Lecture 19

Memory Management System

Page Classes

To each page two bits of reference and dirty bits are assigned. The consents of the two binary digits are (0,0), (0, 1), (1, 0), and (1, 1) which divides the pages into four classes. The candidate for page replacement always belong to the least class and moves to the higher classes until the candidate page can be found.

Reference bit *Dirty bit*

Page Table **Physical Memory**

Memory Management System

LFU (Least Frequently Used)

A counter is used for each page in the memory and it is incremented anytime the page is referenced. (At any page fault, the page with the smallest counter is the candidate for replacement.)

Example: Reference String 7, 0, 1, 0, 1, 2, 2, 1, 0, 0, 7 and 3 frames

Counters for all unique references is set to zero.

P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C
7	1	7	1	7	1	7	1	7	1	2	1	2	2	2	2	2	2	2	2	7	1
		0	1	0	1	0	2	0	2	0	2	0	2	0	2	0	3	0	4	0	4
				1	1	1	1	1	2	1	2	1	1	1	3	1	3	1	3	1	3

Memory Management System

MFU (Most Frequently Used)

A counter is used for each page in the memory and it is incremented anytime that the page is referenced. (At any page fault, the page with the largest counter is the candidate for replacement

Example: Reference String 7, 0, 1, 0, 1, 2, 2, 1, 0, 0, 7 and 3 frames

Counters for all unique references is set to zero.

P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C	P#	C
7	1	7	1	7	1	7	1	7	1	7	1	7	1	7	1	7	1	7	1	7	2
		0	1	0	1	0	2	0	2	2	1	2	2	2	2	2	2	2	2	2	2
				1	1	1	1	1	2	1	2	1	2	1	3	0	1	0	2	0	2

Memory Management System

Ad Hock Algorithm

1- Having a pool of free frames

Frames are used to accommodate the demanded page while the swapping of the victim page is postponed for the time that the paging device is idle or less busy. After paging device written out the victim page, its frame is added to the pool of free frames.

2- Having a pool of free frames and remember which page was in each frame.

The old page could be used directly from the pool without having any I/O operation.

Memory Management System

To refresh your memory:

Degree of multiprogramming (DM) is influenced by the number of frames, X, allocated to each process

$DM = \lfloor$total frames in physical memory/X$\rfloor$)

Problem #1: **What if the process needs more than X frames?**

Problem #2: **How did we arrive at X? (allocation Problem)**

Page Table **Physical Memory**

Memory Management System

Problem #2: *How did we arrive at X? (allocation Problem)*

Answer: *Use of Allocation Algorithms*

Criteria :

1. *Minimum Number of Frames concept*
   a. *ADD A, B (needs maximum of 3 frames for its execution)*
      ADD (A), (B) (needs maximum of 5 frames for its execution)
      .: *X is equal to the maximum number of frames that can support the execution of any one of the instructions in the machine instruction set.*
2. *Global and Local allocation concepts*
   a. *X is limited to the predefined number of frames (Local allocation concept)*
   b. *X* **Page Table** *by steeling frames from other processes (Global allocation concept)* **Physical Memory**

Memory Management System

Allocation Algorithms

1. **Equal Allocation**
   a. Split the M frames among N processes equally.
2. **Proportional Allocation**
   a. There are M frames and N processes and the i-th process is b_i byte long.

Number of frames for process P_i is:

$$x_i = \frac{b_i}{\sum_{j=1}^{N} b_j} * M$$

Example: *P1 and P2 are 150 and 7 byte long and there are 80 frames in RAM*
*X1 = 150/(150 + 7) *80 = 76 frames* *X2 = 7/(150 + 7) *80 = 4 frames*

Page Table Physical Memory

Memory Management System

Threshing

A process is threshing if it spends more time paging than executing.

Page Table Physical Memory

Memory Management System

Before presenting the threshing prevention algorithms, let us introduce the following concepts:

Locality

A number of pages that are actively used together

Example: pages in a segment.

Locality Model

When a process is executed, it goes from one locality to another. This abstract execution of a process is referred to as Locality Model.

Example: Pages of each segment of a process is considered as a locality and the execution of process which goes from one segment to the next represents the locality model.

Number of allocated frames must be enough to accommodate a locality.

Page Table Physical Memory

Memory Management System

Threshing Prevention Algorithms:

1. **Working Set Model**
2. **Page Fault Frequency Algorithm**
3. **Pre-paging**
4. **I/O Interlock**

Page Table Physical Memory

Memory Management System

Final notes:

1. **Page size**
2. **Program structure**

Page Table Physical Memory

Memory Management System

Small page size:

- **Advantages**
  - ✓ **Internal fragmentation is small**
  - ✓ **Data transfer time is small**
  - ✓ **Locality is more precise**
- **Disadvantages**
  - ✓ **Page table is large**
  - ✓ **Latency time is large due to frequent backing store access**
  - ✓ **Number of page faults increases**

There is not a definite answer to a right page size. Some manufacturer have even two different page sizes.

Page Table Physical Memory

Memory Management System

Another fact about page size:

Page size and sector size must support each other.

Page Table

Physical Memory

Memory Management System

Program structure:

```
int [][] arr1=new int [1024][1024];    //page size is 1K and each row
                                       // occupies 1 page
     for(i = 0; i<1024; i++){          //intialize
          for (j = 0; j<1024; j++){    //all array elements
               arr1[i][j] = -1;        //to -1.
          }                            //JAVA is row major
     }
```

```
     for(i = 0; i<1024; i++){
          for (j = 0; j<1024; j++){
               arr1[j][i] = -1;
          }
     }
```

Page Table

Physical Memory

STANDARD INPUT, OUTPUT, AND ERROR IN UNIX

Redirection (> and <)

Redirecting the standard output to a file that will be created on fly.

$ cal 2015 > calender.file

//creates a file containing a calendar

$ wc < abc

//count lines, words, and characters in the file //abc which already exist.

Whenever a UNIX command is run, three standard channels for input and output are opened:

Standard input (or stdin)

Used by program to get the input

Standard output (or stdout)

Used by program to send the output

Standard error (or stderr)

Used by program to display errors

Appending (>>)

Add an output to the end of a file.

$ cal 20015 >>calendar file

Specifically, we are interested to talk about:

- **Redirection**
- **Appending**
- **Grouping**
- **Pipes**
- **Tees**
- **Named pipes**

Grouping (;)

Putting more than one command on the same line

$ Who Return Key

$ ls Return Key

$ cal Return Key

$ who; ls; cal Return Key

Piping (|)

Connects the output from one command to the input of another command

$ who | sort | more

//Who gives currently logged in users' list as output

// list is fed to sort as input and

//the output of sort is fed to command more as input.

Crating and accessing a named pipe

via

a host language

Tees (tee)

Saving the output from a command in a file and at the same time, the output is piped to another command.

$ cal 2015 | tee calfile | more

//the output of the first command serves as input
//to file calfile and the third command.

Named Pipe

An extension to the traditional Pipe concept.

A method for inter process communication

Named pipe (FIFO file)

Named pipe is used to transfer information from one application to another without using an intermediate temporary file. That is, passing information is done internally by Kernel without writing it to the filesystem.

$ mkfifo –m 0777 /tmp/pipe1

$ echo "Today is election day" > /tmp/pipe1

Another process can get the content of the file by using, for example, a cat command

Differences with a traditional pipe:

Duration:

Pipe lasts as long as the execution of the commands involved in piping last.

Named Pipe lasts as long as the system is up even after the processes are terminated. Therefore, it must be deleted when it is no longer needed.

Appearance:

Pipe does not physically exist

Named Pipe must physically be created and named.

How to build a named pipe

1- creating a named pipe, if already does not exist.

2-opening the pipe for writing

3- writing a message into the pipe

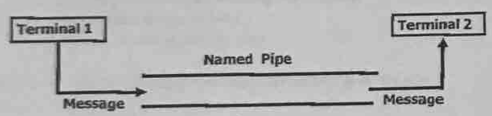
1- a = Mkfifo(const char *pathname, int mode) // build the pipe

b= access(const char *pathname, int mode)

//access checks whether the process would be allowed to read, write, execute, or test for existence of the file.

R_OK | W_OK | X_OK | F_OK

b = access(PF, F_OK) (b = -1 means PF does not exist)



Write a program and run it on terminal 1.

This program builds the named pipe and makes it functional. To do so, a set of constructs needed to complete the following actions:

1- creating a named pipe, if already does not exist.

2-opening the pipe for writing

3- writing a message into the pipe

Write a program and run it on terminal 2.

This program reads from the named pipe and prints it. To do so, a set of constructs needed to complete the following actions:

1-open the pipe for reading

2- read data from the pipe into a buffer

1- creating a named pipe, if already does not exist.

2-opening the pipe for writing

3- writing a message into the pipe

1- a = Mkfifo(const char *pathname, int mode) // build the pipe

b= access(const char *pathname, int mode)

//access checks whether the process would be allowed to read, write, execute, or test for existence of the file.

2- c = open(const char *pathname, int flags) // open the pipe for writing

c = open(PF, O_WRONLY)

// c is the file reference

NAMED PIPE

1- creating a named pipe, if already does not exist.

2- opening the pipe for writing

3- writing a message into the pipe

1- a = mkfifo(const char *pathname, int mode) // build the pipe

a = mkfifo(PF, 0777)

1- creating a named pipe, if already does not exist.

2-opening the pipe for writing

3- writing a message into the pipe

1- a = Mkfifo(const char *pathname, int mode) // build the pipe

b= access(const char *pathname, int mode)

//access checks whether the process would be allowed to read, write, execute, or test for existence of the file.

2- c = open(const char *pathname, int flags) // open the pipe for writing

3- then writing takes place

Your program asks user to enter a message and the message is written in the file with the reference file of c

1- creating a named pipe, if already does not exist.
2-opening the pipe for writing
3- writing a message into the pipe

1- a = Mkfifo(const char *pathname, int mode) // build the pipe

b= access(const char *pathname, int mode)
//access checks whether the process would be allowed to read, write, execute, or test for existence of the file.

2- c = open(const char *pathname, int flags) // open the pipe for writing

3- then writing takes place
res = write(c, buffer, length of the buffer)
buffer ↓ A string typed in by the user

Write a program and run it on terminal 2.
1-Make sure the pipe exist
2- open the pipe for reading
3- read data from the pipe into a buffer

2- c = open(const char *pathname, int flags) // open the pipe for reading
c = open (PF, O_RDONLY)
// c is the file reference

1- creating a named pipe, if already does not exist.
2-opening the pipe for writing
3- writing a message into the pipe

1- a = Mkfifo(const char *pathname, int mode) // build the pipe

b= access(const char *pathname, int mode)
//access checks whether the process would be allowed to read, write, execute, or test for existence of the file.

2- c = open(const char *pathname, int flags) // open the pipe for writing

3- then writing takes place
Writing continues until user ends it.
Upon ending the writing the named pipe needs to be closed
Close (c);

Write a program and run it on terminal 2.
1-open the pipe for reading
2- read data from the pipe into a buffer

1- c= open(const char *pathname, int flags) // open the pipe for reading

2- Reading process takes place
Read from the file with file reference c
and display it on screen

Write a program and run it on terminal 2.
1-Make sure the pipe exist
2- open the pipe for reading
3- read data from the pipe into a buffer

1- *make sure the pipe exist*

Write a program and run it on terminal 2.
1-open the pipe for reading
2- read data from the pipe into a buffer

1- c= open(const char *pathname, int flags) // open the pipe for reading

2- Reading process takes place
res = read(c, buffer, length of the buffer);

Keep reading until the named pipe is closed by writer.
Closing the named pipe
close(c);

Summary

Process #1:

```
a = mkfifo(PF, 0777);          #define PF "/tmp/my_fifo"
b = access(PF,F_OK);           //F_OK (i.e. check for existence of pipe
c = open(PF, O_WRONLY);        //c is integer and the file reference
res = write(c, buffer, length of the buffer); // buffer--a string typed in by the user
```

Process #2:

```
c = open (PF, O_RDONLY) );     //c is integer and the file reference
res = read(c, buffer, length of the buffer); // buffer--a string
```

Closing pipe

```
close(c);
```

New Libraries

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
```