**yield**-voluntary context switch
**Throughput**: maximum # job complete/h
**Turnaround**= Completion Time − Arrival Time
**Round robin**: change process -> restart time
time -> new -> just pre-empted
**Race condition**: outcome depends on the order in which accesses take place
**Spinlock**:

```
Typedef struct _lock_t { int flag; } lock_t;
Void init(lock_t *mutex) {
        Mutex->flag=0; // available}
Void lock(lock_t *mutex) {
 While (test_and_set(&mutex->flag, 1) == 1);}
Void unlock(lock_t *mutex) {
        Mutex -> flag = 0;  }
```

**Using conditional variables**:

```
pthread_mutex_lock(mutex); // struct lock *
While (condition not satisfied) {
      pthread_cond_wait(cond, mutex);
// add thread to cond's wait queue and sleep
// Re-acquires mutex before return  }
.. // do stuff
pthread_cond_signal(cond);
pthread_cond_broadcast(cond);
pthread_mutex_unlock(mutex)
```

**Semaphores**:

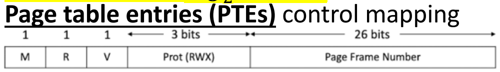
```
Sem mutex = sem_init(1)
Sem empty = sem_init(0)
Sem full = sem_init(N)
```

| | |
|---|---|
| Producer {<br>// block if bf is full<br>Sem_wait(full);<br>Sem_wait(mutex);<br>Add_to_buffer();<br>Sem_signal(mutex);<br>// buffer is no longer empty<br>Sem_signal(empty);} | Consumer {<br>// block if buffer is empty<br>Sem_wait(empty)<br>Sem_wait(mutex)<br>Rm_from_buffer();<br>Sem_signal(mutex)<br>// bf is no longer full<br>Sem_signal(full) } |

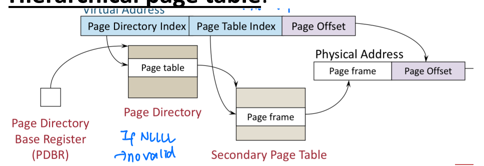
**Lost wake up**: sem doesn't suffer
**Concurrency Bugs**
**Atomicity violation** – lack of mutex inside critical section, violate serializability; **Order violation bugs** – incorrect ordering of ops
**Deadlock bugs** – circular waiting
**Resource Deadlocks**
1.**mutual exclusion** – only one process may use a resource at a time -> use architectural support to create lock-free data structure

```
Void AtomicAdd(int *val, int a){
    do { int old = *val; }
    while (CAS(val, old, old+a) == 0 ); }
```

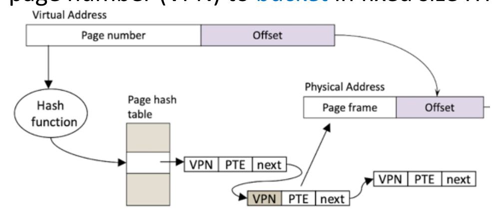
2. **Hold and Wait** – a process may allocated resources while awaiting assignment of others -> Get all needed resources at the same time. Alternative use trylock()
Ready = false; Lock(L1); if (trylock(L2) == -1) {unlock(L1);} else {ready = true;}
3. **No pre-emption** – no resource can be forcibly removed from a process
4. **Circular wait** – a closed chain of processes exits, such that each process holds at least one resource needed by the next process in the chain -> assign a linear ordering to resource types and require that a process holding a resource of one type, R, can only request resources that follow R in the ordering or **Lock ordering**
**Ostrich Algorithm** ignore the problem and hope it doesn't happen often
**Scheduling Priority inversion**: happens when a lower pri thread prevents a high priority thread from running.
**inheritance**: A protocol that temporarily boosts the priority of a lower-priority task (holding a resource) to match the highest-priority task waiting for that resource.
**Unix CPU scheduling**: long CPU time -> lower pri; rescheduling occurs every 0.1s; priority is recomputed at the end of every time slice 1s.
Pj(i) = basej + [CPUj(i-1)]/2 + nicej
CPUj(i) = Uj(i)/2 + CPUj(i – 1)/2
**Memory management**: **Address binding**:
1. **Compile time** (-b) 2. **Load time (static relocation)**: compiler – relocatable logical address in object file -> linker – logical absolute address & relocation table -> loader – physical addr when the prog is loaded into memory; 3.**execution time (use this)**
**Fixed partitioning of physical memory -> internal fragmentation & overlay**
# of partitions determines # of active process
Decide at system configuration (boot) time
**Dynamic partitioning->ext. fragmentation**
->compaction-require process be relocatable
Brk() – sys call – to extend in malloc/free
**Dynamic location**: binding addresses at execution time. Executable object file contains logical addresses for entire program, translated to physical address during execution.
**Paging**: no need to single contiguous physical memory partition; no need both base and bound (limit) registers; still need translate virtual page to physical frame
**PTBR** (base register): CPU need one register to find the start of the page table in memory
Hardware **MMU** converts va into pa using the **page table** in memory -RAM for each process
$1KB = 2^{10}B = 1024\,B;\ 2^{10}KB = 1MB$
# of offset bits = $\log_2^{(page\ size)}$
**Page table entries (PTEs)** control mapping



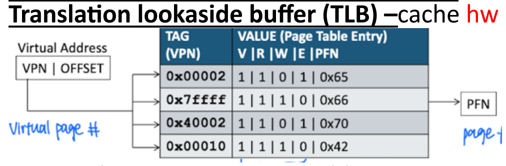

• **Modified bit** (M) says if the page has been written (also called the Dirty bit, D) - Set when a write to a page occurs
• **Referenced bit** (R) says if page has been accessed - Set when a read or write to the page occurs • **Valid bit** (V) says if this PTE can be used - Checked on each use of virtual address • **Protection bits** (Prot) specify what operations are allowed on page - Read/Write/Execute
**Hierarchical page table**:




how many -log-> how many bit
**Hashed page tables**: hash func maps virtual page number (VPN) to bucket in fixed size HT




**Inverted page tables**: 1 table with 1 entry for each physical page frame; entry record which virtual page # is stored in that frame (process id); less space but lookup slower;
**core map**: track state of frame, free or used
**TLB and demand paging**
**Place translation into the TLB (loads TLB)**
**Translation lookaside buffer (TLB)** –cache hw




•**MMU**: know PTBR; access tables in mem directly;Tables must be in HW-defined format
• TLB faults to **OS**, OS finds PTE, loads it in tlb
**Flush** – invalidate all entries
**When a process access a page that evicted?**
1. When it evicts a page, the OS sets the PTE as invalid and stores the location of the page in the swap file in the PTE 2. When a process accesses the page, the invalid PTE will cause a trap (**page fault**) 3. The trap will run the OS page fault handler 4. Handler uses the invalid PTE to locate page in swap file 5. Reads page into a physical frame, updates PTE to point to it 6. Resume process -> 贵, require 2 disk access -> keep free pages
**Demand paging**: pages are evicted to disk when memory is full. Pages loaded from disk when referenced again. References to evicted pages cause a **TLB miss(fault)**. OS allocates a page frame, reads page from disk. When I/O completes, the OS fills in PTE, marks it valid, resume process, retry faulting instruction. Also when a process first starts up.
Cost: Timing-disk read is initiated when the process needs the page; request size: process can only page fault on 1 page at a time, disk sees single page sized read.
Efficient when req size is large & contiguous
**Dirty pages**: be modified need written to disk
**Locality**: all paging schemes depend on ~
**Temporal locality**: Locations referenced recently likely to be referenced again.
**Spatial locality**: loc near recently referenced locations are likely to be referenced soon
**Paging policies - replacement**
**Prepaging (prefetching)**: Predict future page use at time of current fault
**Page replacement policy**: Determines how a victim is chosen for replacement
**Belady's Algorithm**: OPT – lowest fault rate
Replace the page that will not be used for the longest period of time
(cold miss: first access to a page) 即使把其他踢走
**First in first out (FIFO)**: • Ignores locality of process's memory references • suffer from **Belady Anomaly** – the fault rate might actually increase when the algo is given more mem **Cond**: L contain >=1 page that's also in S & S contain >= 1 page that's not in L
**Least Recently Used**: •evict page that has not been used for the longest time in the past • do well-program with high temporal locality •no belady's anomaly •Adds software overhead to every memory references
•Vulnerable to scanning reference patterns
**Approximating LRU**: LRU approximations use the PTE reference bit • All R bits are initially zero. • As processes execute, bits are set to 1 for pages that are used •Periodically examine the R bits •we do not know order of use, but we know pages that were (/not) used
**Additional-Reference-Bits**: Keep a counter for each page • At regular intervals, for every page do: • Shift counter bits to the right by 1 (e.g., 0101→0010) • Shift R bit into high bit of counter (e.g., 0010 → 1010) • Pages with "larger" counters were used more recently
**Second Chance Algorithm**: FIFO, but inspect reference bit •If ref bit is 0, replace the page •If ref bit is 1, clear ref bit, try the next page •Pages that are used often enough to keep reference bits set will not be replaced
**Clock replacement**: hand 从 0 开始,load & set(1) 并移到下一个, (0):放, set(1) & hand 移到下一个;hit 不变(1), 不移;(1)不可动:set(0) & hand 移到下一个;Hand 刚好是 hit, set(1) & 不移
**Thrashing**: When most time is spent by the OS in paging data back and forth from disk than executing user programs

**Simplified 2Q Algorithm** A1 – FIFO; Am - LRU

- On reference to page *p* (frame of p)
  - Equivalently, the frame allocated for page p:
- if *p* is on the Am queue then:
  - move p to MRU position of Am
- else if *p* is on the A1 queue then:
  - remove *p* from A1 most recent use
  - put *p* on Am queue in MRU position
- else // first access we know about for p
  - put *p* on A1 queue in youngest position

- To allocate a frame for page *p*
  - i.e., when all frames are in use
- if A1's size is above its threshold then:
  - evict oldest page from A1 (first-in)
  - put *p* in the freed frame
- else:
  - delete LRU page from Am
  - put *p* in the freed frame

**Advanced VM Functionalities**

**Kernel Virtual Memory**: store page tables, page the entire OS address space

**Managing Swap Space**:
Op1 use raw disk partition swap (faster, request disk reformats to resize)
Op2 use ordinary large file in file system (flexible, swap file can become fragmented)

**Virtual Memory Area**: Track Allocated Addresses; E.g. <start, size> of code area, <start, size> of initialized data area, <start, size> of stack area •Initial VMAs are created as part of handling exec() system call. •Addi VMAs are created in response to mmap() system calls. •Linux doesn't allocate memory on malloc()/mmap()/sbrk() call •Mem isn't allocated until the first time it is used •The page fault handler checks the VMAs first to see if the faulting address is valid.

**Way to allocate mem to process**: Local/variable replacement algorithm

**Working Set Model:** WS(t,D) = {pages P such that P was referenced in the time interval [t, t-D)}

**Sharing**: •Have PTEs in both tables map to the same physical frame • Can map shared memory at same or different virtual addresses in each process' address space • Different: Flexible, but pointers inside the shared memory segment are invalid

**Copy on Write**: Use CoW to defer large copies for as long as possible, hoping to avoid them altogether • Instead of copying pages, create shared mappings of parent pages in child virtual address space • Shared pages are protected as read-only in parent and child • Reads happen as usual • Writes generate a protection fault, trap to OS, copy page, change page mapping in diff page table, enable write permission, restart write instruction mmap(addr, length, prot, flags, fd, offset) Map length bytes from file fd starting at offset to addr

**File systems**

**Directory**: • a list of entries – names and associated metadata (information that describes properties of the data (size, protection, location, etc.)) •unordered • stored in files

**Acyclic graph directories** allows for shared directories **Hard links**: Second directory entry identical to the first(link to same file) ln /usr/local/d/afile /home/bfile • can't create new hard links to directories • can't create link across file sys boundary • no extra processing • rm a hard link only rm the file if it is the last link to file

**Symbolic link (soft link):** Directory entry refers to file that holds "true" path to the linked file ln –s /usr/local/d/afile /home/bfile (bfile contains the path to the linked file) • can create soft links to directories • can point anywhere • need to look up the link to follow • rm a file may lead to **dangling link**

OS views a disk as an array of fixed size block

**Indexed structure – Unix inodes – 128byte** The metadata for files and directories are held by an inode. Each inode contains 15 block pointers • First 12 are direct block pointers (Disk addresses) • 13th is a single indirect block pointer • 14th is a double ~ • 15th is triple indirect block pointer

**Very Simple File System (VSFS)**

**Disks** are block access devices

**Sector size**: Min disk block size (512 or 4096b

**1.Superblock**: • Identifies type of file system (**magic number**) … **2.IB**: 1 bitmap for the inode region **3.DB**: 1 bitmap for the data region **4.Inode Table**: • must reserve blocks for inode table when file system is initialized mkfs.vsfs –i 160 disk.img •pre-defined maximum number of files in file system **5.Data region**

**Multilevel block pointers form a tree.** ->(imbalanced)

**Extent-based allocation:** An extent-a disk pointer plus a length > use less metadata per file, and file allocation is more compact > less flexible

**Linked based**: poor when access last block > use in mem **File Allocation Table** > tracks the next pointer of allocated blocks, faster

**Implement directory entries:**

```
Struct vsfs_dentry {  // fixed len
    unsigned int inode; // inode number
    char name[252]; // file name};
struct ext2_dir_entry { // variable len
    unsigned int inode; // inode number
    unsigned short rec_len; // entry len,
// 4x, header size (8?)+ size of (name)
    unsigned char name_len;
    unsigned char file_type;
    char name[];};
```

last size has to be the remaining size of block

**Access file "/one/two/three"** 1. Read super block for the location of inode for "/" 2. Read inode block containing the inode for "/" 3. Read directory block for "/" • search for entry "one" to get inode number 4. Read inode block containing inode for "one" 5. Read directory block for "one" • search for entry "two" to get inode number 6. Read inode block containing inode for "two" 7. Read directory block for "two" • search for entry "three" to get inode number 8. Read inode block for "three" • Now we can read the first block of "three".

**Disk operation:** Inode [d/f a:0 r:3] (inode# 按顺序, 从 0 开始) a:address of first data block (block # addr) r: # of reference data block [u] (block# 按顺序, 从 0 开始) directory block [(name,inode#) (..)]

**Cons**: poor utilization of disk bandwidth; aging files systems have data blocks scattered across disk. Fragmentation causes seeking. Also going back from inodes to data blocks causes seeks in path traversal, manipulating files/directories

**File system implementation**

**Open**: system & process **open file table** (ram)

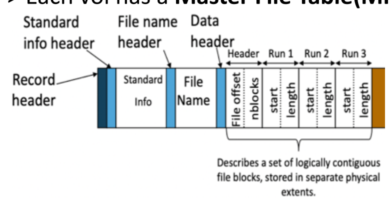
**Closeness:** • Reduce seek times by putting related things close to one another

**Amorization**: • Amortize each positioning delay by grabbing lots of useful data

**Berkeley Fast File System FFS** • Disk partitioned into groups of cylinders • allocated in same cylinder group: Data blocks in same file, Files in same directory • Inodes for files allocated in same cylinder group as file data blocks

**Cons**: Low bandwidth utilization, small max file size (function of block size) (fix: large size & **tail packing**), media failure (replicate super block), Device oblivious(Parameterize FS according to device characteristics)

**New Technology File System NTFS**
>Each volume (partition) is a linear sequence of blocks • Usually 4 KB block size
> Each vol has a **Master File Table(MFT) 1KB**.




header, std info header, standard infor, file name h, file name, data header, file data

**Small file**: kept data in MFT record itself
**Large file**: extent-based 20|4: [20,23]
**Directory**: simple list, large dir use B+ trees
**Pros**: large contiguous chunks of data

**Virtual File Sys VFS**: abstract file sys interface

**File System Reliability** Ensure that the file system metadata is in a consistent state following an operating system failure.

**Create a file**: dir[4] add new dir entry; I[2] for new file; I[1]last modified time; inode bitmap
**Append to file**: I[2](last modify time, block pointer, size), data bitmap, D[3]
**Only inode** -> point to garbage
**& No bitmap** -> Multiple inodes may point to the same data block
Only bitmap -> Data leak (data block is lost for any future use)

**fsck – file system check -> storage /fs bugs** 1.superblock 2.free blocks (bitmap) 3.inode state(mode) 4.inode link (count) 5.duplicate (no 2 inode refer to same block) 6.bad block (pointer outside of valid range) 7.directory (.,.. are first 2, inode is allocated, no dir is linked more than once -> could create loop)

**Limitation**: poor at detect/fix data block corruption, too slow!

**File System Journaling -> deal crash failure** Checkpoint: write changes to file system TxBegin-IBLOCK-DBMAP-DBLOCK-barrier A – TxEnd- barrier B
**Pros**: faster than fsck **cons:** double time (journal then write), may break seq writing

**Metadata Journaling**: Write data, wait until it completes -> write FS metadata (no actual data) -> Metadata journal commit -> Checkpoint metadata -> Free transaction
**Pros**: normal operation performance, recovery performance, Space to store Journel

**Solid State Disks** -Based on **flash memory** • Stores information via electrical charges in memory cells • Page-unit of r/w; block-unit of erase operation • Uniform random access performance **Issue:** write unit!=erase unit, limited endurance **Flash translation layer** -> log based mapping (don't erase until all get written) • Reduce **write amplification** (i.e., the amount of extra copying needed to deal with block-level erases). • A block must be erased before its pages can be written to • **wear leaving** (wear out ->balance # of program/erase cycles that are sent to each block of flash)- always write to new physical loc; keep a map from logical FS block # to current SSD block and page loc; old version of logically overwritten pages are stale • **garbage collection** (no update-in-place): reclaim stale pages and create empty erased blocks

**Defense:** stack canary(A special value between the buffer and the return address. If the canary is changed upon return, crash the program); stack non-exe; address space layout randomization

**page based**: map: logical #-> physical page # write(logical #, content) • Erase on entire block -> all E • can't write on T, I, V • can only write on E • if write on V, change this one to T • new map, don't change old map (V) • garbage collector runs to free the block -> all E & valid be moved other place **block based**: map: chunk # -> block # • write(logical #, content) • logical pg # -> chunk # Offset # -map->block # Offset # • if write on V, change this one to T, others not changed. Move the entire block to a new place (when all invalid -> unma