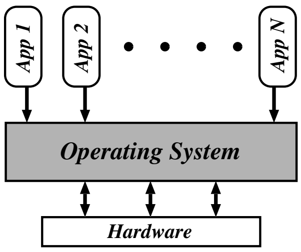
Part 2: Operating System Functions

- Introduction, evolution, structure
- Processes and scheduling
- Memory management
- File management
- note: I/O and device management will be (re)visited in part 3

What is an Operating System?

- A program which controls the execution of all other programs (systems and applications).
- Acts as an intermediary between the user(s) and the computer.
- Objectives:
  - convenience of software development,
  - ensure correct use of hardware,
  - sharing (several apps (several users)),
  - protection,
  - throughput,
  - service to *all*,
  - extensibility.

An Abstract View



- The Operating System (OS):
  - controls all execution.
  - multiplexes resources between applications.
  - abstracts away from complexity.
- Typically also have some *libraries* and some *tools* provided with OS.
- Are these part of the OS? Is IE4 a tool?
  - no-one can agree. . .
- For us, the OS $\approx$ the *kernel*.

In The Beginning. . .

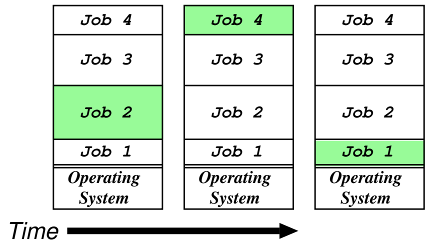
- 1949: First stored-program machine (EDSAC)
- to $\sim$ 1955: "Open Shop".
  - large machines with vacuum tubes.
  - I/O by paper tape / punch cards.
  - user = programmer = operator.
- To reduce cost, hire an *operator*:
  - programmers write programs and submit tape/cards to operator.
  - operator feeds cards, collects output from printer.
- Management like it.
- Programmers hate it.
- Operators hate it.

$\Rightarrow$ need something better.

Batch Systems

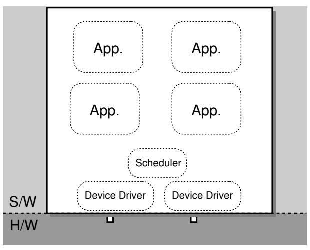
- Introduction of tape drives allow *batching* of jobs:
  - programmers put jobs on cards as before.
  - all cards read onto a tape.
  - operator carries input tape to computer.
  - results written to output tape.
  - output tape taken to printer.
- Computer now has a *resident monitor*:
  - initially control is in monitor.
  - monitor reads job and transfers control.
  - at end of job, control transfers back to monitor.
- Even better: spooling systems
  - use magnetic disk to cache input tape.
  - use interrupt driven I/O.
  - operator redundant?
- Monitor now *schedules* jobs. . .

Multi-Programming



- Use memory to cache jobs from disk ⇒ more than one job active simultaneously, e.g. OS/360
- Two stage scheduling:
  1. select jobs to load: *job scheduling*.
  2. select resident job to run: *CPU scheduling*.
- Where are programs loaded? - see memory management. Fixed partitions (waste space). Contiguous loading - leads to fragmentation.
- Users want more interaction ⇒ *time-sharing*: e.g. CTSS, Unix, VMS, Windows NT. . .

Monolithic Operating Systems



- Pre-multiprogramming OS structure, ("modern" examples are DOS, original MacOS)
- Problem: applications can e.g.
  - trash OS software.
  - trash another application.
  - hog CPU.
  - abuse I/O devices.
  - etc. . .
- No good for fault containment (or multi-user).
- Need a better solution. . .

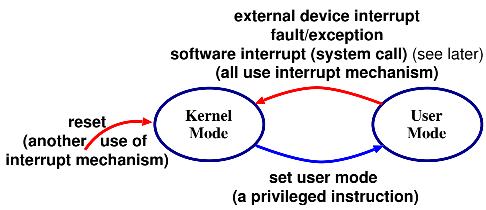
Dual-Mode Operation

- Want to stop buggy (or malicious) program from doing bad things.

⇒ provide *hardware* support to differentiate between (at least) two modes of operation.

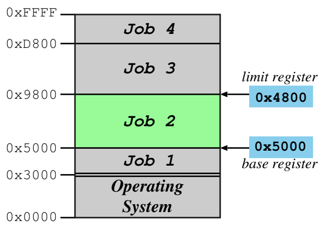
1. *User Mode* : when executing on behalf of a user (i.e. application programs).
2. *Kernel Mode* : when executing on behalf of the operating system.

- Hardware contains a mode-bit, e.g. 0 means kernel, 1 means user.
- Make certain machine instructions only possible in kernel mode. . .



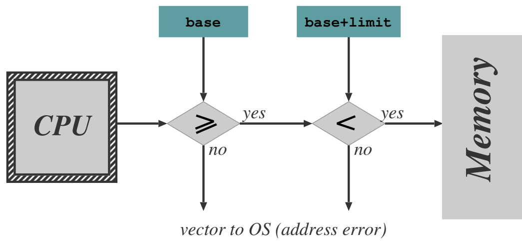
Protecting I/O & Memory

- First try: make I/O instructions privileged. e.g. DEC-10
  - ✔ applications can't mask interrupts.
  - ✔ applications can't control I/O devices.
- But:
  - ✘ Application can rewrite interrupt vectors.
  - ✘ Some devices accessed via *memory-mapped I/O*
- Hence need to protect memory also. . .
- e.g. define a *base* and a *limit* for each program.



- Accesses outside allowed range are detected.

Memory Protection Hardware

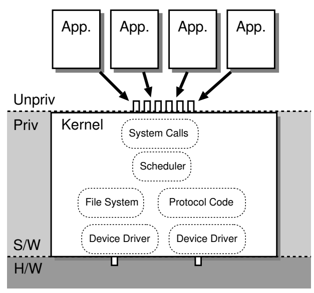


- Hardware checks every memory reference.
- Access out of range ⇒ vector into operating system (use interrupt/exception mechanism).
- Only allow *update* of base and limit registers in kernel mode.
- Typically disable memory protection in kernel mode (although a bad idea).
- In reality, more complex protection h/w used:
  - main schemes are *segmentation* and *paging*
  - (covered later on in course)

Protecting the CPU

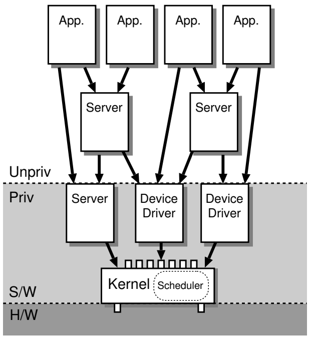
- Need to ensure that the OS stays in control.
  - i.e. need to prevent any given application from 'hogging' the CPU.
  - ⇒ use a *timer* device.
- Usually use a *countdown* timer, e.g.
  1. set timer to initial value (e.g. 0xFFFF).
  2. every *tick* (e.g. $1\mu s$), timer decrements value.
  3. when value hits zero, interrupt.
- (Modern timers have programmable tick rate.)
- Hence OS gets to run periodically and do its stuff.
- Need to ensure only OS can load timer, and that interrupt cannot be masked.
  - use same scheme as for other devices.
  - (viz. privileged instructions, memory protection)
- Same scheme can be used to implement time-sharing (more on this later).

Kernel-Based Operating Systems



- Applications can't do I/O due to protection
  - ⇒ operating system does it on their behalf.
- Need secure way for application to invoke operating system:
  - ⇒ require a special (unprivileged) instruction to allow transition from user to kernel mode.
- Generally called a *software interrupt* since operates similarly to (hardware) interrupt. . .
- Set of OS services accessible via software interrupt mechanism called *system calls*.

Microkernel Operating Systems



- Alternative structure:
  push some OS services into *servers*
  servers may be privileged (operate in kernel mode).
- Increases both *modularity* and *extensibility*.
- small kernel ⇒ known overhead. Delay between event and user-level response can be bounded.
- Still access kernel via system calls, but need new way to access servers ⇒ interprocess communication (IPC) schemes.

Kernels versus Microkernels

So why isn't everything a microkernel?

- Lots of IPC adds overhead ⇒ microkernels usually perform less well.
- Microkernel implementation sometimes tricky: need to worry about synchronisation.
- Microkernels often end up with redundant copies of OS data structures.

Hence today most common operating systems blur the distinction between kernel and microkernel.

- e.g. linux is “kernel”, but has kernel modules and certain servers.
- e.g. Windows NT was originally microkernel (3.5), but now (4.0 onwards) pushed lots back into kernel for performance.
- Still not clear what the best OS structure is, or how much it really matters. . .
- real-time systems need bounded OS delay

Part 2: Summary so far

You should now understand

- What an OS is (abstractly)
- Historical evolution of OS
- Hardware support needed
  - dual mode operation
  - protection (of devices, memory and CPU)
  - need for sharing (of devices, memory and CPU)
  - interrupt mechanism
  - timers
- Different approaches to OS design

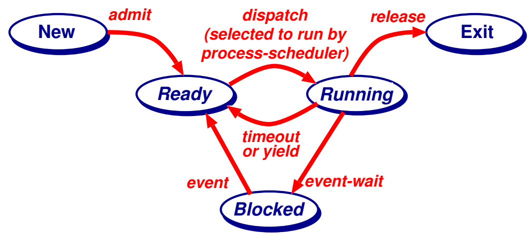
Operating System Functions

- Regardless of structure, OS needs to *securely multiplex resources*, i.e.
  1. protect applications from each other, yet
  2. share physical resources between them.
- tradeoffs:
  - protection -vs- sharing
  - throughput -vs- service to all
- Also usually want to *abstract* away from harware details, i.e. OS provides a *virtual machine*:
  - share CPU (in time) and provide each application with a virtual processor,
  - allocate and protect memory, and provide applications with their own virtual address space,
  - present a set of (relatively) hardware independent virtual devices, and
  - divide up storage space by using filing systems.

Process Concept

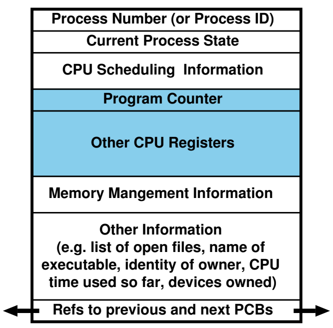
- From a user's point of view, the operating system is there to execute jobs (batch systems) or programs (interactive systems).
- A process is a program/job in execution (Think of "program/job" in their *executable* form after compilation and linking)
- a program/job is *static*, while a process is *dynamic* like a book or music manuscript cf. reading or playing them
- Process includes:
  1. program counter
  2. stack (for temporary variables, procedure parameters, return addresses. Defines dynamic state/scope.)
  3. data section (for global variables - always in scope.)
- Abstraction: processes execute on *virtual processors*

Process States



- As a process executes, it changes *state*:
  - *New*: the process is being created
  - *Ready*: the process is waiting for the CPU (and is prepared to run at any time)
  - *Running*: instructions are being executed
  - *Blocked*: the process is waiting for some event to occur (and cannot run until it does)
  - *Exit*: the process has finished execution.
- The operating system is responsible for maintaining the state of each process.

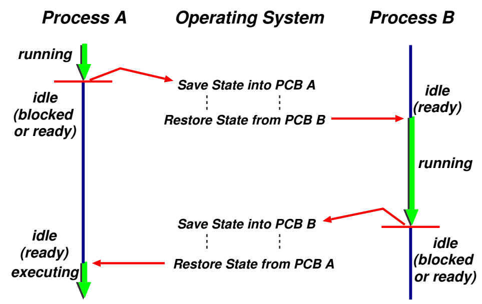
Process Control Block



OS maintains information about every process in a data structure called a *process control block* (PCB):

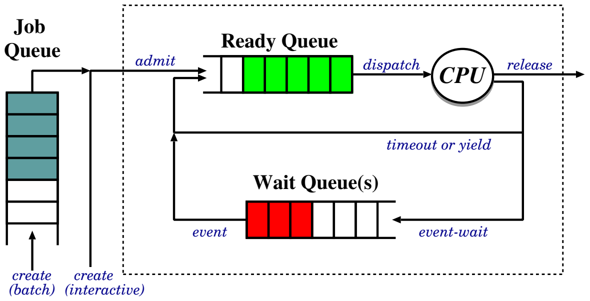
- Unique process identifier
- Process state (*Ready*, *Blocked*, etc.)
- CPU scheduling & accounting information
- Program counter & CPU registers
- Memory management information
- . . .

Context Switching



- *Process Context* = machine environment during the time the process is actively using the CPU.
- i.e. context includes program counter, general purpose registers, processor status register, . . .
- To switch between processes, the OS must:
  1. save the context of the currently executing process (if any), and
  2. restore the context of that being resumed.
- Time taken depends on h/w support.

Scheduling Queues



- Job Queue: batch processes awaiting admission.
- Ready Queue: set of all processes residing in main memory, ready and waiting to execute.
- Wait Queue(s): set of processes waiting for an I/O device (or for other processes)
- Long-term & short-term schedulers:
  - *Job scheduler* selects which processes should be brought into the ready queue.
  - *CPU scheduler* selects which process should be executed next and allocates CPU.

Process Creation

- Nearly all systems are *hierarchical*: parent processes create children processes.
- Resource sharing:
  - parent and children share all resources.
  - children share subset of parent's resources.
  - parent and child share no resources.
- Execution:
  - parent and children execute concurrently.
  - parent can wait until children terminate.
- Address space:
  - child duplicate of parent.
  - child has a program loaded into it.
- e.g. Unix:
  - `fork()` system call creates a new process
  - all resources shared (child is a clone).
  - `execve()` system call used to replace the process' memory space with a new program.
- NT/2000: `CreateProcess()` system call includes name of program to be executed.

Process Termination

- Process executes last statement and asks the operating system to delete it (**exit**):
  - output data from child to parent (**wait**)
  - process' resources are deallocated by the OS.
- Process performs an illegal operation, e.g.
  - makes an attempt to access memory to which it is not authorised,
  - attempts to execute a privileged instruction
- Parent may terminate execution of child processes (**abort**, **kill**), e.g. because
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - parent is exiting ("cascading termination")
  - (many operating systems do not allow a child to continue if its parent terminates)
- e.g. Unix has `wait()`, `exit()` and `kill()`
- e.g. NT/2000 has `ExitProcess()` for self and `TerminateProcess()` for others.

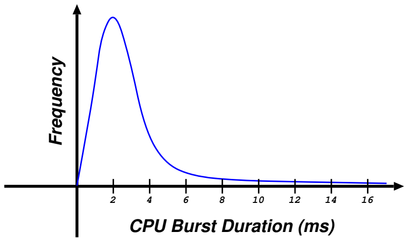
Process Blocking

- In general a process blocks on an *event*, e.g. until
  - an I/O device completes an operation,
  - another process sends a message
- Assume OS provides some kind of general-purpose blocking primitive, e.g. `await()`.
- Need care handling *concurrency* issues, e.g.

```
if(no key being pressed) {
   await(keypress);
   print("Key has been pressed!\n");
}
// handle keyboard input
```

  What happens if a key is pressed at the first '{' ?
- See part 3 for concurrency control

CPU-I/O Burst Cycle



- CPU-I/O Burst Cycle: process execution consists of a *cycle* of CPU execution and I/O wait.
- Processes can be described as either:
  1. I/O-bound: spends more time doing I/O than computation; has many short CPU bursts.
  2. CPU-bound: spends more time doing computations; has few very long CPU bursts.
- Observe most processes execute for at most a few milliseconds before blocking

⇒ need multiprogramming to obtain decent overall CPU utilization.

CPU Scheduler

Recall: CPU (process or thread) scheduler selects one of the ready processes and allocates the CPU to it (dispatches it).

- There are a number of occasions when we can/must choose a new process to run:
  1. a running process blocks (`running` → `blocked`)
  2. a timer expires (`running` → `ready`)
  3. a waiting process unblocks (`blocked` → `ready`)
  4. a process terminates (`running` → `exit`)
- If only make scheduling decision under 1, 4 ⇒ have a *non-preemptive* scheduler:

✔ simple to implement
✘ open to denial of service
– e.g. Windows 3.11, early MacOS.

- Otherwise the scheduler is *preemptive*.

✔ solves denial of service problem
✘ more complicated to implement
✘ introduces concurrency problems. . .

Idle system

What do we do if there is no ready process?

- halt processor (until interrupt arrives)

✔ saves power (and heat!)
✔ increases processor lifetime
✘ might take too long to stop and start.

- busy wait in scheduler

✔ quick response time
✘ ugly, useless

- invent idle process, always available to run

✔ gives uniform structure
✔ could use it to run checks
✘ uses some memory
✘ can slow interrupt response

In general there is a trade-off between responsiveness and usefulness.

Scheduling Criteria

A variety of metrics may be used:

1. CPU utilization: the fraction of the time the CPU is being used (and not for idle process!)
2. Throughput: # of processes that complete their execution per time unit.
3. Turnaround time: amount of time to execute a particular process.
4. Waiting time: amount of time a process has been waiting in the ready queue.
5. Response time: amount of time it takes from when a request was submitted until the first response is produced (in time-sharing systems)

Sensible scheduling strategies might be:

- Maximize throughput or CPU utilization
- Minimize average turnaround time, waiting time or response time.

Also need to worry about *fairness* and *liveness*.

First-Come First-Served Scheduling

- FCFS depends on order processes arrive, e.g.

Process	Burst Time
P_1	25
P_2	4
P_3	7

- If processes arrive in the order P_1, P_2, P_3:

  - Waiting time for P_1=0; P_2=25; P_3=29;
  - Average waiting time: $(0 + 25 + 29)/3 = 18$.
- If processes arrive in the order P_3, P_2, P_1:

  - Waiting time for P_1=11; P_2=7; P_3=0;
  - Average waiting time: $(11 + 7 + 0)/3 = 6$.
  - i.e. three times as good!
- First case poor due to *convoy effect.*

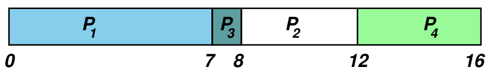
SJF Scheduling

Intuition from FCFS leads us to *shortest job first* (SJF) scheduling.

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the process with the shortest time (FCFS can be used to break ties).

For example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Waiting time for P_1=0; P_2=6; P_3=3; P_4=7;
- Average waiting time: $(0 + 6 + 3 + 7)/4 = 4$.

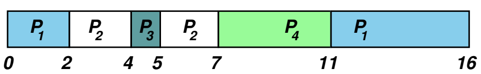
SJF is optimal in that it gives the minimum average waiting time for a given set of processes.

SRTF Scheduling

- SRTF = Shortest Remaining-Time First.
- Just a preemptive version of SJF.
- i.e. if a new process arrives with a CPU burst length less than the *remaining time* of the current executing process, preempt.

For example:

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Waiting time for P_1=9; P_2=1; P_3=0; P_4=2;
- Average waiting time: $(9 + 1 + 0 + 2)/4 = 3$.

What are the problems here?

Predicting Burst Lengths

- For both SJF and SRTF require the next "burst length" for each process $\Rightarrow$ need to estimate it.
- Can be done by using the length of previous CPU bursts, using exponential averaging:
  1. t_n = actual length of n^{th} CPU burst.
  2. τ_{n+1} = predicted value for next CPU burst.
  3. For $\alpha, 0 \leq \alpha \leq 1$ define:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

- If we expand the formula we get:

$$\tau_{n+1} = \alpha t_n + \ldots + (1-\alpha)^j \alpha t_{n-j} + \ldots + (1-\alpha)^{n+1}\tau_0$$

  where τ_0 is some constant.
- Choose value of α according to our belief about the system, e.g. if we believe history irrelevant, choose $\alpha \approx 1$ and then get $\tau_{n+1} \approx t_n$.
- In general an exponential averaging scheme is a good predictor if the variance is small.

Round Robin Scheduling

Define a small fixed unit of time called a *quantum* (or *time-slice*), typically 10-100 milliseconds. Then:

- Process at the front of the ready queue is allocated the CPU for (up to) one quantum.
- When the time has elapsed, the process is preempted and appended to the ready queue.

Round robin has some nice properties:

- Fair: if there are n processes in the ready queue and the time quantum is q, then each process gets $1/n^{th}$ of the CPU.
- Live: no process waits more than $(n-1)q$ time units before receiving a CPU allocation.
- Typically get higher average turnaround time than SRTF, but better average *response time*.

But tricky choosing correct size quantum:

- q too large $\Rightarrow$ FCFS/FIFO
- q too small $\Rightarrow$ context switch overhead too high.

Static Priority Scheduling

- Associate an (integer) priority with each process
- For example:

0	system internal processes
1	interactive processes (staff)
2	interactive processes (students)
3	batch processes.

- Then allocate CPU to the highest priority process:
  - 'highest priority' typically means smallest integer
  - get preemptive and non-preemptive variants.
- e.g. SJF is a priority scheduling algorithm where priority is the predicted next CPU burst time.
- Problem: how to resolve ties?
  - round robin with time-slicing
  - allocate quantum to each process in turn.
  - Problem: biased towards CPU intensive jobs.
    - per-process quantum based on usage?
    - ignore?
- Problem: starvation. . .

Dynamic Priority Scheduling

- Use same scheduling algorithm, but allow priorities to change over time.
- e.g. simple aging:
  - processes have a (static) *base priority* and a dynamic *effective priority*.
  - if process starved for k seconds, increment effective priority.
  - once process runs, reset effective priority.
- e.g. computed priority:
  - first used in Dijkstra's THE
  - time slots: $\ldots, t, t+1, \ldots$
  - in each time slot t, measure the CPU usage of process j: u^j
  - priority for process j in slot $t+1$: $p^j_{t+1} = f(u^j_t, p^j_t, u^j_{t-1}, p^j_{t-1}, \ldots)$
  - e.g. $p^j_{t+1} = p^j_t/2 + ku^j_t$
  - penalises CPU bound $\rightarrow$ supports I/O bound.
- today such computation considered unacceptable. . .

Multilevel Queues

- Ready queue partitioned into separate queues, e.g.
  - foreground (interactive),
  - background (batch)
- Each queue has its own scheduling algorithm, e.g.
  - foreground: RR,
  - background: FCFS
- Scheduling must also be done between the queues:
  - Fixed priority scheduling; i.e., serve all from foreground and then from background. Possibility of starvation.
  - Time slice: each queue gets a certain amount of CPU time which it can divide between its processes, e.g. 80% to foreground via RR, 20% to background in FCFS.
- Also get *multilevel feedback queue*:
  - as above, but processes can move between the various queues.
  - can be used to implement dynamic priority schemes, among others.

Multilevel Feedback Queue

- Example: three queues
  1. Q_0, 8 millisecond quantum,
  2. Q_1, 16 millisecond quantum,
  3. Q_2, FCFS (run to completion).
- Processes enter tail of Q_0 and eventually get to execute for 8ms. If not finished, preempted and moved to tail of Q_1. Eventually gets to execute for 16ms. If still not complete, preempted and moved to tail of Q_2.

Processes - summary

You should now understand:

- What a process is
- Process states and PCBs
- Scheduling queues
- What a CPU scheduler does
- Criteria for scheduling
- Various strategies:
  - first-come first-server
  - shortest job first
  - shortest remaining time first
  - round robin
  - static and dynamic priorities
  - use of more than one scheduling queue

Memory Management

In a multiprogramming system:

- many processes in memory simultaneously
- every process needs memory for:
  - instructions ("code" or "text"),
  - static data (in program), and
  - dynamic data (heap and stack).
- OS also needs memory for its code and data.

⇒ must share memory between OS and k processes.

The memory magagement subsystem handles:

1. Relocation
2. Allocation
3. Protection
4. Sharing
5. Logical Organisation (OS + compiler + runtime system)
6. Physical Organisation

The Address Binding Problem

Consider the following simple program:

```
int x, y;
x = 5;
y = x + 3;
```

We can imagine that this would result in some assembly code which looks something like:

```
str #5, [x]      // store 5 into address x in memory
ldr R1, [x]      // load value of x from memory into R1
add R2, R1, #3   // add 3 to it - into R2
str R2, [y]      // store result in addr y in memory
```

note the distinction between address and contents, e.g. address [x] is loaded with value/contents 5

Then the address binding problem is:
what values do we give to addresses [x] and [y] ?

This is a problem because we don't know where in memory our program will be loaded when we run it:

- e.g. if loaded at 0x1000, then `x` and `y` might be stored at 0x2000, but if loaded at 0x5000, then `x` and `y` might be at 0x6000.

Address Binding and Relocation

To solve the problem, we need to translate between “program addresses” and “real addresses”.

This can be done:

- at compile time:
  - requires knowledge of absolute addresses
  - e.g. DOS .com files
- at load time:
  - when program loaded, work out position in memory and update code with correct addresses
  - must be done every time program is loaded
  - ok for embedded systems / boot-loaders
- at run-time:
  - get some hardware to automatically translate between program and real addresses.
  - no changes at all required to program itself.
  - most popular and flexible scheme, providing we have the requisite hardware (MMU).

beginslide

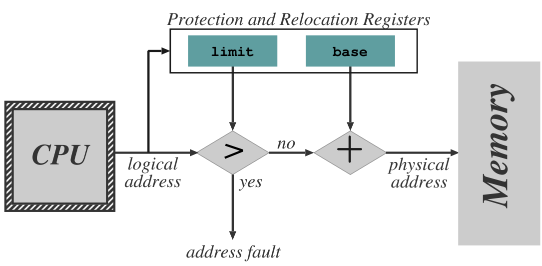
Static relocation - partitions (1970's)

How can we support multiple virtual processors in a single address space?

- statically divide memory into multiple fixed size partitions of different sizes:
  - e.g. bottom partition contains OS, remaining partitions each for exactly one process at once.
  - when a process terminates (or blocks) its partition becomes available to new processes, e.g. OS/360 MFT
  - a process is always loaded into the same partition (static address translation).
- BUT - need to protect OS and user processes from malicious programs:
  - need base and limit registers to restrict process to its partition
  - update values when a new processes is scheduled
  - NB: can be used for relocation as well as protection!
  - then don't need static partitions - processes can be loaded into any available, large-enough space.

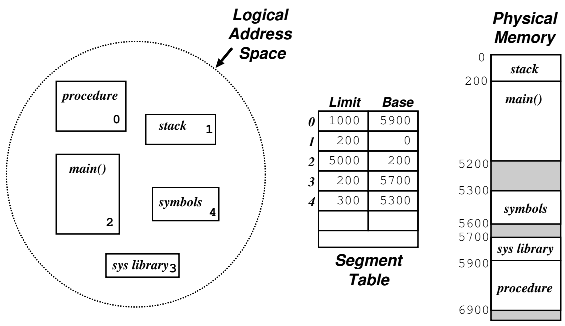
Dynamic address translation

Mapping of logical to physical addresses is done at run-time by Memory Management Unit (MMU), e.g.



1. Relocation register holds the value of the base address owned by the process.
2. Relocation register contents are added to each memory address before it is sent to memory.
3. e.g. DOS on 80x86 — 4 relocation registers, logical address is a tuple (s, o).
4. NB: process never sees physical address — simply manipulates logical addresses.
5. OS has privilege to update relocation register.

Segmentation



- User prefers to view memory as a set of segments of no particular size, with no particular ordering
- Segmentation supports this user-view of memory — logical address space is a collection of (typically disjoint) segments.
- Segments have a name (or a number) and a length — addresses specify segment and offset.

Implementing Segments

- Maintain a segment table for each process:

Segment	Access	Base	Size	Others!

- If program has a very large number of segments then the table is kept in memory, pointed to by ST base register STBR
- Also need a ST length register STLR since number of segs used by different programs will differ widely
- The table is part of the process context and hence is changed on each process switch.

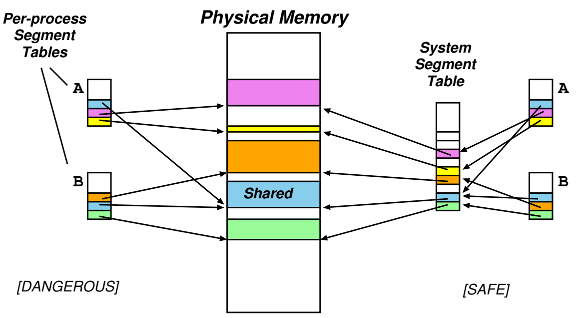
Algorithm:

1. Program presents address (s, d).
   Check that $s <$ STLR. If not, fault
2. Obtain table entry at reference $s+$ STBR, a tuple of form (b_s, l_s)
3. If $0 \leq d < l_s$ then this is a valid address at location (b_s, d), else fault

Sharing and Protection

- Big advantage of segmentation is that protection is per segment; i.e. corresponds to logical view.
- Protection bits associated with each ST entry checked in usual way
- e.g. instruction segments (should be non-self modifying!) thus protected against writes etc.
- e.g. place each array in own seg $\Rightarrow$ array limits checked by hardware
- Segmentation also facilitates sharing of code/data
  - each process has its own STBR/STLR
  - sharing is enabled when two processes have entries for the same physical locations.
  - for data segments can use copy-on-write (see later under paging).
- Several subtle caveats exist with segmentation — e.g. jumps within shared code.
- e.g. Multics, MU5 $\Rightarrow$ ICL 2900, George3 OS.

Sharing Segments



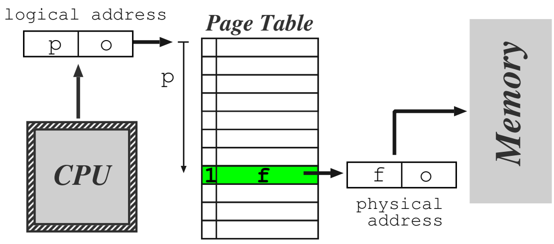
Sharing segments:

- wasteful (and dangerous) to store common information on shared segment in each process segment table
- assign each segment a unique System Segment Number (SSN)
- process segment table simply maps from a Process Segment Number (PSN) to SSN

Fragmentation Returns. . .

- Suppose that all segments of a process must be loaded in memory when it is scheduled to run - must find space for them all.
- Problem is that segs are of variable size $\Rightarrow$ leads to fragmentation of memory.
- Use best/first-fit, buddy algorithms etc.
- Processes may be delayed waiting for space to be made (by swapping out others' segs to compact memory - consolidate free space).
- Tradeoff between memory-compaction/delay depends on average segment size
- In general with small average segment sizes, fragmentation is small.
- Fixed size small segments $\equiv$ paging! - see below.
- Segmentation + paging means that not every segment need be loaded into memory when a process is scheduled - "demand paging". Hardware for "demand segmentation" is possible too.

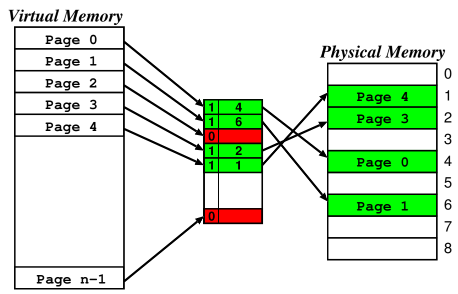
Paged Virtual Memory



Another solution is to allow a process to exist in non-contiguous memory, i.e.

- divide physical memory into relatively small blocks of fixed size, called *frames*
- divide logical memory into blocks of the same size called *pages* (typical value is 4K)
- each address generated by CPU is composed of a page number p and page offset o.
- MMU uses p as an index into a *page table*.
- page table contains associated frame number f
- usually have $|\mathtt{p}| \gg |\mathtt{f}| \Rightarrow$ need valid bit.

Paging Pros and Cons



- ✔ memory allocation easier.
- ✘ OS must keep page table per process
- ✔ no fragmentation of physical memory
- ✘ but get **internal fragmentation**.
- ✔ clear separation between user and system view of memory usage.
- ✘ additional overhead on context switching

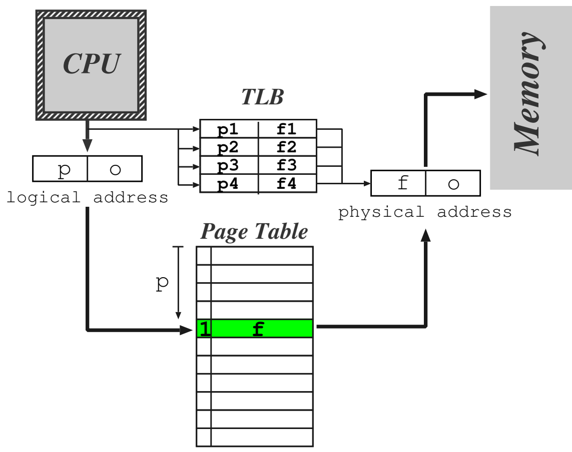
Structure of the Page Table

Different kinds of hardware support can be provided:

- Simplest case: set of dedicated relocation registers
  - one register per page
  - OS loads the registers on context switch
  - fine if the page table is small. . . but what if have large number of pages ?
- Alternatively keep page table in memory
  - only one register needed in MMU (page table base register (PTBR))
  - OS switches this when switching process
- Problem: page tables might still be very big.
  - can keep a page table length register (PTLR) to indicate size of page table.
  - or can use more complex structure (see later)
- Problem: need to refer to memory *twice* for every 'actual' memory reference. . .

$\Rightarrow$ use a translation lookaside buffer (TLB)

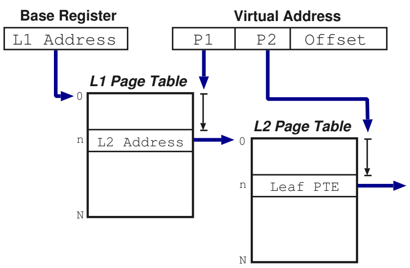
TLB Operation



- On memory reference present TLB with logical memory address
- If page table entry for the page is present then get an immediate result
- If not then make memory reference to page tables, and update the TLB

Multilevel Page Tables

- Most modern systems can support very large ($2^{32}, 2^{64}$) address spaces.
- Solution – split page table into several sub-parts
- Two level paging – page the page table



- For 64 bit architectures a two-level paging scheme is not sufficient: need further levels.
- (even some 32 bit machines have > 2 levels).

Protection Issues

- Associate protection bits with each page – kept in page tables (and TLB).
- e.g. one bit for read, one for write, one for execute.
- May also distinguish whether may only be accessed when executing in *kernel mode*, e.g.

Frame Number	K	R	W	X	V

- At the same time as address is going through page hardware, can check protection bits.
- Attempt to violate protection causes h/w trap to operating system code
- As before, have *valid/invalid* bit determining if the page is mapped into the process address space:
  - if invalid ⇒ trap to OS handler
  - can do lots of interesting things here, particularly with regard to sharing. . .

Shared Pages

Another advantage of paged memory is code/data sharing, for example:

- binaries: editor, compiler etc.
- libraries: shared objects, dlls.

So how does this work?

- Implemented as two logical addresses which map to one physical address.
- If code is *re-entrant* (i.e. stateless, non-self modifying) it can be easily shared between users.
- Otherwise can use *copy-on-write* technique:
  - mark page as read-only in all processes.
  - if a process tries to write to page, will trap to OS fault handler.
  - can then allocate new frame, copy data, and create new page table mapping.
- (may use this for lazy data sharing too).

Requires additional book-keeping in OS, but worth it, e.g. over 40Mb of shared code on my linux box.

Paged segments

Many systems (past and present) support(ed) both segmentation and paging.

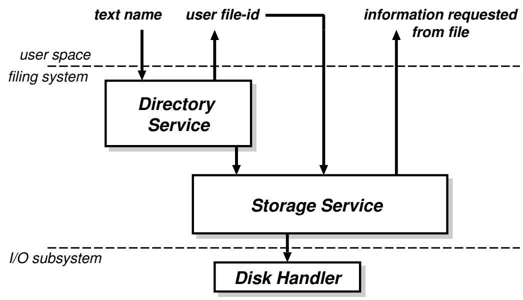
- Segments allow logical structure to be expressed - natural unit for protection and sharing.
- Segment page tables give natural multi-level page tables.
- - much research in 1970's . . .
- Paging supports efficient management of physical memory.
- Demand paging means that segments do not need to be loaded in advance of being addressed (not covered in detail). Pages are brought into memory by the OS when a page fault occurs in the TLB and the page is marked as not present in main memory in the process page table.

Summary of memory management

You should now understand:

- what memory management aims to achieve
- logical/virtual -vs- physical addresses
- static and dynamic address translation
- segmentation: pros and cons, hardware support
- paging: pros and cons, hardware support
- segmentation with paging

File Management



Filing systems have two main components:

1. Directory Service
   - maps from names to file identifiers.
   - handles access & existence control
2. Storage Service
   - provides mechanism to store data on disk
   - includes means to implement directory service

File Concept

What is a file?

- Basic abstraction for non-volatile storage.
- Typically comprises a single contiguous logical address space.
- Internal structure:
  1. None (e.g. sequence of words, bytes)
  2. Simple record structures
     - lines
     - fixed length
     - variable length
  3. Complex structures
     - formatted document
     - relocatable object file
- Can simulate last two with first method by inserting appropriate control characters.
- All a question of who decides:
  - operating system
  - program(mer).

Naming Files

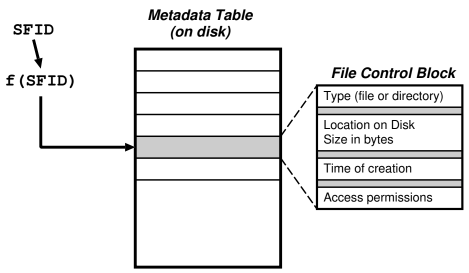
Files usually have at least two kinds of 'name':

1. System file identifier (SFID):
   - (typically) a unique integer value associated with a given file
   - SFIDs are the names used within the filing system itself
2. "Human" name, e.g. `hello.java`
   - What users like to use
   - Mapping from human name to SFID is held in a *directory*, e.g.

Name	SFID
hello.java	12353
Makefile	23812
README	9742

   - Directories also non-volatile ⇒ must be stored on disk along with files.
3. Frequently also get user file identifier (UFID).
   - used to identify *open* files (see later)

File Meta-data I



In addition to their contents and their name(s), files typically have a number of other attributes, e.g.

Location: file location on device (several schemes possible, see UNIX case study)
Size: current file size
Type: if system supports different types
Protection: controls who can read, write, etc.
Time, *date*, and *user identification*: data for protection, security and usage monitoring.

Together this information is called *meta-data*.
It is stored in a *file control block*.

File Meta-data II

From case studies and via background reading, see:

- *Location* via:
  chaining of disk blocks,
  chaining in a map,
  tables of pointers,
  indirect blocks,
  extent lists.
- hard and soft links
- "file types" may be generalised so that directories, devices and other objects may be named and accessed uniformly via the same naming structure and metadata.

Directory Name Space (I)

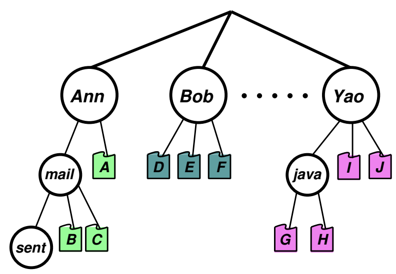
What are the requirements for our name space?

- Efficiency: locating a file quickly.
- Naming: user convenience
  - allow two (or more generally N) users to have the same name for different files
  - allow one file have several different names
- Grouping: logical grouping of files by properties (e.g. all Java programs, all games, . . .)

First attempts:

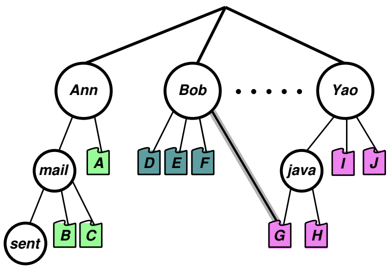
- Single-level: one directory shared between all users
  - ⇒ naming problem
  - ⇒ grouping problem
- Two-level directory: one directory per user
  - access via *pathname* (e.g. `bob:hello.java`)
  - can have same filename for different user
  - but still no grouping capability.

Directory Name Space (II)



- Get more flexibility with a general *hierarchy*.
  - directories hold files or [further] directories
  - create/delete files relative to a given directory
- Human name is full path name, but can get long: e.g. /usr/groups/X11R5/src/mit/server/os/4.2bsd/utils.c
  - offer relative naming
  - login directory
  - current working directory
- What does it mean to delete a [sub]-directory?

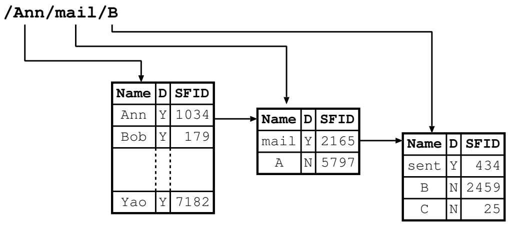
Directory Name Space (III)



- Hierarchy good, but still only one name per file.

⇒ extend to directed acyclic graph (DAG) structure:
  - allow shared subdirectories and files.
  - can have multiple *aliases* for the same thing
- Problem: dangling references
- Solutions:
  - back-references (but variable size records)
  - reference counts.
- Problem: cycles. . .

Directory Implementation



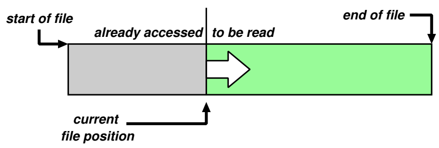
- Directories are non-volatile ⇒ store as "files" on disk, each with own SFID.
- Must be different *types* of file (for traversal)
- Explicit directory operations include:
  - create directory
  - delete directory
  - list contents
  - select current working directory
  - insert an entry for a file (a "link")

File Operations (I)

UFID	*SFID*	*File Control Block (Copy)*
1	*23421*	*location on disk, size,...*
2	*3250*	" "
3	*10532*	" "
4	*7122*	" "

- Opening a file: `UFID = open(<pathname>)`
  1. directory service recursively searches directories for components of `<pathname>`
  2. if all goes well, eventually get `SFID` of file.
  3. copy file control block into memory.
  4. create new `UFID` and return to caller.
- Create a new file: `UFID = create(<pathname>)`
- Once have `UFID` can read, write, etc.
  - various modes (see next slide)
- Closing a file: `status = close(UFID)`
  1. copy [new] file control block back to disk.
  2. invalidate `UFID`

File Operations (II)



- Associate a *cursor* or *file position* with each open file (viz. UFID), initialised to start of file.
- Basic operations: *read next* or *write next*, e.g.
  - `read(UFID, buf, nbytes)`, or
  - `read(UFID, buf, nrecords)`
- Sequential Access: above, plus `rewind(UFID)`.
- Direct Access: *read* N or *write* N
  - allow "random" access to any part of file.
  - can implement with `seek(UFID, pos)`
- Other forms of data access possible, e.g.
  - append-only (may be faster)
  - indexed sequential access mode (ISAM)

Other Filing System Issues

- Access Control: file owner/creator should be able to control what can be done, and by whom.
  - access control normally a function of directory service ⇒ checks done at file *open* time
  - various types of access, e.g.
    - read, write, execute, (append?),
    - delete, list, rename
  - more advanced schemes possible (see later)
- Existence Control: what if a user deletes a file?
  - probably want to keep file in existence while there is a valid pathname referencing it
  - plus check entire FS periodically for garbage
  - existence control can also be a factor when a file is renamed/moved.
- Concurrency Control: need some form of *locking* to handle simultaneous access
  - may be mandatory or advisory
  - locks may be shared or exclusive
  - granularity may be file or subset

Summary of Part 2

You should now understand:

- OS evolution
- alternative OS structures
- OS support for processes
- CPU scheduling
- memory management
  - hardware support for segmentation and paging
  - hardware-software interaction
  - pros and cons of segmentation and paging
- file management
  (UNIX case study contains examples)