Operating System: program that provides useful abstractions of hardware to applications

- It's a resource manager, each program gets time with the resource, each program gets space on the resource
- First generation: run one job at a time
- Second generation: batch systems: bring card, input tape, writing results to output tape
    - Original batch systems used tape drivers, later batch systems used disks for buffering which offered spooling
- Third generation: multiprogramming, multiple jobs in memory, OS protected from each and resources split between jobs
    - Timesharing: several jobs active at one time
- Types of OS: mainframe, server, personal computer, handheld, real-time, and smart card
- Disk drive structure: data stored on surface, data in concentric tracks, data read and written by heads
- Flash memory structure: divided into erase blocks, read and written in pages
- Memory: single base/limit pair: set for each process, two base/limit registers: one for program, one for data
- Device request:
    - Request sent to controller, then to disk
    - Disk responds, signals disk controller which tells interrupt controller
    - Interrupt controller notifies CPU
- Processes: program in execution, address space the program can use, state (registers, PC and SP)
    - OS keeps track of all processes in a process table
    - Processes can create other processes
    - Processes have three segments:
        - Text: program code
        - Data: program data
        - Stack: automatic variables
    - Interprocess communication: through network and pipes: A writes into pipe, and B reads from it
- System calls:
    - OS runs in privileged mode, programs want the OS to perform a service
    - User program enters supervisor mode, program passes relevant info to OS, OS performs the service if able, OS check ifo passes to ame sure it's OK
    - read(fd, buffer, length)
        - Program pushes arguments, calls library
        - Library sets up trap, calls OS
        - OS handles system call
        - Control returns to library
        - Library returns to user program
- Monolithic OS: OS is one big "program"
- Layered OS: CPU supports layers of privilege: good security
- Microkernels: processes don't share memory. Communication via message passing
- Two types of Virtual Machines managers:
    - Type1 hypervisor: runs on bare hardware, manages resources between guest OS
    - Type2 hypervisor: runs a processes in the host OS. May use host OS services

Processes and Threads

- Process:
    - Code, data, and stack
    - Program state: CPU registers, PC, SP
    - Only one process can run in a single CPU core
- Process created in two ways:
    - System initialization: one or more processes created when the OS starts up
    - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from: user request to create a new process, already running processes
- Conditions that terminate processes can be:
    - Voluntary:normal exit, error exit

    - Involuntary: fatal error, killed by another process
- Process Hierarchies :
    - Parent creates a child process and forms a hierarchy
    - If a process terminates, its children are inherited by the terminating process's parent
- Process in one of 5 states:
    - Created
    - Ready
    - Running
    - Blocked
    - Exit
- Each process has a process table entry
- What happens on a trap/interrupt?
    - 1. Hardware saves PC
    - 2. Hardware loads new PC, identifies interrupt
    - 3. Assembly language routine saves registers
    - 4. Assembly language routine sets up stack
    - 5. Assembly language calls C to run service routine
    - 6. Service routine calls scheduler
    - 7. Scheduler selects a process to run next
    - 8. Assembly language routine loads PC and register for the selected process
- Threads: "processes" sharing memory
    - Process <-> address space
    - Thread <-> PC/ stream of instructions
    - Allow a single application to do many things at once
    - Threads are faster to create or destroy
- Three ways tobuild a server:
    - Multithreaded server: parallelism, blocking system calls
    - Single threaded process: slow, no parallelism, blocking system calls
    - Finite-state machine: each activity has its own state, parallelism
- User level threads: non need for kernel support, may be slower than kernel threads, harder to do non blocking I/O
- Kernel-level threads: more flexible scheduling, nonblocking I/O, not portable

Interprocess Communication and Synchronization(IPC)

- Race conditions:
    - Cooperating processes share storage(memory)
    - Both may read and write the shared memory
- Critical regions:
    - Provide mutual exclusion and help fix race conditions
    - 4 conditions must hold:
        - 1. No two processes may simultaneously be in critical region
        - 2. No assumptions may be made about speeds or number of CPUS
        - 3. No process running outside its critical region may block another process
        - 4. A process may not wait forever to enter its critical region
- Spin lock: the waiting process "spins" in a tight loop reading the variable

- Bakery algorithm:

Bakery algorithm: code

```
while (1) {          /* i is the number of the current process */
  choosing[i] = 1;
  number[i] = max(number[0],number[1],…,number[n-1]) + 1;
  choosing[i] = 0;
  for (j = 0; j < n; j++) {
    while (choosing[j]) {          /* wait while j is choosing a */
    }                              /* number */
    /* Wait while j wants to enter and j <<< i */
    while ((number[j] != 0) &&
           ((number[j] < number[i]) ||
            ((number[j] == number[i]) && (j < i)))) {
    }
  }
  /* critical section */
  number[i] = 0;
  /* rest of code */
}
```

Busy waiting: strict alternation

Process 0

```
while (TRUE) {
    while (turn != 0) {
    }    /* loop */
    critical_region ();
    turn = 1;
    noncritical_region ();
}
```

Process 1

```
while (TRUE) {
    while (turn != 1) {
    }    /* loop */
    critical_region ();
    turn = 0;
    noncritical_region ();
}
```

-

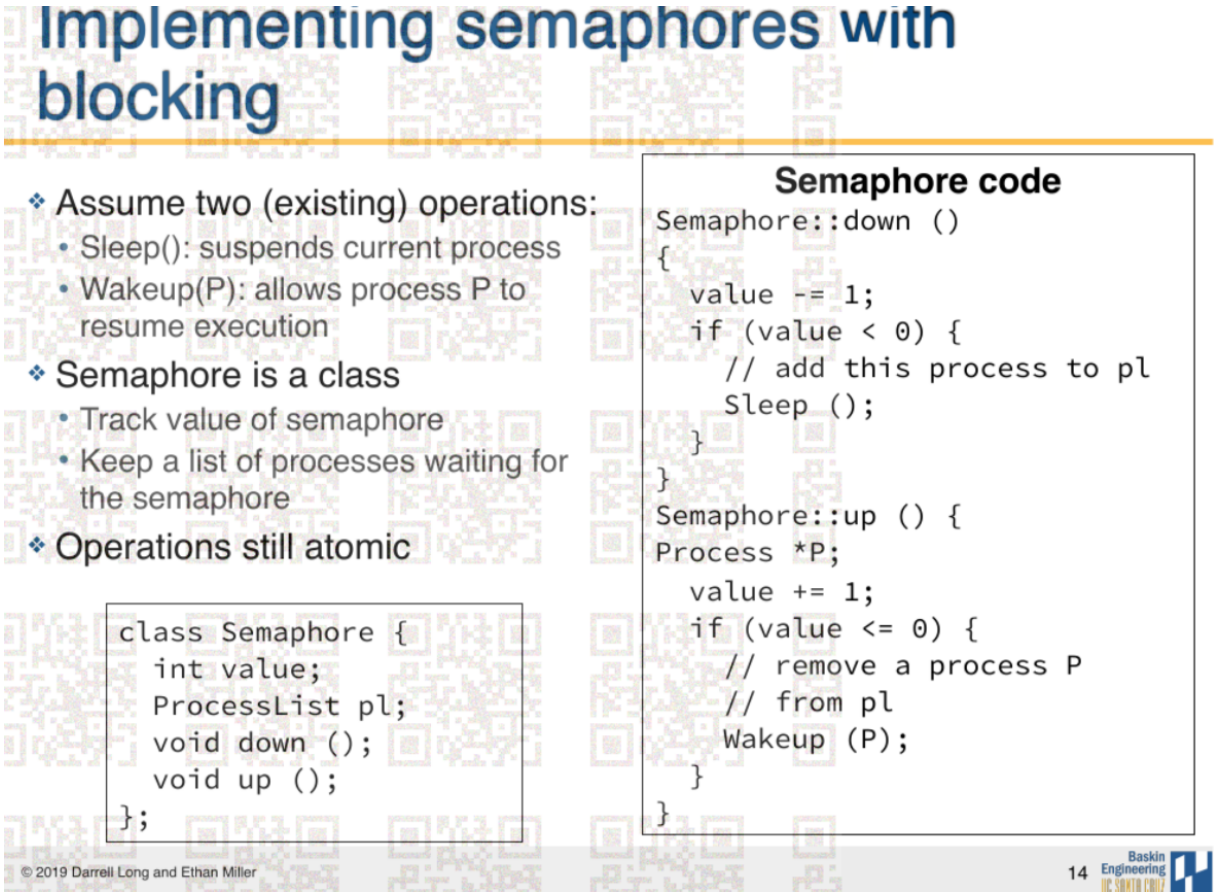
Busy waiting: working solution

```
#define   FALSE     0
#define   TRUE      1
#define   N         2     /* # of processes */
int turn;                 /* Whose turn is it? */
int interested[N];        /* Set to 1 if process j is interested */

void enter_region(int process)
{
  int other = 1 - process;    /* # of the other process */
  interested[process] = TRUE; /* show interest */
  turn = process;             /* Set it to my turn */
  while (turn==process && interested[other]==TRUE) {
  }                           /* Wait while the other process runs */
}

void leave_region (int process)
{
  interested[process] = FALSE;      /* I'm no longer interested */
}
```

- - hardware for synchronization:
    - Test and set: test a variable and set it in one instruction
    - Atomic swap: switch register and memory in one instruction
    - Turn off interrupts: process won't be switches out unless it asks to be suspended
- Mutual exclusion using hardware:
    - Single shared variable lock
    - Two versions
        - Test and set
        - Swap

- Semaphores: synchronization mechanism that doesnt require busy waiting during entire critical section
- semaphores: an integer variable to count the number of wakeup saves for future use
- down: checks to see if the value is greater than 0, if so, decrements the value and just continues. if the value is 0, the process is put to sleep without completing the dow for the moment
- up: increments the value of the semaphore addressed
    - down(s) while (s <=0({}; S <- S-1
    - up(): S <- S + 1
    - 

- Counting semaphores: value can range over an unrestricted range
- Binary semaphore: value 1 : semaphore available, value 0: process has acquired the semaphore
- Monitors: variblese foo, bar and arr are accessible only by func1 and func2, only one process can be executing in either func1 or func2 at any time
    - Use a condition variable to wait inside a monitor
        - wait():suspend this process until signaled
        - signal(): wake up exactly one process waiting on this condition vairable
- Monitors:
    - Locks (Acquire(), release())
    - Condition variables: wait(), signal()
    - Lock usage: acquiring a lock: entering a monitor, releasing a lock: leaving a monitor
- Barriers: synchronizing multiple processes: processes wait at a barrier until all in the group arrive
- Deadlock: two or more processes are waiting indefinitely for an event that can only be caused by a waiting process
- Starvation: indefinite blocking
- Livelock: a process can still run, but not make progress
- Synchronization problems:
    - Bounded buffer: multiple producers and consumers, synchronize access to shared buffer
    - Readers and writers: many processes may read and write
        - Only one writer allowed at any time
        - Many readers allowed, but no while a process is writing
    - Dining philosophers:
        - Resource allocation problem
        - N processes and limited resources to perform sequence of tasks s

Scheduling:

- Busts of CPU usage alternate with periods of I/O wait
- Processes are scheduled at the time they enter the system and at clock interrupts
- All systems:
    - Fairness: give each process a fair share of the CPU
    - Enforcement: ensure that the stated policy is carried out
    - Balance: keep all parts of the system busy
- Measuring performance:
    - Throughput: amount of work completed per second
    - Response time: when a command is submitted until results are returned
- Batch scheduling:
    - First come first served
        - Do jobs in the order they arrive
    - Shortest job first
        - To the shortest job first
        - Sorted in increasing order of execution time
    - Shortest remaining time first
        - Re-evaluate when a new job is submitted
    - Priority(non-preemptive)
- Interactive
    - Round robin
        - Gives each process a quantum to run
        - Rotate through ready processes
    - Priority(preemptive)
        - Assign a priority to each process
    - Multilevel feedback queue
        - Priority is based on 2 things
            - Resource requirements: blocked threads have higher priority when rescheduled
            - Previous CPU usage: CPU hogs have lower priority
    - Lottery scheduling
        - Give processes “tickets” for CPU time
        - Each quantum pick a ticket at random
        - Each process gets the CPU m/n of the time if the process has m of the n existing tickets
- Three-level scheduling:
    - Jobs held in input queue until moved into memory
        - Jobs moved into memory when admitted
    - CPU scheduler picks next job to run
    - Memory scheduler picks some jobs from main memory and moves them to disk if insufficient memory space
- User-level threads
    - Kernel picks a process to run next
    - Run-time system schedules threads
- Kernel-level threads
    - Kernel schedules each thread

Deadlocks:

- Resources: something a process uses
    - Processes need to access resourcesses in a reasonable order
    - Preemptable resources: can be taken away from a process with no ill effects
    - Nonpreemptable resources: will cause the process to fail if take away
- Using resources:
    - Sequence of events required to use a resource:

        - Request the resource
        - Use the resource
        - Release the resource
    - Cant use the resource id request is denied
        - Requesting process has options
            - Block and wait for resource
            - Continue without it, may be able to use an alternate resource
            - Process fails with error code
- Deadlocks occur when
    - Processes are granted exclusive access to devices or software constructs
    - Each deadlocked process needs a resource held by another deadlocked process
- Deadlock: a set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
- In a deadlock: none of the processes can
    - Run
    - Release resources
    - Be awakened

- Four conditions for deadlocks:
    1. Mutual exclusion: each resource is assigned to at most one process
    2. Hold and wait: a process holding resources can request more resources
    3. No preemption: previously granted resources cannot be forcibly taken away
    4. Circular wait: there must be a circular chain of 2 or more processes where each is wating for a resource held by the next member of the chain

- How can OS deal with deadlock:
    - Ignore the problem
        - The ostrich algorithm
    - Detect deadlocks and recover from it
        - Recover through preemption
            - Take a resource from another process
        - Recover through rollback
            - Checkpoint a process periodically
            - Use saved state to restart the process if in deadlock
        - Recover through killing processes
            - Kill one of the processes in the deadlock cycle
            - Other processes can get its resources
    - Dynamically avoid deadlock
        - Find cycles in the resource allocation graph
        - DFS at each node
    - Prevent deadlock
        - Remove at least one of the 4 necessary conditions
        - Removing mutual exclusion: spool everything
            - Spool devices
            - Avoid assigning resource when not absolutely necessary
        - Removing hold and wait: request all resources intiially
            - Require processes to request resources before starting
            - Or a process must give up all resources before making a new request
        - Remvoing non preemption: take resources away if there is not a complete set
            - Not usually aa viable option
        - Removing circular wait: order resources numerically
            - Assign an order to resources

        - Always acquire resources in numerical order
    - Starvartion: solution: first-come-first-served
    - Livelock: sometimes a process can still run but not make progress

Memory Management:

- Memory managers handles the memory hierarchy
    - Components include
        - OS
        - Single process
- Fixed partitions: multiple programs
    - Divide memory into fixed spaces
    - Assign a process to a space when its free
    - Separate input queues for each partition
- More processes -> better utilization, less time per process
- Memory needs relocations and protection for multiprogramming
- The OS must keep processes' memory separate
- Base and limit registers
    - Base: start of the processes memory partition
    - Limit: length of the processes memory partition

- Swapping:
    - Memory allocation changes as processes come into memory and processes leave memory

- Bitmaps: 1: memory used, 0: memory not used
- Linked used:
    - First fit: the first suitable hole on the list
    - Next fit: the first suitable after the previously allocated hole
    - Best fit: smallest hole that is larger than the desired region
    - Worst fite: the largest available hole
- Buddy allocation: slit larger chunks to create two smaller chunks, when a chunk is free, see if it can be combined with its buddy
- Slab allocation

Virtual memory: allow OS to hand out more memory than exists on the system

- Keeps recently used stuff in physical memory
- Move less recently used stuff to disk
- Translation of VA is done by the memory managment unit
- Multi-level page tables