Recap 1

• Deadlocks
  – A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause
  – Detecting
  – Preventing
    • Bankers Algorithm
    • Structural Changes
  – Recovering
Recap 2

• Memory Management
  – Hierarchy of storage with faster generally smaller
  – OS and hardware try to cooperate to have the right things in the fast places at the right time

• Address Translation

• Mono-programming
Multiprogramming 1

• The goal is to increase CPU utilization by allowing overlap between CPU and I/O
• If a job waits for I/O, in a monoprogramming environment, the CPU is idle
• If P is the percentage of time spent waiting for I/O, then CPU utilization is 1-P
• Since P is often > 0.5, CPU utilization in a monoprogramming system is bad
Multiprogramming 2

- With a multiprogramming level of N, the CPU utilization is approximately 1-(P^N)
- The idea is that the probability that all of the jobs are waiting for I/O at the same time is low. The probability that all are waiting that the same time is approximated as P^N
- This is a simple model, but it captures the notion that increasing the multiprogramming level increases CPU utilization
- The limiting factor on the multiprogramming level is the amount of memory
Multiprogramming with Fixed Partitions 1

- Memory divided into \( N \) partitions at boot time
- \( N \) can be chosen by the operator
- When a job arrives, it is put into the queue for the smallest partition it will fit into
- Any left over space in a partition is wasted (fragmentation)
- Partition sizes are fixed until reboot
Multiprogramming with Fixed Partitions 2

- We can run into problems if the queue for the small partition is full, and the queue for the large partition is empty.
- One solution is to have a single queue:
  - Possibly very wasteful if we allow small jobs into big partitions.
  - If we don’t, then we run the risk of starving the small jobs.
- This was used in IBM OS/MFT (Multiprogramming with a Fixed number of Tasks).
Relocation

• We need to decide what base address the program was loaded into (usually with a base register)
  – If the program has a `jmp 100`, we need to do a `jmp 100 + base` (think about how the partitions work)
  – The base register lets us do this. Every address in the program will have the base register added to it
  – With base registers, running programs can be moved

• Another solution is to simply modify every address in the program when it is loaded
  – Requires linker support (the linker must make all of the addresses visible in some way
  – OS/MFT did this
Protection

• How do we stop a program from clobbering memory it shouldn’t (another program, or the OS)?
  – One way is to have a key associated with each memory area, and have the OS associate a key with the job
  – Another way is with a limit register. The limit register tells us what the upper limit on addresses we can access is
  – Base and Limit registers often go together
Multiprogramming with Variable Partitions

- Fixed partitions are unattractive because so much memory gets wasted in the partitions.
- Alternate approach allows the size of the partitions to vary while the OS runs (but not while a job runs).
- So, the number, the size, and the location of the partitions is allowed to vary.
- More flexible, but also more complicated.
Multiprogramming with Variable Partitions
Multiprogramming with Variable Partitions

- Each job still has only one partition
- We now have the possibility of fragmentation that is outside any partition (space that can’t be used)
- We want to keep the free spaces as large as possible
- Compaction
- What do we do if there is no partition big enough for a job we want to run?
  - Compact
  - Swap out a job
Multiprogramming with Variable Partitions

• If the jobs are relocatable while running, then we have much more flexibility about moving jobs etc.
• Base registers are one way we get this flexibility
• Dynamic address translations allow us this flexibility
• Number of processes vs. number of holes
  – holes coalesce, processes don’t
  – on average, we expect about twice as many processes as holes
• Which partition to use...
Choosing a Partition 1

• Given that we want to find some space for a process either to be created or to be swapped in, how do we choose?
  – Best Fit - search through all of the holes, and choose the one that is closest (but bigger) **Slow**
  – First Fit - find the first hole large enough and use that **Leaves slivers, Uses big holes**
  – Worst Fit - find the biggest hole **Difficult to place big processes after a while**
  – Quick Fit - keep a list of common sizes (2K, 4K, etc) and use **Finding a neighbor to merge is expensive**
Choosing a Partition 2

• Buddy System
  – Round size up to next power of 2
  – Look in the list of blocks this size (like Quick Fit)
  – Split higher list into buddies if necessary
  – When block is de allocated, coalesce (merge) with buddy
  – Splits and coalesces are done recursively
  – Fragmentation issues

• Next Fit
  – Like first fit only keep track of where the last search stopped, and start from there
  – Uses the big holes up
Partition Choice Implementation

• Linked List
  – Keep a doubly linked list of allocated and free memory (sorted by address)
  – Each item is either a **Hole** or a **Process** and has the starting location and the length
  – When a process exits, the entry is found and can be coalesced with its neighbors

• Bit Map
  – Keep a bit map of memory
  – If each bit represents a big amount of memory, we have fragmentation
  – If each is small, we have lots of bits to store and handle
So Far...

• All of the schemes we have looked at limit a job to the maximum size of real (or physical) memory
  – We talked about overlays and chaining, but these are not terribly practical

• All of the schemes we looked at have memory contiguously allocated. The process is contiguous in physical as well as virtual memory

• The Solution is Virtual Memory...
Virtual Memory 1

• Divide the memory of the program into fixed size pieces called **pages**

• Divide the memory of the machine (the real or physical memory) into fixed size pieces called **page frames**
  – The size of the pages and the size of the page frames is **always** the same

• Keep a mapping of pages to page frames. This mapping, called the **page table**. The entries in this table show us, for a page \( p_1 \), which frame contains \( p_1 \)
Virtual Memory 2

Assume we have 4K pages, and 16K of actual, physical, memory
Page Faults 1

• As we saw in the previous picture, some pages have frames associated with them at any given time, and some don’t
• When a process tries to access a page that does not have a frame associated with it, a page fault is generated
  – The process blocks
  – The OS picks a frame that contains something that won’t be used again for a while and it (possibly) writes the contents of that frame to backing store (that virtual page has been paged out)
  – The OS gets the contents of the virtual page we want from the backing store and puts it in the page frame (the location will be in the page table entry
Page Faults 2

– How the OS chooses the page frame will be a topic of much discussion...
– Much cleverness can be expended on picking the pages to page out
– Much cleverness can be expended on picking which pages should be brought into memory
– Page faults are expensive, and all of this works because programs usually have a working set and locality of reference
– Working set - the pages that are accessed in a given time window tend to be drawn from a fairly small set because the addresses used by the program are not random
Page Tables 1

• Page Table maps virtual pages on to page frames
  – In our previous picture it tells us where the arrows point
  – It also tells us, for every virtual page where we can find the page on disk

• Two Major Issues of Page Tables
  – the page table can be large
  – the mapping must be fast - there might be more mappings than there are instructions executed on the machine
Page Tables 2

• Assume a machine with 32 bit addresses, and 4k pages, the machine has 1 million pages, so the page table has 1 million entries
  – Since every process has its own page table, this is going to get expensive
  – Looking things up in a table with 1 million entries is expensive, too
• Every memory reference turns into an address translation
  – Many instructions access multiple memory locations, so huge numbers of accesses to the page table happen
• The page table has the potential to be a giant bottleneck
Page Table Implementations

- Page Table implementations are governed by: big, cheap, fast pick two
- We will discuss:
  - Where the page table is kept
    - all in main memory
    - in special hardware
  - Multi level page tables
    - avoid many of the flaws of single page tables
    - added complexity
  - Inverted page tables
  - Translation Lookaside Buffer - a cache of page to frame mappings
main(int ac, char * av[]) {
    char str1[] = “hello world\n”;  
    char *x;
    x = (char *)malloc(128);
    sprintf(x,str1);
    printf(x);
}
## 1 Page Table in Memory

- Recall our picture of a processes memory
  - most of the pages in the processes address space are empty (the stack and heap are growing towards them)
- If the page table has entries for all of these pages (which it has to), then it will be big
- Having it in memory will be slow
- A single page table in memory is really only practical if the address spaces are small
  - PDP 11

<table>
<thead>
<tr>
<th>Page Table Index</th>
<th>Offset</th>
</tr>
</thead>
</table>

3/18/2002 Moran OS Lecture 6
Multi Level Page Tables 1

• Add a level of indirection, so we have a page table that rather than pointing to page frames points to other pages

• Assume we have one page table
  – break it into page sized pieces (a single page table entry is fairly small, so we get a reasonable number in a page)
  – each of these page size pieces is a 2nd level page table
  – the first level page table will point to these 2nd level page tables
  – the first level page table will be in memory, the 2nd level tables will be paged in as needed
Multi Level Page Tables 2
Multi Level Page Tables 3

- One way to structure the virtual addresses is as 3 parts
  - pt1 pt2 offset
  - assume there are 1024 entries in every page table
  - then we need 10 bits for pt1 and pt2, and the offset will be 12 bits (on a 32 bit machine)
  - PT1 is the index in page table 1 PT2 is the index in page table 2 Offset is the offset in the page

<table>
<thead>
<tr>
<th>PT1</th>
<th>PT2</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>
Multi Level Paging 4

• We can obviously extend the two level scheme to three or more levels
  – on a 32 bit machine anything beyond three levels is probably overkill

• The VAX used two level paging

• The SPARC uses three level paging
Structure of Page Table Entries

• What does the page table contain? (on some Unix)
  – The physical address of the page (the frame)
  – Protection bits telling whether the process can read/write/execute the page
  – Bits for the following
    • Valid - says the contents of the page are valid
    • Reference - tells whether a process recently accessed the page
    • Modify - tells whether a process recently modified the page
    • Copy on write - do we create a new copy when the page is written
    • Age - how long the page has been in the processes working set
Translation Lookaside Buffers

• A TLB is a way to cache the page to page frame translation - this can avoid looking in the page table
• TLB uses associative memory - associative memory allows us to search by content; associative memory also allows us to search in parallel
• If the page is not found, we will do a normal page fault and replace one of the entries with the new page
• Important issue is the hit ratio
Inverted Page Tables

- Page Table is organized by page frames rather than virtual pages
- We can check to see whether any of the page frames currently hold the page we are looking for
- Typically, this is very inefficient
  - This can be much more efficient if the size of the virtual memory is much much bigger than the size of the physical memory and there is special hardware
- We need a normal page table backing this, so if we miss, we can do a normal page fault lookup
Page Replacement Algorithms 1

• Given a page fault, how do we decide which page we will sacrifice to bring a new one in?
• Programs have a working set and locality of reference
  – If we access an address at time $t_0$, we are more likely to access that same address again in the near future
    • This tells us we should keep things around
  – If we access an address at time $t_0$, we are more likely to access addresses near it in the near future
    • This tells us we do well to bring in more than a single word
  – Normal programs don’t simply access random addresses
Page Replacement Algorithms 2

• When a program starts, we have no history
  – We have nothing to base our working set or locality of reference on
  – As a result, we will take a large number of page faults

• Programs tend to periodically change behavior
  – Function calls often cause us to need to change our working set
  – Returns from function calls have similar behavior
Page Replacement Algorithms 3

• Algorithms we will look at
  – Random PRA
  – Optimal PRA
  – Not Recently Used (NRU) PRA
  – FIFO PRA
  – Second Chance PRA
  – Clock PRA
  – Least Recently Used (LRU) PRA
  – Not Frequently Used (NFU) PRA
Random PRA

- Pick a page to page out at random
  - Performance is lousy
  - No locality of reference
  - No working set
  - Easy to implement
  - Any other page replacement algorithm should work better than this for a normal program
Optimal PRA

• Replace the page whose next reference will be farthest in the future
  – Imagine that we know the exact sequence of pages the process will access, then we choose the page that it will be the longest time before we need it again
  – Has the best possible performance (provably)
  – Impossible to implement because we don’t know, in advance, what pages a process will access
Not Recently Used PRA 1

• Recall that our PTE had a set of bits to tell us whether a page had been referenced or modified
• These get set on every memory access
• An OS process can periodically go through and set them back to zero
• Divide the pages into classes
  – Class 0: not referenced, not modified
  – Class 1: not referenced, modified
  – Class 2: referenced, not modified
  – Class 3: referenced, modified
Not Recently Used PRA 2

• Remove a page at random from the lowest numbered class that isn’t empty
  – This assumes that references are more important than modifications in the working set
• Easy to implement (assuming hardware support)
• Good locality of reference & working set
• Reasonable performance
FIFO PRA

• Pick the page that has been around the longest, and page it out
  – Not a very good algorithm
  – Bad locality of reference
  – Bad working set behavior
  – Easy to implement
2nd Chance PRA

- In FIFO, we would keep a queue of the pages, and we’d remove (page out) the first one in the queue.
- In 2nd Chance, if the first one in the queue has the read bit clear (it hasn’t been read recently) we page it out.
- If it does have the read bit set, we move it to the back of the queue and clear the read bit.
  - we walk down the queue until we find a page with the R bit set.
- Fairly good performance.
- Somewhat expensive to implement.
Clock PRA 1

• 2nd Chance is somewhat inefficient because we are moving things in the list

• Keep the pages in a circular list, we imagine a clock hand sweeping through the pages
  – If the clock hand hits a page with the read access set, we turn the read access off
  – If the clock hand hits a page with the read access off, we page it out

• One modification is to add a 2nd hand, the second hand does the paging out, and the first hand turns the read access off
Clock PRA 2

- Good performance
- Fairly easy to implement
Least Recently Used PRA

• Approximation of the Optimal PRA

• Assumptions
  – pages used in the last few instructions will be used again in the next few instructions
  – pages not used for a long time probably won’t be used again soon

• Pick the page that has gone the longest without being used

• Difficult to implement (without very special hardware)

• Excellent behavior
Not Frequently Used PRA 1

• Since LRU is difficult to implement, NFU is an approximation

• How it works
  – We associate a counter with each page
  – At each clock interrupt, we increment the counter of every page that has the reference bit set (clearing the reference bit in the process)
  – When we have a page fault, we remove the page with the lowest count

• Doesn’t work too well since it remembers everything
Not Frequently Used PRA 2

• A straight-forward modification is to shift the counter (assume 8 bits) to the right and add on the left
• This has the advantage that the counts get “aged”
• Fairly easy to implement
• Good performance
• Excellent working set behavior
Not Frequently Used PRA 3

• NFU has some differences from LRU
  – Granularity is that of clock interrupts, so we don’t really get the page accessed last if two pages were accessed in the same interrupt period
  – We have limited memory (there is a horizon) and all pages accessed before the horizon are treated the same (if we have an eight bit counter, the horizon is eight clock interrupts)