Operating Systems

Steven Hand

Michaelmas / Lent Term 2008/09

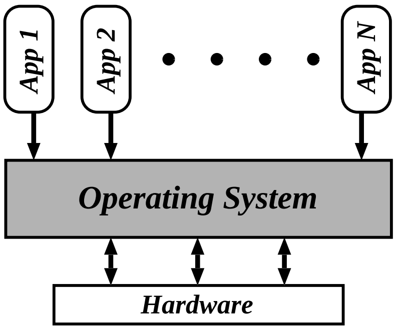
17 lectures for CST IA

Handout 3

What is an Operating System?

- A program which controls the execution of all other programs (applications).
- Acts as an intermediary between the user(s) and the computer.
- Objectives:
  - convenience,
  - efficiency,
  - extensibility.
- Similar to a government. . .

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 IE a tool?
  - no-one can agree. . .
- For us, the OS $\approx$ the *kernel*.

In The Beginning. . .

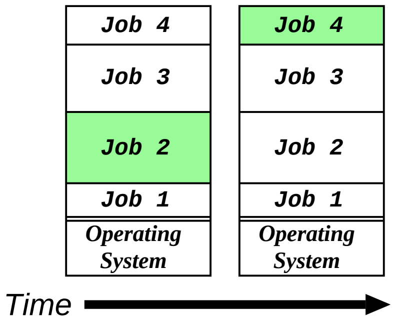
- 1949: First stored-program machine (EDSAC)
- to $\sim$ 1955: "Open Shop".
  - large machines with vacuum tubes.
  - I/O by paper tape / punch cards.
  - user = programmer = operator.
- To reduce cost, hire an *operator*:
  - programmers write programs and submit tape/cards to operator.
  - operator 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 transfer control.
  - at end of job, control transfers back to monitor.
- Even better: *spooling systems*.
  - use interrupt driven I/O.
  - use magnetic disk to cache input tape.
  - fire operator.
- Monitor now *schedules* jobs. . .

Multi-Programming

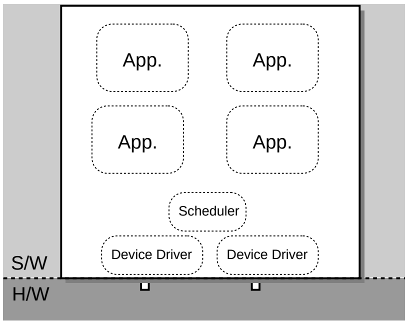


- Use memory to cache jobs from disk ⇒ more than one job active simultaneously.
- Two stage scheduling:
  1. select jobs to load: *job scheduling*.
  2. select resident job to run: *CPU scheduling*.
- Users want more interaction ⇒ *time-sharing*:
- e.g. CTSS, TSO, Unix, VMS, Windows NT. . .

Today and Tomorrow

- Single user systems: cheap and cheerful.
  - personal computers.
  - no other users ⇒ ignore protection.
  - e.g. DOS, Windows, Win 95/98, . . .
- RT Systems: power is nothing without control.
  - hard-real time: nuclear reactor safety monitor.
  - soft-real time: mp3 player.
- Parallel Processing: the need for speed.
  - SMP: 2–8 processors in a box.
  - MIMD: super-computing.
- Distributed computing: global processing?
  - Java: the network is the computer.
  - Clustering: the network is the bus.
  - CORBA: the computer is the network.
  - .NET: the network is an enabling framework. . .

Monolithic Operating Systems



- Oldest kind of OS structure (“modern” examples are DOS, original MacOS)
- Problem: applications can e.g.
  - trash OS software.
  - trash another application.
  - hoard CPU time.
  - abuse I/O devices.
  - etc. . .
- No good for fault containment (or multi-user).
- Need a better solution. . .

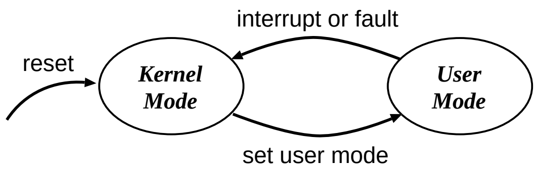
Dual-Mode Operation

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

⇒ provide *hardware* support to distinguish between (at least) two different 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.

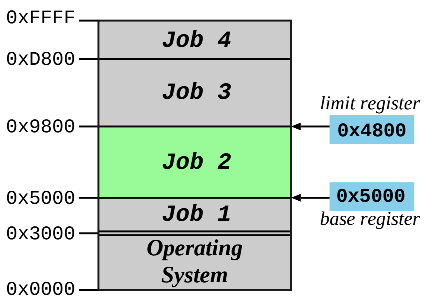
- Hardware contains a mode-bit, e.g. 0 means kernel, 1 means user.



- Make certain machine instructions only possible in kernel mode. . .

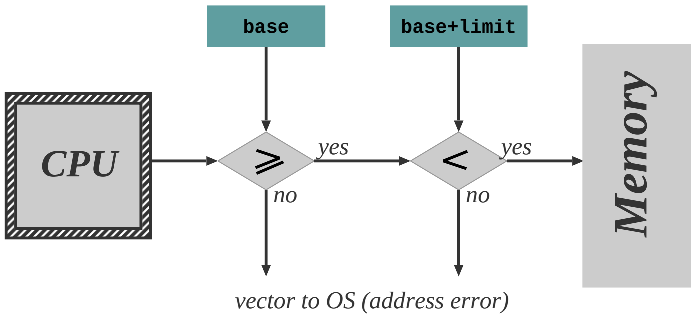
Protecting I/O & Memory

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



- Accesses outside allowed range are protected.

Memory Protection Hardware



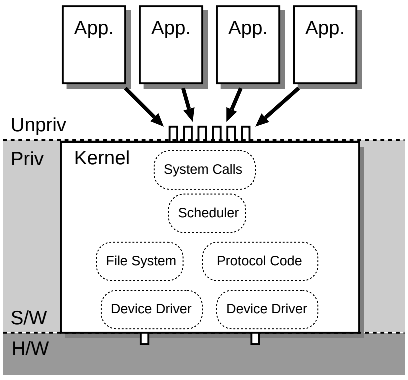
- Hardware checks every memory reference.
- Access out of range ⇒ vector into operating system (just as for an interrupt).
- Only allow *update* of base and limit registers in kernel mode.
- Typically disable memory protection in kernel mode (although a bad idea).
- In reality, more complex protection 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 a malicious or badly-written application from 'hogging' the CPU the whole time.

  ⇒ 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

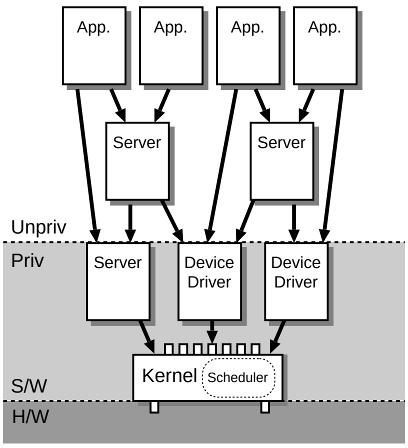


- Applications can't do I/O due to protection

  ⇒ 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 a real (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 (i.e. operate in kernel mode).
- Increases both *modularity* and *extensibility*.
- 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

  $\Rightarrow$ microkernels usually perform less well.

- Microkernel implementation sometimes tricky: need to worry about concurrency and 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 a "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. . .

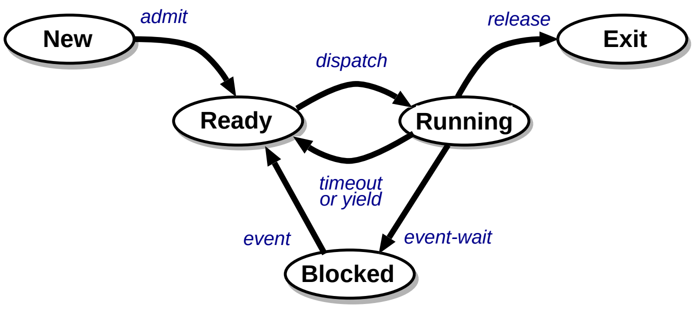
Operating System Functions

- Regardless of structure, OS needs to *securely multiplex resources*:
  1. protect applications from each other, yet
  2. share physical resources between them.
- Also usually want to *abstract* away from grungy harware, i.e. OS provides a *virtual machine*:
  - share CPU (in time) and provide each app 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,
  - divide up storage space by using filing systems, and
  - do all this within the context of a security framework.
- Remainder of this part of the course will look at each of the above areas in turn. . .

Process Concept

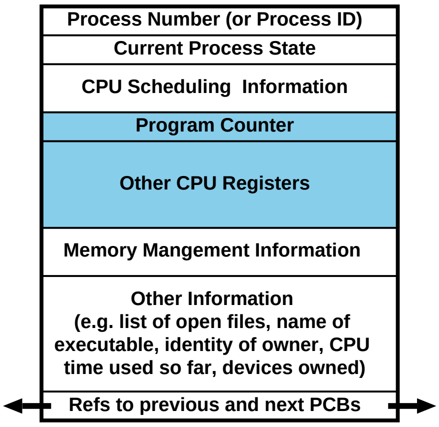
- From a user's point of view, the operating system is there to execute programs:
  - on batch system, refer to *jobs*
  - on interactive system, refer to *processes*
  - (we'll use both terms fairly interchangeably)
- Process $\neq$ Program:
  - a program is *static*, while a process is *dynamic*
  - in fact, a process $\triangleq$ "a program in execution"
- (Note: "program" here is pretty low level, i.e. native machine code or *executable*)
- Process includes:
  1. program counter
  2. stack
  3. data section
- Processes execute on *virtual processors*

Process States



- As a process executes, it changes *state*:
  - New: the process is being created
  - Running: instructions are being executed
  - Ready: the process is waiting for the CPU (and is prepared to run at any time)
  - 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.

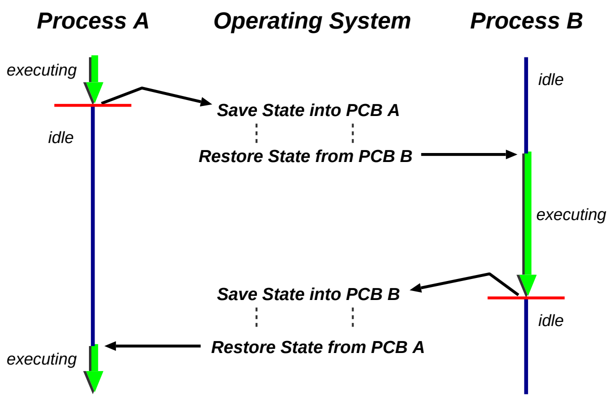
Process Control Block



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

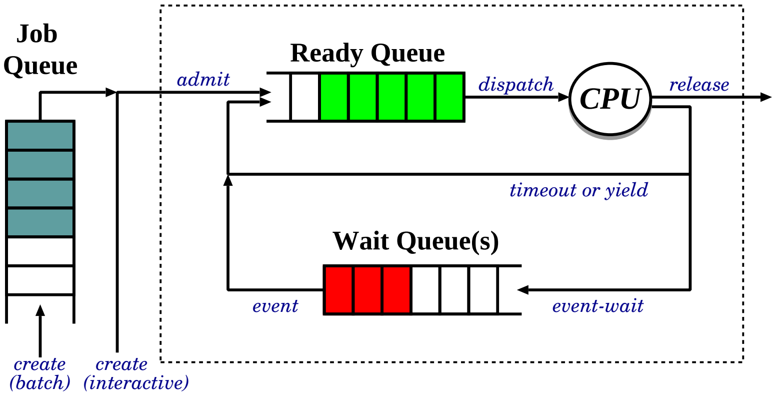
- Unique process identifier
- Process state (*Running*, *Ready*, etc.)
- CPU scheduling & accounting information
- Program counter & CPU registers
- Memory management information
- . . .

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 (with $\mathbf{C}, \mathbf{N}, \mathbf{V}$ and $\mathbf{Z}$ flags), . . .
- To switch between processes, the OS must:
  a) save the context of the currently executing process (if any), and
  b) 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 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 decides which process should be executed next and allocates the CPU to it.

Process Creation

- Nearly all systems are *hierarchical*: parent processes create children processes.
- Resource sharing:
  - parent and children share all resources, or
  - children share subset of parent's resources, or
  - parent and child share no resources.
- Execution:
  - parent and children execute concurrently, or
  - parent waits until children terminate.
- Address space:
  - child is duplicate of parent or
  - child has a program loaded into it.
- e.g. on Unix: `fork()` system call creates a new process
  - all resources shared (i.e. child is a clone).
  - `execve()` system call used to replace process' memory with a new program.
- NT/2K/XP: `CreateProcess()` syscall 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/2K/XP has `ExitProcess()` for self termination and `TerminateProcess()` for killing others.

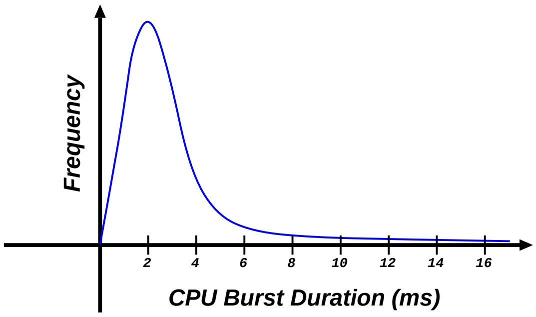
Process Blocking

- In general a process blocks on an *event*, e.g.
  - 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 '{' ?
- (This is a *big* area: lots more detail next year.)
- In this course we'll generally assume that problems of this sort do not arise.

CPU-I/O Burst Cycle



- CPU-I/O Burst Cycle: process execution consists of an on-going *cycle* of CPU execution, I/O wait, CPU execution, . . .
- 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 scheduler selects one of the ready processes and allocates the CPU to 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	Process	Burst Time	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*.

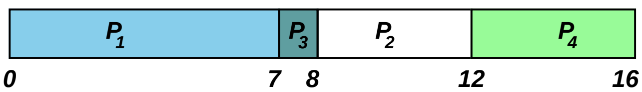
SJF Scheduling

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: $(0 + 6 + 3 + 7)/4 = 4$.

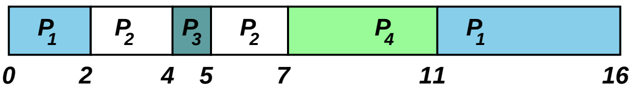
SJF is optimal in the sense that it gives the minimum average waiting time for any given set of processes. . .

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 come up with some way to predict it.
- Can be done by using the length of previous CPU bursts to calculate an exponentially-weighted moving average (EWMA):
  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 EWMA 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 head of the ready queue is allocated the CPU for (up to) one quantum.
- When the time has elapsed, the process is preempted and added to the tail of 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:

Priority	Type	Priority	Type
0	system internal processes	**2**	interactive processes (students)
1	interactive processes (staff)	**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 priority scheduling 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 acceptable. . .

Memory Management

In a multiprogramming system:

- many processes in memory simultaneously, and every process needs memory for:
  - instructions ("code" or "text"),
  - static data (in program), and
  - dynamic data (heap and stack).
- in addition, operating system itself needs memory for instructions and data.

$\Rightarrow$ must share memory between OS and k processes.

The memory magagement subsystem handles:

1. Relocation
2. Allocation
3. Protection
4. Sharing
5. Logical Organisation
6. Physical Organisation

The Address Binding Problem

Consider the following simple program:

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

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

```
str #5, [Rx]        // store 5 into 'x'
ldr R1, [Rx]        // load value of x from memory
add R2, R1, #3      // and add 3 to it
str R2, [Ry]        // and store result in 'y'
```

where the expression '`[ addr ]`' should be read to mean "the contents of the memory at address `addr`".

Then the address binding problem is:

what values do we give Rx and Ry ?

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, 0x2004, but if loaded at 0x5000, then `x` and `y` might be at 0x6000, 0x6004.

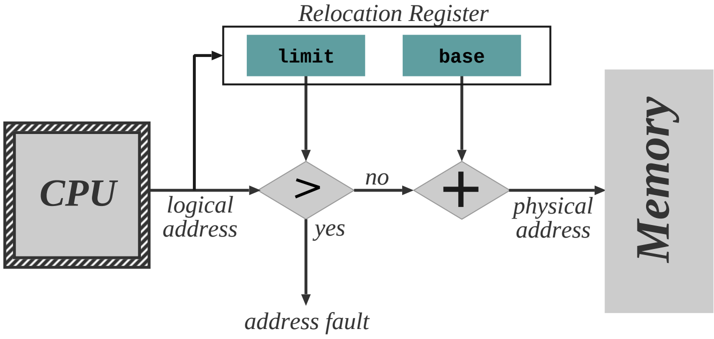
Address Binding and Relocation

To solve the problem, we need to set up some kind of correspondence 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 every relevant instruction in 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 addresses and real addresses.
  - no changes at all required to program itself.
  - most popular and flexible scheme, providing we have the requisite hardware, viz. a memory management unit or MMU.

Logical vs Physical Addresses

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.

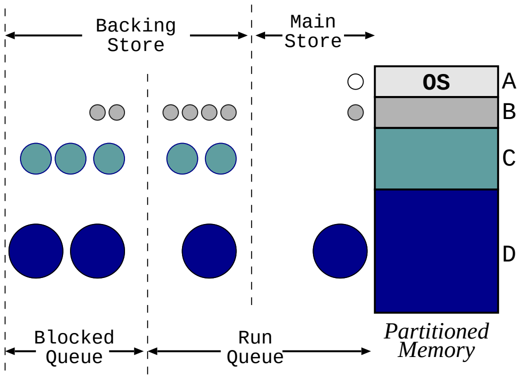
Contiguous Allocation

Given that we want multiple virtual processors, how can we support this in a single address space?

Where do we put processes in memory?

- OS typically must be in low memory due to location of interrupt vectors
- Easiest way is to statically divide memory into multiple fixed size partitions:
  - each partition spans a contiguous range of physical memory
  - bottom partition contains OS, remaining partitions each contain exactly one process.
  - when a process terminates its partition becomes available to new processes.
  - e.g. OS/360 MFT.
- Need to protect OS and user processes from malicious programs:
  - use base and limit registers in MMU
  - update values when a new processes is scheduled
  - NB: solving both relocation and protection problems at the same time!

Static Multiprogramming

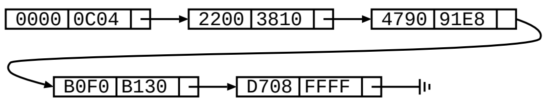


- partition memory when installing OS, and allocate pieces to different job queues.
- associate jobs to a job queue according to size.
- swap job back to disk when:
  - blocked on I/O (assuming I/O is slower than the backing store).
  - time sliced: larger the job, larger the time slice
- run job from another queue while swapping jobs
- e.g. IBM OS/360 MFT, ICL System 4
- **problems**: fragmentation (partition too big), cannot grow (partition too small).

Dynamic Partitioning

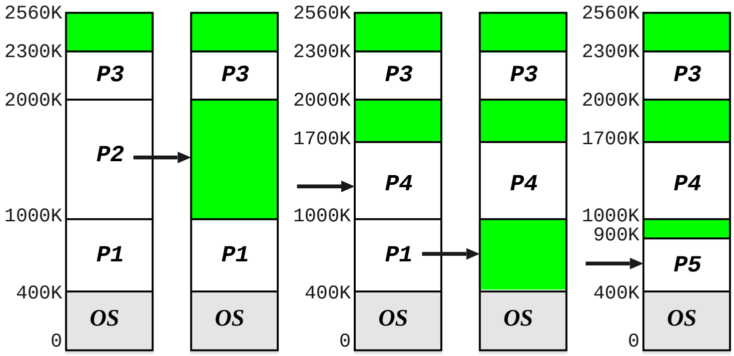
Get more flexibility if allow partition sizes to be dynamically chosen, e.g. OS/360 MVT ("Multiple Variable-sized Tasks"):

- OS keeps track of which areas of memory are available and which are occupied.
- e.g. use one or more *linked lists*:



- When a new process arrives into the system, the OS searches for a hole large enough to fit the process.
- Some algorithms to determine which hole to use for new process:
  - **first fit**: stop searching list as soon as big enough hole is found.
  - **best fit**: search entire list to find "best" fitting hole (i.e. smallest hole which is large enough)
  - **worst fit**: counterintuitively allocate largest hole (again must search entire list).
- When process terminates its memory returns onto the free list, coalescing holes together where appropriate.

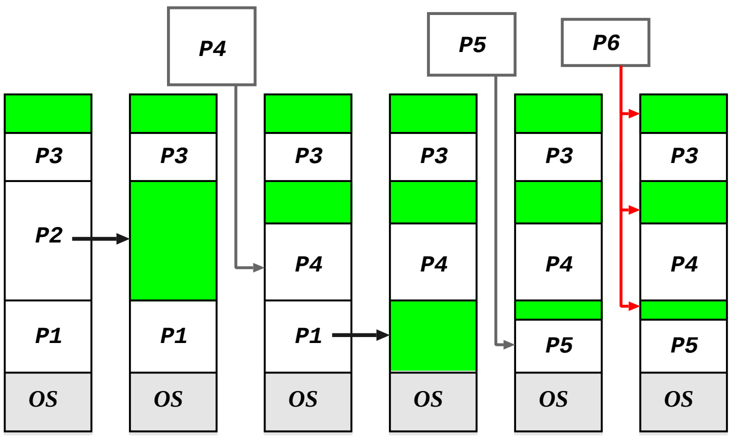
Scheduling Example



- Consider machine with total of 2560K memory, where OS requires 400K.
- The following jobs are in the queue:

Process	Memory Reqd	Total Execution Time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15

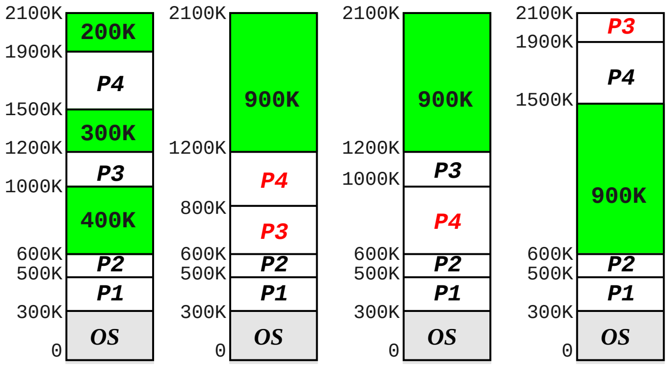
External Fragmentation



- Dynamic partitioning algorithms suffer from external fragmentation: as processes are loaded they leave little fragments which may not be used.
- External fragmentation exists when the total available memory is sufficient for a request, but is unusable because it is split into many holes.
- Can also have problems with tiny holes

Solution: compact holes periodically.

Compaction



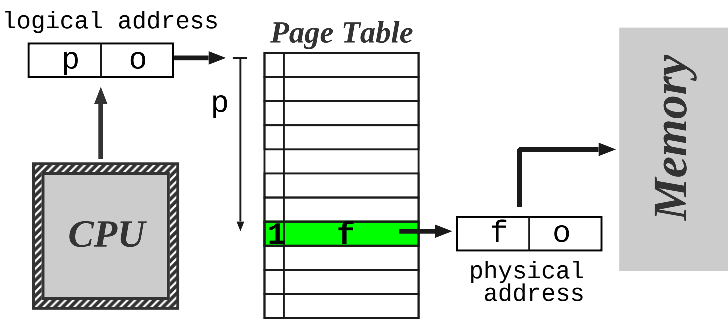
Choosing optimal strategy quite tricky. . .

Note that:

- We require run-time relocation for this to work.
- Can be done more efficiently when process is moved into memory from a swap.
- Some machines used to have hardware support (e.g. CDC Cyber).

Also get fragmentation in *backing store*, but in this case compaction not really a viable option. . .

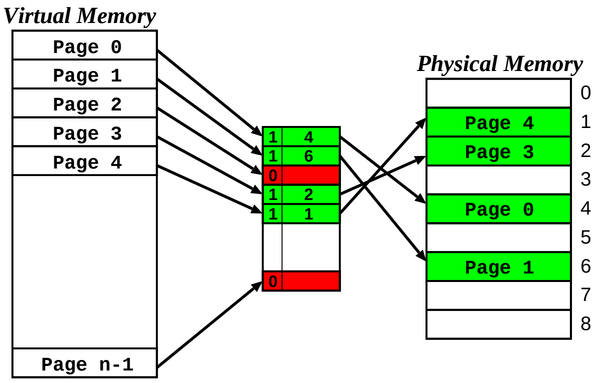
Paged Virtual Memory



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 page sizes are between 512bytes and 8K)
- each address generated by CPU comprises 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 $|\texttt{p}| \gg |\texttt{f}| \Rightarrow$ need valid bit

Paging Pros and Cons



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

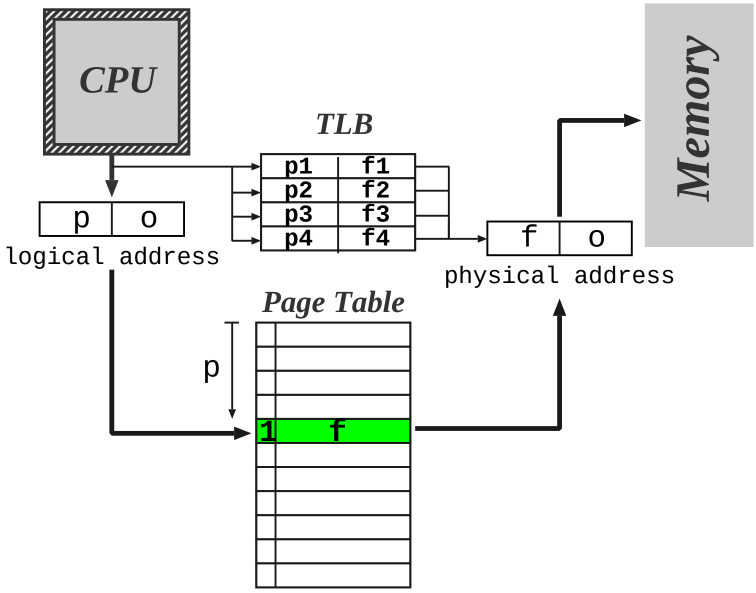
Structure of the Page Table

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. . .

⇒ 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

TLB Issues

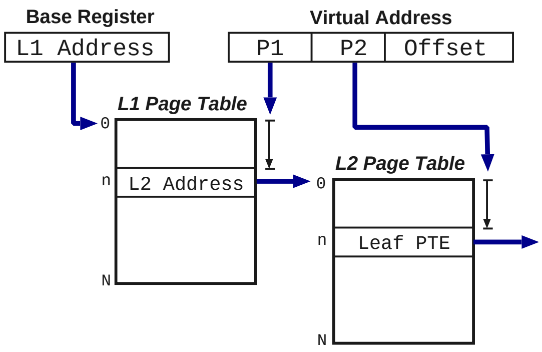
- Updating TLB tricky if it is full: need to discard something.
- Context switch may requires TLB flush so that next process doesn't use wrong page table entries.
  - Today many TLBs support process tags (sometimes called address space numbers) to improve performance.
- Hit ratio is the percentage of time a page entry is found in TLB
- e.g. consider TLB search time of $20ns$, memory access time of $100ns$, and a hit ratio of 80%

⇒ assuming one memory reference required for page table lookup, the *effective* memory access time is $0.8 \times 120 + 0.2 \times 220 = 140ns$.

- Increase hit ratio to 98% gives effective access time of 122ns — only a 13% improvement.

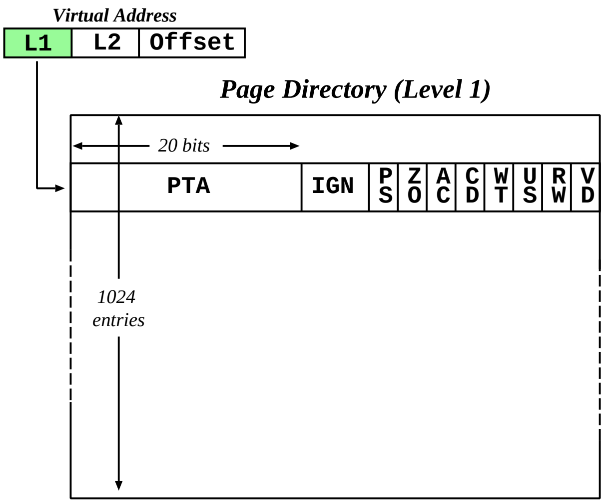
Multilevel Page Tables

- 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



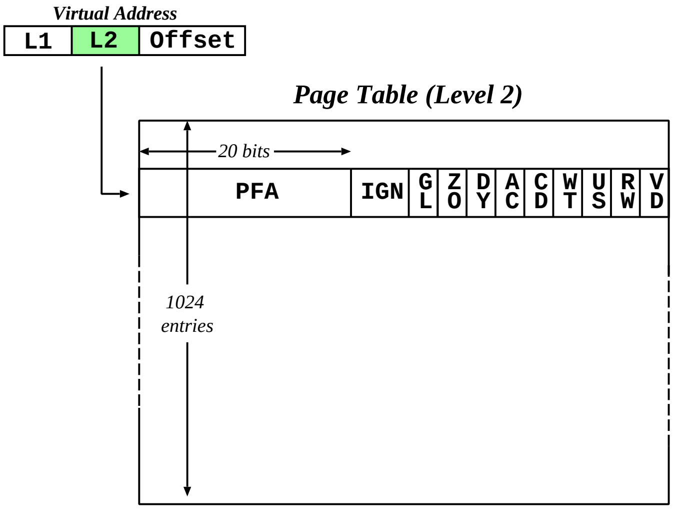
- For 64 bit architectures a two-level paging scheme is not sufficient: need further levels (usually 4, or even 5).
- (even some 32 bit machines have > 2 levels, e.g. x86 PAE mode).

Example: x86



- Page size 4K (or 4Mb).
- First lookup is in the *page directory*: index using most 10 significant bits.
- Address of page directory stored in internal processor register (`cr3`).
- Results (normally) in the address of a *page table*.

Example: x86 (2)



- Use next 10 bits to index into page table.
- Once retrieve page frame address, add in the offset (i.e. the low 12 bits).
- Notice page directory and page tables are exactly one page each themselves.

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 a page may only be accessed when executing in $kernel\ mode$, e.g. a page-table entry may look like:

Frame Number	K	R	W	X	V

- At the same time as address is going through page translation 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 $\Rightarrow$ 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 100MB of shared code on my linux box.

Virtual Memory

- Virtual addressing allows us to introduce the idea of virtual memory:
  - already have valid or invalid pages; introduce a new "non-resident" designation
  - such pages live on a non-volatile backing store, such as a hard-disk.
  - processes access non-resident memory just as if it were 'the real thing'.
- Virtual memory (VM) has a number of benefits:
  - portability: programs work regardless of how much actual memory present
  - convenience: programmer can use e.g. large sparse data structures with impunity
  - efficiency: no need to waste (real) memory on code or data which isn't used.
- VM typically implemented via demand paging:
  - programs (executables) reside on disk
  - to execute a process we load pages in *on demand*; i.e. as and when they are referenced.
- Also get *demand segmentation*, but rare.

Demand Paging Details

When loading a new process for execution:

- we create its address space (e.g. page tables, etc), but mark all PTEs as either "invalid" or "non-resident"; and then
- add its process control block (PCB) to the ready-queue.

Then whenever we receive a page fault:

1. check PTE to determine if "invalid" or not
2. if an invalid reference $\Rightarrow$ kill process;
3. otherwise 'page in' the desired page:
   - find a free frame in memory
   - initiate disk I/O to read in the desired page into the new frame
   - when I/O is finished modify the PTE for this page to show that it is now valid
   - restart the process at the faulting instruction

Scheme described above is *pure* demand paging:

- never brings in a page until required $\Rightarrow$ get lots of page faults and I/O when the process first begins.
- hence many real systems explicitly load some core parts of the process first

Page Replacement

- When paging in from disk, we need a free frame of physical memory to hold the data we're reading in.
- In reality, size of physical memory is limited $\Rightarrow$
  - need to discard unused pages if total demand exceeds physical memory size
  - (alternatively could swap out a whole process to free some frames)
- Modified algorithm: on a page fault we
  1. locate the desired replacement page on disk
  2. to select a free frame for the incoming page:
     (a) if there is a free frame use it
     (b) otherwise select a victim page to free,
     (c) write the victim page back to disk, and
     (d) mark it as invalid in its process page tables
  3. read desired page into freed frame
  4. restart the faulting process
- Can reduce overhead by adding a dirty bit to PTEs (can potentially omit step 2c)
- **Question**: how do we choose our victim page?

Page Replacement Algorithms

- First-In First-Out (FIFO)
  - keep a queue of pages, discard from head
  - performance difficult to predict: have no idea whether page replaced will be used again or not
  - discard is independent of page use frequency
  - in general: pretty bad, although very simple.
- Optimal Algorithm (OPT)
  - replace the page which will not be used again for longest period of time
  - can only be done with an oracle, or in hindsight
  - serves as a good comparison for other algorithms
- Least Recently Used (LRU)
  - LRU replaces the page which has not been used for the longest amount of time
  - (i.e. LRU is OPT with -ve time)
  - assumes past is a good predictor of the future
  - **Question**: how do we determine the LRU ordering?

Implementing LRU

- Could try using counters
  - give each page table entry a time-of-use field and give CPU a logical clock (e.g. an n-bit counter)
  - whenever a page is referenced, its PTE is updated to clock value
  - replace page with smallest time value
  - **problem**: requires a search to find minimum value
  - **problem**: adds a write to memory (PTE) on every memory reference
  - **problem**: clock overflow. . .
- Or a page stack:
  - maintain a $stack$ of pages (a doubly-linked list)
  - update stack on every reference to ensure new (MRU)) page on top
  - discard from bottom of stack
  - **problem**: requires changing 6 pointers per [new] reference
  - possible with h/w support, but slow even then (and extremely slow without it!)
- Neither scheme seems practical on a standard processor $\Rightarrow$ need another way.

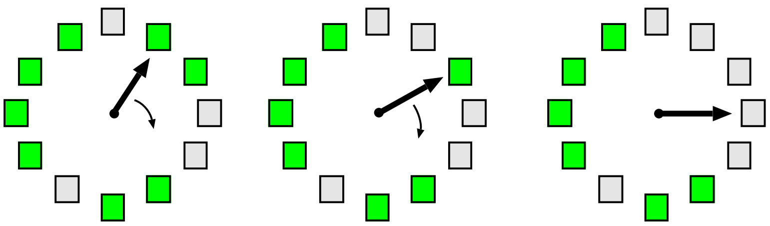
Approximating LRU (1)

- Many systems have a reference bit in the PTE which is set by h/w whenever the page is touched
- This allows not recently used (NRU) replacement:
  - periodically (e.g. 20ms) clear all reference bits
  - when choosing a victim to replace, prefer pages with clear reference bits
  - if we also have a modified bit (or dirty bit) in the PTE, we can extend NRU to use that too:

Ref?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

- Or can extend by maintaining more history, e.g.
  - for each page, the operating system maintains an 8-bit value, initialized to zero
  - periodically (e.g. every 20ms), shift the reference bit onto most-significant bit of the byte, and clear the reference bit
  - select lowest value page (or one of them) to replace

Approximating LRU (2)

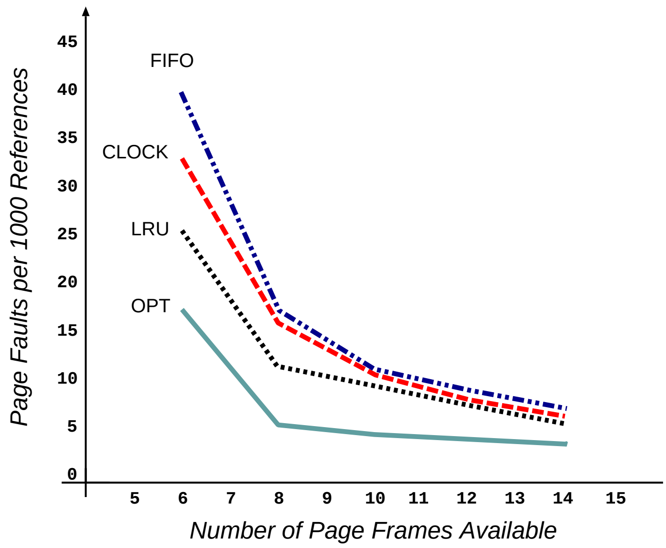


- Popular NRU scheme: second-chance FIFO
  - store pages in queue as per FIFO
  - before discarding head, check its reference bit
  - if reference bit is 0, then discard it, otherwise:
    * reset reference bit, and add page to tail of queue
    * i.e. give it "a second chance"
- Often implemented with circular queue and head pointer: then called clock.
- If no h/w provided reference bit can emulate:
  - to clear "reference bit", mark page no access
  - if referenced ⇒ trap, update PTE, and resume
  - to check if referenced, check permissions
  - can use similar scheme to emulate modified bit

Other Replacement Schemes

- Counting Algorithms: keep a count of the number of references to each page
  - LFU: replace page with smallest count
  - MFU: replace highest count because low count $\Rightarrow$ most recently brought in.
- Page Buffering Algorithms:
  - keep a min. number of victims in a free pool
  - new page read in before writing out victim.
- (Pseudo) MRU:
  - consider access of e.g. large array.
  - page to replace is one application has *just finished with*, i.e. most recently used.
  - e.g. track page faults and look for sequences.
  - discard the k^{th} in victim sequence.
- Application-specific:
  - stop trying to second guess what's going on.
  - provide hook for app. to suggest replacement.
  - must be careful with denial of service. . .

Performance Comparison



Graph plots page-fault rate against number of physical frames for a pseudo-local reference string.

- want to minimise area under curve
- FIFO can exhibit Belady's anomaly (although it doesn't in this case)
- getting frame allocation right has major impact. . .

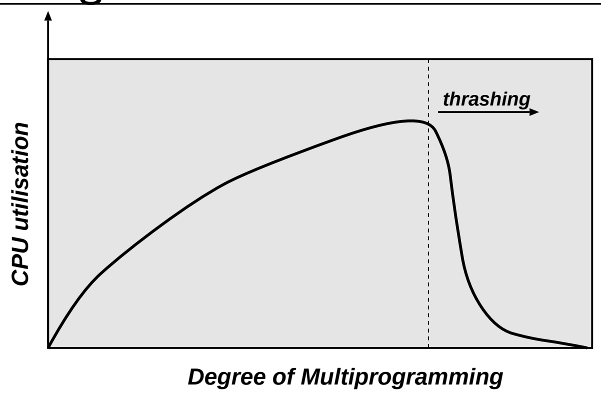
Frame Allocation

- A certain fraction of physical memory is reserved per-process and for core operating system code and data.
- Need an *allocation policy* to determine how to distribute the remaining frames.
- Objectives:
  - Fairness (or proportional fairness)?
    - e.g. divide m frames between n processes as m/n, with any remainder staying in the free pool
    - e.g. divide frames in proportion to size of process (i.e. number of pages used)
  - Minimize system-wide page-fault rate?
    (e.g. allocate all memory to few processes)
  - Maximize level of multiprogramming?
    (e.g. allocate min memory to many processes)
- Most page replacement schemes are *global*: all pages considered for replacement.

⇒ allocation policy implicitly enforced during page-in:

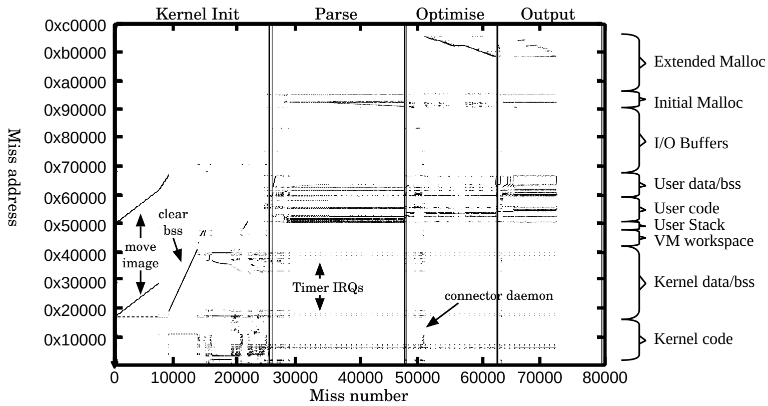
  - allocation succeeds iff policy agrees
  - 'free frames' often in use ⇒ steal them!

The Risk of Thrashing



- As more and more processes enter the system (multi-programming level (MPL) increases), the frames-per-process value can get very small.
- At some point we hit a wall:
  - a process needs more frames, so steals them
  - but the other processes need those pages, so they fault to bring them back in
  - number of runnable processes plunges
- To avoid thrashing we must give processes as many frames as they "need"
- If we can't, we need to reduce the MPL: **better page-replacement won't help!**

Locality of Reference



Locality of reference: in a short time interval, the locations referenced by a process tend to be grouped into a few regions in its address space.

- procedure being executed
- . . . sub-procedures
- . . . data access
- . . . stack variables

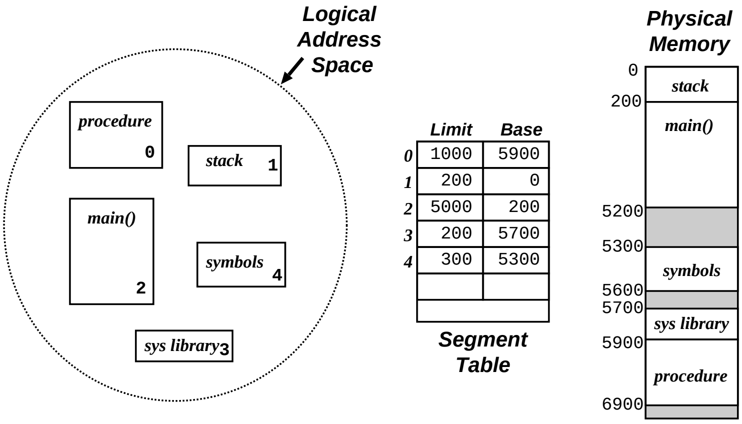
Note: have locality in both space and time.

Avoiding Thrashing

We can use the locality of reference principle to help determine how many frames a process needs:

- define the Working Set (Denning, 1967)
  - set of pages that a process needs to be resident "the same time" to make any (reasonable) progress
  - varies between processes and during execution
  - assume process moves through *phases*:
    * in each phase, get (spatial) locality of reference
    * from time to time get *phase shift*
- OS can try to prevent thrashing by ensuring sufficient pages for current phase:
  - sample page reference bits every e.g. 10ms
  - if a page is "in use", say it's in the working set
  - sum working set sizes to get total demand D
  - if $D > m$ we are in danger of thrashing $\Rightarrow$ suspend a process
- Alternatively use page fault frequency (PFF):
  - monitor per-process page fault rate
  - if too high, allocate more frames to process

Segmentation



- When programming, a user prefers to view memory as a set of "objects" of various sizes, 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.
  - Logical addresses specify segment and offset.
- Contrast with paging where user is unaware of memory structure (one big linear virtual address space, all managed transparently by OS).

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.

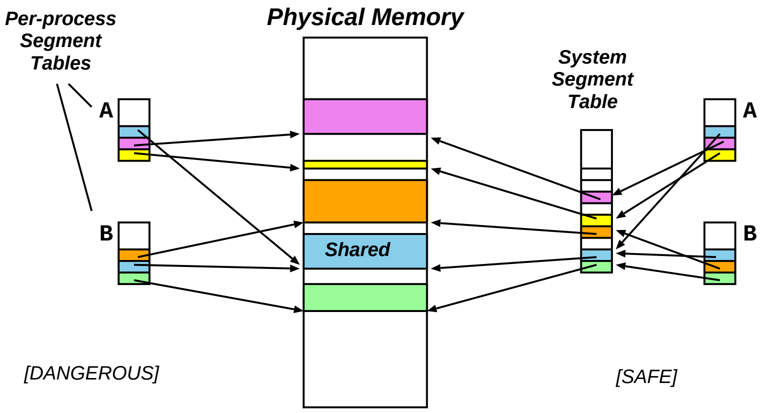
Algorithm:

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 (and programmer's view)
- Protection bits associated with each ST entry checked in usual way
  - e.g. instruction segments (should be non-self modifying!) can be protected against writes
  - e.g. place each array in own seg ⇒ array limits checked by h/w
- Segmentation also facilitates sharing of code/data
  - each process has its own STBR/STLR
  - sharing enabled when two processes have identical entries
  - for data segments can use copy-on-write as per paged case.
- Several subtle caveats exist with segmentation — e.g. jumps within shared code.

Sharing Segments



Sharing segments: dangerously (lhs) and safely (rhs)

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

External Fragmentation Returns. . .

- Long term scheduler must find spots in memory for all segments of a program... but segs are of variable size ⇒ leads to fragmentation.
- Tradeoff between compaction/delay depends on the distribution of segment sizes. . .
  - One extreme: each process gets exactly 1 segment ⇒ reduces to variable sized partitions
  - Another extreme: each byte is a "segment", separately relocated ⇒ quadruples memory use!
  - Fixed size small segments ≡ paging!
- In general with small average segment sizes, external fragmentation is small (consider packing small suitcases into boot of car. . .)

Segmentation versus Paging

	logical view	allocation
Segmentation	✔	✘
Paging	✘	✔

⇒ try combined scheme.

- E.g. paged segments (Multics, OS/2)
  - divide each segment s_i into $k = \lceil l_i/2^n \rceil$ pages, where l_i is the limit (length) of the segment and 2^n is the page size.
  - have seperate page table for every segment.
  - ✘ high hardware cost / complexity.
  - ✘ not very portable.
- E.g. software segments (most modern OSs)
  - consider pages $[m, \ldots, m + l]$ to be a "segment"
  - OS must ensure protection / sharing kept consistent over region.
  - ✘ loss in granularity.
  - ✔ relatively simple / portable.

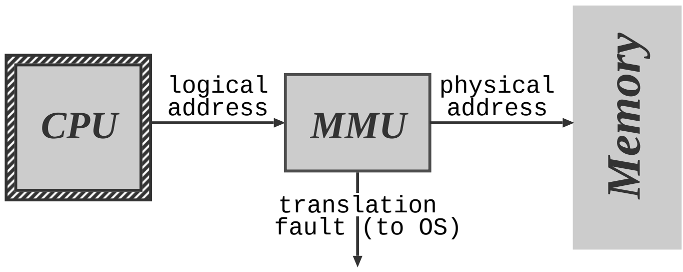
Summary (1 of 2)

Old systems directly accessed [physical] memory, which caused some problems, e.g.

- Contiguous allocation:
  - need large lump of memory for process
  - with time, get [external] fragmentation
  - ⇒ require expensive compaction
- Address binding (i.e. dealing with *absolute* addressing):
  - "`int x; x = 5;`" → "`movl $0x5, ????`"
  - compile time ⇒ must know load address.
  - load time ⇒ work every time.
  - what about swapping?
- Portability:
  - how much memory should we assume a "standard" machine will have?
  - what happens if it has less? or more?

Turns out that we can avoid lots of problems by separating concepts of logical or virtual addresses and physical addresses.

Summary (2 of 2)



Run time mapping from logical to physical addresses performed by special hardware (the MMU). If we make this mapping a per process thing then:

- Each process has own address space.
- Allocation problem solved (or at least split):
  - virtual address allocation easy.
  - allocate physical memory 'behind the scenes'.
- Address binding solved:
  - bind to logical addresses at compile-time.
  - bind to real addresses at load time/run time.

Modern operating systems use paging hardware and fake out segments in software.