PCB: process state, pc, cpu registers, cpu scheduling info (priority), memory management, accounting info, I/O info

Context: pc, stack pointer, status registers, general purpose registers (GPREGS)

Context switch: user mode -> time inter. -> CPU save PC, SP, SR then update them for inter. handling -> inter. handler save GPREGS -> save context -> load context -> restore GPREGS->load instruction from inter handler is iret --> restore .. clear interr req, resume exe

Process create: fork(), exec() (not return)

Process destruct: exit()

Zombies (defunct process): when process exits, address space is free and files are closed, but retain the exit state, process ID

Getting to kernel mode: boot time, hardware interrupt, software exception (trap or fault), explicit system call, hardware has table of "interrupt service routines"

System call: user call C library function -> function include numeric system call identifier -> execute special instruction to trap to system mode (interr/trap vector transfer control to a system call handling routine) -> syscall handler figures out which syscall is needed and calls a routine for that operation syscall(syscall_no, arg1, arg2, …)

ptrace() to implement Trace command, library calls can be traced using ltrace cmd

Kernel – system call & system call # & system call table; **user** -> system call # and argu -> run syscall instruction -> **kernel** – invoke syscall handler -> syscall table -> system call number -> invoke function -> return by iret

Copy_from_user(), copy_to_user()

Threads: multiple "threads of execution" can run in a single address space; single control flow

shmget(): System call to allocate a shared memory segment (region) shmat(): map a shared memory segment to a local address

mmap(): another approach for creating shared memory regions

Kernel level thread (lightweight processer): Thread operation implement in the kernel; OS schedules all the threads in the system

User level thread: implement and manage by run time system (user level library)

Synchronization

race condition: two concurrent threads manipulated a shared resource without any synchronization, the outcome depends on the order in which accesses take place.

Mutual exclusion: a set of n threads, a set of resources shared between threads, a segment of code which accesses shared resources, called **critical section** -> only one thread at a time can execute in the critical section &**atomicity**: done/not done no btw

Lock: lock() – acquire lock or block until it can acquire the lock; unlock() releases the lock; if other threads are waiting to acquire the lock, one of them should be able to complete its lock() operation following an unlock().

Atomic instructions – test and set

Spinlock - Thread busy-waits in lock() function until it sees the lock is available.

Conditional variable -> ensure the order

Lost wake up problem: occurs when signal happens before wait due to race conditions -> add a state variable to indicate whether signal was sent.

```
pthread_mutex_lock(mutex); // struct lock * mutex
While (condition not satisfied) {
        pthread_cond_wait(cond, mutex);  // struct lock * cond
// Releases mutex, adds thread to cv's wait queue cv, and sleeps
// Re-acquires mutex before return}
.. // do stuff
pthread_cond_signal(cond); // wake 1 in Q to check condition
// or pthread_cond_broadcast(cond);  –wake all to check
pthread_mutex_unlock(mutex)
```

The lock protects the shared data that is modified and tested when deciding whether a thread needs to wait or signal (or broadcast) to let another thread proceed.

Semaphores-less restrictive, 不一定 mutex

```
Wait(Sem) {
        Sem.count - -;
        If (sem.count < 0)
                Sleep();}
```

```
Signal(sem) {
        Sem.count ++;
        If (sem.count <= 0)
                Wakeup_one();}
```

• Semaphore does not suffer from lost wake-up problem! • Internal count variable tracks if signal was called the past

Sem mutex = Sem_init(1) -> Mutex (binary)

Sem; Sem_init(N) -> Counting Semaphore

Concurrency Bugs

Atomicity violation – lack of mutex inside critical section, violate serializability **Order violation bugs** – incorrect ordering of ops

Deadlock bugs – circular waiting

Resource Deadlocks

1.**mutual exclusion** – only one process may use a resource at a time -> use architectural support to create lock-free data structure

```
Void AtomicAdd(int *val, int a){
    do { int old = *val; }
    while (CAS(val, old, old+a) == 0 ); }
```

2. **Hold and Wait** – a process may allocated resources while awaiting assignment of others -> Get all needed resources at the same time. Alternative use trylock()

```
Ready = false; Lock(L1);  if (trylock(L2) == -1)
{unlock(L1);} else {ready = true;}
```

3. **No pre-emption** – no resource can be forcibly removed from a process

4. **Circular wait** – a closed chain of processes exits, such that each process holds at least one resource needed by the next process in the chain -> assign a linear ordering to resource types and require that a process holding a resource of one type, R, can only request resources that follow R in the ordering or **Lock ordering**

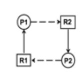
Ostrich Algorithm ignore the problem and hope it doesn't happen often

Scheduling: **FCFS, SJF** (opt with avg waiting time, but starvation) Short term scheduling **(dispatching)** (from ready -> running): fast selection of next process to run, queue manipulation and context switch.

Round Robin (pre-emptive): circular ready queue; quantum (time slice) q;

Multi-Level Queue Scheduling: Have multiple ready queues, one per priority level

Processes are permanently assigned to a Q



Feedback scheduling: Adjust criteria for choosing a particular thread based on history

I/O bound and interactive jobs– higer priority

Priority inversion: Priority inversion happens when a lower priority thread prevents a high priority thread from running -> **priority inheritance**: A protocol that temporarily boosts the priority of a lower-priority task (holding a resource) to match the highest-priority task waiting for that resource.

Proportional-Share Scheduling: Group processes by user or some other means; Ensure that each group receives a proportional share of the CPU

Lottery scheduling: Each group is assigned "tickets" according to its share; Hold a lottery to find next process to run (counter+=ticket)

Unix CPU scheduling: long CPU time -> lower pri; rescheduling occurs every 0.1s; priority is recomputed at the end of every time slice 1s.

Pj(i) = basej + [CPUj(i-1)]/2 + nicej

CPUj(i) = Uj(i)/2 + CPUj(i – 1)/2

Memory management: **Address binding**:

1. **Compile time** (-b) 2. **Load time (static relocation):** compiler – relocatable logical address in object file -> linker – logical absolute address & relocation table -> loader – physical addr when the prog is loaded into memory; 3.**execution time (use this)**

Fixed partitioning of physical memory -> **internal fragmentation & overlay**

# of partitions determines # of active process

Decide at system configuration (boot) time

Dynamic partitioning->ext. fragmentation

->compaction-require process be relocatable

Brk() – sys call – to extend in malloc/free

Address translation: **relocation**: MMU- **base reg** holds the starting physical address of the process's memory; **limit reg** ensures that the process does not exceed its memory bound.

Paging: physical-frame, virtual-page

Page table: mapping of pages to frames PTBR

Virtual address space = $2^{page\ \#\ bit} * pageSize$

Pg = vaddr >> 10 (/1024); **frame** = proc->page_table[pg]; **offset** = vaddr & 0x3FF

paddr = (frame << 10) | offset; Page # -> frame # in decimal -> replace in binary (last 6 bit)

PTE: MRV|prot(RWX)|page frame number

1 KB = 2^{10} byte| 1MB = 2^{10}KB=2^{20}byte

Hierarchical page table: **Two level PT**:

page directory|page table index|pag offset

Page directory base reg

4K page ->12bit offset -> # = pg size/PTE size (4k/1=1k=2^10) -> # of bit (10bit)

Int – 4 byte | 2^{10} = 1024| 1byte = 8 bit