Operating System Concepts

Agenda

- syscall internals
- process hierarchy
- orphan & zombie state
- thread vs process
- thread creation
- synchronization
- semaphore vs mutex
- deadlock

syscall internals

- Refer yesterday's slides

Process Creation

- System Calls
  - Windows: CreateProcess()
  - UNIX: fork()
  - BSD UNIX: fork(), vfork()
  - Linux: clone(), fork(), vfork()

fork() syscall

- To execute certain task concurrently we can create a new process (using fork() on UNIX).
- fork() creates a new process by duplicating calling process.
- The new process is called as "child process", while calling process is called as "parent process".
- "child" process is exact duplicate of the "parent" process except few points pid, parent pid, etc.

- pid = fork();
  - On success, fork() returns pid of the child to the parent process and 0 to the child process.
  - On failure, fork() returns -1 to the parent.
- Even if child is copy of the parent process, after its creation it is independent of parent and both these processes will be scheduled sepeately by the scheduler.
- Based on CPU time given for each process, both processes will execute concurrently.

getpid() vs getppid()

- pid1 = getpid(); // returns pid of the current process
- pid2 = getppid(); // returns pid of the parent of the current process

When fork() will fail?

- When no new PCB can be allocated, then fork() will fail.
- Linux has max process limit for the system and the user. When try to create more processes, fork() fails.
- terminal> cat /proc/sys/kernel/pid_max

Orphan process

- If parent of any process is terminated, that child process is known as orphan process.
- The ownership of such orphan process will be taken by "init" process.

Zombie process

- If process is terminated before its parent process and parent process is not reading its exit status, then even if process's memory/resources is released, its PCB will be maintained. This state is known as "zombie state".
- To avoid zombie state parent process should read exit status of the child process. It can be done using wait() syscall.

wait() syscall

- ret = wait(&s);
  - arg1: out param to get exit code of the process.
  - returns: pid of the child process whose exit code is collected.

- wait() performs 3 steps:
    - Pause execution parent until child process is terminated.
    - Read exit code from PCB of child process & return to parent process (via out param).
    - Release PCB of the child process.
- The exit status returned by the wait() contains exit status, reason of termination and other details.
- Few macros are provided to access details from the exit code.
    - WEXITSTATUS()

waitpid() syscall

- This extended version of wait() in Linux.
- ret = waitpid(child_pid, &s, flags);
    - arg1: pid of the child for which parent should wait.
        - -1 means any child.
    - arg2: out param to get exit code of the process.
    - arg3: extra flags to define behaviour of waitpid().
- returns: pid of the child process whose exit code is collected.
    - -1: if error occurred.

exec() syscall

- exec() syscall "loads a new program" in the calling process's memory (address space) and replaces the older (calling) one.
- If exec() succeed, it does not return (rather new program is executed).
- There are multiple functions in the family of exec():
    - execl(), execlp(), execle(),
    - execv(), execvp(), execve(), execvpe()
- exec() family multiple functions have different syntaxes but same functionality.

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 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, mututal 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):
        - Max num of resources required < Num of resources + Num of processes
            - If condition is true, deadlock will never occur.
            - If condition is false, deadlock may occur

IPC overview

- A process cannot access of memory of another process directly. OS provides IPC mechanisms so that processes can communicate with each other.
- IPC models
    - Shared memory model
        - Processes write/read from the memory region accessible to both the processes.
        - OS only provides access to the shared memory region.
    - Message passing model
        - Process send message to the OS and the other process receives message from the OS.
        - This is slower compared to shared memory model.
- Unix/Linux IPC mechanisms
    - Signals
    - Shared memory
    - Message queue

    - Pipe
    - Socket

VI Editor

- Refer slides
- sudo apt-get install vim
- VI editor works in two modes
    - command mode
    - insert mode
- press i - to go into insert mode
- press Esc - to go into command node
- VI editor commands:

    - :w - write/save into file

    - :q - quit vi editor

    - :wq - save and quit

    - :y - to copy current line

    - yy - to copy current line

    - nyy - copy n lines from current line

    - :m,ny - copy fomr mth line to nth line

    - :d - to cut current line

    - dd - to cut current line

    - ndd - cut n lines from current line

    - :m,nd - cut fomr mth line to nth line

- press p - to paste copied line on next line of current line

- To do setting in vi editor
  - create file into home directory as below:

```
vim ~/.vimrc
```

- add below content into it

```
set number
set autoindent
set tabstop=4
set nowrap
```