L11: Beyond Physical Memory: Mechanisms

(Based on Ch. 21)

Slides revised from
Matthew Tancreti
Iowa State University

Beyond Physical Memory

Users have expectation they can run many programs concurrently

Memory requirements of all processes combined can easily exceed total physical memory

Observation

- At any time, many idle processes
- Only a few pages account for most memory accesses

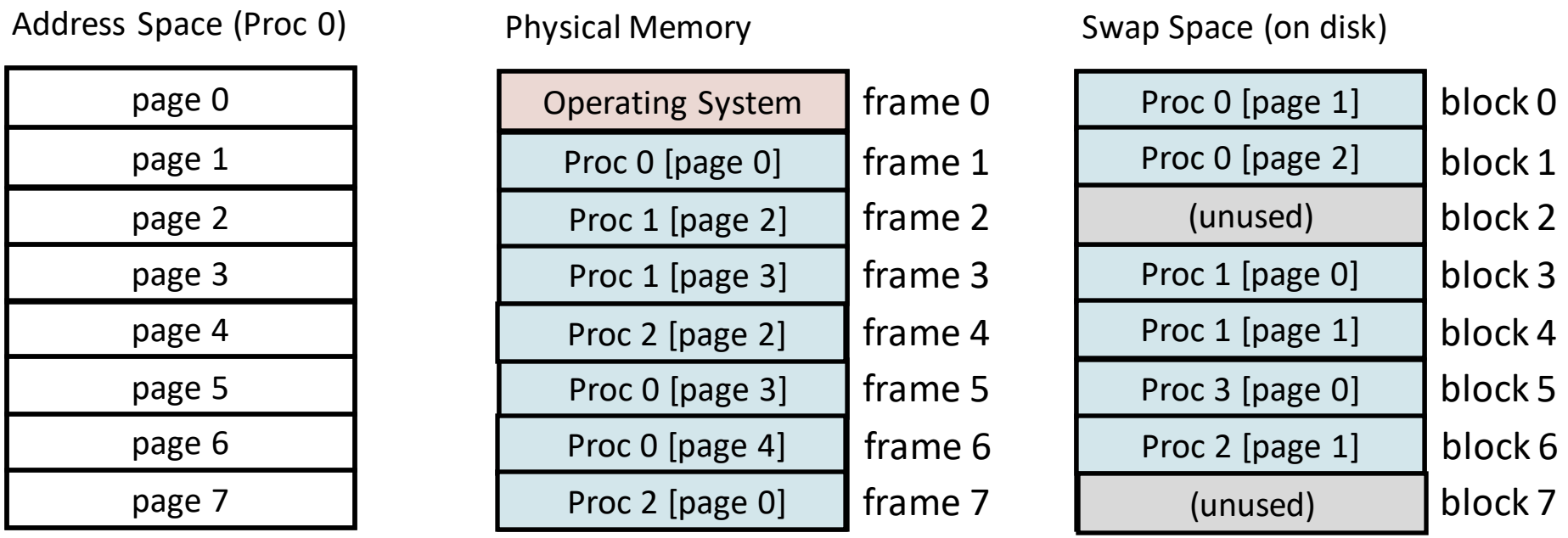
Idea: rarely used pages can be stored on disk

How can OS take advantage of large slow disk space to give illusion of large address spaces?

Swap Space

When physical memory is full, less used pages moved to swap space

Swap space divided into **blocks** that can hold one page



Example: Swap Space in Pyrite

```
pyrite-n4:>swapon –s
Filename                Type        Size            Used        Priority
/dev/dm-1               partition   4194300         0           -2
/dev/zram0              partition   8112124         68352       100

pyrite-n4:>top
Tasks: 289 total,   1 running, 288 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.2 sy,  0.1 ni, 99.4 id,  0.0 wa,  0.1 hi,  0.1 si,  0.0 st
MiB Mem :   7922.5 total,   1773.9 free,   1329.4 used,   4819.1 buff/cache
MiB Swap:  12018.0 total,  11951.2 free,     66.8 used.   6293.8 avail Mem
```

Present Bit

Page table has **present bit** to indicate if page is in physical memory (1) or in swap (0)

Address Space (Proc 0)

page 0
page 1
page 2
page 3
page 4
page 5
page 6
page 7

Page Table (Proc 0)

VPN	PFN	valid	present
0	1	1	1
1	-	1	0
2	-	1	0
3	5	1	1
4	6	1	1
5	-	0	-
6	-	0	-
7	-	0	-

Physical Memory

Operating System	frame 0
Proc 0 [page 0]	frame 1
Proc 1 [page 2]	frame 2
Proc 1 [page 3]	frame 3
Proc 2 [page 2]	frame 4
Proc 0 [page 3]	frame 5
Proc 0 [page 4]	frame 6
Proc 2 [page 0]	frame 7

Swap Space (on disk)

Proc 0 [page 1]	block 0
Proc 0 [page 2]	block 1
(unused)	block 2
Proc 1 [page 0]	block 3
Proc 1 [page 1]	block 4
Proc 3 [page 0]	block 5
Proc 2 [page 1]	block 6
(unused)	block 7

What to do when page not present?

When program tries to access memory that is on a page currently in swap a **page fault** occurs

The page fault is just a trap, it is delt with by OS code called the **page-fault handler**

The page-fault handler needs to move the page back to main memory

It may require making room by swapping some other page into swap space

Memory Access Protocol

On each memory access

- MMU finds page table entry (in TLB cache or in page table)
- If present == 1 then use frame number to complete memory access
- Else (**page fault**) cause trap
- OS handles trap in **page-fault handler**
- OS swaps page into main memory
- OS returns from trap and MMU completes memory access

Example

On each memory access

- MMU finds page table entry (in TLB cache or in page table)
- If present == 1 then use frame number to complete memory access
- Else (**page fault**) cause trap
- OS handles trap in **page-fault handler**
- OS swaps page into main memory
- OS returns from trap and MMU completes memory access

Example: assume process reads from page 2

1. MMU searches for page 2

Page Table (Proc 0)

VPN	PFN	valid	present
0	1	1	1
1	-	1	0
2	-	1	0
3	5	1	1
4	6	1	1
5	-	0	-
6	-	0	-
7	-	0	-

Physical Memory

Operating System	frame 0
Proc 0 [page 0]	frame 1
Proc 1 [page 2]	frame 2
Proc 1 [page 3]	frame 3
Proc 2 [page 2]	frame 4
Proc 0 [page 3]	frame 5
Proc 0 [page 4]	
Proc 2 [page 0]	frame 7

Swap Space (on disk)

Proc 0 [page 1]	block 0
Proc 0 [page 2]	block 1
(unused)	block 2
Proc 1 [page 0]	block 3
Proc 1 [page 1]	block 4
Proc 3 [page 0]	block 5
Proc 2 [page 1]	block 6
(unused)	block 7

Example

On each memory access

- MMU finds page table entry (in TLB cache or in page table)
- If present == 1 then use frame number to complete memory access
- Else (**page fault**) cause trap
- OS handles trap in **page-fault handler**
- OS swaps page into main memory
- OS returns from trap and MMU completes memory access

Example: assume process reads from page 2

Page Table (Proc 0)

VPN	PFN	valid	present
0	1	1	1
1	-	1	0
2	-	1	0
3	5	1	1
4	6	1	1
5	-	0	-
6	-	0	-
7	-	0	-

1. MMU searches for page 2

2. Present bit == 0, cause trap (page fault)

3. OS see memory is full, finds page to evict

4. OS swap the pages

Physical Memory

Operating System	frame 0
Proc 0 [page 0]	frame 1
Proc 1 [page 2]	frame 2
Proc 1 [page 3]	frame 3
Proc 2 [page 2]	frame 4
Proc 0 [page 3]	frame 5
Proc 0 [page 4]	
Proc 2 [page 0]	frame 7

Swap Space (on disk)

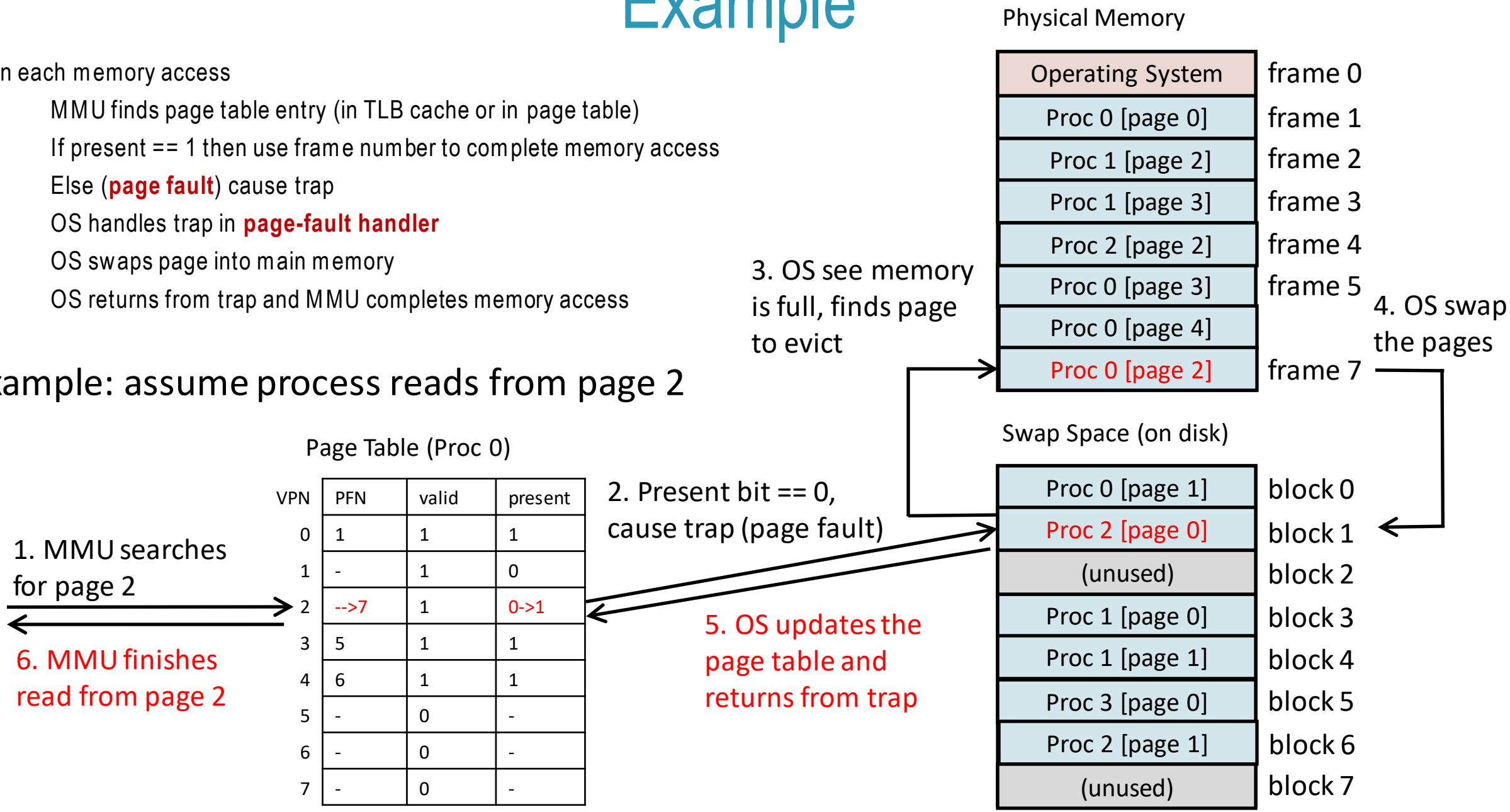
Proc 0 [page 1]	block 0
Proc 0 [page 2]	block 1
(unused)	block 2
Proc 1 [page 0]	block 3
Proc 1 [page 1]	block 4
Proc 3 [page 0]	block 5
Proc 2 [page 1]	block 6
(unused)	block 7

Example

On each memory access

- MMU finds page table entry (in TLB cache or in page table)
- If present == 1 then use frame number to complete memory access
- Else (**page fault**) cause trap
- OS handles trap in **page-fault handler**
- OS swaps page into main memory
- OS returns from trap and MMU completes memory access

Example: assume process reads from page 2



Why Not Put Everything on Disk?

Main memory is 100x slower than CPU

SSD drive is 100x (or more) slower than main memory

Swap enables large easy to use address spaces, but cost of page fault is enormous, really want to avoid page faults

Important considerations

- How to decide which page to replace?
- What happens to performance when memory is full?

Replacement Policy

Need to decide on page to evict (**page out**) so a page can be brought back to memory (**page in**)

There are many **page-replacement policies**, just like we saw with TLB cache

- First In First Out (FIFO)
- Least Recently Used (LRU)
- Random
- …

Proactive Free Space Management

Assume main main memory is full

What happens when user start new application?

Several page faults! Really slow!

OS proactively moves pages to swap to always keep a small amount of memory free for responding to sudden activity

Two thresholds **high watermark** and **low watermark**

- When free memory drops below low watermark OS start background task to start pushing pages to swap space
- Task stop when free memory passes high watermark

Examples in xv6-riscv: get arguments of system call

syscall.c:

```
// Retrieve an argument as a pointer.
// Doesn't check for legality, since
// copyin/copyout will do that.
void argaddr(int n, uint64 *ip)
{
  *ip = argraw(n);
}

//note: the obtain address is VA (virtual address)
```

```
static uint64 argraw(int n)
//get the n-th arg of current process' systemcall
{
  struct proc *p = myproc();
  switch (n) {
  case 0: return p->trapframe->a0;
  case 1: return p->trapframe->a1;
  case 2: return p->trapframe->a2;
  case 3: return p->trapframe->a3;
  case 4: return p->trapframe->a4;
  case 5: return p->trapframe->a5;
  }
  return -1;
}
```

Examples in xv6-riscv

Riscv.h:

```
#define PGSIZE 4096 // bytes per page
#define PGSHIFT 12  // bits of offset within a page

//get the VA of the starting of the page containing VA a
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

//get the VA of the starting of the next page of the page containing VA a
#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
```

Examples in xv6-riscv: copy from kernel to user space

vm.c:

```
Int copyout(pagetable_t pagetable, uint64
dstva, char *src, uint64 len)
{
  uint64 n, va0, pa0;
  while(len > 0){
    va0 = PGROUNDDOWN(dstva);
    //find PA of VA va0
    pa0 = walkaddr(pagetable, va0);
    if(pa0 == 0)
      return -1;
    n = PGSIZE - (dstva - va0);
     if(n > len)
      n = len;
    memmove((void *)(pa0 + (dstva - va0)),
src, n);
    len -= n;
    src += n;
    dstva = va0 + PGSIZE;
  }
  return 0;
}
```

Examples in xv6-riscv: address translation

vm.c:

```
pte_t *
walk(pagetable_t pagetable, uint64 va, ...)
{
  if(va >= MAXVA)
    panic("walk");
  for(int level = 2; level > 0; level--) {
    pte_t *pte = &pagetable[PX(level, va)];
    if(*pte & PTE_V) {
      pagetable = (pagetable_t)PTE2PA(*pte);
    } else {
      ...
    }
  }
  return &pagetable[PX(0, va)];
}
```

Note:

```
// Return the address of the PTE in page table pagetable
// that corresponds to virtual address va.  If alloc!=0,
// create any required page-table pages.
//
// The risc-v Sv39 scheme has three levels of page-table
// pages. A page-table page contains 512 64-bit PTEs.
// A 64-bit virtual address is split into five fields:
//   39..63 -- must be zero.
//   30..38 -- 9 bits of level-2 index.
//   21..29 -- 9 bits of level-1 index.
//   12..20 -- 9 bits of level-0 index.
//    0..11 -- 12 bits of byte offset within the page.
```

Examples in xv6-riscv: address translation

vm.c:

```
Uint64 walkaddr(pagetable_t pagetable, uint64 va)
{
  pte_t *pte;
  uint64 pa;
  if(va >= MAXVA)
    return 0;
  pte = walk(pagetable, va, 0);
  if(pte == 0) return 0;
  if((*pte & PTE_V) == 0) return 0; //valid?
  if((*pte & PTE_U) == 0) return 0; //user allowed to access?
  pa = PTE2PA(*pte);
  return pa;
}
```

Note:

// Look up a virtual address, return the physical address,

// or 0 if not mapped.

// Can only be used to look up user pages.

// va must be the VA of the starting of a page