New Subtopic: Process Scheduling

- Multiprogramming permits us to run multiple jobs (processes)

- More precisely, it permits us to run those jobs that are READY to run (i.e. are queued on the READY QUEUE to wait to be executed):
  - a process is “ready” if:
    - memory has been allocated
    - process is not waiting for I/O
    - process is not waiting for some semaphore or some critical code

- When one job has to wait, the OS switches CPU to execute some other READY job.

- Two general types of processes:
  - CPU-bound: <def?>
  - I/O-bound: <def?>
Process Scheduling (Generic)

TERMS:
- "inswapped": job in memory
- "outswapped": job’s executable image stored in OS-owned area of hard disk

Multiprogramming Example

The benefit of multiprogramming is to increase CPU utilization and job throughput!

Scheduling Possibility 1

R1 W1 R1
R1 R2 R2 R2 W2 R2 R2

Scheduling Possibility 2

R2 R2 R2 W2 R2 R2 R1 W1 R1

Scheduling Possibility 3

R1 R2 R1 R2 W2 R2 R2

Scheduling Possibility 4

R2 R2 R2 R1 R2 R2 R1
Scheduling CPU Access

- The (short-term) SCHEDULER is the part of the OS concerned with the decision of which job out of all READY jobs should be given the CPU next.

- SCHEDULER’s goals:
  - Fairness
  - Efficiency
  - Turnaround Time
  - Wait Time
  - Response Time
  - Throughput

- 2 types of scheduling environments:
  - Deterministic Scheduling (Job Shop Scheduling)
  - Stochastic Scheduling (Queueing Theory, which we’re not covering)

Example Revisited

<table>
<thead>
<tr>
<th>JOB 1</th>
<th>JOB 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>W1</td>
<td>R2</td>
</tr>
<tr>
<td>R1</td>
<td>W2</td>
</tr>
<tr>
<td></td>
<td>R2</td>
</tr>
</tbody>
</table>

Scheduling Possibility 1

| R1 | W1 | R1 | R2 | R2 | W2 | R2 | R2 |

Scheduling Possibility 2

| R2 | R2 | R2 | W2 | R2 | R2 | R1 | W1 |

Scheduling Possibility 3

| R1 | R2 | R1 | R2 | R2 | W2 | R2 | R2 |

Scheduling Possibility 4

| R2 | R2 | R2 | R1 | R2 | R2 | R1 |
Common Scheduling Policies

- First-Come First Served (FCFS or FIFO)
- Shortest Job First
- Round Robin
- Priority Queues
- MultiLevel Feedback Queues

First-Come First Served

- Schedule processes in the order in which they request service
- example:

<table>
<thead>
<tr>
<th>JOB#</th>
<th>Request Order</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Turnaround Time =

Response Time =

Wait Time =

Countermeasures?

Problems?
**First-Come First Served**

- Consider FCFS in dynamic situation: 1 CPU-bound Process, many I/O bound

- What happens?
  - CPU-bound grabs CPU and holds it
  - during this time, I/O-bound processes finish I/O and move into READY queue.
  - while they wait, I/O devices are idle!
  - CPU-bound finishes, and moves to an I/O request
  - I/O-bound finish with CPU quickly
  - CPU idle!
  - CPU-bound finishes I/O, goes back to CPU hogging

- This is a “Convoy Effect”

**Shortest Job First (SJF)**

- LCFS, FCFS do not take job times into account, so let’s have CPU execute jobs with smallest CPU demands first

<table>
<thead>
<tr>
<th>JOB#</th>
<th>Request Order</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Turnaround Time =  
Response Time =  
Wait Time =

Countermeasures?

Problems?
Shortest Job First

• If we don’t know the exact CPU service time a job may require, maybe we can ESTIMATE it:
  – use exponential averaging: (aging)
  – \( T(n) = \text{“estimate” of the duration of the nth service time} \)
  – \( t(n) = \text{actual value of nth service time} \)

• Define: \( T(n+1) = at(n) + (1-a)T(n) \), assume \( 0 \leq a \leq 1 \).
  – \( T(1) = at(0) + (1-a)T(0) \)
  – \( T(2) = at(1) + (1-a)T(1) = at(1) + (1-a)[at(0) + (1-a)T(0)] \)
    = \( at(1) + (1-a)at(0) + ((1-a)^2)T(0) \)
  – etc.

• Question: what is the effect of “a”?

Scheduling Categories

• “Run to Completion”
  – Benefits?
  – Cons?

• “Pre-emptive Scheduling”
  – Benefits?
  – Cons?
  – Round Robin, Shortest Remaining Time
  – Priority Queues,
  – Multilevel Feedback Queues
Round Robin Scheduling

- A small time quantum “q” is defined.
- CPU is allocated to a job “i”.
- If job finishes service before quantum expires, then leave the CPU Q.
  else go to end of CPU Q when quantum expires.

Note: $q \rightarrow \infty$, RR becomes FCFS.
$q \rightarrow 0$, RR becomes “processor sharing”; each of n jobs sees CPU executing at $1/n$ speed.

Round Robin

- Example:

<table>
<thead>
<tr>
<th>JOB#</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

$q = 2$

Turnaround Time = 
Response Time = 
Wait Time =

Countermeasures?

Problems?
**Shortest Remaining Time**

- Preemptive version of SPN
  - CHOOSE JOB WITH SHORTEST EXPECTED REMAINING PROCESSING TIME
- Can preempt whenever a new (shorter) job arrives
- no bias toward long-term jobs getting “better” response times (unlike FCFS)
- Better turnaround (wait time) compared with SPN

---

**Priority Queues (multi-level queues)**

- Maintain multiple queues for different “priority-class” jobs.

  HI  
  Faculty  
  System  

  LOW  
  Students

  Jobs from a lower-priority queue are served
  IFF
  there are no jobs in a higher-level queue
Multilevel Feedback

- Similar idea to priority queues, except jobs move DYNAMICALLY from higher-level queues down to lowest level queues
- Effect: ?
- Each queue can have a different scheduling policy!
Race Conditions

- Asynchronous concurrent processes/threads can access shared data in an ARBITRARILY interleaved manner!

Example:

Two Processes:

Insert(Y, Q)
1. Get last Q element
2. set last element’s ptr to Y

Delete_Last(Q)
A. Get ptr to last Q element
B. set ptr to NIL

Statement Interleavings?

Avoiding Race Conditions

- When can two statements be executed concurrently (arbitrarily interleaved) and produce the same results? (and avoid race conditions)

- Def: R(S) = set of variables referenced by “S” (Read Set)
- Def: W(S) = set of variables updated by “S” (Write Set)

Example:

- S1: a = b - c; R(S1) = {b,c} W(S1) = {a}
- S2: d = a * 7; R(S2) = {a} W(S2) = {d}

Answer: “Bernstein’s Conditions”

- R(S1) ∩ W(S2) = Ø
- R(S2) ∩ W(S1) = Ø
- W(S1) ∩ W(S2) = Ø
**Critical Sections**

- **Defn:** A section of code which modifies/accesses data structures shared among several processes
- **Problem:** arbitrary interleaving of executions may result in undefined or unexpected results
  - shared code that is immune to critical section problem is called “re-entrant,” or “thread-safe”
- **DISCUSS:** How can we solve “critical section problem?”

**Solving Race Conditions**

- Any proposed method for preventing race conditions must enforce the following criteria:
  - **MUTUAL EXCLUSION:** only 1 process/thread in critical section at a time
  - **PROGRESS:** processes/threads in NONcritical portions of code cannot prevent others from entering a critical section
  - **BOUNDED WAITING:** it must be impossible for a process/thread to wait forever to enter its critical section.
- **NB:** solutions should not make assumptions about process execution speeds.
public class Fisherman extends Thread {
    private int id;
    private static final int TIME = 14400;
    public Fisherman(int i) {
        id = i;
    }
    public static void go_fishing() {
        try {
            Thread.sleep((int) (Math.random() * TIME));
        } catch (InterruptedException e) { }
    }
    public static void work_off_ice() {
        try {
            Thread.sleep((int) (Math.random() * TIME));
        } catch (InterruptedException e) { }
    }
    public void run() {
        while (true) {
            go_fishing();
            work_off_ice();
        }
    }
}
"Main" Method

```java
public class FishingProblem {
    /*Globally shared variables here (see later slides)*/
    public static void main() {
        Fisherman first = new Fisherman(0);
        Fisherman second = new Fisherman(1);
        first.start();
        second.start();
    }
}
```

Attempt 1

```java
public volatile int turn = 1;

public void run() {
    while (true) {
        while (turn != 0) Thread.yield();
        go_on_ice();
        turn = 1;
        work_off_ice();
    }
}
```

Problems?
Scenario for attempts 2,3,4

DANGER!
Thin Ice

ICE1
ICE2

Attempt 2

```java
public volatile boolean[] ice = new boolean[2];
ice[0] = false;
ice[1] = false;

public void run() {
    while (true) {
        while (ice[0] == true) Thread.yield();
        ice[0] = true;
        go_on_ice();
        ice[0] = false;
        work_off_ice();
    }
}
```

Problems?
**Attempt 3**

```java
public volatile boolean[] ice = new boolean[2];

ice[0] = false;
ice[1] = false;

public void run() {
  while (true) {
    ice[0] = true;
    while (ice[1] == true) Thread.yield ( );
    go_on_ice ( );
    ice[0] = false;
    work_off_ice ( );
  }
}
```

Problems?

**Attempt 4**

```java
public volatile boolean[] ice = new boolean[2];

ice[0] = false;
ice[1] = false;

public void run() {
  while (true) {
    ice[1] = true;
    while (ice[0] == true) {
      ice[1] = false; Thread.yield ( );
      ice[1] = true; Thread.yield ( );
    }
    go_on_ice ( );
    ice[1] = false;
    work_off_ice ( );
  }
}
```

Problems?
Attempt 5 Scenario

DANGER!
Thin Ice

ICE1

Dekker’s Solution

public volatile boolean[] ice = new boolean[2];
public volatile int turn = 0;/* or 1 */

ice[0] = false;
ice[1] = false;

public void run() {
    while (true) {
        ice[0] = true;
        turn = 1;
        while (((ice[1] == true) &&
                (turn == 1)) Thread.yield ( ));
        go_on_ice();
        ice[0] = false;
        work_off_ice();
    }
}

Problems?

public void run() {
    while (true) {
        ice[1] = true;
        turn = 0;
        while (((ice[0] == true) &&
                (turn == 0)) Thread.yield ( ));
        go_on_ice();
        ice[1] = false;
        work_off_ice();
    }
}
Alternate Solutions to Race Conditions

• **Disabling Interrupts?**
  - Yes, but....

• **Test-and-Set Instruction**
  - single, atomic, uninterruptable instruction
  - test & possibly modify a variable in a single step
  - eg TSL on IBM 360:
    • test-and-set-lock(lock) returns true indicating the lock has already been
      set by some other process,
    • OR returns false, indicating the lock had not been set, BUT NOW IS SET
      BY CURRENT PROCESS/THREAD.
  - Example use:
    
    ```
    REPEAT
      WHILE (test-and-set(lock) == true)
        do /*nothing */
      run_critical_section;
    lock := false;
    FOREVER;
    ```

“Blocking” Solutions

• Avoid busy waiting by getting OS to “block,” or stop and
  swap out, any processes/threads that are just waiting for
  their critical section

• Semaphores to the “rescue!” (Dijkstra, 1966)

• Defn: a Semaphore is an integer variable that can only be
  accessed through the ATOMIC operations DOWN and UP:
  - DOWN(S): (if (S <= 0) BLOCK;
    S := S - 1;)
  - UP(S): (S := S+1;
    WAKEUP one process (if any) blocked on S;)

• “General Semaphore” (counting semaphore) - S can take on
  any value
Assume:
P1 has a DOWN(x)
P3 has a UP(x)

Eventually, this state is reached.

P1 encounters DOWN(x) and BLOCKS on x.

“Eventually,” this state is reached.

P3 encounters UP(x) and issues “wakeup” on x.
Using Semaphores (Mutual Exclusion)

Solving the Critical Section Problem:

```plaintext
REPEAT
  x.down()
  critical_section;
  x.up()
  noncritical_section;
FOREVER;
```

Using Semaphores (Synchronization)

- Suppose statement S1 must be executed by process P1 BEFORE statement S2 is executed by P2?

P1:
```
BEGIN
A1;
A2;
  S1;
  x.up();
A3;
END;
```

P2:
```
BEGIN
B1;
  x.down();
S2;
  B2;
END;
```
Implementing Semaphores

- Atomicity of semaphores is crucial to their use. How might we guarantee this in their implementation?

- Solution #1:

  Problems?:

- Solution #2:

Java and Semaphores

```java
public class Semaphore {
    private int s;

    public void Semaphore (int initial_val) {
        s = initial_val;
    }

    public void Semaphore ( ) {
        s = 1;
    }

    public synchronized void down ( ) {
        if (s <= 0)
            wait ( );
        s--;
    }

    public synchronized void up ( ) {
        s++;
        notify ( );
    }
}
```
The Java “synchronized” declaration

• Every Java object has a “lock”
  – If object method is “synchronized,” the thread of the entity invoking that method must own the object’s lock
  – If lock owned by another entity, thread is blocked & put in “entry set” for lock
  – When a thread exits a synchronized method, it gives up the lock to the next thread in line in the lock’s “entry set.”

• Every Java object has a “wait set”
  – If a thread determines that it cannot make more progress, it can place itself in an object’s wait set by invoking wait()
    • Thread releases object’s lock
    • Thread state is set to BLOCKED
    • Thread is enqueued in wait set
  – Some other thread can invoke notify() method to “wake up” a waiting thread
    • “Arbitrary” thread T from wait set is selected
    • T is moved from wait set to entry set
    • T’s state is changed to RUNNABLE
New Topic: Memory Management

• Basic machine architecture, from storage point of view

- CPU
- Memory
- I/O
- DMA
- Cache: L2, instruction, etc.
- "RAM"
- Hard Disk, ZIP Drive, etc.
**Memory Management Intro**

- ISSUE: there is a finite amount of storage in a computer, arranged in the following hierarchy:
  - Processor memory (e.g. RAM)
  - Secondary storage (e.g. hard drive)
  - Tertiary storage (e.g. ZIP, tape)

**Processor memory** (like processor time) is a **scarce resource** that must be managed.

-- which areas free/used (by whom)
-- decide which processes get memory
-- perform allocation/deallocation

!!!--> need for interaction between CPU allocator & memory allocator

**Physical Memory Allocation Methods**

- Single User Contiguous Allocation
- Fixed Partitions
- Dynamic Partitions
Single User Contiguous Allocation

- “a” is kept in a BASE REGISTER
- “b” is kept in a UPPER-BOUND REGISTER

The single user is allocated as much memory as needed

Problems?

Single User Contiguous Allocation

- OVERLAYS as the solution to size restrictions

Hardware support
  --boundary registers
  --user memory access >b or <a causes addressing exception interrupt

Problems?
Fixed Partitions (multiprogramming support)

- memory divided into fixed partitions
- a process is allocated an ENTIRE partition
- e.g. OS/360
- OS must maintain table:

<table>
<thead>
<tr>
<th>#</th>
<th>SIZE</th>
<th>LOCATION</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15 M</td>
<td>20M</td>
<td>used</td>
</tr>
<tr>
<td>2</td>
<td>20 M</td>
<td>35M</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>45 M</td>
<td>55M</td>
<td>used</td>
</tr>
<tr>
<td>4</td>
<td>55 M</td>
<td>110 M</td>
<td>used</td>
</tr>
</tbody>
</table>

Hardware Support: (protection mechanisms)
- bound registers
- each partition has “protection key” (must equal key in current PSW)

Disadvantage: Internal Fragmentation

Dynamic Partitions

- Partitions dynamically created during job processing
- OS tables record free or used portions of memory
- Linked lists for FREE space
- Example:

```
0x00000000  request: 4MB  4 MB  4 MB  4 MB  4 MB
0x00000000

0x00FFFFFF  request: 8MB  8 MB  8 MB  8 MB
0x00FFFFFF

0x00000000  release: 4MB  4 MB
0x00000000

0x00FFFFFF  request: 8MB  8 MB
0x00FFFFFF
```
Dynamic Partitioning (continued)

• Advantages:
  – eliminates fragmentation
  – can allocate more partitions/processes
  – higher degree of multiprogramming, therefore better memory and processor utilization

• Disadvantages?
  –
  –
  –
  –

Allocation Policies

• First Fit: find free portion with lowest address that can hold the process
  – advantage: large chunks of memory form in high memory locations

• Best Fit: find smallest portion that can hold the process
  – advantage: leave larger chunks of memory available
    – costs:
      •
      •

• Worst Fit: find largest free portion, create new partition for process, leaving a large piece leftover as another free portion.
  – advantages?
  – costs?
Fragmentation

- “Internal Fragmentation” is used memory within partitions
- “External Fragmentation” occurs as small chunks of memory accumulate as a byproduct of partitioning due to imperfect fits.

- solution?
  - coalesce adjacent free areas
  - BURP the memory (compaction/recompaction)
    - combine all free areas into one contiguous region
    - this approach requires user processes to be RELOCATABLE

Virtual Memory: Logical vs. Physical Addresses

- A KEY OS CONCEPT! (make sure you understand this)
- Physical address: memory address as “seen” by memory (i.e., the “real” address in memory)
- Logical address: programmer/code (i.e. your “a.out” file) sees logical addresses 0...progsise.
  - physical location of program in memory is transparent.

- Requires HARDWARE support:
  -- base register
  -- hardware to do logical -> physical address translation
Virtual Memory: Address Translation

• Hardware support
  – base register: value set to address of first physical memory location of user process
  – bounds register: last physical memory location (for detecting illegal memory references and generating an interrupt)

• on EVERY user memory reference, contents of base relocation register are AUTOMATICALLY added to the “effective address” used in code to get the actual physical address

Non-Demand Paged Memory Management

• each process’s logical address space is divided into PAGES
• physical space is divided into pieces called BLOCKS or FRAMES
• page size == block size
• hardware provides a mapping facility from pages into blocks of physical memory
  – Page Map Tables (PMT)
• User sees logically contiguous virtual address space.
Page Map Tables

- stored in memory, cache, or special registers
- each entry in the table contains:
  - resident bit
  - read/write protected
  - dirty bit: has page been modified
  - storage address in secondary store if page is not in memory
  - page frame number (physical address of start of page)

Page Map Table

<table>
<thead>
<tr>
<th>R/W</th>
<th>resident bit</th>
<th>dirty</th>
<th>ss address</th>
<th>phys address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the currently executing process, the PMT is kept in a fixed location known only to the OS
- pointed to by page map table address register (PMTAR) in CPU

PMT’s for non-executing processes stored in each one’s PCB.
**Page Translation**

- **Direct Address Translation:**
  - Each memory reference generated by user process is divided into a page number (p) and offset (d).
  - Page number indexes into the process’ page map table to get the frame number of the corresponding block of memory.
  - Offset in page is the same as the offset in the block.

**Reasoning About Addressing**

- **Problem Specs**
  - NON-demand paging is used.
  - Hardware supports 32-bit addresses.
  - Blocks are 2KB in size.

- **Questions**
  - How many bits in a virtual address?
  - How many bits in the offset portion of a virtual address?
  - How many bits in the page # portion?
Non-Demand Paging Summary

Advantages?

Disadvantages?

Segmentation

- User/compiler given the ability to partition program into logical (semantically related) units, each with its own name, size, and protection attributes.

```
user view
  main
  GUI
  dynamic memory heap

paging view
  main
  GUI
  dynamic memory heap

segmentation view
```
**Segmentation**

- **Software Support**
  - PCB for process has a pointer to a SEGMENT MAP TABLE
  - just like PAGE MAP TABLE, except?
    - segment size must be added to each entry

- **Hardware support**
  - now, all memory references have a SEGMENT# and OFFSET instead of a page# and offset.

**Segmentation Discussion**

- **Advantages**
  - semantically related code remains single physical unit (simplifies protection!)
  - we get sharable segments

- **Problems?**
  - 
  - 
  - 

- **Combined Systems**
  - paged segments
  - segmented pages
  - address translation overhead
Demand Paging

• What if process image is too big for free memory?

• KEY: rather than swap ENTIRE image into memory, only
  swap in a page when needed (when is THAT?)
  — requires: keep unused portions of process’ image on “backing store”
  — requires: recovery mechanism for references to non-resident pages

• PAGE FAULT: occurs when program references a location in
  a non-resident page
  — another example of an interrupt!
  — pagefault interrupt handler activated by fault, initiates actions to swap in
    missing page
  — I/O interrupt handler now has to interact with the page fault mechanism and
    CPU scheduler!

Demand Paging

• Question: when a process either releases its memory
  (finishes) or must leave, where does it go?
  — Answer: swap space reserved for it on the secondary storage (disk)
    • Some OS’s reserve the space immediately when a process is created,
    • other OS’s reserve it only when the process needs to be swapped out.

• Considerations
  — max # of pages in memory per user?
  — how to replace (make room for) old/new pages?
  — how to reduce overhead?
Handling Large Page Map Tables

- Problem: What if page table gets too big?
- Solution 1: Multilevel page tables

- Solution 2: Inverted Page Tables

Virtual Address

\[ p \quad d1 \quad d \]

PMTAR

Physical Address

Hash Table

Inverted Page Table

Frame Start

Offset

Page #

Offset

Page #

Page Table Entry

Chain

Frame #
Speeding Up Page References

• Problem: With paging, all memory references now require at least an extra reference to a page table, with time drains on CPU.
  – effective access time = table access + physical memory access

• Solution: Translation LookAside Buffers!
  – A small table (between 256 & 8K entries) of entries for pages. (fast memory onboard CPU)
  – Each entry has:
    • virtual page number
    • protection bits
    • modified bit
    • physical address of page.
    • a “valid bit”
  – How used: when virtual address is presented to hardware, it first checks associative memory (all entries in parallel) to see if it already has the needed page address.

TLBs (continued)

• If page info is there, OK, otherwise retrieve page info from PMT, and copy info into an entry in TLB memory.

• This assumes that there is a small number of pages that a process typically works on at a time

• “hit ratio:” the percent of time we find a page’s info in TLB
  – LOCALITY of REFERENCE.
Locality of Reference

TLB Usage

Virtual Address
Page # offset

Translation Look-Aside Buffer

Main Memory
Secondary Storage

Page Table

TLB hit

Page is in memory

Frame Start offset

Real Address

Load page

Page Fault
Page Replacement Policies

• Terms
  – page fault rate = fraction of references that require a page to be swapped in.
  – reference string = sequence of memory addresses produced by a process

• How to Evaluate page-replacement policies?
  – need page-trace from reference string (only page# refs, not offsets)
  – need total # of page frames available
  – need page replacement alg.
  – GOAL: minimize page fault rate

Replacement Policies

• Belady’s Min
• FIFO
• Least Recently Used (LRU)
• Not Recently Used (NRU)
• Primary Example:

4 frames; 6 unique 100-word pages; 15 references

reference string:

page string:
4 1 6 4 2 3 4 6 1 6 4 6 5 3 2
Belady’s Min Page Replacement Algorithm

- From all filled-in frames, select the one page for replacement that will be requested FURTHEST in the FUTURE

- Example:

```
  4 1 6 4 2 3 4 6 1 6 4 6 5 3 2
  4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5
  1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2
  6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
  2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

Page fault rate?

\[
\frac{7}{15} = 0.4667
\]

Comments on Belady’s Min Algorithm

- Optimal

- Not implementable!
  - requires future knowledge
  - nevertheless, a good benchmark (minimum)

- Theoretical Questions
  - what is needed to determine a “good” minimum expected number of page faults?
    - # of frames
    - # of references
    - # of unique pages
  - what are upper & lower bounds on page faults for a given set of above numbers?
FIFO Page Replacement Algorithm

• Record order in which pages are brought into memory
• when a page must be replaced, select oldest page and swap it out

```
4 1 6 4 2 3 4 6 1 6 4 6 5 3 2
4 4 4 4 4 3 3 3 3 3 3 5 5 5
1 1 1 1 1 4 4 4 4 4 4 3 3
6 6 6 6 6 6 1 1 1 1 1 2
2 2 2 2 2 6 6 6 6 6 6 6
X X X X X X X X X X X
```

Page fault rate?

\[ \frac{11}{15} = 0.73 \]

Comments on FIFO

• Easy to understand and code
  — used in Win98 & NT
• Does not reflect fact that a page’s reference rate is independent of how long it is in memory
• Belady’s Anomaly: increasing the number of frames will NOT always decrease the page fault ratio...it may increase!
  — Example: 3 2 1 5 3 2 4 3 2 1 5 4 2 3 2 1 5 4
  — 2 frames: 17 page faults
  — 3 frames: 14 page faults
  — 4 frames: 15 (!) page faults
  — WHY? (& can you find other patterns?)
**LRU Page Replacement Algorithm**

- replace the page which has not been referenced for the longest time.

- Example

```
<table>
<thead>
<tr>
<th>4</th>
<th>1</th>
<th>6</th>
<th>4</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>6</th>
<th>1</th>
<th>6</th>
<th>4</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
```

Page fault rate?

$$\frac{9}{15} = 0.60$$

Hardware assistance:
- time stamp registers for each frame
- maintain stack of most recently referenced pages

---

**Not Recently Used Replacement Algorithm**

- Each page has two status bits:
  - R → “recently referenced” (page is read or written)
  - M → ”modified” (page is written)

- once a bit is set to 1, remains set until OS resets it.

- Periodically, all pages’ “R” reset, to indicate they haven’t been used recently

- remove a random page from the lowest-numbered class

```
<table>
<thead>
<tr>
<th>class</th>
<th>R</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
```

NRU approximates LRU time-stamping with a single bit.
Performance of Demand Paging

- Assume the following definitions:
  - \( M \) = Memory Access Time for primary memory
  - \( F \) = Page Fault Time
    - includes interrupt processing time and disk (I/O) time
  - \( P \) = Page Fault Rate (a percentage)

- Effective Access Time is \((1-P)M + PF\)

- Example

<table>
<thead>
<tr>
<th>( M = 10^{-6} )</th>
<th>Effective Access Time = (0.99 \times 10^{-6} + 0.01 \times 10^{-3} = 10^{-5})</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F = 10^{-3} )</td>
<td></td>
</tr>
<tr>
<td>( P = 0.01 )</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: page fault rate of 0.01 increases memory access time by a factor of 10!

Observation: for given \( M \) and \( F \), we need a \( P \) of 0.0001 to limit increase to 10%, instead of the 900% we see here!

Reducing Page Fault Costs

- Just as we want to reduce page translation overhead, we’d also like to reduce page-fault overhead!

- Methods:
  - Prefetching
  - Careful Frame Allocation
Prefetching

- While loading one page, also load in others that might be “predictably” requested later
- Windows 98 & NT use this: when page X is faulted, pages X-n, ..., X-1, X, X+1, ..., X+n are all loaded from swap space
- more loading, but cuts down on I/O and context-switching overhead

Allocating Frames to Processes

- Minimally, each process must get enough frames to hold ALL pages referenceable by a single instruction (how many?)
- Page Replacement
  - Local: no process can change number of frames allocated to it.
  - Global: a process can take more frames from others as needed.
- Allocating based on degree of multiprogramming (N)
  - **Equal Allocation:**
    - let “F” = total # of frames in memory, then \( \left\lfloor \frac{F}{N} \right\rfloor \) will be the number initially allocated
  - **Proportional Allocation:**
    - let \( S_x \) be size of virtual address space used by process x
    - let \( S = \sum S_x \)
    - allocate \( \left\lfloor \frac{S_x}{S} \right\rfloor \cdot F \) to process “x”
Thrashing

- A small allocation of frames means higher paging activity
- if allocation is too small
  - each process spends more time paging than executing
  - high OS overhead, little useful process work

Preventing Thrashing

- Remember LOCALITY?
- Working Set Model
  - assumes locality of execution
  - take D most recent page references (approx 10000)
  - the set of unique pages in D is current working set.
- If working set is too small, it won’t contain all current locality, leading to more frequent page faults
- If working set is too large, it contains more than current locality, leading to lots of pages kept in memory uselessly.
- USE:
  - a process is either allocated enough frames for all pages in working set, and marked READY, or allocated no frames and marked SUSPENDED
Considerations in Determining Page Sizes

- Observation: Page tables take up memory
  - THEREFORE, use large page sizes (need small table)

- Observation: When frames are allocated to a process, on average, 1/2 of the last frame is lost to internal fragmentation
  - THEREFORE, use small page sizes

- Observation: Small pages sizes mean more page faulting
  - THEREFORE, use large page sizes

- Observation: Swapping in a large page might mean we spend time bringing code outside of current locality (i.e. never referenced)
  - THEREFORE, use small page sizes
  - (actually, argues in favor of segmentation)

Demand Paging Summary

- Advantages
  - 
  - 
  - 
  - 
  - 
  - 

- Disadvantages