Universidad del Rosario

UR-OS Memory Management: Contiguous Placement, Segmentation, and Paging

Technical Report

Authors: Simon Velez | Thomas Chisica
Course: Operating Systems | Instructor: Pedro Wightman
Date: October 10, 2025

Abstract

This work presents the implementation and evaluation of memory management strategies in the educational Java simulator UR-OS. For contiguous memory allocation, we integrate and compare two classic placement policies—Best Fit and Worst Fit—and study their impact on external fragmentation. For virtual memory, we complete logical-to-physical translation for Segmentation and Paging, including segment/page derivation, bound checks, and dirty-bit handling on `STORE`. We design a deterministic scenario with controlled arrivals, sizes, and memory traces (LOAD/STORE) tailored to induce fragmentation and expose policy trade-offs. We report manually derived allocation outcomes (base addresses per process) and compare them to simulator traces and system-level metrics (total cycles, CPU utilization, throughput, average turnaround, waiting, and response times). The results confirm consistency between manual reasoning and simulation and illustrate the trade-off between aggressively reusing tight holes (Best Fit) and preserving large contiguous regions (Worst Fit), which affects subsequent admissions and overall responsiveness. Reproducibility instructions and numeric validation examples (including address translations and dirty-bit marking) are provided.

1 Introduction

Memory management governs how processes acquire, translate, and release memory. In UR-OS, we address two goals: (i) implementing and selecting contiguous placement policies (Best Fit and Worst Fit), and (ii) completing address translation for Segmentation and Paging with explicit bound checks and dirty-bit updates on `STORE`. We craft a deterministic workload that induces external fragmentation, contrast manual allocation decisions against simulator behavior, and summarize system-level performance metrics.

2 System Overview

UR-OS comprises a CPU, a memory unit, an OS layer exposing policy toggles, and process models that issue CPU, I/O, and memory instructions. The contiguous placement policy and the addressing mode (contiguous, segmentation, paging) are *configurable in `OS.java`*. The simulator prints memory translation traces and reports performance metrics on completion.

3 Implementation

3.1 Contiguous Placement Policies

Let the free list be a sequence of holes, each $\langle base, size \rangle$. For a request of size R, the allocator selects a hole and either consumes it (exact fit) or splits it (if larger).

Best Fit

Definition: Among all holes that can contain R, pick the one with minimum remainder ($size - R$).
Rationale: Reduce immediate external fragmentation by minimizing waste.
Cost: $O(n)$ scan (unless aided by a min-structure).

Worst Fit

Definition: Among all holes that can contain R, pick the largest hole.
Rationale: Preserve large contiguous regions; avoid creating many tiny unreusable remainders.
Cost: $O(n)$ to find the maximum.

Key filenames (no paths) `BestFitMemorySlotManager.java`, `WorstFitMemorySlotManager.java` `FreeMemorySlotManagerType.java`, `OS.java`.

3.2 Address Translation

Segmentation

Given logical address L, derive segment s and offset d (bit partition is *configurable in* ***`OS.java`***). The segment table provides $(base_s, limit_s)$.

1. Bounds: if $d \geq limit_s$, raise a segment violation.
2. Translation: $physical = base_s + d$.
3. On `STORE`: mark the target as dirty. If the simulator models dirty at segment granularity, set it at the segment descriptor; otherwise, dirty applies at page granularity only.

Filename: `SegmentTable.java`.

Paging

With page size P and logical address L: $p = \lfloor L/P \rfloor$, $o = L \bmod P$.

1. Resolve frame for p; if invalid, handle page fault (write back dirty victim first).
2. Translation: $physical = frame \times P + o$.
3. On `STORE`: set the dirty bit for page p (PTE).

Filenames: `PMM_Paging.java`, `SMM_Paging.java`.

4 Deterministic Simulation Scenario

A deterministic workload induces heterogeneous holes and highlights policy differences.

Workload (illustrative)

Each process has arrival time (t) and size [bytes]:

- P_0 (t=0, 260): CPU, LOAD@40, CPU, STORE@180, CPU
- P_1 (t=4, 120): CPU, LOAD@60, CPU
- P_2 (t=8, 200): CPU, STORE@96, CPU, LOAD@150, CPU

- P_3 (t=12, 140): CPU, IO, CPU, STORE@88, CPU
- P_4 (t=18, 320): CPU, LOAD@220, CPU, STORE@48, CPU
- P_5 (t=30, 110): CPU, LOAD@36, CPU, STORE@72, CPU
- P_6 (t=72, 130): CPU, LOAD@82, CPU, STORE@40, CPU

Memory and page sizes are *configurable in `OS.java`*.

Why this induces fragmentation

Interleaving arrivals and completions creates holes of different sizes. Best-Fit tends to reuse tight holes; Worst-Fit pushes allocations toward the largest tail hole, preserving midsize holes.

5 Manual Allocation Results (Contiguous)

Table 1 shows the manually derived base addresses (bytes). We focus on Best-Fit and Worst-Fit.

Table 1: Manually derived base addresses by policy (contiguous placement).

Process	**Best Fit**	**Worst Fit**
P0	0	0
P1	260	260
P2	380	380
P3	580	580
P4	720	720
P5	260	1040
P6	0	1150

Free-list snapshots (justification)

Assume total memory = 2048 bytes (illustrative; *configurable in `OS.java`*).

Table 2: Free-list before allocating P_5 (size 110).

Hole #	**Base**	**Size**
H1	260	120
H2	1040	1008

Best-Fit $\Rightarrow$ H1 (remainder 10) $\Rightarrow P_5 \rightarrow 260$. Worst-Fit $\Rightarrow$ H2 $\Rightarrow P_5 \rightarrow 1040$.
Best-Fit $\Rightarrow$ H1 (remainder 130) $\Rightarrow P_6 \rightarrow 0$. Worst-Fit $\Rightarrow$ H2 $\Rightarrow P_6 \rightarrow 1150$.

Table 3: Free-list before allocating P_6 (size 130).

Hole #	Base	Size
H1	0	260
H2	1150	898

6 Simulation Results

Using contiguous memory with the current selector, the simulator emits the following metrics (example run):

- Total execution cycles = 99
- CPU utilization = 0.9798
- Throughput = 0.0707 jobs/cycle
- Average turnaround = 47.57 cycles
- Average waiting = 29.29 cycles
- Average response time = 9.14 cycles

These values depend on the exact scenario, memory size, and scheduler configuration (*configurable in `OS.java`*).

Log evidence (verbatim excerpts)

Performance indicators

```
******Performance Indicators******
Total execution cycles: 99
CPU Utilization: 0.9797979797979798
Throughput: 0.0707070707070707
Average Turnaround Time: 47.57142857142857
Average Waiting Time: 29.285714285714285
Average Response Time: 9.142857142857142
```

Memory operations (LOAD/STORE)

```
Process 0 ... LOAD,40,0
The obtained data is: 0
Process 0 ... STORE,180,21
The data 21 is stored in: 180
Process 2 ... STORE,96,11
The data 11 is stored in: 476
Process 5 ... STORE,72,5
The data 5 is stored in: 332
Process 6 ... STORE,40,9
The data 9 is stored in: 40
```

Free-list snapshots (from the simulator)

```
Free Memory Slots (3):
Base: 0 Size: 260
Base: 370 Size: 210
Base: 1040 Size: 1047536

Free Memory Slots (3):
Base: 0 Size: 260
Base: 370 Size: 350
Base: 1040 Size: 1047536
```

Policy comparison (metrics)

Both policies were executed under the same deterministic scenario (§ Deterministic Simulation Scenario). The table summarizes the performance indicators for each policy.

Table 4: Simulation metrics comparison: Best Fit vs Worst Fit

Policy	Cycles	CPU Util.	Throughput	Avg. TAT	Avg. Wait	Avg. Resp.
Best Fit	99	0.9798	0.0707	47.57	29.29	9.14
Worst Fit	99	0.9798	0.0707	47.57	29.29	9.14

Note. Under this particular workload and scheduler configuration (*configurable in `OS.java`*), both policies yield identical aggregate metrics, even though their placement decisions differ at the hole level (see Table 1 and free-list snapshots). Different workloads or time-of-arrival patterns may surface measurable differences.

Worst Fit excerpt (verbatim)

These lines illustrate Worst Fit placing P_5 at a high tail hole and P_6 at another large hole, resulting in larger physical addresses for their stores:

```
Process 5 ... STORE,72,5
The data 5 is stored in: 1112
Process 6 ... STORE,40,9
The data 9 is stored in: 1190

Free Memory Slots (2):
Base: 0 Size: 580
Base: 1150 Size: 1047426
```

Comparison

- **Manual vs. simulation:** The run above is a reference execution; switching the selector in `OS.java` to `BEST_FIT` or `WORST_FIT` reproduces the expected placement differences in Table 1.
- **Fragmentation:** Best-Fit reuses tight holes earlier; Worst-Fit preserves large contiguous spans, often pushing new allocations to the tail.

7 Validation and Reproducibility

Numeric translation checks

Segmentation (illustrative). Segment $s = 2$ with $base_2 = 720$, $limit_2 = 320$. Logical offset $d = 180$: $d < 320 \Rightarrow physical = 720 + 180 = 900$. On `STORE`, mark the target dirty (if modeled at segment granularity; otherwise, dirty applies at page granularity only).

Paging (illustrative). Page size $P = 64$. Logical $L = 260 \Rightarrow p = \lfloor 260/64 \rfloor = 4$, offset $o = 260 \bmod 64 = 4$. If $frame[4] = 5$, $physical = 5 \cdot 64 + 4 = 324$. On `STORE`, set $PTE[4].dirty = 1$.

How to reproduce

1. **Build & run:**

```
ant -f build.xml run
```

2. **Switch contiguous policy:** in `OS.java`, set the selector of type `FreeMemorySlotManagerType` to the desired policy (`BEST_FIT` or `WORST_FIT`).

3. **Addressing mode & sizes:** page size and related flags are *configurable in `OS.java`*. Inspect `SegmentTable.java`, `PMM_Paging.java`, `SMM_Paging.java` for bound checks, translation, and dirty-bit updates.

8 Conclusions and Future Work

We implemented contiguous placement (Best Fit, Worst Fit) and completed logical-to-physical translation for Segmentation and Paging in UR-OS. The deterministic scenario exposes expected policy differences: Best-Fit reuses tight holes; Worst-Fit preserves large regions and shifts allocations to the tail. Manual allocation tables align with simulator behavior, and aggregate metrics are consistent with the observed hole maps. Future work includes explicit fragmentation counters (external/internal), sensitivity studies over memory sizes and arrival patterns, and adaptive policies driven by hole histograms.

Code References (filenames only)

- `BestFitMemorySlotManager.java`
- `WorstFitMemorySlotManager.java`
- `FreeMemorySlotManagerType.java`
- `OS.java`
- `SegmentTable.java`
- `PMM_Paging.java`
- `SMM_Paging.java`