Working Set

\[ T \] Pages referenced in this interval = \( WCT, \Delta \)

Set of pages referenced in this interval = \( WCT, \Delta \)

Keep pages of \( WCT, \Delta \) in RAM and replace pages in RAM that are not in \( WCT, \Delta \).

Qns:
1. \( \Delta = \approx 10,000 \) is good
2. How do we determine \( WCT, \Delta \)?
   - Inexpensively.

one bit/page of program

\( i \)th bit is set to 1 (in HW) when page \( i \) is referenced.
Every 2,500 memory references save bit vector in a circular buffer & clean all the bits of the vector.

![Bit Vector Diagram]

Boolean OR of these vectors, we get \( WCT(\Delta) \), \( \Delta = 0.000 \)

Demand Paging in Unix

Conditions:

(i) Memory & CPU architecture allows paging

(ii) Restartable Execution
Page Map Table Entry Contains
(i) Address [physical] of page in RAM
(ii) Protection bits (3 of them)
     \[ \text{RWX} \]
(iii) 5-bit fields to support demand paging
     (a) Valid (=1 if page in RAM)
     (b) Reference (=1 if recently referenced)
     (c) Modify (=1 if page has been modified in RAM)
     (d) Copy on write
     (e) Age bits

OS maintains a supply of free frames; in two lists
\[ \text{FIFO} \quad \text{Hash} \]
# of free frames

\[ \text{max} \]

\[ \text{min} \]

\[ \text{wake up page stealer process.} \]

Page stealer examines each page in RAM of all the processes and adds those pages in RAM but not in their working sets to free list.

---

State Diagram for a Page
Page stealing process examines each page in RAM of each process.

If ret = 1, { clean ref bit and set age = 0 }

Else { age++;
    if age > threshold,
    add to free list }

November 7, 1913
Unix's demand paging is an instance of working set principle's application.

Approx. to working set: [ ]

1. Bit vector
2. Age based [ Unix use ]
Free frames list

Two lists:

1. Free list
2. Hash list

Used to check if a previously "swapped out" page is still available [not allocated to a program]

Example:

Page 15 of process [pid = 2500] has age > n

Add page 15 of pid 2500 to free list and to hash list [valid for page 15]

Process with pid 2500 page faults on page 15

Check [using hash list] if page 15 of pid 2500 is in free list

If so, no need to access disk [remove this from a free list, set valid to 1]
CPU Scheduling

- **NEW** $ightarrow$ Ready
- Ready $ightarrow$ Schedule
- Schedule $ightarrow$ Running
- Running $ightarrow$ Blocked
- Blocked $ightarrow$ Ready
- Terminated

**CPU Scheduler**

- **Long Term**
- **Short Term**

Invoked to pick next ready job to run

"Good" scheduler?

- "Fair"
- Fast [low CPU load]
- Performance
  - Average waiting time of processes?
- Small memory footprint
  - [no page fault in schedule]
Scheduling Algorithms

Non preemptive

preemptive — later

(i) FCFS

Have a queue

Add newly arrived process to ready queue when currently executing process terminates, pick process from head of queue & schedule it.

Bad Average waiting time.
Simple to implement.

Example: CPU time needed

\[
P_1 \quad 40 \\
P_2 \quad 3 \\
P_3 \quad 3 \\
P_4 \quad 4
\]

\[
\text{Schedule: } \{P_1, P_2, P_3, P_4\}
\]

\[
\text{Average waiting time} = \frac{40 + 3 + 3 + 4}{4} = \frac{50}{4}
\]
(ii) Shortest Job Next

Schedule the processes with shortest execution time needed.

Same Example:

Schedule: \((P_2, P_3, P_4, P_5)\)

\[
\text{Average waiting time} = \frac{(10 + 0 + 3 + 6)}{4} = \frac{19}{4} = 4.75
\]

Can show that SJF minimizes average waiting time.

How do we know CPU time needed?

Estimate using past behavior.

\[ t_n = n\text{th CPU burst duration} \]

\[ t_{e_1} \quad t_{e_2} \quad \cdots \quad t_{e_3} \quad t_{e_4} \quad \text{execute want} \]

\[ \tau_n = \text{our estimate of } n\text{th CPU burst duration} \]
before scheduling for the $n$th burst

$$Z_n = \alpha Z_{n-1} + (1-\alpha) Z_{n-1}$$

$$0 \leq \alpha \leq 1$$

$$= \alpha Z_{n-1} + (1-\alpha) \left[ \alpha Z_{n-2} + (1-\alpha) Z_{n-2} \right]$$

$\alpha = 0$ no history of actual CPU usage

$\alpha = 1$ history is very short [no collective history]

$\alpha = 0.5$

(iii) Highest Response Ratio Next

Response Ratio = \frac{\text{Waiting time} + \text{CPU time needed}}{\text{CPU time needed}}

(iv) Priority based

II preemptive scheduling

(i) Shortest Remaining Time Next

not all processes arrive at same time.
SRTN algorithm minimizes average waiting time

(i) Round Robin Scheduler

Maintain a queue

Each process is given a time quantum. [≈ 15-20ms]

When it finishes its time quantum, preempt [add this to tail of queue] & schedule next process.

(iii) Multi-level Cross-Eyed

>1 CPU? [multiprocessor systems]

m = 2 CPUs

Each task is a separate process

t1, e1

t2, e2 ≤ task t2 needs ≤ time to execute.

tn, eE
How long will it take to complete all n tasks?

Can we complete all tasks in \( \frac{E}{2} \) time?

Self-partition:

No preemption? Hard to solve

With preemption? Yes

\[ L_1 \quad | \quad \bar{0} \quad | \quad L_2 \]