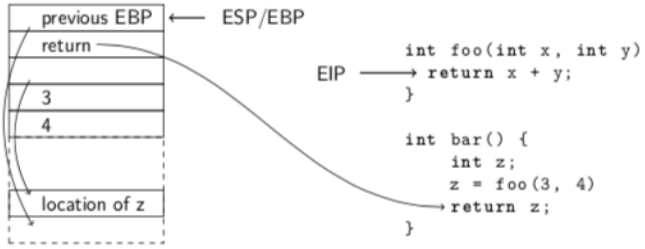
1, Processes

Allocate data structure on the heap:
int *var = (int*)malloc(sizeof(int));
The variable should point to a heap allocated area big enough to hold an integer.

Instruction pointer (EIP): a reference to the next instruction to execute
Stack pointer (ESP): a reference to the top of the stack
A base pointer (EBP): a reference to the current stack frame
General purpose registers: used by a process to store data



1,1 Locks

Unlock and signal must be done in a single instruction.

Threads in user space:
+ You can change scheduler.
+ Very fast task switching.
- If the process is suspended, all threads are.
- A process can not utilize multiple cores.

Threads in kernel space:
+ One thread can suspend while other continue to execute.
+ A process can utilize multiple cores.
- Thread scheduling requires trap to kernel.
- No way to change scheduler for a process.

All threads have their own stack, the heap is shared.

2, Scheduling

Turnaround time = $T_{completion} - T_{arrival}$ (SJF optimal)

Response time = $T_{firstrun} - T_{arrival}$ (Round robin optimal)

MLFQ - Gives each job a priority, uses a queue for each priority, put a job in one queue. Uses history to predict future. A lot of I/O -> High prio, CPU intensive -> low.

Deterministic stride - Each job have stride = N/#tickets where N is large and a pass value. Schedule job with lowest pass, increase its pass with its stride.

Rate monotonic scheduling = works if load < $n * \left(2^{\frac{1}{n}} - 1\right) \approx 69\%$ where n = #process. Schedule with lowest period (highest prio)

Earliest Deadline First: Always works if utilization <100%

3, Memory Management

Adress translation

Base/bounds (segmentation)

Simple, just two registers, base = offset, bounds = the size/max virtual address. Efficient (speed), transparent and protection, but inefficient memory use (between heap & stack) Can relocate processes in memory. Hard to fit program when address space doesn't fit in mem.

In average it's 50% internal fragmentation per segment when utilizing paging.

Architecture for paging

32 bitar ordlängd
512 byte pages
För att kunna adressera en sida på 512 byte krävs ett offset på 9 bitar (2^9). Förslagsvis kan vi ha ett index på 7 bitar om tre nivåer. Varje tabell kan då hålla 128 (2^7) element. Om vi har 4 byte per element blir tabellstorleken 512 byte (4*128). Adressrymden är således 3*7 + 9 = 30 bitar.

Example of a 2 level page table:

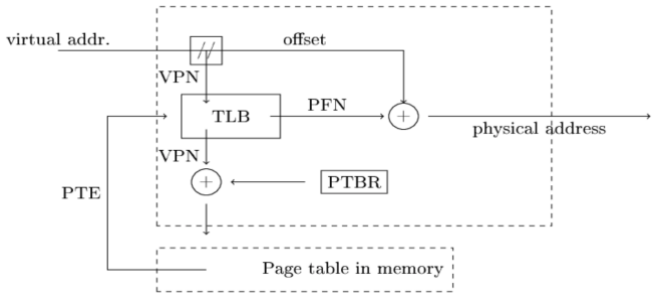
16 KB (14 bit) address space
64 byte pages
#pages = 16KB/64B = 21 4/26 = 28 = 256 pages
PT size = #pages*size of PTE = 256 * 4 byte = 1KB
PDsize=1KB/64B=21 0/ 26 =24 = 16 pages of PTEs
PTEs per page of PTEs = page size/PTE size = 64/4 = 16
Given a virtual address how to index:
PD size = 16 -> Need 4 bits to index
PTEs per page = 16 -> Need 4 bits to index Lowest 6 bits are offset

Inverted page table

One page (hash) table to rule them all.
One entry per physical page: Which process and the Virtual Page Number

Clock Algorithm - Approximates LRU. Add a use bit that is set to 1 when page is referenced. OS clears use bit. Don't evict dirty pages (requires a write too!) Hand points at a page. When replacement must occur -> check:
Page use bit 1?
Set to 0, move to next.
Otherwise replace it.

MMU



Buddy algorithm: Free a block of 32 bytes. Minimum block is 16 bytes. Therefore flip second bit 0b001010 -> 0b001000.

First Fit

+Fastest algorithm because it searches as little as possible.
-The remaining unused memory areas left after allocation become waste if it is too smaller. Thus request for larger memory requirement cannot be accomplished.

Best Fit

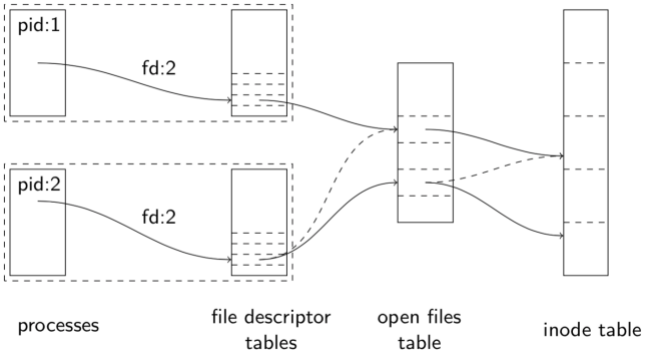
+Memory utilization is much better than first fit as it searches the smallest free partition first available.
-It is slower and may even tend to fill up memory with tiny useless holes.

Worst fit

+Reduces the rate of production of small gaps.
-If a process requiring larger memory arrives at a later stage then it cannot be accommodated as the largest hole is already split and occupied.

5, File systems

En structure CR(checkpoint region) finns på fast position och håller pekare till den sist skrivna inode-mapen. Inode-map mappar inode-id till block. CR måste uppdateras efter varje skrivning.



descriptor table: one table per process, copied when process is forked
open files table: table is global, one entry per open operation
inode table: one table per file system, one entry per *file object*
inode holds
mode: access rights
number of links: when zero the file is deleted from FS
user id: the owner of the file
group id: the associated group
size: file size in bytes
blocks: how many data blocks have been allocated
identifiers:

Data structures
Divide disk into blocks of (equal size is easy, 4KB)
Reserve one large group of blocks for **user data**,
Reserve one (smaller) for **inode-table** for metadata
Reserve blocks for **allocation structures**, one for inodes, one for data
E.g. bitmaps where each bit maps to a object/block 0=free, 1=used
Reserve blocks for **superblock**, info about file system
How many inodes and blocks, where inode-table begins etc.

Journaling
Write to disk in a well known location what you're about to do Then overwrite structures
A slow type of journaling, everything is written twice
Commit change
Transaction id, inode id, bitmap id, data block id Updated inode
Updated bitmap
Updated data block (write direct to block is faster)
Transaction id
Perform changes
Update blocks, inode, bitmap, data block Remove transaction (commit)

Virtual machines
How is it done?
Limited direct execution for each OS handled by the hypervisor
Hypervisor handles machine switching (changes virtual machine)
Save entire machine state
Restore entire machine state
Hypervisor must intercept privileged instructions from other OSes to remain in control
Hypervisor doesn't know how to handle a system call.
But it does know where the OS trap handler is! Because OS tried to install them OSes managed by hypervisor can run in
Supervisor mode
Can't execute privileged instructions
Can access a bit more memory (use this for OS structures)

User mode
Use memory protection (page tables & TLB) to protect OS structures
Hypervisor provides OS with virtual "physical" memory from its machine memory OS thinks it gets actual physical memory
TLB miss
OS does virtual to physical translations
Hypervisor makes physical to machine translations
Process: Trap to OS (ends up in hypervisor)
VMM: Call into OS trap handler
OS: Do page table lookup, update TLB
VMM: Update TLB with VPN-MFN (machine frame number)
OS: Return from trap
VMM: Return from trap Process: Continue execution