CS-446/646

OS & Paging

C. Papachristos
Robotic Workers (RoboWork) Lab
University of Nevada, Reno

OS & Paging

Remember: ***Paging***

Paging from the OS perspective:

- *Pages* are evicted *("Paged-Out")* to Disk when *Memory* is full
- *Pages* loaded *("Paged-In")* from Disk when referenced again
- References to evicted *Pages* cause a TLB *Miss*
- *Page Table Entry* indicates it is *Invalid*, an attempt to access triggers a *Page Fault*
- OS *Page Fault Handler* executed, OS allocates a *Page Frame*, reads *Page* from Disk
- When I/O completes, the OS fills-in *Page Frame*, marks it as *Valid*, and restarts the *Faulting Instruction*

Dirty vs *Clean Pages*

- Actually, only *Dirty (/Modified) Pages* need to be written to Disk
- *Clean Pages* do not – But we need to know where they are on Disk to read them again

Restarting Faulting Instructions

- Hardware provides Kernel with information about *Page Fault*
  - *Faulting Virtual Address* (In **`%CR2`** *Reg* on x86 – e.g. would see it if modifying Pintos **`page_fault`** and use **`fault_addr`**)
  - *Address* of the *Instruction* that caused *Fault*
  - Additional information about: Was the access a read or write? Was it an *Instruction* fetch? Was it caused by *User-level* access to *Kernel-level* mapped *Memory*?
- Hardware must allow resuming after a *Fault*
  - *Idempotent Instructions* are easy to restart
    - e.g. simple **`load`** or **`store`** *Instruction* can be restarted immediately
    - Just re-execute any *Instruction* that only accesses one address
  - *Complex Instructions* must be restarted, too
    - e.g. x86 **`movs`** (move string) *Instruction*
    - Specify **`src`**, **`dst`**, **`count`** in **`%esi`**, **`%edi`**, **`%ecx`** *Registers*
    - On *Fault*, CPU *Registers* adjusted to resume where move left off

Paging Challenges

How to resume a *Process* after a *Fault*?

- Need to save *State* and resume

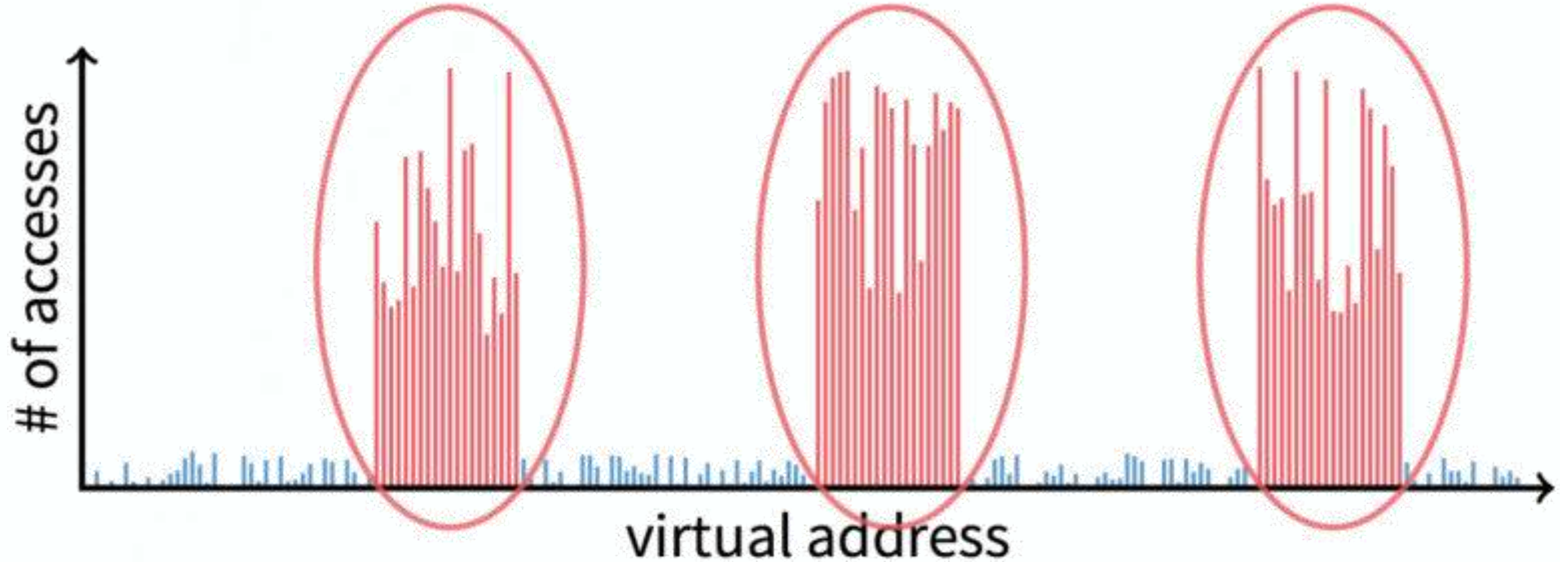
Page Replacement Policy

- What to fetch from Disk?
  - Just needed *Page* or more?
- What to evict?
  - How to allocate *Physical Pages* amongst *Processes*?
  - Which of a particular *Process*' *Pages* to keep in *Memory*?
  - Poor choices can lead to horrible Performance

Locality

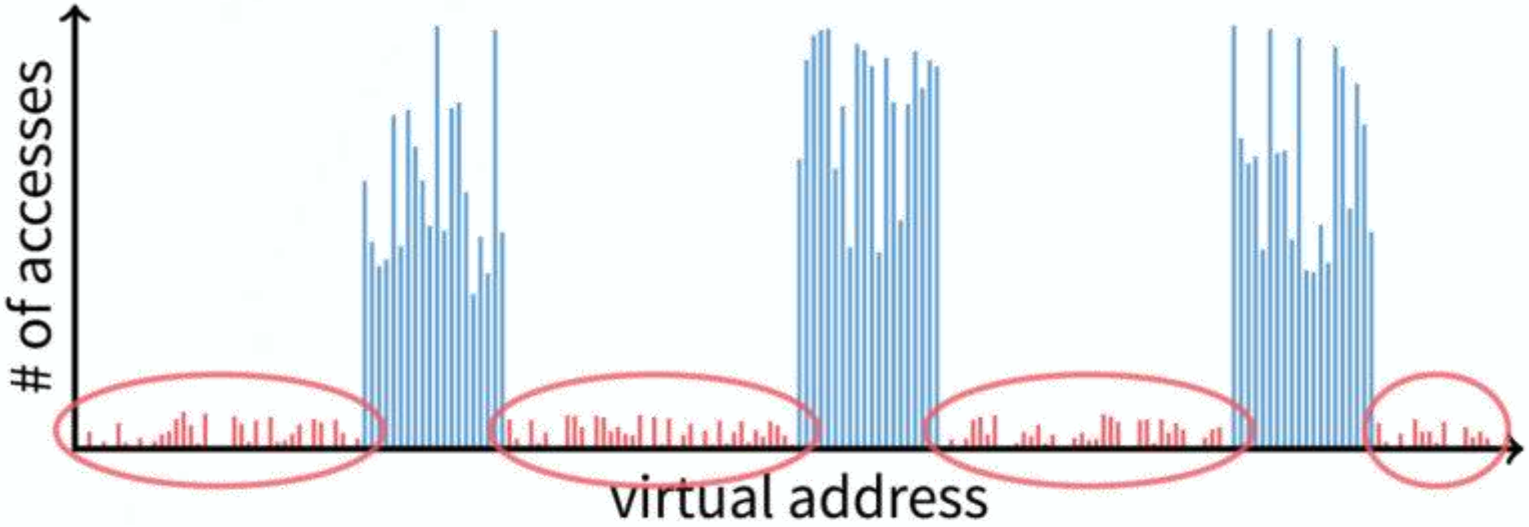
- All *Paging* schemes employ the concept of *Locality*
  - *Processes* reference *Pages* in localized patterns
- *Temporal Locality*
  - Concept: Locations referenced recently likely to be referenced again
- *Spatial Locality*
  - Concept: Locations near recently referenced ones are likely to be referenced soon
- Although the cost of *Paging* is high, if it is infrequent enough it becomes acceptable
  - *Processes* usually exhibit both kinds of *Locality* during their execution, making *Paging* practical

***Working Set* Model** (more later)



- Disk much slower than *Memory*
  - Goal: Run mostly at *Memory* speed, don't get throttled by Disk speed
- *"80/20 Rule"*: 20% of *Memory* gets 80% of *Memory* accesses
  - Keep the *Hot* 20% in *Memory*
  - Keep the *Cold* 80% on Disk

Working Set **Model** (more later)



- Disk much slower than *Memory*
  - Goal: Run mostly at *Memory* speed, don't get throttled by Disk speed
- *"80/20 Rule"*: 20% of *Memory* gets 80% of *Memory* accesses
  - Keep the *Hot* 20% in *Memory*
  - Keep the *Cold* 80% on Disk

Paging Challenges (continued)

What to Fetch?

- Bring in *Page* that caused *Page Fault*
- Pre-fetch surrounding *Pages*?
  - Reading two Disk *Blocks* approximately as fast as reading one
  - As long as no Track/Head switch needed, Disk *Seek Time* is what dominates
  - If application exhibits *Spatial Locality*, then big win to store and read multiple contiguous *Pages*
- Also *Pre-Zero* unused *Pages* in CPU Idle loop
  - Need 0-filled *Pages* for Stack, Heap, *Anonymously* `mmapp()` ed *Memory*
  - Zeroing them only on-demand is slower
  - Hence, many OSes will *Pre-Zero* freed *Pages* while CPU is Idle

Page Replacement

- When a *Page Fault* occurs, the OS loads the *Faulted Page* from Disk into a *Page Frame* of *Physical Memory*
- At some point, the *Process* will have used all the *Page Frames* it is allowed to use
  - This is likely (much) less than all of available *Memory*
  - *Remember:* OS usually keeps a *Pool* of *Free Pages* around so that allocations do not immediately cause evictions
- When this happens, the OS must **replace** a *Page* for each *Page Faulted-In*
  - It must evict a *Page* to free-up a *Page Frame*

The *Page Replacement* Algorithm determines how this is done

- Greatly affects performance of *Paging* (*Virtual Memory* Management)
- Also called the *Page Eviction Policy*

First-In First-Out (FIFO) Page Replacement

- Evict oldest **fetched** *Page*
- Example: *Page* Referencing string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 3 *Physical Pages* : 9 *Page Faults*

Phys Page	1	2	3	4	1	2	5	1	2	3	4	5
0	1	1	1	4	4	4	5			5	5	✓
1		2	2	2	1	1	1	✓		3	3	
2			3	3	3	2	2		✓	2	4	

First-In First-Out (FIFO) Page Replacement

- Evict oldest **fetched** *Page*
- Example: *Page* Referencing string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 4 *Physical Pages* : 10 *Page Faults*

Phys Page	1	2	3	4	1	2	5	1	2	3	4	5
0	1	1	1	1	✓		5	5	5	5	4	4
1		2	2	2		✓	2	1	1	1	1	5
2			3	3			3	3	2	2	2	2
3				4			4	4	4	3	3	3

Belady's Anomaly

- More *Physical Memory* does not necessarily mean fewer *Faults*

Optimal Page Replacement

- What is Optimal (if we knew the future)?
  - Replace *Page* that **will** not be used for longest period of time
- Example: *Page* Referencing string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 4 *Physical Pages* : 6 *Page Faults*

Phys Page	1	2	3	4	1	2	5	1	2	3	4	5
0	**1**	**1**	**1**	**1**	✓		**1**	✓			**5**	✓
1		**2**	**2**	**2**		✓	**2**		✓		**2**	
2			**3**	**3**			**3**			✓	**3**	
3				**4**			**5**				**4**	

Belady's Algorithm

- Known as the *Optimal Page Replacement* Algorithm
  - Rationale: The best *Page* to evict is the one never touched again
  - Never is a long time, so picking over a future *Time Horizon* is the next best thing
  - Proven by *Belady*
- Problem: Have to be able to predict the future
- Why is *Belady's Algorithm* useful then? As a comparative metric
  - Compare implementations of *Page Replacement* algorithms with the Optimal to gauge room for improvement
    - If Optimal is not much better, then our Algorithm is pretty good
    - If Optimal is much better, then our Algorithm could use some work
      - *Random Replacement* Algorithm is often the lower-bound

Least Recently Used (LRU) Page Replacement

- "Estimate" Optimal via *Least Recently Used (LRU)*
  - Rationale: Because past often predicts the future
  - Evict the *Page* that has not been **used** for the **longest time** in the past
- Example: *Page* Referencing string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 4 *Physical Pages* : 8 *Page Faults*

Phys Page	1	2	3	4	1	2	5	1	2	3	4	5
0	**1**	**1**	**1**	**1**	✓		**1**	✓		**1**	**1**	**5**
1		**2**	**2**	**2**		✓	**2**		✓	**2**	**2**	**2**
2			**3**	**3**			**5**			**5**	**4**	**4**
3				**4**			**4**			**3**	**3**	**3**

Least Recently Used (LRU) Page Replacement

- "Estimate" Optimal via *Least Recently Used (LRU)*
  - Rationale: Because past often predicts the future
  - Evict the *Page* that has not been **used** for the **longest time** in the past
- Example: *Page* Referencing string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 4 *Physical Pages* : 8 *Page Faults*
- Problem 1: Can be pessimal
  - e.g. when looping over *Memory*, we actually want *Most Recently Used (MRU)* eviction
- Problem 2: Implementation

Strawman *Least Recently Used (LRU)* Implementations

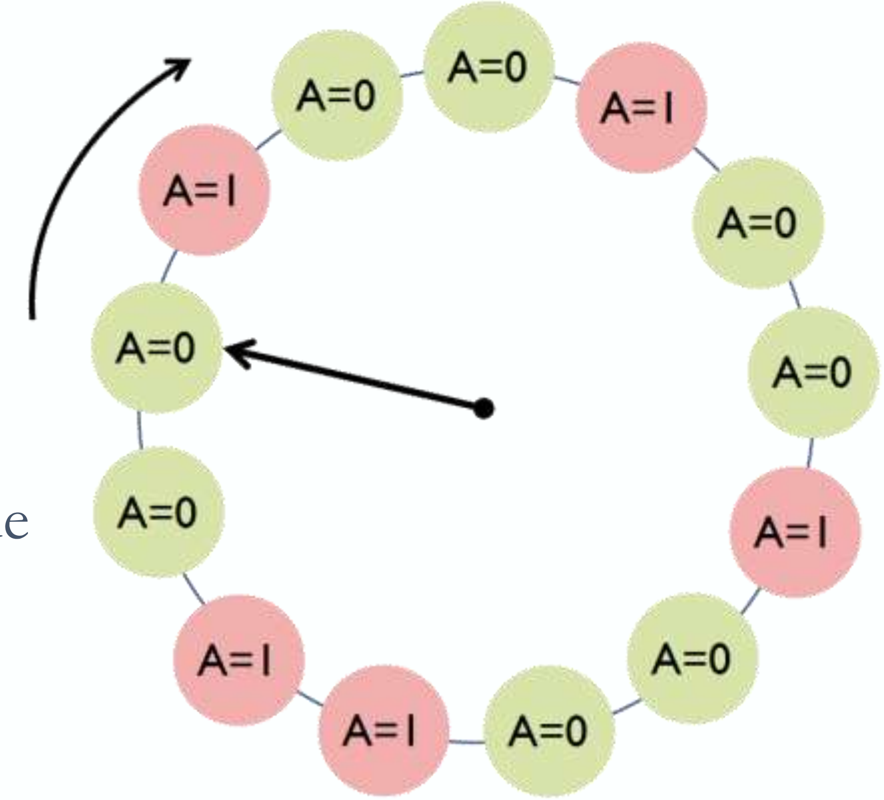
- Stamp *Page Table Entries* with Timer value
  - e.g., CPU has *Cycle* counter
  - Automatically writes value to *Page Table Entry* on each *Page* access
  - Scan *Page Table* to find oldest counter value ≡ LRU *Page*
  - Problem: Would double *Memory* traffic!
- Keep Doubly-Linked List of *Pages*
  - On access find *Page*, remove and re-insert it at Tail of List
  - Problem: Again, very expensive

Solution to accurate but expensive implementations

- Approximate LRU – the "*Clock Algorithm*"

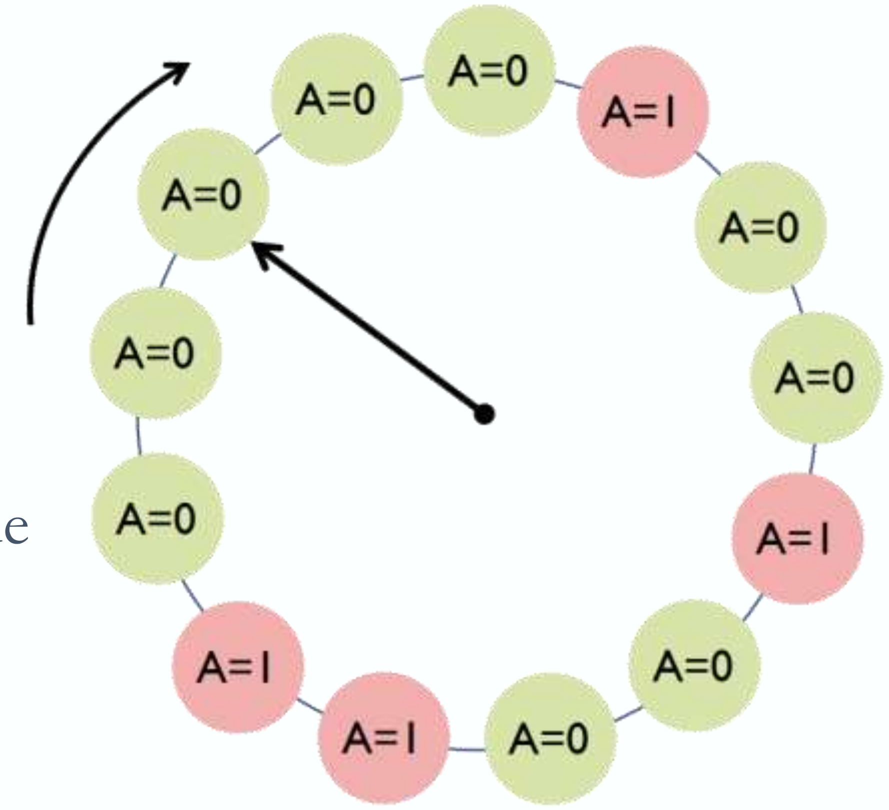
Clock Algorithm

- Use *Accessed* bit supported by most Hardware
  - e.g. Pentium will set *Accessed* bit in *Page Table Entry* on first access
  - Software-managed TLBs like MIPS can do the same
- Do FIFO but skip *Accessed Pages*
  - Keep *Pages* in circular FIFO List
- Scan:
  - if *Page's Accessed* bit == 1 then Set to 0; continue
  - else if *Accessed* bit == 0, Evict
- a.k.a. "*Second-Chance*" *Replacement*



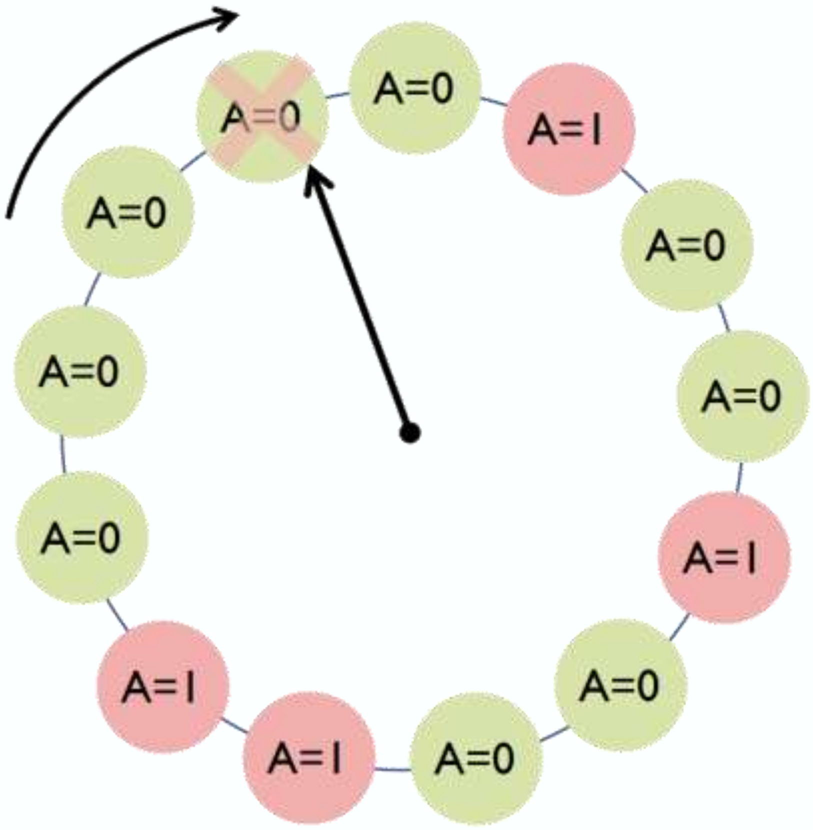
Clock Algorithm

- Use *Accessed* bit supported by most Hardware
  - e.g. Pentium will set *Accessed* bit in *Page Table Entry* on first access
  - Software-managed TLBs like MIPS can do the same
- Do FIFO but skip *Accessed Pages*
  - Keep *Pages* in circular FIFO List
- Scan:
  - if *Page's Accessed* bit == 1 then Set to 0; continue
  - else if *Accessed* bit == 0, Evict
- a.k.a. "*Second-Chance*" *Replacement*



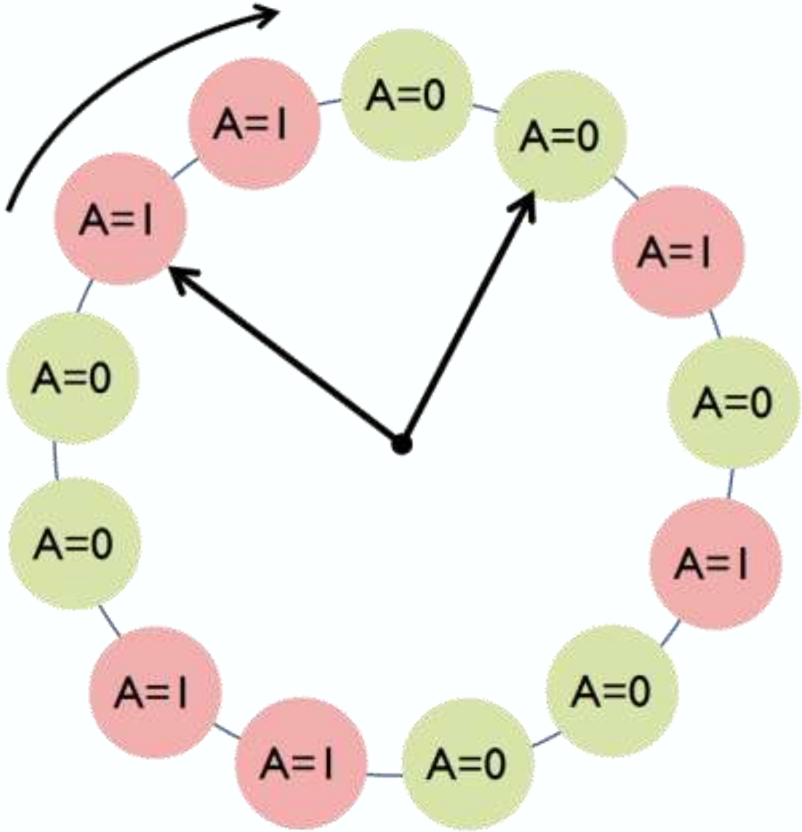
Clock Algorithm

- Use *Accessed* bit supported by most Hardware
  - e.g. Pentium will set *Accessed* bit in *Page Table Entry* on first access
  - Software-managed TLBs like MIPS can do the same
- Do FIFO but skip *Accessed Pages*
  - Keep *Pages* in circular FIFO List
- Scan:
  - if *Page's Accessed* bit == 1 then Set to 0; continue
  - else if *Accessed* bit == 0, Evict
- a.k.a. "*Second-Chance*" *Replacement*



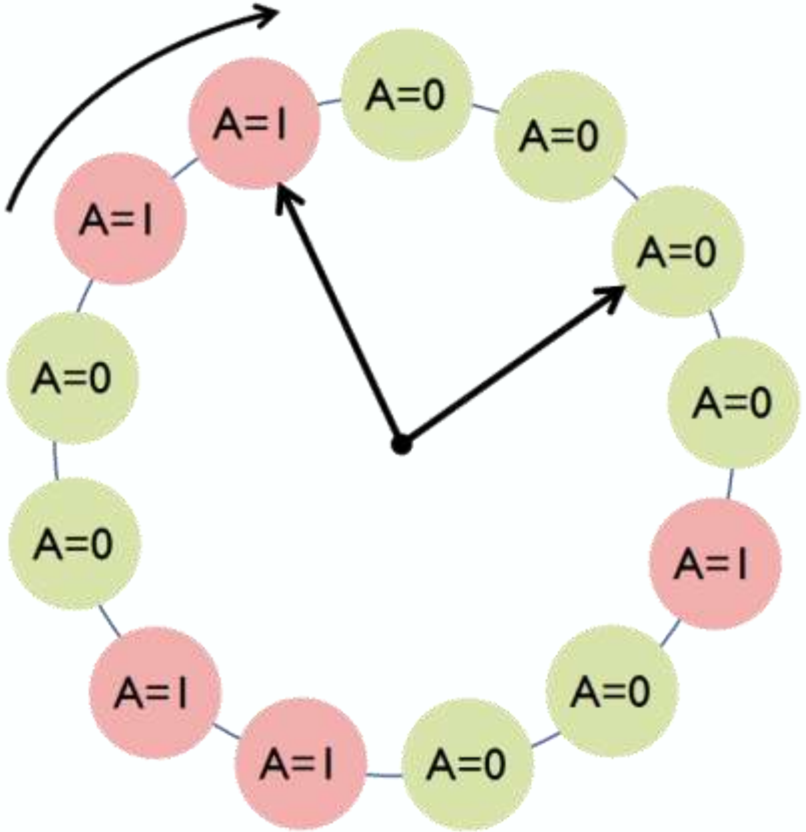
Clock Algorithm (continued)

- Large *Memory* may be a problem
  - Long intervals between re-checking *Pages*
- Add a second *Clock* hand
  - The 2 hands move in lockstep
  - Leading hand clears *Accessed* bits
  - Trailing hand evicts *Pages* with *Accessed*==0
- Can also take advantage of Hardware *Dirty* bit
  - Each *Page* either (*Unaccessed*, *Clean*), (*Unaccessed*, *Dirty*), (*Accessed*, *Clean*), or (*Accessed*, *Dirty*)
  - Consider *Clean Pages* for *Eviction* before *Dirty* ones
- Or use *n-bit Accessed* count instead just 1 *Accessed* bit
  - On sweep: `count = (A << (n - 1)) | (count >> 1)`
  - Evict *Page* with lowest `count`



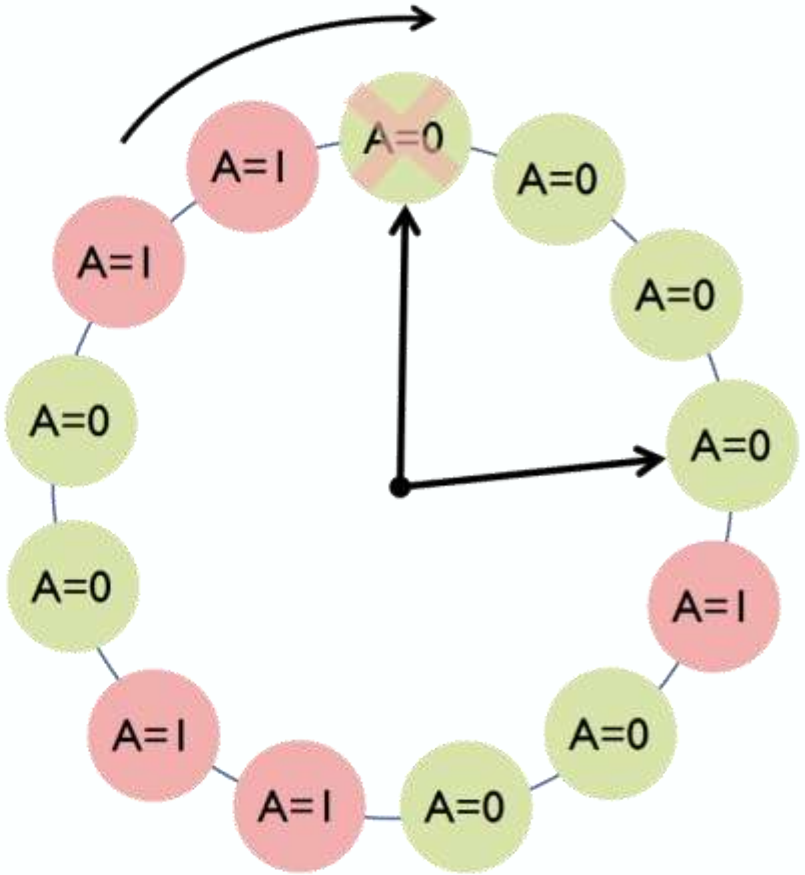
Clock Algorithm (continued)

- Large *Memory* may be a problem
  - Long intervals between re-checking *Pages*
- Add a second *Clock* hand
  - The 2 hands move in lockstep
  - Leading hand clears *Accessed* bits
  - Trailing hand evicts *Pages* with *Accessed*==0
- Can also take advantage of Hardware *Dirty* bit
  - Each *Page* either (*Unaccessed*, *Clean*), (*Unaccessed*, *Dirty*), (*Accessed*, *Clean*), or (*Accessed*, *Dirty*)
  - Consider *Clean Pages* for *Eviction* before *Dirty* ones
- Or use *n-bit Accessed* count instead just 1 *Accessed* bit
  - On sweep: `count = (A << (n - 1)) | (count >> 1)`
  - Evict *Page* with lowest `count`



Clock Algorithm (continued)

- Large *Memory* may be a problem
  - Long intervals between re-checking *Pages*
- Add a second *Clock* hand
  - The 2 hands move in lockstep
  - Leading hand clears *Accessed* bits
  - Trailing hand evicts *Pages* with *Accessed*==0
- Can also take advantage of Hardware *Dirty* bit
  - Each *Page* either (*Unaccessed*, *Clean*), (*Unaccessed*, *Dirty*), (*Accessed*, *Clean*), or (*Accessed*, *Dirty*)
  - Consider *Clean Pages* for *Eviction* before *Dirty* ones
- Or use *n-bit Accessed* count instead just 1 *Accessed* bit
  - On sweep: `count = (A << (n - 1)) | (count >> 1)`
  - Evict *Page* with lowest `count`



Other *Page Replacement* Algorithms

Random Eviction

- Dirt-simple to implement
- Not overly horrible (avoids *Belady's Anomaly* & pathological cases)

Least Frequently Used (LFU)

- Instead of just *Accessed* bit, count # of times each *Page* accessed
- Decay usage counts over time (for *Pages* that fall out of usage)

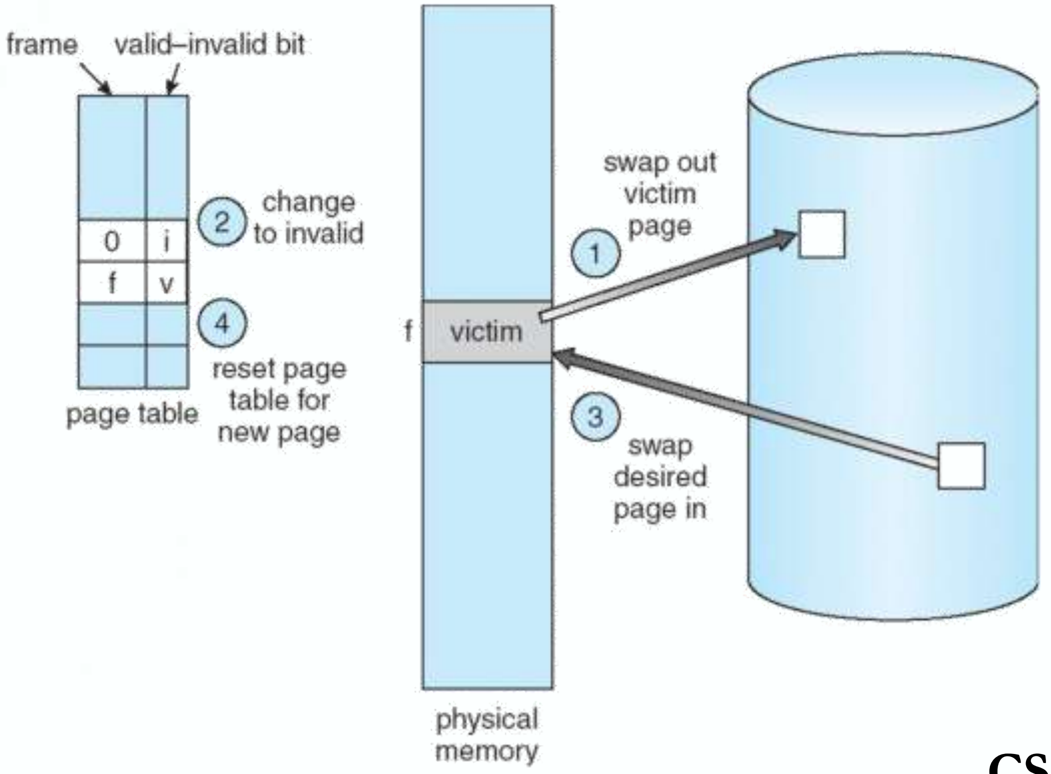
Most Frequently Used (MFU)

- Rationale: *Page* with smallest count was probably just brought in and has yet to be used
- Neither LFU nor MFU used very commonly

Paging Methods

Naïve Page Replacement:

- 2 Disk I/Os per *Page Fault*



***Paging* Methods**

Page Buffering

- Idea: Reduce # of I/Os on the critical path
- Use "*Free Pool*" – keep a Pool of *Free Page Frames*
  - On *Page Fault*, still select victim *Page* to evict
  - But read newly fetched *Page* into an **already** *Free Page Frame* (from the kept *Free Pool*)
  - Can resume execution while writing-out victim *Page*
  - When done writing-out victim *Page*, add it to *Free Pool*
- Allows to also yank *Pages* back from *Free Pool*
  - Contains only *Clean Pages*, but may still have their data
  - If *Page Faults* on a *Page* that is still in the *Free Pool*, recycle it

Fixed **vs** ***Variable Space***

How to determine how much *Memory* to allow for each *Process*?

Fixed Space Algorithms

- Each *Process* is given a limit of *Pages* it can use
- When it reaches the limit, it replaces from its own *Pages*
- *Local Replacement* Policy
  - Some *Processes* may do well while others suffer

Variable Space Algorithms

- Each *Process*' set of *Pages* grows and shrinks dynamically
- *Global Replacement* Policy
  - One *Process* can ruin it for the rest

Working Set **Model**

- A *Working Set* of a *Process* is used to model the *Dynamic Locality* of its *Memory* usage
  - Defined by Peter Denning in 60s, published at the first SOSP Conference

Definition

- $WS(t, w)$ = {*Pages* P such that P was referenced in the time interval (t-w, t)}
- t: time, w: *Working Set* window (measured in *Page Refs*)

- I.e. a *Page* is in the *Working Set (WS)* only if it was referenced inside the last w *Page References*

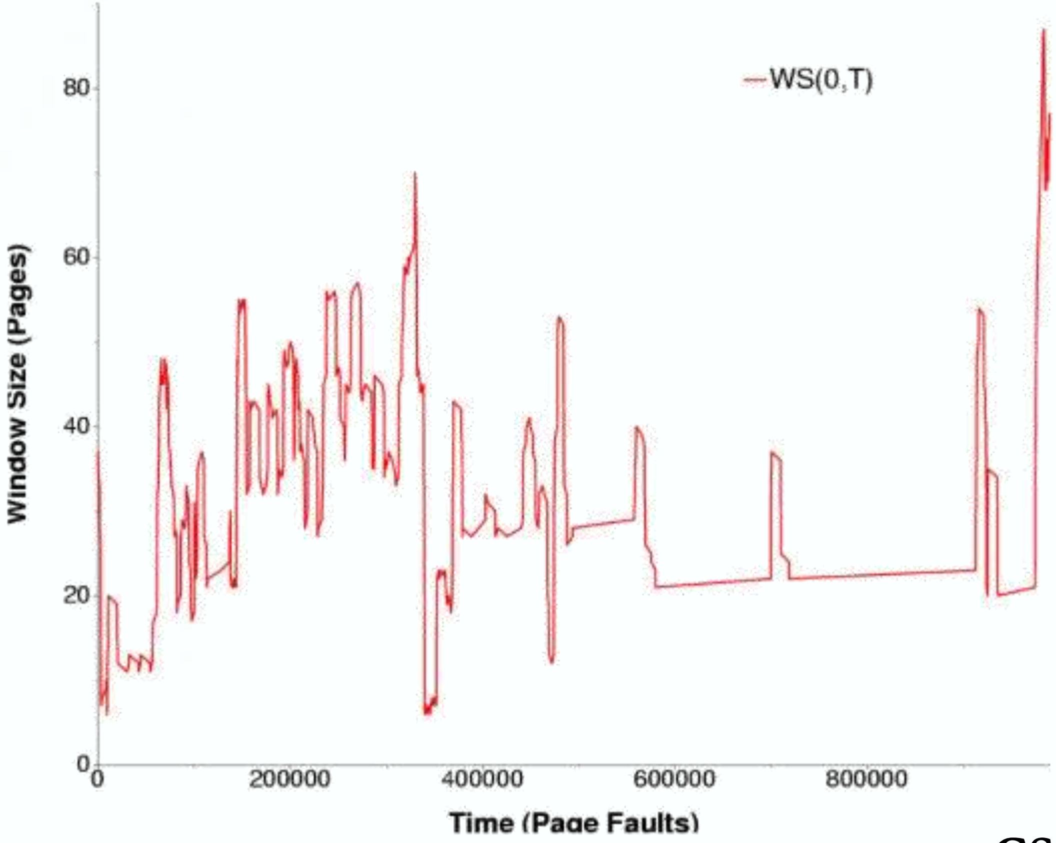
Working Set Size

Definition

- The # of **unique** *Pages* in the *Process' Working Set*
  - The number of **unique** (grows, shrinks) *Pages* referenced in the interval $(t, t - w)$
- The *Working Set Size* changes with program *Locality*
  - During periods of poor *Locality*, you reference more unique *Pages*
  - Within that period of time, the *Working Set Size* is larger
- Intuitively, want the *Working Set* to be the set of *Pages* a *Process* needs in *Memory* to prevent heavy *Faulting*
  - Each *Process* has a param w that determines a *Working Set* with few *Faults*
  - Denning: Don't run a *Process* unless its *Working Set* **exists/is restored** in *Memory*

Working Set Size

Example: **gcc** *Working Set*



Working Set **Problems**

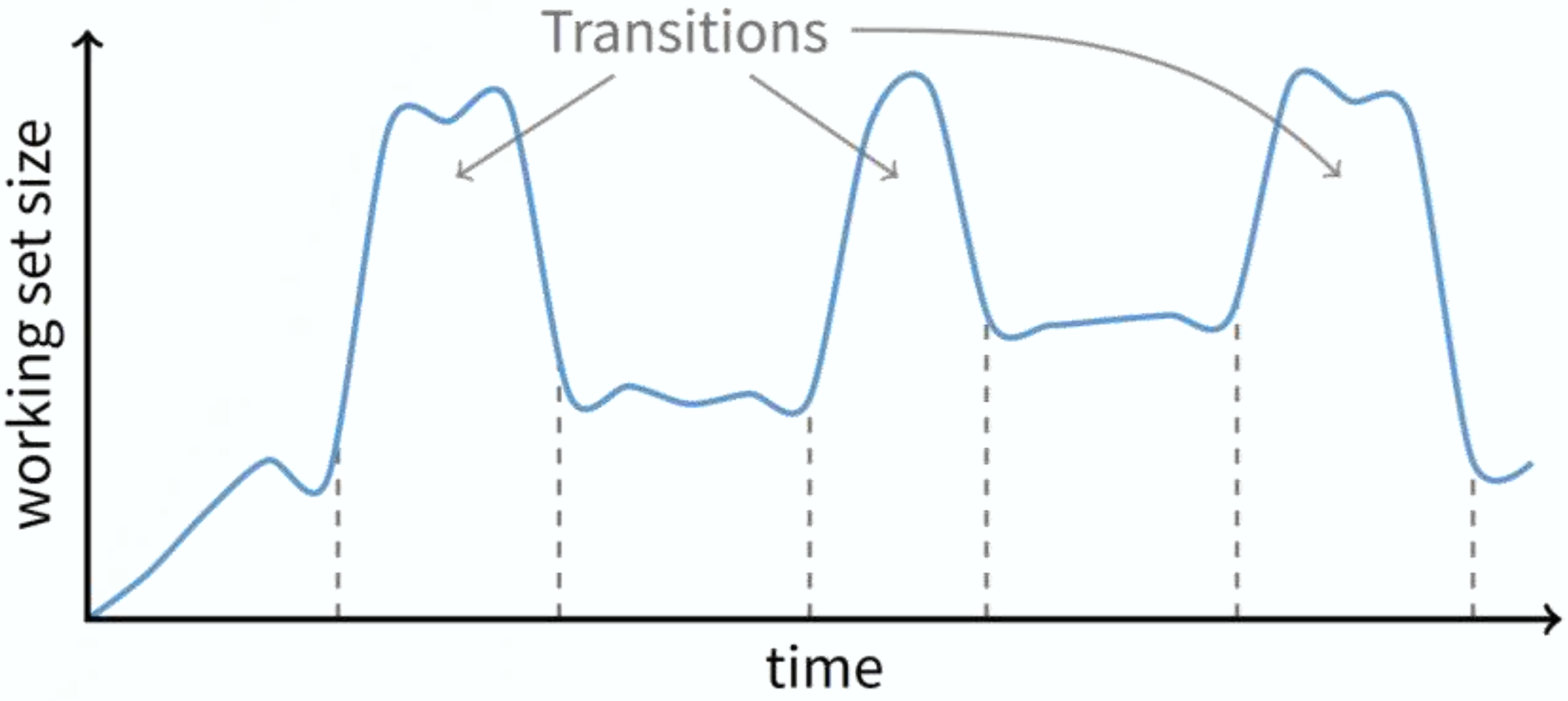
Problems

- How do we determine w ?
- How do we know when the *Process' Working Set* changes, i.e. undergoes a *"Phase Transition"* ?

- Too hard to answer
  - So, *Working Set* is not used in practice as a *Page Replacement* Algorithm

- However, it is still used as an abstraction
  - The intuition is still valid
  - When people ask, "How much *Memory* does Firefox need?", they are in effect asking for Firefox's *Working Set Size*

Working Set Changes across Phases



- *Working Set Size* balloons across *Phase* transitions

Directly Calculating the *Working Set*

Working Set : All **unique** *Pages* that *Process* will access in **next** t time

- Can't calculate without predicting the future

- "Estimate" it by assuming past predicts future
  - Same principle as LRU Clock Algorithm (but now keep track of time)
    - So *Working Set* ≃ **unique** *Pages* accessed during **last** t time
- Keep track of an "*Idle Time*" **for each** *Page*
- Periodically scan all *Resident Pages* in system
  - Is *Accessed* bit set? Clear it and clear the *Page*'s *Idle Time*
  - Is *Accessed* bit clear? Add CPU consumed since last scan to the *Page*'s *Idle Time*
  - *Working Set* is *Pages* with *Idle Time* < t

An "Indirect" Approach: *Page Fault Frequency (PFF)*

- *Page Fault Frequency* is a *Variable Space* Algorithm (to dynamically determine how many *Pages* of *Memory* are allowed to a *Process*) with a more ad-hoc approach

Definition

- *Page Fault Frequency (PFF) = Page Faults / Instructions* executed
  - Monitor the *Fault Rate* for each *Process*
  - If the *Fault Rate* is above a high threshold, give it more *Memory*
    - So that it *Faults* less (but not always – e.g. FIFO, *Belady's Anomaly*)
  - If the *Fault Rate* is below a low threshold, take away *Memory*
    - Expected to lead to more *Faults* (but not always)

- But! Hard to use *Page Fault Frequency* to distinguish between changes in *Locality* and changes in *Working Set Size*

Thrashing

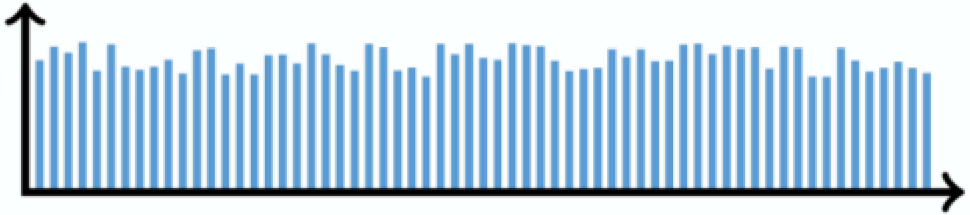
- *Page Replacement* Algorithms avoid the problem of *Thrashing*

Thrashing

- When OS spends most of its time *Paging* data back and forth to Disk
- Little time spent doing useful work (*Process* progress)
- In this situation, the system is *Overcommitted*
  - OS has no idea which *Pages* should be in *Memory* to reduce reoccurring *Faults*

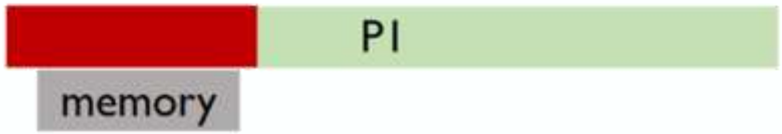
Reasons for *Thrashing*

- Access pattern has no *Temporal Locality*
  - past ≉ future

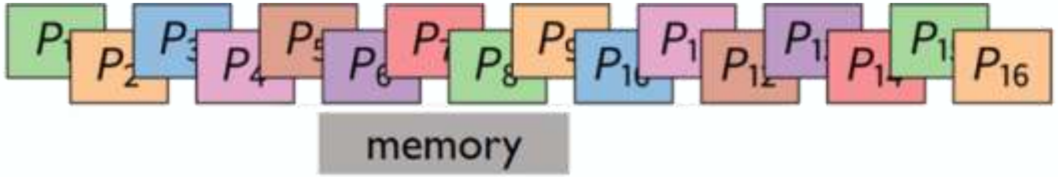


80/20 Rule has broken

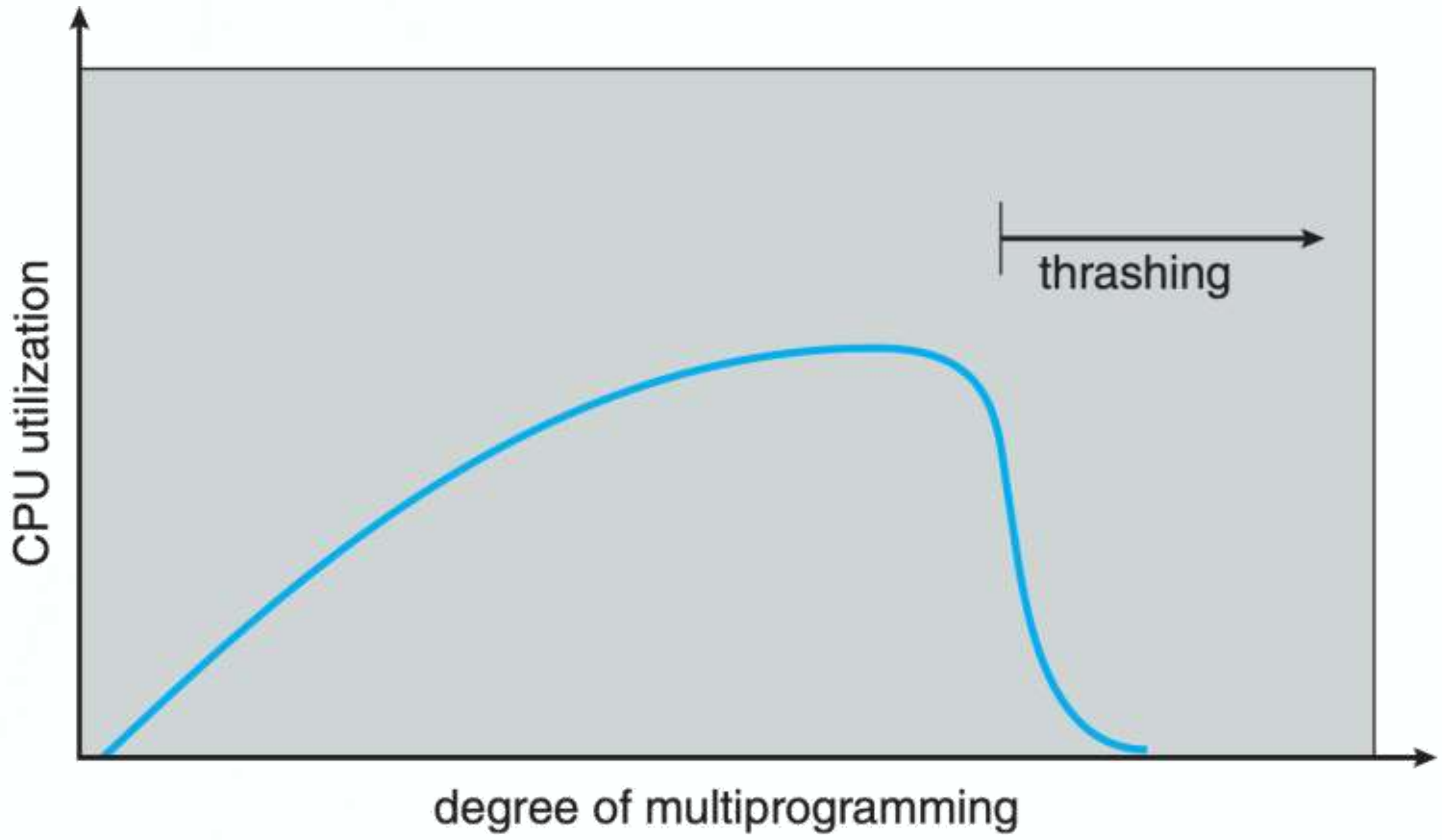
- *Hot* Memory does not fit in *Physical Memory*



- Each *Process* fits individually, but too many for system



Thrashing & Multiprogramming



Dealing with *Thrashing*

- Approach 1: *Working Set*
  - *Thrashing* viewed from a caching perspective: Given *Locality* of References, how big a cache does the *Process* need?
  - I.e. how much *Memory* does the *Process* need in order to make reasonable progress (its *Working Set*)?
  - Only run *Processes* whose *Memory* requirements can be satisfied
- Approach 2: *Page Fault Frequency (Remember:* PFF = *Page Faults / Instructions* executed*)*
  - *Thrashing* viewed as poor ratio of fetching –to– actual work done
  - If *Page Fault Frequency* rises above a high threshold, *Process* needs more *Memory*
    - If not enough *Memory* on the system? Swap out some of its *Pages*
  - If *Page Fault Frequency* sinks below a low threshold, *Memory* can be taken away

Two-Level Scheduler

Divide *Processes* into *Active* & *Inactive*

- *Active* : Means *Process' Working Set* is *Resident* in *Memory*
- *Inactive* : Means *Process' Working Set* is intentionally not loaded
- *Balance Set* : The union of all *Active Working Sets*
  - Goal: Keep *Balance Set* smaller than *Physical Memory*
- Use "*Long-Term*" *Scheduler*
  - Moves *Processes* from *Active Set* → *Inactive Set* until *Balance Set* becomes small enough
  - Periodically allows *Inactive Processes* to become *Active*
  - As *Working Sets* change, must also update *Balance Set*
- Complications
  - How to chose *Idle Time* threshold `t` for *Working Set* calculation?
  - How to pick which *Processes* will be in the *Active Set*
  - How to count *Shared Memory* accesses (e.g. `libc.so`)

OS & Paging

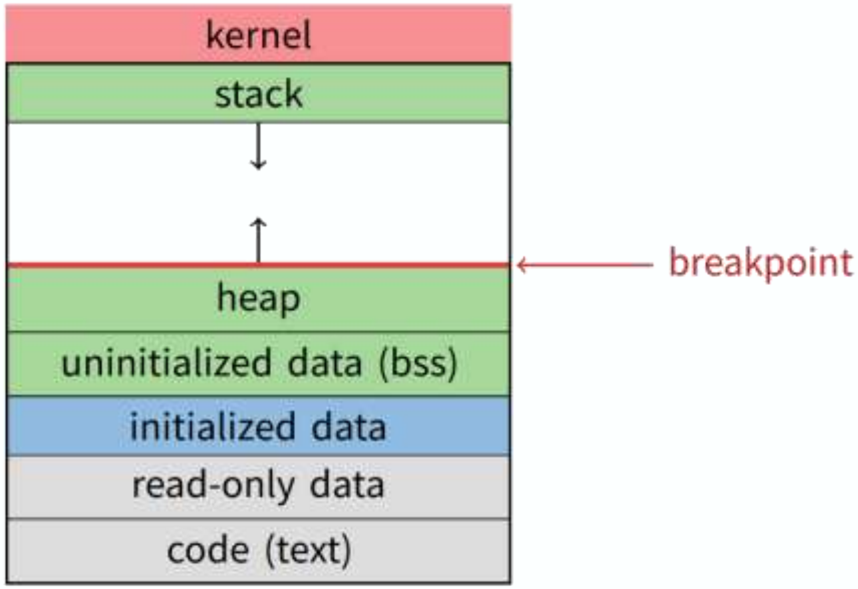
Complications of *Paging*

- What happens to available *Memory*?
  - Some *Physical Memory* remains tied up by Kernel *Virtual Memory* structures
- What happens to *User/Kernel-Level* crossings?
  - More crossings into *Kernel-Level* required (to handle "*Paging-In/Out*")
  - Pointers in *System Call* arguments must be checked (for ***Security*** & ***Reliability***)
    - Also, obviously can't just kill a *Process* if *Page* is not present – Might need to "*Page-it-In*"
- What happens to *Inter-Process Communication* (IPC) ?
  - Must apply changes to Hardware *Address Space* (*Remember: IPC* through *Memory-Mapped Files*)
  - Must change over to other *Process' Virtual Memory Mappings* – Increases TLB *Misses*
    - *Context Switch* flushes TLB entirely on old x86 machines (on each `%CR3` write)
      - But not on MIPS – *Remember*: Flexible *Software-managed* TLB
        - MIPS tags TLB entries with *Process Context IDentifier (PCID)*

The *User-Level* Perspective

Remember: Typical *Virtual Address Space*

- Dynamically Allocated *Memory* goes in Heap
  - Top of Heap called the "*Breakpoint*"
  - Addresses between *Breakpoint* and *Stack* are *Invalid* (almost all – see **`mmap`** later)



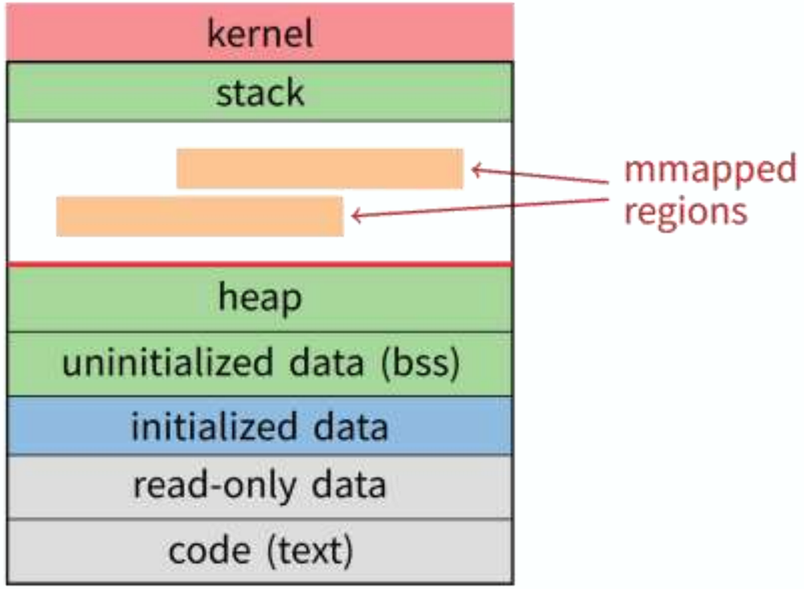
Note: In reality, *Linear Virtual Addresses* of Stack, Heap (and some other sections e.g. between *Shared Libraries* and Shared Libraries and main *Program*) are separated by huge (remember, *Address Space* is *Virtual*) "*Guard*" *Regions* that are permanently unmapped

- Attempt of e.g. Stack / Heap to grow into that *Guard Region* causes a *Protection Fault* → *Segmentation Fault*

The *User-Level* Perspective

Memory-Mapped Files

➢ Other *Memory* objects may be placed between the Heap and the Stack *Virtual Memory Address* regions



The *User-Level* Perspective

The **`mmap()`** *System Call*

```
void *mmap (void *addr, size_t len, int prot,
            int flags, int fd, off_t offset);
```

- Map *File* specified by **`fd`** at *Virtual Address* **`addr`**
  - If **`addr`** is null, let Kernel choose the *Virtual Address*
- **`prot`** : Protection of region
  - Binary OR of: **`PROT_EXEC`** (can be used to store instructions), **`PROT_READ`**, **`PROT_WRITE`**, **`PROT_NONE`** (reserved –e.g. for future use– with no access allowed)
- **`flags`**
  - **`MAP_ANON`** : *Anonymous Memory – Non-File-Backed* (**`fd`** should be -1)
  - **`MAP_PRIVATE`** : Modifications are private
  - **`MAP_SHARED`** : Modifications seen by everyone

The *User-Level* Perspective

More *Virtual Memory System Calls*

```
int msync(void *addr, size_t len, int flags);
```

- Flush changes of *Memory-Mapped File* to Backing Store

```
int munmap(void *addr, size_t len)
```

- Removes *Memory-Mapped* object

```
int mprotect(void *addr, size_t len, int prot)
```

- Changes *Protection* on *Process' Virtual Memory* address range (`PROT_`. . .)

```
int mincore(void *addr, size_t len, char *vec)
```

- Returns `vec` which *Process' Virtual Memory* address range *Pages* are *Present* in Memory

The *User-Level* Perspective

Exposing information of *Page Faults*

```
struct sigaction {
  union { /* signal handler */
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
  };
  sigset_t sa_mask; /* signal mask to apply */
  int sa_flags;
};

int sigaction (int sig,
               const struct sigaction *act,
               struct sigaction *oact);
```

- E.g. can specify callback function to run on **SIGSEGV**
  - Unix *Signal* raised on *Invalid Memory* access

The *User-Level* Perspective

Exposing information of *Page Faults*

Example: OpenBSD/i386 **siginfo**

```
struct sigcontext {
  int sc_gs; int sc_fs; int sc_es; int sc_ds;
  int sc_edi; int sc_esi; int sc_ebp; int sc_ebx;
  int sc_edx; int sc_ecx; int sc_eax;

  int sc_eip; int sc_cs; /* instruction pointer */
  int sc_eflags; /* condition codes, etc. */
  int sc_esp; int sc_ss; /* stack pointer */

  int sc_onstack; /* sigstack state to restore */
  int sc_mask; /* signal mask to restore */

  int sc_trapno;
  int sc_err;
};
```

- Linux uses **ucontext_t** – same idea, just uses nested structures that don't all fit on one slide

The *User-Level* Perspective

User-Level Virtual Memory "Tricks"

Combination of **`mprotect()`**/**`sigaction()`** very powerful

- e.g. *Fault*, *Unprotect Page* (via *User-Space* available *System Call*), return from *Signal Handler*

- Technique used in Object-Oriented Databases
  - Bring in objects on demand
  - Keep track of which objects may be *Dirty*
  - *Memory* is managed and acts as a cache for a much larger Object Database

- Other interesting applications
  - Some *Garbage Collection* Algorithms
  - Efficient snapshots of *Processes* (*Copy-on-Write*)

CS-446/646

Time for Questions !