## Glossary

| | |
|---|---|
| OS | provides the environment within which programs are executed<br>• Provides *services* for programs<br>• Provides *interfaces* for users<br>• Collection of *components* and their interconnections |
| Parallelism | $>1$ task being performed simultaneously (one process may be waiting on another) |
| Concurrency | multiple tasks making progress at one time (nobody is waiting) |
| Throughput | number of processes completed in a unit of time |
| Turnaround | from time of submission to completion, INCLUDING wait time |
| Wait Time | time spent in waiting/ready queue, NOT including I/O queue / time |
| Response Time | time from submission to time when 'first' usable data/output produced |
| TLB | Translation Lookaside Buffer |
| MMU | Memory Management Unit |
| RMS | Rate Monotonic Scheduling |
| EDF | Earliest Deadline First |
| PTBR | Page Table Base Register (pointer to page table in memory) |
| Context Switch | When a process is forced out of CPU (quantum or pre-emption) |
| Thrashing | When a process spends more time swapping than executing |
| Track | The 'ring' that is 1 bit wide that passes under a stationary HD head |
| Cylinder | The set of tracks among all platers for a single seek position |
| Sector | A segment in a track |
| RAID | Redudant Array of Independent Disks |
| DMA | Direct Memory Access |
| FAT | File Allocation Table |

## Parameter Passing

1. Pass paramater via registers
2. Save parameters in block/table (memory), pass via registers the address of the block
3. Placed onto the stack

## Working Set Model

**Main Idea** Consider pages a process needed in the past, and use as indicator of pages needed in the future
**Locality** A set of pages actively used together
**Spatial Locality** Page references close in time are located near to each other in space
**Temporal Locality** A page is referenced repeatedly in some unit of time

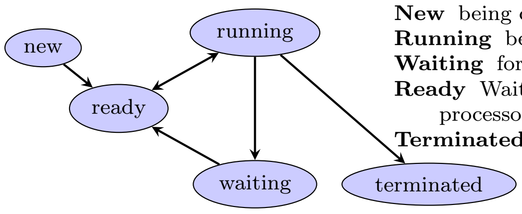
Essentially a *sliding window* of time $t = \Delta$ that moves over the page requests a process makes.

In modern systems, $\Delta$ is in the 5000 - 15000 range.
If the WS size of process $i$ is $WSS_i$...

$$D = \sum_{i=0}^{n} WSS_i$$

If total demand $D$ is greater than count of available frames, thrashing will occur.
The OS monitors WSS for each process, and allocates for each enough frames to its working set size. If there are 'free' frames, initates another process. If too few, a process is suspended.

## Process



**New** being created
**Running** being executed
**Waiting** for some event
**Ready** Waiting to be assigned to a processor
**Terminated** Finished execution

| **Process Control Block** |
|---|
| State |
| Program Counter |
| CPU Registers |
| CPU Scheduling Info |
| Mem Mgmt Info |
| Accounting Info |
| I/O Status |

Two processes are *independent* if the write set of each is disjoint from both the read and write sets of the other.

## Threads

### Many-to-one

All user threads map to a single kernel thread
If a user thread makes a block system call, the entire process (made up of multiple user threads) will block Because only 1 thread can access the kernel at any one time, multiple threads are unable to run concurrently on a multicore computer

### One-to-one

Each user thread is mapped to a unique kernel thread
The creation of a user thread requires (considerable) overhead to create a kernel thread. If a user thread is idle (perhaps waiting on another thread to finish), then the kernel thread with which the user thread 1s associated is needlessly consuming kernel space resources When one thread is blocked (user or kernel thread), the other threads can continue. Concurrency is enabled

### Many-to-many

Many user threads are mapped to a $\leq \#$ kernel threads At the user level, multiple threads are created, and it is up to the OS to schedule/orchestrate/map each of them to a kernel thread

With $m$ threads, $n$ instructions each: $\dfrac{(mn)!}{(n!)^m}$ possible histories

## Critical Section Problem

**Mutual Exclusion** If a process is executing its critical section, no others can be executing theirs
**Progress** If no process is executing its critical section, AND some process wants to enter its, only those processes NOT executing can decide who enters
**Bounded Waiting** There must be a limit on the number of times another process is allowed to enter its critical section after a process has made a request to enter its critical section (i.e., no starvation).

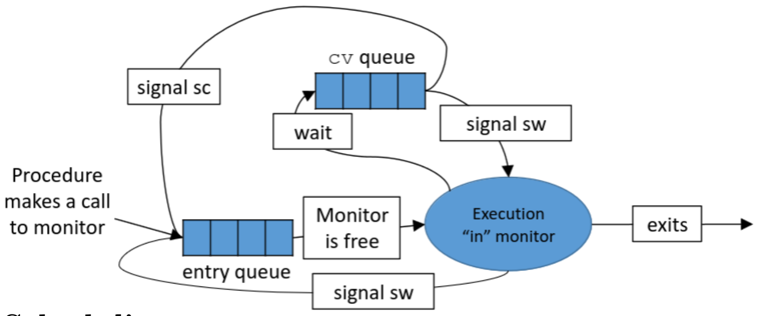
## Semaphores

- Must be careful NOT to omit an inc/dec in code.
- Global, thus must know how entire program works to use them
- Can't infer which waiting process will run next

## Monitors

**SC** The signaler continues, and the signaled executes at some later time
**SW** The signaler waits until some later time and the signaled executes immediately

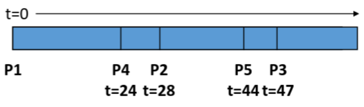
A monitor follows one, but not both.



## Scheduling

**FCFS** First Come First Serve
**SJF** Shortest Job First
**Priority** Highest priority first

| PID | ms | Priority |
|---|---|---|
| P1 | 24 | 1 |
| P2 | 16 | 3 |
| P3 | 2 | 5 |
| P4 | 4 | 2 |
| P5 | 3 | 4 |



Priority

May encounter issues like starvation / aging (hence Round Robin).

## Hard Drive

**FCFS** First Come First Serve
**SSTF** Shortest Seek Time First
**SCAN** Full back and forth, reading during both swings
**C-SCAN** Full back and forth, reading during only one swing (C = Circular)
**LOOK** Back and forth from lowest requested address to highest requested address
**C-LOOK** Back and forth from lowest requested address to highest requested address, reading during only one swing

An I/O request will be for a specific block, in a specific sector, for a certain track, on one of many cylinders.

## File Systems

$$\underbrace{\texttt{-rwx}}_{\text{owner}}\ \underbrace{\texttt{r-x}}_{\text{group}}\ \underbrace{\texttt{r-x}}_{\text{others}}\ \underbrace{\texttt{1}}_{\text{sticky}}\ \underbrace{\texttt{jagodzf}}_{\text{user}}\ \underbrace{\texttt{grp.csci.Faculty}}_{\text{group}}\ \underbrace{\texttt{Dec 1 2024}}_{\text{modified}}\ \underbrace{\texttt{exam.html}}_{\text{name}}$$

## Real-time scheduling

**Periodic** occuring at a constant interval / period (p)
**Process time** time needed to burst (t)
**Deadline** time the process must be completed by (d)

If a requesting process does not 'satisfy' $0 \leq t \leq d \leq p$, the scheduler should reject the process.

Suppose two processes:
**Process 1** p:50ms, t:20ms, d:1/period
**Process 2** p:100ms, t:35ms, d:1/period

Can the CPU process both?

**Utilization (t/p)**

$$P1: 20/50 = 0.4$$
$$P2: 35/100 = 0.35$$
$$0.75$$

Assuming no other process runs, we should be able to service both.

**RMS** Upon entering, a process is assigned a priority inversely proportional to its period. (i.e., the shorter the period, the higher the priority). P2 would be broken up into two chunks (30 / 5) at t=50.
**EDF** Upon entering, a process is assigned a priority inversely proportional to its deadline. (i.e., the sooner the deadline, the higher the priority).

By design, real-time CPU scheduling does not permit a process that has already met its period deadline to start a second time in the same period.

### Storage Hierarchy

Hard Drive
Memory
L2 Cache
L1 Cache
Registers
ALU

### LRU - Second Chance

1. Assign a reference bit to each page
2. When a page is referenced, set the reference bit to 1
3. When evicting, start at the location where last eviction occurred
4. If reference bit is 1, set to 0 and move to next page
5. Repeat until a page with reference bit 0 is found

(referenced, modified)

| (0, 0) | (1, 0) | (0, 1) | (1, 1) |
|---|---|---|---|
| Evict First / Best Choice | | Evict Last / Worst Choice | |

## Memory Allocation

**Best Fit** Fit into a hole such that resulting left-over hole is size minimized (ideally 0)
**First Fit** Fit into first hole (most often reading from bottom addresses to higher) that can accomodate the process
**Worst Fit** Fit into largest hole, resulting in left-over hole whose size is maximized

### Segmentation

Logical Address − > <segment num, offset>

$$\text{Logical Address}- > \begin{cases} \text{Segment Number, offset} \\ S, D \end{cases}$$

S is the index into the table which identifies a row containing limit and base value. In the example table, <3, 200>, <3, 60000> would result in address errors

Segment Table

| limit | base |
|---|---|
| 1400 | 2500 |
| 2600 | 3500 |
| 17000 | 1800 |

### Allocation

If process $p_i$ needs $s_i$ amount of virtual memory...

$$S = \sum_{i=0}^{n} s_i$$

If total amount of available frames is $m$, dole out for process $i$:

$$a_i = (s_i / S) \times m$$

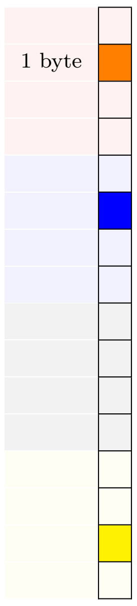
The *minimum* frames per process is defined by the architecture
The *maximum* frames per process is upper bounded by frames available

### Fragmentation

**External** occurs when unused space is non-contiguous
**Internal** refers to the unused space within a frame / page

## Paging

A page size must be a power of 2, usually between 512 bytes and 1 GB.



The $m - n$ highest order bits are needed to address into the page table.

$$\text{Logical Address Space } = 2^m = 2^4 = 16$$
$$\text{Page Size} = 2^n = 2^2 = 4 \text{ bytes}$$

| | p | d |
|---|---|---|
| 🟧 | 00 | 01 |
| 🟦 | 01 | 01 |
| 🟨 | 11 | 10 |

Thus, the logical address gives us some $(p, d)$. We use $p$ to index into the page table to find the correct frame $(f)$, with the offset of $d$ (which is from the $n$ lower bits of the address). Thus our physical address is $(f, d)$.

A single datum in logical address space still requires reserving the entire frame for that datum. Internal fragmentation refers to the unused space within a frame/page.

### Downsides

- Every mapping MUST go through page table
- A page table can be implemented via registers, but this limits their size
- One solution is putting it in main memory, and having the PTBR point to where it is

### Inverted Page Table

A SINGLE page table regardless of how many processes. CPU generates a pid, page number, and offset.

Search the entire page table for $<pid, p>$. The *index* of that entry is the frame number.

Inverted Page Table

| pid | p |
|---|---|

## Virtual Memory

### Page Replacement

**FIFO** First In First Out
**LRU** Least Recently Used
Could be implemented concretely (timestamp), or use a reference bit
**LFU** Least Frequently Used
**RNG** Random

### Belady's Anomaly

The phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns.

### Effective Access Time

$$EAT = (1 - p) \times ma + p(fault)$$

### Copy-on-Write

If two processes rely on the same pages in physical memory, the OS can allocate a single page to both processes. If one process writes to the page, the OS will allocate a new page to the process that wrote to it, and 'copy' the contents of the original page to the new page.

## Miscellaneous

- T/F: If segmentation is used, and the total free (unallocated) main memory is 128MB, then the OS can place into main memory a segment of size 64MB. False - Holes might be tiny
- RMS is neither optimal nor guaranteed to work even if CPU util is < 1
- Given $n$ invocations of `fork`, there will be $2^n - 1$ child processes
- An I/O request would induce a process changing from running -> waiting.
- For a logical address space of 16 bytes, among which there are 8 pages, and assuming the use of a page table:
  - For the corresponding physical memory, each frame is 2 bytes
  - The last byte of frame 1 has a corresponding logical memory address whose offset is 1
  - The 3 highest bit of the logical address are used to index into the page table.

Reasoning:

$$2^4 \Rightarrow m = 4$$
$$2^3 \Rightarrow n = 3 \text{ bits for page table index}$$
$$16 \text{ bytes}/8 \text{ pages} = 2 \text{ bytes per page } \textit{and} \text{ frame}$$
$$m - n = 4 - 3 = 1 \text{ bit for the offset}$$
$$\text{Last byte of first frame } = \underbrace{000}_{\text{page index}}1$$

- As the number of frames allocated to each process decreases, *overall page fault* rate goes up