

Socket

- Socket is defined as communication endpoint.
- Sockets can be used for bi-directional communication.
- Using socket one process can communicate with another process on same machine (UNIX socket) or different machine (INET sockets) in the network.
- Sockets can also be used for communication over bluetooth, CAN, etc.
- terminal> man 7 socket

UNIX socket

- Bi-directional stream based communication mechanism to communicate between two processes running on the same computer.
- UNIX socket is a special file (s).

INET (Internet) socket

- Bi-directional communication mechanism to communicate between two processes running on the same computer or different computers in a network.
- INET socket do not have special file. It is identified by IP address and port number combination.

Shared Memory

Shared Memory SysCalls

- shmget() syscall
 - Create the shared memory object.
 - `shmid = shmget(shm_key, size, flags);`
 - arg1: unique key for shared memory object
 - arg2: size of shared memory region (bytes)
 - arg3: to create shared memory -- `IPC_CREAT | perm`
 - perm --> octal number --> 0600 (rw- --- ---)
 - returns: shm id (index to shared memory table) on success.
- shmctl() syscall
 - To control shared memory object/region e.g. mark shm object for deletion, get info about shared memory.
 - `ret = shmctl(shmid, IPC_RMID, NULL);`
 - arg1: shared memory id to be deleted.
 - arg2: command = `IPC_RMID` to delete the shared memory object
 - arg3: for deletion third arg is not required.
 - This will mark shared memory object for destruction.
 - The shared memory object will be deleted when no processes are attached to it (`nattach = 0`).
- shmat() syscall

- Get the address of the shared memory region i.e. attach shm region to the current process.
 - Internally increments nattach count in shared memory object.
- `ptr = shmat(shmid, virt_addr, flags);`
 - `arg1`: shared memory id of region to be attached to the process.
 - `arg2`: base virtual address in address space of the current process to be mapped to shared memory region.
 - NULL means use any available address.
 - `arg3`: flags (extra information/behaviour).
 - 0 means default behaviour.
 - returns pointer to the shared memory region on success. -1 is returned on failure.
- `shmdt()` syscall
 - Release the shared memory pointer i.e. detach shm region from the current process.
 - Internally decrements nattach count in shared memory object.
 - If nattach count become zero and shared memory region is marked for deletion, delete the shared memory.
 - `shmdt(ptr);`
 - `arg1`: shared memory pointer to be detached.

Thread concept

- Threads are used to execute multiple tasks concurrently in the same program/process.
- Thread is a light-weight process.
 - For each thread new control block and stack is created. Other sections (text, data, heap, ...) are shared with the parent process.
 - Inter-thread communication is much faster than inter-process communication.
 - Context switch between two threads in the same process is faster.
- Thread stack is used to create function activation records of the functions called/executed by the thread.

Process vs Thread

- In modern OS, process is a container holding resources required for execution, while thread is unit of execution/scheduling.
- Process holds resources like memory, open files, IPC (e.g. signal table, shared memory, pipe, etc.).
- PCB contains resources information like pid, exit status, open files, signals/ipc, memory info, etc.
- CPU time is allocated to the threads. Thread is unit of execution.
- TCB contains execution information like tid, scheduling info (priority, sched algo, time left, ...), Execution context, Kernel stack, etc.
- `terminal> ps -e -o pid,nlwp,cmd`
- `terminal> ps -e -m -o pid,tid,nlwp`

main thread

- For each process one thread is created by default called as main thread.
- The main thread executes entry-point function of the process.
- The main thread use the process stack.

- When main thread is terminated, the process is terminated.
- When a process is terminated, all threads in the process are terminated.

thread functions

- `pthread_create()`
 - Create a new thread.
 - `arg1`: posix thread id (out param)
 - `arg2`: thread attributes -- NULL means default attributes
 - stack size
 - scheduling policy
 - priority
 - `arg3`: address of thread function
 - `void* thread_function(void*)`;
 - `arg1`: `void*` -- param to the thread function (can be of any type, array or struct).
 - returns: `void*` -- result of thread (can be of any type)
 - `arg4`: param to thread function
 - returns: 0 on success.
- Join thread
 - The current thread wait for completion of given thread and will collect return value of that thread.
 - `pthread_join(th_id, &res);`
 - `arg1`: given thread (for which current thread is to blocked).
 - `arg2`: address of result variable (out param to collect result of the given thread)
- Thread termination
 - When thread function is completed, the thread is terminated.
 - To terminate current thread early use `pthread_exit()` function.
 - `pthread_exit(result);`
 - `arg1`: result (`void*`) of the current thread
- Thread cancellation
 - A thread in a process can request to cancel execution of another thread.
 - `pthread_cancel(tid)`
 - `arg1`: id of the thread to be cancelled.

Threading models

- Threads created by thread libraries are used to execute functions in user program. They are called as "user threads".
- Threads created by the syscalls (or internally into the kernel) are scheduled by kernel scheduler. They are called as "kernel threads".
- User threads are dependent on the kernel threads. Their dependency/relation (managed by thread library) is called as "threading model".

- There are four threading models:
 - Many to One
 - Many to Many
 - One To One
 - Two Level Model
- Many to One
 - Many user threads depends on single kernel thread.
 - If one of the user thread is blocked, remaining user threads cannot function.
- Many to Many
 - Many user threads depend on equal or less number of kernel threads.
 - If one of the user thread is blocked, other user thread keep executing (based on remaining kernel threads).
- One To One
 - One user thread depends on one kernel thread.
- Two Level Model
 - OS/Thread library supports both one to one and many to many model

Synchronization

- Multiple processes accessing same resource at the same time, is known as "race condition".
- When race condition occurs, resource may get corrupted (unexpected results).
- Peterson's problem, if two processes are trying to modify same variable at the same time, it can produce unexpected results.
- Code block to be executed by only one process at a time is referred as Critical section. If multiple processes execute the same code concurrently it may produce undesired results.
- To resolve race condition problem, one process can access resource at a time. This can be done using sync objects/primitives given by OS.
- OS Synchronization objects are:
 - Semaphore, Mutex

Semaphore

- Semaphore is a sync primitive given by OS.
- Internally semaphore is a counter. On semaphore two operations are supported:
 - wait operation: dec op: P operation:
 - semaphore count is decremented by 1.
 - if $\text{cnt} < 0$, then calling process is blocked.
 - typically wait operation is performed before accessing the resource.
 - signal operation: inc op: V operation:
 - semaphore count is incremented by 1.
 - if one or more processes are blocked on the semaphore, then one of the process will be resumed.
 - typically signal operation is performed after releasing the resource.
- Semaphore types
 - Counting Semaphore
 - Allow "n" number of processes to access resource at a time.
 - Or allow "n" resources to be allocated to the process.

- Binary Semaphore
 - Allows only 1 process to access resource at a time or used as a flag/condition.

Mutex

- Mutex is used to ensure that only one process can access the resource at a time.
- Functionally it is same as "binary semaphore".
- Mutex can be unlocked by the same process/thread, which had locked it.

Semaphore vs Mutex

- S: Semaphore can be decremented by one process and incremented by same or another process.
- M: The process locking the mutex is owner of it. Only owner can unlock that mutex.
- S: Semaphore can be counting or binary.
- M: Mutex is like binary semaphore. Only two states: locked and unlocked.
- S: Semaphore can be used for counting, mutual exclusion or as a flag.
- M: Mutex can be used only for mutual exclusion.

Deadlock

- Deadlock occurs when four conditions/characteristics hold true at the same time.
 - No preemption: A resource should not be released until task is completed.
 - Mutual exclusion: Resources is not sharable.
 - Hold & Wait: Process holds a resource and wait for another resource.
 - Circular wait: Process P1 holds a resource needed for P2, P2 holds a resource needed for P3 and P3 holds a resource needed for P1.

Deadlock Prevention

- OS syscalls are designed so that at least one deadlock condition does not hold true.
- In UNIX multiple semaphore operations can be done at the same time.

Deadlock Avoidance

- Processes declare the required resources in advanced, based on which OS decides whether resource should be given to the process or not.
- Algorithms used for this are:
 - Resource allocation graph: OS maintains graph of resources and processes. A cycle in graph indicate circular wait will occur. In this case OS can deny a resource to a process.
 - Banker's algorithm: A bank always manage its cash so that they can satisfy all customers.
 - Safe state algorithm: OS maintains statistics of number of resources and number processes. Based on stats it decides whether giving resource to a process is safe or not (using a formula):
 - $\text{Max num of resources required} < \text{Num of resources} + \text{Num of processes}$
 - If condition is true, deadlock will never occur.
 - If condition is false, deadlock may occur

Deadlock recovery

- it is done by one of following method

- Resource preemption
- process termination

SUNBEAM