
    - Core includes the execution context and contents of the address space
  - Ignore the signal
  - Suspend the process
  - Resume the process if it was stopped
- **SIGKILL** and **SIGSTOP** signals cannot be handled directly by the process or ignored
- IPC resources
  - Shared memory, semaphores, and message queues
  - Acquired by a process using **shmget(2)**, **semget(2)**, and **msgget(2)**
  - Persistent: Must be explicitly deallocated by creator, current owner, or root
  - **msgsnd(2)** and **msgrcv(2)**
  - Shared memory
    - **shmget(2)** creates shared memory of required size
    - **shmat(2)** gets the starting address of new region within the process address space
    - **shmdt(2)** detaches the shared memory from process address space