VII. Memory Management
# Intended Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Hand out</th>
<th>Submission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.04. Introduction to Operating Systems</td>
<td></td>
<td>Course registration</td>
</tr>
<tr>
<td>1</td>
<td>27.04. Systems Programming using C (File Subsystem)</td>
<td>1. Assignment</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>04.05. Systems Programming using C (Process Control)</td>
<td>2. Assignment</td>
<td>1. Assignment</td>
</tr>
<tr>
<td>3</td>
<td>11.05. Process Scheduling</td>
<td>3. Assignment</td>
<td>2. Assignment</td>
</tr>
<tr>
<td>4</td>
<td>18.05. Process Synchronization</td>
<td>4. Assignment</td>
<td>3. Assignment</td>
</tr>
<tr>
<td>5</td>
<td>25.05. Inter Process Communication</td>
<td>5. Assignment</td>
<td>4. Assignment</td>
</tr>
<tr>
<td>01.06.</td>
<td>Pfingstmontag</td>
<td>6. Assignment</td>
<td>5. Assignment</td>
</tr>
<tr>
<td>6</td>
<td>08.06. Deadlocks</td>
<td>7. Assignment</td>
<td>6. Assignment</td>
</tr>
<tr>
<td>7</td>
<td>15.06. Memory Management</td>
<td>8. Assignment</td>
<td>7. Assignment</td>
</tr>
<tr>
<td>8</td>
<td>22.06. Input / Output</td>
<td>9. Assignment</td>
<td>8. Assignment</td>
</tr>
<tr>
<td>9</td>
<td>29.06. Filesystems</td>
<td>10. Assignment</td>
<td>9. Assignment</td>
</tr>
<tr>
<td>10</td>
<td>06.07. Special subject: Transactional Memory</td>
<td></td>
<td>10. Assignment</td>
</tr>
<tr>
<td>11</td>
<td>13.07. Special subject: XQuery your Filesystem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>20.07. Wrap up session</td>
<td></td>
<td></td>
</tr>
<tr>
<td>27.07.</td>
<td>First examination date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.10.</td>
<td>Second examination date</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
History & Introduction

Programs expand to fill the memory available to hold them
(Parkinson’s Law, paraphrased)
Every program would like
- private,
- infinitely large,
- infinitely fast,
- non-volatile,
- inexpensive memory

This is not possible (today), so every OS works with a memory hierarchy
Performance of Various Levels of Storage

- Movement between levels of storage hierarchy can be explicit or implicit

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>registers</td>
<td>cache</td>
<td>main memory</td>
<td>disk storage</td>
</tr>
<tr>
<td>Typical size</td>
<td>&lt; 1 KB</td>
<td>&gt; 16 MB</td>
<td>&gt; 16 GB</td>
<td>&gt; 100 GB</td>
</tr>
<tr>
<td>Implementation technology</td>
<td>custom memory with multiple ports, CMOS</td>
<td>on-chip or off-chip CMOS SRAM</td>
<td>CMOS DRAM</td>
<td>magnetic disk</td>
</tr>
<tr>
<td>Access time (ns)</td>
<td>0.25 - 0.5</td>
<td>0.5 - 25</td>
<td>80 - 250</td>
<td>5,000,000</td>
</tr>
<tr>
<td>Bandwidth (MB/sec)</td>
<td>20,000 - 100,000</td>
<td>5000 - 10,000</td>
<td>1000 - 5000</td>
<td>20 - 150</td>
</tr>
<tr>
<td>Managed by</td>
<td>compiler</td>
<td>hardware</td>
<td>operating system</td>
<td>operating system</td>
</tr>
<tr>
<td>Backed by</td>
<td>cache</td>
<td>main memory</td>
<td>disk</td>
<td>CD or tape</td>
</tr>
</tbody>
</table>

OS Memory Manager
- Program must be brought (from disk) into memory and placed within a process for it to be run
- Main memory and registers are only storage CPU can access directly
- Register access in one CPU clock (or less)
- Main memory can take many cycles
- **Cache** sits between main memory and CPU registers
- Protection of memory required to ensure correct operation
History: No memory abstraction at all

- Programs use physical memory addresses.
  (a) OS at the bottom in RAM (e.g., early mainframe and minicomputers)
  (b) OS at the top in ROM (e.g., some handhelds and embedded devices)
  (c) device drivers at the top in ROM, rest at bottom in RAM (e.g., early PCs, running DOS, ROM: BIOS)

- Only one process at a time can be run this way!
Multiple processes w/o mem. abstraction

- **Swap** entire processes in & out (→ costly!)
- Early IBM 360s:
  - divide memory into (2kB) blocks, each protected by a (4bit) protection key, held in special CPU register;
  - PSW (pgm status word) also contained 4bit key;
  - hardware trapped mem. access, when PSW did not match memory’s key;
  - (only OS could modify protection keys);
- Processes could not interfere with each other’s (or OS’s) memory
Drawback: Relocation Problem

Two processes loaded into different regions of physical memory
Static relocation

- When loading the program, modify all references to memory addresses (cf. IBM’s early OS/360 solution)
  - requires additional information in all executable programs
  - slows down loading significantly
  - not very a general solution

**Conclusion:** using *physical* memory addresses in programs is not a good idea!

- Yet: many embedded and smart-card systems are implemented that way (radios, washing machines, toasters, microwave ovens, ...). Here, it can work, since all programs are known in advance.
Dynamic relocation: Address Spaces

- **Idea:** shield processes from each other, and from the OS, by giving to each one of them, a **separate, private address space.**
  - “Process” is an abstraction of the physical CPU.
  - “Address Space” is an abstraction of physical memory.
- **Base and limit registers** (special purpose CPU registers) can be used to implement address spaces and simple, dynamic relocation.
  - classical solution, e.g., in CDC 6600 (1st Supercomputer), Intel 8088 (1st PC)
  - only accessible to OS (not so for Intel 8088, which did not even have a limit register!)
  - Comes with a **cost:** each memory reference needs to be translated (addition) and checked (comparison) → significant extra cost in CPU!
MMU: Memory Management Unit

16384

Limit register

0

32764

...

CMP

16412

16408

16404

16400

16396

16392

16388

16384

16380

JMP 28

0

Base register

ADD

28

24

20

16

12

8

4

0

JMP 24

(c)

16384

relocation register

physical address 16384

physical memory

CPU

logical address

physical address

28

16412

<

base + limit

trap to OS monitor: addressing error

Relocation Protection
Swapping
Swapping

- Sum of sizes of address spaces exceeds available physical memory: *memory overload*

- Swapping:
  - bring in and out *entire processes* (i.e., their address spaces) to/from backing store (disk)
  - idle processes swapped out anyway
  - swapping back in to different memory location: needs dynamic relocation!
  - swapping can create holes in (physical) main memory, may need to do memory compaction (→ costly!)
A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution.

**Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images.

**Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed.

Major part of swap time is **transfer time**; total transfer time is directly proportional to the amount of memory swapped.

Round-robin scheduler should adjust **time quantum** to be significantly larger!

Modified versions of swapping are found on many systems (i.e., UNIX, Linux, and Windows), especially under **high system load**.

System maintains a **ready queue** of ready-to-run processes which have memory images on disk.
Example: Swapping in and out
Processes may dynamically grow ...

- Leave a little extra space (and remember not to swap out free space)
Free Space Management

— Contiguous Allocation —
Managing free space

- Two principal alternatives
  - (a) Example situation
  - (b) bitmaps
  - (c) linked lists
Dynamic Storage-Allocation Problem

How to satisfy a request of size $n$ from a list of free holes

- **First-fit**: Allocate the *first* hole that is big enough
- **Best-fit**: Allocate the *smallest* hole that is big enough; must search entire list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate the *largest* hole; must also search entire list
  - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms of speed and storage utilization
External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous

Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used

Reduce external fragmentation by compaction

- Shuffle memory contents to place all free memory together in one large block
- Compaction is possible only if relocation is dynamic, and is done at execution time
- I/O problem (asynchronous I/O running when process shall be swapped out)
  - Latch job in memory while it is involved in I/O
  - Do I/O only into OS buffers

50% rule: even with some optimization, when $N$ blocks are allocated to partitions, another 0.5 $N$ blocks will be lost to fragmentation!
Paging

— Non-contiguous Allocation —
Logical address space of a process can be non-contiguous; process is allocated physical memory whenever the latter is available.

Divide physical memory into fixed-sized blocks called **frames** (size is a power of 2, between 512 bytes and 8,192 bytes).

Divide logical memory into blocks of same size called **pages**.

Keep track of all free frames.

To run a program of size $n$ pages, need to find $n$ free frames and load program.

Set up a page table to translate logical to physical addresses.

Internal fragmentation.
Address generated by CPU is divided into:

- **Page number** \( p \) – used as an index into a page table which contains base address of each page in physical memory

- **Page offset** \( d \) – combined with base address to define the physical memory address that is sent to the memory unit

For given logical address space \( 2^m \) and page size \( 2^n \)
Paging Hardware
Example:

32-byte memory and 4-byte pages
Allocating a new process’s address space

(a) Before allocation

- Free-frame list:
  - 14
  - 13
  - 18
  - 20
  - 15
- Page table:
  - Page 0
  - Page 1
  - Page 2
  - Page 3
- New process

(b) After allocation

- Free-frame list:
  - 15
- Page table:
  - Page 0
  - Page 1
  - Page 2
  - Page 3
- New-process page table

0 14
1 13
2 18
3 20

new process
Page table is kept in main memory

**Page-table base register (PTBR)** points to the page table

**Page-table length register (PRLR)** indicates size of the page table

In this scheme every data/instruction access requires two memory accesses. One for the page table and one for the data/instruction.

The two memory access problem can be solved by the use of a special fast-lookup hardware cache called **associative memory** or **translation look-aside buffers (TLBs)**

Some TLBs store **address-space identifiers (ASIDs)** in each TLB entry – uniquely identifies each process to provide address-space protection for that process

Associative memory is particularly fast, due to **parallel search**: all Page# entries compared at the same time, extremely fast
Paging Hardware With TLB

The diagram illustrates the process of converting a logical address to a physical address in a paging system with a TLB (Translation Lookaside Buffer).

1. The CPU generates a logical address, which consists of a page number (p) and a frame number (d).
2. The logical address is sent to the TLB (Translation Lookaside Buffer), which checks for a TLB hit.
3. If there is a TLB hit, the physical address is returned directly to the CPU.
4. If there is a TLB miss, the page table is consulted to find the corresponding physical address.
5. The physical address is then sent to the physical memory.

This process efficiently handles address translation in a virtual memory system.
Effective Access Time

- Associative Lookup = $\varepsilon$ time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; ratio related to number of associative registers
- Hit ratio = $\alpha$
- Effective Access Time (EAT)

$$EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha)$$

$$= 2 + \varepsilon - \alpha$$
Memory protection implemented by associating protection bit(s) with each frame

- **Valid-invalid** bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page
  - “invalid” indicates that the page is not in the process’ logical address space

- Possibly, more bits can be associated with each page
  - read-only
  - read-write
  - execute only
  - modified („dirty bit“)
  - ...

...
### Valid (v) or Invalid (i) Bit In A Page Table

<table>
<thead>
<tr>
<th>Frame Number</th>
<th>Valid-Invalid Bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>v</td>
</tr>
<tr>
<td>1</td>
<td>v</td>
</tr>
<tr>
<td>2</td>
<td>v</td>
</tr>
<tr>
<td>3</td>
<td>v</td>
</tr>
<tr>
<td>4</td>
<td>v</td>
</tr>
<tr>
<td>5</td>
<td>v</td>
</tr>
<tr>
<td>6</td>
<td>i</td>
</tr>
<tr>
<td>7</td>
<td>i</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page Table</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>page 0</td>
<td>0</td>
</tr>
<tr>
<td>page 1</td>
<td>1</td>
</tr>
<tr>
<td>page 2</td>
<td>2</td>
</tr>
<tr>
<td>page 3</td>
<td>3</td>
</tr>
<tr>
<td>page 4</td>
<td>4</td>
</tr>
<tr>
<td>page 5</td>
<td>5</td>
</tr>
<tr>
<td>page 6</td>
<td>6</td>
</tr>
<tr>
<td>page 7</td>
<td>7</td>
</tr>
<tr>
<td>page 8</td>
<td>8</td>
</tr>
<tr>
<td>page 9</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>page n</td>
<td></td>
</tr>
</tbody>
</table>
Shared Pages

**Shared code**
- One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
- Shared code must appear in same location in the logical address space of all processes.
- OS should be able to enforce read-only characteristics.
- Different from shared memory as communication media.

**Private code and data**
- Each process keeps a separate copy of the code and data.
- The pages for the private code and data can appear anywhere in the logical address space.
Editor code (pages ed1-ed3) loaded into memory only \textit{once}.
Current, large address spaces (32- or 64-bit addressing) makes management of page table non-trivial.

- Multiple levels (page table is itself paged)
- Hashed page tables (esp. for >32bit addressing; large address space only partially used)
- Inverted page table (1 entry per physical frame, instead of 1 per logical page; large address space only partially used)
Memory-management scheme that supports user view of memory

A program is a collection of segments. A segment is a logical unit such as:

- main program,
- procedure,
- function,
- method,
- object,
- local variables, global variables,
- common block,
- stack,
- symbol table, arrays
Logical address consists of a two tuple:

\[ \text{<segment-number, offset>} \]

**Segment table** – maps two-dimensional physical addresses; each table entry has:

- **base** – contains the starting physical address where the segments reside in memory
- **limit** – specifies the length of the segment

**Segment-table base register (STBR)** points to the segment table’s location in memory

**Segment-table length register (STLR)** indicates number of segments used by a program;

segment number \( s \) is legal if \( s < \text{STLR} \)
Protection

- With each entry in segment table associate:
  - validation bit = 0 ⇒ illegal segment
  - read/write/execute privileges

Protection bits associated with segments; code sharing occurs at segment level

Since segments vary in length, memory allocation is a dynamic storage-allocation problem

A segmentation example is shown in the following diagram
Segmentation Hardware

CPU

\[ s \]

segment table

\( \text{limit base} \)

\( s \)

\( + \)

\( < \)

no

yes

trap: addressing error

physical memory
Virtual Memory — Demand Paging —
Typically, programs exhibit *locality* in their memory reference pattern.

Only *part* of a program’s address space needs to be allocated to real memory frames, at any time during program execution.

Virtual memory management includes mechanisms to avoid swapping in/out *entire* processes.
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less physical memory needed
  - Faster response
  - More parallel processes

- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **(demand) pager**

- **valid/invalid** bit in page table (see above) can be used to indicate whether page is in main memory or not
### Page Table When Some Pages Are Not in Main Memory

<table>
<thead>
<tr>
<th>logical memory</th>
<th>page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>i</td>
</tr>
<tr>
<td>C</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>i</td>
</tr>
<tr>
<td>E</td>
<td>i</td>
</tr>
<tr>
<td>F</td>
<td>9</td>
</tr>
<tr>
<td>G</td>
<td>i</td>
</tr>
<tr>
<td>H</td>
<td>i</td>
</tr>
</tbody>
</table>

**valid-invalid bit**

- 0: invalid
- 1: valid

<table>
<thead>
<tr>
<th>frame</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>v</td>
<td>i</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**physical memory**

- A
- B
- C
- D
- E
- F
- H

- Empty frames: 1-3, 5-7,
If there is a reference to a page, first reference to that page will trap to operating system:

**page fault**

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = v
6. Restart the instruction that caused the page fault
Page Fault Rate $0 \leq p \leq 1.0$
- if $p = 0$ no page faults
- if $p = 1$, every reference is a fault

Effective Access Time (EAT)
$EAT = (1 - p) \times \text{memory access} + p \times (\text{page fault overhead} + \text{swap page out} + \text{swap page in} + \text{restart overhead})$

Example:
- Memory access time = 200 ns
- Average page-fault service time = 8 ms
- $EAT = (1 - p) \times 200 + p \times 8,000,000$
  $= 200 + p \times 7,999,800$
  If one access out of 1,000 causes a page fault, then
  $EAT = 8.2 \mu s$. 
  This is a slowdown by a factor of 40!!
Process creation: Copy-on-write

- Demand paging can speed-up process creation significantly
  - Initially, parent and child processes can share all pages
  - Either process creates copies as soon as content is modified

Before process 1 modifies page C

After process 1 modifies page C
Virtual Memory
— Page Replacement —
What happens, if no free frame?

- Page replacement algorithm needed to identify victim page
  - in memory, but not really used
  - swapped out to disk (only if modified)
  - frame used to swap in desired page
  - needs to update page tables

- Performance: want algorithm that minimizes page faults
  - Same page may be swapped out/in several times
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process
Page Replacement Algorithms

- Want lowest page-fault rate

- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

- In the following, we analyze examples with buffer size=3, the reference string is

\[7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1\]
Replace page that will not be used for longest period of time (in the future!)

How do you know this?

Used for measuring how well your algorithm performs

<table>
<thead>
<tr>
<th>reference string</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>page frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 7 7 2 2 2 2 2</td>
</tr>
<tr>
<td>0 0 0 0 4 0 0</td>
</tr>
<tr>
<td>1 1 3 3 3 1</td>
</tr>
<tr>
<td>7 0 1</td>
</tr>
</tbody>
</table>

9 page faults (3 are inevitable, so we’ll count only the other 6 here and in the other analyses)
First-In-First-Out (FIFO) Algorithm

- Replace oldest page in buffer

**reference string**

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |

**page frames**

- 13 page faults (plus 3 initial loads)
- more than twice as much as optimal algorithm
Belady’s Anomaly: more frames $\Rightarrow$ more page faults
Least Recently Used (LRU) Algorithm

- Replace page, whose last reference is oldest

**Reference string**

| 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| 7 | 7 | 7 | 2 | 2 | 4 | 4 | 4 | 0 | 1 | 1 | 1 | 3 | 1 | 0 | 0 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 7 |

**Page frames**

- 9 page faults (plus 3 initial loads)
„Stack“ implementation – keep a „stack“ of page numbers (actually, a doubly linked list):

- Page referenced:
  - move it to the top
  - requires 6 pointers to be changed
- No search for replacement (always replace bottom page)
Counter implementation

- Every page table entry has a time-of-use entry;
  - every time page is referenced through this table entry, copy the CPU clock (or some other counter) into the time-of-use entry;
  - When a page needs to be swapped out, look at the time-of-use entries to determine the victim (oldest one)
  - Entries may need to be updated when page tables are changed (CPU scheduling)

- Overhead at access time is much smaller than with stack-based implementation, but now we need to search for victim page.
Reference bit
- With each page associate a bit, initially = 0
- When page is referenced bit set to 1
- Replace the one which is 0 (if one exists)
  - We do not know the order, however

Second chance
- Need reference bit
- Clock replacement
- If page to be replaced (in clock order) has reference bit = 1 then:
  - set reference bit 0
  - leave page in memory
  - replace next page (in clock order), subject to same rules
Second Chance (Clock, GClock)

---

**CLOCK**
("Second Chance")

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
```

"used" bit

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

**GCLOCK**

possibly initialized with weights

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

ref count

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

---

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
```

---
Counting Algorithms

- Keep a counter of the number of references that have been made to each page

- **LFU Algorithm**: replaces page with smallest count

- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
## Summary of Replacement Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>Not implementable, but useful as a benchmark</td>
</tr>
<tr>
<td>NRU (Not Recently Used)</td>
<td>Very crude approximation of LRU</td>
</tr>
<tr>
<td>FIFO (First-In, First-Out)</td>
<td>Might throw out important pages</td>
</tr>
<tr>
<td>Second chance</td>
<td>Big improvement over FIFO</td>
</tr>
<tr>
<td>Clock</td>
<td>Realistic</td>
</tr>
<tr>
<td>LRU (Least Recently Used)</td>
<td>Excellent, but difficult to implement exactly</td>
</tr>
<tr>
<td>NFU (Not Frequently Used)</td>
<td>Fairly crude approximation to LRU</td>
</tr>
<tr>
<td>Aging</td>
<td>Efficient algorithm that approximates LRU well</td>
</tr>
<tr>
<td>Working set</td>
<td>Somewhat expensive to implement</td>
</tr>
<tr>
<td>WSClock</td>
<td>Good efficient algorithm</td>
</tr>
</tbody>
</table>
Virtual Memory
— Allocation Strategies —
Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation
**Fixed Allocation**

- **Equal allocation** – For example, if there are 100 frames and 5 processes, give each process 20 frames.

- **Proportional allocation** – Allocate according to the size of process
  
  - \( s_i \) = size of process \( p_i \)
  
  - \( S = \sum s_i \)
  
  - \( m = \) total number of frames

  - \( a_i = \) allocation for \( p_i = \frac{s_i}{S} \times m \)

  \[
  
  m = 64 \\
  s_i = 10 \\
  s_2 = 127 \\
  a_1 = \frac{10}{137} \times 64 \approx 5 \\
  a_2 = \frac{127}{137} \times 64 \approx 59 
  
  \]
Use a proportional allocation scheme using priorities rather than size.

If process $P_i$ generates a page fault,
- select for replacement one of its frames
- select for replacement a frame from a process with lower priority number
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another

- **Local replacement** – each process selects from only its own set of allocated frames
If a process does not have “enough” pages, the page-fault rate is very high. This leads to:

- low CPU utilization
- operating system thinks that it needs to increase the degree of multiprogramming
- another process added to the system

**Thrashing** = a process is busy swapping pages in and out
Why does demand paging work?
Locality model
- Process migrates from one locality to another
- Localities may overlap

Why does thrashing occur?
\[ \sum \text{size of locality} > \text{total memory size} \]
Working-Set Model

- \( \Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references} \)
  Example: 10,000 instructions (or references)

- \( WSS_i(t) \) (working set size of process \( P_i \) at time \( t \)) = total number of distinct pages referenced in the most recent \( \Delta \) references, up to and including time \( t \)

- How to best set \( \Delta \)?
  - if \( \Delta \) too small, will not encompass entire locality
  - if \( \Delta \) too large, will encompass several localities
  - if \( \Delta = \infty \Rightarrow \) will encompass entire program

- \( D = \sum WSS_i \equiv \text{total demand of frames by all processes} \)

- if \( D > m \Rightarrow \text{Thrashing} \)

- Policy: if \( D > m \), then suspend (at least) one of the processes
Working-Set Model

Page reference table

\[\ldots 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 3 4 3 4 4 4 1 3 2 3 4 4 3 4 4 4 \ldots\]

\[\Delta\]

\[t_1\]

\[ WS(t_1) = \{1,2,5,6,7\} \]

\[t_2\]

\[ WS(t_2) = \{3,4\} \]
Keeping Track of the Working Set (1)

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$ time units
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever the timer interrupts:
    - copy first to second reference bit, and
    - set the values of all first reference bits to 0
  - If one of the reference bits in memory = 1 $\Rightarrow$ page in working set
- Not completely accurate ...
- Improvement = 10 bits and interrupt every 1000 time units
Keeping Track of the Working Set (2)

Scan all pages examining R bit:
- if (R == 1)
  set time of last use to current virtual time
- if (R == 0 and age > τ)
  remove this page
- if (R == 0 and age ≤ τ)
  remember the smallest time
Establish “acceptable” page-fault rate

- If actual rate too low, process loses frame
- If actual rate too high, process gains frame
Allocating Kernel Memory
Allocating Kernel Memory

- Treated differently from user memory
  - Needs to be more efficient
  - Often not subjected to demand paging (memory resident)
- Often allocated from a free-memory pool (to avoid fragmentation)
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
Allocates memory from fixed-size segment consisting of physically-contiguous pages

Memory allocated using **power-of-2 allocator**

- Satisfies requests in units sized as power of 2
- Request rounded up to next highest power of 2
- When smaller allocation needed than is available, current chunk split into two *buddies* of next-lower power of 2
  - Continue until appropriate sized chunk available
- Since all regions are power of 2, neighboring free regions can easily be joined to form next larger-size region again.
Buddy System Allocator

Physically contiguous pages

- 256 KB
- 128 KB $A_L$
- 128 KB $A_R$
- 64 KB $B_L$
- 64 KB $B_R$
- 32 KB $C_L$
- 32 KB $C_R$
Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction
Slab Allocation

- **kernel objects**
- **caches**
- **slabs**

3 KB objects

7 KB objects

physical contiguous pages
Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume $s$ pages are prepaged and $a$ of the pages are actually used
  - Is cost of $s \times a$ save pages faults $>$ or $<$ than the cost of prepaging $s \times (1 - a)$ unnecessary pages?
  - $a$ near zero $\Rightarrow$ prepaging loses

Page size selection must take into consideration:

- fragmentation
- page table size
- I/O overhead
- locality
Other Issues — TLB Reach

- TLB Reach — The amount of memory accessible from the TLB
  \[ \text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size}) \]

- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high amount of page faults

- Increase the Page Size
  - This may lead to an increase in (internal) fragmentation as not all applications require a large page size

- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation
Program structure

- `int[128,128] data;`
- Each row is stored in one page
- Program 1

```c
for (j = 0; j < 128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

- Program 2

```c
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults
Other Issues — Segmentation

- Virtual address space of a single process may be segmented
  - on an even more indirect level, 1 process = many address spaces
  - for protection reasons (e.g., code vs. data)
  - for independent growth/shrinkage reasons (different compiler-generated variable-size tables)

- Implementation considerations
  - virtual address = segment#, page#, offset
  - internal fragmentation, compaction
  - Some hardware support more than 2 privilege levels, e.g., kernel — system calls — shared libraries — user programs
### Intended Schedule

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Hand out</th>
<th>Submission</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>20.04. Introduction to Operating Systems</td>
<td></td>
<td>Course registration</td>
</tr>
<tr>
<td>1</td>
<td>27.04. Systems Programming using C (File Subsystem)</td>
<td>1. Assignment</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>04.05. Systems Programming using C (Process Control)</td>
<td>2. Assignment</td>
<td>1. Assignment</td>
</tr>
<tr>
<td>3</td>
<td>11.05. Process Scheduling</td>
<td>3. Assignment</td>
<td>2. Assignment</td>
</tr>
<tr>
<td>4</td>
<td>18.05. Process Synchronization</td>
<td>4. Assignment</td>
<td>3. Assignment</td>
</tr>
<tr>
<td>5</td>
<td>25.05. Inter Process Communication</td>
<td>5. Assignment</td>
<td>4. Assignment</td>
</tr>
<tr>
<td>6</td>
<td>01.06. Pfingstmontag</td>
<td>6. Assignment</td>
<td>5. Assignment</td>
</tr>
<tr>
<td>6</td>
<td>08.06. Deadlocks</td>
<td>7. Assignment</td>
<td>6. Assignment</td>
</tr>
<tr>
<td>7</td>
<td>15.06. Memory Management</td>
<td>8. Assignment</td>
<td>7. Assignment</td>
</tr>
<tr>
<td>8</td>
<td>22.06. Input / Output</td>
<td>9. Assignment</td>
<td>8. Assignment</td>
</tr>
<tr>
<td>9</td>
<td>29.06. Filesystems</td>
<td>10. Assignment</td>
<td>9. Assignment</td>
</tr>
<tr>
<td>10</td>
<td>06.07.</td>
<td></td>
<td>10. Assignment</td>
</tr>
<tr>
<td>11</td>
<td>13.07. Special subject: XQuery your Filesystem</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>20.07. Wrap up session</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27.07. First examination date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12.10.</td>
<td>12.10. Second examination date</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
VII. Input/Output