

Review

- System calls for project 2
 - Signal system calls
 - Semaphore system calls
 - Shared memory system calls
- Memory management (Cont.)
 - Multiprogramming with variable partition
 - Swapping
 - Memory compaction
 - Keep track of memory usage
 - * Bit maps
 - * Linked lists

Review

Memory management (Cont.)

- **Keep track of memory usage**
 - Bit maps
 - Linked lists
- **Allocation algorithms**
 - First fit
 - Next fit
 - Best fit
 - Worst fit
 - Quick fit
 - Buddy system

- Analysis of swapping systems

- Fifty percent rule:

At steady state, if n processes then $\frac{n}{2}$ holes.

- Unused memory rule:

$$f = \frac{k}{k + 2}$$

where,

$$k = \frac{\text{Hole size}}{\text{Process size}}$$

- Virtual memory

- Provide a large logical address space
- Address translation: MMU
- Memory access speed
- Paging system
 - * Page size vs. page table size
- Segmentation system

**Why virtual memory can be successful:
because of locality of reference.**

The memory reference tends to be local. Access the nearby locations, not crossover the entire memory space.

Page replacement algorithms

- **Random algorithm**
Pick up any page at random.

- **Optimal algorithm**
Select for replacement that page for which the time to the next reference is the longest.
 - Results in the fewest number of page faults
 - Impossible to implement
 - Good for comparison

- **Not-recently-used algorithm (NRU)**
 - Based on locality of reference. Use the past to predict the future.
 - Hardware support:

Two bits associated with each page:

R: reference bit. Set by hardware on any memory read/write to the page.

M: modified bit. Set by hardware when a page is written.

O.S. can reset these two bits in software.

– **Algorithm:**

- (1) When a process is started, R and M bits for all its pages are set to 0.
- (2) On each clock tick interrupt, clear the R bit.
- (3) When a page fault occurs, choose a page from the lowest numbered nonempty class:

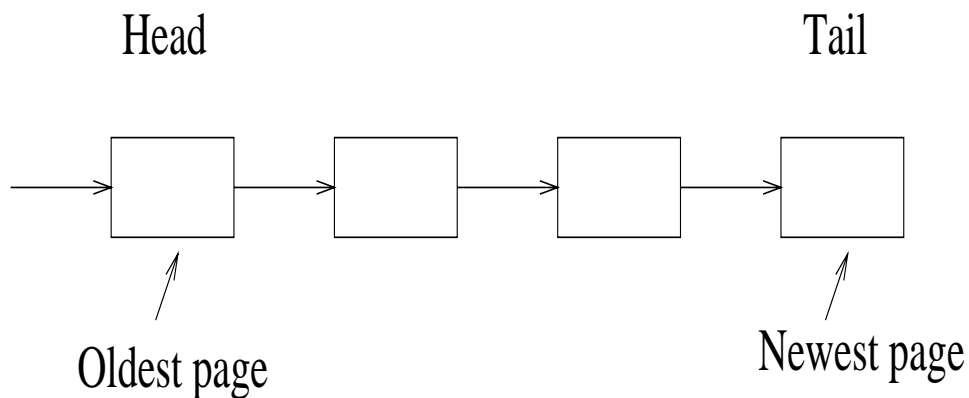
Class	R bit	M bit	
0	0	0	not referenced, not modified
1	0	1	not referenced, modified
2	1	0	referenced, not modified
3	1	1	referenced, modified

- Easy to implement. Performance is ok.
- No history of how long the page is used.

- **First-in-first-out (FIFO)**

Replace the page that has been in memory for the longest time.

- O.S. maintains a list of all pages currently in memory:



- When page fault, remove the head page
- **Problem:** may remove a heavily used page
- **Solution:**
Inspect R and M bits of all pages. Remove the oldest page with $R = 0$.

– **Two variants of FIFO**

* **Second chance**

Look at the oldest page. if $R = 0$, replace it. Otherwise, clear the R bit and put the page at the end of the list.

* **Clock**

Keep all the pages on a circular list.

– **An interesting thing in FIFO:**

More page frames may lead to more page faults.

- **Least-recently-used (LRU)**
Replace the page in memory that has not been referenced for the longest time.
 - Performance close to optimal algorithm
 - Implementation is not cheap. Either requires special hardware or approximate software simulation.
 - Hardware implementation I (counter)
 - * A 64-bit counter, C
 - * Automatically incremented after each instruction
 - * After each memory reference, the current value of C is stored in page table for the page just referenced
 - * Lowest number in the page table is the least recently used

- **Hardware implementation II (matrix)**
 - * For n page frames, use an $n \times n$ bit matrix, initially all 0
 - * After page frame k is referenced, the hardware first sets row k to 1 and then set column k to 0
 - * The row whose binary value is lowest is the least recently used

- **Software approximation I (not-frequently-used, NFU)**
 - * Associate with each page a software counter, initially 0
 - * At each clock tick interrupt, O.S. adds each page's R bit to the counter associated with that page
 - * When a page fault occurs, replace the page with the lowest counter
 - * **Problem: never forgets anything**

- **Software approximation II (aging algorithm)**

Modify NFU as follows:

- * **Counters are shifted right 1 bit before R bits is added in**
- * **R bit is added to the leftmost**
- * **Remove the lowest counter page**
- * **Different from LRU:**
 - **Cannot distinguish the references within the same clock tick**
 - **Counters have a finite number of bits, say, 8 bits, and can remember only the history of 8 clock ticks.**

- **Design issues for paging systems**
Aiming at good performance.

- **Working set:**
the set of pages a process is currently using
- **Thrashing:**
a program causes page fault every few instructions

How to control thrashing:

- * **Keep the entire working set in memory**
- * **Page fault frequency allocation algorithm (PFF)**
In most page replacement algorithms, we can choose a certain number of page frames for each process to keep a certain page fault rate. Dynamically allocate page frames to each process.

– Optimum page size

* Page size p bytes

Average internal fragmentation: $p/2$ bytes

Smaller page \Rightarrow smaller internal fragmentation

But smaller page \Rightarrow larger page table

* Assume:

n segments (logical units) in memory

Average process size s

Each page entry (in page table) requires e bytes

* Memory wasted:

$$\text{Overhead} = (s/p)e + p/2$$

* Optimum page size

$$p = \sqrt{2se}$$

* Example:

$s = 32\text{K}$ and $e = 8 \Rightarrow p = 724$ bytes.

Take 512 or 1024.

Review

Virtual memory

- **Segmentation systems**
 - Segments are logical units
 - Variable segment size
 - More complex address translation

- **Combine segmentation and paging system**
 - Segment starts at a page boundary
 - Easy to share a file

- **Handle a page fault**
- **Page replacement**
- **Page replacement algorithms**
 - Random algorithm
 - Optimal algorithm
 - * Good for comparison

- Not-recently-used (NRU) algorithm
 - * Need two hardware bits (R and M) per page
 - * R is set at every reference and cleared after every clock tick
 - * M is set at every write
 - * Choose the smallest RM page to replace

- First-in-first-out (FIFO) algorithm
 - * Replace the oldest page

- Second chance algorithm
 - * “FIFO” + “R = 0”

- Clock algorithm
 - * Similar to second chance, but difference implementation

Review

- Page replacement algorithms
 - FIFO
 - Least-recently-used (LRU)
 - * Counter implementation
 - * Matrix implementation
 - * Not-frequently-used (NFU)
 - * Aging algorithm
 - Design issues for paging systems
 - * Working set
 - * Thrashing
 - * Control thrashing
 - Keep the entire working set in memory
 - Page fault frequency allocation algorithm (PFF)
 - * Optimum page size

$$p = \sqrt{2se}$$

Deadlocks

- **Non-sharable resource:** the resource can be used by only one process at a time

- **A process may use a resource in only the following sequence:**
 - **Request:** If the resource cannot be granted, the requesting process must wait until it can acquire the resource.
 - **Use:** The process can operate on the resource.
 - **Release:** The process releases the resource.

- **Deadlock:**

Multiple processes are waiting for the availability of a resource that will not become available because it is being held by another process that is in a similar state.

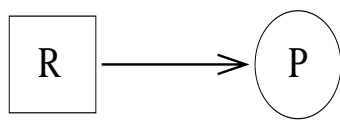
- **Necessary conditions for a deadlock to occur**
 1. **Mutual exclusion:** The resource is non-sharable.
 2. **Hold and wait:** A process that is holding resources can request new resources.
 3. **No preemption:** A resource can be released only by the process holding it.
 4. **Circular wait:** There is a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.
 5. **All four conditions must simultaneously hold.**

- **Resource graph**

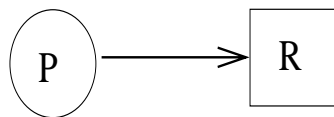
- Directed graph
- Two kinds of nodes:

Processes (circles) and resources (squares)

- **Arcs:**

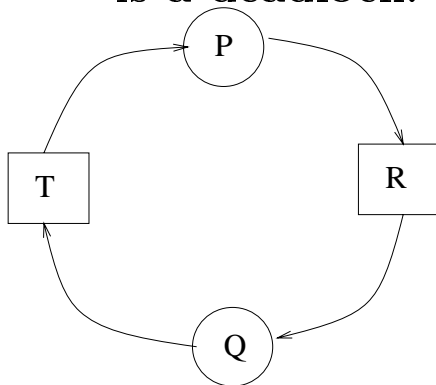


Process P holds resource R



Process P is waiting for resource R

- **A cycle in the directed graph means there is a deadlock:**



Process P holds T and requests R

Process Q holds R and requests T

- Use resource graph to detect deadlocks.

An example:

- * Three processes A, B, and C
 - * Three resources R, S and T
 - * Round robin scheduling
-
- Using resource graph, we can see if a given request/release sequence leads to deadlock:

Carry out the request and release step by step, check if there is any circle after each step.

- **Methods for handling deadlocks**
 1. **Ignore the problem**
 2. **Detection and recovery**
 3. **Prevention**
 4. **Dynamic avoidance**

- **The Ostrich algorithm:**
 - **Stick your head in the sand and pretend that deadlocks never occur.**

 - **Used by most operating systems, including UNIX.**

 - **Tradeoff between convenience and correctness**

- **An example of deadlock in UNIX:**
 - **Process table has 100 slots**
 - **10 processes are running**
 - **Each process needs to fork 12 subprocesses**
 - **After each forks 9 subprocesses, the table is full**
 - **Each original process sits in the endless loop: fork and fail**

- **Detection and recovery**

In a system where a deadlock may occur, the system must provide:

- An algorithm than exams the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

- **Detection:**

Every time a resource is requested or released, check resource graph to see if any cycles exist.

- How to detect cycles in a directed graph?

Depth-first search from each node. See if any repeated node. $O(N)$ algorithm.

– **Recovery:**

- * Abort one process at a time until the deadlock cycle is eliminated.

- * A simpler way (used in large main frame computers):

Do not maintain a resource graph. Only periodically check to see if there are any processes that have been blocked for a certain amount of time, say, 1 hour. Then kill such processes.

- * To recover the killed processes, need to restore any modified files. Keep different versions of the file.

- **Deadlock prevention:**

Use a protocol to ensure that the system will never enter a deadlock state.

Negating one of the four necessary conditions.

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait

1. Mutual exclusion

- Ensure that no resource is assigned exclusively to a single process. Spooling everything.
- Drawback: not all resources can be spooled (such as process table)
- Competition for disk space for spooling itself may lead to deadlock.

2. Hold and wait

- Process requires all its resources before starting
- Problem: processes may not know how many resources needed in advance; not an optimal approach using resources (low utilization)

- **A variant:** a process requesting a resource first temporarily release all the resources it holds. Once the request is granted, it gets all resource back.

3. No preemption

Forcibly take away the resource. Not realistic.

4. Circular wait

- **Solution 1:** A process is entitled only a single resource at any time.
- **Solution 2:** Global numbering all resources:
 - Give a unique number to each resource
 - All requests must be made in numerical order

- An example: two processes and five devices. Number the resources as follows:
 - (a) Card reader
 - (b) Printer
 - (c) Plotter
 - (d) Tape drive
 - (e) Card punch

Assume process A holds i and process B holds j ($i \neq j$).

If $i > j$, A is not allowed to request j .

If $i < j$, B is not allowed to request i .

- Suitable to multiple processes. At any time, there must be a assigned resource with the highest number. This process will not request other assigned resources, only requests higher numbered resource and finishes. Then releases all resources.

- **Deadlock avoidance**

- Analyzing each resource request to see if it can be safely granted.
- Find a general algorithm that can always avoid deadlock by making right choice.
- Banker's algorithm for a single resource:
 - * A small town banker deals with a group of customers with granted credit lines.
 - * The analogy:
 - Customers: processes
 - Units: copies of the resource
 - Banker: O.S.
 - * State of the system:
 - showing the money loaned and the maximum credit available

*** Safe state:**

there exists a sequence of other states that lead to all customers getting loans up to their credit lines.

*** Algorithm:**

For each request, see if granting it leads to a safe state. If it does, the request is granted. Otherwise, it is postponed until later.

Check a safe state:

- (1) See if available resources can satisfy the customer closest to his maximum. If so, these loans are assumed to be repaid.
- (2) Then check the customer now closest to his maximum, and so on.
- (3) If all loans can be eventually paid, the current state is safe.

– Resource trajectories:

A model for two processes and two resources

* An example:

Process A and B

Resources: printer and plotter

A needs printer from I_1 to I_3

A needs plotter from I_2 to I_4

B needs plotter from I_5 to I_7

B needs printer from I_6 to I_8

* Each point in the diagram is a joint state of A & B

* Can only go vertical or horizontal (one CPU)

* Start at point p

Run A to point q

Run B to point r

Run A to point s, granted printer

Run B to point t, request plotter

Can only run A to completion.

– **Banker's algorithm for multiple resources**

* **Processes must state their total resource needs before executing**

* **n processes and m types of resources**

* **Two matrices:**

Current allocation matrix

Request matrix

* **Three vectors:**

Existing resource: $E = (E_1, E_2, \dots, E_m)$

Possessed resource: $P = (P_1, P_2, \dots, P_m)$

Available resource: $A = (A_1, A_2, \dots, A_m)$

$$A = E - P$$

*** Algorithm:**

(1) Look for a row R whose unmet resource needs $\leq A$.

If no such row exists, the system will deadlock.

(2) Assume the process of row R requests all the resources it needs and finishes. Mark that process terminated and add all its resources to vector A .

(3) Repeat Step 1 and 2 until either all processes are marked terminated (the initial state is safe) or a deadlock occurs (the initial state is unsafe).

– **An example.**

Row D $\leq A$, then $A = A + (1101) = (2121)$

Row A $\leq A$, then $A = A + (3011) = (5132)$

Row B $\leq A$, then $A = A + (0100) = (5232)$

Row C $\leq A$, then $A = A + (1110) = (6342)$

Row E $\leq A$, then $A = A + (0000) = (6342) = E$

So, the current state is safe.

Suppose process B requests a printer

Now $A = (1010)$

Row D $\leq A$, then $A = A + (1101) = (2111)$

Row A $\leq A$, then $A = A + (3011) = (5122)$

Row B $\leq A$, then $A = A + (0110) = (5232)$

Row C $\leq A$, then $A = A + (1110) = (6342)$

Row E $\leq A$, then $A = A + (0000) = (6342) = E$

So, the request is still safe.

If E requests the last printer.

$A = (1000)$

No row $\leq A$, will lead to a deadlock.

So E's request should be deferred.

– **Problems:**

(1) **Process don't know the maximum resources they need in advance**

(2) **The number of processes is not fixed**

(3) **Available resources may suddenly break**

● **In summary,**

Prevention: too overly restrictive

Avoidance: required information may not be available

Still no good general solution yet.