Process states

HOLD: you don't actually see until ready to execute
becomes a process
held back by the Job Pool & R.M.

READY: doesn't have processor yet but is ready for it
all loaded in mem., set up

RUN: it is taken off the ram state when the OS decides it had enough time
- back and forth READY -> RUN during the life of a process
- the OS and scheduler are resp. for this

WAIT: - most frequent when doing I/O (read/write to files/devices which are slower than the CPU)
- pointless to have a process waiting for a driver to write on, occupy processor
- when wait is over -> READY
- that is why using more threads that CPU is beneficial

SWAP: - the mem. occupied by the process chosen as a victim by the OS is put into SWAP
- an extention of the memory
- everything becomes really slow and unusable

FINISHED: its resources are freed and the process is done

1. create process **
2. get CPU
3. release CPU
4. move on disk *
5. I/O or wait
6. free disk space, allocate mem.
7. release CPU, free mem., destroy process

** alloc mem
* free mem

Deadlocks

Deadlocks: processes and threads are not aware of each other

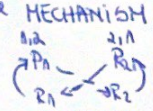
Get out: - stop a process

Detect a deadlock: - there is a cycle
- checking for this is slow

Prevent: always lock them in the same order

Causes: - mutual exclusion
- lock, hold, try to lock again
- non premption (not stealing may lock while I have it)
- circular wait

YOU DON'T LOCK A RESOURCE / YOU LOCK AN ASSOC. SYNCHR. MECHANISM



DEAD LOCK

Process scheduling

- **First come first served**
  - small processes will wait for big ones
- **Shortest job first**
  - burst / estimate job length
  - short jobs stuck in
- **Priorities**
  - higher priority jobs go first no matter size, duration, etc.

Deadline scheduling

A	5	7
B	1	3
C	4	12

=> B, A, C

A	5	7
B	1	3
C	4	12
D	3	9

=> BA, C, D (2 sec. delay)
B, A, D, C (1 sec delay)

Round Robin:
- take each process and give it a quanta of time
- take it out of the processor after that time
- start again

Memory address translation

- use variable names to address locations in memory
- Compiling -> object files (object names = gone)
- Linking -> executable
- Execution -> address -> location in RAM

Allocation methods (where to be loaded)

The compiler hardcodes the exact RAM location (you cannot change that)

A1 Single user/task:
- the OS is at the top
- the process come right after

A2 Multi user/task - fixed partitions:
- the compiler will hardcode the loc. of the partition
- can run multiple processes
- cannot run 2 processes that have to be executed in the same partition
- cannot run a process > the size of the partition
- cannot hardcode physical mem. addresses

~~YOU HAVE TO HARDCODE~~
You have to hardcode into the executable an offset from the beginning of that partition
Computing at runtime the physical address

exec address = physical address

exec address = offset
physical address = partition start + offset

A3 Multi user/task - fixed partitions relocatable
- limited nr. of programs we can run (cannot run more simult. than the nr. of partitions)
- limited by size of the program (cannot run a program larger than the existing partitions)

A4 Multi user/task - variable partitions
- dynamic start of the partition (search in mem)
- no more size limits / process nr. limits
- D: segmentation (a process of large size cannot fit into a contiguous space although there is space)

A5 Virtual allocation - paged
- split physical mem. into pages of fixed bytes

- RAM -> chess board
- no more contiguous mem. space
- an address will not be an offset anymore
- virtual address = virtual page and offset
- lookup is slow

A6 Virtual allocation - segmented
- from the need to access more mem. that could be allowed
- protected memory access
- variable segment
- causes fragmentation
- process = segment start + offset within the segment

A7 Virtual allocation page segmented
- not that used nowadays
- segmentation + paging
- mem. address = segment + page + offset
- 2 lookups

Loading methods (how much do we load)

All at once (beginning)
A: fast exec.
D: slow startup + occupy mem. for unused data

Load first page then when needed
A: no wasted memory
D: need a page = go to the disk
=> slow exec.

Locality principle:
- prefetch pages in advance (the ones next to the one just loaded)

Replacement methods (what to do when full mem)

- which pages we put in SWAP
- this prediction needs to be fast, it happens frequently

FIFO - random guess (don't use)

Not recently used:
- mark every page with 2 bits (read write)
- 1 when read/write and period set to 0
- 0 0 first out
- 0 1 second
- 1 0 third
- 1 1 forth

Least ~~freq~~ recently called
- n x n bits
- when a page is accessed line = 1 col = 0
- once used => full of 1s
- min. sum lin = first out

Malloc / Free calls:
- keep track
- no fragmentation
- keep 2 linked lists (occ. mem. / free mem.)
- no need to search byte by byte -> jump through the list - FRAGMENTATION

Find place in mem:
- **First fit** < fast, doesn't address fragm.
- **Best fit** < slower, really tiny slices
- **Worst fit** < leaves large slices, slower

~~BUDDY: alloc a chunk of 1K a power~~

Buddy: - keep a list of free chunks of powers of 2.
- allocate the largest chunk and distribute the remaining memory in powers of 2

File types: regular, directory, fifo, sockets, hard / symbolic links

Symbolic link: ln -s path name
~~new inode for the same file~~
- can reference to another disk
- delete file => pointing to file that no longer exists
- behaves as the target file/directory
- goes to a file — **several entries in diff. dir pointing to the same inode**

Hard link: ln name link
- new inode for the same file
- has to be root (full rights)
- delete file => a file is deleted only when there are no more hard links linked to it
- goes to data — **multiple inodes pointing to the same data**

Unix file system structure

Block	Content	
0	**Boot block**	know how to run the OS
1	**Super block**	Config., the params of the drive
2 ⋮ n	**inode blocks**	the heads of the lists of blocks for each file
n+1 ⋮	**Data blocks**	

inode: the entry point to the file (a file is a collection of blocks)
a first few positions (128×512)
- direct access (~~...~~ memory block) 1-10
- simple redirect (a pointer to a inode) - 11
- double redirect - 12 ($128^2 \times 512$)
- triple redirect - 13 ($128^3 \times 512$)

Caches:

Direct cache: store the page at loc. page add % cache size
- fast and simple
- may lead to cache trashing (collision)

Associative cache: place each page on a free location found by iterating through the cache
- rather slow

set associative cache: - hash table
- the cache is org. in sets of pages
- the set of a page is computed as page address % cache set
- a free loc. is found by iterating through the cache (set)
- flexible
- avoids collision
- doesn't use all available cache lines

Mounting = associate a storage device to a particular location in the directory tree

Zombie: process that has completed execution but still has an entry in the process table (avoid by using wait
Orphan: still executing but parent dies

Processes VS Threads: - processes can contain 1 or more threads
- quicker to create / to switch between threads — using a / function
- share data easily while avoiding useless duplication (it just creates a stack in the impure part) while for processes it copies the entire impure part

Syncronization mechanism: (creating a thread pthread_create(T[i], NULL, f, (void*)a)
pthread_join(T[i], NULL)

Mutex:
- the threads battle for the mutex and the one to get it gets to execute the instructions inside lock - unlock)
- the rest wait
- when done, unlocks, the battle starts again

Read/Write lock:
- they exclude eachother
- 2 lock methods (read lock, write lock) — pthread_rwlock_rdlock(&rl)
- starvation (reads > writes) — pthread_rwlock_wrlock(&rl)
— " — unlock(&rl)

Semaphore:
- decide how many threads can execute an instruction simultaneously
- reduce congestion (restrict concurrency) — sem_wait, sem_post

Cond variable:
- lets a thread wait until told by another thread to wake up
- signal (or broadcast for more threads) — pthread_cond_wait(&c, &mtx)
- wait (has to have a mutex) — pthread_cond_signal(&c)
- unlocks mutex → wait (sleep) → relocks the mutex (signal) → it comes back
- always used with a mutex

Barrier:
- waits for a given n. of threads to finish executing — pthread_bar_init(&bar, N)
- not reusable (once used it cannot be reused) — pthread_bar_wait(&bar)
- reuse it by destroying → initialize again

Pipes: stored in a buff in memory
- reads waits for any amount of data (until no writing is connected)
- write waits for any space to be made (until no reader is connected)
- inherited — open in the same order

Fifos: files on disk
- open for read (waits until opened for write)
- open for write (waits until opened for read)
- not bidirectional comm.

Regex
. any char.
\ escape
[abc] every char in here
[a-z] - " - interval
[^0-9] every char ∉ the interval
^ beginning of line
$ end of line | \< beginning of word | \> end of word | ( ) group chars in an expression
* previous exp. 0,+ | + previous exp. 1,+ | ? previous exp 0,1 time
{m,n} previous exp at least m, at most n
| logical or

grep -E "..." n ls -l **-n -v -last -rot -uniq -c** count
sed -E "s/ / gi" - duplicate /\1\1/gi - swap /\2/\1/gi
awk -F '...' '$1.....' '$1<100{print $3}' begin{} for
NR - current line
NF - the nr of field on the cur. line
$0 entire input
$1, $2, ...