1 Introduction to OS

Monolithic: One big special program. Unix variants and Windows NT/XP. (+) Well understood,good performance. (-) highly coupled components.
Microkernel: Small and clean. Only provides basic and essential facilities (IPC, scheduling, interrupt handler). Higher level services built on top and run as server process outside of the OS (use IPC to communicate). (+) Kernel extensible and better isolation & protection between kernel and higher level services. (-) lower performance.

2 System Calls

General System Call Mechanism

Library call places the syscall # in designated register, executes TRAP/SYSCALL to switch from user mode to kernel mode. Dispatcher determines syscall handler using syscall # as index. Syscall handler executed, return control to library call and switch to user mode when finished.

Exception and Interrupts

Exception: Synchronous (error in program execution).*Interrupt:* Asynchronous. Some portion of state saving done by hardware. Save state/registers in special interrupt stack. Usually special return from interrupt instruction. Non-maskable interrupt - always handled; Maskable interrupt - can be enabled or disabled (x86: CLI, STI). Vectored interrupt - array of interrupt handlers called by indexing interrupt # (x86: int n). General interrupt handler - some way to determine type (using 'cause' register)
int system(const char *command_line): creates a shell to run command line.

3 Process Abstraction

PCB/PTE: The entire execution context (hardware, memory, OS) for a process.
Process Table: Kernel maintains PCB for all processes in the process table.

4 Process Abstraction in Unix

void exit(int status): released on exit: memory, file descriptors. not yet releasable: PID, status, process accounting info, PTE. Child processes are inherited by init process. Parent sent SIGCHLD signal, status returned to parent using wait(). Cleanup: flush and close open streams from C stdio. Calls (several) exit handlers specified by program. Calls _exit(status) after standard C cleanup.

Parent-Child Synchronization

*int wait(int *status):* Returns PID of terminated child. Parent blocks until at least one child terminates. Cleans up remainder of child system resources. Kill zombie process. *waitid():* Wait for any child process to change status.
init process: Created in kernel at boot up time (PID=1). *Zombie process:* Parent terminates before child, init becomes 'pseudo' parent, init utilizes wait() to cleanup, or; Child terminates before parent but parent didn't call wait(), child process becomes zombie, can fill up process table, may need reboot.

Implementation Issues on fork()

Copy on Write: duplicate a page when it is written to; otherwise parent and child share the same page. CoW with paged VM: Share single page by having PTEs with same frame. Share read-only pages. Duplicate and make new page copy on write, PTE changed to new page and make page writable.

5 Process Scheduling

OS incurs overhead in context switch. Batch processing, Interactive, Real time processing.

Scheduling for Batch Processing

Criteria: turnaround time (finish-arrival time), throughput (number of tasks finished per unit time), CPU utilization
First-Come First-Served (FCFS): Blocked task removed from FIFO queue, only placed at back of queue when it's ready again. (+) No stavation.(-) Convoy effect (First task is CPU-Bound and followed by a number of IO-Bound tasks).
Shortest Job First (SJF): $Predicted_{n+1} = \alpha Actual_n + (1-\alpha)Predicted_n$ (+) Smallest average waiting time. (-) Starvation possible (for long job).
Shortest Remaining Time (SRT): New job with shorter remaining time can preempt current job. (Good service for short job even when arriving late)

Scheduling for Interactive Systems

Preemptive, scheduler runs periodically (from timer interrupt).
Criteria: response time, predictability (less variation in resp time).
Round Robin (RR): Preemptive version of FCFS (ready tasks placed at back of queue, blocked tasks moved to other queue). (+) Reponse time guarantee - n tasks quantum q, time before a task gets CPU bounded by $(n-1)q$. Big quantum: better CPU utilization, longer waiting time. Small quantum: Bigger overhead, shorter waiting time.
Priority Scheduling: Preemptive version: high priority proc can preempt running proc. Non-preemptive version: late coming high priority proc has to wait for next round of schduling. (-) Low priority proc can starve (worse in preemptive) -> decrease prioirty of current proc after each time quantum. (-) Hard to control CPU time. (-) Priority inversion: Lower priority task preempts higher priority task (high priority task requires resources locked by even lower priority task).
Multi-level Feedback Queue (MLFQ): Adaptive, reduce reponse time for IO-bound process, turnaround time for CPU bound process.
Basic rules:
Priority(A) > Priority(B) -> A runs
Priority(A) == Priority(B) -> A, B in RR
Priority setting/changing rules:
New job -> Highest priority
Use up timeslice -> Priority reduced
Give up/blocks -> Priority retained
Lottery Scheduling: Give out 'lottery tickets' to processes for various system resources. Choose ticket randomly and grant winner the resource. (+) Responsive. (+) Good lvl of control (parent gives tickets to child). (+) Simple implementation.

6 Process Scheduling in Unix

Traditional Unix Process Priority

nice: -20 to 19, default 0. priority=base+f(nice)+g(cpu usage estimate), g() is decay factor to reduce importance of long process.

Multi-level Feedback Queues

Dynamic adjustments with scheduling heuristics: age of proc, I/O boost (for interactive process), boost priority of foreground window (WinNT), system load effects.

Linux Scheduling

Multi-level feedback queue: level 0: Normal Time Sharing (SCHED_OTHER). level 1 to 99: Real-time FIFO(SCHED_FIFO) and Real-time Round Robin (SCHED_RR).
SCHED_FIFO: scheduling occurs only when higher priority RT arrives or voluntary yield. (timer interrupt used only for high priority to preempt).
SCHED _RR: preemptive. Preemption by higher priority RT tasks also.
SCHED_OTHER: nice priorities + dynamic feedback adjustments. Only run if no RT task ready.
Static priorities (RT priorities) not changed by scheduler, dynamic priority used for SCHED_OTHER tasks and changed by system. -> Static RT priority > Dynamic priority

Tickless

In new Linux kernels+WinNT. Only use timer interrupt when needed (not in sleep mode, adjust tick to next closest timer event). Tickless isn't no ticks but dynamic ticks.

7 Memory Abstraction

*int brk(void *end_data_segment):* how to deallocate?
*void *sbrk(ptrdiff_t brk_increment):* argument is signed integer, return previous break value.
*void *malloc(size_t size):* returns pointer to new memory region of size bytes.
*void free(void *ptr):* ptr must have been originally from malloc/calloc.
*void *calloc(size_t nitems, size_t size):* sets allocated memory to zero (malloc doesn't).

8 Contiguous Memory Allocation

Contiguous Memory Management

Fixed-size partition: (-) internal fragmentation. (-) external fragmentation (total unused memory > request size but no partition fits)
Dynamic partition: OS keeps track of occupied and free regions, performs splitting and merging. (+) no internal fragmentation (by splitting). (-) external fragmentation
Relocation & compaction: Move partitions around to remove external fragmentation. (-) data which are addresses may refer to wrong address after relocation.

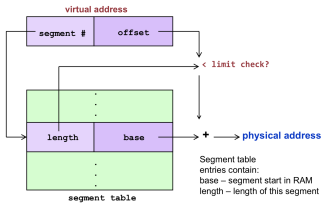
Dynamic Memory Allocation Algorithms

Sequential Fit: In free block: doubly linked list+size= 3 words for bookkeeping (minimum free block size)
First fit: (-) front list split more, many small blocks at start
Best fit: find smallest fitting block. (-) need searching whole list. (-) bad external fragmentation with many tiny blocks. (+) experimentally good memory use results.
Next fit: search from previously searched position. (+) avoids accumulation of small blocks at start. (-) poor locality affects caches.
Freeing a block: boundary tag optimization can coalesce in constant time.
Buddy System:
avail[i] records a free list of blocks of size 2^i (NULL means no free blocks). Splitting $xx..xx00..00$ ->$xx..xx00..00$ and $xx..xx10..00$. Find buddies for block of size 2^i -> flip bit at position $i+1$ from right to find buddy address. May have smallest allocated block size. (avail[0] for illustration)
Buddy allocation n: find smallest i for $2^i \geqslant n$. If avail[i] not empty, select free block there; else, find free block b from smallest avail[k] where $k > i$, split b until avail[i] has block.
Buddy deallocation p: find buddy p'. If p' not free, add p to avail list; else, coalesce p and p' and free it recursively.

9 Logical Addressing Schemes

VM cost: Badly behaved program causes thrashing.

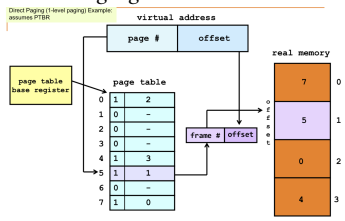
Segmentation for Address Translation



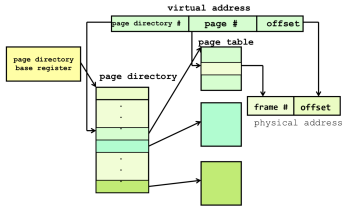
permission info (read, write, exec -> invalid operation leads to seg fault).
(-)external fragmentation (but easier compaction), resizing causes framentation problems, may require explicitly selecting segments, swapping overhead high for large segments.

Paging for Address Translation

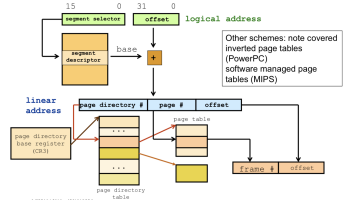
Internal fragmentation, no external fragmentation. Rather than resizing, just use more pages. Overhead of page transfer to/from swap is on the order of few disk blocks.
Memory Sharing: VM not needed for sharing, but allows logical address to be different and still share.
Direct Paging:



Overhead: page table size too big (same regardless of process memory usage)
2-level Paging: save space



Multi-level Paging: tree with k levels for k-level paging, space savings when tree not complete. Pages tables can be in VM and be paged.
x86 hybrid paging: segmentation+paging:



Page Table Entries (multi-level paging)

Non-leaf nodes: address of next level page table, valid bit. *Leaf nodes:* frame, valid(present) bit - page present in RAM, dirty(modified) bit - set when page is written, referenced bit - set when page is used, protection bits - read/write/exec.

Translation Lookaside Buffer (TLB)

$memory\_access\_time = m + (1-p) * 1 + pkm) \approx m$ (k-level paging). Without cache: $(k+1)m$ (p=1). Context switch -> TLB flush, just invalidate entire TLB cache. x86: write to register CR3(PDBR) flushes TLB.

10 Virtual Memory Management

Page Fault

HW virtual address translation: transparent if succeeds (frame in RAM, no protection violation) (common case), page fault exception if fails.

Page fault cases: page not present (valid bit off, OS handles, recursion if dealing with fault in page tables), protection error (OS generates software exception - SIGSEGV (seg fault) or SIGBUS), page not used (valid bit off, could be an auto expanding segment like stack).

Reset page table may need update multi-level tables.

Page Placement Policy: any frame

Page Fetch Policy

Demand paging: only faulting page is fetched. *Prepaging:* other pages can be fetched on page fault. Can also be loaded when starting process.

Page Replacement Algorithms

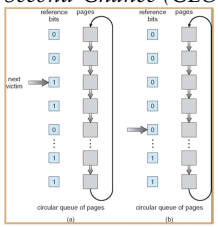
Incoming page can be from swap or executable file. If page to be replaced is dirty, need to write back to swap; if clean, can be discarded. Good algorithm have very few page faults: $T_{access} = (1-p)T_{mem} + pT_{page\_fault}$.

Optimal Page Replacement (OPT): Replace page that will not be used again for the longest time, guarantees min page faults.

FIFO: Evict oldest memory page. (+) simple implementation (OS maintains queue of resident page no.) (-) Belady's Anomaly (more frames, more page faults, since doesn't exploit temporal locality).

Least Recently used (LRU): Replace page that has not been used in longest time (temporal locality). (-) Hard to implement. Appr. 1: counter, PTE with time-of-use field(forever increasing may overflow), need to search through all pages. Appr. 2: stack of page no. If X referenced, remove from stack and push on top. Replace page at stack bottom.

Second-Chance (CLOCK):



- Use **circular queue** to maintain the pages:
  - With a pointer pointing to the oldest page (the **victim page**)
- To find a page to be replaced:
  - Advance (clock hand) until a page with '0' reference bit
  - Clear the reference bit as pointer passes through

On page fault: R (REF), D (Dirty) bits

1. starting at clock hand, scan at most 1 revolution to **find first frame with <R=0,D=0>**. If found, this frame is **selected** for replacement, advance clock hand 1 frame
   - prefer clean pages
2. If step 1 fails, scan at most 1 revolution to **find first frame with <R=0,D=1>** and **reset R=0 for any frames scanned with R=1**
   - deal with dirty pages – flush (write) to swap
3. If step 2 fails, goto step 1

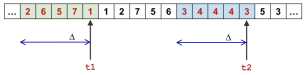
Frame Allocation

Equal allocation(N/M), Proportional allocation($size_p/size_{total} * N$).

Local replacement: (+) Stable performance between multiple runs, (-) not enough frame hinder progress. Global replacement: (+) self-adjustment between processes, (-) bad process affects others.*Thrashing:* not enough frame, global -> cascading thrashing, local -> single process hogs I/O.

Working Set Model:

- Example memory reference strings



- Assume
  - Δ = an interval of 5 memory references
- `W(t1,Δ)={1,2,5,6,7}` (5 frames needed)
- `W(t2,Δ)={3,4}` (2 frames needed)

Δ: too small -> miss page in current locality, too big : contains page from different locality.

11 File System Introduction

File

MSDOS file names: <name>.<extension> : name 9 char, extension 3 char. Letters converted to uppercase. Reserved con, prn, nul...

character set: letters+digits+

- special chars: `$ % ' - _ @ ~ ` ! ( ) { } ^ # &`

Type: regular(ASCII, binary), directories, speical files(devices, symbolic links, named pipes). Distinguishing: extension(Win), magic number(Unix).

Protection: Access control list, Permission bits(owner, group, other).

Structure: array of bytes, fixed length record(instant access), variable length record(hard to locate).

Access method: sequential access(rewound), random access(read, seek), direct access(on fixed length record, general random on record).

Directory

Single-level, tree-structured (every node has only one parent), DAG (aliases, appear in multiple directories), general graph (achieved with symbolic link, infinite path possible -> maximum traversal limit in Unix).

Unix symbolic link: Special link file contains path name. ln -s. Recursive. Win: shortcut does not re-expand further(not recursive).

12 File System Unix

unified I/O: many types of files but same syscalls: data, pipe, devices, proc...

int open(char *path, int flags [, mode]): returns -1 if fail, >=0 for file descriptor. mode is new file permissions. Open checks permissions. Default fd: STDIN(0), STDOUT(1), STDERR(2)

fd sharing: After fork, fd value is copied, internal kernel file object and file offset are shared.

int read(int fd, void *buf, int n): return no. of bytes read, <n means reach EOF. sequential read.

int write(int fd, void *buf, int n): return -1 if error(exceeds file size limit, quota, disk space), >=0 for no. of bytes written. sequential write.

off_t lseek(int fd, off_t offset, int whence): Random I/O. Move file offset, return -1 if error, >=0 for absolute offset. Offset can be +ve or -ve, whence can be SEEK_SET, SEEK_CUR, SEEK_END. Can create holes with implicit zeroes.

int close(int fd): fd can be reused later. Process termination automatically closes all open files.

13 File System Implementation

File Block Allocation

Contiguous: (+) Simple to keep track(start #+length), fast access. (-) external fragmenation, specify file size in advance.

Linked List: Store file start end. (+) No fragmentation. (-) slow random access(disk speed pointer), part of block used for pointer.

Linked List V2.0: Move all block pointers to File Allocation Table(FAT)(in memory). (+) Faster random access(traversal in memory). (-) FAT size depends on disk size, consumes memory.

Indexed allocation: Linked scheme(linked list of index blocks), multilevel index(1st lvl index block points to several 2nd level index blocks), combined scheme.

Free Space Management

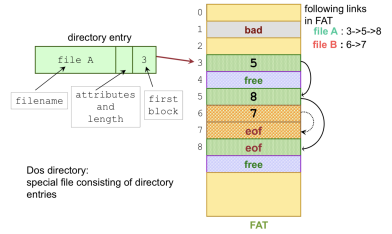
Linked list(linked indexed representation), array(compress to bitmap).

```
0 1 2 3 4 5 6 7 8 9 10  (block #)
11100110001 : bitmap
```

Used blocks: 0-2, 5-6,10; Free blocks: 3-4, 7-9

MSDOS

FAT stored in disk, duplicated in RAM. FAT16 has root dir size limit of 512 entries.



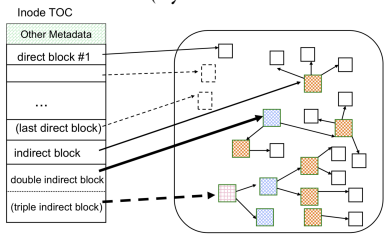
Deleting file: Set first letter in filename to 0xE5, set FAT entries in linked list to FREE.

Cluster: 16 bits numbers in FAT entries, logical block size=cluster size=multiple of sectors. Large cluster size -> large internal fragmenation.

Disk Fragmenation: logical contiguous blocks far apart on disk(about time). FAT: less disk fragmentation with large cluster size, solution: run defragmentation(compaction) on entire FS. s5fs: worse since smaller logical block size.

Unix s5fs

Inode: actual file object, one per file. Contains: reference count for hard links, Table of Contents(TOC) mapping file data to disk blocks(hybrid multi-lvl index).



Allow blocks which do not exist(logical zero filled holes) - set block pointer to NULL in TOC, return zeroes when read.

Directories: Map inode # to filename. Array of 16-byte entries(inode 16 bits, filename 14 bytes). Deleted file has inode 0.

Hard links: same file to have more than one filename(create DAG). ln. Same inode # for different filenames. Remove reference with unlink() syscall. Deleting: remove dir entry, decre inode link count, free file object when link count=0.

Caching: add a buffer cache in memory for disk blocks. Write can be faster than reads(can write to buffer). Use LRU to replace blocks in cache. sync(2) syscall requests dirty buffers be flushed, fsync(2) waits for completion.

14 Synchronization

Critical Section (CS)

Properties: Mutual exclusion, progress(if no process in CS, waiting process evetually granted access), bounded wait.

Assumptions: independence on non-CS, takes finite time.

Problems: Deadlock, starvation, livelock(keeping changing state with no progress).

Peterson's Algorithm

```
want[0]=0; want[1]=0; turn = 0 or 1;   // initial
P0:                                    P1:
want[0]=1;                             want[1]=1;
turn=1;                                turn=0;      // indicate turn
while (want[1] &&                      while (want[0] &&
       (turn == 1)) ;                         (turn == 0)) ;
CS                                     CS
want[0]=0;                             want[1]=0;
```

(-) busy waiting

Semaphore

Wait() and Signal():

Wait(S):
- If **S** <= 0, blocks (go to sleep)
- *// S > 0*
- **S--**
- Also known as `P()` or `Down()`

Signal(S):
- **S++**
- Wakes up one sleeping process if any
- This operation **never** blocks
- Also known as `V()` or `Up()`

$S_{curr} = S_{init} + \#signal(S) - \#wait(S\_comp)$

Mutex: S=1; Wait(S); CS; Signal(S).

Classic Synchronization Problems

Producer Consumer (Blocking):

```
while (TRUE) {                          while (TRUE) {
    Produce Item;                           wait( notEmpty );
    wait( notFull );                        wait( mutex );
    wait( mutex );                          item = buffer[out];
    in = (in+1) % K;                        out = (out+1) % K;
    count++;                                count--;
    signal( mutex );                        signal( mutex );
    signal( notEmpty );                     signal( notFull );
}                                           Consume Item;
                                        }
```

Producer Process | Consumer Process

- Initial Values:
  - `count = in = out = 0`
  - `mutex = S(1), notFull = S(K), notEmpty = S(0)`

Reader Writer:

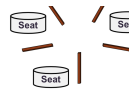
```
while (TRUE) {                          while (TRUE) {
    wait( roomEmpty );                      wait( mutex );
    Modifies data                           nReader++;
    signal( roomEmpty );                    if (nReader == 1)
}                                               wait( roomEmpty );
                                            signal( mutex );
                                            Reads data
                                            wait( mutex );
                                            nReader--;
                                            if (nReader == 0)
                                                signal( roomEmpty );
                                            signal( mutex );
                                        }
```

Writer Process | Reader Process

- Initial Values:
  - `roomEmpty = S(1)`
  - `mutex = S(1)`
  - `nReader = 0`

Dinning Philosopher:

```
void philosopher( int i ){
    while (TRUE){
        Think( );
        wait( seats );
        wait( chpStk[LEFT] );
        wait( chpStk[RIGHT] );
        Eat( );
        signal( chpStk[LEFT] );
        signal( chpStk[RIGHT] );
        signal( seats );
    }
}
```



- Initial Values:
  - `seats = S(4)`
  - `chpStk = S(1)[5]`

OS Implementation

User mode semaphore implemted by OS:

```
wait(S):
if (S.value == 0) {
  p = get PID;
  add p to S.queue;
  set p to BLOCKED;

  reschedule;
}
S.value--;
```

```
signal(S):
S.value++;
if (S.value == 1 &&
    notempty(S.queue)) {
    p = remove one process
        from S.queue;
    set p to READY;
}
```

Kernel: disable interrupt(miss interrupt, interfere scheduling)

Multi-processor: Busy-waiting mutual exclusion(spinlock), HW with atomic read+write

Hardware atomic instruction: TestAndSet

```
mutex = 0;
// Process P1                          // Process P2
while TestNSet(&mutex, 1);             while TestNSet(&mutex, 1);
  Critical Section                       Critical Section
mutex = 0; // unlock                   mutex = 0;
```

(spinlock)