Virtual Memory

Goals

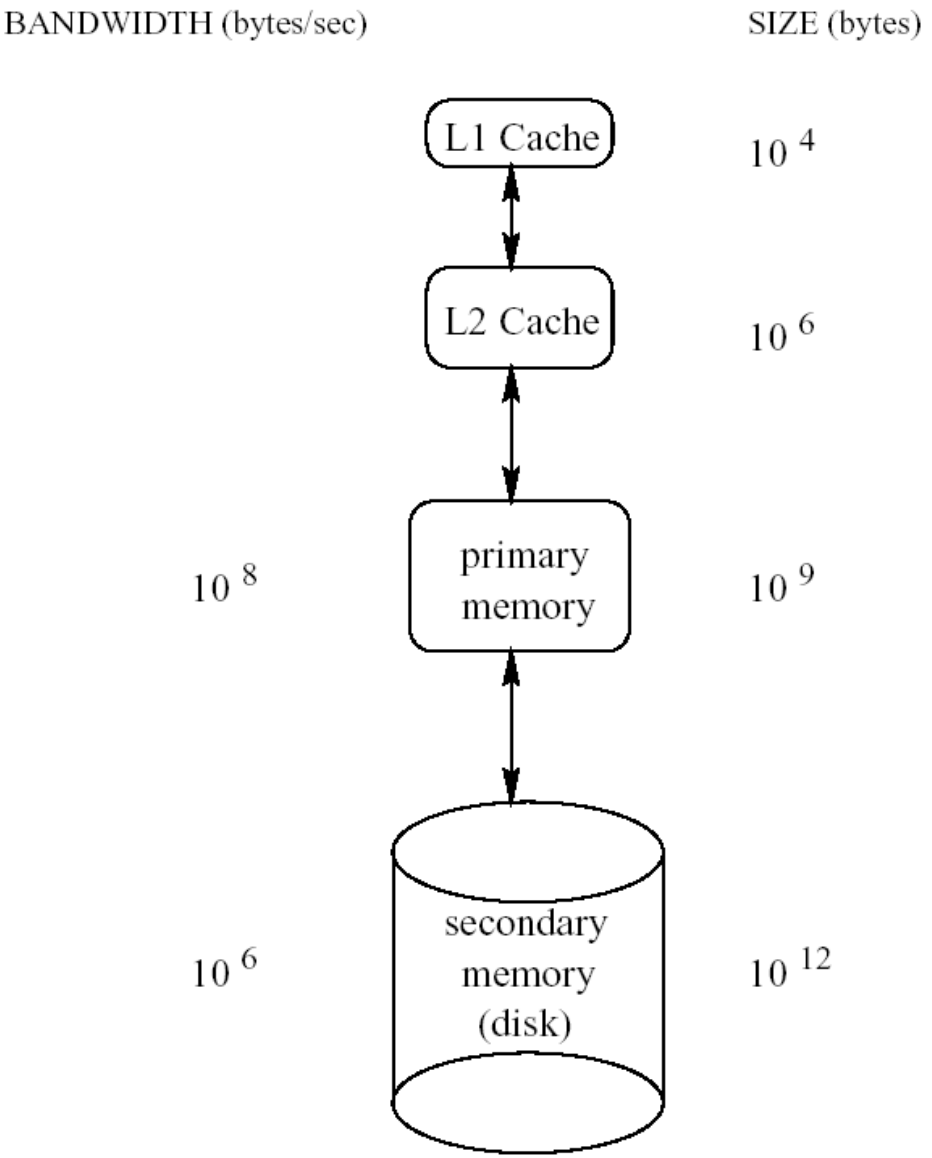
- to allow virtual address spaces that are larger than the physical address space (= available main memory);
- to allow more processes to run in parallel.

Methods

- allow pages (or segments) from a virtual address space to be stored in secondary memory (disk) instead of primary memory (RAM);
- move pages (or segments) between primary and secondary memory as needed.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

The Memory Hierarchy



Stefan Buettcher
<sbuettch@uwaterloo.ca>

Large Virtual Address Spaces

One advantage of paging is that a process can allocate memory without consuming any (or just very little) space in primary memory. The respective entry in the page table indicates “not in RAM”.

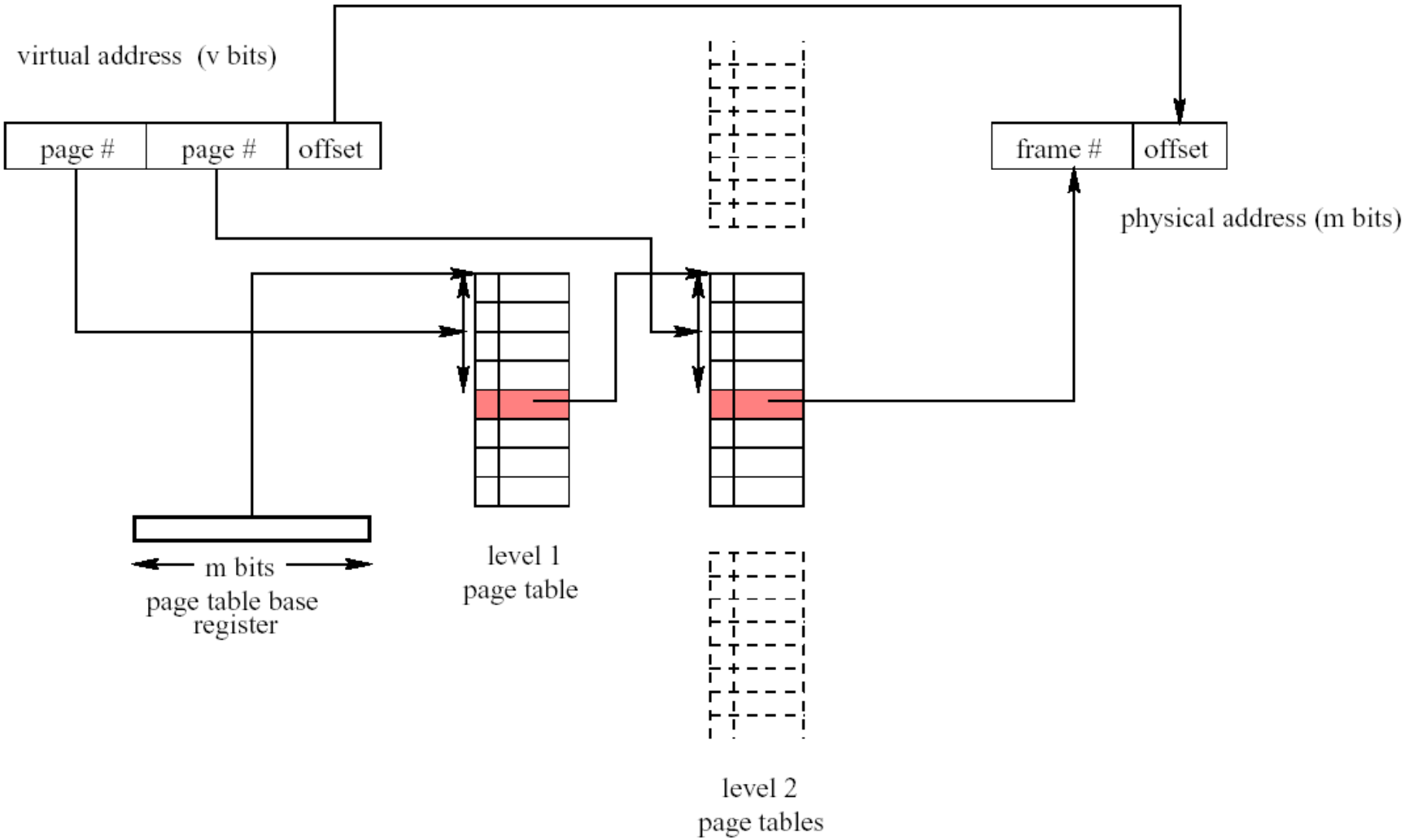
Problem:
For a very large virtual address space, this method still consumes too much memory.

Example:
2^{32} bytes virtual address space. Page size: 4 KB (2^{12} bytes). Each page table entry consumes 4 bytes.
⇒ Page table requires $4 * 2^{32} / 2^{12} = 2^{22}$ bytes (4 MB).
Needs to be stored in physical memory contiguously.

Possible solution:
Use multi-level page tables.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Two-Level Paging



Stefan Buettcher
<sbuettch@uwaterloo.ca>

Limits of Two-Level Paging

Two-level paging works fine for 32-bit machines (although it requires an additional memory access in case of a TLB miss).

On 64-bit hardware, however, two-level paging becomes infeasible.

Virtual address space: 64 bits
Page size 4 KB (12 bits)
Level-1 page table: 2^{26} entries (= 2^{29} bytes)
Level-2 page tables: 2^{26} entries (= 2^{29} bytes)

Possible solution:
Three-level paging (in general: *N*-level paging).

Problem:
Too many memory accesses. High effort to maintain page tables.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Inverted Page Tables

Maintaining per-process page tables requires memory space proportional to the number of pages in the system.

The number of pages can be much larger than the number of frames.

Why not implement paging through an inverted page table, mapping from frame ID to page ID?

Address translation can be realized by scanning the inverted page table, trying to find the frame that contains the given page.

Is this approach realistic? What is the performance?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Paging Policies

Paging allows to keep pages on disk temporarily and only load them into memory when they are used.

Demand paging brings pages into memory just before they are used.

The operating system can use a *prefetching* strategy; it tries to guess which pages are used next and loads them into memory before they are actually accessed.

If a page has to be loaded into memory, and there are no free frames to load it into, the operating system has to use a *replacement policy* to determine which other place to *evict* from memory, i.e. to move from RAM to disk.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Paging Mechanism

To support virtual memory, each entry in the page table gets a *valid* bit *V*. Its value indicates whether the page currently is in RAM or on disk:

V = 1 – valid entry (page in RAM), can be used for translation
V = 0 – invalid entry, MMU raises a *page fault* exception

The OS handles a page fault exception by

- determining which page caused the exception (in Nachos, the virtual address is stored in BadVAddrReg);
- loading the page into memory, possibly evicting another page.

After the OS loads the page into memory, the instruction that caused the page fault can be executed.

What are the implications of allowing unaligned memory access (e.g., on Intel x86 CPUs)?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Paging Mechanism

Paging + virtual memory allows us to treat secondary memory (hard disk) like main memory and use RAM as a cache.

Under this interpretation, special treatment has to be given to pages that are changed while they are in memory.

Definition:
A page that was changed after being loaded into RAM is called *dirty*.

A dirty page page is more costly to evict from RAM because the changed version of the page needs to be written to disk first.

The MMU identifies dirty pages by setting a *dirty* bit in the page table and clearing the dirty bit when it writes the page to disk.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Replacement Policies

How does the OS decide which page to replace when it has to load a page into memory?

Possible heuristics:

- evict the page that has been in memory for the longest time;
- evict the page that has been unused for the longest time;
- evict a page from a low-priority process;
- evict the page that has seen the least number of accesses;
- evict a page that is not dirty.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Optimal Replacement Policy

If the OS has knowledge about future behavior of all processes in the system, then there is an optimal replacement policy.

The OPT policy:
Replace the page that will not be referenced for the longest time.

Example with 3 frames:

Page reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame 1	A	A	A	A	A	A	A	A	A	C	C	C
Frame 2		B	B	B	B	B	B	B	B	B	D	D
Frame 3			C	D	D	D	E	E	E	E	E	E
Page fault?	Yes	Yes	Yes	Yes	No	No	Yes	No	No	Yes	Yes	No

Drawback: Not really realistic; requires knowledge about the internal structure of each process.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

The FIFO Policy

FIFO (first in, first out): Replace the page that has been in memory for the longest time.

Example with 3 frames:

Page reference	A	B	C	D	A	B	E	A	B	C	D
Frame 1	A	A	A	D	D	D	E	E	E	E	E
Frame 2		B	B	B	A	A	A	A	A	C	C
Frame 3			C	C	C	B	B	B	B	B	D
Page fault?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes	Yes

9 page faults (OPT: 7).

Replacement Policies – General Considerations

The principle of locality suggests that recent usage of a page should be considered when making the replacement decision:

- Pages that have been accessed recently are likely to be accessed again in the near future.
- Pages that are close to frequently accessed pages are likely to be accessed in the near future.

In addition, pages that have been heavily used might have a higher chance to be used again than pages that are used infrequently.

Cleanliness might also be worth considering – replacing a dirty page can sometimes be more costly than replacing a clean page.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Types of Locality

Locality is a property of the page reference string (the sequence in which pages are accessed).

Temporal locality – Pages that have been used recently are likely to be used again soon.

Spatial locality – Pages close to other pages that have been used are likely to be used as well.

Page reference strings usually exhibit strong locality. This is especially true for explicit (i.e., non-paged) file accesses.

Spatial locality and temporal locality are really two ways to look at the same phenomenon. Temporal locality is caused by spatial locality within the same page.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

The LRU Policy

LRU: Evict the *least-recently used* (i.e., "not accessed for the longest time") page.

Page reference	A	B	C	D	A	B	E	A	B	C
Frame 1	A	A	A	D	D	D	E	E	E	C
Frame 2		B	B	B	A	A	A	A	A	A
Frame 3			C	C	C	B	B	B	B	B
Page fault?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes

LRU in practice leads to very good results.

An implementation needs to keep track of the last access to each page, e.g., by putting a timestamp into each entry in a page table.

Because of this overhead, LRU is seldomly used as a replacement policy in operating systems.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

The *Use* Bit

To facilitate the implementation of a replacement policy, a *use* bit can be added to each page table entry.

The use bit is set by the MMU whenever a page is accessed.

It can be read and modified by the operating system.

Replacement strategies can base their decision on the values of the use bits of pages currently in memory.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

The CLOCK Replacement Algorithm

The CLOCK algorithm (a.k.a. *Second Chance*) selects a page for eviction based on the value of the *use* bit.

CLOCK behaves exactly like FIFO, except that it skips pages whose *use* bit is set.

When a page is loaded into memory, its *use* bit is set to 1.

Replacement algorithm:

```
while (frames[victim].useBit == 1) {
  frames[victim].useBit = 0;
  victim = (victim + 1) % NUMBER_OF_FRAMES;
}
toEvict = victim;
victim = (victim + 1) % NUMBER_OF_FRAMES;
```

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Enhanced Second-Chance Policy

The CLOCK algorithm can be improved by taking cleanliness into account.

Classify pages according to *use* and *dirty* bits:

- (0,0): not recently used, not dirty;
- (0,1): dirty, but not recently used;
- (1,0): recently used, but not dirty;
- (1,1): recently used and dirty.

Algorithm:

1. Do one rotation looking for a (0,0) page. Do not clear *use* bits.
2. If no victim found, do a second (or third) rotation, looking for (0,0) and (0,1) pages. Clear *use* bits while scanning the list.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Frequency-Based Replacement: LFU

LFU (least-frequently used): Replace the page with the smallest number of references.

In order to implement LFU, the MMU must support it by increasing the reference counter for a page whenever it is accessed.

Potential drawbacks of frequency-based policies:

- A page might exhibit a single burst, experiencing many accesses in a very short period of time, that is a bad basis for predicting the next access to the page.
- Recently loaded pages have small reference count and are likely to be evicted. This contradicts the locality assumption.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Page Cleaning

A dirty page must be cleaned before it can be evicted from primary memory.

Cleaning means copying the modified page to secondary memory.

Page cleaning can be either synchronous or asynchronous:

Synchronous cleaning – The victim page is written to disk just before the new page is loaded into memory. Happens inside the page fault exception handler.

Asynchronous cleaning – The victim page is written to disk before a page fault occurs. Can be done concurrently with other computations. Page faults can be handled more efficiently.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Prefetching

Prefetching refers to the process of loading pages into primary memory before a page fault occurs, by anticipating access patterns.

Prefetching can hide disk latency (just as asynchronous page cleaning).

Potential drawback: If the prefetching algorithm guesses wrong, it replaces a page that might be accessed in the near future.

Most common form: *Sequential prefetching*. If a process accesses page *N*, also load page *N+1* (and *N+2*, ..., *N+k*) into main memory.

Prefetching is a very effective strategy for dealing with file accesses, as most processes happen to access files in a sequential fashion (spatial locality).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Belady's Anomaly

Repeating the FIFO example from a few slides ago, but with 4 frames instead of 3:

Page reference	A	B	C	D	A	B	E	A	B	C	D	E
Frame 1	A	A	A	A	A	A	E	E	E	E	D	D
Frame 2		B	B	B	B	B	B	A	A	A	A	E
Frame 3			C	C	C	C	C	C	B	B	B	B
Frame 4				D	D	D	D	D	D	C	C	C
Page fault?	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes	Yes	Yes	Yes

The number of page faults increases from 9 to 10.

When FIFO is used as a replacement policy, increasing the number of frames does not necessarily decrease the number of page faults.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Stack Policies

Which replacement algorithms are affected by Belady's anomaly?

Let $B(m, t)$ denote the set of pages in a memory of size m at time t.

A replacement policy is called a *stack policy* if, for all reference strings, for all m, and for all t:

$$B(m, t) \subseteq B(m+1, t)$$

If a replacement algorithm imposes a total order on all pages, independent of the size of the memory, and always evicts the smallest page according to that order, then it is a stack policy.

LRU is a stack policy. FIFO and CLOCK are not.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Global vs. Local Page Replacement

A page replacement policy can be either *local* or *global*.

- A global replacement strategy is applied to all in-memory pages, regardless of the process they belong to. A page from process X may be replaced by a page from process Y.
- A local replacement strategy looks at each processes individually. A page from process X can only be replaced by a page from the same process.
- In order for a local replacement policy to work, a separate memory allocation policy first has to decide how much memory to allocate to a given process.

What are the advantages/disadvantages of either policy type?

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Memory Allocation Strategies

How much memory does a process need?

- The *working set model* assumes that at any given time some portion of a program's address space is heavily used, while the rest is not. The heavily used portion of its address space if called the *working set* of the process.
- The working set of a process may change over time.
- The resident set of a process is the set of pages that are located in primary memory (assigned a frame).
- The working set model says: If a process' resident set includes its working set, it will rarely page fault.

Warning

The above statements have close to zero information content!

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Memory Allocation Strategies

Getting more specific:

- Let $WS_P(t, \Delta)$ denote the set of pages accessed by process P during the time interval $(t - \Delta, t)$. $WS_P(t, \Delta)$ is P's working set at time t.
- Let $|WS_P(t, \Delta)|$ denote the size of $WS_P(t, \Delta)$, i.e., the number of distinct pages P's working set at time t.
- If the operating system could keep track of $WS_P(t, \Delta)$ for each P, it could:
  - Use $|WS_P(t, \Delta)|$ to determine the number of frames to allocate to P at time t;
  - Use $WS_P(t, \Delta)$ directly to realize a working-set-based page replacement policy (evict all pages not in the working set).

(Always keep in mind: Temporal locality!)

Page Fault Frequency

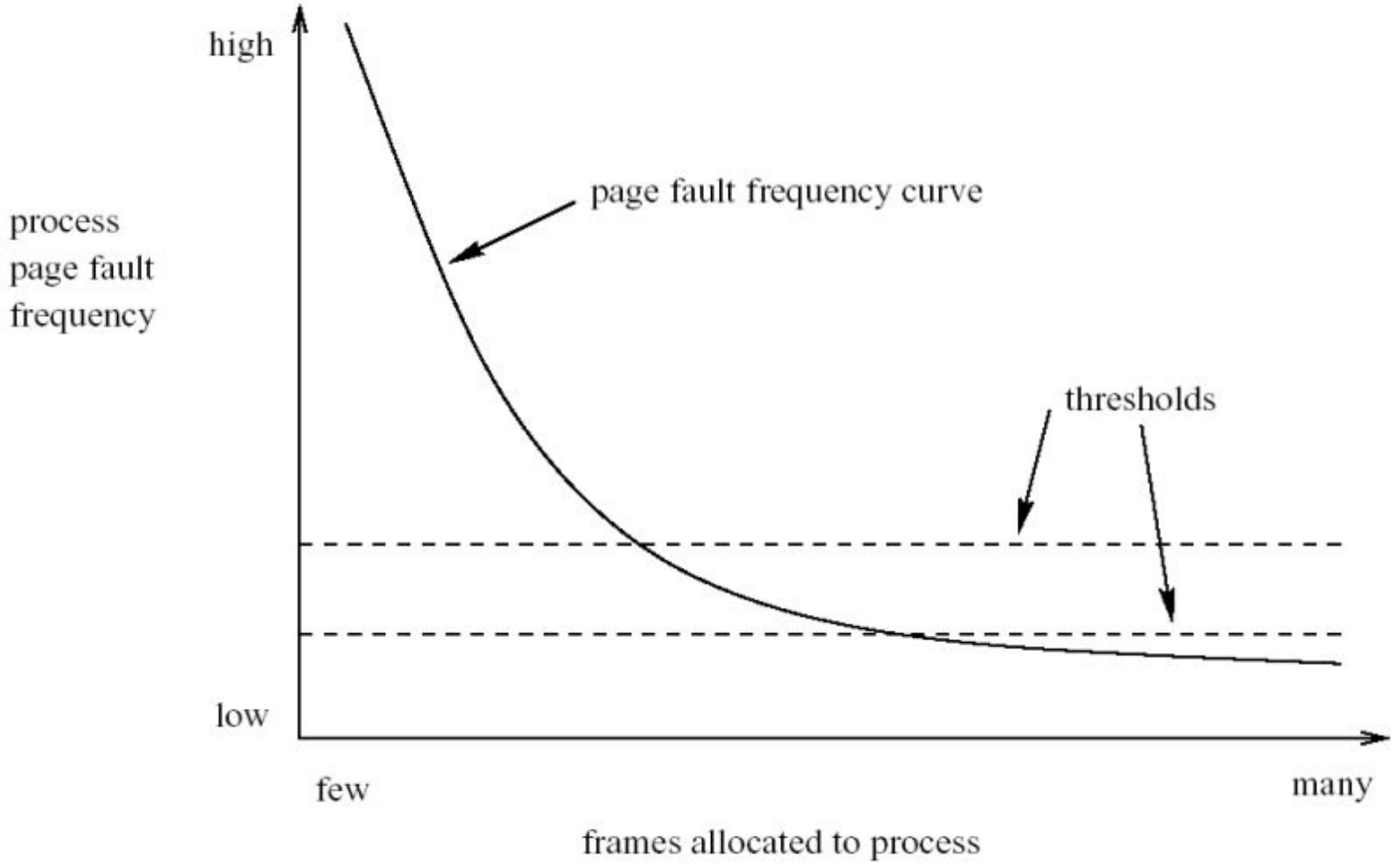
A different way to implement per-process memory allocation:

- Measure the page fault frequency of each process (number of page faults per time unit).
- Processes with lots of page faults need more primary memory.
- Processes with few page faults might be OK with a little less primary memory.
- The working set model suggests that a page fault frequency plot (page fault frequency vs. number of frames allocated for the process) should have a sharp knee.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Page Fault Frequency

A page fault frequency plot:



Stefan Buettcher
<sbuettch@uwaterloo.ca>

Thrashing and Load Control

A system that is spending too much time transferring pages between primary and secondary memory is said to be *thrashing*.

Thrashing happens when too many processes are competing for the primary memory in a computer system.

Thrashing can be cured by *load shedding*:

- killing processes;
- swapping out and suspending processes.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Swapping Out Processes

Swapping out a process means removing *all* its pages from primary memory.

When a process gets swapped out, it makes sense to suspend it (to avoid loading it back into main memory immediately). A *suspended* process is marked as not runnable.

Obvious benefit: Be able to use more memory for other processes.

Which processes should be swapped out?

- low priority processes;
- blocked processes (are suspended anyway);
- large processes (to free lots of primary memory);
- small processes (because easy to reload later on);
- nobody really has an answer to this... (⇒ simply do this as part of the general page replacement policy).

Stefan Buettcher
<sbuettch@uwaterloo.ca>