Task 1

Process representation

Process is represented by a task_structure, that contains the following information:
- State
- Scheduling information (is real-time, allowed exec time, priority)
- Identifiers: user and group id
- Interprocess communication
- Links: link to parent process, to its siblings and children
- Times and timers: process creation time and amount of CPU time so far consumed
- File system: pointers to any files opened by this process
- Address space
- Context: registers and stack information

States:

•Sleeping

1.Present in main memory
2.Present in secondary memory storage (swap space on disk)
When: process needs resources that are not currently available. Goes to sleep either voluntarily or kernel puts is into Sleep state.

•Interruptible

1.Blocked state, in which process is waiting for an event, such as the end of an I/O operation, the availability of a resource, or a signal from another process

Example: read() system call. During handling that, while process is sleeping waiting for I/O, it can receive a async signal (e.g: SIGTERM):
- The system call exits prematurely, return -EINTR (signal for interrupted system call) to userspace
- The signal handler is executed
- If process is still running, it gets return value from the sys call

•Uninterruptible

1.As the one above, but is waiting directly on hardware conditions and therefore will not handle any signals. Used when operation needs to be atomic

Such process is in a system call (kernel function) and thus, cannot be interrupted by a signal

Example:
1. Thread tries to access page, got page fault, need to wait for I/O
2. Kernel is loading it…
3. Process can’t continue until the page is available, not can be interrupted in this state, b/c it can’t handle any signals. If it did, another page fault would happen (that makes a cycle)

Quick example: mkdir is not interruptible, as well as:
Reading CD disk, reading hard drive (hardware-related)

•Terminate/Stop

1.Sends SIGCHLD signal to parent, and releases its data structure, but not its slot in the process table.
2.It’s up to the parent process to release the child process slot.
3.In the meantime, the child enters into Zombie state. Process can remain in Zombie state if the parent process dies before it has a chance to release the process slot of the child process.
4.You cannot kill a Zombie process b/c you cannot send a signal to the process, as it no longer exists

Synchronous:
- I/O
- Waiting for other process
Asynchronous:
- Signal

New -> Ready - OS
Ready -> Running - Scheduler
Running -> Exit - terminated by OS / parent process
Running -> Ready - Scheduler
Running -> Blocked - Kernel, Other process, driver
Blocked -> Ready - Kernel
Ready -> Exit - parent process or kernel

Task 2 - Process creation

UNIX

Fork - creates an exact clone of the calling process.
After fork, the two processes, the parent and the child have the same:
- memory image
- environment strings
- open files (child receives duplicates of all of the parent’s file descriptors)

Conceptually, fork is creating a copy of: parent’s text, data, heap and stack segments

Usually it is followed by execve or a similar system call to change its memory image and run a new program.

Reason? Allow the child manipulate its file descriptors after the fork, but before the execve in order to accomplish redirection of standard I/O and error

Copy-on-write - child share all of the parent’s memory, but in case any side modifies part of the memory, that chunk of memory is explicitely copied

Demand paging - processes are started up with none of their pages in memory. During run-time, it fetches more and more pages, and after a while, the process has most of the pages it needs and settles down to run with relatively few page faults. This strategy is called demand paging, because pages are loaded lazily: on demand, not in advance.

exec(pathname, argv, envp) - loads a new program into a process's memory.
The existing program text is discarded, and the stack, data and heap segments are freshly created for the new program

Windows processes

Created from section objects, each of which describes a memory object backed by a file on disk. When new process is created, the creator receives a handle that allows it to modify the new process by:
- mapping sections
- allocating virtual memory
- writing parameters and environmental data
- duplicating file descriptors
- creating threads

Unix was designed for 16bit single-processor systems that used swapping to share memory among processes. Initially, when there was no virtual memory, fork was created by swapping out the parent and giving physicall memory to the child.

NT used processes as containers for sharing memory and object resources. It used threads as the unit of concurrency for scheduling.

Windows can group processes into jobs. Jobs may apply constraints to the processes and threads they contain - such as limiting resources

Fibers - level below threads. Fibers are created by allocating a stack anda user-mode fiber data structure for storing registers and data associated with the fiber.

Each thread can run a particular subset of fibers.

Process creating is handled by CreateProcess Win32 API function.
Params:
- name of the file to be executed
- cmd strings
- various security attributes
- priority information
- pointer to the environment string
- current working directory
- optional data structure with information about the GUI Window
- rather than just the PID, i returns handle and ID, both for the new process and for its initial thread.

Differences:

1.current working directory is a kernel mode concept in UNIX by a user-mode string in Windows

2.UNIX parses the command line and passes an array of parameters while Win32 leaves argument parsing up to the individual program.

Consequences: Inconsistency. (e.g wildcards might be handled in a different way in every program)

3.Whether file descriptor can be inherited in UNIX is a property of the handle. In Windows it is a property of both the handle and a parameter to process creation.

4.Win32 is GUI oriented, so new processes are directly passed information about their primary window.

5.Windows does not have a SETUID but as a property of executable, but one process can create a process that runs as a different user, as long as it can obtain a token with that users' credentials

6.The process and thread handle returned from Windows can be used at any time to modify the new process/thread in many substantive ways, including modifying the virtual memory, injecting threads into the process and altering the execution of threads.

UNIX only makes modifications to the new process only between the fork and exec call, and only in limited ways, since exec throws out all the user-mode state of the processs

7.Search path for finding the program to execute is buried in the library code for Win32, but managed more explicitely in UNIX

8.No copy-on-write on windows: parent's and child's address spaces are different from the start

Task 3

Signals

Where do we use them:
- Software interrupts
- Processes can send signals to each other
- We can define our own signal handlers
- Debuggers are using them: e.g to stop a program

Signals from kernel:
- Hardware exception (malformed instruction, no access to memory)
- From user (interrupt, suspend)

•limited form of inter-process communication

•async notification sent to a process or a specific thread within the process in order to notify of an event that occured

•When signal is received, OS interrupts the target process' flow of execution to deliver the signal

•If process has registered signal handler, that routine is executed. Otherwise default handler is executed.

•Special signals that kernel unconditionally acts on:

•SIGKILL

- SIGSTOP
- Signals vs Interrupts:
  - Interrupts are handled by kernel
  - Signals are mediated by the kernel and handled by processes
  - Kernel may pass interrupt as a signal to the process that caused it.
    - SIGSEGV - Segmentation fault (attempt to access restricted area of memory)
    - SIGBUS - Bus error: process is trying to access memory that the CPU cannot physicall address: an invalid address for the address bus. Examples:
      - Non-existent address
      - Unaligned access
      - Paging errors: when virtual memory pages cannot be paged in, e.g b/c it has disappeared
    - SIGILL - illegal, malformed unknown or privileged instruction
    - SIGFPE - erroneous arithmetic operation, such as zero division
- Signal route
  - Generated by kernel or kill syscall
  - Kernel confirms the calling process has sufficient privileges to send the signal. If not, an error is returned
  - Next, kernel figures out what to do with the signal:
    - Executes process' signal handler or default one
    - Programs might turn off the delivery of signals. Signals will stay pending until unblocked
    - ???

When handling SIGSEGV or SIGILL makes sense?

```
typedef struct {
int int_code; /* interrupt code /
char *EPIE; / pointer to hardware program check info */
} SEGV_t;
```

EPIE - pointer to control block containing hardware information available at the time the signal occured (include program status word and registers)

You may check int_code (interrupt code) field of structure that can be obtained using siginfo. You can also inspect stacktrace

Other example: when you have memory obtained using mmap(2) that is read-only, you may check if the signal handler that cause seg fault was writing to that memory and use mprotect to change the protection of the memory

Order of signals delivery

Order of delivery is not specified in UNIX, although linux have its own convention.

Synchronous signals: SIGSEGV, SIGBUS, SIGILL, SIGTRAP, SIGFPE, SIGSYS

Order of delivery: at first synchronous signals, then ascending by their number (num 1 is first)

Task 5

Kernel thread

- •Sometimes called LWP
- •Created and scheduled by the kernel
- •Does not operate on user space - only in kernel mode

We distinguish two models

1. many-to-one threading: many user processes maps directly to one kernel thread
2. one-to-one threading: each user thread maps to one kernel thread

Process group

```
- a collection of one or more processes
- When signal is sent to group, it is distributed to all of the members of the group
```

ps output

ps -eo user,pid,pgid,tid,pri,stat,wchan,cmd.

user - effective user name
pid = process' ID
pgid - group ID
tid = lwp - (light weight process or thread) ID of the lwp being reported
pri - process' priority
stat - process' status code:
D - uninterruptible sleep
R - running or runnable
S - interruptibls sleep
T - stopped by job control signal
t - stopped by debugger during the tracing
X - dead (should never be seen_
Z - Zombie

```
Additional characters for BSD formats:

< - high priority
N - low-priority
L - has paged locked into memory
s - session leader
l - is multi-threaded
+ - is in the foreground process group
```

wchan - name of the kernel function is which the process is sleeping
cmd = a command with its all arguments

Kernel threads - CMD is not a command line but bracketed expression [xyz]
Also, are sons of kthreadd, which means its PPID is equal to 2

The parent of init is process '0' - which probably means it has no parent

List all user and kernel processes: ps -ef

pstree:
- n * [{}] - n threads
- n * [] - n same processes

Task 6

/proc - virtual filesystem (sometimes reffered as process information
pseudo-file system)

- •Contains runtime system information (sys memory, devices mounted, hardware conf etc)
- •Can be regarded as control and information centre for the kernel
- •By altering files located in this directory, you can change read/change kernel parameters while the sys is running
- •(almost) all of the files have a size of 0. Why?
  - •The files does not contain any data, they just acts as a pointers to where the actual process information is stored (intuition: window to kernel)

/proc/$PID/maps

File containing the currently mapped memory regions and their access permissions

File format:

- •address - perms - offset - dev - inode - pathname
- •perms:
  - •rwx: read write execute
  - •s: shared
  - •p: private (copy on write)
- •offset:
  - •offset into the file
- •dev:
  - •dev: device (major:minor)
- •inode:
  - •inode on that device. 0 mean that no inode is associated with the memory region
- •pathname:
  - •usually: the file that is backing the mapping
  - •for ELF files, you can easily coordinate with the offset field (readelf -l)
  - [stack] - initial process's stack
  - [stack:tid] - thread's stack
  - [vsdo] - The virtual dynamically linked shared object
  - [heap] - process's heap
  - <blank> - anonymous mapping as obtained via mmap

mmap - map files or devices into memory (method of memory-mapped file I/O)
- Instead of allocating memory with malloc and reading the file, you map the whole file into memory without explicit reading it. Why? It lets you easily process files that are larger that the available memory using the OS-provided paging mechanism

Anonymous mapping: Mapping that is not backed by a file, and is basically a request for a chunk of memory

Inode
- Each object in the filesystem is represented by an inode.
- in short: inode identifies the file and its attributes
- Data structure on a traditional unix file system, such as ext3. Inode stores basic information about a regular file, directory or other file sys object
- Stores:
- Metadata (time of last change, access, modification)
- owner, permission data
- Attributes and disk block locations of the object’s data

Task 7

lsof output:

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

FD - File Descriptor. Values:
- cwd: current working directory
- txt: txt file
- mem: memory mapped file
- mmap: memory mapped device
- unix: UNIX domain socket
- IPv4: IPv4 socket
- sock: socket of unknown domain
- <NUM>: actual file descriptor

Type - Specifies the type of the file. Some of the values:
- REG: regular file
- DIR: Directory
- FIFO: first in first out
- CHR: Character special file

lsof -c firefox - list opened files by given process name

lsof -c firefox | awk '{ sum += $7; } END { print sum; }' "$@" - sum of bytes used

echo $(($x / 1024. / 1024. / 1024)) - convert to GBs