Virtual Memory: Page Replacement Algorithms

Azza Abouzied

From Last Class: Swapping

Swapping

Kernel uses some space on disk as an extension of RAM. It is transparent to the programmer.

- Expands address space available to a process;
- Expands space available to load processes.

Generally not a good thing! We want to keep swapping to a minimum. Why?

What to swap out?

Swapping applies only to:

1. Pages belonging to an anonymous memory region (for instance, a User Mode stack, heap) of a process

2. Modified pages belonging to a private (file) memory mapping of a process

3. Pages belonging to a shared memory region used for inter-process communication.

How to distribute pages in the swap areas?

- A swap area is broken into slots, the PPN tells us which slot.

- Store pages in contiguous slots to reduce disk seeks.

- Multiple swap areas across several storage devices.

How to select the page to be swapped out?

- Steal pages from the process having the largest number of pages in RAM.
- Typically swap out least recently used pages

When to perform page swap-out?

- When free memory falls below a pre-defined threshold.
- Background swapping kernel thread.

Page Replacement Algorithms

Context

Page replacement occurs at a per-process level[1]. The kernel decides, which processes can have their pages swapped out/replaced.

How to pick the victim process?

- The Linux-way? *Survivors on a desert island*
- The Working-set way?

Now for each process, your goal is to swap-out/free the page that is least likely to be accessed immediately.

[1] In linux, page tables are per process. Global page tables also exist where a PTE also contains a process tag.

The Optimal Algorithm

Replace the page that we won't use for a while!

Assume an Oracle that sees into the future

Reference String:
1 2 3 4 1 2 5 1 2 3 4 5

If we have 4 page frames in memory, how many faults?

The Optimal Algorithm

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes

The Optimal Algorithm

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes
1 2 3 4	1	1 2 3 4	
1 2 3 4	2	1 2 3 4	

The Optimal Algorithm

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes
1 2 3 4	1	1 2 3 4	
1 2 3 4	2	1 2 3 4	
1 2 3 4	5	1 2 5 4	Yes

The Optimal Algorithm

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before				Request	After				Fault?
1	1	1	1	1	1	1	1	1	Yes
1	1	1	1	2	1	2	1	1	Yes
1	2	1	1	3	1	2	3	1	Yes
1	2	3	1	4	1	2	3	4	Yes
1	2	3	4	1	1	2	3	4	
1	2	3	4	2	1	2	3	4	
1	2	3	4	5	1	2	5	4	Yes
1	2	5	4	1	1	2	5	4	
1	2	5	4	2	1	2	5	4	

The Optimal Algorithm

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes
1 2 3 4	1	1 2 3 4	
1 2 3 4	2	1 2 3 4	
1 2 3 4	5	1 2 5 4	Yes
1 2 5 4	1	1 2 5 4	
1 2 5 4	2	1 2 5 4	
1 2 5 4	3	1 3 5 4	Yes

The Optimal Algorithm

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before				Request	After				Fault?
1	1	1	1	1	1	1	1	1	Yes
1	1	1	1	2	1	2	1	1	Yes
1	2	1	1	3	1	2	3	1	Yes
1	2	3	1	4	1	2	3	4	Yes
1	2	3	4	1	1	2	3	4	
1	2	3	4	2	1	2	3	4	
1	2	3	4	5	1	2	5	4	Yes
1	2	5	4	1	1	2	5	4	
1	2	5	4	2	1	2	5	4	
1	2	5	4	3	1	3	5	4	Yes
1	3	5	4	4	1	3	5	4	
1	3	5	4	5	1	3	5	4	

The FIFO Algorithm

Queue up pages, throw out oldest page.

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes
1 2 3 4	1	1 2 3 4	
1 2 3 4	2	1 2 3 4	

The FIFO Algorithm

Queue up pages, throw out oldest page.

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes
1 2 3 4	1	1 2 3 4	
1 2 3 4	2	1 2 3 4	
1 2 3 4	5	5 2 3 4	Yes
5 2 3 4	1	5 1 3 4	Yes
5 1 3 4	2	5 1 2 4	Yes
5 1 2 4	3	5 1 2 3	Yes

The FIFO Algorithm

Queue up pages, throw out oldest page.

Reference String: 1 2 3 4 1 2 5 1 2 3 4 5

Before	Request	After	Fault?
1 1 1 1	1	1 1 1 1	Yes
1 1 1 1	2	1 2 1 1	Yes
1 2 1 1	3	1 2 3 1	Yes
1 2 3 1	4	1 2 3 4	Yes
1 2 3 4	1	1 2 3 4	
1 2 3 4	2	1 2 3 4	
1 2 3 4	5	5 2 3 4	Yes
5 2 3 4	1	5 1 3 4	Yes
5 1 3 4	2	5 1 2 4	Yes
5 1 2 4	3	5 1 2 3	Yes
5 1 2 3	4	4 1 2 3	Yes
4 1 2 3	5	4 5 2 3	Yes

Belady's Anomaly

What if we only have 3 page frames in memory?

Before	Request	After	Fault?
1 1 1	1	1 1 1	Yes
1 1 1	2	1 2 1	Yes
1 2 1	3	1 2 3	Yes
1 2 3	4	4 2 3	Yes
4 2 3	1	4 1 3	Yes
4 1 3	2	4 1 2	Yes
4 1 2	5	5 1 2	Yes
5 1 2	1	5 1 2	
5 1 2	2	5 1 2	
5 1 2	3	5 3 2	Yes
5 3 2	4	5 3 4	Yes
5 3 4	5	5 3 4	

Clock Algorithm

A **second-chance** algorithm over FIFO:

1. It checks if the **Accessed** bit is set.
2. This bet is set by the hardware on page access.
3. If set, it clears it and moves on to the next frame.
4. If clear, it replaces the page.

If all pages were recently accessed, it degenerates into FIFO.

Hard to assess its benefit with toy reference strings. Relies on the frequency with which a swapping thread is called, the emergency of finding a free frame and how often pages do get accessed.

The clock name comes from using a hand that points to the last examined page in a FIFO circular queue.

The Dirty Bit

Enhanced Clock/Second Chance:

1. Basic strategy prefers flushing unreferenced pages to referenced ones

2. Use dirty bit to flush clean, unreferenced pages over dirty referenced ones.

Why is this an improvement?

Least Recently Used

Replace the page that has not been used the longest. Need a timestamp for each page

Pros:

1. Best approximation of Optimal
2. Based on the best predictor for tomorrow's weather is today's weather.
3. Most heavily used now are most heavily used in the near future

Cons:

1. Requires hardware support to supply time accessed on every page access!
2. Storing a timestamp in every page entry is 4 bytes of wasted space.

Least Recently Used Approximation

Instead of a full 4-byte timestamp, use a smaller counter.

- Counter incremented after every instruction
- After every memory reference we up the counter in the PTE of the page just accessed.
- Gives us least recently used estimate

A 1-bit counter is just the accessed flag: was the page referenced since the last request for a free page frame?

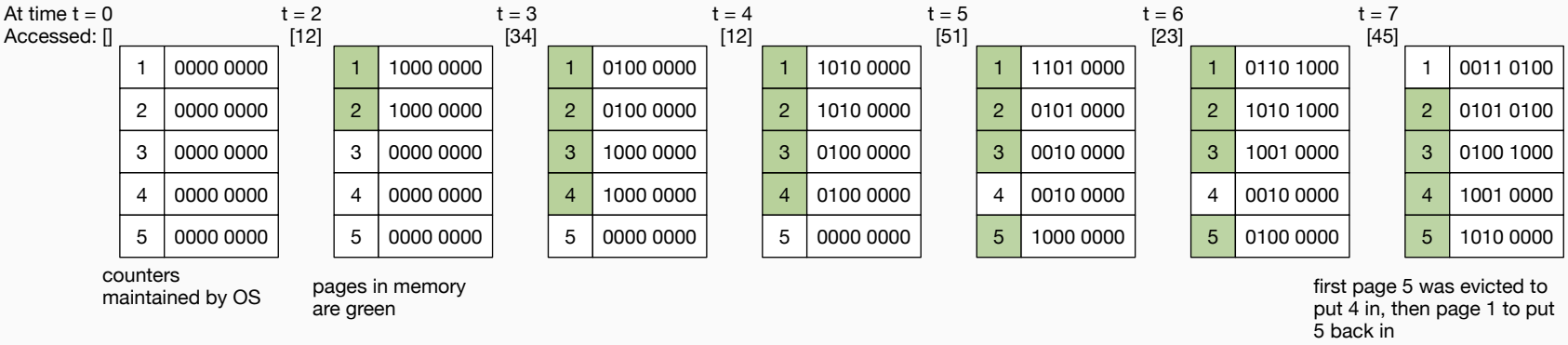
n bits gives us 2^n options.

Aging

In **not-frequently-used**, we increment a counter if a page was referenced at least once in the last time space: e.g. 20ms.

No way to differentiate between a page that was heavily used in the last 20msec vs. one that was only touched once. Both are equal. One can be unfairly evicted!

Aging is a variant of the NFU algorithm.



What pages to keep in? The Working Set

What pages to keep in?

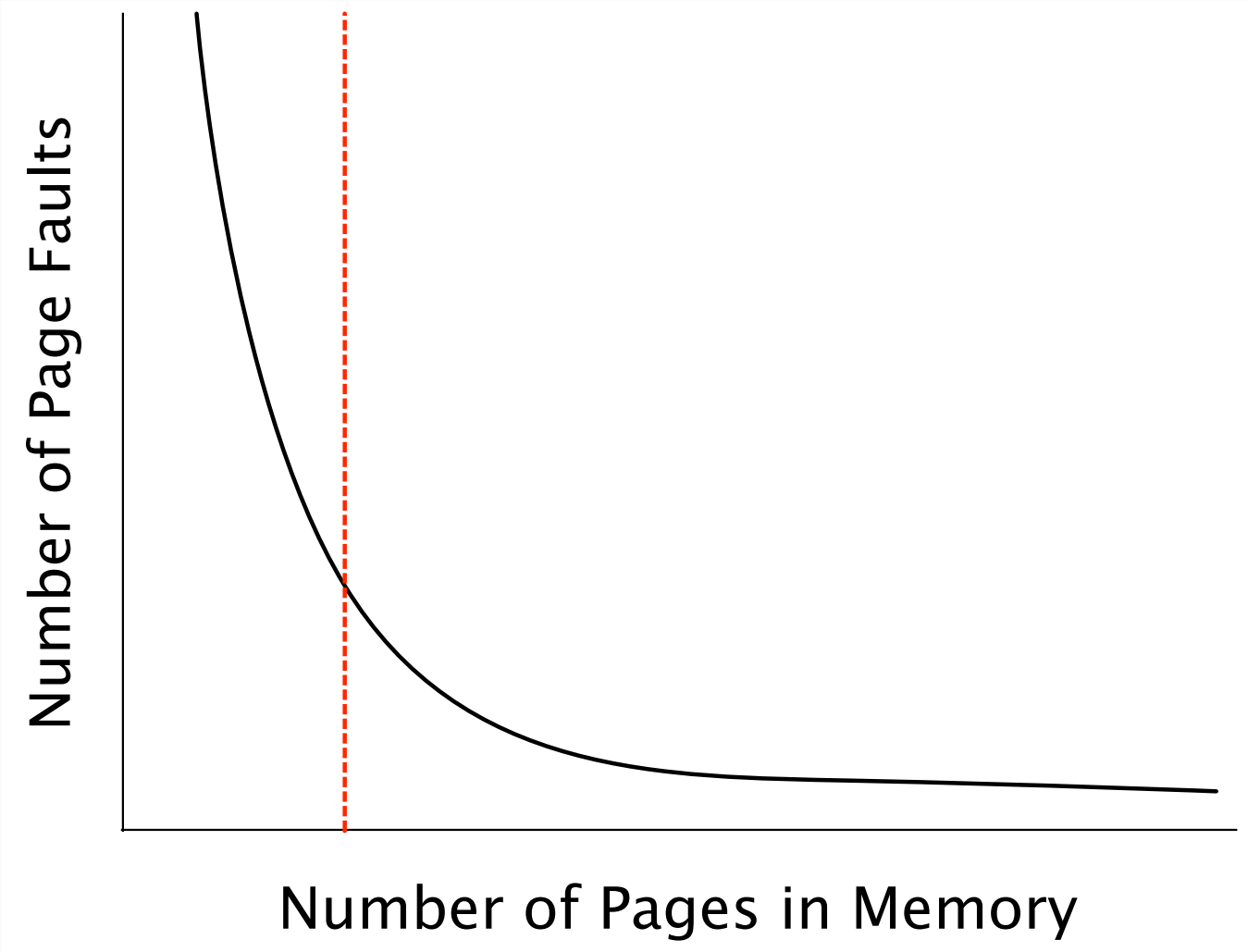


Figure: **Diminishing marginal returns: the 80/20 rule.**

What do we know about program behavior?

```
for(i = 0; i < 100; i++)
    for(j = 0; j < 100; j++)
        x[i] = x[i]*j;
```

Temporal Locality

- The instructions to read and update x[i] are temporally local.
- The memory reference to x[i], i is also temporally local.

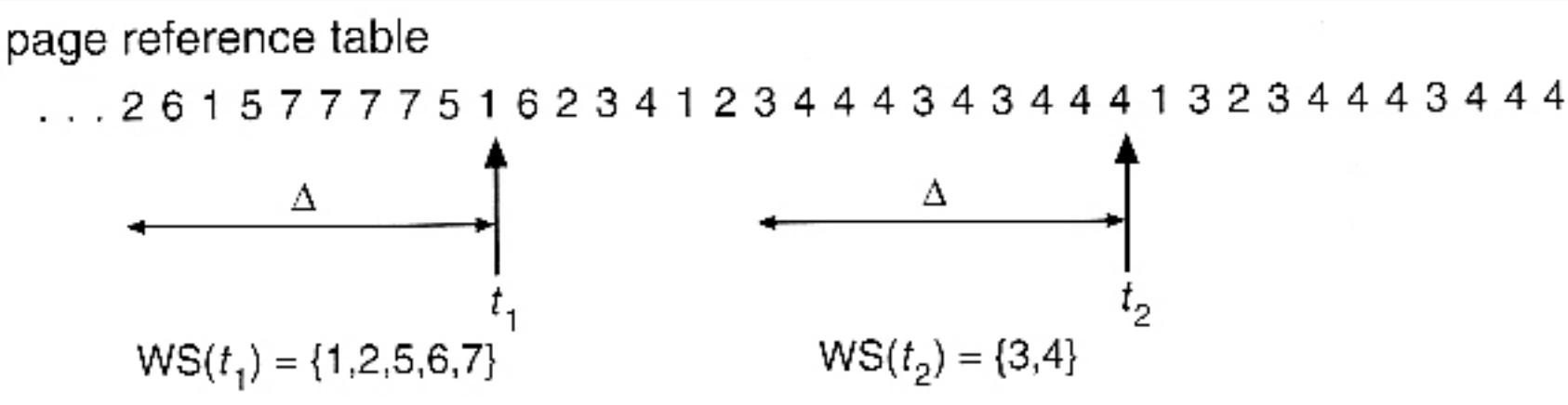
Spatial Locality

- After referring x[1], we expect to refer to x[2].
- Neighboring data items are likely to be accessed next.
- Neighboring code is likely to be accessed next.

Working set

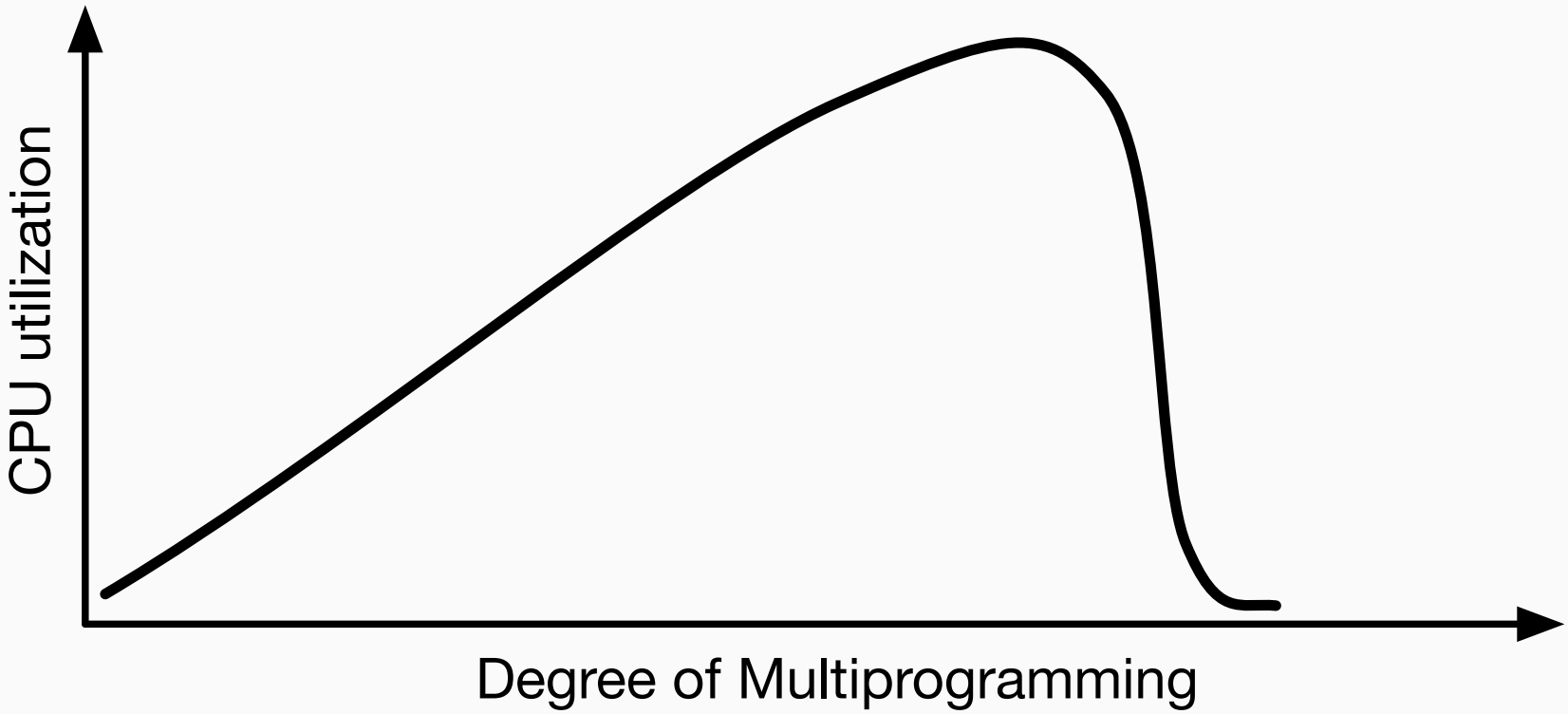
The **working set** of a process is the set of pages it currently needs in memory to function well: *e.g. without page faults*. The set evolves with time.

Denning, 1968 definition: $WS(t, \Delta)$ of a process at time t is the collection of pages referenced by the process during the process time interval $(t - \Delta, t)$.



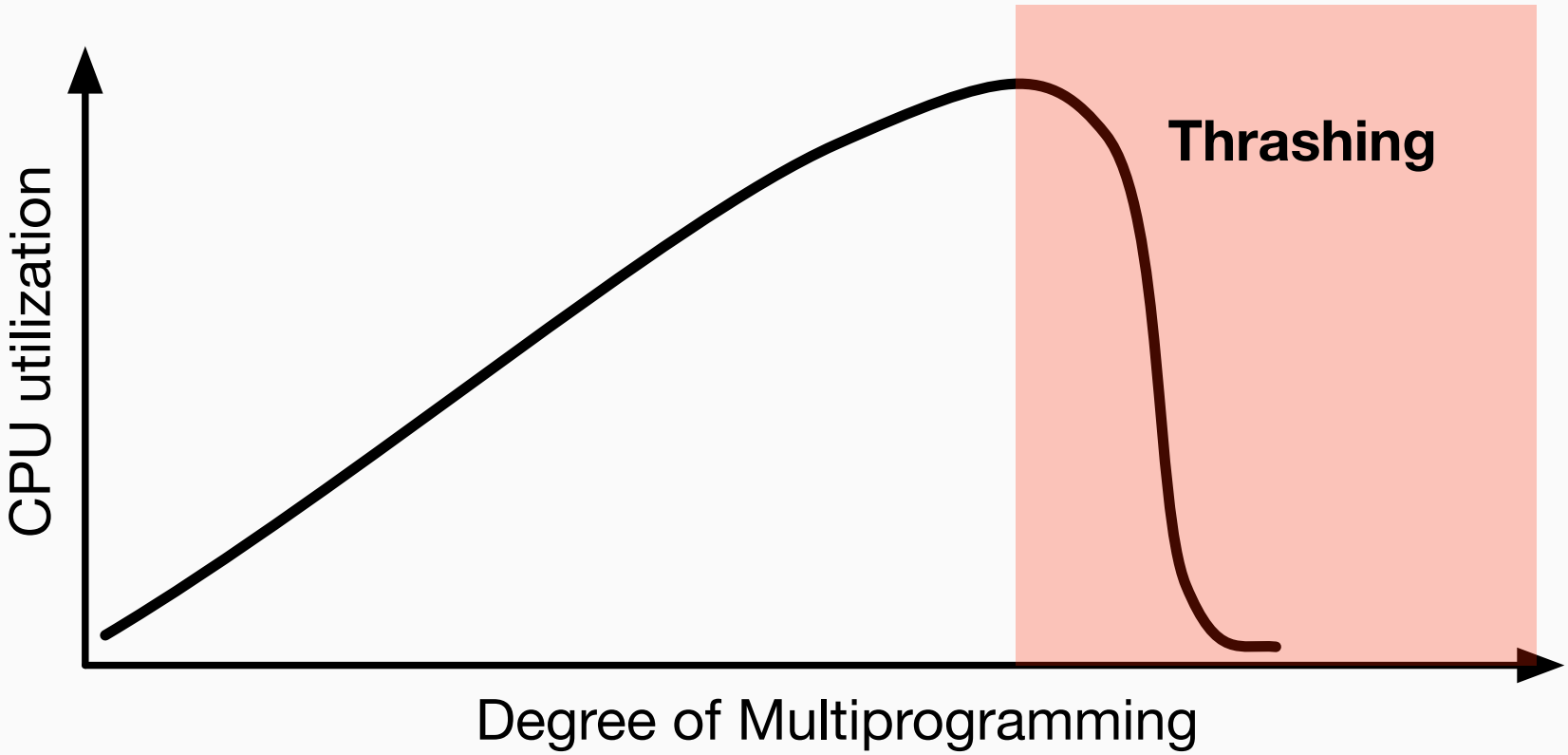
WS is a concept not an algorithm and we can try to design page-replacement algorithms that do not swap out the WS.

What is happening here?



If your CPU is underutilized, you'd expect that adding more processes should increase CPU utilization.

Thrashing



Thrashing: Processes spend more time paging than executing!

Solution 1: Isolation

Isolate the effects of thrashing

If one process starts to thrash, it should not steal frames from other processes and cause them to trash.

But...

1. Increases fault service time because the paging device's swap-in queue is full of the thrashing process's requests.

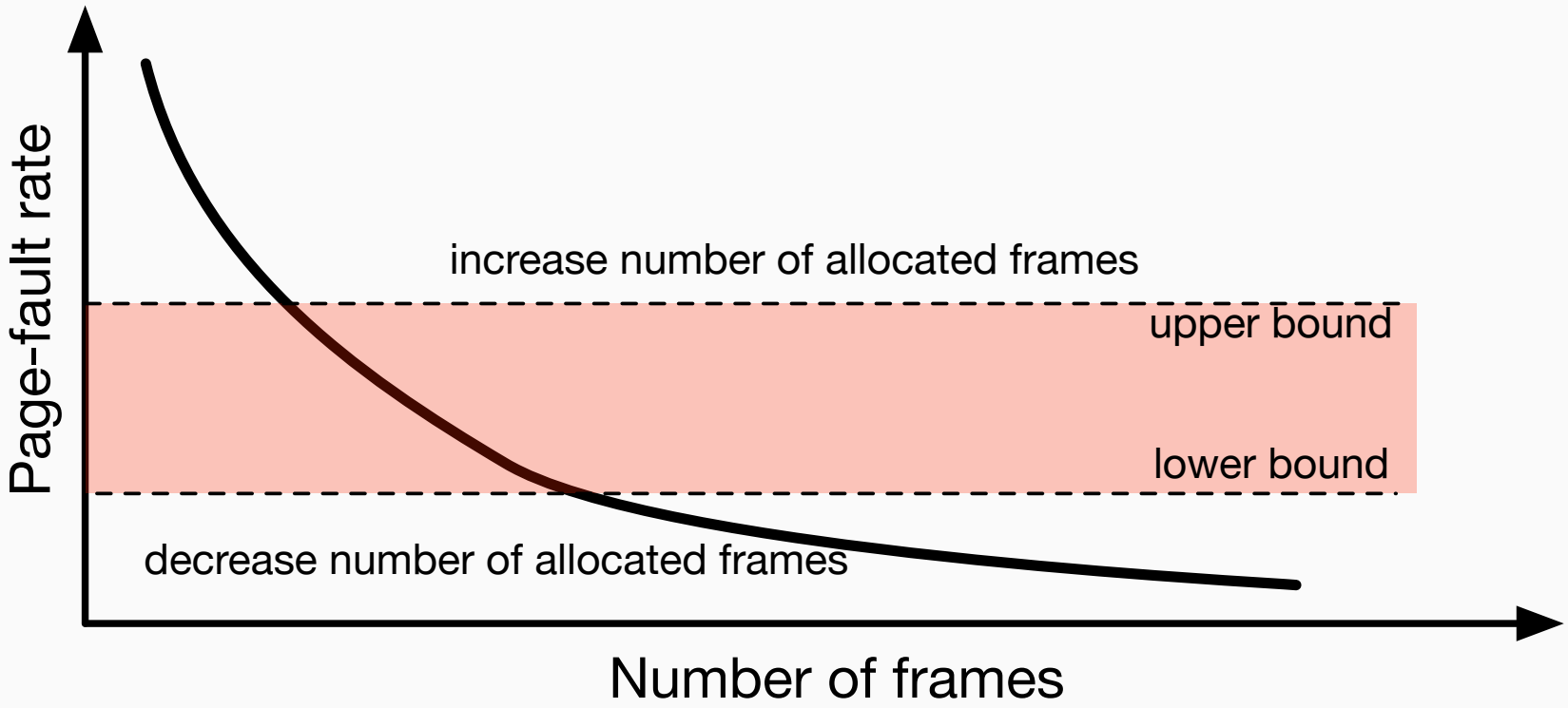
2. So non-thrashing process will still suffer

Solution 2: Working Set Model

If you cannot provide a process with its working set, then suspend the process entirely until you can. Schedule it later!

Can we determine the right # of pages to allocate?

1. Use **Page fault rates** to estimate of the size of the working set.



2. Track previous executions of a process to determine its working set and **pre-page**.

How to write good programs?

Assume each page can hold only 64 integers.

```
int i, j;
int[64][64] data;

for(j = 0; j < 64; j++){
    for(i = 0;
        i < 64; i++){
        data[i][j] = 0;
    }
}
```

```
int i, j;
int[64][64] data;

for(i = 0; i < 64; i++){
    for(j = 0;
        j < 64; j++){
        data[i][j] = 0;
    }
}
```

Hardware Optimizations: TLB

The cost of a memory access

What happens when we access memory movl 0xdeadbeef, %edi

We first have to look up the physical address of 0xdeadbeef.

Recall:

1. L1 Cache: 0.5ns; L2 Cache: 7-10ns
2. Memory reference: 100ns

Assume two-level page table (not in the L2) cache then we need to:

1. look up index in first level: 100ns
2. look up index in the second level : 100ns
3. retrieve the actual data at 0xdeadbeef: 100ns

Each memory access incurs an additional 200ns!

The Transaction Look-aside Buffer

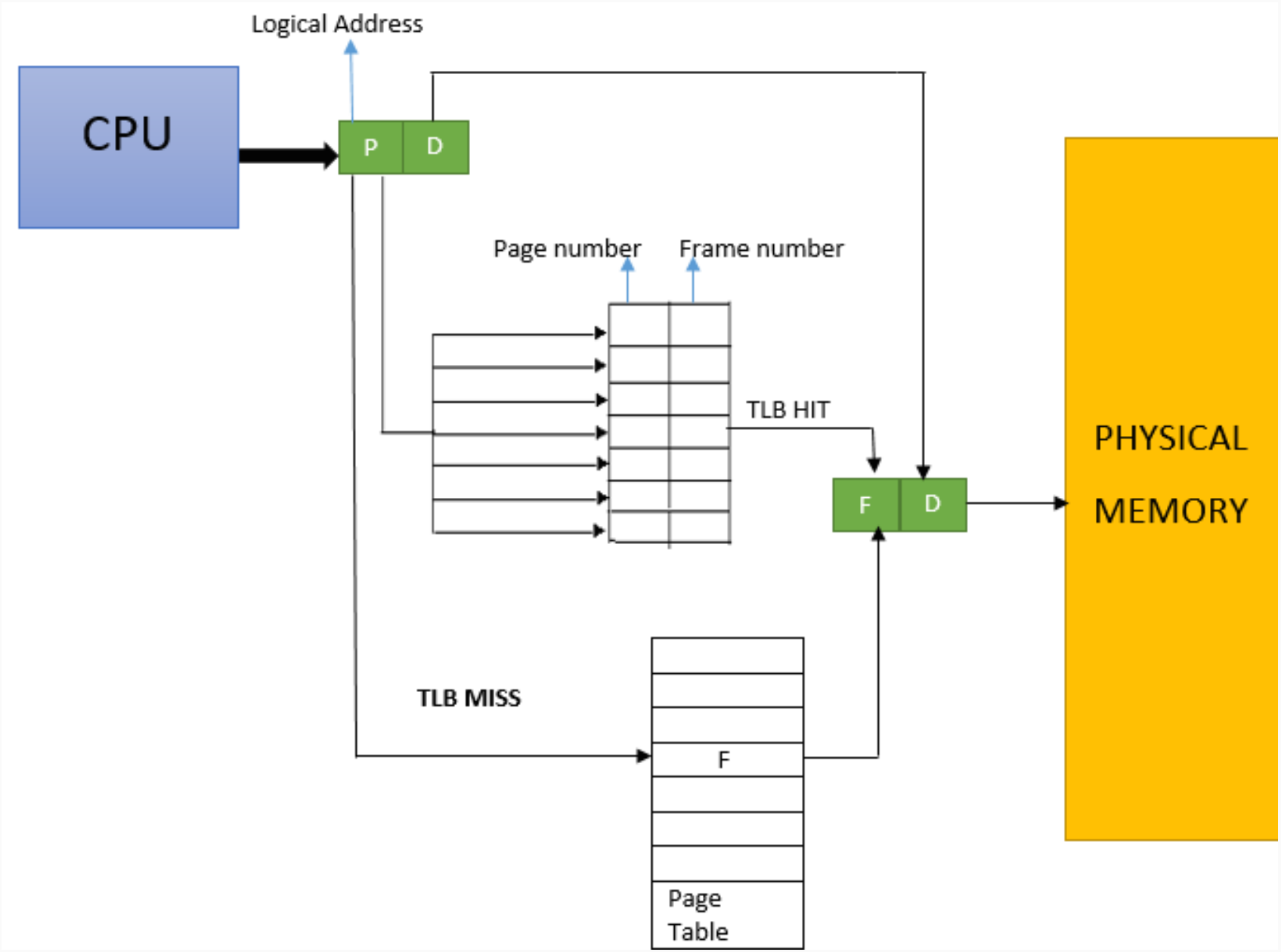
Can we do better?

1. TLB is a **Fully associative cache** (all entries searched in parallel)
2. Small: 16-64 entries. Why?
3. Can achieve 99% hit ratio. How?

Cache hit: address translation does not require an extra memory reference

Cache miss: must walk the page tables to translate.

Translation Lookaside Buffer



What happens on a cache miss?

Hardware-TLB: (x86 architecture). The hardware walks the page table and updates an entry into the TLB by evicting another entry usually by using LRU.

Software-TLB: Traps to the kernel and the kernel decides how to walk the page tables and update the TLB entry. Allows for greater flexibility in terms of page table structure, simpler hardware, but comes at a performance cost!

What happens on a context switch?

TLB must be **purged/flushed** (using a special hardware instruction) unless entries are tagged with a process ID; otherwise, the new process will use the old process's TLB entries and reference its page frames!

What is the cost of a cache miss?

Suppose the hit rate is 80%, accessing the TLB is on the order of accessing an L2 cache (10-20ns).

Again assume 2-level pages:

$$0.8 \times (20ns) + 0.2 \times (200ns + 20ns) = 16ns + 44ns = 60ns$$

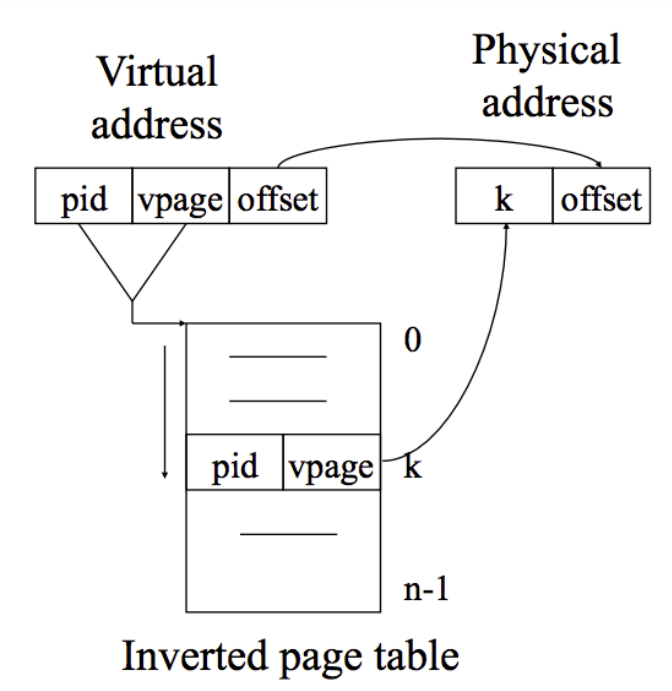
The first term: There is an 80% chance we only need to spend 20ns to obtain the translation.

The second term: In the 20% chance we get a TLB miss, we need to traverse two pointers to get the translation and then populate the entry in the TLB.

Compare only 60ns with 200ns of lookup without the TLB!

Small RAM

Inverted Page Tables



With 1GB of RAM, there is no point representing every possible virtual address, we have a small fixed size of available page frames anyway.

1. Table occupies a fixed fraction of memory. The size is proportional to physical memory, not the virtual address space.

2. The inverted page table is a **global structure**. It stores **reverse mappings** for all processes. Each entry has a tag with the task id and the virtual address for each page.

3. hash (task id, page number): match, translate the address. If not, use **collision resolution** technique (rehash, search, linear probing) and search again.

Complexities of Page Tables

Synchronization

1. Several operations occurring at the same time: swapping in, eviction, DMA (hardware transferring pages without CPU intervention), ...

2. Free frame no longer free but write operation hasn't started yet.

3. Processes sharing the same page

4. Processes dying before their swap-out/in completes

Several data structure locks and page-frame-level locks to lock/pin individual page frames as well as global structures during updates.

Zero-pages?

Zeroing pages: Initialize a page with 0's before allocating to a process.

Copy-on-write

Two process share most of the initial state but diverge over time

For example,

1. *parent & child in a Unix fork()*
   Don't allocate duplicate copies of a page; instead make page read-only, point to same page, on write $\rightarrow$ page fault $\rightarrow$ this copies the page!

2. *Processes that start by requesting large blocks of zeros in address space*
   big global arrays; sbrk() a call to increase heap space; Hand out thousands of pointers to the same empty page and allocate real page only when we need to.

Preferential Treatment for Kernel Pages

Kernel memory is allocated from a free-memory pool different than what is used for user-processes. Why?

1. **Minimize waste:** no internal fragmentation as much as possible. Kernel typically resides within a small space of physical memory.

2. **Contiguous physical memory:**
   - some devices would like to side-step the VM system entirely
   - performance: to make nonphysically contiguous pages contiguous in the virtual address space, must specifically set up the page table entries: results in TLB thrashing!

3. **Kernel buffers:** special data structures for devices to copy data directly to without VM intervention.

Free lists: Allocating and freeing data structures is one of the most common operations inside any kernel. A free list contains a block of available, already allocated, data structures.

Sharing with Page Tables

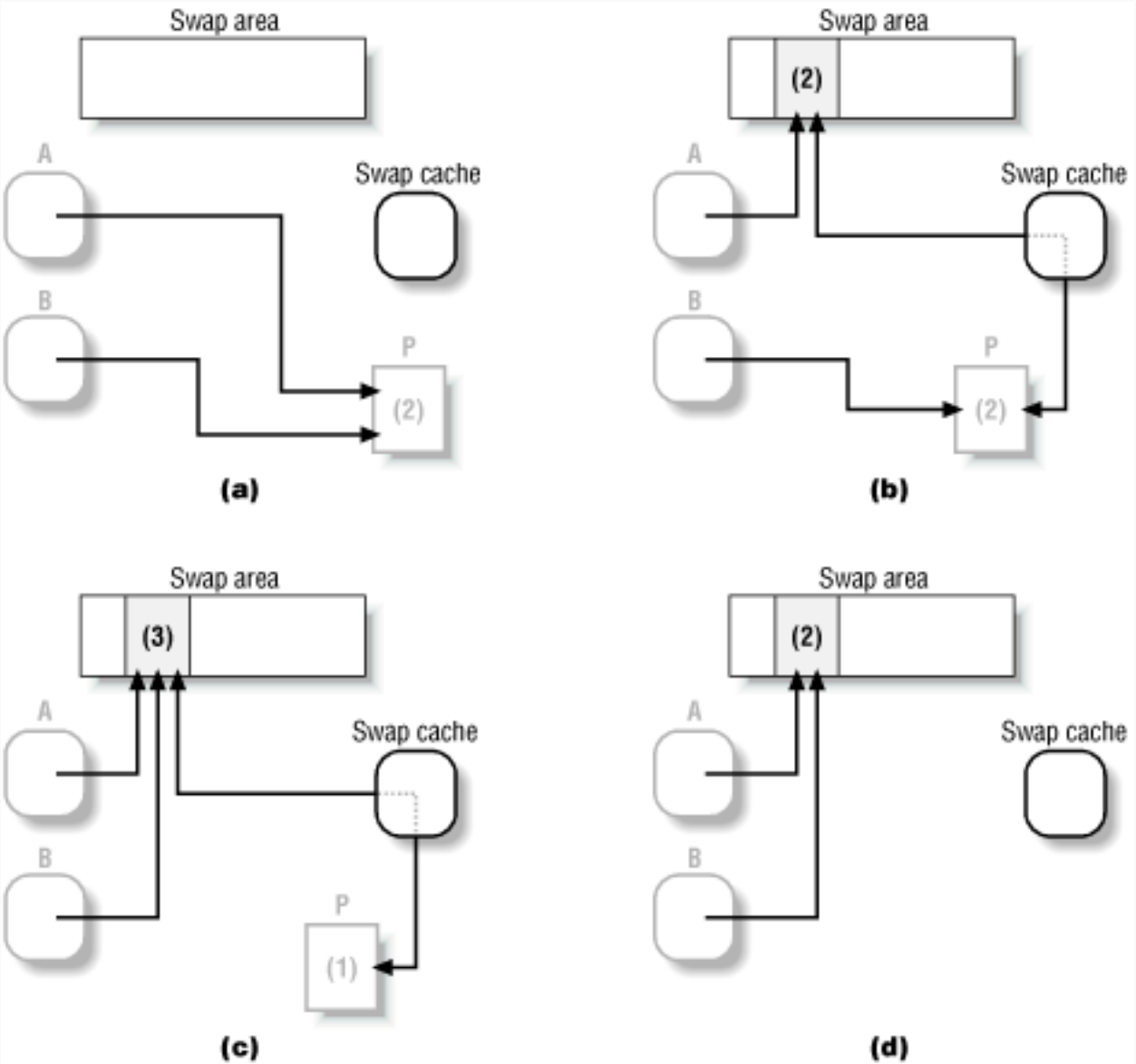
Shared Pages

PTEs from two processes share the same physical pages.
Use cases?

Easier said than done:

1. What happens when one process destroyed?
2. How do we page in/page out shared pages? (All PTEs need to be updated)
3. Pin and Unpin shared pages
4. What represents the working set for each process?

Swapping out shared pages



Questions?