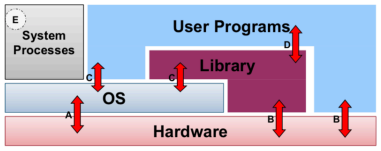
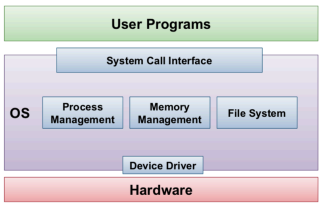
Multiprogramming: loads multiple jobs and runs other jobs when I/O needs to be done
Time-Sharing OS: Allow **multiple users** to interact with machine using terminals
Stack Memory Region: dynamically used by function invocations, a memory region to store info from function invocation
The top of stack region (first unused location) is logically indicated by a **Stack Pointer** (SP)

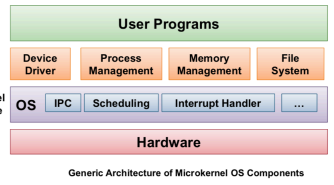


- **A**: OS executing machine instructions
- **B**: normal machine instructions executed (program/library code)
- **C**: calling OS using ***system call interface***
- **D**: user program calls library code
- **E**: system processes
  - Provide high level services, usually part of OS

Monolithic Kernel Illustration



Microkernel Components



Generic Architecture of Microkernel OS Components

Generic Architecture of Monolithic OS Components

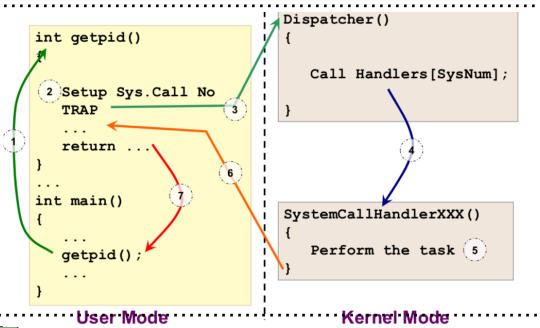
When GPRs are exhausted:
- ❑ Use memory to temporary hold the GPR value
- ❑ That GPR can then be reused for other purpose
- ❑ The GPR value can be restored afterwards
- ❑ known as **register spilling**

on executing function call:
caller: pass arguments with registers and/or stack
caller: save return PC on stack
transfer control from caller to callee
callee: save registers used by callee. Save old FP, SP
callee: allocate space for local variables of callee on stack
callee: adjust SP to point to new stack top
on returning from function call
callee: restore saved registers, FP, SP
transfer control from callee to caller using said PC
caller: continues execution in caller

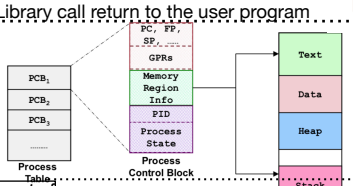
Stack frame contains:
- ❑Return address of the caller
- ❑Arguments for the function
- ❑Storage for local variables
- ❑Other information.... (more later)

exception: machine level instruction: arithmetic errors, memory accessing errors
Have to execute a **exception handler,** Similar to a forced function call
interruption: external events: hardware related
Program execution is suspended, Have to execute an **interrupt handler** (**Process / Task / Job**) is a *dynamic abstraction* for executing program

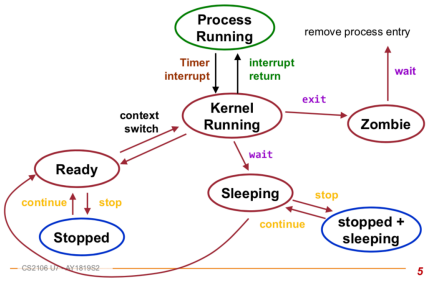
1.User program invokes the library call
2.Library call places the sys call number in a designated location, e.g. reg
3.Library call executes a special instruction to switch to kernel mode: TRAP or SYSCALL
4.The appropriate system call handler is determined, handled by a dispatcher
5.System call handler is executed: carry out the actual request
6.System call handler ended: control return to library call, switch to user mode
7.Library call return to the user program



User Mode | Kernel Mode



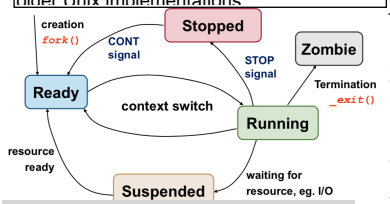
getpid - Library call that has the same name as a system call
printf, exit(0) - Library call that make a system call
getenv()/putenv()
System calls used:
- **fork(), wait() family**
- **getpid**: does actual trap to execute real getpid system call
- **write**: called by **printf**
- **close**: called by **exit** , **_exit**: called by **exit**
- **exec() family**❑ Code + data replacement – essentially changing executable ❑ PID and other state preserved, e.g. open files



How to Context Switch between Processes
Transition from running process A to another running/ready process B: (rough)
- make Process A no longer runnable:
- ❑ save CPU (+ OS) context of process A, e.g. save registers (PC+SP+other registers) to PCB of process A
- ❑ set process A state to READY
- Make Process B runnable:
- ❑ existing B process: restore registers (+ OS) context of process B from PCB of process B
- ❑ new B process: set context to defaults
- ❑ set process B state to RUNNING
- ❑ set PC to be in B (depends on how syscall works)
- memory context should be switched as well (logical view should be of process B's memory, affects caches – so cost is not just in registers but cache misses!)
- switch OS context

- ❑ Kernel maintains PCB for all processes
- ❑ Conceptually stored as one table representing all processes

process:
- ❑ init process becomes "pseudo" parent of child processes
- ❑ Child termination sends signal to init, which utilizes **wait()** to cleanup
- ❑ Child process become a zombie process
- ❑ Can fill up process table
- May need a reboot to clear the table on older Unix implementations



What does OS do for memory management:
- ❑ Allocate memory to new processes
- ❑ Manage memory of process (ideally in transparent fashion)
- ❑ Manage memory between processes + kernel
- ❑ Manage kernel memory for internal use
- ❑ Provide OS services to do with memory: get more memory, free memory, protect memory regions, etc.

Linux Real Time Scheduling
- Soft real time (RT)
- **SCHED_FIFO** policy: scheduling occurs only when:
- ❑ higher priority RT task
- ❑ system call wait, voluntary yield
- ❑ note: timer interrupt still used so higher priorities can preempt
- **SCHED_RR** policy: similar to FIFO but preemptive scheduling applies, round robin scheduling of RT tasks of same priority, preemption by higher RT priority tasks
- **SCHED_OTHER**:
- ❑ normal processes/threads – like traditional Unix
- ❑ unix nice priorities + dynamic feedback adjustments
- ❑ can only run if no real time processes in ready state
- Other policies:
- ❑ **SCHED_DEADLINE** , **SCHED_BATCH**, **SCHED_IDLE**

roughly: process_priority = **base** + f(**nice**) + g(**cpu usage estimate**)
f(): nice adjustment from nice value
g(): decay factor: reduce importance of long processes

Task States: (PS fields for states: **R/S/D/T/Z**, process/task are the equivalent for scheduler)
- **TASK_RUNNING**: running on CPU or waiting to run only these tasks can be scheduled
- TASK_INTERRUPTIBLE / TASK_UNINTERRUPTIBLE: sleeping for condition/ event/IO
- TASK_ZOMBIE: process terminated
- TASK_STOPPED: stopped by signal

Remember: only TASK_RUNNING tasks can be considered for running so scheduler looks at the TASK_RUNNING tasks

3 scheduling classes: (Posix) - soft real-time
- ❑ **Real-Time FIFO**: SCHED_FIFO
- ❑ **Real-Time Round Robin**: SCHED_RR
- ❑ **Normal Time Sharing**: SCHED_OTHER

form of multilevel feedback queues (can be viewed as 100 levels (0-99), level 0 is normal non RT, level 1 to 99 is RT
non-RT tasks **never run** if there are runnable RT tasks
Static Priorities: RT priorities are not changed by scheduler
Dynamic Priority: used for SCHED_OTHER tasks, changed by system
Static RT priority > Dynamic priority

On batch processing system: FCFS, SJF, SRT
- ❑ No user interaction
- ❑ Non-preemptive scheduling is predominant
- **Turnaround time**:
- ❑ Total time taken, i.e. finish-arrival time
- ❑ Related to **waiting time**: time spent waiting for CPU
- **Throughput**:
- ❑ Number of tasks finished per unit time
- **CPU utilization**:
- ❑ Percentage of time when CPU is working on a task

$Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$

Criteria for interactive environment: RR, Priority Based, MLFQ, Lottery
- **Response time**:
- ❑ Time between request and response by system
- **Predictability**:
- ❑ Variation in response time, lesser variation == more predictable

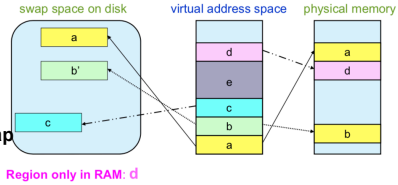
Preemptive scheduling algorithms are used to ensure good response time
➔ Scheduler needs to run **periodically**
➔ **must take over control of CPU**

Process Address Space Layout (conceptual view)
- code: executable machine code of program
- data: (global) data **statically allocated** by compiler and linker. Some data may be initialized from executable, eg. "error messages" in program. May have read-only data
- heap: dynamic memory usage is under program control, may **grow/shrink** dynamically at runtime (think of malloc() / free())
- stack: stores stack frames for procedures/ function calls (dynamic with call/ret)

Traditional Unix Process Layout
- **text segment**: code, may be **read-only**, modern Unix may have several text regions
- **initialised data segment**: global data loaded from executable file, may be in several regions, can be read-only
  `char * mesg[] = {"abc", "def"};`
  eg: initialised data contains (3 objects): 2 strings, array of pointers to start of both strings
- **bss segment**: global data which doesn't need initialization, is **zeroed**
- **heap**
- **stack**
- **arguments to** `main(): argv[], envp[]`
  - might be considered part of stack

- temporal locality: memory address which is used is likely to be **used again**
- spatial locality: memory addresses **close to a used address** is likely to be **used**

Locality of reference is important assumption which makes VM workable since it means that RAM can act as cache for VM, not all of VM needs to be used given good locality



Enlarge memory by using storing some VM in **swap space** on disk | disk| ≥ |memory| and disk access time >> memory access

Region only in RAM: d
In RAM & swap: a & b (**b'** out of date as b has been modified)
Region only in swap: c
Regions not in RAM: e
- **Virtually contiguous** ≠ **physical contiguous**: a+b virtually contiguous

void _exit(int status)
- _exit system call used for immediate voluntary termination of process (never returns!)
- Closes all open file descriptors; children processes are inherited by init process;
- parent sent SIGCHLD signal
- status returned to parent using wait()
- Usually status is used to indicate errors

Some basic process resources not yet releasable:
- PID & status needed: For parent-children synchronization
- Process accounting info, e.g. cpu time ➔ Process table entry **may be** still needed

wait sys call
- ❑ The call is blocking:
- Parent process blocks until at least one child terminates
- ❑ The call cleans up ***remainder*** of child system resources
- Those not removed on **exit()**
- Kill zombie process

Cannot delete all process info
- What if parent ask for the info in a wait() call?
- Remainder of process data structure can be **cleaned up**
- ❑ only when wait() happens

Zombie Process (2 Cases)
1. Parent process terminates before child
2. Child process terminates before parent but parent did not call wait:

Simplified implementation:
1. Create address space of child process
2. Allocate **p'** = new **PID**
3. Create kernel process data structures ■ E.g.Entry in ProcessTable
4. Copy kernel env of parent process ■ E.g.Priority(for process scheduling)
5. Initialize child process context: ■ **PID=p',PPID=parent** id, zero CPU time
6. Copy memory regions from parent ■ Program, Data, Stack (more later)
7. Acquires shared resources: ■ Open files, current working directory etc
8. Initialize hardware context for child process: ■ Copy registers, etc.
9. Child process is now ready to run ■ add to scheduler queue

C function wrapper
assembler code to setup system call no, arguments
trap to kernel
dispatch to correct routine
check arguments for errors
do requested service
return from kernel trap to user mode
return to C wrapper - check for error return

File System provides:
- An abstraction on top of the physical media
- A high level resource management scheme
- Protection between processes and users
- Sharing between processes and users

File Metadata: name, id, type, size, protection, time, date and owner info, Table of contents(TOC)

TOC: info for the FS to determine how to access file

Unix File Types
- regular file: normal ASCII/binary data (including executables)
- directories: special file that map filenames to files
- Special files:
  - devices: character & block devices
  - symbolic links: a soft link
  - named pipes
- Other non-native file types: eg. DOS, CDROMs, NFS (network file system)
- proc: special file interface to kernel internals

File Data: Structure
- **Array of bytes**:
  - The traditional Unix view
  - No interpretation of data: **just raw bytes**
  - Each byte has an unique **offset (distance)** from the first start
- **Fixed length records:**
  - Array of records, can grow/shrink
  - Can jump to any record easily:
  - Offset of the N^{th} record = size of Record * (N-1)
- **Variable length records**
  - Flexible but harder to locate a record

Fit Policies
- **First Fit**: first block which can fit. Can result in front of list being split more giving many small blocks at start
- **Best Fit**: find smallest fitting block. Requires searching whole list!
  - **Worst case: can result in bad external fragmentation with many tiny blocks**
  - **Experimentally good memory use results**
- **Next Fit**: like first fit but search from previously searched position. Avoids accumulation of small blocks at start. May have poor locality (affects caches)

File Data: Access Methods
- **Sequential Access:**
  - Cannot skip but can be rewound
  - implicitly may have offset position in file – access from current offset
- **Random Access:**
  - Data can be read from anywhere in file
  - Can be provided in two ways:

1. Read(Offset): Every read operation explicitly state the position to be accessed

2. Seek(Offset): A special operation is provided to move to a new location in file
E.g. Unix and Windows uses (2)

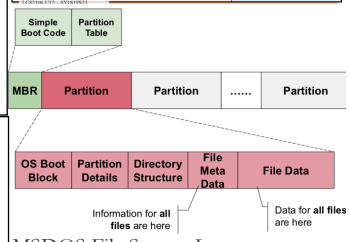
- **Direct Access:**
  - Used for file contains fixed-length records
  - Allow random access to any record directly
  - Very useful where there is a large amount of records e.g. In database

Allocated blocks:
size = **requested memory size** + (**alignment padding may be needed?**) **hidden field(s)**: block size (**required for malloc usage**)

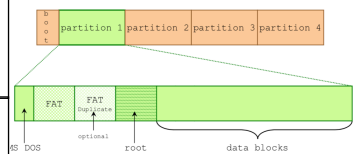
Free blocks:
since its free, can use entire block for any data structures! block size (size of this free block) link pointers
E.g: doubly linked list + size = ~3 words for bookkeeping imposes minimum free block size ≥ 3 words

File System: General Criteria
- **Self-Contained**:
  - Information stored on a media is enough to describe the entire organization
  - "plug-and-play"
- **Persistent:**
  - Beyond the lifetime of OS and processes
  - Does not need power (note: RAM requires power)
- **Efficient:**
  - Provides good management of free and used space
  - Small overheads for bookkeeping information

	Memory Management	File System Management
Underlying Storage	RAM	Disk
Access Speed	Constant (details depend on caches, so approximately constant)	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	**Virtual Memory:** determined by HW & OS	Many different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS) etc.



MSDOS File System Layout



Notes: FS corresponds to 1 volume (logical drive, eg. C:), stored in 1 partition, eg: another drive (D:) in another partition etc. We dont cover extended partition

Inodes
- actual file object
- every file has one inode (many to one mapping because of hard links)
- contains all meta-data about file **except filename**
- contains **reference count**, i.e. # hard links, reference count = 0 means file can be deleted
- meta-data in inode includes **Table of Contents** (TOC) which gives mapping of file data to disk blocks (**TOC** is **per file** - contrast with MSDOS which has only **1 global TOC** (FAT))
  - inode TOC: hybrid multi-level index structure

TOC index blocks
- Direct block pointers:
used for small files, no extra disk overhead, efficient direct access.
VM analogy: TLB (however not a cache)
- Single indirect block:
files bigger than direct blocks & smaller than double indirect. Disk overhead is 1 block.
File access slightly slower than direct.
VM analogy: direct mapped page table
- double + triple indirect blocks:
files bigger than single indirect block (usually not needed) . More disk overhead but is small fraction of file size. Random file access requires looking up the indirection blocks – slower than indirect.
VM analog: 2-3 level page tables

File Read/Write Model
- **Read operation**:
  - reads chunk of data representing portion of file to buffer in process memory – may be reading some existing blocks from disk
- **Write operation**:
  - updates existing chunk of data representing portion of file using buffer in process memory – may be **rewriting** some existing blocks
  - **appending** new data – may be creating new disk blocks for file
  - **combination** of update + append

Contiguous Block Allocation
- **Pros:**
  - Simple to keep track
  - Fast access – disk blocks are consecutive
- **Cons:**
  - **External Fragmentation**
  - File size need to be specified in advance

Linked List
- **Pros:**
  - Solve fragmentation problem
- **Cons:**
  - Random access in a file is very slow – pointer following at disk speed
  - Part of disk block is used for pointer

DOS Directory
- special file (type directory) containing directory entries (32 byte structures, little endian)
- **fat directory entry**: **filename + extension** (8+3 bytes), file **attributes** [1 byte: readonly, hidden, system, **directory flag** (distinguish directories from normal files), archive, volume label (disk volume name is dir entry)], time+date created, last access (read/write) date, time+date last write (creation is write)
  - **first block** (**cluster**) **number**
- **root directory** is special (already known, FAT16 has limited root dir size, 512 entries), other directories distinguished by type

- **Symbolic Link: can be file or directory**
  - B creates a ***special link file***, **G**
- **G** contains the path name of **F**
  - When **G** is accessed:
- Contents of G gives pathname for F
- can be recursive if F is also symbolic link
  - Unix Command: "**ln –s**"
- **In Unix:**
  - Symbolic link can link to any file
- file need not exist, can be directory
- General Graph **can be created**
- **maximum traversal limit**

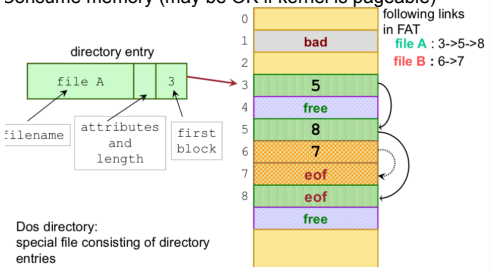
Link + Unlink
- Create new file: new directory entry with new inode
- Hard link: new directory entry (in appropriate dir) with inode of the linked file: eg. **link(path1, path2)**
add new directory entry to dir2: [**i1**, **file2**];
- Deleting (deleting is just **unlink** since graph is DAG): remove directory entry, decrement inode link count, free file object when link count = 0 (plus open fd's condition)

Data Structures on Disk
- need data structure to record which block belongs to which part of file, eg. data at positions 2020-4100 are in which blocks and which part of the block
- How does data in file change?
  - write more data at end of file
  - Unix: write data into holes! basically means can write anywhere!
  - decreases with truncation operation (unlike memory management, no free() !)
- data structure must also be stored on disk (persistent)
- typical data structures: versions of list / trees / arrays
Notes: very similar to data structures for dynamic memory management and page tables

General Disk Structure:
- Can be treated as a 1-D array of **logical blocks**
- Logical block:
- Smallest accessible unit (Usually 512-bytes to 4KB)
- Logical block is mapped into **disk sector(s)**
- Layout of disk sector is **hardware dependent**

Linked List V2.0
- **General Idea:**
  - Move all the block pointers into a single table known as **F**ile **A**llocation **T**able **(FAT)**

FAT is in memory at all time
- Simple yet efficient : Used by MS-DOS
- **Pros:**
  - Faster Random Access
The linked list traversal now takes place in memory
- **Cons:**
  - FAT keep tracks of **all disk blocks** in a partition
FAT size depends on disk size (O(disk))
Consume memory (may be OK if kernel is pageable)



MSDOS FAT16 Cluster Size
- FAT16: fat entry block # is **16 bits** (16 bit numbers in FAT entries), sector size=512 -> 64K*512 (32M)
- Logical block size = multiple of sectors. MSDOS calls this the **cluster size**
- Maximum cluster size = 32K (in some versions)
- Maximum file system size for FAT16 = 64K*32K = 2G
- Maximum file size: slightly less than 2G
- large cluster size means large internal fragmentation!

MSDOS FAT
- MSDOS uses **File Allocation Table** (FAT)
- Linked allocation but stored completely in FAT (after reading from disk)
- FAT kept in **RAM** (stored in disk but duplicated in RAM) – gives fast access to the pointers
- FAT table contains either: **block number** of next block, **EOF** code, **FREE** code, **BAD** block (block is unusable, i.e. disk error) – combines bitmap for free blocks with linked allocation for list of blocks
- FAT table is 1 entry for every block. Space management becomes an array method

The Problem of Disk Fragmentation

Fast disk access:

- contiguous blocks (from geometry/ processing view point)
- blocks in same cylinder

After some operations – block ordering becomes more **random**!

Disk Fragmentation: logical contiguous blocks are "**far apart**" on disk (this is different from memory fragmentation)

Notes: internal fragmentation still exists from block size

FAT:

- affects FAT FS
- fragmentation effect less with large cluster size (but **large** internal fragmentation!)
- MSDOS solution: run defragmentation (like compaction) on entire FS – move all used blocks to be contiguous .. one big free space chunk after defragmentation. May take a long time to defrag! (Windows: Disk Defragmenter)

Unix S5FS:

- also has disk fragmentation
- may be worse than DOS since smaller logical block size

Alternative: FS with **fragmentation resistance** (not necessarily optimal but dont need to defragment). Unix eg: BSD FFS, Linux Ext2/3 – not covered

File Descriptor Sharing

- fork
  - parent + child share file descriptors
  - what is a copy and what is shared?
    - fd value is a copy: just an int
    - internal in the kernel file object is shared
    - parent fd and child fd refer to same kernel file object
    - file offset (within file object) is shared
    - I/O in parent/child changes the common file offset (kernel file object)
    - so parent and child can write to the same file without overwriting each other
- in the shell – how do multiple child processes write to shell terminal (/dev/tty)?
  - parent and child use **stdout** file descriptor (fd = 1) for output
  - sequential write: write data and move shared offset for file descriptor 1
  - multiple child processes may interleave their writes due to concurrency

- parallel execution
  - parallel machine instruction execution
  - multiple instructions execute at the same time
- sequential special case: m = 1
- concurrency execution
  - do not distinguish between parallel execution or simulated parallelism (as in interleaved execution)

Race Condition: CS Approach

- Undesired execution is due to the **unsynchronized access to a shared modifiable resource**
- General outline of solution:
  - ***many possible approaches***
  - ***Designate code fragment with race condition as Critical Section (CS)***
  - At any point in time, only **one process** can execute in the critical section
- code in CS executes **atomically** – no interference from other instructions

➔Other process are prevented from entering the same critical section

Assumptions on CS

- **independence on non-CS**
  - CS not affected by code outside
  - can halt outside CS without affecting CS of other processes
- **CS takes finite time**
  - no infinite loop in CS
- **no assumption on execution speed**
  - instructions execute at non-zero speed
  - processor independent (on speed)

Properties of Correct CS Implementation

Mutual Exclusion:

• If process P_i is executing in critical section, all other processes are prevented from entering the critical section.

Progress:

• If no process is in a critical section, one of the waiting processes should be (eventually) granted access.

Bounded Wait:

• After process P_i requests to enter critical section, there exists an upper bound of number of times other processes can enter the critical section before P_i.

Problems with Synchronization of CS

- **Deadlock:**
  - **All** processes (or tasks/threads) blocked➔no progress
- **Starvation:**
  - **Some** processes are blocked forever
- **Livelock:**
  - Processes keep changing state (to avoid deadlock)

Peterson's Algorithm: Disadvantages

- uses atomic write of turn to resolve waiting (last offer of turn waits), can be generalized to n processes
- Busy Waiting:
  - The waiting process repeatedly test the while-loop condition

while (want[1] && (turn == 1)) ;

  - wastes CPU cycles with **busy waiting**
  - **may not work on modern hardware due to relaxed memory models**
- see special hardware instructions, e.g. **TestAndSet**
- Not general:
  - General synchronization mechanism is desirable
- Not just mutual exclusion

Producer Consumer: Busy Waiting

```
while (TRUE) {
  Produce Item;
  while (!canProduce);
  wait(mutex);
  if (count < K) {
    buffer[in] = item;
    in = (in+1)%K;
    count++;
    canConsume = TRUE;
  } else
    canProduce = FALSE;
  signal(mutex);
}
```

Producer Consumer: Busy Waiting

```
while (TRUE) {
  while (!canConsume);
  wait(mutex);
  if (count > 0) {
    item = buffer[out];
    out = (out+1)%K;
    count--;
    canProduce = TRUE;
  } else
    canConsume = FALSE;
  signal(mutex);
  Consume Item;
}
```

Producer Consumer: Blocking Version

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

Producer Consumer: Blocking Version

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

Dining Philosopher

```
int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher(int i) {
  while(TRUE) {
    Think();
    takeChpStcks(i);
    Eat();
    putChpStcks(i);
  }
}
```

```
void takeChpStcks(i) {
  wait(mutex);
  state[i] = HUNGRY;
  safeToEat(i);
  signal(mutex);
  wait(s[i]);
}
void safeToEat(i) {
  if(state[i]==HUNGRY && state[LEFT] !=
EATING && state[RIGHT] != EATING) {
    state[i] = EATING;
    signal(s[i]);
  }
}
```

```
void putChpStcks(i) {
  wait(mutex);
  state[i]=THINKING;
  safeToEat(LEFT);
  safeToEat(RIGHT);
  signal(mutex);
}
```

Dining Philosopher: Limited Eater

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

- **Semaphore:**
  - Generalized synchronization mechanism programming construct
  - Only behaviors are specified➔can have different implementations
  - Provides
- A way to block a number of processes, **blocked processes are sleeping** (no busy wait)
- A way to unblock/wake up one or more sleeping process

Semaphores: Properties

- Given: initial value
  - $S_{Initial} \geq 0$
- Then, the following **invariant** must be true: $S_{current} = S_{Initial} + \#signal(S) - \#wait(S)$
  - **#signal(S)**: ***number of signals() operations executed***
  - **#wait(S)**: number of **wait**() operations **complete**

Semaphore: Wait() and Signal()

- A semaphore **S** contains an integer value
  - Can be initialized to any non-negative values initially
- Two **atomic** semaphore operations:

`Wait( S ):`	`Signal( S ):`
If **S** <= 0, blocks (go to sleep)	**S++**
// S > 0	Wakes up one sleeping process if any
S--	This operation **never** blocks
Also known as `P()` or `Down()`	Also known as `V()` or `Up()`

- **Reminder:** The above specifies the **behavior**, not the implementations

Informal proof of mutex scheme

#CS: number of processes in CS but not finished

CS **Mutual Exclusion**:

want ($\#CS = \#wait(mutex) - \#signal(mutex)$) ≤ 1

initial mutex = 1

$mutex = 1 + \#signal(mutex) - \#wait(mutex)$ $mutex + \#CS = 1$

From $mutex \geq 0$

- we get $\#CS \leq 1$ (mutual exclusion with at most 1 process in CS)

Deadlock:

for deadlock – assume all processes stuck at wait(mutex)

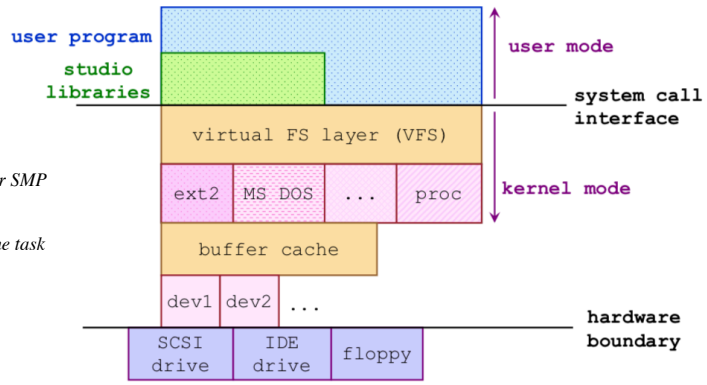
- So mutex=0 and #CS=0
- But mutex + #CS = 1 giving **contradiction**!

No deadlock

Starvation:

- assume 2 processes
- suppose P1 is blocked at wait(mutex)
- P2 is in CS, exits CS with signal(mutex)
- P1 can execute wait(mutex) so can eventually enter CS
- **no starvation** (assumed fairness or P2 cannot grab mutex)

Typical Unix/Linux FS Architecture




```
if (policy(cur_task)==RR && quantum expired(cur_task)) {
reset current_task counter to nice priority;
move current_task to end of queue;
}
REPEAT_SCHEDULE:
best = idle_task; c = −1000;
foreach t in runqueue {
  if (t not running on another cpu) { // true for uniprocessor
    w = goodness(t);
    if (w > c) { c=w; best=t; }
  }
}
if (c == 0) { // all ready tasks have expired quantum
RECALCULATE:
  foreach t in tasklist // apply to all tasks not just ready tasks
    t->counter = t->counter/2 + nice priority; // I/O boost
  goto START;
}
```

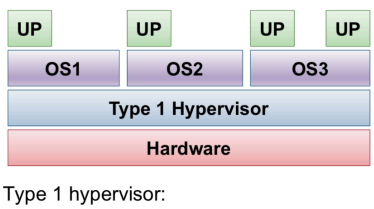
for each t in task list *// apply to **all tasks** not just ready tasks* t->counter = t->counter/2 + nice priority;
so effect of adjustment on counter value is **bounded** (at most double)
adjusts counter value which is used for quantum and also priority in goodness()

❑ higher priority (relative) to jobs which are: I/O-bound, suspended, waiting (includes interactive jobs which have those properties)

❑ quantum is also increased
❑ takes into account nice value
❑ increase is limited to 2*nice
❑ I/O boost heuristic: increases priority + quantum Tickless: non periodic interrupt
❑ only use timer interrupt when needed
❑ don't interrupt when CPU is in sleep mode ❑ adjust tick to next closest timer event
e.g: when current quantum must end, when some alarm must be triggered, ...
implementation may be more complex than it sounds ❑ *tickless* isn't no ticks but dynamic ticks

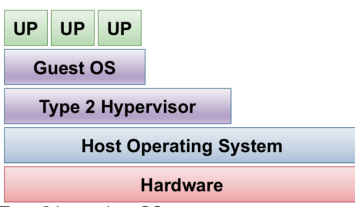
```
goodness(task t) {
if (realtime(t)) w = 1000 + realtime_priority(t);
else {
  w = t->counter;
  if (w > 0) {
    if (t->cpu == current cpu) // prefer same CPU for SMP
      w += PROC_CHANGE_PENALTY;
    if (address space(t) == current address space)
      w += 1; // prefer if TLBs are unchanged, e.g. same task
    w += 20 – t->nice; // factor in task nice value
  }
}
return w;
}
```

Type 1 Hypervisor

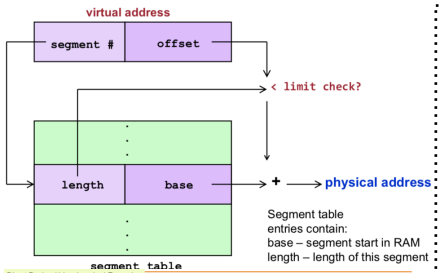


- Type 1 hypervisor:
  - Provides individual *virtual machines* to guest OSes
  - eg. IBM VM/370

Type 2 Hypervisor



- Type 2 hypervisor OS
  - Runs in host OS
  - Guest OS runs inside Virtual Machine
  - e.g. VMware



Segment table entries contain:
base – segment start in RAM
length – length of this segment

- Each memory segment:
  - Has a name (or ID)
- For ease of reference
  - Has a starting address (base)
  - Has a limit
- Indicate the range of the segment
- All memory reference is now specified as: Segment name + Offset

Entries in segment table contain information about each segment:
- base
- length
- permission information, eg. permission for read, write, execute (invalid memory operation - segmentation fault)

Limits of Segmentation

- External fragmentation (does not solve external fragmentation but compaction is easier with segmentation)
- Segment resizing can cause copying + relocation, fragmentation problems
- Not completely transparent to program, may require explicit segments to be used
- Swapping overhead is high for large segments

Recall: compaction is difficult in general without VM support because of difficulty of relocating pointers, segmentation allows for virtual addresses

Paging: Basic Idea

- Divide virtual address space into equal size segments: **pages,** page size is **power of 2** (why?)
- Divide physical memory into same equal size segments: frames
- **page size small**, eg. 4K for x86
- Virtual address: <*page id*, *offset*>

Special kind of segmentation scheme:
- same size – no length needed
- segment size is small
- very large number of segments
- segment number **embedded** in logical address -> part of address

Properties of Paging

- **No external fragmentation** – pages which are contiguous in virtual address space can be mapped to any page frames (frames need not be contiguous)
- **Internal fragmentation** – basic allocation unit now 1 page
- mapping transparent to program
  - done by hardware translation
- rather than segment resizing – using paging means just use more pages
- overhead of page transfer to/from swap is on the order of a few disk blocks (compared with segmentation, e.g. disk block may be 512 bytes)

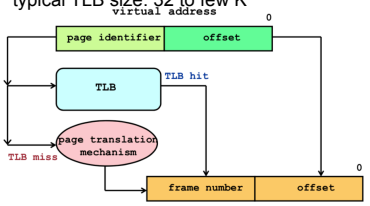
Role of OS

- setup/use translation hardware
  - segment registers
  - page tables
- PTE entries
- flags in PTE – protection, valid, dirty special registers (PTBR, PDBR)
- manage TLBs
- more in VM management

TLB

- cache to store mappings from page ids to frame numbers
- store the rest of the PTE entry (since PTE not read on TLB hit), i.e. other control bits, protection, etc

typical TLB size: 32 to few K



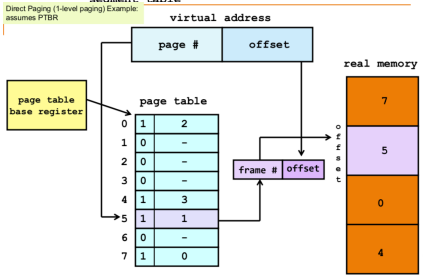
recall actual RAM access has CPU cycle latency which is not low (many 10s to 100s of cycles) Speedup with cache to remember the translation of pages to frames

TLB Flush:
What happens on context switch?
Process P1 → P2 Assuming independent page tables, have to change page table to P2. OS has to flush entries in TLB cache belonging to P1 Simplest way - just invalidate entire TLB cache
OS has to flush entries in TLB cache belonging to P1 Simplest way - invalidate entire TLB cache

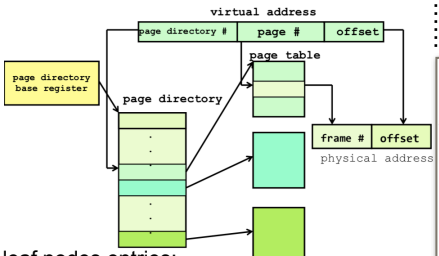
Various goals of VM:

- want to run more processes than can fit in memory
  - traditional goal
- allow process logical address space to exceed available memory
- give each process its own independent logical address space ❑ more important for modern OS
- reduce space wastage from physical memory management
- provide **protection** to address space of processes
- provide **sharing** of memory regions between processes

Direct Paging (1-level paging) Example: assumes PTBR



Each process can have its own mapping with its own page table by using PTBR



Space Savings with 2-Level Paging

Assume process only has 3 page tables of pages in use (remainder of virtual address space is unused)
2-level paging: 1 page directory + 3 page tables
let |page directory| = |page tables| = 10 bits
32 bit PTEs
page tables size = $4*(1+3)*2^{10}$ = 16K

1-level paging (Direct Paging) comparison
space used = $4*2^{20}$ = 4M

Memory byte addressed. How to store multibyte word (object > 1 byte), 2 general schemes, Eg: `0x87654321` (4 byte int)

- **Big Endian**: higher order bytes at lower addresses

Address	a	a+1	a+2	a+3
	87	65	43	21

- **Little Endian**: lower order bytes at lower addresses

Address	a	a+1	a+2	a+3
	21	43	65	87

leaf nodes entries:
- final **frame** corresponding to virtual page
- **valid** (**present**) bit – page present in RAM (invalid either page needs to be swapped in or is unused page – OS needs to figure out)
- **dirty** (modified) bit – set when page is written, dirty pages may have to be written to swap (used in replacement algorithms)
- **referenced** bit – set when page is used (supports replacement algorithms)
- **protection** bits – what operations can be done, eg. is read/write/execute allowed, not all combinations may be supported by h/w (e.g. may not have execute bit), may distinguish user from kernel (supervisor) mode (page may be accessed only from kernel mode)
- other bits, eg. unused bits which OS can use, page is not cacheable, PTE is not flushed

Virtual Address Translation Succeeds

OS not invoked:
- PTE hit or no fault during page translation
- **frame for page is in RAM** (valid)
  - page is present
- no protection violation
- VM translation automatically done by HW

This should be the **common case**:
no OS overheads, handled invisibly by HW (whenever possible), OS not invoked

Main page fault cases

Page fault occurs: run OS page fault handler
- **page not present**, e.g. **valid bit is off**

We now focus on the **page not present** case **can also be TLB miss if SW miss handler is used**
OS handles fixing the fault to make page valid. Note **recursion can occur if dealing with fault in page tables**
- **protection error** – OS deals with this

OS causes some "***OS software exception***" Unix: generates a **SIGSEGV** (segmentation fault) or **SIGBUS** signal
- **page is not used** (valid bit also off since no corresponding frame). Check OS memory maps to see if it is an error. Could be an (auto) expanding segment, eg. stack may not be error (Unix: alloca() allocates memory like malloc from stack)

Note: page fault is not a necessarily an OS error, its a hardware exception and might lead to actual error or not

Windows Blue Screen of Death

Windows kernel mode code:
- referencing invalid memory
- memory which doesn't belong to it
- referencing paged memory (rather than locked memory) at too high an IRQL. Page faults are not permitted at high interrupt request levels (IRQL)

Page Replacement Algo

FIFO
No hardware support needed
Belady's Anomaly -> FIFO doesn't exploit temporal locality

Least Recently Used Page Replacement
replace the page that has not been used in the longest time
1. Use A Counter
- Logical 'time' counters, which is incremented for every memory reference
- PTE with a "time-of-use" field

store the time counter value whenever reference occurs, replace the page with smallest "time-of-use"
Too expensive for actual h/w implementation
Need to search through all pages
"Time-of-use" is 4ever increasing (overflow!)

2. Use A Stack Data Structure
- Maintain a stack of page numbers
- If page X is referenced: remove from the stack, push on top of stack
- Replace the page at the bottom of stack

Not a pure stack: Entries can be removed from any where in the stack
Hard to implement in h/w
LRU cost too high: maintain LRU property at every memory reference

Second-Chance Page Replacement (CLOCK)
each PTE maintains a REF:
1=Accessed, 0=Not accessed
- Algorithm:

1. The oldest FIFO page is selected
2. If reference bit == 0 ➔ Page is replaced
3. If reference bit == 1 ➔ Page is given a 2nd chance
   - Reference bit cleared to 0
   - Arrival time reset➔page taken as newly loaded
   - Next FIFO page is selected, go to Step 2

Degenerate into FIFO algorithm
- When all pages has reference bit == 1

Full CLOCK Algorithm
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

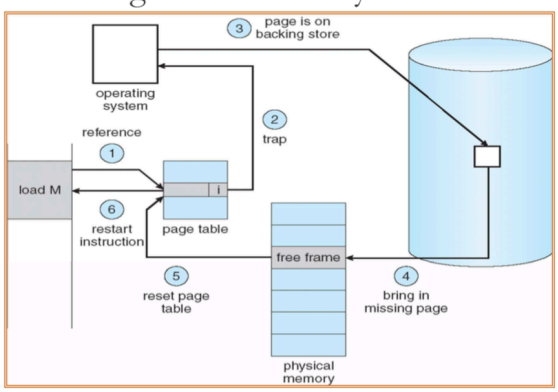
2. If step 1 fails, scan at most 1 revolution to **find first frame with <R=0,D=1>** and **reset R=0 for any frames scanned with R=1**
- deal with dirty pages – flush (write) to swap If step 2 fails, goto step 1

Can tailor scanning rate, eg: periodically scan X pages

- Hard to find the right number of frames:
  - If global replacement is used:
- A thrashing process "steals" page from other process ➔ cause other process to thrashing (**Cascading Thrashing**)
  - If local replacement is used:
- Thrashing can be limited to one process
- But that single process can hog the I/O and degrades the performance of other processes

- **Local Replacement:**
  - **Pros:**
- Frames allocated to a process remain constant ➔Performance is stable between multiple runs
  - **Cons:**
- If frame allocated is not enough ➔hinder the progress of a process
- **Global Replacement:**
  - **Pros:**
- Allow self-adjustment between processes
  - Process that needs more frame can get from other
  - **Cons:**
- Badly behave process can affect others
- Frames allocated to a process can be different from run to run

VM & Page Fault Memory Access



Note: assumes a page fault