Operating Systems

> Author: Jose 胡冠洲 @ ShanghaiTech

Full-ver. Cheatsheet

See below (page 2-6).

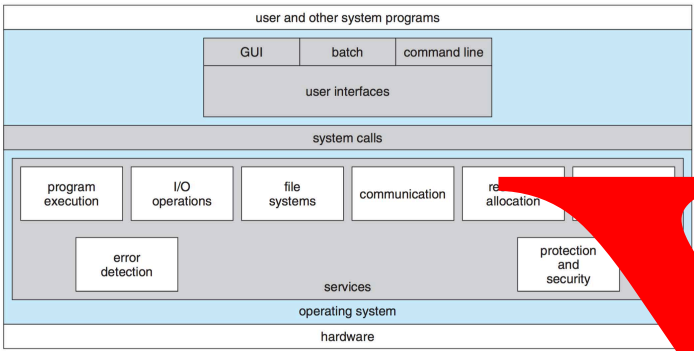
Links

- Stanford CS140 Course Webpage
- PintOS Online Document

This note needs reconstruction.
[ONGOING WORK WITH OSTEP]

What's an OS?

- | *Kernel* + (System programs) | User Apps
  - VM abstraction: APP-SW $\leftrightarrow$ HW resources
  - Protection: SW | $<$ CPU $+$ Memory $>$ | I/O
  - Loader for User programs
- 4 Fundamental OS Concepts
  - *Threads*
  - *Address Space*
  - *Processes*
  - *Dual Mode* Operation
- *bootstrap* program (*Systen Boot*):
  - Stored in firmware
  - Load kernel to run, after *SYSGEN*
- OS Services (provided via `syscall`)

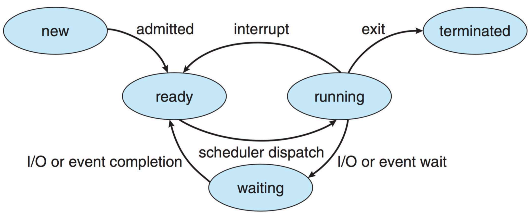


- Design Structures
  - Layered
  - Microkernel: Microkernel + system programs
  - Loadable Modules
  - Hybrid

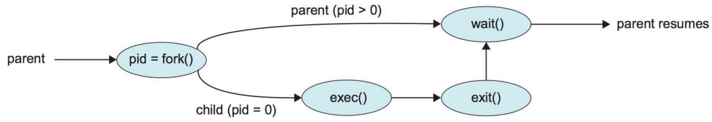
Process

Instance (active) of an executing program (passive).

- $+$ / $-$
  - $+$ Proceed from each other
  - $+$ OS protected from them
  - $-$ "Heavyweight", **different address spaces, page tables & file descriptors**
  - $-$ Can only use kernel synchronization tools
- Process Control Block (**PCB**)
  - Status (new, running, ready, waiting, terminated)
  - Register state (when not ready)
  - Process ID (PID), User, Executable, Priority, ...
  - Execution time, ...
  - Memory space, Translation, ...



- Address space
  - $\neq$ base & bound
  - Owned by a process
  - Virtual, needs translator
- `fork()` : Child process is an **EXACT copy** of parent (separate address space).
  - Return value of `fork()` :
    - $= pid_{child} > 0$, then in parent
    - $= 0$, then in child
    - < 0, then error, in original
  - Waiting
    - *Zombie*: exit whithout parents currently waiting
    - *Orphan*: parent terminated without waiting
  - All processes are children of `init`
    - Have **Copy-on-write** technique




```
/* `syscall` APIs are as follows */
pid = fork();
exit(); // Terminate
pid = wait(&status); // Get status returned from child
abort(pid); // Terminate child process
exec("a_program"); // Flush the program being run currently
```

Threads

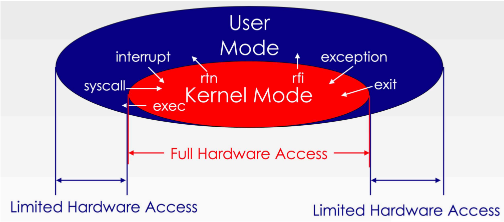
Single unique execution context.

- $+$ / $-$
  - $+$ Efficient, can use user synchronization
  - $+$ "Lite Weight", **Share heap, static data & SAME code**
  - $-$ Lacks protection
- Thread Control Block (**TCB**)
  - State (ready, running, blocked, terminated)
  - CPU register (when not ready)
  - [illegible]cution stack
- [illegible]rnal / External events
  - Internal
    - Blocking on I/O
    - Waiting on other threads
    - Executes `yield()`
  - External
    - Interrupts (I/O, Timer)
- Thread library
  - User threads API
    - POSIX pthread
    - Windows
    - JAVA
  - Kernel threads: supported by kernel
- [illegible] model
  - [illegible] One
  - One $\rightarrow$ One
  - Many $\rightarrow$ Many
  - Two-Level model: M-M + allowing bound
- `pthread` : POSIX thread

```
/* pthread APIs are as follows */
pthread_create(ind, NULL, (void *)worker, &tid);
pthread_join(&tid);
pthread_exit(); // Terminate current thread
pthread_kill(&tid); // Send a kill signal to
/* Mutex... */
/* Conditional Variables... */
```

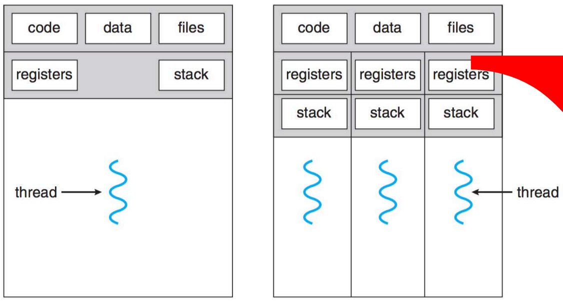
Dual mode & Context Switch

- *Context Switch*: Save & Load PCB / TCB
  - Have pure time overhead QAQ
- 2 Modes
  - **Kernel Mode**: Only “system” can access certain resources
  - **User mode**: User programs isolated from OS (and each other)
- Mode transfer
  - `syscall` : (e.g. `malloc()`)
    - syscall table, buried in run-time library
      - Process control
      - File manipulation
      - Device manipulation
      - Information maintenance
      - Communications
      - Protection
    - Parameter transfering
      - Directly put in register
      - Memory Block and pass address to register
      - Via Stack: User $\nrightarrow$ kernel stack
  - **Interrupts**: HW-invoked context switch (e.g. Timer)
  - Trap / Exception



Multi-xxx

- Definitions
  - Multiprocessing (core): Multiple CPUs (cores)
  - Multiprogramming: Multiple jobs or processes
  - Multithreading: Multiple threads per process
  - Time-sharing (Multitasking): switch frequently

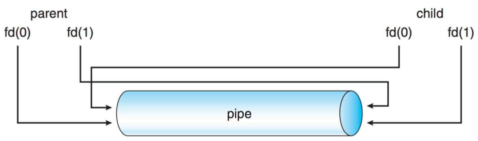


single-threaded process multithreaded process

- **Concurrency**
  - Way: Can *multiplex in time*, virtual CPUs
  - Needs: Scheduler & Context Switch
- **Parallelism**
  - Way: Data / Task parallelism
  - Needs: Multi-processors / Multicore / Hyperthreading ... ing

Interprocess Communication (IPC)

- Shared Memory
  - Unbounded Buffer
  - Bounded Buffer
- Direct Message Passing: Communication Link
  - Name each other explicitly
- Indirect Message Passing (*Mailboxes,ports*), possible solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive
  - Allow the system to select arbitrarily the receiver
  - Sender is notified who the receiver was
- Client-Server Communication
  - Sockets
    - Paired endpoints
    - Identified by IP address
  - Remote Procedure Call (*RPC*)
    - Message Passing between Clients & Server
    - Use Stubs to pass parameters
    - "Exactly Once"
  - Pipes
    - Anonymous (Ordinary)
    - Named (FIFO)



Synchronization

- Requirements
  - *Mutual exclusion*
  - Making *progress*
  - *Bounded waiting*

- HW solutions
  - Atomic operations (i.e. unable Interrupts)
    - Test & Set, Page.210
    - Compare & Swap, Page
- **SW solutions** for
  - `mutex` : Busy waiting (*spinlock*) / *Blocking* lock
  - `cond` : Conditional variable
  - `sema` : Semaphore, Page.216

Specialty QAQ

- `syscall` parameters on thread stack, NOT kernel stack
  1. Kernel pick a free interrupt number (e.g. 2)
  2. fill function into IVT entry #2
  3. User cause interrupt 2
  4. Return from interrupt
- Even on single-processor, Multithreading speed up running
  - Since ove... with computation
- `exec()` only re... s file descriptors
- Dual Mode + ...ual Address Translation = No over-writting ^-^
  - Since ... priviledge instructions allowed
- Proces... n wait / block on:
  - ...uiring a lock (sema, monitor)
  - ...alling sleep
  - ...O call
  - ...t() on child process
- P... fs
  - ... errupts may preempt a thread
  - Id... Simplicity for *scheduling* and *switch*
  - Inter... → Atomic operation
  - "magic ... must be at bottom of `struct thread`
  - User pr... mplemented → ALL using kernel memory, crash all
- Sockets can be used either remotely or locally
- **Interrupts can make locks**
  - Single processor: √
  - **Multiprocessors:** ×
- `fork()` fails, `wait()` will immediately return
- Synchronization constraints
  - XXX must wait if XXX

```
/* Example of bounded-buffer solution */
1. Lock lock;
2. Condition dataready;
   Condition queueready;
3. Queue queue;

AddToQueue(item) {
    lock.Acquire();
    while (queue.isFull()) { /* WHILE LOOP!!! */
        queueready.wait(&lock);
    }
    queue.enqueue(ite );
    dataready.signal();
    lock.Release();
}

RemoveFromQueue() {
    lock.Acquire(); // Get Lock
    while (queue.isEmpty()) {
        dataready.wait(&lock);
    }
    item  = queue.dequeue();
    queueready.signal();
    lock.Release();
    return(item);
```

```
/* Barber Question */
void Barber () {
    while (true) {
        customerReady.P();
        accessWaitRoomSeats.P();
        numberOfFreeWaitRoomSeats += 1;
        accessWaitRoomSeats.V();
        cutHair();
        barberReady.V();
    }
}

void Customer () {
    accessWaitRoomSeats.P();
    if (numberOfFreeWaitRoomSeats > 0) {
        numberOfFreeWRSeats -= 1;
        accessWaitRoomSeats.V();
        customerReady.V();
        barberReady.P();
        getHairCut();
    } else {
        accessWaitRoomSeats.V();
        leaveWithoutHaircut();
    }
}
```

```
/* Synch using *swap* */
void Initialize(int* lock) {
    *lock = 0;
}

void Acquire(int* lock) {
    int l = 1;
    do {
        swap(&l, lock);
    } while (l == 1);
}

void Release(int* lock) {
    *lock = 0;
}
```

6 Synchronization

- **Critical Section** Problem
  - Solution **Requirements**: P194
  - *Software-based* Solutions:
    - **Peterson's**: P195
    - Bakery Algorithm
  - Providing Locks through **Hardware Atomic Instructions**:
    - `TestAndSet()` and `Swap()` : P197
    - Uniprocessor - Disable interrupts
    - Must have cache coherency
  - Providing **Semaphores** for Usage: P200
    - `signal()` and `wait()`
    - *Busy waiting* (spinlock) v.s. *Blocking*
- **Producer-Consumer** Problem (Bounded Buffer)
  - Shared data without synchronization solution
    - Allow at most $n-1$ items in buffer
  - With `counter` *synchronized*: P205
- **Readers & Writers** Problem: P206
- **Dining Philosophers** Problem: P207
- Higher-level Synchronization - **Monitors**: P209
  - *Conditional* cirtical regions, ensure no deadlocks

7 Deadlocks

- **Definition**: P245
  - Deadlock $\Rightarrow$ Starvation, $\not\Leftarrow$
  - Starvation alone might end; Deadlocks cannot
- **Necessary Conditions**: P247
  - Mutual Exclusion
  - Hold $+$ Wait
  - No Preemption
  - Circular Wait
- **System Model** (*Resource-Allocation Graph*): P249
- Methods of Handing Deadlocks: P252
  - **Deadlock Prevention**: P253
  - **Deadlock Avoidance**: P256
    - Banker's Algorithm
  - Ignore $+$ Deadlock **Detection & Recovery**: P262
- Combined Approach
  - Hierarchical ordered resources classes
  - Use different technique for each class

8 Memory Management

- Background
  - **Base & Bound**: P277
  - **Address Binding**, *Logical* v.s. *Physical*: P278
    - Memory Management Unit (*MMU*)
  - Dynamic linking & loading: P280
- Primitive **Swapping**: P282
  - Pending I/O v.s. I/O to kernel space (Double Buffering)
  - Swap time $=$ (Process size $/$ Transfer rate) $\times 2$
- Primitive **Memory Allocation**: P284
  - External *Fragmentations*
- **Paging**: P288

    - Internal *Fragmentations*
    - Transition Look-aside Buffer (*TLB*)
    - Effective Access Time $=$
        - Hit-ratio $\times$ (TLB lookup time $+$ Memory access time) $+$
        - $(1 -$ Hit-ratio $) \times (\cdots +$ Page table access time)
    - *Multilevel* Paging & *Hashed* Paging: P299
    - **Page Sharing** through Paging
        - Copy-on-Write (*CoW*), `vfork()` : P325
- **Segmentation**: P302

9 Virtual Memory

- **Demand Paging**: P319
    - Effective Access Time $=$
        - $(1 -$ Page fault-ratio $) \times$ Memory access time $+$
        - Page fault-ratio $\times$ (Page fault overhead (almost 0) $+$
        - Swap time $+$ Need swap out-ratio $\times$ Swap time)
    - Valid Bit
- **Page Replace** Algorithms, Modify Bit: P328
    - *FIFO*; *Belady's Anomaly*
    - Optimal
    - *LRU*
    - LRU Approximate
        - Reference (Access) Bit
        - *Second Chance* (*Clock*) Alg
    - Counting Based
        - LFU v.s. MFU
    - *Page Buffering*
- **Frame Allocation** Algotithms: P340
    - Equal v.s. Proportional
    - Global v.s. Local
- **Thrashing**: P343
    - Based on *Locality* - *Working Set* & *Page-fault Frequency*
- Other Issues: P357
    - Prepaging
    - Page size
    - TLB Reach
    - Program structure
    - I/O interlock

10-12 Storage Management

- **File Concepts, Operations, Types & Structures**: P373
    - Global open table v.s. Local open table; Offset is local!
- **Access Methods**: P382
    - *Sequential* v.s. *Direct*
- **Directory Concepts & Structures**: P385
    - Single level $\rightarrow$ Two level $\rightarrow$ Tree structure $\rightarrow$ Graph structure
- File **Protection**: P402
    - Access Control
    - *Consistency Semantics*
- **File System Layer Structure**: P411
- File System **Implementation**: P413
    - *Partitions* & *Mounting*
    - Virtual File System (*VFS*)
- **Disk Allocation** Methods: P421
    - Contiguous

- Linked + aFile-Allocation Table (*FAT*)
- *Indexed* (direct v.s. indirect)

- **Moving-head Disk**: P451
  - Transfer Rate
  - Random Access Time = Seek Time + Rotational Latency
    - Average Seek needs $\frac{1}{3}$ of overall Tracks
    - Average Latency needs $\frac{1}{2}$ of a circle
  - Average I/O time =
    - Random Access time +
    - (Amount to transfer / Transfer rate) +
    - Controller overhead
  - Effective Transfer Rate = Amount to transfer / Average I/O time
- **Disk Attachments**: P455
- **Disk Scheduling** Methods: P457
  - FCFS
  - SSTF
  - SCAN & C-SCAN
  - LOOK & C-LOOK
- **RAID** & Extensions: P470
  - Solaris ZFS system

Appendix

- Bakery Algorithm

```
int turn[n];
bool choosing[n];
int j;
while (1) {
    choosing[i] = true;
    turn[i] = 1 + max(turn[0], turn[1], ... turn[n[illegible]
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        if (j != i) {
        // Wait until thread j receives its number:
        while (choosing[j]);
        // Wait until all threads with smaller nu[illegible]rs
        // or with the same number but with hig[illegible] priority
        // finish their work:
        while (turn[j] != 0 && ((turn[j], j[illegible] (turn[i],i)));
        // (a, b) < (c,d) <=> (a < c) || [illegible] c && b < d)
        }
    }
    /* Critical Section. */
    turn[i] = 0;
    /* Remainder Section. */
}
```

- Shared data without synchronization Solution

```
/* Producer. */
while (true) {
    /* Produce an item `next_produced'. */
    while (((in + 1) % BUFFER_SIZE) == out);
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

/* Consumer. */
while (true) {
    while ((in == out);
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* Consume the item `next_consumed'. */
}
```