Virtualization

Processes

Program: A passive collection of instructions

Process: The abstraction provided by the OS of a running program

Machine State: What a program can read and change when it is running (registers, address spaces, open files, etc.)

Process Control Block (PCB): A data structure that contains the state of a process.

Creation of A Process by OS:

- Load data from disk to memory
- Allocate space for the run-time stack and initialize the stack with arguments (i.e. fill in the parameters for argc and argv)
- Allocate memory for program's heap. Initially small, but OS may grow the heap as needed.
- Setup initial file descriptors (stdin, stdout, stderr).
- Transfer control of the CPU to the newly-created process (i.e. $\mathrm{main}()$).

Context Switch: the CPU stops running one process (or thread) and starts running another

- Process A executes
- Hardware: generates timer interrupt, save `regs(A)` to kernel stack, move to kernel mode and jump to trap handler
- OS: Handle the trap, call `switch` routine, save `regs(A)` $\rightarrow$ `proc_t(A)`, restore `regs(B)` $\leftarrow$ `proc_t(B)`, switch to `k-stack(B)`, return-from-trap (into B)
- Hardware: restore `regs(B)` from kernel stack, move to user mode and jump to process B
- Process B executes.

CPU Scheduling

Running Time Metrics:

- $T_{\text{turnaround}} = T_{\text{completion}} - T_{\text{arrival}}$
- $T_{\text{response}} = T_{\text{first run}} - T_{\text{arrival}}$

FIFO/FCFS: First Come First Served, nonpreemptive

SJF: Shortest Job First, nonpreemptive

STCF: Shortest Time to Completion First, preemptive. Always run job that will complete the quickest

MLFQ: Multi-Level Feedback Queue, preemptive

1. If Priority $(A) >$ Priority (B) then A runs
2. If Priority $(A) ==$ Priority (B) then A&B run in RR
3. Processes start at top priority
4. Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced
5. After some time period S, move all the jobs in the system to the topmost queue.

Lottery Scheduler: Randomly selects the next process to run based on ticket probabilities, giving each process CPU time proportional to its number of tickets.

- Ticket Currency: allows a user to allocate tickets among their running processes.
- Ticket Transfer: allows a process to temporarily hand off its tickets to another Process.
- Ticket Inflation: trusted processes can boost tickets to indicate its need for more CPU time.

Unfairness Metric: $U = \frac{T(\text{process1 completion})}{T(\text{process2 completion})}$

Stride Scheduler

- Each process is assigned a stride, which is the inverse proportion to the number of tickets the process has.
- Every time a process runs, its pass value is incremented by its stride.
- Scheduler selects the process with the smallest pass value.

The Linux Completely Fair Scheduler (CFS):

- Divide a time length evenly among n processes.
- Each process has a `vruntime` and a `nice` value.
- The process with the smallest `vruntime` is selected to run.
- Update `vruntime` of the running process by

$$\texttt{vruntime}_i \mathrel{+}= \frac{\texttt{weight}_0}{\texttt{weight}_{\texttt{nice}_i}} \times \texttt{runtime}_i$$

- Ready jobs' `vruntime` are kept in a red-black tree.
- When jobs wake up, their `vruntime` is set to the minimum value in the tree.

Virtualiziang Memory

Transparency: Process is unaware of sharing

Static Allocation: Randomlyewrites each program as it is loaded and placed in memory

- No Protection
- Cannot move addresses space after it has been placed.

Dynamic Allocation: Allocates memory at run-time

- Requires hardware support (Memory Management Unit)
- MMU dynamically changes process address at every memory reference

Sparse Allocation: Allocates memory in chunks, only allocate physical memory when needed.

Base+Bounds:

- MMU compares logical address to bounds register.
- If logical address is out of bounds, raise an error.
- Otherwise, add base register to logical address to get physical address.
- OS sets registers when loading process.
- Process can be moved by updating its base register.

Running Process with Base+Bounds:

- OS: allocate memory in process table, alloc memory for process, set base and bounds registers, then return from trap.
- Hardware: Restore registers, move to user mode, jump to process's **PC** (stores the next instruction address).
- Process A: fetch instruction
- Hardware: translate VA, perform fetch.
- Process A: execute instruction.
- Hardware: if explicit load/store, ensure address is legal and translate the VA.

Segmentation: Divide the address space into segments (Code, Stack, Heap), each segment has separate base+bounds registers and grows independently.

- Explicit Approach: top bits of address select the segment, remaining are the offset.
- Implicit Approach: entire logical address is the offset, the corresponding segment is determined by how logical address is formed:
  - Formed from PC (Program Counter): code segment.
  - Formed from SP (Stack Pointer): stack segment.
  - Anything else: heap segment.

Segment	Base	Size	GrowsPositive?	Protection
00	32 K	2 K	1	R-X
01	34 K	3 K	1	R-W
11	28 K	2 K	0	R-W

Table 1: Segment Register

Paging

Divide virtual and physical memory into fixed-size pages Map virtual pages to physical pages with a page table

TLB Contents

- **VPN:** used for lookup
- **PFN:** change the Virtual address VPN to PFN
- **G:** global bit (shared by all processes, don't check ASID)
- **ASID:** Address Space Identifier (which process's Page Table)
- **D:** dirty bit (changed when page has been written to)
- **V:** valid bit (valid translation present in entry)

Sapping Policy:

- **OPT:** Evict the page that will not be used for the longest time.
- **LRU:** Evict the page that has not been used for the longest time.

- **Random:** Randomly select a page to replace.

Clock Algorithm: Approximating LRU

- Add use bit to PTE, whenever page is referenced, bit set to 1
- Imagine all the pages of the system arranged in a circular list
- A clock hand points to some particular page, P
- When replacement needs to happen, OS checks use bit of page P
- if 1 , (not good candidate) set use bit to 0 and advance P , keep looking
- if 0 , (good candidate) replace this page

2-level Multi-level Paging Example:

```
      VA = 0x0214  (15 bits -> 5|5|5)
      +-------------+-----------+----------+
      |DirIdx=0x00  |PTIdx=0x10  |Off=0x14 |
      +-------------+-----------+----------+
             |
PDBR=13  -->  Physical Page #13 (Page Directory)
             |  Read byte 0 = PDE=0x83
             v  (V=1, Page Table PFN=0x03)
          Physical Page #3 (Page Table)
             |  Read byte 16 = PTE=0x8E
             v  (V=1, Data PFN=0x0E)
          Physical Page #14 (Data Page)
             |  Offset 0x14
             v
    Physical Address = (0x0E<<5) | 0x14 = 0x1D4
```

VAX/VMS Virtual Memory Layout:

- Page 0 invalid
- Segmentation: P0, P1, S
  - P0, P1: User segments
  - S: System segments(Kernel)
- Context Switch changes P0 & P1 PT Registers

Segmented FIFO

- RSS (Referenced Set Size): the maximum number of pages in memory for each process
- FIFO: first-in pages are moved to two **global** second-chance lists before actual eviction:
  - Clean-Page Free List
  - Dirty-Page List
- If another process needs free page, take first page off clean list
- If original process needs page before actual eviction, reclaims it from list.
- As the global list grow, it performs similar to LRU.
- Uses Clustering of pages from dirty list to write to disk.

Lazy Optmization: Demand Zeroing: To prevent process read sensitive data from previous process, the OS first mark PTE invalid. Only on page fault (the process trying to use the page), the OS will zero the page.

Copy-On-Write: Share physical page across different processes. If one is writing, then copy the page and write to the new page.

ASLR: Address Space Layout Randomization: Randomize the address space layout of the process to avoid buffer overflow attacks.

Concurrency

Threads and Locks

Threads: Multiple threads of a process share:

- Process ID (PID)
- Address Space: Code and Heap
- Open file descriptors
- Current working directory
- User and group IDs

Each thread has its own:

- Stack for local variables and return addresses
- Set of registers, including program counter and stack pointer
- Thread ID (TID)

Atomic Hardware Operation:

For x86 systems:

- `xchg(dst, src)`: exchanges the value of `dst` with `src`. Returns the original value of `dst`.
- `CompareAndSwap(dst, expected, new)`: if the value of `dst` is equal to `expected`, exchanges the value of `dst` with `new`. Returns the original value of `dst`.

For ARM systems:

- `LoadLinked(*ptr)`: loads `ptr` and monitors for changes.
- `StoreConditional(*ptr, value)`: if the value of `ptr` has not changed since the last `LoadLinked`, then stores the value of `value` to `ptr` and returns 1. Otherwise, returns 0.

Compare and Swap (CAS):

```
int CompareAndSwap(int *addr, int expected, int new) {
      int actual = *addr;
      if (actual == expected) {
            *addr = new;
      }
      return actual;
}
```

Lock Implementation:

With Atomic Exchange (Xchg):

```
typedef struct __lock_t {
    int flag;
} lock_t;
void init(lock_t *lock) {
    lock->flag = 0;
}
void lock(lock_t *lock) {
    while(Xchg(&lock->flag, 1) == 1) {
        // spin-wait (do nothing)
    }
}
void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

With Compare and Swap (CAS):

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) != 0) {
        //spin
    }
}
```

Condition Variables and Semaphores

Condition Variables: `wait(&c, &m)`: Wait for condition c to be signaled. It assumes the mutex m is locked when called. The call releases the lock and puts the thread to sleep.

After another thread calls `signal(&c)`, the thread will wake up and re-acquire the lock.

Meaning of Signal:

- Mesa Semantics: When a thread signals a condition, the waiting thread is only marked as ready to run.
- Hoare Semantics: When a thread signals a condition variable, it immediately transfers control to a waiting thread.

```
//Mesa Semantics
pthread_mutex_lock(&mutex);
while (!condition)
//The condition might still be false
//when the thread wakes up
    pthread_cond_wait(&cond, &mutex);
```

Semaphores: A semaphore is an object with an integer value that can be manipulated with two atomic routines:

```
int sem_wait(sem_t *s) {
      // decrement the value of semaphore s by one
      // wait if value of semaphore s is negative
}
int sem_post(sem_t *s) {
      // increment the value of semaphore s by one
      // if there are threads waiting, wake one
}
```

Zemaphore implementation:

```
typedef struct __Zem_t {
      int value;
      pthread_cond_t cond;
      pthread_mutex_t lock;
} Zem_t;

void Zem_init(Zem_t *s, int value) {
      s->value = value;
      Cond_init(&s->cond);
      Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}
```

```
void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

Binary Semaphore: A semaphore that can only be 0 or 1.

- `sem_wait`: If value = 1, set it to 0 and proceed. If value = 0, block until it becomes 1.
- `sem_post`: If threads are waiting, wake one of them (value stays 0). Otherwise, set value = 1.

Concurrency Problems

Conditions for Deadlock:

- Mutual exclusion: Threads claim exclusive control of resources that they require.
- Hold and wait: Threads hold resources while waiting for other resources.
- No preemption: Resources cannot be forcibly removed from threads
- Circular wait: Circular chain of threads hold resources that other threads are waiting for.

LiveLock: two threads repeatedly attempt to acquire locks held by each other and repeatedly fail.

```
top:
    pthread_mutex_lock(L1);
    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto top;
    }
```