Chapter 6: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson’s Solution
- Synchronization Hardware
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Atomic Transactions
Problem of Race Condition

- Consider this execution interleaving with “count == 5” initially:

  \[
  \begin{align*}
  T_0: & & \text{producer} & & \text{register}_1 = \text{count} & \{\text{register}_1 == 5\} \\
  T_1: & & \text{producer} & & \text{register}_1 = \text{register}_1 + 1 & \{\text{register}_1 == 6\} \\
  T_2: & & \text{consumer} & & \text{register}_2 = \text{count} & \{\text{register}_2 == 5\} \\
  T_3: & & \text{consumer} & & \text{register}_2 = \text{register}_2 - 1 & \{\text{register}_2 == 4\} \\
  T_4: & & \text{producer} & & \text{count} = \text{register}_1 & \{\text{count} == 6\} \\
  T_5: & & \text{consumer} & & \text{count} = \text{register}_2 & \{\text{count} == 4\}
  \end{align*}
  \]

- The variable “count” could end up with a value of 4, 5, or 6, depending upon how the execution of the instruction streams is interleaved. This is called a race condition.

The Critical-Section Problem

- In a system, multiple processes are competing to use some shared data or resource
- Each process has a code segment, called a critical section, in which shared data or resource is accessed.
- Problem: to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
Solution to Critical-Section Problem

- For any solution to the Critical Section problem to be complete, all of the following conditions must be satisfied:
  - Mutual Exclusion
  - Progress
  - Bounded waiting (Fairness)

Synchronization Hardware

- The `TestAndSet()` instruction tests the value of a word in memory and sets a new value, as an indivisible (atomic) unit.
- The `Swap()` instruction swaps the contents of two words in memory, as an indivisible (atomic) unit.
- Mutual exclusion with `TestAndSet()` and `Swap()`
Semaphores

- Semaphores provide a simple tool for process synchronization.
- A semaphore S is an integer variable that can only be accessed through two standard atomic operations, \texttt{wait()} and \texttt{signal()}
- Two types of semaphores:
  - \textbf{Counting semaphore} – the integer value can range over an unrestricted domain.
  - \textbf{Binary semaphore} – the value can be only 0 or 1.
    - Also known as \texttt{mutex locks}.
- Semaphore Usage
  - Mutual exclusion
  - Order of execution

Semaphore Implementation

- Implementation of \texttt{wait()}:
  ```
  \texttt{wait(s)} \{ \\
  \quad \texttt{S.value--;} \\
  \quad \texttt{if (S.value < 0) \{} \\
  \quad \quad \texttt{add this process to S.list;} \\
  \quad \quad \texttt{block();} \\
  \quad \texttt{\}} \\
  \}
  ```
- Implementation of \texttt{signal()}:
  ```
  \texttt{signal(s)} \{ \\
  \quad \texttt{S.value++;} \\
  \quad \texttt{if (S.value <= 0) \{} \\
  \quad \quad \texttt{remove a process P from S.list;} \\
  \quad \quad \texttt{wakeup(P);} \\
  \quad \texttt{\}} \\
  \}
  ```
Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Monitors

- A high-level language construct that provides for safe sharing of an abstract data type among concurrent processes.
- The monitor type can be characterized as a set of programmer-defined operations that execute with mutual exclusion within the monitor.
- Condition variables for synchronization
Atomic Transactions

- Make sure that a critical section forms a single logical unit of work that is either performed in its entirety or not performed at all.
- Consistency of data is the primary concern.
- Operating systems design can benefit from database systems techniques for atomic transactions.
- **Write-ahead logging** and log-based recovery with checkpoints.
- Concurrent transactions.

Chapter 7: Deadlocks

- The Deadlock Problem
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
The Deadlock Problem

• A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

• Example
  – System has 2 disk drives.
  – $P_1$ and $P_2$ each hold one disk drive and each needs another one.

• Example
  – Semaphores $A$ and $B$, initialized to 1

\[
\begin{align*}
P_0 & \quad P_1 \\
\text{wait}(A); & \quad \text{wait}(B) \\
\text{wait}(B); & \quad \text{wait}(A)
\end{align*}
\]

System Model

• Resource types $R_1, R_2, \ldots, R_m$

  *CPU cycles, memory space, I/O devices*

• Each resource type $R_i$ has $W_i$ instances.

• Each process utilizes a resource as follows:
  – Request
  – Use
  – Release
Deadlock Characterization

• **Mutual exclusion**: At least one resource must be held in a non-sharable, exclusive mode.

• **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources held by other processes.

• **No preemption**: Resources cannot be preempted; i.e., a resource can be released only voluntarily by the process holding it, after that process has completed its task.

Deadlock Characterization (cont.)

• **Circular wait**: There exists a set \( \{P_0, P_1, \ldots, P_n\} \) of waiting processes such that
  - \( P_0 \) is waiting for a resource that is held by \( P_1 \),
  - \( P_1 \) is waiting for a resource that is held by \( P_2 \),
  - \( \ldots \),
  - \( P_{n-1} \) is waiting for a resource that is held by \( P_n \),
  - and \( P_n \) is waiting for a resource that is held by \( P_0 \).
Resource-Allocation Graph

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- Request edge – directed edge $P_i \rightarrow R_j$
- Assignment edge – directed edge $R_j \rightarrow P_i$

Basic Facts About Cycles

- If graph contains no cycles $\Rightarrow$ no deadlock.
- If graph contains a cycle $\Rightarrow$ deadlock may exist.
  - If there is only one instance of each resource type, then deadlock.
  - If there are multiple instances of each resource type, possibility of deadlock.
Methods for Handling Deadlocks

• Use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.

• Allow the system to enter a deadlocked state, detect it, and then recover.

• Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX. Deadlock prevention/handling left up to programmers.

Deadlock Prevention

• Mutual Exclusion – not required for sharable resources, must hold for nonsharable resources.

• Hold and Wait – must guarantee that whenever a process requests a resource, it does not hold any other resources.
  – Require a process to request and be allocated all of its resources before it begins execution, or allow a process to request resources only when the process has none.
  – Low resource utilization; starvation possible.
Deadlock Prevention (cont.)

• No Preemption –
  – If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released. Resource states must be saved.
  – Preempted resources are added to the list of resources for which the process is waiting.
  – Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting. Resource states must be restored.

• Circular Wait – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration.

Deadlock Avoidance

• Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
• The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
• Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes.
Safe State

- When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state.
- The system is in a safe state if it can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence in which the processes can be executed.

Basic Facts About Safe States

- If a system is in a safe state $\Rightarrow$ no deadlocks.
- If a system is in an unsafe state $\Rightarrow$ possibility of a deadlock.
- Deadlock avoidance $\Rightarrow$ ensure that a system will never enter an unsafe state.
Deadlock Avoidance Algorithms

- For a system with only a single instance of each resource type, use a variant of the resource-allocation graph.

- For a system with multiple instances of resource type, use the banker’s algorithm.

Banker’s Algorithm

- For resource types with multiple instances.
- Each process must make an *a priori* claim for its maximum resource use.
- When a process requests a resource it may have to wait, if granting the request would put the system into an unsafe state.
- When a process gets all of its resources it must return them in a finite amount of time.
Deadlock Detection

- Allow system to enter deadlock state.
- Detection algorithm.
- Recovery scheme.

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes.
- Abort one process at a time until the deadlock cycle is eliminated.
- Factors for choosing which process to abort:
  - Priority of the process.
  - How long the process has executed, and how much longer to completion.
  - Resources the process has used.
  - Resources the process needs to complete.
  - How many processes will need to be terminated.
  - Is process interactive or batch?
Chapter 8: Main Memory

• Background
• Swapping
• Contiguous Memory Allocation
• Paging
• Structure of the Page Table
• Segmentation
• Example: The Intel Pentium

Logical vs. Physical Address Space

• The concept of a logical address space that is bound to a separate physical address space is central to memory management.
  – Logical address – generated by the CPU. Also referred to as a virtual address.
  – Physical address – address used by the memory unit to actually reference physical memory.

• Logical and physical addresses are the same in compile-time and load-time address-binding schemes. Logical (virtual) and physical addresses differ in the execution-time address-binding scheme.
Memory Allocation Schemes

• Contiguous Memory Allocation
• Paging
• Segmentation

Contiguous Memory Allocation

• The address space for a process is allocated in a single contiguous partition in main memory.
• Main memory usually divided into two regions:
  – Resident operating system, usually placed in low memory with the interrupt vector.
  – User processes are then placed in high memory.
• Relocation and limit registers can be used to protect user processes from each other, and from changing operating-system code and data.
  – The relocation register contains the address of the smallest physical address, and the limit register specifies the range of permissible logical addresses.
  – The MMU maps the logical address dynamically.
Dynamic Storage-Allocation Problem

- Algorithms for allocating a request of size $n$ from a list of 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, unless sorted. Produces the largest leftover hole.

- Simulations show both first fit and best fit are better than worst fit in terms of speed and storage utilization. But neither is clearly better than the other. First fit is faster.

---

Fragmentation

- If a process needs 380K of space, it cannot be allocated – the holes are all too small.
- Small holes are fragments.
- **External fragmentation** – The total amount of free memory exists to satisfy a request, but it is not contiguous.
  - “External” – unused space outside of the partitions.
Paging

- Avoids the fragmentation problem of contiguous allocation.
- The physical address space of a process can be noncontiguous. Memory is allocated to a process physical address space in fixed size blocks wherever it is available.
- Divide physical memory into fixed-sized blocks called frames (size is a power of 2, between 512 bytes and 16 MB, depending on system architecture).
- Divide logical memory into blocks of the same size called pages.

Paging (cont.)

- OS keeps track of all free frames in memory.
- If a process needs \( n \) pages, need to find \( n \) free frames and load program.
- Set up a page table to translate logical to physical addresses via dynamic relocation in hardware.
- **Internal fragmentation** – unused memory within an allocated frame (or partition). The memory space needed by a process will usually be somewhat less than a multiple of the page size, so part of the last frame may not be used.
  - On average, one-half frame per process.
Address Translation Hardware

Paging Hardware With TLB
Effective Access Time with TLB

- Assume TLB lookup time is $T_a$ time units (e.g. ns).
- Assume memory cycle is $T_m$ time units.
- Hit ratio ($h$) – percentage of times that a page number is found in the TLB ($h = .8$ means 80%).
- Assume $h = 0.8$, $T_a = 20$, $T_m = 100$
- $EAT = (\text{found in TLB}): (T_a + T_m)h$
  + (not in TLB): $(T_a + 2T_m)(1-h)$
  = $120 \times 0.8 + 220 \times 0.2 = 140$
  $<< 200$ (twice memory access times)
- $EAT = 122$ ns if $h = 0.98$

Valid (v) or Invalid (i) Bit in a Page Table
Shared Pages

- An advantage of paging is the possibility of *sharing* common code, which is loaded into *shared pages*.
  - One copy of read-only (*reentrant*) code can be shared among multiple processes (i.e., text editors, compilers, window systems).
  - Shared code must appear in the same location in the logical address space of all processes.
- For 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.

Shared Pages Example

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

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

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

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

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

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

```

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

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

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

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

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

```
<table>
<thead>
<tr>
<th></th>
<th>data 1</th>
<th>data 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>data 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>data 2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>data 3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>data 3</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>
```
Structure of the Page Table

- Hierarchical Paging
- Hashed Page Tables
- Inverted Page Tables

Address-Translation Scheme

[Diagram showing the address-translation scheme with labels such as logcal address, p1, p2, d, outer page table, page of page table, and arrows connecting them.]
Effective Access Time with 2-Level Page Tables

- Assume TLB lookup time: \( T_a = 20 \) ns
- Assume memory access time: \( T_m = 100 \) ns
- Hit ratio: \( h = 0.98 \)
- \( EAT = \begin{cases} \text{found in TLB: } (T_a + T_m)h \\ \text{not in TLB: } (T_a + 3T_m)(1-h) \end{cases} \)
  - \( = 120 \times 0.98 + 320 \times 0.02 = 124 \)
- Slightly larger than the 122 for single-level page table.
  - \( \ll 300 \) ns (time for three memory accesses).

Segmentation Architecture

- Logical address consists of a two tuple:
  \(<\text{segment-number, offset}>\)
- **Segment table** – maps two-dimensional logical addresses to 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 (length of segment table).
  - Segment number \( s \) is legal if \( s < \text{STLR} \)
Chapter 9: Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples
Virtual Memory

• **Virtual memory** completes the separation of user’s logical memory from physical memory.
  – Only part of a process’ virtual memory needs to be in physical memory at any given time.
  – The **virtual address space** of a process can be much larger than the available physical memory.
  – System libraries and memory can be shared by several processes by mapping shared pages into each process’ virtual address space.
  – Shared pages can speed up process creation via `fork()`.

• Virtual memory can be implemented via:
  – Demand paging.
  – Demand segmentation.

Demand Paging

• Bring a page into memory only when it is needed (on demand).
  – Less I/O needed to initially load a process.
  – Pages that are never referenced are never allocated physical memory.

• When a page is referenced:
  – If it is an invalid reference, abort the process.
  – If the page is in memory, access it.
  – If the page is not in memory, bring it into memory.

• When pages are not in memory, they reside on disk. They are paged (swapped) in and out as needed.
Valid-Invalid Bit

• With each page table entry a valid–invalid bit is associated:
  – \( v \Rightarrow \) valid page in memory
  – \( i \Rightarrow \) invalid page, or not in memory

• Initially valid–invalid bit is set to \( i \) on all entries.

• Example of a page table snapshot:

• During address translation, if the valid–invalid bit in page table entry is \( i \) ⇒ page fault interrupt.

Page Fault

• If there is a reference to a page, and the valid-invalid bit is \( i \), a page fault interrupt occurs.

• Basic steps for processing a page fault:
  1. OS checks an internal table for the process (via PCB) to decide if it is a valid reference:
     • If an invalid reference ⇒ abort.
     • If a valid reference, and not in memory, need to bring the page into memory (page it in).
  2. Find an empty frame. (What if none? Later.)
  3. Read the page from disk into the frame.
  4. Update process table and page table entries for the page. Set valid-invalid bit to \( v \).
  5. Restart the instruction that caused the page fault.
Performance of Demand Paging

• Page Fault Rate \(0 \leq p \leq 1.0\)
  – If \(p = 0\), no page faults.
  – If \(p = 1\), every reference causes a page fault.

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

Page Replacement

• What happens when a page fault occurs and there are no free frames available?

• Page replacement is necessary – find some page in memory, but not really in use, and swap (page) it out, freeing up a frame.
  – Need a page replacement algorithm.
  – Performance – we want an algorithm which will result in the minimum number of page faults.

• Page replacement is basic to demand paging, and must be part of page fault handling.
Page Replacement Algorithms

• Want the lowest page-fault rate.
• Evaluate the algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
• In our examples, the reference string is:
  7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

FIFO Page Replacement

• Replace the page that has been in memory the longest.

| reference string | 7 | 0 | 1 | 2 | 0 | 3 | 0 | 4 | 2 | 3 | 0 | 3 | 2 | 1 | 2 | 0 | 1 | 7 | 0 | 1 |
| page frames      | 7 | 7 | 7 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 7 | 7 | 7 | 0 | 0 | 1 | 0 | 0 | 2 | 2 |

• 15 page faults for example reference string.
• Performance is not always good. Maybe a page has been in memory the longest because it contains a variable that was initialized early and is in constant use.
Optimal Page Replacement

- Replace the page that will not be referenced again for the longest period of time into the future.
- 9 page faults for example reference string.
- Does not suffer from Belady’s anomaly.
- Not really possible to implement since it requires knowledge of the future. Mostly useful for comparisons with other algorithms.

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Page Frames</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 0 1 2 0 3 0 2 3 0 2 1 2 0 1 7 0 1</td>
<td>7 7 7 2 2 2 2 2 2 2 7 0 0 0 0 4 0 0 0 3 3 3 1 1</td>
</tr>
</tbody>
</table>

LRU Page Replacement

- Replace the page that has been least recently used (LRU), i.e. the page that has not been referenced for the longest period of time.
- 12 pages faults for example reference string.
- Approximates the optimal algorithm, because most programs exhibit a locality-of-reference property.
- Belongs to a class of algorithms, called stack algorithms, that can never exhibit Belady’s anomaly.

<table>
<thead>
<tr>
<th>Reference String</th>
<th>Page Frames</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>
<td>7 7 7 2 4 4 4 0 1 1 1 0 0 3 3 3 2 2 2 2 2 7</td>
</tr>
</tbody>
</table>
Second-Chance Algorithm:
- When considering a page for replacement, inspect the reference bit.
- If the reference bit is 0, replace it.
- If the reference bit is 1, give the page a second chance – reset the reference bit to 0, and move on to examine the next page.
- Can be implemented with a circular queue of pages, and pointer to indicate which page should be examined next.
  - Sometimes referred to as the clock algorithm.

Enhanced Second-Chance Algorithm:
- Consider both the reference bit and the modify bit as an ordered pair. There are four possible combinations:
  - (0,0) Neither recently used nor modified – best page to replace.
  - (0,1) Not recently used, but modified – not quite as good, since page will have to be written out.
  - (1,0) Recently used, but not modified – probably will be used again soon.
  - (1,1) Recently used and modified – worst page to replace.
- Replace the first page encountered in the lowest non-empty class. May need to scan the circular queue multiple times.
Global vs. Local Allocation

- **Local replacement** – Page replacement is limited to a process’ own set of allocated frames.
  - May hinder a process by not making available to it other, less used frames of memory.
- **Global replacement** – Page replacement can select from the set of all frames in the system; one process can take a frame from another.
  - May want to make higher priority processes immune.
  - Generally results in greater system throughput.

Thrashing

- If a process does not have “enough” frames, the page-fault rate can be very high. This is called **thrashing** – a process spends more time paging than executing.
- Thrashing causes severe performance problems:
  - If global page replacement is used, frames may be taken from other processes, causing their page fault rate to increase.
  - This causes CPU utilization to drop.
  - The OS may think that it needs to increase the degree of multiprogramming to increase CPU utilization, so it admits another process, making things worse.
  - Soon the whole system is thrashing.
Working-Set Model

- $\Delta = \text{working-set window} = \text{a fixed number of page references.}$
  Example: 10,000 instruction
- $WSS_i (\text{working set of Process } P_i) =$
  total number of pages referenced in the most recent $\Delta$
  (varies over time).
  - If $\Delta$ too small, will not encompass entire locality.
  - If $\Delta$ too large, will encompass several localities.
  - If $\Delta = \infty$ will encompass entire program.
- $D = \sum WSS_i : \text{total demand for frames.}$
- If $D > m \Rightarrow \text{ thrashing (} m = \text{total number of frames).}$
- Policy: If $D > m$, then suspend one of the processes,
  swap it out, and allocate its frames to other processes.

Page-Fault Frequency Scheme

- Establish upper and lower bounds for an acceptable
  page-fault rate for a process.
  - If the actual rate exceeds the upper bound, allocate the process
    another frame.
  - If actual rate too falls below the lower bound, remove a frame
    from the process.
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by mapping a disk block to a page in memory.
- A file is initially read using demand paging. A pagesized portion of the file is read from the file system into a physical page. Subsequent reads/writes from/to the file take place via efficient virtual memory accesses.
- File updates may be written to disk periodically, or when the file is closed.
- Multiple processes may map to the same file concurrently to allow sharing of data, and also allowing pages in memory to be shared.
Allocating Kernel Memory

- Kernel memory is often allocated from a free-memory pool different from the list used to satisfy user-mode requests.
  - Kernel requests memory for date structures and tables of varying sizes, many of which are less than a page. Need to minimize fragmentation.
  - Some kernel memory needs to be contiguous.
Buddy System

• Allocates memory from fixed-size segments consisting of physically-contiguous pages.
• Memory allocated using **power-of-2 allocator**.
  – Satisfies requests in units sized as a power of 2.
  – Requests are rounded up to next highest power of 2.
  – When a request is for less than half of an available segment, the segment is split into two buddies of next-lower power of 2. The process continues until an appropriate sized segment is available.
• Adjacent buddies can quickly be combined via **coalescing** when both are free.
• Can still have significant fragmentation, up to 50%.

Slab Allocation

• A strategy for kernel memory allocation with less fragmentation.
• A **slab** is one or more physically contiguous pages.
• A **cache** consists of one or more slabs.
• A single cache is used for each unique kernel data structure.
  – Each cache filled with **objects** – instances 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, the next object is allocated from an empty slab.
  – If no empty slabs, a new slab is allocated.
• Benefits include no fragmentation, fast processing of memory requests.
Other Issues

- Pre-paging
- Block Paging
- Page Size
- TLB Reach
- Program Structure
- I/O Interlock

Chapter 10: File System Interface

- File Concept
- Access Methods
- Directory Structure
- File-System Mounting
- File Sharing
- Protection
File Attributes

- **Name** – For the convenience of human users.
- **Identifier** – Unique tag (number) identifies the file within file system.
- **Type** – Needed for systems that support different types of files.
- **Location** – Pointer to a device and the file location on the device.
- **Size** – Current file size.
- **Protection** – Access control information specifies who can read, write, execute.
- **Time, date, and user identification** – Data for protection, security, and usage monitoring.
- Information about files is kept in the directory structure, which is maintained on the disk.

File Operations

- A file is an **abstract data type**. System calls in the OS implement basic file operations:
  - Create
  - Write
  - Read
  - Reposition within file (file seek)
  - Delete
  - Truncate
- **open**(\(F_i\)) – Search the directory structure for \(F_i\), create entries in system tables (open-file table), allocate buffers, etc.
- **close**(\(F_i\)) – Remove system entries, flush and de-allocate buffers, etc.
Access Methods

• Sequential Access
  read next
  write next
  reset (rewind)
  skip forward/backward

• Direct Access
  read \( n \)
  write \( n \)
  position to \( n \) (seek \( n \))
  read next
  write next

\( n = \) relative block number (not physical block number)

Operations Performed on Directory

• Search for a file.
• Create a file.
• Delete a file.
• List a directory.
  – All files in the directory and the directory entry for each file.
• Rename a file.
  – May allow its position within the directory structure to change.
• Traverse the file system.
  – All directories and files.
Issues for Directory Organization

- Efficiency – Locate a file quickly.
- Naming – Convenience for users.
  - Allow two users to have the same name for different files.
  - Allow the same file to have several different names.
- Grouping – Logical grouping of files by properties (e.g., all C programs, all files for a specific project, etc).

Single-Level Directory

- A single directory for all users:
  - Naming problem
  - Grouping problem
Two-Level Directory

- Separate directory for each user:
  - Path name (e.g. [user]file or /user/file).
  - Different users can have files with the same name.
  - Efficient searching.
  - No grouping capability.

Tree-Structured Directories
Acyclic-Graph Directories

- Allow "shared" subdirectories and files.
  - The same physical file may have the same or different names in different directories.

General Graph Directory
File Sharing – Consistency Semantics

- **Consistency semantics** specify how multiple users are to access a shared file simultaneously.
- Unix semantics:
  - Writes to an open file by a user are visible immediately to other users that have the file open at the same time.
- Session Semantics (e.g. AFS):
  - Writes to an open file by a user are not visible immediately to other users that have the same file open at the same time.
  - Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.
- Immutable-Shared-Files Semantics:
  - Contents are fixed and unalterable (read-only).

Protection

- File owner/creator should be able to control:
  - What can be done,
  - And by whom.
- Types of access:
  - Read (“r” in Unix)
  - Write (“w”)
  - Execute (“x”)
  - Append (“a”)
  - Delete (owner)
  - List (“r” for a directory)
  - Search (“x” for a directory)