Semester 1, Year 2
2019
Esther Shi

Operating Systems

Lecture 1: Introduction, Processes, and Threads

Tasks of OS:
- Offer a common interface from hardware to applications (with a hardware abstraction layer)
- Regulate/manage the interaction with hardware (eg batch processing)
- Manage the execution of (multiple) active apps, called **processes**
- Facilitate the communication (and synchronization) between processes among themselves, and between the OS and processes
- Memory management (both permanent and transient, eg solid state and RAM)

- Conditions for OS: reliability, performance, security, energy efficiency

An OS consists of a program, the **kernel**, that manages the execution of apps and connects apps with the hardware. Components:
- **Scheduler**: switch between processes
- **Memory management**: translation of memory addresses, keeping track of usage
- **File system**: how files and directories are stored on disk
- **Drivers**: software that knows how to manage hardware components

Reliability
- Definition: 'kernel' is the part of OS that runs with elevated permissions (in kernel mode)
- User mode:
    - Restrictions on access to resources and memory
    - Only a subset of hardware instructions are allowed
- Kernel mode
    - Full access to resources and memory
    - All hardware instructions are allowed
- If a program crashes, the system doesn't crash

System call = a way for programs to interact with the OS. Provides an interface between a process and OS to allow user-level processes to request services of the OS. Only entry points into the kernel system.
- Used for: process control, file manipulation, device management, information maintenance, communication

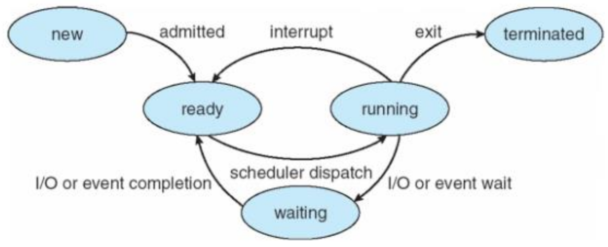
- Typically, a number is associated with each system call, number indexed in table.
- The system call interface invokes the intended system call in the OS kernel and returns the status of the system call and any return values.
- The caller doesn't need to know anything about how the system call is implemented.
- It just needs to obey the API and understand what the OS will do.

Computer program = is a passive collection of instructions.
Process = the actual execution of computer program instruction. Can be assigned and executed on a specific processor.

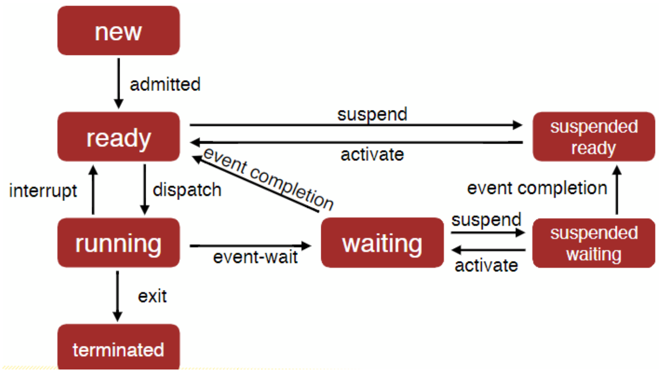
- Process consists of: hardware instructions, current program state, assigned system resources
- Each CPU (core) executes a single task at a time.
- Processes are independent of each other (isolated)
- Every process has their own isolated piece of memory
- Administration of processes: overview of all processes that currently exist in the OS
- Info on processes stored in a **process control block (PCB)**, contains the info to pause and restart a process, without the process itself knowing directly.

Process control block (PCB)
- *Process state* – running, waiting, etc
- *Program counter* – location of instruction to next execute
- *CPU registers* – contents of all process-centric registers
- *CPU scheduling information* – priorities, scheduling queue pointers
- *Memory-management information* – memory allocated to the process
- *Accounting information* – CPU used, clock time elapsed since start, time limits
- *I/O status information* – I/O devices allocated to process, list of open files

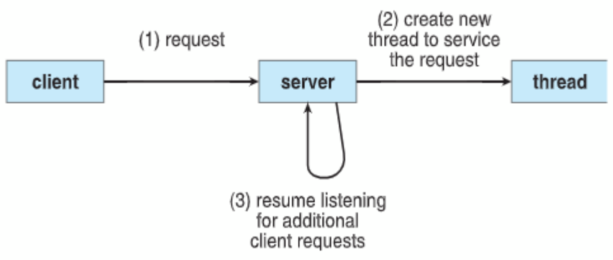
Process state
- *new*: The process is being created
- *running*: Instructions are being executed
- *waiting*: The process is waiting for some event to occur
- *ready*: The process is waiting to be assigned to a processor
- *terminated*: The process has finished execution



Context switches
- Switching the running process on a processor/core implies switching the process context (PCB) → time consuming
- Store the whole context of the old process, and restore the whole context of the new process
- fork(): process creation
- pipe(): communication between processes



Threads
- Threads run within apps
- Multiple tasks with the app can be implemented by separate threads
- Process creation is heavy-weight while thread creation is light-weight
- Simplify code, increase efficiency
- Kernels are generally multithreaded
- Creating single-threaded processes each time is very inefficient


client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests
Multithreaded server architecture

Benefits multithreading

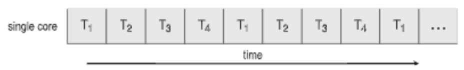
- *Responsiveness* – may allow continued execution if part of process is blocked, especially important for user interfaces
- *Resource Sharing* – threads share resources of process, easier than shared memory or message passing
- *Economy* – cheaper than process creation because allocating memory and resources is costly, thread switching has a lower overhead than context switching
- *Scalability* – process can take advantage of multiprocessor architectures

Challenges multithreading

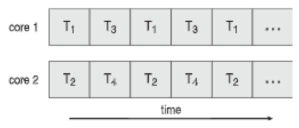
- *Identifying tasks* - Examine applications where areas can be divided into separate concurrent tasks
- *Balance* - Tasks may not be of equal value
- *Data splitting* - Data must be divided to run on separate cores
- *Data dependency* - Tasks may depend on data from other tasks
- *Testing and debugging* - Many possible execution paths make testing and debugging more difficult

- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
    - Single processor/core, **scheduler** providing concurrency

Concurrent execution on single-core system:



Parallelism on a multi-core system:


core 1
T_1 T_3 T_1 T_3 T_1 …
core 2
T_2 T_4 T_2 T_4 T_2 …
time

Data parallelism

- Distribute subsets of the same data across multiple computing units
- Perform the same operation on each core

Task parallelism

- Distributing threads across cores, each thread performing unique operation
- <u>Example</u>: each thread performs a unique statistical operation on the whole array

Amdahl's Law = identifies performance gains from adding additional cores to an app that has both serial and parallel components

- S is the serial portion
- N processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

User threads: management done by user-level threads library, above the kernel

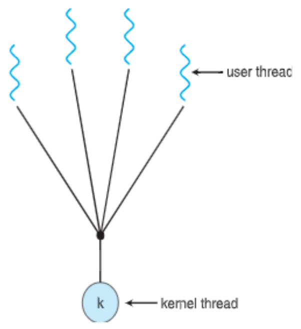
- Example: Windows threads, Java threads

Kernel threads: supported by the kernel, managed directly by the operating system

- User threads must map onto kernel threads
- <u>Examples</u>: Windows, MacOS

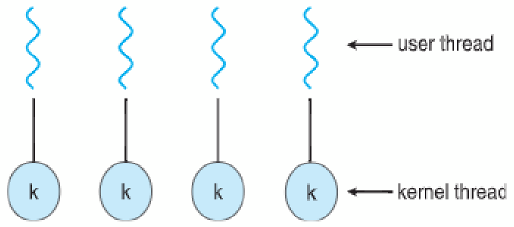
Multithreading model: many-to-one model

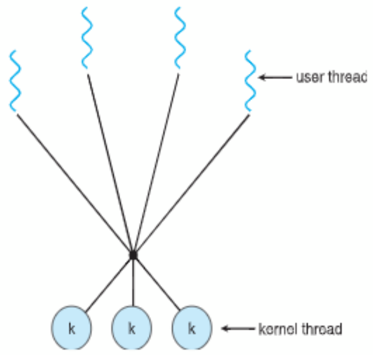
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this mode



Multithreading model: one-to-one model

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Example: Windows, Linux





Multithreading model: many-to-many model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads

Implicit threading

- As numbers of threads increase, program correctness is more difficult to achieve with explicit creation of threads
- Creation and management of threads transferred to compilers and run-time libraries rather than programmers
- Threads are handled **implicitly**
- Three important methods
    - Thread Pools
    - OpenMP
    - Grand Central Dispatch

Thread pools

- Create a number of threads in a pool where they await work
- Advantages:
    - Usually faster to service a request with an existing thread than create a new thread
    - Allows the number of threads in the application(s) to be bound by physical memory, heuristically (number of CPUs), or dynamically
- Threads not necessarily created at the time a task (request) comes in
- Separation allows for different strategies for running task, for instance tasks may be executed after a time delay or periodically

OpenMP

- Set of compiler directives and an API
- Provides supports for parallel programming in shared-memory environments
- Identifies **parallel regions**: blocks of code that can run in parallel
- App developers insert compiler directives into their code at these parallel regions

Semantics of fork() and exec()

- *exec()* is a functionality of an operating system that runs an executable file in the context of an already existing process, replacing the previous executable.
- *fork()* is an operation where by a process creates a copy of itself. It is usually a system call, implemented in the kernel.

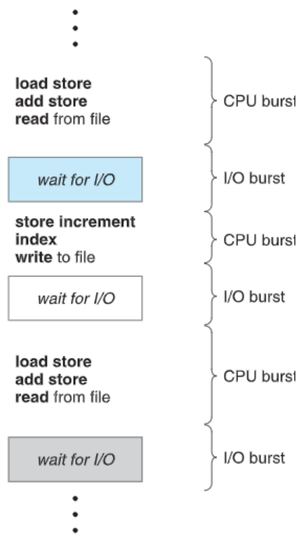
Signal handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- Signal processing:
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled by default or user-defined signal handlers:
- Every signal has a default handler that the kernel runs when handling the signal
  - User-defined signal handler can override default
- Synchronous signals:
  - illegal memory access,
  - division by 0
  - deliver the signal to the thread to which the signal applies
- Asynchronous signals:
  - generated by external events
  - termination by specific key stroke
  - deliver the signal to every thread in the process
- Signal deliverance: assign a specific thread to receive all signals for the process

Thread cancellation: terminating a thread before it has finished

- Thread to be cancelled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Problems
  - In asynchronous cancellation, not all resources may be reclaimed
  - Cancellation only occurs after target thread has checked whether or not it should be cancelled

Lecture 2: Scheduling and start of Synchronization



- Maximum CPU utilization obtained with multiprogramming
- **CPU-I/O Burst Cycle**: process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- Large number of short CPU bursts and small number of long CPU bursts

CPU scheduler

- **Short-time scheduler** selects processes in *ready* queue, and allocates CPU
- CPU needs to schedule when process
  1. Switches from *running to waiting* state
  2. Switches from *running to ready* state
  3. Switches from *waiting to ready*
  4. Terminates

- Scheduling under 1 and 4 is **nonpreemptive**
- All other scheduling is **preemptive**
    - Consider access to shared data
    - Consider preemption while in kernel mode
    - Consider interrupts occurring during crucial OS activities

- **Preemptive**: can interrupt a process and schedule something else, you don't wait until it's waiting or interrupted
- **Non-preemptive**: can't interrupt it

Dispatcher gives control of the CPU to process selected by short-term scheduler
- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

- **Dispatch latency**: time it takes for the dispatcher to stop one process and start another running

Scheduling criteria
- *CPU utilization*: keep the CPU as busy as possible (max)
- *Throughput*: no. of processes that complete their execution per time unit (max)
- *Turnaround time*: amount of time to execute a particular process (min)
- *Waiting time*: amount of time a process waits in the ready queue (min)
- *Response time*: from submission of request until first response (optimize the average measure)

First-Come, First-Server Scheduling (FCFS)
- Non-preemptive
- The first process gets served first
- **Convoy effect**: short process stuck behind long process

Shortest-Job-First Scheduling (SJF)
- Associate length of its next CPU burst to processes and schedule process with shorts time
- SJF is optimal – min average waiting time for set of processes

Length of next CPU burst
- Estimate the length, should be similar to the previous one
    - Then pick process with shortest predicted next CPU burst

- Can be done by using the length of previous CPU bursts, using exponential averaging
  1. t_n = actual length of n^{th} CPU burst
  2. τ_{n+1} = predicted value for the next CPU burst
  3. α, $0 \leq \alpha \leq 1$
  4. Define: $\tau_{n+1} = \alpha t_n + (1- \alpha) \tau_n$

- Commonly, α set to ½
- a=0, recent history doesn't count
- a=1, only the actual last CPU burst counts
- Preemptive version called **shortest-remaining-time-first**

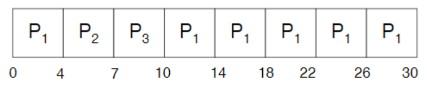
Priority scheduling

- Associate priority number with each process
- CPU allocated to process with highest priority (smallest number = highest priority)
- Process with higher priority arrives at ready queue
    - Preemptive: preempt CPU
    - Non-preemptive: put new process to head of ready queue
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- **Starvation**: low priority processes may never execute
- **Aging**: as time progresses increase the priority of the process

Round-Robin Scheduling (RR)

- Each process gets a time quantum q (unit of CPU time)
- Ready queue as FIFO queue
- After q has elapsed, process is preempted and added to the end of the ready queue
- Timer interrupts every quantum to schedule next process
- Performance:
    - q large → FIFO
    - q small → q should be large compared to context switch time, otherwise
    - overhead is too high
- Typically higher average turnaround than SJF, but better response

Process	**Burst Time**
P_1	24
P_2	3
P_3	3



Multilevel Queue Scheduling

- Partition of processes in groups with different response time requirements: foreground, background
- Multilevel queue partitions ready queue into separate queues

- Each queue has its own scheduling algorithm:
    - foreground – RR (better response time)
    - background – FCFS (less context switches)

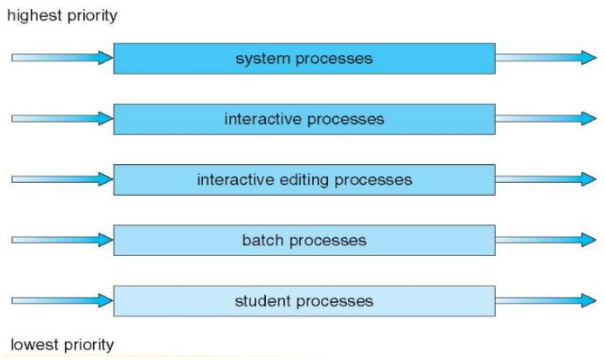
Preemptive, starvation

- Scheduling between the queues:
    - **Fixed priority scheduling** (first foreground then background)
    - **Time slice** – each queue gets a certain amount of CPU time which it can schedule amongst its processes

Scheduling Algorithm Evaluation

- First, define evaluation criteria
    - Maximize CPU utilization with threshold on maximal response time
    - Maximize throughput (# of processes that complete execution per time unit)

- **Deterministic modeling**
    - **Analytic evaluation**: takes predetermined workload and defines the performance of each algorithm for that workload
    - <u>Problem</u>: requires exact input numbers, evaluation applies only to those cases



Multilevel Queue Scheduling

- **Simulation**
    - Program model of the computer system
    - Random number generator to generate processes
    - May be expensive, accuracy at the cost of computing time

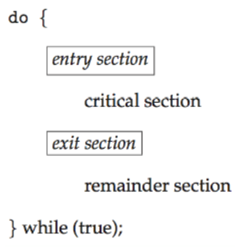
- **Implementation**
    - Program the scheduling algorithm and test it in the operating system
    - Accurate but very costly
    - Changing environment

Synchronization
- Processes may execute *concurrently*: may be interrupted at any time, partially completing execution
- Concurrent access to shared data result in *data inconsistency*
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

Lecture 3: Synchronization and Deadlocks

Producer-consumer problem
- Producer process produces information that is consumed by consumer
- Use shared memory for information
- (Bounded) buffer of items that can be filled by producer and emptied by consumer
- Access must be synchronized
- Introduce an integer `counter` that keeps track of the number of items in the buffer
- Initially, `counter` is set to 0. It is incremented by the producer after it produces a new item and is decremented by the consumer after it consumes an item

Critical Section Problem
- Consider system of n processes {p_0, p_1, ... p_{n-1}}
- Each process has **critical section** segment of code
    - Common variables, updating table, writing file, ...
    - When one process in critical section, no other may be in its critical section
- Critical section problem is to design a protocol that solves the problem
- Each process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section


```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Requirements for solution of CSP
- **Mutual Exclusion**: if process P_i is executing in its critical section, no other processes can be executing in their critical sections
- **Progress**: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- **Bounded Waiting**: A bound must exist 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

Peterson's solution
- Two process solution sharing two variables: int turn and Boolean flag
- Turn indicates whose turn it is to enter the critical section
- Flag indicates if a process is ready to enter the critical section.

Synchronization hardware

- Many systems provide hardware support for implementing critical section code.
- **Locking**: protecting critical regions via locks
- Modern machines provide special **atomic** (non-interruptible) hardware instructions

Software solutions to CSP

- Hardware-based solutions inaccessible
- OS designers build software tools to solve critical section problem: mutex locks, semaphores

Mutex Locks

- Simple software solution to critical-section problem
- Protect critical regions by first `acquire()` a lock then `release()` it
  - Boolean variable indicating if lock is available or not

- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions

<u>Problem?</u>

- This solution requires **busy waiting**: while process is in critical section, other process must loop continuously. Therefore called a **spinlock**.
- Wastes CPU cycles while spinning, but no context switch required.

<u>How realized on multiprocessor systems?</u>

- One thread can spin on one processor, while another thread performs critical section.

Semaphores

- Semaphore S – integer variable (number of resources available)
- Two standard operations modify S: `wait()` and `signal()`
- Can only be accessed via two indivisible (atomic) operations

```
wait (S) {
   while (S <= 0)
         ; // busy wait
   S--;
}
signal (S) {
   S++; }
```

Semaphore usage

- *Counting semaphore*: integer value can range over an unrestricted domain
- *Binary semaphore*: integer value can range only between 0 and 1 (same as mutex lock)
- Can solve various synchronization problems
- Consider P1 and P2 that require S1 to happen before S2. Create a semaphore "synch" initialized to 0

```
P1:                         P2:
      S1;                         wait(synch);
      signal(synch)               S2;
```

Semaphore implementation
- Must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the wait()and signal() codes are placed in the critical section
- Could now have busy waiting in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied

Semaphore with no busy waiting
- Associate a waiting queue to each semaphore
- Each entry in the waiting queue has two data items: value and pointer (to next record in list)
- Transfer control to the CPU scheduler
    - *Block* – place the process invoking the operation in the waiting queue → better than spinning
    - *Wakeup* – remove one of processes from the waiting queue and place it in the ready queue

Deadlock and starvation
- **Deadlock**: two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- **Starvation**: A process may never be removed from the semaphore queue in which it is suspended

Approaches to handle deadlocks
- Ensure that the system will never enter a deadlock state:
    - Deadlock prevention
    - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system

Deadlock conditions
- *Mutual exclusion*: only one process at a time can use a resource
- *Hold and wait*: a process holding at least one resource is waiting to acquire additional resources held by other processes
- *No preemption*: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- *Circular wait*: a set of processes are each waiting for each other, creating a cycle.

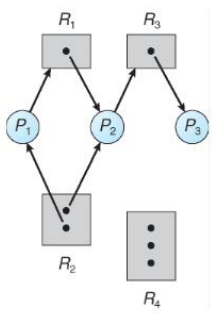
Resource-allocation graph
- Set of vertices V and a set of edges E
- V has two types:

o P = {P1, P2, … , Pn}	all the processes in the system
o R = {R1, R2, …, Rm}	all resource types in the system

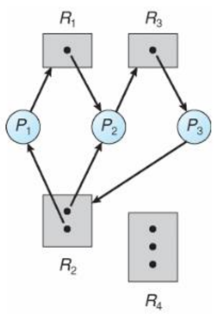
- Request edge: directed edge Pi → Rj
- Assignment edge: directed edge Rj → Pi

Deadlock avoidance: A priori information needed!
- Simple: each process declare the maximum number of resources of each type that it may need
- Ensure that there can never be a circular-wait condition
- Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

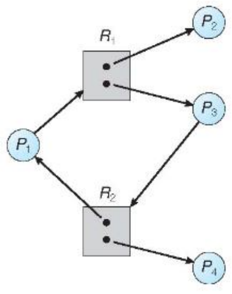
Safe state
- If a system is in safe state (no cycles) → no deadlocks
- If a system is in unsafe state (cycle(s)) → possibility of deadlock
  - If only one instance per resource type → deadlock
  - If several instances per resource type → possibility of deadlock
- Avoidance → ensure that a system will never enter an unsafe state



no deadlock



cycle —> deadlock



cycle —> no deadlock

Deadlock prevention
- *Mutual Exclusion*: not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- *Hold and Wait*: guarantee that whenever a process requests a resource, it does not hold any other resources
- *No Preemption*: if a process requests a resource that cannot be immediately allocated, all its other resources currently being held are released (process is preempted)
- *Circular Wait*: impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlock avoidance via resource-allocation graph
- Claim edge Pi → Rj indicates that process Pj may request resource Rj; represented by a dashed line
- Process request resource: claim edge → request edge
- Resource allocated to process: request edge → assignment edge
- Resource released by process: assignment edge → claim edge
- Resources must be claimed a priori in the system
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle

Lecture 4: Memory

Banker's Algorithm

- Multiple instances
- Each process must a-priori claim its max use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Data structure for the Banker's Algorithm

- n = number of processes
  m = number of resources types
- **Available**: vector of length m. If available[j] = k, there are k instances of resource type R_j available.
- **Max**: n x m matrix. If Max[i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: n x m matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: n x m matrix. If Need[i,j] = k, then P_i may need k more instances of R_j to complete its task
- Need[i,j] = Max[i,j] – Allocation[i,j]

Memory management strategies

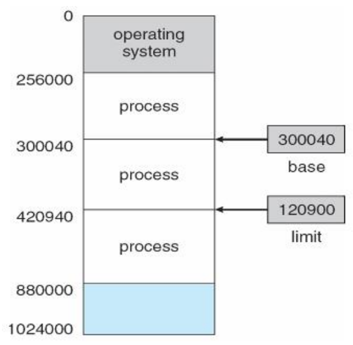
Background, swapping, contiguous memory allocation, segmentation, paging, structure of the page table

Background

- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- *Register* access in one CPU clock (or less)
- Main memory can take many cycles, causing a **stall**
- *Cache* sits between main memory and CPU registers (fast memory)
- Memory unit only sees a stream of addresses + read requests, or address + data and write requests
- *Protection* of memory required to ensure correct operation -> each process gets its own memory space
    - Range of legal addresses that each process may access

Base and limit registers

- A pair of base and limit registers define the logical address space
- CPU must check every memory access generated in user mode to be sure it is between base and limit for that user
- Hardware address protection:
    - CPU hardware compares every address generated in user mode with the registers
    - Only operating system can load the base and limit registers



Address binding

- Addresses represented in different ways at different stages of a program's life
    - Source code addresses usually symbolic
    - Compiled code addresses bind to relocatable addresses (relative addresses)
    - Linkage editor (Linker) or loader will bind relocatable addresses to absolute addresses
    - Each binding maps one address space to another

Binding of Instructions and Data to Memory
Address binding of instructions and data to memory addresses can happen at 3 different stages:
- **Compile time**: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
- **Load time**: Must generate relocatable code if memory location is not known at compile time
- **Execution time**: Binding delayed until run time if the process can be moved during its execution from one memory segment to another

Logical vs. Physical address space
- The concept of a logical address space that is bound to a separate physical address space is central to proper memory management
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme
- **Logical address**: generated by the CPU; aka virtual address.
  Logical address space is the set of all logical addresses generated by a program
- **Physical address**: address seen by the memory unit
  Physical address space is the set of all physical addresses generated by a program

Memory-Management Unit (MMU)
- Hardware device that maps virtual address to physical address at run time
- Base register now called **relocation register**
- Simple scheme: The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with logical addresses; it never sees the real physical addresses
    - Execution-time binding occurs when reference is made to location in memory
    - Logical address bound to physical addresses

Dynamic loading
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- All routines kept on disk in relocatable load format
- Useful when large amounts of code are needed to handle infrequently occurring cases
- No help from OS

Dynamic linking
- **Static linking**: system libraries and program code combined by the loader into the binary program image
- **Dynamic linking**: linking postponed until execution time (system libraries)
- Small piece of code, **stub**, used to locate the appropriate memory-resident library routine
- Stub replaces itself with the address of the routine, and executes the routine
- Operating system checks if routine is in processes' memory address
    - If not in address space, add to address space

Swapping
- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
- Total physical memory space of processes can exceed physical memory
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a ready queue of ready-to-run processes which have memory images on disk
- Context switch time can be high
- Reduce size of memory swapped by knowing how much memory being used

Contiguous allocation
- Partition main memory into two partitions:
    - Resident operating system, usually held in low memory with interrupt vector
    - User processes held in high memory
    - Each process contained in single contiguous section of memory

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
    - Base register contains value of smallest physical address
    - Limit register contains range of logical addresses – each logical address must be less than the limit register

- **Multiple-partition allocation**
    - Degree of multiprogramming limited by number of partitions
    - Variable-partition sizes for efficiency (sized to a given process' needs)
    - **Hole**: block of available memory; holes of various size are scattered throughout memory
    - When a process arrives, it is allocated memory from a hole large enough to accommodate it
    - Process that exits frees its partition, adjacent free partitions combined

Dynamic Storage-Allocation Problem
- **First-fit**: Allocate the first hole that is big enough
- **Best-fit**: Allocate the smallest hole that is big enough; must search entire list, unless ordered by size
    - Produces the smallest leftover hole
- **Worst-fit**: Allocate the largest hole; must also search entire list
    - Produces the largest leftover
- First-fit and best-fit better than worst-fit in terms of speed and storage utilization

Fragmentation
- **External Fragmentation**: total memory space exists to satisfy a request, but it is not contiguous
- **Internal Fragmentation**: allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- Reduce external fragmentation by **compaction**
    - Shuffle memory contents to place all free memory together in one large block
    - Compaction is possible only if relocation is dynamic, and is done at execution time

Segmentation
- Memory-management scheme that supports user view of memory
- A program is a collection of segments, segment is a logical unit (main program, procedure, etc.)
- Logical address consists of a two tuple: <segment-number, offset>
- **Segment table**: maps two-dimensional physical addresses; each table entry has:
    - **Base**: contains the starting physical address where the segments reside in memory
    - **Limit**: specifies the length of the segment
- **Segment-table base register (STBR)** points to the segment table's location in memory
- **Segment-table length register (STLR)** indicates number of segments used by a program;
    - Segment number s is legal if s < STLR

Paging

- Physical address space of a process can be noncontiguous
    - Avoids external fragmentation and varying sized memory chunks
- Divide physical memory into fixed-sized blocks called frames (Size is power of 2)
- Divide logical memory into blocks of same size called pages
- Keep track of all free frames
- To run a program of size N pages, need to find N free frames and load program
- Set up a **page table** to translate logical to physical addresses
- Backing store likewise split into pages
- (Still internal fragmentation)

Memory protection

- Memory protection implemented by associating protection bit with each frame to indicate if read-only or read-write access is allowed
- **Valid-invalid** bit attached to each entry in the page table:
    - "valid" indicates that the page is in the process' logical address space, and is a legal page
    - "invalid" indicates that the page is not in the process' logical address space
    - Or use page-table length register **(PTLR)**
- Any violations result in a trap to the kernel

Shared pages

- *Shared code*
    - One copy of read-only (reentrant) code shared among processes
    - Similar to multiple threads sharing the same process space
    - Also useful for interprocess communication if sharing of read-write pages is allowed
- *Private code and data*
    - Each process keeps a separate copy of the code and data
    - The pages for the private code and data can appear anywhere in the logical address space

Page tables

- *Hierarchical page tables*
    - Break up the logical address space into multiple page tables
    - A simple technique is a two-level page table
    - Page the page table
- *Hashed page tables*
    - The virtual page number is hashed into a page table
    - Virtual page numbers are compared in this chain searching for a match
- Inverted page tables

Lecture 5: Memory & File Systems and I/O Systems

Virtual-Memory Management: Background

- Code needs to be in memory to execute, but entire program rarely used
- **Virtual memory**: separation of user logical memory from physical memory
    - Only part of the program needs to be in memory for execution
    - Logical address space can be much larger than physical address space
    - Allows address spaces to be shared by several processes
    - More efficient process creation, more programs running concurrently
    - Less I/O needed to load or swap processes
- Implemented via: demand paging, demand segmentation

Shared library using virtual memory
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space

Demand Paging
- Naïve: bring entire process into memory at load time
- Alternative: bring a page into memory only when it is needed
    - Less I/O needed, less memory needed, faster response, more users
- Page is needed => reference to it
    - Invalid reference => abort
    - Not-in-memory => bring to memory
- Lazy swapper: never swaps a page into memory unless page will be needed
    - **Pager** = swapper that deals with pages

Valid-Invalid Bit
- With each page table entry a valid-invalid bit is associated
  (v → in-memory – **memory resident**, i → not-in memory)
- Initially valid-invalid bit is set to i on all entries

Page fault
- If there is a reference to a page, check whether reference is valid or invalid
1. Invalid reference → **abort**, or just not in memory
2. Swap page into frame via disk operation (find a free page)
3. Reset tables to indicate page now in memory
   set validation bit = v
4. Restart the instruction that caused the page fault

Aspect of demand paging
- Pure demand paging: extreme case – start process with *no* pages in memory
    - OS sets instruction pointer to first instruction of process, non-memory-resident → page fault, as for every other process pages on first access
- A given instruction could access multiple pages → multiple page faults
    - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
    - Page table with valid / invalid bit
    - Secondary memory (swap device with **swap space**)
    - Instruction restart
- No free frame? Page replacement – find some page in memory, but not really in use, page it out

Basic page replacement
- Find the location of the desired page on disk
- Bring the desired page into the (newly) free frame; update the page and frame tables
- Continue the process by restarting the instruction that caused the trap

Page and frame replacement algorithms
- **Frame-allocation algorithm**: how many frames to give each process, which frames to replace
- **Page-replacement algorithm**: wants lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
    - String is just page numbers, not full addresses
    - Repeated access to the same page does not cause a page fault
- **First-in-first-out (FIFO) algorithm**
- **Optimal algorithm**: replace page that will not be used for longest period of time
- **Least Recently Used (LRU) algorithm**: replace page that hasn't been used in most amount of time

File operations
- *Create*: allocate space for the file, create entry in the file directory
- *Write*: at write pointer location
- *Read*: at read pointer location

Open file locking
- *Shared lock*: several processes can acquire concurrently
- *Exclusive lock*
- *Mandatory*: access is denied depending on locks held and requested
- *Advisory*: processes can find status of locks and decide what to do

- *Sequential access*: one record after the other
- *Direct access*: file is fixed length logical records

Single-level directory
- Users may give the same name to different files
- Grouping only possible via filenames

Two-level directory
- Grouping only possible via filenames
- Users isolated

- Tree structured directory
- Acylic-graph directories

File system mounting
- A file system must be **mounted** before it can be accessed
- An unmounted file system is mounted at a mount point

Blocking and nonblocking I/O
- **Blocking**: process suspended until I/O completed
    - Easy to use and understand, Insufficient for some needs
- **Nonblocking**: I/O call returns as much as available
- **Asynchronous**: process runs while I/O executes
    - Difficult to use, I/O subsystem signals process when I/O completed

- Caching: faster device holding copy of data
- Spooling: hold output for a device
- Device reservation: provides exclusive access to a device
- Error handling: retry a read or write

Lecture 6: Protection security

Goals of protection

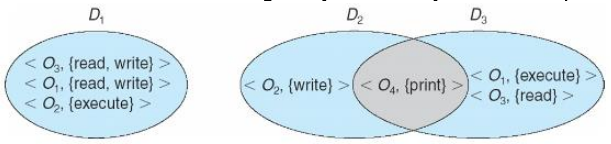
- **Protection model**: each part of the computer has a unique name and can be accessed by processes through a well-defined set of operations
- **Protection problem**: ensure that each part is accessed correctly and only by those processes that are allowed to do so

Protection principles

- Principle of **least privilege**: programs, users, and systems should be given just enough privileges to perform their tasks
- Limits damage
- *Static*: during life of system/process
- *Dynamic*: changed by process as needed – **domain switching, privilege escalation**

Protection domain

- Specifies the resources a process may access
- *Access-right*: <object-name, rights-set>
  rights-set is a subset of all valid operations that can be performed on the object
- *Domain*: set of access-rights
- **Domain switching** = dynamically enable a process to switch from one domain to another



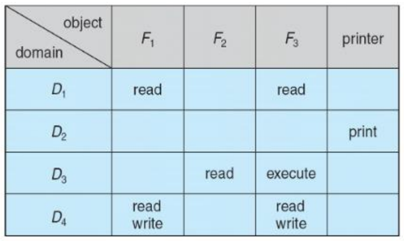
Three protection domains

Domain switching in UNIX

- *Domain* = user-id
- Domain switch via file system, passwords or commands
    - Each file is associated with a domain bit (setuid bit)
    - When file is executed and setuid = on, then user-id is set to owner of the file being executed
    - When execution completes user-id is reset

Access matrix

- Rows represent domains, columns represent objects
- Access(i,j) is the set of operations that a process executing in D_i can invoke on Object$_j$
- User who creates object can define access column for that object
- Can be expanded to **dynamic protection**
- *Copy **: allows to copy rights on objects within the same column
- *Control*: Di can modify access rights of Dj
- *Transfer*: switch from domain Di to Dj
- *Owner* of object can add and remove rights
- Domain switch: switching a process from one domain to another



object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Implementation of access matrix

- *Global table*: ordered triples
- *Access lists for objects*: column is access list for one object
- *Capability list* & *lock-key mechanism*

Categories of security violation

1. Breach of *confidentiality* - Unauthorized reading of data
2. Breach of *integrity* - Unauthorized modification of data
3. Breach of *availability* - Unauthorized destruction of data
4. Theft of service - Unauthorized use of resources
5. *Denial of service(DOS)* - Prevention of legitimate use

Methods of security violation

- *Masquerading* (breach authentication): pretending to be an authorized user to escalate privileges
- *Replay attack –* as is or with message modification
- *Man-in-the-middle attack*: intruder sits in data flow, masquerading as sender to receiver and vice versa
- *Session hijacking*: intercept an already-established session to bypass authentication (prior to man-in-the-middle-attack)

Measures of security levels

- Physical, human, operating system, network

Program Threats

- *Trojan Horse*: code segment that misuses its environment
  - Exploits mechanisms for allowing programs written by users to be executed by other users
  - Spyware, pop-up browser windows, covert channels
- *Trap Door*
  - Specific user identifier or password that circumvents normal security procedures
  - Could be included in a compiler
- *Logic Bomb*
  - Program that initiates a security incident under certain circumstances

Stack and buffer overflow

- Exploits a bug in a program (overflow either the stack or memory buffers)
- Failure to check bounds on inputs, arguments
- Attacker sends more data than the program expects
- Write past arguments on the stack into the return address on stack
- When routine returns from call, returns to hacked address
- Pointed to code loaded onto stack that executes malicious code
- Unauthorized user or privilege escalation

- Attack code can get a shell with the processes' owner's permissions, open a network port, delete files, download a program, etc
- Depending on bug, attack can be executed across a network using allowed connections, bypassing firewalls
- Buffer overflow can be disabled by disabling stack execution or adding bit to page table to indicate "non-executable" state

Viruses

- Code fragment embedded in legitimate program
- Self-replicating, designed to infect other computers
- Very specific to CPU architecture, operating system, applications
- Usually borne via email or as a macro

Cryptography

- Internal to a given computer, source and destination of messages can be known and protected
- Source and destination of messages on network cannot be trusted without cryptography
- Means to constrain potential senders (sources) and / or receivers (destinations) of messages

Encryption

- Set K of keys
- Set M of messages
- Set C of ciphertexts
- A function $E : K \rightarrow (M \rightarrow C)$
  For each $k \in K$, E_k is a function for generating ciphertexts from messages
- A function $D : K \rightarrow (C \rightarrow M)$
  for each $k \in K$, D_k is a function for generating messages from ciphertexts
- Given a ciphertext $c \in C$, a computer can computer m such that $E_k(m) = c$ only if it has k

Symmetric encryption

- Same key used to encrypt and decrypt
- k is secret

Asymmetric encryption

- Public-key encryption based on each user having two keys
  - *Public key*: published key used to encrypt data
  - *Private key*: key known only to individual user used to decrypt data
- RSA
- k_e is the public key, k_d is the private key
- N is the product of two large, randomly chosen prime numbers p and q
  (for example, p and q are 512 bits each)
- **Encryption algorithm** is $E_{ke,N}(m) = m^{ke} \bmod N$, where k_e satisfies $k_e k_d \bmod (p-1)(q-1) = 1$
- **Decryption algorithm** is then $D_{kd,N}(c) = c^{kd} \bmod N$
- Symmetric cryptography based on transformations, asymmetric based on mathematical functions
- Asymmetric much more computing intensive
- Typically not used for bulk data encryption
- Example: page 36

User authentication

- For a message m, a computer can generate an authenticator $a \in A$ such that $V_k(m, a)$ = true only if it possesses k. Thus, computer holding k can generate authenticators on messages so that any other computer possessing k can verify them
- Computer not holding k cannot generate authenticators on messages that can be verified using V_k
- Since authenticators are generally exposed (for example, they are sent on the network with the messages themselves), it must not be feasible to derive k from the authenticators
- Practically, if $V_k(m,a)$ = true then we know m has not been modified and that send of message has k
  - If we share k with only one entity, know where the message originated

Computer security classifications

- D: minimal security
- C: provides discretionary protection through auditing
  - C1: identifies cooperating users with the same level of protection
  - C2: allows user-level access control
- B: all the properties of C, however each object may have unique sensitivity labels
- A: uses formal design and verification techniques to ensure security

Semester 1, Year 2
2019
Esther Shi

Lecture 7: Linux

Components
- **Kernel**: responsible for maintaining the important abstractions of the OS
- Kernel code executes in kernel mode with full access to all the physical resources of the computer
- All kernel code and data structures are kept in the same single address space

System libraries
- Standard set of functions through which applications interact with the kernel
- Implement operating-system functionality that does not need the full privileges of kernel code
- The system utilities perform individual specialized management tasks

Kernel modules
- Sections of kernel code that can be compiled, loaded, and unloaded independent of the rest of the kernel
- The module interface allows third parties to write and distribute device drivers or file systems that could not be distributed under the GPL
- Allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in.
- Four components to Linux module support: module-management system, module loader and unloader, driver-registration system, conflict-resolution mechanism

Module management: loading modules into memory and letting them communicate with the kernel
Driver registration: allows modules to inform the kernel that a new driver is available
Conflict resolution: allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

Process management
- UNIX process management separates the creation of processes and the running of a new program into two distinct operations.
    - The fork() system call creates a new process
    - A new program is run after a call to exec()
- Under UNIX, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program
- Under Linux, process properties fall into three groups: the process's identity, environment, and context