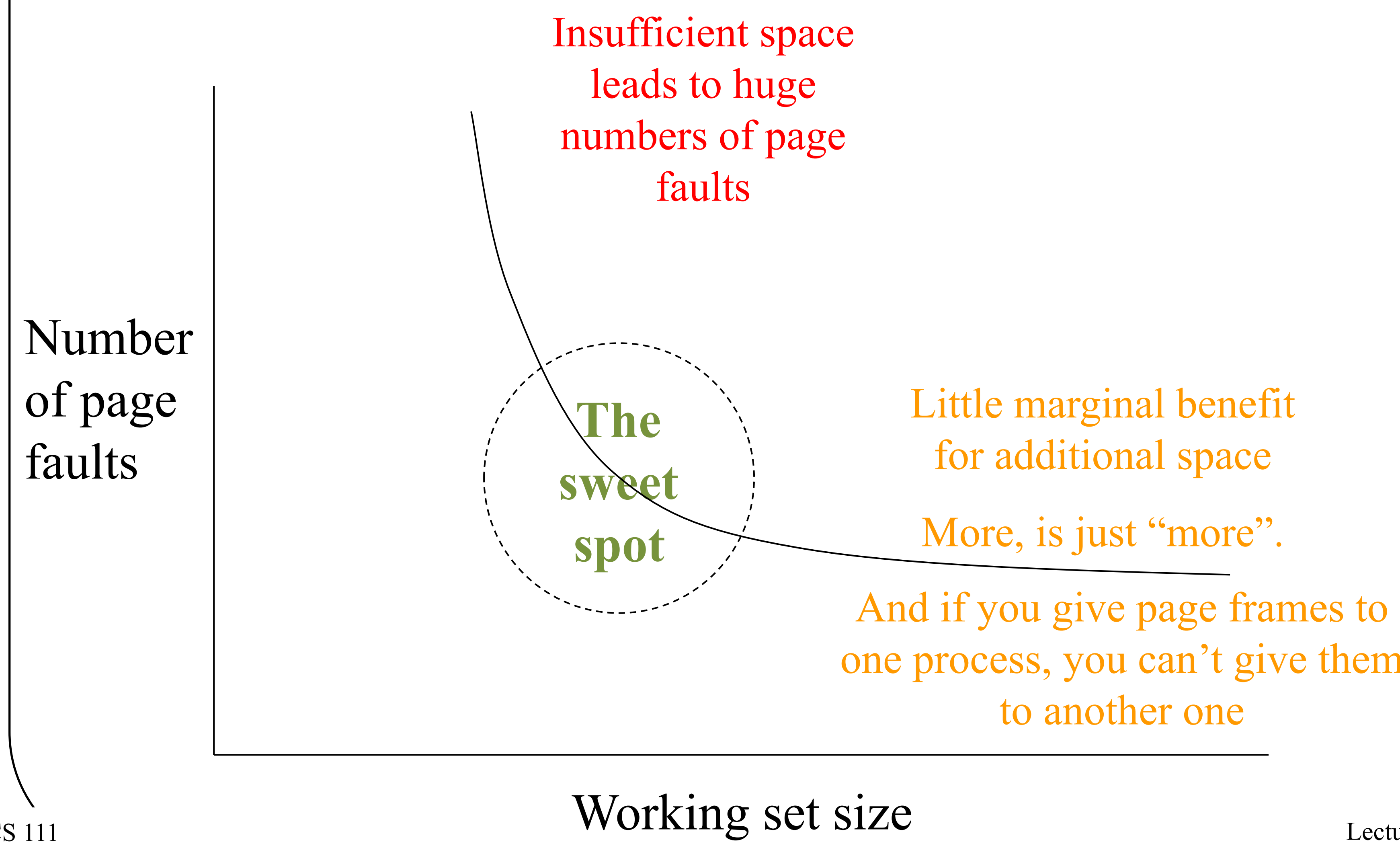
Working Sets

- Give each running process an allocation of page frames matched to its needs
- How do we know what its needs are?
- Use *working sets*
- Set of pages used by a process in a fixed length sampling window in the immediate past[1]
- Allocate enough page frames to hold each process' working set
- Each process runs replacement within its own set

[1]This definition paraphrased from Peter Denning's definition

The Natural Working Set Size



Optimal Working Sets

- What is optimal working set for a process?
  - Number of pages needed during next time slice
- What if try to run the process in fewer pages?
  - Needed pages will replace one another continuously
  - This is called *thrashing*
- How can we know what working set size is?
  - By observing the process' behavior
- Which pages should be in the working-set?
  - No need to guess, the process will fault for them

Implementing Working Sets

- Manage the working set size
  - Assign page frames to each in-memory process
  - Processes page against themselves in working set
  - Observe paging behavior (faults per unit time)
  - Adjust number of assigned page frames accordingly
- Page stealing (WS-Clock) algorithms
  - Track last use time for each page, for owning process
  - Find page least recently used (by its owner)
  - Processes that need more pages tend to get more
  - Processes that don't use their pages tend to lose them

Working Set Clock Algorithm

page frame	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
referenced	0	1	0	1	1	0	0	0	0	1	0	0	1	1	0
process	P_0	P_0	P_1	P_2	P_2	P_1	P_1	P_0	P_2	P_0	P_1	P_2	P_0	P_1	P_2
last ref	15	51	69	65	80	15	75	33	72	54	23	25	45	25	47

Clock pointer (pointing to page frame 6)

current execution times: $P_0 = 55$ | $P_1 = 75$ | $P_2 = 80$ | t = 15

P_0 gets a fault

- page 6 was just referenced
  - clear ref bit, update time
- page 7 is (55-33=22) ms old
  - P_0 replaces his own page

Stealing a Page

Page frame	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
referenced	0	1	0	1	1	0	0	0	0	0	0	0	1	1	0
process	P_0	P_0	P_1	P_2	P_2	P_1	P_1	P_0	P_2	P_0	P_0	P_2	P_0	P_1	P_2
last ref	15	51	69	65	80	15	75	33	72	54		25	45	25	47

Clock pointer

current execution times $P_0 = 55$ $P_1 = 75$ $P_2 = 80$ t = 25

P_0 has been experiencing too many page faults recently

P_0 gets a fault

page 6 was just referenced

page 7 is (55-33=22) ms old

page 8 is (80-72=8) ms old

page 9 is (55-54=1) ms old

page 10 is (75-23=52) ms old

P_0 steals this page from P_1

Thrashing

- Working set size characterizes each process
  - How many pages it needs to run for τ milliseconds
- What if we don't have enough memory?
  - Sum of working sets exceeds available memory
  - We will thrash unless we do something
- We cannot squeeze working set sizes
  - This will also cause thrashing
- Reduce number of competing processes
  - Swap some of the ready processes out
  - To ensure enough memory for the rest to run
- We can round-robin who is in and out

Pre-Loading

- What happens when process comes in from disk?
- Pure swapping
  - All pages present before process is run, no page faults
- Pure demand paging
  - Pages are only brought in as needed
  - Fewer pages per process, more processes in memory
- What if we pre-loaded the last working set?
  - Far fewer pages to be read in than swapping
  - *Probably* the same disk reads as pure demand paging
  - Far fewer initial page faults than pure demand paging

Clean Vs. Dirty Pages

- Consider a page, recently brought in from disk
  - There are two copies, one on disk, one in memory
- If the in-memory copy has not been modified, there is still a valid copy on disk
  - The in-memory copy is said to be "clean"
  - Clean pages can be replaced without writing them back to disk
- If the in-memory copy has been modified, the copy on disk is no longer up-to-date
  - The in-memory copy is said to be "dirty"
  - If swapped out of memory, must be written to disk

Dirty Pages and Page Replacement

- Clean pages can be replaced at any time
  - The copy on disk is already up to date
- Dirty pages must be written to disk before the frame can be reused
  - A slow operation we don't want to wait for
- Could only swap out clean pages
  - But that would limit flexibility
- How to avoid being hamstrung by too many dirty page frames in memory?

Pre-Emptive Page Laundering

- Clean pages give memory scheduler flexibility
  – Many pages that can, if necessary, be replaced
- We can increase flexibility by converting dirty pages to clean ones
- Ongoing background write-out of dirty pages
  – Find and write-out all dirty, non-running pages
    - No point in writing out a page that is actively in use
  – On assumption we will eventually have to page out
  – Make them clean again, available for replacement
- An outgoing equivalent of pre-loading