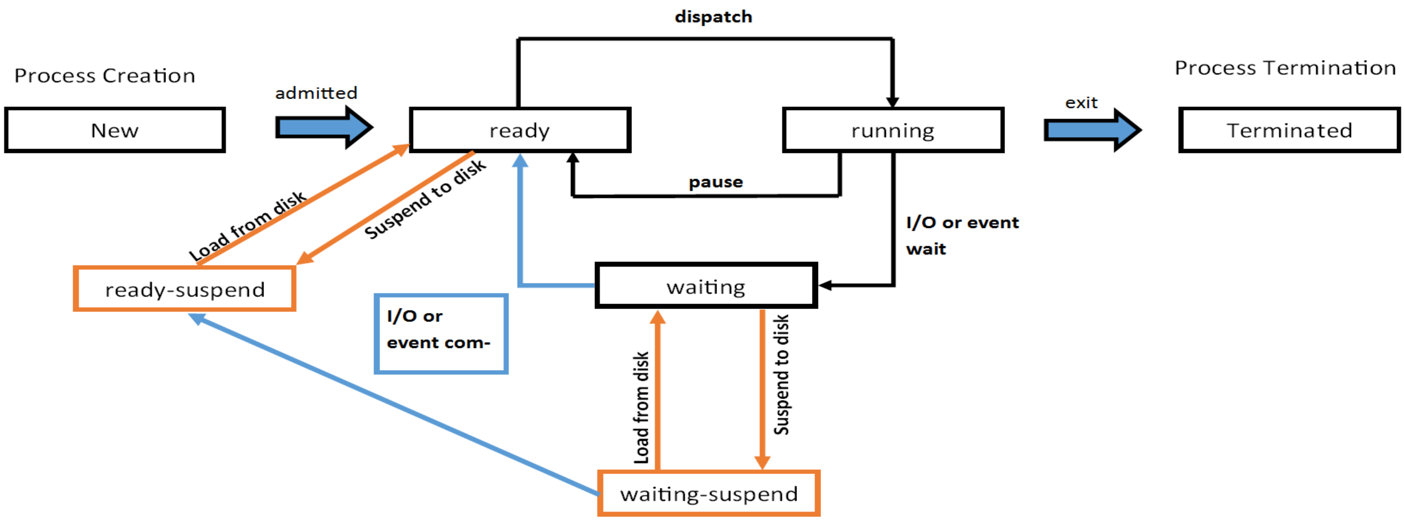
7-state Model

11 April 2017 20:52



Algorithms

10 April 2017 04:17

Pre-emption - process runs in bursts != total execution time

Algorithm (* = Pre, ^ = Pri)	Description	Advantages	Disadvantages
FIFO	First in list	**no one type waits longer** easy to implement	Longer average turnaround and average wait time
SJF	Shortest job in list goes next - after current finishes	Provably optimal - Min Avg waiting time s**hort jobs get good service**	knowing length of next CPU burst is difficult - predicted using estimates of previous or user specified Starve long process
SRJF *	Shortest job in list goes next - can interrupt current	**short jobs get good service** **short response time - good handling of interactive processes** **reduced average turnaround**	knowing length of next CPU burst is difficult - predicted using estimates of previous or user specified Starve long process **Needs preemption**
RR *	Time slices - burst time Preempted and new process go to end of queue If both swap out and new process order = current list - new - swap - end	Fair and simple to implement	hard to appropriate time quantum: Small slice - good response time - scheduler called too often - overhead Large slice - bad response time
PS ^ (*)	Highest priority runs first - FIFO if more than 1 with same priority Can be either preemptive or not	take into account external factors regarding importance of processes higher priority tend to get faster turnaround	might Starve low priority - even more with pre-emption **no pre-emption - slow response time** Solve: aging - too long wait = increase in priority
MQS	More than one queue Assign priority to each queue Each queue has its own algorithm	flexible	**Not good for process changing requirements** (suited for a different queue)
MFQS	Same as MQS - but process can change queue's	Most general	Most Complex

Performance Metrics

11 April 2017 20:52

Avg Waiting time = total waiting time c/ # of processes

Avg Turnaround time = total waiting + execution time / # of processes

Race Condition

13 April 2017 17:43

<u>Race condition:</u>

- Two or more processes are reading/writing some shared data
- Final result depends in who runs precisely when

<u>Critical Region</u> - Part of a program:

- Parts that access shared resource
- Perform computations that can lead to race conditions

- **Ask for permission to enter** CR
- **Signal exit from** CR

is a section of code in a task that cannot overlap with the critical region of any other tasks, as the CR's access shared resources

Locks + Turns

10 April 2017 06:48

Set Lock before entering CR
Clear lock after exiting CR

Pseudo Code

lock(L)
CR;
unlock(L)

Procedure - <u>lock</u> (var L:Lock)

```
Begin
    Repeat
        Do_nothing
    Until (L =0);

    L=1;
End
```

CR;

Possibility of being interrupted
- Solve with Atomic Instruction
- Until(TSL(L) = TRUE)
    - However CPU spends too much time waiting

Procedure - <u>unlock</u> (var L:Lock)

```
Begin
    L = 0;
End
```

<u>Turn Variable</u>
- Only enter CR once Turn set to assigned value
- Then change turn variable after CR to other process value
- If other process not running, can't re-enter CR of current process

Process A

```
While (turn <>0) do
    Being
        Do_nothing;
    End

CR;
Turn = 1;
```

Process B

```
While (turn <>1) do
    Begin
        Do_nothing;
    End

CR;
Turn = 0
```

Peterson's Solution

11 April 2017 20:53

- Uses a Turn variable of type Char
- ALSO with a bool of 'interest'

- Before Entering CR:
  - Interest = TRUE
  - Turn = Other Process
- After Exiting CR
  - Interest = FALSE

- A cannot enter CR if B is interested and Turn = B

Process A

```
Interested_A = TRUE:
Turn = 'B'
While (interested_B = TRUE
        AND Turn = 'B')
                Do_nothing;
CR;
Interested_A = FALSE;
```

Process B

```
Interested_B = TRUE;
Turn = 'A'
While(Interested_A = TRUE
        AND Turn = 'A')
                Do_nothing;
CR;
Interested_B = FALSE;
```

Semaphores

11 April 2017 20:53

Global Variable
Three access functions:

- Initialise
- Signal
- Wait

Blocked Process placed in a Queue

Queue related to the semaphore the processes are waiting for

Init(s,X) - Initialise

- s - name of the semaphore
- X - integer Value

Wait(s)

- Wait for other process to signal 's' is free to use

Signal(s)

- Signal to other process that 's' is free to use

Strong Semaphore - Queue has FIFO process release

Weak Semaphore - Queue has no specified release algorithm

Advantages:

- Avoid long wait
- Easy to synchronise more than 2 processes
- Provided by most modern OS
- Semaphore functions must be uninterruptable

Pseudo Code:

Wait(S); -atomic operation
CR;
Signal(S); -atomic operation

X - Counter

- If X => 0
  - How many processes can enter CR
    - So do wait(s) without being put in the queue
- If X < 0
  - |X| tells how many processes are currently in the queue

Producer-Consumer

14 April 2017 18:30

Uses 3 Semaphores

Init(item, 0) - initially 0 items waiting to be read
Inti(space, n) - initially n spaces available to write to
Init(mutex, 1) - only 1 procedure gets access at a time

Producer

While(TRUE)do
    Produce item

    Wait(space);
    Wait(mutex);
    Write_item;
    Signal(mutex);
    Signal(item);

Consumer

While(TRUE)do
    Wait(item);
    Wait(mutex);
    Get_item;
    Signal(mutex);
    Signal(space);

    Consume_item;

Readers vs Writers

11 April 2017 20:53

Like Producer-Consumer - but now multiple producers and consumers

Only one writer can write
Multiple readers - only if no writers

Readers First Solution

2 Semaphores + counter

Int read_count = 0

Init(mutex, 1) - only one reader at a time can adjust read count
Init(wrt, 1) - only one writer - can only write once read count = 0

<u>Writer</u>

```
While(TRUE)do
    produce_item;

    wait(wrt);
    write_item;
    signal(wrt);
```

<u>Reader</u>

```
While(TRUE)do

    wait(mutex);
    read_count++;
    if read_count = 1 then   // first reader
        wait(wrt);           // must wait for writer to stop
    signal(mutex);

    get_item;    // line to read

    wait(mutex);
    read_count--;
    if read_count  = 0 then    // last reader
        signal(wrt);
    signal(mutex);

    consume_item;
```

Writer's First Solution

15 April 2017 20:38

REMEMBER BY HEART

readcount = 0, writecount = 0;

Init(rmutex,1); - protect readcount
Init(wmutex,1); - protect writecount

Init(z,1);

Init(wsem,1); - protect WRITE_DATA
Init(rsem,1); - let writer finish before allowing new reader

```
procedure_writer()

begin
while(TRUE)
    begin

        wait(wmutex);
            writecount = writecount + 1;
            if(writecount == 1) wait(rsem);
        signal(wmutex);

        wait(wsem);
        WRITE_DATA;
        signal(wsem);

        wait(wmutex);
            writecount = writecount - 1;
            if(writecount == 0) signal(rsem);
        signal(wmutex);

    end
end
```

```
procedure_reader()

begin
while(TRUE)
    begin

        wait(z);
            wait(rsem);
                wait(rmutex);
                    readcount = readcount + 1;
                    if(readcount == 1) wait(wsem);
                signal(rmutex);
            signal(rsem);
        signal(z);

        READ_DATA;

        wait(rmutex);
            readcount = readcount - 1;
            if(readcount == 0) signal(wsem);
        signal(rmutex);

    end
end
```

Deadlocks

11 April 2017 20:54

<u>Four Conditions:</u>

- mutual exclusion - **only one process can access a resource at a certain time**
- Hold & Wait - process can **hold a resource while waiting for another resource**
- No Pre-emption - process **cannot be forced to give up a resource**
- Circular Wait - closed chain exists - each **process holds at least one resource required by the next process** in the chain
    - can prevent by making all processes access resource in a certain order

<u>System Resource Graph</u>

- Nodes:
    - **Circles for Processes**
    - **Boxes for Resources**
- Edges
    - Process -> Resource
      request
    - Resource -> Process
      allocated
- **Circle in graph**
    - circular weight
    - fix by
        - global numbering for all resources
        - request resource in numerical order

Deadlock Avoidance

11 April 2017 20:54

Safe state - provably avoid a deadlock

Unsafe - ***may lead*** to a deadlock

Process	Max Needs	Current Needs
P0	D	A
P1	E	B
P2	F	C

Free at start equal X - (A+B+C)

Find path where X !< 0

<u>Banker's Algorithm</u>

- For each request for a resource by a process, check whether granting the request will lead to an unsafe state
- if it doesn't, it is granted; otherwise it is postponed until a process releases some of its resources
- To check if a state is safe, algorithm:
  1. checks whether it has enough resources to satisfy some process
  2. that process' resources are presumed released, and added to the available resources
  3. back to step 1, and repeat until we find that all current processes can be satisfied

Output as Grant request or Refuse request

Advantage - avoid deadlocks
Disadvantage - have to know resource requirements beforehand

Address Binding

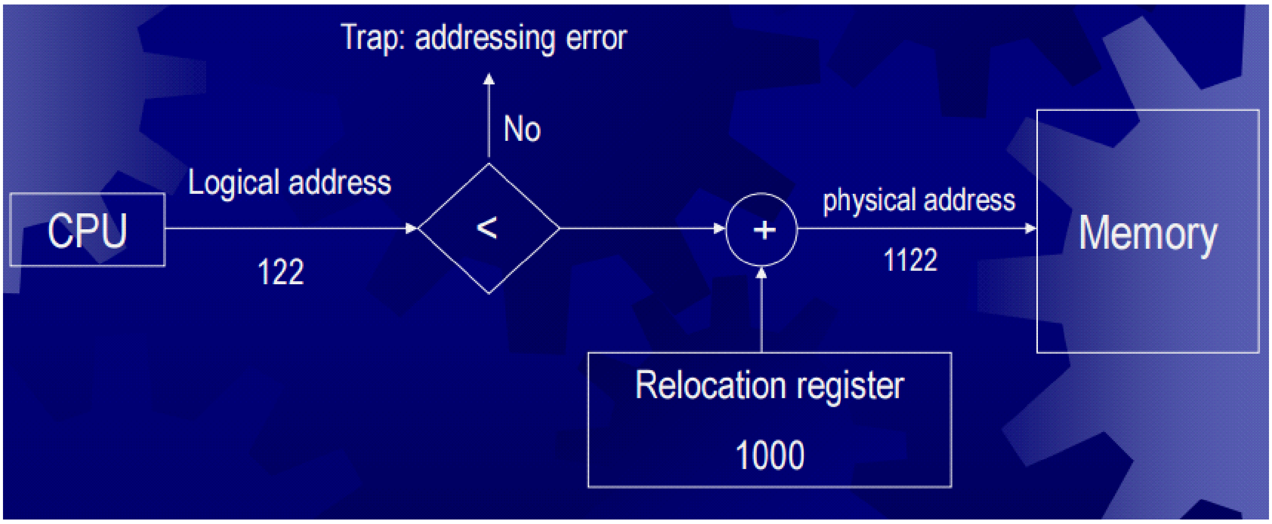
15 April 2017 22:34

Use of a 'relocation register'

Adds value to CPU logical address

Can add a 'limit register' to protects OS and processes from each other

DIAGRAM



Partitioning

10 April 2017 06:48

number of possible programs running at same time = number of partitions

<u>Fixed size partition:</u>

- if process fits, unused space - internal fragmentation
- if process too large cannot run - **BAD**

<u>Dynamic sized partition:</u>

- process might fit, but memory scattered and not sequential
  - EXTERNAL fragmentation

- Three types of allocation
  - First-fit
    - allocate first hole large enough
      - fast
      - can be very inefficient
  - Best-fit
    - allocate smallest hole that is big enough
      - less inefficient than first fit
      - have to search every hole
      - **can produce many tiny fragments**
  - Worst-fit
    - allocate largest hole available
      - remainder of hole might still be usable
      - requires search of all holes

- To fix external fragmentation
  - COMPACTION - reshuffle blocks into large block

Paging

11 April 2017 20:54

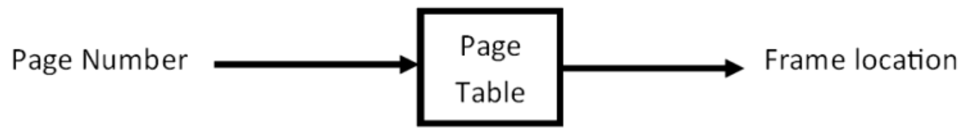
Frame: PHYSICAL memory broken into fixed size blocks

Pages: LOGICAL memory broken into blocks same size as frames

- no external fragmentation
- still possibility for internal fragmentation

CPU addresses - have **page number (p) and page offset (d)**

Page Table: (HW)



Page Table HW cache: **Translation look-aside buffer (TLB)**

- contains most recently used pages
  - should be inside of CPU
- if 'TLB miss' then page table called

Segmentation

11 April 2017 20:54

Divide Logical address into logical segments

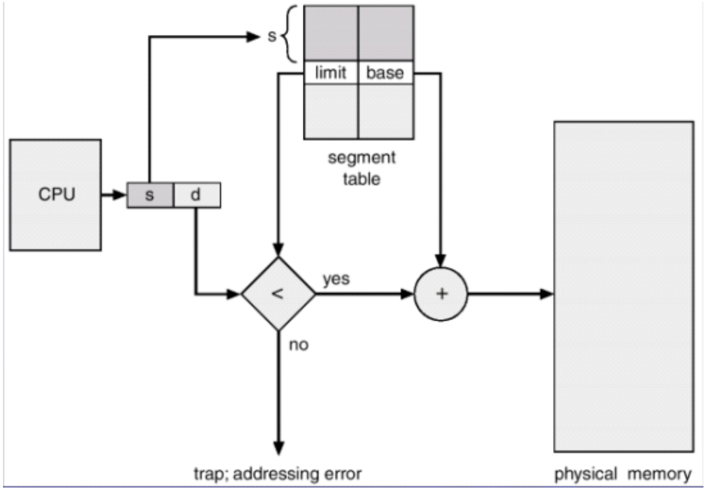
Segment:
- has a name
- has a length - can vary - user specified

Differs from paging by:
- user is aware of segments
- in paging user specifies a logical address - HW creates page number and offset
- in segmentation - user directly specifies segment number and offset

Advantages:
- follows how a user would view memory
- easy to implement protection and sharing

Disadvantages:
- External Fragmentation possible

Segmentation HW



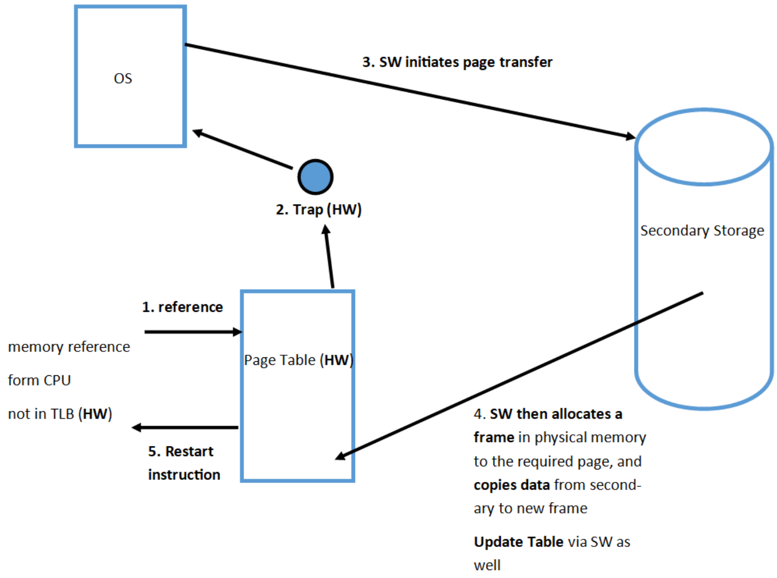
Paging + HW

15 April 2017 23:27

Pager: routine to load pages from secondary storage to main memory - guesswork

Extended page table - 1 bit at end

- Valid - page is in memory
- Invalid - page not in memory
    - page-fault trap
    - control passed to OS (to load page)



<u>Table:</u>

replace page number with frame number

add page offset to frame number

  get physical address

Paging Algorithms

11 April 2017 20:54

Need:

- reference string
- page replacement algorithm
- number of frames available

If more than one process - need to ID references in one super-string

On table - if no page-fault then contents of column not shown

Thrashing - a program spends more time paging than executing

Algorithm	Advantages	Disadvantages
Optimal	lowest page-fault rate	can't implement
FIFO	simple - fast	very bad performance
LRU	simple to implement good at minimising page faults	can give very bad performance

---

Optimal - ideal but impossible (benchmark)

Replace the page that will be:

- not be used again
- for the longest period of time

Look forward from string row

FIFO

replace page that has been in memory for longest

Look backwards on rows of pages to see which is oldest

LRU - Least Recently Used

replace page that was last used furthest back in time

Look backwards from string row

LRU - 2 ways to implement

- Counters
  - global counter that increment every time
  - counters for each page
  - page called - counter = global counter
  - LRU searches page counters
    - picks one with LOWEST value
- Stack - contains all page numbers
  - reference a page - remove from stack and place from bottom
  - bottom of stack is Least Recently Used
  - uses doubly linked list

Frame Allocation

16 April 2017 04:42

1. For n processes
   a. allocate 1/n of the memory - BAD
2. allocate based on process size - PROPORTIONAL
3. allocate based on priority level - PRIORITY
4. **Combo of PROPORTINAL & PRIORITY**

Once Allocated - replacement can be either:

- global
  - replace any frame in memory

- local
  - ONLY replace frames the process was allocated

Normally allocate enough frames to fit programs current *locality*

Locality - set of pages that are often used together by a program

(program has many localities)