CG2271 Cheatsheet (Lect 5–9)

Lect 5: Inter-Process Communication (IPC)

Each process has independent memory; explicit IPC needed. Two main types: **Shared Memory** and **Message Passing**. Unix-specific: **Pipe** and **Signal**.

	Shared Mem	Msg Passing
OS overhead	Low (setup only)	High (every op)
Sync needed?	Yes (manual)	Built-in
Scope	Related procs	Any procs
Notes	Fastest after setup	Portable

Shared Memory

POSIX steps: `shmget()` (create) → `shmat()` (attach) → read/write via pointer → `shmdt()` (detach) → `shmctl(IPC_RMID)` (destroy). OS involved only at create/attach; subsequent ops bypass kernel.

Message Passing

Direct: `Send(P2,Msg)` / `Receive(P1,Msg)` — parties must name each other. **Indirect (mailbox):** `Send(MB,Msg)` / `Receive(MB,Msg)` — shared among many. **Blocking** = synchronous; **Non-blocking** = async (null if no message).

Unix Pipes

Unidirectional FIFO byte buffer. `pipe(fd[])` → `fd[0]` read, `fd[1]` write. Writer **blocks** when full; reader **blocks** when empty. After `fork`: parent closes `fd[0]`, child closes `fd[1]`. `dup/dup2` redirect stdin/stdout. Named pipes (FIFOs) allow unrelated processes.

Unix Signals

Async notification. Register: `signal(SIGXXX, handler)`. Common: `SIGSEGV`, `SIGINT`, `SIGTERM`, `SIGKILL` (uncatchable), `SIGSTOP`, `SIGFPE`.

Lect 6: Synchronization

Race Conditions & Critical Section

Concurrent access to shared mutable resource → non-deterministic output. **Critical Section (CS):** `Enter CS` → work → `Exit CS`.

Property	Requirement
Mutual Exclusion	Only one in CS at a time
Progress	Waiting process eventually enters
Bounded Wait	Upper bound on others entering first
Independence	Non-CS processes don't block others

Deadlock: all blocked. **Livelock:** changing state, no progress. **Starvation:** some blocked forever.

Test-and-Set

Atomic: loads value, stores 1. `while(TestAndSet(Lock)==1);` to enter; `*Lock=0` to exit. Satisfies mutual exclusion & progress. **Not** bounded wait (starvation possible). Busy-waits.

Semaphore

Integer S + sleeping-process list. `Wait(S)`: if $S \leq 0$ block, decrement. `Signal(S)`: increment, wake one. Binary ($S \in \{0,1\}$) = mutex; counting = resource tracker. Incorrect use → deadlock.

Producer–Consumer (buffer size K)

```
mutex=1, notFull=K, notEmpty=0
// Producer              // Consumer
wait(notFull);           wait(notEmpty);
wait(mutex);             wait(mutex);
  add item;                take item;
signal(mutex);           signal(mutex);
signal(notEmpty);        signal(notFull);
```

Readers–Writers

```
roomEmpty=1, mutex=1, nReader=0
// Writer                // Reader
wait(roomEmpty);         wait(mutex); nReader++;
  write;                   if(nReader==1) wait(roomEmpty);
signal(roomEmpty);       signal(mutex);
                         // read
                         wait(mutex); nReader--;
                         if(nReader==0) signal(roomEmpty);
                         signal(mutex);
```

Issue: writers may starve.

POSIX APIs

Semaphore: `sem_init`, `sem_wait`(=Wait), `sem_post`(=Signal), `sem_destroy`. **Mutex:** `pthread_mutex_lock/unlock`. **Cond var:** `pthread_cond_wait/signal/broadcast`.

Lect 7–8: Memory Management

Memory Layout

Text (instructions) · **Data** (globals) · **Heap** (grows ↑) · **Stack** (grows ↓).
Base+Limit: PA = Base + LA; check LA < Limit.

Contiguous Allocation

Fixed partitions: internal fragmentation, simple. **Dynamic partitions:** external fragmentation, needs compaction. **Algorithms:** First-Fit (first hole ≥ N), Best-Fit (smallest hole ≥ N), Worst-Fit (largest hole).

Buddy System

Free lists $A[0..K]$; $A[J]$ = blocks of 2^J. **Alloc:** find smallest $2^S \geq N$; split if needed. **Free:** merge with buddy if free; $\text{buddy}(A,S) = A \oplus 2^S$.

Paging

Fixed-size frames/pages. Logical address: p (page num) | d (offset, n bits). PA = $\text{PT}[p] \times 2^n + d$. **TLB:** hardware cache; flush on context switch.

$$\text{EAT} = h(t_{\text{TLB}} + t_m) + (1-h)(t_{\text{TLB}} + 2t_m)$$

PTE bits: readable (r), writable (w), executable (x), valid (v).
COW: parent+child share pages after `fork`; copy on first write.
Frag: no external, internal ≤ $2^n - 1$ bytes (last page).

Segmentation

Logical: ⟨SegID, Offset⟩. PA = Base[SegID] + Offset; check Offset < Limit[SegID]. Variable-size segments per region.
vs Paging: external frag, no internal frag, programmer-visible, flexible protection.

Segmentation + Paging

Logical: ⟨S, P, D⟩. Use S → segment's page table base; look up P → frame F; PA = F × pagesize + D. No external frag; logical structure preserved.

Scheme	Ext	Int	Notes
Fixed partition	Y	Y	Simple
Dynamic	Y	N	Compaction
Buddy	Some	Y	Fast coalesce
Paging	N	Y	TLB needed
Segmentation	Y	N	Variable
Seg+Paging	N	Y	Best of both

Lect 9: Virtual Memory Management

Virtual Memory: Motivation & Idea

Previous assumptions (process fits entirely in RAM, contiguous) are too restrictive.
Idea: split logical address space into pages; keep some in RAM, rest on secondary storage. Secondary storage >> physical memory. Works because of **locality principles**:

- **Temporal:** recently used address likely reused soon.
- **Spatial:** nearby addresses likely used soon.

If page faults occur most of the time → **Thrashing** (heavy I/O). Locality justifies why thrashing is uncommon in practice.

Page Fault Handling

Steps (hardware → OS):
1. Check page table: memory resident? → access & done.
2. **Page Fault:** trap to OS.
3. OS locates page in secondary storage.
4. OS loads page into a free frame.
5. OS updates page table (set resident bit = T).
6. Retry the instruction.

Demand Paging

Process starts with **no** memory-resident pages. Pages loaded only on page fault.
Pros: fast startup, small footprint. **Cons:** sluggish initially; cascading faults possible (thrashing).

Page Table Structures

Direct Paging: single flat table; 2^P entries. E.g. 32-bit VA, 4KiB page, 2-byte PTE → 2 MiB table per process.

2-Level Paging: split into page directory + smaller page tables. Unneeded regions → no allocation. E.g. same settings, only 3 tables used: 1 KiB directory + 12 KiB tables = 13 KiB total (vs 2 MiB).
Address: `[dir-num | page-num | offset]` → directory lookup → page table lookup → frame.
Inverted Page Table: one global table indexed by **frame number**; entries = ⟨PID, page-num⟩.
Pros: huge space saving (1 table for all processes). **Cons:** slow translation (must search by ⟨PID, page-num⟩); often paired with hash.

Page Replacement Algorithms

When no free frame on page fault, evict a page. **Clean page:** no write-back needed. **Dirty page:** must write back first.

$$T_{\text{access}} = (1-p) \cdot T_{\text{mem}} + p \cdot T_{\text{page\_fault}}$$

Since $T_{\text{page\_fault}} \gg T_{\text{mem}}$, minimise p (page fault rate).

OPT (Optimal)

Replace page not used for longest future time. **Minimum** page faults guaranteed. **Not realizable** (requires future knowledge). Used as benchmark. → **6 faults** on example.

FIFO

Evict oldest-loaded page (queue). Simple, no hardware needed. **Belady's Anomaly:** more frames can → more faults! Reason: ignores temporal locality.

LRU (Least Recently Used)

Replace page unused for longest past time. Exploits temporal locality. No Belady's anomaly. **Hard to implement:**

- **Counter approach:** timestamp each access; replace min timestamp. Cons: search all pages, counter overflow.
- **Stack approach:** move referenced page to top; replace bottom. Cons: arbitrary removal from stack, hard in hardware.

Second-Chance (CLOCK)

Modified FIFO + reference bit per PTE. On eviction: if ref-bit=1, clear bit & give second chance (treat as newly loaded); if ref-bit=0, evict. Degenerates to FIFO if all bits=1. Implemented as circular queue.

Algorithm	Faults	Anomaly	HW needed
OPT	Min	No	Future refs
FIFO	Most	Belady's	None
LRU	Good	No	Substantial
CLOCK	Good	No	Ref bit only

Frame Allocation

Equal: each process gets N/M frames. **Proportional:** process p gets $\left(\frac{s_p}{s_{\text{total}}}\right) \times N$ frames.
Local replacement: victim chosen from faulting process's own frames → stable performance, but may starve the process. **Global replacement:** victim chosen from any frame → self-adjusting, but badly-behaved process can hurt others.
Insufficient frames → Thrashing. Global: cascading thrashing. Local: thrashing contained but hogs I/O.

Working Set Model

Observation: process accesses a relatively stable **working set** of pages in any time window. $\mathbf{W(t, \Delta)}$ = distinct pages referenced in interval [t−Δ, t]. Allocate enough frames for W(t,Δ) to minimise faults.
Δ too small: misses pages in current locality → more faults. **Δ too large:** includes stale pages from old localities → waste.
Example: W(t1,5)={1,2,5,6,7} (5 frames), W(t2,5)={3,4} (2 frames).

Memory Hierarchy (Reference)

Level	Access Time	Size
Registers	0.25–0.5 ns	128 B
Cache	0.5–25 ns	6 MB
Main Memory	80–250 ns	16 GB
Disk	5,000,000 ns	500 GB

Address Translation Quick Reference

Base+Limit: PA = Base + LA; LA < Limit
Paging: $p = \text{LA} \gg n$; $d = \text{LA}\ (2^n - 1)$; $\text{PA} = \text{PT}[p] \times 2^n + d$
Segmentation: PA = ST[S].base + D; D < ST[S].limit
Seg+Paging: $\text{PA} = \text{PT}_S[P] \times 2^n + \text{D}$
Memory access: $T_{\text{access}} = (1-p) \cdot T_{\text{mem}} + p \cdot T_{\text{pf}}$