Process and Kernel

- OS provides execution environment to run user processes
  - Basic framework for code execution
  - Services like file management and I/O, and interface to the same

- Process
  - Single sequence of instructions in user address space
  - Control point or program counter
  - Multiple control points or threads

- Virtual machine and multiprogramming
  - Each process has its own registers and address space
  - Process gets global services (I/O) from the OS
  - Kernel stores address space in different memory objects, including physical memory, disk, swap areas
  - Memory management subsystem shuffles pages between physical memory and other storage objects
  - Every process needs registers but there is only one set of hardware registers
    - Kernel keeps process registers updated and loads and stores them into hardware registers for currently running process

- Resources
  - Processes contend for each resource, including CPU time, memory, peripherals
  - OS acts as a resource manager, distributing the resources fairly as well as optimally
  - Processes that do not get a resource (but need it) are blocked
    - Processes can also get blocked on CPU
    - Processes get CPU for short bursts of time, called a *quantum*, typically about 10 milliseconds
    - *Time slicing*

- OS provides a number of facilities to application programmers, such as use of I/O devices
  - Users do not have to write code to control these devices
  - OS provides high-level abstract programming interface to access these devices (such as *fopen(3)*)
  - OS also provides access synchronization and error recovery with these devices
  - The application programming interface (API) defines the semantics of all interactions between user code and OS

- We will look at the OS as something that provides us with resource management
  - In this respect, the kernel is the OS
  - Other utilities and programs in the OS environment (including shell and basic interface commands) will not be considered as part of the OS
  - Kernel without those utilities is not of much use
  - Kernel is the only indispensable part of the OS
  - There is one, and only one, kernel in the system at any one time
    - This is the reason why you cannot run more than one OS at one time on a machine, even though we may have two OS residing on the system
    - The second OS has to generally run in the emulation mode
• Kernel
  – Special program that runs directly on hardware
  – Implements the process model and other system services
  – Loaded at start up time during the bootstrapping phase
  – Initializes the system and sets up the environment to run processes
  – Creates spontaneous processes (init, swapper or pagedaemon, and scheduler); other processes are created by these processes
  – Kernel remains in memory till the system is shut down

• Unix functionality
  1. System call interface
     – Explicit service request to kernel
     – Central component of Unix API
     – Executed by kernel on behalf of user processes
  2. Hardware exceptions
     – Synchronous errors in process (divide by zero, stack overflow)
     – Handled by kernel on behalf of user process
  3. Interrupt handling
     – Asynchronous conditions
     – Used by devices to inform the kernel of I/O completion and status change
     – Interrupts are global events and are not related to any process
  4. Special system processes
     – Swapper and pagedaemon, to control the number of active processes and manage memory

Mode, Space, and Context

• Two different modes of execution – kernel mode and user mode
  – User programs execute in user mode
  – Kernel protects some parts of the address space from user-mode access
  – Privileged machine instructions, such as the ones to manipulate memory management registers, can only be executed in kernel mode
  – Intel x86 architecture has four rings of execution for security, with innermost rings being more privileged; Unix uses only two of those rings
  – User processes cannot corrupt the state of the operating system, accidentally or maliciously

• Virtual memory
  – Address translation from virtual to physical address, using address translation maps, or page tables
  – Memory management unit (MMU) has a set of registers to identify the translation maps (page tables) of the current process
  – During context switch, the kernel loads these registers with the translation maps of the new process
  – MMU registers are only accessible in kernel mode
    – Process can only access its own space and cannot access/modify the space that belongs to a different process

• Kernel space or system space
- A fixed part of virtual address space
- Only accessible in kernel mode
- Since there is a single kernel, all processes map to a single kernel address space
  - Current process address space can be directly accessed because the information resides in the MMU registers
  - Information on other processes is indirectly accessed through temporary mappings
  - Kernel is reentrant
- Used by kernel to maintain
  - Global data structures
  - Process-specific information
    - Information to access the address space of any process
- Protected from user-mode access
  - Processes must access the system space using system calls

- User area in memory (u area)
  - Contains information about process for use by kernel
  - Table of open files, process identification information, saved values of process registers
  - Process cannot modify this information (but may be able to read it)

- Kernel stack
  - Provided to facilitate reentrant nature of kernel
  - Owned by the kernel but present in process space, just like the u area
  - Process cannot access it (no user mode access)

- Execution context
  - Kernel functions can execute in *process context* or *system context*
  - Process context
    * Kernel acts on behalf of the current process
    * System call
    * Kernel can access and modify address space, u area, and kernel stack
    * Kernel may block the process if process wants to wait for a resource
  - System context
    * Also called *interrupt context*
    * Kernel performs system wide tasks like responding to interrupts or recomputing priority of the processes
    * Not performed on behalf of any process
    * Kernel cannot access the address space, u area, or kernel stack of current process
    * Kernel cannot block (as it is not associated with a process)

- How code runs?
  - User code: user mode and process context in process space
  - System calls and exceptions: kernel mode, process context, and both process and system space
  - Interrupts: Kernel mode, system context, and system space

Process abstraction
• Instance of a running program
  – One process but many programs over its lifetime
  – Process has its place in hierarchy – parent and child

• Process state
  – State of the process (initial/idle, ready to run, kernel running, user running, asleep, zombie)
  – State transitions
  – Two additional states in BSD versions of Unix – stopped and stopped+asleep; they were also incorporated in SVR4
  – Kernel is central to the entire operation and manages the transitions
  – Process can be stopped or suspended by a stop signal (SIGSTOP, SIGSTP, SIGTTIN, or SIGTTOU) each of which change the system state immediately
  – Stopped process can be resumed by a continue signal (SIGCONT)

• Process context
  – User address space
    * program text, data, user stack, shared memory
  – Control information
    * u area
    * proc structure
    * Kernel stack and address translation maps
  – Credentials
    * User and group id
  – Environment variables
    * Inherited from the parent, possibly defined in the shell
    * Stored at the bottom of the user stack
    * Manipulated using the standard library
    * Upon exec, caller may request to retain the environment variables or provide a new set
  – Hardware context
    * Set of general-purpose and system registers
    * Program counter
    * Stack pointer
    * Processor status word (PSW)
      · System state (current and previous execution modes)
      · Current and previous interrupt priority levels
      · Overflow and carry bits
    * Memory management registers (address translation maps)
    * Floating point unit registers
    * Entire context gets saved in process control block (PCB) in the u area upon context switch

• User credentials
  – UID and GID
  – Affect file ownership and access, and ability to signal other processes
  – Super user or root (uid 0, gid 1)
    * Has unlimited access privileges for files
* Can execute privileged system calls, such as `mknod`

- Child inherits credentials from parent

- Real and effective IDs
  * Affect file creation and access
  * `suid` and `sgid` installation modes
  * SVR3 maintains `saved uid` and `saved gid` as the effective values before `exec`
  * BSD allows a user to belong to a set of `supplemental groups`
  * SVR4 combines (and supports) both the features

** u area and proc structure

- Maintained by kernel for each process to keep control information

** proc structure

- Also known as the process table, and may be a fixed size array
- Kept in system space of the process
- Visible to kernel at all times even when the process is not running
- Fixed size of process table puts a limit on the maximum number of processes
- SVR4 allows for dynamic allocation of proc, but with a fixed size array of pointers

- Major fields are:
  - Identification (PID)
  - Location of kernel address map for the u area
  - Current process state
  - Forward and backward pointers in the scheduler queue or sleep queue
  - Sleep channel for blocked processes
  - Scheduling priority
  - Signal handling information (signal masks)
  - Memory management information
  - Pointers for links on active, free, or zombie process lists
  - Miscellaneous flags
  - Pointers for hash queue based on PID
  - Process hierarchy information

- u area

- Part of the process space
- Visible only when the process is running
- May be mapped at the same virtual address in each process, so that kernel can refer to it through the variable u
- Context switch resets the mapping
- Kernel may be able to access the u area of a different process

- Major fields:
  - Process control block
  - Pointer to the proc structure
  - Real and effective UID and GID
  - Arguments to and return value for current system call
  - Signal handler and related information
  - Information from program header (text, dat, stack size, memory management information)
  - Open file descriptor table
  - Pointer to vnodes of current directory and controlling terminal
- CPU usage statistics, profiling information, disk quotas, resource limits

**Executing in kernel mode**

- System enter kernel mode through
  - Device interrupt
  - Exception
  - Trap or software interrupt
- Kernel consults *dispatch table* to get the address of low-level routine to handle the event
- Kernel save the state of the interrupted process (PC + PSW) on its kernel stack
- Process state is restored after completing the requested task
- Interrupts are serviced in the system context and may not access process address space or u area; they must not block
- Exception handler runs in process context and may access process address space or u area
- Software interrupts (traps) are handled synchronously in process context; they may be caused by process to request services
- System call interface
  - To make system call, process executes a sequence of instructions to put the system in kernel mode (mode switch)
    * This is performed by a wrapper from the standard C library
    * Wrapper identifies the system call number and pushes it on user stack, and then, invokes the trap
  - Trap transfer control to kernel
    * Control goes to `syscall()`¹ – the handler for system calls
  - Operations performed in process context but kernel mode
    * Has access to process address space and u area
    * Uses kernel stack of the calling process
    * `syscall()` copies arguments for system call from user stack to the u area and saves hardware context on the kernel stack
    * Uses the system call number to index into system call dispatch vector (sysent[])
  - After completing system call, kernel returns the system to user mode and transfers control back to process
- Interrupt handling
  - Interrupt handler or interrupt service routine
  - Handler runs in kernel mode and system context
  - No need to access the process context
  - Cannot block
  - Time used to service the interrupt is charged to process even though the activity is not related to the process
    * The time for process is to be updated and hence, its proc structure needs to be accessed
    * Potential to corrupt part of the process address space

¹`syscall(3B)` on Solaris
Prioritizing the interrupts

- Interrupt priority levels (IPLs) from 0-31
- Process suspended only if its IPL is higher than the current IPL
- A lower IPL is saved into a saved interrupt register, and handled when the IPL drops sufficiently

Synchronization

- Reentrant kernel may have several processes active in the kernel
  - Only one process actually has the CPU while others are blocked on CPU or some other resource
  - They all share the same copy of kernel data structures
  - Possibility for the loss of integrity of some data structures
- Synchronization through nonpreemptive processing
  - Do not preempt a kernel mode process by another process even if its time quantum has expired
  - Process can voluntarily give up CPU
  - Kernel can work with the data structures without having to lock the same
  - Synchronization is still necessary in three case: blocking operations, interrupts, and multiprocessor synchronization
- Blocking operations
  - Blocks the process/puts it in sleep state
  - Kernel is nonpreemptive and may manipulate most data structures and resources
  - Some objects must be protected from blocking
    * A read from file into disk block buffer memory in kernel
    * Process blocks allowing others to run
    * Kernel must ensure that other processes do not access this buffer since the buffer is in an inconsistent state
  - Kernel protects an object by associating a lock with it
    * lock may be a single-bit flag
    * A process checks the lock before using an object
    * Kernel also associates a wanted flag with the object
    * When a process releases an object, it checks the wanted flag to see if someone else is waiting for it
- Interrupts
  - Kernel is safe from preemption by other processes but not interrupts
  - Interrupt handler may find kernel data structures in an inconsistent state
  - Block interrupts while accessing critical data structures by raising the IPL to access critical regions
  - Interrupts require rapid servicing, so critical regions should be few and brief
  - The only interrupts that must be blocked are the ones that manipulate data in the critical region (disk interrupts)
  - Two different interrupts can have the same priority level
- Multiprocessors
  - Two processes may execute in kernel mode on two different processors
  - Data structures must be locked
– Locking mechanism must be safe across multiple processors

Process scheduling

• CPU time allocated to processes by a scheduler
• Preemptive round-robin scheduling, with a fixed quantum time of 100ms
• A higher priority process preempts the current process, except in kernel mode, before the current process has completed its quantum
  – Process priority is based on nice value and usage factor
  – Users can change the priority by changing the nice value using nice(2) system call
  – Usage factor is a measure of recent CPU usage for the process
  – While a process is not running, the kernel periodically increases its priority
  – When a process receives some CPU time, the kernel decreases its priority
  – This scheme prevents starvation of any process
• Kernel priorities are higher than user priorities
  – Scheduling priorities are integers between 0 and 127, with 0 to 49 being kernel priorities
  – Smaller integers imply higher priority
  – Kernel priorities are not variable and depend on the reason for blocking (sleeping priorities)

Signals

• Used to inform processes of asynchronous events and to handle exceptions
• Explicitly sent using kill(2)
• Each signal has a default response, possibly to terminate the process
• With a user-specified signal handler, other actions are possible
• A process may also choose to ignore the signals, or block it temporarily

New processes and programs

• fork(2) and exec()
  – fork creates a new process
  – Child is almost an exact clone of the parent process
  – Child begins user mode execution by returning from fork
  – exec overlays a new program on existing process and does not return, unless it fails
    * Child returns to user mode with its pc to the first executable instruction of new program
  – Why not do both fork and exec in a single system call?
    * A process may fork many processes that do the same thing as the parent; think of daemons
    * A process may want to exec a different program without forking
• Process creation – Number of tasks are performed by fork such as
- Reserve swap space for child’s data and stack
- New PID and proc for child
- Initialize child’s proc
- Allocate address translation maps
- Allocate child’s u area and allocate it from parent
- Update u area to refer to the new address maps and swap space
- Add the child to the set of processes sharing the text region of the program being executed by parent
- Duplicate parent’s data and stack regions and update child’s address maps to refer to these new pages
- Get references to shared resources (open files, current working directory)
- Initialize the child’s hardware context by copying from parent’s registers
- Make child runnable and put it on scheduler queue
- Arrange for the child to return from fork with a value of zero
- Return the PID of child to parent

• Fork optimization
  - Wasteful to make a copy of parent’s address space
  - Copy-on-write
    - Data and stack pages of parent are temporarily made read-only and marked as copy-on-write
    - Child gets its own copy of address translation maps but shares the actual pages
    - Attempt on page modification (by parent or child) cause a page fault exception because of page being read-only; page fault handler in kernel makes a writable copy of the page
    - If child execs or exits, pages revert to original protection and copy-on-write flag is cleared
  - vfork(2) – Virtual memory efficient fork
    - Used in BSD
    - Useful if the child is to call exec shortly after fork
    - Parent loans the address space to child and blocks until the child returns space borrowed from parent
    - Efficient because no copying takes place
    - Dangerous because it permits the modification of a process’ address space by another process

• Invoking a new program
  - exec gives the process a new address space and loads it with contents of the new program
  - Process resumes at the entry point of the new program
  - Process address space components
    - Text – executable code
    - Initialized data
    - Uninitialized data, or block static storage (bss) section – Data variables declared but not initialized
    - Shared memory
    - Shared libraries – Dynamically linked libraries
    - Heap – Dynamically allocated memory (brk(2), sbrk(2), malloc(3))
    - User stack
  - Executable file formats
    - a.out
      - Oldest executable format
      - 32-bit header, followed by text and data sections, and symbol table
· Header contains size of text, initialized data, and uninitialized data sections, and program entry point
· Header also contains a magic number

- **exec** system call
  * Parse pathname and access the executable
  * Verify execute permission
  * Check that it is valid executable
  * Account for **SUID** and **SGID** bits if set
  * Copy arguments and environment into kernel space
  * Allocate swap space for data and stack
  * Free old address and swap space
  * Allocate address maps for new text, data, and stack
  * Set up new address space
  * Copy arguments and environment variables back into user stack
  * Reset all signal handlers to default actions
  * Initialize the hardware context

- **Process termination**
  - Performed by `exit(2)`
    * Turn off all signals
    * Close all open files
    * Release text file and other resources (current directory)
    * Write to accounting log
    * Save resource usage statistics and exit status in **proc**
    * Change state to **SZOMB**, and put **proc** on zombie process list
    * Give the children of the process to **init**
    * Release address space, **u** area, address translation maps, and swap space
    * Notify the parent by sending a **SIGCHLD** signal
    * Wake up the parent if it is sleeping
    * Call `swtch()` to schedule a new process
  - The **proc** structure is freed by parent after picking up the exit status

- **Awaiting process termination**
  - Done by `wait()` system call

- **Zombie processes**
  - Every process becomes a zombie before being cleaned up by parent
  - Zombies cannot be killed by sending a signal