Virtual Memory
Virtual Memory POV

Computer Systems Chapter 9.1 - 9.6

Caveat

All of what I'm teaching is ONE way virtual memory is managed.

I have "simplified" some of the processing to make things clear.

The actual implementation is more complicated.

Swapping Memory

Bad Idea:

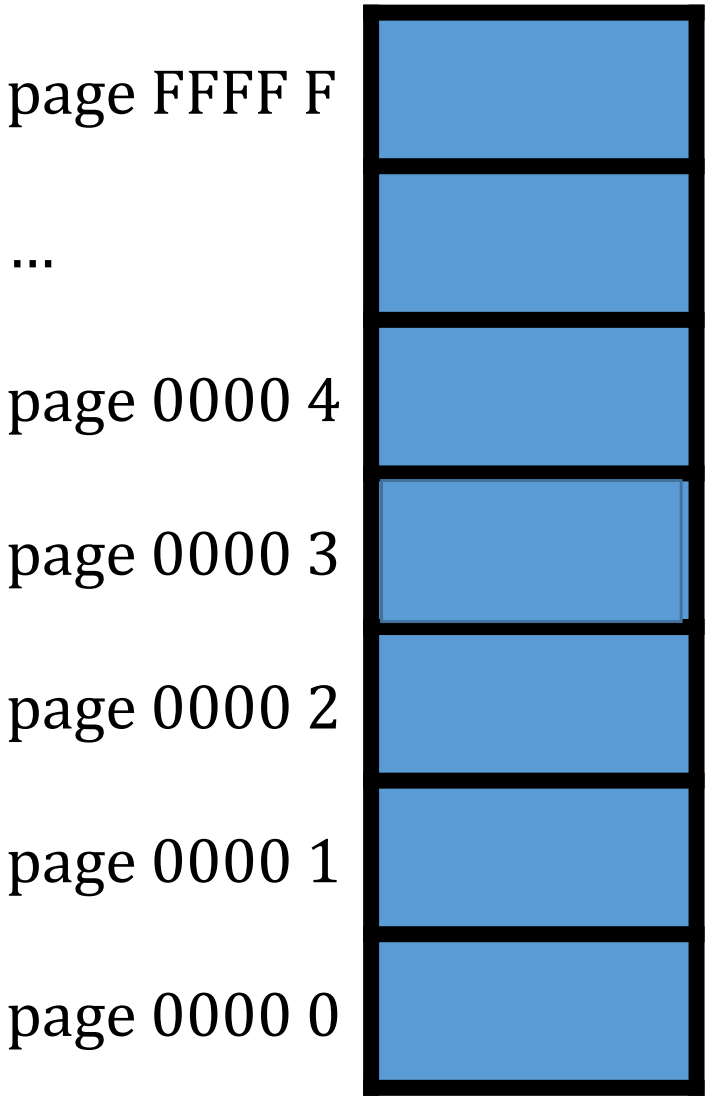
Write Swap Out address space from memory to disk

Read Swap In address space from disk to memory

- A 32 bit address space is 4G
- Writing 4G to disk takes ~1G/sec or 4 seconds
- Times slices are MUCH smaller than 1 second
- You would spend 99.9999% of the time reading/writing memory!

Solution: Stay Tuned

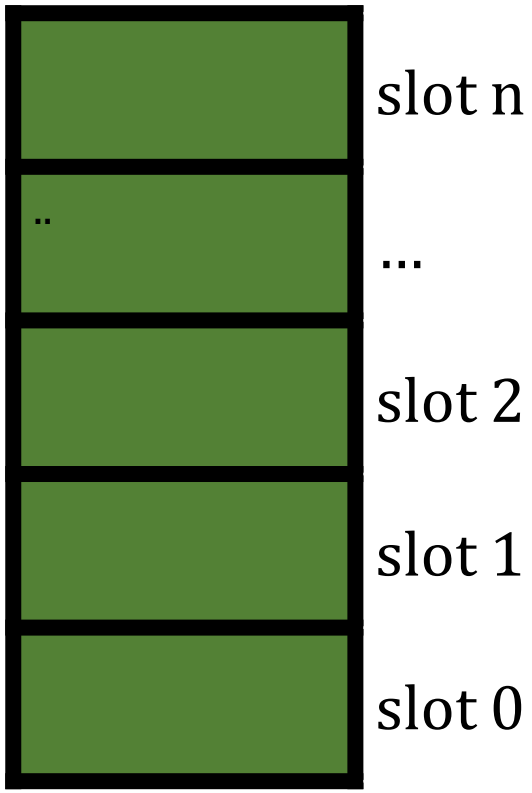
Page Swap In



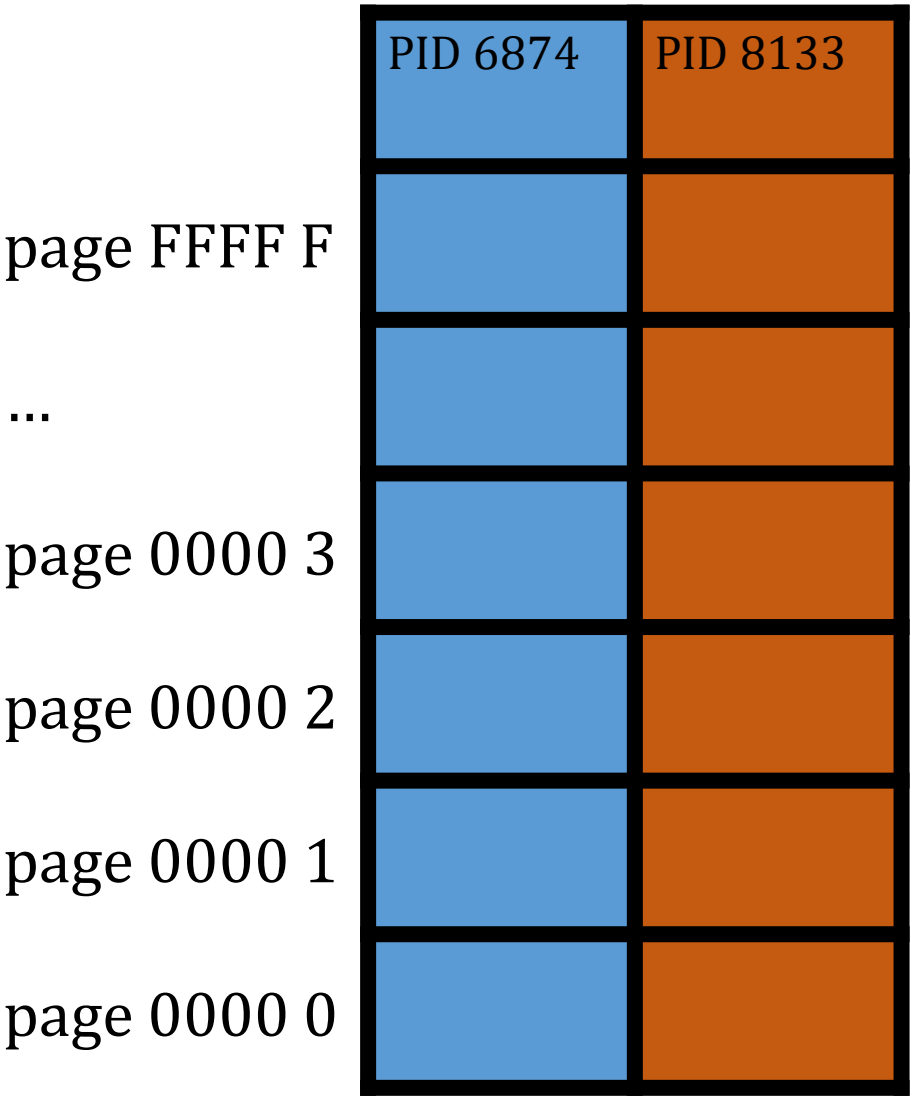
Copy a page from address space into real memory

Page	Slot
0000 3	2

page 0000 3

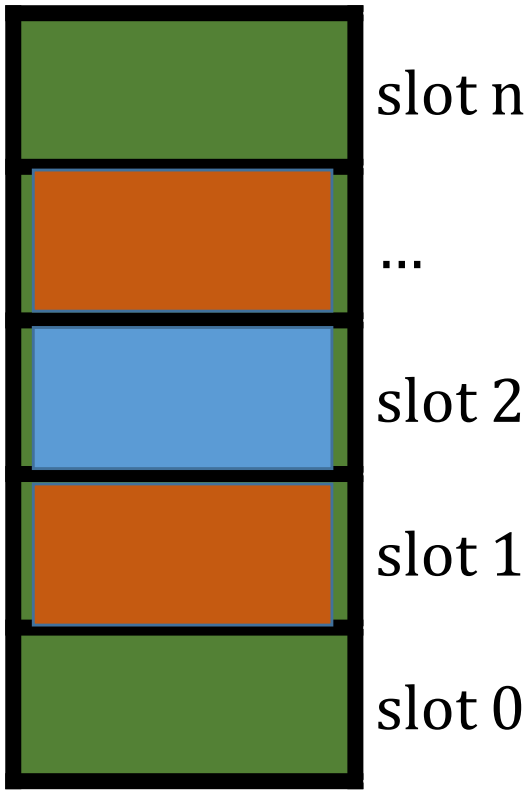


Multi-Process Page Table



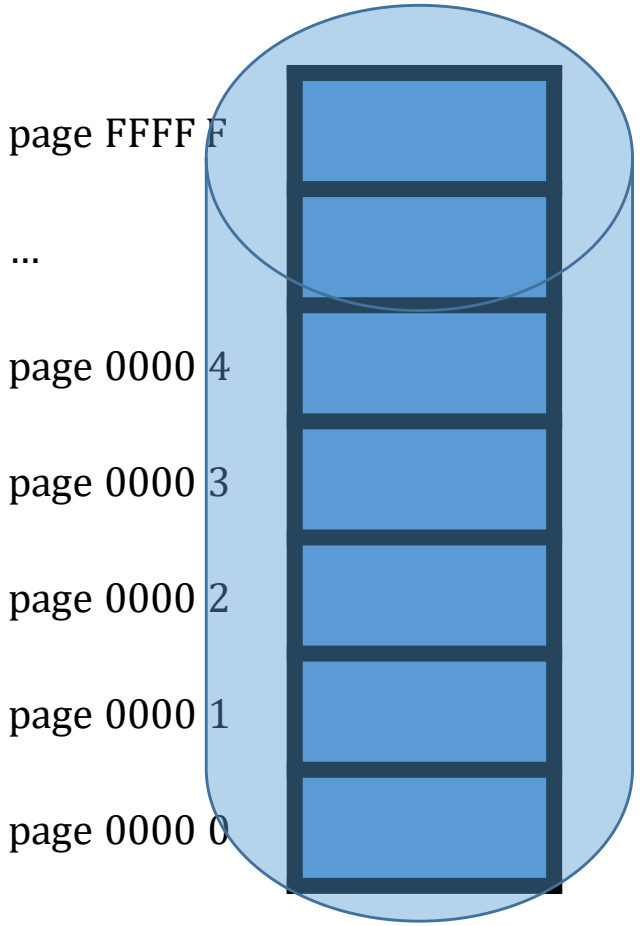
- Keep PID in page table
- When process is active, its pages get high activity and stay in core
- When process is swapped out, its pages get stale and are more likely to get swapped out

PID	Page	Slot
6874	0000 3	2
8133	801C C	1
8133	FFE1 A	5



Abstract Swap Space

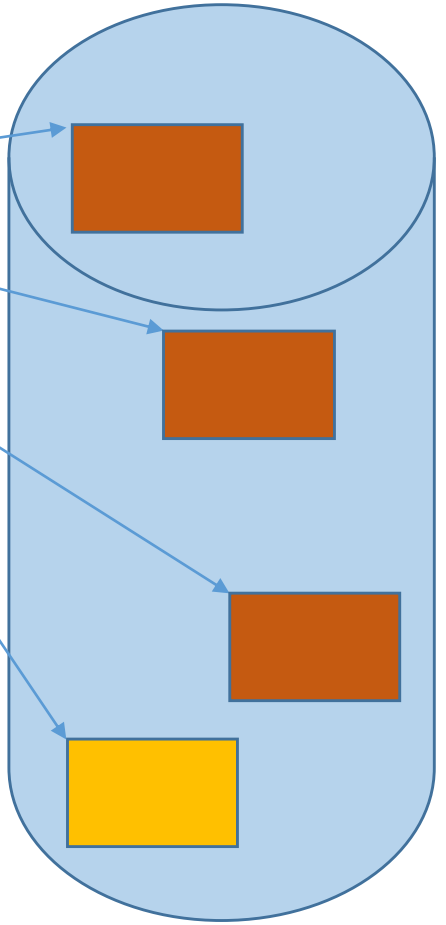
- We think of SWAP space as an entire address space on disk



Leaky Abstraction: Swap Space Details

- Swap space is divided into 4K pages
- And a swap space table….
  - Free – whether this page space is free or currently being used
  - PID – Keeps track of which PID (or PIDs) currently "own" this page
  - Virtual Page – Identifies where this page belongs in the virtual address space
  - Disk Addr – location of page on disk

Free	PID(s)	Virt. Page	Disk Addr
0	8133	801C B	
0	8133	801C C	
0	8133	FFE1 A	
1	----	------	

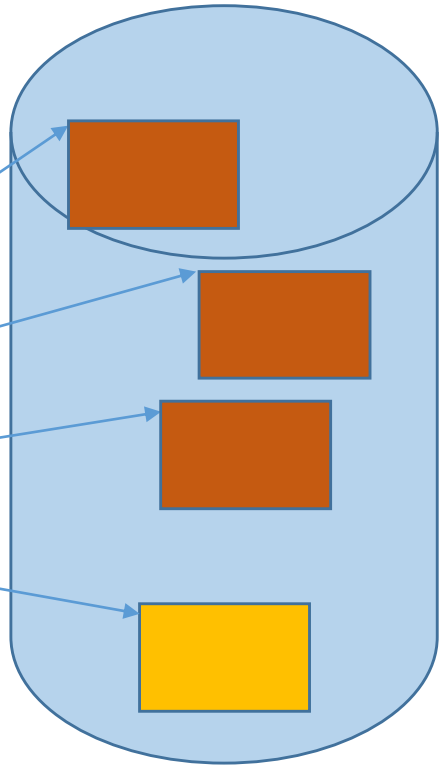


Mutli-Process Memory Request

- PID 8133 requests memory at 0xFFE1A03A
- PID 8133 Page FFE1A is not in the page table
- PAGE FAULT (PID 8133 becomes idle)

PID	Page	Slot	Dirty
6874	0000 3	4	1
8133	801C C	1	0
8133	801C B	2	1

Free	PID(s)	Virt. Page	Disk Addr
0	8133	801C B	
0	8133	801C C	
0	8133	FFE1 A	
1	----	-----	



Before Page Fault

PID 8133 requests memory at 0xFFE1A03A

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	6874	0000 3	0000
0	6874	0000 4	4096
0	8133	801C B	8192
0	8133	801C C	12,288
1	----	---- -	16,384
0	8133	FFE1 A	20,480
			…

Page Table

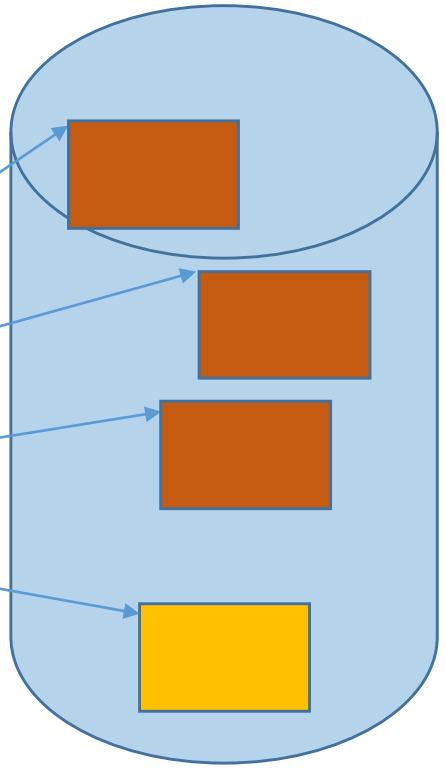
PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	801C B	2	1	->-1
8133	801C D	3	0	->1
6874	0000 3	4	1	->2
6874	0801 3	5	0	H->3
		,,,		

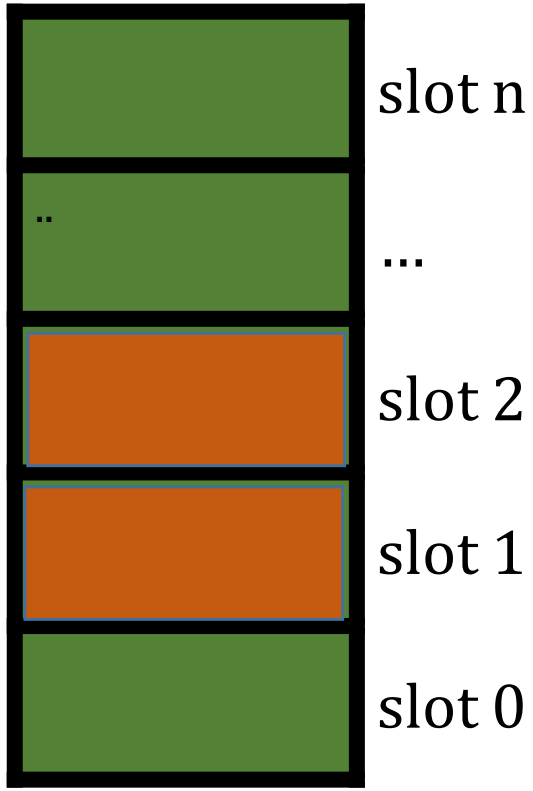
Swap Out

- Pick a victim
- If dirty bit is on, find a free virtual page
- Copy page from real memory to disk

PID	Page	Slot	Dirty
6874	0000 3	4	1
8133	801C C	1	0
8133	801C B	2	1

Free	PID(s)	Virt. Page	Disk Addr
0	8133	801C B	
0	8133	801C C	
0	8133	FFE1 A	
1	----	-----	



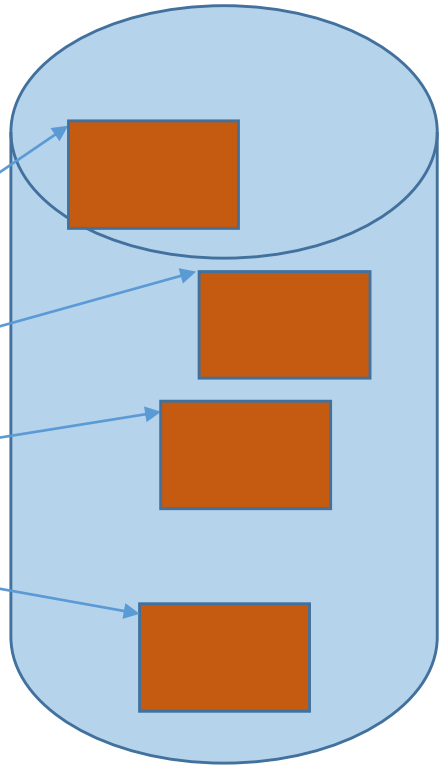


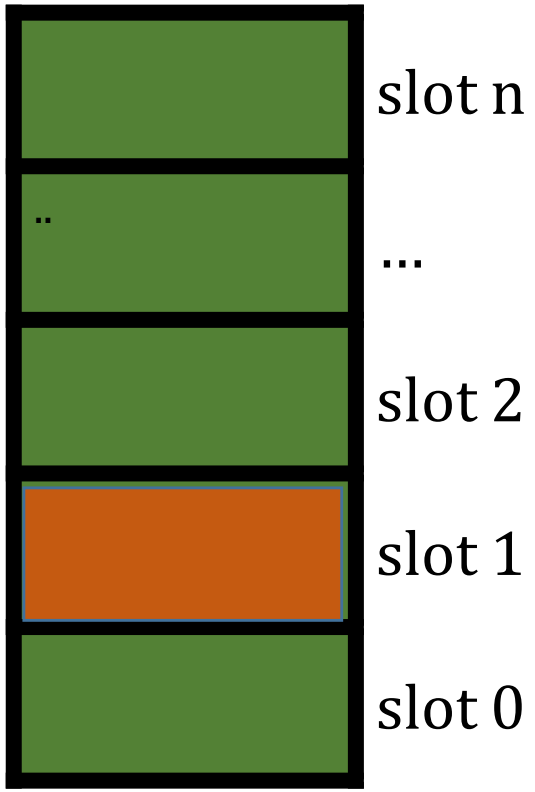
…Swap Out

- Update New Row of Swap Space Table
- Free old Row of Swap Space Table
- Remove Page Table Entry

PID	Page	Slot	Dirty
6874	0000 3	4	1
8133	801C C	1	0
8133	801C B	2	1

Free	PID(s)	Virt. Page	Disk Addr
01	~~8133~~	~~801C B~~	
0	8133	801C C	
0	8133	FFE1 A	
~~0~~ 1	~~8133~~	~~801C B~~	



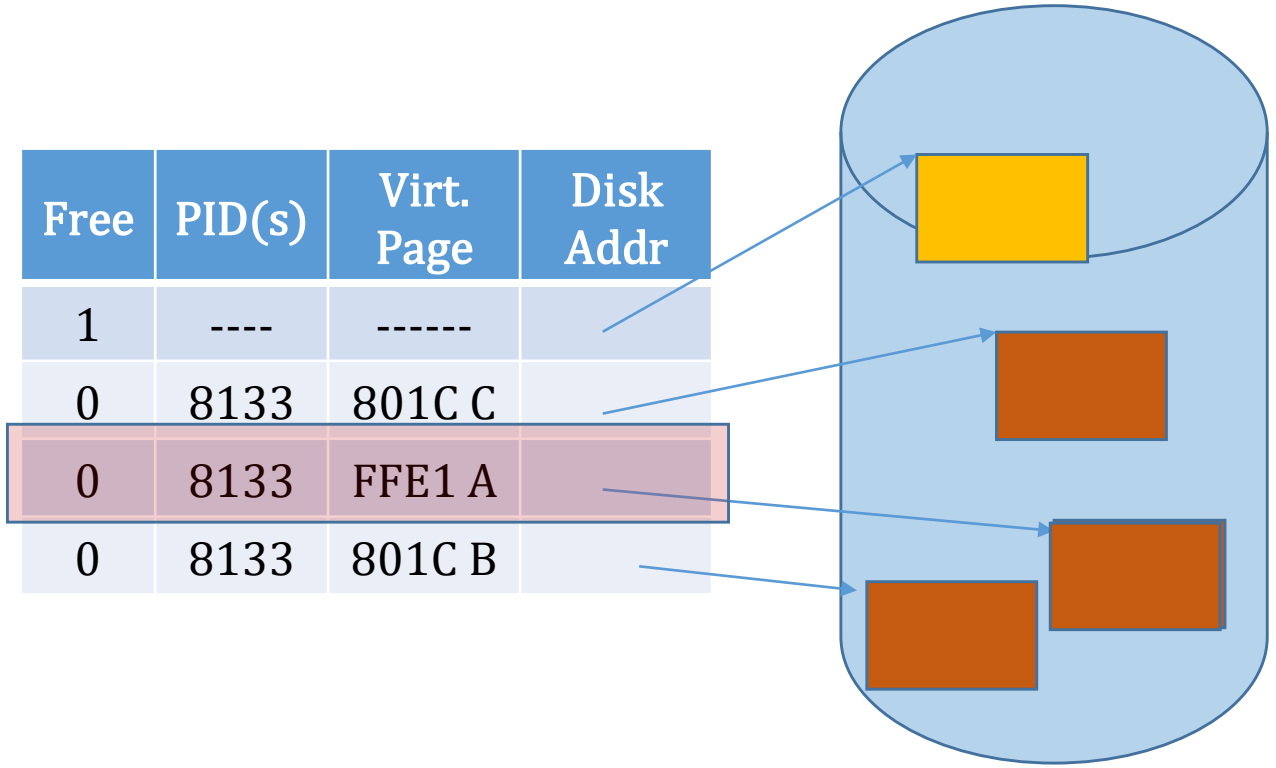


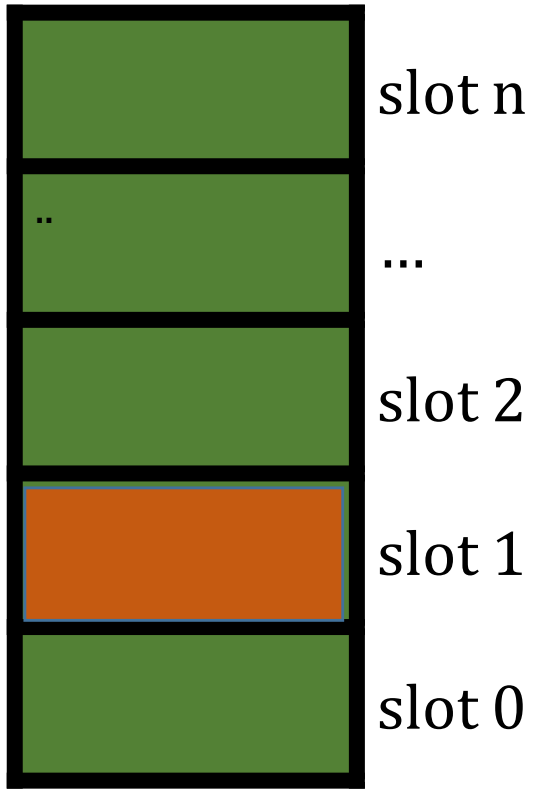
Swap In (to real memory slot)

- Find PID/virtual page in SWAP space table
- Copy page from disk address to real memory
- Update page table

PID	Page	Slot	Dirty
6874	0000 3	4	1
8133	801C C	1	0
8133	FFE1 A	2	0

Free	PID(s)	Virt. Page	Disk Addr
1	----	------	
0	8133	801C C	
0	8133	FFE1 A	
0	8133	801C B	





After Page Fault

PID 8133 requests memory at 0xFFE1A03A

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	6874	0000 3	0000
0	6874	0000 4	4096
1	----	---- -	8192
0	8133	801C C	12,288
0	8133	801C B	16,384
0	8133	FFE1 A	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	FFE1 A	2	0	H->5
8133	801C D	3	0	->1
6874	0000 3	4	1	->-1
6874	0801 3	5	0	->3
		,,,		

Exit Revisited

- When a process exits, dirty bit is turned off for all pages in the page table that belong to that process
- All pages in swap space table associated with this PID are marked as “free”
- No other action is performed
  - The processes pages will eventually get stale and swapped out
  - Dirty bit will never get turned on again because the process is gone.
  - When pages are swapped out, no attempt to write them back to swap space because the dirty bit is off.

Before Exit

PID 6874 about to exit

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	6874	0000 3	0000
0	6874	0000 4	4096
1	----	---- -	8192
0	8133	801C C	12,288
0	8133	801C B	16,384
0	8133	FFE1 A	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	FFE1 A	2	0	H->5
8133	801C D	3	0	->1
6874	0000 3	4	1	->-1
6874	0801 3	5	0	->3
		,,,		

After Exit

PID 6874 exited

Swap Space Table

Free	PID(s)	Page ID	Disk@
1	----	---- -	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133	801C C	12,288
0	8133	801C B	16,384
0	8133	FFE1 A	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	FFE1 A	2	0	H->5
8133	801C D	3	0	->1
6874	0000 3	4	0	->-1
6874	0801 3	5	0	->3
		,,,		

fork revisited

- Add child PID to parent PID in swap space table entries for parent
- When child first accesses memory…
  - Generates page fault… no child process pages in real memory yet.
  - Page swapped in from parent child shared swap space to real memory
  - Page table identifies real page as belonging to child
- When parent process dirty page is swapped out…
  - New swap space is created for that page with only the parent PID
  - Parent PID removed from parent/child shared swap page
- When child process dirty page is swapped out
  - New swap space is created for that page with only the child PID
  - Child PID removed from parent/child shared swap page

Multiple PIDs in Swap Space Table Row

- Most of the time, a page in the swap space table is only associated with a single PID
- When a “fork” occurs, parent and child have exactly the same memory… no need for two copies in swap space… just add child PID to the parents page
- When a page with multiple PIDs is swapped in, it is swapped in by one of those PIDs. In the page table, the page in real memory is owned by a single PID
- If both child and parent access the same page, two pages are in real memory, one for each process

Multiple PIDs in Swap Space Table Row

- When swap-out occurs, memory has changed for the PID that owns the page in real memory
- The page is copied to a NEW swap space page, associated with a single PID… the PID that owned the real page memory
- The PID is removed from the list of PIDs in the old swap space table row for this page
- If, when you removed the PID, there are no PIDs left in the old swap space table row for this page, mark this page as free

Implementation Detail

- For my description of “fork” to work, swap space must match real memory
- Therefore, before adding child PIDs to the parent PID in swap space table, need to copy every page in page table for parent process with Dirty bit on back to swap space.

Before Fork

PID 8133 about to fork

Swap Space Table

Free	PID(s)	Page ID	Disk@
1	----	---- -	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133	801C C	12,288
0	8133	801C B	16,384
0	8133	FFE1 A	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	FFE1 A	2	1	H->5
8133	801C D	3	0	->1
6874	0000 3	4	0	->-1
6874	0801 3	5	0	->3
		,,,		

Before Fork

PID 8133 forking… update swap space

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8133	FFE1 A	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133	801C C	12,288
0	8133	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	FFE1 A	2	0	H->5
8133	801C D	3	0	->1
6874	0000 3	4	0	->-1
6874	0801 3	5	0	->3
		,,,		

Fork – Child PID=8134

PID 8133 forking… add child pid in swap table

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8133,8134	FFE1 A	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->4
8133	801C C	1	0	->0
8133	FFE1 A	2	0	H->5
8133	801C D	3	0	->1
6874	0000 3	4	0	->-1
6874	0801 3	5	0	->3
		,,,		

After Fork / after child page fault

Child PID 8134 requests instruction after fork…

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8133,8134	FFE1 A	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
5934	0801 3	0	0	->-1
8133	801C C	1	0	->0
8133	FFE1 A	2	0	->5
8133	801C D	3	0	->1
8134	801C B	4	0	H->2
6874	0801 3	5	0	->3
		,,,		

After Fork / before parent page swap out

PID 9432 requests instruction at 0x801C A010

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8133,8134	FFE1 A	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
8134	801C C	0	0	H->5
8133	801C C	1	0	->3
8133	FFE1 A	2	1	->-1
8133	801C D	3	0	->4
8134	801C B	4	0	->2
8134	FFE1 A	5	1	->1
		,,,		

After Fork / after parent page swap out

PID 9432 requests instruction at 0x801C A010

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8134	FFE1 A	0000
0	8133	FFE1 A	4096
1	----	---- -	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
8134	801C C	0	0	->5
8133	801C C	1	0	->3
9432	801C A	2	0	H->0
8133	801C D	3	0	->4
8134	801C B	4	0	->-1
8134	FFE1 A	5	1	->1
		,,,		

After Fork / before child page swap out

PID 9432 requests instruction at 0x801C B000

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8134	FFE1 A	0000
0	8133	FFE1 A	4096
1	----	---- -	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
8134	801C C	0	0	->4
8133	801C C	1	0	H->2
9432	801C A	2	0	->0
8133	801C D	3	0	->5
8134	801C B	4	0	->3
8134	FFE1 A	5	1	->-1
		,,,		

After Fork / after child page swap out

PID 9432 requests instruction at 0x801C B000

Swap Space Table

Free	PID(s)	Page ID	Disk@
1	----	---- -	0000
0	8133	FFE1 A	4096
0	8134	FFE1 A	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
8134	801C C	0	0	->4
8133	801C C	1	0	->2
9432	801C A	2	0	->0
8133	801C D	3	0	->-1
8134	801C B	4	0	->3
9432	801C B	5	0	H->1
		,,,		

execve / Load revisited

- Assuming execve is able to find the ELF executable…
- All old entries in the swap space table for this PID marked as free
- Each initialized section of the ELF file is copied to new SWAP page(s) marked with this PID
- Every current entry in Page Table with this PID has PID changed to zero and dirty bit turned off
  - Dirty bit cannot be turned on again… no PID 0
  - Pages will get stale and eventually swapped out
- When first memory fetch is performed, it causes a page fault
  - Loads swap space into real memory and updates page table

Before child execve

Child PID 8134 about to invoke execve

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8133,8134	FFE1 A	0000
1	----	---- -	4096
1	----	---- -	8192
0	8133,8134	801C C	12,288
0	8133,8134	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
8134	801C C	0	0	H->5
8133	801C C	1	0	->3
8133	FFE1 A	2	1	->-1
8133	801C D	3	0	->4
8134	801C B	4	0	->2
8134	FFE1 A	5	1	->1
		,,,		

After child execve

Child PID 8134 after execve

Swap Space Table

Free	PID(s)	Page ID	Disk@
0	8133	FFE1 A	0000
0	8134	801C A	4096
0	8134	801C B	8192
0	8133	801C C	12,288
0	8133	801C B	16,384
1	----	---- -	20,480
			…

Page Table

PID	PageID	Slot	Dirty	PRU
0000	801C C	0	0	->2
8133	801C C	1	0	->3
8133	FFE1 A	2	1	->-1
8133	801C D	3	0	->4
0000	801C B	4	0	H->0
0000	FFE1 A	5	0	->1
		,,,		

Uninitialized Pages

- No data in SWAP required for uninitialized pages of memory
- On first memory access in this page
  - If page not found in swap space table, assume it is uninitialized
  - MMU assigns swapped out page slot to new PID/page
  - Saves disk read for uninitialized memory
- When page is swapped out, if dirty…
  - New swap space page is created for this page (as before)
  - If there is an old page in SWAP for this memory, it is marked as free
  - If there is no old page in SWAP, it must have been uninitialized memory

Shared Memory

- Typically restricted to read-only memory
- Used for standard library functions
- Page Table Entry can be associated with multiple PIDs
  - Multiple processes can share a single page in real memory!
- Saves time
  - Shared pages have high hit rate because they are accessed by multiple processes
  - Shared pages often already in real memory
- Saves real memory
  - if not shared, each process would have its own copy of that page

Shared Memory Problem

- Shared memory must use the same virtual pages for all processes
- Either we must come to global agreement on where “printf” resides (which is REALLY hard) or…
- We must come up with an indirection methodology
  - standard libraries are relocatable – not address specific
  - The first process to call printf (since IPL) “loads” the shared library
  - Somewhere, there is a table that says printf has been loaded at shared virtual address 0x080C 0150 (for instance)
  - When you call printf, you call a non-shared / non-relocatable printf which looks up where printf really is (the first time) and re-directs there

Inter-Process Communication

- Sometime programmers use shared memory for inter-process communication
- Shared page may be written to by multiple processes
- INHERENTLY DANGEROUS!

- Use “ipc” (inter-process communication) utilities instead
  - contains semaphores and message passing protocols

Memory Mapping

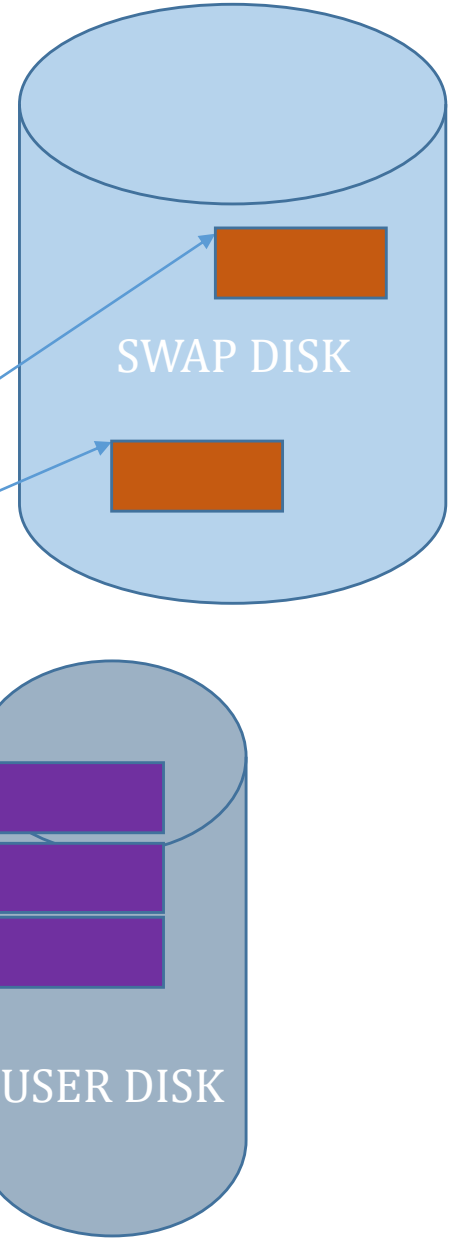
ptr= mmap(NULL,length,PROT_READ,MAP_PRIVATE,fd,0);

- Logically… puts file contents of “fd” into memory and returns a pointer to the start of the file contents
- In fact, divides the file into page sized chunks, and adds those chunks to the swap space table!
- Allows writes as well as reads … dirty bit
- On “munmap”, swaps out all dirty pages

Memory Map

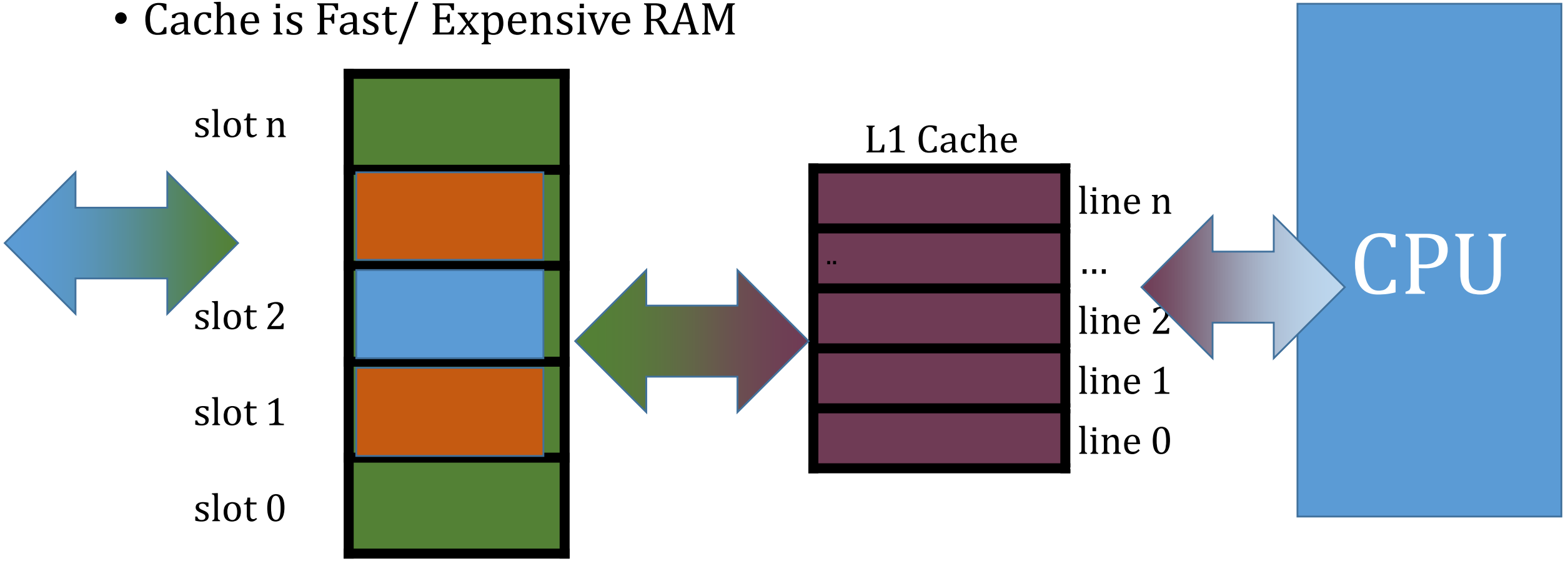
```
int fd=fopen("test.txt",RWONLY);
char *data=mmap(NULL,12288,PROT_WRITE,MAP_PRIVATE,fd,0);
// data=0x0F06 3000
```

Free	PID(s)	Virt. Page	Disk Addr
0	8133	801C C	
0	8133	FFE1 A	
-1	8133	0F06 3	
-1	8133	0F06 4	
-1	8133	0F06 5	



Virtual Memory is not Cache

- Cache uses some of the same concepts as virtual memory
- Cache is Fast/ Expensive RAM



Backup (not presented)

Algorithms for Memory Management

Contents

- Page Table Definition
- Swap Space Table Definition
- realAddr
  - findPageTable
    - getSlot
      - swapOut
        - findSwap
        - newSwap
    - swapIn
  - newRUhead
- exit
- fork
  - flushProcess
- execve

Page Table Definition

PID	Page	Slot	Prot	Dirty	PRU	NRU
0	0	0	-	0	-1	-1
0	0	1	-	0	-1	-1
0	0	…	-	0	-1	-1

- Columns
  - PID – list of process IDs for this page
  - Page – Page ID (first 5 hex chars of addr)
  - Slot – Index of page slot
  - Prot – Protection bits for this page… Read/Write/Execute
  - Dirty – Whether this page differs from Swap space version
  - PRU – Index of previous recently used page
  - NRU – Index of next recently used page
- Initial Values
  - PID=0, Page=0, Slot=0,1,2, up to number of slots in real memory
    Prot=None, Dirty=0, PRU=-1, NRU = -1
- Also: RUhead=-1 RUtail=-1

Swap Space Table Definition

Free	PID	Page	Prot	Disk Addr
1	-	0	-	01000
1	-	0	-	02000
1	-	0	-	…

- Columns
  - Free – 1 if available, ~1 otherwise
  - PID – List of Processes associated with this page in swap
  - Page – Page ID (first 5 hex chars of addr)
  - Prot – Page protection: RWX
  - Disk Addr – Location of page on Swap disk
- Initial values
  - Free – 1, PID – Null, Prot – NULL, Page – 0
  - Disk Addr – all swap page slots available in swap space

Translate Virtual Address to Real Addr

```
addr realAddr(pid,virtAddr,req) {
        pg=(virtAddr & 0xFFFFF000)>>12;
        off=virtAddr & 0x0000FFF;
        ptrow=findPageTable(pid,pg);
        if (req is not in pageTable[ptrow].Prot) {
                // segmentation violation for this pid!
        }
        newNRUhead(ptrow);
        if (req==WRITE) pagetTable[ptrow].dirty=1;
        return (pageTable[ptrow].slot<<12) || off;
}
```

Find Page Table Row

```
row=findPageTable(pid,pg) {
        forEachPageTableRow(row) {
                if (pid is in pageTable[row].PID) {
                        if (pageTable[row].page=pg) return row;
                }
        }
        row=getSlot(pid,pg);
        sRow=findSwap(pid,pg);
        swapIn( pid, pg, row, srow);
        return row;
}
```

Get a new Free Page Table Slot

```
row=getSlot(pid) {
        forEachPageTableRow(row) {
                if (pageTable[row].PID==NULL) {
                        pageTable[row].PID+=pid;
                        return row;
                }
        }
        lru=RUtail;
        RUtail=pageTable[lru].PRU;
        swapOut(lru);
        pageTable[lru].PID+=pid;
        return lru;
}
```

swap a page out of real memory

```
void swapOut(row) {
        if (pageTable[row].dirty) {
                pg=pageTable[row].page;
                foreach pid (pageTable[row].PID) {
                        srow=findSwap(pg,pid); nsrow=newSwap();
                        swapTable[nsrow].PID+=pid; swapTable[nsrow].page=pg;
                        swapTable[nsrow].free=0;
                        //copy from pageTable[row].slot to swapTable[nsrow].diskAddr
                        if (srow!=-1) swapTable[srow].free=1;
                }
                pageTable[row].dirty=0;
        }
        pageTable[row].PID=NULL;
}
```

Find entry in the swap space table

```
row=findSwap(page,pid) {
        foreachSwapSpaceTableRow(row) {
                if (swapTable[row].PID==pid) {
                        if (swapTable[row].page==page)
                                return row;
                }
        }
        return -1;
}
```

Find Free swap space entry

```
row=newSwap() {
        foreachSwapSpaceTableRow(row) {
                if (swapTable[row].free==1)
                        return row;
        }
        // Terminate processes… out of swap space!
}
```

Swap a page into Real memory

```
swapIn( pid, pg, row, srow ) {
        pageTable[row].prot=READ/WRITE; // default
        if (srow != -1) {
                // copy data from swapTable[srow].diskAddr
                //                      to pageTable[row].slot
                pageTable[row].prot=swapTable[srow].prot
        }
        pageTable[row].PID=pid;
        pageTable[row].pg=pg;
        pageTable[row].dirty=0;
}
```

Insert this page in PRU/NRU list

```
newRUhead(row) {
        if (RUhead==-1) { RUhead=row; RUtail=row; return; }
        if (row==RUtail) { RUtail=pageTable[row].PRU; }
        prev=pageTable[row].PRU
        if (prev!=-1) {
                pageTable[prev].NRU=pageTable[row].NRU
        }
        pageTable[RUhead].PRU=row;
        Ruhead=row;
}
```

Process Exit

```
processExit(pid) {
        forEachPageTableRow(row) {
                if (pid is in pageTable[row].PID) {
                        // remove pid from pageTable[row].PID
                }
        }
        foreachSwapSpaceTableRow(row) {
                if (pid is in swapTable[row].PID {
                        // remove pid from swapTable[row].PID
                        if (swapTable[row].PID == NULL)
                                swapTable[row].free=1;
                        }
                }
        }
}
```

Fork a process

```
pid=fork() {
        ppid=getPid(); cpid=newPid();
        flushProcess(ppid); // Make sure swap is up to date
        foreachSwapSpaceTableRow(row) {
                if (ppid is in swapTable[row].PID)
                        swapTable[row].PID+=cpid;
        }
        // Start second instruction stream for cpid
        // Second instruction stream sets cpid to zero!
        return cpid;
}
```

Make sure all Dirty bits are off

```
flushProcess(pid) {
        forEachPageTableRow(row) {
                if (pid is in pageTable[row].PID) {
                        if (pageTable[row].dirty) {
                                oldPids=pageTable[row].PID;
                                swapOut(row);
                                pageTable[row].PID=oldPids;
                        }
                }
        }
```

execve

```
rc=execve(file,args,env) {
        // open file and check ELF header return error if there is a problem
        pid=getPid(); tpid=temporary pid
        foreachELFsection(sec) {
                addr=sectionStart(sec)
                foreachPageinSection(epg) {
                        srow=newSwap();
                        pg=addr & 0xFFFFF000 <<12;
                        // Copy page from ELF file to swapTable[srow].diskAddr
                        swapTable[srow].PID=tpid; swapTable[srow].page=pg;
                        swapTable[srow].Prot = elf protection info; swapTable[srow].free=0;
                        addr+=4K;
                }
        }
        exitProcess(pid);
        // change tpid to pid in process table PID fields
        // jump to start address
```